Block Device Driver Development: From Theory to Implementation

Table of Contents

I Introduction to Device Drivers

1.1 What is a Device Driver?

A **device driver** is a specialized software that acts as a bridge between the operating system and hardware devices, allowing the OS and applications to communicate with hardware without needing to know its internal details.

Example:

A **block device driver** allows the OS to read from and write to a hard disk or SSD using standard system calls (read(), write()), while the driver handles the low-level communication with the disk hardware.

1.2 Role in Operating System Architecture

Device drivers are an **essential part of the operating system** that enable communication between **hardware devices** and the **software layer**. They sit between the **kernel** and the **hardware**, allowing the OS to control devices without needing to know hardware-specific details.

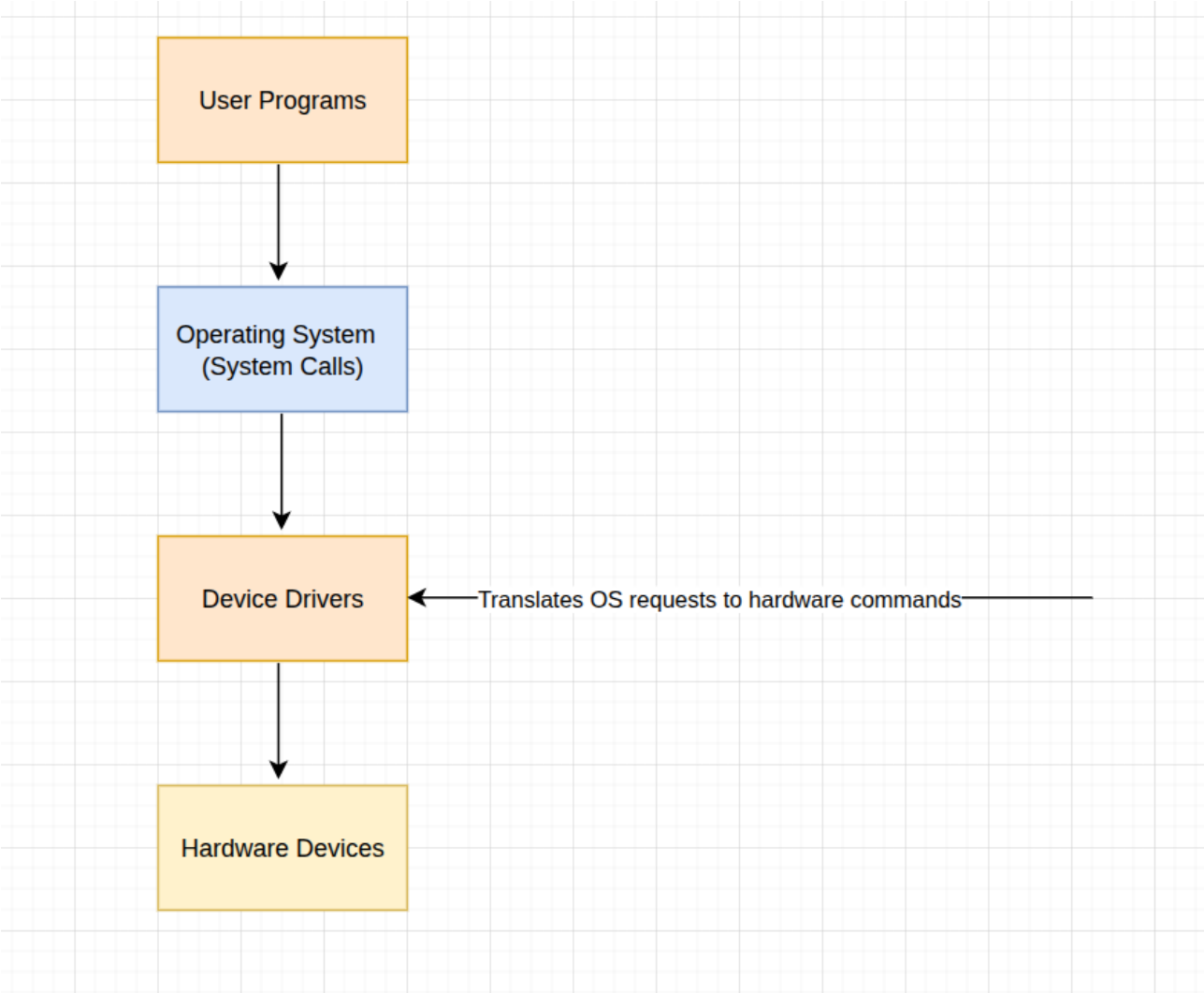
Key Roles

1. **Hardware Abstraction**
    - Device drivers hide the hardware complexity from the OS and applications.
    - They provide a **standard interface** for interacting with devices.
2. **Interface to Kernel**
    - Drivers implement functions that the kernel calls to perform I/O operations.
    - Examples: open(), read(), write(), ioctl().
3. **Resource Management**

- Allocate and manage hardware resources such as I/O ports, memory, DMA channels, and interrupts.
- Ensure devices are used safely and efficiently.

4. **Interrupt Handling**
    - Drivers handle hardware-generated interrupts and notify the OS when an event occurs.
    - Example: Keyboard driver detects a key press and informs the OS.
5. **Data Transfer Management**
    - Manage the flow of data between user-space applications and hardware.
    - Convert high-level OS requests into low-level hardware commands.
6. **Plug-and-Play / Hot-Swapping Support**
    - Drivers allow dynamic detection and initialization of hardware devices.
    - Example: USB driver automatically detects a plugged USB device.

In OS Architecture Diagrammatically



Example:
In a Linux system, a **block device driver** allows applications to read/write files on a hard disk. The OS sends a `read()` system call → the driver translates it into hardware-level commands → data is fetched from disk → returned to the application.

1.3 Device Driver vs. Application Software

Device Driver:

Software that enables the operating system to **communicate with hardware devices**.Interacts **directly** with hardware registers, memory, and interrupts. Acts as a **translator** between OS and hardware. Example: Block device driver, USB driver, Wi-Fi driver.

Application Software:

Software used directly by the user to perform tasks.Interacts with

hardware **indirectly** through system calls (`read()`, `write()`, `ioctl()`). Provides functionality and features to users (editing, browsing, gaming).

Example: Browser, Media player, Text editor.

Feature	Device Driver	Application Software
Purpose	Hardware control	User tasks
Runs In	Kernel space	User space
Hardware Access	Direct	Indirect (via driver)
Stability Impact	OS may crash	App may crash
Example	Disk driver	Media player

Example

- **Device Driver Example:**
  A **block device driver** that controls /dev/sda (hard disk), handling read/write operations.
- **Application Software Example:**
  A **file manager** that lets the user browse, open, and copy files.

II. Types of Device Drivers

2.1 Classification by Access Method

2.1.1 Block Device Drivers

Block device drivers handle devices that read or write data in fixed-size blocks.

Examples: Hard disks, SSDs, USB mass-storage.

They support random access, meaning the system can read any block without reading previous blocks.

2.1.2 Character Device Drivers

Character device drivers manage devices that send or receive data **byte-by-byte (character streams)**.

Examples: UART/Serial port, keyboard, mouse, sensors.
They typically don't support random access; data is read in sequential order.

2.1.3 Network Device Drivers

These drivers manage communication between the system and a network interface.
Examples: Ethernet drivers, Wi-Fi drivers, Bluetooth network adapters.
They package and transmit data packets based on networking protocols.

2.2 Classification by Execution Mode

2.2.1 Kernel Space Drivers

Kernel space drivers run directly inside the operating system kernel.

- Fast and efficient
- Full access to hardware
- Requires careful coding (a bug can crash the OS)
  Examples: Linux device drivers (char/block), kernel modules.

2.2.2 User Space Drivers

User space drivers run outside the kernel, in normal application memory space.

- Safer (crashes don't affect OS)
- Slightly slower due to context switching
  Examples: FUSE file system drivers, some USB drivers using libusb.

2.3 Special Categories

2.3.1 Virtual Device Drivers

They emulate devices entirely in software.
Examples:

- Virtual network adapters (like tun/tap)
- Virtual disk devices used in virtualization (e.g., QEMU, VMware)

They allow the OS to interact with simulated hardware.

2.3.2 Platform Device Drivers

Drivers designed for **SoC (System-on-Chip)** hardware where devices are integrated on a board and do not connect via standard buses like PCI or

USB.
Examples: GPIO controllers, I2C/SPI controllers, onboard sensors in embedded boards (Qualcomm, TI, NXP).

2.3.3 USB Device Drivers

Drivers required to handle USB devices connected through the USB bus.
Examples: USB mouse, USB camera, USB storage, USB Wi-Fi dongles.
They include layers for enumeration, configuration, data transfer, and protocol handling.

III Block Device Drivers

This section explains block device drivers in depth: what block devices are, how their architecture is organized inside the kernel, and the key kernel data structures drivers use to integrate with the block I/O stack.

3.1 What are Block Devices?

3.1.1 Definition and Characteristics

A **block device** is a hardware or virtual device that stores and retrieves data in fixed-size blocks (sectors). Each I/O operation addresses whole blocks (commonly 512 bytes or 4096 bytes) and the OS can read or write any block independently (random access).

Key characteristics

- **Block-addressable:** I/O is expressed in block numbers (sectors), not a byte stream.
- **Random access:** You can read/write any block without sequential processing.
- **Aligned I/O:** Drivers and filesystems expect I/O aligned to block boundaries.
- **Buffered/cached:** Block devices are usually integrated with the OS buffer/cache layers to improve throughput and reduce physical I/O.

3.1.2 Difference from Character Devices

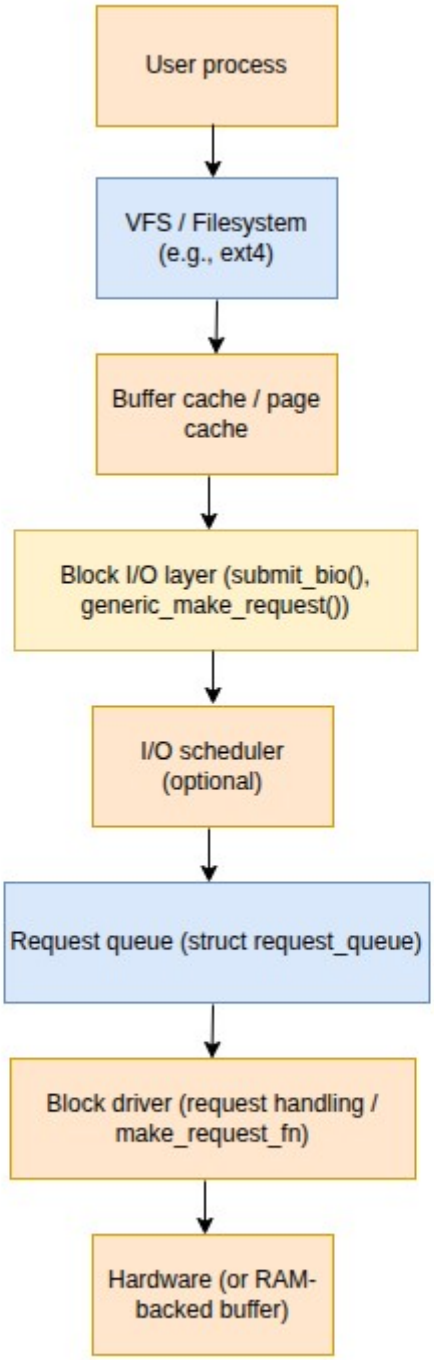
- **Block devices:** operate on fixed-size blocks, support random access, are typically used for storage media (disks). Interfaces often go through the block layer and buffer cache.

- **Character devices:** operate on byte streams, are usually sequential (read/write passes bytes directly to/from device), used for serial ports, keyboards, etc.

3.1.3 Typical Block Devices

- HDDs (mechanical hard drives)
- SSDs (solid-state drives)
- USB mass-storage devices (flash drives)
- SD / eMMC cards
- Virtual disks presented by hypervisors (QCOW, VMDK, virtio-blk)

3.2 Block Device Architecture



3.2.1 Request Queue Structure

The **request queue** is the kernel's per-device queue that aggregates I/O requests destined for a single gendisk. It handles:

- Request merging (combine adjacent requests)
- Reordering (by the I/O scheduler)
- Throttling and dispatching to the driver.

The queue exposes hooks for the driver to plug in:

- make_request_fn (direct bio handling path, bypassing request structures)
- request_fn / request_fn variant used when the generic request path is used

Drivers typically create and attach a struct request_queue via blk_init_queue() or newer blk_alloc_queue_node() APIs and supply a request handling function.

3.2.2 Block I/O Layer

The block I/O layer provides the top-level API for clients (filesystems, VFS) to submit I/O:

- **bio** (struct bio) is the kernel structure representing an I/O consisting of one or more pages (scatter-gather list) and sector range.
- **submit_bio()** / **generic_make_request()** are used to hand a bio to the block layer.
- The block layer hands I/O to the appropriate request queue, possibly runs it through an I/O scheduler, then converts/batches bios into struct request objects for drivers that use that path.

Newer kernels push drivers toward implementing make_request_fn (bio-based), which is more direct and reduces overhead by bypassing some request struct conversions.

3.2.3 Buffer Cache Interaction

- The **page cache / buffer cache** sits above the block I/O layer and caches block data to avoid hitting the device for every read.

- When user-space reads a file, the VFS first checks the page cache. On a cache miss, a bio is created to read the needed pages from the block device.
- Writes are often writeback: the filesystem writes to page cache and schedules a writeback bio to flush the pages to the device later.

Block drivers must cooperate with the cache (respect block sizes, sync/flush operations, and barrier semantics if applicable).

3.3 Key Data Structures (In-depth)

Below are the most important kernel structures a block driver interacts with, with explanations of common fields and how drivers use them.

> Note: field names evolve across kernel versions; the descriptions are conceptual and map to the common, stable pieces you will use.

3.3.1 struct block_device_operations

This structure provides the file-operations-like callbacks for block device special file behavior (opened via /dev/<name>). It is not the same as struct file_operations used by character devices, but it serves a similar purpose for block device semantics.

Common fields (example):

```
struct block_device_operations {
   int (*owner);                /* usually THIS_MODULE or &THIS_MODULE */
   int (*open) (struct block_device *, fmode_t);     /* optional */
   int (*release) (struct gendisk *, fmode_t);       /* optional */
   int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long); /*
often unused */
   int (*getgeo) (struct block_device *, struct hd_geometry *); /* geometry
for old tools */
   // ... other callbacks depending on kernel
};
```

Driver use:

- Provide open/release for device-level open/close handling (rarely needed for simple RAM disks).

- getgeo can be implemented for partitioning tools that query geometry.
- This struct is referenced from the gendisk (see below) via gendisk->fops (older kernels) or via other registration details.

3.3.2 struct gendisk

gendisk represents a whole disk device in the kernel. It describes device size, name, and ties together queue and block ops.

Key fields (common):

```
struct gendisk {
   int major;             /* device major number */
   int first_minor;       /* first minor for this disk */
   unsigned int minors;    /* number of minors */
   char disk_name[32];     /* "sda", "myblock", etc */
   struct request_queue *queue;   /* queue handling the I/O */
   struct block_device_operations *fops; /* block device ops */
   struct block_device *part; /* partition info (if any) */
   sector_t capacity;      /* number of 512-byte sectors */
   // ... many internal fields
};
```

Driver use:

- Allocate with alloc_disk(minors) (specify how many device nodes/minors you need).
- Set gendisk->major, first_minor, queue, fops, disk_name, and capacity (via set_capacity() helper).
- Register device major via register_blkdev() or use a pre-assigned major.
- Publish the device using add_disk(gendisk). This creates block device nodes (/dev/<diskname> entries typically via udev).

3.3.3 struct request_queue

Represents the per-disk queue that accepts bios/requests and dispatches them to the driver.

Important pieces:

```
struct request_queue {
   // scheduler hooks, queue limits
   void (*request_fn)(struct request_queue *q); // old-style request handler
```

```
    void (*make_request_fn)(struct request_queue *q, struct bio *bio); // bio-based
    spinlock_t queue_lock;  // protects queue internals
    // I/O scheduler pointer, limits, nr_requests, etc.
};
```

Driver use:

- Create/init a queue: dev->queue = blk_init_queue(my_request_fn, &dev->lock) (older) or blk_alloc_queue_node() + blk_queue_make_request() for make_request_fn.
- If using request_fn, driver calls blk_fetch_request() inside its handler to process requests.
- If using make_request_fn (preferred), the driver receives a bio directly and processes it (copy data, submit to hardware, complete bio).
- When unloading, cleanup via blk_cleanup_queue().

3.3.4 struct bio

A bio (block I/O) describes a single block I/O operation — it can span multiple pages (scatter-gather). It is central to efficient block I/O and is the unit passed around by the block layer.

Important fields (conceptually):

```
struct bio {
    struct bio_vec *bi_io_vec;  // scatter-gather array (pages + offsets + lengths)
    int bi_vcnt;            // number of bio_vec entries
    sector_t bi_iter.bi_sector; // starting sector for this bio
    unsigned int bi_opf;       // read/write flags + op code (REQ_OP_READ/WRITE)
    struct bio_set *bi_pool;   // pool used to allocate bio
    struct request *bi_private; // driver-used pointer (optional)
    bio_end_io_t *bi_end_io;   // completion callback
    struct page *bi_private;   // sometimes driver data
    // ... other fields like bi_idx, bi_size, etc.
};
```

Driver use:

- In make_request_fn, the driver receives pointer to a bio and must:

    - Inspect bio->bi_iter.bi_sector and bio->bi_size (or bio->bi_iter fields) to know where to read/write.
    - Iterate bio->bi_io_vec to access pages and offsets for copy operations.
    - Either handle the bio synchronously (copy data into device memory and call bio_endio() / bio_endio via bio_end_io_t) or queue it to hardware and return; when complete, call the bio completion callback to notify the block layer that I/O is done.
- Example completion: bio_endio(bio, 0) or call __blk_end_bio_user(bio, 0) depending on API and kernel ver.

Why bio matters: it avoids copying by using page frames directly (zero-copy), supports scatter-gather lists, and is oriented around pages — all of which are vital for high throughput.

Short Example: How these fit together in a simple RAM-backed driver

1. **Initialization (module_init)**

```
dev->queue = blk_init_queue(my_request_fn, &dev->lock);
dev->gd = alloc_disk(1);
dev->gd->queue = dev->queue;
dev->gd->fops = &my_block_ops;
dev->gd->major = register_blkdev(0, "myblock");
set_capacity(dev->gd, num_sectors);
add_disk(dev->gd);
```

2. **I/O Handling (bio-based pseudo-flow)**
    - make_request_fn(queue, bio) or my_request_fn fetches struct request.
    - For each bio:
        - compute offset = bio->bi_iter.bi_sector * sector_size
        - iterate bio_vec pages and copy to/from dev->data + offset
        - on success: bio_endio(bio, 0) (complete the bio)
3. **Cleanup (module_exit)**

- del_gendisk(dev->gd);
- put_disk(dev->gd);
- blk_cleanup_queue(dev->queue);
- unregister_blkdev(major, "myblock");

IV Block Driver Flow and Operations

This section explains how a block driver is initialized, how it handles I/O requests, and how the kernel interacts with the driver through operations and request-processing methods.

4.1 Initialization Flow (How a Block Driver Starts)

A block device driver must register itself in the kernel, allocate its disk structure, and expose a device node (e.g., /dev/myblock) to user space. The initialization process sets up the major structures used by the block layer.

Initialization Flow (Step-by-step)

1. **Register Major Number**
   - The driver calls:

     ```
     major = register_blkdev(0, "myblock");
     ```

   - The kernel assigns a dynamic major number (or use a fixed one).
   - This major number is what user-space uses to identify the driver.
2. **Set Up Request Queue**
   - Driver allocates a struct request_queue which manages all I/O requests for the disk.
   - Two methods:
     - blk_init_queue(request_fn, lock) → old, request-based
     - blk_alloc_queue + blk_queue_make_request() → modern, bio-based
   - The queue handles:
     - request merging
     - scheduling

    - dispatch to the driver
- **Allocate gendisk Structure**
- Driver allocates a block device descriptor:

```
struct gendisk *gd = alloc_disk(minors);
```

Set:

- major, first_minor
- disk_name
- capacity (in sectors)
- fops (block device operations)
- queue (the request queue)

2. **Initialize Device Storage Backend**
    - For RAM-based drivers: allocate memory (vmalloc(), kmalloc()).
    - For real hardware: initialize controller registers, DMA buffers, etc.
3. **Expose Device to System**
    - Driver calls add_disk(gd);
    - udev or static device creation exposes the node as /dev/myblock.

System Perspective

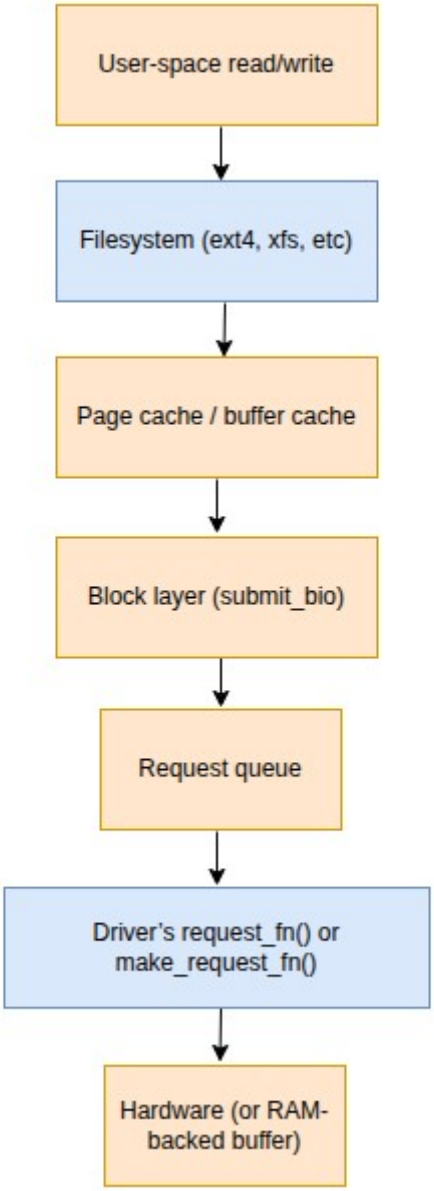
After initialization:

- The device is fully visible in /proc/partitions
- The block layer can route bios to the driver
- Filesystems and user programs can access the device normally

4.2 I/O Request Flow (How Data Moves Through the Kernel)

This describes how a user's read/write request ends up inside the block driver.

Flow Summary



4.3 Block Driver Operations

These operations control how user-space interacts with the device node. They are stored inside struct block_device_operations.

4.3.1 open() and release() — When the Device is Opened/Closed

These callbacks are executed when a user or filesystem opens/closes /dev/myblock.

open()

- Executed on:
    - mount /dev/myblock /mnt
    - fd = open("/dev/myblock")
- Typical responsibilities:
    - Check device availability
    - Increase open counters
    - Lock device if needed
    - Validate media presence for removable storage

release()

- Executed when the last handle to the block device is closed.
- Responsibilities:
    - Release power states
    - Decrement use counters
    - Mark device ready for removal

4.3.2 ioctl() Operations

ioctl() provides additional control operations beyond read/write.

Common operations:

- Get geometry (HDIO_GETGEO)
- Flush buffers
- Secure erase
- Query hardware information

Most simple RAM-disk or virtual drivers either:

- implement minimal ioctl, or
- return -ENOTTY meaning "no special I/O controls supported".

4.3.3 Media Change Detection — (For Removable Storage)

Used especially for:

- USB drives

- SD cards
- Optical drives

Kernel repeatedly checks if media is present/removed.

Driver may implement:

- check_media_change() — returns nonzero when media changed
- revalidate_disk() — rebuild partition table on change

For virtual devices (RAM disk), this is usually omitted.

4.3.4 Revalidation — Refreshing Device Geometry

After media changes or disk resizing, revalidation is used to refresh properties.

Driver implements:

- revalidate_disk(gendisk *)

Uses:

- Update capacity using set_capacity()
- Re-read partitions
- Notify VFS of changes

4.4 Request Processing Methods

(Block drivers can handle I/O in three different ways depending on kernel version and driver design.)

4.4.1 Request-based Approach

Traditional method used by older kernels.

Flow:

1. Kernel groups multiple bios into a struct request.
2. Driver implements:

```
void my_request_fn(struct request_queue *q);
```

3. Driver fetches each request:

```
struct request *req = blk_fetch_request(q);
```

1. Driver processes:
    - sequential requests
    - using rq_for_each_segment() macros
2. Driver completes request:

   __blk_end_request_all(req, 0);

Pros:

- Useful for devices that rely on sequential operations
- I/O scheduler can reorder requests

Cons:

- Extra overhead converting bios → request
- Being phased out in favor of bio-based I/O

4.4.2 Make Request Function

A newer path that bypasses the old request layer.

Driver sets:

blk_queue_make_request(queue, my_make_request_fn);

Driver receives:

my_make_request_fn(struct request_queue *q, struct bio *bio)

Driver responsibilities:

- Parse bio
- Perform read/write on device
- End I/O with bio_endio(bio)

Pros:

- Simple
- Efficient
- Less kernel overhead

Cons:

- No complex request merging/scheduling (unless manually done)

4.4.3 Bio-based Direct Approach (Modern Recommended Method)

Many modern drivers **directly receive bios** without needing request-based queues.

Flow:

- Block layer passes struct bio * directly to driver
- Driver iterates over pages inside bio->bi_io_vec
- Data copied to/from device buffer or DMA
- Driver completes using:

  bio_endio(bio);

Why it is preferred:

- Highest performance
- Zero-copy design
- Simplest implementation
- Best for virtual storage, RAM disks, and modern hardware

V Adding Block Driver to Yocto Project

The Yocto Project is a build system used for creating custom Linux distributions for embedded devices.
Understanding its architecture helps you add your custom kernel module or block driver.

5.1.1 BitBake Build System

BitBake is the *core build engine* of Yocto.

Key roles:

- Processes recipes (.bb) and configuration (.conf)
- Fetches source code
- Applies patches
- Builds kernel, drivers, user-space packages
- Generates final root filesystem + images

Important BitBake concepts:

- **Tasks** (e.g., do_patch, do_compile, do_install)
- **Dependencies** between recipes
- **Metadata** controlling the build

Think of BitBake as a "compiler" for an entire embedded OS.

5.1.2 Layer Structure

Yocto source is organized into **layers**.
A layer contains *recipes*, *classes*, *config fragments*, and *machine definitions*.

Typical layer layout:

```
meta-myblockdriver/
├── conf/
│   └── layer.conf
├── recipes-kernel/
│   └── myblock/
│       ├── myblock.bb
│       ├── files/
│       │   └── myblock.c
│       └── linux/
│           └── myblock.cfg
```

Why use layers?

- Modularity
- Board-specific customization
- Adding new kernel drivers
- Avoiding modification of core Yocto layers

5.1.3 Recipe Files

A recipe (.bb) describes **how to build something**.

For a driver:

- Download/locate source code
- Compile as module

- Install into rootfs and kernel modules directory

Most kernel module recipes inherit:

```
inherit module
```

This gives:

- Kernel headers
- Module build support
- Automatic installation into /lib/modules/<kernel-version>/

5.2 Creating a Custom Layer

To add your block driver into Yocto, you must create your own layer.

Command (real Yocto):

```
bitbake-layers create-layer meta-myblock
```

Inside the layer:

- Add layer.conf
- Create a folder for recipes
- Add kernel config fragments if required

Layer registration:

```
bitbake-layers add-layer meta-myblock
```

Your layer now becomes part of Yocto metadata.

5.3 Writing Recipe for Block Driver

A kernel module recipe looks like:

```
DESCRIPTION = "Custom Block Device Driver"
LICENSE = "GPL-2.0-only"
SRC_URI = "file://myblock.c"


inherit module


S = "${WORKDIR}"


# Install module into /lib/modules/
do_install() {
    install -d ${D}${nonarch_base_libdir}/modules/${KERNEL_VERSION}/extra
    install -m 0644 myblock.ko ${D}${nonarch_base_libdir}/modules/${KERNEL_VERSION}/extra
}
```

Key Points:

- SRC_URI points to your driver source files.
- inherit module handles kernel module building rules.
- do_install installs .ko into correct module directory.

After build:

/lib/modules/<version>/extra/myblock.ko

5.4 Kernel Configuration Integration

To ensure that the block driver gets integrated into the kernel build, you must modify kernel configuration.

This is done with:

- .bbappend
- Config fragments
- Local configuration

5.4.1 Creating .bbappend Files

A .bbappend modifies an existing recipe.
Example:

```
recipes-kernel/linux/linux-yocto_%.bbappend
```

Inside:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://myblock.cfg"
```

This appends your configuration fragment to the kernel config.

5.4.2 Config Fragment Files

A .cfg file enables your driver inside the kernel:

Example (myblock.cfg):

```
CONFIG_BLK_DEV_MYBLOCK=m
CONFIG_BLK_DEV_LOOP=y
CONFIG_BLOCK=y
```

This tells the kernel to:

- Build your block driver as a module
- Enable core block subsystem components

5.4.3 Driver Selection in Local Configuration

In conf/local.conf add:

```
MACHINE_FEATURES += "myblock"
IMAGE_INSTALL:append = " myblock"
```

Or enable module auto-loading:

```
KERNEL_MODULE_AUTOLOAD += "myblock"
```

Now the block driver loads at boot.

5.5 Building and Testing

Building the driver

To build kernel + driver:

bitbake core-image-minimal

or

bitbake myblock

After Build

The driver appears in:

build/tmp/deploy/images/<machine>/
build/tmp/work/.../myblock.ko

Testing on Target Device

1. Copy to target rootfs or load via modules directory.
2. Load module:

modprobe myblock

or

insmod myblock.ko

3. Verify registration:

lsblk
cat /proc/partitions
dmesg | grep myblock

4. Create filesystem:

mkfs.ext4 /dev/myblock

5. Mount:

mount /dev/myblock /mnt

If everything is correct → block driver is operational.

VI Integrating Block Driver in Android BSP

6.1 Android Kernel Structure

Android uses a Linux-based kernel, but with additional patches, security modules, power optimizations, and vendor-specific changes.

6.1.1 Android Common Kernel

The **Android Common Kernel (ACK)** is Google's baseline kernel that contains:

- Upstream Linux kernel features.
- Android-specific patches (wakelocks, binder, low memory killer, etc.).
- Generic drivers and HAL interfaces.

Key Points:

- ACK acts as a **reference kernel** that all OEMs must align with.
- It ensures consistency across different devices.
- Supports the **Generic Kernel Image (GKI)** concept.

6.1.2 Device-Specific Kernels

OEMs take ACK and add:

- Board-specific drivers (display, camera, touch, sensors).
- Vendor SoC drivers (Qualcomm, Mediatek, Samsung, etc.).
- Memory map and device-tree adjustments.

Characteristics:

- Highly customized.
- Usually contains proprietary modules.
- Maintained by vendors, not Google.

6.1.3 Kernel Modules in Android

Android supports:

1. **Built-in kernel drivers**
   → Compiled directly into the kernel .img.
2. **Loadable Kernel Modules (LKMs)**
   → .ko files loaded using:

   insmod <file.ko>
   rmmod <file.ko>

3. **GKI Modules**

- Android 12+ uses **modular drivers** that can be updated without rebuilding the full kernel.
- Vendor/OEM drivers must be compatible with **VTS (Vendor Test Suite)**.

6.2 Adding Driver to Android Kernel

Steps to add a custom block driver:

1. Add Driver Source

Place driver code into one of these:

- kernel/drivers/block/ (built-in)
- vendor/<oem>/drivers/ (for OEM drivers)
- device/<vendor>/<board>/kernel/drivers/

2. Modify Kconfig

In your driver folder:

```
config MY_BLOCK_DRIVER
   tristate "My Custom Block Driver"
   depends on BLOCK
```

3. Modify Makefile

```
obj-$(CONFIG_MY_BLOCK_DRIVER) += my_block_driver.o
```

4. Enable the driver

Using menuconfig:

Device Drivers → Block Devices → [*] My Custom Block Driver

5. Compile

Running:

```
make bootimage
```

6.3 Device Tree Configuration

Block devices usually need:

- Base address
- IRQ numbers

- DMA channels
- Memory region mapping

Example DTS Node:

```
my_block_dev: myblock@12340000 {
   compatible = "myvendor,myblock";
   reg = <0x12340000 0x1000>;
   interrupts = <0 55 4>;
   dma-coherent;
   status = "okay";
};
```

Driver Reads These Using:

- platform_get_resource()
- devm_ioremap_resource()
- of_irq_get()

6.4 Board Configuration Files

6.4.1 BoardConfig.mk Changes

In:
device/<vendor>/<board>/BoardConfig.mk

Add:

```
BOARD_KERNEL_CMDLINE += myblock.enable=1
BOARD_VENDOR_KERNEL_MODULES += my_block_driver.ko
```

If built-in:

```
TARGET_KERNEL_CONFIG += myblock_defconfig
```

6.4.2 device.mk Integration

device/<vendor>/<board>/device.mk

Add module copy rules:

```
PRODUCT_COPY_FILES += \
   device/<vendor>/<board>/modules/my_block_driver.ko:$
(TARGET_COPY_OUT_VENDOR)/lib/modules/my_block_driver.ko
```

6.4.3 SELinux Policies for Block Device

Android is strict; your driver must have correct SELinux rules.

Add file context:

/vendor/lib/modules/my_block_driver.ko u:object_r:vendor_file:s0

Add policy (.te file):

allow hal_block_device self:block_device rw_file_perms;
allow hal_block_device my_block_driver_device:chr_file rw_file_perms;

If permission blocked → logs appear in dmesg or logcat.

6.5 Building and Flashing

Build Commands

For AOSP:

source build/envsetup.sh
lunch <target>
make bootimage -j$(nproc)

For GKI:

make modules

Flash Kernel

On fastboot-supported devices:

fastboot flash boot out/target/product/<device>/boot.img
fastboot reboot

Flash Vendor Modules (if .ko)

adb root
adb remount
adb push my_block_driver.ko /vendor/lib/modules/
adb reboot

VII Driver Registration and Probe Mechanism

7.1 Registration Methods

Drivers must **register themselves** with the Linux kernel so the kernel can detect them, match them to hardware, and call their probe/remove functions.

Linux offers **two major registration models**:

7.1.1 Platform Driver Registration

Platform drivers are used for **on-SoC devices** (non-discoverable hardware) such as:

- eMMC, NAND
- GPIO, I2C-based devices
- Timers, UART, SPI controllers
- Block controllers integrated on SoC

How It Works

- Hardware information is passed through **Device Tree (DT)**.
- The kernel matches .compatible strings in DT with the driver's table.
- When matched → kernel calls **probe()**.

Driver Code Example

```
static const struct of_device_id myblock_of_match[] = {
   { .compatible = "myvendor,myblockdev", },
   { }
};
MODULE_DEVICE_TABLE(of, myblock_of_match);

static struct platform_driver myblock_driver = {
   .probe  = myblock_probe,
   .remove = myblock_remove,
   .driver = {
      .name = "myblock",
      .of_match_table = myblock_of_match,
   },
};

module_platform_driver(myblock_driver);
```

Why Used?

- No dynamic bus discovery.
- Simple, low power, perfect for embedded SoCs.

7.1.2 Traditional Char/Block Registration

Older Linux drivers (and some modern ones) use **manual device number registration**.

Two APIs:

1. **Character driver registration**

   register_chrdev_region()
   cdev_init()
   cdev_add()

2. **Block driver registration**

   register_blkdev()
   add_disk()

When Used?

- Legacy drivers
- Custom device models
- Simple kernel modules that don’t rely on Device Tree

7.2 Device Tree Matching

Linux matches drivers to hardware using a **Compatible String**.

Device Tree Example

```
myblock0: myblock@12340000 {
   compatible = "myvendor,myblockdev";
   reg = <0x12340000 0x1000>;
   interrupts = <0 45 4>;
};
```

Driver Must Provide Matching Table

```
static const struct of_device_id myblock_of_match[] = {
   { .compatible = "myvendor,myblockdev" },
   {}
};
```

Matching Flow:

1. Kernel loads device tree.
2. Kernel loads driver.
3. Kernel compares .compatible strings.
4. If match → **probe() is called**.

This mechanism removes the need for hard-coded addresses.

7.3 Probe Function Implementation

The **probe()** function is the heart of your driver.

Purpose of probe()

- Allocate resources.
- Map registers.
- Request IRQ.
- Configure DMA.
- Create request queues (for block drivers).
- Register gendisk.
- Make device ready for use.

Example (Block Device)

```
static int myblock_probe(struct platform_device *pdev)
{
   struct resource *res;

   // Get and map registers
   res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
   base = devm_ioremap_resource(&pdev->dev, res);

   // Get IRQ
   irq = platform_get_irq(pdev, 0);
   devm_request_irq(&pdev->dev, irq, myblock_irq_handler, 0, "myblock",
NULL);

   // Setup queue
   queue = blk_alloc_queue(GFP_KERNEL);
   blk_queue_make_request(queue, myblock_make_request);
```

```
    // Create gendisk
    disk = alloc_disk(1);
    disk->queue = queue;
    disk->private_data = mydev;
    strcpy(disk->disk_name, "myblk0");
    add_disk(disk);

    dev_info(&pdev->dev, "My Block Device Probed\n");
    return 0;
}
```

Probe = Device Initialization

If probe fails → kernel unloads driver or marks device unusable.

7.4 Remove Function

The **remove()** function undoes everything created in probe.

Purpose of remove()

- Free memory.
- Remove disk.
- Destroy request queue.
- Release interrupts.
- Unmap I/O memory.

Example:

```
static int myblock_remove(struct platform_device *pdev)
{
    del_gendisk(disk);
    put_disk(disk);

    blk_cleanup_queue(queue);

    dev_info(&pdev->dev, "My Block Device Removed\n");
    return 0;
}
```

When remove() is called?

- Driver is unloaded.
- Device is hot-removed.

- Kernel shutdown (cleanup stage).

VIII Steps to Enable Block Driver by Default

Effective integration of a block device driver requires kernel configuration, automatic loading mechanisms, Device Tree enablement, boot-time inclusion, and proper userspace coordination.
Each subsection below explains **what must be done and why**.

8.1 Kernel Configuration Settings

Every kernel driver must be enabled through kernel configuration (make menuconfig, .config, or Yocto fragments).

Two Options Exist

1. **Built-in Driver (=y)**
    - Driver is compiled into the kernel image.
    - Loaded at boot automatically.
    - Required for early-boot storage devices (e.g., rootfs on eMMC).
2. **Loadable Kernel Module (=m)**
    - Created as .ko file.
    - Loaded on-demand via modprobe or udev.
    - Preferred for debugging / development.

Example Kconfig Entry

CONFIG_MYBLOCKDEV=m
CONFIG_BLK_DEV=y
CONFIG_BLOCK=y

Where Kconfig Matters

- Yocto kernel fragments
- Android kernel build
- Custom embedded Linux builds
- Device bring-up on new hardware

8.2 Module Autoload Configuration

If the block driver is compiled as a module (.ko), it must load automatically.

Two methods:

1. Using /etc/modules-load.d/

Create file:

/etc/modules-load.d/myblock.conf

Content:

myblock_driver

2. Using modprobe alias

Driver:

MODULE_ALIAS("platform:myblock");

Udev auto-matches and loads module.

Autoload ensures that once the device is probed (or detected), the kernel loads the module with no user actions.

8.3 Device Tree Default Enablement

Device Tree must include the block driver node to make the kernel attach the driver.

Example DT Node

```
myblock0: myblock@12340000 {
   compatible = "myvendor,myblockdev";
   reg = <0x12340000 0x1000>;
   interrupts = <0 45 4>;
   status = "okay";
};
```

Key Points

- compatible must match driver's table.
- status = "okay" ensures the device is **enabled** by default.
- Missing DT node = **driver never probes**.

8.4 Initramfs Inclusion

Initramfs (or initrd) is the **early userspace** required before root filesystem is mounted.

Block device drivers may be needed **before rootfs mount**, especially if:

- RootFS is located on eMMC
- RootFS is on SD card
- Custom block devices store system metadata

Two requirements:

8.4.1 Adding to INITRD modules

For module-based drivers:

Add driver in:

/etc/initramfs-tools/modules

or Yocto:

INITRAMFS_MODULES += "myblock_driver"

This ensures .ko is present in RAM during early boot.

8.4.2 Early Userspace Loading

Initramfs scripts load the module before mounting rootfs.

Example script snippet:

modprobe myblock_driver

Used when:

- Driver initializes hardware required for rootfs
- Device must appear as /dev/myblk0 before system continues booting

8.5 Systemd Service (for User-Space Drivers)

Some drivers run **partially in userspace**, such as:

- FUSE-based drivers
- Drivers using /dev/uioX
- User-space block emulators

- HAL or daemon-managed hardware

Example systemd service

/etc/systemd/system/myblock.service

[Unit]
Description=MyBlock User-Space Driver
After=udev.service

[Service]
ExecStart=/usr/bin/myblock_daemon
Restart=always

[Install]
WantedBy=multi-user.target

Enable:

systemctl enable myblock.service

This ensures driver daemon starts on boot.

8.6 Udev Rules for Automatic Loading

Udev allows auto device creation and module loading based on hardware properties.

Example udev rule

/etc/udev/rules.d/99-myblock.rules

KERNEL=="myblk*", MODE="0666", GROUP="disk"

To auto-load module based on device

ACTION=="add", SUBSYSTEM=="platform", ATTR{compatible} =="myvendor,myblockdev", RUN+="/sbin/modprobe myblock_driver"

What Udev Does:

- Creates device files under /dev
- Applies permissions
- Loads modules if needed
- Runs scripts or daemons

IX Complete Example: RAM Disk Block Driver

A **RAM Disk Block Driver** is the simplest way to understand Linux block driver concepts because it behaves like a real block device but uses **system RAM instead of physical hardware**.
This example provides:

- Full working source code
- Detailed explanation
- Build instructions
- Module parameters to customize size and name

9.1 Source Code Implementation

ramdisk_block.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>
#include <linux/vmalloc.h>
#include <linux/errno.h>

#define RAMDISK_SECTOR_SIZE 512

static int ramdisk_major = 0;
static int rd_sectors = 1024 * 100;   // 100K sectors = ~50MB
module_param(rd_sectors, int, 0444);

static char *rd_buffer;
static struct request_queue *rd_queue;
static struct gendisk *rd_disk;

// ---- I/O Request Handler ----
static void rd_request(struct request_queue *q)
{
   struct request *req;

   while ((req = blk_fetch_request(q)) != NULL) {
      sector_t sector = blk_rq_pos(req);
      unsigned int len = blk_rq_cur_bytes(req);

      char *buffer_ptr = rd_buffer + (sector * RAMDISK_SECTOR_SIZE);
```

```
        if (rq_data_dir(req) == READ) {
            memcpy(req->buffer, buffer_ptr, len);
        } else {
            memcpy(buffer_ptr, req->buffer, len);
        }

        blk_end_request(req, 0, len);
    }
}

// ---- Driver Initialization ----
static int __init ramdisk_init(void)
{
    rd_buffer = vmalloc(rd_sectors * RAMDISK_SECTOR_SIZE);
    if (!rd_buffer)
        return -ENOMEM;

    ramdisk_major = register_blkdev(0, "ramdisk");
    if (ramdisk_major < 0)
        return ramdisk_major;

    rd_queue = blk_init_queue(rd_request, NULL);

    rd_disk = alloc_disk(1);
    rd_disk->major = ramdisk_major;
    rd_disk->first_minor = 0;
    rd_disk->fops = NULL;  // no ioctl/open needed for basic RAM disk
    rd_disk->queue = rd_queue;
    snprintf(rd_disk->disk_name, 32, "ramdisk0");
    set_capacity(rd_disk, rd_sectors);

    add_disk(rd_disk);

    printk(KERN_INFO "ramdisk: initialized with size %d sectors\n",
rd_sectors);
    return 0;
}

// ---- Driver Exit ----
static void __exit ramdisk_exit(void)
{
    del_gendisk(rd_disk);
    put_disk(rd_disk);
```

```
    unregister_blkdev(ramdisk_major, "ramdisk");
    blk_cleanup_queue(rd_queue);

    vfree(rd_buffer);

    printk(KERN_INFO "ramdisk: removed.\n");
}

module_init(ramdisk_init);
module_exit(ramdisk_exit);

MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Simple RAM Disk Block Device Driver");
MODULE_AUTHOR("Example");
```

9.2 Code Explanation (Deep Notes)

1. RAM Buffer Allocation

rd_buffer = vmalloc(rd_sectors * RAMDISK_SECTOR_SIZE);

- Allocates a block of RAM
- Behaves like a disk backend
- vmalloc ensures contiguous *virtual* memory (physical may be paged)

2. Registering a Block Device Major Number

ramdisk_major = register_blkdev(0, "ramdisk");

- Kernel assigns a unique major number
- Used to identify the block device in /dev

3. Creating the Request Queue

rd_queue = blk_init_queue(rd_request, NULL);

- A queue where all read/write operations are placed
- rd_request() handles those I/O operations

4. I/O Request Handling

blk_fetch_request()

Retrieves requests — both READ and WRITE.

Read Operation:

memcpy(req->buffer, buffer_ptr, len);

Write Operation:

memcpy(buffer_ptr, req->buffer, len);

Notes:

- No caching, no scheduling — simple direct copy
- This makes the driver ideal for learning

5. gendisk Creation

rd_disk = alloc_disk(1);

- Allocates disk structure
- Minor count = 1

set_capacity(rd_disk, rd_sectors);

Defines the disk size.

6. Adding Disk to Kernel

add_disk(rd_disk);

Kernel creates:

- /dev/ramdisk0
- Adds it to /sys/block/ramdisk0

7. Cleanup

Driver removes disk, queue, and frees memory.

9.3 Compilation Instructions

1. Create a Makefile

```
obj-m += ramdisk_block.o

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

2. Compile Driver

make

Creates:

ramdisk_block.ko

3. Insert Module

sudo insmod ramdisk_block.ko rd_sectors=200000

4. Check Device

lsblk
dmesg | grep ramdisk

5. Test With Filesystem

sudo mkfs.ext4 /dev/ramdisk0
sudo mount /dev/ramdisk0 /mnt

Now it acts like a real block device.

9.4 Module Parameters

Module parameter in code:

static int rd_sectors = 1024 * 100;
module_param(rd_sectors, int, 0444);

This parameter allows:

1. Custom Disk Size

insmod ramdisk_block.ko rd_sectors=500000

Creates bigger RAM disk.

2. Runtime Tunability

- Adjust speed
- Set different buffer sizes
- Testing with various workloads
- Useful for benchmarking block layers