↑↑ Home ↑ Noise ↑ UNIX

A close look at ALSA

Intro -- Concepts -- Configuration -- Troubleshooting -- In practice

Intro

Everyone who has more to do with music on their Linux box than listening to stereo sound on a single sound card will sooner or later come into contact with ALSA. It stands for Advanced Linux Sound Architecture and is much more powerful and versatile than the dated Open Sound System which preceded it. In fact, you may already be using ALSA unawares via its OSS emulation feature. Still, when searching for some answers concerning ALSA on the web, I found a great many more questions, contradictory statements and very few actual answers. There are two reasons for this: Some sound issues are not as simple as they seem, and ALSA's documentation is plain lousy (which some have attributed to job security). This web page seeks to explain the first observation and to redress the second.

Before we start, you might want to have a look at other sources. Some of them contain example programs which you can copy blindly if you just want to play or record a sound with ALSA, or may already contain the solution to your specific problem. I can especially recommend this informed and comprehensive page — it covers more ground than this page at the expense of going less deep.

Other web sources include a LINUX Journal article about basic ALSA programming, which includes example programs, tutorials on the ALSA project web site, which also include code fragments, and one of the developers' home page, even though it is a bit old.

Some documentation is embedded in the ALSA library source code and can be generated using the source documentation tool `doxygen`, with the command `make doc`. It can also be read online here. Unfortunately it is incomplete and annoying to navigate (things like a complete function index, which doxygen usually provides, are missing). It seems to me the developers wanted to provide more than just an API documentation but ended up with neither one thing nor another. If you really want to understand how ALSA works, however, you will need to look at both the doxygen documentation and the source code itself.

ALSA concepts

Hardware -- Parameters -- Devices and plugins

Sound cards and hardware devices

ALSA arranges hardware audio devices and their components into a hierarchy of cards, devices and subdevices. It reflects the structure and capabilities of your hardware as seen by ALSA. If there is a discrepancy between a sound card's device structure and its documentation, this may be due to the driver not supporting all its features.

ALSA **cards** correspond one-to-one to hardware sound cards. They do not play a big part except that one can list the devices on each card. A card can be denoted by its ID (a string, see below) or a numerical index starting at zero.

Most of ALSA's hardware access happens at the **device** level. The devices of each card are enumerated starting from zero. Different devices can be opened and used independently of each other. Typically, specifying a sound card and device will be sufficient to determine on which connector or set of connectors your audio signal will come out, or from which it is read.

Subdevices are the most fine-grained objects ALSA can distinguish. The most frequently encountered cases are that a device has a separate subdevice for each channel or that there is only one subdevice altogether. A device's subdevices can in principle be used independently, but playing a multi-channel signal to a subdevice will use (and block) the following subdevice(s) too. Like devices, subdevices are identified by a zero-based index.

PCM parameters and the configuration space

Digitised sound has a number of parameters such as the sampling rate, the number of channels and the format in which sample values are stored. If you have been programming OSS, you may be used to setting these parameters one after the other before playing a sound file. So what is that talk about a "configuration space" which one finds in the ALSA documentation?

The answer is that in reality things are not so simple: Some sound cards cannot combine all sample formats with all sampling rates or channel counts, for example. So the parameters are not independent. ALSA accounts for this fact by arranging sets of parameters in an n-dimensional space, the configuration space. One of these dimensions corresponds to the sampling rate, one to the sample format, and so on. If the parameters of one specific sound card are all independent, all legal configurations lie in one big n-dimensional box. In this case, one could describe them by giving separate ranges for all parameters. If the parameters are not independent, the set of allowed configurations is more complicated, and could not be expressed so simply.

When a hardware device is accessed with ALSA, parameters are not fixed independently of each other. Rather, the legal configuration space for a device is narrowed down successively by restricting specific parameters. This makes it possible, for instance, to set a minimal rather than exact sampling rate. It also potentially leads to the problem that it depends on the order in which the parameters are set which parameter set you end up with (see below). That said, an ALSA plugin is available which automatically chooses the most sensible hardware parameters and performs format conversion as needed. This and other plugins are described in the following section.

ALSA devices and plugins

To avoid confusion later on, a few brief notes on ALSA devices are in order. These are quite different from the hardware devices introduced above. ALSA devices are denoted by strings. They are defined in one of ALSA's configuration files (see below) and are basically wrappers for plugins. To complicate matters further, many standard ALSA devices are generic and have to be followed by a colon and the card, hardware device and (optionally) subdevice separated by commas. But the hardware card and device specification can never serve as an ALSA device on their own, and indeed some ALSA devices have arguments other than these hardware-related variables.

It is no great exaggeration to say that ALSA consists almost entirely of plugins. Whenever a player or other program uses an ALSA device, plugins do the dirty work. A full list of plugins can be found in the file `pcm_plugins.html` in the ALSA library doxygen documentation, which is on the web here. It should be noted however that a list of plugins is not the same as a list of ALSA devices. Some standard devices have the same name as the plugin they use, but some have not, and sometimes different ALSA devices use the same plugin. Therefore this section will give both the name of each plugin and, if applicable, the name of the ALSA device which allows to use it with a specific hardware device.

The most important plugin is no doubt the `hw` plugin. It does no processing of its own, but merely accesses the hardware driver. If an application chooses a PCM parameter (sampling rate, channel count or sample format) which the hardware does not support, the `hw` plugin returns an error. Therefore the next most important plugin is the `plug` plugin which performs channel duplication, sample value conversion and resampling when necessary. Unlike the `hw` plugin which is used by the device `hw:0,0`, the device corresponding to the `plug` plugin is named differently, `plughw:0,0`. Both have as parameters the card (ID string or numerical index), device and optionally subdevice of the hardware to be accessed. (In fact, the `plug` device also exists, which also uses the `plug` plugin and has as its single argument the name of the slave (ALSA) device to send its data to. It thereby can be daisy-chained with other plugins.)

Also a very useful plugin is the `file` plugin which writes sample data to a file. It is behind two ALSA devices, `file` and `tee`. The former has two arguments, the file name and the format (which currently is always "raw"). The latter passes the data to another device in addition to writing it to a file, and has that device as its first argument. If that secondary device has any arguments (like "`plughw:0,0`"), you have to put its name in quotes and protect those from interpretation by the command shell. Assuming you want to use the first device on the first sound card, you could obtain a copy of the sound data output to it with:

```
aplay -Dtee:\'plughw:0,0\',/tmp/alsatee.out,raw xy.wav
```

Admittedly this seems rather pointless with `aplay` (you could simply copy `xy.wav` or convert it with `sox`), however using it with an ALSA-aware movie player allows you to extract the sound track. `tee`'s output is always raw sample data without any header. The `file` plugin can also be used to read data from a file, but there is no predefined device which uses it in this way.

A number of plugins exist for mixing and/or rerouting channels. Due to the large and variable number of parameters they require, there are no pre-defined ALSA devices using them. The `route` plugin is a mixing matrix. Not only can channels be swapped or arbitrarily assigned, but they can also be mixed. (See the section about surround sound for examples.) The `multi` plugin allows only to reroute channels, but can have several slave devices, so that different channels are played via different cards. (If you try this out, you will notice that the cards will drift out of sync over time. Professional audio sound cards have word clock inputs which allow them to synchronise.) The `dmix` and `dshare` plugins are supposed to allow a device to be shared between several clients (player applications). The `dshare` plugin divides the available channels up between clients, while `dmix` mixes together whatever is played on the same channel(s). In my brief experimentation, I haven't got `dmix` to work, though.

So much for a brief overview of the most important plugins and pre-defined ALSA devices. For using the more complicated plugins, you will have to write your own configuration file, described in the following section. Examples for device definitions can be found below as well as in the ALSA project's documentation of its configuration file and its list of plugins.

Configuring ALSA

Configuration files -- Basic format -- Advanced features

Configuration files

Configuration files serve to define ALSA devices. Without these definitions, you could not use any of the features of ALSA — play no sound, adjust no mixer, nada. Still, you do not need to write a configuration file to simply play and record sound. The "built-in" configuration file alsa.conf contains definitions of devices which allow that, and much more besides (see above). But if you have special requirements, or if you experience trouble, you might still want to add definitions of your own.

ALSA supports three tiers of configuration files. The first is alsa.conf, which is located in ALSA's data directory, usually /usr/share/alsa. This directory and its subdirectories may contain further configuration files which are sound card and plugin specific. They are included by alsa.conf depending on the sound cards present. The files in this directory are considered built-in and are not to be changed by the user or the system adminstrator.

The other two configuration files are also included by alsa.conf. The system-wide configuration is stored in /etc/asoundrc. Users can store their own configuration in the file .asoundrc in their home directory. All configuration files are parsed every time an ALSA device is opened. So changes take effect immediately, and it is not necessary to restart anything.

All configuration files share the same format, which is described in the next section.

Basic configuration file format

ALSA configuration files contain hierarchically structured parameter-value pairs. The top-level branches of the hierarchy correspond to the interfaces which ALSA offers. An interface consists of a set of ALSA API functions which allow to open a device, do whatever the interface is for, and close the device. The different interfaces have different purposes. For example, `aplay` and other players use the `pcm` interface, the program `alsactl` uses the `ctl` (control) interface, `amixer` uses the `mixer` interface, `amidi` the `rawmidi` interface and so on. (In fact most of them also use the `ctl` interface, at least for some features.) To create a device to be used by one of these programs, you have to define it for the corresponding interface.

The examples which I give will mostly concern the `pcm` interface, both because it is probably the most important and because I have almost exclusively messed around with it. (PCM stands for pulse code modulation and refers to digitised sound expressed as a stream of sample values.) Besides, the command `aplay -L` allows you to list all definitions in the `pcm` interface, which is a nice cross-check. Other interfaces are `ctl` (control, for controlling hardware mixers and other settings), `seq` (sequencer), `hwdep` (hardware dependent features), `mixer` (abstracted mixer control), and `rawmidi` (MIDI data I/O). The second level of the hierarchy consists of the names of ALSA devices which can be used with the particular interface in whose subtree they reside.

Let's have an example. Imagine you want to create an ALSA PCM device which accesses your first sound card and does format conversion as needed. (Ignore for now that you could do that with the device "`plughw:0,0`".) The plugin which handles automatic format conversion is the `plug` plugin. The new PCM device is created by putting the following lines into your asoundrc:

```
pcm.plug0 {
  type plug
  slave {
    pcm "hw:0,0"
  }
}
```

What this means is the following: There shall be a new device "`plug0`" accessible via the `pcm` interface. Data output on this device shall be handled by the `plug` plugin. The plugin uses as a slave the PCM device `hw:0,0`. This device definition was written in the way which is most common. However, ALSA allows some freedom in the syntax. For instance, it is allowed to put an equals sign between a parameters name and its value, and a comma or a semicolon between consecutive parameter assignments. So we could also have written:

```
pcm.plug0 = {
  type= plug;
  slave= {
    pcm= "hw:0,0";
  },
};
```

As you can see, the two compounds enclosed in curly braces are really the "values" of the parameter names preceding them, and our name for the new PCM device is a freely selectable parameter name. Even the last parameter assignment in a compound may be followed by a comma or semicolon. It should now also be clear why the slace PCM device had to be enclosed in quotes: Otherwise the comma would have been misinterpreted and led to an error message.

There are further freedoms in the syntax which we have not touched yet. It concerns compounds. The names of sub-parameters of compounds may be given either in a dot notation or in braces. The first notation was chosed above for the name of our new device, which is just a parameter in the compound `pcm`. The second was used for all its sub-parameters. Furthermore, the syntax of the configuration files is not line-oriented — line breaks can be inserted anywhere but are treated as white space. So the same device definition as above could also have been written in one of the two following forms:

```
pcm {
  plug0 {
    type plug slave { pcm "hw:0,0" }
  }
}
```

or

```
pcm.plug0.type= plug; pcm.plug0.slave.pcm= "hw:0,0"
```

Now I have given you an idea of the structure of ALSA configuration files, let us have a word about the remaining basics. Parameter names seem to be made up of letters, numbers and underscores (I say this from trial and error, as it seems to be documented nowhere). Parameter values which contain characters other than letters, numbers and underscores should probably be quoted. Both parameter names and values are case sensitive. Comments in configuration files start with "#" and extend up to the end of the line. They seem to be allowed pretty much anywhere, even between a parameter's name and value.

Rather than giving the slave explicitly, we could have used a symbolic slave definition above. Then our definition of `plug0` would have looked like this:

```
pcm_slave.slave0 {
  pcm "hw:0,0"
}
pcm.plug0 {
  type plug
  slave slave0
}
```

This explicit slave definition looks pointless here, but it may have its use in some cases. One can restrict a device's hardware parameter space in the process of creating a slave, for instance to force playback on all available channels and with the maximum sampling rate. (Not that it makes any sense to force resampling by the `plug` plugin.) Then our slave definition would look like this:

```
pcm_slave.slave0 {
  pcm "hw:0,0"
  channels 6
  rate 96000
}
```

If the same slave was used in multiple device definitions, one could save some typing by putting these restrictions into a separate slave definition instead of every `slave` compound.

Another simple feature of configuration files are aliases. They are simply a parameter assignment of an existing device name to the alias. Both are part of the same interface, and the interface name is not repeated in the value of the alias. (Please note that in some texts about the asoundrc, the word "alias" is used erroneously for any device definition.) An example with our `plug0` device:

```
pcm.alias_plug0= plug0
```

(**not** `pcm.alias_plug0= pcm.plug0`) You cannot define aliases for ALSA devices with arguments (see below).

Now we have covered the basics, you might want to read the web pages here and here as well as the file `pcm_plugins.html` in the ALSA library's doxygen documentation (on the web here), which provide many examples.

Advanced configuration file features

Overriding parameters and parameter data types

If you have read other documents about the asoundrc, you may have learnt that you can redefine ALSA's default device with a line like the following:

```
pcm.!default { type hw  card 0 }
```

You may also have read that the exclamation sign causes the previous definition of pcm.default to be overridden. This syntax can be used with any configuration file assignment. Let's have a closer look at it to

see how ALSA's configuration really works. Normal assignments add a leaf (and, if needed, branches) to the tree structure which contains all parameters. If this leaf (parameter) already exists, its value will indeed be overwritten. So if you put the following in your asoundrc, the default sound card will be the second, not the first:

```
pcm.!default { type hw  card 0 }
pcm.default.card 1
```

So what is the exclamation mark needed for? If you remove it from the first definition of `default`, you will receive an error message to the effect that `default` is not a compound. Obviously, parameters come in several data types, and that type cannot be changed by a mere assignment. By overriding a parameter with "!", you can change its type. (The data type is not to be confused with the "type" subparameter of a device definition.) The exclamation mark causes the parameter and all its sub-parameters to be removed before it is created anew. (So don't ever write something like "`!pcm...`" — this would presumably erase all definitions in the `pcm` interface.)

With a little experimentation, one can find out that aliases are in fact parameters of type `string` which contain as their value the name of the PCM device they refer to. `pcm.default` is defined as an alias in alsa.conf, necessitating the use of the exclamation mark if it is to be replaced by a compound definition.

Similarly to the exclamation mark, other prefix characters can be used to qualify a parameter assignment. A question mark causes an assignment to be ignored if a previous value for the same parameter exists. (If you try to assign a subparameter of something which is not a compound, such as `pcm.default.?card` without a previous forced redefinition of `pcm.default`, you will still get an error.) So the following:

```
pcm.?default { type hw card 1 }
```

defines the first device of the second sound card as the default device, but only if no default device has been defined previously (say, in a system-wide configuration file).

The remaining two prefix characters are + and –. Both require the assignment to respect the type of any earlier assignment/creation of the respective parameter. The plus sign prefix gives the default behaviour of creating a new parameter when necessary and can therefore be left out. The minus sign causes an error when trying to assign a parameter which did not previously exist.

Parametrised device definitions

In the section about plugins we have encountered a lot of ALSA devices which require arguments given after a colon following the device name and separated by commas. As I already mentioned above, these devices and the plugins behind them are actually quite different things, but devices can be viewed as wrappers round the plugins. In the basic configuration section we have defined a device using the `plug` plugin. The predefined devices we encountered in the plugin section differ from this simple device in that they are generic. The `plughw` device can be used for any sound card and (hardware) device, because one can give the card and device numbers as arguments after the colon. Though all parametrised devices are defined in alsa.conf and the ALSA documenation does not document their syntax, you can also define such devices of your own.

Let's have a look at an abbreviated form of the definition of the `plughw` device in alsa.conf:

```
plughw {
  @args [ CARD DEV SUBDEV ]
  @args.CARD {
    type string
  }
  @args.DEV {
    type integer
  }
  @args.SUBDEV {
    type integer
  }
  type plug
  slave.pcm {
```

```
    type hw
    card $CARD
    device $DEV
    subdevice $SUBDEV
  }
}
```

The first line in the definition of the `plughw` compound is the declaration of its argument list. This is a data type which is new to us: an array. Its elements can also be assigned one by one:

```
plughw {
  @args.0 CARD
  @args.1 DEV
  @args.2 SUBDEV
  ...
```

As in the assignment of compounds, an equals sign can optionally be inserted, both in the collective and the element assignment. The following lines in the definition of `plughw` define the data type of each argument. (Apparently the value of each array element now serves as the name of a subparameter, which is highly interesting from a formal languages point of view. ;)) In the original definition of `plughw`, the compounds defining the arguments also contain default definitions which use the `@func` keyword to call certain functions. If you are interested in that, have a look at alsa.conf.

Then follows the actual definition of the new PCM device. It strongly resembles the definitions we have seen so far, except that the constant parameter values are replaced by arguments referenced by their name preceded by a dollar sign. To get an idea of the possibilities generic ALSA devices offer, you are encouraged to look at alsa.conf, where many are defined.

There are two ways to use such a generic device. Its arguments can be given either in sequence starting with the first, or in any order as assignments to the argument name. Both "`hw:1,0`" and "`hw:DEV=0,CARD=1`" refer to the first device on the second sound card. For reasons I do not entirely understand, a parametrised device is not allowed in all circumstances where a non-parametrised device would be. The following is legal:

```
pcm.hw0 { type hw card 0 device 0 }
pcm.alias0 hw0
```

But this is not:

```
pcm.alias0 "hw:0,0"
```

If you want to learn about yet more advanced features of the ALSA asoundrc, have a look at `asoundrc.txt` in the `doc` directory of the ALSA library sources. It says something there about servers and references to (possibly external) libraries which I have not taken the time to check out. Up to you.

Troubleshooting

Standard tools -- `alpsacap` -- Mystery #1 -- Mystery #2

Useful tools

The simplest way to find out something about your sound card(s) and how ALSA sees them is calling `aplay -l` for playback and `arecord -l` for recording. These commands list the cards, hardware devices and subdevices available for playback and recording, respectively. It also lists the card ID, which can be used instead of its index to specify the card. My program `alsacap`, described below, can do the same thing.

If you are in doubt how ALSA interprets your asoundrc (or a different one of its configuration files), you can use the command `aplay -L`, which prints out all the configuration of the `pcm` interface in a standardised form. Note that the prefix "`pcm.`" is missing from the top-level parameter names. It is always implicit.

Another interesting option of `aplay` is `-v`, which prints out the hardware parameters of every subdevice used during playback. (Unlike the other two options mentioned, it requires a file to play.) This can for example be useful if you are using the `plug` plugin to play a song and want to check whether it does any resampling. My experience is that it acts smartly when the required parameters are not supported by the hardware and avoids resampling whenever possible, but you might want to check.

But what if you have similar doubts about a different audio or movie player? Then the /proc file system can help. The hardware parameters of a subdevice can be read from /proc/asound/card#/pcm#p/sub#/hw_params while it is in use (where "#" stands for the card, device and subdevice numbers). By printing or copying these files while the player is running and comparing them to the PCM parameters of the file being played, you can find out which parameters are emulated, if any. When a subdevice is not in use, these parameter files contain only the word "closed". If you see this on the subdevice on which you play your sound, clearly your sound stream is not going where it should. Exception: when an ALSA device is being used for multi-channel playback, the first subdevice (usually number 0) will be assigned the total channel count, regardless of its own maximum, and the others will be reported as closed. If ALSA's OSS emulation is used, the hw_params files also contain the parameter settings of OSS. The directories pcm#c (instead of pcm#p) contain equivalent files related to capture (recording).

The `amidi` program is the equivalent of `aplay` for the `rawmidi` interface. Its option `-l` lists all available (hardware) MIDI devices, and the `-L` option prints out the configuration of the `rawmidi` interface.

The tool `speaker-test` is useful if you do not get any sound but are not sure whether you player (or other sound source) is at fault. Intended primarily for assigning surround speakers to channels, it outputs a tone or noise to each speaker in turn, while printing where the tone should go. It has command line options which allow to send a tone to one specific speaker and to determine the frequency of the tone and the sampling rate. (The option `-s` determines the channel/speaker to use, starting from 1, and can in fact take values larger than 2, even though its manual page suggests that it can't.) Have a look at its manual page.

`alsacap`

`alsacap` is a small program I wrote to get to grips with non-trivial hardware configuration spaces and with related peculiarities I had noticed on my system. I think it will be useful to anyone troubleshooting ALSA. A tarball containing the source code and manual page is here. A Makefile for compiling and installing is included. It is licensed under the ISC licence. `alsacap` stands for "ALSA capability", as it displays the capabilities of your sound card and its ALSA driver. You can also read its manual page online here.

When executed with the command line option `-h`, `alsacap` displays a brief usage message. It can be used in one of two ways. Its first purpose is for listing ALSA hardware devices in a similar manner as `aplay -l`. In addition to the information given by `aplay`, it outputs the range of channel counts and sampling rates and all sample formats supported by each device. The device scan can be restricted to a card or a single device by giving the card number and/or the device number with the options `-C` and `-D`, respectively. The option `-R` causes `alsacap` to search for recording devices instead of playback devices.

[Aside: ALSA folks disdain the use of the word "recording" for reading data from the sound card, on the grounds that recording requires a storage medium, which ALSA does not handle. They have a point, as recording more obviously implies a storage medium than playback does. However, for an application for which the term "recording" is genuinely wrong (such as capturing sound, filtering it digitally and outputting it again), "playback" seems strange too. So both this web page and `alsacap` use the terms "playback" and "recording". If that worries you, you can change it in the source ;). End of aside.]

You may remember from my introduction to the configuration space (above) that ranges of hardware parameters tell the whole story only if these parameters are independent. It is for the more complicated cases that the second functionality of `alsacap` was created, which could be called that of a configuration space explorer. An ALSA device *has* to be given for this usage, and it has to be given in the same form as for any ALSA application, rather than as a card and device number as for the device overview. This is done with the `-d` option, which also accepts any valid ALSA device, not just `hw:...` devices. The `-R` option again selects a recording instead of playback device.

The other options allow you to fix the sampling rate (`-r`), the channel count (`-c`) and the sample format (-f) before the remaining parameter ranges are displayed. The important thing is that it sets these parameters in *the same order* in which they are given on the command line. This allows you to find out by trial and error if and how parameters depend on each other. For instance consider the following output for the first device of my second sound card (an Echoaudio GINA3G):

```
vs@schizo, ~/soft/sound-misc > ./alsacap -d hw:1,0
*** Exploring configuration space of device `hw:1,0' for playback ***
1..6 channels
Sampling rate 32000..100000 Hz
Sample formats: U8, S16_LE, S32_LE, S32_BE, S24_3LE
vs@schizo, ~/soft/sound-misc > ./alsacap -d hw:1,0 -c 1
*** Exploring configuration space of device `hw:1,0' for playback ***
Set number of channels to 1.
Parameter ranges remaining after these settings:
1 channel
Sampling rate 32000..100000 Hz
Sample formats: S32_LE, S32_BE
Significant bits: 32
```

Let's go through the output line by line. The first execution of `alsacap` displays the total parameter ranges for the given device. It can handle up to six channels with sampling rates between 32 and 100 kHz and in a number of formats. The second time, `alsacap` is called with the `-c` option to set the number of channels to mono. As expected, the only possible number of channels left is one. But surprise, surprise: The available sample formats have also changed, leaving only two 32-bit formats (little- and big-endian).

This is an example of the allowed range of one parameter (the sample format) depending on the choice of another (the number of channels). The last line in the example shows an additional feature of `alsacap`: If the sample format has been narrowed down to one bit width, the number of significant bits is printed. Some sound cards accept, say, 32 bit sample values, but discard some least significant bits. But all these values only represent the world according to ALSA. My sound card documentation merely mentions 24 bits, as well as a minimum sample rate of 25 kHz (continuously variable mode) and 8 kHz (preset). So the ALSA driver does not include all the card's features. On the other hand, who needs 8 kHz surround sound?

When fixing the sampling rate, `alsacap` prints out which sampling rate could actually be obtained, and the direction in which ALSA reports the actual rate differing from the requested (<, = or >). In my version (and perhaps other versions) of ALSA, this is erroneously always returned as >. I decided nevertheless to keep this feature in the hope that future versions of ALSA will return a meaningful value. In the mean time, ignore it.

ALSA mystery #1: Why does my soundcard play mono on one speaker only?

After buying and installing my second sound card, I was surprised to discover that `aplay` and other ALSA players output mono wave files only on the left channel. Particularly when listening with headphones, this was a pain. I found out that the `play` script which comes with `sox`, adapted for the second sound card device /dev/dsp1, did not suffer from this effect, and henceforth used that and forgot about the problem.

Recently, when setting up a surround amplifier and speaker system, I was confronted with it again, as I wanted to use one player for any file, and `play` cannot play surround sound. I found that when I built a six-channel device out of the subdevices of the sound card device (with the `multi` plugin), mono sound was duplicated on all channels. Only when I used the `hw:...` or `plughw:...` device, mono was left channel only.

The solution is that ALSA was not in the least at fault. The sound card supports single-channel playback, the driver supports it, so it played mono sounds on the device's first subdevice, which happens to be the left stereo channel. With my six-channel `multi` plugin device, I had merely forced the number of channels to be 6, and thereby forced duplication of the mono signal. `sox` and its `play` script uses OSS, and always in stereo. The easiest way to fix the problem is to put something like this into your `.asoundrc`:

```
pcm.plug1s {
  type plug
  slave {
```

```
    pcm "hw:1,0"
    channels 2
  }
}
```

This restricts the number of channels for that specific sound device to two; substitute a larger number if you want surround sound. If you wanted the possibility to restrict any hardware device to two channels, but not always, you could define the following:

```
pcm.stereo {
  @args [ CARD DEV SUBDEV ]
  @args.CARD { type string default 0 }
  @args.DEV { type integer default 0 }
  @args.SUBDEV { type integer default 0 }
  type plug
  slave {
    pcm { type hw  card $CARD  device $DEV  subdevice $SUBDEV }
    channels 2
  }
}
```

This PCM device is a `plug` plugin with a hardware device as a slave, which is restricted to two channels. To apply it to the first device of the second sound card, you have to give "`stereo:1,0`" as the playback device. Note that its default hardware device is the very first device. Refer to your `alsa.conf` for code to use the default ALSA device as a default for your PCM definitions; it contains many similar definitions which support that.

ALSA mystery #2: Why does player X sound better than player Y?

This is something I have not actually encountered myself, but according to reports on the web, it seems to happen. Here are a few ideas as to how to find out more about the problem. The first order of the day is to have a look at /proc/asound/card#/pcm#p/sub0/hw_params. (Subdevice 0 will usually suffice, unless one of the others is controlled separately.) If you find the same parameters there for either player, I am pretty much stumped. Then it is possibly a case of one of the players trying to "improve" the sound, be it successfully or unsuccessfully, or of a better or worse audio codec.

If some of the hardware parameters differ, one of the programs in question may set the PCM parameters the old-school way, by demanding the closest match of one parameter after the other, and not being very smart about it. For sound cards whose parameters depend on each other, this can be a problem.

Consider the following output of `alsacap`:

```
vs@schizo, ~/soft/sound-misc > ./alsacap -d hw:1,0 -c 1 -f S16_LE
*** Exploring configuration space of device `hw:1,0' for playback ***
Set number of channels to 1.
Could not set sample format to S16_LE: Invalid argument.  Continuing regardless.
Parameter ranges remaining after these settings:
1 channel
Sampling rate 32000..100000 Hz
Sample formats: S32_LE, S32_BE
Significant bits: 32
vs@schizo, ~/soft/sound-misc > ./alsacap -d hw:1,0 -f S16_LE -c 1
*** Exploring configuration space of device `hw:1,0' for playback ***
Set sample format to S16_LE.
Could not set # of channels to 1: Invalid argument.  Continuing regardless.
Parameter ranges remaining after these settings:
2..6 channels
Sampling rate 32000..100000 Hz
Sample formats: S16_LE
Significant bits: 16
```

This is again an example using my Echoaudio sound card, which has restricted parameter values for mono playback. In the first case, the number of channels is set first, causing the desired sample format to be unavailable. In the second case, the sample format is fixed first, making it impossible to play mono sound

(with the hardware alone). Here this is not a problem, as both scaling up 16-bit samples to 32 bits and duplicating a mono channel are exact operations which do not degrade audio quality. However, if a sound card has certain restrictions on the sampling rate depending on the channel count or sample format, this might be different as it could cause resampling.

The simplest solution to such problems is to make every player use the `plug` plugin (via a `plughw:...` device) rather than the corresponding hardware device. I do not completely see through its source code, but my impression is that it handles interdependent parameters smartly. If your player does not allow you to determine the ALSA device to be used (many seem to allow only `hw:...` devices), this solution does not work.

If you are so inclined, you can still modify the player's source code. `alsacap` can help you to find out what the player does. First find the PCM parameters of the file being played (from the file, file format or the player's output) and use `alsacap` in its "configuration space explorer" mode to set them. Put those parameters first which the player set correctly according to the /proc filesystem. Then try to find a different order which obviates the need for resampling. Modify the player's source code accordingly and recompile. Obviously, this is an option only for hackers. It can however help you also in debugging your own ALSA application.

ALSA in practice

Surround sound -- ALSA and applications

ALSA and surround sound

For those lucky enough to have a surround speaker system connected to one of their sound cards (like me :)), ALSA provides the generic devices `surround40`, `surround41`, `surround50`, `surround51` and `surround71`. I can say from my own experience that they are neither necessary nor always usable, as they seem to call special functions which do not exist for all sound cards which can handle surround sound. My primary sound card (Echoaudio GINA3G) is not supported by the `surround..` devices, while my on-board sound (Realtek ALC882 / Intel HDA) is. Both work fine playing surround sound with the `hw` and `plughw` devices.

One drawback of surround sound which can lead to problems is that the channel assignment is not standardised (except in the sense that the great thing about standards is that there are so many to choose from). Four channels can represent either 3.1 surround sound (left/right + centre + subwoofer) or 4.0 surround sound (front left/right + surround left/right). Similarly, there are two different conventions for 5.1 surround sound, one upward compatible to 3.1 and one to 4.0. In the first case, the channel order is front left, front right, centre, LFE (low frequency effects = subwoofer), surround left, surround right; in the latter case front left, front right, surround left, surround right, centre, LFE. The latter configuration is the standard under Linux, while the former is common on computer illiterate systems (guess which OS that is...). But reportedly some games assume M$ channel order even under Linux, with the effect that voices of characters behind the player come out of the centre speaker. I do not know about 7.1 channel assignment, but I expect there are the same two configurations, with rear left and right being the last two channels in either case.

You can of course choose your channel assignment as you want — you define it by wiring your sound card to your amplifier in a certain way. Of course it makes most sense to choose one of the two standards. ALSA makes it easy to play audio files with the wrong channel order. The following generic ALSA device converts between the two 5.1 formats:

```
pcm.swap51 {
  @args.0 SLAVE
  @args.SLAVE {
    type string
    default "plughw:0,0"
  }
  type route
  slave {
    pcm $SLAVE
```

```
    channels 6
  }
  ttable {
    0.0= 1
    1.1= 1
    2.4= 1
    3.5= 1
    4.2= 1
    5.3= 1
  }
}
```

This definition uses the `route` plugin to swap channels 2 and 3 with channels 4 and 5. The `route` plugin is used as a switching matrix here, no mixing is done. The numbers to the left of the (optional) equals signs are the source channel numbers followed by a dot and the output channel numbers. The dot is not a decimal point, but the separator between the compound name (which is the source channel number) and the name of its component (the output channel number), like here. The entries of the matrix which we do not give are automatically zeroed.

Using this device, you can play a 5.1 surround audio file with the wrong channel order using the command:

```
aplay -Dswap51:\'plughw:1,0\' surround.wav
```

or just:

```
aplay -Dswap51 surround.wav
```

to play on the sound card and hardware device given above as the default. One could define an analogous swap device for 7.1 sound, of course.

What if you are not blessed with a surround speaker system? ALSA can help here too. The simplest option would be to discard the surround channels and just listen to the front left/right signal in stereo. But that is pretty poor, and if nothing else you will miss out on the reverberation usually present in the surround channels. There is a better way, namely to mix the surround channels into the stereo channels. Here is an example for 5.1 sound:

```
pcm.51to2 {
  @args.0 SLAVE
  @args.SLAVE { type string default "plughw:0,0" }
  type route
  slave {
    pcm $SLAVE
    channels 2
  }
  ttable {
    0.0= 0.3
    2.0= 0.3
    4.0= 0.18
    5.0= 0.21
    1.1= 0.3
    3.1= 0.3
    4.1= 0.18
    5.1= 0.21
  }
}
```

The surround left/right channels are mixed one-to-one with the front channels. The centre and LFE channels are split between the left and right output channel. Strictly both should have a coefficient of 0.212 to give the same intensity as the other channels. (Intensity is amplitude squared, so the total amplitude is $\mathrm{sqrt}(0.212^2+0.212^2)\approx 0.3$.) But that would require rescaling everything to prevent clipping, which may occur when the sum of contributions to one of the output channels exceeds 1. I decided to attenuate the centre channel a little because it is often correlated with the two front channels, while the LFE channel may contain independent information. Change the numbers to your liking.

The opposite is also possible: to extend a stereo source to 5.1 channels. For this purpose I suggest the following ALSA device:

```
pcm.2to51 {
  @args.0 SLAVE
  @args.SLAVE { type string default "plughw:0,0" }
  type route
  slave {
    pcm $SLAVE
    channels 6
  }
  ttable {
    0.0= 1
    0.2= -0.6
    0.3= -0.39
    0.4= 0.5
    0.5= 0.5
    1.1= 1
    1.2= -0.39
    1.3= -0.6
    1.4= 0.5
    1.5= 0.5
  }
}
```

As above, you might want to change some of the mixing coefficients to your liking. The rationale behind the mixing matrix is the following: The two stereo channels go unchanged to the two front speakers. This emphasises them over the other speakers, which are mixed together and have to receive a lower intensity to prevent clipping. The front left and right speakers would have the same intensity as the centre (for instance) for *independent* stereo channels if they had a coefficient of 0.71 (1/sqrt(2)) instead of 1. But often the two stereo channels are very correlated in order to be mono compatible for broadcasting. I chose the coefficient 1 because I found it sounded best for me; try it out.

The centre and subwoofer (LFE) channel are mixed fifty-fifty from the stereo channels, no surprise there. The surround left and right channels have negative coefficients. This serves to simulate a certain delay with respect to the front speakers in the absence of a true delay module. It amounts to a frequency-dependent delay equal to half the wave period. In my experience this makes the surround sound less "flat" and one-dimensional. ALSA's `ladspa` plugin might be used to implement a true delay or even reverberation, a possibility I have not looked into. In my mixing matrix, both surround channels receive a contribution from the opposite front channel, accounting for the fact that in a concert, the reflected sound reaching you from the side or behind would not come from the left or right only. I have reduced this cross-direction contribution from 0.4 to 0.39 after hearing nasty noises from my speakers - apparently ALSA's numerics is sufficiently inaccurate that clipping can happen if you mix two sources without leaving some headroom.

If you did not find here what you were looking for, have a look at the surround sound page of the ALSA Wiki. Among others, it contains links to example 5.1-channel audio files and some notes on using the `ladspa` plugin in relation with surround sound.

ALSA and some applications

This section contains some very brief notes about whether and how some applications work together with ALSA.

The wave file editor `snd` can be compiled to use ALSA for sound output and recording. If compiled so, the environment variable `MUS_ALSA_DEVICE` determines the ALSA device to use.

`ogg123` supports ALSA only partly – only hardware devices can be used. Not only does that preclude the use of many of ALSA's features, but `ogg123` even hangs when the hardware device does not support the needed PCM parameters. In my experience, the following use works well, however:

```
ogg123 -d wav -f - file.ogg | aplay -Ddevice
```

This makes `ogg123` output the song in wav format to its standard output stream and pipes it into `aplay`. The tricky bit is that the wav header should contain the length of the following data. `ogg123` solves this problem by setting this length to its maximum. `aplay` quits playing anyway when its input ends. If you are worried about the non-standard wav header (or if you experience trouble), use AU (Sun audio) format instead:

```
ogg123 -d au -f - file.ogg | aplay -Ddevice
```

Unlike wav, the AU format reserves a magic value for the unknown length of audio streams. So the fact that the length of the audio output is not yet known when the header is output can be adequately represented within the standard.

For reasons I do not understand, I have not succeeded in playing surround audio files with `ogg123`, either using its built-in ALSA hardware device support or in a pipe with `aplay`. What works, however, is piping the output of `oggdec` into `aplay`:

```
oggdec -o - file.ogg | aplay
```

This uses the wav file format, because `oggdec` does not support AU, but as is the case with `ogg123`, the length fields in the header are handled smartly. In contradiction to its documentation, `oggdec` seems even to be able to determine the correct length before writing the header out. If you experience trouble such as playback spuriously cut short, you might want to try using my small program `maxwav`, see below.

The command-line MP3 player `mpg123` is not aware of ALSA at all, but can be used in a pipe together with `aplay` similarly to `ogg123`:

```
mpg123 --au - file.mp3 | aplay
```

Alternatively, and in cases where the AU format is not an option, you can use my micro-program `maxwav` which sets the lengths in the wav header to their maximum:

```
mpg123 -w - file.mp3 | maxwav | aplay
```

This assumes you have installed `maxwav` in a directory which is in your PATH; otherwise give its path explicitly.

The movie and audio player `mplayer` is fully ALSA compatible and allows setting the (any) ALSA device to use on the command line. Here is an example:

```
mplayer -ao alsa:device=plughw=1.0
```

As you see, the device (for the `plug` plugin with the first hardware device of the second sound card) is given as "`plughw=1.0`" rather than "`plughw:1,0`", which would be the correct ALSA device specification. The colon has to be replaced by an equals sign and the comma by a dot to avoid misunderstandings with other features of `mplayer`'s `-ao` option. (As you can see, the colon acts as a separator between the `alsa` audio output specification and the device specification.) This might lead to problems if you define a parametrised device which takes real-valued arguments, but it will be sufficient in most cases.
