Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)

Smilen Dimitrov and Stefania Serafin
Medialogy, Aalborg University Copenhagen
Lautrupvang 15
DK-2750 Ballerup,
Denmark,
{sd, sts}@create.aau.dk

Abstract
Understanding the construction and implementation of sound cards (as examples of digital audio hardware) can be a demanding task, requiring insight into both hardware and software issues. An important step towards this goal, is the understanding of audio drivers - and how they fit in the flow of execution of software instructions of the entire operating system.

The contribution of this project is in providing sample open-source code, and an online tutorial [1] for a mono, capture-only, audio driver - which is completely virtual; and as such, does not require any soundcard hardware. Thus, it may represent the simplest form of an audio driver under ALSA, available for introductory study; which can hopefully assist with a gradual, systematic understanding of ALSA drivers' architecture - and audio drivers in general.

Keywords
Sound card, audio, driver, ALSA, Linux

1 Introduction
Prospective students of digital audio hardware, could choose the sound card as a topic of study: on one hand, it has a clear, singular task of managing the PC’s analog interface for playback and capture of digital audio data - as well as well-established expectations by consumer users in terms of its role; on the other hand, its understanding can be said to be cross-disciplinary, as it encompasses several (not necessarily overlapping) areas of design: analog and digital electronics related to soundcard hardware and PC bus interface implementation; PC operating system drivers; and high-level PC audio software.

Gaining a sufficient understanding of the interplay between these different domains in a working implementation can be an overwhelming task; thus, not surprisingly, the area of digital audio hardware design and implementation (including soundcards) is currently dominated by industry. Recent developments in open source software and hardware may lower the bar for entry of newcomer DIY enthusiast - however, the existence of many open source drivers for commercial cards doesn’t necessarily ease the introductory study of a potential student.

In essence, an implementation of a soundcard will eventually demand dealing with the issue of an operating system driver. In the current situation, a prospective student is then faced with a ‘chicken-and-egg’ problem: proper understanding of drivers requires knowledge of the hardware (which the drivers were written for); and yet understanding the hardware, involves understanding of how the drivers are supposed to interface with it1. A straightforward way out, would be to study a ‘virtual’ driver - that is, a driver not related to an actual hardware; in that case, a student would be able to focus solely on the software interaction between the driver, and the high-level audio program that calls it. Unfortunately, in the case of the ALSA driver architecture for Linux, pre-existing examples of virtual drivers are in fact not trivial2 - and, just as existing ALSA driver tutorials, assume previous knowledge of bus interfaces (and thus hardware).

The minivosc driver source code with the corresponding tutorial (on the ALSA website [1]) represents the simplest possible virtual ALSA driver, that does not require additional hardware. It has already led to the development of the driver used in the (possibly first) demonstration of an open soundcard system in AudioArduino [2] (and fur-

1and the lack of open card hardware designs for study makes this problem more difficult
2and may require existence of real soundcards on the system
ther used in [3]) - and as it limits the discussion to only the software interaction between driver and high-level software, disregarding issues in bus interfacing and hardware - it would represent a conceptually simpler entry level for a prospective student of sound card drivers.

2 Premise

Personal computer users working with audio typically rely on high-level audio software (from media players such as VLC, to more specialized software like Audacity, or the wave editor Pure Data, or the wave editor Audacity) to perform their needed tasks – and the sound card (as hardware) to provide an analog interface to and from audio equipment. This necessarily puts demands on the operating system of the PC, to provide a standardized way to access (what could be different types of) audio hardware. An operating system, in turn, would provide an audio or soundcard driver API (application programming interface), which should allow for programming of a driver that: abstracts some of the ‘inner details’ of the soundcard implementation; and exposes a standardized interface to the high-level audio software (that may want to utilize this driver). This, in principle, allows interfacing between software and hardware released by different vendors/publishers.

Earlier work like [4] attempts to provide a systematic approach to soundcard implementation; however, one clear conclusion from such a naïve approach is that: regardless of the capabilities of the hardware - one cannot achieve a fine control of timing required for audio, by using what corresponds to a simple ‘user space’ C program. Problems like these are typically solved within the driver programming framework of a given operating system - and as such, acquaintance with driver programming becomes a necessity for anyone aiming to understand development of digital audio hardware for personal computers. In terms of FLOSS GNU/Linux-based operating systems, the current driver programming framework - as it relates to soundcards and audio - is provided by the Advanced Linux Sound Architecture (ALSA). ALSA supersedes the previous OSS (Open Sound System) as the default audio/soundcard driver framework for Linux (since version 2.6 of the kernel [5]), and it is the focus of this paper, and the eponymous minivosc driver (and tutorial). The minivosc driver was developed on Ubuntu 10.04 (Lucid), utilizing the 2.6.32 version of the Linux kernel; the code has been released as open source on Sourceforge, and it can be found by referring to the tutorial page [1].

2.1 Initial project issues

The minivosc project starts from the few readily available (and ‘human-readable’) resources related to introductory ALSA driver development: [8], [9], [10], and [11]. Most of these resources base their discussions on conceptual or undisclosed hardware, making them difficult to read for novices. On the other hand, there are few examples of virtual soundcard drivers, such as the driver source files dummy.c (in the Linux kernel source tree [12]) and aloop-kernel.c (in the ALSA source tree [13]); however, these drivers don’t have much documentation, and can present a challenge for novices. All these resources [8; 9; 10; 11; 12; 13] have been used as a basis here, to develop an example of a minimal virtual oscillator (minivosc) driver.

3 Architectural overview of PC audio

Even if the minivosc driver is a virtual one, one still needs an overview of the corresponding hardware architecture - also for understanding in what sense is this driver ‘virtual’. As a simplified illustration, consider Fig. 1.

A driver will typically control transfers of data between the soundcard and the PC, based on instructions from high-level software. The direction from the soundcard to the PC is the capture direction; the opposite direction (from the PC to the soundcard) is the playback direction; a soundcard capable of delivering both data transfer directions

---

3Note that software like JACK - while it can be considered more 'low-level' than consumer audio software - is still intended to route data between ‘devices’. Since it is the driver that provides this ‘device’ (as a OS construct that software can interface with) in the first place - drivers lay in a lower architectural layer than even software like JACK, and so involve different development considerations.

4free/libre/open source software
simultaneously can be said to be a full-duplex device.

While Fig. 1 shows the hard disk as (ultimately) both the source for the playback direction, and the destination for the capture direction - within this process, the CPU may use RAM memory at its discretion. In fact, the driver is typically exposed to pointers to byte arrays (buffers) in memory (in ALSA known as PCM (sub)streams [7, 'PCM (digital audio) interface'], and named dma area), that represent streams in each direction.

In terms of audio streams, Fig. 1 demonstrates a device capable of two mono (or one stereo) inputs, and two mono (or one stereo) outputs. Since audio devices like microphones (or amplifiers for speakers) typically interface through analog electronic signals - this implies that for each 'digital' input [or output] audio stream, a corresponding analog-to-digital (ADC) [or digital-to-analog (DAC)] converter hardware needs to be present on the soundcard.

As the main role of the soundcard is to provide an analog electronic audio interface to the PC - the role of the ADC and DAC hardware is, of course, central. However, the PC will typically interface to external hardware through a dedicated bus for this purpose. This means, that some bus interfacing electronics - that will decode the signals from the PC, and provide signals that will drive (at the very least) the ADC/DAC converters - needs to be present on the soundcard as well.

An ALSA driver uses a particular terminology when addressing these architectural surroundings. The 'soundcard' on Fig. 1 will be considered to be a card by the driver. One level deeper, things can get a bit more complicated: assuming that Fig. 1 represents a stereo soundcard, it would have one input stereo connector (attached to two ADCs), and one output stereo connector (attached to two DACs); an ALSA driver would correspondingly be informed about a card, that has one stereo input device (consisting of two subdevices) - and one stereo output device (consisting, likewise, of two subdevices). Note that: “...we use subdevices mainly for hardware which can mix several streams together [14]” and “typically, specifying a sound card and device will be sufficient to determine on which connector or set of connectors your audio signal will come out, or from which it is read... Subdevices are the most fine-grained objects ALSA can distinguish. The most frequently encountered cases are that a device has a separate subdevice for each channel or that there is only one subdevice altogether [15].”

The ALSA driver is informed about such a hierarchical relationship (between card, devices and subdevices) through structures (C structs, written by the driver author in the driver source files) - defined mostly through use of other structures, predefined by the ALSA framework (alias the ALSA 'middle layer'). The driver code, additionally, establishes a relationship between these structs, and the PCM stream data that will be assigned to each in memory; and connects these to predefined ALSA framework driver functions, which define the driver (and the corresponding hardware) behavior at runtime. Finally, Fig. 1 shows that other types of devices, such as a MIDI interface, can also be present on the soundcard. The ALSA framework

---

6Note, however, that this correspondence could, in principle, be solved by a single ADC or DAC element - along with a (de)multiplexer which would implement time-sharing of the element (for multiple channels)

7noting that, in principle, the buses used for hard disks (such as Integrated Drive Electronics (IDE)) or RAM (known as 'Memory Interconnect') can be distinct

---

8For example, [4] describes a device that interfaces through the Industry Standard Architecture (ISA) bus - and uses standard TTL components (such as 74LS08, 74LS688, 74LS244, etc) to implement a bus interface; [2] describes a device that interfaces through the Universal Serial Bus (USB) - and uses the FT232 chip by FTDL to implement a bus interface

9noting that, in principle, the driver should be able to handle multiple cards; and be able to individually address each one
has facilities to address such needs too - as well as having a so-called mixer interface\(^\text{10}\) - which will not be discussed here.

**Application level** From the PC perspective, a high-level audio software (audio application) is used, in first instance, to issue start and stop of audio playback or capture. When such a high-level command is issued by the user, the audio application communicates to the driver through the application-level API and: obtains a handle to the relevant structures; initializes and allocates variables; opens the PCM device; specifies hardware parameters\(^\text{11}\) and type of access (interleaved or not) - and then starts with reading from (for capture) or writing to (for playback) the PCM device, by using ALSA API functions (such as `snd_pcm_writei/snd_pcm_writen` or `snd_pcm_readi/snd_pcm_readn`) [16]. The PCM device is representation of a source (or destination) of an audio stream\(^\text{12}\). The kernel responds to the application API calls by calling the respective code in the kernel driver, implemented using the kernel (ALSA driver) API [8].\(^\text{13}\)

4 Concept of minivosc

A user would, arguably, expect to hear actual reproduced sound upon clicking 'play'; while recording, in principle, doesn’t involve user sensations other than indication by the audio software (e.g. rendering of captured audio waveform). Taking this into account, it becomes clear that the stated purpose of minivosc - to be a ‘virtual’ driver (independent of any actual additional soundcard hardware) - can only be demonstrated in the capture direction\(^\text{14}\): as the driver simply has refer-

\(^{10}\)Which allows for, say, individual volume control directly from the main OS volume applet

\(^{11}\)Access type, sample format, sample rate, number of channels, number of periods and period size

\(^{12}\)And it can have: "plughw" or "hw" interface; playback or capture direction; and standard (blocking), non-blocking and asynchronous modes (see also [7, 'PCM (digital audio) interface'])

\(^{13}\)Note that the application doesn’t have to talk to the driver directly; there could be intermediate layers, forming a Linux audio software stack (see [17]). However, in this paper, we focus solely on the perspective of the ALSA kernel driver.

\(^{14}\)However, note that `aloop-kernel.c` [13], is also a ‘virtual’ driver, and yet works in both directions; however, since it is intended to ‘loop back’ audio data between applications and devices[18], the virtual setups possible can be reduced to the case when the “loopback” driver routes

ences to data arrays in memory, the effect of playing back (i.e., copying) data to non-existing hardware will be pretty much undetectable\(^\text{15}\). However, even with non-existing hardware, we can always write some sort of predefined or random data to the capture buffers in memory - which would result with visible incoming data in the high-level audio software (like when performing 'record' in Audacity).

To avoid the conceptualization problems of ALSA devices vs. subdevices, the minivosc driver is deliberately defined as a mono, 8-bit, capture-only driver, working at 8 kHz (the next-lowest\(^\text{16}\) rate ALSA supports). The 8-bit resolution allows also for direct correspondence between: the digital representation of a single analog sample; and the storage unit of the corresponding arrays (buffers) in memory, which are defined as `char*`. Hence, one byte in memory buffer represents one analog sample, the next byte represents the next analog sample, etc. This allows for simplification of the process of wrapping data in a ring buffer, and thus easier grasping of the remaining key issues in ALSA driver implementation.

5 Driver structures

The minivosc driver contains four key structures - three of which are required by (and based on predefined types in) the ALSA framework:

- **struct** variable of type `snd_pcm_hardware` (required) - sets the allowed sample formats, sampling rates, number of channels, and buffering properties
- **struct** variable of type `snd_pcm_ops` (required) - assigns the actual functions, that should respond to predefined ALSA callbacks
- **struct** variable of type `platform_driver` (required) - named `minivosc_driver`, it describes the driver, and at the same time, determines the bus interface type
- **struct** variable of type `minivosc_device` - custom structure that contains all other parameters related one audio application’s data written to its playback interface, back to its capture interface; and another audio application grabs data from the 'loopback' capture interface and writes it to disk.

\(^{15}\)Similar to, in Linux parlance, ‘piping’ data to `/dev/null`. While a specific consumer of such data could be programmed, that alone complicates the understanding of interaction between typical audio software and drivers

\(^{16}\)The lowest ALSA rate being 5512 Hz, see `include/sound/pcm.h` in Linux source [19]
to the soundcard, as well as pointers to the digital audio (PCM) data in memory.

The `minivosc_driver` struct variable defines the `_probe` and `_remove` functions, required for any Linux driver; however, by choosing the struct type, we also determine the type of bus this driver is supposed to interface through. For instance, a PCI soundcard driver would be of type `struct pci_driver`; whereas a USB soundcard driver would be of type `struct usb_driver` (see [1]). However, `minivosc` is defined as `platform_driver`, where "platform devices are devices that typically appear as autonomous entities in the system" [20, 'platform.txt'] - and as such, it will not need actual hardware present on any bus on the PC, in order for the driver to be loaded completely\(^\text{17}\).

The `snd_pcm_ops` type variable simply points to the actual functions that are to be executed as the predefined ALSA callbacks, which are discussed in the next section. The different fields in `snd_PCM_Hardware` allow the device capabilities in terms of sampling resolution (i.e., analog sample format) and sampling rate to be specified. For this purpose, there are predefined bit-masks in ALSA's `pcm.h` [19], such as `SNDRV_PCM_RATE_8000` or `SNDRV_PCM_FMTBIT_U8` (for 8 kHz rate, or for sample format of 8-bit treated as unsigned byte, respectively). One should be aware that audio software may treat these specifications differently: for instance, having `arecord` capture from the `minivosc` driver, will result with an 8-bit, 8 kHz audio file - simply because that is the default format for `arecord`. On the other hand, Audacity in the same situation - while acknowledging the driver specifications - will also internally convert all captures to the default 'project settings', for which the minimum possible values are 8000 Hz and 16-bit [21].\(^\text{18}\)

One of the most important structures is what we could call the 'main' `device` structure, here `minivosc_device`. It can also be a bit difficult to understand, especially since it is - in large part - up to the driver authors themselves to set up the structure, and its relationships to built-in ALSA structures. These relationships are of central interest, because a driver author must know the location of memory representing the digital audio streams (`snd_pcm_runtime->dma_area` in Fig. 2), in order to implement any digital audio functionality of the driver. And finding this memory location is not trivial - which is maybe best presented in graphical manner, as in Fig. 2, which shows a partial scope of the 'main' structure `minivosc_device` and its relationships.

On Fig. 2, only `minivosc_device` has been written as part of the driver code - all other structs (with darker backgrounds) are built-ins, provided by ALSA. Pointers are shown on left edge of boxes; self-contained struct variables are on the bottom edge\(^\text{19}\). Some relationships (such as `snd_pcm_substream->runtime` to `snd_pcm_runtime` pointing) are set up internally by ALSA; the relationships to the 'main device' structure (`minivosc_device`) have to be coded by the driver author. Further complication is that the authored relationships can not be established at the same spot in the driver code - as some structures become available only in specific ALSA callbacks.

This is a conceptual departure from the typical basic understanding of program execution - where a predetermined sequential execution of commands is assumed. Instead, driver programming may conceptually be closer to GUI programming, where the author typically writes `callback` functions that run whenever a user performs some action. Additionally, we can expect to encounter different amount of instances of some of these structs! For example, `snd_pcm_substream` can carry data for a given output connector, which could be stereo. So, if a stereo file is loaded in audio software, and 'play' is clicked - we could expect ALSA to pass a single `snd_pcm_substream`, carrying data for both channels, to our driver. However, if we are trying to play a 5.1 surround file, which em-

\(^\text{17}\)which is not the default behavior for actual hardware drivers - they will simply not run some of their predefined callbacks, if the hardware is not present on the bus

\(^\text{18}\)While these captures can be exported from Audacity as 8-bit, 8 kHz audio files - that process implies an additional conversion from the internal 16-bit format.

\(^\text{19}\)Note, the ALSA struct boxes show only a small selection of the structs' actual members; while the 'main device' struct still contains some unused variables, leftover from starting example code. Connections are colored for legibility. Unlike a more detailed UML diagram, a map like Fig. 2 helps only in a specific context: e.g., the driver is supposed to write to the `dma_area` when the `_timer_function` runs, however this function provides a reference to `minivosc_device`; the map then allows for a quick overview of structure field relationship, so a direct pointer to the `dma_area` can be obtained for use within the function.
Figure 2: Partial 'structure relationship map' of the minivosc driver.

plays 2 stereo and one mono connector - we should expect three snd_pcm_substreams to be passed to our driver. This could further confuse high-level programmer newcomers, that might expect to receive something like an array of substreams in such a case: instead, ALSA may call certain callbacks multiple times - and it is up to the driver author to store references to these substreams.

minivosc avoids these problems as a mono-only driver - thus within the code, we can expect only one instance of each struct shown on Fig. 2; and the reference to the only snd_pcm_substream can be found directly on the main 'device' struct, minivosc_device. This allows us easier focus on another important aspect of ALSA - the timing of execution of callbacks, which is necessary for understanding the driver initialization process in general.

6 Execution flow and driver functions

The device driver architecture of Linux specifies a 'driver model' [20], and within that, certain callback functions that a driver should expose. In the case of minivosc, first the __init and __exit macros ([22, Chapter 2.4]) are implemented, as functions named alsa_card_minivosc_init and alsa_card_minivosc_exit. These functions run when a driver module is loaded and unloaded: the kernel will automatically load modules, built in the kernel, at boot time - while modules built 'out of tree' have to be loaded manually by the user, through the use of the insmod program. The __init function in minivosc registers the driver, and attempts to iterate through the attached soundcards. As minivosc is a 'platform' driver, and there is no actual hardware - the __init, in this case, is made to always result with detecting a single (virtual) 'card'. Next in line of predefined callbacks are __probe and __remove [20, 'driver.txt'], in minivosc implemented as minivosc_probe and minivosc_remove. In principle, they would run when a (soundcard) hardware device is attached to/disconnected from the PC bus: for instance, __probe would run when the user connects a USB soundcard to the PC by inserting the USB connector - if the driver is already loaded in memory. For permanently attached devices (think PCI soundcards), __probe would run immediately after __init detects the cards; thus, in the case of minivosc, __probe will run immediately after __init, at the moment when the driver is loaded (by using insmod).

The minivosc driver code informs the system about which are its init/exit functions, by use of module_init/module_exit facility (see [23, 'Chapter 2']); while it specifies which are its probe/remove functions through use of the platform_driver structure. Finally, last in line of predefined callbacks are the ALSA specific callbacks; the driver code tells the system which are these functions, through the predefined ALSA struct snd_pcm_ops.20

While ALSA may define more snd_pcm_ops callbacks [9], there are 8 of them being used in minivosc, by assigning them to functions: one, snd_pcm_lib_ioctl, being defined by ALSA – and seven snd_pcm_ops functions written as part of minivosc: minivosc_pcm_open, minivosc_pcm_suspend, minivosc_pcm_prepare, minivosc_pcm_release, minivosc_pcm_close, minivosc_pcm_format. As clarification - here is the order of execution of above callbacks for the minivosc driver, for some common events:

- driver loading: __init, then __probe
- start of recording: __open, then __hw_params, then

20Note that the term 'PCM' is used in ALSA to refer generally to aspects related to digital audio - and not to the particular 'Pulse Code Modulation' method as known from electronics (although that is where the term derives from [7, 'PCM (digital audio) interface']).
_prepare, then _trigger  
• end of recording: _trigger, then _hw_free, then _close  
• driver unloading: _exit, then _remove

We already mentioned that for the minivosc driver, loading/unloading events happen when the insmod/rmmod commands are executed. 'Start of recording' event would be the moment when the 'record' button has been pressed in Audacity; or the moment when we run arecord from the command line – correspondingly, 'end of recording' event is when we hit the 'stop' button in Audacity; or when arecord exits (if, for instance, it has been set to capture for only a certain amount of time). However, note that – even with all of this in place – the actual performance of the driver in respect to digital audio is still not defined; memory buffer handling is also needed.

6.1 Audio data in memory (buffers) and related execution flow

As noted in Sec. 5 'Driver structures', one of the central issues in ALSA driver programming is the location in memory, where audio PCM data for each substream is kept - the dma_area field being a pointer to it. In principle, each substream can carry multi-channel data: for instance, a 16-bit sample would be represented as two consecutive bytes in the dma_area; while stereo samples could be interleaved [24]. Thus ALSA introduces the concept of frames [25], where a frame represents the size of one analog sample for all channels carried by a substream. As minivosc is specified as a mono 8-bit driver, we can be certain that each byte in its dma_area will represent a single sample - and that one frame will correspond to exactly one byte.

The approach to implementing the sampling rate that minivosc has (taken from [13]), is to use the Linux system timer ([26, 'Kernel Mechanisms'], [23, 'Chapter 6']). Note that standard Linux system timers are "only supported at a resolution of 1 jiffy. The length of a jiffy is dependent on the value of HZ in the Linux kernel, and is 1 millisecond on i386 [27]." However, there also exist so-called high-resolution timers [28] (for their basic use in ALSA, see [12]).

6.2 The sound of minivosc - Driver execution modes

The driver writes in the dma_area capture buffer repeatedly (as controlled by timers), within the _xfer_buf function - or more precisely, within the minivosc_fill_capture_buf function called by it. In the minivosc code, three different variants can be chosen (at compile time), for copying a small predefined 'waveform grain' array repeatedly in the capture buffer, which results in an audible oscillation when the capture is played back (hence oscillator in the name). Note the need to 'wrap' the writing to the capture buffer array, since in ALSA, it is defined as a circular or ring buffer [24]. Finally, all of the three 'audio generation' algorithms can be commented, in which case the minivosc driver will simply write a constant value in the buffer. There is an additional facility, called 'buffermarks', which indicate the start and end of the current chunk, as well as the start and end of the dma_area - which can be used to visualize buffer sizes.

7 Conclusions

The main intent of minivosc is to serve as a basic introduction to one of the most difficult issues in soundcard driver programming: handling of digital audio. Given that many newcomers may have previous acquaintance with 'userland' programming, the conceptual differences from user-space to kernel programming (including debugging [1]) can be a major stumbling block. While a focus on capture only, 8-bit / 8 kHz mono driver leaves out many of the issues that are encountered in working with real soundcards, it can also be seen as a basis for discussion of [2], which demonstrates full-duplex mono @ 8-bit / 44.1 kHz (and can interface with stereo, 16-bit playback). Thus, the main contribution of this paper, driver code and tutorial would be in easing the learning curve of newcomers, interested in ALSA soundcard drivers, and digital audio in general.

8 Acknowledgments

The authors would like to thank the Medialogy department at Aalborg University in Copenhagen, for the support of this work as a part of a currently ongoing PhD project.

References


