*International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868*
*Foundation of Computer Science FCS, New York, USA*
*Volume 10 – No.4, January 2016 – www.ijais.org*

# Fundamental Structure of Linux Kernel based Device Driver and Implementation on Linux Host Machine

Nirav Trivedi
M.Tech. Student
Dept. of Electronics and Communication
Charusat University, Changa

Himanshu Patel
Faculty of Electronics and Communication
Charusat University, Changa

Dharmendra Chauhan
Faculty of Electronics and Communication
Charusat University, Changa

## ABSTRACT
This paper discussed about Fundamental structure of device driver based on Linux Kernel. Motive of the paper is to implement simple Linux kernel device driver on Linux host machine. Linux kernel fundamental structure Explained from root level. In Linux operating system how devices talks with kernel through driver, different classification of devices in Linux, key features that Linux offers to us for implementing device driver demonstrated in this paper. Implementation of methods to insert and remove kernel module demonstrated. Motive of paper is to identify the procedure for handling kernel module. Implementation of sample (hello-world) kernel module on Linux Host Machine Demonstrated in this paper.

## Keywords
Device driver, Linux kernel module, embedded Linux, hello world kernel module, Linux kernel structure, classification of Linux kernel module, implementation of device driver.

## 1. INTRODUCTION
In Linux system many concurrent processes attend to different tasks in which each process requires system resources like computing power, memory, and network connectivity. The kernel is executable code that handling all such resources. For better understanding of it, programmer can also split Role of Kernel in following part:

### 1.1 Process management
The kernel can create and destroy any processes. Kernel can also perform Input/output operations on processes. Communication among different processes through signals, pipes, interposes communication primitives are also handled by Kernel. The process management also includes handling process scheduler which controls how processes share the CPU (central processing units).

### 1.2 Memory management
The Linux machine's memory is a major resource, and the policy used to handle it is a critical task for kernel. In Linux each process has its own private address space. This address space can be divided in three logical segments: text, data and stack. Text segments contain machine instructions and it is read only. The data segments contain the uninitialized and initialized data portion and have read/write permissions. The stack segments hold run-time stacks of the application and it also have read/write permissions.

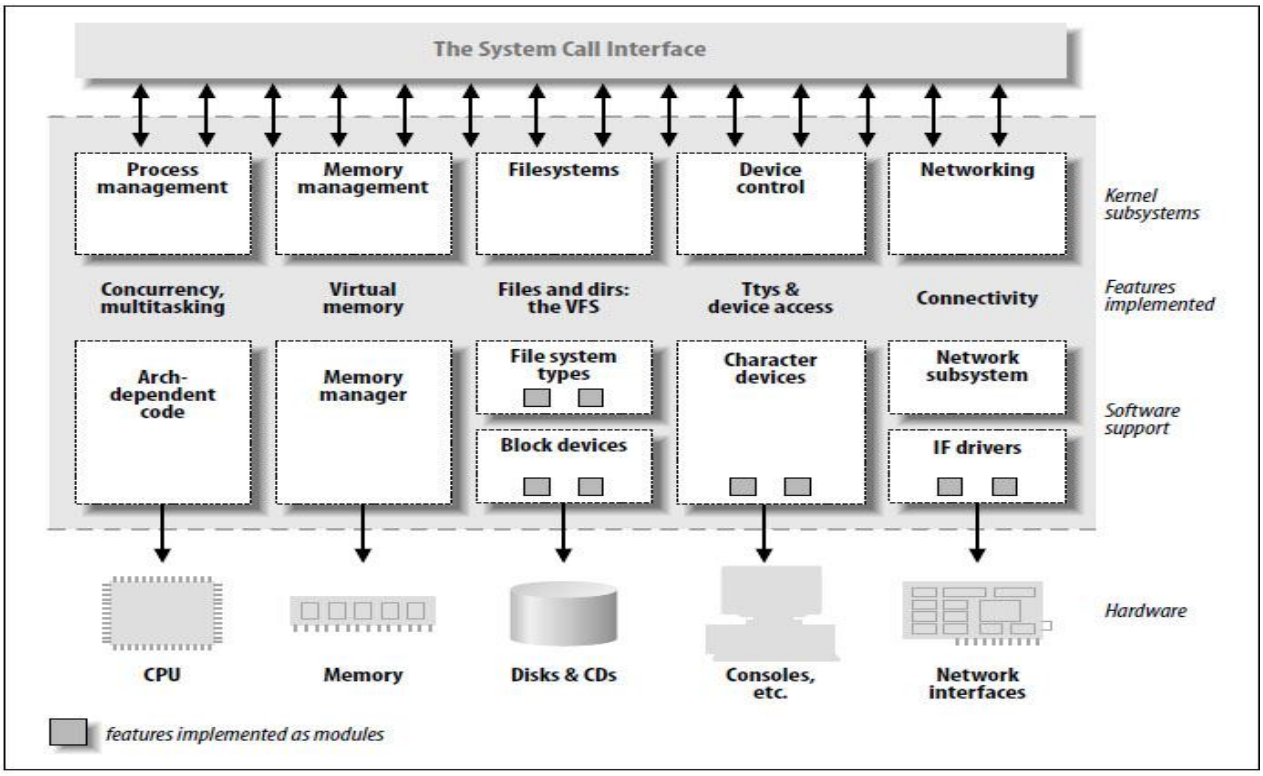
### 1.3 File systems
Everything on Linux can be treated as file. Linux is supporting many file system which makes it flexible and possible to coexist with many other operating system. Linux is supporting many file system like ext, ext2, Xia, minix, umsdos, msdos, vfat, proc, smb, ncp, iso9660, sysv, hpfs, affs, ntfs, ncpfs, ufs, nfs and many other [1]. EXT (extended) file system is first file system designed for linux. The advantage of EXT file system is virtual file system. Virtual file system act as an interface layer, which separates real file system from operating system and system services.

### 1.4 Device control
In Linux many system operations maps to a physical/hardware devices. Programmer has to write code that is specific to the device which is called device driver. Device driver will provide path to interact hardware with kernel and for this programmer has to embed it in kernel.

### 1.5 Networking
Networking must be managed by Linux kernel. Most of networking tasks are not specified to process. For example incoming packets are asynchronous events so the packets must be collected, identified and dispatched before process takes care of them. Eventually Linux kernel handles all the routing and address resolution issues.

## 2. LINUX KERNEL KEY FEATURES
Linux kernel is portable and can run on most architecture. One can see the arch/ directory in the kernel source. For example 32 bit supported architectures are arm, avr32, blackfin, cris, frv, h8300, m32r, m68k,m68knommu, microblaze, mips, mn10300, parisc, s390, sparc, um, xtensa and 64 bit supported architectures are alpha, ia64, sparc64. [1].

Linux kernel is scalable.it can run on super computer as well as small development board like raspberry-pi, beagle bone. (4MB of RAM is enough for Linux kernel)[1].




**(Figure 1 Linux Kernel Architecture [1])**

Linux kernel has Exhaustive networking support.
Linux kernel has modularity as it can include kernel module at run time also.

Linux kernel is open source and easy to program. Many Resources Available For Learning the Kernel Programing.

## 2.1 Loadable modules

Linux has good features of ability to extend the kernel at runtime.it means programmer can add and remove functionality to the kernel runtime. The piece of code that can be add/remove is called modules. The Linux kernel offers deferent type/class of module which is described ahead. Modules can be dynamically linked at runtime by insmod command and unlinked by rmmod command.

# 3. TYPES OF DEVICES AND MODULES

Linux understand different devices as following types:

1. Character devices.
2. Block devices.
3. Network devices.

Thus, each module usually implements one of these types, and thus is classifiable as a Char module, block module, network module.

## 3.1 Character Devices

A character device can be accessed as a stream of bytes. This type of drivers usually implements open, read, write, and close system call. Text console (/dev/console) and serial ports (/dev/ttys0) are examples of this type of devices [2].

## 3.2 Block Devices

Block devices can handle operation on whole block which contains multiple bytes. Block devices and char devices defer by the way data is manage by kernel. Block devices can be handled by file system nodes in the /dev directory. Block devices can host file system. Example of block device is compact disk (C.D.) [1].

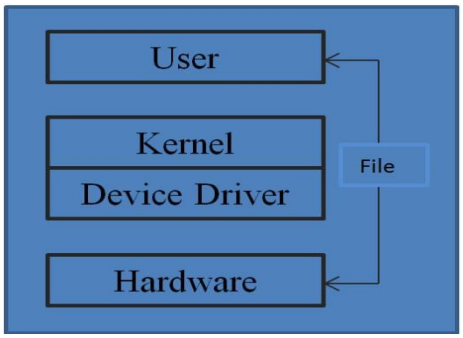
## 3.3 Network Devices

Any network transaction is made through an interface that is able to exchange data between different hosts. This interface can be hardware devices as well as software device. Loopback interface is example of software device. Many network connections protocol are stream oriented (e.g.TCP) but networking devices are generally designed to transmit and receive packages only. Thus network driver handles packets only not the individual connections between hosts [2].

# 4. ANATOMY OF DEVICE DRIVERS

Device driver take on a special role in Linux kernel. They are "black boxes" that make a particular piece of hardware respond to a well-defined internal programing interface. The driver translates between the hardware commands understand by the device and the stylized programing interface used by the kernel. User activities are performed by means of a set of standardized calls that are independent of the specific operations that act on a real hardware is then the role of the device driver [2].

A device driver has three sides: - One talks to the hardware and one talk to the user. While one side talks to the rest of the kernel.




**(Figure 2 Anatomy of Device Drivers)**

# 5. WRITING SAMPLE (HELLO-WORLD) MODULE

Below is demonstration of sample Linux kernel module of displaying hello-world in kernel space. The structure of sample hello.c code is given below:

## 5.1 Linux kernel headers

Upper part of code is Headers specific to the Linux kernel. But it does not have access to usual C library headers.
e.g.

#include<linux/init.h>

#include<linux/module.h>

#inlcude<linux/kernel.h>

## 5.2 Initialization function

This function is called when module is loaded and it is declared by module_init () macro.

e.g.

Static int hello_init (void)

{

Printk (KERN_INFO "hello world sample kernel module");

}

### *5.2.1 Function declaration*

module_init (hello_init);

It returns o for successful operation while negative value indicating error. Here, printk is working same as printf for kernel programing.

## 5.3 Cleanup function

This function is called when module is unloaded and it is declared by module_exit () macro.

e.g.

static void hello_exit(void)

{

Printk(KERN_INFO "good bye unloading module ");

}

### *5.3.1 Function declaration*

module_exit(hello_exit);

## 5.4 Metadata information

### *5.4.1 MODULE_DESCRIPTION*

It is human readable statement of the module's description and module's purpose.

e.g.

MODULE_DESCRIPTION("sample linux kernel module");

### *5.4.2 MODULE_AUTHOR*

It is stating who wrote the module.

e.g.

MODULE_AUTHER("Nirav,Himanshu and Dharmendra");

### *5.4.3 MODULE_LICENCE*

It is stating module's license.

e.g.

MODULE_LICENSE("GPL";)

GPL means General public License. GPL allows everybody to redistribute and sell a product covered by GPL with the recipient has access to the source code and is able to exercise the same rights. The Main goal of GPL is to allow the growth of knowledge by allowing everybody to modify programs.

### *5.4.4 MODULE_VERSION*

It is showing code revision number.

## 5.5 Kernel log

When new module is loaded related information is available in the kernel log. Kernel log messages are available through dmesg command.

## 5.6 Module symbols

When module is loaded any symbol exported by that module becomes part of kernel symbol table. Generally module implements its own functionality without any need to export any symbols. However programmer can export symbols so that other modules may have benefit from using them.

One can export symbol by EXPORT_SYSMBOL(name);

# 6. COMPILING A MODULE

Programmer can compile kernel module in two different ways:

## 6.1 Inside the kernel tree

Kernel module is integrated into the kernel configuration during compilation process. Device driver can be build statistically if needed.

## 6.2 Out of kernel tree

This technique is use when Code is outside of the kernel source tree and located in different directory.

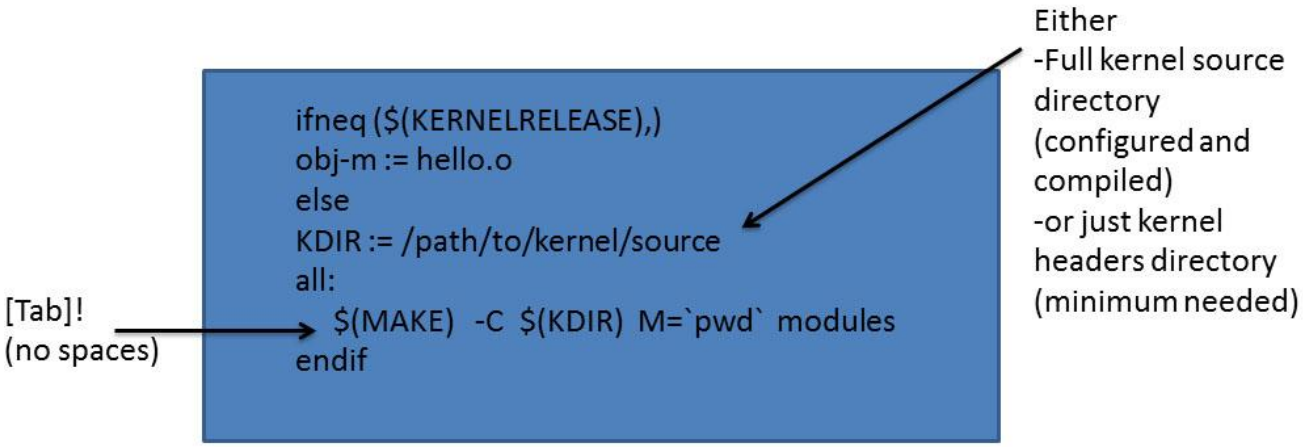
### *6.2.1 Advantage*

It is easier to handle (add/remove) the module than modification to the whole kernel.

### *6.2.2 Drawbacks*

It needs to be build separately as driver cannot be build statically and not integrated to the kernel compilation.

Programmer requires the Makefile for compiling sample driver hello.c. The goal is to generate hello.ko file that can be inserted into kernel. Sample Makefile structure is given below in figure:




**(Figure 3 Structure of Makefile)**

### *6.2.3 Module Utilities*

#### *6.2.3.1 insmod <module_name>.ko*

This will load the given module into kernel.

#### *6.2.3.2 rmmod<module_name>*

This will remove module from kernel.

#### *6.2.3.3 lsmod*

This will displays the list of modules loaded into kernel. It can be also check from /proc/modules directory.

#### *6.2.3.4 modinfo<module_name>*

This will gets information about the module parameters, module license, module descriptions and dependencies.

### *6.2.4 Steps to compile and insert/remove module*

$sudo make

(This will compile Makefile and generates hello.ko file.)

$sudo insmod hello.ko

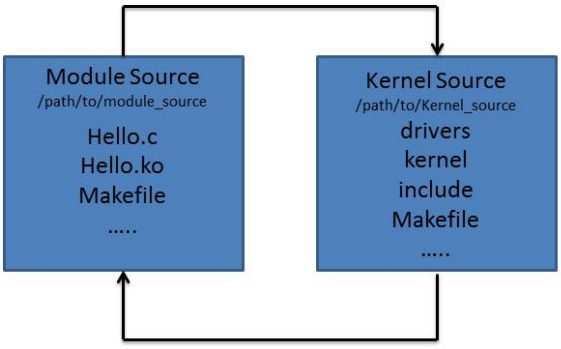
$sudo rmmod hello.ko

## 7. KERNEL SPACE VS USER SPACE

Kernel Modules runs in kernel space whereas application run in user space, this concept is at the base of operating systems theory. Every operating system has different operating levels. These levels have different roles and some operations are prohibited at lower levels. Kernel executes in higher level called kernel space (supervisor mode) where everything is allowed. While applications executes in lower level called user space (user mode).kernel module runs in kernel space.

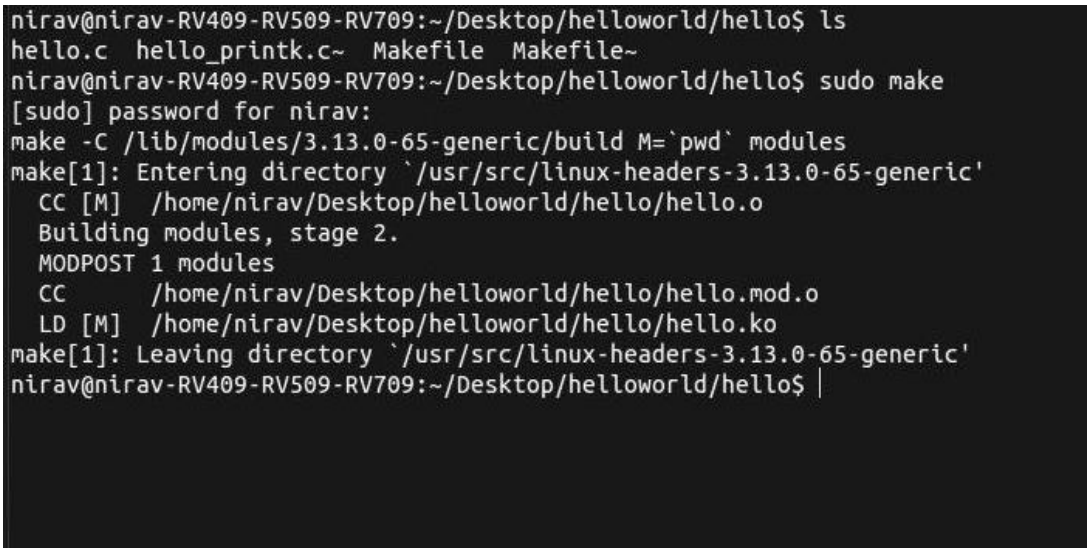
## 8. SIMULATION AND RESULT

The module's Makefile is interpreted with undefined KERNELRELESE so it calls the kernel's Makefile and passing the module directory with M variable. Kernel's Makefile compile module and due to M variable the module's Makefile is interpreted with defined KERNELRELESE.



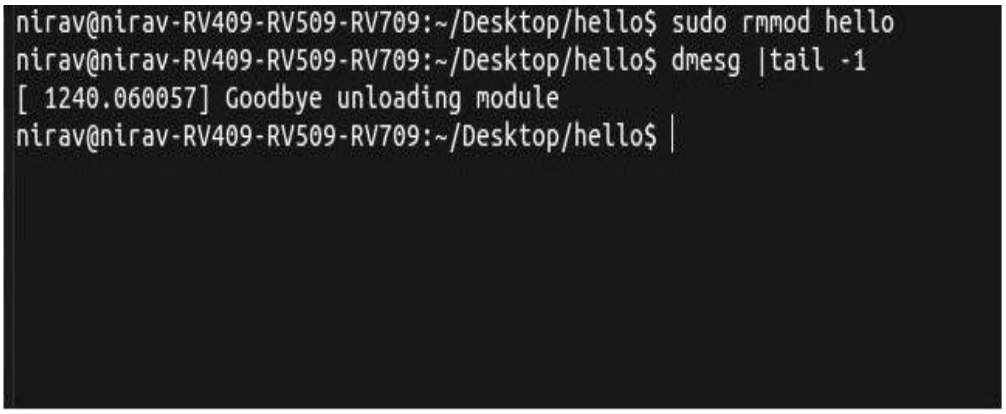

**(Figure 4 Module source vs Kernel source)**

Successful implementation of sample (hello world) device driver on our Linux host machine shown below.(Ubuntu 3.13.0.65-generic kernel). Below figures shows how to compile hello.c file using Makefile and make command. After successful compilation hello.ko generated, this can be inserted/removed into/from kernel as module. Kernel console display shows the messages of insert module/cleanup module using dmesg command.




**(Figure 5 Compiling Hello.c file using Makefile and make command in Linux Host Machine )**


**(Figure 6 insert module hello.ko (using insmod command) and displaying Kernel console (using dmesg) in Host Machine )**




**(Figure 7 remove module hello.ko (using rmmod command) and displaying Kernel console (using dmesg) in Host Machine )**

## 9. CONCLUSION AND FUTURE WORK

Basic Structure of Linux kernel loadable module discussed and evaluated. Fundamental of linux kernel architecture discussed in brief. linux kernel has advantage of loading kernel module during run time compare to other operating system which is one of the reasons why programmer use linux compare to other operating system. Methods for implementing kernel module in linux at run time demonstrated. Also Advantages and Drawbacks of Out of kernel tree module compilation explained in brief. Simple Hello-World device driver simulated on linux host machine.

In future programmer can develop and implement Character Device Driver and miscellaneous device driver on Linux host machine. Also future aim can be to develop driver on embedded development board like raspberry pi, beagle bone black,odorid-XU4 with embedded linux ported on it.

## 10. REFERENCES

[1] Jonathan corbet,Alessandro,Rubini,and Greg Kroah-Hartman. Linux Device Drivers 3e O'REILLY.

[2] XuZhe, LiuZhuo,ZhangHua,HuangWenjiang.Development of Linux Baed USB Device Driver for Portable Spectrometer,2009 IEEE

[3] Shaojie Wang, Sharad Malik. Synthesizing Operating System Based Device Driver in Embedded System,2003 IEEE.

[4] T.K.Damodharan,V.Rhymend Uthariaraj.USB Printer Driver Development for Handheld Devcies,26th Int.Conf. Information Technology Interfaces ITI, JUNE 2004 IEEE.

[5] Juan Zhu,Shuai Wang,Shuyan Zhang,Jinli Wang,Zhaoxi Li. Embedded Driver System for USB Mouse,2011 IEEE.

[6] Gong Yum,Sun Li-hua.Analysis and Implementation of USB Driver Based on VxWorks, 2010 IEEE.

[7] Moritz Jodeit,Martin Johns. USB Device Drivers : A Stepping Stone into your Kernel, 2010 IEEE.