



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 КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Загружаемый модуль ядра, реализующий функционал мыши
при помощи геймпада»

Студент группы ИУ7-75Б

(Подпись, дата)

Сусликов Д.В.

(И.О. Фамилия)

Руководитель курсового проекта

(Подпись, дата)

Рязанова Н.Ю.

(И.О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	3
1 Аналитический раздел	4
1.1 Постановка задачи	4
1.2 Загружаемый модуль ядра	4
1.3 USB шина	5
1.4 USB драйвер	5
1.5 Оконечная точка	6
1.6 Транзакции и пакеты	8
1.7 Блоки запроса USB	9
1.8 Выводы	10
2 Конструкторский раздел	11
2.1 Требования к программному обеспечению	11
2.2 IDEF0	11
2.3 Точки входа драйвера	12
2.4 Алгоритмы эмуляции событий мыши	14
2.5 Обработка сообщений от устройства	15
2.6 Схемы алгоритмов работы драйвера	15
3 Технологический раздел	18
3.1 Выбор языка и среды программирования	18
3.2 Реализация структур и алгоритмов	18
3.3 Makefile	29
4 Исследовательский раздел	31
4.1 Загрузка драйвера	31
4.2 Результат выполнения	31
ЗАКЛЮЧЕНИЕ	33
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	34
ПРИЛОЖЕНИЕ А	35

ВВЕДЕНИЕ

Одной из часто встречающихся задач является изменение функциональности внешнего устройства. Не всегда удобно пользоваться устройствами только в качестве их классического предназначения.

Например, хотелось бы иметь возможность устройство наподобие геймпада использовать в качестве компьютерной мыши, для этого нужно разработать загружаемый модуль ядра, а именно: USB-драйвер, который способен решить поставленную задачу.

Таким образом, данная работа посвящена реализации задачи изменения функциональности геймпада с целью использования его, как компьютерной мыши.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с заданием на курсовую работу по курсу «Операционные системы» необходимо разработать программное обеспечение, позволяющее использовать геймпада в качестве компьютерной мыши.

Для реализации поставленной задачи требуется:

- проанализировать существующие драйвера геймпадов;
- разработать загружаемый модуль, позволяющий использовать функционал компьютерной мыши при помощи геймпада;
- реализовать программное обеспечение;
- выполнить тестирование разработанного модуля.

Программное обеспечение должно позволять использовать клавиши геймпада, как аналоги левой и правой клавиш мыши, перемещать курсор при помощи левого стика, пролистывать страницы при помощи правого.

1.2 Загружаемый модуль ядра

Несмотря на то, что ядро Linux является монолитным, оно позволяет выполнять вставку и удаление кода ядра в процессе работы. Для этого нужны загружаемые модули ядра.

Модуль по своей сути примерно то же, что и обычная программа. Модуль так же имеет точку входа и выхода. Но модули имеют непосредственный доступ к структурам и функциям ядра. Для программ в пространстве пользователя этот доступ ограничен[1].

Использование загружаемых модулей значительно упрощает изменение функциональности ядра и не требует ни полной перекомпиляции, ни перезагрузок. Также преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули.

Загрузка модуля осуществляется с помощью команд *insmod* или *modprobe*. Отличие заключается в том, что *modprobe* разрешает зависимости модулей. При

загрузке вызывается функция входа, определенная в модуле.

Для указания ее назначения используется макрос *module_init*. При корректной загрузке функция входа возвращает 0. В ином случае, модуль не будет загружен.

Для просмотра списка загруженных модулей и информации об отдельном модуле используются команды *lsmod* и *modinfo* соответственно. Команда *rmmod* осуществляет выгрузку модуля, при которой вызывается функция выхода. Для нее используется макрос *module_exit*.

1.3 USB шина

Universal Serial Bus (USB, Универсальная Последовательная Шина) является соединением между компьютером и несколькими периферийными устройствами. Первоначально она была создана для замены широкого круга медленных и различных шин, параллельной, последовательной и клавиатурного соединений, на один тип шины, чтобы к ней могли подключаться все устройства[2].

USB Core — это подсистема ядра Linux, созданная для поддержки USB-устройств и контроллеров шины USB. Ядро USB предоставляет интерфейс для драйверов USB, используемый для доступа и управления USB оборудованием, без необходимости беспокоиться о различных типах аппаратных контроллеров USB, которые присутствуют в системе.

Шина USB обеспечивает обмен данными между хост-компьютером и множеством периферийных устройств. USB является единой централизованной аппаратно-программной системой массового обслуживания устройств и прикладных программных процессов. Связь этих процессов со всеми устройствами обеспечивает хост-контроллер с многоуровневой программной поддержкой.

1.4 USB драйвер

Ядро Linux поддерживает два основных типа драйверов USB: драйверы на хост-системе и драйверы на устройстве. USB драйверы для хост-системы управляют USB-устройствами, которые к ней подключены, с точки зрения хоста (обычно хостом USB является персональный компьютер.) USB-драйверы в

устройстве контролируют, как одно устройство видит хост-компьютер в качестве устройства USB.

Драйверы основного ядра обращаются к прикладным интерфейсам USB ядра. В тоже время принято выделять два основных публичных прикладных интерфейса: один — реализует взаимодействие с драйверами общего назначения (символьное устройство), другой — взаимодействие с драйверами, являющимися частью ядра (драйвер хаба). Второй тип драйверов участвует в управлении USB шиной.

На Рисунке 1 представлена, как USB-устройства состоят из конфигураций, интерфейсов и конечных точек и как USB драйверы связаны с интерфейсами USB, а не всего устройства USB.

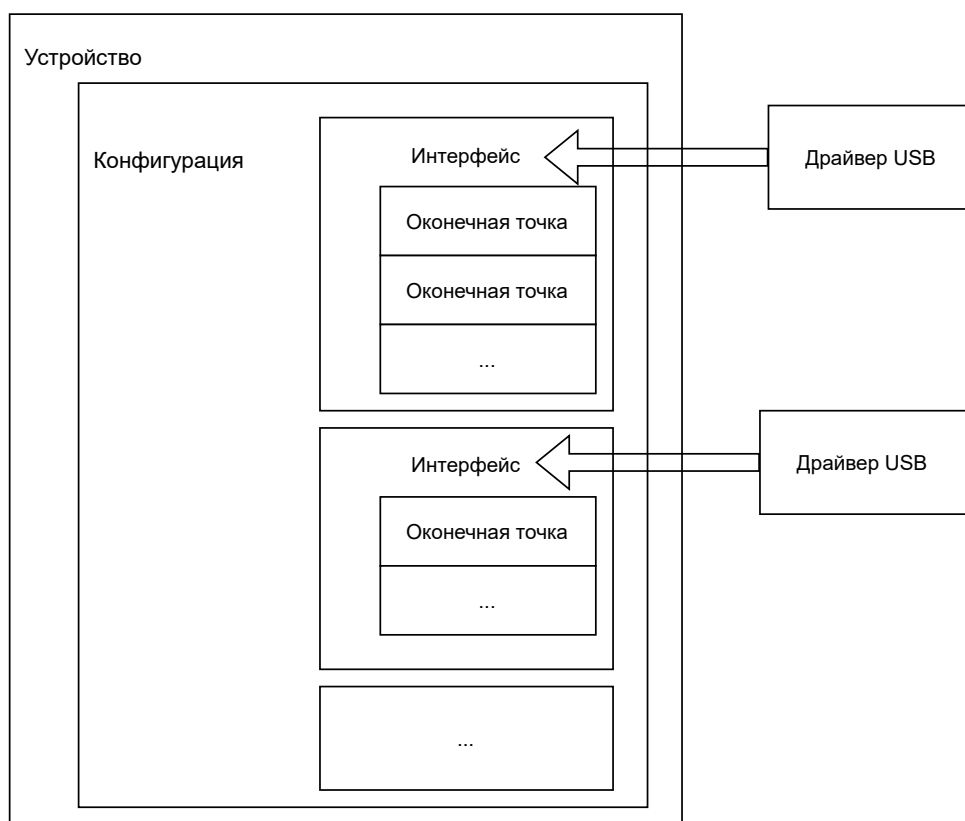


Рисунок 1 – Схема взаимодействия устройства и драйвера

1.5 Оконечная точка

Самый основной формой USB взаимодействия является то, что называется endpoint (оконечная точка). Оконечная точка USB может переносить данные

только в одном направлении, либо со стороны хост-компьютера на устройство (называемая оконечной точкой OUT) или от устройства на хост-компьютер (называемая оконечной точкой IN). Оконечные точки можно рассматривать как односторонне направленные трубы.

Драйвер геймпада имеет только 1 конечную точку типа INTERRUPT (прерывание). Для конечных точек данного типа характерна передача небольшого объема данных с фиксированной частотой.

Этот тип оконечных точек является основным транспортным методом не только для геймпадов, но и для USB клавиатур и мышей. Передачи данного типа имеют зарезервированную пропускную способность.

Помимо типа INTERRUPT есть ещё 3: CONTROL (Управление), BULK (Поток), ISOCHRONOUS (Изохронная).

Управляющие оконечные точки используются для обеспечения доступа к различным частям устройства USB. Они широко используются для настройки устройства, получения информации об устройстве, послыке команд в устройство, или получения статусных сообщений устройства. Эти оконечные точки, как правило, малы по размеру.

Поточные оконечные точки передают большие объёмы данных. Они являются обычными для устройств, которые должны передавать любые данные, которые должны пройти через шину, без потери данных. Эти оконечные точки общеприняты на принтерах, устройствах хранения и сетевых устройствах.

Изохронные оконечные точки также передают большие объёмы данных, но этим данным не всегда гарантирована доставка. Эти оконечные точки используются в устройствах, которые могут обрабатывать потери данных, и больше полагаются на сохранение постоянного потока поступающих данных. При сборе данных в реальном времени, таком, как аудио- и видео-устройства, почти всегда используются такие оконечные точки.

Управляющие и поточные оконечные точки используются для асинхронной передачи данных, когда драйвер решает их использовать. Оконечные точки

прерывания и изохронные точки являются периодическими. Это означает, что эти оконечные точки созданы для передачи данных непрерывно за фиксированное время, что приводит к тому, что их пропускная способность защищена ядром USB

1.6 Транзакции и пакеты

Протокол шины USB обеспечивает обмен данными между хостом и устройством. На протокольном уровне решаются такие задачи, как обеспечение достоверности и надежности передачи, управление потоком. Весь трафик на шине USB передается посредством транзакций, в каждой транзакции возможен обмен только между хостом и адресуемым устройством (его конечной точкой). Все транзакции (обмены) с устройствами USB состоят из двух-трех пакетов, типовые последовательности пакетов в транзакциях приведены на Рисунке 2.

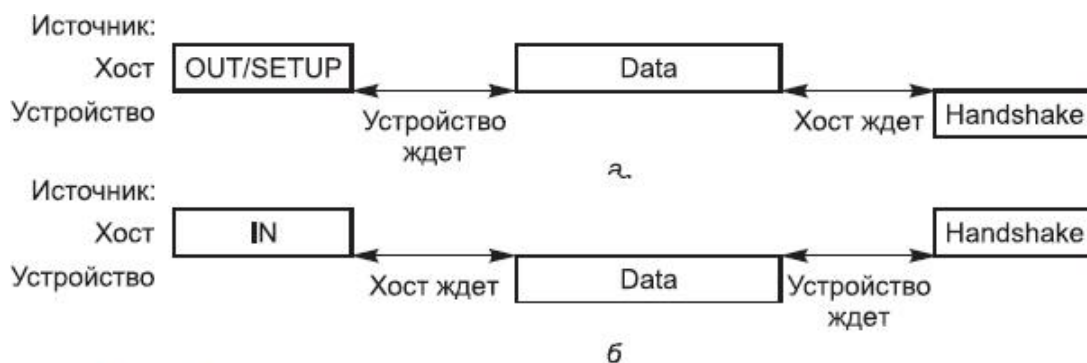


Рисунок 2 – Последовательность пакетов в транзакциях на шине USB:

а - вывод; б - ввод данных

Каждая транзакция планируется и начинается по инициативе хост-контроллера, который посылает пакет-маркер транзакции (token packet). Маркер транзакции описывает тип и направление передачи, адрес выбранного устройства USB и номер конечной точки. Адресуемое маркером устройство распознает свой адрес и готовится к обмену. Источник данных, определенный маркером, передает пакет данных. На этом этапе транзакции, относящиеся к изохронным передачам, завершаются — здесь нет подтверждения приема пакетов. Для остальных типов передач работает механизм подтверждения, обеспечивающий гарантиро-

ванную доставку данных[6]. Типы пакетов приведены ниже на Рисунке 3.

Имя	Код PID	Содержимое и назначение
Пакеты-маркеры (Token)		
OUT	0001	Маркер транзакции вывода, несет идентификатор конечной точки (адрес устройства и номер точки; направление точки определяется кодом PID)
IN	1001	Маркер транзакции ввода, несет идентификатор конечной точки (адрес устройства и номер точки; направление точки определяется кодом PID)
SETUP	1101	Маркер транзакции управления, несет идентификатор конечной точки (адрес устройства и номер точки)
SOF	0101	Маркер начала микрокадра, несет 11-битный номер кадра (вместо полей Addr и EndP)
PING	0100	Пробный маркер управления потоком (в USB 2.0)
Пакеты данных		
DATA0	0011	Пакеты данных; чередование PID позволяет различать четные и нечетные пакеты для контроля правильности подтверждения
DATA1	1011	
DATA2	0111	Дополнительные типы пакетов данных, используемые в транзакциях с широкополосными изохронными точками (в USB 2.0 для HS)
MDATA	1111	
Пакеты квитирования (Handshake)		
ACK	0010	Подтверждение безошибочного приема пакета
NAK	1010	Индикация занятости (неготовности конечной точки к обмену данными, незавершенности обработки транзакции управления)
STALL	1110	Конечная точка требует вмешательства хоста
NYET	0110	Подтверждение безошибочного приема, но указание на отсутствие места для приема следующего пакета максимального размера (в USB 2.0)
Специальные пакеты (Special)		
PRE	1100	Преамбула (маркер) передачи на низкой скорости (разрешает трансляцию данных на низкоскоростной порт хаба)
ERR	1100	Сигнализация ошибки в расщепленной транзакции (в USB 2.0)
SPLIT (SS и CS)	1000	Маркер расщепленной транзакции (в USB 2.0). В зависимости от назначения обозначается как SS (маркер запуска) и CS (маркер завершения), назначение определяется битом SC в теле маркера

Рисунок 3 – Типы пакетов и их идентификаторы PID

1.7 Блоки запроса USB

urb используется для передачи или приёма данных в или из заданной оконечной точки USB на заданное USB устройство в асинхронном режиме. Каждая оконечная точка в устройстве может обрабатывать очередь urb-ов, так что перед тем, как очередь опустеет, к одной оконечной точке может быть отправлено множество urb-ов. Типичный жизненный цикл urb выглядит следующим образом:

- создание драйвером USB;
- назначение в определённую оконечную точку заданного USB устройства;
- передача драйвером USB устройства в USB ядро;
- передача USB ядром в заданный драйвер контроллера USB узла для указанного устройства;
- обработка драйвером контроллера USB узла, который выполняет передачу по USB в устройство;

- после завершения работы с urb драйвер контроллера USB узла уведомляет драйвер USB устройства.

1.8 Выводы

В результате проведенного анализа были рассмотрены принцип работы и способы реализации загружаемого модуля ядра, позволяющего использовать геймпад как компьютерную мышь. В качестве реализации был выбран USB-драйвер геймпада.

2 Конструкторский раздел

2.1 Требования к программному обеспечению

Программное обеспечение состоит из драйвера, реализованного в виде загружаемого модуля ядра, который посредством считывания и обрабатывания информации клавиш и стиков геймпада реализует функционал обычной компьютерной мыши.

2.2 IDEF0

Ниже на Рисунках 4 и 5 представлена IDEF0 диаграмма.

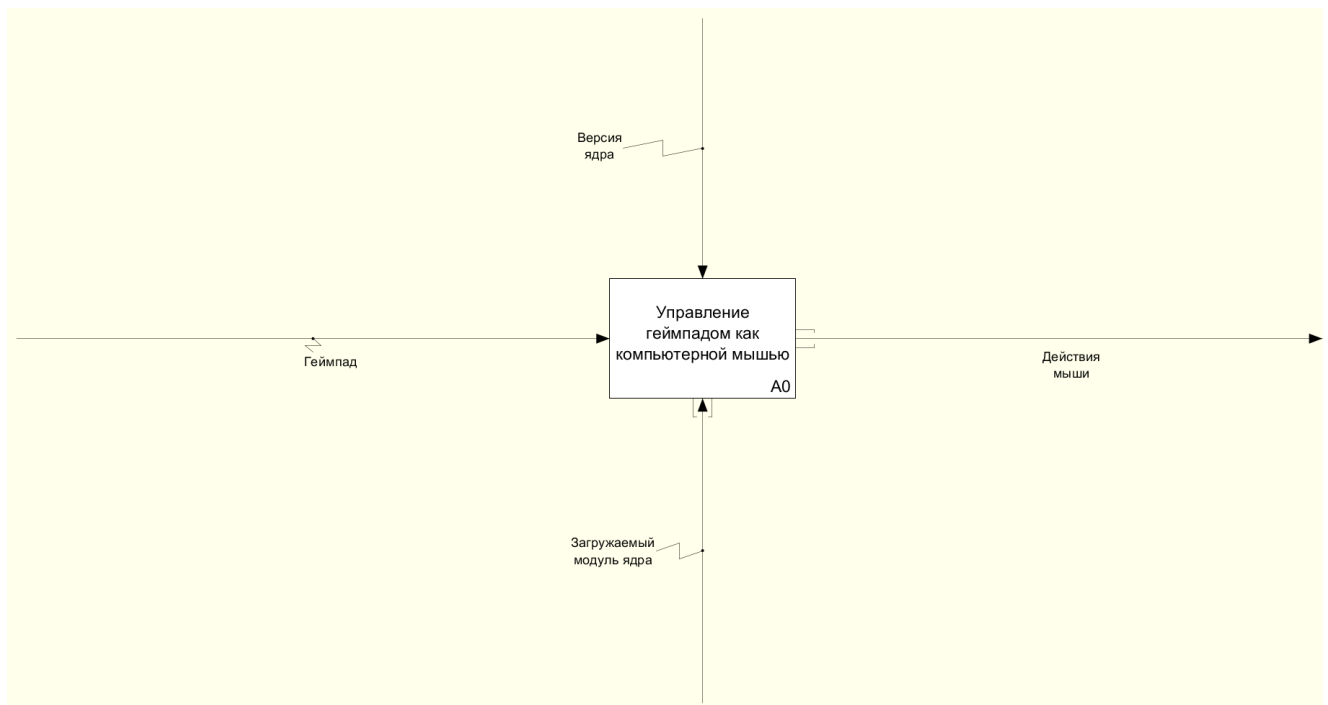


Рисунок 4 – IDEF0 уровня 0

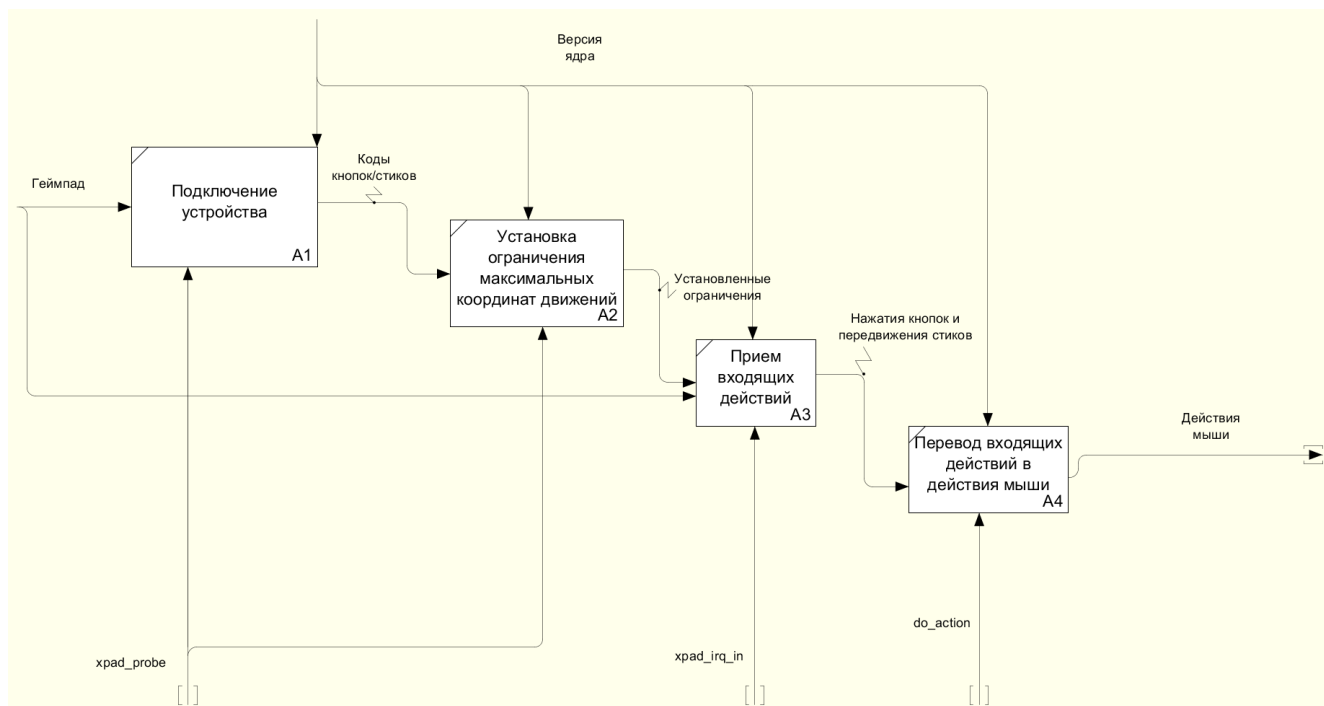


Рисунок 5 – IDEF0 уровня 1

2.3 Точки входа драйвера

Для регистрации драйвера в системе имеется структура *usb_driver*.

struct usb_driver определена в */include/linux/usb.h*. Данная структура представлена ниже на Рисунке 6.

```

struct usb_driver {
    const char *name;

    int (*probe) (struct usb_interface *intf,
                  const struct usb_device_id *id);

    void (*disconnect) (struct usb_interface *intf);

    int (*unlocked_ioctl) (struct usb_interface *intf, unsigned int code,
                           void *buf);

    int (*suspend) (struct usb_interface *intf, pm_message_t message);
    int (*resume) (struct usb_interface *intf);
    int (*reset_resume)(struct usb_interface *intf);

    int (*pre_reset)(struct usb_interface *intf);
    int (*post_reset)(struct usb_interface *intf);

    const struct usb_device_id *id_table;
    const struct attribute_group **dev_groups;

    struct usb_dynids dynids;
    struct usbdrv_wrap drvwrap;
    unsigned int no_dynamic_id:1;
    unsigned int supports_autosuspend:1;
    unsigned int disable_hub_initiated_lpm:1;
    unsigned int soft_unbind:1;
};

```

Рисунок 6 – Структура `usb_driver`

Для регистрации USB драйвера мыши следует рассматривать следующие основные поля структуры:

- *name* — имя драйвера. Оно должно быть уникальным и обычно совпадает с именем модуля;
- *probe* — указатель на функцию, которую подсистема USB ядра вызывает при подключении устройства;
- *disconnect* - указатель на функцию, которую подсистема USB ядра вызывает при отключении устройства. Внутри указанной функции выполняется освобождение памяти и отмена регистрации устройства;
- *id_table* — указатель на структуру *usb_device_id*, описывающую таблицу идентификаторов USB драйверов, необходимую для быстрого подключения устройств. В случае ее отсутствия, функция *probe* не сможет быть вызвана.

Ниже на Рисунке 7 представлен пример *id_table* — *xpad_table*.

```
static struct usb_device_id xpad_table[] = {
    { USB_INTERFACE_INFO('X', 'B', 0) }, /* X-Box USB-IF not approved class */
    XPAD_XBOX360_VENDOR(0x046d), /* Logitech X-Box 360 style controllers */
    { }
};

MODULE_DEVICE_TABLE(usb, xpad_table);
```

Рисунок 7 – Пример `id_table`

После инициализации структуры `usb_device_id` выполняется вызов макроса `MODULE_DEVICE_TABLE`. При компиляции процесс извлекает информацию из всех драйверов и инициализирует таблицу устройств. При подключении устройства, ядро обращается к таблице, где выполняется поиск записи, соответствующей идентификатору устройства. В случае нахождения такой записи, выполняется инициализация и загрузка модуля.

2.4 Алгоритмы эмуляции событий мыши

Чтобы пометить устройство, как способное генерировать определенные события, нужно применить функцию `input_set_capability`. Данная функция определена в `/include/linux/usb.h`. Ниже на Рисунке 8 представлен её вид.

```
void input_set_capability(struct input_dev *dev, unsigned int type, unsigned int code);
```

Рисунок 8 – Сигнатура функции `input_set_capability`

Поля функции:

- *dev* — устройство, способное передавать или принимать событие;
- *type* — тип события (`EV_KEY`, `EV_REL`, `EV_ABS`, etc...);
- *code* — код события (в нашем случае это код нажимаемых клавиш и перемещаемых стиков геймпада).

Используемые в данной работе типы событий:

- `EV_KEY` используется для описания изменений состояния клавиатур, кнопок или других устройств, похожих на клавиши. Этим событием описываются все нажатия кнопок геймпада;
- `EV_REL` используется для описания изменений относительных значений

оси, например, перемещения мыши. Этим событием описываются движения стика, при помощи которого курсор мыши должен изменять своё положение.

2.5 Обработка сообщений от устройства

Внутри драйвера реализована функция *xpad_irq_in*, позволяющая обрабатывать сообщения, отправленные устройством.

```
static void xpad_irq_in(struct urb *urb)
```

Внутри данной функции происходит обработка нажатых на геймпаде клавиш, а также движений стиков. Для того, чтобы сообщить о новом событии нажатия кнопки, изменения положения используется функция *do_action*. Ниже на Рисунке 9 приведён её вид.

```
void do_action(struct input_dev *dev, unsigned int type, unsigned int code, int value)
```

Рисунок 9 – Функция *do_action*

Рассмотрим её поля поподробнее:

- *dev* — устройство, сгенерировавшее событие;
- *type* — тип события;
- *code* — код события;
- *value* — значение события.

2.6 Схемы алгоритмов работы драйвера

Ниже на Рисунках 10 и 11 представлена схема алгоритма работы драйвера геймпада.

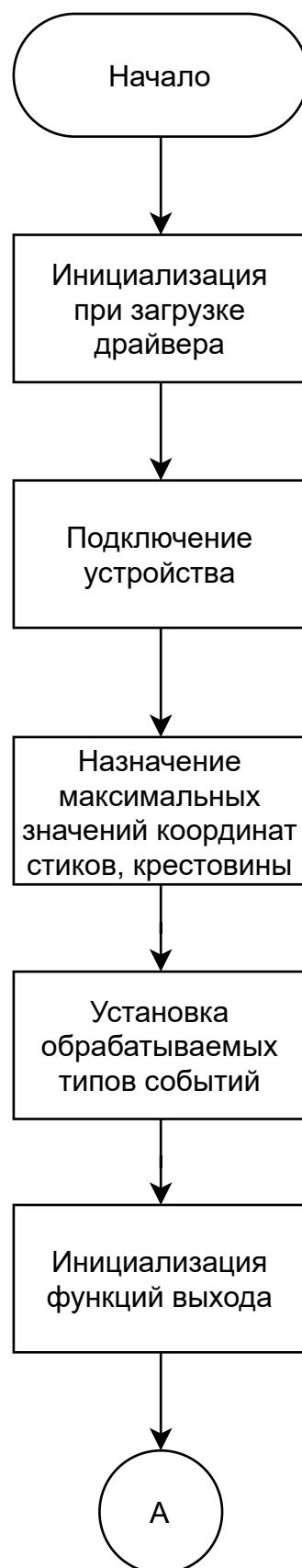


Рисунок 10 – Пример работы драйвера геймпада

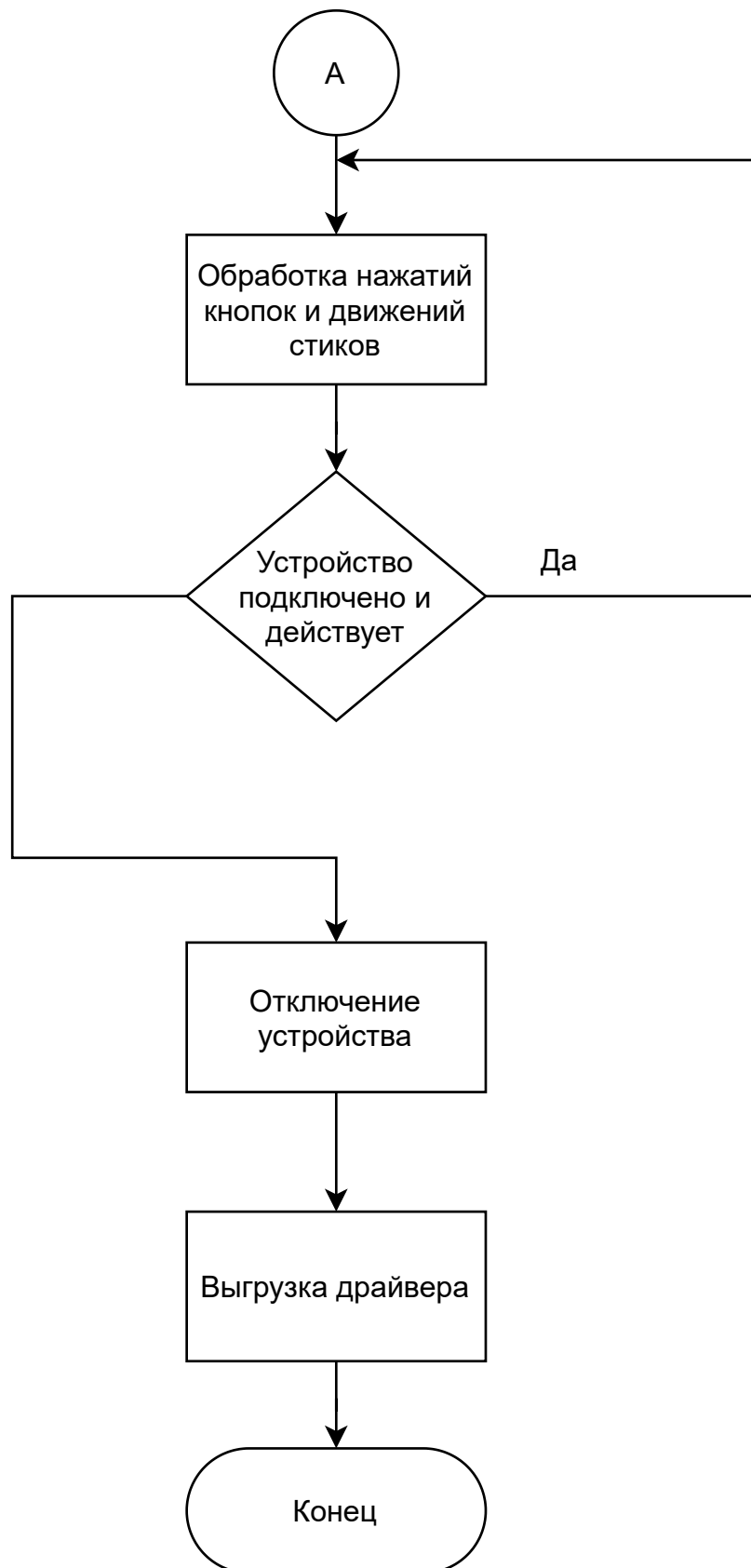


Рисунок 11 – Пример работы драйвера геймпада

3 Технологический раздел

3.1 Выбор языка и среды программирования

В качестве языка программирования для выполнения поставленной задачи был выбран язык C. Он является языком реализации большинства модулей и драйверов ОС Linux. В качестве компилятора был использован компилятор gcc. Средой разработки был выбран текстовый редактор Visual Studio Code.

3.2 Реализация структур и алгоритмов

Основной структурой USB драйвера является *struct usb_driver*. Данная структура представлена в Листинге 1.

Листинг 1 – Структура *usb_driver*

```
1  static struct usb_driver xpad_driver = {
2      .name      = "myxpad",
3      .probe      = xpad_probe,
4      .disconnect = xpad_disconnect,
5      .id_table   = xpad_table,
6  };
```

Ниже приведен Листинг 2 реализации *id_table* и связанной с ней структуры.

Листинг 2 – Структуры *xpad_device* и функция *xpad_table*

```
1  static const struct xpad_device {
2      u16 idVendor;
3      u16 idProduct;
4      char *name;
5      u8 mapping;
6      u8 xtype;
7  } xpad_device[] = {
8      { 0x046d, 0xc21d, "Logitech Gamepad F310", 0, XTYPE_XBOX360 },
9      { 0x046d, 0xc21e, "Logitech Gamepad F510", 0, XTYPE_XBOX360 },
10     { 0x046d, 0xc21f, "Logitech Gamepad F710", 0, XTYPE_XBOX360 }
11 };
12
13 static struct usb_device_id xpad_table[] = {
```

```

14         { USB_INTERFACE_INFO('X', 'B', 0) },      /* X-Box USB-IF not approved
class */
15         XPAD_XBOX360_VENDOR(0x046d),             /* Logitech X-Box 360 style
controllers */
16         { }
17     };

```

Локальной системной структурой является *usb_xpad*. Данная структура показана в Листинге 3.

Листинг 3 – Структура *usb_xpad*

```

1     struct usb_xpad {
2         struct input_dev *dev;      /* input device interface */
3         struct usb_device *udev;    /* usb device */
4         struct usb_interface *intf; /* usb interface */
5
6         int pad_present;
7
8         struct urb *irq_in;          /* urb for interrupt in report */
9         unsigned char *idata;        /* input data */
10        dma_addr_t idata_dma;
11
12        unsigned char *bdata;
13
14        struct urb *irq_out;          /* urb for interrupt out report */
15        unsigned char *odata;        /* output data */
16        dma_addr_t odata_dma;
17        struct mutex odata_mutex;
18
19        char phys[64];               /* physical device path */
20
21        int mapping;                 /* map d-pad to buttons or to axes */
22        int xtype;                   /* type of xbox device */
23    };

```

Ниже в Листинге 4 представлена функция инициализации при загрузке драйвера *xpad_probe*.

Листинг 4 – Функция *xpad_probe*

```

1   static int xpad_probe(struct usb_interface *intf, const struct
usb_device_id *id)
2   {
3       printk("My XPAD is Connected");
4
5       struct usb_device *udev = interface_to_usbdev(intf);
6       struct usb_xpad *xpad;
7       struct input_dev *input_dev;
8       struct usb_endpoint_descriptor *ep_irq_in;
9       int ep_irq_in_idx;
10      int i, error;
11
12      for (i = 0; xpad_device[i].idVendor; i++) {
13          if ((le16_to_cpu(udev->descriptor.idVendor) == xpad_device[i].
idVendor) &&
14              (le16_to_cpu(udev->descriptor.idProduct) == xpad_device[i].
idProduct))
15              break;
16      }
17
18      xpad = kzalloc(sizeof(struct usb_xpad), GFP_KERNEL);
19      input_dev = input_allocate_device();
20      if (!xpad || !input_dev) {
21          error = -ENOMEM;
22          input_free_device(input_dev);
23          kfree(xpad);
24          return error;
25      }
26
27      xpad->idata = usb_alloc_coherent(udev, XPAD_PKT_LEN,
GFP_KERNEL, &xpad->idata_dma);
28      if (!xpad->idata) {
29          error = -ENOMEM;
30          input_free_device(input_dev);
31          kfree(xpad);
32          return error;
33      }
34
35
36      xpad->irq_in = usb_alloc_urb(0, GFP_KERNEL);
37      if (!xpad->irq_in) {

```

```

38         error = -ENOMEM;
39         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->
idata_dma);
40         input_free_device(input_dev);
41         kfree(xpad);
42         return error;
43     }
44
45     xpad->udev = udev;
46     xpad->intf = intf;
47     xpad->mapping = xpad_device[i].mapping;
48     xpad->xtype = xpad_device[i].xtype;
49
50     if (xpad->xtype == XTYPE_UNKNOWN) {
51         if (intf->cur_altsetting->desc.bInterfaceClass ==
USB_CLASS_VENDOR_SPEC) {
52             if (intf->cur_altsetting->desc.bInterfaceProtocol == 129)
53                 xpad->xtype = XTYPE_XBOX360;
54             } else
55                 xpad->xtype = XTYPE_XBOX;
56
57             if (dpad_to_buttons)
58                 xpad->mapping |= MAP_DPAD_TO_BUTTONS;
59             if (triggers_to_buttons)
60                 xpad->mapping |= MAP_TRIGGERS_TO_BUTTONS;
61             if (sticks_to_null)
62                 xpad->mapping |= MAP_STICKS_TO_NULL;
63         }
64
65         xpad->dev = input_dev;
66         usb_make_path(udev, xpad->phys, sizeof(xpad->phys));
67         strlcat(xpad->phys, "/input0", sizeof(xpad->phys));
68
69         input_dev->name = xpad_device[i].name;
70         input_dev->phys = xpad->phys;
71         usb_to_input_id(udev, &input_dev->id);
72         input_dev->dev.parent = &intf->dev;
73
74         input_set_drvdata(input_dev, xpad);
75

```

```

76     input_dev->open = xpad_open;
77     input_dev->close = xpad_close;
78
79     input_dev->evbit[0] = BIT_MASK(EV_KEY);
80
81     if (!(xpad->mapping & MAP_STICKS_TO_NULL)) {
82         input_dev->evbit[0] |= BIT_MASK(EV_ABS);
83         /* set up axes */
84         for (i = 0; xpad_abs[i] >= 0; i++)
85             xpad_set_up_abs(input_dev, xpad_abs[i]);
86     }
87
88     /* set up standard buttons */
89     for (i = 0; xpad_common_btn[i] >= 0; i++)
90         __set_bit(xpad_common_btn[i], input_dev->keybit);
91
92     /* set up model-specific ones */
93     if (xpad->xtype == XTYPE_XBOX360) {
94         for (i = 0; xpad360_btn[i] >= 0; i++)
95             __set_bit(xpad360_btn[i], input_dev->keybit);
96     } else {
97         for (i = 0; xpad_btn[i] >= 0; i++)
98             __set_bit(xpad_btn[i], input_dev->keybit);
99     }
100
101     if (xpad->mapping & MAP_DPAD_TO_BUTTONS) {
102         for (i = 0; xpad_btn_pad[i] >= 0; i++)
103             __set_bit(xpad_btn_pad[i], input_dev->keybit);
104     } else {
105         for (i = 0; xpad_abs_pad[i] >= 0; i++)
106             xpad_set_up_abs(input_dev, xpad_abs_pad[i]);
107     }
108
109     if (xpad->mapping & MAP_TRIGGERS_TO_BUTTONS) {
110         for (i = 0; xpad_btn_triggers[i] >= 0; i++)
111             __set_bit(xpad_btn_triggers[i], input_dev->keybit);
112     } else {
113         for (i = 0; xpad_abs_triggers[i] >= 0; i++)
114             xpad_set_up_abs(input_dev, xpad_abs_triggers[i]);
115     }

```

```

116
117     for (i = 0; gamepad_buttons[i] >= 0; i++)
118         input_set_capability(input_dev, EV_KEY, gamepad_buttons[i]);
119
120     for (i = 0; directional_buttons[i] >= 0; i++)
121         input_set_capability(input_dev, EV_KEY, directional_buttons[i]);
122
123     for (i = 0; gamepad_abs[i] >= 0; i++)
124         input_set_capability(input_dev, EV_REL, gamepad_abs[i]);
125
126     error = xpad_init_output(intf, xpad);
127     if (error){
128         usb_free_urb(xpad->irq_in);
129         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->
130 idata_dma);
131         input_free_device(input_dev);
132         kfree(xpad);
133         return error;
134     }
135
136     ep_irq_in = &intf->cur_altsetting->endpoint[ep_irq_in_idx].desc;
137
138     usb_fill_int_urb(xpad->irq_in, udev,
139 usb_rcvintpipe(udev, ep_irq_in->bEndpointAddress),
140 xpad->idata, XPAD_PKT_LEN, xpad_irq_in,
141 xpad, ep_irq_in->bInterval);
142     xpad->irq_in->transfer_dma = xpad->idata_dma;
143     xpad->irq_in->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
144
145     error = input_register_device(xpad->dev);
146     if (error){
147         if (input_dev)
148             input_ff_destroy(input_dev);
149         xpad_deinit_output(xpad);
150         usb_free_urb(xpad->irq_in);
151         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->
152 idata_dma);
153         input_free_device(input_dev);
154         kfree(xpad);
155         return error;

```

```

154         }
155
156         usb_set_intfdata(intf, xpad);
157
158         return 0;
159     }

```

Ниже в Листинге 5 показана функция *xpad_open*, она используется при подключении геймпада.

Листинг 5 – Функция xpad_open

```

1     static int xpad_open(struct input_dev *dev)
2     {
3         printk("+ Gamepad is opened");
4         struct usb_xpad *xpad = input_get_drvdata(dev);
5
6         xpad->irq_in->dev = xpad->udev;
7         if (usb_submit_urb(xpad->irq_in, GFP_KERNEL))
8             return -EIO;
9
10        return 0;
11    }
12

```

Назначение максимальных значений координат стиков, крестовины производится в функции *xpad_set_up_abs*. Данная функция представлена в Листинге 6.

Листинг 6 – Функция xpad_set_up_abs

```

1     static void xpad_set_up_abs(struct input_dev *input_dev, signed short abs
2     )
3     {
4         struct usb_xpad *xpad = input_get_drvdata(input_dev);
5         set_bit(abs, input_dev->absbit);
6
7         switch (abs) {
8             case ABS_X:
9
10            case ABS_Y:

```



```

9         case ABS_RX:
10        case ABS_RY:    /* the two sticks */
11        input_set_abs_params(input_dev, abs, -32768, 32767, 16, 128);
12        break;
13        case ABS_Z:
14        case ABS_RZ:    /* the triggers (if mapped to axes) */
15        input_set_abs_params(input_dev, abs, 0, 255, 0, 0);
16        break;
17        case ABS_HAT0X:
18        case ABS_HAT0Y: /* the d-pad (only if dpad is mapped to axes) */
19        input_set_abs_params(input_dev, abs, -1, 1, 0, 0);
20        break;
21    }
22 }

```

Установка обрабатываемых типов событий производится в этих строках в функции *xpad_probe*. Ниже в Листинге 7 представлены эти строки.

Листинг 7 – Установка обрабатываемых типов событий

```

1    for (i = 0; gamepad_buttons[i] >= 0; i++)
2        input_set_capability(input_dev, EV_KEY, gamepad_buttons[i]);
3
4    for (i = 0; directional_buttons[i] >= 0; i++)
5        input_set_capability(input_dev, EV_KEY, directional_buttons[i]);
6
7    for (i = 0; gamepad_abs[i] >= 0; i++)
8        input_set_capability(input_dev, EV_REL, gamepad_abs[i]);

```

Ниже в Листинге 8 представлена инициализация функций выхода в *xpad_init_output*.

Листинг 8 – Инициализация функций выхода

```

1    static int xpad_init_output(struct usb_interface *intf, struct usb_xpad *
xpad)
2    {
3        printk("+ init output");
4        struct usb_endpoint_descriptor *ep_irq_out;
5        int ep_irq_out_idx;
6        int error;

```

```

7
8     if (xpad->xtype == XTYPE_UNKNOWN)
9         return 0;
10
11     xpad->odata = usb_alloc_coherent(xpad->udev, XPAD_PKT_LEN,
12                                     GFP_KERNEL, &xpad->odata_dma);
13     if (!xpad->odata) {
14         error = -ENOMEM;
15         return error;
16     }
17
18     mutex_init(&xpad->odata_mutex);
19
20     xpad->irq_out = usb_alloc_urb(0, GFP_KERNEL);
21     if (!xpad->irq_out) {
22         error = -ENOMEM;
23         usb_free_coherent(xpad->udev, XPAD_PKT_LEN, xpad->odata, xpad->
24                             odata_dma);
25         return error;
26     }
27
28     ep_irq_out = &intf->cur_altsetting->endpoint[ep_irq_out_idx].desc;
29
30     usb_fill_int_urb(xpad->irq_out, xpad->udev,
31                     usb_sndintpipe(xpad->udev, ep_irq_out->bEndpointAddress),
32                     xpad->odata, XPAD_PKT_LEN,
33                     xpad_irq_out, xpad, ep_irq_out->bInterval);
34     xpad->irq_out->transfer_dma = xpad->odata_dma;
35     xpad->irq_out->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
36
37     return 0;
38 }

```

Обработка нажатий кнопок и движений стиков происходит в функции *xpad_irq_in*. Данная функция представлена в Листинге 9.

Листинг 9 – Функция xpad_irq_in

```

1 static void xpad_irq_in(struct urb *urb)
2 {
3     printk("+ irq in");

```

```

4     unsigned char *data = urb->transfer_buffer;
5     struct usb_xpad *xpad = urb->context;
6     struct input_dev *dev = xpad->dev;
7
8     int retval;
9
10    switch (urb->status) {
11        case 0:
12            /* success */
13            break;
14        case -ECONNRESET:
15        case -ENOENT:
16        case -ESHUTDOWN:
17            /* this urb is terminated, clean up */
18            printk("%s - urb shutting down with status: %d", __func__, urb->
status);
19            return;
20        default:
21            printk("%s - nonzero urb status received: %d", __func__, urb->status)
;
22            retval = usb_submit_urb(urb, GFP_KERNEL);
23            if (retval)
24                dev_err(&urb->dev->dev, "%s - Error %d submitting interrupt urb\n",
__func__, retval);
25            return;
26    }
27
28    do_action(dev, EV_KEY, KEY_LEFT, data[2] & BUTTON_LEFT);
29    do_action(dev, EV_KEY, KEY_RIGHT, data[2] & BUTTON_RIGHT);
30    do_action(dev, EV_KEY, KEY_UP, data[2] & BUTTON_UP);
31    do_action(dev, EV_KEY, KEY_DOWN, data[2] & BUTTON_DOWN);
32
33    do_action(dev, EV_KEY, BTN_LEFT, data[3] & BUTTON_A);
34    do_action(dev, EV_KEY, BTN_RIGHT, data[3] & BUTTON_B);
35
36    do_action(dev, EV_KEY, KEY_LEFTSHIFT, data[3] & BUTTON_L1);
37    do_action(dev, EV_KEY, KEY_ENTER, data[3] & BUTTON_R1);
38
39    do_action(dev, EV_KEY, KEY_ESC, data[2] & BUTTON_START);
40    do_action(dev, EV_KEY, KEY_LEFTCTRL, data[2] & BUTTON_SELECT);

```

```

41     do_action(dev, EV_KEY, KEY_LEFTALT, data[3] & BUTTON_MODE);
42
43     do_action(dev, EV_KEY, KEY_PAGEDOWN, data[2] & BUTTON_L3);
44     do_action(dev, EV_KEY, KEY_PAGEUP, data[2] & BUTTON_R3);
45
46     do_action(dev, EV_REL, REL_X, ((__s16) le16_to_cpup((__le16 *) (data + 6))
/2048);
47     do_action(dev, EV_REL, REL_Y, ~((__s16) le16_to_cpup((__le16 *) (data + 8))
/2048);
48
49     do_action(dev, EV_REL, REL_HWHEEL, ((__s16) le16_to_cpup((__le16 *) (data +
10)))/8192);
50     do_action(dev, EV_REL, ABS_WHEEL, ~((__s16) le16_to_cpup((__le16 *) (data +
12)))/8192);
51
52     input_sync(dev);
53
54     retval = usb_submit_urb(urb, GFP_KERNEL);
55     if (retval)
56         dev_err(&urb->dev->dev, "%s - Error %d submitting interrupt urb\n",
__func__, retval);
57 }

```

Отключение устройства производится при использовании функции *xpad_disconnect*. Данная функция представлена в Листинге 10.

Листинг 10 – Функция xpad_disconnect

```

1     static void xpad_disconnect(struct usb_interface *intf)
2     {
3         printk("+ Gamepad is disconnected");
4         struct usb_xpad *xpad = usb_get_intfdata (intf);
5
6         input_unregister_device(xpad->dev);
7         xpad_deinit_output(xpad);
8
9         usb_free_urb(xpad->irq_in);
10        usb_free_coherent(xpad->udev, XPAD_PKT_LEN,
11        xpad->idata, xpad->idata_dma);
12
13        kfree(xpad->bdata);

```

```
14         kfree(xpad);
15
16         usb_set_intfdata(intf, NULL);
17     }
```

Выгрузка драйвера происходит в функции *xpad_close*. Данная функция представлена в Листинге 11.

Листинг 11 – Функция xpad_close

```
1     static void xpad_close(struct input_dev *dev)
2     {
3         printk("+ Closing");
4         struct usb_xpad *xpad = input_get_drvdata(dev);
5         usb_kill_urb(xpad->irq_out);
6     }
```

3.3 Makefile

В Листинге 12 приведено содержимое Makefile, содержащего набор инструкций, используемых утилитой make в инструментарии автоматизации сборки.

Листинг 12 – Makefile

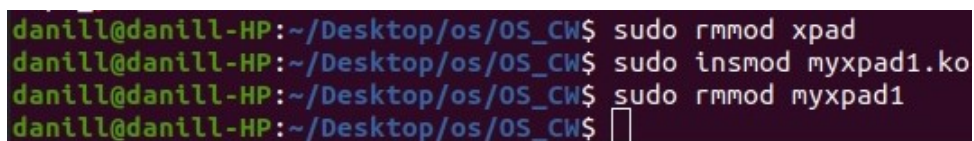
```
1     KBUILD_EXTRA_SYMBOLS = $(shell pwd)/Module.symverscd
2     ifneq ($(KERNELRELEASE),)
3         obj-m := myxpad.o
4     else
5         CURRENT = $(shell uname -r)
6         KDIR = /lib/modules/$(CURRENT)/build
7         PWD = $(shell pwd)
8
9     default:
10         $(MAKE) -C $(KDIR) M=$(PWD) modules
11         make cleanHalf
12
13     cleanHalf:
14         rm -rf *.o *~ *.mod *.mod.c Module.* *.order *.tmp_versions
15
16     clean:
```

```
17     make cleanHalf
18     rm -rf *.ko
19
20     endif
```

4 Исследовательский раздел

4.1 Загрузка драйвера

Для корректного функционирования разработанного ПО необходимо выполнить установку реализованного драйвера. Для этого сперва нужно выгрузить драйвер геймпада, что загружен по умолчанию — `sudo rmmod xpad`. Далее уже загрузить данное ПО — `sudo insmod myxpad1.ko`. На Рисунке 12 показан данный процесс.



```
danill@danill-HP:~/Desktop/os/OS_CW$ sudo rmmod xpad
danill@danill-HP:~/Desktop/os/OS_CW$ sudo insmod myxpad1.ko
danill@danill-HP:~/Desktop/os/OS_CW$ sudo rmmod myxpad1
danill@danill-HP:~/Desktop/os/OS_CW$
```

Рисунок 12 – Установка драйвера

4.2 Результат выполнения

На Рисунке 13 показан вывод программы.

```
[46760.459815] usb 3-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[46760.459819] usb 3-2: Product: Gamepad F310
[46760.459822] usb 3-2: Manufacturer: Logitech
[46760.459824] usb 3-2: SerialNumber: EBEF92AB
[46760.461734] My XPAD is Connected
[46760.461742] + set_up_abs.
[46760.461745] + set_up_abs.
[46760.461746] + set_up_abs.
[46760.461747] + set_up_abs.
[46760.461748] + set_up_abs.
[46760.461749] + set_up_abs.
[46760.461750] + set_up_abs.
[46760.461751] + set_up_abs.
[46760.461753] + init output
[46760.461815] input: Logitech Gamepad F310 as /devices/pci0000:00/0000:00:14.0/
usb3/3-2/3-2:1.0/input/input102
[46760.461821] + Gamepad is opened
[46760.467576] + irq in
[46760.471598] + irq in
[46760.475319] usbcore: registered new interface driver xpad
[46760.475574] + irq in
[46760.479590] + irq in
[46760.483559] + irq in
[46760.487558] + irq in
[46760.491556] + irq in
[46760.495556] + irq in
[46760.499556] + irq in
[46760.503559] + irq in
[46760.507557] + irq in
[46760.511616] + irq in
[46760.515622] + irq in
[46760.519602] + irq in
[46760.523619] + irq in
[46760.527600] + irq in
[46777.360228] usb 3-2: USB disconnect, device number 26
[46777.360401] + irq in
[46777.360417] xpad_irq_in - urb shutting down with status: -108
[46777.360424] + Gamepad is disconnected
[46777.440271] + Closing
[46782.243466] usbcore: deregistering interface driver myxpad
```

Рисунок 13 – Вывод программы

ЗАКЛЮЧЕНИЕ

В соответствии с заданием на курсовую работу по операционным системам был реализован загружаемый модуль ядра операционной системы Linux.

Были показаны и изучены структуры и состав USB шины, USB-драйвера, типы и предназначение оконечных точек, транзакции и пакеты.

В процессе разработки был реализован USB-драйвер, позволяющий управлять геймпадом как компьютерной мышью.

Исследованы способы реализации передачи данных геймпада и выбран наиболее подходящий из них.

Тестирование показало корректную работу данного программного обеспечения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Соловьев А. Разработка модулей ядра ОС Linux Kernel newbie's manual.
2. Corbet J., Rubini A., Kroah-Hartman G. Драйверы устройств Linux, Третья редакция.
3. Исходные коды ядра Linux. [Электронный ресурс]. – URL: <https://elixir.bootlin.com/linux>
4. Описание функций ядра Linux. [Электронный ресурс]. – URL: <https://www.chiark.greenend.org.uk>
5. Описание функций ядра Linux. [Электронный ресурс]. – URL: <https://www.kernel.org>
6. USB. [Электронный ресурс]. – URL: <http://perscom.ru/usb/>

ПРИЛОЖЕНИЕ А

Листинг 1: Драйвер

```
1  #include <linux/slab.h>
2  #include <linux/module.h>
3  #include <linux/usb/input.h>
4
5  //data[3]
6  #define BUTTON_A 0x10
7  #define BUTTON_B 0x20
8  #define BUTTON_X 0x40
9  #define BUTTON_Y 0x80
10
11 //data[2]
12 #define BUTTON_UP      0x01
13 #define BUTTON_DOWN    0x02
14 #define BUTTON_LEFT    0x04
15 #define BUTTON_RIGHT   0x08
16
17 //data[2]
18 #define BUTTON_SELECT   0x20
19 #define BUTTON_START    0x10
20 #define BUTTON_MODE     0x04
21
22 //data[3]
23 #define BUTTON_L1       0x01
24 #define BUTTON_R1       0x02
25
26 //data[2]
27 #define BUTTON_L3       0x40
28 #define BUTTON_R3       0x80
29
30 #define GAMEPAD_PKT_LEN 64
```

```

31 #define USB_GAMEPAD_MINOR_BASE      192
32 #define MAX_TRANSFER (PAGE_SIZE - 512)
33 #define WRITES_IN_FLIGHT 8
34
35 #define DRIVER_AUTHOR "Suslikov Daniil"
36 #define DRIVER_DESC "My Gamepad Driver"
37
38 #define XPAD_PKT_LEN 64
39 // #define EV_MAX 0x1f
40 /* xbox d-pads should map to buttons, as is required for DDR pads
41    but we map them to axes when possible to simplify things */
42 #define MAP_DPAD_TO_BUTTONS          (1 << 0)
43 #define MAP_TRIGGERS_TO_BUTTONS      (1 << 1)
44 #define MAP_STICKS_TO_NULL           (1 << 2)
45 #define DANCEPAD_MAP_CONFIG          (MAP_DPAD_TO_BUTTONS |
46                                     MAP_TRIGGERS_TO_BUTTONS | MAP_STICKS_TO_NULL)
47
48 #define XTYPE_XBOX                    0
49 #define XTYPE_XBOX360                 1
50 #define XTYPE_UNKNOWN                 4
51
52 static bool dpad_to_buttons;
53 module_param(dpad_to_buttons, bool, S_IRUGO);
54 MODULE_PARM_DESC(dpad_to_buttons, "Map D-PAD to buttons rather than
55
56 static bool triggers_to_buttons;
57 module_param(triggers_to_buttons, bool, S_IRUGO);
58 MODULE_PARM_DESC(triggers_to_buttons, "Map triggers to buttons rather
59
60 static bool sticks_to_null;
61 module_param(sticks_to_null, bool, S_IRUGO);
62 MODULE_PARM_DESC(sticks_to_null, "Do not map sticks at all for unknown
63
64
65 /* buttons shared with xbox and xbox360 */

```

```

66 static const signed short xpad_common_btn[] = {
67     BTN_A, BTN_B, BTN_X, BTN_Y,                /* "analog" buttons */
68     BTN_START, BTN_SELECT, BTN_THUMBL, BTN_THUMBR, /* start/back
69     -1                                           /* terminating entry */
70 };
71
72 /* original xbox controllers only */
73 static const signed short xpad_btn[] = {
74     BTN_C, BTN_Z,                /* "analog" buttons */
75     -1                          /* terminating entry */
76 };
77
78 /* used when dpad is mapped to buttons */
79 static const signed short xpad_btn_pad[] = {
80     BTN_TRIGGER_HAPPY1, BTN_TRIGGER_HAPPY2,        /* d-pad left, r
81     BTN_TRIGGER_HAPPY3, BTN_TRIGGER_HAPPY4,        /* d-pad up, dow
82     -1                          /* terminating entry */
83 };
84
85 /* used when triggers are mapped to buttons */
86 static const signed short xpad_btn_triggers[] = {
87     BTN_TL2, BTN_TR2,            /* triggers left/right */
88     -1
89 };
90
91
92 static const signed short xpad360_btn[] = { /* buttons for x360 co
93     BTN_TL, BTN_TR,              /* Button LB/RB */
94     BTN_MODE,                    /* The big X button */
95     -1
96 };
97
98 static const signed short xpad_abs[] = {
99     ABS_X, ABS_Y,                /* left stick */
100    ABS_RX, ABS_RY,              /* right stick */

```

```

101         -1                /* terminating entry */
102     };
103
104     /* used when dpad is mapped to axes */
105     static const signed short xpad_abs_pad[] = {
106         ABS_HAT0X, ABS_HAT0Y,    /* d-pad axes */
107         -1                /* terminating entry */
108     };
109
110     /* used when triggers are mapped to axes */
111     static const signed short xpad_abs_triggers[] = {
112         ABS_Z, ABS_RZ,          /* triggers left/right */
113         -1
114     };
115
116     static const signed short gamepad_buttons[] = {
117         BTN_LEFT, BTN_RIGHT, BTN_MIDDLE, BTN_SIDE,
118         KEY_ESC, KEY_LEFTCTRL, KEY_LEFTALT,
119         KEY_PAGEDOWN, KEY_PAGEUP,
120         KEY_LEFTSHIFT, KEY_ENTER,
121         -1 };
122
123     static const signed short directional_buttons[] = {
124         KEY_LEFT, KEY_RIGHT,      /* d-pad left, right */
125         KEY_UP, KEY_DOWN,         /* d-pad up, down */
126         -1 };
127
128     static const signed short trigger_buttons[] = {
129         BTN_TL2, BTN_TR2,         /* triggers left/right */
130         -1 };
131
132     static const signed short trigger_bumpers[] = {
133         ABS_Z, ABS_RZ,           /* triggers left/right */
134         -1 };
135

```

```

136 static const signed short gamepad_abs[] = {
137     ABS_X, ABS_Y,          /* left stick */
138     ABS_RX,  ABS_WHEEL,    /* right stick */
139     -1 };
140
141
142 /*
143  * Xbox 360 has a vendor-specific class, so we cannot match it with
144  * USB_INTERFACE_INFO (also specifically refused by USB subsystem),
145  * match against vendor id as well. Wired Xbox 360 devices have pro
146  * wireless controllers have protocol 129.
147  */
148 #define XPAD_XBOX360_VENDOR_PROTOCOL(vend,pr) \
149     .match_flags = USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH
150     .idVendor = (vend), \
151     .bInterfaceClass = USB_CLASS_VENDOR_SPEC, \
152     .bInterfaceSubClass = 93, \
153     .bInterfaceProtocol = (pr)
154 #define XPAD_XBOX360_VENDOR(vend) \
155     { XPAD_XBOX360_VENDOR_PROTOCOL(vend,1) }, \
156     { XPAD_XBOX360_VENDOR_PROTOCOL(vend,129) }
157
158 /* The Xbox One controller uses subclass 71 and protocol 208. */
159 #define XPAD_XBOXONE_VENDOR_PROTOCOL(vend, pr) \
160     .match_flags = USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH
161     .idVendor = (vend), \
162     .bInterfaceClass = USB_CLASS_VENDOR_SPEC, \
163     .bInterfaceSubClass = 71, \
164     .bInterfaceProtocol = (pr)
165 #define XPAD_XBOXONE_VENDOR(vend) \
166     { XPAD_XBOXONE_VENDOR_PROTOCOL(vend, 208) }
167
168 static const struct xpad_device {
169     u16 idVendor;
170     u16 idProduct;

```

```

171     char *name;
172     u8 mapping;
173     u8 xtype;
174 } xpad_device[] = {
175     { 0x046d, 0xc21d, "Logitech Gamepad F310", 0, XTYPE_XBOX360 },
176     { 0x046d, 0xc21e, "Logitech Gamepad F510", 0, XTYPE_XBOX360 },
177     { 0x046d, 0xc21f, "Logitech Gamepad F710", 0, XTYPE_XBOX360 },
178 };
179
180 static struct usb_device_id xpad_table[] = {
181     { USB_INTERFACE_INFO('X', 'B', 0) }, /* X-Box USB-IF not app
182     XPAD_XBOX360_VENDOR(0x046d), /* Logitech X-Box 360 style
183     { }
184 };
185
186 MODULE_DEVICE_TABLE(usb, xpad_table);
187
188 struct usb_xpad {
189     struct input_dev *dev; /* input device interface */
190     struct usb_device *udev; /* usb device */
191     struct usb_interface *intf; /* usb interface */
192
193     int pad_present;
194
195     struct urb *irq_in; /* urb for interrupt in report */
196     unsigned char *idata; /* input data */
197     dma_addr_t idata_dma;
198
199     unsigned char *bdata;
200
201     struct urb *irq_out; /* urb for interrupt out report */
202     unsigned char *odata; /* output data */
203     dma_addr_t odata_dma;
204     struct mutex odata_mutex;
205

```



```

206         char phys[64];                /* physical device path */
207
208         int mapping;                   /* map d-pad to buttons or to axes */
209         int xtype;                     /* type of xbox device */
210     };
211
212     void do_action(struct input_dev *dev, unsigned int type, unsigned int value)
213     {
214         unsigned long flags;
215
216         if (is_event_supported(type, dev->evbit, EV_MAX))
217         {
218             spin_lock_irqsave(&dev->event_lock, flags);
219             input_handle_event(dev, type, code, value);
220             spin_unlock_irqrestore(&dev->event_lock, flags);
221         }
222     }
223
224     static void xpad_irq_in(struct urb *urb)
225     {
226         printk("+ irq in");
227         unsigned char *data = urb->transfer_buffer;
228         struct usb_xpad *xpad = urb->context;
229         struct input_dev *dev = xpad->dev;
230
231         int retval;
232
233         switch (urb->status) {
234         case 0:
235             /* success */
236             break;
237         case -ECONNRESET:
238         case -ENOENT:
239         case -ESHUTDOWN:
240             /* this urb is terminated, clean up */

```

```

241         printk("%s - urb shutting down with status: %d", __func__,
242                return;
243     default:
244         printk("%s - nonzero urb status received: %d", __func__, urb->status);
245         retval = usb_submit_urb(urb, GFP_KERNEL);
246         if (retval)
247             dev_err(&urb->dev->dev, "%s - Error %d submitting interface urb\n", __func__,
248                    retval);
249     }
250
251     do_action(dev, EV_KEY, KEY_LEFT, data[2] & BUTTON_LEFT);
252     do_action(dev, EV_KEY, KEY_RIGHT, data[2] & BUTTON_RIGHT);
253     do_action(dev, EV_KEY, KEY_UP, data[2] & BUTTON_UP);
254     do_action(dev, EV_KEY, KEY_DOWN, data[2] & BUTTON_DOWN);
255
256     do_action(dev, EV_KEY, BTN_LEFT, data[3] & BUTTON_A);
257     do_action(dev, EV_KEY, BTN_RIGHT, data[3] & BUTTON_B);
258
259     do_action(dev, EV_KEY, KEY_LEFTSHIFT, data[3] & BUTTON_L1);
260     do_action(dev, EV_KEY, KEY_ENTER, data[3] & BUTTON_R1);
261
262     do_action(dev, EV_KEY, KEY_ESC, data[2] & BUTTON_START);
263     do_action(dev, EV_KEY, KEY_LEFTCTRL, data[2] & BUTTON_SELECT);
264     do_action(dev, EV_KEY, KEY_LEFTALT, data[3] & BUTTON_MODE);
265
266     do_action(dev, EV_KEY, KEY_PAGEDOWN, data[2] & BUTTON_L3);
267     do_action(dev, EV_KEY, KEY_PAGEUP, data[2] & BUTTON_R3);
268
269     do_action(dev, EV_REL, REL_X, ((__s16) le16_to_cpu((__le16 *) &data[4] & REL_X)));
270     do_action(dev, EV_REL, REL_Y, ~((__s16) le16_to_cpu((__le16 *) &data[4] & REL_Y)));
271
272     do_action(dev, EV_REL, REL_HWHEEL, ((__s16) le16_to_cpu((__le16 *) &data[4] & REL_HWHEEL)));
273     do_action(dev, EV_REL, ABS_WHEEL, ~((__s16) le16_to_cpu((__le16 *) &data[4] & ABS_WHEEL)));
274
275     input_sync(dev);

```

```

276
277     retval = usb_submit_urb(urb, GFP_KERNEL);
278     if (retval)
279         dev_err(&urb->dev->dev, "%s - Error %d submitting interrupt
280 }
281
282 static void xpad_irq_out(struct urb *urb)
283 {
284     printk("+ irq out.\n");
285     struct usb_xpad *xpad = urb->context;
286     struct device *dev = &xpad->intf->dev;
287     int retval, status;
288
289     status = urb->status;
290
291     switch (status) {
292     case 0:
293         /* success */
294         return;
295
296     case -ECONNRESET:
297     case -ENOENT:
298     case -ESHUTDOWN:
299         /* this urb is terminated, clean up */
300         dev_dbg(dev, "%s - urb shutting down with status: %d\n",
301             __func__, status);
302         return;
303
304     default:
305         dev_dbg(dev, "%s - nonzero urb status received: %d\n",
306             __func__, status);
307     }
308
309     retval = usb_submit_urb(urb, GFP_ATOMIC);
310     if (retval)

```

```

311         dev_err(dev, "%s - usb_submit_urb failed with result %d\n",
312                 __func__, retval);
313     }
314
315     static int xpad_init_output(struct usb_interface *intf, struct usb_
316     {
317         printk("+ init output");
318         struct usb_endpoint_descriptor *ep_irq_out;
319         int ep_irq_out_idx;
320         int error;
321
322         if (xpad->xtype == XTYPE_UNKNOWN)
323             return 0;
324
325         xpad->odata = usb_alloc_coherent(xpad->udev, XPAD_PKT_LEN,
326                                         GFP_KERNEL, &xpad->odata_dma);
327         if (!xpad->odata) {
328             error = -ENOMEM;
329             return error;
330         }
331
332         mutex_init(&xpad->odata_mutex);
333
334         xpad->irq_out = usb_alloc_urb(0, GFP_KERNEL);
335         if (!xpad->irq_out) {
336             error = -ENOMEM;
337             usb_free_coherent(xpad->udev, XPAD_PKT_LEN, xpad->odata, xp
338             return error;
339         }
340
341         ep_irq_out = &intf->cur_altsetting->endpoint[ep_irq_out_idx].de
342
343         usb_fill_int_urb(xpad->irq_out, xpad->udev,
344                         usb_sndintpipe(xpad->udev, ep_irq_out->bEndpointAddress
345                         xpad->odata, XPAD_PKT_LEN,

```

```

346         xpad_irq_out, xpad, ep_irq_out->bInterval);
347     xpad->irq_out->transfer_dma = xpad->odata_dma;
348     xpad->irq_out->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
349
350     return 0;
351 }
352
353 static void xpad_stop_output(struct usb_xpad *xpad)
354 {
355     usb_kill_urb(xpad->irq_out);
356 }
357
358 static void xpad_deinit_output(struct usb_xpad *xpad)
359 {
360     if (xpad->xtype != XTYPE_UNKNOWN) {
361         usb_free_urb(xpad->irq_out);
362         usb_free_coherent(xpad->udev, XPAD_PKT_LEN,
363             xpad->odata, xpad->odata_dma);
364     }
365 }
366
367 static int xpad_open(struct input_dev *dev)
368 {
369     printk("+ Gamepad is opened");
370     struct usb_xpad *xpad = input_get_drvdata(dev);
371
372     xpad->irq_in->dev = xpad->udev;
373     if (usb_submit_urb(xpad->irq_in, GFP_KERNEL))
374         return -EIO;
375
376     return 0;
377 }
378
379 static void xpad_close(struct input_dev *dev)
380 {

```

```

381     printk("+ Closing");
382     struct usb_xpad *xpad = input_get_drvdata(dev);
383     xpad_stop_output(xpad);
384 }
385
386 static void xpad_set_up_abs(struct input_dev *input_dev, signed short
387 {
388     printk("Check11.\n");
389     struct usb_xpad *xpad = input_get_drvdata(input_dev);
390     set_bit(abs, input_dev->absbit);
391
392     switch (abs) {
393     case ABS_X:
394     case ABS_Y:
395     case ABS_RX:
396     case ABS_RY:    /* the two sticks */
397         input_set_abs_params(input_dev, abs, -32768, 32767, 16, 128,
398         break;
399     case ABS_Z:
400     case ABS_RZ:    /* the triggers (if mapped to axes) */
401         input_set_abs_params(input_dev, abs, 0, 255, 0, 0);
402         break;
403     case ABS_HAT0X:
404     case ABS_HAT0Y:    /* the d-pad (only if dpad is mapped to axes) */
405         input_set_abs_params(input_dev, abs, -1, 1, 0, 0);
406         break;
407     }
408 }
409
410 static int xpad_probe(struct usb_interface *intf, const struct usb_
411 {
412     printk("My XPAD is Connected");
413
414     struct usb_device *udev = interface_to_usbdev(intf);
415     struct usb_xpad *xpad;

```

```

416 struct input_dev *input_dev;
417 struct usb_endpoint_descriptor *ep_irq_in;
418 int ep_irq_in_idx;
419 int i, error;
420
421 for (i = 0; xpad_device[i].idVendor; i++) {
422     if ((le16_to_cpu(udev->descriptor.idVendor) == xpad_device[
423         (le16_to_cpu(udev->descriptor.idProduct) == xpad_device
424         break;
425 }
426
427 xpad = kzalloc(sizeof(struct usb_xpad), GFP_KERNEL);
428 input_dev = input_allocate_device();
429 if (!xpad || !input_dev) {
430     error = -ENOMEM;
431     input_free_device(input_dev);
432     kfree(xpad);
433     return error;
434 }
435
436 xpad->idata = usb_alloc_coherent(udev, XPAD_PKT_LEN,
437     GFP_KERNEL, &xpad->idata_dma);
438 if (!xpad->idata) {
439     error = -ENOMEM;
440     input_free_device(input_dev);
441     kfree(xpad);
442     return error;
443 }
444
445 xpad->irq_in = usb_alloc_urb(0, GFP_KERNEL);
446 if (!xpad->irq_in) {
447     error = -ENOMEM;
448     usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->id
449     input_free_device(input_dev);
450     kfree(xpad);

```

```

451         return error;
452     }
453
454     xpad->udev = udev;
455     xpad->intf = intf;
456     xpad->mapping = xpad_device[i].mapping;
457     xpad->xtype = xpad_device[i].xtype;
458
459     if (xpad->xtype == XTYPE_UNKNOWN) {
460         if (intf->cur_altsetting->desc.bInterfaceClass == USB_CLASS
461             intf->cur_altsetting->desc.bInterfaceProtocol == 12
462             xpad->xtype = XTYPE_XBOX360;
463         } else
464             xpad->xtype = XTYPE_XBOX;
465
466         if (dpad_to_buttons)
467             xpad->mapping |= MAP_DPAD_TO_BUTTONS;
468         if (triggers_to_buttons)
469             xpad->mapping |= MAP_TRIGGERS_TO_BUTTONS;
470         if (sticks_to_null)
471             xpad->mapping |= MAP_STICKS_TO_NULL;
472     }
473
474     xpad->dev = input_dev;
475     usb_make_path(udev, xpad->phys, sizeof(xpad->phys));
476     strlcat(xpad->phys, "/input0", sizeof(xpad->phys));
477
478     input_dev->name = xpad_device[i].name;
479     input_dev->phys = xpad->phys;
480     usb_to_input_id(udev, &input_dev->id);
481     input_dev->dev.parent = &intf->dev;
482
483     input_set_drvdata(input_dev, xpad);
484
485     input_dev->open = xpad_open;

```



```

486     input_dev->close = xpad_close;
487
488     input_dev->evbit[0] = BIT_MASK(EV_KEY);
489
490     if (!(xpad->mapping & MAP_STICKS_TO_NULL)) {
491         input_dev->evbit[0] |= BIT_MASK(EV_ABS);
492         /* set up axes */
493         for (i = 0; xpad_abs[i] >= 0; i++)
494             xpad_set_up_abs(input_dev, xpad_abs[i]);
495     }
496
497     /* set up standard buttons */
498     for (i = 0; xpad_common_btn[i] >= 0; i++)
499         __set_bit(xpad_common_btn[i], input_dev->keybit);
500
501     /* set up model-specific ones */
502     if (xpad->xtype == XTYPE_XBOX360) {
503         for (i = 0; xpad360_btn[i] >= 0; i++)
504             __set_bit(xpad360_btn[i], input_dev->keybit);
505     } else {
506         for (i = 0; xpad_btn[i] >= 0; i++)
507             __set_bit(xpad_btn[i], input_dev->keybit);
508     }
509
510     if (xpad->mapping & MAP_DPAD_TO_BUTTONS) {
511         for (i = 0; xpad_btn_pad[i] >= 0; i++)
512             __set_bit(xpad_btn_pad[i], input_dev->keybit);
513     } else {
514         for (i = 0; xpad_abs_pad[i] >= 0; i++)
515             xpad_set_up_abs(input_dev, xpad_abs_pad[i]);
516     }
517
518     if (xpad->mapping & MAP_TRIGGERS_TO_BUTTONS) {
519         for (i = 0; xpad_btn_triggers[i] >= 0; i++)
520             __set_bit(xpad_btn_triggers[i], input_dev->keybit);

```

```

521     } else {
522         for (i = 0; xpad_abs_triggers[i] >= 0; i++)
523             xpad_set_up_abs(input_dev, xpad_abs_triggers[i]);
524     }
525
526     for (i = 0; gamepad_buttons[i] >= 0; i++)
527         input_set_capability(input_dev, EV_KEY, gamepad_buttons[i]);
528
529     for (i = 0; directional_buttons[i] >= 0; i++)
530         input_set_capability(input_dev, EV_KEY, directional_buttons[i]);
531
532     for (i = 0; gamepad_abs[i] >= 0; i++)
533         input_set_capability(input_dev, EV_REL, gamepad_abs[i]);
534
535     error = xpad_init_output(intf, xpad);
536     if (error){
537         usb_free_urb(xpad->irq_in);
538         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->id);
539         input_free_device(input_dev);
540         kfree(xpad);
541         return error;
542     }
543
544     ep_irq_in = &intf->cur_altsetting->endpoint[ep_irq_in_idx].desc;
545
546     usb_fill_int_urb(xpad->irq_in, udev,
547                     usb_rcvintpipe(udev, ep_irq_in->bEndpointAddress),
548                     xpad->idata, XPAD_PKT_LEN, xpad_irq_in,
549                     xpad, ep_irq_in->bInterval);
550     xpad->irq_in->transfer_dma = xpad->idata_dma;
551     xpad->irq_in->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
552
553     error = input_register_device(xpad->dev);
554     if (error){
555         if (input_dev)

```

```

556         input_ff_destroy(input_dev);
557         xpad_deinit_output(xpad);
558         usb_free_urb(xpad->irq_in);
559         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->id
560         input_free_device(input_dev);
561         kfree(xpad);
562         return error;
563     }
564
565     usb_set_intfdata(intf, xpad);
566
567     return 0;
568 }
569
570 static void xpad_disconnect(struct usb_interface *intf)
571 {
572     printk("+ Gamepad is disconnected");
573     struct usb_xpad *xpad = usb_get_intfdata (intf);
574
575     input_unregister_device(xpad->dev);
576     xpad_deinit_output(xpad);
577
578     usb_free_urb(xpad->irq_in);
579     usb_free_coherent(xpad->udev, XPAD_PKT_LEN,
580                       xpad->idata, xpad->idata_dma);
581
582     kfree(xpad->bdata);
583     kfree(xpad);
584
585     usb_set_intfdata(intf, NULL);
586 }
587
588 static struct usb_driver xpad_driver = {
589     .name          = "myxpad",
590     .probe         = xpad_probe,

```

```
591         .disconnect      = xpad_disconnect,
592         .id_table         = xpad_table,
593     };
594
595     module_usb_driver(xpad_driver);
596
597     MODULE_AUTHOR(DRIVER_AUTHOR);
598     MODULE_DESCRIPTION(DRIVER_DESC);
599     MODULE_LICENSE("GPL");
```