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

http://imi.aau.dk/~sd/phd/index.php?title=Minivosc
http://www.alsa-project.org/main/index.php/Minivosc

S. Dimitrov          S. Serafin
Minivosc - a minimal virtual oscillator driver for ALSA

Introduction - links

- Minivosc is a driver, and a corresponding tutorial (and paper):
  - http://www.alsa-project.org/main/index.php/Minivosc (on ALSA project Wiki)
    - (need syncing + paper link)
Introduction – name and properties

• What's in a name?
  • Minivosc stands for *minimal virtual oscillator*

• What is it?
  • An example of a *capture-only, 8-bit, 8 kHz* driver
  • Written with the intent of being the simplest ALSA driver for study
  • Does not require any actual soundcard hardware
Focus in driver development

- Role of a driver – provide users with a simple *(high-level)* interface to peripheral hardware, in a PC OS
  - What are these high-level actions afforded to a user?

- Two aspects are most important in *(low-level)* understanding of drivers from the PC OS side:
  - How do things happen *(memory-wise)* *(where?)*
  - How do things happen *(time-wise)* *(when?)*
Motivation

• Build a card for the (obsolete) ISA slot
  • Write simple “for” loop in userland C (without any rate/period information)...
  • ... obtain 17 kHz sampling rate ??!
  • Problem – non real-time OS

• Build an FPGA card...
  • Implement a “blinking LED” example without a problem...
  • ... but how to make it play sound ??!

• Need to look at software – drivers !!
Minivosc - a minimal virtual oscillator driver for ALSA

“Chicken-and-egg” problem

driver (software) required to understand (soundcard) hardware ...

(soundcard) hardware required to understand driver (software)...

```c
// note snd_pcm_ops can usually be separate playback control
static struct snd_pcm_ops minivosc_pcm_ops = {
  .open = minivosc_pcm_open,
  .close = minivosc_pcm_close,
  .ioctl = snd_pcm_ioctl,
  .hw_params = minivosc_hw_params,
  .hw_config = minivosc_hw_config,
  .prepare = minivosc_pcm_prepare,
  .trigger = minivosc_pcm_trigger,
  .pointer = minivosc_pcm_pointer,
};

// specifies what func is called @ snd_card_free
// used in snd_device_new
static struct snd_device_ops dev_ops = {
  .dev_free = minivosc_pcm_dev_free,
};
```
Prior related work

• Sources for research and development of minivosc:
  • Takashi Iwai's The ALSA Driver API
    – Documentation
  • Stéphan K.'s HowTo Asynchronous Playback - ALSA wiki
    – Documentation (now offline?)
  • Takashi Iwai's Writing an ALSA Driver
    – Not beginner; undisclosed PCI hardware
  • Ben Collins: Writing an ALSA driver
    – Undisclosed hardware; no memory ops
  • dummy.c driver
    – Virtual driver; no memory ops
  • aloop–kernel.c driver
    – Virtual driver; multichannel
Overview diagram – PC soundcard context
High-level user actions (playback direction)

- Playback direction – from PC to soundcard (speakers)
- User can:
  - Press PLAY (start playback)
  - Press STOP (stop playback)
  - *(user expects to hear sound - card/speakers needed for full user experience!)*
Minivosc - a minimal virtual oscillator driver for ALSA

High-level user actions (capture direction)

- Capture direction – from soundcard (microphone) to PC
- User can:
  - Press RECORD (start capture)
  - Press STOP (stop capture)
  - *(user expects to see recording action - no hardware needed for full user experience!)*

Intermediate buffer/array
(== waveform grain -> oscillator! )

Capture buffer/array
(ALSA does copying)
Initial summary

- Easier to demonstrate **capture** direction in a virtual (no hardware) driver – *while* preserving high-level user expectations (*i.e.* *what happens in audio software*)
- **8 kHz** sampling rate – next lowest possible in ALSA; avoid potential bottleneck problems with fast sampling rates
- **Mono, 8-bit** signal – avoid conceptual complication with ALSA frames:
  - ALSA frame – collection of one *sample* from all channels in a stream
  - With mono, 8-bit: 1 byte ~ 1 sample ~ 1 frame
Linux driver models

- Declaration of driver devices:

- For devices interfacing through the PCI bus:

  ```c
  struct pci_driver my_driver ....
  pci_register_driver(&my_driver) ... // [init]
  ```

- For devices interfacing through the USB bus:

  ```c
  struct usb_driver my_driver ...
  usb_register(&my_driver) ... // [init]
  ```

- For virtual devices (no hardware) – platform model:

  ```c
  struct platform_driver my_driver ...
  platform_driver_register(&my_driver) ... // [init]
  ```
Driver device structure

- Device structure contains references to needed data

```c
struct minivosc_device {
    struct snd_card *card;
    struct snd_pcm *pcm;
    const struct minivosc_pcm_ops *timer_ops;
    /* we have only one substream, so all data in this struct */
    struct mutex cable_lock;
    /* PCM parameters */
    unsigned int pcm_period_size;
    unsigned int pcm_bps; /* bytes per second */
    /* flags */
    unsigned int valid;
    unsigned int running;
    unsigned int period_update_pending : 1;
    /* timer stuff */
    unsigned int irq_pos; /* fractional IRQ position */
    unsigned int period_size_frac;
    unsigned long last_jiffies;
    struct timer_list timer;
    /* copied from struct loopback_pcm: */
    struct snd_pcm_substream *substream;
    unsigned int pcm_buffer_size;
    unsigned int buf_pos; /* position in buffer */
    unsigned int silent_size;
    /* added for waveform: */
    unsigned int wvf_pos; /* position in waveform array */
    unsigned int wvf_lift; /* lift of waveform array */
};
```

References can be established at different stages of driver lifetime!

Should eventually contain a reference to ALSA capture substream buffer/array!
Driver device structure

- Device structure can be difficult to navigate, especially for finding capture buffer/array
- For easier navigation: *partial structure map* diagram

Capture substream buffer/array:
```
minivosc_device->substream->runtime->dma_area
```
Minivosc - a minimal virtual oscillator driver for ALSA

Hardware parameters – sample rate & format

- Definition of possible allowed values – struct `minivosc_pcm_hw`:

```c
#define MAX_BUFFER (32 * 48)
static struct snd_pcm_hardware minivosc_pcm_hw = {
    .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_INTERLEAVED | SNDRV_PCM_INFO_BLOCK_TRANSFER | SNDRV_PCM_INFO_MMAP_VALID),
    .formats = SNDRV_PCM_FMTBIT_U8,
    .rates = SNDRV_PCM_RATE_8000,
    .rate_min = 8000,
    .rate_max = 8000,
    .channels_min = 1,
    .channels_max = 1,
    .buffer_bytes_max = MAX_BUFFER, //(32 * 48) = 1536,
    .period_bytes_min = 48,
    .period_bytes_max = 48,
    .periods_min = 1,
    .periods_max = 32,
};
```

- (Audio software could choose arbitrarily from the allowed values)
Minivosc - a minimal virtual oscillator driver for ALSA

Driver/device initialization functions

- Callbacks that run when device is attached/removed – or when driver is loaded/unloaded
  - Minivosc virtual driver: driver loading ~ device attachment

```c
// * functions for driver/kernel module initialization
static void minivosc_unregister_all(void);
static int __init alsa_card_minivosc_init(void);
static void __exit alsa_card_minivosc_exit(void);

// * declare functions for this struct describing the driver (to be defined later):
static int __devinit minivosc_probe(struct platform_device *devptr);
static int __devexit minivosc_remove(struct platform_device *devptr);

// specifies what func is called @ snd_card_free
// used in snd_device_new
static struct snd_device_ops dev_ops =
{
  .dev_free = minivosc_pcm_dev_free,
};

// ....
// * we need a struct describing the driver:
static struct platform_driver minivosc_driver =
{
  .probe   = minivosc_probe,
  .remove  = __devexit_p(minivosc_remove),
  .driver  = {
    .name = SND_MINIVOSC_DRIVER,
    .owner = THIS_MODULE
  },
};
```
Driver/device initialization functions – exec order

• Execution sequence upon *driver loading*:

  # at insmod:
  [48803.808593] ./minivosc.c: alsa_card_minivosc_init
  [48803.808821] ./minivosc.c: minivosc_probe : probe

• Execution sequence upon *driver unloading*:

  # at rmmod:
  [49005.736089] ./minivosc.c: alsa_card_minivosc_exit
  [49005.736097] ./minivosc.c: minivosc_unregister_all
  [49005.736146] ./minivosc.c: minivosc_remove
  [49005.755433] ./minivosc.c: minivosc_pcm_dev_free
  [49005.755445] ./minivosc.c: minivosc_pcm_free
Digital audio (PCM) Interface functions

- Functions that handle digital audio based on commands from high-level audio software:

```c
// note snd_pcm_ops can usually be separate _playback_ops and _capture_ops
static struct snd_pcm_ops minivosc_pcm_ops = {
    .open = minivosc_pcm_open,
    .close = minivosc_pcm_close,
    .ioctl = snd_pcm_lib_ioctl,
    .hw_params = minivosc_hw_params,
    .hw_free = minivosc_hw_free,
    .prepare = minivosc_pcm_prepare,
    .trigger = minivosc_pcm_trigger,
    .pointer = minivosc_pcm_pointer,
};
```
Digital audio (PCM) Interface functions – exec order

- **Execution sequence upon (a)record start:**

  - [48810.487603] ./minivosc.c: minivosc_pcm_open
  - [48810.488110] ./minivosc.c: minivosc_hw_params
  - [48810.488162] ./minivosc.c: minivosc_pcm_prepare
  - [48810.488170]: bps: 8000; runtime->buffer_size: 1536; mydev->pcm_buffer_size: 1536
  - [48810.488478] ./minivosc.c: minivosc_pcm_trigger - trig 1

- **Execution sequence upon (a)record stop:**

  - [48811.489504] ./minivosc.c: minivosc_pcm_trigger - trig 0
  - [48811.489527] ./minivosc.c: minivosc_hw_free
  - [48811.489588] ./minivosc.c: minivosc_hw_free
  - [48811.489596] ./minivosc.c: minivosc_pcm_close
Populating the device structure

- We need to save references for device structure *ourselves*!

```c
static int __devinit minivosc_probe(struct platform_device *devptr)
{
    struct snd_card *card;
    struct minivosc_device *mydev;
    // ....
    int dev = devptr->id; // from aloop-kernel.c
    // ....
    ret = snd_card_create(index[dev], id[dev],
                          THIS_MODULE, sizeof(struct minivosc_device), &card);
    // ....
    mydev = card->private_data;
    mydev->card = card;
    // ....
    OS kernel/ALSA provides this
    We instantiate using the input argument...
    We save the result in the device structure *ourselves*!
}
```
Populating the device structure

- We need to save references for device structure *ourselves*!

```c
static int minivosc_pcm_open(struct snd_pcm_substream *ss)
{
    struct minivosc_device *mydev = ss->private_data;

    // ...

    ss->runtime->hw = minivosc_pcm_hw;
    mydev->substream = ss;
    ss->runtime->private_data = mydev;

    // ...

    OS kernel/ALSA provides this – _open is the first time the
    substream is defined!

    We assign ourselves...

    We save the references in the device
    structure *ourselves*!

    If we don't save the references to substream here – we will
    not be able to retrieve them, when the time comes to
    handle the capture buffer!
```
The capture process – timing and memory

- Polling or interrupt?
  - There is no actual hardware that can generate interrupts for the PC...
  - ... so we can simulate a polling process by using a *timer function*

- Different Linux kernel timers
  - default, “timer wheel” (jiffies);
  - high-resolution timers.
The capture process – timing and memory

- **Process:**
  - `_pcm_open`: we specify `_timer_function` is our timer function
  - `_pcm_prepare`: buffer positions/sizes are initialized
  - `_pcm_trigger`: here `_timer_start` (or stop) is called
  - `_timer_start`: here timer expiry time is set, and timer is “started” via `add_timer` function

- At this point, the OS kernel/ALSA can arbitrarily call our `_pcm_pointer` function (which then calls `_pos_update`), to find out what are our *current* buffer positions!

- After the timer has expired, `_timer_function` runs;
  - and it also calls `_pos_update`!
  - (additionally, it calls `snd_pcm_period_elapsed` to inform ALSA)
The capture process – timing and memory

• Process (cont.):
  • from _pos_update perspective:
    – If delta jiffies from last _pos_update is zero; then we've been called by _pcm_pointer; ignore
    – If delta jiffies from last _pos_update is >0; then we've been called by _timer_function - execute buffer copying through _xfer_buf!
  • _xfer_buf merely outsources copying algorithm to _fill_capture_buf
  • _fill_capture_buf finally does the copying algorithm:

```c
char *dst = mydev->substream->runtime->dma_area;
...
for (j=0; j<bytes; j++) {
  //... dst[mydev->buf_pos] = wvfdat[mydev->wvf_pos];
  dpos++; mydev->buf_pos++;
  mydev->wvf_pos++;
  //* or by using memcpy...
  //* ...
```
The capture process – timing and memory

- Special problem – wrapping of buffers; in minivosc we can distinguish:
  - intermediate (waveform) buffer/array - `wvfdat` - size 21 bytes
    - size preset by driver programmer
  - 'individual' transfer chunk size - given by bytes / count - size 32 (or 64) bytes
    - size dependent on timing between consecutive executions of `_timer_function_` & stream(s) format
  - `PCM substream buffer/array` - `dev->substream->runtime->dma_area` - size 816 (or 1536) bytes
    - size chosen by software (?): audacity usually claims 816 bytes, arecord 1536 bytes
  - `pcm_period_size` - size 48 bytes,
    - for calling `snd_pcm_period_elapsed`, size set by stream(s) format & kernel timer frequency
The capture process – buffer wrapping

- Special problem – wrapping of buffers; visualisation:
Buffer wrapping - “buffermarks”

- We can write special values in the beginning and end of all respective chunks; then in an audio editor we would obtain samples that will indicate the buffer sizes, or “buffermarks”
Conclusion

- Minivosc led to development of two open soundcard platforms (based on the same ALSA driver)
  - Audio Bare-bones FPGA [http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA](http://imi.aau.dk/~sd/phd/index.php?title=AudioBareBonesFPGA)
Trivia

• First released in 2010 ...
Demonstration

• Here a demonstration of building the driver