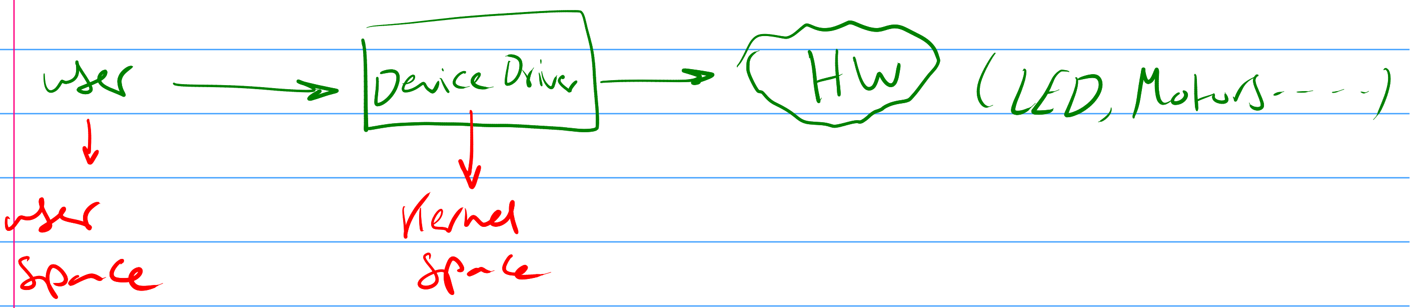
*Linux Device Drivers:

→ What is a device driver?

- software to handle your hardware



→ Types of device drivers

1) Character devices → Serial ports, LEDs.....
2) Block devices → USB, Memory...
3) Network devices → Wifi.....

→ Static & Dynamic

↳ Static in tree → in linux tree
↳ out-of-tree → By vendors in their repos

* Some Drivers is loaded during the booting time
& some Drivers is loaded dynamically (user spaces)

Bootloader
↓
Kernel
(loads the built in drivers)
↓
Init process
(load Kernel modules)

Pseudo-devices

→ files in the /dev, Acting as a bridge between OS and HW

→ No policy inthe Kernel !
- ↳ user space responsible for setting policies
- ↳ udev is responsible for loading kernel modules, such as plugging in the USB

Linux Kernel Modules :

→ we can build simple kernel modules using Kbuild

→ steps :

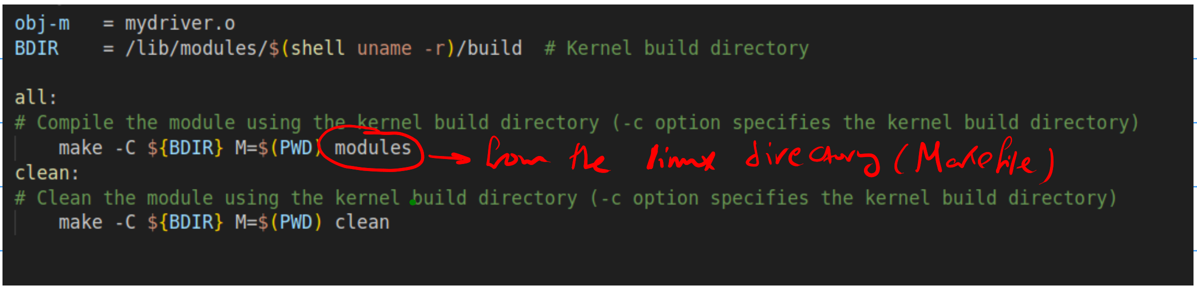
1- Make the source code (No user space headers)

```
#include <linux/module.h>
#include <linux/init.h>
   ⋮
```

2. Make a Makefile

obj-m = module-name.o
- ↳ S (static module)
- ↳ n (No compile)
- ↳ m (dynamic module)

Ex:



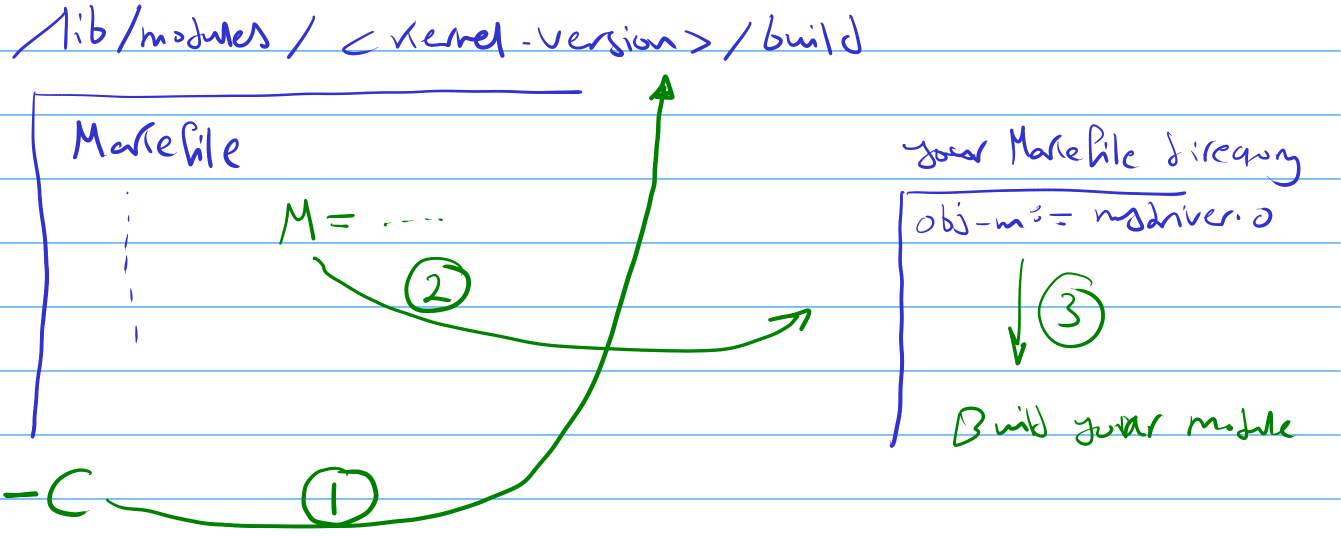
```
obj-m   = mydriver.o
BDIR    = /lib/modules/$(shell uname -r)/build  # Kernel build directory

all:
# Compile the module using the kernel build directory (-c option specifies the kernel build directory)
    make -C ${BDIR} M=$(PWD) modules
clean:
# Clean the module using the kernel build directory (-c option specifies the kernel build directory)
    make -C ${BDIR} M=$(PWD) clean
```

modules → from the linux directory (Makefile)

-C → provide the kernel source directory where the large makefile exists

M → make it in the current directory (CD)



3- running your module

1- use insmod to insert your module
2- use lsmod to list all loaded modules
3- use rmmod to remove the module

* lsmod reads /proc/modules

* Finding linux kernel drivers:

Ex: MAX7313 GPIO expander on I2C

1- git grep -i max7313 (in linux source code)
↳ drivers/gpio/gpio-pca953x.c

2- read the drivers/gpio/Makefile to learn which kernel config option enables this driver. (grep for gpio-pca953x)
↳ obj-$(CONFIG_GPIO_PCA953X)
↳ enable this in the kernel

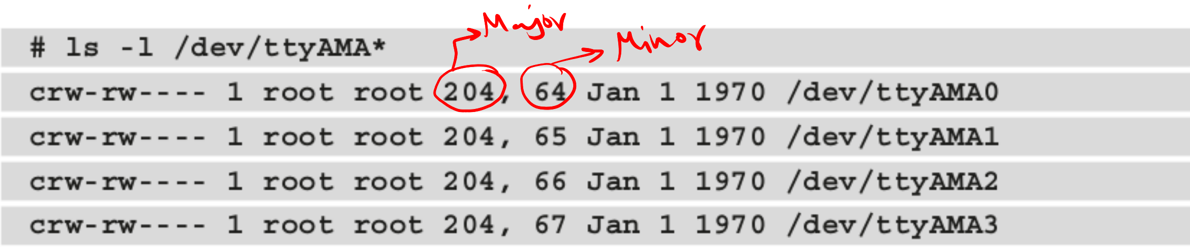
#Device Drivers info:

→ The kernel identify the devices by a triplet of info

1- type (Character or block)

2- Major (typically the Category of device)

3- Minor (typically the ID of the device)



Major

Minor

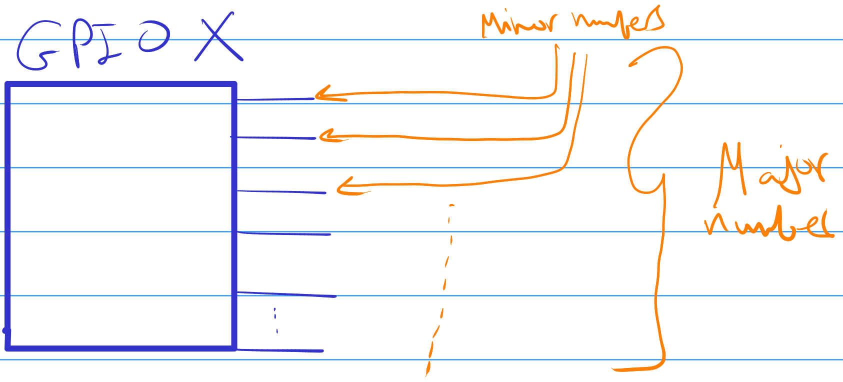
```
# ls -l /dev/ttyAMA*
crw-rw---- 1 root root 204, 64 Jan 1 1970 /dev/ttyAMA0
crw-rw---- 1 root root 204, 65 Jan 1 1970 /dev/ttyAMA1
crw-rw---- 1 root root 204, 66 Jan 1 1970 /dev/ttyAMA2
crw-rw---- 1 root root 204, 67 Jan 1 1970 /dev/ttyAMA3
```

Note: Character devices are identified by a special file Called device node

↳ this will map to a device driver using major and minor numbers

Major nuber → tell which driver to be used

minor number → tell which interface is being accessed



Internal Representation of Device Numbers

→ dev_t (linux/types.h)

- → holds the Major & Minor numbers (32-bit)
- → 12 bit for Major
- → 20 bit for Minor

MAJOR(dev_t dev); → get the Major number
MINOR(dev_t dev); → get the Minor number
MKDEV(int major, int minor); → Create dev with maj & min

Allocating and freeing Device numbers

→ int register_chrdev_region(dev_t first, unsigned int count, char *name);

→ This is not used any more as it need from you to know the Major number & Minor numbers

→ output

→ int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name);

→ here it dynamically allocate dev numbers

→ dev is output

→ void unregister_chrdev_region(dev_t first, unsigned int count);

→ you must free the allocated dev numbers after no longer needing it

→ use it in the exit function

→ Now we only allocated few numbers

→ Kernel don't know what we will be doing with it

→ you will need to connect the driver to the functionality it will use, user space cannot do any thing with it

* → you can read the char devices using (/proc/devices)

Important data structure

1- file_operations
2- file
3- inode

1- file_operations:

→ by using file_operations struct, we can add functionality to our driver.

→ in <linux/fs.h>

→ Contains function pointers

→ implements the sysCalls (open, read, ...)

※ important functions:

1- owner → who owns the struct

2- llseek → change the current read/write position
if NULL it will result in an unpredictable behavior

3- read → retrieve data from device

- .NULL will make it unreadable
- .returns the number of bytes retrived

4- aio_read → Async read

5- write → send data to the device

- .return the number of bytes written

6- aio_write

7- readdir → only for directories

8- open → will notify the driver

2- file struct :

refer to Ch 3

3- inode struct :

* Char device registeration :

1) → struct cdev * my_cdev ;

2) → cdev_init (struct cdev *cdev , struct file_operations *fops);

- .adds the fops

3) → my_cdev. owner = THIS_MODULE;

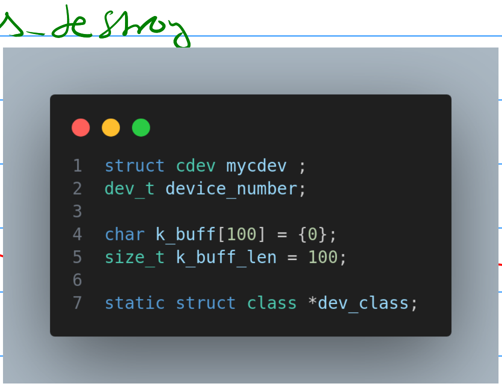
4) → cdev_add (cdev , dev_t num, count) ;

·Tells the kernel about it

5) → cdev_del (cdev) ;

6) → class_create ("file_name") ;

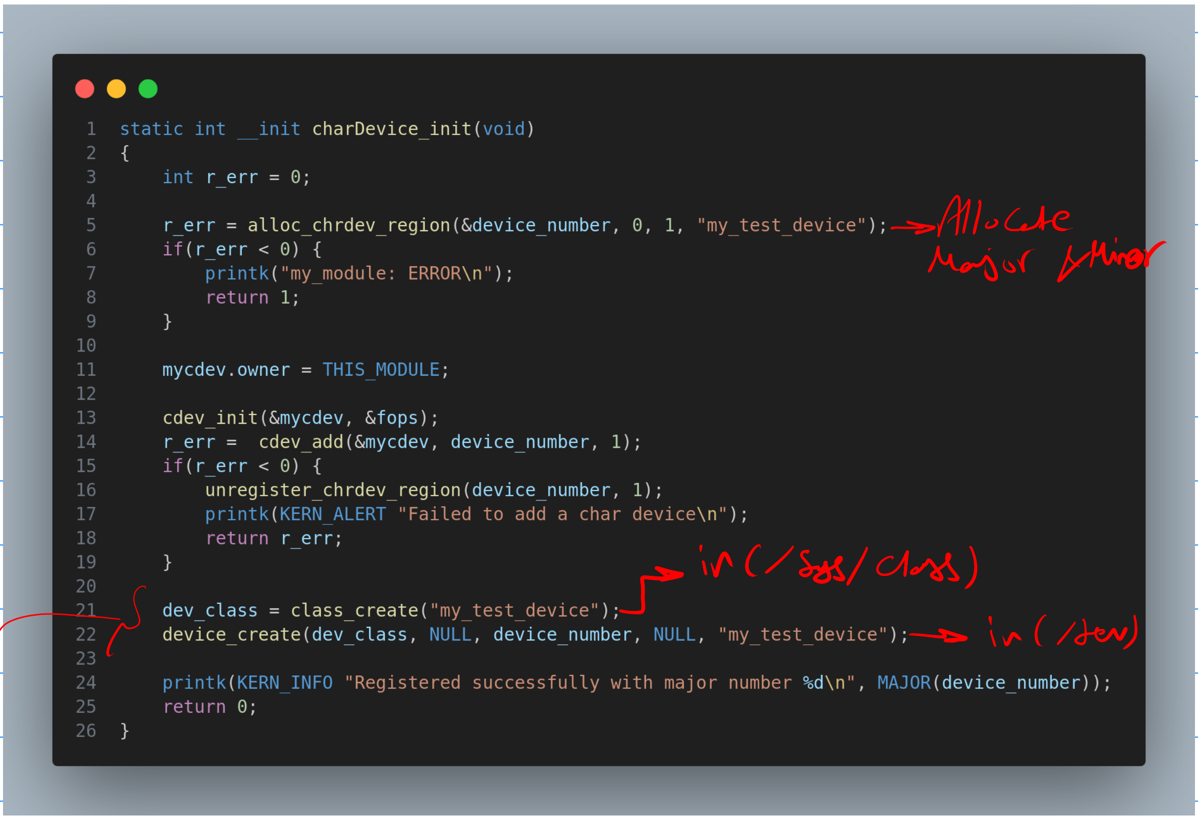
7) → class_destroy



```
struct cdev mycdev ;
dev_t device_number;

char k_buff[100] = {0};
size_t k_buff_len = 100;

static struct class *dev_class;
```



```
static int __init charDevice_init(void)
{
    int r_err = 0;

    r_err = alloc_chrdev_region(&device_number, 0, 1, "my_test_device");
    if(r_err < 0) {
        printk("my_module: ERROR\n");
        return 1;
    }

    mycdev.owner = THIS_MODULE;

    cdev_init(&mycdev, &fops);
    r_err =  cdev_add(&mycdev, device_number, 1);
    if(r_err < 0) {
        unregister_chrdev_region(device_number, 1);
        printk(KERN_ALERT "Failed to add a char device\n");
        return r_err;
    }

    dev_class = class_create("my_test_device");
    device_create(dev_class, NULL, device_number, NULL, "my_test_device");

    printk(KERN_INFO "Registered successfully with major number %d\n", MAJOR(device_number));
    return 0;
}
```

→ Allocate Major & Minor

→ in (/sys/class)

→ in (/dev)

→ Allows the user to interact with your device