**Разработка драйверов для USB-устройств под Linux (gcc driver linux usb)**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | [<< Предыдущая](https://www.opennet.ru/cgi-bin/opennet/jump.cgi?name=write_linux_driver.txt&ref=prev) | [ИНДЕКС](https://www.opennet.ru/cgi-bin/opennet/jump.cgi?name=write_linux_driver.txt&ref=index) | [Правка](https://www.opennet.ru/cgi-bin/opennet/jump.cgi?name=write_linux_driver.txt&ref=edit) | [src](https://www.opennet.ru/cgi-bin/opennet/jump.cgi?name=write_linux_driver.txt&ref=raw) / [Печать](https://www.opennet.ru/base/dev/write_linux_driver.txt.html) | [Следующая >>](https://www.opennet.ru/cgi-bin/opennet/jump.cgi?name=write_linux_driver.txt&ref=next) | |  | | | | | |

Ключевые слова: [gcc](https://www.opennet.ru/cgi-bin/opennet/ks.cgi?mask=gcc), [driver](https://www.opennet.ru/cgi-bin/opennet/ks.cgi?mask=driver), [linux](https://www.opennet.ru/cgi-bin/opennet/ks.cgi?mask=linux), [usb](https://www.opennet.ru/cgi-bin/opennet/ks.cgi?mask=usb), ([найти похожие документы](https://www.opennet.ru/cgi-bin/opennet/ks.cgi?mask=gcc%20driver%20linux%20usb))

From: Павел Курочкин <[http://b4open.spb.ru/>](http://b4open.spb.ru/%3E);

Date: Sun, 19 Nov 2006 17:02:14 +0000 (UTC)

Subject: Разработка драйверов для USB-устройств под Linux

Оригинал: <http://b4open.spb.ru/bin/view/B4/UsbDriversInLinuxKernelArticle>

**Предисловие**

-----------

Недавно мне довелось написать драйверы для специализированного

USB-сканера. Этот сканер работает в режиме непрерывной развертки с

частотой 10 кадров в секунду и обеспечивает скорость потока на чтение

3,2 МБайт/сек.

Драйверы сначала были разработаны под ядро Linux версии 2.6.15, а

затем, по требованию заказчика, адаптированы под ядро версии 2.4.26. К

моему удивлению, в обоих упомянутых версиях ядра написать драйвер не

составило особого труда. Для этого лишь необходимо четко представлять,

как работает USB и какими функциями обеспечивается взаимодействие

драйвера с USB-устройством. Все остальное - уже давно реализовано в

ядре linux.

При написании этой статьи я вижу перед собой следующие задачи:

1. рассказать об основных свойствах обмена устройств по USB

2. описать программный интерфейс взаимодействия драйвера с

USB-устройством

3. привести примеры использования функций ядра

Я надеюсь, что данная информация будет интересна программистам

системного уровня.

Введение в USB

Наверное, нет смысла говорить о таких очевидных вещах, как широкое

распространение USB, высокая скорость обмена по USB, возможность

горячего подключения устройств. Сейчас уже каждый пользователь ПК так

или иначе оценил преимущества и плюсы USB.

Поэтому сразу перейдем к менее очевидным вещам - к тому, как USB

устроен внутри.

Кто главный?

Одна из главных концепций USB заключается в том, что в USB-системе

может быть только один мастер. Им является host-компьютер. USB -

устройства всегда отвечают на запросы host-компьютера - они никогда не

могу посылать информацию самостоятельно.

Есть только одно исключение: после того, как хост перевел устройство в

suspend-режим, устройство может посылать запрос remote wakeup. Во всех

остальных случаях хост формирует запросы, а устройства отвечают на

них.

Направление

Хост всегда является мастером, а обмен данными должен осуществляться в

обоих направлениях:

\* OUT - отсылая пакет с флагом OUT, хост отсылает данные устройству

\* IN - отсылая пакет с флагом IN, хост отправляет запрос на прием

данных из устройства.

К этому нужно просто привыкнуть. Чтобы принять данные из устройства,

хост отсылат пакет с флагом IN. smile

Классификация пакетов в USB

По USB может передаваться несколько типов пакетов:

1. Token - запрос, содержит управляющую информацию: направление

операции (IN, OUT), номер endpoint

2. Data - пакет данных

3. Handshake - служебные пакеты, могут содержать подтверждение (ACK),

сообщение об ошибке, отказ (NACK)

4. Special - служебные пакеты, такие как PING

Более подробную информацию о пакетах, предусмотренных в спецификации

USB, можно прочитать в [2,3]. Рассмотрим несколько примеров

использования этих пакетов.

Пример: отсылка данных устройству

Чтобы отослать данные устройству, хост посылает пакет Token "OUT",

затем пакет Data. Если устройство готово обработать принятые данные,

оно отсылает пакет Handshake "ACK", подтверждающий транзакцию. Если

оно занято, оно отсылает отказ - Handshake "NACK". Если произошла

какая-то ошибка, то устройство может не отсылать Handshake.

Пример: отсылка данных хосту

Как уже говорилось, устройство самостоятельно никогда не отсылает

данные. Только по запросу. Чтобы принять данные, хост посылает пакет

Handshake "IN". Устройство по запросу может отослать пакет Data, а

затем Handshake "ACK". Либо может отослать Handshake "NACK", не

посылая Data.

Типы передачи данных

Спецификация USB определяет 4 типа потоков данных:

1. bulk transfer - предназначен для пакетной передачи данных с

размером пакетов 8, 16, 32, 64 для USB 1.1 и 512 для USB 2.0.

Используется алгоритм перепосылки (в случае возникновения ошибок),

а управление потоком осуществляется с использованием handshake

пакетов, поэтому данный тип является достоверным. Поддерживаются

оба направления - IN и OUT.

2. control transfer - предназначен для конфигурирования и управления

устройством. Также, как и в bulk, используются алгоритмы

подтверждения и перепосылки, поэтому этот тип обеспечивает

гарантированный обмен данными. Направления - IN (status) и

OUT(setup, control).

3. interrupt transfer - похож на bulk. Размер пакета - от 1 до 64

байт для USB 1.1 и до 1024 байт для USB 2.0. Этот тип гарантирует,

что устройство будет опрашиваться (то есть хост будет отсылать ему

token "IN") хостом с заданным интервалом. Направление - IN.

4. isochronous transfer - предназначен для передачи данных без

управления потоком (без подтверждений). Область применения -

аудио-потоки, видео-потоки. Размер пакета - до 1023 байт для USB

1.1 и до 1024 байт для USB 2.0. Предусмотрен контроль ошибок (на

приемной стороне) по CRC16. Направления - IN и OUT.

Endpoint - источник/приемник данных

Спецификация USB определеят endpoint (EP), как источник или приемник

данных. Устройство может иметь до 32 EP: 16 на прием и 16 на передачу.

Обращение к тому или иному endpoint'у происходит по его адресу.

Например, допустим, что хост хочет прочитать пакет данных из EP4 (4го

endpoint'a) устройства, пользуясь типом "bulk transfer". Он отсылает

пакет token "in", в котором указывает адрес endpoint'a.

Соответствующий источник в устройстве отсылает пакет данных хосту.

Аналогично происходит передача пакета данных.

Endpoint No.0

EP0 имеет особое значение для USB. Это Control EP. Он должен быть в

каждом USB-устройстве. Этот EP использует token "setup", чтобы

сигнализировать, что данные, отправляемые после него, предназначены

для управления устройством.

Используя этот EP0, хост может передавать setup-пакет длиной 8 байт и

данные, которые следуют за этим пакетом. Во многих случаях может

хватать передачи только setup-пакета. Однако устройство может

использовать и передачу данных по EP0, например для смены прошивок

компонентов устройства, или получения расширенной информации об

устройстве.

Рассмотрим немного подробнее setup-пакет.

EP0: setup-пакет

Содержимое setup-пакета представлено в таблице

Байт (No.) Имя Назначение

0 bmRequestType Поле для указания типа запроса, направления,

получателя

1 bRequest идентификатор запроса

2 wValueL 16-битное значение wValue, зависит от запроса.

3 wValueH

4 wIndexL 16-битное значение wIndex, зависит от запроса.

5 wIndexH

6 wLengthL количество байт, отсылаемых после setup-пакета.

7 wLengthH

Как видно из таблицы, setup-пакет содержит 5 полей. bmRequestType и

bRequest определяют запрос, а wValue, wIndex и wLength - его свойства.

Спецификация USB резервирует диапазон значений bRequest под

стандартные запросы. Каждое устройство обязано отвечать на все

стандартные запросы. В следующей таблице приведены только несколько

стандартных запросов, с которыми мы будем сталкиваться дальше.

bRequest Имя Описание

0x05 Set Address установка уникального адреса устройства в системе

0x06 Get Descriptor получение информации об устройстве. Тип информации

зависит от поля wValue.

Остальной диапазон устройство может использовать по своему усмотрению.

Более расширенную информацию можно получить в [2,3].

Распознавание устройства

Как происходит распознавание устройства, только что подключившегося к

системе? Уже упоминалось, что каждое устройство обязано обеспечить

доступ к EP0. Но кроме этого, оно еще должно отвечать на запросы,

указанные в спецификации USB для EP0. Пользуясь этими запросами и

происходит распознавание устройства в системе.

Алгоритм детектирования нового устройства следующий:

1. хост отсылает setup-пакет "Get Descriptor" (wValue = "device").

2. хост получает идентифицирующую информацию об устройстве

3. хост отсылает setup-пакет "Set address", после чего устройство

получает уникальный адрес в системе

4. хост отсылает остальные setup-пакеты "Get Descriptor" и получает

дополнительную информацию об устройстве: количество EP, требования

к питанию, и т.п.

Поддержка USB в ядре Linux

Программный интерфейс для взаимодействия с USB устройствами в ядре

Linux очень прост. За простым интерфейсом скрываются все алгоритмы

отсылки запросов, отслеживания подтверждений, контроля ошибок и т.п.

Все тонкости, описанные в предыдущей главе, уже реализованы в ядре

Linux.

В ядре файлы программ располагаются в drivers/usb/, а заголовочные

файлы - в include/linux/. Информации, представленной в этих

директориях, достаточно, чтобы самостоятельно написать драйвер для

любого USB-устройства.

Драйвер, взаимодействующий с USB-устройством(-ами), как правило,

выполняет следующие действия:

1. регистрация/выгрузка драйвера

2. регистрация/удаление устройства

3. обмен данными: управляющий и информационный.

Рассмотрим их по порядку более подробно, применительно к реализации

под ядро 2.6.15. Для большей ясности необходимо понимать основные

принципы UDM [4, 5], появившиеся в ядре 2.6.

Регистрация/выгрузка драйвера.

Регистрация USB-драйвера подразумевает:

1. заполнение структуры usb\_driver

2. регистрацию структуры в системе

Структура usb\_driver описана в include/linux/usb.h Рассмотрим наиболее

важные поля этой структуры.

struct usb\_driver {

// ...

const char \*name;

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

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

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

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

struct device\_driver driver;

// ...

};

Очевидно, что name - это имя драйвера. id\_table - это массив структур

usb\_device\_id. Этот список предназначен для определения cоответствия

подключаемого устройства определенным параметрам. Только те

устройства, которые соответствуют перечисленным параметрам, могут быть

подключены к драйверу. Если массив пуст, система будет пытаться

подключить каждое устройство к драйверу.

Поле driver говорит о том, что usb\_driver унаследован от device\_driver.

В самом простом случае каждый элемент id\_table[i] содержит пару

идентификаторов:

\* идентификатор производителя (Vendor ID)

\* идентификатор устройства (Device ID).

Определение структуры usb\_device\_id можно видеть в

include/linux/mod\_devicetable.h

probe и disconnect - это callback-функции, вызываемые системой при

подключении и отключении USB-устройства. probe будет вызыван для

каждого устройства, если список id\_table пуст, или только для тех

устройств, которые соответствуют параметрам, перечисленным в списке.

Рассмотрим пример.

#include <linux/usb.h>

#define MY\_DEV\_NAME "my\_usb\_device"

#define PRODUCT\_ID 0x1

#define VENDOR\_ID 0x1234

static struct usb\_device\_id my\_table [] = {

{ USB\_DEVICE(VENDOR\_ID, PRODUCT\_ID) },

{ } // терминирующий элемент списка

};

static struct usb\_driver my\_driver = {

.name = MY\_DEV\_NAME,

.probe = my\_probe,

.disconnect = my\_disconnect,

.id\_table = my\_table,

};

static int \_\_init my\_module\_init(void)

{

// регистрируем драйвер

return usb\_register(&my\_driver);

}

static void \_\_exit my\_module\_exit(void)

{

// выгружаем драйвер

usb\_deregister(&my\_driver);

}

В этом примере регистрируется драйвер USB-устройства, которое имеет

значения полей PRODUCT\_ID = 0x1, VENDOR\_ID = 0x1234. Только для

устройства с такими параметрами будет вызвана функция my\_probe.

Вызов my\_probe фактически означает регистрацию устройства в драйвере

my\_driver, а вызов my\_disconnect - удаление устройства. Поэтому

перейдем к следующему этапу - регистрация/удаление устройства.

Регистрация устройства

Один зарегистрированный драйвер может "подключать" несколько

устройств. Для подключения устройства к драйверу система вызывает

функцию драйвера probe, которой передает 2 параметра:

static int my\_probe(struct usb\_interface \*interface,

const struct usb\_device\_id \*id)

{

// ...

}

interface - это интерфейс USB-устройства. Обычно USB-драйвер

взаимодействует не с устройством напрямую, а с его интерфейсом. id -

содержит информацию об устройстве. Если функция возвращает 0, то

устройство успешно зарегистрировано, иначе - система попытается

"привязать" устройство к какому-нибудь другому драйверу.

Для отключения устройства от драйвера система вызывает функцию

disconnect, которой передается один параметр - интерфейс:

static void my\_disconnect(struct usb\_interface \*interface)

{

// ...

}

В общем случае, в функции probe для каждого подключаемого устройства

выделяется структура в памяти, заполняется, затем регистрируется,

например, символьное устройство, и проводится регистрация устройства в

sysfs. Также в функции probe может выполняться проверка на наличие у

устройства необходимых EP.

В качестве примера регистрации/удаления устройства лучше обратиться к

примеру из ядра linux, который находится в drivers/usb/usb-skeleton.c.

Использование USB Major

Остановимся более подробно на регистрации символьного устройства. Как

известно, для регистрации символьного устройства необходимо получить

число major - либо статически (обратиться к maintainer'у ядра и

занести его в include/linux/major.h), либо динамически (вызвать

register\_chrdev с параметром major = 0). Когда символьное устройство

зарегистрировано, необходимо создать файл в директории /dev. Для этого

можно воспользоваться либо командой mknod (из user-space) или

функциями devfs, если данное ядро поддерживает devfs.

В ядре версии 2.6 вышеописанная процедура сильно упрощена благодаря

появлению udev и sysfs. В системе работает программа-daemon udevd,

которая отслеживает появление файлов в sysfs (/sys/class/). На

основании информации, читаемой из этих файлов, она автоматически,

пользуясь правилами udev для данного устройства, создает необходимые

файлы в dev.

В программном интерфейсе USB для этих целей есть функция

usb\_register\_dev, которая выполняет все необходимое, чтобы udev

выполнил вышеописанные процедуры:

extern int usb\_register\_dev(struct usb\_interface \*intf,

struct usb\_class\_driver \*class\_driver);

extern void usb\_deregister\_dev(struct usb\_interface \*intf,

struct usb\_class\_driver \*class\_driver);

usb\_register\_dev принимает на вход interface и class\_driver. Структура

usb\_class\_driver выглядит следующим образом:

struct usb\_class\_driver {

char \*name;

struct file\_operations \*fops;

int minor\_base;

};

name - это имя устройства. Директория с этим именем появится в sysfs.

fops - файловые операции символьного устройства. minor\_base - базовый

minor номер.

Функция usb\_register\_dev выполняет следующие действия:

\* регистрирует символьное устройство с major номером 180 (см

include/linux/major.h) и резервирует диапазон из 16 minor номеров.

Поэтому minor\_base должен иметь младший полубайт = 0.

\* в зарезервированном диапазоне minor номеров выделяет один номер

для данного устройства. Этот номер записывает в interface->minor.

\* создает все необходимые файлы в sysfs: после этого udev создает

файлы в /dev/

Вызов usb\_deregister\_dev выполняет обратные процедуры, поэтому должен

вызываться в функции disconnect.

Обмен данными с устройством

Рассмотрим обмен данными для наиболее часто используемых типов:

control и bulk. Соответствующие функции определены в

drivers/usb/core/message.c.

control transfer

Для отсылки/приема данных в 0й EP используется функция

usb\_control\_msg:

extern int usb\_control\_msg(struct usb\_device \*dev, unsigned int pipe,

\_\_u8 request, \_\_u8 requesttype, \_\_u16 value, \_\_u16 index,

void \*data, \_\_u16 size, int timeout);

dev - указатель на usb\_device. Этот указатель может быть получен с

помощью вызова функции interface\_to\_usbdev(interface). pipe - EP pipe.

Этот параметр хранит в себе: тип передачи данных (bulk, control, ...),

направление, номер EP. Для задания pipe в include/linux/usb.h

определены макросы. Приведем только некоторые:

#define usb\_sndctrlpipe(dev,endpoint) \

((PIPE\_CONTROL << 30) | \_\_create\_pipe(dev,endpoint))

#define usb\_rcvctrlpipe(dev,endpoint) \

((PIPE\_CONTROL << 30) | \_\_create\_pipe(dev,endpoint) | USB\_DIR\_IN)

#define usb\_sndbulkpipe(dev,endpoint) \

((PIPE\_BULK << 30) | \_\_create\_pipe(dev,endpoint))

#define usb\_rcvbulkpipe(dev,endpoint) \

((PIPE\_BULK << 30) | \_\_create\_pipe(dev,endpoint) | USB\_DIR\_IN)

Параметры request, request\_type, value, index - это поля setup-пакета.

Их содержимое зависит от приложения. data, size - это массив для

отсылки/приема. См выше и [2,3].

timeout - параметр, задающий в tick'ах, сколько времени дается на

отсылку/прием.

Необходимо помнить, что функция usb\_control\_msg вернет значение только

после того, как setup-пакет и данные будут доставлены, или, если

произойдет ошибка или тайм-аут. Эта функция использует вызов

interruptible\_sleep\_on, поэтому эту функцию нельзя вызывать в

контексте прерывания.

Пример. Отсылка setup-пакета с заданными bRequest и wValue может

выглядеть следующим образом:

// ...

int ret;

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

ret = usb\_control\_msg(udev, usb\_sndctrlpipe(udev, 0),

bRequest, USB\_TYPE\_VENDOR,

wValue, 0, udev, 0, HZ);

// ...

В этом примере отсылается только setup-пакет. Поэтому data = udev,

size = 0.

**bulk transfer**

Для использования bulk transfer используется функция **usb\_bulk\_msg:**

extern int usb\_bulk\_msg(struct usb\_device \*dev, unsigned int pipe,

void \*data, int size, int \*actual\_size,

int timeout);

Параметры dev, pipe, data, size, timeout имеют тот же смысл, что и в

usb\_control\_msg. actual\_size - количество переданных байт в

действительности. actual\_size <= size.

usb\_bulk\_msg имеет то же свойство, что и usb\_control\_msg - ее нельзя

вызывать в контексте прерываний (см выше).

Пример. Прием 100 байт из 5го EP может выглядеть следующим образом:

// ...

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

u8 buf[[ ;]]

int ret, actual\_size;

ret = usb\_bulk\_msg(udev, usb\_rcvbulkpipe(udev, 5), buf, 100, &actual\_size, HZ)

;

// ...

Заключение

Спецификация USB определяет большое количество типов передачи данных,

пакетов, алгоритмов. Все алгоритмы USB реализованы в ядре linux, а

программисту драйверов предоставляется удобный и простой интерфейс в

виде набора функций, макросов, структур. В ядре linux 2.6 подсистема

USB вписывается в модель UDM [3,4], что делает ее очень гибкой для

взаимодействия с udev, sysfs.

Павел Курочкин, НТЦ Метротек