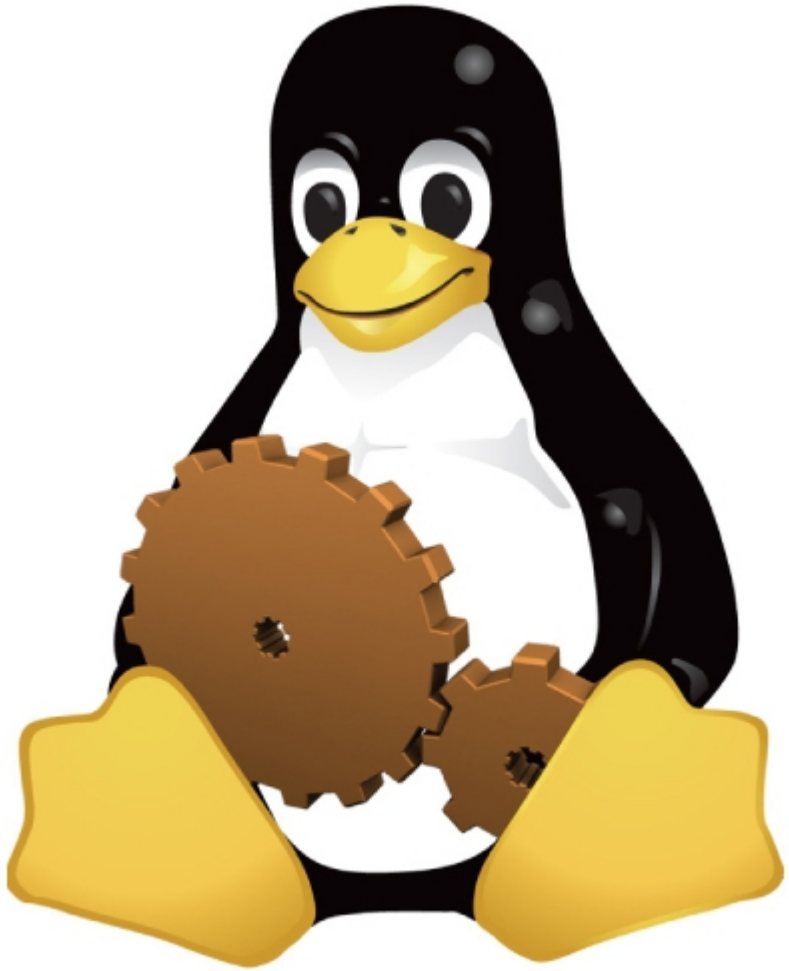
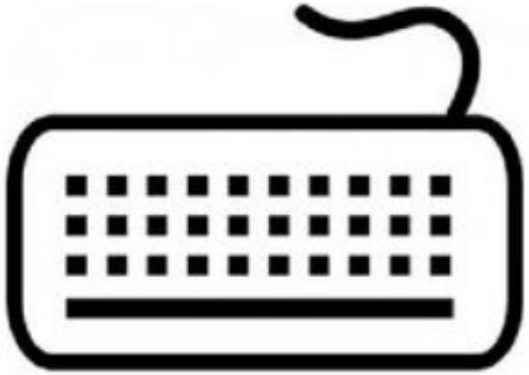
Linux Device Driver: Char Device Driver

Andy Lee

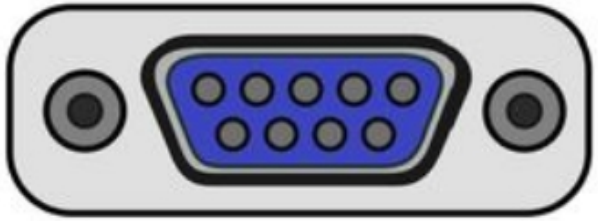
Table of Content



Character device



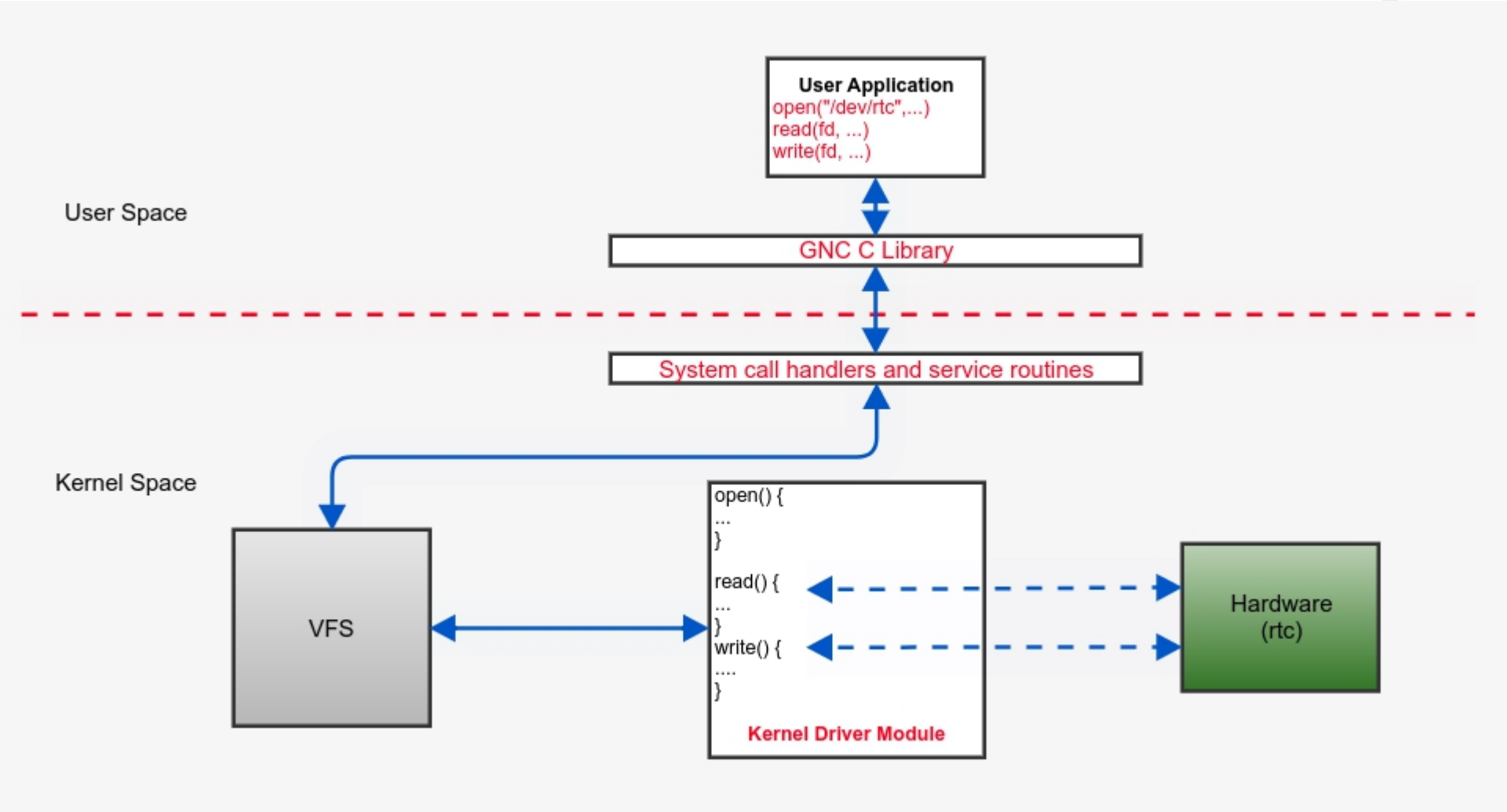
keyboard



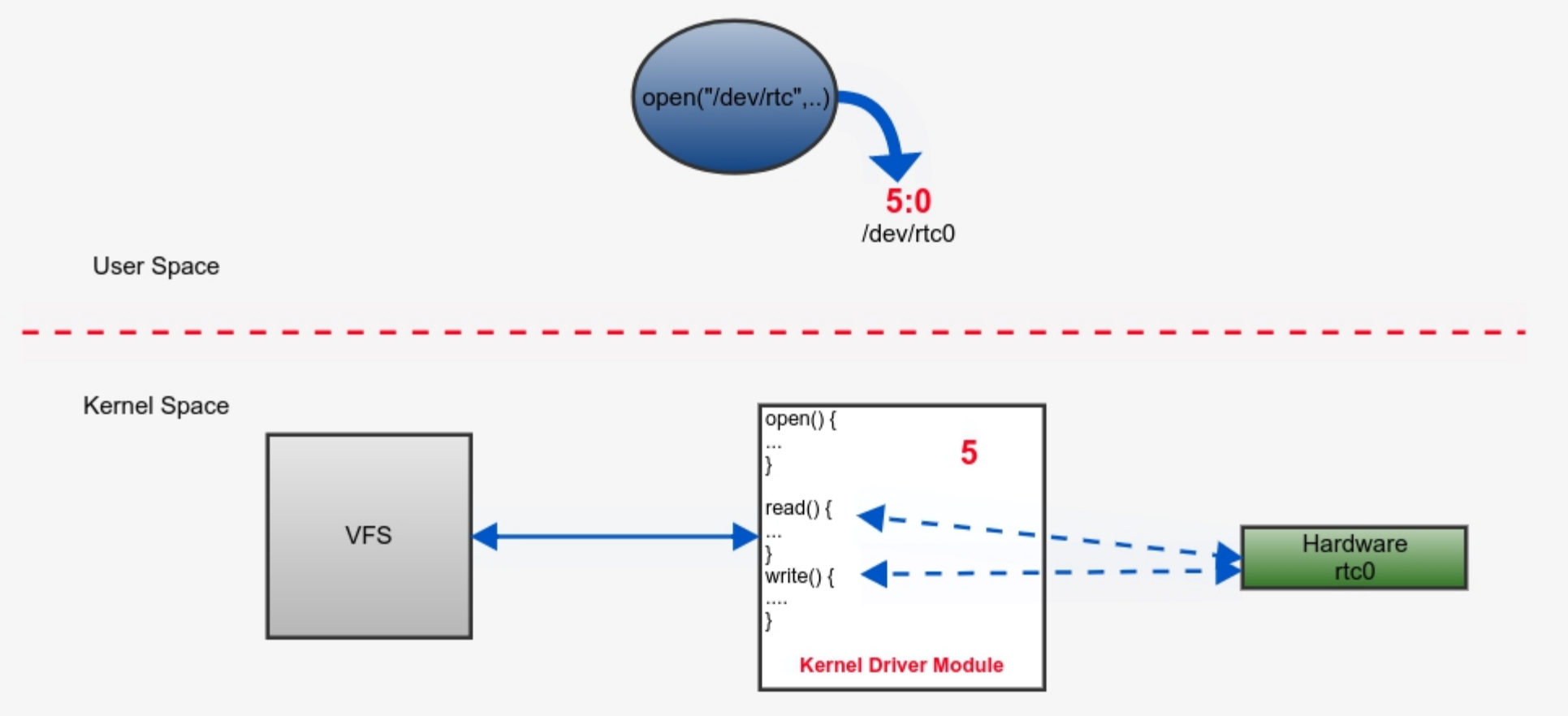
serial port

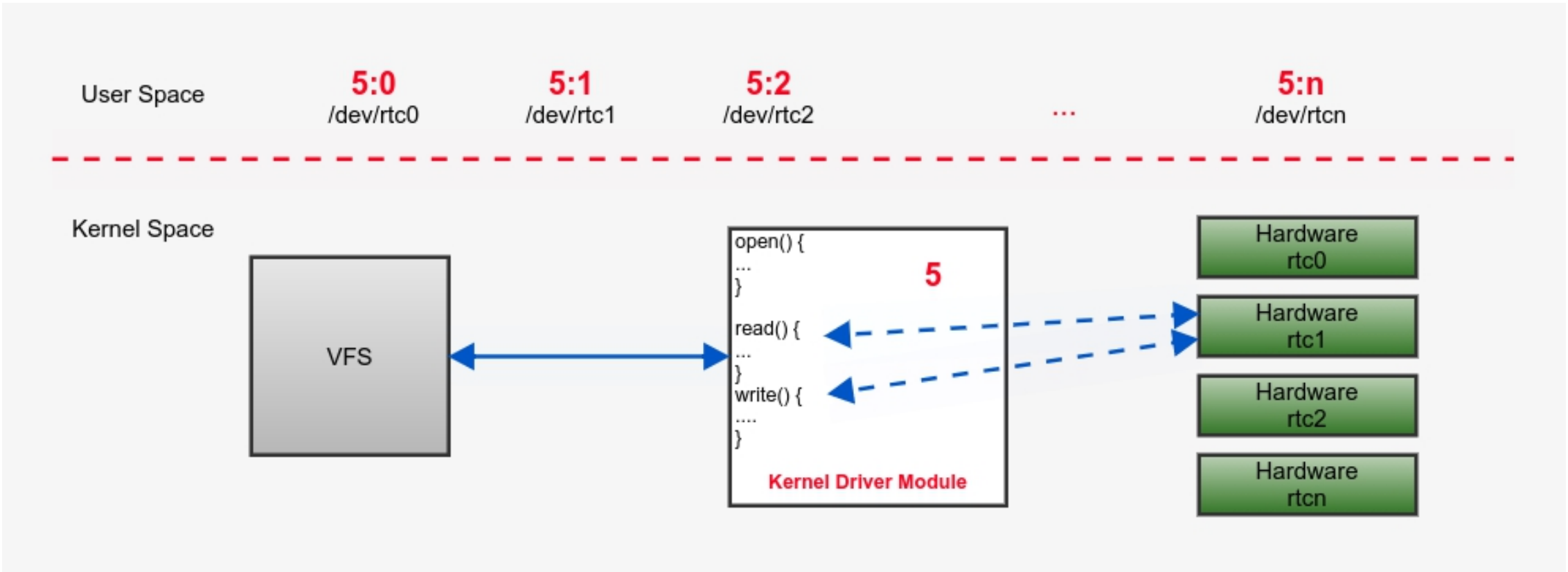
- A Character Device is one with which the Driver communicates by sending and receiving single characters (bytes, octets).
- Character devices provide unbuffered, direct access to the hardware device.

Character driver overview



Device number - Majors and minors




User Space
5:0
/dev/rtc0
5:1
/dev/rtc1
5:2
/dev/rtc2
...
5:n
/dev/rtcn
Kernel Space
VFS
open() {
...
}
read() {
...
}
write() {
....
}
5
Kernel Driver Module
Hardware
rtc0
Hardware
rtc1
Hardware
rtc2
Hardware
rtcn

```
crw-rw-rw-   1 root root      1,   3 Jun 28 07:03 null
crw-------   1 root root     10, 144 Jun 28 07:03 nvram
crw-r-----   1 root kmem      1,   4 Jun 28 07:03 port
crw-------   1 root root    108,   0 Jun 28 07:03 ppp
crw-------   1 root root     10,   1 Jun 28 07:03 psaux
crw-rw-rw-   1 root tty       5,   2 Jun 29 15:13 ptmx
drwxr-xr-x   2 root root           0 Jun 28 07:02 pts/
crw-rw-rw-   1 root root      1,   8 Jun 28 07:03 random
crw-rw-r--+  1 root netdev   10, 242 Jun 28 07:03 rfkill
lrwxrwxrwx   1 root root           4 Jun 28 07:03 rtc -> rtc0
crw-------   1 root root    249,   0 Jun 28 07:03 rtc0
brw-rw----   1 root disk      8,   0 Jun 28 07:03 sda
brw-rw----   1 root disk      8,   1 Jun 28 07:03 sda1
brw-rw----   1 root disk      8,  16 Jun 28 07:03 sdb
brw-rw----   1 root disk      8,  17 Jun 28 07:03 sdb1
crw-rw----   1 root disk     21,   0 Jun 28 07:03 sg0
crw-rw----+  1 root cdrom    21,   1 Jun 28 07:03 sg1
crw-rw----   1 root disk     21,   2 Jun 28 07:03 sg2
drwxrwxrwt   2 root root          40 Jun 29 15:13 shm/
crw-------   1 root root     10, 231 Jun 28 07:03 snapshot
drwxr-xr-x   3 root root         280 Jun 28 07:03 snd/
brw-rw----+  1 root cdrom    11,   0 Jun 28 07:03 sr0
lrwxrwxrwx   1 root root          15 Jun 28 07:02 stderr -> /proc/self/fd/2
lrwxrwxrwx   1 root root          15 Jun 28 07:02 stdin -> /proc/self/fd/0
lrwxrwxrwx   1 root root          15 Jun 28 07:02 stdout -> /proc/self/fd/1
crw-rw-rw-   1 root tty       5,   0 Jun 28 19:28 tty
crw--w----   1 root tty       4,   0 Jun 28 07:03 tty0
crw--w----   1 gdm  tty       4,   1 Jun 28 07:03 tty1
crw--w----   1 root tty       4,  10 Jun 28 07:03 tty10
crw--w----   1 root tty       4,  11 Jun 28 07:03 tty11
crw--w----   1 root tty       4,  12 Jun 28 07:03 tty12
crw--w----   1 root tty       4,  13 Jun 28 07:03 tty13
```

ls -l /dev

How does it work?

Connection establishment between device file access and the driver

- Create device number
- Create devie files
- Make a char device registration with the VFS (CDEV_ADD)
- Implement the driver’s file operation methods for open, read, write, llseek, etc.

Kernel APIs and utilities to be used in driver code

alloc_chrdev_region(); 1. Create device number

cdev_init(); 2. Make a char device registration with the VFS
cdev_add();

class_create(); 3. Create device files
device_create();

Creation

alloc_chrdev_region();

cdev_init();
cdev_add();

class_create();
device_create();

Deletion

unregister_chrdev_region();

cdev_del();

class_destroy();
device_destroy();

Kernel functions and data structures	Kernel header file
alloc_chrdev_region() unregister_chrdev_region()	include/linux/fs.h
cdev_init() cdev_add() cdev_del()	include/linux/cdev.h
device_create() class_create() device_destroy() class_destory()	include/linux/device.h
copy_to_user() copy_from_user()	include/linux/uaccess.h
VFS structure definitions	include/linux/fs.h

Ask the kernel to dynamically allocate the device number(s)

```
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,
                        unsigned count, const char *name)
```

arg1: output parameter for first assigned number

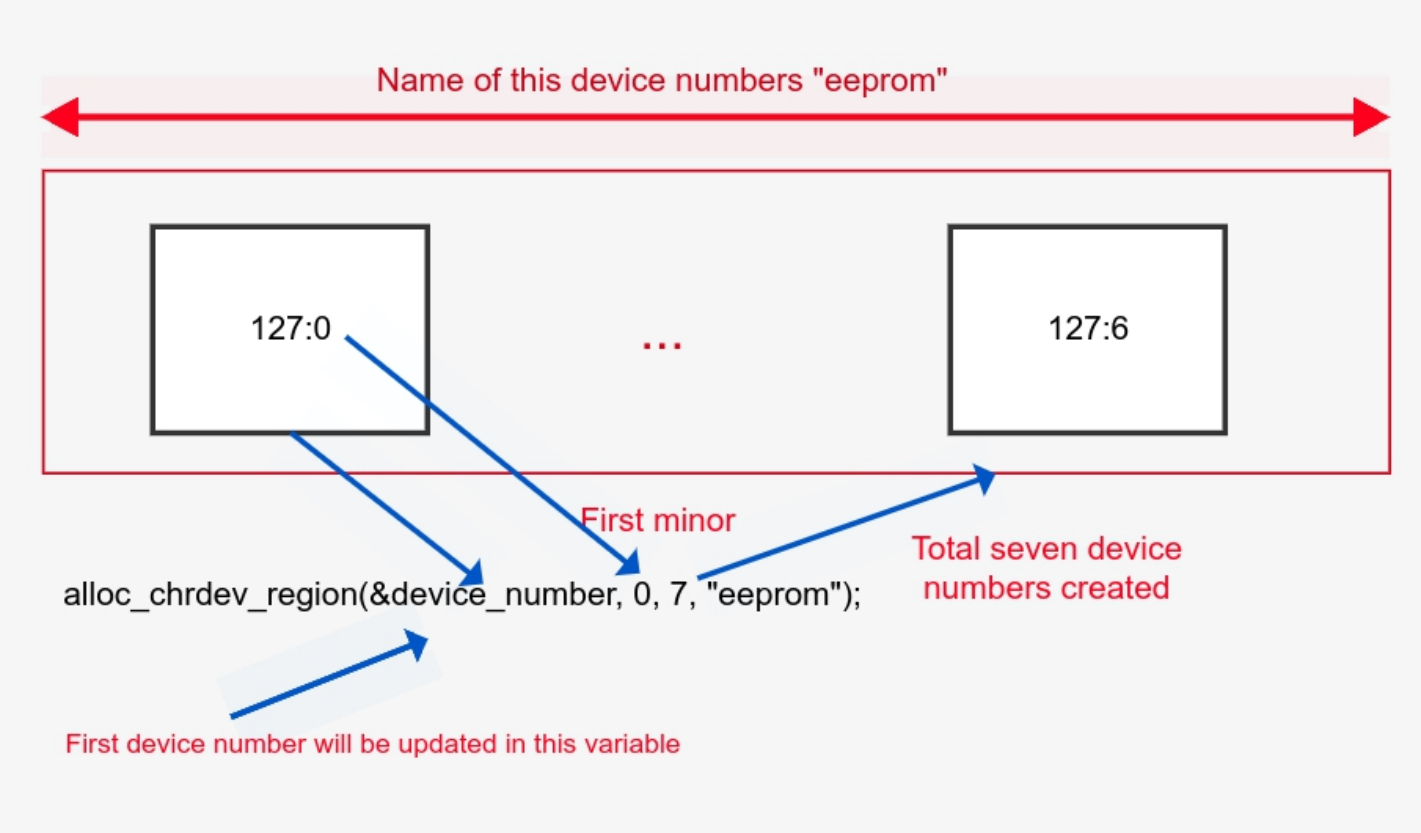
arg2: first of the requested range of minor numbers

arg3: number of minor numbers required

arg4: name of the associated device or driver, not device name, just to identify device number range

Example

```
/* Device number creation */
devt_t device_number;
alloc_chrdev_region(&device_number, 0, 7, "eeprom");
```



Device number representation

- **device_number = major number + minor numbers**
- **Out of 32 bits, 12 bits for major, 20 bits for minor**
- **Macros in linux/kdev_t.h**

```
int minor_no = MINOR(device_number)
int major_no = MAJOR(device_number)

/* Defined with specified number. */
MKDEV(int major, int minor)
```

Register a char device to the Kernel VFS

```
// fs/char_dev.c

/**
 * cdev_init() - initialize a cdev structure
 * @cdev: the structure to initialize
 * @fops: the file_operations for this device
 *
 * Initializes @cdev, remembering @fops, making it ready to add to the
 * system with cdev_add().
 */
void cdev_init(struct cdev *cdev, const struct file_operations *fops)
{
        memset(cdev, 0, sizeof *cdev);
        INIT_LIST_HEAD(&cdev->list);
        kobject_init(&cdev->kobj, &ktype_cdev_default);
        cdev->ops = fops;
}
```

Example

```
void cdev_init(struct cdev *cdev, const struct file_operations *fops)

/*
 * Initialize file ops structure with driver's system call implementation methods
 *
 */
struct file_operations eeprom_fops;
struct cdev eeprom_cdev;
cdev_init(&eeprom_cdev, &eeprom_fops);
```

Add(Register) a char device to the Kernel VFS

int cdev_add(struct cdev *p, dev_t dev, unsigned count)

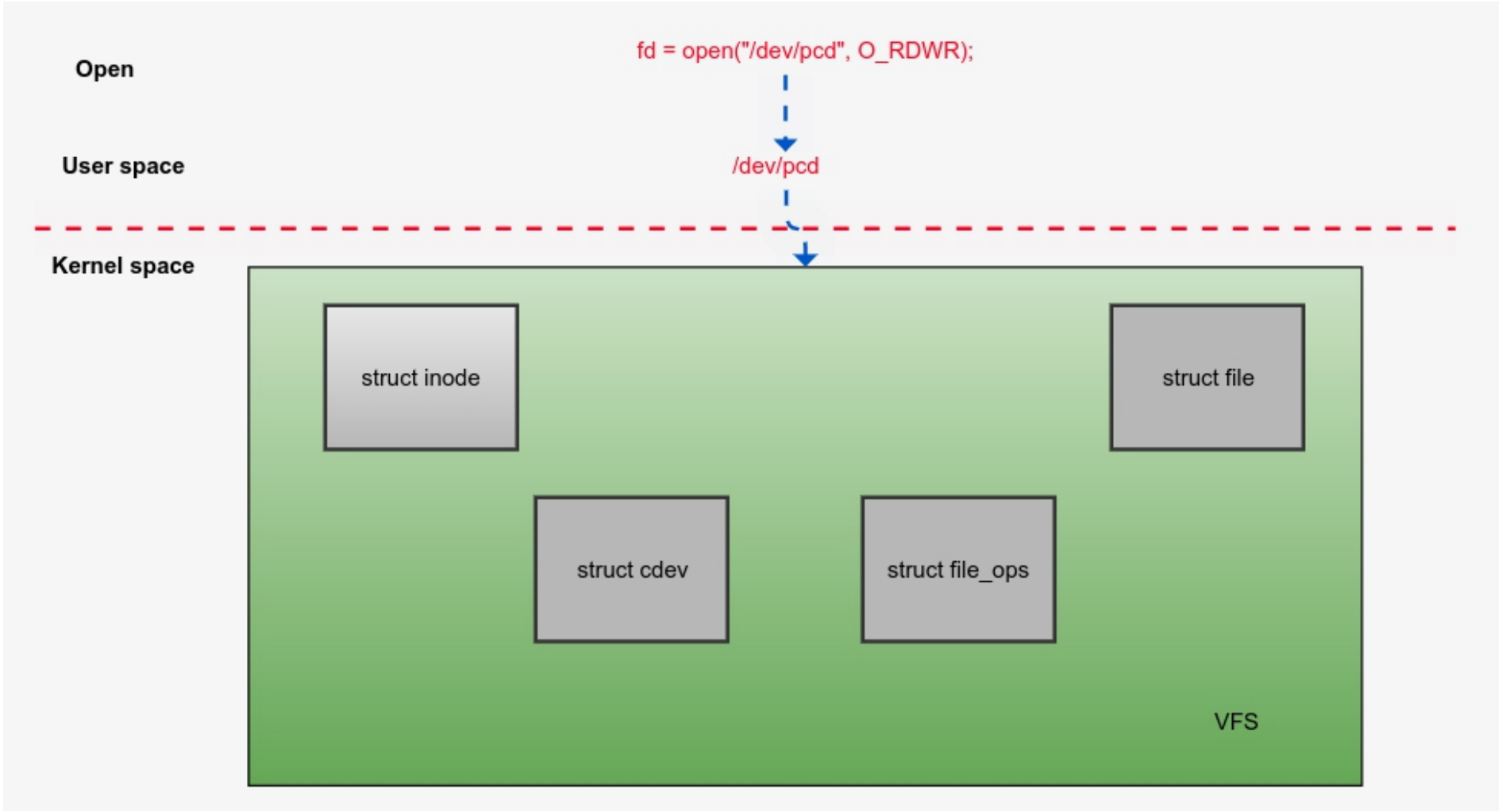
arg1: cdev structure for the device

arg2: number of consecutive minor numbers corresponding to this device

arg3: first device number for which this device is responsible

Data structures for a character device

VFS data structures involved



```
// include/linux/cdev.h

struct cdev {
        struct kobject kobj;
        struct module *owner;
        const struct file_operations *ops;
        struct list_head list;
        dev_t dev;
        unsigned int count;
} __randomize_layout;
```

A pointer to the module that owns this structure;
Usually be initalized to THIS_MODULE.

is used to prevent from being unloaded while the structure is in use.

Pointer to file operation structure of the driver

```
// include/linux/fs.h

struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
        ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
...
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *, fl_owner_t id);
        int (*release) (struct inode *, struct file *);
...
} __randomize_layout;
```

```
// include/linux/export.h

#ifdef MODULE
extern struct module __this_module;
#define THIS_MODULE (&__this_module)          -> point to _this_module
#else
#define THIS_MODULE ((struct module *)0)
#endif
```

```
// include/linux/fs.h

struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        const char * (*get_link) (struct dentry *, struct inode *, struct delayed_call
*);
        int (*permission) (struct inode *, int);
        int (*permission2) (struct vfsmount *, struct inode *, int);
...
        int (*create) (struct inode *,struct dentry *, umode_t, bool);
        int (*link) (struct dentry *,struct inode *,struct dentry *);

        int (*mkdir) (struct inode *,struct dentry *,umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
...
} ____cacheline_aligned;
```

Inode object

- Unix makes a clear distinction between the contents of a file and the information about a file
- An inode is a VFS data structure(struct inode) that holds general information about a file.
- Where as VFS ‘file’ data structure (struct file) tracks interaction on an opened file by the user process
- Inode contains all the information needed by the filesystem to handle a file.
- Each file has its own inode object, which the filesystem uses to identify the file
- Each inode object is associated with an inode number, which uniquely identifies the file within the filesystem.
- The inode object is created and stored in memory as and when a new file (regular or device) get created.

```
// include/linux/fs.h

struct file {
...
        struct path                     f_path;
        struct inode                    *f_inode;       /* cached value */
        const struct file_operations    *f_op;

        unsigned int                    f_flags;
        loff_t                          f_pos;
        struct fown_struct              f_owner;
        const struct cred               *f_cred;
        struct file_ra_state            f_ra;

        u64                             f_version;

        struct address_space    *f_mapping;
        errseq_t                f_wb_err;
}
```

File object

- Whenever a file is opened a file object is created in the kernel space. There will one file object for every open of a regular or device file.
- Stores information about the interaction between an open file and a process
- This information exists only in kernel memory during the period when a process has the file open. The contents of file object is Not written back to disks unlike inode.

Inode object initalization

fs/inode.c

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
        inode->i_mode = mode;
        if (S_ISCHR(mode)) {
                inode->i_fop = &def_chr_fops;
                inode->i_rdev = rdev;
        } else if (S_ISBLK(mode)) {
                inode->i_fop = &def_blk_fops;
                inode->i_rdev = rdev;
        } else if (S_ISFIFO(mode))
...
}
```

Whenever device file is created(udev or mknod) init_special_inode() gets called

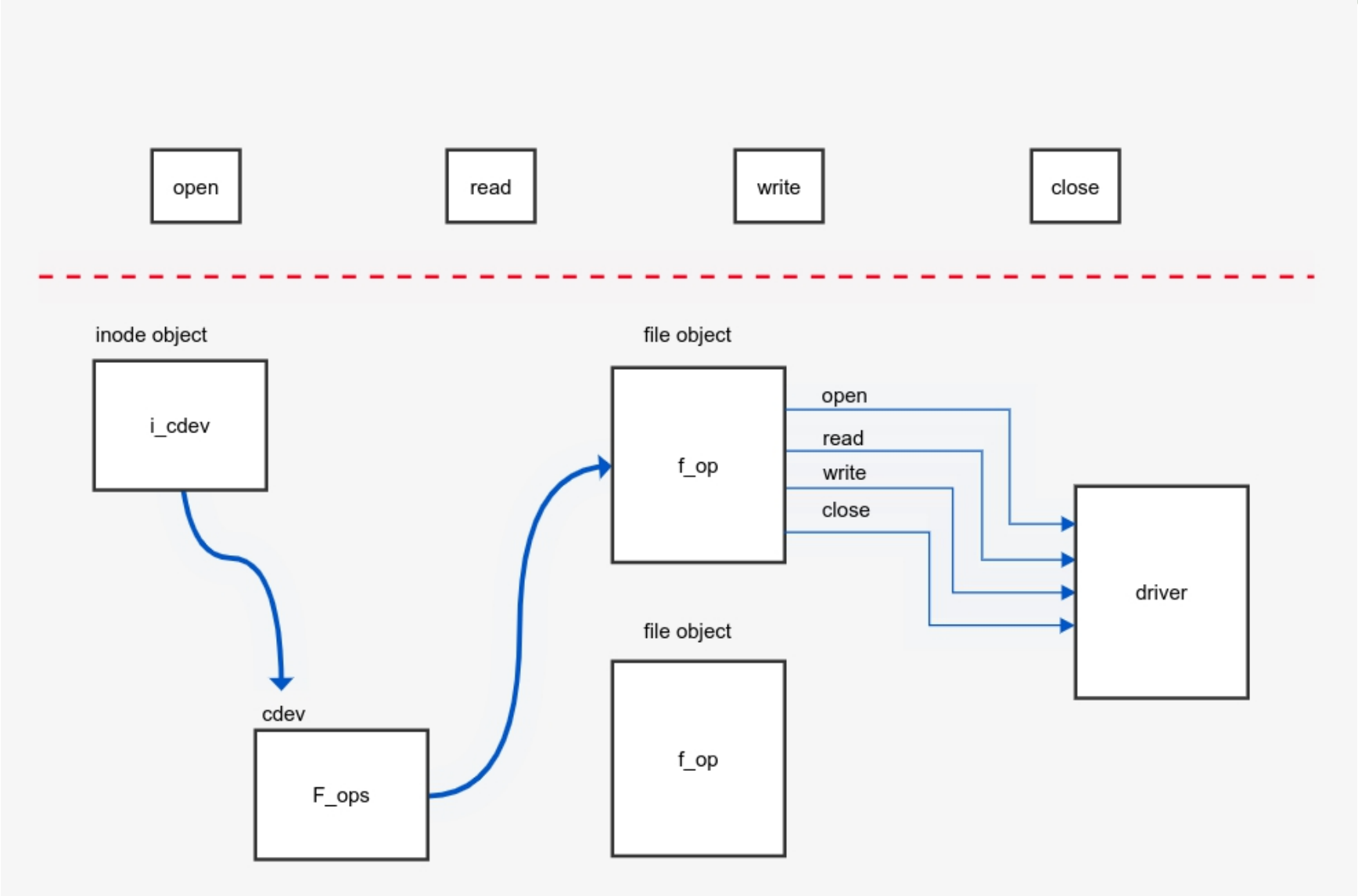
inode->i_fop field is initialized with default file operations (def_chr_fops)

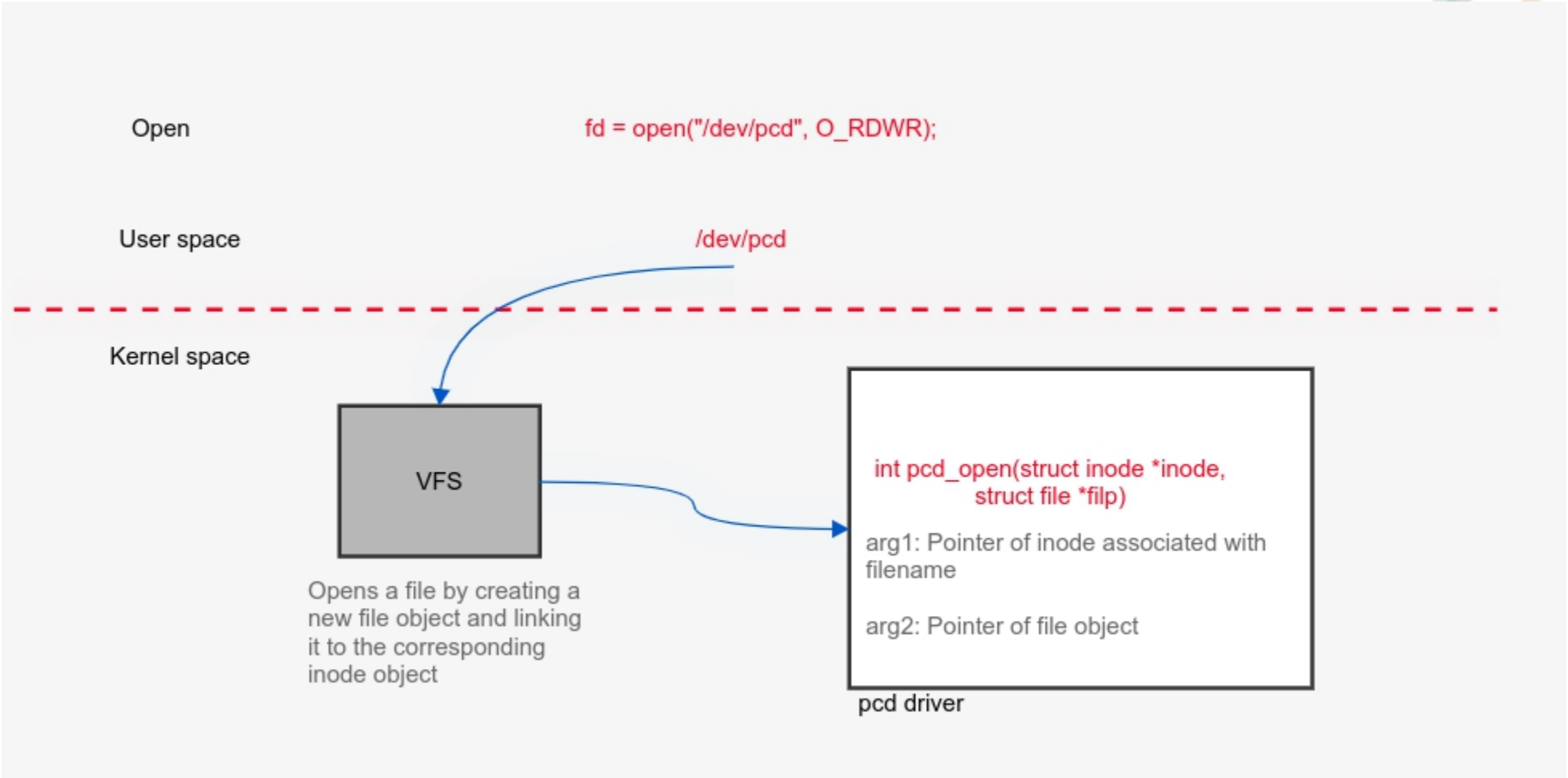
Here, inode object's i_rdev is initiazed(i_rdev is device number)

fs/char_dev.c

```
/*
 * Dummy default file-operations: the only thing this does
 * is contain the open that then fills in the correct operations
 * depending on the special file...
 */
const struct file_operations def_chr_fops = {
        .open = chrdev_open,
        .llseek = noop_llseek,
};
```

How does ‘open’ go?

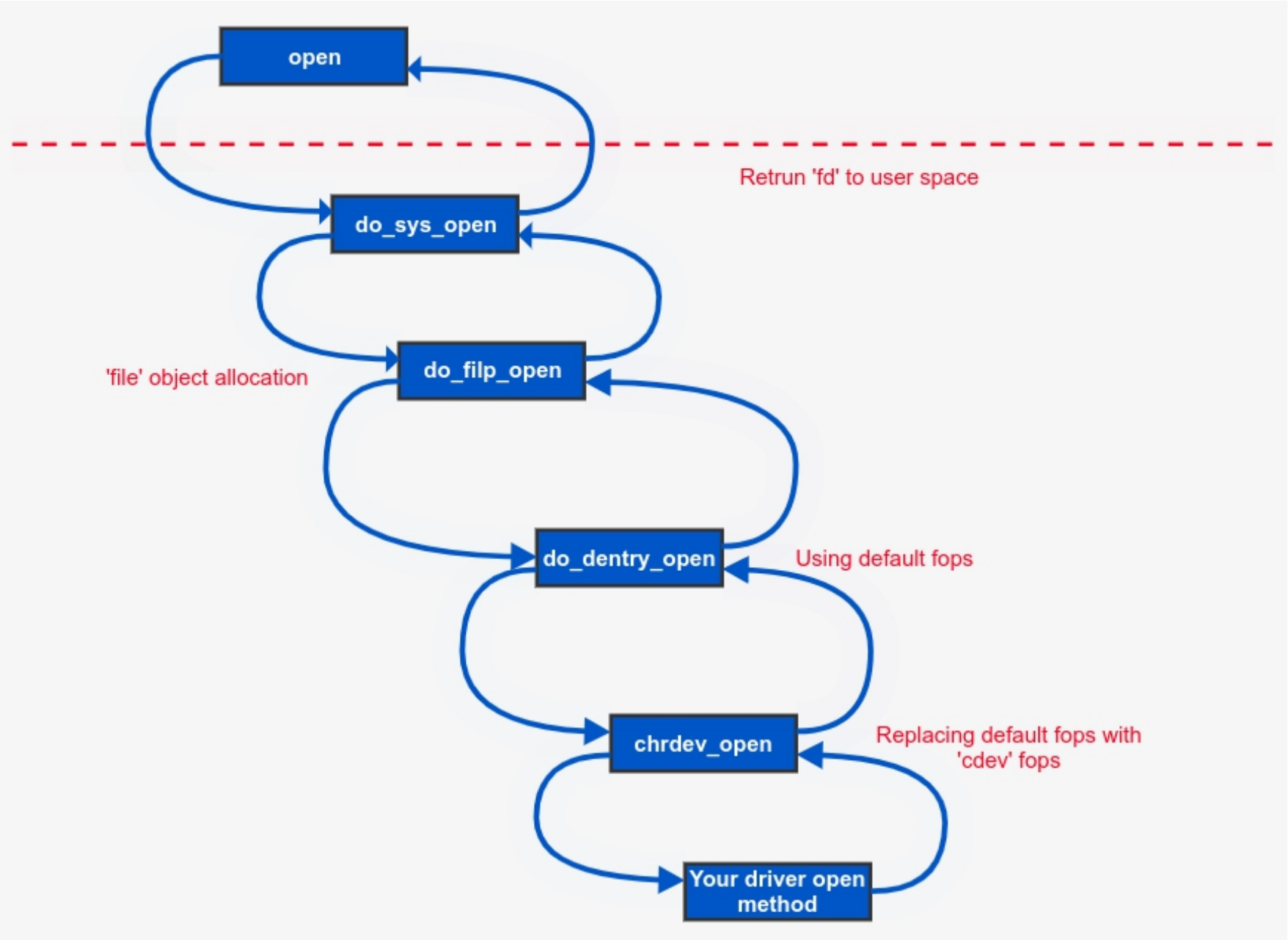



Open
fd = open("/dev/pcd", O_RDWR);
User space
/dev/pcd
Kernel space
VFS
Opens a file by creating a new file object and linking it to the corresponding inode object
int pcd_open(struct inode *inode, struct file *filp)
arg1: Pointer of inode associated with filename
arg2: Pointer of file object
pcd driver

```
// fs/open.c
long do_sys_open(int dfd, const char __user *filename, int flags, umode_t mode)
{
...
                struct file *f = do_filp_open(dfd, tmp, &op);
...
}

// fs/namei.c
struct file *do_filp_open(int dfd, struct filename *pathname,
                const struct open_flags *op)
{
        // allocate file object
        do_dentry_open()    // To be precise, is called by vfs_open() or finish_open()directly
}
```

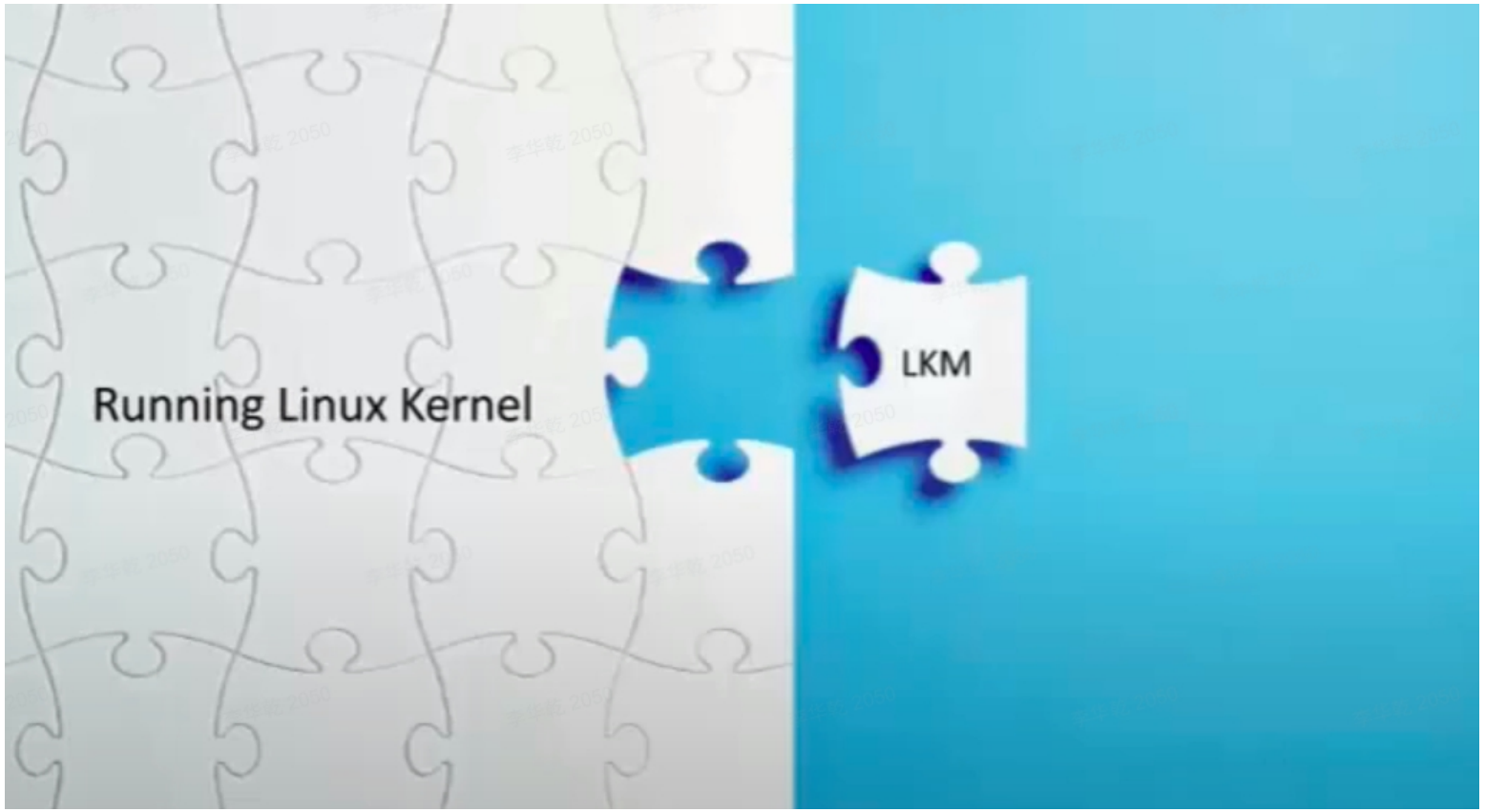
```
// fs/open.c
static int do_dentry_open(struct file *f,
                          struct inode *inode,
                          int (*open)(struct inode *, struct file *))
{
        f->f_op = fops_get(inode->i_fop);      // Coping inode's fops into file object fops

        /* normally all 3 are set; ->open() can clear them if needed */
        f->f_mode |= FMODE_LSEEK | FMODE_PREAD | FMODE_PWRITE;
        if (!open)
                open = f->f_op->open;
        if (open) {
                error = open(inode, f);       // Calling open method of default fops, namely chrdev_open
                if (error)
                        goto cleanup_all;
        }
}
```

```
/*
 * Called every time a character special file is opened
 */
static int chrdev_open(struct inode *inode, struct file *filp)
{
        ...
        if(!p){
                inode->i_cdev = p = new;     // Get cdev, new is added by cdev_add()
        }
        ...
        } else if (!cdev_get(p))
        ...
        fops = fops_get(p->ops);
        ...
        replace_fops(filp, fops);                    // Replace with driver fops
        if (filp->f_op->open) {
                ret = filp->f_op->open(inode, filp);
        }
        ...
}
```


open
Retrun 'fd' to user space
do_sys_open
'file' object allocation
do_filp_open
do_dentry_open
Using default fops
chrdev_open
Replacing default fops with
'cdev' fops
Your driver open
method

Coding

- Building Intree Modules
- Building External Modules



```c
#include <linux/module.h>

/* Module initialization entry point */
static int __init pcd_driver_init(void)
{

        return 0;
}

/* Module clean-up entry point */
static void __exit pcd_driver_exit(void)
{}

/* Registration of entry points with kernel */
module_init(pcd_driver_init);
module_exit(pcd_driver_exit);

/* Descriptive information about the module */
MODULE_LICENSE("GPL");
MODULE_AUTHOR("TS");
MODULE_DESCRIPTION("A Psuedo char driver!");
```

Header section

Your code

Registration

Module description

```
#define DEV_MEM_SIZE 512

/* pseudo device’s memory */
char device_buffer[DEV_MEM_SIZE];

/* This holds the device number */
dev_t device_number;

/* Cdev varialbe */
struct cdev pcd_cdev;

/* File operations of the driver */
struct file_operations pcd_fops;
```

```
static int __init pcd_driver_init(void)
{
        /* 1. Dynamically allocate a device number */
        alloc_chardev_region(&device_number, 0, 1, “pcd”);

        /* 2. Initialize
        cdev_init(&pcd_cdev, &pcd_fops);

        /* 3. Register a device(cdev structure) with VFS */
        pcd_cdev.owner = THIS_MODULE;
        cdev_add(&pcd_cdev, device_number, 1);

        /* 4. Create class and /dev/pcd */
        pcd_class = class_create(THIS_MODULE, “pcd”);
        device_create(pcd_class, NULL,

        return 0;
}

static void __exit pcd_driver_exit(void)
{}
```

```
// Makefile

obj-m += chardev.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

Summary

when device file gets created

- create device file using udev
- inode object gets created in memory and inode’s i_rdev field is initialized with device number.
- inode object’s i_fop field is set to dummy default file operations (dev_chr_fops)

when user process executes open system call

- user invokes open system call on the device file
- file object gets created
- inode’s i_op gets copied to file object’s f_op(dummy default file operations of char device file)
- open function of dummy default file operations gets called(chrdev_open)
- inode object’s i_cdev field is initialized with cdev which you added during cdev_add (lookup happens using inode->i_rdev field)
- inode->cdev->fops(this is a real file operations of your driver)gets copied to file->f_op
- file->f_op->open method gets called(read open method of your driver)

After class

Device nodes on Unix-like systems do not necessarily have to correspond to physical devices. Some of the most commonly used (character-based) pseudo-devices include:

- /dev/mem — Physical memory access
- /dev/kmem — Kernel virtual memory access
- /dev/null — Null device
- /dev/zero — Null byte source
- /dev/random — Nondeterministic random number gen.
- /dev/kmsg — Writes to this come out as printk's, reads export the buffered printk records.

Documentation/devices.txt
drivers/char/.c*

1. Build a intree char device kernel Module.

2. klife(类飞行棋) is a Linux kernel Game of Life implementation.

- The game of life is played on a square grid, where some of the cells are alive and the rest are dead.
- Each generation, based on each cell’s neighbors, we mark the cell as alive or dead.
- With time, amazing patterns develop.
- The only reason to implement the game of life inside the kernel is for demonstration purposes.

Think about /dev/null, /dev/zero, /dev/random, /dev/kmem...

klife uses the following device file operations:

- **open** for starting a game (allocating resources).
- **release** for finishing a game (releasing resources).
- **write** for initializing the game (setting the starting positions on the grid).
- **read** for generating and then reading the next state of the game’s grid.
- **ioctl** for querying the current generation number, and for enabling or disabling hooking into the timer interrupt (more on this later).
- **mmap** for potentially faster but more complex direct access to the game’s grid.

Questions And Suggestions?

Thanks