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, искусственный интеллект и системы управления»

КАФЕДРА ИУ7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Драйвер геймпада Logitech F310 в качестве мыши и клавиатуры»

Студент группы ИУ7-75Б _____ А. К. Клименко

(Подпись, дата)

Руководитель _____ Н. Ю. Рязанова

(Подпись, дата)

Москва, 2022 г.

РЕФЕРАТ

Курсовая работа по дисциплине «Операционные системы» на тему «Драйвер геймпада Logitech F310 в качестве мыши и клавиатуры», студента Клименко А. К.

Работа изложена на 31 страницах машинописного текста. Состоит из: введения, 4 разделов, заключения и списка литературы из 11 источников.

Ключевые слова: загружаемый модуль ядра; драйвер геймпада; ОС Linux.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель данной работы – написать драйвер геймпада Logitech F310 для использования его в качестве мыши и клавиатуры.

1 Аналитический раздел

1.1 Формализация задачи

Цель данной работы – написать драйвер геймпада Logitech F310 для использования его в качестве мыши и клавиатуры.

Для достижения поставленной цели необходимо:

— рассмотреть устройство подсистемы USB,

— проанализировать протокол взаимодействия геймпада,

— разработать загружаемый модуль ядра,

— провести исследование разработанного драйвера.

1.2 Драйверы устройств в ОС Linux

Прикладные программы не могут обращаться к устройствам напрямую. Вся работа с устройствами должна происходить с использованием средств, предоставляемых операционной системой. Реализация взаимодействия операционной системы с новым внешним устройством требует написания драйвера – управляющей программы.

Драйверы устройств играют особую роль в ядре Linux. Они полностью скрывают детали того, как работают устройства, предоставляя интерфейс для взаимодействия с ними. Ролью драйвера устройства является сопоставление набора стандартизированных вызовов с операциями, специфичными для конкретного устройства.

Драйверы могут быть созданы отдельно от остальной части ядра и подключены во время выполнения, когда это необходимо.

1.3 Подсистема USB

Универсальная последовательная шина (USB) – это соединение между компьютером и рядом периферийных устройств. Первоначально протокол USB был создан для замены широкого спектра медленных шин – параллельных, последовательных и клавиатурных подключений – одним типом шины, к которому могли бы подключаться все устройства.

В ядре операционной системы Linux имеется подсистема предназначенная для работы с USB-устройствами [2]. Так как целевое устройство – геймпад – имеет USB интерфейс, в дальнейшем будет использована именно эта подсистема.

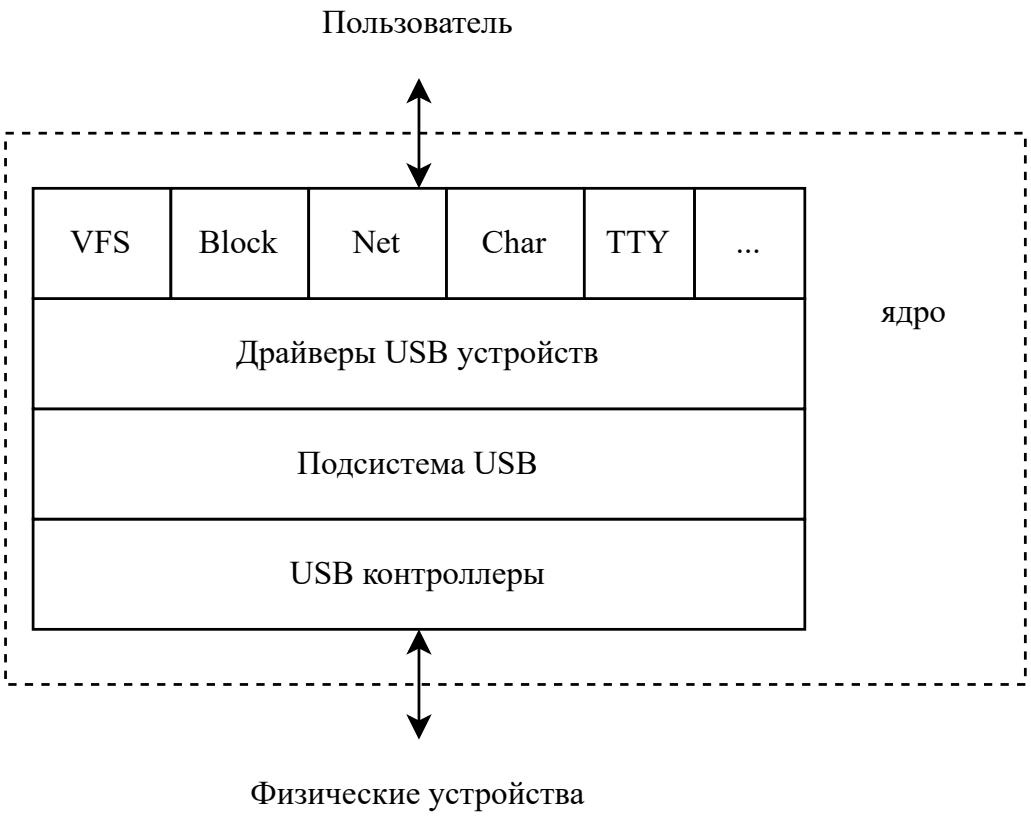

Рисунок 1 – Общее представление подсистемы USB

Для регистрации нового драйвера необходимо проинициализировать структуру `usb_driver` и вызвать функцию `usb_register`. По окончании работы с устройством, драйвер можно снять с учета посредством вызова функции `usb_deregister`.

```
struct usb_driver {
    const char *name;
    int (*probe)(struct usb_interface *intf, struct usb_device_id *id);
    void (*disconnect) (struct usb_interface *intf);
    const struct usb_device_id *id_table;
    /* ... */
};

int usb_register(struct usb_driver *driver);
void usb_deregister(struct usb_driver *driver);
```

Таблица `id_table` предоставляет подсистеме информацию о том, какие именно устройства могут управляться регистрируемым драйвером.

Функции `probe` и `disconnect` вызываются соответственно в моменты подключения и отключения устройства, соответствующего данному драйверу.

1.4 Подсистема ввода

Подсистема ввода – это уровень абстракции между устройствами ввода (клавиатура, мышь, геймпад и т. д.) и обработчиками ввода. Устройства ввода фиксируют входные данные от действий пользователя и генерируют входные события. Входные события проходят через подсистему ввода и отправляются заинтересованным обработчикам. Ядро ввода обеспечивает сопоставление ”многие ко многим” между устройствами ввода и обработчиками событий.

Список наиболее распространенных типов событий, генерируемых с использованием подсистемы ввода:

— `EV_KEY` – используется для описания изменений состояния клавиатур, кнопок или других устройств, похожих на клавиши;
— `EV_REL` – используется для описания изменения значения относительной оси, например, перемещения мыши на 5 единиц влево;
— `EV_ABS` – используется для описания изменений значений абсолютной оси, например, описания координат касания на сенсорном экране.

Для реализации управления мышью через геймпад можно использовать джойстики (один для перемещения мыши, второй для управления колесиком). Кнопки геймпада A и B можно назначить на кнопки мыши (левую и правую соответственно).

Для использования возможностей подсистемы ввода необходимо зарегистрировать драйвер устройства ввода. Создание структуры драйвера может быть выполнено вызовом одной из функции

```
struct input_dev *input_allocate_device(void);
struct input_dev *devm_input_allocate_device(struct device *dev);
```

Вторая функция использует механизм управляемых ресурсов устройств [6]. Это позволяет сократить количество ошибок, связанных с очищением памяти после использования, в связи с применением счетчика ссылок. Для описания ресурсов устройства в ядре имеется специальная структура `devres`.

```
typedef void (*dr_release_t)(struct device *dev, void *res);

struct devres_node {
    struct list_head entry;
    dr_release_t release;
    const char *name;
    size_t size;
};

struct devres {
```

```
    struct devres_node node;
    u8 __aligned(ARCH_KMALLOC_MINALIGN) data[];
};
```

При удалении структуры устройства из системы, все связанные ресурсы будут освобождены посредством вызова функций `dr_release_t`.

1.4.1 Перемещение курсора

Изменение положения курсора на экране происходит посредством генерирования события типа `EV_REL`. Согласно спецификации геймпада [1], движение джойстиков генерирует события типа `EV_ABS`. В связи с этим, реализация движения курсора будет некорректной при простой замене одного типа события на другой – фиксация джойстика в смещенном положении не будет приводить к поступлению новых URB блоков и генерированию новых событий перемещения мыши, и как следствие не будет происходить изменение положения курсора.

Для решения изложенной проблемы необходимо использовать таймер, который должен с постоянной периодичностью генерировать события при возникновении описанной ситуации. Для работы с таймером в ядре существует структура `timer_list`.

```
struct timer_list {
    struct hlist_node entry;
    unsigned long expires;
    void (*function)(struct timer_list *);
    u32 flags;
};

#define timer_setup(timer, callback, flags) /* ... */

int del_timer(struct timer_list *timer);
int mod_timer(struct timer_list *timer, unsigned long expires);
```

Запуск таймера должен происходить только тогда, когда джойстики находятся в смещенном положении. Планирование следующего выполнения осуществляется вызовом функции `mod_timer`.

```
#define TIMER_PERIOD (HZ / 100) // 10 ms
mod_timer(&timer, jiffies + TIMER_PERIOD);
```

1.5 Виртуальная клавиатура

Ввод символов является задачей не свойственной геймпаду – количество имеющихся кнопок на устройстве не позволяет установить их однозначного соответствия символам, вводимым с клавиатуры. Одним из возможных вариантов ввода символов является использование виртуальной клавиатуры и указателя, который можно перемещать по ней с помощью D-pad секции на геймпаде. Ввод символа под указателем можно осуществлять нажатием кнопки LB на геймпаде. Таким образом можно вводить любой символ, располагающийся на виртуальной клавиатуре.

Однако для удобства пользователя нужно иметь возможность отобразить на экране виртуальную клавиатуру вместе с текущей позицией указателя. Так как работа с дисплеем напрямую из ядра требует учета множества факторов, управление отображением виртуальной клавиатуры напрямую из ядра становится трудоемкой задачей. Более оптимальным вариантом является написание демона, который по запросу будет выводить на экран виртуальную клавиатуру.

1.6 Взаимодействие драйвера и демона

Одним из возможных способов передачи информации из пространства ядра в пространство пользователя является использование виртуальной файловой системы proc [7].

Для того, чтобы создать файл в файловой системе proc необходимо вызвать одну из функций

```
struct proc_dir_entry *proc_create(const char *name, umode_t mode,
    struct proc_dir_entry *parent, const struct proc_ops *proc_ops);

struct proc_dir_entry *proc_create_data(const char *name, umode_t
    mode, struct proc_dir_entry *parent, const struct proc_ops
    *ops, void *data);
```

Операции, которые могут быть осуществлены с файлом определяются структурой `proc_ops`. В случае реализации передачи событий из пространства ядра в пространство пользователя, необходимо и достаточно реализовать две операции: открытие и чтение.

При этом, если события не возникают, процесс должен быть заблокирован при попытке чтения. Необходимость блокировки процесса и ожидания появления нового события приводит к использованию очередей ожидания, которые представляются в ядре структурой `wait_queue_head`

```
struct wait_queue_entry {
    unsigned int flags;
    void *private;
    wait_queue_func_t func;
    struct list_head entry;
};

struct wait_queue_head {
    spinlock_t lock;
    struct list_head head;
};
```

Блокировка процесса и ожидание возникновения события осуществляется вызовом макроса

`wait_event_interruptible(wq_head, condition).`

Все ждущие в данной очереди процессы могут быть пробуждены вызовом макроса `wake_up_all(wq_head)`, после чего будет анализироваться выра-

жение `condition` переданное при блокировке. Если оно ложно, то процесс вновь блокируется.

Выводы

Для управления мышью и клавиатурой с использованием геймпада необходимо написать загружаемый модуль ядра с USB-драйвером, а также демона, предоставляющего сервис – отображение виртуальной клавиатуры.

Задача демона будет заключаться в отслеживании поступающих событий. По запросу он должен открывать окно с виртуальной клавиатурой и отображать перемещение виртуального указателя.

2 Конструкторский раздел

В данном разделе приведены ключевые алгоритмы использовавшиеся при написании драйвера и демона с описанием в виде схем.

2.1 Алгоритм обработки URB

На рисунке 2 приведена схема алгоритма обработки URB.

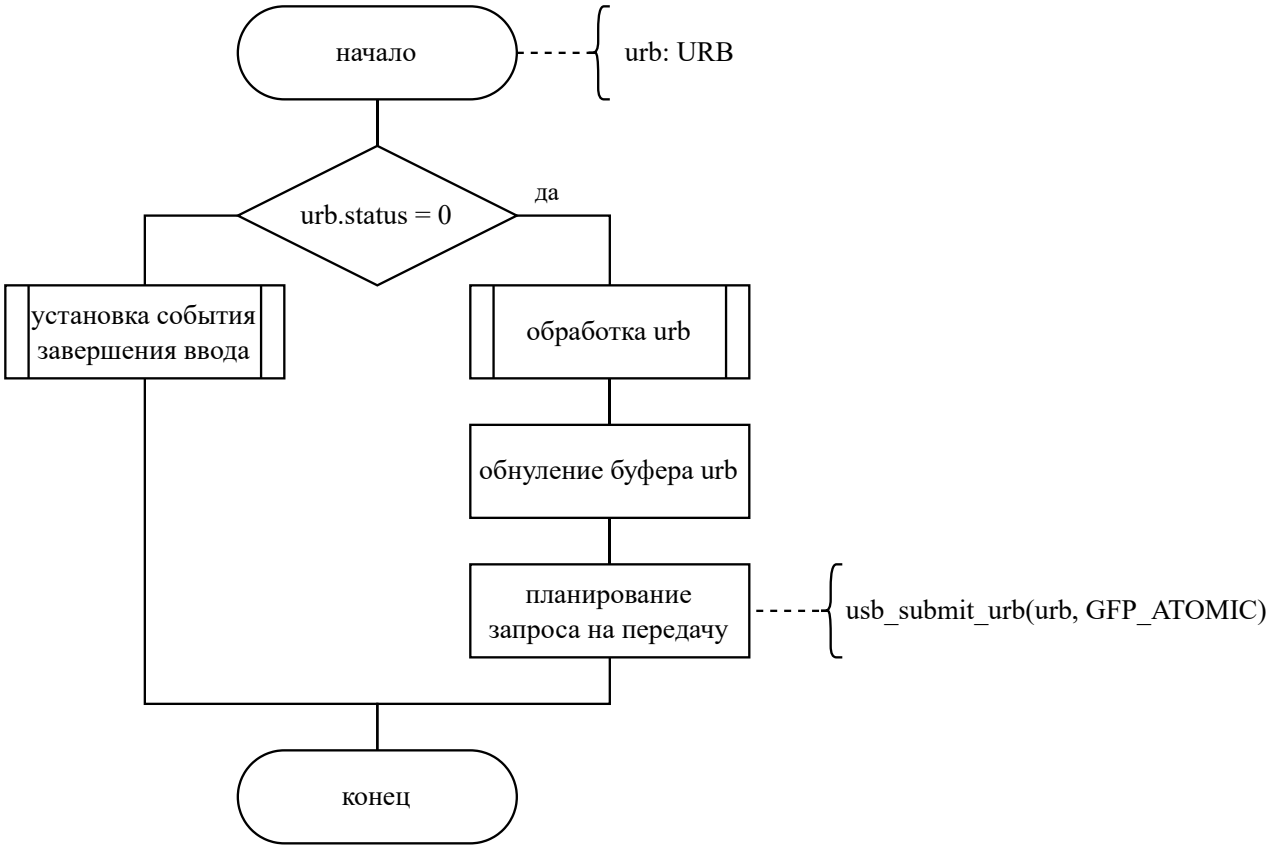

Рисунок 2 – Схема алгоритма обработки URB

На рисунке 3 приведена схема алгоритма обработки корректного URB пакета и генерации необходимых сообщений о вводе.

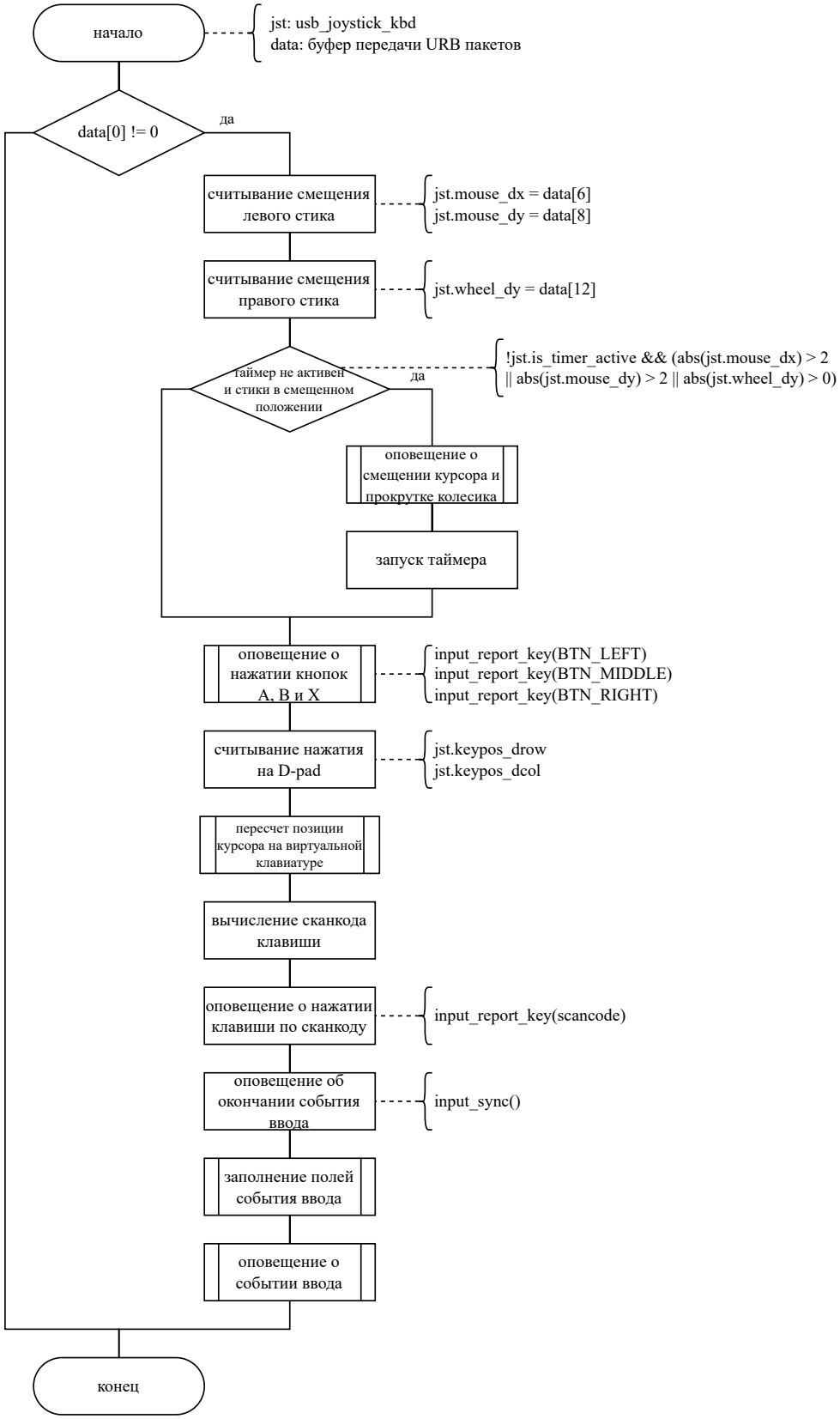

Рисунок 3 – Схема алгоритма обработки корректного URB пакета

2.2 Алгоритмы работы с событиями ввода

На рисунке 4 представлена схема алгоритма издания события с пробуждением ждущих процессов.

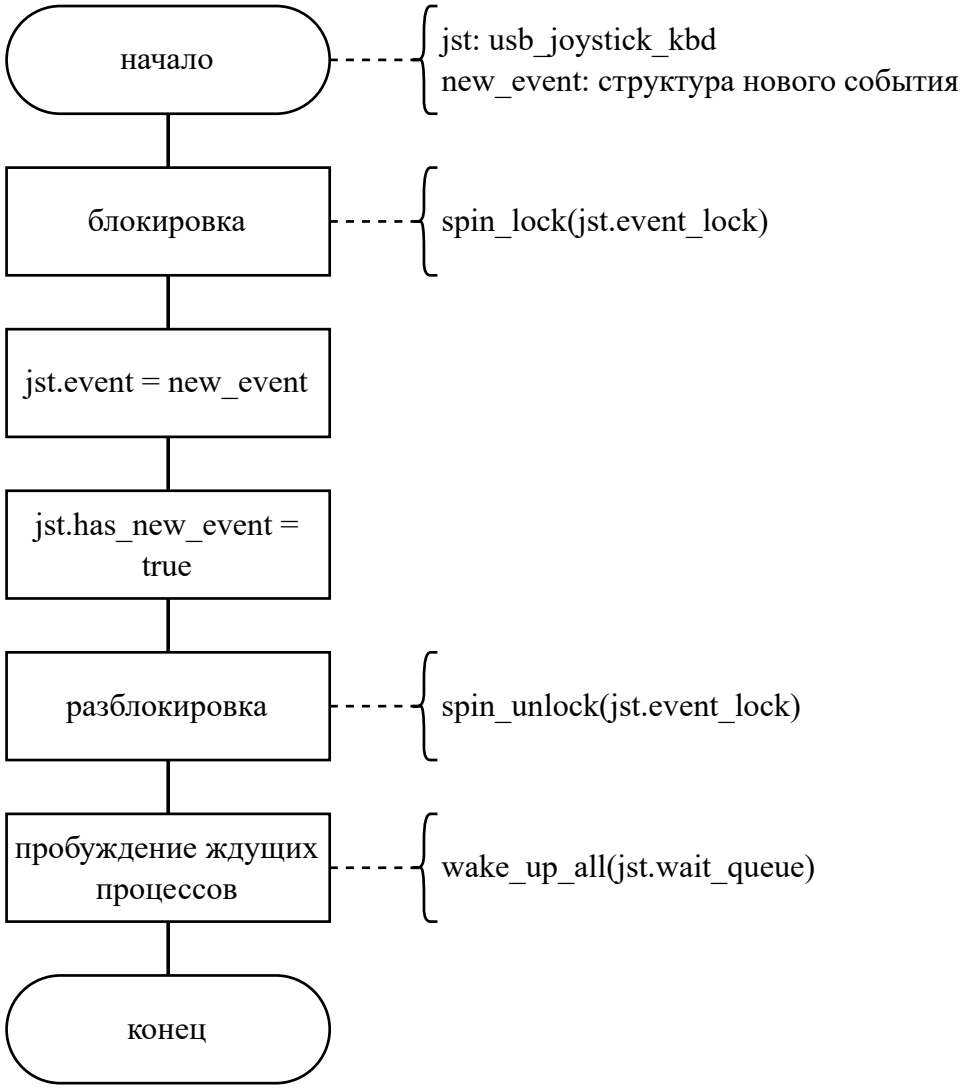

Рисунок 4 – Схема алгоритма издания события

На рисунке 5 приведена схема алгоритма чтения события с блокировкой и ожиданием.

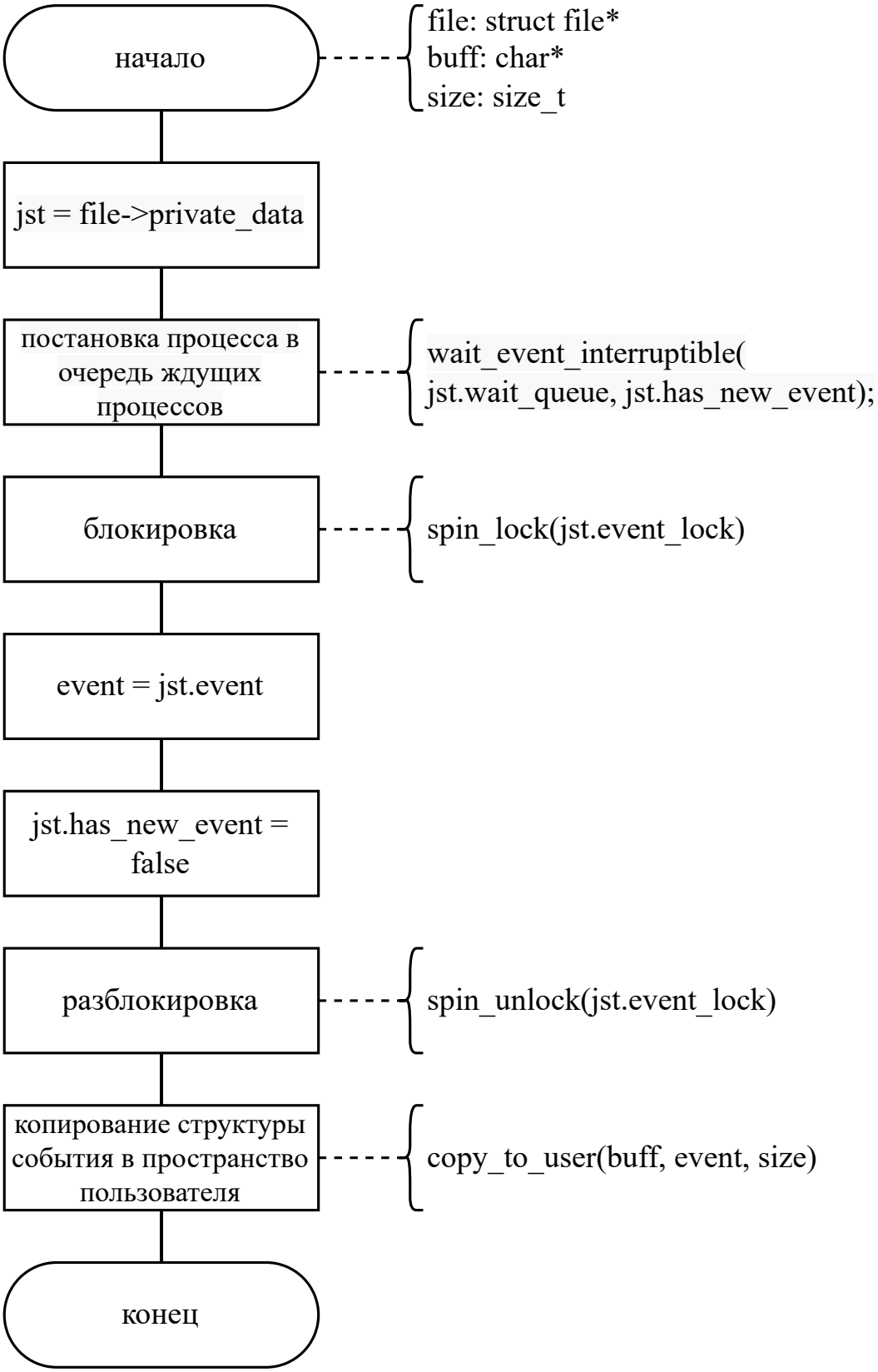

Рисунок 5 – Схема алгоритма чтения события из файла

3 Технологический раздел

На момент написания данной работы новейшей версией ярда Linux была версия 6.1. LTS версия ядра – 5.15.

Загружаемый модуль ядра будет написан для версий ядра 5.15 и 6.1. Для учета особенностей каждой из версий будет использована условная компиляция.

Демон будет написан в виде сервиса утилиты systemd [8].

3.1 Выбор средств реализации

Несмотря на то, что в ядро Linux с версии 6.1 добавлена поддержка языка программирования Rust [9], для написания драйвера был выбран язык программирования C, так как внедрение языка Rust является экспериментальной особенностью.

Для написания демона был выбран язык программирования C++. Также было решено использовать библиотеку Qt для создания и отображения виртуальной клавиатуры.

Для сборки обоих программ будет использована утилита Make.

В качестве среды разработки была выбран редактор VSCode.

3.2 Структуры драйвера

На листинге 1 приведена главная структура разрабатываемого драйвера.

Листинг 1: Структура данных драйвера

```
1  struct usb_joystick_kbd
2  {
3      /* device related section */
4      struct usb_device *usbdev;
```

```
    struct urb *urb;
    unsigned char *transfer_buffer;
    dma_addr_t dma_addr;

    /* mouse input handling stuff */
    struct input_dev *input_dev;
    int mouse_dx;
    int mouse_dy;
    int wheel_dy;
    struct timer_list timer;
    bool is_timer_active;

    /* keyboard input handling */
    unsigned char *keycodes;
    unsigned int keypos_row;
    unsigned int keypos_col;

    // proc entry for events passing between kernel and user space
    struct proc_dir_entry *proc_entry;
    struct wait_queue_head wq;
    bool has_new_event;
    struct joystick_event event;
    struct spinlock event_lock;
    char prev_data[PACKET_LEN];
};
```

На листинге 2 приведена структура события, посылаемого демону.

Листинг 2: Структура события перемещения указателя

```
struct joystick_event
{
    unsigned int keyboard_cursor_row;
    unsigned int keyboard_cursor_col;
};
```

3.3 Реализация алгоритмов

На листинге 3 приведена реализация обработки корректного URB пакета.

Листинг 3: Реализация алгоритма обработки корректного блока URB

```
void dispatch_joystick_input(struct usb_joystick_kbd *jst)
{
    struct joystick_event new_event;
    unsigned char *data;
    unsigned char *prev_data;
    unsigned char keycode;
    int keypos_d_row; // virtual keyboard cursor offset
    int keypos_d_col; //

    data = jst->transfer_buffer;
    prev_data = jst->prev_data;
    if (data[0] != 0x00)
        return;

    jst->mouse_dx = data[6]; // left stick
    jst->mouse_dy = data[8];
    jst->wheel_dy = data[12]; // right stick
    if (((abs(jst->mouse_dx) > 2) || (abs(jst->mouse_dy) > 2) ||
         (abs(jst->wheel_dy) > 0)) && !jst->is_timer_active)
    {
        input_report_rel(jst->input_dev, REL_X, jst->mouse_dx);
        input_report_rel(jst->input_dev, REL_Y, jst->mouse_dy);
        input_report_rel(jst->input_dev, REL_WHEEL, jst->wheel_dy);

        mod_timer(&jst->timer, jiffies + TIMER_PERIOD);
        jst->is_timer_active = true;
    }

    input_report_key(jst->input_dev, BTN_LEFT, data[3] & 0x10);
    input_report_key(jst->input_dev, BTN_RIGHT, data[3] & 0x20);
    input_report_key(jst->input_dev, BTN_MIDDLE, data[3] & 0x40);

    // read D-pad input for virtual keyboard cursor movement
    keypos_d_row = ((int)(data[2] >> 1) & 1) - ((int)(data[2] >> 0) & 1);
```

```
35          keypos_d_col = ((int)(data[2] >> 3) & 1) - ((int)(data[2] >> 2) & 1);
36
37          keycode = move_keyboard_cursor(&jst->keypos_row, &jst->keypos_col,0,0);
38          input_report_key(jst->input_dev, keycode, 0); // release old key
39
40          // get new key
41          keycode = move_keyboard_cursor(&jst->keypos_row, &jst->keypos_col,
42              keypos_d_row, keypos_d_col);
43
44          input_report_key(jst->input_dev, keycode, data[3] & BIT(0));
45          input_sync(jst->input_dev);
46
47          new_event.keyboard_cursor_row = jst->keypos_row;
48          new_event.keyboard_cursor_col = jst->keypos_col;
49
50          memcpy(prev_data, data, PACKET_LEN);
51
52          spin_lock(&jst->event_lock);
53          jst->event = new_event;
54          jst->has_new_event = true;
55          spin_unlock(&jst->event_lock);
56          wake_up_all(&jst->wq);
57      }
```

На листинге 4 приведена реализация функций работы с файлом в виртуальной файловой системе /proc.

Листинг 4: Реализация функций работы с файлом событий

```
static struct proc_ops proc_ops = {
    .proc_open = proc_open,
    .proc_read = proc_read,
};

struct proc_dir_entry *create_joystick_event_entry
    (struct usb_joystick_kbd *usb_joystick_kbd)
{
    return proc_create_data("joystick_kbd", S_IRUGO, NULL, &proc_ops,
        usb_joystick_kbd);
}

int proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE < KERNEL_VERSION(5, 17, 0)
    file->private_data = PDE_DATA(inode);
#else
    file->private_data = pde_data(inode);
#endif
    return 0;
}

ssize_t proc_read(struct file *file, char __user *buff, size_t size,
    loff_t *offset)
{
    struct usb_joystick_kbd *joystick_kbd = file->private_data;
    struct joystick_event event;

    wait_event_interruptible(joystick_kbd->wq,joystick_kbd->has_new_event);

    spin_lock(&joystick_kbd->event_lock);
    event = joystick_kbd->event;
    joystick_kbd->has_new_event = false;
    spin_unlock(&joystick_kbd->event_lock);

```

```
36          size = min(size, sizeof(struct joystick_event));
37          if (copy_to_user(buff, &event, size))
38          {
39              printk(KERN_ERR MOD_PREFIX "failed to copy to user in read");
40              return -1;
41          }
42
43          return size;
44      }
```

На листинге 5 представлена реализация функций для работы с устройством ввода подсистемы `input`.

Листинг 5: Реализация функций для работы с устройством ввода подсистемы `input`

```
1   struct input_dev *allocate_joystick_input_dev(struct usb_device *usb_dev)
2   {
3       struct input_dev *input_dev = devm_input_allocate_device(&usb_dev->dev);
4       if (input_dev != NULL)
5       {
6           usb_to_input_id(usb_dev, &input_dev->id);
7           input_dev->name = INPUT_DEV_NAME;
8           input_dev->open = input_open;
9           input_dev->close = input_close;
10      }
11      return input_dev;
12  }
13
14  int input_open(struct input_dev *input_dev)
15  {
16      struct usb_joystick_kbd *jst = input_get_drvdata(input_dev);
17      timer_setup(&jst->timer, input_timer_callback, 0);
18      mod_timer(&jst->timer, jiffies + TIMER_PERIOD);
19      if (usb_submit_urb(jst->urb, GFP_KERNEL) != 0)
20          return -EIO;
21      return 0;
```

```
22  }
23
24  void input_close(struct input_dev *input_dev)
25  {
26      struct usb_joystick_kbd *jst = input_get_drvdata(input_dev);
27      del_timer_sync(&jst->timer);
28      usb_kill_urb(jst->urb);
29  }
30
31  void input_timer_callback(struct timer_list *timer)
32  {
33      bool was_update = false;
34      struct usb_joystick_kbd *jst = from_timer(usb_joystick_kbd,
35          timer, timer);
36
37      if (abs(jst->mouse_dx) > 2)
38      {
39          input_report_rel(jst->input_dev, REL_X, jst->mouse_dx);
40          was_update = true;
41      }
42      if (abs(jst->mouse_dy) > 2)
43      {
44          input_report_rel(jst->input_dev, REL_Y, jst->mouse_dy);
45          was_update = true;
46      }
47      if (abs(jst->wheel_dy) > 0)
48      {
49          input_report_rel(jst->input_dev, REL_WHEEL, jst->wheel_dy);
50          was_update = true;
51      }
52
53      if (was_update)
54      {
55          input_sync(jst->input_dev);
56          mod_timer(timer, jiffies + TIMER_PERIOD);
57      }
58      else
59          jst->is_timer_active = false;
60  }
```

3.4 Файлы конфигурации сервисов

Для запуска программы в виде демона с использованием утилиты systemd необходимо создать юнит-файл с конфигурацией сервиса. На листинге 6 приведено содержимое файла `joystick_virt_kbd.service`.

Листинг 6: Конфигурационный файл `joystick_virt_kbd.service`

```
[Unit]
Description=Joystick virtual keyboard service

[Service]
Restart=always
RestartSec=1
EnvironmentFile=/var/lib/joystick_virt_kbd/.env
ExecStart=/usr/bin/env joystick_virt_kbd

[Install]
WantedBy=multi-user.target
```

На листинге 7 представлена часть файла сборки, отвечающая за установку и удаление сервиса из подсистемы `systemd`.

Листинг 7: makefile для сборки демона

```
EXECUTABLE := daemon

# install paths
APP_INSTALL_PATH := /usr/local/bin/joystick_virt_kbd
SERVICE_INSTALL_PATH := /etc/systemd/system
PRIVATE_DATA_PATH := /var/lib/joystick_virt_kbd

install: $(EXECUTABLE)
```

```
9       sudo cp $(EXECUTABLE) $(APP_INSTALL_PATH)
10      sudo cp joystick_virt_kbd.service $(SERVICE_INSTALL_PATH)
11      sudo mkdir -p $(PRIVATE_DATA_PATH)
12      sudo -E sh -c 'env > $(PRIVATE_DATA_PATH)/.env'
13      sudo systemctl daemon-reload
14      sudo systemctl start joystick_virt_kbd
15
16  uninstall:
17      sudo systemctl stop joystick_virt_kbd
18      sudo rm -rf $(PRIVATE_DATA_PATH) $(APP_INSTALL_PATH) \
19          $(SERVICE_INSTALL_PATH)/joystick_virt_kbd.service
20      sudo systemctl daemon-reload
21
22  # остальные правила для сборки программы ...
```

3.5 Сборка и запуск

Для сборки загружаемого модуля ядра достаточно выполнить команду `make` в папке с кодом модуля, после чего загрузка модуля в ядро осуществляется командой `insmod joystick_kbd.ko`.

Сборка и запуск демона выполняется аналогичным образом. Выполнение команды `make` без параметров приведет к сборке исполняемого файла. Чтобы установить его в систему, необходимо выполнить команду `make install`. При этом сервис будет сразу же запущен. Для остановки и удаления программы из системы необходимо использовать команду `make uninstall`.

4 Исследовательский раздел

В данном разделе будет проведено исследование зависимости времени обработки URB от работы демона.

4.1 Описание исследования

Замер времени будет происходить для функции драйвера `jskbd_complete` с использованием функции ядра `ktime_get` [10], для двух сценариев:

- при запущенном демоне;
- при остановленном демоне.

При сравнении будут учитываться только первые 1000 измерений с момента запуска драйвера и подключения геймпада. В результате анализа полученных данных будут рассчитаны и сопоставлены средние значения измерений.

4.2 Технические характеристики

Исследование будет проводиться на ноутбуке Dell Vostro 14 5410. Характеристики системы приведены ниже.

- Процессор: 11th Gen Intel i7-11370H (8) @ 4.800GHz.
- Оперативная память: 8 ГБ.
- Операционная система: Arch Linux x86_64.
- Графическая оболочка: KDE Plasma 5.26.4.

4.3 Результаты исследования

На рисунке 6 приведены результаты в виде гистограммы для всех полученных измерений.

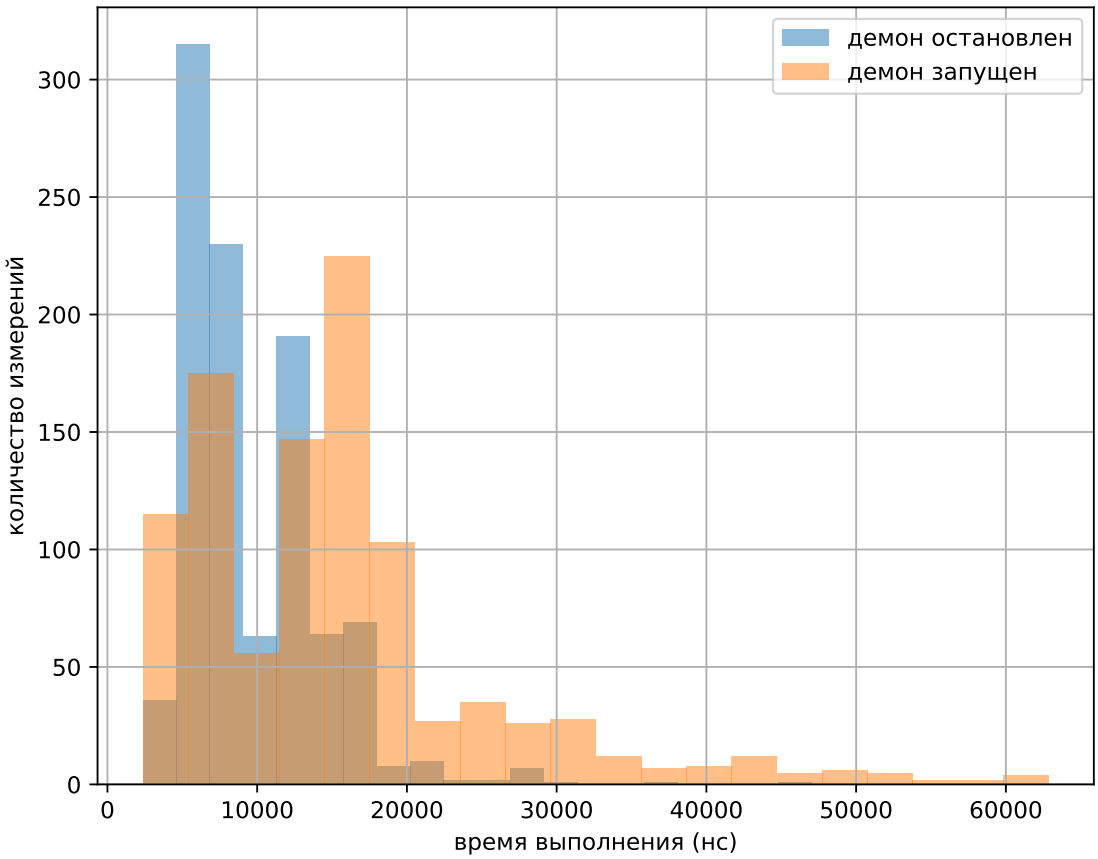

Рисунок 6 – Результаты замеров времени для двух сценариев

По результатам исследования установлены следующие средние значения обработки URB пакетов:

- при запущенном демоне: 15649 нс;
- при остановленном демоне: 9679 нс.

Данный результат можно объяснить тем, что запущенный демон требует больше времени на поддержание очереди ждущих процессов. Данное время в среднем можно считать равным 5970 нс.

ЗАКЛЮЧЕНИЕ

В ходе работы был произведен анализ подсистемы USB и подсистемы ввода, рассмотрены основные моменты использования геймпада в качестве мыши и клавиатуры, решены ключевые проблемы с использованием функциональности, предоставляемой ядром операционной системы Linux.

Был разработан драйвер для геймпада, а также написана программа, запускаемая в режиме демона с использованием утилиты `systemd`.

Работоспособность и корректность выполнения были протестированы на реальном устройстве Logitech F310 для двух версий ядра 5.15 и 6.1.

Также было проведено исследование зависимости времени обработки URB от работы демона.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Linux Gamepad Specification. [Электронный ресурс]. URL: https://docs.kernel.org/input/gamepad.html (Дата обращения: 16.11.2022)

2. Linux USB Basics. [Электронный ресурс]. URL: https://www.kernel.org/doc/htmldocs/writing_usb_driver/basics.html (Дата обращения: 21.11.2022)

3. Human Interface Devices (HID) Specifications and Tools. [Электронный ресурс]. URL: https://www.usb.org/hid (Дата обращения: 16.11.2022)

4. Jonathan Corbet. Linux Device Drivers, 3rd Edition. [Электронный ресурс]. O'REILLY. URL: https://www.oreilly.com/library/view/linux-device-drivers/0596005903 (Дата обращения: 16.11.2022)

5. Linux source code (v6.1) – Bootlin. [Электронный ресурс]. URL: https://elixir.bootlin.com/linux/v6.1/source (Дата обращения: 20.11.2022)

6. Devres – Managed Device Resource – The Linux Kernel documentation. [Электронный ресурс]. URL: https://docs.kernel.org/driver-api/driver-model/devres.html (Дата обращения: 19.11.2022)

7. The /proc Filesystem – The Linux Kernel documentation. [Электронный ресурс]. URL: https://docs.kernel.org/filesystems/proc.html (Дата обращения: 21.11.2022)

8. System and Service Manager. [Электронный ресурс]. URL: https://systemd.io (Дата обращения: 26.11.2022)

9. Rust – The Linux Kernel documentation. [Электронный ресурс]. URL: https://www.kernel.org/doc/html/next/rust/index.html (Дата обращения: 27.11.2022)

10. ktime accessors – The Linux Kernel documentation. [Электронный ресурс]. URL: https://docs.kernel.org/core-api/timekeeping.html (Дата обращения: 29.11.2022)