СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Компьютер даёт возможность получить понятную информацию для человека. Все виды персональных компьютеров взаимодействуют с пользователем посредством устройств ввода-вывода.

Все компьютеры оборудованы разъемом для акустичской системы (наушники, колонки). С развитием компьютеров и их минимизацией разъемы для наушников и микрофона объединились. Устройство, сочетающее микрофон и наушники, называется гарнитурой, а разъем для гарнитуры - гарнитурный разъем.

В данной работе стоит задача написания программу, способную обнаруживать подключение и отключение гарнитуры и модуль ядра, включающий/выключающий LED индикатор клавиатуры при обнаружении подключения или отключения.

1 Аналитическая часть

В данном разделе производится постановка задачи и анализ методов решения поставленной задачи.

1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу необходимо разработать:

- программу, обнаруживающую событие нажатия кнопки гарнитуры;
- протокол взаимодейсвтия пользовательской программы с модулем ядра;
- модуль ядра, работающий с клавиатурой, который включает/выключает LED индикаторы клавиатуры.

1.2 Загружаемые модули ядра

Одной из хороших особенностей Linux является способность расширения функциональности ядра во время работы. Это означает, что вы можете добавить функциональность в ядро (и убрать её), когда система запущена и работает. Часть кода, которая может быть добавлена в ядро во время работы, называется модулем. Ядро Linux предлагает поддержку довольно большого числа типов (или классов) модулей, включая, но не ограничиваясь, драйверами устройств. Каждый модуль является подготовленным объектным кодом (не слинкованным для самостоятельной работы), который может быть динамически подключен в работающее ядро программой «insmod» и отключен программой «rmmod».

За основу был взят подход динамической загрузки драйвера, который представляет собой загрузку при помощи отдельных модулей с расширением *.ko (объект ядра).

Загружаемые модули ядра имеют ряд фундаментальных отличий от элементов, интегрированных непосредственно в ядро, а также от обычных программ. Обычная программа содержит главную процедуру (main)в отличие от загружаемого модуля, содержащего функции входа и выхода (в версии 2.6 эти функции можно именовать как угодно). Функция входа вызывается, когда модуль загружается в ядро, а функция выхода – соответственно при выгрузке из ядра. Поскольку функции входа и выхода являются пользовательскими, для указания назначения этих функций используются макросы module_init и module_exit . Загружаемый модуль содержит также набор обязательных и дополнительных макросов. Они определяют тип лицензии, автора и описание модуля, а также другие параметры[1].

Пример очень простого загружаемого модуля ядра приведен на рисунке 1.1

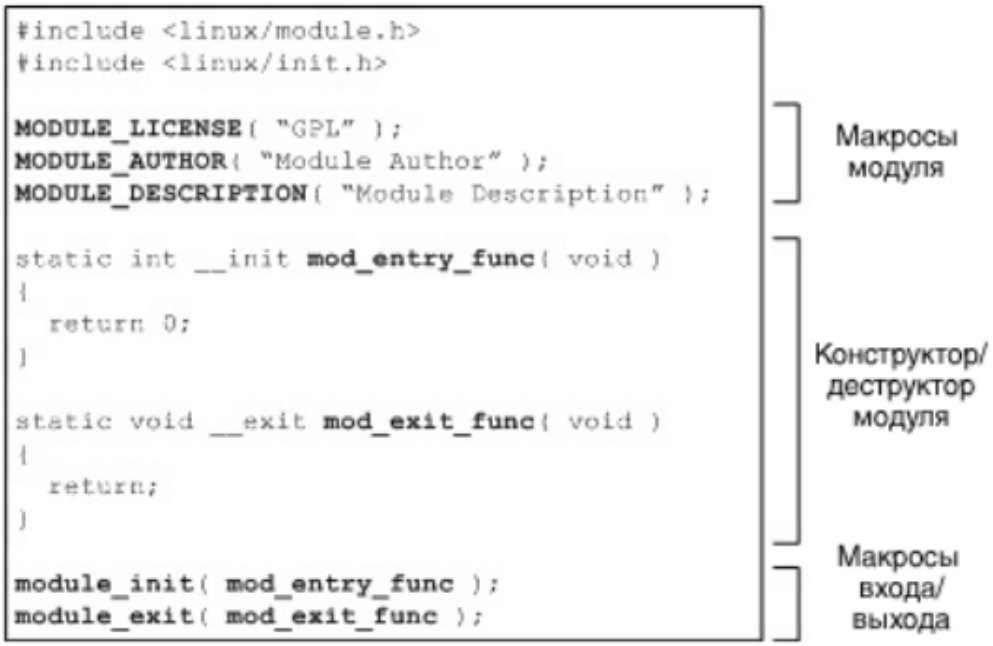


Рисунок 1.1 – Пример загружаемого модуля ядра.

1.3 Пространство пользователя и пространство ядра

Модули работают в пространстве ядра, в то время как приложения работают в пользовательском пространстве.

На практике ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Кроме того, операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих нетривиальных задач становится возможным, только если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Каждый современный процессор позволяет реализовать такое поведение. Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли инекоторые операции на более низких уровнях не допускаются; программный код может переключить один уровень на другой только ограниченным числом способов. Unix системы разработаны для использования этой аппаратной функции с помощью двух таких уровней. Все современные процессоры имеют не менее двух уровней защиты, а некоторые, например семейство x86, имеют больше уровней; когда существует несколько уровней, используются самый высо-

кий и самый низкий уровни. Под Unix ядро выполняется на самом высоком уровне (также называемым режимом супервизора), где разрешено всё, а приложения выполняются на самом низком уровне (так называемом пользовательском режиме), в котором процессор регулирует прямой доступ к оборудованию и несанкционированный доступ к памяти. Unix выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, выполняя системный вызов, работает в контексте процесса - он действует от имени вызывающего процесса и в состоянии получить данные в адресном пространстве процесса. Код, который обрабатывает прерывания, с другой стороны, является асинхронным по отношению к процессам и не связан с каким-либо определённым процессом.

Ролью модуля является расширение функциональности ядра; код модулей выполняется в пространстве ядра. Обычно драйвер выполняет обе задачи, изложенные ранее: некоторые функции в модуле выполняются как часть системных вызовов, а некоторые из них отвечают за обработку прерываний.

1.4 Анализ аудио-подсистем

1.4.1 Open Sound System

Open Sound System (OSS) — унифицированный драйвер для звуковых карт и других звуковых устройств в различных UNIX-подобных операционных системах.

OSS основан на Linux Sound Driver и в настоящее время работает на большом числе платформ: Linux, FreeBSD, OpenSolaris и т. д. [2] /dev/dsp и /dev/audio — основные файлы устройств для цифровых приложений. Любые данные, записанные в эти файлы, воспроизводятся на DAC/PCM/DSP устройстве звуковой карты. Чтение из этих файлов возвращает звуковые данные, записанные с текущего входного источника (по умолчанию это Микрофонный вход).

При чтении из /dev/dsp мы получаем несжатый аудиопоток с микрофона компьютера через вход звуковой карты.

1.4.2 ALSA

ALSA (англ. Advanced Linux Sound Architecture, Продвинутая звуковая архитектура Linux) — архитектура звуковых драйверов, а также широкий их набор для операционной системы Linux, призванный сменить Open Sound System (OSS). ALSA тесно связана с ядром Linux[3]. Архитектуру звуковой подсисте-

мы Linux показан на рисунке 1.2.

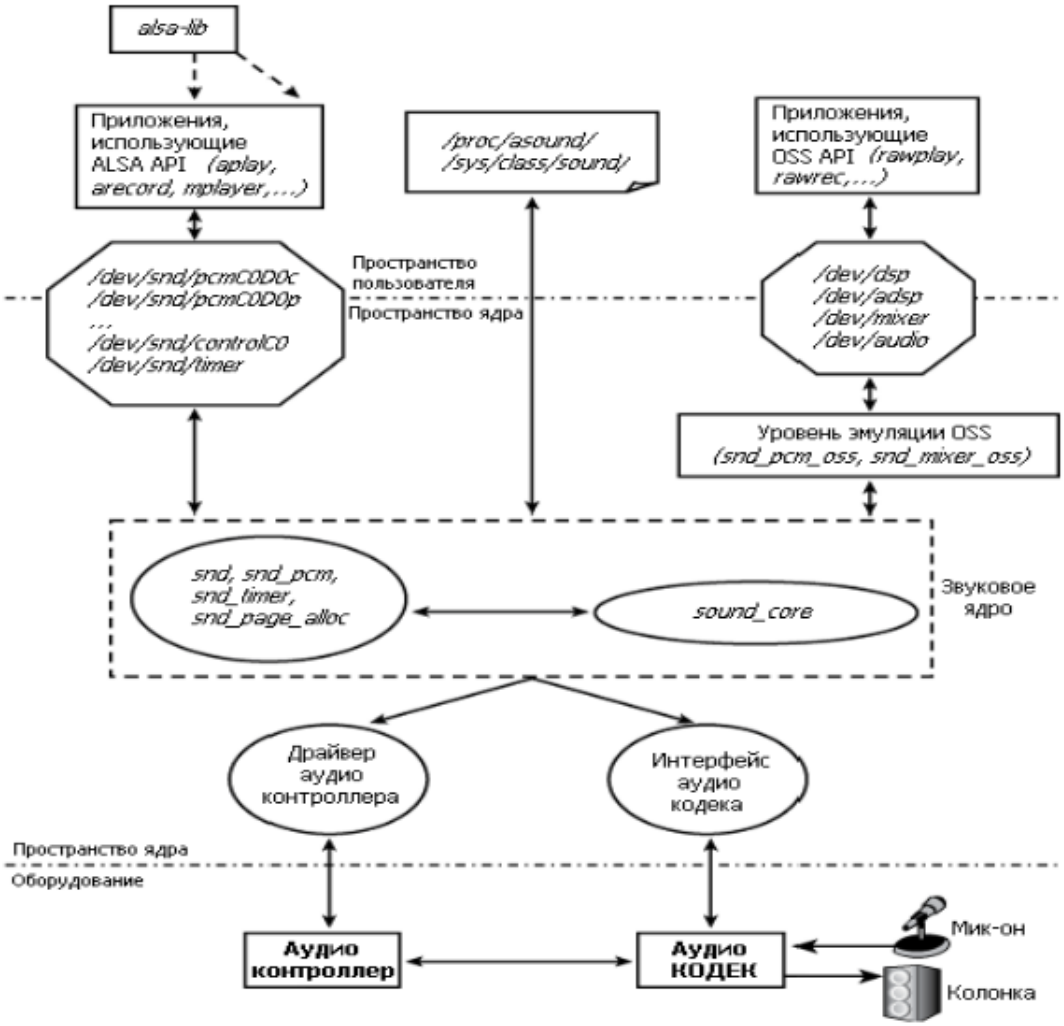


Рисунок 1.2 – Пример загружаемого модуля ядра.

Основными частями подсистемы являются:

а) 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 1, 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 устрой-

ства 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;

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

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

г) 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/pcmC0D а /dev/mixer связан с /dev/snd/controlC0;

д) Интерфейс procfs and sysfs для доступа к информации через /proc/asound/ и /sys/class/sound/;

е) 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;

ж) 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.

1.5 Анализ реализации отложенных действии

Для высокоскоростных операций с потоками ядро Linux предлагает тасклеты и очереди работ. С помощью тасклетов и очередей работ реализуются функции отложенного исполнения, что заменяет старый механизм использова-

ния нижних половин в драйверах.

1.5.1 Тасклеты

Тасклеты являются структурами отложенного исполнения, которые вы можете запланировать на запуск позже в виде зарегистрированных функций. Верхняя половина (обработчик прерываний) выполняет небольшой объем работ, а затем планирует тасклеты, которые будут выполнены позже в нижней половине.

При создании функции тасклета используются соответствующие данные (my_tasklet_function и my_tasklet_data), который затем используется при объявлении нового тасклета с помощью макроса DECLARE_TASKLET. Когда модуль вставляется, то планируется исполнение тасклета, что делает его исполняемым в определенный момент в будущем. Когда модуль выгружается, вызывается функция tasklet_kill для того, чтобы изъять тасклет из состояния запланированного исполнения.

1.5.2 Очереди работ

Вместо того, чтобы предлагать однократную схему отложенного исполнения, как в случае с тасклетами, очереди работ являются обобщенным механизмом отложенного исполнения, в котором функция обработчика, используемая для очереди работ, может ”засыпать”(что невозможно в модели тасклетов). Очереди работ могут иметь более высокую латентность, чем тасклеты, но они имеют более богатое API для отложенного исполнения работ.

В очередях работ предлагается обобщенный механизм, в котором отсроченная функциональность переносится в механизмы выполнения нижних половин. В основе лежит очередь работ (структура workqueue_struct), который является структурой, в которую помещается данные объект work. Работа (т.е. объект work) представлена структурой work_struct, в которой идентифицируется работа, исполнение которой откладывается, и функция отложенного исполнения, которая будет при этом использоваться.

Основная структура для очереди работ сама является очередью. Эта структура используется при обработке очередей механизмом верхней половины, где планируются отложенные действия, передаваемые в нижнюю половину на последующее исполнение. Очередь работ создается с помощью макроса с именем create_workqueue, который возвращает ссылку на workqueue_struct. Вы позже сможете удалить (при необходимости) эту очередь работ с помощью вызова

функции destroy_workqueue.

1.6 Анализ структуры ядра

1.6.1 struct tty_driver

Структура tty_driver используется, чтобы зарегистрировать tty драйвер в ядре tty. Ниже описаны все различные поля в структуре.

Листинг 1 – Описание struct tty_driver

```
struct tty_driver {
    struct kref kref;           // подсчет ссылок.
    struct cdev **cdevs;        // выделенныезарегистрированные/ символьные устройства
    struct module *owner;       // Владелец модуля этого драйвера.
    const char *driver_name;    // Имя драйвера, используемое в /proc/tty и sysfs.
    const char *name;           // Имя узла драйвера.
    int name_base;              // Начальный номер, используемый при создании имени для устройства
    int major;                  // Старший номер для драйвера.
    int minor_start;            // Начальный младший номер для драйвера.
    unsigned int num;           // Количество младших номеров, связанных с драйверов.
    short type;                 // тип драйвера tty
    short subtype;              // подтип драйвера tty
    struct ktermios init_termios;  // termios, который будет первоначально установлен для каждого терминала
    unsigned long flags;        // Флаги драйвера, как описывалось ранее в этой главе.
    struct proc_dir_entry *proc_entry; // Структура записи драйвера в /proc.
    struct tty_driver *other;   // Указатель на ведомый tty драйвер.

    struct tty_struct **ttys;
    // массив активных &struct tty_struct, установленный tty_standard_install()

    struct tty_port **ports;    // массив &struct tty_port
    struct ktermios **termios; // хранилище для termios при каждом закрытии TTY для следующего открытия
    void *driver_state;         // указатель на произвольные данные драйвера

    const struct tty_operations *ops;
    // перехватчики драйвера для TTY. Установите их с помощью tty_set_operations().

    struct list_head tty_drivers; // используется внутри для связывания tty_drivers вместе
}
```

1.6.2 struct tty_operations

Структура tty_operations содержит все функции обратного вызова, который могут быть установлены tty драйверами и вызваны ядром tty. Ниже описаны все различные поля в структуре.

Листинг 2 – Описание struct tty_operations

```
struct tty_operations {
    // lookup - Возвращает устройство tty, соответствующее idx, NULL, если он не используется в данный
    момент, и значение ERR_PTR в случае ошибки.
    struct tty_struct * (*lookup)(struct tty_driver *driver, struct inode *inode, int idx);

    // install - Установите новый tty во внутренние таблицы self.
    int  (*install)(struct tty_driver *driver, struct tty_struct *tty);

    // remove - Удалить закрытый tty из внутренних таблиц self.
    void (*remove)(struct tty_driver *driver, struct tty_struct *tty);

    // open - процедура вызывается при открытии определенного устройства tty.
    int  (*open)(struct tty_struct * tty, struct file * filp);

    // close - процедура вызывается, когда определенное @tty устройство закрывается.
    void (*close)(struct tty_struct * tty, struct file * filp);

    // shutdown - процедура вызывается при блокировке tty, когда конкретное устройство tty закрывается в
    последний раз.
    void (*shutdown)(struct tty_struct *tty);

    // cleanup - процедура вызывается асинхронно, когда конкретное устройство tty закрывается в последний раз,
    освобождая ресурсы.
    void (*cleanup)(struct tty_struct *tty);

    // write - процедура вызывается ядром для записи серии (count) символов (buf) в устройство tty.
    int  (*write)(struct tty_struct * tty, const unsigned char *buf, int count);

    // put_char - подпрограмма вызывается ядром для записи одного символа ch в устройство tty.
```

```
27      int  (*put_char)(struct tty_struct *tty, unsigned char ch);
28
29      // flush_chars - процедура вызывается ядром после того, как оно записало серию символов в устройство tty
        с помощью put_char().
30      void (*flush_chars)(struct tty_struct *tty);
31
32      // write_room - процедура возвращает количество символов, которые драйвер tty примет для записи в очередь.
33      int  (*write_room)(struct tty_struct *tty);
34
35      // chars_in_buffer - процедура возвращает количество символов в частной очереди вывода устройства.
36      int  (*chars_in_buffer)(struct tty_struct *tty);
37
38      // ioctl -  подпрограмма позволяет драйверу tty реализовывать специфичные для устройства ioctls.
39      int  (*ioctl)(struct tty_struct *tty, unsigned int cmd, unsigned long arg);
40
41      // compat_ioctl - Реализовать обработку ioctl для битного32- процесса в битной64- системе.
42      long (*compat_ioctl)(struct tty_struct *tty, unsigned int cmd, unsigned long arg);
43
44      // set_termios - процедура позволяет драйверу @tty уведомлять об изменении настроек termios устройства.
45      void (*set_termios)(struct tty_struct *tty, struct ktermios * old);
46
47      // stop - процедура уведомляет драйвер tty о том, что он должен прекратить вывод символов на устройство tty
48      void (*stop)(struct tty_struct *tty);
49
50      // start - процедура уведомляет драйвер tty о том, что он возобновил отправку символов на устройство tty
51      void (*start)(struct tty_struct *tty);
52
53      // hangup - процедура уведомляет драйвер tty о том, что он должен повесить устройство tty.
54      void (*hangup)(struct tty_struct *tty);
55
56      // flush_buffer - процедура отбрасывает частный выходной буфер устройства.
57      void (*flush_buffer)(struct tty_struct *tty);
58
59      // set_ldisc - процедура позволяет драйверу tty уведомляться, когда дисциплина линии устройства изменяется.
60      void (*set_ldisc)(struct tty_struct *tty);
61
62      // tiocmget- процедура используется для получения битов состояния модема от драйвера tty
63      int (*tiocmget)(struct tty_struct *tty);
64
65      // tiocmset - процедура используется для установки битов состояния модема в драйвер tty.
66      int (*tiocmset)(struct tty_struct *tty,
67              unsigned int set, unsigned int clear);
68
69      // resize - ызывается, когда выдается запрос termios, который изменяет запрошенную геометрию терминала на
        ws.
70      int (*resize)(struct tty_struct *tty, struct winsize *ws);
71
72      // get_icount - Вызывается, когда устройство tty получает %TIOCGICOUNT ioctl.
73      int (*get_icount)(struct tty_struct *tty,
74                  struct serial_icounter_struct *icount);
75  };
```

Вывод

В данной работе для получения потока байтов от микрофонного разъема будем использовать API, предоставляемые ALSA. Для реализации отложенных действии будем использовать тасклет.

2 Конструкторский раздел

В данном разделе рассматривается процесс проектирования структуры программного обеспечения.

2.1 Структура программного обеспечения

Структура программного обеспечения состоит из:

- программы пользователя - программа, использующая ALSA API, обнаруживает события подключения и отключения гарнитуры;
- специального файла устройства - создан загружаемым модулем ядра для обеспечения взаимодействия программы пользователя и модулем ядра;
- модуля ядра, получающего информацию от программы пользователя через созданный модулем специальный файл устройства, работает с клавиатурой, который включает/выключает LED индикаторы клавиатуры.

2.2 Протокол взаимодействия модуля ядра и программы пользователя

Для взаимодействия модуля ядра и программы пользователя модулем ядра создан специальный файл (/dev), куда программа пользователя будет писать сообщения, а модуль ядра будет обрабатывать эти сообщения.

2.3 Алгоритм обнаружения события

Алгоритм пользовательской программы при обнаружения события подключения и отключения гарнитуры.

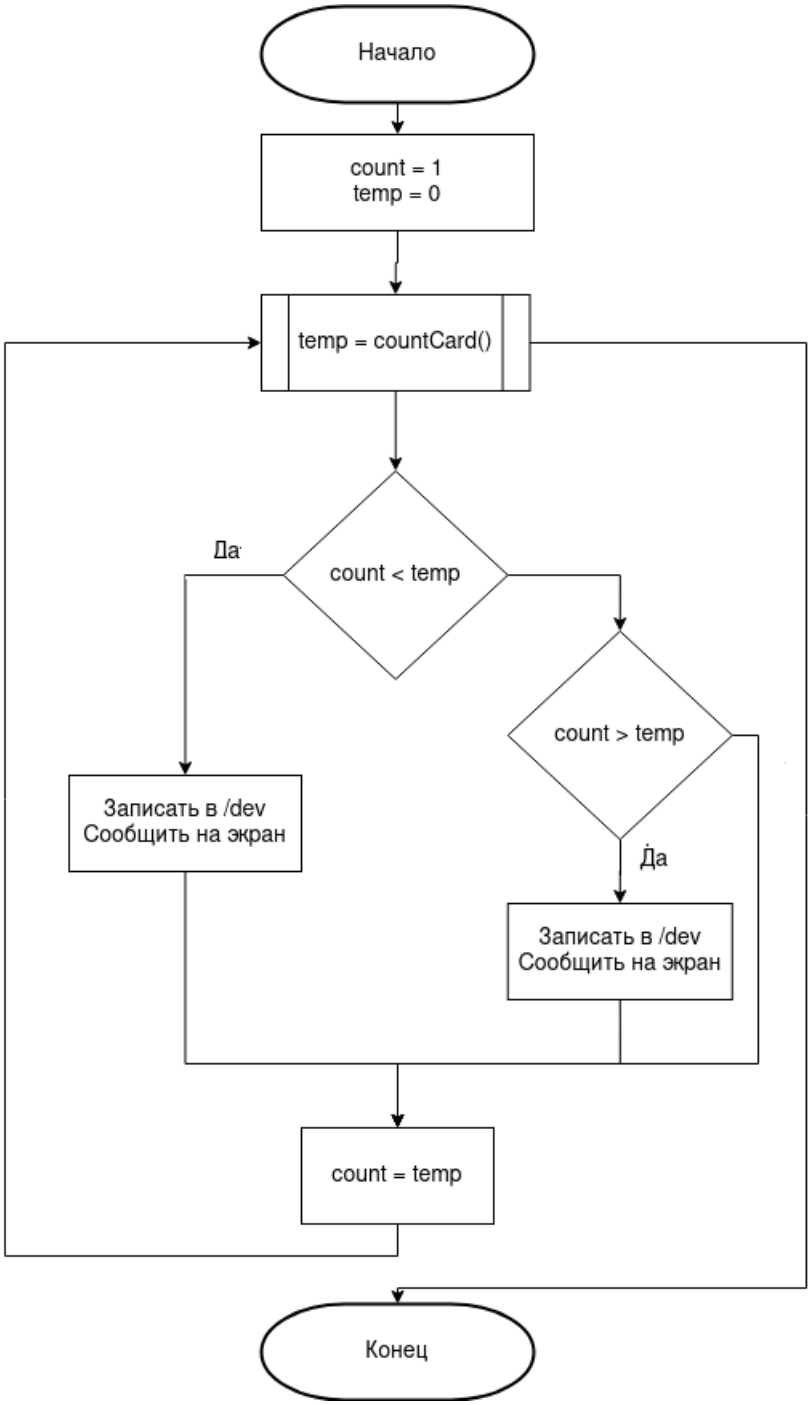


Рисунок 2.1 – Алгоритм пользовательской программы.

count — начальное количество звуковых устройств;

temp — текущее количество звуковых устройств;

/dev — специальный файл, котороыйт обеспечивает протокол взаимодействия пользовательской программы с модулем ядра.

2.4 Алгоритм управления индикатором

Алгоритм управления индикаторм при получении сообщения

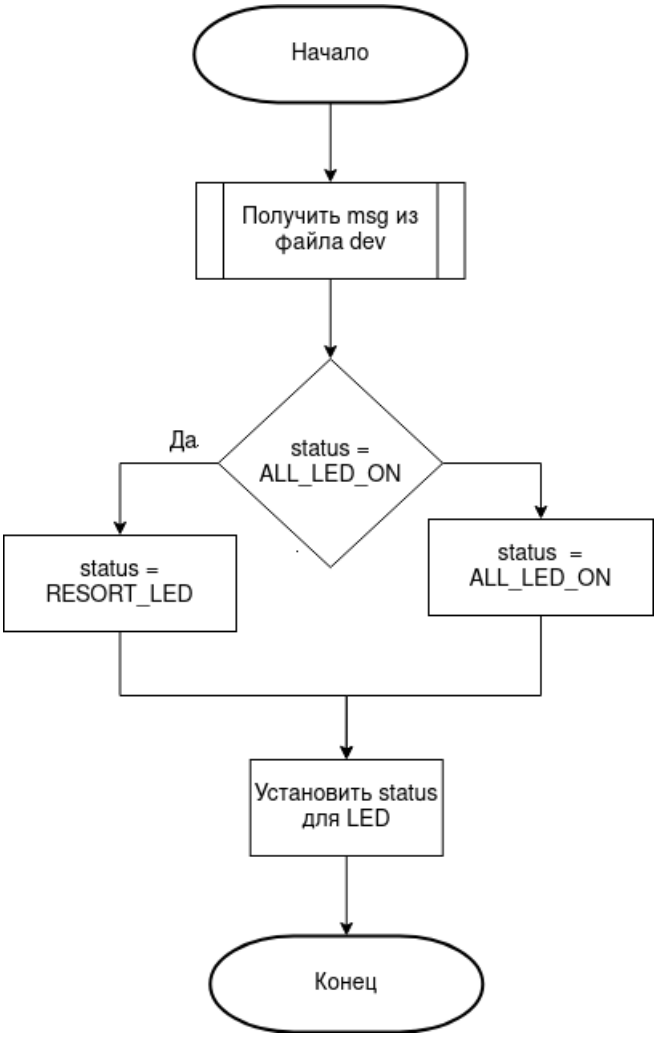


Рисунок 2.2 – Алгоритм управления индикаторм.

3 Технологический раздел

В данном разделе выбирается язык программирования, на котором будет реализована поставленная задача, производится выбор среды разработки и рассматриваются некоторые мометы реализации загружаемого модуля ядра.

3.1 Выбор языка программирования

В качестве языка программирования для реализации данной курсовой работы был выбран язык C. При помощи этого языка реализованы все модули ядра и драйверы с использованием встроенного в ОС Linux компилятора GCC.

3.2 Выбор среды разработки

В качестве среды разработки был выбран стандартный тектовый редактор ОС Linux.

Ниже показано содержимое Makefile, содержащего набор инструкций, используемых утилитой make в инструментарии автоматизации сборки.

Листинг 3 – Содержимое Makefile

```
CONFIG_MODULE_SIG=n

ifneq ($(KERNELRELEASE),)
    obj-m += module_dev.o
else
    KERNELDIR ?= /lib/modules/$(shell uname -r)/build
    PWD := $(shell pwd)
default:
    make -C $(KERNELDIR) M=$(PWD) modules
clean:
    make -C $(KERNELDIR) M=$(PWD) modules clean
    rm -rf a.out
endif
```

3.3 Описание некоторых моментов реализации

3.3.1 Модуль ядра

Код функции, обрабатывающая запись в специальный файл.

```
ssize_t module_dev_write(struct file *file, const char __user *buf, size_t size, loff_t *offset)
{
    printk(KERN_INFO "module_dev: called write %ld\n", size);

    copy_from_user(msg, buf, size);
    msg_length = size;

    is_empty = 0;

    if (msg[0] == '1')
    {
        tasklet_schedule(&my_tasklet);
    }
    printk(KERN_INFO "module_dev: write msg '%s'\n", msg);

    return size;
}
```

3.3.2 Работа с индикаторами клавиатуры

Доступ к индикаторам клавиатуры реализован через виртуальную консоль, доступную в ядре через библиотеку linux/console_struct.h. Отложенное действие реализовано через тасклет.

Поставновка таймера на выполнение

```
struct tty_driver *load_keyboard_led_driver(void)
{
    int i;
    struct tty_driver *driver;

    printk(KERN_INFO "kbleds: loading\n");
    printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
    printk(KERN_INFO "kbleds: MAX_NR_CONSOLES %i", MAX_NR_CONSOLES);

    for (i = 0; i < MAX_NR_CONSOLES; i++)
    {
        if (!vc_cons[i].d)
            break;
        printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
               MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
               (unsigned long)vc_cons[i].d->port.tty);
    }

    printk(KERN_INFO "kbleds: finished scanning consoles\n");

    driver = vc_cons[fg_console].d->port.tty->driver;
    printk(KERN_INFO "kbleds: tty driver magic %x\n", driver->name);

    return driver;
}
```

Функция, включающая/выключающая индикаторы

```
#define ALL_LEDS_ON 0x07
#define RESTORE_LEDS 0xFF

int is_led_on = 0;

void tasklet_fn_toggle_led(unsigned long data)
{
    unsigned long status;

    if (is_led_on)
    {
        is_led_on = 0;
        status = ALL_LEDS_ON;
    }
    else
    {
        is_led_on = 1;
        status = RESTORE_LEDS;
    }

    (my_driver->ops->ioctl)(vc_cons[fg_console].d->port.tty, KDSETLED, status);

}
```

3.3.3 Программа пользователя

```
#include <alsa/asoundlib.h>
#include <stdio.h>
#include <stdlib.h>

void write_signal_to_proc()
{
    FILE *fp = fopen("/proc/module_dir/dev", "w");
    if (fp == NULL)
        printf("unable to open proc file\n");

    fprintf(fp, "1");
    fclose(fp);
}

snd_pcm_t *pcm_device;

int main()
{
    int count = 1;
    while(1)
    {
        int totalCards = 0;
        int cardNum = -1;
        int err;

        for (;;) {
            if ((err = snd_card_next(&cardNum)) < 0) {
                fprintf(stderr, "Can't get the next card number: %s\n",
                                snd_strerror(err));
                break;
            }
```

```
32              if (cardNum < 0)
33                  break;
34              ++totalCards;
35          }
36
37          if(count < totalCards)
38          {
39              printf("%d %d\n", count, totalCards);
40              printf("Headset connected\n");
41              write_signal_to_proc();
42              write_signal_to_proc();
43          }
44          else if(count > totalCards)
45          {
46              printf("%d %d\n", count, totalCards);
47              printf("Headset disconnected\n");
48              write_signal_to_proc();
49              write_signal_to_proc();
50          }
51          count = totalCards;
52          snd_config_update_free_global();
53      }
54  }
```

4 Исследовательский раздел

В данном разделе производится вычислительные эксперименты.

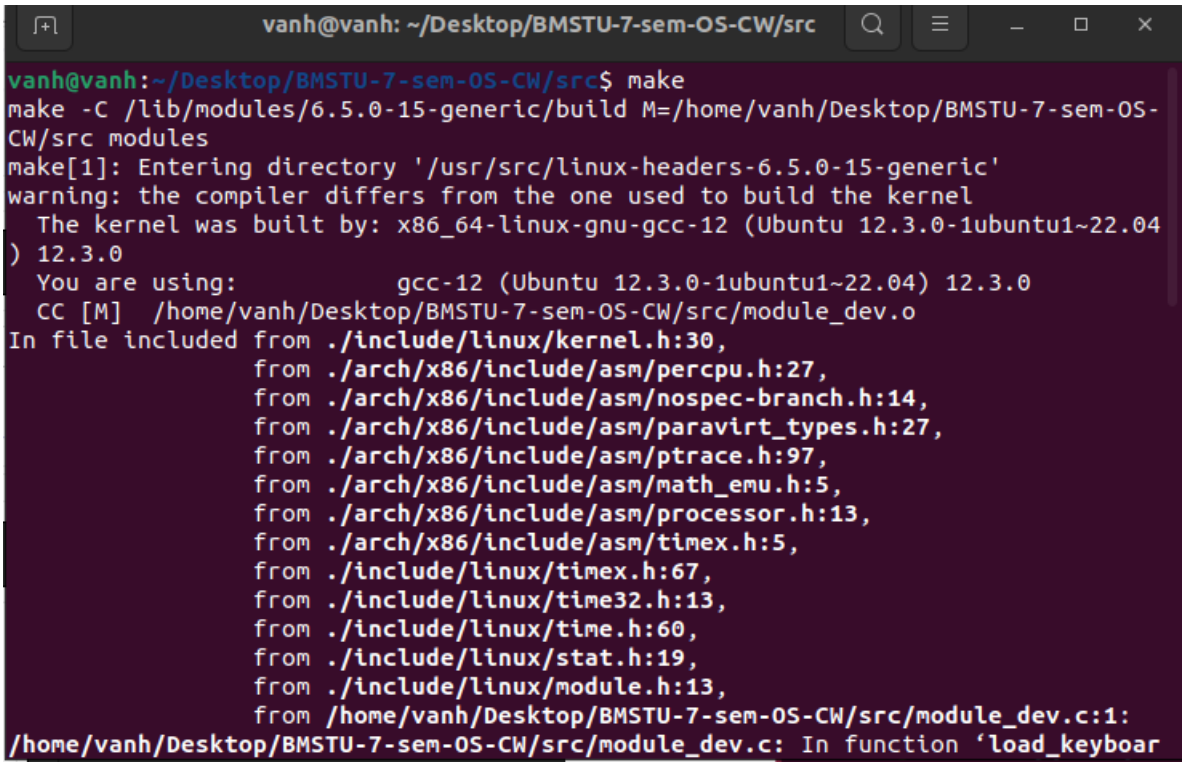
4.1 Системные характеристики

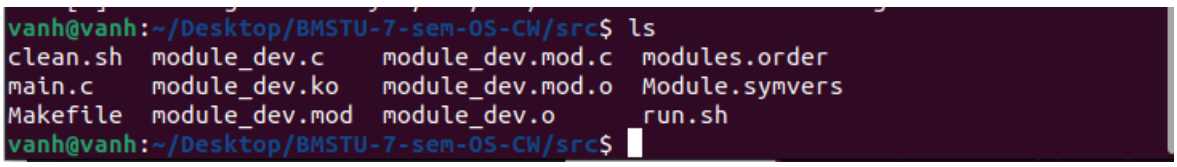
Характеристики виртуальной машины на котором производится эксперимент:

- Операционная система: Ubuntu 22.04 LST;
- Версия ядра: Linux version 6.5.0-15-generic;
- Объем оперативной памяти: 10ГБ;

4.2 Результаты выполнения работы

Запустить Makefile





Виртуальный файл создан

```
vanh@vanh:~/Desktop/BMSTU-7-sem-OS-CW/src$ sudo insmod module_dev.ko
[sudo] password for vanh:
vanh@vanh:~/Desktop/BMSTU-7-sem-OS-CW/src$ cd /proc/module_dir
vanh@vanh:/proc/module_dir$ ls
dev
vanh@vanh:/proc/module_dir$
```

Пользовательская программа обнаруживает события подключения и отключения гарнитуры.

```
Headset connected
Headset disconnected
Headset connected
Headset disconnected
Headset connected
Headset disconnected
```

```
vanh@vanh:~/Desktop/BMSTU-7-sem-OS-CW/src$ sudo dmesg | grep module_dev | tail -
20
[ 2906.683891] module_dev: called open
[ 2906.683897] module_dev: called write 1
[ 2906.683900] module_dev: write msg '1'
[ 2906.683901] module_dev: called release
[ 2911.793145] module_dev: called open
[ 2911.793155] module_dev: called write 1
[ 2911.793159] module_dev: write msg '1'
[ 2911.793551] module_dev: called release
[ 2913.464547] module_dev: called open
[ 2913.464560] module_dev: called write 1
[ 2913.464566] module_dev: write msg '1'
[ 2913.464570] module_dev: called release
[ 2916.745018] module_dev: called open
[ 2916.745029] module_dev: called write 1
[ 2916.745033] module_dev: write msg '1'
[ 2916.745041] module_dev: called release
[ 2917.838688] module_dev: called open
[ 2917.838695] module_dev: called write 1
[ 2917.838698] module_dev: write msg '1'
[ 2917.839073] module_dev: called release
vanh@vanh:~/Desktop/BMSTU-7-sem-OS-CW/src$
```

ЗАКЛЮЧЕНИЕ

В результате выполнения данной курсовой работы был изучен подход динамического написания драйверов под ОС Linux, работа с звуковой системой и устройством ввода (гарнитурой).

Разработана пользовательская программа, обнаруживающая события отключения и подключения гарнитуры и протокол взаимодейсвтия этой программы с модулем ядра.

Разработан модуль ядра, работающий с клавиатурой, который включает и выключает LED индикаторы клавиатуры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. LKM (Loadable Kernel Module) - Национальная библиотека им. Н. Э. Баумана [Электронный ресурс]. Режим доступа: https://ru.bmstu.wiki/LKM_(Loadable_Kernel_Module).

2. Open Sound System [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Open_Sound_System.

3. ALSA System [Электронный ресурс] [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/ALSA.

ПРИЛОЖЕНИЕ А

main.c

```c
#include <alsa/asoundlib.h>
#include <stdio.h>
#include <stdlib.h>

void write_signal_to_proc()
{
    FILE *fp = fopen("/proc/module_dir/dev", "w");
    if (fp == NULL)
        printf("unable to open proc file\n");
    fprintf(fp, "1");
    fclose(fp);
}
snd_pcm_t *pcm_device;
int main()
{
    int count = 1;
    while(1)
    {
        int totalCards = 0;
        int cardNum = -1;
        int err;

        for (;;) {
            if ((err = snd_card_next(&cardNum)) < 0) {
                fprintf(stderr, "Can't get the next card number: %s\n",
                                snd_strerror(err));
                break;
            }
            if (cardNum < 0)
                break;
            ++totalCards;
        }
        if(count < totalCards)
        {
            printf("Headset connected\n");
            write_signal_to_proc();
        }
        else if(count > totalCards)
        {
            printf("Headset disconnected\n");
            write_signal_to_proc();
        }
        count = totalCards;
        snd_config_update_free_global();
    }
}
```

module_dev.c

```c
#include <linux/module.h>
#include <linux/proc_fs.h>
#include <linux/interrupt.h>
#include <linux/timer.h>

/* for led keyboard */
#include <linux/tty.h>            /* For fg_console, MAX_NR_CONSOLES */
#include <linux/kd.h>             /* For KDSETLED */
#include <linux/console_struct.h> /* For vc_cons */
#include <linux/vt_kern.h>

MODULE_AUTHOR("vanh");
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Toggle keyboard LED when '1' is written in file proc");
// Переключить светодиод клавиатуры, когда в файл proc записывается '1'

struct tty_driver *my_driver;

#define ALL_LEDS_ON 0x07
#define RESTORE_LEDS 0xFF

int is_led_on = 0;

void tasklet_fn_toggle_led(unsigned long data)
{
    unsigned long status;

    if (is_led_on)
    {
        is_led_on = 0;
        status = ALL_LEDS_ON;
```

```
32      }
33      else
34      {
35          is_led_on = 1;
36          status = RESTORE_LEDS;
37      }
38      (my_driver->ops->ioctl)(vc_cons[fg_console].d->port.tty, KDSETLED, status);
39
40  }
41
42  struct tty_driver *load_keyboard_led_driver(void)
43  {
44      int i;
45      struct tty_driver *driver;
46
47      printk(KERN_INFO "kbleds: loading\n");
48      printk(KERN_INFO "kbleds: fgconsole is %x\n", fg_console);
49      printk(KERN_INFO "kbleds: MAX_NR_CONSOLES %i", MAX_NR_CONSOLES);
50
51      for (i = 0; i < MAX_NR_CONSOLES; i++)
52      {
53          if (!vc_cons[i].d)
54              break;
55          printk(KERN_INFO "poet_atkm: console[%i/%i] #%i, tty %lx\n", i,
56                 MAX_NR_CONSOLES, vc_cons[i].d->vc_num,
57                 (unsigned long)vc_cons[i].d->port.tty);
58      }
59
60      printk(KERN_INFO "kbleds: finished scanning consoles\n");
61      driver = vc_cons[fg_console].d->port.tty->driver;
62      printk(KERN_INFO "kbleds: tty driver magic %x\n", driver->name);
63
64      return driver;
65  }
66
67  static char msg[8];
68  static ssize_t msg_length;
69  static int is_empty = 1;
70
71  /* elements procfs */
72  struct proc_dir_entry *kmodule_dev, *kmodule_dir;
73  static const char *dev = "dev";
74  static const char *dir = "module_dir";
75
76  DECLARE_TASKLET_OLD(my_tasklet, tasklet_fn_toggle_led);
77
78  int module_dev_open(struct inode *inode, struct file *file)
79  {
80      printk(KERN_INFO "module_dev: called open\n");
81      return 0;
82  }
83
84  int module_dev_release(struct inode *indoe, struct file *file)
85  {
86      printk(KERN_INFO "module_dev: called release\n");
87      return 0;
88  }
89
90  ssize_t module_dev_read(struct file *file, char __user *buf, size_t size, loff_t *offset)
91  {
92      if (is_empty)
93          return 0;
94
95      is_empty = 1;
96
97      printk(KERN_INFO "module_dev: called read\n");
98      copy_to_user(buf, msg, msg_length);
99
100     return msg_length;
101 }
102
103 ssize_t module_dev_write(struct file *file, const char __user *buf, size_t size, loff_t *offset)
104 {
105     printk(KERN_INFO "module_dev: called write %ld\n", size);
106
107     copy_from_user(msg, buf, size);
108     msg_length = size;
109
110     is_empty = 0;
111
112     if (msg[0] == '1')
113         tasklet_schedule(&my_tasklet);
114     printk(KERN_INFO "module_dev: write msg '%s'\n", msg);
115
116     return size;
117 }
118
119 struct proc_ops fops = {
120     .proc_open = module_dev_open,
121     .proc_read = module_dev_read,
```

```
122     .proc_write = module_dev_write,
123     .proc_release = module_dev_release};
124
125 static int __init init_module_dev(void)
126 {
127     printk(KERN_INFO "module_dev: init\n");
128
129     kmodule_dir = proc_mkdir(dir, NULL);
130     kmodule_dev = proc_create(dev, 0666, kmodule_dir, &fops);
131
132     my_driver = load_keyboard_led_driver();
133
134     return 0;
135 }
136
137 static void __exit exit_module_dev(void)
138 {
139     tasklet_kill(&my_tasklet);
140
141     remove_proc_entry(dev, kmodule_dir);
142     remove_proc_entry(dir, NULL);
143
144     printk(KERN_INFO "module_dev: exit\n");
145 }
146
147 module_init(init_module_dev);
148 module_exit(exit_module_dev);
```