Liquidsoap 0.3.6 user's guide.

Built using online documentation available at http://savonet.sf.net on Dec, 17 2007.

David Baelde Samuel Mimram Romain Beauxis Vincent Tabard

February 7, 2008

Contents

Chapter 1

Liquidsoap

1.1 Liquidsoap

Liquidsoap is a powerful tool for building complex audio stream generators, typically targetting internet radios. It consists of a simple script language, which has a first-class notion of source (basically a *stream*) and provides elementary source constructors and source compositions from which you can build the streamer you want. This design makes liquidsoap flexible and easily extensible.

We believe that liquidsoap is easy to use. For basic purposes, the scripts simply consists of the definition of a tree of sources. It is good to use liquidsoap even for simple streams which could be produced by other tools, because it is extensible: when you want to make your stream more complex, you are still able to stay in the same framework, and your script will remain maintainable. Of course, this will require at some point a deeper understanding of liquidsoap's concepts and scripting language.

If you're new to liquidsoap, you'd probably like to read about the installation procedure and take the quickstart tour. Then you may also enjoy to learn more about the main concepts on which liquidsoap is built. When you'll master the basics, you'll only need to take a look at the reference (scripting language, API and settings) and get a few ideas from the recipes to be able to design whatever stream you need. Finally, have a look at the telnet tutorial to find out how to interact in various ways with a running liquidsoap.

Liquidsoap is open-source, written in OCaml and is part of the savonet project.

Acknowledgement for the readers of the PDF version. The file you're reading has been automatically generated from savonet's wiki. It can be useful to get directly there, in particular if you need to copy a code snippet: `http://savonet.sf.net/wiki/Liquidsoap`.

Acknowledgement for the Wiki readers. There is a PDF file automatically generated from selected pages of this wiki. It can be useful for printing, and is available in liquidsoap distribution. Also, beware that this site may contain information that is only relevant for development versions of liquidsoap.

1.1.1 Features

Here are a few things you can easily achieve using Liquidsoap:

- Playing from files, playlists, directories or script playlists (plays the file chosen by an external program).
- Transparent remote file access; easy addition of file resolution protocols.
- Scheduling of many sources, depending on time, priorities, quotas, etc.
- Mixing one source on top of the other.

- Queuing of user requests; editable queues.
- Sound processing: compress, compand, normalize, echo, soundtouch, etc.
- Supports arbitrary transitions: you can have fade, cross-fade, jingle insertion, etc.
- Per-track settings of transitions via metadata `liq_fade_in`, `liq_fade_out`, `liq_start_next`, `liq_append` and `liq_prepend`.
- Highly customizable smart cross-fading based on audio intensity analysis.
- Input of other streams (Vorbis, MP3 and AAC over HTTP): useful for switching to a live show.
- Blank detection.
- Definable event handlers on new tracks, new metadata and excessive blank.
- Metadata rewriting and insertion.
- Multiple outputs in the same instance: you can have several quality settings, use several media or even broadcast several contents from the same instance.
- Output to icecast and peercast (mp3/ogg) or a local file (wav/mp3/ogg).
- Output to speakers using libao.
- Output to ALSA speaker, input from ALSA microphone.
- Distributed encoding using RTP (still very experimental)!
- Interactive control of many operators via telnet and UNIX domain socket, or indirectly using perl/python scripts, pyGtk GUI, web/irc interfaces (not released, mail us)...
- Speech and sound synthesis.

If you need something else, it's highly possible that you can have it – at least by programming new sources/operators. Send us a mail, we'll be happy to discuss these questions.

1.1.2 Non-Features

Liquidsoap is a flexible tool for processing audio streams, that's all. We have used it for several internet radio projects, and we know that this flexibility is useful. However, an internet radio usually requires more than just an audio stream, and the other components cannot easily be built from basic primitives as we do in liquidsoap for streams. We don't have any magic solution for these, although we sometimes have some nice tools which could be adapted to various uses.

Liquidsoap itself doesn't have a nice GUI or any graphical programming environment. You'll have to write the script by hand, and the only possible interaction with a running liquidsoap is the telnet server. However, we have modules for various languages (OCaml, Ruby, Python, Perl) providing high-level communication with liquidsoap. And there is a graphical application using the Python module for controlling a running liquidsoap: liguidsoap.

Liquidsoap doesn't do any database or website stuff. It won't index your audio files, it won't allow your users to score songs on the web, etc. However, liquidsoap makes the interfacing with other tools easy, since it can call an external application (reading from the database) to get audio tracks, another one (updating last-played information) to notify that some file has been successfully played. The simplest example of this is bubble, RadioPi also has a more complex system of its own along these lines.

1.2 Installation

Several ways of installing liquidsoap are possible.

The recommended way for newcomers is to use the liquidsoap-full-xxx.tar.gz tarball. This tarball includes all required OCaml bindings and allows you to compile and install liquidsoap in a single configure, make and make install procedure. You will still need the coresponding C libraries and there developement files, though.

For more advanced users, you can choose which features you want. Here are liquidsoap's dependencies (all OCaml libraries are distributed by Savonet, except Camomile):

- ocamlfind (http://www.ocaml-programming.de/programming/findlib.html)
- ocaml-dtools

And also optional dependencies:

- ocaml-ogg
- ocaml-shout
- ocaml-vorbis
- ocaml-shout
- ocaml-mad for mp3 decoding
- libid3tag (http://www.underbit.com/products/mad/) for reading mp3's id3 metadata
- ocaml-mp3id3 for reading mp3's id3 metadata
- camomile (http://camomile.sourceforge.net/) for detecting metadata encodings and re-encoding them to utf8
- ocaml-lame for mp3 encoding
- ocaml-alsa for alsa input/output
- libortp (http://www.linphone.org/) for RTP input/output
- wget (http://www.gnu.org/software/wget/) for downloading remote files (http, https, ftp)
- ufetch (provided by ocaml-fetch) for downloading remote files (smb, http, ftp)
- festival (http://www.cstr.ed.ac.uk/projects/festival/) for speech synthesis (say)

And other that you'll find on the project page, or in liquidsoap-full tarball.

1.2.1 Install from source tarballs

The primary mean of stable distribution is source tarballs. They are available on the download section (http://sourceforge.net/project/showfiles.php?group_id=89802) of the project's page on sourceforge. They all follow the GNU conventions, and are built and installed using the common `./configure`, `make` and `make install`.

1.2.2 Subversion repository (and other distributions)

If you want a cutting-edge version, you can use the subversion repository. To get a copy of it, just run:

```
svn co https://savonet.svn.sourceforge.net/svnroot/savonet/trunk savonet
```

From every sub-project's directory you can build and install the package using `./bootstrap`, `./configure`, `make` and optionally `make install`.

From the toplevel `savonet` directory you can also directly build a vanilla liquidsoap. It's fast and doesn't require you to install the libraries. The steps to follow are simple:

```
# Edit PACKAGES to choose which feature you want
./bootstrap
./configure
make
# To install liquidsoap, you'll usually need to type the following as root
make install
```

1.2.3 Debian

Debian packages are available on the official repository for testing and unstable.

We are working on backported packages for debian stable (etch).

1.2.4 Ubuntu source install

You can find more informations on how to install LiquidSoap from source (Subversion) on the Ubuntu page.

1.2.5 Gentoo

You can find more informations on how to install LiquidSoap on the Gentoo page.

1.2.6 OSX

There have been successful installations on OSX (both Intel and PPC), using Fink and the Godi distribution of OCaml. Claudio reports his two successes on the OSX page.

1.3 Installation on an Ubuntu system

Be careful that this will not work on an Ubuntu system anterior to Edgy Eft (Ubuntu 6.10) because of uncompatible Camomile versions. If you really want to get this working, you can follow the instructions for compiling ocaml in the "Compiling ocaml from source" in InstallationDebian

To install the required dependencies type:

```
sudo apt-get install ocaml ocaml-base ocaml-base-nox ocaml-tools ocaml-nox ocaml-findlib \
libpcre-ocaml libpcre-ocaml-dev libcamomile-ocaml-dev wget \
subversion automake1.9 autoconf make gcc
```

Optional dependencies:

All optional dependencies are installed using their developpement packages. Beware that names may change, but you can easily check them with apt-cache search:

```
apt-cache search lib-name dev
```

```
sudo apt-get install festival libxml-dom-perl tetex-extra python-gtk2-dev python python-support
```

For Alsa:

```
sudo apt-get install libasound-dev
```

For Ogg Vorbis:

```
sudo apt-get install libvorbis-dev libogg-dev
```

For Shout (Ogg streaming):

```
sudo apt-get install libshout3-dev
```

For Lame (MP3 encoding - you need the multiverse APT repository):

```
sudo apt-get install liblame-dev
```

For Mad (MPEG 1 Layer II/III decoding):

```
sudo apt-get install libmad0-dev
```

For AO:

```
sudo apt-get install libao-dev
```

For RTP:

```
sudo apt-get install libortp4-dev
```

or (starting from Feisty):

```
sudo apt-get install libortp5-dev
```

Then checkout the latest SVN version:

```
svn co https://savonet.svn.sourceforge.net/svnroot/savonet/trunk savonet
```

Edit `savonet/PACKAGES` and select the packages you want to compile.
Compile it:

```
cd savonet
./bootstrap && ./configure && make
```

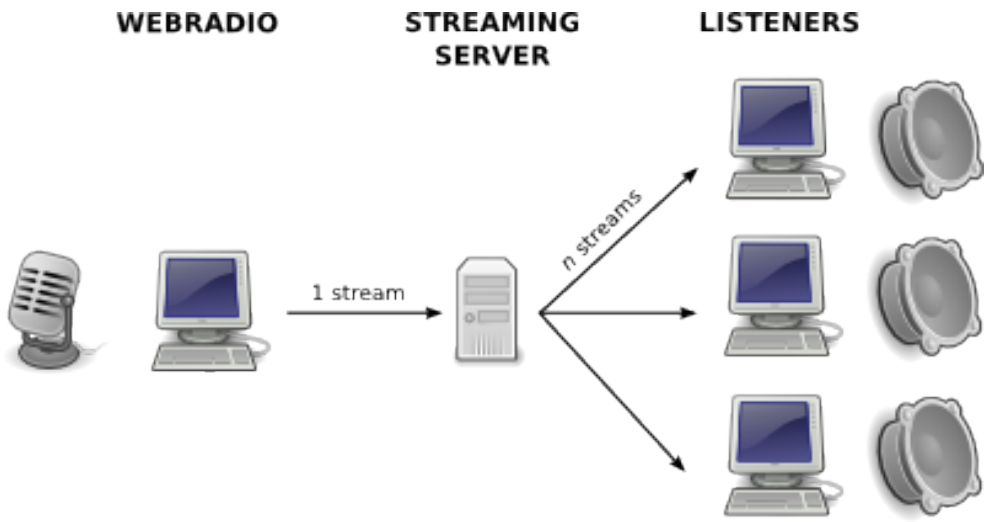


Figure 1.1: Internet radio toolchain

1.3.1 Known bugs

Some camomile packages are broken on Ubuntu – the latest at the time of writing this line. If your file `/usr/lib/ocaml/*/camomile/META` (where * is your version of ocaml) is empty or non-existent, fill it with the following lines:

```
name="camomile"
version="0.6.3"
description="Unicode library for ocaml"
requires="bigarray"
archive(byte)="camomile.cma"
archive(native)="camomile.cmxa"
```

1.4 Quickstart

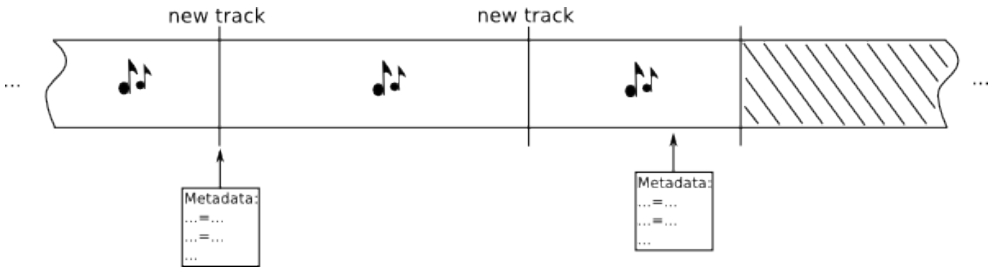
1.4.1 The Internet radio toolchain

Liquidsoap is a general audio stream generator, but is mainly intended for Internet radios. Before starting with the proper Liquidsoap tutorial let's describe quickly the components of the internet radio toolchain, in case the reader is not familiar with it.

The chain is made of:

- the stream generator (Liquidsoap, ices (http://www.icecast.org/ices.php), or for example a DJ-software running on your local PC) which creates an audio stream (Ogg Vorbis or MP3);
- the streaming media server (Icecast (http://www.icecast.org), Shoutcast (http://www.shoutcast.com), ...) which relays several streams from their sources to their listeners;
- the media player (xmms, Winamp, ...) which gets the audio stream from the streaming media server and plays it to the listener's speakers.

The stream is always passed from the stream generator to the server, whether or not there are listeners. It is then sent by the server to every listener. The more listeners you have, the more bandwidth you need.



If you use Icecast, you can broadcast more than one audio feed using the same server. Each audio feed or stream is identified by its "mount point" on the server. If you connect to the `foo.ogg` mount point, the URL of your stream will be `http://localhost:8000/foo.ogg` – assuming that your Icecast is on localhost on port 8000. If you need further information on this you might want to read Icecast's documentation (`http://www.icecast.org`). A proper setup of a streaming server is required for running savonet.

Now, let's create an audio stream.

1.4.2 Starting to use Liquidsoap

In this tutorial we assume that you have a fully installed Liquidsoap. In particular the library `utils.liq` should have been installed, otherwise Liquidsoap won't know the operators which have been defined there. If you installed into the default `/usr/local` you will find it inside `/usr/local/lib/liquidsoap/`.

Sources

A stream is built with Liquidsoap by using or creating sources. A source is an annotated audio stream. In the following picture we represent a stream which has at least three tracks (one of which starts before the snapshot), and a few metadata packets (notice that they do not necessarily coincide with new tracks).

"""""

In a Liquidsoap script, you build source objects. Liquidsoap provides many functions for creating sources from scratch (e.g. `playlist`), and also for creating complex sources by putting together simpler ones (e.g. `switch` in the following example). Some of these functions (typically the `output.*`) create an active source, which will continuously pull its children's stream and output it to speakers, to a file, to a streaming server, etc. These active sources are the roots of a Liquidsoap instance, the sources which bring life into it.

That source is fallible! There may always be errors with your streaming setup. If your source consists of a playlist, this playlist might not be able to find the files it contains because you put them at a wrong place. Liquidsoap also offers you a way to broadcast files from remote places. This remote connection might be broken at the time Liquidsoap expects it to be available.

In Liquidsoap, we say that a source is infallible if it will be always available. Otherwise, it is fallible. A playlist is fallible because it does not check its files in advance, and because files can be remote. A user request queue is an other example of fallible source. If `file.ogg` is a valid local file, then `single("file.ogg")` will be an infallible source. You can also build infallible playlists by using the `playlist.safe` function; it will then check all files at startup, and won't accept remote files.

Liquidsoap checks that the child of an output is infallible, so that you can trust your output. The function `mksafe` takes a source and returns an infallible source, by streaming silence when the input stream becomes unavailable. The default speaker output `out` actually uses `mksafe` in its definition. For other outputs, we ask the user to explicitly write the fallback method, if one is needed. You can use `mksafe` (especially if you trust that your theoretically fallible source will

never fail in practice), but there are other methods for dealing with failures, e.g. playing a default file:

```
fallback([put_your_source_here, single("failure.ogg")])
```

Note: this strict behaviour may change in the future.

1.4.3 One-line expressions

Liquidsoap is a scripting language. Many simple setups can be achieved by evaluating one-line expressions.

Playlists

In the first example we'll play a playlist. Let's put a list of audio files in `playlist.pls`: one filename per line, lines starting with a `#` are ignored. You can also put remote files' URLs, if your Liquidsoap has support for the corresponding protocols. You can also use a directory name instead of a playlist, and Liquidsoap will recursively look for audio files in it. Then just run:

```
liquidsoap 'out(playlist("playlist.m3u"))'
```

(Although full M3U format should be supported by Liquidsoap, we advise you to use the much simpler one-file-per-line format.)

Depending on your configuration, the output `out` will use AO, Alsa or OSS, or won't do anything if you do not have support for these libs. In that case, the next example is for you.

Streaming out to a server

Liquidsoap is capable of playing audio on your speakers, but it can also send audio to a streaming server such as Icecast or Shoutcast. You can choose between two widespread audio codecs: MP3 and Ogg Vorbis. One instance of Liquidsoap can stream one audio feed in many formats (and even many audio feeds in many formats!).

You may already have an Icecast server. Otherwise you can install and configure your own Icecast server. The configuration typically consists in setting the admin and source passwords, in `/etc/icecast2/icecast.xml`. These passwords should really be changed if your server is visible from the hostile internet, unless you want people to kick your source as admins, or add their own source and steal your bandwidth.

We are now going to send an audio stream, encoded as Ogg Vorbis, to an Icecast server:

```
liquidsoap 'output.icecast.vorbis(host = "localhost", port = 8000, \
                                  password = "hackme", mount = "liq.ogg",  \
                                  mksafe(playlist("playlist.m3u")))'
```

The main difference with the previous is that we used `output.icecast.vorbis` instead of `out`. The second difference is the use of the `mksafe` which turns your fallible playlist source into an infallible source.

Streaming to Shoutcast is quite similar, using the `output.shoutcast.mp3` function:

```
liquidsoap 'output.shoutcast.mp3(host="localhost", port = 8000, \
                                  password = "changeme", \
                                  mksafe(playlist("playlist.m3u")))'
```

Input from another streaming server

Liquidsoap can use another stream as an audio source. This may be useful if you do some live shows.

```
liquidsoap 'out(input.http("http://dolebrai.net:8000/dolebrai.ogg"))'
```

Input from the soundcard

If you're lucky and have a working ALSA support, try one of these... but beware that ALSA may not work out of the box.

```
liquidsoap 'output.alsa(input.alsa())'
liquidsoap 'output.alsa(bufferize = false, input.alsa(bufferize = false))'
```

Other examples

You can play with many more examples. Here are a few more. To build your own, lookup the API documentation to check what functions are available, and what parameters they accept.

```
# Listen to your playlist, but normalize the volume
liquidsoap 'out(normalize(playlist("playlist_file")))'
# ... same, but also add smart cross-fading
liquidsoap 'out(smart_crossfade(normalize(playlist("playlist_file"))))'
```

1.4.4 Script files

We have seen how to create a very basic stream using one-line expressions. If you need something a little bit more complicated, they will prove uneasy to manage. In order to make your code more readable, you can write it down to a file, named with the extension `.liq` (eg: `myscript.liq`).

To run the script:

```
liquidsoap myscript.liq
```

On UNIX, you can also put `#!/path/to/your/liquidsoap` as the first line of your script ("shebang"). Don't forget to make the file executable:

```
chmod u+x myscript.liq
```

Then you'll be able to run it like this:

```
./myscript.liq
```

Usually, the path of the Liquidsoap executable is `/usr/bin/liquidsoap`, and we'll use this in the following.

1.4.5 A simple radio

We will start with a basic radio station, that plays songs randomly chosen from a playlist, adds a few jingles (more or less one every four songs), and output an Ogg Vorbis stream to an Icecast server.

Before reading the code of the corresponding liquidsoap script, it might be useful to visualize the streaming process with the following tree-like diagram. The idea is that the audio streams flows through this diagram, following the arrows. In this case the nodes (`fallback` and `random`) select one of the incoming streams and relay it. The final node `output.icecast.vorbis` is an output: it actively pulls the data out of the graph and sends it to the world.

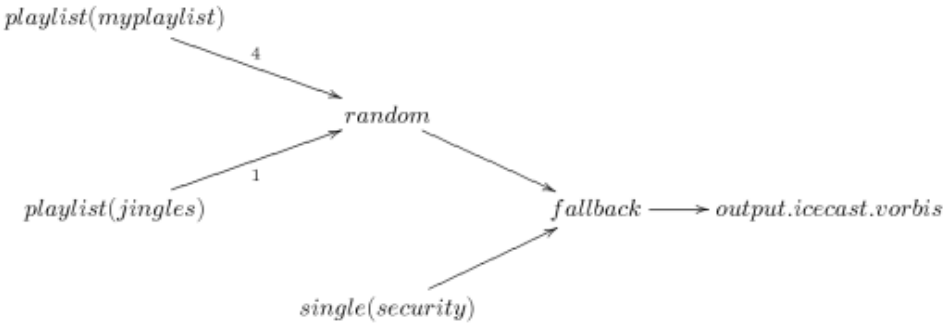


Figure 1.2: Graph for 'basic-radio.liq'

```
#!/usr/bin/liquidsoap

# Log dir
set("log.file.path","/tmp/basic-radio.log")

# Music
myplaylist = playlist("~/radio/music.m3u")
# Some jingles
jingles = playlist("~/radio/jingles.m3u")
# If something goes wrong, we'll play this
security = single("~/radio/sounds/default.ogg")

# Start building the feed with music
radio = myplaylist
# Now add some jingles
radio = random(weights = [1, 4], [jingles, radio])
# And finally the security
radio = fallback(track_sensitive = false, [radio, security])

# Stream it out
output.icecast.vorbis(host = "localhost", port = 8000, password = "hackme",
                          mount = "basic-radio.ogg", radio)
```

1.4.6 What's next?

You can first have a look at a more complex example. There is also a second tutorial about advanced techniques.

You should also learn more about Liquidsoap's scripting language. Once you'll know the syntax and types, you'll probably need to refer to the scripting reference and the settings reference, or see examples. For a better understanding of Liquidsoap, it is also suggested to read more about the concepts of the system.

1.5 A complete case analysis

We will develop here a more complex example, according to the following specifications:

- play different playlists during the day;
- play user requests – done via the telnet server;

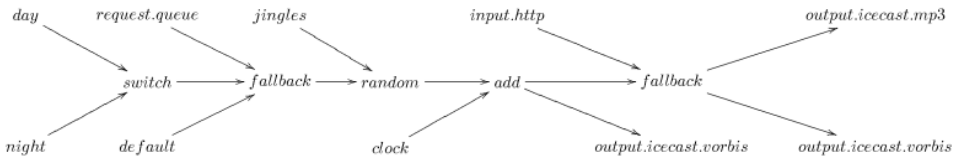


Figure 1.3: Graph for 'radio.liq'

- insert about 1 jingle every 5 songs;
- add one special jingle at the beginning of every hour, mixed on top of the normal stream;
- relay live shows as soon as one is available;
- and set up several outputs.

Once you've managed to describe what you want in such a modular way, you're half the way. More precisely, you should think of a diagram such as the following, through which the audio streams flow, following the arrows. The nodes can modify the stream using some basic operators: switching and mixing in our case. The final nodes, the ends of the paths, are outputs: they are in charge of pulling the data out of the graph and send it to the world. In our case, we only have outputs to icecast, using two different formats.

Now here is how to write that in Liquidsoap.

```
#!/usr/bin/liquidsoap

# Lines starting with # are comments, they are ignored.

# Put the log file in some directory where you have the write permission
set("log.file.path","/tmp/<script>.log")
# Print log messages to the console, this can also be done by passing the -v option to liquidsoap
set("log.stdout", true)
# Use the telnet server for requests
set("server.telnet", true)

# A bunch of files and playlists,
# supposedly all located in the same base dir.

default = single("~/radio/default.ogg")

day     = playlist("~/radio/day.pls")
night   = playlist("~/radio/night.pls")
jingles = playlist("~/radio/jingles.pls")

clock   = single("~/radio/clock.ogg")
start   = single("~/radio/live_start.ogg")
stop    = single("~/radio/live_stop.ogg")

# Play user requests if there are any,
# otherwise one of our playlists,
# and the default file if anything goes wrong.
radio = fallback([ request.queue(id="request"),
                        switch([({ 6h-22h }, day),
                                ({ 22h-6h }, night)]),
                        default])
# Add the normal jingles
radio = random(weights=[1,5],[ jingles, radio ])
```

```
# And the clock jingle
radio = add([radio, switch([({0m0s},clock)])])

# Add the ability to relay live shows
full = fallback(track_sensitive=false,
                        [input.http("http://localhost:8000/live.ogg"),
                         radio])

# Output the full stream in OGG and MP3
output.icecast.mp3(host="localhost",port=8000,password="hackme",
                       mount="radio",full)
output.icecast.vorbis(host="localhost",port=8000,password="hackme",
                          mount="radio.ogg",full)

# Output the stream without live in OGG
output.icecast.vorbis(host="localhost",port=8000,password="hackme",
                          mount="radio_nolive.ogg",radio)
```

To try this example you need to edit the file names. In order to witness the switch from one playlist to another you can change the time intervals. If it is 16:42, try the intervals `0h-16h45` and `16h45-24h` instead of `6h-22h` and `22h-6h`. To witness the clock jingle, you can ask for it to be played every minute by using the `0s` interval instead of `0m0s`.

To try the transition to a live show you need to start a new stream on the `live.ogg` mount of your server. You can send a playlist to it using the first example. To start a real live show you can use `darkice`, or simply Liquidsoap if you have a working ALSA input, with:

```
liquidsoap 'output.icecast.vorbis(mount="live.ogg",host="...",password="...",input.alsa())'
```

1.6 Advanced techniques

1.6.1 Interaction with the server

To enable the telnet server, `set server.telnet = true` (or use the `-t` option).

In "A Complete Case Analysis" we set up a `request.queue` source to play user requests. To push requests in that queue you need to interact with the telnet server, which also provides many other services. By default it is only accessible from the host where Liquidsoap runs. You can learn more on that topic with the telnet tutorial and settings description. Here is a sample session:

```
dbaelde@selassie:~$ telnet localhost 1234
Trying 127.0.0.1...
Connected to localhost.localdomain.
Escape character is '^]'.
request.push /path/to/some/file.ogg
5
END
metadata 5
[...]
END
request.push http://remote/audio.ogg
6
END
trace 6
[...]
END
help
[...]
```

```
END
exit
```

Of course, telnet isn't user friendly. But it is easy to write scripts to interact with Liquidsoap in that way. Examples of such tools are liguidsoap and bottle.

1.6.2 Daemon mode

The full installation of Liquidsoap will typically install `/etc/liquidsoap`, `/etc/init.d/liquidsoap` and `/var/log/liquidsoap`. All these are meant for a particular usage of Liquidsoap when running a stable radio.

Your `.liq` files should go in `/etc/liquidsoap`. You'll then start/stop them using the init script: `/etc/init.d/liquidsoap start`. Your scripts don't need to have the `#!` line. Liquidsoap will automatically be ran on daemon mode (`-d` option) for them.

You should not override the `log.file.path`, because a logrotate configuration is also installed so that log files in the standard directory are truncated and compressed if they grow too big.

It is not very convenient to detect errors when using the init script. We advise users to first check their modified scripts using `liquidsoap --check /etc/liquidsoap/script.liq` before effectively restarting the daemon.

1.7 Concepts

1.7.1 Sources

Using liquidsoap is about writing a script describing how to build what you want. It is about building a stream using elementary streams and stream combinators, etc. Actually, it's a bit more than streams, we call them sources – in liquidsoap's code there is a `Source.source` type, and in `*.liq` scripts one of the elementary datatypes is source.

A source is a stream with metadata and track annotations. It is discretized as a stream of fixed-length buffers of raw audio, the frames. Every frame may have metadata inserted at any point, independently of track boundaries. At every instant, a source can be asked to fill a frame of data. Track boundaries are denoted by a single denial of completely filling a frame. More than one denial is taken as a failure, and liquidsoap chooses to crash in that case.

To build sources in liquidsoap scripts, you need to call functions which return type is `source`. For convenience, we categorize these functions into three classes. The *sources* (sorry for redundancy, poor historical reasons) are functions which don't need a source argument – we might call them elementary sources. The *operators* need at least one source argument – they're more about stream combination or manipulation. Finally, some of these are called *outputs*, because they are active operators (or active sources in a few cases): at every instant they will fill their buffer and do something with it. Other sources just wait to be asked (indirectly or not) by an output to fill some frame.

All sources, operators and outputs are listed in the scripting API reference.

How does it work?

To clarify the picture let's study in more details an example.

```
radio = output.icecast(
        mount="test.ogg",
        random([ jingle , fallback([ playlist1 , playlist2 , playlist3 ])])
        )
```

At every tick, the output asks the "random" node for data, until it gets a full frame of raw audio. Then it encodes it, and sends it to the Icecast server. Suppose "random" has chosen the

"fallback" node, and that only "playlist2" is available, and thus played. At every tick, the buffer is passed from "random" to "fallback" and then to "playlist2", which fills it, returns it to "fallback", which returns it to "random", which returns it to the output. Every step is local.

At some point, "playlist2" ends a track. The "fallback" detects that on the returned buffer, and selects a new child for the next filling, depending on who's available. But it doesn't change the buffer, and returns it to "random", which also selects a new child, randomly, and return the buffer to the output. On next filling, the route of the frame can be different.

It is possible to have the route changed inside a track, for example using the `track_sensitive` option of fallback, which is typically done for instant switches to live shows when they start.

Fallibility

Outputs expect their input source to never fail, so that the stream never ends. Liquidsoap has a mechanism to verify this, and helps you think of all possible failures, and prevent them. Elementary sources are either *fallible* or *infallible*, and this liveness type is propagated through operators to finally compute the type of any source. A `fallback` or `random` source is infallible if and only if at least one of their children is infallible. A `switch` is infallible if and only if it has one infallible child guarded by the trivial predicate `true` . And so on.

On startup, outputs will check the liveness type of their input sources, and you'll get an error if one of these is fallible. The common answer to such errors is op add one fallback to play a default file or a checked playlist (`playlist.safe`) if the normal source fails. Often, the error is excessive, it simply means that your (unchecked) playlists could all be corrupted, which is unlikely. But sometimes, it also helps one to avoid the case where a playlist fails because it spent too much time trying to download remote files.

Caching mode

In some situations, a source must take care about the consistency of its output. If it is asked twice to fill buffers during the same time tick, it should fill them with the same data. Suppose for example that a playlist is listened by two outputs, and that it gives the first frame to the first output, the second frame to the second output: it would give the third frame to the first output during the second tick, and the output will have missed one frame.

Keeping that in mind is required to understand the behaviour of some complex scripts. The high-level picture is enough for users, more details follow for developers and curious readers.

The sources detect if they need to remember (cache) their previous output in order to replay it. To do that, clients of the source must register in advance. If two clients have registered, then the caching should be enabled. Actually that's a bit more complicated, because of transitions. Obviously the sources which use a transition involving some other source must register to it, because they may eventually use it. But a jingle used in two transitions by the same switching operator doesn't need caching. The solution involves two kinds of registering: *dynamic* and *static activations*. Activations are associated with a path in the graph of sources' nesting. The dynamic activation is a pre-registration allowing a real *static activation* to come later, possibly in the middle of a time tick, from a super-path – *i.e.* a path of which the first one is a prefix. Two static activations trigger caching. The other reason for enabling caching is when there is one static activation and one dynamic activation which doesn't come from a prefix of the static activation's path. It means that the dynamic activation can yield at any moment to a static activation and that the source will be used by two sources at the same time.

1.7.2 Execution model

In your script you define a bunch of sources interacting together. The output sources hook their output function to the root thread manager. Then the streaming starts. At every tick the root thread calls the output hooks, and the outputs do their jobs. This task is the most important and shouldn't be disturbed. Thus, other tasks are done in auxiliary threads: file download, audio

validity checking, http polling, playlist reloading... No blocking or expensive call should be done in the root thread. Remote files are completely downloaded to a local temporary file before use by the root thread. It also means that you shouldn't access NFS or any kind of falsely local files.

1.7.3 An abstract notion of files: requests

The request is an abstract notion of file which can be conveniently used for defining powerful sources. A request can denote a local file, a remote file, or even a dynamically generated file. They are resolved to a local file thanks to a set of *protocols*. Then, audio requests are transparently decoded thanks to a set of audio and metadata *formats*.

The systematic use of requests to access files allows you to use remote URIs instead of local paths everywhere. It is perfectly OK to create a playlist for a remote list containing remote URIs: ""`playlist("http://my/friends/playlist.pls")`"".

The resolution process

The nice thing about resolution is that it is recursive and supports backtracking. An URI can be changed into a list of new ones, which are in turn resolved. The process succeeds if some valid local file appears at some point. If it doesn't succeed on one branch then it goes back to another branch. A typical complex resolution would be:

- `bubble:artist = "bodom"`
  - `ftp://no/where`
    - Error
  - `ftp://some/valid.ogg`
    - `/tmp/success.ogg`

On top of that, metadata is extracted at every step in the branch. Usually, only the final local file yields interesting metadata (artist,album,...). But metadata can also be the nickname of the user who requested the song, set using the `annotate` protocol.

At the end of the resolution process, if the request is an audio one, liquidsoap check that the file is decodable: there should be a valid decoder for it (this isn't based on the extension but on the success of a format decoder), the decoder shouldn't yield an empty stream, and opening the decoder should be fast (less than 0.5 seconds), so that the opening of the audio file for its real playing in the main thread doesn't freeze it for too long.

Currently supported protocols

- SMB, FTP and HTTP using ufetch (provided by our ocaml-fetch distribution)
- HTTP, HTTPS, FTP thanks to wget
- SAY for speech synthesis (requires festival): `say:I am a robot` resolves to the WAV file resulting from the synthesis.
- TIME for speech synthesis of the current time: `time: It is exactly @, and you're still listening to Geek Radio.`
- ANNOTATE for manually setting metadata, typically used in ""`annotate:nick="vodka-goo",media=irc,message="special for sam":ftp://bla/bla/bla`"". The extra metadata can then be synthesized in the audio stream, or merged into the standard metadata fields, or used on a rich web interface...

It is also possible to add a new protocol from the script, as it is done with bubble for getting songs from a database query.

Currently supported formats

- MPEG-1 Layer II (MP2) and Layer III (MP3) through libmad and ocaml-mad ;
- Ogg Vorbis through libvorbis and ocaml-vorbis ;
- WAV.
- AAC.

1.8 Liquidsoap's scripting language

Liquidsoap's scripting language is a simple functional language, with labels and optional parameters. It is statically typed, but infers types – you don't have to write any types. To fit its particular purpose, it has first-class sources and requests (see liquidsoap's concepts) and a syntax extension for simply specifying time intervals.

1.8.1 Constants

Constants syntax is quite common:

- integers, such as `42`;
- floats, such as `3.14`;
- booleans, `true` and `false`;
- strings, such as "foo" or 'bar'.

Beware: `3.0` is not an integer and `5` is not a float, the dot matters.

Strings might be surrounded by double or single quotes. In both cases, you can escape the quote you're using: `"He said: \"Hello, you\"."` is valid but `'He said: "Hello, you".'` is equivalent and nicer.

1.8.2 Expressions

You can form expressions by using:

- Constants and variable identifiers. Identifier are made of alphanumerics, underscore and dot: `[a-zA-Z0-9_\.]*`
- Lists and tuples: `[expr,expr,...]` and `(expr,expr,..)`
- Sequencing: expressions may be sequenced, just juxtapose them. Usually one puts one expression per line. Optionally, they can be separated by a semicolon. The type of a sequence of expressions is the type of the last expression – just as a sequence evaluates to its last expression.
- Application `f(x,y)` of arguments to a function. Application of labeled parameters is as follows: `f(x,foo=1,y,bar="baz")`. The relative order of two parameters doesn't matter as long as they have different labels.
- Definitions using def-end: `def source(x) = s = wrap1(x) ; wrap2(s) end` or `def pi = 3.14 end`. The `=` is optional, you may prefer multi-line definitions without it. The definition of a function with two named parameters, the second one being optional with default value `13` is as follows: `def f(~foo,~bar=13) = body end`.
- Shorter definitions using the equality: `pi = 3.14`. This is never an assignment, only a new local definition!

- Anonymous functions: `fun (arglist) -> expr`. Don't forget to use parenthesis if you need more than one expr: `fun (x) -> f1(x) ; f2(x)` will be read as `(fun (x) -> f1(x)) ; f2(x)` not as `fun (x) -> (f1(x) ; f2(x))`.
- Code blocks: `expr` is a shortcut for `fun () -> expr`.

No assignation, only definitions. `x = expr` doesn't modify `x`, it just defines a new `x`. The expression `(x = s1 ; def y = x = s2 ; (x,s3) end ; (y,x))` evaluates to `((s2,s3),s1)`.

Function. The return value of a function is its body where parameters have been substituted. Accordingly, the type of the body is the return type of the function. If the body is a sequence, the return value will thus be its last expression, and the return type its type.

```
# return type of foo will be string.
def foo ()
         a = bar()
         b = 1
         "string"
end
```

Type of an application. The type of an application is the return type of function if all mandatory arguments are applied:

```
def foo ()
        1
end

# a will be an integer
a = foo()
```

Otherwise, the application is "partial", and the expression has the type of a function.

Partial application. Application of arguments can be partial. For example if `f` takes two integer arguments, `f(3)` is the function waiting for the second argument. This can be useful to instantiate once for all dummy parameters of a function:

```
out = output.icecast.vorbis(host="streamer",port="8080",password="sesame")
out(bitrate=112, my_radio)
```

Labels. Labeled and unlabeled parameters can be given at any place in an application. The order of the arguments is up to permutation of arguments of distinct labels. For example `f(x,foo=1)` is the same as `f(foo=1,x)`, both are valid for any function `f(x,~foo,...)`. It makes things easier for the user, and gives its full power to the partial application.

Optional arguments. Functions can be defined with an optional value for some parameter (as in `def f(x="bla",~foo=1) = ... end`), in which case it is not required to apply any argument on the label `foo`. The evaluation of the function is triggered after the first application which instantiated all mandatory parameters.

1.8.3 Types

We believe in static typing especially for a script which is intended to run during weeks: we don't want to notice a mistake only when the special code for your rare live events is triggered! Moreover, we find it important to show that even for a simple script language like that, it is worth implementing type inference. It's not that hard, and makes life easier.

The basic types are `int`, `float`, `bool`, `string`, but also `source` and `request`. Corresponding to pairs and lists, you get `(T*T)` and `[T]` types – all elements of a list should have the same type. For example, `[(1,"un"),(2,"deux)]` has type `[(int*string)]`.

A function type is noted as `(arg_types) -> return_type`. Labeled arguments are denoted as `~label:T` or `?label:T` for optional arguments. For example the following function has type `(source,source,?jingle:string) -> source`.

```
fun (from,to,~jingle=default) ->
        add ([ sequence([single(jingle), fallback([])]), fade.initial(to) ])
```

1.8.4 Time intervals

The scripting language also has a syntax extension for simply specifying time intervals.

A date can be specified as `?w?h?m?s` where `?` are integers and all components `?x` are optional. It has the following meaning:

- `w` stands for weekday, ranging from 0 to 7, where 1 is monday, and sunday is both 0 and 7.
- `h` stands for hours, ranging from 0 to 23.
- `m` stands for minutes, from 0 to 59.
- `s` stands for seconds, from 0 to 59.

It is possible to use 24 (resp. 60) as the upper bound for hours (resp. seconds or minutes) in an interval, for example in `12h-24h`.

It is possible to forget the `m` for minutes if hours are specified – and seconds unspecified, obviously.

Time intervals can be either of the form `DATE-DATE` or simply `DATE`. Their meaning should be intuitive: `10h-10h30` is valid everyday between 10:00 and 10:30; `0m` is valid during the first minute of every hour.

This is typically used for specifying switch predicates:

```
switch([
  ({ 20h-22h30 }, prime_time),
  ({ 1w }, monday_source),
  ({ (6w or 7w) and 0h-12h }, week_ends_mornings),
  ({ true }, default_source)
])
```

1.8.5 Includes

You can include other files, to compose complex configurations from multiple blocks of utility or configuration directives.

```
# Store passwords in another configuration file, so that the main config can be safely version-controlled.
%include "passwords.liq"

# Use the definitions from the other file here.
```

1.9 Liquidsoap settings

Liquidsoap scripts contain settings, of the form `set("settings.variable.path",value)`, for defining a few global variables affecting the behaviour of the application. The settings are typed, and can be `string`, `int`, `bool` or `string list`.

You can have a list of available parameters, with their documentation, by running `liquidsoap --conf-descr`. If you are interested in a particular settings section, like server-related stuff, use `liquidsoap --conf-descr-key server`.

The output of these commands is a valid liquidsoap script, which you can edit to set the values that you want, and load it (implicitly or not) before you other scripts.

1.10 Cookbook

The recipes show how to build a source with a particular feature. You can try short snippets by wrapping the code in an `out(..)` operator and passing it directly to liquidsoap:

```
liquidsoap -v 'out(recipe)'
```

For longer recipes, you might want to create a short script:

```
#!/usr/bin/liquidsoap -v

set("log.file.path","/tmp/<script>.log")
set("log.stdout",true)

recipe = # <fill this>
out(recipe)
```

See the quickstart guide for more information on how to run Liquidsoap, on what is this `out(..)` operator, etc.

1.10.1 Files

A source which infinitely repeats the same URI:

```
single("/my/default.ogg")
```

A source which plays a playlist of requests – a playlist is a file with an URI per line.

```
# Shuffle, play every URI, start over.
playlist("/my/playlist.txt")
# Do not randomize
playlist(mode="normal", "/my/pl.m3u")
# The playlist can come from any URI, can be reloaded every 10 minutes.
playlist(reload=600,"http://my/playlist.txt")
```

When building your stream, you'll need to make it unfallible. Usually, you achieve that using a fallback switch (see below) with a branch made of a safe `single` or `playlist.safe`. Roughly, a single is safe when it is given a valid local audio file. A `playlist.safe` behaves just like a playlist but will check that all files in the playlist are valid local audio files. This is quite an heavy check, you don't want to have large safe playlists.

1.10.2 Transcoding

Liquidsoap can achieve basic streaming tasks like transcoding with ease. You input any number of "source" streams using `input.http`, and then transcode them to any number of formats / bitrates / etc. The only limitation is your hardware : encoding and decoding are both heavy on CPU. Also keep in mind a limitation inherent to OCaml: one Liquidsoap instance can only use a single processor or core. You can easily work around this limitation by launching multiple Liquidsoap instances, and thus take advantage of that 8-core Xeon server laying around in the dust in your garage.

```
# Input the stream for an Icecast server (or any other source)
input = mksafe(input.http("http://streaming.example.com:8000/your-stream.ogg"))

# First transcoder: MP3 32 kbps
```

```
output.icecast.mp3(mount="/your-stream-32.mp3", bitrate = 32,
  samplerate = 22050, stereo = false, host="streaming.example.com",
  port=8000, password="hackme", input)
# Second transcoder : MP3 128 kbps
output.icecast.mp3(mount="/your-stream-128.mp3", bitrate = 128,
  host="streaming.example.com", port=8000, password="hackme",
  input)
```

1.10.3 Scheduling

```
# A fallback switch
fallback([playlist("http://my/playlist"), single("/my/jingle.ogg")])
# A scheduler, assuming you have defined the night and day sources
switch([ ({0h-7h}, night), ({7h-24h}, day) ])
```

1.10.4 Force a file/playlist to be played at least every XX minutes

It can be useful to have a special playlist that is played at least every 20 minutes for instance (3 times per hour).

You may think of a promotional playlist for instance.

Here is the recipe:

```
timed_promotions = delay(1200.,promotions) # 1200 sec = 20 min
main_source = fallback([timed_promotions,other_source])
```

Where promotions is a source selecting the file to be promoted.

1.10.5 Handle special events: mix or switch

```
# Add a jingle to your normal source at the beginning of every hour:
add([normal,switch([({0m0s},jingle)])])
```

Switch to a live show as soon as one is available. Make the show unavailable when it is silent, and skip tracks from the normal source if they contain too much silence.

```
fallback(track_sensitive=false,
             [strip_blank(input.http("http://myicecast:8080/live.ogg")),
              skip_blank(normal)])
```

Without the `track_sensitive=false` the fallback would wait the end of a track to switch to the live. When using the blank detection operators, make sure to fine-tune their `threshold` and `length` (float) parameters.

1.10.6 Unix interface, dynamic requests

`request.dynamic` is a source which takes a custom function for creating its new requests. This function can be used to call an external program. The source expects a `()->request` function. To create the request, the function will have to use the `request` function which has type `(string,?indicators:[string])`. The first string is the initial URI of the request, which is resolved to get an audio file. The second argument can be used to directly specify the first row of URIs (see the concepts page), in which case the initial URI is just here for naming, and the resolving process will try your list of indicators one by one until a valid audio file is obtained.

The simplest example takes the output of an external script as an URI to create a new request:

```
request.dynamic({ request(get_process_output("my_script my_params")) })
```

More complex, the following snippet defines a source which repeatedly plays the first valid URI in the playlist:

```
request.dynamic({ request("bar:foo",
                          indicators=get_process_lines("cat "^quote("playlist.pls"))) })
```

Of course a more interesting behaviour is obtained with a more interesting program than "cat".

Another way of using an external program is to define a new protocol which uses it to resolve URIs. `add_protocol` takes a protocol name, a function to be used for resolving URIs using that protocol. The function will be given the URI parameter part and the time left for resolving – though nothing really bad happens if you don't respect it. It usually passes the parameter to an external program, that's how we use bubble for example:

```
add_protocol("bubble",
             fun (arg,delay) -> get_process_lines("/usr/bin/bubble-query "^quote(arg)))
```

When resolving the URI `bubble:artist="seed"`, liquidsoap will call the function, which will call `bubble-query 'artist="seed"'` which will output 10 lines, one URI per line.

1.10.7 Dynamic input with harbor

The operator `input.harbor` allows you to receive a source stream directly inside a running liquidsoap.

It starts a listening server on where any Icecast2-compatible source client can connect. When a source is connected, its input if fed to the corresponding source in the script, which becomes available.

This can be very useful to relay a live stream without polling the Icecast server for it.

An example can be:

```
# Serveur settings
set("harbor.bind_addr","0.0.0.0")
set("harbor.port",8080)
set("harbor.password","hackme")

# An emergency file
emergency = single("/path/to/emergency/single.ogg")

# A playlist
playlist = playlist("/path/to/playlist")

# A live source
live = input.harbor("live")

# fallback
radio = fallback(track_sensitive=false,[live,playlist,emergency])

# output it
output.icecast.vorbis(radio,mount="test",host="host")
```

This script, when launched, will start a local server, here bound to "0.0.0.0". This means that it will listen on any IP address available on the machine for a connection coming from any IP address. The server will wait for any source stream on mount point "/live" to login.

Then if you start a source client and tell it to stream to your server, on port 8080, with password "hackme", the live source will become available and the radio will stream it immediately.

1.10.8 Lastfm input

You can listen to lastfm (http://www.last.fm/) radios using Liquidsoap. The corresponding operator is `input.lastfm` and is used that way:

```
lastfm_stream = input.lastfm("lastfm://artist/Wackies")
```

Lastfm's URIs start with `lastfm:`:

- `lastfm://user/toots5446/paylist`: a user's playlist
- `lastfm://globaltags/creative commons`: songs tagged with "creative commons"
- `lastfm://user/toots5446/tags/rocksteady`: songs tagged "rocksteady" by the user.

You can find more of them on the website, last.fm (http://www.last.fm/)
Another operator allows to generate `lastfm:` URIs, `lastfm.uri`. Its parameters are:

- user Lastfm user
- password Lastfm password
- discovery Allow lastfm suggestions
- radio URI, e.g. user/toots5446/playlist, globaltags/rocksteady

Example:

```
uri = lastfm.uri(user="toots5446", password="hackme", discovery=false, "user/toots4556/playlist")
```

1.10.9 Transitions

There are two kinds of transitions. Transitions between two different children of a switch are not problematic. Transitions between different tracks of the same source are more tricky, since they involve a fast forward computation of the end of a track before feeding it to the transition function: such a thing is only possible when only one operator is using the source, otherwise it'll get out of sync.

Switch-based transitions

The switch-based operators (`switch`, `fallback` and `random`) support transitions. For every child, you can specify a transition function computing the output stream when moving from one child to another. This function is given two `source` parameters: the child which is about to be left, and the new selected child. The default transition is `fun (a,b) -> b`, it simply relays the new selected child source. Other possible transition functions:

```
# A simple (long) cross-fade
def crossfade(a,b)
  add(normalize=false,
          [ sequence([ blank(duration=5.),
                       fade.initial(duration=10.,b) ]),
            fade.final(duration=10.,a) ])
end

# Partially apply next to give it a jingle source.
# It will fade out the old source, then play the jingle.
# At the same time it fades in the new source.
def next(j,a,b)
```

```
  add(normalize=false,
          [ sequence(merge=true,
                     [ blank(duration=3.),
                       fade.initial(duration=6.,b) ]),
            sequence([fade.final(duration=9.,a),
                      j,fallback([])]) ])
end

# A similar transition, which does a cross-fading from A to B
# and adds a jingle
def transition(j,a,b)
  add(normalize=false,
          [ fade.initial(b),
            sequence(merge=true,
                     [blank(duration=1.),j,fallback([])]),
            fade.final(a) ])
end
```

Finally, we build a source which plays a playlist, and switches to the live show as soon as it starts, using the `transition` function as a transition. At the end of the live, the playlist comes back with a cross-fading.

```
fallback(track_sensitive=false,
             transitions=[ crossfade, transition(jingle) ],
             [ input.http("http://localhost:8000/live.ogg"),
               playlist("playlist.pls") ])
```

Cross-based transitions

The `cross()` operator allows arbitrary transitions between tracks of a same source. Here is how to use it in order to get a cross-fade:

```
def crossfade(~start_next,~fade_in,~fade_out,s)
  s = fade.in(duration=fade_in,s)
  s = fade.out(duration=fade_out,s)
  fader = fun (a,b) -> add(normalize=false,[b,a])
  cross(fader,s)
end
my_source=crossfade(start_next=1.,fade_out=1.,fade_in=1.,my_source)
```

The fade-in and fade-out parameters indicate the duraction of the fading effects. The start-next parameters tells how much overlap there will be between the two tracks. If you want a long cross-fading with a smaller overlap, you should use a sequence to stick some blank section before the beginning of `b` in `fader`.

The three parameters given here are only default values, and will be overriden by values coming from the metadata tags `liq_fade_in`, `liq_fade_out` and `liq_start_next`.

How to get transitions on a mix?

If you `add()` a special source on top of your normal stream, you might notice that the re-normalization is not smooth at all: if the special source suddenly becomes available, the normal one will be re-normalized immediately, which is not very nice to hear, especially if the special source starts with a low noise level. The `add()` operator does not support transitions but there is a solution for this kind of situation. Use a `fallback()` in order to get transitions, and simply keep playing the normal source in the transition. Here is the code.

```
def smooth_add(~normal,~special)
  d = 1.  # delay before mixing after beginning of mix
  p = 0.2 # portion of normal when mixed
  fade.final = fade.final(duration=d*.2.)
  fade.initial = fade.initial(duration=d*.2.)
  q = 1. -. p
  c = change_volume
  fallback(track_sensitive=false,
               [special,normal],
               transitions=[
                 fun(normal,special)->
                   add(normalize=false,[c(p,normal),
                                        c(q,fade.final(normal)),
                                        sequence([blank(duration=d),c(q,special)])]),
                 fun(special,normal)->
                   add(normalize=false,[c(p,normal),c(q,fade.initial(normal))])
               ])
end
```

The first transition is used when the special source becomes available. It sums the special source (after a delay `d`) together with a reduced version of `normal` (`c(p,normal)`) and its faded-out complement (`c(q,normal)`). As a result the amplitude of `normal` will smoothly move from `1=p q` down to `p`.

The second transition is called when `special` becomes unavailable. This time, the reduced version of `normal` is mixed with its faded-in complement.

1.10.10 Alsa unbuffered output

You can use Liquidsoap to capture and play through alsa with a minimal delay. This particulary useful when you want to run a live show from your computer. You can then directly capture and play audio through external speakers without delay for the DJ !

This configuration is not trivial since it relies on your hardware. Some hardware will allow both recording and playing at the same time, some only one at once, and some none at all.. Those note to configure are what works for us, we don't know if they'll fit all hardware.

First launch liquidsoap as a one line program

```
liquidsoap -v --debug 'input.alsa(bufferize=false)'
```

Unless you're lucky, the logs are full of lines like the following:

```
Partial read (940 instead of 1024)! Selecting another buffer size or device can help.
```

The solution is then to fix the captured frame size to this value, which seems specific to your hardware. Let's try this script:

```
# Set correct frame size:
set("frame.size",940)

input = input.alsa(bufferize=false)
output.alsa(bufferize=false,input)
```

If everything goes right, you may hear on your output the captured sound without any delay ! If you want to test the difference, just run the same script with bufferize=true (or without this parameter since it is the default).

If you experience problems it might be a good idea to double the value of the frame size. This increases stability, but also latency.

1.11 Frequently Asked Questions

1.11.1 I started receiving this log on my streams: `We must catchup 0.44 seconds (we've been late for 100 rounds)!` What does it mean?

Liquidsoap is a (sloppy) real-time application. I won't detail what the sloppiness is about, it really doesn't matter in most usages anyway. It obviously has to care about time, since it is streaming (audio) data which has an intended rate. The streaming process (read more about liquidsoap's concepts) is split in rounds: in every round a tiny bit of data is computed and sent. A round has an intended duration. If the computation takes too long, that duration may be exceeded. In that case, liquidsoap will try to make the next rounds smaller to "catchup" on the latency. If the latency gets really high, liquidsoap reports it. It also reports if a little latency persists for more that 100 rounds, which is the case in that message.

That's it for the stupid description. Now, when should one care about these warnings, and how to fix them ? In net radio usages, tiny delays don't matter and are not reported: there are severals buffers which hide the latency. (It matters more with low latency usages, typically hardware I/O.) If the latency gets really high, the users might get underruns, and you might experience disconnections. In that case it means that you liquidsoap setup is really too computation-expensive for your machine: consider downgrading quality, distributing encoding, removing audio processings, etc.

There are two main process where you can suffer a bottleneck:

1) Encoding on the fly

2) Keepalive liquidsoap

The first is normally high, specially if you use a single audio file and deliver it over too many different bitrates. I have noticed that over 40 reencodings on a pentium D liquidsoap starts complaining, if the recoding use high bitrates, it gets worse, for example you should expect more computation-expensive reencoding at 96kb than at 32kb. There is a simple and elegant solution, reencode by hand your audio files, and create parallel folders, normally space on disk is cheaper than processor speed and easier to increment.

The second is the normal processor and RAM resources needed by a liquidsoap script to stay alive and keep communicated with your output, liquidsoap does not consume to much RAM, although situations may vary I have never seen a liquidsoap script using more than 3 Mbytes of RAM, the CPU usage is low if you do not reencode on the fly, and IS CONSTANT. to calculate how many liquidsoap scripts you can keep at the same time, just stop any other services, start a liquidsoap script and check how much processor uses.

Chapter 2

Advanced topics

2.1 Blank detection

Liquidsoap has three operators for dealing with blanks.

On GeekRadio, we play many files, some of which include bonus tracks, which means that they end with a very long blank and then a little extra music. It's annoying to get that broadcasted. The `skip_blank` operator skips the current track when a too long blank is detected, which avoids that. The typical usage is simple:

```
# Wrap it with a blank skipper
source = skip_blank(source)
```

At RadioPi they have another problem: sometimes they have technical problems, and while they think they are doing a live show, they're making noise only in the studio, while only blank is broadcasted; sometimes, the staff has so much fun (or is it something else ?) doing live shows that they live at the end of the show without thinking to turn off the live, and the listeners get some silence again. To avoid that problem we made the `strip_blank` operators which hides the stream when it's too blank (i.e. declare it as unavailable), which perfectly suits the typical setup used for live shows:

```
interlude = single("/path/to/sorryfortheblank.ogg")
# After 5 sec of blank the microphone stream is ignored,
# which causes the stream to fallback to interlude.
# As soon as noise comes back to the microphone the stream comes
# back to the live -- thanks to track_sensitive=false.
stream = fallback(track_sensitive=false,
                      [ strip_blank(length=5.,live) , interlude ])

# Put that stream to a local file
output.file.ogg("/tmp/hop.ogg",stream)
```

If you don't get the difference between these two operators, maybe you need to learn more about the basic concepts of Liquidsoap, especially the notion of source.

Finally, if you need to do some custom action when there's too much blank, we have `on_blank`:

```
def handler()
  system("/path/to/your/script to do whathever you want")
end
source = on_blank(handler,source)
```

2.2 Distributed encoding

Using RTP, liquidsoap can directly output the raw stream with metadata. Then you can set up another liquidsoap instance on an other machine of your network which just inputs this RTP stream and encodes it, for example for sending to Icecast. It allows you to share the load on many machines, and also make the main liquidsoap process more independant of the Icecast servers. These can now crash or be restarted, you'll just have to restart the RTP encoders.

These operators are known to be quite buggy but may work depending on your needs.

Here is how to setup a RTP server:

```
set("log.file.path","/tmp/<script>.log"
set("log.stdout",true)

output.rtp(single("/usr/share/mrpingouin/mp3bis/bodom/TheNail.ogg"))
```

And here is the client, takes the RTP stream and plays it on your speakers:

```
set("log.file.path","/tmp/<script>.log"
set("log.stdout",true)

output.ao(input.rtp())
```

The server port has been specified on the server to be different from the default 1234 used on the client, so that thet don't conflict if ran on the same host. If you want to run the client on another host, specify a sufficient TTL for the RTP output, default being 0: `output.rtp(ttl=1,...)`.

Chapter 3

Other tools

3.1 Bubble

Bubble is a simple program which scans your audio files and stores their metadata in a SQLite database. It can rewrite paths into URI so that you can index remote files mounted locally and rewrite the local path into the general URI before storing it in the database. For example if you mount your samba workground in `/mnt/samba/workgroup` using fusesmb, you'll ask bubble to rewrite "/mnt/samba/workgroup" into "smb://".

Bubble has been designed to be interfaced with liquidsoap to provide a protocol for selecting files by queries on metadata. URI rewriting makes it possible to query from another machine than the one where the indexer runs, and also makes sure that the file will appear as a remote one to liquidsoap, so that it will be fully downloaded it before being played.

To add the bubble protocol to liquidsoap, we use the following code:

```
bubble = "/home/dbaelde/savonet/bubble/src/bubble-query " ^
             "-d /var/local/cache/bubble/bubble.sql "
add_protocol("bubble",
                 fun (arg,delay) -> get_process_lines(bubble^quote(arg)))
```

It allows us to have an IRC bot which accepts queries like `play "Alabama song"` and transforms it into the URI `bubble:title="Alabama song"` before queueing it in a liquidsoap instance. The `bubble` protocol in liquidsoap will call the `bubble-query` script which will transform the query into a SQLite query and return a list of ten random matches, which liquidsoap will try.

Although it has been used for months as distributed on the SVN, bubble is also a proof-of-concept tool. It is very concise and can be tailored to custom needs.

3.2 Bottle

Bottle is a prototype IRC bot written in OCaml. It uses a modular plugin system and is in particular able to communicate with liquidsoap. Currently, it is able to:

- show information about the song currently playing,
- listen to users' song requests,
- skip songs.

It is an example of how to write software which interacts with liquidsoap. You can get its source code via SVN: `http://svn.sourceforge.net/savonet/trunk/bottle/`.

This kind of tools doesn't need to be done in OCaml. It is quite easy to write an interface module for liquidsoap in perl, python or ruby too. We have a perl module in an unpublished bot – available on request. A python module is available in liguidsoap.

Chapter 4

Reference

4.1 Source / Input

4.1.1 blank

`(?id:string, ?duration:float)->source`
Produce silence.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `0.`): Duration of blank tracks in seconds, default means forever.

4.1.2 input.alsa

`(?id:string, ?bufferize:bool, ?device:string)->source`
Stream from an ALSA input device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `bufferize` (`bool` — defaults to `true`): Bufferize input.
- `device` (`string` — defaults to `"default"`): Alsa device to use.

4.1.3 input.harbor

`(?id:string, ?buffer:float, ?max:float, ?on_connect:(()->unit), ?on_disconnect:(()->unit), string)->source`
Retrieves the given http stream from the harbor.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `buffer` (`float` — defaults to `2.`): Duration of the pre-buffered data.
- `max` (`float` — defaults to `10.`): Maximum duration of the buffered data.
- `on_connect` (`()->unit` — defaults to `()`): Functions to excecute when a source is connected
- `on_disconnect` (`()->unit` — defaults to `()`): Functions to excecute when a source is disconnected
- `(unlabeled)` (`string`): Mountpoint to look for.

4.1.4 input.http

`(?id:string, ?autostart:bool, ?buffer:float, ?timeout:float, ?playlist_mode:string, ?max:float, string)->source`

Forwards the given http stream. The relay can be paused/resumed using the start/stop telnet commands.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `autostart` (`bool` — defaults to `true`): Initially start relaying or not.
- `buffer` (`float` — defaults to `2.`): Duration of the pre-buffered data.
- `timeout` (`float` — defaults to `2.`): Timeout for http connection.
- `playlist_mode` (`string` — defaults to `"normal"`): normal—random—randomize—first
- `max` (`float` — defaults to `10.`): Maximum duration of the buffered data.
- `(unlabeled)` (`string`): URL of an http stream (default port is 80).

4.1.5 input.lastfm

`(?id:string, ?autostart:bool, ?buffer:float, ?max:float, string)->source`

Forwards the given lastfm stream. The relay can be paused/resumed using the start/stop telnet commands.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `autostart` (`bool` — defaults to `true`): Initially start relaying or not.
- `buffer` (`float` — defaults to `2.`): Duration of the pre-buffered data.
- `max` (`float` — defaults to `10.`): Maximum duration of the buffered data.
- `(unlabeled)` (`string`): URI of a lastfm stream (e.g. lastfm://user/toots5446/playlist).

4.1.6 input.oss

`(?id:string, ?device:string)->source`

Stream from an OSS input device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `device` (`string` — defaults to `"/dev/dsp"`): OSS device to use.

4.1.7 input.portaudio

`(?id:string, ?buflen:int)->source`

Stream from a portaudio input device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `buflen` (`int` — defaults to `256`): Length of a buffer in samples.

4.1.8 noise

`(?id:string, ?duration:float)->source`

Generate white noise.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `0.`)

4.1.9 playlist

`(?id:string, ?length:float, ?default_duration:float, ?timeout:float, ?mode:string, ?reload:int, ?reload_mode:string, ?timeout:float, string)->source`

Loop on a playlist of URIs.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.
- `mode` (`string` — defaults to `"randomize"`): normal—random—randomize
- `reload` (`int` — defaults to `0`): Amount of time (in seconds or rounds) before which the playlist is reloaded; 0 means never.
- `reload_mode` (`string` — defaults to `"seconds"`): rounds—seconds: unit of the 'reload' parameter.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.
- `(unlabeled)` (`string`): URI where to find the playlist.

4.1.10 playlist.safe

`(?id:string, ?mode:string, ?reload:int, ?reload_mode:string, ?timeout:float, string)->source`

Loop on a playlist of local files, and never fail. In order to do so, it has to check every file at the loading, so the streamer startup may take a few seconds. To avoid this, use a standard playlist, and put only a few local files in a default safe_playlist in order to ensure the liveness of the streamer.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `mode` (`string` — defaults to `"randomize"`): normal—random—randomize
- `reload` (`int` — defaults to `0`): Amount of time (in seconds or rounds) before which the playlist is reloaded; 0 means never.
- `reload_mode` (`string` — defaults to `"seconds"`): rounds—seconds: unit of the 'reload' parameter.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.
- `(unlabeled)` (`string`): URI where to find the playlist.

4.1.11 request.dynamic

`(?id:string, (()->request), ?length:float, ?default_duration:float, ?timeout:float)->source`

Play request dynamically created by a given function.

- `id` (`string` — defaults to `""`): Force the value of the source ID.

- `(unlabeled)` (`()->request`): A function generating requests: an initial URI (possibly fake) together with an initial list of alternative indicators.
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.1.12 request.equeue

`(?id:string, ?length:float, ?default_duration:float, ?timeout:float)->source`
Receive URIs from users, and play them. Insertion and deletion possible at any position.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.1.13 request.queue

`(?id:string, ?queue:[request], ?interactive:bool, ?length:float, ?default_duration:float, ?timeout:float)->source`
Receive URIs from users, and play them.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `queue` (`[request]` — defaults to `[ ]`): Initial queue of requests.
- `interactive` (`bool` — defaults to `true`): Should the queue be controllable via telnet?
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.1.14 saw

`(?id:string, ?duration:float, ?float)->source`
Generate a saw wave.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `0.`)
- `(unlabeled)` (`float` — defaults to `440.`): Frequency of the saw.

4.1.15 sine

`(?id:string, ?duration:float, ?float)->source`
Generate a sine wave.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `0.`)
- `(unlabeled)` (`float` — defaults to `440.`): Frequency of the sine.

4.1.16 single

`(?id:string, string, ?length:float, ?default_duration:float, ?timeout:float)->source`
Loop on a request. It never fails if the request is static, meaning that it can be fetched once. Typically, http, ftp, say requests are static, and time is not.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`string`): URI where to find the file
- `length` (`float` — defaults to `60.`): How much audio (in sec.) should be downloaded in advance.
- `default_duration` (`float` — defaults to `30.`): When unknown, assume this duration (in sec.) for files.
- `timeout` (`float` — defaults to `20.`): Timeout (in sec.) for a single download.

4.1.17 square

`(?id:string, ?duration:float, ?float)->source`
Generate a square wave.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `0.`)
- `(unlabeled)` (`float` — defaults to `440.`): Frequency of the square.

4.2 Source / Output

4.2.1 output.alsa

`(?id:string, ?bufferize:bool, ?device:string, source)->source`
Output the source's stream to an ALSA output device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `bufferize` (`bool` — defaults to `true`): Bufferize output
- `device` (`string` — defaults to `"default"`): Alsa device to use
- `(unlabeled)` (`source`)

4.2.2 output.ao

`(?id:string, ?start:bool, ?driver:string, ?options:[(string*string)], source)->source`

Output stream to local sound card using libao.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `start` (`bool` — defaults to `true`): Start output on operator initialization.
- `driver` (`string` — defaults to `""`): libao driver to use.
- `options` (`[(string*string)]` — defaults to `[ ]`): List of parameters, depends on driver.
- `(unlabeled)` (`source`)

4.2.3 output.dummy

`(?id:string, source)->source`

Dummy output for debugging purposes.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`source`)

4.2.4 output.file.vorbis

`(?id:string, ?samplerate:int, ?stereo:bool, ?append:bool, ?perm:int, ?dir_perm:int, ?reopen_delay:float, ?reopen_on_metadata:bool, ?reopen_when:(()->bool), string, ?start:bool, ?quality:float, source)->source`

Output the source stream as an Ogg Vorbis file in Variable BitRate mode.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `samplerate` (`int` — defaults to `44100`)
- `stereo` (`bool` — defaults to `true`)
- `append` (`bool` — defaults to `false`): Do not truncate but append in the file if it exists.
- `perm` (`int` — defaults to `438`): Permission of the file if it has to be created, up to umask.
- `dir_perm` (`int` — defaults to `511`): Permission of the directories if some have to be created, up to umask.
- `reopen_delay` (`float` — defaults to `120.`): Prevent re-opening of the file within that delay, in seconds.
- `reopen_on_metadata` (`bool` — defaults to `false`): Re-open on every new metadata information.
- `reopen_when` (`()->bool` — defaults to `false`): When should the output file be re-opened.
- `(unlabeled)` (`string`): Filename where to output the stream. Some strftime conversion specifiers are available: %SMHdmY. You can also use $(..) interpolation notation for metadata.
- `start` (`bool` — defaults to `true`): Start output on operator initialization.
- `quality` (`float` — defaults to `2.`): Desired quality level, currently from -1. to 10. (low to high).
- `(unlabeled)` (`source`)

4.2.5 output.file.vorbis.abr

```
(?id:string, ?samplerate:int, ?stereo:bool, ?append:bool, ?perm:int,
?dir_perm:int, ?reopen_delay:float, ?reopen_on_metadata:bool,
?reopen_when:(()->bool), string, ?start:bool, ?bitrate:int, ?min_bitrate:int,
?max_bitrate:int, source)->source
```

Output the source stream as an Ogg Vorbis file in Average BitRate mode.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `samplerate` (`int` — defaults to `44100`)
- `stereo` (`bool` — defaults to `true`)
- `append` (`bool` — defaults to `false`): Do not truncate but append in the file if it exists.
- `perm` (`int` — defaults to `438`): Permission of the file if it has to be created, up to umask.
- `dir_perm` (`int` — defaults to `511`): Permission of the directories if some have to be created, up to umask.
- `reopen_delay` (`float` — defaults to `120.`): Prevent re-opening of the file within that delay, in seconds.
- `reopen_on_metadata` (`bool` — defaults to `false`): Re-open on every new metadata information.
- `reopen_when` (`()->bool` — defaults to `false`): When should the output file be re-opened.
- `(unlabeled)` (`string`): Filename where to output the stream. Some strftime conversion specifiers are available: %SMHdmY. You can also use $(..) interpolation notation for metadata.
- `start` (`bool` — defaults to `true`): Start output on operator initialization.
- `bitrate` (`int` — defaults to `128`): Target bitrate (in kbps).
- `min_bitrate` (`int` — defaults to `118`): Minimum bitrate (in kbps).
- `max_bitrate` (`int` — defaults to `138`): Maximum bitrate (in kbps).
- `(unlabeled)` (`source`)

4.2.6 output.file.vorbis.cbr

```
(?id:string, ?samplerate:int, ?stereo:bool, ?append:bool, ?perm:int,
?dir_perm:int, ?reopen_delay:float, ?reopen_on_metadata:bool,
?reopen_when:(()->bool), string, ?start:bool, ?bitrate:int, source)->source
```

Output the source stream as an Ogg Vorbis file in Constant BitRate mode.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `samplerate` (`int` — defaults to `44100`)
- `stereo` (`bool` — defaults to `true`)
- `append` (`bool` — defaults to `false`): Do not truncate but append in the file if it exists.
- `perm` (`int` — defaults to `438`): Permission of the file if it has to be created, up to umask.
- `dir_perm` (`int` — defaults to `511`): Permission of the directories if some have to be created, up to umask.

- `reopen_delay` (`float` — defaults to `120.`): Prevent re-opening of the file within that delay, in seconds.
- `reopen_on_metadata` (`bool` — defaults to `false`): Re-open on every new metadata information.
- `reopen_when` (`()->bool` — defaults to `false`): When should the output file be re-opened.
- `(unlabeled)` (`string`): Filename where to output the stream. Some strftime conversion specifiers are available: %SMHdmY. You can also use $(..) interpolation notation for metadata.
- `start` (`bool` — defaults to `true`): Start output on operator initialization.
- `bitrate` (`int` — defaults to `128`): Bitrate (in kbps).
- `(unlabeled)` (`source`)

4.2.7 output.file.wav

`(?id:string, ?start:bool, string, source)->source`

Output the source's stream to a WAV file.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `start` (`bool` — defaults to `true`)
- `(unlabeled)` (`string`)
- `(unlabeled)` (`source`)

4.2.8 output.icecast.vorbis

`(?id:string, ?samplerate:int, ?stereo:bool, ?start:bool, ?restart:bool, ?restart_delay:int, ?host:string, ?port:int, ?user:string, ?password:string, ?genre:string, ?url:string, ?description:string, ?public:bool, ?multicast_ip:string, ?sync:bool, ?mount:string, ?name:string, source, ?quality:float)->source`

Output the source stream as an Ogg Vorbis stream to an Icecast-compatible server in Variable BitRate mode.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `samplerate` (`int` — defaults to `44100`)
- `stereo` (`bool` — defaults to `true`)
- `start` (`bool` — defaults to `true`): Start output threads on operator initialization.
- `restart` (`bool` — defaults to `false`): Restart output after a failure. By default, liquidsoap will stop if the output failed.
- `restart_delay` (`int` — defaults to `3`): Delay, in seconds, before attempting new connection, if restart is enabled.
- `host` (`string` — defaults to `"localhost"`)
- `port` (`int` — defaults to `8000`)
- `user` (`string` — defaults to `"source"`)

- password (string — defaults to "hackme")
- genre (string — defaults to "Misc")
- url (string — defaults to "http://savonet.sf.net")
- description (string — defaults to "OCaml Radio!")
- public (bool — defaults to true)
- multicast_ip (string — defaults to "no_multicast")
- sync (bool — defaults to false): let shout do the synchronization
- mount (string — defaults to "Use [name].ogg")
- name (string — defaults to "Use [mount]")
- (unlabeled) (source)
- quality (float — defaults to 2.): Desired quality level, currently from -1. to 10. (low to high).

4.2.9 output.icecast.vorbis.abr

```
(?id:string, ?samplerate:int, ?stereo:bool, ?start:bool, ?restart:bool,
?restart_delay:int, ?host:string, ?port:int, ?user:string, ?password:string,
?genre:string, ?url:string, ?description:string, ?public:bool,
?multicast_ip:string, ?sync:bool, ?mount:string, ?name:string, source,
?bitrate:int, ?min_bitrate:int, ?max_bitrate:int)->source
```

Output the source stream as an Ogg Vorbis stream to an Icecast-compatible server in Average BitRate mode.

- id (string — defaults to ""): Force the value of the source ID.
- samplerate (int — defaults to 44100)
- stereo (bool — defaults to true)
- start (bool — defaults to true): Start output threads on operator initialization.
- restart (bool — defaults to false): Restart output after a failure. By default, liquidsoap will stop if the output failed.
- restart_delay (int — defaults to 3): Delay, in seconds, before attempting new connection, if restart is enabled.
- host (string — defaults to "localhost")
- port (int — defaults to 8000)
- user (string — defaults to "source")
- password (string — defaults to "hackme")
- genre (string — defaults to "Misc")
- url (string — defaults to "http://savonet.sf.net")
- description (string — defaults to "OCaml Radio!")
- public (bool — defaults to true)

- multicast_ip (string — defaults to "no_multicast")
- sync (bool — defaults to false): let shout do the synchronization
- mount (string — defaults to "Use [name].ogg")
- name (string — defaults to "Use [mount]")
- (unlabeled) (source)
- bitrate (int — defaults to 128): Target bitrate (in kbps).
- min_bitrate (int — defaults to 118): Minimum bitrate (in kbps).
- max_bitrate (int — defaults to 138): Maximum bitrate (in kbps).

4.2.10 output.icecast.vorbis.cbr

```
(?id:string, ?samplerate:int, ?stereo:bool, ?start:bool, ?restart:bool,
?restart_delay:int, ?host:string, ?port:int, ?user:string, ?password:string,
?genre:string, ?url:string, ?description:string, ?public:bool,
?multicast_ip:string, ?sync:bool, ?mount:string, ?name:string, source,
?bitrate:int)->source
```

Output the source stream as an Ogg Vorbis stream to an Icecast-compatible server in Constant BitRate mode.

- id (string — defaults to ""): Force the value of the source ID.
- samplerate (int — defaults to 44100)
- stereo (bool — defaults to true)
- start (bool — defaults to true): Start output threads on operator initialization.
- restart (bool — defaults to false): Restart output after a failure. By default, liquidsoap will stop if the output failed.
- restart_delay (int — defaults to 3): Delay, in seconds, before attempting new connection, if restart is enabled.
- host (string — defaults to "localhost")
- port (int — defaults to 8000)
- user (string — defaults to "source")
- password (string — defaults to "hackme")
- genre (string — defaults to "Misc")
- url (string — defaults to "http://savonet.sf.net")
- description (string — defaults to "OCaml Radio!")
- public (bool — defaults to true)
- multicast_ip (string — defaults to "no_multicast")
- sync (bool — defaults to false): let shout do the synchronization
- mount (string — defaults to "Use [name].ogg")
- name (string — defaults to "Use [mount]")
- (unlabeled) (source)
- bitrate (int — defaults to 128): Bitrate (in kbps).

4.2.11 output.oss

`(?id:string, ?device:string, source)->source`
Output the source's stream to an OSS output device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `device` (`string` — defaults to `"/dev/dsp"`): OSS device to use.
- `(unlabeled)` (`source`)

4.2.12 output.portaudio

`(?id:string, ?buflen:int, source)->source`
Output the source's stream to a portaudio output device.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `buflen` (`int` — defaults to `256`): Length of a buffer in samples.
- `(unlabeled)` (`source`)

4.3 Source / Sound Processing

4.3.1 accelerate

`(?id:string, float, ?before:float, ?after:float, source)->source`
Accelerates a stream, possibly only the middle of the tracks. Useful for testing transitions.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`float`)
- `before` (`float` — defaults to `10.`): Do not accelerate during the first ¡before¿ seconds.
- `after` (`float` — defaults to `10.`): Do not accelerate during the last ¡after¿ seconds.
- `(unlabeled)` (`source`)

4.3.2 add

`(?id:string, ?normalize:bool, ?weights:[int], [source])->source`
Mix sources, with optional normalization. Only relay metadata from the first source that is effectively summed.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `normalize` (`bool` — defaults to `true`)
- `weights` (`[int]` — defaults to `[ ]`): Relative weight of the sources in the sum. The empty list stands for the homogeneous distribution.
- `(unlabeled)` (`[source]`)

4.3.3 amplify

`(?id:string, 'a, source)->source where 'a is either float or ()->float`
Multiply the amplitude of the signal.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`anything that is either float or ()->float`): Multiplicative factor.
- `(unlabeled)` (`source`)

4.3.4 bpm

`(?id:string, ?every:int, source)->source`
WARNING: This is only EXPERIMENTAL!
Detect the BPM.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `every` (`int` — defaults to `500`)
- `(unlabeled)` (`source`)

4.3.5 clip

`(?id:string, ?min:float, ?max:float, source)->source`
Clip sound.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `min` (`float` — defaults to `-0.999`): Minimal acceptable value.
- `max` (`float` — defaults to `0.999`): Maximal acceptable value.
- `(unlabeled)` (`source`)

4.3.6 comb

`(?id:string, ?delay:float, ?feedback:'a, source)->source where 'a is either float or ()->float`
Comb filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `delay` (`float` — defaults to `0.001`): Delay in seconds.
- `feedback` (`anything that is either float or ()->float` — defaults to `-6.`): Feedback coefficient in dB.
- `(unlabeled)` (`source`)

4.3.7 compand

`(?id:string, ?mu:float, source)->source`
Compand the signal

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `mu` (`float` — defaults to `1.`)
- `(unlabeled)` (`source`)

4.3.8 compress

`(?id:string, ?ratio:float, ?attack:'a, ?release:'b, ?threshold:'c, ?knee:'d, ?rms_window:float, ?gain:'e, ?debug:bool, source)->source where 'a/'b/'c/'d/'e is either float or ()->float`

Compress the signal.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `ratio` (`float` — defaults to `2.`): Gain reduction ratio (n:1).
- `attack` (`anything that is either float or ()->float` — defaults to `100.`): Attack time (ms).
- `release` (`anything that is either float or ()->float` — defaults to `400.`): Release time (ms).
- `threshold` (`anything that is either float or ()->float` — defaults to `-10.`): Threshold level (dB).
- `knee` (`anything that is either float or ()->float` — defaults to `1.`): Knee radius (dB).
- `rms_window` (`float` — defaults to `0.1`): Window for computing RMS (in sec).
- `gain` (`anything that is either float or ()->float` — defaults to `0.`): Additional gain (dB).
- `debug` (`bool` — defaults to `false`)
- `(unlabeled)` (`source`)

4.3.9 compress.exponential

`(?id:string, ?mu:float, source)->source`

Exponential compressor.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `mu` (`float` — defaults to `2.`): Exponential compression factor (typically ¿ 1).
- `(unlabeled)` (`source`)

4.3.10 cross

`(?id:string, ?duration:float, ?inhibit:float, ?minimum:float, ((source, source)->source), source)->source`

Generic cross operator, allowing the composition of the N last seconds of a track with the beginning of the next track.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `5.`): Duration in seconds of the crossed end of track. This value can be set on a per-file basis using the metadata field 'liq_start_next.
- `inhibit` (`float` — defaults to `-1.`): Minimum delay between two transitions. It is useful in order to avoid that a transition is triggered on top of another when an end-of-track occurs in the first one. Negative values mean 'same as duration'.

- `minimum` (`float` — defaults to `-1.`): Minimum duration (in sec.) for a cross: If the track ends without any warning (e.g. in case of skip) there may not be enough data for a decent composition. Set to 0. to avoid having transitions after skips, or more to avoid transitions on short tracks. With the negative default, transitions always occur.
- `(unlabeled)` (`(source, source)->source`): Composition of an end of track and the next track.
- `(unlabeled)` (`source`)

4.3.11 echo

`(?id:string, ?delay:float, ?feedback:'a, source)->source where 'a is either float or ()->float`

Add echo.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `delay` (`float` — defaults to `0.5`): Delay in seconds.
- `feedback` (`anything that is either float or ()->float` — defaults to `-6.`): Feedback coefficient in dB (¡= 0).
- `(unlabeled)` (`source`)

4.3.12 fade.final

`(?id:string, ?duration:float, ?type:string, source)->source`

Fade a stream to silence.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `3.`)
- `(unlabeled)` (`source`)

4.3.13 fade.in

`(?id:string, ?duration:float, ?type:string, source)->source`

Fade the beginning of tracks. Metadata 'liq_fade_in' can be used to set the duration for a specific track (float in seconds).

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `3.`)
- `(unlabeled)` (`source`)

4.3.14 fade.initial

`(?id:string, ?duration:float, ?type:string, source)->source`

Fade the beginning of a stream.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `3.`)
- `(unlabeled)` (`source`)

4.3.15 fade.out

`(?id:string, ?duration:float, ?type:string, source)->source`

Fade the end of tracks. Metadata 'liq_fade_out' can be used to set the duration for a specific track (float in seconds).

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `3.`)
- `(unlabeled)` (`source`)

4.3.16 filter

`(?id:string, ~freq:'a, ?q:'b, ~mode:string, ?wetness:'c, source)->source where 'a/'b/'c is either float or ()->float`

Perform several kinds of filtering on the signal

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `freq` (`anything that is either float or ()->float`)
- `q` (`anything that is either float or ()->float` — defaults to `1.`)
- `mode` (`string`): low—high—band—notch
- `wetness` (`anything that is either float or ()->float` — defaults to `1.`): How much of the original signal should be added (1. means only filtered and 0. means only original signal).
- `(unlabeled)` (`source`)

4.3.17 filter.fir

`(?id:string, ~frequency:float, ~beta:float, ?coeffs:int, ?debug:bool, source)->source`

Low-pass FIR filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency (frequency at which the response is 0.5 = -6 dB, Hz)
- `beta` (`float`): Beta (0 1)
- `coeffs` (`int` — defaults to `255`): Number of coefficients
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.18 filter.iir.butterworth.bandpass

`(?id:string, ~frequency1:float, ~frequency2:float, ?order:int, ?debug:bool, source)->source`

IIR filter

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency1` (`float`): First corner frequency

- `frequency2` (`float`): Second corner frequency
- `order` (`int` — defaults to `4`): Filter order
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.19 filter.iir.butterworth.bandstop

`(?id:string, ~frequency1:float, ~frequency2:float, ?order:int, ?debug:bool, source)->source`
IIR filter

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency1` (`float`): First corner frequency
- `frequency2` (`float`): Second corner frequency
- `order` (`int` — defaults to `4`): Filter order
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.20 filter.iir.butterworth.high

`(?id:string, ~frequency:float, ?order:int, ?debug:bool, source)->source`
IIR filter

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `order` (`int` — defaults to `4`): Filter order
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.21 filter.iir.butterworth.low

`(?id:string, ~frequency:float, ?order:int, ?debug:bool, source)->source`
IIR filter

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `order` (`int` — defaults to `4`): Filter order
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.22 filter.iir.eq.allpass

`(?id:string, ~frequency:float, ?bandwidth:float, ?debug:bool, source)->source`
All pass biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Center frequency
- `bandwidth` (`float` — defaults to `1.`): Bandwidth (in octaves)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.23 filter.iir.eq.bandpass

`(?id:string, ~frequency:float, ?bandwidth:float, ?debug:bool, source)->source`
Band pass biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Center frequency
- `bandwidth` (`float` — defaults to `1.`): Bandwidth (in octaves)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.24 filter.iir.eq.high

`(?id:string, ~frequency:float, ?q:float, ?debug:bool, source)->source`
High pass biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `q` (`float` — defaults to `1.`): Q
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.25 filter.iir.eq.highshelf

`(?id:string, ~frequency:float, ?slope:float, ?debug:bool, source)->source`
High shelf biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Center frequency
- `slope` (`float` — defaults to `1.`): Shelf slope (in dB/octave)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.26 filter.iir.eq.low

`(?id:string, ~frequency:float, ?q:float, ?debug:bool, source)->source`
Low pass biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `q` (`float` — defaults to `1.`): Q
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.27 filter.iir.eq.lowshelf

`(?id:string, ~frequency:float, ?slope:float, ?debug:bool, source)->source`
Low shelf biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `slope` (`float` — defaults to `1.`): Shelf slope (dB/octave)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.28 filter.iir.eq.notch

`(?id:string, ~frequency:float, ?bandwidth:float, ?debug:bool, source)->source`
Band pass biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Center frequency
- `bandwidth` (`float` — defaults to `0.333333333333`): Bandwidth (in octaves)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.29 filter.iir.eq.peak

`(?id:string, ~frequency:float, ?bandwidth:float, ?gain:float, ?debug:bool, source)->source`
Peak EQ biquad filter.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Center frequency
- `bandwidth` (`float` — defaults to `0.333333333333`): Bandwidth (in octaves)
- `gain` (`float` — defaults to `1.`): Gain (in dB)
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.30 filter.iir.resonator.bandpass

`(?id:string, ~frequency:float, ?q:float, ?debug:bool, source)->source`
IIR filter

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `frequency` (`float`): Corner frequency
- `q` (`float` — defaults to `60.`): Quality factor
- `debug` (`bool` — defaults to `false`): Debug output
- `(unlabeled)` (`source`)

4.3.31 flanger

`(?id:string, ?delay:float, ?freq:'a, ?feedback:'b, source)->source where 'a/'b is either float or ()->float`
Flanger effect.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `delay` (`float` — defaults to `0.001`): Delay in seconds.
- `freq` (`anything that is either float or ()->float` — defaults to `0.5`): Frequency in Hz.
- `feedback` (`anything that is either float or ()->float` — defaults to `0.`): Feedback coefficient in dB.
- `(unlabeled)` (`source`)

4.3.32 insert_metadata

`(?id:string, source)->source`
Interactively insert metadata using the command ¡id¿.insert key1=”val1”,key2=”val2”,...

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`source`)

4.3.33 limit

`(?id:string, ?ratio:float, ?attack:'a, ?release:'b, ?threshold:'c, ?knee:'d, ?rms_window:float, ?gain:'e, ?debug:bool, source)->source where 'a/'b/'c/'d/'e is either float or ()->float`
Limit the signal.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `ratio` (`float` — defaults to `20.`): Gain reduction ratio (n:1).
- `attack` (`anything that is either float or ()->float` — defaults to `100.`): Attack time (ms).
- `release` (`anything that is either float or ()->float` — defaults to `400.`): Release time (ms).

- `threshold` (`anything that is either float or ()->float` — defaults to `-10.`): Threshold level (dB).
- `knee` (`anything that is either float or ()->float` — defaults to `1.`): Knee radius (dB).
- `rms_window` (`float` — defaults to `0.1`): Window for computing RMS (in sec).
- `gain` (`anything that is either float or ()->float` — defaults to `0.`): Additional gain (dB).
- `debug` (`bool` — defaults to `false`)
- `(unlabeled)` (`source`)

4.3.34 mean

`(?id:string, ?channels:[int], source)->source`

Compute the mean of a list of audio channels and use it for all of them.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `channels` (`[int]` — defaults to `[ 0; 1 ]`): List of channels to compute the means.
- `(unlabeled)` (`source`)

4.3.35 mix

`(?id:string, [source])->source`

Mixing table controllable via the telnet interface.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`[source]`)

4.3.36 normalize

`(?id:string, ?target:'a, ?window:float, ?k_up:'b, ?k_down:'c, ?threshold:'d, ?gain_min:'e, ?gain_max:'f, ?debug:bool, source)->source where 'a/'b/'c/'d/'e/'f is either float or ()->float`

Normalize the signal

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `target` (`anything that is either float or ()->float` — defaults to `-13.`): Desired RMS (dB).
- `window` (`float` — defaults to `0.1`): Duration of the window used to compute the current RMS power (second).
- `k_up` (`anything that is either float or ()->float` — defaults to `0.005`): Coefficient when the power must go up (0 ¡ k_up ¡1, slowest to fastest).
- `k_down` (`anything that is either float or ()->float` — defaults to `0.1`): Coefficient when the power must go down (0 ¡ k_down ¡ 1, slowest to fastest).
- `threshold` (`anything that is either float or ()->float` — defaults to `-40.`): Minimal RMS for activaing gain control (dB).

- `gain_min` (`anything that is either float or ()->float` — defaults to `-6.`): Minimal gain value (dB).
- `gain_max` (`anything that is either float or ()->float` — defaults to `6.`): Maximal gain value (dB).
- `debug` (`bool` — defaults to `false`): Show coefficients.
- `(unlabeled)` (`source`)

4.3.37 pan

`(?id:string, ?pan:'a, ?field:'b, source)->source where 'a/'b is either float or ()->float`

swap two channels

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `pan` (`anything that is either float or ()->float` — defaults to `0.`): Pan (-1 pan 1)
- `field` (`anything that is either float or ()->float` — defaults to `90.`): Field width (0 ¡ field ¡ 90) (in degrees)
- `(unlabeled)` (`source`)

4.3.38 smart_cross

`(?id:string, ?duration:float, ?inhibit:float, ?minimum:float, ?width:float, ?conservative:bool, ((float, float, [(string*string)], [(string*string)], source, source)->source), source)->source`

Cross operator, allowing the composition of the N last seconds of a track with the beginning of the next track, using a transition function depending on the relative power of the signal before and after the end of track.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `duration` (`float` — defaults to `5.`): Duration in seconds of the crossed end of track.
- `inhibit` (`float` — defaults to `-1.`): Minimum delay between two transitions. It is useful in order to avoid that a transition is triggered on top of another when an end-of-track occurs in the first one. Negative values mean 'same as duration'. Warning: zero inhibition can cause infinite loops.
- `minimum` (`float` — defaults to `-1.`): Minimum duration (in sec.) for a cross: If the track ends without any warning (e.g. in case of skip) there may not be enough data for a decent composition. Set to 0. to avoid having transitions after skips, or more to avoid transitions on short tracks. With the negative default, transitions always occur.
- `width` (`float` — defaults to `1.`): Width of the power computation window.
- `conservative` (`bool` — defaults to `false`): Do not trust remaining time estimations, always buffering data in advance. This avoids being tricked by skips, either manual or caused by skip_blank().
- `(unlabeled)` (`((float, float, [(string*string)], [(string*string)], source, source)->source`): Transition function, composing from the end of a track and the next track. It also takes the power of the signal before and after the transition, and the metadata.
- `(unlabeled)` (`source`)

4.3.39 soundtouch

`(?id:string, ?rate:'a, ?tempo:'b, ?pitch:'c, source)->source where 'a/'b/'c is either float or ()->float`

WARNING: This is only EXPERIMENTAL!
Change the rate, the tempo or the pitch of the sound.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `rate` (`anything that is either float or ()->float` — defaults to `1.`)
- `tempo` (`anything that is either float or ()->float` — defaults to `1.`)
- `pitch` (`anything that is either float or ()->float` — defaults to `1.`)
- `(unlabeled)` (`source`)

4.3.40 swap

`(?id:string, ?chan1:int, ?chan2:int, source)->source`

swap two channels

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `chan1` (`int` — defaults to `0`): Channel one
- `chan2` (`int` — defaults to `1`): Channel two
- `(unlabeled)` (`source`)

4.4 Source / Track Processing

4.4.1 append

`(?id:string, ?merge:bool, source, (([(string*string)])->source))->source`

Append an extra track to every track.Set the metadata 'liq_append' to 'false' to inhibit appending on one track.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `merge` (`bool` — defaults to `false`): Merge the track with its appended track.
- `(unlabeled)` (`source`)
- `(unlabeled)` (`(([(string*string)])->source`): Given the metadata, build the source producing the track to append. This source is allowed to fail (produce nothing) if no relevant track is to be appended.

4.4.2 delay

`(?id:string, float, source)->source`

Prevents the child from being ready again too fast after a end of track

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`float`): The source won't be ready less than this amount of seconds after any end of track
- `(unlabeled)` (`source`)

4.4.3 eat_blank

`(?id:string, ?at_beginning:bool, ?threshold:float, ?length:float, source)->source`
Eat blanks (i.e. drop the contents of the stream until it is not blank again).

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `at_beginning` (`bool` — defaults to `false`): Only eat at the beginning of a track.
- `threshold` (`float` — defaults to `-40.`): Power in decibels under which the stream is considered silent.
- `length` (`float` — defaults to `20.`): Maximum silence length allowed, in seconds.
- `(unlabeled)` (`source`)

4.4.4 fallback

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source)->source], [source])->source`
At the beginning of each track, select the first ready child.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks.
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `transitions` (`[(source, source)->source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.
- `(unlabeled)` (`[source]`): Select the first ready source in this list.

4.4.5 on_blank

`(?id:string, (()->unit), ?threshold:float, ?length:float, source)->source`
Calls a given handler when detecting a blank.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`()->unit`)
- `threshold` (`float` — defaults to `-40.`): Power in decibels under which the stream is considered silent.
- `length` (`float` — defaults to `20.`): Maximum silence length allowed, in seconds.
- `(unlabeled)` (`source`)

4.4.6 on_metadata

`(?id:string, (([(string*string)])->unit), source)->source`
Call a given handler on metadata packets.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`([(string*string)])->unit`): Function called on every metadata packet in the stream. It should be fast because it is ran in the main thread.
- `(unlabeled)` (`source`)

4.4.7 on_track

`(?id:string, (([(string*string)])->unit), source)->source`

Call a given handler on new tracks.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`([(string*string)])->unit`): Function called on every beginning of track in the stream, with the corresponding metadata as argument. It should be fast because it is ran in the main thread.
- `(unlabeled)` (`source`)

4.4.8 prepend

`(?id:string, ?merge:bool, source, (([(string*string)])->source))->source`

Prepend an extra track before every track. Set the metadata 'liq_prepend' to 'false' to inhibit prepending on one track.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `merge` (`bool` — defaults to `false`): Merge the track with its appended track.
- `(unlabeled)` (`source`)
- `(unlabeled)` (`([(string*string)])->source`): Given the metadata, build the source producing the track to prepend. This source is allowed to fail (produce nothing) if no relevant track is to be appended. However, success must be immediate.

4.4.9 random

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source)->source], ?weights:[int], ?strict:bool, [source])->source`

At the beginning of every track, select a random ready child.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks.
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `transitions` (`[(source, source)->source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.
- `weights` (`[int]` — defaults to `[ ]`): Weights of the children in the choice.
- `strict` (`bool` — defaults to `false`): Do not use random but cycle over the uniform distribution.
- `(unlabeled)` (`[source]`)

4.4.10 rewrite_metadata

`(?id:string, [(string*string)], ?insert_missing:bool, source)->source`
Rewrite metadata on the fly.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `(unlabeled)` (`[(string*string)]`): List of (target,value) rewriting rules.
- `insert_missing` (`bool` — defaults to `true`): Treat track beginnings without metadata as having empty ones.
- `(unlabeled)` (`source`)

4.4.11 sequence

`(?id:string, ?merge:bool, [source])->source`
Play only one track of every successive source, except for the last one which is played as much as available.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `merge` (`bool` — defaults to `false`)
- `(unlabeled)` (`[source]`)

4.4.12 skip_blank

`(?id:string, ?threshold:float, ?length:float, source)->source`
Skip track when detecting a blank.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `threshold` (`float` — defaults to `-40.`): Power in decibels under which the stream is considered silent.
- `length` (`float` — defaults to `20.`): Maximum silence length allowed, in seconds.
- `(unlabeled)` (`source`)

4.4.13 store_metadata

`(?id:string, ?size:int, source)->source`
Keep track of the last N metadata packets in the stream, and make the history available via a server command.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `size` (`int` — defaults to `10`): Size of the history
- `(unlabeled)` (`source`)

4.4.14 strip_blank

`(?id:string, ?threshold:float, ?length:float, source)->source`
Make the source unavailable when it is streaming blank.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `threshold` (`float` — defaults to `-40.`): Power in decibels under which the stream is considered silent.
- `length` (`float` — defaults to `20.`): Maximum silence length allowed, in seconds.
- `(unlabeled)` (`source`)

4.4.15 switch

`(?id:string, ?track_sensitive:bool, ?before:float, ?transitions:[(source, source)->source], ?single:[bool], [(()->bool)*source)])->source`
At the beginning of a track, select the first source whose predicate is true.

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `track_sensitive` (`bool` — defaults to `true`): Re-select only on end of tracks.
- `before` (`float` — defaults to `0.`): EXPERIMENTAL: for track_sensitive switches, trigger transitions before the end of track.
- `transitions` (`[(source, source)->source]` — defaults to `[ ]`): Transition functions, padded with (fun (x,y) -¿ y) functions.
- `single` (`[bool]` — defaults to `[ ]`): Forbid the selection of a branch for two tracks in a row. The empty list stands for [false,. . . ,false].
- `(unlabeled)` (`[(()->bool)*source)]`): Sources with the predicate telling when they can be played.

4.5 Source / Visualization

4.5.1 vumeter

`(?id:string, ?scroll:bool, source)->source`
VU meter (display the volume).

- `id` (`string` — defaults to `""`): Force the value of the source ID.
- `scroll` (`bool` — defaults to `false`): Scroll.
- `(unlabeled)` (`source`)

4.6 Bool

4.6.1 ! =

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.2 <

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.3 <=

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.4 ==

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.5 >

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.6 >=

`('a, 'a)->bool where 'a is an orderable type`
Comparison of comparable values.

4.6.7 and

`(bool, bool)->bool`
Return the conjunction of its arguments

4.6.8 not

`(bool)->bool`
Returns the negation of its argument.

4.6.9 or

`(bool, bool)->bool`
Return the disjunction of its arguments

4.6.10 random.bool

`()->bool`
Generate a random value.

4.7 Control

4.7.1 add_timeout

`(?name:string, float, (()->float))->unit`
Call a function every N seconds. If the output of the function is a positive float it will be used as the new delay. The name of the created thread can be chosen.

4.7.2 ignore

`('a)->unit`
Convert anything to unit, preventing warnings.

4.8 Interaction

4.8.1 interactive_float

`(string, float)->()->float`
Read a float from an interactive input.

4.8.2 print

`(?newline:bool, 'a)->unit`
Print on standard output.

4.9 Liquidsoap

4.9.1 add_protocol

`(string, ((string, float)->[string]))->unit`
Register a new protocol.

4.9.2 get

`(~default:'a, string)->'a where 'a is bool, int, float, string or [string]`
Get a setting's value.

4.9.3 request

`(?indicators:[string], ?persistent:bool, string)->request`
Create a request.

4.9.4 set

`(string, 'a)->unit where 'a is bool, int, float, string or [string]`
Change some setting.

4.9.5 shutdown

`()->unit`
Shutdown the application.

4.9.6 source.id

`(source)->string`
Get source's id.

4.9.7 source.skip

`(source)->unit`
Skip source's current song.

4.10 List

4.10.1 _[_]

`(string, [(string*string)])->string`
l[k] returns v for the first item (k,v) in l.

4.10.2 list.fold

`((('a, 'b)->'a), 'a, ['b])->'a`
Fold a function on every element of a list.

4.10.3 list.hd

`([string])->string`
Returns the head (first element) of a list.

4.10.4 list.iter

`((('a)->unit), ['a])->unit`
Execute a function on every element of a list.

4.10.5 list.length

`(['a])->int`
Returns the length (number of elements) of a list.

4.10.6 list.map

`((('a)->'b), ['a])->['b]`
Map a function on every element of a list.

4.10.7 list.mem

`('a, ['a])->bool where 'a is an orderable type`
Checks if an element is present within a list.

4.10.8 list.nth

`(['a], int)->'a`
Returns the nth element of a list.

4.10.9 list.tl

`(['a])->['a]`
Returns the list without its first element.

4.11 Math

4.11.1 *

`('a, 'a)->'a where 'a is a number type`
Multiplication of numbers.

4.11.2 +

`('a, 'a)->'a where 'a is a number type`
Addition of numbers.

4.11.3 -

`('a, 'a)->'a where 'a is a number type`
Substraction of numbers.

4.11.4 /

`('a, 'a)->'a where 'a is a number type`
Division of numbers.

4.11.5 abs

`('a)->'a where 'a is a number type`
Absolute value.

4.11.6 bool_of_float

`(float)->bool`
Convert a float to a bool.

4.11.7 bool_of_int

`(int)->bool`
Convert an int to a bool.

4.11.8 dB_of_lin

`(float)->float`
Convert linear scale into decibels.

4.11.9 float_of_int

`(int)->float`
Convert an int to a float.

4.11.10 int_of_float

`(float)->int`
Convert a float to a int.

4.11.11 lin_of_dB

`(float)->float`
Convert decibels into linear scale.

4.11.12 pow

`('a, 'a)->'a where 'a is a number type`
Exponentiation of numbers.

4.11.13 random.float

`(?min:float, ?max:float)->float`
Generate a random value.

4.12 String

4.12.1 %

`(string, [(string*string)])->string`
(pattern % [...,(k,v),...]) replaces in pattern occurences of:
- '$(k)' into "v";
- '$(if $(k2),"a","b")' into "a" if k2 is found in the list, "b" otherwise.

4.12.2 ^

`(string, string)->string`
Concatenate strings.

4.12.3 bool_of_string

`(?default:bool, string)->bool`
Convert a string to a bool.

4.12.4 float_of_string

`(?default:float, string)->float`
Convert a string to a float.

4.12.5 int_of_string

`(?default:int, string)->int`
Convert a string to a int.

4.12.6 quote

`(string)->string`
Escape shell metacharacters.

4.12.7 string.concat

`(?separator:string, [string])->string`
Concatenate strings.

4.12.8 string.split

`(~separator:string, string)->[string]`
Split a string at 'separator'.

4.12.9 string_of

`('a)->string`
Convert a value into a string.

4.13 System

4.13.1 argv

`(?default:string, int)->string`
Get command-line parameters.

4.13.2 execute

`(string, ?string)->[string]`
Executes a command.

4.13.3 get_process_lines

`(string)->[string]`
Perform a shell call and return the list of its output lines.

4.13.4 get_process_output

`(string)->string`
Perform a shell call and return its output.

4.13.5 log

`(?label:string, ?level:int, string)->unit`
Log a message.

4.13.6 on_shutdown

`((()->unit))->unit`
Run a callback when Liquidsoap shuts down.

4.13.7 shutdown

`()->unit`
Terminates the whole liquidsoap process.

4.13.8 system

`(string)->unit`
Shell command call.