DEVICE DRIVERS – CONCEPT AND REAL USE
(KERNEL MODULES OVERVIEW)

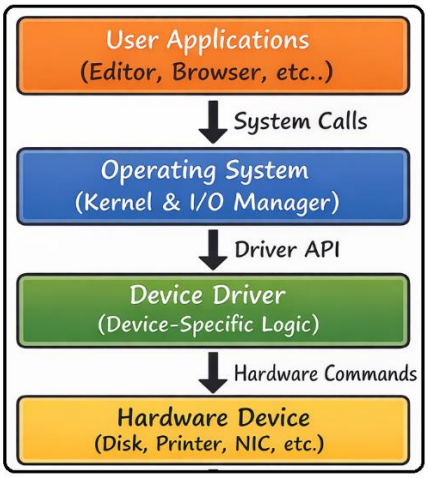
Concept of Device Drivers:

A **device driver** is a specialized system software that allows the **operating system (OS)** to communicate with and control **hardware devices** such as printers, keyboards, disks, network cards, and displays.

- Acts as an **interface between hardware and the OS kernel.**
- Translates **OS requests** into **hardware-specific commands.**
- Ensures hardware works correctly without applications knowing device details.
- Device drivers are part of the **kernel** and operate in **kernel mode**, giving them direct access to hardware resources such as I/O ports, memory, and interrupts.

Why do we need device drivers?

- It is dangerous to give user applications direct access to the hardware. User activities run in *user space* and when in need of hardware resources, implement a set of device-independent standardized calls. *The role of the device driver* is to **map** those calls to device-specific operations that can act on real hardware.



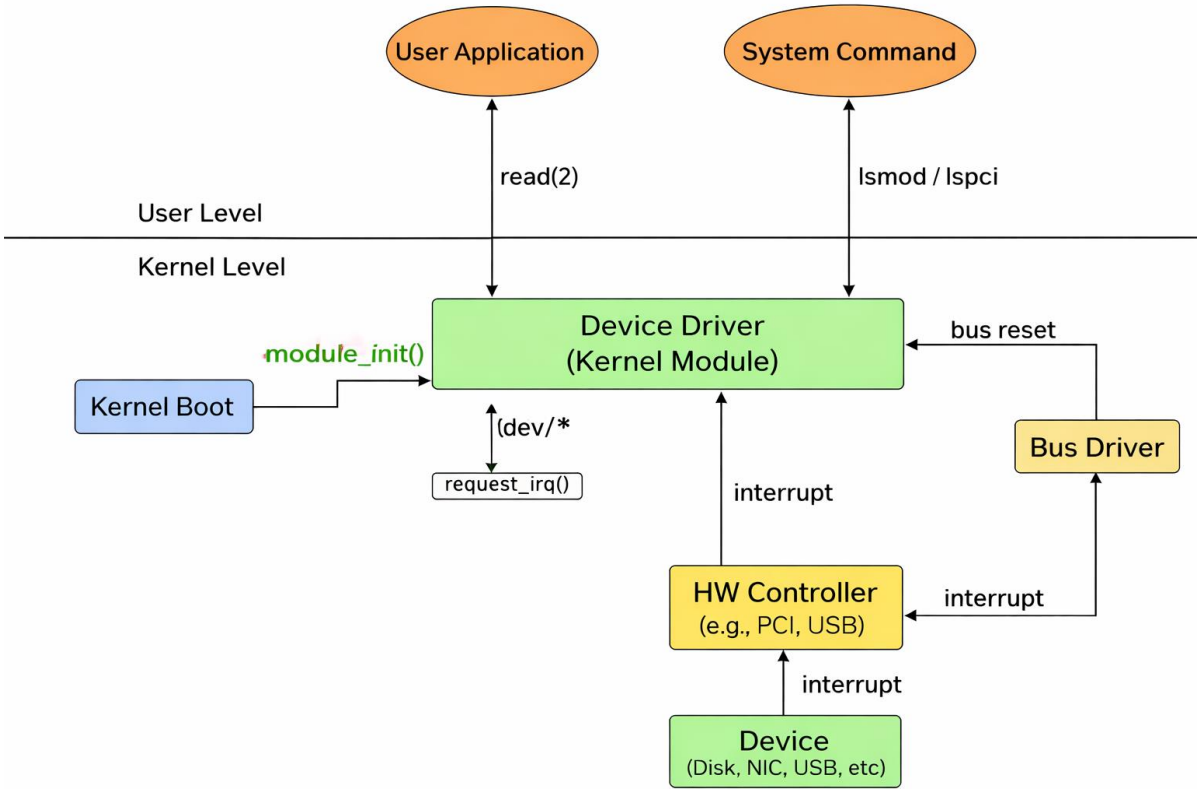
Functions of a Device Driver:

A device driver performs the following key functions:

- **Device initialization** and configuration
- **Handling I/O requests** from the OS
- **Interrupt handling** from devices
- **Error detection and recovery**
- **Data transfer** between memory and device
- **Power management** of devices

Drivers are accessed in the following situations:

- **System initialization.** The kernel calls device drivers during system initialization to determine which devices are available and to initialize those devices.
- **System calls from user processes.** The kernel calls a device driver to perform I/O operations on the device such as open(2), read(2), and ioctl(2).
- **User-level requests.** The kernel calls device drivers to service requests from commands such as prtconf(1M).
- **Device interrupts.** The kernel calls a device driver to handle interrupts generated by a device.
- **Bus reset.** The kernel calls a device driver to re-initialize the driver, the device, or both when the bus is reset. The bus is the path from the CPU to the device.



1. User Level (User Space):

User Application:

- Examples: editor, media player, database program.
- Applications **cannot access hardware directly.**
- They request I/O services using **system calls.**

System Calls:

- read(2) is shown as a typical system call.
- System commands such as lsmod and lspci are used to:

- View loaded kernel modules.
- Detect hardware devices.

- These calls transfer control from **user mode to kernel mode.**

2. Kernel Level (Kernel Space):

Device Driver (Kernel Module):

- The device driver is implemented as a **Loadable Kernel Module (LKM).**
- It runs with **full kernel privileges.**
- Acts as an **interface between the kernel and hardware.**

Key Responsibilities:

- Handle system calls (read, write, ioctl).
- Translate generic OS requests into device-specific commands.
- Manage hardware resources.

3. Module Initialization (Kernel Boot / Dynamic Loading):

- During **kernel boot** or when a module is loaded using modprobe, the kernel invokes: **module_init()**
- This function:
  - Registers the driver with the kernel.
  - Allocates resources.
  - Creates device files under /dev/*.

4. Device File Interface (/dev/*):

- Device files act as a **communication bridge** between user space and kernel space.
- When an application accesses /dev/*, the request is routed to the corresponding driver.

5. Bus Driver (PCI / USB / I2C):

- The device driver communicates with the hardware via a **bus driver.**
- The bus driver:
  - Manages device enumeration.
  - Handles bus reset.
  - Provides standardized access to devices.

6. Hardware Controller:

- Located between the bus and the physical device.
- Executes commands sent by the device driver.
- Manages device registers and data transfer.

7. Hardware Device:

- Actual physical component:
  - Disk
  - Network Interface Card (NIC).
  - USB device.
- Performs requested operations such as reading or writing data.

8. Interrupt Handling:

Interrupt Flow:

1. Device completes an operation.
2. Device raises an **interrupt.**
3. Hardware controller forwards the interrupt.
4. Device driver handles it using: request_irq().
5. Driver executes an **Interrupt Service Routine (ISR).**

Importance of Interrupts:

- Avoids CPU polling.
- Improves system performance.
- Enables asynchronous I/O.

Types of Device Drivers:

1. Character Device Drivers:

A **character device driver** handles devices that transfer data **one byte (character) at a time** in a **sequential manner**.

Key Characteristics:

- Data is transferred as a **stream of bytes.**
- No fixed block size.
- Usually **unbuffered.**
- Supports sequential access only.
- Uses simple read/write operations.

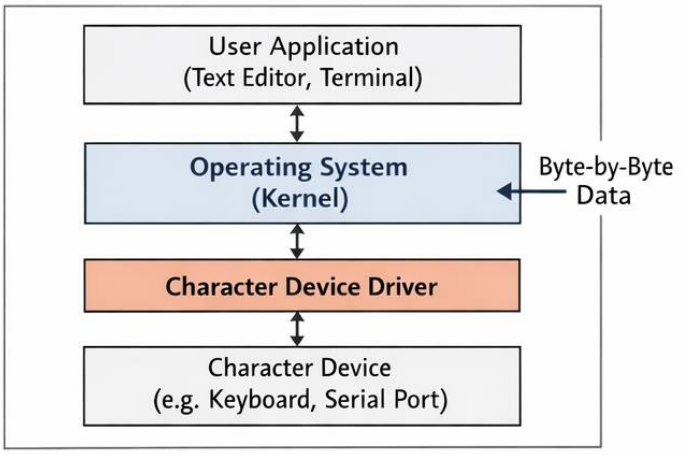
Kernel Interface:

- Represented by **character special files** in /dev.
- Uses the kernel structure: **struct file_operations**
- Supports system calls: open(), read(), write(), close(), ioctl()

Working Mechanism:

1. Application opens a character device file.
2. Read/write request is sent to the kernel.

3. Kernel forwards the request to the character driver.
4. Driver communicates directly with the device.
5. Data is transferred byte by byte.



Examples:

- Keyboard
- Mouse
- Serial ports (/dev/ttyS0)
- Sensors
- Audio input devices

Advantages:

- Simple design.
- Suitable for low-speed devices.
- Minimal overhead.

Limitations

- Not suitable for large data transfer.
- No random access.

2. Block Device Drivers:

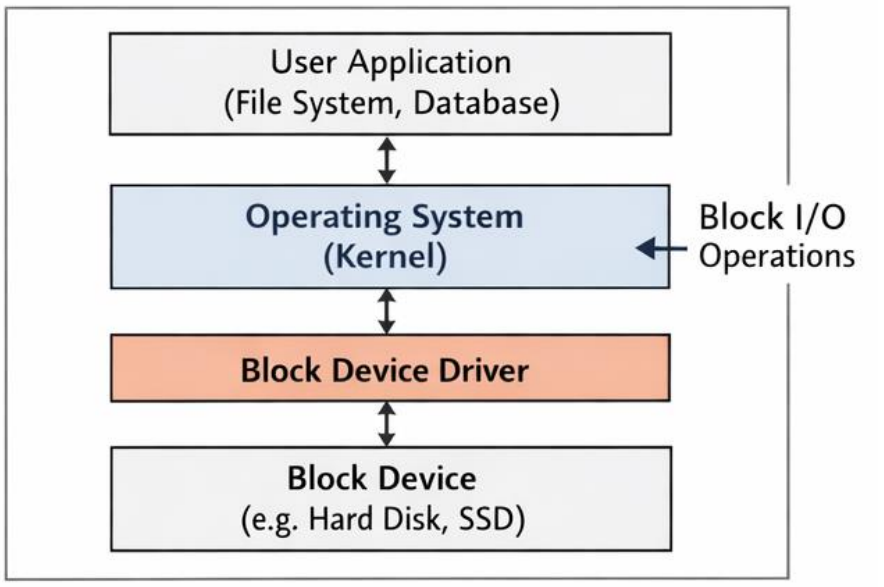
A **block device driver** manages devices that store and transfer data in **fixed-size blocks** and allow **random access**.

Key Characteristics:

- Data transferred in blocks (e.g., 512 bytes).
- Supports random access.
- Uses buffering and caching.
- High throughput.
- Works closely with the file system.

Kernel Interface:

- Represented by **block special files** in /dev.
- Uses kernel structure: **struct block_device_operations.**
- Integrated with the Linux **I/O scheduler.**



Working Mechanism:

1. Application requests file access.
2. File system translates the request into block operations.
3. Kernel sends request to the block driver.
4. Driver reads/writes blocks from storage.
5. Data is returned to the application.

Examples:

- Hard Disk Drives (HDD)
- Solid State Drives (SSD)
- USB storage devices
- SD cards

Advantages:

- Efficient large data handling.
- Supports caching and buffering.
- Enables file systems.

Limitations:

- More complex than character drivers.
- Requires block management overhead.

3. Network Device Drivers:

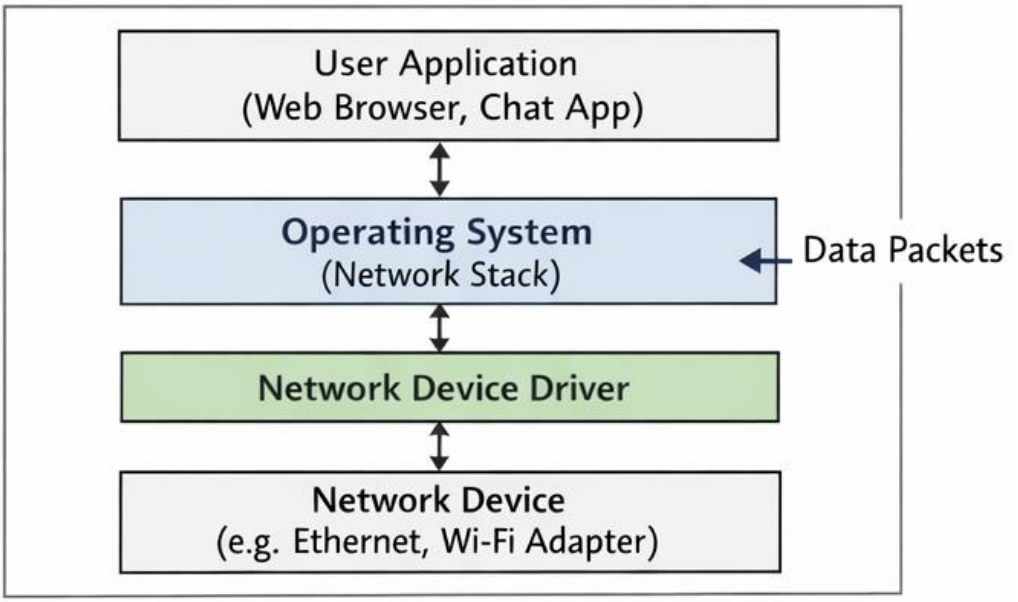
A **network device driver** manages **packet-based communication devices** used for data transmission over a network.

Key Characteristics:

- Data handled as **packets**, not bytes or blocks.
- Does not use traditional read()/write().
- Works with Linux **network stack.**
- Event-driven and interrupt-based.

Kernel Interface:

- Uses kernel structure: **struct net_device**
- Accessed via **socket interface.**
- Integrated with TCP/IP protocol stack.



Working Mechanism:

1. Application sends data using sockets.
2. Network stack processes the data.
3. Network driver converts data into packets.
4. NIC transmits packets over the network.
5. Incoming packets generate interrupts handled by the driver.

Examples:

- Ethernet drivers
- Wi-Fi drivers
- Bluetooth drivers
- Virtual network interfaces

Advantages:

- High-speed communication.
- Protocol-independent.
- Supports multiple network standards.

Limitations:

- Complex implementation.
- Requires synchronization and buffering.

Comparison:

Feature	Character Driver	Block Driver	Network Driver
Data Transfer	Byte-by-byte	Block-based	Packet-based
Access Type	Sequential	Random	Stream/Packet
Buffering	Usually No	Yes	Yes
File System Use	No	Yes	No
Examples	Keyboard	HDD, SSD	Ethernet

Real Use of Device Drivers:

Kernel module device drivers are used in **real systems** to allow the operating system to **detect, control, and communicate with hardware dynamically**, without rebooting the system.

1. Plug-and-Play Hardware (Most Common Use):

Example: USB Pen Drive

- When a USB pen drive is inserted:
  - Kernel detects the device
  - Corresponding **USB storage driver module** is loaded automatically
  - Device becomes available as /dev/sdX

✓ **No reboot required**

✓ Driver loaded **on demand**

2. Network Devices (Wi-Fi / Ethernet):

Example: Wi-Fi Adapter

- When Wi-Fi is enabled:
  - Kernel loads the **network driver module**

- Driver communicates with the network card
- Data packets are transmitted and received

✓ Used in laptops, mobiles, servers

✓ Supports dynamic enable/disable

3. Storage Devices:

Example: Hard Disk / SSD

- Block device drivers handle:
  - File system access
  - Disk read/write operations
- Driver works as a kernel module for:
  - SATA
  - NVMe
  - USB storage

✓ High-speed data transfer

✓ Uses buffering and caching

4. Input Devices:

Example: Keyboard and Mouse

- Character device drivers:
  - Capture key presses
  - Send input events to applications
- Loaded automatically during system startup

✓ Used in desktops, laptops, embedded systems

5. Printers and Scanners:

Example: Printer Driver

- Printer connected via USB or network
- Kernel module driver:
  - Converts OS print requests into device commands
  - Handles interrupts and data flow

✓ Used in offices and institutions

6. Embedded Systems:

Example: Sensor Interface (Temperature, Motion)

- Kernel module drivers control:
  - Sensors
  - GPIO pins

- ADC/DAC
- Used in:
  - IoT devices
  - Medical equipment
  - Industrial automation

✓ Real-time hardware control

✓ Low-level access required

7. Power Management & Battery Control:

Example: Laptop Battery Driver

- Kernel module monitors:
  - Battery level
  - Charging status
- Communicates with hardware controller

✓ Enables power saving

✓ Improves battery life

8. Graphics and Display Drivers:

Example: GPU Driver

- Kernel module controls:
  - Display output
  - Graphics acceleration
- Works with user-space libraries

✓ Used in gaming, design, AI workloads

9. Security Devices:

Example: Biometric / Smart Card Reader

- Driver handles authentication hardware
- Communicates securely with kernel

✓ Used in secure login systems

Real-World System Flow (Simple)

1. Hardware connected
2. Kernel detects device
3. Driver module loaded dynamically
4. Driver communicates with hardware
5. Application accesses device via OS