**Character Driver Design**

Char devices are accessed through names in the filesystem. Those names are called special files or device files or simply nodes of the filesystem tree; they are conventionally located in the */dev* directory. Special files for char drivers are identified by a “c” in the first column of the output of *ls –l*. Block devices appear in */dev* as well, but

they are identified by a “b.”

If you issue the *ls –l* command, you’ll see two numbers (separated by a comma) in the device file entries before the date of the last modification, where the file length normally appears. These numbers are the major and minor device number for the particular device.

crw-rw-rw- 1 root root 1, 3 Apr 11 2002 null

crw------- 1 root root 10, 1 Apr 11 2002 psaux

crw------- 1 root root 4, 1 Oct 28 03:04 tty1

crw-rw-rw- 1 root tty 4, 64 Apr 11 2002 ttys0

crw-rw---- 1 root uucp 4, 65 Apr 11 2002 ttyS1

The above major numbers are 1, 4, and 10, while the minors are 1, 3, 5, 64, 65 and 129.

Traditionally, the major number identifies the driver associated with the device. The minor number is used by the kernel to determine exactly which device is being referred to.

**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. dev\_t is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor

number. 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);*

Upto 255 we can assign major numbers.

**Allocating and Freeing Device Numbers**

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>*:

*int register\_chrdev\_region(dev\_t first, unsigned int count, char \*name);(return 0 on success and –ve value on failuire)*

first - first is the beginning device number of the range you would like to allocate. The minor number portion of first is often 0.

Count:It is the total number of contiguous device numbers you are requesting.

Name: It is the name of the device that should be associated with this number range; it will appear in */proc/devices* and sysfs.

*register\_chrdev\_region* works well if you know ahead of time exactly which device numbers you want.If you don’t know which device number you want,then you can move to use dynamically allocated device numbers by kernel using

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

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.

The count and name parameters work like those given to *request\_chrdev\_region*.

Regardless of how you allocate your device numbers, you should free them when they are no longer in use. Device numbers are freed with:

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

The usual place to call *unregister\_chrdev\_region* would be in your module’s cleanup function.

For e.g in drivers/char/sample.c

#define DEV\_MAJOR 170

dev\_t dev;

Static int \_\_init dev\_init(void)

{

int status;

dev = MKDEV(DEV\_MAJOR, 0);

status = register\_chrdev\_region(dev, 2, "dev\_name");

if (status){

printk("Not registering major-minor\n");

return status;

}

}

static void \_\_exit dev\_exit(void)

{

unregister\_chrdev(DEV\_MAJOR,”driver\_name”);

}

**Driver data structures**

Most of the fundamental driver operations involve three important kernel data structures, called file\_operations, file,and inode.

***File Operations(this can be called as driver prototypes also)***

The file\_operations structure is how a char driver sets up this connection. The structure, defined in *<linux/fs.h>*, is a collection of function pointers. Each open file is associated with its own set of functions. The operations are mostly in charge of implementing the system calls and are therefore, named *open*, *read*, and so on.

A file\_operations structure or a pointer to one is called fops. Each field in the structure must point to the function in the driver that implements a specific operation, or be left NULL for unsupported operations.

Some of the important members:

*struct module \*owner :*

This field is used to prevent the module from being unloaded while its operations are in use. Almost all the time, it is simply initialized to THIS\_MODULE, a macro defined in *<linux/module.h>*.

*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 parameter is a “long offset” and is at least 64 bits wide even on 32-bit platforms.

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

Used to retrieve data from the device.

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

Sends data to the device. If NULL, -EINVAL is returned to the program calling the write system call. The return value, if nonnegative, represents the number of bytes successfully written.

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

The *poll* method is the back end of three system calls: *poll, epoll,* and *select*, all of which are used to query whether a read or write to one or more file descriptors would block. The *poll* method should return a bit mask indicating whether nonblocking reads or writes are possible, and, possibly, provide the kernel with information that can be used to put the calling process to sleep until I/O becomes possible.

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

The *ioctl* system call offers a way to issue device-specific commands (such as 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.(returns –ENOTTY on failure)

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

*mmap* is used to request a mapping of device memory to a process’s address space. If this method is NULL, the *mmap* system call returns -ENODEV.

*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 NULL.

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

The *flush* operation is invoked when a process closes its copy of a file descriptor for a device; it should execute (and wait for) any outstanding operations on the device. This must not be confused with the *fsync* operation requested by user programs.

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

This method is the back end of the *fsync* system call, which a user calls to flush any pending data. If this pointer is NULL, the system call returns -EINVAL.

**The file Structure**

struct file, defined in *<linux/fs.h>*, is the second most important data structure used in device drivers. Note that a file has nothing to do with the FILE pointers 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.

Some of the important fields are

*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 *open* or *ioctl* function, but you don’t need to check permissions for *read* and *write*.

*loff\_t f\_pos;*

The current reading or writing position. loff\_t is a 64-bit value on all platforms (long long in *gcc* terminology). The driver can read this value if it needs to know the current position in the file but should not normally 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. The one exception to this rule is in the *llseek* method, the purpose of which is to change the file position.

*unsigned int f\_flags;*

These are the file flags, such as O\_RDONLY, O\_NONBLOCK, and O\_SYNC. A driver should check the O\_NONBLOCK flag to see if nonblocking operation has been requested. In particular, read/write permission should be checked using f\_mode rather than 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 by the kernel for later reference; this means that you can change the file operations associated with your file, and the new methods will be effective after you return to the caller.

**The inode Structure**

The inode structure contains a great deal of information about the file. As a general rule, only two fields of this structure are of interest for writing driver code:

*dev\_t i\_rdev;*

For inodes that represent device files, this field contains the actual device number.

struct cdev \*i\_cdev;

struct cdev is the kernel’s internal structure that represents char devices; this field contains a pointer to that structure when the inode refers to a char device file.

The kernel developers have added two macros that can be used to obtain the major and minor number from an inode:

*unsigned int iminor(struct inode \*inode);*

*unsigned int imajor(struct inode \*inode);*

**Char driver Registration**

The classic way to register a char device driver is with:

*int register\_chrdev(unsigned int major, const char \*name, struct file\_operations \*fops);*

major is the major number of interest, name is the name of the driver (it appears in */proc/devices*), and fops is the default file\_operations structure. A call to *register\_chrdev* registers minor numbers 0–255 for the given major, and sets up a default cdev structure for each. Drivers using this interface must be prepared to handle *open* calls on all 256 minor numbers (whether they correspond to real devices or not), and they cannot use major or minor numbers greater than 255.

If you use *register\_chrdev*, the proper function to remove your device(s) from the system is:

*int unregister\_chrdev(unsigned int major, const char \*name);*

major and name must be the same as those passed to *register\_chrdev*, or the call will fail.

For e.g

static void \_\_init max1242\_adc\_init (void)

{

int status;

status = register\_chrdev(adc\_MAJOR, "adc", &adc\_fops);

if(status < 0 )

return status;

return;

}

static void \_\_exit max1242\_adc\_cleanup (void)

{

unregister\_chrdev(adc\_MAJOR, "adc");

return;

}

**Using the newer Way of 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.

You should initialize the structure that you have already allocated with:

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

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.

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);*

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

For e.g

static struct cdev hello\_cdev;

cdev\_init(&hello\_cdev, &hello\_fops);

if (status)

{

printk("\nNot registering as a char device\n");

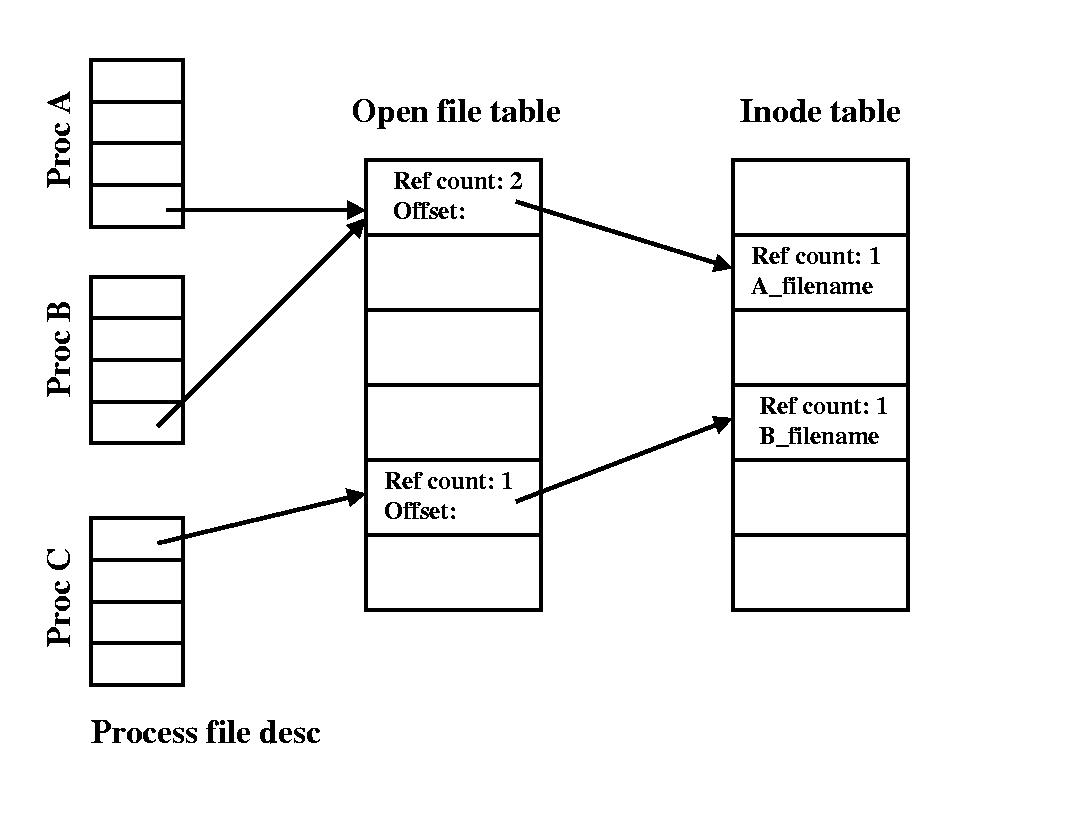
return status;

}

**Relation between inode/File table/File descriptor table**

Each file has one inode but it could have several other names, all which map into the inode. Each of these names is called a link. When a process refers to a file by name, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path and eventually retrieves the inode for the file.When a process creates a new file, the kernel assigns it to an unused inode. Inodes are stored in the file system, but the kernel reads them into an inode table in the core of the primary memory when manipulating files.

The kernel contains two other data structures, the file table and the user file descriptor table. The file table is the global kernel structure, but the user file descriptor table is allocated per process. When a process opens or creats a file, the kernel allocates an entry from each table, corresponding to the file's inode. (Bach 23) The user file descriptor table, file table, and inode table maintain the state of the file and the user's access to it. The file table keeps track of the byte offset in the file where the user's next read or write will start, and the access permissions allowed to the opening process. The user file descriptor table identifies all open files for a process. The kernel returns a file descriptor for the open and creat system calls, which is an index into the user file descriptor table. When executing read and write system calls, the kernel uses the file descriptor to access the user file descriptor table, follows pointers to the file table and inode table entries, and from the inode, finds the data in the file.



There are 3 standard POSIX file descriptors which presumably every process (save perhaps a [daemon](http://en.wikipedia.org/wiki/Daemon_(computer_software))) should expect to have:

|  |  |
| --- | --- |
| **Integer value** | **Name** |
| 0 | Standard Input (stdin) |
| 1 | Standard Output (stdout) |
| 2 | Standard Error (stderr) |