**Device Drivers, Part 1: Linux Device Drivers for Your Girl Friend**

**Anil Kumar Pugalia**

<https://www.opensourceforu.com/tag/linux-device-drivers-series/page/2/>

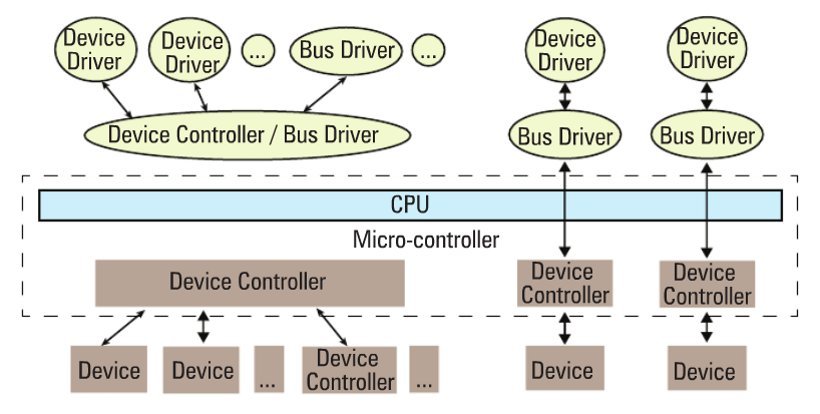
[**https://embetronicx.com/tag/device-driver/**](https://embetronicx.com/tag/device-driver/)

**Of drivers and buses**

A driver drives, manages, controls, directs and monitors the entity under its command. What a bus driver does with a bus, a device driver does with a computer device (any piece of hardware connected to a computer) like a mouse, keyboard, monitor, hard disk, Web-­camera, clock, and more. A specific piece of **hardware** could be **controlled** by a piece of **software** (a **device driver**), or could be **controlled** by **another hardware device**, which in turn could be managed by a **software device driver**. In the latter case, such a controlling device is commonly called a **device controller**. This, being a device itself, often also needs a driver, which is commonly referred to as a **bus driver**.

**Device > Device driver**

**Device > Device controller > Bus driver > Device driver**

General examples of **device controllers** include **hard disk controllers**, **display controllers**, and **audio controllers** that in turn manage devices connected to them. More technical examples would be an **IDE controller, PCI controller, USB controller, SPI controller, I2C controller,** etc. Pictorially, this whole concept is depicted in Figure 1.

**Figure 1: Device and driver interaction**

**Device controllers** are typically connected to **CPU** through their respectively named **buses** (collection of physical lines) – for example, the **PCI bus**, the **IDE bus**, etc. In today’s embedded world, we encountered more micro-controllers than CPUs; these are the **CPU** plus various **device controllers** built onto **a single chip**. This effective embedding of device controllers primarily reduces cost and space, making it suitable for embedded systems. In such cases, the **buses** are **integrated** into the **chip itself** [**SoC**].

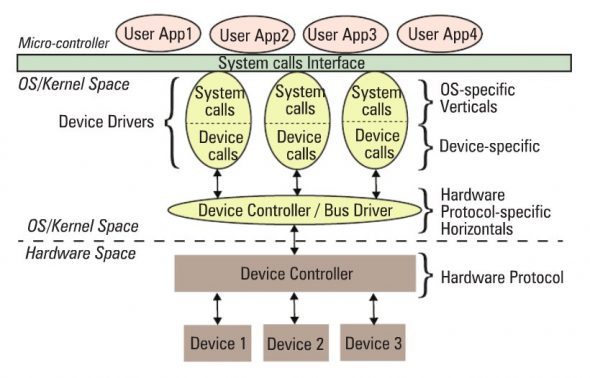
Does this change anything for the drivers, or more generically, on the software front?

The answer is, not much – except that the **bus drivers** corresponding to the **embedded device controllers** are now developed under the **architecture-specific umbrella**.

**Drivers have two parts**

**Bus drivers provide hardware ­specific interfaces for the corresponding hardware protocols**, and **are the bottom-­most horizontal software layers of an operating system (OS)**. Over these sit the actual **device drivers**. These operate on the underlying devices using the horizontal layer interfaces, and hence are device ­specific. However, the whole idea of writing these drivers is to provide an **abstraction** to the user, and so, at the other “end”, these drivers can be used to provide an **interface, which could vary from OS to OS**. In short, a **device driver has two parts**, which are:

a) **device ­specific**, and

b) **OS ­specific**.  
Refer to Figure 2. 

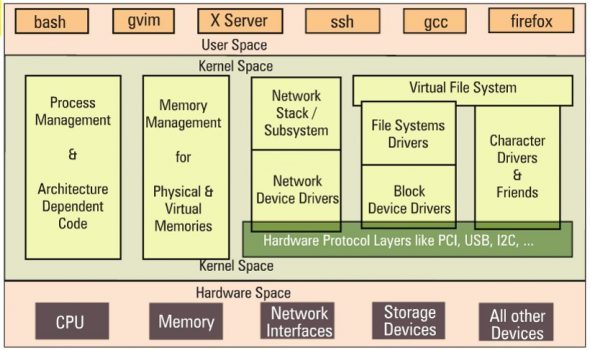
**Figure 2: Linux device driver partition**

The **device ­specific** portion of a device driver **remains the same across all operating systems**, and is more about understanding and decoding the **device data sheets** than software programming. A data sheet for a device is a document with technical details of the device, including its operation, performance, programming, etc. — in short a device user manual.

However, the **OS­ specific** portion is the one that is **tightly coupled with the OS mechanisms of user interfaces**, and thus **differentiates** a **Linux** device driver from a **Windows** device driver and from a **MacOS** device driver.

**Verticals**

In Linux, a **device driver** provides a **“system call” interface** to the **user**; this is the **boundary line** between the so­-called **kernel space** and **user­ space** of Linux, as shown in Figure 2. Figure 3 provides further classification.

** Figure 3: Linux kernel overview**

Based on the **OS­ specific interface of a driver**, in Linux, a driver is broadly classified into **three verticals**:

**1. Packet ­oriented or the network vertical  
2. Block­ oriented or the storage vertical  
3. Byte­ oriented or the character vertical**

The **CPU vertical** and **memory vertical**, taken together with the other three verticals, give the complete overview of the **Linux kernel**, like any textbook definition of an OS:

An **OS performs 5 management functions**:

1. **CPU/process,**
2. **memory,**
3. **network,**
4. **storage,**
5. **device I/O.**

Though these two verticals could be classified as device drivers, where **CPU** and  
**memory** are the respective devices, they are treated differently, for many reasons.  
These are the **core functionalities of any OS**, be it a **micro­kernel**or a **monolithic kernel**. More often than not, **adding code** in these areas is mainly a **Linux porting effort**, which is typically done for a **new CPU** or **architecture**. Moreover, the **code** in these **two verticals cannot be loaded or unloaded on the fly, unlike the other three verticals**.

Henceforth, when we talk about **Linux device drivers**, we mean to talk only about the other/**remaining three verticals: network, storage, device I/O** as in Figure 3.

Let’s get a little deeper into these three verticals.

The **network vertical** consists of two parts:

a) the **network protocol stack**, and

b) the **network interface card (NIC) device drivers**, or simply **network device drivers**, which could be for **Ethernet, Wi­Fi**, or any other network horizontals.

**Storage**, again, consists of **two parts**:

a) **File­system drivers**, to decode the various formats on different partitions, and

b) **Block device drivers** for various storage (hardware) protocols, i.e., horizontals like **IDE, SCSI, MTD**, etc.

Certainly drivers are needed for the whole lot of devices that interface with the system, and Linux does have drivers for them. However, their **byte­ oriented accessibility** puts all of them under the **character vertical** — this is, in reality, the majority bucket. In fact, because of the vast number of drivers in this vertical, **character drivers** have been further sub­-classified — so you have **tty drivers, input drivers, console drivers, frame­buffer drivers, sound drivers**, etc.

The **typical horizontals** here would be **RS232, PS/2, VGA, I2C, I2S, SPI**, etc.

**Multiple ­vertical drivers**

One final note on the complete picture (placement of all the drivers in the Linux driver ecosystem): the **horizontals** like **USB, PCI**, etc, **span below multiple verticals**. Why is that?  
Simple — you already know that you can have a **USB Wi­Fi dongle**, a **USB pen drive**, and a **USB-­to-­serial converter** — **all are USB**, but come under **three different verticals**!

In Linux, **bus drivers** or the **horizontals**, are often split into **two parts**, or even **two drivers**:

a) **device controller­ specific**, and

b) an **abstraction layer (HAL)** **over** the **device controller specific bus driver** for the **network/ storage/ device I/O** **verticals** to interface, commonly called **cores**.

A classic example would be the **USB controller drivers ohci, ehci**, etc., and the **USB abstraction, usbcore.**

**Summing up**

So, to conclude, **a device driver is a piece of software that drives a device**, though there are so many classifications. **If** **it drives only another piece of software**, we call it just a **driver**. Examples are **file­system drivers, usbcore**, etc.

Hence, **all device drivers are drivers, but all drivers are not device drivers.**

**Device Drivers, Part 2: Writing Your First Linux Driver in the Classroom**

Now, we deals with the concept of dynamically loading drivers, first writing a Linux driver, before building and then loading it.

**Dynamically loading and unloading of drivers**

A typical driver installation on Windows needs a reboot for it to get activated. That is really not acceptable; suppose we need to do it on a server? That’s where Linux wins. In Linux, we can load or unload a driver on the fly, and it is active for use instantly after loading. Also, it is instantly disabled when unloaded. This is called dynamic loading and unloading of drivers in Linux.

**Dynamically loading drivers**

These dynamically loadable drivers are more commonly called modules and built into individual files with a .ko (kernel object) extension. Every Linux system has a standard place under the root of the file system (/) for all the pre-built modules. They are organised similar to the kernel source tree structure, under /lib/modules/<kernel\_version>/kernel, where <kernel\_version> would be the output of the command uname -ron the system, as shown in Figure 4.

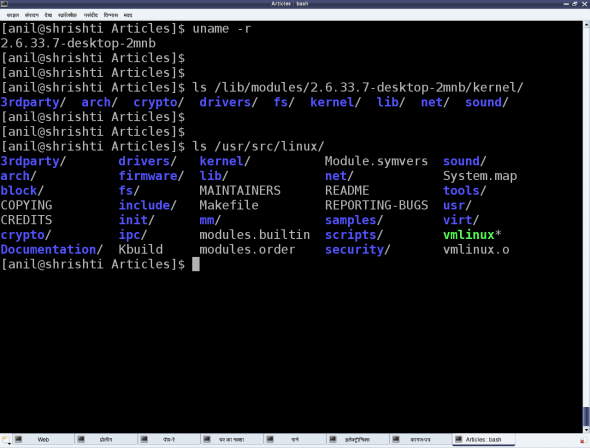


Figure 4: Linux pre-built modules

To dynamically load or unload a driver, use these commands, which reside in the /sbin directory, and must be executed with root privileges:

* lsmod — lists currently loaded modules
* insmod <module\_file> — inserts/loads the specified module file
* modprobe <module> — inserts/loads the module, along with any dependencies
* rmmod <module> — removes/unloads the module

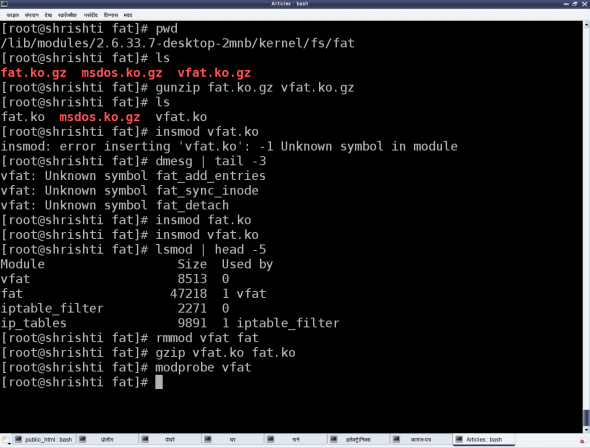


Figure 5: Linux module operations

Let’s look at the FAT filesystem-related drivers as an example. Figure 4 demonstrates this complete process of experimentation. The module files would be fat.ko, vfat.ko, etc., in the fat (vfat for older kernels) directory under /lib/modules/`uname -r`/kernel/fs. If they are in compressed .gz format, you need to uncompress them with gunzip, before you can insmod them.

The vfat module depends on the fat module, so fat.ko needs to be loaded first. To automatically perform decompression and dependency loading, use modprobe instead. Note that you shouldn’t specify the .ko extension to the module’s name, when using the modprobe command. rmmod is used to unload the modules.

**Our first Linux driver**

Before we write our first driver, let’s go over some concepts. A driver never runs by itself. It is similar to a library that is loaded for its functions to be invoked by a running application. It is written in C, but lacks a main() function. Moreover, it will be loaded/linked with the kernel, so it needs to be compiled in a similar way to the kernel, and the header files you can use are only those from the kernel sources, not from the standard /usr/include.

One interesting fact about the kernel is that it is an object-oriented implementation in C, as we will observe even with our first driver. Any Linux driver has a constructor and a destructor. The module’s constructor is called when the module is successfully loaded into the kernel, and the destructor when rmmod succeeds in unloading the module. These two are like normal functions in the driver, except that they are specified as the *init* and *exit* functions, respectively, by the macros module\_init() and module\_exit(), which are defined in the kernel header module.h.

**/\* ofd.c – Our First Driver code \*/**

**#include <linux/module.h>**

**#include <linux/version.h>**

**#include <linux/kernel.h>**

**static int \_\_init ofd\_init(void) /\* Constructor \*/**

**{   printk(KERN\_INFO " Constructor : Namaskar: ofd registered");**

**return 0;**

**}**

**static void \_\_exit ofd\_exit(void) /\* Destructor \*/**

**{    printk(KERN\_INFO " Destructor : Alvida: ofd unregistered");**

**}**

**module\_init(ofd\_init);**

**module\_exit(ofd\_exit);**

**MODULE\_LICENSE("GPL");**

**MODULE\_AUTHOR("Anil Kumar Pugalia ");**

**MODULE\_DESCRIPTION("Our First Driver");**

Given above is the complete code for our first driver; let’s call it ofd.c. Note that there is no stdio.h (a user-space header); instead, we use the analogous kernel.h (a kernel space header). printk() is the equivalent of printf(). Additionally, version.h is included for the module version to be compatible with the kernel into which it is going to be loaded. The MODULE\_XYZ macros populate module-related information, which acts like the module’s “signature”.

**Building our first Linux driver**

Once we have the C code, it is time to compile it and create the module file ofd.ko. We use the kernel build system to do this. The following Makefile invokes the kernel’s build system from the kernel source, and the kernel’s Makefile will, in turn, invoke our first driver’s Makefile to build our first driver.

To build a Linux driver, you need to have the kernel source (or, at least, the kernel headers) installed on your system. The kernel source is assumed to be installed at /usr/src/linux. If it’s at any other location on your system, specify the location in the KERNEL\_SOURCE variable in this Makefile.

**# Makefile – makefile of our first driver**

**# if KERNELRELEASE is defined, we've been invoked from the**

**# kernel build system and can use its language.**

**ifneq (${KERNELRELEASE},)**

**obj-m := ofd.o**

**# Otherwise we were called directly from the command line.**

**# Invoke the kernel build system.**

**else**

**KERNEL\_SOURCE := /usr/src/linux**

**PWD := $(shell pwd)**

**default:**

**${MAKE} -C ${KERNEL\_SOURCE} SUBDIRS=${PWD} modules**

**clean:**

**${MAKE} -C ${KERNEL\_SOURCE} SUBDIRS=${PWD} clean**

**endif**

**With the C** code (ofd.c) and Makefile ready, all we need to do is invoke make to build our first driver (ofd.ko).

|  |
| --- |
| $ **make**  make -C /usr/src/linux SUBDIRS=... modules  make[1]: Entering directory `/usr/src/linux'    CC [M]  .../ofd.o    Building modules, stage 2.    MODPOST 1 modules    CC      .../ofd.mod.o    LD [M]  .../ofd.ko  make[1]: Leaving directory `/usr/src/linux' |

**Summing up**

Once we have the ofd.ko file, perform the usual steps as the root user, or with sudo.

|  |
| --- |
| **# su**  **# insmod ofd.ko**  **# lsmod | head -10** |

lsmod should show you the ofd driver loaded.

Currently, you may not be able to observe anything other than the lsmod listing showing the driver has loaded. Where’s the printk output gone? Find that out for yourselves. Also note that our first driver is a template for any driver you would write in Linux. Writing a specialised driver is just a matter of what gets filled into its constructor and destructor. So, our further learning will be to enhance this driver to achieve specific driver functionalities.

**Device Drivers, Part 3: Kernel C Extras in a Linux Driver**

Here we deals with the kernel’s message logging, and kernel-specific GCC extensions.

Recall the error output demonstration from insmod vfat.ko — running dmesg | tail and to find the printk output there.

**Kernel message logging**

As far as parameters are concerned, printf and printk are the same, except that when programming for the kernel, we don’t bother about the float formats %f, %lf and the like. However, unlike printf, printk is not designed to dump its output to some console.

In fact, it cannot do so; it is something in the background, and executes like a library, only when triggered either from hardware-space or user-space. All printk calls put their output into the (log) ring buffer of the kernel. Then, the syslog daemon running in user-space picks them up for final processing and redirection to various devices, as configured in the configuration file /etc/syslog.conf.

You must have observed the out-of-place macro KERN\_INFO, in the printk calls, in the [last article](https://www.opensourceforu.com/2010/12/writing-your-first-linux-driver/). That is actually a constant string, which gets concatenated with the format string after it, into a single string. Note that there is no comma (,) between them; they are not two separate arguments. There are eight such macros defined in linux/kernel.h in the kernel source, namely:

|  |
| --- |
| #define KERN\_EMERG "<0>"   /\* system is unusable                \*/  #define KERN\_ALERT "<1>"   /\* action must be taken immediately    \*/  #define KERN\_CRIT "<2>"    /\* critical conditions     \*/  #define KERN\_ERR "<3>"     /\* error conditions            \*/  #define KERN\_WARNING "<4>" /\* warning conditions      \*/  #define KERN\_NOTICE "<5>"  /\* normal but significant condition    \*/  #define KERN\_INFO "<6>"    /\* informational           \*/  #define KERN\_DEBUG "<7>"   /\* debug-level messages        \*/ |

Now depending on these log levels (i.e., the first three characters in the format string), the syslog user-space daemon redirects the corresponding messages to their configured locations. A typical destination is the log file /var/log/messages, for all log levels. Hence, all the printk outputs are, by default, in that file. However, they can be configured differently — to a serial port (like /dev/ttyS0), for instance, or to all consoles, like what typically happens for KERN\_EMERG.

Now, /var/log/messages is buffered, and contains messages not only from the kernel, but also from various daemons running in user-space. Moreover, this file is often not readable by a normal user. Hence, a user-space utility, dmesg, is provided to directly parse the kernel ring buffer, and dump it to standard output.

**Kernel-specific GCC extensions**

\_\_init, \_\_exit are not special keywords. Kernel C is not “weird C”, but just standard C with some additional extensions from the C compiler, GCC. Macros \_\_init and \_\_exit are just two of these extensions. However, these do not have any relevance in case we are using them for a dynamically loadable driver, but only when the same code gets built into the kernel. All functions marked with \_\_init get placed inside the init section of the kernel image automatically, by GCC, during kernel compilation; and all functions marked with \_\_exit are placed in the exit section of the kernel image.

What is the benefit of this? All functions with \_\_init are supposed to be executed only once during bootup (and not executed again till the next bootup). So, once they are executed during bootup, the kernel frees up RAM by removing them (by freeing the init section). Similarly, all functions in the exit section are supposed to be called during system shutdown.

Now, if the system is shutting down anyway, why do you need to do any cleaning up? Hence, the exit section is not even loaded into the kernel — another cool optimisation. This is a beautiful example of how the kernel and GCC work hand-in-hand to achieve a lot of optimisation, and many other tricks that we will see as we go along. And that is why the Linux kernel can only be compiled using GCC-based compilers — a closely knit bond.

**The kernel function’s return guidelines**

Any kernel function needing error handling, typically returns an integer-like type — and the return value again follows a guideline. For an error, we return a negative number: a minus sign appended with a macro that is available through the kernel header linux/errno.h, that includes the various error number headers under the kernel sources — namely, asm/errno.h, asm-generic/errno.h, asm-generic/errno-base.h.

For success, zero is the most common return value, unless there is some additional information to be provided. In that case, a positive value is returned, the value indicating the information, such as the number of bytes transferred by the function.

**Kernel C = pure C**

No /usr/include headers can be used for kernel programming. But kernel C is just standard C with some GCC extensions. Why this conflict?

Actually this is not a conflict. Standard C is pure C — just the language. The headers are not part of it. Those are part of the standard libraries built in for C programmers, based on the concept of reusing code.

Does that mean that all standard libraries, and hence, all ANSI standard functions, are not part of “pure” C? Yes, that’s right. Then, was it really tough coding the kernel?

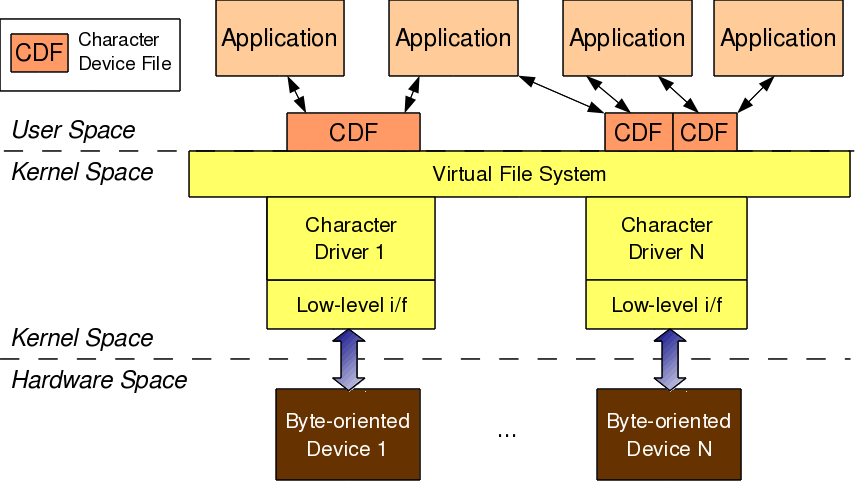
Well, not for this reason. In reality, kernel developers have evolved their own set of required functions, which are all part of the kernel code. The printk function is just one of them. Similarly, many string functions, memory functions, and more, are all part of the kernel source, under various directories like kernel, ipc, lib, and so on, along with the corresponding headers under the include/linux directory. That is why we need to have the kernel source to build a driver. If not the complete source, at least the headers are a must. And that is why we have separate packages to install the complete kernel source, or just the kernel headers.

In Fedora, the kernel sources are typically installed under /usr/src/kernels/<kernel-version>, unlike the standard /usr/src/linux, installed using the command-line yum install kernel-devel. For Ubuntu, just use apt-get utility to fetch the source — possibly apt-get install linux-source.

**Device Drivers, Part 4: Linux Character Drivers**

If we write drivers for byte-oriented operations (or, in C lingo, character-oriented operations), then we refer to them as character drivers. Since the majority of devices are byte-oriented, the majority of device drivers are character device drivers.

For example, **serial drivers, audio drivers, video drivers, camera drivers, and basic I/O drivers**. **All device drivers that are neither storage nor network device drivers are some type of a character driver**.



For any user-space application to operate on a **byte-oriented device (in hardware space),** it should use the corresponding **character device driver (in kernel space).** Character driver usage is done through the corresponding **character device file(s),** linked to it through the **virtual file system (VFS**). What this means is that an application does the usual file operations on the character device file. Those operations are translated to the corresponding functions in the linked character device driver by the VFS. Those functions then do the final low-level access to the actual device to achieve the desired results.

Note that though the application does the usual file operations, their outcome may not be the usual ones. Rather, they would be as driven by the corresponding functions in the device driver. For example, a write followed by a read may not fetch what has just been written to the character device file, unlike for regular files. Remember that this is the usual expected behaviour for device files. Let’s take an audio device file as an example. What we write into it is the audio data we want to play back, say through a speaker. However, the read would get us audio data that we are recording, say through a microphone. The recorded data need not be the played-back data.

In this complete connection from the application to the device, there are four major entities involved:

1. **Application (User space)**
2. **Character device file (CDF) (User space -> VFS)**
3. **Character device driver (Kernel space)**
4. **Character device (Byte-oriented device in hardware space)**

The interesting thing is that all of these can exist independently on a system, without the other being present. The mere existence of these on a system doesn’t mean they are linked to form the complete connection. Rather, they need to be explicitly connected. An **application** gets connected to a **device file** by invoking the **open system call** on the device file.

Device file(s) are linked to the device driver by specific **registrations** done by the driver. The driver is linked to a device by its **device-specific low-level operations**. Thus we form the complete connection. Note that the **character device file** is not the actual device, but **just a place-holder for the actual device**.

## Major and minor numbers

The connection between the application and the device file is based on the name of the device file. However, the connection between the device file and the device driver is based on the number of the device file, not the name. This allows a user-space application to have any name for the device file, and enables the kernel-space to have a trivial index-based linkage between the device file and the device driver. This device file number is more commonly referred to as the <major, minor> pair, or the major and minor numbers of the device file.

Earlier (till kernel 2.4), one major number was for one driver, and the minor number used to represent the sub-functionalities of the driver. With kernel 2.6, this distinction is no longer mandatory; there could be multiple drivers under the same major number, but obviously, with different minor number ranges.

However, this is more common with the non-reserved major numbers, and standard major numbers are typically preserved for single drivers. For example, 4 for serial interfaces, 13 for mice, 14 for audio devices, and so on. The following command would list the various character device files on your system:

|  |
| --- |
| **$ ls -l /dev/ | grep "^c"** |

## <major, minor> related support in kernel 2.6

Type (defined in kernel header **linux/types.h**):

* **dev\_t contains both major and minor numbers**

Macros (defined in kernel header **linux/kdev\_t.h**):

* **MAJOR(dev\_t dev) extracts the major number from dev**
* **MINOR(dev\_t dev) extracts the minor number from dev**
* **MKDEV(int major, int minor) creates the dev from major and minor.**

Connecting the device file with the device driver involves two steps:

1. Registering for the <major, minor> range of device files.
2. Linking the device file operations to the device driver functions.

The first step is achieved using either of the following two APIs, defined in the kernel header **linux/fs.h**:

|  |
| --- |
| **+ int register\_chrdev\_region(dev\_t first, unsigned int cnt, char \*name);**  **+ int alloc\_chrdev\_region(dev\_t \*first, unsigned int firstminor, unsigned int cnt, char \*name);** |

**The first API registers the cnt number of device file numbers, starting from first, with the given name.**

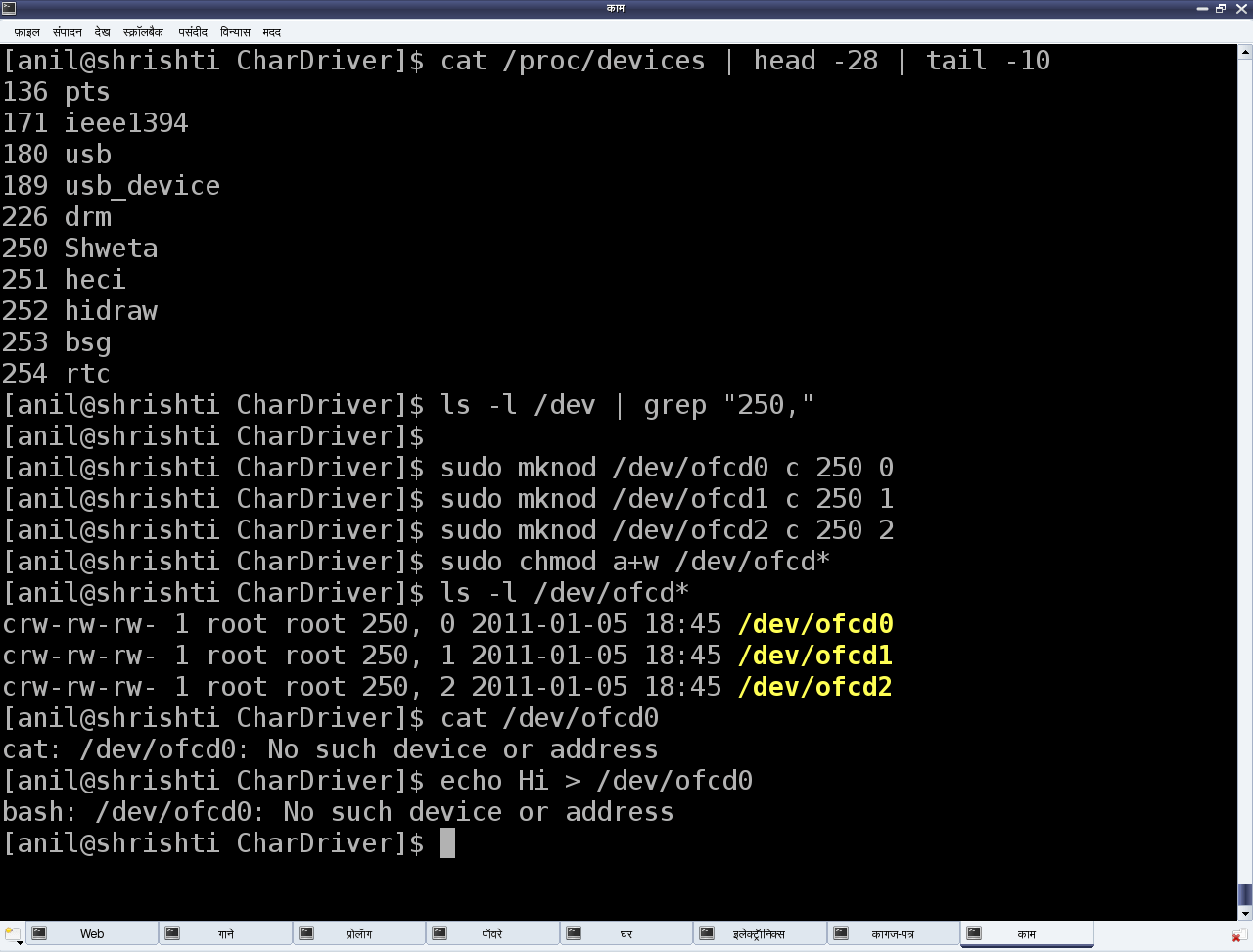
**The second API dynamically figures out a free major number, and registers the cnt number of device file numbers starting from <the free major, firstminor>, with the given name.**

In either case, the /proc/devices kernel window lists the name with the registered major number.

|  |
| --- |
| **#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 charDriver\_init(void) /\* Constructor \*/**  **{**  **printk(KERN\_INFO "Initiating: Our char Driver registered");**  **if (alloc\_chrdev\_region(&first, 0, 3, "charDriverFile") < 0)**  **{     return -1;**  **}**  **printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first));**  **return 0;**  **}**    **static void \_\_exit charDriver\_exit(void) /\* Destructor \*/**  **{**  **unregister\_chrdev\_region(first, 3);**  **printk(KERN\_INFO "Exiting: char Driver unregistered");**  **}**    **module\_init(charDriver\_init);**  **module\_exit(charDriver\_exit);**    **MODULE\_LICENSE("GPL");**  **MODULE\_AUTHOR("D Maitra");**  **MODULE\_DESCRIPTION("Our First Character Driver");** |

* Build the driver (.ko file) by running make.
* Load the driver using insmod.
* List the loaded modules using lsmod.
* Unload the driver using rmmod.

Before unloading the driver, looking into the /proc/devices kernel window shows the registered major number with the name “Shweta”, using cat /proc/devices. Device file can be created under /dev with the same major number, using mknod, and then can be read and written to.



The major number 250 may vary from system to system, based on availability. Figure above also shows the results from reading and writing one of the device files. The second step to connect the device file with the device driver — which is linking the device file operations to the device driver functions — is not yet done.

**Device Drivers, Part 5: Character Device Files — Creation & Operations**

Even with the registration for the <major, minor> device range, the device files are not created under /dev — instead, have to be created them manually, using mknod. To automatically create the device files, the udev daemon is used. The second step to connect the device file with the device driver is to link the device file operations to the device driver functions.

**Automatic creation of device files**

Earlier, in kernel 2.4, the automatic creation of device files was done by the kernel itself, by calling the appropriate APIs of devfs. However, as the kernel evolved, kernel developers realised that device files were more related to user-space and hence, as a policy, that is where they ought to be dealt with, not at the kernel. Based on this idea, the kernel now only populates the appropriate device class and device information into the /sys window, for the device under consideration. User-space then needs to interpret it and take appropriate action. In most Linux desktop systems, **the udev daemon picks up that information, and accordingly creates the device files.**

udev can be further configured via its configuration files to tune the device file names, their permissions, their types, etc. So, as far as the driver is concerned, the **appropriate /sys entries need to be populated using the Linux device model APIs declared in <linux/device.h>.** The rest should be handled by udev. The **device class** is created as follows:

|  |
| --- |
| **struct class \*cl = class\_create(THIS\_MODULE, "<device class name>");** |

Then, the **device info (<major, minor>)** under this class is populated by:

|  |
| --- |
| **device\_create(cl, NULL, first, NULL, "<device name format>", ...);** |

Here, the first is dev\_t with the corresponding <major, minor>. The corresponding complementary or the inverse calls, which should be called in chronologically reverse order, are as follows:

|  |
| --- |
| **device\_destroy(cl, first);**  **class\_destroy(cl);** |

Refer to Figure below for the /sys entries created using chardrv as the <device class name> and mynull as the <device name format>. That also shows the device file, created by udev, based on the <major>:<minor> entry in the dev file.

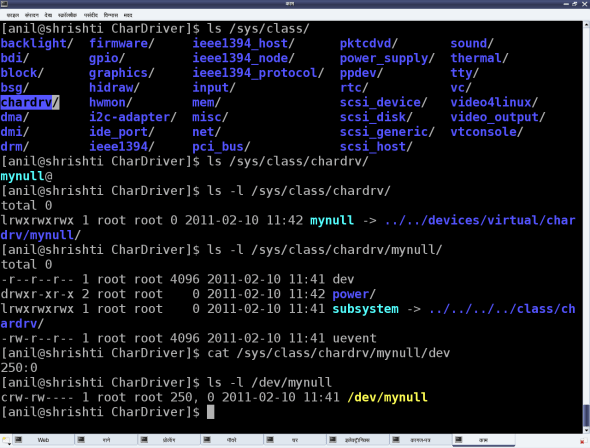
[](https://i1.wp.com/www.opensourceforu.com/wp-content/uploads/2011/04/figure_1_auto_dev_file_creation.png?ssl=1)

Figure 1: Automatic device file creation

In case of multiple minors, the device\_create() and device\_destroy() APIs may be put in the for loop, and the <device name format> string could be useful. For example, the device\_create() call in a for loop indexed by i could be as follows:

|  |
| --- |
| **device\_create(cl, NULL, MKNOD(MAJOR(first), MINOR(first) + i), NULL, "mynull%d", i);** |

**File operations**

Whatever system calls (or, more commonly, file operations) we talk of on a regular file, are applicable to device files as well. That’s what we say: a file is a file, and **in Linux, almost everything is a file from the user-space perspective**. The difference lies in **the kernel space**, where the **virtual file system (VFS)** decodes the **file type** and transfers the file operations to the appropriate channel, like a filesystem module in case of a regular file or directory, and the corresponding **device driver** in case of a **device file**. Our discussion focuses on the second case.

Now, for VFS to pass the device file operations onto the driver, it should have been informed about it. And yes, that is what is called registering the file operations by the driver with the VFS. This involves two steps. (The parenthesised code refers to the “null driver” code below.)

First, let’s fill in a file operations structure (struct file\_operations pugs\_fops) with the desired file operations (my\_open, my\_close, my\_read, my\_write, …) and initialise the character device structure (struct cdev c\_dev) with that, using cdev\_init().

Then, hand this structure to the VFS using the call cdev\_add(). Both cdev\_init() and cdev\_add() are declared in <linux/cdev.h>. Obviously, the actual file operations (my\_open, my\_close, my\_read, my\_write) also had to be coded.

So, to start with, let’s keep them as simple as possible — let’s say, as easy as the “null driver”.

**The null driver**

Following these steps, first character device driver complete code — **ofcd.c:**

**#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>**

**#include <linux/device.h>**

**#include <linux/cdev.h>**

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

**static struct cdev c\_dev; // Global variable for the character device structure**

**static struct class \*cl; // Global variable for the device class**

**static int my\_open(struct inode \*i, struct file \*f)**

**{**

**printk(KERN\_INFO "Driver: open()\n");**

**return 0;**

**}**

**static int my\_close(struct inode \*i, struct file \*f)**

**{**

**printk(KERN\_INFO "Driver: close()\n");**

**return 0;**

**}**

**static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t**

**len, loff\_t \*off)**

**{**

**printk(KERN\_INFO "Driver: read()\n");**

**return 0;**

**}**

**static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf,**

**size\_t len, loff\_t \*off)**

**{**

**printk(KERN\_INFO "Driver: write()\n");**

**return len;**

**}**

**static struct file\_operations my\_fops =**

**{**

**.owner = THIS\_MODULE,**

**.open = my\_open,**

**.release= my\_close,**

**.read = my\_read,**

**.write = my\_write**

**};**

**static int \_\_init ofcd\_init(void) /\* Constructor \*/**

**{**

**printk(KERN\_INFO "Namaskar: ofcd registered");**

**if (alloc\_chrdev\_region(&first, 0, 1, "Shweta") < 0)**

**{**

**return -1;**

**}**

**if ((cl = class\_create(THIS\_MODULE, "chardrv")) == NULL)**

**{**

**unregister\_chrdev\_region(first, 1);**

**return -1;**

**}**

**if (device\_create(cl, NULL, first, NULL, "mynull") == NULL)**

**{**

**class\_destroy(cl);**

**unregister\_chrdev\_region(first, 1);**

**return -1;**

**}**

**cdev\_init(&c\_dev, &my\_fops);**

**if (cdev\_add(&c\_dev, first, 1) == -1)**

**{**

**device\_destroy(cl, first);**

**class\_destroy(cl);**

**unregister\_chrdev\_region(first, 1);**

**return -1;**

**}**

**return 0;**

**}**

**static void \_\_exit ofcd\_exit(void) /\* Destructor \*/**

**{**

**cdev\_del(&c\_dev);**

**device\_destroy(cl, first);**

**class\_destroy(cl);**

**unregister\_chrdev\_region(first, 1);**

**printk(KERN\_INFO "Alvida: ofcd unregistered");**

**}**

**module\_init(ofcd\_init);**

**module\_exit(ofcd\_exit);**

**MODULE\_LICENSE("GPL");**

**MODULE\_AUTHOR("D M");**

**MODULE\_DESCRIPTION("Our First Character Driver");**

Shweta repeated the usual build process, with some new test steps, as follows:

1. Build the driver (.ko file) by running make.
2. Load the driver using insmod.
3. List the loaded modules using lsmod.
4. List the major number allocated, using cat /proc/devices.
5. “null driver”-specific experiments (refer to Figure 2 for details).
6. Unload the driver using rmmod.

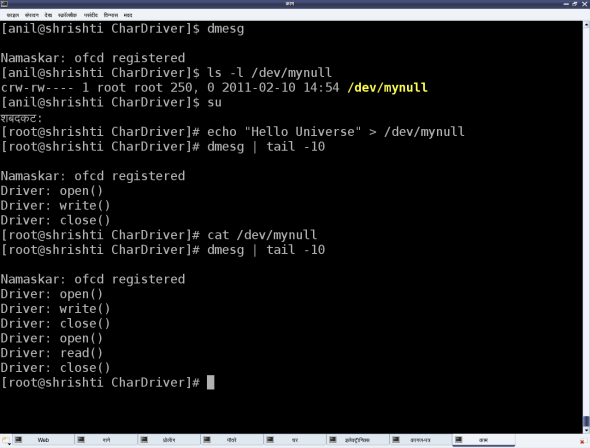
[](https://i2.wp.com/www.opensourceforu.com/wp-content/uploads/2011/04/figure_2_null_driver_experiments.png?ssl=1)

Figure: 'null driver' experiments

**Summing up**

Shweta was certainly happy; all on her own, she’d got a character driver written, which works the same as the standard /dev/null device file. To understand what this means, check the <major, minor> tuple for /dev/null, and similarly, also try out the echo and cat commands with it.

However, one thing began to bother Shweta. She had got her own calls (my\_open, my\_close, my\_read, my\_write) in her driver, but wondered why they worked so unusually, unlike any regular file system calls