The Journey Begins, is it end up ? (Founder Mr. M. Naresh Sir)

Kernel and Embedded Linux and Protocol

Download and Build procedure

Ubuntu Machines

sudo apt-get install libncurses5-dev libssl-dev git exuberant-ctags cscope

Stable Kernel source

git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git

Kernel Build Procedure

1) A build system integrated into kernel source which automatics entire build process.

2) Build System can be triggered through set of interface commands which are initiated through a build tool **make**

Step 1: Choose kernel build configuration

make menuconfig (Default it assumes x86) // This is for light weight selective dialogue box

make ARCH=arm menuconfig (We are giving specific architecture name)

[ ] ← Only static selection (press space bar, you will see * and None)
< > ← Static and Dynamic selection (press space bar, you will see *, M and None)

After selection it will produce .config file
In .config file

=m ← Module
=y ← Yes static

.so .ko both are same dynamic shared objects. Difference is user space and kernel space

Initiate, compile and build

make -j<count of cpu core > // For fastness of making build
We are using -j flag to run on multiple core

make -j4 // It will run on 4 cpu cores

Observation Note: Configuration are two types 1) Interactive 2) non-interactive

Step:2 make

1. Open .config

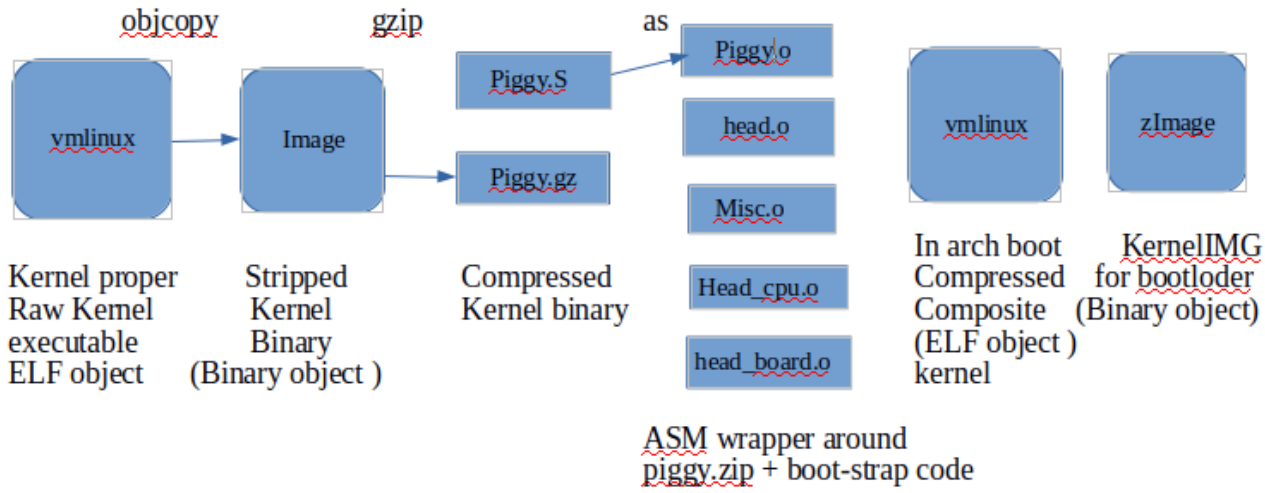
2. Identify source files for each service selected by stepping into appropriate approach

3. Compile identified service code into relocatables using Kbuild script

4. Integrate all relocatables of the options chosen statically to generate ELF (raw), Kernel Image (vmlinux Image)

5. Invoke target arch specific build script (arch Makefile) to generate loadable kernel image.
Arch Makefile script does the following
- Appending arch specific boot-strap
- Enable CPU's MMU
- Setup page tables
- Invokes start function of Kernel proper
- Appends boot loader specific header defining image type load and entry addresses

6. Build module object files (.ko) from relocatable of "=m" entries.
Entries of .config file is arch/arm/boot/zImage ← loadable Image



Kernel Bootstrap

Step:3 Move module images onto root file system (RFS)

make modules-install

Move module onto root file systems kernel header/in-tree modules folder
(/*lib*/modules/kernel_version_string/)

Step:4 Install kernel Image and configure boot loader

make install

1. Copies loadable kernel image and other relevant files into /boot folder of root file system.

2. Updates boot loader with menu entry for new kernel

Distribution provided kernel packages include the following

1) loadable kernel image

2) Modules matching kernel version

3) Config file applied to build the kernel

4) System map_file containing kernel symbol table

5) Initial RAM disk Image initrd.img-4.4.0-1.4.4.-generic

6) Kernel ABI file abi-4.4.0-131-generic

Integrating New Services into kernel source tree

Step 1: Move source file into appropriate branch of kernel source tree

Step 2: Add new listing into kernel configuration menu

- Append new config entry describing new service and its dependencies into Kconfig file of the local branch

[ ] - bool " "
< > - tristate " "
--- > - Sub menu is there

Step 3: Modify Kbuild scripts to compile and build new services as per chosen method
-edit local Kbuild Makefile to have compile target for new source.

In Makefile

obj-$(CONFIG_USB_TESTDRV) += usbdrv.o
| __ If M -Module
Y – Kernel

Linux Kernel Build System

Major component of build

Makefile - The top Makefile Contains target command which trigger build system scripts

.config - The kernel configuration file

arch/$(ARCH)/Makefile – The arch Makefile contains target commands and script used to generate loadable kernel image

scripts/Makefile.* - (Common rules etc. for all Kbuild Makefiles)

Kbuild Makefiles - Local Makefiles located in sub folders of source tree (There are used run a kernel build related scripts)

Kconfig - Kernel config menu scripts.

Build Summary

* The top Makefile reads the .config file, which comes from the kernel configuration process.

* The top Makefile is responsible for building two major components products
vmlinux (the resident / row kernel image) and modules (.ko files).

* It builds these goals by recursively descending into the sub directories of the kernel source tree.

* The list of sub directories visits depends upon the kernel configuration.

* Each sub directory has a build Makefile which carries out the commands passed down from above.

* The Kbuild Makefile uses information from the .config file to construct various file lists used by kbuild to build any built-in (statically linked) or modular targets (module.ko).

* scripts/Makefile.* contains all the definitions/ rules that are used to execute build commands present in Kbuild Makefiles
    For Example: Commands to run for obj-y or obj-m targets.

* The top Makefile textually includes an arch Makefile with the same arch/$(ARCH) Makefile

* The arch Makefile supplies architecture specific information to create loadable kernel Image.

Creating out of tree modules

* Modules are ELF object files through which kernel code and data segments can be extended at run time.

* Depending on location of source file used to generate a module image, a module is it a common entry in-out of tree.

```
        Obj-m := myhello.o

# variable/macro to hold kernel-headers/ kernel source directory path
        KDIR=/lib/modules/$(shell uname -r)/build

this command rules Kbuild scripts of kernel header/ source folder specified
all:    $(MAKE) -c $(KDIR) M=$(PWD) modules
```

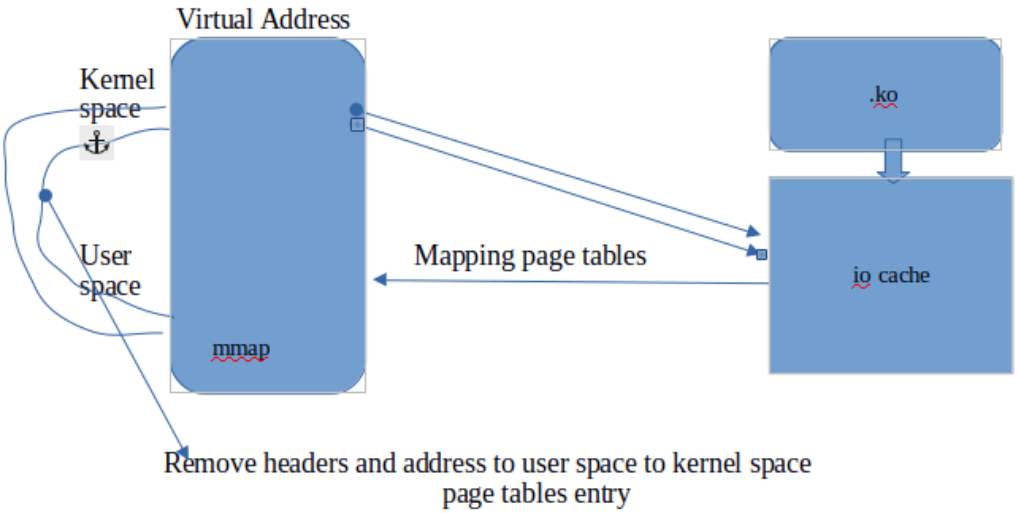
Tools found in modutils package invoke system calls of Kmod subsystem to Carrie out linkage and unlinkage of kernel modules.

init_module, finit_module - **load a kernel module**

int finit_module(int fd, const char *param_values, int flags);

delete_module - **unload a kernel module**

int delete_module(const char *name, int flags);



Mapping after insert modules

* Functions of a kernel module do not execute or run when linked into kernel address space.

* They only become part of kernels code segment and or executed only when they are invoked in the context of an application program.

1) Module source should include only kernel headers
    #include<linux/module.h> // Kmod subsystem
    #include<linux/init.h> // init sequence

2) Modules can have initializer and exit routines.

```
int mod_init(void)
{
….
}
void mod_exit(void)
{
…..
}
module_init(mod_init);
module_exit(mod_exit);
```

3) Initializer is invoked when module is linked into kernels address space. Initilaizer routines is called in the context of finit-module system call.

4) Initializer routine is called in the context of finit_module system call.

5) module deployment is aborted if init function returns -ve number.

6) Module sources can contain appropriate co comment macros carrying metadata.

MODULE_AUTHOR(“.org“);
MODULE_DESCRIPTION(“lcd on ”);
MODULE_LICENSE(“GPL“);

7) comment macros retaining required code information comments into module binary file.

modinfo myhello.ko

usage of comment macros is optional, excepting module-license (mandatory)

License to chose depends on global kernel functions invoked by module.

If module doesn’t invoke any GPL licensed functions of the kernel then it can be declared license as any one of the following.

MIT, PSD, GPL, or Proprietary (I am not give source)

8) Module body is a piece of code and data being attached to kernels address space

9) Module body can be composed any number of functions are data elements.

10) Functions and data declared in a module are by default local in scope (there have not accessible from rest of kernel code)

11) a module can selectively export any of it’s symbols into global name space using the either of following macros.

EXPORT_SYMBOL(sym);
EXPORT_SYMBOL_GPL(sym);
EXPORT_SYMBOL_GPL(veda_counter);

symbol exports makes it possible to implement a kernel service through a set of kernel modules

12) Kernel build system generates a header for every module, this header is composed of following elements.

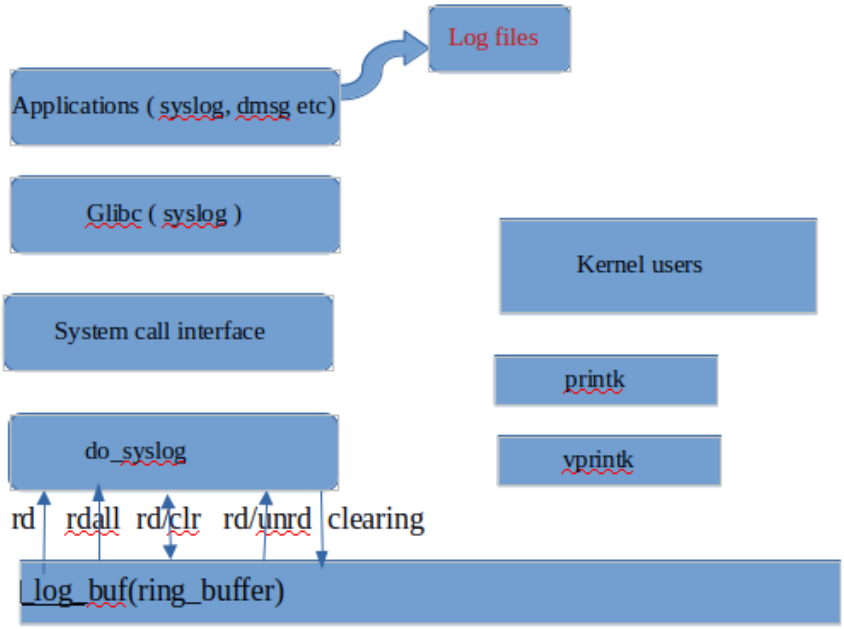
1) Version number of the kernel for which module is being build
2) An instant of structure module which represents module in memory
3) A list of kernel symbols which the module code referring to a list of other modules with the current modules depends on.

Kernel Print Statements

Kernel print statements are used for any of the following purposes
1) Error reporting
2) Test
3) Debug Messages

* printk is a kernel automate printf which is used to print any of the messages

* Since kernel is non interactive print statements of the kernel code, don't show up on terminals
(Terminal is basically for I/p, assigned to applications for user interaction)



* do_syslog is kernel function which manages log buffer, user mode applications can invoke functions of do_syslog through system call interface.

* glibc provide on api to invoke the appropriate system call.

int klogctl(int type, char *bufp, int len);

* To facilitate categorization of print statements at the application layer each print message must begin with on header, which describe type of the message kernel defines for link valid headers which always must pretend the message.

```
KERN_EMERG      // system is unusable
KERN_ALERT      // action must be taken immediately
KERN_CRIT       // critical conditions
KERN_ERR        // error conditions
```

printk(“header” “message”);

testing always user INFO (Debugging) all the rest are reporting.

printk(KERN_INFO “ %s value if counter %d\n”);

* printk doesn’t syntactically enforce message header argument, to ensure that print statements contain user assigned header kernel provides a set of wrapper macros which are to be used in place of printk. Macros defined in printk.h

pr_emerg
pr_alert
pr_crit
pr_err
pr_warning
pr_notice
pr_info
pr_cont
pr_log

LINUX DRIVER ARCHITECTURE

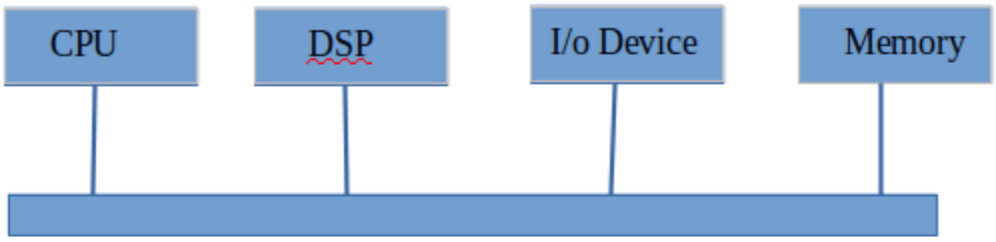
Bus Architecture

* Buses are the simplest and most widely used interconnects between components of a computing system.

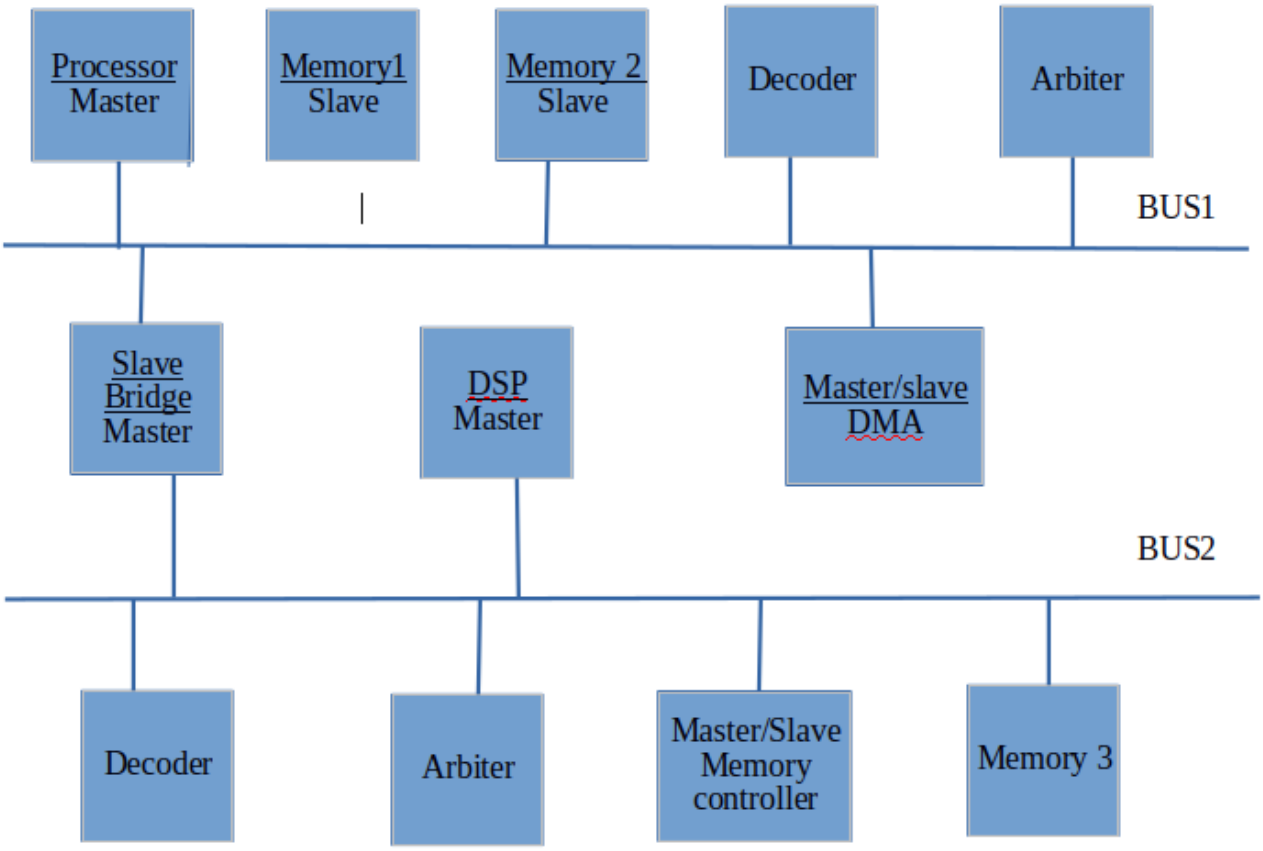
BUS : -

* Collection of signals wires, to which one or more connected, which need to communicate data with each other are connected.

* Only one component can transfer data on the shared bus at any given time.



BUS arbitrary



BUS based Communication system

IP Components :-

* Devices which can initiate transaction or claim a transaction are returned to as IP components

* Master components are connected to bus with the bus, through a set of control beam which are connectively called Master ports.

* Master components can initiate first bus transaction (read/write)

* Components which are interfaced through a slave port are referred to as slaves and these devices can only respond to transaction initiated by master

* Some components can have both master and slave ports which means that they can at a both master and slaves. These components are master/slave hybrid components.

Logic Components :-

Some bus based communication architectures also consists of logic components such as decoders, arbiters, bridges.

Decoder :-

* A Decoder is a logic component that decodes destination address of a data transfer initiated by a master and selects an appropriate slave to receive the data/control signals.

* Decoder can either be a separate logic component (Centralized) or Integrated into a component interface (distributed)

Arbiter :-

* An arbiter is a logic component that determines which master to grant access to the bus when multiple masters attempt to raise bus transactions simultaneously.

* Arbiter's are programmed with some form of a priority scheme to ensure that critical data transfers are not delayed.

* Arbiter's can be (centralized) separate logic component or integrated into a component interface (Distributed).

Bridge :-

* A bridges is a logic component that is used to interconnect between two buses.

* Bridge can be simple implementation if it connects two buses of a same type with identical protocol and clock frequencies.

* If two buses are have different protocols or clock frequencies some form of protocol or frequency conversion is required in the bridge, which adds to its complexity.

* A bridge connects to a bus using a master or a slave port. The type of port used to connect to a bus depends on the direction of the transfers passing through it.

BUS Signals :-

* Address Signals are used to transmit the address of destination for a data transfer on the bus. Number of signals used to transmit the address is typically a power of 2 (16, 32, or 64) and referred to as the address bus width.

* Most bus specifications use single shared address bus for reads and writes.

* It is possible to have separate address buses for read and write data transfers.

* Data signals are used to transmit data values to their destination addresses. The data signals are collectively referred to as the data bus.

* The typical number of signals in a data bus is 16, 32, 64, 128, 256, 512 and 1024 signals (called data bus width)

* Data buses can either be implemented as a single shared bus for both reads and write, or separate data buses for reads and writes.

* It is also possible to combine the data and address buses by multiplexing than over a single set of wires

Control BUS :-

* Control bus carries commands from the master and returns status signals from the slaves.

* Control signals are used to send information about the data transfer and or bus protocol specific.

* 'Request' and 'acknowledge' signals are the most common control signals, which transmit a data transfer request from a master, and the acknowledgment for a request or data received.

* Data size (or byte enable) control signals indicate the size of data being transmitted on the bus.

* Slaves can signal on error condition to the master over special status control signals. If data cannot be read or written at the slave.

* Control signals can also transmit information about the data being transmitted such as whether data is bufferable, write-through or write -back.

Port Mapping & Memory Mapping

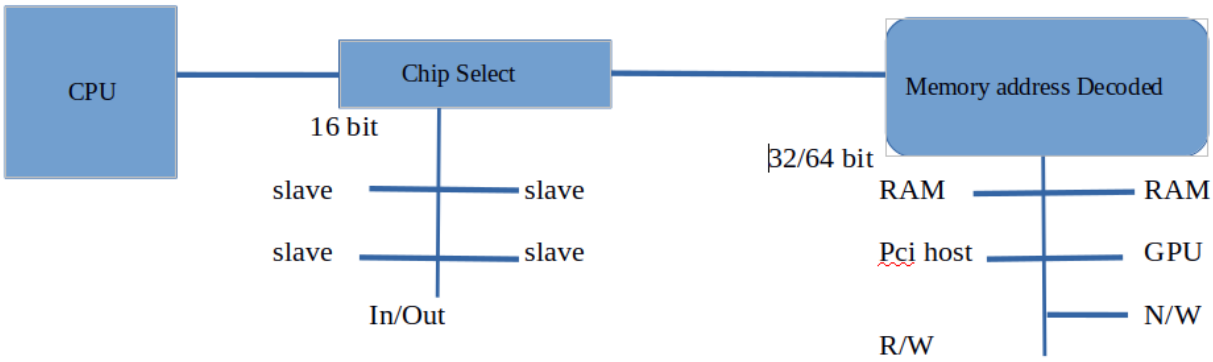
1) x86 architecture implement two address decoder on the system bus

Port i/o decoder: which can decode 16bit address called port i/o address space.

MMIO decoder: which can decode addresses of the primary memory addresses (32bit or 16bit)

2) Devices interfaced through port i/o bus are referred to as Port Mapped

3) Devices interfaces through MMIO device are referred to as memory mapped.



x86 instructions sets supports In/Out family of instructions for drivers operations to port i/o bus and Read / Write families of instruction for driving operations to MMIO operations.

cat /proc/ioports (16 bit addressing) → port is decoder no longer resigned that is fixed addresses even if device present or not

cat /proc/iomem (32 bit / 64 bit addressing) → /proc/ioports & iomem present port i/o and MMIO decoder maps

4) Since address space is finite, a system will not able to support increase Number of devices through port or Memory mapping interfaces.

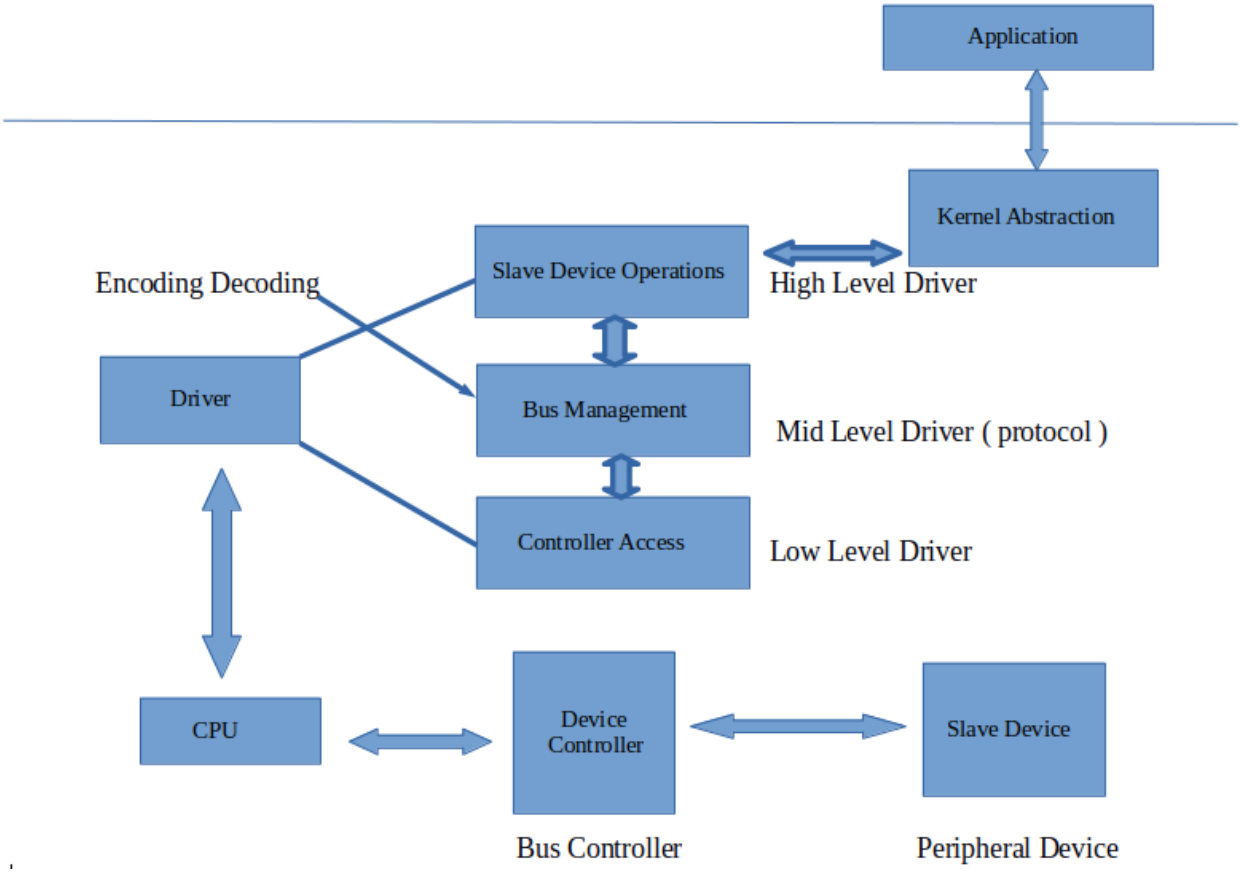
5) Chip select interface and peripheral bus standards are work around to support increase number of devices without a head of assigning dedicated address spaces.

DRIVER STACK

Drivers are port mapped and memory mapped devices can be programmed to access device registers to initiate specific operation

For devices which are ported on peripheral bus, drivers can not directly access device address space.

They need to be programmed to communicate with device through appropriate bus controller.



1) **Low Level Driver (Bus controller / Master chip set specific)**

This driver is controller device specific and is normally initialized during early stages of kernel boot.

This driver must perform following operations.

- Bus-scan (for hot plug buses).

- Assign Identification Numbers to each bus and device found.

- Create stack specific bus structure to represent each physical bus found (struct pci_bus, struct usb_bus,…)

- Identify slave devices found during bus scan and enumerate driver-stack specific device structure (details of device configuration ex: struct pci_dev, struct usb_device, i2_client, …)

- Provide packet transmission and reception routine for send/receive data control packets from host to slave.

- Handle interrupt events raised by host/master bus-controller.

- Register its operations with appropriate driver/bus manager easier.

2) **MID Level Driver (core) (Bus Manager)**

Mid level drivers (Bus Manager) These are kernel provided peripheral drivers and are initialized during every boot. This layer provides following facilities.

* Interface for low level drivers to register.

* Interface for high level drivers to register.
* Maintain driver list with information of registered drivers.

* Match drivers with devices found and initialize high level drivers.

* enumerate device/bus/driver information to kernel’s hot-plug file system (sysfs).

* Provide ready to use function interfaces for high level driver to initiate i/o on slave devices.

* Provides routines to handle bus receptions.

Example: USB-core, Pci-core, I2c-core, Spi-core

3) **High Level Drivers (peripheral specific)**

* These drivers must register with bus manager layer by specifying compatible devices they can handle.

* These are initialized by bus-manager when compatible device specified is found.

* These drivers are programmed to carry out following

- Register with appropriate devices/driver manager.

- Register with appropriate kernel's driver abstraction layer.

- Initiate relevant operations on the target device as per callers request (through routines of selected kernel driver model).

// Bus driver and kernel driver mid level need to initialize high level.

High Level driver binding with bus manager

Example 1:- USB peripheral drivers

```
#define VENDOR 0x0781
#define DEVICE 0x5589

static int test_probe( struct usb_interface *intf, const struct usb_device_id *id )
{


}



/* structure to specify device identification details

static struct usb_device_id dev_table[]={
{ USB_DEVICE(VENDOR,DEVICE); } /* device */
{ } /* Null terminator required */

MODULE_DEVICE_TABLE(usb , dev_table );

/* structure to register with usb-core */

static struct usb_driver my_usb_driver ={
.name = "usb-test",
.probe=test-probe, //driver init call
.dis_connect=test_disconnect // driver's exit call
.idtable=dev_table //table of devices we are looking for
};
```

/* Helper macro to replace module init / exit to get driver object registered with usb core */

module_usb_driver(my_usb_driver);

Example 2:-

Bus/device/driver

USB High level driver registering with class id

```
Static const struct usb_device_id usb_mouse_id_table[]={
{ USB_INTERFACE_INFO(USB_INTERFACE_CLASS_HIS,USB_INTERFACE_SUBCLASS_BO
OT, USB_INTERFACE_PROTOCOL_MOUSE)};
{ } // terminating string

};

MODULE_DEVICE_TABLE(usb,usb_mouse_id_table); //usb sub system

static struct usb_driver usb_mouse_driver ={
.name = “usb mouse”,
.probe = usb_mouse_probe,
.disconnect = usb_mouse_disconnect,
.id_table = usb_mouse_id_table,
};
module_usb_driver(usb_module_driver);
```

Example 3:-

Pci peripheral driver registering with pci core.

```
static  const struct pci_device_id cp_pci_tbl[]={

{ PCI_DEVICE ( PCI_VENDOR_ID_TTECH,PCI_DEVICE_ID_TTTECH_MC322),},
{}

};

MODULE_DEVICE_TABLE(pci,cp_pci_tbl);

static struct pci_driver cp_driver={
.name = DRV_NAME,
.id_table = cp_pci_tbl,
.probe = cp_init_one,
.remove = cp_remove_one,

#ifdef CONFIG_PM
.resume = cp_resume,
.suspend = cp_suspend,
#endif
};
```

module_pci_driver(cp_driver); // Internally call module Init exit function related pci-driver function also

drivers/iio/gyro/ spi example also bmg160_spi.c
vim st_gyro_i2c.c

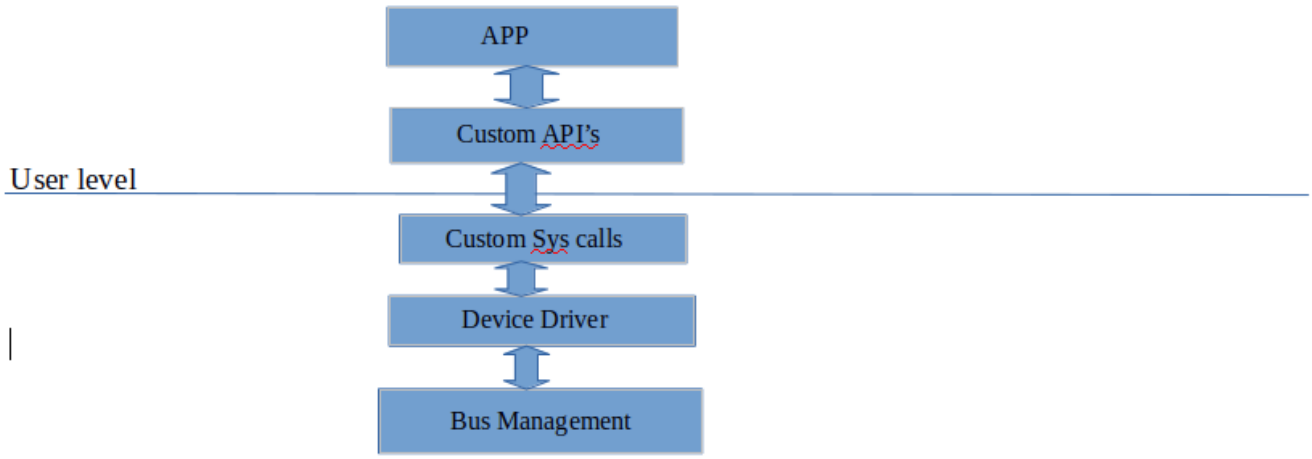
Kernel Abstraction

* A driver can be presented to user space through many of the following methods

1) Custom System Call Interface

- Extend kernel code with new system calls which are programmed to invoke driver operation.

- Provide an api for the application to invoke the system call.



Custom System call interface

Benefits :-

This model allow driver and applications implementation to be vendor specific due to custom api's third party applications can't initiate driver operations

Limitations:-

This model requires new system call to be added to kernel 3, which is only possible through appropriate kernel patches

1) Patch Management for each kernel version involves dedicated and active maintenance

2) Enhancement and bug fixes to driver code might requires changes to system call and api layers

2) Standard kernel abstraction

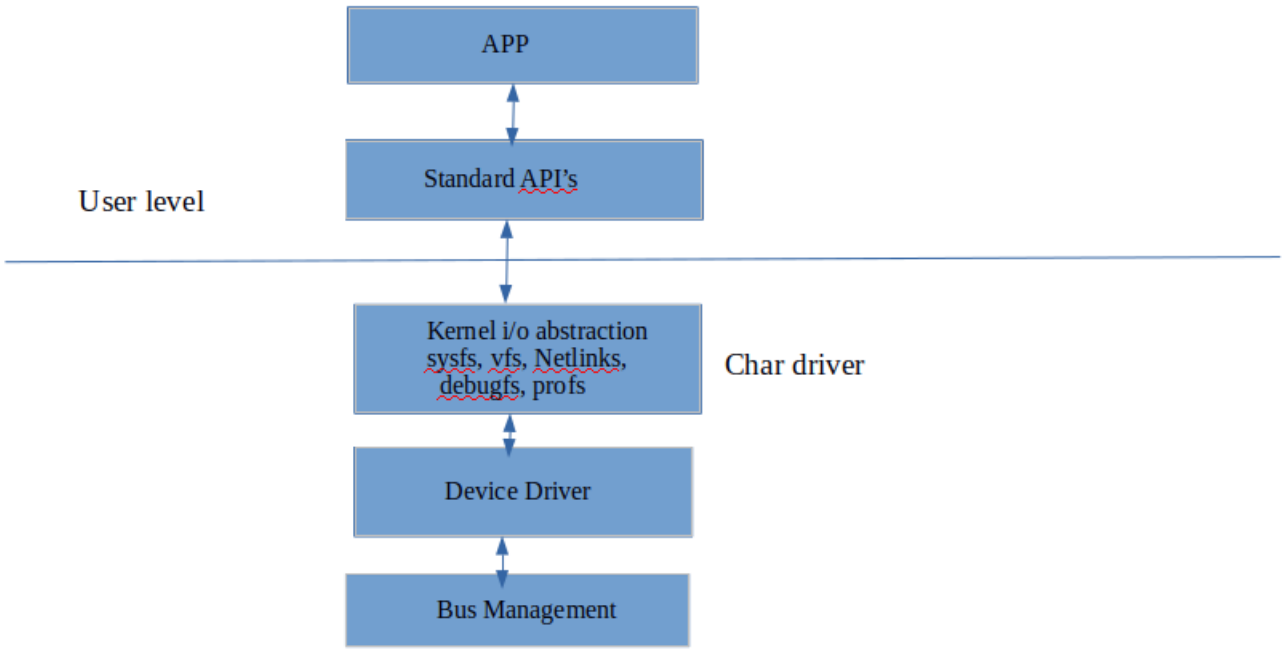
***** In this method requires a driver to bind with existing kernel abstraction (register) layer.

Example :- VFS, sysfs, Debugfs, Netlink sockets, socket api and procfs.

* Drivers program to user this method can be accessed synchronously from user space.

Benefits:- This method simplifies application and driver interaction and is easier to maintain.

Limitation:- Driver are to be carefully design to expose all operations of the device through a limited existing system call interface.



3) Kernel Device Management subsystem

Kernel provides various subsystems which are capable of managing a specific class of devices.

Kernel Device Management system.

Examples:-

Storage subsystem – To Manage persistent storage devices

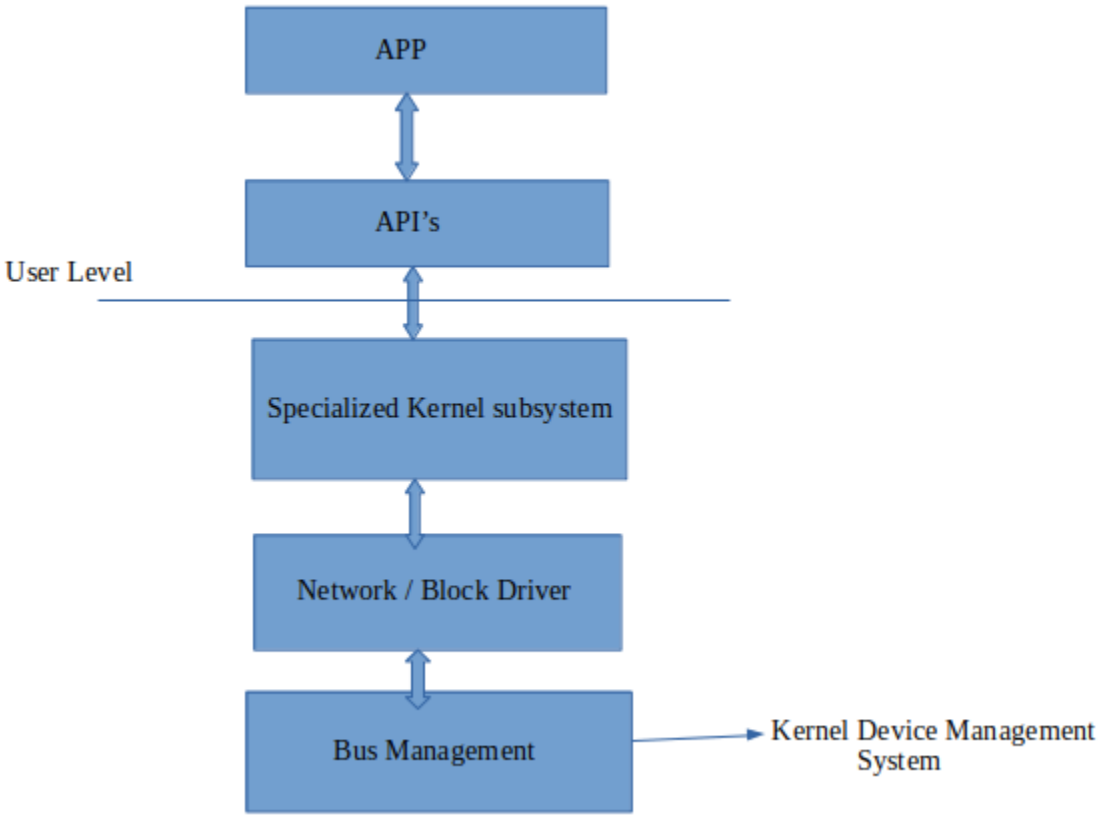
MTD (Memory technology devices) - To manage flash storage devices NOR / NAND

Net device subsystem – To manage network communication hardware (WIFI, ETHERNET)

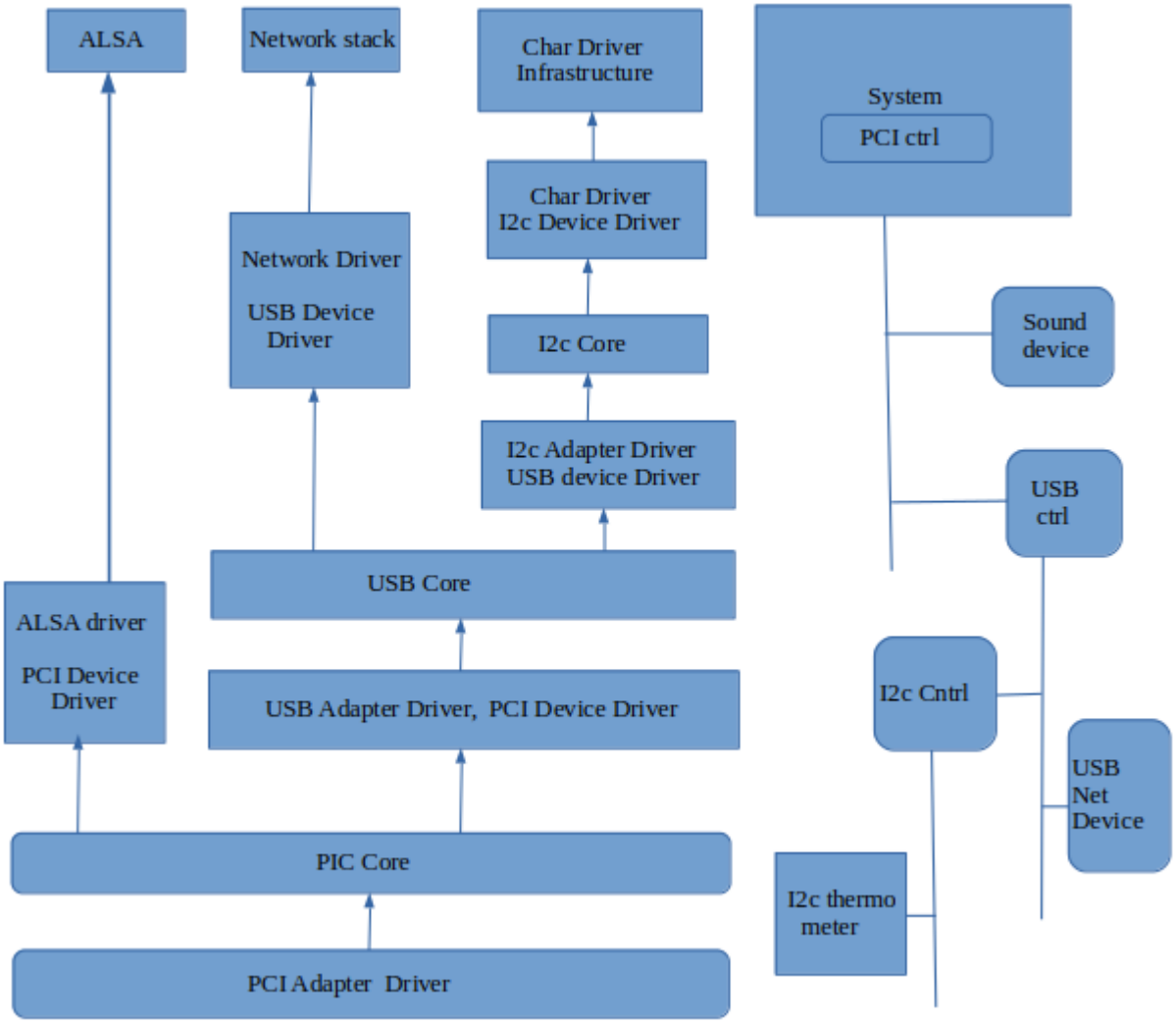
Input subsystem – To manage input devices (touch screen)

Industrial i/o – To manage ADC's and sensor , etc.

* Drivers implemented to use method don't communicate with application directly (All Device operation initiated from the user mode / asynchronous)



Driver Stack



Platform Devices :-

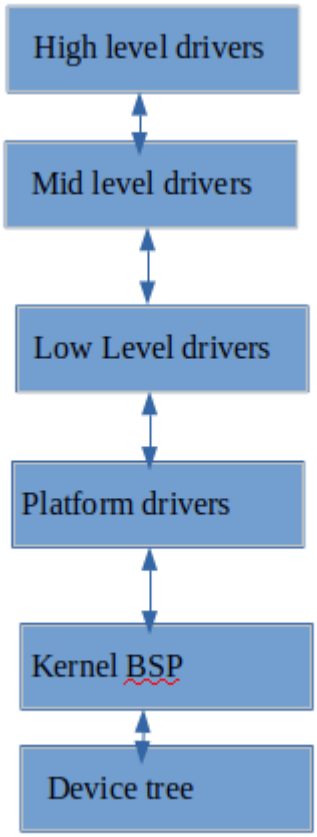
Devices which are directly memory mapped are referred to as platform devices. (Devices found on Soc platform like Device controller, Bus controller, and port devices / bus interfaces) fall into this category.

* Kernel provides a layer called platform core to manage drivers of platform devices.

* Kernel BSP code enumerates data structures which represent platform devices found on soc, this list is managed by platform core,

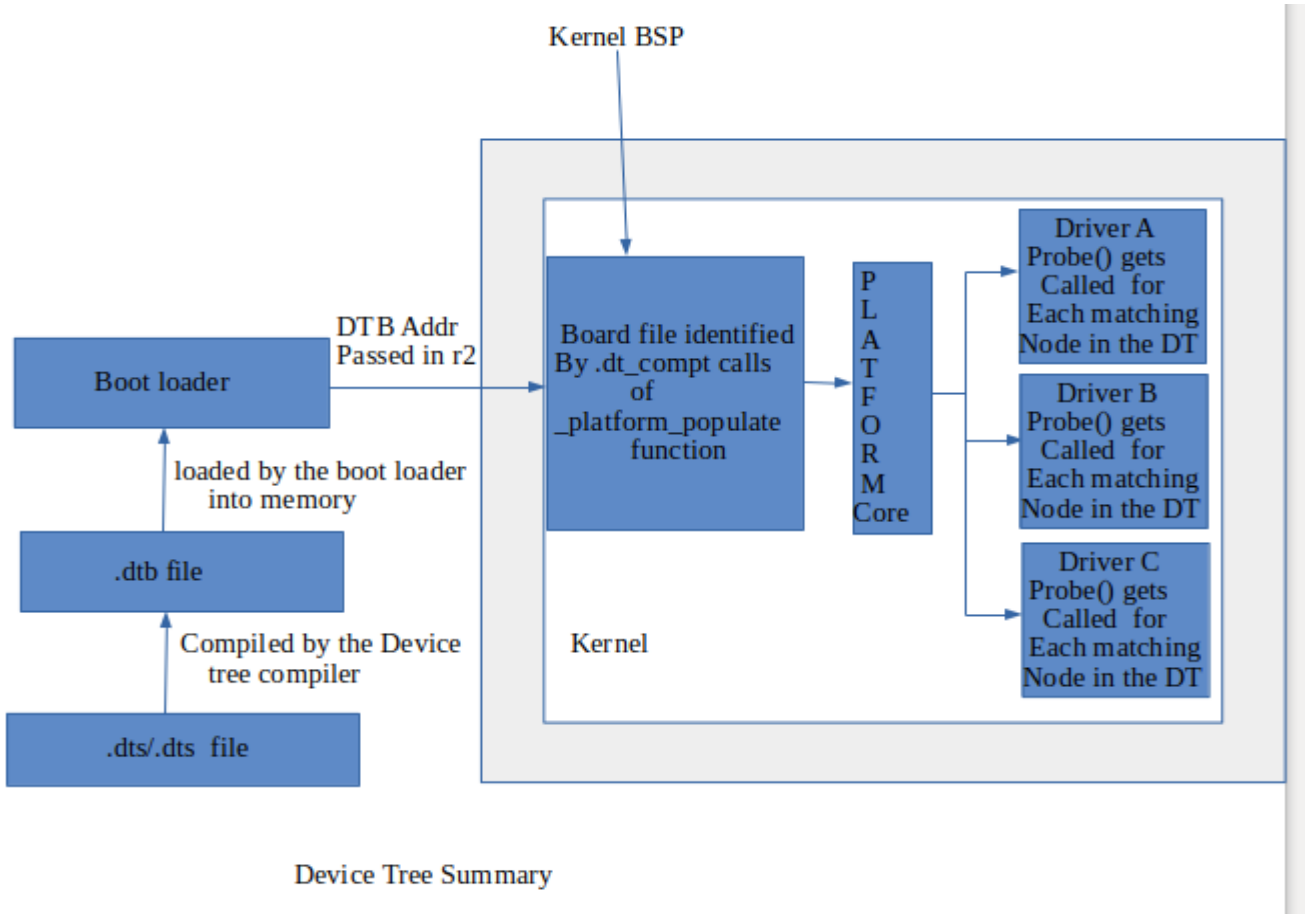
```
struct platform device {
        const char *name;
        u32      id;
        struct device dev;
        u32 num_resources;
        struct resource *resource;
};
```

* All drivers which are programmed to handle platform devices most registered with platform core.



Name	Device tree entry	Data structure	Managed by
GPIO controller	Yes	Platform device	Platform core
I2c controller	Yes	Platform device	Platform core
SPI controller	Yes	Platform device	Platform core
USB controller	Yes	Platform device	Platform core
PCI controller	Yes	Platform device	Platform core
Ethernet controller	Yes	Platform device	Platform core
Bluetooth controller	Yes	Platform device	Platform core
WIFI controller	Yes	Platform device	Platform core
Audio controller	Yes	Platform device	Platform core
Sensor	Yes	i2c_client	I2c core
Flash	Yes	spi_device	Spi core
USB storage	No	usb_interface	USB core
PCI video	No	pci_dev	Pci core

Device Tree Summary



Device Tree Summary

Char driver interface

* Char driver infrastructure is composed of bunch of logical file system through which a kernel service can interact with applications.

* Vfs, sysfs are popular choice among logical file systems.

* Char driver interface invokes kernel services synchronously.

Interfacing driver through VFS

* VFS is a file system abstraction layer which is implemented with an objective of providing common file api to user mode applications.

* Vfs api's perform operations on file descriptor which are applications specific handles to the file inode on which designs operation is to be performed.

* For a driver to be able to use vfs as an interface it must present itself as a file through a valid inode.

* Drivers can enumerate an inode structure into rootfs through a kernel service called devtmpfs.

* devtmpfs is a logical file system introduced as a helper service through which drivers can offload inode management.

* devtmpfs is mounted to dev of the rootfs

Step 1:- Create devtmpfs inode to represent driver / device.

mknod < filename > <inode type > < file id >

mknod /dev/vDev1 c 190 0

Step 2:- Define operations for the inode.

```
static int char_dev_open(struct inode *inode,struct file *file)
{

}
static int char_dev_release(struct inode *inode,struct file *file)
{

}
static int char_dev_write(struct inode *inode,const char user *buf, size_t buf, off_t *offset)
{
```

```
}

static struct file_operations char_dev_fops ={
.owner = THIS_MODULE,
.write = char_dev_write,
.open = char_dev_open,
.release = char_dev_release
};
```

Step 3:- Bind operations with the inode.

```
#define MAJORNO 190         //12bit
#define MINORNO 0           // 20bit
#define CHAR_DEV_NAME “vDev1”
static struct cdev *veda_cdev;
static dev_t mydev;
static int count =1;
```

/* register driver i/o abstraction layer */

1) Reserve Driver /device ID.

```
mydev = MKDEV(MAJORNO, MINORNO);
```

Within the kernel, the **dev_t** type who is defined in is used to hold device numbers (major/minor). **dev_t** is a 32-bit quantity with 12 bits set aside for the major number and 20 for the minor number.

```
register_chardev_region(mydev, count, CHAR_DEV_NAME);
```

2) Register file_operations with devtmpfs.

```
/* Allocate cdev instance */

veda_cdev = cdev_alloc();

/* Initialize cdev with fops object */
cdev_init(veda_cdev, &char_dev_fops);

/* Register cdev with vfs ( devtmpfs ) */
ret= cdev_add(veda_dev,mydev, count);
```

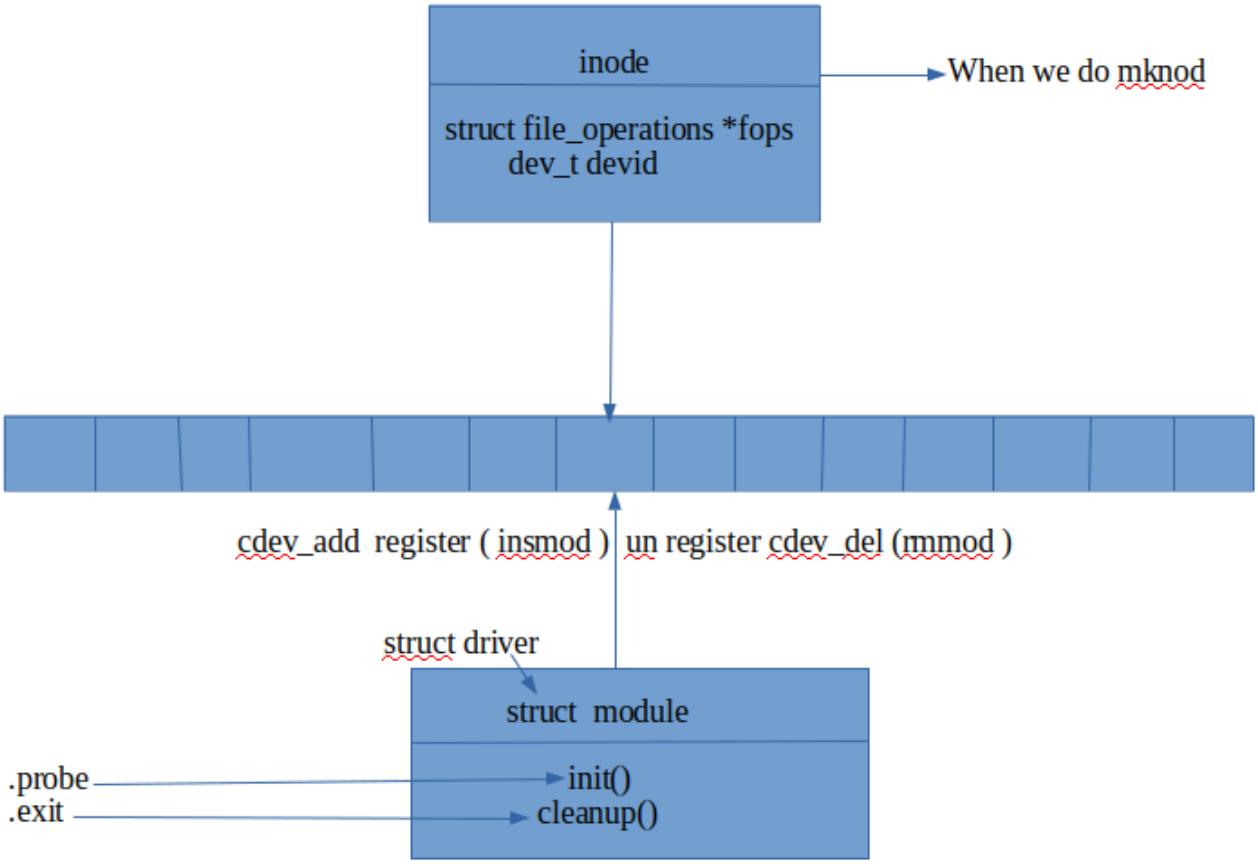
```
/* Deregister Driver */

static void __exit char_dev_exit(void);
{
 /* Remove Device */

cdev_del(veda_dev);

/* free major/ minor no's used */
un register_chrdev_region(mydev,count);

}
```

cat /proc/devices ← Register Driver Checking

Major Number :-

* Major Number is the location of the driver object with in drivers list maintained by vfs.

* Vfs identifies appropriate driver operations board to a device file through major number.



Major Number Array

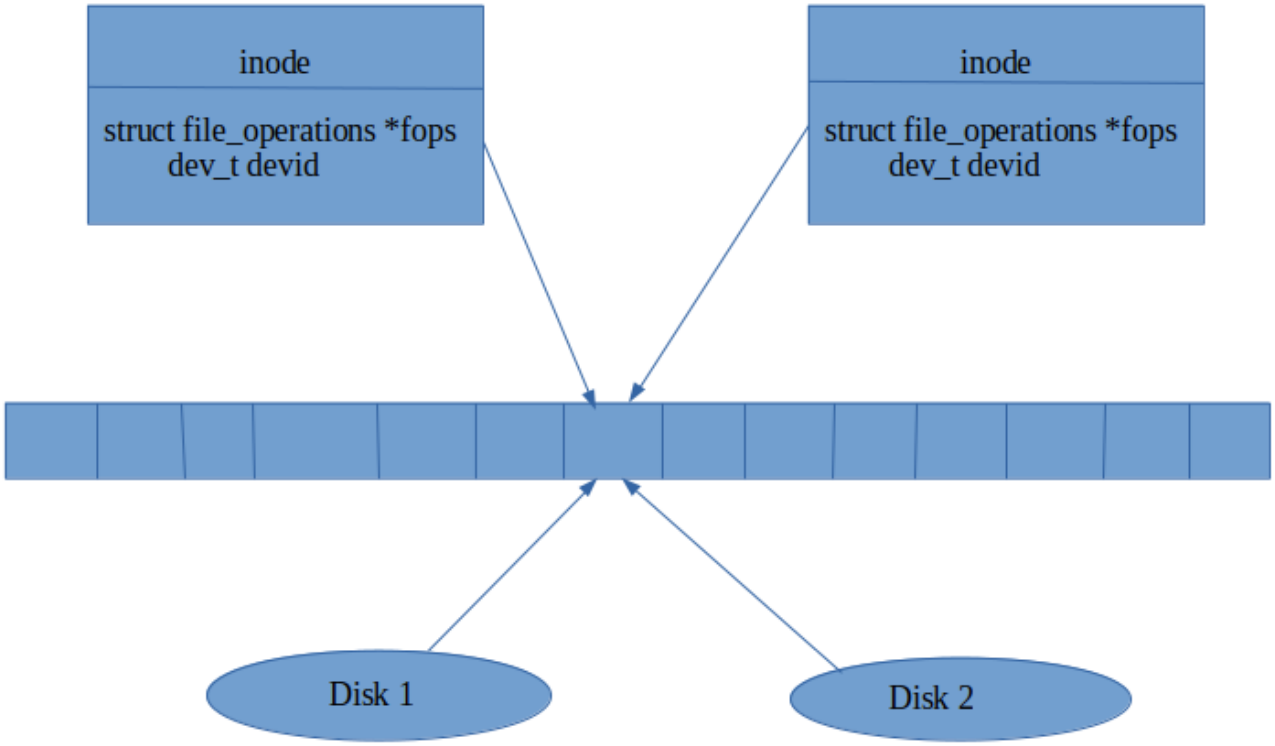
* 12 bit Major Number, so 4k (4096) drivers are possible.

* Major number can be reserved by drivers either statically or dynamically.

* The reserving a major number statically will impact driver portability.

Minor Number :-

* Minor numbers are used by driver code to identify the physical device on which the operation is to be performed.



Minor Number array

1048576 (2 ^ 20) 1MB devices we can handle.

dev_t my dev; int start minor = 0;

alloc_chrdev_region(&mydev, startminor, count, CHAR_DEV_NAME); // if u put start minor as 2 , count as 4, then allocated number is start from 2 to end with 6.

Device Node Automation :

Linux Hot-plug subsystem

Objectives :-

1. Deliver hardware events to application frame works (android, openwt, mobilinux, opencv, LEAF) and let apps in user mode respond to hot-plug hardware events.

2. Provide an interface for administrators to carry out power management ops.

3. Act as a centralized database for all hardware related information.

Components :-

1. Drivers core (unified device information structure)

- A structure that contains information about all buses, devices, and drivers currently found.

- Organizes all information in order as per hardware topology (tree).

- This structure is updated by linux driver (Bus Management layer during initialization and when hardware hot-plug events happen.

2. Sysfs Interface (mounted onto RFS /sys)

- Presents all information found in unified device data structure as files and folders preserving order and relations but various structures.

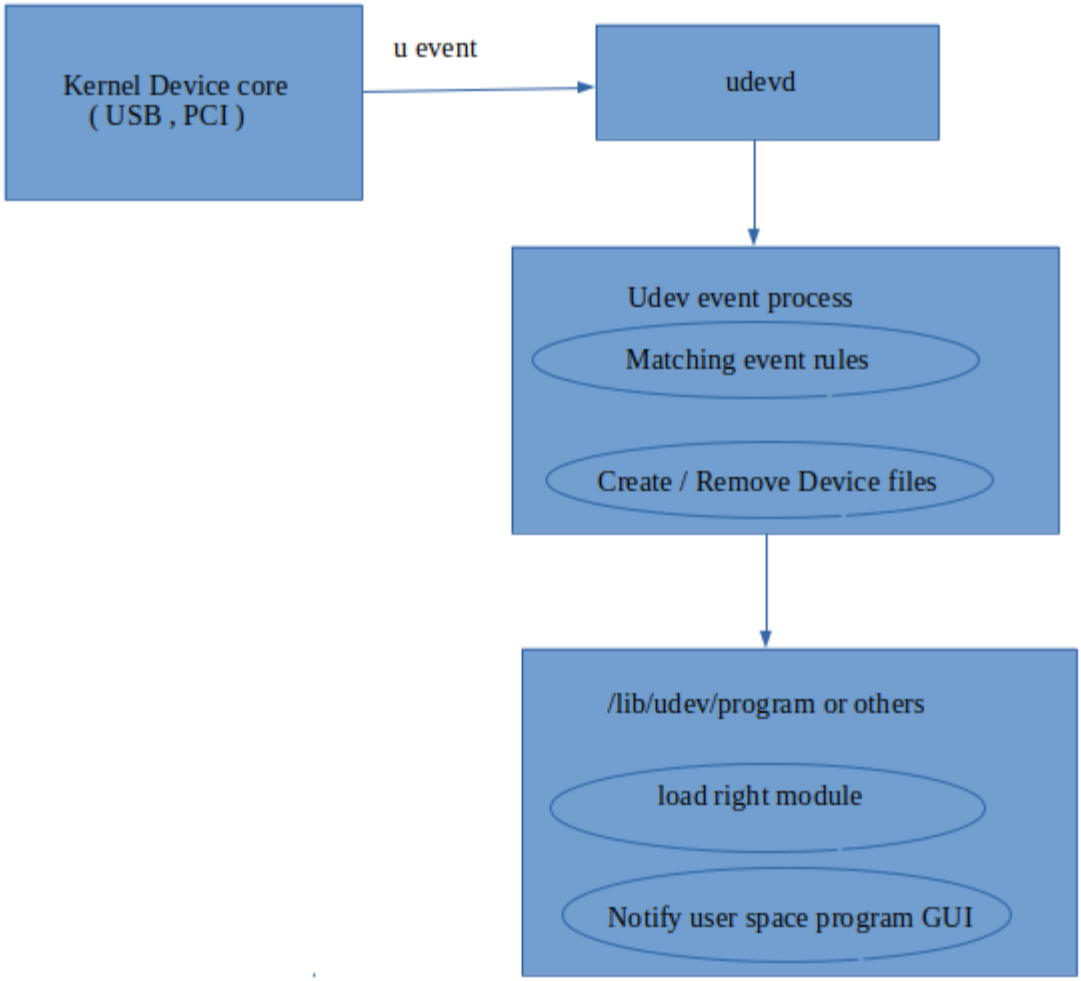
3. Hot-plug agent

- Pre-configured user space demon that receives all hot-plug events. Hot-plug events occurs when changes happen to syscore.

- Demon can be configured by mentioning path to its binary file in /proc/sys/kernel/hotplug

- Most Desktop “linux distribution use “U Dev” as default hot-plug agent”

- Busybox (elinux/ based distributions used “mdev” as default agent for hot-plug events)
Other well know hot-plug agents include “hald”.



Linux Hot-plug Subsystem (Udev)

Configuring udev :-

Step 1: Add a new virtual platform device to platform core

```
struct platform_device *pdev;

static __init int  dummy_init (void)
{
int ret;

Pdev – platform_device_alloc(“sample”,-1);
ret=platform_device_add(pdev);
return 0;

}
```

```
static __exit void dummy_exit( void)
{

platform_device_del(pdev);

}
```

Step 2: Implement driver module matching platform device.

```
Static struct platform_device_id veda_driver_ids[]={
{ “sample”, -1 },
{}


};


/* It creates the device (platform) type table for module-license */

MODULE_DEVICE_TABLE(platform, veda_driver_ids);

/* Populate this with driver and device details */

static struct platform_driver veda_sample_driver = {

        .driver = {
                        .name = “veda_sample”,
                        .owner = THIS_MODULE,
        .d_table = veda_driver_ids;
        .probe = sample_probe,
        .remove = sample_remove, };
/* Macro to register / un register with platform core */

module_platform_devices ( veda_sample_driver);
```

Step 3: write a udev rule describing response for device add event and remove event

/etc/udev/rules.d/ file_name.rules (my_driver_load.rules)

```
ACTION=”add, KERN=”sample “, RUN=”/sbin/insmod /root/hotplug/ptar.ko”
ACTION=”remove”, KERNAL=”sample”,RUN=”/sbin/rmmod platformdev”
```

Step 4: Load the rules udev database

        udevadm control –reload-rules

        man udev Linux dynamic Device Management detailed scripting udev.

* char driver can auto register through an interface register_chardev.

register_chardev(major,CHAR_DEV_NAME,&char_dev=poa
if (my_arr == 0) dynamic major num

* unregister_chardev(major,CHARDEV_NAME);

* Setting up char driver to misc interface

static struct misc_device vdevmisc = {

.minor = MISC_DYNAMIC_MINOR /* Allocate misc number */
.name = DEV_NAME,
.fops = &mar_dev_fops; /* driver fops initiates */

};

/* Register Driver with misc interface */

static int __init run_dev_init(void)
{
int ret;
ret = misc_register(udevmisc);
}

vfs → 10 → misc → dev → device_driver.

Driver Operations

Char driver operations

open Function

open: Function assigned to open interface of fops structure is invoked when application invokes open() api on device filename

open() → sys_open() → drv_open()

```
static int char_dev_open(struct inode *inode,struct file *file)
{
step 1: Validate open request as per access policy of the driver
        if ( !valid )
                return -1; // Negative no indicates error

step 2: Ensure target device is in wake-up statements

step 3:  Allocate caller operation specific data structure ( if any )

return 0;

}
```

To enforce user specific access policy kernel pointer **current** can be used

Current is a Kernel global variable pointer which refer to address of PCB of caller stack any process context function can be use current pointer to look into caller context details.

```
pr_info("Caller Process Name %s \n", current→comm);
pr_info("Caller Pid %ld \n", (long int ) current→pid );
pr_info("Stack start address ", current → stack);
```

release function

release: Function assigned to release interface of fops is invoked when application uses close() api on driver fd.

Close() → | → sys_close() → drv_release()

```
static int char_dev_release( struct inode * inode, struct file *file)
{
step 1: Decrement driver use counter ( If any )

step 2: Deallocate    application specific state info or data ( if tracked )

step 3: return 0;          }
```

read function

read: Function assigned to this interface is invoked when application read() api on device file descriptor

read() → | → sys_read() → drv_read()

read(fd,buf,512);

ssize_t char_drv_read(struct file *file, char __user *buf, size_t size, loff_t *offset)
{

Step 1: Validate read request of the application

Step 2: Verify and check if target device is ready for specific operation

Step 3: Execute device operation and gather database

Step 4: Convert data into application usable format (If required)

Step 5: Transfer data to user mode buffer (copy_to_user())

Step 6: return no of bytes transferred

}

write function

write: Function assigned to this interface is invoked when application write() api on device file descriptor

write → sys_write → drv_write()

write(fd,buf,512);

ssize_t char_dev_write(struct file *file, const char __user *buf, size_t *size, loff_t *off)
{
Step 1: Validate write request of the application

Step 2: Verify and Check if target device is busy , is ready for specific operations

Step 3: Transfer data from user mode buffer (copy_from_user)

Step 4: Convert data into device specific format (if needed)

Step 5: Transfer data to device or execute appropriate device operation

Step 6: Return no of bytes transferred }

Kernel Provides helper functions copy_to_user copy_from_user for ready / writing data for accessing user mode buffers.

Exchanging data with user space

→ Assign value
    get_user(v,p);
The Kernel variable v gets the value pointer by the user_space pointer p

→ pu_user(v,p);

The value pointed by the user space pointer p is set to the contents of variable v

Refer: /kernpro/cdrvex/clock.c
rtc.c chdrv_skel.c
Linux Device Driver Book

Drivers need to support configuration / special operations on a device
    Primary ops config ops

*It is invoiced to avoid to using read/write interface for implementation of configuration and setup operation.

*Unix System provides special api for execution of special operations on a file descriptor called ioctl int ioctl(int d, int request,---); // In device 70 % Special operations 30% core operations(Read/ write)

*The second argument is a device dependent request code. The third argument is an untyped pointer to memory. It's traditionally char *argp (from the days before void * way valid c)

Method 1: Supporting special operations in a driver

Step 1 : Decide on total no of configuration operations to be supported

Step 2: Create a request code for each special operation

Step 3: Implement a routine in a driver to process all supported request type register this function with unlocked_ioctl function pointer of the file operation structure.

Method 2: Request command can be defined either by choosing appropriate constants or generating constants using kernel helper macros.
It is preferred to generate request command through kernel macros for a quick validation and re-usability of code.

To generate request commands drivers will have to decide the following parameters for each request.

1. MAGIC no - (8 bit) usually common for all commands

2. Seq no - (8 bit) incremental

3. Arg type - (can vary for each command may or may not be applicable)

4. Direction of data transfer (in/out param may or may not be applicable)

To encode macros any of the following macros can be used

Encoding Macros

1. _IO(MAGIC,Seq no) - Use this when arg is not involved.

2. _IOW(MAGIC, seqno, type) - Use this when arg is in-parm for application

3. _IOR(MAGIC, seqno,type) - Use this when arg is out-param for application

4. _IOWR(MAGIC, seqno, type) -Use this if arg is used both as in/out param.

```
#define VEDA_MAGIC 'v'
#define SET_SECOND _IOW(VEDAMAGIC,1,char);
#define SET_MINUTE _IOW(VEDAMAGIC,2,char);
```

man ioctl_list

Method 3:

Template
ioctl → sys_ioctl → unlocked_ioctl (file_operations)

```
static long rtc_ioctl(struct file *file, unsigned int end, unsigned long arg);
{
```

Step 1: Validate applications request command

```
if(_IOC_TYPE(cmd) != VEDA_MAGIC )
```

Step 2: Process received command through appropriate conditional check

* A switch case construct is preferred for implementation of configuration under each command .

rtc_ioctl.c

```
 Static struct file_operations ={
unlocked_ioctl = rtc_ioctl;
}
```

```
ioctl(fd,SET_MINUTE,8 //it can be value or data );
```

Limitation of ioctl

ioctl is considered potential source of routine false

Application program is passing an address of integer, while driver is expecting an address of 1K data.

Application

```
int main()
{
int a=10;

ioctl(fd,DRV_ENCODE_CMD, &a);

}
```

Driver

```
        ioctl(---)
{
 case(DRV_ENCODED_CMD)

/* Transferring data to user mode */

copy_to_user(buf,a,1024); // !!! voilation

}
```

Solution: Kernel Provides ioctl request command decode macros through which a driver can verify passed from user mode.

```
If (_IOC_DIR(CMD)&_IOC_READ)
if ( ! access_ok(VERIFY_WRITE, (void*) arg,  _IOC_SIZE(cmd)).
        return -EFAULT;


If (_IOC_DIR(CMD)&_IOC_WRITE)
if ( ! access_ok(VERIFY_READ, (void*) arg,  _IOC_SIZE(cmd)).
        return -EFAULT;
```

* Usage of decode macros to validate user inputs is optional.

* Driver implementation which will not engage for decode macros for validation can be potential sources of run time fault as an alternative to ioctl, sysfs is preferred.

* Sysfs is a logical file systems designed an interface for hot-plug subsystem.

* The same interface can be used by drivers to enumerate configuration files through which driver can support configurations / Special operations.

* Providing a file system interface for supporting configuration operations, makes it simple for user mode apps to initiate config operations through shell command interface

cat → read from file , echo → write to file

Limitation:

* Sysfs represents all data as string constant, and each sysfs file is bound to a page size buffer 4K.

Creating Sysfs files:

Sysfs files can not created from user mode they can only with enumerate from kernel mode.
/* kobject_create_and_add - create a struct kobject dynamically and register it with sysfs */

Step 1: Instantiate new kobject which serves as an inode for new directory at a group of files

```
static struct kobject *my_rtc;

my_rtc = kobject_create_and_add("my_rtc", NULL /* Kernel_kobj */ );
```

Step 2: Assign file attributes

```
/* Linking routines to particular entry */
/* Use __ATTR family to ensure that naming convention */


static struct kobj_attribute tm_attribute =
__ATTR(time, 0666, tm_show, tm_store);
static struct kobj_attribute dt_attribute =
__ATTR(date, 0666, dt_show, dt_store);


static struct attribute *attrs[] = {
        &tm_attribute.attr,
        &dt_attribute.attr,
        NULL,   /* need to NULL terminate the list of attributes */
};

static struct attribute_group attr_group = {
        .attrs = attrs,
};
```

Step 3: Assign Attribute list to kobject to enumerate specify files

```
/* Create the files associated with this kobject */
        retval = sysfs_create_group(my_rtc, &attr_group);
        if (retval)
                kobject_put(my_rtc);
/* Removing sysfs entry */
        kobject_put(my_rtc);
```

Each file is bound with appropriate read/write callback function which are response to cat and echo cmds invoke

The following a invoke a response to cat command on the file

```
/* read routine for date entry */
static ssize_t tm_show(struct kobject *kobj, struct kobj_attribute *attr,
                       char *buf)
{
```

Step 1: Generate the data to be presented through appropriate device operation

Step 2: Write data into buffer received has argument string constant

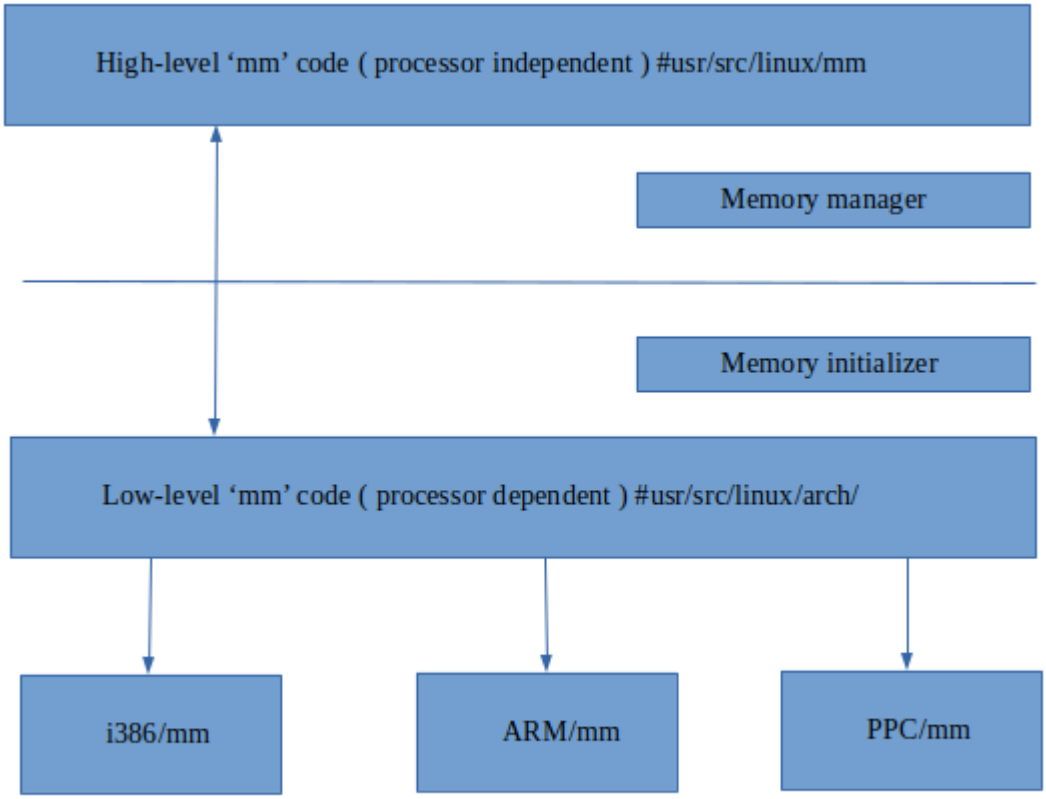
```
}

/* write routine for date entry */
static ssize_t tm_store(struct kobject *kobj, struct kobj_attribute *attr,
                        const char *buf, size_t count)
{
```

Step 1: Receive data from the argument buffer

Step 2: Apply Appropriate device operation

```
            return count  // Remember not write 0 ( Because applications may be check this write count )
}
```

Memory Sub System



Memory Sub System

* Runs at early boot stage end is responsible for following operations

1) Initialization of MMU.

2) Initialization page descriptor.

3) Setting up page frame table.

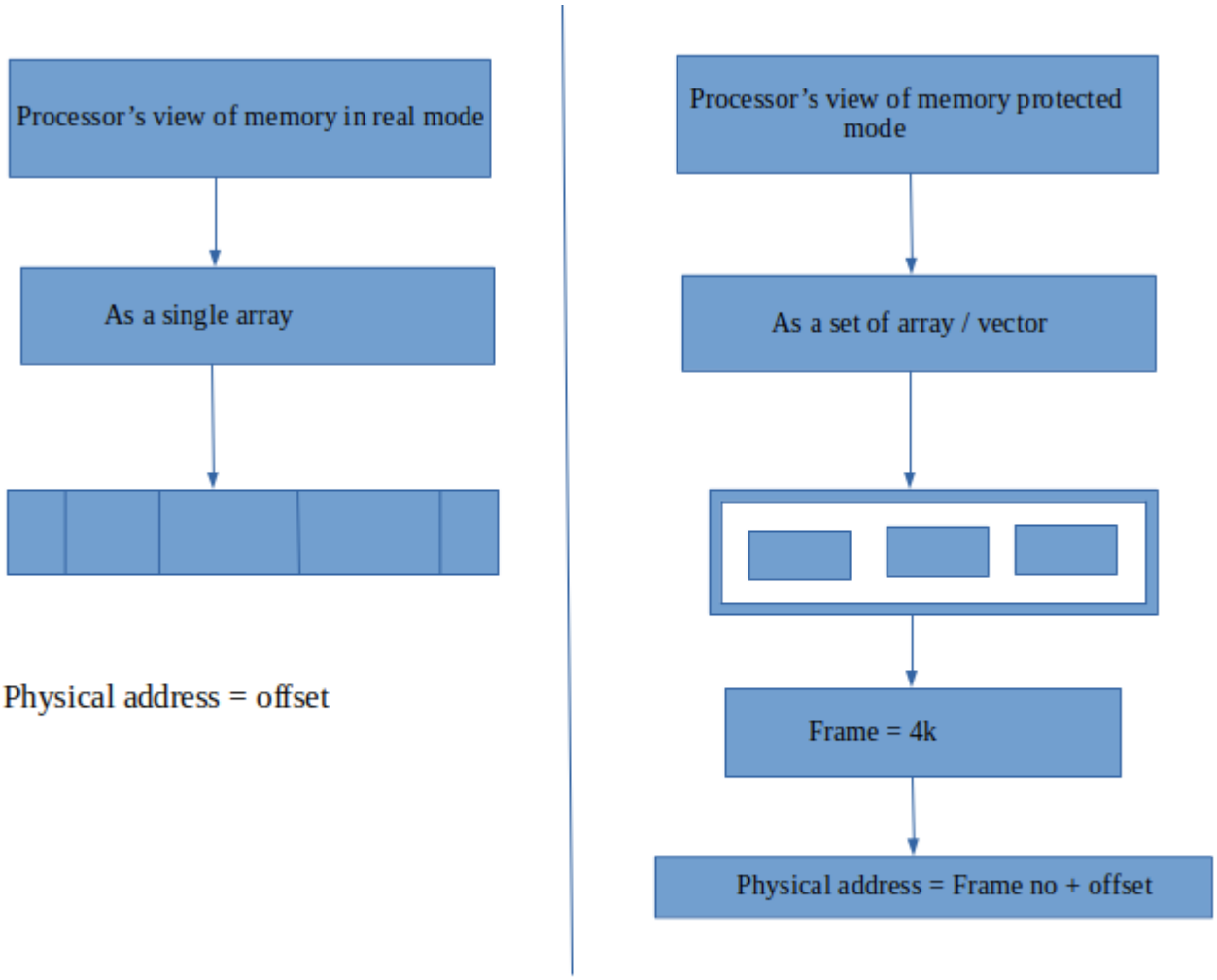
4) Allocation of zone descriptor.

5) Initialization of TLB (cache).

6) Management of TLB Buffers.

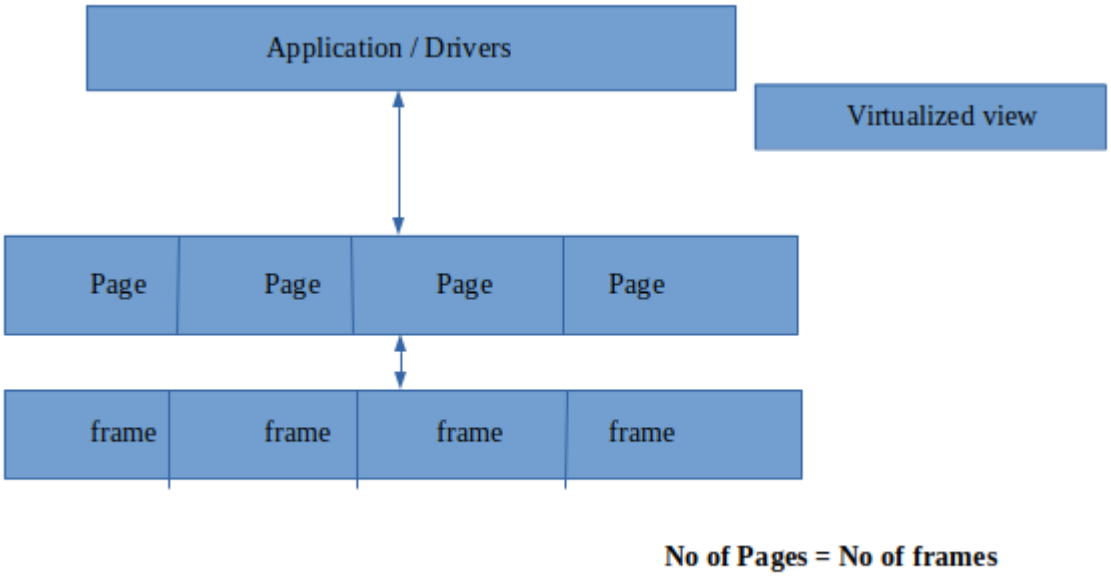
7) Initialization of low level memory allocate.

Hardware perspective changing (Memory View)



Hardware Perspective changing (Memory View)

Page Descriptor list



Page Descriptor list

No of pages = No of frames

* A page descriptor always represents a frame but the binding is dynamic

* When application acquires a page it receives the address to page descriptor.

* Any access operation of the application will required to provide page address which is translated into appropriate frame by the MMU using a data structure called page frame table.

* Page frame table contains page to frame mapping information as per page frame binding valid at particular point of time.

get_num_phys_pages(); ← How many pages

* Get physical address mapped to linear address return by allocator routine from page frame table

pa = (unsigned char *) __pa(ptr);

Acquires a page frame through allocator routine.

64 * 8 = 4k
Total frames may be 2604638

Virtual ← Addresses continues but physically scattered.
Linear ← Address is not contiguous but physically contiguous

* Drivers handing memory mapped devices will lead to create a page frame mapping of the device address space to gain access to device memory.

* This operation can be performed using either of the following kernel functions

1) io_remap_cache(); - io remapping cache
2) ioremap - Map bus memory into CPU Space.

* ioremap create a page frame mapping between device memory or bus memory and kernel pages by disabling CPU caches. This is preferred method of mapping device memorys to kernel linear address space.
* ioremap performs a platform specific sequence of operations to make bus memory cpu accessible via the readb/readw/readl/writeb/writew/writel functions and the other MMIO helpers.

The returned address is not guaranteed to be usable directly as a virtual address.

Ioremap – map bus memory into cpu space.
@offset – Bus address of the memory

arch/x86/include/asm/io.h +214

void iounmap(volatile void __iomem *addr);
releases the page binding for bus memory.

Kernpro/drvskel/ folder

* Implement char driver that supports open, release and ioctl operations on advanced programmable interrupt controller (IOAPIC)

Implement ioctl operations for the following

1. APIC GETID : returns identification no of APIC.

2. APIC_GETIRQS : returns no of IRQ pins on APIC.

3. APIC_GETIRQSTATUS : returns IRQ pin status (enabled, disabled)

4. APIC_GETIRQTYPE : return IRQ signal type.

Write a application to initiate and verify above commands.

Note: Implement two versions of code of above problem statements

I) VFS version.
II) sysfs version.

4) Allocation of Zone descriptor

→ Page frames are categorized into various groups called zones

→ Number of zones and size of each zone depends on architecture and address width of CPU.

DMA_ZONE

page frames representing initial 16MB of memory are categorized into this zone

ZONE_DMA32

*64 bit machine pages within 4gb of RAM are categorized into this zone

*This zone is used for allocations of 32 bit DMA buffers

ZONE_NORMAL

*Physical pages that can be mapped to kernels address space. Be on DMA_ZONE are categorized into normal zone.

* For 32 bit system with 1gb of kernels virtual address space 16 to 896MB of the address space is used for normal zone mapping.

* On 64 bit systems with 128gb of address space is available to kernel all physical memory behand 4gb is categorized into normal zone.

ZONE_HIGHMEM

*This zone is only possible in 32 bit system. All physical memory be hand 1gb which can not be directly mapped to kernel address space is referred to as high memory zone.

ZONE_DEVICE

* All physical memory pages enumerated from hot-plug memories are referred to as device zone.

ZONE MOVABLE

*Memory assigned to CMA allocator in considered movable zone

cat /proc/zoneinfo

cat /proc/buddyinfo

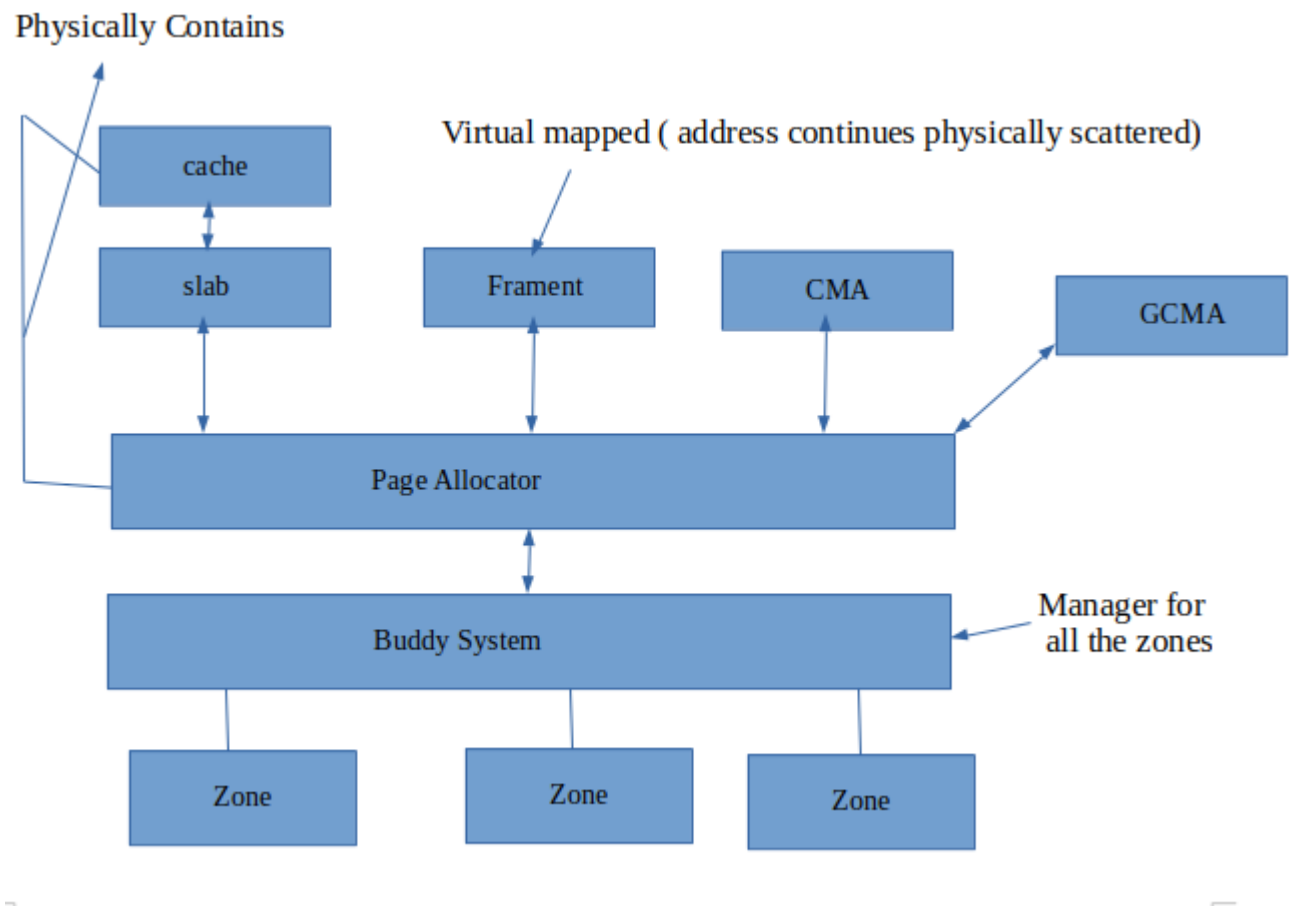
MEMORY MANAGER (UPPER LAYER)

1) Low level mm or memory initialization the buddy system algorithm as a default memory manager for all of the zones.

2) Buddy system initializes memory manager module which contains interface algorithm through which memory allocation deallocation requested/ received by the buddy system.

The following are list of interface algorithms

1) page allocator
2) slab allocator
3) cache allocator
4) Fragment allocator
5) DMA allocator
6) CMA allocator
7) GCMA allocator



1) Page allocator

* Base allocator responsible for managing physical per zone page list.

* Reset of the allocators are extensions to page allocator.

* Provides allocation routines for single page and order of 2 page allocators.

* Engages buddy allocation strategy to ensure requested allocations are physically contiguous

* Ensures (for 32 -bit machines) guaranteed allocator up to 8mb requests (be on the 8MB depends on memory fragmentation state.

Allocation routines

Page allocator provides two district allocation interfaces

1) Alloc_family_of
2) get_free page of family calls

struct page *alloc_pages(unsigned int gfp_mask, order);

To function requests 2 ** order contiguous page frames from the low level memory manager (buddy system) and returns the address of the descriptor of the first allocated page (not the page itself) or NULL. If the function valid.

To allocate a single page user the following function

struct page * alloc_page(unsigned int gfp_mask);

#include<linux/gfp.h>

unsigned long * __get_free_pages (unsigned int gfp_mask, unsigned int order);

gfp_mask : GFP ATOMIC, high priority non blocking.
gfp_KERNEL: kernel space allocation may block
GFP_USER user space allocation may block (contiguous 2^n order)

To allocate single page
unsigned long __get_free_page(unsigned int gfp_mask);

* All memory allocation functions requires a constant called gfp_mask as an argument integer constant.

gfp_mask is an integer encoded to specify A mode of allocation
B zone to allocate from

* gfp.h contains free defines macros for widely used zone and action combinations

* And allocation call can be invoked either by specified existing constants or by choosing appropriate parameters to encoding new constants.

The following are predefined options

1) GFP_ATOMIC - Normal zone high priority allocation (Reserve poll of pages)

2) GFP_KERNEL - Normal zone normal priority

3) GFP_NOWAIT - Normal zone non blocking allocation call

4) GFP_USER - Normal zone normal priority (blocking call)

5) GFP_DMA - DMA zone normal priority

6) GFP_DMA32 - DMA32 zone normal priority

7) HIGH_USER – 32 bit system only high memory zone normal priority

* When allocation function are invoked in normal mode the caller might be put into wait state if requested memory is not directly found in the free poll.

* Allocation algorithm look into process pages and file system caches (io caches) to squeeze out required memory by swapping out pages which are not currently used.

Sample program:

Method 1:

```
struct page *p;

char *lmem;

p= allocate_page(GFP_KERNEL);

lmem = page_address(p);

__free_page(p); // Release page
```

1. Lookup for frame no (page_to_pfn();)

2. look up for address of page descriptor returned to a frame (pfn_to_page())

Method 2:

```
char *lmem;

int module_init(void)
{
lmem = (char *) __get_free_page(GFP_KERNEL);

}
```

Following are important macros

1 PAGE_SIZE – current page size

2 PAGE_SHIFT – return the no of bits used to address offset. Allocators

2. Buddy system

* Internally used by page allocator to sort allocations.

* All free pages in a zone are grouped into n (Usually 2 power 11 lists of free blocks)

* If three is a request for given size ‘S’ and there are no free blocks into corresponding divided until block of size ‘s’ is reached buddy blocks are released (merged) to make a larger block when not in use (when they are freed).

Benefits :

* Minimizes external fragmentation
* Guarantees availability of physically contiguous blocks
* Quick allocation and Deallocations

Limitations :

* Since Allocations are page size or in multiples will cause internal fragmentation when caller does not need memory in multiples of page size.

* For instance a request for 3k buffer is processed by allocating 4k page, a request of 5k is processed by allowing 8k resulting in rest of memory being unused.

Slab Allocator

* Implemented as extension to page allocator
* Handles memory allocations in multiple of 8bytes (8, 16 ,32, 64, 128 ---- upto 4mb)
* Always returns start linear address of physically contiguous chunk
* Fails if it can not find physically contiguous chunk to process an allocator routines.

Kmalloc and family of routines

kmalloc()
kzalloc()
krealloc()
kcalloc()

cat /proc/slabinfo

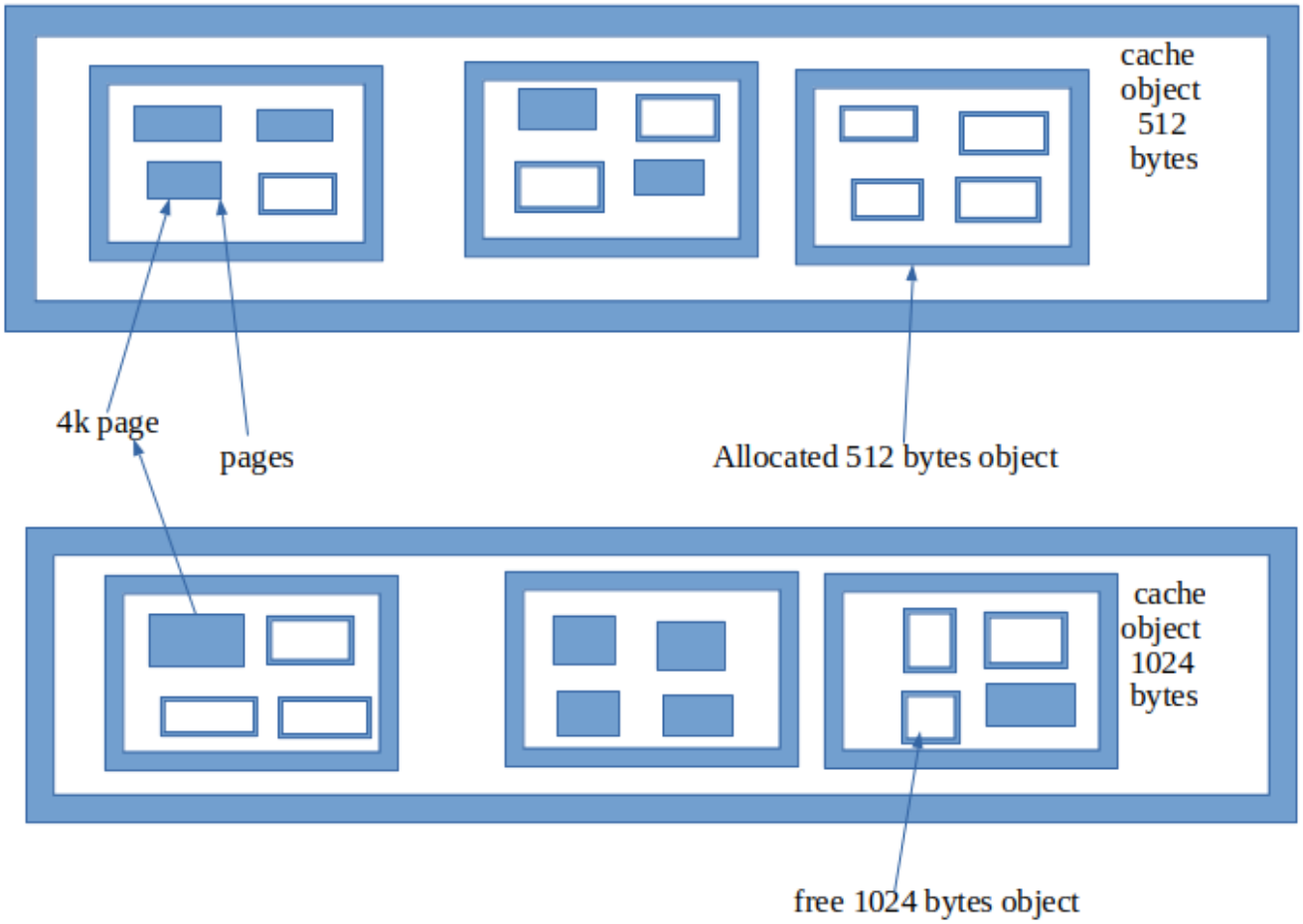
slab allocation design

1) During initialization slab allocator acquires a set of free pages and device then into free lists called slab caches.

2) each slab cache is a list of pages (1 or more) reserved for allocating blocks of a specific size.

cat /proc/slabinfo ← shows of each slab cache.

3) Slabs caches exists from 8bytes allocation block size to 8k



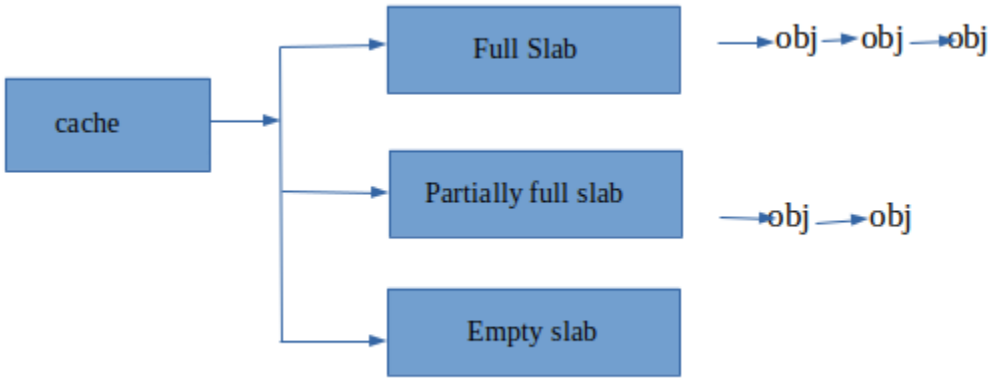
* To minimize allocation time and for efficient slabs and cache management (slab allocator)

* Each cache is structured with slab data structure.

* A cache is divided into slabs. A slab consists of one or more contiguous page frames a slab holds several objects.

* The slab allocator groups object into caches which is a store for object of the same type (same size)

Examples: Process descriptors, semaphores, inodes etc



Slab is
contained with in the pages used for block allocations

* A slab may be in one of three possible states

Full: All objects in the slab are marked as used

Empty: All objects in the slab are marked as free.

Partially full: The slab contains used and free objects.

Slab algorithm

* The slab allocator first attempts to find a free object in a partial slab.
* If no such object exists a free object is assigned from an empty slab.
* If no empty slab is available a new slab is allocated using the buddy system allocator.

Benefits of slabs:

* No (internal) fragmentation the slab allocator gives the exact amount of memory that has been requested to store an object.
* Memory requests can be processed quickly to released objects are only marked as free and can quickly be reassigned

Compile source time if you chhose Embedded system SLAB

* Kernel contains three variants of slab allocator implementation of which only one can be active in a particular build.
* At kernel build time appropriate slab allocator must be chosen.

1) SLAB ← config-SLAB traditional slab allocator designed for single core processor systems

2) config_SLUB (uniqueed allocator) ← A variant of classic SLUB algorithm designed for multi core systems. (per CPU caching is realized using slabs of objects

* SLUB is the default choice for a slab allocator.

3) Config_SLOB ← SLOB replaces the stack allocator with a drastically simpler allocator SLUB is generally more space efficient but does not perform as well on large systems (< 512MB RAM low memory systems)

* Slab allocators allows kernel services to setup private caches these caches can be used to reserve memory blocks (Memory pool) or object blocks (object pool)

3. cache allocator

* Built on top of slab allocator

* Allow kernel services to pre allocate memory caches

* Allows allocation of memory / object pools

* Always serves physically contiguous memory

* Returns start linear address of allocated memory / object

Sample: linux/slab.h

```
strcut kmem_cache *mycache;

void *object;
    mycache = kmem_cache_create("test_cache", 128, 0, SLAB_RED_ZONE, NULL); // 0 is a alignment block

    object = kmem_cache_alloc(mycache, GFP_KERNEL);
kmem_cache_free(mycache, object);
    kmem_cache_destroy(mycache);

struct kmem_cache *my_cache;

typedef struct {
    int a;
    int b;
    int c;
} private_data;

private_data *handle;
```

cache constructor routine

```
* Invoke with address of each object as parameter
* This call gets invoked whenever new objects are setup
static void cache_init(void *data)
{
        private_data *mydata = (private_data *) data;
        mydata->a = 0;
        mydata->b = 0;
        mydata->c = 0;
}

cache specific derived object allocator //deallocator routines
Use these routines from rest of driver code for alloc/dealloc ops
static private_data *myalloc(void)
{
        private_data *mydata;
        mydata = (private_data *) kmem_cache_alloc(my_cache, GFP_KERNEL);
        return mydata;
}
/* create  private slab cache list and align it with hardware cache */
        my_cache =
           kmem_cache_create("mycache", sizeof(private_data), 0,
                        SLAB_HWCACHE_ALIGN, cache_init);


/* create  private slab cache list and align it with hardware cache */
        my_cache =
           kmem_cache_create("mycache", sizeof(private_data), 0,
                        SLAB_HWCACHE_ALIGN, cache_init);
```

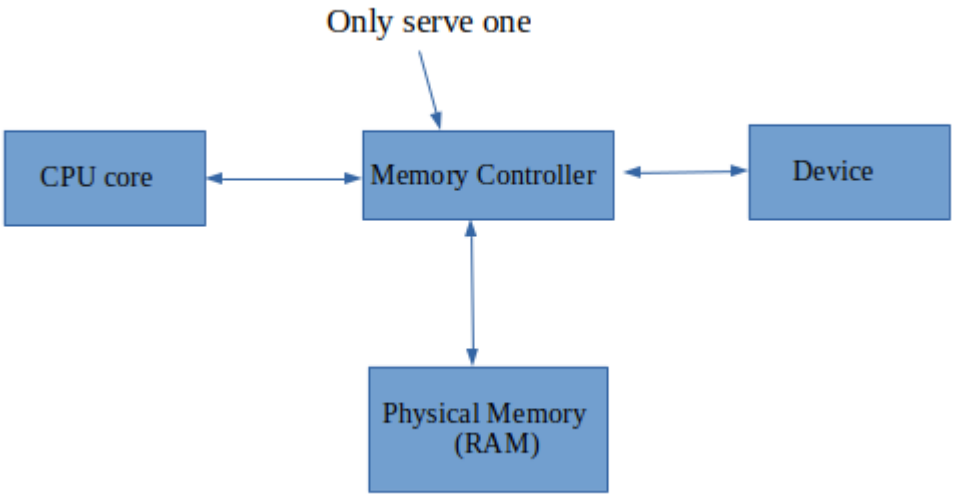
4) Fragment Allocator

* Built on top of page allocator
* handles memory allocation requests of random sizes
* returns virtually contiguous memory by joining fragments list over in pages
* returns start kernel virtual address
* requires unique address translation table for each allocation and is generated at allocation time.

Routines : vmalloc(), vfree vzalloc vmalloc.h
ptr = vmalloc(6000);
linear = page+offset
physical = page +frame
virtual page+offset (no of continuous)

DMA allocation Routine

* DMA is capability of a device to perform direct access operation on primary memory
* DMA capable devices use an additional circuit call DMA engine to drive address and control BUS.
* While DMA operations are in progress CPU does not have access to primary memory.



The Following assume that as allocating buffer for a tx operations through DMA

The Driver will have to except the following steps.

1) Driver setup the buffer

2) Populate Tx data (Write operation)

3) Retrieves address of the buffer and programs it into device.

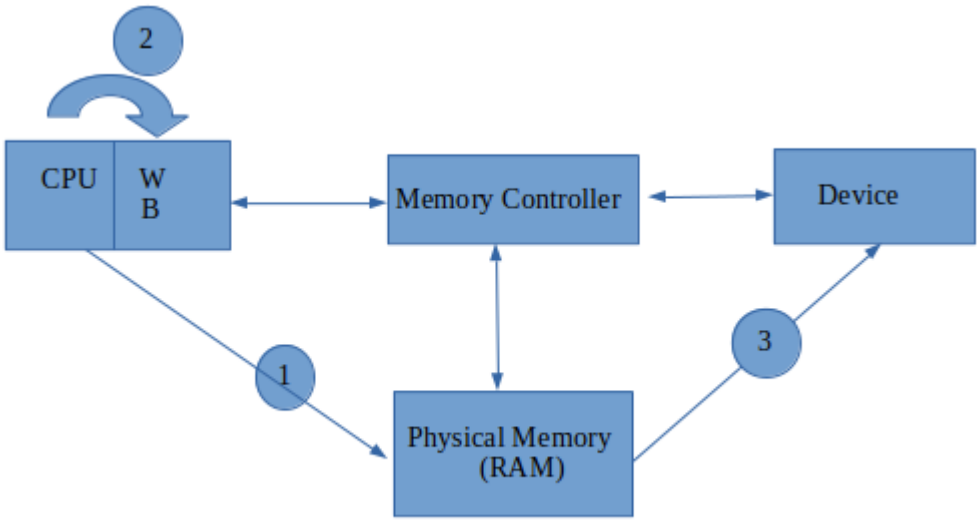
4) Start DMA transferred

Case 1: Transmission operations might not always succeed with above steps due to hardware caches used by CPU.

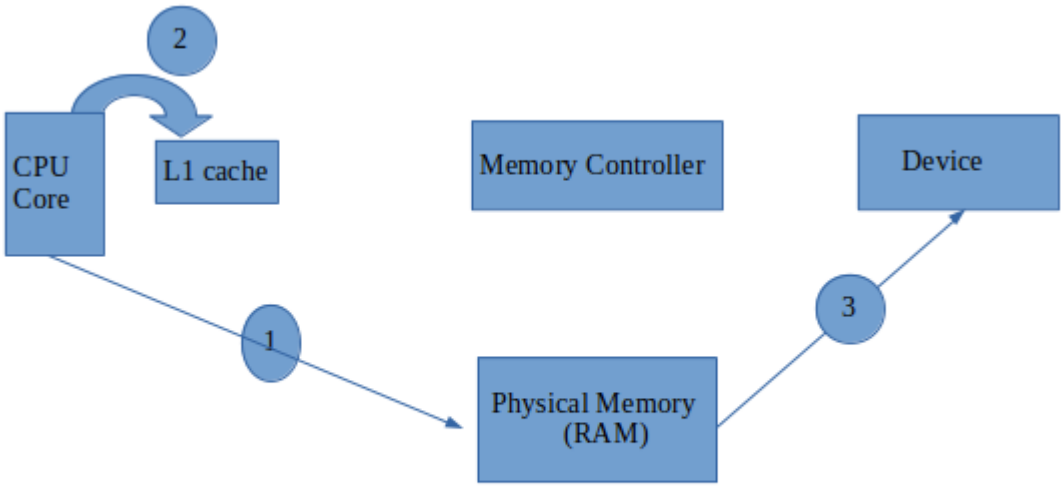
Sol: To ensure successful transmission driver must be programmed to flash hardware caches at a appropriate time or set up DMA buffers as bounce buffer (outside the cache)

CPU's write Buffer

No of cache flush instructions depends on machine CPU configuration the following are few possibilities

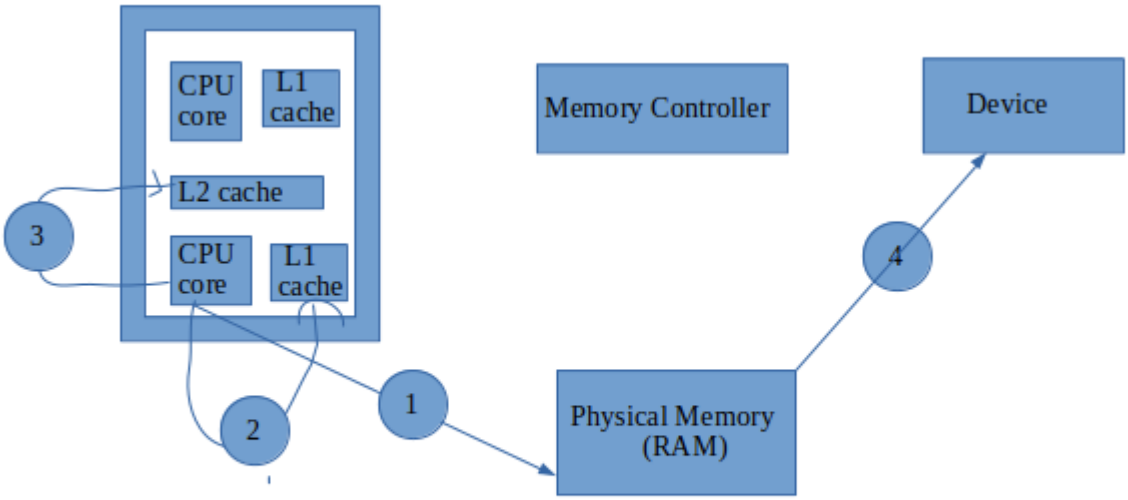


Case 2: CPU’s with RW cache



CPU’s With RW Buffer

Case 3: CPU’s with multiple level of RW caches (L2 cache)

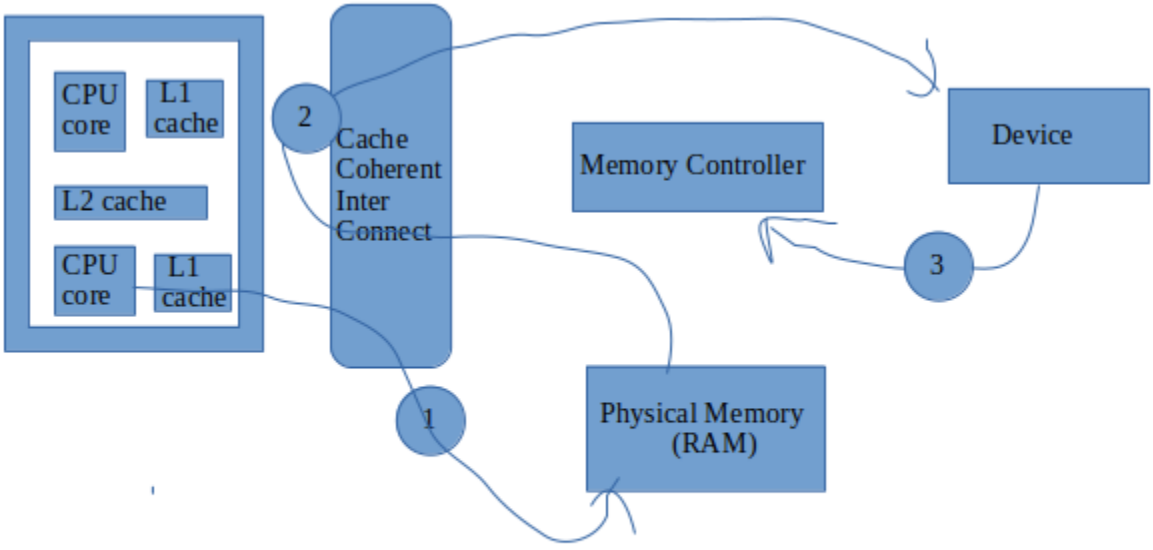


CPU’s With RW Buffer (L2 cache)

* CPU write to memory
* CPU cleans L1 cache
* CPU cleans L2 cache
* Device reads from memory

CPU's with cache coherent interconnect

cache coherent inter connect is a controller with in CPU which synchronizes data in hardware (CPU) cache with memory without the need for software flush instructions.



CPU's with cache coherent interconnect

Flush is not required in this case

BUS Address Space

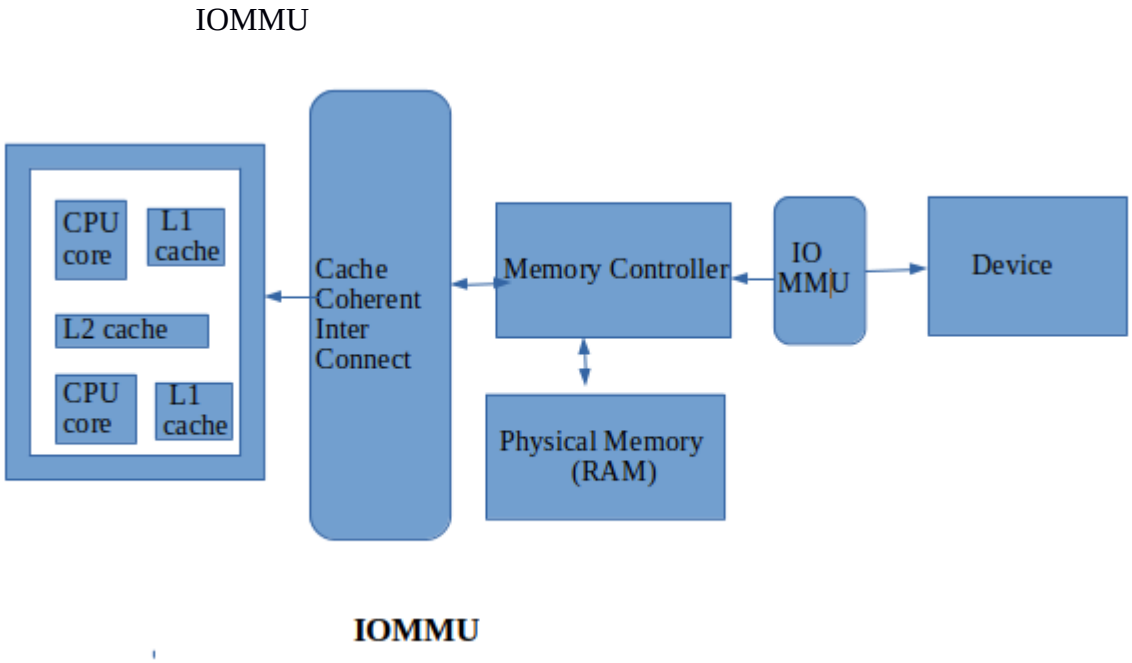
1. DMA controllers directly address primary memory, DMA address space is often referred to as BUS address space.

2. Depending on type and configuration of DMA engine bus address length can be 16 | 24 | 32 | 64 bits memory visible and accessible by DMA engines depends on BUS address size (DMA Mask)

3. On x86 Platforms DMA bus address space coincides with CPU physical address space.

4. On RISC architectures like ARM, DMA bus address space doesn't coincide with CPU's physical address on such platforms DMA's are equipped with additional hardware circuit called IOMMU that translates bus address to CPU physical address.

5. When drivers allocate / setup memory for DMA operations they need to setup appropriate mapping entries in IOMMU.



Information to be obtained from hardware manuals

1. DMA Type – Identify location of DMA engine (Onchip/offchip)
   - local
   - shared

2. DMA Mask : Bus address width.

3. IOMMU

4. cache memory
   -hardware supplied
   - Manual coherency (flush , invalidate, sync)
   - H/W cache details in absence of cache coherent inter-connect.

Linux Kernel Support for DMA ops

1. DMA buffer allocation and deal location interfaces
2. Programming IOMMU
3. Manual synchronization b/w DMA address region and CPU caches through sync routines.

All of the above operations are implemented as part of architecture specific branch of the kernel tree.

Drivers can utilize these services through abstract interfaces which ensure portability.

Kernel provides two groups of apis

1) cache coherent DMA5 allocations
2) DMA streaming api

1. Cache coherent DMA allocations

* This api interface allocates cache coherent buffers which do not requires manual synchronization. on platforms which do not supports hardware cache coherency.
* This api disables caching on DMA buffers.
* This api allocates buffer programs IOMMU. It returns kernel linear address and bus address address of the buffer.
* This api is preferred while setting up DMA buffers which are validate through out drivers life time.
* Buffers setup by this api can be concurrently accessed by device and processor vice-versa.

Void *dma_alloc_coherent(struct device dev,size_t size,dma_addr_t,dma_handle, gfp_t flags)

This routine allocates a region of <size> bytes of consistent memory.

* It returns a pointer to the allocated region (in the kernel linear address space) or NULL if the allocation failed.

* It also returns a <dma_handle> which can be given to the device as the DMA address base of the region.

Void *dma_zalloc_coherent(struct device dev, size,dma_addr_t,dma_handle,gfp_t flag);

void *dma_free_coherent(struct device dev, size,void * cpi_addr,dma_addr_t,dma_handle);

* Free region of consistent memory previously allocated. Dev size and dma_handle must call be the same as those passed into dma_alloc_coherent() cpu addr must be linear address returned by the dma_alloc_coherent();

2) Streaming api

* This interface provides a mapping to an existing kernel buffer such mapping are preferred for temporary transfer operations.

* This interface synchronizes memory contents with hardware cache through manual flush instructions (It doesn't depend on cache coherent protocols)

* This mapping must be accessed either by device or by the driver (not concurrently)

dma_addr dma_map_single(struct device *dev,void cpu_addr,size_t size, enum dma_data_direction direction);

* Maps a piece of processor linear memory so it can be accessed by the device and returns the DMA address of the memory.

The direction can be any of the following

DMA_TO_DEVICE data is going from the memory to the device.

DMA_FROM_DEVICE data is coming from device to the memory.

DMA_BIDIRECTIONAL data flow is bidirectional.

Void dma_unmap_single(struct device *dev,dma_addr_t dma_addr,size_t size, enum dma_data_direction direction);

unmaps the region previously mapped, all the parameters passed in must be identical to those passed in (and returned) by the mapping api.

dma_addr dma_map_page(struct device *dev,dma_addr_t dma_address,size_t size, enum dma_data_direction direction);

void dma_map_page(struct device *dev,struct page *page,unsigned long off_set, size_t ,size_t size, enum dma_data_direction direction);

API for mapping and unmapping for pages all the notes and warnings for the other mapping API's apply here. Also although the < offset > and < size > parameters are provided to do partial page mapping, it is recommended that you never use these unless you really know what the cache width is.

Sample Code:

DMA code api example

Packet transmission routine network controller routine.

```
static int at91ether_start_xmit(struct sk_buff *skb, struct net_device *dev)
{
dma_addr_t skb_physaddr;
skb_physaddr = dma_map_single(NULL, skb->data, skb->len, DMA_TO_DEVICE);

 /* Set address of the data in the Transmit Address register */
      macb_write(lp, TAR, skb_physaddr);
/* Set length of the packet in the Transmit Control register */
      macb_write(lp, TCR, skb→len);
```

An interrupt is triggered when transmission complete drivers interrupt handler is programmed to verify status of transmission and release dma mapping.

Static irq return_t at91_either_interrupt(int irq,void *dev_id);

*Mac interrupt status register indicates when interrupts are pending it is automatically cleared once read.

```
       intstatus = macb_read(lp, ISR);
 if (intstatus & MACB_BIT(RCOMP))        /* Receive complete */
            at91ether_rx(dev);

      if (intstatus & MACB_BIT(TCOMP)) {       /* Transmit complete */
            /* The TCOM bit is set even if the transmission failed. */
            if (intstatus & (MACB_BIT(ISR_TUND)| MACB_BIT(ISR_RLE))
                  dev->stats.tx_errors += 1;
```

```
        if (lp->skb) {
            dev_kfree_skb_irq(lp->skb);
            lp->skb = NULL;
            dma_unmap_single(NULL, lp->skb_physaddr, lp->skb_length, DMA_TO_DEVICE);
        macb.c at91_ether.c  linux-3.14/drivers/net/ethernet/cadence
```

DMA coherent buffers for receiving RX packets.
The following is drivers start function responsible for setting up RX buffer

```
#define MAX_RBUFF_SZ    0x600
/* max number of receive buffers */
#define MAX_RX_DESCR    9

/* Initialize and start the Receiver and Transmit subsystems */
static int at91ether_start(struct net_device *dev)
{
    struct macb *lp = netdev_priv(dev);
    dma_addr_t addr;
    u32 ctl;
    int i;
/* Allocate descriptor  list */
    lp->rx_ring = dma_alloc_coherent(&lp->pdev->dev,
                        (MAX_RX_DESCR *
                         sizeof(struct macb_dma_desc)),
                        &lp->rx_ring_dma, GFP_KERNEL);
/* Allocate buffer list */
lp->rx_buffers = dma_alloc_coherent(&lp->pdev->dev,
                          MAX_RX_DESCR * MAX_RBUFF_SZ,
                          &lp->rx_buffers_dma, GFP_KERNEL);
```

```
/* Initialize descriptor ring */
 addr = lp->rx_buffers_dma;
      for (i = 0; i < MAX_RX_DESCR; i++) {
            lp->rx_ring[i].addr = addr;
            lp->rx_ring[i].ctrl = 0;
            addr += MAX_RBUFF_SZ;
      }

      /* Set the Wrap bit on the last descriptor */
      lp->rx_ring[MAX_RX_DESCR - 1].addr |= MACB_BIT(RX_WRAP);

      /* Reset buffer index */
      lp->rx_tail = 0;

      /* Program address of descriptor list in Rx Buffer Queue register */
      macb_writel(lp, RBQP, lp->rx_ring_dma);

      /* Enable Receive and Transmit */
      ctl = macb_readl(lp, NCR);
      macb_writel(lp, NCR, ctl | MACB_BIT(RE) | MACB_BIT(TE));
```

Problem :coherent and streaming mapping api's might fail when huge DMA buffer blocks are required (Any thing larger than 8MB considered as huge.

Devices which requires huge buffer block can use alternate methods for setting up DMA.

Method 1:

Scatter- gather DMA Buffer

This method allow a driver to allocate an array of blocks memory and array of block of memory and mapped to a contiguous DMA address by programming IOMMU.

For such allocation drivers will have to set up an array of buffers using appropriate memory allocator and invoke the kernel function to map list of buffers to contiguous DMA address space.

Scatter – gather mapping

int dma_map_sg(struct device *dev,struct scatterlist *sg, int nents,enum dma_data_direction, direction);

returns: The no of DMA address segments mapped (this may be shorter than < nents > passed in if same elements of the scatter/ gather list are physically or linearly adjacent and an IOMMU maps then with a single entry).

* DMA address of scatter – gather mapping can be retrived the following function.

hw_address[i]=sg_dma_address(sg);

hw_len[i] = sg_dma_len(sg);

void dma_unmap_sg(struct device *dev,struct scatterlist *sg, int nents,enum dma_data_direction direction);

unmap the previously mapped scatter – gather list all the parameters must be the same as those and passed in to the scatter – gather mapping API.

Method-2:

Reserve Memory at boot

* Memory can be reserved such that the kernel doesn't use it.

→ MEM=512M on the kernel command line causes it to user only 512M of memory

→ The device tree memory can also be changed.

* This is the oldest method allowing large amount of memory to be allocated for DMA.

* Drivers use ioremap() to map the physical memory address into the virtual address space.

* There are multiple versions ioremap() which allow cached and non cached.

* These functions don't allocate any memory they only map the memory into the address space in the page tables.

* The linux ioremap() function causes the memory to be setup as device memory in the MMU which should be slower than normal memory.

Limitation:

Impact on system performance

as amount of memory reserved at boot time increased system performance might be impacted due to reduced memory.

* Since kernel 3.14 a new memory allocator is introduced with a name CMA (contiguous memory allocator) which attempts to bring reserved memories into regular system usage when DMA operations or not active.

CMA allows reserving memory at boot time and presence into buddy system as a zone movable.

* Any kernel component or application can allocate memory from movable zone when available.

* Drivers can get request DMA buffer using regular coherent api’s

* When memory is required to be set up to DMA while it is in use for regular allocation (system allocation) then data in a regular allocation is moved into other zones

to ensure success of DMA allocation call.

CONFIG_CMA (To enable CMA)

This enables the contiguous memory allocator which allows other subsystems to allocate big physical - contiguous blocks of memory CMA reserves a region of memory and allows only movable Pages to be allocated from it. This way the kernel can use the memory for page cache and when a subsystem requests for contiguous areas the allocated pages are migrated away to serve the contiguous request.

CONFIG_DMA_CMA(DMA contiguous Memory allocator)

This enables the contiguous memory allocator which allows drivers to allocate big physically-contiguous blocks of memory for use with hardware components that do not support i/o map or scatter-gather size also need to select CMA deployments when not entirely successful in ensuring successful DMA allocations.

On most Enterprise class system it was observed that CMA allocations held by regular system components (Applications, drivers etc) when not movable instantly if releasing such memory to DMA purposes resulting in failure of critical DMA allocations or performance delays in DMA transaction.

To ensure priority DMA allocations never fail or get delayed on alternate allocator to manage reserved memories called GCMA (Guarantee contiguous memory allocator)

Transcendental Memories (Tmem)

on enterprise class system hot-plug memories are used as a back store for containing pages swapped-out by virtual memory manager, file systems.

* VM Swaps out process pages which are not currently in use into Tmem and schedules and disc sync and file systems move all directly file buffers from the io cache into Tmem (Instead of discards).

* Since Tmem is been used as a (back store of data and files which might be required by a system component in a future) it enhances io performance of the system.

* GCMA allows reserved memory to be used as a Tmem which addresses of limitation of CMA and guarantee successful DMA allocations.

* Front swap [component of virtual memory manager which handles swapping activity] and clean cache [components of FILE i/o cache manager] are clients for GCMA.

Address Translation

* Application in the user mode purse memory as a set of blocks each created by mapping n no of pages

* Each block is represented by kernel data structure vm_area_struct

* Each vm ware region is identified with a start address and is mapped through a set of pages which are either anonymous or file mapped.

* Blocks representing code,data, .so files or file mapped and blocks representing heap, stack and bss contain anonymous pages.

* List of vma instances and their attributes (start address length, access permissions, etc) are maintained in memory descriptor of the applications PCB (task_struct → mm)

* All references to data (stack, data, heap, mmap) from applications code would use virtual address of data (which is block_start_address + offset)

* MMU's segment unit will have to look up into memory descriptor (VMA) of the process and locate vma instance for the specified address vma look up will produce linear address (page + offset) of the data element in memory.

* MMU's paging unit will have to translate linear address (discovered by segment unit) into physical address (frame + offset) using page table entries of the process.

* Linear address translation requires a look up into page tables.

* Kernel must maintain per process page table for ensuring quick address translation.

* On 32 bit address space with 3GB of virtual address per process and 4KB frame sizes 3MB of memory would be required to store translation entries for each process.

* This need for reserving memory for page table results in overhead and doesn't allow effective memory utilization for actual program kernel execution.

* To ensure effective memory utilization still support multitasking. Virtual memory manager is programmed to engage level-indirection algorithms for page table management.

* These algorithms are minimized memory reserved for a page table and facilitate dynamic expansion of page table.

Page table level indirection policy's

→ Two-level paging (32-bit systems)

→ Three level paging (64-bit systems up to kernel 4.6)

→ Four level paging (64-bit systems since kernel 4.6)

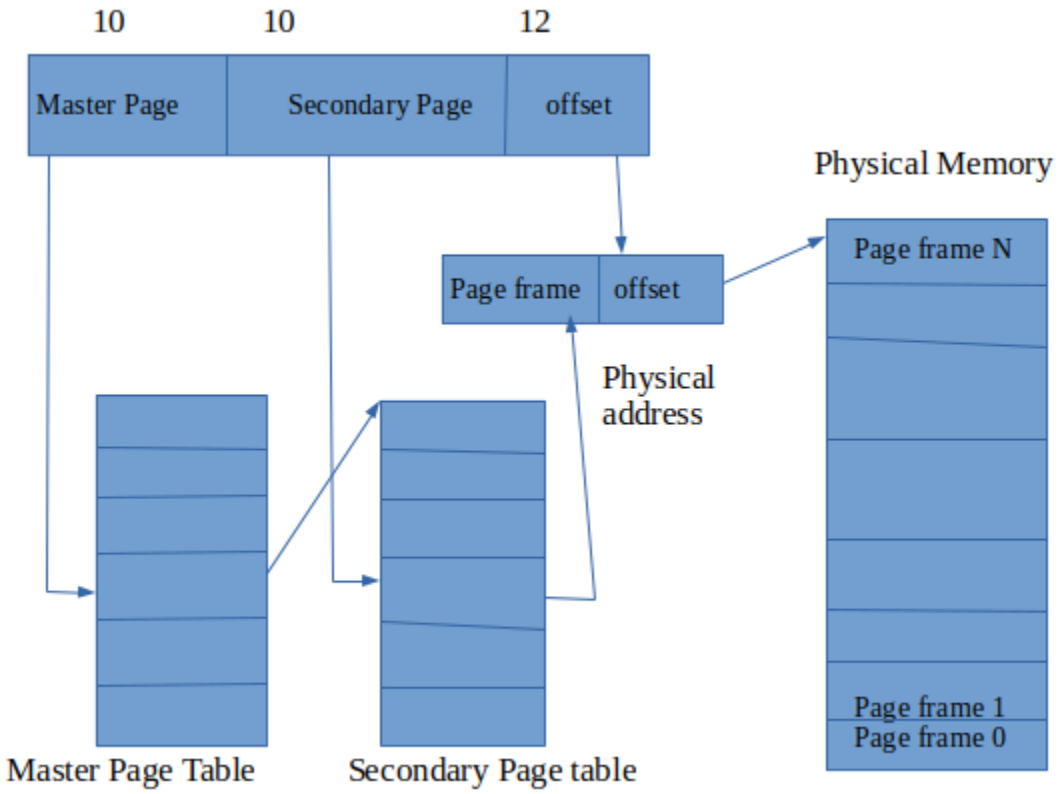
→ five level paging (64-bit systems since kernel 4.6)

Two-level page tables

* Virtual addresses have linear parts.

* Master page table maps master page number to secondary page table. (Must be in memory can't discard)

* Secondary page table maps secondary page number to page frame number.



System can support 1024 process at a give point it time.

Three level paging for 64-bit

32-bit PAE mode

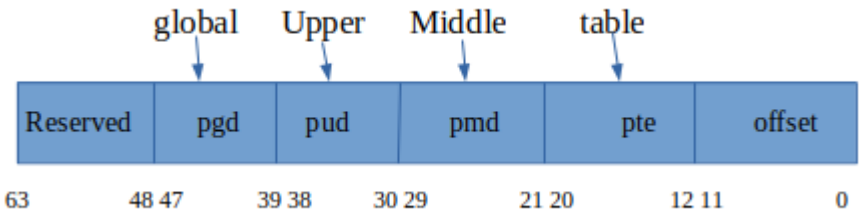
pgb pmd pte offset

offset	0-11 bit	12-bit
pte	12-20bits	9bits
pmd		
pgd		
ppd	30-31	2bits

* Three level paging reserves 2bits of address an used as it an identifier into page global directions (PGD).

PGD is master page table which contains four records each point to page middle directory

* Each page middle directory contains 512 records each reference into a process page table with this model 4 * 512 page table can be supported.



* Virtual Memory subsystem provides a set of macros for looking up into page tables asm/pgtables.h

pgd_offset:

Returns page address of page upper directory by looking up into global directory.

```
                pgd_t *pgd;
/* Access base Address of PUD */
pgd = pgd_offset(current→mm,addr);
if(pgd_none(*pgd ) || pgd_bad( *pgd));
```

pud_offset:

Returns base address of middle directory

puid_t *pud;

/* Access base address of PMD */

pud = pud_offset(pgd,addr);

if(pud_none(*pud) || pud_bad(*pud))

goto out;

pmd-offset:

Returns base address of page table (process)

pmd_t *pmd;

/* Find an entry in the second -level page table */

pmd= pmd_offset(pmd,addr);

if(pmd_none(*pmd) || pmd_bad(*pmd))

goto out;

pte_offset_map:

returns base address of frame by looking up into process page table.

pte_t *ptep,pte;

pstep = pte_offset_map(pmd,addr);

if(!ptep) goto out;

INTERRUPTS

An interrupt is an event that alters the sequence of instructions executed by a processor in corresponding to electrical signals generated by h/w circuits by the inside and outside CPU.

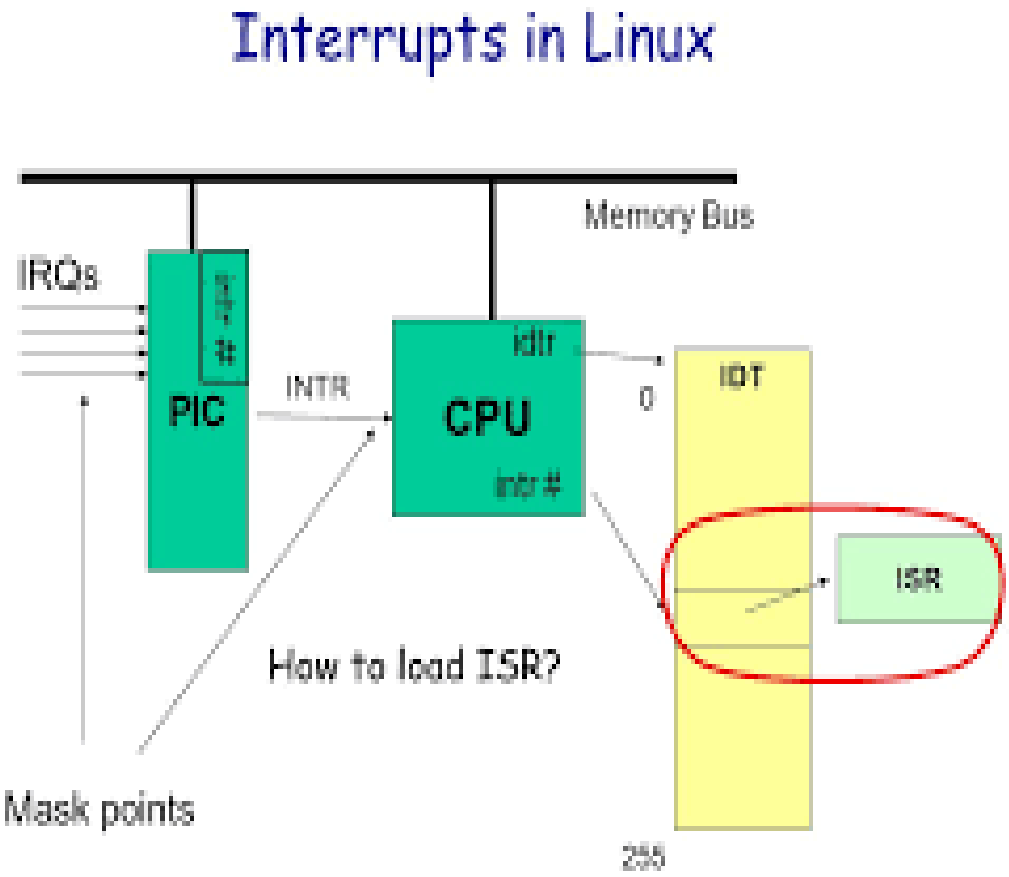
Hard Interrupts:

1) CPU's for most architecture are designed to receive external interrupts through a single input pin usually called INTR.

2) All the hardware controller are capable of generating an interrupt will need to relay an interrupt request signal on the CPU's INTR pin.

3) To ensure that all i/o controllers have access to CPU's INTR. Especial circuit is used to route interrupts from i/o controller to CPU, called interrupt controller.

X86 Interrupt Handling via IDT



* Kernel must setup during system start up (set-and-forget) UDT and SIDT used to set/get the pointer to this table.

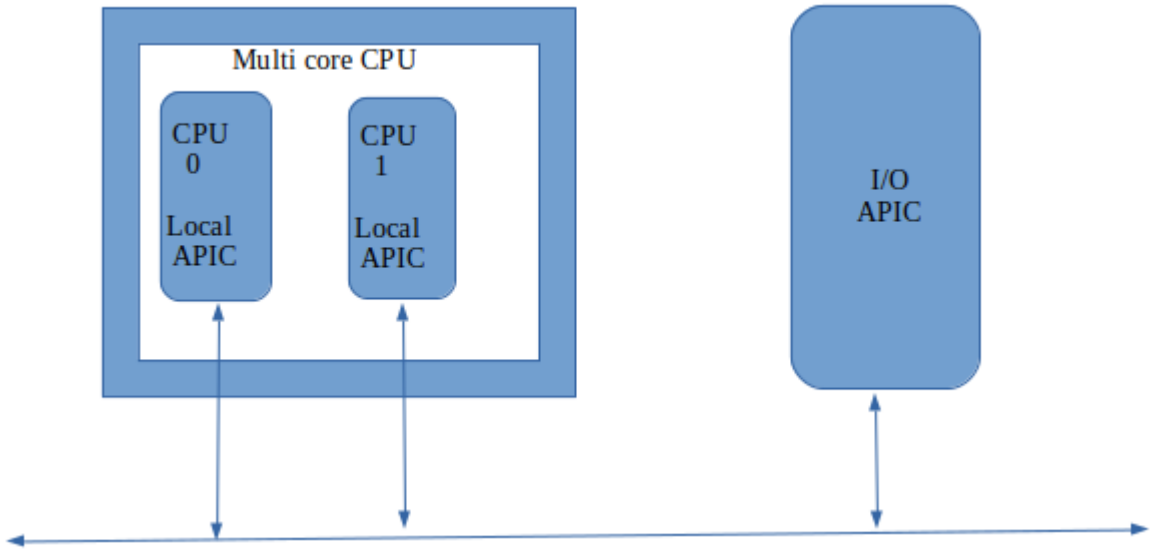
* I/o devices have (unique or shared) Interrupt request lines (IRQ's)

* IRQ's are mapped by special hardware to interrupt vectors, and passed to the CPU.

* This hardware is called as programmable interrupt controller (PIC)

* Possible to "mask" (disable) interrupts at PIC or CPU. (INTR can be disable line or INTR we can)

Interrupt Controllers on many core CPU's :



Advanced Programmable Interrupt Controller is needed to perform routing of I/O requests from peripheral to CPU's

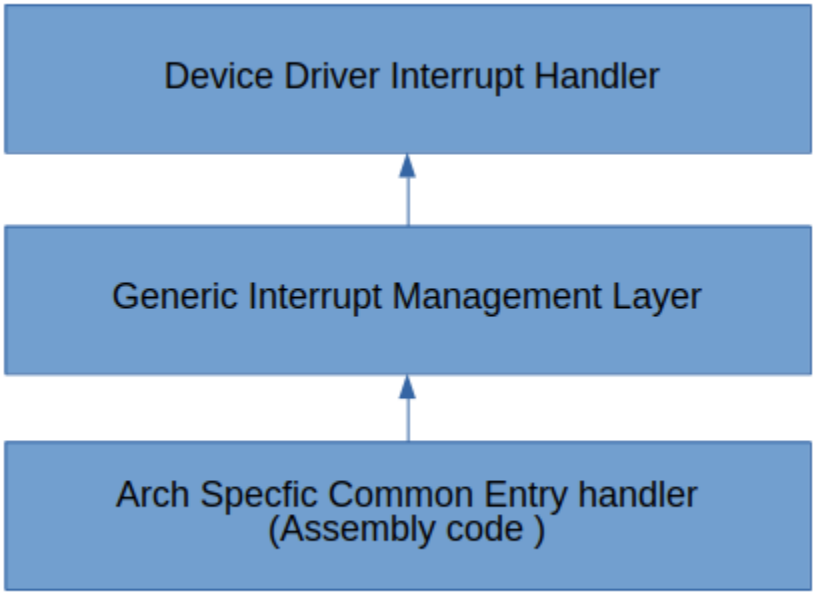
* Interrupts will go to selected CPU only

* The legacy MC save marked when the APIC are enabled

* Device Connect to front end IO-APIC

IO-APIC communicates (over bus) with local APIC.

LINUX KERNEL INTERRUPT SUBSYSTEM



Entry Handler:

A Common interrupt service routine which is applied for the all hardware interrupt vectors, this function is implemented by kernel BSP code.

This function is responsible for following operations

1. Initialization of hard IRQ stack
2. Conversion of the vector number on which interrupt is reported back to IRQ
3. Notifying Interrupt Management layer by passing IRQ number as an argument.

Kernels Interrupt Management layer:

This is responsible for IRQ and internalization of Driver Interrupt Handlers.

* This layer contains implementation of interrupt controller driver and IRQ live management routines.
* This layer carries out following operations upon reception of interrupt event and IRQ number.

1) Converts Hard IRQ number received from the entry handler into kernel IRQ.
2) Look up into IRQ descriptor table for information of driver handler
3) When valid driver handler is found, invokes signal management routines and driver handler.

DRIVER INTERRUPT HANDLER

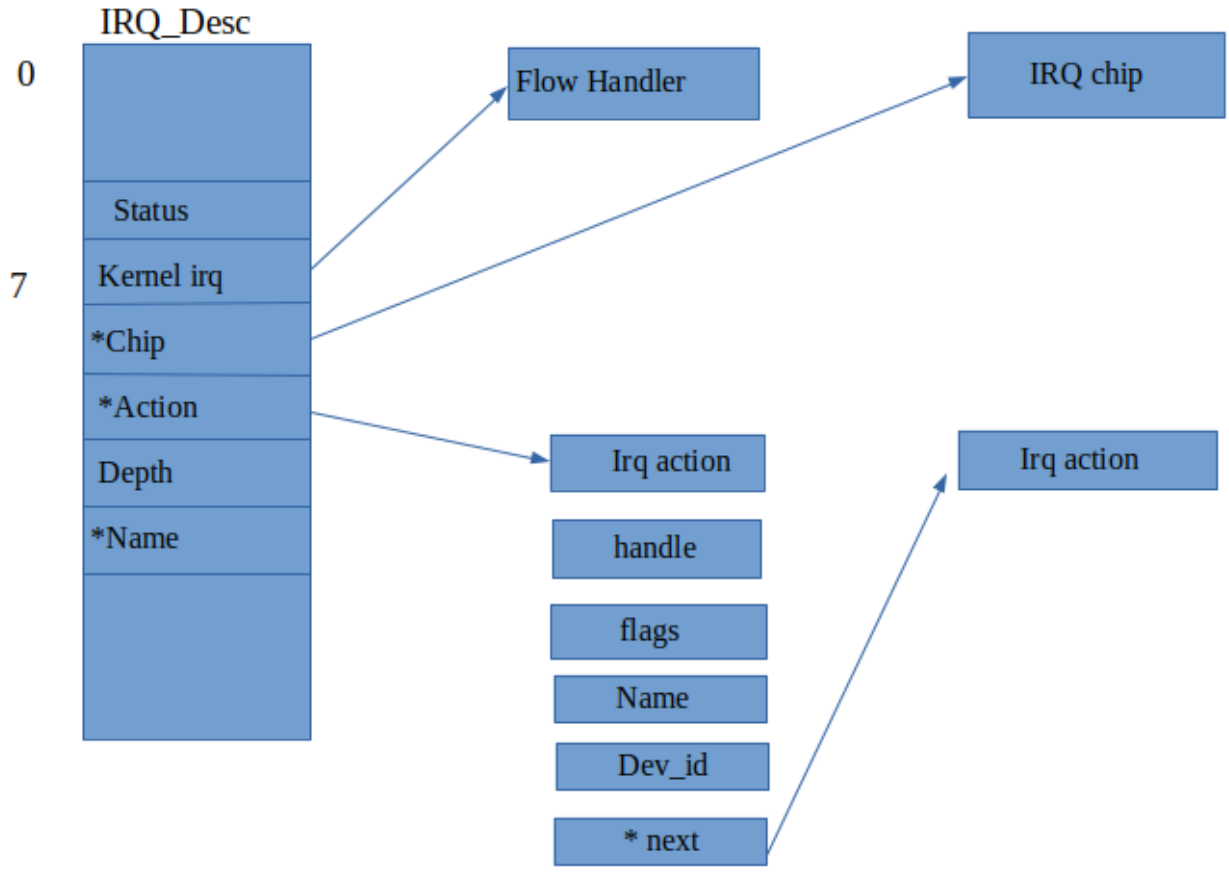
* Driver interrupt handlers register with interrupt management subsystem through kernel IRQ number.

* These functions are invoked by **interrupt management layer upon suspending processor scheduler and masking interrupts in local CPU.** Local CPU's interrupts are tuned only on FSPA multi core.

* Driver ISR's must be register as instance of type struct IRQ action with interrupt subsystem.

* IRQ descriptor table is the core data structure which holds references of driver interrupt handlers and Kernel IRQ properties.

IRQ Descriptors



IRQ Descriptors

This data structure cat /proc/interrupts

No of Pins IRQ's IRQ Name of the Interrupt controller flow

* Driver can register ISR routines through a kernel function request IRQ

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

```
static int __init my_init(void)
{
        if (request_irq
            (irq, my_interrupt, IRQF_SHARED, "my_interrupt", &my_dev_id))
                return -1;
        /* arg1: irq no
           arg2: driver's interrupt handler address
           arg3: priority flag
           arg4: name of the driver
           arg5: unique no to identify interrupt handler
         */
        pr_info("Registered IRQ handler\n");
        return 0;
}

 free_irq(irq, &my_dev_id); // release irq
```

Design of interrupt handler

1. Case study on design of network controller drivers interrupt handling path for reception events

isr routine for processing ingress packet interrupts

```
isr()
{
   step 1: Allocate driver( private  )  buffer.
   step 2: Transfer packet from hardware descriptors to driver buffer
   step 3: processing physical header(Device controller specific header)
   step 4: queue packet for upper layers processing (DLL data link layer)
}
```

fixes to minimize & possibly eliminate packet layer
1) Fully deterministic ISR's
* Ensure interrupt line is disabled for fix intervals
* Design & implement isr's with full determinism (Fixed execution time routines).
2) Interrupt Mitigation mode
* Push network controller into poll mode when first interrupt is received.
* Design interrupt handler to process packets in device buffer through poll condition.
3) Hardware Specific Changes
* Add more processing power (CPU's)
* Partition software stack and assign dedicated CPU's for packet processing

Software options for resource partitioning
* CPU affinity &b interrupt affinity
* Containers (linux)
* Virtualization.

Hardware option for resource portioning / allocations
* Partition hardware to assign dedicated processing power for data processing and control operations.
* Choose network controller (Hardware Changes) with increased reception buffer size.
* Protocol offload engines.

Do’s & don’t s to achieve deterministic execution in ISR’s

don’t :

1. Avoid memory allocation calls.
2. Avoid accessing shared data structures / shared buffers
3. Avoid data copy operations using memcpy or using loop code.
4. Avoid accessing user address space (copy_from_user/copy_to_user, kmap/pad map)
5. Avoid voluntary preemption & delay calls (msleep(), ssleep() , udelay(), mdelay(), schedule())
6. Avoid printk statements
7. Avoid any blocking operations.

do’s:

1. Use pre allocated memory blocks (kmem caches)
2. Engine DMA for data movement b/w device and device buffers
3. Identify interrupt non critical operations and defer them to run later.

Option 1:

Coherent (one-time DMA configuration synchronous DMA) DMA is configured with driver's coherent buffer address during driver initialization.

This mode is generally supported by n/w controller's with local on-chip DMA engines

```
isr()
{
/*  instruct kernel to run deferred work later time */
        schedule_bh(BH);
}
/* Function containing deferred work */
BH()
{
step 3: Process physical header
step 4: Queue packet for upper layer processing ( Data Link Layer )
}
```

Option 2:

Per packet Dma configuration (Asynchronous DMA) This method is applicable for devices wired to peripheral DMA controller(PDC).

```
Isr()
{
1. Check  interrupt status
2. if(INTSTATUS == DMA_COMPLETION )
{
schedule_bh(BH);
}
 if(INTSTATUS == RECEPTION )
{
3. get reference to pre allocated buffer block
4. Configure / start DMA operation
}
return IRQ_HANDLED;
}
```

BH()

{

step 3: Process physical header

step 4: Queue packet for upper layer processing (Data Link Layer)

step 5: Reset DMA registered

step 6: Release DMA buffer

}

BOTTOM HALVES MECHANISM

1) Bottom halves are functions which contains deferred work.

2) Such functions are often required to differ out operations from appropriate function with an objective of optimizing it execution time.

3) Linux kernel providers special scheduler for management and execution of bottom halves functions each scheduler as a defined execution policy and context

Soft irq

Tasklet

Work queues

Soft irq: Available since 2.3

* A set of 32 statically defined bottom halves that can run simultaneously on any processor

- Even 2 of the same type can run concurrently

* Used when performance is critical

* Must be registered statically at compile time.

Usage:

Step 1: Declare a new soft irq

Add new enum entry (Kernel source/ include/linux/interrupt.h

Step 2 : Implement BH function with deferred work and register that as soft irq

open_softirq(MYSOFTIRQ,mybh);

Step 3 : Schedule soft irq fun execution in ISR → raise → softirq (MYSOFTIRQ);

* Kernel maintain per-CPU pending counter of soft irq's raise soft irq function schedules specified soft irq by setting appropriate bit of the pending queue on local CPU.

*Scheduled softirq's or run by do_irq with interrupt lines enabled.
do_irq(conceptual flow)
1. Disable local process scheduler
2. Disable interrupts concurrent CPU
3. Run registered ISR's
4. Enable interrupts (current CPU)
5. Run pending softirq's → do_softirq()
6. Enable process scheduler.

*Bottom halves are allowed to reschedule themselves
* Rescheduling is allowed to ensure optimal utilization of CPU time.
Linux allows reschedule is allowed for reasons shown below

```
BH()
{
void *ptr = kmalloc(4086,GFP_ATOMIC)
if(!ptr)
raise_softirq(MYSOFTIRQ);
        /* Rest of work */
}
```

Bad code as show below

```
mybh()
{
step 1 & 3
raise_softirq(MYSOFTIRQ);
}
```

unconditional rescheduling can cause soft lock ups
Interrupt Latency = CPU latency + kernel latency + driver latency
1. Hardware latency
- Time consumed by CPU to initialize entry handler (jmp int vector)
2. Kernel latency
- isr initialization
- BH initialization

3. Driver latency

- Time consumed by isr

- Time consumed by BH.

MAX_SOFTIRQ_RESTART 10

Solution :

To avoid such posibilities do_irq is implemented with max limit conjunctive softirq runs (MAX_ SOFTIRQ_RESTART)

1. Disable local process scheduler

2. Disable interrupts (on local CPU)

3. Run register ISR's

4. Enable interrupts (local CPU)

5. Run pending softirq's

do{

1. Look up pending bits

2. Unset pending bits and run corresponding softirq bh function

3. Decrement restart counter

} while(restart_counter > 0)

6. Enable process scheduler

Conclusion

1. Softirq's are executed by do_softirq() called by do_irq on return from ISR's with irq lines enabled (interrupt context)

2. Pending softirqs are checked and run also by kernel thread ksoftirq/cpuid (process context)

Limitations

→ Softirq's bottom halves are concurrent which requires them to be programmed as re-entrant routines

→ Re-entrant softirq BH routines will have direct impact on total interrupt latency of the system (can cause escalation).

→ Higher interrupt latency results in process CPU starvation impacting performance of the system as a whole.

Tasklet : (Available since 2.3)

* Are build on top of softirqs

* Two different tasklets can run simultaneously on different processors

→ But 2 of the same type cannot run simultaneously

* Used most of the time for its ease and flexibility

* Code can dynamically register tasklets

Usage

Step 1: Declare and initialize a tasklet is represented by an instance of struct tasklet_struct

Step 2: There are two reasons provided to declare and initialize tasklets

DECLARE_TASKLET(name,function,data)

* Declare a tasklet (of type struct tasklet_struct) with the given name, function and (unsigned long) data value (as the argument when function is later called)

DECLARE_TASKLET_DISABLED (name,function,data);

-Declares a tasklet but with initial state “disabled”, it can be scheduled but will not be executed until enabled at some future time.

Step 3: Schedule tasklet for execution void tasklet_schedule(struct tasklet_struct *t)

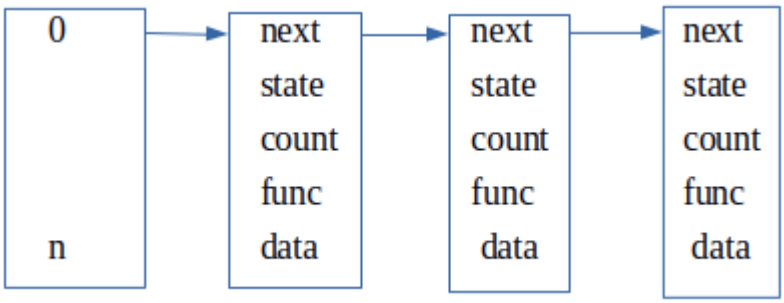
or

void tasklet_hi_schedule(struct_tasklet_struct *t);

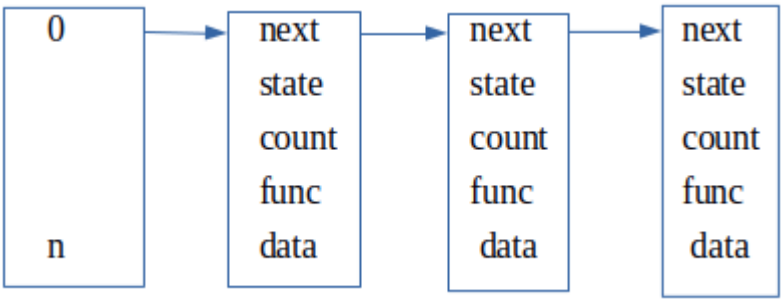
- Both functions add to the beginning of corresponding tasklet list and raise softirq

Kernel maintains two pending lists of tasklets for each CPU, a tasklet must be schedule into either of the list

tasklet_vec(NR_CPUs)



tasklet_hi_vec(NR_CPUs)



static void t_fun(unsigned long t_arg);

static DECLARE_TASKLET(t_name,t_func,0);

tasklet_schedule(&t_name);

* Kernel doesn't provide any guarantee that a bottom halve will run same no of time as it's schedule by the to halves.

* It is often assumed that bottom halve must execute the same no of times as top halve.

* Since bottom halve is executing interrupt non-critical work which deals with processing of event data gather by interrupt handlers (It is not required BH function to executed the same no of times as ISR or top halves)

Interrupt non critical → bottom halves

* If bottom halves functionality is requires execution of each hardware interrupt it is carrying interrupt critical code which must not be in bottom of context.

* Threads can be engaged as bottom halves mechanism.

* Kernel provides an interface for creation of threads through which bottom halves can be deployed.

tsk = kthread_run(t_fun, NULL, "mykthread");

/* arg1: thread routine

* arg2: Arg to thread routine

* arg3: Name to indentify thread

*/

Kthread is terminated with

kthread_stop(tsk);

* A Kernel thread can be put into wait using conditional wait counters

* static DECLARE_WAIT_QUEUE_HEAD(wq);

```
static atomic_t cond;
/* set atomic counter and wake up thread (BH) */
        atomic_set(&cond, 1);
        wake_up_interruptible(&wq);

static int t_fun(void *t_arg)
{
        do {
                atomic_set(&cond, 0);
                /* wait until either thread is suspended or an atomic
                   counter is set */
                wait_event_interruptible(wq, kthread_should_stop()
                                        || atomic_read(&cond));
                pr_info("Entering t_fun, pid = %d ,context of %s\n",
                        current->pid, current->comm);

        } while (!kthread_should_stop());
        return 0;
}
```

Limitation

Problem :

* When kernel threads are directly engaged as bottom halves execution of threads would directly impact on performance of user mode application.

* Have no of kernel threads increase kernel utilization of CPU time increases

* Thread bottom halves allow differed in the process context which provides flexibility for execution of blocking bottom halve work.

* Kernel provide a scheduler called work queue which is designed to run bottom halves in process context through a set of threads spanned dynamically by work queue scheduler.

* For each CPU a pool of threads are enumerated which are responsible to execute bottom halve work different into linked list (Work queue).

Step 1: create struct work_struct instance and initialize it with BH routines

```
static  DECLARE_WORK(work, (work_func_t)w_fun);
```

Step 2: schedule work object by adding it to workqueue(shared).

```
schedule_work(&work);
```

kworker thread will our bottom halve list.

* Work queue interface allows private queues for quicker execution of differ work.

* Kworker threads are programmed to clear private queues before running bottom halves in shared queue

```
static struct workqueue_struct *my_workq;
static struct work_struct work;
static int __init my_init(void)
{
 my_workq = create_workqueue("my_workq");
        INIT_WORK(&work, w_fun);
        queue_work(my_workq, &work);
}
static void __exit my_exit(void)
{
        pr_info("%s: Executed.\n",__func__);
        destroy_workqueue(my_workq);
}
```

Kernel Drivers

Kernel tree measurement:

1. Kernel progress programmable internal timer chip of the platform to generate HZ no of interrupt per second (HZ by default is 100 most arch)

2. On each timer interrupt kernel increments a global counter (unsigned 64-bit) jiffies by 1 (Each incrementation of jiffies counter is called “Kernel tick” and it’s frequency depends on HZ parameter.

3. On SMP/mutlicore systems primary CPU (CPU/0) timer is used for counting jiffie ticks

-CPU/0 timer is default system timer.

4. Kernel functions can access jiffies as read only counter.

5. All time based operations in kernel are carried out on the basis of jiffies stamp.

EX:

1. Progress scheduling

2. Execution of periodic / single shot timer routines.

3. Bounded wait calls

4. Network protocol state machines (TCP , SCTP ..)

5. All other time related operations

6. All kernel functions must treat jiffies as current time.

7. Kernel header file <linux/time.h> provides functions to convert jiffies stamp into user representation formats and vice-versa. (Struct timeval , struct time spec)

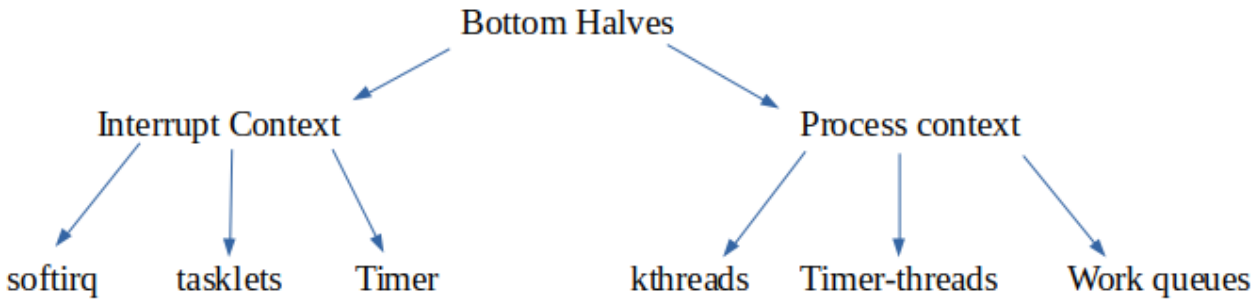
```
static struct timer_list my_timer;
init_timer (&my_timer);     /* intialize */
        my_timer.function = my_timer_function;
        my_timer.expires = jiffies + HZ;    /* one second delay */
        my_timer.data = len;
        pr_info ("Adding timer at jiffies = %ld\n", jiffies);
        add_timer (&my_timer); // timeout absolute


mod_time ( timer,expires )
{
 del_timer (&my_timer)) ; timer→expires=expires;
add_timer(timer);
}
```

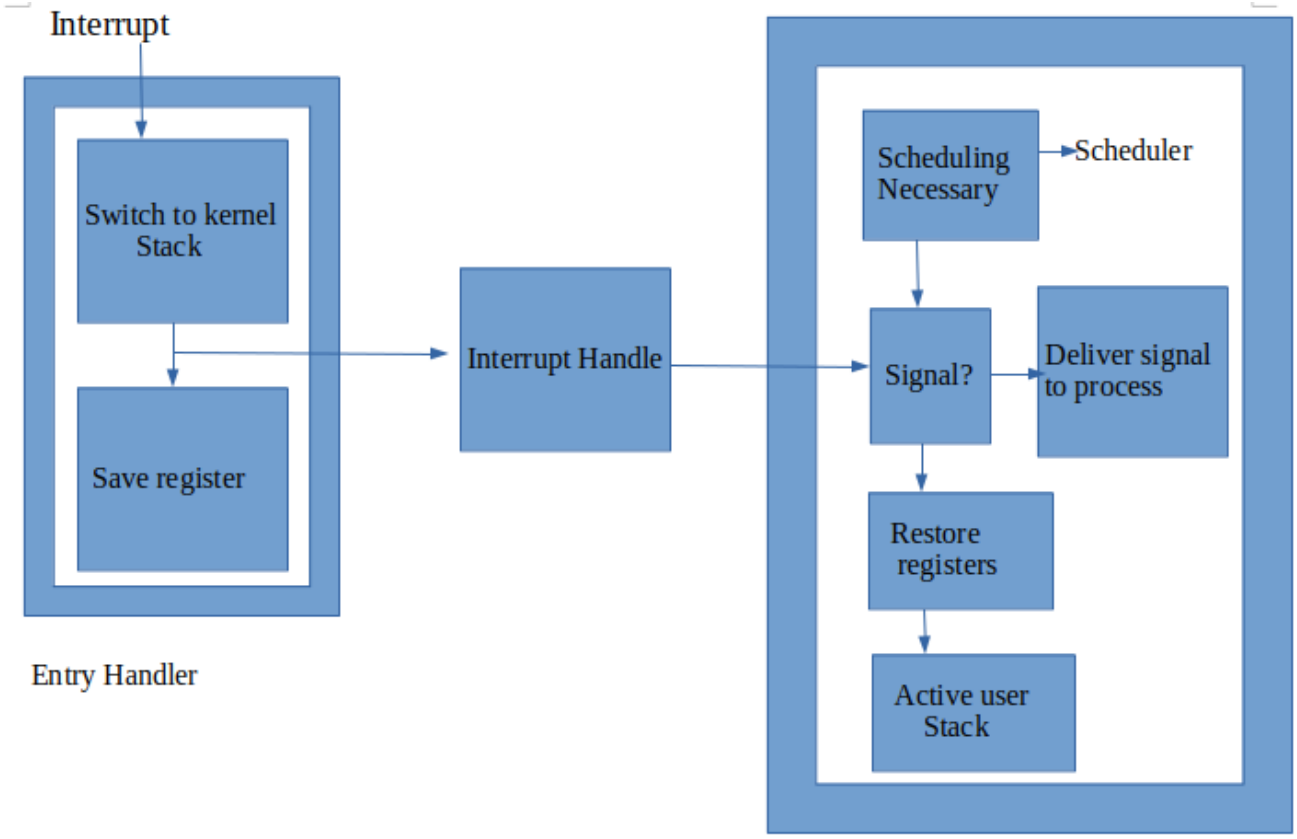
This function returns whether it has modified a pending timer or not (I.e,

mod_timer () of a inactive timer return 0;

mod_timer () of a active timer return 1;

Timers are executed by timer softirq





1. Execution of driver interrupt handlers and bottom halves interrupt context results in variable time interrupt latency.

2. To achieve deterministic interrupt latency both interrupt handlers and bottom halves can be moved to run under process scheduler.

3. Execution of interrupt handler under process scheduler enables interrupt proprieties to be software defined.

int request_threaded_irq(unsigned int irq, irq_handler_t handler,irq_handler_t thread_fn, unsigned long irqflags, const char *devname, void *dev_id);

If you want to set up a threaded irq handler for your device then you need to supply @handler and @thread_fn. @handler is still called in hard interrupt context and has to check whether the interrupt originates from the device. If yes it needs to disable the interrupt on the device and return

IRQ_WAKE_THREAD which will wake up the handler thread and run @thread_fn.

```
static int __init my_init(void)
{
	if (request_threaded_irq
	   (irq, my_interrupt, my_intthread, IRQF_SHARED,
	    "my_interrupt", &my_dev_id))
	      return -1;
	pr_info("Registered IRQ handler\n");
	return 0;
}
static irqreturn_t my_interrupt(int irq, void *dev_id)
{
	/* ensure interupt is valid and handover control to thread handler */
	return IRQ_WAKE_THREAD; // It should return this
}
```

* When thread handlers are assigned to handler hard interrupt corresponding irq line. Most be suspended until the completion of thread handlers.

* IRQ flag IRQF_ONESHOT keeps the corresponding interrupt line disable after completion of thread handler.

The following are other implement significant irq flags.

*** IRQF_NO_THREAD -** Interrupt cannot be threaded. This flag must be applied while setting up an interrupt handle that must be respond to interrupts triggered by low latency device (keyboards, touchscreen HID devices , timers)

*** IRQF_NOBALANCING -** Flag to exclude this interrupt from irq balancing

This flag finds interrupt handler to one of the CPU core usually CPU/0

*** IRQF_NO_SUSPEND -** Do not disable this IRQ during suspend

This flag ensures wake-on-interrupt for the corresponding devices interrupt events are responded even if system is put in the deep suspended mode.

*** IRQF_COND_SUSPEND -** If the IRQ is shared with a NO_SUSPEND user, execute this interrupt handler after suspending interrupts. For system wakeup devices users need to implement wakeup detection in their interrupt handlers.

*** IRQF_PERCPU -** Interrupt is per cpu all static data declared with in an interrupt handler is strictly for CPU. Interrupt stack (IRQ stack) is per CPU.

chrt command manipulate proprieties of the Process

* Interrupt handlers can also be programmed to raise signals on a user process to notify occurrence of interrupt event.

* UIO subsystem provide an interface through which application can receive device interrupt notification.

```
static struct uio_info *info;
static struct device *dev;
static int __init my_init(void)
{
	dev = kzalloc(sizeof(struct device), GFP_KERNEL);
	dev_set_name(dev, "my_uio_device");
	dev->release = my_release;
	device_register(dev);static int __init my_init(void)
{
	dev = kzalloc(sizeof(struct device), GFP_KERNEL);
	dev_set_name(dev, "my_uio_device");
	dev->release = my_release;
	device_register(dev);

	info = kzalloc(sizeof(struct uio_info), GFP_KERNEL);
	info->name = "my_uio_device";
	info->version = "0.0.1";
	info->irq = irq;
	info->irq_flags = IRQF_SHARED;
	info->handler = my_handler;

	if (uio_register_device(dev, info) < 0) {
//		device_unregister(dev);
//		kfree(dev);
//		kfree(info);
		pr_info("Failing to register uio device\n");
		return -1;
	}
```

```
        pr_info("Registered UIO handler for IRQ=%d\n", irq);
        return 0;
}
        info = kzalloc(sizeof(struct uio_info), GFP_KERNEL);
        info->name = "my_uio_device";
        info->version = "0.0.1";
        info->irq = irq;
        info->irq_flags = IRQF_SHARED;
        info->handler = my_handler;

        if (uio_register_device(dev, info) < 0) {
//              device_unregister(dev);
//              kfree(dev);
//              kfree(info);
                pr_info("Failing to register uio device\n");
                return -1;
        }
        pr_info("Registered UIO handler for IRQ=%d\n", irq);
        return 0;
}

User space : open(“/dev/uid0”,O_RDONLY)N<0
                        if ( while(read(fd,&rint,sizeof(rint)>0) // read when interrupt occurs.
```

Kernel implements mutexes and spinlocks for synchronization of concurrent code path.

Mutexes:

Mutexes are mute locks.

Initializing a mutex statically

DEFINE_MUTEX(name);

or

Initializing a mutex dynamically

void mutex_init(struct mutex *lock);

Interface provides interruptable and uninterruptible wait versions of locking api's

uninterruptible:

1) void mutex_lock(struct mutex *lock);

Tries to lock the mutex sleeps otherwise caution, can't be interrupted, resulting in processes you cannot kill.

Interruptable:

2) int mutex_lock_killable(struct mutex *lock); // preferable

same but can be interrupted by a fatal (SIGKILL) signal, if interrupted returns a non zero

3) int mutex_lock_interruptable(struct mutex *lock); // preferable

Same but can be interrupted by any signal.

4) Non blocking lock acquisition can be attempted through

int mutex_trylock(struct mutex *lock);

5) lock can be released through

void mutex_unlock(struct mutex *lock);

6) State of mutex can be verified through

mutex_is_locked(struct mutex *lock);

SPINLOCKS

Spinlocks are poll

spinlock_t my_lock = SPIN_LOCK_UNLOCKED;

or

spinlock_t my_lock;

spin_lock_init(&my_lock);

Functions:

spin_lock(spinlock_t *lock);

spin_unlock(spinlock_t *lock);

Notes:

* Spin lock ops are not interruptable deadlocks are easy to create.

* Both spinlocks & mutexes api implements their locking functions two disable preemption on the local processor. When lock acquisition succeed.

* This ensures continuous CPU time to the owner process. When the lock is released preemption is re enabled.

* Disabling kernel preemption with in critical section results in reduced lock contention and elements the possibility of priority inventions due to lock contentions

* Voluntary (like sleep), wait (like read call) , interrupt preemptions are still possible.

Spinlocks and interrupts

Imagine this scenario

Your driver holds a spin lock the device interrupts

your drivers interrupt handler runs on the same processors

and it tries to take the lock the processor is now hung.

Spinlocks disable kernel preemption blocks some deadlocks scenarios but interrupts are not disabled by default. If attempts are a possibility you must disable them.

Kernel provides function interfaces to disable hard interrupts and soft interrupts (BH)

local_irq_enable()

local_irq_disable()

disable_irq(irq); // system level

enable_irq(irq);

static inline void local_irq_disable(void)

{

local_bh_disable_ip(_THIS_IP_SOFTIRQ_DISABLE_OFFSET);

}

local_bh_enable(void);

Spin lock api provides locking & unlocking interfaces which can turnoff /on local irq's and bottom halves

void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);

Disable interrupts saves previous state note flags is passed. "by value"

void spin_lock_bh(spinlock_t *lock);

only disables software interrupts.

Unlocking api's

void spin_lock_irqrestore(spinlock_t *lock, unsigned long flags);

void spin_unlock_bh(spinlock_t *lock);

Use the one which matches the locking function.

EMBEDDED LINUX PORTING & PROGRAMMING

Linux System Bootup on X86 Class Machines

BIOS (Static memory)

- CPU initialization and POST
    - -Configure clocks
    - -Cache initialization
    - -POST
        - Checks and confirms intergrity of the critical system devices
        - Timer IC’s
        - DMA Controller
        - Video ROM
        - Serial/ Parallel ports
        - Keyboard
    - -Memory Init
        - Intitialization of memory controller (DRAM controller)
        - Verification & calibration of physical RAM

- System Bus Initialization (PCI) (default for PC)
    - Initialization PCI host/ bridge controller
    - PCI probe
- Initialize disk controllers
    - sata
    - USB
    - IDE

*BIOS loads boot programs from configured storage device

* PC storage devices contains MBR (Master Boot Record) which is used to store boot program of the system.

- Loads boot loader from MBR (configured Boot Device)
    - -Master Boot Record first sector of boot disk
        - -OS loader (first 446 bytes)
        - -Partition table info (64 bytes)
- MBR signature / Magic no (2bytes)

* Linux systems use grub as a boot program

* GRUB is designed as a 3 stage boot loader capable of accessing windows & Linux disk partition.

GRUB

-Stage 1

Resides in MBR

Invokes stage 1.5

-Stage 1.5

Resides at MBR gap 0-63 sectors before start of 1st partition (30 KB)

Initialize file system images

lists valid boot images

Invoke stage 2

-Stage 2

Initialize stack and load kernel image

Invoke kernel boot-strap

OS Kernel

-Boot strap

MMU init, setup page tables

Saves kernel args into kernel variables (DT address)

Invokes kernel's main routine (start_kernel)

-Protected-mode kernel

-Start_kernel

- Starts Kernel subsystem initialization

- Setup_arch()

Enumerates physical memory info with mem block layer

Parse kernel boot parameters

Invokes target specific platform device enumeration routines

Setup memory zones info

- High level MM
- Driver Stacks
- Process Manager
- File systems
- Rootfs mount
- Initializes kernel threads

- init
- kthreadd

init, kthread loads first user space binary also with name init from rootfs

* init process of user mode is responsible for initialization of demons (user mode) start up processes (shell, login) ,load shared objects (pre-loading by default librarys (ex ldd))/

* Based on the type of platform the application framework configured, configuration of init process varies

init thread

- initialize first user programmable

User space init implementations and frame works

- system v init (seqence, older unix init level 1-6 like that)
- upstart init (ubuntu concurrent)
- Android init
- Systemd init
- initg
- Openwrt init

Embedded Linux System Boot

Hardware: System-on-chip + Peripheral interface

(Soc = CPU + MMU + memory controller + interrupt controller + DMA + Bus controllers + device controllers)

-Boot-ROM

-Resides at reset vector

-Initializes following

CPU clocks

CPU caches

Static Memory

Boot device controller

-Any or all of the following

Nor flash, Nand flash, SD/MMC controller

Initialization of memory controller or Dynamic Memory initialization (DRAM controller) (optional)

Load Boot loader (FSBL or SSBL) into pre-configured address space.

-FSBL (First Stage Boot Loader)

- Loaded into static RAM from storage

-Initialize DRAM Controller

- Load SSBL into memory

- SSBL (Boot loader)

-Resides of on-board stroage

- Loaded into dynamic memory by either FSBL or Boot-ROM

- Initializes following

- Console

- Storage device controllers

-USB

-SD/MMC

- Network Controllers

- Allocate Address Space for Kerenel

-Fetch & load kernel, BSP(DT) Images from pre-configured storage or network into memory

-Setup kernel parameters

-Invokes Kernel’s boot-strap with board information.

- Kernel boot (Same like x86)

Cross Compiler Tool Chains

->To build linux boot images on a host system an SDK or cross tool chain is required to be present on the host.

* Cross tool chains compresses of following

1. Binary tools for compile and build of application, kernel, boot loader sources.
2. A set of libraries required for application build.
3. Run time libraries which implements ABI standard.
4. Support libraries for dynamically link loader.
5. Kernel headers (System calls)
6. Debug & Placing tools

→ GNU free software community hosts build & compiler tools under different source projects.

1. Compiler suite – GCC compiler project.
   - -Pre processor
   - -Compiler
2. Build tools - Binutils
   - -Assembler
   - -Linker
   - -Link loader
   - -objdump
   - -nm
   - -readelf
   - -strip
3. Standard library: Most include support for standard API can be acquired from appropriate open/free software projects as per application compatibility, needed optimizations and distribution license.

Library options

Glibc(gnu.org / software / libc)

- -Standard GNU C lib
- -Most complete implementation of POSIX
- -Support for most embedded architectures
- -Highly configurable

Ulibc(oclibc.org)

-Micro controller (library)

-Fully configurable

-Low memory foot print

-Lacks full support for POSIX standards.

Other

-Bionic, diectlibc, nwelib,

4. Kernel headers

-Generate Kernel headers from source

make header_install

-Kernel header package available for download (Kernel org)

5. Debugger tools

GDB

GPROF

clfs.org

1. Installing tool chain

A. Custom building

B. Install pre-build tool chain

A. Cross tools chain custom build methods creating target specific tool chain can be carried out through any of the following methods

1. Manual integration

-clfs.org

2. Automated Build

-Buildroot (shell script)

-Cross tool-ng

-Open embedded

-Yocto (Python)

B. Ready to use pre-built tool chain packages

-Target SOC or board vendor specific optimized / test tool chain

- A consortium dedicated to provides tools for a given SOC/ board

Ex: linaro.org

-Third party vendor licensed tools

-Mentor graphics, Time sys, Marvel

-Cross tool packages from desktop linux distributors

apt-get install gcc-arm-linux-gneabi

Note: Custom tool chains are preferred over pre-build tool chains for better kernel & application compatibility.

make toolchain // create new tool chain

dl // All downloaded files

output/host/bin // Cross tools

make sources // Download only

make // Create all images & too chain

Compiling and building boot images

* Boot loader

U-boot project

* Support for most embedded processor architectures
* Support for wide range of boards
* Active & rich community of developers
* Implements both stage 1 and stage 2 boot loader facilities

-Build system can generate both stage1 and stage 2 images

* Support for secure boot and multi boot images
* Full support for open firmware device trees (DT)
* Standard customization interfaces.

Barebox project

- Extensive support for ARM board projects

-Well defined source free layout

-Identical to linux source tree organization.

-Well defined customization interfaces.

-Driver models, BSP model identical with linux kernel.

-Stand kconfig

- Command interface creations linux compatibility.

Sources /uboot-2017.05/

Step 1: Set path to cross tool chain

PATH=$PATH:/home/raghu/

Step 2: Verify board config file

u-boot/configs <- folder arm335x_boneblack_defconfig

Step 3: Apply board config

make ARCH=arm arm335x_boneblack_defconfig

make ARCH=arm menuconfig

make ARCH=arm CROSS_COMPILE=arm-linux-

stage1(MLO) stage2 (boot loader) – uboot.bin u-boot.img

cd linux-4.19-32:

Verify board support config make ARCH=arm help

* make ARCH=arm omap2plus-def-config

* make ARCH=arm menuconfig

compile & build

make ARCH=arm CROSS_COMPILE=arm-linux- dtbs (Device tree)

cd arch/arm/boot (zImage file)

dts ← folder for device tree

cd arch/arm/boot/dts/ All dtb file

am335x_boneblack.dtb

Rootfs

1) File system Images are needed to separate application binaries from kernel image, this separation allows apps to be loaded into memory as files and allows kernel to setup address space for each application.

2. Rootfs is a hierarchy of folders which contain application binaries and other support files that apps might need at run time. (libraries, app name space for new files, mount points for various logical file systems like proc, sys, devtmpfs)

3. Must unix systems follow a standard hierarchy for rootfs to ensure application portability for embedded linux systems structure of root is fully customisable as per application configurations.

4. rootfs images could be deployed into memory as ramdisks or can be mounted from persistent storage devices like nand-flash mmc or ssds.

5. Location of rootfs is passed to kernel at boot time as a boot parameter by board BSP (Either through u-boot config or through device tree script)

Components of rootfs

* Rootfs must contain the following

1. Application binaries
2. Tools and utilities
3. Dynamic libraries
4. User mode initialization frame work (init system v)
5. Mount points for important logic file system (/proc, /sys/ , /dev/modules)
6. Kernel modules (/lib/modules)

Creating rootfs

Step 1: Create a work space folder

Step 2: Create folders as per choosen hierarchy bin dev etc lib

* Populate required application binaries into bin & sbin folder

* To include open domain tools and utilities along with preferred init system a frame work called busybox is used.

Step 3: Busybox a framework which maintains source code of various applications tools utilities found on unix systems.

Step 4: Busybox provides a build system which provides configuration interface to select required tools

busybox.net

make CROSS_COMPILE=arm-linix CONFIG_PREFIX=$(path/to/rootfs) install

Step 5: Populate /etc with initialization scripts busybox provides a default init framework. Which sequentially carries out user mode initialization

* Busybox init depends on a start up script called inittab which is to be placed in etc

* The following commands supported by busybox init.

Script format

id:runlevel:action:process

id - specify which tty action is to be carried out

runlevel - mode in which operation is valid (Busybox ignores this)

action – Action to be initiated

process – Arguments for the action command

Action commands

1. sysinit : run specified action once during startup.
2. respawn : ensure specified process is started during boot and respawn if its killed.
3. askfirst : run specified action but prompt for user confirmation.

4. wait : run specified process and wait for its completion.
5. once : discard specified script after once run.
6. ctrlaltdel : specify action to run when ctl+alt+del is hit.
7. shutdown: action to run on shutdown.
8. restart : action to run carried out on hotboot.

Sample inittab

```
# startup the system
null:sysinit:/etc/init.d/rcs
# start getty on serial port for login
ttyS0::respwan:/sbin/getty -L ttyS0 115200 vt100
# script to do before rebooting
null::shutdown:/bin/umount -a -r
```

* It is common practice to divert all initialization operations into other scripts (rc scripts in most systems) for ease of use & maintenance.
* rc scripts are interpreted by “linuxrc” which is internally used by init.
* rc scripts have to all shell scripting compatibility.
* rc scripts are placed in

/ etc/rc.d/init.d/ or / etc/init.d/ directory.

Kernel → init → inittab → rc scripts

```
   User
    |
veda: :0:0:root:/root/:/bin/sh ← shell
    |          |
 Password   Login folder
```

Populate lib folder :

libraries are copied from cross tool chain rootfs.

```
cp -rf p $(output/target/line/* $(rootfs)lib
```

Populate kernel modules

make ARCH=arm CROSS_COMPILE=arm-linux- modules

make ARCH=arm CROSS_COMPILE=arm-linux- INSTALL_MOD_PATH=$(PATH_TO_ROOTFS) modules-install.

YOCTO

*Yocto is a set of tools and methods that allow users to build custom embedded linux based systems (a distribution) regardless of the hardware architecture.

* Yocto project like build-root can give the same end products but with flexible package management and SDK.

* Boot loader

* root file system image for the embedded device

* Kernel

* Compatible tool chain

* Package Management system

* Core management of yocto build system

→ Poky

→ Open embedded core

→ Bibake

1. Open embedded core:

* Open embedded is a software framework used for creating linux distributions aimed for but not restricted to embedded devices

* It uses meta information (Called recipes) for downloading compiling generation target device specific software package on host

* OE core servers as a base layer for yocto build system

* OE core contain base layer of recipes classes and associated file that is meant to be common among many different open embedded derived open including the yocto project.

2. Bitbake

→ Bitbake is a stack of python and it core build engine of yocto

→ Bitbake is responsible for passing the meta data, generating list of tasks from it and other executing those tasks.

→ If requests configuration file and recipes also called meta data to perform schedule a set of tasks, to download configure and build specified package and file system images, it also determines any dependencies

→ The primary types of metadata files exits all of which are parsed by bit bake.

* configuration files (*.conf)

* Classes (*.bbclass)

* recipes (*.bb)

Other Metadata

* Append files (*.bbappend)

* Include files (*.inc)

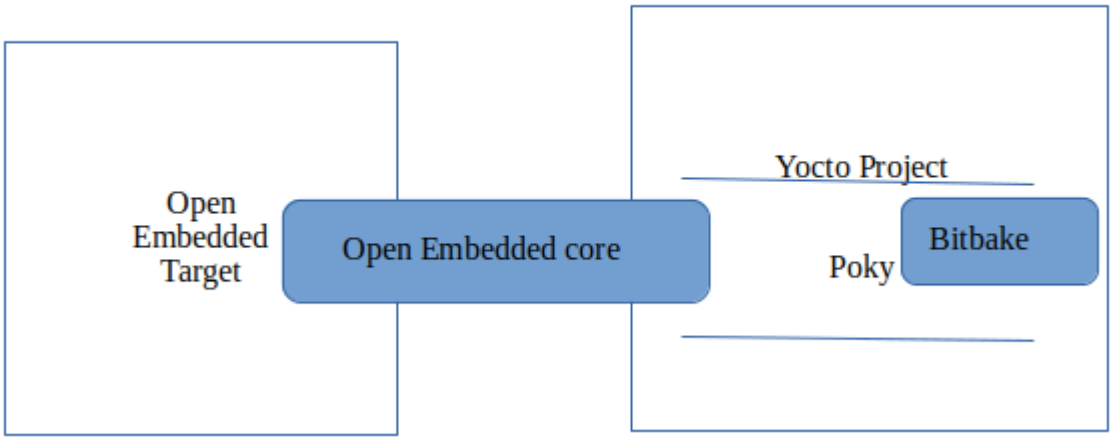
3. Poky:

* Poky the reference system. It is a collection of projects and tools, used to boot-strap a new distribution based on the yocto project.

* Poky helps to build a tailored (linux distribution / product according to the each needs.

* It supports ARM, WPS, x86, PPC architectures.

Yocto Open embedded POKY



To download the poky reference by git clone -b thud git://git:YoctoProject.org/poky.git

Contains folders key files

Bitbake: Holds all scripts used the bitbake command usually matches the stable release of the bitbake project.

meta: Contains the open embedded core metadata

meta-skeleton: Contains template recipes for BSP and kernel development

meta-poky: Holds the configuration for the poky reference distribution.

meta-yocto-bsp: configuration for the yocto project reference hardware board support package.

Oe-init-build-env: Script to set up the open embedded build environment. It will create the build directory. It takes an optional parameter which is the build directory name. By default this is build. This script has to be sourced because it changes environment variables.

Scripts: Contains scripts used to set up the environment development tools and tools to flash the generated image on the target

Build Directory

Build directory contains configuration scripts and output folder the following are elements of output folder. /tmp directory /build directory

conf: Configuration files image specific and layer configuration

downloads: Downloaded upstream tar balls of the recipes used in the builds

sstate-cache: Shared state cache used by all builds

tmp: Holds all the build system output.

tmp/buildstats: Builds statistics for all packages bulit (CPU usage, elapsed time, host, time stamps …)

tmp/deploy: Final output of the build.

tmp/deploy/images: Contains the complete images built by the open embedded build system. These images are used to flash the target.

tmp/work/: Set of specific work directories, split by architecture they are used to

unpack, configure and build the packages contain the patches sources generated objects and logs.

tmp/sysroots: Shared libraries and header used to compile applications for the target

but also for the host (tool chain)

Bitbake

The following are various flags can be passed to bitbake.

-c < task > Execute the given task

-s List all locally available packages and their versions

-f Force the given task to be run by removing its stamp file.

World Keyboard for all recipes

-b < recipe > execute tasks from the given recipe (without resolving dependencies)

Recipes:

Recipes are configuration files parsed by bitbake and recipe describes the following

1. Description of the package
2. Location of source directory
3. Compile flags
4. Install directory

Each of the above is referred to as the task.

The following is the structure of the recipe

* The header : What / who
* The sources: Where
* The tasks: How

Syntax: Recipes can use the following operators

= Expand the value when using the variable

:= immediately expand the value

+= append (with space)

=+ prepend (With space)

.= append (without space)

=. prepend (without space)

?= assign if no other values

?? Same as previous, with a lower precedence

* Avoid /using += , -+ and =. in $BuildDIR/conf/local.conf due to ordering issues

*If += is parsed before ?=, the latter will be discarded

* Using append unconditionally appends the value

Header:

SUMMARY=”simple hello world application”

SECTION=”cust app”

LICENSE=”MIT”

LIC_FILES_CHKSUM=”file://${COMMON_LICENSE_DIR}:MIT:md5=” “

License files (mandatory)

*Header must include license file along with it’s md5 checksum

* Collection of license files can be found in poky

    * meta/files/common-licenses

* File’s md5 checksum can be computed.

    *$md5sum meta/files/common-licenses/<license files >

    for eg: $md5sum meta/files/common-licenses/Mit

SRC_URI=”file://helloworld.c”

S=”$(WORKDIR)”

* SRC_URI defines where and how to retrieve source files. It s a set of URI schemes pointing to the resource locations (local or remote)

* URI scheme syntax: scheme://url; param1;param2

* Scheme can describe a local file using file:// or remote locations with https://, git://, svn:// , ftp:// …

Git scheme:

git://<url>; protocol =< protocol > ; branch=< branch >

eg: SRC_URI=”git://git pengutomix/barebox; branch =$(SRC_BRANCH}” SRC_BRANCH?? =”master”

*When suing git it is necessary to also define SRCREV. If SRCREV is a hash or a tag not present in master, the branch parameter in mandatory.

* The http, https and ftp schemes

SRC_URI=https: //download.osgeo.org/libtiff/tiff${pv}.tar.gz

SRC_URI{md5sum}=” ”

SRC_URI{sha256sum}=” ”

An md5 or an sha256 sum must be provided when the protocol used to retrieve the file (S) does not guarantee their integrity. This is the case for https, http or ftp.

Tasks:

Tasks are functions through with specific actions can be defined

*Syntax of a task

```
do_task() { action()
            action1 }
```

Example: do_compile() { oe_run make }

```
do_install () { install -d ${D} ${bindir}
                install -m 0755 hello ${D}${bindir}
                }
```

Note: Default tasks are already exists in the base classes which are reusable

* do_fetch
* do_unpack
* do_patch
* do_configure
* do_compile
* do_install
* do_package
* do_rootfs

You can get a list of existing tasks for a recipe with:

bitbake < recipe > -c listtasks.

Modifying existing tasks

Tasks can be extended with _prepend or _append

do_install_append()

{

install -d ${D}${sysconfdir}

install -m 0755 hello.conf ${D}${sysconfdir}

}

* Sample recipes

1) libraries

meta-custm/recipes-lb/simplelib

DESCRIPTION=”simple library”

SECTION=”custlib”

LICENSE=”MIT”

LICFILES_CHKSUM=”file://${COMMON_LICENSE_DIR}/MIT;” ”

PR=”r0” // revision number

SRC_URI=”file://sample.c”

S=”${WORKDIR}” // // wher to find a file

do_compile{

${CC} -c sample.c -o sample.o

${AR} rcs libsample.a sample.o

}

You need to put files in files directory .bb file file://

${CC} ${LDFALGS} -c FPIC sample.c -o sample.o

${CC} {LDFLAGS} -shared libsample.so sample.o

Kernel Module

Module base class defines reusable task functions which can be used by any kernel module-recipe

SUMMARY=”Example of host to build

LICENSE=”GPLV2”

LIC_FILES_CHKSUM=”file://COPYING;md5=””

inherit module

SRC_URI=”file://Makefile \

file://hello-mod.c /

file://COPYING \

“

S= “${WORKDIR}”

RPROVIDES_${PN}+=”kernel_module_hello”

#kernel-module =

Extending a recipe

* It is a good practice not to modify recipes available in poky
* But it is sometimes useful to modify an existing recipe, to apply a custom patch for example.
* The bitbake build engine allows to modify a recipe by extending it.
* Multiple extensions can be applied to recipe.
* Metadata can be changed added or appended
* Tasks can be added or appended
* Operations are used extensively to add append prepend or assign values
* The recipe extensions end in .bbappend

*Append files must have the same root name as the recipe they extend.

Example_0.1.bbappend applied to example_0.1.bb

* Append files version specific.

Poky	**meta-custom**
meta/recipes-core/init-if updown	recipes-core/init-ifupdown
init-if updown_1.0.bb	init_ifupdown_1.0.bbappend

meta-ti	
recipes-bsp/u-boot	recipes-bsp/uboot
-->u-boot-2014.07.bb <----------	uboot-custom.bb
u-boot-2014.07.bb <----------	uboot-2013.bbappend

include file (.inc) file

Some recipes can be broken up into multiple file each specific to a version of the package.

Components of recipes which are version independent or preserved in a file with extension .inc(include)

This file is included into all version specific recipes

<application> .inc: Version agnostic metadata

<application><version>.bb:require<application>.inc and version specific metadata.

Classes:

1. Classes are files which contain common tasks that can be reused across multiple recipes

2. Class files are identified with extension .bbclass

3. Class files are located in the classes folder of the layer.

4. Recipes can use the common code by inheriting a class

inherit <class>

Open embedded core (oe-core) and poky provide a set of reusable bass classes which are widely used across BSP, init system auto bake packages.

The following are important base classes meta/classes/

base.bbclass

*Every recipe inherits the basic class automatically

* Contains a set of basic common tasks to fetch, unpack or compile applications.

* Inherits other common classes, providing

    *Mirror definitions:DEBIAN_MIRROR, GNU.MIRROR, KERNELORG_MIRROR

    * The ability to filter patches by SRC_URI

    *Some tasks: clean, listtasks, fetch.

* Defines oe-runmake, using EXTRA_OEMAKE to use custom arguments.

Kernel.bbclass

*Used to build linux kernels

*Defines tasks to configure, compile and install a kernel and its modules.

* The kernel is divided into several packages

    Kernel, kernel-base, kernel-dev, kernel-modules.

* Automatically provides the virtual package virtual /kernel.

*Configuration variables are available.

    *KERNEL_IMAGE_TYPE, defaults ti zImage

    *KERNEL_EXTRA_ARGS

*INITRAMFS_IMAGE

Autotools.bbclass

***** Defines tasks and metadata to handle applications using the autotools build system (autoconf, automake, and libtool);

* do_configure: generates the configure script using autoreconf and loads it with standard arguments or cross compilation

* do_compile: runs make

* do_install: runs make install

*Extra configuration parameter can be passed with EXTRA_OECONF

* Compilation flags can be added thanks to the EXTRA_OEMAKE variable

Three keywords can be used to include files from recipes classes or other configuration files

inherit

include

require.

* Inhert is used to include recipes from classes

* Inherit looks for files ending in .bbclass in classes directories found in BBPATH

*It is possible to include a class conditionally using available

inherit ${Foo}

The include and require keyword

*include and require can be used in all files, to insert the content of another file at that location

* If the specified on the include (a require) path is relative, bitbake will insert the first file found in BBPATH

* include does not produce an error when a file can not be found, wheareas require raises a parsing error.

* To include a local file: include ninvador.inc

* To include a file from another location (which could be in another layer);

include patch/to/file.inc

Work flow

* When bitbake command is initiated it visits directories listed in BBPTAH variables to find recipe for the specified directory.

BBPATH: BBPATH is populated with list of directories as specified in bblayers.conf of the build directory.

*Each layer directory must contain a conf folder with layer configuration file that specifies location of for the specified layer and other layer attributes.

The following

meta-veda

classes

conf ---- layer.conf

#We have a conf and classes directory append to BBPATH

BBPATH.=":$(LAYERDIR)"

#WE have a recipes directory add to BBFILES

BBFILES+="${LAYERDIR}/recipes*/*/*.bb $(LAYERDIR}/recipes*/*/*.bbappend ${LAYERDIR}/image/**bb"

BBFILE_COLLECTIONS+="meta-techveda"

BBFILE_PATTERN_meta_techveda:="${LAYERDIR}"

BBFILE_PRIORITY_meta_tech_veda="16"

LAYERSERIES_COMPAT_meta_techveda="warrior"

Package builds can be found in the o/p directory of the build folder

/tmp/work/cortexa8hf-neon-poky-linux-gunebi

individual applcations, library builds

each build contains following important files

1. Binary file application

2. RPM or tmplipk package and license information

→ /tmp/work/(board/SOC)_poky_linux

/tmp/work/beaglebone-poky-linux-gneabi/ contains kernel image, modules, boot loader images and boot scripts of the init subsystem.

BSP layers:

* BSP layers are device specific layers. They hold metadata with the purpose of supporting specific hardware devices

*BSP layers describe the hardware features and often provide a custom kernel and boot loader with the required modules and drivers

*BSP layers can also provide additional software designed to take advantage of the hardware features

* As a layer it is integrated into the build system as we previously saw.

* A good practice is to name it meta-<bsp name >

BSP layers contain machine folder under conf directory

This folder must contain machine folder conf/machine

config file for each hardware platform supported by layer

* These configuration files are stored under meta-<bsp-name>/conf/machine/*.conf

* The filenames correspond to the value set in MA meta-ticonf/.

Creating a BSP layer

* The bitbake layer create layer command helps us create new layers and ensures this is done right

bitbake layers create_layer -p <prority > < layer >

* The layer created will be pre filled with the following files:

conf/layer.conf the layer’s configurations

COPYING.MIT the license under which layer is released by default MIT.

README A basic description of the layer.

By Default all metadata matching (recipes/*/*/*.bb image or the file system.

Kernel recipes:-

* There are two ways of compiling kernel in the yocto project

* By using the linux-yocto packages provided in poky

* By using a fully custom kernel recipe

The kernel used in selected in the machine file thanks to PREFERED.PROVIDER virtual/kernel

Yocto project maintains compatible versions of kernel source packages. Kernel packages main in yocto project are referred to as linux-yocto (Package name)

* The Primary difference b/w main line kernel package and linux yocto is the organization of the source files in git server.

Yocto-kernels are easier to configure patch and build through yocto tools.

Kernel package provider must be selected in the machine configuration file of the BSP layer.

The following variable define package specific metadata (function)

PREFERRED_PROVIDER_virtual/kernel=”linux-stable”

This variable must be initialized with kernel package repository name (by default linux-yocto)

preferred version (PREFERRED_VERSION_linux_stable?=”4.20%”

variable specifies version no of the name should same as the kernel package to be found in the specified repository.

KERNEL_IMAGTYPE=”zImage” (should be initialized with name of the kernel image to be generated).

KERNEL_DEVICE_TREE=

KERNEL_EXRA_FLAGS+=”LOADADDR=${UBOOT_ENTRYPOINT}

IMAGE_FSTYPE=”tar.xz extra”

Kernel recipes must include sources and compiler functions

SUMMARY=”linux kernel”

SECTION=”kernel”

LICENSE=”GPLV2”

LIFILES_CHKSUM+=”file://COPYING;imds=” ”

DEPENDS+=”bc-native;bison-native openssl module_util_linux nature xl_nature”

inherit kernel-base

(linux-stable.inc) file (header)

linux-stable-4.19.16.ble

COMPATIBLE_MACHINE=”beglebone”

KERNEL_DEVICE_TREE?=”am355x_boneblack_dtb.1 \

$(WORKDIR}linux-stable-4.19.16”

PV=”4.19.16”

SRCREV=” “

SRC_URI=”git://git.kernel.org/.pudsum/linux/kernel/git/stable/linux-stable-git/;branch=linux-${LINUX_VERSION}/file://defconfig/

Configuring linux-yocto

Following bitbake commands support kernel build

1. bitbake -c kernel_configure virual-kernel

2. Edit the configuration; bitbake -c menuconfig linux-yocto

3. Save the configuration difference

bitbake -c

The above command applies default configuration file provided in the recipe to edit configuration following command can be used

bitbake -c menuconfig virtual-kernel

To generate configuration reference between default config and menuconfig changes used in that command.

bitbake -c diffconfig virtual-kernel

The difference will be saved at $(WORKDIR/fragment append new configuration fragment to recipes using the below command

bitbake -c kernel-conficheck-linux-yocto

Kernel metadata is a way to organize and the split kernel configuration and patches in little peace of work.

bitbake -c virtual/kernel compile

To compile any kernel package (virtual kernel)

Kernel Metadata

The following configuration variable are supported by linux-yocto repository linux kernel type

For linux-yocto only

Two main configuration variables

LINUX_KERNEL_TYPE standard (default) tiny or premp* ₹/

standard:generic kernel policy

tiny: bare maximum configuration for small kernels

preempt_rt_applies_the PREEMPT_RT patch Kernel with reference support

KERNEL_FEATURE: list of features to enable.

u boot specific

Boot loader configuration baglebone.conf

PREFERRED_PROVIDER_virtual/bootloader=”u-boot”

must be initialize with preferred boot loader provider name of the repo.

PREFERRED_PROVIDE_uboot=”u-boot”

UBOOT_ENTRYPOINT=”0x8000800”

UBOOT_LOADADDRESS=”Ox8000800”

UBOOT_MACHINE=”am335x_boneblack_config”

BOOTENV_SIZE=”0x2000”

EXTRA_IMGEDEPENDST=”u-boot”

SPL_BINARY=”MLO”

UBOOT_SUFFIX=”img”

IMAGE_BOOT_FILES:=”U-BOOT.${UBOOT_SUFFIX} MLO zImage am385x-bone.dtb am335x_bone_black.dtb am35x_bonegreen.dtb

Yocto-SDK’s

* SDK is a set of tools and libraries using which applications and boot images can be build for a specific target platform. Yocto provides SDK generation options to support building application and boot images **without presence of poky.**

Yocto SDK includes a tool chain libraries and other needed tools.

Yocto categories SDK’s into two types

1. A minimal SDK (They called generic SDK support building boot images)

2. An image board SDK which extends minimal SDK with application specific libraries and sys root environment for the target root file system.

3. The SDK’s generated with poky the distributed in the form of a shell script

4. Executing this script extracts the tools and sets up the environment

Type 1:

The generic SDK (Only for boot images)

* Low level development kernel, boot loaders

bitbake meta-toolchain (Generate SDK)

* The generated script containing all the tools for this SDK is in

$(BUILDDIR/tmp/deploy/SDK

Example: poky_glibc_x86_64_meta_toolchain_contexa8hf_non_toolchain_2.5.sh

Type 2: Image Based SDK

To generate a SDK for core_image_minimal

bitbake -c populate_sdk core_image_minimal

The generated script, containing all the tools for this SDK is an $BuildDIR/tmp/deploy/sdk

Example: poky_glibc_x86_core_image_minimal_cortexa8hf_neon_toolchain_2.5.sh

To install an SDK, retrieve the generated script and execute it

The script asks where to install the SDK, defaults to /opt/poky/<version>

Example: /opt/poky/2.5

To set up build environment

cd /opt/poky/2.5

source (Setup script provided SDK)

source ./environment_setup_cortexa8hf_neon_poky_linux_gnuebi

* The PATH is updated to take into account a libraries installed along side the SDK.

Compiling kernel to SDK

* To build an application for the target

$CC -o exmple example.c

SDK environment variables

CC	Full path to the c compiler library
CFLAGS	C flags used by the compiler
CXX	c++ compiler
CXXFLAGS	c++ flags, used by cpp
LD	linker
LDFLAGS	linker flags used by the linker
ARCH	For kernel compilation
CROSS_COMPILE	For kernel compilation
GDB	SDK GNU debugger
OBJDUMP	SDK objdump

* The LDFLAGS variables is set to be used with the compiler GCC

when building the linux kernel, unset this variable

unset LDFLAGS // U-boot and kernel compilation

make menuconfig make

Additional Config Multi config builds

* This feature allows using a single bitbake command to build multiple images or packages

* Each image or package might be for different target requiring different configuration

(Multiple configuration Builds)

* To enable multi config builds we must define each target's configuration separately using a parallel configuration file in the build directory

* Additionally multi config build must be enabled in build directories local.conf file.

Build Directory

| ---- Conf

|------ local.conf

|------ multi config

|------ x86.conf

|------ arm.conf

* Use the BBMULTICONFIG variable in your conf/local.conf configuration file to specify each multi config

BBMUTLICONFIG="x86arm"

* Use the following bitbake command form to launch the multiple configuration build.

$bitbake multiconfig:x86:core-image-minimal multiconfig:arm:core-image-sato

Feature 2:

Recipe Making

bitbake allows hiding recipes

* Masking feature is provided through a special variable BBMASK

*Bitbake ignores any recipe or recipe append files that match values of BBMASK.

The following example use a complete regular expression to tell bitbake to ignore all recipe and recipe append files in the meta-ti

/recipes-misc/ directory

BBMASK="meta-ti/recipes-misc"

* To mask out multiple directories or recipes we can specify multiple regular expression fragments for example.

BBMASK+="meta-ti/recipes-misc/meta-ti/recipe-ti/

Protocol

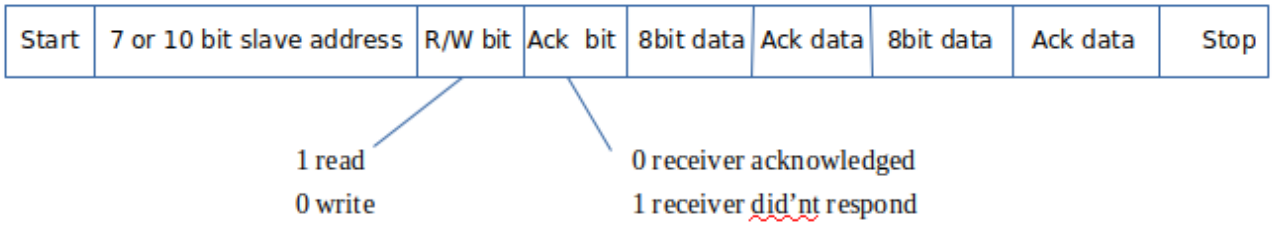
i2c protocol

* Synchronous serial protocol developed by philips
* Half duplex
* Simple two-wire bus interface
* In expensive bus to interface low bandwidth peripherals
* Speed up to 400KHZ with high speed extension up to 5MHZ
* A multi master bus with arbitration and collision detection.

I2c Bus

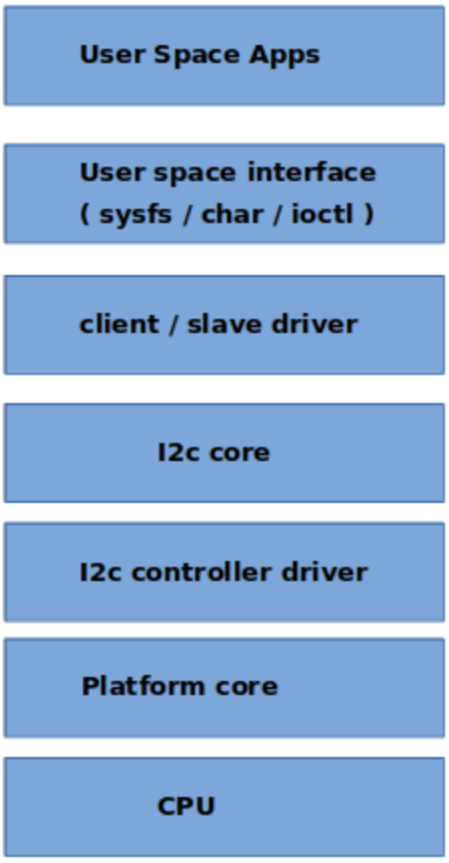
Data transfer on I2c Bus

* Start bit (start condition)
* 7 or 10 bit slave address + 1 R/W bit
*Acknowledgment bit
* Data
* Acknowledgment
* Stop bit (Stop condition)



*Note: Ack is returned by the receiver (for a write op it's slave and for a read op it's master; that acknowledges the sender)

Linux I2c Driver stack



LINUX I2C DRIVER STACK

Data structures specific to i2c stack <include/linux/i2c.h>

struct i2c_adapter:-
I2c controller drivers registered with mid layer through an instance of type i2c_adapter.

struct i2c_client:-
Each i2c slave device specified in the device tree is enumerated as an instance of this type.
I2c client structure will contain a reference to i2c_adapter object which represents controller device.

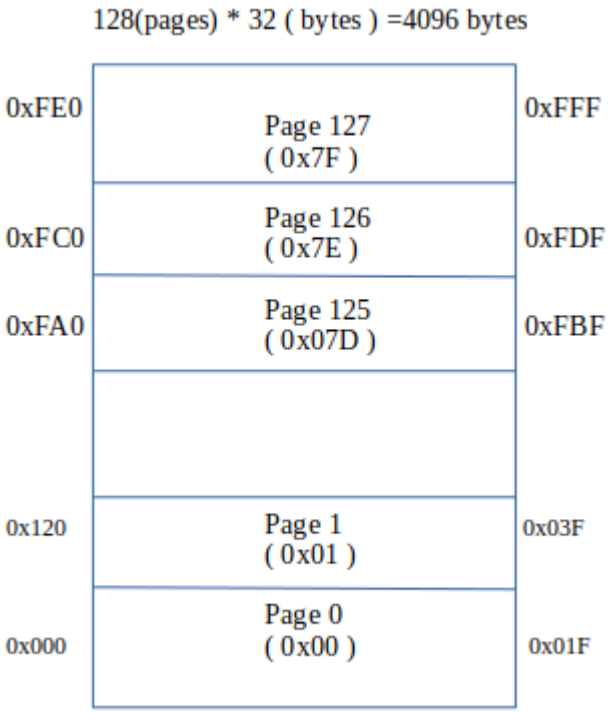
struct i2c_driver:-

Each slave driver will have to register with i2c core through an instance of this type

I2c slave driver implementation:-
Device set up:
AT24C32 (EEPROM) Description

Type	EEPROM
Family/ variant	AT24/C32
Capacity	32kilobits (4096 bytes)
Page size	128 (0x80)
Address Width	12bits (2^12 = 4096)
Op clock	100khz (3.3v) , 400khzz (5v)
Client address	0x50

AT24C32 Memory Organization



Step 1: Define a device tree Node

```
&i2c2 {
        status = " okay "
        /* Node for at 25c32 eeprom */
        at242eeprom: at24@50 {
        compatiable = "24c32"
        reg = < 0x50 >
        size = < 4096 > /* 4K bytes */
        pagesize = < 32 > ;
        address-width = < 12 > /* In bits */
        };
```

```
};
```

Step 2:-
Register driver with i2c core

```
static const struct i2c_device_id at24_ids[]={
{“24c32”, 0x50 },
{ /* END of List }
};


/*struct i2c_driver - represent an I2C device driver*/
static struct i2c_driver at24_driver = {
	.driver = {
		.name = "at24",
		.owner = THIS_MODULE,
	},
	.probe = at24_probe,
	.remove = at24_remove,
	.id_table = at24_ids,
};
module_i2c_driver(at24_driver);
```

Drivers probe routine is invoked with i2c_client instance of the specified device as argument

```
static int at24_probe(struct i2c_client *client, const struct i2c_device_id *id)
{
Step 1: Gather configuration data and set up driver private structure
Step 2: Registered the driver with choosen kernel driver layer ( as per driver model of the kernel )
}
```

i2c core provides a set of functions which can be used to access device tree properties of a specified node include/linux/property.h

```
int device_property_read_u32(struct device *dev , const char *propname,u32 *val) ;
```

* Above function return specified properties value into address provided through last argument (* val)
Header file provide variants above all useful to read, bool, u8, u16, u32, u64 (Types of property value)
device_property_read_u32(&client->dev, "size", &prv→size);

Communicating with device:-

1. i2c core Layer provides two different types of routines for data transfer.
   I) Functions to transfer a single data block.
   ii) Functions to relay a chain of messages between master and slave.

int i2c_master_send(struct i2c_client *client, const char *buf, int count);
int i2c_master_recv(struct i2c_client *client, char *buf, int count);

These routines read and write some bytes from/to a client, the client contains the i2c address, so you do not have to include it. The second parameter contains the bytes to read/write, The third the no of bytes to read/write (must be less than the length of the buffer, also should be less than msg len is u16).
returned is the actual no of readd / written

Type 2:

int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msg, int num);

This sends a series of messages each message can be a read or write, and they can be mixed in anyway. The transactions are combined no stop bit is sent between transaction.

i2c_core data structure to encapsulate data

```
struct i2c_msg {
        _u16 addr; //address of client
         _u16 flags; // r/w
        _u16 len; // length of buff
        _u8  *buf; // address  of buff
                };
```

The following sample code is to transfer data from host on to i2c eeprom client.

```
static ssize_t at24_eeprom_write(struct i2c_client *client, const char *buf,
            unsigned offset, size_t count)
{
      struct i2c_msg msg;
              char msgbuf[40];
```

```
/*
 * Client address(7-bit)
 */
	msg.addr = client->addr;
/*
 * R/W : 1/0(1-bit)
 */
	msg.flags = 0;
	msg.buf = msgbuf;
/*
 * Byte data address/offset should be 12 bits long
 * MSB should be in 1st byte & MSB should be in 2nd byte
 */
	msg.buf[0] = (offset >> 8); // higher 4bit
	msg.buf[1] = offset;	// lower 8bits
/*
 * Copy buf data to i2c message buffer
 */
	memcpy(&msg.buf[2], buf, count);
/*
 * no.of bytes to be sent through i2c msg
 */
	msg.len = count + 2;
		status = i2c_transfer(client->adapter, &msg, 1);
}
```

Reading

```
/*
 * Callback for reading data from i2c slave/client device
 */
static ssize_t at24_eeprom_read(struct i2c_client *client, char *buf,
		unsigned offset, size_t count)
{
	struct i2c_msg msg[2];
```

```
        u8 msgbuf[2];
/*
 * Byte data address/offset should be 12 bits long
 * MSB should be in 1st byte & MSB should be in 2nd byte
 */
        msgbuf[0] = offset >> 8;
        msgbuf[1] = offset;
/*
 * Client address(7-bit)
 */
        msg[0].addr = client->addr;
/*
 * R/W : 1/0(1-bit)
 */
        msg[0].flags = 0;
        msg[0].buf = msgbuf;
/*
 * no.of bytes to be sent through i2c msg
 */
        msg[0].len = 3;

/*
 * data message
 */
        msg[1].addr = client->addr;
        msg[1].flags = I2C_M_RD;
        msg[1].buf = buf;
        msg[1].len = count;

timeout = jiffies + msecs_to_jiffies(write_timeout);
        do {
              status = i2c_transfer(client->adapter, msg, 2);
           }while (time_before(read_time, timeout));
```

```
}
```

```
Struct i2c_client
{
unsigned short flags;   /* dir,
 unsigned short  addr; /* Chip address – NOTE: 7bit */
                              /* Addresses are stored in the lower 7bits */
char name[i2c_NAME_SIZE];
struct i2c_adapter *adapter; /* the adapter wesiton */
struct device dev;  /* the device structure */
int irq; /* irq issued by device */
struct list_head detected;

#if IS_ENABLED(CONFIG_I2C_SLAVE)
i2c_slave_cb_t slave_cb; /* callback for slave mode */
#endif
};
* An i2c_client identifies a single device ( if , chip ) connected to an i2c_bus
```

Device tree syntax

I) Device tree it is a method of de coupling hardware information (SOC , board configuration) from kernels architecture branch.

It allows board and device support to became data driven, to make setup decisions based on data passed into the kernel instead of on per machine hard coded selections.

Three main uses of why use device tree linux uses DT data for three major purposes.

1. Platform identification
2. Run time configuration
3. Device population

Layout of DT script:-

The device tree begins with a root node, represented by a forward slash ,/, which contains subsequent nodes representing the hardware of the system. Each node has a name and contains number of properties in the form

name = “value”

```
/dts-v1/;
/{
model = “TIAM335x Begalebone”
compatible= “ti,am33xx”;
#address-cells=<1>;
#size-cells=<1>;
cpus{
#address-cells=<1>;
#size-cells=<0>;
        cpu@0{          Vendor name, Device component
                compatible= “arm,cortex-a8”;
                device-type = “cpu”;
                reg=<0>;
                };
        };
memory@0x80000000{
                device-type=”memory”;
                reg=<0x8000000-0x200000000>;
                        };

        };
```

* Each node contains ‘compatible’ property. In above sample both ‘root’ and ‘cpu’ nodes have compatible property

* Linux kernel uses this to match the name against compatible strings specified by BSP and drivers (Platform)

* It is standard that value is composed of manufacture name and component name to reduce confusion between similar devices mode by different manufactures hence ti,33xx and arm,cortex-a8.

* It is common convention that the names of nodes include an ‘e’ followed by a address that helps distinguish them from other similar nodes.

* Reg property is used to specify address mapping information.

* Value of reg is expressed as a list of 32 bit integers called cells. Device tree compiler is provided with additional instructions which help identify values passed for each cell.

* #address-cells and $size cells properties of the parent node are used by device tree compiler to interpret base address and length information of the component address space from value passed in reg.

*Reg property is could also be assigned list of doubles in the form to represent multiple

reg=<address1 length1 [address 2 length 2] [address 3 length 3]>

address space assignments for the same hardware components.

```
/{
        #address-cells=<1>;
        #size-cells=<1>;
        gpio@101f3000{
                        compatible= “arm,pl06,”;
                        reg=<0x101f3000 ox1000
                                0x101f4000 0x0010>;
                        interrupt-controller@10143000{
                        ompatible= “arm,pl190”;
                        reg=<0x101f4000 0x0010>;
                        }
```

External (off-chip) Device nodes

Soc’s might support various external buses with discrete chip select lines through which devices can be connected with system devices attached through such interfaces use a different addressing scheme and DT nodes must describe their addresses.

```
External-bus {
#address-cells=<2>;
#size-cells=<1>;
ranges=<0 0 0x10100000 0x1000 // Chip select1 Ethernet
        <1 0 0x10160000 0x1000 // Chip select2 i2c controller
        <2 0 0x30000000 0x1000 // Chip select 3 NOR flash
        ethernet@0,0{
        compatible=”smc,smc91c11”;
```

```
        reg<0 0  0x1000 >;
        };
chipselect, offset
i2c@1,0{
        compatible=”acme,a1234-i2cbus”;
        reg<0 0  0x1000 >;
        };
flash@2,0{
        compatible=”samsang,k8fl3” “cfi-fish”;
        reg<2 0  0x1000 >;
        };
};
```

The external bus uses 2cells for the address value; one of the chip select number and other for the offset from the base of the chip select.

In this example each reg ENTRY contains 3cells

< chip select number, offset, and length >.

* Linux drivers cannot derive the address through chip select numbers they would need physical address.

* In order to translate addresses upward (I.e, into CPU physical addresses) for these external bus nodes we must define “ranges” property.

* “ranges” servers as an extension to ‘reg’ through which compiler resloves address details of chip select devices. It is a table of addresses for each chip select line.

Physical Properties:-

* The structure of the device tree described so far presents a single hierarchy of components (in order of their appearance from CPU and address space assigned).

* Device might also be connected to an interrupt controller, clock source and voltage regulator,

To express these connections we have ‘phandles’

```
/dts-v1/;
{
intc: interrupt-controller@482000000{
compatible=”ti,am33xx-intc”;
interrupt-controller;
```

#interrupt-cells=<1>;
reg=<0x482000000 0x10000>;
};
serial@44e09000{
compatiable=”ti,omap3-uart”;
ti,hwmods=”uart1”;
clock-frequency=<480000000>;
reg=<0x44e09000 0x2000>;
interrupt-parest=<&intc> // phandle
interrupts=<72>;
};
};
phandles can also be used to extend or overwrite properties of a node previously defined.
&mmc1{
status=”okay”

};
Slave devices:-
Representing static slaves
-Dt nodes most also created for represents slave devices on static buses like i2c, spi.

I2c@1,0{
compatible=”acme,1234_i2c_bus”;
#address-cells=<1>;
#size-cells=<0>;
reg=<1 0 0x1000>;
rtc@58{
compatible=”maximum,ds1338”;
reg=<58>;
};
};
spi@foo{

#address-cells=<1>;

#size-cells=<0>;

ethernet-switch@0{

compatible=”micrel,ks8995m”;

spi-max-frequency=<10000000>;

reg=<0>;

};

Interrupt lines:-

Unlike device addressing which is naturally expressed in the device tree, interrupt signals are expressed as links but nodes dependent of the tree.

Four properties are used to describe interrupt connections

* Interrupt controller node

1) Interrupt-controller:- An empty property declaring a node as a device that receives interrupt signals

2)#interrupt-cells – This is a property of the interrupt controller node. It states how many cells are in a interrupt specifier for this interrupt controller

(Similar to #address-cells and #size-cells)

* Using node

3) interrupt-parent: - Property of a device node containing a phandle to the interrupt controller that it is attached to **Nodes that do not have an interrupt-parent property can also inherit the property from their parent node.**

4) Interrupts:- Property of a device node containing a list of interrupt specifiers one for each interrupt o/p s/g on the device

Ex:-

intc: interrupt-controller@10140000{

compatible=”arm,pl190”,

reg=<0x10140000 0x1000>;

interrupt-controller;

#interrupt-cells=<2>;

};

spi@10115000{

compatible=”arm,pl022”;

reg=<0x10115000 0x1000>;

interrupts=<4 0>;

interrupt-parent=<&intc>

};

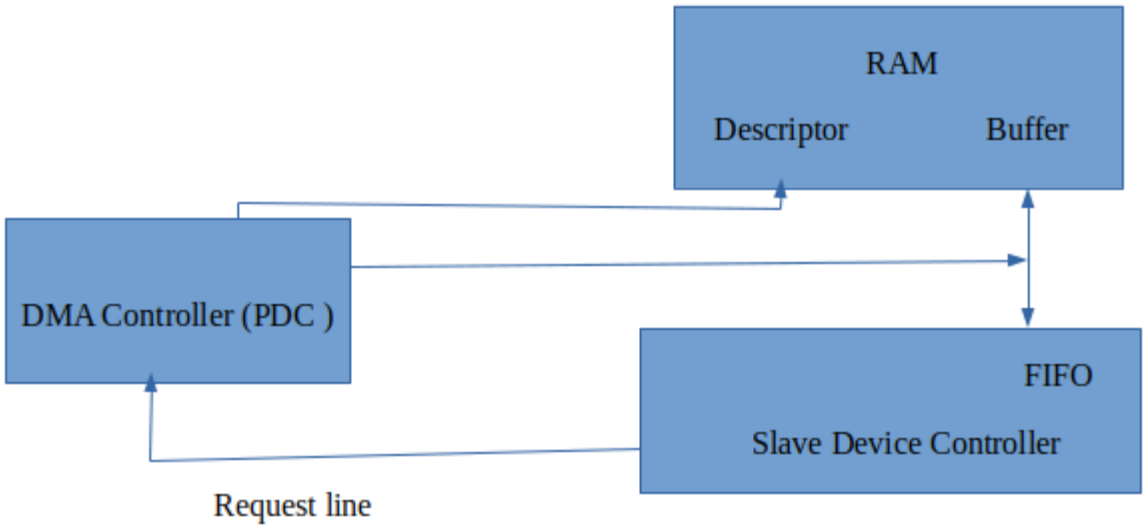
#interrupt-cells is 2, so each interrupts specifier has 2 cells.

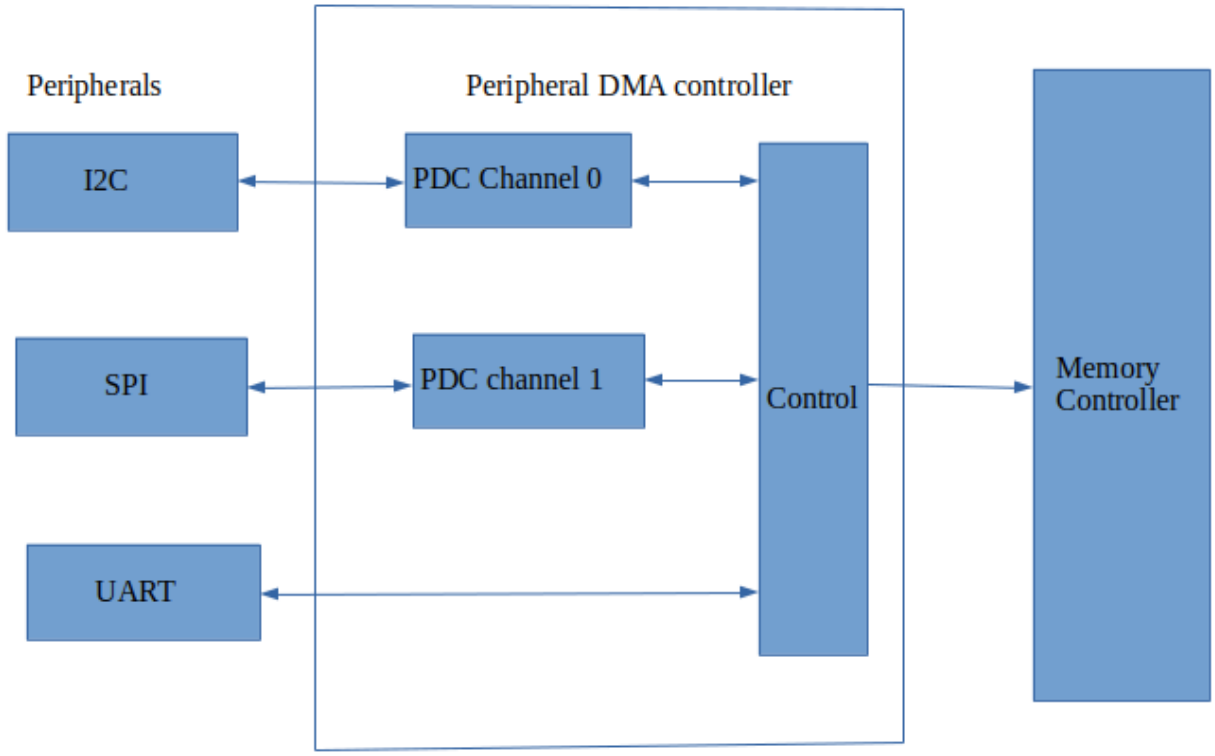
-First cell to encode the interrupt line number and the second cell to encode flags such as active high vs active low or edge vs level sensitive.

Refer binding documentation to learn how the specifier is encoded.

Peripheral DMA :-

* Peripheral DMA engines are used by I2c, SPI, UART bus controllers for data transfers these DMA controllers provide a set of channels through which data transfer operations can be achieved.





*Each channel is a set of registers through which slaves can configure transfer details

*Channels can move data uni direction or bi-directional

* A peripheral device can be assigned multiple channels

* Configuration would involves following details

I) Address of location in memory to be accessed

ii) Direction of data transfer

iii) Length of data transfer.

iv) Mode of data transfer.

* Each channel could be dedicated to a specific mode of transfer like

1. Burst transfer – A single transfer
2. Cyclic transfer - Alternative trans cyclic
3. S.G.transfer – Scatter for array of buffers
4. Interleaved transfer- When CPU idle, then transfer.

On Some SOC paltforms DMA router might be used to multiplex limited no of channels across many slaves such platforms DMA router is considered as a ip block and it is managed by a same driver which handlers DMA controller.

Device tree binding for DMA chips

DMA controller :-

```
dma:dma@480000000{
compatible=”ti,omap-sdma”;
reg=<0x48000000 0x1000>;
interrupts=<0 12 0x4
            0 13 0x4
            0 14 0x4
            0 15 0x4 >;
#dma-cells=<1>;
dm-channels=<32>;
dma_requess=<127>;
};
```

required property;

#dma-cells must be at least 1 used to provide : DMA controller

Optional properties:-

dma-channels: No of DMA

dma-requests: No of DMA request s/gs supported by the controller.

DMA router:

DMA routers are transport IP blocks used to route DMA request lines from devices to the DMA controller.

Some SOCs (like TIDRA7X) have more peripherals integrated with DMA requests than what the DMA controller can handle property.

Example: sdma-xbar: dma-router@4a002b78{
compatible=”ti,dra7-dma-crossbar”;
reg=<0x4a002b78 0xfc>;
#dma-cells=<1>;
dma-requests<205>;
ti,dma-safe-map=<0>; // Vendor properties
dma-masters=<&sdma>; // Standard
};

DMA client:

Client drivers should specify the DMA property using a phandle to the controller followed by DMA controller specific data.

1. A device with one DMA read channel, one DMA write channel.

I2c1:i2c@1
{
dma=<&dma 2 /* Read channel */
        &dma 3>; /* Write channel */
dma-names =”rx”,”tx”;
};

2. A single read-write channel with three alternative DMA controllers:

dmas=<&dma1 5
        &dma2 7
        &dma3 2>
dma-names=”rx-tx”, “rx-tx”, “rx-tx”;

3. A device with three channels one of which has two alternatives

dmas=< &dma1 2 /* Read channel */
        < &dma1 3 /* Read channel */
        < &dma2 0 /* Read channel */

< &dma3 0 > ; /* Alternative error read */

dma-names=”rx”,”tx”,”error”,”error”;

Example gpi-controller node:-

Every GPIO controller node must declare an empty gpio-controller property and #gpio-cells which declares the size of the GPIO-specifier.

Example gpio-controller nodes:-

gpio0:gpio@44e07000{

compatible=”ti,omap4-gpio”;

ti,hwmods=”gpio1”;

gpio-controller;

#gpio-cells=<2>;

interrupt-controller;

#interrupt-cells=<2>;

reg=<0x44e07000 0x1000>;

interrupts=<96>; // Interrupt line.

};

gpio-specifier length is controller dependent, it may encode bank,pin position inside the bank, state of the pin.

Exact meaning of each specifier cell is controller specific and must be documented in the device. Tree binding for the device.

Devices using GPIO pins for various operations must declare GPIO properties as per the following pattern.

1) GPIO property: Node that makes use of GPIO’s should specify them using one or more properties.

*GPIO properties should named “[<name>-]gpios” with <name> being the purpose of this GPIO for the device. While a non – existent <name> is considered valid for compatibility reasons (resolving to the “gpios” property) it’s not allowed for new bindings.

Also, GPIO properties named “[<name>-] gpio are valid and old bindings use it, but are only supported for compatibility reasons and should not

gpio2:gpio2{

gpio-controller

#gpio-cells=<1>;

enable-gpios=<&gpio2 2>; // 2 is the pin number

data-gpios= <&gpio1 12 0>;

<&gpio1 13 0>;

<&gpio1 14 0>;

<&gpio1 15 0>;

Pin control declarations

* Pin control subsystem of linux responsible for managing pin mux controller and setting up each pin in a specific mode.

* This subsystem depends on device tree declarations to read configurations for individual pins

* It may happen that different pin ranges in a Soc in managed by different gpio drivers. This makes it logical to let gpio drivers announce their pin ranges to the pin control subsystem.

&am33xx-pinmux{

gpio0-pins:gpio0-pins{

pinctrl-single,pins=<

0x144(PIN_OUTPUT_PULLUP/MUX_MODE7) /* RMII_REF_CLK as GPIO out */

0x158(PIN_OUTPUT_PULLDOWN/MUX_MODE7) /* Pin conf_spio_dl as GPIO out */

0x15C(PIN_OUTPUT_PULLDOWN/MUX_MODE7) /* Pin conf_spio_iso as GPIO out */

0x160(PINOUT/MUXMODE7)>;

};

};

&am33xx-pinmux{

spio-pins pinmux_spio_pins{

pinctrl_single,pins=<

0x150((PIN_OUTPUT_PULLUP/MUX_MODE0) /* *Spio_sclk INPUT_PULLUP*MODE0/ox30*/

0x154((PIN_OUTPUT_PULLUP/MUX_MODE0) /* *Spio_do INPUT_PULLUP*MODE0/ox30*/

0x158((PIN_OUTPUT_PULLUP/MUX_MODE0) /* *Spio_dl INPUT_PULLUP*MODE0/ox00*/

0x15C((PIN_OUTPUT_PULLUP/MUX_MODE0) /* *Spio_CS0 INPUT_PULLUP*MODE0/ox10*/>;

};

};

};

&spio{

pinctrl-names=”default”;

pincntrl-0=<&spio-pins>;

status=”okay”;

};

GPIO interrupts:-

Step 1: Device tree node

```
        extbutton{
        compatible=”gpio-button-intr”;
        gpios=<&gpio1 13 GPIO_ACTIVE_LOW>;
        };
};
```

Any device direct memory mapped is called as platform devices

Step 2: Register driver with platform core

```
static struct of_device_id gpio_button_ids[]={
{ compatible=”gpio-button-intr”};
{},
};
MODULE_DEVICE_TABLE(of,gpio_button_ids);
static struct platform_driver gpio_button_driver={
        .probe=gpio_button_probe,
        .remove=gpio_button_remove,
        .driver={
                .name =”button-intr”;
                .owner=THIS_MODULE,
                .of_match_table=gpio_button_ids,
                },
};

static int gpio_button_probe(struct paltform_device *pdev)
{

device tree// gpio = of_get_named_gpio(pdev→dev.of_node,”gpios”,0);
ret = gpio_request(gpio,NULL);
gpio_set_debounce(gpio,200); // 200 ms wait and try
Kernel IRQ// g_irq = gpio_to_irq(gpio);
request_threaded_irq(g_irq,button_isr,NULL,IRQ_TRIGGER_RI_SING,”button-isr”,NULL);
```

A33356

GPIO:

There are identical register set for each GPIO module registers are 32 bit wide

Each bit represents corresponding pin in the module

base address of GPIO controller

Module	Start addr	End addr	size
GPIO	0x44e0-7000	0x44e0-7fff	4KB
GPIO1	0x44e0-7000	0x4804-7fff	4KB
GPIO2	0x44e0-7000	0x441a-7fff	4KB
GPIO3	0x44e0-7000	0x448A-Efff	4KB

I/o configuration and operation registers:

GPIO_OE

GPIO_DATAIN

GPIO_DATAOUT

GPIO_CLEARDATAOUT

GPIO_SETDATAOUT

Register description:

GPIO_OE: By default GPIO pins are set as i/p they are set to o/p mode through GPIO_OE register

GPIO_DATAIN: In i/p mode gpio i/p is read from this register

GPIO_DATAOUT: In o/p mode pin state is changed through this register

0x134 R/W 0xFFFF

pin o/p state could be also changed set & clear o/ps

GPIO_CLEARDATAOUT – Reset (10) pin o/p

GPIO_SETDATAOUT – Sets (10) pin o/p

Step 1: Declare device tree node

```
/extled {
compatible =”gpio-extled”
gpios < &gpio0 27 GPIO_ACTIVE_LOW>;
};
};
```

Step 2: Register driver with platform core

```
Static const struct of_device_id led_of_mtable[]={
{ .compatible=”gpio-extled”,},
        { }
};
MODULE_DEVICE_TABLE(of,led_of_mtable);
static struct platform_driver led_driver ={
        .driver={
        .name=”off_board_led”,
        .owner=THIS_MODULE,
        .off_match_table=ledof_mtable,
        };
        .probe=led_probe;
        .remove=led_remove;
        };
module_platform_driver(led_driver);
```

Step 3: Probe Function

```
led_probe(struct platform_device *pdev)
{
```

Step 1: Gather GPIO config data from device tree node

```
struct device node *pnode;
int ret;
/* Instantiate driver private structure */
gpbank = ( struct gpiopin-bank *) kmalloc( sizeof(struct gpiopin_bank), GFP_KERNEL);
/* Gather GPIO pin no from device tree node */
gpbank→ pinno = of_get_named_gpio(pdev→dev.ofnode,”GPIOS”,0);
 /* Gather reference to GPIO controller dev tree node */
pnode = of_parse_phandle(pdev→dev_of_node,”gpios”,0);
/* Access reg property from controller node and gather start address */
      ret = of_property_read_u32(pnode, "reg", &gpbank→gpio_base);
/* Map controller address space to kernel virtual address space */
      gbank_base = ioremap(gpbank->gpio_base, 4095);
```

Step 4: Set pin in output mode

```
          pin_config();
```

Step 1: Provide interface to user mode

```
	kobj = kobject_create_and_add("ext_led", NULL);

		ret = sysfs_create_file(kobj, &led_attr.attr);
}

static struct kobj_attribute led_attr =__ATTR(extled, 0660, extled_read, extled_write);

static ssize_t extled_write(struct kobject *kobj, struct kobj_attribute *attr, const char *buf, size_t count)
{
 ret = sscanf(buf,"%d",&d);
	if(d==0) {
		pr_info("extled_write: led is off\n");
		data = data | (1U << gpbank->pinno);
		writel_relaxed(data, gbank_base + OMAP_GPIO_CLEARDATAOUT);
	}
	else {
		pr_info("extled_write: led is on\n");
		data = data | (1U << gpbank->pinno);
		writel_relaxed(data, gbank_base + OMAP_GPIO_SETDATAOUT);
	   OMAP_GPIO(CLEARDATAOUT);
	}

}
```

* If is not peferred to directly manipulate GPIO register from client drivers

kernel provides a set of functions called GPIO lib through which client drivers must initiate all GPIO operations

* This mode centralized all GPIO physical operations into a common driver which manages controller

GPIO libs:-

This layer offers to categories of GPIO helper routine.

1. Simple Integer interface
2. Descriptor term interface

The following are operations available through both of the interfaces.

gpio_request()	request gpio
gpio_free()	free gpio
gpio_to_desc()	to convert a gpio number to it's descriptor
gpio_direction_output()	set gpio as o/p
gpio_direction_input()	set gpio as i/p
gpio_get_value()	get value from gpio i/p
gpio_set_value()	set value from gpio o/p
gpio_set_debounce()	set_debounce time for gpio i/p
gpio_to_irq()	to request an irq for gpio

* All of the above function interfaces are available in both variant (integer and descriptor) with an exception descriptor function name start with gpiod instead of gpio and first argument is the reference to gpio descriptor structure representing corresponding pin.

SPI Protocol (Lower latency compared to i2c)

* Synchronous serial protocol developed by motorola

* Simple four-wire bus interface

* LCD, SDCard, DAC, ADC, Camera lens mount flash, ROM

SPI Bus lines

Consists of four lines

MOSI (Master out slave in) - unidirectional data line , simplex mode

MISO (Master in slave out) - unidirectional data line , simplex mode

SCLK (Serial clock) - unidirectional clock line , simplex mode

CS (Chip select) - unidirectional data line , simplex mode

* Each slave is interfaced with a master through a dedicated chip select line.

* Master selects a specific slave device by pulling down it's chip select line. And chip select line is pull down. Slave is deselected.

* Master and slave communicate through a set of slaves specific commands.

* Some SPI controllers support multi slave selections. In such cases master can select multiple slaves and initiate communication

* Multi slave select configuration are used only in cases where master doesn't require data to be returned from the slave.

* Some SPI controllers support daisy chain configuration. In this all slaves share a common chip select line a data is moved by the master. Transverse across all of the slave. The last slave in the series can sends back a acknowledgment to master (LCD , LED)

* SPI controllers must be configured with information of clock mode. On must SPI controllers support four different clock modes.

SPI clock modes

* Specifies when to sample date

* Consists of two binary variables.

CPOL (Clock polarity) 0 initially clock is low

1 initially clock is high

CPHA (Clock phase) 0 Sample data at leading edge

1 Sample data at trialing edge

* There are four clock modes

Clock Mode				
(Default Mode)	0	00	Clock Low	Leading edge
	1	01	Clock Low	trailing edge
	2	10	Clock High	Leading edge
	3	11	Clock High	trailing edge

Am335x spi configuration
Two SPI bus controller
Two chip selects per controller
Interface configuration
Up to four external slave devices or one single external master (BBB as slave)

Spio: p9-17 (CS0), 18 (D1), 21 (D0), 22 (SCLK)
spi1: p9-28 (cs0) ,29 (D0) , 30 (D1), 31 (SCLK)

Slave device (W25Q32)

Type	Flash
Family/Variant	w25/q32
Capacity	32 Megabits (4M bytes)
Page size	256 bytes (16384)
Sector size	4096 kilo bytes(1024)
Block size	64 kilo bytes(64)
Address width	24bits
Clock modes	0,3 (Clock modes supports)
op Clock	104MHz, single

External and internal address Nand used for stroage execution (Nor for image)

External Address	Device address (Cs) or Bus 0x0
Internal address	On chip memory address 0x000000 – 0x3FFFFFFF

Driver Steps

Step 1: Device tree node

```
&am35xx_pin_mux{
spio-pins: pinmuxspio-pins{
pinctrl-singlempins=<
0x150(PIN_INPUT_PULLUP/MUX_MODE0)/* spio_sclk INPUT_PULLUPMODE0/ox30*/
0x154((PIN_OUTPUT_PULLUP/MUX_MODE0) /* Spio_do INPUT_PULLUPMODE0/ox30*/
0x158((PIN_OUTPUT_PULLUP/MUX_MODE0) /* Spio_dl INPUT_PULLUPMODE0/ox00*/
0x15C((PIN_OUTPUT_PULLUP/MUX_MODE0) /* Spio_CS0 INPUT_PULLUPMODE0/ox10*/>;
};
};

&spio{
pinctrl-names="default";
pincntrl-0=<&spio-pins>;
status="okay";
/* DT mode for w25932 spi flash chip */
w25q32: w25q32@0 {
        compatible="winbond,w25q32";
spi.max-frequency=<5000>;
reg=<0x0> // Chip select
size=<494304>; /* 4M bytes */
pagesize=<256>;
address-width=<24>; /* In bits */
};
};
```

Step 2: Register driver with SPI core

```
static const struct of_device_id w25_of_match[] = {
	{ .compatible = "winbond,w25q32", },
	{ }
};

static struct spi_driver w25_driver = {
	.driver = {
		.name		= "w25q32",
		.of_match_table = w25_of_match,
	},
	.probe		= w25_probe,
	.remove		= w25_remove,
};

module_spi_driver(w25_driver);
```

(It is not platform device so no arg of =open firmware)

Step 3:

```
static int w25_probe(struct spi_device *spi)
{
```

Step 1: Instantiate private structure

Step 2: Access device tree configuration data device_property_read_u32();

Step 3: Present the device to user space

prv->kobj = kobject_create_and_add("w25q32_flash", NULL);

Communication with device:-

Device supports following operation

→ Write enable 0x06

→ Page program

→ Read data

→ Sector erase

→ Read status

→ Manufacture id

Communication with slaves

* Spi core provide two categories of functions for communication with slave

1) Functions to transfer / Receive data

2) Functions interfaces to queue multiple transfers and execute them.

Type 1 functions :

<linux/spi/spi.h>

int spi_write(struct spi_device *spi,const void *buf, size_t len);

int spi_read(struct spi_device *spi,void *buf, size_t len);

ssize_t spi_w8r8(struct spi_device *spi, u8 cmd)

returns unsigned eight bit data, read from device else a negative error number.

Type 2 functions :

void spi_message_init(struct spi_message *m);

void spi_message_add_tail(struct spi_transfer *t, struct spi_message *m);

int spi_sync(struct spi_device *spi, struct spi_message *m);

synchronous/ blocking spi transfers

int spi_async(struct spi_device *spi, struct spi_message *m);

asynchronous/ blocking spi transfers

int spi_write_then_read(struct spi_device *spi, const void *txbuf,unsigned n_tx, void *rxbuf, unsigned n_rx)

For each transfer details must be specified in instance of type spi-transfer

Transfer object must be queued into a driver specific list of type struct spi_message

struct spi_transfer

{

const void *tx_buf; // Transfer

void *rx_buf; // receive buffer

unsigned len;

u32 speed_hz;

dma_addr_t tx_dma;

dma_addr_t rx_dma;

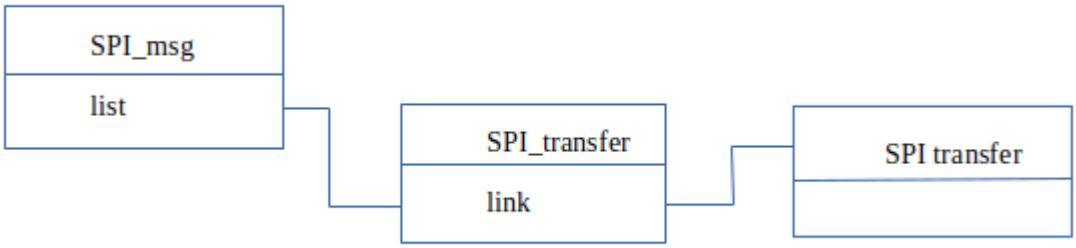
struct list_head transfer_list;

};

struct spi_message {

struct list_head transfers;

struct spi_device *spi;
unsigned is_dma_mapped ;
};



Driver operations:
Erase
Step 1: Users can initiate erase operations through the sysfs file presented
/sys/ w25q32_flash/erase
offset
w25q32
Before writing any data, erase the sector first containing
echo 0x01:0x00 > erase

static ssize_t
w25_flash_erase(struct kobject *kobj, struct kobj_attribute *attr,
const char *buf, size_t count)
{
Step 1: Access user inputs & convert them into integer format
Step 2: Validate specified block and sector
Step 3: Enable write lunch

char cp[32], temp[32];
memset(temp, '\0', sizeof(temp));
temp[0] = W25_WREN; // Write enable command from data sheet
status = spi_write(w25->spi, temp, 1); // 1 byte write
if (status < 0)
pr_info("WREN --> %d\n", (int) status);

Step 4: Initiate erase

```
struct spi_transfer t;
        struct spi_message m;
        unsigned long tmp;
        unsigned int block, sector;
   tmp = (block * 0x10000) + (sector * 0x1000);
temp[0] = (u8)W25_SEC_ERASE;
        temp[1] = tmp >> 16;
        temp[2] = tmp >> 8;
        temp[3] = tmp >> 0;

        t.tx_buf = temp;
        t.len = 4;

        spi_message_init(&m);
        spi_message_add_tail(&t, &m);

        status = spi_sync(w25->spi, &m);
};
```

Read operation:-

Step 1: To initiate read the following operations must be perform from user

Step 2: Set the offset page to be red into offset file.

echo <block>:<sector>:<page> > offset

echo 0x10:0x1:0x1 > offset

Valid ranges for

Block 0x00-0x3F

Sec 0x0-0xF

Page 0x0-0xF

To read data specified location is

cat w25q32

Driver operations

→ Driver gathers the offset and stores it in the private structure

→ Drivers read function will carries out required transfer operations to fetch the data of the specified object.

```
static ssize_t
w25_sys_read(struct kobject *kobj, struct kobj_attribute *attr, char *buf)
{
        struct w25_priv *w25 = prv;
        size_t count = IO_LIMIT;

        return w25_flash_read(w25, buf, (int)w25->offset, count);
}
static ssize_t
w25_flash_read(struct w25_priv *w25, char *buf, unsigned offset, size_t count)
{
Declare spi_transfer & message instances
        struct spi_transfer t[2];
        struct spi_message m;
        ssize_t status;
        u8 cp[5];

Step 2: Initialize transfer instances
 memset(buf, '\0', IO_LIMIT); // Page size is IO_LIMIT
        memset(t, 0, sizeof(t));

        cp[0] = (u8)W25_READ;
        cp[1] = offset >> 16;
        cp[2] = offset >> 8;
        cp[3] = offset >> 0;
        spi_message_init(&m);
 t[0].tx_buf = cp;
```

```
	t[0].len = w25->addr_width/8 + 1;
	spi_message_add_tail(&t[0], &m);

	t[1].rx_buf = buf;
	t[1].len = count;
	spi_message_add_tail(&t[1], &m);

	status = spi_sync(w25->spi, &m);
}
```

Write operation

```
static ssize_t
w25_sys_write(struct kobject *kobj, struct kobj_attribute *attr,
		const char *buf, size_t count)
{
	struct w25_priv *w25 = prv;

	return w25_flash_write(w25, buf, w25->offset, count);
}
```

Steps : Set command and write data

```
static ssize_t
w25_flash_write(struct w25_priv *w25, const char *buf,
		 loff_t off, size_t count)
{
Declare spi_transfer & message instances
 struct spi_transfer t;
	struct spi_message m;
 Initialize transfer instances
 cp[0] = (u8)W25_WREN;
	status = spi_write(w25->spi, cp, 1);
```

```
  tmp[0] = (u8)W25_WRITE;
        tmp[1] = off >> 16;
        tmp[2] = off >> 8;
        tmp[3] = off >> 0;
        memcpy(tmp + 4, buf, count);

        t.tx_buf = tmp;
        t.len = count + 4;

        spi_message_init(&m);
        spi_message_add_tail(&t, &m);

        status = spi_sync(w25->spi, &m);
}
```

THANK YOU