# Writing an ALSA Driver

**Takashi Iwai**

# Writing an ALSA Driver

Takashi Iwai

## Abstract

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

# Table of Contents

# List of Examples

# Preface

This document describes how to write an *ALSA (Advanced Linux Sound Architecture)* [http://www.alsa-project.org/] 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

## 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.

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

## 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**

```
/* 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);
```

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

## 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 {
        ....
};
```

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

## 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**

```
        }
        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_28BIT_MASK) < 0 ||
    pci_set_consistent_dma_mask(pci, DMA_28BIT_MASK) < 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

## General

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.

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

**Example 5-1. PCM Example Code**

```
/* 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 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

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 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;
        ....
}
```

There are many different constraints. Look at `sound/pcm.h` for a complete list. You can even define your own constraint rules. For example, let's suppose my_chip can manage a substream of 1 channel if and only if the format is S16_LE, otherwise it supports any format specified in the snd_pcm_hardware structure (or in any other constraint_list). You can build a rule like this:

**Example 5.6. Example of Hardware Constraints for Channels**

```
static int hw_rule_format_by_channels(struct snd_pcm_hw_params *params,
                                      struct snd_pcm_hw_rule *rule)
{
        struct snd_interval *c = hw_param_interval(params,
              SNDRV_PCM_HW_PARAM_CHANNELS);
        struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
        struct snd_mask fmt;

        snd_mask_any(&fmt);    /* Init the struct */
        if (c->min < 2) {
                fmt.bits[0] &= SNDRV_PCM_FMTBIT_S16_LE;
                return snd_mask_refine(f, &fmt);
        }
        return 0;
}
```

Then you need to call this function to add your rule:

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_CHANNELS,
                    hw_rule_channels_by_format, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                    -1);
```

The rule function is called when an application sets the number of channels. But an application can set the format before the number of channels. Thus you also need to define the inverse rule:

**Example 5.7. Example of Hardware Constraints for Channels**

```
static int hw_rule_channels_by_format(struct snd_pcm_hw_params *params,
                                      struct snd_pcm_hw_rule *rule)
{
        struct snd_interval *c = hw_param_interval(params,
                      SNDRV_PCM_HW_PARAM_CHANNELS);
        struct snd_mask *f = hw_param_mask(params, SNDRV_PCM_HW_PARAM_FORMAT);
        struct snd_interval ch;

        snd_interval_any(&ch);
        if (f->bits[0] == SNDRV_PCM_FMTBIT_S16_LE) {
                ch.min = ch.max = 1;
                ch.integer = 1;
                return snd_interval_refine(c, &ch);
        }
        return 0;
}
```

...and in the open callback:

```
snd_pcm_hw_rule_add(substream->runtime, 0, SNDRV_PCM_HW_PARAM_FORMAT,
                    hw_rule_format_by_channels, 0, SNDRV_PCM_HW_PARAM_CHANNELS,
                    -1);
```

I won't give more details here, rather I would like to say, “Luke, use the source.”

# Chapter 6. Control Interface

## General

The control interface is used widely for many switches, sliders, etc. which are accessed from user-space. Its most important use is the mixer interface. In other words, since ALSA 0.9.x, all the mixer stuff is implemented on the control kernel API.

ALSA has a well-defined AC97 control module. If your chip supports only the AC97 and nothing else, you can skip this section.

The control API is defined in `<sound/control.h>`. Include this file if you want to add your own controls.

## Definition of Controls

To create a new control, you need to define the following three callbacks: *`info`*, *`get`* and *`put`*. Then, define a struct snd_kcontrol_new record, such as:

### Example 6.1. Definition of a Control

```
static struct snd_kcontrol_new my_control __devinitdata = {
        .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
};
```

Most likely the control is created via `snd_ctl_new1()`, and in such a case, you can add the *`__devinitdata`* prefix to the definition as above.

The *`iface`* field specifies the control type, `SNDRV_CTL_ELEM_IFACE_XXX`, which is usually `MIXER`. Use `CARD` for global controls that are not logically part of the mixer. If the control is closely associated with some specific device on the sound card, use `HWDEP`, `PCM`, `RAWMIDI`, `TIMER`, or `SEQUENCER`, and specify the device number with the *`device`* and *`subdevice`* fields.

The *`name`* is the name identifier string. Since ALSA 0.9.x, the control name is very important, because its role is classified from its name. There are pre-defined standard control names. The details are described in the *Control Names* subsection.

The *`index`* field holds the index number of this control. If there are several different controls with the same name, they can be distinguished by the index number. This is the case when several codecs exist on the card. If the index is zero, you can omit the definition above.

The *access* field contains the access type of this control. Give the combination of bit masks, `SNDRV_CTL_ELEM_ACCESS_XXX`, there. The details will be explained in the *Access Flags* subsection.

The *private_value* field contains an arbitrary long integer value for this record. When using the generic *info*, *get* and *put* callbacks, you can pass a value through this field. If several small numbers are necessary, you can combine them in bitwise. Or, it's possible to give a pointer (casted to unsigned long) of some record to this field, too.

The *tlv* field can be used to provide metadata about the control; see the *Metadata* subsection.

The other three are *callback functions*.

# Control Names

There are some standards to define the control names. A control is usually defined from the three parts as “SOURCE DIRECTION FUNCTION”.

The first, `SOURCE`, specifies the source of the control, and is a string such as “Master”, “PCM”, “CD” and “Line”. There are many pre-defined sources.

The second, `DIRECTION`, is one of the following strings according to the direction of the control: “Playback”, “Capture”, “Bypass Playback” and “Bypass Capture”. Or, it can be omitted, meaning both playback and capture directions.

The third, `FUNCTION`, is one of the following strings according to the function of the control: “Switch”, “Volume” and “Route”.

The example of control names are, thus, “Master Capture Switch” or “PCM Playback Volume”.

There are some exceptions:

## Global capture and playback

“Capture Source”, “Capture Switch” and “Capture Volume” are used for the global capture (input) source, switch and volume. Similarly, “Playback Switch” and “Playback Volume” are used for the global output gain switch and volume.

## Tone-controls

tone-control switch and volumes are specified like “Tone Control - XXX”, e.g. “Tone Control - Switch”, “Tone Control - Bass”, “Tone Control - Center”.

## 3D controls

3D-control switches and volumes are specified like “3D Control - XXX”, e.g. “3D Control - Switch”, “3D Control - Center”, “3D Control - Space”.

## Mic boost

Mic-boost switch is set as “Mic Boost” or “Mic Boost (6dB)”.

More precise information can be found in `Documentation/sound/alsa/ControlNames.txt`.

# Access Flags

The access flag is the bitmask which specifies the access type of the given control. The default access type is `SNDRV_CTL_ELEM_ACCESS_READWRITE`, which means both read and write are allowed to this control. When the access flag is omitted (i.e. = 0), it is considered as `READWRITE` access as default.

When the control is read-only, pass `SNDRV_CTL_ELEM_ACCESS_READ` instead. In this case, you don't have to define the *`put`* callback. Similarly, when the control is write-only (although it's a rare case), you can use the `WRITE` flag instead, and you don't need the *`get`* callback.

If the control value changes frequently (e.g. the VU meter), `VOLATILE` flag should be given. This means that the control may be changed without *notification*. Applications should poll such a control constantly.

When the control is inactive, set the `INACTIVE` flag, too. There are `LOCK` and `OWNER` flags to change the write permissions.

# Callbacks

## info callback

The *`info`* callback is used to get detailed information on this control. This must store the values of the given struct snd_ctl_elem_info object. For example, for a boolean control with a single element:

### Example 6.2. Example of info callback

```
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;
}
```

The *`type`* field specifies the type of the control. There are `BOOLEAN`, `INTEGER`, `ENUMERATED`, `BYTES`, `IEC958` and `INTEGER64`. The *`count`* field specifies the number of elements in this control. For example, a stereo volume would have count = 2. The *`value`* field is a union, and the values stored are depending on the type. The boolean and integer types are identical.

The enumerated type is a bit different from others. You'll need to set the string for the currently given item index.

```
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;
}
```

Some common info callbacks are available for your convenience: `snd_ctl_boolean_mono_info()` and `snd_ctl_boolean_stereo_info()`. Obviously, the former is an info callback for a mono channel boolean item, just like `snd_myctl_mono_info` above, and the latter is for a stereo channel boolean item.

# get callback

This callback is used to read the current value of the control and to return to user-space.

For example,

### Example 6.3. Example of 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;
}
```

The *`value`* field depends on the type of control as well as on the info callback. For example, the sb driver uses this field to store the register offset, the bit-shift and the bit-mask. The *`private_value`* field is set as follows:

```
.private_value = reg | (shift << 16) | (mask << 24)
```

and is retrieved in callbacks like

```
static int snd_sbmixer_get_single(struct snd_kcontrol *kcontrol,
                                  struct snd_ctl_elem_value *ucontrol)
```

```
{
        int reg = kcontrol->private_value & 0xff;
        int shift = (kcontrol->private_value >> 16) & 0xff;
        int mask = (kcontrol->private_value >> 24) & 0xff;
        ....
}
```

In the *get* callback, you have to fill all the elements if the control has more than one elements, i.e. *count* > 1. In the example above, we filled only one element (*value.integer.value[0]*) since it's assumed as *count* = 1.

## put callback

This callback is used to write a value from user-space.

For example,

**Example 6.4. Example of 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;
}
```

As seen above, you have to return 1 if the value is changed. If the value is not changed, return 0 instead. If any fatal error happens, return a negative error code as usual.

As in the *get* callback, when the control has more than one elements, all elements must be evaluated in this callback, too.

## Callbacks are not atomic

All these three callbacks are basically not atomic.

# Constructor

When everything is ready, finally we can create a new control. To create a control, there are two functions to be called, `snd_ctl_new1()` and `snd_ctl_add()`.

In the simplest way, you can do like this:

```
err = snd_ctl_add(card, snd_ctl_new1(&my_control, chip));
if (err < 0)
        return err;
```

where *my_control* is the struct snd_kcontrol_new object defined above, and chip is the object pointer to be passed to kcontrol->private_data which can be referred to in callbacks.

`snd_ctl_new1()` allocates a new snd_kcontrol instance (that's why the definition of *my_control* can be with the *__devinitdata* prefix), and `snd_ctl_add` assigns the given control component to the card.

# Change Notification

If you need to change and update a control in the interrupt routine, you can call `snd_ctl_notify()`. For example,

```
snd_ctl_notify(card, SNDRV_CTL_EVENT_MASK_VALUE, id_pointer);
```

This function takes the card pointer, the event-mask, and the control id pointer for the notification. The event-mask specifies the types of notification, for example, in the above example, the change of control values is notified. The id pointer is the pointer of struct snd_ctl_elem_id to be notified. You can find some examples in `es1938.c` or `es1968.c` for hardware volume interrupts.

# Metadata

To provide information about the dB values of a mixer control, use on of the `DECLARE_TLV_xxx` macros from `<sound/tlv.h>` to define a variable containing this information, set the *tlv.p* field to point to this variable, and include the `SNDRV_CTL_ELEM_ACCESS_TLV_READ` flag in the *access* field; like this:

```
static DECLARE_TLV_DB_SCALE(db_scale_my_control, -4050, 150, 0);

static struct snd_kcontrol_new my_control __devinitdata = {
        ...
        .access = SNDRV_CTL_ELEM_ACCESS_READWRITE |
                  SNDRV_CTL_ELEM_ACCESS_TLV_READ,
        ...
        .tlv.p = db_scale_my_control,
};
```

The DECLARE_TLV_DB_SCALE macro defines information about a mixer control where each step in the control's value changes the dB value by a constant dB amount. The first parameter is the name of the variable to be defined. The second parameter is the minimum value, in units of 0.01 dB. The third parameter is the step size, in units of 0.01 dB. Set the fourth parameter to 1 if the minimum value actually mutes the control.

The DECLARE_TLV_DB_LINEAR macro defines information about a mixer control where the control's value affects the output linearly. The first parameter is the name of the variable to be defined. The second parameter is the minimum value, in units of 0.01 dB. The third parameter is the maximum value, in units of 0.01 dB. If the minimum value mutes the control, set the second parameter to TLV_DB_GAIN_MUTE.

# Chapter 7. API for AC97 Codec

## General

The ALSA AC97 codec layer is a well-defined one, and you don't have to write much code to control it. Only low-level control routines are necessary. The AC97 codec API is defined in `<sound/ac97_codec.h>`.

# Full Code Example

### Example 7.1. Example of AC97 Interface

```
struct mychip {
        ....
        struct snd_ac97 *ac97;
        ....
};

static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                           unsigned short reg)
{
        struct mychip *chip = ac97->private_data;
        ....
        /* read a register value here from the codec */
        return the_register_value;
}

static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                                 unsigned short reg, unsigned short val)
{
        struct mychip *chip = ac97->private_data;
        ....
        /* write the given register value to the codec */
}

static int snd_mychip_ac97(struct mychip *chip)
{
        struct snd_ac97_bus *bus;
        struct snd_ac97_template ac97;
        int err;
        static struct snd_ac97_bus_ops ops = {
                .write = snd_mychip_ac97_write,
                .read = snd_mychip_ac97_read,
        };

        err = snd_ac97_bus(chip->card, 0, &ops, NULL, &bus);
        if (err < 0)
                return err;
        memset(&ac97, 0, sizeof(ac97));
        ac97.private_data = chip;
        return snd_ac97_mixer(bus, &ac97, &chip->ac97);
}
```

# Constructor

To create an ac97 instance, first call `snd_ac97_bus` with an ac97_bus_ops_t record with callback functions.

```
struct snd_ac97_bus *bus;
static struct snd_ac97_bus_ops ops = {
      .write = snd_mychip_ac97_write,
      .read = snd_mychip_ac97_read,
};

snd_ac97_bus(card, 0, &ops, NULL, &pbus);
```

The bus record is shared among all belonging ac97 instances.

And then call `snd_ac97_mixer()` with an struct snd_ac97_template record together with the bus pointer created above.

```
struct snd_ac97_template ac97;
int err;

memset(&ac97, 0, sizeof(ac97));
ac97.private_data = chip;
snd_ac97_mixer(bus, &ac97, &chip->ac97);
```

where chip->ac97 is a pointer to a newly created ac97_t instance. In this case, the chip pointer is set as the private data, so that the read/write callback functions can refer to this chip instance. This instance is not necessarily stored in the chip record. If you need to change the register values from the driver, or need the suspend/resume of ac97 codecs, keep this pointer to pass to the corresponding functions.

# Callbacks

The standard callbacks are *`read`* and *`write`*. Obviously they correspond to the functions for read and write accesses to the hardware low-level codes.

The *`read`* callback returns the register value specified in the argument.

```
static unsigned short snd_mychip_ac97_read(struct snd_ac97 *ac97,
                                            unsigned short reg)
{
        struct mychip *chip = ac97->private_data;
        ....
        return the_register_value;
}
```

Here, the chip can be cast from ac97->private_data.

Meanwhile, the *write* callback is used to set the register value.

```
static void snd_mychip_ac97_write(struct snd_ac97 *ac97,
                     unsigned short reg, unsigned short val)
```

These callbacks are non-atomic like the control API callbacks.

There are also other callbacks: *reset*, *wait* and *init*.

The *reset* callback is used to reset the codec. If the chip requires a special kind of reset, you can define this callback.

The *wait* callback is used to add some waiting time in the standard initialization of the codec. If the chip requires the extra waiting time, define this callback.

The *init* callback is used for additional initialization of the codec.

# Updating Registers in The Driver

If you need to access to the codec from the driver, you can call the following functions: `snd_ac97_write()`, `snd_ac97_read()`, `snd_ac97_update()` and `snd_ac97_update_bits()`.

Both `snd_ac97_write()` and `snd_ac97_update()` functions are used to set a value to the given register (`AC97_XXX`). The difference between them is that `snd_ac97_update()` doesn't write a value if the given value has been already set, while `snd_ac97_write()` always rewrites the value.

```
snd_ac97_write(ac97, AC97_MASTER, 0x8080);
snd_ac97_update(ac97, AC97_MASTER, 0x8080);
```

`snd_ac97_read()` is used to read the value of the given register. For example,

```
value = snd_ac97_read(ac97, AC97_MASTER);
```

`snd_ac97_update_bits()` is used to update some bits in the given register.

```
snd_ac97_update_bits(ac97, reg, mask, value);
```

Also, there is a function to change the sample rate (of a given register such as AC97_PCM_FRONT_DAC_RATE) when VRA or DRA is supported by the codec: snd_ac97_set_rate().

```
snd_ac97_set_rate(ac97, AC97_PCM_FRONT_DAC_RATE, 44100);
```

The following registers are available to set the rate: AC97_PCM_MIC_ADC_RATE, AC97_PCM_FRONT_DAC_RATE, AC97_PCM_LR_ADC_RATE, AC97_SPDIF. When AC97_SPDIF is specified, the register is not really changed but the corresponding IEC958 status bits will be updated.

# Clock Adjustment

In some chips, the clock of the codec isn't 48000 but using a PCI clock (to save a quartz!). In this case, change the field bus->clock to the corresponding value. For example, intel8x0 and es1968 drivers have their own function to read from the clock.

# Proc Files

The ALSA AC97 interface will create a proc file such as /proc/asound/card0/codec97#0/ac97#0-0 and ac97#0-0+regs. You can refer to these files to see the current status and registers of the codec.

# Multiple Codecs

When there are several codecs on the same card, you need to call snd_ac97_mixer() multiple times with ac97.num=1 or greater. The *num* field specifies the codec number.

If you set up multiple codecs, you either need to write different callbacks for each codec or check ac97->num in the callback routines.

# Chapter 8. MIDI (MPU401-UART) Interface

## General

Many soundcards have built-in MIDI (MPU401-UART) interfaces. When the soundcard supports the standard MPU401-UART interface, most likely you can use the ALSA MPU401-UART API. The MPU401-UART API is defined in `<sound/mpu401.h>`.

Some soundchips have a similar but slightly different implementation of mpu401 stuff. For example, emu10k1 has its own mpu401 routines.

## Constructor

To create a rawmidi object, call `snd_mpu401_uart_new()`.

```
struct snd_rawmidi *rmidi;
snd_mpu401_uart_new(card, 0, MPU401_HW_MPU401, port, info_flags,
                    irq, irq_flags, &rmidi);
```

The first argument is the card pointer, and the second is the index of this component. You can create up to 8 rawmidi devices.

The third argument is the type of the hardware, `MPU401_HW_XXX`. If it's not a special one, you can use `MPU401_HW_MPU401`.

The 4th argument is the I/O port address. Many backward-compatible MPU401 have an I/O port such as 0x330. Or, it might be a part of its own PCI I/O region. It depends on the chip design.

The 5th argument is a bitflag for additional information. When the I/O port address above is part of the PCI I/O region, the MPU401 I/O port might have been already allocated (reserved) by the driver itself. In such a case, pass a bit flag `MPU401_INFO_INTEGRATED`, and the mpu401-uart layer will allocate the I/O ports by itself.

When the controller supports only the input or output MIDI stream, pass the `MPU401_INFO_INPUT` or `MPU401_INFO_OUTPUT` bitflag, respectively. Then the rawmidi instance is created as a single stream.

`MPU401_INFO_MMIO` bitflag is used to change the access method to MMIO (via readb and writeb) instead of iob and outb. In this case, you have to pass the iomapped address to `snd_mpu401_uart_new()`.

When `MPU401_INFO_TX_IRQ` is set, the output stream isn't checked in the default interrupt handler. The driver needs to call `snd_mpu401_uart_interrupt_tx()` by itself to start processing the output stream in the irq handler.

Usually, the port address corresponds to the command port and port + 1 corresponds to the data port. If not, you may change the *`cport`* field of struct snd_mpu401 manually afterward. However, snd_mpu401 pointer is not returned explicitly by `snd_mpu401_uart_new()`. You need to cast rmidi->private_data to snd_mpu401 explicitly,

```
struct snd_mpu401 *mpu;
mpu = rmidi->private_data;
```

and reset the cport as you like:

```
mpu->cport = my_own_control_port;
```

The 6th argument specifies the irq number for UART. If the irq is already allocated, pass 0 to the 7th argument (*irq_flags*). Otherwise, pass the flags for irq allocation (`SA_XXX` bits) to it, and the irq will be reserved by the mpu401-uart layer. If the card doesn't generate UART interrupts, pass -1 as the irq number. Then a timer interrupt will be invoked for polling.

# Interrupt Handler

When the interrupt is allocated in `snd_mpu401_uart_new()`, the private interrupt handler is used, hence you don't have anything else to do than creating the mpu401 stuff. Otherwise, you have to call `snd_mpu401_uart_interrupt()` explicitly when a UART interrupt is invoked and checked in your own interrupt handler.

In this case, you need to pass the private_data of the returned rawmidi object from `snd_mpu401_uart_new()` as the second argument of `snd_mpu401_uart_interrupt()`.

```
snd_mpu401_uart_interrupt(irq, rmidi->private_data, regs);
```

# Chapter 9. RawMIDI Interface

## Overview

The raw MIDI interface is used for hardware MIDI ports that can be accessed as a byte stream. It is not used for synthesizer chips that do not directly understand MIDI.

ALSA handles file and buffer management. All you have to do is to write some code to move data between the buffer and the hardware.

The rawmidi API is defined in `<sound/rawmidi.h>`.

## Constructor

To create a rawmidi device, call the `snd_rawmidi_new` function:

```
struct snd_rawmidi *rmidi;
err = snd_rawmidi_new(chip->card, "MyMIDI", 0, outs, ins, &rmidi);
if (err < 0)
        return err;
rmidi->private_data = chip;
strcpy(rmidi->name, "My MIDI");
rmidi->info_flags = SNDRV_RAWMIDI_INFO_OUTPUT |
                    SNDRV_RAWMIDI_INFO_INPUT |
                    SNDRV_RAWMIDI_INFO_DUPLEX;
```

The first argument is the card pointer, the second argument is the ID string.

The third argument is the index of this component. You can create up to 8 rawmidi devices.

The fourth and fifth arguments are the number of output and input substreams, respectively, of this device (a substream is the equivalent of a MIDI port).

Set the *`info_flags`* field to specify the capabilities of the device. Set `SNDRV_RAWMIDI_INFO_OUTPUT` if there is at least one output port, `SNDRV_RAWMIDI_INFO_INPUT` if there is at least one input port, and `SNDRV_RAWMIDI_INFO_DUPLEX` if the device can handle output and input at the same time.

After the rawmidi device is created, you need to set the operators (callbacks) for each substream. There are helper functions to set the operators for all the substreams of a device:

```
snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_OUTPUT, &snd_mymidi_output_ops);
snd_rawmidi_set_ops(rmidi, SNDRV_RAWMIDI_STREAM_INPUT, &snd_mymidi_input_ops);
```

The operators are usually defined like this:

```
static struct snd_rawmidi_ops snd_mymidi_output_ops = {
        .open =    snd_mymidi_output_open,
        .close =   snd_mymidi_output_close,
        .trigger = snd_mymidi_output_trigger,
};
```

These callbacks are explained in the *Callbacks* section.

If there are more than one substream, you should give a unique name to each of them:

```
struct snd_rawmidi_substream *substream;
list_for_each_entry(substream,
                    &rmidi->streams[SNDRV_RAWMIDI_STREAM_OUTPUT].substreams,
                    list {
        sprintf(substream->name, "My MIDI Port %d", substream->number + 1);
}
/* same for SNDRV_RAWMIDI_STREAM_INPUT */
```

# Callbacks

In all the callbacks, the private data that you've set for the rawmidi device can be accessed as substream->rmidi->private_data.

If there is more than one port, your callbacks can determine the port index from the struct snd_rawmidi_substream data passed to each callback:

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

## open callback

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

This is called when a substream is opened. You can initialize the hardware here, but you shouldn't start transmitting/receiving data yet.

# `close` callback

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

Guess what.

The `open` and `close` callbacks of a rawmidi device are serialized with a mutex, and can sleep.

# `trigger` callback for output substreams

```
static void snd_xxx_output_trigger(struct snd_rawmidi_substream *substream, int
```

This is called with a nonzero *up* parameter when there is some data in the substream buffer that must be transmitted.

To read data from the buffer, call `snd_rawmidi_transmit_peek`. It will return the number of bytes that have been read; this will be less than the number of bytes requested when there are no more data in the buffer. After the data have been transmitted successfully, call `snd_rawmidi_transmit_ack` to remove the data from the substream buffer:

```
unsigned char data;
while (snd_rawmidi_transmit_peek(substream, &data, 1) == 1) {
        if (snd_mychip_try_to_transmit(data))
                snd_rawmidi_transmit_ack(substream, 1);
        else
                break; /* hardware FIFO full */
}
```

If you know beforehand that the hardware will accept data, you can use the `snd_rawmidi_transmit` function which reads some data and removes them from the buffer at once:

```
while (snd_mychip_transmit_possible()) {
        unsigned char data;
        if (snd_rawmidi_transmit(substream, &data, 1) != 1)
                break; /* no more data */
        snd_mychip_transmit(data);
}
```

If you know beforehand how many bytes you can accept, you can use a buffer size greater than one with the `snd_rawmidi_transmit*` functions.

The `trigger` callback must not sleep. If the hardware FIFO is full before the substream buffer has been emptied, you have to continue transmitting data later, either in an interrupt handler, or with a timer if the hardware doesn't have a MIDI transmit interrupt.

The `trigger` callback is called with a zero *up* parameter when the transmission of data should be aborted.

## `trigger` callback for input substreams

```
static void snd_xxx_input_trigger(struct snd_rawmidi_substream *substream, int u
```

This is called with a nonzero *up* parameter to enable receiving data, or with a zero *up* parameter do disable receiving data.

The `trigger` callback must not sleep; the actual reading of data from the device is usually done in an interrupt handler.

When data reception is enabled, your interrupt handler should call `snd_rawmidi_receive` for all received data:

```
void snd_mychip_midi_interrupt(...)
{
        while (mychip_midi_available()) {
                unsigned char data;
                data = mychip_midi_read();
                snd_rawmidi_receive(substream, &data, 1);
        }
}
```

## `drain` callback

```
static void snd_xxx_drain(struct snd_rawmidi_substream *substream);
```

This is only used with output substreams. This function should wait until all data read from the substream buffer have been transmitted. This ensures that the device can be closed and the driver unloaded without losing data.

This callback is optional. If you do not set *drain* in the struct snd_rawmidi_ops structure, ALSA will simply wait for 50 milliseconds instead.

# Chapter 10. Miscellaneous Devices

## FM OPL3

The FM OPL3 is still used in many chips (mainly for backward compatibility). ALSA has a nice OPL3 FM control layer, too. The OPL3 API is defined in `<sound/opl3.h>`.

FM registers can be directly accessed through the direct-FM API, defined in `<sound/asound_fm.h>`. In ALSA native mode, FM registers are accessed through the Hardware-Dependant Device direct-FM extension API, whereas in OSS compatible mode, FM registers can be accessed with the OSS direct-FM compatible API in `/dev/dmfmX` device.

To create the OPL3 component, you have two functions to call. The first one is a constructor for the opl3_t instance.

```
struct snd_opl3 *opl3;
snd_opl3_create(card, lport, rport, OPL3_HW_OPL3_XXX,
                integrated, &opl3);
```

The first argument is the card pointer, the second one is the left port address, and the third is the right port address. In most cases, the right port is placed at the left port + 2.

The fourth argument is the hardware type.

When the left and right ports have been already allocated by the card driver, pass non-zero to the fifth argument (*`integrated`*). Otherwise, the opl3 module will allocate the specified ports by itself.

When the accessing the hardware requires special method instead of the standard I/O access, you can create opl3 instance separately with `snd_opl3_new()`.

```
struct snd_opl3 *opl3;
snd_opl3_new(card, OPL3_HW_OPL3_XXX, &opl3);
```

Then set *`command`*, *`private_data`* and *`private_free`* for the private access function, the private data and the destructor. The l_port and r_port are not necessarily set. Only the command must be set properly. You can retrieve the data from the opl3->private_data field.

After creating the opl3 instance via `snd_opl3_new()`, call `snd_opl3_init()` to initialize the chip to the proper state. Note that `snd_opl3_create()` always calls it internally.

If the opl3 instance is created successfully, then create a hwdep device for this opl3.

```
struct snd_hwdep *opl3hwdep;
snd_opl3_hwdep_new(opl3, 0, 1, &opl3hwdep);
```

The first argument is the opl3_t instance you created, and the second is the index number, usually 0.

The third argument is the index-offset for the sequencer client assigned to the OPL3 port. When there is an MPU401-UART, give 1 for here (UART always takes 0).

# Hardware-Dependent Devices

Some chips need user-space access for special controls or for loading the micro code. In such a case, you can create a hwdep (hardware-dependent) device. The hwdep API is defined in `<sound/hwdep.h>`. You can find examples in opl3 driver or `isa/sb/sb16_csp.c`.

The creation of the hwdep instance is done via `snd_hwdep_new()`.

```
struct snd_hwdep *hw;
snd_hwdep_new(card, "My HWDEP", 0, &hw);
```

where the third argument is the index number.

You can then pass any pointer value to the *`private_data`*. If you assign a private data, you should define the destructor, too. The destructor function is set in the *`private_free`* field.

```
struct mydata *p = kmalloc(sizeof(*p), GFP_KERNEL);
hw->private_data = p;
hw->private_free = mydata_free;
```

and the implementation of the destructor would be:

```
static void mydata_free(struct snd_hwdep *hw)
{
        struct mydata *p = hw->private_data;
        kfree(p);
}
```

The arbitrary file operations can be defined for this instance. The file operators are defined in the *`ops`* table. For example, assume that this chip needs an ioctl.

```
hw->ops.open = mydata_open;
hw->ops.ioctl = mydata_ioctl;
```

```
hw->ops.release = mydata_release;
```

And implement the callback functions as you like.

# IEC958 (S/PDIF)

Usually the controls for IEC958 devices are implemented via the control interface. There is a macro to compose a name string for IEC958 controls, `SNDRV_CTL_NAME_IEC958()` defined in `<include/asound.h>`.

There are some standard controls for IEC958 status bits. These controls use the type SNDRV_CTL_ELEM_TYPE_IEC958, and the size of element is fixed as 4 bytes array (value.iec958.status[x]). For the *`info`* callback, you don't specify the value field for this type (the count field must be set, though).

“IEC958 Playback Con Mask” is used to return the bit-mask for the IEC958 status bits of consumer mode. Similarly, “IEC958 Playback Pro Mask” returns the bitmask for professional mode. They are read-only controls, and are defined as MIXER controls (iface = `SNDRV_CTL_ELEM_IFACE_MIXER`).

Meanwhile, “IEC958 Playback Default” control is defined for getting and setting the current default IEC958 bits. Note that this one is usually defined as a PCM control (iface = `SNDRV_CTL_ELEM_IFACE_PCM`), although in some places it's defined as a MIXER control.

In addition, you can define the control switches to enable/disable or to set the raw bit mode. The implementation will depend on the chip, but the control should be named as “IEC958 xxx”, preferably using the `SNDRV_CTL_NAME_IEC958()` macro.

You can find several cases, for example, `pci/emu10k1`, `pci/ice1712`, or `pci/cmipci.c`.

# Chapter 11. Buffer and Memory Management

## Buffer Types

ALSA provides several different buffer allocation functions depending on the bus and the architecture. All these have a consistent API. The allocation of physically-contiguous pages is done via `snd_malloc_xxx_pages()` function, where xxx is the bus type.

The allocation of pages with fallback is `snd_malloc_xxx_pages_fallback()`. This function tries to allocate the specified pages but if the pages are not available, it tries to reduce the page sizes until enough space is found.

The release the pages, call `snd_free_xxx_pages()` function.

Usually, ALSA drivers try to allocate and reserve a large contiguous physical space at the time the module is loaded for the later use. This is called "pre-allocation". As already written, you can call the following function at pcm instance construction time (in the case of PCI bus).

```
snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                       snd_dma_pci_data(pci), size, max);
```

where *`size`* is the byte size to be pre-allocated and the *`max`* is the maximum size to be changed via the `prealloc` proc file. The allocator will try to get an area as large as possible within the given size.

The second argument (type) and the third argument (device pointer) are dependent on the bus. In the case of the ISA bus, pass `snd_dma_isa_data()` as the third argument with `SNDRV_DMA_TYPE_DEV` type. For the continuous buffer unrelated to the bus can be pre-allocated with `SNDRV_DMA_TYPE_CONTINUOUS` type and the `snd_dma_continuous_data(GFP_KERNEL)` device pointer, where `GFP_KERNEL` is the kernel allocation flag to use. For the PCI scatter-gather buffers, use `SNDRV_DMA_TYPE_DEV_SG` with `snd_dma_pci_data(pci)` (see the *Non-Contiguous Buffers* section).

Once the buffer is pre-allocated, you can use the allocator in the *`hw_params`* callback:

```
snd_pcm_lib_malloc_pages(substream, size);
```

Note that you have to pre-allocate to use this function.

## External Hardware Buffers

Some chips have their own hardware buffers and the DMA transfer from the host memory is not available. In such a case, you need to either 1) copy/set the audio data directly to the external hardware buffer, or

2) make an intermediate buffer and copy/set the data from it to the external hardware buffer in interrupts (or in tasklets, preferably).

The first case works fine if the external hardware buffer is large enough. This method doesn't need any extra buffers and thus is more effective. You need to define the *copy* and *silence* callbacks for the data transfer. However, there is a drawback: it cannot be mmapped. The examples are GUS's GF1 PCM or emu8000's wavetable PCM.

The second case allows for mmap on the buffer, although you have to handle an interrupt or a tasklet to transfer the data from the intermediate buffer to the hardware buffer. You can find an example in the vxpocket driver.

Another case is when the chip uses a PCI memory-map region for the buffer instead of the host memory. In this case, mmap is available only on certain architectures like the Intel one. In non-mmap mode, the data cannot be transferred as in the normal way. Thus you need to define the *copy* and *silence* callbacks as well, as in the cases above. The examples are found in `rme32.c` and `rme96.c`.

The implementation of the *copy* and *silence* callbacks depends upon whether the hardware supports interleaved or non-interleaved samples. The *copy* callback is defined like below, a bit differently depending whether the direction is playback or capture:

```
static int playback_copy(struct snd_pcm_substream *substream, int channel,
             snd_pcm_uframes_t pos, void *src, snd_pcm_uframes_t count);
static int capture_copy(struct snd_pcm_substream *substream, int channel,
             snd_pcm_uframes_t pos, void *dst, snd_pcm_uframes_t count);
```

In the case of interleaved samples, the second argument (*channel*) is not used. The third argument (*pos*) points the current position offset in frames.

The meaning of the fourth argument is different between playback and capture. For playback, it holds the source data pointer, and for capture, it's the destination data pointer.

The last argument is the number of frames to be copied.

What you have to do in this callback is again different between playback and capture directions. In the playback case, you copy the given amount of data (*count*) at the specified pointer (*src*) to the specified offset (*pos*) on the hardware buffer. When coded like memcpy-like way, the copy would be like:

```
my_memcpy(my_buffer + frames_to_bytes(runtime, pos), src,
          frames_to_bytes(runtime, count));
```

For the capture direction, you copy the given amount of data (*count*) at the specified offset (*pos*) on the hardware buffer to the specified pointer (*dst*).

```
my_memcpy(dst, my_buffer + frames_to_bytes(runtime, pos),
          frames_to_bytes(runtime, count));
```

Note that both the position and the amount of data are given in frames.

In the case of non-interleaved samples, the implementation will be a bit more complicated.

You need to check the channel argument, and if it's -1, copy the whole channels. Otherwise, you have to copy only the specified channel. Please check `isa/gus/gus_pcm.c` as an example.

The *`silence`* callback is also implemented in a similar way.

```
static int silence(struct snd_pcm_substream *substream, int channel,
                   snd_pcm_uframes_t pos, snd_pcm_uframes_t count);
```

The meanings of arguments are the same as in the *`copy`* callback, although there is no *`src/dst`* argument. In the case of interleaved samples, the channel argument has no meaning, as well as on *`copy`* callback.

The role of *`silence`* callback is to set the given amount (*`count`*) of silence data at the specified offset (*`pos`*) on the hardware buffer. Suppose that the data format is signed (that is, the silent-data is 0), and the implementation using a memset-like function would be like:

```
my_memcpy(my_buffer + frames_to_bytes(runtime, pos), 0,
          frames_to_bytes(runtime, count));
```

In the case of non-interleaved samples, again, the implementation becomes a bit more complicated. See, for example, `isa/gus/gus_pcm.c`.

# Non-Contiguous Buffers

If your hardware supports the page table as in emu10k1 or the buffer descriptors as in via82xx, you can use the scatter-gather (SG) DMA. ALSA provides an interface for handling SG-buffers. The API is provided in `<sound/pcm.h>`.

For creating the SG-buffer handler, call `snd_pcm_lib_preallocate_pages()` or `snd_pcm_lib_preallocate_pages_for_all()` with `SNDRV_DMA_TYPE_DEV_SG` in the PCM constructor like other PCI pre-allocator. You need to pass `snd_dma_pci_data(pci)`, where pci is the struct pci_dev pointer of the chip as well. The struct snd_sg_buf instance is created as substream->dma_private. You can cast the pointer like:

```
struct snd_sg_buf *sgbuf = (struct snd_sg_buf *)substream->dma_private;
```

Then call `snd_pcm_lib_malloc_pages()` in the *`hw_params`* callback as well as in the case of normal PCI buffer. The SG-buffer handler will allocate the non-contiguous kernel pages of the given size and map them onto the virtually contiguous memory. The virtual pointer is addressed in runtime->dma_area. The physical address (runtime->dma_addr) is set to zero, because the buffer is physically non-contigous. The physical address table is set up in sgbuf->table. You can get the physical address at a certain offset via `snd_pcm_sgbuf_get_addr()`.

When a SG-handler is used, you need to set `snd_pcm_sgbuf_ops_page` as the *`page`* callback. (See *page callback section*.)

To release the data, call `snd_pcm_lib_free_pages()` in the *`hw_free`* callback as usual.

# Vmalloc'ed Buffers

It's possible to use a buffer allocated via `vmalloc`, for example, for an intermediate buffer. Since the allocated pages are not contiguous, you need to set the *`page`* callback to obtain the physical address at every offset.

The implementation of *`page`* callback would be like this:

```
#include <linux/vmalloc.h>

/* get the physical page pointer on the given offset */
static struct page *mychip_page(struct snd_pcm_substream *substream,
                                unsigned long offset)
{
        void *pageptr = substream->runtime->dma_area + offset;
        return vmalloc_to_page(pageptr);
}
```

# Chapter 12. Proc Interface

ALSA provides an easy interface for procfs. The proc files are very useful for debugging. I recommend you set up proc files if you write a driver and want to get a running status or register dumps. The API is found in `<sound/info.h>`.

To create a proc file, call `snd_card_proc_new()`.

```
struct snd_info_entry *entry;
int err = snd_card_proc_new(card, "my-file", &entry);
```

where the second argument specifies the name of the proc file to be created. The above example will create a file `my-file` under the card directory, e.g. `/proc/asound/card0/my-file`.

Like other components, the proc entry created via `snd_card_proc_new()` will be registered and released automatically in the card registration and release functions.

When the creation is successful, the function stores a new instance in the pointer given in the third argument. It is initialized as a text proc file for read only. To use this proc file as a read-only text file as it is, set the read callback with a private data via `snd_info_set_text_ops()`.

```
snd_info_set_text_ops(entry, chip, my_proc_read);
```

where the second argument (*`chip`*) is the private data to be used in the callbacks. The third parameter specifies the read buffer size and the fourth (*`my_proc_read`*) is the callback function, which is defined like

```
static void my_proc_read(struct snd_info_entry *entry,
                         struct snd_info_buffer *buffer);
```

In the read callback, use `snd_iprintf()` for output strings, which works just like normal `printf()`. For example,

```
static void my_proc_read(struct snd_info_entry *entry,
                         struct snd_info_buffer *buffer)
{
        struct my_chip *chip = entry->private_data;

        snd_iprintf(buffer, "This is my chip!\n");
        snd_iprintf(buffer, "Port = %ld\n", chip->port);
```

```
}
```

The file permissions can be changed afterwards. As default, it's set as read only for all users. If you want to add write permission for the user (root as default), do as follows:

```
entry->mode = S_IFREG | S_IRUGO | S_IWUSR;
```

and set the write buffer size and the callback

```
entry->c.text.write = my_proc_write;
```

For the write callback, you can use `snd_info_get_line()` to get a text line, and `snd_info_get_str()` to retrieve a string from the line. Some examples are found in `core/oss/mixer_oss.c`, core/oss/and `pcm_oss.c`.

For a raw-data proc-file, set the attributes as follows:

```
static struct snd_info_entry_ops my_file_io_ops = {
        .read = my_file_io_read,
};

entry->content = SNDRV_INFO_CONTENT_DATA;
entry->private_data = chip;
entry->c.ops = &my_file_io_ops;
entry->size = 4096;
entry->mode = S_IFREG | S_IRUGO;
```

The callback is much more complicated than the text-file version. You need to use a low-level I/O functions such as `copy_from/to_user()` to transfer the data.

```
static long my_file_io_read(struct snd_info_entry *entry,
                            void *file_private_data,
                            struct file *file,
                            char *buf,
                            unsigned long count,
                            unsigned long pos)
{
        long size = count;
        if (pos + size > local_max_size)
```

```
                size = local_max_size - pos;
        if (copy_to_user(buf, local_data + pos, size))
                return -EFAULT;
        return size;
}
```

# Chapter 13. Power Management

If the chip is supposed to work with suspend/resume functions, you need to add power-management code to the driver. The additional code for power-management should be `ifdef`'ed with `CONFIG_PM`.

If the driver *fully* supports suspend/resume that is, the device can be properly resumed to its state when suspend was called, you can set the `SNDRV_PCM_INFO_RESUME` flag in the pcm info field. Usually, this is possible when the registers of the chip can be safely saved and restored to RAM. If this is set, the trigger callback is called with `SNDRV_PCM_TRIGGER_RESUME` after the resume callback completes.

Even if the driver doesn't support PM fully but partial suspend/resume is still possible, it's still worthy to implement suspend/resume callbacks. In such a case, applications would reset the status by calling `snd_pcm_prepare()` and restart the stream appropriately. Hence, you can define suspend/resume callbacks below but don't set `SNDRV_PCM_INFO_RESUME` info flag to the PCM.

Note that the trigger with SUSPEND can always be called when `snd_pcm_suspend_all` is called, regardless of the `SNDRV_PCM_INFO_RESUME` flag. The `RESUME` flag affects only the behavior of `snd_pcm_resume()`. (Thus, in theory, `SNDRV_PCM_TRIGGER_RESUME` isn't needed to be handled in the trigger callback when no `SNDRV_PCM_INFO_RESUME` flag is set. But, it's better to keep it for compatibility reasons.)

In the earlier version of ALSA drivers, a common power-management layer was provided, but it has been removed. The driver needs to define the suspend/resume hooks according to the bus the device is connected to. In the case of PCI drivers, the callbacks look like below:

```
#ifdef CONFIG_PM
static int snd_my_suspend(struct pci_dev *pci, pm_message_t state)
{
        .... /* do things for suspend */
        return 0;
}
static int snd_my_resume(struct pci_dev *pci)
{
        .... /* do things for suspend */
        return 0;
}
#endif
```

The scheme of the real suspend job is as follows.

1. Retrieve the card and the chip data.

2. Call `snd_power_change_state()` with `SNDRV_CTL_POWER_D3hot` to change the power status.

3. Call `snd_pcm_suspend_all()` to suspend the running PCM streams.

4. If AC97 codecs are used, call `snd_ac97_suspend()` for each codec.

5. Save the register values if necessary.

6. Stop the hardware if necessary.
7. Disable the PCI device by calling `pci_disable_device()`. Then, call `pci_save_state()` at last.

A typical code would be like:

```
static int mychip_suspend(struct pci_dev *pci, pm_message_t state)
{
        /* (1) */
        struct snd_card *card = pci_get_drvdata(pci);
        struct mychip *chip = card->private_data;
        /* (2) */
        snd_power_change_state(card, SNDRV_CTL_POWER_D3hot);
        /* (3) */
        snd_pcm_suspend_all(chip->pcm);
        /* (4) */
        snd_ac97_suspend(chip->ac97);
        /* (5) */
        snd_mychip_save_registers(chip);
        /* (6) */
        snd_mychip_stop_hardware(chip);
        /* (7) */
        pci_disable_device(pci);
        pci_save_state(pci);
        return 0;
}
```

The scheme of the real resume job is as follows.

1. Retrieve the card and the chip data.
2. Set up PCI. First, call `pci_restore_state()`. Then enable the pci device again by calling `pci_enable_device()`. Call `pci_set_master()` if necessary, too.
3. Re-initialize the chip.
4. Restore the saved registers if necessary.
5. Resume the mixer, e.g. calling `snd_ac97_resume()`.
6. Restart the hardware (if any).
7. Call `snd_power_change_state()` with `SNDRV_CTL_POWER_D0` to notify the processes.

A typical code would be like:

```
static int mychip_resume(struct pci_dev *pci)
{
        /* (1) */
```

```
        struct snd_card *card = pci_get_drvdata(pci);
        struct mychip *chip = card->private_data;
        /* (2) */
        pci_restore_state(pci);
        pci_enable_device(pci);
        pci_set_master(pci);
        /* (3) */
        snd_mychip_reinit_chip(chip);
        /* (4) */
        snd_mychip_restore_registers(chip);
        /* (5) */
        snd_ac97_resume(chip->ac97);
        /* (6) */
        snd_mychip_restart_chip(chip);
        /* (7) */
        snd_power_change_state(card, SNDRV_CTL_POWER_D0);
        return 0;
}
```

As shown in the above, it's better to save registers after suspending the PCM operations via `snd_pcm_suspend_all()` or `snd_pcm_suspend()`. It means that the PCM streams are already stoppped when the register snapshot is taken. But, remember that you don't have to restart the PCM stream in the resume callback. It'll be restarted via trigger call with `SNDRV_PCM_TRIGGER_RESUME` when necessary.

OK, we have all callbacks now. Let's set them up. In the initialization of the card, make sure that you can get the chip data from the card instance, typically via *`private_data`* field, in case you created the chip data individually.

```
static int __devinit snd_mychip_probe(struct pci_dev *pci,
                             const struct pci_device_id *pci_id)
{
        ....
        struct snd_card *card;
        struct mychip *chip;
        int err;
        ....
        err = snd_card_create(index[dev], id[dev], THIS_MODULE, 0, &card);
        ....
        chip = kzalloc(sizeof(*chip), GFP_KERNEL);
        ....
        card->private_data = chip;
        ....
}
```

When you created the chip data with `snd_card_create()`, it's anyway accessible via *`private_data`* field.

```
static int __devinit snd_mychip_probe(struct pci_dev *pci,
                             const struct pci_device_id *pci_id)
{
        ....
        struct snd_card *card;
        struct mychip *chip;
        int err;
        ....
        err = snd_card_create(index[dev], id[dev], THIS_MODULE,
                              sizeof(struct mychip), &card);
        ....
        chip = card->private_data;
        ....
}
```

If you need a space to save the registers, allocate the buffer for it here, too, since it would be fatal if you cannot allocate a memory in the suspend phase. The allocated buffer should be released in the corresponding destructor.

And next, set suspend/resume callbacks to the pci_driver.

```
static struct pci_driver driver = {
        .name = "My Chip",
        .id_table = snd_my_ids,
        .probe = snd_my_probe,
        .remove = __devexit_p(snd_my_remove),
#ifdef CONFIG_PM
        .suspend = snd_my_suspend,
        .resume = snd_my_resume,
#endif
};
```

# Chapter 14. Module Parameters

There are standard module options for ALSA. At least, each module should have the *index*, *id* and *enable* options.

If the module supports multiple cards (usually up to 8 = SNDRV_CARDS cards), they should be arrays. The default initial values are defined already as constants for easier programming:

```
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;
```

If the module supports only a single card, they could be single variables, instead. *enable* option is not always necessary in this case, but it would be better to have a dummy option for compatibility.

The module parameters must be declared with the standard `module_param()()`, `module_param_array()()` and `MODULE_PARM_DESC()` macros.

The typical coding would be like below:

```
#define CARD_NAME "My Chip"

module_param_array(index, int, NULL, 0444);
MODULE_PARM_DESC(index, "Index value for " CARD_NAME " soundcard.");
module_param_array(id, charp, NULL, 0444);
MODULE_PARM_DESC(id, "ID string for " CARD_NAME " soundcard.");
module_param_array(enable, bool, NULL, 0444);
MODULE_PARM_DESC(enable, "Enable " CARD_NAME " soundcard.");
```

Also, don't forget to define the module description, classes, license and devices. Especially, the recent modprobe requires to define the module license as GPL, etc., otherwise the system is shown as “tainted”.

```
MODULE_DESCRIPTION("My Chip");
MODULE_LICENSE("GPL");
MODULE_SUPPORTED_DEVICE("{{Vendor,My Chip Name}}");
```

# Chapter 15. How To Put Your Driver Into ALSA Tree

## General

So far, you've learned how to write the driver codes. And you might have a question now: how to put my own driver into the ALSA driver tree? Here (finally :) the standard procedure is described briefly.

Suppose that you create a new PCI driver for the card “xyz”. The card module name would be snd-xyz. The new driver is usually put into the alsa-driver tree, `alsa-driver/pci` directory in the case of PCI cards. Then the driver is evaluated, audited and tested by developers and users. After a certain time, the driver will go to the alsa-kernel tree (to the corresponding directory, such as `alsa-kernel/pci`) and eventually will be integrated into the Linux 2.6 tree (the directory would be `linux/sound/pci`).

In the following sections, the driver code is supposed to be put into alsa-driver tree. The two cases are covered: a driver consisting of a single source file and one consisting of several source files.

## Driver with A Single Source File

1. Modify alsa-driver/pci/Makefile

   Suppose you have a file xyz.c. Add the following two lines

   ```
   snd-xyz-objs := xyz.o
   obj-$(CONFIG_SND_XYZ) += snd-xyz.o
   ```

2. Create the Kconfig entry

   Add the new entry of Kconfig for your xyz driver.

   ```
   config SND_XYZ
           tristate "Foobar XYZ"
           depends on SND
           select SND_PCM
           help
             Say Y here to include support for Foobar XYZ soundcard.

             To compile this driver as a module, choose M here: the module
             will be called snd-xyz.
   ```

   the line, select SND_PCM, specifies that the driver xyz supports PCM. In addition to SND_PCM, the following components are supported for select command: SND_RAWMIDI, SND_TIMER,

SND_HWDEP, SND_MPU401_UART, SND_OPL3_LIB, SND_OPL4_LIB, SND_VX_LIB, SND_AC97_CODEC. Add the select command for each supported component.

Note that some selections imply the lowlevel selections. For example, PCM includes TIMER, MPU401_UART includes RAWMIDI, AC97_CODEC includes PCM, and OPL3_LIB includes HWDEP. You don't need to give the lowlevel selections again.

For the details of Kconfig script, refer to the kbuild documentation.

3. Run cvscompile script to re-generate the configure script and build the whole stuff again.

# Drivers with Several Source Files

Suppose that the driver snd-xyz have several source files. They are located in the new subdirectory, pci/xyz.

1. Add a new directory (`xyz`) in `alsa-driver/pci/Makefile` as below

```
obj-$(CONFIG_SND) += xyz/
```

2. Under the directory `xyz`, create a Makefile

### Example 15.1. Sample Makefile for a driver xyz

```
ifndef SND_TOPDIR
SND_TOPDIR=../..
endif

include $(SND_TOPDIR)/toplevel.config
include $(SND_TOPDIR)/Makefile.conf

snd-xyz-objs := xyz.o abc.o def.o

obj-$(CONFIG_SND_XYZ) += snd-xyz.o

include $(SND_TOPDIR)/Rules.make
```

3. Create the Kconfig entry

   This procedure is as same as in the last section.

4. Run cvscompile script to re-generate the configure script and build the whole stuff again.

# Chapter 16. Useful Functions

## snd_printk() and friends

ALSA provides a verbose version of the printk() function. If a kernel config CONFIG_SND_VERBOSE_PRINTK is set, this function prints the given message together with the file name and the line of the caller. The KERN_XXX prefix is processed as well as the original printk() does, so it's recommended to add this prefix, e.g.

```
snd_printk(KERN_ERR "Oh my, sorry, it's extremely bad!\n");
```

There are also printk()'s for debugging. snd_printd() can be used for general debugging purposes. If CONFIG_SND_DEBUG is set, this function is compiled, and works just like snd_printk(). If the ALSA is compiled without the debugging flag, it's ignored.

snd_printdd() is compiled in only when CONFIG_SND_DEBUG_VERBOSE is set. Please note that CONFIG_SND_DEBUG_VERBOSE is not set as default even if you configure the alsa-driver with --with-debug=full option. You need to give explicitly --with-debug=detect option instead.

## snd_BUG()

It shows the BUG? message and stack trace as well as snd_BUG_ON at the point. It's useful to show that a fatal error happens there.

When no debug flag is set, this macro is ignored.

## snd_BUG_ON()

snd_BUG_ON() macro is similar with WARN_ON() macro. For example,

```
snd_BUG_ON(!pointer);
```

or it can be used as the condition,

```
if (snd_BUG_ON(non_zero_is_bug))
        return -EINVAL;
```

The macro takes an conditional expression to evaluate. When CONFIG_SND_DEBUG, is set, the expression is actually evaluated. If it's non-zero, it shows the warning message such as BUG? (xxx) normally followed by stack trace. It returns the evaluated value. When no `CONFIG_SND_DEBUG` is set, this macro always returns zero.

# Chapter 17. Acknowledgments

I would like to thank Phil Kerr for his help for improvement and corrections of this document.

Kevin Conder reformatted the original plain-text to the DocBook format.

Giuliano Pochini corrected typos and contributed the example codes in the hardware constraints section.