Making Sense of The Audio Stack On Unix

Patrick Louis

2021-02-07

Published online on venam.nixers.net

© Patrick Louis 2021

This publication is in copyright. Subject to statutory exception and to the provision of relevant collective licensing agreements, no reproduction of any part may take place without the written permission of the rightful author.

First published eBook format 2021

The author has no responsibility for the persistence or accuracy of urls for external or third-party internet websites referred to in this publication, and does not guarantee that any content on such websites is, or will remain, accurate or appropriate.

Contents

Introduction

Come see my magical gramophone

Audio on Unix is a little zoo, there are so many acronyms for projects and APIs that it's easy to get lost. Let's tackle that issue! Most articles are confusing because they either use audio technical jargon, or because they barely scratch the surface and leave people clueless. A little knowledge can be dangerous.
In this article I'll try to bridge the gap by not requiring any prerequisite knowledge while also giving a good overview of the whole Unix audio landscape. There's going to be enough details to remove mysticism (Oh so pernicious in web bubbles) and see how the pieces fit.

By the end of this article you should understand the following terms:

- ALSA
- OSS
- ESD
- aRts
- sndio
- PulseAudio
- PipeWire
- GStreamer
- LADSPA

We'll try to make sense of their utility and their link. The article will focus a bit more on the Linux stack as it is has more components than others and is more advanced in that respect. We'll also skip non-open source Unix-like systems.
As usual, if you want to go in depth there's a list of references at the bottom.

Overall, we got:

- Hardware layer: the physical devices, input and output
- Kernel layer: interfacing with the different hardware and managing their specificities (ALSA, OSS)
- Libraries: used by software to interface with the hardware directly, to manipulate audio/video, to interface with an intermediate layer for creating streams (GStreamer, Libcanberra, libpulse, libalsa, etc..), and to have a standard format (LADSPA).
- Sound servers: used to make the user facing (user-level) interaction easier, more abstract, and high level. This often acts as glue, resolving the issue that different software speak different protocols. (PulseAudio, ESD, aRts, PipeWire, sndio)

Let me preface this by saying that I am not a developer in any of these tech, neither am I a sound engineer. I am simply regrouping my general understanding of the tech so that anyone can get an overview of what the pieces involved are, and maybe a bit more.

Hardware layer

It's essential to have a look at the hardware at our disposal to understand the audio stack because anything above it will be its direct representation.
There are many types of audio interfaces, be it input or output, with different varieties of sound cards, internal organizations, and capabilities. Because of this diversity of chipsets, it's simpler to group them into families when interacting with them.

Let's list the most common logical components that these cards can have.

- An interface to communicate with the card connected to the bus, be it interrupts, IO ports, DMA (direct memory access), etc..
- Output devices (DAC: Digital to analog converter)
- Input devices (ADC: Analog to digital converter)
- An output amplifier, to raise the power of output devices
- An input amplifier, same as above but for input devices (ex: microphones).
- Controls mechanism to allow different settings
- Hardware mixer, which controls each devices volume and routing, usually volume is measured in decibel.
- A MIDI (Musical Instrument Digital Interface) device/controller, a standard unified protocol to control output devices (called synthesizers) — think of them like keyboards for sounds.
- A sequencer, a builtin MIDI synthesizer (output of the above)
- A timer used to clock audio
- Any other special features such as a 3D spatializer

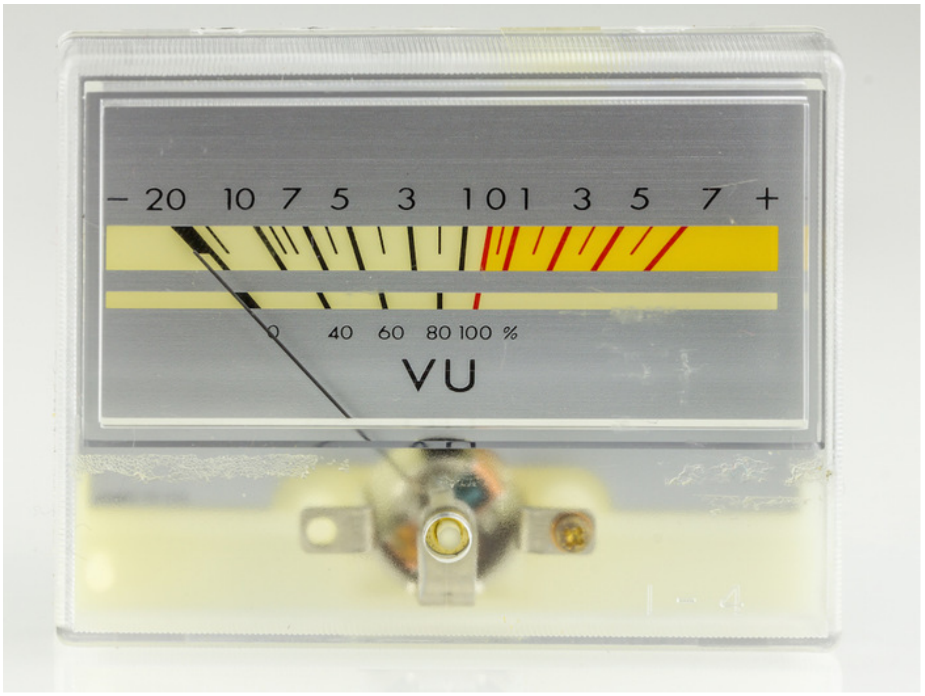
It is important to have a glance at these components because everything in the software layers attempts to make them easier to approach.

Analog to Digital & Digital to Analog (ADC & DAC)

A couple of concepts related to the interaction between the real and digital world are also needed to kick-start our journey.

In the real world, the analog world, sound is made up of waves, which are air pressures that can be arbitrarily large.

Speakers generating sound have a maximum volume/amplitude, usually represented by 0dB (decibels). Volume lower than the maximum is represented by negative decibels: -10dB, -20dB, etc.. And no sound is thus -∞ dB.
This might be surprising and actually not really true either. Decibel doesn't mean much until it's tied to a specific absolute reference point, it's a relative scale. You pick a value for 0dB that makes sense for what you are trying to measure.



vu meter

The measurement above is the dBFS, the dB relative to digital full-scale, aka digital 0. There are other measurements such as dB SPL and dBV.
One thing to note about decibels is that they follow a strictly exponential law, which matches it to human perception. What sounds like a constantly increasing volume is indicated by a constantly rising dB meter, corresponding to an

exponentially rising output power. This is why you can hear both vanishingly soft sounds and punishingly loud sounds. The step from the loudest you can hear up to destroying your ears or killing you is only a few more dB.

While decibels are about loudness, the tone is represented as sine waves of certain frequency, the speed. For example, the note A is a 440Hz sine wave.

Alright, we got the idea of decibel and tone but how do we get from waves to our computer or in reverse? This is what we call going from analog to digital or digital to analog.
To do this we have to convert waves into discrete points in time, taking samples per second — what we call sample rate. The higher the sample rate, the more accurate the representation of the analog sound (a lollipop graph). Each sample has a certain accuracy, how much information we store in it, the number of bits for each sample — what we call the bit rate/depth (the higher the less noise). For example, CDs use 16 bits.
Which value you choose as your sample rate and bit rate will depend on a trade-off between quality and memory use.

NB: That's why it makes no sense to convert from digital low sample rate to digital high sample rate, you'll just be filling the void in the middle of the discrete points with the same data.

Additionally, you may need to represent how multiple channels play sound — multichannel. For example, mono, stereo, 3d, surround, etc..

It's important to note that if we want to play sounds from multiple sources at the same time, they will need to agree on the sample rate, bit rate, and format representation, otherwise it'll be impossible to mix them. That's something essential on a desktop.

The last part in this equation is how to implement the mechanism to send audio to the sound card. That highly depends on what the card itself supports, but the usual simple mechanism is to fill buffers with streams of sound, then let the hardware read the samples, passing them to the DAC (digital to analog converter), to then reach the speaker, and vice versa. Once the hardware has read enough samples it'll do an interrupt to notify the software side that it needs more samples. This cyclic mechanism goes on and on in a ring fashion.
If the buffer samples aren't filled fast enough we call this an underrun or drop-out (aka xruns), which result in a glitch, basically audio stopping for a short period before the buffer is filled again.

The audio can be played at a certain sample rate, with a certain granularity, as we've said. So if we call *buffer-size* the number of samples that can be contained in a cyclic-buffer meant to be read by the hardware, *fragment-size* or *period-size* the number of samples after which an interrupt is generated, *number-fragments* the number of fragments that can fit in a hardware buffer (*buffer-size/fragment-size*), and *sample-rate* the number of samples per seconds.
Then the latency will be *buffer-size/sample-rate*, for example if we can fit 100

samples in a buffer and the samples are played once every 1ms then that's a 100ms latency; we'll have to wait 100ms before the hardware finishes processing the buffer.
From the software side, we'll be getting an interrupt every *period-size/sample-rate*. Thus, if our *buffer-size* is 1024 samples, and *fragment-size* is 512, and our *sample-rate* is *44100 samples/second*, then we get an interrupt and need to refill the buffer every *512/44100 = 11.6 ms*, and our latency for this stage is up to *1024/44100 = 23 ms*. Audio processing pipelines consist of a series of buffers like this, with samples read from one buffer, processed as needed, and written to the next.

Choosing the values of the *buffer-size* and *period-size* are hard questions. We need a buffer big enough to minimize underruns, but we also need a buffer small enough to have low latency. The fragments should be big enough to avoid frequent interrupts, but we also need them small enough so that we're able to fill the buffer and avoid underruns.
What some software choose to do is to not follow the sound card interrupts but to rely on the operating system scheduler instead, to be able to rewrite the buffer at any time so that it stays responsive to user input (aka buffer rewinding). This in turn allows to make the buffer as big as possible. Though, timers often deviate, but that can be fixed with good real-time scheduling.
There are no optimal solutions to this problem, it will depend on requirements, and these values can often be configured.

So we've seen how sound is represented in the real world with strength and tone, to then be converted in the digital world via digital to analog DAC or analog to digital ADC converters. This is done by taking samples at a rate and of a certain accuracy called the bit rate. There can also be other information needed such as the channel, byte ordering, etc.. Sound that needs to be mixed needs to agree on these properties. Lastly, we've seen how software has to manage a buffer of samples so that sound plays continuously on the device, while also being responsive to users.

Libraries

Audio related libraries seem to be an alphabet soup of keywords. Here are some examples: alsaplayer-esd, libesd-alsa, alsa-oss, alsaplayer-jack, gstreamer-alsa, gstreamer-esd, lib-alsa-oss, libpulse, libpulse-simple, libao, and so on.

For programs to be able to use audio hardware and the related functionalities, they rely on libraries offering specific APIs. Over time, some APIs get deprecated and new ones appear. Thus, that creates a situation where multiple software speak differently.
To solve this issue, many glue libraries have appeared to interoperate between them. This is especially true when it comes to sound servers such as aRts, eSD, PulseAudio, and backends. For example ALSA supports an OSS layer, that is the role of lib-alsa-oss.

Apart from libraries used to play or record sound and music, there are libraries that have specific usages.

GStreamer is a popular library for constructing chains of media-handling components. It is the equivalent of a shell pipeline for media. This library is used in multiple software in the GNOME desktop environment. For example, cheese (webcam) uses it to add video effects on the fly. Keep this in mind as the creator of GStreamer is now working on PipeWire and applying some of the mindset and functionalities there.

libcanberra is a library that implements a freedesktop.org specs to play event sounds. Instead of having to play event sounds by loading and playing a sound file from disk every time, desktop components should instead use this library which abstract the lower level layer that will handle playing it on the appropriate backend. It's important considering what we said about them changing over time.
The freedesktop.org event sound files can usually be found in: */usr/share/-sounds/freedesktop/stereo/*, and you can test by calling on the command line:

```
canberra-gtk-play -i bell
canberra-gtk-play -i phone-incoming-call
```

There are also multiple libraries used to abstract the audio backend of any OS, so called cross-platform audio libraries. This includes libraries such as PortAudio (software using it), OpenAL that focuses on 3D audio, libSDL, and libao.

Lastly, there is LADSPA, the *Linux Audio Developer's Simple Plugin API*, which is a library offering a standard way to write audio filter plugins to do signal processing effects. Many programs and libraries support the format including ardour, audacity, GStreamer, Snd, ALSA (with plugin), and PulseAudio (with plugin).

We've seen multiple usages for libraries, from their use as glue, to them helping in chaining audio, to desktop integration, to cross-platform interaction, and to allow a common format for audio filters.

Audio Driver

For us to be able to use the audio hardware components we mentioned, we need a way to communicate with them, what we call a driver. That job is done dynamically by the kernel which loads a module when it encounters a new device.

Every platform got its device management mechanism, be it devd on FreeBSD, systemd-udev, Gentoo's eudev, Devuan's vdev, mdev from BusyBox or Suckless, etc..
For example, on Linux you can take a look at the currently connected components and the driver handling them by executing *lspci*. Similarly, on FreeBSD this is done with the *pciconf -l* command.

To be handled properly, the hardware needs an appropriate driver associated with it.

On Linux, the ALSA kernel layer handles this automatically. The driver names start with the prefix _snd___. Issuing *lsmod* should show a list of them. (supported cards)
In case the device doesn't get associated with the right driver, you can always create specific rules in the device management (udev).

On FreeBSD, the process takes place within the kernel sound infrastructure and is controlled dynamically at runtime using *sysctl* kernel tunables. We'll see how to tune drivers settings in another section, as this is how you interact with them on BSD. The process is similar on most BSDs.
If the driver doesn't load automatically you can always manually activate the kernel module. For example, to load the Intel High Definition Audio bridge device driver on the fly:

```
$ kldload snd_hda
```

Or to keep them always loaded you can set it at boot time in `/boot/loader.conf`:

```
snd_hda_load="YES"
```

On BSDs and Linux the drivers, OSS-derived and ALSA in the kernel, then map the components within the file system, they are the reflection of the hardware we've seen before. Mostly input, output, controllers, mixers, clocks, midi, and more.

On FreeBSD the sound drivers may create the following device nodes:

- `/dev/dsp%d.p%d` Playback channel.
- `/dev/dsp%d.r%d` Record channel.
- `/dev/dsp%d.%d` Digitized voice device.
- `/dev/dspW%d.%d` Like /dev/dsp, but 16 bits per sample.
- `/dev/dsp%d.vp%d` Virtual playback channel.
- `/dev/dsp%d.vr%d` Virtual recording channel.

- `/dev/audio%d.%d` Sparc-compatible audio device.
- `/dev/sndstat` Current sound status, including all channels and drivers.

Example of status:

```
$ cat /dev/sndstat
FreeBSD Audio Driver (newpcm: 64bit 2009061500/amd64)
Installed devices:
pcm0: <NVIDIA (0x001c) (HDMI/DP 8ch)> (play)
pcm1: <NVIDIA (0x001c) (HDMI/DP 8ch)> (play)
pcm2: <Conexant CX20590 (Analog 2.0+HP/2.0)> (play/rec)
   default
```

On OpenBSD it's similar, a SADA-like driver (Solaris Audio API), that has a different and much simpler mapping:

- `/dev/audioN` Audio device related to the underlying device driver (for both playback and recording)
- `/dev/sound` same as `/dev/audioN`, for recording and playback of sound samples (with cache for replaying samples)
- `/dev/mixer` to manipulate volume, recording source, or other mixer functions
- `/dev/audioctlN` Control device, accept same ioctl as `/dev/sound`
- `/dev/midiN` Control device

On Linux, the ALSA kernel module also maps the components to operational interfaces under */dev/snd/*. The files in the latter will generally be named aaaCxDy where aaa is the service name, x the card number, and y the device number. For example:

- `pcmC?D?p` pcm playback devices
- `pcmC?D?c` pcm capture devices
- `controlC?` control devices (i.e. mixer, etc.) for manipulating the internal mixer and routing of the card
- `hwC?D?` hwdep devices
- `midiC?D?` rawmidi devices - for controlling the MIDI port of the card, if any
- `seq` sequencer device - for controlling the built-in sound synthesizer of the card, if any
- `timer` timer device - to be used in pair with the sequencer

The devices will mostly be mapped as either PCM devices, pulse-code modulation — the digital side of the equation, or as CTL devices, the controller and mixer, or as MIDI interface, etc..

The driver status and configuration interface is in the process information pseudo-filesystem under */proc/asound* (instead of kernel tunable like on most BSDs).
The following long list should give you an idea of what's available:

- `/proc/asound/`
- `/proc/asound/cards` (RO) the list of registered cards
- `/proc/asound/version` (RO) the version and date the driver was built
- `/proc/asound/devices` (RO) the list of registered ALSA devices (major=116)
- `/proc/asound/hwdep` (RO) the list of hwdep (hardware dependent) controls
- `/proc/asound/meminfo` (RO) memory usage information this proc file appears only when you build the alsa drivers with memory debug (or full) option so the file shows the currently allocated memories on kernel space.
- `/proc/asound/pcm` (RO) the list of allocated pcm streams
- `/proc/asound/seq/` the directory containing info about sequencer
- `/proc/asound/dev/` the directory containing device files. device files are created dynamically; in the case without devfs, this directory is usually linked to `/dev/snd/`
- `/proc/asound/oss/` the directory containing info about oss emulation
- `/proc/asound/cards` info about cards found in cardX sub dir
- `/proc/asound/cardX/` (X = 0-7) the card-specific directory with information specific to the driver used
  - `id` (RO) the id string of the card
  - `pcm?p` the directory of the given pcm playback stream
  - `pcm?c` the directory of the given pcm capture stream
  - `pcm??/info` (RO) the pcm stream general info (card, device, name, etc.)
  - `pcm??/sub?/info` (RO) the pcm substream general info (card, device, name, etc.)
  - `pcm??/sub?/status` (RO) the current of the given pcm substream (status, position, delay, tick time, etc.)
  - `pcm??/sub?/prealloc` (RW) the number of pre-allocated buffer size in kb. you can specify the buffer size by writing to this proc file

For instance we can issue:

```
$ cat /proc/asound/cards
 0 [HDMI           ]: HDA-Intel - HDA ATI HDMI
                      HDA ATI HDMI at 0xf0244000 irq 32
 1 [Generic        ]: HDA-Intel - HD-Audio Generic
                      HD-Audio Generic at 0xf0240000 irq
                          16
 2 [LX3000         ]: USB-Audio - Microsoft LifeChat
    LX-3000
                      C-Media Electronics Inc. Micros...
```

That gives us an idea of how different Unix-like OS dynamically load the driver for the device, and then maps it to the filesystem, often also giving an interface to get their status and configure them. Now let's dive into other aspects of OSS and ALSA, more on the user-side of the equation.

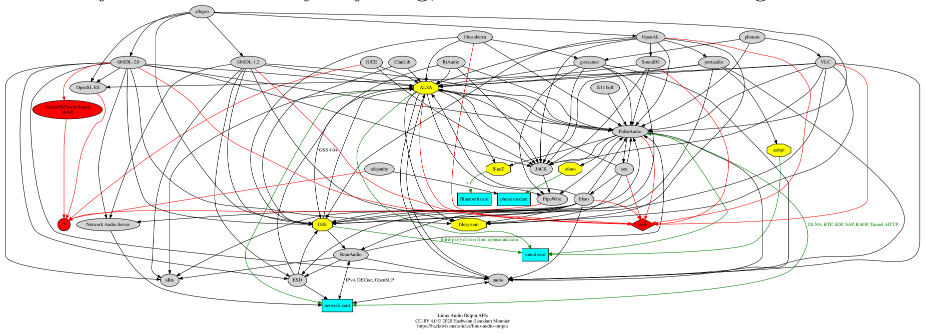
We now have an overview of:

- The basic logical components a card can have (input/output devices, mixers, control mechanisms, etc..)
- How to go from analog to digital and vice-versa
- Some libraries and why software use different ones
- The mapping of hardware devices to the filesystem when discovered

Advanced Linux Sound Architecture (ALSA)

Now let's dive into ALSA in particular and see what's the deal with it.

If you want to get dizzy you can look at this spaghetti diagram. It does more to confuse you than to clarify anything, so it fails as far as meaning is concerned.



ALSA, the Advanced Linux Sound Architecture is an interface provided by the Linux kernel to interact with sound devices.
We've seen so far that ALSA is a kernel module and is responsible for loading drivers for the appropriate hardware, and also maps things in the filesystem on */proc/asound* and in */dev/snd*. ALSA also has a library, a user-facing API for real and virtual devices, and configuration mechanisms that let you interact with the internal audio concepts. Historically, it was designed to replace OSS (Open Sound System) on Linux, which we'll see in the next section. ALSA provides an OSS emulation if needed.

Some features that are often cited:

- Up to 8 audio devices at the same time, modularized
- MIDI functionality like Hardware-based MIDI synthesis.
- Perform hardware mixing of multiple channels
- Full-duplex operation.
- Multiprocessor-friendly
- thread-safe device drivers

Let's see the following: How ALSA represents devices, what are PCM and CTL, plugins, configurations, and tools.

ALSA is good at doing automatic configuration of sound-card hardware. It does that by grouping different cards based on "chipset" and families, — similar cards will have similar interfaces. It also fills the gap by using plugins when it comes to the name of controls by deliberately keeping them similar. For example, the master volume is always called "Master Volume", even when not physically there, the abstraction will exist as a software control plugin.
This grouping allows developers to interact with sound devices in a unified way which makes it much simpler to write applications.

We've previously seen how ALSA maps devices in entries in */dev/snd* (pcm,

control, midi, sequencer, timer) with their meta-information in */proc/asound*. Moreover, ALSA split devices into a hierarchy.
ALSA has the concept of cards, devices, and subdevices.

A card is any audio hardware, be it a USB audio headset, an audio chip, or virtual sound card, etc.. Real hardware are backed by kernel drivers while virtual ones live in user-space. It has 3 identifiers:

- A number, which is incremental after each new insertion (so could change after reboot)
- An ID, which is a text identifier for a card. This is more unique and consistent
- A name, another text identifier, but not a useful one.

Devices are subdivision of a card, for playback or capture. For example it could be “analog input + output”, “digital output”, etc.. It dictates the type of device that the card is, what it can do and is capable of processing. A sort of “profile” for the card. Same as with cards, devices have three identifiers: Number, ID, and Name.
Only one device is active at a time, because the device is the current “function” that the card takes.

Devices themselves have at least one subdevice. All subdevices share the same playback (output) or recording (input) stream. They are used to represent available slots for hardware mixing, joining audio in hardware. However, hardware mixing is rarely used so there is usually a single subdevice unless it is a surround sound system.

Overall, that gives us this type of notation.

```
card CARD_NUMBER: CARD_ID [CARD_NAME], device
   DEVICE_NUMBER: DEVICE_ID [DEVICE_NAME]
Subdevice #SUBDEVICE_NUMBER: SUBDEVICE_NAME
Example:
card 2: LX3000 [Microsoft LifeChat LX-3000], device 0:
   USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

Alternatively, you can go directly to the */proc* tree, which we’ve seen previously, and list cards with *cat /proc/asound/cards*.

```
 2 [LX3000         ]: USB-Audio - Microsoft LifeChat
    LX-3000
                      C-Media Electronics Inc. Microsoft
                      LifeChat LX-3000 at
                         usb-0000:00:12.0-4,
                      full
```

A common notation for ALSA devices looks like *hw:X,Y* where *X* is the card id and *Y* the subdevice id. You do not need the device id because only one can be active at a time.

You can list playback devices by issuing *aplay -l* and recording devices with *arecord -l* (and MIDI devices with *amidi -l*).
Another useful script to dump info about the system is *alsa-info.sh*.

All ALSA clients have to interface with objects in the ALSA world, the most important two are the PCM (pulse code modulation) and the CTL (control) which we've briefly mentioned before.
PCM objects are the representation of a sound stream format used for data flow. These PCMs can be chained and are typically attached to a hardware at one end, but can also be attached to other things such as the filesystem, a server, or even dropping audio completely. When PCMs are chained we call them slave PCM, a virtual PCM, which is an extra step of indirection. The PCM streams connected to the hardware need to follow its characteristics: that is they need to have the right sample rate, bit rate (sample width), sample encoding (endianess), and number of channels (mono, stereo, etc..). List them using *aplay -L*.
CTL objects are control objects telling ALSA how to process non-audio data. That includes things such as volume controls, toggle controls, multiple-choice selections, etc.. These controls can be put in one of 3 categories: playback control (for output device), capture control (for input device), and feature control (for special features).
There are other ALSA objects such as MIXER and UCM (Use Case Manager - for presets separation, like notifications, voice, media, etc..) but they are not important to get the concept across so we'll skip them.

These objects can be defined, modified, and created in ALSA configurations and are often templatized and manipulated through plugins. A lot of them are automatically created by ALSA itself to create "profiles" for cards. Most of ALSA processing is delegated to plugins.
Clients will then read the configuration and most often use the default PCMs and CTL, if not selected explicitly by the user. Practically, that means the software will write or read audio from a stream (PCM) and control it (usually volume) through the CTL or MIXER interfaces.

For example:

- `aplay` and other players use the PCM interface
- `alsactl` uses the ctl (control) interface
- `amixer` uses the mixer interface
- `amidi` the rawmidi interface and so on.
- `alsaucm` the ucm interface

As far as configuration location is concerned, clients will load *alsa.conf* from the ALSA's data directory, so */usr/share/alsa/alsa.conf*, and in turn this configuration will load system- and user-wide configurations in */etc/asound.conf* and *~/.asoundrc* or *~/.config/alsa/asoundrc* respectively.

Change will take place as soon as clients re-read the configuration, normally when they are restarted.

ALSA configuration format is notoriously complex, almost Turing complete. It consists of a dictionary containing key/value pairs of names and object of a given type.
For example, the *pcm* key will contain the list of all PCM definitions, and the *ctl* key will contain the list of all CTL definitions.

The statements are of the form:

```
KEY1.KEY2.KEY3... VALUE
```

KEY1 being one of the object mentioned (*pcm*, *ctl*, and others).

The configuration format supports different value types, they could be either string, number, compound (using braces `{}`), or reference another value.

Additionally, the configuration is hyper flexible, allowing different ways to define the dictionary, which ALSA will later on resolve internally by adding them to its internal global dictionary that has all the keys it accumulated while reading the confs.
For instance, these are equivalent notations, from multiline definition, to using the `=` sign between params and values, comma or semicolon between consecutive param assignments. Like so:

```
pcm.a.b 4
pcm.a.c "hi"

is equivalent to

pcm.a {
    b 4
    c "hi"
}

is equivalent to

pcm.a = {
    b = 4;
    c = "hi";
};
```

The configuration format has special statements that begin with `@` such as *@func*, *@hooks*, and *@args* which have different behavior. *@func* is used to call functions, *@hooks* to load files, and *@args* to define internal variables that can be used within a compound variable type.

Example:

```
{ @func getenv vars [ ENVVAR1 ENVVAR2 ... ] default
    VALUE }
```

Will turn into a string from the specified environment variable. Each environment variable is queried and the first to match is used, otherwise VALUE.

Additionally, you can control how ALSA will act when it finds conflicting entries in the dictionary, how it will merge them. This is done by prefixing the key with one of the following:

- `!` the exclamation mark will cause the previous definition to be overridden instead of adding new values, removing all of the param and sub-param. (`!pcm` would delete all that is under `pcm`)
- `?` the question mark will ignore the assignment if the param exists
- `+` and `-` respect the type of any earlier assignment, `+` creates a new param when necessary, `-` causes error if param didn't previously exist.

Alright, so now we know how to create a gigantic dictionary of ALSA sound-related objects, how do we actually make use of them?
What we do is create a name under one of these object and give it a *type*. This type is a plugin that will dictate what to do with this object. The plugins take different configurations depending on what they do, so you'll have to consult the docs. That gives rise to something like this:

```
pcm.NAME {
    type TYPE
    ...
}

ctl.NAME {
    type TYPE
    ...
}

slave_pcm.NAME {
    pcm PCM
    ...
}
```

So ALSA consists mostly of plugins. You can find the external ones that were installed in: */usr/lib/alsa-lib*, and the others are internal. For example, check the documentation for the internal pcm plugins.

For obvious reasons, the most important *pcm* plugin is the *hw* hardware one, which is used to access the hardware driver. This plugin takes as parameters things that we mentioned such as the card number, the device, subdevice, the format, rate, channels, etc..

Now things should start to make sense: We have clients reading ALSA configurations and manipulating objects having a common interface, which are handled in the backend by plugins, which often end up on the hardware.

Another important plugin is *plug* which performs channel duplication, sample value conversion, and resampling when necessary. That is needed if a stream has the wrong sample rate for a hardware. You can use *aplay -v* to make sure the resampling actually happens.

Yet another one is the *dmix* pcm plugin, the direct mix plugin, which will will merge multiple pcm streams into an output pcm. This is software mixing, which is much more prominent these days compared to hardware mixing.

There really is a long list of fun plugins that can be used in different scenarios, so take a look.

Example:

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

This creates a device called *plugger* that respect the object interface of *pcm*. Whatever is written or read from this *plugger* will be handled by the *plug* plugin, which in turn will use a slave PCM device *hw:0,0*.
Notice how we used the word “device”, that is because any *pcm* connected to a hardware corresponds to an ALSA device. It should start making sense now. These *pcm*, for instance, are the ones that get listed when you issue *aplay -L* or *arecord -L* and these are the objects that clients will interact with — they don’t know if these are even connected to a card or not.

The special name *default* is used to specify the default object interface. So to set, and override, the default playback device you can do:

```
pcm.!default "hw:CARD"
```

ALSA provides a lot of these objects preconfigured, as generic device templates. However, sometimes it requires a bit of fiddling to get right and this isn’t obvious to everyone considering the complexity of ALSA configs.
Sometimes it’s also hard to find a plugin for your case. For example, projects like *alsaequal* creates an audio equalizer, and project *alsa_rnnoise* creates a pcm device that will remove noise.
These difficulties are part of the reasons why we use sound servers, which we’ll see in their own sections.

To get a glimpse at what the final ALSA configuration tree is like, I’ve made a small script that will dump all the configuration.

So, we’ve seen ALSA’s representation of components, the common objects such as PCM and CTL, how to create them in the flexible configuration format, how the configuration is mostly plugins, how these plugins will use the sound components, and how clients use the PCM without caring what’s happening under the hood.

Open Sound System (OSS) and SADA

OSS, the Open Sound System, is the default Unix interface for audio on POSIX-compatible systems. Unfortunately, like such standards, it isn't compatible everywhere. It can be perplexing to understand because different systems have branched out of it.

Untill OSS version 3 Linux was using OSS. The company developing it, 4Front Technology, chose in 2002 to make OSSv4 a proprietary software, then in 2007 they re-released it under GPL.
For that reason OSSv4 isn't used as the default driver of any major OS these days, but can still be installed manually. However, not many applications have support for it and it might requires a translation layer.

In the meantime, Linux had switched to ALSA because of the license issues and the shortcomings of OSS, namely it couldn't play multiple sounds simultaneously, allocating the sound device to one application at a time, and wasn't very flexible.
Similarly, in the BSD world some chose to continue to extend OSS and some only got inspired by it to do something else. FreeBSD continued with its fork of OSS by reimplementing the API and drivers and improving it along the way. They added in-kernel resampling, mixing, an equalizer, surround sound, bit-perfect mode (no resampling or mixing), and independent volume control per application.
On the other side, NetBSD and OpenBSD chose to go with their own audio API that is Sun-like (SADA, Solaris Audio API aka devaudio), with an OSS compatibility mode to keep backward compatibility.
Solaris an OpenSolaris use a fork of OSSv4 called Boomer that combines OSS together with Sun's earlier SADA API, similar to what OpenBSD does.

Due to this arduous history, it is not guaranteed that any of these OS will use a compatible OSS version or OSS layer.

Like ALSA, OSS audio subsystem provides playback, recording, controller, MIDI, and others. We've seen that these are mapped to special files by the driver, all starting with */dev/dsp*. */dev/sndstat* can be used to list which driver controls which device.
Unlike ALSA where clients interact with PCM and CTL objects, in OSS there is a common API that is used to interact with the special files that were mapped on the filesystem. That means that developers will rely on a more Unix/POSIX like model using the common system calls like *open*, *close*, *read*, *write*, *ioctl*, *select*, *poll*, *mmap*, instead of custom library functions.

What these functions do depends on the OS and OSS version. For example, *ioctl* lets you interact with the generic device type features, as can be seen here. That gives rise to much simpler programs, check this example

One thing the FreeBSD audio frameworks support is the use of mmap to allow applications to directly map the audio buffer, and use *ioctl* to deal with head/tail synchronization. ALSA on Linux does the same.

Similarly to OSS, on OpenBSD and NetBSD, the 3 exposed devices */dev/audioN*, */dev/audioctlN*, and */dev/mixerN* can be manipulated with *read*, *write*, and mostly *ioctl*. You can take a look in the *man 4 audio* to get an idea.

When it comes to configuring specific OS related functionalities such as how to mix sound, selecting the sample rates and bit rates, choosing the default output and input hardware, etc.. That depends entirely on the implementation of the particular operating system.

On FreeBSD, audio configurations are available by configuring kernel tunables via *sysctl* or set statically at boot. For example, *dev.pcm.0* is the first instance of the pcm driver and *hw.usb.uaudio* is the usb audio hardware settings. You'll known which one is which by consulting */dev/sndstat*.
Setting the default sound device on FreeBSD:

```
sysctl hw.snd.default_unit=n
```

Where n is the device number.

You can also set the default value for the mixer:

```
sysctl hint.pcm.0.vol="50"
```

As you can notice the mixer is part of the pcm driver. This driver supports Volume Per Channel (VPC), that means you can control the volume of each application independently.

As for OSSv4, it offers configuration files for each driver, along with its own set of tools like *ossinfo*, *ossmix*, *vmixctl*, etc..
The configurations can be found under */usr/lib/oss/conf/*, the *$OSSLIBDIR*. It contains audio config files for different drivers with their tunables. It can help set basic settings such as: virtual mixers, quality of sound, sample rate, etc.. You can consult the man page of each driver to check their settings *man 7 oss_usb*.
Note that OSSv4 isn't as flexible and requires turning the sound on and off for the settings to take effect.

On OpenBSD and NetBSD, similarly to FreeBSD, settings are done through kernel tunable. The SADA-like system also has multiple tools such as *mixerctl* and *audioctl* to make it easier to interact with the audio driver variables.
For example, you can change the sample rate on the fly with:

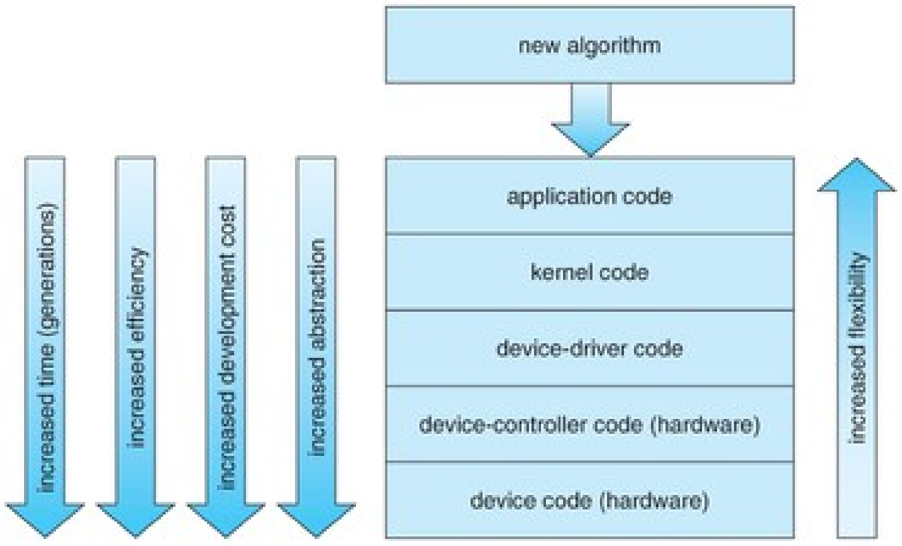
```
audioctl -w play.sample_rate=11025
```

OpenBSD stays true to its security and privacy aspect by disabling recording by default, which can be re-enabled with:

```
$ sysctl kern.audio.record=1
$ echo kern.audio.record=1 >> /etc/sysctl.conf
```

As you can see, multiple systems using different APIs and configurations isn't practical and very limiting. That is why OpenBSD created a layer on top called sndio, a sound server that we'll discover in a bit.

Overall, we've seen the idea of the OSS-like systems. They expose devices in the file system and let you interact with them through the usual Unix system calls. To configure these devices you have to use whatever way the OS gives you, mostly kernel tunables.
While kernel tunables don't offer the crazy configurability that ALSA does, these audio stacks can still give more or less the same basic features by having them in the kernel. Like on FreeBSD, where the mixing of streams and volume per application control happens out of sight.
As with anything, the lower in the stack it is, the less flexible it is, but the more stable, efficient, and abstract it is.



complexity of system

Sound Servers

Sound servers are neither kernel modules nor drivers, but are daemons that run in user-space to provide additional functionalities. They are there to both raise the flexibility and to have a more abstract and common layer.

The operations a sound server allow range from transferring audio between machines, to resampling, changing the channel count, mixing sounds, caching, adding audio effects, etc.. Having these operations done in a modular way in a sound server is more advantageous than having them in the kernel. Moreover, having a sound server could mean having the same API for all audio regardless of the underlying kernel module or API used. So you won't have to worry about the interoperability of running on OSS or ALSA or anything else.

There are many sound servers available, some are deprecated like aRts and ESD, others are in use such as sndio, JACK, and PulseAudio, and new ones are coming out and being pushed into distributions like PipeWire. Every one of these has different features, supports for all kinds of protocols, and run on multiple operating system flavors.

sndio

sndio is the default sound server on BSDs today. It is a small, compact, audio and MIDI framework and user-space server developed by OpenBSD. The server is so simple it doesn't even need a configuration.

sndio's main roles are to abstract the underlying audio driver and to be a single point of access instead of requiring each application to get raw access to the hardware.
Having a sound server solves the issue of the fracture between all OSS implementations. It creates a new standardized layer.

Let's mention some of sndio features:

- Change the sound encoding to overcome incompatibilities between software and hardware. (can change sample rate and bit rate)
- Conversions, resampling, mixing, channel mapping.
- Route the sound from one channel to another, join stereo or split mono.
- Control the per-application playback volume as well as the master volume.
- Monitor the sound being played, allowing one program to record what other programs play.
- Use of ticking mechanism for synchronization (maintained after underruns, when buffer isn't filled fast enough) and latency control
- Support network connections

sndiod, the server, operates as follows: it creates a sub-device that audio programs connect to as if it was one of the device created by the driver (a real hardware). Thus, during playback or recording, sndiod will handle the audio streams and commands for all programs and take care of the result on the fly. In sum, sndiod acts as a proxy while giving a similar interface as the kernel API on BSD (*read*, *write*, *ioctl*).

All programs connected to the same sub-device will be part of the same group, which gives a way to process and configure sound according to which sub-device is used.
These sub-devices are defined in a similar string format:

```
type[@hostname][,servnum]/devnum[.option]
```

Which, as you can see, allows to connect to a remote host.

Here are some examples:

```
snd/0
    Audio device of type snd referred by the first -f
        option passed
    to sndiod(8) server
snd/0.rear
    Sub-device of type snd registered with "-s rear"
        option
```

```
default
    Default audio or MIDI device.
```

The server *sndiod* doesn't have any configuration files, so everything is passed on the command line as an argument. Here are a couple examples of how to do that.

Start the server with a 48kHz sample rate, 240 frame block size (*fragment-size*), and 2-block buffers (`240*2`) (*buffer-size*) (See previous *Analog to Digital & Digital to Analog (ADC & DAC)* for more info on what these mean), this creates a 10ms latency.

```
$ sndiod -r 48000 -b 480 -z 240
```

Start *sndiod* by creating the *default* sub-device with low volume (65) and an additional sub-device called *max* with high volume (127). These will map to *snd/0* and *snd/0.max* respectively.

```
$ sndiod -v 65 -s default -v 127 -s max
```

This example create the `default` sub-device plus another sub-device that outputs to channels 2:3 only (the output speaker will depend on the card). These will map to `snd/0` and `snd/0.rear` respectively.

```
$ sndiod -s default -c 2:3 -s rear
```

The *sndioctl* utility is a helper tool for audio device status and control through sndio. For example, you can change the output level and mute certain sub-devices.

```
$ sndioctl output.level=+0.1
$ sndioctl output.mute=1
$ sndioctl -f snd/0 output[0].level+=0.1
```

The commands passed on the right depend on the actual audio device.

sndio is not only available for BSD, it also has a backend for ALSA, so it can run on top of it.
It is generally well supported by major softwares like media players and web-browsers. However, if a program cannot interface with sndio there are ALSA plugins that provice a PCM that can connect to a sndiod server.

In this sections we've seen sndio, a very simple sound server that creates sub-devices on the filesystem for any type of audio device, output, input, midi, control, etc.. By arbitraging resources, it is able to calibrate sound streams to fit the hardware sampling and bit rate support. We've also seen how to start the server that has no config, and how to use *sndioctl* to interact with it.

aRts (analog Real time synthesizer) and ESD or ESounD (Enlightened Sound Daemon)

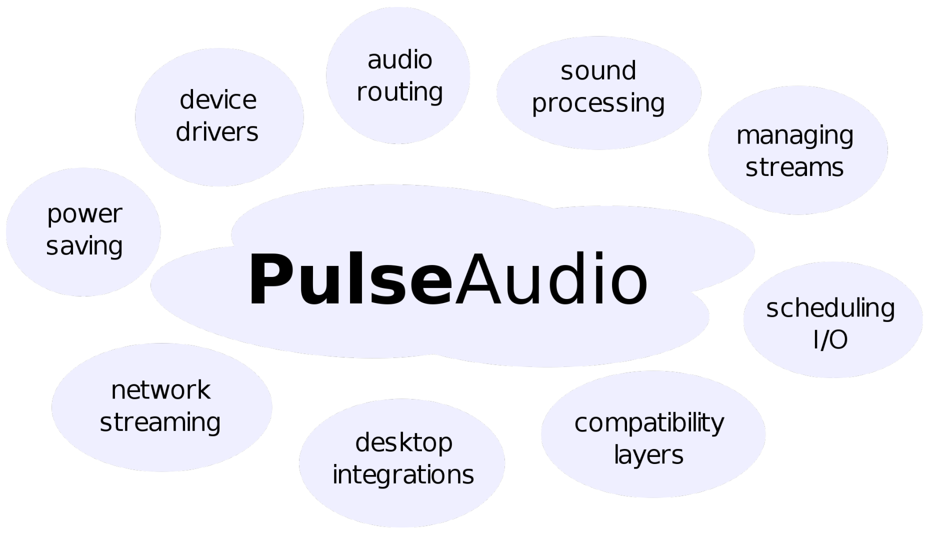
aRts and ESD or ESounD are two deprecated sound servers (audio daemons). Like all sound servers, they accept audio from applications and feed it to the hardware, while manipulating the stream format so that it fits it (resampling and others, you know the drill).

aRts was part of the KDE project and its main big cool feature was that it had a simulation of analog synthesizer.

EsounD was the sound server for the Enlightenment desktop and GNOME. It had similar functionality as any sound server but additionally it had two special cool features: desktop events sound handling, and a mechanism to pipeline audio and videos.

The different teams partnered to synchronize on their projects and have a single sound server. This split EsounD and aRts into pieces: the desktop events sound handling is now *libcanberra* (see the *Libraries* section), the pipeline of audio and video is now GStreamer (see the *Libraries* section), and the sound server was extracted unto PulseAudio (see next section on *PulseAudio*).

PulseAudio



PulseAudio

PulseAudio tends to trigger online flame wars, which are non-constructive.
Let's see what PulseAudio is, what features it has, some examples of it, its design, the definition of its internal objects, the sinks and sources, how it represents things we've seen such as cards/devices/profiles, how it is configured, how the server and clients start, the modular mechanism, what some modules/plugins do, the compatibility with different protocols, the desktop integration, some of the common frontends, how supported it is, and how to temporarily suspend it.
There's a long road ahead!

PulseAudio What Is It?

PulseAudio is a sound server for POSIX OSes, like sndio and others, so its job is similar: abstracting the interaction with the lower layers, whichever backend that is, and offering flexibility in audio manipulation — a proxy for sound applications.
Additionally, PulseAudio's main focus is toward desktop users, as it was primarily created to overcome the limitations of EsounD and aRts. It is heavily influence by Apple's CoreAudio design.

So far, PulseAudio is mainly targeted at the Linux desktop but there exists ports to other operating systems such as FreeBSD, Solaris, Android, NetBSD, MacOS X, Windows 2000, and Windows XP. However, some of the features require integration with the system and so are platform-specific, particularly: timer-scheduler, hot-plug features, and bluetooth interaction. The features in PulseAudio come as "external components" or also called "plugins", so these functionalities aren't inherent to the core server.

Let's have a look at a list of features that PulseAudio provides.

- Extensible plugin architecture (a micro-kernel arch with dynamically loadable modules via *dlopen*)
- A toolset to be able to manipulate the sound server on the fly
- Support for multiple input and output streams
- Flexible, implicit sample type conversion and resampling
- Ability to fully synchronize multiple playback streams
- Support interacting with audio streams of various protocols and backends, be them local or on a network
- Per application independent volume control
- Automatic management and setup of audio device, hotplug, via policies and restoration mechanism (mostly ALSA backend only)
- Sound processing ability and creation of audio pipeline chains: custom modules, mixing, sample rate conversion, echo cancellation, etc..
- Sample cache: in-memory storage for short sounds, useful for desktop events
- Low and accurate latency behaviour: uses features such as clocking and rewinding to keep the buffer responsive and avoid glitches. This is done via a timer-based scheduler per-device. (See previous *Analog to Digital & Digital to Analog (ADC & DAC)* for more info on what these mean) (ALSA backend only)
- Power saving: due to the use of default latency and timer-based scheduler per-device, there is no need to have a high number of interrupts. (ALSA backend only)
- Other desktop integration: X11 bells, D-Bus integration, Media role, hardware control, GConf, etc..

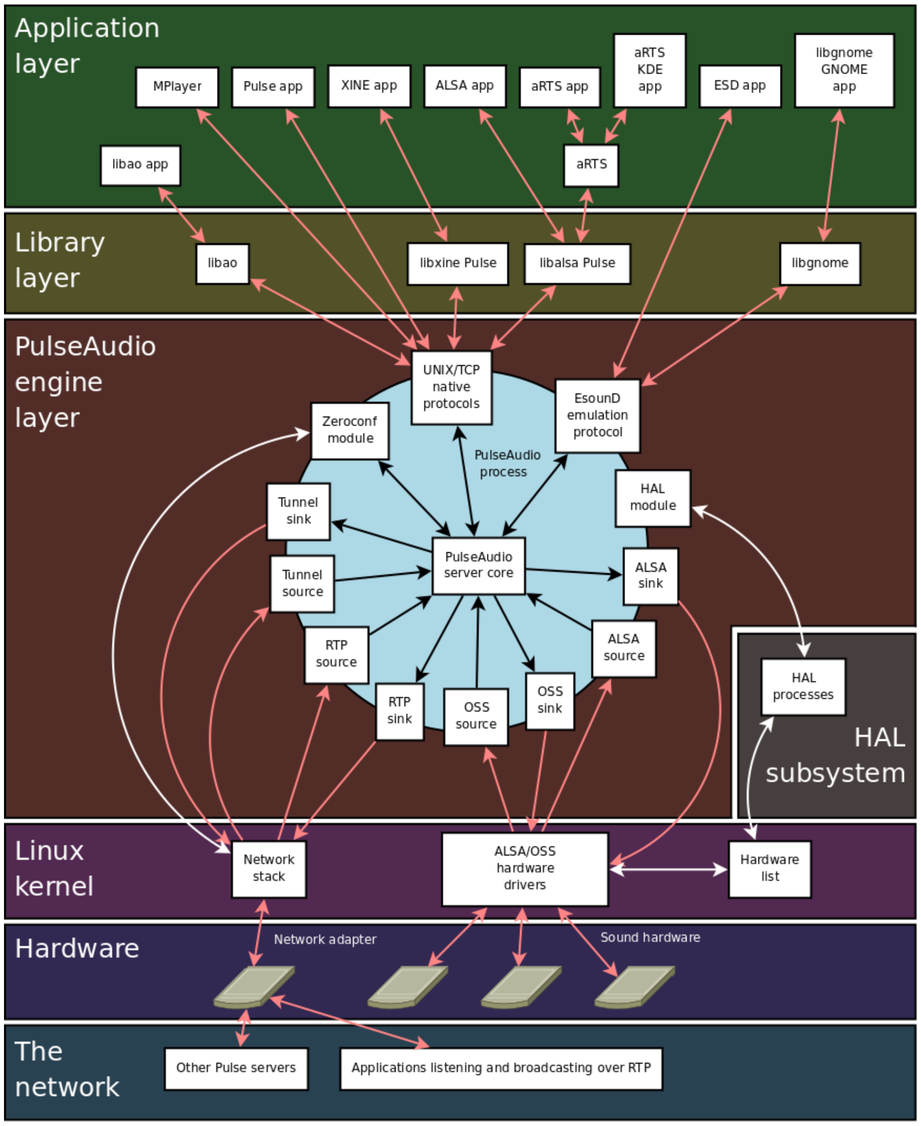
In practice that allows to do things like:

- Automatically setting up a USB headset when it's connected, remembering the configuration it was in the last time it was used.
- A GUI for controlling the sound of specific applications and deciding wether to move the audio stream from one device to another on the fly.
- Dynamically adding sound processing and filters to a currently running application, such as noise cancellation.

Pulseaudio Overall Design

The PulseAudio server consists of 3 logical components:

- A daemon: the piece that configures the core, loads the modules, and starts the main loop
- A core: based on *libpulsecore* this is a building block and shared environment for modules.



PulseAudio Engine Layer

- Modules: dynamically loaded libraries to extend the functionality of the server, relying on the *libpulsecore* library.

Inside the PulseAudio server lives different types of objects that we can manipulate, understanding these objects means understanding how PulseAudio works:

- format info
- source
- source output
- sink
- sink input
- card
- device port
- module
- client
- sample cache entry

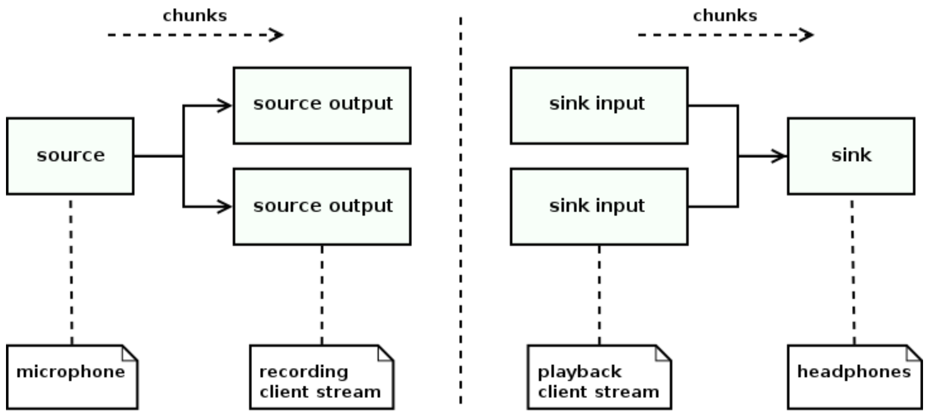
Pulseaudio Sink, Sink Input, Source, and Source Input

The source output and sinks inputs are the most important concepts in PulseAudio. They are the representation of audio streams. A source device generates a stream that is read/receive to a source output, like a process generating sound or a capture device, and a sink device is written/sent to via a sink input, like a sound card, a server, or a process.
In sum, sinks are output devices and sources are inputs devices: a source will be read unto a "source output" stream and the "sink input" stream will write to the sink device.
There can be virtual devices and virtual streams, and the sink input and source output can be moved from one device to another on the fly. This is possible because of the rewinding feature, each stream having its own timer-scheduler.

Additionally, sink monitors are always associated with a sink and get written to when the sink device reads from its sink inputs.

PulseAudio manages these stream in the ALSA backend using a timer-based scheduler to make them efficient for desktop. (See previous *Analog to Digital & Digital to Analog (ADC & DAC)* for more info on what these mean)
Every source, sink, source output, and sink input can have their own audio parameters, be it sample format, sample rate, channel map, etc.. The resampling is done on the fly and PulseAudio allows to select between different resamplers in its configuration and modules (*speex*, *ffmpeg*, *src*, *sox*, *trivial*,*copy*, *peaks*, etc..).
It's good to note that when multiple sink inputs are connected to the same sink then they automatically get mixed.
Each of these components can have their own volume. A configuration called "flat volume" can be set so that the same volume will be used for all sink inputs connected to the same sink.

This flexible concept of streams and devices can be used effectively with a couple



sink and source

of modules that allow juggling with them. For example, the module-loopback forwards audio from a source to a sink, it's a pair of source output and sink input with a queue in between. If you load it and the source is your microphone you'll then be able to hear your voice as echo.

```
pactl load-module module-loopback
pactl unload-module module-loopback # to unload it
```

Another example is the *module-null-source* and *module-null-sink* which will drop data. As with other sinks it will have a monitor associated with it, thus you can convert a sink-input to source-output, basically turning the audio that was supposed to be written to a device back as a readable stream.
We'll see more examples in the module section, but this is enough to whet your appetite.

You can use the tool *pacmd* to check each of these and the properties associated with the object:

```
pacmd list-sources
pacmd list-source-output
pacmd list-sinks
pacmd list-sink-inputs
```

For now, in the info shown from the above commands, you should at least understand a couple of the information shown such as the latency, the volume, the sample format, the number of channels, the resampling method, and others.

NB: the default source and sink are often abbreviated as *@DEFAULT_SOURCE@* and *@DEFAULT_SINK* in PulseAudio configs and commands.

Another example, changing the volume via *pactl* of a sink by its index:

```
pactl set-sink-volume 0 +5%
pactl set-sink-mute 0 toggle
```

Pulseaudio Internal Concepts: Cards, Card Profile, Device Port, Device

We got the idea of streams, but we still need to understand the mapping of actual devices into PulseAudio. This is represented by the concept of cards which could be any sound card or bluetooth device. A card has a card pofiles, device ports, and devices.

A card correspond to a mapping related to the driver in use. When using ALSA, this is the same card as an ALSA card.
For example:

```
aplay -l
card 2: LX3000 [Microsoft LifeChat LX-3000], device 0:
   USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0

pacmd list-sinks | grep card
  driver: <module-alsa-card.c>
  card: 5
     <alsa_card.usb-C-Media_Electronics_Inc._Micros...
    alsa.card = "2"
    alsa.card_name = "Microsoft LifeChat LX-3000"
    alsa.long_card_name = "C-Media Electronics Inc.
       Micros..."
```

The card has profiles which are, when using ALSA, equivalent to the list of *pcm* objects of ALSA that are attached to the hardware, which you can list using *aplay -L*.
You can see the list of profiles discovered by PulseAudio by doing *pacmd list-cards* and checking the *profiles* and *active profile* sections. Note that only one profile can be active for a card at a time.

In practice, a profile is an opaque configuration set of a card, defining the backend-specific configuration of the card, the list of currently available device ports, devices, configurations, and audio parameters. These are all templatized by ALSA as we've seen before.
ALSA manages this when it's the drivers, however PulseAudio doesn't take the profiles as they are, it sometimes uses a mapping to bind them into a new format that will result in a sink or source being created for the card. This is done via a configuration found in */usr/share/alsa-card-profile* and */usr/share/pulseaudio/alsa-mixer/*. You can have the mapping of different ALSA objects to PulseAudio ones in there.

For example, I have the following for the MIXER interface, which tells PulseAudio I can use mute, and capture:

```
/usr/share/alsa-card-profile/mixer/paths/iec958-stereo-input.conf
/usr/share/alsa-card-profile/mixer/paths/iec958-stereo-output.conf
```

After finding the right profile we can now know what are the device ports. They correspond to the single input or output associated with the card, like a microphone or speaker. Multiple device ports may belong to a card.
Then device, be it a source or sink, is the representation of the currently active producer or consumer, that is a card+a device port. For example, playing audio as digital stereo output on a USB headset.

Another mapping is possible using ALSA UCM (Use Case Manager) to group cards, but we'll skip over it. Use case management is used to abstract some of the object configuration like the MIXER (higher level management of CTL) so that you can play the same type of sounds together: notifications, media, video, VOIP, etc..

In summary, that gives the following relation between ALSA backend and PulseAudio objects.

- The PulseAudio ALSA backend will automatically create the cards, card profiles, device ports, sources, and sinks.
- A PulseAudio card is an ALSA card
- A PulseAudio card profile is an ALSA configuration for a certain card, this will dictate the list of available device ports, sources, and sinks for PulseAudio. These can be mapped using configs in a dir.
- A PulseAudio device port defines the active inputs and outputs for a card and other options, it's the selection of one profile function.
- Finally, the source and sink are associated with an ALSA device, a single *pcm* attached to the hardware. A source or sink get connected to a device port and that defines its parameters (sample rate, channels, etc..)

In general, whatever the backend, be it ALSA, OSS, or Bluetooth, PulseAudio's goal is to find out what inputs and outputs are available and map them to device ports.

Pulseaudio Everything Is A Module Thinking

As far as the PulseAudio server is concerned, it only manipulates its internal objects, provides an API, and doesn't do anything else than host modules. Even the backends like ALSA are implemented as modules.
That gives rise to a sort of micro-kernel architecture where most of the functionalities in the server are implemented in modules, and there's a module for everything. Most of what we've mentioned already is done via a module. Let's still show a list of some of them.

- Device drivers

- Protocols
- Audio routing
- Saving information
- Trivia like x11 bell
- Volume control
- Bluetooth
- Filters and Processing

Some modules are autoloaded in the server like the *module-native-protocol-unix*, which is PulseAudio's native protocol, and others are loaded on the fly.
Even the protocol to load modules and interact with the server via the command-line interface is a module in itself: *module-cli-protocol-unix/tcp*.
If you are interested in knowing about the ALSA backend integration, it is done by the *module-alsa-card*.

There's really a lot of modules, the list of the ones that come with the PulseAudio server default installation can be found here and here. There are also many user contributed modules, which you can place on disk in your library path */usr/lib/pulse-/modules/*.

To list the currently loaded modules use the following:

```
pacmd list-modules
```

NB: It can be quite eery to see that PulseAudio has its own module mechanism while we've seen earlier that ALSA does a similar thing through its configuration. However, keep in mind that PulseAudio is relatively easier to use, can work on top of different backends, not only ALSA, and has a different concept when it comes to audio streams (source-output and sink-input).

Now let's see how to load new modules and configure them.

Pulseaudio Startup Process And Configuration

Before we see how to load modules into the server, we first need to check how to run the server.
The PulseAudio server can either run in system-wide mode or per-user basis. The latter is preferred as it is better for desktop integration because some modules use the graphical desktop. It is usually started during the setup of the user session, which is taken care of by the desktop environment autostart mechanism. These days, with the advent of the systemd framework project, PulseAudio is often launched as a user service.

```
$ systemctl --user status pulseaudio
 pulseaudio.service - Sound Service
     Loaded: loaded
        (/usr/lib/systemd/user/pulseaudio.service;
        enabled; vendor preset: enabled)
     Active: active (running) since Sat 2021-02-06
        14:36:22 EET; 19h ago
TriggeredBy:   pulseaudio.socket
   Main PID: 2159374 (pulseaudio)
     CGroup:
        /user.slice/user-1000.slice/user@1000.service/app.…sli
             2159374 /usr/bin/pulseaudio --daemonize=no
                --log-target=journal
             2159380 /usr/lib/pulse/gsettings-helper
```

Another way to start the PulseAudio server is to not start it all. That's surprising, but with the default configuration clients will autospawn the server if they see that it's not running.
The reverse is also true, there is a configuration to autoexit when no clients have used the server for a certain period.

This is how clients start:

- Initialization: finding the server address from somewhere (environment variable, X11 root window, per-user and system-wide client conf files)
- connect: depending on the protocol used (native, tcp localhost, or remote)
- autospawn: if enable spawn a server automatically
- authenticate: using cookies found somewhere (environment variable, X11 root window, explicit, per user or system wide conf, per-use home dir)

The server starts by reading the server configurations, and then loading the modules found in the configuration associated with its running stance (system mode or per-user).
The server configurations are found by first looking in the home directory *~/.config/pulse*, and if not found then by looking in the system-wide config in */etc/pulse*. The directory will contain the following configuration files: *daemon.conf* *default.pa* *system.pa* and *client.conf*.

daemon.conf: contains the settings related to the server itself, things like the base sample rate to be used by modules that will automatically do resampling, the realtime scheduling options, the cpu limitation, if flat-volume will be used or not, the fragment size, latency, etc.. These cannot be changed at runtime.
You can consult *pulse-daemon.conf(5)* manpage for more info.

client.conf, this is the file that will be read by clients, which we mentioned above. It contains runtime options for individual clients.
See *pulse-client.conf(5)* manpage for more info on this one.

default.pa and *system.pa* are the per-user and system-wide startup scripts to load and configure modules. Once the server has finished initializing, it will read and load the modules from this file.
You can also load and manipulate the modules using tools such as *pactl* and *pacmd*, see *pulse-cli-syntax(5)* manpage for more info.

The *.conf* files are simple key-value formatted files while the *.pa* are real command scripts following the CLI protocol format of PulseAudio.

Example:

```
load-sample-lazy x11-bell
   /usr/share/sounds/freedesktop/stereo/bell.oga
load-module module-x11-bell sample=x11-bell
```

When it comes to realtime scheduling, you can either integrate PulseAudio by giving it priority at the OS level, or you can rely on its integration with RealtimeKit (rtkit), which is a D-Bus service that changes the scheduling policy on the fly.
Realtime policy will be applied to all sink and source threads so that timer-based scheduling have lower latency. This is important if you want to play audio in bit-perfect mode, that is about not applying any resampling or mixing to the audio but playing it directly as is.

Pulseaudio Interesting Modules And Features

Let's now have a look at a couple features and modules. We can't list them all as there are so many but let's try to do a roundup of the most interesting ones.

PulseAudio can talk over many protocols by using different plugins, that includes:

- Native protocols over different transport (fd, unix, tcp)
- mDNS (Zeroconf)
- RTP/SDP/SAP
- RAOP
- HTTP
- DLNA and Chromecast (Digital Living Network Alliance)
- ESound

It also offers control protocols to manage the server itself and audio streams:

- D-Bus API
- CLI protocol

There are many modules for post-processing and effects because it's easy to create a chain of sound. Though only two types of connections are allowed: source output are connected to source, and sink input to sink. That means you'll sometimes need to create indirect adapters to have the scenario you want. If you need more advanced chains you are probably better off going to another sound server that specializes in these like JACK.

PulseEffects is a popular software to add audio effects to stream, but note that it is getting deprecated in favor of PipeWire.
The LADSPA plugin called *module-ladspa-sink* allows to load the audio processing effects in the common format we've seen, and apply them to a sink.

There are a couple different equalizers such as the integrated one and others like prettyeq. An equalizer works by becoming the default output/sink.
There are also noise cancellation filters such as the builtin one *module-echo-cancel* and *NoiseTorch*

Some cool desktop integration features:

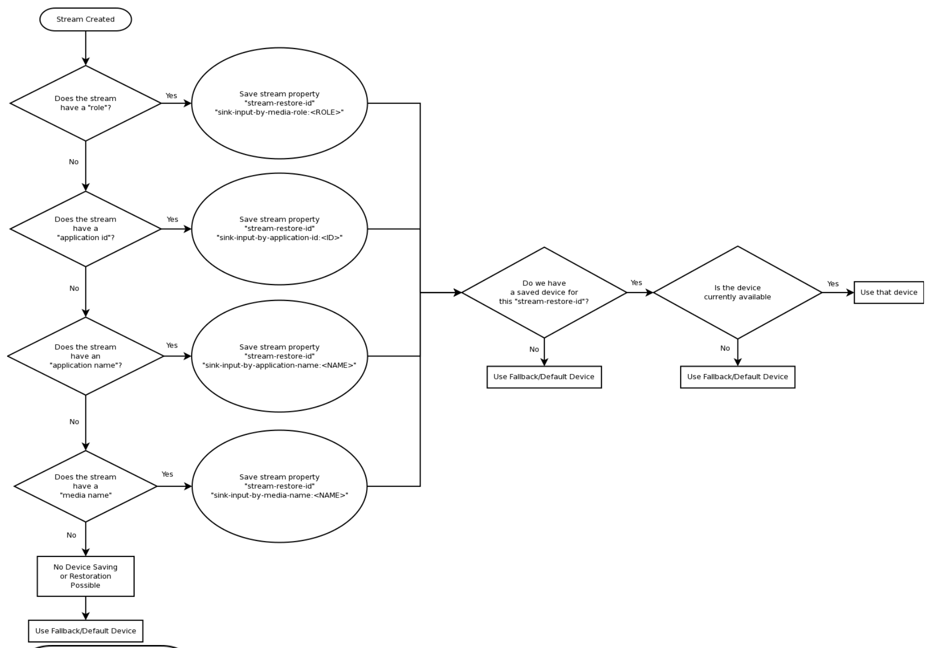
- sample cache, basically loading a small sound sample in the server and cache it (*pacmd list-samples*)
- multimedia buttons
- publish address on X11 root window properties
- x11 bell integration, using XKB bell events and played from sample cache
- Use of GNOME registry to load modules instead of *.pa* configuration files.
- Hotplug based on udev/jackbus/coreaudio/bluetooth/bluez5/bluez4 so that cards are automatically detected.

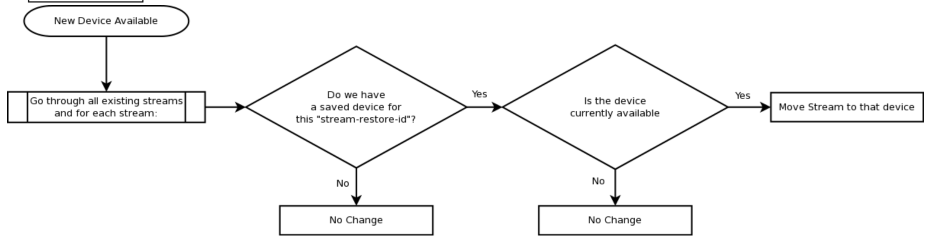
Let's go over two less known but cool features of PulseAudio: The restoration DB and the routing process.

PulseAudio keeps track and restores the parameters used for cards, devices, and streams. When a new object appears the server tries to restore the previous configuration and might move the streams to another device based on what it has seen before.
This neat automatic setup is done via an embedded db, which the user can choose the format. This is done via the following plugins: *module-device-restore*, *module-stream-restore*, and *module-card-restore*. You'll find the files ending in **.tdb* in your *~/.config/pulse/* if you are using the default configuration. (You can use *tdbtool* to inspect them if you're interested, the author has created tools to manipulate these files, https://github.com/venam/pa-resto-edit)

The decision regarding this automatic setup is influence by the media role and other properties associated with the stream and device such as application id and name.
This information is set programmatically on the stream when communicating with the server, for example a stream can be set as video, music, game, event, phone, animation, etc.. (Sort of like the use case scenario)
So based on this information in the restore db, the routing will select which source or sink is best for a new stream.

The actual algorithm, that takes care of this isn't obvious, I advise looking at these two flow charts for more details.


Stream Created
Does the stream have a "role"?
Yes
Save stream property "stream-restore-id" "sink-input-by-media-role:<ROLE>"
No
Does the stream have a "application id"?
Yes
Save stream property "stream-restore-id" "sink-input-by-application-id:<ID>"
No
Does the stream have an "application name"?
Yes
Save stream property "stream-restore-id" "sink-input-by-application-name:<NAME>"
No
Does the stream have a "media name"
Yes
Save stream property "stream-restore-id" "sink-input-by-media-name:<NAME>"
No
No Device Saving or Restoration Possible
Use Fallback/Default Device
Do we have a saved device for this "stream-restore-id"?
Yes
Is the device currently available
Yes
Use that device
No
Use Fallback/Default Device
No
Use Fallback/Default Device


New Device Available
Go through all existing streams and for each stream:
Do we have a saved device for this "stream-restore-id"?
Yes
Is the device currently available
Yes
Move Stream to that device
No
No Change
No
No Change

This can be confusing if you are trying to set a default device because the default device is only used as fallback when the restore db is in place.

Pulseaudio Tools

There are many tools that can be used to interface with PulseAudio, some are full front-ends and some are more specific.

We've seen *pacmd* and *pactl* which both are used to reconfigure the server at runtime.
paplay, *parec*, *pacat*, *pamon*, and others are mini-tools used to test features of PulseAudio.
There are GUIs like *pamixer*, *paprefs* (useful to setup simultaneous output), *pavucontrol*.
There are TUI like *pulsemixer*.

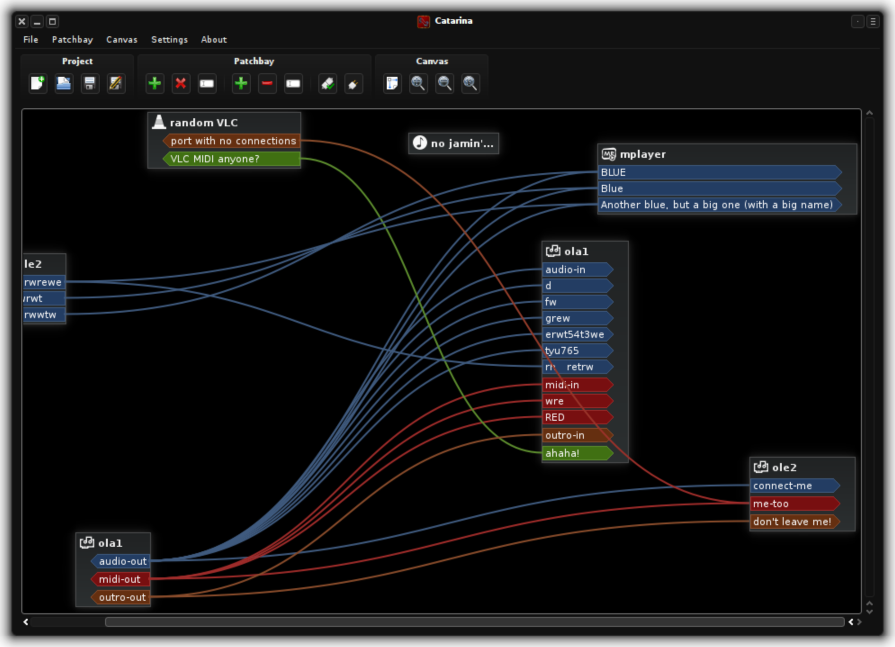
Longer lists can be found here for GUI and here for CLI.

Pulseaudio Suspending

Sometimes it is useful to temporarily suspend PulseAudio. The utility *pasuspender* has this purpose. It is especially useful when running JACK in parallel with PulseAudio.

Another way is to use the D-Bus reservation API to allocate a card to a certain application. This can be done more easily when you include the module for JACK within PulseAudio.

JACK



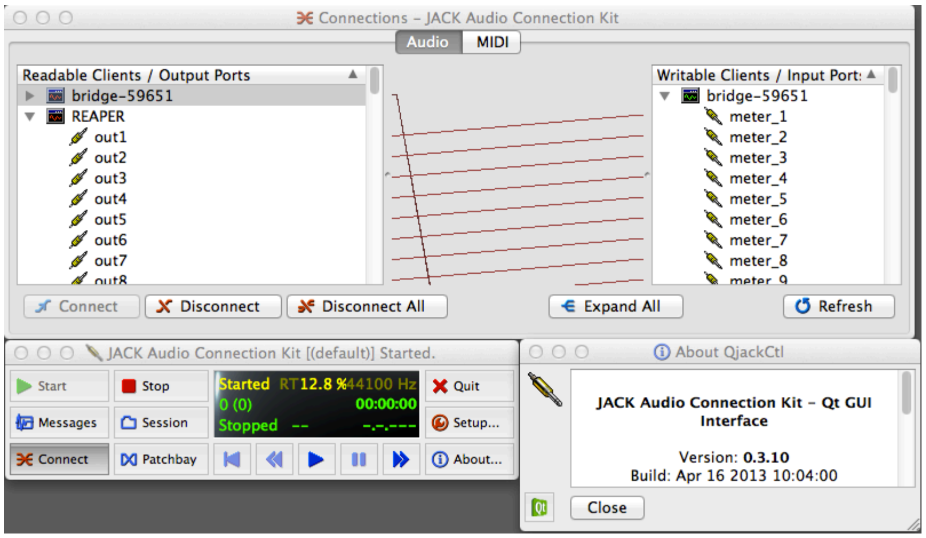
catarina

JACK is a sound server, just like sndio, ESD, PulseAudio, and others, but is designed for professional audio hardware and software. JACK can run on top of many drivers, including on top and along PulseAudio. While PulseAudio is designed for consumer audio for desktop and mobile, JACK is made for music production.

JACK recursive acronym stands for JACK Audio Connection Kit, and as the name implies it specializes in connecting multiple hardware and virtual streams together. As you remember, this was not so simple in PulseAudio. It allows to setup real-time, low-latency connections between streams, and ease the configuration regarding parameters like buffer size, sample rate, fragment size, and others.

The biggest strength of JACK is its integration with professional tooling and its emphasis on MIDI and professional hardware. In a professional environment you often have to deal with multiple devices such as mixers, turntables, microphones, speakers, synthesizer, etc.. The graphical interfaces that come around the JACK server allow to handle this easily but you have to be knowledgeable in the audio world to understand the complex configuration.

JACK also has support for specific professional hardware drivers, like a FireWire driver (IEEE 1394) that PulseAudio doesn't have.



Started RT 12.8 % 44100 Hz
0 (0) 00:00:00
Stopped -- -.-.---

JACK Audio Connection Kit - Qt GUI Interface

Version: **0.3.10**
Build: Apr 16 2013 10:04:00

qjackctl

JACK frontends and software using it are really where it shines, there are so many interfaces and GUIs for professionals. The most widely used software to configure it being qjackctl, a connection manager making links between streams and devices. That's because JACK separates the concerns: one part is about managing the connections, in a graph-like fashion, and the other part is only concerned with passing the audio stream around. This is especially important when there's a lot of equipment and should be easily doable via a GUI.
Let's mention some professional audio engineer software:

- mpk - Virtual MIDI Piano Keyboard
- KX.studio Cadence - A studio with multiple sub-tools like Cadence and Claudia
- Patchage - visual connection manager
- Catia - anoter visual connection manager
- ardour
- Qtractor
- Carla
- QASMixer
- bitwig studio
- drumstick
- QSynth
- Helm
- Calf Studio Gear
- LMMS

Here is a longer list.

There are also a bunch of specialized Linux distributions:

- https://www.ap-linux.com/
- https://www.bandshed.net/avlinux/
- https://linuxmusicians.com/

The audio domain, for sound engineers and musicians, is gigantic and it's not really my place to talk about it, so I'll keep it at that.

PipeWire

PipeWire is a relatively new sound-server but is also much more. It not only handles audio streams but video streams too, it is meant as a generic multimedia processing graph server.
On the audio side, it will try to fill the need of both desktop users, like PulseAudio, and professional audio engineers, like JACK.

Initially the project was supposed to be named PulseVideo and do a similar job to PulseAudio but for Video. The rational to handle all multimedia streams together is that it makes no sense to handle video streams without the audio counter-part to sync them together.

The project was started by GStreamer's creator. The library, as you may remember, already handles audio and video streams.
So in sum, that creates an equation that includes GStreamer + PulseAudio + PulseVideo + JACK-like-graph. The integration with GStreamer means that applications that already use it will automatically be able to interoperate with PipeWire.

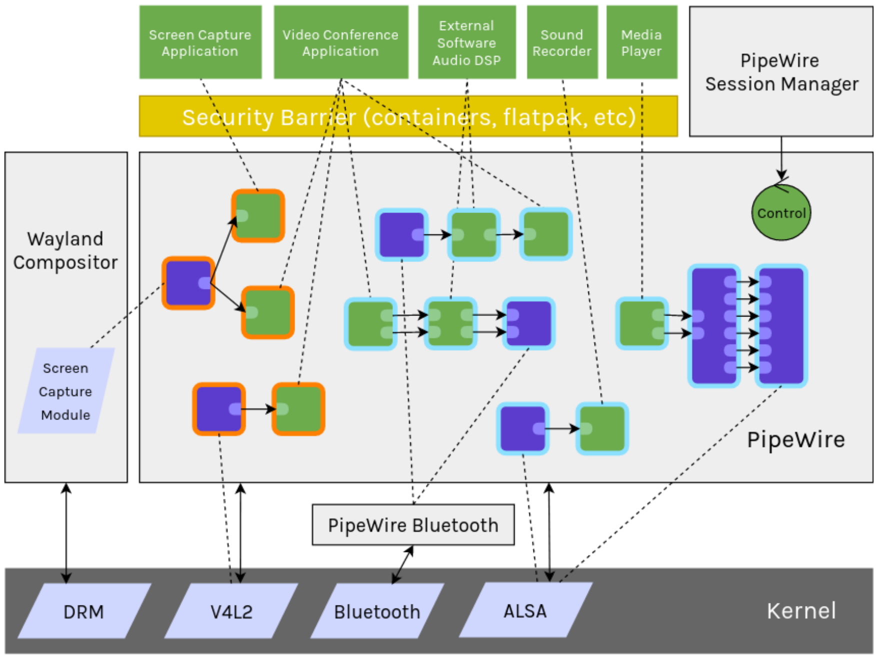
The project is still in early development, not so stable, and the server side currently only supports video and has integration layers with PulseAudio and JACK. Otherwise, it can interfaces with ALSA clients directly, through a new ALSA pcm device that redirects to PipeWire (Like PulseAudio does).

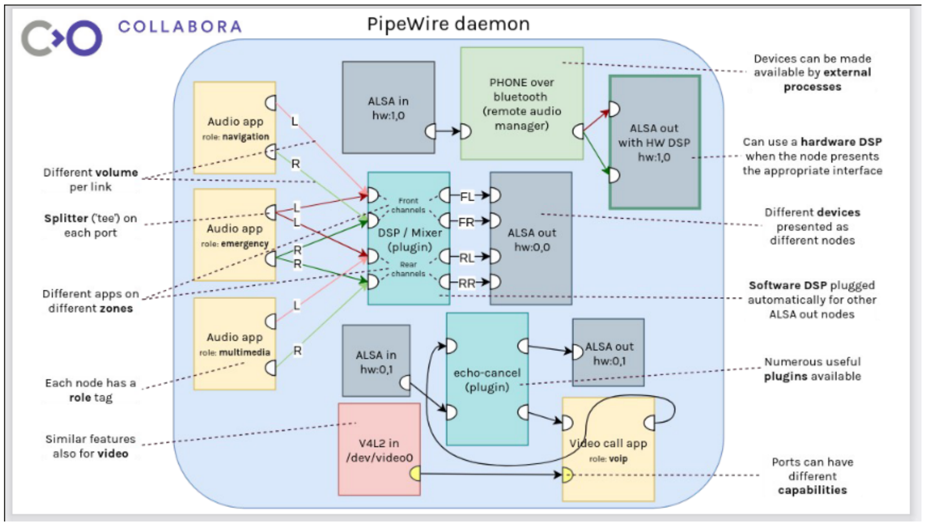
Some of the new goals of PipeWire is that it will give access to media to sandboxed Flatpack applications and allow Wayland compositors to access media streams securely via a mechanism like PolKit for granular access control. This is called a policy/session manager.

PipeWire takes ideas from multiple places. In itself it only cares about creating a graph of nodes that will process and move audio streams around via IPC. Meanwhile, another process, like with JACK, will take care of managing the connections, device discovery, and policies.
So you get a division of roles, a processing media graph and a policy/session manager.

The innovation lies in this graph model taken from JACK, combined with integration of policy management and desktop usage. Each node in the graphs has its own buffer mechanism and dynamic latency handling. That leads to a lower CPU usage because of the timer-based scheduling model that wakes up nodes only when they need to and can dynamically adapt buffers depending on the latency needed.

What's all this talk about nodes and graphs about, what does this actually mean?





COLLABORA
PipeWire daemon
Audio app role: navigation
Audio app role: emergency
Audio app role: multimedia
ALSA in hw:1,0
PHONE over bluetooth (remote audio manager)
ALSA out with HW DSP hw:1,0
DSP / Mixer (plugin)
Front channels
Rear channels
FL
FR
RL
RR
ALSA out hw:0,0
ALSA in hw:0,1
echo-cancel (plugin)
ALSA out hw:0,1
Video call app role: voip
V4L2 in /dev/video0
L
R
Different volume per link
Splitter ('tee') on each port
Different apps on different zones
Each node has a role tag
Similar features also for video
Devices can be made available by external processes
Can use a hardware DSP when the node presents the appropriate interface
Different devices presented as different nodes
Software DSP plugged automatically for other ALSA out nodes
Numerous useful plugins available
Ports can have different capabilities

To understand this we have to get 2 concepts: the **PipeWire media graphs** and the **session management graphs**.

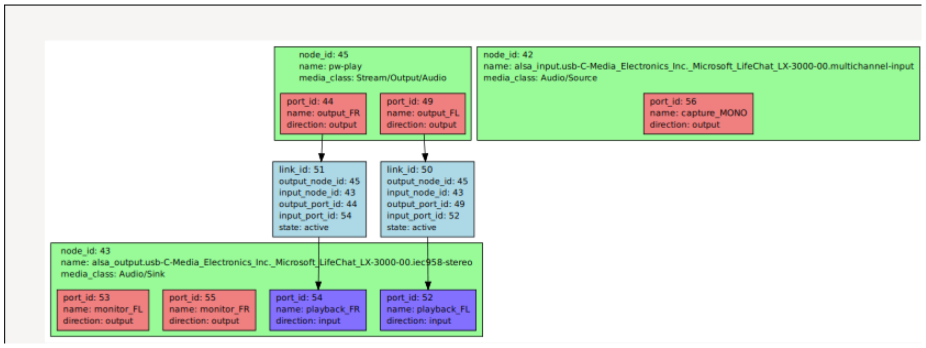
PipeWire is a media stream exchange framework, it embodies this concept through the media graph that is composed of nodes that have ports which are connected through directed links. The media streams flows from node to nodes passing by their ports via their links to reach the port of another node.
A node is anything that can process media, that either consumes it or produces

it. Each node has its own buffer of data and personal preferences and properties such as media class, sample rate, bit rate, format, and latency. These nodes can be anywhere, not limited to inside the PipeWire daemon, but can also be external inside clients. That means the processing of streams can be delegated to other software and passed around from node to node without PipeWire touching the stream. Nodes can be applications, real devices, virtual ones, filters, recording application, echo-cancellation, anything that dabbles with media streams.
To interact with the rest of the system, a node can have ports, which are interfaces for input (sink) or output (source).
A link is a connection between 2 ports, one as the source the other as the sink.

So far that's a very generic concept, nodes that handle media and have ports that can generate media stream or consume it.

Whenever some media needs to be handled by a node, they are woken up by a timer-scheduler mechanism. The connected nodes form a graph, and one of these nodes usually "drives" the graph by starting the processing for all other nodes joined in it. This timer dynamically manages the desired latency for each node to negotiate the most appropriate one within a range.
When two nodes in a graph need to communicate with one another, they have to negotiate a common preferred format and minimum latency based on their buffer size. That's why the nodes are normally wrapped in an adapter that will automatically do the conversion (sample rate, sample format, channel conversion, mixing, volume control). This is all done dynamically which is good for desktop usage, but not so much for pro-audio.
More info about the buffer model here

In practice, that gives a graph that looks like the following, dumped using *pw-dot(1)* utility:



pw-dot

So far so good, we have nodes that exchange streams, a graph processing media, but how do we get these nodes in there in the first place and how do we choose which one links to which one? We need a way to attach these nodes, decide what is connected to what. Like when a client attaches and asks to play an

audio stream, how is that handled?
That's where the second piece of the PipeWire equation comes in: the session connector and policy controller along with its session management graph.

This piece of software is external to PipeWire, it can have multiple implementations. The default one that comes with the installation is called *pipewire-media-session*, another one that is still a work-in-progress is called *wireplumber*. There are talks about including it in desktop environment session managers such as *gnome-session-daemon*.

The role of this software is to keep track of the devices available, their priorities, keeping track of which application uses which device, ensuring policy control and security, keep a device and stream restoration database (not implemented), share global properties of the system, find the default endpoints used by clients when asked to play sound, etc..

When clients connect to PipeWire they announce their session information, what they'd like to do (playback or capture), the type of media they want to handle (video, playback, VOIP, etc..), their preferred latency and sample rate, etc..
Based on this information, the session manager can shortlist which device the client needs.

As soon as the client connects, if its session information is new (PID,GID,UID), it will first be frozen until its permissions are acknowledged. PipeWire default session manager *pipewire-media-session* comes with a series of modules that take care of this called *module-portal* and *module-access*. The portal module will, through desktop integration (via D-Bus, like PolKit), open a pop-up to ask for user confirmation (read,write,execute). After that, the session manager will configure the client permissions in its client object.
So clients are not able to list other nodes or connect to them until the session manager approves.

The session manager can then choose to restore connections based on previous usage of this stream — decide how to connect it, make sure it's linked to the appropriate device and follows the peering rules for its use case (depending on media type). Then this new node can get connected and configured in the media graph.

That is as far as clients are concerned, however, PipeWire doesn't open any device by default either, and it is also the role of the session manager to load devices, configure them, and map them on the media graph.

To achieve this flexibly, some session managers can use what is called a session management graph. In practice, this is the equivalent to how PulseAudio manages devices through the concept of cards and profiles that can create sink and source nodes but with the extra addition of routing based on use case. Internally, the session manager actually reuses the related PulseAudio code and config for device management.

The session management graph is a representation of this, the high-level media flow from the point of view of the session manager. As far as I can see, these graphs are hosted within PipeWire along other objects but they have different types.

```
+-----------------+
|                 |
|        +---------------+
|Endpoint|Endpoint Stream|------.
|        +---------------+      |
|                 |             |
+-----------------+             |
                                |
   .Endpoint Link ._____________|
   |
   |        +-----------------+
   |        |                 |
   | +---------------+        |
   `-|Endpoint Stream|Endpoint|
     +---------------+        |
            |                 |
            +-----------------+
```

Endpoints are where media can be routed to or from (laptop speaker, USB webcam, Bluetooth headset mic, amplifier, radio, etc..). They then get mapped to nodes in the media graph, but not always.
They can be mutually exclusive, this is the equivalent of device ports, which as you remember correspond to a single input or output associated with the card. So the Endpoint is a card+a device port in theory.

The Endpoint Stream are the logical path, the routing associated with a use case (Music, Voice, Emergency, Media, etc..). They are equivalent to PulseAudio sink/source on the device side, and sink-input/source-output on the client side. These can be used to change the routing in the media graph.

The Endpoint Link is what connects and creates the media flow, it can only exist if there are actual links in the media graph or if the link exists physically (real hardware connection).

The session manager is then responsible of knowing which devices are present, what they support, what kind of linking information is there, and if streams need compatibility between them, and share that information if needed.

Additionally, internally the session manager can put on the graph objects of type *Device* which map to the ALSA cards, JACK clients, or others. Like cards in PulseAudio.

Now let's see how to configure PipeWire and its session/policy manager.

When the PipeWire daemon starts it reads the config file located at

$PIPEWIRE_CONFIG_FILE, normally */etc/pipewire/pipewire.conf*. It contains sections, some to set server values, some to load plugins and modules, some to create objects, and some to automatically launch programs.
The goal of this configuration file is to make it easy to configure how the processing happens in the media graph.

The execution section is normally used to automatically launch the session manager.

There are configurations related to how the graph will be scheduled such as the global sample rate used by the processing pipeline, which all signal will be converted to: *default.clock.rate*. The resampling quality can be configured server side too (even though the node are wrapped in an adapter that does that) in case it needs to be done. This resampler is a custom highly optimized one. Moreover, you can control the buffer size minimum and maximum value through a *min-quantum*, *max-quantum*, and default quantum, which are going to be used to dynamically change the latency.

```
default.clock.quantum =     1024
default.clock.min-quantum = 32
default.clock.max-quantum = 8192
```

NB: PipeWire relies on plugins that follow the SPA, Simple Plugin API, based on GStreamer plugin mechanism but lighter.
Most of them can be found in */usr/lib/spa-*.

Now as far as the session manager is concerned, it highly depends on the implementation. It is about modules for policy and matching rules to associate them with specific actions to do in the media graph or the session management graph. Monitor subsystems watch when a new device or stream appear (new node), or when the system creates a new object, and decides what to do with it based on the endpoint configuration.

For example, I have the following rule for WirePlumber in *00-default-output-audio.endpoint-link*:

```
[match-endpoint]
media_class = "Stream/Output/Audio"

[target-endpoint]
media_class = "Audio/Sink"
```

Which will attach a new endpoint of class "Stream/Output/Audio" to the default endpoint with class "Audio/Sink".

However, this all depends on the session manager implementation.

At this point it's easy to picture that this system would be fantastic to create filters and effects streams, however currently this is still very hard to do. So far, the only way to achieve this is with the help of PulseAudio tools such as *pactl*.

You can create sinks and sources with specific media classes so that they map within PipeWire.
For example:

```
pactl load-module module-null-sink object.linger=1
   media.class=Audio/Sink sink_name=my-sink
   channel_map=surround-51
```

pw-cli can also be used instead:

```
pw-cli create-node adapter {
   factory.name=support.null-audio-sink node.name=my-mic
   media.class=Audio/Duplex object.linger=1
   audio.position=FL,FR }
```

It remains that PipeWire is missing the interface toolset to easily interact with it. There aren't any good sound configuration tool that permits to inspect and manipulate it so far. Moreover, the ones that will have to do this will need to be able to portray the internal connection mechanism, similar to JACK's many connection managers.

I quote:

> There is currently no native graphical tool to inspect the PipeWire graph but we recommend to use one of the excellent JACK tools, such as Carla, catia, qjackctl, … You will not be able to see all features like the video ports but it is a good start.

However, someone is working on a new one called helvum: https://gitlab.freedesktop.org/ryuukyu/helvum

PipeWire comes with a set of mini debug tools similar to what PulseAudio provides, they start with the `pw-*` prefix:

- `pw-cli` - The PipeWire Command Line Interface
- `pw-dump` - Dump objects inside PipeWire
- `pw-dot` - The PipeWire dot graph dump in graphviz format
- `pw-mon` - The PipeWire monitor
- `pw-cat` - Play an Record media with PipeWire
- `pw-play` - Like `pw-cat` but for play only
- `pw-metadata` - The PipeWire metadata
- `pw-profiler` - The PipeWire profiler
- `pw-top` - Acts like `top` but for devices nodes inside PipeWire

The most useful tools are *pw-cli*, *pw-dump* and *pw-dot*.

Try:

```
pw-cli info 0
```

Here’s an extract from *pw-dump* showing an Endpoint of class “Audio/Source”, a microphone on the boring headset you’ve encountered in this article.

```
{
  "id": 53,
  "type": "PipeWire:Interface:Endpoint",
  "version": 0,
  "permissions": [ "r", "w", "x", "m" ],
  "props": {
    "endpoint.name":
        "alsa_card.usb-C-Media_Electronics_Inc._Micro..",
    "media.class": "Audio/Source",
    "session.id": 75,
    "client.id": 39,
    "factory.id": 25
  }
},
```

Overall, PipeWire is an interesting sound server, combining a media processing graphs framework along with an external policy/session/connection manager that controls it. The timer and dynamic latency mechanism should have a significant effect on CPU usage.

Unfortunately, after testing it you can clearly see that it is still in its early stage but that it integrates well on the audio part through the backward compatibility with PulseAudio.
Additionally, it remains to be seen if the tooling around it will adapt properly to the graph thinking. Will they build around the concept or dismiss it entirely considering most desktop tools today aren’t used to long sequence of media processing, and neither are users.
Finally, on the session/connection manager side we need more innovation. What is currently available seems to be lacking. I couldn’t find much documentation about the restoration DB mechanism, hot-plug, desktop integration, caching of sample sounds for events, and others.

Conclusion

Anybody who claims one system offers better audio "quality" is just plain wrong and base their assumption on something non-scientifically proven.

All the low-level stacks are relatively the same speed when running in bit-perfect mode. The big differences between all that we've seen relates to the driver support, the ease of use, the desktop integration, and the buffer/latency management.
Some systems are targeted at end users and others at audio engineers.

According to measurements from A Look at Linux Audio (ALSA, PulseAudio) for instance, ALSA performs very well on Linux and keeps up with a Windows machine that is much more performant. Tests with PulseAudio are similar but use 6% more CPU processing.

> Whether in the past with Mac OS X, or Windows, and now Linux, there is no evidence that operating systems make any difference to sound quality if you're playing "bit perfect" to the hardware directly (ie. ALSA to DAC with no software conversion similar to Windows ASIO, Kernel Streaming or WASAPI)

The discussion then rotates around low-latency using real-time scheduling, better IO, using better sampling size, etc.. (See *Analog to Digital & Digital to Analog (ADC & DAC)* for more info on what these mean).

The audio stack is fragmented on all operating systems because the problem is a large one. For example, on Windows the audio APIs being ASIO, DirectSound and WASAPI.
Perhaps MacOs has the cleanest audio stack, CoreAudio, but nobody can clearly say if they can't look at the code. PulseAudio was inspired by it.
The stack of commercial operating systems are not actually better or simpler.

Meanwhile, the BSD stack is definitely the simplest, even though there are discrepancies between the lowest layers and lack of driver support, sndio makes it a breeze.

Linux is the platform of choice for audio and acoustic research and was chosen by the CCRMA (Center for Computer Research in Music and Acoustics).

Let's conclude, we've seen basic concepts about audio such as the typical hardware component, how audio is transferred and converted from the real world to the digital world through digital-to-analog-converters. We've seen the issue about buffering and fragment size. Then we've taken a look at different libraries that can act as translation layers, as processing helper, or as standard format to write filters. After that we went through the drivers: ALSA and OSS, the crazy configuration format and plugins of ALSA and the internal concepts it has to map devices. On the OSS and SADA side, we've seen the historical fracture and how things could be done mainly hidden away inside the kernel via tunable to not freak out the users. Finally, we've attacked sound servers, from sndio, to the deprecated aRts and ESD, to PulseAudio, to JACK, and lastly PipeWire. Each of them has its specialty, sndio being extremely simple, PulseAudio being great for the desktop integration use case, JACK catering to the audio engineers having too much equipment to connect together and having superb integration with professional tools, and PipeWire that is getting inspired by JACK's graphs but wants to take it a step further by including video streams, integrating with the desktop and making things more snappy with wake-up node driving the processing graph.

Bibliography

Morelo, David. “Noob’s Guide to Linux Audio: ALSA, OSS, and Pulse Audio Explained.” *Linux Hint*, 1 Jan. 1969, https://linuxhint.com/guide_linux_audio/.

Willis, Nathan. “Why You Should Care about PulseAudio (and How to Start Doing It).” *Linux.com*, 3 Nov. 2007, https://www.linux.com/news/why-you-should-care-about-pulseaudio-and-how-start-doing-it/.

Turcotte, Mike. “GNU/Linux for Beginners: How Audio Works - GHacks Tech News.” *GHacks Technology News*, Publisher Ghacks Technology NewsLogo, 16 Aug. 2017, https://www.ghacks.net/2017/08/16/linux-audio-explained/.

Crocoduckoducks. “The Linux Audio Anatomy.” *The Crocoduck’s Pond*, 17 Feb. 2018, https://thecrocoduckspond.wordpress.com/2016/11/19/the-linux-audio-anatomy/.

“FreeBSD Handbook - Sound Setup.” *The FreeBSD Project*, https://www.freebsd.org/doc/handbook/sound-setup.html.

“FreeBSD Manual Pages - Sound(4).” *Sound(4)*, https://www.freebsd.org/cgi/man.cgi?query=sound&sektion=4&manpath=FreeBSD%2B7.2-RELEASE.

Gray, Niklas. “Writing a Low-Level Sound System - You Can Do It! · Our Machinery.” *Our Machinery*, Our Machinery, 17 Mar. 2020, https://ourmachinery.com/post/writing-a-low-level-sound-system/.

OpenAL, https://www.openal.org/.

Gustafsson, Dennis. “Low Level Audio.” *Tuxedolabs Blog*, 25 June 2013, https://blog.tuxedolabs.com/2013/06/26/low-level-audio.html.

“Videos/Digital Show and Tell.” *Videos/Digital Show and Tell - XiphWiki*, https://wiki.xiph.org/Videos/Digital_Show_and_Tell.

Burk, Phil. “PortAudio - an Open-Source Cross-Platform Audio API.” *PortAudio Portable Cross-Platform Audio I/O API*, https://portaudio.com/.

GStreamer, https://gstreamer.freedesktop.org/.

“GStreamer.” *Wikipedia*, Wikimedia Foundation, 26 Nov. 2020, https://en.wikipedia.org/wiki/GStreamer.

“Libcanberra.” Libcanberra - *ArchWiki*, https://wiki.archlinux.org/index.php/Libcanberra.

Linux Audio Developer’s Simple Plugin API (LADSPA), https://www.ladspa.org/.

Open Sound System, https://www.opensound.com/oss.html.

OpenBSD FAQ: Multimedia, https://www.openbsd.org/faq/faq13.html#default.

“Main Page.” *Open Sound System*, http://ossnext.trueinstruments.com/wiki/index.php/Main_Page.

“Open Sound System OSS 4.x Programmer’s Guide.” *OSS v4.x API Reference - Developing Applications for Open Sound System Version 4.1*, http://manuals.opensound.com/developer/.

“Open Sound System.” *Open Sound System - ArchWiki*, https://wiki.archlinux.org/index.php/Open_Sound_System.

OSS core configuration file, https://fossies.org/linux/misc/legacy/oss-linux-v4.2-2019-amd64.tar.bz2/usr/lib/oss/conf.tmpl/osscore.conf

Canonical. Ubuntu Manpage: Osscore - Open Sound Sytem Core Audio Framework., https://manpages.ubuntu.com/manpages/bionic/man7/osscore.7.html.

Canonical. Ubuntu Manpage: oss_usb - USB Audio/MIDI/Mixer Driver, https://manpages.ubuntu.com/manpages/bionic/man7/oss_usb.7.html.

"Open Sound System for NetBSD." *OpenSoundSystem*, https://wiki.netbsd.org/opensoundsystem/.

"Exploring Audio in OpenBSD." *MrBool*, https://mrbool.com/exploring-audio-in-openbsd/29890.

"Udev." *Alsa Opensrc Org - Independent ALSA and Linux Audio Support Site*, https://alsa.opensrc.org/Udev.

"Cat -v Harmful Stuff." *ALSA*, https://harmful.cat-v.org/software/operating-systems/linux/alsa.

"ALSA, Exposed!" *Rendaw*, https://rendaw.gitlab.io/blog/2125f09a85f2.html#alsa-exposed.

"Advanced Linux Sound Architecture." *Wikipedia*, Wikimedia Foundation, 31 Mar. 2021, https://en.wikipedia.org/wiki/Advanced_Linux_Sound_Architecture.

"ALSA." *AlsaProject*, https://alsa-project.org/wiki/Main_Page.

A Close Look at ALSA, https://www.volkerschatz.com/noise/alsa.html.

"Slave Definition." *ALSA Project - the C Library Reference: PCM (Digital Audio) Plugins*, https://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html.

notes.for.sabi.co.UK, notes {at}. *Linux ALSA Sound Notes*, https://www.sabi.co.uk/Notes/linuxSoundALSA.html.

Archimago. MEASUREMENTS: *A Look at Linux Audio (ALSA, PulseAudio)*, http://archimago.blogspot.com/2015/10/measurements-look-at-linux-audio-alsa.html.

"alsa_rnnoise." *~Arsen/alsa_rnnoise - Sourcehut Git*, https://git.sr.ht/~arsen/alsa_rnnoise/.

Sound - FreeBSD Wiki, https://wiki.freebsd.org/Sound.

"Sndio." Sndio - Void Linux Handbook, https://docs.voidlinux.org/config/media/sndio.html.

"Sndio Home." *Sndio*, https://sndio.org/.

"OpenBSD Manual Page Server." *Sndio(7) - OpenBSD Manual Pages*, https://man.openbsd.org/sndio.

"Sndio." *Wikipedia*, Wikimedia Foundation, 13 Jan. 2019, https://en.wikipedia.org/wiki/Sndio.

"OpenBSD Manual Page Server." *Sndiod(8) - OpenBSD Manual Pages*, https://man.openbsd.org/sndiod.8.

Ratchov, Alexandre. *Sndio– OpenBSD Audio & MIDI Framework Formusic and Desktop Applications*. AsiaBSDCon 2010, 13 Mar. 2010, https://www.openbsd.org/papers/asiabsdcon2010_sndio_slides.pdf.

"OpenBSD Manual Page Server." *Audio(4) - OpenBSD Manual Pages*, https://man.openbsd.org/audio.4.

"OpenBSD Manual Page Server." *Audioctl(8) - OpenBSD Manual Pages*, https://man.openbsd.org/audioctl.8.

Duncaen. "Duncaen/Alsa-Sndio." *GitHub*, https://github.com/Duncaen/alsa-sndio.

Rudd-O. “How PulseAudio Works.” *Rudd*, 26 June 2013, https://rudd-o.com/linux-and-free-software/how-pulseaudio-works.

“How Does PulseAudio Start?” *Unix & Linux Stack Exchange*, 1 Feb. 1964, https://unix.stackexchange.com/questions/204522/how-does-pulseaudio-start.

“Welcome to PulseAudio!” *PulseAudio*, https://www.freedesktop.org/wiki/Software/PulseAudio/.

“PulseAudio.” *PulseAudio - ArchWiki*, https://wiki.archlinux.org/index.php/PulseAudio.

“PulseAudio Modules.” *Modules – PulseAudio*, https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/Modules/.

“PulseAudio.” *PulseAudio - Debian Wiki*, https://wiki.debian.org/PulseAudio.

Pulseaudio-Equalizer-Ladspa. “Pulseaudio-Equalizer-Ladspa/Equalizer.” *GitHub*, https://github.com/pulseaudio-equalizer-ladspa/equalizer.

Gaydov, Victor. “PulseAudio under the Hood.” *Victor Gaydov*, https://gavv.github.io/articles/pulseaudio-under-the-hood/.

“This Is the Route to Hell.” *Colin.Guthr.ie*, https://colin.guthr.ie/2010/02/this-is-the-route-to-hell/.

Person. “Setting up Virtual Surround Sound for Headphones.” *EndeavourOS*, 13 Aug. 2020, https://forum.endeavouros.com/t/howto-setting-up-virtual-surround-sound-for-headphones/6889.

“How Use PulseAudio and JACK?: JACK Audio Connection Kit.” *How Use PulseAudio and JACK? | JACK Audio Connection Kit*, https://jackaudio.org/faq/pulseaudio_and_jack.html.

CCB, spol. s r.o. “QJackCtl: JACK Rack.” *QJackCtl: JACK Rack - Linux E X P R E S*, https://www.linuxexpres.cz/praxe/qjackctl-jack-rack.

Kučera, František. “Generating and Sending MIDI Messages.” *Generating and Sending MIDI Messages – Relational Pipes*, https://relational-pipes.globalcode.info/v_0/examples-jack-midi-generating-1.xhtml.

Yusof, Khairil. “Audio and MIDI Controller on Ubuntu Linux.” Medium, *Medium*, 25 May 2020, https://medium.com/@kaerumy/audio-and-midi-controller-on-ubuntu-linux-1058e00bc7d0.

“Record, Edit, and Mix on Linux, MacOS and Windows.” *Ardour*, https://ardour.org/.

“Demystifying JACK – A Beginners Guide to Getting Started with JACK.” *Demystifying JACK – A Beginners Guide to Getting Started with JACK | Libre Music Production*, https://linuxaudio.github.io/libremusicproduction/html/articles/demystifying-jack-%E2%80%93-beginners-guide-getting-started-jack.

“Multimedia Processing.” *PipeWire*, https://pipewire.org/.

“PipeWire.” *Wikipedia*, Wikimedia Foundation, 11 May 2021, https://en.wikipedia.org/wiki/PipeWire.

“PipeWire.” *PipeWire - ArchWiki*, https://wiki.archlinux.org/index.php/PipeWire.

“Wiki · PipeWire / Pipewire.” *GitLab*, https://gitlab.freedesktop.org/pipewire/pipewire/-/wikis/Configuration.

“Wiki · PipeWire / Object Design.” *GitLab*, https://gitlab.freedesktop.org/pipewire/pipewire/-/blob/master/doc/objects_design.md

“PipeWire Late Summer Update 2020.” *Christian FK Schaller*, https://blogs.gnome.org/uraeus/2020/09/04/pipewire-late-summer-update-2020/.

“WirePlumber Configuration.” *Configuration*, https://pipewire.pages.freedesktop.org/wireplumber/daemon/configuration.html?gi-language=c.

“WirePlumber, the PipeWire Session Manager.” *Collabora*, https://www.collabora.com/news-and-blog/blog/2020/05/07/wireplumber-the-pipewire-session-manager/.

Taymans, Wim. *Simple Plugin API*, Redhat, 10 Oct. 2016, https://gstreamer.freedesktop.org/data/events/gstreamer-conference/2016/Wim%20Taymans%20-%20Simple%20Plugin%20API%20(SPA).pdf.

Kiagiadakis, George. *The PipeWire Multimedia Framework and Its Potential in AGL*, Collabora, https://wiki.automotivelinux.org/_media/pipewire_agl_20181206.pdf.

“New Graphing Tool for PipeWire Debugging.” *Collabora*, https://www.collabora.com/news-and-blog/blog/2019/12/09/new-graphing-tool-pipewire-debugging/.

Taymans, Wim. *PIPEWIRE: A LOW-LEVEL MULTIMEDIA SUBSYSTEM*, Red Hat, Spain, 25 Nov. 2020, https://lac2020.sciencesconf.org/307881/document.

“PipeWire: The Linux Audio/Video Bus.” *LWN.net*, https://lwn.net/SubscriberLink/847412/d7826b1353e33734/.

“PipeWire.” *PipeWire - Gentoo Wiki*, https://wiki.gentoo.org/wiki/PipeWire.

“Audio Latency Demystified, Part 1/4.” *MINDMusicLabs.com*, 3 Apr. 2019, https://www.mindmusiclabs.com/audio-latency-demystified-part-1/.

“Windows Audio APIs.” *Windows Audio APIs - Official Kodi Wiki*, https://kodi.wiki/view/Windows_audio_APIs.

“Center for Computer Research in Music and Acoustics.” *CCRMA*, https://ccrma.stanford.edu/.

Wang, Yonghao, Ryan Stables, and Joshua Reiss. “Audio latency measurement for desktop operating systems with onboard soundcards.” *Audio Engineering Society Convention 128*. Audio Engineering Society, 2010.

Wright, Matthew, Ryan J. Cassidy, and Michael Zbyszynski. “Audio and gesture latency measurements on linux and osx.” *ICMC*. 2004.