**Day – 1**

gcc -E hello.c -o hello.i

gcc -S hello.c -o hello.s

gcc -C hello.s -o hello.o

**RBL- ROM :**  First Peice of code to run on the SOC when poweron. Its written by venodr. Eg:TI

you dont get source of this . very thiny .Limited functionality.

Its role is load primary boot loader .Eg:SPL/MLO(Secondary program loader/Memory loader)

* SPL/MLO -SRAM : Load and execute third stage bootloader eg:u-boot
* U-boot - DDR: load and execcute linux kernel from the DDR memory
* kernel - DDR
* rootfs

**General Embedded System Arch**

---------------

Application

----------------

Stack

----------------

OS & Drivers

----------------

Hardware

----------------

**Embedded System Components**

- Hardware

- Toolchain

- Bootloader

- Kernel

- Root File System

- Embedded Application Development

Hardware:

Product:

Board: Beagle Bone

SOC: TI-AM335x

ARM Core: ARM Cortex-A8

ARM Arch: armV7a

ARM Core

- Application Core ( ARM9/11, Cortex-A8/A9/A5/A7/A15/A54 )

- Embedded Core ( ARM7, Cortex-M0/M1/M3/M4/M7 )

- Realtime Core (Cortex-R4/R5/R7 )

- Security Core

SOC: Few Vendors

TI, Freescale,NXP, ATmel, QualCom, ST, Samsung

Alwiner, Rockchip, MediaTek

**Build System**

Software components Toolchain, Bootloader, Kernel, Root File System, Embedded Application can be build manually or using build systems.

**Build System**

* Set of scripts used to automate the compilation process

Now the features offered by build systems are much more than only compilation.

* Mannaged build process ( download src, download patch, extract, patch, configure, compile, install, package)
* Version Control ( git )
* Packaging for distribution
* Image creatation
* Testing output binaries and system images
* Application development Tools

**Build Systems List:**

* Buildroot
* LTIB
* OpenEmbedded
  + Poky ( Yocto )

-------------------------------------------------------

beaglebone, raspberry-pi, Wand board ( Community Learning boards)

WEGA Board, Mira board ( Industrial use & Product development )

**Yocto Introduction:**

- Extended OpenEmbedded features with reference project Poky

---------------------------------------------------------------------------

sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib build-essential chrpath socat libsdl1.2-dev xterm

git clone git://git.yoctoproject.org/poky

git checkout dunfell

source oe-init-build-env

conf/local.conf

BB\_NUMBER\_THREADS = "20"

PARALLEL\_MAKE = "-j 4"

bitbake core-image-sato

runqemu qemux86\_64

"poweroff"+"Enter".

bitbake -c cleansstate <recipe-name>

**Adding New reciepe**

bitbake-layers create-layer meta-sample

bitbake-layers show-layers

bitbake-layers add-layer meta-sample

bitbake sample

DESCRIPTION = "A friendly program that prints Hello World!"

PRIORITY = "optional"

SECTION = "examples"

LICENSE = "MIT"

LIC\_FILES\_CHKSUM = "file://${COMMON\_LICENSE\_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC\_URI = "file://helloworld.c"

S = "${WORKDIR}"

do\_compile() {

${CC} ${CFLAGS} ${LDFLAGS} helloworld.c -o helloworld

}

do\_install() {

install -d ${D}${bindir}

install -m 0755 helloworld ${D}${bindir}

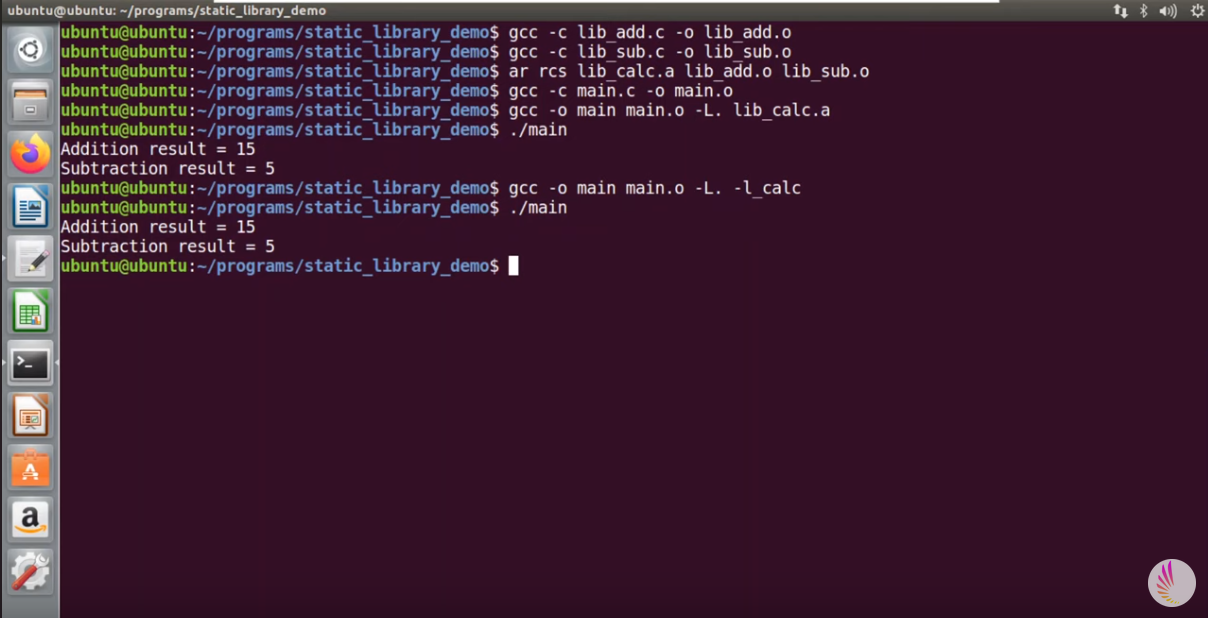
}

/home/luser/Training/Day1/poky/build/tmp/deploy/images/qemux86/test/core-image-minimal-qemux86-20230209083004.rootfs/usr/bin

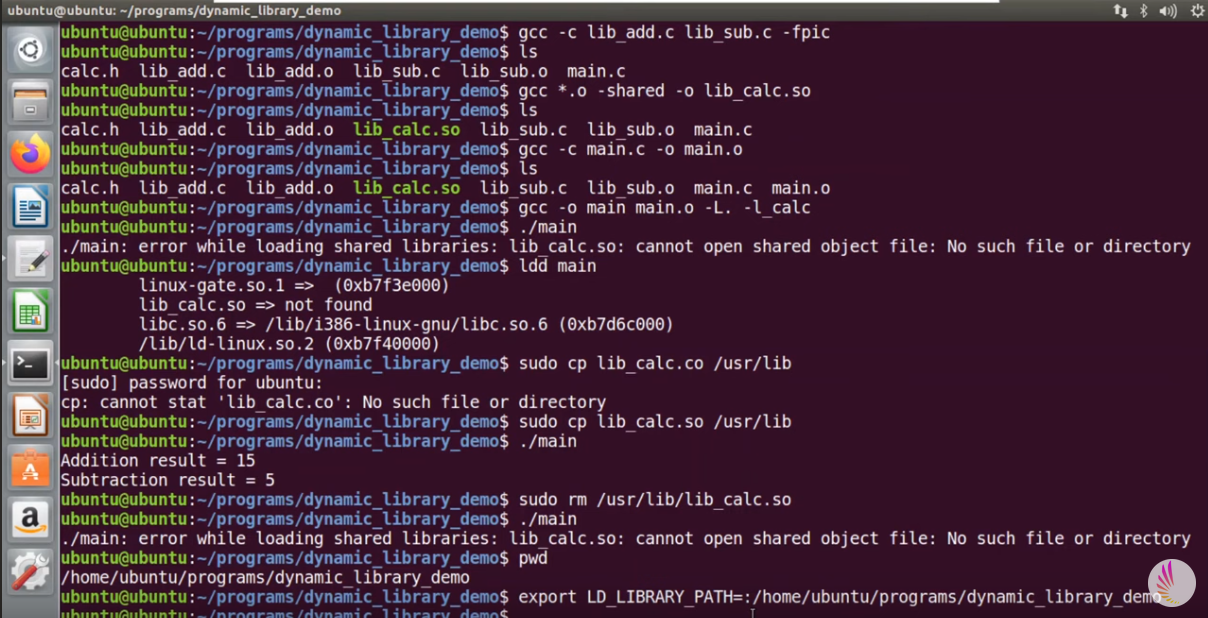
**Day – 2**

Static / Dynamic libraries.

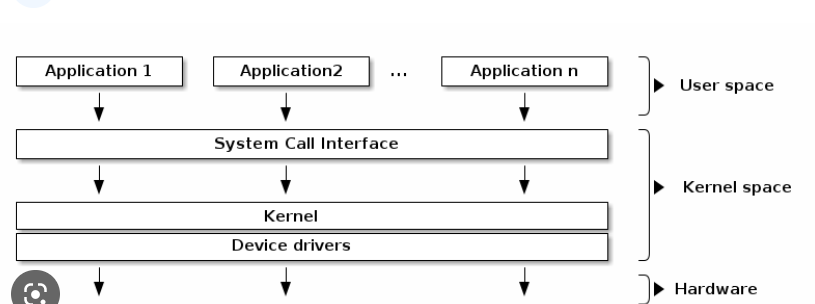
Static: <https://www.youtube.com/watch?v=pkMg_df8gHs>

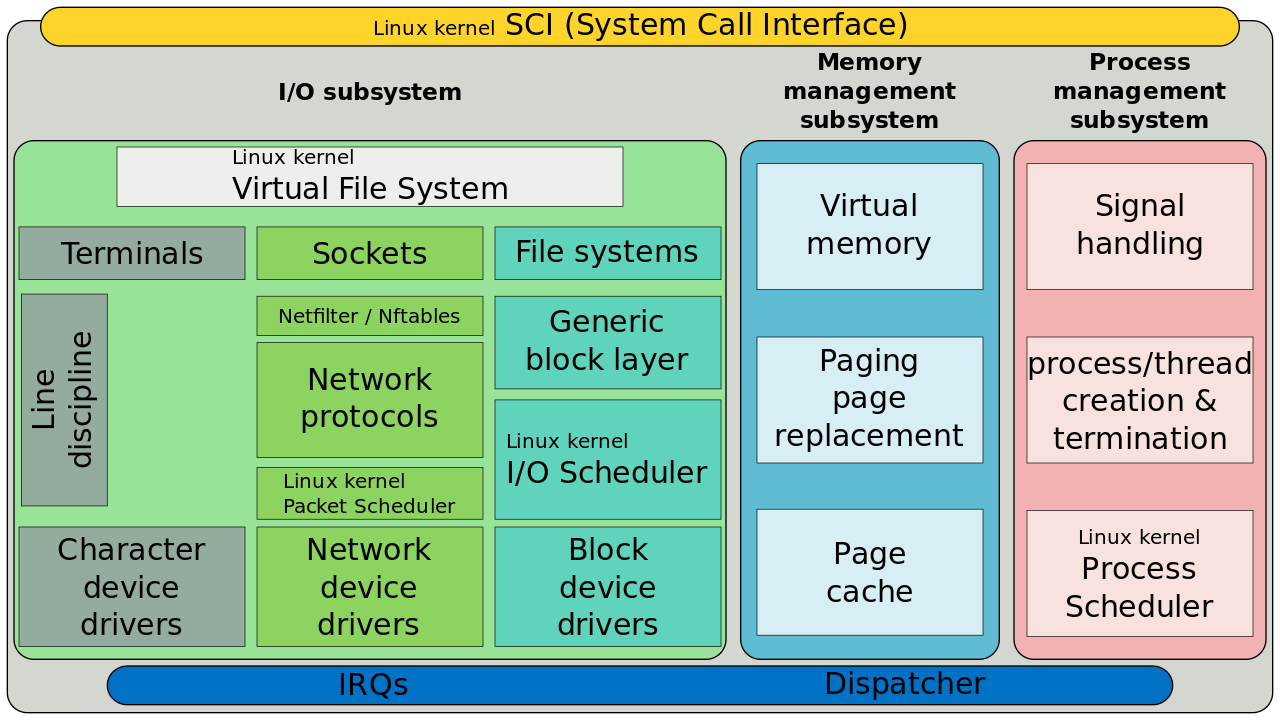


Dynamic: <https://www.youtube.com/watch?v=3RmIVDgPmGk>



**Day – 3**





**Modules** - > Dynamic

Modules are preffered way for development

we are using same gcc and other tool chain for module programming

Modules are nothing but adding new service to existing code

**Points to considered**

1. It should not use user space libs eg: stdlib

2.Include files must be from kernel headers

eg: user space : /usr/include or /usr/include/linux

kernel space : /lib/modules/kernel version/buid/include

uname –r

#include<linux/module.h>

#include<linux/kernel.h>

#include<linux/init.h>

Kmod subsystem is responsible for intialise and maintain module

It provides set of constants,datatypes,functions needed for initialization

kernel.h provides access to sybol tables,global functions

init.h provides functions for register and unregister module

Every module should have init and cleanup routine

-> init\_module called when it is loaded

-> cleanup called when it is removed

insmod hello.ko

rmmod hello

We need to write make files to use kernel build scripts located at

/usr/src/version/build

obj-y -> static

obj-m -> dynamic

lsmod /More

cat /proc/modules | more

**HelloWorld.c**

#include <linux/init.h>

#include <linux/module.h>

#include <linux/kernel.h>

void func(void);

int val=100;

void func(void)

{

printk("Function invoked \n");

}

static int \_\_init hellowolrd\_init(void) {

printk("Hello world loaded!\n");

return 0;

}

static void \_\_exit hellowolrd\_exit(void) {

printk("End of the world\n");

}

module\_init(hellowolrd\_init);

module\_exit(hellowolrd\_exit);

MODULE\_AUTHOR("Sathish <csathish.micro@gmail.com>");

MODULE\_LICENSE("GPL");

**Makefile:**

obj-m := helloworld.o

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

all default: modules

install: modules\_install

modules modules\_install help clean:

$(MAKE) -C $(KERNELDIR) M=$(shell pwd) $@

**Day – 4**

git clone <https://github.com/torvalds/linux>

git checkout v4.1

ls

**arch/:** The Linux kernel is a fast-growing project that supports more and more architectures. That being said, the kernel wants to be as generic as possible. Architecture-specific code is separated from the rest, and falls into this directory. This directory contains processor-specific subdirectories such as alpha/, arm/, mips/, blackfin/, and so on.

**block/:** This directory contains code for block storage devices, actually the scheduling algorithm. **crypto/:** This directory contains the cryptographic API and the encryption algorithms code. **Documentation/:** This should be your favorite directory. It contains the descriptions of APIs used for different kernel frameworks and subsystems. You should look here prior to asking any questions on forums.

**drivers/:** This is the heaviest directory, continuously growing as device drivers get merged. It contains every device driver organized in various subdirectories.

**fs/:** This directory contains the implementation of different filesystems that the kernel actually supports, such as NTFS, FAT, ETX{2,3,4}, sysfs, procfs, NFS, and so on.

**include/:** This contains kernel header files.

**init/:** This directory contains the initialization and start up code.

**ipc/:** This contains implementations of the inter-process communication (IPC) mechanisms, such as message queues, semaphores, and shared memory.

**kernel/:** This directory contains architecture-independent portions of the base kernel.

**lib/:** Library routines and some helper functions live here. They are generic kernel object (kobject) handlers, cyclic redundancy Code (CRC) computation functions, and so on.

**mm/:** This contains memory management code.

**net/:** This contains networking (whatever network type it is) protocols code.

**scripts/:** This contains scripts and tools used during kernel development. There are other useful tools here.

**security/:** This directory contains the security framework code.

**sound/:** Audio subsystems code is here.

**usr/:** This currently contains the initramfs implementation  
**---------------------------------------------------------------------------------------------------------------------------------------------------**

**Device Driver:** A driver is a piece of software whose aim is to control and manage a particular hardware device, hence the name device driver. From an operating system point of view, drivers can be either in the kernel space (running in privileged mode) or in the user space (with lower privileges)

**Kernel space:** This is a set of addresses where the kernel is hosted and where it runs. Kernel memory (or kernel space) is a memory range, owned by the kernel, protected by access flags, preventing any user apps from messing with the kernel (un)knowingly. On the other hand, the kernel can access the whole system memory, since it runs with the highest priority on the system. In kernel mode, the CPU can access the whole memory (both kernel space and user space).

**User space:** This is a set of addresses (locations) where normal programs (such as gedit and so on) are restricted to run. You may consider it a sandbox or a jail, so that a user program can't mess with memory or any other resource owned by another program. In user mode, the CPU can only access memory tagged with user space access rights. **The only way for a user app to run in the kernel space is through system calls. Examples of these are read, write, open, close, mmap, and so on. User space code runs with a lower priority. When a process performs a system call, a software interrupt is sent to the kernel, which turns on privileged mode so that the process can run in kernel space. When the system call returns, the kernel turns off the privileged mode and the process is jailed again**.

**The concept of modules:** A module is to the Linux kernel what a plugin (add-on) is to user software (Firefox is an example). It dynamically extends the kernel functionalities without the need to even restart the computer. Most of the time, kernel modules are plug and play. Once inserted, they are ready to be used. In order to support modules, the kernel must have been built with the following option enabled:

**CONFIG\_MODULES=y**

Module dependencies In Linux, a module can provide functions or variables, exporting them using the EXPORT\_SYMBOL macro, which makes them available for other modules. These are called symbols. A dependency of module B on module A means that module B is using one of the symbols exported by module A.

Module loading and unloading for a module to be operational, you should load it into the kernel, either by using insmod given the module path as an argument, which is the preferred method during development, or by using modprobe, a clever command and one preferred in production system

**Module entry and exit point**

Kernel drivers all have entry and exit points: the former correspond to the function called when the module is loaded (modprobe, insmod) and the latter are the function executed at module unloading (at rmmod or modprobe -r). We all remember the main() function, which is the entry point for every user space program written in C/C++; it exits when that same function returns. With kernel modules, things are different. The entry point can have any name you want, and unlike a user space program that exits when main() returns, the exit point is defined in another function. All you need to do is to inform the kernel which functions should be executed as an entry or exit point. The only thing that is actually mandatory is to identify them as the corresponding loading and removing functions, giving them as parameters to the module\_init() and module\_exit() macros. To sum up, module\_init() is used to declare the function that should be called when the module is loaded (with insmod or modprobe). What is done in the initialization function will define the behavior of the module. module\_exit() is used to declare the function that should be called when the module is unloaded (with rmmod).

**\_\_init and \_\_exit attributes**

\_\_init and \_\_exit are actually kernel macros, defined in include/linux/init.h, shown as follows: #define \_\_init \_\_section(.init.text) #define \_\_exit \_\_section(.exit.text) The \_\_init keyword tells the linker to place the code in a dedicated section into the kernel object file. This section is known in advance to the kernel, and freed when the module is loaded and the init function finished. This applies only to built-in drivers, not to loadable modules. The kernel will run the init function of the driver for the first time during its boot sequence.

Since the driver cannot be unloaded, its init function will not be called again until the next reboot. There is no need to keep references on its init function anymore. The same goes for the \_\_exit keyword, whose corresponding code is omitted when the module is compiled statically into the kernel or when module unloading support is not enabled because, in both cases, the exit function is never called.

**Licensing**

MODULE\_LICENSE ("GPL");

y. Remember that the module without MODULE\_LICENSE() is not considered open source and will taint the kernel too. Refer /linux/module.h for available licenses.

**Message printing – printk()**

printk() is to the kernel what printf() is to the user space. Lines written by printk() can be displayed through the dmesg command. Depending on how important the message you need to print is, you can choose between eight log-level messages, defined in include/linux/kern\_levels.h, along with their meaning:

Module parameters as a user program does, a kernel module can accept arguments from the command line. This allows dynamically changing the behavior of the module according to the given parameters, and can avoid the developer having to indefinitely change/compile the module during a test/debug session. In order to set this up, you should first declare the variables that will hold the values of command line arguments, and use the module\_param() macro on each of these. The macro is defined in include/linux/moduleparam.h (this should be included in the code too: #include ),

shown as follows: module\_param(name, type, perm);

This macro contains the following elements:

**name:** The name of the variable used as the parameter

**type:** The parameter's type (bool, charp, byte, short, ushort, int, uint, long, ulong), where charp stands for char pointer

**perm:** This represents the /sys/module//parameters/ file permissions. Some of them are S\_IWUSR, S\_IRUSR, S\_IXUSR, S\_IRGRP, S\_WGRP, and S\_IRUGO, where: S\_I is just a prefix R: read, W: write, X: execute USR: user, GRP: group, UGO: user, group, others

**Class Work**

* EXPORT\_SYMBOL(func)
* -> usage count
* EXPORT\_SYMBOL\_GPL(func)

#include<linux/module\_param.h>  
module\_param(val,int,S\_IRUGO|S\_IWUSR);  
MODULE\_PARM\_DESC(val,"Init at intialisation");

insmod module\_param val=500

cd /sys/modules/module\_param/parameters

echo 700 > val

static char \*mystr = "hello";  
static int myint = 1;  
static int myarr[3] = {0, 1, 2};

module\_param(myint, int, S\_IRUGO);  
module\_param(mystr, charp, S\_IRUGO);  
module\_param\_array(myarr, int,NULL, S\_IWUSR|S\_IRUSR); /\* \*/

MODULE\_PARM\_DESC(myint,"this is my int variable");  
MODULE\_PARM\_DESC(mystr,"this is my char pointer variable");  
MODULE\_PARM\_DESC(myarr,"this is my array of int")

insmod hellomodule-params.ko mystring="sathish" myint=15 myArray=1,2,3

objdump -h helloworld.ko  
objdump helloworld.ko -d -j .modinfo  
modinfo ./helloworld.ko   
modinfo ./mod\_param.ko

/dev/i2c-0

**Device Drivers**

1. (User Interaction ) Kernel Specific: interact with app

2. Hardware Specific: interact with hardware

**User Interaction**

1. char drivers

2. Block drivers

3. Network drivers

**Steps to implement char drivers:**

1. create device file

2. implement driver operations and register with vfs

3. insert driver using kernel module

4. write application

**Device File Creation:**

mknod /dev/bitsilica\_testdev c 200 0

mknod /dev/bitsilica\_testdev c 200 0

*each of device file identified as unique-id*

cat /proc/devices | more

ls -l /dev/bitsilica\_testdev

/home/luser/Training/Day1/poky/build/tmp/work-shared/qemux86/kernel-source/include/linux/fs.h

struct file\_operations

Youtube -> mycodeschool

***Home work:***

struct list  
{  
 int a;  
 chat b;  
 float c;  
};

static list \*test;

test->b;

**Programs**

**Mod\_param.c**

#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/moduleparam.h>

void func(void);  
static int val=100;  
module\_param(val,int,S\_IWUSR|S\_IRUGO);  
MODULE\_PARM\_DESC(val,"this is my int variable");

void func(void)  
{   
 printk("Function invoked \n");  
 printk("Val = %d\n",val);  
}

static int \_\_init hellowolrd\_init(void) {  
 printk("Hello world loaded!\n");  
 func();  
 return 0;  
}

static void \_\_exit hellowolrd\_exit(void) {  
 printk("End of the world\n");  
}

module\_init(hellowolrd\_init);  
module\_exit(hellowolrd\_exit);  
MODULE\_AUTHOR("Sathish [<csathish.micro@gmail.com](mailto:<csathish.micro@gmail.com)>");  
MODULE\_LICENSE("GPL");

**Helloworld.c**

#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kernel.h>

void func(void);

int val=100;

EXPORT\_SYMBOL\_GPL(func);

void func(void)  
{   
 printk("Function invoked \n");  
}

static int \_\_init hellowolrd\_init(void) {  
 printk("Hello world loaded!\n");  
 return 0;  
}

static void \_\_exit hellowolrd\_exit(void) {  
 printk("End of the world\n");  
}

module\_init(hellowolrd\_init);  
module\_exit(hellowolrd\_exit);  
MODULE\_AUTHOR("Sathish [<csathish.micro@gmail.com](mailto:<csathish.micro@gmail.com)>");  
MODULE\_LICENSE("GPL");

**Depmod.c**

#include <linux/init.h>  
#include <linux/module.h>  
#include <linux/kernel.h>

extern void func(void);

static int \_\_init hellowolrd\_init(void) {  
 printk("depmod Hello world loaded!\n");  
 func();  
 return 0;  
}

static void \_\_exit hellowolrd\_exit(void) {  
 printk("depmod End of the world\n");  
}

module\_init(hellowolrd\_init);  
module\_exit(hellowolrd\_exit);  
MODULE\_AUTHOR("Sathish [<csathish.micro@gmail.com](mailto:<csathish.micro@gmail.com)>");  
MODULE\_LICENSE("MIT");

**Char\_driver.c**

#include <linux/module.h>  
#include <linux/kernel.h>  
#include <linux/version.h>  
#include <linux/fs.h>  
#include <linux/cdev.h>

#define MYMAJOR 110  
#define MYMINOR 0  
#define DRVNAME "bitsilica"

static struct cdev \*mycdev;  
static dev\_t devid;

static int mydrv\_open(struct inode \*inode, struct file \*file)  
{  
 pr\_info("Drivers open call invoked\n");  
 return 0;  
}

static int mydrv\_close(struct inode \*inode, struct file \*file)  
{  
 pr\_info("Drivers release call invoked\n");  
 return 0;  
}

static ssize\_t mydrv\_read(struct file \*file, char \_\_user \*ptr, size\_t count, loff\_t \*off)  
{  
 pr\_info("Drivers read invoked");  
 return 0;  
}

static ssize\_t mydrv\_write(struct file \*file,const char \_\_user \*ptr,size\_t count,loff\_t \*off)  
{  
 pr\_info("Drivers write invoked");  
 return 0;  
}

static struct file\_operations mydrv\_ops = {  
 .owner = THIS\_MODULE,  
 .read = mydrv\_read,  
 .write = mydrv\_write,  
 .open = mydrv\_open,  
 .release = mydrv\_close,  
};

static int \_\_init drv\_init(void)  
{  
 int ret;  
 /\*step 1: Reserve major and minor no's \*/  
 devid = MKDEV(MYMAJOR, MYMINOR);  
 ret = register\_chrdev\_region(devid, 1, DRVNAME);

/\*step 2: register driver with vfs \*/  
 mycdev = cdev\_alloc();  
 cdev\_init(mycdev, &mydrv\_ops);  
 cdev\_add( mycdev, devid, 1);

return 0;  
}

static void \_\_exit drv\_exit(void)  
{  
 unregister\_chrdev\_region(devid, 1);  
 cdev\_del(mycdev);  
}

module\_init(drv\_init);  
module\_exit(drv\_exit);  
MODULE\_AUTHOR("Sathish Reddy ");  
MODULE\_DESCRIPTION("Basic Char driver");  
MODULE\_LICENSE("GPL");

**Makefile**

obj-m := mod\_param.o

KERNELDIR ?= /lib/modules/$(shell uname -r)/build

all default: modules

install: modules\_install

modules modules\_install help clean:$(MAKE) -C $(KERNELDIR) M=$(shell pwd) $@

**Day – 5**

User application program to access Device through Device File.

#include <stdio.h>  
#include <unistd.h>  
#include <fcntl.h>  
#include <string.h>  
#include <stdlib.h>  
#include <sys/types.h>  
#include <errno.h>

int main()  
{  
 int fd, i;  
 ssize\_t ret;  
 char my\_buf[12]="Hello world";

fd = open( "/dev/testdev\_csr", O\_RDWR );

if(fd < 0) {

printf("failed acquiring file descriptor return status %d\n",fd);  
 return fd;  
}

/\* write the contents of my buffer into the device \*/

ret = write( fd, my\_buf, 12 );  
 ret = read(fd,my\_buf,3000);

if(ret<0)

printf("read operation failed with return status %d\n",ret);

close(fd);

}

**Sir Notes:**

Steps to implement char drivers:

1. create device file

2. implement driver operations and register with vfs

3. insert driver using kernel module

4. write application

mknod /dev/testdev\_csr c 230 0

mknod /dev/bitsilica\_testdev c 200 0

**File Systems**

File systems are kernel services which are responsible for managing set of files

File systems are 2 types

1. Persistance storage file systems

2. Logical File systems

Persistance file systems: are stored over storage media. eg: ext2,etx3,FAT

Logical file systems : are stored in volatile memory. It carries some information.

End user cannot interact with these directly

cat /proc/filesystems

cd /proc -> shifting the persistanace storage system to logical filesystem

cat /proc/meminfo

cat /proc/interrupts

cat /proc/cpuinfo

-> demo application

#include<stdio.h>  
int main()  
{  
 int a;  
 printf("Hello world \n");  
 printf("enter number");  
 scanf("%d",&a);  
 return 0;  
}

ps -Af

/proc/2775

cd /proc/2775/maps

cd /proc/2775/status

**Persistance File Systems:**

Format means divide disc into blocks of sectors size

Format will reserve few block for kernel services

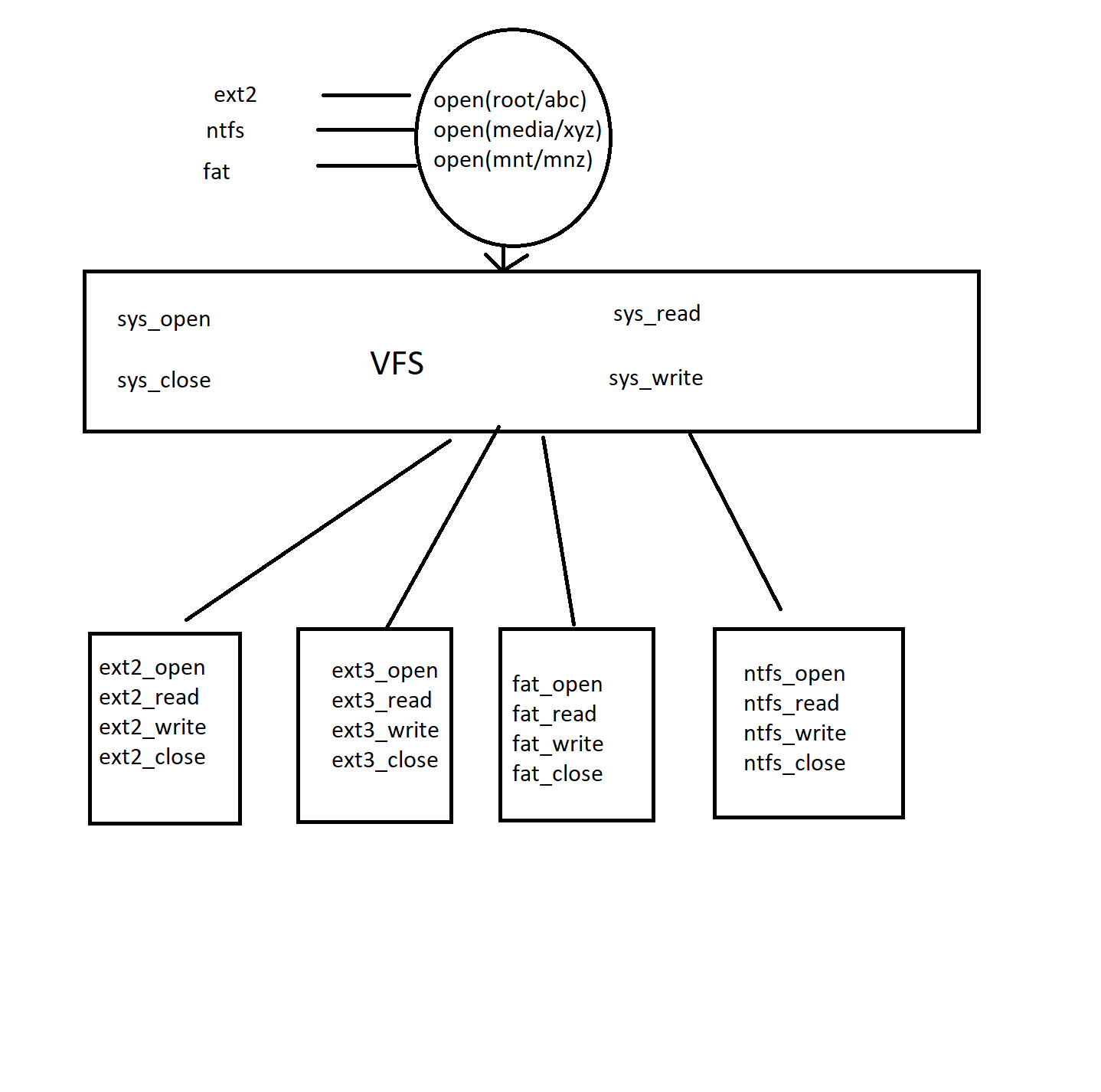
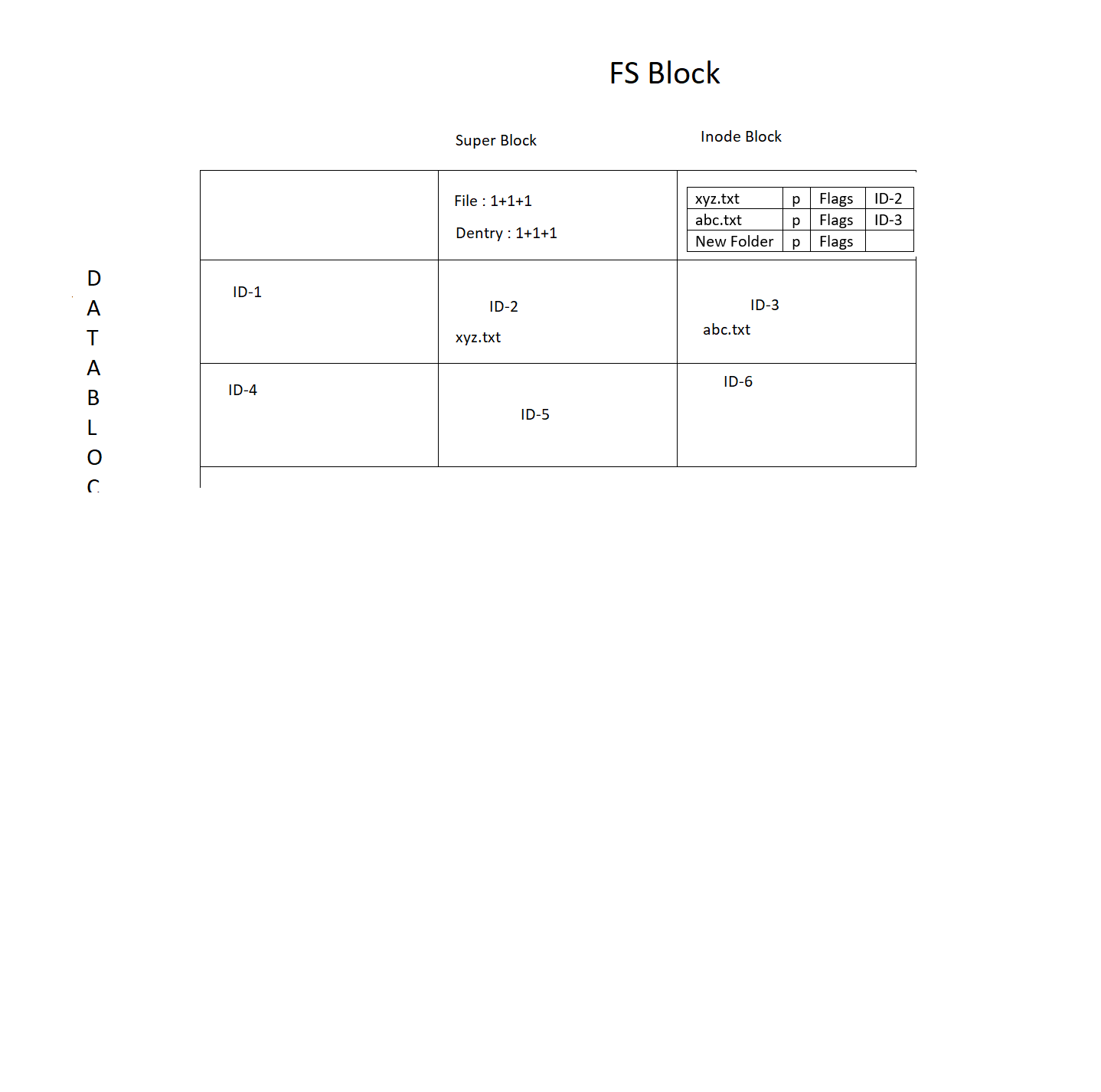
FS block maintain counter for no of files and storage of file blocks where they are stored

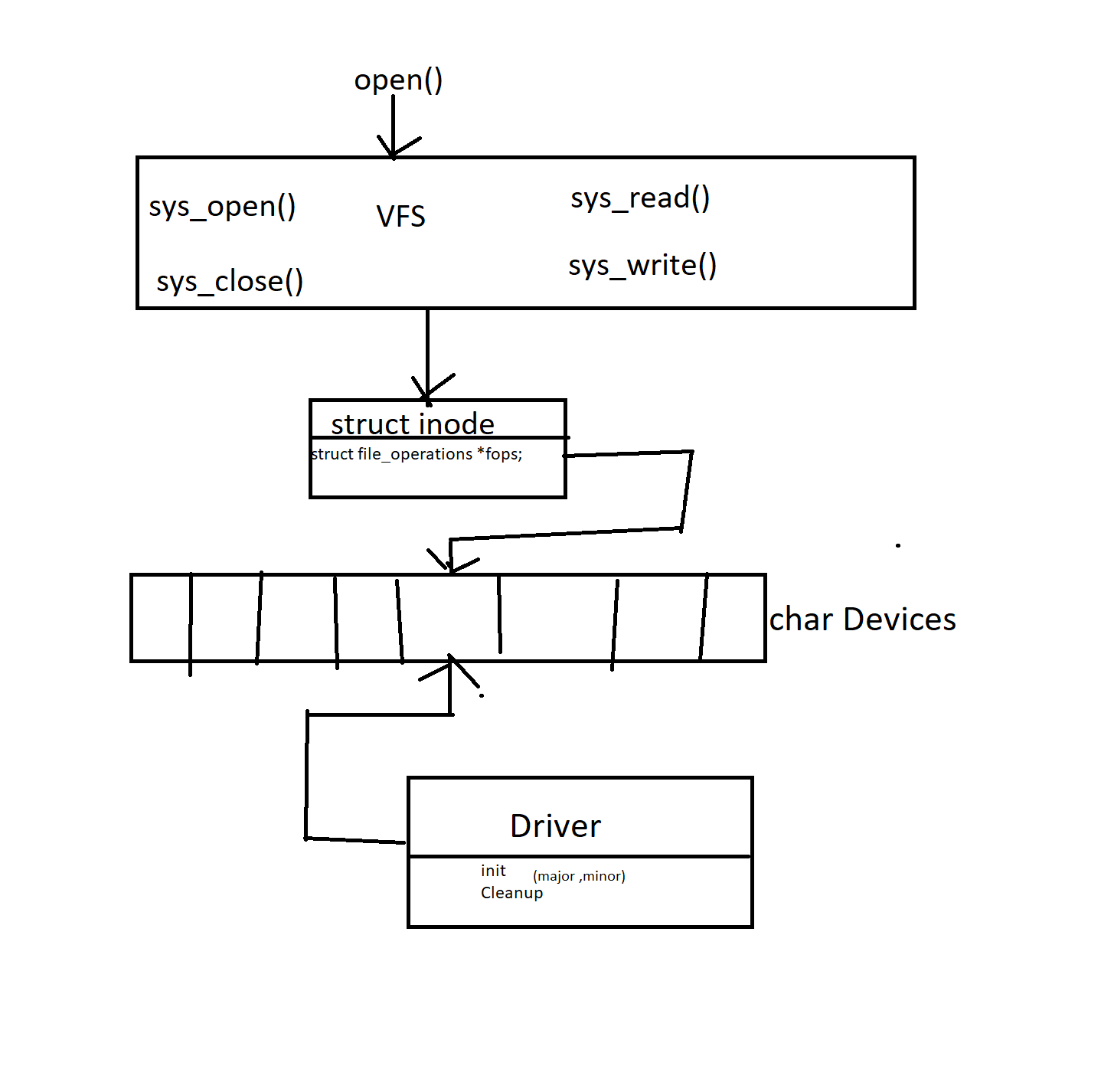
superblock stores info aboout storage ,count

inode blcok stores details aboout file

When format complete when disc connected to OS its Os actually copies superblock and inode block to its memory

mount -t fat /dev/sda /media





**Programs**

**Char\_Driver.c**

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/version.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#define MYMAJOR 230

#define MYMINOR 0

#define DRVNAME "testdrv"

static struct cdev \*mycdev;

static dev\_t devid;

static int mydrv\_open(struct inode \*inode, struct file \*file)

{

pr\_info("Drivers open call invoked\n");

return 0;

}

static int mydrv\_close(struct inode \*inode, struct file \*file)

{

pr\_info("Drivers release call invoked\n");

return 0;

}

static ssize\_t mydrv\_read(struct file \*file, char \_\_user \*ptr, size\_t count, loff\_t \*off)

{

pr\_info("Drivers read invoked");

return 0;

}

static ssize\_t mydrv\_write(struct file \*file,const char \_\_user \*ptr,size\_t count,loff\_t \*off)

{

pr\_info("Drivers write invoked");

return 0;

}

static struct file\_operations mydrv\_ops = {

.owner = THIS\_MODULE,

.read = mydrv\_read,

.write = mydrv\_write,

.open = mydrv\_open,

.release = mydrv\_close,

};

static int \_\_init drv\_init(void)

{

int ret;

/\*step 1: Reserve major and minor no's \*/

devid = MKDEV(MYMAJOR, MYMINOR);

ret = register\_chrdev\_region(devid, 1, DRVNAME);

/\* step 2: register driver with vfs \*/

mycdev = cdev\_alloc();

cdev\_init(mycdev, &mydrv\_ops);

cdev\_add( mycdev, devid, 1);

printk(" Char Driver Inserted \n");

return 0;

}

static void \_\_exit drv\_exit(void)

{

unregister\_chrdev\_region(devid, 1);

cdev\_del(mycdev);

printk("Char driver removed \n");

}

module\_init(drv\_init);

module\_exit(drv\_exit);

MODULE\_AUTHOR("Sathish Reddy ");

MODULE\_DESCRIPTION("Basic Char driver");

MODULE\_LICENSE("GPL");

**Char\_Driver\_1.c**

#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#define CHAR\_DEV\_NAME "Testdev\_1"

#define SUCCESS 0

static struct cdev \*my\_cdev;

static dev\_t mydev;

static int count = 1, inuse = 0;

static int char\_dev\_open(struct inode \*inode, struct file \*file)

{

if (inuse) {

pr\_err("Device busy %s", CHAR\_DEV\_NAME);

return -EBUSY;

}

inuse = 1;

pr\_info("Open operation invoked");

return SUCCESS;

}

static int char\_dev\_release(struct inode \*inode, struct file \*file)

{

inuse = 0;

return SUCCESS;

}

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

size\_t lbuf, loff\_t \* offset)

{

pr\_info("Rec'vd data : %s, of len=%ld", buf, lbuf);

return lbuf;

}

static struct file\_operations char\_dev\_fops = {

.owner = THIS\_MODULE,

.write = char\_dev\_write,

.open = char\_dev\_open,

.release = char\_dev\_release

};

/\* Register Driver with I/O abstarction layer

\*

\* step 1: Reserve Driver/Device ID.

\* step 2: Register file operations

\*

\*/

static int \_\_init char\_dev\_init(void)

{

int ret, fminor = 0;

ret = alloc\_chrdev\_region(&mydev, fminor, count, CHAR\_DEV\_NAME);

if (ret < 0) {

pr\_err("failed to reserve major/minor range");

return ret;

}

if (!(my\_cdev = cdev\_alloc())) {

pr\_err("cdev\_alloc() failed");

unregister\_chrdev\_region(mydev, count);

return -ENOMEM;

}

cdev\_init(my\_cdev, &char\_dev\_fops);

ret = cdev\_add(my\_cdev, mydev, count);

if (ret < 0) {

pr\_err("Error registering device driver");

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

return ret;

}

pr\_info("Device Registered: %s", CHAR\_DEV\_NAME);

pr\_info("Major number = %d, Minor number = %d", MAJOR(mydev),MINOR(mydev));

return SUCCESS;

}

static void \_\_exit char\_dev\_exit(void)

{

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

pr\_info("Driver unregistered");

}

module\_init(char\_dev\_init);

module\_exit(char\_dev\_exit);

MODULE\_AUTHOR("Sathish Reddy");

MODULE\_DESCRIPTION("Character Device Driver - Test");

MODULE\_LICENSE("GPL");

/\* End of code \*/

**Char\_Driver\_2.c**

#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/device.h>

#define CHAR\_DEV\_NAME "Testdev\_1"

#define SUCCESS 0

static struct cdev \*my\_cdev;

static struct class \*my\_class;

static dev\_t mydev;

static int count = 1, inuse = 0;

static int char\_dev\_open(struct inode \*inode, struct file \*file)

{

if (inuse) {

pr\_err("Device busy %s", CHAR\_DEV\_NAME);

return -EBUSY;

}

inuse = 1;

pr\_info("Open operation invoked");

return SUCCESS;

}

static int char\_dev\_release(struct inode \*inode, struct file \*file)

{

inuse = 0;

return SUCCESS;

}

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

size\_t lbuf, loff\_t \* offset)

{

pr\_info("Rec'vd data : %s, of len=%ld", buf, lbuf);

return lbuf;

}

static struct file\_operations char\_dev\_fops = {

.owner = THIS\_MODULE,

.write = char\_dev\_write,

.open = char\_dev\_open,

.release = char\_dev\_release

};

/\* Register Driver with I/O abstarction layer

\*

\* step 1: Reserve Driver/Device ID.

\* step 2: Register file operations

\*

\*/

static int \_\_init char\_dev\_init(void)

{

int ret, fminor = 0;

ret = alloc\_chrdev\_region(&mydev, fminor, count, CHAR\_DEV\_NAME);

if (ret < 0) {

pr\_err("failed to reserve major/minor range");

return -ret;

}

if (!(my\_cdev = cdev\_alloc())) {

pr\_err("cdev\_alloc() failed");

unregister\_chrdev\_region(mydev, count);

return -ENOMEM;

}

cdev\_init(my\_cdev, &char\_dev\_fops);

ret = cdev\_add(my\_cdev, mydev, count);

if (ret < 0) {

pr\_err("Error registering device driver");

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

return ret;

}

my\_class = class\_create(THIS\_MODULE, "VIRTUAL");

device\_create(my\_class, NULL, mydev, NULL, "%s", "vDev3");

pr\_info("Device Registered: %s", CHAR\_DEV\_NAME);

pr\_info("Major number = %d, Minor number = %d", MAJOR(mydev),

MINOR(mydev));

return SUCCESS;

}

static void \_\_exit char\_dev\_exit(void)

{

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

device\_destroy(my\_class, mydev);

class\_destroy(my\_class);

pr\_info("Driver unregistered");

}

module\_init(char\_dev\_init);

module\_exit(char\_dev\_exit);

MODULE\_AUTHOR("Sathish ");

MODULE\_DESCRIPTION("Character Device Driver - Test");

MODULE\_LICENSE("GPL");

/\* End of code \*/

**Day – 6**

* **Concurrency** - Things happening at same time
* The driver programming should take care the safety of shared resource.
* The function which takes care of shared data are called re-entrant.
* We need to ensure driver functions are re-entrant.

**Atomic Variables**

#include <asm/atomic.h>

int global\_variable ;

thread1()

{

global\_variable++; //Accessing the variable

}

thread2()

{

global\_variable++; //Accessing the variable

}

When we are doing atomic operations, that variable should be created using atomic\_t or atomic64\_t. So we have separate special functions for reading, writing, and arithmetic operations

**atomic\_t global\_variable;** /\* define etx\_global\_variable \*/

int atomic\_read(atomic\_t \*v); //This function atomically reads the value of the given atomic variable.

void atomic\_set(atomic\_t \*v, int i);//This function atomically sets the value to the atomic variable.

void atomic\_add(int i, atomic\_t \*v); //This function atomically adds value to the atomic variable.

void atomic\_sub(int i, atomic\_t \*v);//This function atomically subtracts the value from the atomic variable.

void atomic\_inc (atomic\_t \*v);

void atomic\_dec (atomic\_t \*v);

void atomic\_sub\_and\_test(int i, atomic\_t \*v);

//This function atomically subtracts the value from the atomic variable and test the result is zero or not

void atomic\_dec\_and\_test(atomic\_t \*v);

This function atomically decrements the value of the atomic variable by 1 and test the result is zero or not.

eg :

atomic\_t etx\_global\_variable = ATOMIC\_INIT(0); //Atomic integer variable

int thread\_function1(void \*pv)

{

atomic\_inc(&etx\_global\_variable);

}

int thread\_function2(void \*pv)

{

atomic\_inc(&etx\_global\_variable);

}

Atomic\_t is good when we are working on integer arithmetic. But when it comes to bitwise atomic operation, it doesn’t work well. So kernel offers separate functions to achieve that. Atomic bit operations are very fast. These functions are architecture-dependent and are declared in <asm/bitops.h>.

void set\_bit(int nr, void \*addr) Atomically set the nr-th bit starting from addr

void clear\_bit(int nr, void \*addr) Atomically clear the nr-th bit starting from addr

void change\_bit(int nr, void \*addr) Atomically flip the value of the nr-th bit starting from addr

int test\_and\_set\_bit(int nr, void \*addr) Atomically set the nr-th bit starting from addr and return the previous value

int test\_and\_clear\_bit(int nr, void \*addr) Atomically clear the nr-th bit starting from addr and return the previous value

int test\_and\_change\_bit(int nr, void \*addr) Atomically flip the nr-th bit starting from addr and return the previous value

int test\_bit(int nr, void \*addr) Atomically return the value of the nr-th bit starting from addr

int find\_first\_zero\_bit(unsigned long \*addr, unsigned int size) Atomically returns the bit-number of the first zero bit, not the number of the byte containing a bit

int find\_first\_bit(unsigned long \*addr, unsigned int size) Atomically returns the bit-number of the first set bit, not the number of the byte containing a b

void \_set\_bit(int nr, void \*addr) Non-atomically set the nr-th bit starting from addr

void \_clear\_bit(int nr, void \*addr) Non-atomically clear the nr-th bit starting from addr

void \_change\_bit(int nr, void \*addr) Non-atomically flip the value of the nr-th bit starting from addr

int \_test\_and\_set\_bit(int nr, void \*addr) Non-atomically set the nr-th bit starting from addr and return the previous value

int \_test\_and\_clear\_bit(int nr, void \*addr) Non-atomically clear the nr-th bit starting from addr and return the previous value

int \_test\_and\_change\_bit(int nr, void \*addr) Non-atomically flip the nr-th bit starting from addr and return the previous value

int \_test\_bit(int nr, void \*addr) Non-atomically return the value of the nr-th bit starting from addr

**Race Condition**

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, we don’t know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are “racing” to access/change the data.

**Locks:**

1.Waiting locks : semaphores,mutex

2.polling locks : spinlock

**Mutex**

* A mutex is a mutual exclusion lock. Only one thread can hold the lock.
* A mutex can be used to prevent the simultaneous execution of a block of code by multiple threads that are running in single or multiple processes.
* Mutex is used as a synchronization primitive in situations where a resource has to be shared by multiple threads simultaneously.
* A mutex has ownership. The thread which locks a Mutex must also unlock it
* so whenever you are accessing a shared resource that time first we lock the mutex and then access the shared resource. When we are finished with that shared resource then we unlock the Mutex.

Today most major operating systems employ multitasking. Multitasking is where multiple threads can execute in parallel and thereby utilizing the CPU in an optimum way. Even though, multitasking is useful, if not implemented cautiously can lead to concurrency issues (Race condition), which can be very difficult to handle.

#include<linux/mutex.h>

**Initialization**

**Static Method:**

DEFINE\_MUTEX(name)

**Dynamic Method:**

struct mutex etx\_mutex;

mutex\_init(&etx\_mutex);

void mutex\_lock(struct mutex \*lock);

--------------------------------------

* This is used to lock/acquire the mutex exclusively for the current task. If the mutex is not available, the current task will sleep until it acquires the Mutex.
* The mutex must, later on, be released by the same task that acquired it. Recursive locking is not allowed. The task may not exit without first unlocking the mutex. Also, kernel memory where the mutex resides must not be freed with the mutex still locked. The mutex must first be initialized (or statically defined) before it can be locke

int mutex\_lock\_interruptible(struct mutex \*lock);

-------------------------------------------------

* Locks the mutex like mutex\_lock, and returns 0 if the mutex has been acquired or sleeps until the mutex becomes available. If a signal arrives while waiting for the lock then this function returns -EINTR.

int mutex\_trylock(struct mutex \*lock);

--------------------------------------

* This will try to acquire the mutex, without waiting (will attempt to obtain the lock, but will not sleep). Returns 1 if the mutex has been acquired successfully, and 0 on contention.

void mutex\_unlock(struct mutex \*lock);

----------------------------------------

* This is used to unlock/release a mutex that has been locked by a task previously.

int mutex\_is\_locked(struct mutex \*lock);

----------------------------------------

* This function is used to check whether mutex has been locked or not.

eg:

struct mutex etx\_mutex;

int thread\_function1(void \*pv)

{

{

mutex\_lock(&etx\_mutex);

etx\_global\_variable++;

pr\_info("In EmbeTronicX Thread Function1 %lu\n", etx\_global\_variable);

mutex\_unlock(&etx\_mutex);

msleep(1000);

}

return 0;

}

int thread\_function2(void \*pv)

{

{

mutex\_lock(&etx\_mutex);

etx\_global\_variable++;

pr\_info("In EmbeTronicX Thread Function2 %lu\n", etx\_global\_variable);

mutex\_unlock(&etx\_mutex);

msleep(1000);

}

return 0;

}

**SpinLock**

* In the Mutex concept, when the thread is trying to lock or acquire the Mutex which is not available then that thread will go to sleep until that Mutex is available. Whereas in Spinlock it is different. The spinlock is a very simple single-holder lock. If a process attempts to acquire a spinlock and it is unavailable, the process will keep trying (spinning) until it can acquire the lock. This simplicity creates a small and fast lock.
* If the kernel is running on a uniprocessor and CONFIG\_SMP, CONFIG\_PREEMPT aren’t enabled while compiling the kernel then spinlock will not be available. Because there is no reason to have a lock when no one else can run at the same time.
* But if you have disabled CONFIG\_SMP and enabled CONFIG\_PREEMPT then spinlock will simply disable preemption, which is sufficient to prevent any races.

**Initialize**

**Static Method:**

DEFINE\_SPINLOCK(etx\_spinlock);

The macro given above will create a spinlock\_t variable in the name of etx\_spinlock and initialize to   
 UNLOCKED STATE.

**Dynamic Method:**

spinlock\_t etx\_spinlock;

spin\_lock\_init(&etx\_spinlock);

**between Kernel Threads**

spin\_lock(spinlock\_t \*lock) //This will take the lock if it is free, otherwise, it’ll spin until that lock is free (

spin\_trylock(spinlock\_t \*lock)

//Locks the spinlock if it is not already locked. If unable to obtain the lock it exits with an error and does not   
 spin.

spin\_unlock(spinlock\_t \*lock)

//It does the reverse of the lock. It will unlock which is locked by the above call.

spin\_is\_locked(spinlock\_t \*lock)

//This is used to check whether the lock is available or not.

eg:

int thread\_function1(void \*pv)

{

{

spin\_lock(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "In EmbeTronicX Thread Function1 %lu\n", etx\_global\_variable);

spin\_unlock(&etx\_spinlock);

msleep(1000);

}

return 0;

}

int thread\_function2(void \*pv)

{

{

spin\_lock(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "In EmbeTronicX Thread Function2 %lu\n", etx\_global\_variable);

spin\_unlock(&etx\_spinlock);

msleep(1000);

}

return 0;

}

**Locking between interrupts**

same as above

**Locking between threads and interrupts**

spin\_lock\_bh(spinlock\_t \*lock)

-------------------------------

* It disables soft interrupts on that CPU, then grabs the lock. This has the effect of preventing softirqs, tasklets, and bottom halves from running on the local CPU. Here the suffix ‘\_bh‘ refers to “Bottom Halves“.

spin\_unlock\_bh(spinlock\_t \*lock)

------------------------------

* It will release the lock and re-enables the soft interrupts which are disabled by the above call

int thread\_function(void \*pv)

{

{

spin\_lock\_bh(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "In EmbeTronicX Thread Function %lu\n", etx\_global\_variable);

spin\_unlock\_bh(&etx\_spinlock);

msleep(1000);

}

return 0;

}

/\*Tasklet Function\*/

void tasklet\_fn(unsigned long arg)

{

spin\_lock\_bh(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "Executing Tasklet Function : %lu\n", etx\_global\_variable);

spin\_unlock\_bh(&etx\_spinlock);

}

**Locking between Hard IRQ and Bottom Halves**

spin\_lock\_irq(spinlock\_t \*lock) //This will disable interrupts on that CPU, then grab the lock.

spin\_unlock\_irq(spinlock\_t \*lock) // It will release the lock and re-enables the interrupts which are disabled by the above call.

spin\_lock\_irqsave( spinlock\_t \*lock, unsigned long flags );

//This will save whether interrupts were on or off in a flags word and grab the lock.

spin\_unlock\_irqrestore( spinlock\_t \*lock, unsigned long flags );

//This will release the spinlock and restores the interrupts using the flags argument.

void tasklet\_fn(unsigned long arg)

{

spin\_lock\_irq(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "Executing Tasklet Function : %lu\n", etx\_global\_variable);

spin\_unlock\_irq(&etx\_spinlock);

}

//Interrupt handler for IRQ 11.

static irqreturn\_t irq\_handler(int irq,void \*dev\_id) {

spin\_lock\_irq(&etx\_spinlock);

etx\_global\_variable++;

printk(KERN\_INFO "Executing ISR Function : %lu\n", etx\_global\_variable);

spin\_unlock\_irq(&etx\_spinlock);

/\*Scheduling Task to Tasklet\*/

tasklet\_schedule(tasklet);

return IRQ\_HANDLED;

**Day – 7**

**Exchanging data between kernel space and user space**

* unsigned long **copy\_from\_user**(void \*to, const void \_\_user \*from, unsigned long n)
* unsigned long **copy\_to\_user**(void \_\_user \*to, const void \*from, unsigned long n)

In both cases,

pointers prefixed with **\_\_user** point to user space (untrusted) memory.

n-> represents the number of bytes to copy.

from ->represents the source address,

to-> is the destination address.

Each of these returns the number of bytes that could not be copied. On success, the return value should be 0.

When it comes to copying single and simple variables, such as char and int, but not larger

data types, such as structures or arrays, the kernel offers dedicated macros in order to

quickly perform the desired operation.

These macros are

* put\_user(x, ptr)
* get\_user(x, ptr)

**put\_user(x, ptr);** This macro copies a variable from kernel space to user space. x represents the value to copy to user space, and ptr is the destination address in user space. The macro returns 0 on success, or -EFAULT on error. X must be assignable to the result of dereferencing ptr. In other words, they must have (or point to) the same type.

**get\_user(x, ptr);** This macro copies a variable from user space to kernel space, and returns 0 on success or -EFAULT on error. Please do note that x is set to 0 on error. x represents the kernel variable to store the result, and ptr is the source address in user space. The result of dereferencing ptr must be assignable to x without a cast. Guess what it means

**IOCTL**

A typical Linux system contains around 350 system calls (syscalls), but only a few of them are linked with file operations. Sometimes devices may need to implement specific commands that are not provided by system calls, and especially the ones associated with files and thus device files. In this case, the solution is to use input/output control(ioctl), which is a method by which you extend a list of syscalls (actually commands) associated with a device. You can use it to send special commands to devices (reset, shutdown, configure, and so on). If the driver does not define this method, the kernel will return an -ENOTTY error to any ioctl() system call.

In order to be valid and safe, an ioctl command needs to be identified by a number, which should be unique to the system. The uniqueness of ioctl numbers across the system will prevent it from sending the right command to the wrong device, or passing the wrong argument to the right command (given a duplicated ioctl number). Linux provides four helper macros to create an ioctl identifier, depending on whether there is data transfer or not and on the direction of the transfer.

\_IO(MAGIC, SEQ\_NO)

\_IOW(MAGIC, SEQ\_NO, TYPE)

\_IOR(MAGIC, SEQ\_NO, TYPE)

\_IORW(MAGIC, SEQ\_NO, TYPE)

\_IO: The ioctl does not need data transfer

\_IOW: The ioctl needs write parameters (copy\_from\_user or get\_user)

\_IOR: The ioctl needs read parameters (copy\_to\_user or put\_user)

\_IOWR: The ioctl needs both write and read parameters

1. A number coded on 8 bits (0 to 255), called a magic number

2. A sequence number or command ID, also on 8 bits

3. A data type, if any, that will inform the kernel about the size to be copied

* Documentation/ioctl/ioctl-decoding.txt
* Documentation/ioctl/ioctlnumber.txt, a good place to start when you need to create an ioctl command.

**Generating ioctl numbers (command)**

You should generate their own ioctl number in a dedicated header file. It is not mandatory, but it is recommended, since this header should be available in user space too. In other words, you should duplicate the ioctl header file so that there is one in the kernel and one in the user space, which you can include in user apps.

steps:

1. identify special operations on device

2. create request commands

3. implement support for commands

#ifndef PACKT\_IOCTL\_H

#define PACKT\_IOCTL\_H

/\*

\* We need to choose a magic number for our driver, and sequential numbers

\* for each command:

\*/

#define EEP\_MAGIC 'E'

#define ERASE\_SEQ\_NO 0x01

#define RENAME\_SEQ\_NO 0x02

#define ClEAR\_BYTE\_SEQ\_NO 0x03

#define GET\_SIZE 0x04

/\*

\* Partition name must be 32 byte max

\*/

#define MAX\_PART\_NAME 32

/\*

\* Now let's define our ioctl numbers:

\*/

#define EEP\_ERASE \_IO(EEP\_MAGIC, ERASE\_SEQ\_NO)

#define EEP\_RENAME\_PART \_IOW(EEP\_MAGIC, RENAME\_SEQ\_NO, unsigned long)

#define EEP\_GET\_SIZE \_IOR(EEP\_MAGIC, GET\_SIZE, int \*)

#endif

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

There is only one step: use a switch ... case statement and

return an -ENOTTY error when an undefined ioctl command is called.

**container\_of macro :**

container\_of macro is used to find the container of the given field of a structure

include/linux/kernel.h

#define container\_of(ptr, type, member) ({ \

const typeof(((type \*)0)->member) \* \_\_mptr = (ptr); \

(type \*)((char \*)\_\_mptr - offsetof(type, member));

})

eg:

container\_of(pointer, container\_type, container\_field);

pointer: This is the pointer to the field in the structure

container\_type: This is the type of structure wrapping (containing) the pointer

container\_field: This is the name of the field to which pointer points inside the structure

struct person {

int age;

int salary;

char \*name;

} p;

struct person somebody;

int \*age\_ptr = &somebody.age;

struct person \*the\_person;

the\_person = container\_of(age\_ptr, struct person, age);

eg 2:

struct family {

struct person \*father;

struct person \*mother;

int number\_of\_sons;

int family\_id;

} f;

int \*fam\_id\_ptr = &f.family\_id;

struct family \*fam\_ptr;

/\* now let us retrieve back its family \*/

fam\_ptr = container\_of(fam\_id\_ptr, struct family, family\_id);

The container\_of macro won't work for char \* or array members.

It means the first member of container\_of must not be a pointer to another pointer to char nor to array in the structure.

In other words, in our first example with the struct person, it would have been wrong to use name field to retrieve the containing structure.

Giving a pointer to father or mother as first parameter to container\_of should be wrong, since those members are already pointer fields in the structure. But what about retrieving the structure holding a pointer with container\_of macro ?If you only have a pointer to father or mother (that is, if you just have struct person \*dad or struct person \*mom), you cannot use the container\_of macro to retrieve the container struct family. To use it properly, you'll need a pointer to a pointer to struct person (that is, struct person \*\*dad or struct person \*\*mom) and use it like   
struct family \*fam = container\_of(dad, struct family, father);.

**Delay and timer management**

* defer work, sleep, scheduling, timeout, and many other tasks.
* There are two categories of time. The kernel uses absolute time to know what time it is, that is, the date and time of the day, whereas relative time is used by, for example, the kernel scheduler. For absolute time, there is a hardware chip called the real-time clock (RTC).
* On the other hand,to handle relative time, the kernel relies on a CPU feature (peripheral) called a timer, from the kernel's point of view, is called a kernel timer.

**jiffies:**

Its a counter basically.

Constant HZ:

which is the number of times jiffies is incremented in one second,. Each increment is called a tick.

In other words, HZ represents the size of a jiffy.

HZ depends on the hardware and on the kernel version, and also determines how frequently the clock interrupt fires

jiffies = (seconds \* HZ)

seconds = (jiffies / Hz)

<linux/jiffies.h>

extern unsigned long volatile jiffies;

extern u64 jiffies\_64;

HZ = 100, 32-bit machine = 497 days

HZ=1000 , 32-BIT machine = 49.7 days

eg:

#include <linux/init.h>

#include <linux/kernel.h>

#include <linux/module.h>

#include <linux/timer.h>

static struct timer\_list my\_timer;

void my\_timer\_callback(unsigned long data)

{

printk("%s called (%ld).\n", \_\_FUNCTION\_\_, jiffies);

}

static int \_\_init my\_init(void)

{

int retval;

printk("Timer module loaded\n");

setup\_timer(&my\_timer, my\_timer\_callback, 0);

printk("Setup timer to fire in 300ms (%ld)\n", jiffies);

retval = mod\_timer( &my\_timer, jiffies + msecs\_to\_jiffies(300) );

if (retval)

printk("Timer firing failed\n");

return 0;

}

static void my\_exit(void)

{

int retval;

retval = del\_timer(&my\_timer);

/\* Is timer still active (1) or no (0) \*/

if (retval)

printk("The timer is still in use...\n");

pr\_info("Timer module unloaded\n");

}

module\_init(my\_init);

module\_exit(my\_exit);

MODULE\_AUTHOR("csathish.micro@gmail.com>");

MODULE\_DESCRIPTION("Standard timer example");

MODULE\_LICENSE("GPL");

#include <linux/hrtimer.h>

#include <linux/delay>

ndelay(unsigned long nsecs)

udelay(unsigned long usecs)

mdelay(unsigned long msecs)

**Programs**

**CharDriver.c**

#include <linux/module.h>

#include <linux/version.h>

#include <linux/kernel.h>

#include <linux/fs.h>

#include <linux/cdev.h>

#include <linux/device.h>

#include <asm/uaccess.h>

#include <linux/uaccess.h>

#include <linux/init.h>

#include <linux/sched.h>

#include <linux/errno.h>

#include <asm/current.h>

#define MAX\_LENGTH 4000

#define CHAR\_DEV\_NAME "Testdev\_2"

#define SUCCESS 0

static struct cdev \*my\_cdev;

static struct class \*my\_class;

static dev\_t mydev;

static int count = 1, inuse = 0;

static char \*char\_device\_buf;

static int char\_dev\_open(struct inode \*inode, struct file \*file)

{

if (inuse) {

pr\_err("Device busy %s", CHAR\_DEV\_NAME);

return -EBUSY;

}

inuse = 1;

pr\_info("Open operation invoked");

return SUCCESS;

}

static int char\_dev\_release(struct inode \*inode, struct file \*file)

{

inuse = 0;

return SUCCESS;

}

static ssize\_t char\_dev\_write(struct file \*file,const char \*buf, size\_t lbuf, loff\_t \*ppos)

{

int nbytes; /\* Number of bytes written \*/

int bytes\_to\_do; /\* Number of bytes to write \*/

int maxbytes; /\* Maximum number of bytes that can be written \*/

maxbytes = MAX\_LENGTH - \*ppos;

pr\_info("Drivers Write invoked");

if (maxbytes > lbuf ) bytes\_to\_do = lbuf;

else bytes\_to\_do = maxbytes;

if( bytes\_to\_do == 0 )

{

printk("Reached end of device\n");

return -ENOSPC; /\* Returns OF at write() \*/

}

nbytes = bytes\_to\_do - copy\_from\_user( char\_device\_buf + \*ppos, /\* to \*/

buf, /\* from \*/

bytes\_to\_do ); /\* how many bytes \*/

\*ppos += nbytes;

//up (Smysem);

return nbytes;

}

static ssize\_t char\_dev\_read(struct file \*file,char \*buf, size\_t lbuf, loff\_t \*ppos)

{

int maxbytes; /\* number of bytes from ppos to MAX LENGTH \*/

int bytes\_to\_do; /\* number of bytes to read \*/

int nbytes; /\* number of bytes actually read \*/

maxbytes = MAX\_LENGTH - \*ppos;

pr\_info("Drivers read invoked");

if (maxbytes > lbuf ) bytes\_to\_do = lbuf;

else bytes\_to\_do = maxbytes;

if( bytes\_to\_do == 0 )

{

printk( "Reached end of device\n");

return -ENOSPC; /\* Causes read() to return EOF \*/

}

/\*if (down interruptible (&mysem))

{

printk (KERN INFO "process %i woken up by a signal\n", current-›pid);

return -ERESTARTSYS;

}\*/

nbytes = bytes\_to\_do - copy\_to\_user( buf, /\* to \*/

char\_device\_buf + \*ppos, /\* from \*/

bytes\_to\_do ); /\* how many bytes \*/

\*ppos += nbytes;

return nbytes;

}

static ssize\_t mychardev\_read(struct file \*file, char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

uint8\_t \*data = "Hello from the kernel world!\n";

size\_t datalen = strlen(data);

if (count > datalen) {

count = datalen;

}

if (copy\_to\_user(buf, data, count)) {

return -EFAULT;

}

printk(" Read call Invoked \n");

return count;

}

static ssize\_t mychardev\_write(struct file \*file, const char \_\_user \*buf, size\_t count, loff\_t \*offset)

{

size\_t maxdatalen = 30, ncopied;

uint8\_t databuf[maxdatalen];

if (count < maxdatalen) {

maxdatalen = count;

}

ncopied = copy\_from\_user(databuf, buf, maxdatalen);

if (ncopied == 0) {

printk("Copied %zd bytes from the user\n", maxdatalen);

} else {

printk("Could't copy %zd bytes from the user\n", ncopied);

}

databuf[maxdatalen] = 0;

printk("Data from the user: %s\n", databuf);

return count;

}

static struct file\_operations char\_dev\_fops = {

.owner = THIS\_MODULE,

.write = mychardev\_write,

.read = mychardev\_read,

.open = char\_dev\_open,

.release = char\_dev\_release

};

/\* Register Driver with I/O abstarction layer

\*

\* step 1: Reserve Driver/Device ID.

\* step 2: Register file operations

\*

\*/

static int \_\_init char\_dev\_init(void)

{

int ret, fminor = 0;

ret = alloc\_chrdev\_region(&mydev, fminor, count, CHAR\_DEV\_NAME);

if (ret < 0) {

pr\_err("failed to reserve major/minor range");

return -ret;

}

if (!(my\_cdev = cdev\_alloc())) {

pr\_err("cdev\_alloc() failed");

unregister\_chrdev\_region(mydev, count);

return -ENOMEM;

}

cdev\_init(my\_cdev, &char\_dev\_fops);

ret = cdev\_add(my\_cdev, mydev, count);

if (ret < 0) {

pr\_err("Error registering device driver");

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

return ret;

}

my\_class = class\_create(THIS\_MODULE, "VIRTUAL");

device\_create(my\_class, NULL, mydev, NULL, "%s", "bitsilica\_device");

pr\_info("Device Registered: %s", CHAR\_DEV\_NAME);

pr\_info("Major number = %d, Minor number = %d", MAJOR(mydev),

MINOR(mydev));

return SUCCESS;

}

static void \_\_exit char\_dev\_exit(void)

{

cdev\_del(my\_cdev);

unregister\_chrdev\_region(mydev, count);

device\_destroy(my\_class, mydev);

class\_destroy(my\_class);

pr\_info("Driver unregistered");

}

module\_init(char\_dev\_init);

module\_exit(char\_dev\_exit);

MODULE\_AUTHOR("Sathish ");

MODULE\_DESCRIPTION("Character Device Driver - Test");

MODULE\_LICENSE("GPL");

/\* End of code \*/

**Reader.c**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <stdlib.h>

#include <sys/types.h>

int main()

{

int fd,i,ret;

char data[40];

unsigned long size;

bzero(data,40);

fd = open("/dev/bitsilica\_device",O\_RDWR);

if(fd < 0)

{

printf("Device Open Failed\n");

return 1;

}

if( (ret = read(fd,data,40)) < 0)

{

printf("Bytes Read %s \n",data);

}

printf("Read data = %s",data);

close(fd);

}

**Writer.c**

#include <stdio.h>

#include <unistd.h>

#include <fcntl.h>

#include <string.h>

#include <stdlib.h>

#include <sys/types.h>

int main()

{

int fd,i;

char data[40];

unsigned long size;

strcpy(data,"Welcome to BitSilica\n");

fd = open("/dev/bitsilica\_device",O\_RDWR);

if(fd < 0)

{

printf("Device Open Failed\n");

return 1;

}

size = (unsigned long)write(fd,data,strlen(data));

// printf("Bytes Written %d\n",size);

// bzero(data,20);

close(fd);

}