



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)

2022 г.

СОДЕРЖАНИЕ

| | |
|---|-----------|
| ВВЕДЕНИЕ | 3 |
| 1 Аналитическая часть | 4 |
| 1.1 Постановка задачи | 4 |
| 1.2 Загружаемый модуль ядра | 4 |
| 1.3 USB ядро и USB драйвер | 5 |
| 1.4 Оконечная точка | 6 |
| 1.5 Блоки запроса USB | 7 |
| 2 Конструкторская часть | 9 |
| 2.1 Требования к программному обеспечению | 9 |
| 2.2 Реализация драйвера | 9 |
| 2.3 Регистрация драйвера | 9 |
| 2.4 Подготовка к эмуляции событий мыши | 10 |
| 2.5 Обработка сообщений от устройства | 11 |
| 2.6 Схемы алгоритмов работы драйвера | 12 |
| 3 Технологическая часть | 15 |
| 3.1 Выбор языка и среды программирования | 15 |
| 3.2 Описание ключевых моментов реализации | 15 |
| 3.3 Makefile | 25 |
| 3.4 Установка драйвера | 25 |
| 3.5 Результат выполнения | 26 |
| ЗАКЛЮЧЕНИЕ | 27 |
| СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ | 28 |
| ПРИЛОЖЕНИЕ А | 29 |

ВВЕДЕНИЕ

Системное ядро Linux способно к модификации за счёт расширения функциональных возможностей. Это достигается несколькими способами, но самым оптимальным является подключение загружаемых модулей ядра. Этот способ позволяет вести разработку нужного функционала для ядра независимо от самого ядра.

Модуль ядра - это программа, которая может быть загружена в ядро операционной системы, или выгружена из него по запросу без перекомпиляции ядра или перезагрузки системы. Модули предназначены для расширения функциональности ядра. Ядро Linux позволяет драйверам оборудования, файловых систем, и некоторым другим компонентам быть скомпилированными отдельно — как модули, а не как часть самого ядра. Таким образом, вы можете обновлять драйвера для различных устройств не пересобирая ядро, а также динамически расширять его функциональность.

В качестве примера может быть создан загружаемый модуль ядра для геймпада. Геймпад — тип игрового манипулятора. Представляет собой пульт, который удерживается двумя руками, для контроля его органов управления используются большие пальцы рук (в современных геймпадах также часто используются указательные и средние пальцы).

Цель данной курсовой работы - разработать загружаемый модуль ядра Linux, который позволяет использовать функционал мыши при помощи геймпада. Реализовать аналоги левого и правого кликов мыши, прокрутки колеса и перемещение курсора.

1 Аналитическая часть

1.1 Постановка задачи

Цель данной работы является разработка и реализация загружаемого модуля ядра операционной системы Linux, который позволяет использовать функционал мыши при помощи геймпада. Реализовать аналоги левого и правого кликов мыши, прокрутки колеса и перемещение курсора. Для достижения поставленной цели следует решить следующие задачи:

- выполнить анализ системного драйвера геймпада;
- разработать загружаемый модуль, позволяющий использовать функционал компьютерной мыши при помощи геймпада;
- реализовать программное обеспечение.

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 динамическую вставку и удаление кода ядра в процессе работы.

Загружаемый объект ядра называется модулем. Модуль по своей сути примерно то же, что и обычная программа. Модуль так же имеет точку входа и выхода и находится в своем бинарном файле. Но модули имеют непосредственный доступ к структурам и функциям ядра. Для программ в пространстве пользователя этот доступ ограничен библиотечными интерфейсами компилятора[1].

Использование загружаемых модулей значительно упрощает изменение функциональности ядра и не требует ни полной перекомпиляции, ни перезагрузки.

Также преимущество загружаемых модулей заключается в возможности сократить расход памяти для ядра, загружая только необходимые модули.

1.3 USB ядро и USB драйвер

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.

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 оконечных точек и как USB драйверы связаны с интерфейсами USB, а не всего устройства USB.

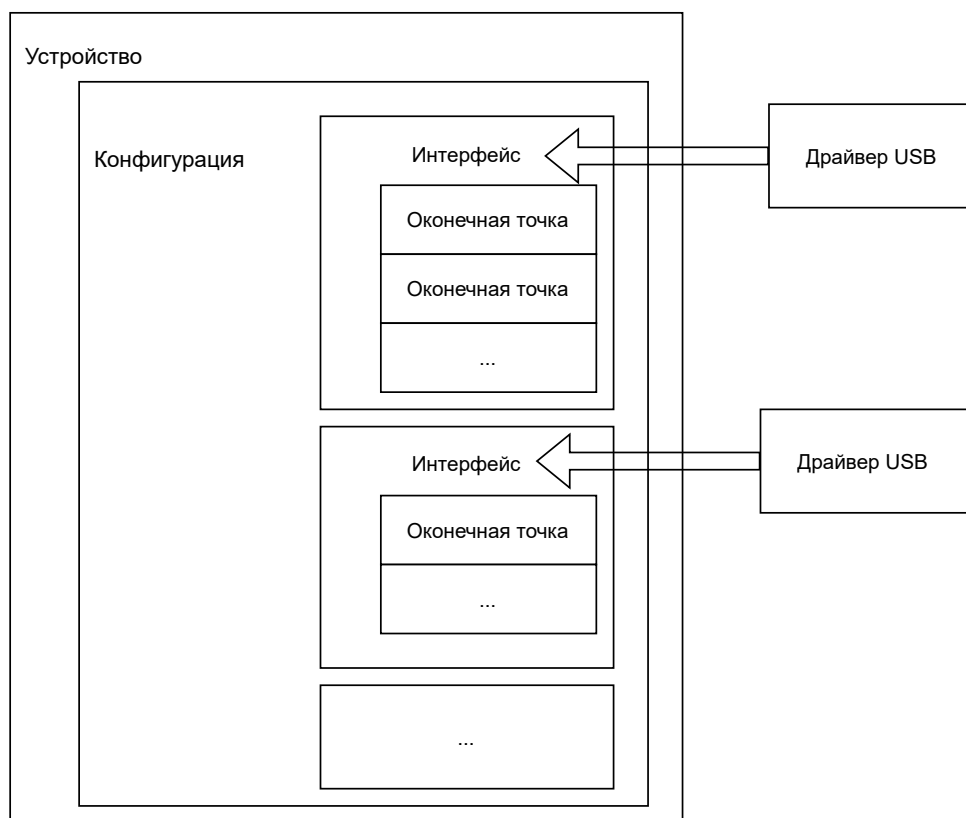


Рисунок 1 – Схема взаимодействия устройства и драйвера

1.4 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 односторонние 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, посылке 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

1.5 Блоки запроса USB

usb 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 usb-ов, так что перед тем, как очередь опустеет, к одной оконечной точке может быть отправлено множество usb-ов. Типичный жизненный цикл usb 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.

2 Конструкторская часть

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 Реализация драйвера

В основе данного ПО лежит реализация исходного USB драйвера геймпада. Его работа заключается в регистрации своего объекта драйвера с USB подсистемой.

2.3 Регистрация 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 2.

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.

После инициализации структуры *usb_device_id* выполняется вызов мак-

```

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 2 – Структура `usb_driver`

роса `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 Подготовка к 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 3 представлен её вид.

```

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

```

Рисунок 3 – 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 используется функция *input_report_key*, для изменения положения стиков — *input_report_rel*. Эти функции описаны в `/include/linux/input.h`. Ниже на Рисунке 4 приведён их вид.

```
static inline void input_report_key(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_KEY, code, !!value);
}

static inline void input_report_rel(struct input_dev *dev, unsigned int code, int value)
{
    input_event(dev, EV_REL, code, value);
}
```

Рисунок 4 — Функции *input_report_key* и *input_report_rel*

Видно, что по факту используется функция *input_event* с различными типами событий (EV_KEY и EV_REL).

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

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 5 и 6 представлена схема алгоритма работы драйвера геймпада.



Рисунок 5 – Схема алгоритма работы драйвера геймпада



Рисунок 6 – Схема алгоритма работы драйвера геймпада

3 Технологическая часть

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 Описание ключевых моментов реализации

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  };
```

Локальной системной структурой является *usb_xpad*. Данная структура показана в Листинге 2.

Листинг 2 – 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 3 представлена функция инициализации при загрузке драйвера *xpad_probe*.

Листинг 3 – 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,
28     GFP_KERNEL, &xpad->idata_dma);
29     if (!xpad->idata) {
30         error = -ENOMEM;
31         input_free_device(input_dev);
32         kfree(xpad);
33         return error;
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,

```

```

140     xpad, ep_irq_in->bInterval);
141     xpad->irq_in->transfer_dma = xpad->idata_dma;
142     xpad->irq_in->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
143
144     error = input_register_device(xpad->dev);
145     if (error){
146         if (input_dev)
147             input_ff_destroy(input_dev);
148         xpad_deinit_output(xpad);
149         usb_free_urb(xpad->irq_in);
150         usb_free_coherent(udev, XPAD_PKT_LEN, xpad->idata, xpad->
151 idata_dma);
152         input_free_device(input_dev);
153         kfree(xpad);
154         return error;
155     }
156
157     usb_set_intfdata(intf, xpad);
158
159     return 0;
160 }

```

Ниже в Листинге 4 показана функция *xpad_open*, она используется при подключении геймпада.

Листинг 4 – 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 5.

Листинг 5 – Функция *xpad_set_up_abs*

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

Листинг 6 – 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 7 представлена инициализация функций
выхода в *xpad_init_output*.

Листинг 7 – 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->
odata_dma);
24             return error;
25         }
26
27         ep_irq_out = &intf->cur_altsetting->endpoint[ep_irq_out_idx].desc;
28
29         usb_fill_int_urb(xpad->irq_out, xpad->udev,
30 usb_sndintpipe(xpad->udev, ep_irq_out->bEndpointAddress),
31 xpad->odata, XPAD_PKT_LEN,
```

```

32     xpad_irq_out, xpad, ep_irq_out->bInterval);
33     xpad->irq_out->transfer_dma = xpad->odata_dma;
34     xpad->irq_out->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
35
36     return 0;
37 }

```

Обработка нажатий кнопок и движений стиков происходит в функции *xpad_irq_in*. Данная функция представлена в Листинге 8.

Листинг 8 – Функция *xpad_irq_in*

```

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->
odata_dma);
24             return error;
25         }
26
27         ep_irq_out = &intf->cur_altsetting->endpoint[ep_irq_out_idx].desc;

```

```

28
29     usb_fill_int_urb(xpad->irq_out, xpad->udev,
30     usb_sndintpipe(xpad->udev, ep_irq_out->bEndpointAddress),
31     xpad->odata, XPAD_PKT_LEN,
32     xpad_irq_out, xpad, ep_irq_out->bInterval);
33     xpad->irq_out->transfer_dma = xpad->odata_dma;
34     xpad->irq_out->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
35
36     return 0;
37 }

```

Отключение устройства производится при использовании функции *xpad_disconnect*. Данная функция представлена в Листинге 9.

Листинг 9 – 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 10.

Листинг 10 – 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 11 приведено содержимое Makefile, содержащего набор инструкций, используемых утилитой make в инструментарии автоматизации сборки.

Листинг 11 – Makefile

```
1 KBUILD_EXTRA_SYMBOLS = $(shell pwd)/Module.symverscd
2 ifneq ($(KERNELRELEASE),)
3     obj-m := myxpad.o vms.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
```

3.4 Установка 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 myxpad.ko*. На Рисунке 7 показан дан-

ный 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 7 – Установка драйвера

3.5 Результат выполнения

На Рисунке 8 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 8 – Вывод программы

ЗАКЛЮЧЕНИЕ

В результате данной работы были достигнуты поставленные цели и задачи: был выполнен анализ системного драйвера геймпада, разработан и реализован загружаемый модуль ядра, предоставляющий функционал мыши при помощи нажатий клавиш и движений стиков.

В ходе работы был исследован системный драйвер геймпада и способы реализации собственного. Тестирование реализованного ПО показало способность выполнять поставленные задачи.

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>

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
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 static const struct xpad_device {
65     u16 idVendor;

```

```

66     u16 idProduct;
67     char *name;
68     u8 mapping;
69     u8 xtype;
70 } xpad_device[] = {
71     { 0x046d, 0xc21d, "Logitech Gamepad F310", 0, XTYPE_XBOX360 },
72     { 0x046d, 0xc21e, "Logitech Gamepad F510", 0, XTYPE_XBOX360 },
73     { 0x046d, 0xc21f, "Logitech Gamepad F710", 0, XTYPE_XBOX360 }
74 };
75
76 /* buttons shared with xbox and xbox360 */
77 static const signed short xpad_common_btn[] = {
78     BTN_A, BTN_B, BTN_X, BTN_Y,          /* "analog" buttons */
79     BTN_START, BTN_SELECT, BTN_THUMBL, BTN_THUMBR, /* start/back
80     -1                                     /* terminating entry */
81 };
82
83 /* original xbox controllers only */
84 static const signed short xpad_btn[] = {
85     BTN_C, BTN_Z,          /* "analog" buttons */
86     -1                     /* terminating entry */
87 };
88
89 /* used when dpad is mapped to buttons */
90 static const signed short xpad_btn_pad[] = {
91     BTN_TRIGGER_HAPPY1, BTN_TRIGGER_HAPPY2, /* d-pad left, r
92     BTN_TRIGGER_HAPPY3, BTN_TRIGGER_HAPPY4, /* d-pad up, dow
93     -1                                     /* terminating entry */
94 };
95
96 /* used when triggers are mapped to buttons */
97 static const signed short xpad_btn_triggers[] = {
98     BTN_TL2, BTN_TR2,      /* triggers left/right */
99     -1
100 };

```

```

101
102
103 static const signed short xpad360_btn[] = { /* buttons for x360 co
104     BTN_TL, BTN_TR,          /* Button LB/RB */
105     BTN_MODE,                /* The big X button */
106     -1
107 };
108
109 static const signed short xpad_abs[] = {
110     ABS_X, ABS_Y,            /* left stick */
111     ABS_RX, ABS_RY,          /* right stick */
112     -1                        /* terminating entry */
113 };
114
115 /* used when dpad is mapped to axes */
116 static const signed short xpad_abs_pad[] = {
117     ABS_HAT0X, ABS_HAT0Y,     /* d-pad axes */
118     -1                        /* terminating entry */
119 };
120
121 /* used when triggers are mapped to axes */
122 static const signed short xpad_abs_triggers[] = {
123     ABS_Z, ABS_RZ,           /* triggers left/right */
124     -1
125 };
126
127 static const signed short gamepad_buttons[] = {
128     BTN_LEFT, BTN_RIGHT, BTN_MIDDLE, BTN_SIDE,
129     KEY_ESC, KEY_LEFTCTRL, KEY_LEFTALT,
130     KEY_PAGEDOWN, KEY_PAGEUP,
131     KEY_LEFTSHIFT, KEY_ENTER,
132     -1 };
133
134 static const signed short directional_buttons[] = {
135     KEY_LEFT, KEY_RIGHT,      /* d-pad left, right */

```



```

136     KEY_UP, KEY_DOWN,          /* d-pad up, down */
137     -1 };
138
139 static const signed short trigger_buttons[] = {
140     BTN_TL2, BTN_TR2,          /* triggers left/right */
141     -1 };
142
143 static const signed short trigger_bumpers[] = {
144     ABS_Z, ABS_RZ,             /* triggers left/right */
145     -1 };
146
147 static const signed short gamepad_abs[] = {
148     ABS_X, ABS_Y,              /* left stick */
149     ABS_RX, ABS_WHEEL,         /* right stick */
150     -1 };
151
152
153 /*
154  * Xbox 360 has a vendor-specific class, so we cannot match it with
155  * USB_INTERFACE_INFO (also specifically refused by USB subsystem),
156  * match against vendor id as well. Wired Xbox 360 devices have pro
157  * wireless controllers have protocol 129.
158  */
159 #define XPAD_XBOX360_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 = 93, \
164     .bInterfaceProtocol = (pr)
165 #define XPAD_XBOX360_VENDOR(vend) \
166     { XPAD_XBOX360_VENDOR_PROTOCOL(vend,1) }, \
167     { XPAD_XBOX360_VENDOR_PROTOCOL(vend,129) }
168
169 /* The Xbox One controller uses subclass 71 and protocol 208. */
170 #define XPAD_XBOXONE_VENDOR_PROTOCOL(vend, pr) \

```

```

171     .match_flags = USB_DEVICE_ID_MATCH_VENDOR | USB_DEVICE_ID_MATCH
172     .idVendor = (vend), \
173     .bInterfaceClass = USB_CLASS_VENDOR_SPEC, \
174     .bInterfaceSubClass = 71, \
175     .bInterfaceProtocol = (pr)
176 #define XPAD_XBOXONE_VENDOR(vend) \
177     { XPAD_XBOXONE_VENDOR_PROTOCOL(vend, 208) }
178
179 static struct usb_device_id xpad_table[] = {
180     { USB_INTERFACE_INFO('X', 'B', 0) }, /* X-Box USB-IF not app
181     XPAD_XBOX360_VENDOR(0x046d), /* Logitech X-Box 360 style
182     { }
183 };
184
185 MODULE_DEVICE_TABLE(usb, xpad_table);
186
187 struct usb_xpad {
188     struct input_dev *dev; /* input device interface */
189     struct usb_device *udev; /* usb device */
190     struct usb_interface *intf; /* usb interface */
191
192     int pad_present;
193
194     struct urb *irq_in; /* urb for interrupt in report */
195     unsigned char *idata; /* input data */
196     dma_addr_t idata_dma;
197
198     unsigned char *bdata;
199
200     struct urb *irq_out; /* urb for interrupt out report */
201     unsigned char *odata; /* output data */
202     dma_addr_t odata_dma;
203     struct mutex odata_mutex;
204
205     char phys[64]; /* physical device path */

```

```

206
207     int mapping;                /* map d-pad to buttons or to axes */
208     int xtype;                  /* type of xbox device */
209 };
210
211 static void xpad_irq_in(struct urb *urb)
212 {
213     printk("+ irq in");
214     unsigned char *data = urb->transfer_buffer;
215     struct usb_xpad *xpad = urb->context;
216     struct input_dev *dev = xpad->dev;
217
218     int retval;
219
220     switch (urb->status) {
221     case 0:
222         /* success */
223         break;
224     case -ECONNRESET:
225     case -ENOENT:
226     case -ESHUTDOWN:
227         /* this urb is terminated, clean up */
228         printk("%s - urb shutting down with status: %d", __func__, urb->status);
229         return;
230     default:
231         printk("%s - nonzero urb status received: %d", __func__, urb->status);
232         retval = usb_submit_urb(urb, GFP_KERNEL);
233         if (retval)
234             dev_err(&urb->dev->dev, "%s - Error %d submitting interrupt urb\n", __func__, retval);
235         return;
236     }
237
238     input_report_key(dev, KEY_LEFT, data[2] & BUTTON_LEFT);
239     input_report_key(dev, KEY_RIGHT, data[2] & BUTTON_RIGHT);
240     input_report_key(dev, KEY_UP, data[2] & BUTTON_UP);

```

```

241     input_report_key(dev, KEY_DOWN, data[2] & BUTTON_DOWN);
242
243
244     input_report_key(dev, BTN_LEFT, data[3] & BUTTON_A);
245     input_report_key(dev, BTN_RIGHT, data[3] & BUTTON_B);
246     //input_report_key(dev, BTN_MIDDLE, data[3] & BTN_X);
247     //input_report_key(dev, BTN_SIDE, data[3] & BTN_Y);
248     input_report_key(dev, KEY_LEFTSHIFT, data[3] & BUTTON_L1);
249     input_report_key(dev, KEY_ENTER, data[3] & BUTTON_R1);
250
251     /* start/back buttons */
252     input_report_key(dev, KEY_ESC, data[2] & BUTTON_START);
253     input_report_key(dev, KEY_LEFTCTRL, data[2] & BUTTON_SELECT);
254     input_report_key(dev, KEY_LEFTALT, data[3] & BUTTON_MODE);
255
256     /* stick press left/right */
257     input_report_key(dev, KEY_PAGEDOWN, data[2] & BUTTON_L3);
258     input_report_key(dev, KEY_PAGEUP, data[2] & BUTTON_R3);
259
260     //digital bumpers
261     // input_report_key(dev, BTN_TL2, data[4]);
262     // input_report_key(dev, BTN_TR2, data[5]);
263
264     //analog bumpers
265     // input_report_abs(dev, ABS_VOLUME, data[4]);
266     // input_report_abs(dev, ABS_VOLUME, data[5]);
267
268     /* left stick */
269     input_report_rel(dev, REL_X, (__s16) le16_to_cpup((__le16 *) (data[4] & 0x0000ffff)));
270     input_report_rel(dev, REL_Y, ~((__s16) le16_to_cpup((__le16 *) (data[4] & 0x0000ffff))));
271
272     /* right stick */
273     input_report_rel(dev, REL_HWHEEL, (__s16) le16_to_cpup((__le16 *) (data[5] & 0x0000ffff)));
274     input_report_rel(dev, ABS_WHEEL, ~((__s16) le16_to_cpup((__le16 *) (data[5] & 0x0000ffff))));
275

```

```

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

```

```

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

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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