ChatGPT

Block Device Drivers in Linux 6.x: A Comprehensive Tutorial

Introduction to Block Devices and Their Importance

Block devices are kernel interfaces for randomly accessible, fixed-size data storage devices (primarily disks). Unlike character devices (which stream data sequentially), block devices allow reading/writing in **blocks** of data identified by sector number [1] [2]. A typical block size is 4096 bytes (one page), though the kernel defines a logical sector size of 512 bytes for addressing. (If hardware sectors differ, the kernel adapts by scaling sector numbers and I/O sizes [3].) Block devices are crucial to system performance because they underlie filesystems and virtual memory swap. The OS relies on block drivers to efficiently shuttle data between RAM and secondary storage [4]. In contrast to many char devices, block devices often must maximize throughput and handle **I/O scheduling** to reorder and merge requests for optimal access patterns [5].

Block vs. Character Devices: The kernel treats block devices very differently from char devices. Applications can **mount filesystems** on block devices, and I/O to them goes through the buffer cache/page cache. When a user reads or writes a block device (or a file on a filesystem on it), the kernel doesn't call a `.read()` or `.write()` method directly as with char drivers. Instead, it uses the **block I/O layer** to buffer data and assemble I/O requests. These requests are handled asynchronously by the block driver, often out of order, to improve performance. Char devices, by contrast, present a direct byte-stream interface with relatively straightforward, immediate read/write operations. Block devices must deal with request queues, caching, and random-access patterns, whereas char devices handle data in a linear fashion with minimal caching. In short, **only block devices can support filesystems** (the kernel's VFS uses block drivers to read/write disk blocks), while char devices are accessed via system calls without such scheduling and caching [6]. This difference mandates a distinct driver framework for block devices.

Performance Considerations: Because block drivers sit on the critical path of storage and memory management, they are highly optimized. The block layer will queue and schedule I/O operations globally to minimize disk seeks and combine adjacent requests. As a driver writer, you must integrate with these mechanisms to get good throughput. The Linux 2.6+ block API significantly improved on earlier kernels by simplifying how drivers hook into the I/O scheduler [7]. Modern Linux kernels (5.x and 6.x) further evolved this with **multi-queue block I/O (blk-mq)** for better scaling on multi-core systems and fast devices.

Registering a Block Driver: Major Numbers and `gendisk`

Like char drivers, block drivers must register a device name and major number. This is done with `register_blkdev()` (declared in `<linux/fs.h>`), which is analogous to `register_chrdev()` [8]. For example:

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

This call requests a dynamic major if `0` is passed, or reserves a specific major. It also registers the driver name for `/proc/devices`. In modern kernels, calling `register_blkdev` is **optional** – it mainly allocates a major and creates the proc entry [9]. Many in-tree drivers simply use a static major or dynamic allocation and rely on devtmpfs/udev to create device nodes. However, for simplicity, this tutorial will use `register_blkdev` to obtain a major number for our device. Always unregister in cleanup with `unregister_blkdev(major, "name")` to release the major [10].

The `gendisk` Structure: Registering as a block driver is only the first step. The kernel represents each actual disk (or disk-like device) with a `struct gendisk` (defined in `<linux/genhd.h>`). A `gendisk` holds information about the device such as its name, size (capacity), supported partition layout, and the operations (methods) to handle it [11] [12]. Unlike char devices that are accessed via a simple major/minor number and `cdev`, a block driver typically creates one or more `gendisk` structures – one for each physical or logical disk it manages.

To set up a `gendisk`:

1. **Allocate and initialize the gendisk** – In older kernels you'd use `alloc_disk(minors)` to allocate a gendisk and its partition array. In Linux 6.x, this is replaced by newer APIs. If using the modern multi-queue API, you can use `blk_alloc_disk(node)` or `blk_mq_alloc_disk(tag_set, queuedata)`. For example, for a simple driver we might do:

```
dev->gd = blk_alloc_disk(NUMA_NO_NODE);
```

This allocates a gendisk (and an associated request queue) on the given NUMA node [13] [14]. Alternatively, `blk_mq_alloc_disk(&tag_set, dev)` can be used to allocate a gendisk *and* set up its request queue using a provided tag set (more on tag sets in the I/O scheduling section). If these functions return an error (NULL or `IS_ERR`), allocation failed (e.g. out of memory). Always check and handle that.

1. **Assign a major and minors** – Set `gd->major` to the driver's major number (from `register_blkdev`) and `gd->first_minor` to the first minor number for this disk (often 0 if it's the first disk for that driver) [15]. Also specify the number of minor numbers (partitions) this disk supports. In older code this was passed to `alloc_disk()`. With `blk_alloc_disk`, you can set `dev->gd->minors` to e.g. 16 or use `device_add_disk` which handles partition registration. For instance, `minors=16` means the disk can have up to 15 partitions (minor 1–15) in addition to the whole-disk device (minor 0) [16]. If you only want a whole-device node with no partitions, set minors to 1 [16]. (Using more minors than needed is fine; unused minor numbers won't have device nodes until partitions are created.)

2. **Assign the block device operations** – Attach your driver's operation table (a `struct block_device_operations`) to the gendisk by setting `gd->fops`. This structure is like the char driver's `file_operations` but tailored to block device needs (see next section for details) [17]. Also

set `gd->private_data` to point to your device's internal structure (so your operations can access driver-specific data) [17] .

3. **Set the device capacity** – The kernel needs to know the size of the block device. This is expressed in sectors of 512 bytes. Calculate the total number of 512-byte sectors your device holds and use `set_capacity(gd, n_sectors)`. For example, if you allocated a 1 MiB RAM buffer for a ramdisk, `n_sectors = 1*1024*1024 / 512 = 2048` sectors. The `set_capacity` call updates the kernel's view of the disk size [18] . (If your hardware uses a different logical block size, adjust the sector count accordingly. The kernel will ensure I/O requests align to the logical block size you specify.)

4. **Attach a request queue** – Every gendisk must be associated with a request queue (`gd->queue`) which manages I/O scheduling for that disk. If you used `blk_alloc_disk`, a request queue was created for you. If not, you need to allocate one (for older interfaces, `blk_init_queue()` was used; with multi-queue, you use `blk_mq_init_queue()` or similar). In either case, you assign `gd->queue` to the queue pointer. Also, if not done by the alloc call, set the queue's `queuedata` to point to your device structure (this is analogous to `private_data`, used inside your request handler to get device context) [19] [20] . We will discuss how to set up the request queue in detail later.

5. **Name the disk** – Provide a name for the disk by setting `gd->disk_name`. This name will appear in sysfs and in user-visible device node creation. It's common to use a base name plus an index (e.g. `"mydisk0"`). The name should be unique system-wide. For example: `snprintf(dev->gd->disk_name, 32, "myblock%d", dev_index);`.

6. **Finally, register the disk** – Call `add_disk(dev->gd)`. This announces the new disk to the kernel. After `add_disk()`, the device node (e.g. `/dev/myblock0`) will be created (if using devtmpfs/udev), and the disk is live – the kernel can start calling your driver's operations to handle I/O [21] . **Important:** once you call `add_disk`, your driver must be ready to handle requests (the kernel may issue reads/writes immediately). If your hardware isn't ready yet, you might want to set a flag or block requests until it is, but in our simple example, readiness is immediate.

In summary, registering a block device involves creating and populating a `gendisk` structure and hooking it into the system. The code might look like this (for a simplified RAM disk driver):

```
/* Pseudocode for block device initialization (error checking omitted for
brevity) */
dev = kzalloc(sizeof(*dev), GFP_KERNEL);
dev->size = 2048; // number of sectors (512-byte blocks)
dev->data = kzalloc(dev->size * 512, GFP_KERNEL); // allocate backing store

dev->gd = blk_alloc_disk(NUMA_NO_NODE);
dev->gd->major        = my_major;
dev->gd->first_minor  = 0;
dev->gd->minors       = 16;                        // support up to 15
partitions
dev->gd->fops         = &myblock_ops;
```

```
dev->gd->private_data = dev;
strscpy(dev->gd->disk_name, "myblock0", DISK_NAME_LEN);
set_capacity(dev->gd, dev->size);                  // total sectors in 512-byte
units

/* If not using blk_alloc_disk, here we would need to create a request queue and
set dev->gd->queue */
dev->gd->queue        = dev->queue;                // assign prepared request
queue
add_disk(dev->gd);
```

After this, the block device `/dev/myblock0` would be available for use, with a major number and the next available minor number (or a fixed one if configured). The kernel will also create additional device nodes for partitions if any are present (e.g. `/dev/myblock0p1`, etc.), up to the `minors` count. (The partition nodes may appear after a partition table is written and `ioctl(BLKRRPART)` is invoked to re-read the partition table.)

Cleaning Up on Driver Removal

When unloading the driver (module exit), you must unregister and clean up everything in reverse order. This includes:

- Remove the disk from the system: call `del_gendisk(dev->gd)`. This stops new I/O and removes the disk and any partitions from sysfs (which in turn causes device nodes to disappear).
- Delete the request queue and gendisk: in Linux 6.x, use `blk_cleanup_disk(dev->gd)`. This will mark the queue dead, drain any outstanding requests, free the request queue, and put the gendisk (freeing its memory) [22]. In older code, you would individually call `blk_cleanup_queue(queue)` and `put_disk(gd)`, but `blk_cleanup_disk` is a convenient combined API [22]. (Do **not** call `put_disk` after `blk_cleanup_disk` – it already does that.)
- Free any memory used for the device's data (if it's a RAM disk, e.g. free `dev->data`).
- Unregister the major number if you called `register_blkdev`.

For example:

```
del_gendisk(dev->gd);
blk_cleanup_disk(dev->gd);    // this replaces blk_cleanup_queue + put_disk
unregister_blkdev(my_major, "myblock");
kfree(dev->data);
kfree(dev);
```

At this point, the system no longer knows about your disk. It's crucial to call `del_gendisk` before cleaning up the disk structure to ensure the kernel isn't still accessing it. The combination of `del_gendisk` and `blk_cleanup_disk` ensures all pending I/O is done and the resources are freed [23] [24].

The Block Device Operations Structure

Every block driver must implement a set of operations, provided via `struct block_device_operations` (often abbreviated as `block_device_ops` or `bdev_ops`). This is analogous to char drivers' `file_operations`, but it's attached to the `gendisk` rather than to a device node directly [25] [26]. The block device operations define how the kernel should interact with your device for **non-data** operations (open, release, ioctl, etc.). They do **not** include read/write methods for data transfer – data I/O is handled via the request queue mechanism, described later.

Important fields in `struct block_device_operations` include [27] [28]:

- `open(struct block_device *bdev, fmode_t mode)` – Called when the device is opened (e.g. `mount` or `fdisk` opening `/dev/yourdisk`). Similar to a char device open, you can use it to initialize or lock the device. The `struct block_device` (`bdev`) contains a pointer to the `struct gendisk` (accessible via `bdev->bd_disk`), and you can get your `private_data` via `bdev->bd_disk->private_data`. The `mode` indicates whether it's read-only or read-write open. Many simple drivers don't need to do much in open; they might just increment a usage counter or check media presence. Our ramdisk example might not need special handling, so a stub that returns 0 is fine.

- `release(struct gendisk *disk, fmode_t mode)` – Called on the last close of the block device. You can clean up any per-open state here. In drivers for removable media, `release` might unlock eject mechanisms, etc. The `disk` corresponds to the gendisk being closed. Often, open/release are used to manage a device lock or simulate media removal. In many cases (like a memory disk), these can be no-ops.

- `ioctl(struct block_device *bdev, fmode_t mode, unsigned int cmd, unsigned long arg)` (and the similar `compat_ioctl` for 32-bit compatibility) – Handles ioctl calls from user space on the block device (e.g. issued by `hdparm` or partitioning tools). The block layer intercepts many standard ioctls (like those to get sector count or geometry) before they reach your driver [29]. For example, `BLKGETSIZE` and `BLKGETSIZE64` (to get device size) are handled by the kernel using the gendisk capacity, and `BLKRRPART` (re-read partition table) will cause the kernel to call your driver's media change check and revalidate methods. So your ioctl handler usually only needs to handle device-specific commands or those not covered by generic code. If you have none, you can omit ioctl or use the provided `noop_ioctl` (which returns `-ENOTTY` for unsupported commands). In our simple driver, we might implement support for `HDIO_GETGEO` (get geometry) for compatibility with older tools – typically by fabricating a geometry.

- `media_changed(struct gendisk *disk)` – Should return nonzero if the media has changed (and thus the kernel should invalidate its cache of the device, like re-reading the partition table). This is relevant for removable drives (CD-ROMs, USB card readers, etc.). For a ramdisk or other non-removable storage, you can omit this or return 0. If implemented, the block layer will call it periodically or on open to check if it should discard buffered data. The classic usage is for floppy drives, where opening the device file triggers a check if a new floppy has been inserted.

- `revalidate_disk(struct gendisk *disk)` – Invoked after `media_changed` returns true, or when the user requests a revalidate (e.g. `BLKRRPART` ioctl or on resume from sleep). This method should update the device's capacity (using `set_capacity`) and any other parameters that may change when media is changed [30] [31]. After `revalidate_disk`, the kernel will attempt to re-read the partition table. In our ramdisk, "media change" might be simulated (for example, some ramdisks toggle a flag to test the mechanism), but typically it's not needed. If you do implement it, make sure to clear the media-change flag and update geometry. The sbull example from LDD3 simply reset its internal change flag and zeroed out its data on revalidate [32].

- `getgeo(struct block_device *bdev, struct hd_geometry *geo)` – (Optional) Provides disk geometry (heads, sectors, cylinders) for old BIOS-based software that uses it (like fdisk on very old systems). Modern systems mostly ignore geometry, but some ioctl calls (HDIO_GETGEO) use it. A simple implementation can invent values that roughly approximate the size (e.g. 64 heads, 32 sectors, X cylinders). This is only needed for compatibility; you can omit it if not needed. If implemented, fill the `hd_geometry` struct with `.heads`, `.sectors`, `.cylinders`, and `.start` (the starting offset, usually 0). Our example can safely skip it or use a dummy.

- `owner` – The module owner (set to `THIS_MODULE`). This is important to prevent the module from unloading while operations are in use.

- `submit_bio(struct bio *bio)` – (Advanced/optional) This is a newer addition for "BIO-based" drivers. If you set this, the block layer will bypass the request queue and call your `submit_bio` function directly for each bio (a bio represents a low-level I/O operation on a range of sectors). This effectively puts the burden of handling I/O scheduling and merging on the driver or higher layers. Drivers like Device-mapper or network block devices sometimes use this mode. We'll touch on this in the request handling section. For most drivers, you will **not** implement `submit_bio` and will use the standard request-queue path instead (i.e. handle I/O in your request handler). If `submit_bio` is set, the block layer won't queue requests in the usual way [33] [34].

Below is a minimal example of defining `block_device_operations` for our device:

```
static int myblock_open(struct block_device *bdev, fmode_t mode) {
    /* Prevent writes if device is write-protected, etc. For ramdisk, nothing to
do. */
    return 0;
}
static void myblock_release(struct gendisk *gd, fmode_t mode) {
    /* Cleanup if necessary (ramdisk doesn't need anything here). */
}
static int myblock_ioctl(struct block_device *bdev, fmode_t mode,
                          unsigned int cmd, unsigned long arg) {
    switch(cmd) {
      case HDIO_GETGEO: {  // example of handling getgeo
          struct hd_geometry geo;
          long size = get_capacity(bdev->bd_disk); // total sectors
          geo.heads = 1;
```

```
            geo.sectors = 32;
            geo.cylinders = size / (geo.heads * geo.sectors);
            geo.start = 0;
            if (copy_to_user((void __user *)arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
        }
        default:
            return -ENOTTY;
    }
}
static const struct block_device_operations myblock_ops = {
    .owner    = THIS_MODULE,
    .open     = myblock_open,
    .release = myblock_release,
    .ioctl    = myblock_ioctl,
    /* .getgeo = myblock_getgeo, (could be provided instead of ioctl handling)
*/
    /* .media_changed = NULL (not needed for non-removable) */
    /* .revalidate_disk = NULL (not needed unless media can change) */
};
```

In this example, open/release do nothing special, and ioctl handles one case (HDIO_GETGEO) just to illustrate. Many drivers would omit ioctl entirely or only handle a few specific commands. The exact set of operations you implement depends on your hardware's needs. For instance, a physical disk driver might implement `media_changed` and `revalidate_disk` if it deals with removable media. A flash storage driver might not need those at all.

One key point: **there are no `read` or `write` methods in block_device_operations**. Regular read/write requests are not handled via the `.open/.release` and file I/O operations as with char devices [35]. Instead, they go through the block I/O scheduler and your **request queue handler**. We turn to that next.

The Block I/O Layer and Request Queues

When user-space or filesystem code issues I/O on a block device, the requests pass through the kernel's block layer before reaching your driver. The block layer's job is to collect I/O operations (reads, writes, discards, flushes, etc.), merge adjacent requests, schedule them efficiently (to minimize seek time or leverage parallelism), and then dispatch them to drivers at the right time. This process revolves around the **request queue** (`struct request_queue`), which is created for each block device (gendisk). The request queue holds pending I/O requests (`struct request`), which in turn contain one or more BIOS (`struct bio`) with the actual data buffers.

I/O Flow Overview: A simplified flow of a read operation might be: an application calls `read()` on a file -> the VFS and filesystem determine which block(s) on the disk are needed -> the filesystem (or generic block layer) creates one or more `bio` structures describing those blocks -> the `bio` is submitted to the block layer (`submit_bio()` is called) [36] [37] -> the block layer assigns the bio to the appropriate device's request queue, possibly merging it with other pending requests -> the I/O scheduler decides an optimal order -> the block driver is called to process the request -> the driver transfers data (from device or in our case, from RAM buffer) -> when done, the driver signals completion -> the block layer then ends the bio, waking up any waiting processes and marking the data ready in the page cache.

Key components in this flow: - **`struct bio`:** Represents a low-level I/O operation on a contiguous range of sectors on a device, with a list of memory segments (bio vectors) describing the buffer(s) [38] [39]. Each bio can span one or more physically contiguous memory regions (pages or parts of pages). - **`struct request`:** Higher-level structure that may contain one or more bios. The I/O scheduler merges bios into a single request if they are adjacent on disk and have the same direction (read/write) [40] [41]. A request essentially represents a command that the driver should perform: "read X sectors starting at position Y" or "write N sectors starting at position M," possibly already merging what were separate user requests. The request holds a list of bios (via `req->bio` linked list) and some metadata (like the operation type, e.g. read vs write, and flags such as FUA or flush). - **`struct request_queue`:** The queue that holds pending requests for a device. It also stores device-specific limits (e.g., maximum segments in a request, alignment requirements, whether the device can do merges, etc.) [42] [43]. The request queue is managed by the block layer: it accepts bios (`blk_queue_bio` internally), turns them into requests, sorts/merges them, and then dispatches to the driver when appropriate.

Block device driver components and relationships: *This diagram (conceptually from a 2.6 kernel) shows how a gendisk, its request queue, and requests/bios relate. The driver provides a `request_fn` (or in modern kernels, a multi-queue ops structure) to handle queued requests. The gendisk holds the `block_device_operations` (`fops`), points to the request queue, and has a partition table (array of `hd_struct`). The request queue links to request structures, which in turn link to bios with the actual data buffers.* (For Linux 6.x, the single `request_fn` is superseded by multi-queue as described below, but the core relationships remain similar.)

Modern Linux kernels use the **Multi-Queue Block I/O (blk-mq)** subsystem. Prior to multi-queue (in Linux 3.x and earlier), each block device had a single request queue protected by a spinlock. In multi-core systems and with fast storage (e.g., NVMe SSDs), that single lock became a bottleneck. **blk-mq splits the queue into multiple per-CPU (or per-hardware-thread) submission queues and multiple hardware dispatch queues** [44] [45]:

- **Software staging queues (one per CPU/core):** These are where bios are initially queued. Because each CPU can queue I/O independently, contention is reduced (each CPU has its own queue lock). The I/O scheduler (elevator) can merge and sort requests *within* each CPU's queue [46] [47]. This means merging is localized (requests from different CPUs won't be merged unless they end up on the same hardware queue later).

- **Hardware dispatch queues (one or more per device)**: These represent actual submission queues to the device hardware. blk-mq will move requests from the per-CPU software queues to hardware queues for the driver to consume [48]. If the device (or driver) supports multiple hardware queues (e.g., an NVMe device with many submission queues), blk-mq can use them for parallel dispatch. If not (e.g., a simple SATA disk), blk-mq can still use one hardware queue but still benefits from parallel software queues up to the point of dispatch.

From the driver's perspective, blk-mq means that instead of one `request_fn` processing one queue, the driver might implement a **queue request callback** that can be invoked concurrently on multiple CPUs/ hardware queues. However, blk-mq abstracts a lot of this: you register how many hardware queues you want and a set of ops, and the block layer handles the rest.

I/O Scheduling (Elevator): Linux historically allowed pluggable I/O schedulers (elevators) like **CFQ**, **deadline**, **noop**, etc., to optimize request ordering on HDDs. With blk-mq, the concept persists but implementation changed. Now we have MQ-deadline, BFQ, and Kyber as schedulers that work on the multi-queue model. The default scheduler can depend on the device type (often "none" for fast SSD/NVMe and "mq-deadline" or "bfq" for rotational media). As a driver writer, you typically don't need to interact with the I/ O scheduler directly – the block layer handles merging and sorting before you see requests. You can, however, set flags in the request queue to hint how the scheduler should behave. For example, if your device doesn't benefit from merging (maybe it's virtual or has its own caching), you can set `blk_queue_flag_set(QUEUE_FLAG_NOMERGE, queue)` or in blk-mq, use `tag_set.flags = BLK_MQ_F_NO_SCHED` / `BLK_MQ_F_SHOULD_MERGE` appropriately [49] [50]. In our simple example, we'll allow merging (since it can only help sequential I/O to our linear memory buffer) and use the default scheduler.

Queue Limits: The request queue also stores limits that describe the device's capabilities. For instance, maximum request size in sectors (`queue_max_sectors`), max number of segments in a request (`queue_max_segments`), max segment size, DMA alignment, etc [51] [52]. The block layer will not create requests that violate these limits. For example, if your device cannot handle more than 128 KiB in one request, you would set that, and the block layer would split larger bios. Similarly, `blk_queue_logical_block_size(queue, size)` informs the kernel of your device's native sector size (e.g., 4096) so that it aligns and sizes I/O accordingly [53] [54]. In our ramdisk, we'll stick to the default 512-byte logical sectors, but we will show where to set this if needed.

Request Queue Initialization (blk-mq): In older drivers, one would call `blk_init_queue(request_fn, spinlock)` to set up a request queue and provide a single request-handling callback. In Linux 6.x with blk-mq, initialization is a bit different: - You define a `struct blk_mq_ops` with your callback(s), primarily `.queue_rq` for handling requests. - You allocate and configure a `struct blk_mq_tag_set` which describes the number of hardware queues, queue depth (max requests in flight per queue), and associates your `blk_mq_ops` with the device. - You call `blk_mq_alloc_tag_set(&tag_set)` to allocate resources for the tag set (this sets up the software/hardware queue structures). - Then create the request queue with `blk_mq_init_queue(&tag_set)` which returns a `struct request_queue *`. This queue is already set up to use your `.queue_rq` handler. At this point, you can adjust queue parameters (like logical block size, max sectors, etc.). - Alternatively, as mentioned, you can skip separate queue allocation by using `dev->gd = blk_mq_alloc_disk(&tag_set, dev)`, which internally does both `blk_mq_init_queue` and `alloc_disk` for you [55] [56]. After that, you still set up `gd->major`, etc., and call `add_disk`.

For example, setting up a single-queue blk-mq for our device:

```
/* In our device init function, after allocating dev and dev->gd: */
blk_mq_tag_set_init(&dev->tag_set);            // pseudo-code for filling
tag_set
dev->tag_set.ops = &my_mq_ops;                // blk_mq_ops with .queue_rq
```

```
dev->tag_set.nr_hw_queues =
1;                // one hardware queue (since device isn't really parallel)
dev->tag_set.queue_depth = 128;             // can handle up to 128 requests at
a time
dev->tag_set.numa_node = NUMA_NO_NODE;
dev->tag_set.flags = BLK_MQ_F_SHOULD_MERGE; // allow merging of requests
blk_mq_alloc_tag_set(&dev->tag_set);

dev->queue = blk_mq_init_queue(&dev->tag_set);
blk_queue_logical_block_size(dev->queue, 512);
dev->queue->queuedata = dev;   // associate our device context

dev->gd->queue = dev->queue;
```

This is roughly what happens under the hood when using `blk_mq_alloc_disk` as well [57] [58]. If any step fails, you'd handle errors (e.g., free tag_set on failure, etc.). Once the queue is set up and attached to the gendisk, calling `add_disk` makes the device available.

Handling I/O Requests in the Driver

The **central responsibility** of a block driver is processing I/O requests from the block layer. In our driver, this means copying data between the kernel's buffer (from the bio/request) and our device's data store (the RAM buffer). With multi-queue, the block layer will invoke our request handler (`queue_rq`) potentially from multiple contexts (though for simplicity we use 1 hardware queue). We must not sleep in this context; it's like a softirq/tasklet environment, so the handler should be atomic (no blocking calls) [59]. If long processing is needed (e.g., actual disk I/O with interrupts), one would typically queue work to a thread and return quickly. But for our in-memory device, the operations are CPU-bound copies, which we can perform immediately.

Request Handler (`queue_rq`): We define a function to handle a request, for example:

```
static blk_status_t myblock_queue_rq(struct blk_mq_hw_ctx *hctx,
                                     const struct blk_mq_queue_data *qd)
{
    struct request *req = qd->rq;
    struct myblock_dev *dev = req->q->queuedata;
    blk_status_t status = BLK_STS_OK;

    blk_mq_start_request(req);  // notify block layer we are starting on this
req

    if (blk_rq_is_passthrough(req)) {
        printk(KERN_NOTICE "myblock: skipping non-fs request\n");
        status = BLK_STS_IOERR;
        blk_mq_end_request(req, status);  // complete it with error
        return status;
    }

    const bool write = (rq_data_dir(req) == WRITE);
    sector_t sector = blk_rq_pos(req);       // starting sector of the request
    unsigned int nr_sectors = blk_rq_sectors(req); // total sectors in request
    void *buf_addr;

    struct bio_vec bvec;
    struct req_iterator iter;
    rq_for_each_segment(bvec, req, iter) {
        unsigned long offset = bvec.bv_offset;
        unsigned int bytes = bvec.bv_len;
        buf_addr = kmap_atomic(bvec.bv_page);    // map the memory page
        if (write) {
            // Copy from bio buffer to device (write to disk)
            memcpy(dev->data + (sector * 512), buf_addr + offset, bytes);
        } else {
            // Copy from device to bio buffer (read from disk)
            memcpy(buf_addr + offset, dev->data + (sector * 512), bytes);
        }
        kunmap_atomic(buf_addr);
        sector += bytes / 512;  // advance sector by number of sectors processed
    }

    blk_mq_end_request(req, BLK_STS_OK);
    return BLK_STS_OK;
}
```

Let's unpack this:

- We retrieve the `struct request *req` from the `blk_mq_queue_data` provided. We also get our device structure via the queue's `queuedata` (set earlier) [19].
- `blk_mq_start_request(req)` informs the block layer that we've taken responsibility for the request [60] [61]. This is mostly relevant if the request might be requeued or handled asynchronously, but it's good practice to call it at the start of processing.
- We then check `blk_rq_is_passthrough(req)`. "Passthrough" in this context means a request that isn't a normal read/write (for example, SCSI ioctl commands or other special requests that some higher-level drivers might send down). Our simple driver doesn't know how to handle those, so we log and end the request with an I/O error [62]. Many block drivers use this check to filter out unrecognized request types.
- We determine if it's a read or write request by `rq_data_dir(req)` (0 for read, 1 for write) [63]. Alternatively, newer kernels also encode the operation in `req_op(req)` (which could be `REQ_OP_READ`, `REQ_OP_WRITE`, `REQ_OP_FLUSH`, etc.). We could also handle `REQ_OP_FLUSH` (flush cache) by simply returning success (since our ramdisk has no volatile cache to flush). In our code, any non-read/write would have been caught by the passthrough check (flush might not count as passthrough though, so for completeness one could add a case: if `req_op(req) == REQ_OP_FLUSH` then just end request successfully).

- We get the starting sector (`blk_rq_pos`) and number of sectors (`blk_rq_sectors`). These refer to 512-byte sectors (the kernel's unit) [63] . We will use them to calculate memory offsets. In our example, `dev->data` is a linear array of bytes representing the disk. The offset into `dev->data` in bytes is `sector * 512`.
- We then iterate over each segment of the request. A request may be composed of multiple disjoint memory segments (bios) if the data buffer wasn't physically contiguous or if multiple bios were merged [64] [65] . Linux provides the `rq_for_each_segment(bvec, req, iter)` macro to loop through each segment of each bio in the request [66] . Each `bvec` (of type `struct bio_vec`) gives us a **page** (`bvec.bv_page`), an offset within that page (`bvec.bv_offset`), and length (`bvec.bv_len`) for that segment [67] [68] . These segments collectively cover `nr_sectors * 512` bytes of data.
- For each segment, we map the page into kernel address space with `kmap_atomic` (since the page might be high memory not directly accessible) [67] . This returns a pointer (`buf_addr`) that we can copy data to/from. We add the `bv_offset` to get to the correct starting position in the page.
- If it's a write operation (from OS to device), we copy data **from the bio buffer to our device**: `memcpy(dev->data + (sector*512), buf_addr + offset, bytes)` . If it's a read, we do the opposite copy from device to bio buffer.
- After copying, we unmap the page with `kunmap_atomic`.
- We then increment the `sector` by the number of sectors we processed in this segment (`bytes/512`). The `rq_for_each_segment` macro ensures we iterate in order over the request; we keep track of the sector to know where in our device we are currently writing/reading. An alternative is to use `iter.iter.bi_sector` provided by the macro's iterator state, which gives the sector number corresponding to the start of the current segment [69] . For clarity, we manually updated `sector` here.
- Once all segments are processed, we call `blk_mq_end_request(req, BLK_STS_OK)` to mark the request as completed successfully [60] . This will notify the block layer to complete all bios in the request, wake up any waiting processes, etc.
- We return `BLK_STS_OK` as the status of the dispatch. In this synchronous handling, we've already ended the request. The return value here mostly matters if we were doing asynchronous handling or if we encountered an error without ending the request ourselves. (If we had not called `blk_mq_end_request`, returning `BLK_STS_OK` would indicate to blk-mq that it can consider the request handled, but the request would still be "in flight" until we end it later. If we return `BLK_STS_RESOURCE` or `BLK_STS_DEV_RESOURCE`, blk-mq would keep the request on the queue to retry later, which is useful for flow control if a device's internal queue is full. In our simple case, we handle everything synchronously.)

The above function is our implementation of `.queue_rq` in the `blk_mq_ops`. We would set it like:

```
static struct blk_mq_ops myblock_mq_ops = {
    .queue_rq = myblock_queue_rq,
    // (there are other optional callbacks like .init_request, .timeout, etc.,
not needed here)
};
```

Data Integrity Considerations: In this simple driver, we do not implement any locking around the `dev->data` memory. Because blk-mq could, in theory, issue multiple requests to us concurrently (especially if `nr_hw_queues > 1` or multiple CPU contexts), simultaneous access could occur. For a single-queue device on a single CPU at a time, the requests will be serialized. However, if our device allowed multiple hardware queues, two `queue_rq` calls could run in parallel on different CPUs (processing different requests). If those requests happen to target overlapping regions of `dev->data`, a race condition arises. In a real disk, concurrent writes to the same sector are undefined (and the device firmware handles any internal locking). For our ramdisk, we could add a spinlock around the memory access if we wanted to serialize all I/O, or more granular locking per region. For simplicity, we assume that the block layer (and filesystem semantics) won't issue overlapping writes concurrently in a problematic way. Generally, upper layers avoid writing to the same sector from two threads without synchronization. Therefore, we will not add extra locking, but it's something to keep in mind for more complex drivers.

Partial I/O and Errors: Our example assumes the entire request succeeds. If an error occurred partway (e.g., a bad sector on a real disk), a driver can complete the request partially. In old kernels, there were functions like `end_that_request_first` and `end_that_request_last` to handle partial completion. In modern blk-mq, one would typically split the request or use bio-level completion to inform exactly how many bytes were transferred before error. This is beyond our scope – our ramdisk either completes everything or doesn't fail at all (unless there's a memory fault). If we wanted to simulate an error, we could, for example, refuse to read beyond a certain sector by checking `sector` and returning an error for those. But typically, for a well-behaved block device, errors are rare.

Flushing and Discard: Two other operations worth noting are flush (force writeback of caches) and discard (aka TRIM). The block layer may send a flush request (`REQ_OP_FLUSH`) usually with no data, just as a barrier to ensure all prior writes are committed to non-volatile storage. Our ramdisk has no volatile cache, so the correct behavior is simply to succeed flushes. If flush requests come through, we could detect them via `req_op(req)` or by seeing that `blk_rq_sectors(req)` is 0 and it's a write with special flags. In practice, since our device doesn't set any write-back cache flags, the block layer might not issue flush at all. Discard requests (`REQ_OP_DISCARD`) indicate that certain sectors are no longer in use (for SSDs to free blocks). On a ramdisk, we could choose to zero the specified range or simply ignore it (treat as success). If we did implement discard, we'd look at `req_op(req)` and possibly the range indicated in the request's bios (the bios would contain no actual data, just a range). For brevity, we won't explicitly handle these, effectively ignoring them (which the block layer will interpret as not supported unless we set the queue discard capability).

By implementing the request function as above and registering it via `blk_mq_ops`, our driver can now perform actual data transfer for read and write requests. This is equivalent to the `sbull_transfer` functionality in LDD's example but updated to the bio/request model of modern kernels.

Modern Block Driver Patterns vs. Legacy Interfaces

Linux's block layer APIs have evolved since the 2.6 era described in the original chapter. It's important to highlight deprecated interfaces and their modern replacements:

- **Request Queue Initialization:** The original `blk_init_queue(request_fn, spinlock)` is deprecated and not available in new kernels (as of 5.15+, it's removed from exports). The modern approach is to use blk-mq as we did. If you come across older code using `blk_init_queue` or `blk_queue_headactive` etc., know that you should switch to blk-mq. Even for devices that effectively have one queue, you use the multi-queue API in single-queue mode (by setting `nr_hw_queues = 1`). The spinlock argument is no longer needed – locking is handled internally by blk-mq. In fact, **drivers no longer manage the queue lock**; this removes a lot of complexity from driver code.

- `struct request_fn_proc` **vs.** `struct blk_mq_ops`**:** Previously, drivers provided a single `request_fn` (like `sbull_request()`) that pulled requests off the queue via `elv_next_request()` in a loop [70]. In multi-queue, you provide `queue_rq` which is called by the block layer for each request dispatch. You no longer manually dequeue or requeue requests with functions like `blkdev_dequeue_request` or `elv_requeue_request` – blk-mq does that if you return a special status. For example, returning `BLK_STS_RESOURCE` from `queue_rq` tells blk-mq that your device cannot handle more requests right now (maybe its internal queue is full), so it should stop dispatch on that hardware queue for a moment. The driver can later call `blk_mq_run_hw_queue(queue)` to restart dispatch when ready, or blk-mq will retry automatically after a delay. In our simple driver, we never defer requests, so we always return `BLK_STS_OK`.

- **Ending Requests:** Old drivers would use `end_request()` or `__blk_end_request_all()` (and earlier, `end_that_request_last()`) to signal completion [71]. In blk-mq, the equivalent is `blk_mq_end_request(req, status)`. You provide a `blk_status_t` (which corresponds to SCSI result codes, but BLK_STS_OK and BLK_STS_IOERR are the main ones) [62]. This will handle finishing the request and calling all end_io callbacks on bios.

- **Partial Completions:** If you ever need to complete only part of a request and continue later, blk-mq provides `blk_update_request(req, bytes)` to update how much has been done and then you can `blk_mq_end_request` with an error or success for the remainder. This is more advanced and not needed in our scenario.

- `bio` **Handling:** The original text mentioned `bio_endio()` to finish a bio, and functions like `bio_sectors(bio)`, `bio_data_dir(bio)` [72]. Those are still around (though `bio_endio` is typically called by block layer when the driver ends the request). We now often use iterators like shown above. The macro `bio_for_each_segment()` is used if you handle one bio at a time outside of request context, whereas `rq_for_each_segment()` handles all bios in a request [73]. The concepts of `struct bio_vec` and `bi_io_vec` array are the same as described in LDD3 (Figure 16-1) [74] [75]. In modern kernels, a bio can also have a front and rear padding (for encryption metadata, etc.), but for basic data transfer, you usually ignore those.

- **No Request (Make Request) Mode:** The original chapter discussed a “no queue” mode using `blk_queue_make_request()` and a `make_request` callback [76]. This has been superseded by the `submit_bio` mechanism we described in block_device_operations. In Linux 6.x, if you want to bypass the entire request merging/scheduling, you can implement `.submit_bio` in your bdev ops. The kernel will call it for each incoming bio [33] [34]. In that case, you *must* handle splitting or processing bios on your own and call `bio_endio(bio)` when done. Most drivers do not need to do this, except for pseudo-drivers like Network Block Device (nbd), or layering drivers like Device Mapper, which handle routing of bios to other devices. For a regular hardware driver, the blk-mq request model is preferred as it automatically gives you scheduling, merging, and multi-queue scalability. The `nbd` driver and some others were historically “bio-based”. As of kernel 6.x, many of those have been converted to request-based as well (for performance), or they use `submit_bio` if necessary. **In summary:** you will rarely use `blk_queue_make_request` in new code – instead either go all in with blk-mq (request-based) or, if you truly need a bio-based approach, implement `.submit_bio` in bdev ops.

- **Deprecated Capacity Functions:** In early kernels you'd use `blksize_size[]` and `block_size()` to set the logical block size, but now we use `blk_queue_logical_block_size()` and friends. Also, `hardsect_size` is not much of a concern nowadays except for informing the VM of page-cache alignment; `blk_queue_logical_block_size` covers what you need in most cases [77].

- **Partition Handling:** The kernel automatically handles partition table reading. After `add_disk`, the block subsystem will read the partition table of the device (unless you suppressed it via flags). For example, when you `add_disk`, if the device is not marked as removable and has a size > 0, the kernel might attempt to read the MBR/GPT to create partition devices. If nothing is there, no partitions are registered (besides the whole disk). When a user writes a new partition table (with a tool like fdisk) and issues the `BLKRRPART` ioctl, the kernel will call your `media_changed` and `revalidate_disk` to update and then re-read the partition table from sector 0. Our ramdisk doesn't have a partition table initially, so none will appear. If we create one and re-read, the kernel will create devices like `/dev/myblock0p1`, etc. Because our driver didn't explicitly create them, it might appear magical – but it's the block core doing it. One catch: the driver should support the revalidate call to update capacity if needed (for a ramdisk, capacity doesn't change). Also, one should ensure `gd->minors` is large enough to hold the maximum partition number. In our case we set 16, which is standard (p1–p15 possible).

- **Device Features and Flags:** Newer kernels have flags in gendisk and request_queue for certain features. For example, if your device is non-rotational (an SSD/ramdisk), the block layer sets a flag so that the I/O scheduler knows it doesn't need to do typical HDD optimizations. You can explicitly set this via `queue_flag_set_unlocked(QUEUE_FLAG_NONROT, queue)` or older `blk_queue_flag_set(QUEUE_FLAG_NONROT, queue)`. Also, you can indicate whether write cache is present (`QUEUE_FLAG_WC`), and if it's volatile or not (which affects flush logic). On a ramdisk, you'd mark it as non-rotational and maybe no volatile cache (so flushes aren't needed). The kernel's default for `blk_alloc_disk` is to assume non-rotational=0 (meaning "rotational") unless changed, I believe, so we might want to explicitly clear that if it matters. Actually, modern defaults might treat unknown devices as non-rotational? (In any case, we can set `blk_queue_flag_clear(QUEUE_FLAG_ADD_RANDOM, queue)` too if we don't want to contribute to entropy pool – not critical here.)

For our purposes, it's enough to know that **the modern API uses `blk_mq_alloc_disk` and `blk_cleanup_disk`** to simplify driver code [23] [24]. Many old functions (alloc_disk, put_disk, etc.) still exist but are thin wrappers or have been phased out. The code we've written above follows current best practices.

Putting It All Together: A Simple Ramdisk Driver Example

Let's summarize the steps to write a minimal block driver (for a RAM disk) in Linux 6.x, and then we'll discuss how to test it with QEMU:

Driver Implementation Outline:

1. **Define Device Parameters:** e.g. name, size, etc. For example, a module parameter for disk size or use a constant number of sectors.

2. **Allocate/Initialize Device Structure:** Include a pointer for the memory buffer and metadata like the tag_set, request_queue, gendisk, etc.

3. **Setup the request handling function** (`queue_rq` as shown earlier). This function copies data to/from the memory buffer. Also implement any needed bdev ops (open, release, ioctl, etc., mostly trivial as discussed).

4. **Module Initialization (`myblock_init`):**

5. Register the block device major: `register_blkdev`.
6. Allocate the ram disk memory (`vmalloc` or `kzalloc` for large size if needed).
7. Initialize the tag_set (with 1 hw queue, an appropriate depth, and point to `myblock_mq_ops`).
8. Allocate the tag_set (`blk_mq_alloc_tag_set`).
9. Allocate the gendisk with queue: either call `blk_mq_alloc_disk(&tag_set, dev_ptr)` or do `blk_mq_init_queue` + `alloc_disk`. Using the helper, e.g.:

   ```
   dev->gd = blk_mq_alloc_disk(&dev->tag_set, dev);
   ```

   Check for errors (IS_ERR). If it fails, free the tag_set and memory and unregister major.
10. Set `dev->gd->major = <major>; dev->gd->first_minor = 0; dev->gd->minors = <n>; dev->gd->fops = &myblock_ops; dev->gd->private_data = dev; disk_name; set_capacity`.
11. Optionally set `blk_queue_logical_block_size(dev->gd->queue, 512)` (or another if using a different sector size) [78] .
12. If the device is truly non-rotational, you can hint: `blk_queue_flag_set(QUEUE_FLAG_NONROT, dev->gd->queue)`.

13. Finally, call `add_disk(dev->gd)`.

14. **Module Exit (`myblock_exit`):**

15. Call `del_gendisk(dev->gd)` to remove disk.
16. Call `blk_cleanup_disk(dev->gd)` to cleanup queue and disk [23] .
17. Free the tag_set (`blk_mq_free_tag_set(&dev->tag_set)`).
18. Free the memory buffer.

19. Unregister the major (`unregister_blkdev`).

If all goes well, after inserting the module, you should see a new block device (check `dmesg` for confirmation and major number). You can also see it in `/proc/devices` under block devices. On a system with devtmpfs (typical), a device node `/dev/<name>` will be created automatically.

Testing the Driver with QEMU:

To verify the driver's functionality, we can use a virtual machine (QEMU) running a Linux kernel that includes or can load our module. The steps to test are:

1. **Build the module:** Compile the driver module against the kernel headers. For example, if using kernel build system, ensure the Makefile adds your module to the build or use an external module build with `obj-m += myblock.o`. Load the module into the QEMU VM (e.g., via `insmod`).

2. **Check dmesg:** Upon insertion, you should see logs from `register_blkdev` (if dynamic major, it will print the allocated major) and any `printk` from your init. Also confirm "Add disk" message if you put one, etc. For example:

   ```
   myblock: registered with major 240
   myblock: disk size 2048 sectors (1 MB)
   ```

   (These are hypothetical log lines if you added them.)

3. **Find the device node:** There should be `/dev/myblock0` (assuming disk_name was set to "myblock0"). Verify it exists. You can also check `lsblk` or `cat /proc/partitions` to see it listed with the correct size (in our example ~1MB).

4. **Read/write tests:** You can use `dd` or `hexdump` to test raw I/O. For example,

   ```
   dd if=/dev/zero of=/dev/myblock0 bs=512 count=10
   dd if=/dev/myblock0 bs=512 count=10 | hexdump -C
   ```

   This should write zeros to the first 10 sectors and read them back (as zeros). Try writing non-zero data:

   ```
   echo "Hello Block Device" | dd of=/dev/myblock0 bs=1 seek=1024
   dd if=/dev/myblock0 bs=1 count=20 skip=1024
   ```

   Ensure the output matches "Hello Block Device".

5. **Filesystem test:** Create a filesystem on the device. For instance:

```
mkfs.ext4 /dev/myblock0
```

The `mkfs` should succeed and report the number of blocks, etc., for a 1MB device. Mount it:

```
mkdir /mnt/test
mount /dev/myblock0 /mnt/test
```

Then copy a small file or create one:

```
echo "Testing 123" > /mnt/test/file.txt
ls /mnt/test
cat /mnt/test/file.txt
```

You should see the file and its content. The data is being stored in your driver's memory buffer. If you unmount and remount, it should persist (as long as the module remains loaded).

6. **Partitioning (optional):** You can use `fdisk /dev/myblock0` or `parted` to create a partition. For example, in `fdisk`, make a single partition spanning the whole device, write the table, then use the `blkdevparts` or re-read partition ioctl. The kernel should then detect `myblock0p1`. If your driver did not implement `media_changed`/`revalidate`, you might need to manually invoke `partprobe` or re-load module to see the partition. (The original `sbull` driver simulated media changes with a timer to test this path [79], but in our case it's static media.)

7. **Cleanup:** Unmount any filesystem and remove the module (`rmmod`). Ensure it unloads without issues (no kernel oops or hangs). If you wrote to the device, data in the buffer is lost when module unloads (since it's just RAM), but that's expected for a ramdisk.

By following these steps, you'll have built and tested a basic block driver. While this example is a in-memory disk (no hardware needed), it demonstrates all key aspects: registration, request handling, and integration with the Linux block layer. The same principles apply to real hardware drivers – the main difference is that instead of a memory copy, a real driver would program a DMA transfer or PIO to the device in the request handler, and then asynchronously end the request when the device interrupt signals completion.

QEMU-Compatible Exercise: Build and Verify a Minimal Block Driver

Goal: Create a simple block device driver that implements a RAM-backed disk, register it with the kernel, and verify its operation by formatting and mounting it in a QEMU virtual machine.

Steps to Complete:

1. **Write the Driver Code:** Using the outline above, implement the `myblock` driver. Ensure you include:
2. Module init/exit functions registering the device (with `register_blkdev` if needed), setting up `gendisk` + `request_queue` (using blk-mq), and adding the disk.

3. A `struct block_device_operations` with at least `.open`, `.release`, and `.owner` filled (can be no-ops for this exercise).
4. A request handler (`queue_rq` via `blk_mq_ops`) that handles read and write requests by copying data to/from a memory buffer.
5. Allocation of a memory buffer for the disk's storage. (Tip: use `vzalloc` for larger sizes to safely allocate contiguous virtual memory.)

6. Proper cleanup of all resources on module exit (`del_gendisk`, `blk_cleanup_disk`, freeing memory, unregistering major).

7. **Choose Disk Size:** For testing in QEMU, a few megabytes is plenty (to avoid long mkfs times). For example, set `disk_sectors = 10240` (5 MB) or so. Remember to multiply appropriately if using bytes.

8. **Compile and Insert Module in QEMU:**

9. If using an environment like Linux Kernel Labs, you can use their provided build system to compile the module and then use their QEMU setup. Otherwise, compile the module with the kernel build system (`make M=path/to/your/module`) or an external Makefile.
10. Boot the QEMU VM (with a kernel that has support for loadable modules or include the driver statically if you prefer).

11. Insert the module: `insmod myblock.ko`. Check `dmesg` for success messages or any errors.

12. **Verify Device Node:** Ensure `/dev/myblock0` (or whatever name you gave) exists. If not, create it with `mknod` using the major number from `dmesg` (and minor 0).

13. **Test Read/Write:** Use `dd` or `cat` as described to test raw I/O. A convenient test is:

```
echo "hello" | dd of=/dev/myblock0 bs=1 seek=1000
dd if=/dev/myblock0 bs=5 count=1 skip=200 >/tmp/out.txt
cat /tmp/out.txt   # should show "hello"
```

    This writes "hello" at offset 1000 and reads it back from offset 1000 (which is skip=200 blocks of 5 bytes each, i.e., 1000 bytes).

14. **Make a Filesystem:** Run `mkfs.ext4 /dev/myblock0` (or `mkfs.xfs`, or even `mke2fs` etc.). It should detect the device size and create a filesystem. Then mount it on a directory:

```
mkdir -p /mnt/myblocktest
mount /dev/myblock0 /mnt/myblocktest
```

    If mount succeeds, create a file: `echo "Block Device OK" > /mnt/myblocktest/test.txt`. Then read it back: `cat /mnt/myblocktest/test.txt`.

15. **Unmount and Clean Up:**

```
umount /mnt/myblocktest
```

Remove the module: `rmmod myblock`. Check that the system did not crash and the module removed cleanly (the device node will vanish if using devtmpfs).

16. **Stretch Goal – Partitioning:** (Optional) After creating a filesystem directly, you might also test partitioning:

17. Use `fdisk /dev/myblock0` to create a partition (e.g., a single primary partition). After writing the table, run `partprobe /dev/myblock0` or reinsert the module to see `/dev/myblock0p1`. Then try formatting `/dev/myblock0p1` and mounting it. This will validate your `revalidate_disk` path if implemented. If not implemented, the kernel might not realize the partition changed; in that case, reloading the module (which calls `add_disk` again) will cause the kernel to read the new partition table on startup. This is not ideal for a real driver, but acceptable for this exercise.

By completing the above, you'll have a working block driver. This driver, while simple, contains the core logic that any block driver uses. From here, you could experiment with enhancements: for example, add support for multiple disks (multiple gendisks with different minors), or simulate a removable media by implementing `media_changed` (perhaps via a module parameter that you can toggle via sysfs), or measure performance with `hdparm -t`. The fundamental concepts – registration, request queues, and data transfer via bios/requests – will apply to any block device you write in the future.

Good luck, and happy hacking on the block layer! [80] [81]

---

[1] [2] [3] [4] [5] [7] [8] [9] [10] [11] [18] [21] [29] [30] [31] [32] [42] [43] [71] [72] [76] [79] ch16.pdf
file://file_000000004a90722fab95880cca6427bf

[6] Chapter 11 Drivers for Block Devices (Writing Device Drivers)
https://docs.oracle.com/cd/E19683-01/806-5222/block-34861/index.html

[12] [15] [17] [19] [20] [25] [26] [27] [28] [33] [34] [35] [36] [37] [38] [39] [40] [41] [44] [45] [46] [47] [48] [53] [54] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [73] [77] [78] [80] [81] SO2 Lab 07 - Block Device Drivers — The Linux Kernel documentation
https://linux-kernel-labs.github.io/refs/heads/master/so2/lab7-block-device-drivers.html

[13] [14] [22] [Drbd-dev] [PATCH 05/26] block: add blk_alloc_disk and blk_cleanup_disk APIs
https://lists.linbit.com/pipermail/drbd-dev/2021-May/005752.html

[16] Linux Block Device Architecture
https://yannik520.github.io/blkdevarch.html

[23] How does a linux block driver ensure all I/O has finished before ...
https://stackoverflow.com/questions/76894167/how-does-a-linux-block-driver-ensure-all-i-o-has-finished-before-unloading

[24] Linux 5.17: brd.c - Cregit
https://cregit.linuxsources.org/code/5.17/drivers/block/brd.c.html

49 50 Linux v6.6.1 - include/linux/blk-mq.h
https://sbexr.rabexc.org/latest/sources/69/11346615006444.html

51 52 70 Драйверы устройств Линукс: 16.3. Request Processing
http://linuxdrivers.blogspot.com/2011/05/163-request-processing.html

55 56 57 [PATCH OLK-5.10 v2 19/38] blk-mq: add the blk_mq_alloc_disk APIs - Kernel - mailweb.openeuler.org
https://mailweb.openeuler.org/archives/list/kernel@openeuler.org/message/YARP6VIXGOLISTCULMYJIN7SLD3TS4J3/

74 75 ,ch16.28124
http://static.lwn.net/images/pdf/LDD3/ch16.pdf