Санкт-Петербургский политехнический университет

Институт Компьютерных Наук и Технологий

Кафедра компьютерных систем и программных технологий

Лабораторная работа по дисциплине «Проектирование ОС и компонентов» на тему:

Разработка драйверов ОС Linux. Драйвер сетевой карты.

Выполнил: студент группы № 63501/3

Дедков С.В.

Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Душутина Е.В.

Санкт-Петербург

2016

# Оглавление

[Оглавление 2](#_Toc445775622)

[Цель работы 3](#_Toc445775623)

[Введение 4](#_Toc445775624)

[Разработка каркаса драйвера – загружаемый модуль 5](#_Toc445775625)

[Работа с устройствами. Символьный драйвер 9](#_Toc445775626)

[Драйвер сетевой карты 16](#_Toc445775627)

[Список литературы 16](#_Toc445775628)

# Цель работы

* Изучить общие принципы разработки драйверов ОС Linux. Написать тестовый модуль – каркас, для последующей разработки драйвера.
* Изучить принципы написания драйверов символьных устройств. Написать драйвер, для общего понимания работы драйверов в ОС Linux.
* Изучить принципы разработки драйверов для PCI устройств. Разработать драйвер сетевой карты.

# Введение

Драйвер - программное обеспечение, с помощью которого другое программное обеспечение (операционная система) получает доступ к аппаратному обеспечению некоторого устройства. (Драйвер, 2016)

Таким образом драйвер – программное обеспечение, которое реализует программный интерфейс аппаратуры, скрывая детали работы устройства. Таким образом, пользователю не приходиться вдаваться в детали аппаратных средств, достаточно выполнять стандартизированные вызовы, которые не зависят от специфики драйвера. Задача драйвера – перевести данные вызовы в специфичный для данного устройства операции.

В ОС Linux драйверы могут быть собраны отдельно от ядра и подгружены в процессе работы, когда это необходимо. Для этой цели используются LKM (Loadable Kernel Module) – загружаемые модули. (Jonathan Corbet, 2005)

В ОС Linux устройства разделяются на 3 типа:

* Символьные
* Блочные
* Сетевые

В данной работе рассматриваются драйвера для:

* символьного устройства – для понимания общих принципов разработки драйверов
* сетевого устройства – из задания к лабораторной работе. Используемая сетевая карта – TP-Link TF-3200. Версия ядра Linux – 2.6.38.8.

В ходе данной работы использовались Ubuntu 14.04.4 LTS и 11.04, которые основаны на Debian GNU/Linux. Соответственно использовался пакетный менеджер APT(Advanced Packaging Tool).

К примерам кода, приводимого в работе, указываются файлы листингов, а также версия ядра на котором модуль запускался.

Полные листинги находятся в репозитории по ссылке - <https://github.com/dsvgit/os_lab_syscalls>.

# Разработка каркаса драйвера – загружаемый модуль

Загружаемый модуль ядра (loadable kernel module, LKM) - в информатике - объектный файл, содержащий код, расширяющий возможности ядра операционной системы. Модули используются, чтобы добавить поддержку нового оборудования или файловых систем или для добавления новых системных вызовов. Когда функциональность, предоставляемая модулем, больше не требуется, он может быть выгружен, чтобы освободить память и другие ресурсы. (Загружаемый модуль ядра, 2016)

Модули ядра могут быть собраны и подгружены/выгружены во время работы ОС.

Реализация загрузки/выгрузки модуля ядра похожа на конструктор/деструктор объекта в терминах объектно-ориентированного программирования. Иными словами – модуль ядра должен реализовывать функцию загрузки и функцию выгрузки (что не всегда обязательно, например, для целей отладки). Функция регистрации модуля просто подготавливает работу модуля, для дальнейшей работы (регистрирует функции для последующего вызова, когда это будет необходимо).

Важно понимать, что код модуля ядра выполняется в пространстве ядра, привилегированном режиме работы процессора. Переход в этот режим осуществляется посредством системных вызовов (Jonathan Corbet, 2005).

Для работы с модулями потребуются заголовочные файлы текущей версии ядра, для их установки используется команда:

# sudo apt-get install linux-headers-$(uname -r)

Команда uname –r возвращает версию текущего ядра ОС.

При программировании в пространстве ядра не используются библиотеки, которые обычно подключаются при работе в пользовательском режиме. Таким образом программирование в режиме ядра предполагает ознакомление основными функциями и типами ядра ОС Linux.

Так же потребуются команды (Драйверы устройств в Linux. Часть 2, 2012):

* lsmod — список модулей, загруженных в текущий момент
* insmod <module\_file> — добавление / загрузка указанного файла модуля
* modprobe <module> — добавление / загрузка модуля вместе со всеми его зависимостями
* rmmod <module> — удаление / выгрузка модуля

Простейший модуль может содержать в себе функции регистрации и удаления модуля.

Ниже приводится пример простейшего модуля:

Листинг: *kernel\_module\_ex1/module\_ex1.c*

Версия ядра: *3.19.0-51*

1. #include <linux/init.h>
2. #include <linux/module.h>
4. MODULE\_LICENSE("GPL");
5. MODULE\_AUTHOR("Sergey Dedkov dsv.mail@yandex.ru");
6. MODULE\_DESCRIPTION("driver skeleton");
8. **static** **int** \_\_init md\_init(**void**) {
9. printk("+++ driver skeleton: md\_init()\n");
10. **return** 0;
11. }
13. **static** **void** \_\_exit md\_exit(**void**) {
14. printk("+++ driver skeleton: md\_exit()\n");
15. }
17. module\_init(md\_init);
18. module\_exit(md\_exit);

Для сборки используется утилита make и следующий Makefile

Листинг: *kernel\_module\_ex1/Makefile*

Версия ядра: *3.19.0-51*

1. ifeq ($(KERNELRELEASE),)
2. KERNELDIR ?= /lib/modules/$(shell uname -r)/build
3. PWD := $(shell pwd)
4. .PHONY: build clean
5. build:
6. $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
7. clean:
8. rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c
9. **else**
10. $(info Building with KERNELRELEASE = ${KERNELRELEASE})
11. obj-m := module\_ex1.o
12. endif

Рассмотренный модуль не делает ничего коме вывода сообщений при регистрации и удалении модуля в системный лог.

Ниже приводится лог сборки, установки, удаления модуля и системного вывода.

Сборка:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ make  
make -C /lib/modules/3.19.0-51-generic/build M=/home/dsv/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1 modules  
make[1]: Entering directory `/usr/src/linux-headers-3.19.0-51-generic'  
Building with KERNELRELEASE = 3.19.0-51-generic  
Building modules, stage 2.  
Building with KERNELRELEASE = 3.19.0-51-generic  
MODPOST 1 modules  
make[1]: Leaving directory `/usr/src/linux-headers-3.19.0-51-generic'

Результат сборки:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ ls  
Makefile module\_ex1.ko module\_ex1.mod.o modules.order  
module\_ex1.c module\_ex1.mod.c module\_ex1.o Module.symvers

Информация о модуле:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ modinfo module\_ex1.ko  
filename: /home/dsv/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1/module\_ex1.ko  
description: driver skeleton  
author: Sergey Dedkov dsv.mail@yandex.ru  
license: GPL  
srcversion: 9BAF37AF4A25F9CA7603D3B  
depends:   
vermagic: 3.19.0-51-generic SMP mod\_unload modversions

Установка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ sudo insmod module\_ex1.ko

Проверка, того, что модуль установлен:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ lsmod | grep module  
module\_ex1 16384 0

Удаление модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ sudo rmmod module\_ex1

Чтения лога:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ dmesg | grep +++  
[ 6313.243085] +++ driver skeleton: md\_init()  
[ 6328.301227] +++ driver skeleton: md\_exit()

Далее следует пояснить некоторые части исходного кода модуля.

Самая важная часть – макросы module\_init и module\_exit(*linux/init.h*).

В них передаются функции инициализации и удаления модуля.

Со следующими сигнатурами:

**static** **int** md\_init(**void**)

**static** **void** md\_exit(**void**)

Сигнатуры данных функций включают ключевое слово static, т.к. не экспортируемые и будут использоваться только один раз при загрузке/выгрузке.

Функция md\_init возвращает int.

* 0 – модуль установлен
* отрицательное значение – ошибка при установке модуля

В некоторых случаях полезно возвращать, например -1, тогда модуль не будет установлен, не придется его удалять, но код до возврата выполнится в пространстве ядра. Такой способ удобно использовать для отладки. В таких случаях нет необходимости использовать функцию удаления модуля.

Помимо этого, перед именем функций можно увидеть макросы \_\_init и \_\_exit – они используются для оптимизации(*linux/module.h*). Они актуальны только тогда, когда тот же самый код будет встроен в ядро. Все функции, отмеченные как \_\_init, автоматически помещаются компилятором *GCC* при компиляции в секцию инициализации *init*, а все функции, отмеченные как \_\_exit, помещаются в секцию выхода *exit* образа ядра. Предполагается, что все функции с отметкой \_\_init должны выполняться только один раз во время загрузки системы (и не должны выполняться снова до следующей загрузки системы). Так что, как только они будут выполнены во время загрузки системы, ядро удалит их и высвободит оперативную память (будет удалена вся секция *init*). (Драйверы устройств в Linux. Часть 3, 2012)

Макросы MODULE\_LICENSE, MODULE\_AUTHOR, MODULE\_DESCRIPTION(*linux/module.h*) используются для того, чтобы задать информацию о модуле. Выше в логах было показано как посмотреть информацию собранного модуля, с расширением *.ko*. с помощью команды modinfo.

Для ввода в журнал сообщений ядра используется функция printk. Данная функция похожа на привычную printf, за исключением того, что не может выводить числа с плавающей точкой и не предназначена для выдачи дампа своих данных в консоль.

Все вызовы команды printk помещают свои выходные данные в кольцевой буфер (журнал) ядра. Затем демон *syslog*, работающий в пользовательском пространстве, берет их для окончательной обработки и перенаправляет на различные устройства в соответствие с тем, что задано в файле конфигурации */etc/syslog.conf*.

При использовании printk есть возможность задавать уровни журналирования при помощи макросов(*linux/kernel.h*):

#define KERN\_EMERG "<0>" /\* system is unusable \*/

#define KERN\_ALERT "<1>" /\* action must be taken immediately \*/

#define KERN\_CRIT "<2>" /\* critical conditions \*/

#define KERN\_ERR "<3>" /\* error conditions \*/

#define KERN\_WARNING "<4>" /\* warning conditions \*/

#define KERN\_NOTICE "<5>" /\* normal but significant condition \*/

#define KERN\_INFO "<6>" /\* informational \*/

#define KERN\_DEBUG "<7>" /\* debug-level messages \*/

Теперь, в зависимости от этих уровней журналирования (т. е. первых трех символов в строке формата), демон пользовательского пространства syslog перенаправляет каждое сообщения в соответствие с заданной конфигурацией. Обычно местом, куда перенаправлются сообщения всех уровней журналирования, является журнальный файл */var/log/messages*. Таким образом, все данные, выдаваемые командой printk, по умолчанию находятся в этом файле. Можно изменить настройку - например, пересылать сообщения на последовательный порт (например, */dev/ttyS0*).

Сообщения теперь находятся в буфере */var/log/messages*, причем в нем находятся сообщения не только из ядра, но и от различных демонов, работающих в пользовательском пространстве. К тому же, этот файл обычно нельзя читать от имени обычного пользователя. Поэтому для непосредственного разбора сообщений, находящихся в кольцевом буфере ядра, предоставляется утилита пользовательского пространства dmesg, которая выводит дамп буфера в стандартный выходной поток.

# Работа с устройствами. Символьный драйвер

Как было сказано выше в ОС Linux устройства разделяются на типы: символьные, блочные, сетевые. Для понимания общих принципов написания драйверов этот раздел содержит информацию о написании символьного драйвера.

Для любого приложения пользовательского пространства, предназначенного для работы с байт-ориентированным устройством (в пространстве аппаратных средств), следует использовать соответствующий драйвер символьного устройства (в пространстве ядра). Использование символьных драйверов осуществляется через соответствующие файлы символьных устройств, которые прикомпонованы к виртуальной файловой системе (*VFS*). Это означает, что приложение выполняет обычные файловые операции с файлом символьного устройства. Эти операции будут перетранслированы виртуальной файловой системой *VFS* в соответствующие функции в прикомпонованном драйвере символьного устройства. Затем для того, чтобы получить нужные результаты, с помощью этих функций осуществляется окончательный низкоуровневый доступ к реальному устройству (Драйверы устройств в Linux. Часть 4, 2012).

В этом полном подключении из приложения к устройству участвуют следующие четыре основных компонента:

* Приложение
* Файл символьного устройства
* Драйвер символьного устройства
* Символьное устройство

Приложение подключается к файлу устройства при помощи системного вызова *open*, открывающего файл устройства. Файлы устройств подключаются к драйверу устройства с помощью специального механизма регистрации, что осуществляется драйвером. Драйвер связывается с устройством с помощью специальных низкоуровневых операций, характерных для конкретного устройства. Таким образом, формируется полное соединение. При этом, файл символьного устройства не является реальным устройством, это просто специальная методика (*place-holder*) подключения реального устройства.

Для подключения приложения к файлу устройства используется имя файла устройства. Для подключения файла устройства к драйверу устройства используется номер файла устройства, а не имя файла. Приложение пользовательского пространства может использовать для файла устройства любое имя, а в пространстве ядра для связи между файлом устройства и драйвером устройства можно использовать тривиальный механизм индексации. Таким номером файла обычно является пара *<major, minor>* (старший и младший номера файла устройства). (Драйверы устройств в Linux. Часть 4, 2012)

* *major* – номер драйвера устройства (еще можно ассоциировать с типом устройства)
* *minor* – файл устройства.

Файлы всех устройств можно найти в директории /dev (файловая система dev). Так, например, для того, чтобы вывести весь список символьных устройств можно воспользоваться утилитой ls.

Ниже приводится лог вывода первых 10 символьных устройств.

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ ls -l /dev | grep "^c" | head  
crw-rw--— 1 root video 10, 175 Mar 14 17:26 agpgart  
crw-----— 1 root root 10, 235 Mar 14 17:26 autofs  
crw-----— 1 root root 10, 234 Mar 14 17:26 btrfs-control  
crw-----— 1 root root 5, 1 Mar 14 17:26 console  
crw-----— 1 root root 10, 60 Mar 14 17:26 cpu\_dma\_latency  
crw-----— 1 root root 10, 203 Mar 14 17:26 cuse  
crw-rw----+ 1 root audio 14, 9 Mar 14 17:26 dmmidi  
crw-----— 1 root root 10, 61 Mar 14 17:26 ecryptfs  
crw-rw--— 1 root video 29, 0 Mar 14 17:26 fb0  
crw-rw-rw- 1 root root 1, 7 Mar 14 17:26 full

В каждой строчке между группой и датой последней модификации модно увидеть два номера через запятую. Первый – *major* номер устройства – указывает на то, каким драйвером обслуживается устройство. Второй – *minor* номер файла устройства.

Для отображения драйверов устройств пользуются файловой системой *proc*. В файле */proc/devices* можно найти все драйвера, которые используются в текущей системе, разделенные на символьные и блочные. Для вывода можно воспользоваться утилитой cat. Ниже приводится частичный лог вывода файла */proc/devices*.

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/kernel\_module\_ex1$ cat /proc/devices  
Character devices:  
1 mem  
4 /dev/vc/0  
4 tty  
4 ttyS

…  
  
Block devices:  
1 ramdisk  
2 fd  
259 blkext  
7 loop  
8 sd

По этим данным можно сопоставить файл устройства с драйвером, который его обслуживает.

Для понимания, того как регистрируются драйвера символьных устройств, рассмотрим пример модуля драйвера символьного устройства, зарегистрируем драйвер и создадим файл устройства.

Листинг: *chr\_device\_driver\_ex2/chr\_driver\_ex2.c*

Версия ядра: *3.19.0-51*

1. #include <linux/module.h>
2. #include <linux/version.h>
3. #include <linux/kernel.h>
4. #include <linux/types.h>
5. #include <linux/kdev\_t.h>
6. #include <linux/fs.h>
7. #include <linux/device.h>
8. #include <linux/cdev.h>
9. #include <linux/uaccess.h>
11. MODULE\_LICENSE("GPL");
12. MODULE\_AUTHOR("Sergey Dedkov");
13. MODULE\_DESCRIPTION("Test Character Driver");
15. **static** dev\_t first;
16. **static** **struct** cdev c\_dev;
17. **static** **struct** **class** \*cl;
19. **static** **char**   message[256] = {0};
20. **static** **short**  size\_of\_message;
22. **static** **int**      dev\_open(**struct** inode \*, **struct** file \*);
23. **static** **int**      dev\_release(**struct** inode \*, **struct** file \*);
24. **static** ssize\_t  dev\_read(**struct** file \*, **char** \*, **size\_t**, loff\_t \*);
25. **static** ssize\_t  dev\_write(**struct** file \*, **const** **char** \*, **size\_t**, loff\_t \*);
27. **static** **struct** file\_operations fops =
28. {
29. .owner = THIS\_MODULE,
30. .open = dev\_open,
31. .release = dev\_release,
32. .read = dev\_read,
33. .write = dev\_write
34. };
36. **static** **int** \_\_init chr\_init(**void**) /\* Constructor \*/
37. {
38. **if** (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)
39. {
40. **return** -1;
41. }
43. **if** ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)
44. {
45. unregister\_chrdev\_region(first, 1);
46. **return** -1;
47. }
49. **if** (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)
50. {
51. class\_destroy(cl);
52. unregister\_chrdev\_region(first, 1);
53. **return** -1;
54. }
56. cdev\_init(&c\_dev, &fops);
58. **if** (cdev\_add(&c\_dev, first, 1) == -1)
59. {
60. device\_destroy(cl, first);
61. class\_destroy(cl);
62. unregister\_chrdev\_region(first, 1);
63. **return** -1;
64. }
66. **return** 0;
67. }
69. **static** **void** \_\_exit chr\_exit(**void**) /\* Destructor \*/
70. {
71. cdev\_del(&c\_dev);
72. device\_destroy(cl, first);
73. class\_destroy(cl);
74. unregister\_chrdev\_region(first, 1);
75. }
77. **static** **int** dev\_open(**struct** inode \*i, **struct** file \*f)
78. {
79. printk(KERN\_INFO "Character driver: open()\n");
80. **return** 0;
81. }
83. **static** **int** dev\_release(**struct** inode \*i, **struct** file \*f)
84. {
85. printk(KERN\_INFO "Character driver: close()\n");
86. **return** 0;
87. }
89. **static** ssize\_t dev\_read(**struct** file \*f, **char** \_\_user \*buffer, **size\_t** len, loff\_t \*offset)
90. {
91. **int** error\_count = 0;
92. **int** size = size\_of\_message;
94. error\_count = copy\_to\_user(buffer, message, size\_of\_message);
96. **if** (error\_count != 0)
97. {
98. printk(KERN\_INFO "Failed to send %d characters to the user\n", error\_count);
99. **return** -EFAULT;
100. }
102. printk(KERN\_INFO "Sent %d characters to the user, buffer: %s, message: %s \n", size\_of\_message, buffer, message);
103. size\_of\_message = 0;
104. **return** size;
105. }
107. **static** ssize\_t dev\_write(**struct** file \*f, **const** **char** \_\_user \*buffer, **size\_t** len, loff\_t \*offset)
108. {
109. sprintf(message, "%s(%d letters)\n", buffer, (**int**)len);   // appending received string with its length
110. size\_of\_message = strlen(message);                 // store the length of the stored message
111. printk(KERN\_INFO "Received %d characters from the user, message: %s \n", (**int**)len, message);
112. **return** len;
113. }
115. module\_init(chr\_init);
116. module\_exit(chr\_exit);

Сборка осуществляется аналогично предыдущему примеру. Makefile используется такой же.

Результат работы:

Сборка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ make  
make -C /lib/modules/3.19.0-51-generic/build M=/home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2 modules  
make[1]: Entering directory `/usr/src/linux-headers-3.19.0-51-generic'  
Building with KERNELRELEASE = 3.19.0-51-generic  
CC [M] /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.o  
Building modules, stage 2.  
Building with KERNELRELEASE = 3.19.0-51-generic  
MODPOST 1 modules  
CC /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.mod.o  
LD [M] /home/dsv/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2/chr\_driver\_ex2.ko  
make[1]: Leaving directory `/usr/src/linux-headers-3.19.0-51-generic'

Установка модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo insmod chr\_driver\_ex2.ko

Проверка установки модуля:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ lsmod | grep chr  
chr\_driver\_ex2 16384 0

Проверка добавления класса устройств:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ ls /sys/class/char\_test\_drv/  
chrnull

Проверка добавления драйвера. Частичный лог:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ cat /proc/devices  
Character devices:

…  
226 drm  
250 chr\_test\_driver  
251 hidraw  
…  
Block devices:  
1 ramdisk  
2 fd

…

Изменение прав доступа к файлу устройства, чтобы моно было с им работать:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo chmod 777 /dev/chrnull

Вывод в файл устройства сообщения:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo echo "test message" > /dev/chrnull

Чтение из файла устройства сообщения. Вывод в консоль:

dsv@ubuntu:~/projects/scalls/os\_lab\_syscalls/driver/chr\_device\_driver\_ex2$ sudo cat /dev/chrnull   
test message  
(13 letters)

Далее представлен разбор кода модуля символьного драйвера. Опущены части кода, объяснение которых происходило в предыдущем разделе.

Драйвер символьного устройства для работы должен знать *major* и *minor* номера устройств. Для этих целей используется структура dev\_t(*linux/types.h*).

1. **static** dev\_t first;

Для работы с данной структурой используются следующие макросы(*linux/kdev\_t.h*):

* *MAJOR(dev\_t dev)* - из *dev* извлекается старший номер
* *MINOR(dev\_t dev)* - из *dev* извлекается младший номер
* *MKDEV(int major, int minor)* - из старшего и младшего номеров создается *dev*

Подключение файла устройства к драйверу выполняется за 2 шага:

* Выполняется регистрация файлов устройств для номеров *major, minor*
* Подключение операций, выполняемых над файлом устройства, к функциям драйвера устройства.

Первый шаг в модуле выполняет следующий код:

1. **if** (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)
2. {
3. **return** -1;
4. }

С помощью функции alloc\_chrdev\_region (*linux/fs.h*) динамически определяется свободный старший номер и регистрируется число *count* среди номеров файлов устройств, начинающиеся с *baseminor*, с заданным именем файла *name.*

1. **int** alloc\_chrdev\_region(dev\_t \*dev, unsigned baseminor, unsigned count, **const** **char** \*name);

Первый параметр является выходным и заполняется свободными страшим и младшим номерами.

После этого регистрируется класс устройства (class\_create) и файл устройства (chrdev\_region). В функцию создания файла устройства третьим параметром передается структура dev\_t, ранее заполненная.

1. **if** ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)
2. {
3. unregister\_chrdev\_region(first, 1);
4. **return** -1;
5. }
7. **if** (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)
8. {
9. class\_destroy(cl);
10. unregister\_chrdev\_region(first, 1);
11. **return** -1;
12. }

Затем выполняется второй шаг. Устройство инициализируется операциями. Для этого используется функция cdev\_init (*linux/cdev.h*), в которую передается ссылка на структуру описывающее символьное устройство cdev (*linux/cdev.h*) и ссылка на структуру file\_operations (*linux/fs.h*). В последней реализован интерфейс устройства, а именно операции *open, release, write, read.* Код этих операций не рассматривается, так как прозрачен для понимания.

1. **static** **struct** file\_operations fops =
2. {
3. .owner = THIS\_MODULE,
4. .open = dev\_open,
5. .release = dev\_release,
6. .read = dev\_read,
7. .write = dev\_write
8. };

# Драйвер сетевой карты

В данном разделе рассматривается написание простейшего драйвера для сетевой карты TP-Link TF-3200. При разработке использована Ubuntu 11.04, ядро Linux - 2.6.38.8.

Данная карта подключается к компьютеру посредством PCI.

*PCI* (*Peripheral component interconnect*) — шина ввода-вывода для подключения периферийных устройств к материнской плате компьютера. (PCI, 2015)

Все PCI устройства содержат по крайней мере 256 байт адресного пространства. Первые 64 байт стандартизованы, а остальные зависят от устройства. Рисунок ниже показывает схему не зависящего от устройства конфигурационного пространства (Jonathan Corbet, 2005).

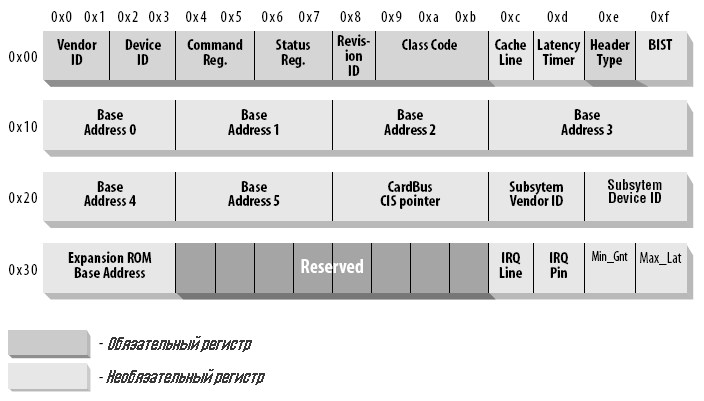


Рисунок 1. Стандартизированные конфигурационные регистры PCI (Jonathan Corbet, 2005)

Устройство идентифицируют три или пять регистров *PCI*: *vendorID, deviceID и class* являются теми тремя, которые используются всегда. Каждый производитель PCI присваивает собственные значения этим регистрам, предназначенным только для чтения, и драйвер может использовать их для поиска устройства. Кроме того, с целью дальнейшего различия похожих устройств поставщик иногда устанавливает поля *subsystem vendorID и subsystem deviceID*.

* vendorID. Этот 16-ти разрядный регистр идентифицирует изготовителя оборудования. Например, каждое устройство Intel отмаркировано одним и тем же числом поставщика, 0x8086. Существует глобальный реестр таких чисел, который ведёт PCI Special Interest Group, и производители должны обратиться туда, чтобы получить уникальный номер.
* deviceID. Это другой 16-ти разрядный регистр, выбранный производителем; для ID устройства не требуется никакой официальной регистрации. Этот ID, как правило, используется в паре с ID поставщика, создавая уникальный 32-х разрядный идентификатор аппаратного устройства. Мы используем слово сигнатура для обращения к ID поставщика и ID устройства в паре. Драйвер устройства обычно полагается на сигнатуру, чтобы определить своё устройство; вы можете найти, какое значение искать, в руководстве оборудования для целевого устройства.
* class. Каждое периферийное устройство относится к class (классу). Регистр class является 24-х разрядным значением, чьи старшие 8 бит идентифицируют "базовый класс" (или группу). Например, "ethernet" и "token ring" являются двумя классами, принадлежащими к группе "network", а классы "serial" и "parallel" относятся к группе "communication". Некоторые драйверы могут поддерживать несколько аналогичных устройств, каждое из них имеет свою сигнатуру, однако все они принадлежат к одному классу; эти драйверы могут рассчитывать на регистр class для определения своих периферийных устройства, как показано ниже.

Используя эти разные идентификаторы, драйвер PCI может сказать ядру, какие виды устройств он поддерживает. Чтобы определить список различных типов устройств PCI, которые поддерживает драйвер, используется структура struct pci\_device\_id.

Структура pci\_device\_id должна быть экспортирована в пользовательское пространство, чтобы позволить системам горячего подключения и загрузки модулей узнать, с какими устройствами работает модуль. Эту задачу решает макрос MODULE\_DEVICE\_TABLE. Пример:

MODULE\_DEVICE\_TABLE(pci, i810\_ids);

Этот оператор создаёт локальную переменную, называемую \_\_mod\_pci\_device\_table, которая указывает на список struct pci\_device\_id. Позже, в процессе сборки ядра, программа depmod просматривает все модули на символ \_\_mod\_pci\_device\_table. Если этот символ найден, она вынимает данные из модуля и добавляет их в файл /lib/modules/KERNEL\_VERSION/modules.pcimap. После завершения работы depmod, все PCI устройства, поддерживаемые модулем в ядре, вместе с именами их модулей, перечисляются этом в файле. Когда ядро сообщает системе горячего подключения, что было найдено новое устройство PCI, система горячего подключения использует файл modules.pcimap, чтобы найти для загрузки правильный драйвер.

Основной структурой, которую должны создать все драйверы PCI для того, чтобы быть правильно зарегистрированными в ядре, является структура struct pci\_driver. Эта структура состоит из ряда функций обратного вызова и переменных, описывающих драйвер PCI для ядра PCI.

Таким образом, чтобы создать правильную структуру struct pci\_driver должны быть проинициализированы только четыре поля :

static struct pci\_driver pci\_driver = {

.name = "pci\_skel",

.id\_table = ids,

.probe = probe,

.remove = remove,

};

Чтобы зарегистрировать struct pci\_driver в ядре PCI, выполняется вызов pci\_register\_driver с указателем на struct pci\_driver. Это традиционно делается в коде инициализации модуля PCI драйвера:

static int \_\_init pci\_skel\_init(void)

{

return pci\_register\_driver(&pci\_driver);

}

# Список литературы

Jonathan Corbet, A. R.-H. (2005). *Linux Device Drivers, Third Edition.* Sebastopol: O'Reilly.

*PCI*. (16 ноябрь 2015 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/PCI

*Драйвер*. (17 02 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Драйвер

*Драйверы устройств в Linux. Часть 2*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-02.html

*Драйверы устройств в Linux. Часть 3*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-03.html

*Драйверы устройств в Linux. Часть 4*. (июнь 2012 г.). Получено из Linux по-русски. Виртуальная энциклопедия: http://rus-linux.net/MyLDP/BOOKS/drivers/linux-device-drivers-04.html

*Загружаемый модуль ядра*. (11 02 2016 г.). Получено из wikipedia: https://ru.wikipedia.org/wiki/Загружаемый\_модуль\_ядра

Лав, Р. (2014). *Linux Системное программирование.* СПб: Питер.