## Part1:

## 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.

Further, a “pilot” could be a person or even an automatic system monitored by a person (an auto-pilot system in airliners, for example). Similarly, 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.

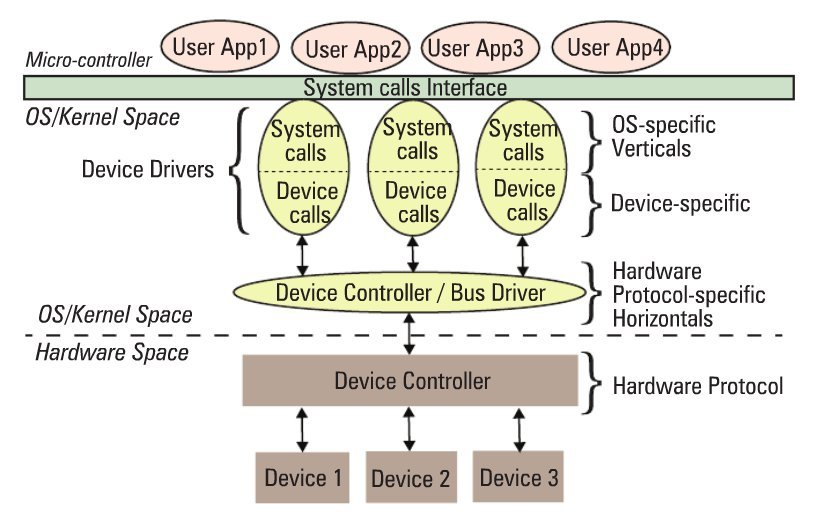
General examples of device controllers include hard disk controllers, display controllers, and audio controllers that in turn manage devices connected to them. More technical examples would be an IDE controller, PCI controller, USB controller, SPI controller, I2C controller, etc. Pictorially, this whole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

Device controllers are typically connected to the 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 encounter 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. 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 do provide an interface (which varies 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.

he 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. 

Later, I shall show some examples of decoding data sheets as well. 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

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

* Packet-oriented or the network vertical
* Block-oriented or the storage vertical
* 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: CPU/process, memory, network, storage, 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 micro-kernel 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 latter three verticals 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, Wi-Fi, or any other network horizontals. Storage, again, consists of two parts: a) File-system 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.

With this, you may wonder if that is the only set of devices for which you need drivers (or for which Linux has drivers). Hold on a moment; you certainly need drivers for the whole lot of devices that interface with the system, and Linux does have drivers for them. However, their byte-oriented cessibility 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, frame-buffer 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 Wi-Fi 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 over that for the verticals to interface, commonly called cores. A classic example would be the USB controller drivers ohci, ehci, etc., and the USB abstraction, usbcore.

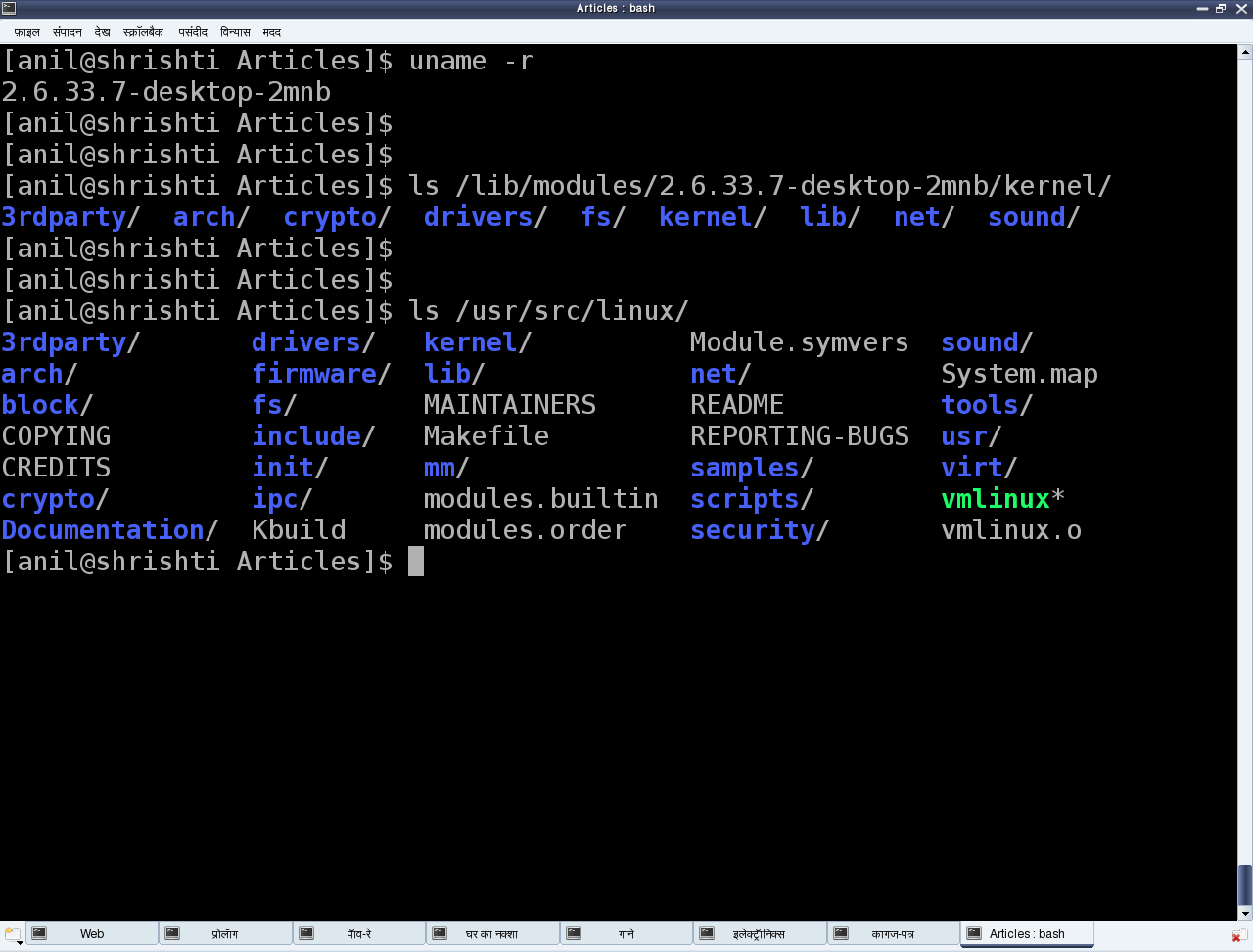
## Summing up

So, to conclude, a device driver is a piece of software that drives a device, though there are so many classifications. In case it drives only another piece of software, we call it just a driver. Examples are file-system drivers, usbcore, etc. Hence, all device drivers are drivers, but all drivers are not device drivers.

## Part 2:

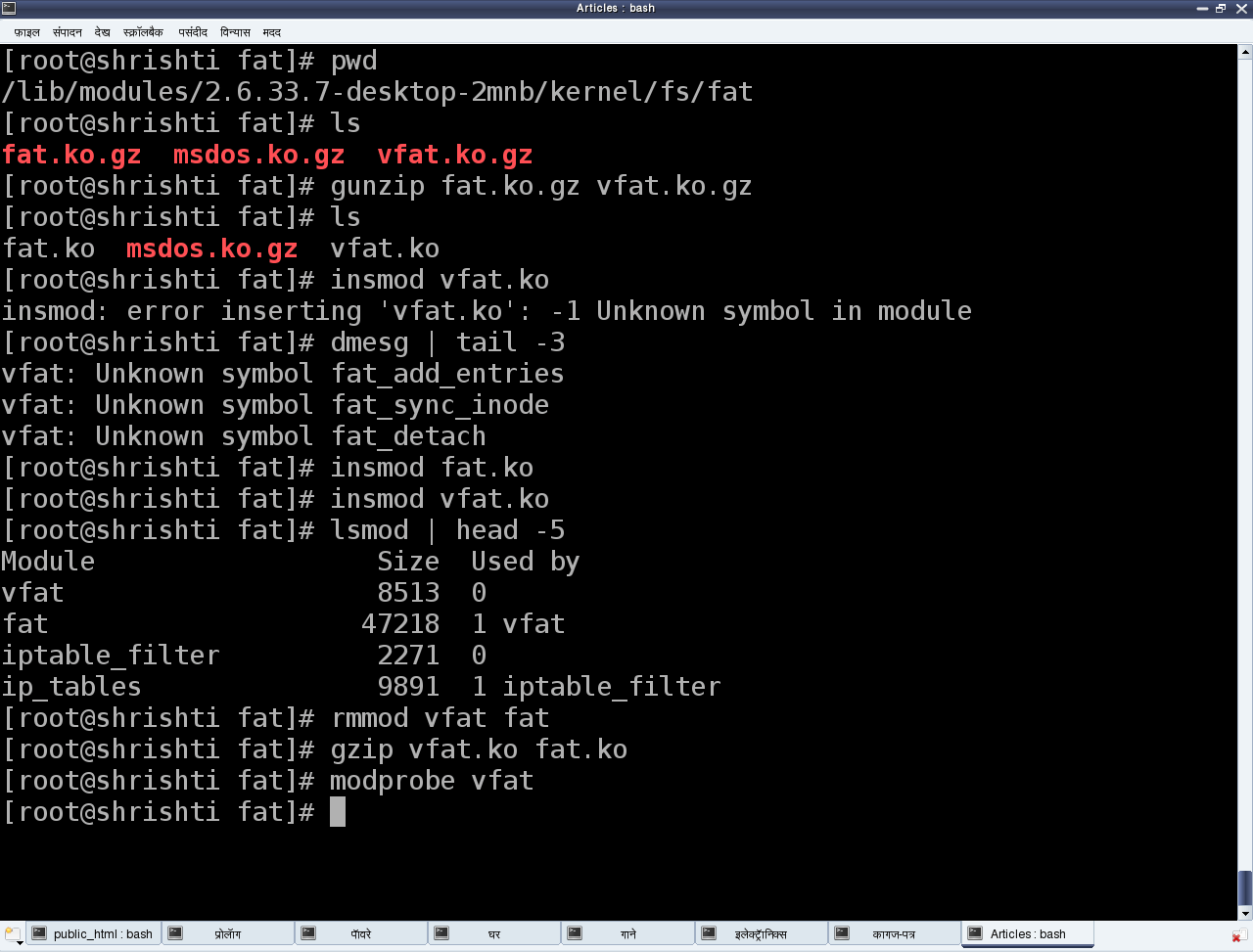
## Dynamically loading drivers

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 1.

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

Let’s look at the FAT filesystem-related drivers as an example. Figure 2 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 insmodthem.

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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | /\* 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 "Namaskar: ofd registered");      return 0;  }    static void \_\_exit ofd\_exit(void) /\* Destructor \*/  {      printk(KERN\_INFO "Alvida: ofd unregistered");  }    module\_init(ofd\_init);  module\_exit(ofd\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  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\_\* 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.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | # 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.

While the students were trying their first module, the bell rang, marking the end of the session. Professor Gopi concluded, “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, in the lab session, and update me with your findings. 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.

## Part 3:

## Kernel message logging

Over coffee, Pugs began his explanation.

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](http://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. Figure 1 shows snippets from the two.

## Kernel-specific GCC extensions

Shweta, frustrated since she could no longer show off as having discovered all these on her own, retorted, “Since you have explained all about printing in the kernel, why don’t you also tell me about the weird C in the driver as well — the special keywords \_\_init, \_\_exit, etc.”

These 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

While returning from coffee, Pugs kept praising OSS and the community that’s grown around it. Do you know why different individuals are able to come together and contribute excellently without any conflicts, and in a project as huge as Linux, at that? There are many reasons, but most important amongst them is that they all follow and abide by inherent coding guidelines.

Take, for example, the kernel programming guideline for returning values from a function. 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

Once back in the lab, Shweta remembered their professor mentioning that no /usr/include headers can be used for kernel programming. But Pugs had said that 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.

“Oh yes! That is why we need to have the kernel source to build a driver,” agreed Shweta.

“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,” added Pugs.

“In the lab, all the sources are set up. But if I want to try out drivers on my Linux system in my hostel room, how do I go about it?” asked Shweta.

“Our lab has Fedora, where the kernel sources are typically installed under /usr/src/kernels/<kernel-version>, unlike the standard /usr/src/linux. Lab administrators must have installed it using the command-line yum install kernel-devel. I use Mandriva, and installed the kernel sources using urpmi kernel-source,” replied Pugs.

“But I have Ubuntu,” Shweta said.

“Okay! For that, just use apt-get utility to fetch the source — possibly apt-get install linux-source,” replied Pugs.

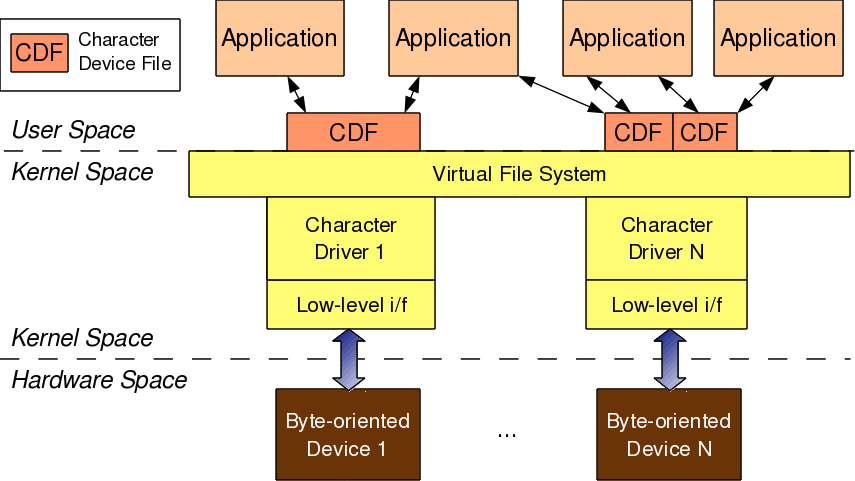
## Part 4:

## W’s of character drivers

We already know what drivers are, and why we need them. What is so special about 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.

Take, for example, serial drivers, audio drivers, video drivers, camera drivers, and basic I/O drivers. In fact, all device drivers that are neither storage nor network device drivers are some type of a character driver. Let’s look into the commonalities of these character drivers, and how Shweta wrote one of them.

## The complete connection

As shown in Figure 1, 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
2. Character device file
3. Character device driver
4. Character device

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. With this, 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. With this information, Shweta added the following into the first driver code:

|  |
| --- |
| #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>    static dev\_t first; // Global variable for the first device number |

In the constructor, she added:

|  |
| --- |
| if (alloc\_chrdev\_region(&first, 0, 3, "Shweta") < 0)  {      return -1;  }  printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first)); |

In the destructor, she added:

|  |
| --- |
| unregister\_chrdev\_region(first, 3); |

It’s all put together, as follows:

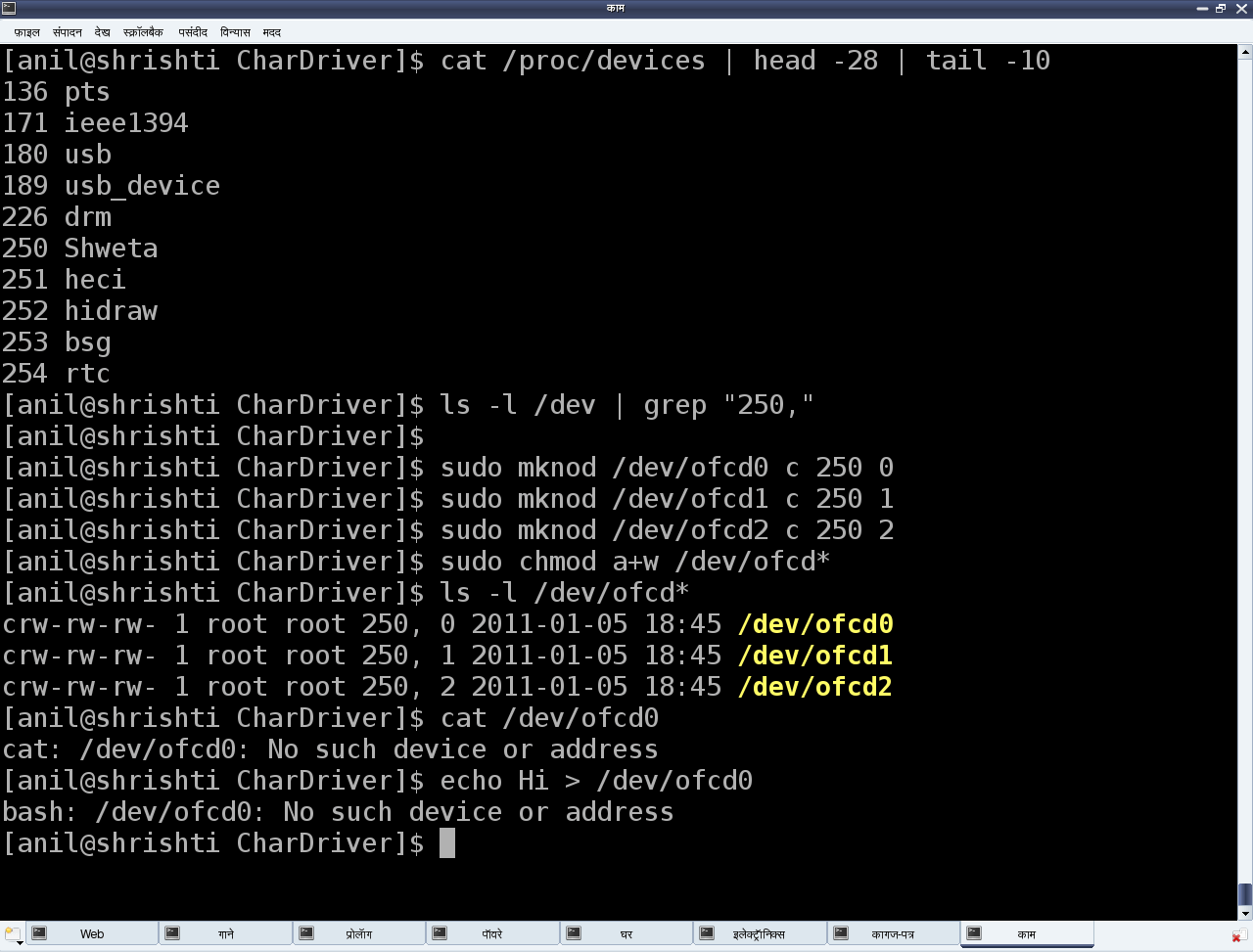
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32 | #include <linux/module.h>  #include <linux/version.h>  #include <linux/kernel.h>  #include <linux/types.h>  #include <linux/kdev\_t.h>  #include <linux/fs.h>    static dev\_t first; // Global variable for the first device number    static int \_\_init ofcd\_init(void) /\* Constructor \*/  {      printk(KERN\_INFO "Namaskar: ofcd registered");      if (alloc\_chrdev\_region(&first, 0, 3, "Shweta") < 0)      {          return -1;      }      printk(KERN\_INFO "<Major, Minor>: <%d, %d>\n", MAJOR(first), MINOR(first));      return 0;  }    static void \_\_exit ofcd\_exit(void) /\* Destructor \*/  {      unregister\_chrdev\_region(first, 3);      printk(KERN\_INFO "Alvida: ofcd unregistered");  }    module\_init(ofcd\_init);  module\_exit(ofcd\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Our First Character Driver"); |

Then, Shweta repeated the usual steps that she’d learnt for the first 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.

## Summing up

Additionally, before unloading the driver, she peeped into the /proc/devices kernel window to look for the registered major number with the name “Shweta”, using cat /proc/devices. It was right there. However, she couldn’t find any device file created under /dev with the same major number, so she created them by hand, using mknod, and then tried reading and writing those. Figure 2 shows all these steps.

Please note that the major number 250 may vary from system to system, based on availability. Figure 2 also shows the results Shweta got from reading and writing one of the device files. That reminded her that the second step to connect the device file with the device driver — which is linking the device file operations to the device driver functions — was not yet done. She realised that she needed to dig around for more information to complete this step, and also to figure out the reason for the missing device files under /dev

## Part 5:

## 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>"); |

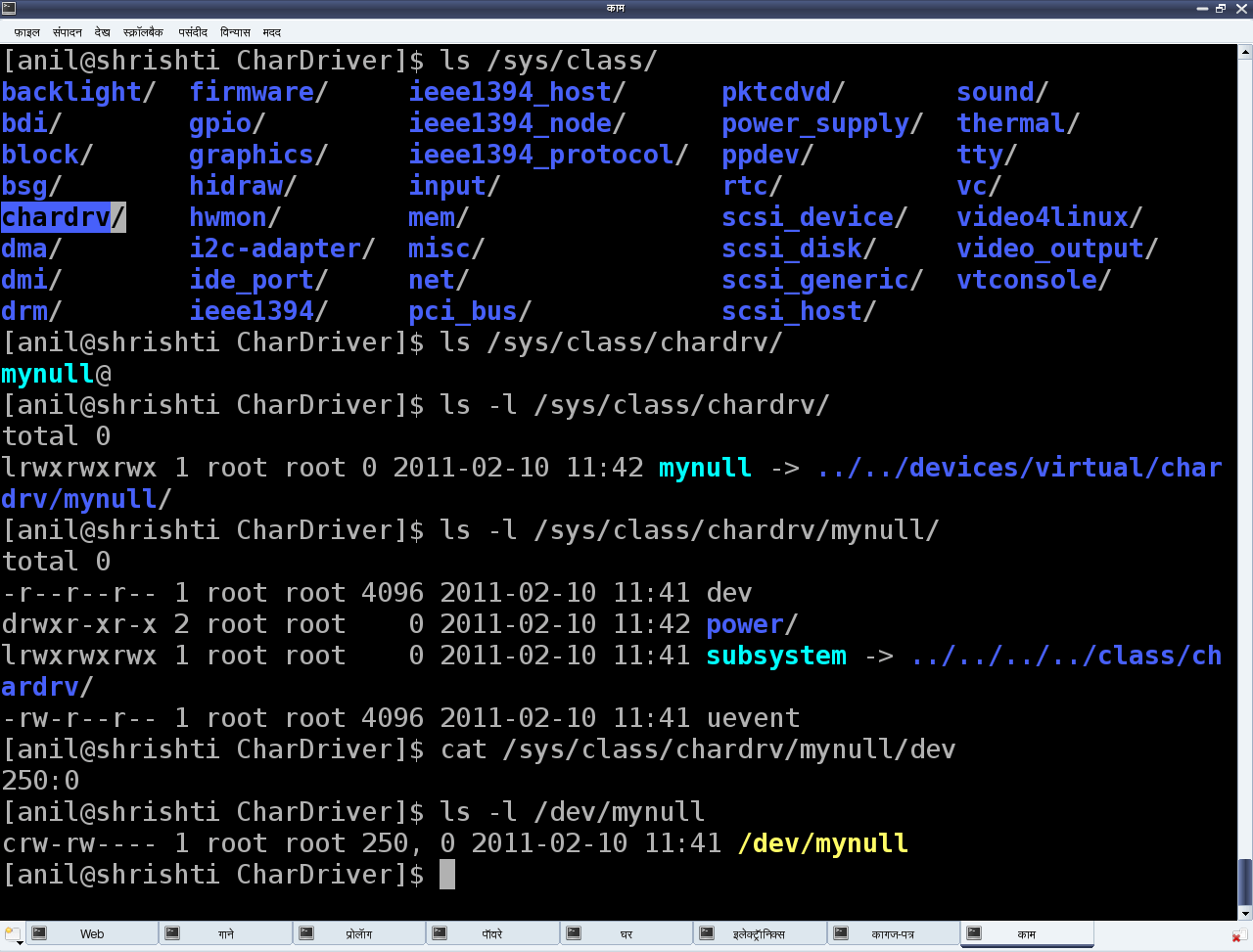
Then, the device info (<major, minor>) under this class is populated by:

|  |
| --- |
| 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 1 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.

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”.

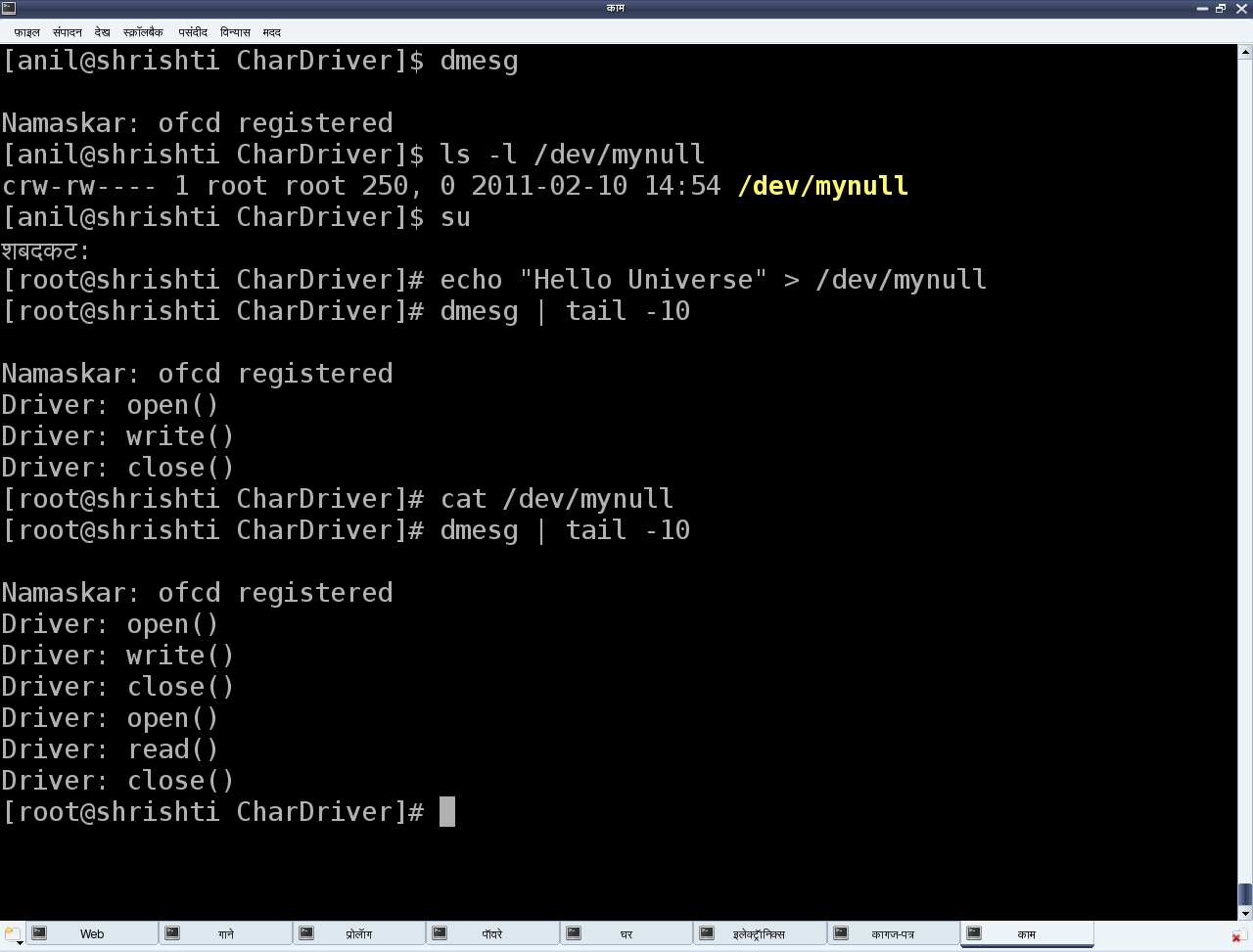
## The null driver

Following these steps, Shweta put the pieces together, attempting her first character device driver. Let’s see what the outcome was. Here’s the complete code — ofcd.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86 | #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 pugs\_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, &pugs\_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("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  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.

Shweta felt that was a good idea. She tail‘ed the dmesg log to observe the printk output from her driver. Alongside, she opened her null driver code on her console, specifically observing the device file operations my\_open, my\_close, my\_read, and my\_write.

|  |
| --- |
| 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;  } |

Based on the earlier understanding of the return value of the functions in the kernel, my\_open() and my\_close() are trivial, their return types being int, and both of them returning zero, means success.

However, the return types of both my\_read() and my\_write() are not int, rather, it is ssize\_t. On further digging through kernel headers, that turns out to be a signed word. So, returning a negative number would be a usual error. But a non-negative return value would have additional meaning. For the read operation, it would be the number of bytes read, and for the write operation, it would be the number of bytes written.

## Part 6:

## Reading the device file

To understand this in detail, the complete flow has to be given a relook. Let’s take the read operation first. When the user does a read from the device file /dev/mynull, that system call comes to the virtual file system (VFS) layer in the kernel. VFS decodes the <major, minor> tuple, and figures out that it needs to redirect it to the driver’s function my\_read(), that’s registered with it. So from that angle, my\_read() is invoked as a request to read, from us — the device-driver writers. And hence, its return value would indicate to the requesters (i.e., the users), how many bytes they are getting from the read request.

In our null driver example, we returned zero — which meant no bytes available, or in other words, the end of the file. And hence, when the device file is being read, the result is always nothing, independent of what is written into it.

“Hmmm… So, if I change it to 1, would it start giving me some data?” asked Pugs, by way of verifying.

Shweta paused for a while, looked at the parameters of the function my\_read() and answered in the affirmative, but with a caveat — the data sent would be some junk data, since my\_read() is not really populating data into buf (the buffer variable that is the second parameter of my\_read(), provided by the user). In fact, my\_read() should write data into buf, according to len (the third parameter to the function), the count in bytes requested by the user.

To be more specific, it should write less than, or equal to, len bytes of data into buf, and the number of bytes written should be passed back as the return value. No, this is not a typo — in the read operation, device-driver writers “write” into the user-supplied buffer. We read the data from (possibly) an underlying device, and then write that data into the user buffer, so that the user can read it. “That’s really smart of you,” said Pugs, sarcastically.

## Writing into the device file

The write operation is the reverse. The user provides len (the third parameter of my\_write()) bytes of data to be written, in buf (the second parameter of my\_write()). The my\_write() function would read that data and possibly write it to an underlying device, and return the number of bytes that have been successfully written.

“Aha!! That’s why all my writes into /dev/ mynull have been successful, without actually doing any read or write,” exclaimed Shweta, filled with happiness at understanding the complete flow of device file operations.

## Preserving the last character

With Shweta not giving Pugs any chance to correct her, he came up with a challenge. “Okay. Seems like you are thoroughly clear with the read/write fundamentals; so, here’s a question for you. Can you modify these my\_read() and my\_write() functions such that whenever I read /dev/mynull, I get the last character written into /dev/mynull?”

Confidently, Shweta took on the challenge, and modified my\_read() and my\_write() as follows, adding a static global character variable:

|  |
| --- |
| static char c;    static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: read()\n");      buf[0] = c;      return 1;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: write()\n");      c = buf[len – 1];      return len;  } |

“Almost there, but what if the user has provided an invalid buffer, or if the user buffer is swapped out. Wouldn’t this direct access of the user-space buf just crash and oops the kernel?” pounced Pugs.

Shweta, refusing to be intimidated, dived into her collated material and figured out that there are two APIs just to ensure that user-space buffers are safe to access, and then updated them. With the complete understanding of the APIs, she rewrote the above code snippet as follows:

|  |
| --- |
| static char c;    static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: read()\n");      if (copy\_to\_user(buf, &c, 1) != 0)          return -EFAULT;      else          return 1;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      printk(KERN\_INFO "Driver: write()\n");      if (copy\_from\_user(&c, buf + len – 1, 1) != 0)          return -EFAULT;      else          return len;  } |

Then Shweta repeated the usual build-and-test steps as follows:

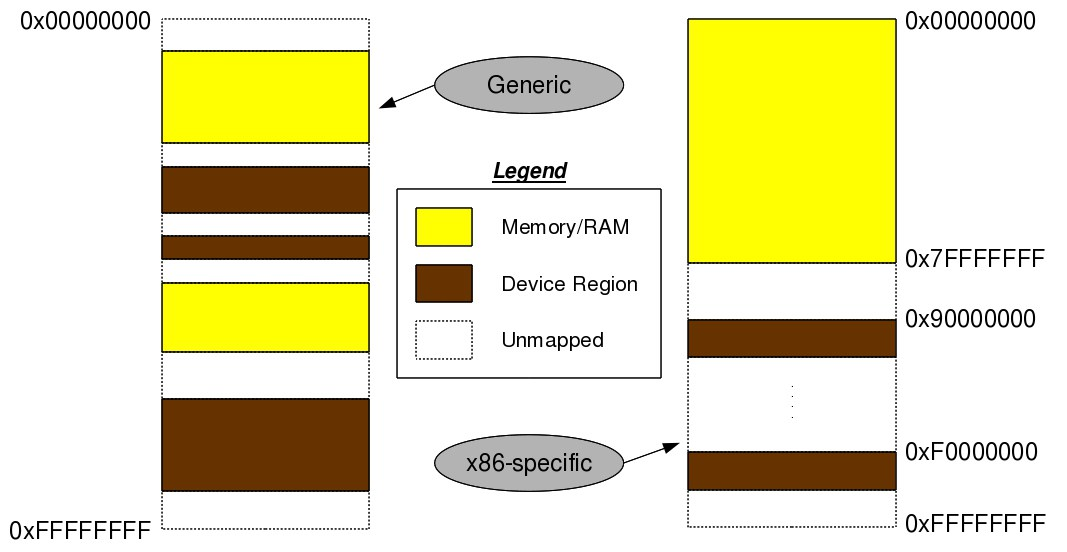
1. Build the modified “null” driver (.ko file) by running make.
2. Load the driver using insmod.
3. Write into /dev/mynull, say, using echo -n "Pugs" > /dev/ mynull
4. Read from /dev/mynull using cat /dev/mynull (stop by using Ctrl+C)
5. Unload the driver using rmmod.

On cat‘ing /dev/mynull, the output was a non-stop infinite sequence of s, as my\_read() gives the last one character forever. So, Pugs intervened and pressed Ctrl+C to stop the infinite read, and tried to explain, “If this is to be changed to ‘the last character only once’, my\_read() needs to return 1 the first time, and zero from the second time onwards. This can be achieved using off (the fourth parameter of my\_read()).”

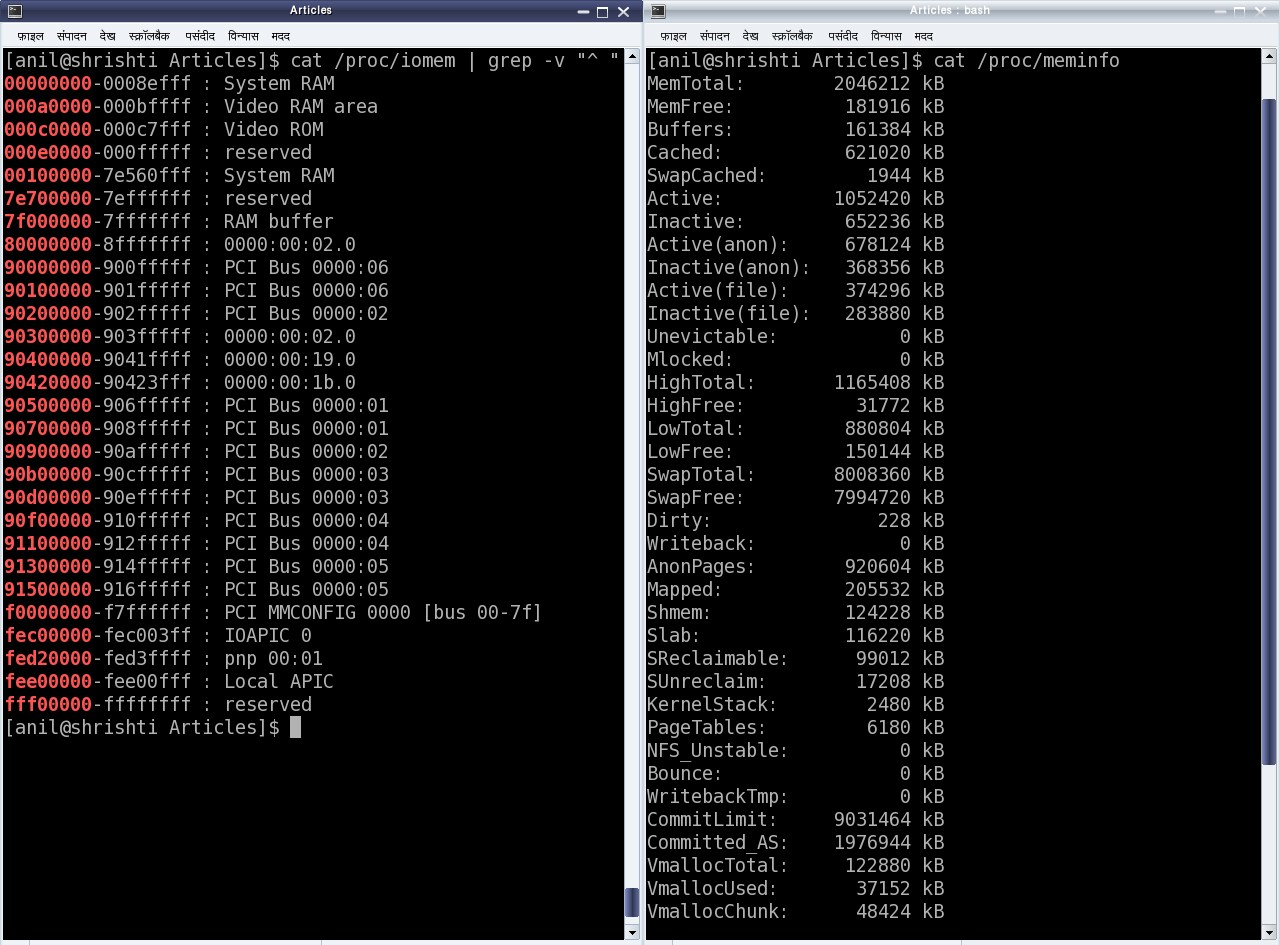
## Part 7:

## Generic hardware interfacing

As every one settled down in the laboratory, lab expert Priti started with an introduction to hardware interfacing in Linux. Skipping the theoretical details, the first interesting slide was about generic architecture-transparent hardware interfacing (see Figure 1).

The basic assumption is that the architecture is 32-bit. For others, the memory map would change accordingly. For a 32-bit address bus, the address/memory map ranges from 0 (0x00000000) to “232 – 1” (0xFFFFFFFF). An architecture-independent layout of this memory map would be like what’s shown in Figure 1 — memory (RAM) and device regions (registers and memories of devices) mapped in an interleaved fashion. These addresses actually are architecture-dependent. For example, in an x86 architecture, the initial 3 GB (0x00000000 to 0xBFFFFFFF) is typically for RAM, and the later 1GB (0xC0000000 to 0xFFFFFFFF) for device maps. However, if the RAM is less, say 2GB, device maps could start from 2GB (0x80000000).

Run cat /proc/iomem to list the memory map on your system. Run cat /proc/meminfo to get the approximate RAM size on your system. Refer to Figure 2 for a snapshot.

Irrespective of the actual values, the addresses referring to RAM are termed as physical addresses, and those referring to device maps as bus addresses, since these devices are always mapped through some architecture-specific bus — for example, the PCI bus in the x86 architecture, the AMBA bus in ARM architectures, the SuperHyway bus in SuperH architectures, etc.

All the architecture-dependent values of these physical and bus addresses are either dynamically configurable, or are to be obtained from the data-sheets (i.e., hardware manuals) of the corresponding architecture processors/controllers. The interesting part is that in Linux, none of these are directly accessible, but are to be mapped to virtual addresses and then accessed through them — thus making the RAM and device accesses generic enough. The corresponding APIs (prototyped in <asm/io.h>) for mapping and unmapping the device bus addresses to virtual addresses are:

|  |
| --- |
| void \*ioremap(unsigned long device\_bus\_address, unsigned long device\_region\_size);  void iounmap(void \*virt\_addr); |

Once mapped to virtual addresses, it depends on the device datasheet as to which set of device registers and/or device memory to read from or write into, by adding their offsets to the virtual address returned by ioremap(). For that, the following are the APIs (also prototyped in <asm/io.h>):

|  |
| --- |
| unsigned int ioread8(void \*virt\_addr);  unsigned int ioread16(void \*virt\_addr);  unsigned int ioread32(void \*virt\_addr);  unsigned int iowrite8(u8 value, void \*virt\_addr);  unsigned int iowrite16(u16 value, void \*virt\_addr);  unsigned int iowrite32(u32 value, void \*virt\_addr); |

## Accessing the video RAM of ‘DOS’ days

After this first set of information, students were directed for the live experiments. The suggested initial experiment was with the video RAM of “DOS” days, to understand the usage of the above APIs.

Shweta got onto the system and went through /proc/iomem (as in Figure 2) and got the video RAM address, ranging from 0x000A0000 to 0x000BFFFF. She added the above APIs, with appropriate parameters, into the constructor and destructor of her existing “null” driver, to convert it into a “vram” driver. Then she added the user access to the video RAM through read and write calls of the “vram” driver; here’s her new file — video\_ram.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136 | #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>  #include <linux/uaccess.h>  #include <asm/io.h>    #define VRAM\_BASE 0x000A0000  #define VRAM\_SIZE 0x00020000    static void \_\_iomem \*vram;  static dev\_t first;  static struct cdev c\_dev;  static struct class \*cl;    static int my\_open(struct inode \*i, struct file \*f)  {      return 0;  }  static int my\_close(struct inode \*i, struct file \*f)  {      return 0;  }  static ssize\_t my\_read(struct file \*f, char \_\_user \*buf, size\_t len, loff\_t \*off)  {      int i;      u8 byte;        if (\*off >= VRAM\_SIZE)      {          return 0;      }      if (\*off + len > VRAM\_SIZE)      {          len = VRAM\_SIZE - \*off;      }      for (i = 0; i < len; i++)      {          byte = ioread8((u8 \*)vram + \*off + i);          if (copy\_to\_user(buf + i, &byte, 1))          {              return -EFAULT;          }      }      \*off += len;        return len;  }  static ssize\_t my\_write(struct file \*f, const char \_\_user \*buf, size\_t len, loff\_t \*off)  {      int i;      u8 byte;        if (\*off >= VRAM\_SIZE)      {          return 0;      }      if (\*off + len > VRAM\_SIZE)      {          len = VRAM\_SIZE - \*off;      }      for (i = 0; i < len; i++)      {          if (copy\_from\_user(&byte, buf + i, 1))          {              return -EFAULT;          }          iowrite8(byte, (u8 \*)vram + \*off + i);      }      \*off += len;        return len;  }    static struct file\_operations vram\_fops =  {      .owner = THIS\_MODULE,      .open = my\_open,      .release = my\_close,      .read = my\_read,      .write = my\_write  };    static int \_\_init vram\_init(void) /\* Constructor \*/  {      if ((vram = ioremap(VRAM\_BASE, VRAM\_SIZE)) == NULL)      {          printk(KERN\_ERR "Mapping video RAM failed\n");          return -1;      }      if (alloc\_chrdev\_region(&first, 0, 1, "vram") < 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, "vram") == NULL)      {          class\_destroy(cl);          unregister\_chrdev\_region(first, 1);          return -1;      }        cdev\_init(&c\_dev, &vram\_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 vram\_exit(void) /\* Destructor \*/  {      cdev\_del(&c\_dev);      device\_destroy(cl, first);      class\_destroy(cl);      unregister\_chrdev\_region(first, 1);      iounmap(vram);  }    module\_init(vram\_init);  module\_exit(vram\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Video RAM Driver"); |

## Summing up

Shweta then repeated the usual steps:

1. Build the “vram” driver (video\_ram.ko file) by running make with a changed Makefile.
2. Load the driver using insmod video\_ram.ko.
3. Write into /dev/vram, say, using echo -n "0123456789" > /dev/vram.
4. Read the /dev/vram contents using od -t x1 -v /dev/vram | less. (The usual cat /dev/vram can also be used, but that would give all the binary content. od -t x1 shows it as hexadecimal. For more details, run man od.)
5. Unload the driver using rmmod video\_ram.

## Part 8:

## x86-specific hardware interfacing

Unlike most other architectures, x86 has an additional hardware accessing mechanism, through direct I/O mapping. It is a direct 16-bit addressing scheme, and doesn’t need mapping to a virtual address for access. These addresses are referred to as port addresses, or ports. Since this is an additional access mechanism, it has an additional set of x86 (assembly/machine code) instructions. And yes, there are the input instructions inb, inw, and inl for reading an 8-bit byte, a 16-bit word, and a 32-bit long word, respectively, from I/O mapped devices, through ports. The corresponding output instructions are outb, outw and outl, respectively. The equivalent C functions/macros (available through the header <asm/io.h>) are as follows:

|  |
| --- |
| u8 inb(unsigned long port);  u16 inw(unsigned long port);  u32 inl(unsigned long port);  void outb(u8 value, unsigned long port);  void outw(u16 value, unsigned long port);  void outl(u32 value, unsigned long port); |

The basic question that may arise relates to which devices are I/O mapped and what the port addresses of these devices are. The answer is pretty simple. As per x86-standard, all these devices and their mappings are predefined. Figure 1 shows a snippet of these mappings through the kernel window /proc/ioports. The listing includes predefined DMA, the timer and RTC, apart from serial, parallel and PCI bus interfaces, to name a few.

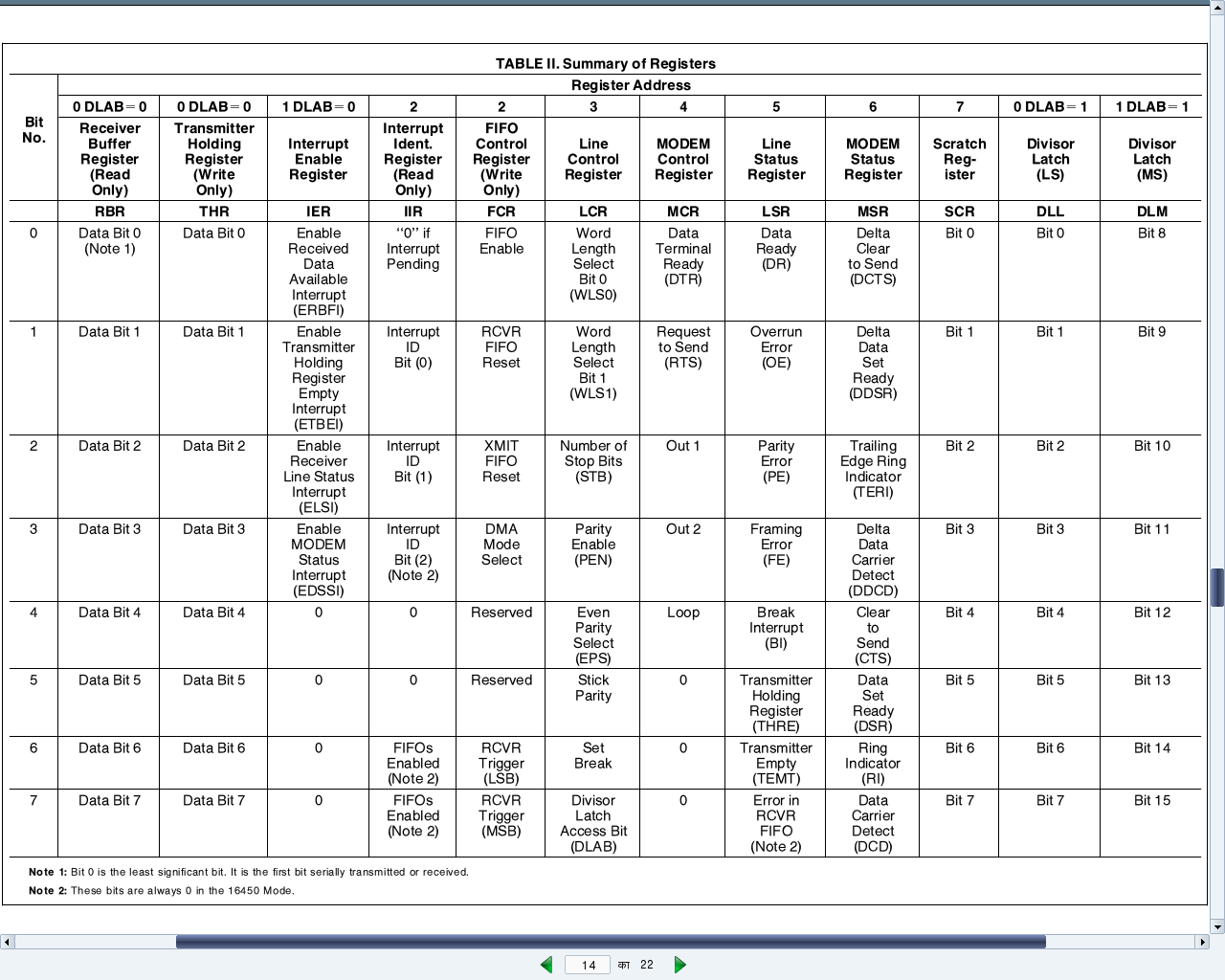
## Simplest: serial port on x86

For example, the first serial port is always I/O mapped from 0x3F8 to 0x3FF. But what does this mapping mean? What do we do with this? How does it help us to use the serial port? That is where a data-sheet of the corresponding device needs to be looked up.

A serial port is controlled by the serial controller device, commonly known as an UART (Universal Asynchronous Receiver/Transmitter) or at times a USART (Universal Synchronous/Asynchronous Receiver/Transmitter). On PCs, the typical UART used is the PC16550D. The [data-sheet for this [PDF]](http://www.national.com/ds/PC/PC16550D.pdf) can be downloaded as part of the [self-extracting package [BIN file]](http://esrijan.com/DDK/LDDK-Package) used for the Linux device driver kit, available at [lddk.esrijan.com](http://esrijan.com/index.php?pagefile=lddk).

Generally speaking, from where, and how, does one get these device data sheets? Typically, an online search with the corresponding device number should yield their data-sheet links. Then, how does one get the device number? Simple… by having a look at the device. If it is inside a desktop, open it up and check it out. Yes, this is the least you may have to do to get going with the hardware, in order to write device drivers. Assuming all this has been done, it is time to peep into the data sheet of the PC16550D UART.

Device driver writers need to understand the details of the registers of the device, as it is these registers that writers need to program, to use the device. Page 14 of the data sheet (also shown in Figure 2) shows the complete table of all the twelve 8-bit registers present in the UART PC16550D.

Each of the eight rows corresponds to the respective bit of the registers. Also, note that the register addresses start from 0 and goes up to 7. The interesting thing about this is that a data sheet always gives the register offsets, which then needs to be added to the base address of the device, to get the actual register addresses. 

Who decides the base address and where is it obtained from? Base addresses are typically board/platform specific, unless they are dynamically configurable like in the case of PCI devices. In this case, i.e., a serial device on x86, it is dictated by the x86 architecture—and that precisely was the starting serial port address mentioned above—0x3F8.

Thus, the eight register offsets, 0 to 7, exactly map to the eight port addresses 0x3F8 to 0x3FF. So, these are the actual addresses to be read or written, for reading or writing the corresponding serial registers, to achieve the desired serial operations, as per the register descriptions.

All the serial register offsets and the register bit masks are defined in the header <linux/serial\_reg.h>. So, rather than hard-coding these values from the data sheet, the corresponding macros could be used instead. All the following code uses these macros, along with the following:

|  |
| --- |
| #define SERIAL\_PORT\_BASE 0x3F8 |

Operating on the device registers  
To summarise the decoding of the PC16550D UART data sheet, here are a few examples of how to do read and write operations of the serial registers and their bits.

Reading and writing the ‘Line Control Register (LCR)’:

|  |
| --- |
| u8 val;    val = inb(SERIAL\_PORT\_BASE + UART\_LCR /\* 3 \*/);  outb(val, SERIAL\_PORT\_BASE + UART\_LCR /\* 3 \*/); |

Setting and clearing the ‘Divisor Latch Access Bit (DLAB)’ in LCR:

|  |
| --- |
| u8 val;    val = inb(SERIAL\_PORT\_BASE + UART\_LCR /\* 3 \*/);    /\* Setting DLAB \*/  val |= UART\_LCR\_DLAB /\* 0x80 \*/;  outb(val, SERIAL\_PORT\_BASE + UART\_LCR /\* 3 \*/);    /\* Clearing DLAB \*/  val &= ~UART\_LCR\_DLAB /\* 0x80 \*/;  outb(val, SERIAL\_PORT\_BASE + UART\_LCR /\* 3 \*/); |

Reading and writing the ‘Divisor Latch’:

|  |
| --- |
| u8 dlab;  u16 val;  dlab = inb(SERIAL\_PORT\_BASE + UART\_LCR);  dlab |= UART\_LCR\_DLAB; // Setting DLAB to access Divisor Latch  outb(dlab, SERIAL\_PORT\_BASE + UART\_LCR);    val = inw(SERIAL\_PORT\_BASE + UART\_DLL /\* 0 \*/);  outw(val, SERIAL\_PORT\_BASE + UART\_DLL /\* 0 \*/); |

## Blinking an LED

To get a real experience of low-level hardware access and Linux device drivers, the best way would be to play with the Linux device driver kit (LDDK) mentioned above. However, just for a feel of low-level hardware access, a blinking light emitting diode (LED) may be tried, as follows:

Connect a light-emitting diode (LED) with a 330 ohm resistor in series across Pin 3 (Tx) and Pin 5 (Gnd) of the DB9 connector of your PC.

Pull up and down the transmit (Tx) line with a 500 ms delay, by loading and unloading the blink\_led driver, using insmod blink\_led.ko and rmmod blink\_led, respectively.

Driver file blink\_led.ko can be created from its source file blink\_led.c by running make with the usual driver Makefile. Given below is the complete blink\_led.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | #include <linux/module.h>  #include <linux/version.h>  #include <linux/types.h>  #include <linux/delay.h>  #include <asm/io.h>    #include <linux/serial\_reg.h>    #define SERIAL\_PORT\_BASE 0x3F8    int \_\_init init\_module()  {      int i;      u8 data;        data = inb(SERIAL\_PORT\_BASE + UART\_LCR);      for (i = 0; i < 5; i++)      {          /\* Pulling the Tx line low \*/          data |= UART\_LCR\_SBC;          outb(data, SERIAL\_PORT\_BASE + UART\_LCR);          msleep(500);          /\* Defaulting the Tx line high \*/          data &= ~UART\_LCR\_SBC;          outb(data, SERIAL\_PORT\_BASE + UART\_LCR);          msleep(500);      }      return 0;  }    void \_\_exit cleanup\_module()  {  }    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("Blinking LED Hack"); |

## Part 9:

## Introducing ioctl()

Input/Output Control (ioctl, in short) is a common operation, or system call, available in most driver categories. It is a one-bill-fits-all kind of system call. If there is no other system call that meets a particular requirement, then ioctl() is the one to use.

Practical examples include volume control for an audio device, display configuration for a video device, reading device registers, and so on — basically, anything to do with device input/output, or device-specific operations, yet versatile enough for any kind of operation (for example, for debugging a driver by querying driver data structures).

The question is: how can all this be achieved by a single function prototype? The trick lies in using its two key parameters: command and argument. The command is a number representing an operation. The argument command is the corresponding parameter for the operation. The ioctl() function implementation does a switch … case over the commmand to implement the corresponding functionality. The following has been its prototype in the Linux kernel for quite some time:

|  |
| --- |
| int ioctl(struct inode \*i, struct file \*f, unsigned int cmd, unsigned long arg); |

However, from kernel 2.6.35, it changed to:

|  |
| --- |
| long ioctl(struct file \*f, unsigned int cmd, unsigned long arg); |

If there is a need for more arguments, all of them are put in a structure, and a pointer to the structure becomes the ‘one’ command argument. Whether integer or pointer, the argument is taken as a long integer in kernel-space, and accordingly type-cast and processed.

ioctl() is typically implemented as part of the corresponding driver, and then an appropriate function pointer is initialised with it, exactly as in other system calls like open(), read(), etc. For example, in character drivers, it is the ioctl or unlocked\_ioctl (since kernel 2.6.35) function pointer field in the struct file\_operations that is to be initialised.

Again, like other system calls, it can be equivalently invoked from user-space using the ioctl() system call, prototyped in <sys/ioctl.h> as:

|  |
| --- |
| int ioctl(int fd, int cmd, ...); |

Here, cmd is the same as what is implemented in the driver’s ioctl(), and the variable argument construct (...) is a hack to be able to pass any type of argument (though only one) to the driver’s ioctl(). Other parameters will be ignored.

Note that both the command and command argument type definitions need to be shared across the driver (in kernel-space) and the application (in user-space). Thus, these definitions are commonly put into header files for each space.

## Querying driver-internal variables

To better understand the boring theory explained above, here’s the code set for the “debugging a driver” example mentioned earlier. This driver has three static global variables: status, dignity, and ego, which need to be queried and possibly operated from an application. The header file query\_ioctl.h defines the corresponding commands and command argument type. A listing follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | #ifndef QUERY\_IOCTL\_H  #define QUERY\_IOCTL\_H  #include <linux/ioctl.h>    typedef struct  {      int status, dignity, ego;  } query\_arg\_t;    #define QUERY\_GET\_VARIABLES \_IOR('q', 1, query\_arg\_t \*)  #define QUERY\_CLR\_VARIABLES \_IO('q', 2)  #define QUERY\_SET\_VARIABLES \_IOW('q', 3, query\_arg\_t \*)    #endif |

Using these, the driver’s ioctl() implementation in query\_ioctl.c would be as follows:

## Part 10:The debugger challenge in kernel-space

As we need some interface to be up to run a debugger to debug anything, a kernel debugger could be visualised in two possible ways:

* Put the debugger into the kernel itself, accessible via the usual console. For example, in the case of kdb, which was not official until kernel 2.6.35, one had to download source code (two sets of patches — one architecture-dependent, one architecture-independent) from [this FTP address](ftp://oss.sgi.com/projects/kdb/download/) and then patch these into the kernel source. However, since kernel 2.6.35, the majority of it is in the officially released kernel source. In either case, kdb support needs to be enabled in kernel source, with the kernel compiled, installed and booted with. The boot screen itself would give the kdb debugging interface.
* Put a minimal debugging server into the kernel; a client would connect to it from a remote host or local user-space over some interface (say serial or Ethernet). This is kgdb, the kernel’s gdb server, to be used with gdb as its client. Since kernel 2.6.26, its serial interface is part of the official kernel release. However, if you’re interested in a network interface, you still need to patch with one of the releases from the [kgdb project page](http://sourceforge.net/projects/kgdb/). In either case, you need to enable kgdb support in the kernel, recompile, install and boot the new kernel.

Please note that in both the above cases, the complete kernel source for the kernel to be debugged is needed, unlike for building modules, where just headers are sufficient. Here is how to play around with kgdb over the serial interface.

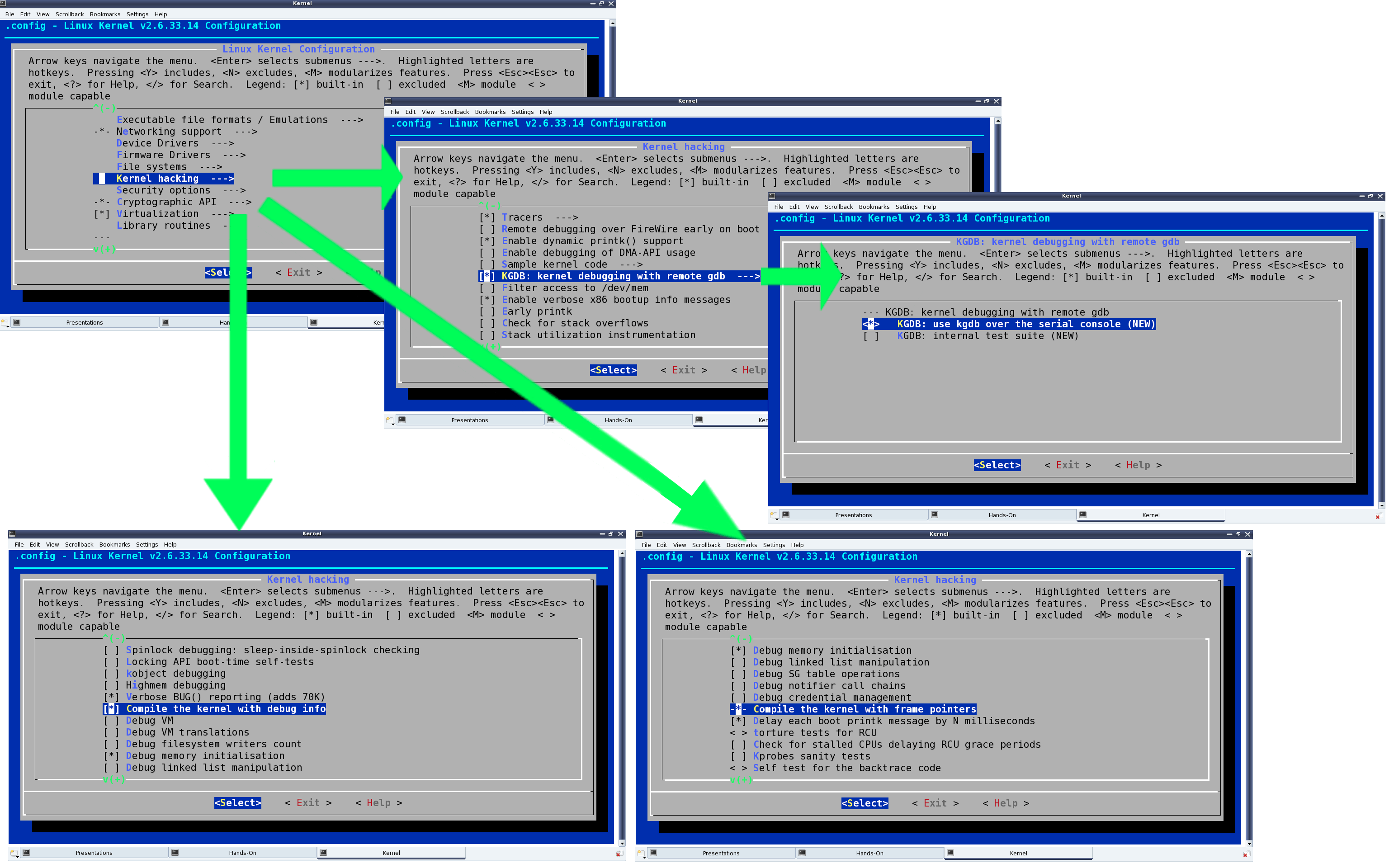
## Setting up the Linux kernel with kgdb

Here are the prerequisites: Either the kernel source package for the running kernel should be installed on your system, or a corresponding kernel source release should have been downloaded from [kernel.org](http://kernel.org/).

First of all, the kernel to be debugged needs to have kgdb enabled and built into it. To achieve that, the kernel source has to be configured with CONFIG\_KGDB=y. Additionally, for kgdb over serial, CONFIG\_KGDB\_SERIAL\_CONSOLE=y needs to be configured. And CONFIG\_DEBUG\_INFO is preferred for symbolic data to be built into the kernel, to make debugging with gdb more meaningful. CONFIG\_FRAME\_POINTER=y enables frame pointers in the kernel, allowing gdb to construct more accurate stack back-traces. All these options are available under “Kernel hacking” in the menu obtained in the kernel source directory (preferably as root, or using sudo), by issuing the following command:

|  |
| --- |
| $ make mrproper      # To clean up properly  $ make oldconfig     # Configure the kernel same as the current running one  $ make menuconfig    # Start the ncurses based menu for further configuration |

![](data:None;base64,)



See the highlighted selections in Figure 1, for how and where these options would be:

* “KGDB: kernel debugging with remote gdb” –> CONFIG\_KGDB
* “KGDB: use kgdb over the serial console” –> CONFIG\_KGDB\_SERIAL\_CONSOLE
* “Compile the kernel with debug info” –> CONFIG\_DEBUG\_INFO
* “Compile the kernel with frame pointers” –> CONFIG\_FRAME\_POINTER

Once configuration is saved, build the kernel (run make), and then a make install to install it, along with adding an entry for the installed kernel in the GRUB configuration file. Depending on the distribution, the GRUB configuration file may be /boot/grub/menu.lst, /etc/grub.cfg, or something similar. Once installed, the kgdb-related kernel boot parameters need to be added to this new entry, as shown in the highlighted text in Figure 2.

kgdboc is for gdb connecting over the console, and the basic format is kgdboc= <serial\_device>, <baud-rate> where:

* <serial\_device> is the serial device file (port) on the system running the kernel to be debugged
* <baud-rate> is the baud rate of this serial port

kgdbwait tells the kernel to delay booting till a gdb client connects to it; this parameter should be given only after kgdboc.

With this, we’re ready to begin. Make a copy of the vmlinux kernel image for use on the gdb client system. Reboot, and at the GRUB menu, choose the new kernel, and then it will wait for gdb to connect over the serial port.

All the above snapshots are with kernel version 2.6.33.14. The same should work for any 2.6.3x release of the kernel source. Also, the snapshots for kgdb are captured over the serial device file /dev/ttyS0, i.e., the first serial port.

## Setting up gdb on another system

Following are the prerequisites:

* Serial ports of the system to be debugged, and the other system to run gdb, should be connected using a null modem (i.e., a cross-over serial) cable.
* The vmlinux kernel image built, with kgdb enabled, needs to be copied from the system to be debugged, into the working directory on the system where gdb is going to be run.

To get gdb to connect to the waiting kernel, launch gdb from the shell and run these commands:

|  |
| --- |
| (gdb) file vmlinux  (gdb) set remote interrupt-sequence Ctrl-C  (gdb) set remotebaud 115200  (gdb) target remote /dev/ttyS0  (gdb) continue |

In the above commands, vmlinux is the kernel image copied from the system to be debugged.

## Debugging using gdb with kgdb

After this, it is all like debugging an application from gdb. One may stop execution using Ctrl+C, add break points using b[reak], stop execution using s[tep] or n[ext] … — the usual gdb way. There are enough GDB tutorials available online, if you need them. In fact, if you are not comfortable with text-based GDB, use any of the standard GUI tools over gdb, like ddd, Eclipse, etc.

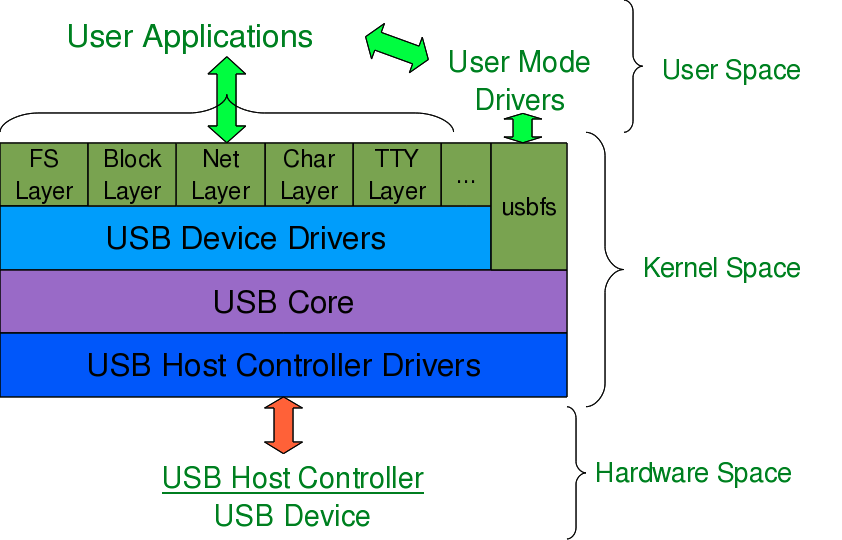
Pugs’ pen drive was the device Shweta was playing with, when both of them sat down to explore the world of USB drivers in Linux. The fastest way to get the hang of it, and Pugs’ usual way, was to pick up a USB device, and write a driver for it, to experiment with. So they chose a pen drive (a.k.a. USB stick) that was at hand — a JetFlash from Transcend, with vendor ID 0x058f and product ID 0x6387.

## Part: 11

## USB device detection in Linux

Whether a driver for a USB device is there or not on a Linux system, a valid USB device will always be detected at the hardware and kernel spaces of a USB-enabled Linux system, since it is designed (and detected) as per the USB protocol specifications. Hardware-space detection is done by the USB host controller — typically a native bus device, like a PCI device on x86 systems. The corresponding host controller driver would pick and translate the low-level physical layer information into higher-level USB protocol-specific information. The USB protocol formatted information about the USB device is then populated into the generic USB core layer (the usbcore driver) in kernel-space, thus enabling the detection of a USB device in kernel-space, even without having its specific driver.

After this, it is up to various drivers, interfaces, and applications (which are dependent on the various Linux distributions), to have the user-space view of the detected devices. Figure 1 shows a top-to-bottom view of the USB subsystem in Linux.

A basic listing of all detected USB devices can be obtained using the lsusb command, as root. Figure 2 shows this, with and without the pen drive plugged in. A -v option to lsusbprovides detailed information.

In many Linux distributions like Mandriva, Fedora,… the usbfs driver is loaded as part of the default configuration. This enables the detected USB device details to be viewed in a more techno-friendly way through the /proc window, using cat /proc/bus/usb/devices. Figure 3 shows a typical snippet of the same, clipped around the pen drive-specific section. The listing basically contains one such section for each valid USB device detected on the system.

## Decoding a USB device section

To further decode these sections, a valid USB device needs to be understood first. All valid USB devices contain one or more configurations. A configuration of a USB device is like a profile, where the default one is the commonly used one. As such, Linux supports only one configuration per device — the default one. For every configuration, the device may have one or more interfaces. An interface corresponds to a function provided by the device.

There would be as many interfaces as the number of functions provided by the device. So, say an MFD (multi-function device) USB printer can do printing, scanning and faxing, then it most likely would have at least three interfaces, one for each of the functions. So, unlike other device drivers, a USB device driver is typically associated/written per interface, rather than the device as a whole — meaning that one USB device may have multiple device drivers, and different device interfaces may have the same driver — though, of course, one interface can have a maximum of one driver only.

It is okay and fairly common to have a single USB device driver for all the interfaces of a USB device. The Driver=... entry in the proc window output (Figure 3) shows the interface to driver mapping — a (none) indicating no associated driver.

For every interface, there would be one or more end-points. An end-point is like a pipe for transferring information either into or from the interface of the device, depending on the functionality. Based on the type of information, the endpoints have four types: Control, Interrupt, Bulk and Isochronous.

As per the USB protocol specification, all valid USB devices have an implicit special control end-point zero, the only bi-directional end-point. Figure 4 shows the complete pictorial representation of a valid USB device, based on the above explanation.

Coming back to the USB device sections (Figure 3), the first letter on each line represents the various parts of the USB device specification just explained. For example, D for device, C for configuration, I for interface, E for endpoint, etc. Details about these and various others are available in the kernel source, in Documentation/usb/proc\_usb\_info.txt.

## The USB pen drive driver registration

“Seems like there are so many things to know about the USB protocol, to be able to write the first USB driver itself — device configuration, interfaces, transfer pipes, their four types, and so many other symbols like T, B, S, … under a USB device specification,” sighed Shweta.

“Yes, but don’t you worry — all of that can be covered in detail later. Let’s do first things first — get the pen drive’s interface associated with our USB device driver (pen\_register.ko),” consoled Pugs.

Like any other Linux device driver, here, too, the constructor and the destructor are required — basically the same driver template that has been used for all the drivers. However, the content would vary, as this is a hardware protocol layer driver, i.e., a horizontal driver, unlike a character driver, which was one of the vertical drivers discussed earlier. The difference would be that instead of registering with and unregistering from VFS, here this would be done with the corresponding protocol layer — the USB core in this case; instead of providing a user-space interface like a device file, it would get connected with the actual device in hardware-space.

The USB core APIs for the same are as follows (prototyped in <linux/usb.h>):

|  |
| --- |
| int usb\_register(struct usb\_driver \*driver);  void usb\_deregister(struct usb\_driver \*); |

As part of the usb\_driver structure, the fields to be provided are the driver’s name, ID table for auto-detecting the particular device, and the two callback functions to be invoked by the USB core during a hot plugging and a hot removal of the device, respectively.

Putting it all together, pen\_register.c would look like what follows:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46 | #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/usb.h>    static int pen\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id)  {      printk(KERN\_INFO "Pen drive (%04X:%04X) plugged\n", id->idVendor, id->idProduct);      return 0;  }    static void pen\_disconnect(struct usb\_interface \*interface)  {      printk(KERN\_INFO "Pen drive removed\n");  }    static struct usb\_device\_id pen\_table[] =  {      { USB\_DEVICE(0x058F, 0x6387) },      {} /\* Terminating entry \*/  };  MODULE\_DEVICE\_TABLE (usb, pen\_table);    static struct usb\_driver pen\_driver =  {      .name = "pen\_driver",      .id\_table = pen\_table,      .probe = pen\_probe,      .disconnect = pen\_disconnect,  };    static int \_\_init pen\_init(void)  {      return usb\_register(&pen\_driver);  }    static void \_\_exit pen\_exit(void)  {      usb\_deregister(&pen\_driver);  }    module\_init(pen\_init);  module\_exit(pen\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("USB Pen Registration Driver"); |

Then, the usual steps for any Linux device driver may be repeated:

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

But surprisingly, the results wouldn’t be as expected. Check dmesg and the proc window to see the various logs and details. This is not because a USB driver is different from a character driver — but there’s a catch. Figure 3 shows that the pen drive has one interface (numbered 0), which is already associated with the usual usb-storage driver.

Now, in order to get our driver associated with that interface, we need to unload the usb-storage driver (i.e., rmmod usb-storage) and replug the pen drive. Once that’s done, the results would be as expected. Figure 5 shows a glimpse of the possible logs and a procwindow snippet. Repeat hot-plugging in and hot-plugging out the pen drive to observe the probe and disconnect calls in action.

## Summing up

“Finally! Something in action!” a relieved Shweta said. “But it seems like there are so many things (like the device ID table, probe, disconnect, etc.), yet to be understood to get a complete USB device driver in place.”

“Yes, you are right. Let’s take them up, one by one, with breaks,” replied Pugs, taking a break himself.

Pugs continued, “Let’s build upon the USB device driver coded in our previous session, using the same  
handy JetFlash pen drive from Transcend, with the vendor ID 0x058f and product ID 0x6387. For that, let’s dig further into the USB protocol, and then convert our learning into code.”

## USB endpoints and their types

Depending on the type and attributes of information to be transferred, a USB device may have one or more endpoints, each belonging to one of the following four categories:

* Control — to transfer control information. Examples include resetting the device, querying information about the device, etc. All USB devices always have the default control endpoint point as zero.
* Interrupt — for small and fast data transfers, typically of up to 8 bytes. Examples include data transfer for serial ports, human interface devices (HIDs) like keyboards, mouse, etc.
* Bulk — for big but comparatively slower data transfers. A typical example is data transfers for mass-storage devices.
* Isochronous — for big data transfers with a bandwidth guarantee, though data integrity may not be guaranteed. Typical practical usage examples include transfers of time-sensitive data like audio, video, etc.

Additionally, all but control endpoints could be “in” or “out”, indicating the direction of data transfer; “in” indicates data flow from the USB device to the host machine, and “out”, the other way.

Technically, an endpoint is identified using an 8-bit number, the most significant bit (MSB) of which indicates the direction — 0 means “out”, and 1 means “in”. Control endpoints are bi-directional, and the MSB is ignored.

Figure 1 shows a typical snippet of USB device specifications for devices connected on a system.

To be specific, the E: lines in the figure show examples of an interrupt endpoint of a UHCI Host Controller, and two bulk endpoints of the pen drive under consideration. Also, the endpoint numbers (in hex) are, respectively, 0x81, 0x01 and 0x82 — the MSB of the first and third being 1, indicating ‘in’ endpoints, represented by (I) in the figure; the second is an (O) or ‘out’ endpoint. MxPS specifies the maximum packet size, i.e., the data size that can be transferred in a single go. Again, as expected, for the interrupt endpoint, it is 2 (<=8), and 64 for the bulk endpoints. Ivl specifies the interval in milliseconds to be given between two consecutive data packet transfers for proper transfer, and is more significant for the interrupt endpoints.

## Decoding a USB device section

As we have just discussed regarding the E: line, it is the right time to decode the relevant fields of others as well. In short, these lines in a USB device section give a complete overview of the device as per the USB specifications, as [discussed in our previous article](http://www.opensourceforu.com/2011/10/usb-drivers-in-linux-1/).

Refer back to Figure 1. The first letter of the first line of every device section is a T, indicating the position of the device in the USB tree, uniquely identified by the triplet <usb bus number, usb tree level, usb port>. D represents the device descriptor, containing at least the device version, device class/category, and the number of configurations available for this device.

There would be as many C lines as the number of configurations, though typically, it is one. C, the configuration descriptor, contains its index, the device attributes in this configuration, the maximum power (actually, current) the device would draw in this configuration, and the number of interfaces under this configuration.

Depending on this, there would be at least that many I lines. There could be more in case of an interface having alternates, i.e., the same interface number but with different properties — a typical scenario for Web-cams.

I represents the interface descriptor with its index, alternate number, the functionality class/category of this interface, the driver associated with this interface, and the number of endpoints under this interface.

The interface class may or may not be the same as that of the device class. And  
depending on the number of endpoints, there would be as many E lines, details of which have already been discussed earlier.

The \* after the C and I represents the currently active configuration and interface, respectively. The P line provides the vendor ID, product ID, and the product revision. S lines are string descriptors showing up some vendor-specific descriptive information about the device.

“Peeping into cat /proc/bus/usb/devices is good in order to figure out whether a device has been detected or not, and possibly to get the first-cut overview of the device. But most probably this information would be required to write the driver for the device as well. So, is there a way to access it using C code?” Shweta asked.

“Yes, definitely; that’s what I am going to tell you about, next. Do you remember that as soon as a USB device is plugged into the system, the USB host controller driver populates its information into the generic USB core layer? To be precise, it puts that into a set of structures embedded into one another, exactly as per the USB specifications,” Pugs replied.

The following are the exact data structures defined in <linux/usb.h>, ordered here in reverse, for flow clarity:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | struct usb\_device  {      …      struct usb\_device\_descriptor descriptor;      struct usb\_host\_config \*config, \*actconfig;      …  };  struct usb\_host\_config  {      struct usb\_config\_descriptor desc;      …      struct usb\_interface \*interface[USB\_MAXINTERFACES];      …  };  struct usb\_interface  {      struct usb\_host\_interface \*altsetting /\* array \*/, \*cur\_altsetting;      …  };  struct usb\_host\_interface  {      struct usb\_interface\_descriptor desc;      struct usb\_host\_endpoint \*endpoint /\* array \*/;      …  };  struct usb\_host\_endpoint  {      struct usb\_endpoint\_descriptor desc;      …  }; |

So, with access to the struct usb\_device handle for a specific device, all the USB-specific information about the device can be decoded, as shown through the /proc window. But how does one get the device handle?

In fact, the device handle is not available directly in a driver; rather, the per-interface handles (pointers to struct usb\_interface) are available, as USB drivers are written for device interfaces rather than the device as a whole.

Recall that the probe and disconnect callbacks, which are invoked by the USB core for every interface of the registered device, have the corresponding interface handle as their first parameter. Refer to the prototypes below:

|  |
| --- |
| int (\*probe)(struct usb\_interface \*interface, const struct usb\_device\_id \*id);  void (\*disconnect)(struct usb\_interface \*interface); |

So, with the interface pointer, all information about the corresponding interface can be accessed — and to get the container device handle, the following macro comes to the rescue:

|  |
| --- |
| struct usb\_device device = interface\_to\_usbdev(interface); |

Adding this new learning into last month’s registration-only driver gets the following code listing (pen\_info.c):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73 | #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/usb.h>    static struct usb\_device \*device;    static int pen\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id)  {      struct usb\_host\_interface \*iface\_desc;      struct usb\_endpoint\_descriptor \*endpoint;      int i;        iface\_desc = interface->cur\_altsetting;      printk(KERN\_INFO "Pen i/f %d now probed: (%04X:%04X)\n",              iface\_desc->desc.bInterfaceNumber, id->idVendor, id->idProduct);      printk(KERN\_INFO "ID->bNumEndpoints: %02X\n",              iface\_desc->desc.bNumEndpoints);      printk(KERN\_INFO "ID->bInterfaceClass: %02X\n",              iface\_desc->desc.bInterfaceClass);        for (i = 0; i < iface\_desc->desc.bNumEndpoints; i++)      {          endpoint = &iface\_desc->endpoint[i].desc;            printk(KERN\_INFO "ED[%d]->bEndpointAddress: 0x%02X\n",                  i, endpoint->bEndpointAddress);          printk(KERN\_INFO "ED[%d]->bmAttributes: 0x%02X\n",                  i, endpoint->bmAttributes);          printk(KERN\_INFO "ED[%d]->wMaxPacketSize: 0x%04X (%d)\n",                  i, endpoint->wMaxPacketSize, endpoint->wMaxPacketSize);      }        device = interface\_to\_usbdev(interface);      return 0;  }    static void pen\_disconnect(struct usb\_interface \*interface)  {      printk(KERN\_INFO "Pen i/f %d now disconnected\n",              interface->cur\_altsetting->desc.bInterfaceNumber);  }    static struct usb\_device\_id pen\_table[] =  {      { USB\_DEVICE(0x058F, 0x6387) },      {} /\* Terminating entry \*/  };  MODULE\_DEVICE\_TABLE (usb, pen\_table);    static struct usb\_driver pen\_driver =  {      .name = "pen\_driver",      .probe = pen\_probe,      .disconnect = pen\_disconnect,      .id\_table = pen\_table,  };    static int \_\_init pen\_init(void)  {      return usb\_register(&pen\_driver);  }    static void \_\_exit pen\_exit(void)  {      usb\_deregister(&pen\_driver);  }    module\_init(pen\_init);  module\_exit(pen\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email@sarika-pugs.com>");  MODULE\_DESCRIPTION("USB Pen Info Driver"); |

Then, the usual steps for any Linux device driver may be repeated, along with the pen drive steps:

* Build the driver (pen\_info.ko file) by running make.
* Load the driver using insmod pen\_info.ko.
* Plug in the pen drive (after making sure that the usb-storage driver is not already loaded).
* Unplug the pen drive.
* Check the output of dmesg for the logs.
* Unload the driver using rmmod pen\_info.

Figure 2 shows a snippet of the above steps on Pugs’ system. Remember to ensure (in the output of cat /proc/bus/usb/devices) that the usual usb-storage driver is not the one associated with the pen drive interface, but rather the pen\_info driver.

## Summing up

Before taking another break, Pugs shared two of the many mechanisms for a driver to specify its device to the USB core, using the struct usb\_device\_id table. The first one is by specifying the <vendor id, product id> pair using the USB\_DEVICE() macro (as done above). The second one is by specifying the device class/category using the USB\_DEVICE\_INFO() macro. In fact, many more macros are available in <linux/usb.h> for various combinations. Moreover, multiple of these macros could be specified in the usb\_device\_id table (terminated by a null entry), for matching with any one of the criteria, enabling to write a single driver for possibly many devices.

“Earlier, you mentioned writing multiple drivers for a single device, as well. Basically, how do we selectively register or not register a particular interface of a USB device?”, queried Shweta. “Sure. That’s next in line of our discussion, along with the ultimate task in any device driver — the data-transfer mechanisms,” replied Pugs.

Now, let’s talk about the ultimate — data transfers to and from a USB device,” continued Pugs.

“But before that, tell me, what is this MODULE\_DEVICE\_TABLE? This has been bothering me since you explained the USB device ID table macros,” asked Shweta, urging Pugs to slow down.

“That’s trivial stuff. It is mainly for the user-space depmod,” he said. ‘Module’ is another term for a driver, which can be dynamically loaded/unloaded. The macro MODULE\_DEVICE\_TABLE generates two variables in a module’s read-only section, which is extracted by depmod and stored in global map files under /lib/modules/<kernel\_version>. Two such files are modules.usbmap and modules.pcimap, for USB and PCI device drivers, respectively. This enables auto-loading of these drivers, as we saw the usb-storage driver getting auto-loaded.

## Part 13:

## USB data transfer

“Time for USB data transfers. Let’s build upon the USB device driver coded in our previous sessions, using the same handy JetFlash pen drive from Transcend, with vendor ID 0x058f and product ID 0x6387,” said Pugs, enthusiastically.

USB, being a hardware protocol, forms the usual horizontal layer in the kernel space. And hence, for it to provide an interface to user-space, it has to connect through one of the vertical layers. As the character (driver) vertical has already been discussed, it is the current preferred choice for the connection with the USB horizontal, in order to understand the complete data transfer flow.

Also, we do not need to get a free unreserved character major number, but can use the character major number 180, reserved for USB-based character device files. Moreover, to achieve this complete character driver logic with the USB horizontal in one go, the following are the APIs declared in <linux/usb.h>:

|  |
| --- |
| int usb\_register\_dev(struct usb\_interface \*intf, struct usb\_class\_driver \*class\_driver);  void usb\_deregister\_dev(struct usb\_interface \*intf, struct usb\_class\_driver \*class\_driver); |

Usually, we would expect these functions to be invoked in the constructor and the destructor of a module, respectively. However, to achieve the hot-plug-n-play behaviour for the (character) device files corresponding to USB devices, these are instead invoked in the probe and disconnect callbacks, respectively.

The first parameter in the above functions is the interface pointer received as the first parameter in both probe and disconnect. The second parameter, struct usb\_class\_driver, needs to be populated with the suggested device file name and the set of device file operations, before invoking usb\_register\_dev. For the actual usage, refer to the functions pen\_probe and pen\_disconnect in the code listing of pen\_driver.c below.

Moreover, as the file operations (write, read, etc.,) are now provided, that is exactly where we need to do the data transfers to and from the USB device. So, pen\_write and pen\_ read below show the possible calls to usb\_bulk\_msg() (prototyped in <linux/usb.h>) to do the transfers over the pen drive’s bulk end-points 0x01 and 0x82, respectively. Refer to the ‘E’ lines of the middle section in Figure 1 for the endpoint number listings of our pen drive.

Refer to the header file <linux/usb.h> under kernel sources, for the complete list of USB core API prototypes for other endpoint-specific data transfer functions like usb\_control\_msg(), usb\_interrupt\_msg(), etc. usb\_rcvbulkpipe(), usb\_sndbulkpipe(), and many such other macros, also defined in <linux/usb.h>, compute the actual endpoint bit-mask to be passed to the various USB core APIs.

Note that a pen drive belongs to a USB mass storage class, which expects a set of SCSI-like commands to be transacted over the bulk endpoints. So, a raw read/write as shown in the code listing below may not really do a data transfer as expected, unless the data is appropriately formatted. But still, this summarises the overall code flow of a USB driver. To get a feel of a real working USB data transfer in a simple and elegant way, one would need some kind of custom USB device, something like the one available [here](http://lddk.eSrijan.com/).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137 | #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/usb.h>    #define MIN(a,b) (((a) <= (b)) ? (a) : (b))  #define BULK\_EP\_OUT 0x01  #define BULK\_EP\_IN 0x82  #define MAX\_PKT\_SIZE 512    static struct usb\_device \*device;  static struct usb\_class\_driver class;  static unsigned char bulk\_buf[MAX\_PKT\_SIZE];    static int pen\_open(struct inode \*i, struct file \*f)  {      return 0;  }  static int pen\_close(struct inode \*i, struct file \*f)  {      return 0;  }  static ssize\_t pen\_read(struct file \*f, char \_\_user \*buf, size\_t cnt, loff\_t \*off)  {      int retval;      int read\_cnt;        /\* Read the data from the bulk endpoint \*/      retval = usb\_bulk\_msg(device, usb\_rcvbulkpipe(device, BULK\_EP\_IN),              bulk\_buf, MAX\_PKT\_SIZE, &read\_cnt, 5000);      if (retval)      {          printk(KERN\_ERR "Bulk message returned %d\n", retval);          return retval;      }      if (copy\_to\_user(buf, bulk\_buf, MIN(cnt, read\_cnt)))      {          return -EFAULT;      }        return MIN(cnt, read\_cnt);  }  static ssize\_t pen\_write(struct file \*f, const char \_\_user \*buf, size\_t cnt, loff\_t \*off)  {      int retval;      int wrote\_cnt = MIN(cnt, MAX\_PKT\_SIZE);        if (copy\_from\_user(bulk\_buf, buf, MIN(cnt, MAX\_PKT\_SIZE)))      {          return -EFAULT;      }        /\* Write the data into the bulk endpoint \*/      retval = usb\_bulk\_msg(device, usb\_sndbulkpipe(device, BULK\_EP\_OUT),              bulk\_buf, MIN(cnt, MAX\_PKT\_SIZE), &wrote\_cnt, 5000);      if (retval)      {          printk(KERN\_ERR "Bulk message returned %d\n", retval);          return retval;      }        return wrote\_cnt;  }    static struct file\_operations fops =  {      .open = pen\_open,      .release = pen\_close,      .read = pen\_read,      .write = pen\_write,  };    static int pen\_probe(struct usb\_interface \*interface, const struct usb\_device\_id \*id)  {      int retval;        device = interface\_to\_usbdev(interface);        class.name = "usb/pen%d";      class.fops = &fops;      if ((retval = usb\_register\_dev(interface, &class)) < 0)      {          /\* Something prevented us from registering this driver \*/          err("Not able to get a minor for this device.");      }      else      {          printk(KERN\_INFO "Minor obtained: %d\n", interface->minor);      }        return retval;  }    static void pen\_disconnect(struct usb\_interface \*interface)  {      usb\_deregister\_dev(interface, &class);  }    /\* Table of devices that work with this driver \*/  static struct usb\_device\_id pen\_table[] =  {      { USB\_DEVICE(0x058F, 0x6387) },      {} /\* Terminating entry \*/  };  MODULE\_DEVICE\_TABLE (usb, pen\_table);    static struct usb\_driver pen\_driver =  {      .name = "pen\_driver",      .probe = pen\_probe,      .disconnect = pen\_disconnect,      .id\_table = pen\_table,  };    static int \_\_init pen\_init(void)  {      int result;        /\* Register this driver with the USB subsystem \*/      if ((result = usb\_register(&pen\_driver)))      {          err("usb\_register failed. Error number %d", result);      }      return result;  }    static void \_\_exit pen\_exit(void)  {      /\* Deregister this driver with the USB subsystem \*/      usb\_deregister(&pen\_driver);  }    module\_init(pen\_init);  module\_exit(pen\_exit);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email\_at\_sarika-pugs\_dot\_com>");  MODULE\_DESCRIPTION("USB Pen Device Driver"); |

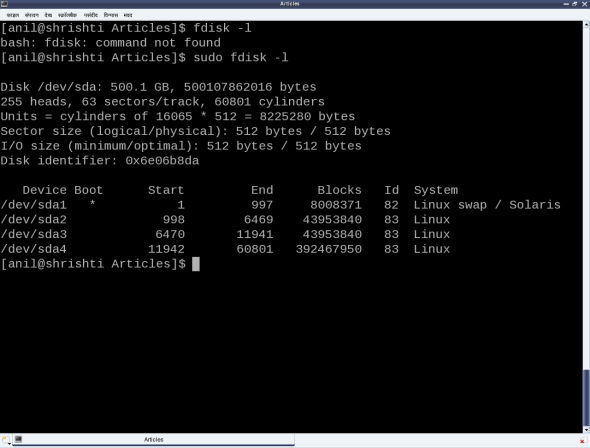
As a reminder, the usual steps for any Linux device driver may be repeated with the above code, along with the following steps for the pen drive:

* Build the driver (pen\_driver.ko) by running make.
* Load the driver using insmod pen\_driver.ko.
* Plug in the pen drive (after making sure that the usb-storage driver is not already loaded).
* Check for the dynamic creation of /dev/pen0 (0 being the minor number obtained — check dmesg logs for the value on your system).
* Possibly try some write/read on /dev/pen0 (you most likely will get a connection timeout and/or broken pipe errors, because of non-conforming SCSI commands).
* Unplug the pen drive and look for /dev/pen0 to be gone.
* Unload the driver using rmmod pen\_driver.

Meanwhile, Pugs hooked up his first-of-its-kind creation — the Linux device driver kit (LDDK) — into his system for a live demonstration of the USB data transfers.

“Aha! Finally a cool complete working USB driver,” quipped Shweta, excited. “Want to have more fun? We could do a block driver over it…,” added Pugs. “Oh! Really?” asked Shweta, with glee. “Yes. But before that, we need to understand the partitioning mechanisms,” commented Pugs.

**Part 14:**

The first line shows the hard disk size in human-friendly format and in bytes. The second line mentions the number of logical heads, logical sectors per track, and the actual number of cylinders on the disk — together known as the geometry of the disk. 

The 255 heads indicate the number of platters or disks, as one read-write head is needed per disk. Let’s number them, say D1, D2,  D255. Now, each disk would have the same number of concentric circular tracks, starting from the outside to the inside. In the above case, there are 60,801 such tracks per disk. Let’s number them, say T1, T2,  T60801. And a particular track number from all the disks forms a cylinder of the same number. For example, tracks T2 from D1, D2,  D255 will together form the cylinder C2. Now, each track has the same number of logical sectors — 63 in our case, say S1, S2,  S63. And each sector is typically 512 bytes. Given this data, one can actually compute the total usable hard disk size, using the following formula:

|  |
| --- |
| Usable hard disk size in bytes = (Number of heads or disks) \* (Number of tracks per disk) \* (Number of sectors per track) \* (Number of bytes per sector, i.e. sector size) |

For the disk under consideration, it would be: 255 \* 60801 \* 63 \* 512 bytes = 500105249280 bytes.

Note that this number may be slightly less than the actual hard disk (500107862016 bytes, in our case). The reason is that the formula doesn’t consider the bytes in the last partial or incomplete cylinder. The primary reason for that is the difference between today’s technology of organising the actual physical disk geometry and the traditional geometry representation using heads, cylinders and sectors.

Note that in the fdisk output, we referred to the heads and sectors per track as logical not physical. One may ask that if today’s disks don’t have such physical geometry concepts, then why still maintain them and represent them in a logical form? The main reason is to be able to continue with the same concepts of partitioning, and be able to maintain the same partition table formats, especially for the most prevalent DOS-type partition tables, which heavily depend on this simplistic geometry. Note the computation of cylinder size (255 heads \* 63 sectors / track \* 512 bytes / sector = 8225280 bytes) in the third line and then the demarcation of partitions in units of complete cylinders.

## DOS-type partition tables

This brings us to the next important topic: understanding DOS-type partition tables. But first, what is a partition, and why should we partition? A hard disk can be divided into one or more logical disks, each of which is called a partition. This is useful for organising different types of data separately, for example, different operating system data, user data, temporary data, etc.

So, partitions are basically logical divisions and need to be maintained by metadata, which is the partition table. A DOS-type partition table contains four partition entries, each a 16-byte entry. Each of these entries can be depicted by the following ‘C’ structure:

|  |
| --- |
| typedef struct  {      unsigned char boot\_type; // 0x00 - Inactive; 0x80 - Active (Bootable)      unsigned char start\_head;      unsigned char start\_sec:6;      unsigned char start\_cyl\_hi:2;      unsigned char start\_cyl;      unsigned char part\_type;      unsigned char end\_head;      unsigned char end\_sec:6;      unsigned char end\_cyl\_hi:2;      unsigned char end\_cyl;      unsigned long abs\_start\_sec;      unsigned long sec\_in\_part;  } PartEntry; |

This partition table, followed by the two-byte signature 0xAA55, resides at the end of the disk’s first sector, commonly known as the Master Boot Record (MBR). Hence, the starting offset of this partition table within the MBR is 512 - (4 \* 16 + 2) = 446. Also, a 4-byte disk signature is placed at offset 440.

The remaining top 440 bytes of the MBR are typically used to place the first piece of boot code, that is loaded by the BIOS to boot the system from the disk. The part\_info.c listing contains these various definitions, along with code for parsing and printing a formatted output of the partition table.

From the partition table entry structure, it could be noted that the start and end cylinder fields are only 10 bits long, thus allowing a maximum of 1023 cylinders only. However, for today’s huge hard disks, this is in no way sufficient. Hence, in overflow cases, the corresponding <head, cylinder, sector> triplet in the partition table entry is set to the maximum value, and the actual value is computed using the last two fields: the absolute start sector number (abs\_start\_sec) and the number of sectors in this partition (sec\_in\_part).

The code for this too is in part\_info.c:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102 | #include <stdio.h>  #include <sys/types.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <unistd.h>    #define SECTOR\_SIZE 512  #define MBR\_SIZE SECTOR\_SIZE  #define MBR\_DISK\_SIGNATURE\_OFFSET 440  #define MBR\_DISK\_SIGNATURE\_SIZE 4  #define PARTITION\_TABLE\_OFFSET 446  #define PARTITION\_ENTRY\_SIZE 16 // sizeof(PartEntry)  #define PARTITION\_TABLE\_SIZE 64 // sizeof(PartTable)  #define MBR\_SIGNATURE\_OFFSET 510  #define MBR\_SIGNATURE\_SIZE 2  #define MBR\_SIGNATURE 0xAA55  #define BR\_SIZE SECTOR\_SIZE  #define BR\_SIGNATURE\_OFFSET 510  #define BR\_SIGNATURE\_SIZE 2  #define BR\_SIGNATURE 0xAA55    typedef struct {      unsigned char boot\_type; // 0x00 - Inactive; 0x80 - Active (Bootable)      unsigned char start\_head;      unsigned char start\_sec:6;      unsigned char start\_cyl\_hi:2;      unsigned char start\_cyl;      unsigned char part\_type;      unsigned char end\_head;      unsigned char end\_sec:6;      unsigned char end\_cyl\_hi:2;      unsigned char end\_cyl;      unsigned long abs\_start\_sec;      unsigned long sec\_in\_part;  } PartEntry;    typedef struct {      unsigned char boot\_code[MBR\_DISK\_SIGNATURE\_OFFSET];      unsigned long disk\_signature;      unsigned short pad;      unsigned char pt[PARTITION\_TABLE\_SIZE];      unsigned short signature;  } MBR;    void print\_computed(unsigned long sector) {      unsigned long heads, cyls, tracks, sectors;        sectors = sector % 63 + 1 /\* As indexed from 1 \*/;      tracks = sector / 63;      cyls = tracks / 255 + 1 /\* As indexed from 1 \*/;      heads = tracks % 255;      printf("(%3d/%5d/%1d)", heads, cyls, sectors);  }    int main(int argc, char \*argv[]) {      char \*dev\_file = "/dev/sda";      int fd, i, rd\_val;      MBR m;      PartEntry \*p = (PartEntry \*)(m.pt);        if (argc == 2) {          dev\_file = argv[1];      }      if ((fd = open(dev\_file, O\_RDONLY)) == -1) {          fprintf(stderr, "Failed opening %s: ", dev\_file);          perror("");          return 1;      }      if ((rd\_val = read(fd, &m, sizeof(m))) != sizeof(m)) {          fprintf(stderr, "Failed reading %s: ", dev\_file);          perror("");          close(fd);          return 2;      }      close(fd);      printf("\nDOS type Partition Table of %s:\n", dev\_file);      printf("  B Start (H/C/S)   End (H/C/S) Type  StartSec    TotSec\n");      for (i = 0; i < 4; i++) {          printf("%d:%d (%3d/%4d/%2d) (%3d/%4d/%2d)  %02X %10d %9d\n",              i + 1, !!(p[i].boot\_type & 0x80),              p[i].start\_head,              1 + ((p[i].start\_cyl\_hi << 8) | p[i].start\_cyl),              p[i].start\_sec,              p[i].end\_head,              1 + ((p[i].end\_cyl\_hi << 8) | p[i].end\_cyl),              p[i].end\_sec,              p[i].part\_type,              p[i].abs\_start\_sec, p[i].sec\_in\_part);      }      printf("\nRe-computed Partition Table of %s:\n", dev\_file);      printf("  B Start (H/C/S)   End (H/C/S) Type  StartSec    TotSec\n");      for (i = 0; i < 4; i++) {          printf("%d:%d ", i + 1, !!(p[i].boot\_type & 0x80));          print\_computed(p[i].abs\_start\_sec);          printf(" ");          print\_computed(p[i].abs\_start\_sec + p[i].sec\_in\_part - 1);          printf(" %02X %10d %9d\n", p[i].part\_type,              p[i].abs\_start\_sec, p[i].sec\_in\_part);      }      printf("\n");      return 0;  } |

As the above is an application, compile it with gcc part\_info.c -o part\_info, and then run ./part\_info /dev/sda to check out your primary partitioning information on /dev/sda. Figure 2 shows the output of ./part\_info on the presenter’s system. Compare it with the fdisk output in Figure 1.

## Partition types and boot records

Now, as this partition table is hard-coded to have four entries, that’s the maximum number of partitions you can have. These are called primary partitions, each having an associated type in the corresponding partition table entry. These types are typically coined by various OS vendors, and hence sort of map to various OSs like DOS, Minix, Linux, Solaris, BSD, FreeBSD, QNX, W95, Novell Netware, etc., to be used for/with the particular OS. However, this is more a formality than a real requirement.

Besides this, one of the four primary partitions can be labelled as something called an extended partition, which has a special significance. As the name suggests, it is used to further extend hard disk division, i.e., to have more partitions. These are called logical partitions and are created within the extended partition. The metadata of these is maintained in a linked-list format, allowing an unlimited number of logical partitions (at least theoretically).

For that, the first sector of the extended partition, commonly called the Boot Record (BR), is used like the MBR to store (the linked-list head of) the partition table for the logical partitions. Subsequent linked-list nodes are stored in the first sector of the subsequent logical partitions, referred to as the Logical Boot Record (LBR). Each linked-list node is a complete 4-entry partition table, though only the first two entries are used — the first for the linked-list data, namely, information about the immediate logical partition, and the second as the linked list’s next pointer, pointing to the list of remaining logical partitions.

To compare and understand the primary partitioning details on your system’s hard disk, follow the steps (as the root user — hence with care) given below:

|  |
| --- |
| ./part\_info /dev/sda ## Displays the partition table on /dev/sda  fdisk -l /dev/sda ## To display and compare the partition table entries with the above |

In case you have multiple hard disks (/dev/sdb, …), hard disk device files with other names (/dev/hda, …), or an extended partition, you may try ./part\_info <device\_file\_name> on them as well. Trying on an extended partition would give you the information about the starting partition table of the logical partitions.

Right now, we have carefully and selectively played (read-only) with the system’s hard disk. Why carefully? Since otherwise, we may render our system non-bootable. But no learning is complete without a total exploration. Hence, in our next session, we will create a dummy disk in RAM and do destructive exploration on it.

## Part 15:

## Disk On RAM source code

Let’s us create a directory called DiskOnRAM which holds the following six files — three ‘C’ source files, two ‘C’ headers, and one Makefile.

### partition.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | #ifndef PARTITION\_H  #define PARTITION\_H    #include <linux/types.h>    extern void copy\_mbr\_n\_br(u8 \*disk);  #endif |

### partition.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176 | #include <linux/string.h>    #include "partition.h"    #define ARRAY\_SIZE(a) (sizeof(a) / sizeof(\*a))    #define SECTOR\_SIZE 512  #define MBR\_SIZE SECTOR\_SIZE  #define MBR\_DISK\_SIGNATURE\_OFFSET 440  #define MBR\_DISK\_SIGNATURE\_SIZE 4  #define PARTITION\_TABLE\_OFFSET 446  #define PARTITION\_ENTRY\_SIZE 16 // sizeof(PartEntry)  #define PARTITION\_TABLE\_SIZE 64 // sizeof(PartTable)  #define MBR\_SIGNATURE\_OFFSET 510  #define MBR\_SIGNATURE\_SIZE 2  #define MBR\_SIGNATURE 0xAA55  #define BR\_SIZE SECTOR\_SIZE  #define BR\_SIGNATURE\_OFFSET 510  #define BR\_SIGNATURE\_SIZE 2  #define BR\_SIGNATURE 0xAA55    typedef struct  {      unsigned char boot\_type; // 0x00 - Inactive; 0x80 - Active (Bootable)      unsigned char start\_head;      unsigned char start\_sec:6;      unsigned char start\_cyl\_hi:2;      unsigned char start\_cyl;      unsigned char part\_type;      unsigned char end\_head;      unsigned char end\_sec:6;      unsigned char end\_cyl\_hi:2;      unsigned char end\_cyl;      unsigned long abs\_start\_sec;      unsigned long sec\_in\_part;  } PartEntry;    typedef PartEntry PartTable[4];    static PartTable def\_part\_table =  {      {          boot\_type: 0x00,          start\_head: 0x00,          start\_sec: 0x2,          start\_cyl: 0x00,          part\_type: 0x83,          end\_head: 0x00,          end\_sec: 0x20,          end\_cyl: 0x09,          abs\_start\_sec: 0x00000001,          sec\_in\_part: 0x0000013F      },      {          boot\_type: 0x00,          start\_head: 0x00,          start\_sec: 0x1,          start\_cyl: 0x0A, // extended partition start cylinder (BR location)          part\_type: 0x05,          end\_head: 0x00,          end\_sec: 0x20,          end\_cyl: 0x13,          abs\_start\_sec: 0x00000140,          sec\_in\_part: 0x00000140      },      {          boot\_type: 0x00,          start\_head: 0x00,          start\_sec: 0x1,          start\_cyl: 0x14,          part\_type: 0x83,          end\_head: 0x00,          end\_sec: 0x20,          end\_cyl: 0x1F,          abs\_start\_sec: 0x00000280,          sec\_in\_part: 0x00000180      },      {      }  };  static unsigned int def\_log\_part\_br\_cyl[] = {0x0A, 0x0E, 0x12};  static const PartTable def\_log\_part\_table[] =  {      {          {              boot\_type: 0x00,              start\_head: 0x00,              start\_sec: 0x2,              start\_cyl: 0x0A,              part\_type: 0x83,              end\_head: 0x00,              end\_sec: 0x20,              end\_cyl: 0x0D,              abs\_start\_sec: 0x00000001,              sec\_in\_part: 0x0000007F          },          {              boot\_type: 0x00,              start\_head: 0x00,              start\_sec: 0x1,              start\_cyl: 0x0E,              part\_type: 0x05,              end\_head: 0x00,              end\_sec: 0x20,              end\_cyl: 0x11,              abs\_start\_sec: 0x00000080,              sec\_in\_part: 0x00000080          },      },      {          {              boot\_type: 0x00,              start\_head: 0x00,              start\_sec: 0x2,              start\_cyl: 0x0E,              part\_type: 0x83,              end\_head: 0x00,              end\_sec: 0x20,              end\_cyl: 0x11,              abs\_start\_sec: 0x00000001,              sec\_in\_part: 0x0000007F          },          {              boot\_type: 0x00,              start\_head: 0x00,              start\_sec: 0x1,              start\_cyl: 0x12,              part\_type: 0x05,              end\_head: 0x00,              end\_sec: 0x20,              end\_cyl: 0x13,              abs\_start\_sec: 0x00000100,              sec\_in\_part: 0x00000040          },      },      {          {              boot\_type: 0x00,              start\_head: 0x00,              start\_sec: 0x2,              start\_cyl: 0x12,              part\_type: 0x83,              end\_head: 0x00,              end\_sec: 0x20,              end\_cyl: 0x13,              abs\_start\_sec: 0x00000001,              sec\_in\_part: 0x0000003F          },      }  };    static void copy\_mbr(u8 \*disk)  {      memset(disk, 0x0, MBR\_SIZE);      \*(unsigned long \*)(disk + MBR\_DISK\_SIGNATURE\_OFFSET) = 0x36E5756D;      memcpy(disk + PARTITION\_TABLE\_OFFSET, &def\_part\_table, PARTITION\_TABLE\_SIZE);      \*(unsigned short \*)(disk + MBR\_SIGNATURE\_OFFSET) = MBR\_SIGNATURE;  }  static void copy\_br(u8 \*disk, int start\_cylinder, const PartTable \*part\_table)  {      disk += (start\_cylinder \* 32 /\* sectors / cyl \*/ \* SECTOR\_SIZE);      memset(disk, 0x0, BR\_SIZE);      memcpy(disk + PARTITION\_TABLE\_OFFSET, part\_table,          PARTITION\_TABLE\_SIZE);      \*(unsigned short \*)(disk + BR\_SIGNATURE\_OFFSET) = BR\_SIGNATURE;  }  void copy\_mbr\_n\_br(u8 \*disk)  {      int i;        copy\_mbr(disk);      for (i = 0; i < ARRAY\_SIZE(def\_log\_part\_table); i++)      {          copy\_br(disk, def\_log\_part\_br\_cyl[i], &def\_log\_part\_table[i]);      }  } |

### ram\_device.h

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | #ifndef RAMDEVICE\_H  #define RAMDEVICE\_H    #define RB\_SECTOR\_SIZE 512    extern int ramdevice\_init(void);  extern void ramdevice\_cleanup(void);  extern void ramdevice\_write(sector\_t sector\_off, u8 \*buffer, unsigned int sectors);  extern void ramdevice\_read(sector\_t sector\_off, u8 \*buffer, unsigned int sectors);  #endif |

### ram\_device.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | #include <linux/types.h>  #include <linux/vmalloc.h>  #include <linux/string.h>    #include "ram\_device.h"  #include "partition.h"    #define RB\_DEVICE\_SIZE 1024 /\* sectors \*/  /\* So, total device size = 1024 \* 512 bytes = 512 KiB \*/    /\* Array where the disk stores its data \*/  static u8 \*dev\_data;    int ramdevice\_init(void)  {      dev\_data = vmalloc(RB\_DEVICE\_SIZE \* RB\_SECTOR\_SIZE);      if (dev\_data == NULL)          return -ENOMEM;      /\* Setup its partition table \*/      copy\_mbr\_n\_br(dev\_data);      return RB\_DEVICE\_SIZE;  }    void ramdevice\_cleanup(void)  {      vfree(dev\_data);  }    void ramdevice\_write(sector\_t sector\_off, u8 \*buffer, unsigned int sectors)  {      memcpy(dev\_data + sector\_off \* RB\_SECTOR\_SIZE, buffer,          sectors \* RB\_SECTOR\_SIZE);  }  void ramdevice\_read(sector\_t sector\_off, u8 \*buffer, unsigned int sectors)  {      memcpy(buffer, dev\_data + sector\_off \* RB\_SECTOR\_SIZE,          sectors \* RB\_SECTOR\_SIZE);  } |

### ram\_block.c

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243 | /\* Disk on RAM Driver \*/  #include <linux/module.h>  #include <linux/kernel.h>  #include <linux/fs.h>  #include <linux/types.h>  #include <linux/genhd.h>  #include <linux/blkdev.h>  #include <linux/errno.h>    #include "ram\_device.h"    #define RB\_FIRST\_MINOR 0  #define RB\_MINOR\_CNT 16    static u\_int rb\_major = 0;    /\*   \* The internal structure representation of our Device   \*/  static struct rb\_device  {      /\* Size is the size of the device (in sectors) \*/      unsigned int size;      /\* For exclusive access to our request queue \*/      spinlock\_t lock;      /\* Our request queue \*/      struct request\_queue \*rb\_queue;      /\* This is kernel's representation of an individual disk device \*/      struct gendisk \*rb\_disk;  } rb\_dev;    static int rb\_open(struct block\_device \*bdev, fmode\_t mode)  {      unsigned unit = iminor(bdev->bd\_inode);        printk(KERN\_INFO "rb: Device is opened\n");      printk(KERN\_INFO "rb: Inode number is %d\n", unit);        if (unit > RB\_MINOR\_CNT)          return -ENODEV;      return 0;  }    static int rb\_close(struct gendisk \*disk, fmode\_t mode)  {      printk(KERN\_INFO "rb: Device is closed\n");      return 0;  }    /\*   \* Actual Data transfer   \*/  static int rb\_transfer(struct request \*req)  {      //struct rb\_device \*dev = (struct rb\_device \*)(req->rq\_disk->private\_data);        int dir = rq\_data\_dir(req);      sector\_t start\_sector = blk\_rq\_pos(req);      unsigned int sector\_cnt = blk\_rq\_sectors(req);        struct bio\_vec \*bv;      struct req\_iterator iter;        sector\_t sector\_offset;      unsigned int sectors;      u8 \*buffer;        int ret = 0;        //printk(KERN\_DEBUG "rb: Dir:%d; Sec:%lld; Cnt:%d\n", dir, start\_sector, sector\_cnt);        sector\_offset = 0;      rq\_for\_each\_segment(bv, req, iter)      {          buffer = page\_address(bv->bv\_page) + bv->bv\_offset;          if (bv->bv\_len % RB\_SECTOR\_SIZE != 0)          {              printk(KERN\_ERR "rb: Should never happen: "                  "bio size (%d) is not a multiple of RB\_SECTOR\_SIZE (%d).\n"                  "This may lead to data truncation.\n",                  bv->bv\_len, RB\_SECTOR\_SIZE);              ret = -EIO;          }          sectors = bv->bv\_len / RB\_SECTOR\_SIZE;          printk(KERN\_DEBUG "rb: Sector Offset: %lld; Buffer: %p; Length: %d sectors\n",              sector\_offset, buffer, sectors);          if (dir == WRITE) /\* Write to the device \*/          {              ramdevice\_write(start\_sector + sector\_offset, buffer, sectors);          }          else /\* Read from the device \*/          {              ramdevice\_read(start\_sector + sector\_offset, buffer, sectors);          }          sector\_offset += sectors;      }      if (sector\_offset != sector\_cnt)      {          printk(KERN\_ERR "rb: bio info doesn't match with the request info");          ret = -EIO;      }        return ret;  }    /\*   \* Represents a block I/O request for us to execute   \*/  static void rb\_request(struct request\_queue \*q)  {      struct request \*req;      int ret;        /\* Gets the current request from the dispatch queue \*/      while ((req = blk\_fetch\_request(q)) != NULL)      {  #if 0          /\*           \* This function tells us whether we are looking at a filesystem request           \* - one that moves block of data           \*/          if (!blk\_fs\_request(req))          {              printk(KERN\_NOTICE "rb: Skip non-fs request\n");              /\* We pass 0 to indicate that we successfully completed the request \*/              \_\_blk\_end\_request\_all(req, 0);              //\_\_blk\_end\_request(req, 0, blk\_rq\_bytes(req));              continue;          }  #endif          ret = rb\_transfer(req);          \_\_blk\_end\_request\_all(req, ret);          //\_\_blk\_end\_request(req, ret, blk\_rq\_bytes(req));      }  }    /\*   \* These are the file operations that performed on the ram block device   \*/  static struct block\_device\_operations rb\_fops =  {      .owner = THIS\_MODULE,      .open = rb\_open,      .release = rb\_close,  };    /\*   \* This is the registration and initialization section of the ram block device   \* driver   \*/  static int \_\_init rb\_init(void)  {      int ret;        /\* Set up our RAM Device \*/      if ((ret = ramdevice\_init()) < 0)      {          return ret;      }      rb\_dev.size = ret;        /\* Get Registered \*/      rb\_major = register\_blkdev(rb\_major, "rb");      if (rb\_major <= 0)      {          printk(KERN\_ERR "rb: Unable to get Major Number\n");          ramdevice\_cleanup();          return -EBUSY;      }        /\* Get a request queue (here queue is created) \*/      spin\_lock\_init(&rb\_dev.lock);      rb\_dev.rb\_queue = blk\_init\_queue(rb\_request, &rb\_dev.lock);      if (rb\_dev.rb\_queue == NULL)      {          printk(KERN\_ERR "rb: blk\_init\_queue failure\n");          unregister\_blkdev(rb\_major, "rb");          ramdevice\_cleanup();          return -ENOMEM;      }        /\*       \* Add the gendisk structure       \* By using this memory allocation is involved,       \* the minor number we need to pass bcz the device       \* will support this much partitions       \*/      rb\_dev.rb\_disk = alloc\_disk(RB\_MINOR\_CNT);      if (!rb\_dev.rb\_disk)      {          printk(KERN\_ERR "rb: alloc\_disk failure\n");          blk\_cleanup\_queue(rb\_dev.rb\_queue);          unregister\_blkdev(rb\_major, "rb");          ramdevice\_cleanup();          return -ENOMEM;      }        /\* Setting the major number \*/      rb\_dev.rb\_disk->major = rb\_major;      /\* Setting the first mior number \*/      rb\_dev.rb\_disk->first\_minor = RB\_FIRST\_MINOR;      /\* Initializing the device operations \*/      rb\_dev.rb\_disk->fops = &rb\_fops;      /\* Driver-specific own internal data \*/      rb\_dev.rb\_disk->private\_data = &rb\_dev;      rb\_dev.rb\_disk->queue = rb\_dev.rb\_queue;      /\*       \* You do not want partition information to show up in       \* cat /proc/partitions set this flags       \*/      //rb\_dev.rb\_disk->flags = GENHD\_FL\_SUPPRESS\_PARTITION\_INFO;      sprintf(rb\_dev.rb\_disk->disk\_name, "rb");      /\* Setting the capacity of the device in its gendisk structure \*/      set\_capacity(rb\_dev.rb\_disk, rb\_dev.size);        /\* Adding the disk to the system \*/      add\_disk(rb\_dev.rb\_disk);      /\* Now the disk is "live" \*/      printk(KERN\_INFO "rb: Ram Block driver initialised (%d sectors; %d bytes)\n",          rb\_dev.size, rb\_dev.size \* RB\_SECTOR\_SIZE);        return 0;  }  /\*   \* This is the unregistration and uninitialization section of the ram block   \* device driver   \*/  static void \_\_exit rb\_cleanup(void)  {      del\_gendisk(rb\_dev.rb\_disk);      put\_disk(rb\_dev.rb\_disk);      blk\_cleanup\_queue(rb\_dev.rb\_queue);      unregister\_blkdev(rb\_major, "rb");      ramdevice\_cleanup();  }    module\_init(rb\_init);  module\_exit(rb\_cleanup);    MODULE\_LICENSE("GPL");  MODULE\_AUTHOR("Anil Kumar Pugalia <email@sarika-pugs.com>");  MODULE\_DESCRIPTION("Ram Block Driver");  MODULE\_ALIAS\_BLOCKDEV\_MAJOR(rb\_major); |

You can also [download the code demonstrated from here](http://www.opensourceforu.com/article_source_code/feb12/device_driver.zip).

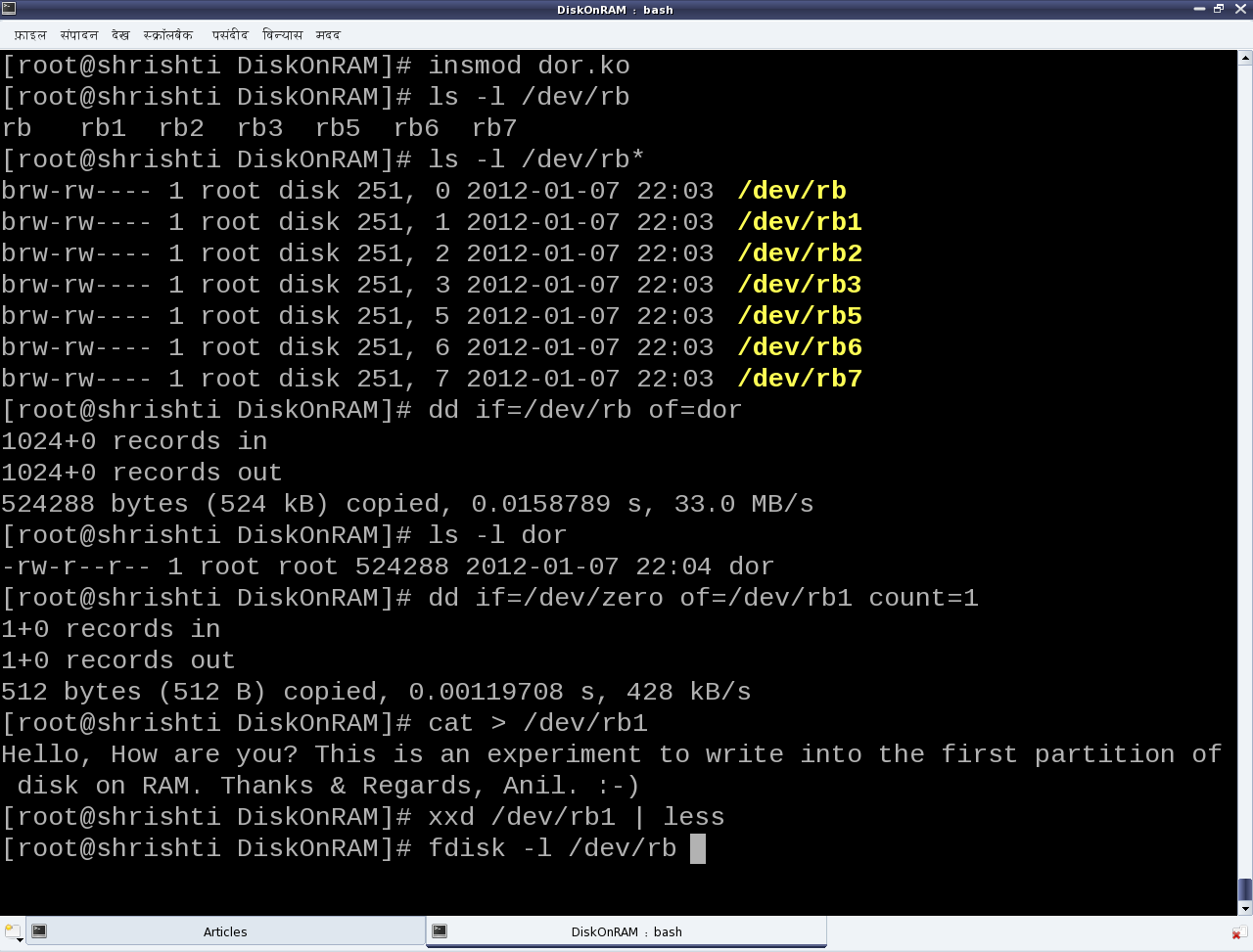
As usual, executing make will build the ‘Disk on RAM’ driver (dor.ko), combining the three ‘C’ files. Check out the Makefile to see how.

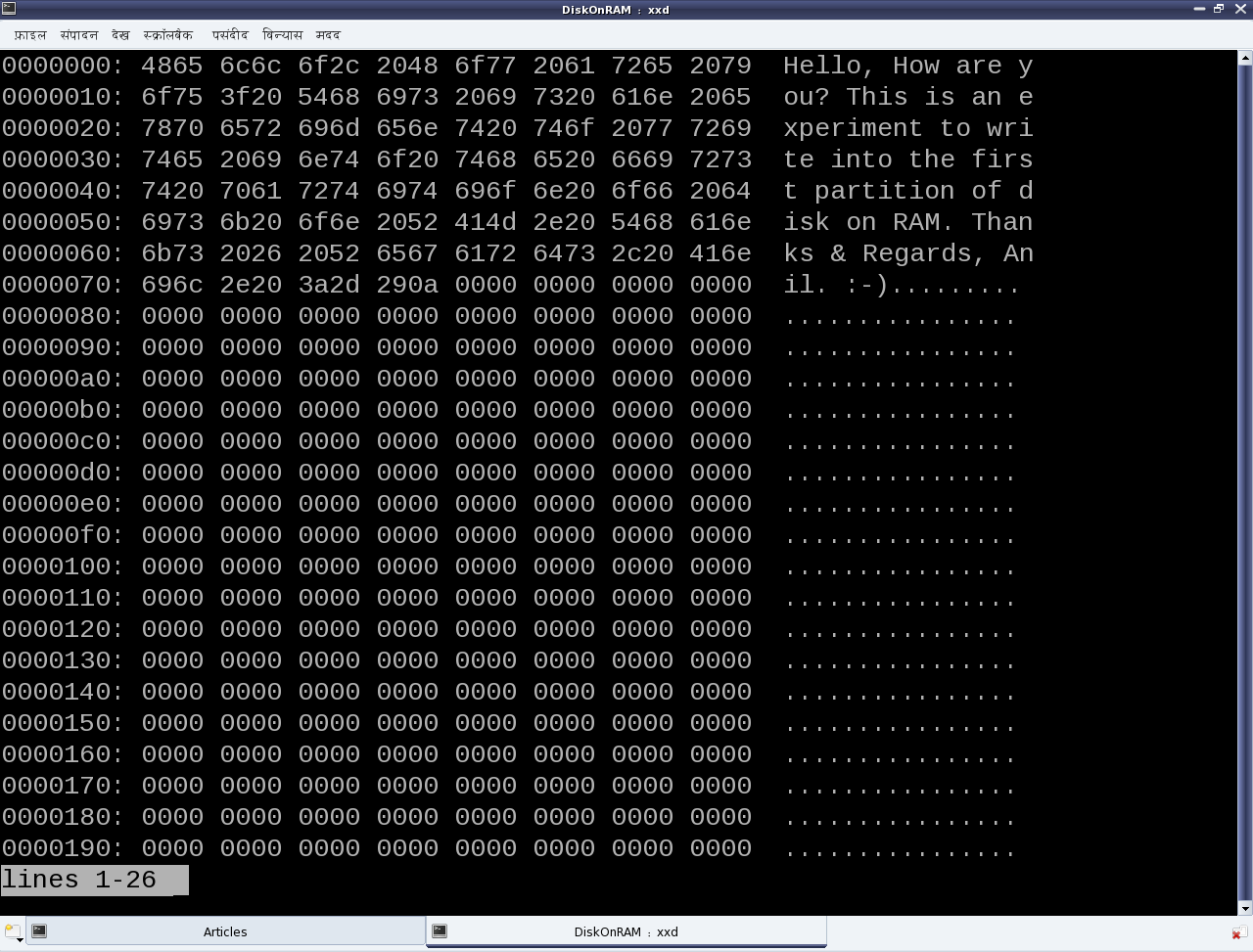
### Makefile

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | # If called directly from the command line, invoke the kernel build system.  ifeq ($(KERNELRELEASE),)        KERNEL\_SOURCE := /usr/src/linux      PWD := $(shell pwd)  default: module    module:      $(MAKE) -C $(KERNEL\_SOURCE) SUBDIRS=$(PWD) modules    clean:      $(MAKE) -C $(KERNEL\_SOURCE) SUBDIRS=$(PWD) clean    # Otherwise KERNELRELEASE is defined; we've been invoked from the  # kernel build system and can use its language.  else        obj-m := dor.o      dor-y := ram\_block.o ram\_device.o partition.o    endif |

To clean the built files, run the usual make clean.

Once built, the following are the experimental steps (refer to Figures 1 to 3).







Please note that all these need to be executed with root privileges:

* Load the driver dor.ko using insmod. This would create the block device files representing the disk on 512 KiB of RAM, with three primary and three logical partitions.
* Check out the automatically created block device files (/dev/rb\*). /dev/rb is the entire disk, which is 512 KiB in size. rb1, rb2 and rb3 are the primary partitions, with rb2 being the extended partition and containing three logical partitions rb5, rb6 and rb7.
* Read the entire disk (/dev/rb) using the disk dump utility dd.
* Zero out the first sector of the disk’s first partition (/dev/rb1), again using dd.
* Write some text into the disk’s first partition (/dev/rb1) using cat.
* Display the initial contents of the first partition (/dev/rb1) using the xxd utility. See Figure 2 for xxd output.
* Display the partition information for the disk using fdisk. See Figure 3 for fdisk output.
* Quick-format the third primary partition (/dev/rb3) as a vfat filesystem (like your pen drive), using mkfs.vfat (Figure 3).
* Mount the newly formatted partition using mount, say at /mnt (Figure 3).
* The disk usage utility df would now show this partition mounted at /mnt (Figure 3). You may go ahead and store files there, but remember that this is a disk on RAM, and so is non-persistent.
* Unload the driver using rmmod dor after unmounting the partition using umount /mnt. All data on the disk will be lost.

## Now let’s learn the rules

We have just played around with the disk on RAM (DOR), but without actually knowing the rules, i.e., the internal details of the game. So, let’s dig into the nitty-gritty to decode the rules. Each of the three .c files represent a specific part of the driver; ram\_device.c and ram\_device.h abstract the underlying RAM operations like vmalloc/vfree, memcpy, etc., providing disk operation APIs like init/cleanup, read/write, etc.

partition.c and partition.h provide the functionality to emulate the various partition tables on the DOR. Recall the [pre-lunch session (i.e., the previous article)](http://www.opensourceforu.com/2012/01/device-drivers-partitions-hard-disk/) to understand the details of partitioning.

The code in this is responsible for the partition information like the number, type, size, etc., that is shown using fdisk. The ram\_block.c file is the core block driver implementation, exposing the DOR as the block device files (/dev/rb\*) to user-space. In other words, four of the five files ram\_device.\* and partition.\* form the horizontal layer of the device driver, and ram\_block.c forms the vertical (block) layer of the device driver. So, let’s understand that in detail.

## The block driver basics

Conceptually, the block drivers are very similar to character drivers, especially with regards to the following:

* Usage of device files
* Major and minor numbers
* Device file operations
* Concept of device registration

So, if you already know character driver implementation, it would be easy to understand block drivers.

However, they are definitely not identical. The key differences are as follows:

* Abstraction for block-oriented versus byte-oriented devices.
* Block drivers are designed to be used by I/O schedulers, for optimal performance. Compare that with character drivers that are to be used by VFS.
* Block drivers are designed to be integrated with the Linux buffer cache mechanism for efficient data access. Character drivers are pass-through drivers, accessing the hardware directly.

And these cause the implementation differences. Let’s analyse the key code snippets from ram\_block.c, starting at the driver’s constructor rb\_init().

The first step is to register for an 8-bit (block) major number (which implicitly means registering for all 256 8-bit minor numbers associated with it). The function for that is as follows:

|  |
| --- |
| int register\_blkdev(unsigned int major, const char \*name); |

Here, major is the major number to be registered, and name is a registration label displayed under the kernel window /proc/devices. Interestingly, register\_blkdev() tries to allocate and register a freely available major number, when 0 is passed for its first parameter major; on success, the allocated major number is returned. The corresponding de-registration function is as follows:

|  |
| --- |
| void unregister\_blkdev(unsigned int major, const char \*name); |

Both these are prototyped in <linux/fs.h>.

The second step is to provide the device file operations, through the struct block\_device\_operations (prototyped in <linux/blkdev.h>) for the registered major number device files.

However, these operations are too few compared to the character device file operations, and mostly insignificant. To elaborate, there are no operations even to read and write, which is surprising. But as we already know that block drivers need to integrate with the I/O schedulers, the read-write implementation is achieved through something called request queues. So, along with providing the device file operations, the following need to be provided:

* The request queue for queuing the read/write requests
* The spin lock associated with the request queue to protect its concurrent access
* The request function to process the requests in the request queue

Also, there is no separate interface for block device file creations, so the following are also provided:

* The device file name prefix, commonly referred to as disk\_name (rb in the dor driver)
* The starting minor number for the device files, commonly referred to as first\_minor.

Finally, two block-device-specific things are also provided, namely:

* The maximum number of partitions supported for this block device, by specifying the total minors.
* The underlying device size in units of 512-byte sectors, for the logical block access abstraction.

All these are registered through the struct gendisk using the following function:

|  |
| --- |
| void add\_disk(struct gendisk \*disk); |

The corresponding delete function is as follows:

|  |
| --- |
| void del\_gendisk(struct gendisk \*disk); |

Prior to add\_disk(), the various fields of struct gendisk need to initialised, either directly or using various macros/functions like set\_capacity(). major, first\_minor, fops, queue, disk\_name are the minimal fields to be initialised directly. And even before the initialisation of these fields, the struct gendisk needs to be allocated, using the function given below:

|  |
| --- |
| struct gendisk \*alloc\_disk(int minors); |

Here, minors is the total number of partitions supported for this disk. And the corresponding inverse function would be:

|  |
| --- |
| void put\_disk(struct gendisk \*disk); |

All these are prototyped in <linux/genhd.h>.

## Request queue and the request function

The request queue also needs to be initialised and set up into the struct gendisk, before add\_disk(). The request queue is initialised by calling:

|  |
| --- |
| struct request\_queue \*blk\_init\_queue(request\_fn\_proc \*, spinlock\_t \*); |

We provide the request-processing function and the initialised concurrency protection spin-lock as parameters. The corresponding queue clean-up function is given below:

|  |
| --- |
| void blk\_cleanup\_queue(struct request\_queue \*); |

The request (processing) function should be defined with the following prototype:

|  |
| --- |
| void request\_fn(struct request\_queue \*q); |

It should be coded to fetch a request from its parameter q, for instance, by using the following:

|  |
| --- |
| struct request \*blk\_fetch\_request(struct request\_queue \*q); |

Then it should either process it, or initiate processing. Whatever it does should be non-blocking, as this request function is called from a non-process context, and also after taking the queue’s spin-lock. Moreover, only functions not releasing or taking the queue’s spin-lock should be used within the request function.

A typical example of request processing, as demonstrated by the function rb\_request() in ram\_block.c is given below:

|  |
| --- |
| while ((req = blk\_fetch\_request(q)) != NULL) /\* Fetching a request \*/  {          /\* Processing the request: the actual data transfer \*/          ret = rb\_transfer(req); /\* Our custom function \*/           /\* Informing that the request has been processed with return of ret \*/          \_\_blk\_end\_request\_all(req, ret);  } |

## Requests and their processing

Our key function is rb\_transfer(), which parses a struct request and accordingly does the actual data transfer. The struct request primarily contains the direction of data transfer, the starting sector for the data transfer, the total number of sectors for the data transfer, and the scatter-gather buffer for the data transfer. The various macros to extract these from the struct request are as follows:

|  |
| --- |
| rq\_data\_dir(req); /\* Operation type: 0 - read from device; otherwise - write to device \*/  blk\_req\_pos(req); /\* Starting sector to process \*/  blk\_req\_sectors(req); /\* Total sectors to process \*/  rq\_for\_each\_segment(bv, req, iter) /\* Iterator to extract individual buffers \*/ |

rq\_for\_each\_segment() is the special one which iterates over the struct request (req) using iter, and extracting the individual buffer information into the struct bio\_vec (bv: basic input/output vector) on each iteration. And then, on each extraction, the appropriate data transfer is done, based on the operation type, invoking one of the following APIs from ram\_device.c:

|  |
| --- |
| void ramdevice\_write(sector\_t sector\_off, u8 \*buffer, unsigned int sectors);  void ramdevice\_read(sector\_t sector\_off, u8 \*buffer, unsigned int sectors); |

Check out the complete code of rb\_transfer() in ram\_block.c.

## Summing up

With that, we have actually learnt the beautiful block drivers by traversing through the design of a hard disk and playing around with partitioning, formatting and various other raw operations on a hard disk. Thanks for patiently listening. Now, the session is open for questions — please feel free to leave your queries as comments.