|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОМУ ПРОЕКТУ***

***НА ТЕМУ:***

***«Загружаемый модуль ядра для мониторинга действий пользователя»***

Студент ИУ7-75Б **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  Т.М.Оберган

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  Н.Ю.Рязанова

(Подпись, дата) (И.О.Фамилия)

*2020 г.*

**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7

(Индекс)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ И. В. Рудаков

(И.О.Фамилия)

« \_\_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**

**на выполнение курсового проекта**

по дисциплине Операционные системы

Студент группы ИУ-75Б

Оберган Татьяна Максимовна

(Фамилия, имя, отчество)

Тема курсового проекта Загружаемый модуль ядра для мониторинга действий пользователя

Направленность КП (учебный, исследовательский, практический, производственный, др.)

Учебная

Источник тематики (кафедра, предприятие, НИР) Кафедра

График выполнения проекта: 25% к 4 нед., 50% к 7 нед., 75% к 11 нед., 100% к 14 нед.

***Задание*** Разработать загружаемый модуль ядра для мониторинга различных действий пользователя. Предоставить возможность просмотра и выгрузку лога отслеживаемых событий.

***Оформление курсового проекта:***

Расчетно-пояснительная записка на 25-35 листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

На защиту работы должна быть представлена презентация, состоящая из 10-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО.

Дата выдачи задания « 22 » сентября 2020 г.

**Руководитель курсового проекта**  Рязанова Н. Ю.

(Подпись, дата) (И.О.Фамилия)

**Студент**  Оберган Т.М.

(Подпись, дата) (И.О.Фамилия)

Оглавление

[Введение 4](#_Toc59507387)

[1. Аналитическая часть 5](#_Toc59507388)

[1.1 Формализация задачи 5](#_Toc59507389)

[1.2 Анализ способов решения поставленной задачи 5](#_Toc59507390)

[1.3 /dev/input/event 6](#_Toc59507391)

[1.4 Keyboard notifier 6](#_Toc59507392)

[1.5 Драйвер устройства 7](#_Toc59507393)

[1.6 Обработчик прерывания 9](#_Toc59507394)

[1.7 sys\_call\_table 10](#_Toc59507395)

[1.8 Передача данных в пространство пользователя 11](#_Toc59507396)

[1.9 Выводы из аналитического раздела 12](#_Toc59507397)

[2. Конструкторская часть 13](#_Toc59507398)

[2.1 Требования к программе 13](#_Toc59507399)

[2.2 Драйвер USB мыши 13](#_Toc59507400)

[2.3 Модуль - логгер 14](#_Toc59507401)

[2.4 Схемы, демонстрирующие работу модуля 16](#_Toc59507402)

[2.5 Вывод 18](#_Toc59507403)

[3. Технологическая часть 19](#_Toc59507404)

[3.1 Выбор языка программирования и среды разработки 19](#_Toc59507405)

[3.2 Модуль – логгер 19](#_Toc59507406)

[3.3 Драйвер мыши 22](#_Toc59507407)

[3.4 Makefile 26](#_Toc59507408)

[3.5 Взаимодействие с модулями 26](#_Toc59507409)

[3.6 Вывод 29](#_Toc59507410)

[Заключение 30](#_Toc59507411)

[Список использованной литературы 31](#_Toc59507412)

# Введение

Мышь и клавиатура - интерактивные средства ввода-вывода. Прерывания от мыши возникают очень часто.

Информацию о событиях клавиатуры и мыши можно использовать как во вред, так и во благо: для тестирования приложений и интерфейса, для слежения за пользователем. При помощи подобных программ можно получить персональные данные: пароли, номера банковских карт, и т.д.

Целью данной работы является реализация программного обеспечения для перехвата сообщений USB мыши и клавиатуры и их последующего журналирования.

# 1. Аналитическая часть

В данном разделе будет формализована задача и рассмотрены способы ее решения.

## 1.1 Формализация задачи

Целью данного курсового проекта является реализация программного обеспечения, фиксирующего события в системе, инициирующиеся действиями пользователя – взаимодействие с мышью и клавиатурой.

Программное обеспечение должно обеспечивать перехват нажатий кнопок, перемещения мыши и движения колесика, фиксировать эту информацию для того, чтобы в дальнейшем было возможно произвести анализ этой информации.

Для достижения поставленной цели необходимо решить следующие задачи:

* проанализировать способы перехвата сообщений от мыши и клавиатуры;
* проанализировать структуру драйвера мыши;
* проанализировать методы передачи информации из модулей ядра в пространство пользователя;
* спроектировать и реализовать модуль ядра;

## 1.2 Анализ способов решения поставленной задачи

Существует несколько способов решения поставленной задачи:

* чтение информации из системного файла устройства “/dev/input/event\*”[1];
* использование keyboard notifier;
* разработка драйвера устройства в виде загружаемого модуля ядра;
* установка на линию прерывания нескольких обработчиков прерываний;
* изменение соответствующих указателей в sys\_call\_table.

## 1.3 /dev/input/event

Основными компонентами подсистемы ввода-вывода являются драйверы, управляющие внешними устройствами, и файловая система.

Драйвер взаимодействует, с одной стороны, с модулями ядра ОС, а с другой стороны — с контроллерами внешних устройств. Драйверы таких устройств, как мышь, клавиатура, и др. генерируют события, которые можно посмотреть в директории “/dev/input”. Например, чтобы проверить, что мышь эмулирована, необходимо ввести в консоли «cat /dev/input/mouse0», при движении мышью на экране должны появляться символы. [2]

Листинг 1.3.1 – структура input\_event [2]

**struct** input\_event {

**struct** timeval time; // время возникновения события

**unsigned** **short** type; // тип события (Например: EV\_REL, EV\_KEY)

**unsigned** **short** code; // код события (Например: REL\_X, KEY\_BACKSPACE)

**unsigned** **int** value; // статус события (Например для EV\_KEY: 0-release, 1-keypress, 2-autorepeat)

};

Преимуществом данного подхода является простота реализации, однако, ввиду того что считывание данных подобным образом возможно реализовать в пространстве пользователя, данный способ не подходит для курсовой работы по операционным системам.

## 1.4 Keyboard notifier

При помощи вызова register\_keyboard\_notifier можно “подписаться” на события клавиатуры, в качестве аргумента передается структура notifier\_block. В этой структуре поле notifier\_call – указатель на функцию обработки поступающего события. Описание структур и заголовки функций предоставлены на листингах 1.4.1 и 1.4.2.

Листинг 1.4.1 – содержимое файла notifier.h [3]

**typedef** **int** (\*notifier\_fn\_t)(**struct** notifier\_block \*nb,

**unsigned** **long** action, **void** \*data);

**struct** notifier\_block {

notifier\_fn\_t notifier\_call;

**struct** notifier\_block \_\_rcu \*next;

**int** priority;

};

Листинг 1.4.2 – содержимое файла keyboard.h [4]

**struct** notifier\_block;

**extern** **unsigned** **short** \*key\_maps[MAX\_NR\_KEYMAPS];

**extern** **unsigned** **short** plain\_map[NR\_KEYS];

**struct** keyboard\_notifier\_param {

**struct** vc\_data \*vc; /\* VC on which the keyboard press was done \*/

**int** down; /\* Pressure of the key? \*/

**int** shift; /\* Current shift mask \*/

**int** ledstate; /\* Current led state \*/

**unsigned** **int** value; /\* keycode, unicode value or keysym \*/

};

**extern** **int** register\_keyboard\_notifier(**struct** notifier\_block \*nb);

**extern** **int** unregister\_keyboard\_notifier(**struct** notifier\_block \*nb);

Данный подход подразумевает написание модуля ядра, внутри которого будут получены и обработаны события клавиатуры.

## 1.5 Драйвер устройства

Класс HID (Human Interface Device) драйверов предназначен для управления различными типами внешних устройств. Если устройство поддерживает HID интерфейс, т.е. написано в соответствии с его спецификацией, то его подключение не требует разработки нового драйвера. Однако при необходимости подключения устройства, не отвечающего HID-спецификации, необходим соответствующий HID-драйвер. [5]

К классу USB HID принадлежат все устройства для взаимодействия с пользователем — клавиатуры, мыши, джойстики и т.д.

Регистрация USB-драйвера подразумевает:

1. заполнение структуры usb\_driver;
2. регистрация структуры в системе.

В ОС Linux код драйвера находится в файле usbmouse.c, а структура usb\_driver описана в includr/linux/usb.h.

Листинг 1.5.1 – структуры usb\_driver и usbdrv\_wrap (файл usb.h) [6]

/\*\*

 \* struct usbdrv\_wrap - wrapper for driver-model structure

 \* @driver: The driver-model core driver structure.

 \* @for\_devices: Non-zero for device drivers, 0 for interface drivers.

 \*/

**struct** usbdrv\_wrap {

**struct** device\_driver driver;

**int** for\_devices;

};

**struct** usb\_driver {

**const** **char** \*name;

**int** (\*probe) (**struct** usb\_interface \*intf,

**const** **struct** usb\_device\_id \*id);

**void** (\*disconnect) (**struct** usb\_interface \*intf);

// ...

**const** **struct** usb\_device\_id \*id\_table;

// ...

**struct** usbdrv\_wrap drvwrap;

// ...

};

**Name** – это имя драйвера, должно быть уникальным среди USB-драйверов в ядре и таким же как имя модуля.

**id\_table** – это массив структур usb\_device\_id, который содержит список всех типов USB-устройств, которые обслуживает драйвер. В самом простом случае каждый элемент id\_table[i], который определяет интерфейс подключаемого устройства, содержит пару идентификаторов: идентификатор производителя и устройства.

Поле **drvwrap** – структура usbdrv\_wrap, которая является оберткой для device\_driver, которая говорит о том, что usb\_driver унаследован от device\_driver.

**probe** и **disconnect** – это функции обратного вызова (callbacks), вызываемые системой в контексте потока ядра USB-хаба. Probe() является точкой входа драйвера, которая инициализирует и регистрирует другие точки входа, она будет вызвана для каждого устройства, если список id\_table пуст, или только для тех устройств, которые соответствуют параметрам, перечисленным в списке.

Один зарегистрированный драйвер может "подключать" несколько устройств. Для установления связи устройства и драйвера система вызывает функцию драйвера probe(), которой передает 2 параметра:

**static** **int** usb\_mouse\_probe(**struct** usb\_interface \*intf, **const** **struct** usb\_device\_id \*id);

**interface** – это интерфейс USB-устройства. Обычно USB-драйвер взаимодействует не с устройством напрямую, а с его интерфейсом. **id** - содержит информацию об устройстве. Если функция возвращает 0, то устройство успешно зарегистрировано, иначе - система попытается "привязать" устройство к какому-нибудь другому драйверу.

Для отключения устройства от драйвера система вызывает функцию disconnect, которой передается один параметр - интерфейс:

**static** **void** usb\_mouse\_disconnect(**struct** usb\_interface \*intf);

В общем случае, в функции probe для каждого подключаемого устройства выделяется структура в памяти, заполняется, затем регистрируется, например, символьное устройство, и проводится регистрация устройства в sysfs.

При установке собственного драйвера сначала необходимо выгрузить модуль usbhid, который автоматически регистрирует все стандартные драйверы в системе. Данный модуль устанавливает стандартный драйвер мыши и не позволяет установить свой. Если драйвер установлен, то его можно увидеть в ядре в sysfs, указав путь /sys/bus/usb/drivers/. [5]

## 1.6 Обработчик прерывания

При возникновении аппаратного прерывания вызванный обработчик запрещает прерывания на локальном процессоре. В результате другая работа выполняться не может. Такая ситуация в системе должна быть краткосрочной. Быстрые прерывания блокируют все другие прерывания, во время длинных IRQ могут обрабатываться другие прерывания (но не от того же устройства). Медленные прерывания разбивают на две части: исполняемую сразу при возникновении аппаратного прерывания и работу, которая может быть отложена на некоторое время. Эти части обработчиков прерываний получили название "верхняя" и "нижняя" половины. [5]

“Верхняя половина“ занимается чтением и сохранением в буфере данных или передачей их из буфера в регистры контроллера и завершается постановкой “нижней половины“ в очередь на выполнение и разрешением прерываний. [7]

“Нижняя половина“, как правило, выполняется сразу после завершения “верхней“ и запускает все необходимые операции причем ей доступно всё то, что доступно обычным модулям ядра. Выполняется отдельным потоком ядра, имеет более низкий уровень приоритета и является отложенным действием, которое может быть прервано во время его выполнения. [7]

Существует несколько способов реализации “нижней половины“ обработчика: гибкое прерывание (softirq), тасклет (tasklet) и очереди работ (workqueue). [5]

## 1.7 sys\_call\_table

Когда процесс запрашивает какую-либо услугу, используется механизм системных вызовов. Чтобы исполнить системный вызов, процесс заполняет регистры микропроцессора соответствующими значениями и выполняет специальную инструкцию, которая производит переход в предопределенное место в пространстве ядра. Микропроцессор воспринимает это как переход из ограниченного пользовательского режима в защищенный режим ядра.

Если необходимо изменить поведение некоторого системного вызова, то первое, что необходимо сделать – это написать функцию, которая выполняет требуемые действия и вызывает первоначальную функцию, реализующую системный вызов, затем – изменить указатель в sys\_call\_table так, чтобы он указывал на новую функцию. Поскольку модуль впоследствии может быть выгружен, то следует предусмотреть восстановление системы в ее первоначальное состояние, чтобы не оставлять ее в нестабильном состоянии. [8]

С целью предотвращения потенциальной опасности, связанной с подменой адресов системных вызовов, **ядро более не экспортирует sys\_call\_table** [8]. Поэтому, надлежит наложить "заплату" на ядро.

Подмена системных вызовов потенциально опасна и может стать причиной потери данных.

## 1.8 Передача данных в пространство пользователя

Поставленная задача подразумевает под собой передачу данных из пространства ядра в пространство пользователя. Для этой цели можно использовать файловую систему procfs. Она предоставляет все ресурсы для реализации интерфейса между пространством пользователя и пространством ядра. Часто используется в исходных кодах Linux.

Листинг 1.8.1 – основные поля структуры file\_operations (/include/linux/fs.h) [9]

**struct** file\_operations {

**struct** module \*owner;

loff\_t (\*llseek) (**struct** file \*, loff\_t, **int**);

ssize\_t (\*read) (**struct** file \*, **char** \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*write) (**struct** file \*, **const** **char** \_\_user \*, size\_t, loff\_t \*);

ssize\_t (\*read\_iter) (**struct** kiocb \*, **struct** iov\_iter \*);

ssize\_t (\*write\_iter) (**struct** kiocb \*, **struct** iov\_iter \*);

// ...

**int** (\*open) (**struct** inode \*, **struct** file \*);

**int** (\*flush) (**struct** file \*, fl\_owner\_t id);

**int** (\*release) (**struct** inode \*, **struct** file \*);

// ...

}

Copy\_to\_user – функция копирования данных из пространства ядра в пространство пользователя. Возвращает количество нескопированных байт, т.е. 0 – успешное завершение функции. [10]

Листинг 1.8.2 – заголовок функции copy\_to\_user

#include <asm/uaccess.h>

**int** copy\_to\_user(**void** \*dst, **const** **void** \*src, **unsigned** **int** size);

## 1.9 Выводы из аналитического раздела

В данном разделе была формализована задача и рассмотрены способы ее решения. Выяснилось, что использование /dev/input/event и sys\_call\_table не подходят в рамках данной курсовой работы. Было принято решение использовать keyboard notifier для слежения за клавиатурой пользователя и написание драйвера для слежения за мышью.

# 2. Конструкторская часть

В данном разделе будут рассмотрены требования к программе, основные сведения о реализуемых модулях, предоставлены схемы, описывающие работу модуля - логгера.

## 2.1 Требования к программе

Необходимо реализовать загружаемый модуль ядра, который будет получать данные о событиях мыши и клавиатуры и предоставлять доступ к журналу этих событий через procfs.

Также необходимо реализовать драйвер usb мыши, который будет отсылать данные в модуль – логгер.

## 2.2 Драйвер USB мыши

За основу драйвера мыши стоит взять USB HID драйвер мыши, который можно найти в исходном коде linux: /drivers/hid/usbhid/usbmouse.c.

Внутри этого драйвера, сообщения, отправленные устройством, обрабатываются функцией usb\_mouse\_irq. Необходимо поместить в эту процедуру вызов экспортируемой функции из модуля-логгера, в которую будут передаваться данные, пришедшие от устройства.

Листинг 2.2.1 – заголовок usb\_mouse\_irq

**static** **void** usb\_mouse\_irq(**struct** urb \*urb);

Листинг 2.2.2 – заголовок экспортируемой функции

**extern** **int** send\_mouse\_coordinates(**char** buttons, **char** dx, **char** dy, **char** wheel);

В листинге 2.2.2 предоставлен заголовок экспортируемой функции, вызов которой должен быть помещен в функцию usb\_mouse\_irq.

**Buttons** – состояние кнопок мыши, стандартно: 0 – нейтральное; 1 – нажата левая, 2 – правая кнопка мыши.

**Dx, dy** – смещение мыши по осям x и у соответственно.

**Wheel** – смещение колесика мыши.

В драйвере мыши эта информация хранится в data. На листинге 2.2.3 подробнее показан способ получения этой информации.

Листинг 2.2.3 – получение информации о событии мыши

**struct** usb\_mouse \*mouse = urb->context;

**signed** **char** \*data = mouse->data;

**char** buttons = data[0];

**char** dx = data[1];

**char** dy = data[2];

**char** wheel = data[3];

## 2.3 Модуль - логгер

Для хранения журнала выделяется массив символов. При записи в лог очень важно не допускать переполнение. Для этих целей вводится переменная my\_log\_len, которая хранит количество занятых символов в логе на данный момент. При полном заполнении журнала, он сбрасывается и отсчет начинается с 0.

Листинг 2.3.1 – массив символов для журналирования

**static** **int** my\_log\_len;

**static** **char** my\_log[MY\_LOG\_SIZE];

Как уже было выяснено выше, загружаемый модуль ядра для журналирования должен реализовывать экспортируемую драйвером мыши функцию. Полученные данные от драйвера должны сохраняться в лог.

Листинг 2.3.2 – функция обработки события мыши

**extern** **int** send\_mouse\_coordinates(**char** buttons, **char** dx, **char** dy, **char** wheel);

Также необходимо реализовать функцию keylogger\_notify, указатель на которую устанавливается в структуру notifier\_block, которая используется в качестве аргументов в функциях register\_keyboard\_notifier и unregister\_keyboard\_notifier.

Листинг 2.3.3 – заполнение структуры notifier\_block

**static** **struct** notifier\_block keylogger\_nb = {

.notifier\_call = keylogger\_notify

};

Также необходимо заполнить структуру file\_operations, самое нужное поле в рамках поставленной задачи – read. В функции procfs\_read нужно использовать функцию copy\_to\_user для копирования журнала модуля ядра в пространство пользователя.

Листинг 2.3.4 – заполнение структуры file\_operations

**int** procfs\_open(**struct** inode \*inode, **struct** file \*file);

**static** ssize\_t procfs\_read(**struct** file \*filp, **char** \*buffer, size\_t length, loff\_t \* offset);

**static** ssize\_t procfs\_write(**struct** file \*file, **const** **char** \*buffer, size\_t length, loff\_t \* off);

**int** procfs\_close(**struct** inode \*inode, **struct** file \*file);

**static** **const** **struct** file\_operations fops = {

.owner = THIS\_MODULE,

.read = procfs\_read,

.write = procfs\_write,

.open = procfs\_open,

.release = procfs\_close,

};

Листинг 2.3.5 – ключевые моменты функций инициализации и завершения работы модуля

**static** **int** \_\_init procInit( **void** )

{

// ...

our\_proc\_file = proc\_create(DEVICE\_NAME, 0644 , NULL, &fops);

**if** (!our\_proc\_file)

**return** -ENOMEM; // out of memory

register\_keyboard\_notifier(&keylogger\_nb);

// ...

**return** 0;

}

**static** **void** \_\_exit procExit( **void** )

{

// ...

unregister\_keyboard\_notifier(&keylogger\_nb);

remove\_proc\_entry(DEVICE\_NAME, NULL);

// ...

}

## 2.4 Схемы, демонстрирующие работу модуля

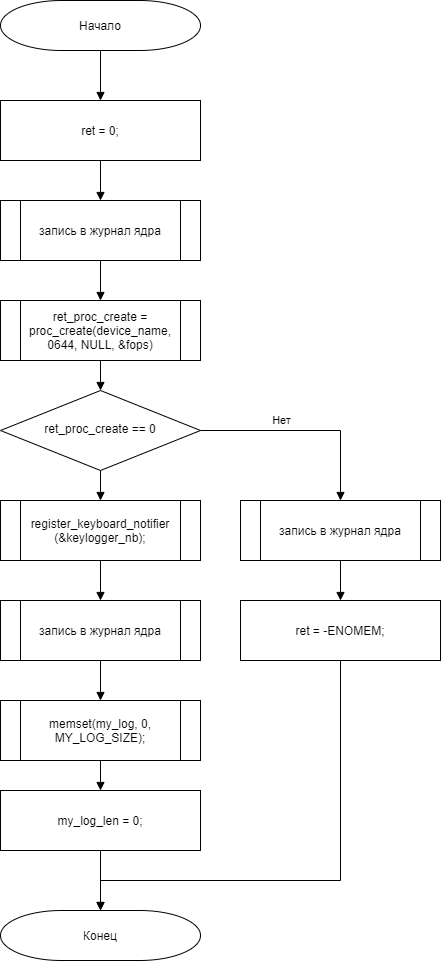
****

Рис. 2.4.1 – схема инициализации работы модуля

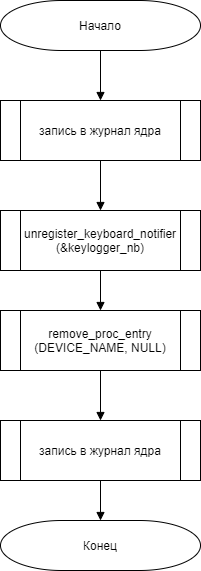


Рис. 2.4.2 – схема завершения работы модуля

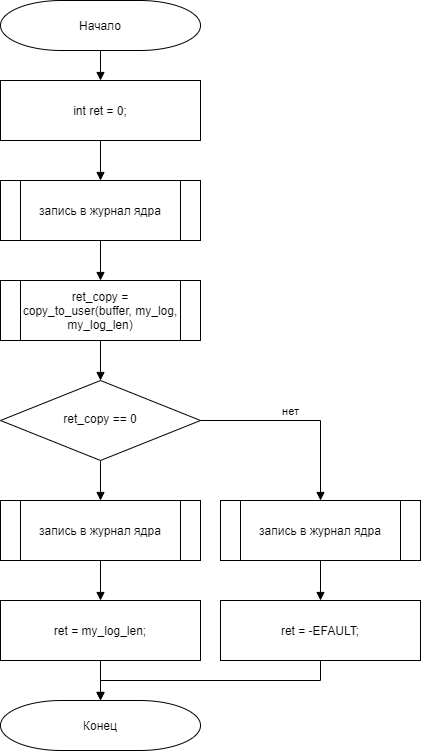


Рис. 2.4.3 – схема обработки чтения из /proc

## 2.5 Вывод

В данном разделе были рассмотрены требования к программе, основные сведения о реализуемых модулях, предоставлены схемы, описывающие работу модуля - логгера.

# 3. Технологическая часть

В данном разделе будет предоставлен листинг реализованных модулей и проведена апробация.

## 3.1 Выбор языка программирования и среды разработки

В качестве языка программирования был выбран C.

В качестве среды разработки была выбрана «Visual Studio Code» т.к. она бесплатна, кроссплатформенная, имеет множество плагинов для расширения функционала.

## 3.2 Модуль – логгер

Листинг 3.2.1 – файл my\_logger.c

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/init.h>

#include <linux/keyboard.h>

#include <linux/proc\_fs.h>

#include <asm/uaccess.h>

#include "my\_logger.h"

#define MODULE\_INFO\_PREFIX "myLogger"

**extern** **int** send\_mouse\_coordinates(**char** buttons, **char** dx, **char** dy, **char** wheel);

#define DEVICE\_NAME "my\_logger"

#define MY\_LOG\_SIZE 2048

**static** **struct** proc\_dir\_entry\* our\_proc\_file;

**static** **int** my\_log\_len;

**static** **char** my\_log[MY\_LOG\_SIZE];

**static** **int** isShiftKey = 0;

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Obergan T.M");

MODULE\_DESCRIPTION("Logging mouse and keyboard");

/\* keylogger\_notify start \*/

**int** keylogger\_notify(**struct** notifier\_block \*nblock, **unsigned** **long** code, **void** \*\_param){

**struct** keyboard\_notifier\_param \*param = \_param;

**char** buf[128];

**int** len;

**if** ( code == KBD\_KEYCODE )

{

**if**( param->value==42 || param->value==54 )

{

**if**( param->down )

isShiftKey = 1;

**else**

isShiftKey = 0;

**return** NOTIFY\_OK;

}

**if**( param->down )

{

// если не предусмотреть выход за границы массива,

// то система может попросту зависнуть

**if** (param->value > MyKeysMax)

sprintf(buf, "keyboard: id-%d\n", param->value);

**if**( isShiftKey == 0 )

sprintf(buf, "keyboard: %s\n", MyKeys[param->value]);

**else**

sprintf(buf, "keyboard: shift + %s\n", MyKeys[param->value]);

len = strlen(buf);

**if**(len + my\_log\_len >= MY\_LOG\_SIZE)

{

memset(my\_log, 0, MY\_LOG\_SIZE);

my\_log\_len = 0;

}

strcat(my\_log, buf);

my\_log\_len += len;

}

}

**return** NOTIFY\_OK;

}

**static** **struct** notifier\_block keylogger\_nb = {

.notifier\_call = keylogger\_notify

};

/\* keylogger\_notify end \*/

/\* procf start \*/

**int** procfs\_open(**struct** inode \*inode, **struct** file \*file)

{

printk(KERN\_INFO "%s in procfs\_open\n", MODULE\_INFO\_PREFIX);

try\_module\_get(THIS\_MODULE);

**return** 0;

}

**static** ssize\_t procfs\_read(**struct** file \*filp, **char** \*buffer, size\_t length, loff\_t \* offset)

{

**static** **int** ret = 0;

printk(KERN\_INFO "%s in procfs\_read\n", MODULE\_INFO\_PREFIX);

**if** (ret)

ret = 0;

**else**

{

**if** ( raw\_copy\_to\_user(buffer, my\_log, my\_log\_len) )

**return** -EFAULT;

printk(KERN\_INFO "%s read %lu bytes\n", MODULE\_INFO\_PREFIX, my\_log\_len);

ret = my\_log\_len;

}

**return** ret;

}

**static** ssize\_t procfs\_write(**struct** file \*file, **const** **char** \*buffer, size\_t length, loff\_t \* off)

{

printk(KERN\_INFO "%s in procfs\_write\n", MODULE\_INFO\_PREFIX);

**return** 0;

}

**int** procfs\_close(**struct** inode \*inode, **struct** file \*file)

{

printk(KERN\_INFO "%s in procfs\_close\n", MODULE\_INFO\_PREFIX);

module\_put(THIS\_MODULE);

**return** 0;

}

**static** **const** **struct** file\_operations fops = {

.owner = THIS\_MODULE,

.read = procfs\_read,

.write = procfs\_write,

.open = procfs\_open,

.release = procfs\_close,

};

/\* procf end \*/

**static** **int** \_\_init procInit( **void** )

{

printk(KERN\_INFO "%s in init\n", MODULE\_INFO\_PREFIX);

our\_proc\_file = proc\_create(DEVICE\_NAME, 0644 , NULL, &fops);

**if** (!our\_proc\_file)

{

printk(KERN\_INFO "%s proc\_create %s failed\n", MODULE\_INFO\_PREFIX, DEVICE\_NAME);

**return** -ENOMEM;

}

register\_keyboard\_notifier(&keylogger\_nb);

printk(KERN\_INFO "%s logger registered\n", MODULE\_INFO\_PREFIX);

memset(my\_log, 0, MY\_LOG\_SIZE);

my\_log\_len = 0;

**return** 0;

}

**static** **void** \_\_exit procExit( **void** )

{

printk(KERN\_INFO "%s in exit\n", MODULE\_INFO\_PREFIX);

unregister\_keyboard\_notifier(&keylogger\_nb);

remove\_proc\_entry(DEVICE\_NAME, NULL);

printk(KERN\_INFO "%s unregistered\n", MODULE\_INFO\_PREFIX);

}

**extern** **int** send\_mouse\_coordinates(**char** buttons, **char** dx, **char** dy, **char** wheel)

{

printk(KERN\_INFO "%s received send\_mouse\_coordinates %d %d %d %d\n", MODULE\_INFO\_PREFIX, buttons, dx, dy, wheel);

**char** buf[32];

**int** len;

sprintf(buf, "mouse: %d %d %d %d \n", buttons, dx, dy, wheel);

len = strlen(buf);

**if**(len + my\_log\_len >= MY\_LOG\_SIZE)

{

memset(my\_log, 0, MY\_LOG\_SIZE);

my\_log\_len = 0;

}

strcat(my\_log, buf);

my\_log\_len += len;

**return** 0;

}

EXPORT\_SYMBOL(send\_mouse\_coordinates);

module\_init(procInit);

module\_exit(procExit);

## 3.3 Драйвер мыши

Листинг 3.3.1 – файл my\_usb\_mouse\_driver.c

//основа кода взята из https://github.com/torvalds/linux/blob/master/drivers/hid/usbhid/usbmouse.c

#include <linux/kernel.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/init.h>

#include <linux/usb/input.h>

#include <linux/hid.h>

/\*

 \* Version Information

 \*/

#define DRIVER\_VERSION "v1.6"

#define DRIVER\_AUTHOR "Obergan T.M"

#define DRIVER\_DESC "USB HID Boot Protocol mouse driver"

#define MODULE\_INFO\_PREFIX "usbmouse:"

**extern** **int** send\_mouse\_coordinates(**char** buttons, **char** dx, **char** dy, **char** wheel);

MODULE\_AUTHOR(DRIVER\_AUTHOR);

MODULE\_DESCRIPTION(DRIVER\_DESC);

MODULE\_LICENSE("GPL");

**struct** usb\_mouse {

**char** name[128];

**char** phys[64];

**struct** usb\_device \*usbdev;

**struct** input\_dev \*dev;

**struct** urb \*irq;

**signed** **char** \*data;

dma\_addr\_t data\_dma;

};

**static** **void** usb\_mouse\_irq(**struct** urb \*urb)

{

**struct** usb\_mouse \*mouse = urb->context;

**signed** **char** \*data = mouse->data;

**struct** input\_dev \*dev = mouse->dev;

**int** status;

**if**(send\_mouse\_coordinates(data[0], data[1], data[2], data[3]) != 0)

printk(KERN\_INFO "%s usb\_mouse\_irq can't send data\n", MODULE\_INFO\_PREFIX);

**else**

printk(KERN\_INFO "%s usb\_mouse\_irq success\n", MODULE\_INFO\_PREFIX);

**switch** (urb->status) {

**case** 0: /\* success \*/

**break**;

**case** -ECONNRESET: /\* unlink \*/

**case** -ENOENT:

**case** -ESHUTDOWN:

**return**;

/\* -EPIPE: should clear the halt \*/

**default**: /\* error \*/

goto resubmit;

}

input\_report\_key(dev, BTN\_LEFT, data[0] & 0x01);

input\_report\_key(dev, BTN\_RIGHT, data[0] & 0x02);

input\_report\_key(dev, BTN\_MIDDLE, data[0] & 0x04);

input\_report\_key(dev, BTN\_SIDE, data[0] & 0x08);

input\_report\_key(dev, BTN\_EXTRA, data[0] & 0x10);

input\_report\_rel(dev, REL\_X, data[1]);

input\_report\_rel(dev, REL\_Y, data[2]);

input\_report\_rel(dev, REL\_WHEEL, data[3]);

input\_sync(dev);

resubmit:

status = usb\_submit\_urb (urb, GFP\_ATOMIC);

**if** (status)

dev\_err(&mouse->usbdev->dev,

"can't resubmit intr, %s-%s/input0, status %d\n",

mouse->usbdev->bus->bus\_name,

mouse->usbdev->devpath, status);

}

**static** **int** usb\_mouse\_open(**struct** input\_dev \*dev)

{

**struct** usb\_mouse \*mouse = input\_get\_drvdata(dev);

printk(KERN\_INFO "usbmouse: in usb\_mouse\_open\n");

mouse->irq->dev = mouse->usbdev;

**if** (usb\_submit\_urb(mouse->irq, GFP\_KERNEL))

**return** -EIO;

**return** 0;

}

**static** **void** usb\_mouse\_close(**struct** input\_dev \*dev)

{

**struct** usb\_mouse \*mouse = input\_get\_drvdata(dev);

printk(KERN\_INFO "usbmouse: in usb\_mouse\_close\n");

usb\_kill\_urb(mouse->irq);

}

**static** **int** usb\_mouse\_probe(**struct** usb\_interface \*intf, **const** **struct** usb\_device\_id \*id)

{

printk(KERN\_INFO "usbmouse: in usb\_mouse\_probe\n");

**struct** usb\_device \*dev = interface\_to\_usbdev(intf);

**struct** usb\_host\_interface \*interface;

**struct** usb\_endpoint\_descriptor \*endpoint;

**struct** usb\_mouse \*mouse;

**struct** input\_dev \*input\_dev;

**int** pipe, maxp;

**int** error = -ENOMEM;

interface = intf->cur\_altsetting;

**if** (interface->desc.bNumEndpoints != 1)

**return** -ENODEV;

endpoint = &interface->endpoint[0].desc;

**if** (!usb\_endpoint\_is\_int\_in(endpoint))

**return** -ENODEV;

pipe = usb\_rcvintpipe(dev, endpoint->bEndpointAddress);

maxp = usb\_maxpacket(dev, pipe, usb\_pipeout(pipe));

mouse = kzalloc(**sizeof**(**struct** usb\_mouse), GFP\_KERNEL);

input\_dev = input\_allocate\_device();

**if** (!mouse || !input\_dev)

goto fail1;

mouse->data = usb\_alloc\_coherent(dev, 8, GFP\_ATOMIC, &mouse->data\_dma);

**if** (!mouse->data)

goto fail1;

mouse->irq = usb\_alloc\_urb(0, GFP\_KERNEL);

**if** (!mouse->irq)

goto fail2;

mouse->usbdev = dev;

mouse->dev = input\_dev;

**if** (dev->manufacturer)

strlcpy(mouse->name, dev->manufacturer, **sizeof**(mouse->name));

**if** (dev->product) {

**if** (dev->manufacturer)

strlcat(mouse->name, " ", **sizeof**(mouse->name));

strlcat(mouse->name, dev->product, **sizeof**(mouse->name));

}

**if** (!strlen(mouse->name))

snprintf(mouse->name, **sizeof**(mouse->name),

"USB HIDBP Mouse %04x:%04x",

le16\_to\_cpu(dev->descriptor.idVendor),

le16\_to\_cpu(dev->descriptor.idProduct));

usb\_make\_path(dev, mouse->phys, **sizeof**(mouse->phys));

strlcat(mouse->phys, "/input0", **sizeof**(mouse->phys));

input\_dev->name = mouse->name;

input\_dev->phys = mouse->phys;

usb\_to\_input\_id(dev, &input\_dev->id);

input\_dev->dev.parent = &intf->dev;

input\_dev->evbit[0] = BIT\_MASK(EV\_KEY) | BIT\_MASK(EV\_REL);

input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] = BIT\_MASK(BTN\_LEFT) |

BIT\_MASK(BTN\_RIGHT) | BIT\_MASK(BTN\_MIDDLE);

input\_dev->relbit[0] = BIT\_MASK(REL\_X) | BIT\_MASK(REL\_Y);

input\_dev->keybit[BIT\_WORD(BTN\_MOUSE)] |= BIT\_MASK(BTN\_SIDE) |

BIT\_MASK(BTN\_EXTRA);

input\_dev->relbit[0] |= BIT\_MASK(REL\_WHEEL);

input\_set\_drvdata(input\_dev, mouse);

input\_dev->open = usb\_mouse\_open;

input\_dev->close = usb\_mouse\_close;

usb\_fill\_int\_urb(mouse->irq, dev, pipe, mouse->data,

(maxp > 8 ? 8 : maxp),

usb\_mouse\_irq, mouse, endpoint->bInterval);

mouse->irq->transfer\_dma = mouse->data\_dma;

mouse->irq->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

error = input\_register\_device(mouse->dev);

**if** (error)

goto fail3;

usb\_set\_intfdata(intf, mouse);

**return** 0;

fail3:

usb\_free\_urb(mouse->irq);

fail2:

usb\_free\_coherent(dev, 8, mouse->data, mouse->data\_dma);

fail1:

input\_free\_device(input\_dev);

kfree(mouse);

**return** error;

}

**static** **void** usb\_mouse\_disconnect(**struct** usb\_interface \*intf)

{

printk(KERN\_INFO "usbmouse: in usb\_mouse\_disconnect\n");

**struct** usb\_mouse \*mouse = usb\_get\_intfdata (intf);

usb\_set\_intfdata(intf, NULL);

**if** (mouse) {

usb\_kill\_urb(mouse->irq);

input\_unregister\_device(mouse->dev);

usb\_free\_urb(mouse->irq);

usb\_free\_coherent(interface\_to\_usbdev(intf), 8, mouse->data, mouse->data\_dma);

kfree(mouse);

}

}

**static** **const** **struct** usb\_device\_id usb\_mouse\_id\_table[] = {

{ USB\_INTERFACE\_INFO(USB\_INTERFACE\_CLASS\_HID, USB\_INTERFACE\_SUBCLASS\_BOOT,

USB\_INTERFACE\_PROTOCOL\_MOUSE) },

{ } /\* Terminating entry \*/

};

MODULE\_DEVICE\_TABLE (usb, usb\_mouse\_id\_table);

**static** **struct** usb\_driver usb\_mouse\_driver = {

.name = "my\_usb\_mouse\_driver",

.probe = usb\_mouse\_probe,

.disconnect = usb\_mouse\_disconnect,

.id\_table = usb\_mouse\_id\_table,

};

module\_usb\_driver(usb\_mouse\_driver);

## 3.4 Makefile

Листинг 3.4.1 – makefile для сборки модулей

KBUILD\_EXTRA\_SYMBOLS = $(**shell** pwd)/Module.symverscd

ifneq ($(KERNELRELEASE),)

obj-m := my\_logger.o my\_usb\_mouse\_driver.o

else

CURRENT = $(**shell** uname -r)

KDIR = /lib/modules/$(CURRENT)/build

PWD = $(**shell** pwd)

default**:**

$(**MAKE**) -C $(KDIR) M=$(PWD) modules

make cleanHalf

cleanHalf**:**

rm -rf **\***.o **\*~** **\***.mod **\***.mod.c Module.**\*** **\***.order .tmp\_versions

clean**:**

make cleanHalf

rm -rf **\***.ko

endif

## 3.5 Взаимодействие с модулями

На рис. 3.5.1 продемонстрирована загрузка модуля логгера при помощи команды insmod и показано содержимое /proc/my\_logger.

На рис. 3.5.2 продемонстрирована выгрузка модуля логгера при помощи команды rmmod и показано содержимое, что /proc/my\_logger более недоступна.

На рис. 3.5.3 продемонстрирована последовательность загрузки драйвера мыши. Модуль логгера должен быть загружен раньше драйвера.

На рис 3.5.4 - 3.5.6 продемонстрировано соответствие событий пользовательского взаимодействия с мышью и соответствующих записей в журнале.

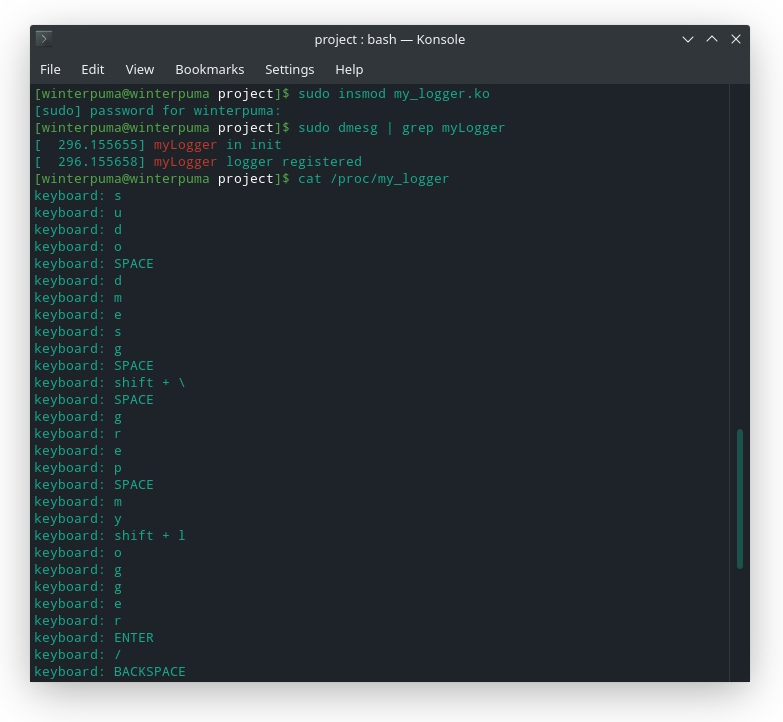


Рис. 3.5.1 – загрузка модуля логгера

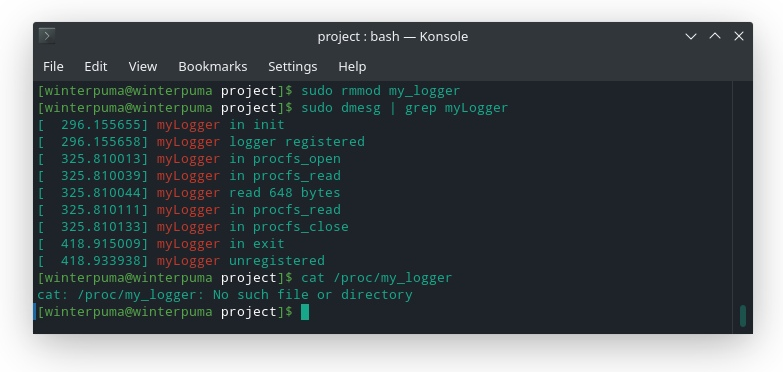


Рис 3.5.2 – выгрузка модуля логгера

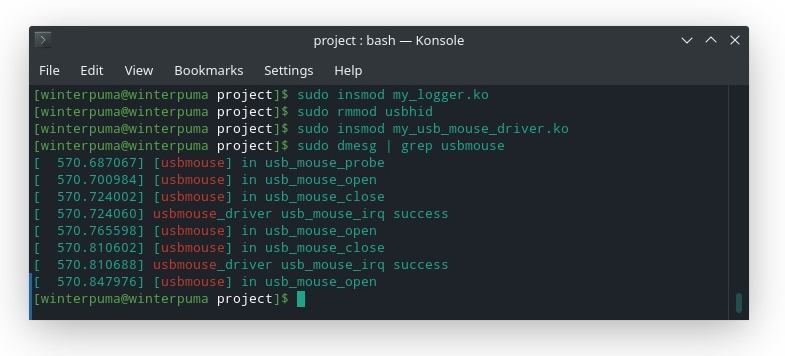


Рис 3.5.3 – загрузка модуля драйвера

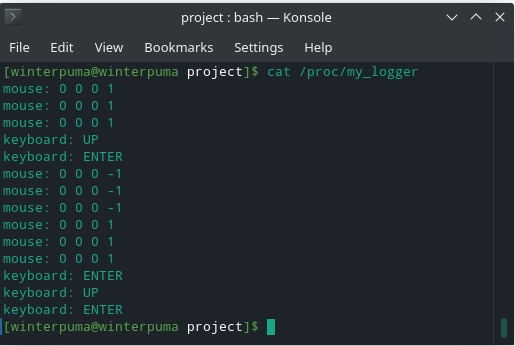


Рис 3.5.4 – движение колесиком мыши

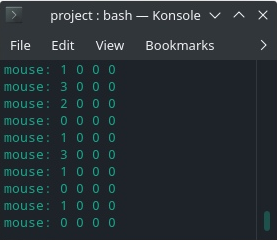


Рис. 3.5.5 – нажатие клавиш мыши

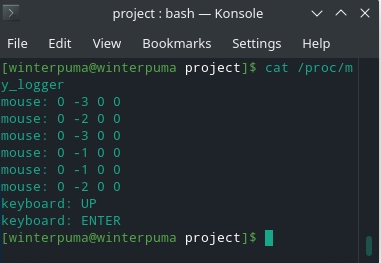


Рис. 3.5.6 – движение мыши влево

## 3.6 Вывод

Были реализованы модуль драйвера usb мыши и модуль-логгер, листинг которых был предоставлен в данном разделе. Также была проведена апробация различных сценариев использования устройств: нажатие клавиш клавиатуры и мыши, движение мыши и ее колесика.

# Заключение

Во время выполнения курсового проекта были достигнуты поставленные цель и задачи: проанализированы способы перехвата сообщений от мыши и клавиатуры; проанализирована структура драйвера мыши; проанализированы методы передачи информации из модулей ядра в пространство пользователя; спроектированы и реализованы модуль ядра и драйвер.

Были реализованы загружаемый модуль ядра, выполняющий перехват сообщений клавиатуры и мыши и драйвер USB мыши.

В ходе выполнения поставленных задач были изучены возможности языка C, получены знания в области написания загружаемых модулей ядра, драйверов.

# Список использованной литературы

1. Exploring /dev/input. Keerthi Vasan G.C, Suresh. B, 21.04.2017. The hacker Diary. [Электронный ресурс]. – Режим доступа: <https://thehackerdiary.wordpress.com/2017/04/21/exploring-devinput-1/>
2. Linux Input drivers v1.0. Vojtech Pavlik. [Электронный ресурс]. – Режим доступа: <https://www.kernel.org/doc/Documentation/input/input.txt>
3. Исходный код файла notifier.h [Электронный ресурс]. – Режим доступа: <https://github.com/torvalds/linux/blob/master/include/linux/notifier.h>
4. Исходный код файла keyboard.h [Электронный ресурс]. – Режим доступа: <https://github.com/torvalds/linux/blob/master/include/linux/keyboard.h>
5. А.Н. Васюнин, Н.Ю. Рязанова, канд. техн. наук, доц., Е.В. Тарасенко, С.В. Тарасенко. Анализ методов изменения функциональности внешних устройств в ОС Linux. В сб. «Автоматизация. Современные технологии», 2016 №10. С. 3-8.
6. Исходный код файла usb.h [Электронный ресурс]. – Режим доступа: <https://github.com/torvalds/linux/blob/master/include/linux/usb.h>
7. Corbet J., Rubini A., Kroan-Hartman G. Linux device drivers. O’Reilly Media, 2005. 567 p.
8. Peter Jay Salzman, Michael Burian, Ori Pomerantz. The Linux Kernel Module Programming Guide. 2007, ver. 2.6.4.
9. Исходный код файла fs.h [Электронный ресурс]. – Режим доступа: <https://github.com/torvalds/linux/blob/master/include/linux/fs.h>
10. Robert Love. Linux Kernel: How does copy\_to\_user work? [Электронный ресурс]. – Режим доступа: <https://www.quora.com/Linux-Kernel-How-does-copy_to_user-work>