Как написать драйвер ALSA

Takashi Iwai

Как написать драйвер ALSA

Takashi Iwai

Краткое описание

Этот документ описывает, как написать драйвер ALSA (Advanced Linux Sound Architecture).

Оглавление

Как написать драйвер ALSA

http://www.alsa-project.org/main/index.php/ALSA_Driver_Documentation

Writing an ALSA Driver

Takashi Iwai

Краткое описание

Этот документ описывает, как написать драйвер ALSA (Advanced Linux Sound Architecture).

Предисловие

Этот документ описывает, как написать драйвер ***ALSA (Advanced Linux Sound Architecture, Улучшенная Архитектура Звука Linux)*** [http://www.alsaproject.org/]. Документ ориентирован в основном на звуковые карты PCI. В случае других типов устройств API также может быть другим. Однако, по крайней мере API ядра ALSA является неизменным и поэтому всё ещё немного поможет при их написании.

Этот документ ориентирован на людей, которые уже имеют достаточный уровень знания языка Си и основные знания в области программирования ядра Linux. Этот документ не объясняет общие подходы к программированию ядра Linux и не описывает низкоуровневые детали реализации драйвера. Он описывает только стандартный способ написания звукового драйвера с использованием ALSA.

Если вы уже знакомы со старым API ALSA версии 0.5.x, вы можете посмотреть такие драйверы, как **sound/pci/es1938.c** или **sound/pci/maestro3.c**, которые имеют почти такой же базовый код в дереве ALSA 0.5.x, так что можно посмотреть отличия.

Этот документ всё ещё является черновой версией. Любые отзывы и корректировки, пожалуйста!!

Глава 1. Структура файлового дерева

Общие сведения

Драйверы ALSA представлена двумя способами.

Одним из них является дерево, предоставляемое, как tar-архив или через CVS с FTP сайта ALSA, а другим - дерево ядра Linux версии 2.6 (или более поздней). Для синхронизации обоих, дерево драйверов ALSA состоит из двух различных деревьев: ядро ALSA и драйверы ALSA. Первое содержит только исходный код для дерева Linux 2.6 (или более поздней версии). Это дерево предназначено исключительно для компиляции в среде версии 2.6 или более поздней. Второе, драйверы ALSA, содержит множество небольших файлов для компиляции драйверов ALSA вне дерева ядра Linux, функции-обёртки для старых версий ядра 2.2 и 2.4, чтобы адаптировать самое последнее API ядра, и дополнительные драйверы, которые всё ещё находятся в разработке или тестируются. Драйверы в дереве драйверов ALSA будут перемещены в ядро ALSA (и в конечном итоге в дерево ядра 2.6), когда они будут завершены и подтверждено, что они работают хорошо.

Файловая структура дерева драйверов ALSA показана ниже. И ядро ALSA, и драйверы ALSA имеют почти одинаковую файловую структуру, за исключением каталога "core". В дереве драйверов ALSA он называется "acore".

Пример 1.1. Структура файлового дерева ALSA

```
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
```

Каталог core

Этот каталог содержит центральный уровень, который является сердцем драйверов ALSA. В этом каталоге хранятся родные модули ALSA. Подкаталоги содержат различные модули и зависят от конфигурации ядра.

core/oss

В этом каталоге хранятся коды для модулей эмуляции PCM и микшера OSS. Эмуляция rawmidi OSS включена в код rawmidi ALSA, так как он очень небольшой. Код секвенсора хранится в каталоге ***core/seq/oss*** (смотрите ниже 3).

core/ioctl32

Этот каталог содержит обёртки 32-х разрядных ioctl для 64-х разрядных архитектур, таких как x86-64, ppc64 и sparc64. Для 32-х разрядных архитектур и alpha они не собираются.

core/seq

Этот каталог и его подкаталоги предназначены для секвенсора ALSA. Этот каталог содержит ядро секвенсора и основные модули секвенсора, такие как snd-seq-midi, snd-seq-virmidi и так далее. Они компилируются только тогда, когда в конфигурации ядра устанавливается **CONFIG_SND_SEQUENCER**.

core/seq/oss

Этот каталог содержит коды эмуляции секвенсора OSS.

core/seq/instr

Этот каталог содержит модули для уровня инструментов секвенсора.

Каталог include

Это место для общедоступных файлов заголовков драйверов ALSA, которые должны быть экспортированы в пространство пользователя, или подключаются некоторыми файлами в других каталогах. В основном, не предназначенные для общего использования файлы заголовков не должны быть помещены в этот каталог, но такие файлы всё равно можно найти здесь в силу исторических причин :)

Каталог drivers

Этот каталог содержит код, используемый совместно различными драйверами на разных архитектурах. Следовательно, он не должен быть архитектурно-зависимым. Например, в этом каталоге можно найти драйвер-пустышку PCM и последовательный драйвер MIDI. В подкаталогах есть код для компонентов, которые не зависят от архитектуры шины и процессора.

drivers/mpu401

Здесь хранятся модули MPU401 и MPU401-UART.

drivers/opl3 и opl4

Здесь находится материал для FM-синтезатора OPL3 и OPL4.

Каталог i2c

Этот каталог содержит компоненты ALSA i2c.

Хотя на Linux есть стандартный уровень i2c, ALSA имеет свой собственный код i2c для некоторых карт, потому что звуковой карте необходимы только простые операции и стандартное API i2c является слишком сложным для такой цели.

i2c/l3

Это подкаталог для ARM L3 i2c.

Каталог synth

Этот каталог содержит модули синтезатора центрального уровня.

Пока в подкаталоге ***synth/emux*** есть только драйвер синтезатора Emu8000/Emu10k1.

Каталог pci

В этом каталоге и его подкаталогах содержатся высокоуровневые модули карт для звуковых карт PCI и код, относящийся к шине PCI.

Драйверы, скомпилированные из одного файла, хранятся прямо в каталоге ***pci***, а драйверы из нескольких исходных файлов хранятся в своих собственных подкаталогах (например, emu10k1, ice1712).

Каталог isa

В этом каталоге и его подкаталогах содержатся высокоуровневые модули карты для звуковых карт ISA.

Каталоги arm, ppc и sparc

Эти каталоги используются для высокоуровневых модулей карты, которые являются специфическими для одной из этих архитектур.

Каталог usb

Этот каталог содержит аудио драйвер USB. В последней версии драйвер USB MIDI интегрирован в драйвер звука USB.

Каталог pcmcia

PCMCIA, особенно драйверы PCCard, будут находиться здесь. Драйверы CardBus будут в каталоге PCI, потому что их API такой же, как для стандартных карт PCI.

Каталог oss

Здесь в Linux 2.6 (или более поздней версии) дерева хранятся исходные файлы OSS/Lite. В tar-архиве драйверов ALSA этот каталог, конечно, пуст :)

Глава 2. Основная технология для PCI драйверов

Краткое описание

Минимальная структура для звуковых карт PCI выглядит следующим образом:

- определение таблицы идентификаторов PCI (см. раздел Регистрация PCI 21).
- создание обратного вызова ***probe()***.
- создание обратного вызова ***remove()***.
- создание структуры **pci_driver**, содержащую три вышеописанные указателя.
- создание функции ***init()***, просто вызывающей ***pci_register_driver()*** для регистрации таблицы **pci_driver**, определённой до этого.
- создание функции ***exit()*** для вызова функции ***pci_unregister_driver()***.

Полный пример кода

Ниже приведён пример кода. Некоторые части на данный момент оставлены нереализованными, но пробелы будут заполнены в следующих разделах. Номера в строках комментариев функции ***snd_mychip_probe()*** относятся к деталям, объясняемым в следующем разделе.

Пример 2.1. Основная технология для PCI драйверов - пример

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

/* параметры модуля (смотрите "Параметры модуля") */
/* SNDRV_CARDS: максимальное число карт, поддерживаемых этим модулем */
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;

/* определение объекта, зависящего от используемой микросхемы */
struct mychip {
    struct snd_card *card;
    /* остальная часть реализации будет в разделе
     * "Управление ресурсами PCI"
     */
};

/* деструктор, зависящий от используемой микросхемы
 * (смотрите "Управление ресурсами PCI")
 */
static int snd_mychip_free(struct mychip *chip)
{
    .... /* будет реализовано позже... */
}
```

```
/* деструктор компонента
 * (смотрите "Управление картами и компонентами")
 */
static int snd_mychip_dev_free(struct snd_device *device)
{
    return snd_mychip_free(device->device_data);
}

/* конструктор, зависящий от используемой микросхемы
 * (смотрите "Управление картами и компонентами")
 */
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
    struct mychip *chip;
    int err;
    static struct snd_device_ops ops = {
       .dev_free = snd_mychip_dev_free,
    };

    *rchip = NULL;

    /* здесь проверяем доступность PCI
     * (смотрите "Управление ресурсами PCI")
     */
    ....

    /* выделяем обнулённую память для зависимых от микросхемы данных */
    chip = kzalloc(sizeof(*chip), GFP_KERNEL);
    if (chip == NULL)
        return -ENOMEM;

    chip->card = card;

    /* здесь остальная инициализация; будет реализована
     * позже, смотрите "Управление ресурсами PCI"
     */
    ....

    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;
}

/* конструктор -- смотрите подраздел "Конструктор" */
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) */
    .... /* реализовано позже */
    /* (6) */
    err = snd_card_register(card);
    if (err < 0) {
        snd_card_free(card);
        return err;
    }
    /* (7) */
    pci_set_drvdata(pci, card);
    dev++;
    return 0;
}

/* деструктор -- смотрите подраздел "Деструктор" */
static void __devexit snd_mychip_remove(struct pci_dev *pci)
{
    snd_card_free(pci_get_drvdata(pci));
    pci_set_drvdata(pci, NULL);
}
```

Конструктор

Настоящим конструктором драйверов PCI является обратный вызов ***probe***. Обратный вызов ***probe*** и другие конструкторы компонентов, вызывающиеся из обратного вызова ***probe***, должны быть определены с префиксом **__devinit**. Для них нельзя использовать префикс

__init, потому что любое устройство PCI может быть автоопределяемым устройством.

В обратном вызове ***probe*** часто используется следующая схема.

1) Проверяем и увеличиваем индекс устройства.

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

где **enable[dev]** является параметром модуля.

Каждый раз, когда выполняется обратный вызов ***probe***, проверяется наличие устройства. Если оно не доступно, просто увеличиваем индекс устройства и возвращаемся. **dev** будет увеличиваться и позднее (на шаге 7 10).

2) Создаём экземпляр карты.

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

Подробности будут объяснены в разделе Управление картами и компонентами 12.

3) Создаём основной компонент.

В этой части выделяется память для ресурсов PCI.

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

Подробности будут объяснены в разделе Управления ресурсами PCI 15.

4) Устанавливаем идентификатор драйвера и строки названий.

```
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);
```

Поле **driver** содержит минимальную строку идентификатора чипа. Это используется

конфигуратором ALSA-lib, поэтому делайте её простой, но уникальной. Даже один и тот же драйвер может иметь разные идентификаторы драйвера, чтобы различать функциональность чипов разных типов.

Поле **shortname** содержит строку, показывающуюся, как более подробное название. Поле **longname** содержит информацию, показываемую в */proc/asound/cards*.

5) Создаём другие компоненты, такие как микшер, MIDI, и так далее.

Здесь определяются основные компоненты, такие как PCM 23, микшер (например, AC97 51), MIDI (например, MPU-401 55), и другие интерфейсы. Кроме того, если необходим файл proc 68, определите здесь и его.

6) Регистрируем экземпляр карты.

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

Будет тоже объяснено в разделе Управление картами и компонентами 12.

7) Устанавливаем данные PCI драйвера и возвращаем ноль.

```
pci_set_drvdata(pci, card);
dev++;
return 0;
```

Как говорилось выше, это сохранение объекта, описывающего карту. Этот указатель также используется в обратных вызовах удаления и управления питанием.

Деструктор

Деструктор, обратный вызов ***remove***, просто освобождает экземпляр карты. Затем центральный уровень ALSA автоматически освободит все подключенные компоненты.

Это будет выглядеть, как правило, следующим образом:

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

Вышеприведённый код предполагает, что указатель карты указывает на закрытые данные PCI драйвера.

Заголовочные файлы

Для приведенного выше примера необходимы по крайней мере следующие файлы заголовков.

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

где последний необходим только тогда, когда параметры модуля определены в исходном файле. Если код разделён на несколько файлов, файлам без параметров модуля он не нужен.

В дополнение к этим заголовкам необходим ***<linux/interrupt.h>*** для обработки прерываний и ***<asm/io.h>*** для доступа к вводу/выводу. Если используются функции ***mdelay()*** или ***udelay()***, необходимо также подключить ***<linux/delay.h>***.

Интерфейсы ALSA, такие как API PCM и управления, определены в других файлах заголовков ***<sound/xxx.h>***. Они должны быть подключены после ***<sound/core.h>***.

Глава 3. Управление картами и компонентами

Экземпляр карты

Для каждой звуковой карты должна быть выделена память для объекта "card".

Объект **card** является основой звуковой карты. Он управляет целым списком устройств (компонентов) звуковой карты, такими как PCM, микшеры, MIDI, синтезатор и так далее. Кроме того, объект **card** хранит ID и строку названия карты, управляет корнем файлов proc и контролирует состояния управлением питания и отключения устройств с автоопределением. Список компонентов в объекте card используется для управления правильным освобождением ресурсов при уничтожении.

Как упоминалось выше, для создания экземпляра карты вызывается ***snd_card_create()***.

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

Функция принимает пять аргументов, число индекса карты, строку идентификации, указатель модуля (обычно **THIS_MODULE**), размер пространства для дополнительных данных и указатель для получения экземпляра карты. Аргумент **extra_size** используется для выделения памяти **card->private_data** для данных, зависящих от используемого чипа. Обратите внимание, что память для этих данных выделяется с помощью ***snd_card_create()***.

Компоненты

После того, как карта создана, можно подключить компоненты (устройства) к данному экземпляру карты. В драйвере ALSA компонент представлен в виде объекта структуры **snd_device**. Компонент может быть экземпляром PCM, интерфейсом управления, интерфейсом raw MIDI и так далее. Каждый такой экземпляр имеет одну запись компонента.

Компонент может быть создан с помощью функции ***snd_device_new()***.

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

Она получает указатель на карту, уровень устройства (**SNDRV_DEV_XXX**), указатель на данные и указатели обратных вызовов (**&ops**). Уровень устройства определяет тип компонентов и порядок регистрации и разрегистрации. Для большинства компонентов уровень устройства уже определён. Для определяемых пользователем компонентов можно использовать **SNDRV_DEV_LOWLEVEL**.

Эта функция сама по себе не выделяет память для данных. Память для данных должна быть выделена вручную заранее и указатель на неё передаётся в качестве аргумента. Этот указатель используется в качестве (идентификатор **chip** в вышеприведённом примере) данного экземпляра.

Каждый предопределённый компонент ALSA, такой как ac97 и PCM, вызывает внутри своего конструктора ***snd_device_new()***. Деструктор для каждого компонента определяется в указателях обратного вызова. Таким образом нет необходимости заботиться о вызове

деструктора для таких компонентов.

Если вы хотите создать свой собственный компонент, необходимо указать функцию деструктора в обратном вызове ***dev_free*** в **ops**, так что он может быть освобожден автоматически через ***snd_card_free()***. Следующий пример покажет реализацию работы с зависимыми от чипа данными.

Данные, зависимые от используемого чипа

Зависимая от чипа информация, например, адрес порта ввода/вывода, указатель на его ресурсы, или номер прерывания, хранится в объекте, зависящем от чипа.

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

В общем, есть два способа выделения памяти для объекта чипа.

1. Создание с помощью snd_card_create().

Как уже упоминалось выше, можно передать размер дополнительных данных в 4-м аргументе ***snd_card_create()***, то есть

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

Структура **mychip** представляет собой тип объекта чипа.

По возвращении, выделенная память для объекта может быть доступна как

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

С помощью этого метода, не требуется выделять память два раза. Объект освобождается вместе с экземпляром карты.

2. Создание дополнительного устройства.

После создания экземпляра карты через ***snd_card_create()*** (с 0 в 4-м аргументе) вызываем ***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);
```

Объект чипа должна иметь по крайней мере поле для хранения указателя на карту,

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

Затем устанавливаем указатель на карту, возвращаемый экземпляром чипа.

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

Далее, инициализируем поля и регистрируем этот объект чипа как низкоуровневое устройство с указанными **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() является функцией деструктора устройства, которая будет вызывать настоящий деструктор.

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

где ***snd_mychip_free()*** является настоящим деструктором.

Регистрация и освобождение памяти

После определения всех компонентов зарегистрируем экземпляр карты, вызывая ***snd_card_register()***. В этой точке разрешается доступ к файлам устройства. То есть до вызова ***snd_card_register()*** компоненты гарантировано недоступны снаружи. Если этот вызов не удался, выходим из функции ***probe*** после освобождения карты через ***snd_card_free()***.

Для освобождения экземпляра карты можно просто вызвать ***snd_card_free()***. Как упоминалось ранее, этим вызовом автоматически освобождаются все компоненты.

Обратите внимание, что деструкторы (как ***snd_mychip_dev_free***, так и ***snd_mychip_free***) не могут быть определены с префиксом **__devexit**, потому что они могут быть вызваны также из конструктора при обработке ошибок.

Для устройства, которое позволяет горячее подключение, можно использовать ***snd_card_free_when_closed***. Она отложит уничтожение до момента, пока не будут закрыты все устройств.

Глава 4. Управление ресурсами PCI

Полный пример кода

В этом разделе мы завершим зависящий от используемого чипа конструктор, деструктор и регистрацию PCI. Сначала покажем пример кода.

Пример 4.1. Пример управления ресурсами PCI

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

    unsigned long port;
    int irq;
};

static int snd_mychip_free(struct mychip *chip)
{
    /* отключаем оборудование, если необходимо */
    .... /* (в этом документе не реализовано) */

    /* освобождаем прерывание */
    if (chip->irq >= 0)
        free_irq(chip->irq, chip);
    /* освобождаем порты ввода/вывода и память */
    pci_release_regions(chip->pci);
    /* отключаем регистрацию PCI */
    pci_disable_device(chip->pci);
    /* освобождаем данные */
    kfree(chip);
    return 0;
}

/* конструктор, зависящий от используемого чипа */
static int __devinit snd_mychip_create(struct snd_card *card,
                                       struct pci_dev *pci,
                                       struct mychip **rchip)
{
    struct mychip *chip;
    int err;
    static struct snd_device_ops ops = {
        .dev_free = snd_mychip_dev_free,
    };

    *rchip = NULL;

    /* инициализируем регистрацию PCI */
    err = pci_enable_device(pci);
    if (err < 0)
        return err;
    /* проверяем доступность PCI (28-х разрядный DMA) */
    if (pci_set_dma_mask(pci, DMA_BIT_MASK(28)) < 0 ||
```

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

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

    /* инициализируем параметры */
    chip->card = card;
    chip->pci = pci;
    chip->irq = -1;

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

    /* (2) инициализируем оборудование чипа */
    .... /* (в этом документе не реализовано) */

    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;
}

/* идентификаторы PCI */
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);

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

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

/* удаление модуля */
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; /* только для старых версий ядра */
```

Кое-что из того, что требуется сделать

Выделение ресурсов PCI выполняется в функции ***probe()***, и, как правило, в дополнительной функции ***xxx_create()***, написанной для этой цели.

В случае устройств PCI, необходимо перед выделением ресурсов сначала вызвать функцию ***pci_enable_device()***. Кроме того, необходимо установить корректную маску PCI DMA для ограничения доступного диапазона ввода-вывода. В некоторых случаях также будет необходимо вызвать функцию ***pci_set_master()***.

Пусть маска будет 28-ми разрядной, тогда код для добавления был бы таким:

```
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;
}
```

Выделение ресурсов

Выделение портов ввода/вывода и прерываний осуществляется через стандартные функции ядра. В отличие от ALSA версии 0.5.x. помощников для этого нет. И эти ресурсы

должны быть освобождены в функции деструктора (смотрите ниже). Кроме того, в ALSA 0.9.x не требуется выделять (псевдо-) DMA для PCI, как в ALSA 0.5.x.

Теперь предположим, что устройство PCI имеет 8-ми байтовый порт ввода/вывода и прерывание. Тогда структура **mychip** будет иметь следующие поля:

```
struct mychip {
    struct snd_card *card;

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

Для порта ввода/вывода (а также области памяти) необходимо иметь указатель ресурсов для стандартного управления ресурсами. Для прерывания необходимо хранить только номер прерывания (целое число). Но необходимо проинициализировать этот номер как -1 до фактического выделения, так как прерывание 0 правомерно. Адрес порта и его указатель ресурсов можно проинициализировать нулями автоматически с помощью ***kzalloc()***, поэтому не придётся заботиться об их обнулении.

Выделение порта ввода/вывода осуществляется следующим образом:

```
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);
```

Это будет резервировать 8-ми байтовый порт ввода/вывода данного устройства PCI. Возвращаемое значение, **chip->res_port**, выделяется через ***kmalloc()*** с помощью ***request_region()***. Указатель должен быть освобождён через ***kfree()***, но с этим есть проблема. Эта проблема будет объяснятся позже.

Выделение источника прерывания выполняется примерно так:

```
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;
```

где ***snd_mychip_interrupt()*** является обработчиком прерывания, определяемым позже. Заметим, что **chip->irq** должен быть определён только если вызов ***request_irq()*** был успешным.

На шине PCI прерывания могут быть общими. Таким образом, в качестве флага прерывания в ***request_irq()*** используется **IRQF_SHARED**.

Последним аргументом ***request_irq()*** является указатель на данные, передаваемые в обработчик прерывания. Обычно для этого используется объект чипа, но также можно использовать то, что нравится.

На данный момент я не буду давать подробную информацию об обработчике прерывания, но сейчас по крайней мере можно объяснить, как он выглядит. Обработчик прерывания обычно выглядит так:

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

Теперь давайте напишем соответствующий деструктор для вышеописанных ресурсов. Роль деструктора проста: отключить оборудование (если оно уже активировано) и освободить ресурсы. Пока у нас нет аппаратной части, так что код отключения здесь не написан.

Для освобождения ресурсов метод "проверить и освободить" является безопасным способом. Для прерывания сделаем так:

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

Так как номер прерывания может начинаться с 0, необходимо инициализировать **chip->irq** отрицательным значением (например, -1), так что можно проверять достоверность номер прерывания, как показано выше.

Когда порты ввода/вывода или области памяти запрашиваются с помощью ***pci_request_region()*** или ***pci_request_regions()***, как в этом примере, освобождайте ресурс(ы) с помощью соответствующей функции, ***pci_release_region()*** или ***pci_release_regions()***.

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

Если запрос выполняется вручную через ***request_region()*** или ***request_mem_region()***, можно освободить его через ***release_resource()***. Предположим, что указатель, возвращённый из ***request_region()***, хранится в **chip->res_port**, тогда процедура освобождение выглядит следующим образом:

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

Не забудьте перед окончанием вызвать ***pci_disable_device()***.

И, наконец, освобождаем объект, описывающий чип.

```
kfree(chip);
```

Опять же, помните, что для этого деструктора нельзя использовать префикс **__devexit**.

Выше мы не реализовали часть отключения оборудования. Если необходимо сделать это, обратите внимание, что деструктор может быть вызван ещё до завершения инициализации

чипа. Было бы лучше иметь флаг, чтобы пропускать отключение оборудования, если оборудование ещё не было проинициализировано.

Если данные для чипа выделяются для карты используя ***snd_device_new()*** с **SNDRV_DEV_LOWLELVEL**, их деструктор вызывается последним. То есть гарантируется, что все другие компоненты, такие как PCM-ы и элементы управления, уже были освобождены. Вы не должны останавливать PCM-ы и другие явно, необходимо просто вызвать остановку низкоуровневого оборудования.

Управление областью отображаемой памяти почти такое же, как управление портом ввода/вывода. Вы будете нуждаться в трёх полях, как показано ниже:

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

и выделение было бы таким, как показано ниже:

```
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));
```

и соответствующий деструктор будет:

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

Регистрация структуры устройства

В какой-то момент, как правило, после вызова ***snd_device_new()***, необходимо зарегистрировать структуру устройства чипа, где выполняется обработка **udev** и других. ALSA предоставляет макрос для совместимости со старыми ядрами.

Просто сделайте следующий вызов:

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

так что он сохраняет указатель устройства PCI в объекте карты. Он будет передан функциям ядра ALSA позже, когда устройства зарегистрированы.

В случае не PCI вместо шины передаётся надлежащий указатель на структуру устройства. (В случае устаревшей ISA без PnP, вы не должны ничего делать.)

Регистрация PCI

Пока всё идет хорошо. Давайте завершим с отсутствующим материалом PCI. Сначала для этого чипсета необходима таблица **pci_device_id**. Это таблица с идентификационными номерами поставщика/устройства PCI и некоторые маски.

Например,

```
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);
```

Первое и второе поле структуры **pci_device_id** являются идентификаторами поставщика и устройства. Если нет причин для фильтрации совпадающих устройств, можно оставить остальные поля, как показано выше. Последнее поле структуры **pci_device_id** содержит закрытые данные для этой записи. Здесь можно задать любое значение, например, для определения специфических операций для идентификаторов поддерживаемых устройств. Такой пример можно найти в драйвере intel8x0.

Последняя запись в этом списке является признаком завершения. Вы должны указать этот нулевой элемент.

Затем подготовим объект **pci_driver**:

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

Функции ***probe*** и ***remove*** уже определены в предыдущих разделах. Функция ***remove*** должна быть определена с помощью макроса **__devexit_p()**, так что она не определена для встроенного (и без горячей замены) случая. Поля **name** является строкой названия этого устройства. Отметим, что в этой строке вы не должны использовать символ "/".

И, наконец, регистрация модуля:

```
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)
```

Заметим, что эти функции модуля помечены префиксами **__init** и **__exit**, а не **__devinit** или **__devexit**.

Ах да, была забыта ещё одна вещь. Если у вас нет экспортируемых символов, необходимо задекларировать это в ядрах версии 2.2 или 2.4 (это не обязательно в ядрах версии 2.6).

```
EXPORT_NO_SYMBOLS;
```

Это всё!

Глава 5. Интерфейс PCM

Общие сведения

Центральный уровень PCM ALSA является достаточно мощным и единственно необходимым для каждого драйвера для реализации низкоуровневых функций доступа к оборудованию.

Для доступа к уровню PCM необходимо в первую очередь подключить ***<sound/pcm.h>***. Кроме того, может потребоваться ***<sound/pcm_params.h>***, если используется доступ к каким-либо функциям, связанным с **hw_param**.

Каждая карта устройства может иметь до четырёх экземпляров PCM. Экземпляр PCM соответствует PCM файлу устройства. Ограничение числа экземпляров является лишь следствием имеющейся разрядности номеров устройств Linux. Когда станут использоваться 64-х разрядные номера устройств, станет возможным иметь больше экземпляров PCM.

Экземпляр PCM содержит PCM потоки воспроизведения и захвата, а каждый PCM поток состоит из одного или более субпотоков PCM. Некоторые звуковые карты поддерживают сложные функции воспроизведения. Например, emu10k1 имеет возможность воспроизведение PCM потока из 32-х стерео субпотоков. В этом случае во время каждого открытия (как правило) автоматически выбирается и открывается свободный субпоток. Между тем, если существует только один субпоток и он был уже открыт, успешное открытое будет либо блокироваться, либо выдаваться ошибка **EAGAIN**, в зависимости от режима открытия файла. Но вы не должны заботиться о таких деталях в своём драйвере. О таком поведении будет заботиться центральный уровень PCM.

Полный пример кода

Пример, приведённый ниже, не включает процедуры доступа к оборудованию, а показывает только скелет построения интерфейсов PCM.

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

/* параметры оборудования */
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,
};

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

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

    runtime->hw = snd_mychip_playback_hw;
    /* здесь будет дополнительный код инициализации оборудования */
    ....
    return 0;
}

/* обратный вызов close */
static int snd_mychip_playback_close(struct snd_pcm_substream *substream)
{
    struct mychip *chip = snd_pcm_substream_chip(substream);
    /* здесь будет дополнительный код инициализации оборудования */
    ....
    return 0;
}

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

    runtime->hw = snd_mychip_capture_hw;
    /* здесь будет дополнительный код инициализации оборудования */
    ....
    return 0;
}
```

```
/* обратный вызов close */
static int snd_mychip_capture_close(struct snd_pcm_substream *substream)
{
    struct mychip *chip = snd_pcm_substream_chip(substream);
    /* здесь будет код, зависящий от оборудования */
    ....
    return 0;
}

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

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

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

    /* настраиваем оборудование, используя текущую конфигурацию,
     * например...
     */
    mychip_set_sample_format(chip, runtime->format);
    mychip_set_sample_rate(chip, runtime->rate);
    mychip_set_channels(chip, runtime->channels);
    mychip_set_dma_setup(chip, runtime->dma_addr,
                        chip->buffer_size,
                        chip->period_size);
    return 0;
}

/* обратный вызов trigger */
static int snd_mychip_pcm_trigger(struct snd_pcm_substream *substream,
                                  int cmd)
{
    switch (cmd) {
    case SNDRV_PCM_TRIGGER_START:
        /* делаем что-нибудь для запуска работы с PCM */
        ....
        break;
    case SNDRV_PCM_TRIGGER_STOP:
        /* делаем что-нибудь для остановки работы с PCM */
        ....
        break;
    default:
```

```
        return -EINVAL;
    }
}

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

    /* получаем текущий указатель на оборудование */
    current_ptr = mychip_get_hw_pointer(chip);
    return current_ptr;
}

/* операции */
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,
};

/* операции */
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,
};

/*
 *  определения захвата здесь опущены...
 */

/* создание pcm устройства */
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;
    /* устанавливаем операции */
    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);
    /* предварительное создание буферов */
    /* ЗАМЕЧАНИЕ: это может окончиться неудачей */
    snd_pcm_lib_preallocate_pages_for_all(pcm, SNDRV_DMA_TYPE_DEV,
                                          snd_dma_pci_data(chip->pci),
                                          64*1024, 64*1024);
    return 0;
}
```

Конструктор

Экземпляр PCM создаётся функцией ***snd_pcm_new()***. Но лучше создать конструктор для PCM, а именно:

```
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;
}
```

Функция ***snd_pcm_new()*** принимает четыре аргумента. Первым аргументом является указатель на карту, для которой предназначен этот PCM, а вторым - строка идентификатора.

Третий аргумент (индекс, 0 в приведённом выше примере) - индекс этого нового PCM. Он начинается с нуля. Если вы создаёте более одного экземпляра PCM, указывайте другие числа в этом аргументе. Например, для второго устройства PCM индекс = 1.

Четвертый и пятый аргументы представляют собой количество субпотоков для воспроизведения и захвата, соответственно. Здесь для обоих аргументов использовано 1. Если нет субпотоков воспроизведения или захвата, в соответствующем аргументе передаётся 0.

Если чип поддерживает несколько субпотоков воспроизведения или захвата, можно указать большее количество, но они должны быть правильно обработаны во время открытия/закрытия и других обратных вызовах. Если необходимо узнать, какой субпоток имеется в виду, его можно получить из структуры данных **snd_pcm_substream**, передаваемой в каждом обратном вызове, следующим образом:

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

После создания PCM необходимо установить операции для каждого потока PCM.

```
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);
```

Операции определяются обычно следующим образом:

```
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,
};
```

Все обратные вызовы описаны в подразделе Операции 34.

После установки операций, вы, вероятно, захотите предварительно выделить память для буфера. Для предварительного выделения просто вызовите следующую функцию:

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

По умолчанию она выделяет буфер до 64 КБ. Подробности управления буфером будут описаны дальше в разделе Управление буфером и памятью 64.

В **pcm->info_flags** можно дополнительно установить некоторую дополнительную информацию для этого PCM. Допустимые значения определяются в ***<sound/asound.h>*** как **SNDRV_PCM_INFO_XXX**, которые используются для указания параметров оборудования (смотрите ниже). Если ваша звуковая микросхема поддерживает только полудуплекс, укажите это следующим образом:

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

... А деструктор?

Деструктор для экземпляра PCM необходим не всегда. Так как PCM устройство будет освобождаться в кодом центрального уровня автоматически, вы не должны вызывать этот деструктор явно.

Деструктор будет необходим, если вы создали специальные внутренние объекты и необходимо их освободить. В таком случае укажите функцию деструктора в **pcm-**

>private_free:

Пример 5.2. Экземпляр PCM с деструктором

```
static void mychip_pcm_free(struct snd_pcm *pcm)
{
    struct mychip *chip = snd_pcm_chip(pcm);
    /* освобождаем свои данные */
    kfree(chip->my_private_pcm_data);
    /* делаем что-либо ещё */
    ....
}

static int __devinit snd_mychip_new_pcm(struct mychip *chip)
{
    struct snd_pcm *pcm;
    ....
    /* выделяем память для своих данных */
    chip->my_private_pcm_data = kmalloc(...);
    /* указываем деструктор */
    pcm->private_data = chip;
    pcm->private_free = mychip_pcm_free;
    ....
}
```

Рабочий указатель - хранение информации PCM

Когда открывается субпоток PCM, создаётся рабочий экземпляр PCM и присваивается субпотоку. Этот указатель доступен через **substream->runtime**. Это рабочий указатель хранит наибольшее количество информации, необходимой для управления PCM: копию конфигураций **hw_params** и **sw_params**, указатели буферов, данные о **mmap**, спин-блокировки и другое.

Определение рабочего экземпляра находится в **<sound/pcm.h>**. Вот содержимое этого файла:

```
struct _snd_pcm_runtime {
    /* -- Состояние -- */
    struct snd_pcm_substream *trigger_master;
    snd_timestamp_t trigger_tstamp; /* метка времени триггера */
    int overrange;
    snd_pcm_uframes_t avail_max;
    snd_pcm_uframes_t hw_ptr_base; /* Позиция во время перезагрузки буфера */
    snd_pcm_uframes_t hw_ptr_interrupt; /* Позиция во время прерывания */

    /* -- параметры оборудования -- */
    snd_pcm_access_t access; /* режим доступа */
    snd_pcm_format_t format; /* SNDRV_PCM_FORMAT_* */
    snd_pcm_subformat_t subformat; /* субформат */
    unsigned int rate; /* частота в Гц */
    unsigned int channels; /* каналы */
    snd_pcm_uframes_t period_size; /* размер периода */
```

```
    unsigned int periods; /* периоды */
    snd_pcm_uframes_t buffer_size; /* размер буфера */
    unsigned int tick_time; /* время тика */
    snd_pcm_uframes_t min_align; /* минимальное выравнивание для данного
формата */
    size_t byte_align;
    unsigned int frame_bits;
    unsigned int sample_bits;
    unsigned int info;
    unsigned int rate_num;
    unsigned int rate_den;

    /* -- программные параметры -- */
    struct timespec tstamp_mode; /* обновляемая метка времени mmap */
    unsigned int period_step;
    unsigned int sleep_min; /* минимальное число тиков для сна */
    snd_pcm_uframes_t start_threshold;
    snd_pcm_uframes_t stop_threshold;
    snd_pcm_uframes_t silence_threshold; /* Если шум меньше этого, происходит
заполнение тишиной */
    snd_pcm_uframes_t silence_size; /* размер заполнения тишиной */
    snd_pcm_uframes_t boundary; /* указывает на точку перехода в начало */

    snd_pcm_uframes_t silenced_start;
    snd_pcm_uframes_t silenced_size;

    snd_pcm_sync_id_t sync; /* идентификатор синхронизации оборудования */

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

    /* -- блокировка / планировщик -- */
    spinlock_t lock;
    wait_queue_head_t sleep;
    struct timer_list tick_timer;
    struct fasync_struct *fasync;

    /* -- закрытая секция -- */
    void *private_data;
    void (*private_free)(struct snd_pcm_runtime *runtime);

    /* -- описание оборудования -- */
    struct snd_pcm_hardware hw;
    struct snd_pcm_hw_constraints hw_constraints;

    /* -- обратные вызовы прерывания -- */
    void (*transfer_ack_begin)(struct snd_pcm_substream *substream);
    void (*transfer_ack_end)(struct snd_pcm_substream *substream);

    /* -- таймер -- */
    unsigned int timer_resolution; /* точность таймера */

    /* -- DMA -- */
```

```
    unsigned char *dma_area; /* область DMA */
    dma_addr_t dma_addr; /* адрес физической шины (недоступной основному CPU)
*/
    size_t dma_bytes; /* размер области DMA */

    struct snd_dma_buffer *dma_buffer_p; /* выделенный буфер */

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

Для операций (обратных вызовов) каждого звукового драйвера большинство из этих записей должны быть только читаемыми. Их изменяет/обновляет только центральный уровень PCM. Исключениями являются: описание оборудования (**hw**), обратные вызовы прерывания (**transfer_ack_xxx**), информация о буфере DMA и закрытые данные. Кроме того, при использовании стандартного метода выделения буфера с помощью ***snd_pcm_lib_malloc_pages()*** нет необходимости самостоятельно устанавливать информацию о DMA буфере.

Важные записи объясняются в следующих разделах.

Параметры оборудования

Дескриптор оборудования (структура **snd_pcm_hardware**) содержит параметры основной конфигурации оборудования. Прежде всего, необходимо определить их в обратном вызове ***open***. Обратите внимание, рабочий экземпляр имеет копию данного дескриптора, а не указатель на существующий дескриптор. То есть в обратном вызове ***open*** можно изменить скопированный дескриптор (**runtime->hw**), как это требуется. Например, если максимальное число каналов равно 1 только на некоторых моделях чипа, можно по-прежнему использовать один и тот же дескриптор оборудования и изменять **channels_max** позже:

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

Как правило, дескриптор оборудования будет таким, как показано ниже:

```
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,
};
```

- Поле **info** содержит тип и возможности этого PCM. Битовые флаги определены в ***<sound/asound.h>*** как **SNDRV_PCM_INFO_XXX**. Здесь, как минимум, необходимо указать, поддерживается ли **mmap** и какой из форматов чередования данных поддерживается. Если поддерживается, добавьте здесь флаг **SNDRV_PCM_INFO_MMAP**. В зависимости от того, поддерживает ли оборудование формат с чередованием или без чередования, должен быть установлен флаг **SNDRV_PCM_INFO_INTERLEAVED** или **SNDRV_PCM_INFO_NONINTERLEAVED**, соответственно. Если оба они поддерживаются, можно также установить оба.

  В приведённом выше примере, для режима OSS **mmap** указаны **MMAP_VALID** и **BLOCK_TRANSFER**. Обычно устанавливаются оба. Конечно, **MMAP_VALID** устанавливается только если **mmap** действительно поддерживается.

  Другими возможными флагами являются **SNDRV_PCM_INFO_PAUSE** и **SNDRV_PCM_INFO_RESUME**. Бит **PAUSE** означает, что этот PCM поддерживает операцию "пауза", а бит **RESUME** означает, что данный PCM поддерживает полноценную операцию "приостановка/возобновление". Если установлен флаг **PAUSE**, обратный вызов ***trigger***, описанный ниже, должен обрабатывать соответствующие команды (нажатие/отпускание паузы). Команды триггера приостановка/возобновление могут быть определены даже без флага **RESUME**. Подробности описаны в разделе Управления питанием 70.

  Если субпотоки PCM могут быть синхронизированы (как правило, синхронизированы потоки при начале/остановке воспроизведения и потоки захвата), можно также указать **SNDRV_PCM_INFO_SYNC_START**. В этом случае необходимо проверить связанный список субпотоков PCM в обратном вызове ***trigger***. Это будет описано в следующем разделе.

- Поле **formats** содержит битовые флаги поддерживаемых форматов (**SNDRV_PCM_FMTBIT_XXX**). Если оборудование поддерживает более одного формата, укажите все, объединив их с помощью ИЛИ. В приведённом выше примере указан знаковый 16-ти разрядный формат с прямым порядком байтов (сначала младший).

- Поле **rates** содержит битовые флаги поддерживаемых частот дискретизации (**SNDRV_PCM_RATE_XXX**). Если чип поддерживает произвольные частоты, дополнительно установите бит **CONTINUOUS**. Предопределённые биты частот дискретизации предоставляются только для типичных частот. Если ваш чип поддерживает нетрадиционные частоты, необходимо добавить бит **KNOT** и настроить ограничение оборудования вручную (объяснено далее).

- **rate_min** и **rate_max** определяют минимальную и максимальную частоту дискретизации. Они должны соответствовать битам, описывающим частоты дискретизации.

- **channel_min** и **channel_max** определяют, как вы могли уже ожидать, минимальное и

максимальное количество каналов.

- **buffer_bytes_max** определяет максимальный размер буфера в байтах. Не существует поля **buffer_bytes_min**, так как оно может быть вычислено исходя из минимального размера периода и минимального числа периодов. Таким образом, **period_bytes_min** и определяет минимальный и максимальный размер периода в байтах. **periods_max** и **periods_min** определяют максимальное и минимальное число периодов в буфере.

  "Период" - это термин, соответствующий в мире OSS фрагменту. Период определяет частоту с которой генерируются прерывания PCM. Эта частота сильно зависит от оборудования. Вообще, меньший размер периода даст вам большее число прерываний, то есть более точное управление. В случае захвата этот размер определяет входную задержку. С другой стороны, выходную задержку при воспроизведении определяет размер полного буфера.

- Существует также поле **fifo_size**. Оно определяет размер аппаратного буфера FIFO, но в настоящее время оно не используется ни в драйвере, ни в ALSA-Lib. Таким образом, вы можете игнорировать это поле.

Конфигурации PCM

Итак, давайте снова вернёмся к полям рабочего указателя PCM. Наиболее часто упоминаемыми полями в рабочем экземпляре являются конфигурации PCM. Конфигурации PCM сохраняются в рабочем экземпляре после того, как приложение отправляет данные **hw_params** через ALSA-Lib. Многие полей скопированы из структур **hw_params** и **sw_params**. Например, ***format*** хранит тип формата, выбранного приложением. Это поле содержит значение перечисления **SNDRV_PCM_FORMAT_XXX**.

Единственно, следует отметить, что во время выполнения размеры настроенного буфера и периода хранятся в "кадрах" . В мире ALSA 1 кадр = число каналов * размер сэмпла. Для преобразования между кадрами и байтами можно использовать вспомогательные функции **frames_to_bytes()** и **bytes_to_frames()**.

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

Кроме того, в кадрах так же хранятся многие программные параметры (**sw_params**). Пожалуйста, проверяйте тип поля. Для кадров как целое число без знака используется **snd_pcm_uframes_t**, а как целое знаковое число - **snd_pcm_sframes_t**.

Информация о буфере DMA

Буфер DMA определяется следующими четырьмя полями: **dma_area**, **dma_addr**, **dma_bytes** и **dma_private**. **dma_area** хранит указатель на буфер (логический адрес). Для копирования из/в этот указатель можно вызывать ***memcpy***. Физический адрес буфера хранит **dma_addr**. Это поле указывается только когда буфер является линейным буфером. **dma_bytes** хранит размер буфера в байтах. **dma_private** используется для распределителя памяти ALSA DMA.

Если для выделения буфера вы используете стандартную функцию ALSA,

snd_pcm_lib_malloc_pages(), эти поля устанавливаются центральным уровнем ALSA и вы ***не*** должны изменять их самостоятельно. Вы можете читать их, но не записывать в них. С другой стороны, если вы хотите выделять буфер самостоятельно, вам необходимо управлять им в обратном вызове **hw_params**. Обязательным является, по крайней мере, **dma_bytes**. **dma_area** необходимо, если буфер используется для ***mmap***. Если ваш драйвер не поддерживает ***mmap***, это поле не является необходимым. **dma_addr** также является необязательным. Вы также можете использовать по своему желанию **dma_private**.

Статус выполнения

Статус выполнения может быть передан с помощью **runtime->status**. Это указатель на структуру **snd_pcm_mmap_status**. Например, можно получить текущий аппаратный указатель DMA с помощью **runtime->status->hw_ptr**.

Указатель DMA приложения может быть получен с помощью **runtime->control**, который указывает на структуру **snd_pcm_mmap_control**. Однако, непосредственный доступ к этому параметру не рекомендуется.

Закрытые данные

Вы можете выделить объект для субпотока и сохранить его в **runtime->private_data**. Обычно это делается в обратном вызове open 35. Не путайте его с **pcm->private_data**. **pcm->private_data** обычно указывает на экземпляр статически созданного объекта чипа при создании PCM, а **runtime->private_data** указывает на динамическую структуру данных, созданную в обратном вызове PCM ***open***.

```
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;
    ....
}
```

Выделенный объект должен быть освобождён в обратном вызове close 35.

Обратные вызовы прерывания

Функция ***transfer_ack_begin*** и ***transfer_ack_end*** вызывается в начале и в конце ***snd_pcm_period_elapsed()***, соответственно.

Операции

Итак, теперь позвольте представить подробную информацию о каждом обратном вызове PCM (**ops**). В общем, каждый обратный вызов должен возвращать 0 в случае успеха или отрицательное число ошибки, такое, как **-EINVAL**. Для выбора подходящего номера ошибки рекомендуется проверить, какие значения возвращают другие части ядра, когда такой же вид запроса оканчивается неудачей.

Функции обратного вызова требуется по крайней мере один аргумент с указателем

snd_pcm_substream. Чтобы получить объект чипа из данного экземпляра субпотока, можно использовать следующий макрос:

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

Макрос читает **substream->private_data**, которые являются копией **pcm->private_data**. Вы можете изменить **substream->private_data**, если необходимо указать другие значения данных для субпотока PCM. Например, драйвер cmi8330 назначает другие **private_data** для воспроизведения и захвата, потому что он использует для воспроизведения и захвата два разных кодека (SB- и AD-совместимый).

Обратный вызов open

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

Он вызывается, когда субпоток PCM открывается.

Здесь вы как минимум должны инициализировать параметр **runtime->hw**. Как правило, это делается так:

```
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;
}
```

где **snd_mychip_playback_hw** является заранее сделанным описанием оборудования.

В этом обратном вызове можно выделить память для закрытых данных, как описано в разделе Закрытые данные 34.

Если конфигурация оборудования требует больше ограничений, установите эти аппаратные ограничения здесь. Для более подробной информации смотрите Ограничения 41.

Обратный вызов close

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

Очевидно, что он вызывается, когда субпоток PCM закрывается.

Любой экземпляр закрытых данных для субпотока PCM, созданный в обратном вызове open, будет здесь освобождаться.

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

```
{
    ....
    kfree( substream->runtime->private_data);
    ....
}
```

Обратный вызов ioctl

Он используется для всех специальных вызовов **ioctl** PCM. Но обычно можно передавать универсальный обратный вызов **ioctl**, ***snd_pcm_lib_ioctl***.

Обратный вызов hw_params

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

Он вызывается, когда приложением устанавливается параметр оборудования (**hw_params**), то есть после того, как для субпотока PCM определены размер буфера, величина периода, формат и другие.

В этом обратном вызове должно быть сделано большинство аппаратных установок, в том числе выделены буферы.

Параметры для инициализации извлекаются макросом **params_xxx()**. Чтобы выделить буфер, можно вызвать вспомогательную функцию,

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

snd_pcm_lib_malloc_pages() доступна только если буферы DMA были выделены заранее. Для более подробной информации смотрите раздел Типы буферов 64.

Заметим, что этот обратный вызов и ***prepare*** могут быть вызваны для инициализации несколько раз. Например, эмуляция OSS может вызывать эти обратные вызовы при всех изменениях через его **ioctl**.

Таким образом, вы должны быть осторожны и не выделять одни и те же буферы много раз, что приведёт к утечкам памяти! Вызов вспомогательной функции во много раз лучше. Он освободит предыдущий буфер автоматически, если он уже был выделен.

Обратите также внимание, что это обратный вызов не атомарный (планируемый). Это важно, потому что обратный вызов ***trigger*** является атомарным (не планируемым). То есть в обратном вызове ***trigger*** мьютексы или любые другие функции, связанные с планировщиком, недоступны. Подробности смотрите в подразделе Атомарность 41.

Обратный вызов hw_free

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

Он вызывается для освобождения ресурсов, выделенных через ***hw_params***. Например,

освобождение буфера, выделенного через ***snd_pcm_lib_malloc_pages()***, осуществляется следующим вызовом:

```
snd_pcm_lib_free_pages(substream);
```

Эта функция всегда вызывается перед вызовом обратного вызова ***close***. Кроме того, этот метод так же может быть вызван несколько раз. Проверяйте, не был ли ресурс уже освобождён.

Обратный вызов prepare

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

Этот обратный вызов вызывается, когда PCM является "подготавливаемым". Здесь можно установить тип формата, частоту дискретизации и так далее. Отличие от ***hw_params*** в том, что обратный вызов ***prepare*** будет вызываться каждый раз, когда вызывается ***snd_pcm_prepare()***, то есть при восстановлении после опустошения буфера и так далее.

Обратите внимание, что этот обратный вызов теперь не атомарный. В этом обратном вызове можно безопасно использовать функции, связанные с планировщиком задач.

В этом и следующих обратных вызовах можно обращаться к параметрам с помощью объекта **runtime**, **substream->runtime**. Например, чтобы получить текущую частоту дискретизации, формат или число каналов, обращаясь к **runtime->rate**, **runtime->format** или **runtime->channels**, соответственно. Физический адрес выделенного буфера находится в **runtime->dma_area**. Размеры буфера и периода находятся в **runtime->buffer_size** и **runtime->period_size**, соответственно.

Будьте осторожны, потому что этот обратный вызов тоже будет вызываться много раз при каждой установке параметров.

Обратный вызов trigger

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

Он вызывается, когда PCM запускается, останавливается или приостанавливается (ставится в паузу).

Действие указывается во втором аргументе, **SNDRV_PCM_TRIGGER_XXX** в ***<sound/pcm.h>***. В этом обратном вызове должны быть определены по крайней мере команды **START** и **STOP**.

```
switch (cmd) {
case SNDRV_PCM_TRIGGER_START:
    /* делаем что-нибудь для запуска движка PCM */
    break;
case SNDRV_PCM_TRIGGER_STOP:
    /* делаем что-нибудь для остановки движка PCM */
    break;
default:
```

```
    return -EINVAL;
}
```

Если PCM поддерживает операцию "пауза" (указываемую в поле **info** при описании оборудования), здесь также должны быть обработаны команды **PAUSE_PUSE** и **PAUSE_RELEASE**. Первая является командой приостановки PCM, а последняя - запуском PCM снова.

Если PCM поддерживает операцию приостановить/возобновить, независимо от полной или частичной поддержки приостановки/возобновления также должны быть обработаны команды **SUSPEND** и **RESUME**. Эти команды выдаются, когда изменяется статус управления питанием. Очевидно, что команды **SUSPEND** и **RESUME** приостанавливают и возобновляют субпоток PCM, и, как правило, они идентичны командам **STOP** и **START**, соответственно. Подробности смотрите в разделе Управление питанием 70.

Как упоминалось, этот обратный вызов является атомарным. Нельзя вызывать функции, которые могут заснуть. Обратный вызов ***trigger*** должен быть настолько малым, как возможно, действительно только переключая DMA. Остальное должно быть правильно проинициализировано заранее обратными вызовами ***hw_params*** и ***prepare***.

Обратный вызов pointer

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

Этот обратный вызов вызывается, когда центральный уровень PCM запрашивает текущее положение в аппаратном буфере. Позиция должна быть возвращена в кадрах, в диапазоне от **0** до **buffer_size - 1**.

Он обычно вызывается из процедуры обновления буфера в центральном уровне PCM, которая вызывается, если в процедуре обработки прерывания вызывается ***snd_pcm_period_elapsed()***. Затем центральный уровень PCM обновляет позицию и рассчитывает свободное пространство, будит спящие потоки опроса и так далее.

Этот обратный вызов также атомарен.

Обратные вызовы copy и silence

Эти обратные вызовы не являются обязательными и в большинстве случаев могут быть опущены. Эти обратные вызовы используются, когда аппаратный буфер не может быть в обычном пространстве памяти. Некоторые чипы имеют свой собственный аппаратный буфер, который не является отображаемым. В таком случае, необходимо передавать данные вручную из буфера в памяти в аппаратный буфер. Эти обратные вызовы также должны быть определены в случае, если буфер не непрерывен и в физическом, и виртуальном пространстве памяти.

Если эти два обратных вызова определены, ими выполняются операции копирования и установки тишины. Подробности будут описаны позже в разделе Управление буфером и памятью 64.

Обратный вызов ack

Этот обратный вызов также не является обязательным. Этот обратный вызов вызывается, когда при операциях чтения или записи обновляется **appl_ptr**. Некоторым драйверам, подобным emu10k1-fx и cs46xx, необходимо отслеживать текущее значение **appl_ptr** для внутреннего буфера и этот обратный вызов используется только для этой цели.

Этот обратный вызов атомарен.

Обратный вызов page

Этот обратный вызов тоже не является обязательным. Этот обратный вызов используется в основном для не непрерывных буферов. **mmap** делает этот обратный вызов, чтобы получить адрес страницы. Некоторые примеры будут также объяснены позже в разделе Управление буфером и памятью 64.

Обработчик прерывания

Последней составляющей PCM является обработчик прерывания PCM. Роль обработчика прерывания PCM в звуковом драйвере - обновление позиции в буфере и информирование центрального уровня PCM, когда позиция буфера проходит через установленный размер периода. Чтобы сообщить это, вызывается функция ***snd_pcm_period_elapsed()***.

Есть несколько типов звуковых чипов для генерации прерываний.

Прерывания на границе периода (фрагмента)

Это наиболее часто встречающийся тип: оборудование генерирует прерывание на границе каждого периода. В этом случае можно вызывать ***snd_pcm_period_elapsed()*** в каждом прерывании.

snd_pcm_period_elapsed() принимает в качестве аргумента указатель субпотока. Таким образом, необходимо сохранять указатель субпотока доступным из экземпляра чипа. Например, определите поле **substream** в объекте чипа для хранения рабочего указателя субпотока, и установите значение указателя в обратном вызове ***open*** (и очистите в обратном вызове ***close***).

Если вы захватили спин-блокировку в обработчике прерывания и эта блокировка также используется в других обратных вызовах PCM, то вы должны снять блокировку перед вызовом ***snd_pcm_period_elapsed()***, потому что ***snd_pcm_period_elapsed()*** вызывает внутри другие функции обратного вызова PCM.

Типичный код будет таким:

Пример 5.3. Случай обработчика прерываний #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)) {
        /* вызов обновления, перед этим снимаем блокировку */
        spin_unlock(&chip->lock);
        snd_pcm_period_elapsed(chip->substream);
        spin_lock(&chip->lock);
        /* подтверждение прерывания, если необходимо */
    }
    ....
    spin_unlock(&chip->lock);
    return IRQ_HANDLED;
}
```

Прерывания по высокочастотному таймеру

Это происходит, когда оборудование не может генерировать прерывания на границе периода, но выдаёт прерывания по таймеру на фиксированной частоте таймера (например, драйверы es1968 или ymfpci). В этом случае во время каждого прерывания необходимо проверять текущую позицию оборудования и накапливать длины обрабатываемых кусочков. Когда накопленный размер превысит размер периода, вызовите ***snd_pcm_period_elapsed()*** и сбросьте значение накопителя.

Типичный код был бы примерно таким.

Пример 5.4. Случай обработчика прерываний #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;
        /* получаем текущее значение указателя оборудования (в кадрах) */
        last_ptr = get_hw_ptr(chip);
        /* вычисляем число обработанных кадров со времени
         * последнего обновления
         */
        if (last_ptr < chip->last_ptr)
            size = runtime->buffer_size + last_ptr
                    - chip->last_ptr;
        else
            size = last_ptr - chip->last_ptr;
        /* запоминаем последнюю точку обновления */
        chip->last_ptr = last_ptr;
        /* накапливаем длину */
        chip->size += size;
        /* граница периода превышена? */
        if (chip->size >= runtime->period_size) {
            /* сбрасываем значение накопителя */
            chip->size %= runtime->period_size;
            /* вызываем функцию обновления */
            spin_unlock(&chip->lock);
```

```
            snd_pcm_period_elapsed(substream);
            spin_lock(&chip->lock);
        }
        /* подтверждение прерывания, если необходимо */
    }
    ....
    spin_unlock(&chip->lock);
    return IRQ_HANDLED;
}
```

О вызове snd_pcm_period_elapsed()

В обоих случаях, даже если истёк более, чем один период, вы не должны вызывать ***snd_pcm_period_elapsed()*** много раз. Вызывайте только один раз. А уровень PCM проверит текущий указатель оборудования и обновит до последнего значения статуса.

Атомарность

Одной из наиболее важных (и, таким образом, трудной для отлаживания) проблем в программировании ядра является состояние гонок. В ядре Linux, они, как правило, избегаются с помощью спин-блокировок, мьютексов или семафоров. В общем, если в обработчике прерывания может произойти состояние гонок, он должен быть управляемым атомарно, а для защиты критических секций вы должны использовать спин-блокировки. Если критическая секция не является кодом обработчика прерывания и если приемлемо относительно большое время выполнения, вы должны взамен использовать мьютексы или семафоры.

Как уже говорилось, некоторые обратные вызовы PCM являются атомарными, а некоторые - нет. Например, обратный вызов ***hw_params*** не является атомарным, а обратный вызов ***trigger*** - атомарный. Это означает, что последний вызывается при уже удерживаемой центральным уровнем PCM спин-блокировке. Пожалуйста, примите эту атомарность во внимание при выборе схемы блокировки в обратных вызовах.

В атомарных обратных вызовах вы не можете использовать функции, которые могут вызывать планировщик или заснуть. Семафоры и мьютексы могут спать, и, следовательно, они не могут быть использованы внутри атомарных обратных вызовах (например, в обратном вызове ***trigger***). Для реализации какой-либо задержки в таком обратном вызове, пожалуйста, используйте ***udelay()*** или ***mdelay()***.

Все три атомарные обратные вызовы (***trigger***, ***pointer*** и ***ack***) вызываются с запрещёнными локальными прерываниями.

Ограничения

Если ваш чип поддерживает нетрадиционные частоты дискретизации или только некоторые типы сэмплов, необходимо установить ограничение для использования.

Например, чтобы ограничить частоты дискретизации до некоторых поддерживаемых значений, используйте ***snd_pcm_hw_constraint_list()***. Вы должны вызвать эту функцию в обратном вызове ***open***.

Пример 5.5. Пример аппаратных ограничений

```
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;
    ....
}
```

Есть много разных ограничений. Для получения полного списка посмотрите в ***sound/pcm.h***. Вы даже можете определить свои собственные ограничивающие правила. Например, предположим, что **my_chip** может управлять субпотоком из 1 канала, только если форматом является S16_LE, в противном случае он поддерживает любой формат, указанный в структуре **snd_pcm_hardware** (или в любом другом **constraint_list**). Вы можете создать правило так:

Пример 5.6. Пример аппаратных ограничений для каналов

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

Затем необходимо вызвать эту функцию, чтобы добавить правило:

```
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);
```

Функция установки правила вызывается, когда приложение устанавливает число каналов. Но приложение может установить формат перед числом каналов. Таким образом, необходимо

также определить обратное правило:

Пример 5.7. Пример аппаратных ограничений для каналов

```
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;
}
```

... а в обратном вызове ***open***:

```
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);
```

Я не буду давать здесь более подробную информацию, сказав, "используйте исходники".

Глава 6. Интерфейс Control

Общие сведения

Интерфейс управления **control** широко используется для многих переключателей, регуляторов и тому подобному, доступному из пользовательского пространства. Наиболее важным является использование интерфейса микшера. Другими словами, начиная с ALSA 0.9.x, всё необходимое для микшера реализовано как API управления ядра.

ALSA имеет отдельный модуль управления AC97. Если ваш чип поддерживает только AC97 и больше ничего, вы можете пропустить этот раздел.

API управления определён в ***<sound/control.h>***. Подключите этот файл, если хотите добавить свои собственные элементы управления.

Создание элементов управления

Чтобы создать новый элемент управления, необходимо определить три следующих обратных вызова: ***info***, ***get*** и ***put***. Затем создать объект **struct snd_kcontrol_new**, выглядящий следующим образом:

Пример 6.1. Создание элемента управления

```
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
};
```

Чаще всего элемент управления создаётся с помощью ***snd_ctl_new1()***, и в таком случае, можно добавить к определению префикс **__devinitdata**, как показано выше.

Поле **iface** определяет тип элемента управления, **SNDRV_CTL_ELEM_IFACE_XXX**, который, как правило, **MIXER**. Для общих элементов управления, которые не являются логическими частями микшера, используйте **CARD**. Если элемент управления тесно связан с некоторыми определёнными устройствами звуковой карты, используйте **HWDEP**, **PCM**, **RAWMIDI**, **TIMER**, или **SEQUENCER**, и укажите номер устройства с помощью полей **device** и **subdevice**.

Поле **name** - строка идентификатора. Начиная с ALSA 0.9.x имя элемента управления очень важно, потому что его роль определяется из его названия. Есть предопределённые стандартные имена элементов управления. Подробности описаны в подразделе Имена элементов управления 45.

Поле **index** содержит порядковый номер этого элемента управления. Если существуют

несколько разных элементов управления с одинаковым именем, их можно отличить по номеру индекса. Это случай, когда на карте имеется несколько кодеков. Если индекс равен нулю, вышеописанный параметр можно опустить.

Поле **access** содержит тип доступа данного элемента управления. Здесь указывается комбинация битовых масок, **SNDRV_CTL_ELEM_ACCESS_XXX**. Подробности будут объяснены в подразделе Флаги доступа 46.

Поле **private_value** содержит произвольное значение long integer для этого объекта. При использовании универсальных обратных вызовов ***info***, ***get*** и ***put***, через это поле можно передать какое-либо значение. Если необходимы несколько небольших чисел, их можно побитово объединить. Или в этом поле можно передать указатель (приведённый к типу unsigned long) на какой-то объект.

Поле **tlv** может быть использовано для предоставления метаданных об элементе управления; смотрите подраздел Метаданные 49.

Остальные три поля являются функциями обратного вызова.

Имена элементов управления

Для определения имён элементов управления есть некоторые стандарты. Имя элемента управления обычно состоит из трёх частей: "ИСТОЧНИК НАПРАВЛЕНИЕ ФУНКЦИЯ".

Первая, **ИСТОЧНИК**, указывает источник для элемента управления, и представляет собой такую строку, как "Master", "PCM", "CD" и "Line". Есть много предопределенных источников.

Вторая, **НАПРАВЛЕНИЕ**, является одной из следующих строк в зависимости от направления управления: "Playback" ("Воспроизведение"), "Capture" ("Захват"), "Bypass Playback" ("Обход Воспроизведение") и "Bypass Capture" ("Обход Захвата"). Она может быть опущена, что означает оба направления, воспроизведение и захват.

Третья, **ФУНКЦИЯ**, - одна из следующих строк в соответствии с функцией управления: "Switch" ("Переключатель"), "Volume" ("Громкость") и "Route" ("Маршрут").

Примерами имён элементов управления являются, таким образом, "Master Capture Switch" ("Главный переключатель захвата") или "PCM Playback Volume" ("Громкость воспроизведения PCM").

Есть некоторые исключения:

Основной захват и воспроизведение

"Capture Source", "Capture Switch" и "Capture Volume" используются для источника, переключателя и громкости основного захвата (входа). Аналогично, "Playback Switch" ("Переключатель воспроизведения") и "Playback Volume" ("Громкость воспроизведения") используются для переключателя уровня и громкости основного выхода.

Элементы управления тембром

Переключатель управления тембром и регуляторы указываются как "Tone Control - XXX",

например, "Tone Control - Switch", "Tone Control - Bass", "Tone Control - Center".

Элементы управления 3D

Переключатели управлением 3D и регуляторы указываются как "3D Control - XXX", например, "3D Control - Switch", "3D Control - Center", "3D Control - Space".

Увеличение усиления микрофона

Переключатель увеличения усиления микрофона указывается как "Mic Boost" или "Mic Boost (6 дБ)".

Более точную информацию можно найти в ***Documentation/sound/alsa/ControlNames.txt***.

Флаги доступа

Флаг **access** представляет собой битовую маску, которая определяет тип доступа данного элемента управления. Типом доступа по умолчанию является **SNDRV_CTL_ELEM_ACCESS_READWRITE**, что означает разрешение на чтение и запись для этого элемента управления. Когда флаг доступа опущен (то есть = 0), он рассматривается как и доступ по умолчанию на чтение и запись.

Когда элемент управление предназначен только для чтения, вместо этого указывается **SNDRV_CTL_ELEM_ACCESS_READ**. В этом случае вы не должны определять обратный вызов ***put***. Аналогичным образом, когда элемент управления предназначен только для записи (хотя это редкий случай), можно использовать взамен флаг **WRITE**, и нет необходимости в обратном вызове ***get***.

Если значение элемента управления происходит часто (например, измеритель уровня, VU-метр), должен быть указан флаг **VOLATILE** (нестабильный). Это означает, что элемент управления может быть изменён без уведомления. Приложения должны постоянно опрашивать такой элемент управления.

Когда элемент управления неактивен, устанавливается также флаг **INACTIVE**. Для изменения записи разрешений записи есть флаги **LOCK** (блокировка) и **OWNER** (владелец).

Обратные вызовы

Обратный вызов info

Обратный вызов ***info*** используется для получения подробной информации о данном элементе управления. Он должен хранить эти значения в объекте **struct snd_ctl_elem_info**. Например, для булева элемента управления из одного элемента:

Пример 6.2. Пример обратного вызова info

```
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;
}
```

Поле **type** определяет тип управления. Используются **BOOLEAN**, **INTEGER**, **ENUMERATED**, **BYTES**, **IEC958** и **INTEGER64**. Поле **count** определяет число элементов в этом элементе. Например, стереофонический регулятор громкости имел бы **count** = 2. Поле **value** представляет собой объединение и значения хранятся в зависимости от типа. Логический и целый типы являются идентичными.

Перечислимый тип немного отличается от других. Необходимо будет установить строку для типа, индекс которого используется в настоящее время.

```
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;
}
```

Для удобства доступны некоторые обычные обратные вызовы получения информации: ***snd_ctl_boolean_mono_info()*** и ***snd_ctl_boolean_stereo_info()***. Очевидно, что первый является обратным вызовом ***info*** для одноканального булева элемента, так же, как показанный выше ***snd_myctl_mono_info***, а последний - для двухканального булева элемента.

Обратный вызов get

Этот обратный вызов используется для чтения текущего значения элемента управления и передачи в пространство пользователя.

Пример 6.3. Пример обратного вызова get

```
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;
}
```

Поле **value** зависит от типа элемента управления, а также от обратного вызова ***info***. Например, драйвер sb использует это поле для хранения регистра смещения, битов сдвига и

битовой маски. Поле **private_value** устанавливается следующим образом:

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

и извлекается в обратных вызовах следующим образом

```
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;
    ....
}
```

В обратном вызове ***get*** вы должны заполнить все элементы, если имеется более одного элемента, то есть **count** > 1. В приведённом выше примере заполняется только один элемент (**value.integer.value[0]**), так как предполагается, что **count** = 1.

Обратный вызов put

Этот обратный вызов используется для записи значения, полученного из пользовательского пространства.

Пример 6.4. Пример обратного вызова put

```
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;
}
```

Как видно из примера, необходимо вернуть 1, если значение изменилось. Если значение не изменилось, возвращается 0. Если произошла какая-либо фатальная ошибка, возвращается отрицательный код ошибки, как обычно.

Также как и в обратном вызове ***get***, если элемент управления имеет более одного элемента, в этом обратном вызове должны быть установлены значения всех элементов.

Обратные вызовы не атомарны

Все эти три функции обратного вызова в своей основе являются не атомарными.

Конструктор

Наконец, когда всё будет готово, можно создать новый элемент управления. Чтобы создать элемент управления, могут быть вызваны две функции, ***snd_ctl_new1()*** и ***snd_ctl_add()***.

В простейшем случае это можно сделать следующим образом:

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

где **my_control** является объектом **struct snd_kcontrol_new**, созданным выше, а **chip** - указатель на объект, передаваемый в **kcontrol->private_data**, который может быть использован в обратных вызовах.

snd_ctl_new1() создаёт новый экземпляр **snd_kcontrol** (именно поэтому определение **my_control** может быть с префиксом **__devinitdata**), а ***snd_ctl_add*** связывает данный компонент управления с картой.

Уведомление об изменении

Если в процедуре обработки прерывания необходимо изменить и обновить какой-нибудь элемент управления, можно вызвать ***snd_ctl_notify()***.
Например,

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

Эта функция для уведомления принимает указатель на карту, маску события и идентификатор элемента управления. Маска события определяет типы уведомлений, например, в приведённом выше примере сообщается об изменении управляемых значений. Указатель идентификатора - это указатель на **struct snd_ctl_elem_id**, получающую уведомление. Вы можете найти некоторые примеры в ***es1938.c*** или ***es1968.c*** в обработке аппаратных прерываний громкости.

Метаданные

Чтобы предоставить информацию о значениях элемента управления микшером в дБ, используйте макрос **DECLARE_TLV_xxx** из ***<sound/tlv.h>***, чтобы определить переменную, содержащую эту информацию, установите поле **tlv.p**, чтобы указать на эту переменную, и включите флаг **SNDRV_CTL_ELEM_ACCESS_TLV_READ** в поле **access** следующим образом:

```
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,
};
```

Макрос **DECLARE_TLV_DB_SCALE** указывает информацию об элементе управления микшером, каждый шаг значения которого изменяет своё значение в дБ на постоянную величину, указанную в дБ. Первым параметром является имя определяемой переменной. Вторым параметром является минимальное значение, в единицах 0.01 дБ. Третьим параметром является величина шага, в единицах 0.01 дБ. Установите четвёртый параметр в 1, если минимальное значение на самом деле отключает данный элемент управления (выключает звук).

Макрос **DECLARE_TLV_DB_LINEAR** указывает информацию об элементе управления микшером, значение которого влияет на выход линейно. Первым параметром является имя определяемой переменной. Вторым параметром является минимальное значение, в единицах 0.01 дБ. Третьим параметром является максимальное значение, в единицах 0.01 дБ. Если минимальное значение отключает данный элемент управления, установить второй параметр в **TLV_DB_GAIN_MUTE**.

Глава 7. API для кодека AC97

Общие сведения

Уровень кодека AC97 в ALSA чётко определён, и вам не придётся писать много кода, чтобы им управлять. Необходимы только процедуры низкоуровневого управления. API кодека AC97 определён в ***<sound/ac97_codec.h>***.

Полный пример кода

Пример 7.1. Пример интерфейса AC97

```
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;
    ....
    /* здесь из кодека читаем значение регистра */
    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;
    ....
    /* записываем в кодек значение данного регистра */
}

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);
}
```

Конструктор

Чтобы создать экземпляр AC97, сначала вызывается ***snd_ac97_bus*** используя объект **ac97_bus_ops_t**, содержащий функции обратного вызова.

```
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);
```

Объект **bus** является общим объектом для всего, связанного с экземплярами AC97.

А затем вызывается ***snd_ac97_mixer()*** с объектом **struct snd_ac97_template** вместе с указателем **bus**, созданным до этого.

```
struct snd_ac97_template ac97;
int err;

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

где **chip->ac97** является указателем на вновь созданный экземпляр **ac97_t**. В данном случае указатель **chip** сохраняется в закрытых данных, так что функции обратного вызова для чтения/записи могут обращаться к этому экземпляру **chip**. Экземпляр **ac97** не обязательно хранить в объекте **chip**. Если вам необходимо изменять значения регистров из драйвера или необходимо приостанавливать/возобновлять работу кодеков AC97, сохраните этот указатель для передачи соответствующим функциям.

Обратные вызовы

Стандартными обратными вызовами являются ***read*** и ***write***. Очевидно, что они соответствуют низкоуровневым функциям обращения к оборудованию для чтения и записи.

Обратный вызов ***read*** возвращает значение регистра, указанного в аргументе.

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

Здесь **chip** может быть получен приведением **ac97->private_data**.

Обратный же вызов ***write*** используется для установки значения регистра.

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

Эти обратные вызовы не являются атомарными, подобно обратным вызовам API управления.

Есть также другие функции обратного вызова: ***reset***, ***wait*** и ***init***.

Обратный вызов ***reset*** используется для сброса кодека. Если чип требует особый способ сброса, можно определить этот обратный вызов.

Обратный вызов ***wait*** используется для добавления некоторого времени ожидания в стандартную инициализацию кодека. Если чип требует дополнительного времени ожидания, определите этот обратный вызов.

Обратный вызов ***init*** используется для дополнительной инициализации кодека.

Обновление регистров в драйвере

Если необходимо обращаться к кодеку из драйвера, можно вызывать следующие функции: ***snd_ac97_write()***, ***snd_ac97_read ()***, ***snd_ac97_update()*** и ***snd_ac97_update_bits()***.

Обе функции, ***snd_ac97_write()*** и ***snd_ac97_update()***, используются для установки значения указанного регистра (**AC97_XXX**). Разница между ними в том, что ***snd_ac97_update()*** не записывает значение, если данное значение уже установлено, а ***snd_ac97_write()*** переписывает значение всегда.

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

snd_ac97_read() используется для чтения значения заданного регистра. Например,

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

snd_ac97_update_bits() используется для обновления некоторых битов в заданном регистре.

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

Кроме того, есть функция изменения частоты дискретизации (значение в указанном регистре, таком как **AC97_PCM_FRONT_DAC_RATE**), если кодеком поддерживается VRA (Variable Rate PCM audio, звук с PCM и изменяемой частотой дискретизации) или DRA (Double-Rate PCM audio, звук с PCM с удвоенной частотой дискретизации): ***snd_ac97_set_rate()***.

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

Для установки частоты дискретизации доступны следующие регистры: **AC97_PCM_MIC_ADC_RATE**, **AC97_PCM_FRONT_DAC_RATE**, **AC97_PCM_LR_ADC_RATE**, **AC97_SPDIF**. Если указан **AC97_SPDIF**, регистр на самом деле не изменяется, а будут обновляться соответствующие биты статуса IEC958.

Регулировка тактовой частоты

В некоторых чипах частота кодека не 48000, а используется частота PCI (для экономии кварца!). В этом случае измените поле **bus->clock** на соответствующее значение. Например, драйверы intel8x0 и es1968 имеют свои собственные функции для чтения частоты.

Файлы интерфейса Proc

ALSA AC97 интерфейс создаёт файлы ***proc***, такие как ***/proc/asound/card0/codec97#0/ac97#0-0*** и ***ac97#0-0+regs***. Можно обращаться к этим файлам, чтобы увидеть текущее состояние и регистры кодека.

Несколько кодеков

Если на одной карте есть несколько кодеков, необходимо вызывать ***snd_ac97_mixer()*** несколько раз с **ac97.num=1** или большим значением. Поле **num** указывается количество кодеков.

Если указано несколько кодеков, необходимо либо писать разные функции обратного вызова для каждого кодека, либо в процедурах обратного вызова проверять **ac97->num**.

Глава 8. Интерфейс MIDI (MPU401-UART)

Общие сведения

Многие звуковые карты имеют встроенные интерфейсы MIDI (MPU401-UART). Если звуковая карта поддерживает стандартный интерфейс MPU401-UART, скорее всего, можно использовать ALSA MPU401-UART API. MPU401-UART API определён в ***<sound/mpu401.h>***.

Некоторые звуковые чипы имеют похожую, но немного отличающуюся реализацию mpu401. Например, emu10k1 имеет свои собственные процедуры mpu401.

Конструктор

Чтобы создать объект **rawmidi**, вызовите ***snd_mpu401_uart_new()***.

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

Первым аргументом является указатель карты, а вторым - индекс этого компонента. Можно создать до 8 устройств rawmidi.

Третий аргумент - тип оборудования, **MPU401_HW_XXX**. Если он не какой-то особый, можно использовать **MPU401_HW_MPU401**.

4-м аргументом является адрес порта ввода/вывода. Многие обратно совместимые MPU401 имеют такой порт ввода/вывода, как 0x330. Или он может быть частью его собственной области ввода/вывода PCI. Это зависит от дизайна чипа.

Пятый аргумент - битовый флаг для дополнительной информации. Когда вышеупомянутый адрес порта ввода/вывода является частью области ввода/вывода PCI, порт ввода/вывода MPU401 может быть уже выделен (зарезервирован) самим драйвером. В таком случае передайте битовый флаг **MPU401_INFO_INTEGRATED** и уровень MPU401-UART выделит порты ввода/вывода самостоятельно.

Если контроллер поддерживает только поток ввод или вывода MIDI, передайте соответственно битовый флаг **MPU401_INFO_INPUT** или **MPU401_INFO_OUTPUT**. В таком случае экземпляр rawmidi создаётся как однопоточный.

Битовый флаг **MPU401_INFO_MMIO** используется для изменения метода доступа к MMIO (через ***readb*** и ***writeb***) вместо ***iob*** и ***outb***. В этом случае в ***snd_mpu401_uart_new()*** необходимо передать отображённый в пространство памяти адрес ввода/вывода.

Когда установлен **MPU401_INFO_TX_IRQ**, выходной поток не контролируется в обработчике прерывания, установленном по умолчанию. Драйвер должен сам вызывать ***snd_mpu401_uart_interrupt_tx()*** для начала обработки выходного потока в обработчике прерывания.

Обычно, адрес порта соответствует командному порту, а адрес порта + 1 соответствует порту данных. Если это не так, можно изменить поле ***cport*** структуры **snd_mpu401** вручную

позже. Однако, указатель **snd_mpu401** функцией ***snd_mpu401_uart_new()*** не возвращается явно. Вы должны явно привести **rmidi->private_data** к типу **snd_mpu401**,

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

и переустановить **cport**, как требуется:

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

(а как установить порт данных?)

Шестой аргумент указывает номер прерывания для UART. Если это прерывание уже выделено, передайте 0 в 7-м аргументе (**irq_flags**). В противном случае передайте флаги для выделения прерывания (биты **SA_XXX**) к нему, и прерывание будет зарезервировано уровнем MPU401-UART. Если карта не генерирует прерывания UART, в качестве номера прерывания передайте -1. Тогда для опроса будут вызываться прерывания по таймеру.

Обработчик прерывания

Если прерывание выделено в ***snd_mpu401_uart_new()***, используется закрытый (встроенный) обработчик прерывания, поэтому не придётся ничего делать, кроме создания создания объекта MPU401. В противном случае, необходимо явно вызывать ***snd_mpu401_uart_interrupt()***, если прерывание UART вызывается и управляется в собственном обработчике прерывания.

В этом случае в качестве второго аргумента ***snd_mpu401_uart_interrupt()*** необходимо передать **private_data** возвращённого из ***snd_mpu401_uart_new()*** объекта **rawmidi**.

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

Глава 9. Интерфейс RawMIDI

Обзор

Интерфейс **raw MIDI** (необработанный MIDI) используется для доступа к аппаратным портам MIDI, которые могут быть доступны как поток байтов. Он не используется для чипов-синтезаторов, которые непосредственно не понимают MIDI.

ALSA обрабатывает управление файлом и буфером. Все, что необходимо сделать, это написать код для перемещения данных между буфером и оборудованием.

API rawMIDI определён в ***<sound/rawmidi.h>***.

Конструктор

Чтобы создать устройство rawmidi, вызовите функцию **snd_rawmidi_new**:

```
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;
```

Первым аргументом является указателем карты, второй аргумент - строка идентификации.

Третий аргумент является индексом этого компонента. Вы можете создать до 8 устройств rawmidi.

Четвертый и пятый аргументы - это количество выходных и входных субпотоков, соответственно, этого устройства (субпоток эквивалентен порту MIDI).

Чтобы указать возможности устройства, установите поле **info_flags**.
Установите **SNDRV_RAWMIDI_INFO_OUTPUT**, если есть хотя бы один выходной порт, **SNDRV_RAWMIDI_INFO_INPUT**, если есть хотя бы один входной порт, и **SNDRV_RAWMIDI_INFO_DUPLEX**, если устройство может одновременно обрабатывать ввод и вывод.

После создания устройства rawmidi для каждого субпотока необходимо установить операции (обратные вызовы). Для установки операций для всех субпотоков устройства имеются вспомогательные функции:

```
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);
```

Операции обычно определяются следующим образом:

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

Эти обратные вызовы описаны в разделе Обратные вызовы 58.

Если есть более одного субпотока, каждому из них необходимо дать уникальное имя:

```
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);
}
/* что-нибудь для SNDRV_RAWMIDI_STREAM_INPUT */
```

Обратные вызовы

Во всех обратных вызовах закрытые данные, которые установлены для устройства rawmidi, могут быть доступны как **substream->rmidi->private_data**.

Если имеется более одного порта, ваши обратные вызовы могут определить индекс порта из данных структуры **snd_rawmidi_substream**, передаваемой в каждый обратный вызов:

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

Обратный вызов open

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

Он вызывается, когда субпоток открывается. Вы можете здесь проинициализировать оборудование, но ещё не должны начинать передачу/приём данных.

Обратный вызов close

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

Угадайте, зачем.

Последовательность работы обратных вызовов ***open*** и ***close*** устройства rawmidi управляется с помощью мьютекса и они могут спать.

Обратный вызов trigger для выходных субпотоков

```
static void snd_xxx_output_trigger(struct snd_rawmidi_substream *substream,
int up);
```

Он вызывается с ненулевым параметром **up**, когда есть данные в буфере субпотока, который должен быть передан.

Чтобы прочитать данные из буфера, вызовите ***snd_rawmidi_transmit_peek***. Она вернёт количество прочитанных байт; оно будет меньше, чем количество запрошенных байтов, если в буфер нет больше данных. После того, как данные были успешно переданы, вызовите ***snd_rawmidi_transmit_ack***, чтобы удалить данные из буфера субпотока:

```
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; /* аппаратный буфер FIFO полон */
}
```

Если вы знаете заранее, что оборудование примет данные, можно использовать функцию ***snd_rawmidi_transmit***, которая считывает данные и сразу удаляет их из буфера:

```
while (snd_mychip_transmit_possible()) {
    unsigned char data;
    if (snd_rawmidi_transmit(substream, &data, 1) != 1)
        break; /* данных больше нет */
    snd_mychip_transmit(data);
}
```

Если вы знаете заранее, сколько байтов могут быть приняты, в функциях ***snd_rawmidi_transmit**** можно использовать буфер большего размера, чем единица.

Обратный вызов ***trigger*** не должен спать. Если перед передачей буфера субпотока оказалось, что аппаратный FIFO полон, вы должны продолжить передавать данные позже, либо в обработчике прерывания, либо с помощью таймера, если оборудование не имеет прерывания передачи MIDI.

Обратный вызов ***trigger*** вызывается с нулевым параметром **up**, если передача данных должна быть прервана.

Обратный вызов trigger для субпотоков ввода

```
static void snd_xxx_input_trigger(struct snd_rawmidi_substream *substream, int
up);
```

Он вызывается с ненулевым параметром **up** для разрешения получения данных, или с нулевым параметром **up** для запрета приёма данных.

Обратный вызов ***trigger*** не должен спать; фактическое чтение данных из устройства обычно выполняется в обработчике прерывания.

Если приём данных разрешён, ваш обработчик прерывания должен вызывать ***snd_rawmidi_receive*** для всех полученных данных:

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

Обратный вызов drain

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

Используется только с выходными субпотоками. Эта функция должна ждать, пока не будут переданы все данные, считанные из буфера субпотока. Это гарантирует, что устройство может быть закрыто, а драйвер выгружен, без потери данных.

Этот обратный вызов не является обязательным. Если ***drain*** не установлен в структуре **struct snd_rawmidi_ops**, ALSA вместо этого будет просто ждать в течение 50 миллисекунд.

Глава 10. Прочие устройства

FM OPL3

FM OPL3 по-прежнему используется во многих чипах (в основном для обратной совместимости). ALSA также имеет хороший уровень управления OPL3 FM. OPL3 API определён в ***<sound/opl3.h>***.

Регистры FM могут быть доступны непосредственно через direct-FM API (программный интерфейс прямого доступа к FM), определённый в ***<sound/asound_fm.h>***. В родном режиме ALSA регистры FM доступны через расширенное API аппаратно-зависимого прямого доступа к FM, тогда как в режиме совместимости с OSS регистры FM могут быть доступны с помощью OSS совместимого API прямого доступа к FM в устройстве ***/dev/dmfmX***.

Чтобы создать компонент OPL3, у вас есть две функции для вызова. Первая представляет собой конструктор для экземпляра **opl3_t**.

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

Первый аргумент является указателем карты, второй - адресом порта левого канала, а третий - адресом порта правого канала. В большинстве случаев порт правого канала находится по адресу "левый порт + 2".

Четвертый аргумент - тип оборудования.

Если порты левого и правого канала были уже выделены драйвером карты, в пятом аргументе (**integrated**) передаётся ненулевое значение. В противном случае, модуль OPL3 будет сам запрашивать указанные порты.

Если доступ к оборудованию требует специального метода вместо стандартного доступа ввода/вывода, можно создать экземпляр OPL3 отдельно с помощью ***snd_opl3_new()***.

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

Затем укажите ***command***, ***private_data*** и ***private_free*** для функции доступа к закрытым данным, закрытых данных и деструктора. **l_port** и **r_port** устанавливать необязательно. Достаточно правильно установить только ***command***. Вы можете получать данные из поля **opl3->private_data**. (а откуда тогда берётся поле private_data? каков его размер? деструктор по умолчанию тоже есть?)

После создания экземпляра OPL3 через ***snd_opl3_new()***, для инициализации чипа в необходимое состояние вызовите ***snd_opl3_init()***. Обратите внимание, что ***snd_opl3_create()*** всегда называет её внутри себя.

Затем, если экземпляр OPL3 был создан успешно, для этого OPL3 создаём устройство **hwdep**.

```
struct snd_hwdep *opl3hwdep;
```

```
snd_opl3_hwdep_new(opl3, 0, 1, &opl3hwdep);
```

Первым аргументом является созданный экземпляр **opl3_t**, а вторым - порядковый номер, как правило, 0.

Третий аргумент является индексным смещением для клиента-секвенсора, назначенного этому порту OPL3. Если есть MPU401-UART, укажите здесь 1 (UART всегда получает индекс 0).

Устройства, зависящие от оборудования

Некоторым чипам необходим доступ к пространству пользователя для специального управления или загрузки микро-кода. В таком случае можно создать **hwdep** (аппаратно-зависимое) устройство. **hwdep** API определён в ***<sound/hwdep.h>***. Вы можете найти примеры в драйвере ***opl3*** или ***isa/sb/sb16_csp.c***.

Создание экземпляра **hwdep** осуществляется через ***snd_hwdep_new()***.

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

где третий аргумент является порядковым номером.

Затем можно передать любое значения указателя в **private_data**. Если вы указали закрытые данные, вы также должны определить деструктор. Функция деструктора указывается в поле **private_free**.

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

а реализацией деструктора было бы:

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

Для данного экземпляра могут быть определены произвольные файловые операции. Файловые операции определены в таблице **ops**. Например, предположим, что данному чипу необходимо ***ioctl***.

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

И реализуйте функции обратного вызова, как вам нравится.

IEC958 (S/PDIF)

Обычно элементы управления для устройств IEC958 реализуются через интерфейс Control. Для формирования строки названия элементов управления IEC958 существует макрос

SNDRV_CTL_NAME_IEC958(), определённый в ***<include/asound.h>***.

Для статусных битов IEC958 есть несколько стандартных элементов управления. Эти элементы управления используют тип **SNDRV_CTL_ELEM_TYPE_IEC958**, а размер элемента зафиксирован в виде 4-х байтового массива (**value.iec958.status[x]**). Для обратного вызова ***info*** для этого типа не указывается поле **value** (однако, поле **count** должно быть установлено).

"IEC958 Playback Con Mask" используется для возвращения битовой маски статусных битов IEC958 потребительского режима (consumer mode). Аналогично, "IEC958 Playback Pro Mask" возвращает битовую маску для профессионального режима (professional mode). Они доступны только для чтения и определяются как элементы управления MIXER (**iface = SNDRV_CTL_ELEM_IFACE_MIXER**).

Для получения и установки используемых в настоящее время битов по умолчанию IEC958 определён элемент управления "IEC958 Playback Default". Обратите внимание, что он обычно определяется как элемент управления PCM (**iface = SNDRV_CTL_ELEM_IFACE_PCM**), хотя в некоторых местах он определён как элемент управления MIXER.

Кроме того, вы можете определить переключатели для включения/выключения или установки нестандартного режима. Реализация будет зависеть от чипа, но элемент управления должен быть назван "IEC958 xxx", желательно с помощью макроса **SNDRV_CTL_NAME_IEC958()**.

Разные варианты можно найти, например, в ***pci/emu10k1***, ***pci/ice1712*** или ***pci/cmipci.c***.

Глава 11. Управление буфером и памятью

Типы буферов

ALSA предоставляет несколько разных функций для выделения буфера в зависимости от шины и архитектуры. Все они имеют соответствующее API. Выделение физически непрерывных страниц выполняется с помощью функции ***snd_malloc_xxx_pages()***, где **xxx** является типом шины.

Выделение страниц с альтернативой выполняется ***snd_malloc_xxx_pages_fallback()***. Эта функция пытается выделить указанные страницы, но если страницы не доступны, она пытается уменьшить размеры страниц, пока не будет найдено достаточное пространство.

Для освобождения страниц вызовите функцию ***snd_free_xxx_pages()***.

Как правило, в момент загрузки модуля драйверы ALSA пытаются выделить и зарезервировать большое непрерывное физическое пространство для дальнейшего использования. Это называется "предварительное выделение". Как уже писалось, во время создания экземпляра PCM можно вызвать следующую функцию (в случае шины PCI):

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

где **size** - размер в байтах для предварительного выделения, а **max** - максимальный размер, изменяемый с помощью файла ***prealloc*** интерфейса proc. Функция выделения памяти будет пытаться получить как можно большую область в пределах заданного размера.

Второй аргумент (тип) и третий аргумент (указатель устройства) зависят от шины. В случае шины ISA в качестве третьего аргумента передаётся ***snd_dma_isa_data()***, а в качестве типа - **SNDRV_DMA_TYPE_DEV**. Непрерывный буфер не связанный с шиной может предварительно выделяется с помощью типа **SNDRV_DMA_TYPE_CONTINUOUS** и указателем устройства ***snd_dma_continuous_data(GFP_KERNEL)***, где **GFP_KERNEL** является используемым для выделения флагом ядра. Для PCI буферов со сборкой/разборкой используйте **SNDRV_DMA_TYPE_DEV_SG** с ***snd_dma_pci_data(pci)*** (смотрите раздел Буферы, состоящие из несмежных участков 66).

После предварительного выделения буфера в обратном вызове ***hw_params*** можно использовать следующий распределитель памяти:

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

Обратите внимание, чтобы использовать эту функцию, вы должны предварительно выделить память.

Внешние аппаратные буферы

Некоторые чипы имеют свои собственные аппаратные буферы и передача DMA из основной памяти не доступна. В таком случае, вам необходимо либо 1) копировать/помещать аудио-данные непосредственно во внешний аппаратный буфер, либо 2) сделать промежуточный буфер и копировать/помещать данные из него во внешний аппаратный буфер в прерываниях

(или, что предпочтительнее, в тасклетах).

Первый вариант отлично работает, если внешний аппаратный буфер достаточно велик. Этот метод не требует каких-либо дополнительных буферов и, следовательно, является более эффективным. Для передачи данных необходимо определить обратные вызовы ***copy*** и ***silence***. Однако, есть недостаток: он не может быть отображён через ***mmap***. Примерами являются GF1 PCM от GUS или таблица волнового синтеза PCM в emu8000.

Второй случай позволяет отобразить буфер на память, хотя вы должны обрабатывать прерывание или тасклет для передачи данных из промежуточного буфера в аппаратный буфер. Вы можете найти пример в драйвере vxpocket.

Ещё одним случаем является использование чипом для буфера отображённой области памяти PCI, а не основной памяти. В этом случае ***mmap*** доступна только на определённых архитектурах, например, Intel. В режиме без ***mmap*** данные не могут быть переданы, как в обычном способе. Таким образом, необходимо определить обратные вызовы ***copy*** и ***silence*** также, как и в случаях выше. Примеры можно найти в ***rme32.c*** и ***rme96.c***.

Реализация обратных вызовов ***copy*** и ***silence*** зависит от того, поддерживает оборудование данные с чередованием (interleaved) или без чередования (non-interleaved). Обратный вызов copy определяется, как показано ниже, и немного отличается в зависимости от используемого направления, воспроизведение или захват:

```
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);
```

В случае с данных с чередованием, второй аргумент (**channel**, канал) не используется. Третий аргумент (**pos**) указывает текущее положение смещения в кадрах.

Смысл четвёртого аргумента отличается для воспроизведения и захвата. Для воспроизведения он хранит указатель данных источника, а для захвата - это указатель данных назначения.

Последний аргумент является количеством кадров для копирования.

То, что вы должны сделать в этом обратном вызове снова зависит от направления, воспроизведение и захват. В случае воспроизведения, копируется заданное количество данных (**count**) по заданному указателю (**src**) с указанным смещением (**pos**) в аппаратный буфер. Код в стиле ***memcpy*** будет выглядеть так:

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

В случае захвата копируется заданное количество данных (**count**) с указанным смещением (**pos**) в аппаратном буфере по указанному указателю (**dst**).

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

Отметим, что и позиция, и объём данных указаны в кадрах.

В случае данных без чередования реализация будет немного сложнее.

Вы должны проверить аргумент **channel**, и если он -1, скопировать все каналы. В противном случае, вы должны скопировать только указанный канал. Пожалуйста, используйте в качестве примера ***isa/gus/gus_pcm.c***.

Обратный вызов ***silence*** также реализован аналогично.

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

Значения аргументов такие же, как в обратном вызове ***copy***, хотя нет аргумента ***src***/***dst***. В случае данных с чередованием аргумент ***channel*** не имеет смысла, также как в обратном вызове ***copy***.

Ролью обратного вызова ***silence*** является установка определенного количества (**count**) данных тишины по указанному смещению (**pos**) в аппаратном буфере. Предположим, что формат данных знаковый (то есть, данными для тишины является 0), тогда реализация в стиле ***memset*** была бы такой:

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

Аналогично, в случае данных без чередования реализация становится немного сложнее. В качестве примера смотрите ***isa/gus/gus_pcm***.

Буферы, состоящие из несмежных участков

Если ваше оборудование поддерживает таблицы страниц, как emu10k1, или дескрипторы буферов, как via82xx, можно использовать DMA с разборкой/сборкой (scatter-gather, SG). ALSA предоставляет интерфейс для обработки SG-буферов. API предоставляется в ***<sound/pcm.h>***.

Для создания обработчик SG-буфера, вызовите ***snd_pcm_lib_preallocate_pages()*** или ***snd_pcm_lib_preallocate_pages_for_all()*** с **SNDRV_DMA_TYPE_DEV_SG** в конструкторе PCM, как и при другом предварительном выделении памяти для PCI. Вам необходимо передать ***snd_dma_pci_data(pci)***, где **pci** также является указателем **struct pci_dev** чипа. Экземпляр структуры **snd_sg_buf** создаётся как **substream->dma_private**. Вы можете привести указатель следующим образом:

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

Затем, так же как случае обычного буфера PCI вызовите в обратном вызове ***hw_params*** функцию ***snd_pcm_lib_malloc_pages()***. Обработчик SG-буфера выделит несмежные страницы ядра заданного размера и отобразит их на виртуально непрерывную память. Виртуальный указатель доступен в **runtime->dma_area**. Физический адрес (**runtime->dma_addr**) равен нулю, так как буфер является физически несмежным. Таблица физических адресов создана в **sgbuf->table**. Вы можете получить физический адрес для определённого смещения через ***snd_pcm_sgbuf_get_addr()***.

Когда используется SG-обработчик, необходимо в качестве обратного вызова ***page***

установить ***snd_pcm_sgbuf_ops_page***. (Смотрите раздел Обратный вызов page 39.)

Для освобождения данных в обратном вызове ***hw_free*** вызовите ***snd_pcm_lib_free_pages()***, как обычно.

Буферы, выделенные с помощью vmalloc

Возможно использование буфера, выделенного через ***vmalloc***, например, для промежуточного буфера. Так как выделенные страницы не являются смежными, для получения физического адреса для каждого смещения вам необходимо указать обратный вызов ***page***.

Реализация обратного вызова ***page*** была бы такой:

```
#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);
}
```

Глава 12. Интерфейс Proc

ALSA предоставляет простой интерфейс для файловой системы procfs. Файлы proc очень полезны для отладки. Я рекомендую вам создавать proc файлы, если вы пишете драйвер и хотите получать статус работы или распечатку регистров. API находится в ***<sound/info.h>***.

Чтобы создать proc файл, вызовите ***snd_card_proc_new()***.

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

где второй аргумент указывает имя proc файла, который должен быть создан. В приведённом выше примере будет создан файл ***my-file*** внутри каталога карты, например, ***/proc/asound/card0/my-file***.

Как и другие компоненты, запись в proc, созданная через ***snd_card_proc_new()***, будет автоматически регистрироваться и освобождаться в функциях регистрации и освобождения карты.

При успешном создании функция сохраняет новый экземпляр в указателе, указанном в третьем аргументе. Он инициализируется как текстовый proc файл доступный только для чтения. Чтобы использовать данный proc файл как только читаемый текстовый файл, укажите обратный вызов ***read*** с закрытыми данными через ***snd_info_set_text_ops()***.

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

где второй аргумент (**chip**) является закрытыми данными, которые будут использоваться в обратных вызовах. Третий параметр указывает размер буфера чтения, а четвёртый (***my_proc_read***) является функцией обратного вызова, которая определяется как

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

Для вывода строк в обратном вызове ***read*** используйте ***snd_iprintf()***, которая работает так же, как обычная ***printf()***.
Например,

```
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);
}
```

Права доступа к файлам в последствии могут быть изменены. По умолчанию они установлены для всех пользователей как доступные только для чтения. Если вы хотите добавить для пользователя разрешение на запись (по умолчанию root-у), сделайте следующее:

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

и установите размер буфера записи и обратный вызов

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

Для получения строки текста в обратном вызове ***write*** можно использовать ***snd_info_get_line()***, а для получения из неё кусочка строки - ***snd_info_get_str()***. Некоторые примеры можно найти в ***core/oss/mixer_oss.c*** и ***core/oss/pcm_oss.c***.

Для proc файла для необработанных данных укажите атрибуты следующим образом:

```
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;
```

Обратный вызов является гораздо более сложным, чем версия для текстового файла. Для передачи данных необходимо использовать низкоуровневые функции ввода/вывода, такие как ***copy_from/to_user()***.

```
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;
}
```

Глава 13. Управление питанием

Если чип допускает работу с функциями приостановки/возобновления работы, необходимо добавить в драйвер код управления питанием. Дополнительный код для управления питанием должен использовать **ifdef** с **CONFIG_PM**.

Если драйвер ***полностью*** поддерживает приостановку/возобновление, то есть устройство может правильно восстановить своё состояние после вызова приостановки, можно установить в поле PCM **info** флаг **SNDRV_PCM_INFO_RESUME**. Как правило, это возможно, если регистры чипа могут быть безопасно сохранены и восстановлены из ОЗУ. Если флаг установлен, после завершения обратного вызова ***resume*** вызывается обратный вызов ***trigger*** с **SNDRV_PCM_TRIGGER_RESUME**.

Даже если драйвер не поддерживает полностью управление питанием, но частичная приостановка/возобновление всё же возможна, обратные вызовы приостановки/возобновления по-прежнему стоит реализовать. В таком случае приложения сбрасывали бы состояние вызывая ***snd_pcm_prepare()*** и надлежащим образом перезапускали поток. Таким образом, можно в дальнейшем определить обратные вызовы приостановки/возобновления, но не устанавливать в PCM в поле **info** флаг **SNDRV_PCM_INFO_RESUME**.

Обратите внимание, что независимо от флага **SNDRV_PCM_INFO_RESUME**, если вызвана ***snd_pcm_suspend_all***, всегда может быть вызван ***trigger*** с **SUSPEND**. Флаг **RESUME** влияет только на поведение ***snd_pcm_resume()***. (Таким образом, в теории, когда флаг **SNDRV_PCM_INFO_RESUME** не установлен, в обратном вызове ***trigger*** нет необходимости обрабатывать **SNDRV_PCM_TRIGGER_RESUME**. Но лучше сохранить его по соображениям совместимости.)

В более ранней версии драйверов ALSA предоставлялся универсальный уровень управления питанием, но он был удалён. Драйвер должен определить обработку приостановки/возобновления в соответствии с шиной, к которой подключено устройство. В случае драйверов PCI обратные вызовы выглядят следующим образом:

```
#ifdef CONFIG_PM
static int snd_my_suspend(struct pci_dev *pci, pm_message_t state)
{
    .... /* делаем что-нибудь для приостановки */
    return 0;
}
static int snd_my_resume(struct pci_dev *pci)
{
    .... /* делаем что-нибудь для возобновления */
    return 0;
}
#endif
```

Последовательность реальной приостановки работы следующая:

1. Получить данные карты и чипа.
2. Вызвать ***snd_power_change_state()*** с **SNDRV_CTL_POWER_D3hot** для изменения состояния питания.
3. Вызвать ***snd_pcm_suspend_all()***, чтобы приостановить работающие потоки PCM.
4. Если используются кодеки AC97, вызвать для каждого кодека ***snd_ac97_suspend()***.

5. Сохранить значения регистров, если необходимо.
6. Остановить оборудования, если это необходимо.
7. Отключить устройство PCI вызовом ***pci_disable_device()***. Затем, наконец, вызвать ***pci_save_state()***.

Типичный код будет таким:

```
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;
}
```

Последовательность реального возобновления работы следующая:

1. Получить данные карты и чипа.
2. Настроить PCI. Во-первых, вызвать ***pci_restore_state()***. Затем снова включить устройство PCI, вызвав ***pci_enable_device()***. При необходимости также вызвать ***pci_set_master()***.
3. Переинициализировать чип.
4. Восстановить сохранённые регистры, если необходимо.
5. Возобновить работу микшера, вызвав, например, ***snd_ac97_resume()***.
6. Перезапустить оборудование (если это надо).
7. Вызвать ***snd_power_change_state()*** с **SNDRV_CTL_POWER_D0** для уведомления процессов.

Типичный код будет таким:

```
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;
}
```

Как показано выше, лучше сохранять регистры после приостановления операций PCM через ***snd_pcm_suspend_all()*** или ***snd_pcm_suspend()***. Это означает, что при получении снимка состояния регистров потоки PCM уже остановлены. Но помните, что в обратном вызове ***resume*** нет необходимости перезапускать PCM поток. Он будет перезапущен через вызов ***trigger*** с **SNDRV_PCM_TRIGGER_RESUME**, когда это необходимо.

Хорошо, сейчас у нас есть все обратные вызовы. Давайте установим их. При инициализации карты убедитесь, что вы можете получить данные чипа из экземпляра карты, обычно через поле **private_data**, в случае, если вы самостоятельно создали объект чипа.

```
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;
    ....
}
```

Если объект чипа создан с помощью ***snd_card_create()***, он всё равно доступен через поле **private_data**.

```
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;
    ....
}
```

Если необходимо место для сохранения регистров, также выделите буфер для этого здесь, поскольку было бы смертельным, если вы бы не смогли выделить память в фазе приостановки работы. Выделенный буфер должен быть освобождён в соответствующем деструкторе.

И, наконец, устанавливаем функции обратного вызова приостановки/возобновления в **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
};
```

Глава 14. Параметры модуля

Для ALSA есть стандартные опции модуля. Каждый модуль должен иметь, по крайней мере, параметры **index** (порядковый номер), **id** (идентификатор) и **enable** (разрешение работы).

Если модуль поддерживает несколько карт (обычно до **8 = SNDRV_CARDS** карт), они должны быть массивами. Для упрощения программирования начальные значения по умолчанию уже определены как константы:

```
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;
```

Если модуль поддерживает только одну карту, вместо этого они могли бы быть одиночными переменными. В этом случае опция **enable** необходима не всегда, но лучше иметь фиктивную опцию для совместимости.

Параметры модуля должны быть объявлены с помощью стандартных макросов ***module_param_array()***, ***module_param()*** и **MODULE_PARM_DESC()**.

Типичный код будет выглядеть, как показано ниже:

```
#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.");
```

Кроме того, не забудьте определить описание модуля, классов, лицензию и устройства. В частности, последняя версия ***modprobe*** требует определить лицензию модуля как GPL, и так далее, в противном случае система отображается как "испорченная".

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

Глава 15. Как поместить драйвер в дерево ALSA

Общие сведения

До сих пор вы учились писать код драйверов. И вы, возможно, теперь имеете вопрос: как поместить свой драйвер в дерево драйверов ALSA? Здесь (в конце концов :) вкратце описана стандартная процедура.

Предположим, что вы создаёте новый драйвер для PCI карты "xyz". Названием модуля карты было бы snd-xyz. Новый драйвер обычно помещается в дерево драйверов ALSA, в случае карт PCI это каталог ***alsa-driver/pci***. Затем драйвер оценивается, проверяется и испытывается разработчиками и пользователями. Через некоторое время драйвер перейдёт в дерево ядра ALSA (в соответствующий каталог, например, ***alsa-kernel/pci***) и в конечном итоге будет интегрирован в дерево Linux 2.6 (каталогом был бы ***linux/sound/pci***).

В следующих разделах предполагается, что код драйвера должен быть помещён в дерево драйверов ALSA. Рассматриваются два случая: драйвер, состоящий из одного исходного файла, и драйвер, состоящий из нескольких исходных файлов.

Драйверы с исходниками в одном файле

1. Изменяем ***alsa-driver/pci/Makefile***

Предположим, что у вас есть файл ***xyz.c***. Добавляем следующие две строки

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

2. Создаём запись в Kconfig

Добавляем новую запись Kconfig для драйвера xyz.

```
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.
```

строка select SND_PCM указывает, что драйвер xyz поддерживает PCM. В дополнение к **SND_PCM** для команды select поддерживаются следующие компоненты: **SND_RAWMIDI**, **SND_TIMER**, **SND_HWDEP**, **SND_MPU401_UART**, **SND_OPL3_LIB**, **SND_OPL4_LIB**, **SND_VX_LIB**, **SND_AC97_CODEC**. Добавьте команду select для каждого поддерживаемого компонента.

Заметим, что некоторые варианты выбора подразумевают низкоуровневые выборы вариантов. Например, **PCM** включает **TIMER**, **MPU401_UART** включает **RAWMIDI**, **AC97_CODEC** включает **PCM**, а **OPL3_LIB** включает **HWDEP**. Однако, вам не требуется

указывать выбор низкоуровневых вариантов.

Для получения подробной информации о скриптах Kconfig, обратитесь к документации по ***kbuild***.

3. Запускаем скрипт cvscompile для повторной генерации скрипта конфигурации и собираем всё необходимое ещё раз.

Драйверы с исходниками в нескольких файлах

Предположим, что драйвер snd-xyz имеет несколько исходных файлов. Они расположены в новом каталоге, pci/xyz.

1. Добавляем новый каталог (xyz) в ***alsa-driver/pci/Makefile***, как показано ниже

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

2. В каталоге xyz создаём Makefile

Пример 15.1. Пример Makefile для драйвера 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. Создаём запись KConfig

   Эта процедура такая же, как и в предыдущем разделе.

4. Запускаем скрипт cvscompile для повторной генерации скрипта конфигурации и собираем всё необходимое ещё раз.

Глава 16. Полезные функции

snd_printk() и друзья

ALSA обеспечивает версию функции ***printk()***, печатающую подробную информацию. Если в конфигурации ядра установлен **CONFIG_SND_VERBOSE_PRINTK**, эта функция выводит заданное сообщение вместе с именем файла и строкой вызова. Также как в оригинальной ***printk()*** обрабатывается префикс **KERN_XXX**, поэтому рекомендуется добавлять этот префикс, например,

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

Есть также варианты ***printk()*** для отладки. Для общих целей отладки может быть использована ***snd_printd()***. Если установлен **CONFIG_SND_DEBUG**, эта функция компилируется и работает как ***snd_printk()***. Если ALSA собрана без флага отладки, она игнорируется.

snd_printdd() компилируется только тогда, когда установлен **CONFIG_SND_DEBUG_VERBOSE**. Обратите внимание, что **CONFIG_SND_DEBUG_VERBOSE** не установлен по умолчанию, даже если вы конфигурируете драйвер ALSA с опцией **--with-debug=full**. Вместо этого вы должны явно указать опцию **--with-debug=detect**.

snd_BUG()

Это показывает сообщение **"BUG?"** и трассировку стека, также как делает ***snd_BUG_ON***. Он полезен, чтобы показать, что здесь происходит фатальная ошибка.

Если флаг отладки не установлен, этот макрос игнорируется.

snd_BUG_ON()

Макрос **snd_BUG_ON()** аналогичен макросу **WARN_ON()**. Например,

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

или он может быть использован в качестве условия,

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

Макрос принимает условное выражение для оценки. Когда установлен **CONFIG_SND_DEBUG**, выражение оценивается на самом деле. Если оно не равно нулю, он показывает предупреждающее сообщение, такое как **BUG? (xxx)**, следующее обычно за трассировкой стека. Он возвращает оценённое значение. Если **CONFIG_SND_DEBUG** не установлен, этот макрос всегда возвращает ноль.

Глава 17. Благодарности

Я хотел бы поблагодарить Phil Kerr за помощь в улучшении и корректировке этого документа.

Kevin Conder переформатировал оригинальный простой текст в формат DocBook.

Giuliano Pochini исправил опечатки и внёс вклад в коды примеров в разделе аппаратных ограничений.