**Character Device Drivers**

**Why character device drivers?** [**https://tldp.org/LDP/lkmpg/2.4/html/x579.html**](https://tldp.org/LDP/lkmpg/2.4/html/x579.html)

Whenever we plug a hardware device into system, it need a ‘translator’ and ‘space’ to store and transfer data in between -> Device drivers

‘Character’ describes the way they transfer data – by sending and receiving every single character. It’s good for unclear/unpredictable size of data.

For known size of data transfer, we can use block device driver.

**The file\_operations Structure**

The file\_operations structure is defined in linux/fs.h, and holds pointers to functions defined by the driver that perform various operations on the device. Each field of the structure corresponds to the address of some function defined by the driver to handle a requested operation.

For example, every character driver needs to define a function that reads from the device. The file\_operations structure holds the address of the module's function that performs that operation.

struct file\_operations {

struct module \*owner;

loff\_t (\*llseek) (struct file \*, loff\_t, int);

ssize\_t (\*read) (struct file \*, char \*, size\_t, loff\_t \*);

ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);

int (\*readdir) (struct file \*, void \*, filldir\_t);

unsigned int (\*poll) (struct file \*, struct poll\_table\_struct \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned int, unsigned long);

int (\*mmap) (struct file \*, struct vm\_area\_struct \*);

int (\*open) (struct inode \*, struct file \*);

int (\*flush) (struct file \*);

int (\*release) (struct inode \*, struct file \*);

int (\*fsync) (struct file \*, struct dentry \*, int datasync);

int (\*fasync) (int, struct file \*, int);

int (\*lock) (struct file \*, int, struct file\_lock \*);

ssize\_t (\*readv) (struct file \*, const struct iovec \*, unsigned long,

loff\_t \*);

ssize\_t (\*writev) (struct file \*, const struct iovec \*, unsigned long,

loff\_t \*);

};

But we can use a simpler way to express :

struct file\_operations fops = {

.read = device\_read,

.write = device\_write,

.open = device\_open,

.release = device\_release

};

**The file structure**

Each device is represented in the kernel by a file structure, which is defined in linux/fs.h. Be aware that a file is a kernel level structure and never appears in a user space program. It's not the same thing as a FILE, which is defined by glibc and would never appear in a kernel space function. Also, its name is a bit misleading; it represents an abstract open `file', not a file on a disk, which is represented by a structure named inode. (explaination: for example, when we put an USB into device, a virtual folder appear on our workspace despite it doesn’t really exist on our system)

**Registering A Device**

char devices are accessed through device files, usually located in /dev. The major number tells you which driver handles which device file. The minor number is used only by the driver itself to differentiate which device it's operating on, just in case the driver handles more than one device.

For example, let's consider a hypothetical scenario where we have two character device drivers: **driver\_A** and **driver\_B**. Both drivers are responsible for handling device files in the **/dev** directory.

* **driver\_A** is assigned major number **10**.
* **driver\_B** is assigned major number **20**.

Now, let's say both drivers are capable of managing multiple devices. To differentiate between the devices managed by each driver, the minor number is utilized.

For instance:

* **driver\_A** operates on two devices: **device\_A1** and **device\_A2**.
* **driver\_B** operates on three devices: **device\_B1**, **device\_B2**, and **device\_B3**.

The device files for each device will be created in the **/dev** directory with the appropriate major and minor numbers:

* **/dev/device\_A1** will have major number **10** and a unique minor number assigned by **driver\_A**.
* **/dev/device\_A2** will also have major number **10** but with a different minor number assigned by **driver\_A**.
* **/dev/device\_B1** will have major number **20** and a unique minor number assigned by **driver\_B**.
* **/dev/device\_B2** will have major number **20** but with a different minor number assigned by **driver\_B**.
* **/dev/device\_B3** will also have major number **20** but with a different minor number assigned by **driver\_B**.

In this way, the major number indicates which driver handles the device file, while the minor number distinguishes between individual devices managed by the same driver.

Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization. You do this by using the register\_chrdev function, defined by linux/fs.h.

int register\_chrdev(unsigned int major, const char \*name,

struct file\_operations \*fops);

where unsigned int major is the major number you want to request, const char \*name is the name of the device as it'll appear in /proc/devices and struct file\_operations \*fops is a pointer to the file\_operations table for your driver. A negative return value means the registertration failed. Note that we didn't pass the minor number to register\_chrdev. That's because the kernel doesn't care about the minor number; only our driver uses it. Without assign a major number, system cannot call driver to control device, even though we had the driver assigned.

Now the question is, how do you get a major number without hijacking one that's already in use? The easiest way would be to look through Documentation/devices.txt and pick an unused one. That's a bad way of doing things because you'll never be sure if the number you picked will be assigned later. The answer is that you can ask the kernel to assign you a dynamic major number.

If you pass a major number of 0 to register\_chrdev, the return value will be the dynamically allocated major number. The downside is that you can't make a device file in advance, since you don't know what the major number will be. There are a couple of ways to do this. First, the driver itself can print the newly assigned number and we can make the device file by hand. Second, the newly registered device will have an entry in /proc/devices, and we can either make the device file by hand or write a shell script to read the file in and make the device file. The third method is we can have our driver make the the device file using the mknod system call after a successful registration and rm during the call to cleanup\_module.

**Unregistering A Device**

Before unregistering a device, we need to make sure that all processes stop using /connecting to the device otherwise it will raise an error or jump into a ramdom memory space

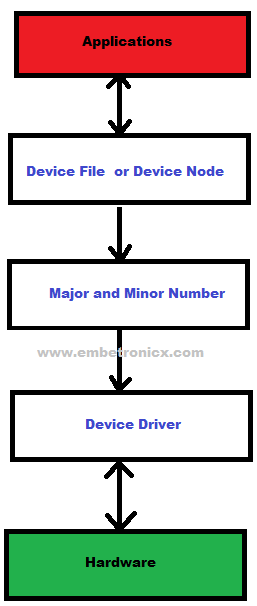
<https://embetronicx.com/tutorials/linux/device-drivers/device-file-creation-for-character-drivers/>

<https://lwn.net/Kernel/LDD3/> (class\_create and device\_create are not covered in the book).

**Linux modules and drivers**

<https://github.com/Johannes4Linux/Linux_Driver_Tutorial/tree/main>

**Simple Driver**



Note:

* Include some kernel header only
* MODULE\_LICENSE is important (some distribution only loading, GPL for free soft)
* Run sudo insmod our\_module.ko to register driver and sudo rmmod our\_module.ko to remove it. To check use lsmod | mymodule + dmesg | tail

\_\_init (module is loaded) and \_\_exit (module is removed):

We can’t use printf (there is no command line) so we use printk instead.

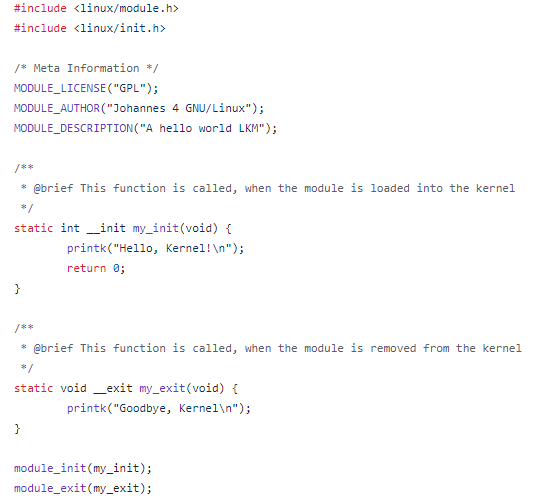


Figure : Simple Driver

* **Makefile** : make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

This command line will go to the directory containing the kernel header files and kernel build tools, using the currently running kernel version. It will then build the kernel modules from the source code in the current directory

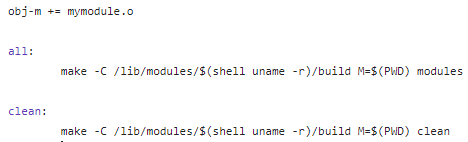


Figure :Makefile Simple Driver

**Device numbers and files**

We connect devices files with device number(major), see room for Character devices by using command : cat /proc/devices

In this section, we need to input #include <linux/fs.h> (fs stand for file system)

By using file operation struct, we can call callback function to operate printk whenever open and close file

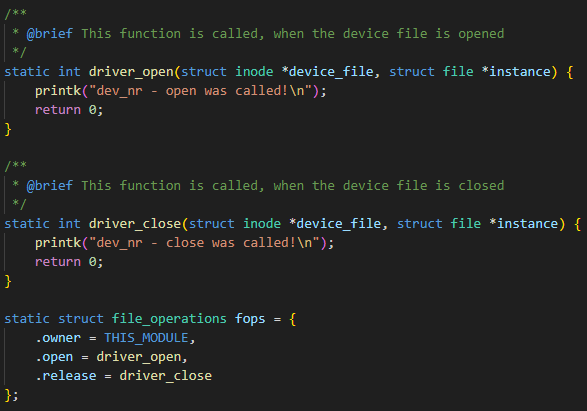


Figure : File operation callback and implement file operation

In the next part we will use int register\_chrdev(unsigned int major, const char \*name, const struct file\_operations \*fops); to register a driver file operation ([this method is register statically, we won’t use this](https://embetronicx.com/tutorials/linux/device-drivers/character-device-driver-major-number-and-minor-number/))

!!!! register\_chrdev = alloc\_chrdev\_region + cdev\_init + cdev\_add

#define MYMAJOR 64

/\*\*

 \* @brief This function is called, when the module is loaded into the kernel

 \*/

static int \_\_init ModuleInit(void) {

    int return\_value;

    printk("Hello, Kernel!\n");

    /\* register device nr. \*/

    return\_value = register\_chrdev(MYMAJOR, "my\_dev\_nr", &fops);

    if(return\_value == 0) {

        printk("dev\_nr - registered Device number Major: %d, Minor: %d\n", MYMAJOR, 0);

    }

    else if(return\_value > 0) {

        printk("dev\_nr - registered Device number Major: %d, Minor: %d\n", return\_value>>20, return\_value&0xfffff);

    }

    else {

        printk("Could not register device number!\n");

        return -1;

    }

    return 0;

}

Figure : register character device

In this code, we can see that register\_chrdev will run when ModuleInit is called -> fops will operate 3 function .owner, .open, .release

**Auto Device File creation & Read- Write-Callbacks**

In the last section, we have done driver\_open and driver\_close

In this secton, we use callback function driver\_read and driver\_write :

* Driver\_read

static ssize\_t driver\_read(struct file \*File, char \*user\_buffer, size\_t count, loff\_t \*offs)

ssize\_t return the size of character read in driver file, user\_buffer is where we send the data to, count is the amount of character and offs is offset – where we start reading

! we can’t use memcpy or things like that because it’s only fit with user space (which have different memory regions. For kernel space we use this from uacess.h

copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n)

This function will return the amount of byte that cannot be copied (0 if its done completely)

* Driver\_write

static ssize\_t driver\_write(struct file \*File, const char \*user\_buffer, size\_t count, loff\_t \*offs)

We use almost the same structure, but add const at user\_buffer because we only want to read from that

* Create var for device and class

/\* Variables for device and device class \*/

static dev\_t my\_device\_nr;

static struct class \*my\_class;

static struct cdev my\_device;

* Allocating major and minor number for device ( put 0 into major number -> auto allocate)

    /\* Allocate a device nr \*/

    if( alloc\_chrdev\_region(&my\_device\_nr, 0, 1, DRIVER\_NAME) < 0) {

        printk("Device Nr. could not be allocated!\n");

        return -1;

    }

To use a character driver, first you should register it with the system. Then you should expose it to the user space. <https://stackoverflow.com/questions/50377327/diffrences-between-cdev-add-and-device-create-function>

cdev\_init and cdev\_add functions perform the character device registration. [cdev\_add](https://elixir.bootlin.com/linux/v3.8/source/fs/char_dev.c#L472) adds the character device to the system. When cdev\_add function successfully completes, the device is live and the kernel can invoke its operations.

In order to access this device from user space, you should create a device node in /dev. You do this by creating a virtual device class using [class\_create](https://elixir.bootlin.com/linux/v4.4/source/include/linux/device.h#L517), then creating a device and registering it with sysfs using the [device\_create](https://elixir.bootlin.com/linux/v3.9/source/drivers/base/core.c#L1607) function. device\_create will create a device file in /dev.

Inconclusion:

* First allocate major and minor number dynamically -> store in dev\_t variable (1)
* Use fops(file operation with open, close, read, write function) + cdev\_init to put all that func into cdev variable(2)
* Cdev\_add +cdev var (2) + dev\_t var(1) -> register a device with major and minor number
* Name of device driver declared in source.c (check using cat /proc/devices), name of module is name\_of\_module.ko (check using lsmod)
* To see last/first 10 line of dmesg -> dmesg | tail/head –n 10

Task1, we need:

1. **Implement file operation in driver**: investigate struct file\_operations and how to build function for open, read, write, and release.
2. **Register file operation**: understand how to use register\_chrdev or cdev\_add to register a file into VFS (Virtual File System).
3. **Init function for driver**:
   * **Allocating device number**: get used to alloc\_chrdev\_region or register\_chrdev\_region.
   * **Create device file**: learn how to create device in /dev using device\_create in userspace.
4. **Cleanup function for driver**:
   * **Destroy device file**: know how to destroy device using device\_destroy.
   * **Unregister device number**: investigate unregister\_chrdev\_region.

! manual add and delete device file:

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

To run this example start with this command to register a driver with major number into 'cat /proc/devices': (allocating device number included)

sudo insmod dev\_nr.ko

~~~~~

https://embetronicx.com/tutorials/linux/device-drivers/device-file-creation-for-character-drivers/

Then to create a device file manually:

sudo mknod /dev/mydevice c 64 0

~~~~~

to remove device file manually :

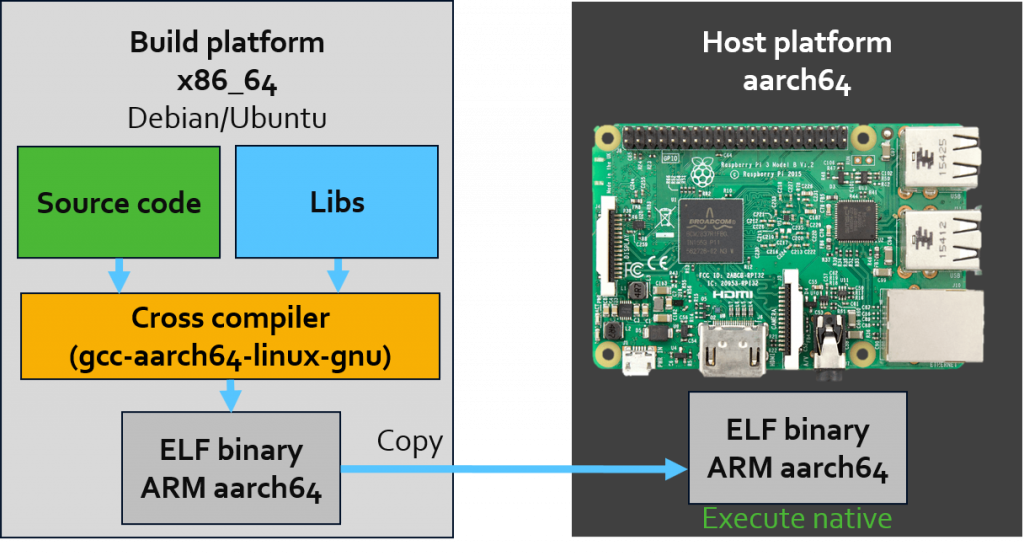
sudo rm /dev/mydevice

~~~~~

Unregister device number: (included in the driver)

sudo rmmod dev\_nr.ko~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

**Cross compiler for raspberry pi**

  
<https://embear.ch/blog/compiling-a-kernel-module>

<https://jensd.be/1126/linux/cross-compiling-for-arm-or-aarch64-on-debian-or-ubuntu>

<https://stackoverflow.com/questions/61527102/cross-compilation-kernel-and-kernel-modules-problems>

!need to point to same/almost the same linux version with raspberry pi to make

**Download the kernel**

Downloading the right kernel can be a little bit tricky. It is often not clear from where the kernel comes from. The SoC manufacturer as well as the SoM/CoM provider do some downstream modifications and provide their version in their own repository. Here a list of some kernel repositories:

* TI Sitara: [**https://git.ti.com/cgit/ti-linux-kernel/ti-linux-kernel/**](https://git.ti.com/cgit/ti-linux-kernel/ti-linux-kernel/)
* NXP iMX: [**https://source.codeaurora.org/external/imx/linux-imx/**](https://source.codeaurora.org/external/imx/linux-imx/)
* Toradex: [**http://git.toradex.com/cgit/linux-toradex.git/**](http://git.toradex.com/cgit/linux-toradex.git/)
* NXP S32G: [**https://source.codeaurora.org/external/autobsps32/linux/**](https://source.codeaurora.org/external/autobsps32/linux/)
* Raspberry Pi: [**https://github.com/raspberrypi/linux**](https://github.com/raspberrypi/linux)

For more info, follow the [link](https://embear.ch/blog/compiling-a-kernel-module) , <https://github.com/raspberrypi/linux/blob/rpi-5.10.y/arch/arm/boot/dts/bcm2710-rpi-3-b-plus.dts>

syntax

make ARCH=arm64/mips CROSS\_COMPILE=your-cross-toolchain-prefix- -C /path/to/linux M=$(pwd) modules

Additional things:

What is the difference between ioctl(), unlocked\_ioctl() and compat\_ioctl()?

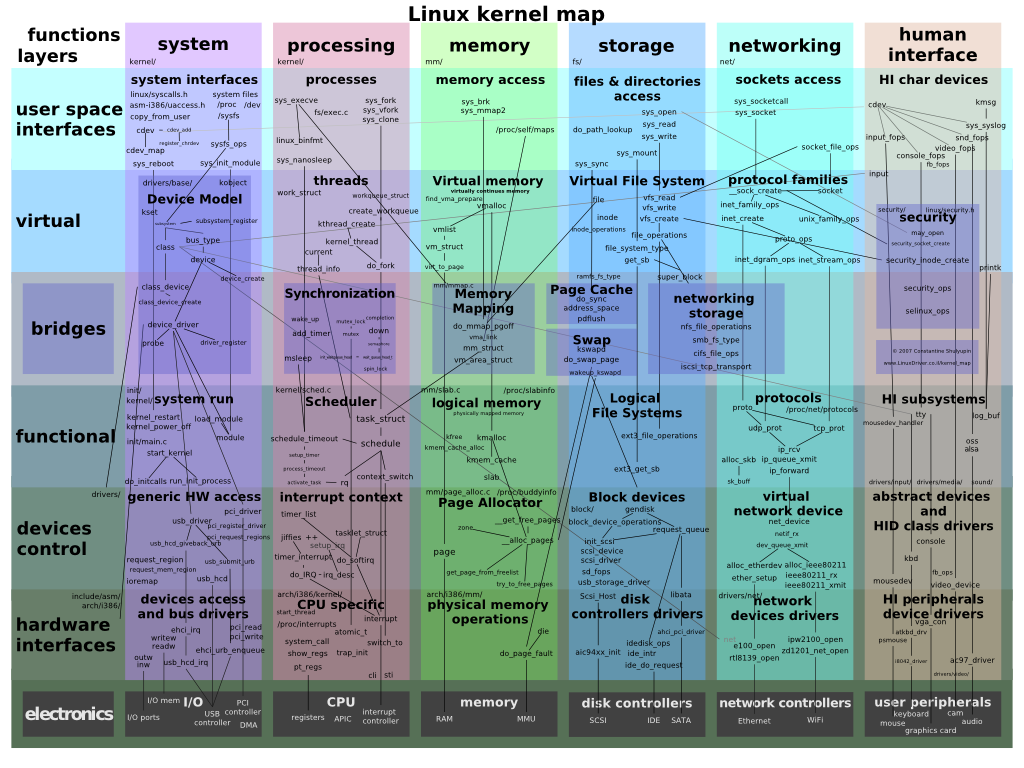
From [The new way of ioctl()](http://lwn.net/Articles/119652/) by [Jonathan Corbet](http://lwn.net/Kernel/LDD3/):

ioctl() is one of the remaining parts of the kernel which runs under the Big Kernel Lock (BKL). In the past, the usage of the BKL has made it possible for long-running ioctl() methods to create long latencies for unrelated processes.

Follows an explanation of the patch that introduced unlocked\_ioctl and compat\_ioctl into 2.6.11. The [removal of the ioctl field](http://lwn.net/Articles/394724/) happened a lot later, in 2.6.36.

Explanation: When ioctl was executed, it took the [Big Kernel Lock](http://en.wikipedia.org/wiki/Giant_lock) (BKL), so nothing else could execute at the same time. This is very bad on a multiprocessor machine, so there was a big effort to get rid of the BKL. First, unlocked\_ioctl was introduced. It lets each driver writer choose what lock to use instead. This can be difficult, so there was a period of transition during which old drivers still worked (using ioctl) but new drivers could use the improved interface (unlocked\_ioctl). Eventually all drivers were converted and ioctl could be removed.

compat\_ioctl is actually unrelated, even though it was added at the same time. Its purpose is to allow 32-bit userland programs to make ioctl calls on a 64-bit kernel. The meaning of the last argument to ioctl depends on the driver, so there is no way to do a driver-independent conversion.



<https://stackoverflow.com/questions/27703195/internal-linux-kernel-interfaces>