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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

Драйвер графического планшета Huion H640P

Студент ИУ7-73б _____ Вивчарук Р. В.
(группа) (подпись, дата) (И. О. Фамилия)

Руководитель курсового проекта _____ Ковтушенко А. П.
(подпись, дата) (И. О. Фамилия)

Консультант _____ _____
(подпись, дата) (И. О. Фамилия)

Москва, 2020 г.

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

УТВЕРЖДАЮ
Заведующий кафедрой ___ИУ7___
(Индекс)
__И. В.___Рудаков_
(И.О.Фамилия)
« _____ » _____ 20 ____ г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине _____Операционные системы_____

Студент группы _ИУ7-73б__

___Вивчарук Ростислав Владимирович__
(Фамилия, имя, отчество)

Тема курсового проекта Драйвер графического планшета Huion H640P_____
_____
_____

Направленность КП (учебный, исследовательский, практический, производственный, др.)
_____учебный_____
Источник тематики (кафедра, предприятие, НИР) __кафедра_____

График выполнения проекта: 25% к _4_ нед., 50% к _7_ нед., 75% к 11 нед., 100% к 14 нед.

Задание *__Разработать драйвер нулевого уровня привелегий для графического USB планшета под операционную систему Linux. Драйвер должен получать от устройства данные о_____ перемещении пера (стилуса) по поверхности планшета, касании пера поверхности планшета, нажатии кнопок на стилусе, степени давления на перо и корректно интерпретировать их__ операционной системе для использования планшета в качестве устройства взаимодействия_ человека с компьютером._____*
_____

Оформление курсового проекта:

Расчетно-пояснительная записка на _30-35_ листах формата А4.
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)
_____
_____
_____
_____

Дата выдачи задания « ___ » _____ 20__ г.

Руководитель курсового проекта _____ ____Ковтушенко А. П._
(Подпись, дата) (И.О.Фамилия)

Студент _____ _____Вивчарук Р. В___
(Подпись, дата) (И.О.Фамилия)

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Ядро операционный системы Linux - это сложный, монолитный программный комплекс, широко используемый в серверных и встраиваемых системах. Более половины всех ЭВМ мира использует Linux в качестве базовой системы. Конфигурации таких машин сильно варьируются, существует великое множество устройств обработки, ввода, вывода, хранения и передачи информации, отличающихся в физической и программной реализации. Именно поэтому для работы каждого устройства системы необходим собственный драйвер.

Драйвер - программа или часть кода ядра, предназначенная для управления конкретным видом оборудования [3]. Драйверы устройств играют первостепенную роль в производительности операционной системы в целом. С повышением популярности Linux стабильно растет и необходимость написания новых драйверов устройств.

Целью данной работы является создание драйвера для графического планшета Huion H640P, который может быть использован как устройство ввода похожее на мышь, но расширяющее ее возможности путем привнесения дополнительных параметров ввода, например, таких как степень давления на перо.

1 Аналитическая часть

1.1 Постановка задачи

В соответствии с техническим заданием, необходимо разработать драйвер для графического планшета Huion H640P для операционной системы Linux. Для достижении поставленной цели необходимо выполнить следующий ряд задач:

1. изучить особенности взаимодействия ОС Linux с USB-устройствами;
2. рассмотреть устройство графического планшета Huion H640P;
3. изучить подсистему ввода в ОС Linux;
4. спроектировать и реализовать драйвер для USB устройства.

1.2 USB-устройства

USB (англ. *Universal Serial Bus* — универсальная последовательная шина) — последовательный интерфейс для подключения периферийных устройств к вычислительной технике, получивший широчайшее распространение. ОС Linux имеет подсистему, называемую USB-ядром (USB-core), которая предоставляет интерфейс для USB драйверов. USB утилизирует модель главный/подчиненный (англ. *master/slave*), главным является только хост-компьютер, USB устройства являются подчиненными и не могут самостоятельно посылать информацию хост-компьютеру, а только отвечать на его запросы [3].

1.2.1 Конечные точки

Передача данных осуществляется между буфером в памяти хост-машины и так называемой конечной точкой USB-устройства (англ. *endpoint*). Это базовый объект связи интерфейса USB. Конечная точка может использоваться для

передачи данных с usb-устройства на хост-компьютер (IN) или наоборот (OUT). USB устройство может иметь до 16 конечных точек, каждая из которых может иметь два буфера (адреса): входной и выходной. Таким образом устройство может обладать 32 адресами конечных точек.

Данные отправляются и принимаются посредством передач, состоящих из ряда транзакций. Существует четыре типа передач, они определяют каким именно образом будут переданы данные:

1. `control`: используются для отправки устройству управляющих команд, получения информации об устройстве. Каждое usb-устройство имеет конечную точку данного типа;
2. `interrupt`: используются для регулярной передачи данных малого размера за фиксированный промежуток времени. Конечные точки данного типа являются основным методом передачи данных для usb клавиатур и мышей, не гарантируют доставку;
3. `bulk`: используются для надежной передачи больших объемов данных, не гарантируют транспортировку данных за фиксированное время. В основном используются принтерами, устройствами хранения данных, сетевыми устройствами, имеют самый низкий приоритет;
4. `isochronous`: используются для транспортировки больших объемов данных, где требуется поток в режиме реального времени, не гарантируют надежность передачи. Используются видео и аудио устройствами.

В ОС Linux данные конечной точки располагаются в структуре `struct usb_endpoint_descriptor` внутри `struct usb_host_endpoint`, листинг 1.

Листинг 1. Дескриптор конечной точки

```
1.  struct usb_endpoint_descriptor {
2.      __u8  bLength;
3.      __u8  bDescriptorType;
4.      __u8  bEndpointAddress;        // содержит адрес конечной точки и направление.
5.      __u8  bmAttributes;            // атрибуты, среди которых тип конечной точки.
```

```
6.      __le16 wMaxPacketSize;          // максимальный размер usb пакета.
7.      __u8  bInterval;                // интервал между запросами на прерывание для
8.      …                               // для конечных точек типа INTERRUPT.
9.  }
```

1.2.2 Интерфейсы

Конечные точки комплектуются в интерфейсы [1]. Один интерфейс обрабатывает одно логическое соединение. Некоторые устройства имеют несколько интерфейсов, например МФУ может быть подключено к компьютеру по одному USB-кабелю, но иметь несколько отдельных интерфейсов для управления принтером, факсом и сканером. Обычно каждый интерфейс контролируется отдельным драйвером. В Linux USB-интерфейс представлен структурой `struct usb_interface` (листинг 2).

Листинг 2. Структура USB-интерфейса

```
1.  struct usb_interface {
2.      struct usb_host_interface *altsetting;
3.      struct usb_host_interface *cur_altsetting;
4.      unsigned num_altsetting;
5.      ...
6.      int minor;
7.      ...
8.      struct device dev;
9.      struct device *usb_dev;
10.     ...
11. }
```

поле `altsetting` — массив структур `struct usb_host_interface`, которые содержат параметры интерфейса. Для одного интерфейса может существовать несколько альтернативных параметров, число которых определено в поле `num_altsettings`. На активные на данный момент параметры указывает указатель `cur_altsettings`. Сама структура `struct usb_host_interface`

агрегирует в себе структуру дескриптора конечной точки, которая была описана выше. Таким образом, один интерфейс может содержать в себе несколько конечных точек.

Поле `minor` хранит младший номер устройства и становится валидным после регистрации usb-устройства вызовом `usb_register_dev`.

Интерфейсы в свою очередь пакуются в конфигурации. Устройство может иметь несколько конфигураций, но обычно используется только одна, к тому же Linux плохо поддерживает работу с несколькими конфигурациями [1].

Связь конечных точек, интерфейсов, конфигураций и драйверов приведена на рисунке 1.

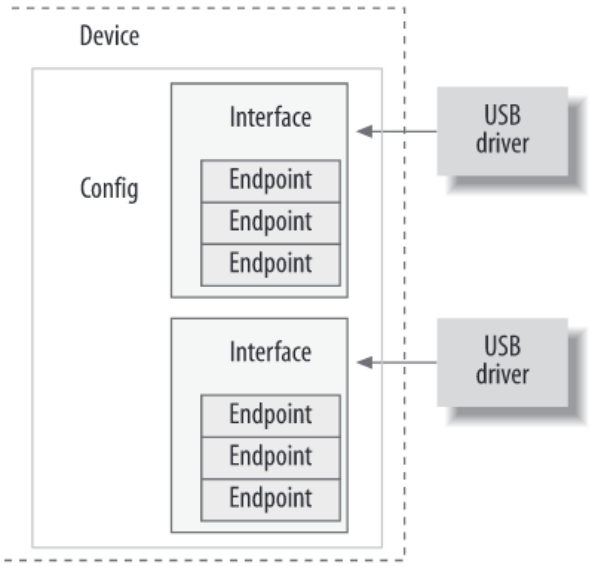

Рис. 1. Концептуальная схема usb-устройства

1.2.3 Блок запроса USB

Взаимодействие со всеми USB устройствами осуществляется с помощью URB (USB Request Block — блок запроса USB). URB используется для передачи/приема данных в/из конечной точки в асинхронном режиме. Драйвер usb-

устройства может аллоцировать несколько urb для одной конечной точки, либо использовать один urb для множества различных конечных точек в зависимости от нужд драйвера [1]. В структуре URB содержится обработчик, который вызывается после того как URB было выполнено. В листинге 3 приведена структура URB [2].

Листинг 3. Структура USB request block

```
1.  struct urb {
2.      struct usb_device *dev;        // указатель на ассоциированное usb-устройство.
3.      unsigned int pipe;
4.      unsigned int transfer_flags;
5.      void *context;                    // данные для обработчика по завершению urb.
6.      usb_complete_t complete; // указатель на функцию-обработчик по завершению urb.
7.      int status;
8.      void *transfer_buffer;       // буфер для передачи данных.
9.      u32 transfer_buffer_length;
10.     ...
11.     int interval;
12. }
```

Экземпляр URB должен обязательно создаваться динамически с помощью функции `usb_alloc_urb`. После чего он должен быть правильно инициализирован в зависимости от типа конечной точки, для чего существует ряд специальных функций. Например, инициализация urb для работы с конечной точкой типа `interrupt` производится с помощью следующей функции:

```
1.  void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
2.                        unsigned int pipe, void *transfer_buffer,
3.                        int buffer_length, usb_complete_t complete,
4.                        void *context, int interval);
```

После инициализации, запрос urb может быть отправлен на usb устройство для передачи или получения данных с помощью следующей функции:

```
1.  int usb_submit_urb(struct urb *urb, int mem_flags);
```

Запрос urb может быть отозван с помощью одной из двух следующих функций:

```
1.  int usb_unlink_urb(struct urb *urb);
2.  void usb_kill_urb(struct urb *urb);
```

1.3 Управление устройствами в Linux

Одна из задач операционной системы — управление устройствами. Unix/Linux рассматривают внешние устройства как специальные файлы устройств, который обеспечивают связь между файловой системой и драйверами устройств. Такая интерпретация специальных файлов обеспечивает доступ к внешним устройствам как к файлам, которые могут быть открыты/закрыты, из которых можно читать и в которые можно писать. Каждому внешнему устройству ОС ставит в соответствие минимум один файл в каталоге `/dev`.

Драйвер — это программа или часть кода ядра, которая предназначена для управления конкретным видом внешнего устройства. Необходимость драйверов устройств обусловлена тем, что каждое отдельное устройство воспринимает только свой строго фиксированный набор команд. Поэтому каждое устройство должно иметь свой программный драйвер, который выполняет роль связующего звена между аппаратной частью устройства и приложениями, использующими это устройство.

Драйверы в Linux бывают трёх типов:

1. встроенные в ядро (драйверы стандартного видеоконтроллера VGA, материнской платы и т. д.) ;
2. драйверы — загружаемые модули ядра;
3. код драйверов третьего типа поделен между кодом ядра и специальной утилитой, предназначенной для управления устройством.

Таким образом, взаимодействие прикладных программ с аппаратной частью компьютера в ОС Linux осуществляется по следующей схеме [3]:

устройство ↔ ядро ↔ специальный файл устройства ↔ программа пользователя.

1.3.1 Загружаемые модули ядра

UNIX/Linux — системы с монолитным ядром. Для расширения функционала такое ядро требует перекомпиляции, поэтому существуют средства для внесения изменений в ядро без его перекомпиляции. В UNIX/Linux они представлены загружаемыми модулями ядра. Это объектные модули с расширением .ko, которые могут быть загружены и выгружены во время работы операционной системы. Код простейшего загружаемого модуля ядра представлен в листинге 4.

Листинг 4. Простейший загружаемый модуль ядра

```
1.  #include <linux/init.h>
2.  #include <linux/module.h>
3.
4.  MODULE_AUTHOR(...);
5.  MODULE_DESCRIPTION(...);
6.  MODULE_LICENSE(«GPL»);
7.
8.  static int __init lkm_init(void) {
9.      printk(KERN_INFO «Hello world!\n»);
10.     return 0;
11. }
12.
13. static void __exit lkm_exit(void) {
14.     printk(KERN_INFO «Goodbye, world!\n»);
15. }
16
17  module_init(lkm_init);
18  module_exit(lkm_exit);
```

функции `init` и `exit` являются обязательными, они вызываются при загрузке и выгрузке модуля соответственно. Для загрузки модуля используется команда `insmod <имя модуля>`, для выгрузки `rmmod <имя модуля>`. Список всех загруженных модулей ядра можно получить с помощью команды `lsmod`.

Для компиляции модуля используется специальный Makefile:

Листинг 5. Makefile для загружаемого модуля ядра

```
1.  obj-m := module_name.o
2.  KDIR = /lib/modules/$(shell uname -r)/build
3.
4.  default:
5.      Make -C $(KDIR) M=$(shell pwd) modules
6.  clean:
7.      Make -C $(KDIR) M=$(shell pwd) clean
```

1.3.2 Драйверы USB-устройств

Драйвер USB-устройства в Linux представляется следующей структурой:

Листинг 6. Структура USB-драйвера

```
1.  struct usb_driver {
2.      const char *name;
3.      int (*probe)(struct usb_interface *intf, const struct usb_device_id *id);
4.      void (*disconnect)(struct usb_interface *intf);
5.      …
6.      const struct usb_device_id *id_table;
7.      …
8.  };
```

Поле `name` содержит имя драйвера, которое должно быть уникальным среди всех usb-драйверов в ядре и обычно устанавливается равным имени модуля драйвера.

Функция обратного вызова `probe` вызывается чтобы узнать, готов ли драйвер управлять определенным интерфейсом устройства. Если да, то

происходит создание нужных драйверу структур, связывание внутренних данных драйвера с интерфейсом с помощью функции usb_set_infdata. Параметр функции probe, const struct usb_device_id *id является структурой, которая содержит список всех устройств, с которыми может работать данный драйвер. Данная структура обязательна к заполнению, в противном случае функция probe не будет вызвана совсем. Пример заполнения таблицы устройств описан в листинге 7.

Листинг 7. Заполнение структуры `struct usb_device_id`

```
1.  #define USB_VENDOR_ID  0x046d       // USB идентификатор производителя устройства.
2.  #define USB_PRODUCT_ID 0xc52f       // USB идентификатор устройства.
3.  static struct usb_device_id id_table[] = {
4.      { USB_DEVICE(USB_VENDOR_ID, USB_PRODUCT_ID) },
5.      …
6.  };
7.  …
8.  MODULE_DEVICE_TABLE(usb, id_table);
```

Функция disconnect вызывается, если интерфейс больше не нужен в случае его отключения или выгрузки модуля драйвера.

Для регистрации/дерегистрации устройства драйвером используются функции usb_register_dev, usb_deregister_dev.

Для регистрации/дерегистрации самого usb-драйвера в системе используются функции usb_register и usb_deregister, в качестве параметров им передается указатель на экземпляр структуры драйвера struct usb_driver. Обычно эти процедуры вызываются в функциях init и exit загружаемого модуля ядра.

1.4 Обработка прерываний в Linux

Прерывание — это сигнал контроллеру о наступлении высокоприоритетного события. Если сигнал поступил, то процесс переходит на

выполнение обработчика прерывания. Все прерывания от устройств ввода/вывода и системного таймера аппаратные, они выполняются на высоком уровне приоритета, другие операции не могут выполняться пока аппаратное прерывание не завершено, поэтому прерывание должно быть выполнено как можно быстрее, чтобы не снижать отзывчивость системы.

Существуют быстрые прерывания и медленные. В настоящее время в Linux быстрым является только прерывание от системного таймера. Медленные прерывания в Linux принято делить на две части:

- верхняя половина (англ. *top half*);
- нижняя половина (англ. *bottom half*).

Фактически аппаратным прерыванием является верхняя половина, в ее задачи входит получение данных из регистра устройства, помещение их в буфер ядра, и инициализация выполнения нижней половины как отложенного действия, которое будет выполнено позже. Такой принцип обработки прерываний позволяет верхней половине запланировать новое пришедшее прерывание, пока нижняя половина все еще выполняется.

Для реализации нижних половин используют следующие механизмы:

- `softirq` – отложенные прерывания;
- тасклеты (англ. *tasklet*);
- очереди работ (англ. *work queue*).

1.4.1 Тасклеты

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний, для которых обработчик не может выполняться одновременно на нескольких процессорах. Тасклет планируется для одного конкретного процессора и на нем же выполняется. Они могут быть зарегистрированы в системе как статически так и динамически. Структура тасклета представлена в листинге 8.

Листинг 8. Структура тасклета

```
1.  struct tasklet_struct {
2.      struct tasklet_struct *next;            // следующий тасклет в списке.
3.      unsigned long state;                    // состояние тасклета.
4.      atomic_t count;
5.      void (*func) (unsigned long);           // функция-обработчик.
6.      unsigned long data;                     // аргумент функции-обработчика.
7.  };
```

Когда тасклет запланирован, он добавляется в очередь. Пока он находится в этом состоянии, запланировать его еще раз не получится — в этом случае просто ничего не произойдет. После того как он был запланирован, он выполнится один раз.

Для статического создания тасклета используется макрос `DECLARE_TASKLET(name, func, data)`.

Для того, чтобы запланировать тасклет на выполнение используется функция `void tasklet_schedule(struct tasklet_struct *t)`.

1.6 Подсистема ввода в Linux

Начиная с версии ядра 2.4 в Linux существует подсистема ввода (input subsystem), призванная управлять всеми устройствами ввода: клавиатуры, мыши, джойстики, планшеты и др. Подсистема предоставляет единообразную обработку функционально похожих устройств ввода, удобный интерфейс для отправки сообщений о событиях ввода. На рисунке 2 представлена связь между драйверами, обработчиками событий ввода и ядром подсистемы ввода (input core) [4].

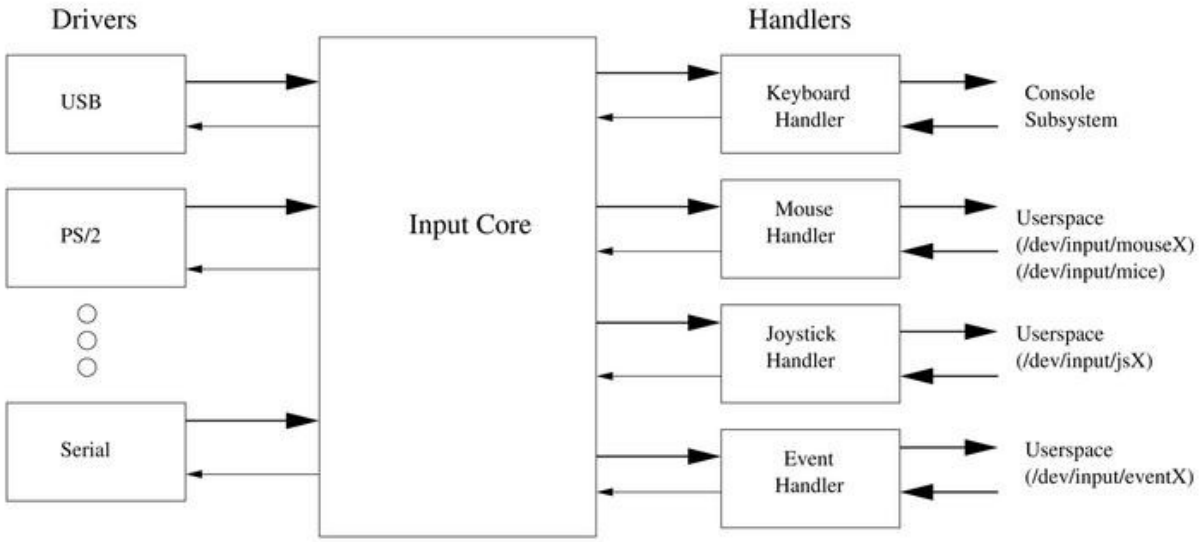

Рис. 2. input subsystem

1.6.1 Регистрация устройства ввода

Для того чтобы драйвер можно было использовать с подсистемой ввода, он должен создать экземпляр структуры `struct input_dev` с помощью функции `input_allocate_device`, проинициализировать ее определенным образом и затем зарегистрировать устройство ввода с помощью функции `input_register_device`.

Листинг 9. Структура данных устройства ввода

```
1.  struct input_dev {
2.      const char *name;
3.      const char *phys;
4.      …
5.      unsigned long evbit[BITS_TO_LONGS(EV_CNT)];
6.      unsigned long keybit[BITS_TO_LONGS(EV_CNT)];
7.      unsigned long relbit[BITS_TO_LONGS(EV_CNT)];
8.      unsigned long absbit[BITS_TO_LONGS(EV_CNT)];
9.      unsigned long mscbit[BITS_TO_LONGS(EV_CNT)];
10.     …
11.     struct device dev;
12. };
```

поле evbit является битовой картой типов событий, которые поддерживаются устройством (могут быть им сгенерированы). Существуют события следующих типов:

- keybit — события клавиатуры;
- relbit — события устройств с относительной системой координат (мышь, тачпад, трекбол);
- absbit — события устройств с абсолютной системой координат (джойстик, графический планшет);
- mscbit — события, которые нельзя отнести к вышеперечисленным категориям.

В процессе инициализации структуры struct input_device необходимо установить нужные биты для событий ввода с помощью функции set_bit. Определения соответствующих битов могут быть найдены в файле /usr/include/linux/input-event-codes.h. После чего можно регистрировать устройство ввода. Пример инициализации описываемой структуры для мыши приведен в листинге 10.

Листинг 10. Пример инициализации struct input_dev

```
1.  struct input_dev *mouse = input_allocate_device();
2.  // … обработка кода ошибки
3.  // … заполнение полей name, phys
4.
5.  set_bit(EV_KEY, mouse→evbit);
6.  set_bit(EV_REL, mouse→evbit);
7.  set_bit(BTN_LEFT, mouse→keybit);
8.  set_bit(BTN_RIGHT, mouse→keybit);
9.  set_bit(REL_X, mouse->relbit);
10. set_bit(REL_Y, mouse->relbit);
11. input_register_device(mouse);
```

Для освобождения структуры используется вызов input_free_device.

1.6.2 Генерация событий ввода

Для отправки событий ввода определенного типа используется ряд следующих функций:

Листинг 11. Отправка событий ввода

```
1.  input_report_key(struct input_dev *dev, uint code, int value);
2.  input_report_rel(struct input_dev *dev, uint code, int value);
3.  input_report_abs(struct input_dev *dev, uint code, int value);
4.  input_event(struct input_dev * dev, uint type, uint code, int value);
```

первые три функции являются оберткой над общей функцией отчета о событии ввода `input_event`. События, рапортированные описанными функциями буферизуются и будут отправлены ядру подсистемы только после вызова функции `input_sync(struct input_dev *dev)`. Данный вызов означает, что устройство передало все необходимые данные и они могут быть синхронизированы с общим состоянием подсистемы ввода [4].

1.7 Устройство графического планшета Huion H640P

Графический планшет Huion H640P состоит из плоского планшета, подключаемого к компьютеру через USB-интерфейс и пера (стилуса). Перо содержит две кнопки и слегка подвижный наконечник, который регистрирует степень давления на него. Уровень давления ранжируется от 0 до 8191. USB устройство имеет два интерфейса, каждый из которых содержит по одной конечной точке типа `IN` с типом передачи `interrupt`. Интерфейс, ответственный за передачу данных стилуса (координаты, кнопки, давление) имеет номер 1. Информация об интерфейсах может быть получена с помощью команды `lsusb -v -d <VENDOR_ID>:<PRODUCT_ID>`. Изображение устройства приведено на рисунке 3.

Рис. 3. Huion H640P

1.7.1 События, генерируемые устройством

Как уже было описано ранее, графические планшеты — это устройства, которые работают в абсолютной системе координат для имитации процесса рисования на обычном листе бумаги, следовательно, они генерируют события типа `EV_ABS`. Так как на стилусе присутствуют кнопки, определены события типа `EV_KEY`.

Среди абсолютных событий присутствуют следующие:

- `ABS_X` — передача координаты стилуса по оси O*x*;
- `ABS_Y` — передача координаты стилуса по оси O*y*;
- `ABS_PRESSURE` — передача уровня давления на стилус.

События клавиш:

- `BTN_TOOL_PEN` — событие, означающее, что стилус находится в рабочей области планшета (< 2 сантиметров над поверхностью планшета);

- BTN_TOUCH - событие, означающее касание пера поверхности планшета. Код этого события аналогичен коду нажатия левой кнопки мыши BTN_LEFT;
- BTN_STYLUS, BTN_STYLUS2 — события, означающие нажатие кнопок, расположенных на стилусе.

1.7.2 Структура передаваемых данных и их интерпретация

Устройство передает пакеты размером в 8 байт, структура пакета приведена на рисунке 4.

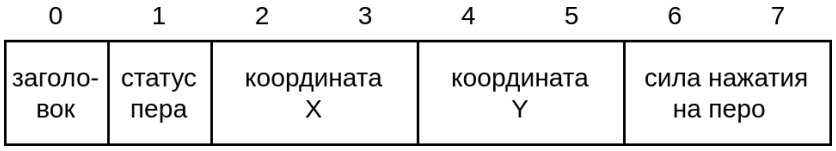

Рис. 4. Структура пакета

Байт 1 (статус пера) содержит 4 активных флага, установка которых означает следующее:

- бит 0: перо находится в рабочей зоне планшета;
- бит 5: на стилусе зажата кнопка №2;
- бит 6: на стилусе зажата кнопка №1;
- бит 7: перо касается поверхности планшета.

1.8 Вывод

В ходе аналитической части проекта были изучены все необходимые для реализации драйвера принципы взаимодействия ОС Linux с USB-устройствами, устройство графического планшета Huion H640P.

2 Конструкторская часть

2.1 Структура загружаемого модуля ядра

Спроектированный модуль состоит из следующих функций:

- `drawpad_driver_init` — точка входа в драйвер;
- `drawpad_driver_exit` — точка выхода из драйвера;
- `probe` — функция, вызываемая при подключении устройства для проверки, готов ли драйвер управлять конкретным интерфейсом устройства;
- `probe_interface_1` — функция инициализации данных драйвера для интерфейса устройства №1;
- `disconnect` — функция, вызываемая при отключении устройства;
- `disconnect_interface_1` — функция, назначение которой — освобождение ресурсов, используемых драйвером для управления интерфейсом №1, вызывается в теле функции `disconnect`;
- `open_interface_1` — функция, вызываемая при открытии файла устройства (интерфейса №1);
- `close_interface_1` — функция, вызываемая при закрытии файла устройства (интерфейса №1);
- `drawpad_irq` — верхняя половина прерывания, вызывается, когда блок запроса USB завершил передачу данных из устройства в буфер драйвера;
- `tasklet_handler` — нижняя половина прерывания, реализованная как тасклет, ставится на выполнение верхней половиной;

2.2 Инициализация загружаемого модуля ядра

Необходимо заполнить таблицу идентификаторов поддерживаемых драйвером устройств. Для производителя *Huion Animation* идентификатор равен `0x256c`, для планшета *H640P* — `0x006d`. Таблица представлена в листинге 2.1

Листинг 2.1. Таблица поддерживаемых драйвером устройств

```
1.  #define VENDOR_ID  0x256c
2.  #define PRODUCT_ID 0x006d
3.
4.  static struct usb_device_id devices_table[] = {
5.      { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
6.      { }
7.  };
8.  MODULE_DEVICE_TABLE(usb, devices_table);
```

Для инициализации драйвера требуется заполнить структуру `struct usb_driver` (листинг 2.2).

Листинг 2.2. struct usb_driver

```
1.  static struct usb_driver drawpad_driver = {
2.      .name = DRIVER_NAME,
3.      .id_table = devices_table,
4.      .probe = probe,
5.      .disconnect = disconnect,
6.  };
```

Функции `probe` и `disconnect` будут описаны позже.

Обязательными для инициализации загружаемого модуля ядра являются функции `__init` и `__exit`, где происходит регистрация и дерегистрация драйвера соответственно (рисунок 5).

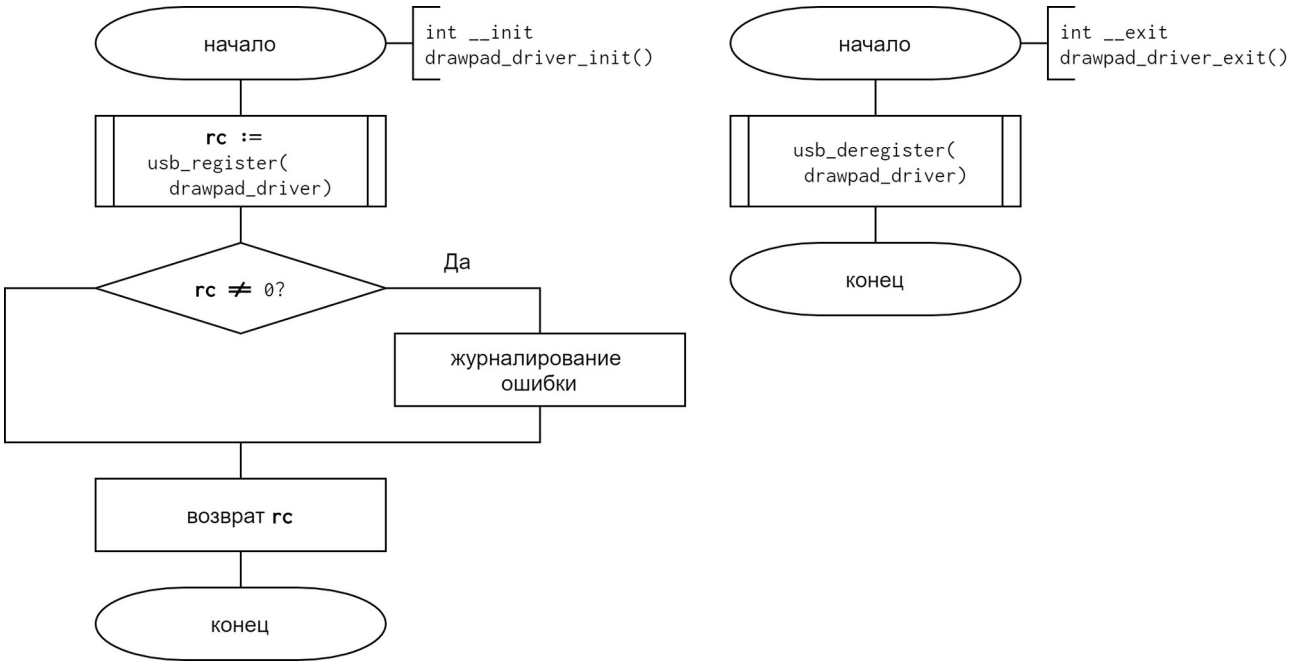

Рис. 5. схемы алгоритмов функций init, exit

2.3 Подключение устройства

При подключении USB-устройства функция `probe` вызывается для каждого интерфейса. В рассматриваемом планшете интерфейс отвечающий за передачу данных о работе стилуса имеет индекс 1. Интерфейс содержит одну конечную точку с адресом `0x82`. На рисунках 6-8 приведены схемы алгоритма функций `probe` и `probe_interface_1`.

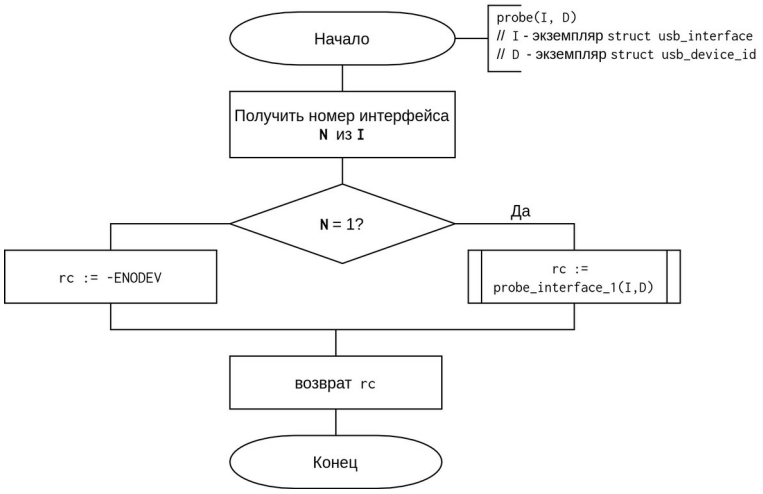

Рис. 6. схема алгоритма функции probe, часть 1

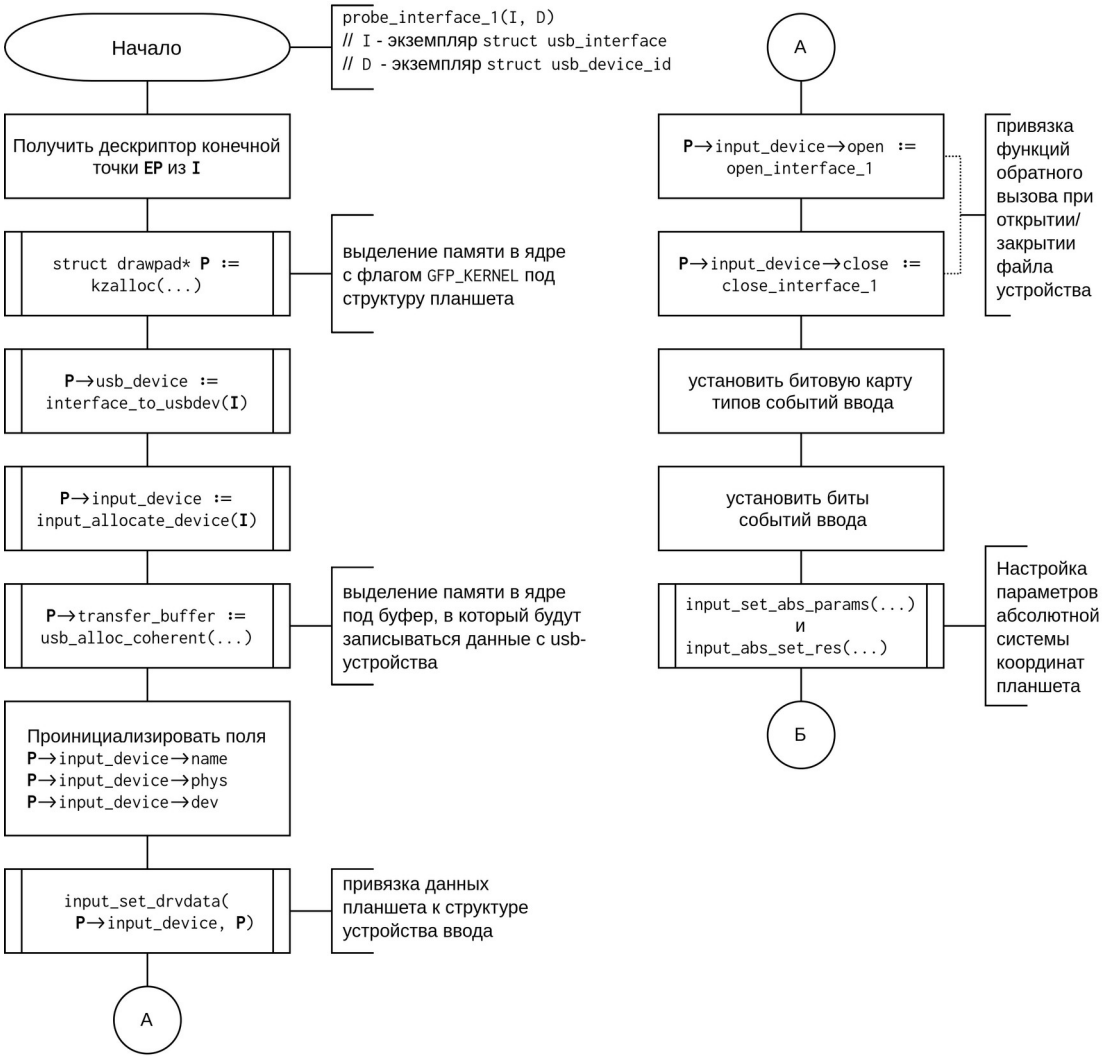

Рис. 7. схема алгоритма функции probe, часть 2

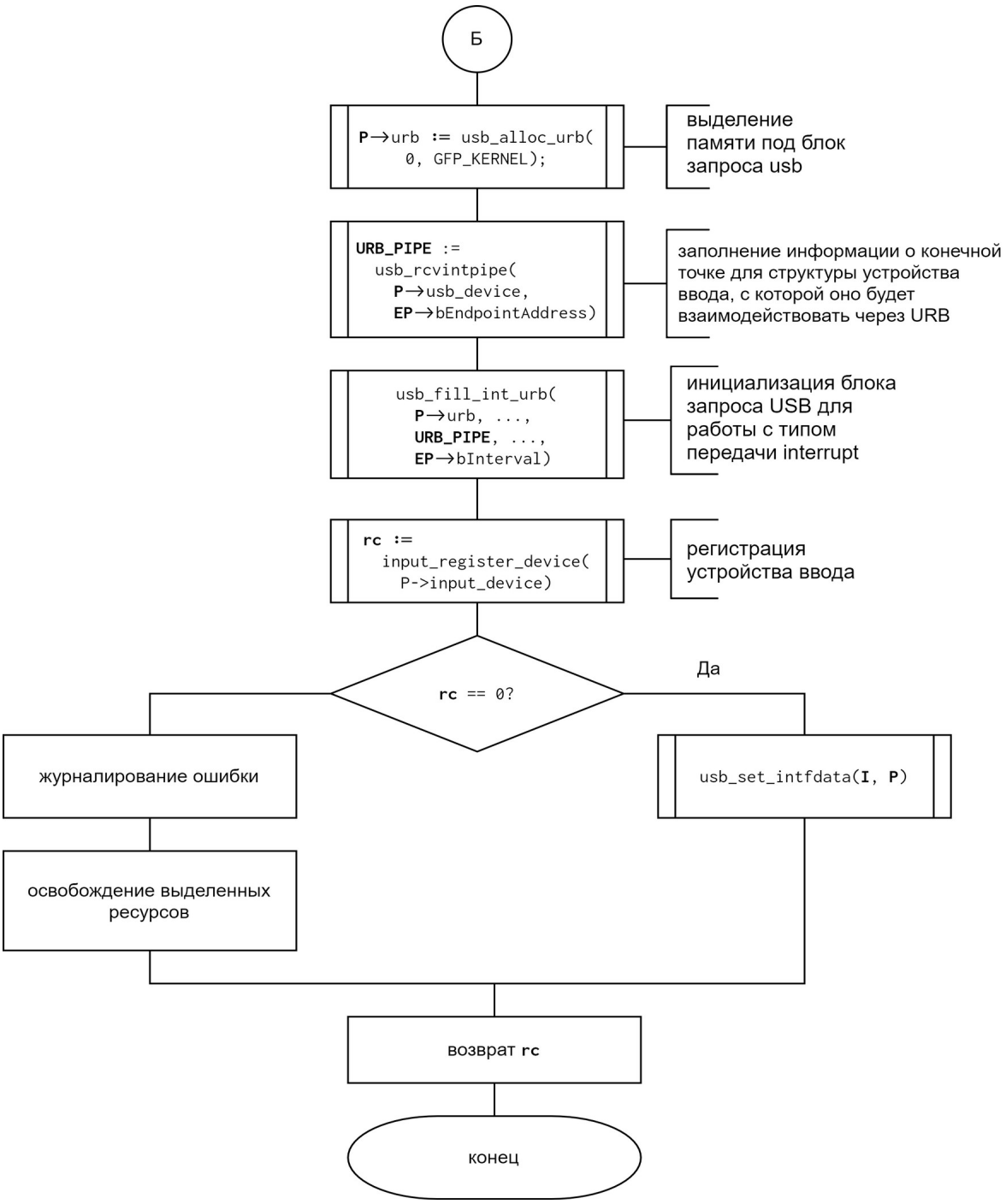

Рис. 8. схема алгоритма функции probe, часть 3

2.4 Отключение устройства

Схема функции обратного вызова `disconnect` приведена на рисунках 9,10.

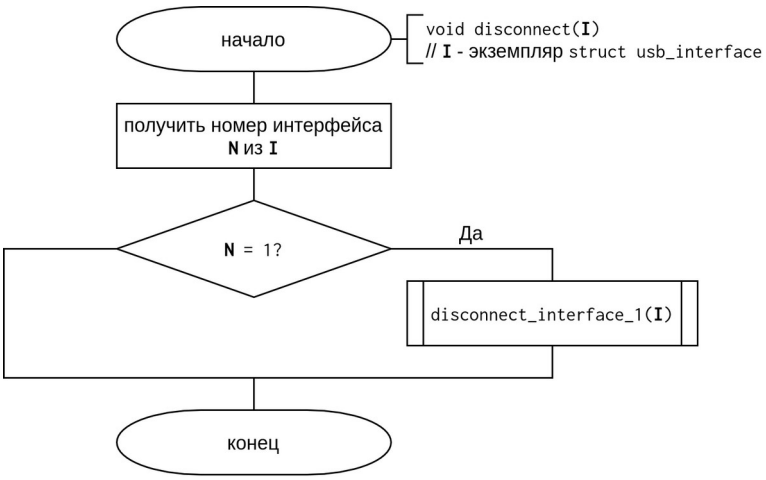

Рис. 9. схема алгоритма функции disconnect, часть 1

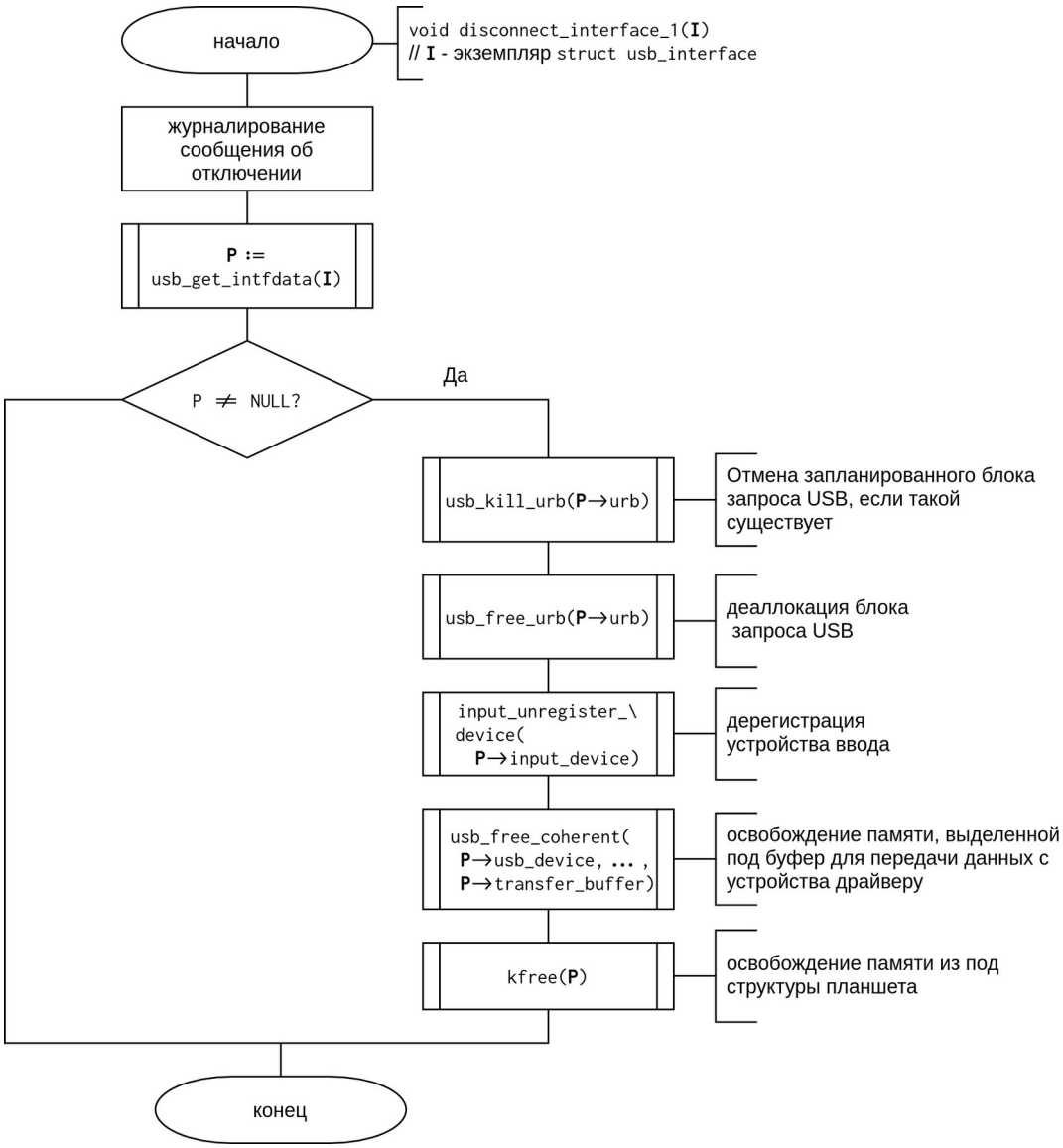

Рис. 10. схема алгоритма функции disconnect, часть 2

2.5 Обработка прерывания

2.5.1 Верхняя половина

На рисунке 11 изображена схема алгоритма функции верхней части прерывания.

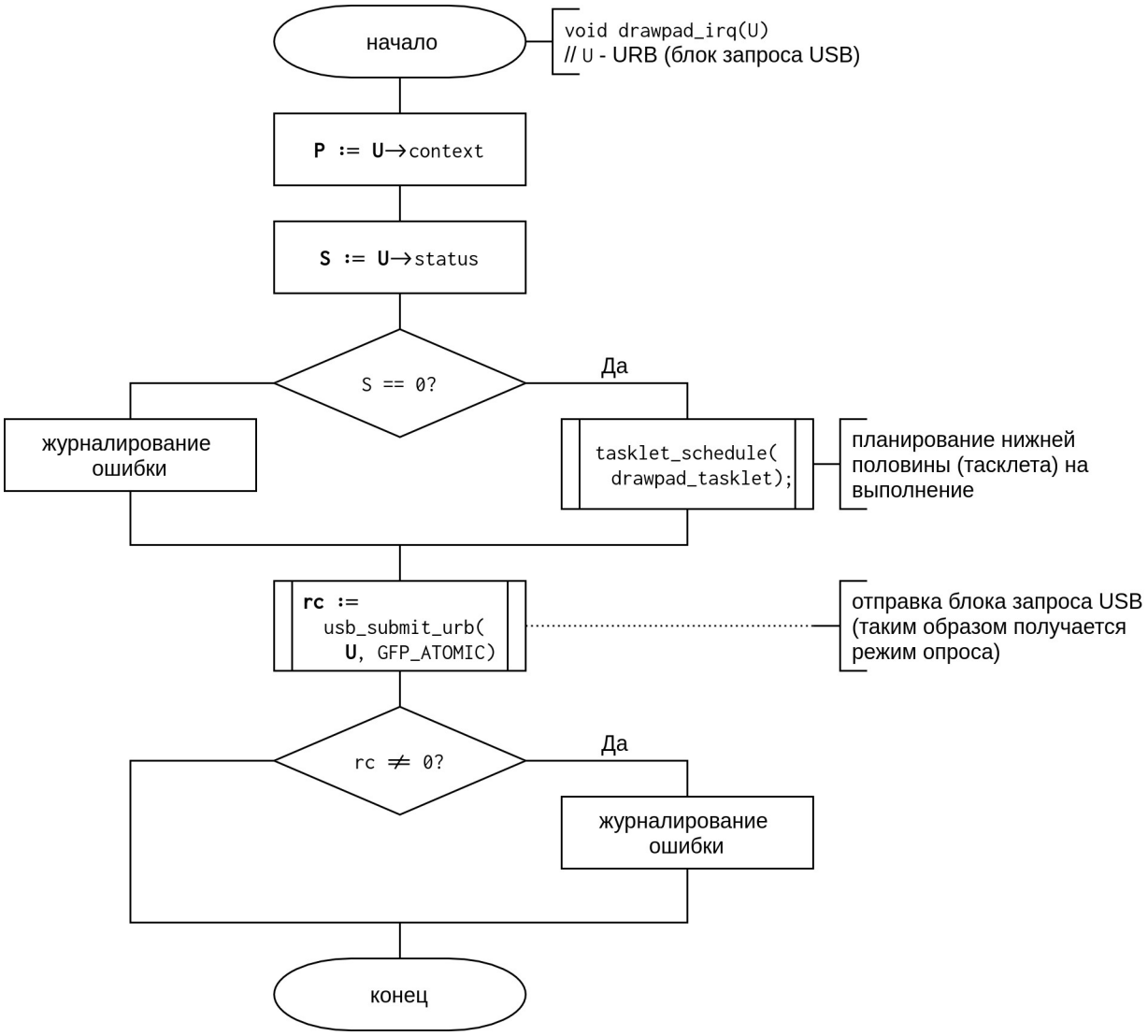

Рис. 11. верхняя половина обработчика прерывания

2.5.2 Нижняя половина

Схема алгоритма обработчика тасклета представлена на рисунке 12.

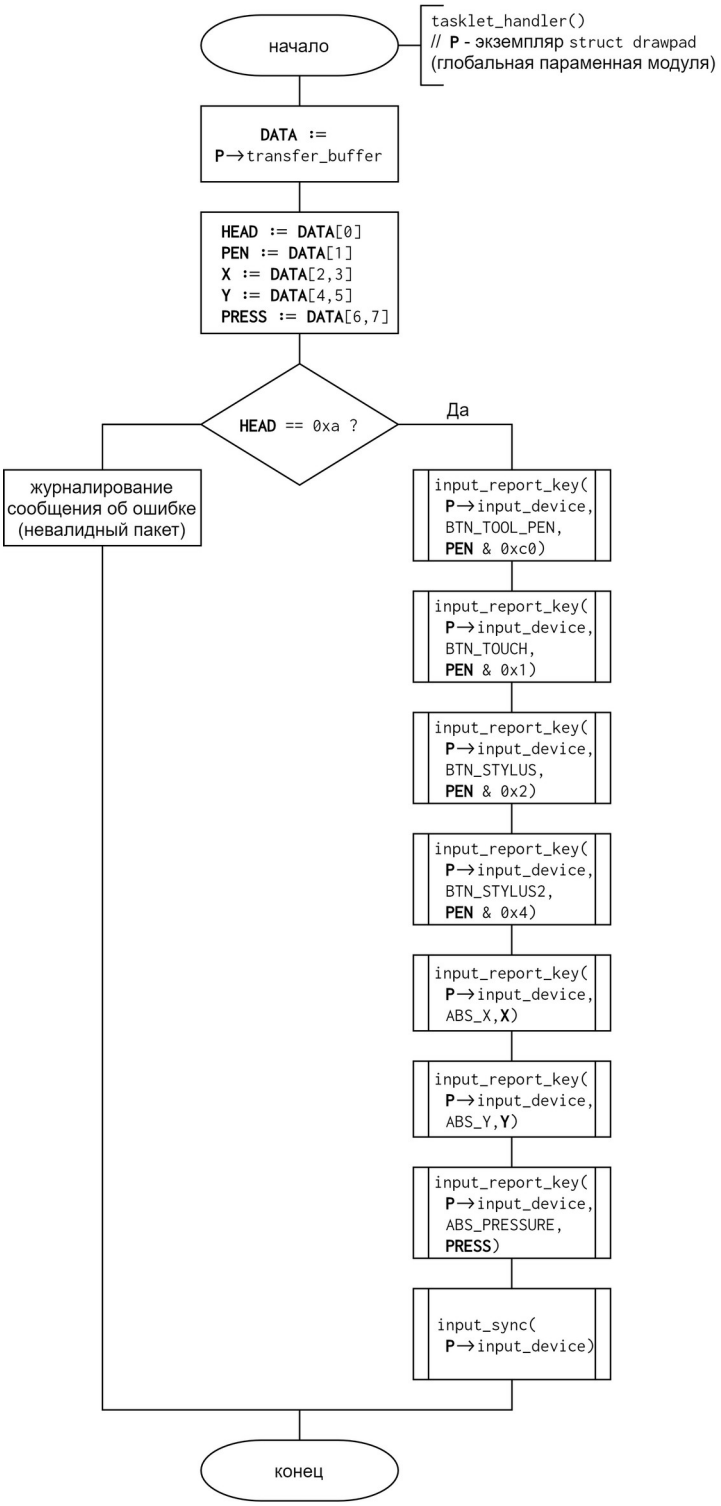

Рис. 12. схема алгоритма нижней половины прерывания

3. Технологическая часть

3.1 Выбор инструментов разработки

В качестве языка программирования выбран язык C, так как на нем реализована большая часть ядра Linux. Стандарт языка — C99.

В качестве комплилятора используется GNU Compiler Collection (GCC) — стандартный компилятор UNIX-подобных операционных систем.

В листинге 3.1 приведен код сценария сборки для утилиты Make.

Листинг 3.1 Makefile

```
1.  CONFIG_MODULE_SIG=n
2.  ccflags-y := -std=gnu99
3.  obj-m = drawpad_driver.o
4.
5.  all:
6.      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
7.  clean:
8.      make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

3.2 Демонстрация работы драйвера

```
sunz@MSI-PS63:~/Documents/os/HUION-H640P-Linux-Driver/src$ make
make -C /lib/modules/5.8.0-44-generic/build M=/home/sunz/Documents/os/HUION-H640P-Linux-Dr
iver/src modules
make[1]: Entering directory '/usr/src/linux-headers-5.8.0-44-generic'
  CC [M]  /home/sunz/Documents/os/HUION-H640P-Linux-Driver/src/drawpad_driver.o
  MODPOST /home/sunz/Documents/os/HUION-H640P-Linux-Driver/src/Module.symvers
  CC [M]  /home/sunz/Documents/os/HUION-H640P-Linux-Driver/src/drawpad_driver.mod.o
  LD [M]  /home/sunz/Documents/os/HUION-H640P-Linux-Driver/src/drawpad_driver.ko
make[1]: Leaving directory '/usr/src/linux-headers-5.8.0-44-generic'
sunz@MSI-PS63:~/Documents/os/HUION-H640P-Linux-Driver/src$ sudo insmod drawpad_driver.ko
sunz@MSI-PS63:~/Documents/os/HUION-H640P-Linux-Driver/src$ dmesg
[98645.397049] usbcore: registered new interface driver Huion H640P Driver
```

Рис. 13. компиляция и загрузка модуля

```
[99330.599332] usb 1-9.2: New USB device found, idVendor=256c, idProduct=006d, bcdDevice=
0.00
[99330.599343] usb 1-9.2: New USB device strings: Mfr=5, Product=6, SerialNumber=0
[99330.607131] [Huion H640P Interface 1]: probe device (256c:006D) Interface: 0
[99330.607144] Huion H640P Driver: probe of 1-9.2:1.0 failed with error -1
[99330.607450] [Huion H640P Interface 1]: probe device (256c:006D) Interface: 1
[99330.607472] [Huion H640P Interface 1]: pipe: 1078007936, endpoint: 0x82
[99330.607748] input: Huion H640P Pad as /devices/pci0000:00/0000:00:14.0/usb1/1-9/1-9.2/1
-9.2:1.1/input/input71
```

Рис. 14. журнал ядра после подключения устройства

На рисунке 15 приводится демонстрация журнала ядра во время взаимодействия с планшетом.

```
[Huion H640P Interface 1]: head: a pen: c1, x: 995, y: 683, press: 2702
[Huion H640P Interface 1]: head: a pen: c1, x: 972, y: 690, press: 2126
[Huion H640P Interface 1]: head: a pen: c1, x: 951, y: 697, press: 1551
[Huion H640P Interface 1]: head: a pen: c1, x: 938, y: 704, press: 1000
[Huion H640P Interface 1]: head: a pen: c1, x: 933, y: 711, press: 484
[Huion H640P Interface 1]: head: a pen: c1, x: 941, y: 716, press: 38
[Huion H640P Interface 1]: head: a pen: c0, x: 964, y: 719, press: 0
[Huion H640P Interface 1]: head: a pen: c0, x: 1006, y: 717, press: 0
[Huion H640P Interface 1]: head: a pen: c0, x: 1064, y: 710, press: 0
[Huion H640P Interface 1]: head: a pen: c0, x: 1064, y: 710, press: 0
[Huion H640P Interface 1]: head: a pen: 0, x: 1064, y: 710, press: 0
```

Рис. 15. логи ядра при взаимодействии с планшетом

После того как перо покидает рабочую область планшета, устройство передает значение `0` в байте `pen` и перестает транслировать какие-либо данные до тех пор пока перо не войдет в рабочую область снова.

Драйвер протестирован в графическом редакторе *Krita* для более точной оценки работы устройства (рисунок 16). На рисунке видно, что степень нажатия передается корректно, передача координат происходит достаточно плавно.

Рис. 16. тест драйвера в редакторе Krita

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Alessandro Rubini, Greg Kroah-Hartman, Jonathan Corbet. Linux Device Drivers, 3 издание. Глава 13.
2. The Linux Kernel, The Linux driver implemeter's API guide, Linux USB API. https://www.kernel.org/doc/html/latest/driver-api/usb/URB.html
3. Рязанова Н. Ю, Лекции 2020. Управление устройствами в Linux.
4. Brad Hards, Linux Journal, The Linux USB Input Subsystem, Part 1 https://www.linuxjournal.com/article/6396

ПРИЛОЖЕНИЕ

Листинг A1. Файл `log_utils.h`

```
1.  #ifndef __LOG_UTILS__
2.  #define __LOG_UTILS__
3.
4.  #include <linux/kernel.h>
5.
6.  #define LOG(x, ...) \
7.      "\033[0;35m" "[Huion H640P Interface 1]: " \
8.      "\x1B[0m" x __VA_OPT__(,) __VA_ARGS__
9.
10. #define LOG_INFO(x, ...) \
11.     printk(KERN_INFO LOG(x, __VA_ARGS__))
12.
13. #define LOG_WARN(x, ...) \
14.     printk(KERN_WARNING LOG(x, __VA_ARGS__))
15.
16. #define LOG_ERR(x, ...) \
17.     printk(KERN_ERR LOG(x, __VA_ARGS__))
18.
19. #endif // __LOG_UTILS__
```

Листинг A2. Файл `drawpad_properties.h`

```
1.  #ifndef __DRAWPAD_PROPERTIES__
2.  #define __DRAWPAD_PROPERTIES__
3.
4.  #include <linux/input-event-codes.h>
5.
6.  #define VENDOR_ID       0x256c    // Huion Animation Co.
7.  #define PRODUCT_ID      0x006d    // H640P Drawpad
8.
9.  #define MAX_PEN_PRESSURE            8191
10. #define MAX_PAD_RESOLUTION_VALUE    200
11. #define MAX_ABS_X                   32000
12. #define MAX_ABS_Y                   20000
13.
14. static const int input_event_types[] = {
```

```
15.     EV_ABS,
16.     EV_KEY,
17. };
18. 
19. static const int abs_events[] = {
20.     ABS_X,
21.     ABS_Y,
22.     ABS_PRESSURE,
23. };
24. 
25. static const int button_events[] = {
26.     BTN_TOOL_PEN,
27.     BTN_STYLUS,
28.     BTN_STYLUS2,
29.     BTN_TOUCH,
30. };
31. 
32. static const int drawpad_properties[] = {
33.     INPUT_PROP_DIRECT,
34. };
35. 
36. #endif // __DRAWPAD_PROPERTIES__
```

Листинг А3. Файл drawpad_driver_intf_1.c

```
1.  #ifndef __DRAWPAD_DRIVER__
2.  #define __DRAWPAD_DRIVER__
3.  
4.  #include <linux/module.h>
5.  #include <linux/init.h>
6.  #include <linux/usb/input.h>
7.  #include <linux/interrupt.h>
8.  
9.  #include "log_utils.h"
10. #include "drawpad_properties.h"
11. 
12. #define DRIVER_NAME     "Huion H640P Interface 1 Driver"
13. #define DRIVER_AUTHOR   "Rostislav V."
14. 
15. static int probe(struct usb_interface *interface,
16.                  const struct usb_device_id *dev_id);
17. static int probe_interface_1(struct usb_interface *interface,
```

```
18.                                 const struct usb_device_id *dev_id);
19.  static int open_interface_1(struct input_dev* input_device);
20.
21.  static void disconnect(struct usb_interface* interface);
22.  static void disconnect_interface_1(struct usb_interface* interface);
23.  static void close_interface_1(struct input_dev* input_device);
24.
25.
26.  static struct usb_device_id devices_table[] = {
27.      { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
28.      {}
29.  };
30.
31.  MODULE_DEVICE_TABLE(usb, devices_table);
32.
33.  static struct usb_driver drawpad_driver = {
34.      .name = DRIVER_NAME,
35.      .id_table = devices_table,
36.      .probe = probe,
37.      .disconnect = disconnect,
38.  };
39.
40.  static int __init drawpad_driver_init(void) {
41.      int rc = usb_register(&drawpad_driver);
42.      if (rc != 0) {
43.          printk(KERN_ERR "call usb_register: FAILED\n");
44.      }
45.
46.      return rc;
47.  }
48.
49.  static void __exit drawpad_driver_exit(void) {
50.      usb_deregister(&drawpad_driver);
51.  }
52.
53.
54.  struct drawpad {
55.      char    phys[32];
56.      struct usb_device   *usb_device;
57.      struct input_dev    *input_device;
58.      struct urb      *urb;
59.      unsigned char   *transfer_buffer;
60.      unsigned int    transfer_buffer_size;
```

```
61.     dma_addr_t      dma;
62. };
63. 
64. static struct drawpad *drawpad;
65. 
66. 
67. static void tasklet_handler(unsigned long tasklet_data) {
68.     uint8_t header;
69.     uint8_t pen_status;
70.     uint16_t x;
71.     uint16_t y;
72.     uint16_t pressure;
73. 
74.     unsigned char *data = drawpad->transfer_buffer;
75. 
76.     header = data[0];
77.     pen_status = data[1];
78.     memcpy(&x, &data[2], 2);
79.     memcpy(&y, &data[4], 2);
80.     memcpy(&pressure, &data[6], 2);
81. 
82.     if (header != 0xa) {
83.         LOG_ERR("Invalid packet recieved. Header = %x\n", header);
84.         return;
85.     }
86. 
87.     input_report_key(drawpad->input_device, BTN_TOOL_PEN, pen_status & 0xc0);
88.     input_report_key(drawpad->input_device, BTN_TOUCH, pen_status & 0x1);
89.     input_report_key(drawpad->input_device, BTN_STYLUS, pen_status & 0x2);
90.     input_report_key(drawpad->input_device, BTN_STYLUS2, pen_status & 0x4);
91. 
92.     input_report_abs(drawpad->input_device, ABS_X, x);
93.     input_report_abs(drawpad->input_device, ABS_Y, y);
94.     input_report_abs(drawpad->input_device, ABS_PRESSURE, pressure);
95. 
96.     input_sync(drawpad->input_device);
97. }
98. 
99. DECLARE_TASKLET(drawpad_tasklet, tasklet_handler, 0);
100. 
101. 
102. static void drawpad_irq(struct urb *urb) {
103.     struct drawpad *drawpad = urb->context;
```

```
104.
105.     if (urb->status == 0) {
106.         tasklet_schedule(&drawpad_tasklet);
107.
108.         int rc = usb_submit_urb(drawpad->urb, GFP_ATOMIC);
109.         if (rc) {
110.             LOG_ERR("\tfailed to submit urb\n");
111.         }
112.     } else {
113.         LOG_WARN("warning: urb status recieved: ");
114.
115.         switch (urb->status) {
116.             case -ENOENT:
117.                 LOG_ERR("\tENOENT (killed by usb_kill_urb)\n");
118.                 break;
119.             default:
120.                 LOG_ERR("\tanother error: %d\n", urb->status);
121.                 break;
122.         }
123.     }
124. }
125.
126.
127. static int probe(struct usb_interface *interface,
128.                  const struct usb_device_id *dev_id) {
129.
130.     int rc = -ENODEV;
131.
132.     struct usb_host_interface *interface_desc = interface->cur_altsetting;
133.     int interface_number = interface_desc->desc.bInterfaceNumber;
134.
135.     LOG_INFO("probe device (%04x:%04X) Interface: %d\n",
136.               dev_id->idVendor, dev_id->idProduct, interface_number);
137.
138.     if (interface_number == 1) {
139.         rc = probe_interface_1(interface, dev_id);
140.     }
141.
142.     return rc;
143. }
144.
145. static int probe_interface_1(struct usb_interface *interface,
146.                              const struct usb_device_id *dev_id) {
```

```
147.
148.     int rc = -ENOMEM;
149.
150.     struct usb_endpoint_descriptor *endpoint =
151.         &interface->cur_altsetting->endpoint[0].desc;
152.
153.     drawpad = kzalloc(sizeof(struct drawpad), GFP_KERNEL);
154.     if (!drawpad) {
155.         LOG_ERR("\tstruct drawpad allocation FAILURE\n");
156.         return rc;
157.     }
158.
159.     drawpad->usb_device = interface_to_usbdev(interface);
160.     drawpad->input_device = input_allocate_device();
161.     if (!drawpad->input_device) {
162.         LOG_ERR("\tinput_allocate_device FAILURE\n");
163.         kfree(drawpad);
164.         return rc;
165.     }
166.
167.     drawpad->transfer_buffer_size = endpoint->wMaxPacketSize;
168.     drawpad->transfer_buffer = usb_alloc_coherent(drawpad->usb_device,
169.                                                   drawpad->transfer_buffer_size,
170.                                                   GFP_KERNEL, &drawpad->dma);
171.     if (!drawpad->transfer_buffer) {
172.         LOG_ERR("\ttransfer buffer allocation FAILURE\n");
173.         input_free_device(drawpad->input_device);
174.         kfree(drawpad);
175.         return rc;
176.     }
177.
178.     usb_make_path(drawpad->usb_device, drawpad->phys, sizeof(drawpad->phys));
179.     strlcat(drawpad->phys, "/input1", sizeof(drawpad->phys));
180.
181.     drawpad->input_device->name = "Huion H640P Drawpad Interface 1";
182.     drawpad->input_device->phys = drawpad->phys;
183.     usb_to_input_id(drawpad->usb_device, &drawpad->input_device->id);
184.     drawpad->input_device->dev.parent = &interface->dev;
185.
186.     input_set_drvdata(drawpad->input_device, drawpad);
187.
188.     drawpad->input_device->open = open_interface_1;
189.     drawpad->input_device->close = close_interface_1;
```

```
190.     for (int i = 0; i < (sizeof(input_event_types) / sizeof(int)); i++) {
191.         set_bit(input_event_types[i], drawpad->input_device->evbit);
192.     }
193.
194.     for (int i = 0; i < (sizeof(abs_events) / sizeof(int)); i++) {
195.         set_bit(abs_events[i], drawpad->input_device->absbit);
196.     }
197.
198.     for (int i = 0; i < (sizeof(button_events) / sizeof(int)); i++) {
200.         set_bit(button_events[i], drawpad->input_device->keybit);
201.     }
202.
203.     for (int i = 0; i < (sizeof(drawpad_properties) / sizeof(int)); i++) {
204.         set_bit(drawpad_properties[i], drawpad->input_device->propbit);
205.     }
206.
207.
208.     input_set_abs_params(drawpad->input_device, ABS_X, 0, MAX_ABS_X, 0, 0);
209.     input_abs_set_res(drawpad->input_device, ABS_X, MAX_PAD_RESOLUTION_VALUE);
210.     input_set_abs_params(drawpad->input_device, ABS_Y, 0, MAX_ABS_Y, 0, 0);
211.     input_abs_set_res(drawpad->input_device, ABS_Y, MAX_PAD_RESOLUTION_VALUE);
212.     input_set_abs_params(drawpad->input_device, ABS_PRESSURE,
213.                          0, MAX_PEN_PRESSURE, 0, 0);
214.
215.     drawpad->urb = usb_alloc_urb(0, GFP_KERNEL);
216.     if (!drawpad->urb) {
217.         LOG_ERR("\tusb_alloc_urb FAILURE\n");
218.         input_free_device(drawpad->input_device);
219.         usb_free_coherent(drawpad->usb_device, drawpad->transfer_buffer_size,
220.                           drawpad->transfer_buffer, drawpad->dma);
221.         kfree(drawpad);
222.         return rc;
223.     }
224.
225.     drawpad->urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
226.     int pipe = usb_rcvintpipe(drawpad->usb_device, endpoint->bEndpointAddress);
227.     LOG_INFO("endpoint address: 0x%x\n", endpoint->bEndpointAddress);
228.
229.     usb_fill_int_urb(drawpad->urb, drawpad->usb_device, pipe,
230.                      drawpad->transfer_buffer, drawpad->transfer_buffer_size,
231.                      drawpad_irq, drawpad,
232.                      endpoint->bInterval);
233.     drawpad->urb->transfer_dma = drawpad->dma;
```

```
234.
235.     rc = input_register_device(drawpad->input_device);
236.     if (rc == 0) {
237.         usb_set_intfdata(interface, drawpad);
238.     } else {
239.         LOG_ERR("\tinput_register_device FAILURE\n");
240.         usb_free_urb(drawpad->urb);
241.         input_free_device(drawpad->input_device);
242.         usb_free_coherent(drawpad->usb_device, drawpad->transfer_buffer_size,
243.                           drawpad->transfer_buffer, drawpad->dma);
244.         kfree(drawpad);
245.     }
246.
247.     return rc;
248. }
249.
250. static void disconnect(struct usb_interface* interface) {
251.     struct usb_host_interface *interface_desc = interface->cur_altsetting;
252.     int interface_number = interface_desc->desc.bInterfaceNumber;
253.
254.     if (interface_number == 1) {
255.         disconnect_interface_1(interface);
256.     }
257. }
258.
259. static void disconnect_interface_1(struct usb_interface *interface) {
260.     LOG_INFO("disconnect device\n");
261.
262.     struct drawpad *drawpad = usb_get_intfdata(interface);
263.     if (drawpad) {
264.         usb_kill_urb(drawpad->urb);
265.         usb_free_urb(drawpad->urb);
266.         input_unregister_device(drawpad->input_device);
267.         usb_free_coherent(drawpad->usb_device, drawpad->transfer_buffer_size,
268.                           drawpad->transfer_buffer, drawpad->dma);
269.         kfree(drawpad);
270.     }
271. }
272.
273.
274. static int open_interface_1(struct input_dev* input_device) {
275.     struct drawpad *drawpad = input_get_drvdata(input_device);
276.     if (usb_submit_urb(drawpad->urb, GFP_KERNEL)) {
```

```
277.         return -EIO;
278.     }
279.
280.     return 0;
281. }
282.
283. static void close_interface_1(struct input_dev* input_device) {
284.     struct drawpad *drawpad = input_get_drvdata(input_device);
285.     usb_kill_urb(drawpad->urb);
286. }
287.
288. MODULE_LICENSE("GPL");
289. MODULE_AUTHOR(DRIVER_AUTHOR);
290.
291. module_init(drawpad_driver_init);
292. module_exit(drawpad_driver_exit);
293.
294.
295. #endif // __DRAWPAD_DRIVER__
```