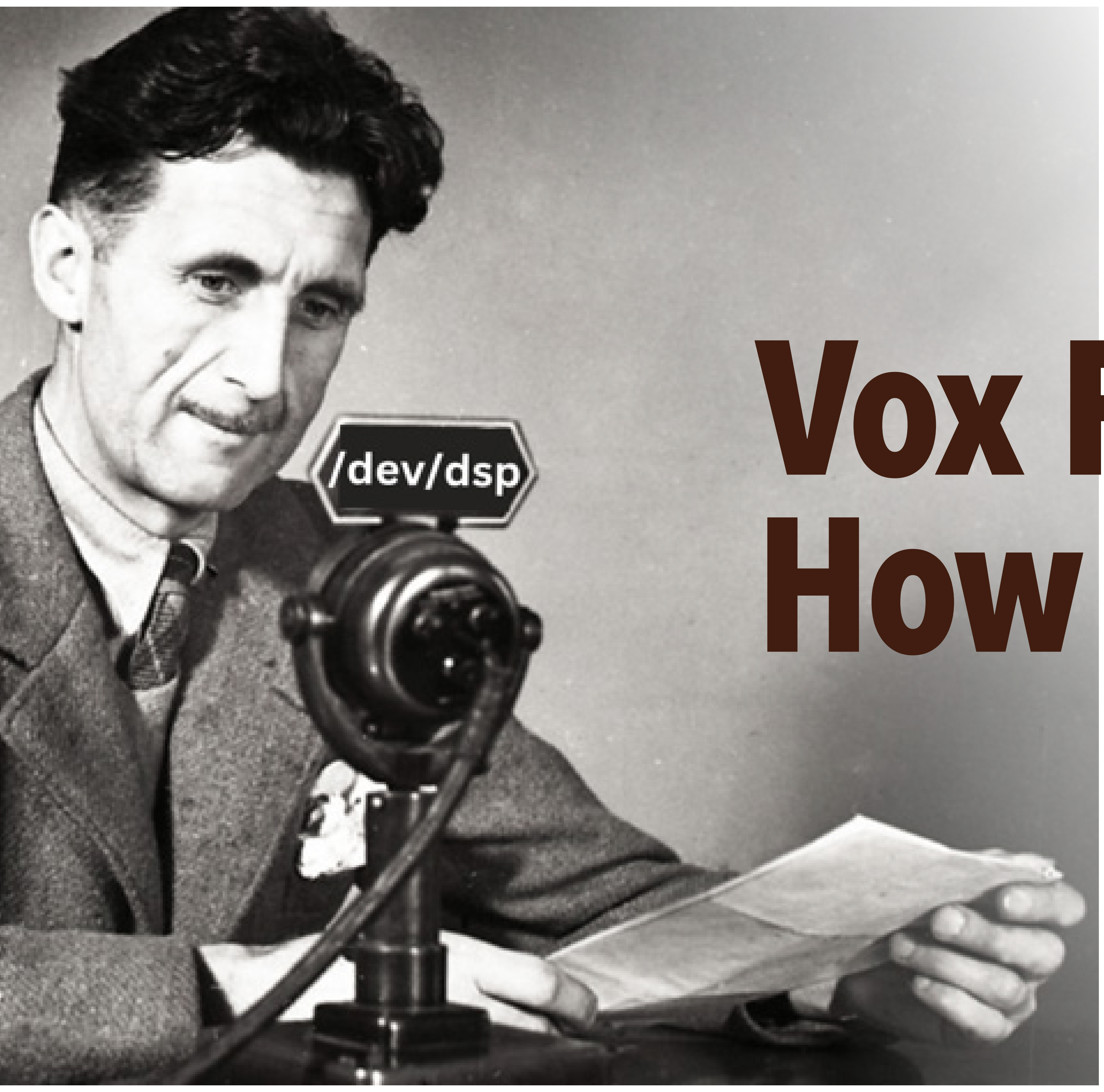

Vox FreeBSD: How Sound Works

BY CHRISTOS MARGIOLIS

Sound support for FreeBSD began in 1993, when Jordan K. Hubbard imported the generic Linux sound driver into FreeBSD, later known as the VoxWare sound drivers, written by Hannu Savolainen. Several new versions of the VoxWare drivers were imported (and modified) into the FreeBSD base system between 1993 and 1997. At this time, Amancio Hasty and Jim Lowe did most of the work on sound support in FreeBSD. The VoxWare drivers eventually became what we know today as OSS, the Open Sound System, maintained by Hannu and his team at 4Front Technologies.

However, things changed in 1997 as more of the foundation for the sound system we have today was laid out by Luigi Rizzo. In 1999, Cameron Grant rewrote the sound system for FreeBSD 4.0, now using the newbus interface, which supported a great deal of hardware. This superseded Luigi's driver, which was removed from the tree a month later. In the following years, many things changed in the sound area. The number of drivers for different hardware chips increased drastically (especially for PCI and USB devices), and many improvements were made to the sound infrastructure, thanks to Cameron Grant and Orion Hodson. Cameron passed away in 2005, a major loss for the FreeBSD community.

Ariff Abdullah took over the maintainership of the sound code in FreeBSD in 2005. Since then, we've seen some dramatic changes in sound support, as well as several rounds of device driver restructuring.

Ariff Abdullah took over the maintainership of the sound code in FreeBSD in 2005. Since then, we've seen some dramatic changes in sound support, as well as several rounds of device driver restructuring.

OSS

FreeBSD's sound API is called OSS, which stands for "Open Sound System". Interestingly enough, it also used to be Linux's default sound API, until it was eventually replaced by ALSA. FreeBSD, however, still uses it, and it provides a simple and clean API by just exposing standard device files (`/dev/dsp*` corresponding to sound cards and `/dev/mixer*` to mixers), which are operated on using a few common POSIX system calls:

Syscall	Description
`open(2)`	Open device.
`close(2)`	Close device.
`read(2)`	Record audio.
`write(2)`	Play audio.
`ioctl(2)`	Query/manipulate settings (sample rate, format, volume, and much more).
`select(2)`, `poll(2)`, `kqueue(2)`	Poll for events. `kqueue(2)` is FreeBSD-only (15.0 onwards).
`mmap(2)`	Memory-mapped I/O.

The official manual can be found here: http://manuals.opensound.com/developer/

Since interaction with the sound device is done using common syscalls, it is possible to do things like this in the terminal:

1. Play white noise: `cat /dev/random > /dev/dsp`
2. Record raw PCM (Pulse Code Modulation) stream into a file: `cat /dev/dsp > foo`
3. Very crude input monitoring, which can be used to make sure your microphone is working: `cat /dev/dsp > /dev/dsp`

For more elaborate things, however, actual programs have to be written — Fortunately, the OSS API is straightforward to use. `/usr/share/examples/sound` contains several example programs and is, in fact, growing as of the writing of this article.

In addition to the `/dev/dsp*` devices, there is a `/dev/dsp` device, which is what was used in the examples above. This isn't a real device, but an alias to the currently default device. Applications that want to use whatever the default device is at any given time are strongly encouraged to access this device instead of hardcoding specific ones.

An additional nice feature is that OSS works with POSIX syscalls only is that it is trivial to use OSS in any programming language, without the need for language bindings. The only thing one would need to do to fully "port" OSS to another language would be to define some global OSS API constants and structures, which can be found in `/usr/include/sys/soundcard.h`.

sound(4)

The OSS API is implemented inside `sound(4)`. It abstracts some generic functionality into a kernel module, so that individual device drivers (e.g., `snd_hda(4)`, `snd_uaudio(4)`, etc.) do not need to duplicate code that would be the same across all of them. This generic functionality includes:

- Device creation, deletion, and access.
- Buffer management.
- Audio processing.
- Providing global and (most of) device-specific sysctls.

Device drivers have to attach to `sound(4)` and set up a communication pipeline with it, during their initialization stage. In other words, `sound(4)` is the bridge between userland

and device drivers. This is particularly convenient, because the FreeBSD kernel exposes the same `/dev/dsp*` files and sysctls (`hw.snd.*` and `dev.pcm.*`) for every connected sound device, and provides uniform access and configuration to users and application programmers, while also avoiding the need for duplication at the device driver level.

`sound(4)` also exposes `/dev/sndstat`, which provides information about all connected sound devices and active channels, and is used internally by `virtual_oss(8)` and `sndctl(8)`.

`sound(4)` works with PCM audio streams, the specifics of which (sample rate, sample format, etc.) can also be manually configured by the user with `sndctl(8)`, if needed. This means that applications working with audio represented in other formats, such as WAV, Opus, MP3, etc., need to convert the stream to PCM during playback, and from PCM during recording.

High-level implementation overview

Device access

As was mentioned earlier and shown in the example program, applications access the sound subsystem through `/dev/dsp*`, using the `open(2)` syscall. The following flags can be specified:

`open(2)` flag	Description
`O_RDONLY`	Recording.
`O_WRONLY`	Playback.
`O_RDWR`	Recording and playback.

The `O_NONBLOCK` and `O_EXCL` flags can be additionally specified in the `open(2)` call, for non-blocking I/O and exclusive access to the device respectively. For `sound(4)` to know which channels belong to which file descriptor, and to route audio and information properly, it uses `DEVFS_CDEVPRIV(9)`.

As alluded to earlier, there are also `/dev/mixer*` devices. This is a legacy interface, mainly used for volume setting and recording source selection, by applications such as `mixer(8)`. `/dev/mixer*` devices have a 1:1 relationship with `/dev/dsp*` ones, so for instance, `/dev/mixer0` is the mixer device corresponding to `/dev/dsp0`. This interface also provides some functionality for physical mixers. As of OSS version 4.0, the mixer API has been rewritten, but is not currently fully implemented on FreeBSD.

Channels

Channels hold some important information about their state (e.g., configuration, PID, and name of process consuming it, diagnostic values, etc.), as well as the most important component; the buffer(s). In other words, the actual audio stream. Channels also have their own volume, in addition to the device-wide master volume. This is especially useful because applications get their own volume knobs and usually do not need to touch the master one.

At this point, it is essential to note that there are two types of channels in `sound(4)`; primary/"hardware" and virtual.

The device drivers allocate primary channels, and there is *usually* one for playback and one for recording, depending on what is supported by the hardware. However, some drivers might make the number of primary channels equal to the number of physical playback and recording ports provided by the hardware. Each primary channel comes with a pair of buffers, a software-facing one and a hardware-facing one. The software-facing buffer is responsible for exchanging audio data with userland, while the hardware-facing one is for exchanging data with the hardware. Whenever the device driver is ready to read from or write data to the hardware, it interrupts **`sound(4)`**, and data is copied from one buffer to the other. During playback, since we are writing data *to* the hardware, we copy the software-facing buffer to the hardware one, so that the driver can feed that data to the hardware. The reverse is true for recording.

Virtual channels, commonly referred to as VCHANs, are treated as children of primary channels, but, unlike primary channels, they do not have any connection to the hardware, so only their software-facing buffer is used. The reason for virtual channels existing in the first place is that we want an indefinite number of applications to be able to access the device simultaneously. Without virtual channels, the number of processes that can access the device simultaneously is equal to the number of primary channels, since each channel has only one software-facing buffer, which means that all processes would have to share the same buffer, which is not really ideal for audio, so each buffer has to be dedicated to one process.

The reason for virtual channels existing in the first place is that we want an indefinite number of applications to be able to access the device simultaneously.

Earlier, we explained how audio streams are exchanged between userland and hardware using primary channels. When virtual channels are enabled (as is the case by default), **`sound(4)`**, instead of simply copying the primary channel's software-facing buffer to the hardware-facing one (during playback), and vice-versa (during recording), it first has to mix all the audio streams of the primary channel's children virtual channels, and then supply the final mixed stream to the primary channel's hardware-facing buffer. As you can imagine, this additional layer introduces a slight overhead, which is irrelevant for most use cases, but might not be ideal in some low-latency music production workflows. For those cases, virtual channels can simply be disabled.

To view channel states, you can use **`sndctl(8)`**:

```
$ sndctl -v
```

Processing chain

An interesting feature of **`sound(4)`** is its processing chain. This includes:

- Mixing. This is actually exactly what was explained in the previous section, about how virtual channel streams are (de-)mixed.
- Volume control.
- Channel matrixing. **`sound(4)`** is capable of doing any-to-any channel matrixing, for example, mono to stereo, or stereo to 5.1 surround. This is done by converting streams from one interleaved PCM format to another.
- Basic parametric equalization.
- Format conversions.

- Resampling. There are three different resampling types, namely:
    - Linear.
    - Zero-order-hold (ZOH).
    - Sine Cardinal (SINC).

Each channel gets its own processing chain, and it includes only the necessary components. For instance, if the channel is configured to have a sample rate of 44100Hz, but the application feeds it audio sampled at 48000Hz, then `sound(4)` will need to include resampling in the channel's processing chain. Similarly, if the stream has the same sample rate as the channel, then that component will not be needed. The same applies to the other components.

A helpful way to visualize the processing chain is to print it using `sndctl(8)`. The following command will print the chain of each active channel:

```
$ sndctl feederchain
```

In the next section, we will discuss how and why, in some specialized cases, you might want to bypass the processing chain entirely.

Memory-mapped I/O and bit-perfect audio

Two of `sound(4)`'s liked features by low-latency application developers and audio enthusiasts, are that it provides bit-perfect mode support, as well as memory-mapped I/O.

Bit-perfect mode means that the audio stream skips all of `sound(4)`'s processing chain, and is fed more or less directly to the sound card. Applications have the added responsibility of making sure the stream's configuration (sample rate, format, channel matrix) matches that of the sound card. For instance, if the application wants to play audio sampled at 48000Hz, but the sound card does not support that sample rate, then it needs to take care of resampling the stream. As a result, this feature is disabled by default and is enabled only by applications that implement their own processing, and/or users who are sure their sound card will work properly in bit-perfect mode.

Memory-mapped I/O is similar to bit-perfect, in that the audio stream skips all processing done by `sound(4)`; in fact, bit-perfect has to be enabled in order to do memory-mapped I/O. However, the major difference between bit-perfect and memory-mapped I/O, is that the latter puts all of the audio buffer handling responsibilities entirely on the application, which means that it has to take care of not only the same things that an application using bit-perfect would, but to also make sure the buffer is synchronized correctly and that read/writes happen at the right time, with some help from `sound(4)`. If done right, and in the right environment, this can yield performance improvements, but is quite tricky and tedious to implement correctly, and so is mostly discouraged, unless the programmer really knows what they are doing.

Device drivers

Just like `sound(4)` is the bridge between userland and the device drivers, the device drivers are the bridge between `sound(4)` and the actual hardware. Apart from the fact that all sound drivers attach to `sound(4)` and communicate with it, the rest of their functionality depends on the driver itself. In a future article, we could present how to write a sound driver from scratch.

FreeBSD ships with support for the following sound cards:

Driver	Soundcards	Enabled by default
`snd_ai2s(4)`	Apple I2S	powerpc
`snd_als4000(4)`	Avance Logic ALS4000	
`snd_atiixp(4)`	ATI IXP	
`snd_cmi(4)`	CMedia CMI8338/CMI8738	amd64, i386
`snd_cs4281(4)`	Crystal Semiconductor CS4281	
`snd_csa(4)`	Crystal Semiconductor CS461x /462x/4280	amd64, i386
`snd_davbus(4)`	Apple Davbus	powerpc
`snd_emu10k1(4)`	SoundBlaster Live! and Audigy	
`snd_emu10kx(4)`	Creative SoundBlaster Live! and Audigy	amd64, i386
`snd_envy24(4)`	VIA Envy24 and compatible	
`snd_envy24ht(4)`	VIA Envy24HT and compatible	
`snd_es137x(4)`	Ensoniq AudioPCI ES137x	amd64, i386
`snd_fm801(4)`	Forte Media FM801	
`snd_hda(4)`	Intel High Definition Audio	amd64, i386
`snd_hdsp(4)`	RME HDSP	
`snd_hdspe(4)`	RME HDSPe	
`snd_ich(4)`	Intel ICH AC'97 and compati-ble	amd64, i386
`snd_maestro3(4)`	ESS Maestro3/Allegro-1	
`snd_neomagic(4)`	NeoMagic 256AV/ZX	
`snd_solo(4)`	ESS Solo-1/1E	
`snd_spicds(4)`	I2S SPI	
`snd_t4dwave(4)`	Trident 4DWave	
`snd_uaudio(4)`	USB audio and MIDI	auto-loaded on device plug
`snd_via8233(4)`	VIA Technologies VT8233	amd64, i386
`snd_via82c686(4)`	VIA Technologies 82C686A	
`snd_vibes(4)`	S3 SonicVibes	

There is also support for the following ARM chips:
- Allwinner A10/A20 and H3.
- Broadcom BCM2835.
- Freescale Vybrid.
- Freescale i.MX6.

If you own a sound card whose driver is not enabled by default on your machine's architecture, or you are using a custom kernel configuration without sound compiled in, and are unsure which driver your sound card uses, you can run the following command:

```
# kldload snd_driver
```

`snd_driver` is a meta-driver that loads all available drivers. Once you figure out which driver attaches to your sound card, you can load that one only.

Recent improvements

The sound subsystem has (and still is) undergone many improvements in the last two years, including a number of bug and crash fixes, the introduction of a growing Kyua test suite and a testing driver (`snd_dummy(4)`), as well as multiple cleanups and refactors.

A few important user-facing improvements include:

- Hot-unplugging is now possible. Users of USB sound cards on older versions of FreeBSD might remember that hot-unplugging the sound card usually resulted in the USB bus hanging, until the application using the now-detached device was manually killed (PR 194727).
- Floating-point audio support. This is a bit misleading, though, because we do not really, at least currently, support floating-point audio on the device driver level, but rather, we allow userland applications to use OSS with floating-point audio. This already fixes quite a few ports, such as Wine, that needed floating-point audio support from OSS.
- `sound(4)` now only exposes a single `/dev/dsp*` file for each device and does all the audio stream routing internally, using `DEVFS_CDEVPRIV(9)`, as opposed to exposing a `/dev/dsp*` file for each allocated audio stream. The current approach is cleaner both in implementation and in what is exposed to userland.
- Better out-of-the-box support for High Definition Audio (`snd_hda(4)`) sound cards. These cards are a constant pain for both developers and users, because they tend to come with non-standard configurations, meaning that we have to compensate for that by adding manual patches inside the driver or `/boot/device.hints`. A commonly reported issue is that sound is not automatically redirected to the headphones once they are plugged in, and vice versa. Several patches have recently been written for various sound cards that experience this issue, especially Framework laptops. It is very likely that you also have fallen victim to that issue. With that being said, since FreeBSD 15.0, there is a `devd(8)` configuration, `/etc/devd/snd.conf`, which attempts to automate this issue. The basic idea of the implementation is that whenever `snd_hda(4)` detects that a jack has been (un-)plugged, it issues a `devd(8)` notification, and `/etc/devd/snd.conf` will make sure to redirect sound to the appropriate device using `virtual_oss(8)`. This feature is still experimental, so there should be more refining as more people provide feedback.
- kqueue(2) support for `sound(4)`.

Userland utilities

You can find examples and more information for each of the following utilities in their respective manual pages.

sndctl(8)

sndctl(8) lists and manipulates sound card settings, such as the sample rate, sample format, bit-perfect and realtime mode settings, among others. It aims to be a replacement for **/dev/sndstat** (in fact, it uses it internally) and some of **sound(4)**'s sysctls, at least for most use cases:

```
$ sndctl
pcm3: <Realtek ALC295 (Analog 2.0+HP/2.0)> on hdaa1 (play/rec)
    name                 = pcm3
    desc                 = Realtek ALC295 (Analog 2.0+HP/2.0)
    status               = on hdaa1
    devnode              = dsp3
    from_user            = 0
    unit                 = 3
    caps                 = INPUT,MMAP,OUTPUT,REALTIME,TRIGGER
    bitperfect           = 0
    autoconv             = 1
    realtime             = 0
    play.format          = s16le:2.0
    play.rate            = 48000
    play.vchans          = 1
    play.min_rate        = 1
    play.max_rate        = 2016000
    play.min_chans       = 2
    play.max_chans       = 2
    play.formats         = s16le,s32le
    rec.rate             = 48000
    rec.format           = s16le:2.0
    rec.vchans           = 1
    rec.min_rate         = 1
    rec.max_rate         = 2016000
    rec.min_chans        = 2
    rec.max_chans        = 2
    rec.formats          = s16le,s32le
```

mixer(8)

mixer(8) deals with volume control, (un-)muting, recording source(s) selection, and default device setting. It was completely rewritten on FreeBSD 14.0, and comes with an improved interface and functionality:

```
$ mixer
pcm3:mixer: <Realtek ALC295 (Analog 2.0+HP/2.0)> on hdaa1 (play/rec) (default)
    vol       = 0.75:0.75     pbk
    pcm       = 1.00:1.00     pbk
    rec       = 0.37:0.37     pbk
    ogain     = 1.00:1.00     pbk
    monitor   = 0.67:0.67     rec src
```

virtual_oss(8)

virtual_oss(8) is a powerful sound server for OSS written by the late Hans Petter Selasky . It was part of ports (**audio/virtual_oss**) for years, but has been part of the base system since FreeBSD 15.0. It is again in active development, and there are already plenty of significant improvements being worked on and planned for the future.

As is mentioned in the 15.0 release notes, pre-FreeBSD 15.0 users of `virtual_oss(8)` can simply uninstall the `audio/virtual_oss` port and use the base system version. The only thing to keep in mind is that some functionality, which depends on third-party libraries, has been moved to separate ports, namely:

- `sndio` backend support: `audio/virtual_oss_sndio`
- bluetooth backend support: `audio/virtual_oss_bluetooth`
- `virtual_equalizer(8)`: `audio/virtual_oss_equalizer`

mididump(1)

`mididump(1)` is a simple utility that prints MIDI events for a given device in real time. This is useful for making sure a MIDI device works properly and that keys work and are mapped correctly.

```
$ mididump /dev/umidi0.0
Note on                   channel=1, note=53 (F3, 174.61Hz), velocity=109
Note off                  channel=1, note=53 (F3, 174.61Hz), velocity=127
Note on                   channel=1, note=55 (G3, 196.00Hz), velocity=100
Note off                  channel=1, note=55 (G3, 196.00Hz), velocity=127
Pitch bend                channel=1, change=1
```

beep(1)

`beep(1)`, as the name suggests, plays a beep sound. This is an easy way to verify sound works.

More

Apart from the utilities mentioned, there are a few more things provided by the sound subsystem:

What	Description	Documentation
`mixer(3)`	A C library for interacting with the OSS mixer.	`man 3 mixer`
`sndstat(4)`	An `nv(9)` interface for listing device information, as well as registering userland sound devices. Used internally by `sndctl(8)` and `virtual_oss(8)`.	`man 4 sndstat`
`hw.snd.*`	Global `sysctl(8)` variables.	`man 4 sound`
`dev.pcm.*`	Device-specific `sysctl(8)` variables.	`man 4 sound`
Driver-specific `sysctl(8)` variables		Refer to the respective driver's manual page.

FreeBSD for music production?!

You might be thinking this is a joke, but it is, in fact, a topic that has been coming up more and more in recent years, and we have already seen a few related talks in recent conferences, more specifically:

- Goran Mekić, FOSDEM 2019
- Goran Mekić, EuroBSDCon 2022
- Charlie Li, BSDCan 2024
- Christos Margiolis, FreeBSD DevSummit 09/2024
- Christos Margiolis, BSDCan 2025
- Charlie Li, EuroBSDCon 2025

FreeBSD is, without a doubt, not the operating system a musician or producer would typically think about when it comes to music production, however, this is partially the case because of a lack of "marketing", for lack of a better word. In reality, FreeBSD offers a solid, fast, and highly configurable sound subsystem, it has a consistently rapidly growing collection of open source Digital Audio Workstations, LV2 plugins, and other types of production/music software, and it can work with any non-native sound subsystem (ALSA, sndio, JACK, Pulseaudio, Pipewire, etc.), in case OSS is not desirable.

I genuinely think that if we continue this trend of consistently maintaining and developing the sound subsystem, porting and developing more software, as well as showcasing in practice why FreeBSD can be an alternative for audio and music production, both in conferences and online, we could, one day, see FreeBSD gaining significant popularity among audiophiles and musicians.

Reporting and resolving bugs

All software might contain bugs from time to time, and the sound subsystem is no exception. Providing sufficient information is always necessary, and opening a bug report or sending an email with just a "sound does not work on my machine" is not really helpful. Attaching the output of the following commands should be enough in most cases:

1. `uname -a`
2. `sndctl -v`
3. `mixer -a`
4. `sysctl hw.snd dev.pcm`, as well as the driver-specific sysctls, if any.
5. `dmesg`, after setting `hw.snd.verbose=4` and reproducing the bug.
6. Logs, if any, from the application with which the bug is reproduced.

Conclusion

Hopefully, this article has helped with presenting the general structure of the sound subsystem as a whole, at its current state. It would be great to see even more people interested in FreeBSD sound in the future! The `freebsd-multimedia@FreeBSD.org` mailing list is where most of the discourse happens, so make sure to keep an eye on it.

CHRISTOS MARGIOLIS is an independent contractor and FreeBSD src committer from Greece.