# Содержание

[Содержание 1](#_Toc484007252)

[Теоретическая часть 2](#_Toc484007253)

[Загружаемый модуль ядра 3](#_Toc484007254)

[Драйвер символьного устройства 4](#_Toc484007255)

[Практическая часть 6](#_Toc484007256)

[Загружаемый модуль 6](#_Toc484007257)

[Драйвер символьного устройства 10](#_Toc484007258)

[Источники 19](#_Toc484007259)

# Теоретическая часть

В операционной системе ОС Linux фактическая архитектура ввода/вывода скрыта от прикладного процесса несколькими интерфейсами. Первой обработку пользовательских запросов принимают на себя интерфейсы высокого уровня: файловая система, интерфейс сокетов и другие.

Однако возможны ситуации, когда прикладному процессу требуется взаимодействие с периферийными устройствами на более низком уровне. Хотя в этом случае роль файловой подсистемы не столь велика, как при работе с обычными файлами, всё равно ядро ОС Linux предоставляет процессу унифицированную схему, скрывающую истинную архитектуру того или иного устройства.

В конечном итоге работа всех этих интерфейсов, как высокого уровня, так и низкого (взаимодействие с физическим устройством), обеспечивается подсистемой ввода/вывода ядра операционной системы, основным компонентом которой являются драйверы – модули ядра, обеспечивающие непосредственную работу с периферийными устройствами.

Характеристики периферийных устройств могут сильно различаться, но как правило, любое устройство можно отнести к одному из трёх классов:

**Символьные устройства**

Доступ к символьным устройствам похож на доступ к файлам, а задача драйвера символьного устройства – обеспечивать этот доступ. Символьные драйвера обычно способны обрабатывать такие системные вызовы как open, close, read, write. Примером устройств такого типа могут служить консоль и параллельные порты, поскольку с ними хорошо вести потоковый обмен информацией. Доступ к символьным устройствам может быть получен через файловую систему, например, через /dev/tty1 или /dev/lp1.

Единственное отличие символьных устройств от обычных файлов - возможность всегда вернуться к уже просмотренной информации и пройтись вперёд, тогда как большинство символьных устройств функционируют как каналы данных, к которым вы можете обращаться только последовательно. Тем не менее существуют устройства, выглядящие как область данных, по которой можно преспокойно продвигаться и вперёд, и назад.

**Блочные устройства**

Блочные устройства - это нечто, на чём может содержатся файловая система, например, жёсткий диск. В большинстве Unix-подобных систем доступ к блочным устройствам осуществляется посредством блоков. ОС Linux позволяет вам читать и писать на блочные устройства так же, как и на символьные – произвольным количеством байтов. В результате, блочные и символьные устройства отличаются лишь в способе организации хранения данных. Доступ к блочным устройствам (так же, как и к символьным) можно получить из файловой системы.

**Сетевые устройства**

Не являясь потоково-ориентированными, сетевые устройства не так просто отобразить на файловую систему, как например /dev/tty1. В ОС Linux сетевым устройствам присваивается лишь уникальное имя (например, eth0), поскольку в файловой системе для них нет соответствующего файла. Взаимодействие между ядром и сетевым драйвером в корне отличается от взаимодействия с драйвером блочного или символьного устройства – вместо системных вызовов read и write ядро вызывает функции, относящиеся к передаче пакетов.

## Загружаемый модуль ядра

В ОС Linux драйверы могут быть собраны отдельно от ядра и подгружены в процессе работы, когда это необходимо. Для этой цели используются загружаемые модули.

*Загружаемый модуль ядра* (Loadable Kernel Module, LKM) – 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.

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.

## Драйвер символьного устройства

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.

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).

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

# Практическая часть

## Загружаемый модуль

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

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

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

Так же потребуются команды:

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

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

Создадим файл **moduleTest.c** с кодом модуля:

Листинг 1. moduleTest.c

|  |
| --- |
| #include <linux/module.h>  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Gizzatullina Anzhela");  MODULE\_DESCRIPTION("test modules");  static int \_\_init md\_init(void) {  printk("Hello, world\n");  return 0;  }  static void \_\_exit md\_exit(void) {  printk("Goodbye, cruel world\n");  }  module\_init(md\_init);  module\_exit(md\_exit); |

Для сборки используется утилита **make** и следующий Makefile:

Листинг 2. Makefile

|  |
| --- |
| ifeq ($(KERNELRELEASE),)  KERNELDIR ?= /lib/modules/$(shell uname -r)/build  PWD := $(shell pwd)  .PHONY: build clean  build:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  clean:  rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c  else  $(info Building with KERNELRELEASE = ${KERNELRELEASE})  obj-m := moduleTest.o  endif |

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

Макросы **module\_init** и **module\_exit** (linux/init.h) – самая важная часть кода: в них передаются функции инициализации и удаления модуля.

Сигнатуры функций **static int md\_init(void)** и **static void md\_exit(void)** включают ключевое слово **static**, указывающий на то, что они не экспортируемые и будут использоваться только один раз при загрузке/выгрузке.

Функция **md\_init** возвращает:

* 0 – если модуль успешно установлен;
* отрицательное значение – при ошибке установки модуля.

Перед именем функций можно увидеть макросы **\_\_init** и **\_\_exit** – они используются для оптимизации (linux/module.h) и 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*).

Макросы MODULE\_LICENSE, MODULE\_AUTHOR, MODULE\_DESCRIPTION(linux/module.h) используются для того, чтобы задать информацию о модуле.

Информацию собранного модуля с расширением *.ko*. можно посмотреть с помощью команды **modinfo**.

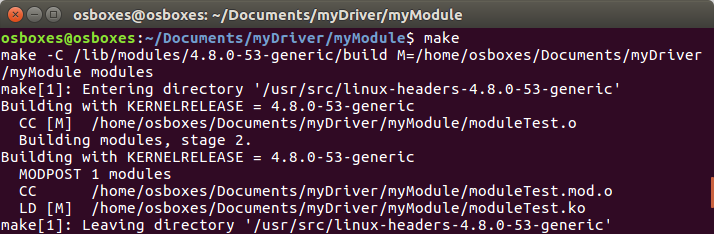
Для ввода в журнал сообщений ядра используется функция **printk**. Данная функция похожа на привычную **printf,** за исключением того, что не может выводить числа с плавающей точкой и не предназначена для выдачи дампа своих данных в консоль.

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*.

Обычно местом, куда перенаправляются сообщения всех уровней журналирования, является журнальный файл */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.

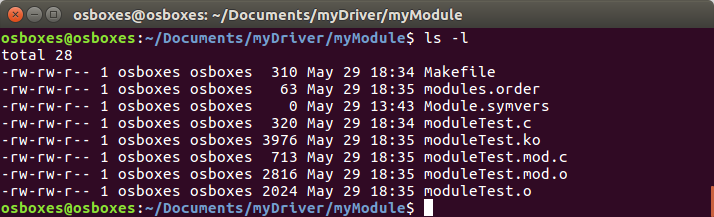
Произведем сборку модуля:

|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ make |



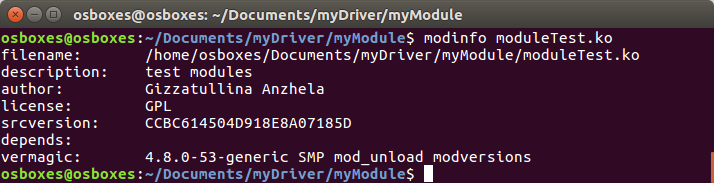
В текущей директории **~/Documents/myDriver/myModule** появились следующие файлы:

|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ ls -l |



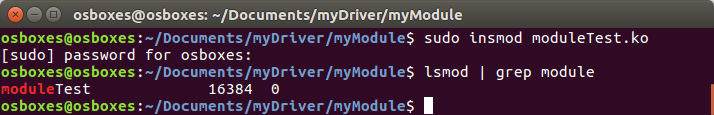
Посмотрим информацию о модуле с помощью команды **modinfo**:

|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ modinfo moduleTest.ko |



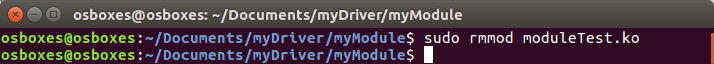
Произведем установку модуля c помощью команды **insmod** и проверим, установился ли модуль, командой **lsmod**:

|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ sudo insmod moduleTest.ko osboxes@osboxes:~/Documents/myDriver/myModule$ lsmod | grep module |



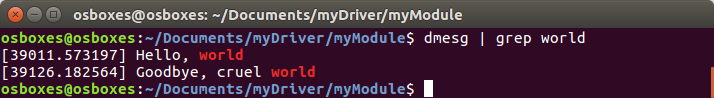
Команда **rmmod** удаляет модуль:

|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ sudo rmmod moduleTest.ko |



Посмотрим, что записалось в лог:

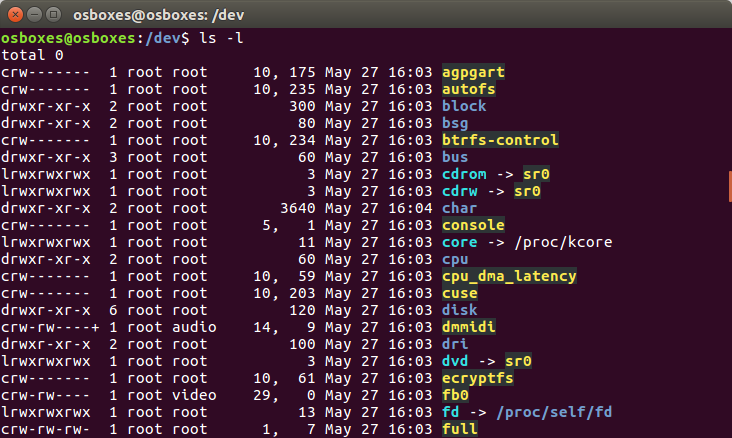
|  |
| --- |
| osboxes@osboxes:~/Documents/myDriver/myModule$ dmesg | grep test |



Видим, что модуль отработал и записал в лог информацию, которую мы задали при установке и удалении.

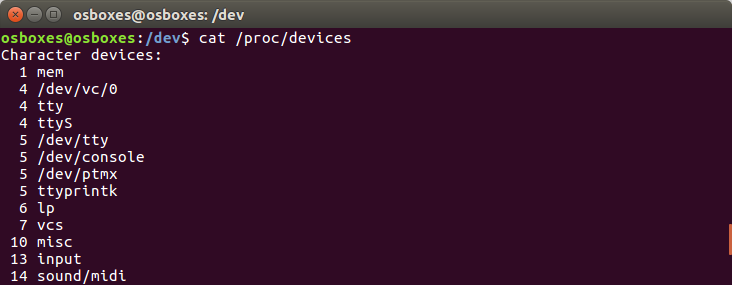
## Драйвер символьного устройства

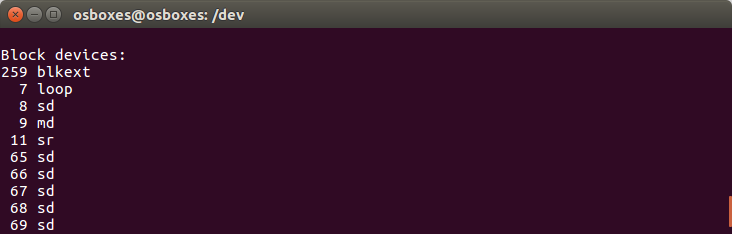
Файлы всех устройств можно найти в директории /dev (файловая система dev).



В каждой строчке между группой и датой последней модификации можно 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*:





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

Создание драйвера символьного устройства подразумевает следующие шаги:

* Создание модуля драйвера символьного устройства
* Регистрация драйвера
* Создание файла устройства.

В папке **/Documents/myDriver/** создадим файл **driverTest.c** с кодом модуля:

Листинг 3. driverTest.c

|  |
| --- |
| #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>  #include <linux/device.h>  #include <linux/cdev.h>  #include <linux/uaccess.h>  MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Gizzatullina Anzhela");  MODULE\_DESCRIPTION("Test Character Driver");  static dev\_t first;  static struct cdev c\_dev;  static struct class \*cl;  static char message[256] = {0};  static short size\_of\_message;  static int dev\_open(struct inode \*, struct file \*);  static int dev\_release(struct inode \*, struct file \*);  static ssize\_t dev\_read(struct file \*, char \*, size\_t, loff\_t \*);  static ssize\_t dev\_write(struct file \*, const char \*, size\_t, loff\_t \*);  static struct file\_operations fops =  {  .owner = THIS\_MODULE,  .open = dev\_open,  .release = dev\_release,  .read = dev\_read,  .write = dev\_write  };  static int \_\_init chr\_init(void) /\* Constructor \*/  {  if (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)  {  return -1;  }    if ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)  {  unregister\_chrdev\_region(first, 1);  return -1;  }    if (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)  {  class\_destroy(cl);  unregister\_chrdev\_region(first, 1);  return -1;  }    cdev\_init(&c\_dev, &fops);    if (cdev\_add(&c\_dev, first, 1) == -1)  {  device\_destroy(cl, first);  class\_destroy(cl);  unregister\_chrdev\_region(first, 1);  return -1;  }  return 0;  }  static void \_\_exit chr\_exit(void) /\* Destructor \*/  {  cdev\_del(&c\_dev);  device\_destroy(cl, first);  class\_destroy(cl);  unregister\_chrdev\_region(first, 1);  }  static int dev\_open(struct inode \*i, struct file \*f)  {  printk(KERN\_INFO "Character driver: open()\n");  return 0;  }  static int dev\_release(struct inode \*i, struct file \*f)  {  printk(KERN\_INFO "Character driver: close()\n");  return 0;  }  static ssize\_t dev\_read(struct file \*f, char \_\_user \*buffer, size\_t len, loff\_t \*offset)  {  int error\_count = 0;  int size = 0;  // copy\_to\_user has the format ( \* to, \*from, size) and returns 0 on success  error\_count = copy\_to\_user(buffer, message, size\_of\_message);  if (error\_count != 0) { // if true then have success  printk(KERN\_INFO "Failed to send %d characters to the user\n", error\_count);  return -EFAULT; // Failed -- return a bad address message (i.e. -14)  }    printk(KERN\_INFO "Sent %d characters to the user\n", size\_of\_message);  size = size\_of\_message;  size\_of\_message = 0;  return size; // clear the position to the start and return 0  }  static ssize\_t dev\_write(struct file \*f, const char \_\_user \*buffer, size\_t len, loff\_t \*offset)  {  sprintf(message, "%s(%d letters) \n", buffer, (int)len); // appending received string with its length  size\_of\_message = strlen(message); // store the length of the stored message  printk(KERN\_INFO "Received %d characters from the user\n", (int)len);  return len;  }  module\_init(chr\_init);  module\_exit(chr\_exit); |

Makefile аналогичен файлу загрузочного модуля, который рассматривался выше:

Листинг 4. Makefile

|  |
| --- |
| ifeq ($(KERNELRELEASE),)  KERNELDIR ?= /lib/modules/$(shell uname -r)/build  PWD := $(shell pwd)  .PHONY: build clean  build:  $(MAKE) -C $(KERNELDIR) M=$(PWD) modules  clean:  rm -rf \*.o \*~ core .depend .\*.cmd \*.ko \*.mod.c  else  $(info Building with KERNELRELEASE = ${KERNELRELEASE})  obj-m := driverTest.o  endif |

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

|  |
| --- |
| 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:

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

Первый шаг в модуле выполняет следующий код:

|  |
| --- |
| if (alloc\_chrdev\_region(&first, 0, 1, "chr\_test\_driver") < 0)  {  return -1;  } |

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.*

|  |
| --- |
| 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.

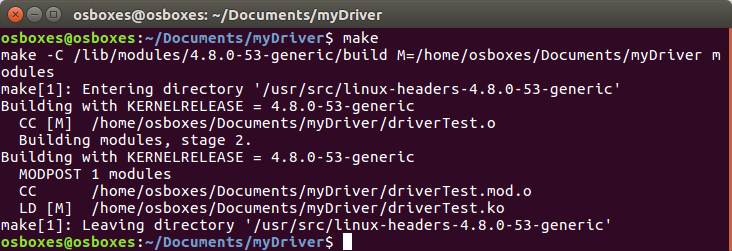
После этого регистрируется класс устройства (**class\_create**) и файл устройства (**chrdev\_region**). В функцию создания файла устройства третьим параметром передается ранее заполненная структура **dev\_t**.

|  |
| --- |
| if ((cl = class\_create(THIS\_MODULE, "char\_test\_drv")) == NULL)  {  unregister\_chrdev\_region(first, 1);  return -1;  }    if (device\_create(cl, NULL, first, NULL, "chrnull") == NULL)  {  class\_destroy(cl);  unregister\_chrdev\_region(first, 1);  return -1;  } |

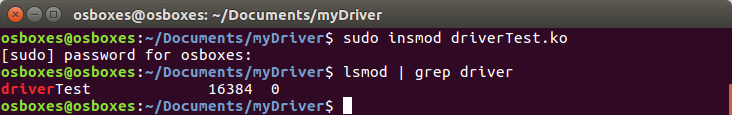
Затем выполняется второй шаг. Устройство инициализируется операциями. Для этого используется функция **cdev\_init** (linux/cdev.h), в которую передается ссылка на структуру описывающее символьное устройство **cdev** (linux/cdev.h) и ссылка на структуру **file\_operations** (linux/fs.h). В последней реализован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

|  |
| --- |
| static struct file\_operations fops =  {  .owner = THIS\_MODULE,  .open = dev\_open,  .release = dev\_release,  .read = dev\_read,  .write = dev\_write  }; |

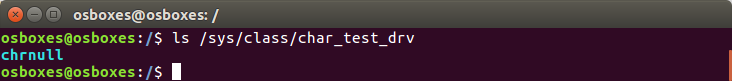
Результат сборки программы:



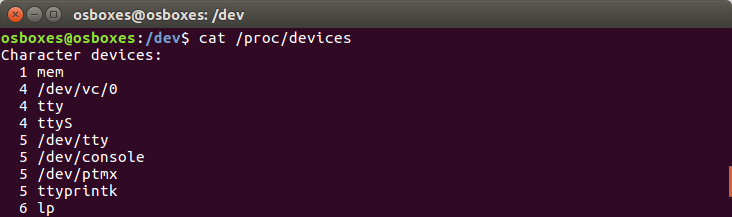
Произведем установку модуля и проверим, прошла ли она успешно:

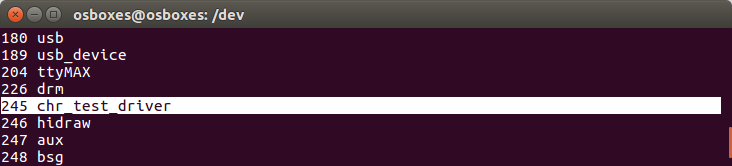


Проверим добавление класса устройств:



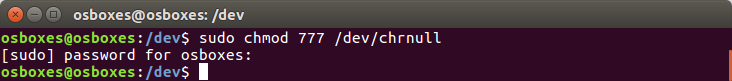
И проверим, произошло ли добавление драйвера:



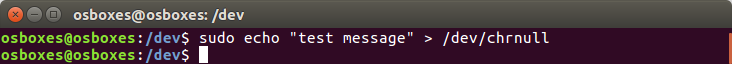


Как видим, наш драйвер создан со старшим номером 245.

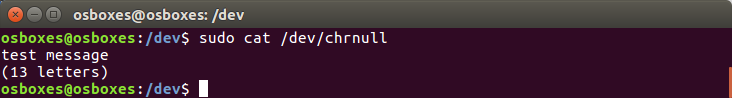
Изменим права доступа к файлу устройства, чтобы можно было с ним работать:



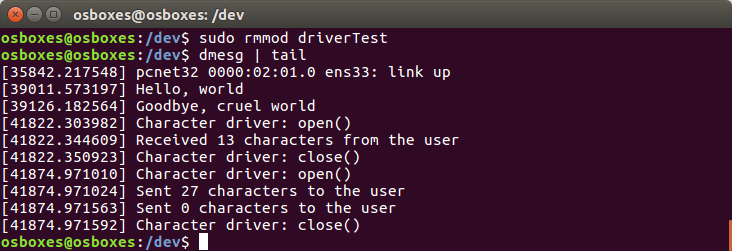
Вывод в файл устройства сообщения:



Чтение из файла устройства сообщения. Вывод в консоль:



Драйвер работает корректно. Чтобы выгрузить модуль из ядра используем следующую команду:



# Источники

1. Создание драйверов для ОС Linux.  
   URL: <http://linux-doc.narod.ru/development/drivers.html>