PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Essential Linux Device Drivers

"Probably the most wide ranging and complete Linux device driver book I've read."

—Alan Cox, Linux Guru and Key Kernel Developer

Sreekrishnan Venkateswaran

Foreword by Arnold Robbins

Перевод

Краткая информация

V*D*V

Перевод глав книги "Essential Linux Device Drivers"

Оглавление

Глава 7. Драйверы Ввода 1

Драйверы входных событий 3
Драйверы устройств ввода 10
Отладка 21
Где искать информацию 22

Глава 12. Драйверы Видео 24

Архитектура отображения 25
Видео подсистема Linux 28
Параметры дисплея 30
API кадрового буфера 32
Драйверы кадрового буфера 35
Консольные драйверы 48
Отладка 54
Где искать информацию 55

Глава 13. Драйверы Звука 58

Звуковая архитектура 59
Звуковая подсистема Linux 61
Пример устройства: MP3 плеер 64
Отладка 78
Где искать информацию 79

Индекс 81

Глава 7. Драйверы Ввода

В этой главе

- Драйверы входных событий 3
- Драйверы устройств ввода 10
- Отладка 21
- Где искать информацию 22

Подсистема ввода ядра была создана с целью объединить рассеянные драйверы, которые обрабатывают различные классы устройств ввода данных, таких как клавиатуры, мыши, трекболы, джойстики, роликовые колёса, сенсорные экраны, акселерометры и планшеты. Подсистема ввода даёт следующие преимущества, представленные в виде таблицы:

- Единообразную обработку функционально похожих устройств ввода, даже если они физически различны. Например, все мыши, такие, как PS/2, USB или Bluetooth, будут обрабатываться одинаково.

- Удобный интерфейс событий для отправки пользовательским приложениям сообщений о вводе. Ваш драйвер не должен создавать и управлять узлами ***/dev*** и связанных с ними методов доступа. Вместо этого он может просто вызвать API для ввода, чтобы отправить движения мыши, нажатия клавиш или сообщения о касании вверх по направлению к пространству пользователя. Такие приложения, как X Windows, хорошо стыкуются с интерфейсами событий, экспортируемыми подсистемой ввода.

- Выделение из входных драйверов общих частей и как результат - абстракция, которая упрощает драйверы и предоставляет взаимодействие. Например, подсистема ввода предлагает коллекцию низкоуровневых драйверов, названных ***serio***, которые обеспечивают доступ к оборудованию ввода, такому, как последовательные порты и контроллеры клавиатуры.

Функционирование подсистемы ввода иллюстрирует Рисунок 7.1. Подсистема состоит из двух классов драйверов, которые работают в тандеме: драйверы ***событий*** и драйверы ***устройств***. Драйверы событий отвечают за взаимодействие с приложениями, в то время как драйверы устройств отвечают за низкоуровневое взаимодействие с устройствами ввода. Генератор событий мыши, ***mousedev***, является примером первого, а драйвер мыши PS/2 является примером последнего. И драйверы событий, и драйверы устройств могут воспользоваться услугами эффективного, свободного от ошибок, годного для многократного использования ядра, которое лежит в основе подсистемы ввода.

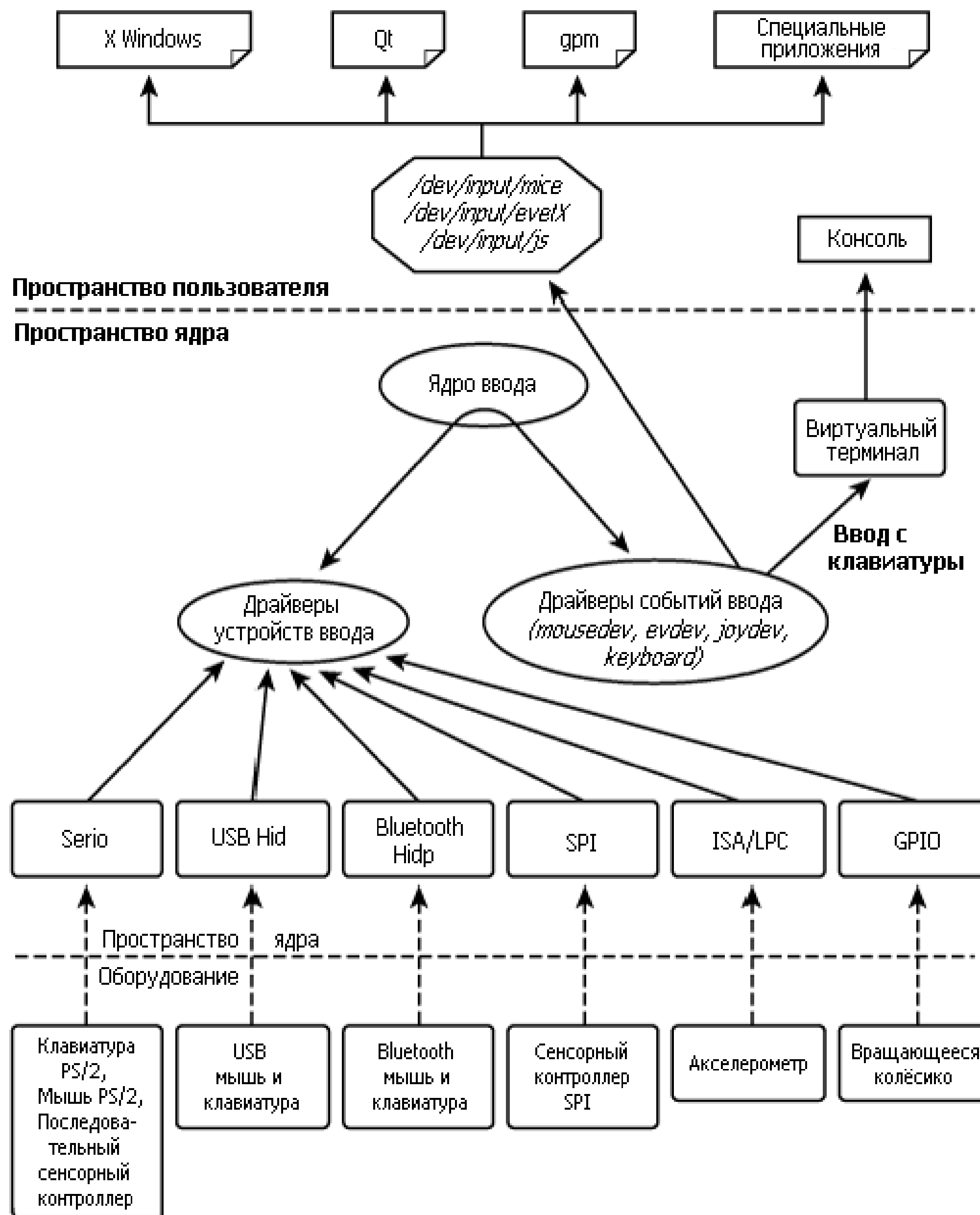

Рисунок 7.1. Подсистема ввода.

Поскольку драйверы событий являются стандартизированными и доступными для всех классов ввода, вам скорее потребуется реализовать драйвер устройства, чем драйвер событий. Для взаимодействия с пользовательскими приложениями ваш драйвер устройства может использовать подходящий существующий драйвер событий в ядре ввода. Обратите внимание, что в настоящей главе для обозначения драйвера устройства ввода используется термин ***драйвер устройства***, а не драйвер входных событий.

Драйверы входных событий

Интерфейс событий, экспортируемый подсистемой ввода, эволюционировал в стандарт, который понимают многие графические оконные системы. Для общения с устройствами ввода драйверы событий предлагают аппаратно-независимую абстракцию, так же как интерфейс кадрового буфера (этот вопрос рассматривается в Главе 12, "Драйверы Видео" 24) представляет собой общий механизм для взаимодействия с устройствами отображения.

Драйверы событий в тандеме с драйверами кадрового буфера изолируют графические пользовательские интерфейсы (GUI) от капризов базового оборудования.

Интерфейс Evdev

Evdev - это универсальный драйвер событий ввода. Каждый пакет события, создаваемый evdev, имеет следующий формат, определённый в ***include/linux/input.h***:

```
struct input_event {
    struct timeval time; /* Метка времени */
    __u16 type;   /* Тип события */
    __u16 code;   /* Код события */
    __s32 value; /* Значение события */
};
```

Чтобы узнать, как использовать evdev, давайте реализуем драйвер устройства ввода для виртуальной мыши.

Пример устройства: Виртуальная мышь

Вот как работает наша виртуальная мышь: приложение (***coord.c***) эмулирует движения мыши и отправляет информацию о координатах в драйвер виртуальной мыши (***vms.c***) через узел в sysfs, ***/sys/devices/platform/vms/coordinates***. Драйвер виртуальной мыши (для краткости, драйвер ***vms***) отправляет эти движения наверх через evdev. Детали показаны на Рисунке 7.2.

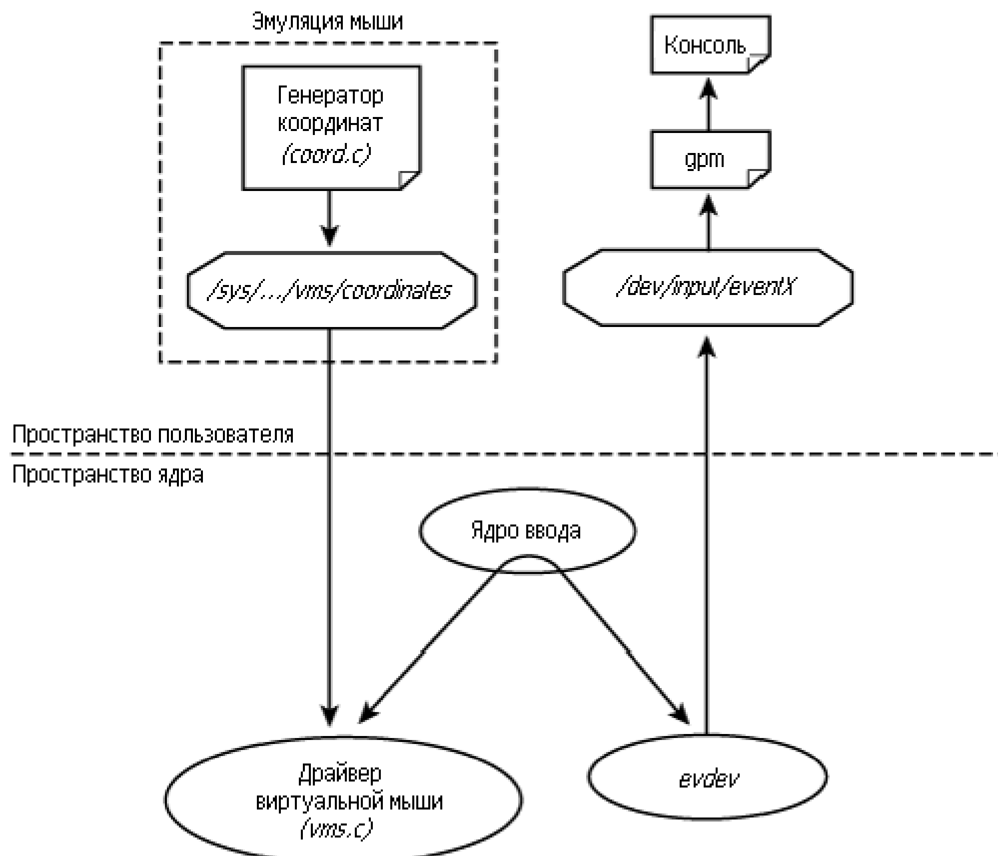

Рисунок 7.2. Драйвер ввода для виртуальной мыши.

Мышь общего назначения (***General-purpose mouse, gpm***) представляет собой сервер, который позволяет использовать мышь в текстовом режиме без помощи со стороны X сервера. GPM понимает события evdev, так что драйвер vms может взаимодействовать с ним непосредственно. После того, как вы соберёте всё вместе, вы сможете увидеть курсор, танцующий на вашем экране, что соответствует движениям виртуальной мыши, передаваемым ***coord.c***.

Распечатка 7.1 содержит программу ***coord.c***, которая постоянно генерирует случайные координаты X и Y. Мышь, в отличие от джойстиков или сенсорных панелей, передаёт относительные координаты, и это то, что делает ***coord.c***. Драйвер vms показан в Распечатке 7.2.

Распечатка 7.1. Приложение, эмулирующее движения мыши (coord.c)

Код:
```
#include <fcntl.h>

int
main(int argc, char *argv[])
{
    int sim_fd;
    int x, y;
    char buffer[10];

    /* Открываем узел с координатами в sysfs */
    sim_fd = open("/sys/devices/platform/vms/coordinates", O_RDWR);
    if (sim_fd < 0) {
        perror("Couldn't open vms coordinate file\n");
        exit(-1);
    }
    while (1) {
        /* Генерируем случайные относительные координаты */
        x = random()%20;
        y = random()%20;
        if (x%2) x = -x; if (y%2) y = -y;

        /* Отправляем съэмулированные координаты драйверу виртуальной мыши */
        sprintf(buffer, "%d %d %d", x, y, 0);
        write(sim_fd, buffer, strlen(buffer));
        fsync(sim_fd);
        sleep(1);
    }

    close(sim_fd);
}
```

Распечатка 7.2. Драйвер ввода для виртуальной мыши (vms.c)

Код:
```
#include <linux/fs.h>
#include <asm/uaccess.h>
#include <linux/pci.h>
```

```
#include <linux/input.h>
#include <linux/platform_device.h>

struct input_dev *vms_input_dev;        /* Представление устройства ввода */
static struct platform_device *vms_dev; /* Структура устройства */

/* Метод для ввода из sysfs съэмулированных координат в драйвер виртуальной
мыши */
static ssize_t
write_vms(struct device *dev,
          struct device_attribute *attr,
          const char *buffer, size_t count)
{
    int x,y;
    sscanf(buffer, "%d%d", &x, &y);

    /* Сообщаем относительные координаты через интерфейс событий */
    input_report_rel(vms_input_dev, REL_X, x);
    input_report_rel(vms_input_dev, REL_Y, y);
    input_sync(vms_input_dev);
    return count;
}

/* Подключаем метод записи sysfs */
DEVICE_ATTR(coordinates, 0644, NULL, write_vms);

/* Дескриптор атрибутов */
static struct attribute *vms_attrs[] = {
    &dev_attr_coordinates.attr,
    NULL
};

/* Группа атрибутов */
static struct attribute_group vms_attr_group = {
    .attrs = vms_attrs,
};

/* Инициализация драйвера */
int __init
vms_init(void)
{

    /* Регистрируем устройство платформы */
    vms_dev = platform_device_register_simple("vms", -1, NULL, 0);
    if (IS_ERR(vms_dev)) {
        printk("vms_init: error\n");
        return PTR_ERR(vms_dev);
    }

    /* Создаём в sysfs узел для чтения съэмулированных координат */
    sysfs_create_group(&vms_dev->dev.kobj, &vms_attr_group);

    /* Создаём структуру данных устройства ввода */
    vms_input_dev = input_allocate_device();
    if (!vms_input_dev) {
```

```
        printk("Bad input_allocate_device()\n");
        return -ENOMEM;
    }

    /* Сообщаем, что эта виртуальная мышь будет генерировать относительные
координаты */
    set_bit(EV_REL, vms_input_dev->evbit);
    set_bit(REL_X, vms_input_dev->relbit);
    set_bit(REL_Y, vms_input_dev->relbit);

    /* Регистрируемся в подсистеме ввода */
    input_register_device(vms_input_dev);

    printk("Virtual Mouse Driver Initialized.\n");
    return 0;
}

/* Выход из драйвера */
void
vms_cleanup(void)
{

    /* Отменяем регистрацию в подсистеме ввода */
    input_unregister_device(vms_input_dev);

    /* Освобождаем узел в sysfs */
    sysfs_remove_group(&vms_dev->dev.kobj, &vms_attr_group);

    /* Отменяем регистрацию драйвера */
    platform_device_unregister(vms_dev);

    return;
}

module_init(vms_init);
module_exit(vms_cleanup);
```

Давайте внимательнее посмотрим на Распечатку 7.2. Во время инициализации драйвер vms регистрирует себя в качестве драйвера устройства ввода. Для этого он сначала выделяет память для структуры **input_dev** с использованием API ядра, **input_allocate_device()**:

```
vms_input_dev = input_allocate_device();
```

Затем он объявляет, что виртуальная мышь генерирует события с относительными координатами:

```
set_bit(EV_REL, vms_input_dev->evbit); /* Типом события является EV_REL */
```

Далее он декларирует коды событий, которые создаёт виртуальная мышь:

```
set_bit(REL_X, vms_input_dev->relbit); /* Относительное движение по 'X' */
set_bit(REL_Y, vms_input_dev->relbit); /* Относительное движение по 'Y' */
```

Если ваша виртуальная мышь также способна генерировать нажатия на кнопки мыши, вы

должны добавить к этому **vms_init()**:

```
set_bit(EV_KEY, vms_input_dev->evbit); /* Типом события является EV_KEY */
set_bit(BTN_0, vms_input_dev->keybit); /* Кодом события является BTN_0 */
```

И, наконец, регистрация:

```
input_register_device(vms_input_dev);
```

write_vms() является методом sysfs **store()**, который связан с ***/sys/devices/platform/vms/coordinates***. Когда ***coord.c*** пишет в этот файл пару X/Y, **write_vms()** выполняет следующие действия:

```
input_report_rel(vms_input_dev, REL_X, x);
input_report_rel(vms_input_dev, REL_Y, y);
input_sync(vms_input_dev);
```

Первый оператор генерирует событие **REL_X**, или относительное движение устройства по оси X. Второй создаёт событие **REL_Y**, или относительное движение по оси Y. **input_sync()** показывает, что это событие является полным, так что подсистема ввода собирает эти два события в один пакет evdev и отправляет его за дверь через ***/dev/input/eventX***, где ***X*** представляет собой номер интерфейса, присвоенный драйверу vms. Приложение, читающее этот файл, получит пакеты событий в описанном ранее формате **input_event**. Чтобы попросить gpm подключиться к этому интерфейсу событий и, соответственно, погонять курсор по экрану, сделайте следующее:

```
bash> gpm -m /dev/input/eventX -t evdev
```

Драйвер контроллера сенсорного экрана ADS7846 и драйвер акселерометра, обсуждаемые позднее соответственно в разделах "Сенсорные контроллеры" 18 и "Акселерометры" 19, также являются пользователями evdev.

Дополнительная информация об интерфейсе событий

Драйвер vms использует общий интерфейс событий evdev, но устройства ввода, такие как клавиатуры, мыши и контроллеры касаний, имеют специальные драйверы событий. Мы рассмотрим их, когда будем обсуждать соответствующие драйверы устройств.

Чтобы написать свой собственный драйвер событий и экспортировать его в пользовательское пространство с помощью ***/dev/input/mydev***, вы должны заполнить структуру, называемую **input_handler**, и зарегистрировать её в ядре ввода следующим образом:

Код:

```
static struct input_handler my_event_handler = {
    .event = mydev_event,    /* Обработка сообщений о событиях, посылаемых
драйверами устройств ввода,
                               которые пользуются услугами этого драйвера
событий */
    .fops = &mydev_fops,     /* Методы для управления /dev/input/mydev */
    .minor = MYDEV_MINOR_BASE, /* Младший номер /dev/input/mydev */
    .name = "mydev",         /* Имя драйвера событий */
```

```
    .id_table = mydev_ids,  /* Этот драйвер событий может обрабатывать запросы
с такими ID-ами */
    .connect = mydev_connect, /* Вызывается, если есть совпадение ID */
    .disconnect = mydev_disconnect, /* Вызывается для отмены регистрации
драйвера */
};

/* Инициализация драйвера */
static int __init
mydev_init(void)
{
    /* ... */

    input_register_handler(&my_event_handler);

    /* ... */
    return 0;
}
```

Для полноценного примера посмотрите на реализацию mousedev (***drivers/input/mousedev.c***)
.

Драйверы устройств ввода

Давайте обратим наше внимание на драйверы для распространённых устройств ввода, таких как клавиатуры, мыши и сенсорные экраны. Но сначала давайте кратко рассмотрим готовый сервис для доступа к оборудованию, доступный для драйверов ввода.

Serio

Уровень ***serio*** предлагает библиотечные подпрограммы для доступа к устаревшему оборудованию ввода, такому как i8042-совместимые контроллеры клавиатуры и последовательный порт. Клавиатуры PS/2 и мыши подключаются к первому, а сенсорные контроллеры с последовательным интерфейсом подключаются к последнему. Для взаимодействия с оборудованием, обслуживаемым serio, например, для передачи команды для PS/2 мыши, предписанные процедуры обратного вызова serio регистрируются с помощью **serio_register_driver()**.

Чтобы добавить новый драйвер как часть serio, с помощью **serio_register_port()** регистрируются точки входа **open()**/**close()**/**start()**/**stop()**/**write()**. Для примера посмотрите ***drivers/input/serio/serport.c***.

Как можно увидеть на Рисунке 7.1, serio - это только один из маршрутов доступа к низкоуровневому оборудованию. Некоторые драйверы устройств ввода вместо него полагаются на низкоуровневую поддержку от шинных уровней, таких как USB или SPI.

Клавиатуры

Клавиатуры бывают на любой вкус - устаревшие PS/2, USB, Bluetooth, ИК, и так далее. Каждый тип имеет специальный драйвер устройства ввода, но все используют один и тот же драйвер событий клавиатуры, обеспечивая тем самым единый интерфейс для пользователей. Драйвер событий клавиатуры, однако, имеет отличительную особенность по сравнению с другими драйверами событий: он передаёт данные другой подсистеме ядра (уровню tty), а не в пользовательское пространстве с помощью узлов ***/dev***.

Клавиатуры ПК

Клавиатуры ПК (также называемые клавиатурами PS/2 или AT клавиатурами) взаимодействует с процессором через i8042-совместимый контроллер клавиатуры. ПК обычно имеют специальный контроллер клавиатуры, но на ноутбуках взаимодействие с клавиатурой является одной из обязанностей встроенного контроллера общего назначения (смотрите раздел "Встроенные контроллеры" в Главе 20, "Дополнительные устройства и драйверы"). Когда вы нажимаете клавишу на клавиатуре компьютера, это происходит по такому пути:

1. Контроллер клавиатуры (или встроенный контроллер) сканирует и декодирует клавиатурную матрицу и заботится о нюансах, таких как устранение дребезга контактов.

2. Клавиатурный драйвер устройства с помощью serio для каждого нажатия и отпускания клавиши читает с контроллера клавиатуры сырые ***коды сканирования***. Разницей между нажатием и отпусканием является самый старший бит, который для последнего случая установлен. Например, нажатие на кнопку "a" даёт пару скан-кодов, **0x1e** и **0x9e**. Специальные кнопки экранируются с помощью **0xE0**, так что нажатие кнопки со стрелкой

вправо производит последовательность (**0xE0 0x4D 0xE0 0xCD**). Для наблюдения выходящих из контроллера скан-кодов вы можете использовать утилиту **showkey** (после символа → идут пояснения):

```
bash> showkey -s
кл-ра была в режиме UNICODE
[ если вы пробуете это под X, это может не работать, так как
X сервер также читает /dev/console ]

нажмите любую кнопку (программа завершится спустя 10с после
последнего нажатия на кнопку)...
...
0x1e 0x9e → Нажатие кнопки "a"
```

3. Клавиатурный драйвер устройства преобразует полученные скан-коды в коды клавиш, основываясь на режиме ввода. Чтобы увидеть код клавиши, соответствующий кнопке "a":

```
bash> showkey
...
код кнопки 30 нажатие → Нажатие на кнопку "a"
код кнопки 30 отпускание → Отпускание кнопки "a"
```

   Чтобы сообщить эти коды клавиш дальше вверх, драйвер генерирует событие ввода, которое передаёт управление драйверу событий клавиатуры.

4. Драйвер событий клавиатуры берёт на себя работу по преобразованию кода клавиши в зависимости от загруженной карты кодов клавиш. (Смотрите страницы справки ***loadkeys*** и map-файлы в ***/lib/kbd/keymaps***.) Он проверяет, является ли преобразованный код клавиш такими действиями, как переключение виртуальной консоли или перезагрузка системы. Чтобы вместо перезагрузки системы в ответ на нажатие Ctrl+Alt+Del зажглись светодиоды **CAPSLOCK** и **NUMLOCK**, добавьте в обработчик Ctrl+Alt+Del драйвера событий клавиатуры, ***drivers/char/keyboard.c***, следующее:

```
static void fn_boot_it(struct vc_data *vc, struct pt_regs *regs)
{
+    set_vc_kbd_led(kbd, VC_CAPSLOCK);
+    set_vc_kbd_led(kbd, VC_NUMLOCK);
-    ctrl_alt_del();
}
```

5. Для обычных клавиш преобразованный код нажатия отправляется ассоциированному виртуальному терминалу и дисциплине линии **N_TTY**. (Мы обсуждали виртуальные терминалы и дисциплины линий в Главе 6, "Драйверы последовательных портов.") ***drivers/char/keyboard.c*** делает это следующим образом:

```
/* Добавляем код клавиши в переключаемый буфер */
tty_insert_flip_char(tty, keycode, 0);
/* Планируем */
con_schedule_flip(tty);
```

Дисциплина линии **N_TTY** обрабатывает ввод таким образом, что полученные с помощью клавиатуры данные отображаются на виртуальной консоли и позволяет приложениям пользовательского пространства читать символы из узла ***/dev/ttyX***, подключённого к

виртуальному терминалу.

На Рисунке 7.3 показано движение данных от момента нажатия клавиши на клавиатуре, до момента его появления на виртуальной консоли. Левая половина рисунка является зависимой от оборудования, а правая половина носит общий характер. В соответствии с целью разработки подсистемы ввода, нижележащий аппаратный интерфейс является прозрачным для драйвера событий клавиатуры и уровня tty. Таким образом, ядро ввода и чётко определённые интерфейсы событий ограждают пользователей ввода от нюансов оборудования.

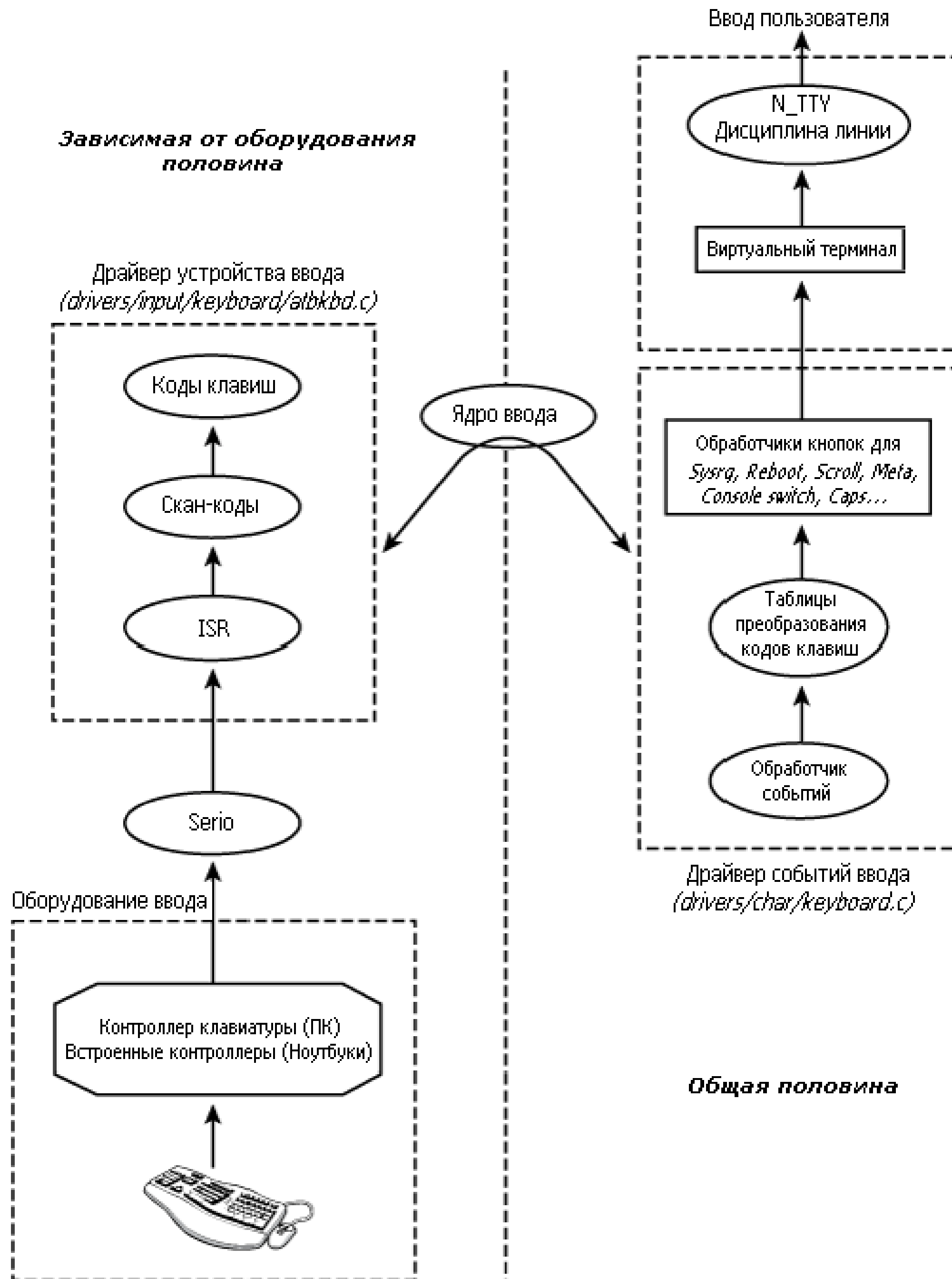

Рисунок 7.3. Поток данных от PS/2-совместимой клавиатуры.

USB и Bluetooth клавиатуры

Спецификациями USB, связанными с устройствами взаимодействия с человеком (HID), предусмотрен протокол, по которому для взаимодействия используются USB клавиатуры, мыши, наборы кнопок и другие периферийные устройства ввода. На Linux это осуществляется через клиентский драйвер USB ***usbhid***, который отвечает за класс USB HID (**0x03**). Usbhid регистрирует себя в качестве драйвера устройства ввода. Он соответствует API ввода и сообщает о соответствующих событиях ввода подключенных HID.

Для того, чтобы понять путь кода для USB клавиатуры, вернёмся к Рисунку 7.3 и изменим аппаратно-зависимую левую половину. Заменим контроллер клавиатуры в квадратике "Оборудование ввода" на контроллер USB, serio на уровень ядра USB и квадратик "Драйвер устройства ввода" на драйвер usbhid.

Для Bluetooth клавиатуры заменим на Рисунке 7.3 контроллер клавиатуры на набор микросхем Bluetooth, serio на уровень ядра Bluetooth и квадратик "Драйвер устройства ввода" на драйвер Bluetooth hidp.

USB и Bluetooth подробно рассматриваются в Главе 11, "Универсальная последовательная шина" и в Главе 16, "Linux без проводов", соответственно.

Мыши

Мыши, как и клавиатуры, бывают с разными возможностями и имеют различные варианты взаимодействия. Давайте посмотрим на обычно использующиеся.

PS/2 мышь

Мыши генерируют относительные передвижения по осям X и Y. Кроме того, они имеют одну или несколько кнопок. Некоторые из них также имеют колёсико прокрутки. Драйвер устройства ввода для устаревшей PS/2-совместимой мыши для взаимодействия с контроллером основывается на уровне serio. Драйвер событий ввода для мышей, называемый **mousedev**, сообщает события мыши пользовательским приложениям с помощью ***/dev/input/mice***.

Пример драйвера: Мышь-колёсико

Чтобы получить настоящий драйвер устройства мыши, давайте преобразуем вращающееся колёсико, рассматриваемое в Главе 4, "Создание основы", в вариант обычной PS/2 мыши. "Мышь-колёсико" создаёт одномерное движение по оси Y. Повороты колёсика по часовой стрелке и против часовой стрелки создают положительные и отрицательные относительные Y координаты соответственно (как колесо прокрутки на мыши), а нажатие на колёсико приводит к событию нажатия на левую кнопку мыши. Мышь-колёсико идеально подходит для навигации по меню в таких устройствах, как смартфоны, КПК и музыкальные плееры.

Драйвер устройства мыши-колёсика, реализованный в Распечатке 7.3, работает с оконными системами, такими как X Windows. Чтобы увидеть, как драйвер заявляет о своих похожих на мышь возможностях, посмотрите на **roller_mouse_init()**. В отличие от драйвера вращающегося колёсика в Распечатке 4.1 Главы 4, драйверу мыши-колёсика не нужны методы **read()** или **poll()**, так как о событиях сообщается с использованием API ввода.

Обработчик прерывания от колёсика **roller_isr()** также соответственно изменяется. Убираем служебные действия, выполняемые в обработчике прерывания, использующие очередь ожидания, спин-блокировку и процедуру **store_movement()** для поддержки **read()** и **poll()**.

В Распечатке 7.3, + и - в начале строк обозначают отличия от драйвера вращающего колёсика, реализованного в Распечатке 4.1 Главы 4.

Распечатка 7.3. Драйвер мыши-колёсика

Код:

```
+ #include <linux/input.h>
+ #include <linux/interrupt.h>
+ /* Device structure */
+ struct {
+     /* ... */
+     struct input_dev dev;
+ } roller_mouse;
+ static int __init
+ roller_mouse_init(void)
+ {
+     /* Выделяем память для структуры устройства ввода */
+     roller_mouse->dev = input_allocate_device();
+
+     /* Можем генерировать щелчок и относительное передвижение */
+     roller_mouse->dev->evbit[0] = BIT(EV_KEY) | BIT(EV_REL);
+     /* Можеи двигаться только по оси Y */
+     roller_mouse->dev->relbit[0] = BIT(REL_Y);
+
+     /* Клик должен восприниматься как щелчок левой кнопки мыши */
+     roller_mouse->dev->keybit[LONG(BTN_MOUSE)] = BIT(BTN_LEFT);
+     roller_mouse->dev->name = "roll";
+
+     /* Для записи в /sys/class/input/inputX/id/ */
+     roller_mouse->dev->id.bustype = ROLLER_BUS;
+     roller_mouse->dev->id.vendor = ROLLER_VENDOR;
+     roller_mouse->dev->id.product = ROLLER_PROD;
+     roller_mouse->dev->id.version = ROLLER_VER;
+     /* Регистрируемся в подсистеме ввода */
+     input_register_device(roller_mouse->dev);
+}

/* Глобальные переменные */
- spinlock_t roller_lock = SPIN_LOCK_UNLOCKED;
- static DECLARE_WAIT_QUEUE_HEAD(roller_poll);

/* Обработчик прерывания колёсика */
static irqreturn_t
roller_interrupt(int irq, void *dev_id)
{
    int i, PA_t, PA_delta_t, movement = 0;

    /* Получить входные сигналы с битов 0, 1 и 2
       порта D, как показано на Рисунке 7.1 */
```

```
    PA_t = PA_delta_t = PORTD & 0x07;

    /* Ждём, пока состояние порта меняется.
      (Добавляем небольшое время ожидания в цикле) */
    for (i=0; (PA_t==PA_delta_t); i++){
        PA_delta_t = PORTD & 0x07;
    }

    movement = determine_movement(PA_t, PA_delta_t);
-   spin_lock(&roller_lock);
-
-   /* Сохраняем движение колёсика в буфере для
-      дальнейшего доступа через входные точки read()/poll() */
-   store_movements(movement);
-
-   spin_unlock(&roller_lock);
-
-   /* Будим входную точку опроса, которая должна бы
-      спать, ожидая вращение колёсика */
-   wake_up_interruptible(&roller_poll);
-
+   if (movement == CLOCKWISE) {
+       input_report_rel(roller_mouse->dev, REL_Y, 1);
+   } else if (movement == ANTICLOCKWISE) {
+       input_report_rel(roller_mouse->dev, REL_Y, -1);
+   } else if (movement == KEYPRESSED) {
+       input_report_key(roller_mouse->dev, BTN_LEFT, 1);
+   }
+   input_sync(roller_mouse->dev);
    return IRQ_HANDLED;
}
```

Устройства позиционирования

Устройство позиционирования (Trackpoint) является указывающим устройством, поставляемым на некоторых ноутбуках интегрированным с клавиатурой типа PS/2. Это устройство включает в себя джойстик, расположенный между клавишами, и кнопки мыши, расположенные под кнопкой пробела. Оно в основном функционирует как мышь, так что вы можете работать с помощью драйвера мыши PS/2.

В отличие от обычной мыши, устройство позиционирования предлагает большее управление движением. Вы можете скомандовать контроллеру устройства изменить свойства, такие как чувствительность и инерция. Для создания и управления связанными узлами sysfs ядро имеет специальный драйвер, ***drivers/input/mouse/trackpoint.c***. Для полного набора конфигураций параметров устройства позиционирования смотрите в ***/sys/devices/platform/i8042/serioX/serioY/***.

Сенсорные панели

Сенсорная панель является мыше-подобным указывающим устройством, обычно встречающимся на ноутбуках. В отличие от обычной мыши, сенсорная панель не имеет движущихся частей. Она может создавать совместимые с мышью относительные

координаты, но обычно используется операционными системами в более мощном режиме, который даёт абсолютные координаты. Протокол связи, используемый в абсолютном режиме, похож на протокол мыши PS/2, но не совместим с ним.

Основной драйвер PS/2 мыши способен поддерживать устройства, которые отвечают разным вариантам простого протокола PS/2 мыши. Можно добавить в базовый драйвер поддержку нового протокола мыши, поставляя драйвер протокола через структуру **psmouse**. Например, если ваш ноутбук использует сенсорную панель Synaptics в абсолютном режиме, базовый драйвер PS/2 мыши для интерпретации потока данных пользуется услугами драйвера протокола Synaptics. Для полного понимания того, как протокол Synaptics работает в тандеме с базовым драйвером PS/2, посмотрим на следующие четыре куска кода, собранные в Распечатке 7.4:

- Драйвер PS/2 мыши, ***drivers/input/mouse/psmouse-base.c***, создаёт экземпляр структуры **psmouse_protocol** с информацией о поддерживаемых протоколах мыши (в том числе о протоколе сенсорной панели Synaptics).

- Структура **psmouse**, определённая в ***drivers/input/mouse/psmouse.h***, связывает вместе различные протоколы PS/2. **synaptics_init()** заполняет структуру **psmouse** адресами соответствующих функций протокола.

- Функция обработчика протокола **synaptics_process_byte()**, установленная в **synaptics_init()**, вызывается из контекста прерывания, когда serio чувствует движение мыши. Если открыть **synaptics_process_byte()**, то можно увидеть, как движения по сенсорной панели сообщаются пользовательским приложениям через mousedev.

Распечатка 7.4. Драйвер протокола PS/2 мыши для сенсорной панели Synaptics

Код:
drivers/input/mouse/psmouse-base.c:

```
/* List of supported PS/2 mouse protocols */
static struct psmouse_protocol psmouse_protocols[] = {
{
    .type = PSMOUSE_PS2,        /* обычный обработчик PS/2 */
    .name = "PS/2",
    .alias = "bare",
    .maxproto = 1,
    .detect = ps2bare_detect,
},
/* ... */
{
    .type = PSMOUSE_SYNAPTICS,  /* Протокол сенсорной панели Synaptics */
    .name = "SynPS/2",
    .alias = "synaptics",
    .detect = synaptics_detect, /* Протокол обнаружен? */
    .init = synaptics_init,     /* Инициализация обработчика протокола */
},
/* ... */
}
```

drivers/input/mouse/psmouse.h:

```
/* Структура, которая объединяет вместе разные протоколы мыши */
```

```
struct psmouse {
    struct input_dev *dev; /* Устройство ввода */
    /* ... */

    /* Методы протокола */
    psmouse_ret_t (*protocol_handler)
            (struct psmouse *psmouse, struct pt_regs *regs);
    void (*set_rate)(struct psmouse *psmouse, unsigned int rate);
    void (*set_resolution)
            (struct psmouse *psmouse, unsigned int resolution);
    int (*reconnect)(struct psmouse *psmouse);
    void (*disconnect)(struct psmouse *psmouse);
    /* ... */
};
```

drivers/input/mouse/synaptics.c:

```
/* Метод init() протокола Synaptics */
int synaptics_init(struct psmouse *psmouse)
{
    struct synaptics_data *priv;
    psmouse->private = priv = kmalloc(sizeof(struct synaptics_data),
            GFP_KERNEL);
    /* ... */

    /* Это вызывается в контексте прерывания, когда ощущается движение мыши */
    psmouse->protocol_handler = synaptics_process_byte;

    /* Другие методы протокола */
    psmouse->set_rate = synaptics_set_rate;
    psmouse->disconnect = synaptics_disconnect;
    psmouse->reconnect = synaptics_reconnect;

    /* ... */
}
```

drivers/input/mouse/synaptics.c:

```
/* Если открыть synaptics_process_byte() и посмотреть в
   synaptics_process_packet(), можно увидеть сообщения ввода,
   сообщаемые пользовательским приложениям через mousedev */
static void synaptics_process_packet(struct psmouse *psmouse)
{
    /* ... */
    if (hw.z > 0) {
        /* Абсолютная X координата */
        input_report_abs(dev, ABS_X, hw.x);
        /* Абсолютная Y координата */
        input_report_abs(dev, ABS_Y,
        YMAX_NOMINAL + YMIN_NOMINAL - hw.y);
    }
    /* Абсолютная Z координата */
    input_report_abs(dev, ABS_PRESSURE, hw.z);
    /* ... */
    /* Левая кнопка TouchPad */
    input_report_key(dev, BTN_LEFT, hw.left);
    /* Правая кнопка TouchPad */
```

```
    input_report_key(dev, BTN_RIGHT, hw.right);
    /* ... */
}
```

USB и Bluetooth мыши

USB мышь обрабатываются тем же драйвером ввода (***usbhid***), который поддерживает USB клавиатуры. Аналогично, драйвер ***hidp***, который реализует поддержку Bluetooth клавиатур, также заботится о Bluetooth мышах. Как и следовало ожидать, драйверы USB и Bluetooth мыши отправляют данные устройства через mousedev.

Сенсорные контроллеры

В Главе 6 мы реализовали драйвер устройства для последовательного сенсорного контроллера в виде дисциплины линии, называемой **N_TCH**. Подсистема ввода предлагает лучший и более простой способ для реализации такого драйвера. Переделаем конечный автомат в **N_TCH** в драйвер устройства ввода с помощью следующих изменений:

1. Для доступа к устройствам, подключенным к последовательному порту, serio предоставляет дисциплину линии, называемую ***serport***. Воспользуемся услугами serport для общения с сенсорным контроллером.

2. Вместо передачи координатной информации на уровень tty, сгенерируем отчёты о вводе с помощью evdev, как это сделано для виртуальной мыши в Распечатке 7.2.

При этом сенсорный экран доступен для пользовательского пространства через ***/dev/input/eventX***. Фактическая реализация драйвера остаётся в качестве упражнения.

Примером сенсорного контроллера, который не подключается через последовательный порт, является микросхема Analog Devices ADS7846, который взаимодействует через ***последовательный интерфейс периферийных устройств*** (Serial Peripheral Interface, ***SPI***). Драйвер для этого устройства пользуется услугами ядра SPI, а не serio. SPI рассматривается в разделе "Шина последовательного интерфейса периферийных устройств" в Главе 8, "Протокол связи между микросхемами". Как и большинство драйверов сенсорных устройств, драйвер ADS7846 использует для отправки информации пользовательским приложениям интерфейс evdev.

Некоторые сенсорные контроллеры подключаются через USB. Примером может служить сенсорный USB контроллер 3M, поддерживаемый ***drivers/input/touchscreen/usbtouchscreen.c***.

Многие КПК имеют 4-проводные резистивные сенсорные панели, наложенные на их ЖК дисплеи. X и Y пластины панели (два провода для каждой оси) подключаются к аналого-цифровому преобразователю (АЦП), который обеспечивает цифровое считывание аналоговой разности потенциалов, возникающей от прикасания к экрану. Драйвер ввода забирает координаты от АЦП и отправляет их в пользовательское пространство.

Различные экземпляры одной сенсорной панели могут давать несколько различные

диапазоны координат (максимальные значения в направлениях X и Y) в связи с нюансами производственных процессов. Чтобы изолировать приложения от такого изменения, сенсорные экраны перед использованием ***калибруются***. Калибровка, как правило, выполняется через GUI путём показа крестиков на границах экрана и других точках, с просьбой к пользователю прикоснуться к этим точкам. Сгенерированные координаты программируются в сенсорный контроллер с помощью соответствующих команд, если он поддерживает самокалибровку, или используются для масштабирования потока координат в программном обеспечение в противном случае.

Подсистема ввода также содержит драйвер событий, называемый ***tsdev***, который генерирует координатную информацию в соответствии с протоколом сенсорного экрана Compaq. Если ваша система сообщает о событиях прикосновений через tsdev, приложения, которые понимают этот протокол, могут извлекать данные сенсорного ввод из ***/dev/input/tsX***. Однако, этот драйвер запланирован на удаление из основной ветки ядра в пользу библиотеки пользовательского пространства ***tslib***. То, что уйдёт из дерева исходных кодов ядра, перечисляет ***Documentation/feature-removalschedule.txt***.

Акселерометры

Акселерометр измеряет ускорение. Некоторые ноутбуки IBM/Lenovo имеют акселерометр, который определяет внезапное движение. Сгенерированная информация используется для защиты от повреждений жёсткого диска с использованием механизма, называемого ***Активная система защиты жёсткого диска*** (Hard Drive Active Protection System, HDAPS), аналогично тому, как автомобильные подушки безопасности защищают от травм пассажира. Драйвер HDAPS реализован как драйвер платформы, который регистрируется в подсистеме ввода. Для передачи X и Y компонентов обнаруженного ускорения он использует evdev. Для выявления таких условий, как удар и вибрация, и выполнения защитных действий, таких как парковка головки жёсткого диска, приложения могут читать события ускорений через ***/dev/input/eventX***. Следующая команда выплёскивает вывод, если сдвинуть ноутбук (предположим, что для HDASP определён event3):

```
bash> od -x /dev/input/event3
0000000 a94d 4599 1f19 0007 0003 0000 ffed ffff
...
```

Акселерометр также предоставляет такую информацию, как температура, активность клавиатуры и мыши, всё, что можно извлечь с помощью файлов в ***/sys/devices/platform/hdaps/***. Из-за этого драйвер HDAPS является частью подсистемы ядра аппаратного мониторинга (***hwmon***). Мы поговорим о мониторинге оборудования в разделе "Мониторинг оборудования с помощью LM-Sensors" следующей главы.

События вывода

Некоторые драйверы устройств ввода также обрабатывают события вывода. Например, драйвер клавиатуры может зажечь индикатор **CAPSLOCK**, а драйвер динамика ПК может издать гудок. Давайте последнее рассмотрим поближе. Во время инициализации драйвер динамика заявляет о своей возможности вывода путём установки соответствующих ***evbits*** и регистрации процедуры обратного вызова для обработки события вывода:

Код:
drivers/input/misc/pcspkr.c:

```
static int __devinit pcspkr_probe(struct platform_device *dev)
{
    /* ... */

    /* Биты возможностей */
    pcspkr_dev->evbit[0] = BIT(EV_SND);
    pcspkr_dev->sndbit[0] = BIT(SND_BELL) | BIT(SND_TONE);

    /* Процедура обратного вызова */
    pcspkr_dev->event = pcspkr_event;

    err = input_register_device(pcspkr_dev);
    /* ... */
}
/* Процедура обратного вызова */
static int pcspkr_event(struct input_dev *dev, unsigned int type,
                        unsigned int code, int value)
{
    /* ... */

    /* Программирование ввода/вывода для издавания гудка */

    outb_p(inb_p(0x61) | 3, 0x61);
    /* дать команду для счётчика 2, записать 2 байта */
    outb_p(0xB6, 0x43);
    /* выбрать требуемую частоту */
    outb_p(count & 0xff, 0x42);
    outb((count >> 8) & 0xff, 0x42);

    /* ... */
}
```

Чтобы издать звук, драйвер событий клавиатуры генерирует звуковое событие (**EV_SND**) следующим образом:

```
input_event(handle->dev, EV_SND,    /* Тип */
                         SND_TONE,  /* Код */
                         hz         /* Значение */);
```

Это вызывает выполнение процедуры обратного вызова, **pcspkr_event()**, и вы слышите звуковой сигнал.

Отладка

Если вы разрабатываете драйвер ввода, в качестве помощи при отладке можно использовать модуль ***evbug***. Он распечатывает набор (***тип***, ***код***, ***значение***) (смотрите структуру **input_event**, определённую ранее), соответствующий событиям, порождённым подсистемой ввода. На Рисунке 7.4 приведены данные, перехваченные evbug при работе с некоторыми устройствами ввода:

Рисунок 7.4. Вывод из Evbug.

Код:

```
/* Движение по сенсорной панели */
evbug.c Event. Dev: isa0060/serio1/input0: Type: 3, Code: 28, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 1, Code: 325, Value: 0
evbug.c Event. Dev: isa0060/serio1/input0: Type: 0, Code: 0, Value: 0

/* Передвижение устройства позиционирования */
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 0, Value: -1
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 2, Code: 1, Value: -2
evbug.c Event. Dev: synaptics-pt/serio0/input0: Type: 0, Code: 0, Value: 0

/* Движение USB мыши */
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 1, Value: -1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 2, Code: 0, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0

/* Нажатие кнопки 'a' на PS/2 клавиатуре */
evbug.c Event. Dev: isa0060/serio0/input0: Type: 4, Code: 4, Value: 30
evbug.c Event. Dev: isa0060/serio0/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: isa0060/serio0/input0: Type: 0, Code: 0, Value: 0

/* Нажатие кнопки 'a' на USB клавиатуре */
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 1, Code: 30, Value: 1
evbug.c Event. Dev: usb-0000:00:1d.1-1/input0: Type: 0, Code: 0, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 1, Code: 30, Value: 0
evbug.c Event. Dev: usb-0000:00:1d.1-2/input0: Type: 0, Code: 0, Value: 0
```

Чтобы разобраться в дампе на Рисунке 7.4, вспомните, что сенсорные панели генерируют абсолютные координаты (**EV_ABS**) или событие типа **0x03**, устройства позиционирования дают относительные координаты (**EV_REL**) или события типа **0x02**, и клавиатуры создают события кнопок (**EV_KEY**) или события типа **0x01**. Тип события **0x0** соответствует вызову **input_sync()**, который выполняет следующие действия:

```
input_event(dev, EV_SYN, SYN_REPORT, 0);
```

Это приводит к набору (***тип***, ***код***, ***значение***) (**0x0**, **0x0**, **0x0**) и завершает каждое событие ввода.

Где искать информацию

Большинство драйверов событий ввода находятся в каталоге ***drivers/input/***. Однако, драйвер событий клавиатуры находится в ***drivers/char/keyboard.c***, потому что он связан с виртуальными терминалами, а не узлами устройств в ***/dev/input/***.

Вы можете найти драйверы устройств ввода в нескольких местах. Драйверы для старых клавиатур, мышей и джойстиков находятся в отдельных подкаталогах в ***drivers/input/***. Драйверы ввода Bluetooth находятся в ***net/bluetooth/hidp/***. Вы также можете найти драйверы ввода в таких местах, как ***drivers/hwmon/*** и ***drivers/media/video/***. Типы событий, коды и значения определены в ***include/linux/input.h***.

Подсистема serio находится в ***drivers/input/serio/***. Исходником дисциплины линии serport является ***drivers/input/serio/serport.c***. Более подробная информация о различных интерфейсах ввода содержится в ***Documentation/input/***.

Таблица 7.122 суммирует основные структуры данных, используемые в этой главе, и их расположение внутри дерева исходных текстов.
Таблица 7.222 перечисляет основные программные интерфейсы ядра, которые вы использовали в этой главе с указанием места их определения.

Таблица 7.1. Список структур данных

Структура данных	Местоположение	Описание
input_event	***include/linux/input.h***	Каждый пакет события, создаваемый evdev, имеет этот формат.
input_dev	***include/linux/input.h***	Представление устройства ввода.
input_handler	***include/linux/serial_core.h***	Содержит точки входа, поддерживаемые драйвером событий.
psmouse_protocol	***drivers/input/mouse/psmouse-base.c***	Информация о драйвере протокола, поддерживающего PS/2 мышь.
psmouse	***drivers/input/mouse/psmouse.h***	Методы, поддерживаемые драйвером PS/2 мыши.

Таблица 7.2. Список программных интерфейсов ядра

Интерфейс ядра	Местоположение	Описание
input_register_device()	***drivers/input/input.c***	Регистрирует устройство в ядре ввода
input_unregister_device()	***drivers/input/input.c***	Удаляет устройство из ядра ввода
input_report_rel()	***include/linux/input.h***	Генерирует относительное

		перемещение в заданном направлении
input_report_abs()	***include/linux/input.h***	Генерирует абсолютное перемещение в заданном направлении
input_report_key()	***include/linux/input.h***	Генерирует нажатие на клавишу или кнопку
input_sync()	***include/linux/input.h***	Показывает, что подсистема ввода может собрать предыдущие сгенерированные пакеты в пакет evdev и отправить их в пространство пользователя через ***/dev/input/inputX***
input_register_handler()	***drivers/input/input.c***	Регистрирует специальный драйвер событий
sysfs_create_group()	***fs/sysfs/group.c***	Создаёт группу узлов sysfs с указанными атрибутами
sysfs_remove_group()	***fs/sysfs/group.c***	Удаляет группу sysfs, созданную с помощью **sysfs_create_group()**
tty_insert_flip_char()	***include/linux/tty_flip.h***	Отправляет символ на уровень дисциплины линии
platform_device_register_simple()	***drivers/base/platform.c***	Создаёт простое устройство платформы
platform_device_unregister()	***drivers/base/platform.c***	Отменяет регистрацию устройства платформы

Глава 12. Драйверы Видео

В этой главе

- Архитектура отображения 25
- Видео подсистема Linux 28
- Параметры дисплея 30
- API кадрового буфера 32
- Драйверы кадрового буфера 35
- Консольные драйверы 48
- Отладка 54
- Где искать информацию 55

Видео оборудование создаёт для компьютерной системы визуальный вывод для отображения. Давайте выясним в этой главе, как ядро поддерживает видео контроллеры и откроем для себя преимущества абстракции кадрового буфера. Давайте также научимся писать консольные драйверы, которые отображают сообщения, выдаваемые ядром.

Архитектура отображения

Рисунок 12.1 показывает сборку изображения на ПК-совместимой системе. Графический контроллер, который является частью Северного Моста (смотрите врезку "Северный мост" 25) соединяется с разными типами устройств отображения с использованием нескольких стандартов интерфейса (смотрите врезку "Стандарты видеокабелей" 26).

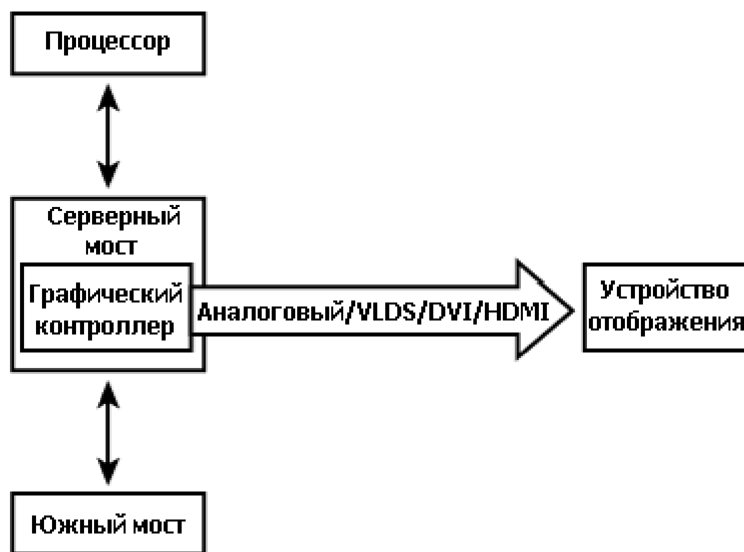

Рисунок 12.1. Подключения дисплея в системе ПК.

Логическая матрица видеографики, Video Graphics Array (VGA) является оригинальным стандартом отображения, введённым IBM, но сегодня существует больше спецификаций. VGA использует разрешение 640x480, тогда как новые стандарты, такие как ***Большая графическая матрица, Super Video Graphics Array*** (SVGA) и ***Расширенная графическая матрица, eXtended Graphics Array*** (XGA) обеспечивают более высокие разрешения экрана 800x600 и 1024x768. ***Четверть VGA, Quarter VGA*** (QVGA) панели с разрешением 320x240 являются обычными во встраиваемых устройствах, таких как КПК и смартфоны.

В мире x86 графические контроллеры совместимы с VGA и его производными, предлагающими символьный текстовый режим и графический пиксельный режим. Однако, не-x86 встраиваемые системы используют не VGA, и не имеют понятия о работе в текстовом режиме.

Северный мост

В предыдущих главах вы узнали о периферийных шинах, таких как LPC, I^{2}C, PCMCIA, PCI и USB, все из которых имеют источником Южный мост на ПК-ориентированных системах. Архитектура отображения, однако, ведёт нас к Северному мосту. Северный мост в ПК на базе архитектуры Intel - это либо

Концентратор, управляющий графикой и памятью, Graphics and Memory Controller (GMCH) или ***Концентратор, управляющий памятью, Memory Controller Hub*** (MCH). Первый содержит контроллер памяти, контроллер ***Управляющей шины, Front Side Bus*** (FSB) и графический контроллер. В последнем встроенный контроллер графики отсутствует, но обеспечивается канал ***Ускоренного графического порта, Accelerated Graphics Port*** (AGP) для подключения внешнего графического оборудования.

Рассмотрим, например, серверный мост GMCH в наборе микросхем Intel 855. Это контроллер FSB в 855 GMCH, взаимодействующий с процессорами Pentium M. Контроллер памяти поддерживает микросхемы памяти с ***Удвоенной скоростью передачи данных, Dual Data Rate*** (DDR) SDRAM. Интегрированный графический контроллер позволяет подключаться к устройствам отображения с использованием аналогового VGA, LVDS или DVI (смотрите врезку "Стандарты видеокабелей" 26). 855 GMCH позволяет одновременно направить вывод на два монитора, так что можно, например, в одно и тоже время отправить одинаковую или разную информацию на ЖК-дисплей вашего ноутбука и на внешний монитор с ЭЛТ.

Последние чипсеты Северного моста, такие как AMD 690G, в дополнение к VGA и DVI включают в себя поддержку HDMI (смотрите следующую врезку).

Стандарты видеокабелей

Соединение между видео-контроллерами и устройствами отображения определяют несколько интерфейсных стандартов. Устройства отображения и технологии подключения, которые они используют, следующие:

- Аналоговый дисплей, такой как монитор с ***электронно-лучевой трубкой*** (ЭЛТ), имеет стандартный разъем VGA.

- Цифровой плоский дисплей, такой как ЖК-дисплей на ***тонкопленочных транзисторах*** (TFT) на ноутбуке, имеет разъём с ***низковольтными дифференциальными сигналами*** (low voltage differential signaling, LVDS).

- Монитор, который соответствует спецификации ***Цифровой видеоинтерфейс***, Digital Visual Interface (DVI). DVI - это стандарт, разработанный ***Рабочей группой по цифровому изображению, Digital Display Working Group*** (DDWG) для передачи видео высокого качества. Есть три подкласса DVI: только цифровой (DVI-D), только аналоговый (DVI-A), и цифровой-и-аналоговый (DVI-I).

- Монитор, который соответствует спецификации ***Телевидения высокой чёткости, High-Definition Television*** (HDTV), использующей ***Мультимедийный интерфейс высокой чёткости, High-Definition Multimedia Interface*** (HDMI). HDMI является современным стандартом цифрового аудио-видео кабеля, который поддерживает высокую скорость передачи данных. В отличие от ориентированных только на видео стандартов, таких как DVI, HDMI, может передавать как изображение, так и звук.

Встроенные однокристальные системы как правило имеют встроенный контроллер LCD, как показано на Рисунке 12.2. Выход LCD контроллера является сигналами TTL (***транзисторно-транзисторной логики***), которые упаковывают 18 бит видео данных плоских панелей, по шесть на каждый из трёх основных цветов, красный, зелёный и синий. Некоторые КПК и телефоны используют внутренние ЖК-панели типа QVGA, которые непосредственно получают видео данные TTL, выдаваемые контроллерами LCD.

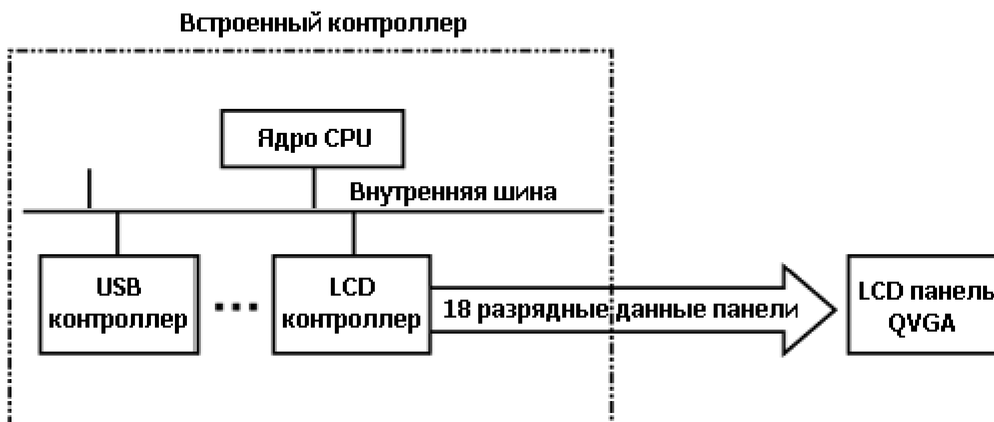

Рисунок 12.2. Подключение дисплея на встроенной системе.

Встроенное устройство, как показано на Рисунке 12.3, поддерживает две панели отображения: внутренний плоский ЖК-экран с LVDS и внешний монитор с DVI. Внутренний TFT LCD имеет разъем LVDS в качестве входа, так что для преобразования сигналов в LVDS используется микросхема-передатчик LVDS. Примером микросхемы-передатчика LVDS является DS90C363B от National Semiconductor. Внешний монитор с DVI имеет только разъём DVI, так что для преобразования сигналов используется передатчик DVI, преобразующий 18-ти разрядные видеосигналы в DVI-D. Чтобы драйвер устройства смог настроить регистры DVI передатчика, используется интерфейс I²C. Например, микросхемой-передатчиком DVI является SiI164 от Silicon Image.

Рисунок 12.3. Подключение LVDS и DVI на встроенной системе.

Видео подсистема Linux

Концепция кадровых буферов занимает центральное место для отображения на Linux, так что давайте сначала выясним, что это она предлагает.

Поскольку видеоадаптеры могут быть сделаны на основе разных аппаратных архитектур, возможно, для разных видеокарт необходимо менять реализации высших уровней ядра и приложения. Это приводит к неоднородным схемам обработки различных видеокарт. Следующая за этим непереносимость и дополнительный код требуют более значительных инвестиций и технического обслуживания. Концепция буфера кадров решает эту проблему, описывая общие абстракции и определяя программный интерфейс, что позволяет приложениям и более высоким уровням ядра быть написанными независимым от платформы образом. Рисунок 12.4 показывает преимущества кадрового буфера.

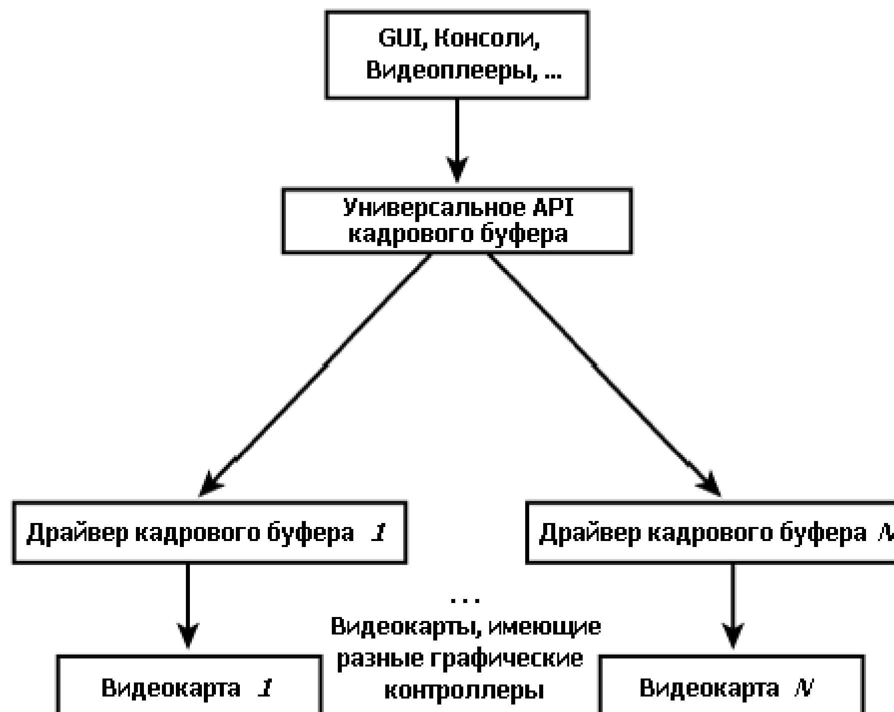

Рисунок 12.4. Преимущество кадрового буфера.

Таким образом, интерфейс кадрового буфера ядра позволяет приложениям быть независимыми от капризов аппаратной графики. Приложения запускаются неизменными поверх видео оборудования разного типа, если они и драйверы дисплея соответствуют интерфейсу кадрового буфера. Как вы скоро узнаете, универсальный программный интерфейс кадрового буфера также приносит аппаратную независимость для таких уровней ядра, как кадровый буфер драйвера консоли.

Сегодня существует множество приложений, таких как веб-браузеры и видео-плееры, работающих напрямую через интерфейс кадрового буфера. Такие приложения могут рисовать графику без помощи оконной системы.

Сервер X Windows (Xfbdev) может работать через интерфейс кадрового буфера, как показано на Рисунке 12.5.

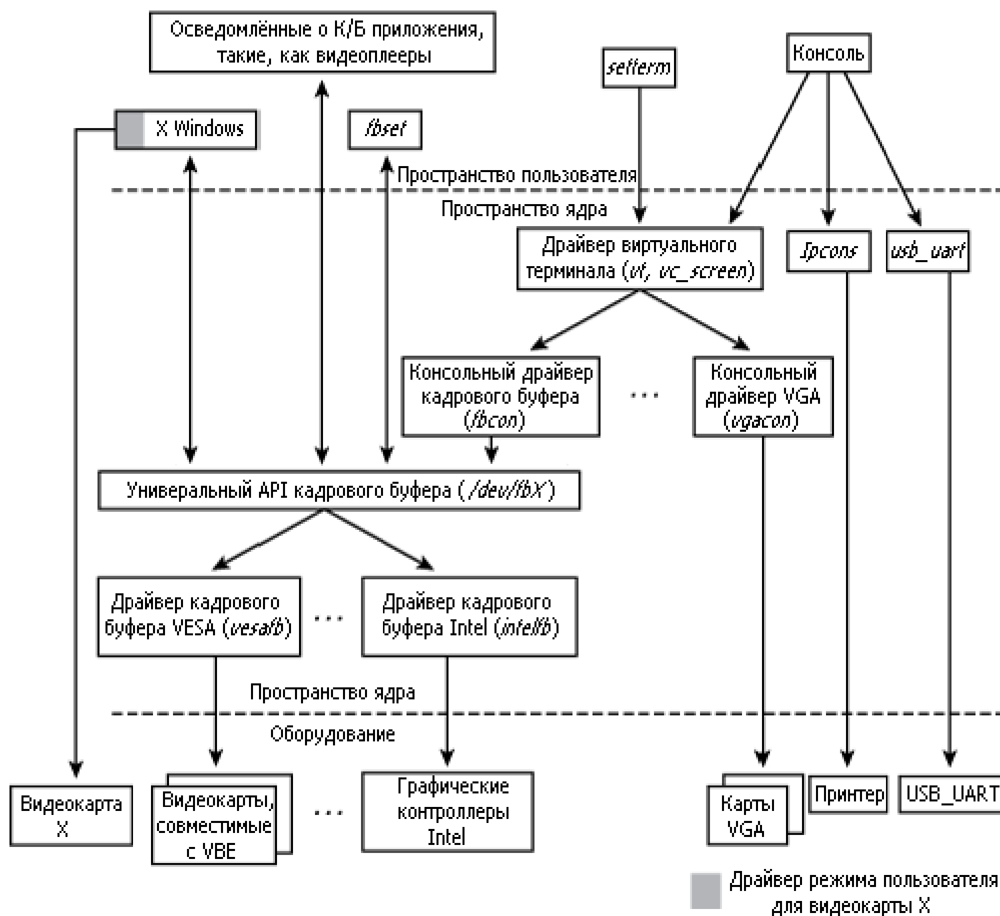

Рисунок 12.5. Видео подсистема Linux.

Видео подсистема Linux, показанная на Рисунке 12.5, представляет собой набор драйверов дисплеев низкого уровня, кадровый буфер на среднем уровне и уровень консоли, высокоуровневый драйвер виртуального терминала, драйверы пользовательского пространства, как часть X Windows, а также утилиты для настройки параметров отображения. Проследим рисунок сверху вниз:

- GUI X Windows имеет два варианта для работы с видеокартами. Он может использовать для карты любой подходящий встроенный драйвер пользовательского пространства или работать через подсистему кадрового буфера.

- Текстовый режим консоли функционирует поверх символьного драйвера виртуального терминала. Виртуальные терминалы, введенные в разделе "Драйверы TTY" в Главе 6, "Драйверы последовательных портов", являются полноэкранными текстовыми терминалами, которые вы получаете при входе в систему в текстовом режиме. Как и X Windows, текстовые консоли имеют два варианта работы. Они могут либо работать поверх аппаратно-зависимого драйвера консоли, либо использовать универсальный кадровый буфер драйвера консоли (***fbcon***), если ядро поддерживает низкоуровневый драйвер кадрового буфера для используемой карты.

Параметры дисплея

Иногда настройка свойств, связанных с вашим дисплеем, может быть сделана только изменениями в драйвере, которые вам необходимо сделать, чтобы включить видео на вашем устройстве, так что давайте начнём изучение драйверов видео, глядя на общие параметры отображения. Будем считать, что соответствующий драйвер удовлетворяет интерфейсу кадрового буфера, и используем для получения характеристик дисплея утилиту ***fbset***:

```
bash> fbset
mode "1024x768-60"
    # D: 65.003 MHz, H: 48.365 kHz, V: 60.006 Hz
    geometry 1024 768 1024 768 8
    timings 15384 168 16 30 2 136 6
    hsync high
    vsync high
    rgba 8/0,8/0,8/0,0/0
endmode
```

D: значение выхода, установленное для частоты точки, ***dotclock***, которая является скоростью, с которой видео оборудование рисует на экране пиксели. Значение 65.003 МГц в предыдущей распечатке означает, что это займёт у видео контроллера для отрисовки одного пикселя (1/65.003*1000000), или около 15 384 пикосекунд. Такая продолжительность называется ***pixclock*** и показывается как первый цифровой параметра в строке, начинающейся с ***timings***. Числа после "**geometry**" сообщают, что в видимое и виртуальное разрешение экрана 1024x768 (SVGA), и что числом битов, необходимых для хранения информации для пикселя, является 8.

H: значение определяет горизонтальную частоту сканирования, которая является числом горизонтальных линий отображения, сканируемых видеокартой за одну секунду. Эта величина является обратной времени pixclock, умноженному на разрешения по оси X. **V**: значение частоты обновления всего экрана. Это величина, обратная pixclock, умноженной на видимое разрешение по оси X и видимое разрешение по оси Y, которая составляет в этом примере около 60 Гц. Другими словами, ЖК дисплей обновляется 60 раз в секунду.

Видео контроллеры вырабатывают в конце каждой строки импульс горизонтальной синхронизации (**HSYNC**), а после отображения каждого кадра - импульс вертикальной синхронизации (**VSYNC**). Продолжительность HSYNC (в терминах пикселей) и VSYNC (в пересчете на линию пикселей) показывается как два последних параметра в строке, начинающейся с "**timings**". Чем больше дисплей, тем больше вероятные значения **HSYNC** и **VSYNC**. Четыре числа до продолжительности HSYNC в строке **timings** сообщают длину отступа от правого края дисплея, левый отступ, отступ снизу и верхний отступ, соответственно. Эти параметры наглядно показывают ***Documentation/fb/framebuffer.txt*** и страница справки ***fb.modes***.

Чтобы связать эти параметры вместе, давайте вычислим значение pixclock для данной частоты обновления, которая в нашем примере равна 60.006 Гц:

```
dotclock = (разрешение по X + левый отступ + правый отступ
          + длительность HSYNC) * (разрешение по Y + верхний отступ
          + нижний отступ + длительность VSYNC) * частота обновления
         = (1024 + 168 + 16 + 136) * (768 + 30 + 2 + 6) * 60.006
```

```
         = 65.003 MHz
pixclock = 1/dotclock
         = 15384 пикосекунд (которая соответствует показанному выше выводу
fbset)
```

API кадрового буфера

Давайте теперь промочим наши ноги в API кадрового буфера. Уровень ядра кадрового буфера экспортирует узлы устройств в пользовательское пространство, так что приложения могут обращаться к любому поддерживаемому видео устройству. Узлом, связанным с кадровым буфером устройства ***X***, является ***/dev/fbX***. Ниже перечислены основные структуры данных, которые интересны пользователям API кадрового буфера. Внутри ядра они определены в ***include/linux/fb.h***, тогда как на стороне пользователя их определения находятся в ***/usr/include/linux/fb.h***:

1. Изменяемая информация, относящаяся к видеокарте, которую вы видели в выводе ***fbset*** предыдущего раздела, находилась в структуре **fb_var_screeninfo**. Эта структура содержит такие поля, как разрешение по оси X, разрешение по оси Y, биты, необходимые для хранения пикселя, pixclock, продолжительность **HSYNC**, продолжительность **VSYNC** и размер отступов. Эти значения программируется пользователем:

```
struct fb_var_screeninfo {
    __u32 xres;           /* Видимое разрешение по оси X */
    __u32 yres;           /* Видимое разрешение по оси Y */
    /* ... */
    __u32 bits_per_pixel; /* Число бит, требуемое для хранения пикселя */
    /* ... */
    __u32 pixclock;       /* Частота пикселя в пикосекундах */
    __u32 left_margin;    /* Время от импульса синхронизации до картинки */
    __u32 right_margin;   /* Время от картинки до импульса синхронизации */
    /* ... */
    __u32 hsync_len;      /* Длительность горизонтальной синхронизации */
    __u32 vsync_len;      /* Длительность вертикальной синхронизации */
    /* ... */
};
```

2. В структуре **fb_fix_screeninfo** содержится неизменяемая информация о видео оборудовании, такая как начальный адрес и размер памяти кадрового буфера. Эти значения не могут быть изменены пользователем:

```
struct fb_fix_screeninfo {
    char id[16];                /* Строка идентификации */
    unsigned long smem_start;   /* Начальный адрес памяти кадрового буфера */
    __u32 smem_len;             /* Размер памяти кадрового буфера */
    /* ... */
};
```

3. Структура **fb_cmap** определяет карту цветов, которые используются для передачи определённых пользователем цветов видео оборудованию. Вы можете использовать эту структуру для определения соотношений RGB (красного, зелёного, синего), которые вы желаете иметь для разных цветов:

```
struct fb_cmap {
    __u32 start;    /* Первая запись */
    __u32 len;      /* Число записей */
    __u16 *red;     /* Значения красного */
    __u16 *green;   /* Значения зелёного */
    __u16 *blue;    /* Значения синего */
```

```
    __u16 *transp; /* Прозрачность. Обсуждается позже */
};
```

Распечатка 12.1 является простым приложением, которое работает с API кадрового буфера. Программа очищает экран, воздействуя на */dev/fb0*, узел устройства кадрового буфера, соответствующий дисплею. Сначала аппаратно-независимым образом с использованием API кадрового буфера расшифровывается видимое разрешение экрана и число бит на пиксель, **FBIOGET_VSCREENINFO**. Эта команда интерфейса собирает переменные параметры дисплея работая со структурой **fb_var_screeninfo**. Затем программа переходит к выполнению для памяти кадрового буфера **mmap()** и очищает каждый бит, составляющий пиксель.

Распечатка 12.1. Очистка дисплея аппаратно-независимым способом

Код:

```
#include <stdio.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <stdlib.h>

struct fb_var_screeninfo vinfo;

int
main(int argc, char *argv[])
{
    int fbfd, fbsize, i;
    unsigned char *fbbuf;

    /* Открываем видеопамять */
    if ((fbfd = open("/dev/fb0", O_RDWR)) < 0) {
        exit(1);
    }

    /* Получаем изменяемые параметры изображения */
    if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
        printf("Bad vscreeninfo ioctl\n");
        exit(2);
    }

    /* Размер кадрового буфера =
        (разрешение по X * разрешение по Y * байты на пиксель) */
    fbsize = vinfo.xres*vinfo.yres*(vinfo.bits_per_pixel/8);

    /* Отображаем видеопамять */
    if ((fbbuf = mmap(0, fbsize, PROT_READ|PROT_WRITE,
        MAP_SHARED, fbfd, 0)) == (void *) -1){
        exit(3);
    }

    /* Очищаем экран */
    for (i=0; i<fbsize; i++) {
        *(fbbuf+i) = 0x0;
```

```
    }

    munmap(fbbuf, fbsize);
    close(fbfd);
}
```

Мы рассмотрим другое приложение, работающее с кадровым буфером, когда изучим доступ к областям памяти из пользовательского пространства в Главе 19, "Драйверы в пространстве пользователя".

Драйверы кадрового буфера

Теперь, когда вы понимаете идею в API кадрового буфера и как он обеспечивает аппаратную независимость, давайте изучим архитектуру низкоуровнего драйвера устройства с кадровым буфером на примере навигационной системы.

Пример устройства: Навигационная система

Рисунок 12.6 показывает видео операции на примере навигационной системы машины, построенной на встроенном процессоре. Приёмник GPS передаёт потоки координат на процессор через интерфейс UART. Приложение рисует графику на основе полученной информации о местоположении и обновляет кадровый буфер в системной памяти. Драйвер кадрового буфера через DMA передаёт данные изображения в буферы дисплея, которые являются частью LCD контроллера процессора. Контроллер передаёт пиксельные данные для отображения на ЖК панель QVGA.

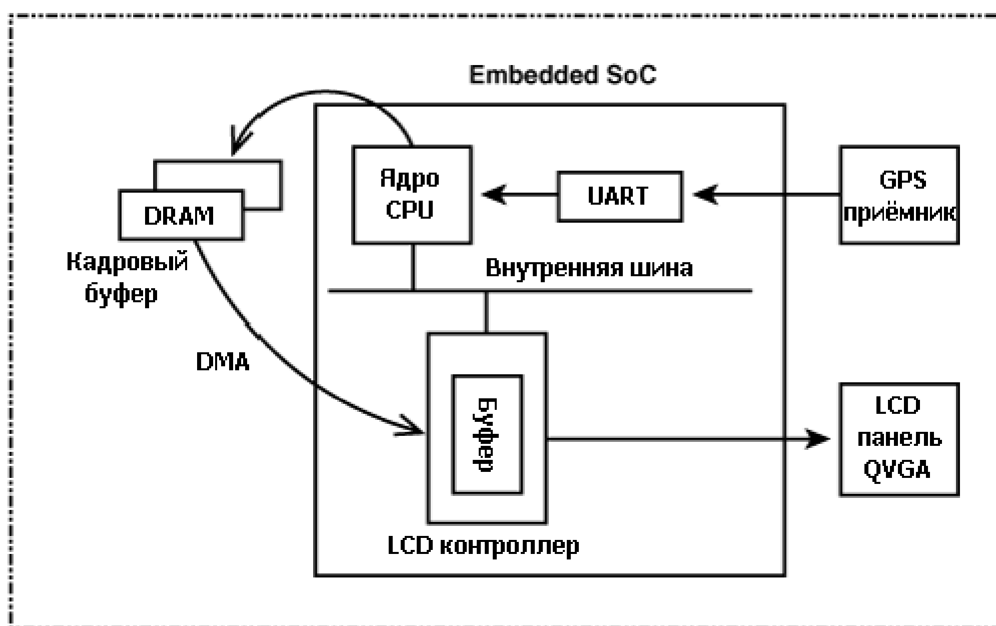

Рисунок 12.6. Отображение на навигационном устройстве с Linux.

Наша цель заключается в разработке программного обеспечения для этой системы, работающего с видео. Давайте предположим, что Linux поддерживает процессор, используемый в этом навигационном устройстве, и что все зависящие от архитектуры интерфейсы, такие как DMA, поддерживаются ядром.

> Одна из возможных аппаратных реализаций устройства, показанного на Рисунке 12.6, заключается в использовании процессоре Freescale i.MX21. Ядром процессора в этом случае является ядро ARM9, а встроенным видеоконтроллером является контроллер жидкокристаллического дисплея Liquid Crystal Display Controller (LCDC). Микропроцессоры, как правило, имеют высокопроизводительную внутреннюю шину, которая подключена к таким контроллерам, как DRAM и видео. В случае iMX.21 эта шина называется ***Улучшенная высокопроизводительная шина, Advanced High-Performance Bus*** (AHB). LCDC подключается к AHB.

Программное обеспечение навигационной системы для работы с видео построено в целом

как приложение GPS, работающее через низкоуровневый драйвер кадрового буфера для контроллера ЖК-дисплея. Приложение получает координат местоположения от приёмника GPS читая ***/dev/ttySX***, где ***X*** - это номер UART, подключённого к ресиверу. Затем оно переводит зафиксированную географическую информацию в картинку и записывает пиксельные данные в кадровый буфер, связанных с контроллером ЖК-дисплея. Это делается по аналогии с Распечаткой 12.1, за исключением того, что данные изображения отличаются от нулей для очистки экрана.

Далее в этом разделе основное внимание уделяется только низкоуровневому драйверу кадрового буфера. Как и многие другие драйверные подсистемы, полный набор объектов, режимов и опций, предлагаемых уровнем ядра кадрового буфера, является сложным и может быть изучен только при получении опыта кодирования. Драйвер кадрового буфера для примера системы навигации является относительно простым и является лишь отправной точкой для более глубоких исследований.

Таблица 12.1 описывает регистровую модель LCD контроллера показанного на Рисунке 12.6. Драйвер кадрового буфера в Распечатке 12.2 работает с этими регистрами.

Таблица 12.1. Назначение регистров контроллера LCD, показанного на Рисунке 12.6

Имя регистра	**Используется для конфигурации**
SIZE_REG	Максимальные размеры ЖК панели по X и Y
HSYNC_REG	Длительность **HSYNC**
VSYNC_REG	Длительность **VSYNC**
CONF_REG	Число битов на пиксель, полярность пикселей, делители частоты для генерации pixclock, режим цветной/монохромный и так далее
CTRL_REG	Включение/выключение LCD контроллера, частоты и DMA
DMA_REG	Начальный адрес для DMA кадрового буфера, размер и размеры меток
STATUS_REG	Статусные значения
CONTRAST_REG	Уровень контрастности

Наш драйвер кадрового буфера (названный ***myfb***) реализован в Распечатке 12.2 в виде драйвера платформы. Как вы узнали в Главе 6, платформа представляет собой псевдо шину, обычно используемую для подключения простых устройств, интегрированных на кристалле, к модели устройств ядра. Архитектурно-зависимый код настройки (в ***arch/your-arch/your-platform/***) добавляет платформу, используя **platform_device_add()**; но для простоты, метод **probe()** драйвера ***myfb*** выполняет это до регистрации себя в качестве драйвера платформы. Обратитесь к разделу "Пример устройства: Сотовый телефон" в Главе 6, где описана общая архитектура драйвера платформы и соответствующие точки входа.

Структуры данных

Давайте посмотрим на основные структуры данных и методы, связанные с драйверами кадрового буфера, а затем рассмотрим ***myfb***. Основными структурами являются следующие две:

1. Структура **fb_info** является центральной структурой данных драйверов кадрового буфера. Эта структура определена в ***include/linux/fb.h*** следующим образом:

```
struct fb_info {
    /* ... */
    struct fb_var_screeninfo var; /* Изменяемая информация экрана. Обсуждалась ранее. */
    struct fb_fix_screeninfo fix; /* Неизменяемая информация экрана. Обсуждалась ранее. */
    /* ... */
    struct fb_cmap cmap;          /* Таблица цветов. Обсуждалась ранее. */
    /* ... */
    struct fb_ops *fbops;         /* Операции драйвера. Обсуждается далее. */
    /* ... */
    char __iomem *screen_base;    /* Виртуальный адрес кадрового буфера */
    unsigned long screen_size;    /* Размер кадрового буфера */
    /* ... */
    /* Остальное зависит от устройства */
    void *par; /* Private area */
};
```

   Память для **fb_info** выделяется через **framebuffer_alloc()**, библиотечная процедура предоставляется ядром кадрового буфера. Эта функция также имеет в качестве аргумента размер выделяемой области и добавляет в конец выделенной памяти для **fb_info**. Обратиться к этой области можно используя указатель **par** структуры **fb_info**. Значения полей **fb_info**, таких как **fb_var_screeninfo** и **fb_fix_screeninfo** были обсуждены в разделе "API кадрового буфера".

2. Структура **fb_ops** содержит адреса всех точек входа, предоставляемых низкоуровневым драйвером кадрового буфера. Первые несколько методов в **fb_ops** необходимы для функционирования драйвера, а остальные являются необязательными, обеспечивающими ускорение графики. Ответственность каждой функции кратко объясняется комментариями:

```
struct fb_ops {
    struct module *owner;
    /* Открытие драйвера */
    int (*fb_open)(struct fb_info *info, int user);
    /* Закрытие драйвера */
    int (*fb_release)(struct fb_info *info, int user);
    /* ... */
    /* Проверка видео параметров на здравый смысл */
    int (*fb_check_var)(struct fb_var_screeninfo *var,
                        struct fb_info *info);
    /* Конфигурирование регистров видео контроллера */
    int (*fb_set_par)(struct fb_info *info);
    /* Создание карты палитры псевдо цветов */
    int (*fb_setcolreg)(unsigned regno, unsigned red,
                        unsigned green, unsigned blue,
```

```
                         unsigned transp, struct fb_info *info);
     /* Включение/отключение дисплея */
     int (*fb_blank)(int blank, struct fb_info *info);
     /* ... */
     /* Метод ускорения для заполнения прямоугольника линиями пикселей */
     void (*fb_fillrect)(struct fb_info *info,
                         const struct fb_fillrect *rect);
     /* Метод ускорения копирования одной прямоугольной области в другую */
     void (*fb_copyarea)(struct fb_info *info,
                 const struct fb_copyarea *region);
     /* Метод ускорения рисования на экране картинки */
     void (*fb_imageblit)(struct fb_info *info,
                     const struct fb_image *image);
     /* Метод ускорения для поворота изображения */
     void (*fb_rotate)(struct fb_info *info, int angle);
     /* Интерфейс ioctl для поддержки зависимых от устройства команд */
     int (*fb_ioctl)(struct fb_info *info, unsigned int cmd,
                     unsigned long arg);
     /* ... */
 };
```

Давайте теперь посмотрим на методы драйвера, реализованные для драйвера ***myfb*** в Распечатке 12.2.

Проверка и установка параметров

Проверку разумности переменных, таких как разрешение по X, разрешение по Y и число битов на пиксель, выполняет метод **fb_check_var()**. Таким образом, если вы используете fbset для установки разрешения по X менее минимального, поддерживаемого контроллером LCD (64 в нашем примере), то эта функция будет ограничивать его до минимального значения, позволяемого оборудованием.

fb_check_var() также устанавливает соответствующий формат RGB. Наш пример использует 16 бит на пиксель, и контроллер связывает каждое слово данных в кадровом буфере с часто используемым кодом RGB565: 5 бит для красного, 6 бит для зелёного и 5 бит для синего. Смещения в словах данных для каждого из трёх цветов также устанавливаются соответственно.

Метод **fb_set_par()** настраивает регистры контроллера LCD в зависимости от значений, указанных в **fb_info.var**. Это включает в себя настройку

- Длительности горизонтальной синхронизации, отступ слева и справа в **HSYNC_REG**;
- Длительности вертикальной синхронизации, отступ сверху и снизу в **VSYNC_REG**;
- Видимое разрешение по X и Y в **SIZE_REG**;
- Параметры DMA в **DMA_REG**.

Предположим, что приложение GPS пытается изменить разрешение QVGA дисплея на 50x50. Ниже приводится цепь событий:

1. Дисплей проинициализирован в разрешение QVGA:

```
bash> fbset
```

```
mode "320x240-76"
    # D: 5.830 MHz, H: 18.219 kHz, V: 75.914 Hz
    geometry 320 240 320 240 16
    timings 171521 0 0 0 0 0 0
    rgba 5/11,6/5,5/0,0/0
endmode
```

2. Приложение делает следующее:

```
struct fb_var_screeninfo vinfo;
fbfd = open("/dev/fb0", O_RDWR);
vinfo.xres = 50;
vinfo.yres = 50;
vinfo.bits_per_pixel = 8;

ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo);
```

   Заметим, что это эквивалент команды

```
fbset -xres 50 -yres 50 -depth 8
```

3. ioctl **FBIOPUT_VSCREENINFO** на предыдущем этапе вызывает срабатывание **myfb_check_var()**. Этот метод драйвера выражает недовольство и округляет запрошенное разрешение до минимума, поддерживаемого оборудованием, которое в этом случае 64x64.

4. из ядра кадрового буфера вызывается **myfb_set_par()**, которая программирует новые параметры отображения в регистры контроллера LCD.

5. **fbset** теперь выводит новые параметры:

```
bash> fbset
mode "64x64-1423"
    # D: 5.830 MHz, H: 91.097 kHz, V: 1423.386 Hz
    geometry 64 64 320 240 16
    timings 171521 0 0 0 0 0 0
    rgba 5/11,6/5,5/0,0/0
endmode
```

Цветовые режимы

Обычные цветовые режимы, поддерживаемые видео оборудованием, включают ***псевдо-цвет*** и ***естественные цвета*** (true color). В первом случае индексные номера преобразуются в пиксели формата RGB путём кодировок. Выбрав подмножество доступных цветов и с помощью индексирования соответствующих цветов вместо значения самого пикселя, вы можете снизить требования к памяти кадрового буфера. Однако, используемое оборудование должно поддерживать эту схему изменения цветового набора (или ***палитры***).

В режиме true color (который есть то, что поддерживает наш пример LCD контроллера), изменение палитр не являются актуальным. Тем не менее, вам всё равно придётся удовлетворить потребности консольного драйвера кадрового буфера, который использует только 16 цветов. Для этого вам необходимо создать псевдо палитру кодированием

соответствующих 16-ти необработанных значений RGB в биты, которые могут быть напрямую переданы оборудованию. Эта псевдо палитра хранится в поле **pseudo_palette** структуры **fb_info**. В Распечатке 12.2, **myfb_setcolreg()** заполняет её следующим образом:

```
((u32*)(info->pseudo_palette))[color_index] =
            (red << info->var.red.offset)     |
            (green << info->var.green.offset) |
            (blue << info->var.blue.offset)   |
            (transp << info->var.transp.offset);
```

Наш LCD контроллер использует 16 бит на пиксель и формат RGB565, и как вы видели ранее, метод **fb_check_var()** гарантирует, что значения красного, зелёного и синего имеют смещения 11, 5 и 0, соответственно. В дополнение к индексу цвета и значениям красного, синего и зелёного, **fb_setcolreg()** принимает аргумент **transp**, чтобы указать желаемые эффекты прозрачности. Этот механизм, называемый альфа-канал, сочетает указанное значение пикселя с цветом фона. LCD контроллер в данном примере не поддерживает альфа-смешивание, так что **myfb_check_var()** устанавливает смещение и размер **transp** в ноль.

Абстракция кадрового буфера достаточно мощна, чтобы изолировать приложения от характеристик панели дисплея, будь это RGB или BGR, или что-то другое. Смещения красного, синего и зелёного, установленные **fb_check_var()**, передаются в пользовательское пространство с помощью структуры **fb_var_screeninfo**, получаемой через **ioctl() FBIOGET_VSCREENINFO**. Поскольку такие приложения, как X Windows совместимы с кадровым буфером, они рисуют пиксели в кадровом буфере в зависимости от смещения цветов, возвращаемых этим **ioctl()**.

Размеры битовых полей, используемые кодировкой RGB (5+6+5 = 16 в данном случае) называются глубиной цвета, которая используется консольным драйвером кадрового буфера для выбора файла логотипа для отображения во время загрузки (смотрите раздел "Логотип при загрузке" 53).

Гашение экрана

Поддержку для гашения и включения дисплея обеспечивает метод **fb_blank()**. Это используется в основном для управления питанием. Для гашения дисплея навигационной системы после 10-ти минутного периода бездействия сделайте следующее:

```
bash> setterm -blank 10
```

Эта команда проходит вниз по уровням до уровня кадрового буфера и в результате вызывает **myfb_blank()**, который программирует соответствующие биты в CTRL_REG.

Методы ускорения

Если ваш пользовательский интерфейс должен выполнять тяжёлые операции, таких как смешивание, изменение размера, двигать рисунки, или генерировать динамический градиент, вам, вероятно, потребуется ускорение графики, чтобы получить приемлемый уровень производительности. Давайте кратко рассмотрим методы **fb_ops**, которые можно использовать, если ваша видеокарта поддерживает ускорение графики.

Метод **fb_imageblit()** рисует на дисплее изображение. Эта точка входа даёт возможность драйверу использовать любые специальные возможности, которыми может обладать ваш видео контроллер, чтобы ускорить эту операцию. **cfb_imageblit()** является универсальной функцией библиотеки, предоставляемой ядром кадрового буфера, чтобы выполнять это, если у оборудование не имеет ускорителя. Она используется, например, для вывода логотипа на экран во время загрузки. **fb_copyarea()** копирует прямоугольную область из одной области экрана в другую. **cfb_copyarea()** представляет собой оптимальный способ сделать это, если ваш графический контроллер не обладает магией, чтобы ускорить эту операцию. Метод **fb_fillrect()** быстро заполняет прямоугольник пиксельными линиями. **cfb_fillrect()** предлагает общий способ достижения этой цели без ускорителя. Контроллер LCD в нашей навигационной системы не предоставляет ускорения, так что драйвер в примере заполняет эти методы используя универсальные оптимизированные процедуры, предлагаемые ядром кадрового буфера.

DirectFB

DirectFB (www.directfb.org) является библиотекой, построенной поверх интерфейса кадрового буфера, которая предоставляет основу простого оконного менеджера и способы для аппаратного ускорения графики и виртуальные интерфейсы, которые позволяют сосуществовать нескольким приложениям, работающим с кадровым буфером. DirectFB, наряду с поддерживающими ускорение драйверами кадрового буфера внизу и использующим DirectFB движком рендеринга, таким как Cairo (www.cairographics.org) поверх, иногда используется в интенсивно работающих с графикой встроенных устройствах вместо более традиционных решений, таких как X Windows.

DMA из кадрового буфера

LCD контроллер в навигационной системе содержит механизм DMA, который получает кадры изображения из системной памяти. Контроллер отправляет полученные графические данные на панель отображения. Скорость DMA поддерживает частоту обновления экрана. Некэшируемый кадровый буфер, подходящий для согласованного доступа, выделяется с использованием **dma_alloc_coherent()** в **myfb_probe()**. (Мы обсудили согласованного отображение DMA в Главе 10, "Соединение компонентов периферии".) **myfb_set_par()** записывает этот выделенный адрес DMA в регистр **DMA_REG** LCD контроллера.

Когда драйвер включает DMA вызывая **myfb_enable_controller()**, контроллер начинает переправлять пиксельные данные из кадрового буфера на дисплей с помощью синхронного DMA. Таким образом, когда приложение GPS отображает кадровый буфер (с помощью **mmap ()**) и записывает в неё информацию о местоположении, на LCD закрашиваются пиксели.

Контраст и подсветка

LCD контроллер навигационной системы поддерживает управление контрастом с помощью регистра **CONTRAST_REG**. Драйвер экспортирует это в пространство пользователя с помощью **myfb_ioctl()**. Приложение GPS управляет контрастом следующим образом:

```
unsigned int my_fd, desired_contrast_level = 100;
/* Открываем кадровый буфер */
my_fd = open("/dev/fb0", O_RDWR);
ioctl(my_fd, MYFB_SET_BRIGHTNESS, &desired_contrast_level);
```

ЖК панель навигационной системы подсвечивается с помощью задней подсветки. Процессор управляет инвертером подсветки через выводы GPIO, поэтому вы можете включить или отключить свет управляя соответствующими контактами. Ядро абстрагирует универсальный интерфейс подсветки через узлы sysfs. Чтобы связаться с этим интерфейсом, ваш драйвер должен заполнить структуру **backlight_ops** методами получения и обновления настроек подсветки, и зарегистрировать её в ядре с помощью **backlight_device_register()**. Загляните в ***drivers/video/backlight/*** для поиска исходников интерфейса подсветки и поищите в дереве ***drivers/*** по названию **backlight_device_register()** видео драйверы, использующие этот интерфейс. В Распечатке 12.2 операции управления подсветкой не реализованы.

Распечатка 12.2. Драйвер кадрового буфера для навигационной системы

Код:

```
#include <linux/fb.h>
#include <linux/dma-mapping.h>
#include <linux/platform_device.h>

/* Карта адресов регистров LCD контроллера */
#define LCD_CONTROLLER_BASE 0x01000D00
#define SIZE_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE))
#define HSYNC_REG    (*(volatile u32 *)(LCD_CONTROLLER_BASE + 4))
#define VSYNC_REG    (*(volatile u32 *)(LCD_CONTROLLER_BASE + 8))
#define CONF_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE + 12))
#define CTRL_REG     (*(volatile u32 *)(LCD_CONTROLLER_BASE + 16))
#define DMA_REG      (*(volatile u32 *)(LCD_CONTROLLER_BASE + 20))
#define STATUS_REG   (*(volatile u32 *)(LCD_CONTROLLER_BASE + 24))
#define CONTRAST_REG (*(volatile u32 *)(LCD_CONTROLLER_BASE + 28))
#define LCD_CONTROLLER_SIZE 32

/* Ресурсы для LCD контроллера как устройства платформы */
static struct resource myfb_resources[] = {
    [0] = {
        .start = LCD_CONTROLLER_BASE,
        .end   = LCD_CONTROLLER_SIZE,
        .flags = IORESOURCE_MEM,
    },
};

/* Определение устройства платформы */
static struct platform_device myfb_device = {
    .name = "myfb",
    .id = 0,
    .dev = {
        .coherent_dma_mask = 0xffffffff,
    },
    .num_resources = ARRAY_SIZE(myfb_resources),
    .resource = myfb_resources,
```

```
};

/* Установка параметров LCD контроллера */
static int
myfb_set_par(struct fb_info *info)
{
    unsigned long adjusted_fb_start;
    struct fb_var_screeninfo *var = &info->var;
    struct fb_fix_screeninfo *fix = &info->fix;

    /* Старшие 16 бит HSYNC_REG содержат длительность HSYNC, следующие 8
       содержат отступ слева, 8 младших содержат отступ справа */
    HSYNC_REG = (var->hsync_len << 16) |
                (var->left_margin << 8)|
                (var->right_margin);

    /* Старшие 16 бит VSYNC_REG содержат длительность VSYNC, следующие 8
       содержат отступ сверху, 8 младших содержат отступ снизу */
    VSYNC_REG = (var->vsync_len << 16) |
                (var->upper_margin << 8)|
                (var->lower_margin);

    /* Старшие 16 бит SIZE_REG содержат xres, младшие 16 содержат yres */
    SIZE_REG = (var->xres << 16) | (var->yres);

    /* Устанавливаем число битов на пиксель, полярность битов, делители
частоты
       для pixclock, и режим цвет/монохром в CONF_REG */
    /* ... */

    /* Заполняем DMA_REG начальным адресом кадрового буфера
       выделенного когерентно в myfb_probe(). Изменяем этот адрес
       для учёта смещения начала области экрана */
    adjusted_fb_start = fix->smem_start +
        (var->yoffset * var->xres_virtual + var->xoffset) *
        (var->bits_per_pixel) / 8;
    __raw_writel(adjusted_fb_start, (unsigned long *)DMA_REG);
    /* Устанавливаем размер DMA и размер меток в DMA_REG */
    /* ... */

    /* Устанавливаем неизменяемую информацию */
    fix->accel = FB_ACCEL_NONE;        /* Аппаратного ускорения нет */
    fix->visual = FB_VISUAL_TRUECOLOR; /* Режим True color */
    fix->line_length = var->xres_virtual * var->bits_per_pixel/8;
    return 0;
}

/* Включение LCD контроллера */
static void
myfb_enable_controller(struct fb_info *info)
{
    /* Включение LCD контроллера, старт DMA, запуск тактовых частот
       и включение питания через запись в CTRL_REG */
    /* ... */
}
```

```
/* Выключение LCD контроллера */
static void
myfb_disable_controller(struct fb_info *info)
{
    /* Включение LCD контроллера, остановка DMA, выключение тактовых частот
       и питания через запись в CTRL_REG */
    /* ... */
}

/* Проверка разумности и изменение переменных */
static int
myfb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
{
    /* Округление вверх до минимального разрешения, поддерживаемого
       LCD контроллером */
    if (var->xres < 64) var->xres = 64;
    if (var->yres < 64) var->yres = 64;

    /* ... */
    /* Это оборудование поддерживает формат цвета RGB565.
       Для подробностей смотрите раздел "Режимы цвета" */
    if (var->bits_per_pixel == 16) {
        /* Кодирование красного */
        var->red.length = 5;
        var->red.offset = 11;
        /* Кодирование зелёного */
        var->green.length = 6;
        var->green.offset = 5;
        /* Кодирование синего */
        var->blue.length = 5;
        var->blue.offset = 0;
        /* Оборудование не поддерживает альфа канал */
        var->transp.length = 0;
        var->transp.offset = 0;
    }
    return 0;
}

/* Включение/выключение экрана */
static int
myfb_blank(int blank_mode, struct fb_info *info)
{
    switch (blank_mode) {
    case FB_BLANK_POWERDOWN:
    case FB_BLANK_VSYNC_SUSPEND:
    case FB_BLANK_HSYNC_SUSPEND:
    case FB_BLANK_NORMAL:
        myfb_disable_controller(info);
        break;
    case FB_BLANK_UNBLANK:
        myfb_enable_controller(info);
        break;
    }
    return 0;
```

```
}

/* Конфигурирование карты палитры псевдо цветов */
static int
myfb_setcolreg(u_int color_index, u_int red, u_int green,
            u_int blue, u_int transp, struct fb_info *info)
{
    if (info->fix.visual == FB_VISUAL_TRUECOLOR) {
        /* Делаем требуемые преобразования для конвертации красного, синего,
зелёного и
           прозрачности в значения, которые могут быть напрямую восприняты
оборудованием */
        /* ... */

        ((u32 *)(info->pseudo_palette))[color_index] =
                    (red << info->var.red.offset)     |
                    (green << info->var.green.offset) |
                    (blue << info->var.blue.offset)   |
                    (transp << info->var.transp.offset);
    }
    return 0;
}

/* Зависимое от устройства определение ioctl */
#define MYFB_SET_BRIGHTNESS _IOW('M', 3, int8_t)

/* Зависимая от устройства ioctl */
static int
myfb_ioctl(struct fb_info *info, unsigned int cmd,
unsigned long arg)
{
    u32 blevel ;
    switch (cmd) {
    case MYFB_SET_BRIGHTNESS :
        copy_from_user((void *)&blevel, (void *)arg,
                    sizeof(blevel)) ;
        /* Пишем blevel в CONTRAST_REG */
        /* ... */
        break;
    default:
        return -EINVAL;
    }
    return 0;
}

/* Структура fb_ops */
static struct fb_ops myfb_ops = {
    .owner        = THIS_MODULE,
    .fb_check_var = myfb_check_var,/* Проверка параметров */
    .fb_set_par   = myfb_set_par,  /* Программирование регистров контроллера
*/
    .fb_setcolreg = myfb_setcolreg,/* Установка карты цветов */
    .fb_blank     = myfb_blank,    /* Включение/выключение экрана */
    .fb_fillrect  = cfb_fillrect,  /* Универсальная функция для заливки
прямоугольника */
```

```
    .fb_copyarea  = cfb_copyarea,  /* Универсальная функция для копирования
области */
    .fb_imageblit = cfb_imageblit, /* Универсальная функция для рисования */
    .fb_ioctl     = myfb_ioctl,    /* Зависимая от устройства ioctl */
};

/* Процедура устройства платформы probe() */
static int __init
myfb_probe(struct platform_device *pdev)
{
    struct fb_info *info;
    struct resource *res;

    info = framebuffer_alloc(0, &pdev->dev);
    /* ... */
    /* Получаем соответствующие ресурсы, заданные при регистрации
       данного platform_device */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    /* Получаем разрешение ядра на использование куска памяти ввода/вывода,
       начиная с LCD_CONTROLLER_BASE и размером в
       LCD_CONTROLLER_SIZE байт */
    res = request_mem_region(res->start, res->end - res->start + 1,
                             pdev->name);

    /* Заполняем структуру fb_info постоянными (info->fix) и переменными
      (info->var) значениями, такими как размер кадрового буфера, xres, yres,
      bits_per_pixel, fbops, cmap, и так далее */
    initialize_fb_info(info, pdev); /* Отсутствует */
    info->fbops = &myfb_ops;
    fb_alloc_cmap(&info->cmap, 16, 0);

    /* Связываем когерентно память кадрового буфера с DMA. info->screen_base
       содержит адрес процессора для отображаемого буфера,
    info->fix.smem_start carries the associated hardware address */
    info->screen_base = dma_alloc_coherent(0, info->fix.smem_len,
                        (dma_addr_t *)&info->fix.smem_start,
                        GFP_DMA | GFP_KERNEL);
    /* Устанавливаем информацию в info->var для соответствующих
       регистров LCD контроллера */
    myfb_set_par(info);

    /* Регистрируемся в ядре кадрового буфера */
    register_framebuffer(info);
    return 0;
}

/* Процедура драйвера платформы remove() */
static int
myfb_remove(struct platform_device *pdev)
{
    struct fb_info *info = platform_get_drvdata(pdev);
    struct resource *res;

    /* Запрет обновления экрана, выключение DMA,... */
    myfb_disable_controller(info);
```

```
    /* Отмена регистрации драйвера кадрового буфера */
    unregister_framebuffer(info);
    /* Освобождение карты цветов */
    fb_dealloc_cmap(&info->cmap);
    kfree(info->pseudo_palette);

    /* Освобождение кадрового буфера */
    framebuffer_release(info);
    /* Освобождение области памяти */
    res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
    release_mem_region(res->start, res->end - res->start + 1);
    platform_set_drvdata(pdev, NULL);
    return 0;
}

/* Структура драйвера платформы */
static struct platform_driver myfb_driver = {
    .probe = myfb_probe,
    .remove = myfb_remove,
    .driver = {
        .name = "myfb",
    },
};

/* Инициализация модуля */
int __init
myfb_init(void)
{
    platform_device_add(&myfb_device);
    return platform_driver_register(&myfb_driver);
}

/* Выгрузка модуля */
void __exit
myfb_exit(void)
{
    platform_driver_unregister(&myfb_driver);
    platform_device_unregister(&myfb_device);
}

module_init(myfb_init);
module_exit(myfb_exit);
```

Консольные драйверы

Консоль является устройством, которое показывает сообщения **printk()**, генерируемые ядром. Если посмотреть на Рисунок 12.5, то можно увидеть, что консольные драйверы лежат в двух ярусах: верхний уровень составляют такие драйверы, как драйвер виртуального терминала, консольный драйвер печати и, например, драйвер консоли **USB_UART** (обсуждается в ближайшее время), а низкий уровень - драйверы, которые несут ответственность за дополнительные операции. Следовательно, есть две основных определённых структуры интерфейса, используемые консольными драйверами. Высокоуровневые консольные драйверы вращаются вокруг структуры **console**, которая определяет такие основные операции, как **setup()** и **write()**. Низкоуровневые драйверы сконцентрированы на структуре **consw**, которая определяет дополнительные операции, такие, как установка свойств курсора, переключение консоли, гашение, изменение размеров и установка информации палитры. Эти структуры определяются в ***include/linux/console.h*** следующим образом:

```
1.struct console {
      char name[8];
      void (*write)(struct console *, const char *, unsigned);
      int (*read)(struct console *, char *, unsigned);
      /* ... */
      void (*unblank)(void);
      int (*setup)(struct console *, char *);
      /* ... */
  };

2.struct consw {
      struct module *owner;
      const char *(*con_startup)(void);
      void (*con_init)(struct vc_data *, int);
      void (*con_deinit)(struct vc_data *);
      void (*con_clear)(struct vc_data *, int, int, int, int);
      void (*con_putc)(struct vc_data *, int, int, int);
      void (*con_putcs)(struct vc_data *,
      const unsigned short *, int, int, int);
      void (*con_cursor)(struct vc_data *, int);
      int (*con_scroll)(struct vc_data *, int, int, int, int);
      /* ... */
  };
```

Как вы уже могли догадаться, глядя на Рисунок 12.5, большинство консольных устройств требуют драйверов обоих уровней, работающих в тандеме. Во многих ситуациях драйвером консоли верхнего уровня является ***vt***. В ПК-совместимых системах, консольный драйвер VGA (***vgacon***) является обычно низкоуровневым консольным драйвером, тогда как на встроенных устройствах низкоуровневым драйвером часто является консольный драйвер кадрового буфер (***fbcon***). Из-за косвенности, предлагаемой абстракцией кадрового буфера, ***fbcon***, в отличие от других низкоуровневых консольных драйверов, является аппаратно-независимым.

Давайте кратко рассмотрим архитектуру обоих уровней консольных драйверов:

- Драйвер верхнего уровня заполняет структуру **console** заданными точками входа и

регистрирует её в ядре с помощью **register_console()**. Отмена регистрации осуществляется с использованием **unregister_console()**. Это драйвер, который взаимодействует с **printk()**. Входные точки, принадлежащие этому драйверу, вызываются службы связанного низкоуровневого драйвера консоли.

- Низкоуровневый драйвер консоли заполняет структуру **consw** заданными точками входа и регистрирует её в ядре с помощью **register_con_driver()**. Отмена регистрации осуществляется с помощью **unregister_con_driver()**. Когда система поддерживает несколько драйверов консоли, чтобы зарегистрировать себя, драйвер может вместо этого вызвать **take_over_console()** и взять на себя обслуживание существующей консоли. **give_up_console()** выполняет обратное. Для обычных дисплеев низкоуровневые драйверы взаимодействуют с консольным драйвером верхнего уровня ***vt*** и символьным (текстовым) драйвером ***vc_screen***, который позволяет получить доступ к памяти консоли.

Некоторым простым консолям, таким, построчно-печатающие устройства и **USB_UART**, обсуждаемые далее, необходим только высокоуровневый драйвер консоли.

Драйвер ***fbcon*** в ядре 2.6 также поддерживает поворот консоли. Дисплейные панели на КПК и мобильных телефонах обычно монтируются в портретной ориентации, в то время как автомобильные панели и IP телефоны являются примерами систем, где дисплейная панель находится в альбомном режиме. Иногда из-за экономических или других факторов встроенные устройства могут потребовать альбомный LCD, смонтированный в режиме портрета, или наоборот. В таких ситуациях удобна поддержка поворота консоли. Поскольку ***fbcon*** аппаратно-независим, реализация поворота консоли также универсальна. Для включения поворота консоли включите при конфигурации ядра **CONFIG_FRAMEBUFFER_CONSOLE_ROTATION** и добавьте в командную строчку ядра **fbcon=rotate:X**, где **X** равно 0 для нормальной ориентации, 1 для поворота на 90 градусов, 2 для поворота на 180 градусов, и 3 для поворота на 270 градусов.

Пример устройства: Снова сотовый телефон

Чтобы узнать, как писать консольные драйверы, давайте вновь обратимся к сотовому телефону с Linux, который мы использовали в Главе 6. Наша задача в этом разделе заключается в разработке драйвера консоли, который работает в мобильном телефоне поверх **USB_UART**. Для удобства на Рисунке 12.7 воспроизводится сотовый телефон из Рисунка 6.5 Главы 6. Давайте напишем консольный драйвер, который выдаёт сообщения **printk()** наружу через **USB_UART**. Сообщения забираются ПК и отображаются для пользователя с помощью сессии, эмулирующей терминал.

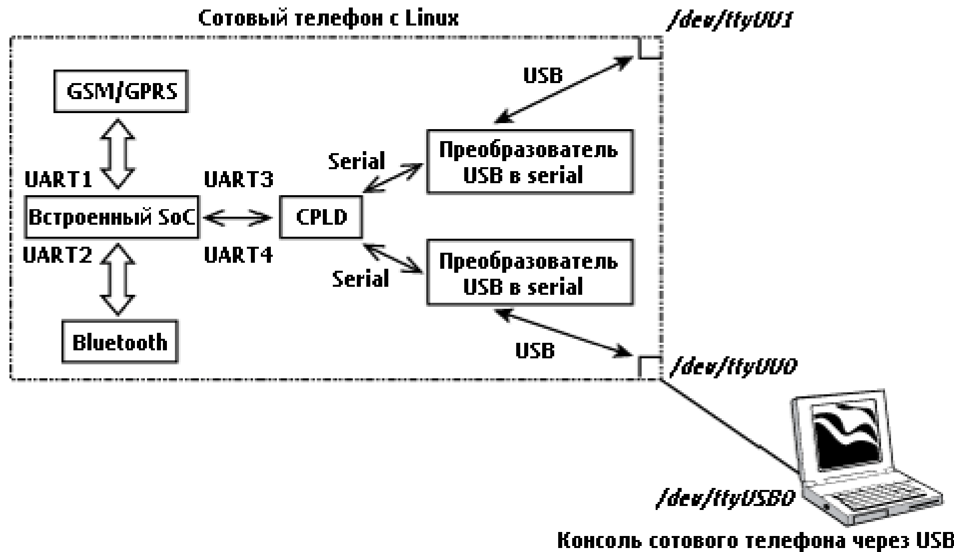

Рисунок 12.7. Консоль через USB_UART.

Распечатка 12.3 содержит реализацию консольного драйвера, который работает через **USB_UART**-ы. Структура **usb_uart_port[]** и некоторые определения, используемые драйвером **USB_UART** в Главе 6, также включены в эту распечатку, чтобы создать полноценный драйвер. Действия драйвера объясняют комментарии распечатки.

Рисунок 12.5 показывает положение нашего примера консольного драйвера **USB_UART** в подсистеме видео в Linux. Как вы можете видеть, **USB_UART** - это простое устройство, которому необходим только высокоуровневый консольный драйвер.

Распечатка 12.3. Консоль через USB_UART

Код:

```
#include <linux/console.h>
#include <linux/serial_core.h>
#include <asm/io.h>

#define USB_UART_PORTS          2 /* Этот сотовый телефон имеет 2 порта
USB_UART */
/* Каждый USB_UART имеет 3-х байтовый набор регистров, состоящий из
   UU_STATUS_REGISTER по смещению 0, UU_READ_DATA_REGISTER по
   смещению 1, и UU_WRITE_DATA_REGISTER по смещению 2, как показано
   Таблице 1 Главы 6, "Драйверы последовательный устройств" */
#define USB_UART1_BASE          0xe8000000 /* Базовый адрес памяти для
USB_UART1 */
#define USB_UART2_BASE          0xe9000000 /* Базовый адрес памяти для
USB_UART2 */
#define USB_UART_REGISTER_SPACE 0x3
/* Назначение битов регистра состояния */
#define USB_UART_TX_FULL        0x20
#define USB_UART_RX_EMPTY       0x10
#define USB_UART_STATUS         0x0F

#define USB_UART1_IRQ           3
#define USB_UART2_IRQ           4
#define USB_UART_CLK_FREQ       16000000
```

```
#define USB_UART_FIFO_SIZE      32

/* Параметры каждого из поддерживаемых портов USB_UART */
static struct uart_port usb_uart_port[] = {
    {
        .mapbase  = (unsigned int)USB_UART1_BASE,
        .iotype   = UPIO_MEM,           /* Отображённая память */
        .irq      = USB_UART1_IRQ,      /* IRQ */
        .uartclk  = USB_UART_CLK_FREQ,  /* Частота в Гц */
        .fifosize = USB_UART_FIFO_SIZE, /* Размер FIFO */
        .flags    = UPF_BOOT_AUTOCONF,  /* Флаг порта UART */
        .line     = 0,                  /* Номер линии UART */
    },
    {
        .mapbase  = (unsigned int)USB_UART2_BASE,
        .iotype   = UPIO_MEM,           /* Отображённая память */
        .irq      = USB_UART2_IRQ,      /* IRQ */
        .uartclk  = USB_UART_CLK_FREQ,  /* Частота в Гц */
        .fifosize = USB_UART_FIFO_SIZE, /* Размер FIFO */
        .flags    = UPF_BOOT_AUTOCONF,  /* Флаг порта UART */
        .line     = 1,                  /* Номер линии UART */
    }
};

/* Запись символа в порт USB_UART */
static void
usb_uart_putc(struct uart_port *port, unsigned char c)
{
    /* Ждём, пока появится место в TX FIFO этого USB_UART.
       Проверяем это опрашивая бит USB_UART_TX_FULL
       регистра состояния */
    while (__raw_readb(port->membase) & USB_UART_TX_FULL);

    /* Записываем символ в порт данных */
    __raw_writeb(c, (port->membase+1));
}

/* Консольная запись */
static void
usb_uart_console_write(struct console *co, const char *s, u_int count)
{
    int i;

    /* Пишем каждый символ */
    for (i = 0; i < count; i++, s++) {
        usb_uart_putc(&usb_uart_port[co->index], *s);
    }
}

/* Получение параметров коммуникации */
static void __init
usb_uart_console_get_options(struct uart_port *port,
                    int *baud, int *parity, int *bits)
{
    /* Читаем текущие настройки (возможно, установленные через bootloader)
```

```
        или возвращаем настройки по умолчанию для чётности, числа бит данных
        и скорости передачи */
    *parity = 'n';
    *bits = 8;
    *baud = 115200;
}

/* Установка параметров консоли для коммуникации */
static int __init
usb_uart_console_setup(struct console *co, char *options)
{
    struct uart_port *port;
    int baud, bits, parity, flow;

    /* Проверяем номер порта и получаем индекс
       соответствующей структуры */
    if (co->index == -1 || co->index >= USB_UART_PORTS) {
        co->index = 0;
    }
    port = &usb_uart_port[co->index];

    /* Используем функции, предоставляемые уровнем serial для обработки
параметров */
    if (options) {
        uart_parse_options(options, &baud, &parity, &bits, &flow);
    } else {
        usb_uart_console_get_options(port, &baud, &parity, &bits);
    }
    return uart_set_options(port, co, baud, parity, bits, flow);
}

/* Заполняем структуру консоли */
static struct console usb_uart_console = {
    .name   = "ttyUU",                 /* Имя консоли */
    .write  = usb_uart_console_write, /* Как делать printk в консоли */
    .device = uart_console_device,    /* Предоставлена ядром serial */
    .setup  = usb_uart_console_setup, /* Как настроить консоль */
    .flags  = CON_PRINTBUFFER,         /* Флаг по умолчанию */
    .index  = -1,                      /* Инициализация в неправильное значение
*/
};

/* Инициализация консоли */
static int __init
usb_uart_console_init(void)
{
/* ... */
/* Регистрация этой консоли */
register_console(&usb_uart_console);
return 0;
}

console_initcall(usb_uart_console_init); /* Метка инициализации консоли */
```

После того, как драйвер будет собран как часть ядра, вы сможете активировать его с

помощью добавления в командную строку ядра **console=ttyUUX** (где **X** равен 0 или 1).

Логотип при загрузке

Популярной функцией, предлагаемой подсистемой кадрового буфера является отображение при загрузке логотипа. Для отображения логотипа при конфигурации ядра включите **CONFIG_LOGO** и выберите доступный логотип. Вы также можете добавить своё изображение логотипа в каталог ***drivers/video/logo/***.

Широко используемым форматом изображения для логотипа загрузки является ***CLUT224***, который поддерживает 224 цветов. Работа в таком формате похожа на псевдо-палитры, описанные в разделе "Цветовые режимы" 39. Изображение в CLUT224 является файлом языка Си, содержащим две структуры:

- CLUT (***Color Look Up Table, Справочная таблица цветов***), которая является символьным массивом из 224 наборов RGB (тем самым имеет размер 224*3 байт). Каждый 3-х байтовый элемент CLUT представляет собой комбинацию красного, зеленого и синего цветов.

- Массив данных, каждый байт которого является индексом в CLUT. Индексы имеют значения от 32 до 255 (поддерживая тем самым 224 цветов). Индекс 32 относится к первому элементу в CLUT. Код работы с логотипом (в ***drivers/video/fbmem.c***) создаёт пиксельные данные кадрового буфера из наборов CLUT, соответствующих каждому индексу в массиве данных. Визуальное отображение информации осуществляется с использованием метода низкоуровневого драйвера кадрового буфера **fb_imageblit()**, как рассказывалось в разделе "Методы ускорения" 40.

Другими поддерживаемыми форматами логотипа являются 16-ти цветный ***vga16*** и черно-белый ***mono***. Скрипты для конвертации стандартных файлов в ***Переносимую пиксельную карту, Portable Pixel Map*** (PPM) доступны на верхнем уровне каталога ***scripts/***.

Если устройство кадрового буфера также является консолью, под логотипом прокручиваются сообщения при загрузке. Вы можете предпочесть отключить консольные сообщения на системе, готовой к поставке (добавив в командную строку ядра **console=/dev/null**), и показывать в качестве загрузочного логотипа собственно подготовленное изображение "заставка" в формата CLUT224.

Отладка

Драйвер виртуального кадрового буфера, включаемый установкой в меню настройки **CONFIG_FB_VIRTUAL**, работает поверх адаптера псевдо-графики. Вы можете использовать помощь этого драйвера для отладки подсистемы кадрового буфера. Некоторые драйверы кадрового буфера, такие, как **intelfb**, предлагают дополнительные опции конфигурации, которые могут позволить генерировать зависимую от драйвера отладочную информацию.

Для обсуждения вопросов, связанных с драйверами кадрового буфера, подпишитесь на рассылку linux-fbdev-devel, https://lists.sourceforge.net/lists/listinfo/linux-fbdev-devel/.

Отладка драйверов консоли - не простая работа, потому что вы не можете вызвать изнутри драйвера **printk()**. Если у вас есть запасные консольные устройства, такие как последовательный порт, вы можете реализовать сначала ваш драйвер консоли в виде UART/tty (как мы это делали в Главе 6 для устройства **USB_UART**, используемого в этой главе) и отлаживать такой драйвер работая с ***/dev/tty*** и печатая сообщения на запасную консоль. Затем можно переупаковать отлаженные куски кода в виде консольного драйвера.

Где искать информацию

Уровень ядра кадрового буфера и низкоуровневые драйверы кадрового буфера находятся в каталоге ***drivers/video/***. Общие структуры кадрового буфера определяются в ***include/linux/fb.h***, а заголовки, зависимые от набора микросхем, находятся внутри ***include/video/***. Драйвер ***fbmem***, ***drivers/video/fbmem.c***, создаёт символьные устройства ***/dev/fbX*** и внешний интерфейс для обработки команд ioctl кадрового буфера, выдаваемых пользовательскими приложениями.

Драйвер ***intelfb***, ***drivers/video/intelfb/****, является низкоуровневым драйвером кадрового буфера для нескольких графических контроллеров Intel, таких, как Северный мост, интегрированный с 855 GME. Драйвер ***radeonfb***, ***drivers/video/aty/****, является драйвером кадрового буфера для графических карт Radeon Mobility AGP от ATI technologies. Все исходные файлы вида ***drivers/video/*fb.c*** являются драйверами кадрового буфера для графических контроллеров, в том числе тех, которые интегрированы в некоторые микропроцессоры. Если вы пишете собственный низкоуровневый драйвер кадрового буфера, вы можете использовать в качестве отправной точки ***drivers/video/skeletonfb.c***. Для получения дополнительной документации об уровне кадрового буфера смотрите ***Documentation/fb/****.

Главной страницей проекта кадрового буфера Linux является www.linux-fbdev.org. Этот веб-сайт содержит разделы HOWTO (Как сделать), ссылки на драйверы кадрового буфера и утилиты, а также ссылки на соответствующие веб-страницы.

Консольные драйверы, и на основе кадрового буфера, и другие, находятся внутри ***drivers/video/console/***. Чтобы узнать, как printk() выводит сообщения ядра во внутренний буфер и вызывает консольные драйверы, смотрите ***kernel/printk.c***. Таблица 12.2 содержит основные структуры данных, используемые в этой главе и их расположение в дереве исходных текстов. В Таблице 12.3 перечислены основные программные интерфейсы ядра, которые вы использовали в этой главе вместе с расположением их определений.

Таблица 12.2. Список структур данных

Структура данных	Местоположение	Описание
fb_info	***include/linux/fb.h***	Центральная структура данных, используемая низкоуровневыми драйверами кадрового буфера
fb_ops	***include/linux/fb.h***	Содержит адреса всех точек входа, предоставляемых низкоуровневыми драйверами кадрового буфера
fb_var_screen_info	***include/linux/fb.h***	Содержит изменяемую информацию, передаваемую видео оборудованию, такую, как разрешение по X, разрешение по Y и длительности HYSNC/VSYNC
fb_fix_screen_info	***include/linux/fb.h***	Неизменяемая информация от

		видео оборудования, такая, как начальный адрес кадрового буфера
fb_cmap	*include/linux/fb.h*	Цветовая таблица RGB для устройства с кадровым буфером
console	*include/linux/console.h*	Представление высокоуровневого консольного драйвера
consw	*include/linux/console.h*	Представление низкоуровневого консольного драйвера

Таблица 12.3. Список программных интерфейсов ядра

Интерфейс ядра	Местоположение	Описание
register_framebuffer()	*drivers/video/fbmem.c*	Регистрирует низкоуровневое устройство с кадровым буфером.
unregister_framebuffer()	*drivers/video/fbmem.c*	Отменяет регистрацию низкоуровневое устройство с кадровым буфером.
framebuffer_alloc()	*drivers/video/fbsysfs.c*	Выделяет память для структуры **fb_info**.
framebuffer_release()	*drivers/video/fbsysfs.c*	Обратна к **framebuffer_alloc ()**.
fb_alloc_cmap()	*drivers/video/fbcmap.c*	Выделяет память для карты цветов.
fb_dealloc_cmap()	*drivers/video/fbcmap.c*	Освобождает память карты цветов.
dma_alloc_coherent()	*include/asmgeneric/dma-mapping.h*	Выделяет и отображает согласованный буфер DMA. Смотрите **pci_alloc_consistent()** в Главе 10.
dma_free_coherent()	*include/asmgeneric/dma-mapping.h*	Освобождает согласованный буфер DMA. Смотрите **pci_free_consistent()** в Главе 10.
register_console()	*kernel/printk.c*	Регистрирует высокоуровневый консольный драйвер.
unregister_console()	*kernel/printk.c*	Отменяет регистрацию

		высокоуровневого консольного драйвера.
register_console() **take_over_console()**	***drivers/char/vt.c***	Регистрирует/подключает низкоуровневый консольный драйвер.
unregister_console() **give_up_console()**	***drivers/char/vt.c***	Разрегистрирует/отключает низкоуровневый консольный драйвер.

Глава 13. Драйверы Звука

В этой главе

- Звуковая архитектура 59
- Звуковая подсистема Linux 61
- Пример устройства: MP3 плеер 64
- Отладка 78
- Где искать информацию 79

Аудио оборудование предоставляет компьютерным системам возможность генерировать и записывать звук. Звук является неотъемлемым компонентом и в ПК и во встраиваемых системах, для общения на ноутбуке, звонка с мобильного телефона, прослушивания MP3 плеера, потокового мультимедиа от приставки, или объявления инструкций в системе медицинского назначения. Если вы запускаете Linux на любом из этих устройств, вы нуждаетесь в услугах, предлагаемых звуковой подсистемой Linux.

Давайте выясним в этой главе, как ядро поддерживает аудио контроллеры и кодеки. Давайте изучим архитектуру звуковой подсистемы Linux и модель программирования, которую она экспортирует.

Звуковая архитектура

Рисунок 13.1 показывает подключение звука на ПК-совместимой системе. Аудио контроллер Южного моста, а также внешний кодек, подключённый к аналоговой звуковой схеме.

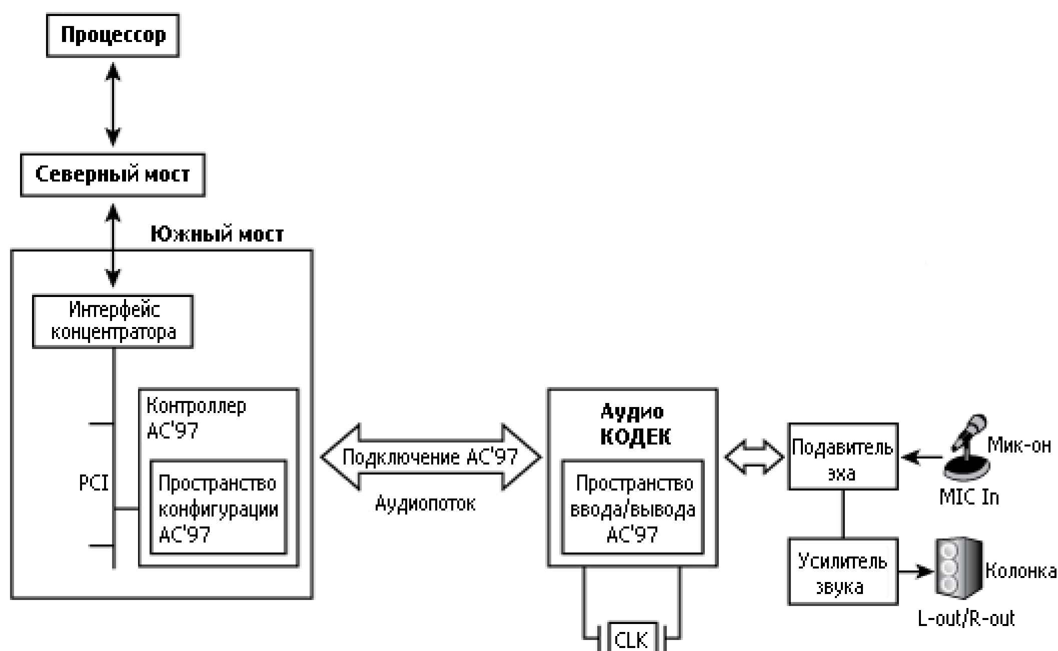

Рисунок 13.1. Звук в среде ПК.

Аудио кодек преобразует данные цифрового звука в аналоговые звуковые сигналы для воспроизведения через динамики и выполняет обратную операцию для записи через микрофон. Другие обычные звуковые входы и выходы, которые взаимодействуют с кодеком, включают гарнитуры, наушники, телефоны, громкую связь, линейный вход и линейный выход. Кодек также предлагает функциональность ***микшера***, который подключён к комбинации этих аудио входов и выходов, а также управляет уровнем громкости соответствующих звуковых сигналов.

Это определение микшера с точки зрения программного обеспечения. *Микширование звука* или *микширование данных* относится к способности некоторых кодеков смешивать несколько звуковых потоков и создавать единый поток. Это необходимо, например, если вы хотите наложить объявление во время ведения голосового общения по IP телефону. Библиотека alsa-lib, которая обсуждается в последней части этой главы, поддерживает

подключаемый модуль, называющийся *dmix*, который выполняет микширование данных программным образом, если кодек не способен выполнять эту операцию на аппаратном уровне.

Цифровые аудио данные получаются путём измерения аналоговых звуковых сигналов с определёнными частотами, с использованием техники, названной ***импульсно-кодовой модуляцией, pulse code modulation*** (PCM). Качеством компакт диска, например, является звук с частотой дискретизации 44.1 кГц, использующий 16 бит для каждого отсчёта. Кодек отвечает за записи звука путём дискретизации на поддерживаемых PCM скоростях передачи и воспроизведение звука, изначально дискретизированного с различными скоростями PCM.

Звуковая карта может поддерживать один или несколько кодеков. Каждый кодек, в свою очередь, поддерживает один или более аудио подпотоков в моно или стерео.

Примерами стандартных интерфейсов подключения звуковых контроллеров являются кодеки Audio Codec'97 (AC'97) и шина Inter-IC Sound (I2S):

- Спецификация Intel AC'97, доступная на http://download.intel.com/, определяет семантику и адреса аудио-регистров. Регистры конфигурации являются частью звукового контроллера, а пространство регистров ввода/вывода находится внутри этого кодека. Запросы для работы с регистрами ввода/вывода пересылаются аудио контроллером кодеку через соединение AC'97. Например, регистр, который управляет громкостью входной линии, находится в пространстве ввода/вывода AC'97 по смещению **0x10**. Система ПК на Рисунке 13.1 для связи с внешним кодеком использует AC'97.

- Спецификация I2S, доступная на www.nxp.com/acrobat_download/various/I2SBUS.pdf, является стандартным интерфейсом кодека, разработанным Philips. Встроенное устройство, показанное на Рисунке 13.2, для передачи аудио данных в кодек использует I2S. Программирование регистров ввода/вывода кодека осуществляется через шину I2C.

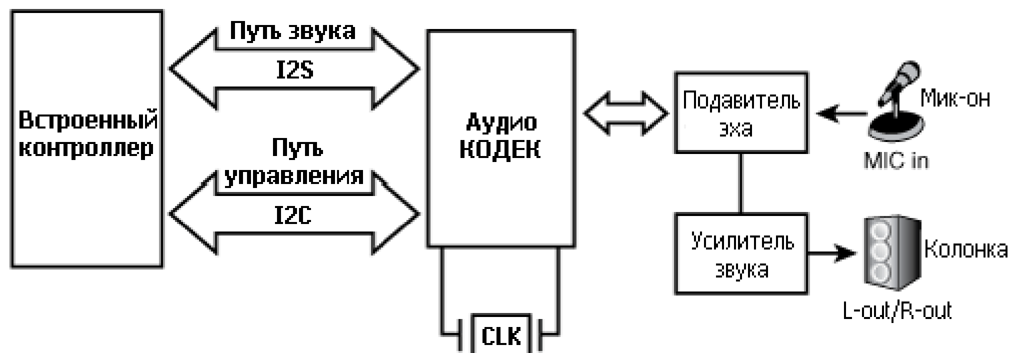

Рисунок 13.2. Подключение звука на встроенной системе

AC'97 имеет ограничения, касающиеся количества поддерживаемых каналов и скоростей передачи. Последние наборы микросхем Южного моста от Intel поддерживают новую технологию под названием High Definition (HD) Audio (Звук высокого качества), которая предлагает высококачественный, объёмный звук и возможности многопоточности.

Звуковая подсистема Linux

Улучшенная Архитектура Звука Linux, Advanced Linux Sound Architecture (ALSA) является звуковой подсистемой, выбранной для ядра версии 2.6. ***Открытая Звуковая Система, Open Sound System*** (OSS), звуковой уровень в ядре версии 2.4, в настоящее время устарел и не рекомендуется к использованию. Для перехода от OSS к ALSA последняя предоставляет эмуляцию OSS, которая позволяет приложениям, использующим API OSS, запускаться без изменений на ALSA. Звуковые ядра Linux, такие как ALSA и OSS, делают аудио приложения независимыми от базового оборудования, тогда как стандарты кодеков, такие как AC'97 и I2S, избавляют от необходимости написания отдельных звуковых драйверов для каждой звуковой карты.

Чтобы понять архитектуру звуковой подсистемы Linux посмотрите на Рисунок 13.3. Основными частями подсистемы являются:

- Звуковое ядро, которое является базовым кодом, состоящим из процедур и структур, доступных другим компонентам звукового уровня Linux. Как и уровни ядра, принадлежащие другим драйверным подсистемам, звуковое ядро обеспечивает уровень косвенности, что делает каждый компонент в звуковой подсистеме не зависящим от других. Ядро играет важную роль в экспорте API ALSA вышележащим приложениям. Узлами ***/dev/snd/****, показанными на Рисунке 13.3, которые создаются и управляются из ядром ALSA, являются: ***/dev/snd/controlC0*** - узел управления (используемый в приложениях для управления уровнем громкости и тому подобному), ***/dev/snd/pcmC0D0p*** - устройство воспроизведения (***p*** в конце имени устройства означает playback, воспроизведение), и ***/dev/snd/pcmC0D0c*** - записывающее устройство (***c*** в конце имени устройства означает capture, захват). В этих именах устройств целое число после ***C*** является номером карты, а после ***D*** - номером устройства. ALSA драйвер для карты, которая имеет голосовой кодек для телефонии и стерео кодек для музыки, может экспортировать ***/dev/snd/pcmC0D0p*** для чтения аудио потоков, предназначенный для первого, и ***/dev/snd/pcmC0D1p*** для качественного музыкального канала для последнего.

- Драйверы аудио контроллера зависят от оборудования контроллера. Например, для управления аудио контроллером, находящимся в Южном мосте Intel ICH, используется драйвер ***snd_intel8x0***.

- Интерфейсы аудиокодеков, которые помогают взаимодействию между контроллерами и кодеками. Для кодеков AC'97 используйте ***snd_ac97_codec*** и модули ***ac97_bus***.

- Уровень эмуляции OSS, который выступает в качестве посредника между приложениями, использующими OSS, и ядром с поддерживающим ALSA. Этот уровень экспортирует узлы ***/dev***, изображающие поддержку уровня OSS в ядре версии 2.4. Эти узлы, такие как ***/dev/dsp***, ***/dev/adsp*** и ***/dev/mixer***, позволяют приложениям OSS работать поверх ALSA без изменений. Узел OSS ***/dev/dsp*** связан с узлами ALSA ***/dev/snd/pcmC0D0****, ***/dev/adsp*** соответствует ***/dev/snd/pcmC0D1****, а ***/dev/mixer*** связан с ***/dev/snd/controlC0***.

- Интерфейс procfs and sysfs для доступа к информации через ***/proc/asound/*** и ***/sys/class/sound/***.

- Библиотека ALSA пользовательского пространства, ***alsa-lib***, которая предоставляет объект ***libasound.so***. Эта библиотека упрощает работу программиста приложения ALSA, предлагая несколько готовых процедур для доступа к драйверам ALSA.

- Пакет ***alsa-utils***, который включает в себя такие утилиты, как ***alsamixer***, ***amixer***, ***alsactl*** и ***aplay***. ***alsamixer*** или ***mixer*** используются для изменения громкости звуковых сигналов, таких как линейный вход, линейный выход или микрофон, а ***alsactl*** - для управления параметрами драйверов ALSA. ***aplay*** используется для воспроизведения звука через ALSA.

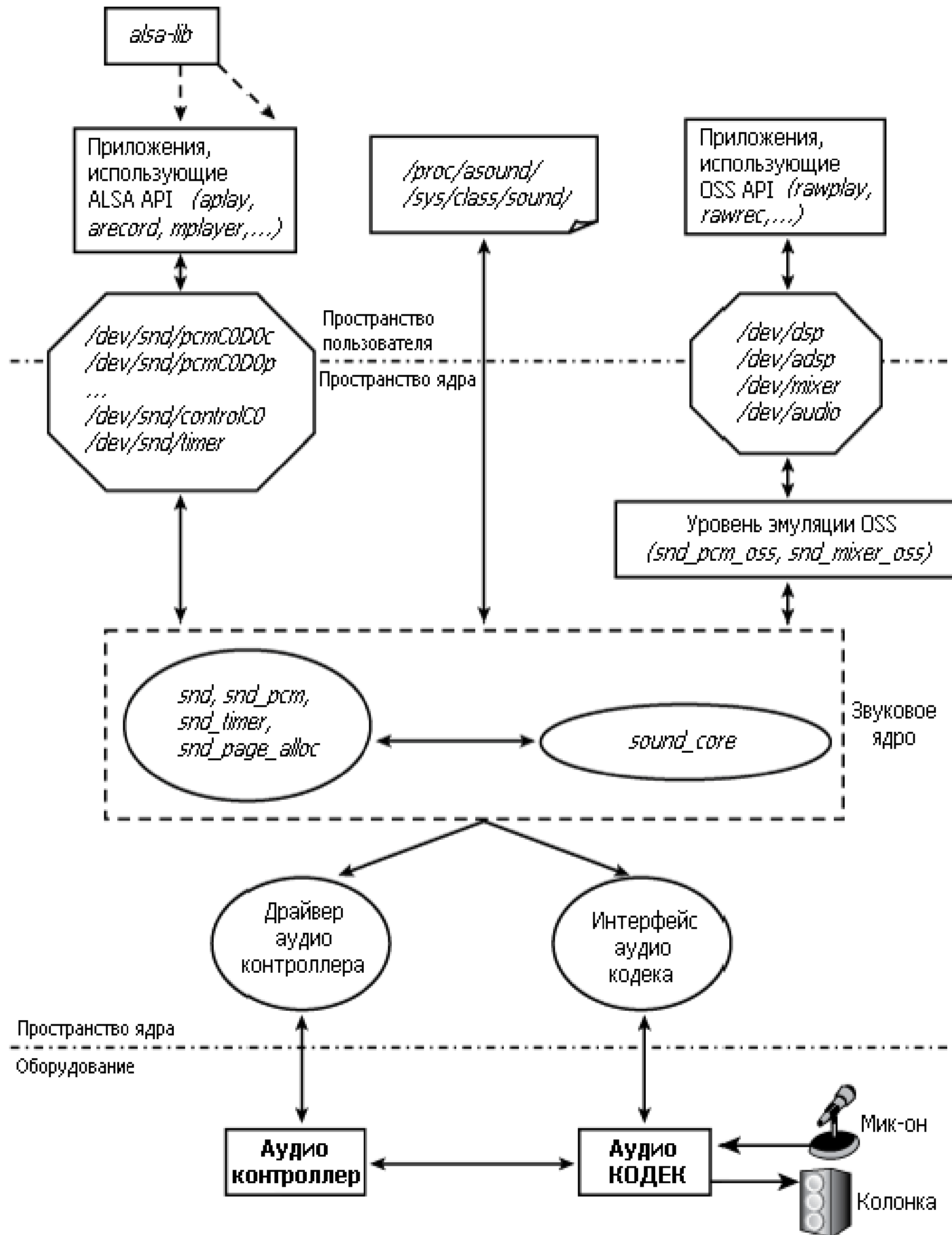

Рисунок 13.3. Звуковая подсистема Linux (ALSA).

Чтобы получить более полное представление об архитектуре звукового уровня Linux, давайте посмотрим на драйверные модули ALSA, работающие на ноутбуке, в тандеме с Рисунком 13.3 (→ используется для добавления комментариев):

Код:

```
bash> lsmod| grep snd
snd_intel8x0      33148  0                                         →Драйвер
аудио контроллера
snd_ac97_codec    92000  1 snd_intel8x0                            →Интерфейс
аудио кодека
ac97_bus          3104   1 snd_ac97_codec                          →Шина
аудио кодека
snd_pcm_oss       40512  0                                         →Эмуляция
OSS
snd_mixer_oss     16640  1 snd_pcm_oss
→Управление громкостью OSS
snd_pcm           73316  3 snd_intel8x0,snd_ac97_codec,snd_pcm_oss  →Уровень
ядра
snd_timer         22148  1 snd_pcm                                 →Уровень
ядра
snd               50820  6 snd_intel8x0,snd_ac97_codec,snd_pcm_oss, →Уровень
ядра
                           snd_mixer_oss,snd_pcm,snd_timer
soundcore         8960   1 snd                                     →Уровень
ядра
snd_page_alloc    10344  2 snd_intel8x0,snd_pcm                    →Уровень
ядра
```

Пример устройства: MP3 плеер

Рисунок 13.4 показывает операции, необходимые для проигрывания звука, на примере управляемого по Bluetooth MP3 плеера Linux, построенного на встроенном микропроцессоре. Вы можете запрограммировать сотовый телефон на Linux (который мы использовали в Главе 6, "Драйверы последовательных портов", и в Главе 12, "Драйверы Видео" 24), чтобы загрузить песни из интернета ночью, когда телефонные тарифы предположительно дешевле, и загрузить их на ***Compact Flash*** (CF) диск MP3 плеера через Bluetooth, чтобы можно было послушать песни на следующий день во время поездки до офиса.

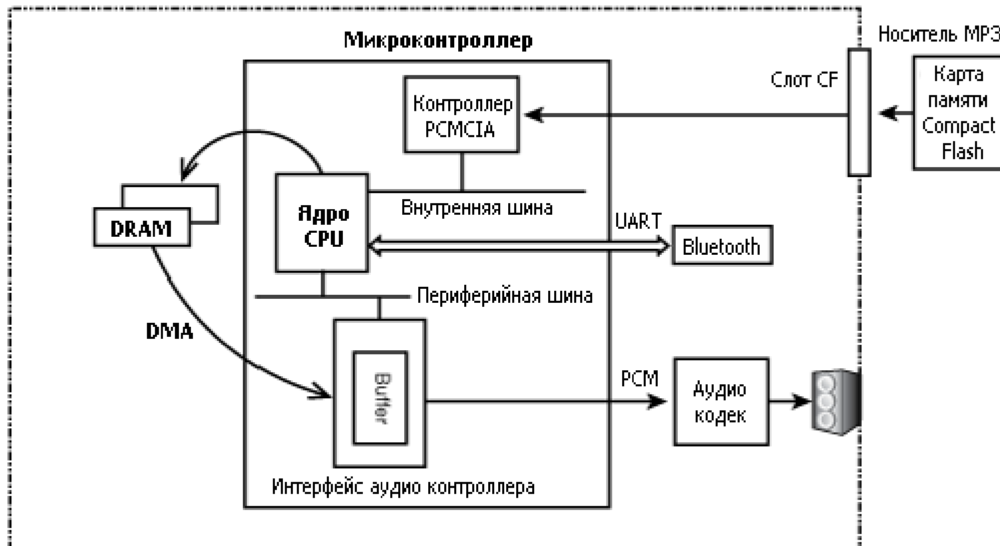

Рисунок 13.4. Звук в MP3 плеере на Linux.

Наша задача заключается в разработке звукового программного обеспечения для этого устройства. Приложение на плеере считывает песни с CF диска и декодирует их в системной памяти. Драйвер ALSA в ядре собирает музыкальные данные из системной памяти и отправляет их для передачи буферов, которые являются частью звукового контроллера процессора. Эти данные PCM направляются в кодек, который проигрывает музыку через динамик устройства. Как и в случае навигационной системы, которая обсуждалась в предыдущей главе, мы будем считать, что Linux поддерживает этот процессор, и что все архитектурно-зависимые сервисы, такие как DMA, поддерживаются ядром.

Таким образом, звуковое программное обеспечение для этого MP3 плеера состоит из двух частей:

1. Пользовательская программа, которая декодирует MP3 файлы, читаемые с CF диска, и преобразует их в простой PCM. Чтобы написать "родное" приложение-декодер ALSA, можно использовать вспомогательные процедуры, предлагаемые библиотекой ***alsa-lib***. В разделе "Программирование ALSA" 75 показано, как ALSA приложения взаимодействуют с драйверами ALSA.

   Для создания такого устройства у вас также есть возможность использовать общедоступные MP3 плееры, таких как ***madplay*** (http://sourceforge.net/projects/mad/).

2. Низкоуровневый аудио ALSA драйвер ядра. Следующий раздел посвящён написанию этого драйвера.

Одна из возможных аппаратных реализаций устройства показана на Рисунке 13.4 и заключается в использовании PowerPC 405LP SoC и аудио кодека Texas Instruments TLV320. Процессорным ядром в этом случае является процессор 405, а встроенным интерфейсом аудио контроллера является ***Codec Serial Interface, последовательный интерфейс кодека*** (CSI). Микроконтроллеры обычно имеют высокопроизводительную внутреннюю шину, которая соединяет такие контроллеры, как DRAM и видео, и отдельную встроенную периферийную шину для связи с низкоскоростными периферийными устройствами, такими как последовательные порты, I2C и GPIO. В случае 405LP, первая называется ***Локальная шина процессора, Processor Local Bus*** (PLB), а вторая называется ***Встроенная периферийная шина, Onchip Peripheral Bus*** (OPB). Контроллер PCMCIA/CF подключён к PLB, а интерфейс аудио контроллера подключён к OPB.

Аудио драйвер состоит из трёх основных компонентов:

1. Процедуры, которые занимаются воспроизведением

2. Процедуры, которые занимаются захватом звука

3. Функции управления микшированием

Наш драйвер реализует воспроизведение, но не поддерживает запись, так как MP3 плеер в данном примере не имеет микрофона. Драйвер также упрощает функцию микширования. Вместо того, чтобы предлагать полный комплекс для управления уровнем громкости, то есть громкоговорителем, наушниками и линейным выходом, он позволяет использовать только один общий регулятор громкости.

Назначение регистров аудиооборудования MP3 плеера приведено в Таблице 13.1 и отражает эти допущения и упрощения, и не соответствует стандартам, таким как упоминавшийся ранее AC'97. Таким образом, кодек имеет **SAMPLING_RATE_REGISTER** для настройки частота дискретизации воспроизведения (цифро-аналогового преобразования), но не имеет регистров для настройки частоты записи (аналого-цифрового преобразования). **VOLUME_REGISTER** настраивает одну общую громкость.

Таблица 13.1. Назначение регистров аудио оборудования, показанного на Рисунке 13.4

Имя регистра	Используется для конфигурации
VOLUME_REGISTER	Управляет общей громкостью кодека
SAMPLING_RATE_REGISTER	Устанавливает частоту дискретизации цифро-аналогового преобразования.
CLOCK_INPUT_REGISTER	Конфигурирует частоты, делители кодека и так далее.
CONTROL_REGISTER	Разрешает прерывания, конфигурирует причину прерывания (например, завершение передачи буфера), перезапускает оборудование, включает/выключает управление на шине и так далее.
STATUS_REGISTER	Состояние аудио событий кодека.
DMA_ADDRESS_REGISTER	Этот пример оборудования поддерживает один

	дескриптор буфера DMA. Реальные карты могут поддерживать несколько дескрипторов и, возможно, дополнительные регистры для хранения таких параметров, как дескриптор, который в настоящее время обрабатывается, позиция текущего сэмпла внутри буфера и так далее. DMA выполняется для буферов в аудио контроллере, так что этот регистр находится в пространстве памяти контроллера.
DMA_SIZE_REGISTER	Хранит размер передач DMA к/от процессора.

Распечатка 13.1 является скелетом ALSA аудио драйвера для MP3 плеера и щедро использует псевдо-код (в комментариях), чтобы убрать посторонние детали. ALSA является сложным ядром и соответствующие аудио драйверы обычно имеют несколько тысяч строк. В Распечатке 13.1 вы только знакомитесь с аудио драйвером. Для продолжения обучения стоит углубиться в исходные коды Linux внутри каталога верхнего уровня ***sound/***.

Методы и структуры драйвера

В нашем примере драйвер реализован в виде драйвера платформы. Давайте рассмотрим шаги, выполняемые методом **probe()** драйвера платформы, **mycard_audio_probe()**. Мы будем немного отвлекаться на каждом шагу, чтобы объяснить соответствующие концепции и важные структуры данных, с которыми сталкиваемся, и это приведёт нас к другим частям драйвера и поможет связать всё воедино.

mycard_audio_probe() выполняет следующие действия:

1. Создаёт экземпляр звуковой карты, вызывая **snd_card_new()**:

```
struct snd_card *card = snd_card_new(-1, id[dev->id], THIS_MODULE, 0);
```

   Первым аргументом **snd_card_new()** является индекс карты (который идентифицирует эту карту среди нескольких звуковых карт системы), вторым аргументом является идентификатор, который будет храниться в поле **id**, возвращённой структуры **snd_card**, третьим аргументом является владелец модуля, а последним аргументом является размер поля собственных данных, которое будет предоставляться через поле **private_data** возвращаемой структуры **snd_card** (обычно для хранения данных, зависящих от микросхемы, таких как уровни прерывания, адреса ввода/вывода).

   snd_card представляет созданную звуковую карту и в ***include/sound/core.h*** определяется следующим образом:

```
struct snd_card {
    int number;             /* Индекс карты */
    char id[16];            /* ID карты */
    /* ... */
    struct module *module; /* Владелец модуля */
    void *private_data;     /* Внутренние данные */
    /* ... */
    struct list_head controls;/* Список управления для этой карты */
    struct device *dev;     /* Устройство, связанное с этой картой */
    /* ... */
```

```
};
```

remove(), обратный методу **probe**, **mycard_audio_remove()**, отключает **snd_card** от ядра ALSA с помощью **snd_card_free()**.

2. Создаёт экземпляр объекта для воспроизведения PCM и связывает его с картой, созданной на Шаге 1, используя **snd_pcm_new()**:

```
int snd_pcm_new(struct snd_card *card, char *id,
                int device,
                int playback_count, int capture_count,
                struct snd_pcm **pcm);
```

Аргументами являются, соответственно, экземпляр звуковой карты, созданной на Шаге 1, строка идентификации, индекс устройства, количество поддерживаемых потоков воспроизведения, количество поддерживаемых потоков захвата (0 в нашем примере) и указатель для сохранения созданного экземпляра PCM. Созданный экземпляр PCM определяется в ***include/sound/pcm.h*** следующим образом:

Код:
```
struct snd_pcm {
struct snd_card *card;         /* Связанная snd_card */
/* ... */
struct snd_pcm_str streams[2]; /* Потоки воспроизведения и захвата этого
компонента
                                  PCM. Каждый поток может поддерживать
подпотоки,
                                  если ваше оборудование их поддерживает */
/* ... */
struct device *dev;            /* Связанное аппаратное устройство */
};
```

Процедура **snd_device_new()** лежит в основе **snd_pcm_new()**, как и другие подобные функции создания экземпляра компонента. **snd_device_new()** связывает компонент и набор операций с указанной **snd_card** (смотрите Шаг 3).

3. Подключает операции воспроизведения к экземпляру PCM, созданному на Шаге 2, вызывая **snd_pcm_set_ops()**. Структура **snd_pcm_ops** определяет эти операции для передачи PCM звука в кодек. Распечатка 13.1 выполняет это следующим образом:

Код:
```
/* Операторы для воспроизведения потока PCM */
static struct snd_pcm_ops mycard_playback_ops = {
    .open      = mycard_pb_open,    /* Открытие */
    .close     = mycard_pb_close,   /* Закрытие */
    .ioctl     = snd_pcm_lib_ioctl,/* Используйте для обработки специальных
команд,
                                      в противном случае укажите общий
обработчик
                                      ioctl, snd_pcm_lib_ioctl() */
    .hw_params = mycard_hw_params, /* Вызывается, когда вышележащие уровни
устанавливают
                                      параметры оборудования,такие как формат
```

```
звука.
                                            Выделение буфера DMA также выполняется
здесь */
    .hw_free    = mycard_hw_free,    /* Освобождение ресурсов, выделенных в
                                            mycard_hw_params() */
    .prepare    = mycard_pb_prepare,/* Подготовка к передаче потока звука.
                                            Устанавливает формат звука, например,
S16_LE
                                            (обсуждается позже), разрешает
прерывания,... */
    .trigger    = mycard_pb_trigger,/* Вызывается, когда движок PCM стартует,
                                            останавливается, или делает паузу.
Второй аргумент
                                            указывает причину вызова. Эта функция
                                            не может спать */
};

/* Связывание операций с экземпляром PCM */
snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK, &mycard_playback_ops);
```

В Распечатке 13.1 **mycard_pb_prepare()** настраивает частоту дискретизации в **SAMPLING_RATE_REGISTER**, частоту источника в **CLOCK_INPUT_REGISTER** и передаёт полное разрешение прерывания в **CONTROL_REGISTER**. Метод **trigger()**, **mycard_pb_trigger()**, отображает на лету аудио буфер, полученный от ядра ALSA, используя **dma_map_single()**. (Мы обсудили потоковый DMA в Главе 10, "Подключение компонентов периферии".) Адрес отображённого буфера DMA запрограммирован в **DMA_ADDRESS_REGISTER**. Этот регистр является частью звукового контроллера процессора, в отличие от предыдущих регистров, которые находятся внутри кодека. Звуковой контроллер направляет данные DMA в кодек для воспроизведения.

Другим связанным объектом является структура **snd_pcm_hardware**, которая объявляет аппаратные возможности компонента PCM. В нашем примере устройства она определена в Распечатке 13.1 следующим образом:

Код:
```
/* Возможности оборудования для проигрывания PCM потока */
static struct snd_pcm_hardware mycard_playback_stereo = {
    .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_PAUSE |
             SNDRV_PCM_INFO_RESUME);    /* поддерживается mmap(). Поток имеет
возможности
                                          для паузы/продолжения
воспроизведения */
    .formats = SNDRV_PCM_FMTBIT_S16_LE,/* 16 бит со знаком на канал, сначала
младший */
    .rates = SNDRV_PCM_RATE_8000_48000,/* Диапазон частот дискретизации ЦАП */
    .rate_min = 8000,                 /* Минимальная частота дискретизации */
    .rate_max = 48000,                /* Максимальная частота дискретизации
*/
    .channels_min = 2,                /* Поддерживается левый и правый канал
*/
    .channels_max = 2,                /* Поддерживается левый и правый канал
*/
    .buffer_bytes_max = 32768,        /* Максимальный размер буфера */
```

```
};
```

Этот объект связан с соответствующей **snd_pcm** из оператора **open()**, **mycard_playback_open()**, с помощью созданного во время выполнения экземпляра PCM. Каждый открытый поток PCM имеет объект выполнения, называемый **snd_pcm_runtime**, который содержит всю информацию, необходимую для управления этим потоком. Это гигантская структура конфигурации программного обеспечения и оборудования определена в ***include/sound/pcm.h*** и содержит в качестве одного из своих полей **snd_pcm_hardware**.

4. Заранее выделяет память для буферов с использованием **snd_pcm_lib_preallocate_pages_for_all()**. В дальнейшем **mycard_hw_params()** получает из этой заранее созданной области буферы DMA с помощью **snd_pcm_lib_malloc_pages()** и сохраняет их в созданном во время работы экземпляре PCM, о котором рассказывалось на Шаге 3. **mycard_pb_trigger()** подключает этот буфер к DMA во время запуска операции PCM и отключает его при остановке PCM операции.

   Связывает элемент управления микшером звуковой карты для глобального контроля громкости с помощью **snd_ctl_add()**:

```
snd_ctl_add(card, snd_ctl_new1(&mycard_playback_vol, &myctl_private));
```

   snd_ctl_new1() принимает структуру **snd_kcontrol_new** в качестве первого аргумента и возвращает указатель на структуру **snd_kcontrol**. Распечатка 13.1 определяет её следующим образом:

```
static struct snd_kcontrol_new mycard_playback_vol = {
    .iface = SNDRV_CTL_ELEM_IFACE_MIXER,/* Элементом управления является тип
MIXER */
    .name  = "MP3 volume",             /* Имя */
    .index = 0,                        /* Номер кодека: 0 */
    .info  = mycard_pb_vol_info,       /* Информация о громкости */
    .get   = mycard_pb_vol_get,        /* Получение громкости */
    .put   = mycard_pb_vol_put,        /* Установка громкости */
};
```

   Структура **snd_kcontrol** описывает элемент управления. Наш драйвер использует его в качестве ручки общего регулятора громкости. **snd_ctl_add()** регистрирует элемент **snd_kcontrol** в ядре ALSA. Указанные методы управления вызываются, когда выполняются такие пользовательские приложения, как ***alsamixer***. В Распечатке 13.1 метод **put()** **snd_kcontrol**, **mycard_playback_volume_put()**, записывает запрошенные настройки громкости в **VOLUME_REGISTER** кодека.

5. И, наконец, регистрирует звуковую карту в ядре ALSA:

```
snd_card_register(card);
```

codec_write_reg() (используется, но оставлена нереализованной в Распечатке 13.1) пишет значения в регистры кодека, общаясь через шину, которая соединяет аудио контроллер в процессоре с внешним кодеком. Если, например, шинным протоколом является

I²C или SPI, **codec_write_reg()** использует функции интерфейса, о которых говорилось в Главе 8, "Протокол связи между микросхемами".

Если вы хотите создать в вашем драйвере для распечатки регистров во время отладки интерфейс */proc* или экспортировать какой-либо параметр при нормальной эксплуатации, пользуйтесь услугами **snd_card_proc_new()** и ей подобных. В Распечатке 13.1 файлы интерфейса */proc* не используются.

Если вы соберёте и загрузите модуль драйвера из Распечатки 13.1, то увидите два новых узла устройств, появившихся в MP3 плеере: */dev/snd/pcmC0D0p* и */dev/snd/controlC0*. Первый является интерфейсом для воспроизведения звука, а второй представляет собой интерфейс для управления микшером. Приложение декодирует MP3 с помощью *alsa-lib* и управляет потоком музыки через эти узлы устройства.

Распечатка 13.1. ALSA драйвер для MP3 плеера на Linux

Код:

```
#include <linux/platform_device.h>
#include <linux/soundcard.h>
#include <sound/driver.h>
#include <sound/core.h>
#include <sound/pcm.h>
#include <sound/initval.h>
#include <sound/control.h>

/* Частоты воспроизведения, поддерживаемые кодеком */
static unsigned int mycard_rates[] = {
    8000,
    48000,
};

/* Аппаратные ограничения для канала воспроизведения */
static struct snd_pcm_hw_constraint_list mycard_playback_rates = {
    .count = ARRAY_SIZE(mycard_rates),
    .list = mycard_rates,
    .mask = 0,
};

static struct platform_device *mycard_device;
static char *id[SNDRV_CARDS] = SNDRV_DEFAULT_STR;

/* Аппаратные возможности для PCM потока */
static struct snd_pcm_hardware mycard_playback_stereo = {
    .info = (SNDRV_PCM_INFO_MMAP | SNDRV_PCM_INFO_BLOCK_TRANSFER),
    .formats = SNDRV_PCM_FMTBIT_S16_LE, /* 16 бит на канал, сначала младший */
    .rates = SNDRV_PCM_RATE_8000_48000, /* Диапазон частот дискретизации ЦАП
*/
    .rate_min = 8000,                   /* Минимальная частота дискретизации
*/
    .rate_max = 48000,                  /* Максимальная частота дискретизации
*/
    .channels_min = 2,                  /* Поддерживается левый и правый канал
*/
```

```
    .channels_max = 2,                  /* Поддерживается левый и правый канал
*/
    .buffer_bytes_max = 32768,          /* Максимальный размер буфера */
};

/* Открываем устройство в режиме воспроизведения */
static int
mycard_pb_open(struct snd_pcm_substream *substream)
{
    struct snd_pcm_runtime *runtime = substream->runtime;

    /* Инициализируем структуры драйвера */
    /* ... */
    /* Инициализируем регистры кодека */
    /* ... */
    /* Подключаем возможности оборудования этого компонента PCM */
    runtime->hw = mycard_playback_stereo;

    /* Информируем ядро ALSA об ограничениях, которые имеет
       кодек. Например, в данном случае, он поддерживает
       частоты дискретизации PCM только от 8000 Гц до 48000 Гц */
    snd_pcm_hw_constraint_list(runtime, 0,
                        SNDRV_PCM_HW_PARAM_RATE,
                        &mycard_playback_rates);
    return 0;
}

/* Закрытие */
static int
mycard_pb_close(struct snd_pcm_substream *substream)
{
    /* Выключение кодека, остановка DMA, освобождение структур данных */
    /* ... */
    return 0;
}

/* Запись в регистры кодека через шину, соединяющую процессор и кодек */
void
codec_write_reg(uint codec_register, uint value)
{
    /* ... */
}

/* Подготовка к передаче потока звука в кодек */
static int
mycard_pb_prepare(struct snd_pcm_substream *substream)
{
    /* Разрешаем прерывание по завершении передачи DMA, записывая
       CONTROL_REGISTER с помощью codec_write_reg() */

    /* Устанавливаем частоту дискретизации, записывая SAMPLING_RATE_REGISTER
*/

    /* Конфигурируем источник частоты и включаем тактирование,
       делая запись в CLOCK_INPUT_REGISTER */
```

```
    /* Выделение дескрипторов DMA для передачи звука */

    return 0;
}

/* Включение звука/стоп/... */
static int
mycard_pb_trigger(struct snd_pcm_substream *substream, int cmd)
{
    switch (cmd) {
    case SNDRV_PCM_TRIGGER_START:
        /* Связываем созданный буфер звука подпотока (который является
           смещением в runtime->dma_area) с помощью dma_map_single(),
           помещает полученный в результате адрес в регистр контроллер звука
           DMA_ADDRESS_REGISTER и начинаем DMA */
        /* ... */
        break;

    case SNDRV_PCM_TRIGGER_STOP:
        /* Останавливаем поток. Отключаем буфер DMA с помощью dma_unmap_single
() */
        /* ... */
        break;

    default:
        return -EINVAL;
        break;
    }

    return 0;
}

/* Создание буферов DMA с помощью заранее выделенной для DMA памяти в
   методе probe(). dma_[map|unmap]_single() выполняется для этой области позже
*/
static int
mycard_hw_params(struct snd_pcm_substream *substream,
                 struct snd_pcm_hw_params *hw_params)
{
    /* Для удовлетворения этого запроса памяти используем память, заранее
       выделенную в mycard_audio_probe() */
    return snd_pcm_lib_malloc_pages(substream,
                params_buffer_bytes(hw_params));
}

/* Обратна для mycard_hw_params() */
static int
mycard_hw_free(struct snd_pcm_substream *substream)
{
    return snd_pcm_lib_free_pages(substream);
}

/* Информация о громкости */
static int
```

```
mycard_pb_vol_info(struct snd_kcontrol *kcontrol,
                   struct snd_ctl_elem_info *uinfo)
{
    uinfo->type = SNDRV_CTL_ELEM_TYPE_INTEGER;/* Целочисленный тип */
    uinfo->count = 1;                         /* Число регуляторов */
    uinfo->value.integer.min = 0;             /* Минимальный уровень громкости
*/
    uinfo->value.integer.max = 10;            /* Максимальный уровень
громкости */
    uinfo->value.integer.step = 1;            /* Шаг изменения */
    return 0;
}

/* Регулятор громкости воспроизведения */
static int
mycard_pb_vol_put(struct snd_kcontrol *kcontrol,
                  struct snd_ctl_elem_value *uvalue)
{
    int global_volume = uvalue->value.integer.value[0];

    /* Записываем global_volume в VOLUME_REGISTER
       с помощью codec_write_reg() */
    /* ... */
    /* Если громкость изменилась относительно текущего значения, возвращаем 1.
       Если получена ошибка, возвращаем отрицательное значение. Иначе
возвращаем 0 */
}

/* Получение громкости воспроизведения */
static int
mycard_pb_vol_get(struct snd_kcontrol *kcontrol,
                  struct snd_ctl_elem_value *uvalue)
{
    /* Читаем global_volume из VOLUME_REGISTER
       и возвращаем её через uvalue->integer.value[0] */
    /* ... */
    return 0;
}

/* Точки входа для микшера воспроизведения */
static struct snd_kcontrol_new mycard_playback_vol = {
    .iface = SNDRV_CTL_ELEM_IFACE_MIXER,/* Управление имеет тип MIXER */
    .name  = "MP3 Volume",              /* Имя */
    .index = 0,                         /* Номер кодека: 0 */
    .info  = mycard_pb_vol_info,        /* Информация о громкости */
    .get   = mycard_pb_vol_get,         /* Получение громкости */
    .put   = mycard_pb_vol_put,         /* Установка громкости */
};

/* Операторы для потока воспроизведения PCM */
static struct snd_pcm_ops mycard_playback_ops = {
    .open = mycard_playback_open,       /* Открытие */
    .close = mycard_playback_close,     /* Закрытие */
    .ioctl = snd_pcm_lib_ioctl,         /* Универсальный обработчик ioctl */
    .hw_params = mycard_hw_params,      /* Параметры оборудования */
```

```
    .hw_free = mycard_hw_free,          /* Освобождение параметров h/w */
    .prepare = mycard_playback_prepare, /* Подготовка для передачи аудио
потока */
    .trigger = mycard_playback_trigger, /* Вызывается, когда движок PCM
                                           стартует/останавливается/встаёт в
паузу */
};

/* Метод драйвера платформы probe() */
static int __init
mycard_audio_probe(struct platform_device *dev)
{
    struct snd_card *card;
    struct snd_pcm *pcm;
    int myctl_private;

    /* Создаём экземпляр структуры snd_card */
    card = snd_card_new(-1, id[dev->id], THIS_MODULE, 0);

    /* Создаём новый экземпляр PCM с 1 потоком воспроизведения
       и 0 потоков захвата */
    snd_pcm_new(card, "mycard_pcm", 0, 1, 0, &pcm);

    /* Создаём свои начальные буферы DMA */
    snd_pcm_lib_preallocate_pages_for_all(pcm,
                    SNDRV_DMA_TYPE_CONTINUOUS,
                    snd_dma_continuous_data
                    (GFP_KERNEL), 256*1024,
                    256*1024);

    /* Объединяем операции воспроизведения с экземпляром PCM */
    snd_pcm_set_ops(pcm, SNDRV_PCM_STREAM_PLAYBACK,
                    &mycard_playback_ops);

    /* Связываем элемент управления микшером с этой картой */
    snd_ctl_add(card, snd_ctl_new1(&mycard_playback_vol,
                &myctl_private));
    strcpy(card->driver, "mycard");

    /* Регистрируем звуковую карту */
    snd_card_register(card);

    /* Сохраняем карту для доступа из других методов */
    platform_set_drvdata(dev, card);

    return 0;
}

/* Метод драйвера платформы remove() */
static int
mycard_audio_remove(struct platform_device *dev)
{
    snd_card_free(platform_get_drvdata(dev));
    platform_set_drvdata(dev, NULL);
    return 0;
```

```
}

/* Определение драйвера платформы */
static struct platform_driver mycard_audio_driver = {
    .probe  = mycard_audio_probe,   /* Метод probe */
    .remove = mycard_audio_remove, /* Метод remove */
    .driver = {
        .name = "mycard_ALSA",
    },
};

/* Инициализация драйвера */
static int __init
mycard_audio_init(void)
{
    /* Регистрируем драйвер платформы и устройство */
    platform_driver_register(&mycard_audio_driver);
    mycard_device = platform_device_register_simple("mycard_ALSA",
                                -1, NULL, 0);
    return 0;
}

/* Отключение драйвера */
static void __exit
mycard_audio_exit(void)
{
    platform_device_unregister(mycard_device);
    platform_driver_unregister(&mycard_audio_driver);
}

module_init(mycard_audio_init);
module_exit(mycard_audio_exit);
MODULE_LICENSE("GPL");
```

Программирование ALSA

Чтобы понять, как библиотека пользовательского пространства ***alsa-lib*** взаимодействует с ALSA драйверами пространства ядра, давайте напишем простое приложение, которое устанавливает уровень громкости MP3 плеера. Мы свяжем сервисы ***alsa-lib***, используемые приложением, с методами управления микшером, определёнными в Распечатке 13.1. Начнём с загрузки драйвера и изучения возможностей микшера:

```
bash> amixer contents
...
numid=3,iface=MIXER,name="MP3 Volume"
; type=INTEGER,...
...
```

Сначала в приложении для управления громкостью выделяется пространство для объектов ***alsa-lib***, необходимых для выполнения операции регулировки громкости:

```
#include <alsa/asoundlib.h>
snd_ctl_elem_value_t *nav_control;
```

```
snd_ctl_elem_id_t *nav_id;
snd_ctl_elem_info_t *nav_info;

snd_ctl_elem_value_alloca(&nav_control);
snd_ctl_elem_id_alloca(&nav_id);
snd_ctl_elem_info_alloca(&nav_info);
```

Далее устанавливается тип интерфейса в **SND_CTL_ELEM_IFACE_MIXER**, как указано в структуре **mycard_playback_vol** в Распечатке 13.1:

```
snd_ctl_elem_id_set_interface(nav_id, SND_CTL_ELEM_IFACE_MIXER);
```

Теперь устанавливается **numid** для громкости MP3, полученный из вышеприведённого вывода из ***amixer***:

```
snd_ctl_elem_id_set_numid(nav_id, 3); /* num_id=3 */
```

Открывается микшер узел, ***/dev/snd/controlC0***. Третий аргумент для **snd_ctl_open()** определяет номер карты в имени узла:

```
snd_ctl_open(&nav_handle, card, 0);
/* Connect data structures */
snd_ctl_elem_info_set_id(nav_info, nav_id);
snd_ctl_elem_info(nav_handle, nav_info);
```

Проверяется тип поля в структуре **snd_ctl_elem_info**, определённой в **mycard_pb_vol_info()** в Распечатке 13.1, следующим образом:

```
if (snd_ctl_elem_info_get_type(nav_info) !=
            SND_CTL_ELEM_TYPE_INTEGER) {
    printk("Mismatch in control type\n");
}
```

Запрашивается поддерживаемый кодеком диапазон гормкости с помощью метода драйвера **mycard_pb_vol_info()**:

```
long desired_volume = 5;
long min_volume = snd_ctl_elem_info_get_min(nav_info);
long max_volume = snd_ctl_elem_info_get_max(nav_info);
/* Убеждаемся, что desired_volume в диапазоне от min_volume
   до max_volume */
/* ... */
```

Как это определено в **mycard_pb_vol_info()** в Распечатке 13.1, минимальное и максимальное значения, которые возвращаются показанными выше вспомогательными процедурами ***alsa-lib***, равны соответственно 0 и 10.

Наконец, устанавливается желаемая громкость и записывается в кодек:

```
snd_ctl_elem_value_set_integer(nav_control, 0, desired_volume);
snd_ctl_elem_write(nav_handle, nav_control);
```

Вызов **snd_ctl_elem_write()** приводит к вызову **mycard_pb_vol_put()**, которая записывает желаемый уровень громкости в VOLUME_REGISTER кодека.

Сложность декодирования MP3

Приложение-декодер MP3 работающее на плеере, как показано на Рисунке 13.4, требует постоянную поставку фреймов MP3 с диска CF, который может поддерживать часто используемый MP3 со скоростью 128 кБит/с. Обычно это не является проблемой для большинства маломощных устройств, но в данном случае, требуется буферизация каждой песни в памяти перед её декодированием. (Фреймы MP3 при 128 кБит/с потребляют примерно 1 Мб на минуту музыки.)

MP3 декодирование является простой операцией и обычно происходит на лету, но MP3 кодирование является тяжёлой операцией, и не может быть выполнено в режиме реального времени без помощи оборудования. Голосовые кодеки, такие как G.711 и G.729, используемые в средах передачи Голоса поверх IP (VoIP), могут, однако, кодировать и декодировать аудио данные в реальном времени.

Отладка

Вы можете включить опции в Device Drivers -> Sound -> Advanced Linux Sound Architecture в меню конфигурации ядра для включения кода отладки ALSA (**CONFIG_SND_DEBUG**), подробных сообщений **printk()** (**CONFIG_SND_VERBOSE_PRINTK**), а также подробного содержания procfs (**CONFIG_SND_VERBOSE_PROCFS**).

Информация procfs, относящаяся к драйверам ALSA, находится в каталоге ***/proc/asound/***. Информацию о модели устройства, связанной с каждым устройством звукового класса, ищите в ***/sys/class/sound/***.

Если вы думаете, что нашли ошибку в драйвере ALSA, сообщите об этом в списке рассылки alsa-devel (http://mailman.alsaproject.org/mailman/listinfo/alsa-devel). Список рассылки linux-audio-dev (http://music.columbia.edu/mailman/listinfo/linux-audio-dev/), также называемый списком ***Разработчиков звука Linux, Linux Аудио Developers*** (LAD), обсуждает вопросы, связанные с архитектурой звука Linux и аудио приложениями.

Где искать информацию

Звуковое ядро, аудио шины, архитектуры и устаревших набор OSS имеют свои собственные отдельные подкаталоги в ***sound/***. Чтобы увидеть реализацию интерфейса AC'97, загляните внутрь ***sound/pci/ac97/***. Для примера аудио драйвера на основе I2S, посмотрите в ***sound/soc/at91/at91-ssc.c***, это аудио драйвер для серии AT91 от Atmel на основе встраиваемых процессоров ARM. Если вы не можете найти близкое соответствие, то в качестве отправной точки для разработки своего драйвера ALSA используйте ***sound/drivers/dummy.c***.

Информация о драйверах ALSA и OSS содержится в ***Documentation/sound/****. ***Documentation/sound/alsa/DocBook/*** содержит DocBook по написанию драйверов ALSA. Руководство по конфигурации ALSA доступно в ***Documentation/sound/alsa/ALSA-Configuration.txt***. **Sound-HOWTO**, доступный на http://tldp.org/HOWTO/Sound-HOWTO/, отвечает на некоторые часто задаваемые вопросы, связанные с поддержкой аудио устройств в Linux.

Madplay - это программный декодер MP3 и проигрыватель, который работает и с ALSA, и с OSS. Вы можете посмотреть его исходники для получения советов о программировании звука в пользовательском пространстве.

Двумя инструментами OSS без излишеств для воспроизведения и записи являются ***rawplay*** и ***rawrec***, чьи исходники можно загрузить с http://rawrec.sourceforge.net/.

Домашнюю страницу проекта Linux ALSA можно найти на www.alsa-project.org. Здесь вы найдёте последние новости о драйверах ALSA, подробную информацию о программных API ALSA, а также информацию о подписке на соответствующие списки рассылок. Исходники ***alsa-utils*** и ***alsa-lib***, которые можно загрузить с этой страницы, помогут вам при разработке приложений, работающих с ALSA.

Таблица 13.2 содержит основные структуры данных, используемые в этой главе, и их расположение в дереве исходных текстов. В Таблице 13.3 перечислены основные программные интерфейсы ядра, которые использовались в этой главе с указанием места их определения.

Таблица 13.2. Список структур данных

Структура данных	Местоположение	Описание
snd_card	***include/sound/core.h***	Представление звуковой карты
snd_pcm	***include/sound/pcm.h***	Экземпляр объекта PCM
snd_pcm_ops	***include/sound/pcm.h***	Используется для подключения операций с объектом PCM
snd_pcm_substream	***include/sound/pcm.h***	Информация о текущем звуковом потоке
snd_pcm_runtime	***include/sound/pcm.h***	Подробности об аудио потоке во время выполнения
snd_kcontrol_new	***include/sound/control.h***	Представление элемента

		управления ALSA

Таблица 13.3. Список программных интерфейсов ядра

Интерфейс ядра	Местоположение	Описание
snd_card_new()	*sound/core/init.c*	Создание экземпляра структуры **snd_card**
snd_card_free()	*sound/core/init.c*	Удаляет экземпляр **snd_card**
snd_card_register()	*sound/core/init.c*	Регистрирует звуковую карту в ядре ALSA
snd_pcm_lib_preallocate_pages_for_all()	*sound/core/pcm_memory.c*	Предварительное создание буферов для звуковой карты
snd_pcm_lib_malloc_pages()	*sound/core/pcm_memory.c*	Выделение буферов DMA для звуковой карты
snd_pcm_new()	*sound/core/pcm.c*	Создание экземпляра объекта PCM
snd_pcm_set_ops()	*sound/core/pcm_lib.c*	Подключение операций воспроизведения или захвата к объекту PCM
snd_ctl_add()	*sound/core/control.c*	Подключение элемента управления микшером к звуковой карте
snd_ctl_new1()	*sound/core/control.c*	Создание структуры **snd_kcontrol** и инициализация её соответствующими операциями управления
snd_card_proc_new()	*sound/core/info.c*	Создание записи в */proc* и связывание её с экземпляром карты

Индекс

- A -

API кадрового буфера 32

- P -

PS/2 мышь
Пример драйвера: Мышь-колёсико 13

- А -

Архитектура отображения 25
Северный мост 25
Стандарты видеокабелей 26

- В -

Видео подсистема Linux 28

- Г -

Где искать информацию 22, 55, 79
Где искать исходники
Таблица 12.2. Список структур данных 55
Таблица 12.3. Список программных интерфейсов ядра 56
Таблица 13.2. Список структур данных 79
Таблица 13.3. Список программных интерфейсов ядра 80
Таблица 7.1. Список структур данных 22
Таблица 7.2. Список программных интерфейсов ядра 22
Глава 12. Драйверы Видео 24
API кадрового буфера 32
Архитектура отображения 25
Видео подсистема Linux 28
Где искать информацию 55
Драйверы кадрового буфера 35
Консольные драйверы 48
Отладка 54
Параметры дисплея 30
Глава 13. Драйверы Звука 58
Где искать информацию 79
Звуковая архитектура 59
Звуковая подсистема Linux 61
Отладка 78
Пример устройства: MP3 плеер 64
Глава 7. Драйверы ввода 1
Где искать информацию 22
Драйверы входных событий 3
Драйверы устройств ввода 10
Отладка 21

- Д -

Драйверы входных событий 3
Дополнительная информация об интерфейсе событий 8
Интерфейс Evdev 4
Драйверы кадрового буфера 35
Пример устройства: Навигационная система 35
Драйверы устройств ввода 10
Serio 10
Акселерометры 19
Клавиатуры 10
Мыши 13
Сенсорные контроллеры 18
События вывода 19

- З -

Звуковая архитектура 59
Звуковая подсистема Linux 61

- И -

Интерфейс Evdev
Пример устройства: Виртуальная мышь 4

- К -

Клавиатуры
USB и Bluetooth клавиатуры 13
Клавиатуры ПК 10
Консольные драйверы 48
Логотип при загрузке 53
Пример устройства: Снова сотовый телефон 49

- М -

Мыши
    PS/2 мышь 13
    USB и Bluetooth мыши 18
    Сенсорные панели 15
    Устройства позиционирования 15

- О -

Отладка 21, 54, 78

- П -

Параметры дисплея 30
Пример устройства: MP3 плеер 64, 77
    Методы и структуры драйвера 66
    Программирование ALSA 75
Пример устройства: Навигационная система
    DMA из кадрового буфера 41
    Гашение экрана 40
    Контраст и подсветка 41
    Методы ускорения 40
    Проверка и установка параметров 38
    Структуры данных 36
    Цветовые режимы 39