Training Agenda

- Linux Booting Process
- X86 Booting
- ARM Booting
- U-Boot
- Kernel
- Linux Device Driver

C4. Advanced Kernel Topics and Debugging (1/2)

Duration: 40 hours

- 11. Linux Boot Process and Partitions
- - Understanding the Linux Boot Process
- - Boot stages: BIOS/UEFI, Bootloader, Kernel, Init.
- - Role of U-Boot in the boot process.
- - Configuring Bootloader and Managing Partitions
- - Configuring U-Boot environment variables.
- - Partitioning SD card and configuring `fstab`.
- Sub-Task: Analyze and modify the boot sequence of Telechips, configure partitions.

Tools

Hardware: Telechips
Software: U-Boot, fdisk, GParted

Maneeesh>> Need 3-4 Days to prepare slides for this topics

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (1/3)

Duration: 40 hours

- 8. Character, Block, and Network Driver Writing
- - Writing Character, Block, and Network Drivers
- - Differences between character, block, and network drivers.
- - Key functions and structures in writing each type of driver.
- - File Operations in Drivers
- - File operations (`open`, `read`, `write`, `ioctl`).
- - Understanding the VFS (Virtual File System) layer.
- - Understanding the Kernel Driver Model
- - Driver registration and unregistration.
- - Major and minor numbers.
- Project 5: Character Driver Implementation
- - Implement a character driver to control GPIOs on Telechips.

Tools

Hardware: Telechips
Software: Linux Kernel Source, GPIO Tools

Maneesh>> Need 3-4 Days to prepare slides for this topics

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (2/3)

Duration: 40 hours

- 9. Platform Device Driver Creation and APIs
- - Creating Platform Device Drivers
- - Platform devices and drivers.
- - Registering platform devices.
- - Understanding Platform APIs and Their Usage
- - Overview of platform-specific APIs.
- - Managing power and clock resources for devices.
- Sub-Task: Develop a platform driver for an onboard peripheral like SPI or I2C.

Tools

Hardware: Telechips
Software: Linux Kernel Source, SPI/I2C Tools

Maneesh>> Need 3-4 Days to prepare slides for this topics

Note: Telechips as hardware, offered by Faurecia

C2. Kernel Module Development and Cross Compilation(3/3)

Duration: 40 hours

- 7. Device Tree and HW-SW Interface
- - Understanding the Hardware-Software Interface
- - Introduction to device trees and their purpose.
- - Device tree source (DTS) and device tree blob (DTB).
- - Device Tree Concepts, Syntax, and Structure
- - Syntax of device tree source files.
- - Nodes, properties, and overlays.
- - Board Configuration Using the Device Tree
- - Configuring board-specific peripherals.
- - Device tree binding and its role in driver loading.
- Project 4: Device Tree Configuration
- - Modify the device tree to enable and configure specific peripherals on Telechips.

Tools

Hardware: Telechips
Software: Device Tree Compiler, U-Boot

Maneeesh>> Need 2-3 Days to prepare slides for this topics

Note: Telechips as hardware, offered by Faurecia

C3. Driver Development and Peripheral Interface (3/3)

Duration: 40 hours

- 10. Peripheral Interface (GPIO, SPI, I2C, etc.)
- \- Detailed Study of Peripheral Subsystems
- \- Overview of common peripherals: GPIO, SPI, I2C, UART, PWM.
- \- Implementing and Configuring Peripherals
- \- Configuring GPIO pins for input and output.
- \- Writing drivers for SPI, I2C communication.
- \- Hardware-Software Interfacing
- \- Interfacing peripherals with device drivers.
- Project 6: Peripheral Driver Development
- \- Write and test drivers for specific peripherals such as SPI and I2C on Telechips.

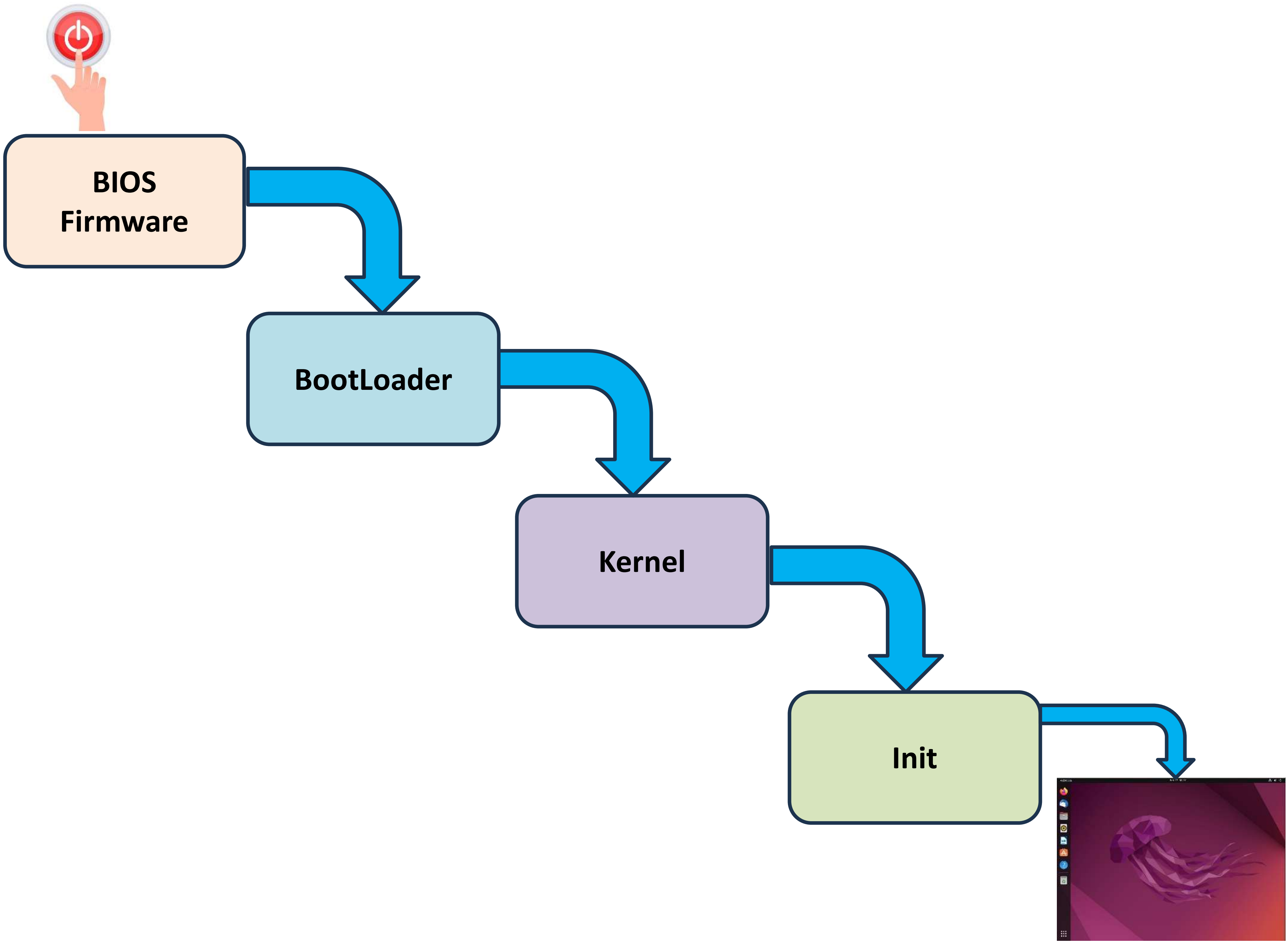
Tools

Hardware: Telechips
Software: Peripheral Tools, Linux Kernel Source

Maneesh>> Need 4-5 Days to prepare slides for this topics

Note: Telechips as hardware, offered by Faurecia

Linux Boot Flow – x86


BIOS
Firmware
BootLoader
Kernel
Init

What is BIOS

x86 platforms shipped before 2005-2006 include a firmware called BIOS

- BIOS = Basic Input Output System
- Part of the hardware platform, closed-source, rarely modifiable
- Implements the booting process
- Provides runtime services that can be invoked - not commonly used
- Stored in some flash memory, outside of regular user-accessible storage devices

To be bootable, the first sector of a storage device is "special"

- MBR = Master Boot Record
- Contains the partition table
- Contains up to 446 bytes of bootloader code, loaded into RAM and executed
- The BIOS is responsible for the RAM initialization

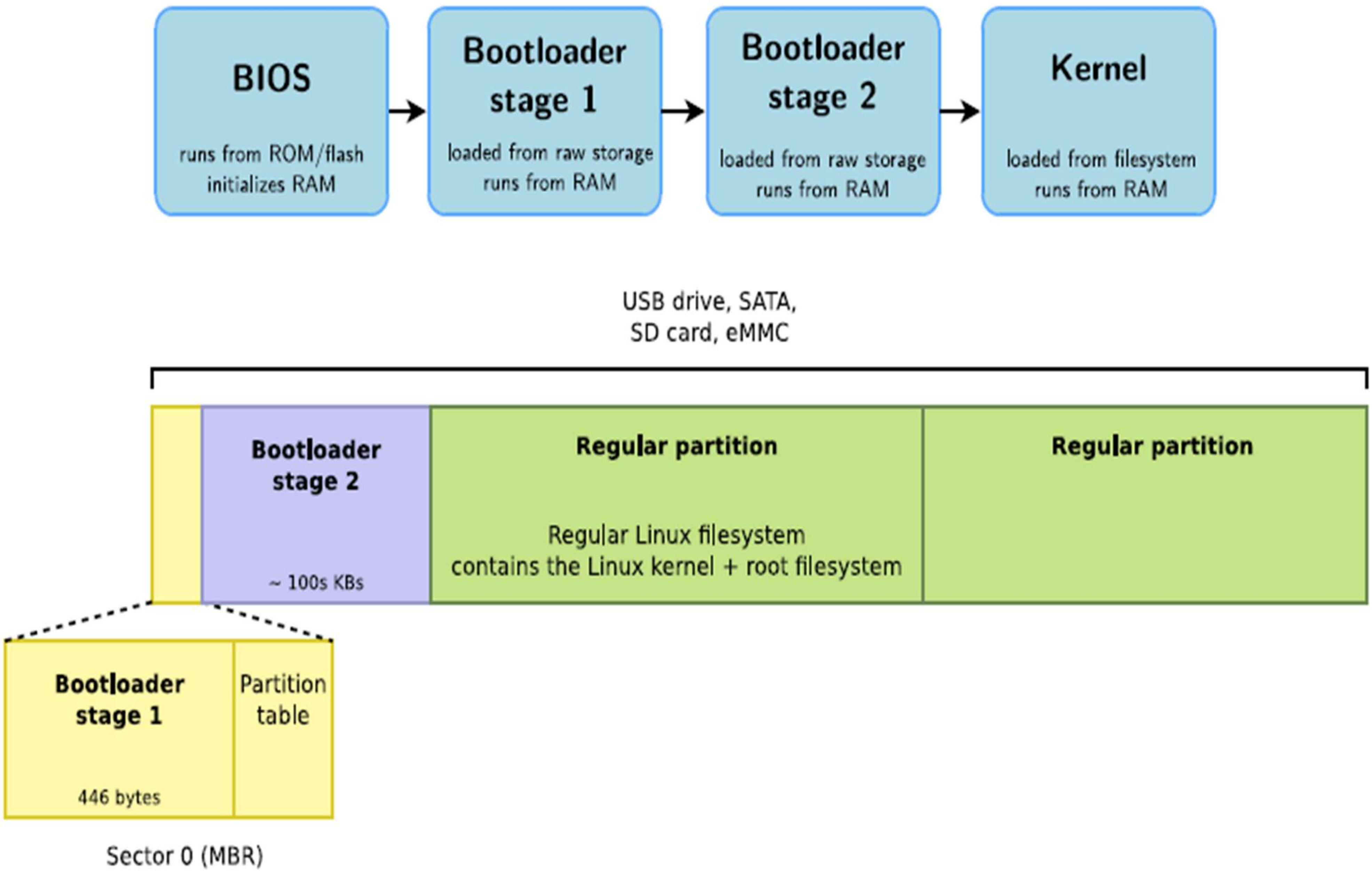
https://en.wikipedia.org/wiki/BIOS

What is Bootloader

- **The bootloader is a piece of code responsible for**
  - Basic hardware initialization
  - Loading of an application binary, usually an operating system kernel, from flash
  - storage, from the network, or from another type of non-volatile storage.
  - Possibly decompression of the application binary
  - Execution of the application
- **Besides these basic functions, most bootloaders provide a shell or menu**
  - Menu to select the operating system to load
  - Shell with commands to load data from storage or network, inspect memory, perform
  - hardware testing/diagnostics
- **The first piece of code running by the processor that can be modified by us i.e. developers.**

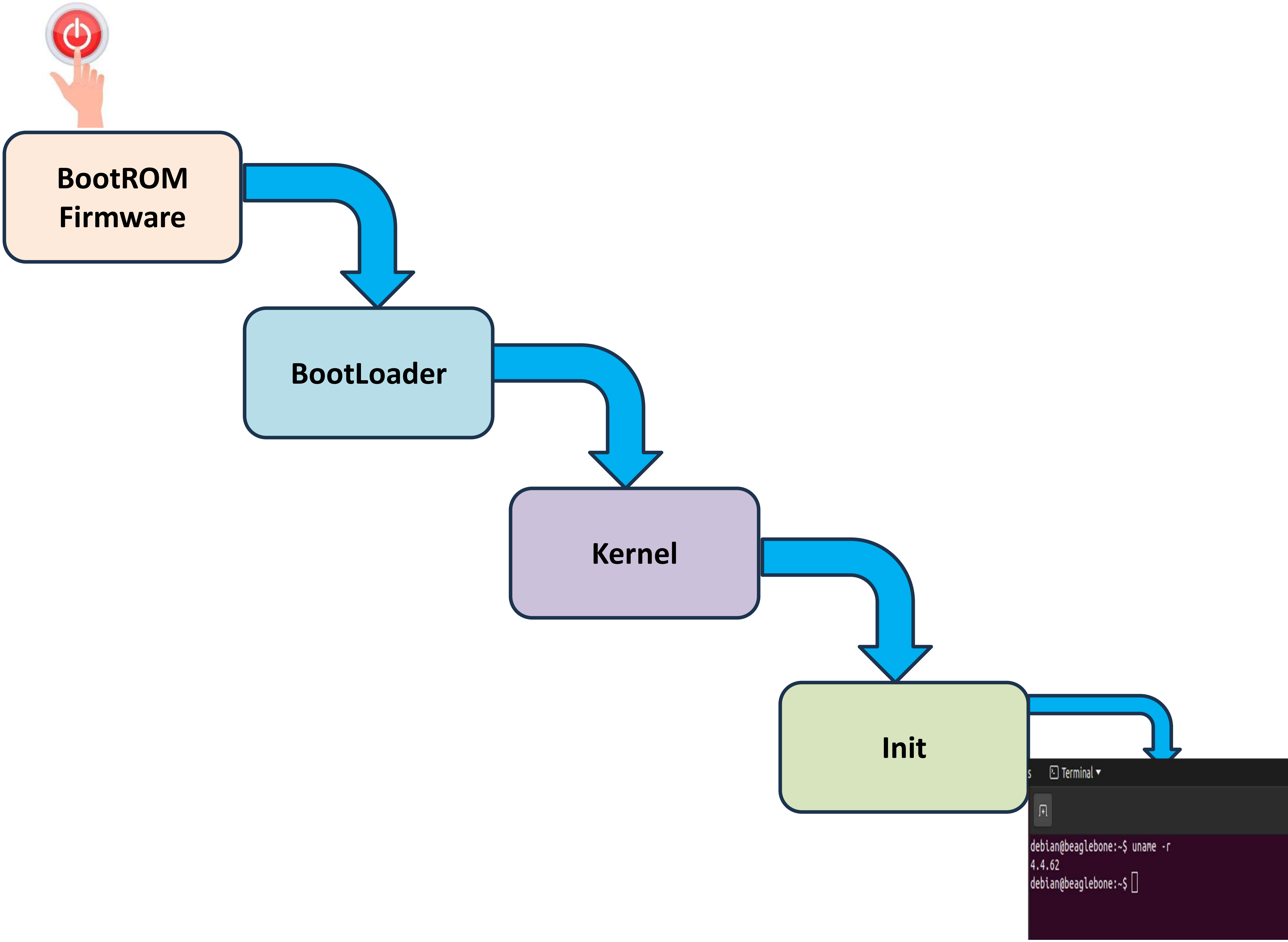
What is Bootloader

- **Due to the limitation in size of the bootloader, bootloaders are split into two stages**
  - Stage 1, which fits within the 446 bytes constraint
  - Stage 2, which is loaded by stage 1, and can therefore be bigger
- **Stage 2 is typically stored outside of any filesystem, at a fixed offset → simpler to load by stage 1**
- **Stage 2 generally has filesystem support, so it can load the kernel image from a filesystem**


BIOS
runs from ROM/flash
initializes RAM
Bootloader stage 1
loaded from raw storage
runs from RAM
Bootloader stage 2
loaded from raw storage
runs from RAM
Kernel
loaded from filesystem
runs from RAM
USB drive, SATA,
SD card, eMMC
Bootloader stage 2
~ 100s KBs
Regular partition
Regular Linux filesystem
contains the Linux kernel + root filesystem
Regular partition
Bootloader stage 1
446 bytes
Partition table
Sector 0 (MBR)

Practical : 1

Linux Boot Flow – ARM


BootROM Firmware
BootLoader
Kernel
Init
Terminal
debian@beaglebone:~$ uname -r
4.4.62
debian@beaglebone:~$

What is ROM Code

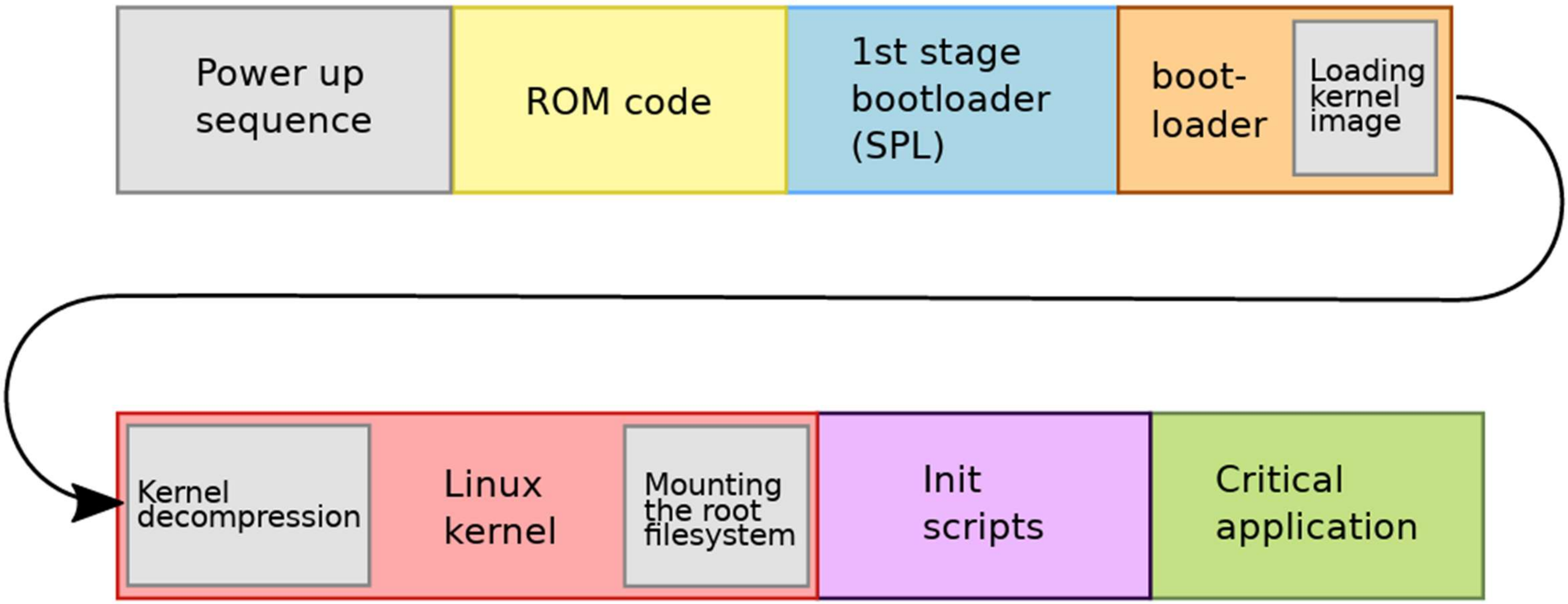
- Most embedded processors include a ROM code that implements the initial step of the boot process

- The ROM code is written by the processor vendor and directly built into the processor
  - Cannot be changed or updated
  - Its behavior is described in the processor datasheet

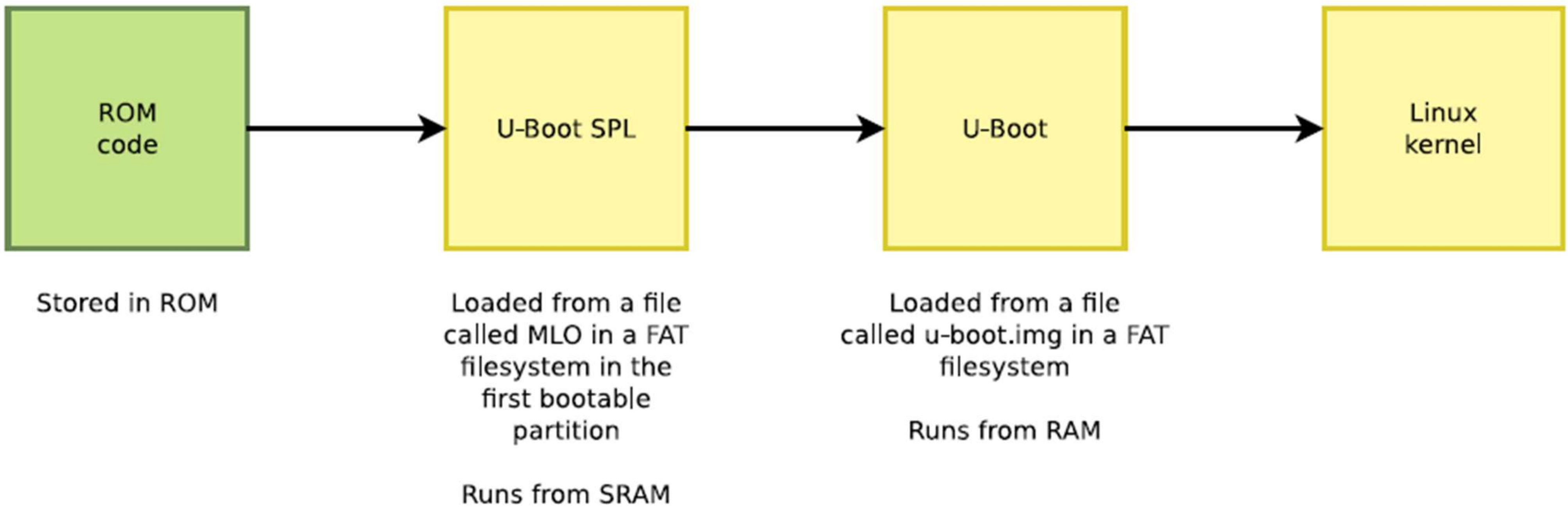
- Responsible for finding a suitable bootloader, loading it and running it
  - From NAND/NOR flash, from USB, from SD card, from eMMC, etc.
  - Well defined location/format

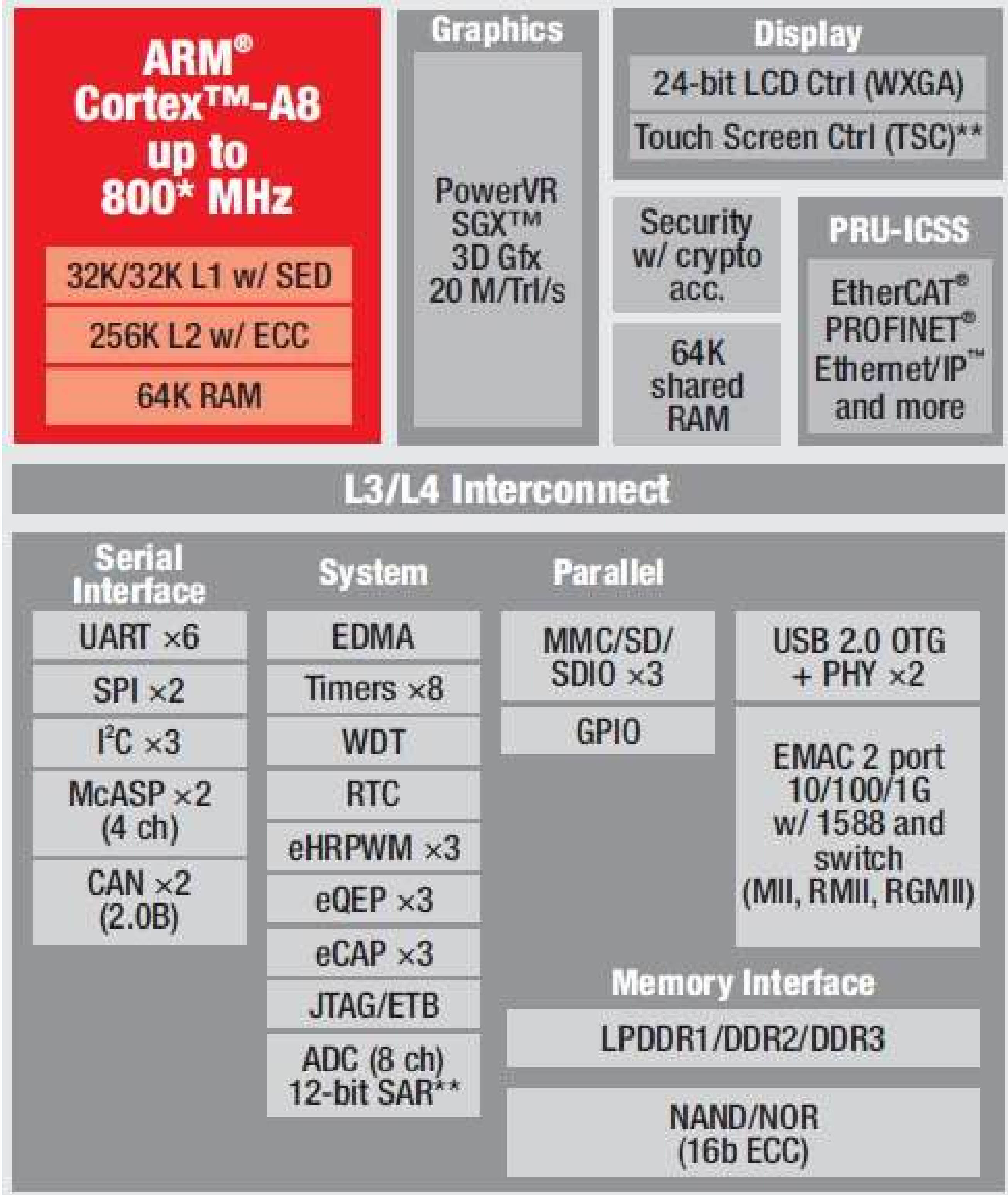
- Generally, runs with the external RAM not initialized, so it can only load the bootloader into an internal SRAM
  - Limited size of the bootloader, due to the size of the SRAM
  - Forces the boot process to be split in two steps: first stage bootloader (small, runs from SRAM, initializes external DRAM), second stage bootloader (larger, runs from external DRAM)

Multi-stage bootloader



ARMv7 Boot Sequence




ARM® Cortex™-A8 up to 800* MHz
32K/32K L1 w/ SED
256K L2 w/ ECC
64K RAM
Graphics
PowerVR SGX™ 3D Gfx 20 M/Trl/s
Display
24-bit LCD Ctrl (WXGA)
Touch Screen Ctrl (TSC)**
Security w/ crypto acc.
64K shared RAM
PRU-ICSS
EtherCAT® PROFINET® Ethernet/IP™ and more
L3/L4 Interconnect
Serial Interface
UART ×6
SPI ×2
I2C ×3
McASP ×2 (4 ch)
CAN ×2 (2.0B)
System
EDMA
Timers ×8
WDT
RTC
eHRPWM ×3
eQEP ×3
eCAP ×3
JTAG/ETB
ADC (8 ch) 12-bit SAR**
Parallel
MMC/SD/SDIO ×3
GPIO
USB 2.0 OTG + PHY ×2
EMAC 2 port 10/100/1G w/ 1588 and switch (MII, RMII, RGMII)
Memory Interface
LPDDR1/DDR2/DDR3
NAND/NOR (16b ECC)

Practical : 2

Environment variables commands

- U-Boot can be configured through environment variables, which affect the behaviour of the different commands.
- Environment variables are loaded from ash to RAM at U-Boot startup, can be modified and saved back to ash for persistence
- There is a dedicated location in ash to store U-Boot environment, defined in the board configuration file.
- Commands to manipulate environment variables:
  - **printenv**, shows all variables
  - **printenv <variable-name>,** shows the value of one variable
  - **setenv <variable-name> <variable-value>,** changes the value of a variable, only in RAM
  - **saveenv**, saves the current state of the environment to flash

Important U-Boot env variables

- **bootcmd**, contains the command that U-Boot will automatically execute at boot time after a configurable delay, if the process is not interrupted
- **bootargs**, contains the arguments passed to the Linux kernel, covered later
- **serverip**, the IP address of the server that U-Boot will contact for network related commands
- **ipaddr**, the IP address that U-Boot will use
- **netmask**, the network mask to contact the server
- **ethaddr**, the MAC address, can only be set once
- **bootdelay**, the delay in seconds before which U-Boot runs bootcmd
- **autostart**, if yes, U-Boot starts automatically an image that has been loaded into memory

Kernel Images (zImage vs uImage)

vmlinux :-
vmlinux is the uncompressed, ELF format kernel image produced during compilation

zImage :-
A compressed Linux kernel image created by the kernel build system.

uImage :-
A zImage + a U-Boot-specific 64-byte header added via the mkimage tool.
Header contains:

- Kernel Load address
- Entry point
- Image type
- OS type
- Compression type
- CRC checksum

Older uboot versions require a special kernel image format: uImage
NOTE - But Recent versions of U-Boot can directly boot the zImage binary

Telechip Boot Mode Selection

- TCC805x consists of five processors: MICOM, HSM, SC, CA72, and CA53.
- TCC805x can load the boot image from different types of storages (such as SNOR, eMMC, or UFS).

Processors	Boot Level	Description
MICOM: Cortex-R5 (CR5)	CR5-BL0	MICOM Chipboot ROM Code
	CR5-BL1	MICOM Bootloader
Hardware Security Module (HSM): SC000	HSM-BL0	HSM Chipboot ROM Code
	HSM-F/W	HSM Firmware
Storage Core (SC): Cortex-M4 (CM4)	SC-BL0	Storage Core Chipboot ROM Code
	SC-F/W	Storage Core Firmware
Main Core: Cortex-A72 (CA72) Sub-core: Cortex-A53 (CA53)	CA72/CA53-BL0	CA72/CA53 Trusted Firmware-A (TF-A) BL1 (ROM Code)
	CA72/CA53-BL1	CA72/CA53 TF-A BL2
	CA72/CA53-BL2	CA72/CA53 TF-A BL31
	CA72/CA53-BL3	CA72/CA53 U-Boot

Telechip Boot Mode Selection

- This Table describes the processors and the corresponding Boot Levels (BLs).
- Depending on the BL, it is determined that which processor runs, and which image is loaded

Processors	Boot Level	Description
MICOM: Cortex-R5 (CR5)	CR5-BL0	MICOM Chipboot ROM Code
	CR5-BL1	MICOM Bootloader
Hardware Security Module (HSM): SC000	HSM-BL0	HSM Chipboot ROM Code
	HSM-F/W	HSM Firmware
Storage Core (SC): Cortex-M4 (CM4)	SC-BL0	Storage Core Chipboot ROM Code
	SC-F/W	Storage Core Firmware
Main Core: Cortex-A72 (CA72) Sub-core: Cortex-A53 (CA53)	CA72/CA53-BL0	CA72/CA53 Trusted Firmware-A (TF-A) BL1 (ROM Code)
	CA72/CA53-BL1	CA72/CA53 TF-A BL2
	CA72/CA53-BL2	CA72/CA53 TF-A BL31
	CA72/CA53-BL3	CA72/CA53 U-Boot

Telechip Boot Mode Selection

- This Table describes the Boot Modes (BMs).
- Depending on whether MICOM is used or not, BMs are categorized into two boot sequences:
    - Normal Boot with MICOM
    - Normal Boot without MICOM

Boot Sequence	Boot Mode	Description
Normal Boot with MICOM	Normal Boot (eMMC)	MICOM loads the images from SNOR and SC, CA72, and CA53 load the images from eMMC.
	Normal Boot (UFS)	MICOM loads the images from SNOR and SC, CA72, and CA53 load the images from UFS.
Normal Boot without MICOM	Normal Boot (eMMC) without MICOM	SC, CA72, and CA53 load the images from eMMC. MICOM is not operating.
	Normal Boot (UFS) without MICOM	SC, CA72, and CA53 load the images from UFS. MICOM is not operating.
-	Firmware Download	MICOM receives the images from USB Host. SC/CA72/CA53 are not operating.

Telechip Boot Sequence

- Depending on the operation of MICOM, BMs are categorized into two boot sequences:
  - Normal Boot with MICOM
  - Normal Boot without MICOM.

The Normal Boot with MICOM includes the following BMs:
  - Normal Boot (eMMC)
  - Normal Boot (UFS)

The Normal Boot without MICOM includes the following BMs:
  - Normal Boot (eMMC) without MICOM
  - Normal Boot (UFS) without MICOM

NOTE :-
- The BM sequence of using eMMC and using UFS are basically the same except the storage.
- In the case of BM with MICOM, SNOR is required because MICOM loads the images from SNOR and executes the firmware as XIP in SNOR

Linux Device Drivers

What is device driver

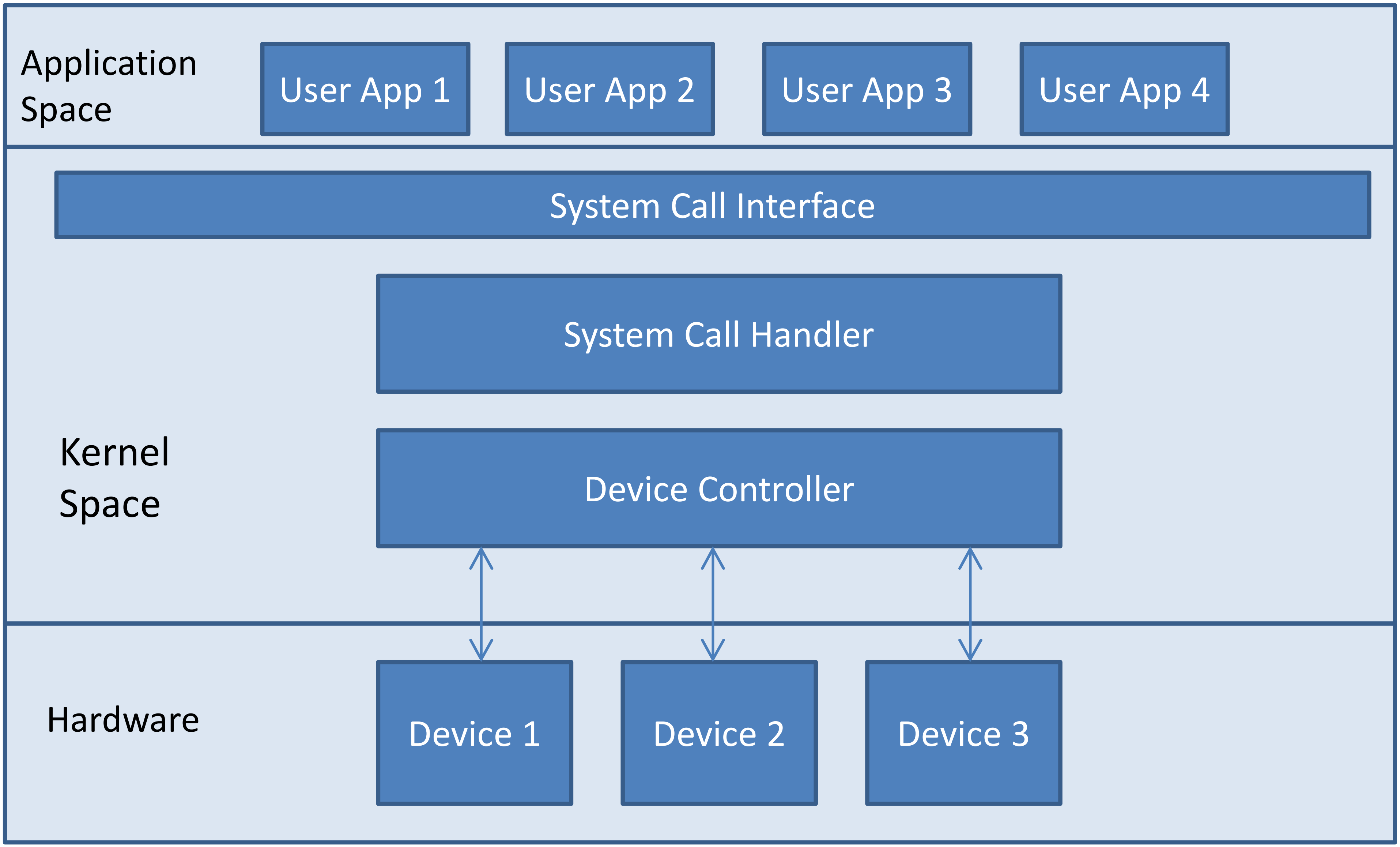
- Software layer between application & hardware
- Used to control the device control & access the data from the device
- Linux kernel must have device driver for each peripheral of the system
- Linux Device Driver
  – Present in built with kernel
  – Loadable as module in run time

Why Drivers Are Needed

Kernel cannot directly control hardware or interact with user programs.

- A driver provides the glue between:
  - User space: Apps, commands (cat, echo, etc.)
  - Kernel space: Hardware, memory, buses

Linux Device Driver Architecture



Types of Device Driver

- Character Driver
  – Can be accessed as a stream of bytes like a file
  – Examples: UART, GPIO etc.
- Block Driver
  – Device that can host a file systems like a disk
  – Handles I/O operation with one or more blocks
  – Examples: HDD, SSD etc.
- Network Driver
  – Device that any network transaction through an interface
  – Interface is in charge of sending & receiving data packets
  – Examples: Ethernet, WiFi etc.

Comparison Table

Feature	Character Drivers	Block Drivers	Network Drivers
Data Unit	Stream of bytes	Fixed-size blocks	Packets
Access Pattern	Sequential	Random	Packet-oriented
Examples	Keyboards, serial ports	Hard drives, SSDs	Ethernet cards, Wi-Fi
Device Files	/dev/ttyS0, /dev/input	/dev/sda, /dev/sdb	No device files (e.g., eth0)
System Calls	read(), write()	read(), write()	socket(), send(), recv()
Buffering	Unbuffered	Buffered	Packet-based
Use Case	Low-latency devices	Storage devices	Network communication

Driver vs Module

Kernel Module
A kernel module is any piece of code that can be loaded into or removed from the Linux kernel at runtime (using insmod/rmmod).

Device Driver

- A driver is a specific type of kernel module that knows how to:
- Communicate with hardware (like I2C, UART, GPIO, etc.)
- Or represent virtual devices (like loopback, /dev/null)
- Register with subsystems like:
    - Character device (/dev/xyz)
    - Block device (/dev/sda)
    - Platform bus / I2C bus / USB stack

“All drivers are modules, but not all modules are drivers.”

Driver vs Module

Feature	Kernel Module	Driver (Kernel Module)
Is it loadable?	Yes	Yes
Talks to hardware?	Not usually	Yes
Creates /dev/ file?	No	Yes (character/block drivers)
Registers with subsystem?	No	Yes
Needed for hardware?	Optional (demo only)	Required
Example	hello.ko	my_gpio_driver.ko

Practical : 3

Module Initialization

- module_init
  - Adds a special section, stating where the module's initialization function to be found
  - Without this, module initialization is never called
  - Can register many different types of facilities
- Tags

  __init => module loader drops the initialization function after loading

  __initdata => data used only during initialization

  __exit => can only be called at module unload

Cleanup function

- Every module also contains a cleanup function
  - Unregisters interface & return all the resources to system before module is removed
  - Cleanup function does not have a return value
- Tags

  __exit => Functions will be used only in module cleanup

  __exitdata => data used only during module cleanup
- module_exit
  - Enables the kernel to find the cleanup function
  - If module does not have cleanup function, then it cannot be removed from kernel

Hello Module (hello.c)

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE(“GPL");
static int hello_init(void)
{
    printk(KERN_ALERT “Inserting a module\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT “Removing a module\n");
}
module_init(hello_init);
module_exit(hello_exit);
```

Compiling & loading

- Compilation can be done with make
  - After compilation, Kernel object will be created with a extension of .ko
- Inserting a module into kernel
  - insmod hello.ko
- Removing a module from kernel
  - rmmod hello.ko

Module Utilities

- insmod
- rmmod
- modprobe
- lsmod
- /sys/modules
- /proc/modules
- /proc/devices

Major & Minor Numbers

- Device Number represented in 32 bit
  - 12 Bits => Major Number
  - 20 Bits => Minor Number
- Device identification
  - c => character driver
  - b => block driver
- dev_t => device number representation
- MAJOR(dev_t dev);
- MINOR(dev_t dev);
- MKDEV(int major, int minor);

Allocating Device Numbers

- register_chrdev_region
  - dev_t first => beginning device number
  - count => number of contiguous device numbers
  - name => name of the device
  - 0 on success, negative error code on failure
- alloc_chrdev_region
  - dev_t *dev => output parameter on completion
  - firstminor => requested first minor
  - count => number of contiguous device numbers
  - name => name of the device
  - 0 on success, negative error code on failure

Freeing device number

- unregister_chrdev_region
  - dev_t first => beginning of device number range to be freed
  - count => number of contiguous device numbers
  - Returns void
  - The usual place to call would be in module’s cleanup

mknod

- Creates block or character special device files
- mknod [OPTIONS] NAME TYPE [MAJOR MINOR]
  - NAME => special device file name
  - TYPE
    - c, u => creates a character (unbuffered) special file
    - b => creates a block special file
    - p => creates a FIFO
  - MAJOR => major number of device file
  - MINOR => minor number of device file

File Operations

- owner => pointer to module that owns structure
- open
- release
- read
- write
- poll
- ioctl
- mmap
- llseek
- readdir
- flush
- fsync

File Ops - Example

```
struct file_operations scull_fops = {
    .owner = THIS_MODULE,
    .llseek = hello_llseek,
    .read = hello_read,
    .write = hello_write,
    .ioctl = hello_ioctl,
    .open = hello_open,
    .release = hello_release,
};
```

Char Device Registration

- struct cdev
- Allocating dynamically
  - cdev_alloc
- cdev_init
  - cdev => cdev structure pointer
  - fops => file operations structure pointer
- cdev_add
  - cdev => cdev structure pointer
  - dev_t num => first device number
  - count => number of device numbers to be associated

Char device removal

- cdev_del
  - cdev => cdev structure pointer
- cdev structure pointer should not be accessed after passing it to cdev_del

FOPS - open & release

- inode => inode structure pointer
- filp => file structure pointer
- returns 0 or error code based on open

FOPS – read & write

- filp => file structure pointer
- buff => character buffer to be written by read / read by write function
- count => number of bytes
- offp => offset
- Returns 0 or negative error code accordingly

FOPS - ioctl

- inode => inode structure pointer
- filp => file structure pointer
- cmd => ioctl command
- args => ioctl command arguements
- returns 0 or error code

copy_to_user

- to => user pointer where data to be copied
- from => kernel pointer is data source
- count => number of bytes to be copied
- Used during driver read operations

copy_from_user

- to => kernel pointer where data to be copied
- from => user pointer is data source
- count => number of bytes to be copied
- Used during driver write operation

Practical:

- Write Simple Character Driver
- Write Character Driver to control GPIO

VFS (Virtual File System)

- The Virtual Filesystem (VFS) is the subsystem of the kernel that implements the filesystem-related interfaces provided to user-space programs.
- All filesystems rely on the VFS to allow them to coexist and interoperate. •
- This enables programs to use standard Unix system calls to read and write to different filesystems on different media.

VFS (Virtual File System)

- The VFS is the glue that enables system calls such as open(), read(), write(),copy() and move() to work regardless of the filesystem or underlying physical medium.
- In older operating systems (think DOS), this would never have worked.
- Modern operating systems abstract access to the filesystems via a virtual interface that such interoperation and generic access is possible.
- New filesystems and new varieties of storage media can find their way into Linux, and programs need not be rewritten or even recompiled.

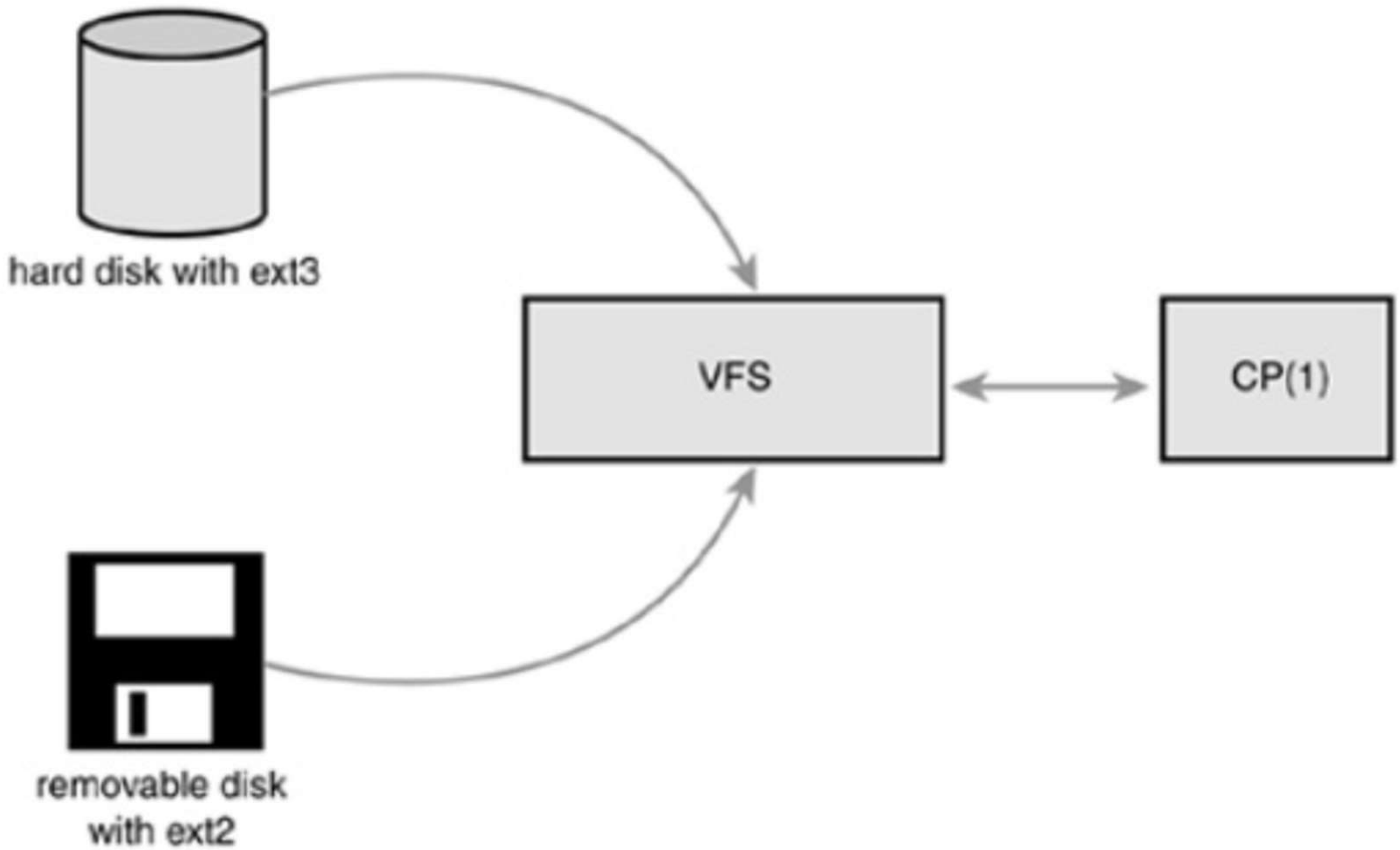


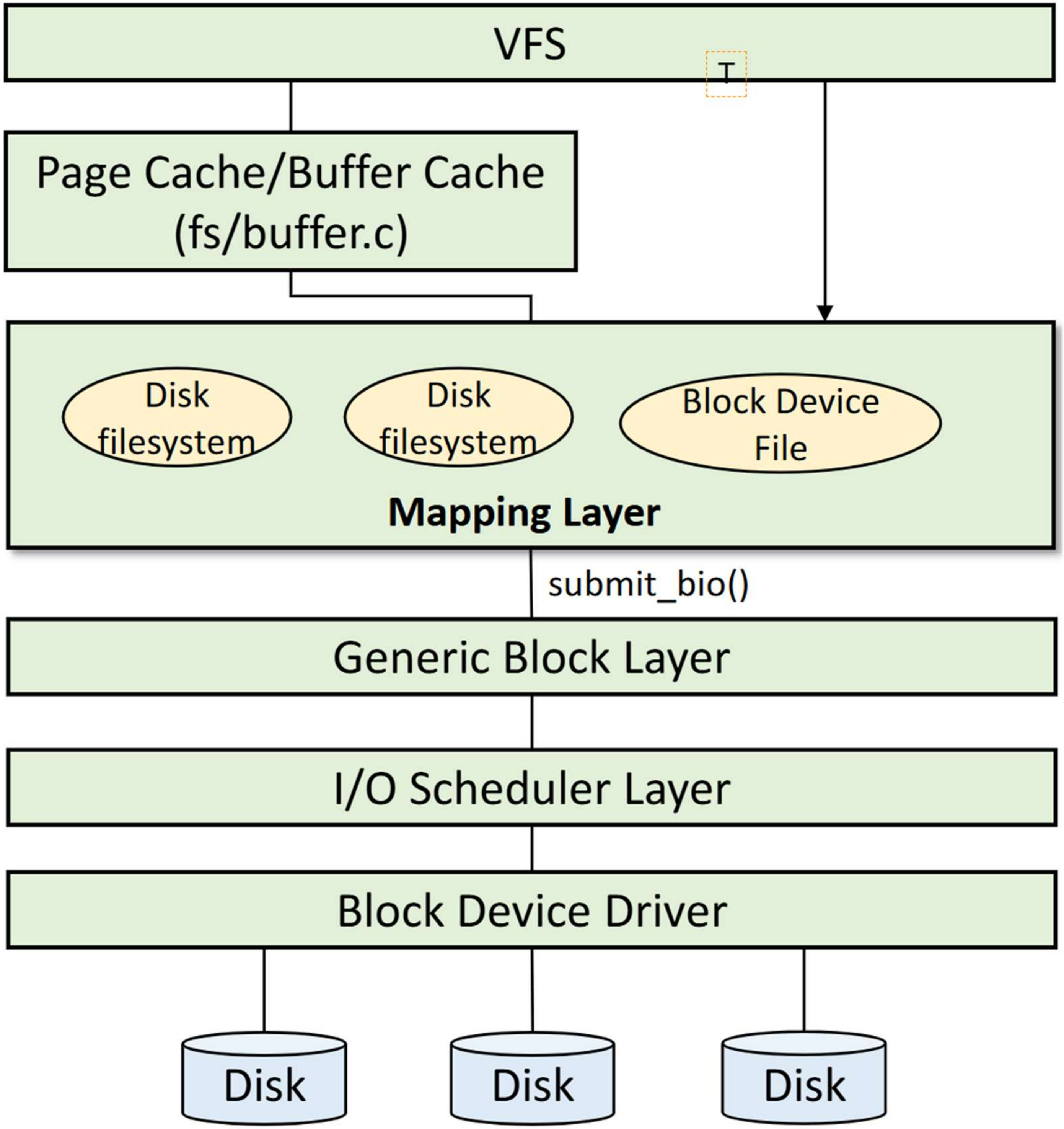
Figure1. The VFS in action: Using the cp(1) utility to move data from a hard disk mounted as ext3 to a removable disk mounted as ext2. Two different filesystems, two different media. One VFS.

write(f, &buf, len);

For Example =>

Consider a simple user-space program that does **write(f, &buf, len);**

The flow of data from user-space issuing a write() call, through the VFS's generic system call, into the filesystem's specific write method, and finally arriving at the physical media.


VFS
T
Page Cache/Buffer Cache
(fs/buffer.c)
Disk
filesystem
Disk
filesystem
Block Device
File
Mapping Layer
submit_bio()
Generic Block Layer
I/O Scheduler Layer
Block Device Driver
Disk
Disk
Disk

Platform Drivers

GPIO Drivers

GPIO Drivers

- Discuss and Explain GPIO Initialization Framework
- Discuss and Explain GPIO Operations Framework
- Discuss and Explain GPIO key Initialization Framework
- Discuss and Explain GPIO Key Operations Framework

NOTE – Write the GPIO Driver