LF *live*

MENTORSHIP SERIES

LF live MENTORSHIP SERIES

ALSA: Writing the soundcard driver

Ivan Orlov, SW Engineer, Codethink

LF live MENTORSHIP SERIES

Why does this talk exist:

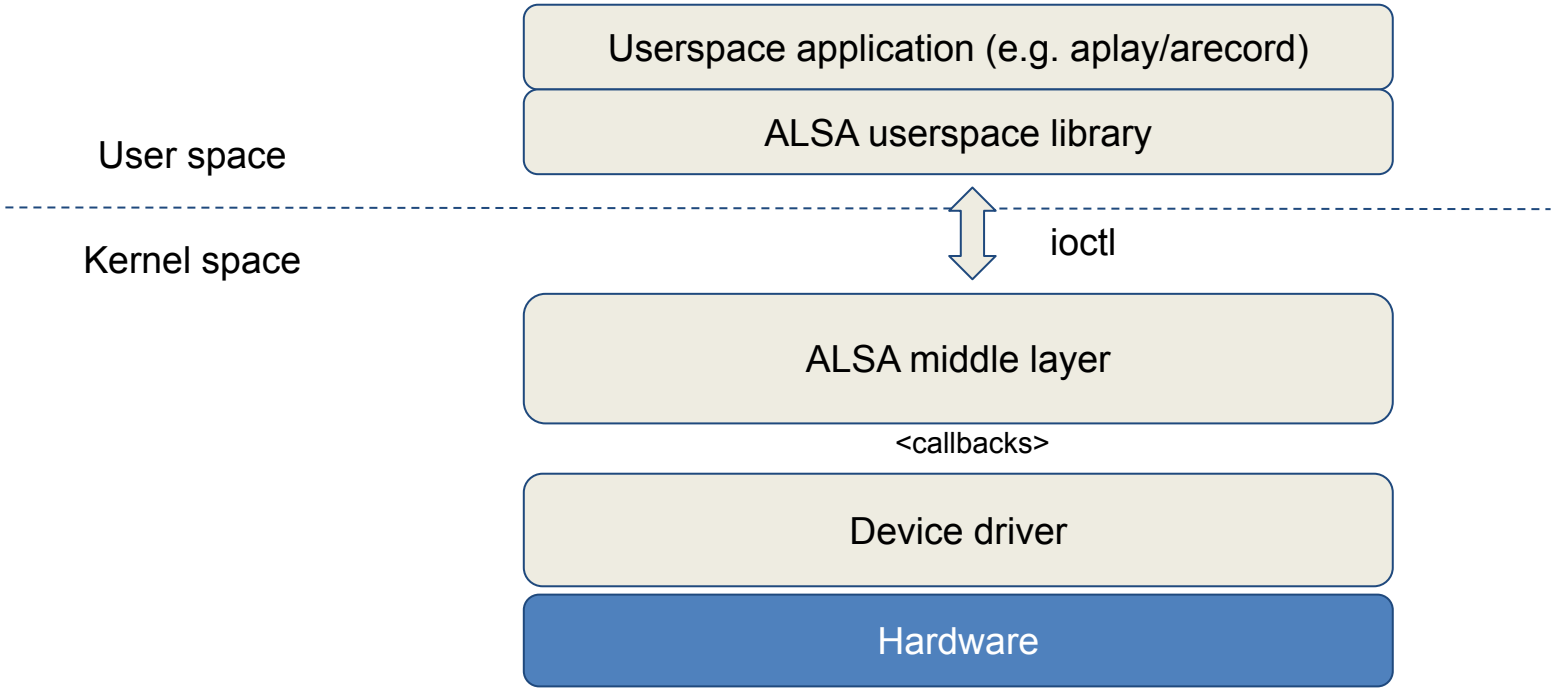
- There are not many talks covering the details of the sound drivers development yet
- Some aspects of the subsystem are poorly documented
- Bring more talents in it!

Approximate plan:

- Basic structure of ALSA
- Structure of a sound card driver: components (PCM, Controls, Timers), their initialization and use
- What are PCM devices, their role in the sound subsystem
- Basic terms: what do you need to know to understand the sound code
- What happens in the ALSA middle layer when you want to capture or play some sound
- XRUNs: our enemies N1
- Debugging approaches
- A few words about Virtual PCM Driver and why developing virtual drivers is important

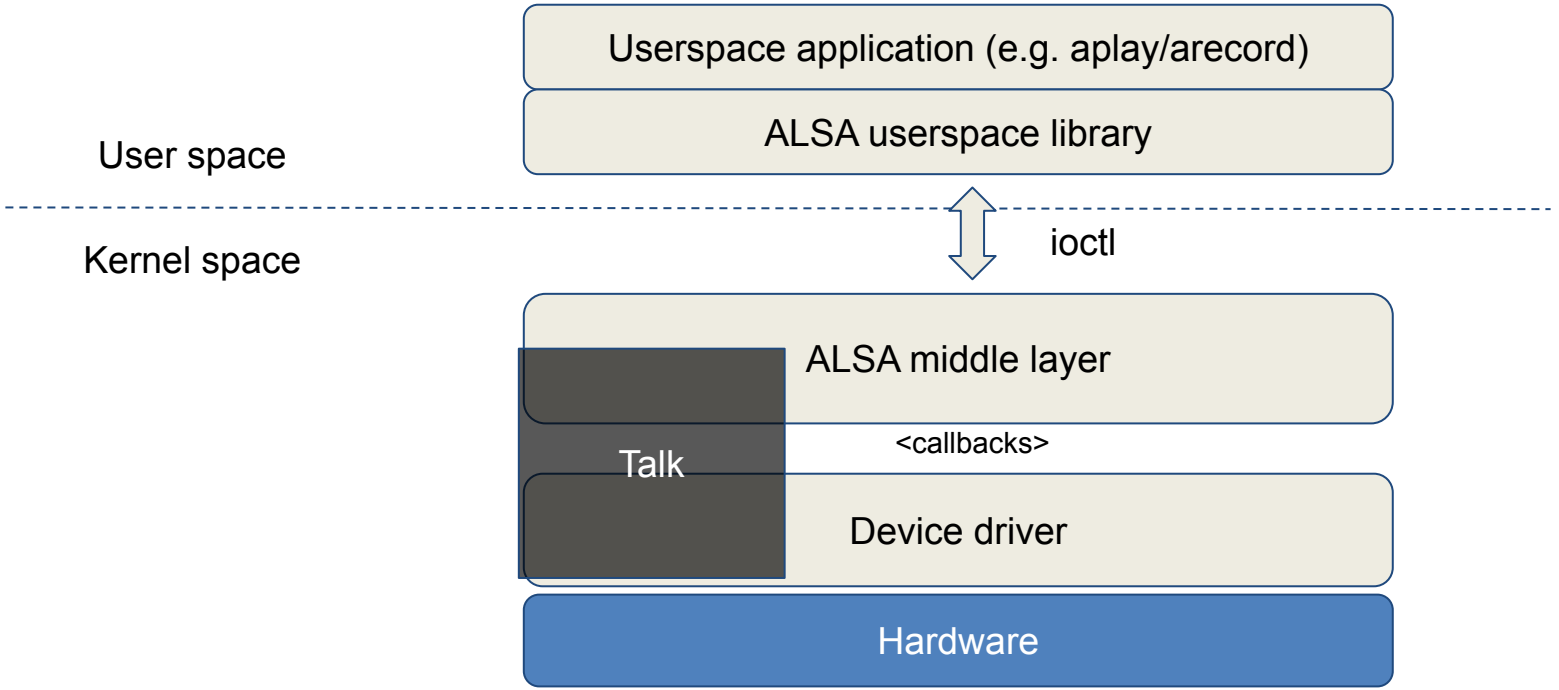
LF Live MENTORSHIP SERIES

ALSA (Advanced Linux Sound Architecture)



LF Live MENTORSHIP SERIES

ALSA (Advanced Linux Sound Architecture)

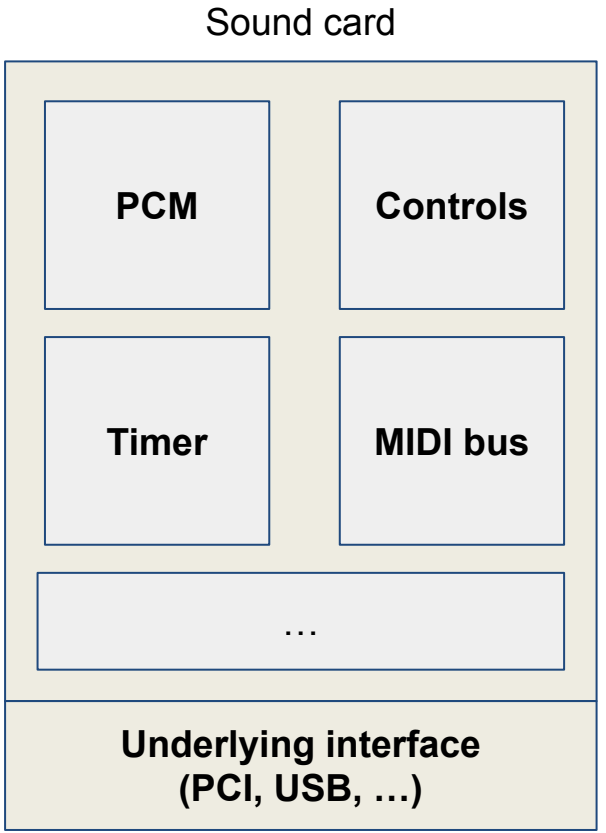


Soundcard driver structure

struct: snd_card

We could “split” the sound card into a set of components:

- **Controls (volume, distortion, mixer controls, …)**
- **PCM**
- **Timers**
- MIDI
- … (we can define our own components)



How to initialize the soundcard?

In probe:

```
snd[_devm]_card_new(&parent_dev, index, id, THIS_MODULE,
sizeof(chip-specific struct), &chip_specific_struct);

strcpy(card->driver, "My Chip");
strcpy(card->shortname, "My Own Chip 123");
sprintf(card->longname, "%s at 0x%lx irq %i", card->shortname, chip->port,
chip->irq);
…
<initialize the components of the card>
…
snd_card_register(card);
```

=> New entry in /proc/asound/

How to view sound cards in your system?

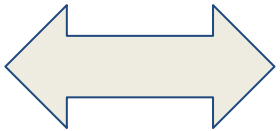
$ **aplay -l**
…
$ **arecord -l**
…
$ **cat /proc/asound/cards**

```
 0 [sofhdadsp      ]: sof-hda-dsp - sof-hda-dsp
                      …
 1 [<card id>      ]: <Driver name> - <Short name>
                      <Long name>
 3 [Loopback       ]: Loopback - Loopback
                      Loopback 1
```

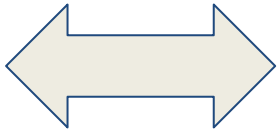
LF live MENTORSHIP SERIES

Soundcard components

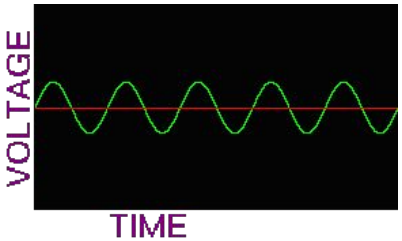
PCM (Pulse Code Modulation) devices

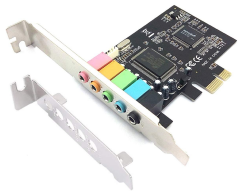


PCM

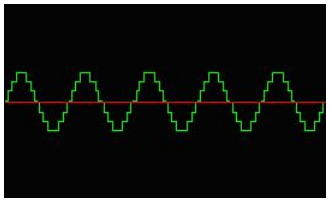


0x00, 0x0A,
0x0B, …





ADC, DAC, …



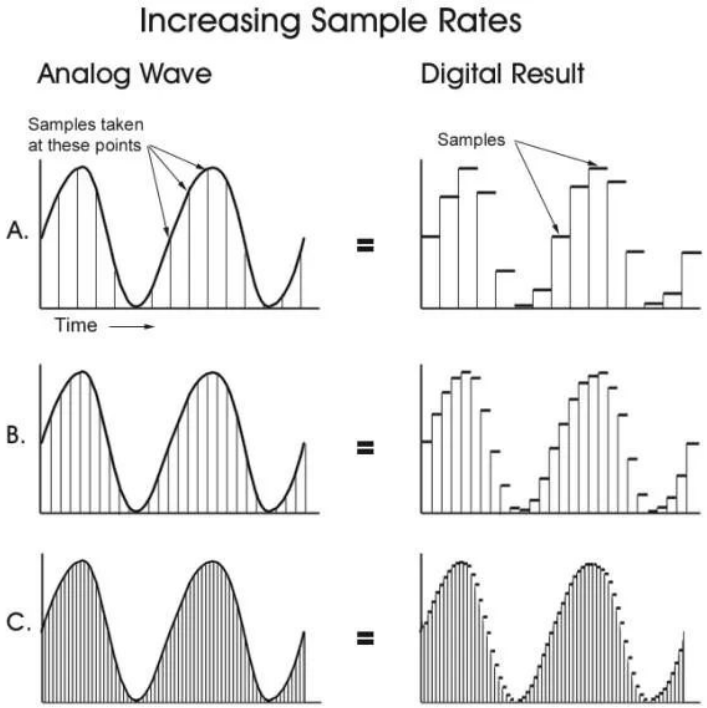
PCM. Sampling

- When playback, consumes volume values (samples)
- When capture, produces volume values (samples)

The amount of samples consumed or produced per second is called **rate** (Hz).
Frame - group of samples for each channel at one moment of time.

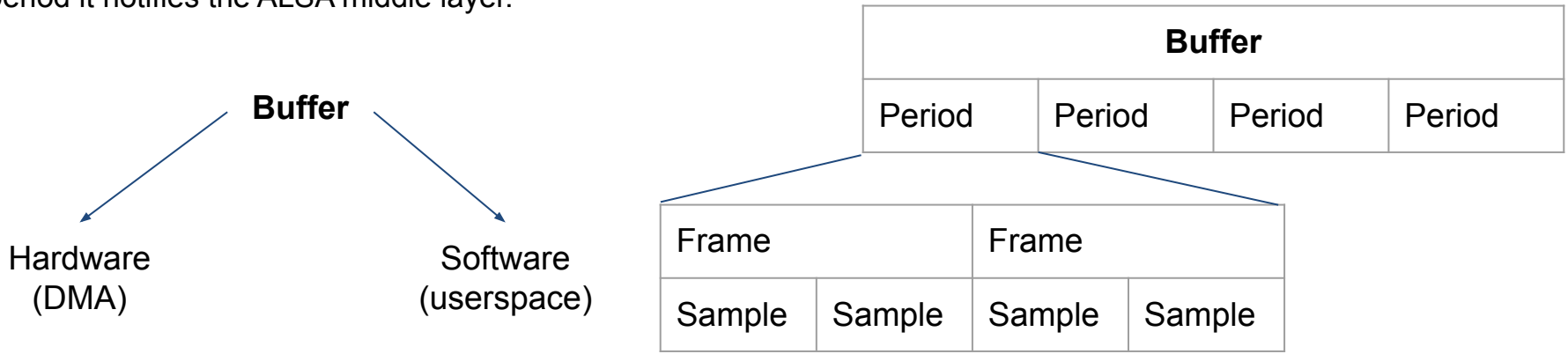
The higher rate => the higher resolution
Rate = 8000 (8kHz) => 8000 measurements per second

Frame (4 bytes)	
Sample (2 byte)	Sample (2 byte)



Periods and buffers

HW Buffer (DMA) contains frames which are processed by hardware.
SW Buffer (Userspace memory) contains frames processed by userspace application. To make the data transfer fast we copy the data from one buffer to another by small chunks (periods). Every time the hardware processes a period it notifies the ALSA middle layer.



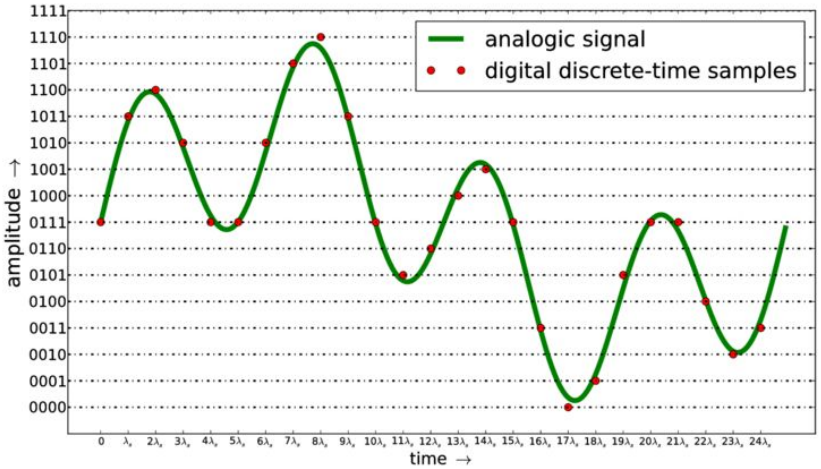
Samples format

Describes the format of volume **sample**. Examples:

- SND_PCM_FORMAT_S16_LE
- SND_PCM_FORMAT_U8
- …

Type in the kernel: snd_pcm_format_t

Quality of sound depends on the format as well!



Access mode

snd_pcm_access_t

The order of samples in the hardware buffer supported by hardware.

INTERLEAVED	S1	S2	S1	S2	…
NON-INTERLEAVED	S1	S1	…	S2	S2
COMPLEX	??	??	??	??	…

mmaped / non-mmaped

PCM & Substreams

Struct name: snd_pcm

- Can have multiple substreams (subdevices)
- Has callbacks for different events

Substreams are streams which could be mixed together by the hardware(so you can play/capture multiple streams simultaneously), but it has very small use nowadays (usually you will find only 1 substream for play and 1 for capture).

Substreams have a *direction* (playback/capture)

Struct name: snd_pcm_substream

PCM			
Substream #0 capture	Substream #1 capture	Substream #2 playback	…

Sound card		
PCM#0	PCM1	…

How to initialize the PCM?

```
snd_pcm_new(card, “ID string”, id, PLAYBACK_CNT, CAPTURE_CNT, &pcm);
```

View PCMs in your system

- aplay -l / arecord -l
- cat /proc/asound/pcm
- ls /proc/asound/card0/pcm*

PCM runtime

Struct: *snd_pcm_runtime*

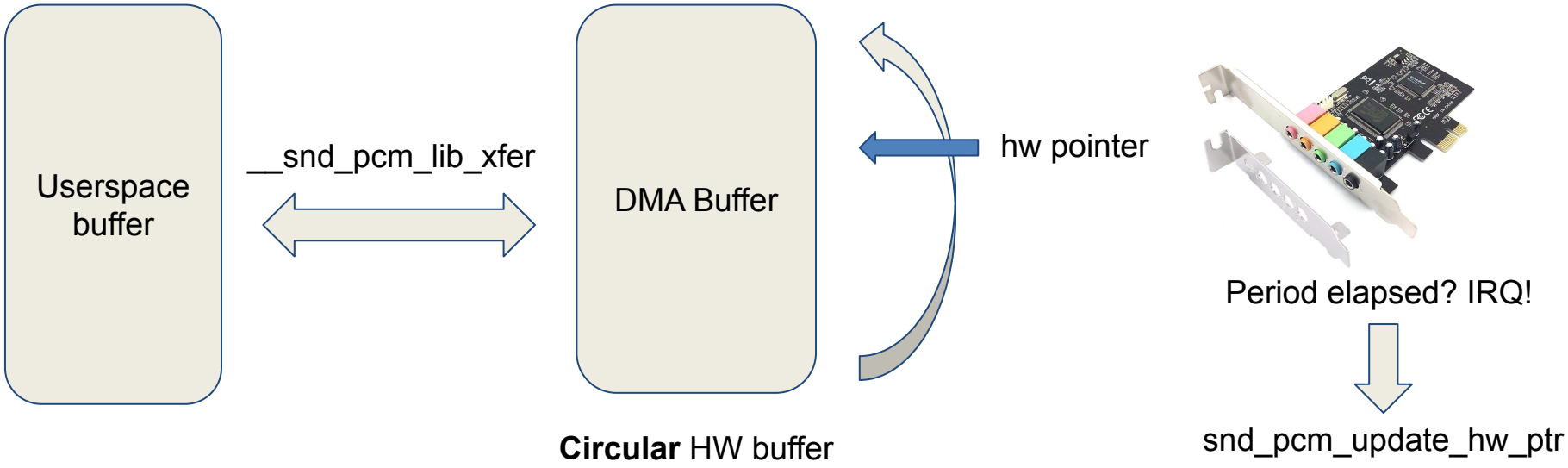
When we open the PCM device for capture/playback, the PCM layer:

1) Checks if we have a free substream on the PCM device. If we don't, returns an error
2) Takes the free substream, creates the *snd_pcm_runtime* struct and assigns it to the substream.

Runtime struct stores the information about particular capture/playback process: buffer pointers, configuration, spinlocks, …

PCM		
Substream #0	Substream #1	…
Runtime #1 DMA Addr, HW params, SW params, Status …	***Free***	

Application <-> Sound hardware



(Example) Reading process

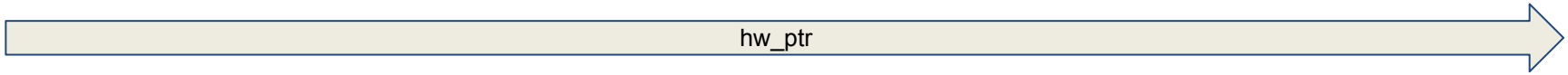
ALSA middle layer	Soundcard driver
When the application calls ‘readi’, alsa layer: 1) ALSA userspace lib calls the SNDRV_PCM_IOCTL_READI_FRAMES ioctl 2) It goes to the __snd_pcm_lib_xfer function, passes the amount of bytes to read 3) Checks the **available frames count**, reads the available frames 4) If there is no available frames, sleeps until they are available (if non-blocking flag isn’t set) 5) Copies the data from the DMA buffer to the userspace buffer, updates the appl_ptr	Every time we have a period-elapsed IRQ, our driver calls the snd_pcm_period_elapsed function, which triggers the ALSA layer to call the ‘pointer’ callback of the driver and update the hw_ptr variable.

hw_ptr & appl_ptr

snd_pcm_update_hw_ptr

__snd_pcm_lib_xfer

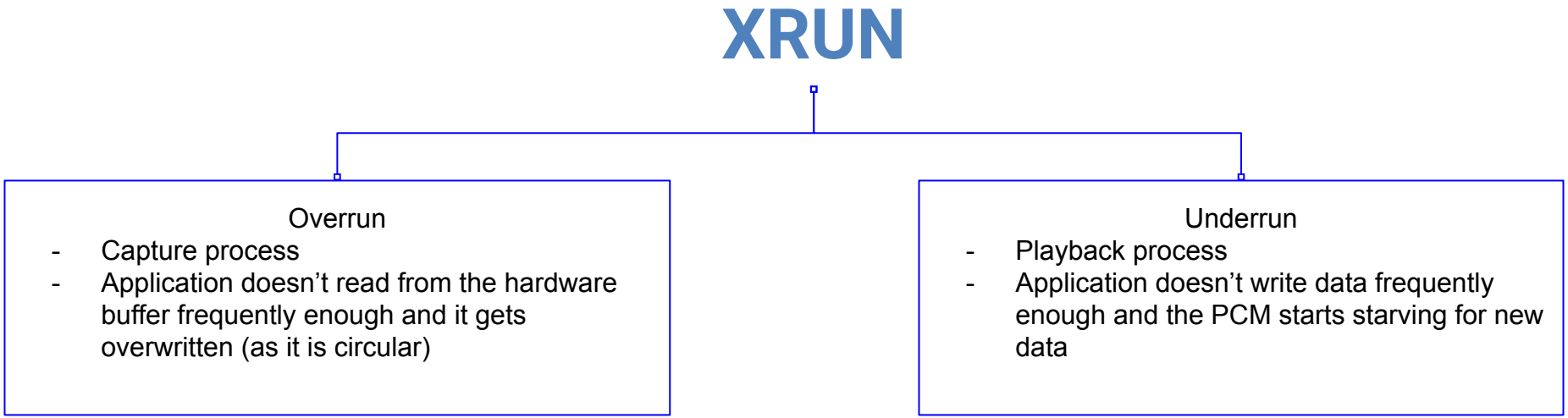
Capture



available frames

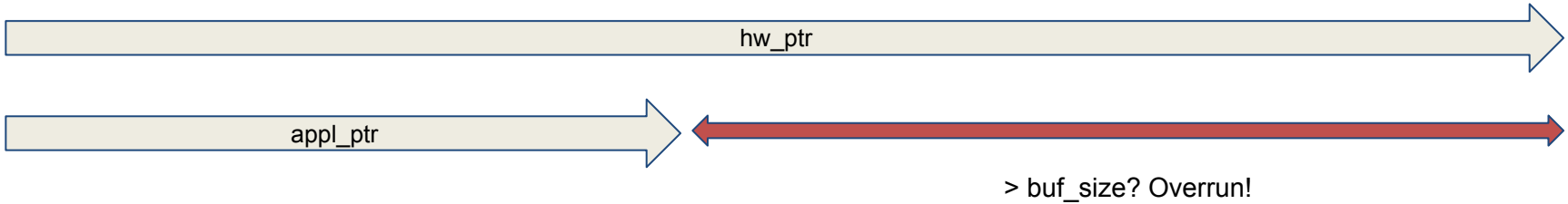
Playback

XRUN

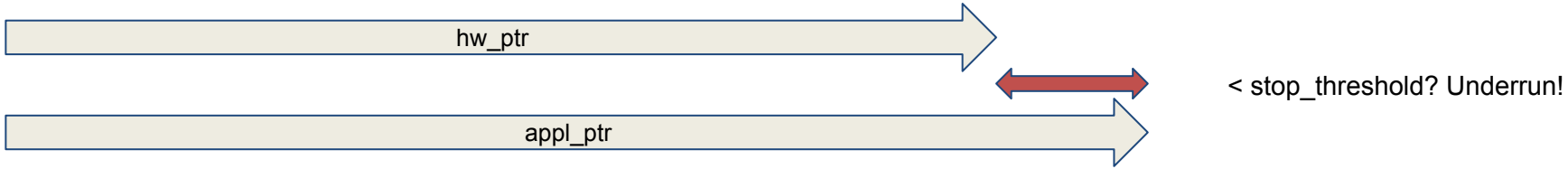

XRUN
Overrun
- Capture process
- Application doesn’t read from the hardware buffer frequently enough and it gets overwritten (as it is circular)
Underrun
- Playback process
- Application doesn’t write data frequently enough and the PCM starts starving for new data

hw_ptr & appl_ptr

Capture



Playback



LF live MENTORSHIP SERIES

How exactly the ALSA middle layer communicates with a PCM?

How exactly the ALSA middle layer communicates with a PCM?

Callbacks

snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &ops);

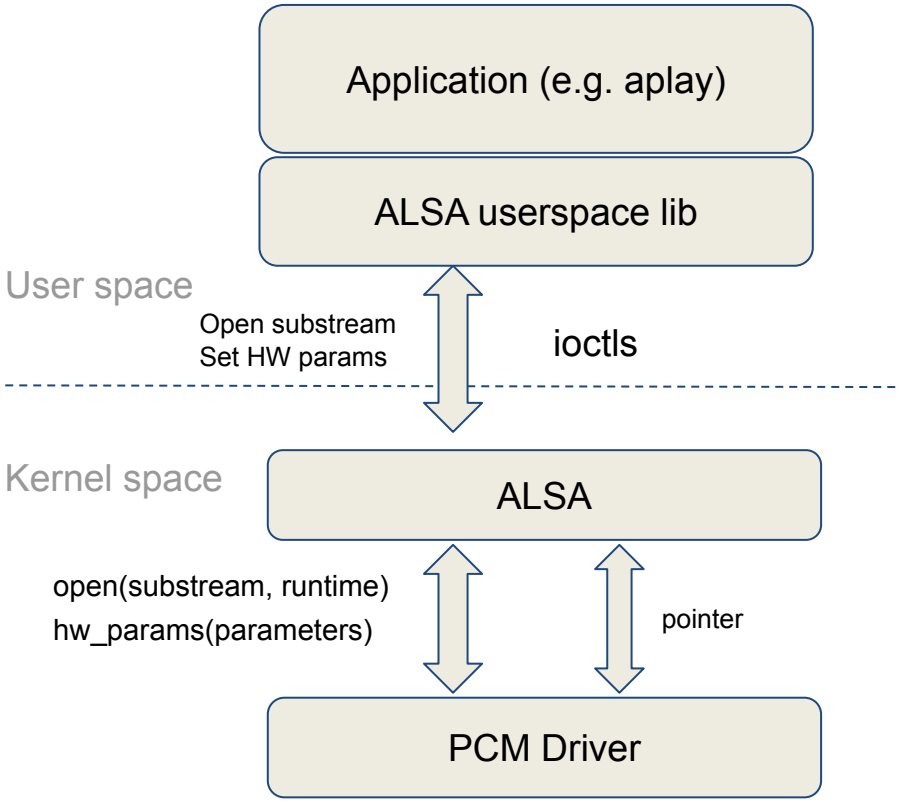
```
struct snd_pcm_ops {
        int (*open)(struct snd_pcm_substream *substream);
        int (*close)(struct snd_pcm_substream *substream);
        int (*ioctl)(struct snd_pcm_substream * substream,
                     unsigned int cmd, void *arg);
        int (*hw_params)(struct snd_pcm_substream *substream,
                         struct snd_pcm_hw_params *params);
        int (*hw_free)(struct snd_pcm_substream *substream);
        int (*prepare)(struct snd_pcm_substream *substream);
        int (*trigger)(struct snd_pcm_substream *substream, int cmd);
        int (*sync_stop)(struct snd_pcm_substream *substream);
        snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream);
        int (*get_time_info)(struct snd_pcm_substream *substream,
                        struct timespec64 *system_ts, struct timespec64 *audio_ts,
                        struct snd_pcm_audio_tstamp_config *audio_tstamp_config,
                        struct snd_pcm_audio_tstamp_report *audio_tstamp_report);
        int (*fill_silence)(struct snd_pcm_substream *substream, int channel,
                            unsigned long pos, unsigned long bytes);
        int (*copy)(struct snd_pcm_substream *substream, int channel,
                    unsigned long pos, struct iov_iter *iter, unsigned long bytes);
        struct page *(*page)(struct snd_pcm_substream *substream,
                             unsigned long offset);
        int (*mmap)(struct snd_pcm_substream *substream, struct vm_area_struct *vma);
        int (*ack)(struct snd_pcm_substream *substream);
};
```

PCM callbacks

Struct: snd_pcm_ops

PCM has a variety of callbacks to communicate with the “upper” layers. There are some standard implementations of the callbacks, so usually we have to define only part of them when writing our driver:

- open
- close
- pointer
- hw_params
- trigger
- prepare



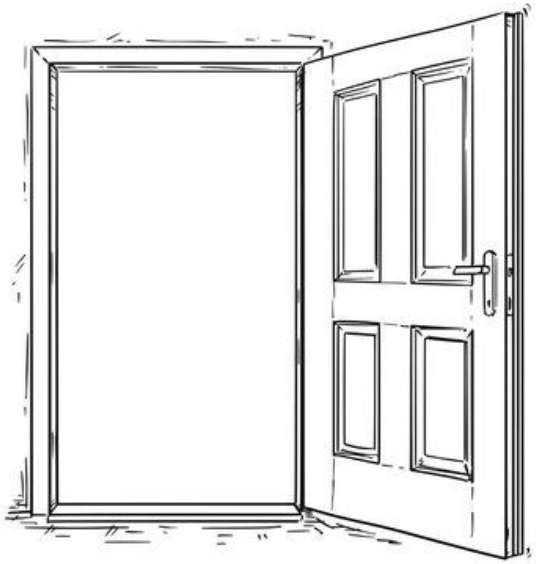
PCM callbacks. ‘Open’ callback

Is called when the substream is opened for capture/playback.

Prototype: int (*open)(struct snd_pcm_substream *substream);
Non-atomic

Purpose: Set runtime’s hardware description*, private data allocation, setting up the **constraints**.

* *struct snd_pcm_hardware - contains accepted rates, count of channels, period counts and sizes, maximum buffer size*



LF live MENTORSHIP SERIES

PCM callbacks. ‘Close’ callback

Is called when the substream is closed.

Prototype: int (*close)(struct snd_pcm_substream *substream);
Non-atomic

Purpose: Free runtime private data

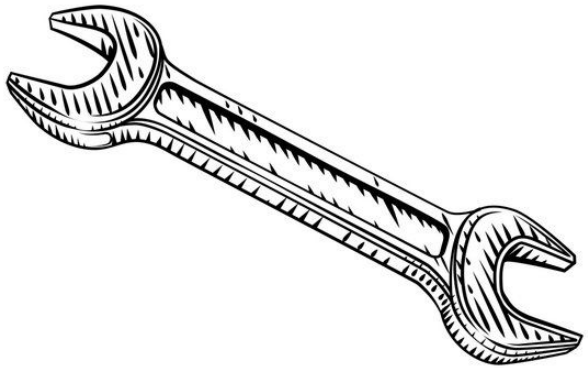
‘hw_params’ and ‘hw_free’ callbacks

‘hw_params’ is called when the application sets the hardware settings (buffer size, period size, format, …)

Prototype: int (*hw_params)(struct snd_pcm_substream *substream, struct snd_pcm_hw_params *params);
Non-atomic and could be called multiple times!

Purpose: hardware setup, unmanaged buffer allocation

‘hw_free’ is called just before ‘close’ and allows us to free all of the allocated resources. Could be called multiple times!



Memory allocation

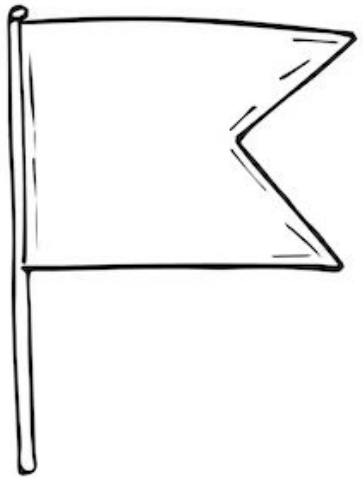
Unmanaged	Managed
You still have to “pre-allocate” pages: *snd_pcm_lib_preallocate_pages_for_all* (usually done at initialization, for instance in probe) And allocate/free pages every time in the hw_params and hw_free callbacks	Call once: *snd_pcm_set_managed_buffer_all* during the initialization. No need to allocate/free pages in callbacks.

PCM callbacks. ‘prepare’ callback

Is called every time snd_pcm_prepare is called (for instance, after overruns/underruns). We can set some of the hw params here (as rate, format, …).

Prototype: int (*prepare)(struct snd_pcm_substream *substream);

Non-atomic, can be called multiple times!



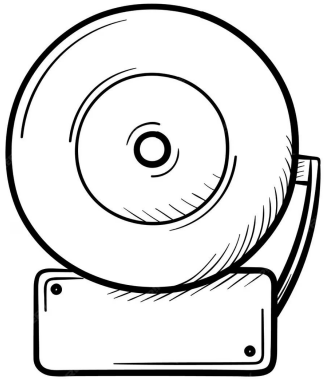
PCM callbacks. ‘trigger’ callback

Is called every time the PCM is started, stopped, paused, resumed or suspended.
Events:

- SNDRV_PCM_TRIGGER_STOP
- SNDRV_PCM_TRIGGER_START
- SNDRV_PCM_TRIGGER_PAUSE_PUSH
- SNDRV_PCM_TRIGGER_PAUSE_RELEASE
- SNDRV_PCM_TRIGGER_SUSPEND
- SNDRV_PCM_TRIGGER_RESUME
- SNDRV_PCM_TRIGGER_DRAIN

Prototype: int (*trigger)(struct snd_pcm_substream *substream,
int cmd);

Atomic!



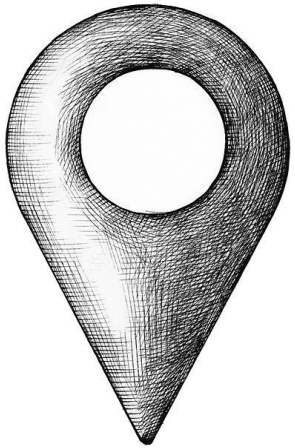
PCM callbacks. ‘pointer’ callback

This callback is used by PCM core to get the hardware buffer pointer. Returns value **in frames!**

Prototype: snd_pcm_uframes_t (*pointer)(struct snd_pcm_substream *substream)

Atomic

(Usually we read the hardware register here)



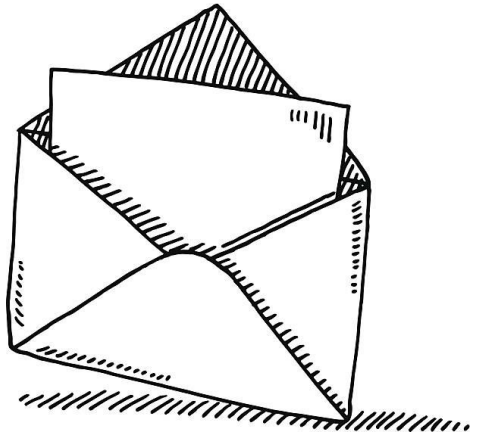
PCM callbacks. ‘ioctl’ callback

Allows redefinition of **some** of the PCM **read** ioctls:

- SNDRV_PCM_IOCTL1_FIFO_SIZE
- SNDRV_PCM_IOCTL1_CHANNEL_INFO
- SNDRV_PCM_IOCTL1_RESET

Prototype: int (*ioctl)(struct snd_pcm_substream * substream,
unsigned int cmd, void *arg);

Usually, this callback is undefined as the ALSA middle layer provides the generic implementation for all of the possible ioctls



How to test the PCM?

- arecord:
  $ arecord -D hw:CARD=0,DEV=0 -c 4 -i -f S16_LE -r 48000 --duration=10 out.wav
- aplay:
  $ aplay -D hw:CARD=0,DEV=0 -c 4 -f S16_LE -r 48000 src.wav

Both of them support different formats, rates, channel count and access modes.
For recording/playing non-interleaved sound:

- *$ arecord -D hw:CARD=0,DEV=0 -c 4 -i -f S16_LE -r 48000 -I --duration=10 out.wav*

LF Live MENTORSHIP SERIES

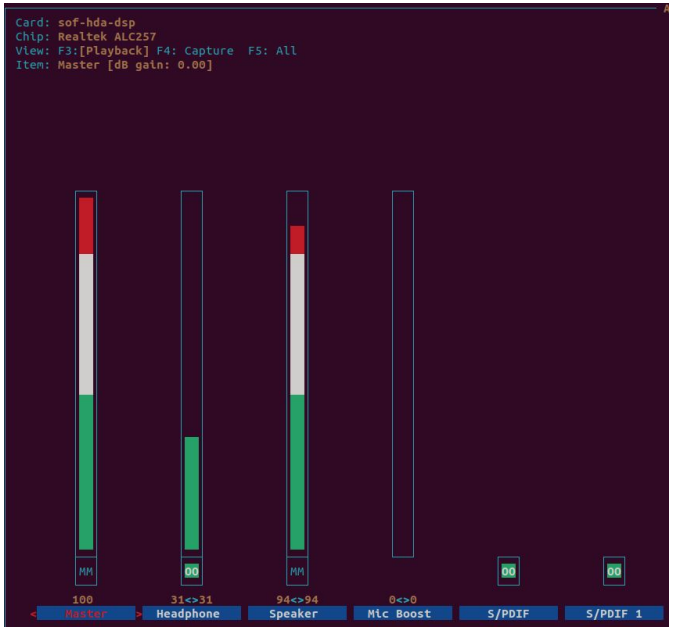
Questions?

Controls

struct: snd_kcontrol_new
Controls are abstractions for control elements of the soundcard.
Types:

- Integer (min, max, and step could be set)
- Switch (boolean)
- Enumeration

Controls **should** follow the naming convention, but it is not obligatory, for instance:
PCM Playback Volume
<Source> <Direction> <Function>



How to define control?

```
static struct snd_kcontrol_new my_control = {
        .iface = SNDRV_CTL_ELEM_IFACE_MIXER,
        .name = "PCM Playback Switch",
        .index = 0,
        .access = SNDRV_CTL_ELEM_ACCESS_READWRITE,
        .private_value = 0xffff,
        .info = my_control_info,
        .get = my_control_get,
        .put = my_control_put
};
```

```
snd_kcontrol *ctl = snd_ctl_new1(&my_control, private_data)

snd_ctl_add(card, ctl);
```

Info callback

```
static int snd_myctl_enum_info(struct snd_kcontrol *kcontrol,
                               struct snd_ctl_elem_info *uinfo)
{
        static char *texts[4] = {
                "First", "Second", "Third", "Fourth"
        };
        uinfo->type = SNDRV_CTL_ELEM_TYPE_ENUMERATED;
        uinfo->count = 1;
        uinfo->value.enumerated.items = 4;
        if (uinfo->value.enumerated.item > 3)
                uinfo->value.enumerated.item = 3;
        strcpy(uinfo->value.enumerated.name,
               texts[uinfo->value.enumerated.item]);
        return 0;
}
```

```
static int snd_myctl_mono_info(struct snd_kcontrol *kcontrol,
                               struct snd_ctl_elem_info *uinfo)
{
        uinfo->type = SNDRV_CTL_ELEM_TYPE_BOOLEAN;
        uinfo->count = 1;
        uinfo->value.integer.min = 0;
        uinfo->value.integer.max = 1;
        return 0;
}
```

Get callback

```
static int snd_myctl_get(struct snd_kcontrol *kcontrol,
                         struct snd_ctl_elem_value *ucontrol)
{
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        ucontrol->value.integer.value[0] = get_some_value(chip);
        return 0;
}
```

```
static int snd_myctl_get_enumerated(struct snd_kcontrol *kcontrol,
                         struct snd_ctl_elem_value *ucontrol)
{
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        ucontrol->value.enumerated.item[0] = get_some_value(chip);
        return 0;
}
```

```
struct snd_ctl_elem_value {
        struct snd_ctl_elem_id id;      /* W: ele
        unsigned int indirect: 1;       /* W: ind
        union {
                union {
                        long value[128];
                        long *value_ptr;        /
                } integer;
                union {
                        long long value[64];
                        long long *value_ptr;   /
                } integer64;
                union {
                        unsigned int item[128];
                        unsigned int *item_ptr; /
                } enumerated;
                union {
                        unsigned char data[512];
                        unsigned char *data_ptr;
                } bytes;
                struct snd_aes_iec958 iec958;
        } value;                /* RO */
        unsigned char reserved[128];
};
```

Put callback

```
static int snd_myctl_put(struct snd_kcontrol *kcontrol,
                         struct snd_ctl_elem_value *ucontrol)
{
        struct mychip *chip = snd_kcontrol_chip(kcontrol);
        int changed = 0;
        if (chip->current_value !=
            ucontrol->value.integer.value[0]) {
                change_current_value(chip,
                                ucontrol->value.integer.value[0]);
                changed = 1;
        }
        return changed;
}
```

Input validation goes here as well.

How to view and manipulate controls?

- Console interface: amixer
  Show all controls: *amixer -c <card_number> controls*
  Show all controls with values: *amixer -c <card_number> contents*

- Pseudo-graphical interface: alsamixer
  Show controls for particular card: alsamixer -c 0

ALSA Timers

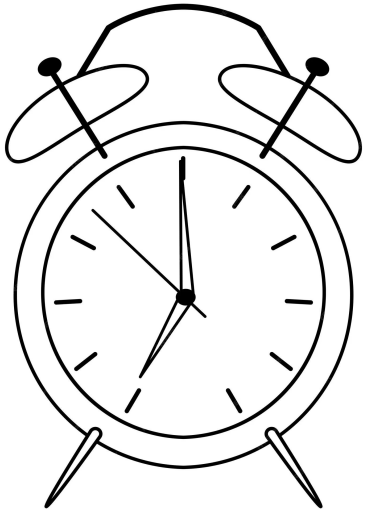
ALSA supports exporting the hardware timer interface, so other kernel modules and the userspace applications could bind to them!

struct: snd_timer

To view available timers on your system:

$ cat /proc/asound/timers

ALSA creates a timer instance for every PCM substream, and triggers it when snd_pcm_period_elapsed is called.



How to define a timer?

```
static struct snd_timer_hardware some_sample_timer_hw = {
        .flags = SNDRV_TIMER_HW_AUTO,
        .resolution = <resolution in nsec>,
        .ticks = <max ticks count which could be set>,
        .start = timer_start_callback,
        .stop = timer_stop_callback,
};
…
struct snd_timer_id tid;
tid.dev_class = SNDRV_TIMER_CLASS_CARD;
tid.dev_sclass = SNDRV_TIMER_SCLASS_NONE;
tid.card = chip->card->number;
tid.device = device;
tid.subdevice = 0;
```

```
snd_timer_new(card, "Timer name", &tid, &timer);
timer->hw = some_sample_timer_hw;
```

Timer will appear in:
/proc/asound/timers

Example: sound/core/hrtimer.c

Timers callbacks. timer_start

*Prototype: int (*start) (struct snd_timer *timer);*

Is called when the userspace application or another kernel module starts the timer instance.

Desired ticks value is set in ‘sticks’ field of the timer struct.

Timers callbacks. timer_stop

*Prototype: int (*stop) (struct snd_timer *timer);*
Is called when the timer instance is stopped.

Bind to timer. Timer instance

struct: snd_timer_instance

If we want to bind to the timer, we have to create a timer instance and define a few callbacks for it.

.callback - the callback which will be called after count of ticks we set.
.ccallback - the callback, which is called with every event happened with the timer

After that, we will be able to start the timer with *snd_timer_start* function.

Example: sound/drivers/aloop.c

Debugging the sound-related issues

- Tracing (ftrace/strace)
- xrun_debug

LF live MENTORSHIP SERIES

Debugging the sound-related issues

- Tracing (ftrace/strace)
- xrun_debug

snd_printk! 😎

Tracing

$ trace-cmd record -l snd_* -p function_graph

…

$ trace-cmd report

When it could be useful: **Timing issues, XRUNs**

$ strace aplay …

When it could be useful: **Device errors, failed alsa-lib ioctls**

CONFIG_FTRACE
CONFIG_DYNAMIC_FTRACE

xrun_debug

1 Basic debugging - show xruns in ksyslog interface
2 Dump stack - dump stack for basic debugging
4 Jiffies check - compare the position with kernel jiffies (a sort of in-kernel monotonic clock), show what's changed when basic debugging is enabled
8 Dump positions on each period update call
16 Dump positions on each hardware pointer update call
32 Enable logging of last 10 ring buffer positions
64 Show the last 10 ring buffer position only once (when first error situation occured)

CONFIG_SND_PCM_XRUN_DEBUG
CONFIG_SND_VERBOSE_PROCFS
CONFIG_SND_DEBUG

```
$ echo 3 > /proc/asound/card0/pcm0c/xrun_debug
```

https://www.alsa-project.org/wiki/XRUN_Debug

ALSA selftests

- Help to detect timing issues on different frame rates, buffer and period sizes
- Can test mixer controls as well

snd-aloop driver

- Creates a pair of software loopback PCM devices
- Can be used as a sample device when investigating the timing issues

snd_printk and friends

snd_printk, pcm_dbg, pcm_err, pcm_warn, …

CONFIG_SND_DEBUG

$ dmesg -n 8

When to use: ~~**ALWAYS**~~

Virtual PCM driver (snd-pcmtest)

Why?

- We need to cover the ALSA layer with tests (more tests less bugs!)

What can it do?

- Sample virtual sound driver (less complicated than snd-aloop)
- Can generate template-based or random capturing data which could be validated from the userspace
- Can check the playback data for containing the expected pattern
- Can inject errors and delays into the PCM callbacks and capture/playback processes to test the userspace application behavior

Documentation: https://docs.kernel.org/sound/cards/pcmtest.html
Tests: tools/testing/selftests/alsa/test-pcmtest-driver.c

LF Live MENTORSHIP SERIES

Useful resources

- Writing an ALSA driver: https://docs.kernel.org/sound/kernel-api/writing-an-alsa-driver.html
- Introduction to sound programming with ALSA: https://www.linuxjournal.com/article/6735
- Sources of the snd-aloop and snd-pcmtest drivers
- ALSA sources

Sources for the webinar

LF live MENTORSHIP SERIES

Thank you for your attention! Questions?

LF live MENTORSHIP SERIES

Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.