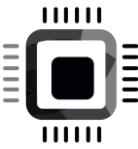




LINUX DEVICE DRIVER PROGRAMMING (LDD1)

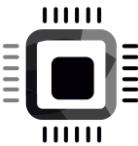
FASTBIT EMBEDDED BRAIN ACADEMY

www.fastbitlab.com



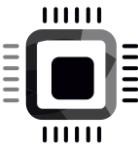
Copyright notice

Linux device driver programming power point presentation by **BHARATI SOFTWARE** is licensed under [CC BY-SA 4.0](#). To view a copy of this license, visit <https://creativecommons.org/licenses/by-sa/4.0>



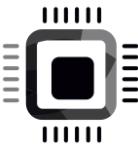
Links

- For the full video course please visit
 - <https://www.udemy.com/course/linux-device-driver-programming-using-beaglebone-black/>
- Course repository
 - <https://github.com/niekiran/linux-device-driver-1>
- Explore all Fastbit EBA courses
 - <http://fastbitlab.com/course1/>
- For suggestions and feedback
 - contact@fastbitlab.com



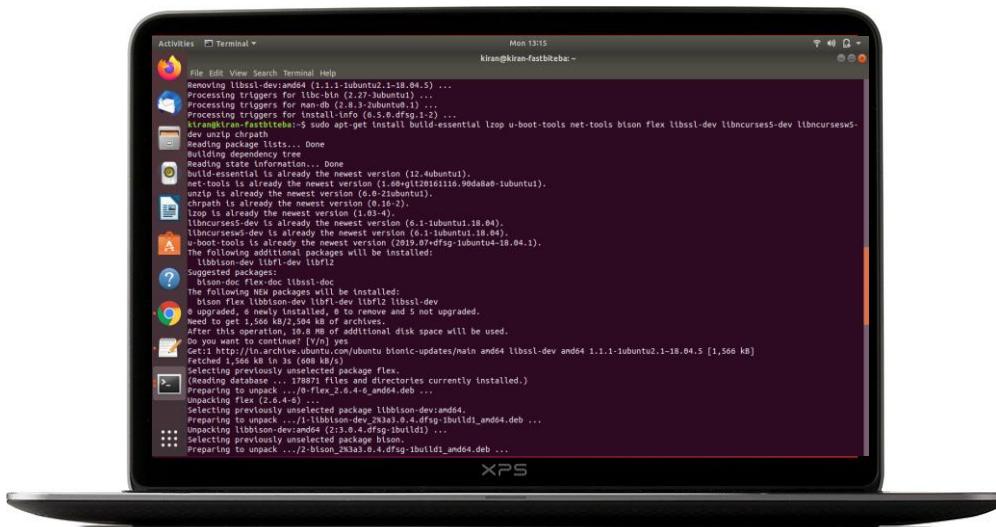
Social media

- Join our Facebook private group for technical discussion
 - <https://www.facebook.com/groups/fastbiteba/>
- LinkedIn
 - <https://www.linkedin.com/company/fastbiteba/>
- Facebook
 - <https://www.facebook.com/fastbiteba/>
- YouTube
 - <https://www.youtube.com/channel/UCa1REBV9hyrzGp2mjJCagBg>
- Twitter
 - <https://twitter.com/fastbiteba>

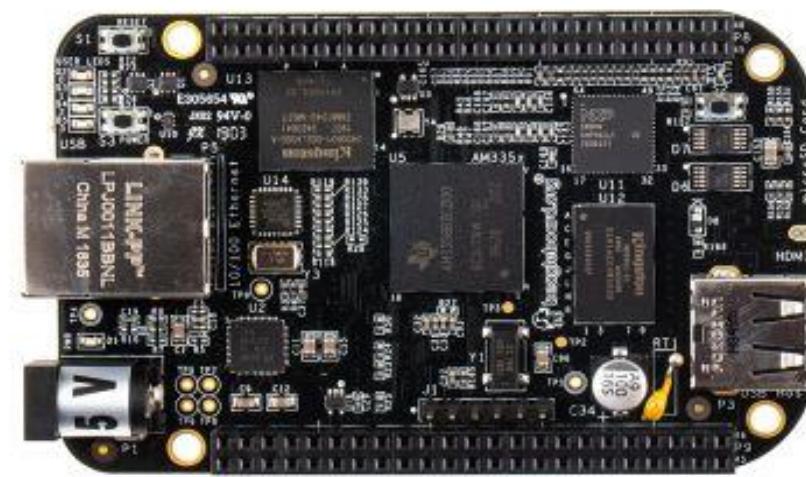


HOST and Target

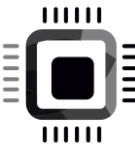
Host (PC running Ubuntu 18.04 OS 32/64bit)



Target Beaglebone Black Rev A5



Installing required packages on the host

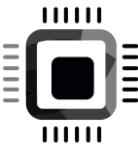


1. Install these packages on the HOST

`sudo apt-get update`

`sudo apt-get install build-essential lzop u-boot-tools net-tools bison flex
libssl-dev libncurses5-dev libncursesw5-dev unzip chrpath xz-utils
minicom`

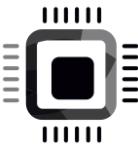
- These commands are given in the text file attached with this video



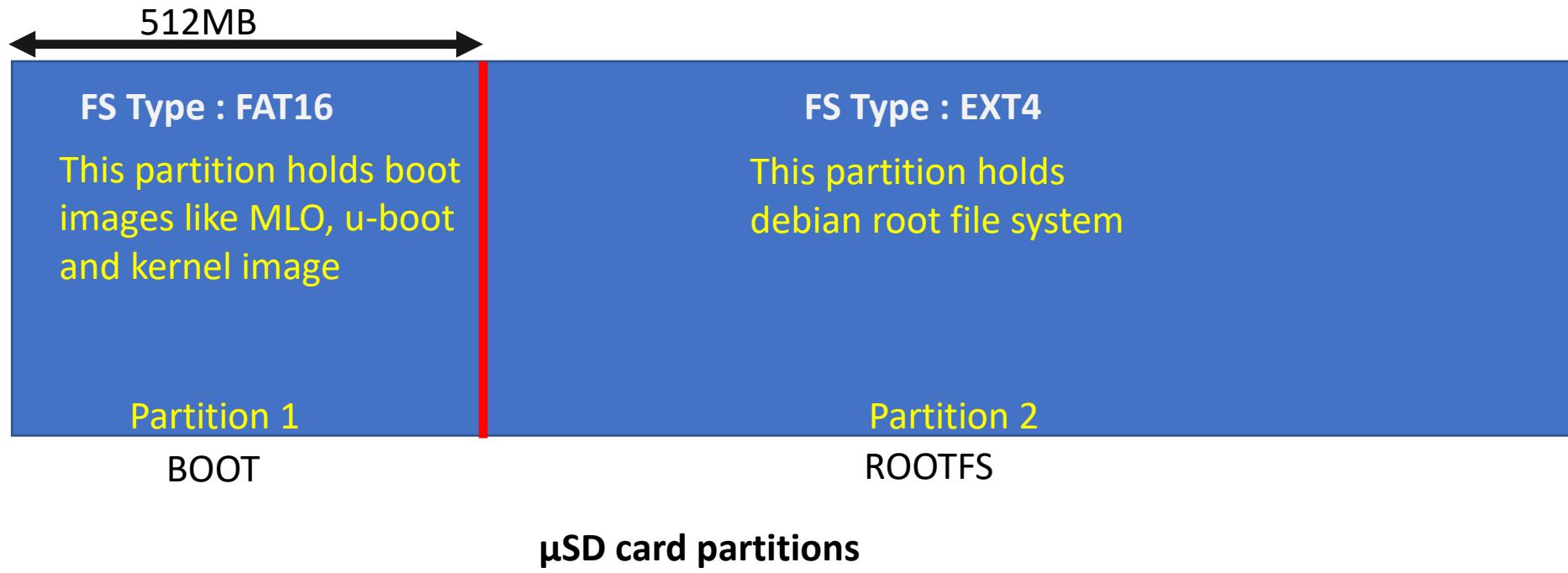
Workspace setup

- You may follow the below folder setup to work with this course

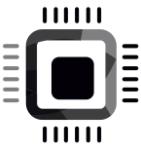
```
/home/kiran/workspace
.
└── ldd
    ├── custom_drivers
    ├── downloads
    ├── patches
    └── source
```



Download boot images and Root file system



In this course we are going to boot our board that is beaglebone black using sd card so to boot the board we need boot images as well as file system.
first we are going to boot using pre build images and then after that we going to update the kernel by cross compiling the kernel source which we are going to clone from repository.



2. Download boot images and Root file system

- Download the boot images file attached with this lecture **for boot images**
- Visit this link and download the latest Debian image for beglebone black **for root file system**
- <https://beagleboard.org/latest-images>

beagleboard.org/latest-images

Download the latest firmware for your BeagleBoard, BeagleBoard-xM, BeagleBoard-X15, BeagleBone, BeagleBone Black, BeagleBone Black Wireless, BeagleBone AI, BeagleBone Blue, SeeedStudio BeagleBone Green, SeeedStudio BeagleBone Green Wireless, SanCloud BeagleBone Enhanced, element14 BeagleBone Black Industrial, Arrow BeagleBone Black Industrial, Mentor BeagleBone uSomIQ, Neuromeka BeagleBone Air, or PocketBeagle

See the [Getting Started guide](#) and the [community wiki page](#) for hints on loading these images.

Recommended Debian Images

Stretch LXQT (with graphical desktop) for [BeagleBoard-X15](#) and [BeagleBone AI](#) via microSD card

► [Debian 9.9 2019-08-03 4GB SD LXQT](#)
image for [BeagleBoard-X15](#) and [BeagleBone AI](#) - more info - sha256sum: 9aa4bc0d20d941cd5b0e89eecfd1f415a645f8419b865ee8274d5757926991c5

Stretch IoT (without graphical desktop) for [BeagleBone](#) and [PocketBeagle](#) via microSD card

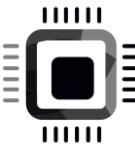
► [Debian 9.9 2019-08-03 4GB SD IoT](#)
image for [PocketBeagle](#), [BeagleBone](#), [BeagleBone Black](#), [BeagleBone Black Wireless](#), [BeagleBone Blue](#), [SeeedStudio BeagleBone Green](#), [SeeedStudio BeagleBone Green Wireless](#), [SanCloud BeagleBone Enhanced](#), [element14 BeagleBone Black Industrial](#), [Arrow BeagleBone Black Industrial](#) and [Mentor BeagleBone uSomIQ](#) - more info - sha256sum: 0c2afa722979b5ebc835c6b7063584edc959a70fc6cf64f39714869a0ba7fc99

Stretch LXQT (with graphical desktop) for [BeagleBone](#) via microSD card

► [Debian 9.9 2019-08-03 4GB SD LXQT](#)
image for [BeagleBone](#), [BeagleBone Black](#), [BeagleBone Black Wireless](#), [BeagleBone Blue](#), [SeeedStudio BeagleBone Green](#), [SeeedStudio BeagleBone Green Wireless](#), [SanCloud BeagleBone Enhanced](#), [element14 BeagleBone Black Industrial](#), [Arrow BeagleBone Black Industrial](#) and [Mentor BeagleBone uSomIQ](#) - more info - sha256sum: ff4af8aa309b7842de0759cf1f732c09a80b4ade1e89157a442f1bf93a359318

Stretch for [BeagleBoard-X15](#) via microSD card

► [Debian 9.5 2018-10-07 4GB SD LXQT](#) image for [BeagleBoard-X15](#) - more info - sha256sum: db3f2b6a1305de715d33c1cb330ae51befef15379d4bd797ed3c1aa685b9729d



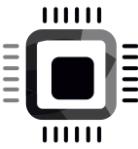
3. Cross compiler and toolchain download

- To cross-compile Linux kernel, Linux application, and kernel modules to ARM Cortex Ax architecture, we need a cross compiler.
- The SOC AM335x from TI, is based on ARM Cortex A8 processor of ARMv7 architecture
- Download the cross compiler from here
- <https://www.linaro.org/downloads/>
- Link is given in the resource section of this lecture

<https://snapshots.linaro.org/gnu-toolchain/11.3-2022.06-1/arm-linux-gnueabihf/>

download this :- [gcc-linaro-11.3.1-2022.06-aarch64_arm-linux-gnueabihf.tar.xz](https://snapshots.linaro.org/gnu-toolchain/11.3-2022.06-1/arm-linux-gnueabihf/gcc-linaro-11.3.1-2022.06-aarch64_arm-linux-gnueabihf.tar.xz)

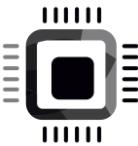
A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.



4. Install gparted application

- This is a graphical user interface application which we will use to partition the Micro SD card

this software installed from software center of linux, using gui based interface in the lecture.

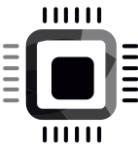


Tool-chain download

this is cross compiler only, of previos slide

Name	Last modified	Size	License
Parent Directory			
gcc-linaro-7.5.0-2019.12-i686-mingw32_arm-linux-gnueabihf.tar.xz	05-Dec-2019 10:01	341.6M	open
gcc-linaro-7.5.0-2019.12-i686-mingw32_arm-linux-gnueabihf.tar.xz.asc	05-Dec-2019 10:02	99	open
gcc-linaro-7.5.0-2019.12-i686_arm-linux-gnueabihf.tar.xz	05-Dec-2019 10:02	104.0M	open
gcc-linaro-7.5.0-2019.12-i686_arm-linux-gnueabihf.tar.xz.asc	05-Dec-2019 10:02	91	open
gcc-linaro-7.5.0-2019.12-linux-manifest.txt	05-Dec-2019 10:02	11.1K	open
gcc-linaro-7.5.0-2019.12-win32-manifest.txt	05-Dec-2019 10:02	11.1K	open
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf.tar.xz	05-Dec-2019 10:02	104.6M	open
gcc-linaro-7.5.0-2019.12-x86_64_arm-linux-gnueabihf.tar.xz.asc	05-Dec-2019 10:02	93	open
runtime-gcc-linaro-7.5.0-2019.12-arm-linux-gnueabihf.tar.xz	05-Dec-2019 10:02	6.4M	open
runtime-gcc-linaro-7.5.0-2019.12-arm-linux-gnueabihf.tar.xz.asc	05-Dec-2019 10:02	94	open
sysroot-glibc-linaro-2.25-2019.12-arm-linux-gnueabihf.tar.xz	05-Dec-2019 10:02	40.9M	open
sysroot-glibc-linaro-2.25-2019.12-arm-linux-gnueabihf.tar.xz.asc	05-Dec-2019 10:02	157	open

```
kiran@kiran-fastbiteba:~/workspace$ uname -a
Linux kiran-fastbiteba 5.3.0-40-generic #32~18.04.1-Ubuntu SMP Mon Feb 3 14:05:59 UTC
2020 x86_64 x86_64 x86_64 GNU/Linux
```



Tool-chain PATH settings

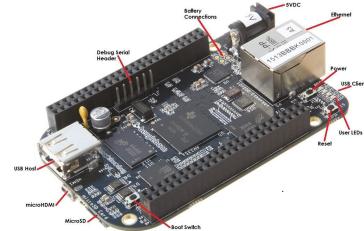
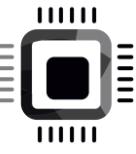
- Add the tool chain binary path to home directory **.bashrc** file

Steps:

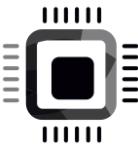
1. Go to your home directory
2. Open **.bashrc** using vim or gedit text editor
3. Copy the below export command with path information to .bashrc file
export PATH=\$PATH:<path_to_tool_chain_binaries>
4. Save and close

or simply do

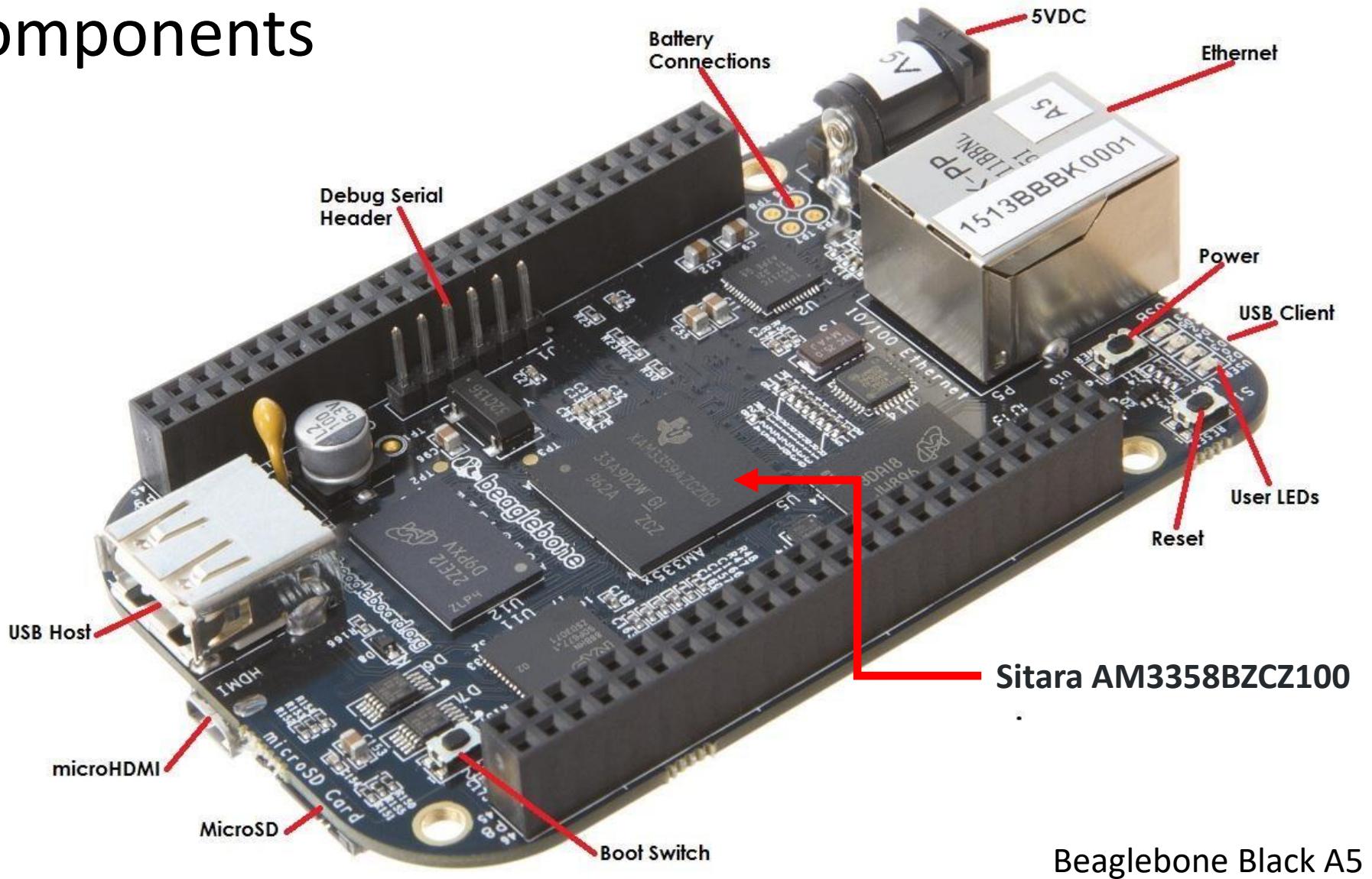
```
echo "export PATH=$PATH:<path_to_tool_chain_binaries>" > ~/.bashrc
```

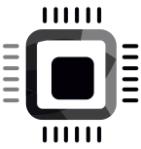


Target preparation

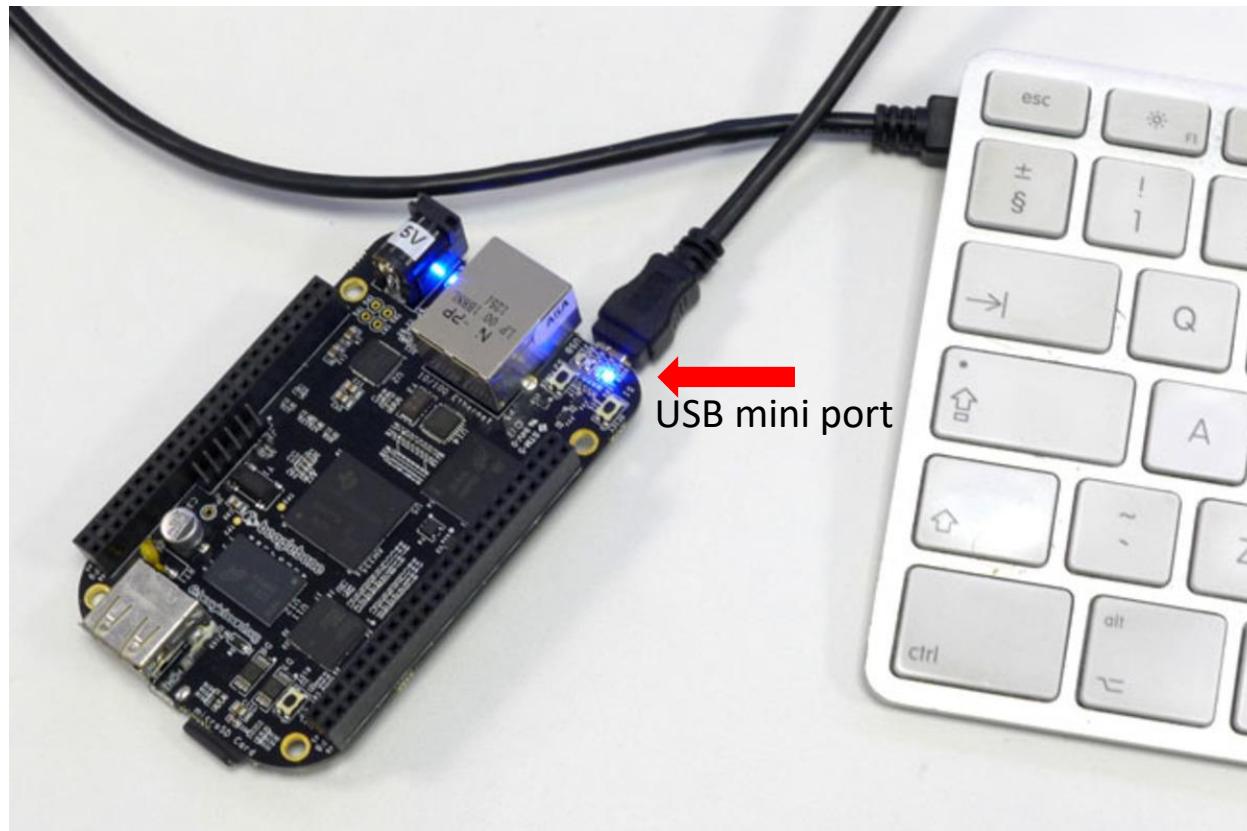


Board components



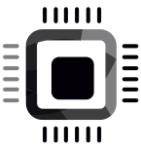


Powering

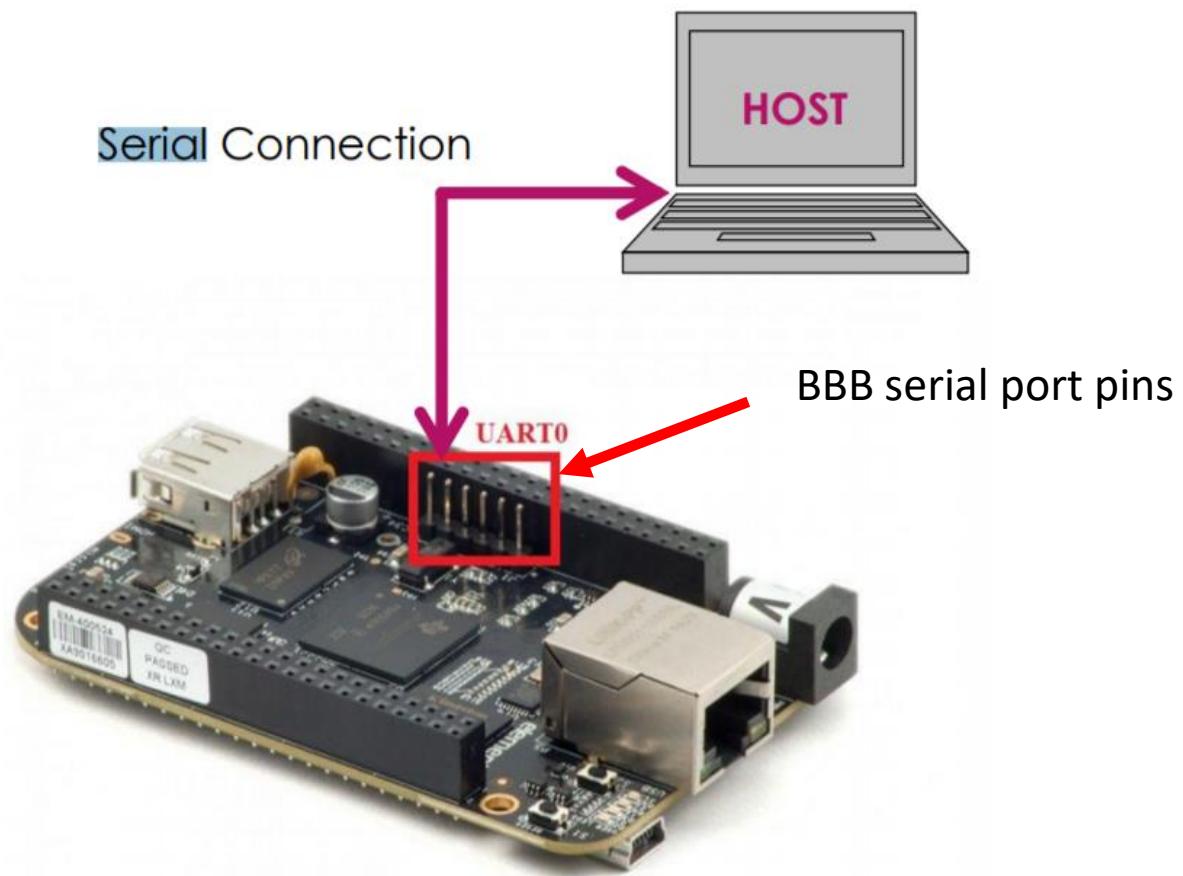


USB Cable, Type A Plug to Mini Type B Plug

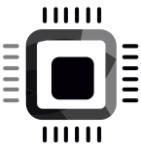




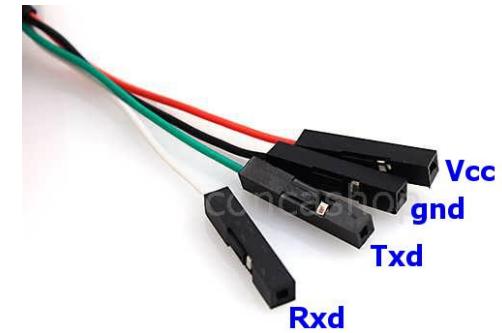
Serial debug port connection



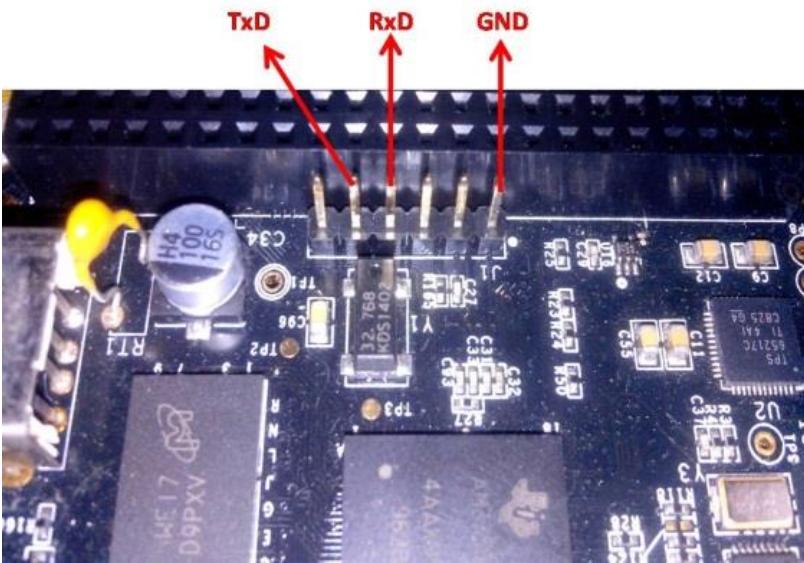
USB to TTL Serial Cable

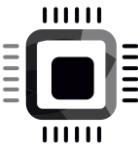


USB to TTL serial converter dongle



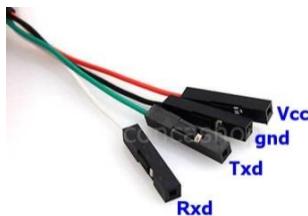
USB to TTL serial converter cable





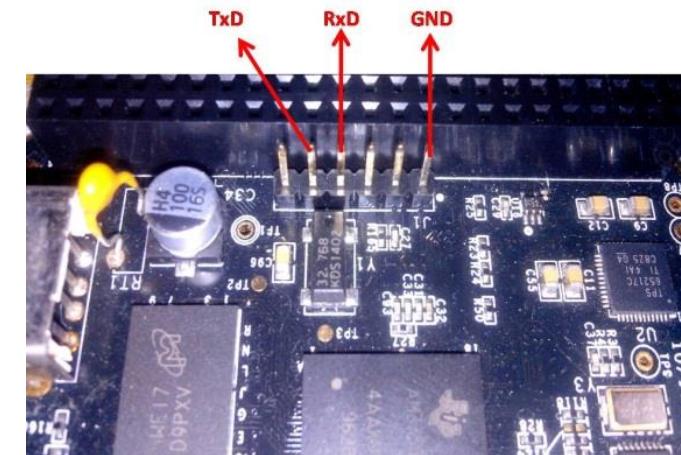
USB to Serial TTL cable pins

VCC
TXD
RXD
GND

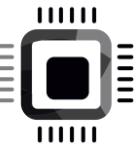


BBB J1 header pin outs

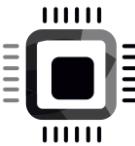
GND (Pin 1)
RXD (Pin 4)
TXD (Pin 5)



Important Note : RXD of the cable must be connected to TXD of the BBB serial header (pin 5)
TXD of the cable must be connected to RXD of the BBB serial header(pin 4)
GND of the cable must be connected to GND of the BBB serial header(pin 1)



Boot sequence of BBB



BOOT sequence

Boot button(S2) NOT pressed
during power up

- 1. MMC1(eMMC) internal memory
- 2. MMC0 (μ SD)
- 3. UART0
- 4. USBO

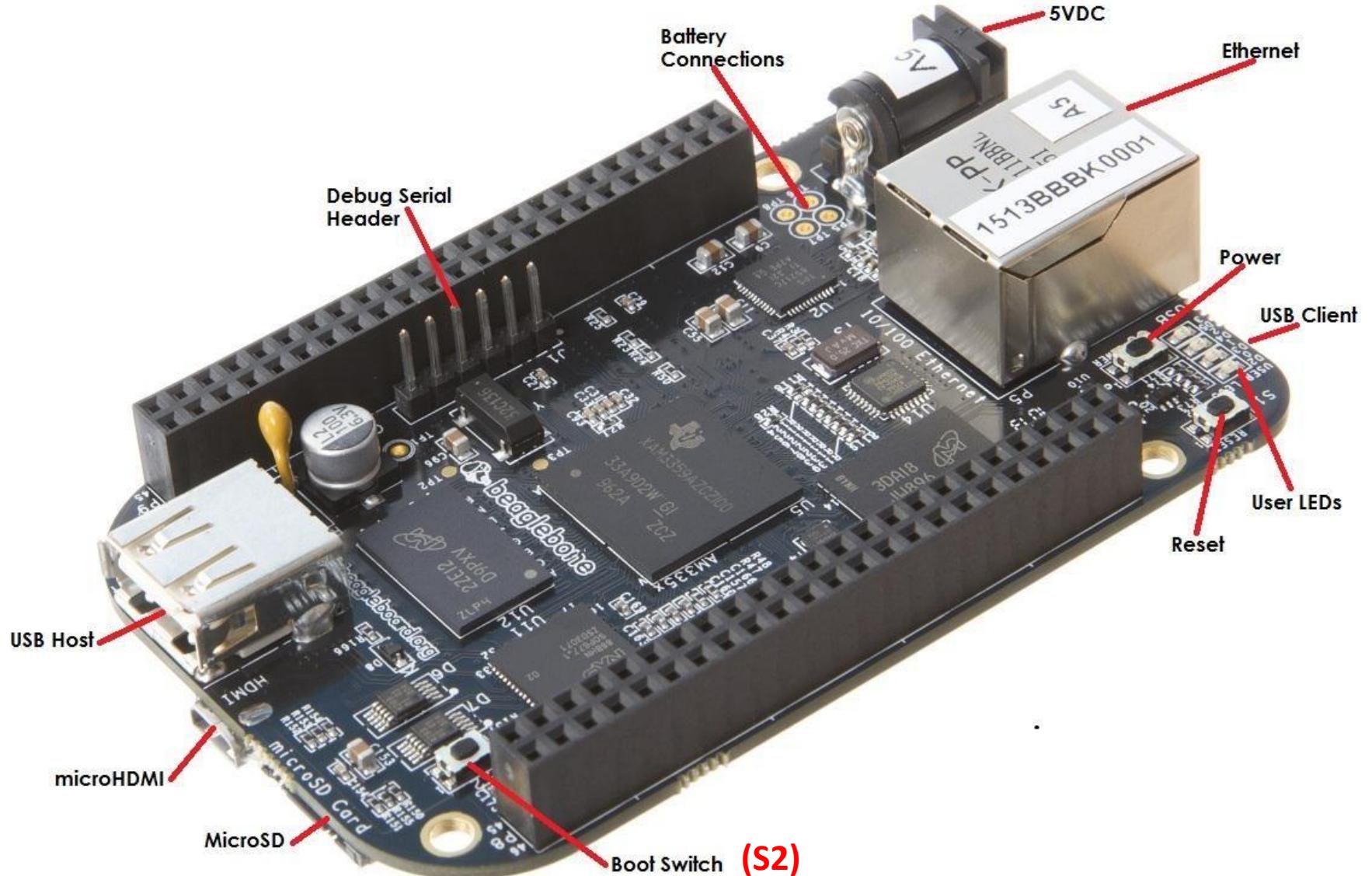
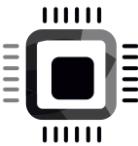
AM335X Boot Sequence

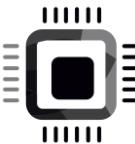
actually the boot button affects register called SYSBOOT

Boot button(S2) pressed
during power up

- 1. SPI0
- 2. MMC0 external usb card
- 3. USBO
- 4. UART0

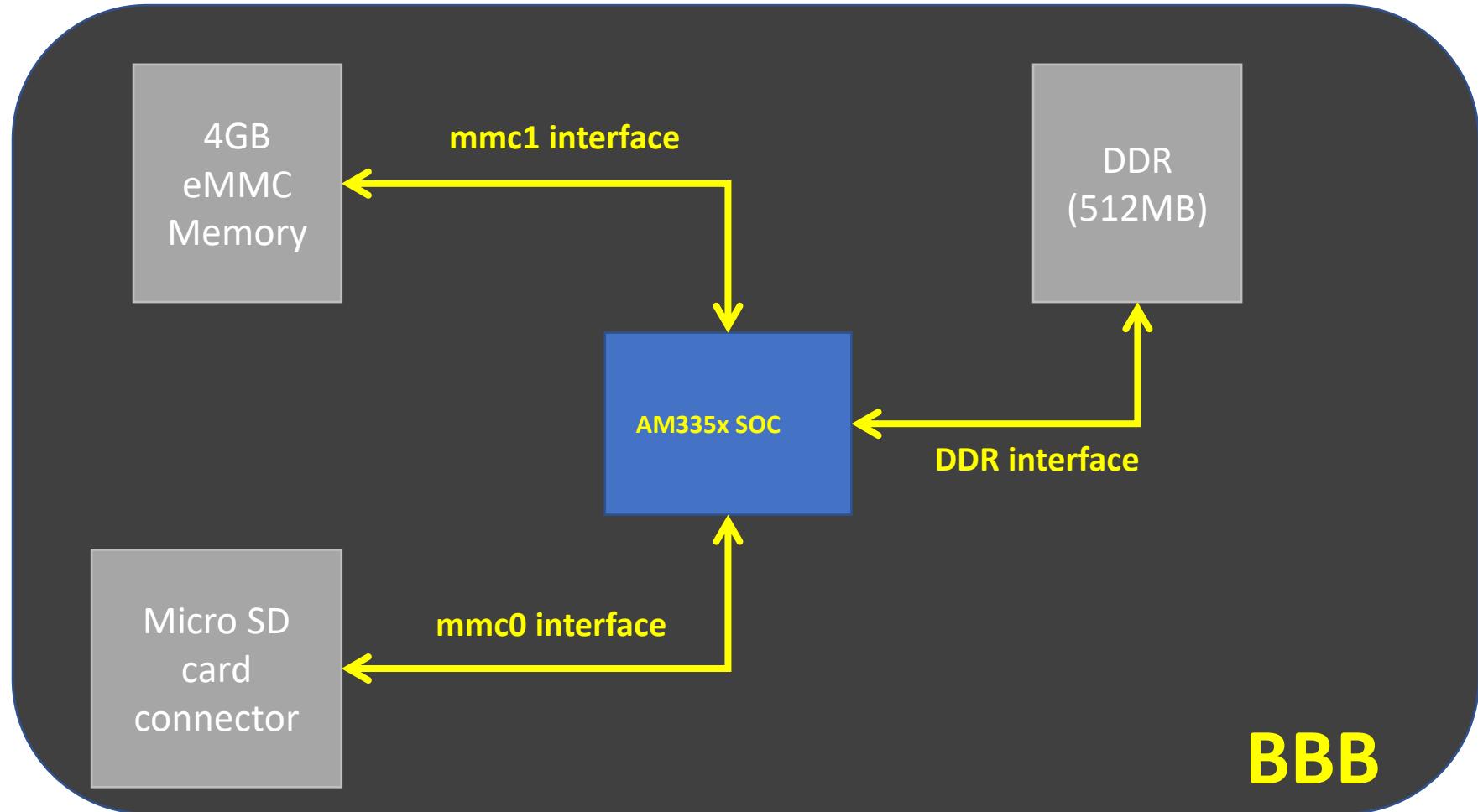
BOOT sequence

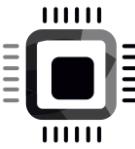




Usage of Power button , reset button and boot button

- **Power button:** By press and hold this button for 10 to 20 seconds, you can power down the board. Once you power down the board, gently pressing this button one time will again power up the board. Instead of connecting and disconnecting power sources to your board now and then, you can use this button to power down and power up.
- **Reset button:** Pressing this button resets the board. Note that the boot sequence is not affected by the reset action.
- **Boot button:** you can use this button to change the boot sequence during power-up of the board.





Boot button(S2) NOT pressed
during power up

SYSBOOT[4:0]

1. MMC1(eMMC)
2. MMC0 (μ SD)
3. UART0
4. USBO

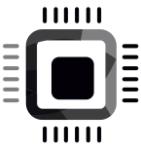
BOOT sequence

Boot button(S2) pressed
during power up

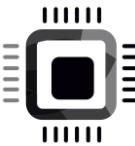
1. SPI0
2. MMC0(μ SD)
3. USBO
4. UART0

BOOT sequence

AM335X Boot Sequence

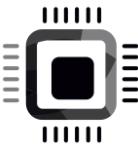


Preparing μSD card for SD boot



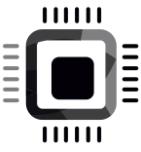
μSD card preparation

1. You can use 8/16/32 GB μSD card
2. Connect the μSD card to PC using card reader
3. Launch the **gparted application**
4. Make 2 partitions (fat16 and ext4)
5. Configure the boot, lba flags on the boot partition
6. Copy boot images on FAT16 partition (this is boot partition)
7. Copy debian root file system on ext4 partition
8. Unmount and remove the μSD card from the PC
9. Insert the μSD card into BBB μSD card slot
10. Boot from SD card interface (mmc0 interface)

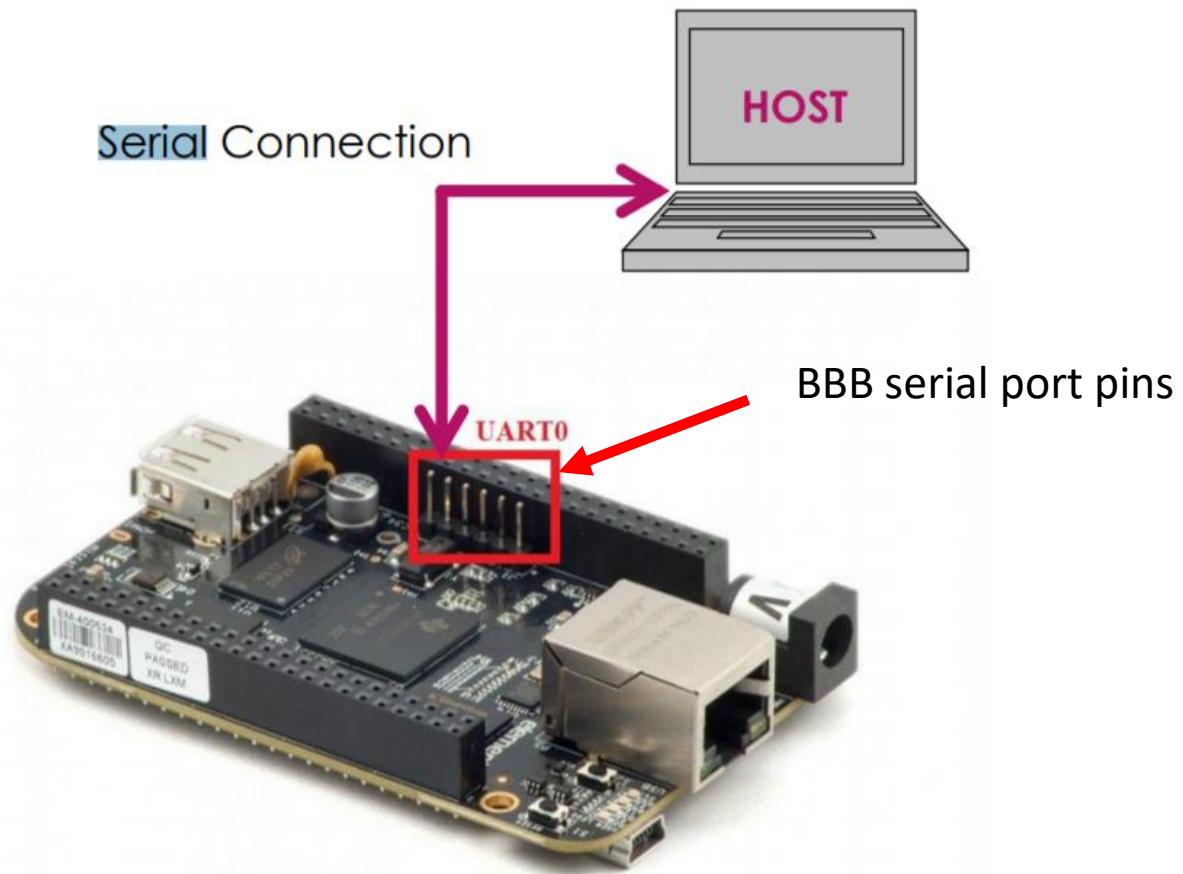


Booting from µSD card interface

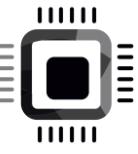
1. Make sure that BBB is not powerd up
2. Connect serial debug cable between BBB and Host
3. Insert the SD card to BBB
4. Give power to the board using mini usb cable
5. Press and hold the boot button (S2)
6. Press and hold the power button (S3) until the blue LED turns off and turns ON again. (if blue LED doesn't turn ON, gently press the power button)
7. Release the S2 button after 2 to 5 seconds



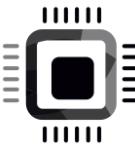
Serial debug port connection



USB to TTL Serial Cable

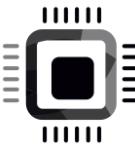


Linux kernel image update



Steps to update the linux kernel image

- Clone the latest stable kernel source from BBB official github
 - <https://github.com/beagleboard/linux>
- Compile and generate the kernel image
- Update the SD card with new kernel image and boot again



Kernel compilation steps

STEP 1:

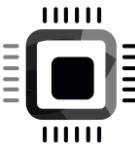
`/*removes all the temporary folder, object files, images generated during the previous build. This step also deletes the .config file if created previously */`

`make ARCH=arm distclean`

STEP 2:

`/*creates a .config file by using default config file given by the vendor */`

`make ARCH=arm bb.org_defconfig`



Kernel compilation steps

STEP 3:

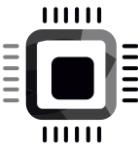
*/*This step is optional. Run this command only if you want to change some kernel settings before compilation */*

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig
```

STEP 4:

*/*Kernel source code compilation. This stage creates a kernel image "ulmage" also all the device tree source files will be compiled, and dtbs will be generated */*

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- ulmage dtbs LOADADDR=0x80008000 -j4
```



Kernel compilation steps

STEP 5:

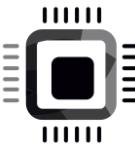
```
/*This step builds and generates in-tree loadable(M) kernel modules(.ko) */
```

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- modules -j4
```

STEP 6:

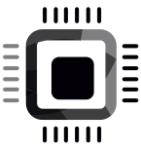
```
/* This step installs all the generated .ko files in the default path of the computer  
(/lib/modules/<kernel_ver>) */
```

```
sudo make ARCH=arm modules_install
```

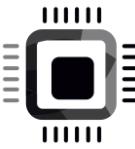


Update kernel image and kernel modules in SD card

- Copy ulmage to board and then update the boot partition of the SD card
- Copy newly installed **4.14.108** folder to board's **/lib/modules/** folder
- Reset the board (you should see BBB boots with newly updated kernel image)

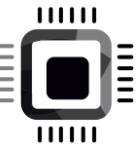


Enabling internet over USB



Internet over USB

- Beaglebone board can communicate to the internet over the USB cable by sharing your PC's internet connection.
- You need not to use a separate ethernet cable to connect your board to internet.
- The required drivers are enabled by default in the kernel and loaded when Linux boots on the BBB
- But you must enable internet sharing on your HOST

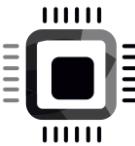


Internet over USB

- Target settings :
 1. Add name server address in : [/etc/resolv.conf](#)
 2. Add name server address in : [/etc/network/interfaces](#)

```
iface usb0 inet static
    address 192.168.7.2
    netmask 255.255.255.252
    network 192.168.7.0
    gateway 192.168.7.1
    dns-nameservers 8.8.8.8
    dns-nameservers 8.8.4.4
```

1. Add default gateway address by running the below command
 - [route add default gw 192.168.7.1](#) (Using PC as default gateway)



Internet over USB

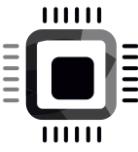
Host settings :

Run the below commands

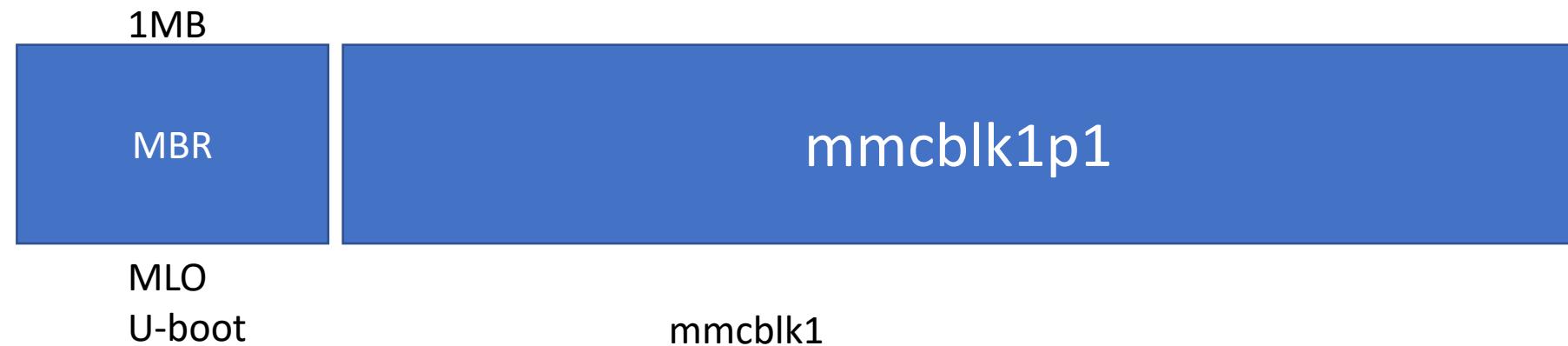
```
sudo iptables --table nat --append POSTROUTING --out-interface wlp2s0 -j MASQUERADE  
sudo iptables --append FORWARD --in-interface wlp2s0 -j ACCEPT  
sudo echo 1 > /proc/sys/net/ipv4/ip_forward
```

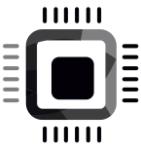
Network interface name . You must
use the name as listed by the
command 'ifconfig'

***If you reboot your machine, again you must run these commands.
So, its better if you create a small script and execute when your machine reboots.***

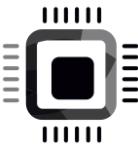


BBB eMMC partitions after using eMMC flasher Debian image



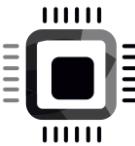


Hello world kernel module



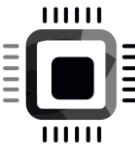
Linux Kernel module (LKM)

- Let's learn how to write a simple hello world kernel module
- Compiling the kernel module using kbuild
- Transferring kernel module to BBB, loading, and unloading

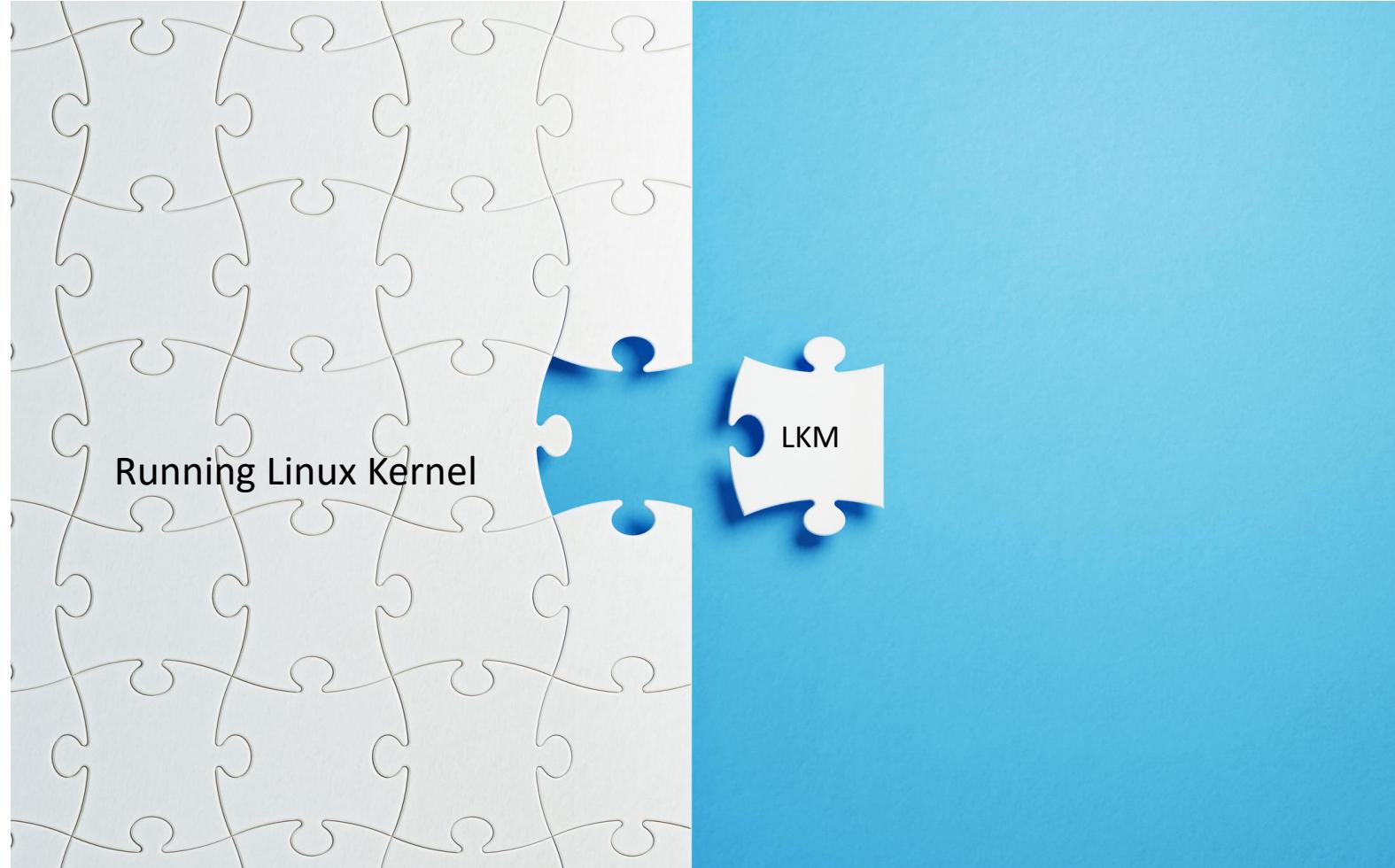


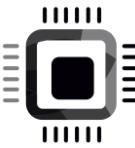
Linux Kernel module (LKM)

- Linux supports dynamic insertion and removal of code from the kernel while the system is up and running. The code what we add and remove at run time is called a kernel module
- Once the LKM is loaded into the Linux kernel, you can start using new features and functionalities exposed by the kernel module without even restarting the device.
- LKM dynamically extends the functionality of the kernel by introducing new features to the kernel such as security, device drivers, file system drivers, system calls etc. (modular approach)
- Support for LKM allows your embedded Linux systems to have only minimal base kernel image(less runtime storage) and optional device drivers and other features are supplied on demand via module insertion
- Example: when a hot-pluggable new device is inserted the device driver which is an LKM gets loaded automatically to the kernel



A Linux Kernel Module (often referred to as a kernel module or just module) is a piece of code that can be dynamically loaded and unloaded into the Linux kernel. These modules provide additional functionality to the kernel without requiring you to recompile or rebuild the entire kernel. Kernel modules are commonly used to add device drivers, filesystems, and other kernel-level features to a running Linux system.





Static and dynamic LKMs

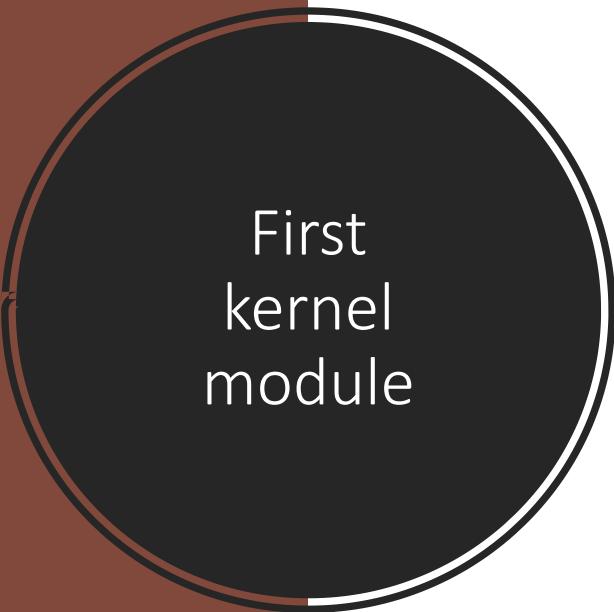
- **Static(y)**

➤ When you build a Linux kernel, you can make your module statically linked to the kernel image (module becomes part of the final Linux kernel image). This method increases the size of the final Linux kernel image. Since the module is ‘built-in’ into the Linux kernel image, you can not ‘unload’ the module. It occupies the memory permanently during run time

- **Dynamic(m)**

➤ When you build a Linux kernel, these modules are NOT built into the final kernel image, and rather there are compiled and linked separately to produce .ko files. You can dynamically load and unload these modules from the kernel using user space programs such as insmod, modprobe , rmmod.

So, when you’re building the kernel, you can either link modules directly into the kernel or build them as separate modules that can be loaded into the kernel at some other time.



First kernel module

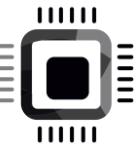
```
#include <linux/module.h>

/* This is module initialization entry point */
static int __init my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}

/* This is module clean-up entry point */
static void __exit my_kernel_module_exit(void)
{
    pr_info("Good bye World\n");
}

/* This is registration of above entry points with kernel */
module_init(my_kernel_module_init);
module_exit(my_kernel_module_exit);

/* This is descriptive information about the module */
MODULE_LICENSE("GPL"); /*This module adhers to GPL licensing */
MODULE_AUTHOR("www.fastbitlab.com");
MODULE_DESCRIPTION("A kernel module to print some messages");
```



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

/* This is module initialization entry point */
static int __init my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}

/* This is module clean-up entry point */
static void __exit my_kernel_module_exit(void)
{
    pr_info("Good bye World\n");
}

/* This is registration of above entry points with kernel */
module_init(my_kernel_module_init);
module_exit(my_kernel_module_exit);

/* This is descriptive information about the module */
MODULE_LICENSE("GPL"); /*This module adhers to GPL licensing */
MODULE_AUTHOR("www.fastbitlab.com");
MODULE_DESCRIPTION("A kernel module to print some messages");
```

Header section

}

}

Your code

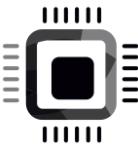
}

}

Registration

}

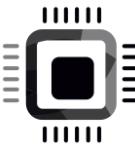
Module description



Header

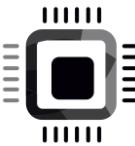
```
#include <linux/module.h>
```

Every kernel module should to include this header file



Kernel header vs user-space header

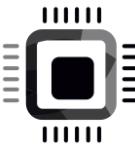
- Since you write a kernel module that is going to be executed in kernel space, you should be using kernel headers, never include any user-space library headers like C std library header files.
- No user space library is linked to the kernel module
- Most of the relevant kernel headers live in [linux_source_base/include/linux/](#)



Your code

```
/* This is module initialization entry point */
static int __init my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}

/* This is module clean-up entry point */
static void __exit my_kernel_module_exit(void)
{
    pr_info("Good bye World\n");
}
```



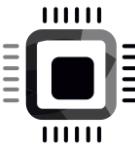
Your code

```
/* This is module initialization entry point */
int my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}
```

```
/* This is module clean-up entry point */
void my_kernel_module_exit(void)
{
    pr_info("Good bye World\n");
}
```

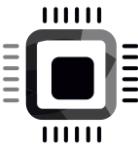
module initialization function

Module clean-up function



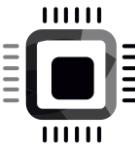
Module initialization function

- Prototype: `int fun_name(void);`
- Must return a value ; 0 for success, nonzero means module initialization failed. So the module will not get loaded in the kernel.
- This is an entry point to your module (like main). This function will get called during boot time in the case of static modules
- In the case of dynamic modules, this function will get called during module insertion
- There should be one module initialization entry point in the module



Module initialization function

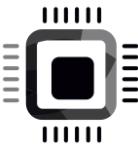
- You may do some initialization of devices
- Initialization of device private data structures
- Requesting memory dynamically for various kernel data structures and services
- Request for allocation of major-minor numbers
- Device file creation



Understanding the complete syntax.

```
/* This is module initialization entry point */
static int __init my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}
```

The module initialization function is module-specific and should never be called from other modules of the kernel. It should not provide any services or functionalities which may be requested by other modules. Hence it makes sense to make this function private using 'static' though it is optional.

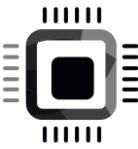


Module clean-up function

- Prototype

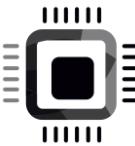
`void fun_name(void);`

- This is an entry point when the module is removed
- Since you can not remove static modules, clean-up function will get called only in the case of dynamic modules when it is removed using user space command such as `rmmmod`
- If you write a module and you are sure that it will always be statically linked with the kernel, then there is no need to implement this function.
- Even if your static module has a clean-up function, the kernel build system will remove it during the build process if there is an `__exit` marker.



Module clean-up function

- Typically, you must do exact reverse operation what you had done in the module init function. undoing init function.
- Free memory which are requested in init function
- De-init the devices or leave the device in the proper state



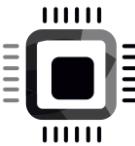
Understanding the complete syntax.

```
/* This is module initialization entry point */
static int __init my_kernel_module_init(void)
{
    /* kernel's printf */
    pr_info("Hello World !\n");
    return 0;
}

/* This is module clean-up entry point */
static void __exit my_kernel_module_exit(void)
{
    pr_info("Good bye World\n");
}
```

once the function finishes , then we do not need the function, as no one going to call it, so its better to remove the memory occupied by this function code.

thats why __init is a technique in which we push the code in .init section a separate section which a kernel can free later.



Function section attributes

- `__init`
- `__exit`

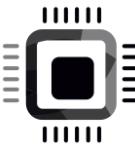
Compiler directive, which directs the compiler to keep data or code in an output section called “.init”

```
#define __init          __section(.init.text)
#define __initdata       __section(.init.data)
#define __initconst      __section(.init.rodata)
```

```
#define __exit         __section(.exit.text)
```

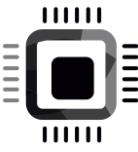
Compiler directive, which directs the compiler to keep data or code in an output section called “.exit”

when you use `__init` macro with any function then that function code will be placed at a section called `.init` in the final kernel image , same as for `__exit`



__init

- `__init` and `__exit` makes sense only for static modules (built-in modules)
- `__init` is a macro which will be translated into compiler directive, which instructs the compiler to put the code in `.init` section of the final ELF of linux kernel image.
- `.init` section will be freed from the memory by the kernel during boot time once all the initialization functions get executed.
- Since the built-in driver cannot be unloaded, its init function will not be called again until the next reboot, that's why there is no need to keep references to its init function anymore.
- so using `__init` macro is a technique, when used with a function, the kernel will free the code memory of that function after its execution.
- Similarly, you can use `__initdata` with variables that will be dropped after the initialization. `__initdata`, which works similarly to `__init` but for init variables rather than functions.

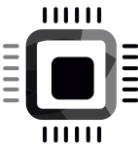


__exit

- You know that for built-in modules clean-up function is not required
- So, when you use the __exit macro with a clean-up function, the kernel build system will exclude those functions during the build process itself.

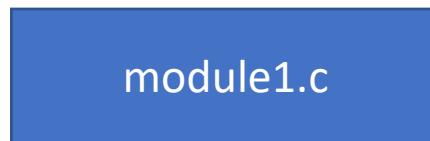
__exit act as a marker for the build system to drop, the clean up function in built processes itself

In Linux kernel modules, "cleanup functions" or "cleanup handlers" are functions that are used to perform cleanup and resource release tasks when a module is unloaded or removed from the kernel.

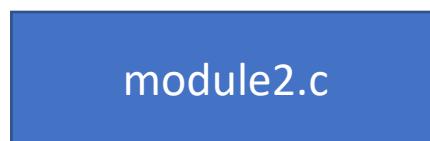


Static modules

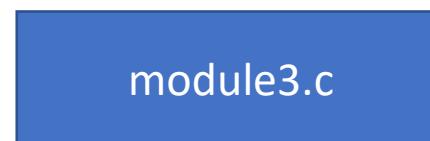
```
int __init m1_init_fun (void) { .... }
```



```
int __init m2_init_fun (void) { .... }
```

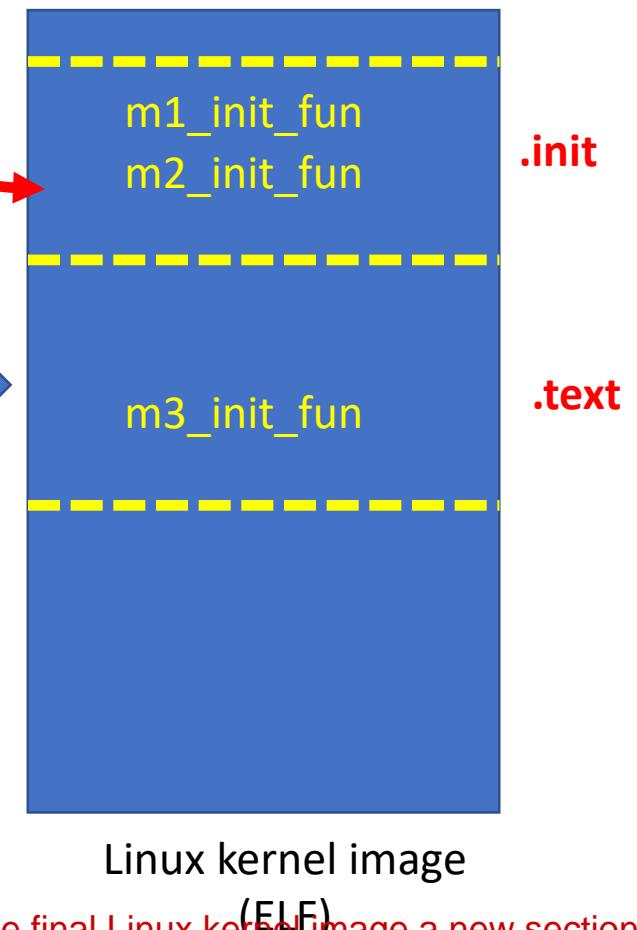


```
int m3_init_fun (void) { .... }
```



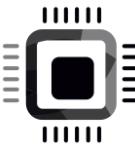
.init section

memory will be freed by the kernel during boot.



All `__exit` marked functions will be dropped by the kernel build system hence they will not be part of final image.

When you trigger the kernel build, all these 3 modules will be part of the final Linux kernel image. In the final Linux kernel image a new section will be created, that's called ".init". And `m1_init_function` and `m2_init_function` will be part of that .init section. That means, the codes of `m1_init_function` and `m2_init_function` will be placed in the section called .init. When the kernel boots, the kernel calls `m1_init_function` and `m2_init_function`. And after that, kernel frees the memory which has been consumed by the .init section. But, in this case `m3_init_function` will not be a part of .init. Because, it is not tagged with the macro, `__init`. So, it will be a part of .text region. That's why, `m3_init_function` will permanently consume the memory during run time of the kernel. And also remember that, all `__exit` marked functions will be dropped by the kernel build system hence they will not be part of final image.



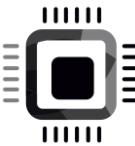
Module entry points registration

You have to inform or you have to register your kernel modules For that, we use some macros, which is given by the kernel, we use `module_init` and `module_exit` macros.

```
/* This is registration of above entry points with kernel */
module_init(my_kernel_module_init);
module_exit(my_kernel_module_exit);
```

- These are the macros used to register your module's init function and clean-up function with the kernel.
- Here `module_init/module_exit` is not a function, but a macro defined in `linux/module.h`
- For example, `module_init()` will add its parameter to the init entry point database of the kernel
- `module_exit()` will add its parameter to exit entry point database of the kernel

Basically, you just have to give your entry point name, as an argument to these macros, that's it.



Module description

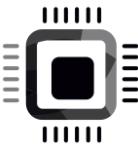
These are some metadata which we are going to include in your kernel module. You can include MODULE_LICENSE, MODULE_AUTHOR, and MODULE_DESCRIPTION. For that, these macros can be used.

```
/* This is descriptive information about the module */
MODULE_LICENSE("GPL"); /*This module adheres to GPL licensing */
MODULE_AUTHOR("www.fastbitlab.com"); to mention author of your module
MODULE_DESCRIPTION("A kernel module to print some messages");
```

Out of these macro's MODULE_LICENSE macro is very much important where you have to mention license type of the a kernel module what you are going to write.

- **MODULE_LICENSE** is a macro used by the kernel module to announce its license type .
- If you load module whose license parameter is non-GPL(General Public License), then kernel triggers warning of being tainted. Its way of kernel letting the users and developers know its non-free license based module.
- The developer community may ignore the bug reports you submit after loading the proprietary licensed module
- The declared module license is also used to decide whether a given module can have access to the small number of "GPL-only" symbols in the kernel.
- Go to linux/module.h to find out what are the allowed parameters which can be used with this macro to load the module without tainting the kernel.

default path:---- /lib/modules/\$(uname -r)/build/include/linux/module.h



MODULE_INFO

MODULE_INFO(name, "string_value");

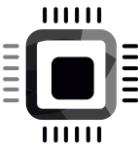
- MODULE_INFO(board,"Beagle bone");

You can see the module information by running the below command on the .ko file

arm-linux-gnueabihf-objdump -d -j .modinfo helloworld.ko

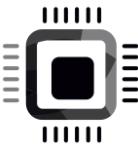
In a Linux system, the "include" folder typically contains header files that are essential for software development.

path:- /usr/include



Building a kernel module

- Kernel module can be built in 2 ways
 - Statically linked against the kernel image
 - Dynamically loadable.
- In most of the exercises in this course we will be writing and using dynamically loadable kernel modules.



TWO types of dynamically loadable kernel module

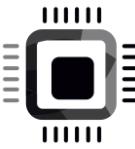


In tree module
(internal to the linux kernel tree)

which are already part of the Linux kernel are called In tree modules.

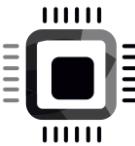
Out of tree module
(external to the linux kernel tree)

This method taints the kernel.
Kernel issues a warning saying out of tree
module has been loaded.
We can safely ignore the warning !



In-tree and out of tree

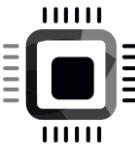
- Basically, out of tree means outside of the Linux kernel source tree.
- The modules which are already part of the Linux kernel are called in-tree modules. (approved by the kernel developers and maintainers)
- When you write a module separately(which is not approved and may be buggy), build and link it against the running kernel, then its called as out of the tree module.
- Hence when you load an out of tree kernel module, kernel throws a warning message saying it got tainted.



Building a kernel module(out of tree)

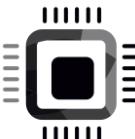
- Modules are built using "kbuild" which is the build system used by the Linux kernel
- Modules must use "kbuild" to stay compatible with changes in the build infrastructure and to pick up the right flags to GCC
- To build external modules, you must have a prebuilt kernel source available that contains the configuration and header files used in the build
- This ensures that as the developer changes the kernel configuration, his custom driver is automatically rebuilt with the correct kernel configuration

Reference : <https://www.kernel.org/doc/Documentation/kbuild/modules.txt>



Important note:

- When you are building out of tree(external) module, you need to have a complete and precompiled kernel source tree on your system
- The reason is, modules are linked against object files found in the kernel source tree
- You can not compile your module against one Linux kernel version and load it into the system, which is running kernel of different version. The module load may not be successful, and even if it is successful, you will encounter run time issues with the symbols.
- Thumb rule: “Have a precompiled Linux kernel source tree on your machine and build your module against that”
- There are two ways to obtain a prebuilt kernel version
 - Download kernel from your distributor and build it by yourself
 - Install the Linux-headers- of the target Linux kernel



Building a kernel module(out of tree)

the key thing here is that you have to trigger the top level make file of the linux kernel source tree, and for this we use :- { make -C }, make first enters into the linux kernel source tree and executes top level make file, thats how kernel build system triggers.

The command to build an external module is:

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

Kbuild rules

- Compiler switches
- Dependency list
- Version string

Linux kernel source tree

Top level Makefile



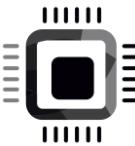
and when make invokes top level make file, all the compiler features ,depemdy list
virsion string, every thing will be utilised.

when make invokes top level make file , you then have to direct the top
level make file to local makefile, or local folder using { M= }

Local Makefile



Your working directory
Where external modules to
be compiled are stored



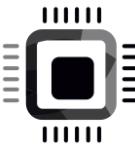
Command syntax

`make -C $KDIR M=$PWD [Targets]`

-C \$KDIR: The directory where the kernel source is located. "make" will actually change to the specified directory when executing and will change back when finished

M=\$PWD: Informs kbuild that an external module is being built. The value given to "M" is the absolute path of the directory where the external module (kbuild file) is located.

we used PWD to get the path of current working directory.

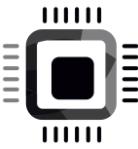


Command syntax

make -C \$KDIR M=\$PWD [Targets]



- **modules** :The default target for external modules. It has the same functionality as if no target was specified.
- **modules_install**: Install the external module(s). The default location is /lib/modules/<kernel_release>/extra/, but a prefix may be added with INSTALL_MOD_PATH
- **clean** : Remove all generated files in the module directory only.
- **help**:List the available targets for external modules



Creating a local Makefile

In the local makefile you should define a **kbuild** variable like below

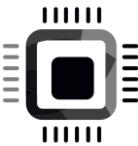
obj-<X> := <module_name>.o

Here **obj-<X>** is kbuild variable and '**X**' takes one of the below values

X = n , Do not compile the module

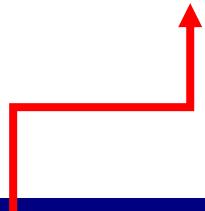
X= y, Compile the module and link with kernel image

X = m , Compile as dynamically loadable kernel module



Creating a local Makefile

obj-m := main.o



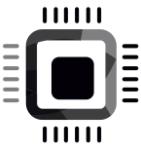
The kbuild system will build main.o from main.c and after linking, will result in the kernel module main.ko.

Here we do not have to include main.c
as main.c is source

our target is main.o

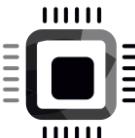
you can built kernel module for your host or for target, first lets try building our kernel module for host matchine, that is linux matcine.
for that we need pre-build kernel source of linux kernel version which is running on your system.

First of all, what's a version of Linux kernel which is running on our host. To check that, just run:--- uname -r



In-tree building

- You have to add the Linux kernel module inside the Linux kernel source tree and let the Linux build system builds that.
- If you want to list your kernel module selection in kernel menuconfig, then create and use a Kconfig file

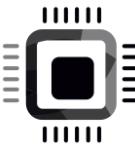


Steps to add in-tree module to kernel menu configuration

1. create a folder in `drivers/char/my_c_dev`
2. copy `main.c`
3. create Kconfig file and add the below entries

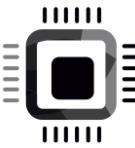
```
menu "my custom modules"
    config CUSTOM_HELLOWORLD
        tristate "hello world module support"
        default m
endmenu
```

4. add the local Kconfig entry to upper level Kconfig
5. create a local Makefile
6. add `obj-$(config_item) += <module>.o` in to local Makefile
7. add the local level makefile to higher level makefile



Debugging with printk

- printf is one of the best debugging tool we have in user-level applications.
- When you work in kernel space, you will not have any access to the C standard library to access functions like printf or scanf.
- But don't worry kernel has its own printf like kernel API called printk, where the letter 'k' signifies kernel space printing.
- `printf("Hello this is user application running \n");`
- `printk("Hello this is kernel code running \n");`
- `printf(" value of data1 = %d data = %d\n" , d1,d2);`
- `printk(" value of data1 = %d data = %d\n" , d1,d2);`



Debugging with printk

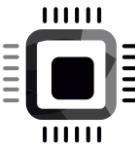
- When using `printk` , the message will go into the kernel ring buffer simply we call “Kernel log,” and we can print and control the kernel ring buffer using the command `dmesg`.

- So if you want to check the latest 5 kernel messages, just run

`dmesg | tail -5`

`dmesg | head -20`

- The `printk` does not support floating-point formats (`%e`, `%f`, `%g`)



How to get printk format specifiers right

=====

:Author: Randy Dunlap <rdunlap@infradead.org>
:Author: Andrew Murray <amurray@mpc-data.co.uk>

Integer types

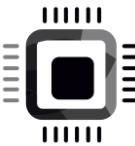
=====

::

If variable is of Type,	use printk format specifier:
-------------------------	------------------------------

char	%d or %x
unsigned char	%u or %x
short int	%d or %x
unsigned short int	%u or %x
int	%d or %x
unsigned int	%u or %x
long	%ld or %lx
unsigned long	%lu or %lx
long long	%lld or %llx
unsigned long long	%llu or %llx
size_t	%zu or %zx
ssize_t	%zd or %zx
s8	%d or %x
u8	%u or %x
s16	%d or %x
u16	%u or %x
s32	%d or %x
u32	%u or %x
s64	%lld or %llx
u64	%llu or %llx

linux/Documentation/printk-formats.txt



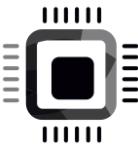
Kernel log levels

Based on the kernel log level you can control the priority of your `printk` messages. There are 8 such log levels. A lower number indicates a higher priority. And the default `printk` log level or priority is usually set to 4. i.e., `KERN_WARNING`

```
#define KERN_SOH      "\001"          /* ASCII Start Of Header */  
#define KERN_SOH_ASCII '\001'  
  
#define KERN_EMERG    KERN_SOH "0"    /* system is unusable */  
#define KERN_ALERT    KERN_SOH "1"    /* action must be taken immediately */  
#define KERN_CRIT     KERN_SOH "2"    /* critical conditions */  
#define KERN_ERR      KERN_SOH "3"    /* error conditions */  
#define KERN_WARNING  KERN_SOH "4"    /* warning conditions */  
#define KERN_NOTICE   KERN_SOH "5"    /* normal but significant condition */  
#define KERN_INFO     KERN_SOH "6"    /* informational */  
#define KERN_DEBUG    KERN_SOH "7"    /* debug-level messages */  
  
#define KERN_DEFAULT  ""             /* the default kernel loglevel */
```

include/linux/kern_levels.h

The log level will be used by the kernel to understand the priority of the message . Based on the priority kernel decides whether the message should be presented to the user immediately, by printing directly on to the console.



Kernel log levels

- All kernel messages will have their own log level
- You may have to specify the log level while using printk

```
printk("Hello this is kernel code running \n");
```

- in the above `printk`, there is no log level visible, right? but the kernel will add the default log level set by the kernel config item

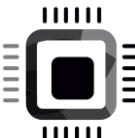
`CONFIG_MESSAGE_LOGLEVEL_DEFAULT` whose value is 4

- *So, the above statement is equivalent to*

`printk(KERN_WARNING "Hello this is kernel code running \n");`



Note : There is no comma(,) between log level and message



Significance of kernel log level

```
 printk(KERN_ALERT "Hello this action should be taken immediately \n");  
 printk(KERN_INFO "Hello this is just for your information\n");
```

The kernel message log level will be compared with the **current console log level**. If the kernel message log level is lower than the current console log level, then the message will be directly printed on the current console.

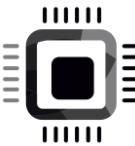
By default console log level will have the value of config item: CONFIG_CONSOLE_LOGLEVEL_DEFAULT
Its default value is set to 7. You can change it via kernel menuconfig or running commands.

To know the current log level status just run

```
cat /proc/sys/kernel/printk
```

At run time you can change the log level values using the below command

```
echo 6 > /proc/sys/kernel/printk
```

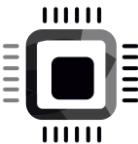


printk wrappers

Name	Log level	Alias function
KERN_EMERG	"0"	pr_emerg
KERN_ALERT	"1"	pr_alert
KERN_CRIT	"2"	pr_crit
KERN_ERR	"3"	pr_err
KERN_WARNING	"4"	pr_warning
KERN_NOTICE	"5"	pr_notice
KERN_INFO	"6"	pr_info
KERN_DEBUG	"7"	pr_debug (works only if DEBUG is defined)
KERN_DEFAULT	""	

`printf(KERN_INFO "Hello this is just for your information\n");`

`pr_info("Hello this is just for your information\n");`

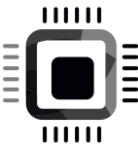


printk wrappers

```
/*
 * These can be used to print at the various log levels.
 * All of these will print unconditionally, although note that pr_debug()
 * and other debug macros are compiled out unless either DEBUG is defined
 * or CONFIG_DYNAMIC_DEBUG is set.
 */
#define pr_emerg(fmt, ...) \
    printk(KERN_EMERG pr_fmt(fmt), ##__VA_ARGS__)
#define pr_alert(fmt, ...) \
    printk(KERN_ALERT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_crit(fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_err(fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_notice(fmt, ...) \
    printk(KERN_NOTICE pr_fmt(fmt), ##__VA_ARGS__)
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
/*
 * Like KERN_CONT, pr_cont() should only be used when continuing
 * a line with no newline ('\n') enclosed. Otherwise it defaults
 * back to KERN_DEFAULT.
 */
#define pr_cont(fmt, ...) \
    printk(KERN_CONT fmt, ##__VA_ARGS__)

/* pr-devel() should produce zero code unless DEBUG is defined */
#ifndef DEBUG
#define pr-devel(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr-devel(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif
```

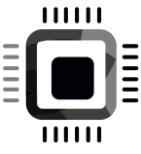
Include/linux/printk.h



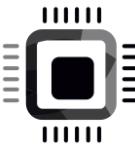
Kernel hexdump facilities

```
void print_hex_dump_bytes(const char *prefix_str, int prefix_type,  
                           const void *buf, size_t len);  
  
void print_hex_dump(const char *level, const char *prefix_str, int prefix_type,  
                    int rowsize, int groupsize, const void *buf, size_t len, bool ascii);
```

lib/hexdump.c



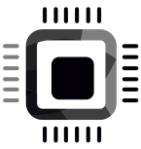
Device driver



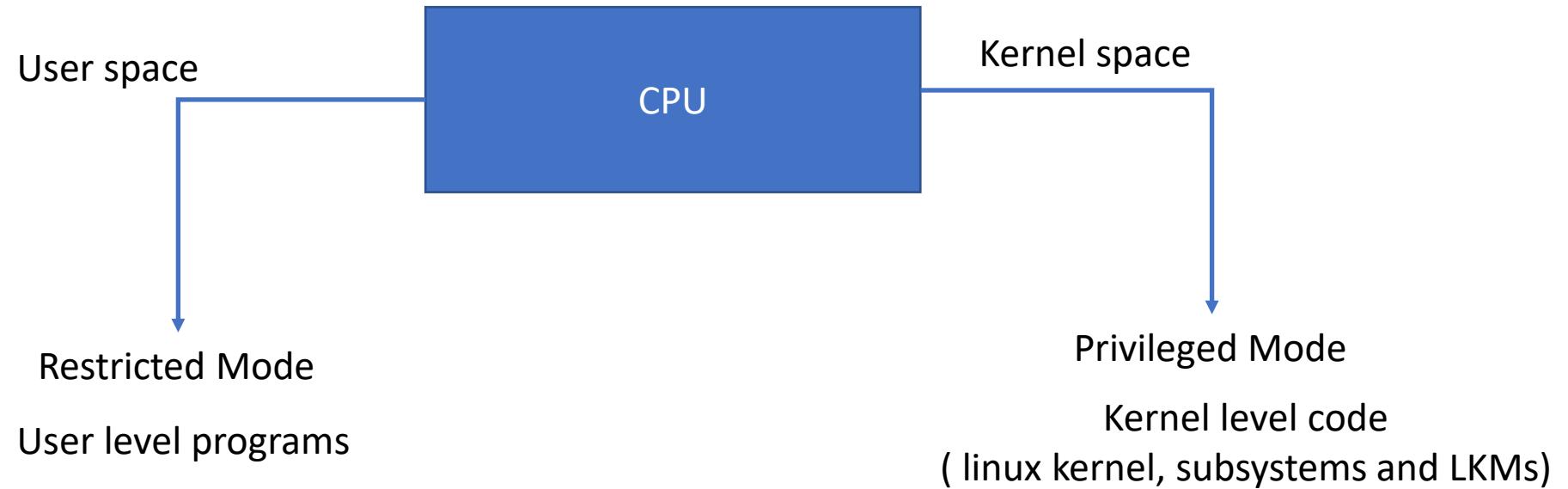
Section 3

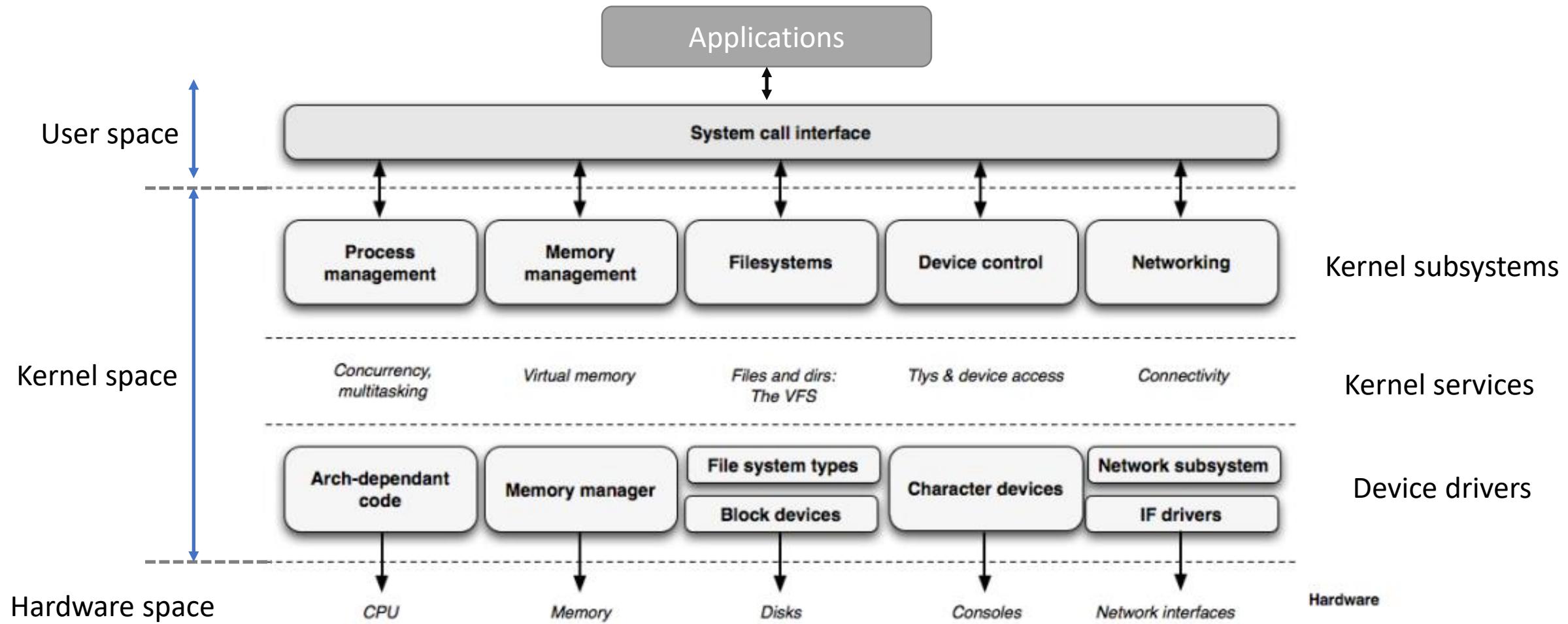
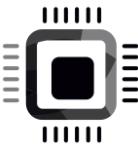
What is device driver ?

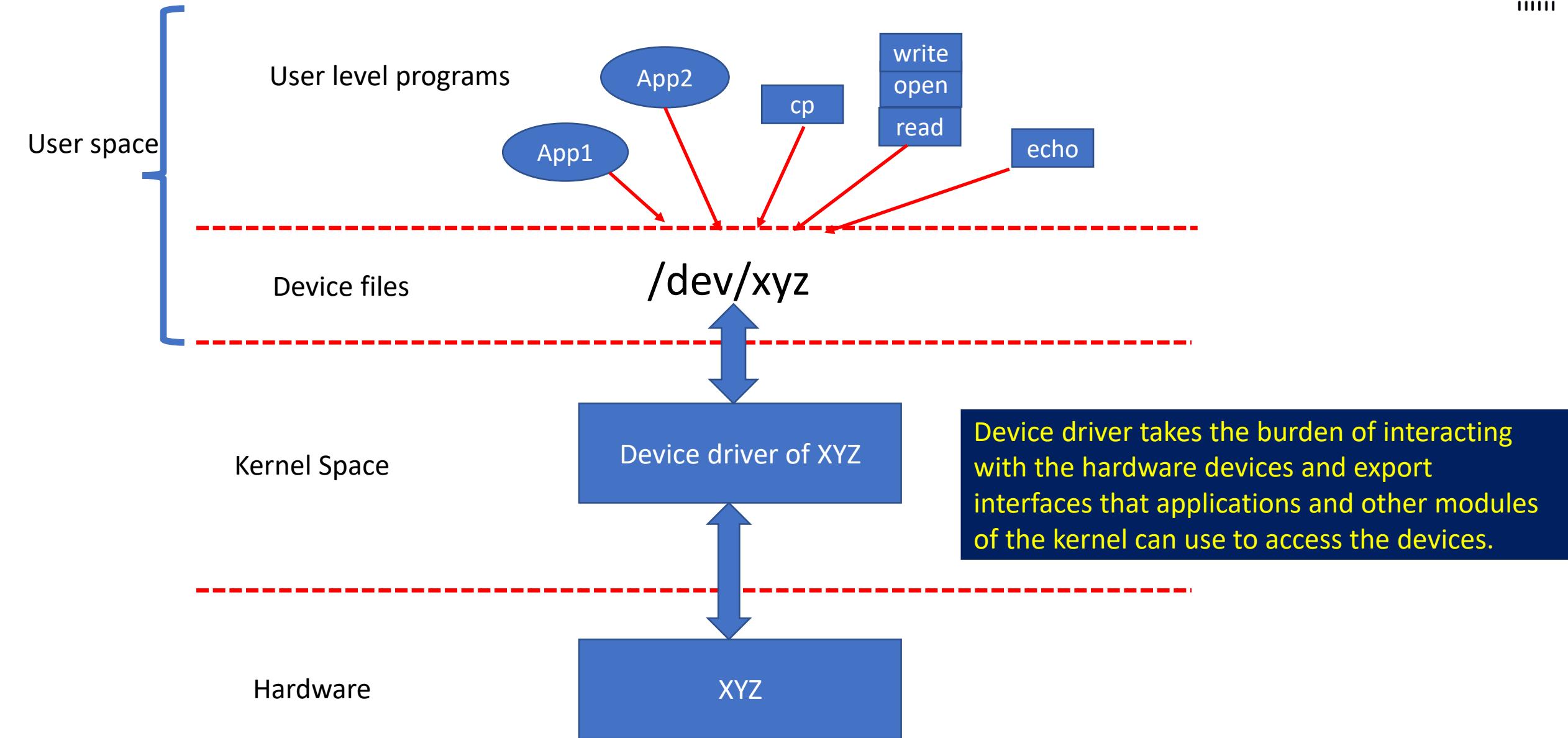
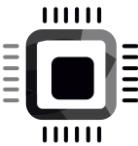
- A device driver is a piece of code that configures and manages a device.
- The device driver code knows, how to configure the device, sending data to the device, and it knows how to process requests which originate from the device.
- When the device driver code is loaded into the operating system such as Linux, it exposes interfaces to the user-space so that the user application can communicate with the device.
- Without the device driver, the OS/Application will not have a clear picture of how to deal with a device

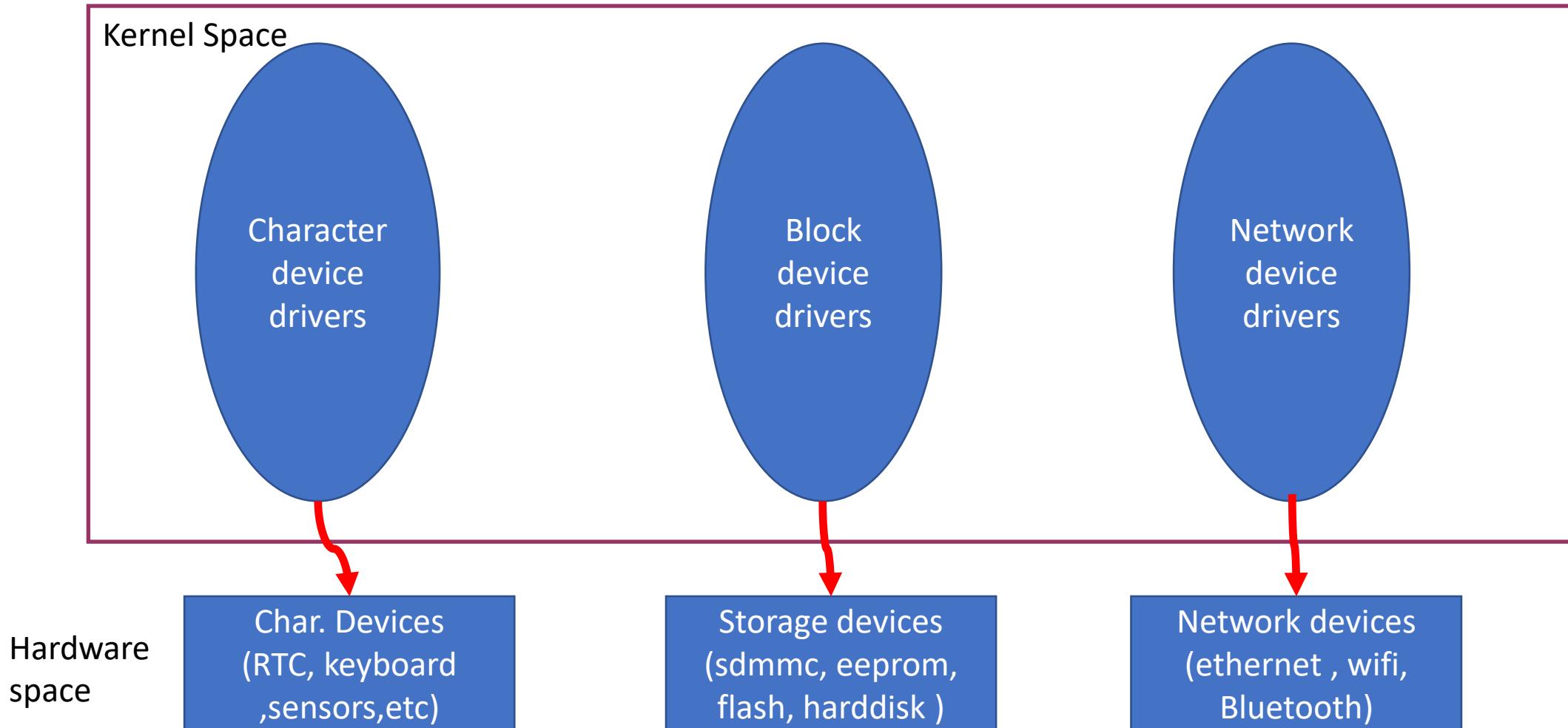
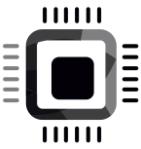


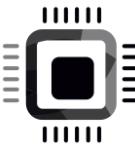
User space Vs Kernel Space





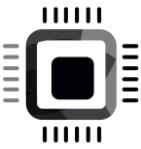






Character driver (char driver)

- Character driver accesses data from the device sequentially. i.e., byte by byte (like a stream of characters) not as a chunk of data
- Sophisticated buffering strategies are usually not involved in char drivers. Because when you write 1 byte, it directly goes to the device without any intermediate buffering, delayed write back, dirty buffer management.
- Char devices: sensors, RTC, keyboard, serial port, parallel port, etc.



User Application

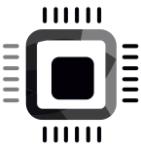
Kernel space

Hardware

write(fd,0xAB);
echo 0xAB > /dev/rtc

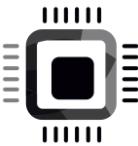
0xAB





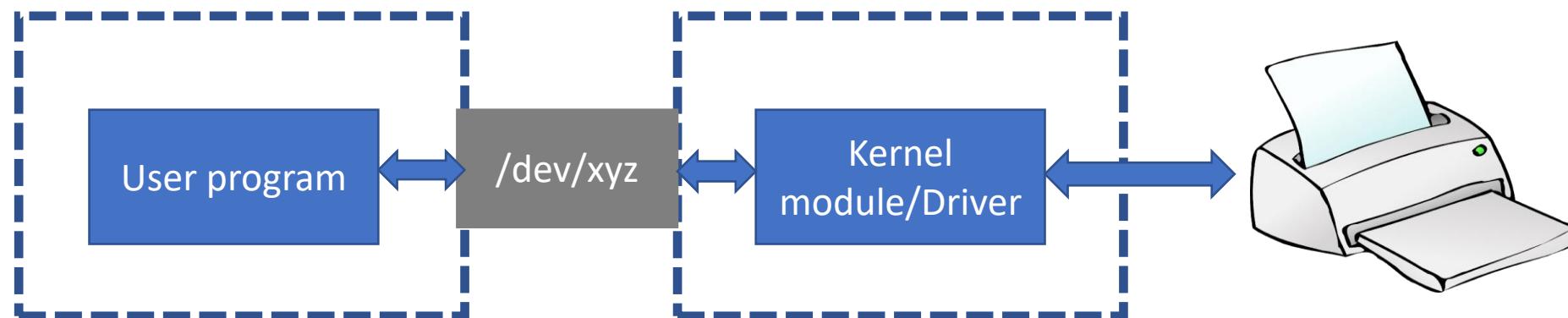
Block drivers

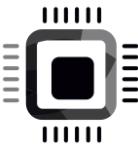
- The device which handles data in chunks or blocks is called a block device.
- Block drivers are more complicated than char drivers because block drivers should implement advanced buffering strategies to read and write to the block devices, and disk caches are involved.
- Examples: mass storage devices such as hard disks, SDMMC, Nand flash, USB camera, etc



Device files

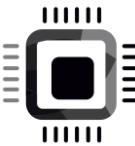
- Devices are accessed as a file in Unix/Linux systems.
- A device file is a special file or a node which gets populated in `/dev` directory during kernel boot time or device/driver hot plug events
- By using device file, user application and drivers communicate with each other.
- Device files are managed as part of VFS subsystem of the kernel





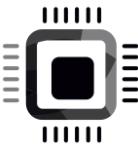
Device files

```
kiran@kiran-fastbiteba:/dev$ ls -l
crw----- 1 root  root      10, 227 Mar 16 09:52 mcelog
crw-rw---- 1 root  video     240,    0 Mar 16 09:52 media0
crw----- 1 root  root      241,    0 Mar 16 09:52 mei0
crw-r---- 1 root  kmem      1,     1 Mar 16 09:52 mem
crw----- 1 root  root      10,   56 Mar 16 09:52 memory_bandwidth
drwxrwxrwt 2 root  root          40 Mar 16 09:52 mqueue
drwxr-xr-x 2 root  root          60 Mar 16 09:52 net
crw----- 1 root  root      10,   58 Mar 16 09:52 network_latency
crw----- 1 root  root      10,   57 Mar 16 09:52 network_throughput
crw-rw-rw- 1 root  root      1,    3 Mar 16 09:52 null
crw----- 1 root  root     10, 144 Mar 16 09:52 nvram
crw-r---- 1 root  kmem      1,    4 Mar 16 09:52 port
crw----- 1 root  root     108,   0 Mar 16 09:52 ppp
crw----- 1 root  root      10,    1 Mar 16 09:52 psaux
crw-rw-rw- 1 root  tty       5,    2 Mar 16 12:08 ptmx
drwxr-xr-x 2 root  root          0 Mar 16 09:52 pts
crw-rw-rw- 1 root  root      1,    8 Mar 16 09:52 random
crw-rw-r--+ 1 root  netdev    10, 242 Mar 16 09:52 rfkill
lrwxrwxrwx 1 root  root          4 Mar 16 09:52 rtc -> rtc0
crw----- 1 root  root     249,   0 Mar 16 09:52 rtc0
brw-rw---- 1 root  disk      8,    0 Mar 16 09:52 sda
brw-rw---- 1 root  disk      8,    1 Mar 16 09:52 sda1
crw-rw---- 1 root  disk     21,    0 Mar 16 09:52 sg0
```



Exercise : pseudo character driver

- Write a character driver to deal with a pseudo character device
- The pseudo-device is a memory buffer of some size
- The driver what you write must support reading, writing and seeking to this device
- Test the driver functionality by running user-level command such as echo, dd, cat and by writing user lever programs



when user calls open system call, then this system call should connect to the open system call implementation of driver in kernel space

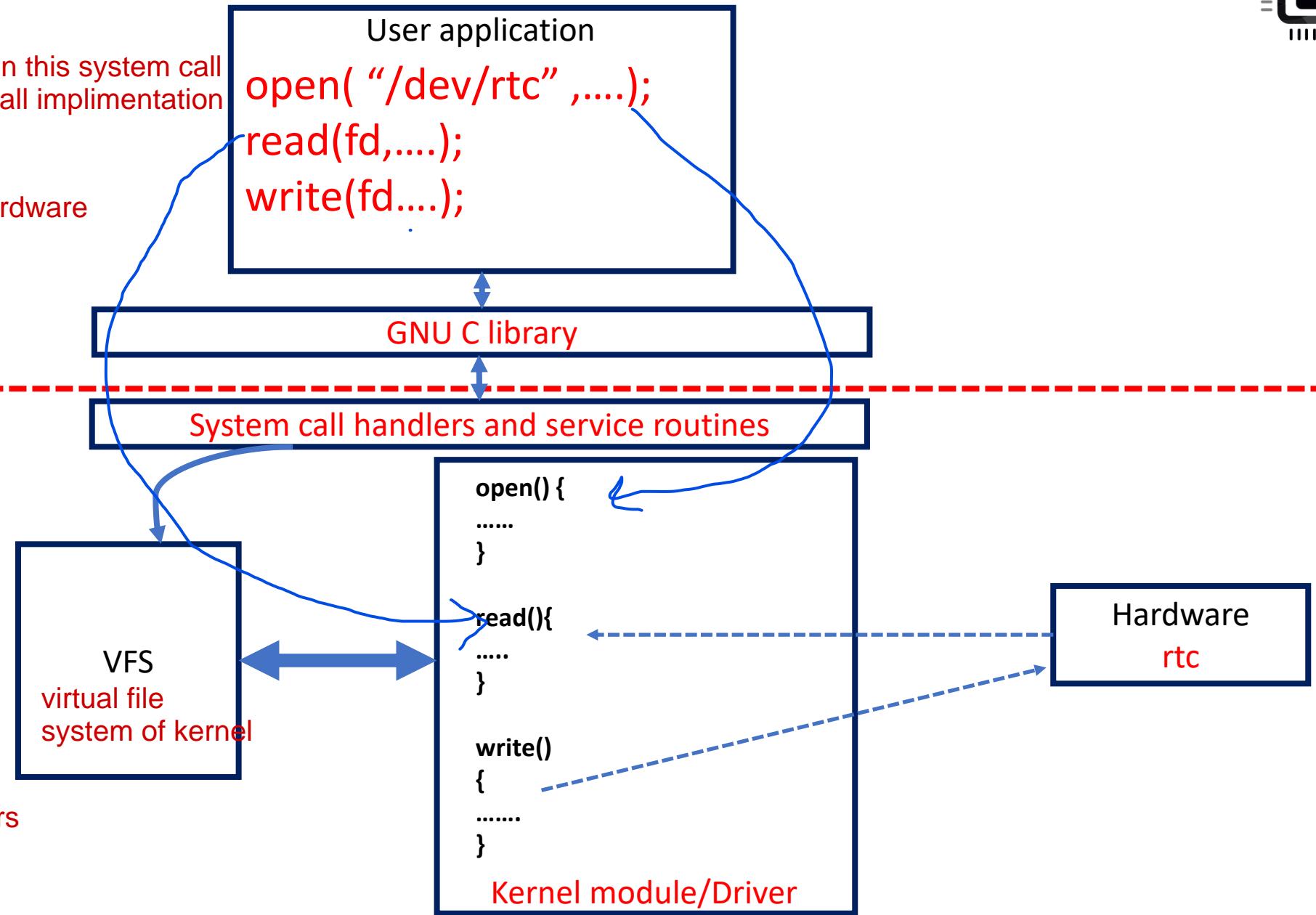
then driver will access data from hardware

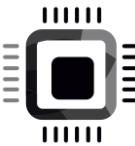
User space

Kernel space

this connection bw user space system call and driver system call handlers is taken care by VFS

that means the device driver handlers should be registered in VFS





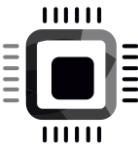
when you use an open system call on the device file, how does the kernel connects the open system call to the intended drivers open method?

to establish the connection the kernel uses something called "Device number".

lets assign a number to this file lets say 4.

and also to this device file lets 4:0 4 denotes number to the driver which should be used to connect device file access , and 0 is device instance there could be multiple device file .

Device number (Major & minor)



to establish connection the kernel uses device number

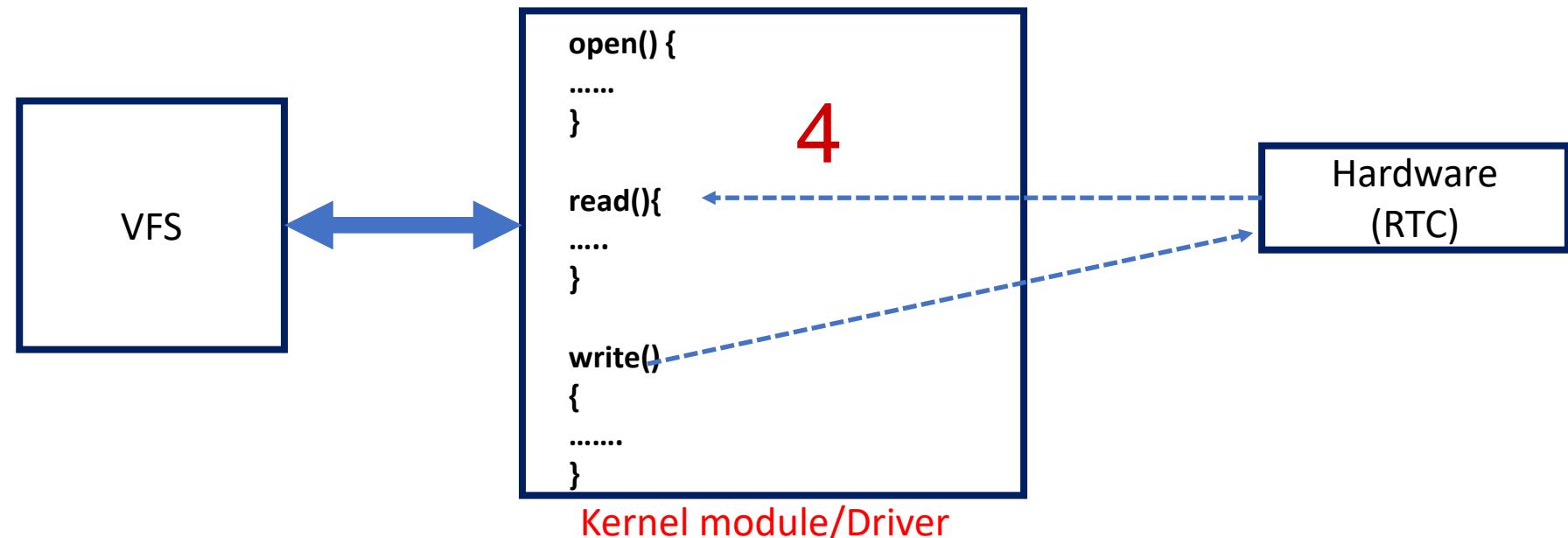
lets assign a number to the driver lets say 4 and also to this device file 4:0

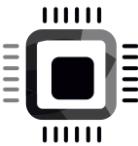
so here 4 denotes the number of the driver which should be used to connect this device file aces to the drivers method and 0 here is device instance, there could be multiple device all are handled by single driver.

/dev/rtc 4:0

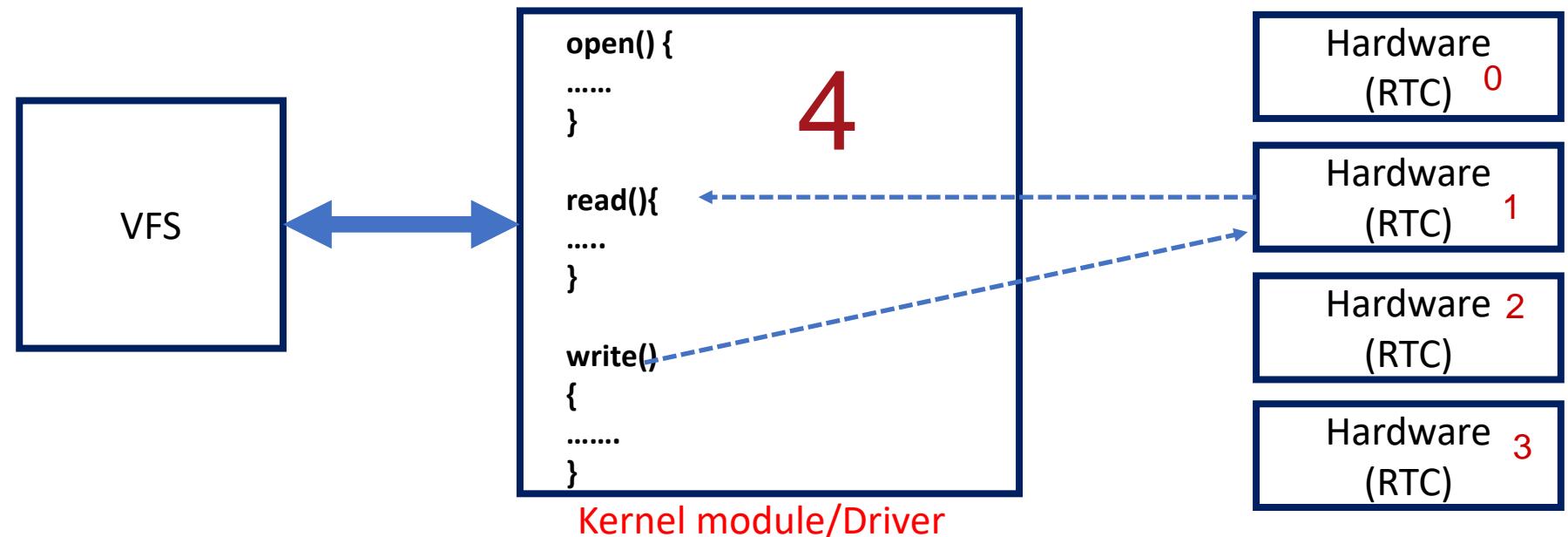
User space

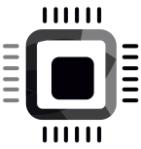
Kernel space





Kernel space





minor no help driver to differentiate bw device file.

127:0

127:1

127:2

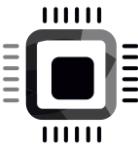
127:3

127:4

127:5

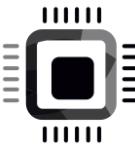
127:6

First device number will be updated in this variable



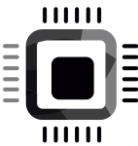
Major and minor numbers

kiran@kiran-fastbiteba:/dev\$ ls -l										
c rw-----	1	root	root	10,	227	Mar 16	09:52	mcelog		
c rwx-----	1	root	video	240,	0	Mar 16	09:52	media0		
c r-----	1	root	root	241,	0	Mar 16	09:52	mei0		
c r-----	1	root	kmem	1,	1	Mar 16	09:52	mem		
c r-----	1	root	root	10,	56	Mar 16	09:52	memory_bandwidth		
d rwxrwxrwt	2	root	root		40	Mar 16	09:52	mqueue		
d rwxr-xr-x	2	root	root		60	Mar 16	09:52	net		
c r-----	1	root	root	10,	58	Mar 16	09:52	network_latency		
c r-----	1	root	root	10,	57	Mar 16	09:52	network_throughput		
c r-w-rw-	1	root	root	1,	3	Mar 16	09:52	null		
c r-----	1	root	root	10,	144	Mar 16	09:52	nvram		
c r-----	1	root	kmem	1,	4	Mar 16	09:52	port		
c r-----	1	root	root	108,	0	Mar 16	09:52	ppp		
c r-----	1	root	root	10,	1	Mar 16	09:52	psaux		
c r-w-rw-	1	root	tty	5,	2	Mar 16	12:08	ptmx		
d rwxr-xr-x	2	root	root		0	Mar 16	09:52	pts		
c r-w-rw-	1	root	root	1,	8	Mar 16	09:52	random		
c r-w-r--+	1	root	netdev	10,	242	Mar 16	09:52	rkill		
lrwxrwxrwx	1	root	root		4	Mar 16	09:52	rtc -> rtc0		
c r-----	1	root	root	249,	0	Mar 16	09:52	rtc0		
b r-w----	1	root	disk	8,	0	Mar 16	09:52	sda		
b r-w----	1	root	disk	8,	1	Mar 16	09:52	sda1		
c r-w----	1	root	disk	21,	0	Mar 16	09:52	sg0		
.										



Connection establishment between device file access and the driver

- 1.**• Create device number
- 2.**• Create device files
- 3.**• Make a char device registration with the VFS (CDEV_ADD)
- 4:** Implement the driver's file operation methods for open, read, write, llseek , etc.



Kernel APIs and utilities to be used in driver code

alloc_chrdev_region();

1.Create device number

cdev_init();

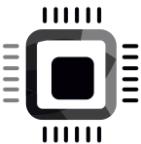
2. Make a char device registration
with the VFS

cdev_add();

class_create();

3. Create device files

device_create();



Kernel APIs and utilities to be used in driver code

Creation

alloc_chrdev_region();

cdev_init();
cdev_add();

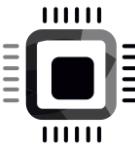
class_create();
device_create();

Deletion

unregister_chrdev_region();

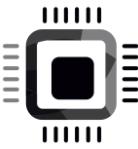
cdev_del();

class_destroy();
device_destroy();



Kernel Header file details

Kernel functions and data structures	Kernel header file
alloc_chrdev_region() unregister_chrdev_region()	include/linux/fs.h
cdev_init() cdev_add() cdev_del()	include/linux/cdev.h
device_create() class_create() device_destroy() class_destroy()	include/linux/device.h
copy_to_user() copy_from_user()	include/linux/uaccess.h
VFS structure definitions	include/linux/fs.h

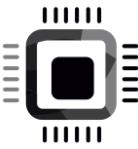


how to use alloc_chrdev_region api to create Dynamically,register a range of char device numbers

```
pointer where the first device number will  
be updated by this API  
  
output parameter  
for first assigned  
number  
  
int alloc_chrdev_region(dev_t *dev, unsigned baseminor,  
                        unsigned count, const char *name);  
  
number of minor  
numbers required  
  
name of the associated  
device or driver
```

The diagram illustrates the parameters of the `alloc_chrdev_region` function. It shows the function signature `int alloc_chrdev_region(dev_t *dev, unsigned baseminor, unsigned count, const char *name);`. Four parameters are highlighted with callouts:

- Output parameter for first assigned number:** Points to the `*dev` parameter.
- Number of minor numbers required:** Points to the `count` parameter.
- First of the requested range of minor numbers:** Points to the `baseminor` parameter.
- Name of the associated device or driver:** Points to the `*name` parameter.



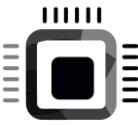
Example

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

how to use this API

```
/* Device number creation */  
dev_t device_number;  
  
alloc_chrdev_region(&device_number, 0, 7, "eeprom");
```

device numbers
here 7 device number would be
created.



Name of this device numbers “eeprom”

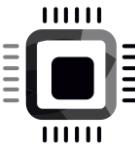
127:0 127:1 127:2 127:3 127:4 127:5 127:6

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

First device number
will be updated in
this variable

First minor

Total seven device
numbers created



Device number representation

- The device number is a combination of major and minor numbers
- In Linux kernel, `dev_t` (typedef of `u32`) type is used to represent the device number.
- Out of 32 bits, 12 bits to store major number and remaining 20 bits to store minor number
- You can use the below macros to extract major and minor parts of `dev_t` type variable

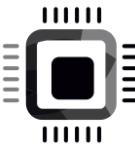
```
dev_t device_number;
```

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

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

- You can find these macros in `linux/kdev_t.h`
- If you have, major and minor numbers , use the below macro to turn them into `dev_t` type device number

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



Kernel APIs and utilities to be used in driver code

to create device number and add it to VFS we are going to use some API's.

alloc_chrdev_region();

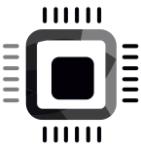
1.Create device number

cdev_init();
cdev_add();

2. Make a char device registration
with the VFS

class_create();
device_create();

3. Create device files

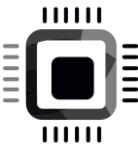


Initialize a cdev structure

structure to initialize

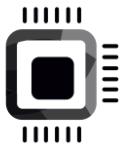
```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);
```

File operations for this device



Example

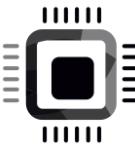
```
void cdev_init(struct cdev *cdev, const struct file_operations *fops);  
  
/*  
 *Initialize file ops structure with driver's  
 *system call implementation methods  
 */  
struct file_operations eeprom_fops;  
  
struct cdev eeprom_cdev;  
  
cdev_init(&eeprom_cdev, &eeprom_fops);
```



fs/char_dev.c

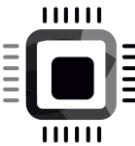
```
/**  
 * cdev_init() - initialize a cdev structure  
 * @cdev: the structure to initialize  
 * @fops: the file_operations for this device  
 *  
 * Initializes @cdev, remembering @fops, making it ready to add to the  
 * system with cdev_add().  
 */  
void cdev_init(struct cdev *cdev, const struct file_operations *fops)  
{  
    memset(cdev, 0, sizeof *cdev);  
    INIT_LIST_HEAD(&cdev->list);  
    kobject_init(&cdev->kobj, &ktype_cdev_default);  
    cdev->ops = fops;  
}
```

cdev_init is a kernel API which is implemented in char_dev.c



VFS file operation structure

```
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 *);  
    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);  
    -----  
    -----  
    -----  
}  
Include/linux/fs.h
```



Structure member elements initialization

```
struct CarModel
```

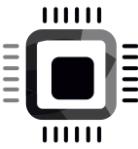
```
{
```

```
    unsigned int carNumber;  
    uint32_t    carPrice;  
    uint16_t    carMaxSpeed;  
    float       carWeight;
```

```
};
```

```
struct CarModel CarBMW ={2021,15000,220,1330 }; // C89 method . Order is important
```

```
struct CarModel CarBMW ={.carNumber= 2021,.carWeight =1330 ,.carMaxSpeed =220,.carPrice =15000 };  
                           //C99 method using designated initializers
```



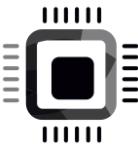
/ include / linux / cdev.h

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

Pointer to file operation structure of
the driver

A pointer to the module that owns this
structure; it should usually be initialized
to **THIS_MODULE**.

This field is used to prevent the module from
being unloaded while the structure is in use.

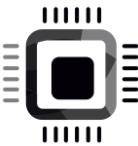


THIS_MODULE

- THIS_MODULE is a macro which resolves in to ‘pointer to a **struct module variable** which corresponds to our current module’
- You can find this macro in **linux/export.h**

/ include / linux / export.h

```
#ifdef MODULE
extern struct module __this_module;
#define THIS_MODULE (&__this_module)
#else
#define THIS_MODULE ((struct module *)0)
#endif
```



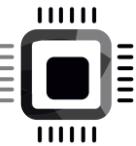
Add a char device to the Kernel VFS

cdev structure for the device

number of consecutive minor
numbers corresponding to this
device

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

first device number for
which this device is
responsible



Example

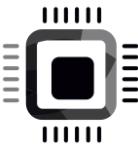
```
int cdev_add(struct cdev *p, dev_t dev, unsigned count);

/*
 *Initialize file ops structure with driver's
 *system call implementation methods
 */
struct file_operations eeprom_fops;

struct cdev eeprom_cdev;

cdev_init(&eeprom_cdev,&eeprom_fops);

cdev_add(&eeprom_cdev, device_number, 1);
```



Kernel APIs and utilities to be used in driver code

alloc_chrdev_region();

1.Create device number

cdev_init();

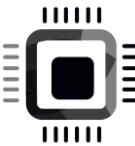
2. Make a char device registration
with the VFS (CDEV_ADD)

cdev_add();

class_create();

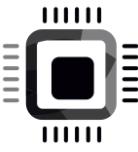
3. Create device files

device_create();



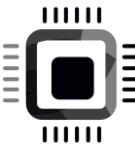
Dynamic device file creation in Linux

- In Linux, you can create a device file dynamically(on demand), i.e you need not manually create the device files under `/dev` directory to access your hardware.
- User-level program such as `udevd` can populate `/dev` directory with device files dynamically
- `udev` program listens to the `uevents` generated by hot plug events or kernel modules. When `udev` receives the `uevents`, it scans the subdirectories of `/sys/class` looking for the '`dev`' files to create device files.
- For each such '`dev`' file, which represents a combination of major and minor number for a device, the `udev` program creates a corresponding device file in `/dev` directory



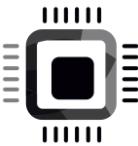
udev

- So, udev relies on device information being exported to user space through sysfs
- uevents are generated when device driver takes the help of kernel APIs to trigger the dynamic creation of device files or when a hot-pluggable device such as a USB peripheral is plugged into the system



Dynamic device file creation in Linux

- All that a device driver needs to do , for udev to work properly with it, is ensure that any major and minor numbers assigned to a device controlled by the driver are exported to user space through sysfs
- The driver exports all the information regarding the device such as device file name, major , minor number to sysfs by calling the function **device_create**
- udev looks for a file called '**dev**' in the **/sys/class/** tree of sysfs, to determine what the major and minor number is assigned to a specific device



class_create and device_create

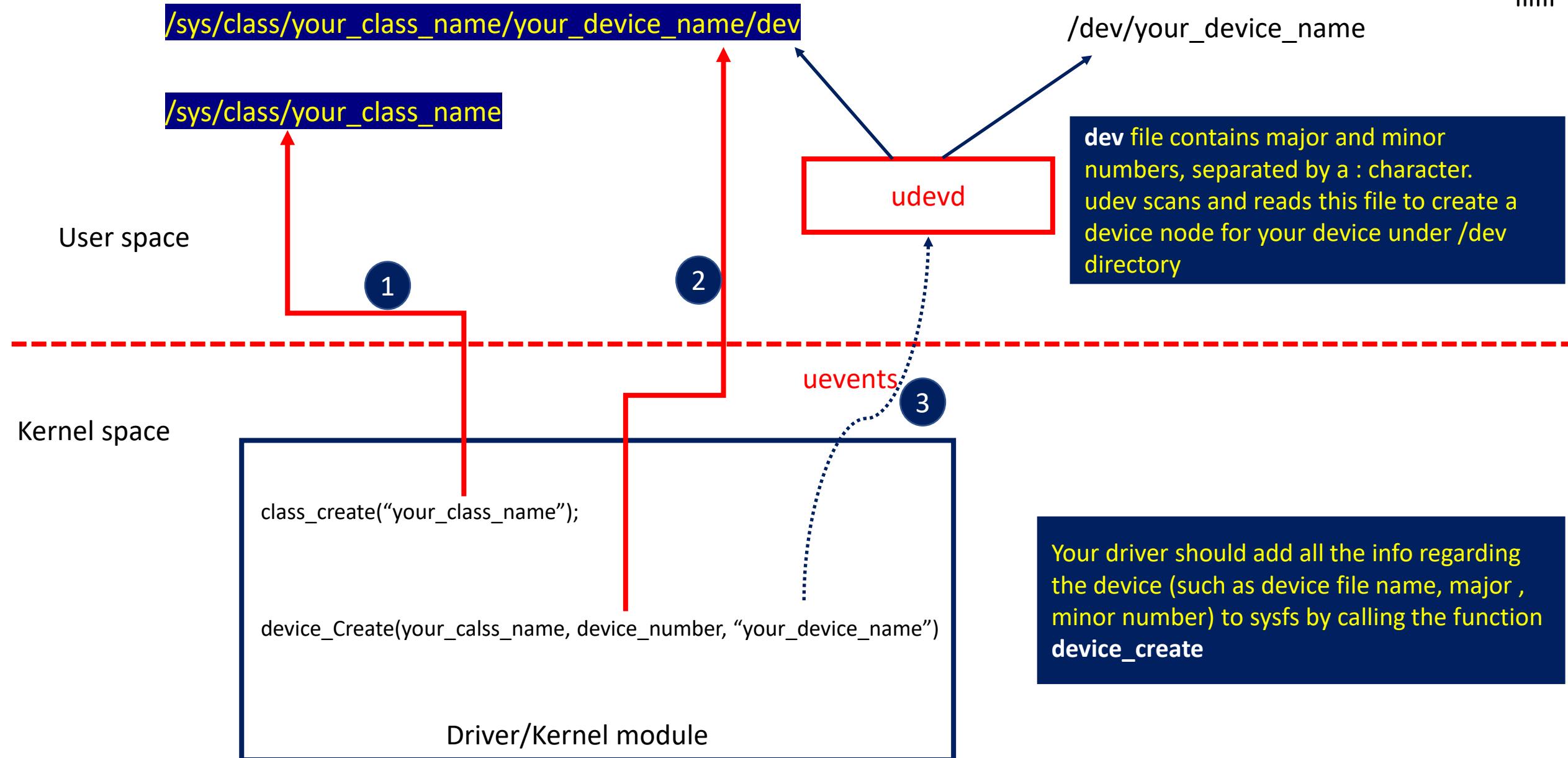
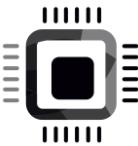
class_Create:

Create a directory in sysfs : `/sys/class/<your_class_name>`

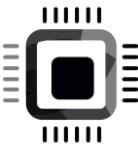
device_create :

This function creates a subdirectory under
`/sys/class/<your_class_name>` with your device name.

This function also populates sysfs entry with dev file which consists of the major and minor numbers, separated by a : character



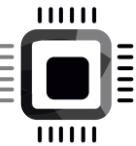
Your driver should add all the info regarding the device (such as device file name, major , minor number) to sysfs by calling the function `device_create`



Create and register a class with sysfs

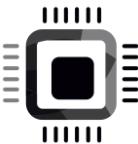
```
pointer to the module  
that is to "own" this  
struct class  
  
struct class * class_create ( struct module *owner, const char *name);
```

string for the
name of this
class



Example

```
struct class * class_create ( struct module *owner, const char *name);  
  
struct class *eeprom_class;  
  
eeprom_class = class_create(THIS_MODULE, "eeprom_class");
```



Populates the sysfs class you created in previous step
with device numbers and device names

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

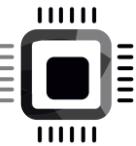
pointer to the struct class
that this device should be
registered to

pointer to the parent struct
device of this new device

dev_t for the char
device to be added

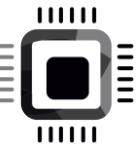
string for the device's name

data to be added to the
device for callbacks

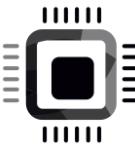


Example

```
struct device *device_create(struct class *class, struct device *parent,  
                           dev_t devt, void *drvdata, const char *fmt, ...);  
  
struct device *eeprom_dev;  
  
eeprom_dev = device_create(eeprom_class, NULL, device_number, NULL, "eeprom");
```



Character driver file operation methods

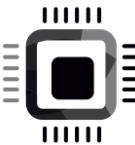


Character driver file operation methods

- In this character driver we will give support to handle the below user level system calls
 - open
 - close
 - read
 - write
 - llseek

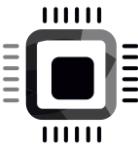
in above section we created structure of type struct `file_operations` , this structure is collection of various function pointer and also have possible file operation methods for regular file or fore a device file.

in this course we will be using llseek - read ,write , open, release methods in your driver and after that you have to initialize these member elements by using struct `file_operation` structure variable which you defined in your driver file.



VFS file operation structure

```
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 *);  
    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);  
    -----  
    -----  
    -----  
}  
Include/linux/fs.h
```



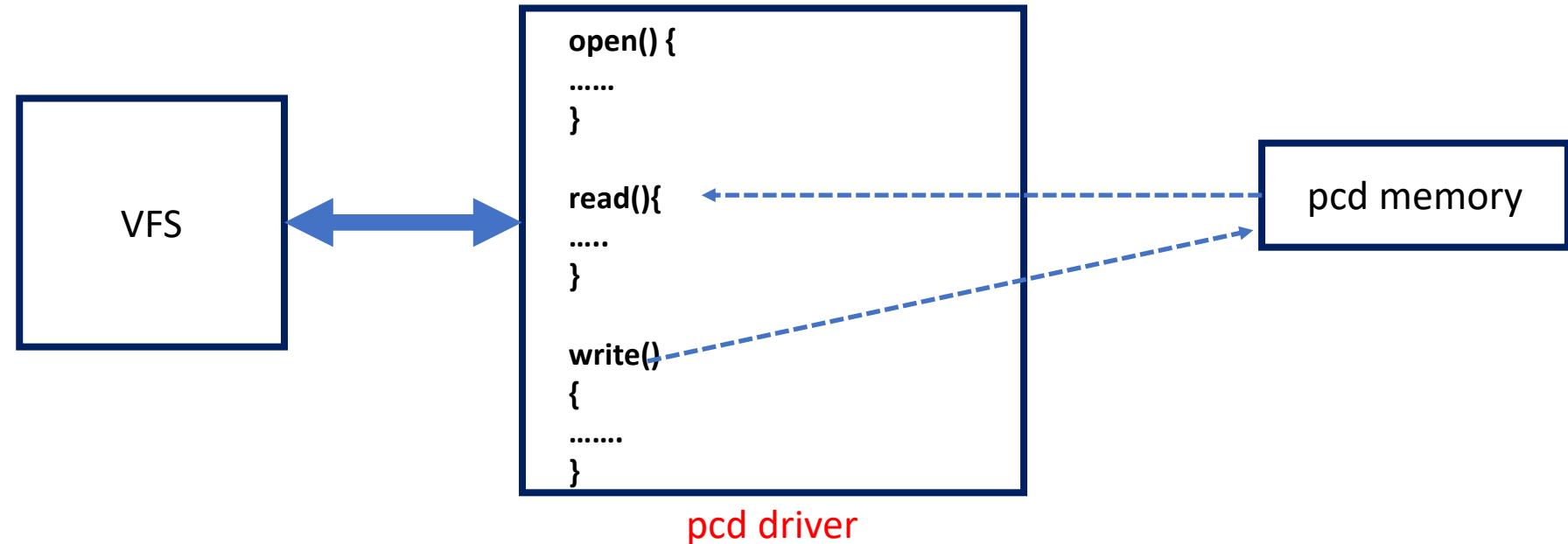
let you have loaded your pcd driver, pcd stands for pseudo character driver, and lets say your driver has created a device file /dev/pcd

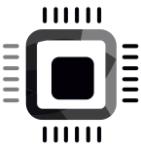
lets assume that user is going to use open system call, to open this device file

User space

/dev/pcd

Kernel space





1. Open

User space

`fd = open("/dev/pcd", O_RDWR);`



`/dev/pcd`

when open system call is used on device file { /dev/pcd } ,system call is transferred

From user level, the control is first passed to the kernels VFS subsystem.

Kernel space

And from the VFS subsystem,
the control is passed to the
appropriate drivers open method.



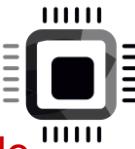
Opens a file by creating a new file object and linking it to the corresponding inode object

This is a open method, let's call it as pcd_open.

```
int pcd_open(struct inode *inode, struct file *filp)
{
    Pointer of Inode
    associated with
    filename
    Pointer of file object
}
```

The first argument is pointer of Inode associated with file name, the device file name.
And the second argument what it passes is a pointer of file object.

pcd driver



VFS data structures involved

struct cdev, which represent a character device and we registered this struct.

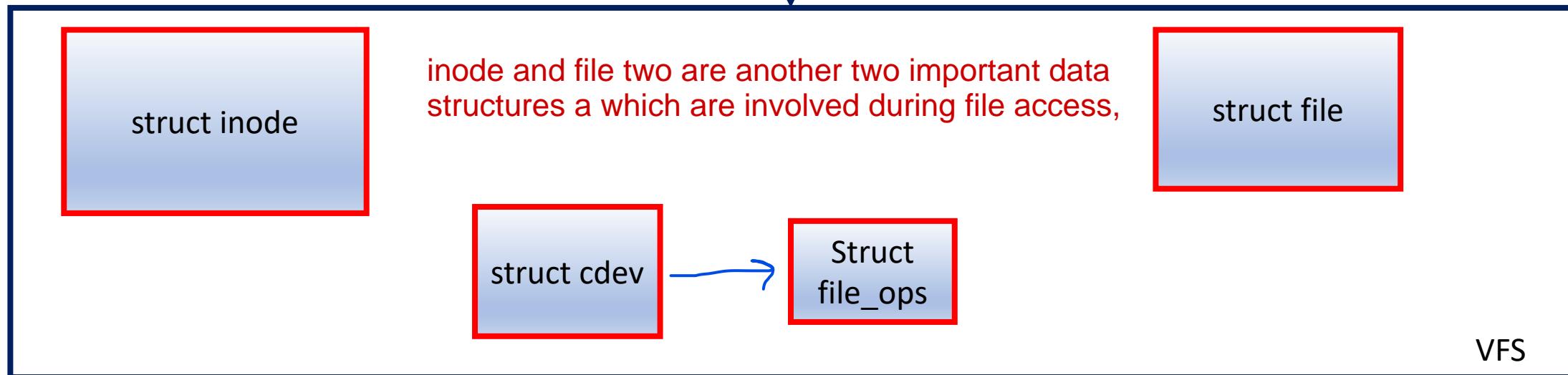
While registering the cdev, we initialize the cdev's file operation with our drivers file operation methods. Basically, cdev points to file_ops, Cdev has a field, which holds pointer to struct file_ops

User space

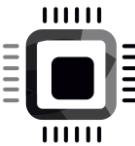
`fd = open("/dev/pcd",O_RDWR);`

`/dev/pcd`

Kernel space

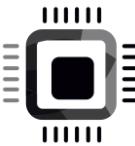


During file operation of a regular file or a device file, there are various VFS data structures involved. The important data structures involved are `struct inode`, `struct cdev`, `struct file operations`, and `struct file`



Inode object

- Unix makes a clear distinction between the contents of a file and the information about a file
- An inode is a VFS data structure(**struct inode**) that holds general information about a file.
- Whereas VFS ‘file’ data structure (**struct file**) tracks interaction on an opened file by the user process
- Inode contains all the information needed by the filesystem to handle a file.
- Each file has its own inode object, which the filesystem uses to identify the file
- Each inode object is associated with an inode number, which uniquely identifies the file within the filesystem.
- The inode object is created and stored in memory as and when a new file (regular or device) gets created
only single inode object is created in memory for regular or device file when it is created.



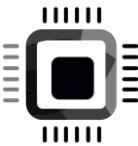
File object

- Whenever a file is opened a file object is created in the kernel space .
There will one file object for every open of a regular or device file.

That means, if you open the same file, let's say 10 times, then there will be 10 file objects will be created in the memory. There will be one file object for every open of a regular or device file.

- Stores information about the interaction between an open file and a process
- This information exists only in kernel memory during the period when a process has the file open. The contents of file object is NOT written back to disks unlike inode.

So, the file object stores information about the interaction between an open file and a user process.



Inode object initialization

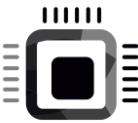
/ fs / inode.c

```
void init_special_inode(struct inode *inode, umode_t mode, dev_t rdev)
{
    inode->i_mode = mode;
    if (S_ISCHR(mode)) {
        inode->i_fop = &def_chr_fops;
        inode->i_rdev = rdev;
    } else if (S_ISBLK(mode)) {
        inode->i_fop = &def_blk_fops;
        inode->i_rdev = rdev;
    } else if (S_ISFIFO(mode))
        inode->i_fop = &pipefifo_fops;
    else if (S_ISSOCK(mode))
        ; /* Leave it no_open_fops */
    else
        printk(KERN_DEBUG "init_special_inode: bogus i_mode (%o) for"
               " inode %s:%lu\n", mode, inode->i_sb->s_id,
               inode->i_ino);
}
```

- Whenever device file is created(udev or mknod) **init_special_inode()** gets called
- Here, inode object's **i_rdev** is initialized (**i_rdev** is device number)
- **inode->i_fop** field is initialized with default file operations (**def_chr_fops**)

This is called by the VFS. And this function you can find in fs/inode.c
 Here, you see the virtual file system passes the device number, rdev of the newly created device file.

And I said, whenever you create a device file inode object will be created. That inode objects address will be passed here. And the VFS also sends what kind of device file it is. Whether it is a character device file, or a block device file, or a FIFO or something like that using the mode in this argument.



mode means whether it is character device S_ISCHR file, block S_ISBLK , fifo S_ISFIFO or S_ISSOCK

/ fs / char_dev.c

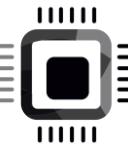
```
/*
 * Dummy default file-operations: the only thing this does
 * is contain the open that then fills in the correct operations
 * depending on the special file...
 */
const struct file_operations def_chr_fops = {
    .open = chrdev_open,
    .llseek = noop_llseek,
};
```

flow of above function :- first code decodes the mode. next in if Here, it is checking for ISCHR is it a character device file, something like that. If true, i_rdev, that's a device number. Where 'i' stands for inode actually. Is initialized to the rdev, That is the device number of the newly created device file, which is passed here. That means, now inode is formed and inodes device number field is initialized with the device number of the newly created device file.

And after that, inode has one more field called i_fop. i_fop is a pointer variable of type struct file operations. And this is initialized to default character file operations. All these things happen when you create a device file.

What is this def_chr_fops? == This is defined in fs/char_dev.c. This is actually a dummy default file operations.

These two main activities happen when you create a device file. VFS calls this function and it initializes newly created inodes object with the default character file operations, and it initializes the device number field of the newly created inode object with the device number of the newly created device file.



So, in the previous slide you understood that whenever a device file is created, an inode object is created in the memory, and inode objects a device number is initialized, and inode objects file operation field is also initialized with dummy default file operation methods.

open

Inode object is there in the memory now.

User space

do_sys_open

Return 'fd' to user space (-1 means failure)

Kernel space

do_filp_open

Whenever the user level program executes open system call on a device file, these are the kernel level functions a get involved in processing the open system call. It first executes a do_sys_open kernel function in the kernel space, and that calls do_filp_open function.

'file' object allocation

do_dentry_open

Using default fops

and after that do_filp_open calls, do_dentry_open. In this function, what happens is, the default dummy file operations will be called. chrdev_open

Replacing default fops with 'cdev' fops

chrdev_open

from chrdev_open, your driver's open method will be called. These are the activities involved in a open system call processing in the kernel space.

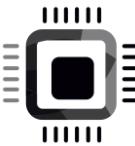
your driver open
method

This is a place where a file object will be created.

Remember what I said in the previous slide, for every open a file object will be created in the kenel space. That file object creation happens in this kernel function

For more details refer Understanding linux kernel 3rd edition, page 524 "The open() System Call "

Let's check the source code of do_dentry_open. Here it is. So, this function you can find in fs/open.c , and here *f is a newly created file object pointer.



```
static int do_dentry_open(struct file *f,
                         struct inode *inode,
                         int (*open)(struct inode *, struct file *),
                         const struct cred *cred)

{
    f->f_op = fops_get(inode->i_fop);
    if (unlikely(WARN_ON(!f->f_op))) {
        error = -ENODEV;
        goto cleanup_all;
    }

    error = security_file_open(f, cred);
    if (error)
        goto cleanup_all;

    error = break_lease(locks_inode(f), f->f_flags);
    if (error)
        goto cleanup_all;

    if (!open)
        open = f->f_op->open;
    if (open) {
        error = open(inode, f);
        if (error)
            goto cleanup_all;
    }
}
```

take a look here, The struct file also has file operation variable, a pointer variable. That gets initialized here. That means, inode's file operation field is copied into 'file' objects file operation field.

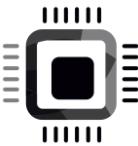
Copying 'inode's' fops into 'file' object's fops

Calling open method of default fops

And after that, you can see that, at this point open method of dummy default file operation is called. That means, this function get's called, chrdev_open get's called.

Here, let's explore this function in next slide.

Note: Partial code of the function is shown



/ fs / char_dev.c

```
* Called every time a character special file is opened
*/
static int chrdev_open(struct inode *inode, struct file *filp)
{
    const struct file_operations *fops;
    struct cdev *p;
    struct cdev *new = NULL;
    int ret = 0;

    spin_lock(&cdev_lock);
    p = inode->i_cdev;

    fops = fops_get(p->ops);
    if (!fops)
        goto out_cdev_put;

    replace_fops(filp, fops);
    if (filp->f_op->open) {
        ret = filp->f_op->open(inode, filp);
        if (ret)
            goto out_cdev_put;
    }

    return 0;
}
```

Note: Partial code of the function is shown

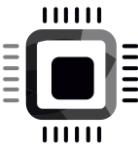
first the cdev field is checked. Inode cdev filed. Initially, it will be null actually, right? Because, this function initializes i_rdev and i_fop. It doesn't initialize i_cdev.

This is a dummy open which calls your driver's real open method

here complete code of this function is not present, it is given in lecture - section -3, part - 40
time= 16:00

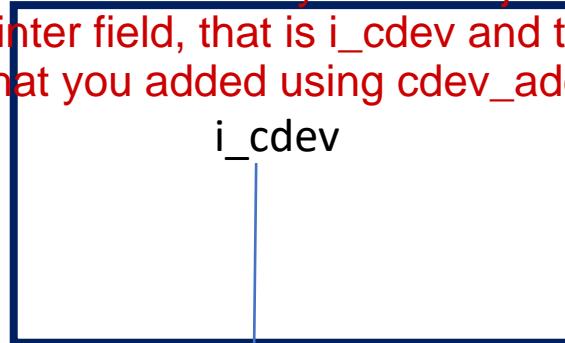
path to complete info of chrdev_open =
root/fs/char_dev.c

the final aim of this default chrdev_open is to call the drivers open method.



User space

Inode object
There will be a memory Inode object, and in memory Inode object has a pointer field, that is `i_cdev` and this will be pointing to your `cdev` object what you added using `cdev_add`.

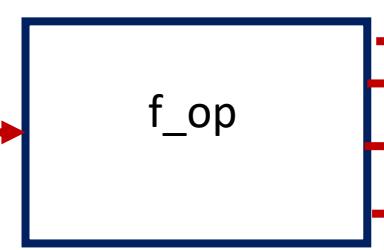
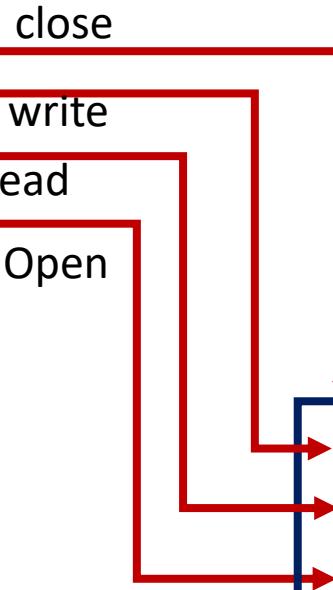
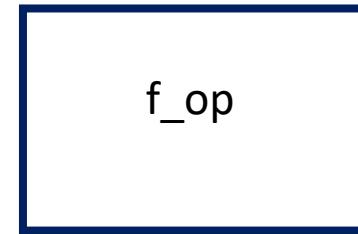


And this `cdev` object has `f_ops` field.
`F_ops`

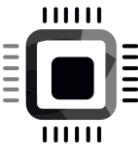
This file operation field is pointing to your drivers file operation.

That file operation field is copied into file objects file operation(`f_op`) field.

And this is a file object which controls an opened file.
File object

**File object****Driver**

From here the open, read, write, close, all system calls will get connected to your drivers methods.
When you use the open system call on the device file, it returns. file descriptor, `fd` is actually a reference to the opened file object.



/ include / linux / fs.h

```

struct file {
    union {
        struct llist_node          fu_llist;
        struct rcu_head            fu_rcuhead;
    } f_u;
    struct path                 f_path;
    struct inode                *f_inode;
    const struct file_operations *f_op;

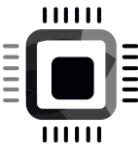
    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t                  f_lock;
    enum rw_hint                f_write_hint;
    atomic_long_t                f_count;
    unsigned int                 f_flags;
    fmode_t                      f_mode;
    struct mutex                 f_pos_lock;
    loff_t                        f_pos;
    struct fown_struct           f_owner;
    const struct cred             *f_cred;
}

```

When a process opens the file, the VFS initializes the **f_op** field of the new file object with the address stored in the inode so that further calls to file operations can use these functions.

The main information stored in a file object is the **current file offset**—the current position in the file from which the next operation will take place.

Because several processes may access the same file concurrently, the file pointer must be kept in the file object rather than the inode object.



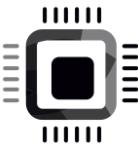
Summary

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

- 1) user invokes open system call on the 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 operations 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 a real file operations of your driver) gets copied to **file->f_op**
- 7) **file->f_op->open** method gets called (read open method of your driver)



2. Open method

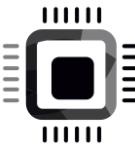
```
int pcd_open(struct inode *inode, struct file *filp)
{
    return 0;
}
```

The diagram illustrates the parameters of the `pcd_open` function. Two blue boxes with white text are positioned above the function definition. A red arrow points from the left box to the `inode` parameter in the first line of the function. Another red arrow points from the right box to the `filp` parameter in the same line.

Return :

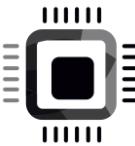
0 if open is successful

Negative error code if open fails



What you do inside the open method ?

- Initialize the device or make device respond for read/write access
- Detect device initialization errors
- Check open permission (O_RDONLY, O_WRONLY, O_RDWR)
- Identify the device being opened using minor number
- Prepare device private data structure if required
- Update f_pos if required
- Open method is optional. If not provided , open will always succeed and driver is not notified.



3.Close

When close is issued, the VFS releases the file object. And the release method is called when the last reference to an open file is closed. That is, when the `f_count` field of the file object becomes 0.

User space

`close(fd);`



`/dev/pcd`

Whenever a close is issued on the fd, that may not trigger the release method immediately. The release method is triggered only when all the references to an open file is closed. The VFS tracks this by using `f_count` field of the file object.

Kernel space

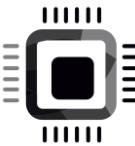


Releases the file object.
Called when the last reference to an open file is closed—that is, when the `f_count` field of the file object becomes 0.

```
int pcd_release(struct inode *inode, struct file *filp)
{
}
```

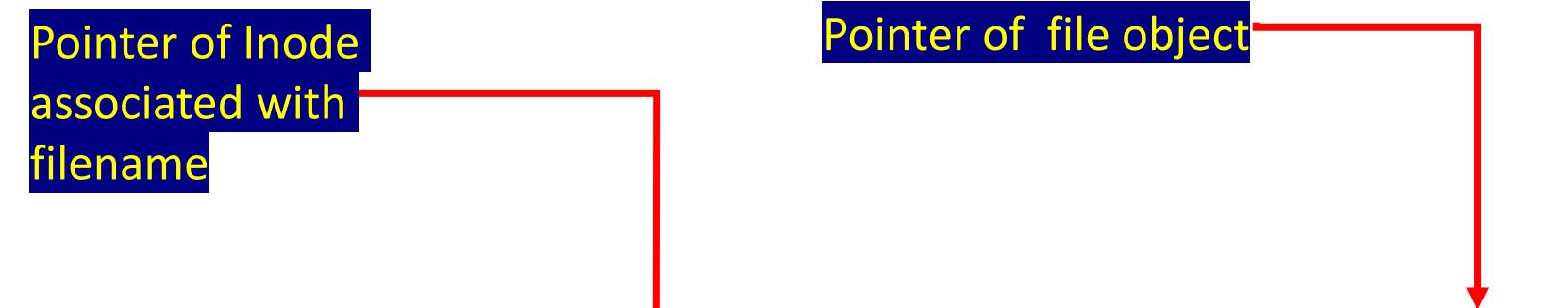
So, whenever close is issued, the `f_count` field of the file object decrements. When that reaches to 0, then only the release method gets called in your driver.

pcd driver

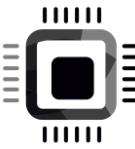


Release method

it passes 2 arguments Pointer of Inode associated with the device file and pointer of file object. And we will be using this name pcd_release in our character driver file.

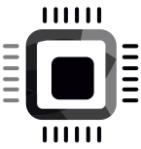


```
int pcd_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

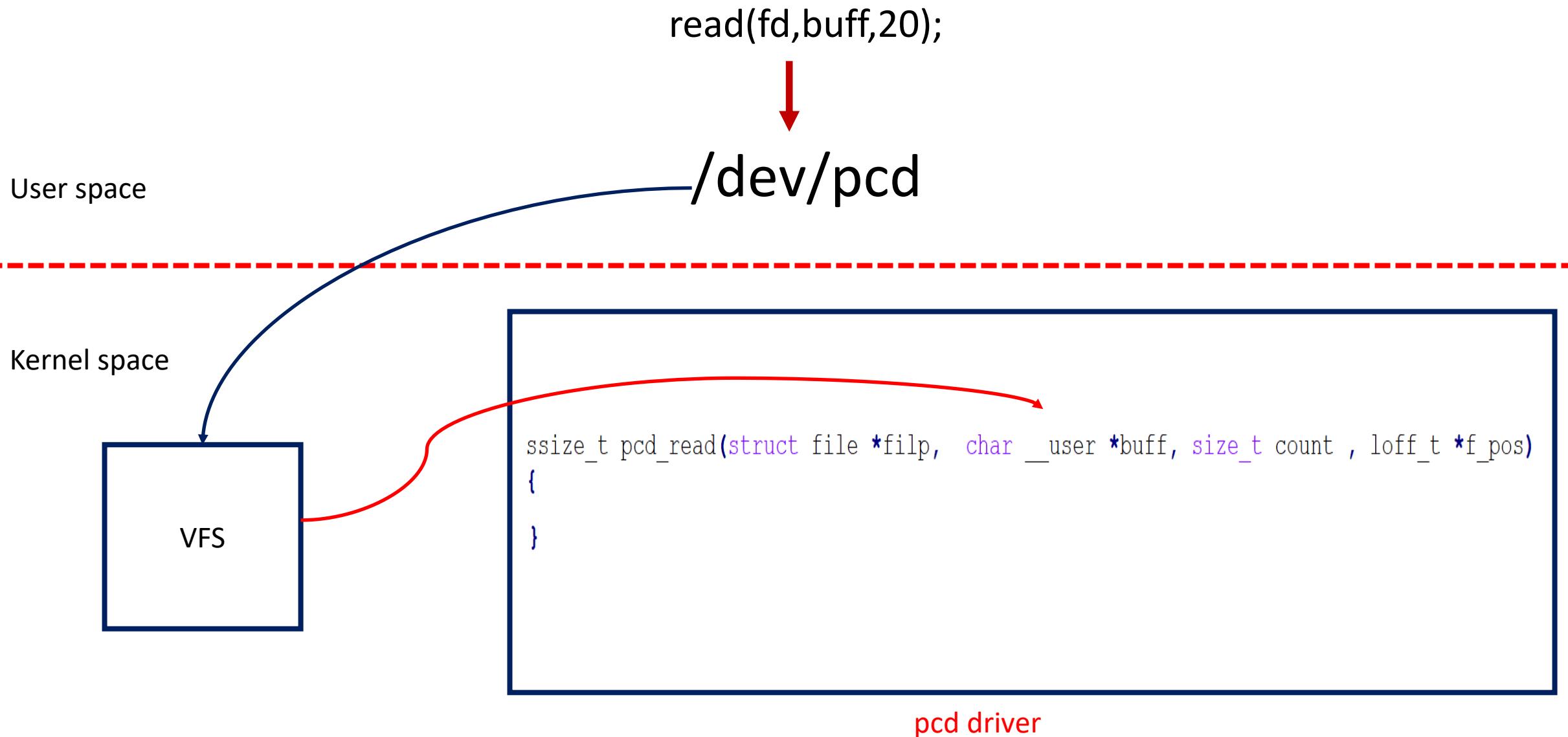


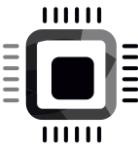
Release method

- In release method the driver can do reverse operation of what open had done.
 - E.g. if open method brings the device out of low power mode, then release method may send the device back to the low power mode.
 - Basically you should leave the device in its default state, the state which was before the open call.
 - Free any data structures allocated by the open method
- Return 0 on success . Negative error code if any errors
 - For example you try to de-initialize the device and the device doesn't respond

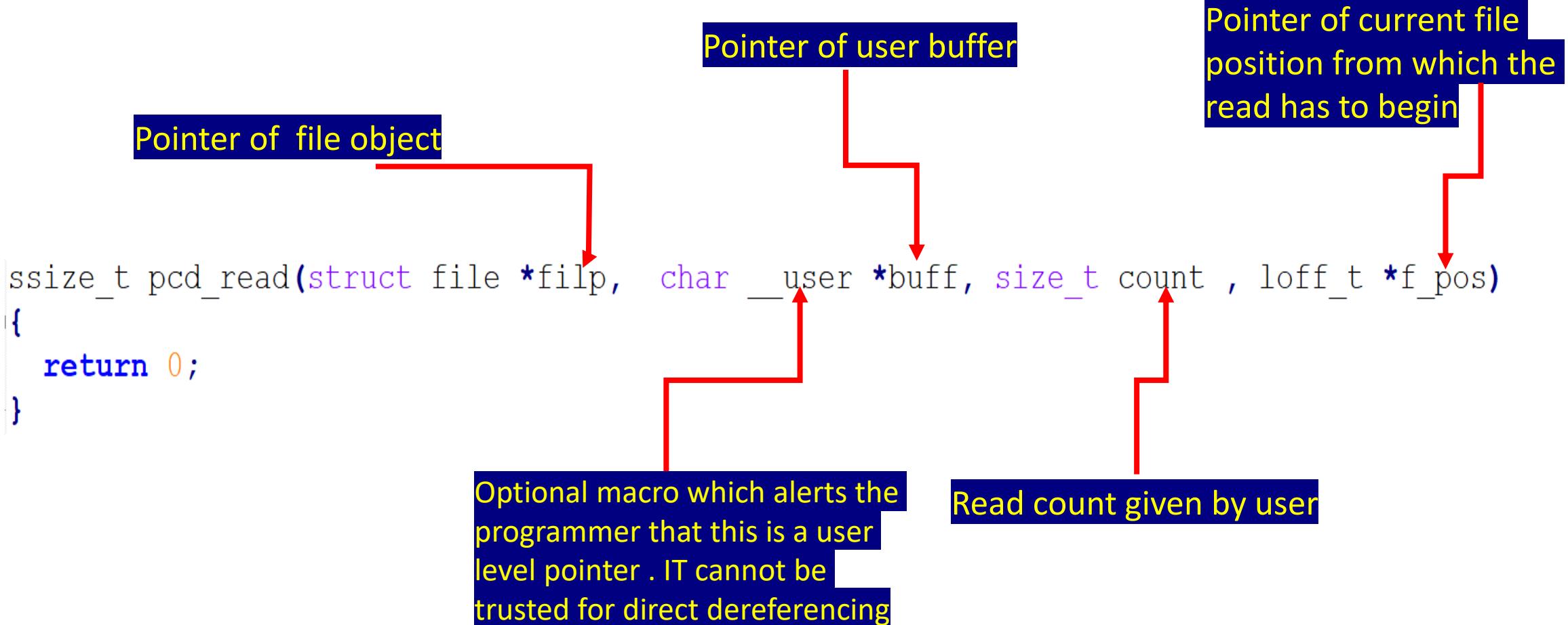


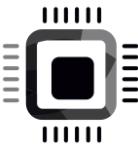
4. Read





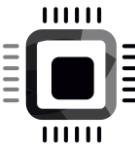
Read method





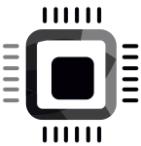
Read method

- Read ‘**count**’ bytes from a device starting at position ‘**f_pos**’.
- Update the ‘**f_pos**’ by adding the number bytes successfully read
- Return number of bytes successfully read
- Return 0 if there is no bytes to read (EOF)
- Return appropriate error code (-ve value) if any error
- A return value less than ‘**count**’ does not mean that an error has occurred.

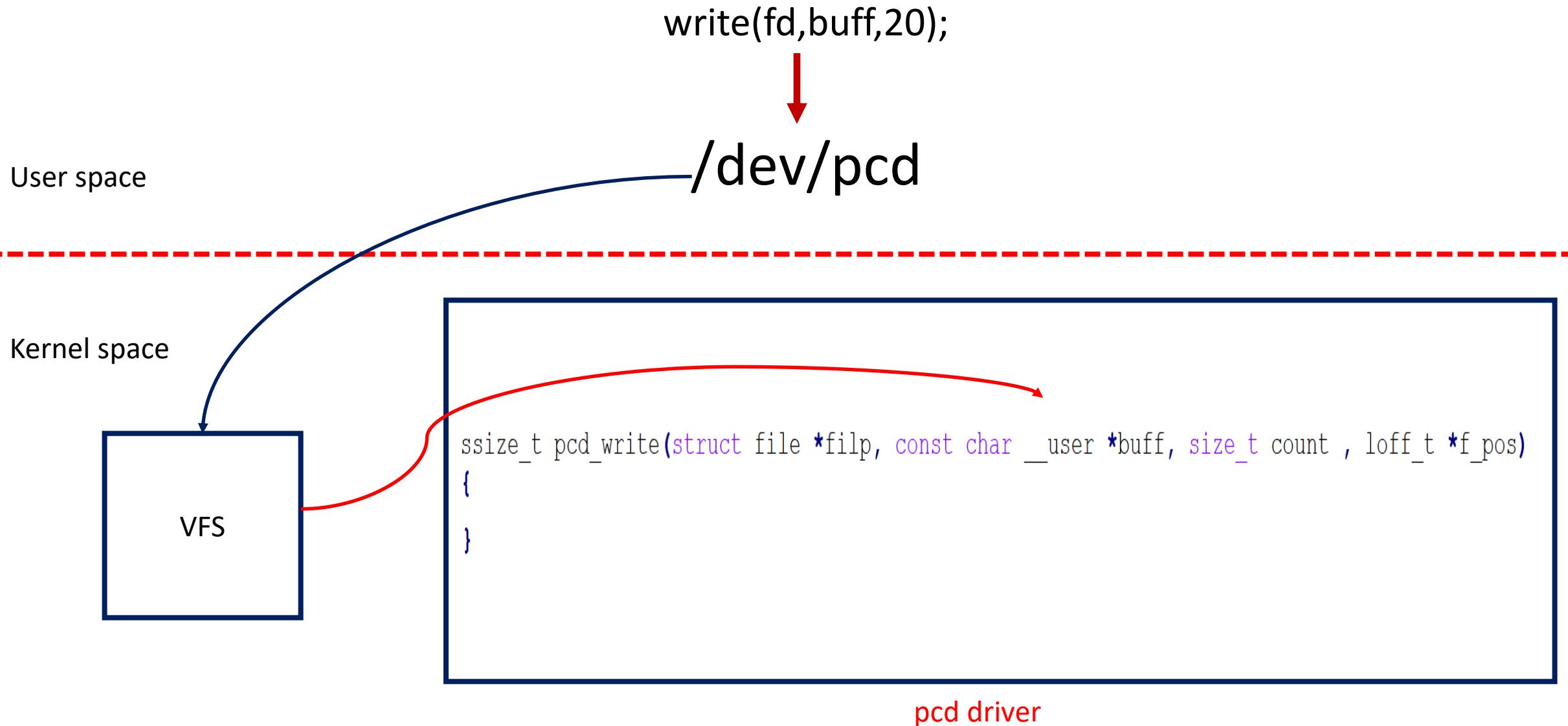


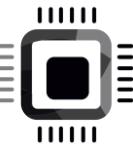
__user macro

- It's a macro used with user level pointers which tells the developer not to trust or assume it as a valid pointer to avoid kernel faults.
- Never try to dereference user given pointers directly in kernel level programming . Instead use dedicated kernel functions such as `copy_to_user` and `copy_from_user`
- GCC doesn't care whether you use `__user` macro with user level pointer or not. This is checked by `sparse` , a semantic checker tool of linux kernel to find possible coding faults .



5. Write



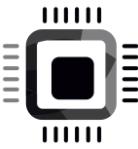


Write method

```
ssize_t pcd_write(struct file *filp, const char __user *buff, size_t count , loff_t *f_pos)  
{  
    return 0;  
}
```

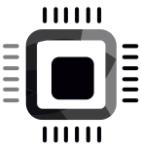
Diagram illustrating the parameters of the pcd_write function:

- Pointer of file object (red arrow pointing to `struct file *filp`)
- Pointer of user buffer (red arrow pointing to `const char __user *buff`)
- write count given by user (red arrow pointing to `size_t count`)
- Pointer of current file position from which the write has to begin (red arrow pointing to `loff_t *f_pos`)

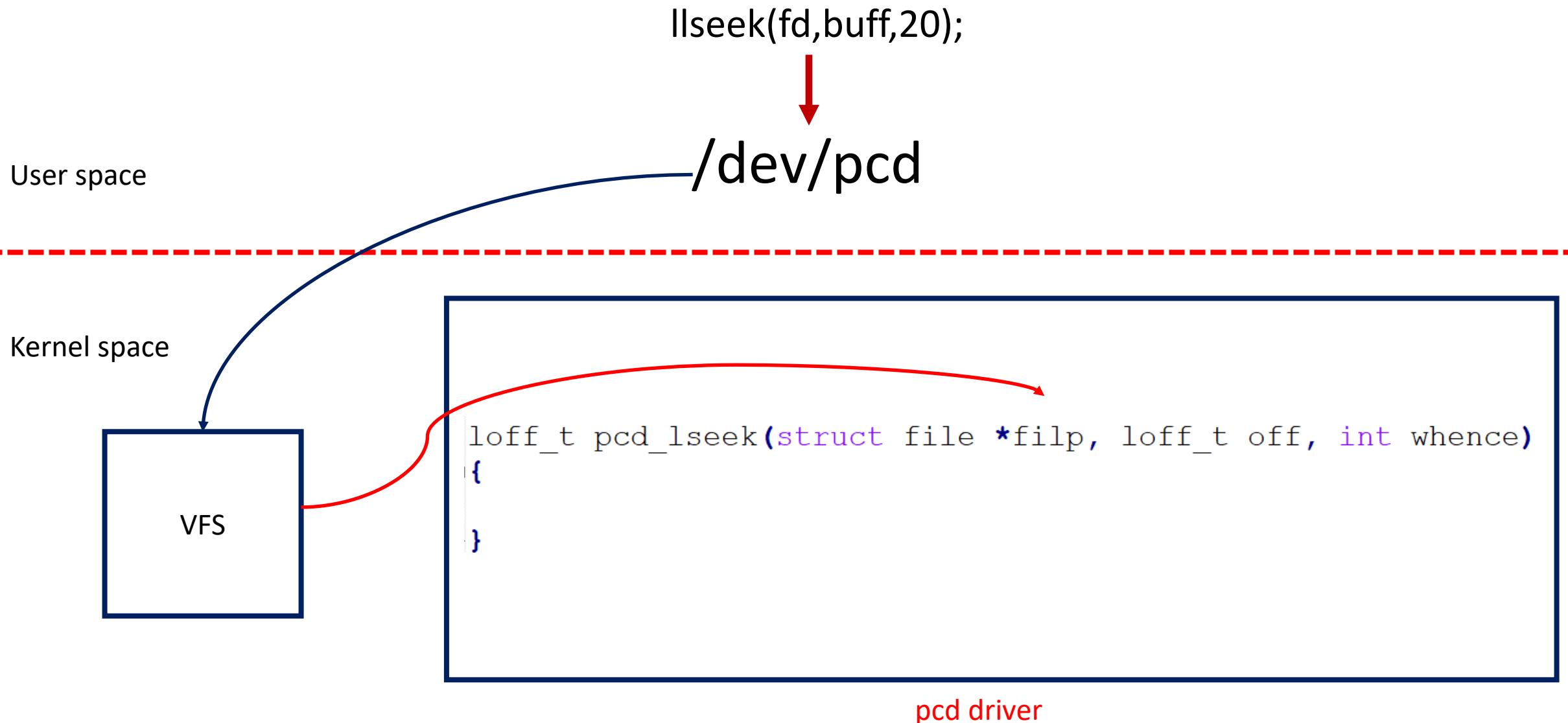


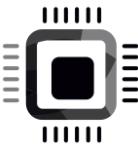
Write method

- Write ‘**count**’ bytes into the device starting at position ‘**f_pos**’.
- Update the ‘**f_pos**’ by adding the number bytes successfully written
- Return number of bytes successfully written
- Return appropriate error code (-ve value) if any error



6. lseek





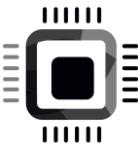
lseek

```
Pointer of file object
loff_t pcd_lseek(struct file *filp, loff_t off, int whence)
{
    return 0;
}
Origin
Offset value
```

SEEK_SET The file offset is set to '*off*' bytes.

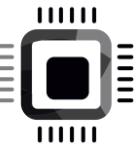
SEEK_CUR The file offset is set to its current location plus '*off*' bytes.

SEEK_END The file offset is set to the size of the file plus '*off*' bytes.

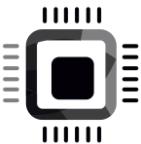


lseek method

- In the lseek method, driver should update the file pointer by using ‘offset’ and ‘whence’ information
- The lseek handler should return , newly updated file position or error



Character driver clean-up function



Kernel APIs and utilities to be used in driver code

Creation

alloc_chrdev_region();

cdev_init();
cdev_add();

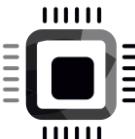
class_create();
device_create();

Deletion

unregister_chrdev_region();

cdev_del();

class_destroy();
device_destroy();



Remove a device that was created with device_create()

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

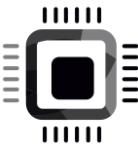
pointer to the struct class that this
device was registered with

dev_t of the device that was previously registered

Destroys a struct class structure

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

pointer to the struct
class that is to be
destroyed



Remove cdev registration from the kernel VFS

```
void cdev_del(struct cdev *p)
```

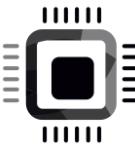
cdev structure to be removed

Unregister a range of device numbers

```
void unregister_chrdev_region(dev_t from, unsigned count);
```

the first in the
range of numbers
to unregister

number of
device numbers
to unregister



goto statement for error handling

```
if ( try_something_1 != Err)
{
    if(try_something_2 != Err)
    {
        if(try_something_3 != Err)
        {

        }else
        {
            undo_try_something_2;
            undo_try_something_1;
        }
    }else
    {
        undo_try_something_1;
    }
}
```

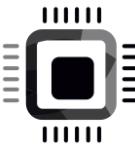
```
if(try_something_1 == Err)
    goto err1;

if(try_something_2 = Err)
    goto err2;

if(try_something_3 == Err)
    goto err3;

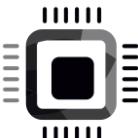
err3:
    undo_try_something_2;
err2:
    undo_try_something_1;
err1:
    return ret;
```





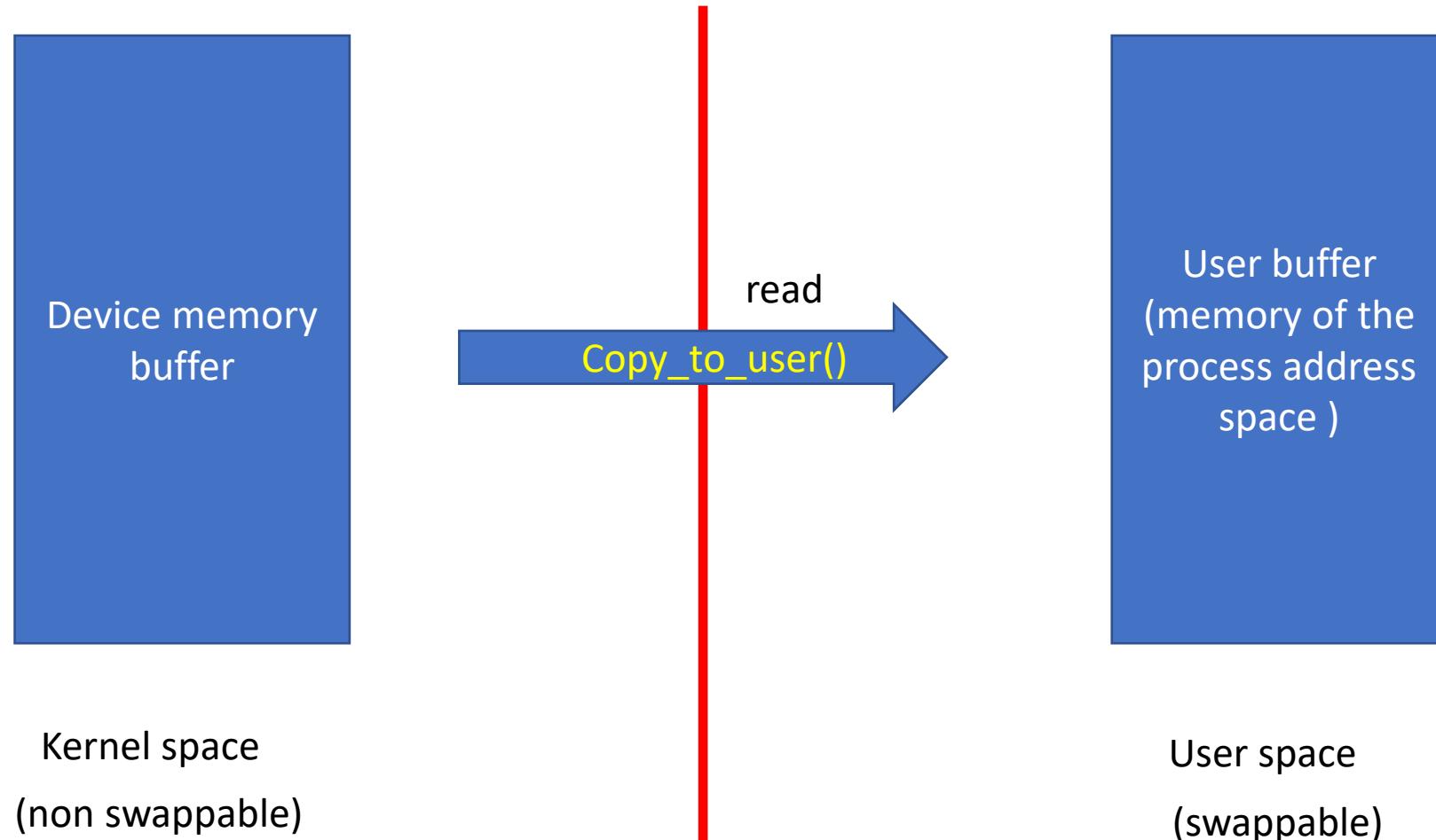
Error handling of pointers during kernel function return

- You can use some kernel macros to deal with return of error pointers by kernel functions.
- The below macros help to understand what made kernel function to fail.
 - `IS_ERR()`
 - `PTR_ERR()`
 - `ERR_PTR()`
- These macros can be found in include include/linux/err.h

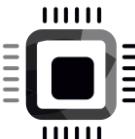


section 4

Read method implementation



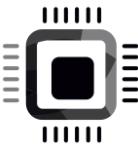
the user level process execute read system call to read from a file, in our case file is not regular file but a device file, which handles pseudo character device, pseudo character device is nothing but a array in memory, so i will call it device memory buffer. whenever user program issues read system call on our device file we should transfer data from our device memory buffer to the user buffer. data copy happens from kernel side to user side, that is read, but write would be in reverse direction.



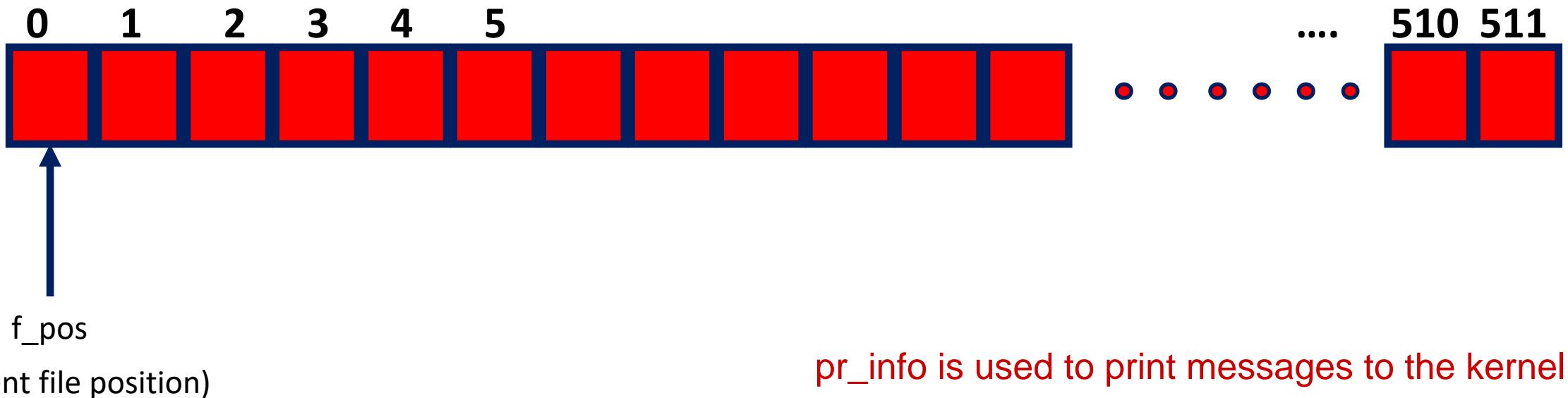
Read method implementation

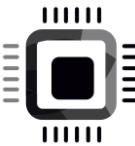
```
ssize_t pcd_read(struct file *filp, char __user *buff, size_t count , loff_t *f_pos)
{
    return 0;
}
```

- 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 the 'count' . **count = DEV_MEM_SIZE - f_pos**
- 2) copy 'count' number of bytes from device memory to user buffer
- 3) Update the **f_pos**
- 4) Return number of bytes successfully read or error code
- 5) If **f_pos** at EOF, then return 0;



Pseudo character device memory access





/ include / linux / fs.h

```

struct file {
    union {
        struct llist_node          fu_llist;
        struct rcu_head            fu_rcuhead;
    } f_u;
    struct path                 f_path;
    struct inode                *f_inode;
    const struct file_operations *f_op;

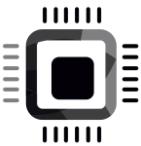
    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t                  f_lock;
    enum rw_hint                 f_write_hint;
    atomic_long_t                f_count;
    unsigned int                  f_flags;
    fmode_t                      f_mode;
    struct mutex                 f_pos_lock;
    loff_t                        f_pos;
    struct fown_struct           f_owner;
    const struct cred             *f_cred;
}

```

When a process opens the file, the VFS initializes the **f_op** field of the new file object with the address stored in the inode so that further calls to file operations can use these functions.

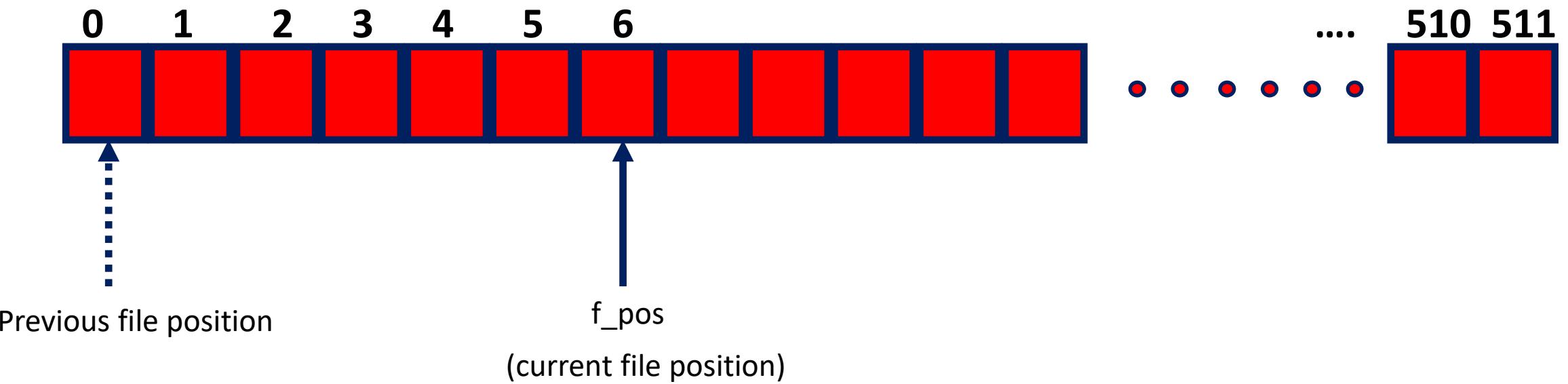
The main information stored in a file object is the **current file offset**—the current position in the file from which the next operation will take place.

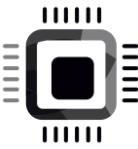
Because several processes may access the same file concurrently, the file pointer must be kept in the file object rather than the inode object.



Pseudo character device memory access

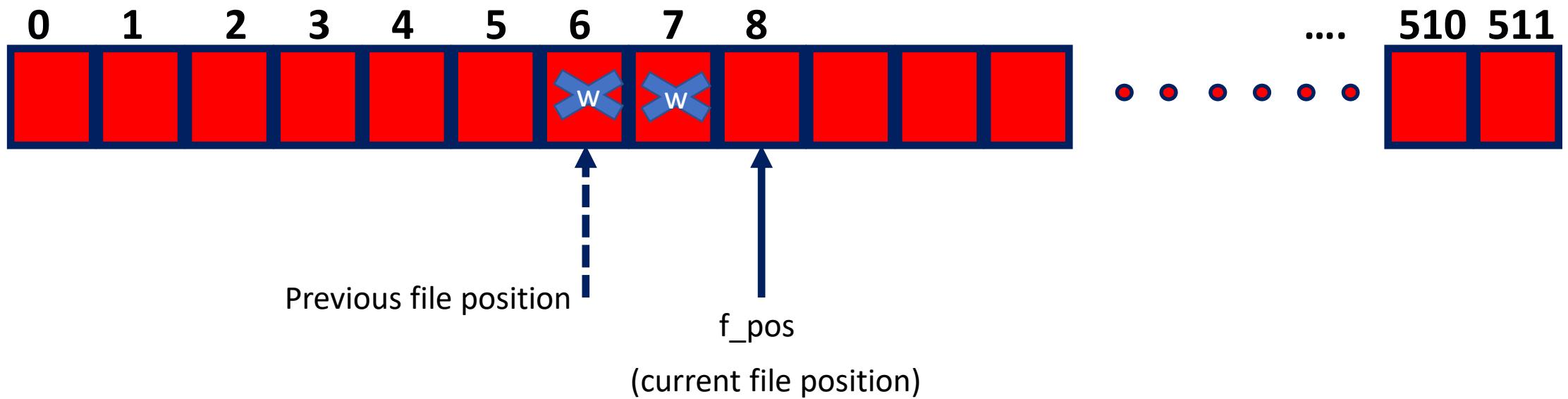
After read of 6 bytes

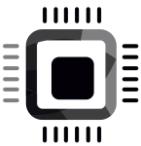




Pseudo character device memory access

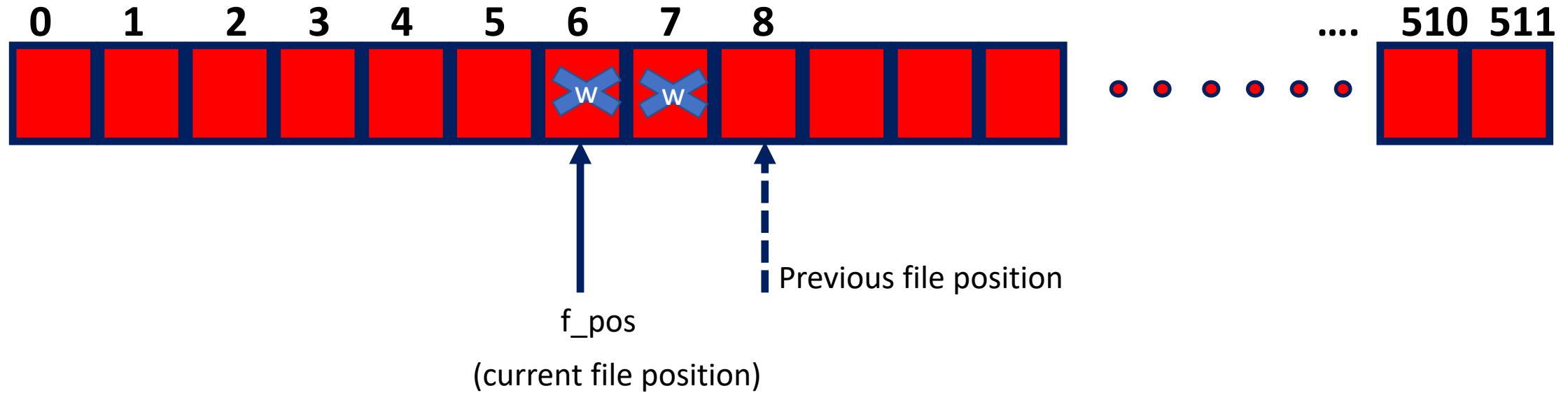
After write of 2 bytes

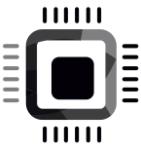




Pseudo character device memory access

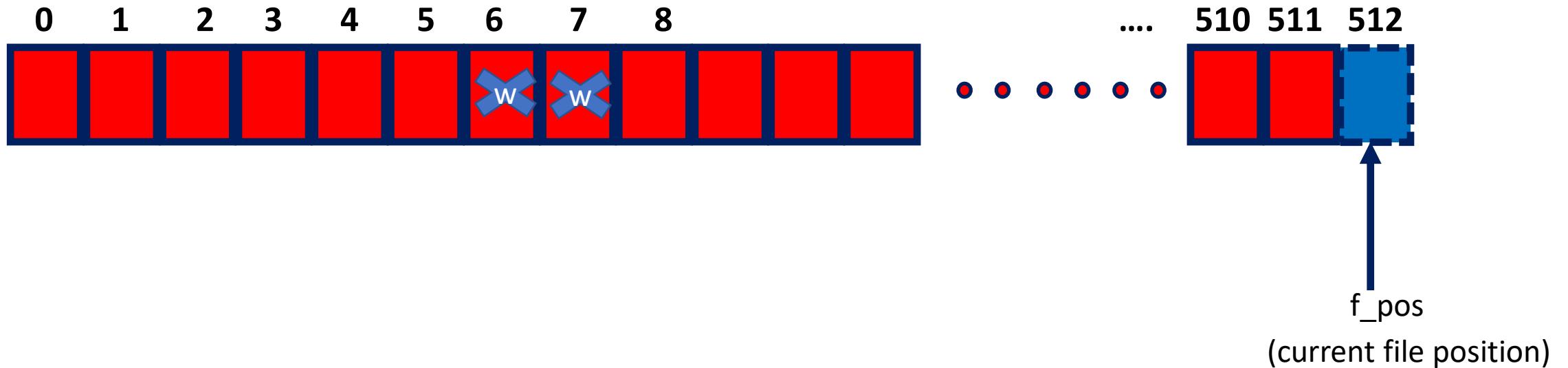
After lseek of -2 from current position

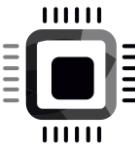




Pseudo character device memory access

EOF Scenario



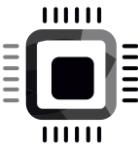


Data copying between Kernel space and user space

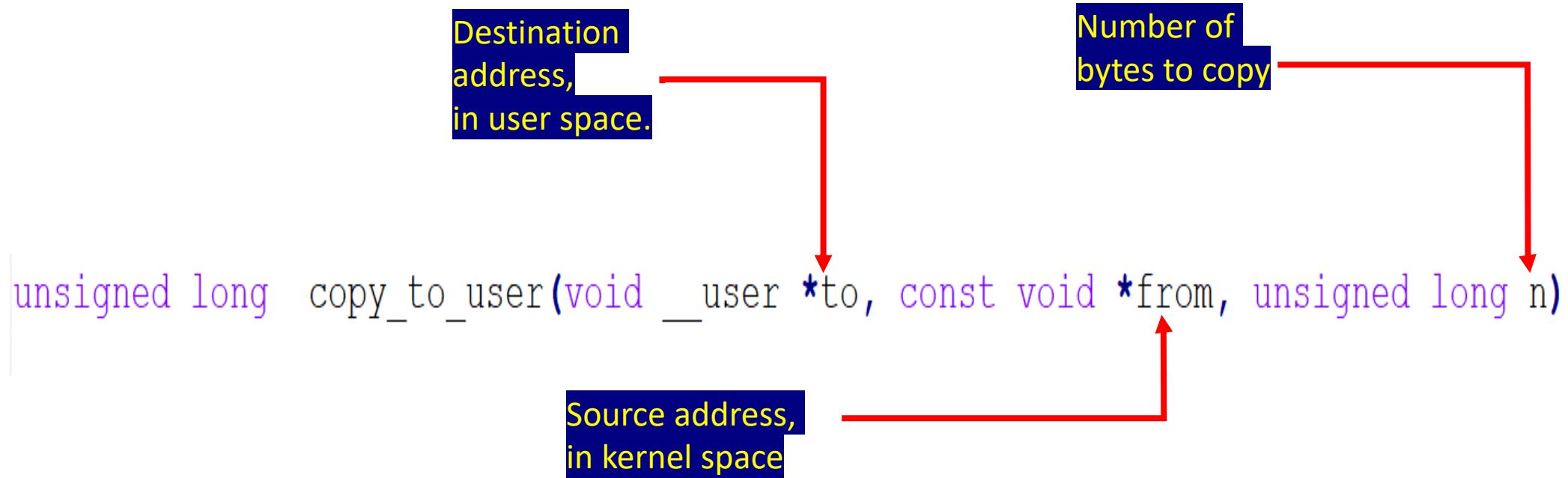
- **copy_to_user()**
- **copy_from_user()**

Role of these two functions

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

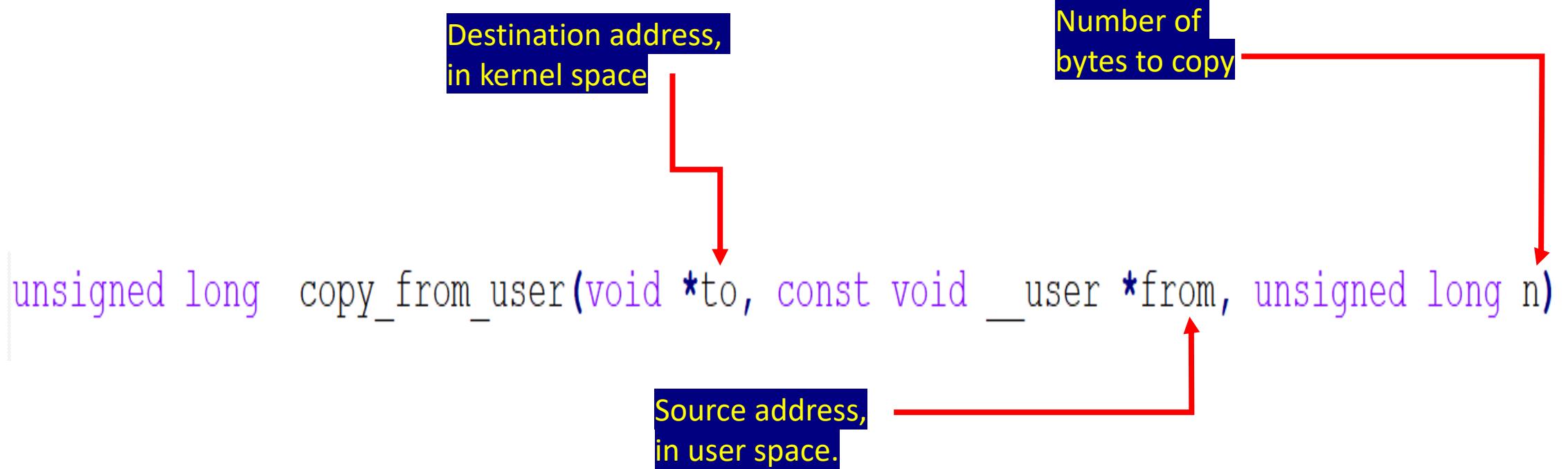
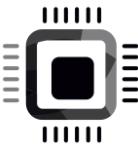


for copy_to_user and copy_from_user we use a header file #include<linux/uaccess.h>



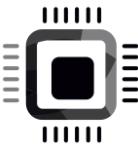
Returns 0 on success or number of bytes that could not be copied

If this function returns non zero value , you should assume that there was a problem during data copy.So return appropriate error code (-EFAULT)



Returns 0 on success or number of bytes that could not be copied

If this function returns non zero value , you should assume that there was a problem during data copy.So return appropriate error code (-EFAULT)



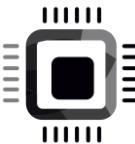
lseek

```
Pointer of file object
loff_t pcd_lseek(struct file *filp, loff_t off, int whence)
{
    return 0;
}
Origin
Offset value
```

SEEK_SET The file offset is set to '*off*' bytes.

SEEK_CUR The file offset is set to its current location plus '*off*' bytes.

SEEK_END The file offset is set to the size of the file plus '*off*' bytes.

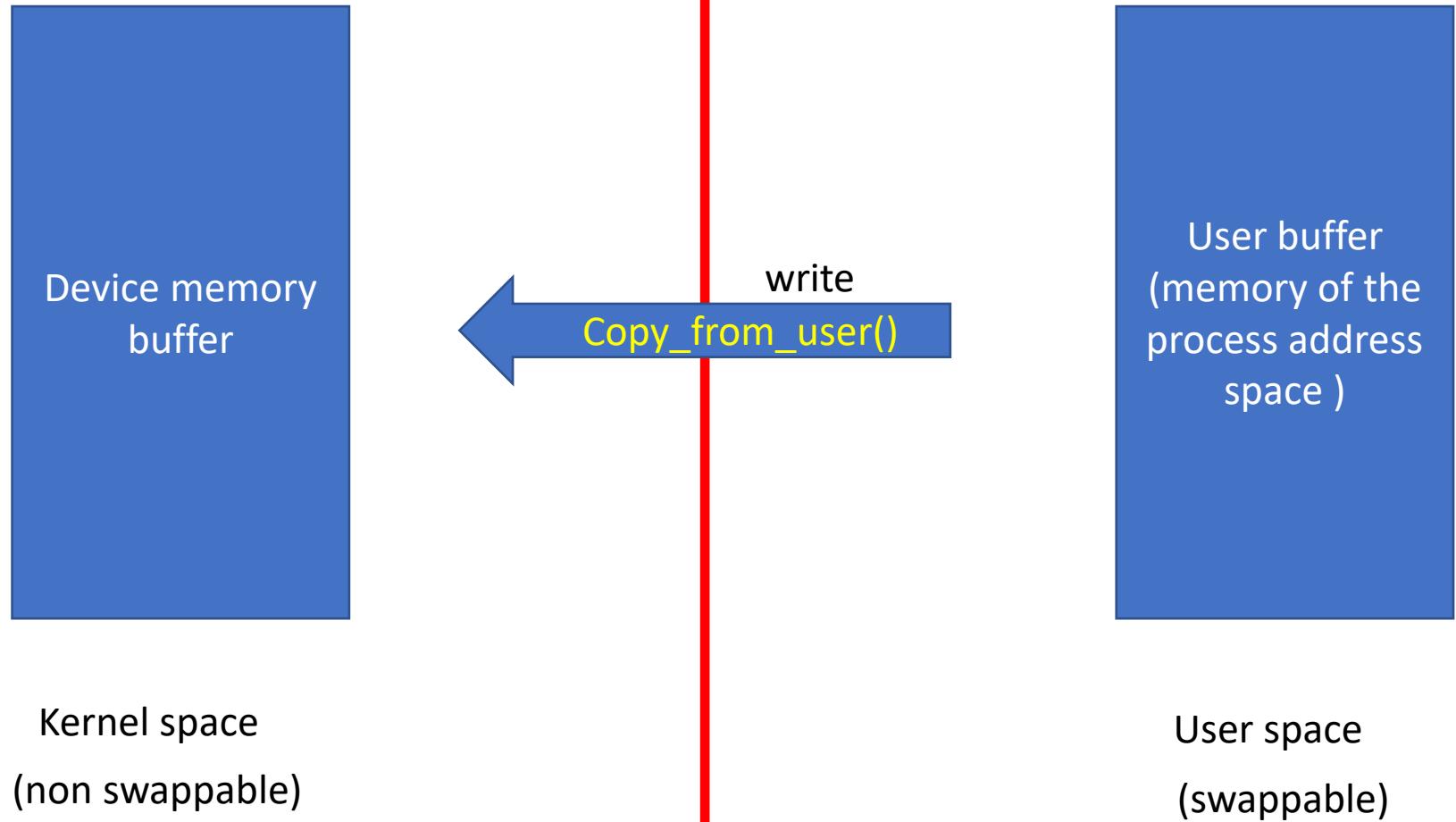
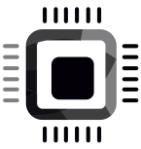


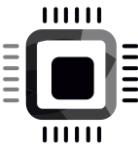
Error codes and descriptions

Preprocessor define	Description
E2BIG	Argument list too long
EACCES	Permission denied
EAGAIN	Try again
EBADF	Bad file number
EBUSY	Device or resource busy
ECHILD	No child processes
EDOM	Math argument outside of domain of function
EEXIST	File already exists

[include/uapi/asm-generic/errno-base.h](#)

Preprocessor define	Description
EFAULT	Bad address
EFBIG	File too large
EINTR	System call was interrupted
EINVAL	Invalid argument
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENFILE	File table overflow
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOMEM	Out of memory
ENOSPC	No space left on device
ENOTDIR	Not a directory
ENOTTY	Inappropriate I/O control operation
ENXIO	No such device or address
EPERM	Operation not permitted

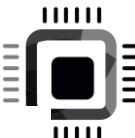




Write method implementation

```
ssize_t pcd_write(struct file *filp, const char __user *buff, size_t count , loff_t *f_pos)
{
    return 0;
}
```

- 1) check the user requested 'count' value against **DEV_MEM_SIZE** of the device .
 - if **current_file_position + count > MAX_SIZE**
 - Adjust the 'count' . **count = MAX_SIZE - current_file_position**
- 2) copy 'count' number of bytes from user buffer to device memory
- 3) Update the **current_file_positon**
- 4) Return number of bytes successfully written or error code



```
loff_t pcd_lseek(struct file *filp, loff_t off, int whence)
{
    return 0;
}
```

as the current file position is one of the member element of struct file
so we get current file position from there.

If whence = SEEK_SET
flip->f_pos = off

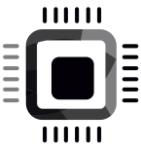
If whence = SEEK_CUR
flip->f_pos = flip->f_pos + off

if whence = SEEK_END
flip->f_pos = DEV_MEM_SIZE + off

if whence is SEEK_SET then cureent file position
will be set to offset

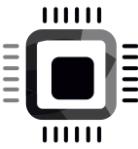
here current file position will be set to previous file
position + offset

here current file position will be set to end of the file.



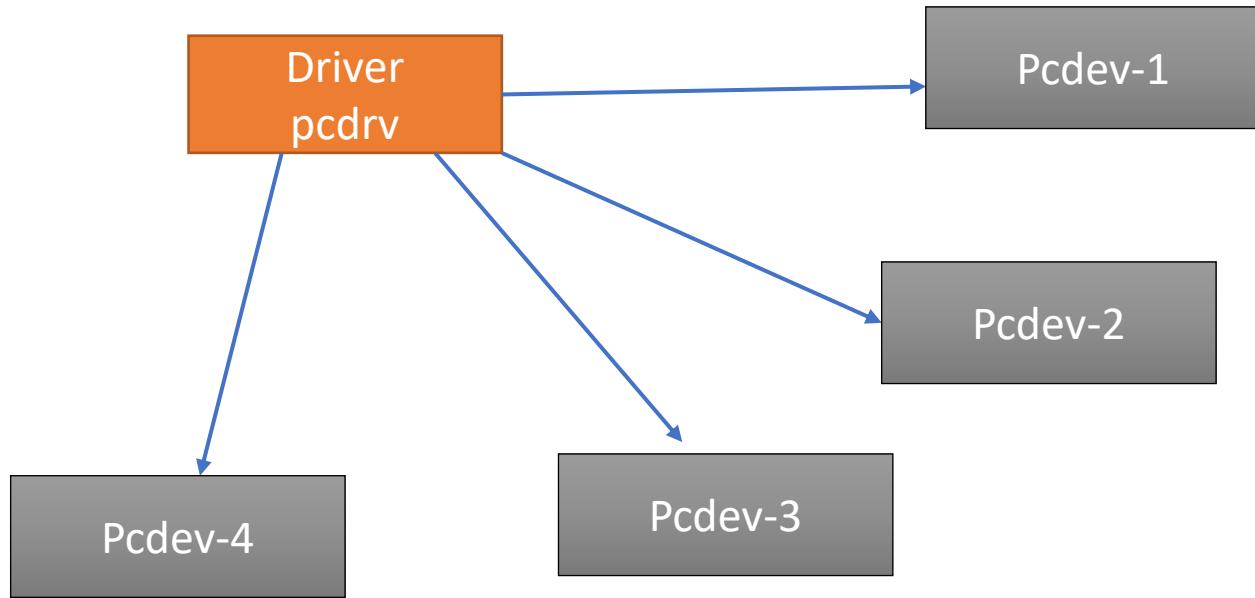
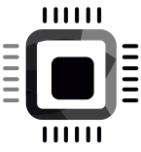
section 5

Pseudo character driver with multiple devices

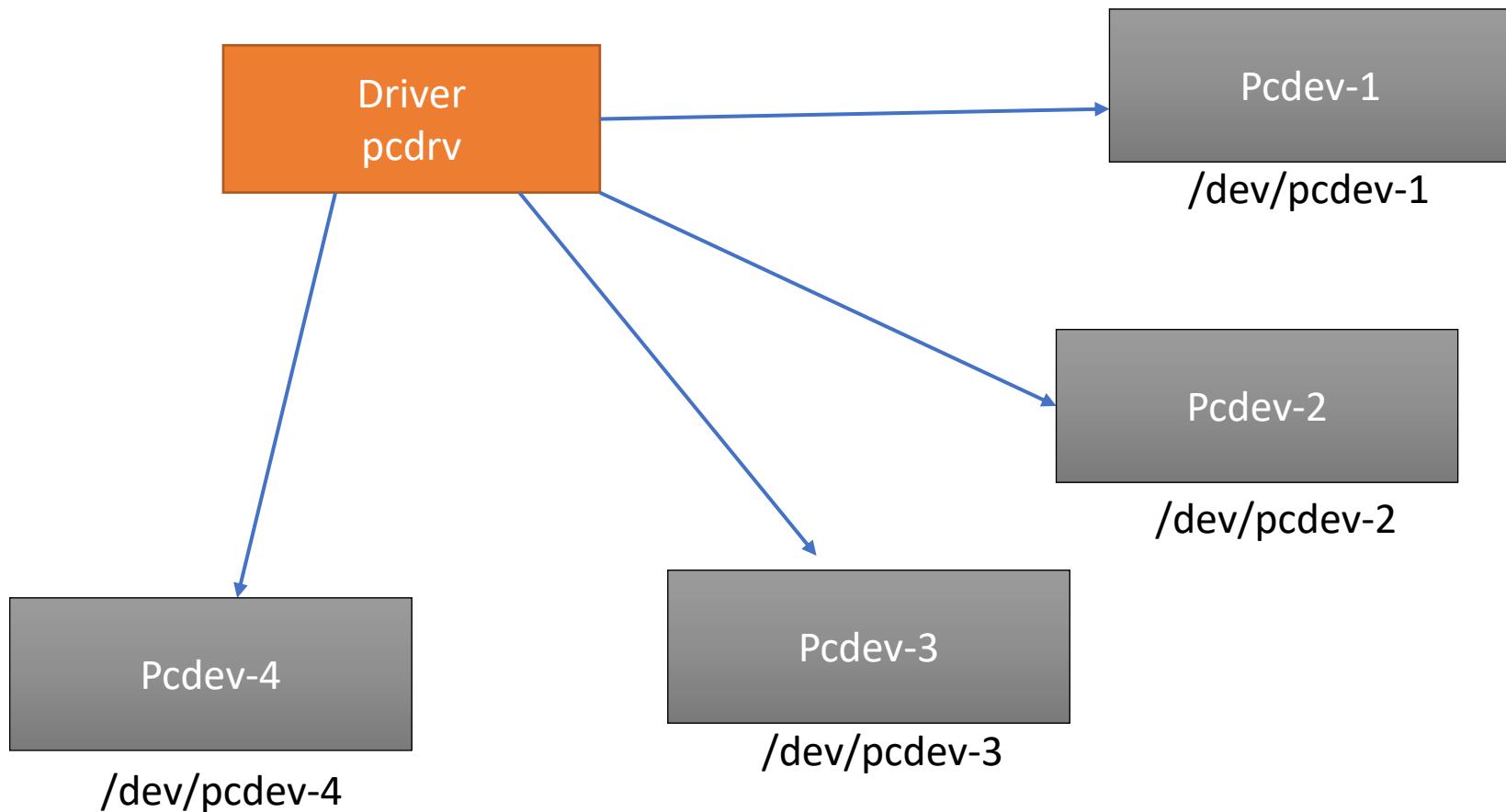
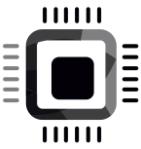


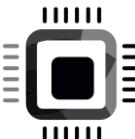
Exercise

- Modify the previous pseudo character device driver to support four pseudo character devices.
- Implement open, release, read, write, lseek driver methods to handle user requests.

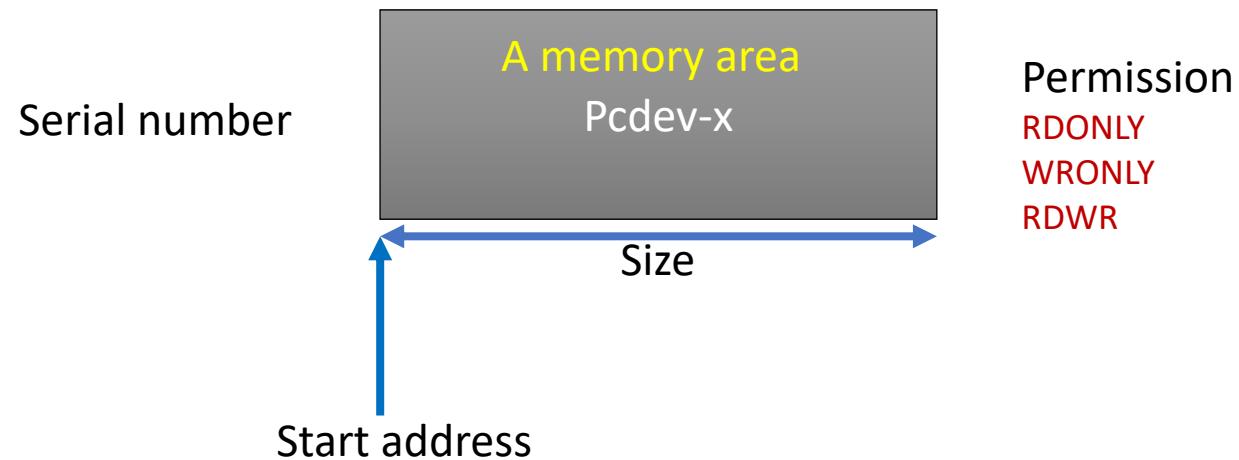


- The driver, which we about to write, should manage 4 devices.
- There will be single driver managing 4 devices.
- There will be only 1 implementation of open, read, write, release, and Iseek driver methods.
- That also means the driver should first determine which device is being accessed from the user space to fulfill the request



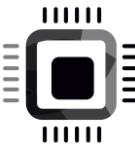


Private data of a device(Device-specific information)



Every device has its own private data

here in our case a device is nothing but a memory area. What are the attributes or device specific information of our device? So, the first thing could be a serial number., Every device may have its own serial number.,A serial number could be a collection of characters or a string data. Since, our device is a memory area, so the starting address of the memory area is very important., That could be another device specific information.,What is the size of that memory area, and the permission of that memory, area also become device specific information. That means, in our case, there are 4 device specific information. The permission could be read only, write only, and read-write.



echo "Good morning" > /dev/pcdev-1

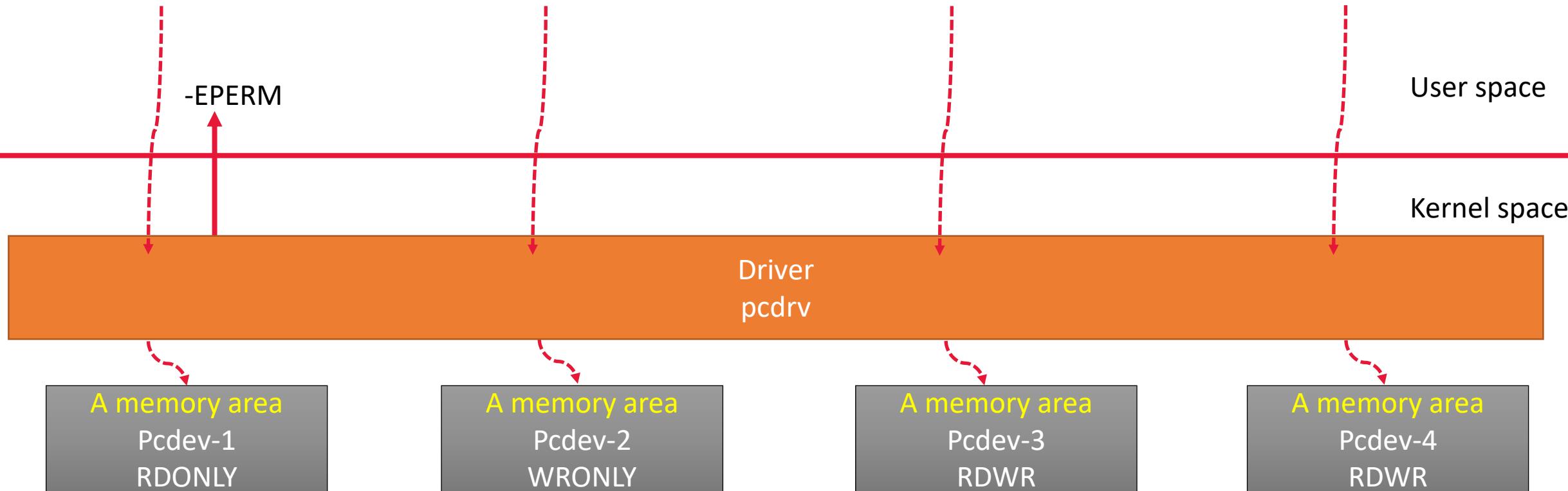
echo "Good morning" > /dev/pcdev-3

/dev/pcdev-1

/dev/pcdev-2

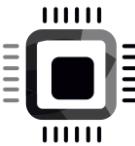
/dev/pcdev-3

/dev/pcdev-4



This is how your driver is going to manage a 4 different devices. Remember that, in this exercise, every device represents a unique memory area. You can create a the memory area either statically or by using dynamic memory allocation. But, let's stick to static allocation that is array based, because we have not yet discussed the dynamic memory allocation in the kernel space. The user space will see your device files Pcdv_1, Pcdv_2, Pcdv_3 and four. And the driver is going

Every device represents a unique memory area.



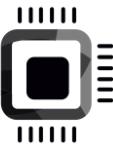
Device - Driver data structure

In the driver, we'll maintain two structures

Structure which holds
driver's private data

Structure which holds
device's private data

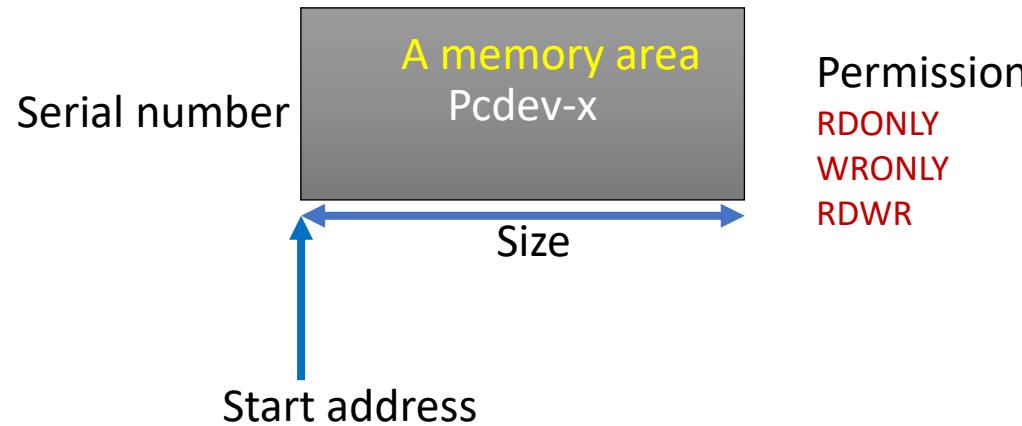
to manage 4 memory areas. The Pcd-1 will be read only in our case, and Pcd-2 will be write only. Pcd-3 and Pcd-4 will be read-write memory. This is the configuration we are going to use for this exercise. Let's say, there is a userspace application echo, which tries to write some data to Pcd-1. In that case, the driver should detect that and it should return the error code EPERM. That means, operation is not permitted. Because, this is read-only, right? In the same way, if an application tries to read from this device, then only the driver should detect that and it should return the appropriate error code. And for these two devices pcd3 and pcd4 the user space application can read and write. Let's say, there is a user space application echo tries to write some data to the pcd-3, then the driver should appropriately copy that data to the write device. Drivers should not put that data in the memory areas of other devices. That means, the driver should detect to which device the request is coming in.



Let me also discuss about the data structure we are going to use for this exercise. What will do for this driver is.

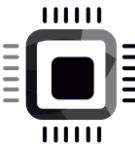
Structure to represent device's private data

In the driver, we will maintain 2 structures., The first structure is going to hold drivers private data and another structure is going to hold a device private data.



```
/*This structure represents device private data */
struct pcdev_private_data
{
    char *buffer;
    unsigned size;
    const char *serial_number;
    int perm;
    struct cdev cdev;
};
```

Let's understand a structure to represent device private data. As I explained in the previous slide, every device has its own private data., In our case, the pcdev device has couple of private data such as start address, size, permission, serial number, We will put all these data in a structure format. That's why, I'm going to include or I'm going to create this structure. **struct pcdev_private_data**,
 So, any private data if you think it is specific to device, then you are free to keep that in this structure. Will later see when we use synchronization related a kernel object such as pin logs and other things. That also if it is a device specific, then it goes into this structure.

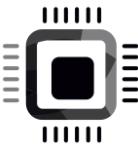


Structure to represent driver's private data

Let's move forward and now let's explore the structure to represent drivers private data. In our case, driver is the one who is going to handle 4 or N number of devices. That's why, I include only two member elements in the driver's private data structure. The first one will be total devices. Because, since driver is handling N number of devices, Now, the next member element is nothing but device private data instances. So, this driver is handling N number of devices. That means, each device should have its own pcdev_private data struct variable.

```
/*This structure represents driver private data */
struct pcdrv_private_data
{
    int total_devices;
    struct pcdev_private_data pcdev_data[NO_OF_DEVICES];
};
```

That's a reason I just created an array of device private data. Here, number of devices macro represents a number of devices. The reason for embedding pcdev_private_data. Inside the driver structure is because, driver is the one who is going to operate on the device specific data, isn't it? For example, driver is the one who is going to alter the memory address of a device to read or write data. That's a reason why we can create this array of struct pcdev_private_data inside this driver structure.



Pcdev-1

Pcdev-2

Pcdev-3

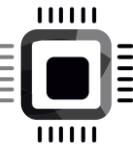
Pcdev-4

127:0 ← 127:1

127:2

127:3

```
alloc_chrdev_region(&device_number, 0, NO_OF_DEVICES, "pcdevs");(4)
```



```

struct file {
    union {
        struct llist_node fu_llist;
        struct rcu_head fu_rcuhead;
    } f_u;
    struct path f_path;
    struct inode *f_inode; /* cached value */
    const struct file_operations *f_op;

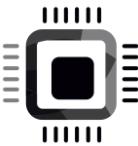
    /*
     * Protects f_ep_links, f_flags.
     * Must not be taken from IRQ context.
     */
    spinlock_t f_lock;
    enum rw_hint f_write_hint;
    atomic_long_t f_count;
    unsigned int f_flags;
    fmode_t f_mode;
    struct mutex f_pos_lock,
    loff_t f_pos;
    struct fown_struct f_owner;
    const struct cred *f_cred;
    struct file_ra_state f_ra;

    u64 f_version;
#ifdef CONFIG_SECURITY
    void *f_security;
#endif
    /* needed for tty driver, and maybe others */
    void *private_data;

#ifdef CONFIG_EPOLL
    /* Used by fs/eventpoll.c to link all the hooks to this file */
    struct list_head f_ep_links;
    struct list_head f_tfile_llink;
#endif /* #ifdef CONFIG_EPOLL */
    struct address_space *f_mapping;
    errseq_t f_wb_err;
}

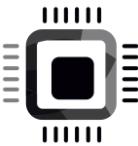
```

#define O_ACCMODE	00000003
#define O_RDONLY	00000000
#define O_WRONLY	00000001
#define O_RDWR	00000002

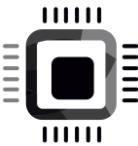


```
/*
 * flags in file.f_mode. Note that FMODE_READ and FMODE_WRITE must correspond
 * to O_WRONLY and O_RDWR via the strange trick in __dentry_open()
 */

/* file is open for reading */
#define FMODE_READ          ((__force fmode_t)0x1)
/* file is open for writing */
#define FMODE_WRITE         ((__force fmode_t)0x2)
/* file is seekable */
#define FMODE_LSEEK          ((__force fmode_t)0x4)
/* file can be accessed using pread */
#define FMODE_PREAD         ((__force fmode_t)0x8)
/* file can be accessed using pwrite */
#define FMODE_PWRITE        ((__force fmode_t)0x10)
/* File is opened for execution with sys_execve / sys_uselib */
#define FMODE_EXEC          ((__force fmode_t)0x20)
/* File is opened with O_NDELAY (only set for block devices) */
#define FMODE_NDELAY        ((__force fmode_t)0x40)
/* File is opened with O_EXCL (only set for block devices) */
#define FMODE_EXCL          ((__force fmode_t)0x80)
/* File is opened using open(..., 3, ..) and is writeable only for ioctl
   (specialy hack for floppy.c) */
#define FMODE_WRITE_IOCTL    ((__force fmode_t)0x100)
/* 32bit hashes as llseek() offset (for directories) */
#define FMODE_32BITHASH     ((__force fmode_t)0x200)
/* 64bit hashes as llseek() offset (for directories) */
#define FMODE_64BITHASH     ((__force fmode_t)0x400)
```



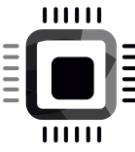
```
struct file {  
    union {  
        struct llist_node    fu_llist;  
        struct rcu_head      fu_rcuhead;  
    } f_u;  
    struct path      f_path;  
    struct inode     *f_inode;    /* cached value */  
    const struct file_operations  *f_op;  
  
    /*  
     * Protects f_ep_links, f_flags.  
     * Must not be taken from IRQ context.  
     */  
    spinlock_t          f_lock;  
    enum rw_hint       f_write_hint;  
    atomic_long_t      f_count;  
    unsigned int       f_flags;  
    fmode_t            f_mode;  
    struct mutex       f_pos_lock;  
    loff_t              f_pos;  
    struct fown_struct f_owner;  
    const struct cred   *f_cred;  
    struct file_ra_state f_ra;  
  
    u64                f_version;  
#ifdef CONFIG_SECURITY  
    void               *f_security;  
#endif  
    /* needed for tty driver, and maybe others */  
    void               *private_data;  
  
#ifdef CONFIG_EPOLL  
    /* Used by fs/eventpoll.c to link all the hooks to this file */  
    struct list_head    f_ep_links;  
    struct list_head    f_tfile_llink;  
#endif /* #ifdef CONFIG_EPOLL */  
    struct address_space  *f_mapping;  BHARATI SOFTWARE , CC BY-SA 4.0 , 2020  
    errseq_t           f_wb_err;
```



container_of macro

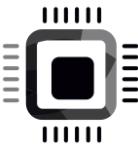
```
/**  
 * container_of - cast a member of a structure out to the containing structure  
 * @ptr:    the pointer to the member.  
 * @type:   the type of the container struct this is embedded in.  
 * @member: the name of the member within the struct.  
 *  
 */  
#define container_of(ptr, type, member) ({  
    void * __mptr = (void *)(ptr);  
    BUILD_BUG_ON_MSG(!__same_type(*(ptr), ((type *)0)->member) &&  
        !__same_type(*(ptr), void),  
        "pointer type mismatch in container_of()");  
    ((type *)(__mptr - offsetof(type, member))); })
```

/include/linux/kernel.h



container_of

- Container_of macro helps you to get the address of the containing structure by taking an address of its member element .
- As its name indicates it gives you the “container” address of the member element of a struct
- It takes three arguments – a *pointer*, *type* of the container, and the *name* of the member the pointer refers to.

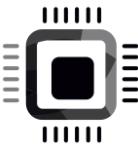


Pointer to the member

name of the member
the 'ptr' refers to

#define container_of(ptr, type, member)

Type of the container struct in
which 'member' is embedded in



```
struct some_data
{
    char    a;
    int     b;
    char    c;
    int     d;
};
```

```
struct some_data data;
```

```
int init()
{
```

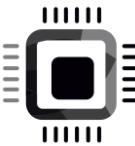
```
    data.a = 10;
    data.b = 5;
    data.c = 'a';
    data.d = 100;
```

```
//passing member's address
get_container( &data.c);
```

```
return 0;
```

```
}
```

```
// a function may be defined in some other file
void get_container(char *ptr)
{
    //by using the 'ptr', get the address of 'data'(container)
    struct some_data *pdata = container_of(ptr,struct some_data,c);
```



```
struct some_data
{
    char    a;
    int     b;
    char    c;
    int     d;
};
```

```
struct some_data data;
```

```
int init()
{
```

```
    data.a = 10;
    data.b = 5;
    data.c = 'a';
    data.d = 100;
```

```
//passing member's address
get_container( &data.c);
```

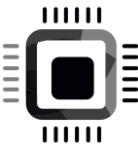
```
return 0;
```

Address of the
member element

Name of the member
to which 'ptr' refers to

```
// a function may be defined in some other file
void get_container(char *ptr)
{
    //by using the 'ptr', get the address of 'data'(container)
    struct some_data *pdata = container_of(ptr, struct some_data,c);
```

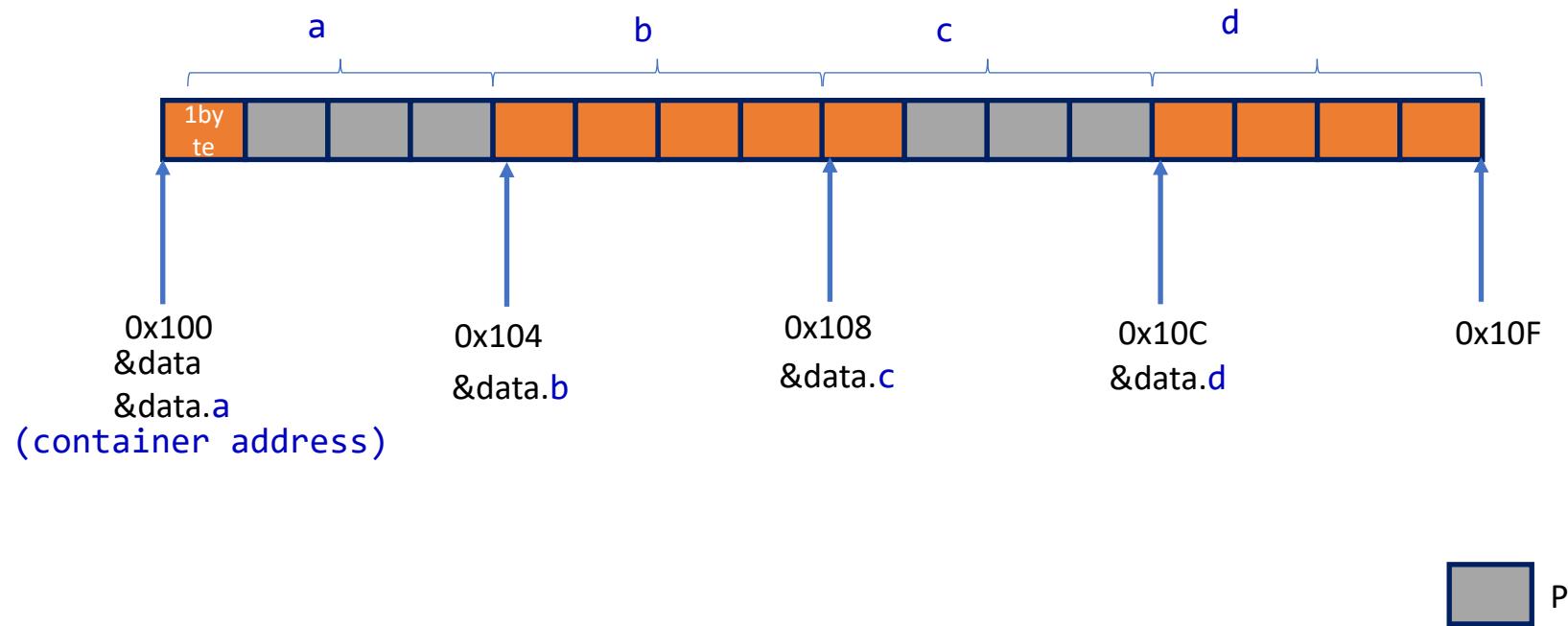
Structure type

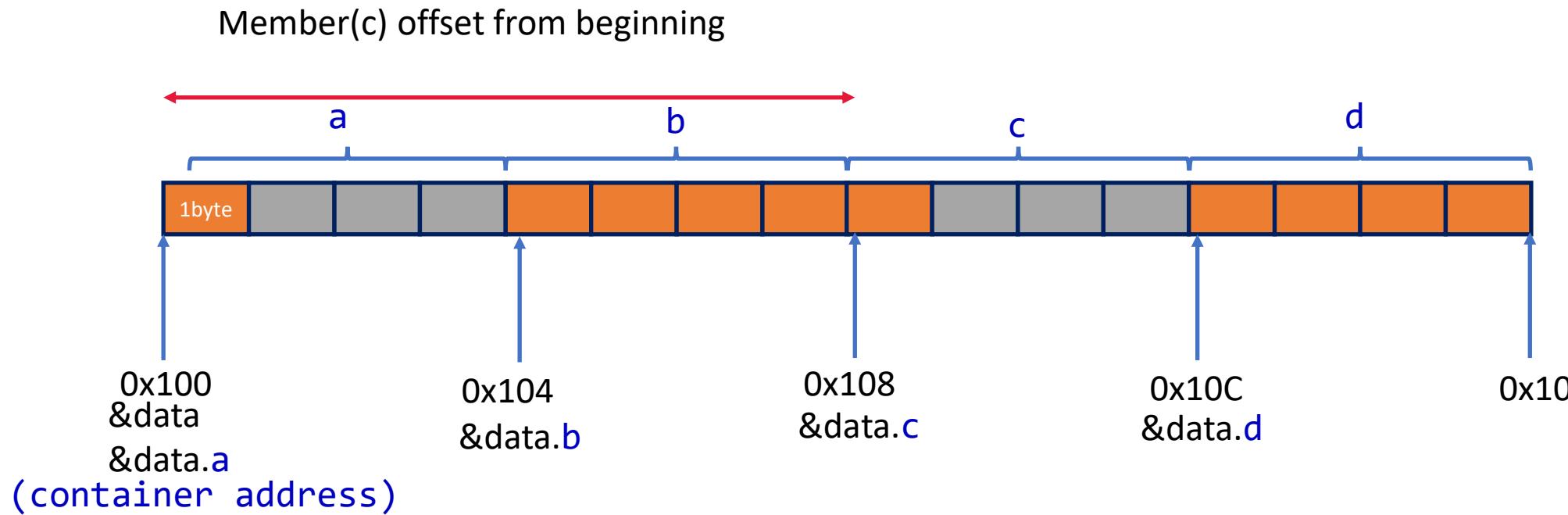
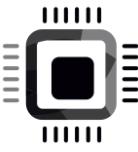


```
struct some_data
{
    char    a;
    int     b;
    char    c;
    int     d;
};

struct some_data data;
```

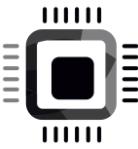
Memory plot of struct variable – data





`&data.c - member_offset_from_beginning = &data`



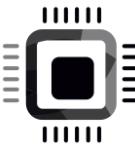


linux/kernel.h

```
#ifndef offsetof  
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)  
#endif
```

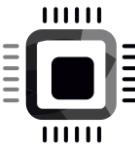
The macro `offsetof()` returns the offset of the field **MEMBER** from the start of the structure **TYPE**

.



f_mode

- This is one of the fields of `struct file` defined in `<linux/fs.h>`
- you can check this field in your driver to understand access mode requested from the user space application
- `f_mode` has bit fields to indicate access modes read or write . So, use macros `FMODE_READ` and `FMODE_WRITE` to decode this field (defined in `<linux/fs.h>`)

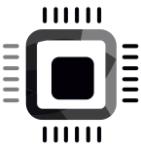


Decoding f_mode field

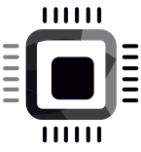
(`filp->f_mode & FMODE_READ`) File is opened for just read or read-write

(`filp->f_mode & FMODE_WRITE`) File is opened for just write or read-write

(`filp->f_mode & FMODE_READ`) && ! (`filp->f_mode & FMODE_WRITE`)
 File is opened for read-only

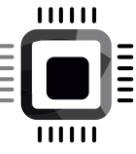


Platform devices and drivers



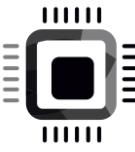
Platform devices and drivers

- Platform bus
- Platform devices
- Platform drivers



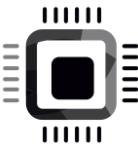
Bus

- In computer science, a bus is a collection of electrical wirings which transfers information (data or address or control) between devices.

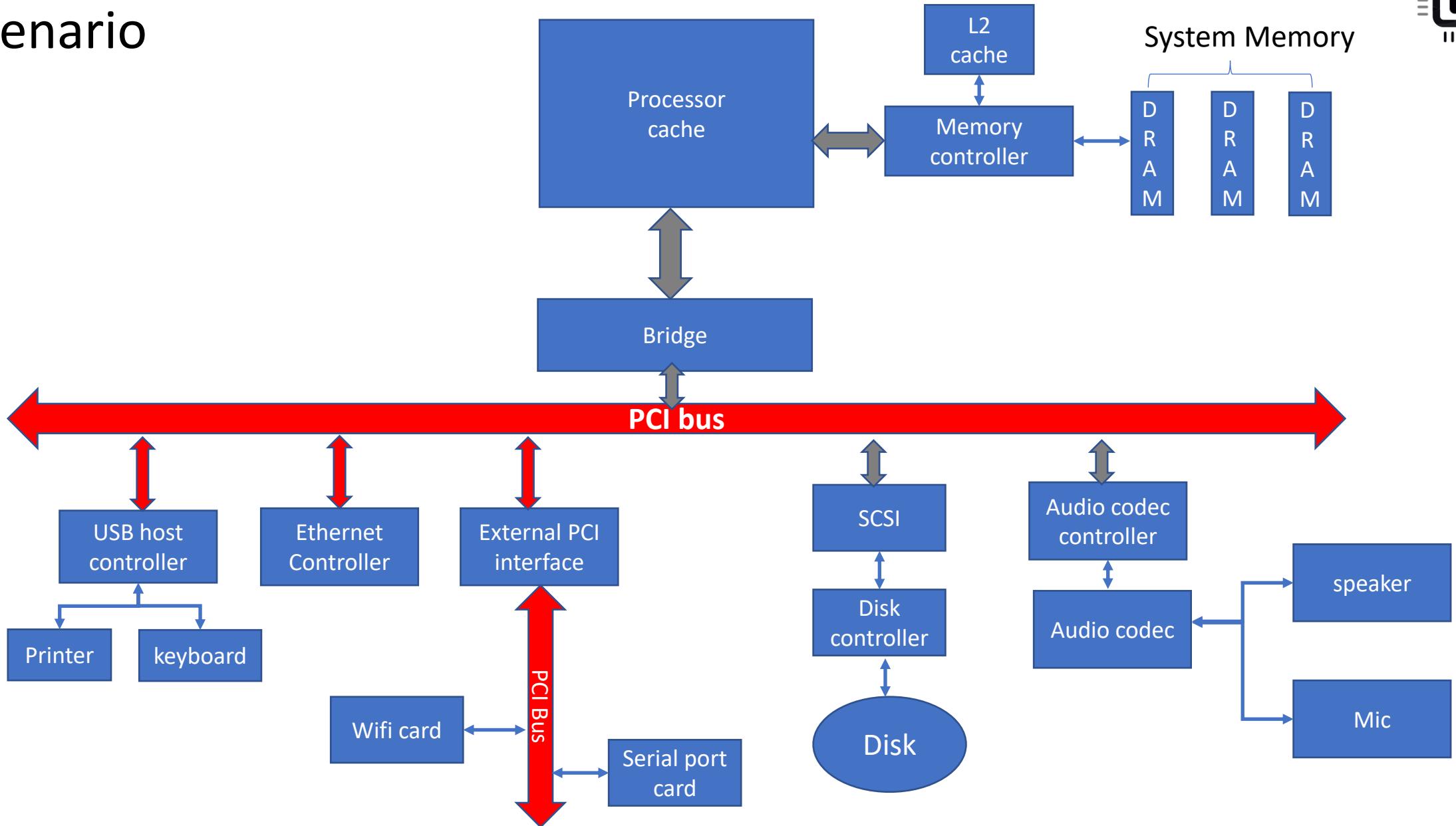


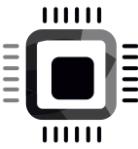
What is a platform bus ?

- Platform bus is a term used by Linux device model to represent all non-discoverable busses of an embedded platform.
- It is a pseudo bus or Linux's virtual bus.

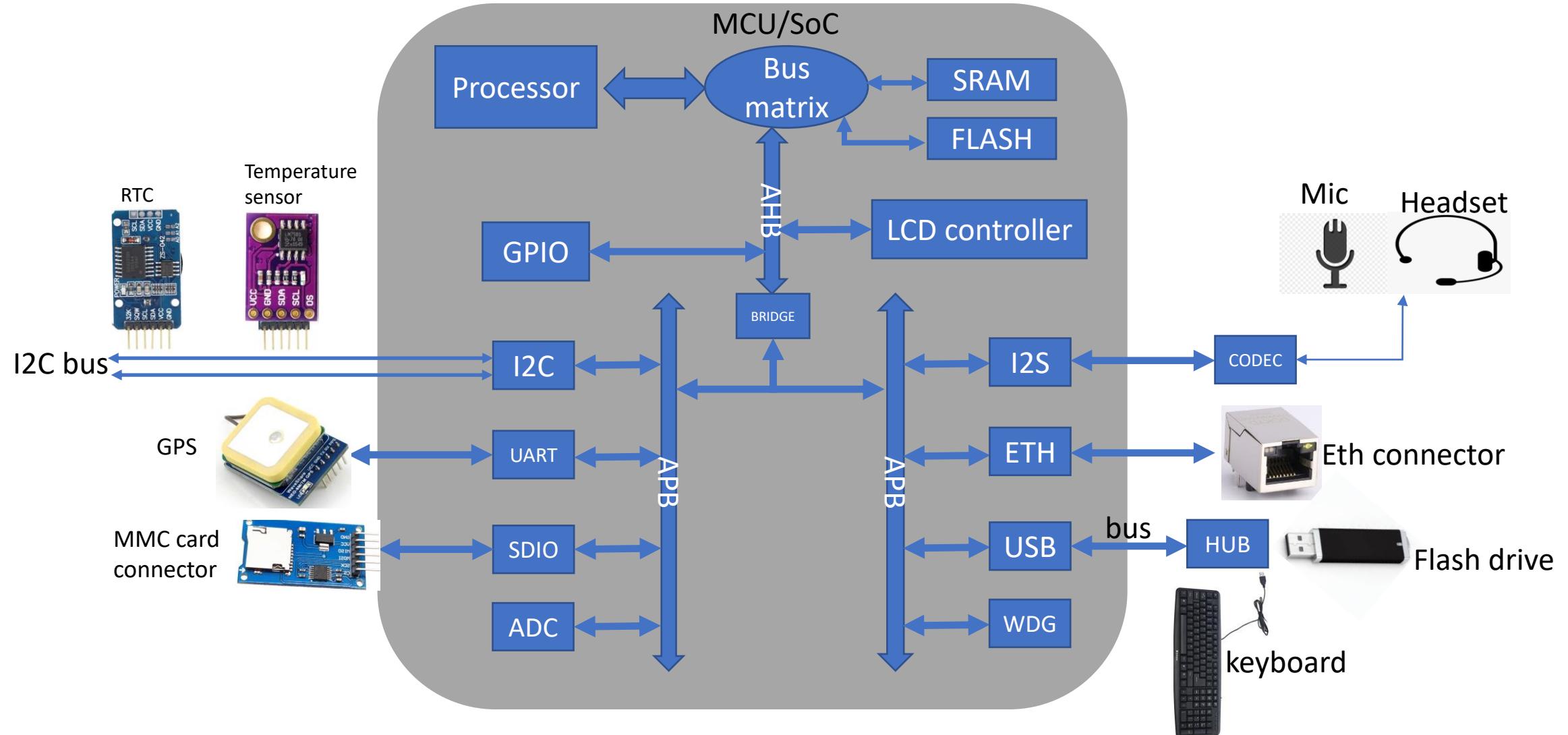


PC Scenario

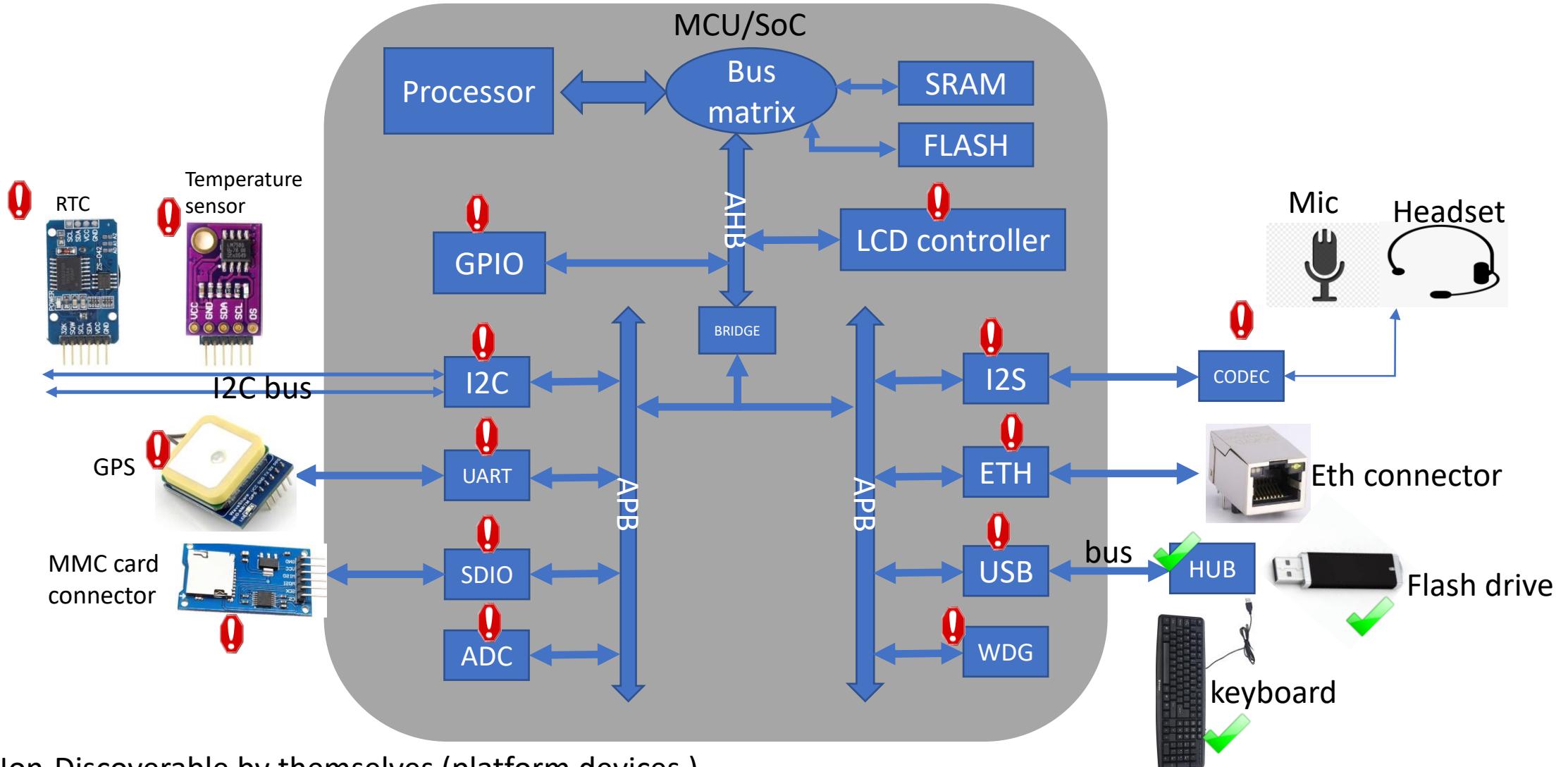
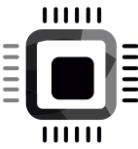




An Embedded platform

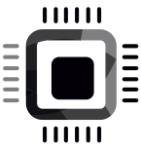


An Embedded platform

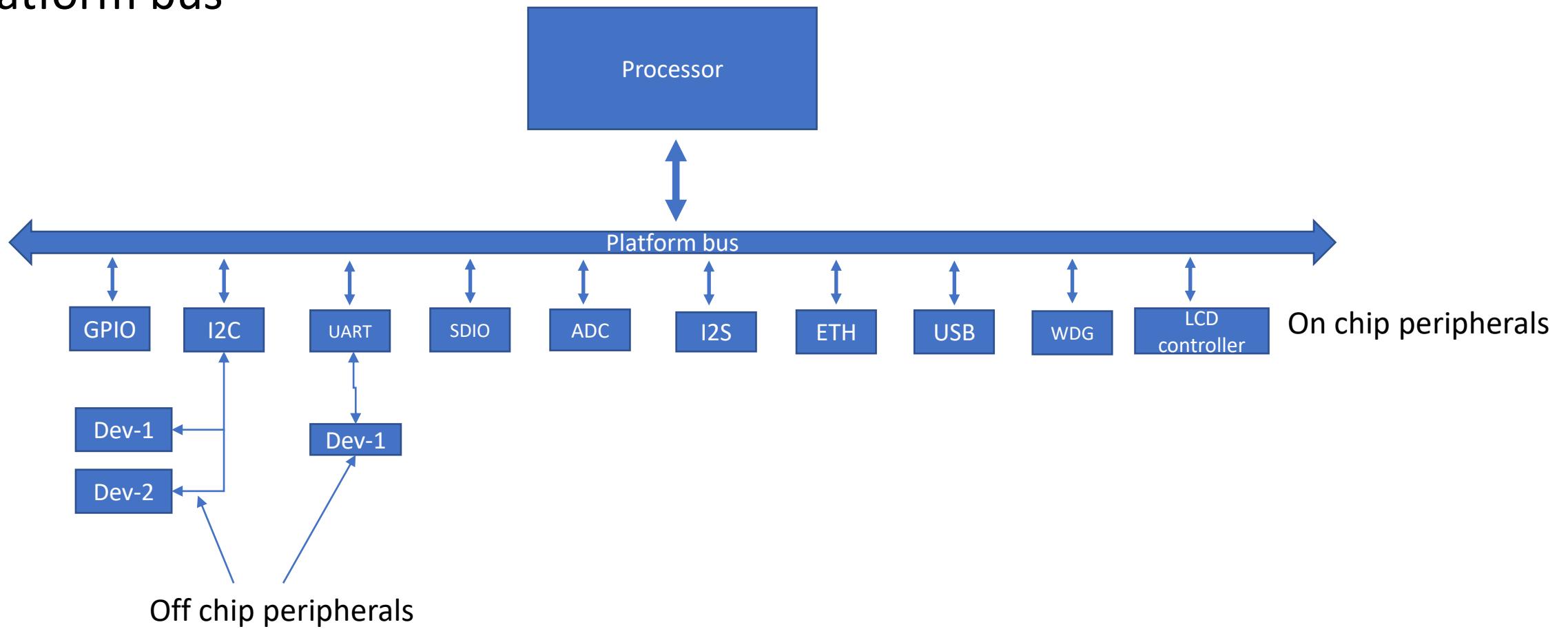


⚠ Non-Discoverable by themselves (platform devices)

✓ Self Discoverable

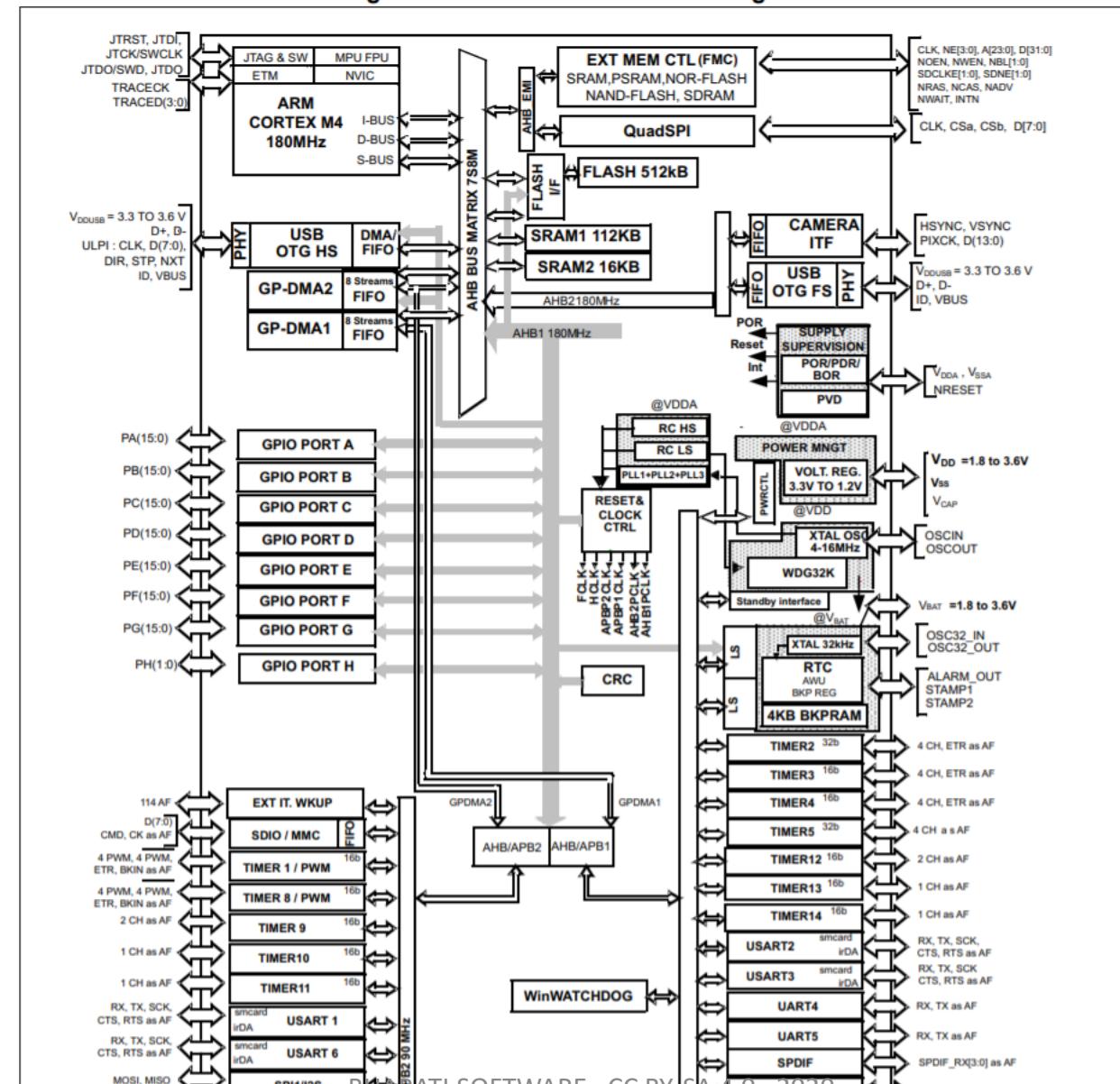


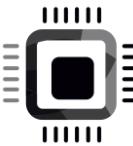
Platform bus



Embedded Scenario-STM32

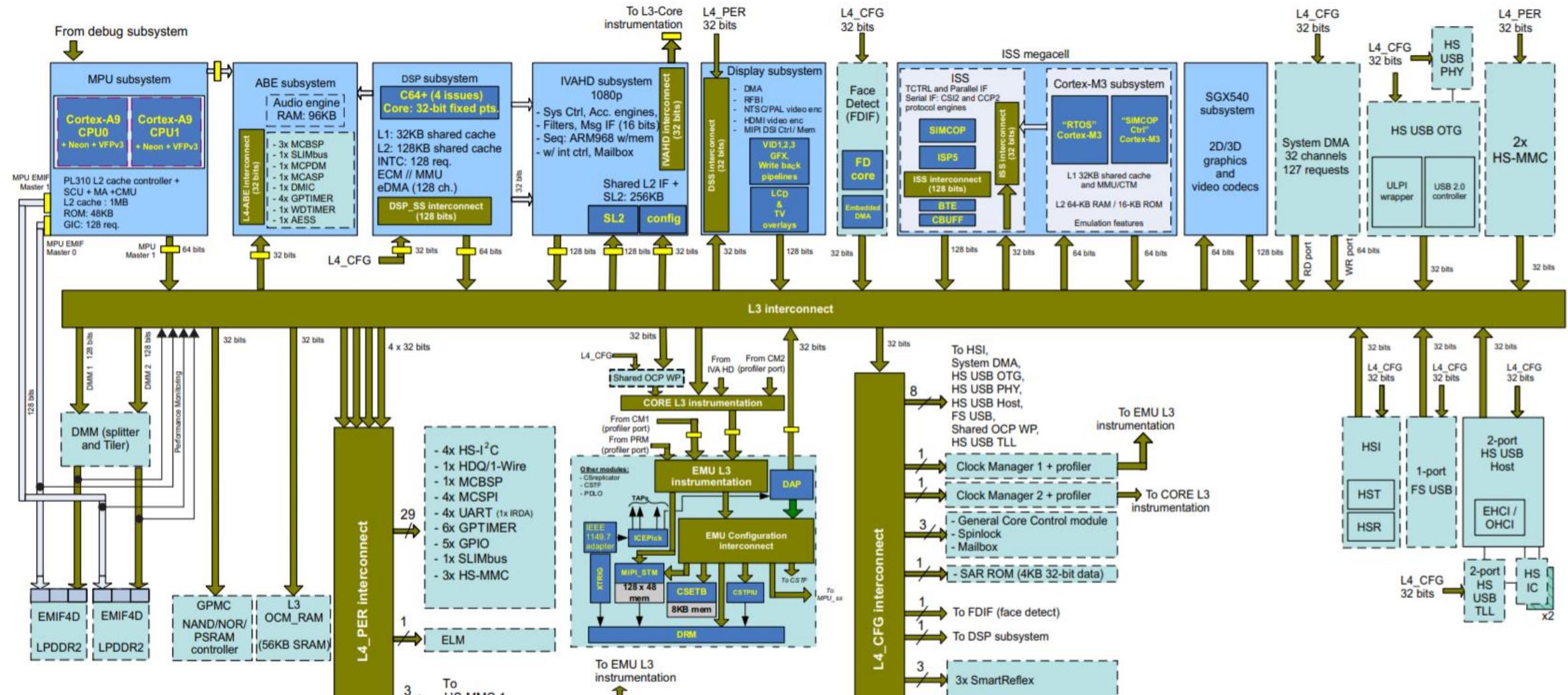
Figure 3. STM32F446xC/E block diagram

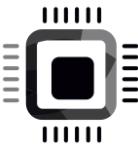




Embedded Scenario-OMAP4460

OMAP4460 Block Diagram





Functional Block Diagram

Figure 1-1 shows the AM335x microprocessor functional block diagram.

Embedded Scenario-AM335x

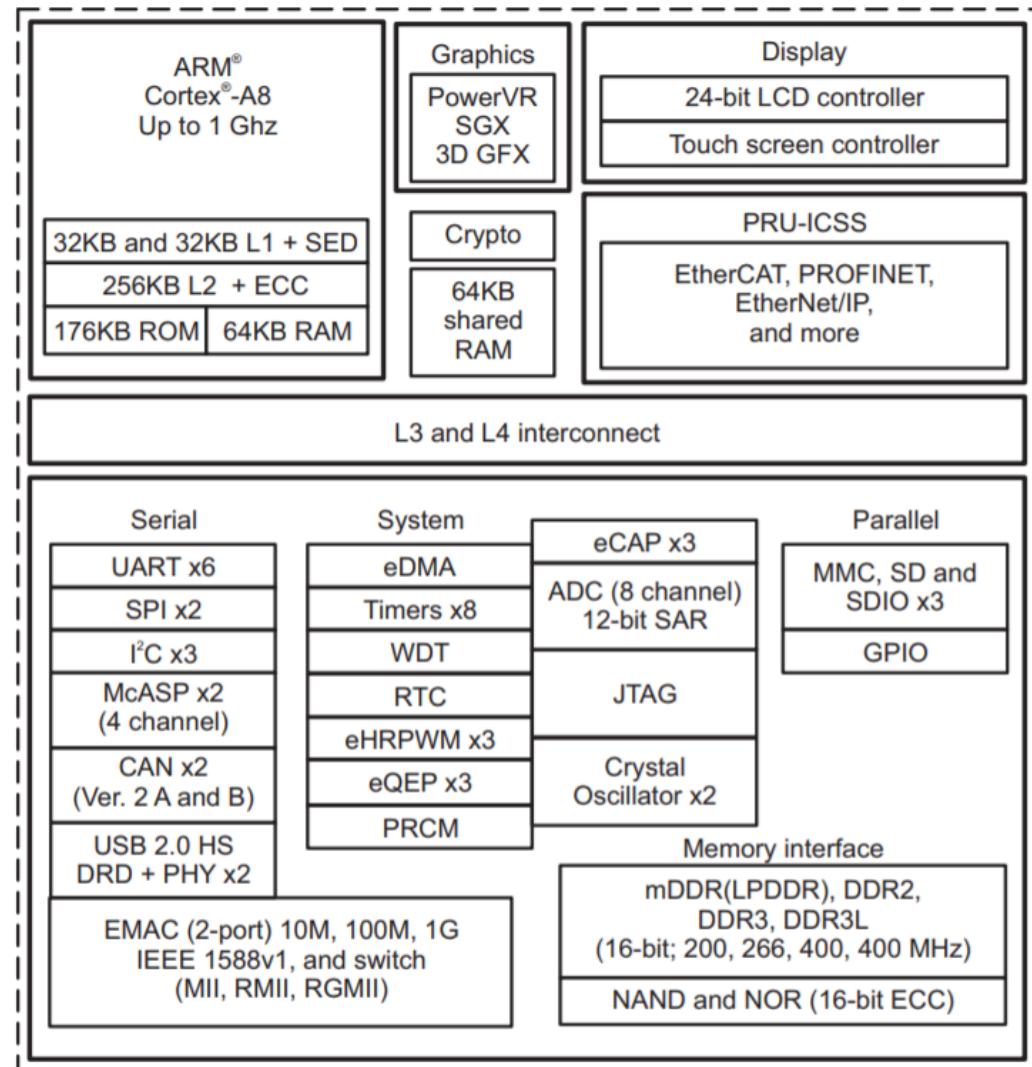
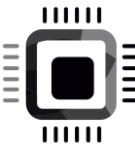
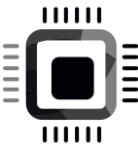


Figure 1-1. AM335x Functional Block Diagram



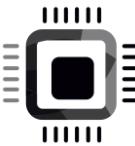
Discovery of devices

- Every device has its configuration data and resources, which need to be reported to the OS, which is running on the computer system.
- An operating system such as Windows or Linux, running on the computer, can auto-discover these data. Thus the OS learn about the connected devices automatically (Device enumeration)
- Enumeration is a process through which the OS can inquire and receive information, such as the type of the device, the manufacturer, device configuration, and all the devices connected to a given bus.
- Once the OS gathers information about a device, it can autoload the appropriate driver for the device. In the PC scenario, buses like PCI and USB support auto enumeration/hotplugging of devices.
- However, on embedded system platforms, this may not be the case since most peripherals are connected to the CPU over buses that don't support auto-discovery or enumeration of devices. We call them as platform devices.
- All these platform devices which are non-discoverable by nature, but they must be part of the Linux device model, so the information about these devices must be fed to the Linux kernel manually either at compile time or at boot time of the kernel.



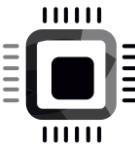
Device information

1. Memory or I/O mapped base address and range information
2. IRQ number on which the device issues interrupt to the processor
3. Device identification information
4. DMA channel information
5. Device address
6. Pin configuration
7. Power , voltage parameters
8. Other device specific data



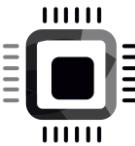
Adding Platform devices information to the kernel

- During compilation of kernel
 - Static method
 - Hardware details are part of kernel files (board file , drivers)
 - Deprecated and not recommended
- Loading dynamically
 - As a kernel module
 - Not recommended
- During kernel boot
 - Device tree blob
 - Latest and recommended



Board file approach

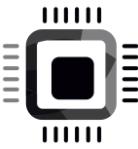
- This method was used before the kernel version 3.7 to add hardware configuration details to the kernel.
 - Details about the onboard peripherals
 - Pin configuration details
- You have to recompile the kernel if any device property changes
- All information about hardware configuration is hardcoded in the kernel source files and board files.



Device tree method

- The DT was originally created by Open Firmware as part of the communication method for passing data from Open Firmware to a client program (like to an operating system).
- An operating system used the Device Tree to discover the topology of the hardware at runtime, and thereby support a majority of available hardware without hard coded information (assuming drivers were available for all devices).

<https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>

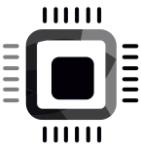


Platform devices

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

Platform driver

- A driver who is in charge of handling a platform device is called a platform driver

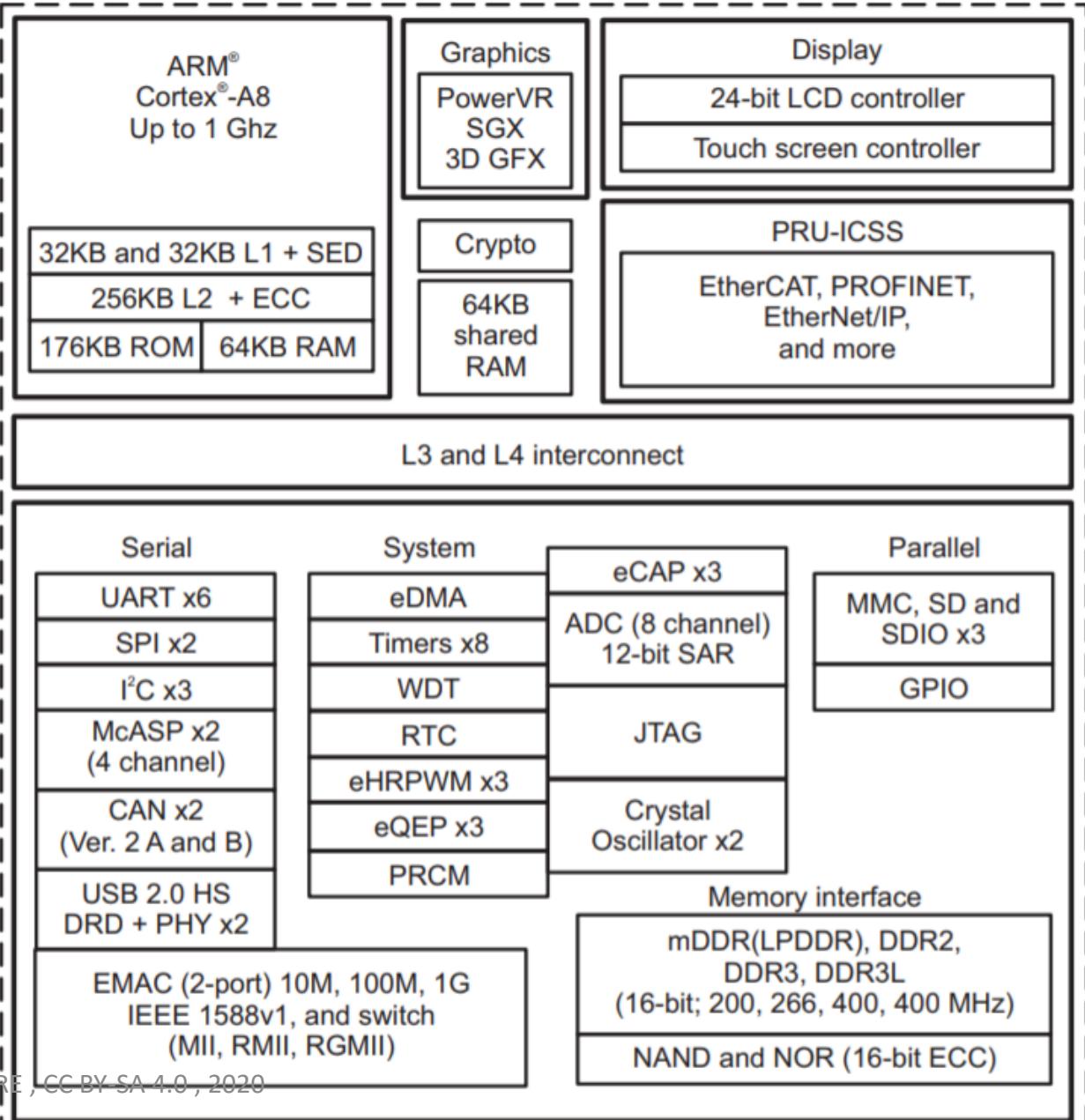


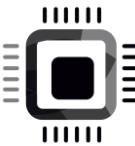
Examples of platform drivers

AM335X block diagram

Functional Block Diagram

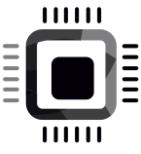
Figure 1-1 shows the AM335x microprocessor functional block diagram.



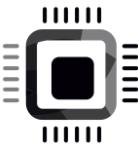


TI AM335x On-chip peripheral support drivers

Peripheral	Location of the platform driver	Description
SPI	drivers/spi/spi-omap2-mcspi.c	OMAP2 McSPI controller driver
I2C	drivers/i2c/busses/i2c-omap.c	TI OMAP I2C master mode driver
USB OTG	drivers/usb/musb/musb_am335x.c	musb controller
CAN	drivers/net/can/c_can/c_can_platform.c	Platform CAN bus driver for Bosch C_CAN controller
MMC	drivers/mmc/host/omap_hsmmc.c	Driver for OMAP2430/3430 MMC controller.
GPIO	drivers/gpio/gpio-omap.c	Support functions for OMAP GPIO
UART	drivers/tty/serial/8250/8250_omap.c	8250-core based driver for the OMAP internal UART
LCD controller	drivers/gpu/drm/tilcdc/tilcdc_drv.c	LCDC DRM driver, based on da8xx-fb
Touch screen controller	drivers/input/touchscreen/ti_am335x_tsc.c	TI Touch Screen driver



Registering platform device and driver

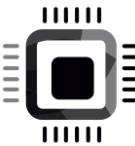


Registering a platform driver

Use this 'C' macro to register your platform driver with the Linux platform core

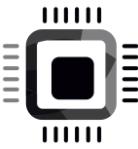
```
/*  
 * use a macro to avoid include chaining to get THIS_MODULE  
 */  
#define platform_driver_register(drv) \  
    __platform_driver_register(drv, THIS_MODULE)  
extern int __platform_driver_register(struct platform_driver *,  
                                     struct module *);
```

/include/linux/platform_device.h



Platform driver structure

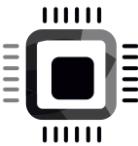
```
struct platform_driver {  
    int (*probe)(struct platform_device *);  
    int (*remove)(struct platform_device *);  
    void (*shutdown)(struct platform_device *);  
    int (*suspend)(struct platform_device *, pm_message_t state);  
    int (*resume)(struct platform_device *);  
    struct device_driver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



Registering a platform device

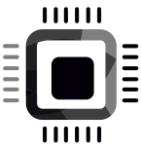
```
int platform_device_register(struct platform_device *pdev);
```

(Deprecated)

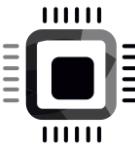


Platform device structure

```
struct platform_device {  
    const char *name;  
    int id;  
    bool id_auto;  
    struct device dev;  
    u32 num_resources;  
    struct resource *resource;  
  
    const struct platform_device_id *id_entry;  
    char *driver_override; /* Driver name to force a match */  
  
    /* MFD cell pointer */  
    struct mfd_cell *mfd_cell;  
  
    /* arch specific additions */  
    struct pdev_archdata archdata;  
};
```



Platform device – driver matching



Now you understood,
How to register a platform device and platform driver with the Linux kernel.

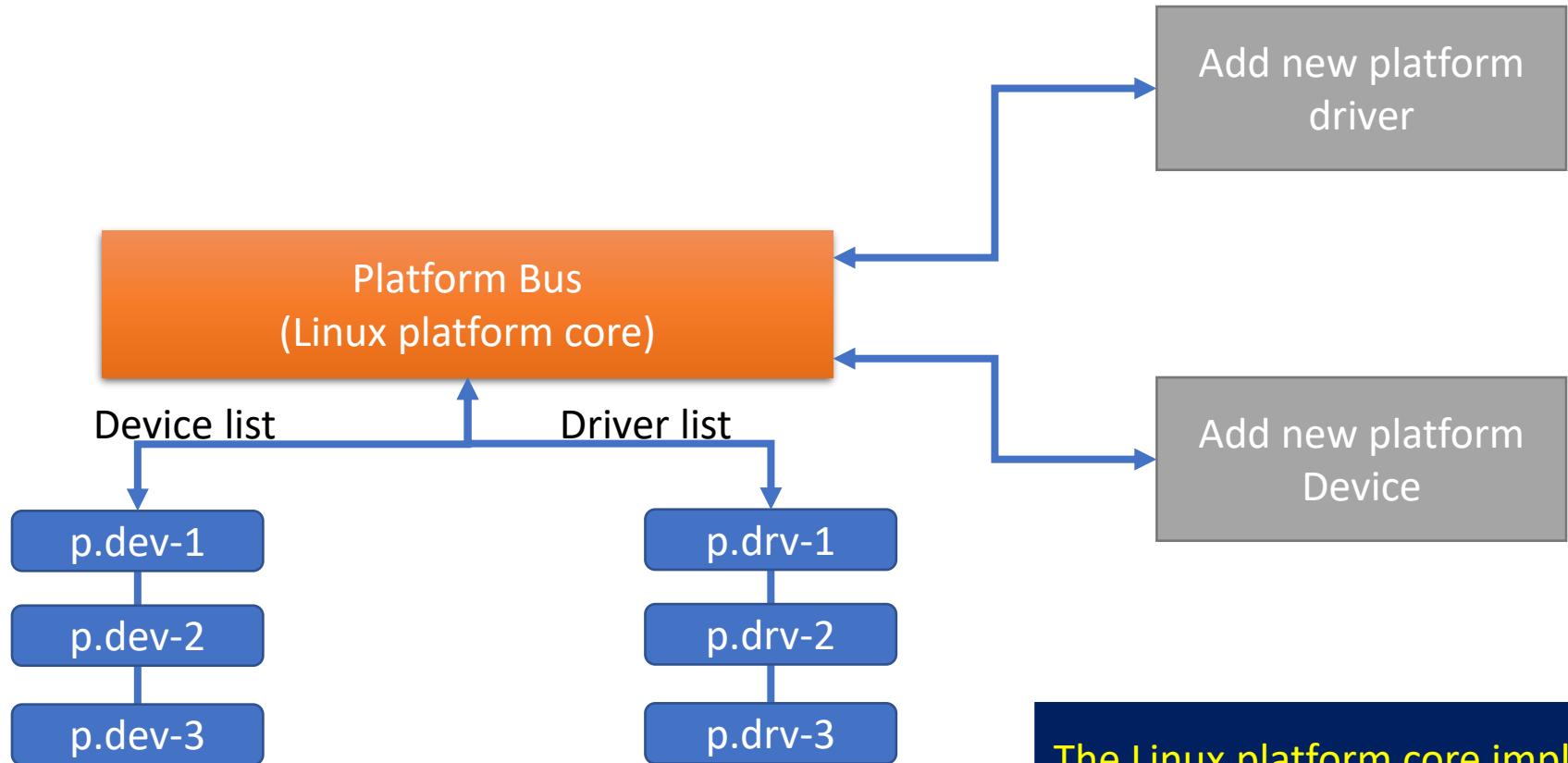
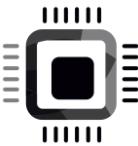
Now let's attend a very important question.

How does the driver know that you have added exactly the same device
which the driver is looking for so that it can configure the device?

Or

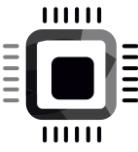
How do you make the correct driver gets autoloaded whenever you add the
new platform device?

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

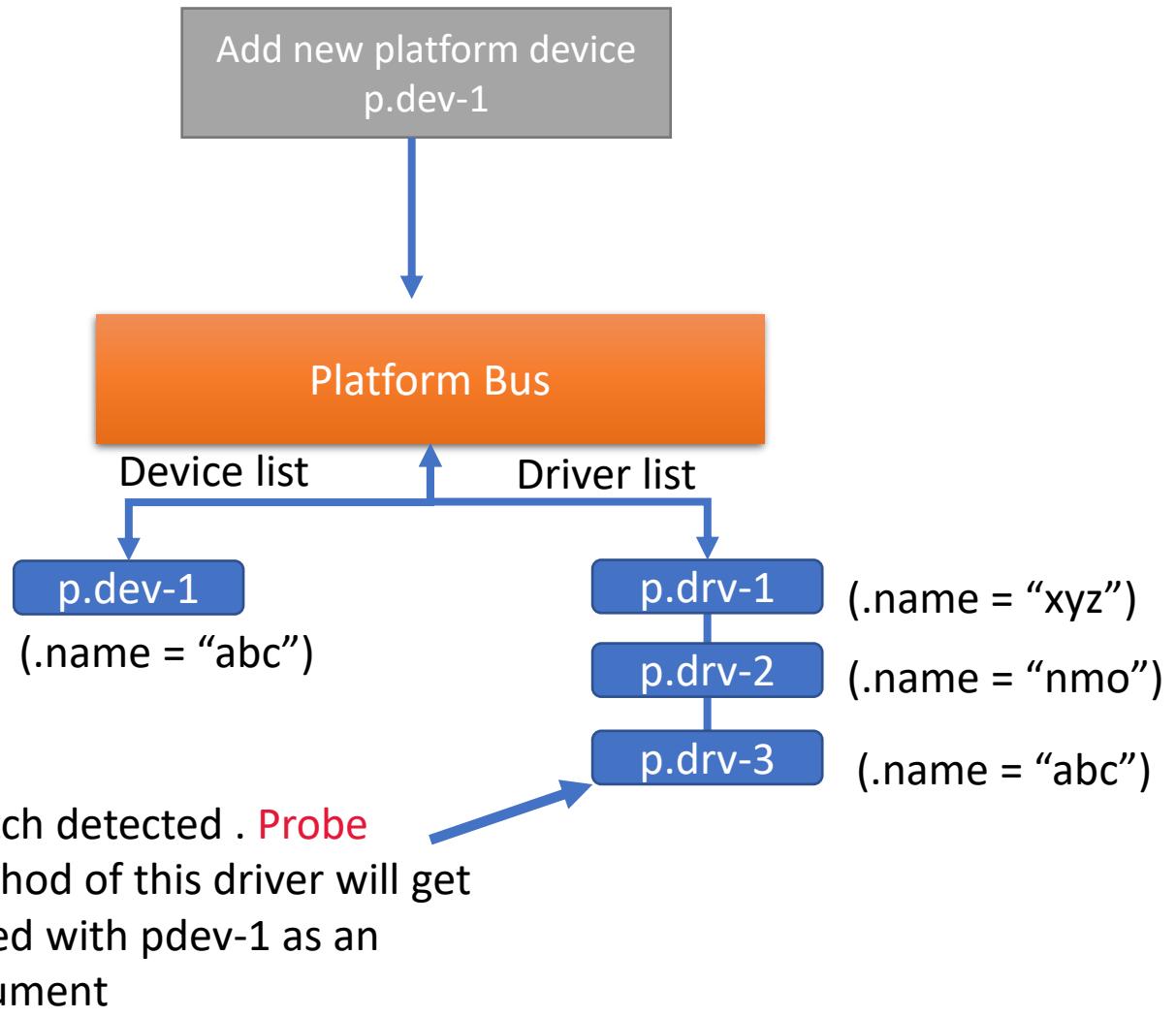


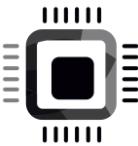
Every bus type has its match function, where the device and driver list will be scanned.

The Linux platform core implementation maintains platform device and driver lists. Whenever you add a new platform device or driver, this list gets updated and matching mechanism triggers.

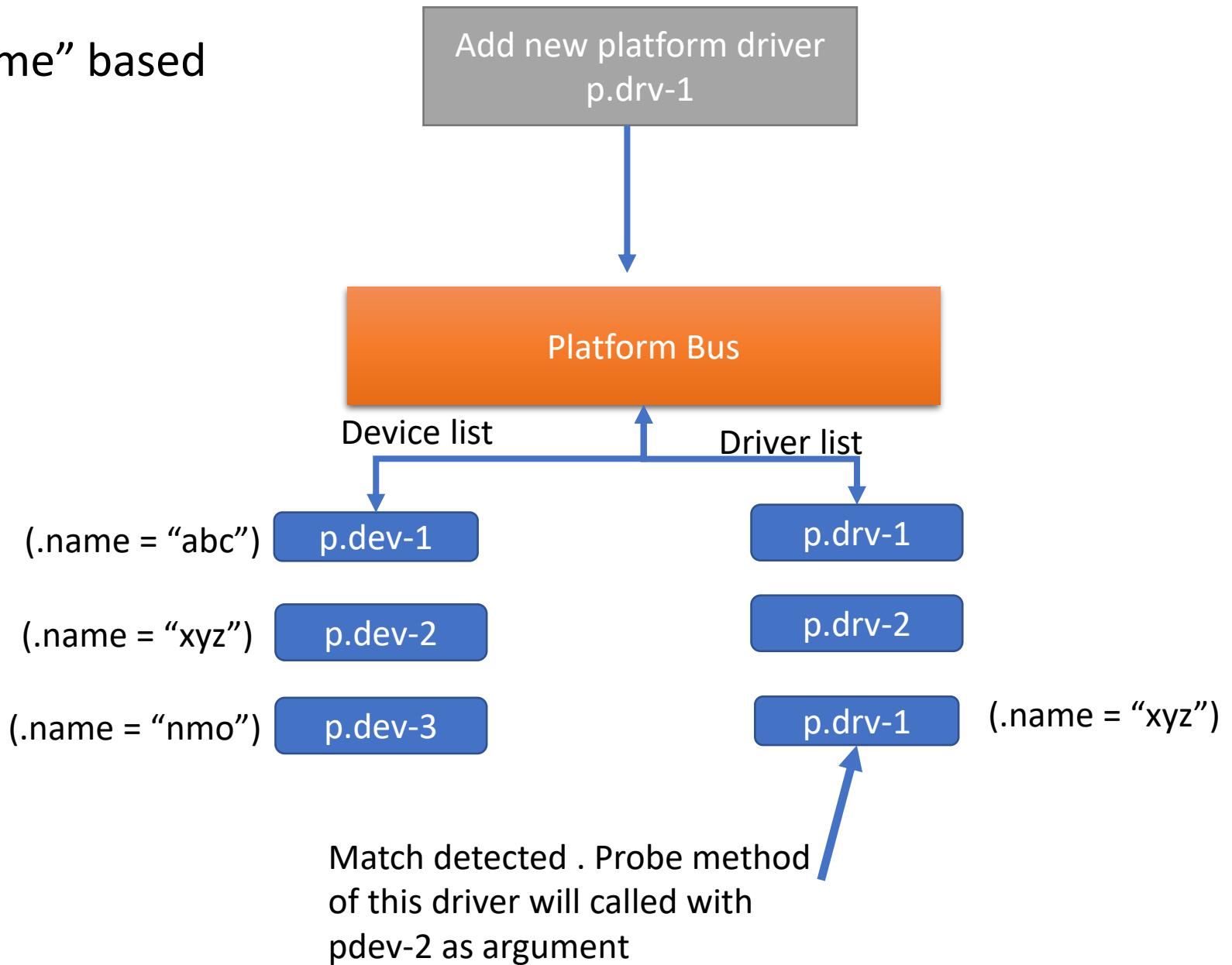


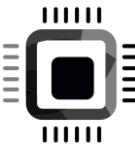
A simple “name” based matching





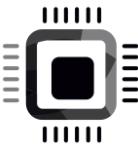
A simple “name” based matching





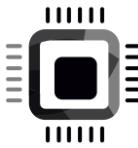
Points to remember

- Whenever a new device or a new driver is added, the matching function of the platform bus runs, and if it finds a matching platform device for a platform driver, the probe function of the matched driver will get called. Inside the probe function, the driver configures the detected device.
- Details of the matched platform device will be passed to the probe function of the matched driver so that driver can extract the platform data and configure it.



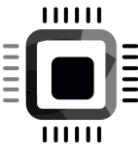
Platform driver structure

```
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;  
};
```



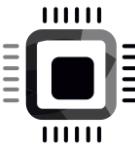
Probe function of the platform driver

- Probe function must be implemented by the platform driver and should be registered during `platform_driver_register()`.
- When the bus matching function detects the matching device and driver, probe function of the driver gets called with detected platform device as an input argument
- Note that `probe()` should in general, verify that the 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`.
- The probe function is responsible for
 - Device detection and initialization
 - Allocation of memories for various data structures,
 - Mapping i/o memory
 - Registering interrupt handlers
 - Registering device to kernel framework, user level access point creations, etc
- The probe may return 0(Success) or error code. If probe function returns a non-zero value, meaning probing of a device has failed.



Remove function of the platform driver

- Remove function gets called when a platform device is removed from the kernel *to unbind* a device from *the driver* or when the kernel no longer uses the platform device
- Remove function is responsible for
 - Unregistering the device from the kernel framework
 - Free memory if allocated on behalf of a device
 - Shutdown/De-initialize the device



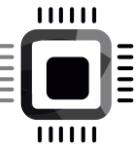
Other methods of platform driver

```
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;  
};
```

Called at shut-down time to quiesce the device

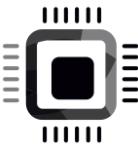
Called to bring a device from sleep mode

*Called to put the device to sleep mode.
Usually to a low power state*



For more info

- <https://www.kernel.org/doc/Documentation/driver-model/platform.txt>



```
/*remove function of the driver */
int pcd_driver_remove(struct platform_device *pdev)
{
    return 0;
}

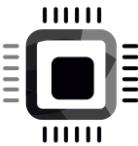
/*probe function of the driver */
int pcd_driver_probe(struct platform_device *pdev)
{
    return 0;
}

/*module init function */
int __init pcd_driver_init(void)
{
    platform_driver_register();
    return 0;
}

/*module cleanup function */
void __exit pcd_driver_exit(void)
{
    platform_driver_unregister();
}

/*registration macros for module's init and cleanup functions */
module_init(pcd_driver_init);
module_exit(pcd_driver_exit);
```

Platform driver essential methods



```
/*remove function of the driver */
int pcd_driver_remove(struct platform_device *pdev)
{
    return 0;
}
```

```
/*probe function of the driver */
int pcd_driver_probe(struct platform_device *pdev)
{
    return 0;
}
```

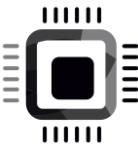
```
/*module init function */
int __init pcd_driver_init(void)
{
    platform_driver_register();
    return 0;
}

/*module cleanup function */
void __exit pcd_driver_exit(void)
{
    platform_driver_unregister();
}

/*registration macros for module's init and cleanup functions */
module_init(pcd_driver_init);
module_exit(pcd_driver_exit);
```

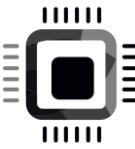
module_platform_driver(__platform_driver);





```
/* module_platform_driver() - Helper macro for drivers that don't do
 * anything special in module init/exit. This eliminates a lot of
 * boilerplate. Each module may only use this macro once, and
 * calling it replaces module_init() and module_exit()
 */
#define module_platform_driver(__platform_driver) \
    module_driver(__platform_driver, platform_driver_register, \
                  platform_driver_unregister)
```

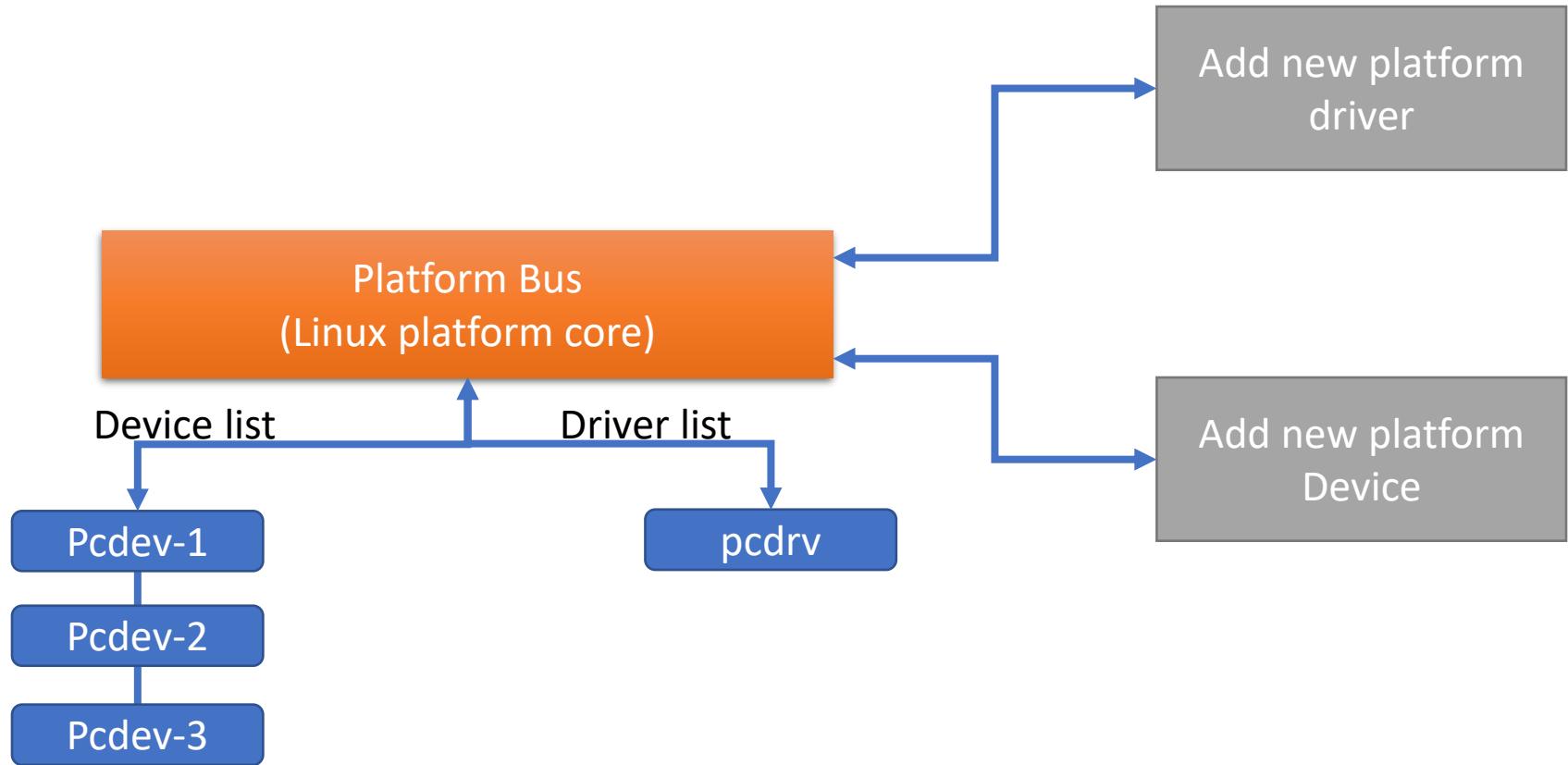
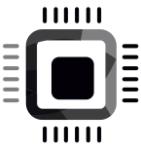
Include/linux/platform_device.h

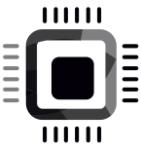


Code exercise:

Implementation of pseudo character driver as platform driver

- Repeat the exercise pseudo character driver with multiple devices as a platform driver.
- The driver should support multiple pseudo character devices(pcdevs) as platform devices
- Create device files to represent platform devices
- The driver must give open, release, read, write, lseek methods to deal with the devices





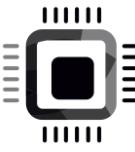
Implementation

Platform driver

Kernel module-1

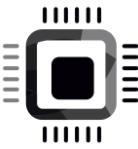
Platform device setup

Kernel module-2



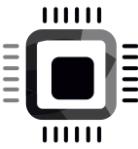
Platform device release function

- *Callback to free the device after all references have gone away.*
- *This should be set by the allocator of the device*



Platform device setup

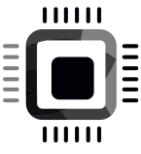
1. Create 2 platform devices and initialize them with required information
 1. Name of a platform device
 2. Platform data
 3. Id of the device
 4. Release function for the device
2. Register platform devices with the Linux kernel



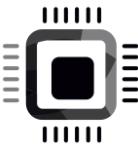
```
/*Device private data structure */

struct pcdev_private_data
{
    struct pcdev_platform_data pdata;
    char *buffer;
    dev_t dev_num; /*holds device number */
    struct cdev cdev;
};

/*Driver private data structure */
struct pcdrv_private_data
{
    int total_devices;
    dev_t device_num_base;
    struct class *class_pcd;
    struct device *device_pcd;
};
```



Matching of platform driver and device using platform device ids



Use case : Different versions of a chip

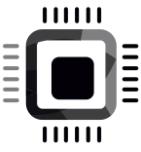
Different version of a temperature sensing chip



Configuration items the driver uses to configure the matched platform device

Platform driver must support different device ids

TS-A1x
TS-B1x
TS-C1x



Modify the pcd platform driver to support different version of pcdevs

pcdev-A1x

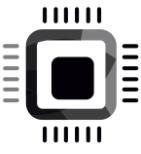
pcdev-B1x

pcdev-C1x

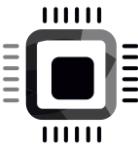
Config items of
pcdev-A1x

Config items of
pcdev-B1x

Config items of
pcdev-C1x

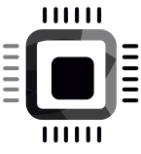


Device tree

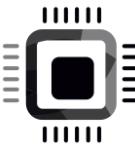


Device tree

- Introduction to the device tree
- Device tree structure
- Example of a device node
- properties of device trees
- Device tree overlays



What is device tree(DT) ?

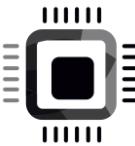


What is device tree?

- The "Open Firmware Device Tree", or simply Device Tree (DT), is a data exchange format used for exchanging hardware description data with the software or OS.
- More specifically, it is a description 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 a new and recommended way to describe non-discoverable devices(platform devices) to the Linux kernel, which was previously hardcoded into kernel source files.

Source :

Documentation/devicetree/usage-model.txt

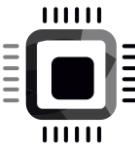


Device tree

- An operating system uses the Device Tree to discover the topology of the hardware at runtime, and thereby support a majority of available hardware without hardcoded information (assuming drivers were available for all devices)
- The most important thing to understand is that the DT is simply a data structure that describes the hardware. There is nothing magical about it, and it does not magically make all hardware configuration problems go away.

Source :

Documentation/devicetree/usage-model.txt

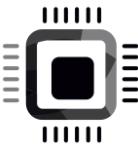


Device tree

- DT provides a language for decoupling the hardware configuration from the device driver and board support files of the Linux kernel (or any other operating system for that matter).
- Using it allows device drivers to become data-driven. To make setup decisions based on data passed into the kernel instead of on per-machine hardcoded selections.
- Ideally, a data-driven platform setup should result in less code duplication and make it easier to support a wide range of hardware with a single kernel image.

Source :

Documentation/devicetree/usage-model.txt

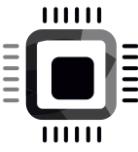


Why DT is used ?

Linux uses DT for,

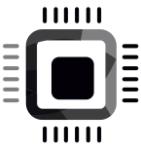
- Platform identification
- Device population:
 - The kernel parses the device tree data and generates the required software data structure, which will be used by the kernel code.

Ideally, the device tree is independent of any os; when you change the OS, you can still use the same device tree file to describe the hardware to the new OS. That is, the device tree makes “adding of device information “ independent of OS

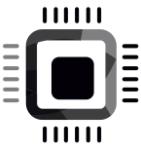


More reading

- https://elinux.org/Device_Tree_What_It_Is
- <https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>

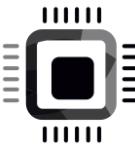


Writing device tree



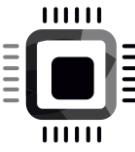
Device tree specification

- You can get the full specification here
- <https://www.devicetree.org/>



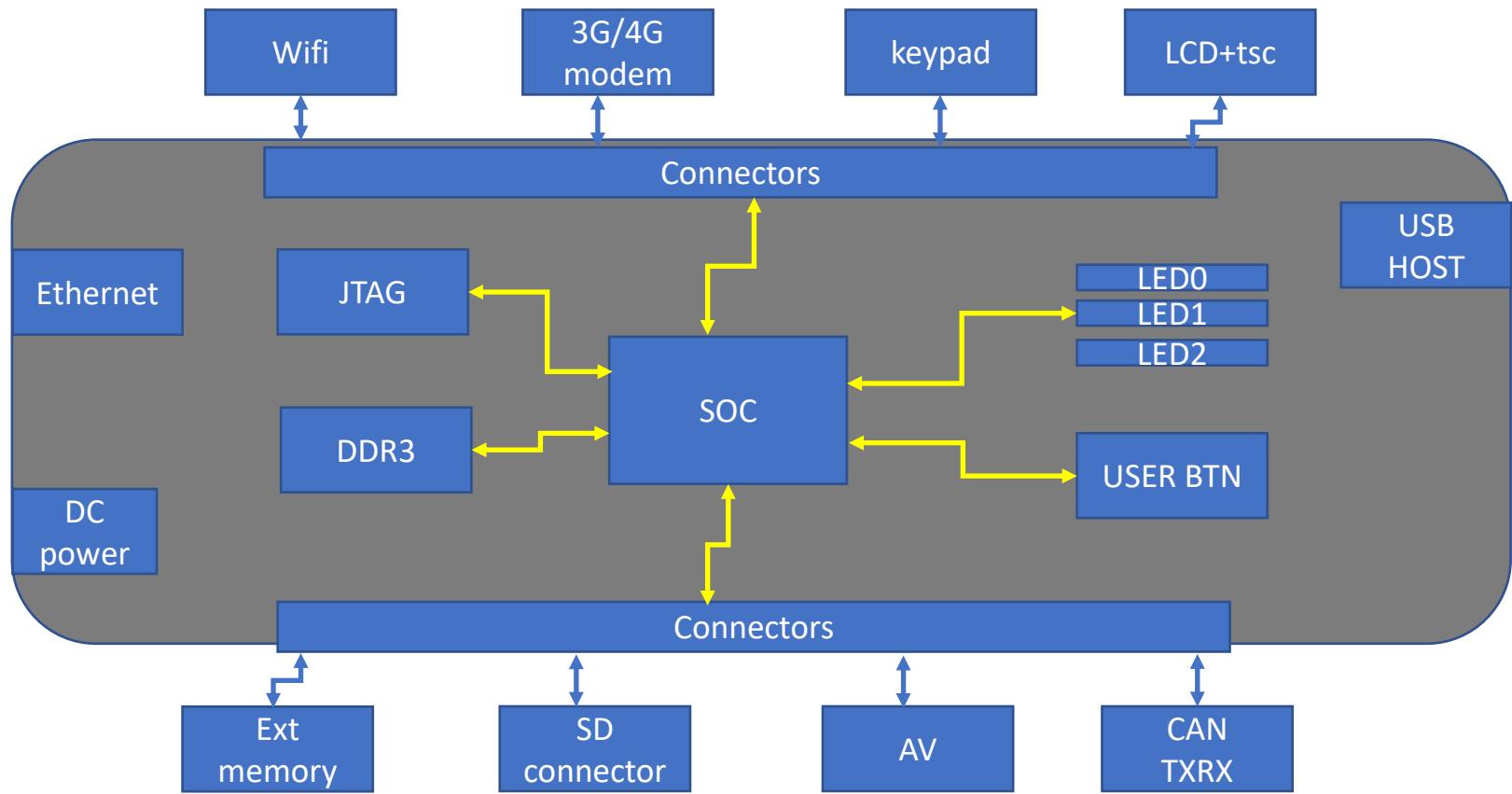
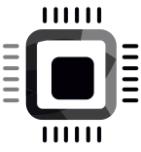
Writing device tree

- The device tree supports a hierarchical way of writing hardware description at the soc level, common board level, and board-specific level. Most of the time, writing a new device tree is not difficult, and you can reuse most of the common hardware information from the device tree file of the reference board.
- For example, when you design a new board, which is slightly different from another reference board, then you can reuse the device tree file of the reference board and only add that information which is new in your custom board.

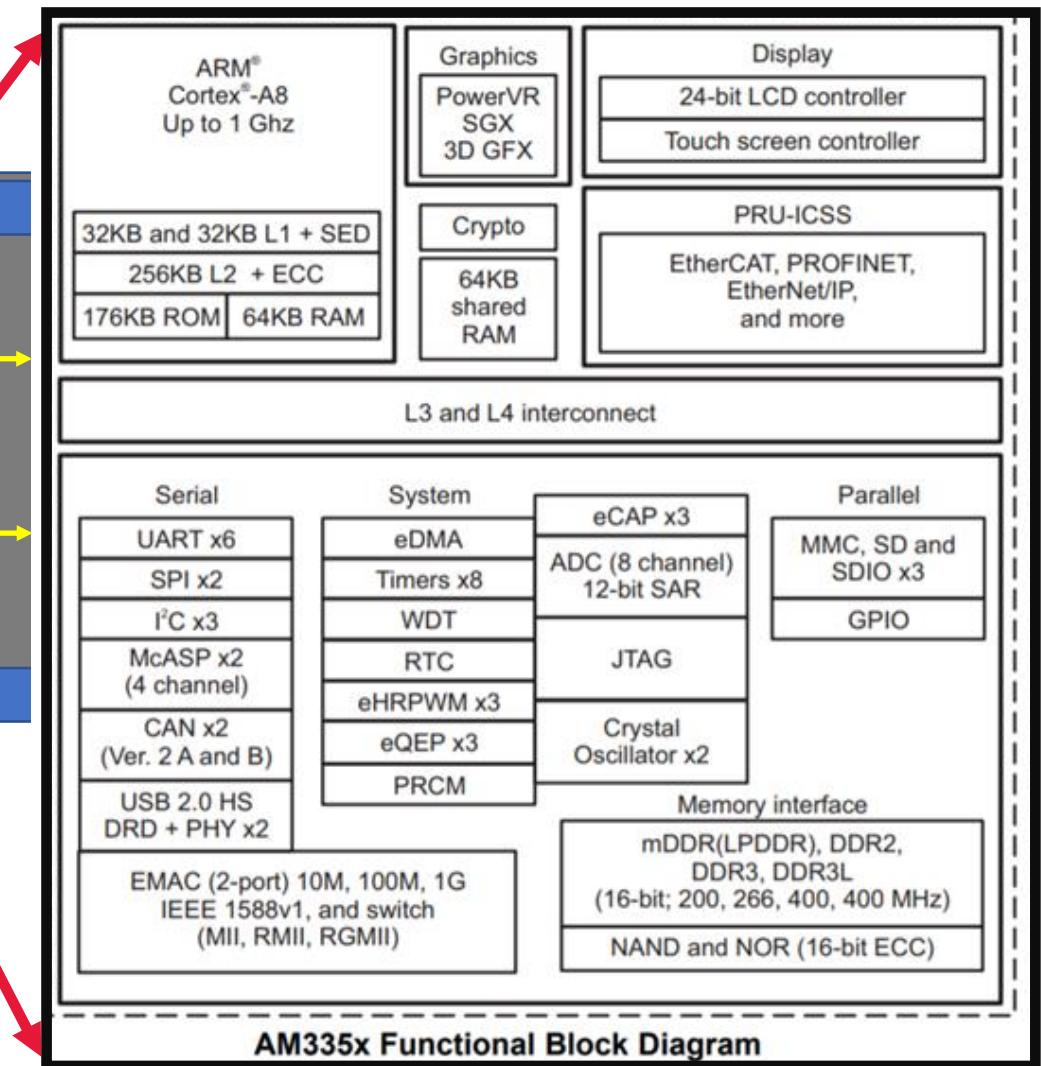
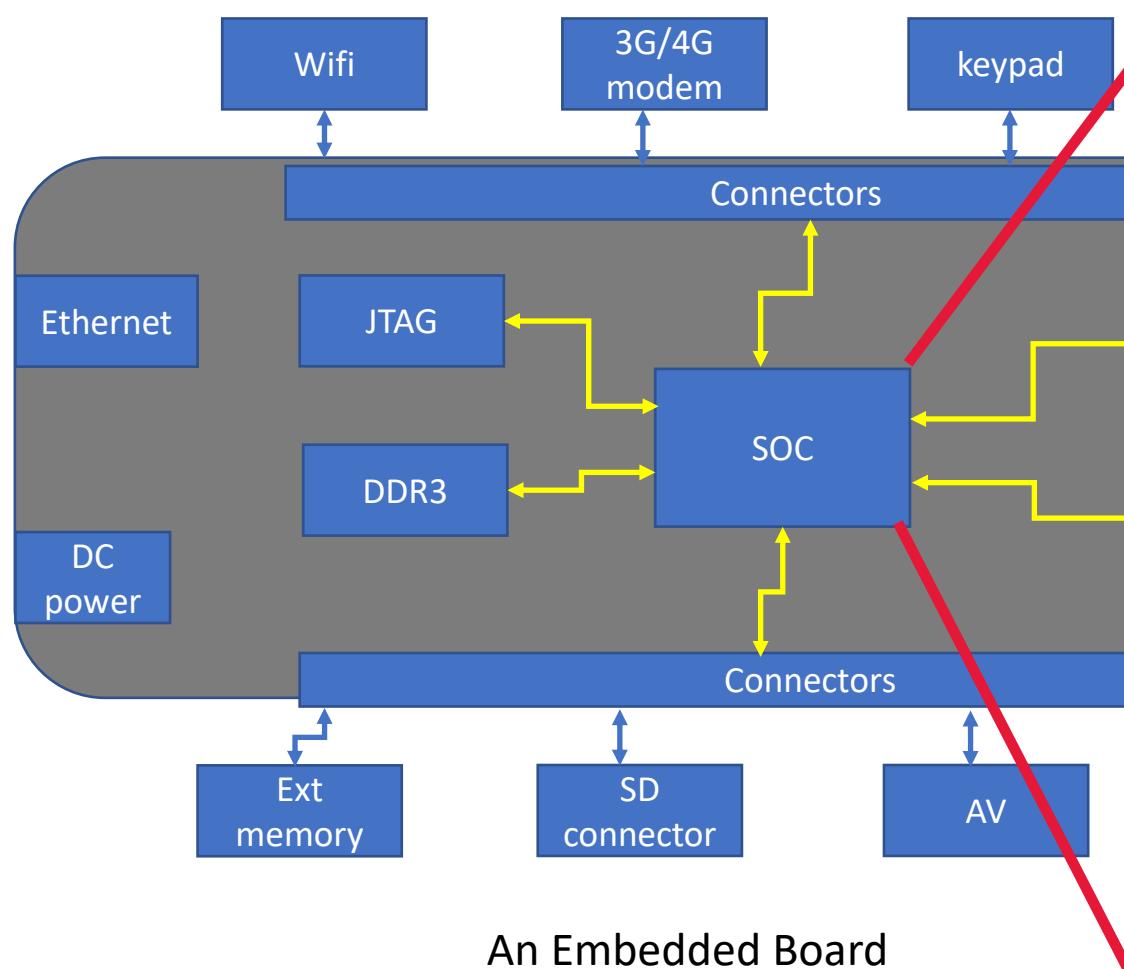
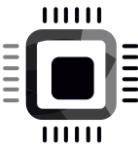


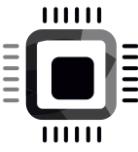
Describing hardware hierarchy

- It comes at various level because the board has many device blocks
 - SOC
 - SOC has an on-chip processor and on-chip peripherals
 - The board also has various peripherals onboard, like sensors, LEDs, buttons, joysticks, external memories, touchscreen, etc

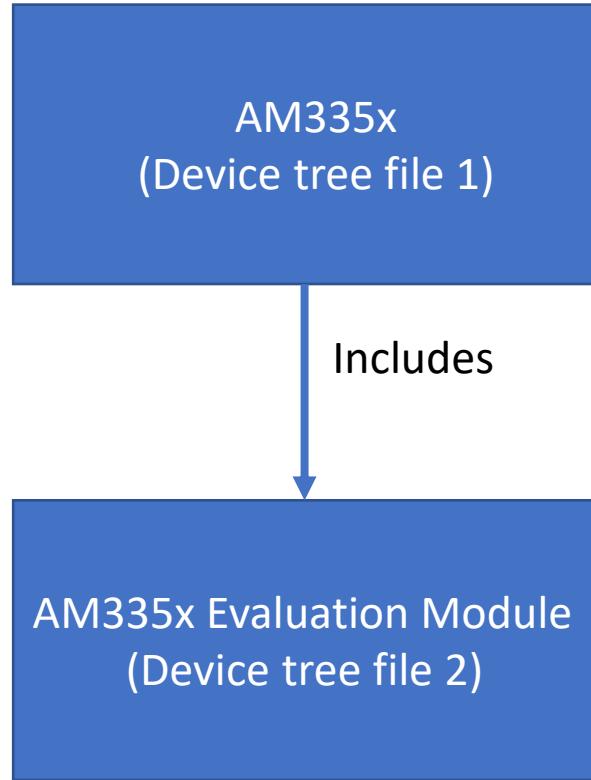


An Embedded Board



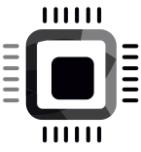


Modular approach to manage DT files

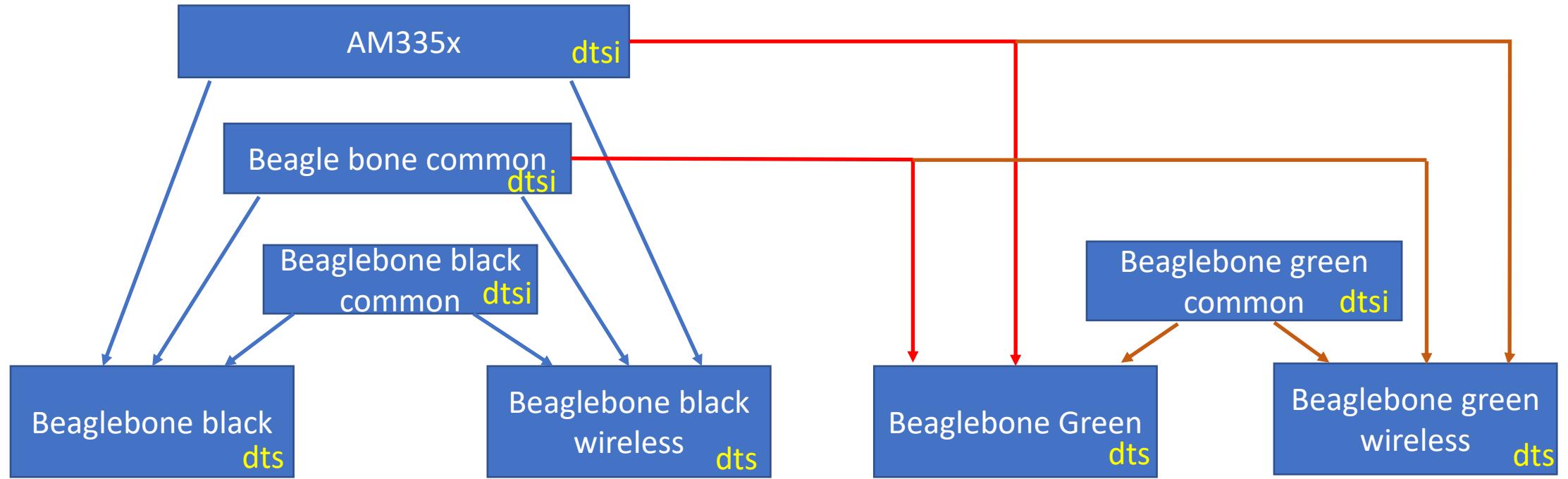


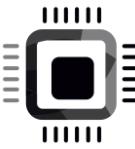
SOC specific device tree file
(This DT file is used as an include file and can be used with another board which is based on same SOC)

Board specific device tree file



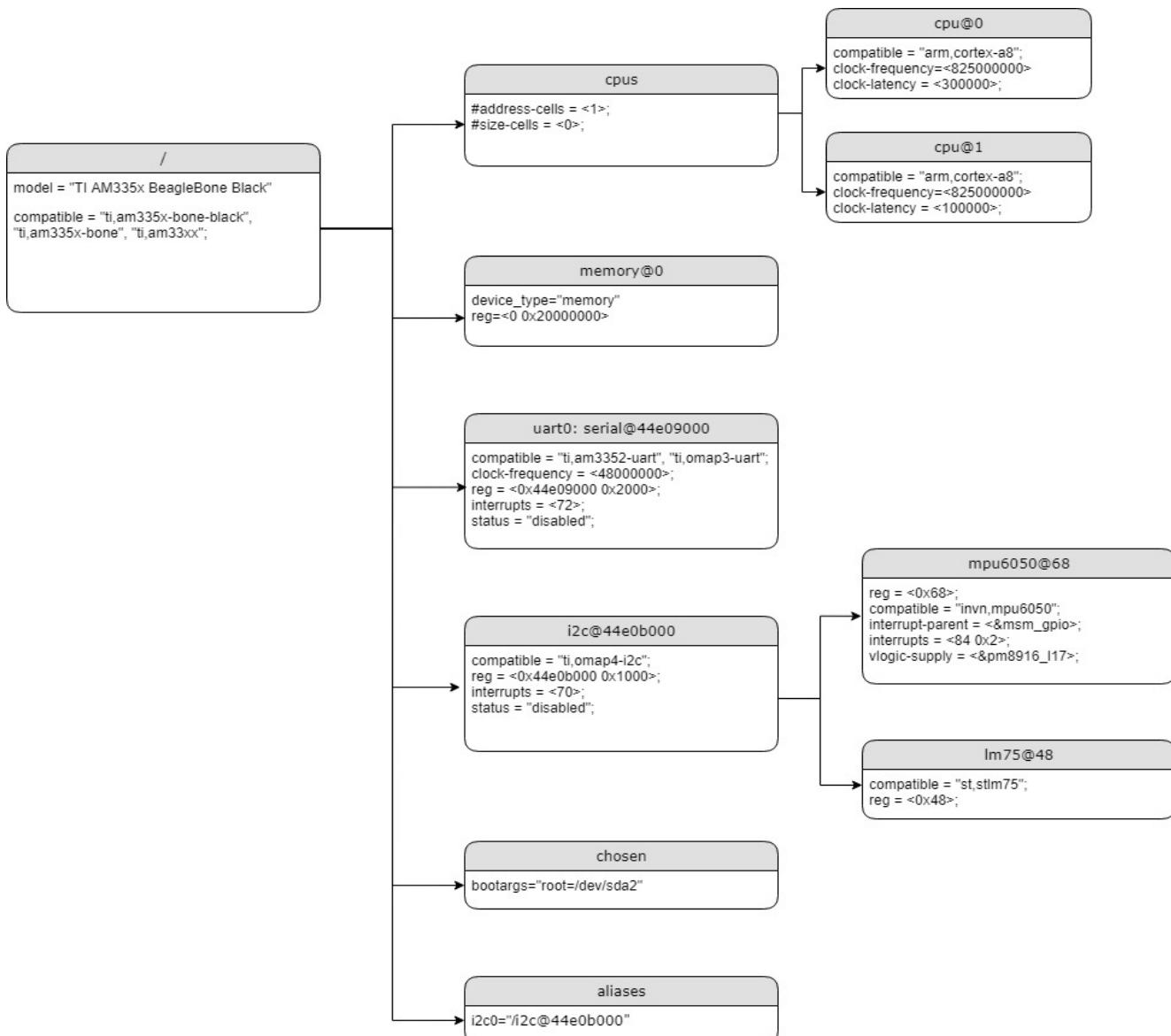
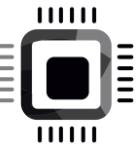
Modular approach to manage DT files

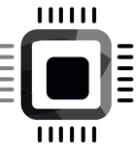




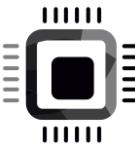
Overview of device tree structure

- ✓ Device tree is a collection of device nodes
- ✓ A ‘device node’ or simply called ‘a node’ represents a device. Nodes are organized in some systematic way inside the device tree file.
- ✓ They also have parent and child relationship, and every device tree must have one root node
- ✓ A node explains itself, that is, reveals its data and resources using its “properties.”





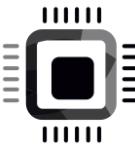
```
/ {  
    Node-1 {  
        a-string-property = "A string";  
        A-32-bit-integer-property = <800>;  
        string-list-property = "1ststring",2ndstring";  
        Child-Node-1 {  
            byte-array-property = <0x30 0x20 0xFE 0x10>;  
        };  
    };  
    Node-2 {  
        A-Boolean-property;  
        Child-Node-1 {  
        };  
        Child-Node-2 {  
        };  
    };  
};
```



Root node

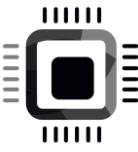
- The device tree has a single root node of which all other device nodes are descendants. The full path to the root node is /.
- All device trees shall have a root node, and the following nodes shall be present at the root of all device trees:
 - One /CPUs node
 - At least one /memory node

*Chapter 3 :DEVICE NODE REQUIREMENTS
DeviceTree Specification Release v0.3a*

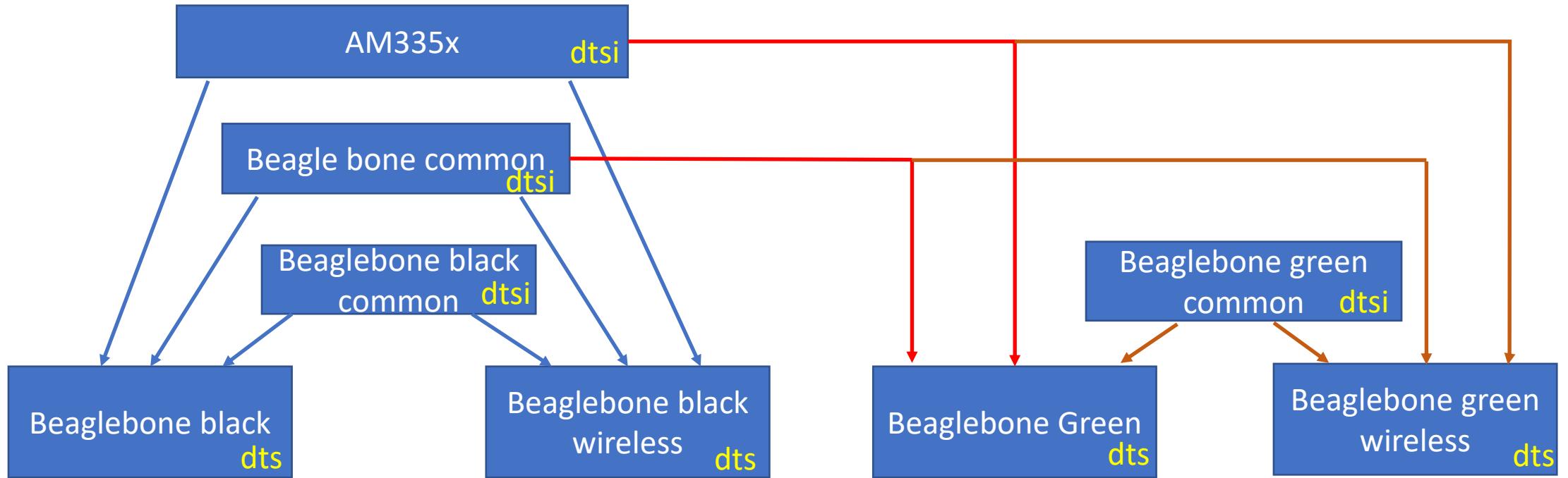


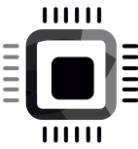
Summary

- Remember that you most probably be writing device tree addons or overlays for your board-related changes but not for entire soc.
- The soc specific device tree will be given by the vendor in the form of device tree inclusion file (.dtsi) and you just need to include that in your board-level device tree
- Follow modulatory approach while writing device tree



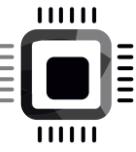
Device tree modularity





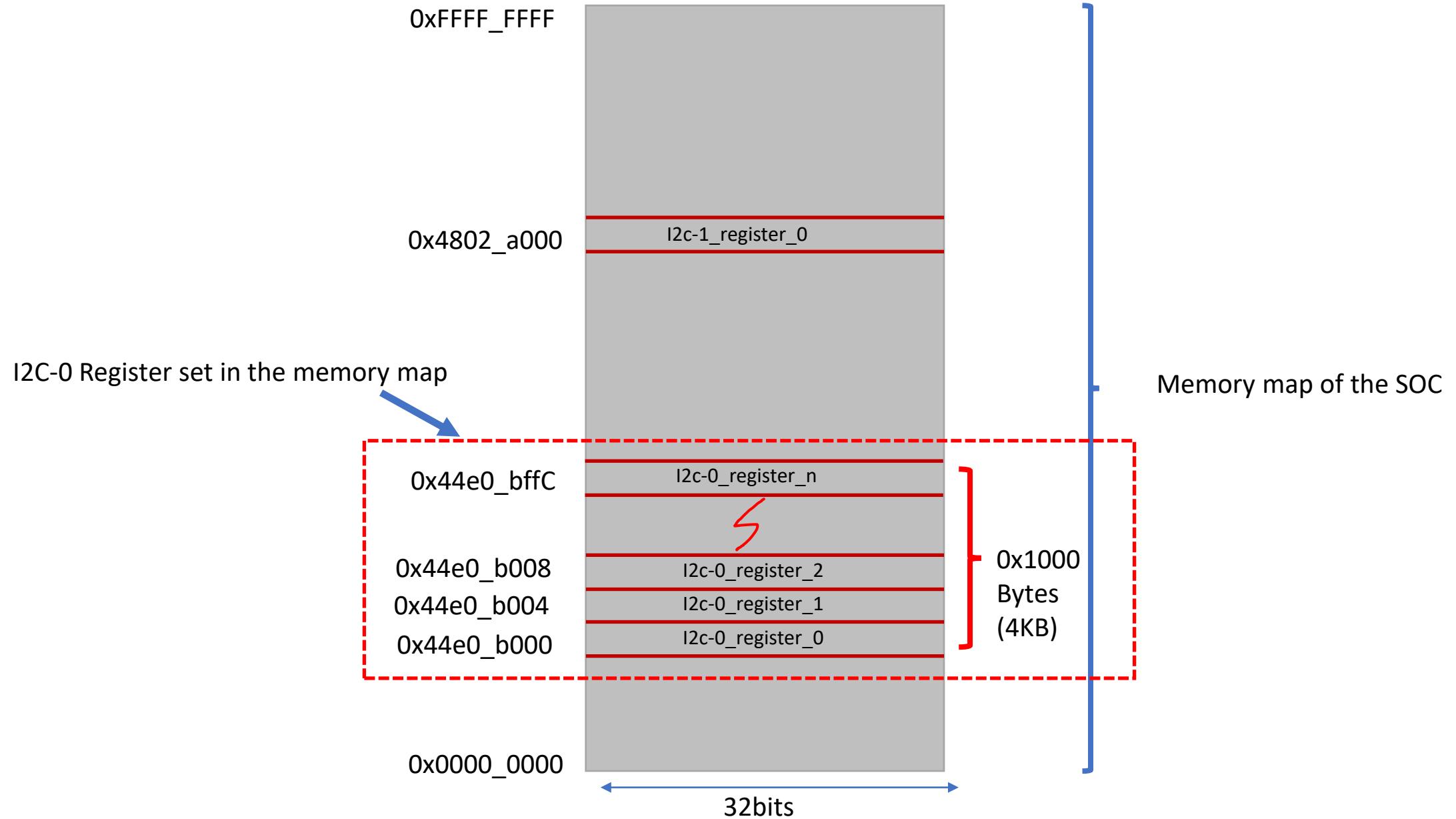
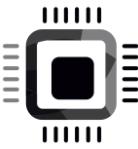
Device tree writing syntax

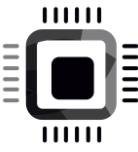
- Node name
- Node Label
- Standard and non-standard property names
- Different data type representation (u32, byte, byte stream, string, stream of strings, Boolean, etc)



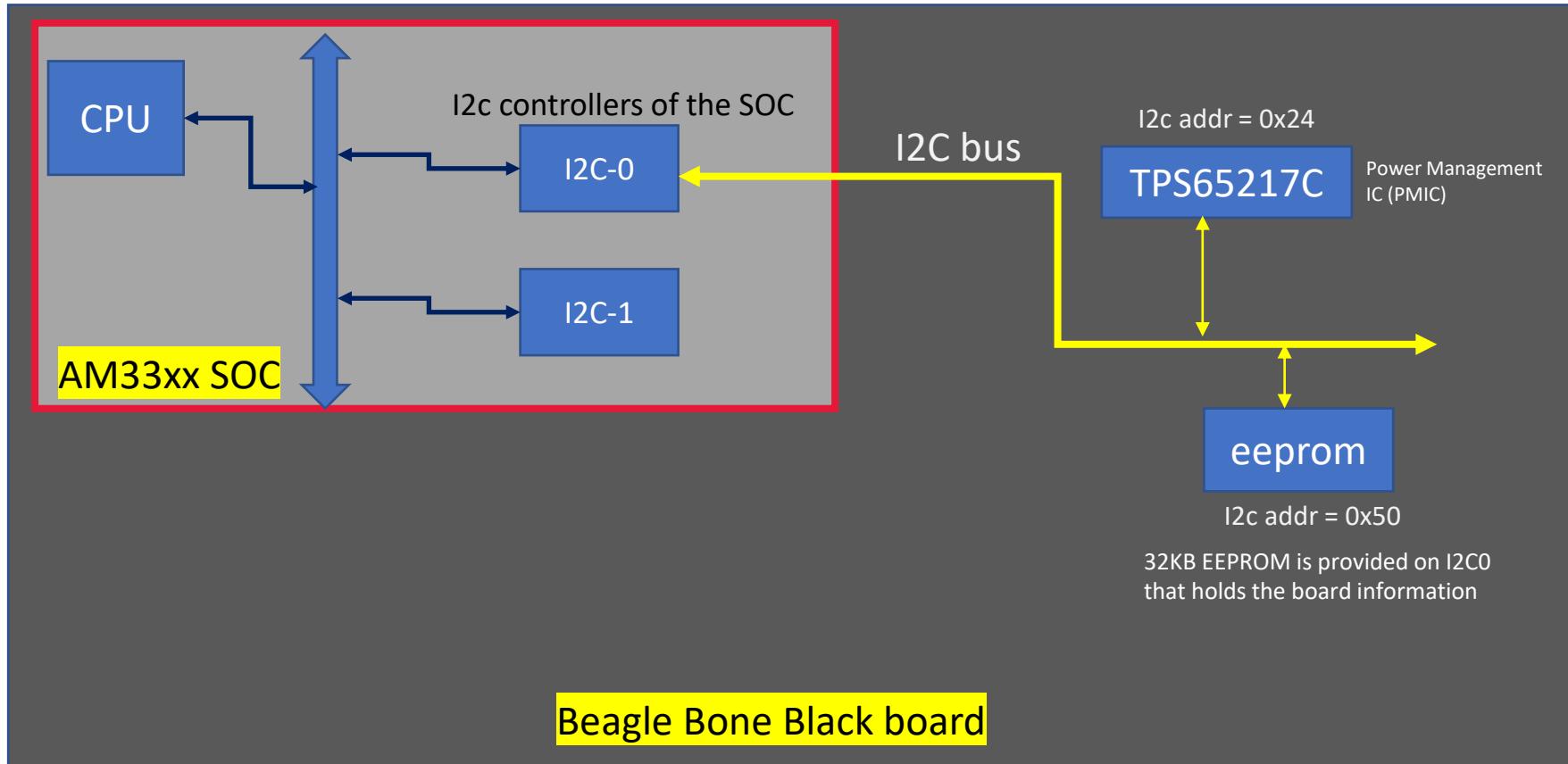
Node name

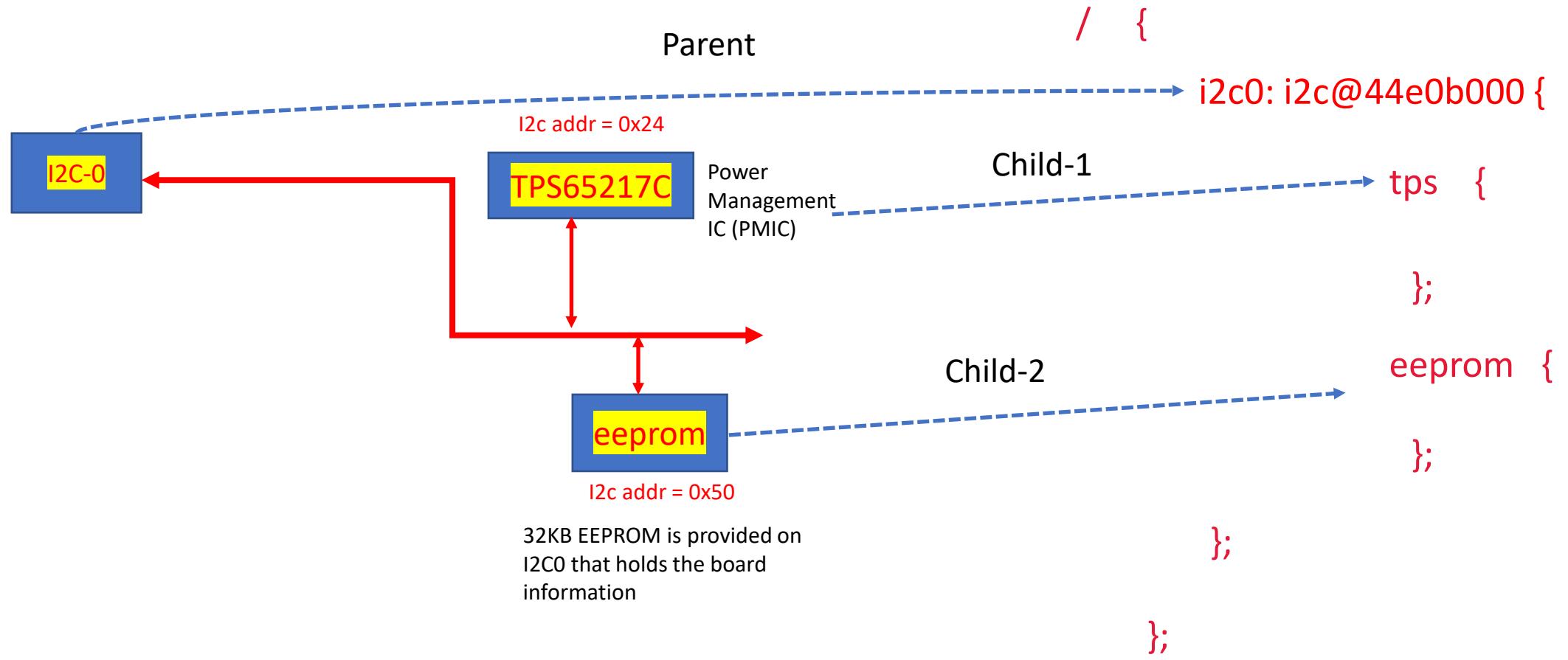
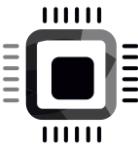
- Refer device tree specification Release v0.3 from devicetree.org



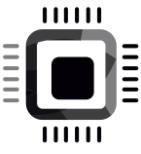


Parent and child nodes

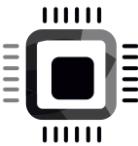




Explaining hardware details via device tree nodes



Example from bbb



```

leds {
    pinctrl-names = "default";
    pinctrl-0 = <&user_leds_s0>;
    compatible = "gpio-leds";

    led2 {
        label = "beaglebone:green:usr0";
        gpios = <&gpio1 21 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "heartbeat";
        default-state = "off";
    };

    led3 {
        label = "beaglebone:green:usr1";
        gpios = <&gpio1 22 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "mmc0";
        default-state = "off";
    };

    led4 {
        label = "beaglebone:green:usr2";
        gpios = <&gpio1 23 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "cpu0";
        default-state = "off";
    };

    led5 {
        label = "beaglebone:green:usr3";
        gpios = <&gpio1 24 GPIO_ACTIVE_HIGH>;
        linux,default-trigger = "mmc1";
        default-state = "off";
    };
};

```

am335x-bone-common.dtsi

```

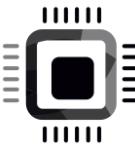
uart0: serial@44e09000 {
    compatible = "ti,am3352-uart", "ti,omap3-uart";
    ti,hwmods = "uart1";
    clock-frequency = <48000000>;
    reg = <0x44e09000 0x2000>;
    interrupts = <72>;
    status = "disabled";
    dmas = <&edma 26 0>, <&edma 27 0>;
    dma-names = "tx", "rx";
};

uart1: serial@48022000 {
    compatible = "ti,am3352-uart", "ti,omap3-uart";
    ti,hwmods = "uart2";
    clock-frequency = <48000000>;
    reg = <0x48022000 0x2000>;
    interrupts = <73>;
    status = "disabled";
    dmas = <&edma 28 0>, <&edma 29 0>;
    dma-names = "tx", "rx";
};

uart2: serial@48024000 {
    compatible = "ti,am3352-uart", "ti,omap3-uart";
    ti,hwmods = "uart3";
    clock-frequency = <48000000>;
    reg = <0x48024000 0x2000>;
    interrupts = <74>;
    status = "disabled";
    dmas = <&edma 30 0>, <&edma 31 0>;
    dma-names = "tx", "rx";
};

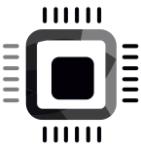
```

am33xx.dtsi

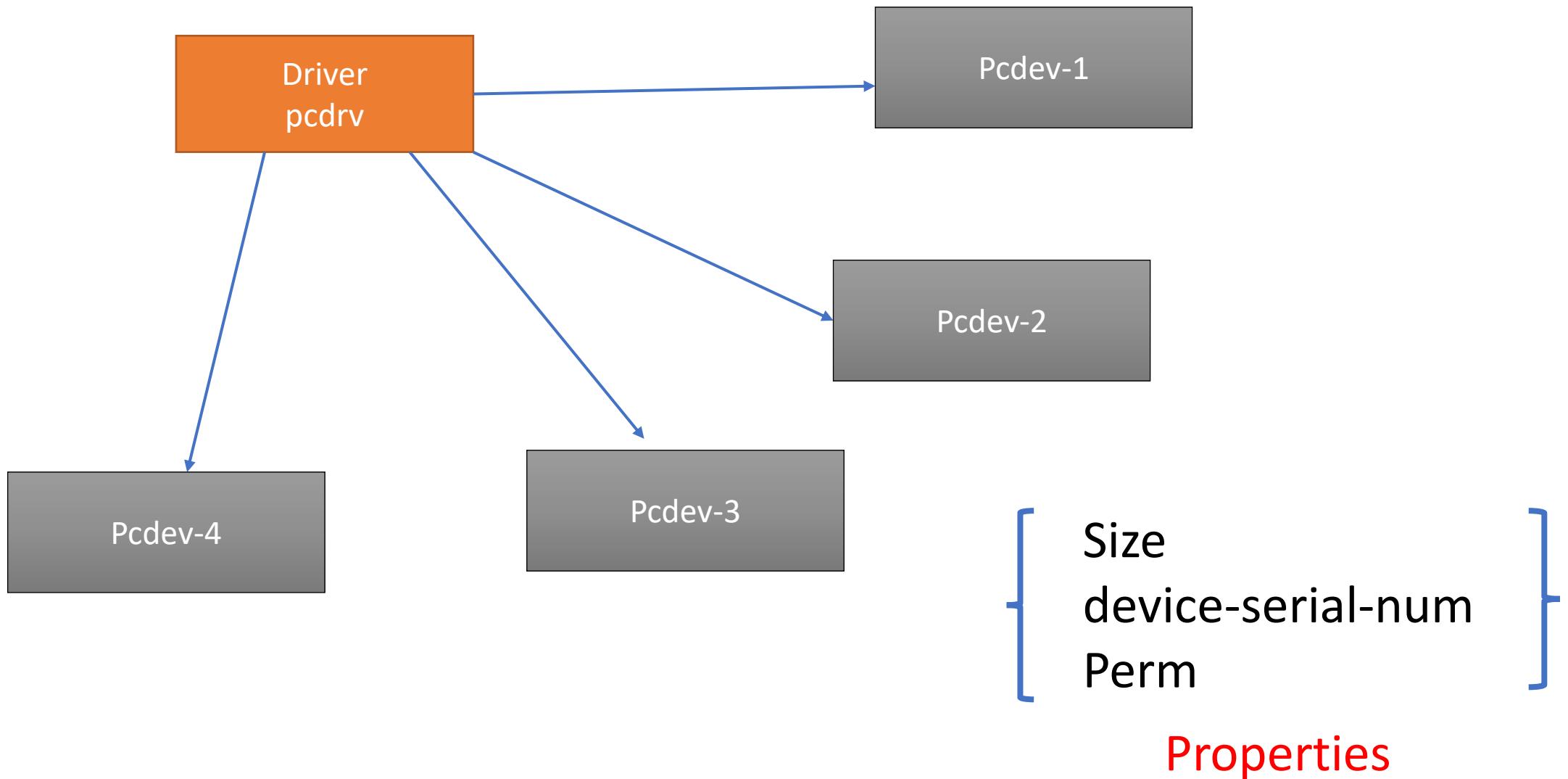
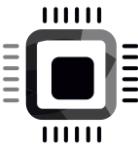


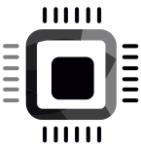
Key points

- Node name is in the format **node-name@unit-address**
- This combination differentiates the node from other nodes at the same level in the tree
- To write the “node-name” alphabet (stick to lower case), and number combinations are allowed. So choose an appropriate name , it would be good if it represents the general class of a device like serial, i2c, led, etc
- The unit-address must match the first address specified in the **reg property** of the node. If the node has no reg property, the **@unit-address** must be omitted

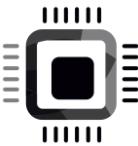


PCD example





Properties



What is a property ?

```
leds {  
    pinctrl-names = "default";  
    pinctrl-0 = <&user_leds_s0>;  
  
    compatible = "gpio-leds";  
  
    led2 {  
        label = "beaglebone:green:usr0";  
        gpios = <&gpio1 21 GPIO_ACTIVE_HIGH>;  
        linux,default-trigger = "heartbeat";  
        default-state = "off";  
    };  
  
    led3 {  
        label = "beaglebone:green:usr1";  
        gpios = <&gpio1 22 GPIO_ACTIVE_HIGH>;  
        linux,default-trigger = "mmc0";  
        default-state = "off";  
    };  
  
    led4 {  
        label = "beaglebone:green:usr2";  
        gpios = <&gpio1 23 GPIO_ACTIVE_HIGH>;  
        linux,default-trigger = "cpu0";  
        default-state = "off";  
    };  
  
    led5 {  
        label = "beaglebone:green:usr3";  
        gpios = <&gpio1 24 GPIO_ACTIVE_HIGH>;  
        linux,default-trigger = "mmc1";  
        default-state = "off";  
    };  
};
```

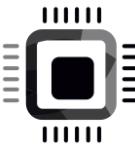
Property name(key)

label = "beaglebone:green:usr0";

Value

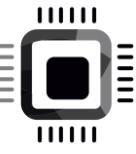
A single property

Properties

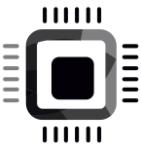


Different types of properties

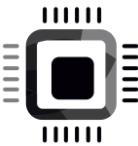
- Standard property
- Custom property (non-standard)
- Standard properties are those which is explained by the specification and the device-driver binding documentation
- Custom properties are specific to a particular vendor or organization which is not documented by the specification.
- That is why, when you use custom property, always begin with your organization name.



- Lets check the specification to view some properties



'compatible' property



Property :compatible

compatible = <string-list>

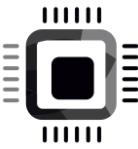
property name(Standard)

Value type

e.g. :

```
compatible = "string1" , "string2" , "string3";
```

- Compatible property of a device node is used by the Linux kernel to match and load an appropriate driver to handle that device node (driver selection)
- The string list provided by the compatible property is matched against the string list supported by the driver. If the kernel finds any match, the driver will be loaded, and probe function is invoked



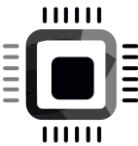
Root compatible property of BBB

```
/ {  
    model = "TI AM335x BeagleBone Black";  
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";  
};
```



Sorted string list from most compatible to least.

Root compatible property is used for machine identification



Root compatible property of beaglebone bone black

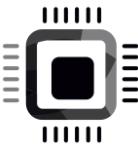
```
/ {  
    model = "TI AM335x BeagleBone Black";  
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";  
};
```

It claims that, this is an exact compatible name of the board for kernel to support it

It claims that, it is also compatible with kernel which supports bone devices based on am335x, if the kernel doesn't support previous exact model.

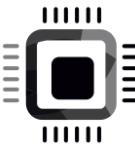
It claims that, it is also compatible with kernel which supports only am33xx soc based board, if kernel doesn't support previous options.

The machine identification fails when none of these string values matches with kernel supported compatible machine identification string-list.



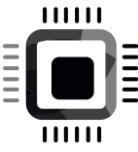
Uses of compatible property

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



Device tree bindings

- How do you know which property name and value pair should be used to describe a node in the device tree?
 - Device tree binding document. The driver writer must document these details
- The properties that are necessary to describe a device in the device tree depends on the requirements of the Linux driver for that device
- When all the required properties are provided, the driver of charge can detect the device from the device tree and configure it.



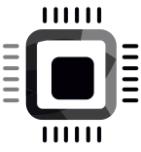
Note:

- Compatible strings and properties are first defined by the client program (OS , drivers) then shared with DT writer



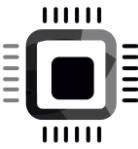
Driver decides compatible strings and documents in device tree binding document

DT writer refers to the binding document and initializes relevant values for the compatible property



Some examples of device and driver binding

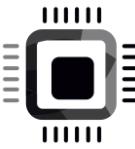
- Example of lm75
- mpu 6050



Device tree bindings documentation

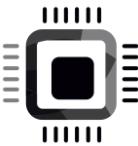
- All device tree binding document can be found here

<https://www.kernel.org/doc/Documentation/devicetree/bindings/>



Device tree bindings- points to remember

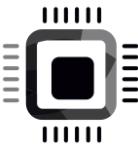
- Case 1 : When the driver is already available in the Linux kernel for the device ‘x,’ but you just need to add device ‘x’ entry in the device tree then you must consult ‘x’ drivers binding document which guides you through creating device tree node for device ‘x.’
- Case 2 : When the driver is not available for the device ‘x,’ then you should write your own driver, you should decide what properties to use (could be a combination of standard and non-standard property), you should then provide the device tree binding document describing what are all the properties and compatible strings a device tree write must include.



Linux conventions to write device tree

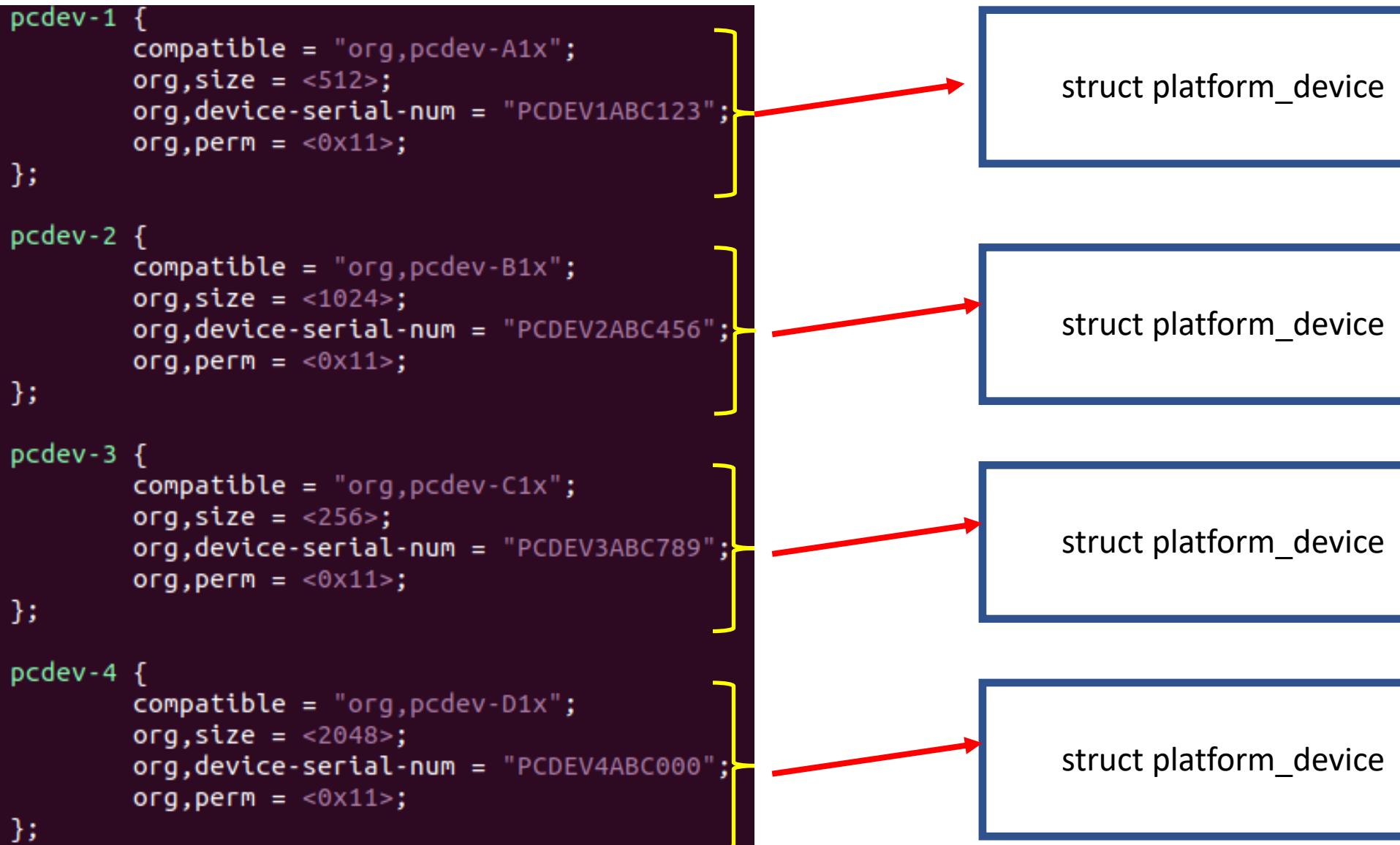
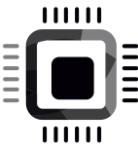
- hex constants are lower case
 - use "0x" instead of "0X"
 - use a..f instead of A..F, eg 0xf instead of 0xF
- node names
 - should begin with a character in the range 'a' to 'z', 'A' to 'Z'
 - unit-address does not have a leading "0x" (the number is assumed to be hexadecimal)
 - unit-address does not have leading zeros
 - use dash "-" instead of underscore "_"
- label names
 - should begin with a character in the range 'a' to 'z', 'A' to 'Z'
 - should be lowercase
 - use underscore "_" instead of dash "-"
- property names
 - should be lower case
 - should begin with a character in the range 'a' to 'z'
 - use dash "-" instead of underscore "_"

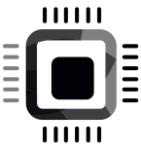
https://elinux.org/Device_Tree_Linux#Linux_vs_ePAPR_Version_1.1



pcdev properties

```
org,size = <512>; //this is a non standard property and represents integer data;  
org,device-serial-num = "PCDEV1ABC123"; //this is a non standard property and  
represents string data  
org,perm = <0x11>; //this is a non standard property and represents integer data
```

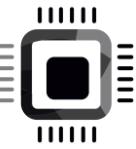




In Linux every device is represented by an instance of struct device.

platform_device

Struct device



```
pcdev-1 {  
    compatible = "org,pcdev-A1x";  
    org.size = <512>;  
    org.device-serial-num = "PCDEV1ABC123";  
    org.perm = <0x11>;  
};
```

For every device node **struct device_node** represents details of the device node

associated device tree node

firmware device node

Struct platform_device

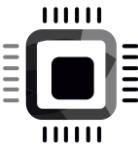
Struct device

Struct device_node

of_node

Struct fwnode_handle

fwnode

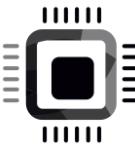


Functions to interpret device tree nodes

- Refer : include/linux/of.h
- Procedures for creating, accessing and interpreting the device tree
- Linux/drivers/of/base.c

```
/*associated device tree node */
struct device_node *np = dev->of_node;
int size;

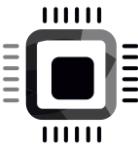
if(of_property_read_u32(np, "fb, size", &size)){
    dev_info(dev, "Missing size property\n");
    return -EINVAL;
}
```



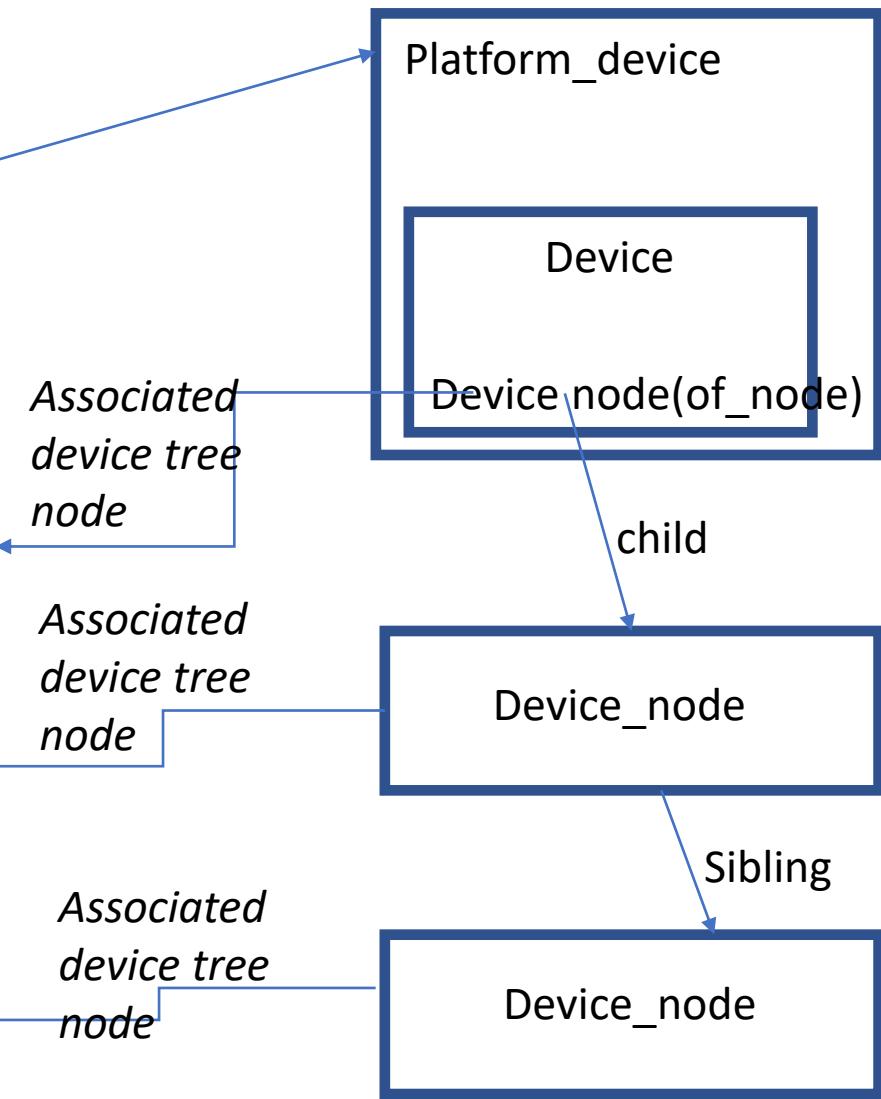
```
i2c@3000 {  
    #address-cells = <1>;  
    #size-cells = <0>;  
    cell-index = <0>;  
    compatible = "fsl-i2c";  
    reg = <0x3000 0x100>;  
    interrupts = <43 2>;  
    interrupt-parent = <&mpic>;  
    dfsrr;  
    dtt@48 {  
        compatible = "national,lm75";  
        reg = <0x48>;  
    };  
    rtc@68 {  
        compatible = "dallas,ds1337";  
        reg = <0x68>;  
    };  
};
```

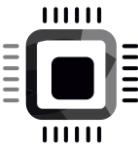
we can make the following observations about this example node:

- The I2C controller is located at offset 0x3000 from its parent.
- The driver for the I2C controller is fsl-i2c.
- The first child is named dtt, at offset 0x48 from its parent; the driver is national lm75.
- The second child is named rtc, at offset 0x68 from its parent; the driver is Dallas ds1337.



```
i2c@3000 {  
    #address-cells = <1>;  
    #size-cells = <0>;  
    cell-index = <0>;  
    compatible = "fsl-i2c";  
    reg = <0x3000 0x100>;  
    interrupts = <43 2>;  
    interrupt-parent = <&mpic>;  
    dfsrr;  
    dtt@48 {  
        compatible = "national,lm75";  
        reg = <0x48>;  
    };  
    rtc@68 {  
        compatible = "dallas,ds1337";  
        reg = <0x68>;  
    };  
};
```

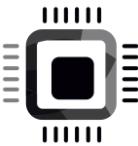




Device node

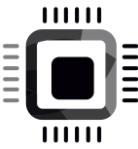
```
struct device_node {
    const char *name;
    phandle phandle;
    const char *full_name;
    struct fwnode_handle fwnode;

    struct property *properties;
    struct property *deadprops; /* removed properties */
    struct device_node *parent;
    struct device_node *child;
    struct device_node *sibling;
#if defined(CONFIG_OF_KOBJ)
    struct kobject kobj;
#endif
    unsigned long _flags;
    void *data;
#if defined(CONFIG_SPARC)
    unsigned int unique_id;
    struct of_irq_controller *irq_trans;
#endif
};
```



Property

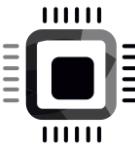
```
struct property {
    Char      *name;
    Int       length;
    Void      *value;
    struct property *next;
#ifndef CONFIG_OF_DYNAMIC || !defined(CONFIG_SPARC)
    unsigned long _flags;
#endif
#ifndef CONFIG_OF_PROMTREE
    unsigned int unique_id;
#endif
#ifndef CONFIG_OF_KOBJ
    struct bin_attribute attr;
#endif
};
```



Get driver data from matched device

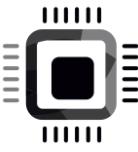
```
void *of_device_get_match_data(const struct device *dev);
```

```
include/linux/of_device.h
```



CONFIG_OF configuration item

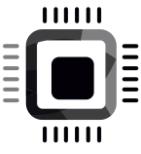
- In Linux, CONFIG_OF configuration item decides Device Tree and Open Firmware support
- If CONFIG_OF is not enabled, Linux doesn't support hardware enumeration via device tree.
- All device tree processing functions which begin with `of_*` will be excluded from the kernel build
- For latest kernel this configuration item is enabled by default



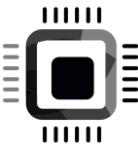
```
//if CONFIG_OF enabled
#define of_match_ptr(_ptr)  (_ptr)

//if CONFIG_OF disabled
#define of_match_ptr(_ptr)  NULL
```

Include/linux/of.h

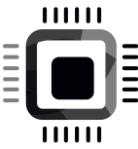


Device tree overlays



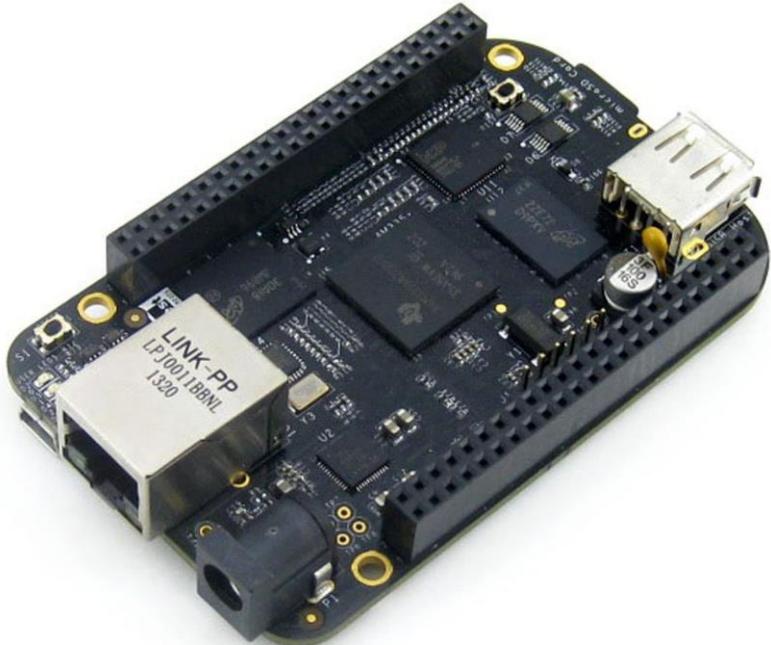
Device tree overlays

DT overlays are device tree patches(dtbo) which are used to patch or modify the existing main device tree blob(dtb)



am335x-boneblack.dtb

(This is board specific. This explains the hardware topology of the board)

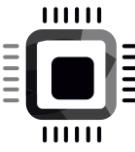


BBB



BBB LCD cape

The main dtb doesn't include the hardware details (device nodes, properties , pin configs) to configure the cape.

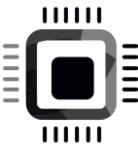


Two ways you can include the device nodes for the cape device in the main dtb

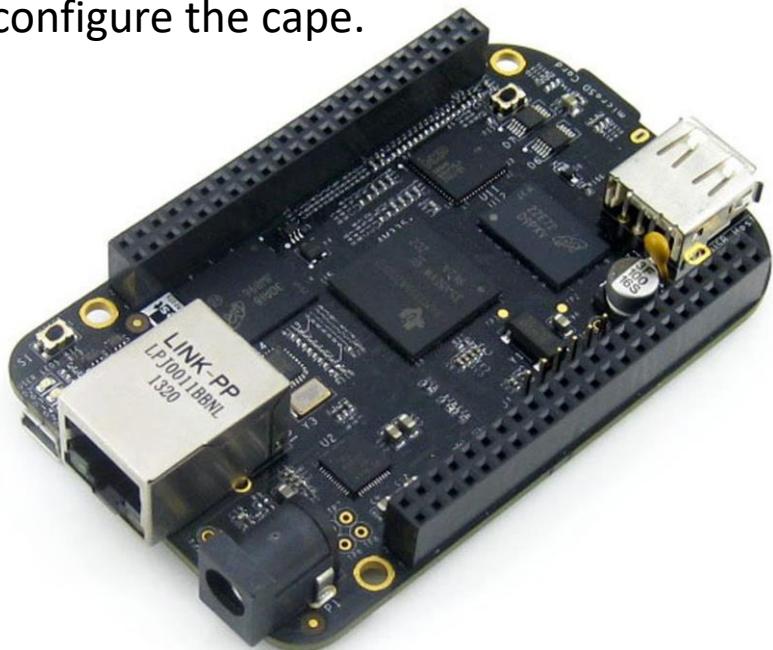
- 1.Edit the main dtb itself (not recommended)
- 2.Overlay(a Patch which overlays the main dtb) (recommended)

Uses of overlays

- 1)To support and manage hardware configuration (properties, nodes, pin configurations) of various capes of the board
- 2)To alter the properties of already existing device nodes of the main dtb
- 3)Overlays approach maintains modularity and makes capes management easier



The main dtb doesn't include the hardware details (device nodes, and properties) to configure the cape.



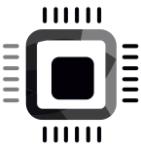
BBB

am335x-boneblack.dtb (this is board specific. this explains the hardware topology of the board)



BBB LCD cape

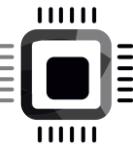
I2c-touch-lcd.dts(this is overlay source file)
I2c-touch-lcd.dtbo(.dtbo indicates that this is a overlay which should be used with this LCD)



Overlay DTS Format

- Refer

<https://www.kernel.org/doc/Documentation/devicetree/overlay-notes.txt>



```
/*----board.dts-----*/
/dts-v1/;
/{
    compatible = "org, board-am335x";

    ocp: ocp {
        /* on chip peripherals*/
        i2c0: i2c@48000000{
            status = "disabled"; /* This device node is disabled */
            /*some other properties of i2c0*/
        };
    };
}/*----board.dts-----*/
```



```
/*----board.dts + rtc.dts(overlay)-----*/
/dts-v1/;
/{
    compatible = "org, board-am335x";

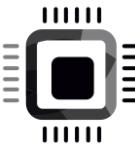
    ocp: ocp {
        /* on chip peripherals*/
        i2c0: i2c@48000000{
            status = "ok";
            /*some other properties of i2c0*/
            /*RTC peripheral*/
            rtc_xyx@51 {
                reg = <0x51>;
            };
        };
    };
}/*----board.dts + rtc.dts(overlay)-----*/
```

```
/*----rtc.dts(overlay)-----*/
/dts-v1/;
/plugin/;
/{
    fragment@0 {
        target = <&i2c0>;

        __overlay__ {
            status = "ok";
            /*RTC peripheral*/
            rtc_xyx@51 {
                reg = <0x51>;
            };
        };
    };
}; /*End of fragment */
}; /*End of overlay */

/*----rtc.dts(overlay)-----*/
```

/dts-v1/; → Indicates DTS file version
/plugin/; → Indicates it's a plugin (an overlay)

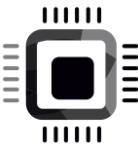


DT Overlay exercise

Write a device tree overlay to disable/modify pcdev device nodes from the main dts

STEPS

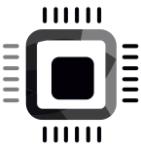
1. Create a device tree overlay file and add fragments to modify the target device nodes
2. Compile the device tree overlay file to generate .dtbo file (Device tree overlay binary)
3. Make u-boot to load the .dtbo file during board start-up



Overlay compilation

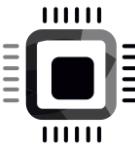
- Make sure that device tree compiler(dtc) is installed on your system
- Run the below command to generate .dtbo from .dts file

```
dtc -O dtb -o <output-file-name> -I <input-file-name>
```



Kernel memory allocation APIs

- kmalloc ()
- kfree ()



```
void* kmalloc( size_t size, gfp_t flags);
```

include/linux/slab.h

Used to allocate memory in kernel space by drivers and kernel functions

Memory obtained are physically(RAM) contiguous

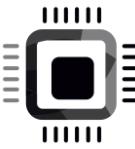
From the kernel documentation:

“ The maximal size of a chunk that can be allocated with *kmalloc* is limited. The actual limit depends on the hardware and the kernel configuration, but it is a good practice to use *kmalloc* for objects smaller than page size ”

Page size = 4KiB (4 * 1024 Bytes)

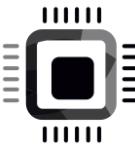
flags : changes the behaviour of the underlying memory allocator

return = NULL if allocation fails, on success virtual address of the first page allocated

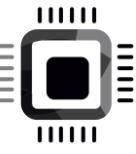


```
/**  
 * kmalloc - allocate memory  
 * @size: how many bytes of memory are required.  
 * @flags: the type of memory to allocate.  
  
 * kmalloc is the normal method of allocating memory  
 * for objects smaller than page size in the kernel.  
  
 * The allocated object address is aligned to at least ARCH_KMALLOC_MINALIGN  
 * bytes. For @size of power of two bytes, the alignment is also guaranteed  
 * to be at least to the size.  
  
 * The @flags argument may be one of the GFP flags defined at  
 * include/linux/gfp.h and described at  
 * :ref:`Documentation/core-api/mm-api.rst <mm-api-gfp-flags>`  
  
 * The recommended usage of the @flags is described at  
 * :ref:`Documentation/core-api/memory-allocation.rst <memory-allocation>`  
  
 * Below is a brief outline of the most useful GFP flags
```

include/linux/slab.h

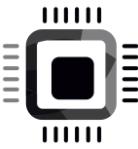


```
/*
 *%GFP_KERNEL : Allocate normal kernel ram. May sleep.
 *%GFP_NOWAIT : Allocation will not sleep.
 *%GFP_ATOMIC Allocation will not sleep. May use emergency pools.
 *%GFP_HIGHUSER Allocate memory from high memory on behalf of user.
 *
 * Also it is possible to set different flags by OR'ing
 * in one or more of the following additional @flags:
 *
 * %__GFP_HIGH This allocation has high priority and may use emergency pools.
 * %__GFP_NOFAIL Indicate that this allocation is in no way allowed to fail
 * (think twice before using).
 * %__GFP_NORETRY If memory is not immediately available, then give up at once.
 * %__GFP_NOWARN If allocation fails, don't issue any warnings.
 * %__GFP_RETRY_MAYFAIL Try really hard to succeed the allocation but fail
 * eventually.
 */
include/linux/slab.h
```



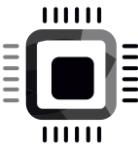
Usage

```
struct bar *k;  
k = kmalloc(sizeof(*k), GFP_KERNEL);  
if (!k)  
    return -ENOMEM;
```



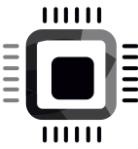
kfree

```
/**  
 * kfree - free previously allocated memory  
 * @objp: pointer returned by kmalloc.  
 *  
 * If @objp is NULL, no operation is performed.  
 *  
 * Don't free memory not originally allocated by kmalloc()  
 * or you will run into trouble.  
 */  
void kfree(const void *objp);
```



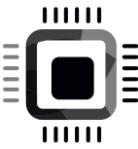
Kmalloc and friends

- **kmalloc_array()**
- **kcalloc()**
- **kzalloc()**
- **krealloc()**



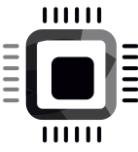
Kzalloc is preferred over kmalloc

```
/**  
 * kzalloc - allocate memory. The memory is set to zero.  
 * @size: how many bytes of memory are required.  
 * @flags: the type of memory to allocate (see kmalloc).  
 */  
void *kzalloc(size_t size, gfp_t flags);
```



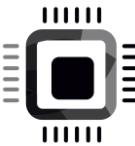
```
struct bar {  
    int a ;  
    int b;  
    char *name;  
}  
  
struct bar *pbar;  
  
pbar = kmalloc( sizeof(*pbar) , GFP_KERNEL);  
  
bar_processing_fun(pbar);
```

```
void bar_processing_fun(struct bar *pbar)  
{  
    if(! (pbar->name) )  
        //allocate memory for 'name'  
    else  
        //memory for 'name' is already allocated  
  
    /* This may crash if pbar->name is not a valid pointer */  
    memcpy(pbar->name, "name",5);  
}
```



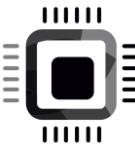
kzfree

```
/**  
 * kzfree - like kfree but zero memory  
 * @p: object to free memory of  
 *  
 * The memory of the object @p points to is zeroed before freed.  
 * If @p is %NULL, kzfree() does nothing.  
 *  
 * Note: this function zeroes the whole allocated buffer which can be a good  
 * deal bigger than the requested buffer size passed to kmalloc(). So be  
 * careful when using this function in performance sensitive code.  
 */  
void kzfree(const void *p);
```



```
/**  
 * krealloc - reallocate memory. The contents will remain unchanged.  
 * @p: object to reallocate memory for.  
 * @new_size: how many bytes of memory are required.  
 * @flags: the type of memory to allocate.  
  
 * The contents of the object pointed to are preserved up to the  
 * lesser of the new and old sizes. If @p is %NULL, krealloc()  
 * behaves exactly like kmalloc(). If @new_size is 0 and @p is not a  
 * %NULL pointer, the object pointed to is freed.  
  
 * Return: pointer to the allocated memory or %NULL in case of error  
 */
```

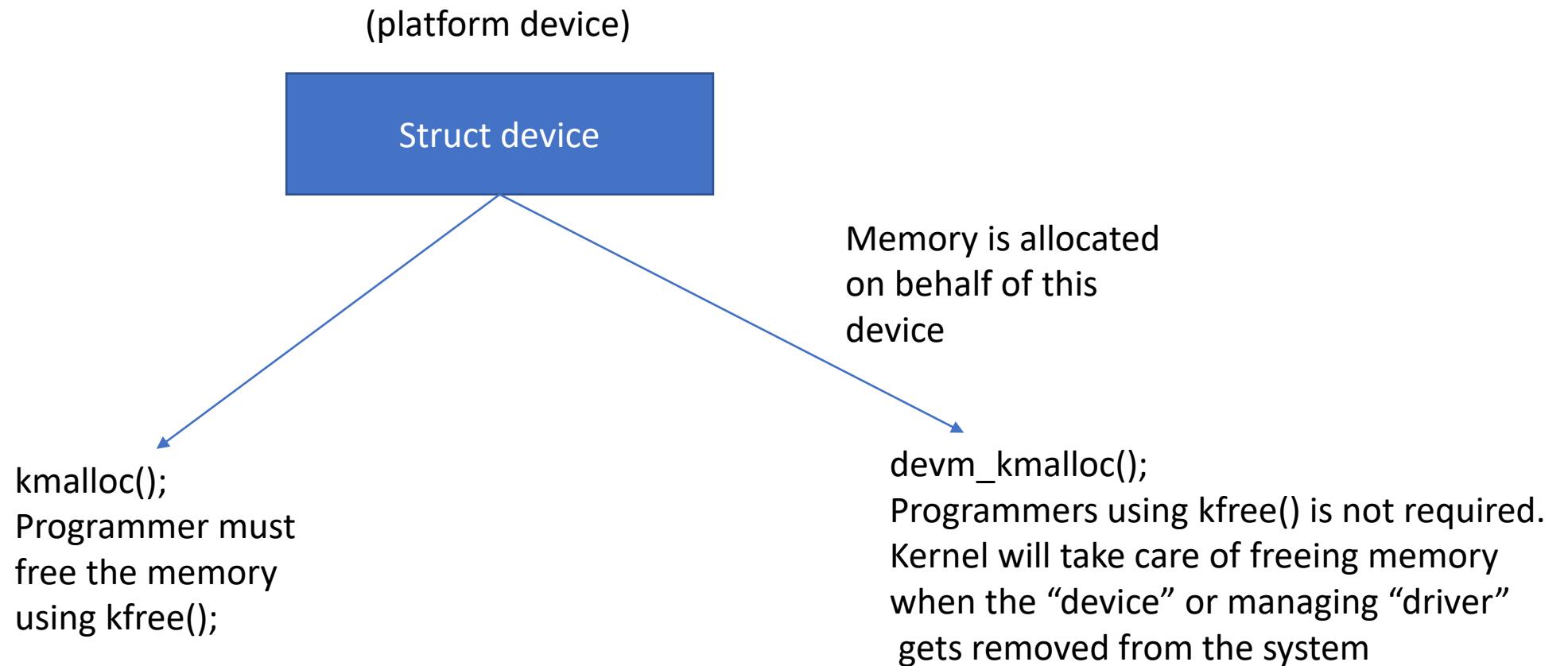
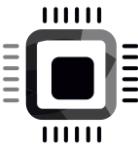
void *krealloc(const void *p, size_t new_size, gfp_t flags);

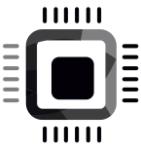


Resource managed kernel APIs

`kmalloc()` //This allocates a resource (kernel memory)

`devm_kmalloc()` //This also allocates a resource but it “remembers” what has been allocated. (This is resource managed API)





Examples

Kmalloc()



Kfree()



gpiod_get()



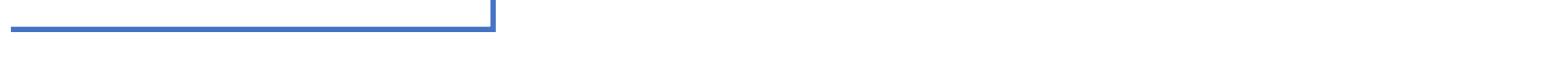
gpiod_put()



request_irq()

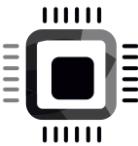


free_irq()



<https://www.kernel.org/doc/Documentation/driver-model/devres.txt>

Refer to this link to check all resource managed kernel apis



Probe function

```
int probe_function(struct platform_device *pdev)
{
    struct dev_data *data ;

    data = kmalloc();
    if(!data)
        return -ENOMEM;

    ret = do_something();
    if(ret < 0)
        /*Something went wrong */
        goto free_mem;

    return 0;

free_mem:
    kfree(data);

    return ret;
}
```

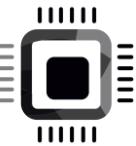


```
int probe_function(struct platform_device *pdev)
{
    struct dev_data *data ;

    data = devm_kmalloc(&pdev->dev, );
    if(!data)
        return -ENOMEM;

    ret = do_something();
    if(ret < 0)
        /*Something went wrong */
        return -EINVAL;

    return 0;
}
```



remove function

```

/*gets called when the device is removed from the system */
int pcd_platform_driver_remove(struct platform_device *pdev)
{
#ifndef PCD_PLATFORM_DRIVER_REMOVE
    struct pcdev_private_data  *dev_data = dev_get_drvdata(&pdev->dev);
    /*1. Remove a device that was created with device_create() */
    device_destroy(pcdrv_data.class_pcd,dev_data->dev_num);

    /*2. Remove a cdev entry from the system*/
    cdev_del(&dev_data->cdev);
    kfree(dev_data->buffer);
    kfree(dev_data);
    pcdrv_data.total_devices--;
#endif
    dev_info(&pdev->dev,"A device is removed\n");
    return 0;
}

```

```

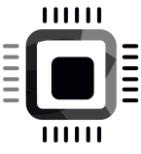
/*gets called when the device is removed from the system */
int pcd_platform_driver_remove(struct platform_device *pdev)
{
#ifndef PCD_PLATFORM_DRIVER_REMOVE
    struct pcdev_private_data  *dev_data = dev_get_drvdata(&pdev->dev);
    /*1. Remove a device that was created with device_create() */
    device_destroy(pcdrv_data.class_pcd,dev_data->dev_num);

    /*2. Remove a cdev entry from the system*/
    cdev_del(&dev_data->cdev);

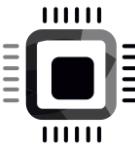
    pcdrv_data.total_devices--;
#endif
    dev_info(&pdev->dev,"A device is removed\n");
    return 0;
}

```

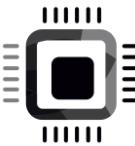
When you use devm_kmalloc() to allocate the memory in “probe” function, using kfree() in remove function is no longer required



Linux Device model

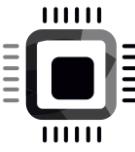


- Linux device model is nothing but a collection of various data structures , and helper functions that provide a unifying and hierarchical view of all the busses, devices, drivers present on the system. You can access the whole Linux device and driver model through a pseudo filesystem called sysfs. Which is mounted at /sysfs.
- Different components of the Linux device model is represented as files and directories through sysfs
- Sysfs exposes underlying bus, device, and driver details and their relationships in the Linux device model.



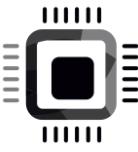
Components of the device model

- Device → struct device
- Device driver → struct device_driver
- Bus → struct bus_type
- Kobject → struct kobject
- Ksets → struct kset
- Kobject type → struct kobj_type



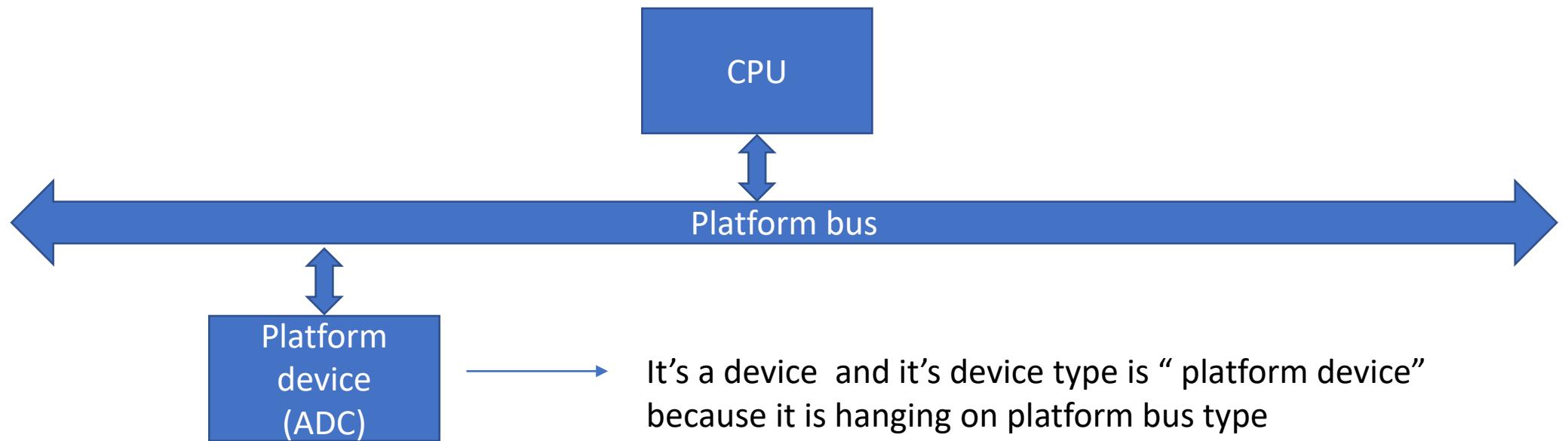
Device and Driver representation

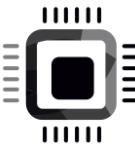
- Definition of a device
 - Under Linux device driver model, anything which can be represented by an instance of the data structure `struct device` is a device
- Definition of a driver
 - Anything which can be represented by an instance of the data structure `struct device_driver` is a driver



Example

Consider the case of a platform device





```

struct device {
    struct kobject kobj;
    struct device     *parent;

    struct device_private  *p;

    const char      *init_name; /* initial name of the device */
    const struct device_type *type;

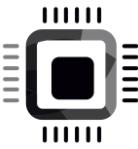
    struct bus_type *bus;        /* type of bus device is on */
    struct device_driver *driver; /* which driver has allocated this
                                    device */
    void          *platform_data; /* Platform specific data, device
                                    core doesn't touch it */
    void          *driver_data;   /* Driver data, set and get with
                                    dev_set_drvdata/dev_get_drvdata */

#ifdef CONFIG_PROVE_LOCKING
    struct mutex      lockdep_mutex;
#endif
    struct mutex      mutex; /* mutex to synchronize calls to
                            * its driver.
                            */
};


```

Include/linux/device.h

- At the lowest level, every device in a Linux system is represented by an instance of struct device
- The device structure contains the information that the device model core needs to model the system
- Most subsystems, however, track additional information about the devices they host. As a result, it is rare for devices to be represented by bare device structures; instead, that structure, like kobject structures, is usually embedded within a higher-level representation of the device



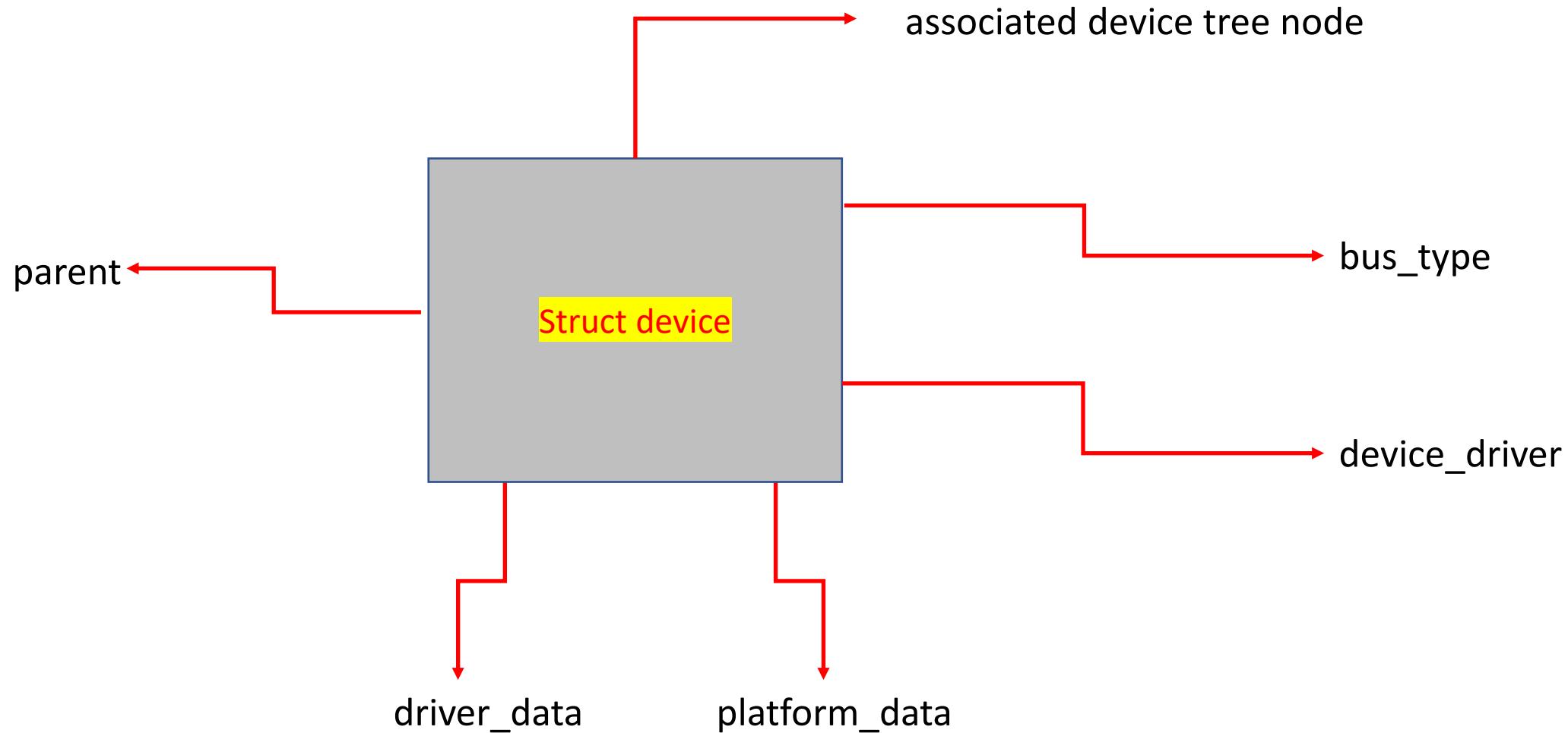
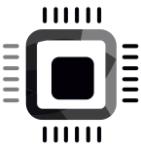
```
struct platform_device {  
    const char *name;  
    int id;  
    bool id_auto;  
    struct device dev;  
    u64 platform_dma_mask;  
    u32 num_resources;  
    struct resource *resource;  
  
    const struct platform_device_id *id_entry;  
    char *driver_override;  
  
    /* MFD cell pointer */  
    struct mfd_cell *mfd_cell;  
  
    /* arch specific additions */  
    struct pdev_archdata archdata;  
};
```

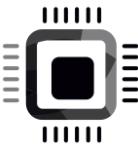
Struct platform_device

Subsystem specific
information

struct device

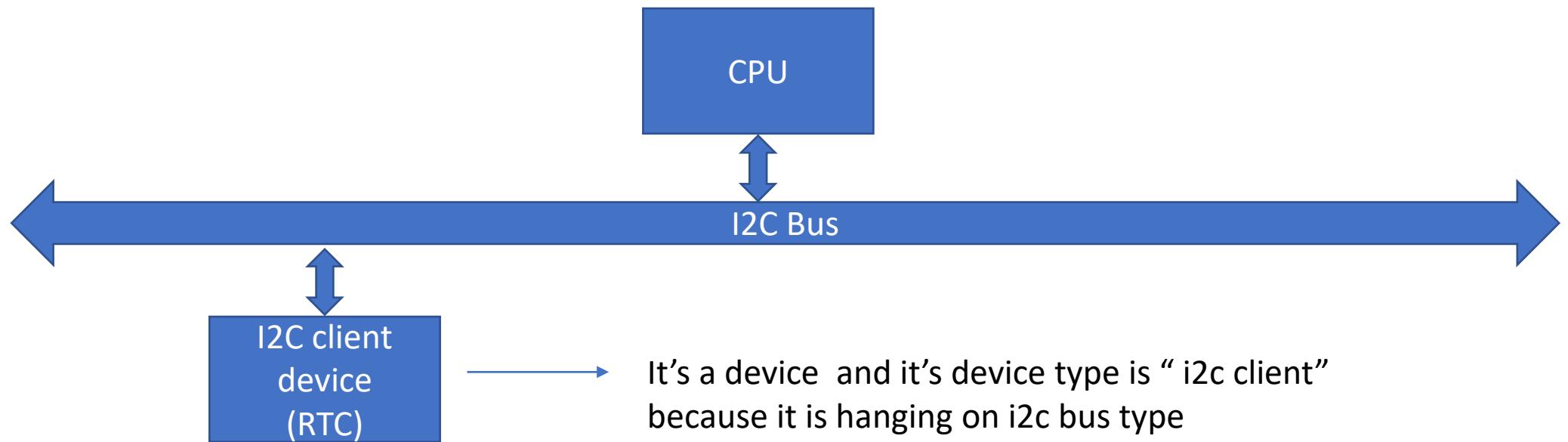
Device specific
information

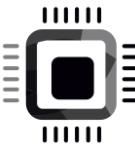




Example

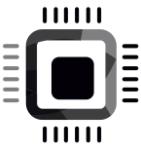
Consider the case of a platform device .





```
struct i2c_client {  
    /* div., see below      */  
    unsigned short flags;  
    /* Use Packet Error Checking */  
    #define I2C_CLIENT_PEC      0x04  
    /* we have a ten bit chip address */  
    /* Must equal I2C_M_TEN below */  
    #define I2C_CLIENT_TEN      0x10  
    /* we are the slave */  
    #define I2C_CLIENT_SLAVE     0x20  
    /* We want to use I2C host notify */  
    #define I2C_CLIENT_HOST_NOTIFY 0x40  
    /* chip address - NOTE: 7bit      */  
    /* addresses are stored in the   */  
    /* _LOWER_ 7 bits             */  
    unsigned short addr;  
    char name[I2C_NAME_SIZE];  
    /* the adapter we sit on      */  
    struct i2c_adapter *adapter;  
    /* irq set at initialization */  
    int init_irq;  
    /* irq issued by device      */  
    int irq;  
    /* the device structure      */  
    struct device dev;  
};
```

An `i2c_client` identifies a single device (i.e. chip) connected to an i2c bus.
The behaviour exposed to Linux is defined by the driver managing the device.

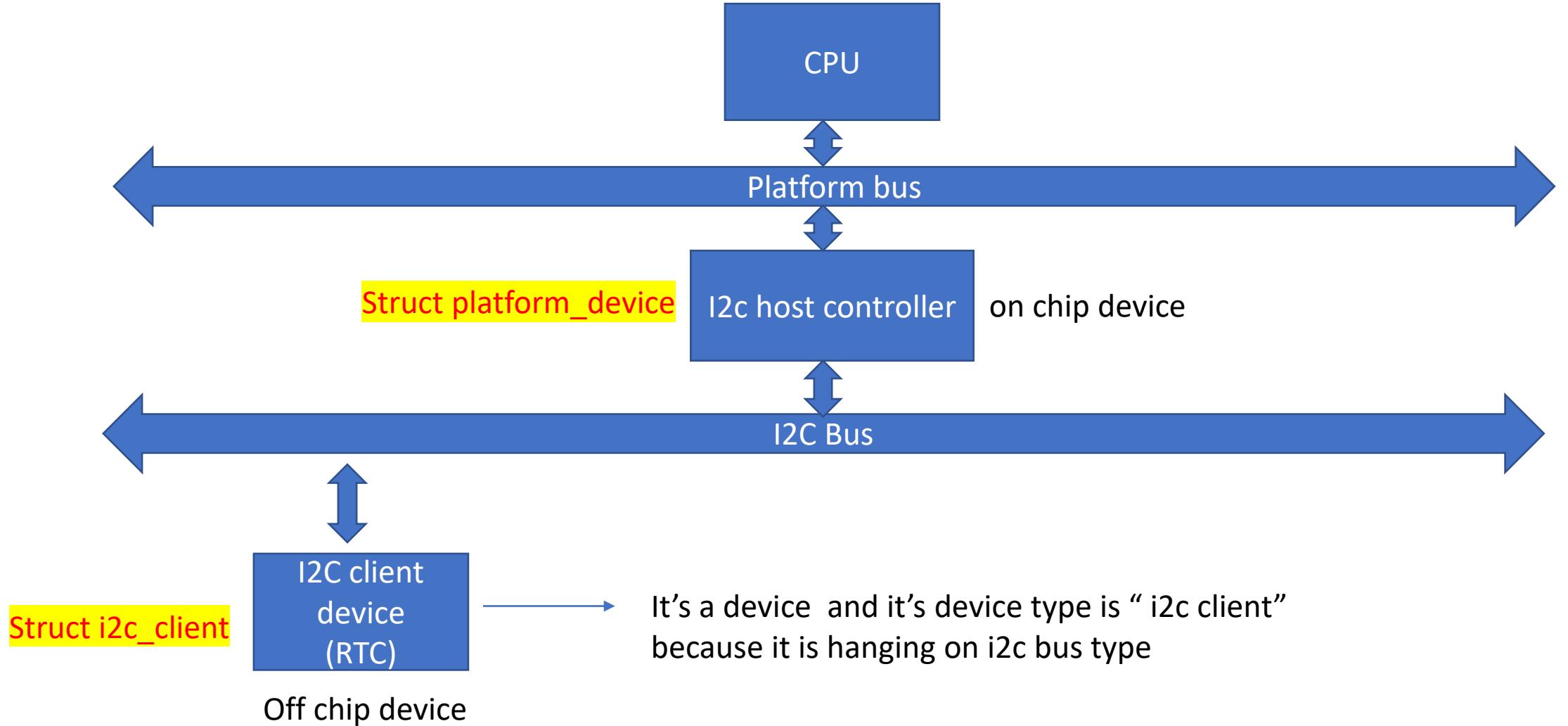
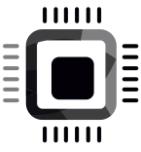


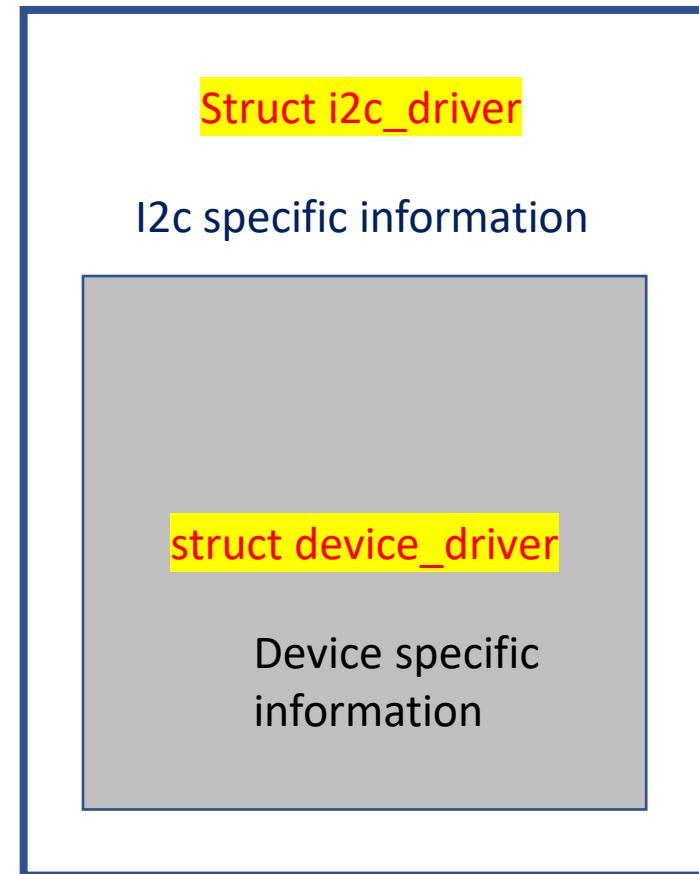
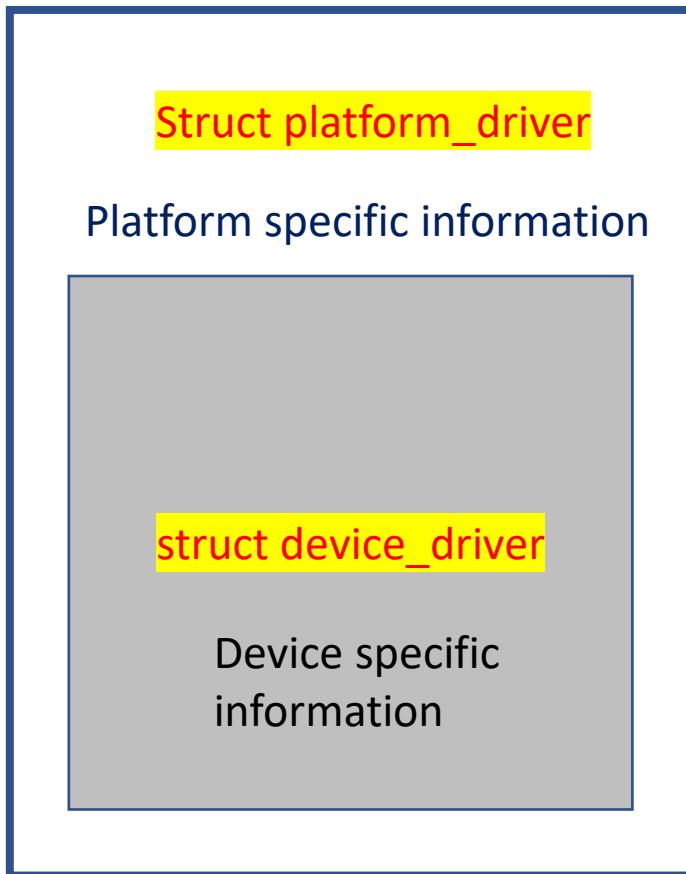
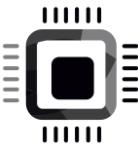
Struct i2c_client

I2c specific information

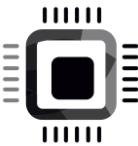
struct device

Device specific
information





Each driver of the system is represented in the Linux device driver model as an object of **struct device_driver**



```
struct device_driver {
    const char      *name;
    struct bus_type *bus;

    struct module     *owner;
    const char      *mod_name; /* used for built-in modules */

    bool suppress_bind_attrs; /* disables bind/unbind via sysfs */
    enum probe_type probe_type;

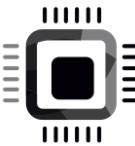
    const struct of_device_id  *of_match_table;
    const struct acpi_device_id *acpi_match_table;

    int (*probe) (struct device *dev);
    int (*remove) (struct device *dev);
    void (*shutdown) (struct device *dev);
    int (*suspend) (struct device *dev, pm_message_t state);
    int (*resume) (struct device *dev);
    const struct attribute_group **groups;
    const struct attribute_group **dev_groups;

    const struct dev_pm_ops *pm;
    void (*coredump) (struct device *dev);

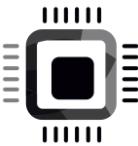
    struct driver_private *p;
};
```

The shutdown, suspend, and resume methods are invoked on a device when the kernel must change its power state.



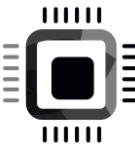
Kobject

- Kobject stands for kernel object which is represented by `struct kobject`
- Kobjects are a fundamental building block of Linux device and driver hierarchy
- Kobjects are used to represent the ‘containers’ in the sysfs virtual filesystem
- Kobjects are also used for reference counting of the ‘containers’.
- It has got its name, type, and parent pointer to weave the Linux device and driver hierarchy
- Using kobjects you can add attributes to the container, which can be viewed/ altered by the user space.
- The sysfs filesystem gets populated because of kobjects, sysfs is a user space representation of the kobjects hierarchy inside the kernel



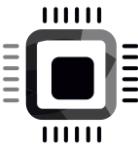
```
struct kobject {
    const char          *name;
    struct list_head    entry;
    struct kobject      *parent;
    struct kset         *kset;
    struct kobj_type   *ktype;
    struct kernfs_node *sd; /* sysfs directory entry */
    struct kref         kref;
#ifndef CONFIG_DEBUG_KOBJECT_RELEASE
    struct delayed_work release;
#endif
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

Include/kobject.h



What are containers ?

- Kobjects are rarely or never used as stand-alone kernel objects. Most of the time, they are embedded in some other structure that we call container structure, which describes the device diver model's components.
- Some example of container structure could be **struct bus**, **struct device**, **struct device_driver**
- The container structure's embedded kobject enables the container to become part of an object hierarchy.

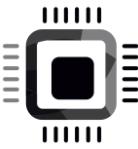


```
struct kobject {
    const char *name;
    struct list_head entry;
    struct kobject *parent;
    struct kset *kset;
    struct kobj_type *ktype;
    /* sysfs directory entry */
    struct kernfs_node*sd;
    struct kref kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```



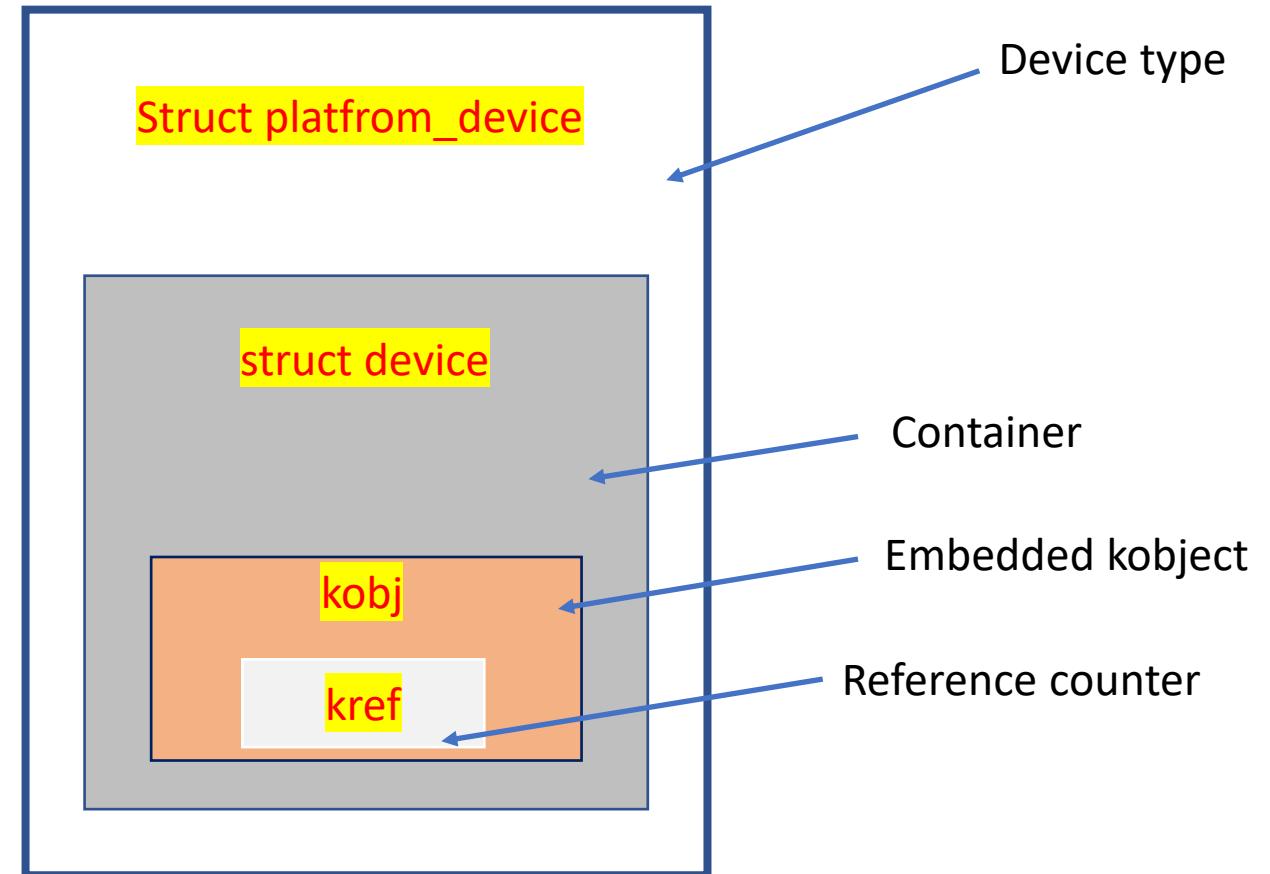
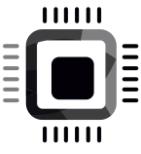
```
struct kref {
    refcount_t refcount;
};
```

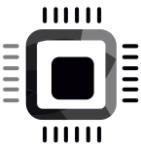
Kobject reference counter



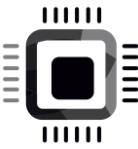
```
struct kobject {  
    const char *name;  
    struct list_head entry;  
    struct kobject *parent;  
    struct kset *kset;  
    struct kobj_type *ktype; _____  
/* sysfs directory entry */  
    struct kernfs_node*sd;  
    struct kref kref;  
    unsigned int state_initialized:1;  
    unsigned int state_in_sysfs:1;  
    unsigned int state_add_uevent_sent:1;  
    unsigned int state_remove_uevent_sent:1;  
    unsigned int uevent_suppress:1;  
};
```

```
struct kobj_type {  
    void (*release)(struct kobject *kobj);  
    const struct sysfs_ops *sysfs_ops;  
    struct attribute **defaultAttrs;  
    const struct attribute_group **defaultGroups;  
    .....  
};
```



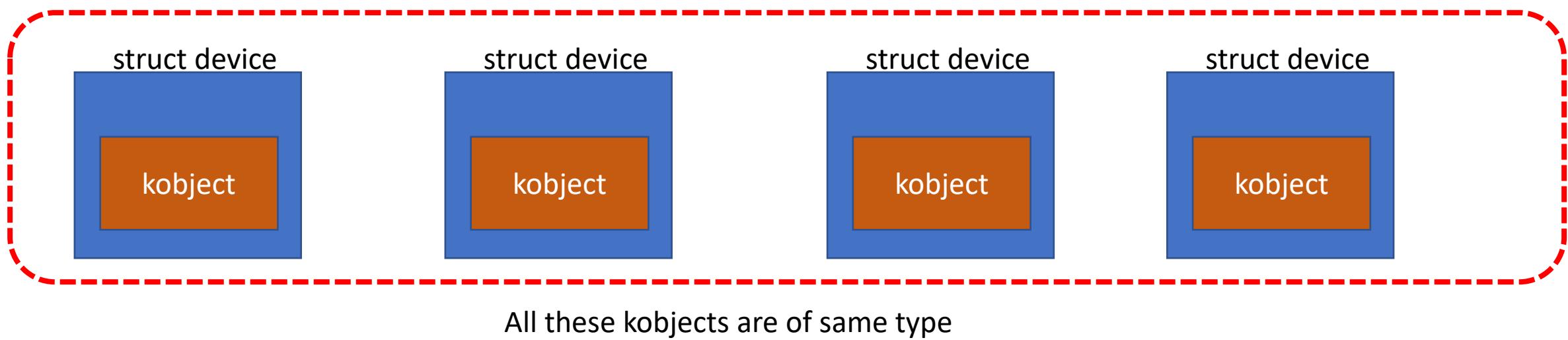


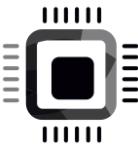
Kobject type and kset



Kobject type

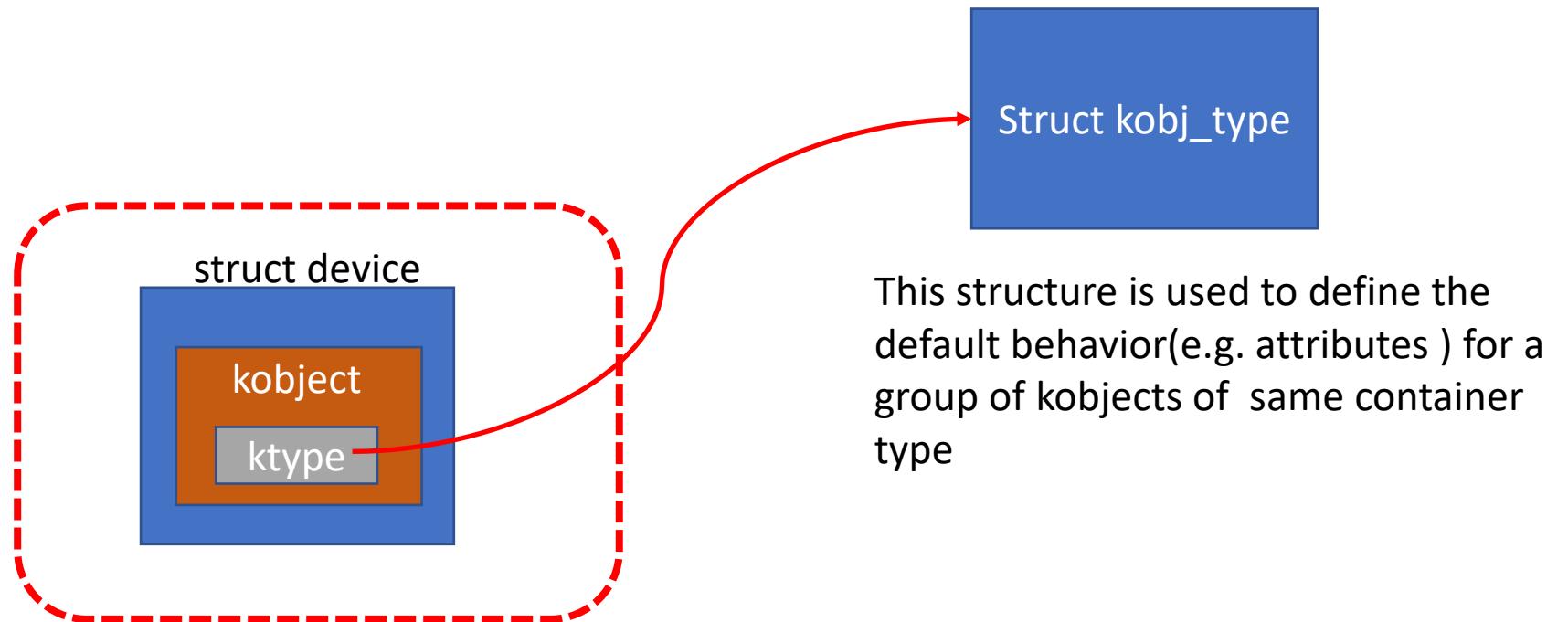
- Type of a kobject is determined based on, type of the container in which the kobject is embedded

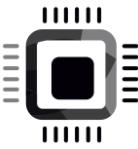




struct kobj_type (ktype)

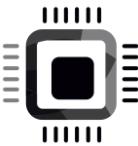
Now, the type of a kobject is controlled using the structure called **struct kobj_type**



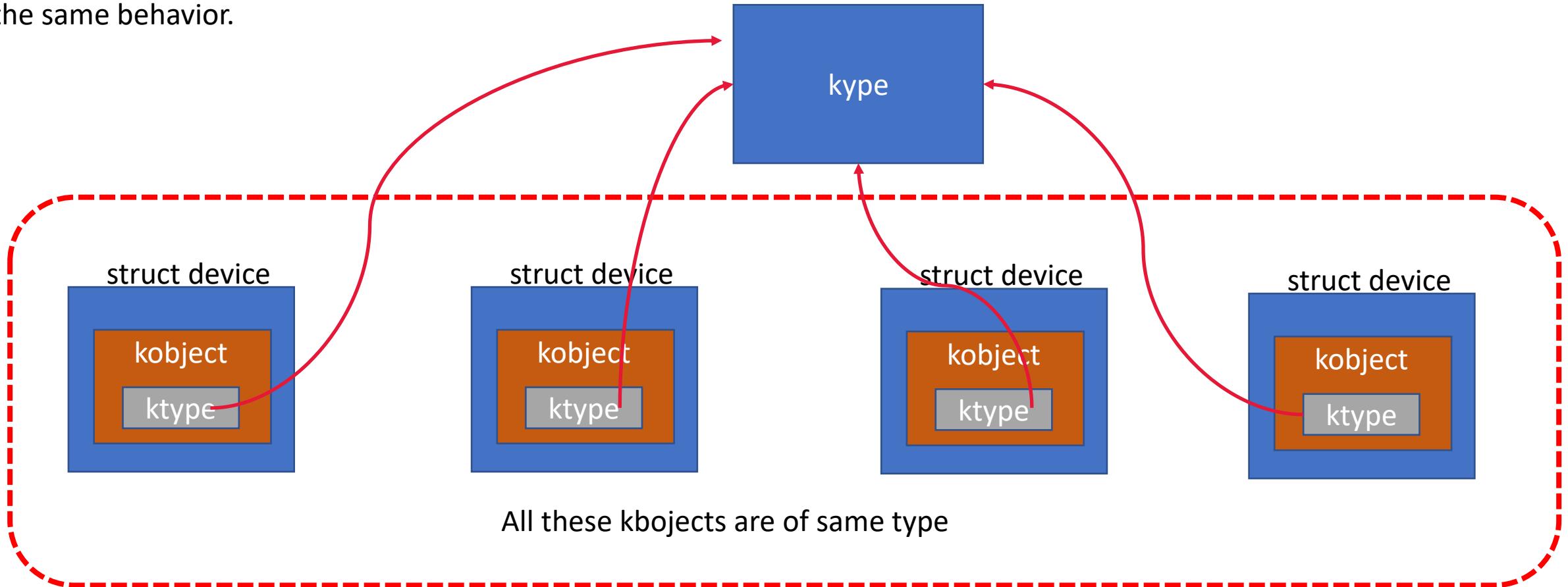


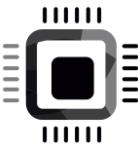
Struct kobj_type

- struct kobj_type object or simply ktype object is an object which defines the behavior for the container object.
- Behaviors are manifested in terms of attributes and file operation methods that handle those attributes.
- The ktype also controls what happens to the kobject when it is created and destroyed.
- Every structure that embeds a kobject needs a corresponding ktype.



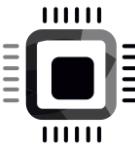
Instead of each kobject defining its own behavior, the behavior is stored in a ktype, and kobjects of the same “type” point at the same ktype structure, thus sharing the same behavior.





```
struct kobject {
    const char *name;
    struct list_head entry;
    struct kobject *parent;
    struct kset *kset;
    struct kobj_type *ktype; /* sysfs directory entry */
    struct kernfs_node*sd;
    struct kref kref;
    unsigned int state_initialized:1;
    unsigned int state_in_sysfs:1;
    unsigned int state_add_uevent_sent:1;
    unsigned int state_remove_uevent_sent:1;
    unsigned int uevent_suppress:1;
};
```

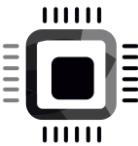
If you want to assign a type to your kobject , then you have to create an object of type 'struct kobj_type' and initialize the 'ktype' field of the kobject.



```
struct kobj_type {  
    void (*release)(struct kobject *kobj);  
    const struct sysfs_ops *sysfs_ops;  
    struct attribute **default_attrs;  
    const struct attribute_group **default_groups;  
    ....  
};
```

```
void my_object_release(struct kobject *kobj)  
{  
    struct my_object *mine = container_of(kobj, struct my_object, kobj);  
    /* Perform any additional cleanup on this object, then... */  
    kfree(mine);  
}
```

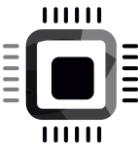
each kobject doesn't have its own release method.
release method provided by kobj_type object.



sysfs_ops points to sys operation structure which lists the methods to operate on the default attributes created for the kobject of this ktype

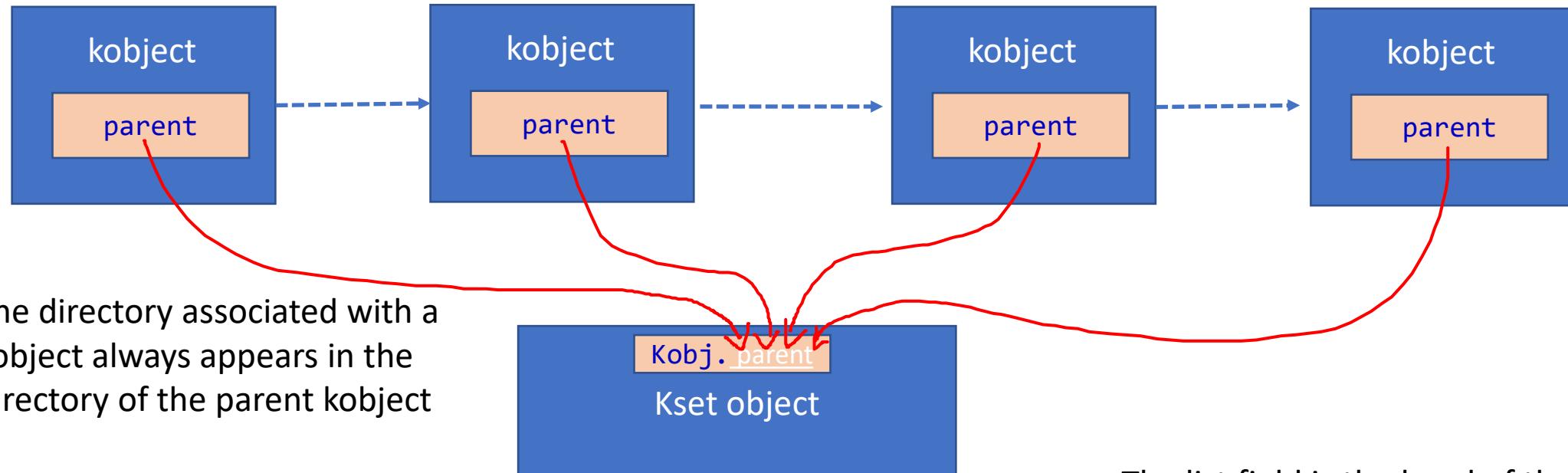
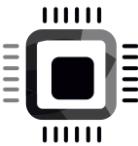
```
struct kobj_type {  
    void (*release)(struct kobject *kobj);  
    const struct sysfs_ops *sysfs_ops;  
    struct attribute **default_attrs;  
    const struct attribute_group **default_groups;  
    .....  
};
```

The default_attrs pointer is a list of default attributes that will be automatically created for any kobject that is registered with this ktype.



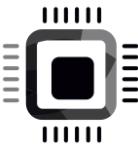
Kset

- Kset as its name indicates it's a set of kobjects of same type and belongs to a specific subsystem.
- Basically kset is a higher-level structure which 'collects' all lower level kobjects which belong to same type



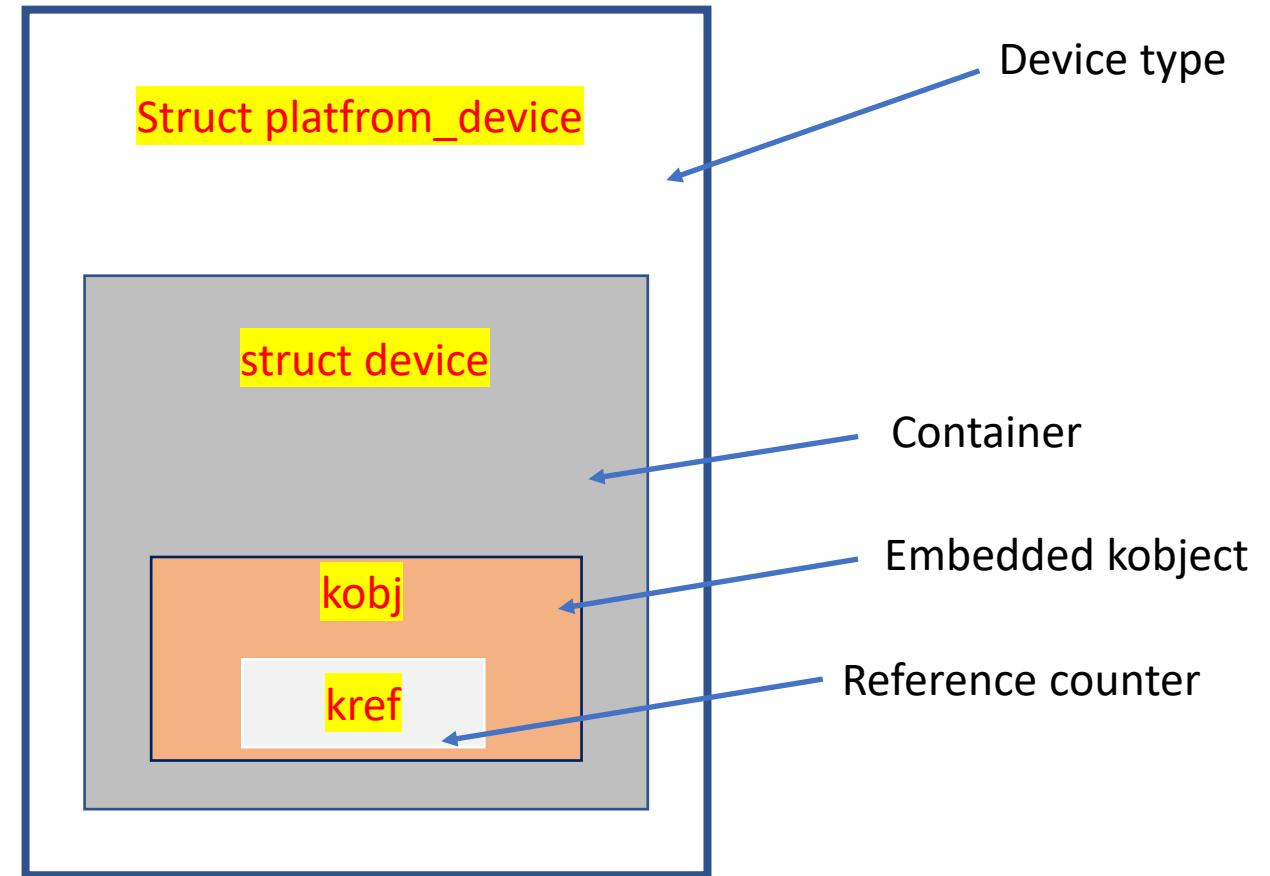
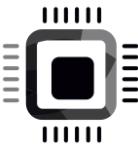
```
struct kset {  
    struct list_head list;  
    spinlock_t list_lock;  
    struct kobject kobj;  
    const struct kset_uevent_ops *uevent_ops;  
} __randomize_layout;
```

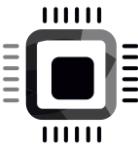
The list field is the head of the doubly linked circular list of `kobjects` included in the `kset`



Linux device driver model hierarchy

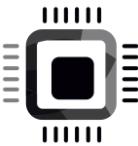
```
root@kiran-fastbiteba:/sys/devices/platform# ls
ACPI0003:00          intel_rapl_msgr.0    PNP0C14:00
ACPI000C:00          microcode           PNP0C14:01
coretemp.0            pcdev-A1x.0        power
eisa.0                pcdev-B1x.1        reg-dummy
'Fixed MDIO bus.0'   pcdev-C1x.2        regulatory.0
HPIC0003:00          pcdev-D1x.3        rtc_cmos
HPQ6001:00            pcspkr             serial8250
hp-wmi                PNP0C0A:00       uevent
i8042                 PNP0C0B:00       vesa-framebuffer.0
INT33A1:00            PNP0C0C:00
INT3400:00            PNP0C0D:00
```





Linux device driver model hierarchy

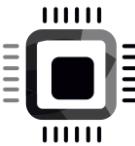
```
root@kiran-fastbiteba:/sys/devices/platform# ls
ACP[kobject]3:00          subsystem      kset
ACP[kobject]C:00          intel_ipl_msr.0  PNP[kobject]4:00
cor[kobject]p.0           microcode       PNP[kobject]4:01
eis[kobject]              pcdev-A1x.0    poh[kobject]
'Fixed MDIO bus.0'        pcdev-A1x.1    reg_dummy
[kobject]                 pcdev-A1x.2    regulatory.0
HPI[kobject]3:00          pcdev-A1x.3    rts_cros
HPQ[kobject]2:00          pcspn          ser218250
hp-wmi[kobject]           PNP0C0A.00   uevent
i804[kobject]             PNP0C0B.00   vesafb-framebuffer.0
INT33A1:00                PNP0C0C.00
INT3400:00                PNP0C0D.00
```



```
root@kiran-fastbiteba:/sys/devices/platform# ls
ACPI0003:00      HPQ6001:00      microcode      PNP0C0A:00      power      vesafb-framebuffer.0
ACPI000C:00      hp-wmi        pcdev-A1x.0    PNP0C0B:00      reg-dummy
coretemp.0        i8042         pcdev-B1x.1    PNP0C0C:00      regulatory.0
eisa.0            INT33A1:00    pcdev-C1x.2    PNP0C0D:00      rtc_cmos
'Fixed MDIO bus.0' INT3400:00    pcdev-D1x.3    PNP0C14:00      serial8250
HPIC0003:00      intel_rapl_msrm.0 pcspkr       PNP0C14:01      uevent

root@kiran-fastbiteba:/sys/devices/platform# cd pcdev-A1x.0/
root@kiran-fastbiteba:/sys/devices/platform/pcdev-A1x.0# ls -l
total 0
lrwxrwxrwx 1 root root    0 Jul 24 10:19 driver -> ../../bus/platform/drivers/pseudo-char-device
-rw-r--r-- 1 root root 4096 Jul 24 10:19 driver_override
-r--r--r-- 1 root root 4096 Jul 24 10:19 modalias
drwxr-xr-x 2 root root    0 Jul 24 10:19 power
lrwxrwxrwx 1 root root    0 Jul 24 10:19 subsystem -> ../../bus/platform
-rw-r--r-- 1 root root 4096 Jul 24 10:18 uevent
root@kiran-fastbiteba:/sys/devices/platform/pcdev-A1x.0# 
```

Default attributes of a kobject. Default attributes depend on the kobj_type



```
root@kiran-fastbiteba:/sys/bus/i2c/drivers# ls
88PM860x                               i2c_hid           sx150x-pinctrl
aat2870                                intel_soc_pmic_i2c tps65090
ab3100                                lp8788           tps6586x
adp5520                                max14577         tps65910
as3711                                max77693         tps65912
'CHT Whiskey Cove PMIC'                 max77843         tps68470
da903x                                 max8925           tps80031
da9052                                 max8997           twl
da9055-pmic                            max8998           twl6040
da9063                                 palmas          wm831x
dummy                                  rc5t583          wm8350
elants_i2c                            sec_pmic        WM8400
htcpld-chip                           smsc
```

root@kiran-fastbiteba: /sys/bus/i2c



File Edit View Search Terminal Help

root@kiran-fastbiteba:/sys/bus# ls

acpi	edac	isa	mmc	pci_express	serio	workqueue
cec	eisa	machinecheck	nd	platform	snd_seq	xen
clockevents	event_source	mdio_bus	node	pnp	spi	xen-backend
clocksource	gpio	media	nvmem	rapidio	usb	
container	hdaudio	mei	parport	scsi	virtio	
cpu	hid	memory	pci	sdio		
dax	i2c	mipi-dsi	pci-epf	serial		

Different bus_types
(bus subsystems)

root@kiran-fastbiteba:/sys/bus# cd i2c/

root@kiran-fastbiteba:/sys/bus/i2c# ls -l

total 0						
drwxr-xr-x	2	root root	0 Jul 24 10:08	devices		
drwxr-xr-x	40	root root	0 Jul 24 10:08	drivers		
-rw-r--r--	1	root root	4096 Jul 24 10:29	drivers_autoprobe		
--w-----	1	root root	4096 Jul 24 10:29	drivers_probe		
--w-----	1	root root	4096 Jul 24 10:08	uevent		

Device and drivers attached
to an i2c bus type

root@kiran-fastbiteba:/sys/bus/i2c#

root@kiran-fastbiteba: /sys/bus

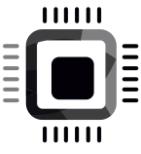


File Edit View Search Terminal Help

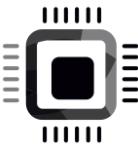
root@kiran-fastbiteba:/sys/bus# tree i2c | head -20

```
i2c bus_type (subsystem)
├── devices [kset]
│   ├── i2c-0 -> ../../../../devices/pci0000:00/0000:00:02.0/i2c-0
│   ├── i2c-1 -> ../../../../devices/pci0000:00/0000:00:02.0/i2c-1
│   ├── i2c-2 -> ../../../../devices/pci0000:00/0000:00:02.0/i2c-2
│   └── i2c-3 -> ../../../../devices/pci0000:00/0000:00:02.0/drm/card0/card0-eDP-1/i2c-3
└── drivers [kset]
    ├── 88PM860x [kobject]
    │   ├── bind Attribute
    │   ├── uevent Attribute
    │   └── unbind Attribute
    ├── aat2870 [kobject]
    │   └── uevent Attribute
    ├── ab3100 [kobject]
    │   └── uevent Attribute
    ├── adp5520 [kobject]
    │   └── uevent Attribute
    └── as3711 [kobject]
        ├── bind Attribute
        └── uevent Attribute
```

root@kiran-fastbiteba:/sys/bus#



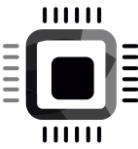
Sysfs and creating kobject attributes



What is sysfs ?

Sysfs is a virtual in-memory file system which provides

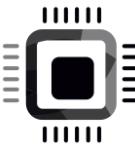
1. Representation of the kobject hierarchy of the kernel.
2. The complete topology of the devices and drivers of the system in terms of directories and attributes.
3. Attributes to help user space application to interact with devices and drivers
4. Standard methods to access devices using ‘classes’



Using sysfs

- sysfs is always compiled in if **CONFIG_SYSFS** is defined
- You can access it by doing **mount -t sysfs sysfs /sys**
- Documentation :

<https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt>



Creating kobject attributes of the device/driver (sysfs attributes)

What is a kobject attribute?

kobject attributes are regular files or symbolic names that appear in sysfs's kobject directory, and they are used to expose details about kobject's "container" to user-space.

In short kobject attributes are files through which device/driver data can be exposed to sysfs so that user application can view or modify.



File Edit View Search Terminal Help

root@kiran-fastbiteba: /sys/class/pcd_class/pcdev-0#

- - -

```
root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0# ls -l
total 0
-r--r--r-- 1 root root 4096 Jul 24 1Attribute5 dev
lrwxrwxrwx 1 root root    Jul 24 1Attribute5 device -> ../../../../../../pcdev-A1x.0
drwxr-xr-x 2 root root    0 Jul 24 1kobject5 power
lrwxrwxrwx 1 root root    0 Jul 24 1Attribute5 subsystem -> ../../../../../../class/pcd_class
-rw-r--r-- 1 root root 4096 Jul 24 1Attribute5 uevent
root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0#
```

Default attributes

kobject

Symbolic link to the parent device

Activities Terminal

Sun 12:31 •

Wi-Fi Battery

root@kiran-fastbiteba: /sys/class/pcd_class/pcdev-0/power



File Edit View Search Terminal Help

root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0/power# ls -l

total 0

```
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9  async
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 autosuspend_delay_ms
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 control
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_active_kids
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_active_time
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_enabled
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_status
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_suspended_time
-r--r--r-- 1 root root 4096 Jul 26 1Attribute9 runtime_usage
```

root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0/power# cat runtime_active_time

0

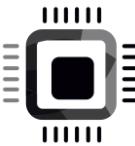
root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0/power# cat runtime_usage

0

root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0/power# cat runtime_status

unsupported

root@kiran-fastbiteba:/sys/class/pcd_class/pcdev-0/power#



Exercise : Creating custom attributes of a device

```
/*Device private data structure */
struct pcdev_private_data
{
    struct pcdev_platform_data pdata;
    char *buffer;
    dev_t dev_num;
    struct cdev cdev;
};

struct pcdev_platform_data
{
    int size;
    int perm;
    const char *serial_number;
};
```

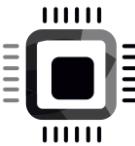
Create 2 sysfs attributes :

1) **max_size** (world-read/owner-write)

Through this attribute user reads or changes the value of 'size' information

2) **serial_num** (world-read)

Through this attribute user just reads the serial number of the device

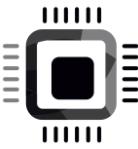


Creating sysfs attributes

- sysfs attributes are represented by the below structure

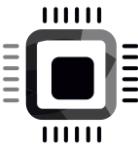
```
struct attribute
{
    const char *name;           ← Name of the attribute. This name will show in the sysfs
                                kobject directory as attribute's name
    umode_t mode;              ← This controls the read/write permission for the
                                attribute file from user space programs
    #ifdef CONFIG_DEBUG_LOCK_ALLOC
        bool ignore_lockdep:1;
        struct lock_class_key *key;
        struct lock_class_keys key;
    #endif
};
```

Include/linux/sysfs.h



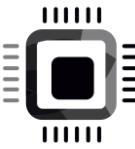
Creating 2 device attributes

- 1) Create **max_size** attribute with mode ‘world-read and only owner change’ (S_IRUGO|S_IWUSR)
- 2) Create **serial_number** attribute with mode ‘world-read-only’ (S_IRUGO)



Mode of an attribute

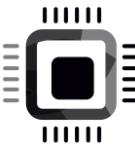
- S_IRUGO – world-read-only
- S_IRUSR – owner read-only
- S_IRUGO | S_IWUSR : world-read and only owner write



APIs for managing sysfs files (Attributes)

```
int sysfs_create_file(struct kobject *kobj,const struct attribute *attr);  
  
void sysfs_remove_file(struct kobject *kobj,const struct attribute *attr);  
  
int sysfs_create_groups(struct kobject *kobj,const struct attribute_group **groups);  
  
int sysfs_create_group(struct kobject *kobj,const struct attribute_group *grp);  
  
int sysfs_chmod_file(struct kobject *kobj,const struct attribute *attr, umode_t mode);
```

include/linux/sysfs.h

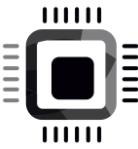


file I/O operations on sysfs attributes

- Once you create a sysfs files (attributes), you should provide read and write methods for them so that user can read value of the attribute or write a new value to the attribute .
- If you see **struct attribute** there is no place to hook read/write methods for an attribute. Basically it just represents name of a attribute and the mode

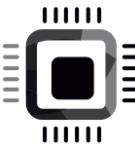
```
/* interface for exporting device attributes */  
struct device_attribute  
{  
    struct attribute attr;  
    ssize_t (*show)(struct device *dev, struct device_attribute *attr, char *buf);  
    ssize_t (*store)(struct device *dev, struct device_attribute *attr,  
                     const char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

Use this structure if your goal is to create attributes for a device and to provide show/store methods



```
/* interface for exporting device attributes */
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);
};
```

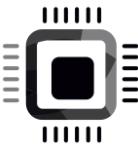
include/linux/device.h



Creating device_attribute variables

- Instead of manually creating variables of *struct device_attribute* and initializing them, use the DEVICE_ATTR_XX macros given in include/linux/device.h

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
    struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
#define DEVICE_ATTR_PREALLOC(_name, _mode, _show, _store) \
    struct device_attribute dev_attr_##_name = \
        __ATTR_PREALLOC(_name, _mode, _show, _store)
#define DEVICE_ATTR_RW(_name) \
    struct device_attribute dev_attr_##_name = __ATTR_RW(_name)
#define DEVICE_ATTR_RO(_name) \
    struct device_attribute dev_attr_##_name = __ATTR_RO(_name)
#define DEVICE_ATTR_WO(_name) \
    struct device_attribute dev_attr_##_name = __ATTR_WO(_name)
```

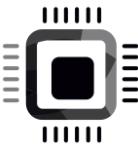


Usage

```
static struct device_attribute dev_attr_bar = {  
    .attr = {  
        .name = "bar",  
        .mode = S_IWUSR | S_IRUGO,  
    },  
    .show = show_bar,  
    .store = store_bar,  
};
```

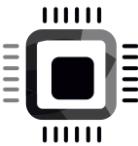
is equivalent to doing:

```
static DEVICE_ATTR(bar, S_IWUSR | S_IRUGO, show_bar, store_bar);
```



Show and store methods

- Show method is used to export attribute value to the user space
- Store method is used to receive a new value from the user space for an attribute



Show method

Prototype:

```
ssize_t (*show)(struct device *dev, struct device_attribute *attr , char *buf);
```

This function is invoked when user space program tries to read the value of the attribute.

Here you must copy the value of the attribute in to the ‘buf’ pointer.

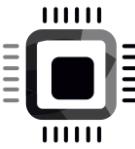
Note that ‘buf’ is not user level pointer .

Its provided by the kernel buffer so its size is limited to PAGE_SIZE .

For ARM architecture it is 4KB (4096 Bytes) long

So , while copying data to ‘buf’ pointer the size shouldn’t exceed 4096 bytes.

To copy data in to ‘buf’ you can either use sprintf() or scnprintf()



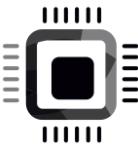
Show method

Difference between drivers read method and drivers show method

- 1) Read method is used by the user space to read large amounts of data from the driver .
- 2) Show method is used for reading a single value data or an array of similar values or data whose length is less than PAGE_SIZE
- 3) Use show method to read any configuration data of your driver or device.

Return value of show method :

show() methods should return the number of bytes copied into the buffer or an error code



Store method

```
ssize_t (*store)(struct device *dev, struct device_attribute *attr, \
                  const char *buf, size_t count);
```

This is invoked when user wants to modify the value of the sysfs file .

In this method , 'buf' points to the user data.

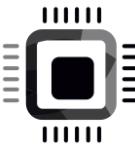
'count' parameter is the amount of user data being passed in.

The maximum amount of data which the 'buf' pointer can carry is limited PAGE_SIZE

The data carried by the 'buf' is NULL terminated .

store() should return the number of bytes used from the buffer.

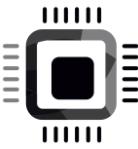
If the entire buffer has been used, just return the count argument.



```
/**  
 * kstrtol - convert a string to a long  
 * @s: The start of the string. The string must be null-terminated, and may also  
 * include a single newline before its terminating null. The first character  
 * may also be a plus sign or a minus sign.  
 * @base: The number base to use. The maximum supported base is 16. If base is  
 * given as 0, then the base of the string is automatically detected with the  
 * conventional semantics - If it begins with 0x the number will be parsed as a  
 * hexadecimal (case insensitive), if it otherwise begins with 0, it will be  
 * parsed as an octal number. Otherwise it will be parsed as a decimal.  
 * @res: Where to write the result of the conversion on success.  
 *  
 * Returns 0 on success, -ERANGE on overflow and -EINVAL on parsing error.  
 * Used as a replacement for the obsolete simple strtoull. Return code must  
 * be checked.  
 */
```

int kstrtol(const char *s, unsigned int base, long *res)

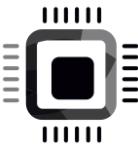
include/linux/kernel.h



Attribute grouping

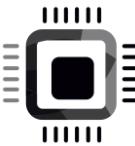
- Instead of calling `sysfs_create_file` to create a sysfs file for every attribute. You can finish it off in one call using attribute grouping

```
int sysfs_create_group(struct kobject *kobj,const struct attribute_group *grp);
```

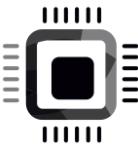


Attribute grouping

```
int sysfs_create_group(struct kobject *kobj,\n                      const struct attribute_group *grp);
```



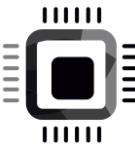
```
/**  
 * struct attribute_group - data structure used to declare an attribute group.  
 * @name: Optional: Attribute group name  
 * If specified, the attribute group will be created in  
 * a new subdirectory with this name.  
 * @is_visible: Optional: Function to return permissions associated with an  
 * attribute of the group. Will be called repeatedly for each  
 * non-binary attribute in the group. Only read/write  
 * permissions as well as SYSFS_PREALLOC are accepted. Must  
 * return 0 if an attribute is not visible. The returned value  
 * will replace static permissions defined in struct attribute.  
 * @is_bin_visible:  
 * Optional: Function to return permissions associated with a  
 * binary attribute of the group. Will be called repeatedly  
 * for each binary attribute in the group. Only read/write  
 * permissions as well as SYSFS_PREALLOC are accepted. Must  
 * return 0 if a binary attribute is not visible. The returned  
 * value will replace static permissions defined in  
 * struct bin_attribute.  
 * @attrs: Pointer to NULL terminated list of attributes.  
 * @bin_attrs: Pointer to NULL terminated list of binary attributes.  
 * Either attrs or bin_attrs or both must be provided.  
 */
```



```
struct attribute_group {  
    const char          *name;  
    umode_t             (*is_visible)(struct kobject *,  
                                     struct attribute *, int);  
    umode_t             (*is_bin_visible)(struct kobject *,  
                                         struct bin_attribute *, int);  
    struct attribute   **attrs;←  
    struct bin_attribute **bin_attrs;]  
};
```

Pointer to NULL terminated list of
attributes

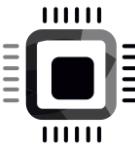
Include/linux/sysfs.h



```
/*
 * sysfs_streq - return true if strings are equal, modulo trailing newline
 * @s1: one string
 * @s2: another string
 *
 * This routine returns true iff two strings are equal, treating both
 * NUL and newline-then-NUL as equivalent string terminations. It's
 * geared for use with sysfs input strings, which generally terminate
 * with newlines but are compared against values without newlines.
 */
bool sysfs_streq(const char *s1, const char *s2)
{
    while (*s1 && *s1 == *s2) {
        s1++;
        s2++;
    }

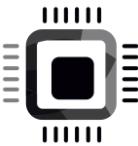
    if (*s1 == *s2)
        return true;
    if (!*s1 && *s2 == '\n' && !s2[1])
        return true;
    if (*s1 == '\n' && !s1[1] && !*s2)
        return true;
    return false;
}
```

lib/string.c



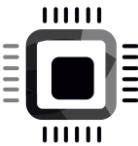
APIs for managing devices inside a class

- `int device_create_file(struct device *, const struct device_attribute *);`
- `void device_remove_file(struct device *, const struct device_attribute *);`



sysfs interface for exporting driver attributes

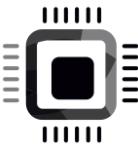
```
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *driver, char *buf);
    ssize_t (*store)(struct device_driver *driver, const char *buf,
                     size_t count);
};
```



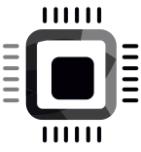
```
#define DRIVER_ATTR_RW(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RW(_name)

#define DRIVER_ATTR_RO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_RO(_name)

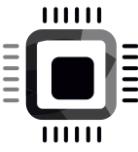
#define DRIVER_ATTR_WO(_name) \
    struct driver_attribute driver_attr_##_name = __ATTR_WO(_name)
```



```
int driver_create_file(struct device_driver *driver, \
                      const struct driver_attribute *attr);  
  
void driver_remove_file(struct device_driver *driver, \
                        const struct driver_attribute *attr);
```

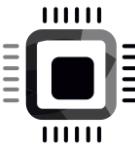


```
int device_create_file(struct device *device,\n                      const struct device_attribute *entry);\nvoid device_remove_file(struct device *dev,\n                      const struct device_attribute *attr);
```



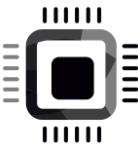
Device unregister

```
void device_unregister(struct device *dev)
```



Helper macros for defining device attribute structures

- `#define DEVICE_ATTR(_name, _mode, _show, _store) \ struct
device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show,
_store)`

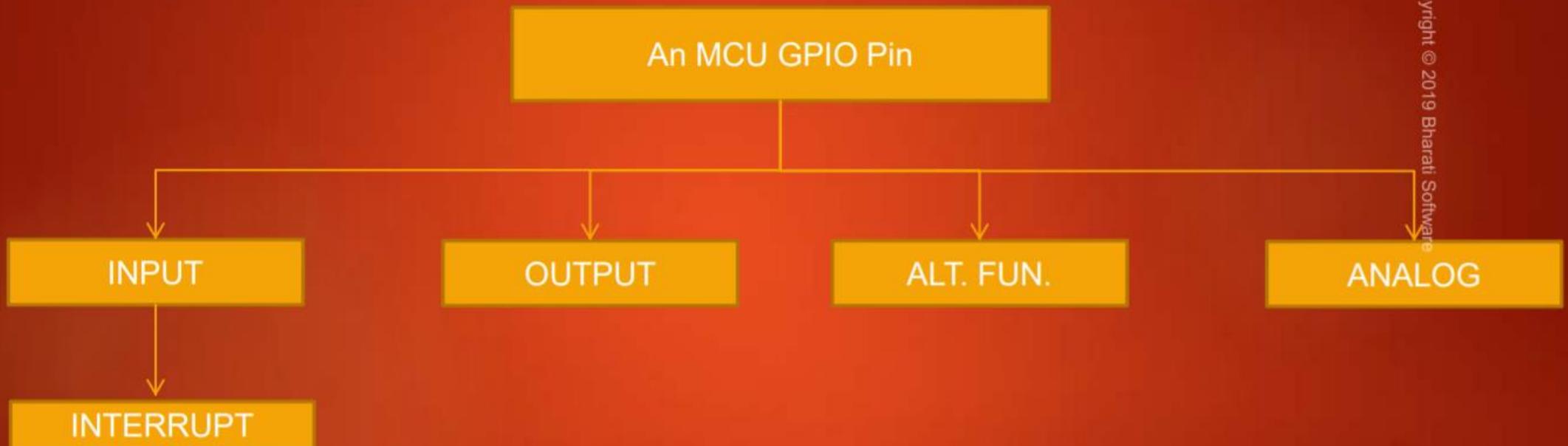


What's an GPIO ?

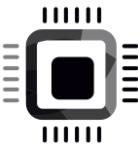
A GPIO Pin can be used for many purposes as shown here . That's why it is called as “General” Purpose.

Some pins of the MCU can not be used for all these purposes. So those are called as just pins but not GPIOs

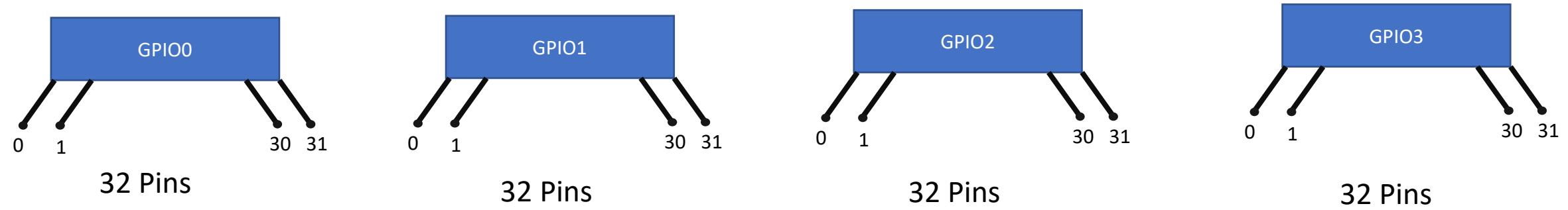
Copyright © 2019 Bharati Software



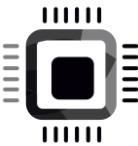
When MCU pin is in INPUT mode
it can be configured to issue an
interrupt to the processor



AM335x GPIO controllers



The AM335x soc gives in total $32 * 4 = 128$ GPIO pins



25.1 Introduction

25.1.1 Purpose of the Peripheral

The general-purpose interface combines four general-purpose input/output (GPIO) modules. Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities; thus, the general-purpose interface supports up to 128 (4×32) pins. These pins can be configured for the following applications:

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell
- Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.

25.1.2 GPIO Features

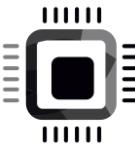
Each GPIO module is made up of 32 identical channels. Each channel can be configured to be used in the following applications:

- Data input/output
- Keyboard interface with a de-bouncing cell
- Synchronous interrupt generation (in active mode) upon the detection of external events (signal transition(s) and/or signal level(s))
- Wake-up request generation (in Idle mode) upon the detection of signal transition(s)

Global features of the GPIO interface are:

- Synchronous interrupt requests from each channel are processed by two identical interrupt generation sub-modules to be used independently by the ARM Subsystem
- Wake-up requests from input channels are merged together to issue one wake-up signal to the system
- Shared registers can be accessed through "Set & Clear" protocols

AM335x GPIO
controllers

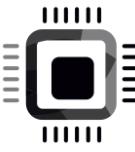


Memory map

Region Name	Start Address (hex)	End Address (hex)	Size	Description
GPIO0	0x44E0_7000	0x44E0_7FFF	4KB	GPIO0 Registers
GPIO1	0x4804_C000	0x4804_CFFF	4KB	GPIO1 Registers
GPIO2	0x481A_C000	0x481A_CFFF	4KB	GPIO2 Registers
GPIO3	0x481A_E000	0x481A_EFFF	4KB	GPIO3 Registers

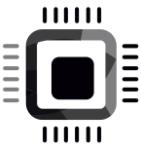
GPIOx register addresses

GPIO pins are configured through memory-mapped GPIO configuration registers

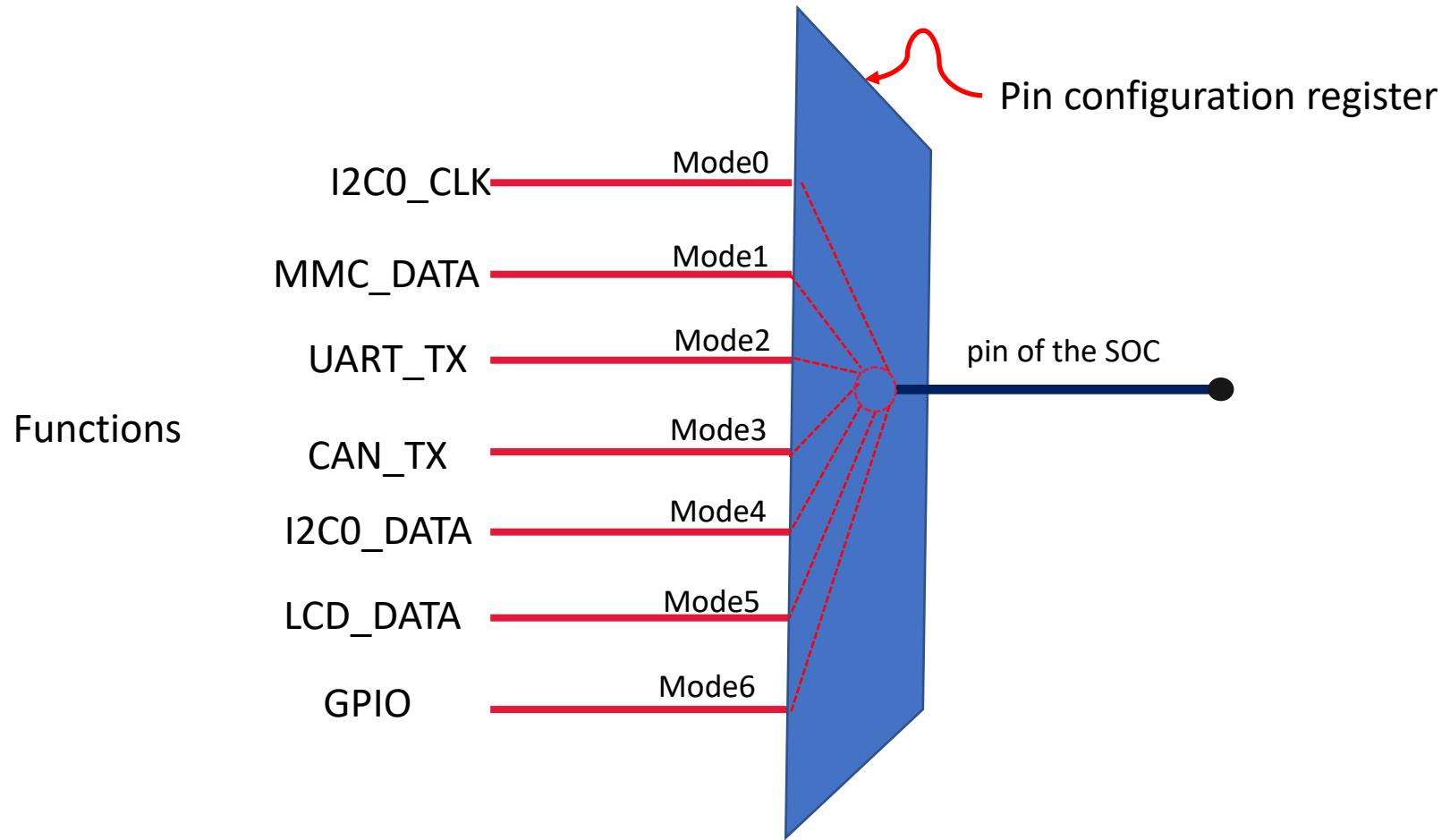


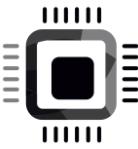
AM335x pin modes

- In a microcontroller/SOC, most of the pins have multiple functionalities. Also called modes.
- For example, a pin of the SOC can act as a simple GPIO, or it can act as UART_TX line, or UART_RX line or MMC_DATA line, etc. This is called pin multiplexing.
- Pin multiplexing helps SOC designers to produce an SOC with lesser pin counts
- When a Pin is used for one functionality, it cannot be used for another functionality unless you reconfigure it . (mode setting or pad configuration)



Pin multiplexing

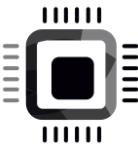




Accessing GPIOs on BBB header

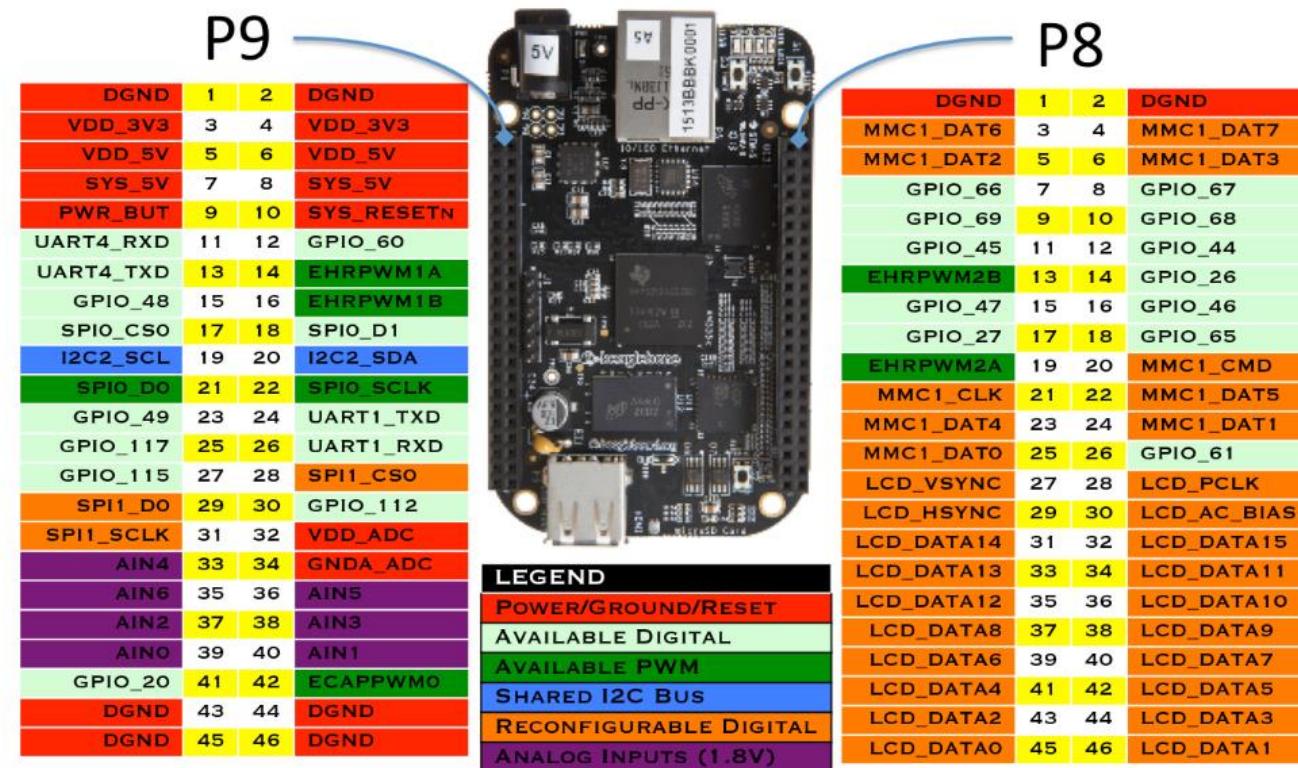
- <https://github.com/beagleboard/beaglebone-black/wiki/System-Reference-Manual#section-7-1>

7.1 Expansion Connectors

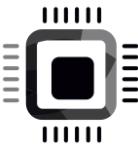


BBB expansion headers

Cape Expansion Headers



Reference : <https://beagleboard.org/Support/bone101>



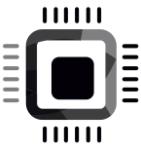
65 possible digital I/Os

P9

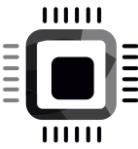
DGND	1	2	DGND
VDD_3V3	3	4	VDD_3V3
VDD_5V	5	6	VDD_5V
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESETN
GPIO_30	11	12	GPIO_60
GPIO_31	13	14	GPIO_50
GPIO_48	15	16	GPIO_51
GPIO_5	17	18	GPIO_4
I2C2_SCL	19	20	I2C2_SDA
GPIO_3	21	22	GPIO_2
GPIO_49	23	24	GPIO_15
GPIO_117	25	26	GPIO_14
GPIO_115	27	28	GPIO_113
GPIO_111	29	30	GPIO_112
GPIO_110	31	32	VDD_ADC
AIN4	33	34	GNDA_ADC
AIN6	35	36	AIN5
AIN2	37	38	AIN3
AIN0	39	40	AIN1
GPIO_20	41	42	GPIO_7
DGND	43	44	DGND
DGND	45	46	DGND

P8

DGND	1	2	DGND
GPIO_38	3	4	GPIO_39
GPIO_34	5	6	GPIO_35
GPIO_66	7	8	GPIO_67
GPIO_69	9	10	GPIO_68
GPIO_45	11	12	GPIO_44
GPIO_23	13	14	GPIO_26
GPIO_47	15	16	GPIO_46
GPIO_27	17	18	GPIO_65
GPIO_22	19	20	GPIO_63
GPIO_62	21	22	GPIO_37
GPIO_36	23	24	GPIO_33
GPIO_32	25	26	GPIO_61
GPIO_86	27	28	GPIO_88
GPIO_87	29	30	GPIO_89
GPIO_10	31	32	GPIO_11
GPIO_9	33	34	GPIO_81
GPIO_8	35	36	GPIO_80
GPIO_78	37	38	GPIO_79
GPIO_76	39	40	GPIO_77
GPIO_74	41	42	GPIO_75
GPIO_72	43	44	GPIO_73
GPIO_70	45	46	GPIO_71



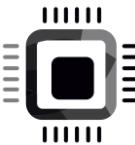
Pad configuration register



Pin count of AM335x



Pin Package type : ZCZ
Total pin count = 324



9.2.2 Pad Control Registers

The Pad Control Registers are 32-bit registers to control the signal muxing and other aspects of each I/O pad. After POR, software must set the pad functional multiplexing and configuration registers to the desired values according to the requested device configuration. The configuration is controlled by pads or by a group of pads. Each configurable pin has its own configuration register for pullup/down control and for the assignment to a given module.

The following table shows the generic Pad Control Register Description.

Table 9-1. Pad Control Register Field Descriptions

Bit	Field	Value	Description
31-7	Reserved		Reserved. Read returns 0.
6	SLEWCTRL	0 1	Select between faster or slower slew rate. Fast Slow ⁽¹⁾
5	RXACTIVE	0 1	Input enable value for the Pad. Set to 0 for output only. Set to 1 for input or output. Receiver disabled Receiver enabled
4	PULLTYPESEL	0 1	Pad pullup/pulldown type selection Pulldown selected Pullup selected
3	PULLUDEN	0 1	Pad Pullup/pulldown enable Pullup/pulldown enabled. Pullup/pulldown disabled.
2-0	MUXMODE		Pad functional signal mux select

⁽¹⁾ Some peripherals do not support slow slew rate. To determine which interfaces support each slew rate, see *AM335x ARM Cortex-A8 Microprocessors (MPUs)* (literature number [SPRS717](#)).

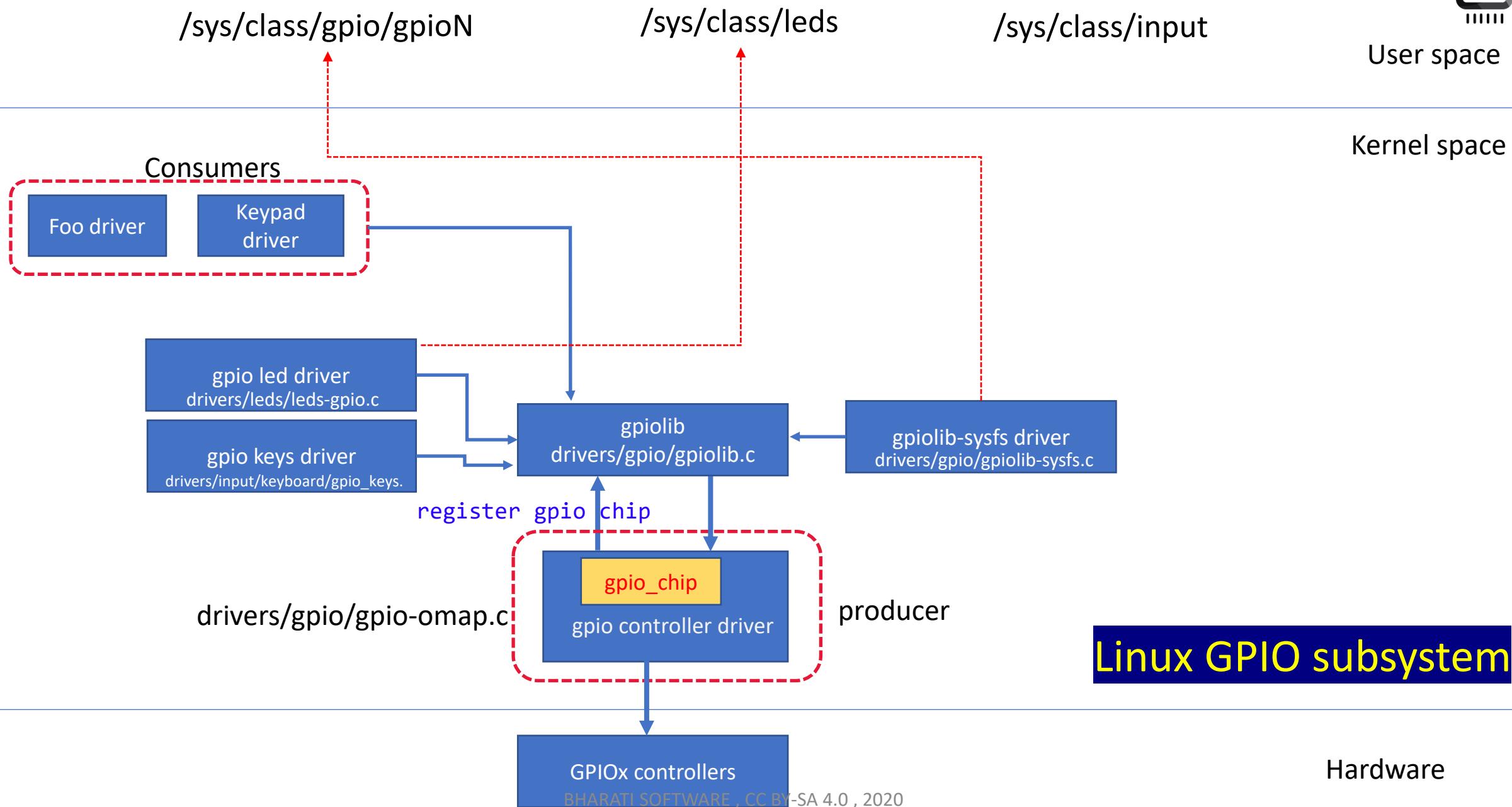
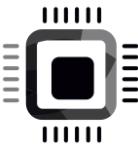
9.2.2.1 Mode Selection

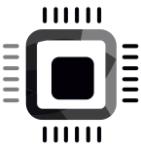
The MUXMODE field in the pad control registers defines the multiplexing mode applied to the pad. Modes are referred to by their decimal (from 0 to 7) or binary (from 0b000 to 0b111) representation. For most pads, the reset value for the MUXMODE field in the registers is 0b111. The exceptions are pads to be used at boot time to transfer data from selected peripherals to the external flash memory.

Table 9-2. Mode Selection

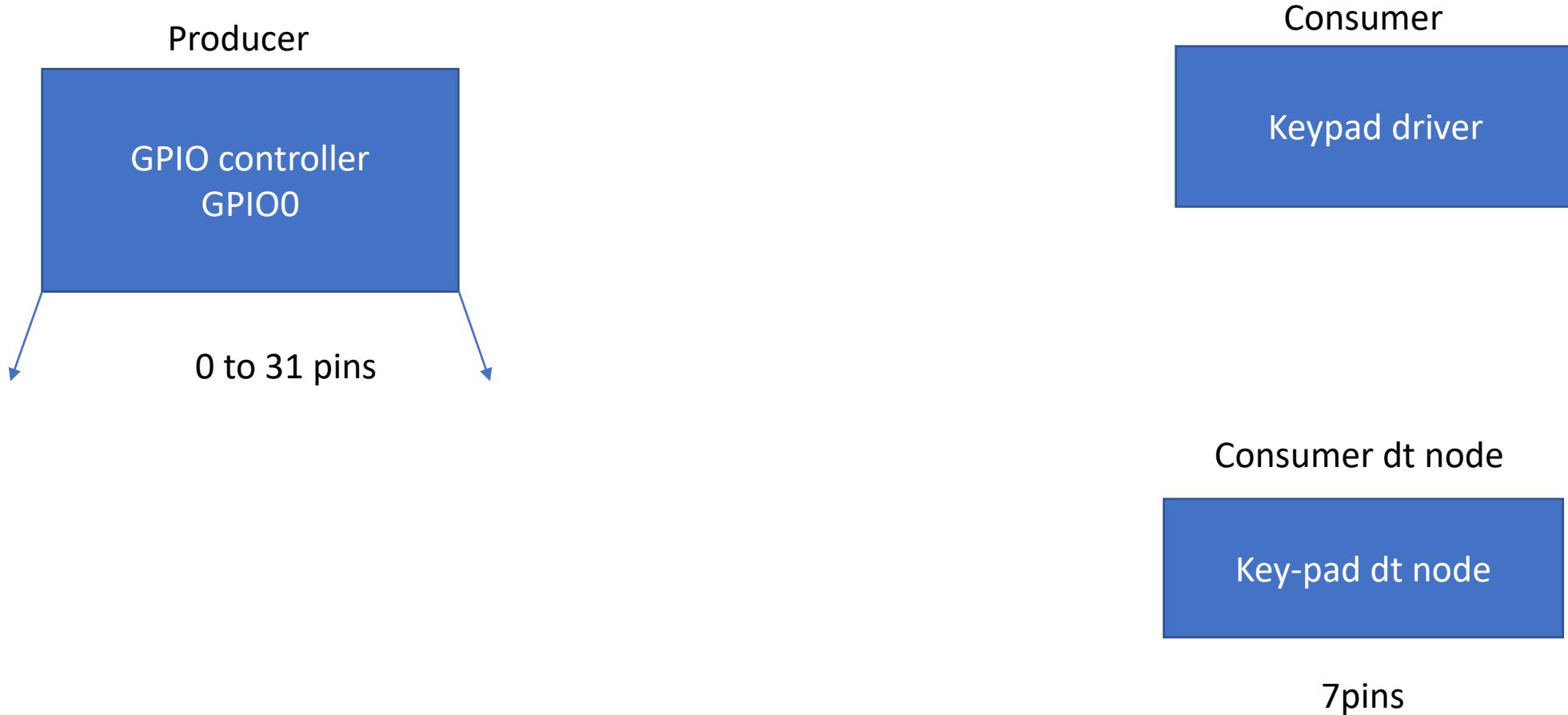
MUXMODE	Selected Mode
000b	Primary Mode = Mode 0
001b	Mode 1
010b	Mode 2
011b	Mode 3
100b	Mode 4
101b	Mode 5
110b	Mode 6
111b	Mode 7

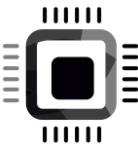
Mode 0 is the primary mode. When mode 0 is set, the function mapped to the pin corresponds to the name of the pin. Mode 1 to mode 7 are possible modes for alternate functions. On each pin, some modes are used effectively for alternate functions, while other modes are unused and correspond to no functional configuration.





Consumer accessing GPIO pins



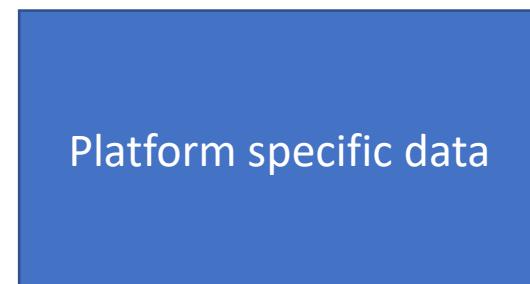


Representing GPIOs in consume dt node

GPIO number

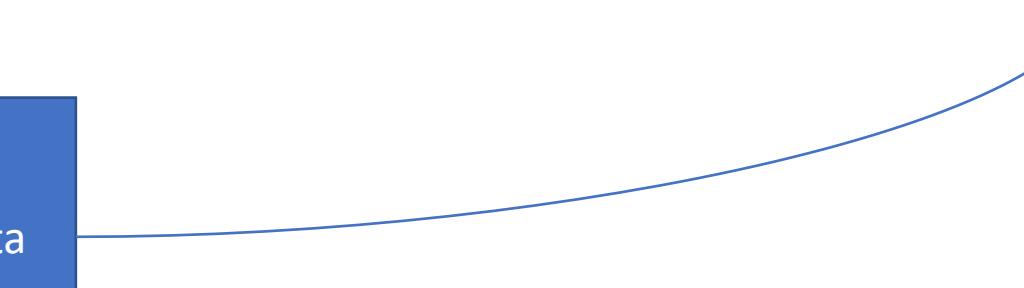
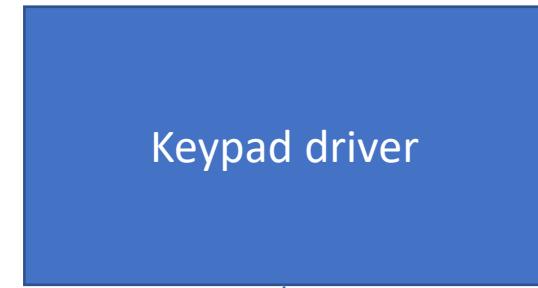
GPIO controller number

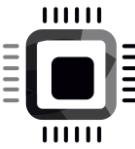
Active high or active low status



Dt node for the keypad device

The consumer driver accesses the dt node to extract the information about the gpios used to connect the keypad and configures them

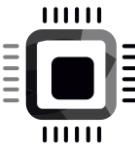




GPIO device tree property

GPIOs being consumed are represented by **<function>-gpios** property

- <function> being the purpose of this GPIO for the consumer
- Omitting <function > is OK but not recommended
- <function>-gpio is OK but not recommended
- So best practice is always use in the form :<function>-gpios

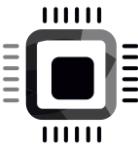


GPIOs mappings are defined in the consumer device's node, in a property named <function>-gpios, where <function> is the function the driver will request through gpiod_get(). For example:

```
foo_device {  
    compatible = "acme,foo";  
    ...  
    Consumer node  
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */  
                <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */  
                <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */  
  
    consumer flags  
    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;  
};
```

Properties named <function>-gpio are also considered valid and old bindings use it but are only supported for compatibility reasons and should not be used for newer bindings since it has been deprecated.

This property will make GPIOs 15, 16 and 17 available to the driver under the "led" function, and GPIO 1 as the "power" GPIO:



phandle to the GPIO controller
to which this pin belongs to

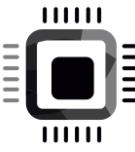
local offset to the GPIO line

```
foo_device {  
    compatible = "acme,foo";  
    ...  
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */  
                <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */  
                <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */  
  
    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;  
};
```

Function name . Driver
binding document tells you
what name to use.
'Conn-id'

gpio flags

A GPIO consumer dt node

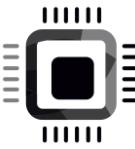


GPIO consumer flags

flags is defined to specify the following properties:

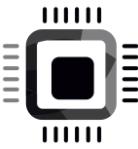
- * `GPIO_ACTIVE_HIGH` - GPIO line is active high
- * `GPIO_ACTIVE_LOW` - GPIO line is active low
- * `GPIO_OPEN_DRAIN` - GPIO line is set up as open drain
- * `GPIO_OPEN_SOURCE` - GPIO line is set up as open source
- * `GPIO_PERSISTENT` - GPIO line is persistent during suspend/resume and maintains its value
- * `GPIO_TRANSITORY` - GPIO line is transitory and may loose its electrical state during suspend/resume

include/dt-bindings/gpio/gpio.h



GPIO kernel functions to manage GPIOs

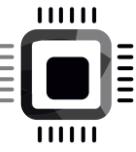
- The GPIO subsystem exposes many kernel function which the consumer driver can use to manage the GPIOs.
- Consumer driver should include *linux/gpio/consumer.h* header file where gpio manipulation function prototypes are available.



Acquiring and Disposing GPIOs

```
foo_device {  
    compatible = "acme,foo";  
    ...  
    led-gpios = <&gpio 15 GPIO_ACTIVE_HIGH>, /* red */  
                <&gpio 16 GPIO_ACTIVE_HIGH>, /* green */  
                <&gpio 17 GPIO_ACTIVE_HIGH>; /* blue */  
  
    power-gpios = <&gpio 1 GPIO_ACTIVE_LOW>;  
};
```

A gpio consumer dt node



Acquire and dispose a GPIO

Acquire

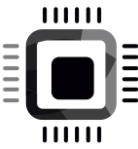
```
struct gpio_desc* gpiod_get(struct device *dev, const char *con_id, \
                           enum gpiod_flags flags)
```

Dispose

```
void gpiod_put(struct gpio_desc *desc)
```

Device-managed variants of GPIO functions are also available .

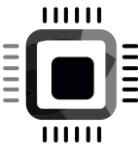
Check consumer.txt



Acquiring GPIO using index

when the function is implemented by using several GPIOs together

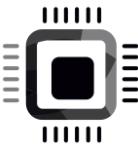
```
struct gpio_desc* gpiod_get_index(struct device *dev, const char *con_id, \
                                  unsigned int idx, enum gpiod_flags flags)
```



GPIO initialization flags

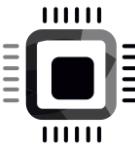
```
/**  
 * Optional flags that can be passed to one of gpiod_* to configure direction  
 * and output value. These values cannot be OR'd.  
 */  
enum gpiod_flags {  
    GPIOD_ASIS = 0,  
    GPIOD_IN = GPIOD_FLAGS_BIT_DIR_SET,  
    GPIOD_OUT_LOW = GPIOD_FLAGS_BIT_DIR_SET | GPIOD_FLAGS_BIT_DIR_OUT,  
    GPIOD_OUT_HIGH = GPIOD_FLAGS_BIT_DIR_SET | GPIOD_FLAGS_BIT_DIR_OUT |  
                    GPIOD_FLAGS_BIT_DIR_VAL,  
    GPIOD_OUT_LOW_OPEN_DRAIN = GPIOD_OUT_LOW | GPIOD_FLAGS_BIT_OPEN_DRAIN,  
    GPIOD_OUT_HIGH_OPEN_DRAIN = GPIOD_OUT_HIGH | GPIOD_FLAGS_BIT_OPEN_DRAIN,  
};
```

Include/linux/consumer.h



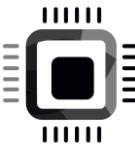
GPIO manipulations

- What you do with GPIO lines?
 - Configure its direction (input or output)
 - Change its output state to 0 or 1
 - Read input state
 - Configure output type (push-pull/open-drain)
 - Enable/Disable pull-up/pull-down resistors



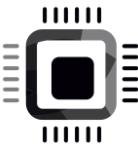
Configuring direction and flags

Function	note
<code>int gpiod_direction_input(struct gpio_desc *desc);</code>	set the GPIO direction to input
<code>int gpiod_direction_output(struct gpio_desc *desc, int value);</code>	set the GPIO direction to output by taking initial output value of the GPIO
<code>int gpiod_direction_output_raw (struct gpio_desc *desc, int value)</code>	set the GPIO direction to output without regard for the ACTIVE_LOW status
<code>int gpiod_configure_flags (struct gpio_desc *desc, const char *con_id, unsigned long lflags, enum gpiod_flags dflags)</code>	helper function to configure a given GPIO



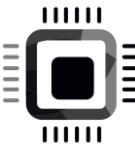
Get and set

Function	note
<code>int gpiod_get_direction(struct gpio_desc *desc)</code>	return the current direction of a GPIO Returns 0 for output, 1 for input
<code>int gpiod_get_value(const struct gpio_desc *desc);</code>	Return the GPIO's logical value, i.e. taking the ACTIVE_LOW status into account
<code>int gpiod_get_raw_value(const struct gpio_desc *desc);</code>	Return the GPIO's raw value, i.e. the value of the physical line disregarding its ACTIVE_LOW status
<code>void gpiod_set_value(struct gpio_desc *desc, int value);</code>	Set the logical value of the GPIO, i.e. taking its ACTIVE_LOW, OPEN_DRAIN and OPEN_SOURCE flags into account.
<code>void gpiod_set_raw_value(struct gpio_desc *desc, int value);</code>	Set the raw value of the GPIO, i.e. the value of its physical line without regard for its ACTIVE_LOW status.
<code>int gpiod_set_debounce(struct gpio_desc *desc, unsigned debounce);</code>	sets debounce time for a GPIO
<code>int gpiod_is_active_low(const struct gpio_desc *desc);</code>	test whether a GPIO is active-low or not Returns 1 if the GPIO is active-low, 0 otherwise



Accessing GPIO value from atomic context

```
int gpiod_cansleep(const struct gpio_desc *desc);
void gpiod_set_value_cansleep(struct gpio_desc *desc, int value);
void gpiod_set_raw_value_cansleep(struct gpio_desc *desc, int value);
int gpiod_get_value_cansleep(const struct gpio_desc *desc);
int gpiod_get_raw_value_cansleep(const struct gpio_desc *desc);
```



Exercise

Write a GPIO sysfs driver.

The goal of this exercise is to handle GPIOs of the hardware through Sysfs interface

The driver should support the below functionality

- 1)The driver should create a class "bone_gpios" under /sys/class (class_create)
- 2)For every detected GPIO in the device tree, the driver should create a device under /sys/class/bone_gpios (device_create)
- 3) the driver should also create 3 sysfs files(attributes) for every gpio device

attribute 1) direction:

used to configure the gpio direction

possible values: 'in' and 'out'

mode : (read /write)

attribute 2) value:

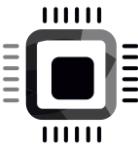
used to enquire the state of the gpio or to write a new value to the gpio

possible values : 0 and 1 (read/write)

attribute 3) label:

used to enquire label of the gpio (read-only)

- 4) implement show and store methods for the attributes



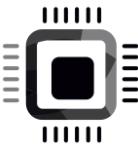
Device-Driver binding information

The device tree for this driver must be created like below

```
gpio_devs
{
    compatible = "org,bone-gpio-sysfs";

    gpio_1 {
        label = "gpio1.21"; /* optional */
        bone-gpios = <&gpio1 21 GPIO_ACTIVE_HIGH>; /* mandatory */
    };

    gpio_2 {
        label = "gpio1.22";
        bone-gpios = <&gpio1 22 GPIO_ACTIVE_HIGH>;
    };
}
```



```
struct device
```

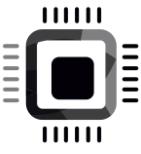
```
/* associated device tree  
node */  
  
struct device_node  
*of_node  
  
/* firmware device node */  
  
struct fwnode_handle  
*fwnode;
```

@fwnode: firmware node containing GPIO reference
@con_id: function within the GPIO consumer

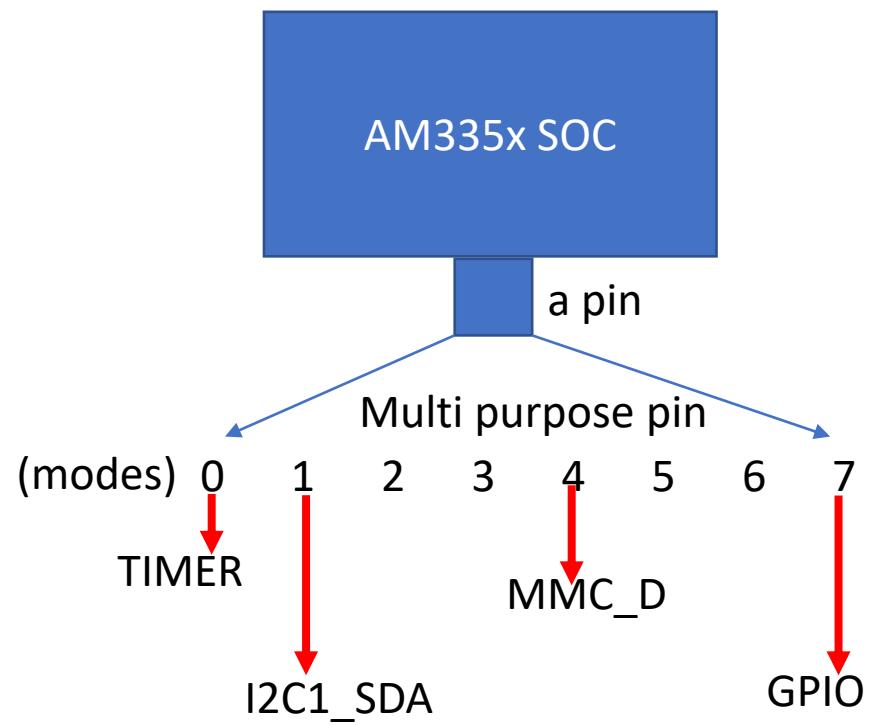
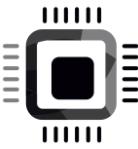
```
struct device_node
```

```
struct fwnode_handle  
fwnode
```

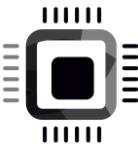
* The function properly finds the corresponding GPIO using whatever is the * underlying firmware interface and then makes sure that the GPIO * descriptor is requested before it is returned to the caller



Pin control subsystem



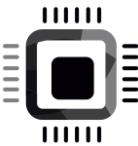
The functionality of a pin depends on its mode configured in mode/pad configuration register.



Pin count of AM335x



Pin Package type : ZCZ
Total pin count = 324



Pad configuration register of AM335x

Table 10. Expansion Header P8 Pinout

PIN	PROC	NAME	MODE0	MODE1	MODE2	MODE3	MODE4	MODE5	MODE6	MODE7
1,2					GND					
3	R9	GPIO1_6	gpmc_ad6	mmc1_dat6						gpio1[6]
4	T9	GPIO1_7	gpmc_ad7	mmc1_dat7						gpio1[7]
5	R8	GPIO1_2	gpmc_ad2	mmc1_dat2						gpio1[2]
6	T8	GPIO1_3	gpmc_ad3	mmc1_dat3						gpio1[3]
7	R7	TIMER4	gpmc_advn_ale		timer4					gpio2[2]
8	T7	TIMER7	gpmc_oen_ren		timer7					gpio2[3]
9	T6	TIMER5	gpmc_be0n_cle		timer5					gpio2[5]
10	U6	TIMER6	gpmc_wen		timer6					gpio2[4]
11	R12	GPIO1_13	gpmc_ad13	lcd_data18	mmc1_dat5	mmc2_dat1	eQEP2B_in			gpio1[13]
12	T12	GPIO1_12	GPCM_AD12	LCD_DATA19	Mmc1_dat4	MMC2_DAT0	EQEP2A_IN			gpio1[12]
13	T10	EHRPWM2B	gpmc_ad9	lcd_data22	mmc1_dat1	mmc2_dat5	ehr pwm2B			gpio0[23]
14	T11	GPIO0_26	gpmc_ad10	lcd_data21	mmc1_dat2	mmc2_dat6	ehr pwm2_tripzone_in			gpio0[26]
15	U13	GPIO1_15	gpmc_ad15	lcd_data16	mmc1_dat7	mmc2_dat3	eQEP2_strobe			gpio1[15]
16	V13	GPIO1_14	gpmc_ad14	lcd_data17	mmc1_dat6	mmc2_dat2	eQEP2_index			gpio1[14]
17	U12	GPIO0_27	gpmc_ad11	lcd_data20	mmc1_dat3	mmc2_dat7	ehr pwm0_sync			gpio0[27]
18	V12	GPIO2_1	gpmc_clk_mux0	lcd_memory_clk	gpmc_wait1	mmc2_clk		mcasp0_fsr	gpio2[1]	
19	U10	EHRPWM2A	gpmc_ad8	lcd_data23	mmc1_dat0	mmc2_dat4	ehr pwm2A			gpio0[22]
20	V9	GPIO1_31	gpmc_csn2	gpmc_be1n	mmc1_cmd					gpio1[31]
21	U9	GPIO1_30	gpmc_csn1	gpmc_clk	mmc1_clk					gpio1[30]
22	V8	GPIO1_5	gpmc_ad5	mmc1_dat5						gpio1[5]
23	U8	GPIO1_4	gpmc_ad4	mmc1_dat4						gpio1[4]
24	V7	GPIO1_1	gpmc_ad1	mmc1_dat1						gpio1[1]
25	U7	GPIO1_0	gpmc_ad0	mmc1_dat0						gpio1[0]
26	V6	GPIO1_29	gpmc_csn0							gpio1[29]
27	U5	GPIO2_22	lcd_vsync	gpmc_a8						gpio2[22]
28	V5	GPIO2_24	lcd_pclk	gpmc_a10						gpio2[24]

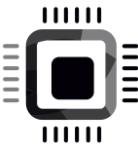
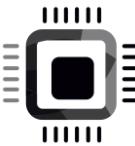


Table 4-2. Pin Attributes (ZCE and ZCZ Packages) (continued)

ZCE BALL NUMBER [1]	ZCZ BALL NUMBER [1]	PIN NAME [2]	SIGNAL NAME [3]	MODE [4]	TYPE [5]	BALL RESET STATE [6] ⁽²⁵⁾	BALL RESET REL. STATE [7]	RESET REL. MODE [8]	ZCE POWER / ZCZ POWER [9]	HYS [10]	BUFFER STRENGTH (mA) [11]	PULLUP /DOWN TYPE [12]	I/O CELL [13]
V12	R8	GPMC_AD2	gpmc_ad2	0	I/O	L	L	7	VDDSHV1 / VDDSHV1	Yes	6	PU/PD	LVCMOS
			mmc1_dat2	1	I/O								
			gpio1_2	7	I/O								
W13	T8	GPMC_AD3	gpmc_ad3	0	I/O	L	L	7	VDDSHV1 / VDDSHV1	Yes	6	PU/PD	LVCMOS
			mmc1_dat3	1	I/O								
			gpio1_3	7	I/O								
V13	U8	GPMC_AD4	gpmc_ad4	0	I/O	L	L	7	VDDSHV1 / VDDSHV1	Yes	6	PU/PD	LVCMOS
			mmc1_dat4	1	I/O								
			gpio1_4	7	I/O								
W14	V8	GPMC_AD5	gpmc_ad5	0	I/O	L	L	7	VDDSHV1 / VDDSHV1	Yes	6	PU/PD	LVCMOS
			mmc1_dat5	1	I/O								
			gpio1_5	7	I/O								
U14	R9	GPMC_AD6	gpmc_ad6	0	I/O	L	L	7	VDDSHV1 / VDDSHV1	Yes	6	PU/PD	LVCMOS
			mmc1_dat6	1	I/O								
			gpio1_6	7	I/O								



9.3.51 *conf_<module>_<pin>* Register (offset = 800h–A34h)

See the device datasheet for information on default pin mux configurations. Note that the device ROM may change the default pin mux for certain pins based on the SYSBOOT mode settings.

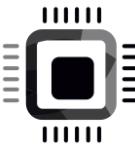
See [Table 9-10, Control Module Registers Table](#), for the full list of offsets for each module/pin configuration.

conf_<module>_<pin> is shown in [Figure 9-54](#) and described in [Table 9-61](#).

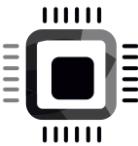
Figure 9-54. *conf_<module>_<pin>* Register

31	30	29	28	27	26	25	24
Reserved							
R-0h							
23	22	21	20	19	18	17	16
Reserved				Reserved			
R-0h				R-0h			
15	14	13	12	11	10	9	8
Reserved							
R-0h							
7	6	5	4	3	2	1	0
Reserved	<i>conf_<module>_<pin></i> _slewctrl	<i>conf_<module>_<pin></i> _rxactive	<i>conf_<module>_<pin></i> _putypesel	<i>conf_<module>_<pin></i> _puden	<i>conf_<module>_<pin></i> _mmode		
R-0h	R/W-0h	R/W-1h	R/W-0h	R/W-0h	R/W-0h		

LEGEND: R/W = Read/Write; R = Read only; W1toCl = Write 1 to clear bit; -n = value after reset

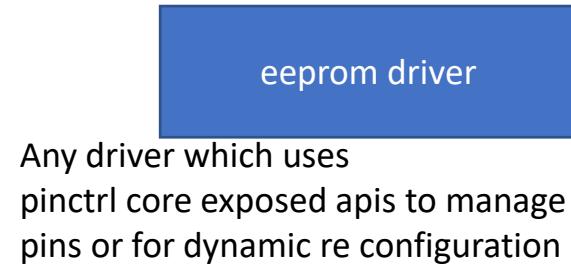
**Table 9-61. conf_<module>_<pin> Register Field Descriptions**

Bit	Field	Type	Reset	Description
31-20	Reserved	R	0h	
19-7	Reserved	R	0h	
6	conf_<module>_<pin>_slewctrl	R/W	X	Select between faster or slower slew rate 0: Fast 1: Slow Reset value is pad-dependent.
5	conf_<module>_<pin>_rxactive	R/W	1h	Input enable value for the PAD 0: Receiver disabled 1: Receiver enabled
4	conf_<module>_<pin>_pu typesel	R/W	X	Pad pullup/pulldown type selection 0: Pulldown selected 1: Pullup selected Reset value is pad-dependent.
3	conf_<module>_<pin>_pu den	R/W	X	Pad pullup/pulldown enable 0: Pullup/pulldown enabled 1: Pullup/pulldown disabled Reset value is pad-dependent.
2-0	conf_<module>_<pin>_m mode	R/W	X	Pad functional signal mux select. Reset value is pad-dependent.

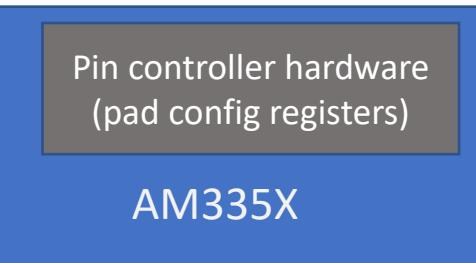
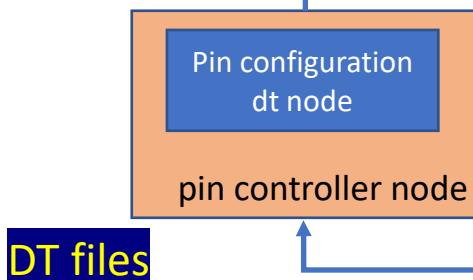
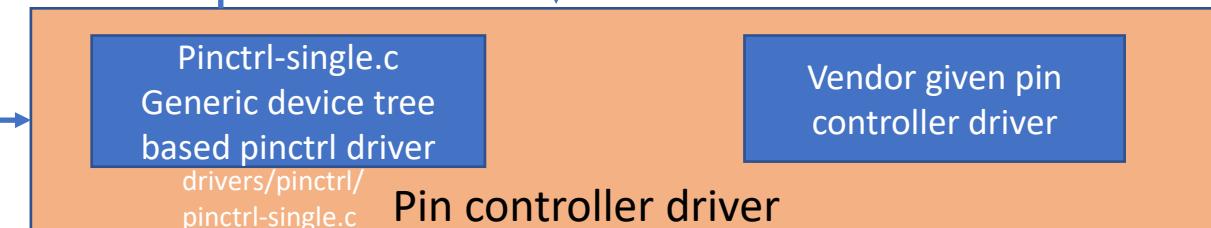
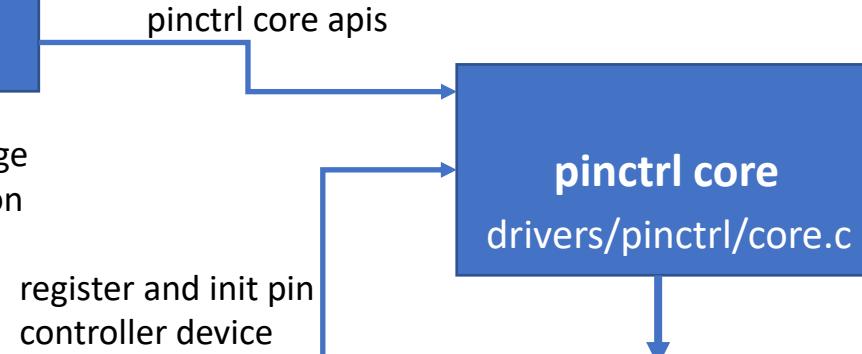


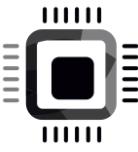
Using pin-control subsystem of Linux

User space

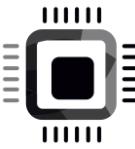


Kernel space



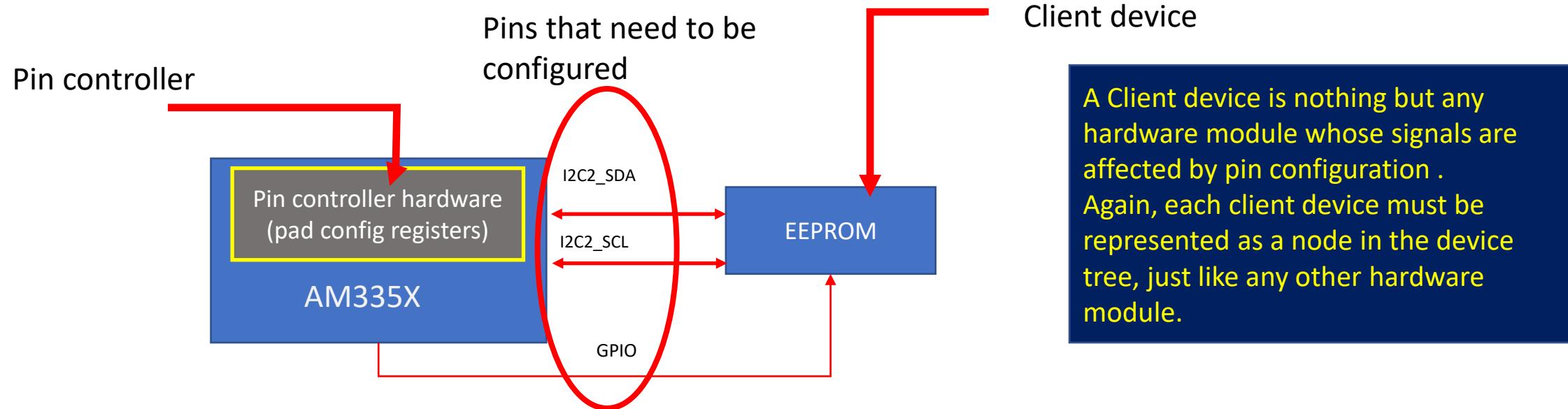
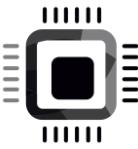


- Pin control core
 - Pinctrl core implementation you can find in *drivers/pinctrl/core.c*
 - This provides pin control helper functions to any consumer drivers
 - Maintains pin exclusivity for a device
 - Provides debug interface to user space through sysfs



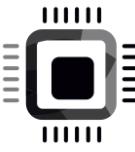
Pin control subsystem of linux

- Pin control subsystem of Linux is used to configure pins of a microprocessor/microcontroller
- What is pin configuration ?
 - Pin mode configuration for alternate functions
 - Slew rate adjustment
 - Driver strength
 - Pull up and pull down resistor enable/disable
 - Output type control (push pull, open drain)



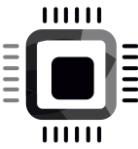
Device tree nodes

1. Node which explains the pin controller
2. Node which explains configuration details for individual pins
3. Node which claims the pins (Client device node)



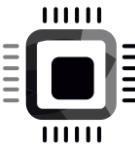
What is a pin-controller ?

- Hardware modules that control pin multiplexing and configuration parameters such as pull-up/down, tri-state, drive-strength are designated as pin controllers.
- Each pin controller must be represented as a node in the device tree, just like any other hardware module node.
- For am335x, the pad config registers are called as pin controllers



How to write a pin-controller node ?

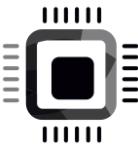
- This is explained by your pin controller driver's binding document
- If you are using generic pinctrl driver then refer *pinctrl-single.txt* under *Documentation/devicetree/bindings/pinctrl/*



Example of pin controller node

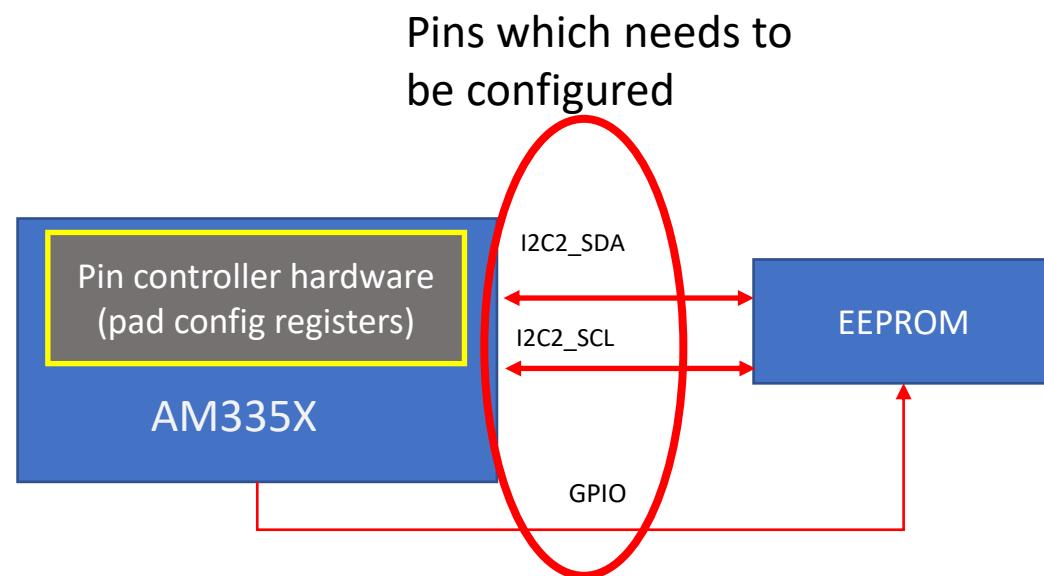
```
am33xx_pinmux: pinmux@800 {
    compatible = "pinctrl-single";
    reg = <0x800 0x238>;
    #pinctrl-cells = <1>;
    pinctrl-single,register-width = <32>;
    pinctrl-single,function-mask = <0x7f>;
};
```

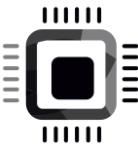
am33xx-i4.dtsi



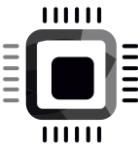
How to write pin configuration node ?

- Should be explained by your pin controller driver





```
&am33xx_pinmux {  
    i2c2_pins: pinmux_i2c2_pins {  
        pinctrl-single,pins = <  
            AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLUP, MUX_MODE0)  
            AM33XX_PADCONF(AM335X_PIN_I2C2_SCL, PIN_INPUT_PULLUP, MUX_MODE0)  
        >;  
    };  
};  
  
pinctrl-single,pins
```



pinctrl-single,pins property

The pin configuration nodes for pinctrl-single are specified as pinctrl register offset and value pairs using pinctrl-single,pins.

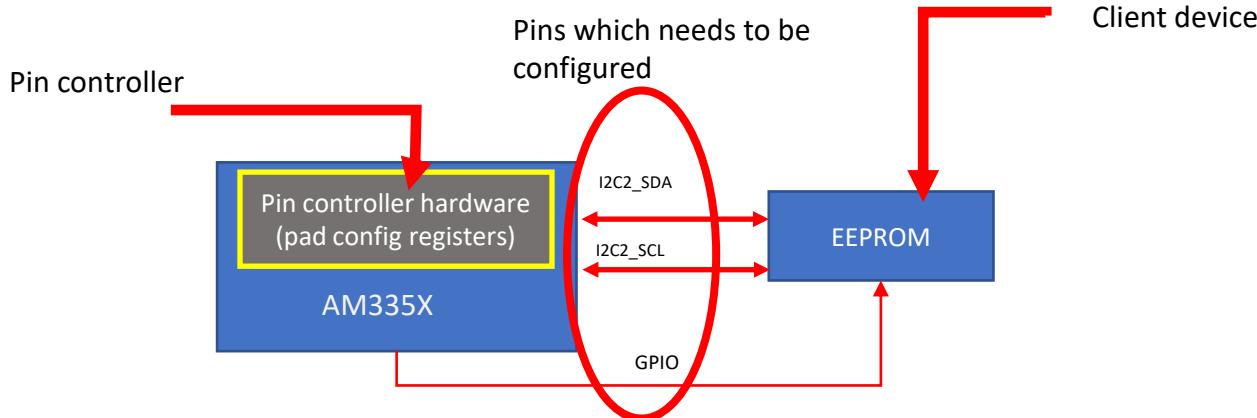
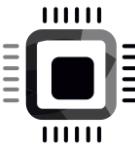
For example, setting a pin for a device could be done with:

```
pinctrl-single,pins = <0xdc 0x118>;
```

Where 0xdc is the offset from the pinctrl register base address for the device pinctrl register, and 0x118 contains the desired value of the pinctrl register. See the device example and static board pins example below for more information.

Reference :

Documentation/devicetree/bindings/pinctrl/pinctrl-single.txt

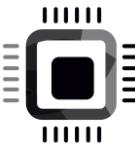


```
&am33xx_pinmux {
    i2c2_pins: pinmux_i2c2_pins {
        pinctrl-single,pins = <
            AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLUP, MUX_MODE0)
            AM33XX_PADCONF(AM335X_PIN_I2C2_SCL, PIN_INPUT_PULLUP, MUX_MODE0)
        >;
    };
};
```

Pin configuration node embedded in pin controller node

```
cape_eeprom0: cape_eeprom0@54 {
    pinctrl-names = "default";
    pinctrl-0 = <&i2c2_pins>;
    compatible = "atmel,24c256";
    reg = <0x54>;
    #address-cells = <1>;
    #size-cells = <1>;
    cape0_data: cape_data@0 {
        reg = <0 0x100>;
    };
};
```

Client device node

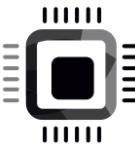


Pinctrl related properties to be used with client device nodes

- For full discussion visit :
Documentation/devicetree/bindings/pinctrl/pinctrl-bindings.txt

pinctrl-names : The list of names to assign states. List entry 0 defines the name for integer state ID 0, list entry 1 for state ID 1, and so on.

pinctrl-<id> : List of phandles, each pointing at a pin configuration node within a pin controller

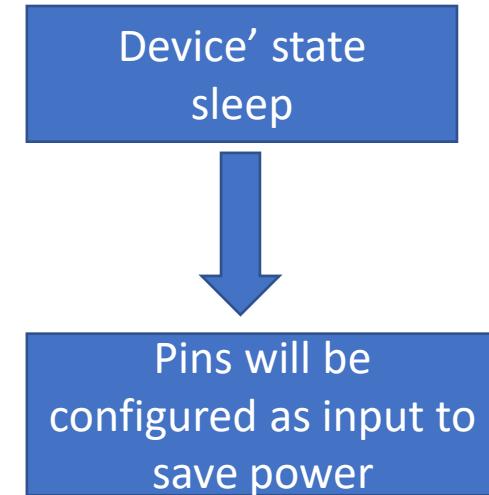
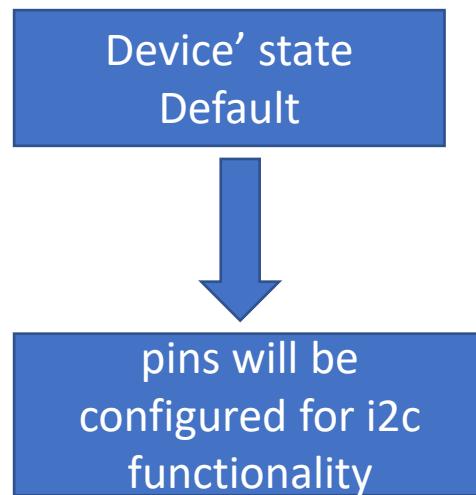


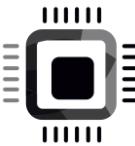
Pin state id and pin state name

You can define, different set of pin configuration values for different state of the device

A device's state could be *default, sleep, idle, active, etc*

Different pin states help to achieve dynamic configurability of pins





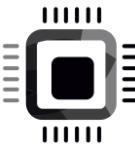
```
i2c2_pins: pinmux_i2c2_pins {
    pinctrl-single,pins = <
        AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLUP, MUX_MODE0)
        AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLUP, MUX_MODE0)
    >;
};

i2c2_pins_sleep: pinmux_i2c2_pins_sleep {

    pinctrl-single,pins = <
        AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLDOWN, MUX_MODE7)
        AM33XX_PADCONF(AM335X_PIN_I2C2_SDA, PIN_INPUT_PULLDOWN, MUX_MODE7)
    >;
};
```

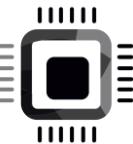
Here, a pin controller has 2 nodes. The first one configures the pins for i2c functionality and the second one configures the same pins for gpio input functionality .

The client device will use the first one during normal operation , and when driver decides to put device to sleep, it will use the second one to reconfigure the pins to stop leakage of current thus achieving low power



Client device node with 2 pin states

```
cape_eeprom0: cape_eeprom0@54 {  
  
    pinctrl-names = "default", "sleep";  
    pinctrl-0 = <&i2c2_pins>;  
    pinctrl-1 = <&i2c2_pins_sleep>;  
  
    compatible = "atmel,24c256";  
    reg = <0x54>;  
    #address-cells = <1>;  
    #size-cells = <1>;  
    cape0_data: cape_data@0 {  
        reg = <0 0x100>;  
    };  
};
```



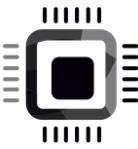
AM335x pad configuration macro

```
/*
 * Macros to allow using the absolute physical address instead of the
 * padconf registers instead of the offset from padconf base.
 */
#define OMAP_IOPAD_OFFSET(pa, offset)  (((pa) & 0xffff) - (offset))

#define AM33XX_PADCONF(pa, dir, mux)    OMAP_IOPAD_OFFSET((pa), 0x0800) ((dir) | (mux))
```

Pad configuration register offset Pad direction and Pupd control Mode(0...7)

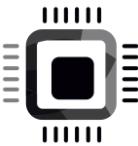
Include/bt-bindings/pinctrl/omap.h



```
/*
 * Macros to allow using the absolute physical address instead of the
 * padconf registers instead of the offset from padconf base.
 */
#define OMAP_IOPAD_OFFSET(pa, offset)  (((pa) & 0xffff) - (offset))

#define AM33XX_PADCONF(pa, dir, mux)    OMAP_IOPAD_OFFSET((pa), 0x0800) ((dir) | (mux))
                                         ^                         ^           ^
                                         Address of pad-conf reg   value
```

include/bt-bindings/pinctrl/omap.h



```
#define AM335X_PIN_GPMC_AD0          0x800
#define AM335X_PIN_GPMC_AD1          0x804
#define AM335X_PIN_GPMC_AD2          0x808
#define AM335X_PIN_GPMC_AD3          0x80c
#define AM335X_PIN_GPMC_AD4          0x810
#define AM335X_PIN_GPMC_AD5          0x814
#define AM335X_PIN_GPMC_AD6          0x818
#define AM335X_PIN_GPMC_AD7          0x81c
#define AM335X_PIN_GPMC_AD8          0x820
#define AM335X_PIN_GPMC_AD9          0x824
#define AM335X_PIN_GPMC_AD10         0x828
#define AM335X_PIN_GPMC_AD11         0x82c
#define AM335X_PIN_GPMC_AD12         0x830
#define AM335X_PIN_GPMC_AD13         0x834
#define AM335X_PIN_GPMC_AD14         0x838
#define AM335X_PIN_GPMC_AD15         0x83c
#define AM335X_PIN_GPMC_A0           0x840
#define AM335X_PIN_GPMC_A1           0x844
#define AM335X_PIN_GPMC_A2           0x848
#define AM335X_PIN_GPMC_A3           0x84c
#define AM335X_PIN_GPMC_A4           0x850
#define AM335X_PIN_GPMC_A5           0x854
#define AM335X_PIN_GPMC_A6           0x858
#define AM335X_PIN_GPMC_A7           0x85c
#define AM335X_PIN_GPMC_A8           0x860
#define AM335X_PIN_GPMC_A9           0x864
#define AM335X_PIN_GPMC_A10          0x868
#define AM335X_PIN_GPMC_A11          0x86c
#define AM335X_PIN_GPMC_WAIT0        0x870
#define AM335X_PIN_GPMC_WPN          0x874
#define AM335X_PIN_GPMC_BEN1         0x878
#define AM335X_PIN_GPMC_CSN0         0x87c
#define AM335X_PIN_GPMC_CSN1         0x880
#define AM335X_PIN_GPMC_CSN2         0x884
#define AM335X_PIN_GPMC_CSN3         0x888
```

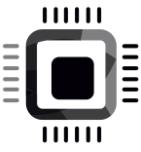
Pad configuration
register offset

Pad direction and Pupd control

```
#define PULL_DISABLE          (1 << 3)
#define INPUT_EN                (1 << 5)
#define SLEWCTRL_SLOW           (1 << 6)
#define SLEWCTRL_FAST            0

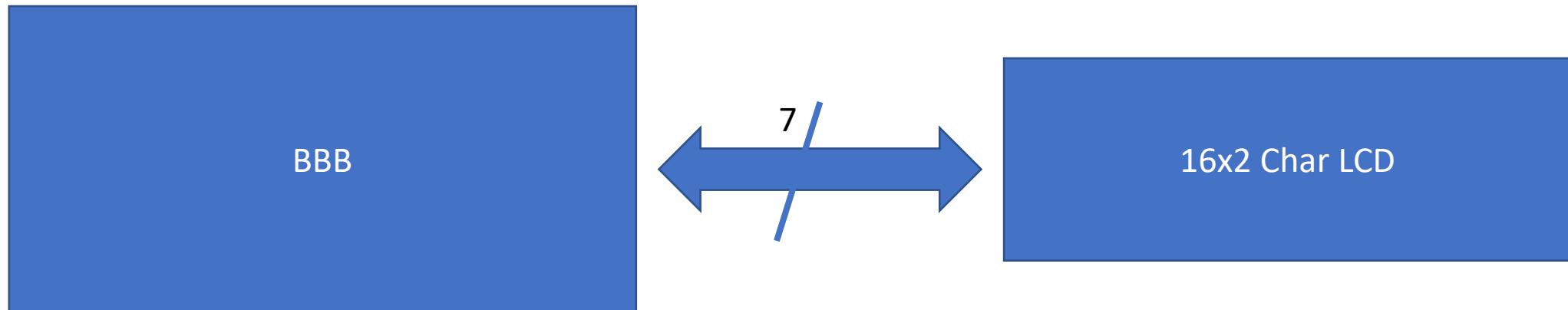
#define PIN_OUTPUT              (PULL_DISABLE)
#define PIN_OUTPUT_PULLUP        (PULL_UP)
#define PIN_OUTPUT_PULLDOWN      0
#define PIN_INPUT                (INPUT_EN | PULL_DISABLE)
#define PIN_INPUT_PULLUP          (INPUT_EN | PULL_UP)
#define PIN_INPUT_PULLDOWN        (INPUT_EN)
```

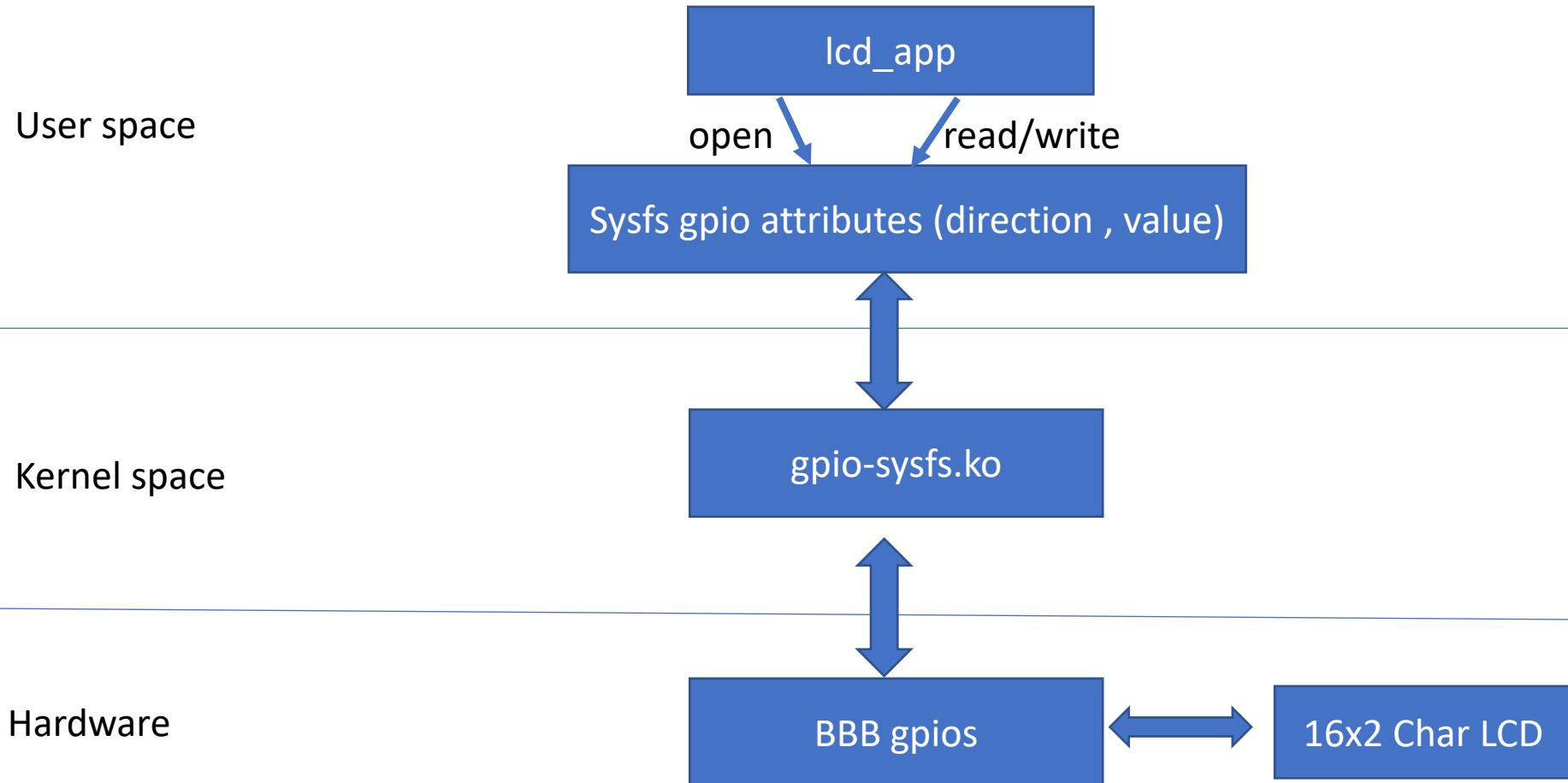
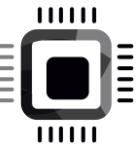
Include/bt-bindings/pinctrl/am33xx.h

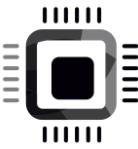


Exercise

Interfacing 16x2 LCD to BBB and testing with gpio-sysfs driver by writing an LCD application

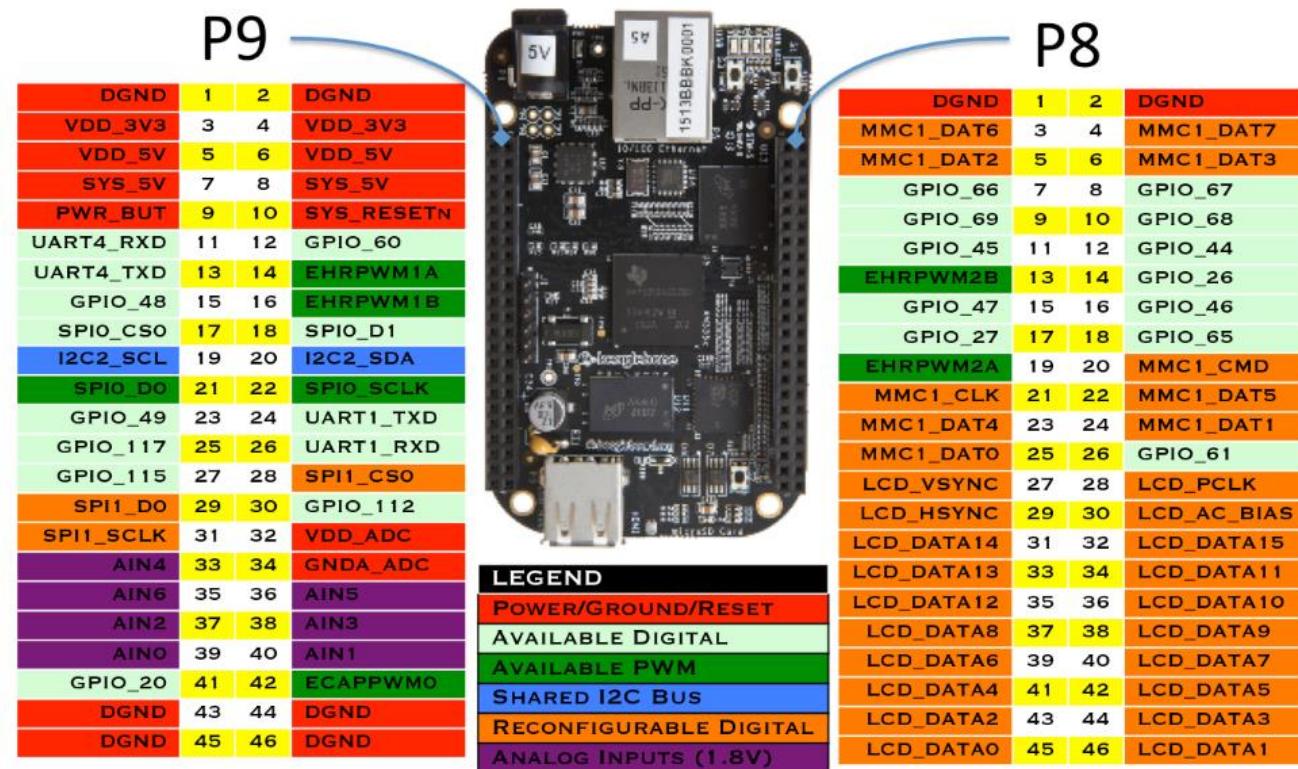


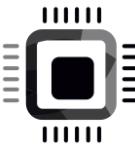




16x2 character LCD connection with BBB

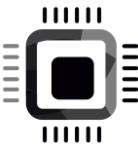
Cape Expansion Headers





Components required

- 16x2 Character LCD (1602A HD44780 LCD)
- 10K potentiometer
- Breadboard
- Connecting wires
- Beaglebone black

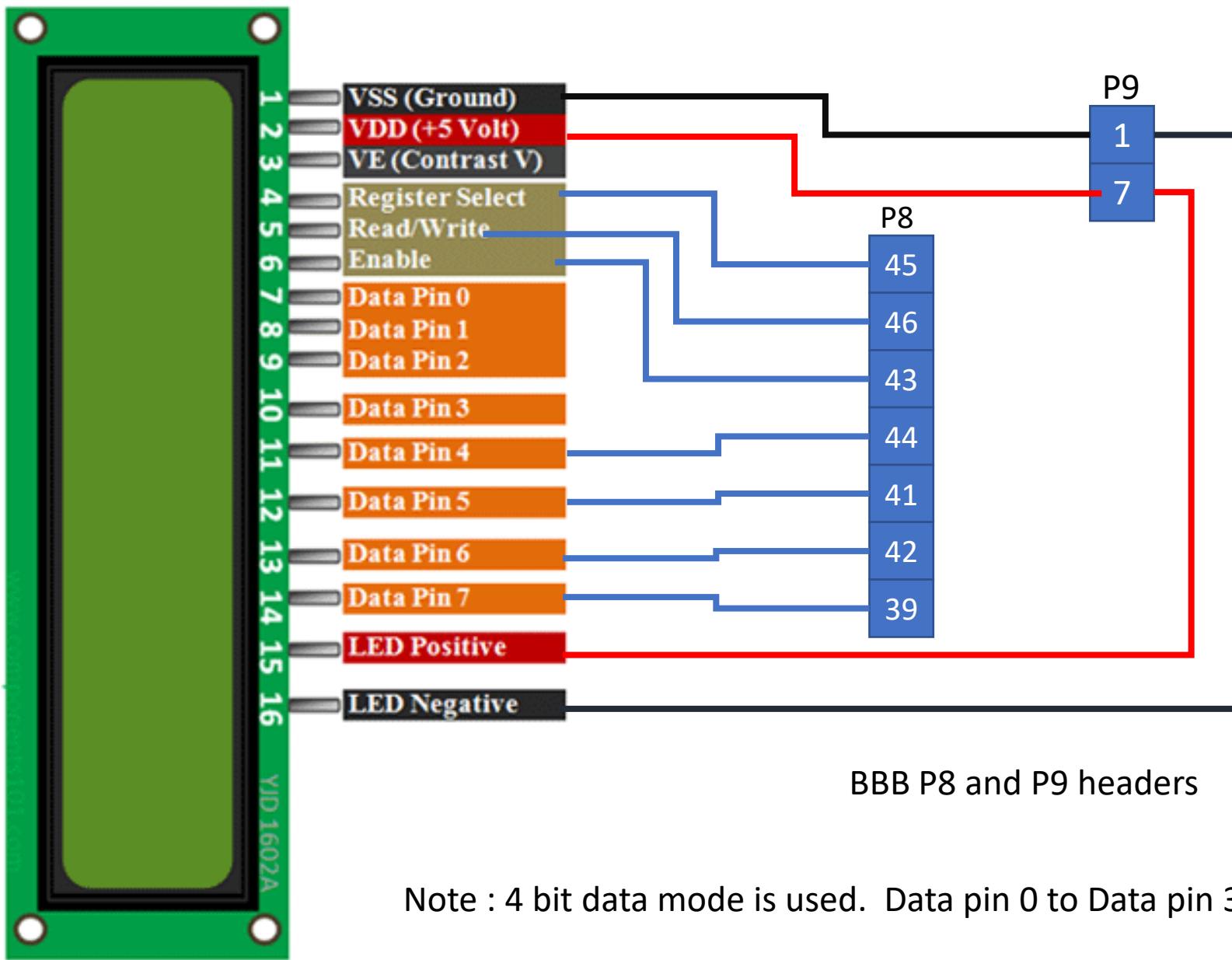
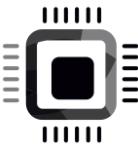


GPIOs of BBB

```
/*=====
BBB_expansion_header_pins      GPIO number    16x2 LCD pin      Purpose
=====

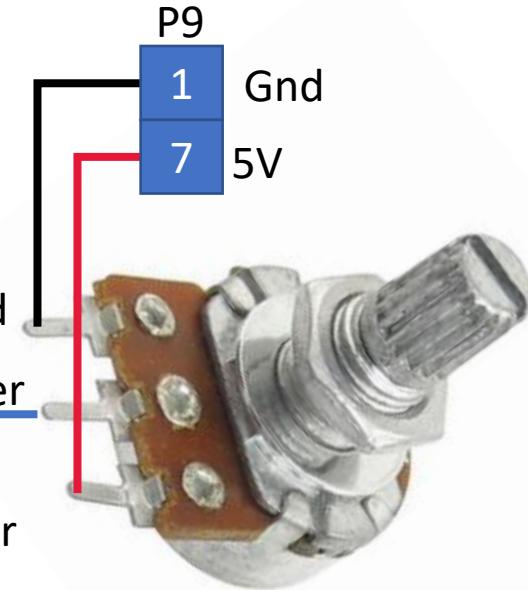
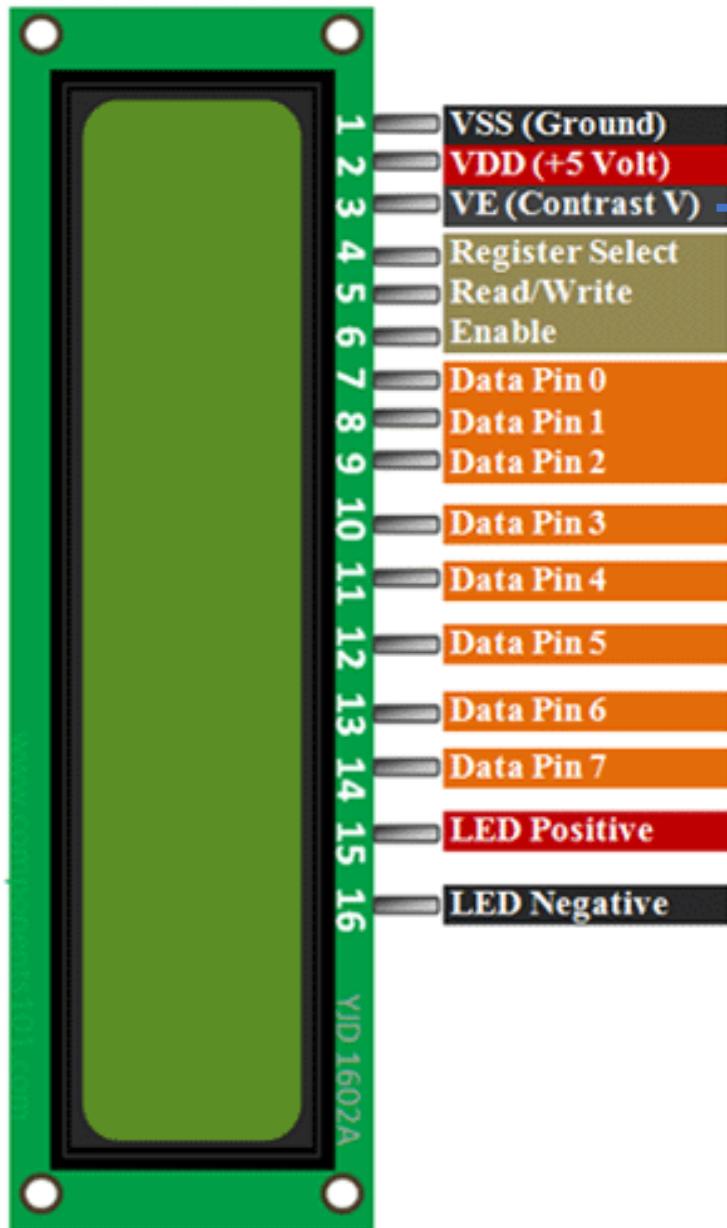
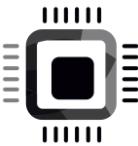
P8.45                      gpio2.6        4(RS)          Register selection (Character vs. Command)
P8.46                      gpio2.7        5(RW)          Read/write
P8.43                      gpio2.8        6(EN)          Enable
P8.44                      gpio2.9        11(D4)         Data line 4
P8.41                      gpio2.10       12(D5)         Data line 5
P8.42                      gpio2.11       13(D6)         Data line 6
P8.39                      gpio2.12       14(D7)         Data line 7
P9.1(GND)                   15(BKLTA)      Backlight anode(+)
P9.7(sys_5V supply)         16(BKLTK)      Backlight cathode(-)

P9.1 (GND)                  ----           1(VSS/GND)     Ground
P9.7(sys_5V supply)         ----           2(VCC)          +5V supply
===== */
```



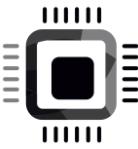
BBB P8 and P9 headers

Note : 4 bit data mode is used. Data pin 0 to Data pin 3 are unused



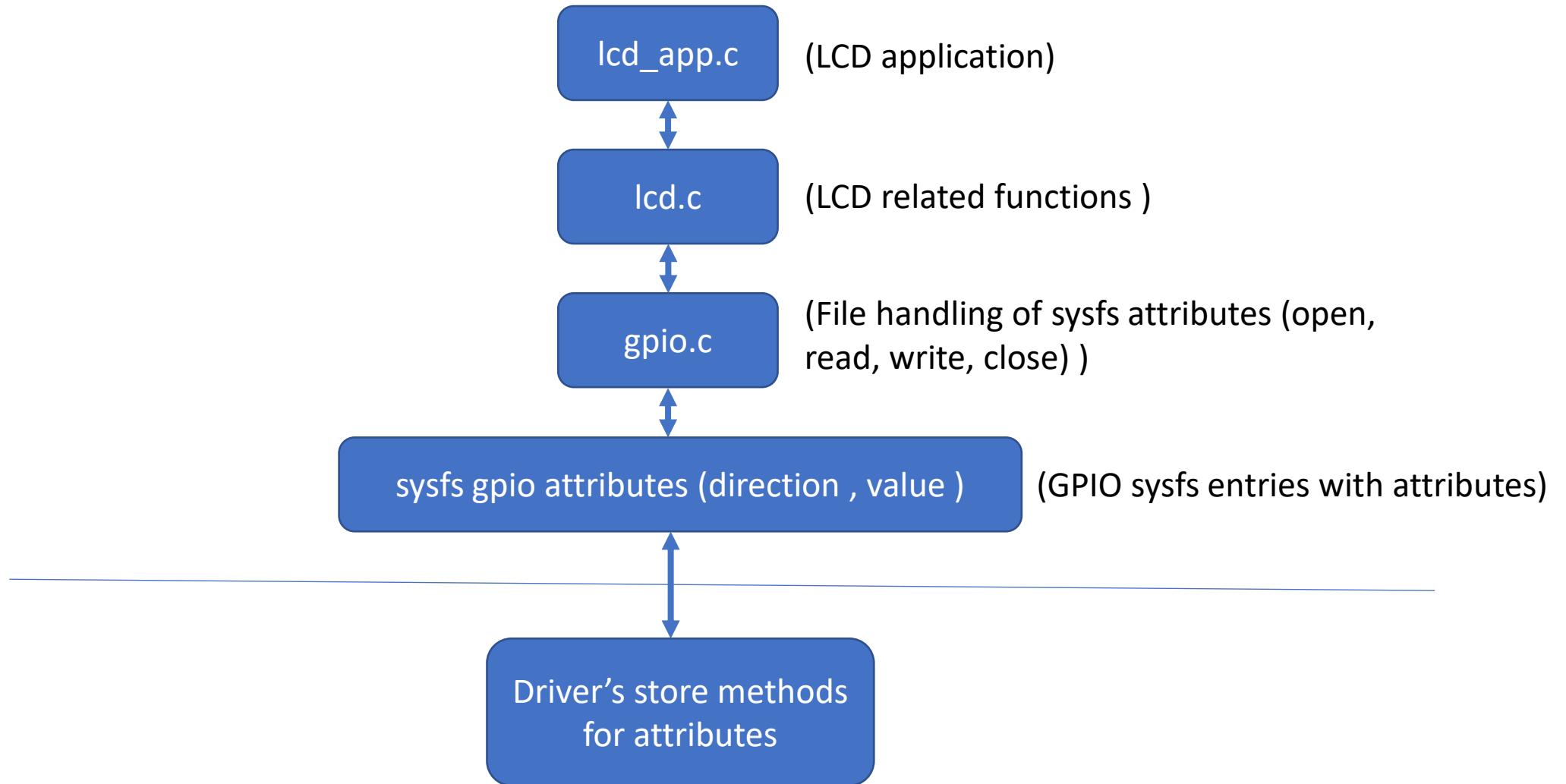
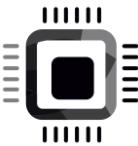
10K Potentiometer

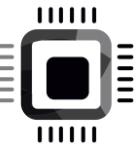
LCD Contrast control using Potentiometer



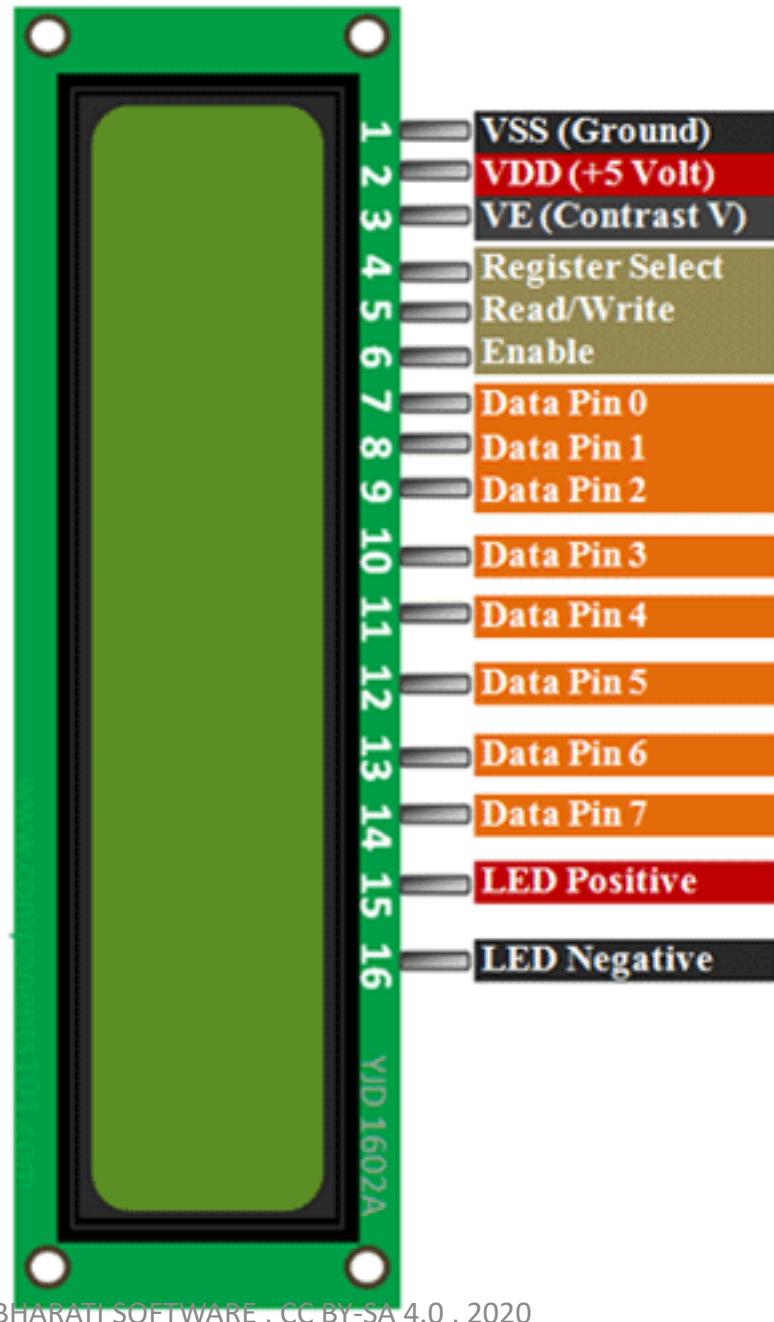
Steps

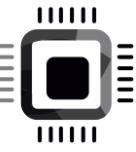
1. Connect LCD to BBB
2. Add required gpio entries to the gpio-sysfs device tree node
3. Recompile the dts file and make BBB boots with modified dtb
4. Load the gpio-sysfs driver
5. Make sure that all required gpio devices are formed under
`/sys/class/bone_gpios`
6. Download the lcd application files attached with this video.
`lcd_app.c`, `lcd.c`, `gpio.c`
7. Cross compile the lcd application and test it on the target





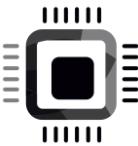
Significance of LCD pins





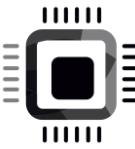
LCD commands

- Send commands to initialize and control the operation of the LCD
- Commands are 8bits long (a byte)



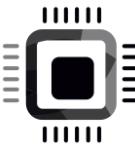
LCD commands

1. Function set
2. Display on/off, cursor on/off and blink control
3. Entry mode set
4. LCD clear display
5. Cursor return home
6. Set co-ordinates
7. Display right/left shift
8. Cursor on/off, blink on/off
9. Address counter read/write

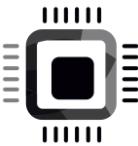


Sending command/data

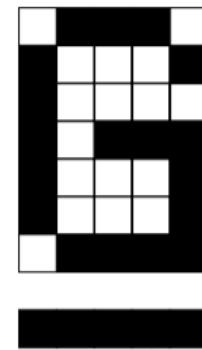
- A command or data byte is of 8bits (1byte)
- You can send all 8 bits in one go over 8 data lines, or you can split into 2 data transmissions of 4 bits each.
- For 4-bit data transmission, you only need 4 data lines connected between LCD and MCU
- For 4-bit data transmission, you must use data lines D4 ,D5, D6,D7



DB4 to DB7	4	I/O	MPU	Four high order bidirectional tristate data bus pins. Used for data transfer and receive between the MPU and the HD44780U. DB7 can be used as a busy flag.
DB0 to DB3	4	I/O	MPU	Four low order bidirectional tristate data bus pins. Used for data transfer and receive between the MPU and the HD44780U. These pins are not used during 4-bit operation.

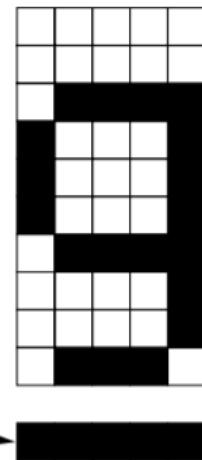


HD44780U

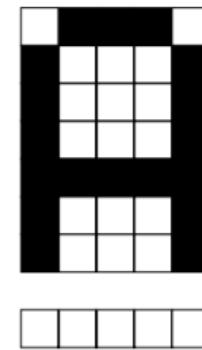


5 × 8 dot
character font

Cursor display example



5 × 10 dot
character font

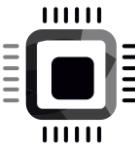


Alternating display



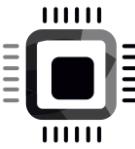
Blink display example





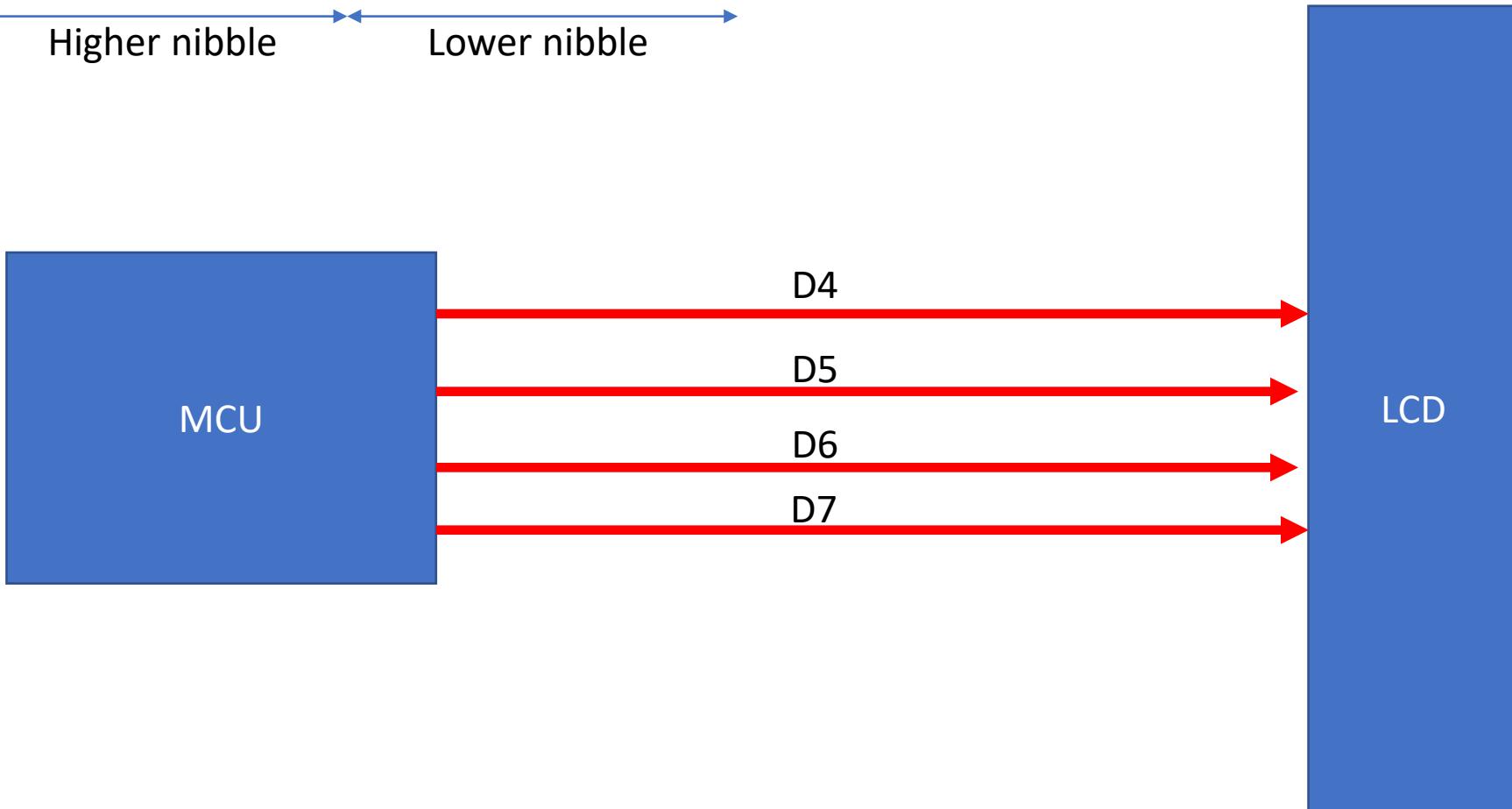
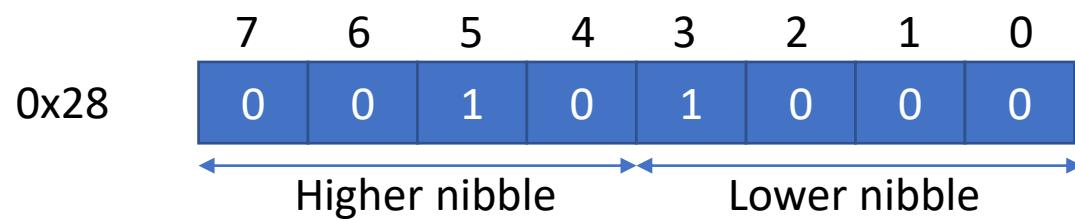
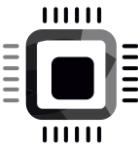
Sending a command

- 1.Create the command code
- 2.Make RS pin low
- 3.Make RnW pin low
- 4.First send higher nibble(4-bits) of the command code to data lines
- 5.Make LCD enable pin high to low (when LCD detects high to low transition on enable pin it reads the data from the data lines)
- 6.Next send the lower nibble of the command code to data lines
- 7.Make LCD enable pin high to low (when LCD detects high to low transition on enable pin it reads the data from the data lines)
8. Wait for the instruction execution time before sending the next command or confirm the LCD is not busy by reading the busy flag status on D7 pin .

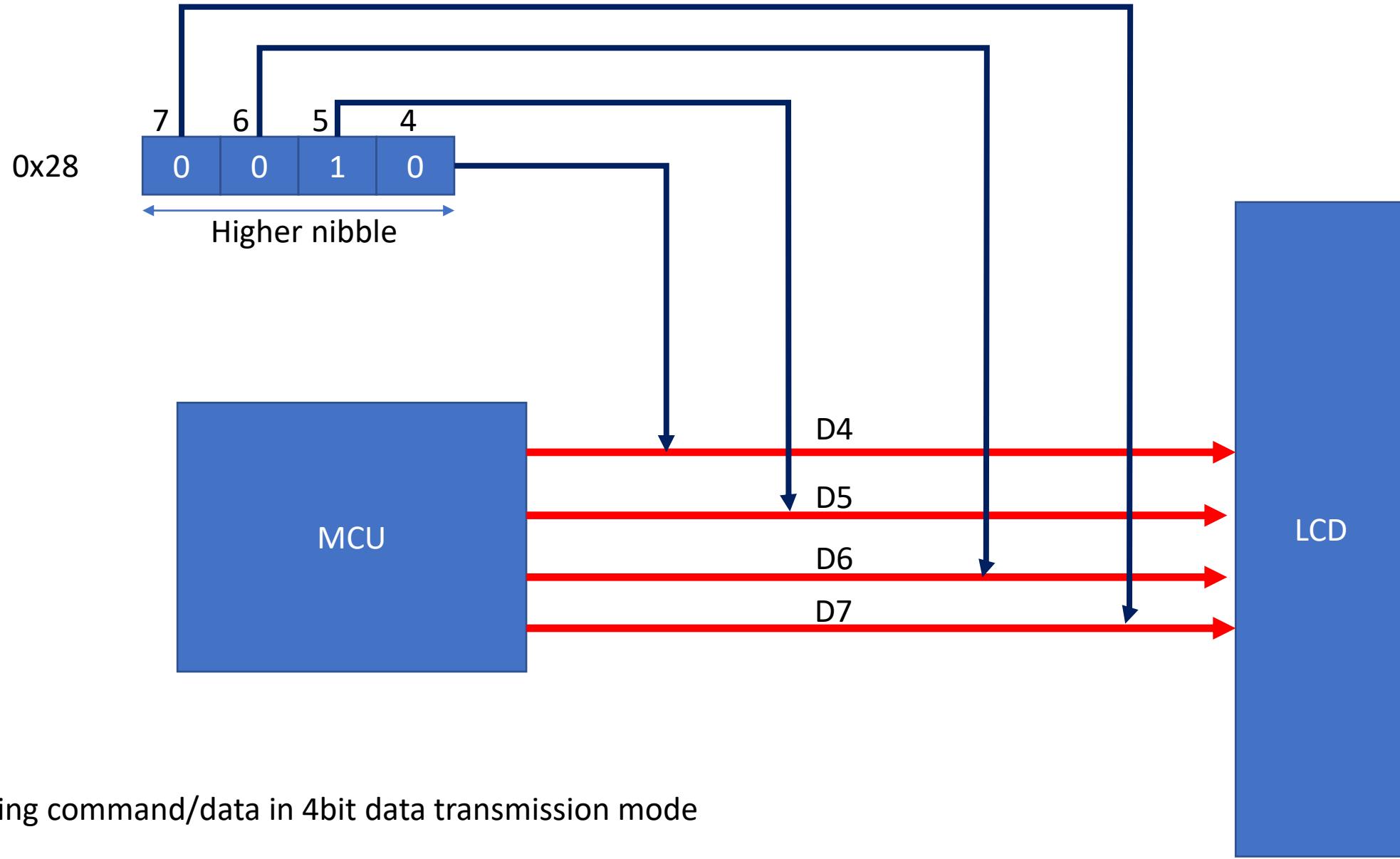
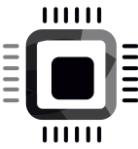


Sending a data byte

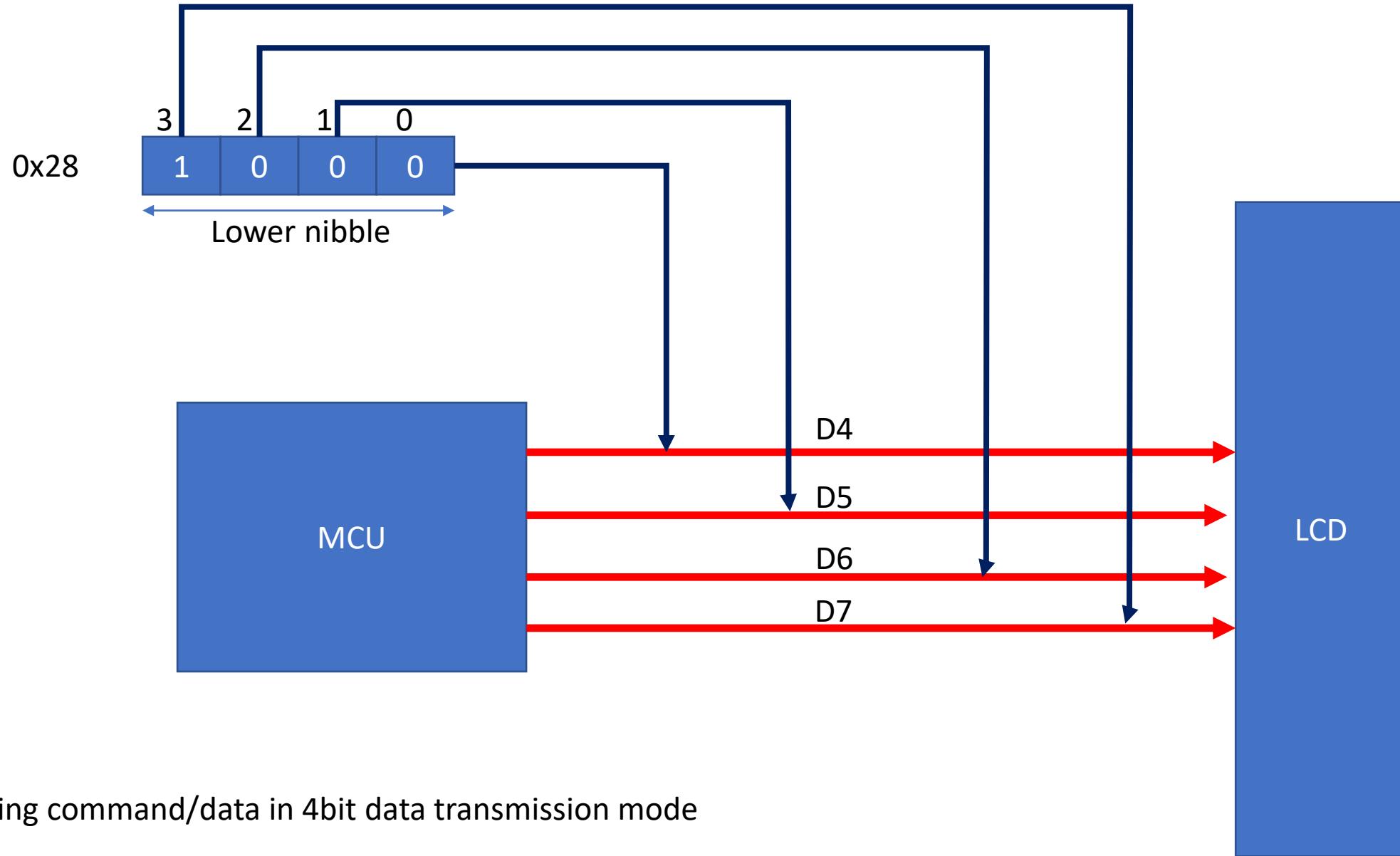
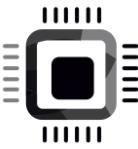
1. Make RS high
2. Make RnW low
3. First send higher nibble of the data to data lines
4. Make LCD enable pin high to low (when LCD detects high to low transition on enable pin it reads the data from the data lines)
5. Next send the lower nibble of the data to data lines
6. Make LCD enable pin high to low (when LCD detects high to low transition on enable pin it reads the data from the data lines)



Sending command/data in 4bit data transmission mode



Sending command/data in 4bit data transmission mode



Sending command/data in 4bit data transmission mode

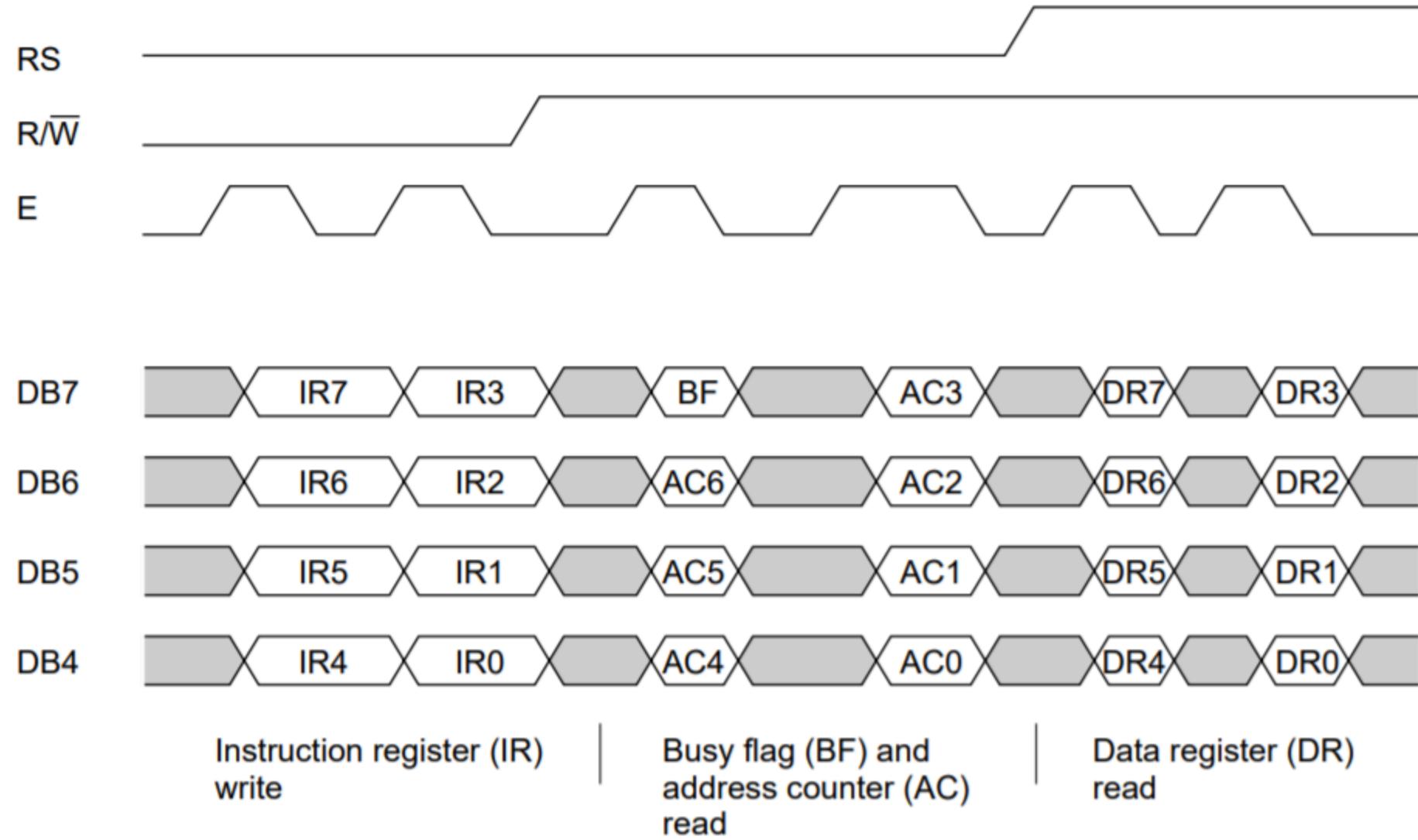
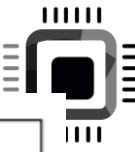
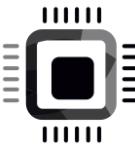


Figure 9 4-Bit Transfer Example



Interface to MPUs

- Interfacing to an 8-bit MPU

See Figure 16 for an example of using a I/O port (for a single-chip microcomputer) as an interface device.

In this example, P30 to P37 are connected to the data bus DB0 to DB7, and P75 to P77 are connected to E, R/W, and RS, respectively.

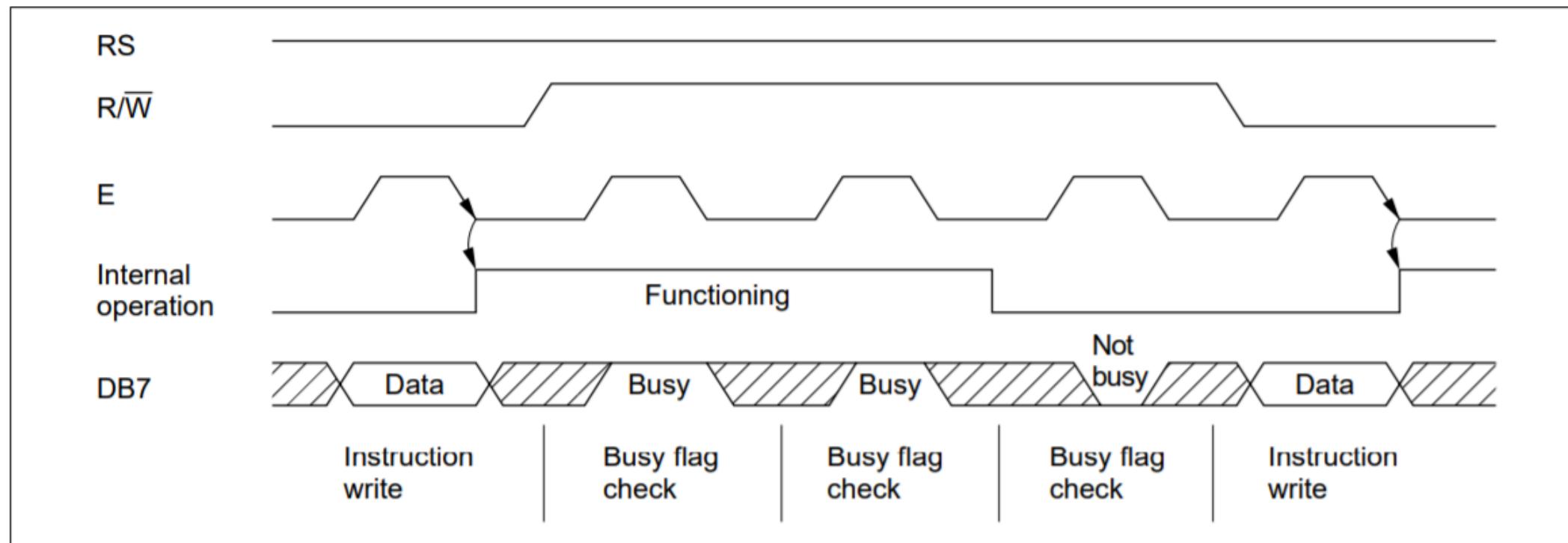
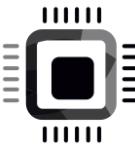


Figure 15 Example of Busy Flag Check Timing Sequence



- Interfacing to a 4-bit MPU

The HD44780U can be connected to the I/O port of a 4-bit MPU. If the I/O port has enough bits, 8-bit data can be transferred. Otherwise, one data transfer must be made in two operations for 4-bit data. In this case, the timing sequence becomes somewhat complex. (See Figure 17.)

See Figure 18 for an interface example to the HMCS4019R.

Note that two cycles are needed for the busy flag check as well as for the data transfer. The 4-bit operation is selected by the program.

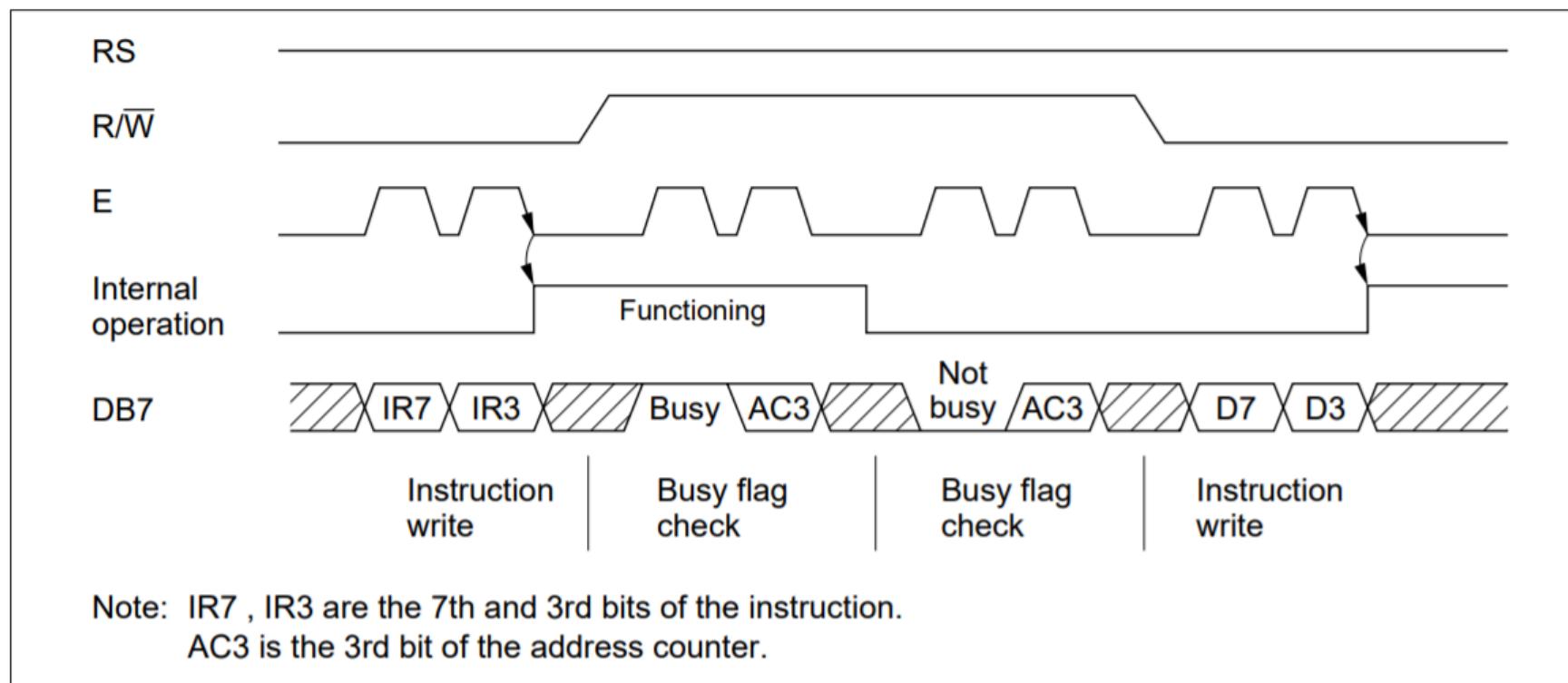


Figure 17 Example of 4-Bit Data Transfer Timing Sequence