

**РТУ МИРЭА (ТУ)**

Центр проектирования интегральных схем, устройств наноэлектроники и  
микросистем

**Методическое пособие по лабораторным  
работам**

«Багет-ПЛК1-01»  
(ОС Linux, Debian 10)



Москва

2023

# Оглавление

Введение	2
1 Установка ОС	9
2 Ознакомительная	17
3 Модуль ядра	25

# Введение

Методическое пособие описывает основные шаги по разработке программного обеспечения для встраиваемых систем. В качестве целевой платформы используется плата «Багет ПЛК1-01» с установленным микроконтроллером Комдив-МК К5500ВК018, производства ФГУ ФНЦ НИИСИ РАН. В состав К5500ВК018 входит микропроцессорное ядро с архитектурой MIPS64, включая сопроцессор вещественной арифметики (IEEE754), кэш-память первого уровня (16+16 кбайт), кэш-память второго уровня (128 кбайт), буфер трансляции виртуальных адресов на 64 адреса, контроллер динамической памяти DDR3L, восемь таймеров-счетчиков, сторожевой таймер, часы реального времени, два контроллера Ethernet 10/100, контроллеры последовательных интерфейсов RS-232C (4 шт.), I2C (2 шт.), SPI (3 шт.), QSPI, CAN 2.0 (2 шт.), USB 2.0. Серийный выпуск начат в 2021 году.

ПЛК1-01 выполнен в виде печатной платы без корпуса, на которой расположены соединители для подключения первичного электропитания и внешних устройств, органы управления (пользовательские кнопки SW1, SW2), светодиодные индикаторы (VD1 – VD5), переключики для управления режимами работы ПЛК1-01 (SA4, SA5, SA6, SA7, SA9).

Назначение	Вывод процессора
Кнопка SW2	GPIO D 6
Зелёный светодиод VD2	GPIO D 4
Жёлтый светодиод VD5	GPIO D 5

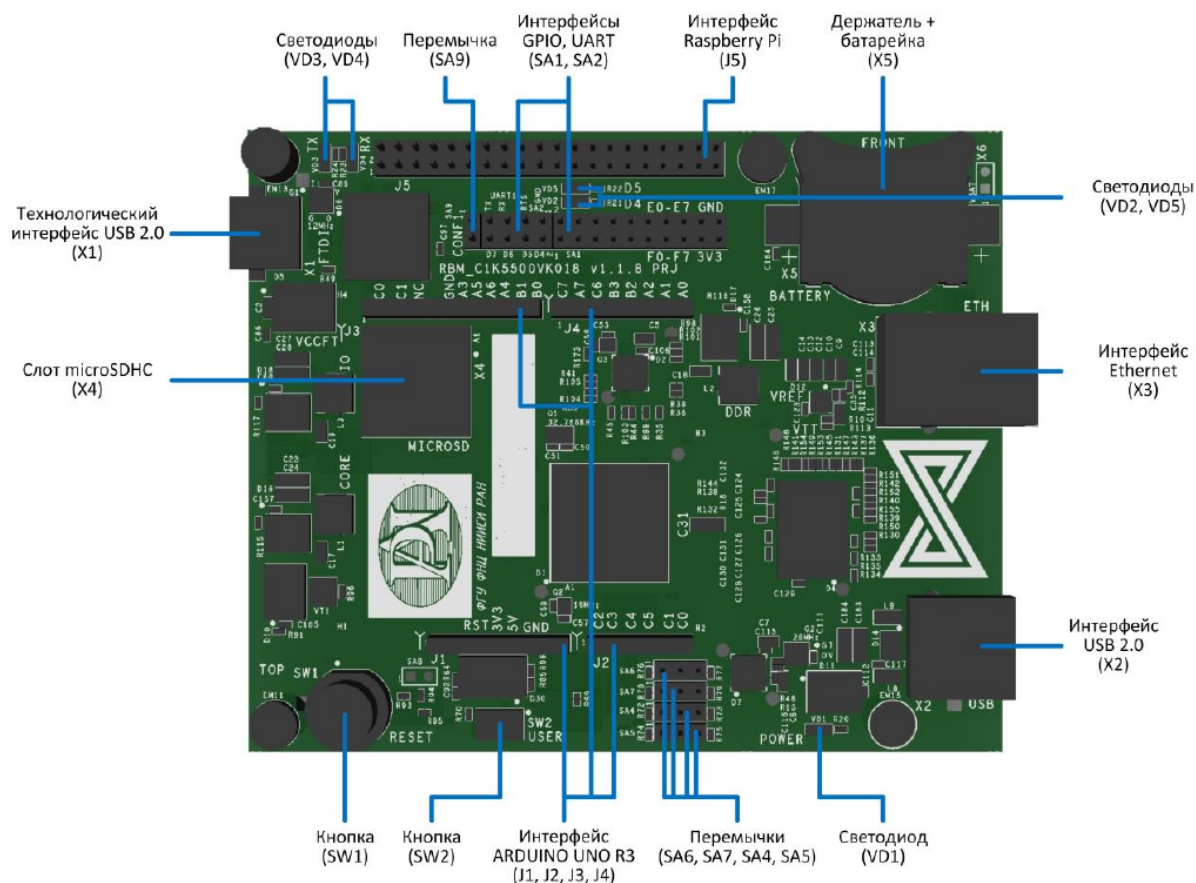


Рисунок 1. Расположение портов ввода-вывода

Для организации питания часов реального времени (RTC) микроконтроллера K5500BK018 при отсутствии основного питания предусмотрена установка автономного элемента питания – батарейки типа CR2032 +3 В. Расположение соединителей, органов управления и индикаторов ПЛК1-01 приведено на рисунке 1, назначение выводов на некоторых из разъёмов приведены на рисунке 2.

## Переходная плата

Для удобства подключения к интерфейсам целевой платы, была разработана переходная плата, с выведенными сигнальными линиями, сгруппированными по интерфейсам (рисунок 3).

UART2 можно использовать как для работы с интерфейсом UART2,

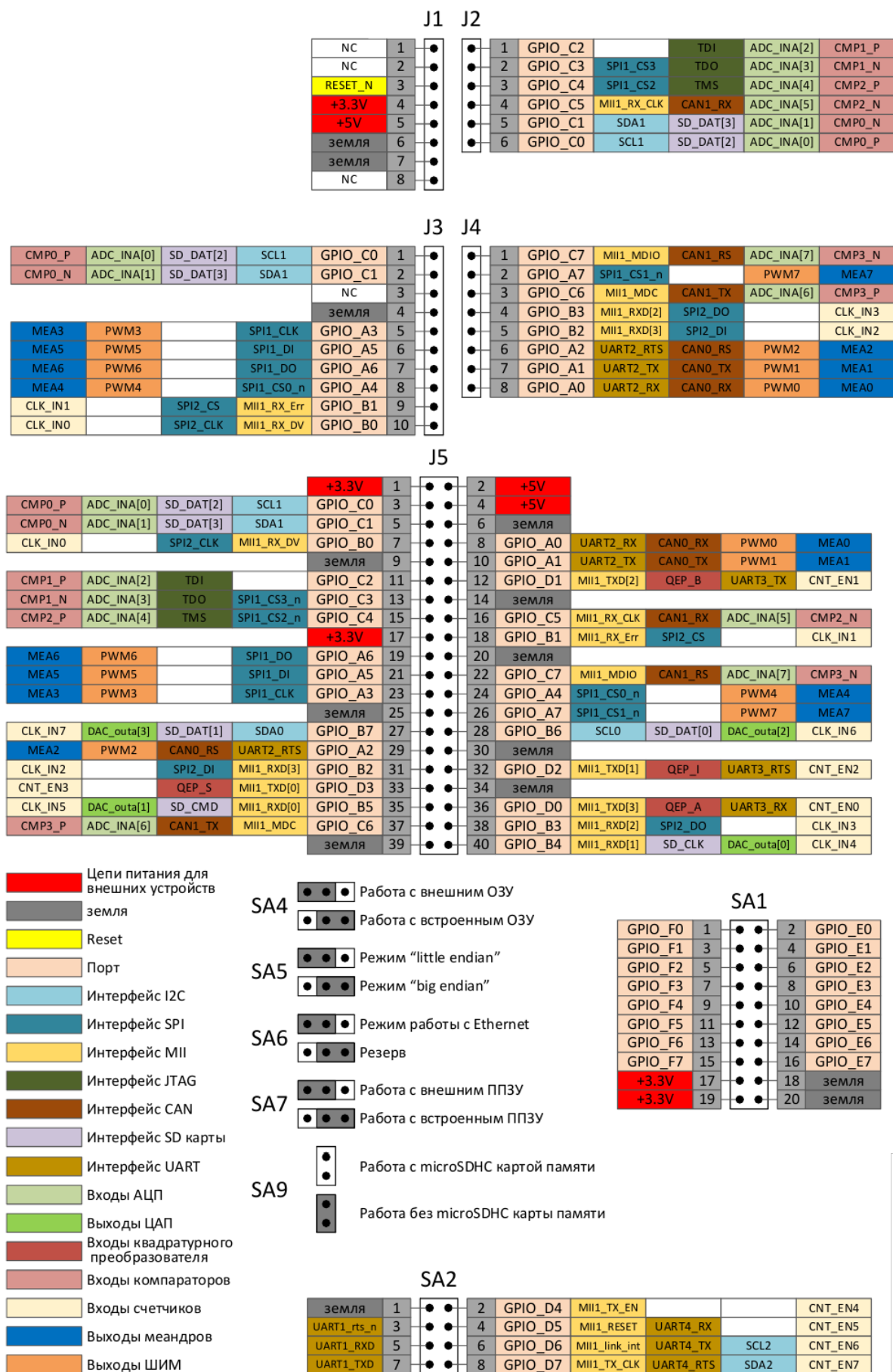


Рисунок 2. Разводка выводов процессора по разъёмам ввода-вывода

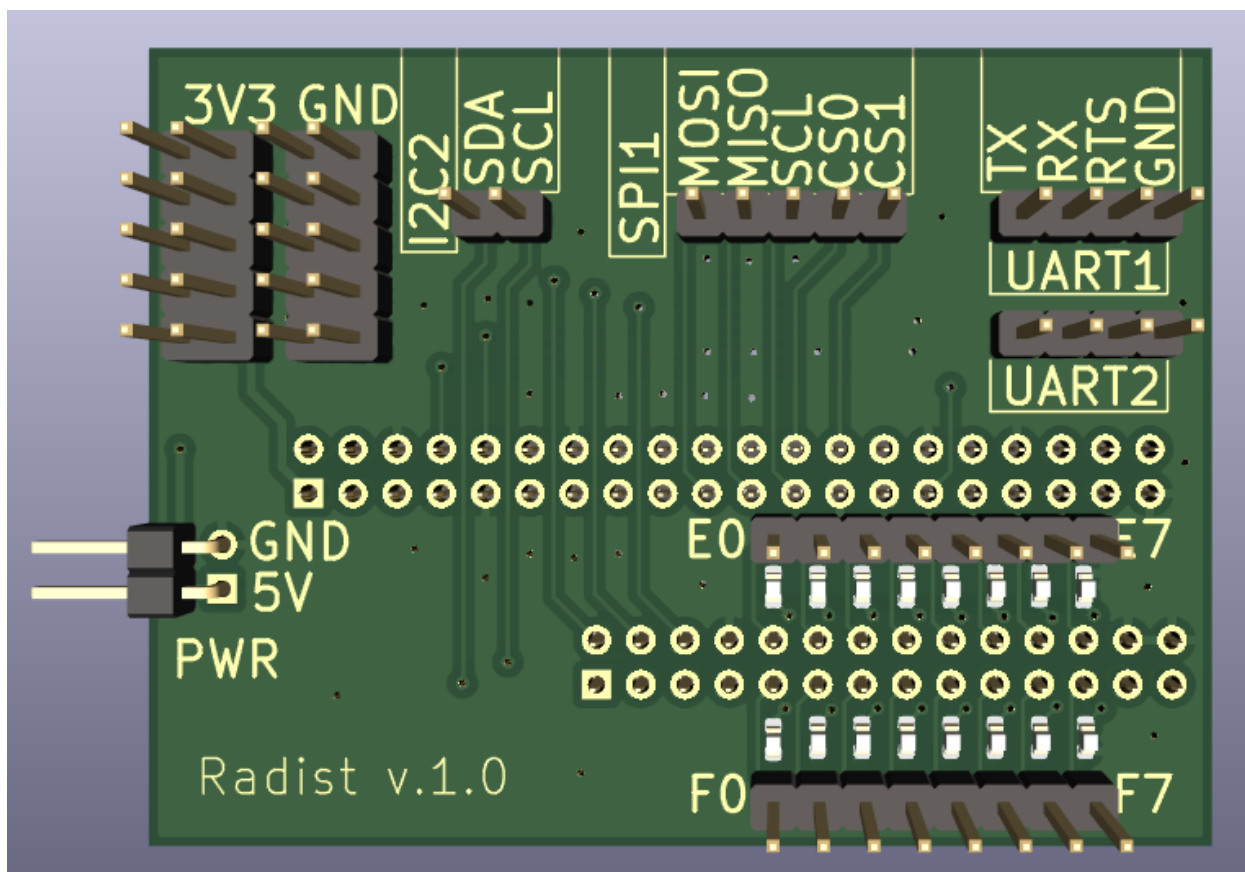


Рисунок 3. Переходная плата

так и для работы с интерфейсом CAN0. Выводы GPIO банков E и F дублируются на светодиоды для отладочных целей. При необходимости, можно подключать к выводам кнопки, или использовать их для формирования управляющих сигналов для модулей. Также обратите внимание, что одна из сигнальных линий I2C интерфейса используется кнопкой SW2 на плате, по этой причине не стоит использовать кнопку при активации этого интерфейса.

## Как запускается Linux?

Для запуска ОС Linux в общем случае должны пройти следующие этапы:

1. ROM BOOT — запуск встроенного загрузчика, определяющего ис-

точник дальнейшей загрузки. Этот код нельзя исправить его работа и формат образа (заголовки, значение бит и прочее) зависит от производителя.

2. First Stage Bootloader (FSBL) — загрузчик первой ступени, задача этого загрузчика, начальная инициализация процессора, и загрузка пользовательского кода в DDR память. Данный загрузчик опциональный, и у некоторых производителей может отсутствовать. Работает из встроенной RAM памяти процессора.
3. Second Stage Bootloader (SSBL) — это загрузчик второго уровня, задача которого заключается в подготовке системы к запуску ядра Linux, с пользовательскими параметрами (точка монтирования rootfs, скорость работы терминала для логов и прочее). Наиболее распространённый вариант загрузчика проекта Das U-Boot. В целевой плате используется другой загрузчик, набирающий популярность — Bearbox.
4. Kernel — на этом этапе за работу процессора отвечает ядро, происходит финальная конфигурация, активация дополнительных ядер (если они есть и активирована функция многоядерной поддержки), монтирование rootfs, запуск сервисов и прочее.

## Описание окружения

Для работы с платой используется виртуальная машина на базе Ubuntu 20.04. На виртуальной машине установлены пакеты для компиляции кода под Linux для MIPS64 архитектуры `mips64el-linux-gnu-gcc` (C компилятор) и `mips64el-linux-gnu-g++` (C++ компилятор), компилятор дерева устройств `device-tree-compiler`, архив с заголовочными файлами ядра Linux установленного на плате, утилита `debootstrap`, IDE VSCode и прочие полезные инструменты.

Имя пользователя **student** пароль **usrstudent**. Пользователь поддерживает работу через `sudo` (получение прав суперпользователя он же администратор).

В конце некоторых работ есть задания для самостоятельного выполнения. При этом есть простые задачи, предполагающие незначительное изменение кода, а также есть помеченные \*. Последние выполняются по желанию, при этом нужно быть готовым, что придётся активно пользоваться поиском, и расходовать клетки серого вещества.

**ВНИМАНИЕ!** Запомните, что далее по тексту, если просят выполнить команду, в начале которой написано **\$** , значит команду нужно выполнить на целевой плате, а если с **#** - в консоли виртуальной машины. При этом **\$** или **#** писать не нужно!

Если команда заканчивается символом **\** и на следующей строке есть запись, то это запись относится к одной команде, Вы можете записать как видите, так как символ **\** в консоли Linux означает многострочную команду или, опустив символ **\**, продолжить вводить текст приведённый на следующей строке методички.

### Примеры:

Авторы хотят, что бы Вы в консоли виртуальной машины (ВМ) записали `echo Hello` и нажали кнопку Enter:

---

```
# echo Hello
```

---

Авторы хотят, что бы Вы в консоли целевой платы записали `echo FOOBAR` и нажали кнопку Enter:

---

```
$ echo FOOBAR
```

---

Авторы хотят, что бы Вы в консоли виртуальной машины (ВМ) написали очень длинную команду (да-да, вечер не обещает быть томным) и



нажали кнопку Enter:

---

```
# echo \  
FOOBAR
```

---

# Лабораторная работа № 1

## Установка ОС

**Цель:** Установить дистрибутив Debian 10 на целевую платформу.

**Описание:** Нужно понимать, что ОС Linux делится на две части. Ядро (Kernel) и файловую систему (rootfs). Ядро определяет какие возможности по взаимодействию с железом доступны пользователю и приложениям. Rootfs определяет какие инструменты есть в системе, порядок инициализации, запуска сервисов и прочие прикладные штуки. Дистрибутив это в большей степени наполнение rootfs. Не каждый дистрибутив имеет поддержку иной от x86 архитектур.

Для установки debian воспользуемся инструментом под названием debootstrap, который отвечает за создание rootfs. Подробнее про этот инструмент, и возможности по его управлению, Вы можете найти на wiki проекта Debian.

### 1.1 Подготовка rootfs

**1.1.1** Запустите виртуальную машину. Логин и пароль для входа: student / usrstudent.

**1.1.2** Запустите консоль нажав комбинацию клавиш на клавиатуре Ctrl+Alt+T

**Внимание!** В консоли есть возможность автоматического продолжения ввода. Для этого необходимо ввести первые символы команды, или пути, и нажать клавишу TAB. Ввод продолжится до тех пор, пока не появится неопределённостью (к примеру у Вас есть два файла `foo_bar1` и `foo_bar2`, при нажатии TAB будет вставлен текст до цифры). Двойное нажатие TAB приведёт к выводу всех возможных вариантов продолжения, если таковые есть.

**1.1.3** Создайте и перейдите в рабочий каталог в котором будет создан образ `rootfs`.

---

```
# mkdir -p $BAGET/lab_01
# cd $BAGET/lab_01
```

---

**1.1.4** Запустите утилиту `debootstrap` (при необходимости введите пароль `usrstudent`):

---

```
# sudo debootstrap --include=aptitude,nano,wget \
--foreign \
--arch=mips64el buster rootfs
```

---

*--include=A,B,C..* - добавить в сборку указанные пакеты

*--foreign* — только сгенерировать наполнение, применяется когда архитектура на которой запускается утилита отлична от архитектуры назначения.

*--arch=mips64el* — указываем целевую архитектуру

*buster* — версия сборки

*rootfs* — путь к папке назначения, где будут размещены файлы

**1.1.5** Для продолжения установки, нам понадобится утилита `qemu` позволяющая эмулировать различные архитектуры. Скопируем исполняемый

файл qemu:

---

```
# sudo cp /usr/bin/qemu-mips64el-static ./rootfs/usr/bin
```

---

и перейдём в созданную rootfs (привет дедушка контейнеров, chroot)

---

```
# sudo chroot ./rootfs
```

---

## 1.2 Настройка rootfs

### 1.2.1 Завершим работу debootstrap

---

```
# export LANG=en_US.UTF-8
# /debootstrap/debootstrap --second-stage
```

---

**1.2.2** Добавим источники для установки ПО, для этого выполним следующие команды

---

```
# echo deb http://ftp.debian.org/debian buster \
main contrib non-free >> /etc/apt/sources.list

# echo deb-src http://ftp.debian.org/debian buster \
main contrib non-free >> /etc/apt/sources.list

# echo deb http://ftp.debian.org/debian buster-updates \
main contrib non-free >> /etc/apt/sources.list

# echo deb-src http://ftp.debian.org/debian buster-updates \
main contrib non-free >> /etc/apt/sources.list
```

---

### 1.2.3 Обновим список, и установим ряд приложений

---

```
# apt-get update
# apt-get install -y dialog sudo less i2c-tools evtest mc \
openssh-server resolvconf hwinfo net-tools
```

---

### 1.2.4 Зададим пароль для root пользователя

---

```
# passwd root
```

---

После чего введите root (внимание, курсор двигаться не будет, это политика безопасности Linux, при вводе пароля курсор не перемещается, что бы нельзя было установить количество символов). И нажмите Enter

Затем Вас попросят повторить пароль, снова введите root и нажмите Enter

### 1.2.5 Настроим работу сетевого интерфейса со статическим адресом, для этого создадим файл в паке /etc/network/interfaces.d/

---

```
# nano /etc/network/interfaces.d/eth0
```

---

и впишите туда следующие строки:

```
auto eth0
    iface eth0 inet static
        address 192.168.100.200
        netmask 255.255.255.0
        network 192.168.100.0
```

Первая строка определяет когда настраивается интерфейс. Auto означает, что при загрузке ОС. Для USB адаптеров, лучше выбирать allow-hotplug, что бы сократить время загрузки ОС.

Вторая строка определяет, что мы будем использовать статический ip адрес, далее идёт конфигурация адресов.

Выходим из редактора nano, нажав комбинацию клавиш Ctrl+X. Вам будет задан вопрос о сохранении изменения в файле, жмём Y и клавишу Enter.

### 1.2.6 Настроим имя компьютера

---

```
# echo baget > /etc/hostname
```

---

## 1.2.7 Завершим настройку rootfs

```
# exit
# sudo rm ./rootfs/usr/bin/qemu-mips64el-static
```

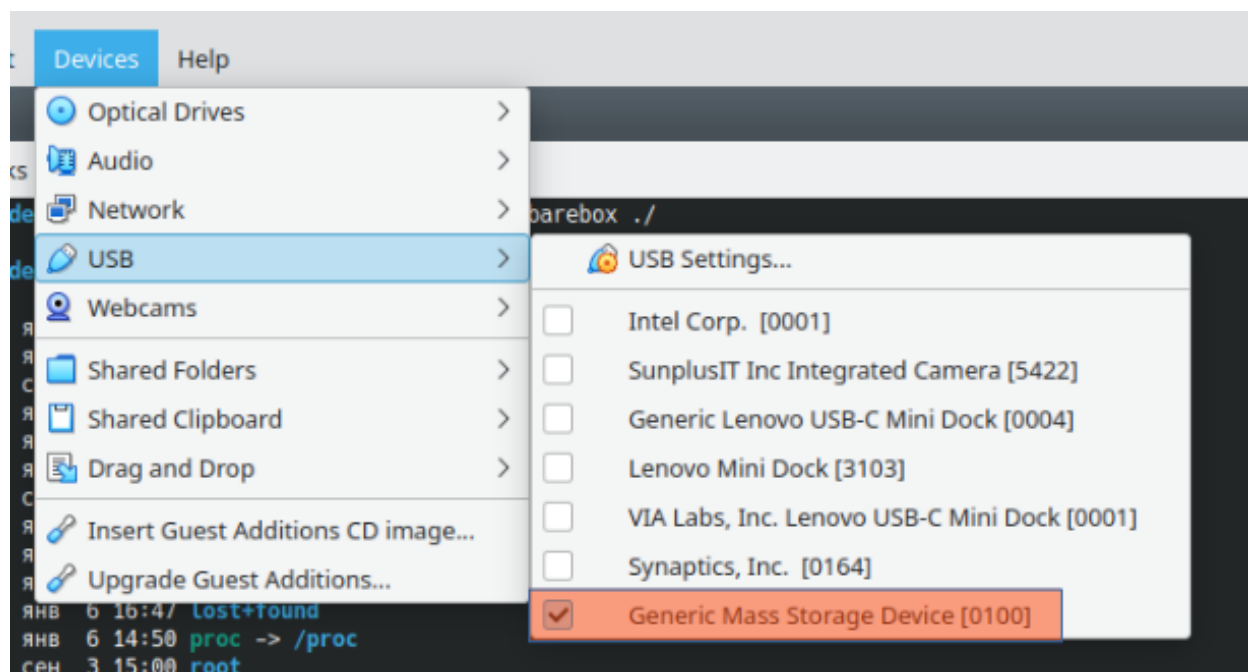
## 1.2.8 Скопируйте папку barebox из каталога support в корень созданной rootfs

```
# sudo cp -r $BAGET/support/barebox/ ./rootfs/
```

В данной папке лежит скомпилированное ядро Linux (vmlinux...), скомпилированное дерево устройств (k5500vk018\_rbm.dtb) и скрипт выполняемый загрузчиком (barebox.sh).

# 1.3 Проверка

## 1.3.1 Вставьте SD карту в ПК, и подключите её к виртуальной машине



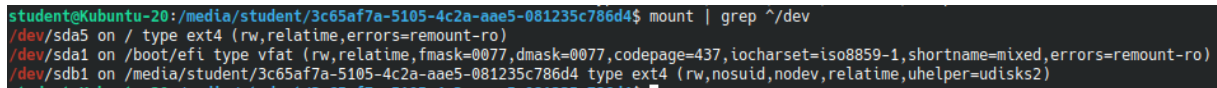
1.3.2 Необходимо узнать точку монтирования SD карты. Она может отличаться в зависимости от ридера, что Вы используете. Для упрощения ввода, выполните команду в консоле:

---

```
# mount | grep ^/dev
```

---

для получения списка примонтированных устройств, при этом командой `grep` происходит фильтрация лишнего вывода, и отображается только список блочных устройств. Вывод будет примерно как на рисунке



```
student@Kubuntu-20:/media/student/3c65af7a-5105-4c2a-aae5-081235c786d4$ mount | grep ^/dev
/dev/sda5 on / type ext4 (rw,relatime,errors=remount-ro)
/dev/sda1 on /boot/efi type vfat (rw,relatime,fmask=0077,dmask=0077,codepage=437,iocharset=iso8859-1,shortname=mixed,errors=remount-ro)
/dev/sdb1 on /media/student/3c65af7a-5105-4c2a-aae5-081235c786d4 type ext4 (rw,nosuid,nodev,relatime,uhelper=udisks2)
```

Первые две строчки, это разделы виртуального жёсткого диска, далее точка монтирования SD карты (`/media/student/...`)

Для упрощения ввода, создадим временную системную переменную, которой назначим путь, для примера(!) на рисунке выше:

---

```
# export BAGET_SD=\
/media/student/3c65af7a-5105-4c2a-aae5-081235c786d4
```

---

Нужно помнить, что эта переменная существует только в том терминале, где был вызван `export`, в остальных окнах терминала она не существует, если Вы закроете текущее окно, или Вам понадобится эта переменная в другом окне, необходимо будет повторить процедуру присвоения.

### 1.3.3 Удалите все файлы с sd карты

---

```
# sudo rm -rf $BAGET_SD/*
```

---

скопируйте созданную `rootfs` на sd карту

---

```
# sudo cp -a $BAGET/lab_01/rootfs/* $BAGET_SD/
```

---

и отмонтируйте SD карту

---

```
# sudo umount $BAGET_SD
```

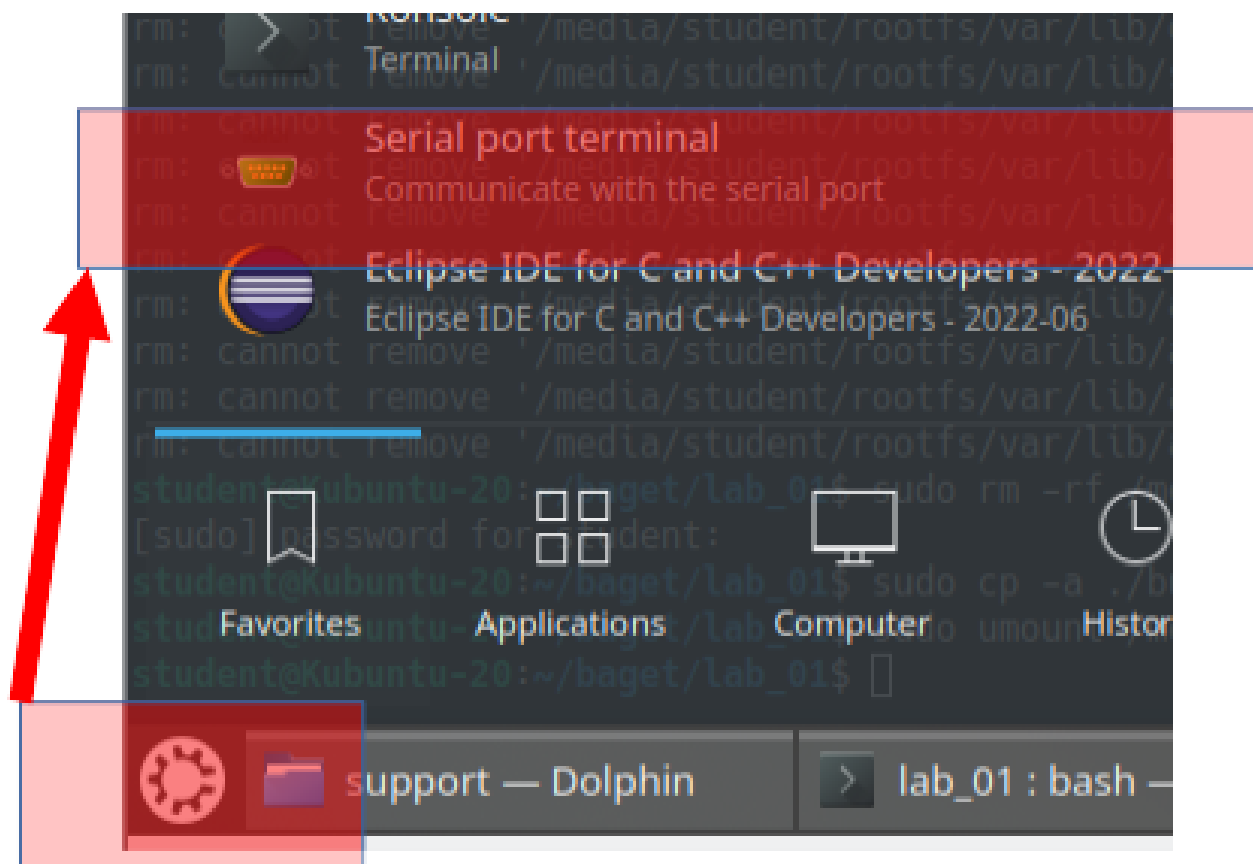
---

(это важно, так как в ОС Linux запись данных на носители происходит в фоне, и если вы отключите накопитель «неправильно» то нет гарантии,

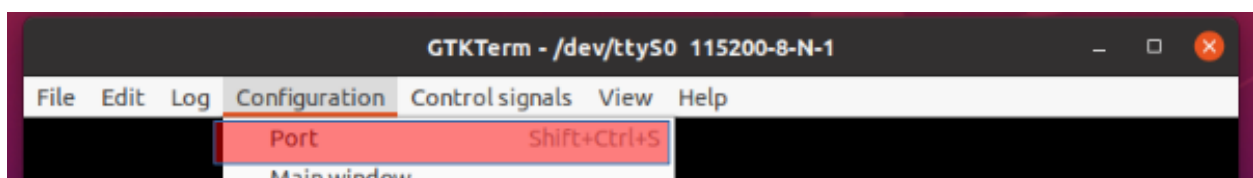
что данные успели записаться на носитель!). Дождитесь когда команда выполниться, после чего достаньте SD карту и вставьте её в плату.

**1.3.4** Подключите USB шнуром плату к ПК. Проверьте, и при необходимости подключите USB устройство FTDI RBM\_C1K5500VK018 к виртуальной машине (меню Device→USB).

**1.3.5** Запустите программу gtkterm или нажмите Ctrl + Shift + T для открытия новой вкладки в окне терминала и выполните команду minicom (это консольная утилита, в виртуальной машине эта утилита уже настроена на нужный порт, с нужными параметрами)



**1.3.6** Выберите порт /dev/ttyUSB1 через меню Configuration->Port.





Нажмите кнопку ОК.

Если в окне не появился текст, и на нажатие клавиши Enter так же нет реакции, нажмите на кнопку Reset на плате. Изучите вывод, там будет написано, что пошло не так.

Один из вариантов, когда при записи файлов на SD карту что-то поби-лось. В этом случае, вы увидите много записей как в примере ниже:

```
srisa-sdhci 1b50b000.sdhci@1b50b000.of: registered as mci0
srisa-sdhci 1b50b000.sdhci@1b50b000.of: SDHCI timeout while waiting for
done
srisa-sdhci 1b50b000.sdhci@1b50b000.of: error on command 8
srisa-sdhci 1b50b000.sdhci@1b50b000.of: state = 0400 03ff, interrupt =
8001 0001
srisa-sdhci 1b50b000.sdhci@1b50b000.of: r0 00000000 r1 00000000 r2
00000000 r3 00000000
```

Нужно повторить все шаги этого раздела. **При вынимании SD кар-ты убедитесь, что плата обесточена!**

### 1.3.7 Введите в качестве логина и пароля root

Поздравляем, Вы запустили Debian дистрибутив на плате.

### 1.3.8 Выключите плату, для чего в начале введите команду

---

```
$ poweroff
```

---

дождитесь, как появиться надпись

```
reboot: System halt
```

после чего отключите USB кабель от ПК или платы.

# Лабораторная работа № 2

## Ознакомительная

**Цель:** Запустить плату, подключиться к ней через консоль и по сетевому интерфейсу, работа с кнопкой и светодиодами через sysfs.

**Описание:** работа с ОС Linux на целевой платформе мало чем отличается от работы с данной ОС на обычном компьютере. Среди особенностей можно выделить наличие меньшего набора инструментов, и рабочих программ, что делается для экономии свободного места, и ускорения загрузки ОС. Так же может отсутствовать привычный рабочий стол, а вся работа сводиться к взаимодействию через рабочий терминал, она же консоль. Для подключения к консоли можно воспользоваться интерфейсом UART (он же COM порт, или последовательный порт), или при помощи сетевого соединения (ssh или реже telnet).

При этом нужно понимать, что возможность работы через консоль настраивается, и не всегда может быть доступна по-умолчанию (обычно возможность сетевого подключения отключают, для повышения безопасности).

## 2.1 Запуск и подключение к устройству

**2.1.1** Запустите виртуальную машину. Логин и пароль для входа: student / usrstudent.

**2.1.2** Подключите по USB плату к ПК. Проверьте, и при необходимости подключите USB устройство FTDI RBM\_C1K5500VK018 к виртуальной машине (меню Device→USB).

**2.1.3** Откройте программу gterm, и подключитесь к порту /dev/ttyUSB1

**2.1.4** Если в окне терминала нет текста, нажмите клавишу Enter на клавиатуре. Вы должны увидеть следующий вывод:



Вы подключились к консоли устройства. Введите логин root и пароль root.

## 2.2 Подключение по сетевому интерфейсу

Рассмотрим какие шаги необходимо выполнить, для того, что бы можно было организовать сетевое подключение к плате, для получения доступа к консоли, и файловой системе через SSH соединение. Так же настройки, которые будут выполнены в данном разделе помогут в дальнейшем копировать файлы на плату через команду scp.

**2.2.1** Проверьте текущие настройки интерфейса eth0 выполнив команду:

---

```
$ ip addr show dev eth0
```

---

Вы должны увидеть, что выдаче появилась строка:

```
...
inet 192.168.100.200/24 scope global eth0
...
```

Если inet адрес отличается, то удалите текущий адрес (вместо <inet\_val> впишите адрес, который отобразился у вас)

---

```
$ ip addr del <inet_val> dev eth0
```

---

затем установите новый адрес командой

---

```
$ ip addr add 192.168.100.200/24 dev eth0
```

---

Для активации изменения необходимо переподключиться, для чего выполните следующие команды

---

```
$ ip link set eth0 down
$ ip link set eth0 up
```

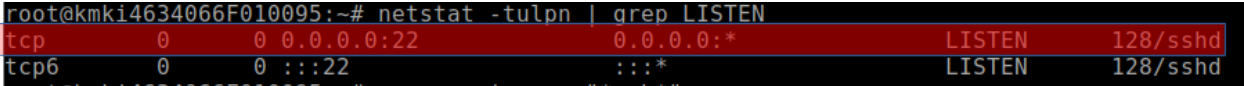
---

**2.2.2** Проверьте, что на плате запущен SSH сервер. Проверить открытые порты устройства, выполнив команду:

---

```
$ netstat -tulpn | grep LISTEN
```

---



```
root@kmki4634066F010095:~# netstat -tulpn | grep LISTEN
tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN 128/sshd
tcp6 0 0 :::22 :::* LISTEN 128/sshd
```

Если вы не увидели, вывода, как на картинках выше, то выполните команду:

---

```
$ systemctl start sshd
```

---

и проверьте снова.

**2.2.3** Заходить через сеть в пользователя root не безопасно, поэтому создадим пользователя для дальнейшей работы, для этого введите следующие команды (пароль для пользователя netuser - usernetuser):

---

```
$ useradd -s /bin/bash -m netuser
$ passwd netuser
```

---

Первая команда для создания нового пользователя netuser с созданием домашнего каталога (/home/netuser) и также назначаем для него использовать интерпретатора bash.

**2.2.4** Подключите сетевой шнур к рабочему ПК и плате

**2.2.5** Откройте на виртуальной машине консоль, нажав на клавиатуре комбинацию клавиш Ctrl + Alt + T.

**2.2.6** В открывшемся окне терминала введите команду:

---

```
# ssh netuser@192.168.100.200
```

---

**2.2.7** Вам предложат ввести пароль пользователя, вводим тот, что установили в пункте 2.2.3 и получаем доступ к терминалу платы.

Если вместо ввода пароля, Вы увидите надпись похожую на:

```
The authenticity of host '192.168.100.200 (192.168.100.200)' can't be
established.
ECDSA key fingerprint is
SHA256:+0SEJhr2dIiTvlWZxq5fZ0UFdYYY+egbTgabe6F7zZE.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Введите yes и нажмите Enter, и Вам предложат ввести пароль.

Если же получите вывод как на картинке ниже

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@  WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!  @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
The fingerprint for the ECDSA key sent by the remote host is
SHA256:0YNVHntNns828eL8NE12XPC1J+TInBg8AFkAzY6Yjnw.
Please contact your system administrator.
Add correct host key in /home/student/.ssh/known_hosts to get rid of this message.
Offending ECDSA key in /home/student/.ssh/known_hosts:1
  remove with:
  ssh-keygen -f "/home/student/.ssh/known_hosts" -R "192.168.100.200"
ECDSA host key for 192.168.100.200 has changed and you have requested strict checking.
Host key verification failed.
lost connection
```

Выполните команду

---

```
$ ssh-keygen -f "/home/student/.ssh/known_hosts" -R "192.168.100.200"
```

---

и вернитесь к пункту 2.2.6

**2.2.8**      Завершите сеанс, введя команду `exit`

## 2.3    Аппаратный Hello world

В этой части Вы будете управлять выводами микроконтроллера, для переключения светодиодов и считывания состояния кнопки, при помощи виртуальной файловой системы `sysfs`.

Для работы с периферией в ОС Linux помимо прочего есть две виртуальные файловые системы: `procfs` (точка входа `/proc`) и `sysfs` (точка входа `/sys`).

`Sysfs` экспортирует в пространство пользователя информацию ядра Linux о присутствующих в системе устройствах и драйверах, что позволяет не только считывать значение (например с датчиков температуры), но и менять поведение, если это было заложено разработчиками.

За формирование наполнения виртуальных файловых систем отвечает модуль ядра, который выполняет роль драйвера, если речь идёт про

взаимодействие с периферийным модулями (подробно будет рассмотрен в следующих лабораторных работах).

### 2.3.1 Перейдите в каталог `/sys/class/gpio`:

---

```
$ cd /sys/class/gpio
```

---

### 2.3.2 Выведите список файлов и каталогов в текущей директории:

---

```
$ ls -l
```

---

```
root@kmk14634066F010095:/sys/class/gpio# ls -l
total 0
--w----- 1 root root 4096 Feb 14 10:12 export
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip464 -> ../../devices/platform/lb400480.gpio/gpio/gpiochip464
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip472 -> ../../devices/platform/lb400400.gpio/gpio/gpiochip472
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip480 -> ../../devices/platform/lb400380.gpio/gpio/gpiochip480
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip488 -> ../../devices/platform/lb400300.gpio/gpio/gpiochip488
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip496 -> ../../devices/platform/lb400280.gpio/gpio/gpiochip496
lrwxrwxrwx 1 root root    0 Feb 14 10:12 gpiochip504 -> ../../devices/platform/lb400200.gpio/gpio/gpiochip504
--w----- 1 root root 4096 Feb 14 10:12 unexport
```

Разберём то, что Вы видите. Первый и последний файлы используются для того, что бы подключать драйвер для управления отдельным выводом в файловую систему `sysfs` или отключать.

Далее идёт серия символьных ссылок на каталоги, каждый из которых описывает один банк сигналов ввода-вывода общего назначения (GPIO). Тут нужно отметить, что в рабочем чипе присутствует 6 банков (A, B, C, D, E и F), по 8 выводов в каждом, по этой причине мы видим 6 каталогов.

Обратите внимание на имена каталогов, внутри `platform`. Первая часть имени является базовым адресом периферийного модуля, отвечающего за управление банком выводов. Первому банку выводов A соответствует адрес `0x1B400200`, остальные по возрастанию.

### 2.3.3 У выводов общего назначения в рамках ядра Linux нумерация сквозная. Таким образом, можно определить, что вывод подключённый к кнопке SW2 находится под номером 486. Выполним следующую команду:

---

```
$ echo 486 > export
```

---

**2.3.4** Выполните эту же команду для экспорта выводов пары выводов банка F (504 и 505)

**2.3.5** После каждой команды будет создан каталог `./gpioXXX` где вместо XXX будет номер вывода. Если зайти в любой из этих каталогов, то можно увидеть там одинаковый набор файлов, но нас интересуют следующие:

- `direction` — задаёт направление вывода (`in` вывод настроен как вход, `out` — как выход)
- `value` — если вывод настроен как вход, то читая этот файл можем узнать уровень логического сигнала на выводе. Если настроен на выход, то запись в этот файл 0 или 1 будет устанавливать соответствующий уровень выходного сигнала.

**2.3.6** Так как два вывода банка F управляют светодиодами, изменим значение в файле `direction` для них на `out`

---

```
$ echo out > gpio504/direction
$ echo out > gpio505/direction
```

---

**2.3.7** Переключать светодиоды можно записью 0 или 1 в файл `value` соответствующего вывода

---

```
$ echo 1 > gpio504/value
$ echo 0 > gpio504/value
```

---



**2.3.8** Для взаимодействия с кнопкой менять ничего не нужно. Для удобства отладки запустить считывание файла через утилиту `watch` позволяющую выполнять заданную команду через указанный интервал времени.

---

```
$ watch -n 1 cat gpio486/value
```

---

После чего можете нажимать кнопку, и смотреть как меняется значение считанное из файла. В данном примере файл будет считываться каждую секунду (из-за параметра `-n 1` переданного при запуске утилите `watch`).

**2.3.9** Остановите работу программы `watch` комбинацией клавиш `Ctrl + C`

**2.3.10** Введите команду `poweroff`, дождитесь, пока не появится надпись: `reboot: System halt`, после чего отключите USB кабель от ПК.

## 2.4 Задание для закрепления

Экспортируйте все выводы банков F и E, и включите все светодиоды.

\* Написать `bash` скрипт, который будет делать экспорт нужных выводов, если необходимо, и в зависимости от того, нажата кнопка или нет поочередно включать или выключать все светодиоды.

# Лабораторная работа № 3

## Модуль ядра

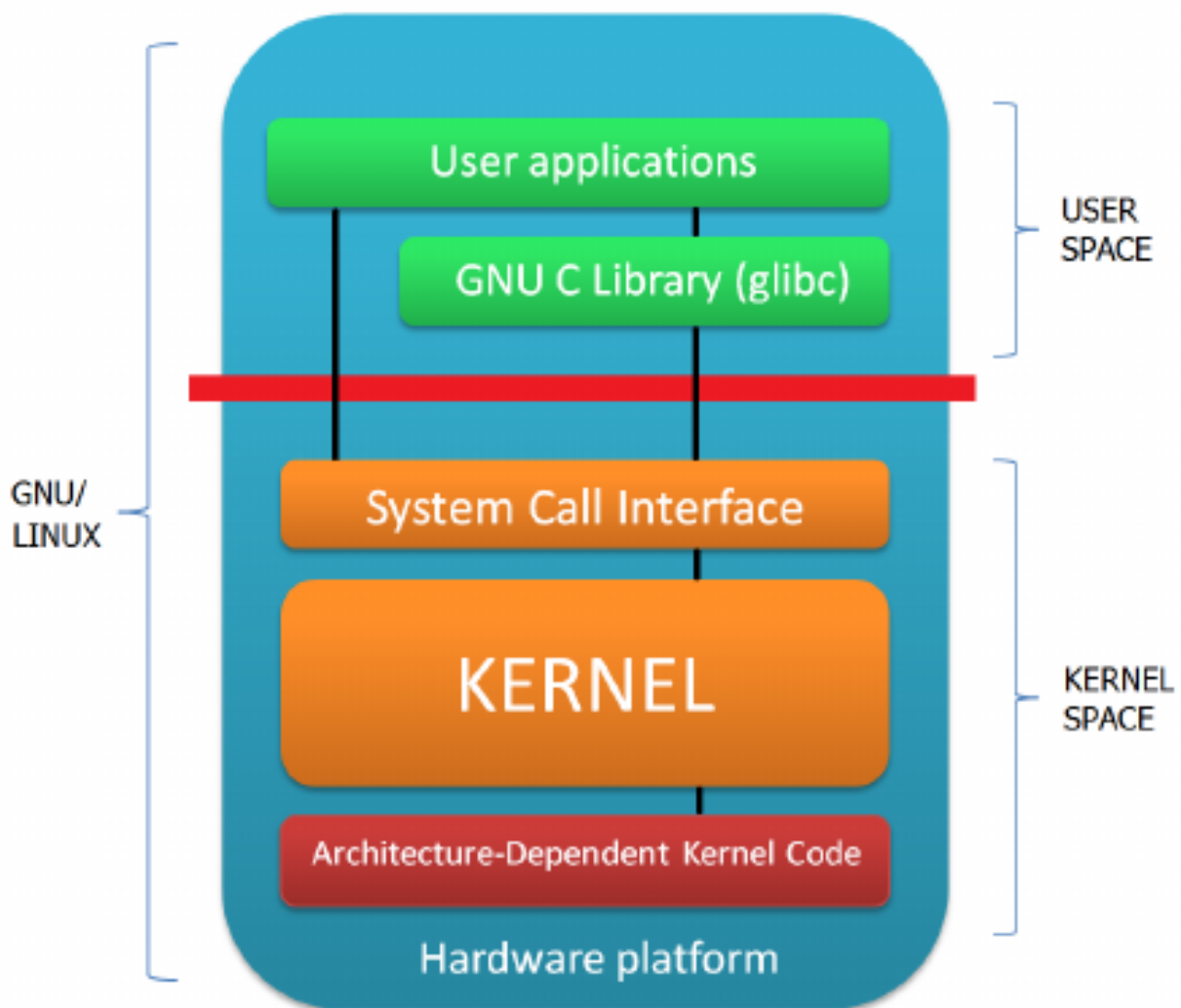
**Цель:** Сборка модуля ядра, работа с device tree.

**Полезные ссылки:**

- Linux Kernel Labs: Character device drivers.
- DiGi: Use Device Tree Overlays to Patch Your Device Tree.

**Описание:** Для начала нужно понять, что в ОС Linux выделяется два пространства: пространство ядра (Kernel Space) и пользовательское пространство (User Space).

Эти два пространства разделены между собой, и взаимодействие может происходить только при помощи специального системного интерфейса (System Call Interface).



В пространстве ядра работает, как не удивительно, само ядро ОС, а также его модули (которые могут быть встроены в основное ядро, или могут быть представлены файлами в rootfs с расширением ko).

Пользовательские приложения и библиотеки работают в пользовательском пространстве.

Ядро ОС Linux умеет работать с подгружаемыми модулями (если эта опция была активирована при сборке ядра). В качестве отдельных модулей могут выступать как определённые сервисы, предоставляемые ядром, так и драйверы устройств.

В Linux выделяют три типа устройств:

- символьное устройство — представлено в файловой системой, минимальный объём данных один символ (или байт), примеры: клавиша-

тура, мышь, принтер, тачскрин, экран, камера, spi устройство, i2c устройство и т.д.

- блочное устройство — представлено в файловой системе, минимальный объём это блок данных, работа ведётся путём монтирования устройства. Примеры: карты памяти, SD карты и пр.
- пакетное устройство — не представлено в файловой системе, устройство для взаимодействия пакетами, примеры: сетевой интерфейс, Wi-Fi, Bluetooth и т.д.

В данной работе, мы создадим шаблон для символьного устройства, с которым можно будет взаимодействовать через пользовательское приложение.

## 3.1

### Сборка модуля

**3.1.1** Откройте консоль комбинацией клавиш Ctrl+Alt+T

**3.1.2** Создайте рабочий каталог

---

```
# mkdir -p $BAGET/lab_03
```

---

**3.1.3** Скопируйте в него рабочие файлы

---

```
# cp -r $BAGET/support/mychar $BAGET/lab_03/
```

---

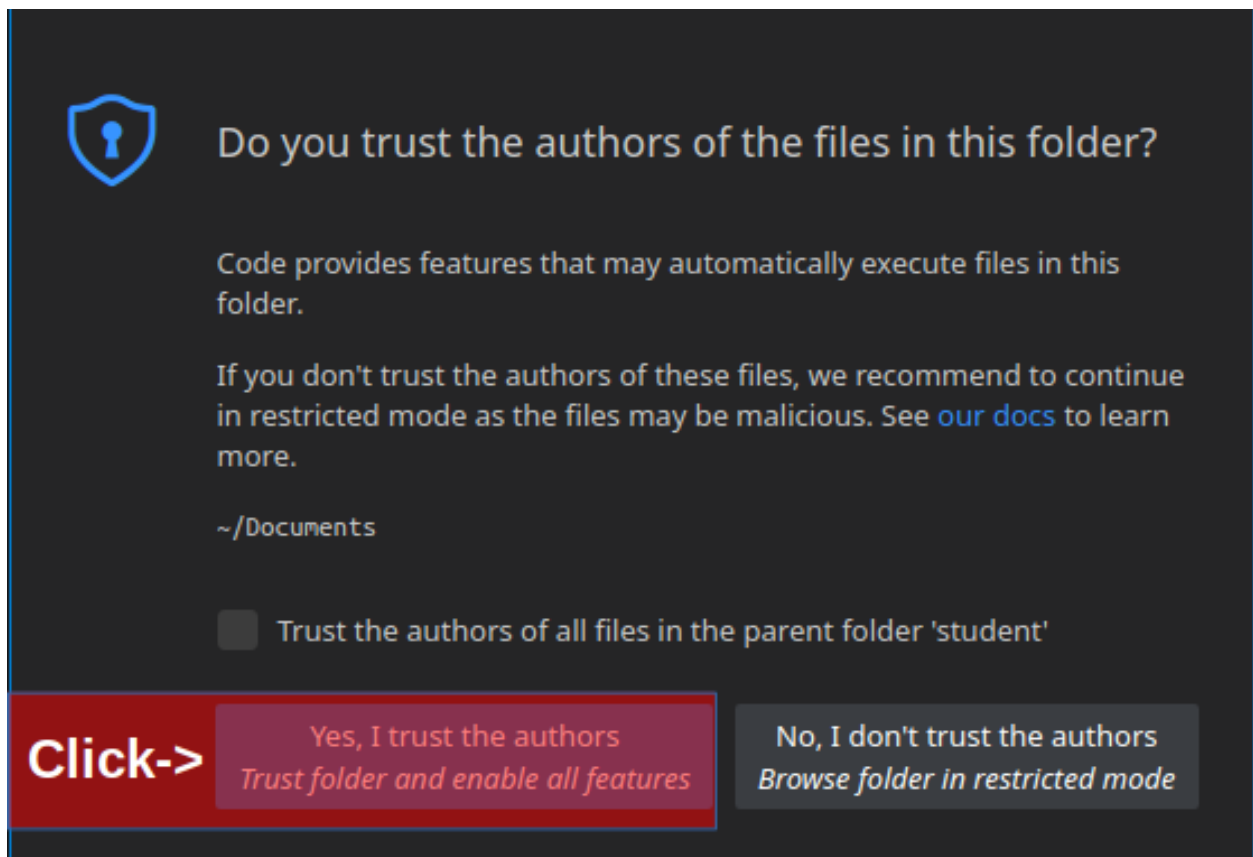
**3.1.4** Перейдите в рабочий каталог и запустите vscode

---

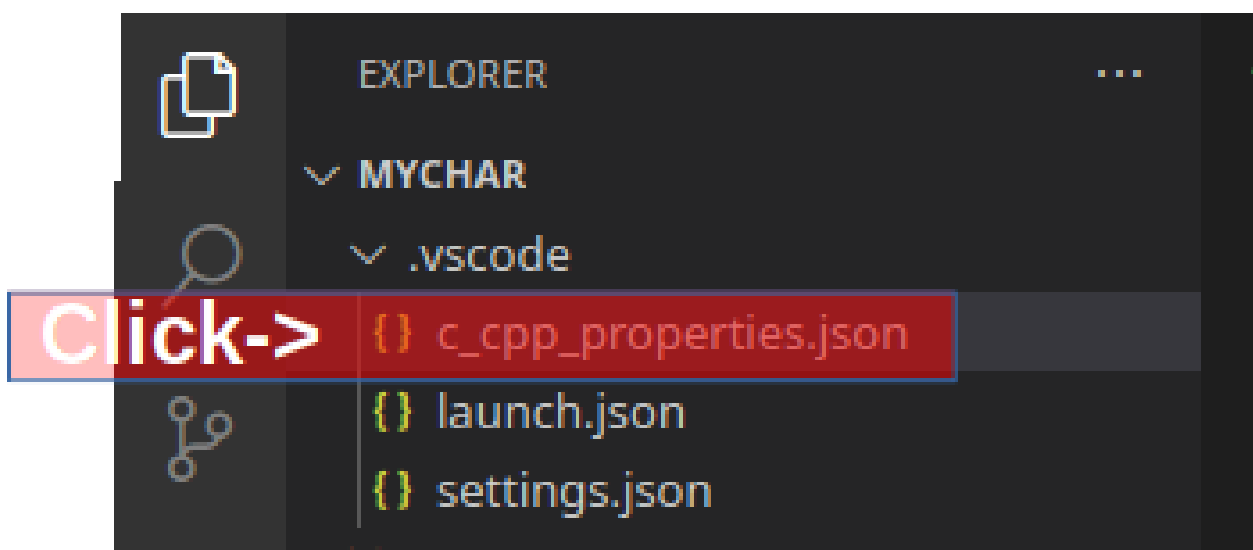
```
# cd $BAGET/lab_03/mychar; code .
```

---

**Замечание:** При первом входе, Вас могут спросить, доверяете ли вы автору, нужно нажать на кнопку Yes,



**3.1.5** Добавим путей для разрешения части include директив. Откройте .vscode -> c\_cpp\_properties.json



**3.1.6** Допишите в поле includePath следующие строки:

```
"configurations": [
  {
    "name": "linux-gcc-x64",
    "includePath": [
      "${workspaceFolder}/**",
      "/home/student/baget/support/linux-sources-4.15/include/",
      "/home/student/baget/support/linux-sources-4.15/arch/mips/include/",
      "/home/student/baget/support/linux-sources-4.15/arch/mips/include/uapi/",
      "/home/student/baget/support/linux-sources-4.15/arch/mips/include/generated/",
      "/home/student/baget/support/linux-sources-4.15/arch/mips/include/asm/mach-baget",
      "/home/student/baget/support/linux-sources-4.15/arch/mips/include/asm/mach-generic"
    ],
  }
]
```

для удобства, воспользуйтесь файлом `vscode_lines.md` в папке `support`.

Ctrl+S для сохранения

### 3.1.7 Откройте Makefile. Разберём основные моменты

`obj-m` - тут мы задаём имя объектного файла верхнего уровня (файла из которого по итогу будет собран модуль).

`chardev-y` — описываем, из каких объектных файлов будет собираться итоговый.

`KERNEL_SRC` — тут мы создаём переменную, которая будет хранить путь к заголовочным файлам ядра.

`SRC` — в этой переменной мы записываем путь к исходникам модуля, через вызов утилиты `pwd`. Таким образом, можно рабочую папку переносить куда угодно, без ручных правок. При сборке модуля, текущий путь меняется на путь `KERNEL_SRC`, поэтому путь к исходникам модуля ядра должен быть абсолютным.

`all`: — список команд выполняемых при сборке

`clean`: — список команд выполняемых при очистке

### 3.1.8 Посмотрите на содержимое файла `muchar_init.c`. Разберём основные моменты

В данном файле содержится минимальный набор кода, необходимый для сборки модуля ядра, и для его функционирования в системе. Нужно помнить, что данный код выполняется в пространстве ядра (Kernel space),

соответственно не все парадигмы и функции привычные по работе с приложениями тут будут работать, хотя большинство из них имеют тут аналоги.

Обратите внимание на структур в районе 36 строки. Эта структура считается ядром, при обращении к модулю, для установления связи между командами, отправляемыми через системный интерфейс, и функциями, которые их будут исполнять. Таким образом, достигается возможность исключить коллизий при работе с группой различных ядер. По той же причине, все структуры и функции объявлены с ключевым словом `static`.

Следующий момент, это структура объявляемая со строки 25. Параметр `.compatible` используется ядром, при сканировании дерева устройств. И при наличии совпадения, автоматически загружает модуль ядра.

**3.1.9** Посмотрите на содержимое файла `mychar_dev.c`. В этом файле описан процесс инициализации символьного устройства, а так же работа с основными операциями (открытие файла, закрытие файла, считывание и запись данных). Обратите внимание на функции работы с данными (`srisa_pdrv_read` и `srisa_pdrv_write`), для обмена данными между `user space` и `kernel space` используются специальные функции `copy_to_user` и `copy_from_user`.

**3.1.10** Если кликать мышкой на вызываемую функцию, при зажатой клавише `Ctrl`, то редактор откроет вам реализацию этой функции в отдельной вкладке (если реализация находится в другом файле). Попробуйте, так как далее Вам этот функционал понадобится, при выполнении самостоятельных работ.

**3.1.11** Перейдём в терминал (можно открыть вкладку `TERMINAL` в `vscode`, если её не видно, выберите в меню `Terminal -> New Terminal`) и соберём модуль командой

```
# ARCH=mips CROSS_COMPILE=mips64el-linux-gnuabi64- make
```

### 3.1.12 Вы должны получить следующий вывод

```
student@Kubuntu-20:~/baget/lab_03/mychar$ ARCH=mips CROSS_COMPILE=mips64el-linux-gnuabi64- make
make -C /home/student/baget/support/linux-sources-4.15 M=/home/student/baget/lab_03/mychar/
make[1]: Entering directory '/home/student/baget/support/linux-sources-4.15'
CC [M] /home/student/baget/lab_03/mychar/mychar.o
In file included from /home/student/baget/lab_03/mychar/mychar.c:1:
./include/linux/module.h:137:7: warning: 'cleanup_module' specifies less restrictive attribute than its target 'srisa_pdrv_exit': 'cold' [-Wmissing-attr]
137 | void cleanup_module(void) __attribute__((alias(#exitfn)));
    |
/home/student/baget/lab_03/mychar/mychar.c:74:1: note: in expansion of macro 'module_exit'
74 | module_exit(srisa_pdrv_exit);
    |
/home/student/baget/lab_03/mychar/mychar.c:70:20: note: 'cleanup_module' target declared here
70 | static void __exit srisa_pdrv_exit(void)
    |
In file included from /home/student/baget/lab_03/mychar/mychar.c:1:
./include/linux/module.h:131:6: warning: 'init_module' specifies less restrictive attribute than its target 'srisa_pdrv_init': 'cold' [-Wmissing-attr]
131 | int init_module(void) __attribute__((alias(#initfn)));
    |
/home/student/baget/lab_03/mychar/mychar.c:64:1: note: in expansion of macro 'module_init'
64 | module_init(srisa_pdrv_init);
    |
/home/student/baget/lab_03/mychar/mychar.c:60:19: note: 'init_module' target declared here
60 | static int __init srisa_pdrv_init(void)
    |
LD [M] /home/student/baget/lab_03/mychar/chardev.o
Building modules, stage 2.
MODPOST 1 modules
LD [M] /home/student/baget/lab_03/mychar/chardev.ko
make[1]: Leaving directory '/home/student/baget/support/linux-sources-4.15'
student@Kubuntu-20:~/baget/lab_03/mychar$
```

А в рабочей папке должны появиться несколько файлов, в том числе chardev.ko

## 3.2 Настройка платы

Для работы с модулями в Linux есть ряд команд

**insmod** <path\_to\_module> - подключение модуля ядра, с явным указанием пути к файлу, при этом не проверяются зависимости (бывает что один модуль требует предварительного запуска другого модуля).

**modprobe** <module\_name> - это «умная» версия insmod, проверяет зависимости (файл modules.dep), и подгружает их при необходимости. Не требуется указывать полный путь к модулю, однако модуль должен находиться в папке /lib/modules/<kernel\_version>/

**lsmod** — выводит список активированных модулей, с указанием счётчика зависимостей (показывает сколько потоков использует данный модуль)

**rmmod** <module\_name> - выгрузить указанный модуль (указывается только имя модуля, без расширения ko)



**3.2.1** Откройте gkterm и подключитесь к плате (/dev/ttyUSB1 скорость 115200)

**3.2.2** Создадим каталог, для нормальной работы утилиты modprobe. Для этого выполните команду

---

```
$ mkdir -p /lib/modules/$(uname -r)/
```

---

**3.2.3** Создадим два пустых файла

---

```
$ touch /lib/modules/$(uname -r)/modules.order  
$ touch /lib/modules/$(uname -r)/modules.builtin
```

---

В данном случае, допустим такой ход, но в общем случае, содержимое этих файлов должно отражать реальность.

**3.2.4** Вернёмся в консоль виртуальной машины, и скопируем скомпилированный модуль на плату (проверьте, что плата подключена сетевым шнуром к рабочему ПК)

---

```
# scp $BAGET/lab_03/mychar/chardev.ko \  
netuser@192.168.100.200:/home/netuser/
```

---

пароль - usrnetuser

**3.2.5** Вернитесь в терминал платы, и скопируйте ядро

---

```
$ cp /home/netuser/chardev.ko /lib/modules/$(uname -r)/
```

---

**3.2.6** Определим зависимости для модуля, для чего выполните команду

---

```
$ depmod
```

---

Данная команда просканирует все модули, в папке /lib/modules/\$(uname -r)/ и запишет о них информацию. После чего модули могут подгружаться

командой `modprobe`.

### 3.2.7 Проверим работу нашего модуля, выполнив команду

---

```
$ modprobe chardev
```

---

Вы должны увидеть надпись

<pre>srisa_pdrv: srisa_pdrv_init</pre>
--

### 3.2.8 Просмотрим список загруженных модулей

---

```
$ lsmod
```

---

Вы должны увидеть надпись

Module	Size	Used by
chardev	4620	0

Как мы видим, никто не использует наше ядро (Used by равен 0).

**3.2.9** Выполните команду `ls /dev/*` Обратите внимание, что отсутствует файл `srisa_pdrv`, так как он создаётся, только при вызове функции `pdrv_probe`, вызываемая только при наличии совместимого периферийного модуля.

### 3.2.10 Выгрузим из памяти наше ядро

---

```
$ rmmod chardev
```

---

## 3.3 Правка дерева устройств (device-tree)

Мы успешно собрали модуль устройства, однако его сейчас никто не использует. Как говорилось выше, часто модули ядра являются драйвером,

для взаимодействия с периферией (как внутрипроцессорной, типа интерфейса spi, так и внешней). Для описания взаимодействия между ядром я устройством служит специальная структура дерево устройств (device tree). В ней описывается основная информация о устройстве, а так же могут добавлять параметры, применяемые модулем ядра, при работе с тем или иным устройством. Среди прочего, дерево устройств поддерживает функцию перезаписи, что позволяет создавать мелкие файлы, вносящие исправление в существующее описание, или дополняющее его, в соответствии с требованиями конкретного проекта.

**3.3.1** Создадим рабочую папку в виртуальной машине для дополнения дерева устройств и перейдём в неё

---

```
# mkdir $BAGET/lab_03/devtree
# cd $BAGET/lab_03/devtree
```

---

**3.3.2** Создадим файл и откроем его для редактирования

---

```
# kate of_mychar.dts
```

---

**3.3.3** Запишите в файл следующие строки

```
/dts-v1/;

/ {
    fragment@0 {
        target-path = "/";
        __overlay__ {

            pdrv1 {
                compatible = "srisa,platform-device";
                status = "okay";
            };
        };
    };
};
```

```
};
```

### 3.3.4 Сохраните Ctrl+S и закройте редактор Ctrl+Q

Мы создали дополнение, которое описывает виртуальное устройство, совместимое с драйвером, который мы создали ранее (обратите внимание на поле `compatible` в написанном фрагменте, и сравните его значение со значением одноименного параметра в исходном коде модуля).

### 3.3.5 Скомпилируем дополнение к дереву устройств, для этого выполним следующую команду

```
# dtc -@ -O dtb -o ./of_mychard.dtb ./of_mychard.dts
```

Если у Вас есть уже скомпилированное дерево устройств, Вы можете вернуть его в «человеческий вид» командой

```
# dtc --symbol -O dts -o ./out.dts ./source.dtb
```

### 3.3.6 Скопируйте полученный файл на плату

```
# scp $BAGET/lab_03/devtree/of_mychard.dtb \
netuser@192.168.100.200:/home/netuser/
```

### 3.3.7 Перейдите в терминал платы, и переместите скопированный файл в папку barebox

```
$ mv /home/netuser/of_mychard.dtb /barebox/
```

### 3.3.8 Отредактируйте скрипт barebox.sh

```
$ nano /barebox/barebox.sh
```

и впишите в него, после строки `DTB=k5500vk018_rbm.dtb`

```
fdt_apply -i $DTB -l of_mychar.dtb -o /dtb && DTB=/dtb
```

Ctrl+S, Ctrl+X

### 3.3.9 Перезагрузите плату командой reboot

## 3.4 Проверка работы модуля

**3.4.1** Войдите как пользователь root, и выполните команду `lsmod`. Модуль должен быть уже загружен, так как ядро при запуске должно было обнаружить в дереве устройств, совместимое с нашим модулем.

### 3.4.2 Посмотрим последние строки лога загрузки ядра

---

```
$ dmesg | tail
```

---

Вы должны увидеть следующие строки в выводе

```
...
[ 27.263923] srisa_pdrv: srisa_pdrv_init
[ 27.266836] srisa_pdrv: pdrv_probe
...
```

Строку с `init` мы видели ранее, но теперь после неё есть строка `srisa_pdrv: pdrv_probe`, которая говорит нам, что наш модуль успешно связался с устройством описанным нами в дополнении к дереву устройств.

**3.4.3** Выполните команду `ls /dev/*` и обратите внимание, что должен появиться файл `srisa_pdrv`.

### 3.4.4 Отправим данные в `srisa_pdrv`, и проверьте ответ

---

```
$ echo 1234 > /dev/srisa_pdrv
```

---

```
srisa_pdrv: srisa_pdrv_open
srisa_pdrv: srisa_pdrv_ioctl
srisa_pdrv: srisa_pdrv_write: Get 5 bytes
srisa_pdrv: srisa_pdrv_release
```

Замечание: принято пять байт, так как последний байт это байт перевода каретки (`\r`).

**3.4.5**      Посмотрим на скорость записи данных, для этого выполните команду:

---

```
$ dd count=1 bs=1M if=/dev/zero of=/dev/srisa_pdrv
```

---

Команда `dd` может часто встречаться Вам при активной работе с Linux. Она обеспечивает считывание, или копирование данных из одного файла в другой. В данном случае, мы указываем, что ходим один раз (`count=1`) отправить 1 мегабайт (`bs=1M`) из файла `/dev/zero` (генератор 0) в наш драйвер. Чем больше мы отправим данных, тем точнее получим значение скорости, однако из-за особенностей реализации считывания в нашем драйвере, большой объём данных может привести к печальным последствиям, так что увлекаться не стоит. Результат должен быть таким

```
srisa_pdrv: srisa_pdrv_open
srisa_pdrv: srisa_pdrv_write: Get 1048576 bytes
srisa_pdrv: srisa_pdrv_release
1+0 records in
1+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.0318856 s, 32.9 MB/s
```

Последняя строка показывает сколько данных было передано, и на какой скорости. Результат может меняться от каждого запуска, но порядок величин должен быть одинаковый.

**3.4.6** Попробуем считать данные из драйвера, для этого выполним следующую команду:

---

```
$ dd count=1 bs=10 if=/dev/srisa_pdrv
```

---

Тут мы опять используем команду `dd`, но на этот раз не указываем выходной файл, поэтому сможем увидеть вывод в консоли

```
srisa_pdrv: srisa_pdrv_open
srisa_pdrv: srisa_pdrv_read: Send 10 bytes
ABCDEFGHIsrisa_pdrv: srisa_pdrv_release
1+0 records in
1+0 records out
10 bytes copied, 0.0131135 s, 0.8 kB/s
```

Как видите, в выводе мы увидели набор букв, который был сгенерирован нашим драйвером.

**3.4.7** Измерим скорость чтения данных, но на этот раз перенаправим вывод в файл:

---

```
$ dd count=1 bs=1M if=/dev/srisa_pdrv of=/root/drv.out
```

---

```
srisa_pdrv: srisa_pdrv_open
srisa_pdrv: srisa_pdrv_read: Send 1048576 bytes
srisa_pdrv: srisa_pdrv_release
1+0 records in
1+0 records out
1048576 bytes (1.0 MB, 1.0 MiB) copied, 0.140926 s, 7.4 MB/s
```

Скорость на чтение меньше, чем на запись, так как драйвер в начале заполняет свой внутренний массив, и затем отправляет данные. Так же тормозит и запись файла. Проверьте это, заменив файл назначения на `/dev/null`. Для просмотра вывода выполните команду `cat /root/drv.out` и немного подождём, пока через терминал будет передан 1 мегабайт алфавита. Вывод можно прервать комбинацией `Ctrl+C`

### 3.4.8 Выгрузим ядро

---

```
$ rmmod chardev
```

---

И снова мы видим, что есть два вывода, один от модуля, который видели ранее, второй сообщает нам, что была выполнена функция по отключению от устройства.

**3.4.9** Введите команду `poweroff`, дождитесь, пока не появится надпись: `reboot: System halt`, после чего отключите USB кабель от ПК.

## 3.5 Задание для закрепления

несите правку в модуль ядра, так, что бы командой считывания можно было получить данные, записанные через команду записи. Другими словами, пользователь два раза записал `Hello` после чего запросил на считывание 1Мбайт, в ответ он должен получить `HelloHello`.

Замечание: тут нам понадобится работа со структурой `kfifo` (в коде переменная `data_fifo`). В коде уже вставлены части по её инициализации (`mychar_dev.c`), всё что Вам остаётся, вписать в нужные места вызовы функций для считывания и записи данных. При этом функция `copy_from_user` и `copy_to_user` уже не нужны.

Проверять работу лучше так, запись делать с помощью команды `echo`, а считывание через команду `dd`.

Пример для записи данных в драйвере.

```
static ssize_t fifo_write(struct file *file, const char __user *buf,
    size_t count, loff_t *ppos)
{
    int ret;
    unsigned int copied;

    ret = kfifo_from_user(&test, buf, count, &copied);
```



```
    return ret ? ret : copied;
}
```

Пример для считывания данных из драйвера.

```
static ssize_t fifo_read(struct file *file, char __user *buf, size_t
    count, loff_t *ppos)
{
    int ret;
    unsigned int copied;

    ret = kfifo_to_user(&test, buf, count, &copied);

    return ret ? ret : copied;
}
```

\* Модифицировать код таким образом, что бы можно было независимо обрабатывать два устройства (для проверки добавьте в device tree ещё одно устройство, pdrv2)