Kernel Module Development using IMX6ULL

Download and Build the Linux kernal

Click the below link and follow the steps to set your host machine for Cross-Compilation then download and build the kernel from the source code.

Link for download and build kernal

Boot from TFTP/NFS server

Setup Networking

Click the below link and follow the steps to Install and configure a DHCP, TFTP and NFS server on your host PC.

Link for Network setup

Boot

Root File System:

Create a root file system for your embedded Linux system. You can use Buildroot, Yocto Project, or create a custom file system.

NFS Configuration

Create an export configuration

```
#vim /etc/exports
```

```
# /etc/exports: the access control list for filesystems which may be exported
#               to NFS clients.  See exports(5).
#
# Example for NFSv2 and NFSv3:
# /srv/homes       hostname1(rw,sync,no_subtree_check) hostname2(ro,sync,no_subtree_check)
#
# Example for NFSv4:
# /srv/nfs4        gss/krb5i(rw,sync,fsid=0,crossmnt,no_subtree_check)
# /srv/nfs4/homes  gss/krb5i(rw,sync,no_subtree_check)
#
/media/neeveecomm/data1/deepak/rootfsDemo 192.168.1.0/24(rw,nohide,insecure,no_subtree_check,async,no_root_squash)
```

TFTP Configuration

- Set up a TFTP server on your host machine to serve the kernel image and root file system.

- Copy the kernel image **zImage** and **.dtb file** to the TFTP server's root directory.

```
$cd linux/arch/arm/boot/
$sudo cp zImage /tftp
$cd linux/arch/arm/boot/dts
$sudo cp imx6ull-colibri-wifi-eval-v3.dtb /tftp
```

```
-rw-r--r-- 1 root root    46536 Sep 15 12:34 imx6ull-colibri-wifi-eval-v3.dtb
-rw-r--r-- 1 root root 11790768 Sep 22 20:16 zImage
neeveecomm@neeveecomm-System-Product-Name:/tftp$
```

Hardware Connections

Ensure that your Colibri i.MX6ULL board is properly connected to your network using an Ethernet cable.

U-Boot Configuration

- Power on the Colibri i.MX6ULL board and interrupt (by press any key) the U-Boot bootloader to access its command line interface.
- Use U-Boot commands to set the necessary environment variables for networking.

```
setenv ipaddr <IP_ADDRESS>
setenv serverip <TFTP_SERVER_IP>
setenv gatewayip <GATEWAY_IP>
```

Boot Arguments

Configure the U-Boot bootargs variable to specify the kernel command-line parameters, including the root file system's location.

```
setenv bootargs 'console=ttymxc0,115200 root=/dev/nfs
nfsroot=<NFS_SERVER_IP>:<NFS_ROOT_PATH>
ip=<BOARD_IP>:<HOST_IP>:<GATEWAY_IP>:<NETMASK>:<HOSTNAME>:<INTERFACE>:<DEVICE>:<NFS_OP
TIONS>'
```

Booting

Use U-Boot to load and boot the Linux kernel via TFTP.

```
tftpboot ${loadaddr} zImage
bootz ${loadaddr} - ${fdt_addr}
```

Login

Login using the root user.

```
192 login: root
root@192:~#
```

First Device Driver

Init and Exit function

We you to two separate function for starting and ending. For this init and exit function require a set of header file.

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

Init function

The function will be executed first when device driver is loaded into the Kernel.

```
static int __init function_name(void)
{
    return 0;
}
```

The Init function should be register by using the **module_init()** function.

```
module_init(Init_function_name);
```

Exit function

The function will be executed last when the device driver is unloaded from the Kernel.

```
void __exit function_name(void)
{

}
```

The exit function should be register by using the **module_init()** function.

```
module_exit(Init_function_name);
```

Printk function

printk() is the kernel level function using this function we can able to print out to different log levels in kernel space. we can able to see the print by using the **dmesg** command.

Example 1. printk()

```
printk(KERN_INFO "First Device Driver");
```

Macros used for printk:

To control the priority of the printk message.

KERN_EMERG
: Used for emergency messages, usually those that precede a crash.

KERN_ALERT
: A situation requiring immediate action.

KERN_WARNING
: Warnings about problematic situations that do not, in themselves, create serious problems with the system.

KERN_NOTICE
: Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

KERN_INFO
: Informational messages. Many drivers print information about the hardware they find at startup time at this level.

KERN_DEBUG
: Used for debugging messages.

License

GPL, or the GNU General Public License, is an open-source license meant for software. If your software is licensed under the terms of the GPL, it is free.

```
MODULE_LICENSE("GPL");
MODULE_LICENSE("GPL v2");
MODULE_LICENSE("Dual BSD/GPL");
```

hello.c

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

static int __init hello_init(void)
{
    printk(KERN_INFO "My first module is inserted\n");
    return 0;
}

static void __exit hello_exit(void)
{
    printk(KERN_INFO "My first module is Removed\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE("GPL");
```

Makefile

We need currently running kernel version of the **target board** to get that give the command.

```
root@192:~# uname -r
6.1.0s
```

Building a Kernel Module.

make -C <path to linux Kernel tree> M=<path to your module> [target]

You can cross-compile the Linux device driver using the below command.

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -C <path to linux Kernel tree> M=<path to your module> [target]

```
obj-m := hello.o
KER_DIR=/media/neeveecomm/data1/deepak/rootfsDemo/lib/modules/6.1.0/build
all:
        $(MAKE) ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C $(KER_DIR) M=$(PWD)
modules
clean:
        $(MAKE) ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C $(KER_DIR) M=$(PWD)
clean
```

To compile the module for your **host** pc. In this make file remove cross-compile command.

Compiling, Inserting and Removing

In the terminal to create the **.ko** you need to enter the command.

```
make all
```

```
neeveecomm@neeveecomm-System-Product-Name:/media/neeveecomm/data1/deepak/rootfsDemo/module_program$ ls -l
total 12
drwxrwxr-x 2 neeveecomm neeveecomm 4096 Sep 28 12:42 char
-rw-rw-r-- 1 neeveecomm neeveecomm  663 Sep 20 19:49 hello.c
-rw-rw-r-- 1 neeveecomm neeveecomm  270 Sep 28 12:43 Makefile
neeveecomm@neeveecomm-System-Product-Name:/media/neeveecomm/data1/deepak/rootfsDemo/module_program$ make all
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- -C /media/neeveecomm/data1/deepak/rootfsDemo/lib/modules/6.1.0/
/rootfsDemo/module_program modules
make[1]: Entering directory '/home/neeveecomm/workdir/linux'
warning: the compiler differs from the one used to build the kernel
  The kernel was built by: arm-none-linux-gnueabihf-gcc (GNU Toolchain for the A-profile Architecture 9.2-2019.1
  You are using:           arm-linux-gnueabihf-gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
  CC [M]  /media/neeveecomm/data1/deepak/rootfsDemo/module_program/hello.o
  MODPOST /media/neeveecomm/data1/deepak/rootfsDemo/module_program/Module.symvers
  CC [M]  /media/neeveecomm/data1/deepak/rootfsDemo/module_program/hello.mod.o
  LD [M]  /media/neeveecomm/data1/deepak/rootfsDemo/module_program/hello.ko
make[1]: Leaving directory '/home/neeveecomm/workdir/linux'
neeveecomm@neeveecomm-System-Product-Name:/media/neeveecomm/data1/deepak/rootfsDemo/module_program$
```

compilation has been done without any error now the **hello.ko** is created.

```
neeveecomm@neeveecomm-System-Product-Name:/media/neeveecomm/data1/deepak/rootfsDemo/module_program$ ls -l
total 40
drwxrwxr-x 2 neeveecomm neeveecomm 4096 Sep 28 12:42 char
-rw-rw-r-- 1 neeveecomm neeveecomm  663 Sep 20 19:49 hello.c
-rw-rw-r-- 1 neeveecomm neeveecomm 5556 Sep 28 12:56 hello.ko
-rw-rw-r-- 1 neeveecomm neeveecomm   65 Sep 28 12:56 hello.mod
-rw-rw-r-- 1 neeveecomm neeveecomm  935 Sep 28 12:56 hello.mod.c
-rw-rw-r-- 1 neeveecomm neeveecomm 2616 Sep 28 12:56 hello.mod.o
-rw-rw-r-- 1 neeveecomm neeveecomm 3516 Sep 28 12:56 hello.o
-rw-rw-r-- 1 neeveecomm neeveecomm  270 Sep 28 12:43 Makefile
-rw-rw-r-- 1 neeveecomm neeveecomm   66 Sep 28 12:56 modules.order
-rw-rw-r-- 1 neeveecomm neeveecomm    0 Sep 28 12:56 Module.symvers
neeveecomm@neeveecomm-System-Product-Name:/media/neeveecomm/data1/deepak/rootfsDemo/module_program$
```

To cleanup enter command.

```
make clean
```

Module Inserting

Go to the terminal of target device open the created module directory. Then to load the module use the command **insmod**.

```
root@192:/module_program# insmod hello.ko
```

To see the currently loaded modules enter command **lsmod**.

```
root@192:/module_program# lsmod
Module                  Size  Used by
hello                  16384  0
mwifiex_sdio           36864  0
mwifiex               286720  1 mwifiex_sdio
imx_sdma               24576  2
evbug                  16384  0
root@192:/module_program#
```

To unload the module use the command **rmmod**.

```
root@192:/module_program# rmmod hello.ko
```

Use the command **dmesg** to see the kernel print.

```
root@192:/module_program# dmesg | tail -5
[ 1241.456458] The data[1]= 0
[ 1241.459295] The data[2]= 0
[ 1241.462190] The data[3]= 0
[ 1545.531110] My first module is inserted
[ 2134.413584] My first module is Removed
root@192:/module_program#
```

Passing Arguments to module

In your kernel module source code, declare the variables that you want to make configurable as module parameters using the **module_param** family of macros. These macros allow you to define the name, data type, permissions, and a description for each parameter.

Module parameters API

- module_param()
- module_param_array()
- module_param_cb()

Permissions of the varaible

- S_IWUSR -user write permission
- S_IRUSR -user read permission
- S_IXUSR -user execute permission
- S_IRGRP -Group read permission
- S_IWGRP -Group write permission
- S_IXGRP -Group execute permission

module_param()

- The **module_param()** macro specifies the name, data type, and permissions of a module parameter, making it accessible and configurable from outside the module.
- module_param() macro creates the sub-directory under **/sys/module**.

```
module_param(name, type, perm);
```

name variable name

type Data type (int, char, long etc...)

perm The permissions for the module parameter.

example

```
module_param(buffer, int, 0644);
```

The permission value 0644 is an octal representation of file permissions and is typically used to specify read-write permissions for the owner (module owner) and read-only permissions for others.

- 0 in the most significant digit indicates the file type (regular file).
- 6 represents read and write permissions for the owner.
- 4 represents read-only permissions for the group and others.

This will create a sysfs entry. **/sys/module/hello/parameters/buffer**

module_param_array()

This macro is used to send the array as an argument to the Linux device driver.

```
module_param_array(name,type,num,perm);
```

num is an integer variable (optional) otherwise NULL

module_param_cb()

This macro is used to register the callback. Whenever the argument (parameter) got changed, this callback function will be called.

```
module_param_cb(name, ops, get, set, perm);
```

name variable name

ops A pointer to a struct kernel_param_ops that specifies the operations for getting and setting the parameter's value.

get A callback function that is called when the parameter is read.

set A callback function that is called when the parameter is written.

perm Permission

Example code

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

int buffer = 0;
int data[4];

static int __init helloworld_init(void)
{
   pr_info("The data inside the buffer : %d \n",buffer);
   for(int i =0; i<4;i++){
           printk("The data[%d]= %d \n",i,data[i]);}
                   return 0;
}

static void __exit helloworld_cleanup(void)
{
        pr_info("good bye world\n");
        printk("The data removed from the buffer : %d \n",buffer);
          for(int i =0; i<4;i++){
          printk("The data[%d]= %d \n",i,data[i]);}

}

module_param(buffer, int, 0644);
module_param_array(data,int,NULL,0644);
module_init(helloworld_init);
module_exit(helloworld_cleanup);

MODULE_LICENSE("GPL");
```

Compile this code using the **make all** command **.ko** file is created. Then load the module using the **insmod** command. Now the variable is created at the path **/sys/modules/hello/parameters**

```
root@192:/sys/module/hello/parameters# ls -l
-rw-r--r--    1 root     root          4096 Apr 28 21:57 buffer
-rw-r--r--    1 root     root          4096 Apr 28 21:57 data
root@192:/sys/module/hello/parameters#
```

Now we pass the arguments to the varaible, unload the module using **rmmod** command.

Then load our module with arguments.

```
# insmod hello.ko buffer=30 data=10,21,30,23
```

we can check the value is correctly written in the variable **buffer** and **data** by using the command.

```
# cat /sys/module/hello/parameters/data
```

```
root@192:/module_program#  cat /sys/module/hello/parameters/data
10,21,30,23
root@192:/module_program#  cat /sys/module/hello/parameters/buffer
30
```

We can also write into the variable using the **echo** command.

```
root@192:/module_program# echo 40 > /sys/module/hello/parameters/buffer
root@192:/module_program#  cat /sys/module/hello/parameters/buffer
40
root@192:/module_program# echo 1,2,3,4 > /sys/module/hello/parameters/data
root@192:/module_program# cat /sys/module/hello/parameters/data
1,2,3,4
```

Device file

At inside the path /**proc**/**devices** which contain the character and block device. **mknod** is a command used to create special device files in the file system.

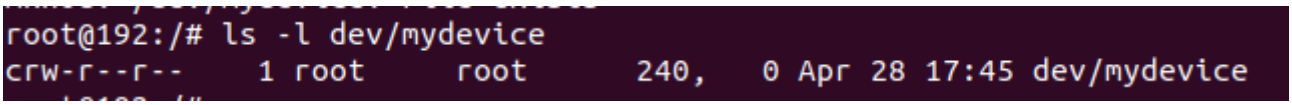
```
mknod <filename> <type> <major> <minor>
```

<**filename**>	The name of the device file to be created.
<**type**>	Specifies whether it's a character device (c) or a block device (b).
<**major**>	The major device number, which identifies the device driver.
<**minor**>	The minor device number, which identifies a specific device handled by the driver.

example

```
mknod /dev/mydevice c 240 0
```

After you enter the above example command, Now the device file is created at the file name of **mydevice** with the respective major number **240** and minor number as **0**.



Major and Minor number Allocation

We can allocate the major and minor numbers in two ways.

1. Statically allocating
2. Dynamically Allocating

Statically allocating:

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

first first device number in the range that you want to reserve.

count The total number of contiguous device numbers you are requesting.

name The name of the device that should be associated with this number range.

The return value from **register_chrdev_region** will be **0** if the allocation was successfully performed.

The **dev_t** type (defined in <linux/types.h>) is used to hold device numbers—both the major and minor parts. To create the dev_t structure variable for your major and minor number, please use the below function.

```
MKDEV(int major, int minor);
```

If you want to get your major number and minor number from dev_t, use the below method.

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Example

```
dev_t dev = MKDEV(235, 0);
```

```
register_chrdev_region(dev, 1, "Device_name");
```

Dynamically Allocating

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char
*name);
```

dev	is an output-only parameter that will, on successful completion, hold the first number in your allocated range.
firstminor	should be the requested first minor number to use; it is usually 0.
count	is the total number of contiguous device numbers you are requesting.
name	is the name of the device that should be associated with this number range; it will appear in /proc/devices and sysfs.

Unregister

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

The usual place to call unregister_chrdev_region would be in your module's cleanup function (Exit Function).

Creating Device file

The automatic creation of device files can be handled with udev. Udev is the device manager for the Linux kernel that creates/removes device nodes in the /dev directory dynamically.

Step 1:

Include the header file

```
#include <linux/device.h>
#include <linux/kdev_t.h>
```

step 2:

Create the struct Class, This will create the struct class for our device driver. It will create a structure under/sys/class/.

```
struct class * class_create(struct module *owner, const char *name);
```

owner pointer to the module that is to “own” this struct class

name pointer to a string for the name of this class

Step 3:

This function can be used by char device classes. A struct device will be created in sysfs, and registered to the specified class.

```
struct device *device_create(struct *class, struct device *parent, dev_t dev, void *
drvdata, const char *fmt, ...);
```

class pointer to the struct class that this device should be registered to

parent pointer to the parent struct device of this new device, if any

devt the dev_t for the char device to be added

drvdata the data to be added to the device for callbacks

fmt string for the device’s name

... variable arguments

step 4:

you can destroy the device using **device_destroy()**.

```
void device_destroy (struct class * class, dev_t devt);
```

File Operation

cdev Structure

- struct cdev is one of the elements of the inode structure.
- The struct cdev is the kernel’s internal structure that represents char devices. This field contains a pointer to that structure when the inode refers to a char device file.

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    const struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

There is the two ways of allocating and initializing the structure **Runtime allocation** and **own allocation**

standalone cdev structure at runtime:

```
struct cdev *my_cdev = cdev_alloc( );
my_cdev->ops = &my_fops;
```

Own allocation:

```
void cdev_init(struct cdev *cdev, struct file_operations *fops);
```

The cdev_add function in Linux device driver programming is used to add a character device represented by a struct cdev to the system and associate it with a range of device numbers (dev_t) within the kernel.

```
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
```

dev is the cdev structure.

num is the first device number to which this device responds.

count is the number of device numbers that should be associated with the device. Often count is one, but there are situations where it makes sense to have more than one device number correspond to a specific device.

If this function returns a **negative error code**, your device has not been added to the system.

To remove a char device from the system, call:

```
void cdev_del(struct cdev *dev);
```

Creating File_operation

File operation defined in,

```
#include <linux/fs.h>
```

A file_operations structure is called fops. Each field in the structure must point to the function in the driver that implements a specific operation or have to left NULL for unsupported operations.

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
```

```
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **, void **);
    long (*fallocate)(struct file *file, int mode, loff_t offset,
              loff_t len);
    void (*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
    unsigned (*mmap_capabilities)(struct file *);
#endif
    ssize_t (*copy_file_range)(struct file *, loff_t, struct file *,
            loff_t, size_t, unsigned int);
    int (*clone_file_range)(struct file *, loff_t, struct file *, loff_t,
            u64);
    ssize_t (*dedupe_file_range)(struct file *, u64, u64, struct file *,
            u64);
};
```

Example

```
static struct file_operations fops =
{
.owner          = THIS_MODULE,
.read           = chr_read,
.write          = chr_write,
.open           = chr_open,
.release        = chr_release,
```

```
};
```

Copy from user and copy to user

Copy form user

This function is used to Copy a block of data from user space to kernel space

```
unsigned long copy_from_user(void *to, const void __user *from, unsigned long  n);
```

to	Destination address, in the kernel space
from	The source address in the user space
n	Number of bytes to copy

Copy to user

This function is used to Copy a block of data from kernel space to user space.

```
unsigned long copy_to_user(const void __user *to, const void *from, unsigned long  n);
```

to	Destination address, in the user space
from	The source address in the kernel space
n	Number of bytes to copy

IOCTL

IOCTL is referred to as Input and Output Control, which is used to talk to device drivers.

Create IOCTL Command in the Driver

```
#define "ioctl name" __IOX("magic number","command number","argument type")
```

“IO“	an ioctl with no parameters
“IOW“	an ioctl with write parameters (copy_from_user)
“IOR“	an ioctl with read parameters (copy_to_user)
“IOWR“	an ioctl with both write and read parameters

The **Magic Number** is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. **Command Number** is the number that is assigned to the ioctl. The last is the **type** of data.

Header file The header file that we need to include.

```
#include <linux/ioctl.h>
```

Write IOCTL Function in the Driver

```
int  ioctl(struct inode *inode,struct file *file,unsigned int cmd,unsigned long arg)
```

<**inode**> is the inode number of the file being worked on.

<**file**> is the file pointer to the file that was passed by the application.

<**cmd**> is the ioctl command that was called from the userspace.

<**arg**> are the arguments passed from the userspace

we need to inform the kernel that the ioctl calls are implemented. This is done by making the **fops** pointer **unlocked_ioctl**

```
static struct file_operations chrdev_fops = {
        .owner = THIS_MODULE,
        .open = chrdev_open,
        .release = chrdev_release,
        .read = chrdev_read,
        .write = chrdev_write,
        .poll = chrdev_poll,
        .unlocked_ioctl = chrdev_ioctl,
};
```

Example

```
static long chrdev_ioctl(struct file *file, unsigned int command, unsigned long arg)
{
    switch(command){
        case WRITE_VALUE:
            if(copy_from_user(&value, arg, sizeof(value)))
            {
                pr_err("Data Write : Err!\n");
            }
            printk(KERN_INFO "value = %d\n",value);
            break;
        case READ_VALUE:
```

```
            if(copy_to_user(arg, &value, sizeof(value)))
            {
                pr_err("Data Read : Err!\n");
            }
            break;
        default:
            printk(KERN_INFO "Default \n");
            break;
    }
    return 0;
}
```

Create IOCTL Command in a Userspace Application

```
#define WRITE_VALUE _IOW('a','a',int*)
#define READ_VALUE _IOR('a','b',int*)
```

Use IOCTL System Call in Userspace

Include the header file.

```
#include <sys/ioctl.h>
```

```
long ioctl( "file descriptor","ioctl command","Arguments");
```

<**file descriptor**>	This the open file on which the ioctl command needs to be executed, which would generally be device files.
<**ioctl command**>	ioctl command which is implemented to achieve the desired functionality
<**arguments**>	The arguments need to be passed to the ioctl command.

Example

```
ioctl(fd, WRITE_VALUE, &number);
ioctl(fd, READ_VALUE, &value);
```

Sysfs

- Sysfs is a virtual filesystem exported by the kernel, similar to /proc. The files in Sysfs contain information about devices and drivers.
- Some files in Sysfs are even writable, for configuration and control of devices attached to the

system.

- Sysfs is always mounted on /sys.
- Sysfs is the commonly used method to export system information from the kernel space to the user space for specific devices.

Kernel object

heart of the sysfs model is the kernel object.

Defined in

```
include <linux/kobject.h>
```

A struct kobject represents a kernel object, maybe a device or so, such as the things that show up as a directory in the sysfs filesystem.

```
struct kobject {
 char *k_name;
 char name[KOBJ_NAME_LEN];
 struct kref kref;
 struct list_head entry;
 struct kobject *parent;
 struct kset *kset;
 struct kobj_type *ktype;
 struct dentry *dentry;
};
```

struct kobject

name	(Name of the kobject. Current kobject is created with this name in sysfs.)
parent	(This is kobject's parent. When we create a directory in sysfs for the current kobject, it will create under this parent directory)
ktype	(the type associated with a kobject)
kset	(a group of kobjects all of which are embedded in structures of the same type)
sd	(points to a sysfs_dirent structure that represents this kobject in sysfs.)
kref	(provides reference counting)

Create a directory in /sys

```
struct kobject * kobject_create_and_add ( const char * name, struct kobject * parent);
```

name – the name for the kobject

parent – the parent kobject of this kobject, if any.

If you pass kernel_kobj to the second argument, it will create the directory under **/sys/kernel/**. If you pass firmware_kobj to the second argument, it will create the directory under **/sys/firmware/**. If you pass fs_kobj to the second argument, it will create the directory under **/sys/fs/**. If you pass NULL to the second argument, it will create the directory under **/sys/**.

This function creates a kobject structure dynamically and registers it with sysfs. If the kobject was not able to be created, NULL will be returned.

Example

```
struct kobject *kobj_ref;
```

Creating a directory in /sys/kernel/

```
 kobj_ref = kobject_create_and_add("demo_sysfs",kernel_kobj);
```

I created the directory name as **demo_sysfs**.

Create Sysfs file

Include the header for creating the sysfs file.

```
#include <linux/sysfs.h>
```

Create attribute:

Kobj_attribute is defined as,

```
struct kobj_attribute {
 struct attribute attr;
 ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
 ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
size_t count);
};
```

attr the attribute representing the file to be created,

show the pointer to the function that will be called when the file is read in sysfs,

store the pointer to the function which will be called when the file is written in sysfs.

We can create an attribute using __ATTR macro.

```
__ATTR(name, permission, show_ptr, store_ptr);
```

Store and Show functions

```
ssize_t (*show)(struct kobject *kobj, struct kobj_attribute *attr, char *buf);
ssize_t (*store)(struct kobject *kobj, struct kobj_attribute *attr, const char *buf,
size_t count);
```

- The store function will be called whenever we are writing something to the sysfs attribute.
- The show function will be called whenever we are reading the sysfs attribute.

Create sysfs file:

```
int sysfs_create_file ( struct kobject *  kobj, const struct attribute * attr);
```

kobj – object we're creating for.

attr – attribute descriptor.

sysfs_remove_file

```
void sysfs_remove_file ( struct kobject *  kobj, const struct attribute * attr);
```

Example

```
static ssize_t sysfs_show(struct kobject *kobj, struct kobj_attribute *attr, char *
buf);

static ssize_t sysfs_store(struct kobject *kobj, struct kobj_attribute *attr, const
char *buf, size_t count);

struct kobj_attribute sys_attr = __ATTR(sys_value, 0660, sysfs_show, sysfs_store);

static ssize_t sysfs_show(struct kobject *kobj, struct kobj_attribute *attr, char *
buf)
{
        pr_info("sysfs read \n");
        return sprintf(buf, "%d",sys_value);
}

static ssize_t sysfs_store(struct kobject *kobj, struct kobj_attribute *attr, const
char *buf, size_t count){
        pr_info("sys write \n");
        sscanf(buf, "%d", &sys_value);
        return count;
```

```
}
```

call this kobject_put and sysfs_remove_file inside the exist function.

```
        kobject_put(kobj_ref);
        sysfs_remove_file(kernel_kobj, &sys_attr.attr);
```

Directory name **demo_sysfs**

```
root@192:/sys# ls
block     bus       class     dev       devices   firmware  fs        kernel    module    power
root@192:/sys# cd kernel/
root@192:/sys/kernel# ls -l
drwxr-xr-x    2 root     root            0 Apr 28 19:13 cgroup
drwxr-xr-x    3 root     root            0 Jan  1  1970 config
drwx------   34 root     root            0 Jan  1  1970 debug
drwxr-xr-x    2 root     root            0 Apr 28 19:13 demo_sysfs
-r--r--r--    1 root     root         4096 Apr 28 19:13 fscaps
drwxr-xr-x  223 root     root            0 Apr 28 19:13 irq
```

Read and write the data at the attribute **sys_values**

```
-r--r--r--    1 root     root          4096 Apr 28 19:
-r--r--r--    1 root     root          4096 Apr 28 19:
root@192:/sys/kernel# cd demo_sysfs/
root@192:/sys/kernel/demo_sysfs# ls
sys_value
root@192:/sys/kernel/demo_sysfs# cat sys_value
0root@192:/sys/kernel/demo_sysfs# echo 23 > sys_value
root@192:/sys/kernel/demo_sysfs# cat sys_value
23root@192:/sys/kernel/demo_sysfs#
```

GPIO

In embedded Linux systems, you can interface with GPIO (General Purpose Input/Output) pins using the sysfs interface. Sysfs is a virtual filesystem that provides access to various kernel parameters, including GPIO pins. Here are the steps to interface with GPIO pins using sysfs.

Determine the GPIO Pin Number

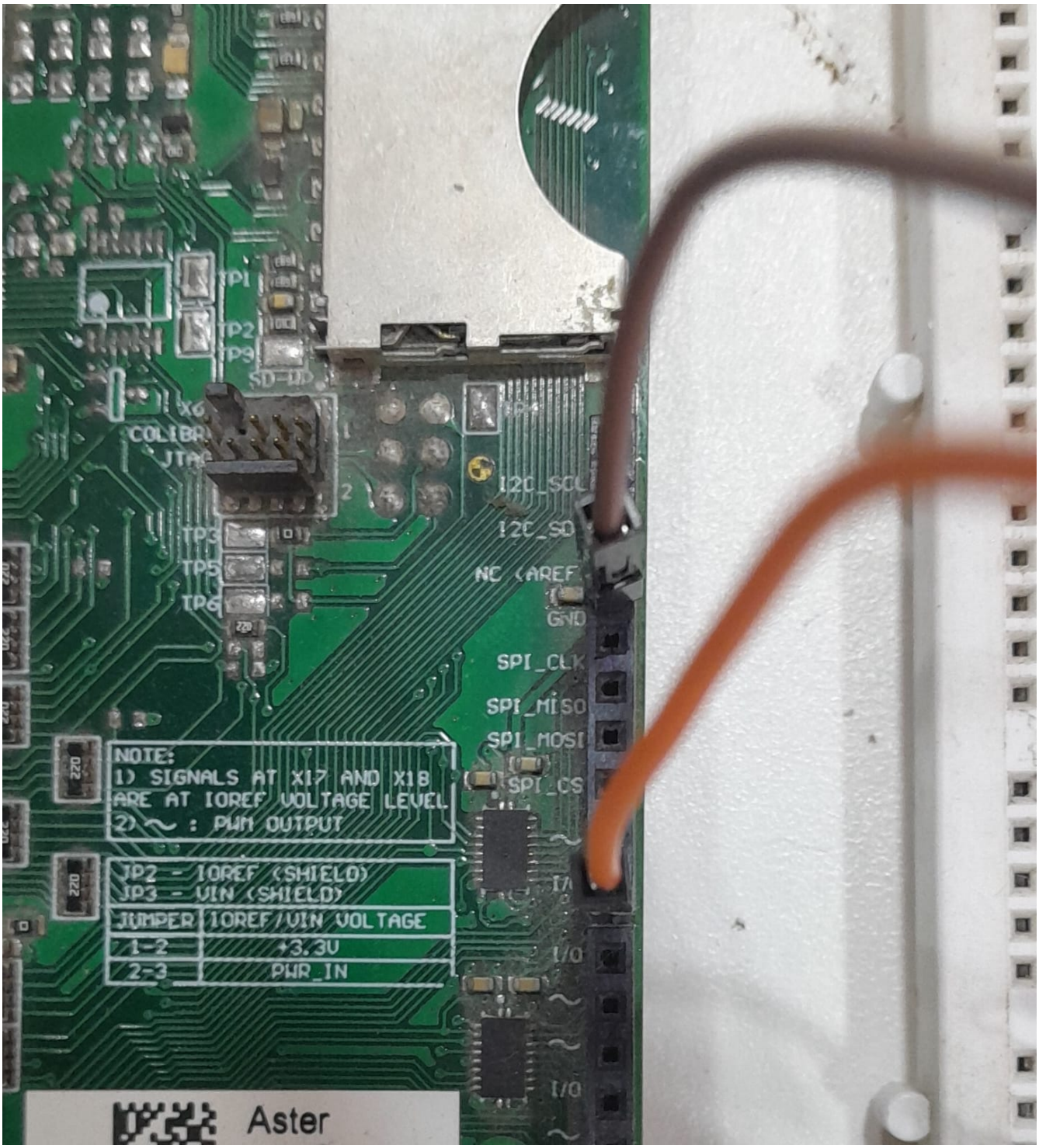
First, identify the GPIO pin number you want to work with. I am going to make a Digital I/O pins of X18 **UNO_GPIO7** of Aster v1.1 board as the input pin.

Datasheet for Aster v1.1

3.9.2.4. Arduino Shield header: Digital I/O Pins (X18)

Connector type: 1x8 Pin Header Female, 2.54 mm

Pin	Signal Name	SODIMM Number	I/O Type	Voltage	Pull-up/Pull-down
1	UNO_GPIO7	63 (via IC12)	I/O	VCC_IOREF	
2	UNO_GPIO8	59 (via IC12)	I/O	VCC_IOREF	
3	UNO_SPI_CS	86 (via IC12)	O	VCC_IOREF	
4	UNO_SPI_MOSI	92 (via IC12)	O	VCC_IOREF	
5	UNO_SPI_MISO	90 (via IC12)	I	VCC_IOREF	



An orange colour wire is connected with pin UNO_GPIO7 and connect the pin with push button using pullup configuration.

Convert GPIO Alphanumeric to GPIO Numeric Assignment

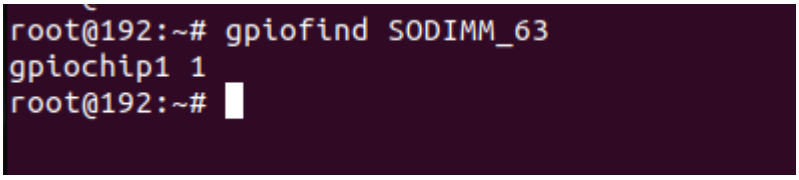
To get the gpio information using the command:

```
# gpioinfo
```

To find the GPIO name give the command.

```
# gpiofind SODIMM_63
```

In our case SODIMM number for X18 **UNO_GPIO7** is **SODIMM_63**



Then we need to convert GPIO Alphanumeric to GPIO Numeric Assisgnment using this formula.

```
32 * [controller] + gpio

32 * 1 + 1 = 33
```

Our gpio pin is **33**.

Export the GPIO Pin

Export the gpio pin using this command.

```
echo <pin_number> > /sys/class/gpio/export
```

Set the GPIO Pin Direction

You can configure the GPIO pin as an input or output by writing "in" or "out" to the direction file in the GPIO directory.

```
echo "out" > /sys/class/gpio/gpio<pin_number>/direction echo "in" >
```

/sys/class/gpio/gpio<pin_number>/direction

Read or Write to the GPIO Pin:

read command:

cat /sys/class/gpio/gpio<pin_number>/value

write command:

echo 1 > /sys/class/gpio/gpio<pin_number>/value

```
root@192:/sys/class/gpio# echo 33 > export
root@192:/sys/class/gpio# ls -l
--w-------    1 root     root           4096 Apr 28 20:00 export
lrwxrwxrwx    1 root     root              0 Apr 28 20:00 gpio33 -> ../../devices/pl
lrwxrwxrwx    1 root     root              0 Apr 28 17:42 gpiochip0 -> ../../devices
lrwxrwxrwx    1 root     root              0 Apr 28 17:42 gpiochip128 -> ../../devic
lrwxrwxrwx    1 root     root              0 Apr 28 17:42 gpiochip32 -> ../../device
lrwxrwxrwx    1 root     root              0 Apr 28 17:42 gpiochip64 -> ../../device
lrwxrwxrwx    1 root     root              0 Apr 28 17:42 gpiochip96 -> ../../device
--w-------    1 root     root           4096 Apr 28 17:42 unexport
```

When I press the button the value goes from 1 to 0.

```
root@192:/sys/class/gpio/gpio33# ls -l
-rw-r--r--    1 root     root          4096 Apr 28 20:03 active_low
lrwxrwxrwx    1 root     root             0 Apr 28 20:03 device -> ../../../gpiochip1
-rw-r--r--    1 root     root          4096 Apr 28 20:03 direction
-rw-r--r--    1 root     root          4096 Apr 28 20:03 edge
drwxr-xr-x    2 root     root             0 Apr 28 20:03 power
lrwxrwxrwx    1 root     root             0 Apr 28 20:03 subsystem -> ../../../../../../../../class/gpio
-rw-r--r--    1 root     root          4096 Apr 28 20:00 uevent
-rw-r--r--    1 root     root          4096 Apr 28 20:03 value
root@192:/sys/class/gpio/gpio33# cat direction
in
root@192:/sys/class/gpio/gpio33# cat value
1
root@192:/sys/class/gpio/gpio33# cat value
0
root@192:/sys/class/gpio/gpio33#
```

Interrupt

1. Interrupt handlers can not exchange data with the userspace.
2. Interrupt handlers can not be called repeatedly. When a handler is already executing, its corresponding IRQ must be disabled until the handler is done.
3. Interrupt handlers can be interrupted by higher authority handlers.

Functions Related to Interrupt

```
request_irq
(
unsigned int irq,
irq_handler_t handler,
unsigned long flags,
const char *name,
void *dev_id)
```

irq IRQ number to allocate.

handler This is Interrupt handler function.This function will be invoked whenever the operating system receives the interrupt.The data type of return is irq_handler_t, if its return value is IRQ_HANDLED, it indicates that the processing is completed successfully, but if the return value is IRQ_NONE, the processing fails.

flags can be either zero or a bit mask of one or more of the flags defined in linux/interrupt.h. The most important of these flags are: IRQF_DISABLED IRQF_SAMPLE_RANDOM IRQF_SHARED IRQF_TIMER

name Used to identify the device name using this IRQ, for example, cat / proc / interrupts will list the IRQ number and device name.

dev_id IRQ shared by many devices.

returns zero on success and nonzero value indicates an error.

```
free_irq(unsigned int irq,void *dev_id)
```

irq IRQ number.

dev_id is the last parameter of request_irq.

```
enable_irq(unsigned int irq)
```

Re-enable interrupt disabled by disable_irq or disable_irq_nosync

```
disable_irq(unsigned int irq)
```

Disable an IRQ from issuing an interrupt.

```
disable_irq_nosync(unsigned int irq)
```

Disable an IRQ from issuing an interrupt, but wait until there is an interrupt handler being executed.

```
in_irq()
```

returns true when in interrupt handler

```
in_interrupt()
```

returns true when in interrupt handler or bottom half

Interrupt Flags

IRQF_DISABLED	The IRQ line has been deliberately disabled by a device driver.
IRQF_SAMPLE_RANDOM	used to feed random generator
IRQF_TIMER	Marks this interrupt as timer interrupt
IRQF_SHARED	Allows sharing the irqs among other devices
IRQF_PERCPU	Interrupt is per cpu
IRQF_NOBALANCING	Excludes this interrupt from irq balacing
IRQF_IRQPOLL	used for polling
IRQF_ONESHOT	One time interrupt request. Disabled after the interrupt handler finished.

GPIO Interrupt

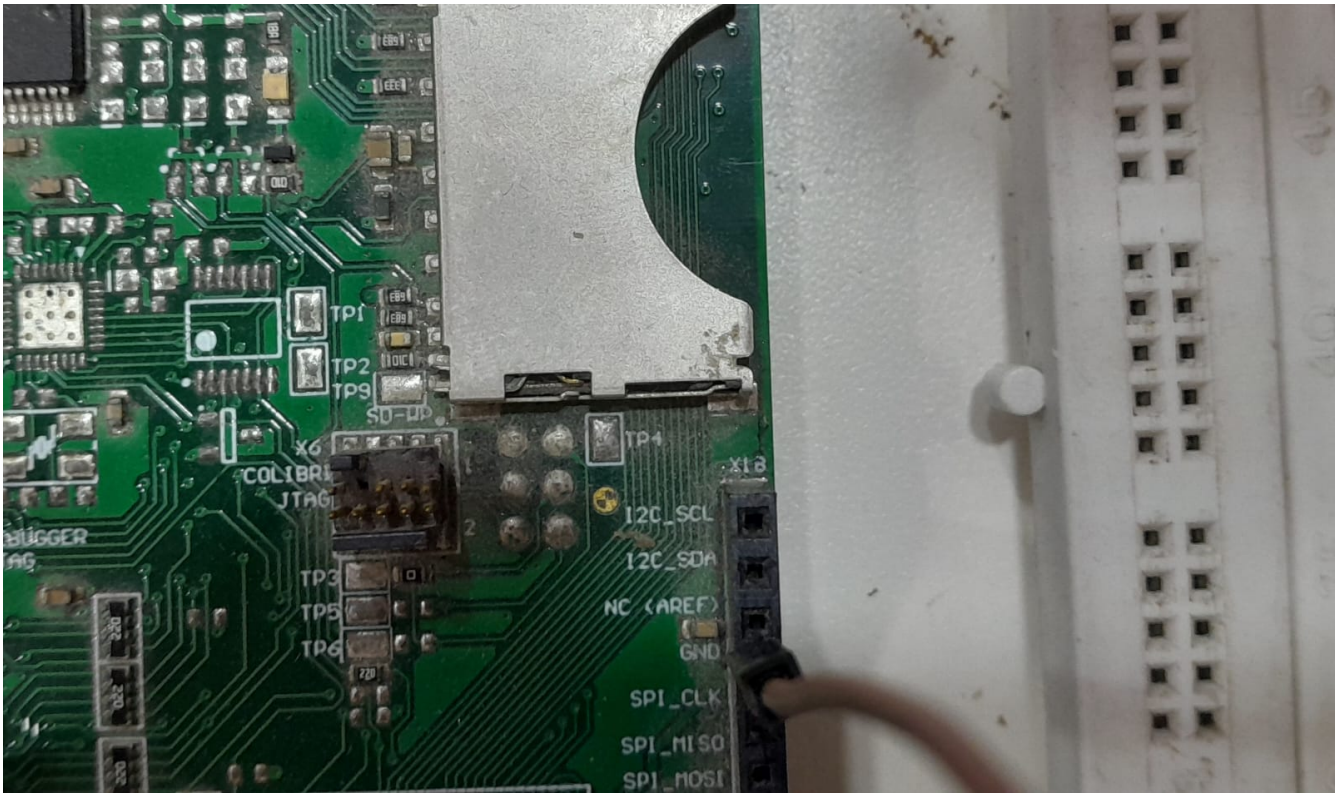
Make a Digital I/O pins of X18 **UNO_GPIO7** of Aster v1.1 board as the interrupt pin.

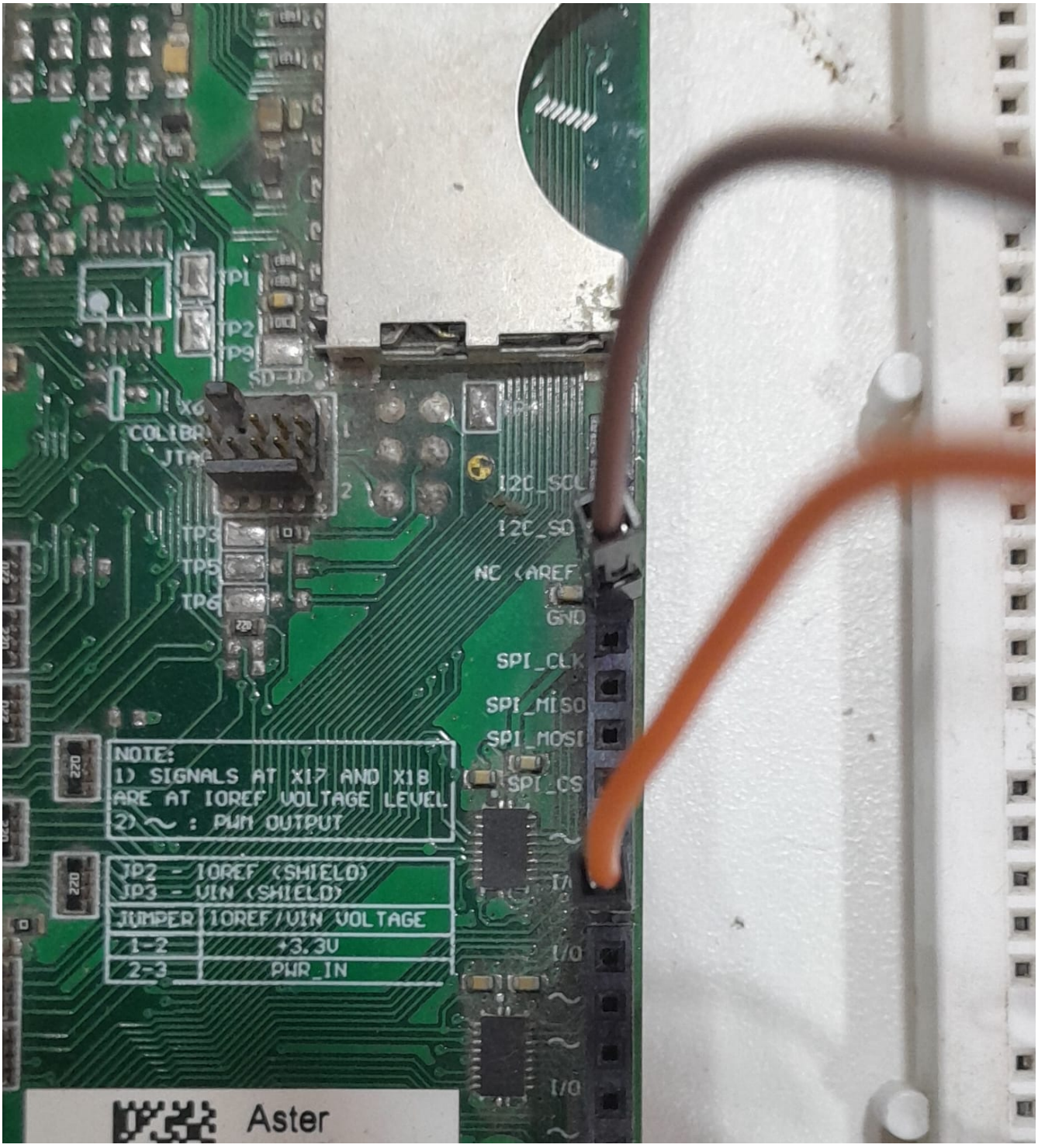
Datasheet for Aster v1.1

3.9.2.4. Arduino Shield header: Digital I/O Pins (X18)

Connector type: 1x8 Pin Header Female, 2.54 mm

Pin	Signal Name	SODIMM Number	I/O Type	Voltage	Pull-up/Pull-down
1	UNO_GPIO7	63 (via IC12)	I/O	VCC_IOREF	
2	UNO_GPIO8	59 (via IC12)	I/O	VCC_IOREF	
3	UNO_SPI_CS	86 (via IC12)	O	VCC_IOREF	
4	UNO_SPI_MOSI	92 (via IC12)	O	VCC_IOREF	
5	UNO_SPI_MISO	90 (via IC12)	I	VCC_IOREF	


TP1
TP2
TP9
SD-UP
X6
COLIBRI
JTAG
TP3
TP5
TP6
TP4
X18
I2C_SCL
I2C_SDA
NC (AREF)
GND
SPI_CLK
SPI_MISO
SPI_MOSI



An orange colour wire connected with the Digital I/O pins of X18 **UNO_GPIO7** of Aster v1.1 board.

Kernel GPIO subsystems have provided the APIs to access the GPIOs.

To include the header file for gpio access.

```
#include <linux/gpio.h>
```

Access the GPIO as Input

Step 1: Check GPIO is valid

```
bool gpio_is_valid(int gpio_number);
```

gpio_number : GPIO that you are planning to use

It returns **false** if it is not valid otherwise, it returns **true. Example**

```
        if(gpio_is_valid(33)==false){                   //checking the gpio is valid or
not
                pr_err("error : gpio is not valid \n");
                goto cleanup_device;
        }
```

Step 2: Request the GPIO

```
int gpio_request(unsigned gpio, const char *label)
```

gpio: GPIO that you are planning to use.

label: label used by the kernel for the GPIO in sysfs.

It returns **0** in success and a negative number in failure.

```
if(gpio_request(33,"GPIO 33 input") < 0)  //requesting the GPIO
        {
                pr_err("error : The gpio is not requested \n");
                goto cleanup_gpio;
        }
```

step 3: Set the direction of the GPIO

Set the GPIO as the INPUT:

```
int  gpio_direction_input(unsigned gpio)
```

gpio: GPIO that you want to set the direction as input.

Returns **0** on success, else an error.

set the GPIO direction as output.

```
int  gpio_direction_output(unsigned gpio, int value)
```

gpio: GPIO that you want to set the direction as output.

value: The value of the GPIO once the output direction is effective.

Returns **0** on success, else an error.

Example

```
 gpio_direction_input(33);   // set the gpio pin direction
```

Set GPIO Interrupt

Get IRQ number

you can get the IRQ number for the specific GPIO.

```
int gpio_to_irq(unsigned gpio);
```

gpio: GPIO that you want to get the IRQ number.

Interrupt Flags

1. IRQF_TRIGGER_RISING
2. IRQF_TRIGGER_FALLING
3. IRQF_TRIGGER_HIGH
4. IRQF_TRIGGER_LOW

Example

Now to make the **gpio33** as the interrupt pin the aster board.

Registering an Interrupt Handler

```
 if(request_irq(gpio_to_irq(33),(void *)irq_handler, IRQF_TRIGGER_FALLING,
"chardevice",NULL))
        {
                printk(KERN_INFO "cannot register IRQ \n");
                goto cleanup_irq;
        }
```

Interrupt Handler

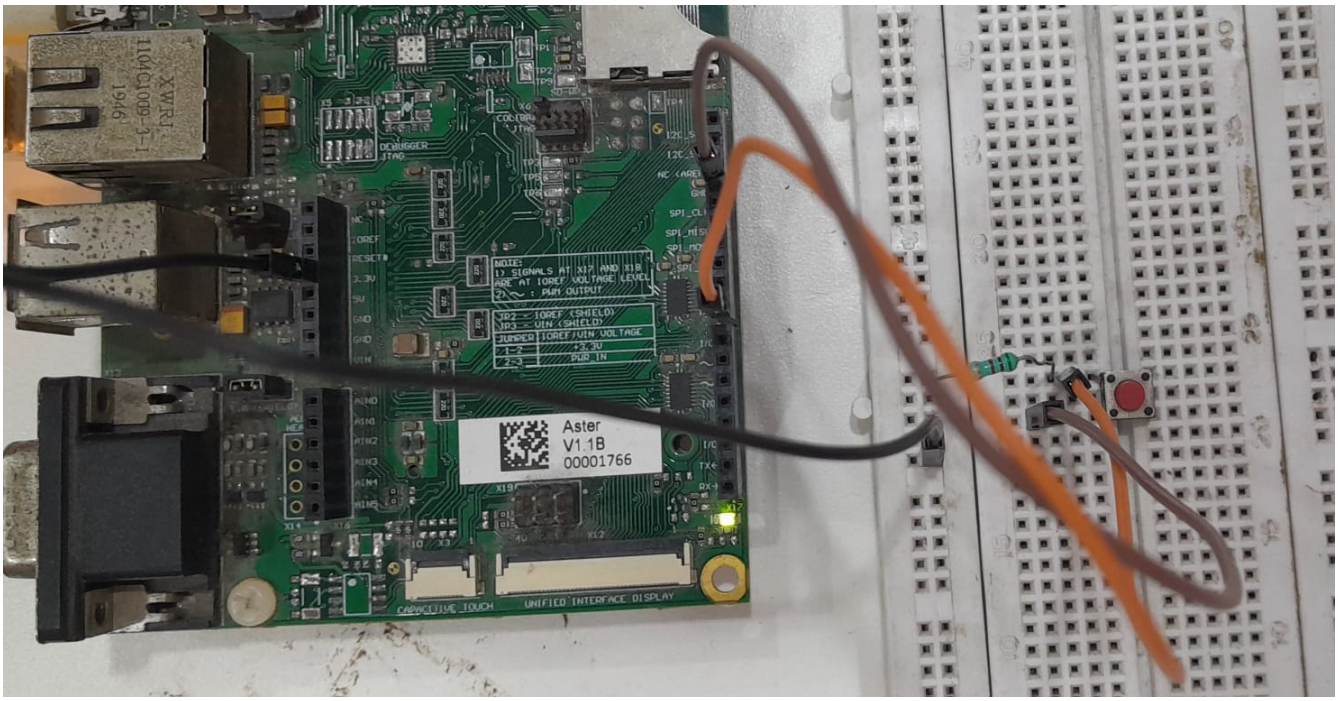
```
static irqreturn_t irq_handler(int irq, void *data)
{                                                       //Interrrupt handler
        printk(KERN_INFO "Interrupt Occurred");
```

```
        return IRQ_HANDLED;
}
```

Freeing an Interrupt Handler

```
gpio_free(33);
free_irq(gpio_to_irq(33),NULL);
```

When you press the button the pin goes form high to low transition an interrupt aquires.




```
[ 7796.301717] hardirqs last  enabled at (4131): [<c019945c>] __up_console_sem+0x50/0x60
[ 7796.309772] hardirqs last disabled at (4150): [<c0199448>] __up_console_sem+0x3c/0x60
[ 7796.317824] softirqs last  enabled at (4168): [<c01017d0>] __do_softirq+0x2c0/0x624
[ 7796.325658] softirqs last disabled at (4179): [<c012faa0>] __irq_exit_rcu+0x138/0x17c
[ 7796.333668] ---[ end trace 0000000000000000 ]---
[ 7798.036005] register for chardevice region is success
[ 7798.065277] Device driver insert has done
[ 7812.142728] Interrupt Occurred
[ 7817.600096] Interrupt Occurred
[ 7821.697876] Interrupt Occurred
root@192:/module_program/char#
```

Tasklet

- Executed in an atomic context.
- A tasklet only runs on the same core (CPU) that schedules it.
- Different tasklets can be running in parallel. But at the same time, a tasklet cannot be called concurrently with itself, as it runs on one CPU only.

Tasklet Structure

```
struct tasklet_struct
{
    struct tasklet_struct *next;
    unsigned long state;
    atomic_t count;
    void (*func)(unsigned long);
    unsigned long data;
};
```

next The next tasklet in line for scheduling.

state This state denotes Tasklet's State. TASKLET_STATE_SCHED (Scheduled) or TASKLET_STATE_RUN (Running).

count It holds a nonzero value if the tasklet is disabled and 0 if it is enabled.

func This is the main function of the tasklet. Pointer to the function that needs to schedule for execution at a later time.

data Data to be passed to the function "func".

The function used for Tasklet

DECLARE_TASKLET

```
DECLARE_TASKLET(name, func, data);
```

name name of the structure to be created.

func This is the main function of the tasklet.

data Data to be passed to the function "func".

DECLARE_TASKLET_DISABLED

```
DECLARE_TASKLET_DISABLED(name, func, data);
```

tasklet_enable

This used to enable the tasklet.

```
void tasklet_enable(struct tasklet_struct *t);
```

t - points to the tasklet structure

tasklet_disable

This used to disable the tasklet wait for the completion of the tasklet's operation.

```
void tasklet_disable(struct tasklet_struct *t);
```

tasklet_disable_nosync

This used to disable immediately.

```
void tasklet_disable_nosync(struct tasklet_struct *t);
```

tasklet_schedule

Schedule a tasklet with a normal priority.

```
void tasklet_schedule (struct tasklet_struct *t);
```

tasklet_hi_schedule

Schedule a tasklet with high priority.

```
void tasklet_hi_schedule (struct tasklet_struct *t);
```

tasklet_hi_schedule_first

```
void tasklet_hi_schedule_first(struct tasklet_struct *t);
```

tasklet_kill

This will wait for its completion and then kill it.

```
void tasklet_kill( struct tasklet_struct *t );
```

tasklet_kill_immediate

This is used only when a given CPU is in the dead state.

```
void tasklet_kill_immediate( struct tasklet_struct *t, unsigned int cpu );
```

Example

First I am going to declare the tasklet check the definition of the function **DECLARE_TASKLET** for your kernel version. To find the kernel version of board. **command**

```
# uname -r
```

To get the definition click the link below and select the kernel version then search for your identifier **DECLARE_TASKLET**

lxr Linux

In my case, kernel version is 6.1.0 definition of **DECLARE_TASKLET**

```
#define DECLARE_TASKLET(name, _callback)        \
struct tasklet_struct name = {              \
    .count = ATOMIC_INIT(0),            \
    .callback = _callback,              \
    .use_callback = true,               \
}
```

I declared that name as **tasklet** and than made the callback as **NULL** , also created the tasklet handler function as **tasklet_fun**

```
/*****************************************Tasklet*****************/
void tasklet_fun(unsigned long arg)
{
        printk(KERN_INFO "Executing Tasklet function %ld", arg);
}

DECLARE_TASKLET(tasklet,NULL);
```

Initialize the tasklet inside the init function.

```
tasklet_init(&tasklet,tasklet_fun,0);       // Initialize the tasklet function
```

Call the tasklet_kill function inside the exit function.

```
tasklet_kill(&tasklet);
```

schedule the tasklet inside the irq_handler.

```
static irqreturn_t irq_handler(int irq, void *data)
{
```

```
//Interrrupt handler
        printk(KERN_INFO "Interrupt Occurred");
      tasklet_schedule(&tasklet); //scheduling task to tasklet
        return IRQ_HANDLED;
}
```

when I press the button the interrupt acquires we schedule the tasklet to execute at the very convenient time.

Workqueue

1. workqueue allows users to create a kernel thread and bind work to the kernel thread.
2. This will run in the process context and the work queue can sleep.
3. Code deferred to a work queue has all the usual benefits of process context.
4. Most importantly, work queues are schedulable and can therefore sleep.

Workqueue function

creates a workqueue by the name and the function that we are passing in the second argument gets scheduled in the queue.

```
DECLARE_WORK(name, void (*func)(void *))
```

Schedule_work

```
int schedule_work( struct work_struct *work );
```

Scheduled_delayed_work

```
int scheduled_delayed_work( struct delayed_work *dwork, unsigned long delay );
```

Schedule_work_on

```
int schedule_work_on( int cpu, struct work_struct *work );
```

Scheduled_delayed_work_on

```
int scheduled_delayed_work_on(int cpu, struct delayed_work *dwork, unsigned long delay
);
```

Delete work from workqueue

```
int flush_work( struct work_struct *work );
void flush_scheduled_work( void );
```

Cancel Work from workqueue

```
int cancel_work_sync( struct work_struct *work );
int cancel_delayed_work_sync( struct delayed_work *dwork );
```

Check the workqueue

```
work_pending( work );
delayed_work_pending( work );
```

Example

Similar to the tasklet I am going to declare workqueue then create the workqueue handler function after that schedule the workqueue inside the irq_handler function.

```
/*******************workqueue*************************************/

void workqueue_fun(struct work_struct *work)
{
printk(KERN_INFO "Executing Workqueue function \n");
}

DECLARE_WORK(workqueue, workqueue_fun);

/************************************************************************/

static irqreturn_t irq_handler(int irq, void *data)
{                                                                    //Interrrupt
handler
        printk(KERN_INFO "Interrupt Occurred");
        schedule_work(&workqueue);  // scheduling task to workqueue.
        return IRQ_HANDLED;
}
```

when you press the button the interrupt acquires inside the interrupt handler function we scheduled the task to workqueue.

Atomic Operation

Atomic operations are often used when dealing with shared resources, such as variables, that multiple threads or processes can access concurrently. In a Linux device driver, atomic operations

are typically used to implement synchronization mechanisms, such as locks or semaphores, to protect critical sections of code.

Two different atomic variables are there

1. Atomic variables that operate on Integers
2. Atomic variables that operate on Individual Bits

Atomic Operation functions:

```
int atomic_read(atomic_t *v);
void atomic_set(atomic_t *v, int i);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc (atomic_t *v);
void atomic_dec (atomic_t *v);
void atomic_sub_and_test(int i, atomic_t *v);
void atomic_dec_and_test(atomic_t *v);
void atomic_inc_and_test(atomic_t *v);
void atomic_add_negative(int i, atomic_t *v);
void atomic_add_return(int i, atomic_t *v);
atomic_add_unless (atomic_t *v, int a, int u);
```

Atomic Bitwise operation functions

```
void set_bit(int nr, void *addr)
void clear_bit(int nr, void *addr)
void change_bit(int nr, void *addr)
int test_and_set_bit(int nr, void *addr)
int test_and_clear_bit(int nr, void *addr)
int test_and_change_bit(int nr, void *addr)
int test_bit(int nr, void *addr)
int find_first_zero_bit(unsigned long *addr, unsigned int size)
int find_first_bit(unsigned long *addr, unsigned int size)
```

Using the atomic operation going to check the file is already opened or not.

```
static int chrdev_open(struct inode *node, struct file *filp)
//open file
{
    printk(KERN_INFO "device file is opened \n");
    if(test_and_set_bit(0, &is_open))
    {
    printk(KERN_INFO "File is already opened \n");
    return -EBUSY;
    }
    return 0;
```

```
}
static int chrdev_release(struct inode *node, struct file *filp)
//release
{
        printk(KERN_INFO "device file is closed \n");
        clear_bit(0, &is_open);
        return 0;
}
```

Semaphore

In Linux device driver programming, semaphores are often used as synchronization mechanisms to control access to shared resources or critical sections of code among different parts of the kernel, including multiple processes or threads. Semaphores help prevent race conditions and ensure that only one process or thread can access a particular resource or execute a specific section of code at a time.

Including the Necessary Header Files:

```
#include <linux/semaphore.h>
```

you can declare semaphores using the struct semaphore data structure.

```
static struct semaphore my_semaphore;
```

Semaphore functions

```
void sema_init(struct semaphore *sem, int val);
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
void up(struct semaphore *sem);
void sema_exit(struct semaphore *sem);
```

Spinlock

1. Spinlock is a kernel synchronization service, or locking service used to avoid race condition while accessing shared resources.
2. only one thread execution can hold spinlock at any given point in time.
3. when a thread tries to acquire a spinlock which already held the busy loops or spins on the processor until the lock becomes available.
4. Since a thread busy loops on the processor, wasting processor execution cycles, it is not wise to hold the spinlock for a longer duration.

5. Spinlock are suitable for short critical sections.
6. Spinlocks comes in various versions, importantly IRQ and non-IRQ versions, and they protect shared resource by concurrent access by disabling kernel pre-emption and interrupts on the local processor.

Spinlock function:

```
spin_lock(spinlock_t *lock)
spin_trylock(spinlock_t *lock)
spin_unlock(spinlock_t *lock)
spin_is_locked(spinlock_t *lock)
spin_lock_bh(spinlock_t *lock)
spin_unlock_bh(spinlock_t *lock)
spin_lock_irq(spinlock_t *lock)
spin_unlock_irq(spinlock_t *lock)
spin_lock_irqsave( spinlock_t *lock, unsigned long flags )
spin_unlock_irqrestore( spinlock_t *lock, unsigned long flags )
```

header Include the header for spinlock.

```
#include <linux/spinlock.h>
```

Example In our case I am going to share the memory for two function

```
static char critData[64] = "Hello world\n";
```

To avoid the critical condition I am going to use spinlock.

```
/*************************************************************/

static irqreturn_t irq_handler(int irq, void *data)
{
        unsigned long flag;     //Interrrupt handler
        printk(KERN_INFO "Interrupt Occurred");
        spin_lock_irqsave(&lddlock,flag);
        critData[0] = 'h';
        spin_unlock_irqrestore(&lddlock,flag);

        return IRQ_HANDLED;
}

static ssize_t chrdev_read(struct file *filp, char __user *buf, size_t len, loff_t
*offset)  //read
{
        unsigned long flag;
```

```
        int ret = -1;
        printk(KERN_INFO "Read from device file \n");
        spin_lock_irqsave(&lddlock,flag);
        if(copy_to_user(buf,critData,12)==0){
                ret = 12;
        }
        spin_unlock_irqrestore(&lddlock, flag);
        return ret;
}
```

In this two functions when the interrupt is occurred the critData is accesed by the irq_handler during that time chrdev_read is trying to access the critData it can't because the memory is lock, untill unlock it can not access the data.

Device tree

- The open firmware Device Tree or simply Device tree (DT) is a data exchange format used for exchange hardware description data with the software or os.
- More specifically it is a decription of hardware that is readable by an operating system so that the operating system doesn't need to hard code details of the machine.
- In short, it is an new and recommended way to describe non-discoverable device to the linux kernel which was previously hardcoded into kernel source files.

Linux used Device tree for

- Platform identification
- Device population, The kernel parses the device tree data and generates the required software data structure, which will be used by the kernel code.

Device tree writing sysntax

- Node name
- Node Label
- Standard and non-standard property names
- Different data type representation (u32, byte, byte stream, string, stream of strings, Boolean, etc)

Refer this Link to learn about device tree (DeviceTree Specification Release v0.3)

device tree specifications

Platfrom drivers

platform driver is a type of device driver that is used to manage and control hardware devices that are integrated or closely associated with a specific hardware platform or system-on-chip (SoC). Platform drivers are a subset of device drivers and are designed to work with hardware

components that are tightly coupled with the underlying hardware architecture.

we can do that with the platform driver structure.

```
struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
    bool prevent_deferred_probe;
    bool driver_managed_dma;
};
```

Include the header file:

```
#include <linux/platform_device.h>
```

Sample platform driver

Define the platform driver structure

```
static struct platform_driver sample_platform_driver = {
        .driver ={
                .name = "ldd-sample",
                .of_match_table = of_match_ptr(sample_dt_ids),
        },
        .probe = sample_platform_driver_probe,
        .remove = sample_platform_driver_remove,
};
```

Function to match the driver with the device in the Device Tree

```
static const struct of_device_id sample_dt_ids[] =
{
        {.compatible = "ldd-sample"},
        {},
};
```

Function to probe the device

```
static int sample_platform_driver_probe(struct platform_device *pdev)
{
```

```
        printk(KERN_INFO "platform driver probe is called \n");

         return chardrv_init(pdev);
}

static int sample_platform_driver_remove(struct platform_device *pdev)
{
        printk(KERN_INFO "Platform driver remove is called \n");
        chardrv_exit(pdev);
        return 0;
}
```

Module initialization

```
module_platform_driver(sample_platform_driver);
```

Add the compatible in the device tree, goto the kernel source code and the path **arch/arm/boot/dts/** the device tree version I use imx6ull-colibri-wifi-eval-v3.dts

path - arch/arm/boot/dts/imx6ull-colibri-wifi-eval-v3.dts

inside this file add the compatible and add the pin control information for interrupt pin gpio33.

```
#include "imx6ull-colibri-wifi.dtsi"
#include "imx6ull-colibri-eval-v3.dtsi"

/ {
        model = "Toradex Colibri iMX6ULL 512MB on Colibri Evaluation Board V3";
        compatible = "toradex,colibri-imx6ull-wifi-eval", "fsl,imx6ull";

        ldd{
             compatible = "ldd-sample";
};

};

&iomuxc {
intr_key_gpio7: intr_gpio1_01 {
                fsl,pins = <
                        MX6UL_PAD_GPIO1_IO01__GPIO1_IO01        0x3000 /* SODIMM 6 */
                 >;
        };

};
```

- After adding the information again rebuild the kernel source and copy the **zImage** and **imx6ull-colibri-wifi-eval-v3.dtb** to the **/tftp**.
- Now reboot the board.

check this link to rebuild the kernel source code

rebuild

Now we shall insmod the platform driver .ko file, the compatible is matches with the device tree. After that probe function get called and then if we press the push button the interrupt is occurred accordingly.

```
[ 1739.002230] softirqs last disabled at (4593): [<c012faa0>] __irq_exit_rcu+0x138/0x1
[ 1739.010199] ---[ end trace 0000000000000000 ]---
[ 1739.331279] ldd-sample: probe of ldd failed with error -1
[ 1898.033506] platform driver probe is called
[ 1898.041155] register for chardevice region is success
[ 1898.063451] Device driver insert has done
[ 1972.189592] Interrupt Occurred
[ 1975.005354] Interrupt Occurred
[ 1977.371895] Interrupt Occurred
[ 1977.571643] Interrupt Occurred
[ 2124.153097] Platform driver remove is called
[ 2124.160904] The module removed successfully
root@192:/module_program/platform_driver#
```

Platform driver for backlight_device

Include the header file

```
#include <linux/backlight.h>
```

backlight operation structure might look like:

```
struct backlight_device {
    /* Function pointers for controlling the backlight */
    struct {
        int (*get_brightness)(struct backlight_device *bd);
        int (*set_brightness)(struct backlight_device *bd, int brightness);
        void (*update_status)(struct backlight_device *bd);
    } ops;

    /* Backlight properties and capabilities */
    struct backlight_properties props;

    /* Mutex for synchronization (optional) */
    struct mutex lock;
};
```

Backlight properties give the information about the brightness, max brightness, power etc...

Backlight operation structure.

```
static struct backlight_device *bldev;
```

```
static int bldev_get_brightness(struct backlight_device *bd)
{
        printk(KERN_INFO "get Brightness\n");
        return 0;
}

static int bldev_update_status(struct backlight_device *bd)
{
        printk(KERN_INFO "update status \n");
        return 0;
}

static int bldev_check_fb(struct backlight_device *bd, struct fb_info *fb)
{
        printk(KERN_INFO "check fb\n");
        return 0;
}

static struct backlight_ops blops =
{
         .get_brightness = bldev_get_brightness,
         .update_status = bldev_update_status,
         .check_fb = bldev_check_fb,
};

static struct backlight_properties blprops = {
        .max_brightness = 255,
        .type = BACKLIGHT_RAW,
};
```

Function to match the driver with the device in the Device Tree

```
static const struct of_device_id sample_dt_ids[] =
{
        {.compatible = "ldd-bklight"},
        {},
};

static struct platform_driver sample_platform_driver = {
        .driver ={
                .name = "ldd-backlight",
                .of_match_table = of_match_ptr(sample_dt_ids),
        },
        .probe = sample_platform_driver_probe,
        .remove = sample_platform_driver_remove,
};
```

Backlight device register and unregister call this function inside the probe function

```
static int sample_platform_driver_probe(struct platform_device *pdev)
{
        printk(KERN_INFO "platform driver probe is called \n");
         bldev = devm_backlight_device_register(&pdev->dev,"ldd-bklight",NULL,NULL,
&blops, &blprops);
         if(bldev == NULL){
                 printk(KERN_INFO "backlight device register is failed \n");
                 return -1;
         }
         return 0;
}

static int sample_platform_driver_remove(struct platform_device *pdev)
{
        printk(KERN_INFO "Platform driver remove is called \n");
        devm_backlight_device_unregister(&pdev->dev, bldev);
        return 0;
}
```

Add the compatible ldd-bklight

```
#include "imx6ull-colibri-wifi.dtsi"
#include "imx6ull-colibri-eval-v3.dtsi"

/ {
        model = "Toradex Colibri iMX6ULL 512MB on Colibri Evaluation Board V3";
        compatible = "toradex,colibri-imx6ull-wifi-eval", "fsl,imx6ull";

        ldd{
             compatible = "ldd-sample";
             pinctrl-name = "default";
             pinctrl-0 = <&intr_key_gpio7>;
             gpios = <&gpio1 1 GPIO_ACTIVE_LOW>;
};
        ldd-bklight {
             compatible = "ldd-bklight";
};

};

&iomuxc {
intr_key_gpio7: intr_gpio1_01 {
                fsl,pins = <
                        MX6UL_PAD_GPIO1_IO01__GPIO1_IO01        0x3000 /* SODIMM 6 */
                 >;
        };

};
```

After making changes in the device tree rebuild the kernel source code. And copy the .dtb file to the /tftp.

Then insmod the .ko file now the compatible is matches with the device tree and the probe function is called accordingly. The ldd-bakclight driver is created inside the /sys/class/backlight directory.

```
root@192:/module_program/backlight# cd /sys/class/backlight/
root@192:/sys/class/backlight# ls -l
lrwxrwxrwx    1 root     root             0 Apr 28 20:01 backlight -> ../../devices/platform/backlight/backlight/backlight
lrwxrwxrwx    1 root     root             0 Apr 28 20:03 ldd-bklight -> ../../devices/virtual/backlight/ldd-bklight
root@192:/sys/class/backlight# cd ldd-bklight
root@192:/sys/class/backlight/ldd-bklight# ls
actual_brightness  brightness         power              subsystem          uevent
bl_power           max_brightness     scale              type
root@192:/sys/class/backlight/ldd-bklight#
```

Now I am going set the brightness and also going to read the maximum brightness of the backlight.

```
root@192:/sys/class/backlight/ldd-bklight# cat max_brightness
255
root@192:/sys/class/backlight/ldd-bklight# echo 100 > max_brightness
-sh: max_brightness: Permission denied
root@192:/sys/class/backlight/ldd-bklight# echo 100 > brightness
root@192:/sys/class/backlight/ldd-bklight# cat brightness
100
root@192:/sys/class/backlight/ldd-bklight#
```