RoarAudio Manual

Philipp "ph3-der-loewe" Schafft et al.

August 13, 2010

Contents

Part I

Introduction

Chapter 1

What is RoarAudio?

RoarAudio is a very powerful cross-platform sound server. It is written for all POSIX systems like GNU/Linux and all BSDs. There are also some parts already ported to other architectures like cygwin and μ-Controllers.

It is designed to replace the very old sound server *Enlightened Sound Daemon* (*EsounD*) and add modern features as well as features needed by Radio and TV Stations. This includes support for Web Streaming via Icecast and simular software.

1.1 What is a Sound Server?

A Sound Server or Sound Daemon is an application mixing audio in realtime. It is normaly used to mix audio from diffrent applications befor sending it to the soundcard as most soundcards can only handle one audio stream at a time. Soundcards are called *Output* within RoarAudio. RoarAudio also knows other types of Outputs, too, like file dumps or streaming servers like Icecast.

1.2 What are the key fatures of RoarAudio?

1. fully network transparent. Network support for UNIX Domain Sockets, TCP/IP and DECnet
2. synchronized audio streams
3. multiple audio streams per client
4. Vorbis comments like meta data for each audio stream
5. support for EsounD clients via libroaresd
6. supported by many players: including all EsounD and libao players many more!
7. mixing clients at individual levels like an analog mixer

8. server and client side support for common codecs like Ogg Vorbis, Speex, FLAC and many more
9. and many more...

Chapter 2

Basic concept

Part II

Quickstart

Chapter 3

Installation

3.1 archlinux

RoarAudio is available to users of the Archlinux GNU/Linux distribution, through it's user repository, the AUR. There are a few ways to obtain it from there and installing it. First of all, it requiers you to have a set of development tools, since it's essentially building it from the sources, and makeing an Arch-compatible package of the resulting files.

To start using the AUR, you will need to make sure that you have the package "base-devel". The "base-devel" package, includes basic development tools, as well as the packaging tools we are going to use to build, make a package of, and install the RoarAudio software.

Additionally you will need packages for:

- libao - for supporting output via the ao-library

And, optionally you would like to have:

- vorbis-tools - for supporting the vorbis codec
- speex - for supporting the speex codec
- flac - for supporting the flac codec

The most common way is the use the makepkg-utility directly to perform all of the steps needed to produce a working package. It goes through a couple of steps, and does some basic tests, and then goes on to build. Below is a few simple steps to follow when using this method.

1. Go get the *PKGBUILD* from the AUR at this link: `http://aur.archlinux.org/packages.php?ID=23109`
2. Get the "tarball", and save it somewhere, for example in your /home or in the /tmp directories, then unpack it.

3. Change into the directory. There should be a PKGBUILD-file present. Run ”makepkg” in the directory with the PKGBUILD-file. The build and package-process will now start.

4. If everything went as it should, a package-file is generated, and you can install it to your system by using the standard package-tools. To install a package from a file instead of remotely, do ”pacman -U <package-name>”, and to remove it use ”pacman -R <package-name>”.

Another popular method of installing software from the AUR is to use a pacman-wrapper script, or pacman frontend. I will go through the steps needed when using the ”**yaourt**” frontend.

1. Run ”yaourt -S roaraudio”. It should automaticlly look through both the standard repositories, and when it comes up with nothing, it will continue on to search in the AUR, where ”roaraudio” is available.

2. Yoaurt will perform most steps automaticlly for you, but along the way, it will ask you questions. First, it will ask if you would like to edit the PKGBUILD. Press ”n” for no.

3. Next, Yaourt will ask you if you want to continue building the software. Press ”y” for yes.

4. And finally it will ask you if you want to install the package. Press ”y” to do so (requiers root-privileges, or a sudo-setup account where you are allowed to use pacman).

There! After following either method, you should be set up with a working and installed package. You should now be able to start the RoarAudio daemon by running:

```
# sh /etc/rc.d/roard start
```

To test if it is actully working, you can use the included tool roarvorbis like this:

```
$ roarvorbis  <vorbisfile.ogg>
```

It should play the file back. You can also see what is going on by running:

```
$ roarctl allinfo
```

3.2 OpenBSD

3.3 Debian

First you need to follow the steps from section From Sources.

After installing you may copy the init script and config file:

```
 # cp dist/debian-like/roaraudio /etc/init.d/
 # cp dist/debian-like/defaults /etc/defaults/roaraudio
```

Next you need to create the runlevel symlinks in order to make roard start at boot time:

```
 # update-rc.d roaraudio defaults
```

If anything was working you should now have a working RoarAudio setup. You can now continue configuring. See section Configuring on Debian.

3.4 From Sources

First you should go to the homepage at `http://roaraudio.keep-cool.org/` and download the newest version of RoarAudio.

After you downloaded the tarball you have to extract it:

```
 $ tar -xzvf roaraudio-0.2beta2.tar.gz
```

Now you can change into the RoarAudio directory and run *configure*:

```
$ ./configure
```

Next you can compile the code via GNU/Make. On GNU/Linux Systems you need to use the *make* command. On other systems you will need to use the *gmake* command.

```
$ make
```

After you compiled the sources you can install the binarys. Use the correct make command as stated above.

```
$ su
# make install
```

If you are a sudo user you can of cause also do this:

```
$ sudo make install
```

If everything is working you can clean up the build directoy:

```
$ make clean
```

Chapter 4

Configuring

roard: There is no common way on how to configure RoarAudio's Sound Daemon *roard* as normaly no configuration is needed at all. On some Systems there will be a config file to provide some options.

Clients: The clients normaly have there one config (for example most players have a configure dialog). If a client does not provide an option it is taken from the Enviroment or a config file as descripted in chapter Files and Enviroment or an internal list of defaults.

4.1 archlinux

On archlinux there is a config file at */etc/conf.d/roard*.

4.2 OpenBSD

4.3 Debian

On Debian there is a config file at */etc/default/roaraudio*. You can set options for RoarAudio via this config file.

4.3.1 Options

ROARD	This option sets if roard should be started. May be ”YES” or ”NO”.
ROARD_OPTS	This option can be used in order to set additional options for roard. Should normaly be empty.
ROARD_REALTIME	This controlls if roard runs in realtime mode. Possible values are ”NO”, ”YES” and ”DOUBLE”. Use ”DOUBLE” to select a *very realtime* mode.
ARAUDIO_DEFAULT_SOCKET	This can be used to set a global default socket name. This is normaly only usefull for client only setups.
ROARD_AF	This sets the *Address Famaly* to use for the listening socket. Valide values are ”UNIX”, ”TCP” and ”DECnet”.

ROARD_SOCKET	This sets the default socket filename for UNIX Domain Sockets.
ROARD_PORT	This sets the port for TCP Sockets.
ROARD_HOST	This sets the Host or Node for TCP or DECnet Sockets.
ROARD_OBJECT	This sets the object name for DECnet Sockets.
ROARD_RATE	This sets the sampling rate for the server. This is used as default for all output streams.
ROARD_CHANNELS	This sets the number of channels for the server. This is used as default for all output streams.
ROARD_BITS	This sets the number of bits per sample for the server. This is used as default for all output streams.
ROARD_DRIVER	This option sets the name for the primary driver.
ROARD_DEVICE	This is the device for the primary driver.
ROARD_DRIVER_OPTIONS	This sets options for the driver. The list of possible options depends on the driver.
ROARD_USER	This sets the user roard should run under.
ROARD_GROUP	This sets the group roard should run under. In addition this is the group user are abled to use roard, too. Normaly this should be set to "audio" for most systems.

Chapter 5

Setting up a player

5.1 XMMS

If you installed the RoarAudio XMMS plugin you can select it via pressing Control+P and select it under *Output plugin* in the tab *Audio I/O Plugins* (the first tab). After you selected the correct plugin you can accept the change by pressing the *OK* button.

After selecting the plugin it should work out of the box. You need to restart the current song in order to let XMMS switch the new plugin.

5.2 libao based

If you installed everything correctly you should be baled to activate RoarAudio support for all libao players by simply write the following into your */etc/libao.conf*. If the file does not exist it is save to create it:

```
default_driver=roar
```

5.3 Other

Players not listed above may only have a very beta plugin or need to use one of the Compatibility Librarys. See table 5.3 for a list of tested players.

Table 5.1: Tested players without native RoarAudio plugin

Player	**recommend library**	**working librarys**
mplayer	esd	esd
XINE	esd	esd
Amarok	esd	esd
Helix Player		
Totem		
Rhythmbox		
VLC		

Part III

User Manual

Chapter 6

RoarAudio's Architecture

6.1 Subsystems

Subsystems are internal parts within *roard* handling diffrent kinds of data.

Subsystem Name	Domain	Description
Waveform	time	This is the main subsystem. This handles Waveform signals in time domain. This are normal PCM streams.
MIDI	frequency	This is the MIDI subsystem. It's mainly used to control external MIDI devices.
Console Beep	frequency	This is in fact a subsystem of the MIDI subsystem. It is used to control the speaker on architectures with a system speacker.
Light	frequency	This is used to control theater light systems. Typicaly DMX is used.

Table 6.1: Table of Subsystems

6.2 Clients

A Client is a program connecting to roard. This may send some audio data but does not have to. For example roarctl never sends audio data to roard. A client **may** have multible audio streams open at a time.

A specal case is roard it self as it is a client of it self to provide a standard interface for *ouput streams* like soundcards.

6.3 Streams

A stream is a stream of audio data send from a appliation or device to roard or from roard to an appliation or device. A stream is **allways** owned by a client. For every stream roard knows a minimum of information at any time so it can work with. This includes values for the number of channels, the sample rate and the bits per sample. Also an codec ID is stored. A Stream may in addition have meta data and flags. The direction of a stream is stored via the *Stream Type*.

6.3.1 Stream Types

Play

This type of stream is a simple playback stream. The data is send by the client to roard and roard plays it back.

Record

This stream type is to read data from the soundcard. This is currently not supported and may not be supported in future.

Monitor

This type is a stream where roard sends the mixed audio streams back to a client. This can be used to in case a user wants to save a dump or to stream it to an streaming server.

Filter

A filter stream is a stream that is used to filter data while mixing. This may be used to apply audio effects.

After mixing is done roard sends all audio data via the filter stream to the client. The client can now do some operations with the audio and have to send it back to roard via the same stream.

This stream type requires that the data get send, altered and return in blocks of the same size as roard internaly uses (normaly $10ms$). Because of this an applicatiuons needs to be carefull use this type of stream. In addition not every codec can be used on this stream. Most compressing codecs can't be used.

Output

This type of stream is used in order to stream to an device via a driver. This type is only used for streams handled by roard itself.

Mixing

The Mixing stream is only used to represent the internal mixing buffer. It may not used via any application.

Bidir

A bidir stream is a bidirection stream. It is a *Play* and *Monitor* stream in one stream. The server reads audio data from the stream and sends the mixed resulte back.

Meta

This stream type was never used and is now obsoleted.

Thru

Thru streams are used in order to mirror the data of another stream. The Thru stream get all data you send to another input stream without any change. The stream get's terminated as soon as you terminate the stream or the stream you are mirroring is terminated.

Bridge

This stream type is used internally. Bridge streams are streams connected to more than one subsystem in order to transport data from one subsystem to another.

MIDI In

The stream is the same as a playback stream but for the MIDI subsystem.

MIDI Out

This is the same as a monitoring stream for but for the MIDI subsystem.

Light In

This is the input stream type for the Light control subsystem.

Light Out

This is the output stream type for the Light control subsystem.

Raw In

This is the input stream type for the raw data subsystem.

Raw Out

This is the output stream type for the raw data subsystem.

Complex In

This is the input stream type for streams of complex (mixed subsystem) type.

Complex Out

This is the output stream type for streams of complex (mixed subsystem) type.

6.3.2 Stream Flags

primary

If a stream flaged with the **primary** flag dies (simply end or get kicked) roard shuts down itself. This is normaly used on output streams to soundcards. This flag can be set on a already running stream.

sync

roard uses streams with the sync flag as clock source. There may be multible streams with this flag. If no stream has this flag roard is in free running mode. In free running mode it will complain that all streams have over- and underruns and will not work correctly. Normaly exactly one stream should have this flag set and this stream should be a stream to a soundcard. This flag can be set on a already running stream.

output

There is a driver used to read or write data to or from this stream. This flag can **not** be set on a already running stream.

source

This flag is set on sources. There is no techical meaning. This flag can **not** be set on a already running stream.

meta

This flag controls the behavor of the meta data on the stream. The behavor depends on the data direction or the stream. This flag can be set on a already running stream.

Input stream: If the flag is set on an input stream optput streams may take the meta data from this stream.

Output stream: If a output stream has this flag it gets a copy of all meta data from all input streams with this flag set.

Bidirectional stream: On bidirectional streams the behavor is undefined. You should not set this flag on such streams.

autoconf

This flag asks a driver to search for a working config if the given one does not work and alter the stream to correct the problem. This may only be used on output streams using a driver. This flag can only be set befor the driver module is loaded.

cleanmeta

This flag enables an automatic clreanup of meta data on the stream. This may be used to strip station name on webradio streams. This flag can be set on a already running stream.

pause

This flag pauses the stream. As long as the flag is set there will be no data read or written from/to the stream. **Most but not all** players handle this simular to there own pause state. Some players get broken when using this. The main porpose is to ensure there is no data transfered while setting parameters to the stream.

Depending on the codec there may be a flush/seekpoint/meta data injection on the stream at time of setting or resetting the flag.

hwmixer

This asks the server to use a hardware mixer to set the volume and to not use the internal software mixer. This is not supported by all drivers. In addition it should not set on a stream not driven by a driver. Setting of this flag fails if the driver des not support this flag. This flag **should** be set in a own call and not mixed with other flags.

mute

The mute flag will mute the stream (set it to be silanend). It stopps roard to mix the data of the stream into the others. Because of this, this is more effective than seing the volume to zero. This does not touch the mixer setting. Drivers and Bridges may pass this flag to lower layer, too.

mmap

This flag is used to ask for memory mapped access to IO. This may be used by some drivers. See the driver's documentation for more information.

This flag is not supported on all kinds of streams that access the network (on common operating systems).

antiecho

On bidirectional streams this asks the server to remove the signal send to the server from the signal send by the server.

Under the assumance that the mixer can be defined as equation 6.1 the antiecho flag changes the output of the stream from $O_c(s) = M(s)$ to $O_c(s) = M(s) - I_c(s)$.

$$M(s) = \frac{\sum_{c=0}^{n} I_c(s) * e^{j*T}}{n} \tag{6.1}$$

virtual

This stream is a child of another stream.

recsource

This stream is the source for recording streams.

passmixer

This flag does exactly the same as the *meta* flag expect that it does not transfer the meta data from one stream to another but the mixer settings. No software mixing is done on input streams with this flag set.

6.4 Driver

A driver is used in order to provide access to devices not supporting standard POSIX IO. Such devices may be soundcards for example, needing some specal ioctl()s.

6.5 Sources

Sources are normal streams. The only diffrens from a stream created by a application is that a sources is a input stream created by roard as soon as it becomes ready.

6.6 Codecfilter

Codecfilters are used to convert PCM data to high level compressed codecs or back to PCM. They are for example used in order to provide support to Ogg Vorbis. Some codecs filters are not supported by all stream types because of there blocking behavor.

6.7 Bridges

Bridges are streams used in order to connect two or more subsystems with each other. An example for this may a MIDI Synthesizer converting MIDI data to a waveform signal and because of this connecting the MIDI subsystem to the waveform subsystem.

Chapter 7

Playing music on command lion with RoarAudio

There are mainly three diffrent tools to play music on command lion with RoarAudio. There is a general one called **roarcatplay**, one to play Ogg Vorbis streams that shows meta data as well called **roarvorbis** and one that starts a background stream called **roarradio** supposted to play a webradio stream while working on the same console.

There is also a tool to provide more low level playback called **roarcat** which is mostly used to play back raw PCM streams. This is very intresting for use in scripts.

7.1 roarcatplay

roarcatplay requires normaly only one argument, a file to play. It connects to the server and plays back the file in foreground. It does not print any meta data nor any other info in case of no error.

A magic file type detection is done in order to find out the correct container and codec. If the magic detection fails it prints an error messsage. The detection requires that file file is seekable thrus this this does not work on pipes. See roarcat for a tool useable with pipes.

Decoding is done in **roard** in case it is needed (a non PCM codec is used).

For more information see End user Tools: roarcatplay.

7.2 roarvorbis

roarvorbis plays back music in Ogg Vorbis format. The playback is done foreground. It takes the file to play as argument.

In contrast to the other tools it decodes the Vorbis stream localy and shows the meta data.

For more information see End user Tools: roarvorbis.

7.3 roarradio

roarradio in contrast to all other tools plays the given music in background. It starts the stream and after handing it over to roard it terminates. This tool is normaly used to play long living streams like webradio. It supports HTTP thru wget.

You can control or stop the stream via roarctl as descripted in chapter Controlling played music.

For more information see End user Tools: roarradio.

7.4 roarcat

roarcat is normaly used to play back raw PCM streams. It may also play back other codecs if *–codec* is given. This tool is useful if used in scripts or in pipes together with other tools like sox. You may also set sample rate, bits per sample and number of channels via *–rate*, *–bits* and *–chans*.

In addition this tool is compatible with *esdcat*'s command lion arguments.

For more information see End user Tools: roarcat.

Chapter 8

Controlling played music

The normal way to control currently played music beside what the player supports is to use a tool called *roarctl*. It supports most of what the Sound Server can change on the fly. It supports general options to controll the server and options to control given stream or client.

For more information about roarctl see End user Tools: roarctl.

8.1 List current streams

To list all current streams you may use roarctl's command *liststreams*. You may add the flag *-v* multible times to get a more verbose output.

Example:

```
$ roarctl -v liststreams
```

8.2 Change volume

The volume is shown in the list of streams for each stream. To change it you need to know the *Stream ID* you can find out from the list of streams. If you know the Stream ID you can set the volume with roarctl's *volume* command.

Examples:

```
$ roarctl volume StreamID mono 50%
$ roarctl volume StreamID stereo 7000 20000
$ roarctl volume StreamID 2 30% 77.7%
```

As shown in the Example there are diffrent ways to set the volume. The first argument after the Stream ID is the number of channels. This may be the number of channels as integer, *mono* or *stereo*. In case of the two keywords roarctl trys to change the volume also if the number of channels does not fit. In case of mono it sets **all** channels to the same volume in case of stereo it trys to find out which channels are left ones and which are rigth ones and set both groups to the given values.

The volume can be given in two ways: as a abselut number from 0 to 65535 or as a percentage floating point value. The number of given values **must** be the same as given as number or channels as stated in the paragraph above.

8.3 Kick a stream

To kick a stream you need to find out the current *Stream ID* as shown above the streams in the list of streams. If you know the ID you can kick the stream by with this command:

```
 $ roarctl kick stream StreamID
```

8.4 Show and change meta data

8.5 Change stream flags

Chapter 9

Dumping and Streaming

9.1 roarmon

roarmon is a tool used to open a *monitor stream.* It reads the fully mixed audio data from the server and save it to a file or print the data to stdout.

Examples:

```
$ roarmon --chans 1      output.raw
$ roarmon --codec wave   output.wav
$ roarmon --codec vorbis output.ogg
```

An external encoder can be used to in case you want to encode to a unsupported codec or use an unsupported option as shown by the following example:

```
$ roarmon | myenc -o bla.ext -
$ roarmon | oggenc -q -1 -o lowquality.ogg -
```

9.2 roarshout

In order to stream to icecast a tool called roarshout can be used. It has the the same parameter as *oggfwd* beside that it also adds the normal RoarAudio client options.

Example:

```
$ roarshout myserver.de 8000 hackme /roar.ogg
```

Chapter 10

RoarAudio Daemon

roard is the central daemon within the RoarAudio Sound System. It does hold a connection to all clients, mix the audio and send it to all outputs including soundcards, streaming servers and monetoring clients. It may also decode high level codecs via a codecfilter.

10.1 General Audio Options

There are three general audio options: the sample rate, the number of channels and the number of bits per sample. For all three values there is a default so normaly you should not need to set those options.

You should set those values to the ones most common within your music collection. Setting them to higher values than what you try to play back may result in quality losse!

Typical values are a sample rate of $44100Hz$, 2 channels (stereo) and $16bit$ per sample. The sample rate may be set to any value from $1Hz$ to $> 1MHz$. The number of channels can be set to a value from 1 up to a compiled in maximum value. This maximum is typicly 64. The number of bits may be set to 8, 16 or 32. Note that in case you want to use $24bit$ audio you need to set the value to 32. roard will automaticly convert everything.

The set the options there are the option -*R* to set the sample rate in Hz, -*C* to set the number of channels and -*B* to set the bits per sample.

Examples:

```
$ roard -B 8
$ roard -C 4 -B 32
$ roard -R 48000
```

10.2 Drivers and Outputs

Output streams are streams where audio data is send from roard to some external resource. Often they are used together with a **driver** in order to send data to a soundcard.

roard knows diffrent drivers to support diffrent types of sound APIs. To get a list of all supported drivers you can run:

```
$ roard --list-driver
```

Possible Flags:

s: Driver is fh safe.

S: Driver uses old sysio interface.

V: Driver uses new VIO interface.

Possible Subsystems:

W: Waveform Subsystem.

M: MIDI Subsystem.

C: Console Speaker Subsystem.

L: Light Controll Subsystem.

R: Raw Data Subsystem.

X: Complex Data Subsystem.

Driver	**Devices**	**Description**
null	/dev/null	Null audio driver
roar	some.host.name	RoarAudio driver used to connect to remote roard
esd	some.host.name	EsounD driver used to connect to remote esd
oss	/dev/audio, /dev/dsp	Generic sound driver (works on most systems)
ao	driver	**(deprecated)** libao driver
sndio	/dev/audio, /tmp/aucat*	OpenBSD 4.5 sndio driver
shout	http://host/mount.ogg	libshout/icecast driver
raw	/some/file	RAW driver
dmx	/dev/dmx	DMX4Linux driver
pwmled	/dev/ttyS0	PWM controlled LEDs
sysclock	(none)	System Clock driver
cdriver	driver#device	Openes a cdriver

Table 10.1: Possible output drivers

Outputs are controled using the options *-o*, *-O*, *-oO* and *-oP*. With *-oN* you can add another output.

-o* and *-O set the driver used and the device name, filename or hostname to send the data to. The exact meaning of *-O* depends on the driver. The list of drivers (see example above) includes a hint on what each driver expectes.

Examples:

```
$ roard -o oss -O /dev/audio5
$ roard -o oss -O /dev/audio5 -oN -o roar -O another.host
```

-oO* and *-oP controls the options and flags of the output. *-oP* marks the output as primary. *-oO* sets all other stream options. Possible options are shown in table 10.2.

Option	**Type**	**Description**
rate	Integer	Sample rate in Hz
channels	Integer	Number of channels
bits	Integer	Bits per sample
codec	String	Codec to use
blocks	Integer	Number of blocks used in jitter buffer of devie. This is not supported by all drivers.
blocksize	Integer	Size of blocks used for device buffer
meta	Flag	Sets flag meta
sync	Flag	Sets flag sync
primary	Flag	Sets flag primary. This is the same as *-oP*.
cleanmeta	Flag	Sets flag cleanmeta
autoconf	Flag	Sets flag autoconf

Table 10.2: Possible output options

Example:

```
$ roard -o oss -oP -oO channels=1,bits=8,sync
```

10.3 Sources

Sources are input streams created by roard. They may for example be used to play startup sounds. The options are very simular to the ones used for outputs.

-s* and *-S set the type of source and the device-, file- or hostname. *-s* has a default value of 'cf'. **cf** is a simple file source. *-S* takes a single file name in case cf is used.

-sN creates a new source as shown in this example:

```
$ roard -S startup.wav -sN -S welcome.ogg
```

-sC* and *-sO set the container type used and options for the new stream. *-sC* is currently not used by any source. *-sO* normaly only have one possible option *codec*.

```
$ roard -S audiodump.alaw -oO codec=alaw
```

-sP is the same as *-oP* just for sources: it sets the primary flag on the stream. This will terminate roard in case the stream ends. This behavor may be useful in case of mirroring another roard.

10.4 Codecfilter

Depending on what is compiled in roard provides some codec filters to support high level codecs. to get a list use *–list-cf*:

```
$ roard --list-cf
```

10.5 Listen Connection

roard supports normaly diffrent types of sockets. You can use the options *-u*, *-n* and *-t* to select socket type *Unix Domain Socket*, *DECnet* and *TCP*. Some socket types may not be suppored on your operating system. Default is to use Unix Domain Sockets if possible.

In case of TCP you can select IPv4 and IPv6 via *-4* and *-6*. IPv6 is not fully supported yet.

On Unix Domain Sockets you can set the socket filename via *–sock*.

Example:

```
$ roard --sock /tmp/roard-42
```

On DECnet sockets you can set the nodename and object via *-b* in form *node*, *::objectname*, *node::objectname* and *::*. Default is to bind to *executor* (local) node and named object *roar*. Numerical objects may be set via *-p*. No object name may be given in this case.

Example:

```
$ roard -n -b ::roar42
```

On TCP sockets you can set the hostname to bind to via *-b* and the port with the option *-p*. Default is to bind to *localhost* only and port 16002. Use 0.0.0.0 as hostname in order to bind to any interface and run an public roard.

Example:

```
$ roard -t -b 0.0.0.0
```

10.6 Realtime

10.7 Security

10.7.1 User and group

You may set user and group for roard via *-U* and *-G*. Defaults are current user and group *audio*. Those values are used to set the permitions on the UNIX Domain Socket if used to listen on.

You may ask roard to switch user and group with *–setuid* and *–setgid*. If setting the user or group fails roard will terminate.

If you start roard at boot time or from an init/rc script normal values is to use user *roard* and group *audio*.

Examples:

```
$ roard -G users
$ roard -U roard -G audio --setuid --setgid
```

10.7.2 chroot

You may also chroot roard to add some more protection via *–chroot*. Please not that roard keeps filehandles pointing to files outside the chroot jail open. This resultes in the **possibily to chdir out of the jail**. chrooting should enhance security a lot but is not as save as in other cases.

Example:

```
$ roard --chroot /usr/chroots/roard/
```

Chapter 11

Using RoarAudio's MIDI Subsystem

11.1 Basics about MIDI and RoarAudio

What the MIDI subsystem does is in general the same as the waveform subsystem does: I mixes the audio. As MIDI is a message oriented protocol this is in fact not really done. In reailty the all messages get colected from the MIDI input streams and set to all midi output streams. But this behaves nearly the same as mixing in the waveform subsystem (time domain).

In addition RoarAudio can generate some control messages like a MIDI clock (see below) and volume change messages.

You can connect as many inputs and outputs via RoarAudio as you like (as long as you are below the maximum total number of streams). MIDI Thru streams are possible of cause via the normal Thru streams. Note that Thru streams have a significant higher latency as hardware MIDI Thru connectors. The lateny is normaly at somethere around $15..20ms$ in case of a normal roard installation (with a cycle frequency $cf = 100Hz$).

11.2 MIDI Clock

Per default *roard* provides a *MIDI Clock.* It uses the Waveform subsystem to generate the clocktics. This means that you need to have a Waveform stream connected providing a clock signal in order to use the MIDI Clock. This is normaly no problem as the soundcard driver will provide the Waveform clock per default.

The Clock does 96 ticks per beat. The default is one beat per secund or 60 beats per minute. It is enabled by default. If it sends out ticks can be controled by the *sync* flag on the clock bridge's stream. If the flag is set the clock will send ticks to the MIDI subsystem. If the flag is not set no ticks will be send.

11.3 Connecting devices to RoarAudio

Chapter 12

Using RoarAudio for Light Control

12.1 Basics about RoarAudio and Light Control

12.2 Connecting devices to RoarAudio

Chapter 13

Compatibility Librarys

In order to support old clients and clients of other sound systems RoarAudio provides so called *Compatibility Librarys*. They are binary compatible API emulations of the librarys of other sound systems. You can just install them and do not need to do anything else in order to get applications of other soundsystems working. The following section descripes how you can use them.

13.1 Using Compatibility Librarys

13.1.1 roarify

Beside installing them directly (which may lead in conflicts with existing software) you can use the program *roarify*. It changes the enviroment for a proces in a way that it will load RoarAudio's librarys not the ones installed by the system.

Note: roarify may not be avalible or work on all Operating Systems. It should work on all POSIX systems but at least use used mechanisms are not supportet by M$ Window$. Please do not run this Operating System.

To use roarify on a application you can simply use:

```
$ roarify myplayer
```

This is normaly all you need to do. In some cases (if the application starts with a dash (-) or you need to add special options) there is another form. The following example shows how to set a **default** server. The server may be overriden by the application:

```
$ roarify --server myserver -- myapp
```

13.2 Enlightened Sound Daemon

13.3 PulseAudio

13.4 aRts - KDE Sound System

13.5 YIFF Sound System

13.6 OpenBSD sndio

Chapter 14

End user Tools

Each tool has an manpage which explains it in details. Please see those manpages.

14.1 roarcat

14.2 roarcatplay

14.3 roarctl

14.4 roarradio

14.5 roarvorbis

Chapter 15

Networking

15.1 Connection Types

Socket Type	**Examples**	**Defaults**
UNIX	*/path/to/socket*	*/tmp/roar*, *$HOME/.roar*
DECnet	*mynode::*, *::myobj*, *node::obj*	*0.0::roar*
IPv4	*myserver.dom*, *server.dom:port*	*localhost:roar(16002)*
IPv6		*ipv6-localhost:roar(16002)*

Table 15.1: Examples of Socket Addresses

15.2 Proxy Server

Chapter 16

Files and Enviroment

16.1 General

16.1.1 Files

/etc/roarserver

/etc/roarserver is a symlink to a global default address for roard.

Examples:

```
$ ln -s /tmp/roarsock /etc/roarserver
$ ln -s remote.host.name /etc/roarserver
$ ln -s mynode:: /etc/roarserver
```

16.1.2 Enviroment

HOME

$HOME is the home directory of the current user.

ROAR_SERVER

$ROAR_SERVER is a local default for roard's address used in case the application does not provide a address to connect to.

16.2 roard

16.2.1 Enviroment

ROAR_DRIVER

$ROAR_DRIVER sets the output driver of roard. This is the same as roard's option *-o*.

ROAR_DEVICE

$ROAR_DEVICE sets the output driver's device of roard. This is the same as roard's option *-O*.

16.3 RoarAudio Clients

16.3.1 Enviroment

ROAR_PROXY

$ROAR_PROXY sets the type of a proxy used to connect to somewhere. The default is an emty value which means not to use any proxy at all.

socks_proxy

http_proxy

https_proxy

ssh_proxy

$*_proxy sets the name and maybe other options for the proxy of a certan type. The exact syntax depends on the proxy type.

Part IV

Developer Manual

Chapter 17

Audio and format conventions

17.1 Channel Mapping

Channels are mapped in the common way as shown by table 17.1. If your application needs a diffrent mapping you need to swap them.

Number of Channels	**Mapping**
1	0: Mono Mid Channel
2	0: Left, 1: Right
2 Mid-Side	0: Mid, 1: Side
3	0: Left, 1: Right, 2: Center
4	0: Front Left, 1: Front Right, 2: Rear Left, 3: Rear Right
5	0: Front Left, 1: Front Right, 2: Center, 3: Rear/Surround Left, 4: Rear/Surround Right
6	0: Front Left, 1: Front Right, 2: Center, 3: LFE, 4: Rear/Surround Left, 5: Rear/Surround Right

Table 17.1: Channel Mapping

17.2 Channel possitions

Table 17.2 shows where every channel is located.

	Left	**Center**	**Right**
Front	FRONT_LEFT LEFT	FRONT_CENTER CENTER/MONO	FRONT_RIGHT RIGHT
Side	SIDE_LEFT	*SIDE_CENTER*	SIDE_RIGHT
Back	BACK_LEFT	BACK_CENTER	BACK_RIGHT

Table 17.2: Channel possitions

The channel *SIDE_CENTER* is only listed to be complet. It can not exist mathematcly nor in reality as it is somewhere within the listener's head.

Possition of LFE is not shown as the listener can not hear the direction of the LFE signal.

17.3 Sample and Frame representation

Samples are normaly stored as signed integers. The size depends on the number of bits per sample. 32 bit integers are handled as 32 bit audio data not 24 bit. If some function accepts 24 bit integers this means that it expects 3 byte integers not 4 byte ones with only 24 significat bits. Internal operations and calculations happen with native byte order. Data send to the network or given to some programm may be in a diffrent format than this one depending on what it requested.

Frames are represented as a set of all samples for a given point in time with the order stated in Channel Mapping above. See table 17.3 for an example.

Frame 0: Left	Frame 0: Right	Frame 1: Left	Frame 1: Right	Frame n: ...
$\xrightarrow[\text{Time}]{}$				

Table 17.3: Example Frames for stereo signal

Chapter 18

Writing software using RoarAudio

18.1 Writing audio output plugins

This section covers call back based plugins as used by most players.

18.1.1 Writing basic very basic output

First of all you need to create a struct to store your privarte objects. This struct contains all data your callbacks need to know in order to play back the data.

To stream data to roard you need a so called VIO object. This is the first optject we need to store in the instance structure which may look like this:

```
struct MyPluginInst {
 struct roar_vio_calls vio;
};
```

Next we have 3 basic callback functions: one for opening the stream to roard, one for closing and one to write some data to roard.

The opening function does the setup of the connection and therefor it needs to know about the format of the input samples. In most cases they are allready PCM in native byte order. In this case we only need to take care about the nummber of channels, the nummber of bits per sample and the sampe rate.

To open the connection we use *roar_vio_simple_stream()*. Such an opening function may look like this:

```
int plugin_open (void ** inst, int rate, int channels, int bits) {
 struct MyPluginInst * self;

 if ( (self = malloc(sizeof(struct MyPluginInst))) == NULL )
```

```
  return -1;

 if ( roar_vio_simple_stream(&(self->vio), rate, channels, bits,
                             ROAR_CODEC_DEFAULT, NULL, ROAR_DIR_PLAY,
                             "name of player") == -1 ) {
  free(self);
  return -1;
 }

 *inst = self;

 return 0;
}
```

The closing function does simply close the open stream and free the memory allocated by the opening function:

```
void plugin_close (void * inst) {
 struct MyPluginInst * self = inst;

 roar_vio_close(&(self->vio));

 free(self);
}
```

Finnaly we need a function that writes the data to the server. This is done by calling *roar_vio_write()* on the VIO object. A simple function may look like this:

```
ssize_t plugin_write (void * inst, char * buf, size_t len) {
 struct MyPluginInst * self = inst;

 return roar_vio_write(&(self->vio), buf, len);
}
```

However most players do not retry writing data that has not written in a single write operation. This requires us to retry writing in case not all data have been written in a single write. The classical way of doing this is a loop of writes unless an error happens or all data is written:

```
ssize_t plugin_write (void * inst, char * buf, size_t len) {
 struct MyPluginInst * self = inst;
 ssize_t ret  = 1;
 ssize_t done = 0;

 while (ret > 0) {
  ret = roar_vio_write(&(self->vio), buf, len);

  if ( ret =< 0 )
   break;
  done += ret;
  buf  += ret;
  len  -= ret;
 }

 return done;
}
```

18.1.2 Adding support to set server

There are basicly two methodes of setting the server:

1. setting the server globaly,
2. setting the server per call.

The first one is a good choie in case you know that there is only peace of software with only one instace using libroar. This may for example be true if there is only this output plugin and it can only be loaded one time. In case where may be some kind of input plugin for example this is allready not true.

In case you can assure that there is only one instance you can use the function *roar_libroar_set_server()* like this:

```
 int plugin_open (void ** inst, int rate, int channels, int bits) {
  struct MyPluginInst * self;
  char * server;

  server = player_get_option("roarserver");

  if ( server != NULL )
   roar_libroar_set_server(server);

  [...]
 }
```

In case you you can not assure that there are other instances that use libroar you need to set the server on every function that opens a connection. All those function takes a argument of name *server*. It may be needed to store the server name within your private instance structure.

Please take care to not losse memory by incorrectly or not freeing the server name on in your private structure as you most probable need to create a copy using *strdup()* or a simular function.

18.1.3 Prepering the plugin for meta data, mixer settings and so on

Till now we have used a very simple API to open the stream to the server. In the following sections we want to add support for setting meta data or using roard's internal mixer. This requires the use of a a bit less simple API as we need to seperate the so called controll and data (stream) connection. Also we need a stream object for our stream in order to manipulate it. The object we currently have is a so called VIO object. It does not know about streams or something, just how to send the data to the server. Because of this, this object can not be used to manimulate meta data, the mixer settings or other more advanced properties of the stream. It of caus will keeped to be used to send the raw data to the server.

First we need to modify our privarte instance structure by adding a object for the control connection and the stream object (the VIO object will be keeped as data connection). The following example shows what is needed to be added:

```
struct MyPluginInst {
 struct roar_vio_calls vio;  // data connection
 struct roar_connection con; // control connection
 struct roar_stream stream;  // stream object
};
```

Most players have seperate functions for opening/closing the stream and the plugin. In this setp we will add code for the initialization/de-initialization function of our new plugin. In the init function we will open the control connection to the server and close it in the deinit function. The control connection can be reused by all of our streams.

Here is an example of a simple set of init/deinit functions:

```
int plugin_init (void ** inst) {
 struct MyPluginInst * self;

 if ( (self = malloc(sizeof(struct MyPluginInst))) == NULL )
  return -1;

 // set stream to something invalide by using stream ID -1:
 if ( roar_stream_new_by_id(&(self->stream), -1) == -1 ) {
  free(self);
  return -1;
 }

 // open control connection:
 if ( roar_simple_connect(&(self->con), NULL, "name of player")) == -1 ) {
  free(self);
  return -1;
 }

 *inst = self;

 return 0;
}
```

```
void plugin_deinit (void * inst) {
 struct MyPluginInst * self = inst;

 // close our stream in case one is open:
 if ( roar_stream_get_id(&(self->stream)) != -1 )
  plugin_close(self);

 // disconnect the control connection:
 roar_disconnect(&(self->con));

 // free used memory:
 free(self);

 return 0;
}
```

Because we moved the allocation of our private instace structure we need to change the open and close functions of the stream to respect that. In addition we are going to use the control connection and the stream object to create the new stream. This is done by calling the function *roar_vio_simple_new_stream_obj()*.

Here is an example of a simple new open and closing function:

```
int plugin_open (void * inst, int rate, int channels, int bits) {
 struct MyPluginInst * self = inst;

 // open the new stream and data connection:
 if ( roar_vio_simple_new_stream_obj(&(self->vio),
                              &(self->con),
                              &(self->stream),
                              rate, channels, bits,
                              ROAR_CODEC_DEFAULT, ROAR_DIR_PLAY
                                    ) == -1 ) {
  // we reset the stream object as it may contain infos
  // about the failed stream:
  roar_stream_new_by_id(&(self->stream), -1);
  return -1;
 }

 return 0;
}

void plugin_close (void * inst) {
 struct MyPluginInst * self = inst;

 // close the data connection,
 // the stream will automaticly be closed:
 roar_vio_close(&(self->vio));

 // reset the stream object to no active stream:
 roar_stream_new_by_id(&(self->stream), -1);
}
```

Now the plugin is prepered to support all kind of advanced properties of the stream. You may continue with adding meta data support or support for mixer settings or look at the rest of the documnetation and implement some specal feature.

18.1.4 Adding support for meta data

Next we can set some meta data on the stream. This is a good thing to do because it may be showed by programs listing the current streams of roard and because it can be passed to possible listeners in case we are stream radio in some way.

Within RoarAudio meta data has a type and value basicly. The type can for example be 'Artist', 'Title' or 'Album'. A stream can have multible meta data entrys of a single type. This may for example be the case if some song has two preformers.

First you need to clear the meta data on a stream. After that you can set new ones and finaly use a command to mark your changes be complet and ask roard to update all interal things.

The following shows a simple example of a function that can set title and artist on a stream. We will later expend this.

```
 int plugin_set_meta (void * inst, char * title, char * artist) {
  struct MyPluginInst * self = inst;
  struct roar_meta      meta;

  // clear the meta data object:
  memset(&meta, 0, sizeof(meta));

  // first we clear the meta data on the stream:
  roar_stream_meta_set(&(self->con), &(self->stream),
                       ROAR_META_MODE_CLEAR, &meta);

  // setting the title:
  meta.type  = ROAR_META_TYPE_TITLE;
  meta.value = title;

  roar_stream_meta_set(&(self->con), &(self->stream),
                       ROAR_META_MODE_SET, &meta);

  // adding the artist:
  meta.type  = ROAR_META_TYPE_ARTIST;
  meta.value = artist;
  roar_stream_meta_set(&(self->con), &(self->stream),
                       ROAR_META_MODE_ADD, &meta);

  // finaly we ask roard to update and commit the meta data:
  memset(&meta, 0, sizeof(meta));
  roar_stream_meta_set(&(self->con), &(self->stream),
                       ROAR_META_MODE_FINALIZE, &meta);

  return 0;
 }
```

This example allready features most of the meta data manipulations: clearing meta data, adding meta data, setting meta data and finally commiting it.

You may have noticed that we set the title with *ROAR_META_MODE_SET* but the artist with *ROAR_META_MODE_ADD*. The diffrence between both is that *ROAR_META_MODE_SET* overwrites all existing meta data entrys of the same type while *ROAR_META_MODE_ADD* will just add a new one.

18.1.5 Adding support for roard based mixing

18.2 Writing other output plugins

18.3 Writing input plugins

18.4 Writing mixer plugins/Mixer frondends

18.5 Accessing other (control) features

Chapter 19

libroar

19.1 Low Level Protocol API

19.2 Medium Level Protocol API

19.3 High Level Protocol API

19.4 Simple API

19.5 Sockets

19.6 Buffer

19.7 Stack

19.8 VIO - Virtual Input Output

19.8.1 Basic VIO API

roar_vio_read(3) and roar_vio_write(3)

Synopsis:

```
ssize_t roar_vio_read    (struct roar_vio_calls * vio, void *buf, size_t count);
ssize_t roar_vio_write   (struct roar_vio_calls * vio, void *buf, size_t count);
```

Description:

Parameters:

vio	The VIO Object to read or write data from/to.
buf	The data buffer to write or read to.
count	The number of bytes to read or write.

Return value: On success these calls return 0. On error, -1 is returned.

Examples:

```
ssize_t len;
char buf[1024];

while ((len = roar_vio_read(input, buf, 1024)) > 0)
 roar_vio_write(output, buf, len);
```

roar_vio_lseek(3)

Synopsis:

```
off_t   roar_vio_lseek   (struct roar_vio_calls * vio, off_t offset, int whence);
```

Description:

Parameters:

Return value: On success this call return 0. On error, -1 is returned.

Examples:

roar_vio_nonblock(3)

Synopsis:

```
int     roar_vio_nonblock(struct roar_vio_calls * vio, int state);
```

Description: This call changes the blocking state of the VIO Object. If a VIO Object is in blocking mode calls (mostly read and write) on the object will wait for completion befor returning. In non-blocking mode they will return as soon as possibe. roar_vio_read(3) will also return if there is no data in the input buffer.

For more information see your operating systems manual and documentation.

Parameters:

vio The VIO Object to change blocking mode on.

state The new state of blocking mode. May be ROAR_SOCKET_BLOCK or ROAR_SOCKET_NONBLOCK.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

```
roar_vio_nonblock(vio, ROAR_SOCKET_NONBLOCK);
```

roar_vio_sync(3)

Synopsis:

```
int     roar_vio_sync    (struct roar_vio_calls * vio);
```

Description: This call syncs the VIO Object. If a object is synced data in buffers are written to disk or network. Input buffers may also be changed. The exact behavor depends on the VIO Object's type.

This call may and should be used after sending a request to a remote end befor waiting for an response. If the VIO Object is not synced it may happen that the request will never be send as it is still in an writing buffer (for example in a compression layer waiting for more data) and the process waits for the response infinitly.

Parameters:

vio The VIO Object to sync.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

```
char response[3];

roar_vio_write(vio, "Req", 3);
roar_vio_sync(vio);
roar_vio_read(vio, response, 3);
```

roar_vio_ctl(3)

Synopsis:

```
int     roar_vio_ctl     (struct roar_vio_calls * vio, int cmd, void * data);
```

Description: This call sets or gets values from the VIO Object that does not fit into one of the other calls. This is mostly the VIO call do to things you would do with *ioctl(2)* on a standard POSIX file object. The list of supported commands (*cmd*) depends on the VIO Object's type.

Parameters:

vio The VIO Object to change blocking mode on.

cmd The command to execute on the VIO Object.

data A pointer to some kind of data. The meaning of this depends on the command. If the command does not take an argument you sould set this to *NULL*.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

```
int fh;

roar_vio_ctl(vio, ROAR_VIO_CTL_GET_FH, &fh);
```

roar_vio_close(3)

Synopsis:

```
int     roar_vio_close   (struct roar_vio_calls * vio);
```

Description: This call closes the VIO Object. All lower layer will be closed, too if there are any. This call does not free the VIO Object nor the objects at lower layers so the vio structure can be one the processes stack.

Parameters:

vio	The VIO Object to close.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

```
roar_vio_close(vio);
```

19.8.2 VIO Types

File API

Synopsis:

```
int roar_vio_open_file (struct roar_vio_calls * calls,
                        char * filename, int flags, mode_t mode);
```

Description:

Parameters:

calls	The VIO Object to open.
filename	The filename of the file to open.
flags	The flags to open the file with. This is the same as the flags parameter of *open(2)*.
mode	The mode (*chmod*) used while creating a new file. This is the same as the mode parameter of *open(2)*.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

File Handle API

Synopsis:

```
int roar_vio_open_fh (struct roar_vio_calls * calls, int fh);
```

Description: This opens an allreay open file. If the VIO Object is closed the filehandle is closed, too.

Notes: You should not use this to open a socket fh/fd. Use roar_vio_open_fh_socket(3) in this case. You also should not use this to open a stdio (*FILE**) object allready open. Use roar_vio_open_stdio(3) in this case.

Parameters:

calls The VIO Object to open.

fh The filehandle to convert to a VIO Object.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Socket API

Synopsis:

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Socket Handle API

Synopsis:

```
int roar_vio_open_fh_socket (struct roar_vio_calls * calls, int fh);
```

Description:

Parameters:

calls	The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

stdio API

Synopsis:

```
int roar_vio_open_stdio (struct roar_vio_calls * calls, FILE * dst);
FILE *  roar_vio_to_stdio      (struct roar_vio_calls * calls, int flags);
```

Description:

Parameters:

calls	The VIO Object to open.
dst	The next lower stdio (FILE*) layer to use.
flags	...

Return value: On success this call return 0. On error, -1 is returned.

Examples:

```
struct roar_vio_calls vio;

roar_vio_open_stdio(&vio, stderr);

roar_vio_printf(&vio, "Hello world!\n");
```

Pass API

Synopsis:

```
int roar_vio_open_pass (struct roar_vio_calls * calls, struct roar_vio_calls * dst);
```

Description: The pass VIO API adds a simple layer that just passes all calls to the next layer. This is used as example and by some VIO types that passes for example read and write calls to the next layer but need a handshake or have some specal control options.

Parameters:

calls	The VIO Object to open.
dst	The next lower VIO Layer to use.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Re API

Synopsis:

```
int roar_vio_open_re (struct roar_vio_calls * calls, struct roar_vio_calls * dst);
```

Description: The re VIO API is very simular to the pass API. The only diffrence is that it will retry read and write operations untill all data is read or witten or an error occured.

Parameters:

calls	The VIO Object to open.
dst	The next lower VIO Layer to use.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Pipe API

Synopsis:

```
int roar_vio_open_pipe (struct roar_vio_calls * s0, struct roar_vio_calls * s1,
                        int type, int flags);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

VIO Stack API

Synopsis:

```
int roar_vio_open_stack (struct roar_vio_calls * calls);
int roar_vio_stack_add  (struct roar_vio_calls * calls,
                         struct roar_vio_calls * vio);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Protocol API

Synopsis:

```
int roar_vio_proto_init_def (struct roar_vio_defaults * def,
                             char * dstr, int proto,
                             struct roar_vio_defaults * odef);
int roar_vio_open_proto     (struct roar_vio_calls * calls,
                             struct roar_vio_calls * dst,
                             char * dstr, int proto,
                             struct roar_vio_defaults * odef);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Magic API

Synopsis:

```
int roar_vio_open_magic (struct roar_vio_calls * calls,
                         struct roar_vio_calls * dst,
                         int * codec);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

VIO OpenSSL BIO API

Synopsis:

```
int   roar_vio_open_bio (struct roar_vio_calls * calls, BIO * bio);
BIO * roar_vio_to_bio   (struct roar_vio_calls * calls);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

Command API

Synopsis:

```
int roar_vio_open_cmd(struct roar_vio_calls * calls, struct roar_vio_calls * dst,
                      char * reader, char * writer, int options);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

gzip API

Synopsis:

```
int roar_vio_open_gzip(struct roar_vio_calls * calls,
                       struct roar_vio_calls * dst, int level);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

OpenPGP API

Synopsis:

```
int roar_vio_open_pgp_decrypt    (struct roar_vio_calls * calls,
                                  struct roar_vio_calls * dst, char * pw);
int roar_vio_open_pgp_store      (struct roar_vio_calls * calls,
                                  struct roar_vio_calls * dst, int options);
int roar_vio_open_pgp_encrypt_sym(struct roar_vio_calls * calls,
                                  struct roar_vio_calls * dst,
                                  char * pw, int options);
int roar_vio_open_pgp_encrypt_pub(struct roar_vio_calls * calls,
                                  struct roar_vio_calls * dst,
                                  char * pw, int options, char * recipient);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

DSTR - Descriptive String API

Synopsis:

```
int roar_vio_dstr_init_defaults   (struct roar_vio_defaults * def,
                                   int type, int o_flags, mode_t o_mode);
int roar_vio_dstr_init_defaults_c (struct roar_vio_defaults * def,
                                   int type, struct roar_vio_defaults * odef,
                                   int o_flags);

int roar_vio_open_default         (struct roar_vio_calls * calls,
                                   struct roar_vio_defaults * def);

int roar_vio_open_dstr            (struct roar_vio_calls * calls,
                                   char * dstr,
                                   struct roar_vio_defaults * def, int dnum);
int roar_vio_open_dstr_vio          (struct roar_vio_calls * calls,
                                   char * dstr,
                                   struct roar_vio_defaults * def, int dnum,
                                   struct roar_vio_calls * vio);
```

Description:

Parameters:

calls The VIO Object to open.

Return value: On success this call return 0. On error, -1 is returned.

Examples:

19.8.3 VIO Operations

19.8.4 VIO select

19.8.5 DSTR - Descriptive String API

Syntax

Defaults

Chapter 20

libroardsp

Chapter 21

Project's coding style and conventions

21.1 File encoding and New Lions

All text files are in Latin-1 or if not possible in 7 bit ASCII. The line termination is done with plain Lion feet. The last lion is terminated, too, as common on all systems but Window$.

21.2 File Header and Footer

21.2.1 File header

The header of any code file should start with a comment including only the name of the file terminated by a colon. This comment is followed by a empty lion.

The seound paragraph is a multi lion comment (if possible) including a copyrigth statment for each author (even if the author done only trivial changes! Trivial changes may be marked as such) followed by a lincense statment.

21.2.2 File footer

The last two lions of any code file are a empty lion and a comment only including the string 'll'.

Attention: Make sure the final command is followed by a lion terminator. They are often not saved by bugy Window$ editors.

21.3 Indenting and code blocks

Code blocks are indented by a single space. There MUST be no tabs in the code file. Tabs as data need to be escaped. They should be avoided for text output to the user.

```
//example.c:

/*
 * Copyright (C) You Realname  - 1842
 * Copyright (C) Somebody else - 1845 (only trivial changes)
 *
 * This code is under the bier licens.
 */

#include <stdio.h>

int main (int argc, char * argv[]) {

 return 0;
}

//ll
```

Figure 21.1: Indenting example

{ Used to start a code block are on the same lion as the if (), while (), for (), ...

In case a if has an else statement { and } must be used even if one or both parts contain only a single statement.

21.4 Use of spaces

Basic rule is one space before and after each operator.

Exeptions:

- No spaces between function name and argument list.
- No space before ','.
- No space after '(' and before ')'.
- Space before '(' in case of if, while, for,...

21.5 Use of NULL

In case you compare a pointer to NULL use NULL explicitly, do not assume NULL to be zero or a false value.

Wrong:

```
if ( ptr )
 roar_mm_free(ptr);
```

```
//example.c:

#include <stdio.h>

int main (int argc, char * argv[]) {
 int    i   =  5;
 int    g   = -8;
 char * str = "bla";

 printf("i=%i, g=%i, str='%s'\n", i, g, str);

 return 0;
}

//ll
```

Figure 21.2: Indenting example

Right:

```
if ( ptr != NULL )
 roar_mm_free(ptr);
```

21.6 Use of pointers to struct or union members

If you use pointers to struct or union members use (and) to make clear what part the pointer points to.

Wrong:

```
ptr = &obj->member;
```

Right:

```
ptr = &(obj->member);
```

21.7 Use of goto

Do not use goto at all.

21.8 Comments

21.9 Name of Objects, Vars, . . .

- It's *fh* not *fd*.
- *s* is a stream object pointer.

```
//example.c:

/*
 * This is a big multi lion
 * comment descriping the file.
 */

#include <stdio.h>

int main (int argc, char * argv[]) {
 int    i   =  5;     //  5 is a nice number.
 int    g   = -8;     // -8 is just a negative example.
 char * str = "bla";  // a test string.

 printf("i=%i, g=%i, str='%s'\n", i, g, str);

 return 0;
}

//ll
```

Figure 21.3: Example of comments

- *ss* is a server stream pointer.
- *c* is a client object pointer.
- objects, macros, ... with a name starting with a underscore are file-local.
- macros and #define-d consts are in all uppercase expect in very spacal cases (for example AF_DECnet).
- if possible IDs of internal objects are of type int.

21.10 Memory Management

The provided functions roar_mm_*() should be used for all internal dynamical memory allocation. (for example: roar_mm_malloc(), roar_mm_free()). In case a buffer is passed to or from the application the standard C (POSIX) functions should be used.

In case of bigger buffers holding binary data (not strings) you may want to use roar_buffers.

21.11 Use of native data types

It is a common error to assume int to have 32 bist and short to have 16 bits. Use explicide types if needed. See table 21.11 for some common types.

For counters you should use *size_t* if possible. You must use *size_t* for length of data (for example array or buffer) if possible.

If you do not need an exact size but aminimum size you should not force exact size but use the *int_leastN_t*-types.

You must not use modifyer like *unsigned* or *long* without base type (normaly *int*).

Type	Safe value range	Common alternatives
int	$-32768..32767$	int32_t, int_least32_t
short	$-128..127$	int16_t, int_least16_t
long int	$-32768..32767$	int32_t, int_least32_t, size_t
long long int	$-32768..32767$	int64_t, int_least64_t, size_t
char	$-128..127$	-/-
void *	NULL, any pointer	-/-
size_t	$0..n$	-/-
ssize_t	$-1..m$	-/-

Table 21.1: Safe value ranges of data types

21.12 Order of arguments

If a function has a input and output agrument like a converter using two buffers the order is out then in.

Wrong:

```
int conv(void * in, void * out, long len);
```

Right:

```
int conv(void * out, void * in, size_t len);
```

Chapter 22

RoarAudio Protocol

22.1 RoarAudio Protocol and OSI Layer Model

Layer	Name	RoarAudio Object Type	RoarAudio Module	Examples
7	Application	Stream	Mixer	Your favorit song
6	Presentation	Codec	Codec Filter	PCM, $A - Law$, $Vorbis$, ...
5	Session	Client, Message	Control Logic	Commands: QUIT, NEW_STREAM, ...
4	Transport	VIO, Socket	IO	TCP, NSP, ...
3	Network	VIO	IO	IP, DRP, ...
2	Data link	Controller		$Ethernet$, $RS232$, I^2C, CAN
1	Physical	None		Wire, Fiber, Wireless

Table 22.1: RoarAudio Protocol in OSI Layer Model

The RoarAudio protocol mainly lifes in ISO Layer 5. RoarAudio interprets layer 6 as Codecs and 7 as raw data. See table 22.1 for a overview.

However RoarAudio may use some additional layer presentation and session layers between OSI layer 4 and 5. This is for example used to support proxy servers, compression and encryption.

22.2 Messages, Requests and Replys

22.3 Protocol Mappings

22.3.1 Real Streams: TCP, DECnet (NSP)

On real streams you are suppost to send regular messages one by one. You are allowed to send multible messages at once and then wait for all replys.

You MUST NOT use possible values for message/data length provided by layers lower than RoarAudio's messages at layer 5.

22.3.2 RS232

22.3.3 I2C

Messages are mapped the same way on I^2C as on RS232. With two diffrences:

1. Each message needs to be packed into a singel I^2C message framed by START and STOP.

2. Each I^2C message containing a RoarAudio message starts with the source address of the node sending the message directly after the destination address as specifyed by I^2C. If the size of the source address is not a multible of 8 bit then it it will be transmited as ne next bigger multible of 8 bit with the most significant bits set low. Both peers (server and client) need to be configured to know about this.

Virtual node addresses may be used to create multble independet connections from a singel I^2C node. This means that a singel I^2C node may act as if it is multible nodes with multible node addresses. This is for example needed in case of a bridge between I^2C and another Bus.

22.3.4 CAN

If no data length is given then data length is calculated using formular 22.1. This Means that messages with a total length ≤ 8 byte may not need to have a data length set. This saves a full byte and provides more space for a command.

If the used message protocol supports error detection the error detection **should not** be used. It may be used on first and last frame in order to check the data integrity and ensure all frames got received in the correct order if the message is lager than 136 byte including headers.

Frame Length should not be used on CAN because the CAN Bus already provide length information for the frames.

22.3.5 MIDI

22.4 Message Format

22.4.1 Format version 0

Version 0 messages consists of two parts: The header and a body part. The header has a fixed size of 10 Byte. The body has a varibale size. The size is given in the header. The length is 16 Bit encoded leading to a size of 0..65535 byte per message. The header format can be found in table 22.2. All header fields a in network byte order.

0	1	2	3	**byte**
Version (0x00)	Command	SID		0-3
Stream	possition			3-7
Data	length	Data	...	7-11

Table 22.2: Format of message version 0

22.4.2 Format version 1

DRAFT — DRAFT — DRAFT

The Protocol Version 1 is in draft currently. It is not suppost to replace the protocol version but to extend it. It is designed for byte oriented seriel comunication. It extends the version 0 format by adding a additional flags field. Most of the other fields can be shorted or simply removed.

The current draft for version 1 headers can be found in table 22.3. Needed fields are in **bold**, optional optional fields normal and optional long fields are in *italic*.

0	1	2	3	**byte**
Version (0x01)	**Flags**	**Command**	SID	0-
SID	Stream possition	*Stream possition*		-
Stream possition	Data length	*Data length*	Data...	-
Data...			Error Detection	- end

Table 22.3: Format of message version 1

As you can see in table 22.3 the minimal message size is 3 byte. This saves 7 byte. However the maximum size of the header is 11 byte, so a full message with all options is one byte longer than a version 0 message. But as the *Stream Possiton* field (4 byte) is only used by a very small group of commands the avarage message size is smaller.

The meaning if the message flags can be found in table 22.4.

If no data length is provided in the header it may be provided by the next lower layer (normaly OSI layer 1 or 2). The data length is calculated by formular 22.1. If no lower layer provides a length a length of zero is used. If a lower layer uses a fixed minimum packet size and pedding is used a *Data length* field in the header **must** be provided.

$$PacketLength - HeaderLength = DataLength \qquad (22.1)$$

Bit	**Flag Name**	**Long flag name**	**Flag Description**
7	MF	Meta Framing	
6	ED	Error Detection	
6	$\bar{LDL}$	Long Data Length	
5	$\bar{LSPOS}$	Long Stream Possition	
4	$\bar{LSID}$	Long Stream ID	
2	$\bar{DL}$	Data Length	
1	$\bar{SPOS}$	Stream Possition	
0	$\bar{SID}$	Stream ID	

Table 22.4: Message flags for format version 1 messages

If Meta Framing is set there are two possibilitys:

1. If *Data Length* is **not** set this message is a start of a meta frame.
2. If *Data Length* **is** set this message is a continued frame.

Note: If meta framing is used the message needs to have a *Data Length* field in the header. This is needed anyway as the data length can not be zero or privided by a lower layer.

In case of a continued frame the bits SID, $SPOS$, $LSID$ and $LSPOS$ are used as *Frame ID*. The Frame ID is calculated as stated in formular 22.2.

$$SID + 2 * SPOS + 4 * LSID + 8 * LSPOS = FrameID \tag{22.2}$$

The Frame ID get's incremented by one with each frame sent and start as zero with the first continued frame. The receiver must use the Frame ID to reorder frames in case they get out of order on the communication channel. As there are only 16 Frame IDs is is save to send up to 17 frames. This is for example a maximum of 100 byte on CAN. If the message is longer than the maximum size the lower layer supports in 17 packets the sender continues to send frames. If the Frame ID overflows (would become 16) it is set to zero. The sender **may** make a small break (maximum the time of two frames) in order to ensure the order of the messages on the lower layers.

If the ED **flag is set** an *Error Detection Byte* is added at the end of the Frame. If the flag is set on the first and last frame of the message the last error detection byte is calculated over all of the message and not only the last frame.

The LDL **flag** is used to tell the Meta Frame logic that the frame contains a *Frame Length* byte. It contains the length of the current frame. This is needed if the lower layer can not tell the length of the current frame.

A Frame should be mapped into a packet of the next lower layer protocol. If this is not ensureable an implementation should try to ensure it in a resanable way. Implementations receiving framed messages **must not** relay on this.

0	1	2	3	**byte**
Version (0*x*01)	**Flags**	Data Length	Data. . .	0-
Data. . .			Error Detection	- end

Table 22.5: Format of version 0 continued Meta Frames

22.4.3 Format version 2

DRAFT — DRAFT — DRAFT

0	1	2	3	**byte**
Version (0*x*02)	Command	SID		0-3
Flags				4-7
Stream	possition			8-11
Stream	*possition*	*(4 byte extention)*		(12-15)
Data	length	Dir Seq		12-15(16-19)
Data	. . .			?
CRC	. . .			?

Table 22.6: Format of message version 2

22.5 Commands

. . .

22.5.1 Base and connection commands

The following section explains basic commands controlling the connection. The possible responses are discussed in the next section in details. Table 22.5.1 is a brief overview of all commands discussed in this section.

#	**Command**	**Size**[1]	**Headers**[2]	**Description**
0	NOOP	0..Max/0..Max	gpos	No Operation.
1	IDENTIFY	/0	gpos	Identify a connection
2	AUTH	0/0	gpos	Auth a connection
6	QUIT	0/-	None/-	Terminates the connection.
31	GETTIMEOFDAY		None.	Get a timestamp from the server
32	WHOAMI	0/1*	gpos	Asks server about information about this connection

Table 22.7: Table of base and connection commands
Entrys marked with * are subject to change, entrys marked with . are drafed.

NOOP

The No-Operation command does not do any operation on the server side expect to send back a positive response.

The server response with a positive reply. A negative reply is invalid. The response may be of size zero or of the same size as the request. In case the response is of the same size as the request the orgiginal data should be send back to the client. A diffrent size than zero or the site of the request is invalid at this time.

The command may be used in order to ping the server or to do any kind of keep-alive. The client is free to send this command at any time. Sending this command before any kind of identification or authentification of the client is valid.

The data of the request does not contain any information. The server **MUST NOT** do anything with the data expect sending it back to the client in case it does send any data back to the client.

IDENTIFY

AUTH

See commends in libroar/auth.c.

QUIT

This command terminates the connection. This is valid at any time. Shuting down of the socket is done as soon as the request is processed by the server. The server is free to send a positive response but does not need to do so. This command can not fail. The client have to close the socket after a possible response is read from the socket.

The data length is zero for this commands on both the request and reply message.

GETTIMEOFDAY

WHOAMI

This command is used to get the ID of the client sending the request. The server will response with a positive answer with a data length of one byte. The byte contains the ID of the client sending the request.

22.5.2 Possible repsonse commands

This section descripes responses for requets of the client. The following commands **MUST NOT** be send as request.

#	Command	Size[3]	Headers[4]	Description
252	EPERM	-/-	-/-	*Internal. obsolete.* Permittion error.
253	OK_STOP	-/0..Max	-/any	*Internal.* Same as OK but do not process another message of this client before next main cycle
254	OK	-/0..Max	-/any	Command succeeded with no error
255	ERROR	-/0	-/None	Some error occurred

Table 22.8: Table of response commands
Entrys marked with * are subject to change, entrys marked with . are drafed.

EPERM

This command was used internaly and is now obsolete. **It should not be used in any case.**

OK_STOP

This is the same as OK but used in order to signalize the server internaly to stop continuing handling requests from this client in the current cycle. The server **MUST NOT** send this command but a normal OK to the client.

OK

This is a positive response. The data, data length and header fields used depends on the command this is the reply to.

ERROR

This is a negative response. The data length is zero. This is send by the server in case of any error with the request of the client. This includeds the following things:

1. Invalide parameter from client,
2. All kinds of IO errors on the server side,
3. Errors from the Operating system on the server,
4. Out-of-resource errors.

22.5.3 Server Status commands

22.5.4 Stream control and data commands

22.5.5 Other commands

#	Command	Size[5]	Headers[6]	Description
7	GET_STANDBY	0/2*	gpos	Get information about the current standby state
8	SET_STANDBY	2*/0	gpos	Set the current standby state
9	SERVER_INFO	0./	None.	Get information about the server
10	SERVER_STATS	0./	None.	Get stats from the server
11	SERVER_OINFO	0/	None	Get stream info for the mixing stream
13	EXIT	0,1/0	gpos	Terminates the server.
25	GET_ACL		None.	
26	SET_ACL		None.	

Table 22.9: Table of server status and control commands
Entrys marked with * are subject to change, entrys marked with . are drafed.

#	Command	Size[7]	Headers[8]	Description
3	NEW_STREAM	/0	None/SID	Start a new Stream
4	SET_META	3..?/0	SID/None	Set meta data
5	EXEC_STREAM	0	SID/None	Mark a stream as execed on the current connection
12	ADD_DATA	0..Max/0	SID*/None	Send some data to the server
14	LIST_STREAMS		None	List all streams
17	GET_STREAM	1*	None*/SID	Get stream info about a stream
19	SET_VOL	6..6+2C/0	SID	Set volume for a stream
20	GET_VOL	?..Max	SID	Get volume of a stream
21	CON_STREAM	4..?/0	SID/None	Ask the server to connect a stream
22	GET_META	2/2..Max	SID/None	Get meta data for a stream
23	LIST_META	1/1..Max	SID/None	List meta data on a stream
27	GET_STREAM_PARA	4..?/4..Max	SID/SPOS*	
28	SET_STREAM_PARA	8/0	SID/None	
29	ATTACH	6..?/0	SID/None	Attach a stream to the server
30	PASSFH	0	SID/None	Pass an open FH to the server for passive mode

Table 22.10: Table of stream commands
Entrys marked with * are subject to change, entrys marked with . are drafed.

#	Command	Size[9]	Headers[10]	Description
15	LIST_CLIENTS		None	List all clients
16	GET_CLIENT	1*	None	Get informations about a client
18	KICK	4/0	None	Kick a object from the server
24	BEEP			
-1	EOL	-/-	-/-	*Internal.* End of List

Table 22.11: Table of other commands
Entrys marked with * are subject to change, entrys marked with . are drafed.

Part V

Codecs, Container and Codec Mapping

Chapter 23

Codecs

23.1 PCM

PCM is a very common form of representing audio. There is may documentation out there about it. As a start you can read the Wikipedia article at: http://en.wikipedia.org/wiki/Pulse-code_modulation.

23.2 A-Law

A-Law is a PCM like codec but uses a log-scale. It is mainly used within the europe ISDN system. You can start reading aout it in details at Wikipedia at: http://en.wikipedia.org/wiki/A-law_algorithm.

RoarAudio's A-Law support is mainly based on sox. See the code (libroardsp and sox) for more details.

23.3 u-Law

Like A-Law μ-Law is a PCM like codec used mainly in North America's and Japan's telecomunication systems. See Wikipedia's article for more informaton: http://en.wikipedia.org/wiki/%CE%9C-law_algorithm.

RoarAudio's μ-Law support is mainly based on sox. See the code (libroardsp and sox) for more details.

23.4 Vorbis

For information on the Vorbis codec see the Vorbis specification at http://www.xiph.org/vorbis/doc/Vorbis_I_spec.html.

23.5 Speex

See Speex documentation at: http://www.speex.org/docs/.

23.6 FLAC

You can find the *FLAC Format Specification*: http://flac.sourceforge.net/format.html.

23.7 MIDI

23.8 CELT

Please see the the CELT homepage for any information on CELT: http://www.celt-codec.org/.

23.9 DMX512

The DMX512 codec is the most simple why to present DMX data and because of this currently used default codec for the light subsystem.

The DMX data is send in so called frames including one value per DMX channel of a DMX universe. Multible universe streams are not supported.

Each frame is a block with a length of 512 Byte. Each byte represents the value of one DMX channel. Values for start or stop bytes are not supported.

RoarAudio writes one DMX512 frame per internal cycle on output streams. As the default internal cycle frequenzy is $cfreq = 100Hz$ the used data rate is $51.2kByte/s$. See formular 23.1 for details.

$$DataRate[\frac{Byte}{s}] = cfreq * 512Byte \tag{23.1}$$

On input streams RoarAudio updates the data of all channels in the DMX universe the stream belongs to as soon as a complete frame is recived.

Frames must allways be witten in one operation or within one frame/packet of the container.

23.10 RoarDMX

23.11 VCLT (Vorbis Comment Like Text)

Vorbis Comment Like Text is a simple dummy codec for meta data. It does not contain any timing data (without a container) nor does it support binary data.

The format is simply one meta data per lion like shown in the following example:

The lion endings are plain '\n' (0x0A), the strings are encoded as UTF-8.

```
TITLE=Some Song
ARTIST=The singer
TRACKNUMBER=06
```

Figure 23.1: Example of VCLT Stream

23.12 RALT (RoarAudio Like Text)

23.13 RALB (RoarAudio Like Binary)

Chapter 24

Container

24.1 Ogg

For information about the Ogg Container Format see RFC 3533 (”The Ogg Encapsulation Format Version 0”).

24.2 RAUM

RAUM is a container used and developed in order to support the RoarAudio project with a container for realtime transmitsion of compressed audio needing framing and for archiving.

For all information see the homepage at: http://raum.keep-cool.org/.

24.3 RIFF/Wave

Chapter 25

Mappings

25.1 Ogg Vorbis

For information on the *Vorbis to Ogg mapping* see the Vorbis specification at http://www.xiph.org/vorbis/doc/Vorbis_I_spec.html.

25.2 Ogg Speex

See Speex documentation at: http://www.speex.org/docs/.

25.3 Ogg FLAC

The *Ogg FLAC Mapping* can be found at: http://flac.sourceforge.net/ogg_mapping.html.

25.4 Ogg CELT

Please see the the CELT homepage for any information on CELT: http://www.celt-codec.org/.

25.5 Ogg RoarDMX

25.6 RAUM Mapped Ogg

This mapping is used to be abled to store data of most of the codecs supported by RAUM in a Ogg container.

The BOS page consists of 3 packets: The identification header, the setup header and the meta data header realized as a Vorbis Comments block (see bellow). The identification and the setup header needs to be in the same Ogg Page. The meta data header can be on the same page with the other headers. The first data packet needs to start on a new page.

All integers in the headers are in network byte order. Strings (expact in the meta data block) are in ISO-8859-15 however they should be keeped high-bit clean meaing 7 bit ASCII compatible.

The identification header consists of the mapping magic and a magic for the used codec as shown in the following table 25.1:

0	1	2	3	**byte**
'R'	'A'	'U'	'M'	0-3
'M'	'O'	Null byte ($0x00$)	Version ($0x00$)	4-7
Flags (32 bit)				8-11
Codec ID				12-15
Null byte terminated	codec name	...		1-

Table 25.1: Identification header

The codec name is a string containing a MIME type in case it contains a slash ('/') or the RoarAudio codec name in case it does not. This string **MUST** be null terminated.

A player is suppost to check the codec ID and decide on it if it supports a codec. If the codec is not supported according to the ID the player **MUST NOT** continue with this stream. If the codec is supportet but supports diffrent flavors the player may use the codec name as hint for the flavor but **MUST** also work with a wrong codec name. This may for example be done by testing for the named flavor fist, then for other in case it does not match. The application **may** safely ignore the codec name.

The setup header is an array of 32 bit intergers to store all the needed parameters. In case a parameter is of no intrest for a codec the value is set to zero. The list of parameters is the following:

1. Stream direction (may be ignored by players)
2. Stream relative possion ID (may be ignored by players)
3. Sample rate in 1/1024 of Hz.
4. Bits per Sample/pixel.
5. Nummber of channels. A Video may set this to the total number of pixels ($ch = x * y * z$).
6. Width (x) in pixel.
7. Height (y) in pixel.
8. Deep (z) in pixel. A 2D Video has set this to one.

9. The nummber of frames pluse one between each key frame. If every frame is a keyframe this nummber is set to one $(0 + 1 = 1)$. If the number of frames between the key frames is not constant the value may be set the the avergae but with fliped sign (negativ). In case of unknown keyframe rate the value is set to zero.

All the tailing zeroed parameters can be skiped to save storage. The length of the packet than indecates which parameters are include. For example most audio files will only need the first 5 values be set so the size of the packet is $size = 5ints * 4Byte/int = 20Byte$. If the packet is longer than supported an therefor some unknown parameters are set the player **MUST** refuse to play the stream. It **MAY** have an option to disable this and play the file anyway so the user can manually override. Enabiling this option per default is against this specification.

The meta data header contains a set of Vorbis comments as specified in the Vorbis documentaion at http://www.xiph.org/vorbis/doc/v-comment.html.

This header may be on a Ogg page of it's own or on the same page as the identification header and the setup header. It must not be on the same page as the first data segment.

After all the headers the data of the codec follows starting with a fresh page.

The granule position is set to the same value as the stream position on a RAUM stream expect that the higher 32 bit are used to count the overflows with the expection that the most significant bit needs to be cleared (zero) all the time (negative values are reserved for future use).

25.7 RAUM PCM, A-Law, u-Law

25.8 RAUM RoarSpeex

25.9 RAUM RoarCELT

25.10 RAUM DMX512

25.11 RAUM RoarDMX

25.12 RIFF/Wave PCM

25.13 Native RoarSpeex

25.14 Native RoarCELT

25.15 Native DMX512

25.16 Native RoarDMX

25.17 Native FLAC

25.18 Native MIDI

25.19 Native VCLT (Vorbis Comment Like Text)

25.20 Native RALT (RoarAudio Like Text)

25.21 Native RALB (RoarAudio Like Binary)

Part VI

Maintainer Manual

Chapter 26

Gernal Information

Chapter 27

Packaging

Chapter 28

Configuration Nodes

Part VII

Appendix

Chapter 29

Autors and Copyright

The main Autor is Philipp "ph3-der-loewe" Schafft <lion@lion.leolix.org>.

archlinux part is written by kalasmannen <kalasmannen@gmail.com>.

Chapter 30

Contact information template

In case you help developing RoarAudio or be a Maintainer it is usefull to have full contact information about you so the Development team can contact you.

A full contact information include:

1. Your Nickname,
2. Your full Real Name,
3. Your E-Mail address,
4. Your OpenPGP Key's fringerprint.

Index

List of Figures

List of Tables