Writing an ALSA Driver

Takashi Iwai

<tiwai@suse.de>

Abstract

This document describes how to write an ALSA (Advanced Linux Sound Architecture) driver.

Table of Contents

Preface
1. File Tree Structure

- General
- core directory
  - core/oss
  - core/ioctl32
  - core/seq
  - core/seq/oss
  - core/seq/instr
- include directory
- drivers directory
  - drivers/mpu401
  - drivers/opl3 and opl4
- i2c directory
  - i2c/l3
- synth directory
- pci directory
- isa directory
- arm, ppc, and sparc directories
- usb directory
- pcmcia directory
- oss directory

2. Basic Flow for PCI Drivers

- Outline
- Full Code Example
- Constructor
  - 1) Check and increment the device index.
  - 2) Create a card instance
  - 3) Create a main component
  - 4) Set the driver ID and name strings.
  - 5) Create other components, such as mixer, MIDI, etc.
  - 6) Register the card instance.
  - 7) Set the PCI driver data and return zero.
- Destructor
- Header Files

3. Management of Cards and Components

- Card Instance
- Components
- Chip-Specific Data
  - 1. Allocating via `snd_card_create()`.
  - 2. Allocating an extra device.
- Registration and Release

4. PCI Resource Management

- Full Code Example
- Some Hafta's
- Resource Allocation
- Registration of Device Struct
- PCI Entries

5. PCM Interface

- General
- Full Code Example
- Constructor
- ... And the Destructor?
- Runtime Pointer - The Chest of PCM Information
  - Hardware Description
  - PCM Configurations
  - DMA Buffer Information
  - Running Status
  - Private Data
  - Interrupt Callbacks
- Operators
  - open callback
  - close callback
  - ioctl callback
  - hw_params callback

- hw_free callback
- prepare callback
- trigger callback
- pointer callback
- copy and silence callbacks
- ack callback
- page callback

Interrupt Handler

- Interrupts at the period (fragment) boundary
- High frequency timer interrupts
- On calling `snd_pcm_period_elapsed()`

Atomicity
Constraints

6. Control Interface

General
Definition of Controls
Control Names

- Global capture and playback
- Tone-controls
- 3D controls
- Mic boost

Access Flags
Callbacks

- info callback
- get callback
- put callback
- Callbacks are not atomic

Constructor
Change Notification
Metadata

7. API for AC97 Codec

General
Full Code Example
Constructor
Callbacks
Updating Registers in The Driver
Clock Adjustment
Proc Files
Multiple Codecs

8. MIDI (MPU401-UART) Interface

General
Constructor
Interrupt Handler

9. RawMIDI Interface

Overview
Constructor
Callbacks

`open` callback
`close` callback
`trigger` callback for output substreams
`trigger` callback for input substreams
`drain` callback

10. Miscellaneous Devices

FM OPL3
Hardware-Dependent Devices
IEC958 (S/PDIF)

11. Buffer and Memory Management

Buffer Types
External Hardware Buffers
Non-Contiguous Buffers
Vmalloc'ed Buffers

12. Proc Interface
13. Power Management
14. Module Parameters
15. How To Put Your Driver Into ALSA Tree

General
Driver with A Single Source File
Drivers with Several Source Files

16. Useful Functions

`snd_printk()` and friends
`snd_BUG()`
`snd_BUG_ON()`

17. Acknowledgments

List of Examples

1.1. ALSA File Tree Structure
2.1. Basic Flow for PCI Drivers - Example
4.1. PCI Resource Management Example
5.1. PCM Example Code
5.2. PCM Instance with a Destructor
5.3. Interrupt Handler Case #1
5.4. Interrupt Handler Case #2
5.5. Example of Hardware Constraints
5.6. Example of Hardware Constraints for Channels
5.7. Example of Hardware Constraints for Channels
6.1. Definition of a Control
6.2. Example of info callback
6.3. Example of get callback
6.4. Example of put callback
7.1. Example of AC97 Interface
15.1. Sample Makefile for a driver xyz

Preface

This document describes how to write an *ALSA (Advanced Linux Sound Architecture)* driver. The document focuses mainly on PCI soundcards. In the case of other device types, the API might be different, too. However, at least the ALSA kernel API is consistent, and therefore it would be still a bit help for writing them.

This document targets people who already have enough C language skills and have basic linux kernel programming knowledge. This document doesn't explain the general topic of linux kernel coding and doesn't cover low-level driver implementation details. It only describes the standard way to write a PCI sound driver on ALSA.

If you are already familiar with the older ALSA ver.0.5.x API, you can check the drivers such as `sound/pci/es1938.c` or `sound/pci/maestro3.c` which have also almost the same code-base in the ALSA 0.5.x tree, so you can compare the differences.

This document is still a draft version. Any feedback and corrections, please!!

Chapter 1. File Tree Structure

Table of Contents

General
core directory

- core/oss
- core/ioctl32
- core/seq
- core/seq/oss
- core/seq/instr

include directory
drivers directory

- drivers/mpu401
- drivers/opl3 and opl4

i2c directory

- i2c/l3

synth directory
pci directory
isa directory
arm, ppc, and sparc directories
usb directory
pcmcia directory
oss directory

General

The ALSA drivers are provided in two ways.

One is the trees provided as a tarball or via cvs from the ALSA's ftp site, and another is the 2.6 (or later) Linux kernel tree. To synchronize both, the ALSA driver tree is split into two different trees: alsa-kernel and alsa-driver. The former contains purely the source code for the Linux 2.6 (or later) tree. This tree is designed only for compilation on 2.6 or later environment. The latter, alsa-driver, contains many subtle files for compiling ALSA drivers outside of the

Linux kernel tree, wrapper functions for older 2.2 and 2.4 kernels, to adapt the latest kernel API, and additional drivers which are still in development or in tests. The drivers in alsa-driver tree will be moved to alsa-kernel (and eventually to the 2.6 kernel tree) when they are finished and confirmed to work fine.

The file tree structure of ALSA driver is depicted below. Both alsa-kernel and alsa-driver have almost the same file structure, except for “core” directory. It's named as “acore” in alsa-driver tree.

Example 1.1. ALSA File Tree Structure

```
sound
        /core
                /oss
                /seq
                        /oss
                        /instr
        /ioctl32
        /include
        /drivers
                /mpu401
                /opl3
        /i2c
                /l3
        /synth
                /emux
        /pci
                /(cards)
        /isa
                /(cards)
        /arm
        /ppc
        /sparc
        /usb
        /pcmcia /(cards)
        /oss
```

core directory

This directory contains the middle layer which is the heart of ALSA drivers. In this directory, the native ALSA modules are stored. The sub-directories contain different modules and are dependent upon the kernel config.

core/oss

The codes for PCM and mixer OSS emulation modules are stored in this directory. The rawmidi OSS emulation is included in the ALSA rawmidi code since it's quite small. The sequencer code is stored in `core/seq/oss` directory (see *below*).

core/ioctl32

This directory contains the 32bit-ioctl wrappers for 64bit architectures such like x86-64, ppc64 and sparc64. For 32bit and alpha architectures, these are not compiled.

core/seq

This directory and its sub-directories are for the ALSA sequencer. This directory contains the sequencer core and primary sequencer modules such like snd-seq-midi, snd-seq-virmidi, etc. They are compiled only when `CONFIG_SND_SEQUENCER` is set in the kernel config.

core/seq/oss

This contains the OSS sequencer emulation codes.

core/seq/instr

This directory contains the modules for the sequencer instrument layer.

include directory

This is the place for the public header files of ALSA drivers, which are to be exported to user-space, or included by several files at different directories. Basically, the private header files should not be placed in this directory, but you may still find files there, due to historical reasons :)

drivers directory

This directory contains code shared among different drivers on different architectures. They are hence supposed not to be architecture-specific. For example, the dummy pcm driver and the serial MIDI driver are found in this directory. In the sub-directories, there is code for components which are independent from bus and cpu architectures.

drivers/mpu401

The MPU401 and MPU401-UART modules are stored here.

drivers/opl3 and opl4

The OPL3 and OPL4 FM-synth stuff is found here.

i2c directory

This contains the ALSA i2c components.

Although there is a standard i2c layer on Linux, ALSA has its own i2c code for some cards, because the soundcard needs only a simple operation and the standard i2c API is too complicated for such a purpose.

i2c/l3

This is a sub-directory for ARM L3 i2c.

synth directory

This contains the synth middle-level modules.

So far, there is only Emu8000/Emu10k1 synth driver under the `synth/emux` sub-directory.

pci directory

This directory and its sub-directories hold the top-level card modules for PCI soundcards and the code specific to the PCI BUS.

PCI BUS.

The drivers compiled from a single file are stored directly in the pci directory, while the drivers with several source files are stored on their own sub-directory (e.g. emu10k1, ice1712).

isa directory

This directory and its sub-directories hold the top-level card modules for ISA soundcards.

arm, ppc, and sparc directories

They are used for top-level card modules which are specific to one of these architectures.

usb directory

This directory contains the USB-audio driver. In the latest version, the USB MIDI driver is integrated in the usb-audio driver.

pcmcia directory

The PCMCIA, especially PCCard drivers will go here. CardBus drivers will be in the pci directory, because their API is identical to that of standard PCI cards.

oss directory

The OSS/Lite source files are stored here in Linux 2.6 (or later) tree. In the ALSA driver tarball, this directory is empty, of course :)

Chapter 2. Basic Flow for PCI Drivers

Table of Contents

Outline
Full Code Example
Constructor

- 1) Check and increment the device index.
- 2) Create a card instance
- 3) Create a main component
- 4) Set the driver ID and name strings.
- 5) Create other components, such as mixer, MIDI, etc.
- 6) Register the card instance.
- 7) Set the PCI driver data and return zero.

Destructor
Header Files

Outline

The minimum flow for PCI soundcards is as follows:

- define the PCI ID table (see the section *PCI Entries*).

- create `probe()` callback.
- create `remove()` callback.
- create a pci_driver structure containing the three pointers above.
- create an `init()` function just calling the `pci_register_driver()` to register the pci_driver table defined above.
- create an `exit()` function to call the `pci_unregister_driver()` function.

Full Code Example

The code example is shown below. Some parts are kept unimplemented at this moment but will be filled in the next sections. The numbers in the comment lines of the `snd_mychip_probe()` function refer to details explained in the following section.

Example 2.1. Basic Flow for PCI Drivers - Example

```
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>

/* module parameters (see "Module Parameters") */
/* SNDRV_CARDS: maximum number of cards supported by this module */
static int index[SNDRV_CARDS] = SNDRV_DEFAULT_IDX;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;
static int enable[SNDRV_CARDS] = SNDRV_DEFAULT_ENABLE_PNP;

/* definition of the chip-specific record */
struct mychip {
        struct snd_card *card;
        /* the rest of the implementation will be in section
         * "PCI Resource Management"
         */
};

/* chip-specific destructor
 * (see "PCI Resource Management")
 */
static int snd_mychip_free(struct mychip *chip)
{
        .... /* will be implemented later... */
}

/* component-destructor
 * (see "Management of Cards and Components")
 */
static int snd_mychip_dev_free(struct snd_device *device)
{
        return snd_mychip_free(device->device_data);
}

/* chip-specific constructor
 * (see "Management of Cards and Components")
 */
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
        struct mychip *chip;
```

```
        int err;
        static struct snd_device_ops ops = {
               .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* check PCI availability here
         * (see "PCI Resource Management")
         */
        ....

        /* allocate a chip-specific data with zero filled */
        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL)
                return -ENOMEM;

        chip->card = card;

        /* rest of initialization here; will be implemented
         * later, see "PCI Resource Management"
         */
        ....

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        snd_card_set_dev(card, &pci->dev);

        *rchip = chip;
        return 0;
}

/* constructor -- see "Constructor" sub-section */
static int __devinit snd_mychip_probe(struct pci_dev *pci,
                             const struct pci_device_id *pci_id)
{
        static int dev;
        struct snd_card *card;
        struct mychip *chip;
        int err;

        /* (1) */
        if (dev >= SNDRV_CARDS)
                return -ENODEV;
        if (!enable[dev]) {
                dev++;
                return -ENOENT;
        }

        /* (2) */
        err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
        if (err < 0)
                return err;

        /* (3) */
        err = snd_mychip_create(card, pci, &chip);
        if (err < 0) {
                snd_card_free(card);
                return err;
        }

        /* (4) */
        strcpy(card->driver, "My Chip");
        strcpy(card->shortname, "My Own Chip 123");
```

```
        sprintf(card->longname, "%s at 0x%lx irq %i",
                card->shortname, chip->ioport, chip->irq);

        /* (5) */
        .... /* implemented later */

        /* (6) */
        err = snd_card_register(card);
        if (err < 0) {
                snd_card_free(card);
                return err;
        }

        /* (7) */
        pci_set_drvdata(pci, card);
        dev++;
        return 0;
}

/* destructor -- see the "Destructor" sub-section */
static void __devexit snd_mychip_remove(struct pci_dev *pci)
{
        snd_card_free(pci_get_drvdata(pci));
        pci_set_drvdata(pci, NULL);
}
```

Constructor

The real constructor of PCI drivers is the `probe` callback. The `probe` callback and other component-constructors which are called from the `probe` callback should be defined with the *`__devinit`* prefix. You cannot use the *`__init`* prefix for them, because any PCI device could be a hotplug device.

In the `probe` callback, the following scheme is often used.

1) Check and increment the device index.

```
static int dev;
....
if (dev >= SNDRV_CARDS)
        return -ENODEV;
if (!enable[dev]) {
        dev++;
        return -ENOENT;
}
```

where enable[dev] is the module option.

Each time the `probe` callback is called, check the availability of the device. If not available, simply increment the device index and returns. dev will be incremented also later (*step 7*).

2) Create a card instance

```
struct snd_card *card;
int err;
....
err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
```

```
err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
```

The details will be explained in the section *Management of Cards and Components*.

3) Create a main component

In this part, the PCI resources are allocated.

```
struct mychip *chip;
....
err = snd_mychip_create(card, pci, &chip);
if (err < 0) {
        snd_card_free(card);
        return err;
}
```

The details will be explained in the section *PCI Resource Management*.

4) Set the driver ID and name strings.

```
strcpy(card->driver, "My Chip");
strcpy(card->shortname, "My Own Chip 123");
sprintf(card->longname, "%s at 0x%lx irq %i",
        card->shortname, chip->ioport, chip->irq);
```

The driver field holds the minimal ID string of the chip. This is used by alsa-lib's configurator, so keep it simple but unique. Even the same driver can have different driver IDs to distinguish the functionality of each chip type.

The shortname field is a string shown as more verbose name. The longname field contains the information shown in `/proc/asound/cards`.

5) Create other components, such as mixer, MIDI, etc.

Here you define the basic components such as *PCM*, mixer (e.g. *AC97*), MIDI (e.g. *MPU-401*), and other interfaces. Also, if you want a *proc file*, define it here, too.

6) Register the card instance.

```
err = snd_card_register(card);
if (err < 0) {
        snd_card_free(card);
        return err;
}
```

Will be explained in the section *Management of Cards and Components*, too.

7) Set the PCI driver data and return zero.

```
        pci_set_drvdata(pci, card);
        dev++;
```

```
        return 0;
```

In the above, the card record is stored. This pointer is used in the remove callback and power-management callbacks, too.

Destructor

The destructor, remove callback, simply releases the card instance. Then the ALSA middle layer will release all the attached components automatically.

It would be typically like the following:

```
static void __devexit snd_mychip_remove(struct pci_dev *pci)
{
        snd_card_free(pci_get_drvdata(pci));
        pci_set_drvdata(pci, NULL);
}
```

The above code assumes that the card pointer is set to the PCI driver data.

Header Files

For the above example, at least the following include files are necessary.

```
#include <linux/init.h>
#include <linux/pci.h>
#include <linux/slab.h>
#include <sound/core.h>
#include <sound/initval.h>
```

where the last one is necessary only when module options are defined in the source file. If the code is split into several files, the files without module options don't need them.

In addition to these headers, you'll need `<linux/interrupt.h>` for interrupt handling, and `<asm/io.h>` for I/O access. If you use the `mdelay()` or `udelay()` functions, you'll need to include `<linux/delay.h>` too.

The ALSA interfaces like the PCM and control APIs are defined in other `<sound/xxx.h>` header files. They have to be included after `<sound/core.h>`.

Chapter 3. Management of Cards and Components

Table of Contents

[Card Instance]
[Components]
[Chip-Specific Data]

  [1. Allocating via `snd_card_create()`.]
  [2. Allocating an extra device.]

[Registration and Release]

Card Instance

For each soundcard, a “card” record must be allocated.

A card record is the headquarters of the soundcard. It manages the whole list of devices (components) on the soundcard, such as PCM, mixers, MIDI, synthesizer, and so on. Also, the card record holds the ID and the name strings of the card, manages the root of proc files, and controls the power-management states and hotplug disconnections. The component list on the card record is used to manage the correct release of resources at destruction.

As mentioned above, to create a card instance, call `snd_card_create()`.

```
struct snd_card *card;
int err;
err = snd_card_create(index, id, module, extra_size, &card);
```

The function takes five arguments, the card-index number, the id string, the module pointer (usually `THIS_MODULE`), the size of extra-data space, and the pointer to return the card instance. The extra_size argument is used to allocate card->private_data for the chip-specific data. Note that these data are allocated by `snd_card_create()`.

Components

After the card is created, you can attach the components (devices) to the card instance. In an ALSA driver, a component is represented as a struct snd_device object. A component can be a PCM instance, a control interface, a raw MIDI interface, etc. Each such instance has one component entry.

A component can be created via `snd_device_new()` function.

```
snd_device_new(card, SNDRV_DEV_XXX, chip, &ops);
```

This takes the card pointer, the device-level (`SNDRV_DEV_XXX`), the data pointer, and the callback pointers (*`&ops`*). The device-level defines the type of components and the order of registration and de-registration. For most components, the device-level is already defined. For a user-defined component, you can use `SNDRV_DEV_LOWLEVEL`.

This function itself doesn't allocate the data space. The data must be allocated manually beforehand, and its pointer is passed as the argument. This pointer is used as the (*`chip`* identifier in the above example) for the instance.

Each pre-defined ALSA component such as ac97 and pcm calls `snd_device_new()` inside its constructor. The destructor for each component is defined in the callback pointers. Hence, you don't need to take care of calling a destructor for such a component.

If you wish to create your own component, you need to set the destructor function to the dev_free callback in the *`ops`*, so that it can be released automatically via `snd_card_free()`. The next example will show an implementation of chip-specific data.

Chip-Specific Data

Chip-specific information, e.g. the I/O port address, its resource pointer, or the irq number, is stored in the chip-specific record.

```
struct mychip {
```

```
struct mychip {
        ....
};
```

In general, there are two ways of allocating the chip record.

1. Allocating via `snd_card_create()`.

As mentioned above, you can pass the extra-data-length to the 4th argument of `snd_card_create()`, i.e.

```
err = snd_card_create(index[dev], id[dev], THIS_MODULE,
                      sizeof(struct mychip), &card);
```

struct mychip is the type of the chip record.

In return, the allocated record can be accessed as

```
struct mychip *chip = card->private_data;
```

With this method, you don't have to allocate twice. The record is released together with the card instance.

2. Allocating an extra device.

After allocating a card instance via `snd_card_create()` (with `0` on the 4th arg), call `kzalloc()`.

```
struct snd_card *card;
struct mychip *chip;
err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
.....
chip = kzalloc(sizeof(*chip), GFP_KERNEL);
```

The chip record should have the field to hold the card pointer at least,

```
struct mychip {
        struct snd_card *card;
        ....
};
```

Then, set the card pointer in the returned chip instance.

```
chip->card = card;
```

Next, initialize the fields, and register this chip record as a low-level device with a specified *ops*,

```
static struct snd_device_ops ops = {
        .dev_free =        snd_mychip_dev_free,
};
```

```
};
....
snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
```

`snd_mychip_dev_free()` is the device-destructor function, which will call the real destructor.

```
static int snd_mychip_dev_free(struct snd_device *device)
{
        return snd_mychip_free(device->device_data);
}
```

where `snd_mychip_free()` is the real destructor.

Registration and Release

After all components are assigned, register the card instance by calling `snd_card_register()`. Access to the device files is enabled at this point. That is, before `snd_card_register()` is called, the components are safely inaccessible from external side. If this call fails, exit the probe function after releasing the card via `snd_card_free()`.

For releasing the card instance, you can call simply `snd_card_free()`. As mentioned earlier, all components are released automatically by this call.

As further notes, the destructors (both `snd_mychip_dev_free` and `snd_mychip_free`) cannot be defined with the `__devexit` prefix, because they may be called from the constructor, too, at the false path.

For a device which allows hotplugging, you can use `snd_card_free_when_closed`. This one will postpone the destruction until all devices are closed.

Chapter 4. PCI Resource Management

Table of Contents

[Full Code Example](#)
[Some Hafta's](#)
[Resource Allocation](#)
[Registration of Device Struct](#)
[PCI Entries](#)

Full Code Example

In this section, we'll complete the chip-specific constructor, destructor and PCI entries. Example code is shown first, below.

Example 4.1. PCI Resource Management Example

```
struct mychip {
        struct snd_card *card;
        struct pci_dev *pci;

        unsigned long port;
        int irq;
};

static int snd_mychip_free(struct mychip *chip)
```

```
{
        /* disable hardware here if any */
        .... /* (not implemented in this document) */

        /* release the irq */
        if (chip->irq >= 0)
                free_irq(chip->irq, chip);
        /* release the I/O ports & memory */
        pci_release_regions(chip->pci);
        /* disable the PCI entry */
        pci_disable_device(chip->pci);
        /* release the data */
        kfree(chip);
        return 0;
}

/* chip-specific constructor */
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
        struct mychip *chip;
        int err;
        static struct snd_device_ops ops = {
               .dev_free = snd_mychip_dev_free,
        };

        *rchip = NULL;

        /* initialize the PCI entry */
        err = pci_enable_device(pci);
        if (err < 0)
                return err;
        /* check PCI availability (28bit DMA) */
        if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
            pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
                printk(KERN_ERR "error to set 28bit mask DMA\n");
                pci_disable_device(pci);
                return -ENXIO;
        }

        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        if (chip == NULL) {
                pci_disable_device(pci);
                return -ENOMEM;
        }

        /* initialize the stuff */
        chip->card = card;
        chip->pci = pci;
        chip->irq = -1;

        /* (1) PCI resource allocation */
        err = pci_request_regions(pci, "My Chip");
        if (err < 0) {
                kfree(chip);
                pci_disable_device(pci);
                return err;
        }
        chip->port = pci_resource_start(pci, 0);
        if (request_irq(pci->irq, snd_mychip_interrupt,
                        IRQF_SHARED, "My Chip", chip)) {
                printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
                snd_mychip_free(chip);
                return -EBUSY;
        }
        chip->irq = pci->irq;
```

```
        /* (2) initialization of the chip hardware */
        .... /*   (not implemented in this document) */

        err = snd_device_new(card, SNDRV_DEV_LOWLEVEL, chip, &ops);
        if (err < 0) {
                snd_mychip_free(chip);
                return err;
        }

        snd_card_set_dev(card, &pci->dev);

        *rchip = chip;
        return 0;
}

/* PCI IDs */
static struct pci_device_id snd_mychip_ids[] = {
        { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
        ....
        { 0, }
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);

/* pci_driver definition */
static struct pci_driver driver = {
        .name = "My Own Chip",
        .id_table = snd_mychip_ids,
        .probe = snd_mychip_probe,
        .remove = __devexit_p(snd_mychip_remove),
};

/* module initialization */
static int __init alsa_card_mychip_init(void)
{
        return pci_register_driver(&driver);
}

/* module clean up */
static void __exit alsa_card_mychip_exit(void)
{
        pci_unregister_driver(&driver);
}

module_init(alsa_card_mychip_init)
module_exit(alsa_card_mychip_exit)

EXPORT_NO_SYMBOLS; /* for old kernels only */
```

Some Hafta's

The allocation of PCI resources is done in the `probe()` function, and usually an extra `xxx_create()` function is written for this purpose.

In the case of PCI devices, you first have to call the `pci_enable_device()` function before allocating resources. Also, you need to set the proper PCI DMA mask to limit the accessed I/O range. In some cases, you might need to call `pci_set_master()` function, too.

Suppose the 28bit mask, and the code to be added would be like:

```
err = pci_enable_device(pci);
if (err < 0)
        return err;
if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
    pci_set_consistent_dma_mask(pci, DMA_BIT_MASK(28)) < 0) {
        printk(KERN_ERR "error to set 28bit mask DMA\n");
        pci_disable_device(pci);
        return -ENXIO;
}
```

Resource Allocation

The allocation of I/O ports and irqs is done via standard kernel functions. Unlike ALSA ver.0.5.x., there are no helpers for that. And these resources must be released in the destructor function (see below). Also, on ALSA 0.9.x, you don't need to allocate (pseudo-)DMA for PCI like in ALSA 0.5.x.

Now assume that the PCI device has an I/O port with 8 bytes and an interrupt. Then struct mychip will have the following fields:

```
struct mychip {
        struct snd_card *card;

        unsigned long port;
        int irq;
};
```

For an I/O port (and also a memory region), you need to have the resource pointer for the standard resource management. For an irq, you have to keep only the irq number (integer). But you need to initialize this number as -1 before actual allocation, since irq 0 is valid. The port address and its resource pointer can be initialized as null by `kzalloc()` automatically, so you don't have to take care of resetting them.

The allocation of an I/O port is done like this:

```
err = pci_request_regions(pci, "My Chip");
if (err < 0) {
        kfree(chip);
        pci_disable_device(pci);
        return err;
}
chip->port = pci_resource_start(pci, 0);
```

It will reserve the I/O port region of 8 bytes of the given PCI device. The returned value, chip->res_port, is allocated via `kmalloc()` by `request_region()`. The pointer must be released via `kfree()`, but there is a problem with this. This issue will be explained later.

The allocation of an interrupt source is done like this:

```
if (request_irq(pci->irq, snd_mychip_interrupt,
                IRQF_SHARED, "My Chip", chip)) {
        printk(KERN_ERR "cannot grab irq %d\n", pci->irq);
        snd_mychip_free(chip);
        return -EBUSY;
}
chip->irq = pci->irq;
```

where `snd_mychip_interrupt()` is the interrupt handler defined *later*. Note that chip->irq should be defined only when `request_irq()` succeeded.

On the PCI bus, interrupts can be shared. Thus, `IRQF_SHARED` is used as the interrupt flag of `request_irq()`.

The last argument of `request_irq()` is the data pointer passed to the interrupt handler. Usually, the chip-specific record is used for that, but you can use what you like, too.

I won't give details about the interrupt handler at this point, but at least its appearance can be explained now. The interrupt handler looks usually like the following:

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        ....
        return IRQ_HANDLED;
}
```

Now let's write the corresponding destructor for the resources above. The role of destructor is simple: disable the hardware (if already activated) and release the resources. So far, we have no hardware part, so the disabling code is not written here.

To release the resources, the “check-and-release” method is a safer way. For the interrupt, do like this:

```
if (chip->irq >= 0)
        free_irq(chip->irq, chip);
```

Since the irq number can start from 0, you should initialize chip->irq with a negative value (e.g. -1), so that you can check the validity of the irq number as above.

When you requested I/O ports or memory regions via `pci_request_region()` or `pci_request_regions()` like in this example, release the resource(s) using the corresponding function, `pci_release_region()` or `pci_release_regions()`.

```
pci_release_regions(chip->pci);
```

When you requested manually via `request_region()` or `request_mem_region`, you can release it via `release_resource()`. Suppose that you keep the resource pointer returned from `request_region()` in chip->res_port, the release procedure looks like:

```
release_and_free_resource(chip->res_port);
```

Don't forget to call `pci_disable_device()` before the end.

And finally, release the chip-specific record.

```
kfree(chip);
```

Again, remember that you cannot use the *__devexit* prefix for this destructor.

We didn't implement the hardware disabling part in the above. If you need to do this, please note that the destructor may be called even before the initialization of the chip is completed. It would be better to have a flag to skip hardware disabling if the hardware was not initialized yet.

When the chip-data is assigned to the card using `snd_device_new()` with `SNDRV_DEV_LOWLELVEL` , its destructor is called at the last. That is, it is assured that all other components like PCMs and controls have already been released. You don't have to stop PCMs, etc. explicitly, but just call low-level hardware stopping.

The management of a memory-mapped region is almost as same as the management of an I/O port. You'll need three fields like the following:

```
struct mychip {
        ....
        unsigned long iobase_phys;
        void __iomem *iobase_virt;
};
```

and the allocation would be like below:

```
if ((err = pci_request_regions(pci, "My Chip")) < 0) {
        kfree(chip);
        return err;
}
chip->iobase_phys = pci_resource_start(pci, 0);
chip->iobase_virt = ioremap_nocache(chip->iobase_phys,
                                    pci_resource_len(pci, 0));
```

and the corresponding destructor would be:

```
static int snd_mychip_free(struct mychip *chip)
{
        ....
        if (chip->iobase_virt)
                iounmap(chip->iobase_virt);
        ....
        pci_release_regions(chip->pci);
        ....
}
```

Registration of Device Struct

At some point, typically after calling `snd_device_new()`, you need to register the struct device of the chip you're handling for udev and co. ALSA provides a macro for compatibility with older kernels. Simply call like the following:

```
snd_card_set_dev(card, &pci->dev);
```

so that it stores the PCI's device pointer to the card. This will be referred by ALSA core functions later when the devices are registered.

In the case of non-PCI, pass the proper device struct pointer of the BUS instead. (In the case of legacy ISA without PnP, you don't have to do anything.)

PCI Entries

So far, so good. Let's finish the missing PCI stuff. At first, we need a pci_device_id table for this chipset. It's a table of PCI vendor/device ID number, and some masks.

For example,

```
static struct pci_device_id snd_mychip_ids[] = {
        { PCI_VENDOR_ID_FOO, PCI_DEVICE_ID_BAR,
          PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0, },
        ....
        { 0, }
};
MODULE_DEVICE_TABLE(pci, snd_mychip_ids);
```

The first and second fields of the pci_device_id structure are the vendor and device IDs. If you have no reason to filter the matching devices, you can leave the remaining fields as above. The last field of the pci_device_id struct contains private data for this entry. You can specify any value here, for example, to define specific operations for supported device IDs. Such an example is found in the intel8x0 driver.

The last entry of this list is the terminator. You must specify this all-zero entry.

Then, prepare the pci_driver record:

```
static struct pci_driver driver = {
        .name = "My Own Chip",
        .id_table = snd_mychip_ids,
        .probe = snd_mychip_probe,
        .remove = __devexit_p(snd_mychip_remove),
};
```

The *`probe`* and *`remove`* functions have already been defined in the previous sections. The *`remove`* function should be defined with the `__devexit_p()` macro, so that it's not defined for built-in (and non-hot-pluggable) case. The *`name`* field is the name string of this device. Note that you must not use a slash "/" in this string.

And at last, the module entries:

```
static int __init alsa_card_mychip_init(void)
{
        return pci_register_driver(&driver);
}

static void __exit alsa_card_mychip_exit(void)
{
        pci_unregister_driver(&driver);
}

module_init(alsa_card_mychip_init)
module_exit(alsa_card_mychip_exit)
```

Note that these module entries are tagged with `__init` and `__exit` prefixes, not `__devinit` nor `__devexit`.

Oh, one thing was forgotten. If you have no exported symbols, you need to declare it in 2.2 or 2.4 kernels (it's not necessary in 2.6 kernels).

```
EXPORT_NO_SYMBOLS;
```

That's all!

Chapter 5. PCM Interface

Table of Contents

General
Full Code Example
Constructor
... And the Destructor?
Runtime Pointer - The Chest of PCM Information

- Hardware Description
- PCM Configurations
- DMA Buffer Information
- Running Status
- Private Data
- Interrupt Callbacks

Operators

- open callback
- close callback
- ioctl callback
- hw_params callback
- hw_free callback
- prepare callback
- trigger callback
- pointer callback
- copy and silence callbacks
- ack callback
- page callback

Interrupt Handler

- Interrupts at the period (fragment) boundary
- High frequency timer interrupts
- On calling `snd_pcm_period_elapsed()`

Atomicity
Constraints

General

The PCM middle layer of ALSA is quite powerful and it is only necessary for each driver to implement the low-level

The PCM middle layer of ALSA is quite powerful and it is only necessary for each driver to implement the low-level functions to access its hardware.

For accessing to the PCM layer, you need to include <sound/pcm.h> first. In addition, <sound/pcm_params.h> might be needed if you access to some functions related with hw_param.

Each card device can have up to four pcm instances. A pcm instance corresponds to a pcm device file. The limitation of number of instances comes only from the available bit size of the Linux's device numbers. Once when 64bit device number is used, we'll have more pcm instances available.

A pcm instance consists of pcm playback and capture streams, and each pcm stream consists of one or more pcm substreams. Some soundcards support multiple playback functions. For example, emu10k1 has a PCM playback of 32 stereo substreams. In this case, at each open, a free substream is (usually) automatically chosen and opened. Meanwhile, when only one substream exists and it was already opened, the successful open will either block or error with EAGAIN according to the file open mode. But you don't have to care about such details in your driver. The PCM middle layer will take care of such work.

Full Code Example

The example code below does not include any hardware access routines but shows only the skeleton, how to build up the PCM interfaces.

Example 5.1. PCM Example Code

```
#include <sound/pcm.h>
....

/* hardware definition */
static struct snd_pcm_hardware snd_mychip_playback_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                 SNDRV_PCM_INFO_INTERLEAVED |
                 SNDRV_PCM_INFO_BLOCK_TRANSFER |
                 SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};

/* hardware definition */
static struct snd_pcm_hardware snd_mychip_capture_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                 SNDRV_PCM_INFO_INTERLEAVED |
                 SNDRV_PCM_INFO_BLOCK_TRANSFER |
                 SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024
```

```
        .periods_max =          1024,
};

/* open callback */
static int snd_mychip_playback_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_playback_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_playback_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;

}

/* open callback */
static int snd_mychip_capture_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_capture_hw;
        /* more hardware-initialization will be done here */
        ....
        return 0;
}

/* close callback */
static int snd_mychip_capture_close(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        /* the hardware-specific codes will be here */
        ....
        return 0;

}

/* hw_params callback */
static int snd_mychip_pcm_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *hw_params)
{
        return snd_pcm_lib_malloc_pages(substream,
                                   params_buffer_bytes(hw_params));
}

/* hw_free callback */
static int snd_mychip_pcm_hw_free(struct snd_pcm_substream *substream)
{
        return snd_pcm_lib_free_pages(substream);
}

/* prepare callback */
static int snd_mychip_pcm_prepare(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        /* set up the hardware with the current configuration
         * for example
```

```
          * for example...
          */
         mychip_set_sample_format(chip, runtime->format);
         mychip_set_sample_rate(chip, runtime->rate);
         mychip_set_channels(chip, runtime->channels);
         mychip_set_dma_setup(chip, runtime->dma_addr,
                              chip->buffer_size,
                              chip->period_size);
         return 0;
 }

 /* trigger callback */
 static int snd_mychip_pcm_trigger(struct snd_pcm_substream *substream,
                                   int cmd)
 {
         switch (cmd) {
         case SNDRV_PCM_TRIGGER_START:
                 /* do something to start the PCM engine */
                 ....
                 break;
         case SNDRV_PCM_TRIGGER_STOP:
                 /* do something to stop the PCM engine */
                 ....
                 break;
         default:
                 return -EINVAL;
         }
 }

 /* pointer callback */
 static snd_pcm_uframes_t
 snd_mychip_pcm_pointer(struct snd_pcm_substream *substream)
 {
         struct mychip *chip = snd_pcm_substream_chip(substream);
         unsigned int current_ptr;

         /* get the current hardware pointer */
         current_ptr = mychip_get_hw_pointer(chip);
         return current_ptr;
 }

 /* operators */
 static struct snd_pcm_ops snd_mychip_playback_ops = {
         .open =        snd_mychip_playback_open,
         .close =       snd_mychip_playback_close,
         .ioctl =       snd_pcm_lib_ioctl,
         .hw_params =   snd_mychip_pcm_hw_params,
         .hw_free =     snd_mychip_pcm_hw_free,
         .prepare =     snd_mychip_pcm_prepare,
         .trigger =     snd_mychip_pcm_trigger,
         .pointer =     snd_mychip_pcm_pointer,
 };

 /* operators */
 static struct snd_pcm_ops snd_mychip_capture_ops = {
         .open =        snd_mychip_capture_open,
         .close =       snd_mychip_capture_close,
         .ioctl =       snd_pcm_lib_ioctl,
         .hw_params =   snd_mychip_pcm_hw_params,
         .hw_free =     snd_mychip_pcm_hw_free,
         .prepare =     snd_mychip_pcm_prepare,
         .trigger =     snd_mychip_pcm_trigger,
         .pointer =     snd_mychip_pcm_pointer,
 };

 /*
  *  definitions of capture are omitted here...
  */
```

```
/* create a pcm device */
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        int err;

        err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
        if (err < 0)
                return err;
        pcm->private_data = chip;
        strcpy(pcm->name, "My Chip");
        chip->pcm = pcm;
        /* set operators */
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                        &snd_mychip_playback_ops);
        snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                        &snd_mychip_capture_ops);
        /* pre-allocation of buffers */
        /* NOTE: this may fail */
        snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                              snd_dma_pci_data(chip->pci),
                                              64*1024, 64*1024);
        return 0;
}
```

Constructor

A pcm instance is allocated by the `snd_pcm_new()` function. It would be better to create a constructor for pcm, namely,

```
static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        int err;

        err = snd_pcm_new(chip->card, "My Chip", 0, 1, 1, &pcm);
        if (err < 0)
                return err;
        pcm->private_data = chip;
        strcpy(pcm->name, "My Chip");
        chip->pcm = pcm;
        ....
        return 0;
}
```

The `snd_pcm_new()` function takes four arguments. The first argument is the card pointer to which this pcm is assigned, and the second is the ID string.

The third argument (*`index`*, 0 in the above) is the index of this new pcm. It begins from zero. If you create more than one pcm instances, specify the different numbers in this argument. For example, *`index`* = 1 for the second PCM device.

The fourth and fifth arguments are the number of substreams for playback and capture, respectively. Here 1 is used for both arguments. When no playback or capture substreams are available, pass 0 to the corresponding argument.

If a chip supports multiple playbacks or captures, you can specify more numbers, but they must be handled properly in open/close, etc. callbacks. When you need to know which substream you are referring to, then it can be obtained in open/close, etc. callbacks. When you need to know which substream you are referring to, then it can be obtained from struct snd_pcm_substream data passed to each callback as follows:

```
struct snd_pcm_substream *substream;
int index = substream->number;
```

After the pcm is created, you need to set operators for each pcm stream.

```
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                &snd_mychip_playback_ops);
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_CAPTURE,
                &snd_mychip_capture_ops);
```

The operators are defined typically like this:

```
static struct snd_pcm_ops snd_mychip_playback_ops = {
        .open =        snd_mychip_pcm_open,
        .close =       snd_mychip_pcm_close,
        .ioctl =       snd_pcm_lib_ioctl,
        .hw_params =   snd_mychip_pcm_hw_params,
        .hw_free =     snd_mychip_pcm_hw_free,
        .prepare =     snd_mychip_pcm_prepare,
        .trigger =     snd_mychip_pcm_trigger,
        .pointer =     snd_mychip_pcm_pointer,
};
```

All the callbacks are described in the *Operators* subsection.

After setting the operators, you probably will want to pre-allocate the buffer. For the pre-allocation, simply call the following:

```
snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                      snd_dma_pci_data(chip->pci),
                                      64*1024, 64*1024);
```

It will allocate a buffer up to 64kB as default. Buffer management details will be described in the later section *Buffer and Memory Management*.

Additionally, you can set some extra information for this pcm in pcm->info_flags. The available values are defined as `SNDRV_PCM_INFO_XXX` in `<sound/asound.h>`, which is used for the hardware definition (described later). When your soundchip supports only half-duplex, specify like this:

```
pcm->info_flags = SNDRV_PCM_INFO_HALF_DUPLEX;
```

... And the Destructor?

The destructor for a pcm instance is not always necessary. Since the pcm device will be released by the middle layer code automatically, you don't have to call the destructor explicitly.

The destructor would be necessary if you created special records internally and needed to release them. In such a case, set the destructor function to pcm->private_free:

Example 5.2. PCM Instance with a Destructor

```
static void mychip_pcm_free(struct snd_pcm *pcm)
{
        struct mychip *chip = snd_pcm_chip(pcm);
        /* free your own data */
        kfree(chip->my_private_pcm_data);
        /* do what you like else */
        ....
}

static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
        struct snd_pcm *pcm;
        ....
        /* allocate your own data */
        chip->my_private_pcm_data = kmalloc(...);
        /* set the destructor */
        pcm->private_data = chip;
        pcm->private_free = mychip_pcm_free;
        ....
}
```

Runtime Pointer - The Chest of PCM Information

When the PCM substream is opened, a PCM runtime instance is allocated and assigned to the substream. This pointer is accessible via `substream->runtime`. This runtime pointer holds most information you need to control the PCM: the copy of hw_params and sw_params configurations, the buffer pointers, mmap records, spinlocks, etc.

The definition of runtime instance is found in `<sound/pcm.h>`. Here are the contents of this file:

```
struct _snd_pcm_runtime {
        /* -- Status -- */
        struct snd_pcm_substream *trigger_master;
        snd_timestamp_t trigger_tstamp; /* trigger timestamp */
        int overrange;
        snd_pcm_uframes_t avail_max;
        snd_pcm_uframes_t hw_ptr_base;  /* Position at buffer restart */
        snd_pcm_uframes_t hw_ptr_interrupt; /* Position at interrupt time*/

        /* -- HW params -- */
        snd_pcm_access_t access;        /* access mode */
        snd_pcm_format_t format;        /* SNDRV_PCM_FORMAT_* */
        snd_pcm_subformat_t subformat;  /* subformat */
        unsigned int rate;              /* rate in Hz */
        unsigned int channels;          /* channels */
        snd_pcm_uframes_t period_size;  /* period size */
        unsigned int periods;           /* periods */
        snd_pcm_uframes_t buffer_size;  /* buffer size */
        unsigned int tick_time;         /* tick time */
        snd_pcm_uframes_t min_align;    /* Min alignment for the format */
        size_t byte_align;
        unsigned int frame_bits;
        unsigned int sample_bits;
        unsigned int info;
        unsigned int rate_num;
```

```
        unsigned int rate_den;

        /* -- SW params -- */
        struct timespec tstamp_mode;    /* mmap timestamp is updated */
        unsigned int period_step;
        unsigned int sleep_min;         /* min ticks to sleep */
        snd_pcm_uframes_t start_threshold;
        snd_pcm_uframes_t stop_threshold;
        snd_pcm_uframes_t silence_threshold; /* Silence filling happens when
                                                noise is nearest than this */
        snd_pcm_uframes_t silence_size; /* Silence filling size */
        snd_pcm_uframes_t boundary;     /* pointers wrap point */

        snd_pcm_uframes_t silenced_start;
        snd_pcm_uframes_t silenced_size;

        snd_pcm_sync_id_t sync;         /* hardware synchronization ID */

        /* -- mmap -- */
        volatile struct snd_pcm_mmap_status *status;
        volatile struct snd_pcm_mmap_control *control;
        atomic_t mmap_count;

        /* -- locking / scheduling -- */
        spinlock_t lock;
        wait_queue_head_t sleep;
        struct timer_list tick_timer;
        struct fasync_struct *fasync;

        /* -- private section -- */
        void *private_data;
        void (*private_free)(struct snd_pcm_runtime *runtime);

        /* -- hardware description -- */
        struct snd_pcm_hardware hw;
        struct snd_pcm_hw_constraints hw_constraints;

        /* -- interrupt callbacks -- */
        void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
        void (*transfer_ack_end)(struct snd_pcm_substream *substream);

        /* -- timer -- */
        unsigned int timer_resolution;  /* timer resolution */

        /* -- DMA -- */
        unsigned char *dma_area;        /* DMA area */
        dma_addr_t dma_addr;            /* physical bus address (not accessible from main CPU) */
        size_t dma_bytes;               /* size of DMA area */

        struct snd_dma_buffer *dma_buffer_p;    /* allocated buffer */

#if defined(CONFIG_SND_PCM_OSS) || defined(CONFIG_SND_PCM_OSS_MODULE)
        /* -- OSS things -- */
        struct snd_pcm_oss_runtime oss;
#endif
};
```

For the operators (callbacks) of each sound driver, most of these records are supposed to be read-only. Only the PCM middle-layer changes / updates them. The exceptions are the hardware description (hw), interrupt callbacks (transfer_ack_xxx), DMA buffer information, and the private data. Besides, if you use the standard buffer allocation method via `snd_pcm_lib_malloc_pages()`, you don't need to set the DMA buffer information by yourself.

In the sections below, important records are explained.

Hardware Description

Hardware Description

The hardware descriptor (struct snd_pcm_hardware) contains the definitions of the fundamental hardware configuration. Above all, you'll need to define this in *the open callback*. Note that the runtime instance holds the copy of the descriptor, not the pointer to the existing descriptor. That is, in the open callback, you can modify the copied descriptor (`runtime->hw`) as you need. For example, if the maximum number of channels is 1 only on some chip models, you can still use the same hardware descriptor and change the channels_max later:

```
struct snd_pcm_runtime *runtime = substream->runtime;
...
runtime->hw = snd_mychip_playback_hw; /* common definition */
if (chip->model == VERY_OLD_ONE)
        runtime->hw.channels_max = 1;
```

Typically, you'll have a hardware descriptor as below:

```
static struct snd_pcm_hardware snd_mychip_playback_hw = {
        .info = (SNDRV_PCM_INFO_MMAP |
                 SNDRV_PCM_INFO_INTERLEAVED |
                 SNDRV_PCM_INFO_BLOCK_TRANSFER |
                 SNDRV_PCM_INFO_MMAP_VALID),
        .formats =          SNDRV_PCM_FMTBIT_S16_LE,
        .rates =            SNDRV_PCM_RATE_8000_48000,
        .rate_min =         8000,
        .rate_max =         48000,
        .channels_min =     2,
        .channels_max =     2,
        .buffer_bytes_max = 32768,
        .period_bytes_min = 4096,
        .period_bytes_max = 32768,
        .periods_min =      1,
        .periods_max =      1024,
};
```

- The *`info`* field contains the type and capabilities of this pcm. The bit flags are defined in `<sound/asound.h>` as `SNDRV_PCM_INFO_XXX`. Here, at least, you have to specify whether the mmap is supported and which interleaved format is supported. When the is supported, add the `SNDRV_PCM_INFO_MMAP` flag here. When the hardware supports the interleaved or the non-interleaved formats, `SNDRV_PCM_INFO_INTERLEAVED` or `SNDRV_PCM_INFO_NONINTERLEAVED` flag must be set, respectively. If both are supported, you can set both, too.

  In the above example, `MMAP_VALID` and `BLOCK_TRANSFER` are specified for the OSS mmap mode. Usually both are set. Of course, `MMAP_VALID` is set only if the mmap is really supported.

  The other possible flags are `SNDRV_PCM_INFO_PAUSE` and `SNDRV_PCM_INFO_RESUME`. The `PAUSE` bit means that the pcm supports the “pause” operation, while the `RESUME` bit means that the pcm supports the full “suspend/resume” operation. If the `PAUSE` flag is set, the *`trigger`* callback below must handle the corresponding (pause push/release) commands. The suspend/resume trigger commands can be defined even without the `RESUME` flag. See *Power Management* section for details.

  When the PCM substreams can be synchronized (typically, synchronized start/stop of a playback and a capture streams), you can give `SNDRV_PCM_INFO_SYNC_START`, too. In this case, you'll need to check the linked-list of PCM substreams in the trigger callback. This will be described in the later section.

- *`formats`* field contains the bit-flags of supported formats (`SNDRV_PCM_FMTBIT_XXX`). If the hardware supports more than one format, give all or'ed bits. In the example above, the signed 16bit little-endian format is specified.

- *`rates`* field contains the bit-flags of supported rates (`SNDRV_PCM_RATE_XXX`). When the chip supports

- *rates* field contains the bit-flags of supported rates (SNDRV_PCM_RATE_XXX). When the chip supports continuous rates, pass CONTINUOUS bit additionally. The pre-defined rate bits are provided only for typical rates. If your chip supports unconventional rates, you need to add the KNOT bit and set up the hardware constraint manually (explained later).

- *rate_min* and *rate_max* define the minimum and maximum sample rate. This should correspond somehow to *rates* bits.

- *channel_min* and *channel_max* define, as you might already expected, the minimum and maximum number of channels.

- *buffer_bytes_max* defines the maximum buffer size in bytes. There is no *buffer_bytes_min* field, since it can be calculated from the minimum period size and the minimum number of periods. Meanwhile, *period_bytes_min* and define the minimum and maximum size of the period in bytes. *periods_max* and *periods_min* define the maximum and minimum number of periods in the buffer.

  The “period” is a term that corresponds to a fragment in the OSS world. The period defines the size at which a PCM interrupt is generated. This size strongly depends on the hardware. Generally, the smaller period size will give you more interrupts, that is, more controls. In the case of capture, this size defines the input latency. On the other hand, the whole buffer size defines the output latency for the playback direction.

- There is also a field *fifo_size*. This specifies the size of the hardware FIFO, but currently it is neither used in the driver nor in the alsa-lib. So, you can ignore this field.

PCM Configurations

Ok, let's go back again to the PCM runtime records. The most frequently referred records in the runtime instance are the PCM configurations. The PCM configurations are stored in the runtime instance after the application sends hw_params data via alsa-lib. There are many fields copied from hw_params and sw_params structs. For example, *format* holds the format type chosen by the application. This field contains the enum value SNDRV_PCM_FORMAT_XXX.

One thing to be noted is that the configured buffer and period sizes are stored in “frames” in the runtime. In the ALSA world, 1 frame = channels * samples-size. For conversion between frames and bytes, you can use the frames_to_bytes() and bytes_to_frames() helper functions.

```
period_bytes = frames_to_bytes(runtime, runtime->period_size);
```

Also, many software parameters (sw_params) are stored in frames, too. Please check the type of the field. snd_pcm_uframes_t is for the frames as unsigned integer while snd_pcm_sframes_t is for the frames as signed integer.

DMA Buffer Information

The DMA buffer is defined by the following four fields, *dma_area*, *dma_addr*, *dma_bytes* and *dma_private*. The *dma_area* holds the buffer pointer (the logical address). You can call memcpy from/to this pointer. Meanwhile, *dma_addr* holds the physical address of the buffer. This field is specified only when the buffer is a linear buffer. *dma_bytes* holds the size of buffer in bytes. *dma_private* is used for the ALSA DMA allocator.

If you use a standard ALSA function, snd_pcm_lib_malloc_pages(), for allocating the buffer, these fields are set by the ALSA middle layer, and you should *not* change them by yourself. You can read them but not write them. On the other hand, if you want to allocate the buffer by yourself, you'll need to manage it in hw_params callback. At least, *dma_bytes* is mandatory. *dma_area* is necessary when the buffer is mmapped. If your driver doesn't support mmap, this field is not necessary. *dma_addr* is also optional. You can use *dma_private* as you like, too.

Running Status

The running status can be referred via `runtime->status`. This is the pointer to the struct snd_pcm_mmap_status record. For example, you can get the current DMA hardware pointer via `runtime->status->hw_ptr`.

The DMA application pointer can be referred via `runtime->control`, which points to the struct snd_pcm_mmap_control record. However, accessing directly to this value is not recommended.

Private Data

You can allocate a record for the substream and store it in `runtime->private_data`. Usually, this is done in *the open callback*. Don't mix this with `pcm->private_data`. The `pcm->private_data` usually points to the chip instance assigned statically at the creation of PCM, while the `runtime->private_data` points to a dynamic data structure created at the PCM open callback.

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
        struct my_pcm_data *data;
        ....
        data = kmalloc(sizeof(*data), GFP_KERNEL);
        substream->runtime->private_data = data;
        ....
}
```

The allocated object must be released in *the close callback*.

Interrupt Callbacks

The field *`transfer_ack_begin`* and *`transfer_ack_end`* are called at the beginning and at the end of `snd_pcm_period_elapsed()`, respectively.

Operators

OK, now let me give details about each pcm callback (*`ops`*). In general, every callback must return 0 if successful, or a negative error number such as `-EINVAL`. To choose an appropriate error number, it is advised to check what value other parts of the kernel return when the same kind of request fails.

The callback function takes at least the argument with snd_pcm_substream pointer. To retrieve the chip record from the given substream instance, you can use the following macro.

```
int xxx() {
        struct mychip *chip = snd_pcm_substream_chip(substream);
        ....
}
```

The macro reads `substream->private_data`, which is a copy of `pcm->private_data`. You can override the former if you need to assign different data records per PCM substream. For example, the cmi8330 driver assigns different private_data for playback and capture directions, because it uses two different codecs (SB- and AD-compatible) for different directions.

open callback

```
static int snd_xxx_open(struct snd_pcm_substream *substream);
```

This is called when a pcm substream is opened.

At least, here you have to initialize the runtime->hw record. Typically, this is done by like this:

```
static int snd_xxx_open(struct snd_pcm_substream *substream)
{
        struct mychip *chip = snd_pcm_substream_chip(substream);
        struct snd_pcm_runtime *runtime = substream->runtime;

        runtime->hw = snd_mychip_playback_hw;
        return 0;
}
```

where *snd_mychip_playback_hw* is the pre-defined hardware description.

You can allocate a private data in this callback, as described in *Private Data* section.

If the hardware configuration needs more constraints, set the hardware constraints here, too. See *Constraints* for more details.

close callback

```
static int snd_xxx_close(struct snd_pcm_substream *substream);
```

Obviously, this is called when a pcm substream is closed.

Any private instance for a pcm substream allocated in the open callback will be released here.

```
static int snd_xxx_close(struct snd_pcm_substream *substream)
{
        ....
        kfree(substream->runtime->private_data);
        ....
}
```

ioctl callback

This is used for any special call to pcm ioctls. But usually you can pass a generic ioctl callback, `snd_pcm_lib_ioctl`.

hw_params callback

```
static int snd_xxx_hw_params(struct snd_pcm_substream *substream,
                             struct snd_pcm_hw_params *hw_params);
```

This is called when the hardware parameter (*hw_params*) is set up by the application, that is, once when the buffer size, the period size, the format, etc. are defined for the pcm substream.

Many hardware setups should be done in this callback, including the allocation of buffers.

Parameters to be initialized are retrieved by `params_xxx()` macros. To allocate buffer, you can call a helper function,

```
snd_pcm_lib_malloc_pages(substream, params_buffer_bytes(hw_params));
```

`snd_pcm_lib_malloc_pages()` is available only when the DMA buffers have been pre-allocated. See the section *Buffer Types* for more details.

Note that this and *prepare* callbacks may be called multiple times per initialization. For example, the OSS emulation may call these callbacks at each change via its ioctl.

Thus, you need to be careful not to allocate the same buffers many times, which will lead to memory leaks! Calling the helper function above many times is OK. It will release the previous buffer automatically when it was already allocated.

Another note is that this callback is non-atomic (schedulable). This is important, because the *trigger* callback is atomic (non-schedulable). That is, mutexes or any schedule-related functions are not available in *trigger* callback. Please see the subsection *Atomicity* for details.

hw_free callback

```
static int snd_xxx_hw_free(struct snd_pcm_substream *substream);
```

This is called to release the resources allocated via *hw_params*. For example, releasing the buffer via `snd_pcm_lib_malloc_pages()` is done by calling the following:

```
snd_pcm_lib_free_pages(substream);
```

This function is always called before the close callback is called. Also, the callback may be called multiple times, too. Keep track whether the resource was already released.

prepare callback

```
static int snd_xxx_prepare(struct snd_pcm_substream *substream);
```

This callback is called when the pcm is “prepared”. You can set the format type, sample rate, etc. here. The difference from *hw_params* is that the *prepare* callback will be called each time `snd_pcm_prepare()` is called, i.e. when recovering after underruns, etc.

Note that this callback is now non-atomic. You can use schedule-related functions safely in this callback.

In this and the following callbacks, you can refer to the values via the runtime record, substream->runtime. For example, to get the current rate, format or channels, access to runtime->rate, runtime->format or runtime->channels, respectively. The physical address of the allocated buffer is set to runtime->dma_area. The buffer and period sizes are in runtime->buffer_size and runtime->period_size, respectively.

Be careful that this callback will be called many times at each setup, too.

trigger callback

```
static int snd_xxx_trigger(struct snd_pcm_substream *substream, int cmd);
```

This is called when the pcm is started, stopped or paused.

Which action is specified in the second argument, `SNDRV_PCM_TRIGGER_XXX` in `<sound/pcm.h>`. At least, the `START` and `STOP` commands must be defined in this callback.

```
switch (cmd) {
case SNDRV_PCM_TRIGGER_START:
        /* do something to start the PCM engine */
        break;
case SNDRV_PCM_TRIGGER_STOP:
        /* do something to stop the PCM engine */
        break;
default:
        return -EINVAL;
}
```

When the pcm supports the pause operation (given in the info field of the hardware table), the `PAUSE_PUSE` and `PAUSE_RELEASE` commands must be handled here, too. The former is the command to pause the pcm, and the latter to restart the pcm again.

When the pcm supports the suspend/resume operation, regardless of full or partial suspend/resume support, the `SUSPEND` and `RESUME` commands must be handled, too. These commands are issued when the power-management status is changed. Obviously, the `SUSPEND` and `RESUME` commands suspend and resume the pcm substream, and usually, they are identical to the `STOP` and `START` commands, respectively. See the *Power Management* section for details.

As mentioned, this callback is atomic. You cannot call functions which may sleep. The trigger callback should be as minimal as possible, just really triggering the DMA. The other stuff should be initialized hw_params and prepare callbacks properly beforehand.

pointer callback

```
static snd_pcm_uframes_t snd_xxx_pointer(struct snd_pcm_substream *substream)
```

This callback is called when the PCM middle layer inquires the current hardware position on the buffer. The position must be returned in frames, ranging from 0 to buffer_size - 1.

This is called usually from the buffer-update routine in the pcm middle layer, which is invoked when `snd_pcm_period_elapsed()` is called in the interrupt routine. Then the pcm middle layer updates the position and calculates the available space, and wakes up the sleeping poll threads, etc.

This callback is also atomic.

copy and silence callbacks

These callbacks are not mandatory, and can be omitted in most cases. These callbacks are used when the hardware buffer cannot be in the normal memory space. Some chips have their own buffer on the hardware which is not mappable. In such a case, you have to transfer the data manually from the memory buffer to the hardware buffer. Or, if the buffer is non-contiguous on both physical and virtual memory spaces, these callbacks must be defined, too.

If these two callbacks are defined, copy and set-silence operations are done by them. The detailed will be described in the later section *Buffer and Memory Management*.

ack callback

This callback is also not mandatory. This callback is called when the appl_ptr is updated in read or write operations. Some drivers like emu10k1-fx and cs46xx need to track the current appl_ptr for the internal buffer, and this callback is useful only for such a purpose.

This callback is atomic.

page callback

This callback is optional too. This callback is used mainly for non-contiguous buffers. The mmap calls this callback to get the page address. Some examples will be explained in the later section *Buffer and Memory Management*, too.

Interrupt Handler

The rest of pcm stuff is the PCM interrupt handler. The role of PCM interrupt handler in the sound driver is to update the buffer position and to tell the PCM middle layer when the buffer position goes across the prescribed period size. To inform this, call the `snd_pcm_period_elapsed()` function.

There are several types of sound chips to generate the interrupts.

Interrupts at the period (fragment) boundary

This is the most frequently found type: the hardware generates an interrupt at each period boundary. In this case, you can call `snd_pcm_period_elapsed()` at each interrupt.

`snd_pcm_period_elapsed()` takes the substream pointer as its argument. Thus, you need to keep the substream pointer accessible from the chip instance. For example, define substream field in the chip record to hold the current running substream pointer, and set the pointer value at open callback (and reset at close callback).

If you acquire a spinlock in the interrupt handler, and the lock is used in other pcm callbacks, too, then you have to release the lock before calling `snd_pcm_period_elapsed()`, because `snd_pcm_period_elapsed()` calls other pcm callbacks inside.

Typical code would be like:

Example 5.3. Interrupt Handler Case #1

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        spin_lock(&chip->lock);
        ....
        if (pcm_irq_invoked(chip)) {
                /* call updater, unlock before it */
                spin_unlock(&chip->lock);
                snd_pcm_period_elapsed(chip->substream);
                spin_lock(&chip->lock);
                /* acknowledge the interrupt if necessary */
        }
        ....
        spin_unlock(&chip->lock);
        return IRQ_HANDLED;
}
```

High frequency timer interrupts

This happense when the hardware doesn't generate interrupts at the period boundary but issues timer interrupts at a fixed timer rate (e.g. es1968 or ymfpci drivers). In this case, you need to check the current hardware position and accumulate the processed sample length at each interrupt. When the accumulated size exceeds the period size, call `snd_pcm_period_elapsed()` and reset the accumulator.

Typical code would be like the following.

Example 5.4. Interrupt Handler Case #2

```
static irqreturn_t snd_mychip_interrupt(int irq, void *dev_id)
{
        struct mychip *chip = dev_id;
        spin_lock(&chip->lock);
        ....
        if (pcm_irq_invoked(chip)) {
                unsigned int last_ptr, size;
                /* get the current hardware pointer (in frames) */
                last_ptr = get_hw_ptr(chip);
                /* calculate the processed frames since the
                 * last update
                 */
                if (last_ptr < chip->last_ptr)
                        size = runtime->buffer_size + last_ptr
                                 - chip->last_ptr;
                else
                        size = last_ptr - chip->last_ptr;
                /* remember the last updated point */
                chip->last_ptr = last_ptr;
                /* accumulate the size */
                chip->size += size;
                /* over the period boundary? */
                if (chip->size >= runtime->period_size) {
                        /* reset the accumulator */
                        chip->size %= runtime->period_size;
                        /* call updater */
                        spin_unlock(&chip->lock);
                        snd_pcm_period_elapsed(substream);
                        spin_lock(&chip->lock);
                }
                /* acknowledge the interrupt if necessary */
        }
        ....
        spin_unlock(&chip->lock);
        return IRQ_HANDLED;
}
```

On calling `snd_pcm_period_elapsed()`

In both cases, even if more than one period are elapsed, you don't have to call `snd_pcm_period_elapsed()` many times. Call only once. And the pcm layer will check the current hardware pointer and update to the latest status.

Atomicity

One of the most important (and thus difficult to debug) problems in kernel programming are race conditions. In the Linux kernel, they are usually avoided via spin-locks, mutexes or semaphores. In general, if a race condition can happen in an interrupt handler, it has to be managed atomically, and you have to use a spinlock to protect the critical session. If the critical section is not in interrupt handler code and if taking a relatively long time to execute is acceptable, you should use mutexes or semaphores instead.

As already seen, some pcm callbacks are atomic and some are not. For example, the *`hw_params`* callback is non-atomic, while *`trigger`* callback is atomic. This means, the latter is called already in a spinlock held by the PCM middle layer. Please take this atomicity into account when you choose a locking scheme in the callbacks.

In the atomic callbacks, you cannot use functions which may call `schedule` or go to `sleep`. Semaphores and mutexes can sleep, and hence they cannot be used inside the atomic callbacks (e.g. *`trigger`* callback). To implement some delay in such a callback, please use `udelay()` or `mdelay()`.

All three atomic callbacks (trigger, pointer, and ack) are called with local interrupts disabled.

Constraints

If your chip supports unconventional sample rates, or only the limited samples, you need to set a constraint for the condition.

For example, in order to restrict the sample rates in the some supported values, use `snd_pcm_hw_constraint_list()`. You need to call this function in the open callback.

Example 5.5. Example of Hardware Constraints

```
static unsigned int rates[] =
        {4000, 10000, 22050, 44100};
static struct snd_pcm_hw_constraint_list constraints_rates = {
        .count = ARRAY_SIZE(rates),
        .list = rates,
        .mask = 0,
};

static int snd_mychip_pcm_open(struct snd_pcm_substream *substream)
{
        int err;
        ....
        err = snd_pcm_hw_constraint_list(substream->runtime, 0,
                                         SNDRV_PCM_HW_PARAM_RATE,
                                         &constraints_rates);
        if (err < 0)
                return err;
```