A Tutorial on Using the ALSA Audio API

This document attempts to provide an introduction to the ALSA Audio API. It is not a complete reference manual for the API, and it does not cover many specific issues that more complex software will need to address. However, it does try to provide enough background and information for a reasonably skilled programmer but who is new to ALSA to write a simple program that uses the API.

Contents

Understanding Audio Interfaces

Let us first review the basic design of an audio interface. As an application developer, you don't need to worry about this level of operation - its all taken care of by the device driver (which is one of the components that ALSA provides). But you do need to understand what is going at a conceptual level if you want to write efficient and flexible software.

An audio interface is a device that allows a computer to receive and to send audio data from/to the outside world. Inside of the computer, audio data is represented a stream of bits, just like any other kind of data. However, the audio interface may send and receive audio as either an analog signal (a time-varying voltage) or as a digital signal (some stream of bits). In either case, the set of bits that the computer uses to represent a particular sound will need to be transformed before it is delivered to the outside world, and likewise, the external signal received by the interface will need to be transformed before it is useful to the computer. These two transformations are the raison d'etre of the audio interface.

Within the audio interface is an area referred to as the "hardware buffer". As an audio signal arrives from the outside world, the interface converts it into a stream of bits usable by the computer and stores it in the part hardware buffer used to send data to the computer. When it has collected enough data in the hardware buffer, the interface interrupts the computer to tell it that it has data ready for it. A similar process happens in reverse for data being sent from the computer to the outside world. The interface interrupts the computer to tell it that there is space in the hardware buffer, and the computer proceeds to store data there. The interface later converts these bits into whatever form is needed to deliver it to the outside world, and delivers it. It is very important to understand that the interface uses this buffer as a "circular buffer". When it gets to the end of the buffer, it continues by wrapping around to the start.

For this process to work correctly, there are a number of variables that need to be configured. They include:

- what format should the interface use when converting between the bitstream used by the computer and the signal used in the outside world?
- at what rate should samples be moved between the interface and the computer?
- how much data (and/or space) should there be before the device interrupts the computer?
- how big should the hardware buffer be?

The first two questions are fundamental in governing the quality of the audio data. The second two questions affect the "latency" of the audio signal. This term refers to the delay between

1. data arriving at the audio interface from the outside world, and it being available to the computer ("input latency")
2. data being delivered by the computer, and it being delivered to the outside world ("output latency")

Both of these are very important for many kinds of audio software, though some programs do not need be concerned with such matters.

What a typical audio application does

A typical audio application has this rough structure:

```
open_the_device();
set_the_parameters_of_the_device();
while (!done) {
     /* one or both of these */
     receive_audio_data_from_the_device();
     deliver_audio_data_to_the_device();
}
close the device
```

A Minimal Playback Program

This program opens an audio interface for playback, configures it for stereo, 16 bit, 44.1kHz, interleaved conventional read/write access. Then its delivers a chunk of random data to it, and exits. It represents about the simplest possible use of the ALSA Audio API, and isn't meant to be a real program.

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>

main (int argc, char *argv[])
{
	int i;
	int err;
	short buf[128];
	snd_pcm_t *playback_handle;
	snd_pcm_hw_params_t *hw_params;

	if ((err = snd_pcm_open (&playback_handle, argv[1], SND_PCM_STREAM_PLAYBACK, 0)) < 0) {
		fprintf (stderr, "cannot open audio device %s (%s)\n", 
			 argv[1],
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
		fprintf (stderr, "cannot allocate hardware parameter structure (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_any (playback_handle, hw_params)) < 0) {
		fprintf (stderr, "cannot initialize hardware parameter structure (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_set_access (playback_handle, hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
		fprintf (stderr, "cannot set access type (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_set_format (playback_handle, hw_params, SND_PCM_FORMAT_S16_LE)) < 0) {
		fprintf (stderr, "cannot set sample format (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_set_rate_near (playback_handle, hw_params, 44100, 0)) < 0) {
		fprintf (stderr, "cannot set sample rate (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params_set_channels (playback_handle, hw_params, 2)) < 0) {
		fprintf (stderr, "cannot set channel count (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	if ((err = snd_pcm_hw_params (playback_handle, hw_params)) < 0) {
		fprintf (stderr, "cannot set parameters (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	snd_pcm_hw_params_free (hw_params);

	if ((err = snd_pcm_prepare (playback_handle)) < 0) {
		fprintf (stderr, "cannot prepare audio interface for use (%s)\n",
			 snd_strerror (err));
		exit (1);
	}

	for (i = 0; i < 10; ++i) {
		if ((err = snd_pcm_writei (playback_handle, buf, 128)) != 128) {
			fprintf (stderr, "write to audio interface failed (%s)\n",
				 snd_strerror (err));
			exit (1);
		}
	}

	snd_pcm_close (playback_handle);
	exit (0);
}
```

A Minimal Capture Program

This program opens an audio interface for capture, configures it for stereo, 16 bit, 44.1kHz, interleaved conventional read/write access. Then its reads a chunk of random data from it, and exits. It isn't meant to be a real program.

```
#include <stdio.h>
#include <stdlib.h>
#include <alsa/asoundlib.h>

main (int argc, char *argv[])
{
	int i;
	int err;
```

```
        short buf[128];
        snd_pcm_t *capture_handle;
        snd_pcm_hw_params_t *hw_params;

        if ((err = snd_pcm_open (&capture_handle, argv[1], SND_PCM_STREAM_CAPTURE, 0)) < 0) {
                fprintf (stderr, "cannot open audio device %s (%s)\n",
                         argv[1],
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
                fprintf (stderr, "cannot allocate hardware parameter structure (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_any (capture_handle, hw_params)) < 0) {
                fprintf (stderr, "cannot initialize hardware parameter structure (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_set_access (capture_handle, hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
                fprintf (stderr, "cannot set access type (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_set_format (capture_handle, hw_params, SND_PCM_FORMAT_S16_LE)) < 0) {
                fprintf (stderr, "cannot set sample format (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_set_rate_near (capture_handle, hw_params, 44100, 0)) < 0) {
                fprintf (stderr, "cannot set sample rate (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params_set_channels (capture_handle, hw_params, 2)) < 0) {
                fprintf (stderr, "cannot set channel count (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        if ((err = snd_pcm_hw_params (capture_handle, hw_params)) < 0) {
                fprintf (stderr, "cannot set parameters (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        snd_pcm_hw_params_free (hw_params);

        if ((err = snd_pcm_prepare (capture_handle)) < 0) {
                fprintf (stderr, "cannot prepare audio interface for use (%s)\n",
                         snd_strerror (err));
                exit (1);
        }

        for (i = 0; i < 10; ++i) {
                if ((err = snd_pcm_readi (capture_handle, buf, 128)) != 128) {
                        fprintf (stderr, "read from audio interface failed (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }
        }

        snd_pcm_close (capture_handle);
        exit (0);
}
```

A Minimal Interrupt-Driven Program

This program opens an audio interface for playback, configures it for stereo, 16 bit, 44.1kHz, interleaved conventional read/write access. It then waits till the interface is ready for playback data, and delivers random data to it at that time. This design allows your program to be easily ported to systems that rely on a callback-driven mechanism, such as JACK, LADSPA, CoreAudio, VST and many others.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <poll.h>
#include <alsa/asoundlib.h>

snd_pcm_t *playback_handle;
short buf[4096];

int
playback_callback (snd_pcm_sframes_t nframes)
{
        int err;

        printf ("playback callback called with %u frames\n", nframes);

        /* ... fill buf with data ... */
```

```
                if ((err = snd_pcm_writei (playback_handle, buf, nframes)) < 0) {
                        fprintf (stderr, "write failed (%s)\n", snd_strerror (err));
                }

                return err;
        }

        main (int argc, char *argv[])
        {

                snd_pcm_hw_params_t *hw_params;
                snd_pcm_sw_params_t *sw_params;
                snd_pcm_sframes_t frames_to_deliver;
                int nfds;
                int err;
                struct pollfd *pfds;

                if ((err = snd_pcm_open (&playback_handle, argv[1], SND_PCM_STREAM_PLAYBACK, 0)) < 0) {
                        fprintf (stderr, "cannot open audio device %s (%s)\n",
                                 argv[1],
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_malloc (&hw_params)) < 0) {
                        fprintf (stderr, "cannot allocate hardware parameter structure (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_any (playback_handle, hw_params)) < 0) {
                        fprintf (stderr, "cannot initialize hardware parameter structure (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_set_access (playback_handle, hw_params, SND_PCM_ACCESS_RW_INTERLEAVED)) < 0) {
                        fprintf (stderr, "cannot set access type (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_set_format (playback_handle, hw_params, SND_PCM_FORMAT_S16_LE)) < 0) {
                        fprintf (stderr, "cannot set sample format (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_set_rate_near (playback_handle, hw_params, 44100, 0)) < 0) {
                        fprintf (stderr, "cannot set sample rate (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params_set_channels (playback_handle, hw_params, 2)) < 0) {
                        fprintf (stderr, "cannot set channel count (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                if ((err = snd_pcm_hw_params (playback_handle, hw_params)) < 0) {
                        fprintf (stderr, "cannot set parameters (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                snd_pcm_hw_params_free (hw_params);

                /* tell ALSA to wake us up whenever 4096 or more frames
                   of playback data can be delivered. Also, tell
                   ALSA that we'll start the device ourselves.
                */

                if ((err = snd_pcm_sw_params_malloc (&sw_params)) < 0) {
                        fprintf (stderr, "cannot allocate software parameters structure (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }
                if ((err = snd_pcm_sw_params_current (playback_handle, sw_params)) < 0) {
                        fprintf (stderr, "cannot initialize software parameters structure (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }
                if ((err = snd_pcm_sw_params_set_avail_min (playback_handle, sw_params, 4096)) < 0) {
                        fprintf (stderr, "cannot set minimum available count (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }
                if ((err = snd_pcm_sw_params_set_start_threshold (playback_handle, sw_params, 0U)) < 0) {
                        fprintf (stderr, "cannot set start mode (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }
                if ((err = snd_pcm_sw_params (playback_handle, sw_params)) < 0) {
                        fprintf (stderr, "cannot set software parameters (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                /* the interface will interrupt the kernel every 4096 frames, and ALSA
```

```
                   will wake up this program very soon after that.
                */

                if ((err = snd_pcm_prepare (playback_handle)) < 0) {
                        fprintf (stderr, "cannot prepare audio interface for use (%s)\n",
                                 snd_strerror (err));
                        exit (1);
                }

                while (1) {

                        /* wait till the interface is ready for data, or 1 second
                           has elapsed.
                        */

                        if ((err = snd_pcm_wait (playback_handle, 1000)) < 0) {
                                fprintf (stderr, "poll failed (%s)\n", strerror (errno));
                                break;
                        }

                        /* find out how much space is available for playback data */

                        if ((frames_to_deliver = snd_pcm_avail_update (playback_handle)) < 0) {
                                if (frames_to_deliver == -EPIPE) {
                                        fprintf (stderr, "an xrun occured\n");
                                        break;
                                } else {
                                        fprintf (stderr, "unknown ALSA avail update return value (%d)\n",
                                                 frames_to_deliver);
                                        break;
                                }
                        }

                        frames_to_deliver = frames_to_deliver > 4096 ? 4096 : frames_to_deliver;

                        /* deliver the data */

                        if (playback_callback (frames_to_deliver) != frames_to_deliver) {
                                fprintf (stderr, "playback callback failed\n");
                                break;
                        }
                }

                snd_pcm_close (playback_handle);
                exit (0);
        }
```

A Minimal Full-Duplex Program

Full duplex can be implemented by combining the playback and capture designs show above. Although many existing Linux audio applications use this kind of design, in this author's opinion, it is deeply flawed. The the interrupt-driven example represents a fundamentally better design for many situations. It is, however, rather complex to extend to full duplex. This is why I suggest you forget about all of this.

Terminology

capture
: Receiving data from the outside world (different from "recording" which implies storing that data somewhere, and is not part of ALSA's API)

playback
: Delivering data to the outside world, presumably, though not necessarily, so that it can be heard.

duplex
: A situation where capture and playback are occuring on the same interface at the same time.

xrun
: Once the audio interface starts running, it continues to do until told to stop. It will be generating data for computer to use and/or sending data from the computer to the outside world. For various reasons, your program may not keep up with it. For playback, this can lead to a situation where the interface needs new data from the computer, but it isn't there, forcing it use old data left in the hardware buffer. This is called an "underrun". For capture, the interface may have data to deliver to the computer, but nowhere to store it, so it has to overwrite part of the hardware buffer that contains data the computer has not received. This is called an "overrun". For simplicity, we use the generic term "xrun" to refer to either of these conditions

PCM
: Pulse Code Modulation. This phrase (and acronym) describes one method of representing an analog signal in digital form. Its the method used by almost computer audio interfaces, and it is used in the ALSA API as a shorthand for "audio".

channel

frame
: A sample is a single value that describes the amplitude of the audio signal at a single point in time, on a *single channel*. When we talk about working with digital audio, we often want to talk about the data that represents all channels at a single point in time. This is a collection of samples, one per channel, and is generally called a "frame". When we talk about the passage of time in terms of frames, its roughly equivalent to what people when they measure in terms of samples, but is more accurate; more importantly, when we're talking about the amount of data needed to represent all the channels at a point in time, its the only unit that makes sense. Almost every ALSA Audio API function uses frames as its unit of measurement for data quantities.

interleaved
: a data layout arrangement where the samples of each channel that will be played at the same time follow each other sequentially. See "non-interleaved"

non-interleaved
: a data layout where the samples for a single channel follow each other sequentially; samples for another channel are either in another buffer or another part of this buffer. Contrast with "interleaved"

sample clock
: a timing source that is used to mark the times at which a sample should be delivered and/or received to/from the outside world. Some audio interfaces allow you to use an external sample clock, either a "word clock" signal (typically used in many studios), or "autosync" which uses a clock signal present in incoming digital data. All audio interfaces have at least one sample clock source that is present on the interface itself, typically a small crystal clock. Some interfaces do not allow the rate of the clock to be varied, and some have clocks that do not actually run at precisely the rates you would expect (44.1kHz, etc). No two sample clocks can ever be expected to run at precisely the same rate - if you need two sample streams to remain synchronized with each other, they *MUST* be run from the same sample clock.

How to do it ...

Opening the device

ALSA separates capture and playback ....

Setting parameters

We mentioned above that there are number of parameters that need to be set in order for an audio interface to do its job. However, because your program will not actually be interacting with the hardware directly, but instead with a device driver that controls the hardware, there are really two distinct sets of parameters:

Hardware Parameters

These are parameters that directly affect the hardware of the audio interface.

Sample rate
: This controls the rate at which either A/D/D/A conversion is done, if the interface has analog I/O. For fully digital interfaces, it controls the speed of the clock used to move digital audio data to/from the outside world. On some interfaces, other device-specific configuration may mean that your program cannot control this value (for example, when the interface has been told to use an external word clock source to determine the sample rate).

Sample format
: This controls the sample format used to transfer data to and from the interface. It may or may not correspond with a format directly supported by the hardware.

Number of channels
: Hopefully, this is fairly self-explanatory.

Data access and layout
: This controls the way that the program will deliver/receive data from the interface. There are two parameters controlled by 4 possible settings. One parameter is whether or not a "read/write" model will be used, in which explicit function calls are used to transfer data. The other option here is to use "mmap mode" in which data is transferred by copying between areas of memory, and API calls are only necessary to note when it has started and finished.

  The other parameter is whether the data layout will be interleaved or non-interleaved.

Interrupt interval
: This determines how many interrupts the interface will generate per complete traversal of its hardware buffer. It can be set either by specifying a number of periods, of the size of a period. Since this determines the number of frames of space/data that have to accumulate before the interface will interrupt the computer. It is central in controlling latency.

Buffer size
: This determines how large the hardware buffer is. It can be specified in units of time or frames.

Software Parameters

These are parameters that control the operation of the device driver rather than the hardware itself. Most programs that use the ALSA Audio API will not need to set any of these; a few will need set a subset of them.

When to start the device
: When you open the audio interface, ALSA ensures that it is not active - no data is being moved to or from its external connectors. Presumably, at some point you want this data transfer to begin. There are several options for how to make this happen.

  The control point here the start threshold, which defines the number of frames of space/data necessary to start the device automatically. If set to some value other than zero for playback, it is necessary to prefill the playback buffer before the device will start. If set to zero, the first data written to the device (or first attempt to read from a capture stream) will start the device.

  You can also start the device explicitly using `snd_pcm_start`, but this requires buffer prefilling in the case of the playback stream. If you attempt to start the stream without doing this, you will get -EPIPE as a return code, indicating that there is no data waiting to deliver to the playback hardware buffer.

What to do about xruns
: If an xrun occurs, the device driver may, if requested, take certain steps to handle it. Options include stopping the device, or silencing all or part of the hardware buffer used for playback.

  the stop threshold
  : if the number of frames of data/space available meets or exceeds this value, the driver will stop the interface.

  the silence threshold

if the number of frames of space available for a playback stream meets or exceeds this value, the driver will fill part of the playback hardware buffer with silence.

silence size
: when the silence threshold level is met, this determines how many frames of silence are written into the playback hardware buffer

Available minimum space/data for wakeup
: Programs that use `poll(2)` or `select(2)` to determine when audio data may be transferred to/from the interface may set this to control at what point relative to the state of the hardware buffer, they wish to be woken up.

Transfer chunk size
: this determines the number of frames used when transferring data to/from the device hardware buffer.

There are a couple of other software parameters but they need not concern us here.

Receiving and delivering data

NOT YET WRITTEN

Why you might want to forget about all of this

In a word: JACK.