Sl No	Particulars	Page No

Device driver:-

* It is a piece of code that configures and manages a device.
* It knows how to configure the device, sending data to device and it knows how to process requests.
* When it is loaded into OS, it exposes interfaces to user-space so that user application can communicate with device.

Kernel's Job :-

→ To connect system calls to drivers system call implementation method.

→ In linux, we access the device by "file"

eg:- for RTC device, we need to read, write.

(ie) why device driver has to create "device file" interface at user's space, so that user

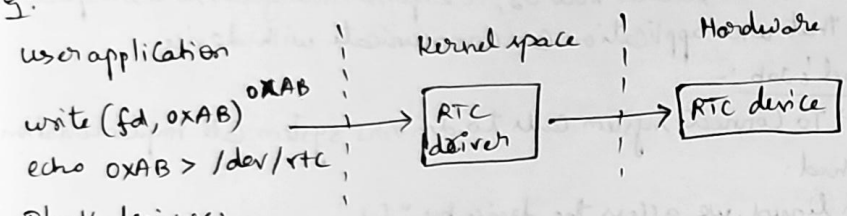
Kernel Job:- To connect system call execution from user space to drivers system call handler methods. This will be taken care by Kernel.

character driver (char driver):-

→ Character driver accesses data from device sequentially.
(ie) byte by byte not as chunk of data.

→ Eg: Sensors, RTC, Keyboard, serial port...

Eg:



Block drivers:-

→ Device which handles data in chunks (or) blocks is called block device.

→ These are complicated than char drivers because block drivers should implement advance buffering strategies.

Eg: Mass storage devices such as hard disks, Nand, flash, USB Camera, etc...

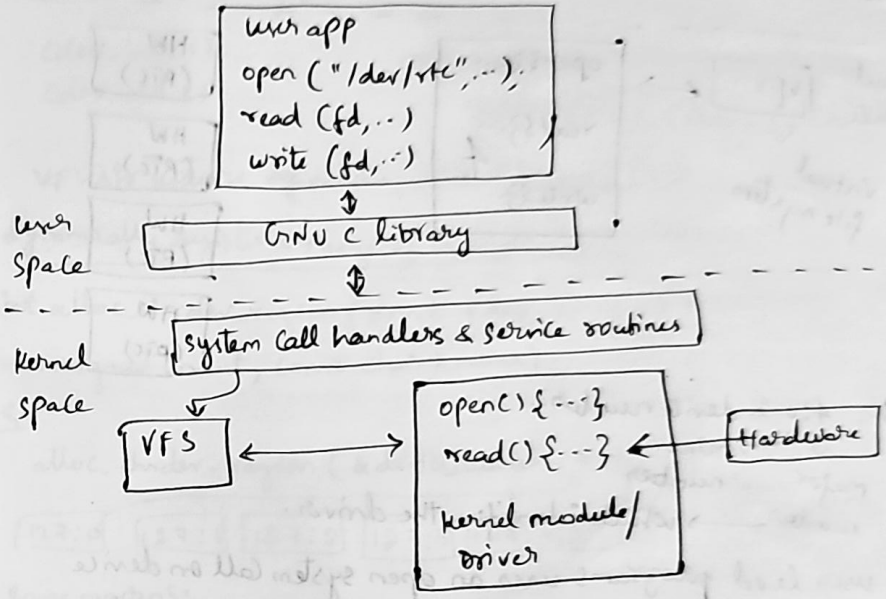
Device files:-

→ A device file is a special file (or) node which gets populated in /dev directory during kernel boot time (or) device/driver hot plug events.

→ By using this, user app & drivers communicate with each other.

→ Device files are managed as part of VFS subsystem.

Eg: $ cd /dev
$ ls -l



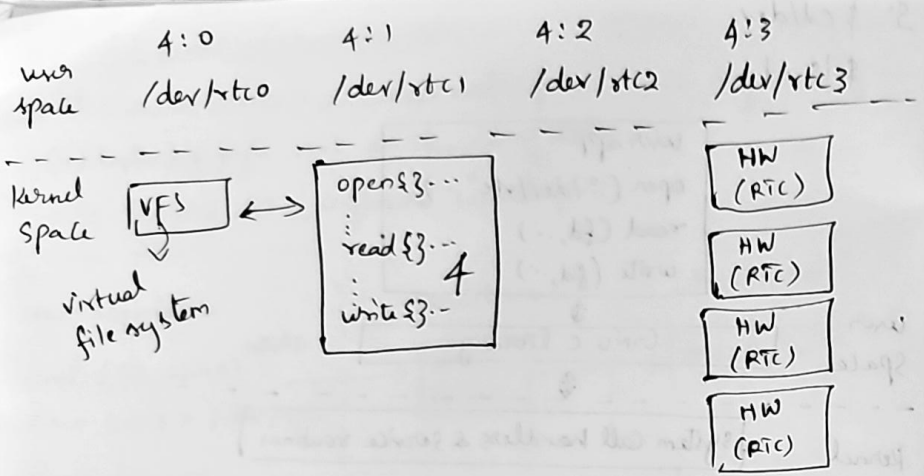
Device number:-

Let's say open ("/dev/rtc") user space
Kernel space

How Kernel connect the open system call to intended driver's to open method? How does the connection establish?

→ To establish the connection, Kernel uses "Device number".

Let's say device number 4:0



So, 4:0 is device number.
major number → which identifies the driver.
minor number

→ user level programs uses an open system call on device file. System call will first handled by VFS. VFS gets the device number & compares with driver's registration list. That means driver has to get registered with VFS using device. That what we call as CDEV_ADD.

```
$ cd /dev
$ ls -l
```

o/p:-

```
crw------- 1 root root (10, 144) Mar 16 09:52 nvram
crw-r----- 1 root kmem (1, 4)    Mar 16 09:52 port
```

(10, 144) → Device number
(1, 4) → Device_number.

Kernel Api's & utilities to be used in driver Code

alloc_chrdev_region(); → Create device number [linux/fs.h]

cdev_init() → Make char device registration with VFS. [/linux/cdev.h]
cdev_add()

VFS structure definitions → /linux/fs.h

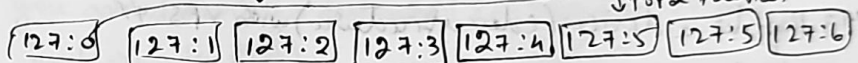
Dynamically register a range of char device numbers :-

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

Ex:

```
alloc_chrdev_region (&device_number, 0, 7, "eeprom");
```

↓ Total 7 devices.



Some macros:-

```
int minor_no = MINOR (device_number);
int major_no = MAJOR (device_number);
```

] linux/kdev_t.h

If we have major & minor number, turn them to "dev_t"

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

Eg: [As Same Kernel mode code]

```
dev_t device_number;
/* cdev variable */
/* file operations of the driver */
struct file_operations pcd_fops;

static int __init pcd_driver_init (void)
{
    /* 1. Dynamically allocate a device number */
    alloc_chrdev_region (&device_number, 0, 1, "pcd");

    /* 2. Initialize cdev structure with fops */
    cdev_init (&pcd_cdev, &pcd_fops);

    /* 3. Register a device (cdev structure) with VFS */
    pcd_cdev.owner = THIS_MODULE;
    cdev_add (&pcd_cdev, device_number, 1);
    return 0;
}
```

Summary

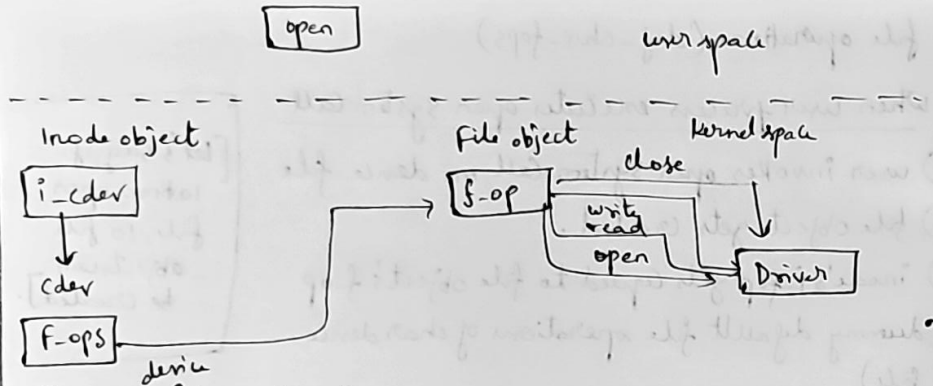
When device file gets created

1) create device file using udev
2) inode object gets created in memory and inode's i_rdev field is initialized with device number
3) inode object's i_fop field is set to dummy default file operations (def_chr_fops)

When user process executes open system call

[Let's say, if 10 times open file, 10 file objects will be created].

① user invokes open system call on device file
② file object gets created.
③ inode's i_fop gets copied to file object's f_op (dummy default file operations of char device file)
④ open function of dummy default file operation gets called (chrdev_open).
⑤ inode object's i_cdev field is initialized with cdev which you added during cdev_add (lookup happens using inode->i_rdev field)
⑥ inode->cdev->fops (this is read file operations of our driver) gets copied to file->f_op.
⑦ file->f_op->open method gets called (read open method of our driver)



Dynamic file creation :-

→ In linux, we can create device file dynamically. (ie) we need not manually create devices files under /dev directory.

→ user-level prgm such as udevd can populate /dev direct. with device files dynamically.

→ udev program listens to uevents generated by hotplug events (or) kernel modules. When udev receives uevents, it scans subdirectories of /sys/class looking for 'dev' files to create device files.

→ for each such 'dev' file, which represents combination of major and minor number for device, udev creates corresponding device file in /dev.

udev :-

→ so, udev relies on device info being exported to user space through sysfs.

→ uevents are generated when device driver takes the help of kernel APIs to trigger dynamic creation of device files (or) when hot pluggable device such as USB peripheral is plugged into system

Dynamic file creation (continues)

→ All device driver needs to do, for udev to work properly with it, is ensure any major & minor numbers assigned to device controlled by driver are exported to user space through sysfs.

→ Driver exports all info regarding device such as device filename, major, minor number to sysfs by calling function device_create

→ udev looks for file called 'dev' in /sys/class tree of sysfs, to determine what major & minor no. is assigned to a specific device.

class_create & device_create :-

class_create → create dir in sysfs : /sys/class/<class_name>

device_create :

↳ Creates Sub directory under /sys/class/<class_name> with our device name.

This fn also populates Sysfs entry with devfile which Consists of major & minor numbers ~~separated~~ separated by a : character.

Create & register class with Sysfs :-

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

owner: pointer to module that is to "own" this struct class

name: string for name of this class

Eg :-

```
Struct class * eeprom_class;
eeprom_class = class_create (THIS_MODULE, "eeprom_class");
```

populating sysfs :-

```
Struct device * device_create (struct class * class, Struct device
* parent, dev_t devt, void * drvdata, Const char *fmt, ...)
```

devt: dev_t for char device to be added

drvdata: data to be added for Callback.

Remove a device that was created with device_create() :-

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

class: pointer to struct class that this device was registered with

Destroy class structure :-

```
void class_destroy (struct class * class);
```

Removing cdev registration from Kernel VFS :-

```
Void cdev_del (Struct cdev *p)
```

Unregister range of device numbers :-

```
Void unregister_chrdev_region (dev_t from, unsigned Count);
```

count: no. of devices to unregister.

Example Code :-

```
#include <linux/Kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#define DEV_MEM_SIZE 512
char devbuf[DEV_MEM_SIZE];
dev_t device_num;
Struct cdev cdev1;
Struct class * pcd_class;
```

```
struct device * pcd_dev;

loff_t pcd_llseek (struct file *pf, loff_t offset, int whence)
{
 return 0;
}
ssize_t pcd_read (struct file *pf, char __user* devbuf,
 size_t count, loff_t *offp)
{
 return 0;
}
ssize_t pcd_write(struct file *pf, const char __user *devbuf,
 size_t count, loff_t *offp)
{
 return 0;
}
int pcd_open (struct inode * pcd_inode, struct file *pf)
{
 return 0;
}
int pcd_release (struct inode * pcd_inode, struct file *pf)
{
 return 0;
}
struct file_operations fops =
{
    .open = pcd_open,
    .read = pcd_read,
    .write = pcd_write,
    .llseek = pcd_llseek,
    .release = pcd_release,
    .owner = THIS_MODULE
};
__init static int pcd_init (void)
{
  pr_info("%s: Module initialization starts\n", __func__);
 /* Dynamically creating device number */
  alloc_chrdev_region (&device_num, 0, 1, "pcd-driver");
  pr_info ("%s: <major: %d> <minor: %d> \n", __func__,
  MAJOR (device_num), MINOR (device_num));
 /* Initialize cdev */
  cdev_init (&cdev1, &fops);
  cdev1.owner = THIS_MODULE;
 /* Register with VFS */
  cdev_add (&cdev1, device_num, 1);
 /* Create class & register with sysfs */
  pcd_class = class_create (THIS_MODULE, "pcd-class");
 /* populate sysfs */
  pcd_dev = device_create (pcd_class, NULL, device_num,
  NULL, "pcd-device");
  pr_info ("%s: Module initialization is done \n", __func__);
  return 0; }
```

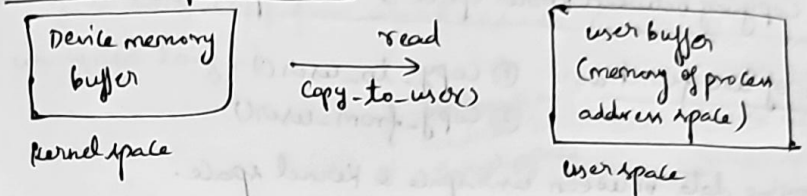
```
-exit static void pcd_exit (void)
{
    pr_info ("%s: Module cleanup start\n", __func__);
    device_destroy (pcd_class, device_num);
    class_destroy (pcd_class);
    cdev_del (&cdev1);
    unregister_chrdev_region (device_num, 1);
    pr_info ("%s: Module is unloaded \n", __func__);
}
module_init (pcd_init);
module_exit (pcd_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("pseudo driver");
MODULE_AUTHOR ("VIGNESH");
```

→ load module
→ check /sys/class
  → pcd_class folder is created
→ check /sys/class/pcd_class
  → pcd_device folder is created

```
$ ls
dev power subsystem uevent.

$ cd /dev/
pcd_device → will be there
```

Read method implementation :-

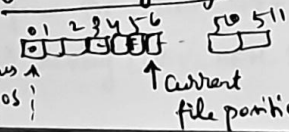



```
ssize_t pcd_read (struct file *fp, char __user *buff, size_t count,
loff_t *fpos) { return 0;
}
```

① Check the user requested 'count' value against DEV_MEM_SIZE of the device.
  * if f_pos (current_file_pos) + count > DEV_MEM_SIZE
  * Adjust 'count' count = DEV_MEM_SIZE - f_pos
② Copy 'count' no. of bytes from device memory to user buffer.
③ Update f_pos.
④ Return no. of bytes succesfully read (or) error code.
⑤ If f_pos at EOF, then return 0;

Ex:- count ⇒ 513, but buff size is 512, current_file_pos = 0,
so, count will be modified ⇒ 0 + 513 > 512
↳ yes → count = 512 - 0
C = 512

After read of 6 bytes



Data Copying between Kernel space & user space :-

Role of two functions:- ① copy_to_user()
② copy_from_user()

→ Copying data between user space & Kernel space.
→ Check whether user space pointer is valid (or) not.
→ If pointer is invalid, no copy is performed.
→ If an invalid address is encountered during the copy, only part of data is copied. In both cases, return value is the amount of memory still to be copied.

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

to: Destination address in user space
from: Source address in Kernel space

→ Returns '0' on success (or) no. of bytes that could not be copied.
→ If this function returns non zero value, you should assume that there was problem during data copy. So return appropriate error code

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

to: Destination address in Kernel space
from: Source address in user space.

llseek method:-

```
loff_t pcd_llseek (struct file *filp, loff_t off, int whence)
{
return 0;
}
if whence = SEEK_SET
filp -> f_pos = off
if whence = SEEK_CUR
filp -> f_pos = filp -> f_pos + off
if whence = SEEK_END
filp -> f_pos = DEV_MEM_SIZE + off   <- Moving beyond this may cause problem
```

pseudo driver:-

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#define DEV_MEM_SIZE 512
char devbuf [DEV_MEM_SIZE];
```

```
dev_t device_num;
struct cdev cdev1;
struct class * pcd_class;
struct device * pcd_dev;

loff_t pcd_llseek (struct file * pf, loff_t offset, int whence)
{
    loff_t temp;
    switch (whence)
    {
      Case SEEK_SET:
      if ((offset > DEV_MEM_SIZE) || (offset < 0))
      return -EINVAL;
      pf->f_pos = offset;
      break;
      Case SEEK_CUR:
      temp = pf->f_pos + offset;
      if ( (temp > DEV_MEM_SIZE) || (temp < 0))
      return -EINVAL;
      pf->f_pos = temp;
      break;
      Case SEEK_END:
      temp = offset + DEV_MEM_SIZE;
      if ((temp > DEV_MEM_SIZE) || (offset < 0))
      return -EINVAL;
      pf->f_pos = temp;
      break;
    default:
        return -EINVAL;
    }
    return pf->f_pos;
}
ssize_t pcd_read (struct file * pf, char __user *buf, size_t
count, loff_t * offp)
{
pr_info ("%s : User requested for %zu bytes to read\n",
  __func__, count);
pr_info ("%s : Before reading - Current file position addres
 : %lld\n", __func__, *offp);
 /* Adjust the Count */
if ((*offp + count) > DEV_MEM_SIZE)
count = (DEV_MEM_SIZE - *offp);
 /* Copy to user */
if (copy_to_user (buf, (const void *) &devbuf [*offp], count))
return ~~-EINVAL~~ -EFAULT;
/* Update the file position */
*offp += count;
```

```
pr_info("%s: After reading - Current file position address
 : %lld \n", __func__, *offp);
pr_info("%s: %zu bytes read from device sucessfully \n",
 __func__, Count);
 return Count;
}
ssize_t pcd_write (struct file *pf, const char __user *buf,
 size_t Count, loff_t * offp)
{
pr_info("%s: user requested for %zu bytes to write \n",
 __func__, Count);
pr_info("%s: Before writing - Current file position address: %lld \n",
 __func__, * offp);
 /* Adjust the Count */
If ((*offp + Count) > DEV_MEM_SIZE)
 Count = (DEV_MEM_SIZE - *offp);
 /* Copy to user */
 if (copy_from_user (&devbuf [*offp], buf, Count))
 return -EFAULT;
 If (!Count)
 return -ENOMEM;
 /* update the file position */
 *offp += Count;
 pr_info("%s: After writing - Current file position address
 : %lld \n", __func__, *offp);
pr_info("%s: %zu bytes are written to device successfully \n",
 __func__, Count);
 return Count;
}
int pcd_open (struct inode *pcd_inode, struct file *pf)
{
 pr_info ("%s: pcd device is opened \n", __func__);
 return 0;
}
int pcd_release (struct inode * pcd_inode, struct file *pf)
{
 pr_info ("%s: pcd device is released \n", __func__);
 return 0;
}
struct file_operations fops
 =
 {
 .open = pcd_open,
```

```
    .read = pcd_read,
    .write = pcd_write,
    .llseek = pcd_llseek,
    .release = pcd_release,
    .owner = THIS_MODULE
};

__init static int pcd_init (void)
{
    pr_info ("%s : Module initialization starts \n", __func__);
    /* Dynamically creating device number */
    alloc_chrdev_region (&device_num, 0, 1, "pcd_driver");
pr_info("%s: <major:%d><minor:%d>\n", __func__,
MAJOR(device_num), MINOR(device_num));
    /* Initialize cdev */
    cdev_init(&cdev1, &fops);
    cdev1.owner = THIS_MODULE;
    /* Register cdev with VFS */
    cdev_add(&cdev1, device_num, 1);
    /* Create class & register with sysfs */
    pcd_class = class_create (THIS_MODULE, "pcd_class");
    /* populate sysfs */
    pcd_dev = device_create (pcd_class, NULL, device_num,
    NULL, "pcd-device");
    pr_info ("%s : Module initialization is done \n", __func__);
    return 0;
}
__exit static void pcd_exit(void)
{ pr_info("%s: Module cleanup starts\n", __func__);
    device_destroy (pcd_class, device_num);
    class_destroy (pcd_class);
    cdev_del (&cdev1);
    unregister_chrdev_region (device_num, 1);
    pr_info ("%s : Module is unloaded \n", __func__);
}
module_init(pcd_init);
module_exit (pcd_exit);
MODULE_LICENSE ("GPL");
MODULE_DESCRIPTION ("pseudodriver");
```

Platform devices & drivers

Bus:-

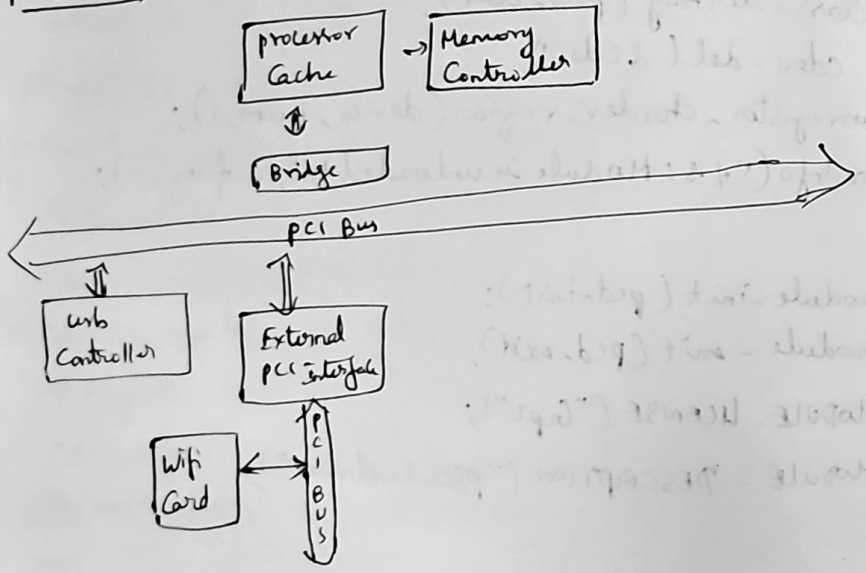
→ It is Collection of electrical wirings which transfers information between devices.

Platform bus:- → Term used by linux device model to represent all non-discoverable buses of an embedded platform.

→ It is pseudo bus (or) Linux's virtual bus. It doesn't have any physical existence.

→ representing bus interfaces that do not have auto discoverable & hot plugging capability.

PC Scenario:-



→ If we consider pc scenario, there is central bus system which is always based on PCI (or) PCI express bus. It is known bus to OS.

→ Devices which are connected over PCI bus interface are actually auto discoverable & bus support hot plugging of devices

→ OS learns about the connected peripherals automatically. This is what we call device discovery. we need not to feed the information about devices manually.

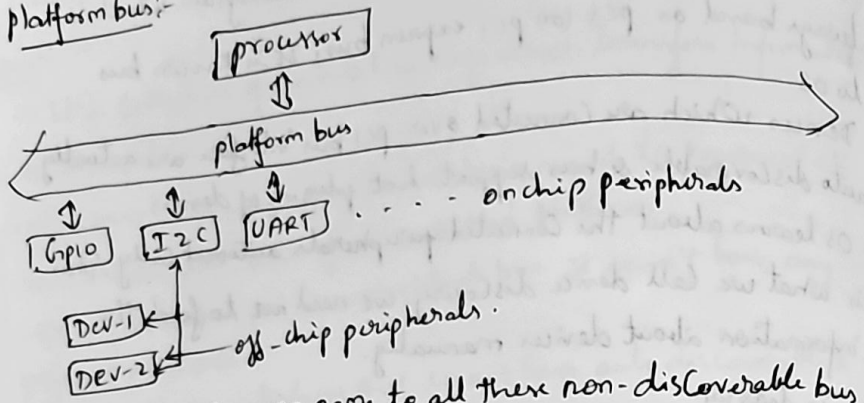
Eg: In diagram.

Wifi Cards host bus is PCI bus. And PCI has inbuilt capability to discover wifi Card automatically by accessing Configuration registers. we need not to write any piece of Code to add Wifi Card details to Kernel manually. Everything happens at hardware level. so when Kernel detects wifi Card. for example Kernel automatically loads driver which is incharge of that wifi Card. And driver configures it & ready to use.

→ But if we consider Embedded platform, bus interfaces on SoC is Purely vendor specific. There is no Central bus to discover all on chip as well as off chip devices. so they neither support hot plugging nor the auto discovery of connected devices.

→ Since everything is permanent embedded. so it doesn't make any sense to implement PCI inside embedded system.

platform bus:-



→ Linux gives common name to all these non-discoverable bus and that name is platform Bus. so from linux point of view, all these devices which are non discoverable are connected to processor via platform bus.

→ Linux doesn't care whether it is I2C (or) AHB

→ yes, crt. from HW point of view, it is just platform they are connected to different bus interfaces. But from linux point of view, it is just platform bus.

→ Devices which are connected to platform bus are called as platform devices (or) non discoverable devices.

Device Discovery:-

→ Every device has its configuration data & resources which need to be reported to OS which is running on Computer system.

→ An OS such as windows/linux running on Computer can auto discover these data. Thus OS learn about the connected device automatically (Device enumeration).

→ Enumeration is a process through which OS can inquire & receive information such as type of device, manufacturer, device configuration.

→ Once OS gathers information about device, it can autoload the appropriate driver for the device. In pc scenario, buses like pci & USB support auto enumeration/hot plugging of devices.

→ However on embedded system, this may not be the case since most peripherals are connected to cpu over buses that don't support auto discovery. We call them as platform devices.

→ All these platforms devices which are non discoverable by nature, but they must be part of linux device model, so the info about the devices must be fed to linux kernel either at compile time / boot time.

Adding platform devices information to kernel

① → During Compilation of Kernel :-

→ static method

→ HW details are part of kernel files (board file, drivers)

→ Deprecated and not recommended.

② Loading dynamically
→ As a Kernel module
→ Not recommended.

③ During Kernel boot time
→ Device tree blob
→ Latest & recommended.

Old-one Board file approach :-
→ This method was used before Kernel version 3.7 to add HW Configuration details to the Kernel.
→ Details about onboard peripherals.
→ pin configuration details
→ you have to recompile kernel if any device property changes.
→ All information about hardware configuration is hardcoded in the Kernel source files & board files.

Device tree method :-
→ DT was originally created by open firmare as part of Communication method for passing data from open firmware to client pgm.
→ An OS used Device Tree to discover topology of HW at runtime & thereby support a majority of available HW without Hardcoded information (assuming drivers were available for all devices)

Platform devices :-
→ Devices which are connected to platform bus are called platform devices.
→ A device if its parent bus doesn't support enumeration of connected devices then it becomes platform device.

Platform driver :-
→ A driver who is in charge of handling platform device is called platform driver.
→ Writing an I2C-driver, it means writing I2C driver for device, which is connected to I2C controller over I2C bus.

Registering a platform driver :-
#define platform_driver_register(drv) ...
↓
#include/linux/platform_device.h
→ platform driver structure [platform_driver] ⇒ structure name

Registering platform device :-
int platform_device_register (struct platform_device *pdev);
↓
deprecated

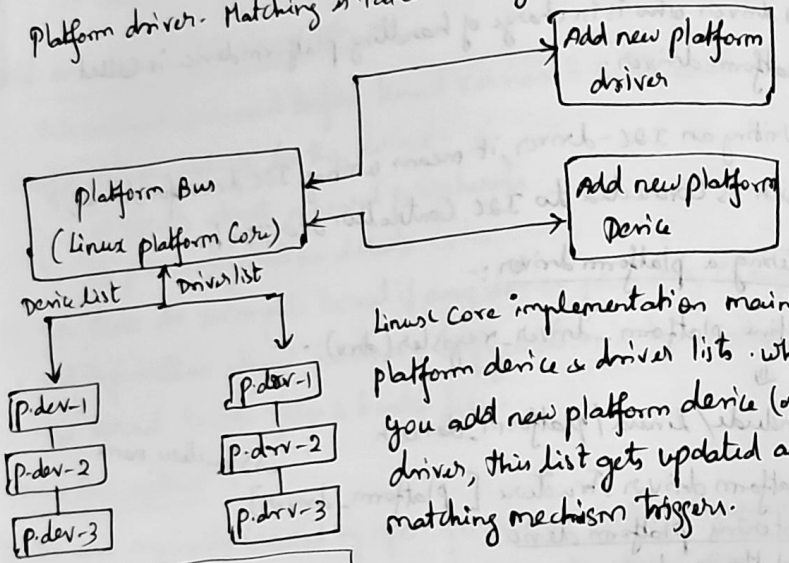
structure ⇒ platform_device

Platform device - driver matching :-

How do we make correct driver gets autoloaded whenever we add new platform device?

The answer is due to "matching" mechanism of bus core.

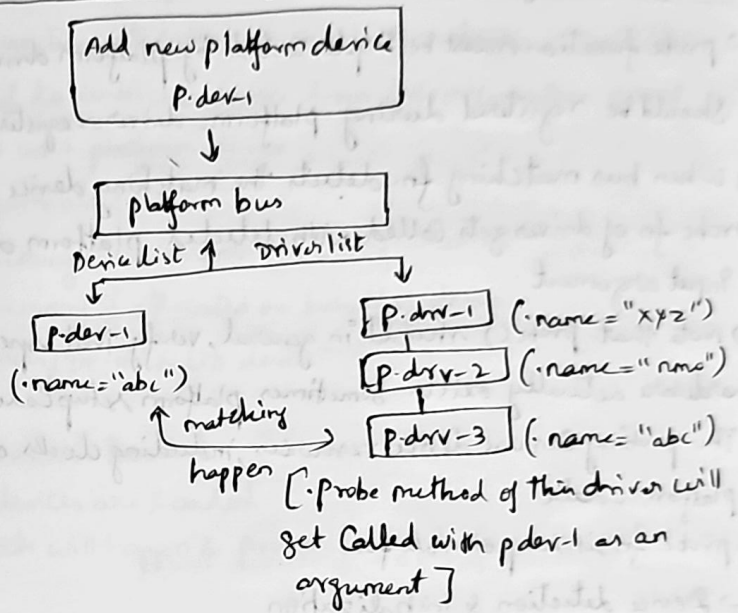
→ Match should be established b/n platform device & its corresponding platform driver. Matching is taken care by Linux kernel itself.



Linux Core implementation maintains platform device & driver lists. Whenever you add new platform device (or) driver, this list gets updated and matching mechanism triggers.

Every bus type has its match function, where device & driver list will be scanned.

A Sample "name" based matching



→ Whenever a new device (or) new driver is added, the matching function of the platform bus runs, and if it finds a matching platform device for platform driver, the probe fn. of matched driver will get called. Inside probe fn, driver configures the detected device.

→ Details of the matched platform device will be passed to the probe fn. of matched driver so that driver can extract the platform data & configure it.

Probe function of platform driver

→ probe function must be implemented by platform driver and should be registered during platform_driver_register().

→ When bus matching fn. detects the matching device & driver, probe fn of driver gets called with detected platform device as an input argument.

→ Note that probe() should in general, verify that specified device hardware actually exists. Sometimes platform setup code can't be sure. The probing can use device resources, including clocks and device platform_data.

→ probe fn is responsible for
- → Device detection & initialization
- → Allocation of memories for various data structures.
- → Mapping i/o memory
- → Registering interrupt handlers
- → Registering device to kernel framework, user level access point.

→ probe may return 0 (success) (or) error code. If probe function returns non-zero value meaning probing of device has failed.

Remove function of platform driver

→ Remove function gets called when platform device is removed from kernel to unbind a device from driver (or) when kernel no longer uses platform device.

→ It is responsible for
- * unregistering device from kernel framework
- * Free memory if allocated on behalf of device
- * Shutdown/De-initialize device

Example:-

① platform driver is loaded.

② platform devices are loaded

③ Now match will happen & probe fn. of platform driver will be executed.

Platform device setup.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/device.h>
#include <linux/platform_device.h>
#include "platform.h"

struct platform_data_for_device data_for_dev1 =

{
    .serial_no = 542,
    .market = "ECE"
};
```

```
struct platform_data data_for_device data_for_dev2 =
{
    .serial=123,
    .market="JPN"
};
Void pcd_dev_release(struct device *dev)
{
    pr_info("%s: PCD device is released\n", __func__);
}
struct platform_device pdev1 =
{
    .name="pcd_driver",
    .id=0,
    .dev=
    {
        .platform_data=&data_for_dev1,
        .release = pcd_dev_release
    }
};
struct platform_device pdev2 =
{
    .name="pcd_driver",
    .id=1,
    .dev=
    {
        .platform_data=&data_for_dev2,
        .release = pcd_dev_release
    }
};
```

```
static __init int pcd_device_start(void)
{
    pr_info("%s: pcd device setup module is loaded succesfully!\n",
    __func__);
    platform_device_register(&pdev1);
    platform_device_register(&pdev2);
    return 0;
}
static __exit void pcd_device_end(void)
{
    pr_info("%s: pcd device setup module is unloaded succesfully!\n",
    __func__);
    platform_device_unregister(&pdev1);
    platform_device_unregister(&pdev2);
}
module_init(pcd_device_start);
module_exit(pcd_device_end);
MODULE_LICENSE("GPL");
```

platform.h

```
struct platform_data_for_device
{
    int serial_no;
    char *market };
```

platform-driver.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/platform_device.h>
#include <fs.h>
int pcd_probe(struct platform_device *pdrv)
{
  pr_info("%s: pcd device is detected\n", __func__);
  return 0;
}
int pcd_remove (struct platform_device *pdrv)
{
  pr_info("%s: pcd device is removed\n", __func__);
  return 0;
}
struct platform_driver pcdrv =
{
    .probe = pcd_probe,
    .remove = pcd_remove,
    .driver = {.name = "pcd_driver"}
};
static __init int pcdrv_starts(void)
{
  pr_info("%s: pcd_driver is loaded succesfully\n", __func__);
  platform_driver_register(&pcdrv);
  return 0; }
static __exit void pcdrv_end(void)
{
  pr_info("%s: pcd driver is unloaded succesfully\n", __func__);
  platform_driver_unregister(&pcdrv);
}
```

Resource managed functions:- [only for platform device]

Kmalloc:- → programmer must free the memory using free();

devm_Kmalloc:-

→ programmers using Kfree() is not required. Kernel will take care of freeing memory when "device" (or) managing "driver" gets removed from the system.

Examples:-

Kmalloc() } ⟶ devm_Kmalloc()
Kfree()

gpiod_get() } ⟶ devm_gpiod_get()
gpiod_put()

Refer to bootlin search this apis. otherwise /Documentation/driver-model/devres.txt

If we use this as resource managed APIs, then we can simplify our probe & remove fns.

Probe function:

```
int probe_function (struct platform_device *pdev)
{
  struct dev_data *data;
  data = kmalloc(5);
  if (!data)
    return -ENOMEM;
  ret = do_some_things();
  if(ret < 0)
    go to free_mem;
return 0;
}
free_mem:
  kfree(data);
  return ret;
}
```

↓ modified to

```
int probe_function (struct platform_device *pdev)
{
  struct devdata *pdata;
  data = devm_kmalloc(&pdev->dev, );
  if (!data)
    return -ENOMEM;
  ret = do_some_things();
  if (ret < 0)
      return -EINVAL;
return 0;
}
```

In remove function,

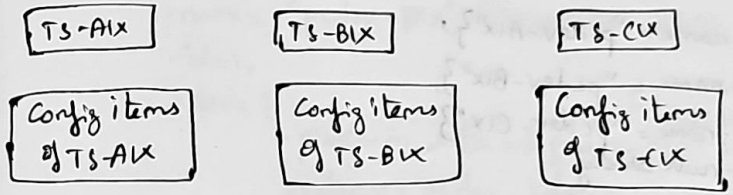
kfree → of buffer, dev_data can be removed.

It is always better to use resource managed Kernel functions.
Because most of time programmers forget to free memories.
This will lead to memory leak.

platform driver matching using platform ids :-

UseCase: Different versions of a chip.

Different version of temperature sensing chip



→ Vendor won't provide kfree drivers to handle/Configure three versions of chip.
→ only single driver will be provided.
→ only our platform driver must support different device ids to compare.

Explore platform_driver structure where we can see structure

platform_device_id → id matching.

We need to include <linux/mod_devicetable.h>

```
struct platform_device_id
{
    char name[PLATFORM_NAME_SIZE];
    kernel_ulong_t driver_data;
};
```

Example - using previous example :-

In platform driver file,

```
struct platform_device_id pcdevs_ids[] =
{
    [0] = { .name = "pcdev-A1x" },
    [1] = { .name = "pcdev-B1x" },
    [2] = { .name = "pcdev-C1x" }
};   { } <- null array

struct platform_driver pcd_platform_driver =
{
    .probe = pcd_platform_driver_probe,
    .id_table = pcdevs_ids,
    .remove = pcd_platform_driver_remove,
    .driver = {
                .name = "pseudo_char_device"
              }
};
```

In platform device file, [Go & Refer old example]

```
struct platform_device platform_pcdev_1 = {
    .name = "pcdev-A1x",
    .id = 0,
    .dev = {
              .platform_data = &pcdev_pdata[0],
              .release = pcdev_release
            }
};
struct platform_device platform_pcdev_2 = {
    .name = "pcdev-B1x",
    .id = 1,
    .dev = {
              .platform_data = &pcdev_pdata[1],
              .release = pcdev_release
            }
};
struct platform_device platform_pcdev_3 = {
    .name = "pcdev-C1x"
    .id = 2,
    .dev = {
              .platform_data = &pcdev_pdata[2],
              .release = pcdev_release
            }
};
```

Explore platform_match ⇒ in platform.c file.

In that function,

```
:
/* then try to match against the id table */
if (pdrv->id_table)
    return platform_match_id(pdrv->id_table, pdev) != NULL;
/* fall-back to driver name match */
return (strcmp(pdev->name, drv->name) == 0);
}
```

Device tree:-

What is DT?

→ "Open firmware Device Tree" (or) simply Device Tree (DT) is a data exchange format used for exchanging hardware description data with software OS.

→ It is a description of HW that is readable by OS, so OS doesn't need to hard code details of the machine.

→ In short, it is new and recommended way to describe non discoverable devices (platform devices) to linux kernel, which was previously hardcoded into kernel source files.

→ An OS use DT to discover topology of HW at run time, & by support a majority of available HW without hardcoded information.

Why DT is used?

Linux uses DT for

→ platform identification

→ Device population: Kernel parses the device tree data & generates the required SW data structure which will be used by Kernel Code.

Ideally device tree is independent of any OS. When you change OS, you can still use same device tree file to describe hardware to new OS. That is device tree makes "adding of device information" independent of OS.

writing device tree:-

→ DT supports a hierarchical way of writing HW description at SOC level, common board & board specific level.

→ for example, when we design new board which is slightly different from another reference board then you can reuse the device tree file of reference board & only add information which is new in custom board.

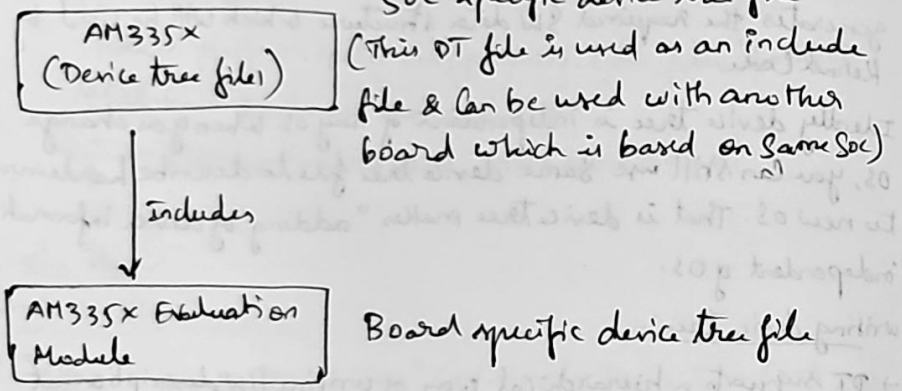
Describing HW hierarchy:-

→ It comes at various level because board has many device blocks

* SOC
* SOC has an on-chip processor & on-chip peripherals.

* Board also have various peripherals onboard like Sensors, LEDs, buttons ..

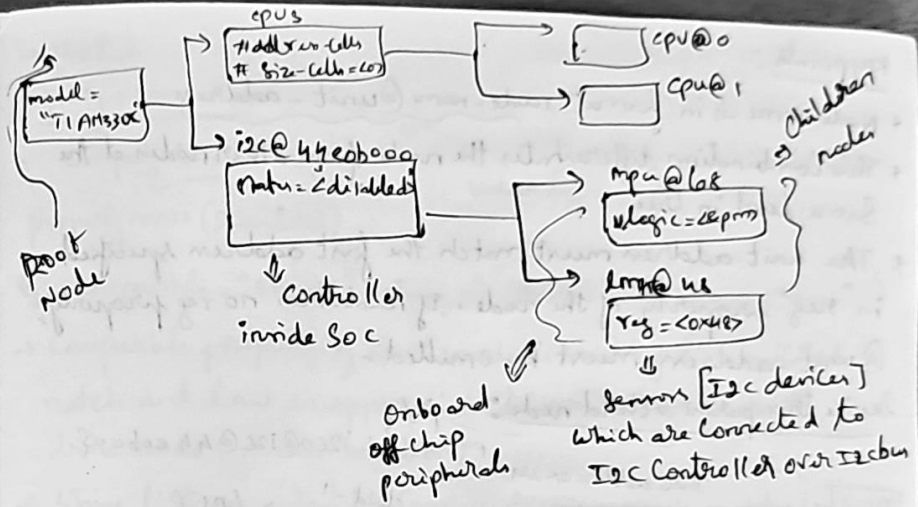
Modular approach to manage DT files :-



Eg:-

Overview of ~~data~~ device tree structure

→ Device tree is collection of device nodes.
→ Device node (or) simply called 'a node' represents a device.
→ They also have parent & child relationship and every device tree must have one root node.
→ A node explains about, that is reveals its data & resources using its 'properties'



How to write a device-tree?

→ Remember that you most probably by writing device tree addons (or) overlays for your board-related changes but not for the entire SoC.
→ SoC specific device tree will be given by Vendor in form of device tree inclusion file (.dtsi) & you just need to include that in your board-level device tree.

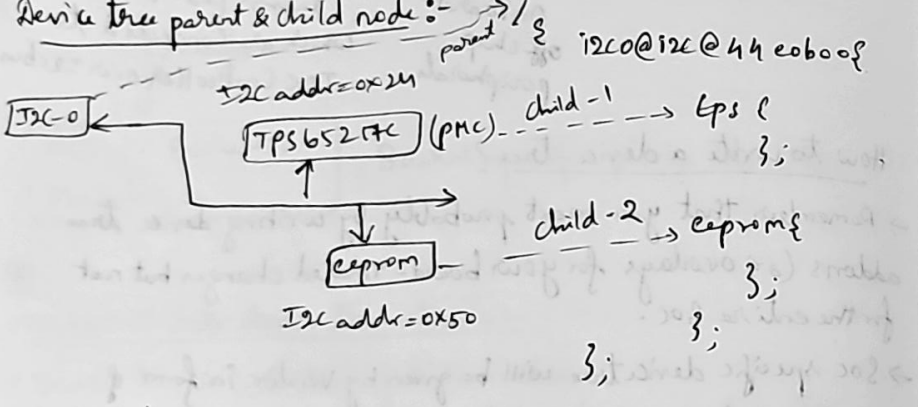
Device-tree writing syntax

→ Node name
→ Node label
→ Standard & non-standard property names
→ Different data type representation (u32, byte, Boolean, etc.)
→ SoC nodes and children

Key points :-

* Node name is in format node-name@unit-address.
* This combination differentiates the node from other nodes at the same level in tree.
* The unit-address must match the first address specified in "reg" property of the node. If node has no reg property, @unit-address must be omitted.

Device tree parent & child node :-



- Different types of properties :-

→ standard property
→ Custom property (non-standard)
→ standard properties are those which is explained by the specification & device-driver binding documentation.
→ Custom properties are specified to particular Vendor (or) Organization which is not documented by specification.

→ ~~That is~~

property : Compatible

Compatible = <string-list>
property name (standard) — Value type

Eg: Compatible = "string1", "string2", "string3"

→ Compatible property of device node is used by linux Kernel to match and load an appropriate driver to handle that device node (driver selection)
→ string list provided by Compatible property is matched against string list supported by driver. If Kernel finds any match, driver will be loaded & probe function is invoked.

Recommendation :- Compatible = <Manufacturer, model>

Root Compatible property of beagleBone black :-

/ {
model = "TI AM335x BeagleBone Black";
Compatible = "ti,am335x-bone-black", "ti-am335x-bone",
"ti,am3xx";

first this string will be searched
if not found, then next string.
... If last string is also not found, device match is failed.

Uses of Compatible property :-

1. Machine identification & initialization
2. Match & load the appropriate driver for the device.

Let's say in Soc file,

```
&i2c0 : i2c@44e6b000 {
    compatible = "ti,omap4-i2c";        → driver
    :
    status = "disabled";                → disabled
};
```

Now in ~~board~~ driver file,

```
static struct platform_driver omap_i2c_driver = {
    .probe = omap_i2c_probe,
    :
    .driver = {
        :
        .of_match_table = of_match_ptr(omap_i2c_of_match),
    }
};
```

If you check the array,

```
static const struct of_device_id omap_i2c_of_match[] = {
    { .compatible = "ti,omap4-i2c",
      .data = &omap4_pdata,
    }, ...
```

In Board level file,

```
&i2c0 {
    status = "okay";        → Here only enabled i2c.
    clock-frequency = <400000>
    tps : {
        }
    eeprom: {
        }
}
```

Device tree bindings

→ How do you know which property name & value pair should be used to describe node in device tree?

Ans → Device tree binding document. The driver writer must document these details.

→ The properties that are necessary to describe device in that device tree depends on requirements of Linux driver for that device.

→ When all the required properties are provided, the driver of charge can detect device from device tree & configure it.

Example: Source Code → Documentation → device tree → Binding → I2C_stm32f4.txt

We can refer this text file to write I2C node for stm32.

points to ~~render~~ remember :-

Case1 :- when driver is already available in linux Kernel for device 'x', but just need to add device 'x' entry in device tree then we must Consult 'x' drivers binding document which guides you through creating device tree node for device 'x'.

Case2 :- when driver is not available for device 'x', then we should write our own driver, we should decide what properties to use, we should provide device tree binding document describing what are all properties & Compatible strings a device tree write must include.

Linux Conventions to write device tree :-

→ hex constants are lowercase :-
  → use "0x" instead of "0X".
  → use a..f instead of A..F, eg 0xf instead of 0XF.

→ node names :-
  → should begin with character in range 'a' to 'z', 'A' to 'Z'
  → unit-address does not have leading "0x" (number is assumed to be hexadecimal)
  → unit-address does not have leading zeros.
  → use dash "-" instead of underscore "_"

→ label names :-
  → should begin with character in range of 'a' to 'z', 'A' to 'Z'.
  → should be lowercase
  → use underscore "_" instead of dash "-".

→ property names :-
  → should be lowercase
  → should begin with character in the range 'a' to 'z'.
  → use dash "-" instead of underscore "_".

pcd driver using device tree node :-

create one dtsi file [am335x-blackbone.dtsi]

```
/{
    pcdev-1 {
            org,size=<512>;
            org,device_serial-num= "PCDEV1ABC123";
            org,perm=<0x11>;
    };
    pcdev-2{
            org,size=<1024>;
            org,device-serial-num = "PCDEV2ABCH16";
    };
    .
    .
};
```

Driver should have own table of list of nodes. When node is gets detected. (ie) device is detected. At the time of_match will search driver's provided table and getting detected.

Inside platform_match function,

```
:
if(of_driver_match_device(dev, drv))
                  ↑ open firmware
    return 1;
```

↳ Inside this func,

```
{
    return of_match_device(drv->of_match_table, dev)!=NULL;
}
```

driver's match table. That's why we need to provide match table list in driver.

Q:-

```
struct of_device_id org_pcdev_dt_match[] =
{
    {.compatible = "pcdev-A1x", .data = (void *) PCDEVA1X},   → This info, we have added in DTS file.
    {.
    "
     {}
};
```

```
struct platform_driver pcd_platform_driver =
{
    .probe = pcd_platform_driver_probe,
    .remove = pcd_platform_driver_remove,
    .id_table = pcdevs_ids,
    .driver = {
                .name = "pseudo_char_device",
                .of_match_table = org_pcdev_dt_match
              }
};
```

Device Tree overlays :-

Two ways you can include device nodes for cape device in main dtb

① Edit main dtb itself (not recommended)

② Overlay (a patch which overlays main dtb) (recommended)

uses of overlays :-

① To support and managed hardware configurations.

② To alter properties of existing device nodes of main dtb.

③ Overlays approach maintains modularity and makes management easier.

Overlay DTS format :-

DTS of an overlay should have following format

```
{
    fragment@0{ */ first child node */
            target=<phandle>;
            __overlay__{
                    property-a;
                    node-a {
                    };
            };
    fragment@1{ ...
        };
}
```

Eg:

In board.dts

```
/dts-v1/;
/{
    compatible = "org, board-am335";
    ocp: ocp {
            i2c0: i2c@4800000{
                    status = "disabled";
            };
    };
};
```

In overlay file,

```
/dts-v1/;
/plugin/;
/{
    fragment@0{
            target = <&i2c0>;   → handle which want to modify.
            __overlay__{
                    status = "ok";
                    ..
            };
    };
};  /* End of overlay
```

Let's write for pcd device

By default, all device nodes will be enabled -

```
/dts-v1/;
/plugin/;
/{
    fragment@0{
            target = <&pcdev1>;
            __OVERLAY__{
                    status = "disabled";
            };
    };
};
```

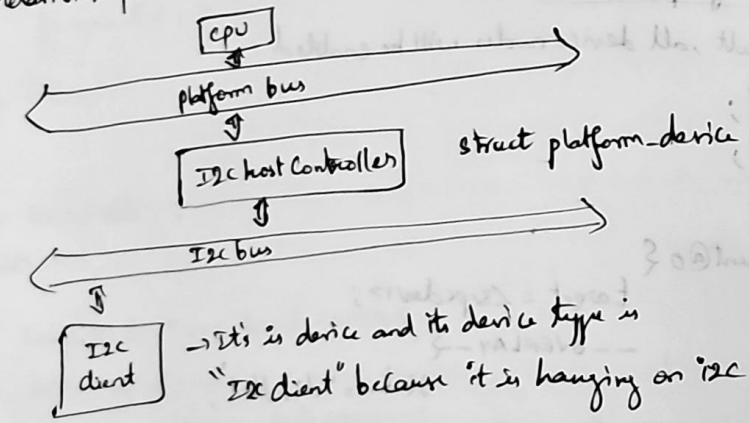
" ↑ for all 4 device nodes, need to do.

Linux device model :-

→ Linux device model is nothing but collection of various data structures and helper functions that provide unifying and hierarchical view of all the buses, devices, drivers present on the system. We can access whole linux device and driver model through pseudo filesystem called sysfs which is mounted at /sys

→ Different components of linux device model is represented as file & directories through sysfs.

→ sysfs exposes underlying bus, device, driver details and their relationships in linux device model.



→ It's is device and its device type is "I2C client" because it is hanging on i2c bus type.

Kobject :-

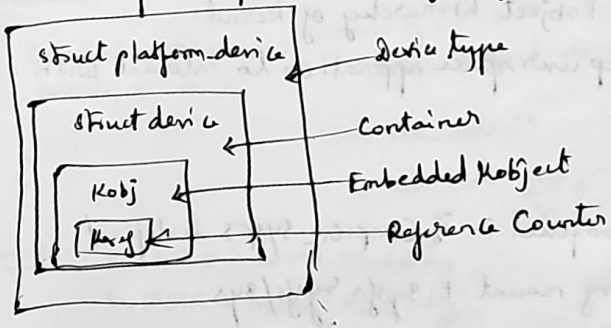
→ It stands for kernel object which is represented by struct kobject

→ Kobjects are fundamental building blocks of linux device and driver hierarchy.

→ Kobjects are used to represent "Containers" in sysfs virtual filesystem.

→ Kobjects are used for reference counting of "Containers".

→ sysfs filesystem gets populated because of Kobjects, sysfs is user space representation of Kobjects hierarchy.



Kobject type

→ Type of Kobject is determined based on type of the Container in which Kobject is embedded.

→ struct kobj_type

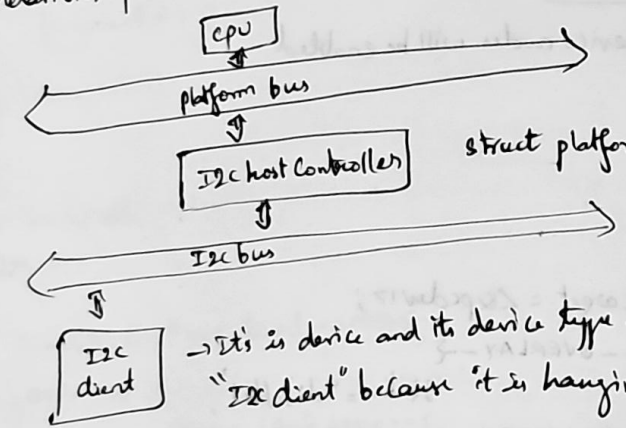
This structure is used to define the default behavior.

Linux device model :-

→ Linux device model is nothing but collection of various data structures and helper functions that provide unifying and hierarchical view of all the buses, devices, drivers present on the system. We can access whole linux device and driver model through pseudo filesystem called sysfs which is mounted at /sys.

→ Different components of linux device model is represented as file & directories through sysfs.

→ sysfs exposes underlying bus, device, driver details and their relationships in linux device model.



→ It's is device and its device type is "I2C client" because it is hanging on i2c bus type.

Kobject :-

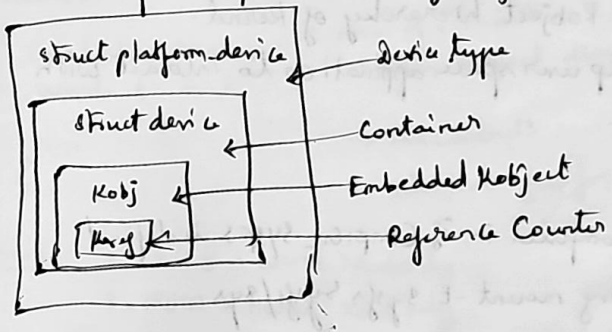
→ It stands for kernel object which is represented by struct kobject

→ Kobjects are fundamental building blocks of linux device and driver hierarchy.

→ Kobjects are used to represent "Containers" in sysfs virtual filesystem.

→ Kobjects are used for reference counting of "Containers".

→ sysfs filesystem gets populated because of Kobjects, sysfs is user space representation of Kobjects hierarchy.



Kobject type

→ Type of Kobject is determined based on type of the Container in which Kobject is embedded.

→ struct kobj-type

This structure is used to define the default behavior.

Kset :- → Aggregation and collection of Kobjects of same type.

$ /sys/devices/platform # ls
↓ Subsystem ↓ Kset ↓ o/p

#
Acp 3:00
pcdev_A1X.0
pcdev-A2X.1
pcdev-B1X.
pcdev-B2.X3

→ Kobject.

Sysfs :- → It is virtual in-memory file system which provides
1. Representation of Kobject hierarchy of Kernel.
2. Attributes to help user space application to interact with devices and drivers.

Using Sysfs
→ It is always compiled in if CONFIG_SYSFS is defined.
→ Can access by doing mount -t sysfs sysfs/sys

Kobject attribute :-
→ They are regular files (or) symbolic names that appear in sysfs's Kobject attributes directory and they are used to expose details about Kobject's "Contains" to user-space

Eg:- /sys/class/pcd_class/pcdev-0/ $ ls -l

o/p: total 0
-r--r--r-- dev
lrwxrwxrwx device → ../../pcdev-A1X.0
-rw-r----- power
-rw-r--r-- uevent

Kobject => attributes

Creating custom attributes of device :-
sysfs attributes are represented by below structure
struct attribute
{
  const char * name; → name of attribute → This name will show in sysfs kobject directory as attribute's name
  umode_t mode; → This controls read/write permissions for attribute file from user space programs.
  :
}

mode: S_IRUGO → world read only
S_IRUSR → owner read only
S_IRUGO | S_IWUSR: world-read and only owner write

Apis for managing sysfs files (Attributes) :-
int sysfs_create_file(struct Kobject *Kobj, const struct attribute *attr);
int sysfs_chmod_file(struct Kobject *Kobj, const struct attribute *attr, umode_t mode);

File I/O operations on sysfs attribute :-

→ Once we create sysfs file (attributes), you should provide read and write methods for them so that user can read value of the attribute (or) write new value to attribute

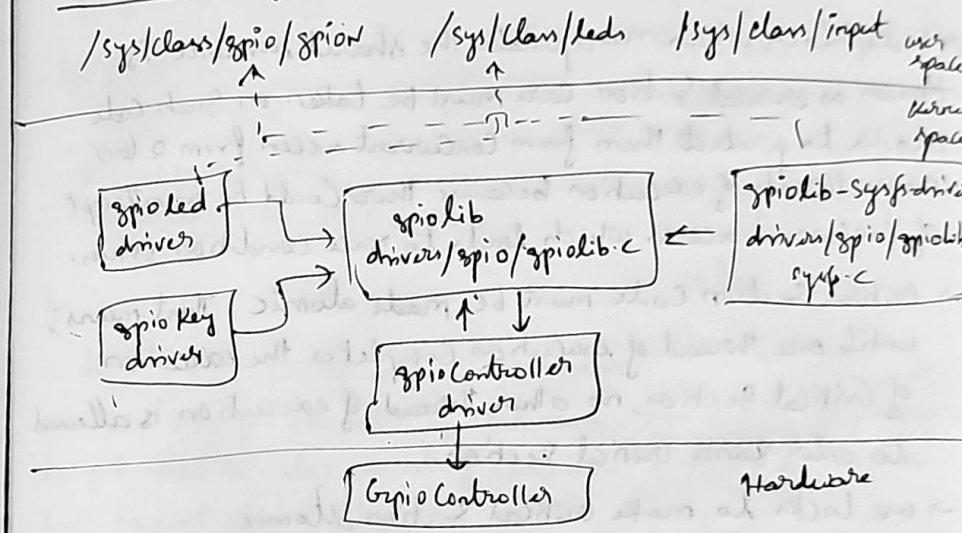
Eg: Cat uevent
→ like this

→ But if we see struct attribute there is no place to hook read/write methods for an attribute. Basically it just represents name of attribute and mode.

```
struct device_attribute
{
    struct attribute attr;
    ssize_t (*show)(struct device *dev, struct device_attribute
        *attr, char *buf);
    ssize_t (*store)(struct device *dev, struct device_attribute
        *attr, const char *buf, size_t count);
};
```

Instead of manually creating variable struct device_attribute and initializing them use DEVICE_ATTR_xx macros given in include/linux/device.h

Linux Gpio subsystem



Race condition :-

→ Race condition is runtime phenomenon where 2 (or) more threads of execution race towards shared resources to access it. This may result in inconsistent state of shared resources and can cause software bug.

Shared resources and critical section:-

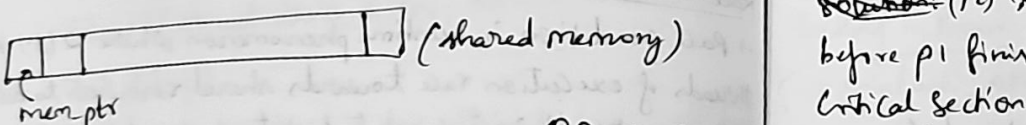
→ A shared resource is a resource such as global variable, a shared memory (or) peripheral that can be accessed by multiple threads of execution executing on same processor (or) different processors.

Critical Section :-

→ Code block which manipulates the shared resource is known as critical section. Care must be taken on such code blocks to protect them from concurrent access from 2 or more threads of execution because there could be an attempt of concurrent access which leads to race condition often.

→ Critical section code must be made atomic. That means, until one thread of execution completes the execution of critical section, no other thread of execution is allowed to enter same critical section.

→ use locks to make critical section atomic



P1 process

Store(10, mem_ptr)

Increment(mem_ptr)

is preempt happen ?

P2 process

Store(99, mem_ptr)

Increment(mem_ptr)

→ results in needed for locking.

Software Lock (global locking)

lock = 0 0 → available
1 → not available.

P1 process

```
if(lock==0){
  lock=1
  store(10, mem_ptr)
  increment(mem_ptr)
  lock=0
}
```

P2 process

```
if(lock==0){
  lock=1
  store(99, mem_ptr)
  increment(mem_ptr)
  lock=0
}
```

Will it solve problem of race condition?

No, let's say P1 executes if statement, immediately preempted to P2, that can also execute if statement because at the time lock is not changed by P1.

(i.e) Atomicity of critical section is not achieved. before P1 finishes critical section, P2 raced towards the critical section.

Problem to solve :-

```
if(lock==0)
{
  lock=1;
}
```

This should be made atomic.

Most of the processor architecture provides a dedicated atomic instruction for "test and set functionality" which can do exclusive load and store.

Solution:- lock=0

P1 process

```
if(!test_and_set(&lock)){
store(10, mem_ptr)
Increment(mem_ptr)
Lock=0
}
```

P2 process

```
if(!test_and_set(&lock)){
store(99, mem_ptr))
Increment(mem_ptr)
}
```

(atomic)

```
int test_and_set(int *addr)
{
    int old = *addr; // realised -> LOAD
    *addr = 1;  // -> STR
    return old;
}
```

SMP & UP Systems

-> Symmetric Multiprocessor System and UP stands for uniprocessor System.

-> Linux 2.0 & later versions support SMP.

Scenario 1:- UP system + No kernel process pre-emption + No intrs.

-> No locking required for shared resource.

Scenario 2:- UP system + No kernel process pre-emption + intrs enabled

No locking is required b/n p1 & p2. Locking is required if ISR accesses shared resource otherwise not.

Scenario 3:- UP system + kernel process pre-emption enabled

Locking is required.

Scenario 4:- SMP system + Kernel process pre-emption disabled.

Locking is required.

Scenario 5:- SMP System + Kernel process pre-emption enabled

locking is required.

Scenario 6:- -> SMP system + interrupts enabled.

-> locking is required b/n p1 & ISR

Locking in linux

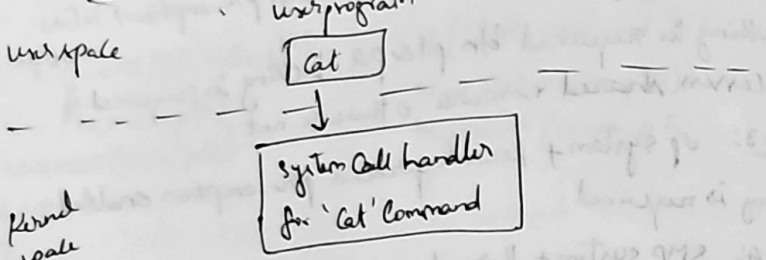
-> spinlock, -> Mutex, -> Semaphore

Guidelines:

-> If you are protecting critical section in the code which runs on behalf of the user request (process Context (or) user Context) then you can either use spinlock (or) mutex to guard the critical section.

-> If you want to protect critical section of ISR, then use spinlock.

User Context Vs Interrupt Context



Never use 'mutex' as locking mechanism when critical Section executes in interrupt Context.

Spinlock

→ It is Kernel synchronisation Service used to avoid race Condition while accessing shared resource.

→ Only one thread of execution can hold spinlock at any given point in time

→ When thread tries to acquire a spinlock which is already held, the thread busy loops (or) spins on the processor until the lock becomes available.

→ Since thread busy loops on processor, wasting processor execution cycles, it is not wise to hold spinlock for longer duration.

→ They protect shared resources by disabling Kernel pre-emption & interrupts on the local processor

→ spinlocks are non-recursive in linux. A thread tries to acquire spinlock which is already held by same thread, the thread busy loops waiting for itself to release spinlock that will never happen because it is busy looping, so this leads to self deadlock.

→ It is not recommended to use spinlock to protect the critical section that sleeps (blocks)

spinlock initialization

→ 2 types of initialization → Static
→ Dynamic

→ #include/linux/spinlock_types.h → static DEFINE_SPINLOCK → macro

→ Dynamic → spinlock_t xxx_lock;
spinlock_init (&xxx_lock);

Mutexes

→ While spinlock is spinning lock of linux, mutex is sleeping lock of linux.

→ When thread tries to acquire mutex which is already held, the thread will be put on to wait queue and it sleeps. The processor can execute other threads meanwhile. When the mutex holder releases the lock, one of the tasks waiting in the wait queue will be awakened to acquire the lock.

→ Mutex implements the ownership of lock. So, the owner who locked it should unlock it.
→ Mutex should not be used from interrupt Context Code such as H/W/S/W I&Rs.
→ Mutex may be used in process Context, not in interrupt Context. During lock contention, thread will be put in to sleep and scheduler schedules other tasks.
- ⊗ Never use mutex in interrupt Context Codes such as H/W handlers, timers because interrupt Context is not Schedulable
- ⊗ Since mutex puts tasks to sleep; it is suitable for locking the critical section, which is time-Consuming (where lock time is longer)

Embedded linux Audio

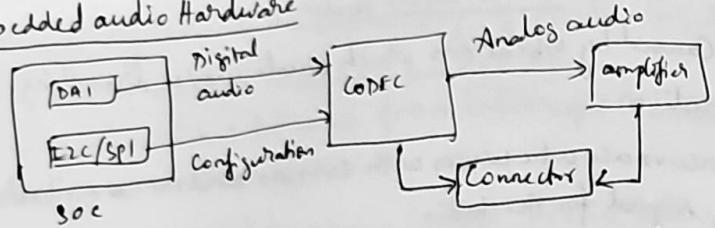
Sound → Caused by vibrations which create waves, travelling through medium.

* Human perceive acoustic waves with ears as eardrum are vibrating converting signal for the brain.
* Measured in frequency (Hz) & amplitude measured in (dB).
* Sample rate (or) Sampling freq is no. of samples taken per second.
* Shannon-Nyquist theorem states that Sampling freq needs to be atleast twice maximum signal freq to accurately digitized signal. Hearable range → 20Hz - 20KHz.
* Common Sample sizes are 16 & 24 bits.
* 8 bits is getting very rare due to poor audio Quality and 32 bits Samples can be used when specific alignment is required.
* Ways to store Samples in memory → Signed, unsigned, floating.
* For 24 bits Samples, packing can also differ: either they are packed on 3 bytes (or) they can be packed in 32bit.
* Store Sound as sequence of Samples

WAV is format based on RIFF & has following header :-

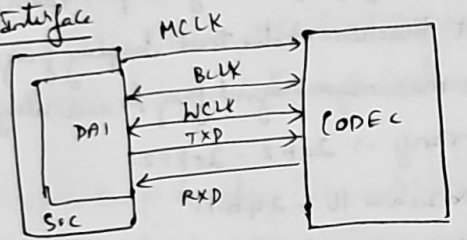
1, 4 → RIFF FOURCC Code, 5-8 → file size, 9-12 → "WAVE"
13→16 → "fmt", 17-20 16 → length of format data, 21→22 → Audio format, 23→24 → No. of channels, 25-28 → Sample rate, 29-32 → Byte rate 33-34 → Block Align, 35-36 → Bit per Sample, 37-40 → data, 41-44 → Size of data

Embedded audio Hardware



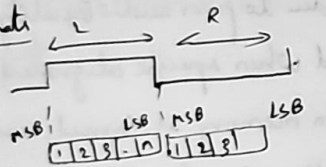
CODECS → It is device that CODECs & DECodes audio samples, It integrates ADCs & DAC into single chip.

Common PCM Interface

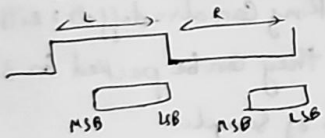


Digital formats

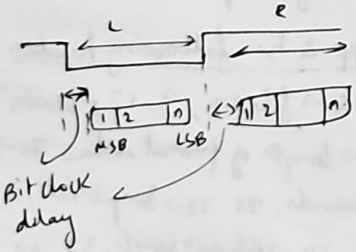
L-J:

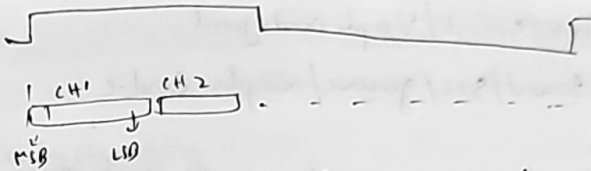


R-J:



I2S:





PDM: → Two signals per channels, clock & Data. Data has only one bit.

ASOC Components:-

ASOC → ALSA System on chip → Linux Kernel Subsystem created to provide better ALSA support for SoC & portable audio codecs. It allows to reuse codec drivers across multiple architectures and provides API to integrate them with SoC audio interface.

① Codec class drivers → define codec capabilities (audio interface, audio Controls)

② platform class drivers → define SoC audio interface (also referred as CPU DAI) sets up DMA when applicable.

③ Codec to platform Integration → usually done through device-tree, previously required writing machine driver in C.

Simple-Card :-

* Most sound cards can now be described using Device tree. This is done using sound node with simple-audio-card Compatible string.

Binding - bindings/sound/simple-card.yaml

Driver handling - sound/soc/generic/simple-card.c

Machine driver:-

It registers struct snd_soc_card → #include/sound/soc.h

struct snd_soc_dai_link is used to create link between CPU DAI and Codec DAI.

Routing:- → After linking Codec driver with SOC SAI driver, it is still necessary to define what are Codec outputs & inputs that are actually used onboard. This is called routing.

① statically ② from device tree

① statically:- → using .dapm_routes & .num_dapm_routes members of struct snd_soc_card.

② From device tree

Eg:- snd_soc_of_parse_audio_routing(card, "atmel,audio-routing");

In DT, => atmel,audio-routing = "Headphone Jack", "HPOUTL";

Codec pins ⇒ defined in Codec driver ⇒ SND_SOC_DAPM_INPUT & SND_SOC_DAPM_OUTPUT

Board Connectors → defined in machine driver ⇒ struct snd_soc_dapm_widget

docking: producer/Consumer ⇒ declared part of .dai_fmt field of struct snd_soc_dai_link.

dynamically changing docks

.ops member of struct snd_soc_dai_link Contains useful Callbacks. snd_soc_ops ⇒ /sound/soc.h

.hw_params is called when setting up the audio stream. The struct snd_pcm_hw_params Contains audio characteristics.

clocking: hw_params:-

params_rate, params_channels, params_format.

ASoC Component Controls

→ Controls allow to export Configuration Knobs of the Component to userspace.

→ KControl names

ASoC Component Callback:-

hw_params:-

→ the most useful Callback

→ It is used to Configure the Component to match the parameters of the audio stream.

→ Called when stream is ready to be played, before any data is transferred.

→ If requested parameters cannot be supported by hardware hw_params callback (or) returns an error code to indicate stream cannot be opened.

ASOC DAPM:-

→ Dynamic Audio power Management
→ Goal is to save much power as possible by shutting down audio routes that are not in use.
→ Having two objects → DAPM widgets
→ DAPM routes
→ DAPM widgets represent various components of an audio system such as audio inputs, outputs, mixers and amplifiers.
→ Routes are connecting widgets together.

CPU DAI driver:-

→ It is now component driver like codec ones.
→ It is usually more complex as it need to handle IRQs and take care of pinmuxing, clocks and DMA.

alsa-lib:-

→ libasound → library name
→ alsa/asoundlib.h

ALSA

Advanced Linux Sound architecture.

→ ALSA cards corresponds to one-to-one hardware sound card.
→ Most of ALSA's hardware access happens at device level.

plugins → whenever player (or) other program uses ALSA's device plugins do the work.

hw plugin:- → It does no processing of its own, but merely accesses the hardware driver. If app. chooses pcm parameters (sampling rate, sample format (or) channel count) which hardware does not support, hw plugin returns an error.

plug plugin:- which performs channel duplication, sample value conversion and resampling when necessary. unlike the 'hw' plugin which is used by the device hw:0,0 the device corresponding to 'plug' plugin is named differently 'plughw:0,0'.

file plugin:- → which writes sample data to a file. It has two arguments the file name and the format (which is always "raw").

tee plugin:- → which passes data to another device in addition to writing it to a file and has device as its first argument.

Eg: aplay -Dtee:"'plughw:0,0'", /tmp/alltee-out, raw xy.wav

tee's o/p is always raw sample data without header.
'file' plugin can be used to read data from file.

Configuration files:-
Without these definitions, we could not use any of the features of ALSA - play no sound, adjust no mixer, still no need to write conf file which is already available as 'builtin' conf ① alsa.conf (usr/share/alsa). This directory contains further conf. files which are sound card and plugin specific → not to be changed by the user (or) system administrator.

② System-wide configuration file → /etc/asoundrc. users can store their configuration file in asoundrc in their home directory. All conf. files are parsed everytime ALSA device is opened. So changes takes effect immediately.

aplay -L => List all definitions in pcm interface.
other interfaces are ctl (Control for controlling hardware) seq (sequencer), hwdep (hardware dependent features), mixer (mixer control)

Eg: pcm.plug0 {
    type plug
    slave {
        pcm "hw:0,0"
    }
}

This means new pcm device is created by above "plug0" accessible via pcm interface. Data o/p on this device shall be handled by 'plug' plugin. The plugin uses as a slave the PCM device hw:0,0. This device definition was written in the way which is common.

Advanced Configuration file features

→ overriding parameters and parameter data types

pcm.!default { type hw card 0 }

LFE → Low frequency effect → in subwoofer. There will be separate channel for "LFE".

ALSA topology

→ It provides method of audio drivers to load their mixers, routing, PCMs and capabilities from user space at runtime without changing any driver source code. The intention is to write driver once and do the differentiation in topology.

Why we need topology?

→ It removes the need of re-writing (or) porting audio drivers to different devices (or) different firmwares. A single driver can be used on different devices by updating topology data from file system.

Topology objects can be configured by user space include: Controls, widgets, routes, PCMs and configurations for physical DAI & DAI Links

User space: → topology library is part of alsa-lib. users can define topology objects that describe topology of customer firmware.

→ Topology objects can be defined either in text configuration file.

→ Topology library will parse them and generate binary file for kernel.

How to generate topology binary file

→ users can define topology objects in text configuration file.

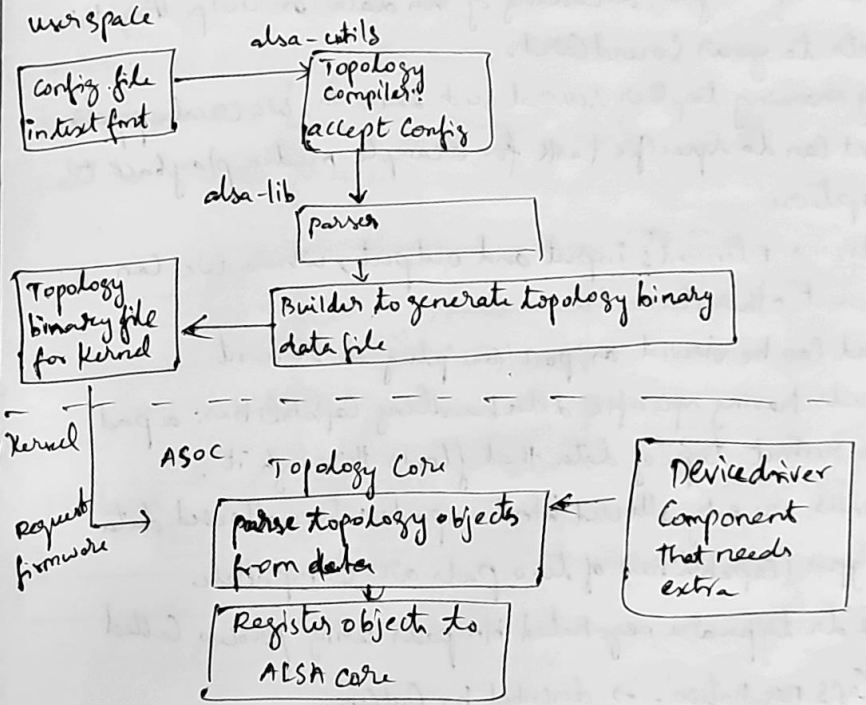
→ Syntax is based on alsa conf. [alsa-lib/include/alsa/topology.h]

→ users can use alsatplg → topology tool in alsa-utils, to convert topology text configuration file to binary.

alsatplg -c <path of textfile> -o <path_of_binaryfile>.

Kernel :-

Kernel driver provides API for device drivers to load the topology binary from user space. It will bind the objects with vendor specific ops & pass them to device driver.



Gstreamer

→ Gstreamer is framework for creating streaming media applications.

Basic Concepts of Gstreamer

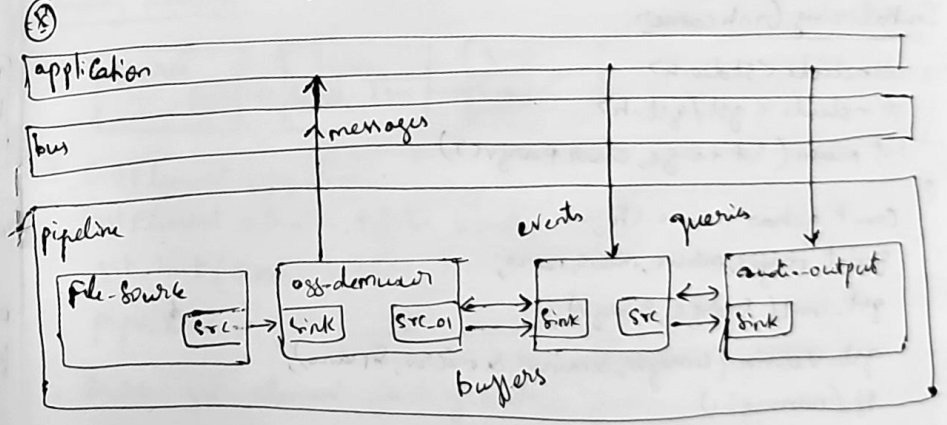
Elements → most important class of objects in Gstreamer.
→ usually create chain of elements linked together and let data flow through this chain of elements.
→ Element has one specific function which can be the reading of data from file, decoding of this data (or) outputting this data to your SoundCard.
→ By chaining together several such elements, we create pipeline that can do specific task for example media playback (or) Capture.

pads → Element's input and output, where we can connect other elements.
→ pad can be viewed as 'port' (or) 'plug' on element.
→ pads having specific data handling capabilities: a pad can restrict type of data that flows through it.
→ Links are only allowed b/n two pads when allowed data types (Capabilities) of two pads are compatible.
→ Data types are negotiated b/n pads using process called 'Caps negotiation'. → described by GstCaps.
→ Data flows out of one element through one (or) more source pads and elements accepting incoming data through one (or) more sink pads.

Bins and pipelines :-

→ A bin is container for collection of elements, thereby abstracting away lot of complexity for our application.
→ For example, Change state on all elements in bin by changing the state of the bin itself.

pipeline → It is top level bin.
→ As we set it to "paused" (or) "playing" state, data flow will start and media processing will take place.



→ Buffers are objects for passing streaming data b/n elements in the pipeline.
→ Events are objects sent b/n elements (or) from applications to elements.
→ Messages are objects posted by elements on the pipeline's message bus, where they will be held for collection by the application → used to transmit information such as errors, tags, state changes
→ Queries allow applications to request information such as duration (or) current playback position from pipeline. Elements can also use queries to request information from their peer elements.

Initializing GstReamer

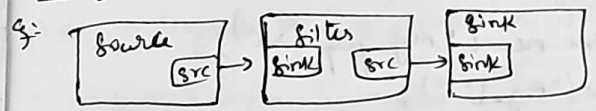
```
#include <stdio.h>
#include <gst/gst.h>
int main (int *argc, char *argv[])
{
  const gchar * nano_str;
  guint major, minor, micro, nano;
  gst_init( &argc, &argv);
  gst_version (&major, &minor, &micro, &nano);
  if (nano == 1)
    nano_str = "(CVS)";
  else if (nano == 2)
    nano_str = "(prerelease)";  return 0; }
```

Creating GstElement

* Simplest way to create an element is to use gst_element_factory_make(). → This fn takes factory name and an element name for the newly created element.
* When we don't need element anymore, we need to unref it using gst_object_unref(). → This element has refCount of 1 when it gets created, an element gets destroyed when refCount is decreased to '0'.

```
element = gst_element_factory_make ("fakesrc", "source");
.....
gst_object_unref (GST_OBJECT (element));
```

Linking elements

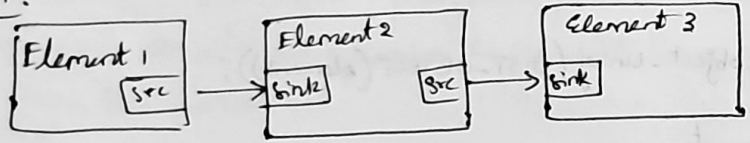



```
GstElement *pipeline,
GstElement *source, *filter, *sink;
gst_init (&argc, &argv);
pipeline = gst_pipeline_new ("mypipeline");
source = gst_element_factory_make ("fakesrc", "source");
filter = gst_element_factory_make ("identity", "filter");
```

```
sink = gst_element_factory_make ("fakesink", "sink");
/* must add elements to pipeline before linking them */
gst_bin_add_many (GST_BIN(pipeline), source, filter, sink, NULL);
/* Link */
if ( ! (gst_element_link_many (source, filter, sink, NULL))
    {
      g_warning ("failed to link element !");
    }
?
```

Change the state of an element using gst_element_set_state()

Bin :-




```
=> gst_init (&argc, &argv);
   pipeline = gst_pipeline_new ("my_pipeline");
     bin = gst_bin_new ("my_bin");
   source = gst_element_factory_make ("fakesrc", "source");
   sink = gst_element_factory_make ("fakesink", "sink");
   gst_bin_add_many (GST_BIN(bin), source, sink, NULL);
   gst_bin_add (GST_BIN(pipeline), bin);
   gst_element_link (source, sink);
```

Bus :- → * Every pipeline contains bus by default, so applications do not need to create bus.

* only things applications should do is set message handler on bus, which is similar to signal handler to an object. When mainloop is running, the bus will periodically be checked for new messages and callback will be called when any message is available.

Eg: =>
```
bus = gst_pipeline_get_bus (GST_PIPELINE (pipeline));
gst_bus_add_signal_watch (bus);
g_signal_connect (bus, "message::error", G_CALLBACK
  (cb_message_error), NULL);
g_signal_connect (bus, "message::eos", G_CALLBACK
  (cb_message_eos), NULL);
```

pads :- → Defined by two properties → Direction
                                    → availability

* Gstreamer defines two pad directions → source pads
                                       → sink pads

* pad availability → Always
                   → Sometimes
                   → on request

Buffers :-

Buffer is created, memory allocated, data put in it & passed to the next element. That elements reads the data, does something and unreferences the buffer. This causes

data to be free'd and buffer to be destroyed.

Events :-

→ Events are control particles that are sent both up-and-downstream in pipeline along with buffers.

Tutorials → Hello world ①

```
int main ()
{
 GstElement * pipeline,
 GstBus * bus,
 GstMessage *msg,
 gst_init(&argc, &argv);
 pipeline = gst_parse_launch
  ("playbin uri = https://gstreamer.../data/media/sintel_480p.webm",
    NULL);
gst_element_set_state (pipeline, GST_STATE_PLAYING);
 bus = gst_element_get_bus (pipeline);
msg = gst_bus_timed_pop_filtered (bus, GST_CLOCK_TIME_NONE,
      GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
 gst_message_unref (msg);
 gst_object_unref (bus);
 gst_element_set_state (pipeline, GST_STATE_NULL);
 gst_object_unref (pipeline);
 return 0;
}
```

gst_parse_launch :-

↳ In Gstreamer we usually build the pipeline by manually assembling the individual elements, but when pipeline is easy enough, we don't need advanced features, we can go shortcut of this function.

playbin → special element which acts as source and as sink, is whole pipeline. Internally it creates and connects all the necessary elements to play the media, we don't have to worry about it.

→ here we are only passing one parameter to playbin which is URI of the media we want to play.

bus_timed_pop_filtered :-

→ These lines will wait until an error occurs (or) the end of the stream is found.

Code → This will open a window and displays a movie with audio, media is fetched from internet.

Tutorial → Hello world ② → Manual

```
int main ()
{
 GstElement *pipeline, *source, *sink;
 GstBus *bus;
 GstMessage *msg;
 gst_init (&argc, &argv);
```

```
source = gst_element_factory_make("videotestsrc", "source");
sink = gst_element_factory_make("autovideosink", "sink");
pipeline = gst_pipeline_new("test-pipeline");
gst_bin_add_many(GST_BIN(pipeline), source, sink, NULL);
gst_element_link(source, sink);
g_object_set(source, "pattern", 0, NULL);
gst_element_set_state(pipeline, GST_STATE_PLAYING);
bus = gst_element_get_bus(pipeline);
msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE,
  GST_MESSAGE_ERROR | GST_MESSAGE_EOS);
```

!

Tutorial ③: Dynamic pipelines

Gstreamer allows building pipeline "on the fly" as info available. Instead of having monolithic pipeline defined at beginning of your application.

→ Main Complexity when dealing with demuxers is that they cannot produce any information until they have received some data.

→ Solution is to build pipeline from the source down to the demuxer, set it to run (play). When demuxer has received info, it will start creating source pads.

```
data.source = gst_element_factory_make("videotestsrc", "source");
data.convert = gst_element_factory_make("audioconvert", "convert");
data.resample = gst_element_factory_make("audioresample", "resample");
data.sink = gst_element_factory_make("autoaudiosink", "sink");
data.pipeline = gst_pipeline_new("test-pipeline");
gst_bin_add_many(GST_BIN(data.pipeline), data.source,
                 data.sink, data.convert, data.resample,
  data.sink, NULL);
if(!gst_element_link_many(data.convert, data.resample,
                          data.sink, NULL)
  {
  return -1;
}
g_object_set(data.source, "uri", "https://...webm");
g_signal_connect(data.source, "pad-added", G_CALLBACK
  (pad_added_handler), &data);
gst_element_set_state(data.pipeline, GST_STATE_PLAYING);
bus = gst_element_get_bus(data.pipeline);
do {
  msg = gst_bus_timed_pop_filtered(bus, GST_CLOCK_TIME_NONE,
  GST_MESSAGE_STATE_CHANGED | ...);
```

```
static void pad_added_handler (GstElement *src, GstPad *new_pad,
CustomData *data)
{
GstPad *sink_pad = gst_element_get_static_pad (data->Convert,
  "sink");
GstCaps *new_pad_caps = NULL;
GstStructure *new_pad_struct = NULL;
const gchar *new_pad_type = NULL;
g_print ("Received new pad '%s' from '%s':\n", GST_PAD_NAME
(new_pad), GST_ELEMENT_NAME (src));
if (gst_pad_is_linked (sink_pad)
  {...
  }
new_pad_caps = gst_pad_get_current_caps (new_pad);
new_pad_struct = gst_caps_get_structure (new_pad_caps, 0);
new_pad_type = gst_structure_get_name (new_pad_struct);
if (!g_str_has_prefix (new_pad_type, "audio/x-raw"))
{...
  }
```

```
ret = gst_pad_link (new_pad, sink_pad);
  :
```

Walkthrough of the code

uridecodebin → will internally instantiate all the necessary elements (sources, demuxers & decoders) to turn URI into raw audio (or) video streams. It does the half the work that playbin does.

audioconvert : useful for converting b/n different audio formats.

audioresample : useful for converting b/n different audio sample rates.

autoaudiosink : render the audio stream to the audio card.

Here we link elements converter, resample and sink but we don't link them with the source, since at this point it contains no source pads.

g_signal_connect() : we are attaching to "pad-added" signal of our source (an uridecodebin element). To do so, we use g_signal_connect () & provide callback function to be used (pad_added_handler) & data pointer.

Callback function Code :-

```
GstPad *sink_pad = gst_element_get_static_pad (data->convert, "sink");
```

We extract the Converter element & retrieve its sink pad using gst_element_get_static_pad(). This is the pad to which we want to link new_pad.

decodebin can create many pads as web sees fit, and for each one, this callback will be called. These lines of code will prevent us from trying to link to new pad once we are already linked.

```
if (gst_pad_is_linked (sink_pad))
  {
    ...
    goto exit;
  }
```

→ Now we will check type of data this new pad is going to output, because we are only interested in pads producing audio.

gst_pad_get_current_caps() → retrieves the Current Capabilities of the pad.

(*) A pad can offer many capabilities & hence GstCaps can contain many GstStructure each representing different Capability. The current caps on pad will have single GstStructure and represent single media format ~~only~~.

→ In our case, we know the pad we want only had one Capability (audio), we retrieve the first GstStructure with gst_caps_get_structure()

→ finally gst_structure_get_name() → recover the name of the structure which contains main description of the format.

→ If name is not "audio/x-raw", this is not decoded o/p & we are not interested in it.

Otherwise attempt link

(i.e) gst_pad_link (new_pad, sink_pad);

Tutorial :- Media formats and pad Capabilities

pads :- → It allows information to enter and leave an element. The Capabilities of a pad, then specify what kind of information can travel through pad.

⇒ 16 bits-per-sample audio 5.1 channels at 44100 samples per second.

(E)(F) → pads can support multiple capabilities [type of RGB like video and capabilities can be specified as ranges [sample rate ...].

→ However actual information traveling from pad to pad must have only one well-specified type. Thus process known as negotiation, two linked pads agree on common type and capabilities of the pads becomes fixed.

(G) In order for two elements to be linked together, they must share common subset of capabilities.

Tutorial - multithreading :-

Multithreading

→ GStreamer is a multithreaded framework. → That mean Internally, it create and destroys threads as it needs them.

→ For example to decouple streaming from application thread.

Queue element

→ Sink pad just enqueues data & returns control. & on different thread data is dequeued and pushed downstream. This element is also used for buffering.

* Source is synthetic audio signal which is split using "tee" element (it sends through its source pads everything it receives through its sink pad).

* One branch then sends the signal to the audio card, other renders a video of the waveform and send its to the screen.

Tutorial: Short-Cutting the pipeline

* Inject external data into Gstreamer pipeline
* extract data from general Gstreamer pipeline.
* Access & manipulate this data.

→ Element used to inject application data into Gstreamer pipeline is appsrc.

→ Element used to extract Gstreamer data back to the application is appsink.

Buffers → * represents data travels through Gstreamer pipeline in chunks called buffers.

* Source pad produces buffers that are consumed by sinkpads; Gstreamer take these buffers and passes them from element to element.

Tutorial: GstFreamer-tools:

gst-launch-1.0:

L) Tool accepts textual description of pipeline, instantiates it & sets it to PLAYING state.

Is primarily a debugging tool for developers. we should not build app on it, instead use gst_parse_launch().

Eg: gst-launch-1.0 videotestsrc ! videoconvert ! autovideosink

properties:

Named elements: → Elements can be named using "name" property. Named elements are referred to using their named followed by dot.

Eg:

[gst-launch-1.0 videotestsrc ! videoconvert ! tee name=t !
queue ! autovideosink t. ! queue ! autovideosink]

→ Example instantiates "videotestsrc" linked to "videoconvert", linked to tee (tee copies to each of its output pads everything coming through its input pad). tee is named simply 't' (using the name property), then linked into "queue" and "autovideosink". The same "tee" is referred to using 't'. (mind dot) & linked to second "queue" & second "autovideosink".

pads:

→ Adding dot plus pad name after name of the element.

Eg:

gst-launch-1.0 souphttpsrc location=https://gstreamer.freedesktop.org/media/intel-trailer-480p.webm ! matroskademux name=d d.video_0 ! matroskamux ! filesink location=sintel_video.mkv

→ Fetches media file from the internet using SoupHttpSrc, which is the webm format (special kind of Matroska Container). We then open the Container using matroskademux.
→ This media Contains both audio and video, so matraskademux will create two output pads, named video_0 & audio_0.
→ We link video_0 to matroskamux element to re-pack the video stream into new Container and finally linked to filesink, which will write the stream into file named 'bintel_video.mkv'.

If we wanted to keep only audio :-

[gst-launch-1.0 SoupHttpSrc location=https://gstreamer.freedesktop.org/media/bintel-trailer-480p.webm ! matroskademux name=d d.audio_0 ! vorbisparse ! matroskamux ! filesink location=bintel-audio.mka]

→ vorbisparse element is required to extract some information from stream & put it in pad Caps, so the next element matroskamux knows how to deal with stream. In Case of video, this was not necessary because already matroskademux extracted this information & added in Caps.

Note: No media has been decoded (or) played.
We have just moved from one Container to another.
(demultiplexing & remultiplexing again)

Cap filters:

Eg:
[gst-launch-1.0 SoupHttpSrc location=https://gstreamer.org/data/media/bintel-trailer-480p.webm ! matroskademux ! filesink location=test]

→ Input Caps of filesink are ANY meaning that it Can accept any kind of media. Which one of the two o/p pads of matroskademux will be linked against filesink? video_0 (or) audio_0?
We Can solve by using Caps filters.

[gst-launch-1.0 SoupHttpSrc location=https://gstreamer---480p.webm ! matroskademux ! video/x-vp8 ! matroskamux ! filesink location=bintel_video.mkv]

⊛ Cap-filters behaves like pass-through element which does nothing & only accepts media with given Caps, Eg: we added video/x-vp8 Caps filter to specify that we are interested in o/p of matroskademux which Can produce this kind of video.

Examples of gst-launch-1.0 :-

gst-launch-1.0 playbin uri=https://gstreamer---webm

fully operation playback

gst-launch-1.0 souphttpsrc location=https://gstreamer.webm ! matroskademux name=d ! queue ! vp8dec ! videoconvert ! autovideosink d. ! queue ! vorbisdec ! audioconvert ! audioresample ! autoaudiosink.

gst-inspect-1.0 :-

Eg: gst-inspect-1.0 vp8dec

gst-discoverer-1.0 :-

Eg: gst-discoverer-1.0 https://gstreamer.freedesktop.org/data/sintel_trailer-480p.webm

Handy elements :-

Bins :- → which ~~does~~ can be treated as single element & they take care of ~~installing~~ instantiating all the necessary internal pipeline to accomplish their task.

playbin → It manages all aspects of media playback, from source to display passing through demuxing & decoding.

uridecodebin → This element decodes data from URI to raw media. It selects source element that can handle the given URI scheme & connects it to "decodebin" element. It acts like demuxer, so it offers as many source pads as streams are found in media.

decodebin :- → This element automatically constructs decoding pipeline using available decoders & demuxers via auto-plugging until raw media is obtained.

File Input/output :- filesrc :-

This element reads a local file & produces media with "ANY" caps. If you want to obtain the correct caps for the media, explore the stream by using typefind element or by setting typefind property of filesrc to TRUE.

filesink :- This element writes to file all the media it receives. use "location" property to specify the file name.

Eg: gst-launch-1.0 audiotestsrc ! vorbisenc ! oggmux ! filesink location=test.ogg.

Network :-

souphttpsrc :- → This element receives data as client over the network via HTTP using "libsoup" library. Set URL to retrieve through "location" property.

Eg:
gst-launch-1.0 souphttpsrc location=https://gstreamer.... webm

Test media generation:-

Is These elements are very useful to check if other parts of pipeline are working, by replacing source by one of these test sources which are guaranteed to work.

① videotestsrc → This element produces video pattern. use it to test videopipelines.

Eg: gst-launch-1.0 videotestsrc ! videoconvert ! autovideosink

② audiotestsrc → This element produces an audio wave.

Eg: gst-launch-1.0 audiotestsrc ! audioconvert ! autoaudiosink.

Audio adaptors:-

① audioconvert:- → Element converts raw audio buffers between various possible formats.

Eg: gst-launch-1.0 audiotestsrc ! audioconvert ! autoaudiosink

② audioresample:- This element resamples raw audio buffers to different sampling rates using configurable windowing function.

gst-launch-1.0 uridecodebin uri=http:---webm ! audioresample! audio/x-raw, rate=4000 ! audioconvert ! autoaudiosink

multithreading:-

① queue: perform two tasks

→ Data is queued until selected limit is reached.

→ queue creates new thread on source pad to decouple the processing on sink & source pads.

② tee → split data to multiple pads.

Eg: when capturing video where video is shown on screen & also encoded and written to a file.

Eg: playing music & hooking up visualization module.

gst-launch-1.0 audiotestsrc ! tee name=t ! queue ! audioconvert ! autoaudiosink t. ! queue ! wavescope ! videoconvert ! autovideosink

Capabilities:-

① Caps-filter:-

Eg: gst-launch-1.0 videotestsrc ! video/x-raw, format=GRAY8 ! videoconvert ! autovideosink.

Voice application software

VoiceSeeker:-

→ a library providing high resolution beamforming and multichan... Acoustic Echo Cancellation (AEC).

→ VoiceSeeker is in charge of removing noise from audio so that Voicespot only has to focus on detecting wake-word. If wake word is detected, voicespot will tell VoiceSeeker where to focus it's attention.

Overview:-

→ It is removing unwanted noise from audio that is being captured by microphones. The library allows user to clean the microphone's input, leaving the signal with just artifacts of human voice.

→ NXP provides wrapper around the library to make the experience with library more friendly and easy to use. This wrapper is compiled and built as shared library.

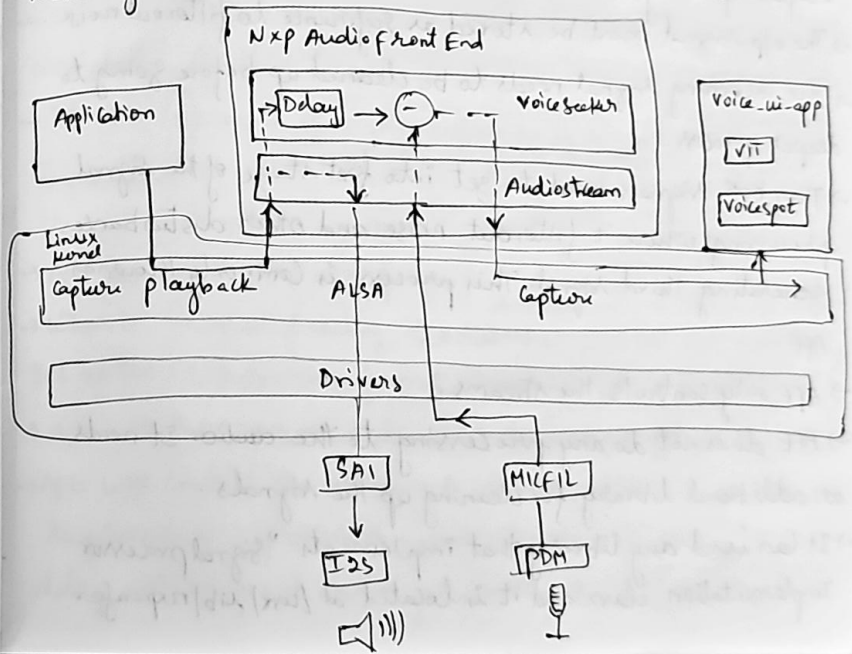
→ AFE is another wrapper NXP that allows to select the desire engine for voice processing at the runtime. For example, Selecting VoiceSeeker (or) Conversa.

→ Voicespot can be found, waiting for output of VoiceSeeker to detect the spoken word. The spoken word can be either wake-word (or) the voice-command itself. If wake word is found on the stream, Voicespot can notify other applications (VIT)

Architecture :-

→ VoiceSeeker and Voicespot were designed to work independently from each other. However it is recommended to use together to achieve best behavior of each library.

→ If Voice-seeker is working alone, we might want to implement mechanism to notify VoiceSeeker when wake word is detected, to reach the best performance on beamforming. The reason to do so is that each library is running on separate process and VoiceSeeker needs some feedback to be able to adjust the incoming data.



→ Voiceseeker (which is running on AFE) finishes its process Voicespot will be notified to look for wakeword in another different thread. Finally if wake word is found, Voicespot can notify third application for processing of the voice command and will give some feedback to voiceseeker to adjust beamforming.

NXP AFE:-

→ AFE was created as a solution for voice assistant applications, where we need to be able to control audio inputs and outputs of the system.
→ The o/p signal must be stored as reference to filter out noise.
→ The incoming signal needs to be cleaned up before going to deeper process.
→ Then both signals need to get into first stage of the signal processing. where it filters out noise and other disturbances, generating third signal. This process is commonly known as AFE.
→ AFE only controls the streams.
→ AFE does not do any processing to the audio. It needs an additional library for cleaning up the signals.
→ It can load any library that implements "signal processor Implementation" class and it is located at /usr/lib/nxp-afe.
→ If the library follows these rules, AFE would be able to load it and work with it.
→ The way of telling AFE which Engine to use is by giving its name as an argument.
Eg: ./afe libdummy
Eg: ./afe libvoiceseekerlight.so

VoiceSeeker:- Full name is voiceseeker light (VSL)
→ Detect from which microphone the spoken word comes from & focus on that microphone (Beamforming).
→ Remove ambient background noise & give clean version of the word being spelled (AEC).
→ It is an example of implementation of signal processor Implementation class, which has all its methods and it built as share object allowing AFE to load if it required.
→ AFE can load VSL with following command: afe libvoiceseekerlight.
→ When it finished loading, it voiceseeker will tell AFE how to set up the ALSA devices, format, channels, size of the buffers, etc.
→ AFE will then warm up devices & then start streaming.
→ AFE will control the streams & when there is enough data to process, VSL process is called. Inside VSL process the delay will be fixed before processing the data and then

it will clean the audio.

→ When VSL finish processing the data, it will then write data to the output buffer which AFE will be in charge to write it to the loopback interface where Voicespot will be listening.

Config.ini file:-

Voiceseeker reads a file to to be properly configured. This file is called Config.ini & it is located on /usr/unit-tests/nxp-afe/

Voicespot:-

Voiceseeker notifies when there is chunk of data available and Voicespot will tell Voiceseeker if there is speech on that incoming data so Voiceseeker can focus its attention on that channel and readjust its filter on the run. This program is called Voice_ui_app, since Voicespot is an object there must be main process that controls the workflow of the data and creates an instance of the library.

Model:-

Voicespot uses machine learning (ML) model to detect the Wake word which had been trained with best Qualities and environment.

File	Location	Description
LibVoiceseekerlight.so	/usr/lib/nxp-afe/	Voiceseeker wrapper. Loaded by AFE at runtime.
afe	/unit-tests/nxp-afe/	AFE binary
Voice_ui_app	" "	Main app
asound-conf-imx8mp	" "	ALSA Configuration file
Config.ini	" "	Voiceseeker, Voicespot.
HeyNXP_1-params.bin	" "	Nxp parameters for Wake word.
HeyNXP-en-US-1.bin	" "	NXP Wake-word model
Snd-aloop	/lib/modules/_version_ /Kernel/Sound/drivers	Driver.