Contents

I Introduction 6

1 What is RoarAudio? 8
  1.1 What is a Sound Server? 8
  1.2 What are the key features of RoarAudio? 8

2 Basic concept 10

II Quickstart 11

3 Installation 12
  3.1 archlinux 12
  3.2 OpenBSD 13
  3.3 Debian 13
  3.4 From Sources 14

4 Configuring 15
  4.1 archlinux 15
  4.2 OpenBSD 15
  4.3 Debian 15
  4.3.1 Options 15

5 Setting up a player 17
  5.1 XMMS 17
  5.2 libao based 17
  5.3 Other 17

III User Manual 18

6 RoarAudio’s Architecture 19
  6.1 Subsystems 19
  6.2 Clients 19
  6.3 Streams 19
    6.3.1 Stream Types 20
    6.3.2 Stream Flags 22
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Driver</td>
<td>24</td>
</tr>
<tr>
<td>6.5 Sources</td>
<td>24</td>
</tr>
<tr>
<td>6.6 Codefilter</td>
<td>24</td>
</tr>
<tr>
<td>6.7 Bridges</td>
<td>24</td>
</tr>
<tr>
<td>7 Playing music on command lion with RoarAudio</td>
<td>25</td>
</tr>
<tr>
<td>7.1 roarcatplay</td>
<td>25</td>
</tr>
<tr>
<td>7.2 roarvorbis</td>
<td>25</td>
</tr>
<tr>
<td>7.3 roarradio</td>
<td>26</td>
</tr>
<tr>
<td>7.4 roarcat</td>
<td>26</td>
</tr>
<tr>
<td>8 Controlling played music</td>
<td>27</td>
</tr>
<tr>
<td>8.1 List current streams</td>
<td>27</td>
</tr>
<tr>
<td>8.2 Change volume</td>
<td>27</td>
</tr>
<tr>
<td>8.3 Kick a stream</td>
<td>28</td>
</tr>
<tr>
<td>8.4 Show and change meta data</td>
<td>28</td>
</tr>
<tr>
<td>8.5 Change stream flags</td>
<td>28</td>
</tr>
<tr>
<td>9 Dumping and Streaming</td>
<td>29</td>
</tr>
<tr>
<td>9.1 roarmon</td>
<td>29</td>
</tr>
<tr>
<td>9.2 roarshout</td>
<td>29</td>
</tr>
<tr>
<td>10 RoarAudio Daemon</td>
<td>30</td>
</tr>
<tr>
<td>10.1 General Audio Options</td>
<td>30</td>
</tr>
<tr>
<td>10.2 Drivers and Outputs</td>
<td>31</td>
</tr>
<tr>
<td>10.3 Sources</td>
<td>32</td>
</tr>
<tr>
<td>10.4 Codefilter</td>
<td>33</td>
</tr>
<tr>
<td>10.5 Listen Connection</td>
<td>33</td>
</tr>
<tr>
<td>10.6 Realtime</td>
<td>34</td>
</tr>
<tr>
<td>10.7 Security</td>
<td>34</td>
</tr>
<tr>
<td>10.7.1 User and group</td>
<td>34</td>
</tr>
<tr>
<td>10.7.2 chroot</td>
<td>34</td>
</tr>
<tr>
<td>11 Using RoarAudio's MIDI Subsystem</td>
<td>35</td>
</tr>
<tr>
<td>11.1 Basics about MIDI and RoarAudio</td>
<td>35</td>
</tr>
<tr>
<td>11.2 MIDI Clock</td>
<td>35</td>
</tr>
<tr>
<td>11.3 Connecting devices to RoarAudio</td>
<td>36</td>
</tr>
<tr>
<td>12 Using RoarAudio for Light Control</td>
<td>37</td>
</tr>
<tr>
<td>12.1 Basics about RoarAudio and Light Control</td>
<td>37</td>
</tr>
<tr>
<td>12.2 Connecting devices to RoarAudio</td>
<td>37</td>
</tr>
<tr>
<td>13 Compatibility Libraries</td>
<td>38</td>
</tr>
<tr>
<td>13.1 Using Compatibility Libraries</td>
<td>38</td>
</tr>
<tr>
<td>13.1.1 roarify</td>
<td>38</td>
</tr>
<tr>
<td>13.2 Enlightened Sound Daemon</td>
<td>39</td>
</tr>
<tr>
<td>13.3 PulseAudio</td>
<td>39</td>
</tr>
<tr>
<td>13.4 aRts - KDE Sound System</td>
<td>39</td>
</tr>
<tr>
<td>13.5 YIFF Sound System</td>
<td>39</td>
</tr>
<tr>
<td>13.6 OpenBSD sndio</td>
<td>39</td>
</tr>
</tbody>
</table>
# CONTENTS

## 14 End user Tools

14.1 roarcat ........................................... 40  
14.2 roarcatplay ........................................... 40  
14.3 roarctl ........................................... 40  
14.4 roarradio ........................................... 40  
14.5 roarvorbis ........................................... 40  

## 15 Networking

15.1 Connection Types ........................................... 41  
15.2 Proxy Server ........................................... 41  

## 16 Files and Environment

16.1 General ........................................... 42  
16.1.1 Files ........................................... 42  
16.1.2 Environment ........................................... 42  
16.2 roard ........................................... 42  
16.2.1 Environment ........................................... 42  
16.3 RoarAudio Clients ........................................... 43  
16.3.1 Environment ........................................... 43  

## IV Developer Manual

## 17 Audio and format conventions

17.1 Channel Mapping ........................................... 45  
17.2 Channel positions ........................................... 45  
17.3 Sample and Frame representation ........................................... 46  

## 18 Writing software using RoarAudio

18.1 Writing audio output plugins ........................................... 47  
18.1.1 Writing basic very basic output ........................................... 47  
18.1.2 Adding support to set server ........................................... 49  
18.1.3 Preparing the plugin for meta data, mixer settings and so on ........................................... 49  
18.1.4 Adding support for meta data ........................................... 52  
18.1.5 Adding support for roard based mixing ........................................... 53  
18.2 Writing other output plugins ........................................... 53  
18.3 Writing input plugins ........................................... 53  
18.4 Writing mixer plugins/Mixer frontends ........................................... 53  
18.5 Accessing other (control) features ........................................... 53  

## 19 libroar

19.1 Low Level Protocol API ........................................... 54  
19.2 Medium Level Protocol API ........................................... 54  
19.3 High Level Protocol API ........................................... 54  
19.4 Simple API ........................................... 54  
19.5 Sockets ........................................... 54  
19.6 Buffer ........................................... 54  
19.7 Stack ........................................... 54  
19.8 VIO - Virtual Input Output ........................................... 54
## CONTENTS

23.3 u-Law ................................................. 80
23.4 Vorbis ............................................... 80
23.5 Speex ............................................... 80
23.6 FLAC ............................................... 81
23.7 MIDI ............................................... 81
23.8 CELT ............................................... 81
23.9 DMX512 ........................................... 81
23.10 RoarDMX ........................................ 81
23.11 VCLT (Vorbis Comment Like Text) .................. 81
23.12 RALT (RoarAudio Like Text) ........................ 82
23.13 RALB (RoarAudio Like Binary) ...................... 82

24 Container ........................................... 83
24.1 Ogg ................................................ 83
24.2 RAUM ............................................. 83
24.3 RIFF/Wave ......................................... 83

25 Mappings ............................................. 84
25.1 Ogg Vorbis ......................................... 84
25.2 Ogg Speex .......................................... 84
25.3 Ogg FLAC ......................................... 84
25.4 Ogg CELT ......................................... 84
25.5 Ogg RoarDMX ...................................... 84
25.6 RAUM Mapped Ogg .................................. 84
25.7 RAUM PCM, A-Law, u-Law .......................... 87
25.8 RAUM RoarSpeex ................................... 87
25.9 RAUM RoarCELT .................................... 87
25.10 RAUM DMX512 ..................................... 87
25.11 RAUM RoarDMX .................................... 87
25.12 RIFF/Wave PCM ................................... 87
25.13 Native RoarSpeex .................................. 87
25.14 Native RoarCELT ................................... 87
25.15 Native DMX512 .................................... 87
25.16 Native RoarDMX .................................... 87
25.17 Native FLAC ....................................... 87
25.18 Native MIDI ....................................... 87
25.19 Native VCLT (Vorbis Comment Like Text) ......... 87
25.20 Native RALT (RoarAudio Like Text) ............... 87
25.21 Native RALB (RoarAudio Like Binary) ............. 87

VI Maintainer Manual ............................... 88

26 General Information ............................... 89

27 Packaging ........................................... 90

28 Configuration Nodes .............................. 91
CONTENTS

VII Appendix 92
29 Authors and Copyright 93
30 Contact Information Template 94
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 similar software.

1.1 What is a Sound Server?

A Sound Server or Sound Daemon is an application mixing audio in realtime. It is normally used to mix audio from different applications before 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 features 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 its user repository, the AUR. There are a few ways to obtain it from there and installing it. First of all, it requires you to have a set of development tools, since it’s essentially building it from the sources, and making 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 automatically 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. Yaourt will perform most steps automatically 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 (requires 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 actually 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.
CHAPTER 3. INSTALLATION

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 normally no configuration is needed at all. On some Systems there will be a config file to provide some options.

Clients: The clients normally have their own config (for example most players have a configure dialog). If a client does not provide an option it is taken from the Environment or a config file as described in chapter Files and Environment 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 normally be empty.

ROARD_REALTIME This controls if roard runs in realtime mode. Possible values are "NO", "YES" and "DOUBLE". Use "DOUBLE" to select a very realtime mode.

ROARAUDIO_DEFAULT_SOCKET This can be used to set a global default socket name. This is normally only useful for client only setups.

ROARD_AF This sets the Address Family to use for the listening socket. Valid values are "UNIX", "TCP" and "DECnet".
CHAPTER 4. CONFIGURING

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 able to use roard, too. Normally 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 Libraries. See table 5.3 for a list of tested players.

<table>
<thead>
<tr>
<th>Player</th>
<th>recommend library</th>
<th>working library</th>
</tr>
</thead>
<tbody>
<tr>
<td>mplayer</td>
<td>esd</td>
<td>esd</td>
</tr>
<tr>
<td>XINE</td>
<td>esd</td>
<td>esd</td>
</tr>
<tr>
<td>Amarok</td>
<td>esd</td>
<td>esd</td>
</tr>
<tr>
<td>Helix Player</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Totem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rhythmbox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLC</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Part III

User Manual
Chapter 6

RoarAudio’s Architecture

6.1 Subsystems

Subsystems are internal parts within roard handling different kinds of data.

<table>
<thead>
<tr>
<th>Subsystem Name</th>
<th>Domain</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waveform</td>
<td>time</td>
<td>This is the main subsystem. This handles Waveform signals in time domain. This are normal PCM streams.</td>
</tr>
<tr>
<td>MIDI</td>
<td>frequency</td>
<td>This is the MIDI subsystem. It’s mainly used to control external MIDI devices.</td>
</tr>
<tr>
<td>Console Beep</td>
<td>frequency</td>
<td>This is in fact a subsystem of the MIDI subsystem. It is used to control the speaker on architectures with a system speaker.</td>
</tr>
<tr>
<td>Light</td>
<td>frequency</td>
<td>This is used to control theater light systems. Typically DMX is used.</td>
</tr>
</tbody>
</table>

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 multiple audio streams open at a time.

A special case is roard itself as it is a client of itself to provide a standard interface for output streams like soundcards.

6.3 Streams

A stream is a stream of audio data send from an application or device to roard or from roard to an application or device. A stream is always 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 internally uses (normally 10ms). Because of this an applications needs to be careful 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 gets all data you send to another input stream without any change. The stream gets 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 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 flagged with the primary flag dies (simply end or get kicked) roard shuts down itself. This is normally used on output streams to soundcards. This flag can be set on a already running stream.

case

roard uses streams with the sync flag as clock source. There may be multiple 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. Normally 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 technical meaning. This flag can not be set on a already running stream.

meta

This flag controls the behavior of the meta data on the stream. The behavior 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 output 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 behavior 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 before the driver module is loaded.
cleanmeta

This flag enables an automatic cleanup of meta data on the stream. This may be used to strip station name on webradio streams. This flag can be set on an 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 similar to their own pause state. Some players get broken when using this. The main purpose is to ensure there is no data transferred 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 does 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 silent). It stops roard to mix the data of the stream into the others. Because of this, this is more effective than setting 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 assumption 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 $O_c(s) = M(s) - I_c(s)$.

$$M(s) = \frac{\sum_{c=0}^{n} I_c(s) * e^{j*2\pi}}{n}$$  \hspace{1cm} (6.1)
virtual
This stream is a child of another stream.

resource
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 special ioctl(s).

6.5 Sources

Sources are normal streams. The only differences from a stream created by an 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 behavior.

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 line with RoarAudio

There are mainly three different tools to play music on command line 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` supposed 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 interesting for use in scripts.

7.1 roarcatplay

`roarcatplay` requires normally 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 message. The detection requires that file file is seekable thus this does not work on pipes. See `roarcat` for a tool usable 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 locally 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 normally used to play long living streams like webradio. It supports HTTP thru wget.

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

For more information see End user Tools: roarradio.

7.4 roarcat

roarcat is normally 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 line 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 control 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 different 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 tries 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 tries to find out which channels are left ones and which are right ones and set both groups to the given values.
The volume can be given in two ways: as an absolute 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 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 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 monitoring clients. It may also decode high level codecs via a codeclifier.

### 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 normally 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 44100 Hz, 2 channels (stereo) and 16bit per sample. The sample rate may be set to any value from 1Hz to > 1 MHz. The number of channels can be set to a value from 1 up to a compiled in maximum value. This maximum is typically 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 automatically 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 different drivers to support different 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.

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

Table 10.1: Possible output drivers
Outputs are controlled 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 expects.

Examples:

\[
\begin{align*}
\text{\$ roard -o oss -O /dev/audio5} \\
\text{\$ roard -o oss -O /dev/audio5 -oN -o roar -O another.host}
\end{align*}
\]

\(-oO\) 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.

<table>
<thead>
<tr>
<th>Option</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rate</td>
<td>Integer</td>
<td>Sample rate in Hz</td>
</tr>
<tr>
<td>channels</td>
<td>Integer</td>
<td>Number of channels</td>
</tr>
<tr>
<td>bits</td>
<td>Integer</td>
<td>Bits per sample</td>
</tr>
<tr>
<td>codec</td>
<td>String</td>
<td>Codec to use</td>
</tr>
<tr>
<td>blocks</td>
<td>Integer</td>
<td>Number of blocks used in jitter buffer of device. This is not supported by all drivers.</td>
</tr>
<tr>
<td>blocksize</td>
<td>Integer</td>
<td>Size of blocks used for device buffer</td>
</tr>
<tr>
<td>meta</td>
<td>Flag</td>
<td>Sets flag meta</td>
</tr>
<tr>
<td>sync</td>
<td>Flag</td>
<td>Sets flag sync</td>
</tr>
<tr>
<td>primary</td>
<td>Flag</td>
<td>Sets flag primary. This is the same as (-oP).</td>
</tr>
<tr>
<td>cleanmeta</td>
<td>Flag</td>
<td>Sets flag cleanmeta</td>
</tr>
<tr>
<td>autoconf</td>
<td>Flag</td>
<td>Sets flag autoconf</td>
</tr>
</tbody>
</table>

Table 10.2: Possible output options

Example:

\[
\text{\$ roard -o oss -oP -oO channels=1,bits=8,sync}
\]

10.3 Sources

Sources are input streams created by roard. They may for example be used to play startup sounds. The options are very similar 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:

\[
\text{\$ roard -S startup.wav -sN -S welcome.ogg}
\]
CHAPTER 10. ROARAUDIO DAEMON

-\texttt{-sC} and -\texttt{-sO} set the container type used and options for the new stream. -\texttt{-sC} is currently not used by any source. -\texttt{-sO} normally only have one possible option \texttt{codec}.

$ \texttt{roard -S audiodump.alaw -oO codec=alaw}$

-\texttt{-sP} is the same as -\texttt{-oP} just for sources: it sets the primary flag on the stream. This will terminate roard in case the stream ends. This behavior 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 \texttt{--list-cf}:

$ \texttt{roard --list-cf}$

10.5 Listen Connection

\texttt{roard} supports normally different types of sockets. You can use the options -\texttt{-u}, -\texttt{-n} and -\texttt{-t} to select socket type Unix Domain Socket, DECnet and TCP. Some socket types may not be supported on your operating system. Default is to use Unix Domain Sockets if possible.

\textbf{In case of TCP} you can select IPv4 and IPv6 via -\texttt{-4} and -\texttt{-6}. IPv6 is not fully supported yet.

\textbf{On Unix Domain Sockets} you can set the socket filename via \texttt{--sock}.

Example:

$ \texttt{roard --sock /tmp/roard-42}$

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

Example:

$ \texttt{roard -n -b ::roar42}$

\textbf{On TCP sockets} you can set the hostname to bind to via -\texttt{-b} and the port with the option -\texttt{-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:

$ \texttt{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 permissions 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 results in the possibly 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 reality 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 latency 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 second 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 libraries 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 describes 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 environment 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 available or work on all Operating Systems. It should work on all POSIX systems but at least used mechanisms are not supported by M$ Window$. Please do not run this Operating System.

To use roarify on a application you can simply use:

$ roarify myplayer

This is normally 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 overridden 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

<table>
<thead>
<tr>
<th>Socket Type</th>
<th>Examples</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>/path/to/socket</td>
<td>/tmp/roar, $HOME/.roar</td>
</tr>
<tr>
<td>DECnet</td>
<td>mynode::, ::myobj, node::obj</td>
<td>0.0::roar</td>
</tr>
<tr>
<td>IPv4</td>
<td>myserver.dom, server.dom:port</td>
<td>localhost:roar(16002)</td>
</tr>
<tr>
<td>IPv6</td>
<td>ipv6-localhost:roar(16002)</td>
<td></td>
</tr>
</tbody>
</table>

Table 15.1: Examples of Socket Addresses

15.2 Proxy Server
Chapter 16

Files and Environment

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 Environment

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 Environment

ROAR_DRIVER

$ROAR_DRIVER sets the output driver of roard. This is the same as roard’s option -o.
CHAPTER 16. FILES AND ENVIROMENT

ROAR_DEVICE

$ROARDEVICE sets the output driver’s device of roard. This is the same as roard’s option -O.

16.3 RoarAudio Clients

16.3.1 Environment

ROAR_PROXY

$ROAR_PROXY sets the type of a proxy used to connect to somewhere. The default is an empty 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 certain 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 different mapping you need to swap them.

<table>
<thead>
<tr>
<th>Number of Channels</th>
<th>Mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0: Mono Mid Channel</td>
</tr>
<tr>
<td>2</td>
<td>0: Left, 1: Right</td>
</tr>
<tr>
<td>2 Mid-Side</td>
<td>0: Mid, 1: Side</td>
</tr>
<tr>
<td>3</td>
<td>0: Left, 1: Right, 2: Center</td>
</tr>
<tr>
<td>4</td>
<td>0: Front Left, 1: Front Right, 2: Rear Left, 3: Rear Right</td>
</tr>
<tr>
<td>5</td>
<td>0: Front Left, 1: Front Right, 2: Center, 3: Rear/Surround Left, 4: Rear/Surround Right</td>
</tr>
<tr>
<td>6</td>
<td>0: Front Left, 1: Front Right, 2: Center, 3: LFE, 4: Rear/Surround Left, 5: Rear/Surround Right</td>
</tr>
</tbody>
</table>

Table 17.1: Channel Mapping

17.2 Channel positions

Table 17.2 shows where every channel is located.

<table>
<thead>
<tr>
<th>Front</th>
<th>Center</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRONT_LEFT</td>
<td>FRONT_CENTER CENTER/MONO</td>
<td>FRONT_RIGHT</td>
</tr>
<tr>
<td>Side</td>
<td>SIDE_LEFT</td>
<td>SIDE_CENTER</td>
</tr>
<tr>
<td>Back</td>
<td>BACK_LEFT</td>
<td>BACK_CENTER</td>
</tr>
</tbody>
</table>

Table 17.2: Channel positions
The channel $SIDE\_CENTER$ is only listed to be complet. It can not exist mathematically nor in reality as it is somewhere within the listener’s head.

Position 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 normally 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 significant bits. Internal operations and calculations happen with native byte order. Data send to the network or given to some program may be in a different 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.

<table>
<thead>
<tr>
<th>Frame 0: Left</th>
<th>Frame 0: Right</th>
<th>Frame 1: Left</th>
<th>Frame 1: Right</th>
<th>Frame n: ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 private 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 object we need to store in the instance structure which may look like this:

```c
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 therefore it needs to know about the format of the input samples. In most cases they are already PCM in native byte order. In this case we only need to take care about the number of channels, the number of bits per sample and the sample rate.

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

```c
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);
}

Finally 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;
18.1.2 Adding support to set server

There are basically two methods of setting the server:

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

The first one is a good choice in case you know that there is only peace of software with only one instance using liberoar. 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 already not true.

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

```c
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 can not assure that there are other instances that use liberoar 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 similar function.

18.1.3 Preparing 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 separate the so called control 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 cannot be used to manipulate metadata, the mixer settings or other more advanced properties of the stream. It of course will keep as used to send the raw data to the server.

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

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

**Most players** have separate 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:

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

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

    // set stream to something invalid 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 instance 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 automatically 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 prepared to support all kinds 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 documentation and implement some special 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 basically. The type can for example be 'Artist', 'Title' or 'Album'. A stream can have multiple meta data entries of a single type. This may for example be the case if some song has two performers.

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

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

```c
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);

// finally 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 already features most of the meta data manipulations: clearing meta data, adding meta data, setting meta data and finally committing it.

You may have noticed that we set the title with ROAR_META_MODE_SET but the artist with ROAR_META_MODE_ADD. The difference between both is that ROAR_META_MODE_SET overwrites all existing meta data entries 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:

```c
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:

```c
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:

```c
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 before returning. In non-blocking mode they will return as soon as possible. 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:

```c
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 an object is synced data in buffers are written to disk or network. Input buffers may also be changed. The exact behavior depends on the VIO Object’s type.

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

Parameters:

vio The VIO Object to sync.

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

Examples:

```c
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 to do 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 should set this to NULL.

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

```c
int fh;
roar_vio_ctl(vio, ROAR_VIO_CTL_GET_FH, &fh);
roar_vio_close(3)
```

Synopsis:

```c
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:

```c
roar_vio_close(vio);
```

### 19.8.2 VIO Types

**File API**

Synopsis:

```c
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:
CHAPTER 19. LIBROAR

File Handle API

Synopsis:

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

Description: This opens an already 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 already 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:

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 special 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:

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

Description: The re VIO API is very similar to the pass API. The only difference is that it will retry read and write operations until all data is read or written or an error occurs.

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:

```c
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:

```c
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);
```
CHAPTER 19. LIBROAR

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:

```c
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:

```c
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:**

```c
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:**

```c
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:**

```c
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,
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 second paragraph is a multi lion comment (if possible) including a copyright statement for each author (even if the author done only trivial changes! Trivial changes may be marked as such) followed by a license statement.

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;
}

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.

Exceptions:
  • 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);
CHAPTER 21. PROJECT’S CODING STYLE AND CONVENTIONS

//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.
CHAPTER 21. PROJECT’S CODING STYLE AND CONVENTIONS

//example.c:

/*
 * This is a big multi lion
 * comment describing 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;
}

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 spacial 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 roarBuffers.

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.
CHAPTER 21. PROJECT’S CODING STYLE AND CONVENTIONS  70

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 a minimum size you should not force exact size but use the `int_leastN_t`-types.

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Safe value range</th>
<th>Common alternatives</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>−32768..32767</td>
<td>int32_t, int_least32_t</td>
</tr>
<tr>
<td>short</td>
<td>−128..127</td>
<td>int16_t, int_least16_t</td>
</tr>
<tr>
<td>long int</td>
<td>−32768..32767</td>
<td>int32_t, int_least32_t, size_t</td>
</tr>
<tr>
<td>long long int</td>
<td>−32768..32767</td>
<td>int64_t, int_least64_t, size_t</td>
</tr>
<tr>
<td>char</td>
<td>−128..127</td>
<td>-/-</td>
</tr>
<tr>
<td>void *</td>
<td>NULL, any pointer</td>
<td>-/-</td>
</tr>
<tr>
<td>size_t</td>
<td>0..n</td>
<td>-/-</td>
</tr>
<tr>
<td>ssize_t</td>
<td>−1..m</td>
<td>-/-</td>
</tr>
</tbody>
</table>

Table 21.1: Safe value ranges of data types

21.12 Order of arguments

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

Wrong:

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

Right:

```c
int conv(void * out, void * in, size_t len);
```
Chapter 22

RoarAudio Protocol

22.1 RoarAudio Protocol and OSI Layer Model

<table>
<thead>
<tr>
<th>Layer</th>
<th>Name</th>
<th>Object Type</th>
<th>Module</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Application</td>
<td>Stream</td>
<td>Mixer</td>
<td>Your favorite song</td>
</tr>
<tr>
<td>6</td>
<td>Presentation</td>
<td>Codec</td>
<td>Codec Filter</td>
<td>PCM, A-Law, Vorbis, ...</td>
</tr>
<tr>
<td>5</td>
<td>Session</td>
<td>Client, Message</td>
<td>Control Logic</td>
<td>Commands: QUIT, NEW_STREAM, ...</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
<td>VIO, Socket</td>
<td>IO</td>
<td>TCP, NSP, ...</td>
</tr>
<tr>
<td>3</td>
<td>Network</td>
<td>VIO</td>
<td>IO</td>
<td>IP, DRP, ...</td>
</tr>
<tr>
<td>2</td>
<td>Data link</td>
<td>Controller</td>
<td></td>
<td>Ethernet, RS232, I2C, CAN</td>
</tr>
<tr>
<td>1</td>
<td>Physical</td>
<td>None</td>
<td></td>
<td>Wire, Fiber, Wireless</td>
</tr>
</tbody>
</table>

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 I²C as on RS232. With two differences:

1. Each message needs to be packed into a single I²C message framed by START and STOP.
2. Each I²C message containing a RoarAudio message starts with the source address of the node sending the message directly after the destination address as specified by I²C. If the size of the source address is not a multiple of 8 bit then it will be transmitted as the next bigger multiple 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 multiple independent connections from a single I²C node. This means that a single I²C node may act as if it is multiple nodes with multiple node addresses. This is for example needed in case of a bridge between I²C and another bus.

### 22.3.4 CAN

If no data length is given then data length is calculated using formula 22.1. This means that messages with a total length \( \leq 8 \) 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 larger 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 consist of two parts: The header and a body part. The header has a fixed size of 10 Byte. The body has a variable 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 are in network byte order.
22.4.2 Format version 1

The Protocol Version 1 is in draft currently. It is not supposed to replace the protocol version but to extend it. It is designed for byte oriented serial communication. It extends the version 0 format by adding an additional flags field. Most of the other fields can be shortened 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.

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 Position field (4 byte) is only used by a very small group of commands the average message size is smaller.

The meaning of 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 (normally OSI layer 1 or 2). The data length is calculated by formula 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 padding is used a Data length field in the header must be provided.

\[ \text{PacketLength} - \text{HeaderLength} = \text{DataLength} \quad (22.1) \]
CHAPTER 22. ROARAUDIO PROTOCOL

<table>
<thead>
<tr>
<th>Bit</th>
<th>Flag Name</th>
<th>Long flag name</th>
<th>Flag Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>MF</td>
<td>Meta Framing</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ED</td>
<td>Error Detection</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>LDL</td>
<td>Long Data Length</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LSPOS</td>
<td>Long Stream Position</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LSID</td>
<td>Long Stream ID</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DL</td>
<td>Data Length</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>SPOS</td>
<td>Stream Position</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>SID</td>
<td>Stream ID</td>
<td></td>
</tr>
</tbody>
</table>

Table 22.4: Message flags for format version 1 messages

If Meta Framing is set there are two possibilities:
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 provided 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 formula 22.2.

\[ SID + 2 \times SPOS + 4 \times LSID + 8 \times LSPOS = \text{Frame ID} \]  

The Frame ID gets 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 resonable way. Implementations receiving framed messages must not relay on this.
CHAPTER 22. ROARAUDIO PROTOCOL

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version (0x00)</td>
<td>Flags</td>
<td>Data Length</td>
<td>Data...</td>
<td>0-7</td>
</tr>
<tr>
<td>Data...</td>
<td>Error Detection</td>
<td>- end</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 22.5: Format of version 0 continued Meta Frames

### 22.4.3 Format version 2

DRAFT — DRAFT — DRAFT

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version (0x02)</td>
<td>Command</td>
<td>SID</td>
<td>0-3</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>4-7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream position</td>
<td>8-11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stream position</td>
<td>4 byte extension</td>
<td>(12-15)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data length</td>
<td>Dir Seq</td>
<td>12-15(16-19)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data ...</td>
<td></td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CRC ...</td>
<td></td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Size¹</th>
<th>Headers²</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOOP</td>
<td>0..Max/0..Max</td>
<td>gpos</td>
<td>No Operation.</td>
</tr>
<tr>
<td>1</td>
<td>IDENTIFY</td>
<td>/0</td>
<td>gpos</td>
<td>Identify a connection</td>
</tr>
<tr>
<td>2</td>
<td>AUTH</td>
<td>0/0</td>
<td>gpos</td>
<td>Auth a connection</td>
</tr>
<tr>
<td>6</td>
<td>QUIT</td>
<td>0/-</td>
<td>None/-</td>
<td>Terminates the connection.</td>
</tr>
<tr>
<td>31</td>
<td>GETTIMEOFDAY</td>
<td>None.</td>
<td>Get a timestamp from the server</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>WHOAMI</td>
<td>0/1*</td>
<td>gpos</td>
<td>Asks server about information about this connection</td>
</tr>
</tbody>
</table>

Table 22.7: Table of base and connection commands

Entries marked with * are subject to change, entries marked with . are drafted.
NOOP

The No-Operation command does not do any operation on the server side except 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 original data should be send back to the client. A different size than zero or the size 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 authentication 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 commands 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 response commands

This section describes responses for requests of the client. The following commands MUST NOT be send as request.
## Table 22.8: Table of response commands

<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Size</th>
<th>Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>252</td>
<td>EPERM</td>
<td>-/-</td>
<td>-/-</td>
<td>Internal, obsolete. Permission error.</td>
</tr>
<tr>
<td>253</td>
<td>OK_STOP</td>
<td>-/0..Max</td>
<td>-/any</td>
<td>Internal. Same as OK but do not process another message of this client before next main cycle</td>
</tr>
<tr>
<td>254</td>
<td>OK</td>
<td>-/0..Max</td>
<td>-/any</td>
<td>Command succeeded with no error</td>
</tr>
<tr>
<td>255</td>
<td>ERROR</td>
<td>-/0</td>
<td>-/None</td>
<td>Some error occurred</td>
</tr>
</tbody>
</table>

Entries marked with * are subject to change, entries marked with . are draft.

**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. Invalid parameter from client,
2. All kinds of IO errors on the server side,
3. Errors from the Operating system on the server,

### 22.5.3 Server Status commands

### 22.5.4 Stream control and data commands

### 22.5.5 Other commands
CHAPTER 22. ROARAUDIO PROTOCOL

<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Size</th>
<th>Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>GET_STANDBY</td>
<td>0/2*</td>
<td>gpos</td>
<td>Get information about the current standby state</td>
</tr>
<tr>
<td>8</td>
<td>SET_STANDBY</td>
<td>2*/0</td>
<td>gpos</td>
<td>Set the current standby state</td>
</tr>
<tr>
<td>9</td>
<td>SERVER_INFO</td>
<td>0/</td>
<td>None.</td>
<td>Get information about the server</td>
</tr>
<tr>
<td>10</td>
<td>SERVER_STATS</td>
<td>0/</td>
<td>None.</td>
<td>Get stats from the server</td>
</tr>
<tr>
<td>11</td>
<td>SERVER_OINFO</td>
<td>0/</td>
<td>None.</td>
<td>Get stream info for the mixing stream</td>
</tr>
<tr>
<td>13</td>
<td>EXIT</td>
<td>0/1/0</td>
<td>gpos</td>
<td>Terminates the server.</td>
</tr>
<tr>
<td>25</td>
<td>GET_ACL</td>
<td></td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>SET_ACL</td>
<td></td>
<td>None.</td>
<td></td>
</tr>
</tbody>
</table>

Table 22.9: Table of server status and control commands
Entries marked with * are subject to change, entries marked with . are drafted.

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

Table 22.10: Table of stream commands
Entries marked with * are subject to change, entries marked with . are drafted.

<table>
<thead>
<tr>
<th>#</th>
<th>Command</th>
<th>Size</th>
<th>Headers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>LIST_CLIENTS</td>
<td></td>
<td>None.</td>
<td>List all clients</td>
</tr>
<tr>
<td>16</td>
<td>GET_CLIENT</td>
<td>1*</td>
<td>None.</td>
<td>Get informations about a client</td>
</tr>
<tr>
<td>18</td>
<td>KICK</td>
<td>4/0</td>
<td>None.</td>
<td>Kick a object from the server</td>
</tr>
<tr>
<td>24</td>
<td>BEEP</td>
<td></td>
<td></td>
<td>Internal. End of List</td>
</tr>
</tbody>
</table>

Table 22.11: Table of other commands
Entries marked with * are subject to change, entries marked with . are drafted.
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 document-
tation out there about it. As a start you can read the Wikipedia article at:

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:

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 telecommunication systems. See Wikipedia’s article for more inforatton:

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

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

23.9 DMX512

The DMX512 codec is the most simple way 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. Multiple 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 frequency is \( cfreq = 100Hz \) the used data rate is \( 51.2kB/s \). See formula (23.1) for details.

\[
\text{DataRate}[\frac{\text{Byte}}{s}] = cfreq \times 512\text{Byte} \tag{23.1}
\]

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 received.

Frames must always be written 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 metadata. 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' (0xA), the strings are encoded as UTF-8.
CHAPTER 23. CODECS

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 transmission 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 able to store data of most of the codecs supported by RAUM in an 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 below). 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 (except in the meta data block) are in ISO-8859-15 however they should be kept high-bit clean meaning 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:

<table>
<thead>
<tr>
<th>byte</th>
<th>'R'</th>
<th>'A'</th>
<th>'M'</th>
<th>0-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>'M'</td>
<td>'O'</td>
<td>Null byte (0x00)</td>
<td>Version (0x00)</td>
</tr>
<tr>
<td>Flags (32 bit)</td>
<td></td>
<td></td>
<td>8-11</td>
<td></td>
</tr>
<tr>
<td>Codec ID</td>
<td></td>
<td></td>
<td>12-15</td>
<td></td>
</tr>
<tr>
<td>Null byte terminated codec name</td>
<td></td>
<td></td>
<td>1-</td>
<td></td>
</tr>
</tbody>
</table>

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 supposed 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 supported but supports different 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 first, 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 integers to store all the needed parameters. In case a parameter is of no interest 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 position ID (may be ignored by players)
3. Sample rate in 1/1024 of Hz.
4. Bits per Sample/pixel.
5. Number of channels. A Video may set this to the total number of pixels \((ch = x \cdot y \cdot z)\).
6. Width (x) in pixel.
7. Height (y) in pixel.
8. Deep (z) in pixel. A 2D Video has set this to one.
CHAPTER 25. MAPPINGS

9. The number of frames plus one between each key frame. If every frame
is a keyframe this number 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 average but with flipped sign (negative). In case of unknown keyframe
rate the value is set to zero.

All the tailing zeroed parameters can be skipped to save storage. The
length of the packet then indicates which parameters are included. For example
most audio files will only need the first 5 values be set so the size of the packet
is \( size = 5 \text{ints} \times 4 \text{Byte/int} = 20 \text{Byte} \). If the packet is longer than supported
an therefore 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. Enabling this option per default is against
this specification.

The meta data header contains a set of Vorbis comments as specified in the
Vorbis documentation at \( \text{http://www.xiph.org/vorbis/doc/v-comment.html} \).

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 except that the higher 32 bit are used to count the overflows
with the exception that the most significant bit needs to be cleared (zero) all
the time (negative values are reserved for future use).
CHAPTER 25. MAPPINGS

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>.

This document is licenced under Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported. The license text can be found at http://creativecommons.org/licenses/by-nc-sa/3.0/.
Chapter 30

Contact information template

In case you help developing RoarAudio or be a Maintainer it is useful 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 fingerprint.
Index

μ-Law, 49
A-Law, 49
Add, 49
AMP, 49
API, 37
archlinux, 7, 10
aRts, 30
AUR, 7

BIO, 45
Bits, 14, 24, 49
Bits per sample, 14, 24, 49
Bridge, 18
BSD, 3
Buffer, 37

CAN, 55, 56
Channel Mapping, 35
Channels, 14, 24, 49
chroot, 28
Client, 14, 55
Clip, 49
Codec, 18, 26, 55
Codecfiler, 18, 26
Codecs, 49
coding style, 52
Command, 45, 55
Command API, 45
Commands, 59
Compatibility Librarys, 30
compression, 46
Config, 10
Console Beep, 14
conventions, 35, 52
cryptography, 45, 46
cygwin, 3

Data Length, 57
DCBlock, 49
Debian, 8, 10
DECnet, 31

Descriptive String, 47, 48
DL, 58
DMX, 14
downmix, 49
Driver, 18, 25
DRP, 55
DSTR, 47, 48
DSTR - Descriptive String API, 47, 48
Dumping, 23

ED, 58
encryption, 45, 46
Enlightened Sound Daemon, 3, 30
Enviroment, 32, 33
ESD, 3, 30
esdcat, 20
EsounD, 3, 30
Ethernet, 55

fd, 41, 42
fh, 41, 42
Fiber, 55
File API, 40
File Handle API, 41
Files, 32
Filter, 49
Filterchains, 49
Frame, 35
Frame ID, 58

GID, 27
GNU/Linux, 3
GnuPG, 46
Gopher, 44, 48
gpg, 46
Group, 27
gzip, 46
gzip API, 46

High Level, 37
High Level Protocol API, 37
Highpass, 49
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>37, 44, 48</td>
</tr>
<tr>
<td>HTTP CONNECT</td>
<td>37</td>
</tr>
<tr>
<td>fC</td>
<td>55, 56</td>
</tr>
<tr>
<td>Icecast</td>
<td>3, 23</td>
</tr>
<tr>
<td>IP</td>
<td>31, 55</td>
</tr>
<tr>
<td>IPv4</td>
<td>31</td>
</tr>
<tr>
<td>IPv6</td>
<td>31</td>
</tr>
<tr>
<td>KDE Sound System</td>
<td>30</td>
</tr>
<tr>
<td>LDL</td>
<td>58</td>
</tr>
<tr>
<td>libao</td>
<td>12</td>
</tr>
<tr>
<td>libao based</td>
<td>12</td>
</tr>
<tr>
<td>libroar</td>
<td>37</td>
</tr>
<tr>
<td>libroardsp</td>
<td>49</td>
</tr>
<tr>
<td>Light</td>
<td>14</td>
</tr>
<tr>
<td>Listening Socket</td>
<td>26</td>
</tr>
<tr>
<td>Low Level</td>
<td>37</td>
</tr>
<tr>
<td>Low Level Protocol API</td>
<td>37</td>
</tr>
<tr>
<td>Lowpass</td>
<td>49</td>
</tr>
<tr>
<td>LSID</td>
<td>58</td>
</tr>
<tr>
<td>LSPOS</td>
<td>58</td>
</tr>
<tr>
<td>Magic</td>
<td>45</td>
</tr>
<tr>
<td>Magic API</td>
<td>45</td>
</tr>
<tr>
<td>Mapping</td>
<td>35, 55</td>
</tr>
<tr>
<td>Medium Level</td>
<td>37</td>
</tr>
<tr>
<td>Medium Level Protocol API</td>
<td>37</td>
</tr>
<tr>
<td>Message</td>
<td>55, 56</td>
</tr>
<tr>
<td>Meta Data</td>
<td>19, 22</td>
</tr>
<tr>
<td>Meta Frame</td>
<td>58, 59</td>
</tr>
<tr>
<td>MF</td>
<td>58</td>
</tr>
<tr>
<td>MIDI</td>
<td>14, 49, 56</td>
</tr>
<tr>
<td>Mixer</td>
<td>55</td>
</tr>
<tr>
<td>Network</td>
<td>26, 31, 55</td>
</tr>
<tr>
<td>Networking</td>
<td>31</td>
</tr>
<tr>
<td>NSP</td>
<td>55</td>
</tr>
<tr>
<td>Ogg Vorbis</td>
<td>19</td>
</tr>
<tr>
<td>OpenBSD</td>
<td>8, 10, 30</td>
</tr>
<tr>
<td>sndio</td>
<td>30</td>
</tr>
<tr>
<td>OpenPGP</td>
<td>46</td>
</tr>
<tr>
<td>OpenPGP API</td>
<td>46</td>
</tr>
<tr>
<td>OpenSSL</td>
<td>45</td>
</tr>
<tr>
<td>OSI Layer Model</td>
<td>55</td>
</tr>
<tr>
<td>Output</td>
<td>3, 25, 32, 33</td>
</tr>
<tr>
<td>pacman</td>
<td>7</td>
</tr>
<tr>
<td>Pass</td>
<td>43</td>
</tr>
<tr>
<td>Pass API</td>
<td>43</td>
</tr>
<tr>
<td>Password API</td>
<td>48</td>
</tr>
<tr>
<td>PCM</td>
<td>14, 18, 20, 49</td>
</tr>
<tr>
<td>pinentry</td>
<td>48</td>
</tr>
<tr>
<td>Pipe</td>
<td>43</td>
</tr>
<tr>
<td>Pipe API</td>
<td>43</td>
</tr>
<tr>
<td>Player</td>
<td>12</td>
</tr>
<tr>
<td>POSIX</td>
<td>3</td>
</tr>
<tr>
<td>Protocol</td>
<td>44, 48, 55</td>
</tr>
<tr>
<td>Protocol API</td>
<td>44, 48</td>
</tr>
<tr>
<td>Protocol Mapping</td>
<td>55</td>
</tr>
<tr>
<td>Proxy Server</td>
<td>31, 33, 37</td>
</tr>
<tr>
<td>PulseAudio</td>
<td>30</td>
</tr>
<tr>
<td>Quantifier</td>
<td>49</td>
</tr>
<tr>
<td>Rate</td>
<td>14, 24, 49</td>
</tr>
<tr>
<td>Re</td>
<td>43</td>
</tr>
<tr>
<td>Re API</td>
<td>43</td>
</tr>
<tr>
<td>Real Streams</td>
<td>55</td>
</tr>
<tr>
<td>Realtime</td>
<td>27</td>
</tr>
<tr>
<td>Reply</td>
<td>55</td>
</tr>
<tr>
<td>Request</td>
<td>55</td>
</tr>
<tr>
<td>roar-config</td>
<td>50</td>
</tr>
<tr>
<td>ROAR_SOCKET_BLOCK</td>
<td>39</td>
</tr>
<tr>
<td>ROAR_SOCKET_NONBLOCK</td>
<td>39</td>
</tr>
<tr>
<td>roar_vio_close</td>
<td>40</td>
</tr>
<tr>
<td>roar_vio_ctl</td>
<td>39</td>
</tr>
<tr>
<td>ROAR_VIO_CTL_GET_FH</td>
<td>40</td>
</tr>
<tr>
<td>roar_vio_dstr_init_defaults</td>
<td>47</td>
</tr>
<tr>
<td>roar_vio_dstr_init_defaults_c</td>
<td>47</td>
</tr>
<tr>
<td>roar_vio_lseek</td>
<td>38</td>
</tr>
<tr>
<td>roar_vio_nonblock</td>
<td>38</td>
</tr>
<tr>
<td>roar_vio_open_bio</td>
<td>45</td>
</tr>
<tr>
<td>roar_vio_open_cmd</td>
<td>45</td>
</tr>
<tr>
<td>roar_vio_open_default</td>
<td>47</td>
</tr>
<tr>
<td>roar_vio_open_dstr</td>
<td>47</td>
</tr>
<tr>
<td>roar_vio_open_dstr_vio</td>
<td>47</td>
</tr>
<tr>
<td>roar_vio_open_fh</td>
<td>41</td>
</tr>
<tr>
<td>roar_vio_open_fh_socket</td>
<td>42</td>
</tr>
<tr>
<td>roar_vio_open_file</td>
<td>40</td>
</tr>
<tr>
<td>roar_vio_open_gzip</td>
<td>46</td>
</tr>
<tr>
<td>roar_vio_open_magic</td>
<td>45</td>
</tr>
<tr>
<td>roar_vio_open_pass</td>
<td>43</td>
</tr>
<tr>
<td>roar_vio_open_pgp_decrypt</td>
<td>46</td>
</tr>
<tr>
<td>roar_vio_open_pgp_encrypt_pub</td>
<td>46</td>
</tr>
<tr>
<td>roar_vio_open_pgp_encrypt_sym</td>
<td>46</td>
</tr>
<tr>
<td>roar_vio_open_pgp_store</td>
<td>46</td>
</tr>
<tr>
<td>roar_vio_open_pipe</td>
<td>43</td>
</tr>
<tr>
<td>roar_vio_open_proto</td>
<td>44</td>
</tr>
</tbody>
</table>
INDEX

roar_vio_open_re(3), 43
roar_vio_open_stack(3), 44
roar_vio_open_stdio(3), 42
roar_vio_proto_init_def(3), 44
roar_vio_read(3), 37
roar_vio_stack_add(3), 44
roar_vio_sync(3), 39
roar_vio_to_bio(3), 45
roar_vio_to_stdio(3), 42
roar_vio_write(3), 37
RoarAudio Client
  roar-config, 50
  roarbidir, 29
  roarcat, 29
  roarcat2sock, 29
  roarcatad, 51
  roarcatpassfh, 51
  roarcatplay, 29
  roarcatsendfile, 51
  roarcatvio, 29
  roarctl, 29
  roarfctest, 51
  roarfilt, 29
  roarfish, 29
  roarmon, 29
  roarmonhttp, 29
  roarradio, 29
  roarshout, 29
  roarsin, 29
  roarsockconnect, 51
  roarsocktypes, 50
  roartypes, 50
  roarvorbis, 19, 29
  roarvumeter, 29
  RS232, 55, 56

Sample, 35
Security, 27
select(), 48
Shell, 45
Shout, 23
SID, 58
Simple API, 37
sndio, 30
Socket, 26, 31, 37, 41, 42, 55
Socket API, 41
Socket Handle API, 42
SOCKS, 37
SOCKS4, 37
SOCKS4a, 37
SOCKS4d, 37
Sound Daemon, 3
Sound Server, 3
Source, 18, 25
Sources, 9
sox, 20
SPOS, 58
SSH, 37
ssh-askpass, 48
Stack, 37, 44
stderr, 42
stdin, 42
stdio, 42
stdio API, 42
stdout, 42
Stream, 14, 21, 22, 55
INDEX

Flags, 16
autoconf, 17
cleanmeta, 17
hwmixer, 18
meta, 17
mute, 18
output, 17
pause, 17
primary, 16
source, 17
sync, 16
Types, 15
Bidir, 15
Bridge, 16
Filter, 15
Light In, 16
Light Out, 16
Meta, 16
MIDI In, 16
MIDI Out, 16
Mixing, 15
Monitor, 15
Output, 15
Play, 15
Record, 15
Thru, 16
Streaming, 23
style, 52
Subsystem, 18
subsystem, 14
symlink, 32
System Speaker, 14
TCP, 55
UID, 27
UNIX, 27, 31
UNIX Domain Sockets, 27, 31
User, 27
VIO, 37, 55
VIO API, 37
VIO OpenSSL BIO API, 45
VIO Operations, 48
VIO select, 48
VIO Stack, 44
VIO Stack API, 44
Virtual Input Output, 37
Volume, 21
Vorbis, 19

Waveform, 14
Wire, 55
Wireless, 55
XMMS, 12
yaourt, 8
YIFF, 30
YIFF Sound System, 30
List of Figures

21.1 Indenting example ........................................ 67
21.2 Indenting example ........................................ 68
21.3 Example of comments ................................. 69

23.1 Example of VCLT Stream ............................ 82
List of Tables

5.1 Tested players without native RoarAudio plugin .............. 17
6.1 Table of Subsystems ........................................... 19
10.1 Possible output drivers ....................................... 31
10.2 Possible output options ....................................... 32
15.1 Examples of Socket Addresses ................................... 41
17.1 Channel Mapping ............................................... 45
17.2 Channel positions .............................................. 45
17.3 Example Frames for stereo signal .............................. 46
21.1 Safe value ranges of data types ................................. 70
22.1 RoarAudio Protocol in OSI Layer Model ...................... 71
22.2 Format of message version 0 .................................. 73
22.3 Format of message version 1 .................................. 73
22.4 Message flags for format version 1 messages ............... 74
22.5 Format of version 0 continued Meta Frames ............... 75
22.6 Format of message version 2 .................................. 75
22.7 Table of base and connection commands ..................... 75
22.8 Table of response commands .................................. 77
22.9 Table of server status and control commands ............. 78
22.10 Table of stream commands ................................... 78
22.11 Table of other commands .................................... 78
25.1 Identification header ........................................... 85