### What is a character device?

Character devices do data transfer character by character example: keyboard, mouse, and parallel port.

### What is a character driver?

A character driver drives a character device

A char driver is in charge of implementing the behavior of character device.

### How does linux identify device and their drivers?

Linux identifies devices and their drivers with the help of Device Numbers

* Major number
* Minor number

### Major and Minor Numbers

The **major number identifies the driver associated with the device.** The kernel uses the major number at open time to dispatch execution to the appropriate driver. The minor number is used only by the driver; other parts of the kernel don't use it, and merely pass it along to the driver. It is common for a driver to control several devices; the minor number provides a way for the driver to differentiate among them.

### The Internal Representation of Device Numbers

Within the kernel, the dev\_t type (defined in <linux/types.h>) is used to hold device numbers—both the major and minor parts. As of Version 2.6.0 of the kernel, dev\_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number. Your code should, never make any assumptions about the internal organization of device numbers; it should, instead, make use of a set of macros found in <linux/kdev\_t.h>. To obtain the major or minor parts of a dev\_t, use:

MAJOR(dev\_t dev);

MINOR(dev\_t dev);

If, instead, you have the major and minor numbers and need to turn them into a dev\_t,

use:

MKDEV(int major, int minor);

### How to create a device file?

The command to create a device node on a filesystem is mknod; superuser privileges are required for this operation. The command takes three arguments in addition to the name of the file being created. For example, the command

mknod mydev c 254 0

Creates a char device (c) whose major number is 254 and whose minor number is 0. Minor numbers should be in the range 0 to 255 because, for historical reasons, they are sometimes stored in a single byte. There are sound reasons to extend the range of available minor numbers, but for the time being, the eight-bit limit is still in force.

### Two ways of Allocation of a major no. for a driver:

1. Statically
2. Dynamically

**Static Allocation**: Some major device numbers are statically assigned to the most common devices. A list of those devices can be found in *Documentation/devices.txt* within the kernel

source tree. So, as a driver writer, you have a choice: you can simply pick a number that appears to

be unused. ( may lead to conflicts and hence trouble) to avoid this use dynamic allocation

**Dynamic allocation**: use dynamic allocation to obtain your major device number, rather than choosing a number randomly from the ones that are currently free. Using *alloc\_chrdev\_region.*

One of the first things your driver will need to do when setting up a char device is to obtain one or more device numbers to work with. The necessary function for this task is *register\_chrdev\_region*, which is declared in *<linux/fs.h>*:

**Static request for device numbers**

int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name);

Arguments to register\_chrdev\_region and return value:

* first is the beginning device number of the range you would like to allocate
* count is the total number of contiguous device numbers you are requesting. name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.
* return value from register\_chrdev\_regionwill be 0 if the allocation was successfully erformed. In case of error, a negative error code will be returned, and you will not have access to the requested region.

**Dynamicly request for numbers**

int alloc\_chrdev\_region(dev\_t \*dev, unsigned int firstminor, unsigned int count, char \*name);

Arguments to register\_chrdev\_region and return value:

* dev is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
* firstminor should be the requested first minor number to use; it is usually 0.
* count is the total number of contiguous device numbers you are requesting.
* name is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.

### Freeing the device

When a module is unloaded from the system, the major number must be released. This is accomplished with the following function, which you call from the module's cleanup function:

void unregister\_chrdev\_region(dev\_t first, unsigned int count);

Arguments to unregister\_chrdev\_region and return value:

* dev is the device no.
* count is the count of the minor numbers.

Example:

#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>

static dev\_t first; // Global variable for the first device number

static int \_\_init myinit (void) {

    printk(KERN\_INFO "sample driver registered");

    if (alloc\_chrdev\_region(&first, 0, 3, "sample") < 0)

    {

        return -1;

    }

    printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n",MAJOR(first), MINOR(first));

    return 0;

}

static void \_\_exit myexit(void)

{

    unregister\_chrdev\_region(first, 3);

    printk(KERN\_INFO "sample driver unregistered\n");

}

module\_init(myinit);

module\_exit(myexit);

MODULE\_LICENSE("GPL");

MODULE\_AUTHOR("Type your name");

MODULE\_DESCRIPTION("Our First sample character Driver");

### Important kernel data structures

Some of the datastructures required to implement fundamental driver operations are file\_operations, file,and inode.

### File Operations *<linux/fs.h>*:

struct file\_operatios contains pointers to functions performed on our device through our driver. Open and check the fs.h it is a vary vast structure, discussing few important function pointers

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

Though this is always the first operation performed on the device file, the driver is not required to declare a corresponding method.

If this entry is NULL, opening the device always succeeds, but your driver isn't notified.

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

This operation is invoked when the file structure is being released. Like *open*, *release* can be missing.

[Note that *release* isn't invoked every time a process calls *close*. Whenever a file structure is shared (for example, after a *fork* or a *dup*), *release* won't be invoked until all copies are closed. If you need to flush pending data when any copy is closed, you should implement the *flush* method.]

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

Used to retrieve data from the device. A null pointer in this position causes the *read* system call to fail with -EINVAL ("Invalid argument").

A non-negative return value represents the number of bytes successfully read (the return value is a "signed size" type, usually the native integer type for the target platform).

**ssize\_t (\*write) (struct file \*, const char \*, size\_t, loff\_t \*);**Sends data to the device. If missing, -EINVAL is returned to the program calling the *write* system call. The return value, if non-negative, represents the number of bytes successfully written.

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

The *llseek* method is used to change the current read/write position in a file, and the new position is returned as a (positive) return value. The loff\_t is a "long offset" and is at least 64 bits wide even on 32-bit platforms. Errors are signaled by a negative return value. If the function is not specified for the driver, a seek relative to end-of-file fails, while other seeks succeed by modifying the position counter in the file structure

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

The *ioctl* system call offers a way to issue device-specific commands (like formatting a track of a floppy disk, which is neither reading nor writing). Additionally, a few *ioctl* commands are recognized by the kernel without referring to the fops table. If the device doesn't offer an *ioctl* entry point, the system call returns an error for any request that isn't predefined (-ENOTTY, "No such ioctl for device"). If the device method returns a non-negative value, the same value is passed back to the calling program to indicate successful completion.

### The file Structure <linux/fs.h>:

Note that a file has nothing to do with the FILEs of user-space programs. A FILE is defined in the C library and never appears in kernel code. A struct file, on the other hand, is a kernel structure that never appears in user programs.

The file structure represents an open file. (It is not specific to device drivers; every open file in the system has an associated struct file in kernel space.) It is created by the kernel on open and is passed to any function that operates on the file, until the last close.

After all instances of the file are closed, the kernel releases the data structure. An open file is different from a disk file, represented by struct inode.

In the kernel sources, a pointer to struct file is usually called either file or filp ("file pointer”) . We'll consistently call the pointer filp to prevent ambiguities with the structure itself. Thus, file refers to the structure and filp to a pointer to the structure.

The most important fields of struct file are shown here. As in the previous section, the list can be skipped on a first reading. In the next section though, when we face some real C code, we'll discuss some of the fields, so they are here for you to refer to.

**mode\_t f\_mode;**

The file mode identifies the file as either readable or writable (or both), by means of the bits FMODE\_READ and FMODE\_WRITE. You might want to check this field for read/ write permission in your ioctl function, but you don't need to check permissions for read and write because the kernel checks before invoking your method. An attempt to write without permission, for example, is rejected without the driver even knowing about it.

**loff\_t f\_pos;**

The current reading or writing position. L off\_t is a 64-bit value (long long in gcc terminology). The driver can read this value if it needs to know the current position in the file, but should never change it (read and write should update a position using the pointer they receive as the last argument instead of acting on filp->f\_pos directly).

**unsigned int f\_flags;**

These are the file flags, such as O\_RDONLY, O\_NONBLOCK, and O\_SYNC. A driver needs to check the flag for nonblocking operation, while the other flags are seldom used. In particular, read/write permission should be checked using f\_mode instead of f\_flags. All the flags are defined in the header <linux/fcntl.h>.

**struct file\_operations \*f\_op:**

The operations associated with the file. The kernel assigns the pointer as part of its implementation of open, and then reads it when it needs to dispatch any operations. The value in filp->f\_op is never saved for later reference; this means that you can change the file operations associated with your file whenever you want, and the new methods will be effective immediately after you return to the caller.

For example, the code for open associated with major number 1 (/dev/null, /dev/zero, and so on) substitutes the operations in filp->f\_op depending on the minor number being opened.

This practice allows the implementation of several behaviors under the same major number without introducing overhead at each system call. The ability to replace the file operations is the kernel equivalent of "method overriding" in object-oriented programming.

**void \*private\_data;**

The open system call sets this pointer to NULL before calling the openmethod for the driver. The driver is free to make its own use of the field or to ignore it.

The driver can use the field to point to allocated data, but then must free memory in the release method before the file structure is destroyed by the kernel. Private\_data is a useful resource for preserving state information across system calls and is used by most of our sample modules.

**struct dentry \*f\_dentry;**

The directory entry (dentry) structure associated with the file. Dentries are an optimization introduced in the 2.1 development series. Device driver writers normally need not concern themselves with dentry structures, other than to access the inode structure as filp->f\_dentry->d\_inode.

The real structure has a few more fields, but they aren't useful to device drivers. We can safely ignore those fields because drivers never fill file structures; they only access structures created elsewhere.

### Char Device Registration

The kernel uses structures of type struct cdev to represent char devices internally. Before the kernel invokes your device’s operations, you must allocate and register one or more of these structures. To do so, your code should include <linux/cdev.h>, where the structure and its associated helper functions are defined.

There are two ways of allocating and initializing one of these structures. If you wish to obtain a standalone cdev structure at runtime, you may do so with code such as:

struct cdev \*my\_cdev = cdev\_alloc( );

my\_cdev->ops = &my\_fops;

however, that you will want to embed the cdev structure within a device-specific structure of your own; In that case, you should initialize the structure that you have already allocated with:

void cdev\_init(struct cdev \*cdev, struct file\_operations \*fops);

Either way, there is one other struct cdev field that you need to initialize. Like the file\_operations structure, struct cdev has an owner field that should be set to THIS\_MODULE.

Once the cdev structure is set up, the final step is to tell the kernel about it with a call to:

int cdev\_add(struct cdev \*dev, dev\_t num, unsigned int count);

Here, dev is the cdev structure, num is the first device number to which this device responds, and count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.

There are a couple of important things to keep in mind when using cdev\_add. The first is that this call can fail. If it returns a negative error code, your device has not been added to the system. It almost always succeeds, however, and that brings up the other point: as soon as cdev\_add returns, your device is “live” and its operations can be called by the kernel. You should not call cdev\_add until your driver is completely ready to handle operations on the device.

To remove a char device from the system, call:

void cdev\_del(struct cdev \*dev);

you should not access the cdev structure after passing it to cdev\_del.

### Example with cdev:

struct scull\_dev {

data \*data; /\* Pointer to first quantum set \*/

int current,max\_size

struct cdev cdev; /\* Char device structure \*/

};

The struct cdev interfaces our device to the kernel. Thisistructure must be initialized and added to the system as described above;

static void my\_init( )

{

int err, devno = MKDEV(scull\_major, scull\_minor);

cdev\_init(&dev->cdev, &scull\_fops);

dev->cdev.owner = THIS\_MODULE;

err = cdev\_add (&dev->cdev, devno, 1);

/\* Fail gracefully if need be \*/

if (err)

printk(KERN\_NOTICE "Error %d adding scull%d", err, index);

}

Since the cdev structure is embedded within struct scull\_dev, cdev\_init must be called to perform the initialization of that structure.