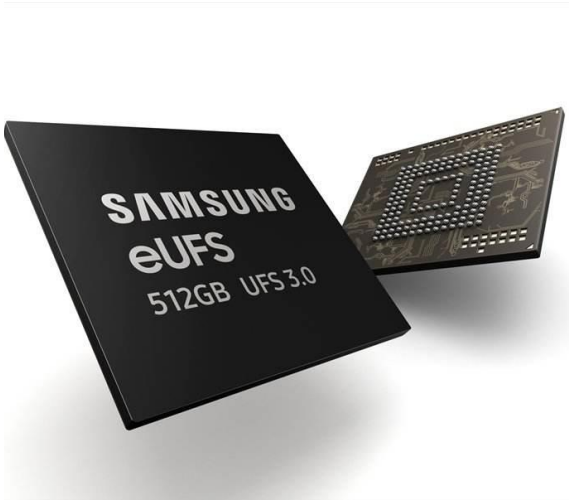
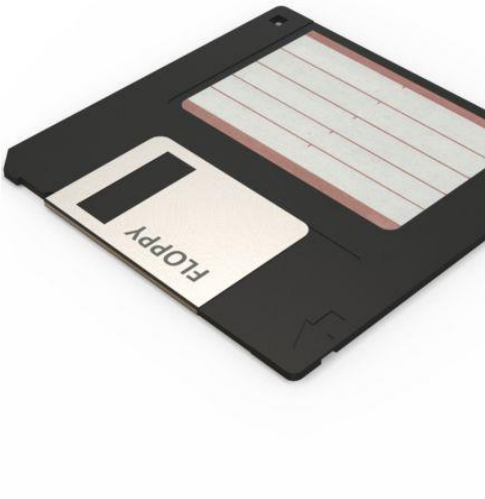
块设备驱动开发

李山文 著 第一版

基于 Linux 5.14 内核





官方技术交流群 QQ: 806452875

第一章 绪论

1.1 从存储技术说起

不管是在何种年代，数据的记录永远是最基础的技术也是最重要的技术之一。早在 1877 年，爱迪生发明了世界上第一台留声机，在今天看来，其原理非常简单，但在当时算得上是技术上的一大跨越，从此，人类有能力将声音记录下来。随着技术的不断发展，特别是半导体技术的突飞猛进，半导体存储技术成为信息存储的主流已经成为事实。

早在计算机发展初期，程序都是存储在卡带上的，如下图所示，卡带上面有很多孔，这些孔和位置记录了二进制信息，科学家将事先写好的程序做成这种卡带，然后放在计算机中就可以开始运行了。

图 1-1 早期计算机使用纸带记录程序

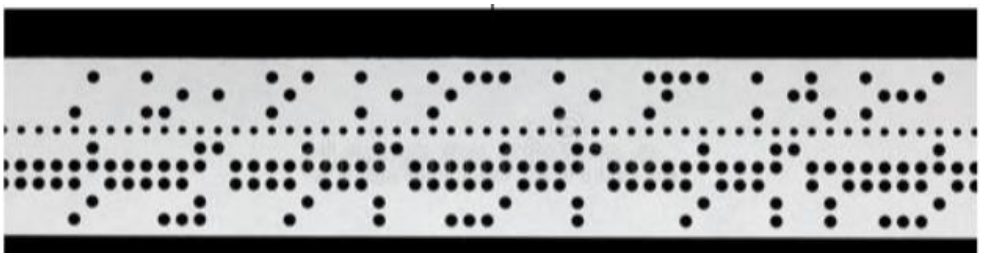


图 1-2 早期计算机程序纸带

这种方式和卡带音乐盒的音乐记录方式尤其类似，虽然随着技术的不断发展，存储介质的不断更新换代，但其最基本的原理还未变化，也就是任然是以二进制的形式存储在介质中。

1.2 半导体存储技术

半导体技术的发展同时推动着存储器的发展，人类从最开始的磁带存储大容量信息，随后发展到密度更高的磁盘（软盘）存储信息。之后人类发明了巨磁电阻效应之后，硬盘（Hard Disk Drive）便开始占领计算机存储介质的垄断地位，在此期间，虽然出现过 Flash 存

储器，但由于无法做到大容量导致其市场占用率低。随后 NAND Flash 的出现打破的这一局面，开始人们不太看好 NAND 存储器，主要原因是其寿命太短，同时其稳定性较差，但各种硬件纠错算法解决了这一问题，随后的 MLC 和 QLC 的出现使得 NAND Flash 的容量呈几何增长，其价格也变得与硬盘相当。从此 NAND Flash 开始成为主流存储介质，其代表有 SSD（Solid State Disk）、eMMC、UFS、NVMe[1]。

图 1-3 存储器发展过程

1.2.1 机械硬盘（HDD）

也许再过十几年，人们都不曾听过这个名字，机械硬盘顾名思义即具有机械装置的硬盘，其基本原理就是利用巨磁电阻效应来实现的一种磁存储设备。磁盘是记录数据的主体，所有的数据都记录在磁盘上。读取和写入到过程正好相反，正因为由于巨磁电阻效应的发现，才使得如今的 HDD 容量呈几何增长。机械硬盘的结构如下图所示，主要有盘面、读写头、电源接口、数据线接口、驱动配置接口、电路控制板。

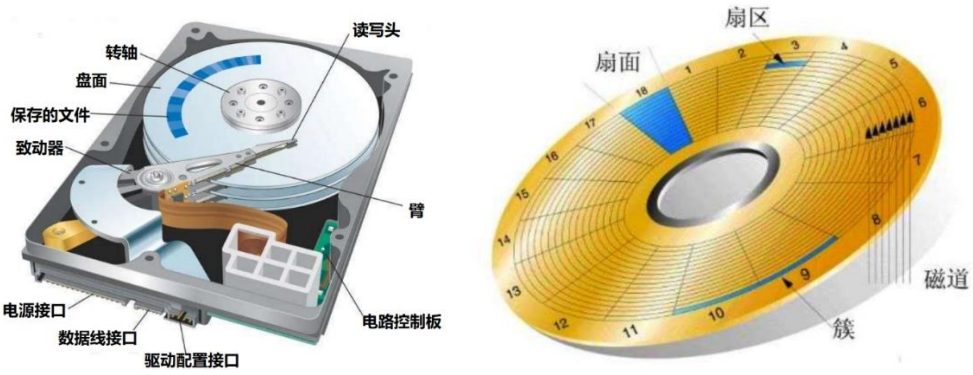


图 1-4 机械硬盘结构体

由于一般所有的硬盘都有控制器，在写数据时不会立刻写到硬盘上，而是先缓存在 RAM 中，当数据达到一定数量时才会写入到硬盘中，这样做的目的是尽可能减少磁头的读写次数，从而延长硬盘的使用寿命，因此每次写入的数据量都是一块一块的。对于 HDD 而言，有一个最小的写入数据量，我们称为最小写入粒度，这个大小由磁盘的扇区决定。什么是扇区呢？在 HDD 中，磁盘的盘面被分为一圈一圈的磁道，而每个磁道又被分割为一个一个的扇区，一个扇区就是磁盘的最小写入数据量，大部分的 HDD 的扇区大小一般是 512Byte。那什么是蔟呢？在很多操作系统中，读写的数据量最小单位并不是一个扇区，例如 DOS 认为一个扇区的读写粒度太大，读写次数频繁了，因此 DOS 规定 16KB 为一个最小的读写数据量，我们将这个 16KB 称为一个蔟，也就是蔟的大小由操作系统决定，一般 DOS 分区表中为对其进行说明。那什么

[1] 实际上，这些都是 SSD，只是其协议不同，此处是为了更细化区分。

是柱面呢？实际的 HDD 一般不会是单个盘，而是由多个盘叠加在一起组合而成的，我们将不同的盘面的相同的磁道称为一个柱面。

但随着半导体技术的飞速发展，机械硬盘逐渐被固态硬盘所取代，然而很多术语却保留了下来，很多书中，NAND Flash 的最小读写数据仍称为扇区，实际上对于 Flash 而言，根本不存在扇区，这个只是历史的术语延续下来了。

1.2.2 掩膜 ROM

在电脑的初期，很多 BIOS 芯片所使用的正是掩膜 ROM，那什么是掩膜 ROM 呢？在很早之前，还没有出现 Flash，甚至连 EPROM 都没有出现，这个时候一些关键代码如何存储呢？存储在硬盘中？这个显然不行，因为硬盘是不可寻址的，所谓不可寻址就是说 CPU 中的地址和数据总线无法访问该区域，因为硬盘是需要控制器的，所有的数据都需要控制器来读写，然而控制器是需要协议通信的，因此 CPU 是不可能在硬盘中执行最初的代码。那么这段代码一定是存放在能够直接寻址的存储器中，早期的解决办法很简单，就是利用晶体管来实现数据存储阵列。

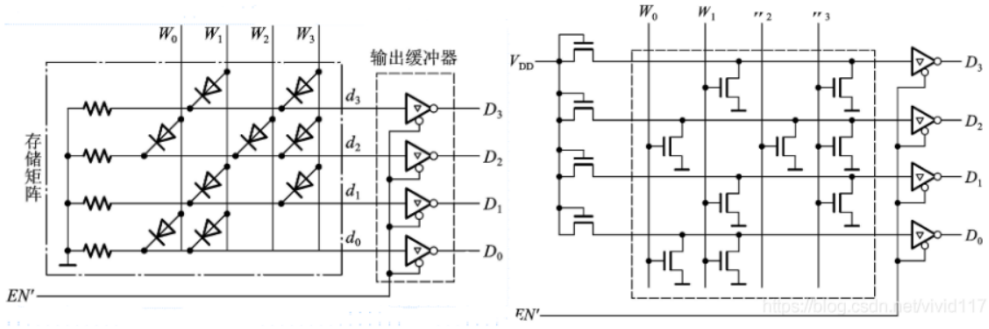


图 1-5 利用晶体管实现 ROM

上图左边是利用二极管来实现存储整列的，右边是利用 MOS 管来实现存储整列的，这些晶体管通过掩膜工艺集成在特定的芯片中，因此这种芯片中的数据是不可再修改的。

1.2.3 EPROM 和 EEPROM

掩膜 ROM 只能在生产的时候将预先需要写的数据直接制作出来，但在大多数时候，人们总是希望数据能够二次修改，随后人们就发明了一种能够通过紫外线来擦除 ROM 中的数据，这也就是 EPROM[2]。

图 1-6 EPROM、EEPROM 存储器

[2] EPROM：Erasable Programmable Read-Only Memory

在这种 EPROM 的原理是基于浮栅晶体管的特性，这总行浮栅晶体管是在 MOS 管的基础上改进而来。在 MOS 的中间层添加一个浮空的可以存储电荷的栅极，这个栅极是悬浮在中间的，因此称为浮栅。在浮栅上面会有一个透明电极，当需要写入数据时，只需要在透明电极上施加高压，这样由于隧穿效应，衬底中的电子会进入浮栅中。即使此时透明电极的电压消失，此时浮删中的电子仍然存在，只要源极和漏极存在电势差，那么由于浮栅中存在大量电子，此沟道会积累很多的正电荷，这样就形成了导电沟道。当浮栅被强烈的紫外线照射时，此时浮栅中的电子就会流入源极或者漏极中（电子吸收紫外线后，变得非常活跃，隧穿效应加剧）。 这样就实现了擦除。

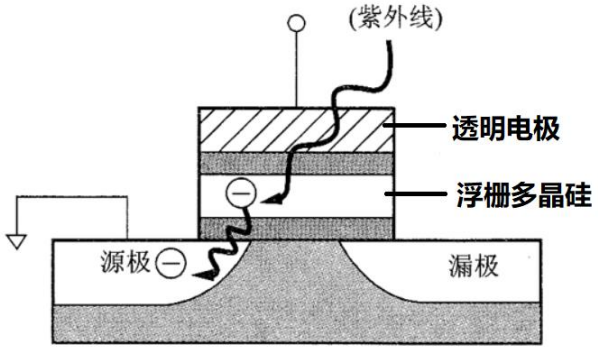


图 1-7 EPROM 原理

然而每次利用紫外线擦除非常不方便，科学家又再此基础上进行了改进，发明了 EEPROM[3]，即电可擦除只读存储器。顾名思义，该存储器可以利用电来擦除，不需要强烈的紫外线了。其基本原理和 EEPROM 很相似，将浮栅中的电子消除，我们需要用紫外线，能否有其他办法呢？答案是有。不就是将浮栅中的电子拉出来嘛，我们施加反向的电场不就行了么，没错，在透明电极和存底之间施加强电场，就可以将浮栅中的电子拉出来。这样 EEPROM 的透明电极也就不用做成透明的了，可以直接密封在芯片内部了。

1.2.4 Flash 存储器

虽然 EEPROM 可以实现电可擦除了，但是其速度比较慢，为了能够加快 EEPROM 的读写速度，人们发明了 Flash 存储器。

我们的目的就是希望加快 EEPROM，因此这就需要缩短浮栅中的电子的写入和拉出的时间。最简单的办法便是减小浮栅的厚度以及浮栅到沟道的厚度，但是减小厚度这样就造成了寿命短不稳定问题。方法总比问题多，实际上写入和擦除方式有很多种，EEPROM 的擦除方式是利用隧穿效应，这种方式速度一般在 ms 级别。而还有一种方式是 us 级别的，这就是热电子注入方式。这里我们不去深究这种方式，我们直需要直到这种方式速度比隧穿效应快，Flash 正是利用这种方式来实现编程。

但是 EEPROM 的擦除电路较为复杂，如果 Flash 仍然采用这种擦除的电路，那么面积会占用很大，为此，Flash 简化了擦除电路，只能支持一个块为最小单位擦除，这样电路就可以做的比较简单，集成度就可以增加。 同时 Flash 的擦除方式为热电子注入方式，这种方式有个缺陷，即只能写入 0，不能写入 1，因此 Flash 在写入数据之前必须对所在的块进行擦除（全部

[3] EEPROM：Electrically Erasable Programmable read only memory

写入 1）[4]，然后才能下入数据。因此 Flash 的容量一般比 EEPROM 要大得多。

半导体存储中最常见的存储器便是 NOR Flash（NOT OR，即或非）和 NAND Flash（NOT AND，即与非）。

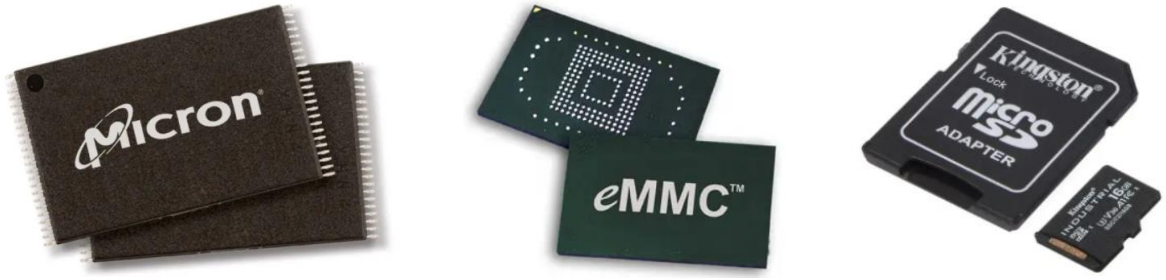


图 1-8　NOR Flash、NAND Flash(eMMC、SD 卡)

NOR Flash 和 NAND Flash 的结构如下图所示：

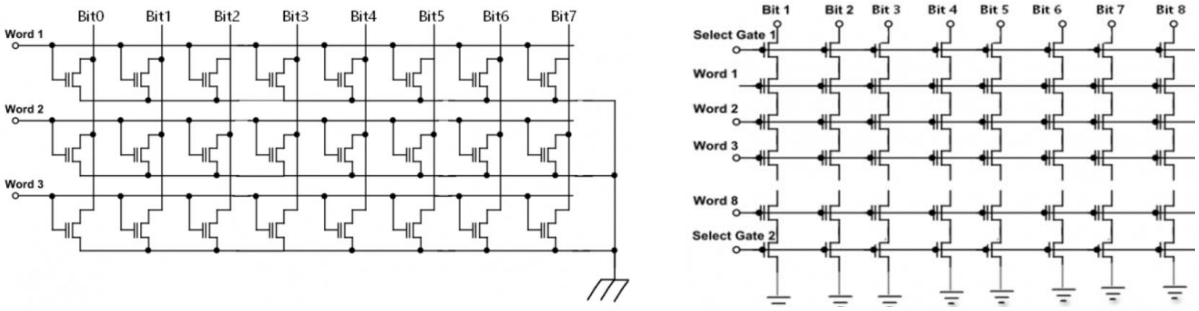


图 1-9　NOR Flash、NAND Flash 结构图

NOR Flash 和 NAND Flash 不同之处在于两者的连接方式不同，对于 NOR Flash 而言，其连接方式为并联的方式，这样每一位都可以通过行和列线来读写，也就是我们说的 word 线和 bit 线；而对于 NOR Flash 而言，其连接方式为串联，并不是每一位都能够读写，而是必须以同一个 word 线来读写，因此这就造成了其存储方式的不同，这样就导致 NOR Flash 需要配合比较复杂的控制器来完成读写。下图是这两种 Flash 的读写速度和寿命的对比图：

表 1-1 NOR 与 NAND 的参数对比

对比项	NOR	NAND
是否够独立寻址	是	否
读速度	快	稍慢
写入/擦除速度（一个扇区）	750ms	2ms
控制器复杂度	简单	复杂
寿命	10 万次	10 以上万次（SLC）
可靠性性	可靠	不可靠（需要纠错）

[4] 向浮栅中注入电荷表示写入了'0'，没有注入电荷表示'1'

1.3 NAND 存储技术

如今 NAND 存储器已经广泛应用于电子产品中，因此有必要对 NAND 存储器有一个非常清晰的认识。首先，我们看下

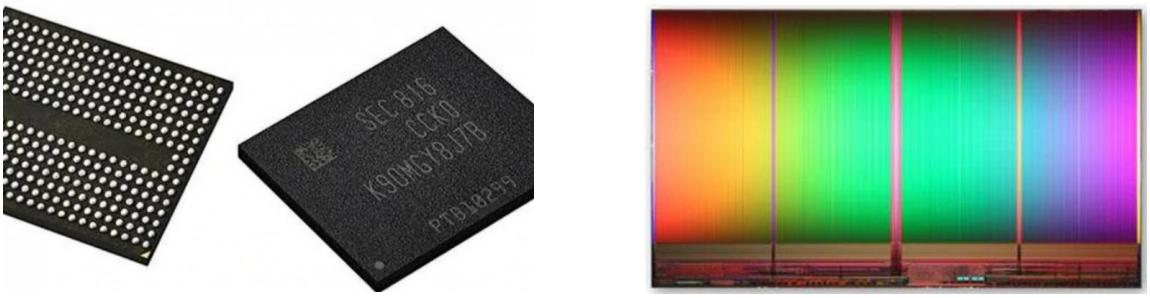


图 1-10 NAND 颗粒和晶圆

对于 HDD 而言，写数据是可以直接在

第二章 块设备驱动

为了规范块设备驱动，Linux 内核对存储设备进行了合理的管理，Linux 下的所有存储设备都被抽象为块设备，最常见的便是我们的文件读写。下面我们来简单认识下并讲解块设备的基本框架。

2.1 文件系统

什么是文件系统？所谓文件系统就是将文件以规定的格式存放在硬盘上，这个存储格式就是文件系统。如果我们的数据不以规定的格式存放，那么我们再去读取数据时就无法知道数据是啥了，而是一堆二进制数据，因此所有的数据必须按规定的格式存放，这就出现了文件系统。我们大多数人接触的最常见的文件系统便是 FAT32 文件系统，该文件系统是由 DOS 文件系统发展而来，我们都知道，DOS 文件系统是 DOS 操作系统中的文件系统。在字符设备驱动中，我们讲过 FAT32 的分区表，该文件系统在 Windows 中非常常见，其结构如下图所示：

图 2-1 FAT32 文件系统结构

2.2 虚拟文件系统

在字符设备驱动中提到过虚拟文件系统，我们都知道根文件系统是 Linux 挂载的第一个文件系统，然而实际上虚拟文件系统（Virtual File System）才是 Linux 挂载的第一个文件系统。为何会存在虚拟文件系统呢？这个就是 Linux 的强大之处，由于 Linux 操作系统可以支持非常多的文件系统，例如 ext、ext2、ext4、minix、umsdos，msdes、fat32 、ntfs、proc、stub、ncp、hpfs、affs、sysfs 等等。这些文件系统都有各自的格式，为了将这些文件系统统一起来，Linux 内核实现了一种中间层的文件系统接口，这就是虚拟文件系统。虚拟文件系统让驱动开发者不

需要去关心此时使用的是何种文件系统，只需要调用统一的文件操作接口即可。

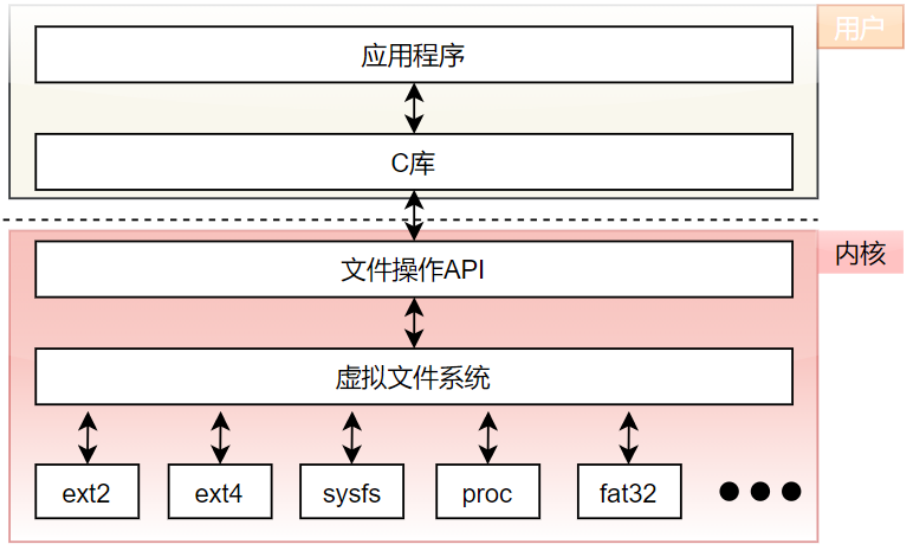


图 2-2 虚拟文件系统

从上图可以看到，虚拟文件系统并不是真正的文件系统，应用程序通过系统调用之后会进入内核态，C 库操作的是虚拟文件系统，而虚拟文件系统需要经过真实的文件系统才能对实际的存储介质进行读写。

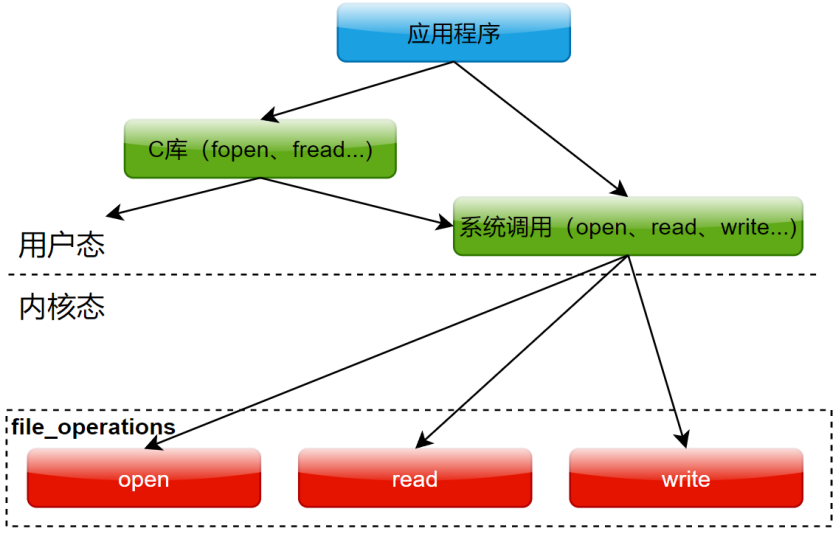


图 2-3 C 库通过系统调用来实现调用内核中的库函数接口

上图给出了系统调用的关系图，可以看到系统应用程序调用 C 库或者系统调用函数来编写各个的任务，而 C 库则通过系统调用来实现对内核中的接口访问，那为何要存在 C 库呢？应用程序直接系统调用岂不更快？在实际的过程中，每个应用程序的开发都是完全不同的，如果每个人都去直接操作系统调用，那么内核将需要提供无穷多个接口来满足不同应用场景，显然这是不现实的。这也是 POSIX 规范诞生的主要原因。为了让应用程序调用统一的接口，C 库就诞生了，C 库封装了基本的文件读写操作函数，同时还封装了大量的通用接口，这样内核只要能够为这些接口提供对应的操作即可。可以看到，C 库实际上是应用程序调用的接口，当然现在几乎所有的操作系统都集成了 C 库，在 Linux 中一般使用系统调用函数更多一些。

2.3 块设备驱动框架

由于操作系统和块设备紧密相关，因此 Linux 内核对块设备做了非常多的工作，例如对块设备进行了分层，对上层提供统一的块设备读写接口，对下层提供标准的实现对硬件的读写接口，当然中间还有各个不同设备的协议层，这些标准的协议被封装为不同的子系统，例如我们常见的 SCSI、MMC、MTD 等，我们先有一个简单的认识，后面再深入理解。

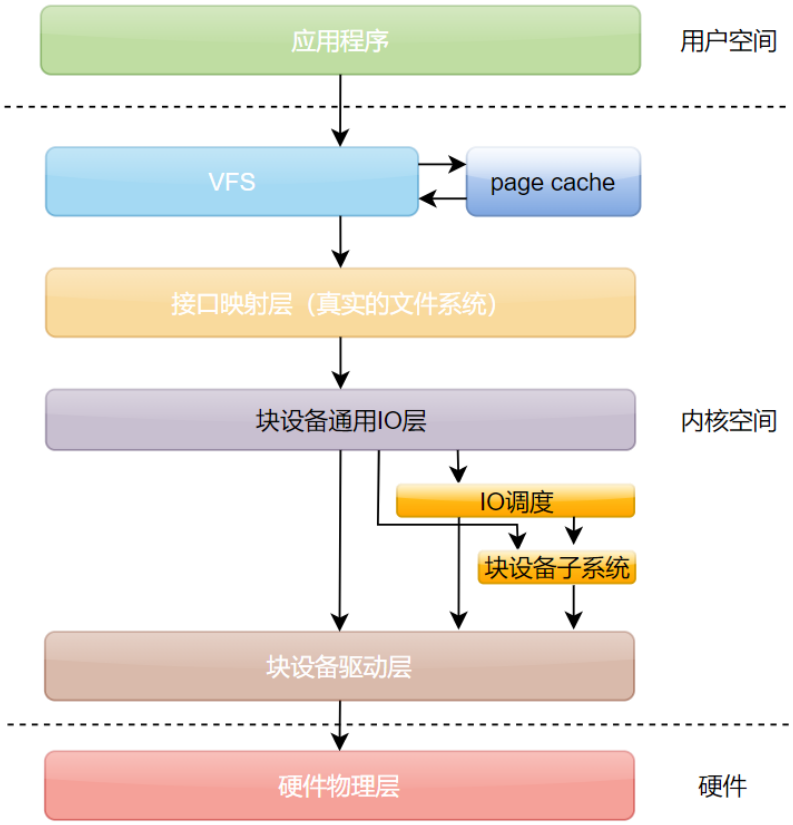


图 2-4 Linux 中块设备驱动结构

可以看到，就驱动开发而言，我们并不需要过多的了解上层的机制，我们只需要去实现块设备驱动层的接口，这些接口是需要实际操作各个硬件的。Linux 内核的这种层次化设备很大程度上降低了驱动开发者的难度，同时也提高了代码的可重用性。

对于机械硬盘、光盘等设备，块设备驱动会走 IO 调度，这是因为此类设备的读写都是机械的，顺序读写的性能较好，而随机读写的性能非常差，因此，为了提高读写性能，我们可以将多个读写命令先按读写的位置顺序排列后，再发送给磁盘控制器，这样就可以减少随机读写的概率。

而对于 SSD、UFS、NVMe、MMC 等 Flash 存储设备，由于其结构并不是机械结构，因此可以实现快读的随机读写（较机械硬盘而言）。我们的 IO 调度中的排序对读写性能并没有改善，相反由于引入了一个 IO 调度层导致其路径加长反而下降，因此对于此类块设备，Linux 中没有走 IO 调度层，而是直接都块设备驱动层。

page cache 的作用是能够缓存一部分的数据，当 VFS 读取数据时，其最先会查找 page cache 中有无数据，如果没有数据才会去往下层走，这样做的目的是减少磁盘的读写次数。

在 Linux 中，所有的块设备读写都被抽象为一个 request，该结构中包含了最重要的 bio 结构体，这个结构体描述了读写的所有信息。如下图所示，在字符设备中，我们的 ops 包含了设备的 open、release、read、write 等，其中 read、write 对设备进行读写操作。在最新的 Linux 内核中，其读写被抽象为 submit_bio 操作，该操作来完成硬盘的读写。为何要引入这个呢？这个是由于历史原因，在最较老版本的内核中，使用的是 make_request_fn 函数来实现读写，该部分是独立出去的。

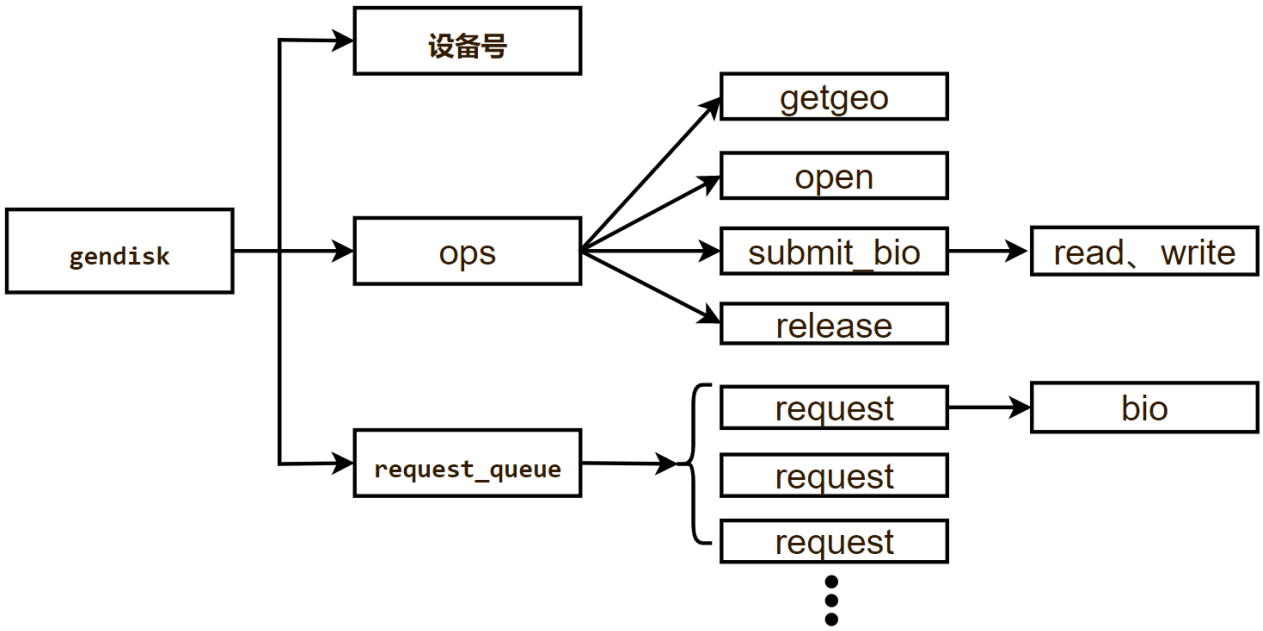


图 2-5 块设备驱动结构体关系图

下面我们将详细对各个结构体进行介绍，同时通过一个简单的块设备驱动来讲解简单的块设备驱动开发流程。

2.4 块设备相关结构体

上面我们简单描述了下块设备驱动的整体结构，下面我们将对块设备驱动中涉及到了结构体进行说明。对于机械硬盘的块驱动而言，IO 层发送的 request 都会被放在 IO 调度层中进行排序，完毕之后再发送给硬盘驱动程序来将数据封装成命令对磁盘设备进行读写。而对于固态硬盘而言，其流程中就不需要进行 IO 调度，而是直接发送命令，当然，在 Linux 中也有很多子系统，大部分时候我们的块设备驱动都会经过子系统来实现。首先我们先来了解下块设备驱动相关的基本结构体，然后在利用两个简单的块设备驱动说明其驱动的编写步骤。

2.4.1 gendisk

该结构体类似于字符设备驱动中的 cdev 结构体，其对所有的块设备进行了抽象，具体如下所示：

```
struct gendisk {
    /* major, first_minor and minors are input parameters only,
     * don't use directly.  Use disk_devt() and disk_max_parts().
```

```
     */
    int major;          /* major number of driver */
    int first_minor;
    int minors;                     /* maximum number of minors, =1 for
                                     * disks that can't be partitioned. */
    char disk_name[DISK_NAME_LEN];  /* name of major driver */
    unsigned short events;      /* supported events */
    unsigned short event_flags; /* flags related to event processing */
    struct xarray part_tbl;
    struct block_device *part0;
    const struct block_device_operations *fops;
    struct request_queue *queue;
    void *private_data;
    int flags;
    unsigned long state;
#define GD_NEED_PART_SCAN       0
#define GD_READ_ONLY            1
#define GD_QUEUE_REF            2
    struct mutex open_mutex;    /* open/close mutex */
    unsigned open_partitions;   /* number of open partitions */
    struct kobject *slave_dir;
    struct timer_rand_state *random;
    atomic_t sync_io;       /* RAID */
    struct disk_events *ev;
#ifdef  CONFIG_BLK_DEV_INTEGRITY
    struct kobject integrity_kobj;
#endif  /* CONFIG_BLK_DEV_INTEGRITY */
#if IS_ENABLED(CONFIG_CDROM)
    struct cdrom_device_info *cdi;
#endif
    int node_id;
    struct badblocks *bb;
    struct lockdep_map lockdep_map;
};
```

可以看到，块设备的结构体比 cdev 结构体复杂的多，其主要成员如下：

- major：主设备号
- first_minor：次设备号，对于多个分区的设备，每个分区都应该有一个次设备号
- minors：次设备号个数（分区个数），只有一个分区而言，此值为 1
- disk_name：设备名称
- fops：操作集合
- queue：请求队列，该指针指向磁盘的读写操作

主设备号和次设备号与字符设备类似；这里的 fops 后面会讲到，该指针为块设备的一些操作集合，例如 open、release、ioctl 等，但是该操作集合中没有 write 和 read，对于块设备驱

动而言，其读写操作单独用 request_queue 来表示；queue 便是指向 request_queue 的指针。

2.4.2 block_device_operations

该结构体便是 gendisk 结构体中的 fops，从结构体的名字上看，和 file_operations，block 也是需要对实际的块设备进行操作，因此需要提供必要的接口供上层调用。此结构体如下：

```
struct block_device_operations {
    blk_qc_t (*submit_bio) (struct bio *bio);
    int (*open) (struct block_device *, fmode_t);
    void (*release) (struct gendisk *, fmode_t);
    int (*rw_page)(struct block_device *, sector_t, struct page *, unsigned
         int);
    int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
    int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned
         long);
    unsigned int (*check_events) (struct gendisk *disk,unsigned int clearing);
    void (*unlock_native_capacity) (struct gendisk *);
    int (*getgeo)(struct block_device *, struct hd_geometry *);
    int (*set_read_only)(struct block_device *bdev, bool ro);
    /* this callback is with swap_lock and sometimes page table lock held */
    void (*swap_slot_free_notify) (struct block_device *, unsigned long);
    int (*report_zones)(struct gendisk *, sector_t sector,
            unsigned int nr_zones, report_zones_cb cb, void *data);
    char *(*devnode)(struct gendisk *disk, umode_t *mode);
    struct module *owner;
    const struct pr_ops *pr_ops;
};
```

我们看几个较为重要的成员：

- submit_bio：该成员是块设备通用 IO 层之上用来将 bio 提交到通用 IO 层，如下图所示，接口映射层调用 submit_bio 来将 bio 请求转发给 generic block layer 层。

图 2-5 submit_io 转发 bio

- open：和 file_operations 中的 open 一样，实现设备的打开。
- release：和 file_operations 中的 release 一样，实现设备的关闭。
- ioctl：和 file_operations 中的 ioctl[5]一样，实现设备的命令操作，大部分命令内核已经实现，驱动开发人员只需要实现一小部分的命令即可。
- check_events：检测插拔事件。
- getgeo：获取磁盘属性，例如柱头（磁头）、柱面、扇区大小等。
- owner：一般初始化为 THIS_MODULE

2.4.3 bio

该结构体目的是对块设备的属性进行描述，例如一个机械硬盘，其属性有：存储容量、磁盘数量、柱面、柱头、扇区大小、磁道数量等。其具体结构体如下所示：

```
struct bio {
    struct bio      *bi_next;   /* request queue link */
    struct block_device *bi_bdev;
    unsigned int        bi_opf; /* bottom bits req flags,* top bits REQ_OP.Use
                                 * accessors.*/
    unsigned short      bi_flags;  /* BIO_* below */
    unsigned short      bi_ioprio;
    unsigned short      bi_write_hint;
    blk_status_t        bi_status;
    atomic_t        __bi_remaining;
    struct bvec_iter    bi_iter;
    bio_end_io_t        *bi_end_io;
    void            *bi_private;
#ifdef CONFIG_BLK_CGROUP
    /*
     * Represents the association of the css and request_queue for the bio.
     * If a bio goes direct to device, it will not have a blkg as it will
     * not have a request_queue associated with it.  The reference is put
     * on release of the bio.
     */
    struct blkcg_gq     *bi_blkg;
    struct bio_issue    bi_issue;
#ifdef CONFIG_BLK_CGROUP_IOCOST
    u64         bi_iocost_cost;
#endif
#endif
#ifdef CONFIG_BLK_INLINE_ENCRYPTION
    struct bio_crypt_ctx    *bi_crypt_context;
#endif
```

[5] 在 Linux 2.6 版本之后，file_operations 中的 ioctl 已经删除不再使用，而是使用 locked ioctl 替代。

```
    union {
#if defined(CONFIG_BLK_DEV_INTEGRITY)
        struct bio_integrity_payload *bi_integrity; /* data integrity */
#endif
    };
    unsigned short      bi_vcnt;    /* how many bio_vec's */
    /*
     * Everything starting with bi_max_vecs will be preserved by bio_reset()
     */
    unsigned short      bi_max_vecs;    /* max bvl_vecs we can hold */
    atomic_t        __bi_cnt;   /* pin count */
    struct bio_vec      *bi_io_vec; /* the actual vec list */
    struct bio_set      *bi_pool;
    /*
     * We can inline a number of vecs at the end of the bio, to avoid
     * double allocations for a small number of bio_vecs. This member
     * MUST obviously be kept at the very end of the bio.
     */
    struct bio_vec      bi_inline_vecs[];
};
```

我们来看些这个结构体中比较重要的成员变量，如下：

- bi_next：指向写一个 struct bio 结构体。
- bi_bdev：记录该 bio 所关联的块设备。
- bi_opf：操作标志，末尾 bit 表示读写标志，开始 bit 表示优先级标志。
- bi_iter：bio 迭代器，保存着要写入或者读取的信息，例如扇区地址、数据长度等。
- bi_private：指向 bio 私有数据
- bi_vcnt：bio 容器的数量
- bi_io_vec：bio 容器，记录着要写入的数据的页的物理地址、长度、偏移地址等。
- bi_inline_vecs：访问该结构体成员的零长度数组。

注：bi_inline_vecs 是一个 0 长度的数组变量，该变量不占用内存空间，这个是 gnu C 的语法特性，标准的 C 不支持该特性，在 Linux 中用的比较多。例如，如果我们需要记录一个字符串，但该字符串没有指定长度，那么我们也许会这样写：

```
struct string
{
        int length;
        char data[100];
}
```

如果我们的字符串小于 100，那势必会浪费空间，因此我们可以这样定义：

```
struct string
{
        int length;
        char *data;
}
```

这样我们使用动态分配内存来实现，但是这样 data 就会占用 4 个字节的空间，但如果使用零长度数

组，这样就不会浪费这个 4 个字节了，如下：

```
struct string
{
        int length;
        char data[0];
}
```

为了更加清除的看到 bio 的结构，笔者将该结构绘制成图的形式，如下所示：

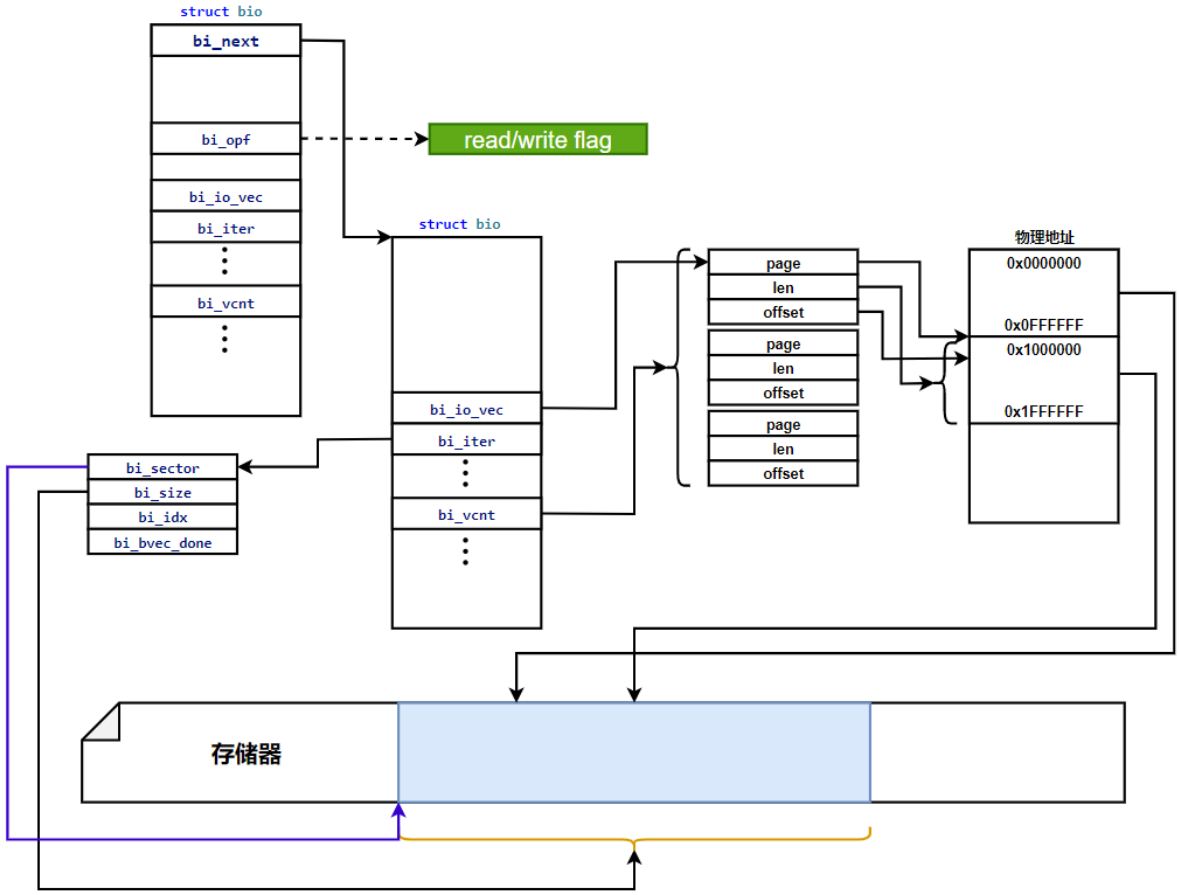


图 2-6 bio 结构图

2.4.5 request_queue

该结构体较为庞大，这里省略其他不相关的成员，其具体结构如下所示：

```
struct request_queue {
    /* 省略无关成员 */
    struct request          *last_merge;

    struct elevator_queue   *elevator;

    struct percpu_ref   q_usage_counter;
    const struct blk_mq_ops *mq_ops;
```

```
    unsigned long       nr_requests;    /* Max # of requests */

    struct blk_flush_queue  *fq;

    struct list_head    requeue_list;
    spinlock_t      requeue_lock;
    /* 省略无关成员 */
};
```

其中主要的成员说明如下：

- last_merge：指向最先可能合并的 queue
- elevator：指向电梯算法
- q_usage_counter：记录 queue 的使用情况
- mq_ops：request 的底层的操作方法
- nr_requests：queue 中 request 的最大个数
- fq：指向 flush 操作，即强制将缓存数据写入到硬盘中
- requeue_list：指向 request 的链表头
- requeue_lock：指向 request 的自旋锁

2.4.4 request

该结构体是块设备驱动的读写操作的抽象集合，其中包含了 bio 结构体，块设备驱动中的所有读写都有 request 来完成，可以看出该结构体的重要性。下面是该结构体的具体成员：

```
struct request {
        /* 省略无关成员 */
    struct request_queue *q;

    /* the following two fields are internal, NEVER access directly */
    unsigned int __data_len;    /* total data len */
    sector_t __sector;      /* sector cursor */

    struct bio *bio;
    struct bio *biotail;
    struct list_head queuelist;
    struct gendisk *rq_disk;
    struct block_device *part;
    /* 省略无关成员 */
};
```

其成员说明如下：

- q：指向该 request 的 request_queue
- __data_len：记录整个数据的长度
- __sector：记录当前扇区的位置

- bio：指向 bio 结构体
- biotail：指向 bio 结构体的
- queuelist：指向下一个 request，该结构是一个链表结构
- rq_disk：指向 gendisk，即要操作的磁盘设备
- part：指向分区位置

2.5 请求队列的结构图

上面我们对块设备驱动中涉及到的结构体进行了简单讲解，但这些结构体都比较复杂，下面我们对这个结构体联系起来，其整体结构图如下所示：

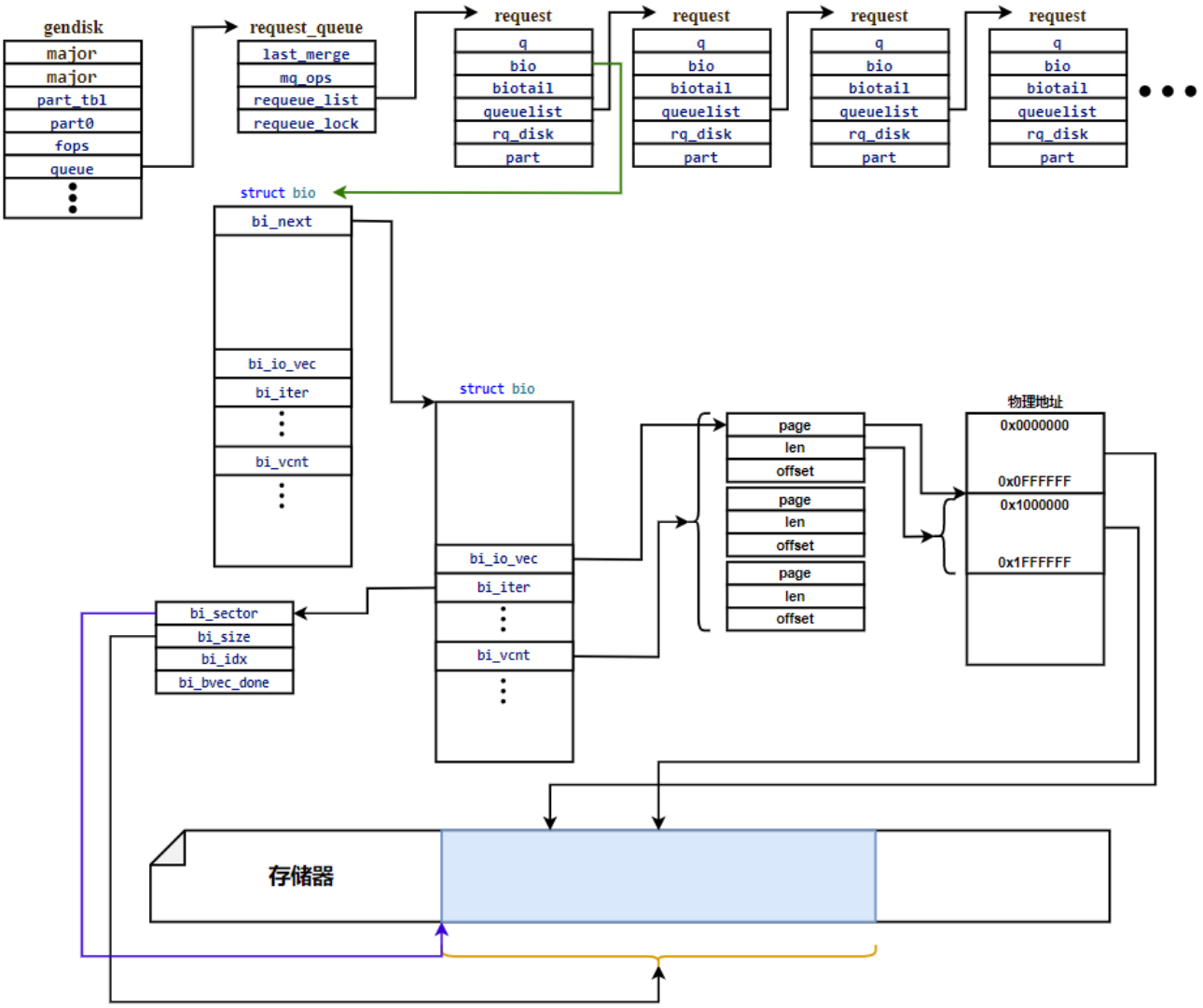


图 2-7 使用请求队列的块设备驱动结构图

上面的结构中会涉及到三个概念，即扇区(sector)、块(Block)、段(Segment)，其中扇区就是在第一章节中提到的，大部分块设备的一个扇区规定为 512Byte，块由多个扇区组成，而段由多个块组成，其结构如下图所示。对于大部分块设备而言，一个 page 的大小为 4Kbyte，一个 segment 的大小为 2Kbyte，一个 block 的大小为 1Kbyte，一个 sector 的大小为 512Byte。在 Linux

中，划分为 Segment 的主要原因是 DMA 的通道数据宽度大小与段的大小相等，这样传输数据就可以利用 DMA 来实现数据的读写，而不用 CPU 参与。在 Linux 中，page 的大小由一个宏来表示，这个宏就是 PAGE_SIZE，该值一般为 4KByte。需要注意的是，这里的 page 是一个虚拟的概念，与实际的介质没有关系。

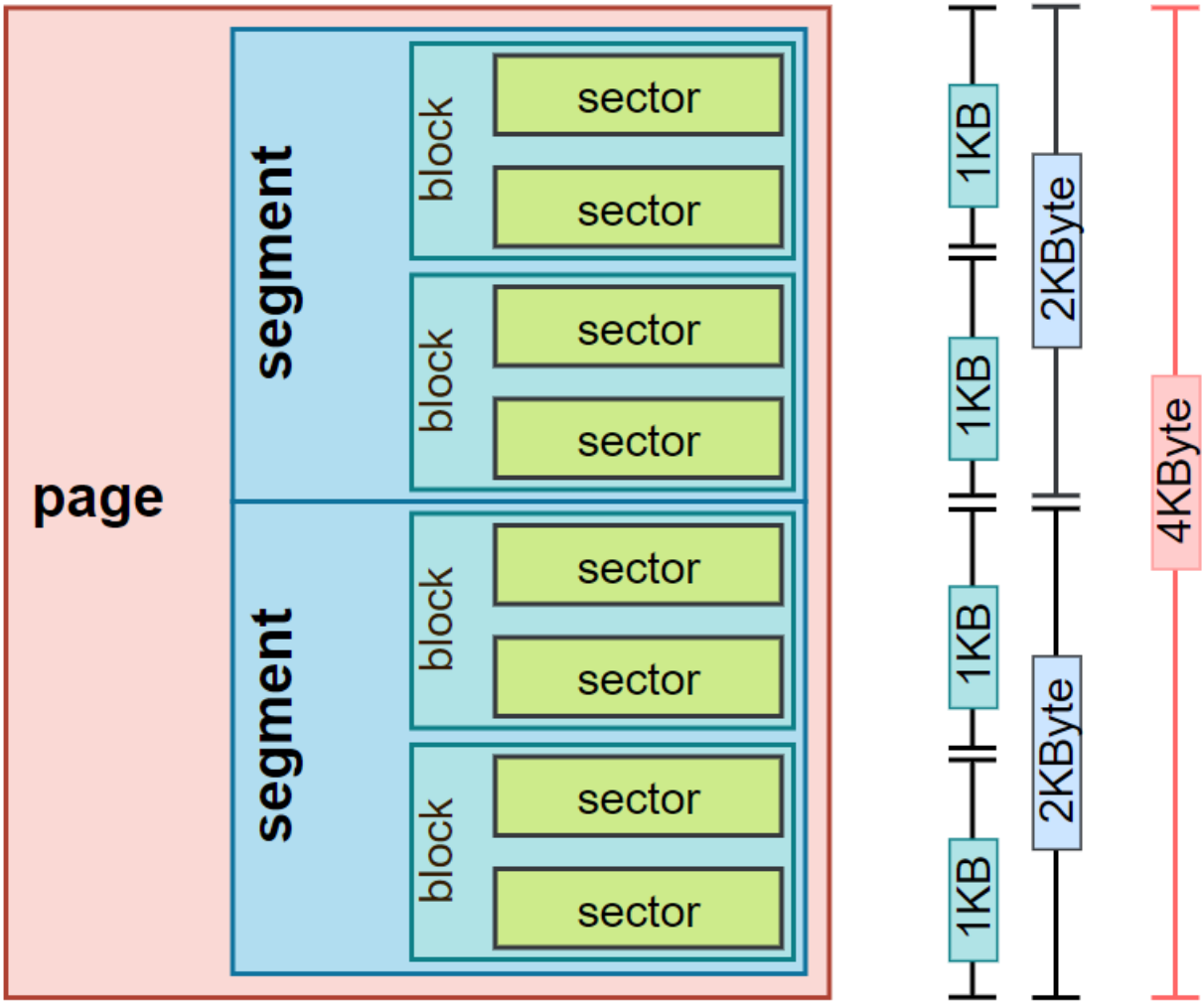


图 2-8 page、segment、block、sector 大小说明

2.5 无请求队列的结构图（续）

读者需要注意的是，由于引入请求队列的目的是使用特定算法能够将多个 request 合并，这样对于多个离散的地址读写时，可以将这些离散的地址按地址的大小顺序重新排列，这样的好处就是对于机械硬盘来说，其读写速度将有非常大的提高。然而技术的发展，当前固态硬盘已经开始逐渐取代机械硬盘了，这样引入请求队列的机制将对硬盘的读写没有任何改善，反而由于引入请求导致其路径变长而速度下降。因此在 Linux 中，也同样支持没有请求队列的驱动框架。结构图如下所示，可以看到，在不使用请求队列时，此时 gendisk 中的 fops 中的 submit_bio 函数直接将 bio 进行处理，并没有将 bio 用 queue 保存起来。

这种没有使用 request queue 的块设备驱动通常在 Flash 存储器中用的非常多，例如 SSD、U 盘、UFS、NVMe 等。当不适用请求队列时，我们就不需要分配 request queue 了，当然也就

无需去实现请求处理了，我们只需要实现 submit_bio 函数指针的实例。

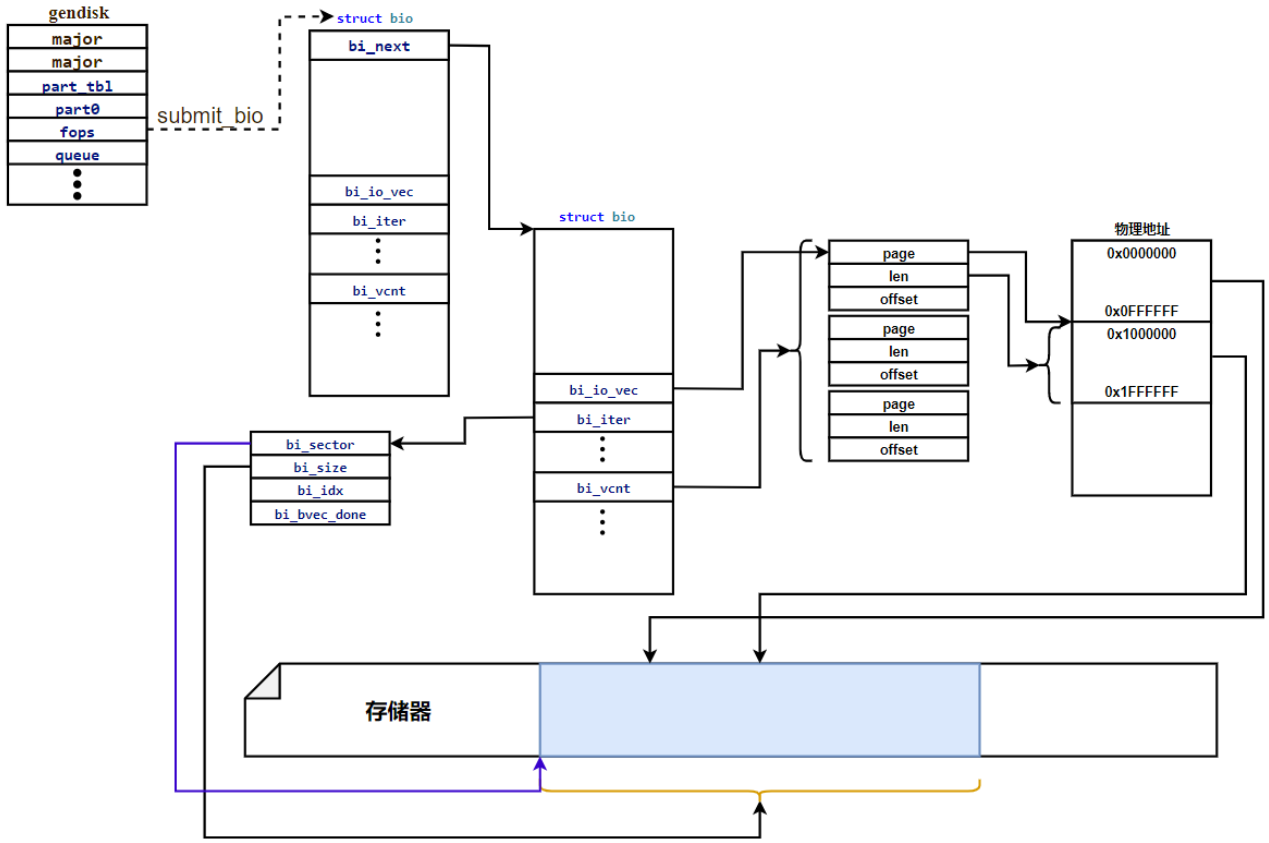


图 2-8 不使用请求队列的块设备驱动结构图

2.6 块设备驱动注册

上面我们对各个重要的结构体进行了简单描述，现在我们对块设备驱动的注册进行说明，即如何在 Linux 中注册一个块设备驱动。创建一个块设备驱动的步骤如下：

1. 创建一个块设备
2. 分配一个申请队列
3. 分配一个 gendisk 结构体
4. 设置 gendisk 结构体的成员
5. 注册 gendisk 结构体

下面我们首先对块设备驱动中用到的常用 API 进行介绍。

2.6.1 块设备 API

2.6.1.1 分配一个块设备结构体

在 Linux 中，结构体可以用定义一个结构体，由编译器来分配内存，还有一种就是使用 Linux 提供的动态分配方式。但对于块设备驱动而言，Linux 中只能用动态方式分配，下面是

Linux 内核提供了块设备结构体的分配函数：

```
struct gendisk *alloc_disk(int minors)
```

其中 minors 为分区个数，如果不分区，则填写 1 即可。

2.6.1.2 删除一个块设备结构体

当我们的块设备卸载时，我们需要删除该块设备结构体，即释放内存资源，Linux 内核提供了如下 API 供驱动开发者使用：

```
void del_gendisk(struct gendisk *gp)
```

该函数的参数为 gendisk，即需要删除的块设备结构体。

2.6.1.3 注册块设备驱动

```
int register_blkdev(unsigned int major, const char *name)
```

该函数实现对一个块设备的注册，其中第一个参数 major 为主设备号，第二个参数 name 为设备的名称。如果 major 为 0，则表示动态分配一个主设备号。注册成功返回主设备号值，否则返回负数。

2.6.1.4 注销块设备驱动

```
void unregister_blkdev(unsigned int major, const char *name)
```

该函数实现对一个块设备进行注销，其中第一个参数 major 为主设备号，第二个参数 name 为设备名称。

2.6.1.5 将 gendisk 添加到内核

```
inline void add_disk(struct gendisk *disk)
```

该函数将一个 gendisk 添加到内核中，其中参数为 gandisk 结构体。需要注意的是，在添加 gendisk 之前必须对 gendisk 结构体中的成员进行初始化。

2.6.1.6 将 gendisk 从内核中删除

```
void put_disk(struct gendisk *disk)
```

该函数将一个 gendisk 从内核中删除，其中参数为 gandisk 结构体。

2.6.1.7 设置 gendisk 的扇区数

```
void set_capacity(struct gendisk *disk, sector_t sectors)
```

该函数用来设置 gendisk 的扇区数，即硬盘容量，在分配 gendisk 之后，我们需要用此函数来设置 gendisk 的扇区数。

2.6.1.8 分配一个请求队列6

```
struct request_queue *blk_mq_init_queue(struct blk_mq_tag_set *set)
```

该函数用来申请一个请求队列同时初始化，即 request_queue，该请求队列用来容纳 request。其参数为 blk_mq_tag_set 即用来存储器件相关的 tag 集合。

2.6.1.9 清除请求队列中的所有请求

```
void blk_cleanup_queue(struct request_queue *q)
```

该函数用于清除 request_queue 中的所有 request，当 gendisk 释放后，内核中可能还存在没有处理完的请求，因此我们在删除块设备之前，必须保证内核中没有请求了。

2.7 一个简单的示例

上面我们讲解了块设备驱动的基本框架和步骤，现在我们以一个虚拟的块设备驱动来具体说明开发过程。

2.7.1 不使用请求队列

由于现在大部分硬盘都是固态硬盘，因此这里我们先以一个不使用请求队列的驱动来讲解。首先是块设备驱动的分配和注册，然后最重要的是实现 submit_bio 函数指针的实例。为了方便我们讲解块设备驱动，这里以 RAM 作为存储介质，即使用内存来实现块设备的数据存储，这样读写操作就变得比较方便。

和字符设备驱动一样，我们这里为了让结构变得简单，使用最简单的驱动框架，而不使用 platform 框架，实际开发的时候最好使用 platform 驱动框架。在__init 函数需要实现块设备驱动的注册，其中包括块设备的设备号分配、设备注册、设置扇区大小等。代码如下所示：

```
static int __init ram_blk_init(void)
{
    /* 注册一个块设备，返回值为主设备号 */
    ram_blk_major = register_blkdev(0, "ram_blk")
    if ( ram_blk_major < 0) {
        printk(KERN_ERR"ram disk register failed!\n");
        return -1;
    }
    ram_gendisk = alloc_disk(1);//分配一个 gendisk
```

6 在之前的内核版本中，使用的是 blk_init_queue 函数，但在较新的内核中已经废除了该函数，这是因为之前的函数都是 single queue，之后由于全部使用 multi queue，之前的单队列就被废除了。

```
    ram_gendisk->major = ram_blk_major;//设置主设备号
    ram_gendisk->first_minor = 0;//设置第一个分区的次设备号
    ram_gendisk->minors = 1;//设置分区个数：1
    ram_gendisk->fops = &ram_blk_ops;//指定块设备 ops 集合
    set_capacity(ram_gendisk, 2048);//设置扇区数量：1MiB/512B=2048
    add_disk(ram_gendisk);//添加硬盘
    spin_lock_init(&ram_blk_lock);//初始化自旋锁
    ram_blk_addr = (char*)vmalloc(2048*512);//分配 1M 空间作为硬盘
    if(ram_blk_addr == NULL)
    {
        printk(KERN_ERR"alloc memory failed!\n");
    }
    return 0;
}
```

上面的函数较为简单，需要注意的是设置硬盘扇区数的时候我们以 512Byte 作为一个扇区，这样我们的 1MiB 的硬盘大小就有 2048 个扇区了。

```
static const struct block_device_operations ram_blk_ops = {
    .owner      = THIS_MODULE,
    .submit_bio = ram_blk_submit_bio,
    .open       = ram_blk_open,
    .release    = ram_blk_release,
};
```

我们再来上面函数中的 ram_blk_ops，这个是块设备的操作集合，其中包含了 open、release、submit_bio 等，open 和 release 函数如下所示：

```
static int ram_blk_open(struct block_device *bdev, fmode_t mode)
{
    return 0;
}

static void ram_blk_release(struct gendisk *disk, fmode_t mode)
{

}
```

可以看到，上面的这两个函数啥也没干，因为我们使用的是 RAM 来模拟硬盘的。我们再来看下 ram_blk_submit_bio 这个函数，这个函数实现了数据的读写，因此较为关键，我们这里对这个函数详细讲解。

该函数是块设备驱动最终要执行的读写操作，首先应用层通过读写文件来实现对介质的读写，这个过程需要调用 C 库，然后 C 库会通过系统调用来进入实际的读写操作，之后会进入虚拟文件系统，然后映射到具体的文件系统上，这时内核的块设备框架会调用 submit 函数来实现数据的读写，实际上这个过程会有缓存，也就是数据不会立刻写入到块设备驱动，而是先放在缓存中，也就是内存中，当数据满足一定数量后，块设备驱动就会触发 submit 来提交 bio。

```
static blk_qc_t ram_blk_submit_bio(struct bio *bio)
{
    struct bio_vec bvec;
    struct bvec_iter iter;
    sector_t sector = bio->bi_iter.bi_sector;
    bio_for_each_segment(bvec, bio, iter) {
        char *buffer = kmap_atomic(bvec.bv_page) + bvec.bv_offset;
        unsigned len = bvec.bv_len >> SECTOR_SHIFT;

        ram_blk_transfer(sector, len, buffer,
                bio_data_dir(bio) == WRITE);
        sector += len;
        kunmap_atomic(buffer);
    }
    bio_endio(bio);
    return BLK_QC_T_NONE;
}
```

上面是一个虚拟块设备驱动的 submit 函数，该函数实际上做的事情就是遍历每个 bio 中的数据。将需要写入的数据写入到介质中（硬盘）。上面有一个宏 bio_for_each_segment，这个宏实际上就是一个 for 循环[7]。通过 bio_data_dir(bio)来获取 BIO 的读写属性，可以看到，每个 bio 中一定保存这读写的标志位。kmap_atomic(bvec.bv_page)是将保存在 page 中的数据映射到永久映射区中，为何这里需要将 bv_page 映射到这个区域呢？这是因为 bvec.bv_page 记录的数据在用户空间的地址，内核空间要想访问用户空间，则必须将用户空间映射到内核空间中。由于我们每次读写的数据都不是持续的（写入完毕就无效了），因此我们可以利用 kmap_atomic 函数来实现临时的内存映射。

ram_blk_transfer 函数将数据写入到实际的介质中，该函数的定义如下：

```
static void ram_blk_transfer(unsigned long sector,unsigned long nsect, char
*buffer, int write)
{
    unsigned long offset = sector << SECTOR_SHIFT;
    unsigned long nbytes = nsect << SECTOR_SHIFT;

    spin_lock(&ram_blk_lock);
    unsigned long io;
    ram_blk_sursor = ram_blk_addr;
    ram_blk_sursor += offset;
    READ_ONCE(*buffer);
    if (write)
        io = memcpy(ram_blk_sursor,buffer,nbytes);
    else
        io = memcpy(buffer,ram_blk_sursor,nbytes);
```

[7] 具体可参考文件：include/linux/bio.h

```
        if (io == -1) {
            printk(KERN_ERR"ram disk failed\n");
        }
        spin_unlock(&ram_blk_lock);
    }
```

上面的过程较为简单，也就是将数据写入到介质中，这里我们写入的是 RAM 中。最后我们需要实现出口函数，也就是 exit 函数，函数定义如下：

```
static void __exit ram_blk_exit(void)
{
    del_gendisk(ram_gendisk);//删除硬盘
    blk_cleanup_disk(ram_gendisk);//删除所有未完成的请求
    unregister_blkdev(ram_blk_major, "ram_blk");//注销块设备
}
```

上面的操作较为简单，硬盘卸载后需要删除硬盘设备号，同时删除未完成了 bio，然后注销块设备即可。

2.7.1.1 Linux 5.9 版本之前与之后区别

Linux 5.9 中提交了一个 patch，该 patch 将 blk_init_queue 函数删除，同时将在 block_device_operations 结构体中添加了 submit_bio 函数指针，此后用户将不需要使用 blk_init_queue 函数来预先分配一个请求队列，然后再实现一个 make_request 函数。现在我们只需要实现 submit_bio 函数即可，也就是我们上面例子中讲到的对 request 进行遍历所有的 bio。

2.7.1.2 源码

注意：本源码的内核版本为 Linux 5.14 版本。

```
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/string.h>
#include <linux/blkdev.h>
#include <linux/bio.h>
#include <linux/proc_fs.h>
#include <linux/uaccess.h>
#include <linux/hdreg.h>

int ram_blk_major;
struct gendisk * ram_gendisk;
spinlock_t ram_blk_lock;
char *ram_blk_addr = NULL;
char *ram_blk_sursor=NULL;//定义一个写入游标
```

```
static void ram_blk_transfer(unsigned long sector,
        unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector << SECTOR_SHIFT;
    unsigned long nbytes = nsect << SECTOR_SHIFT;

    spin_lock(&ram_blk_lock);
    unsigned long io;
    ram_blk_sursor = ram_blk_addr;
    ram_blk_sursor += offset;
    READ_ONCE(*buffer);
    if (write)
        io = memcpy(ram_blk_sursor,buffer,nbytes);
    else
        io = memcpy(buffer,ram_blk_sursor,nbytes);
    if (io == -1) {
        printk(KERN_ERR"ram disk failed\n");
    }
    spin_unlock(&ram_blk_lock);
}

static blk_qc_t ram_blk_submit_bio(struct bio *bio)
{
    struct bio_vec bvec;
    struct bvec_iter iter;
    sector_t sector = bio->bi_iter.bi_sector;
    bio_for_each_segment(bvec, bio, iter) {
        char *buffer = kmap_atomic(bvec.bv_page) + bvec.bv_offset;
        unsigned len = bvec.bv_len >> SECTOR_SHIFT;

        ram_blk_transfer(sector, len, buffer,
                bio_data_dir(bio) == WRITE);
        sector += len;
        kunmap_atomic(buffer);
    }
    bio_endio(bio);
    return BLK_QC_T_NONE;
}

static int ram_blk_open(struct block_device *bdev, fmode_t mode)
{
    return 0;
}
```

```
static void ram_blk_release(struct gendisk *disk, fmode_t mode)
{

}

static int ram_blk_getgeo(struct block_device *bdev, struct hd_geometry *geo)
{

    geo->heads = 1;
    geo->sectors = get_capacity(ram_gendisk);
    geo->cylinders = 1;
    return 0;
}

static int ram_blk_ioctl(struct block_device *bdev,fmode_t mode,unsigned int
cmd,unsigned long arg)
{

    return 0;
}

static const struct block_device_operations ram_blk_ops = {
    .owner      = THIS_MODULE,
    .submit_bio = ram_blk_submit_bio,
    .open       = ram_blk_open,
    .release    = ram_blk_release,
    .getgeo     = ram_blk_getgeo,
    .ioctl      = ram_blk_ioctl,
};

static int __init ram_blk_init(void)
{
    /* 注册一个块设备，返回值为主设备号 */
    ram_blk_major = register_blkdev(0, "ram_blk");
    if ( ram_blk_major < 0)
    {
        printk(KERN_ERR"ram disk register failed!\n");
        return -1;
    }
    ram_gendisk = blk_alloc_disk(NUMA_NO_NODE);//分配一个 gendisk
    if(!ram_gendisk)
    {
        printk(KERN_ERR"alloc_disk failed!\n");
```

```
        return -1;
    }
    strcpy(ram_gendisk->disk_name, "ram_blk");
    ram_gendisk->major = ram_blk_major;//设置主设备号
    ram_gendisk->first_minor = 0;//设置第一个分区的次设备号
    ram_gendisk->minors = 1;//设置分区个数：1
    ram_gendisk->fops = &ram_blk_ops;//指定块设备 ops 集合
    set_capacity(ram_gendisk, 20480); //设置扇区数量：10MiB/512B=20480
    add_disk(ram_gendisk);//添加硬盘
    spin_lock_init(&ram_blk_lock);  //初始化自旋锁
    ram_blk_addr = (char*)vmalloc(10*2048*512); //分配 10M 空间作为硬盘
    if(ram_blk_addr == NULL)
    {
        printk(KERN_ERR"alloc memory failed!\n");
        return -1;
    }
    ram_blk_sursor = ram_blk_addr;
    return 0;
}

static void __exit ram_blk_exit(void)
{
    del_gendisk(ram_gendisk);//删除硬盘
    blk_cleanup_disk(ram_gendisk);//删除所有未完成的请求
    unregister_blkdev(ram_blk_major, "ram_blk");//注销块设备
}

module_init(ram_blk_init);
module_exit(ram_blk_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("1477153217@qq.com");     //作者
MODULE_VERSION("0.1");          //版本
MODULE_DESCRIPTION("led_dev");  //简单的描述
```

2.7.1.3 测试

将上面的源码编译为.ko 文件，然后拷贝到开发板上的根文件系统中，使用 insmod 命令挂载驱动，此时可以看到在/dev 目录下生成一个 ram_blk 设备文件节点。现在我们使用 mkfs.ext4 工具对该文件进行文件系统格式化，然后使用 mount 命令挂载该文件系统。

现在我们使用 insmod 挂载编译好的.ko 文件，如下：

```
# insmod ram_blk.ko
```

```
[ 1103.578083] ram_blk: loading out-of-tree module taints kernel.
```

此时我们可以在/dev 目录下看到生成了一个 ram_blk 设备节点文件，如下：

```
ptyq5          ram_blk          ttyeb          ttyy6
ptyq6          random           ttyec          ttyy7
```

现在我们对该设备节点（虚拟硬盘）使用 fdisk 工具进行分区，现在我们输入 n，表示新建分区，然后输入 p，表示主分区，再输入 1，表示新建一个分区，然后输入 2，表示从 2 扇区（即 1Mbyte 地址处）开始，再输入 325，表示在 325 扇区结束。最后输入 w，表示将分区表写入到磁盘中，如下：

```
# fdisk /dev/ram_blk
Device contains neither a valid DOS partition table, nor Sun, SGI, OSF or GPT disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.


Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (2-325, default 2): 2
Last cylinder or +size or +sizeM or +sizeK (2-325, default 325): 325

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
fdisk: WARNING: rereading partition table failed, kernel still uses old table: Invalid argument
```

至此，我们的分区表已经建立了，现在我们还需要使用 mkfs 工具对该硬盘进行格式化，即格式化一个文件系统，如下所示：

```
 # mkfs.ext4    /dev/ram_blk
mke2fs 1.43.9 (8-Feb-2018)
Found a dos partition table in /dev/ram_blk
Proceed anyway? (y,N) y
Creating filesystem with 10240 1k blocks and 2560 inodes
Filesystem UUID: 75aaa0e6-ffaa-4a2d-ba06-26986acf00f3
Superblock backups stored on blocks: 8193
Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

现在我们的虚拟硬盘已经被格式化为 ext4 格式了，此时我们使用 mount 命令将其挂载在/mnt 目录下，如下：

```
# mount /dev/ram_blk    /mnt/

[    130.913328] EXT4-fs (ram_blk): mounted filesystem with ordered data mode. Opts: (null). Quota mode: disabled.
```

可以看到，我们的虚拟硬盘已经挂载到/mnt 目录下了，现在我们进入 mnt 目录，然后新建一个文件，如下所示：

```
# cd /mnt/
# ls
lost+found
# touch dummy.txt
# ls
dummy.txt      lost+found
#
```

可以看到，我们已经可以在该虚拟硬盘中新建文件夹了，现在我们可以使用 fdisk -l 命令来查看下系统中的所有硬盘信息：

2.7.2 使用请求队列

在 Linux 中，块设备驱动的写入分为两种，一种就是上面的不使用请求队列的方式，这种方式每发起一个 bio 请求就会直接写入到介质中。另一种就是下面我们将要举例说明的使用请求方式，为何要使用队列呢？这是由于在 Flash 存储器广泛使用之前，HDD（固态硬盘）占据绝对统治地位，但固态硬盘有一个特点，就是顺序写入和读取的效率最高（因为读写靠磁头临近磁盘，磁盘旋转）。这样每次 bio 请求不能直接写入到介质中，而是先按地址的先后顺序排列之后，再写入，这样就可以大大提到读写速度（这便是赫赫有名的电梯算法）。当然，目前 HDD 已经开始退出观众视野，大部分的嵌入式设备也不会出现 HHD，但作为一种 Linux 的重要的框架，还是有必要对其简单了解下。

2.7.2.1 版本变化导致的 API 变化

2.8 总结

可以看到，块设备驱动要比字符设备驱动复杂的多，这主要是因为块设备驱动牵扯到的东西非常多且杂。而字符设备驱动相对要简单很多，在字符设备中，整个框架相对简单且薄，这是因为字符设备驱动都是单个单个写入的，而且由于字符设备文件太杂乱，不可能将很多东西

都抽象出来，大部分需要用户来自己完成。但对于块设备驱动而言，其种类要少的多，当前无非就是 HD，SSD，EMMC 等，这些设备都是有严格的协议规范，因此 Linux 内核为了让块设备驱动开发变得更加简单，引入了大量的子系统，这也就是我们下面将会介绍的各种块设备子系统。

同样，我们以一个 RAM 存储方式来实现一个虚拟的块设备驱动，这次我们使用请求队列的方式实现。

第三章 MMC 子系统

MMC 即 Multi Media Card，多媒体卡。随着 Flash 存储技术的不断发展，市面上出现了诸多种类的 NAND Flsh 存储卡，其中较为常见的便是 mmc 卡、SD 卡以及 TF 卡。其中 mmc 是较早的一种，随后出现了 SD 卡，为了保持与之前的卡兼容，SD 的协议一般都支持 mmc，因此我们有时候也会将 SD 卡和 mmc 卡不加区分。

图 3-1 mmc 卡、SD 卡、TF 卡

Linux 内核为块设备驱动做了大量了工作，驱动工程师几乎可以不用关心如何去实现的，但对于一些芯片厂家而言，他们需要知道 MMC 的框架，因为他们的 BSP 工程师需要为他们的产品做底层主机的驱动开发。

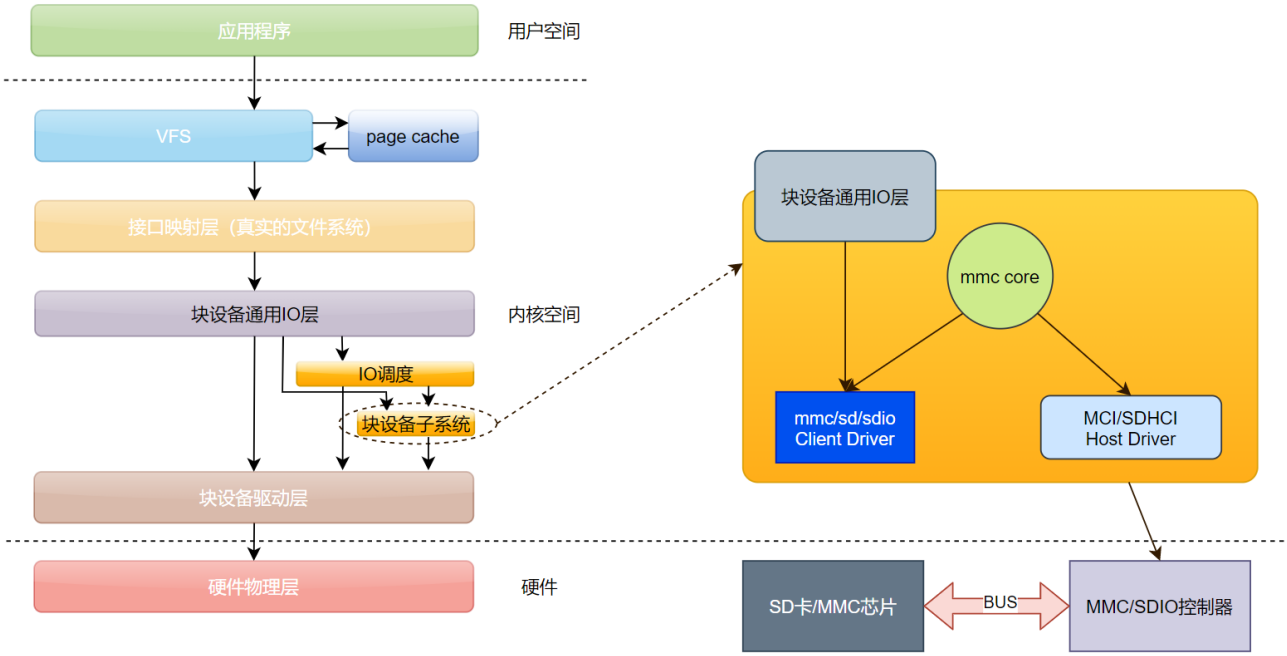


图 3-2 mmc 子系统框架图

上图是 MMC 子系统的框架图，MMC 子系统属于块设备驱动子系统之一，从图中可以看到，块设备驱动子系统分为三个部分，分别是：Client Driver、MMC Core、Host Driver。其中 Client Driver 表示卡的驱动，该驱动也称为 Card Driver 层，该层就是我们所说的 MMC 驱动层。Host Driver 表示控制器驱动，每个芯片所使用的 MMC 控制器都不一样，这主要表现为其寄存器、总线以及中断号都不相同，因此主机控制器驱动一般由芯片原厂的 BSP 工程师来完

成驱动的开发，我们也称其为 MMC 的 BSP 驱动。MMC Core 是 Linux 中 mmc 子系统的核心，该部分由 Linux 内核开发者来完成，这部分是不需要驱动工程师开发的。对于 MMC 而言，全世界的厂家就那么多，而且都遵循严格的协议，因此 Linux 内核已经实现了几乎所有的 MMC 驱动，对于驱动工程师只需要做非常小的改动（修改一些时序参数）甚至不需要做任何修改就可以直接使用。MMC 子系统中唯一需要开发的就是 Host Driver 驱动，该部分也是驱动工程师修改最多的地方。

3.1 SDIO 协议分析

SDIO（Secure Digital Input and Output），即安全数字输入输出接口。随着 SDIO 的不断发展，其不仅仅用于 SD 卡中，还广泛应用于其他芯片之间的通信，例如 WIFI 芯片。SDIO 版本有很多，目前最新的版本为 4.2 版本，协议是向前兼容的。

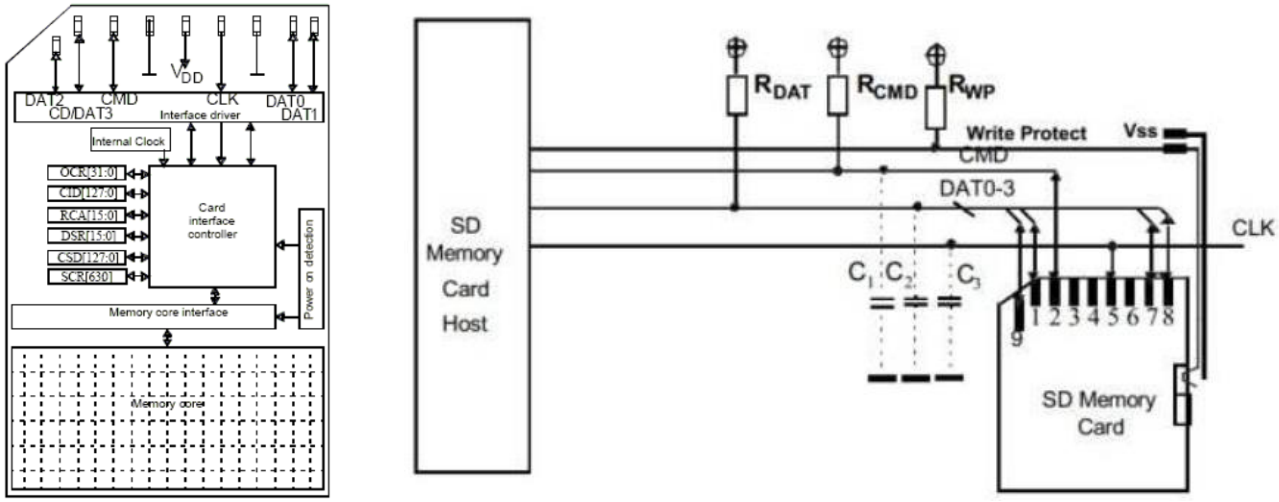


图 3-3 SD 卡内部结构和电路连接图

废话不多说了，我们现在重点来看下 SDIO 的协议部分。SDIO 协议传输过程中有三种传输类型，分别是命令传输（Command）、响应传输（Response）、数据传输（Data），我们来分别看下这几个传输过程。

3.1.1 数据传输类型

3.1.1.1 无响应无数据传输

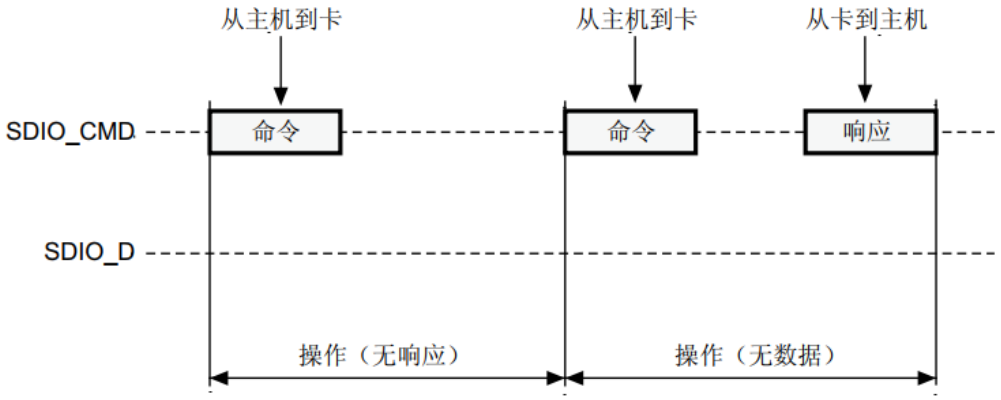


图 3-4 无响应无数据传输

该类数据传输没有响应或者没有数据传输，如上图所示，当主机（SDIO 控制器）发出命令，设备（SD 卡）无数据回应；当主机发送命令时，没有数据传输。

3.1.1.2 多个块读取操作

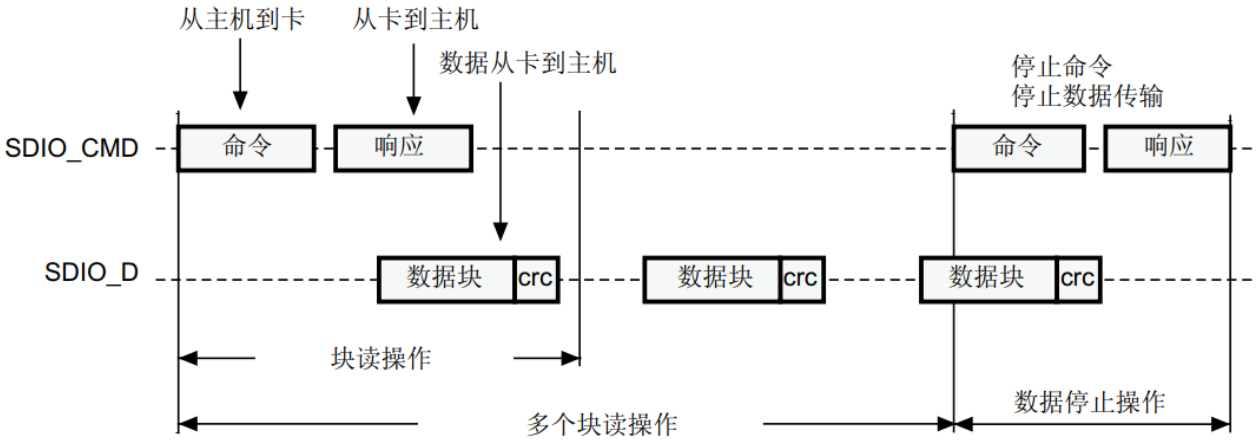


图 3-5 多个块读取操作

该类数据传输对设备的多个块进行读取，首先，主机会发送令牌，然后设备做出回应，主机收到回应后，准备好内存，随后设备开始传输多个块数据给主机，重复次操作，最后主机发送停止命令结束传输。

3.1.1.3 多个块写入操作

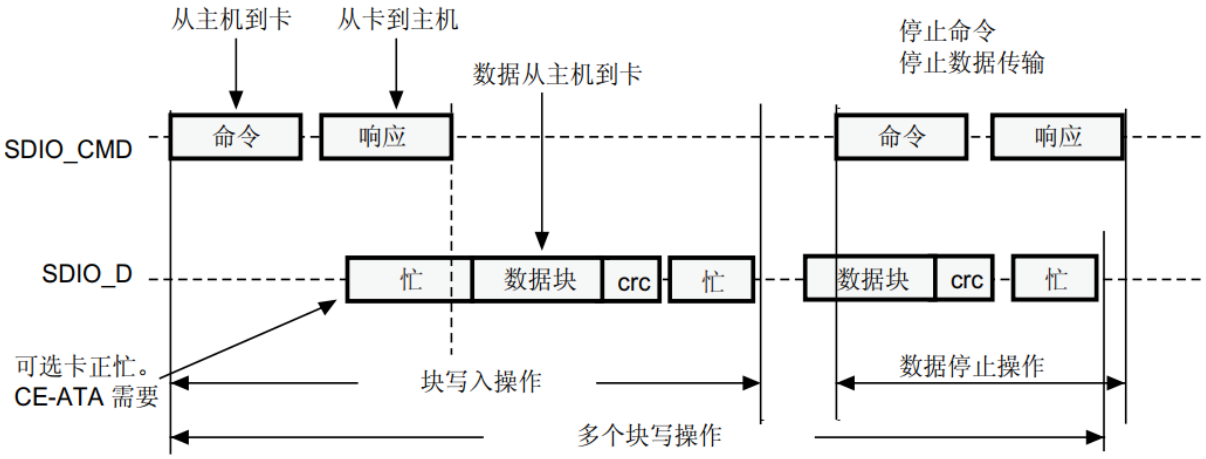


图 3-6 多个块写入操作

和多个块读取一样，当主机需要对设备进行多个块写入数据时，主机需要发送令牌包，然后设备做出回应，此时当主机检测到设备不忙时，此时主机开始发送多个块数据，重复次操作，最后主机发送停止命令结束传输。

3.1.1.4 连续读取操作

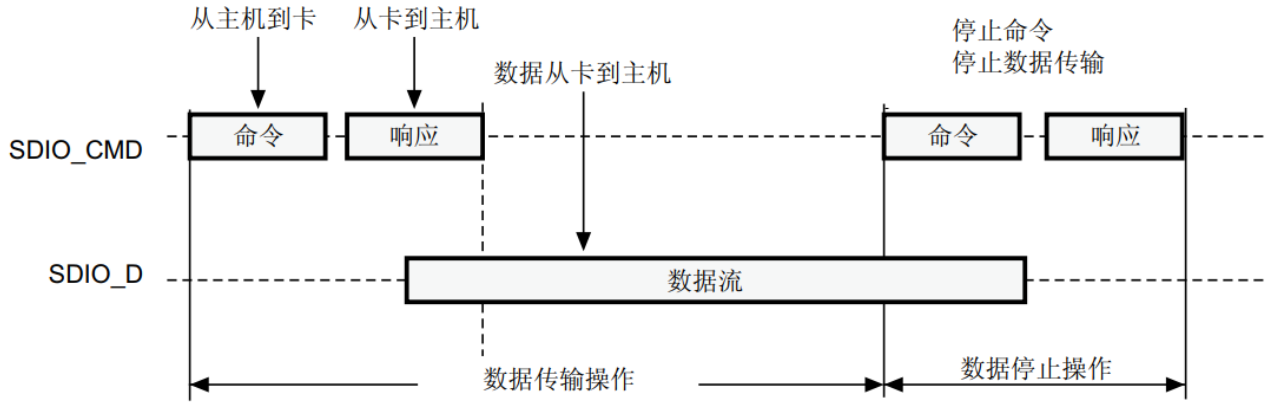


图 3-7 连续读取操作

当主机需要对设备进行连续的数据流读取时，此时主机只需要发送一个连续读取数据的令牌包，然后设备会给出回应，此时主机准备好内存，然后设备开始返回数据流给主机，最后主机发送停止令牌来结束数据传输。

3.1.1.5 连续写入操作

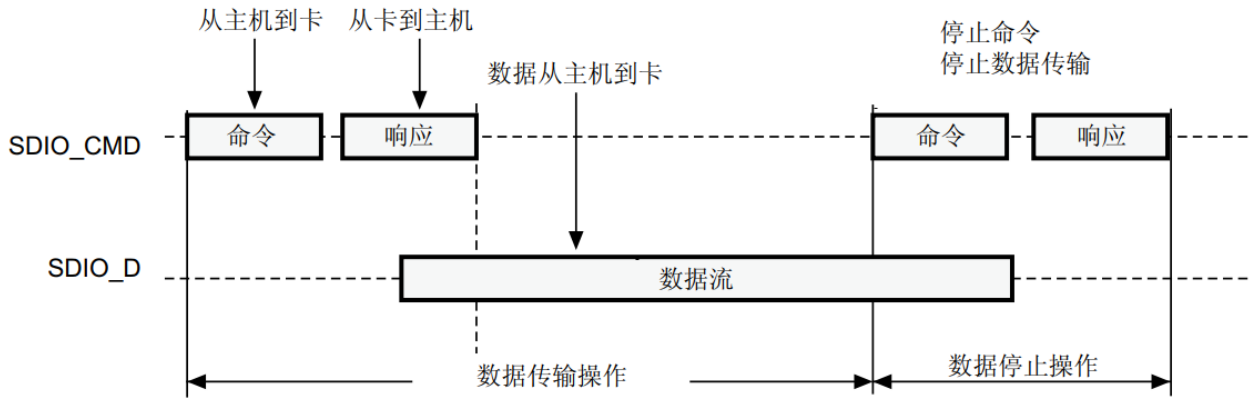


图 3-8 连续写入操作

当主机需要对设备进行连续的数据流写入时，此时主机只需要发送一个连续写入的令牌包，然后设备给出回应，此时主机开始发送数据流给设备，最后主机发送停止令牌来结束数据传输。

可以看到，主机与设备之间的传输较为简单，大致上就是主机发送命令（令牌），然后设备给出回应，此时开始数据传输。

3.1.2 控制器说明

上面的所有数据传输不需要驱动开发者来实现，由于这些传输的协议都是规定好的，这些操作都由控制器来完成了，因此我们只需要对控制器进行控制即可，控制器会按我们要求的方式来传输数据，我们下面来简单了解下 SDIO 的控制器。

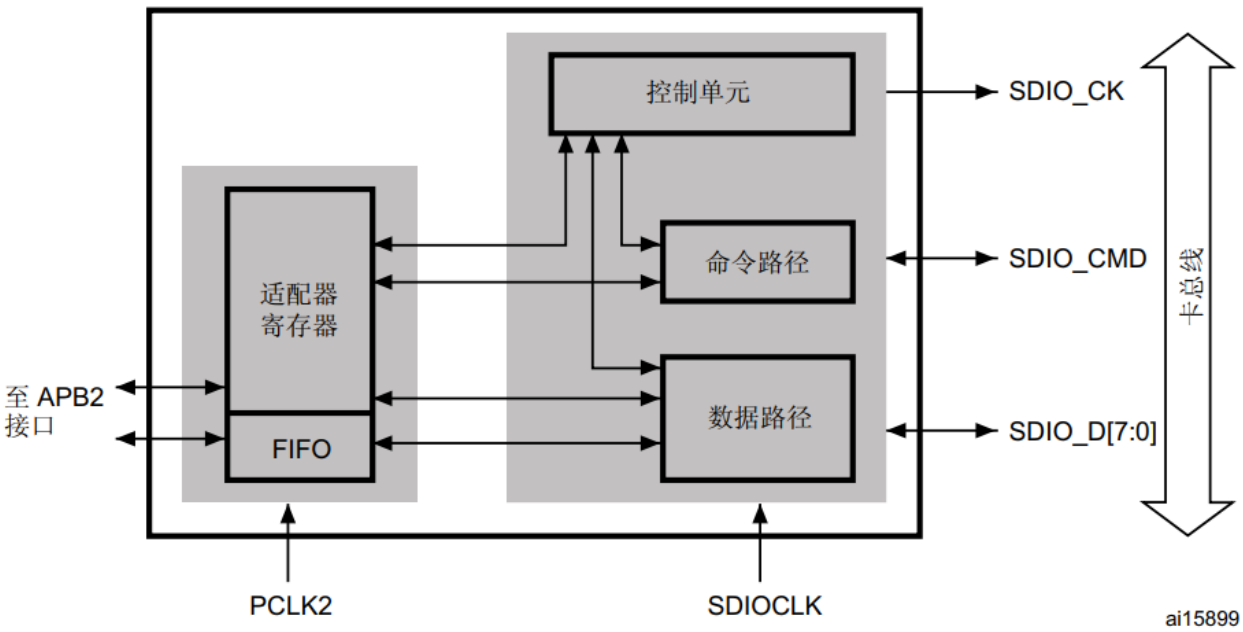


图 3-9 SDIO 控制器结构图

上图是典型的一个 SDIO 的控制器拓扑图，其主要包含三个部分：

- 适配器寄存器
- 命令路径
- 控制路径
- 控制单元

CPU 通过 AHP 或者 APB 总线访问适配器寄存器，适配器寄存器控制命令路径和数据路径，命令路径和数据路径与控制单元进行交互。当然寄存器也可以直接控制控制单元，适配器为数据路径提供了 FIFO 作为数据缓存。

数据总线挂载在 APB 总线上或者 AHB 总线上，同时 SDIO 还提供了中断和 DMA 请求。Linux 驱动程序会通过 AHB 总线或者 APB 总线来访问 SDIO，驱动通过读写不同的 SDIO 控制器寄存器来实现对 SDIO 的传输控制，然后 SDIO 控制器就会自动的按照寄存器写好的方式开始对设备（SD 卡）进行读写。

需要注意的是，最开始上电之后，只有 SDIO_D0 作为数据传输，即传输位为 1bit。随后 SDIO 控制器通过 SDIO_D0 信号线与设备进行读写，来设置设备（SD 卡）的传输数据宽度，设置完毕后，SDIO 控制器的数据宽度也将变化为设定的值。

- SDIO_CK

  时钟线，一个位在每个时钟周期内同时在命令和数据线上传输。对于 MMC V3.31，时钟频率可以在 0 MHz 到 20 MHz 之间变化，对于 MMC V4.0/4.2，可以在 0 到 48 MHz 之间变化，对于 SD/SD I/O 卡，可以在 0 到 25 MHz 之间变化

- SDIO_CMD

  - 开漏输出：用于初始化（仅限于 MMC V3.31 版本及以下）
  - 推挽输出：用于命令传输

- SDIO_D[7:0]

  数据线，默认数据线为 SDIO_D0，当设置完毕后即可变化线宽。

表 3-1 SDIO 引脚说明

引脚	方向	说明
SDIO_CK	输出	时钟线
SDIO_CMD	双向	命令线
SDIO_D[7:0]	双向	数据线

数据路径和命令路径是最为关键的，这两个路径直接控制着控制单元与设备之间的数据传输，下面我们来详细了解下命令路径和数据路径。

3.1.2.1 命令路径

从图 3-9 可以看到，命令路径需要由适配器来控制，而 CPU 需要通过对 APB 或者 AHB 总线对适配器寄存器进行读写来控制。下图给出了适配器与命令路径之间的连接关系，可以看到，SDIO 控制器的 CMD 信号线有输入和输出两个方向。 SDIO_CMD 共用一个移位寄存器，当命令由控制器发出时，首先我们需要完成 CMD 和参数的配置，移位寄存器会将数据一位一

位的发送到 CRC 单元，进行循环校验位添加，然后发送给设备。同理，当命令由设备发出时，此时命令会发送给移位寄存器，然后移位寄存器会将数据转换为并行数据保存在响应的 CMD 和寄存器中，这样我们就可以通过 AHB 或者 APB 总线对其进行访问了，从而获取设备数据。

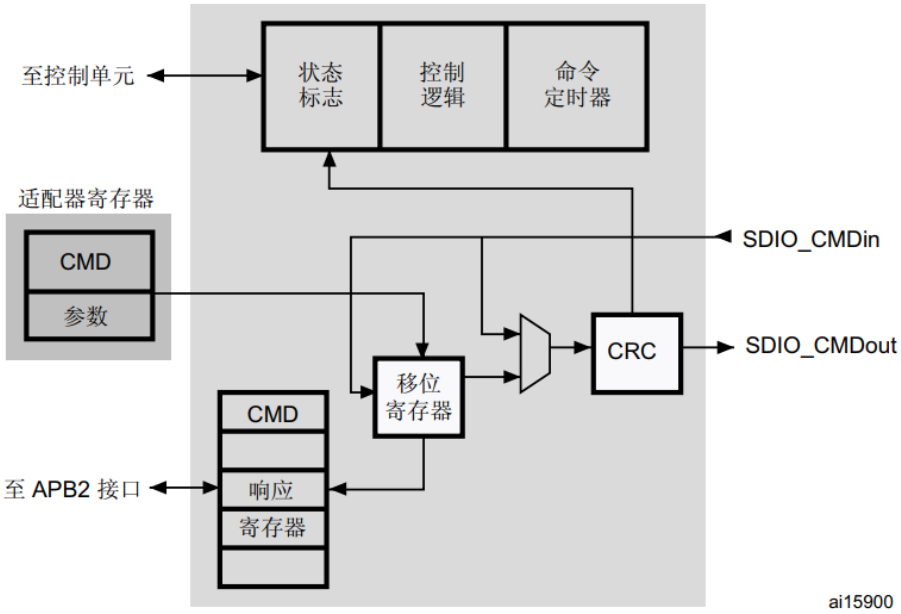


图 3-10 命令路径与适配器寄存器

那数据如何发送呢？当命令寄存器的使能位置 1 时，此时命令开始发送。同时，命令路径中的状态标志立刻会置 1。如果此时是无响应传输，则会进入空闲状态，如果有设备响应，则此时会等待数据回应。

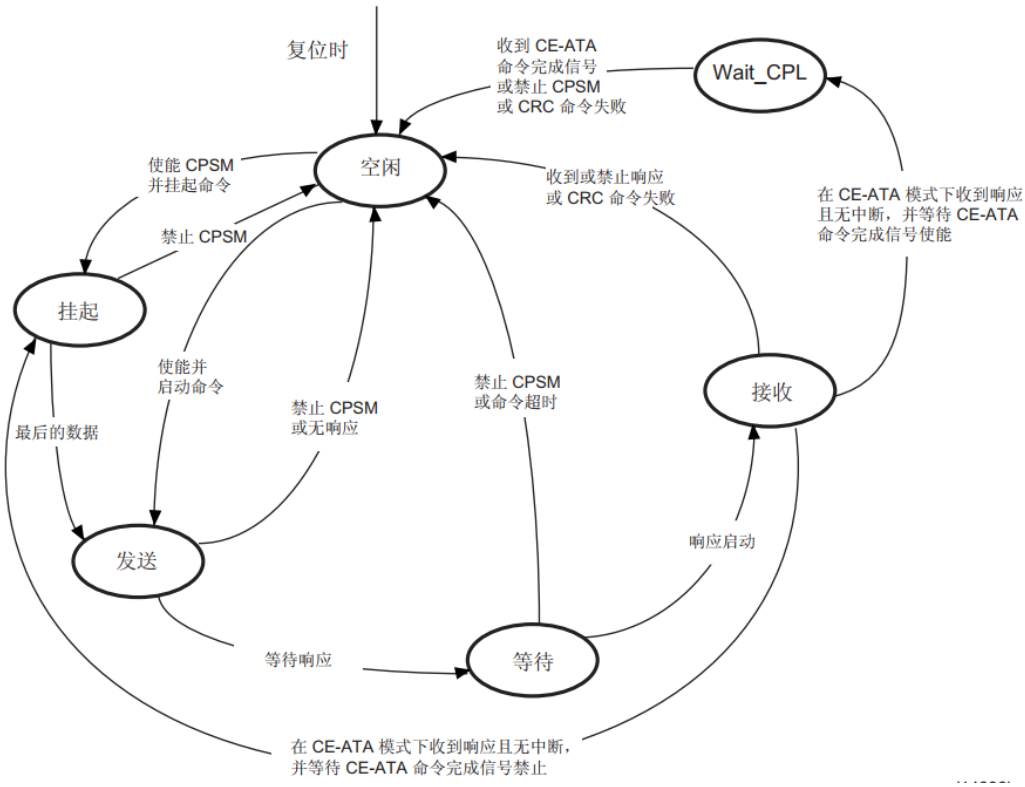


图 3-11 命令路径状态机（CPSM）

上图是命令路径状态机，该状态机详细说明了命令路径的控制过程中状态的转换。首先控制器进入复位状态，此时命令默认为空闲状态，现在我们配置好 CMD 寄存器后，然后将 CMD 寄存器的使能位置 1，此时进入发送状态，当设备有响应时，此时控制器会进入等待响应状态，如果设备正常回复响应，此时会进入接收数据状态，接收来自设备的数据，最后等待 CE-ATA 命令完成信号使能，中间的状态转换过程中的任何出错或者超时都会使控制器进入空闲状态。现在我们已经知道了如何传输命令，但是上面涉及到两个，一个是主机发送的命令，一个是设备的响应，因此我们有必要了解下这两个格式。

◆ SDIO 主机命令格式

下面是 SDIO 的命令格式：

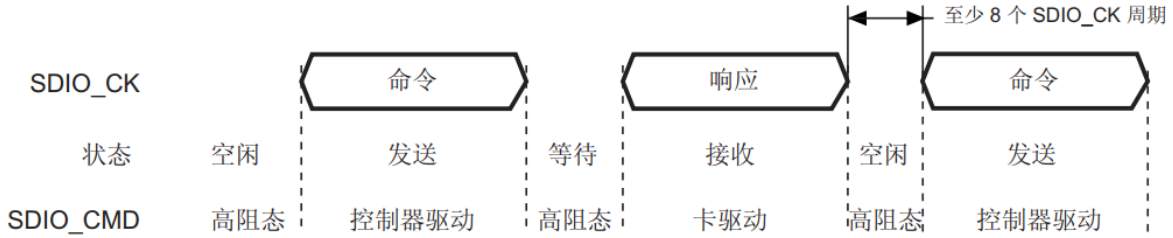


图 3-12 SDIO 命令格式

命令格式较为简单，首先最开始由控制器端发起，此时 SDIO_CMD 信号线会进入高阻态，等待设备响应，当设备回应控制器时，此时 SDIO_CMD 由设备控制，然后进入下一个命令传输。

命令是用于启动操作的令牌。命令从主机发送到单个卡（编址命令），或发送到所有已连接的卡（ MMC V3.31 或更低版本可以使用广播命令）。命令在 CMD 线上以串行方式传输。所有命令都为固定长度 48 位。 表 3-2 中显示了多媒体卡、SD 存储卡和 SDIO 卡的命令令牌的常规格式。 CE-ATA 命令是 MMC 命令 V4.2 的扩展，因此格式相同。

表 3-2 命令格式说明

位的位置	宽度	值	说明
47	1	0	起始位
46	1	1	传输位
45:40	6	-	命令索引
39:8	32	-	参数
7:1	7	-	CRC7
0	1	1	结束位

◆ 设备响应格式

SDIO 控制器支持两种响应格式，分别是 48 位短响应和 136 位长响应，下面两个表格详细说明了这两种格式的位定义，具体传输过程中使用哪种由命令寄存器指定。

表 3-3 短响应格式说明

位的位置	宽度	值	说明
47	1	0	起始位
46	1	0	传输位
45:40	6	-	命令索引
39:8	32	-	参数
7:1	7	-	CRC7（或 1111111）
0	1	1	结束位

表 3-4 长响应格式说明

位的位置	宽度	值	说明
135	1	0	起始位
134	1	0	传输位
133：128	6	111111	保留
127:1	127	-	CID 或 CSD
0	1	1	结束位

3.1.2.2 数据路径

上面我们介绍了命令路径的细节部分，命令路径负责将主机端的命令发送到设备端，同时也负责接收设备端的命令。但传输不仅仅只有命令，还有数据部分，因此 SDIO 控制器还存在数据路径，下面我们来详细了解下数据路径。

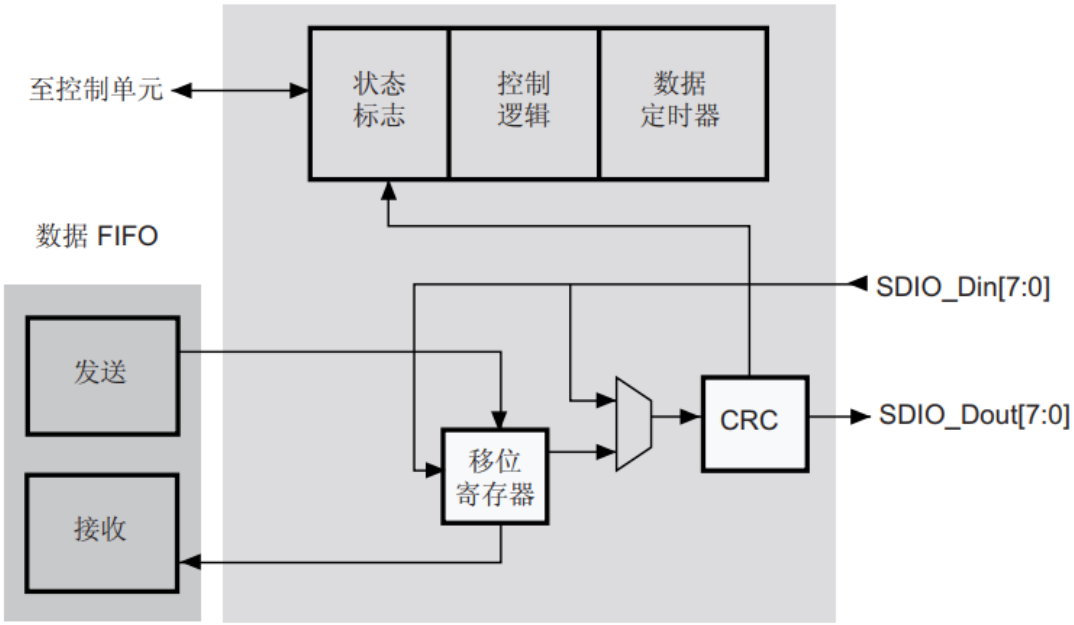


图 3-13 数据路径拓扑图

数据路径和命令路径很相似，也存在移位寄存器，这里的移位寄存器主要是当位宽为 1 位和 4 位时准备的，同时数据路径中最重要的是存在数据 FIFO，其目的就是用来存放主机发送给设备的数据以及接收来自设备端的数据。控制单元会控制数据路径中的状态寄存器、控制逻

辑、数据定时器。很显然，数据路径要比命令路径简单，当我们传输一个有数据传输命令的时候，我们只需要将数据准备好并放入数据 FIFO 中，然后执行命令路径即可发送出去。

可以使用时钟控制寄存器对卡数据总线宽度进行编程。如果使能了 4 位宽度的总线模式，则使用所有四个数据信号线 (SDIO_D[3:0]) 在每个时钟周期内传输 4 个数据位。如果使能了 8 位宽度的总线模式，则使用所有八个数据信号线 (SDIO_D[7:0]) 在每个时钟周期内传输 8 个数据位。如果未使能宽总线模式，则每个时钟周期仅传输一位，且使用 SDIO_D0

命令路径有命令路径状态机，同理，数据路径也有数据路径状态机（DPSM），下图给出了数据命令的状态转换图。

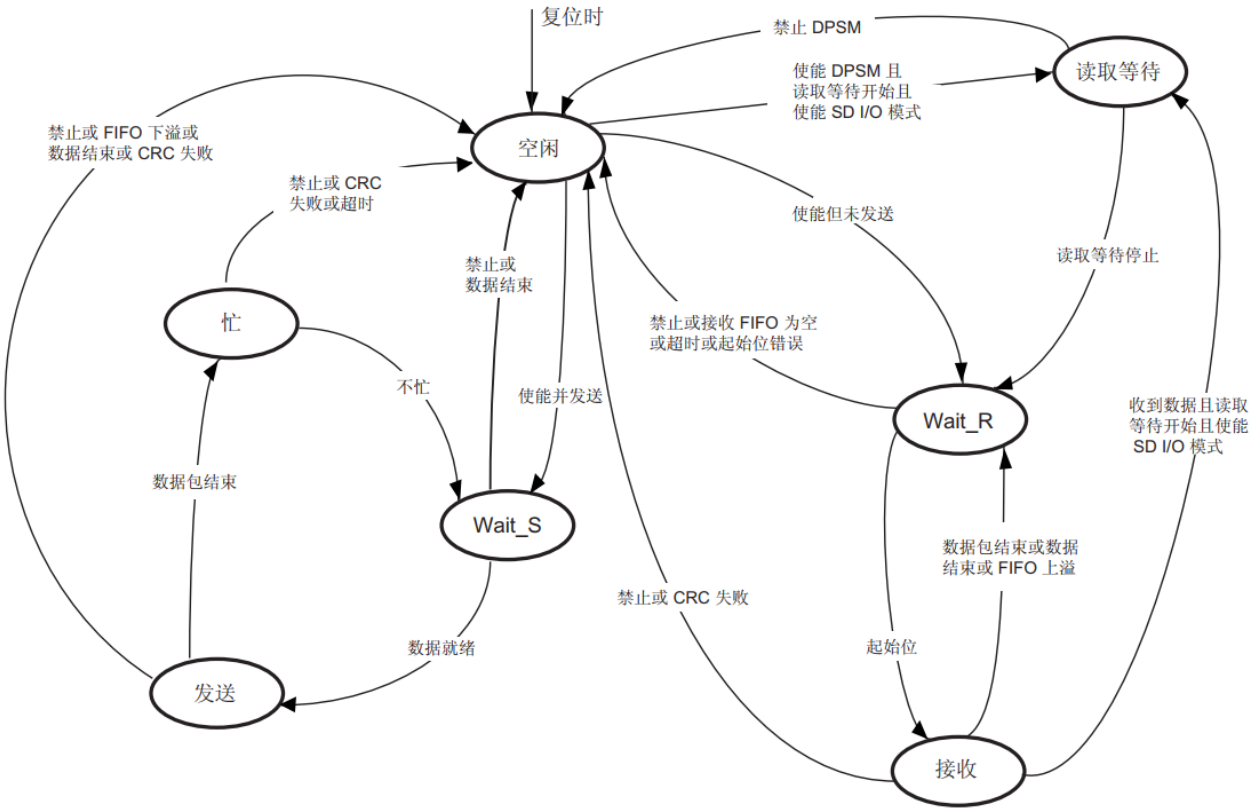


图 3-14 数据路径状态机

3.1.2.3 全志 SD/MMC 控制器

下面我们来看下全志公司的 SDIO 控制器，下图是全志的控制器拓扑图，可以看到其挂载在 AHB 总线上，而且配备了 DMAC 单元，该 DMAC 用于传输 FIFO 数据。

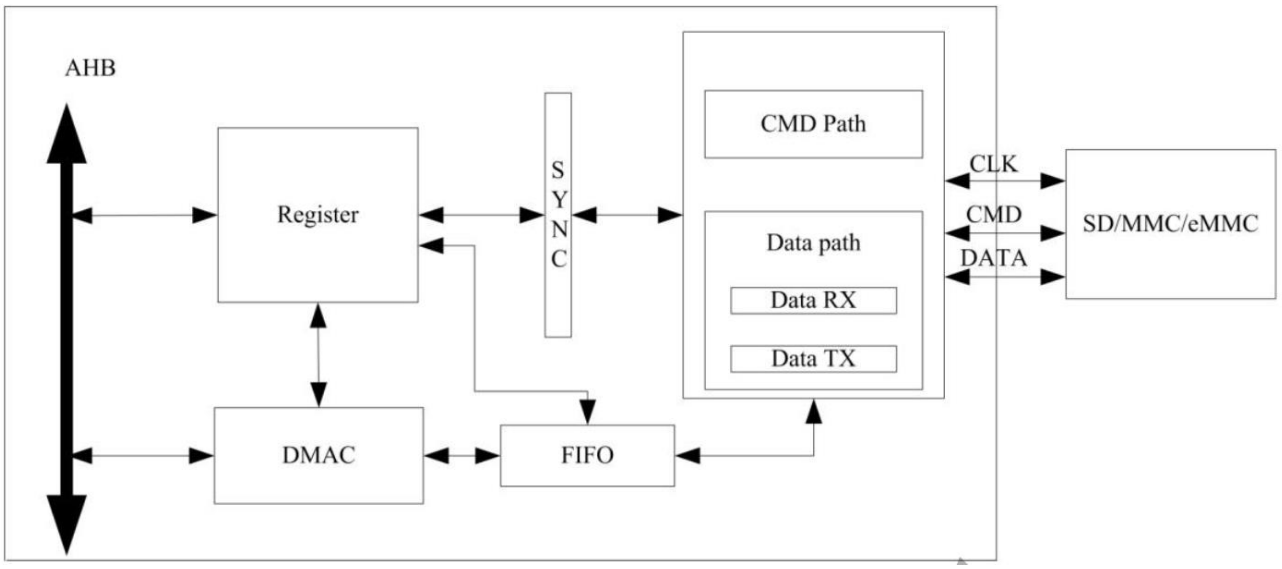


图 3-15 全志 SDIO 控制器拓扑图

这里有必要对 DMAC 控制器说明下，首先每个 DMAC 控制器都是不同的，这里给出全志公司的 SDIO 控制器中 DMAC 的拓扑图，如下图所示：

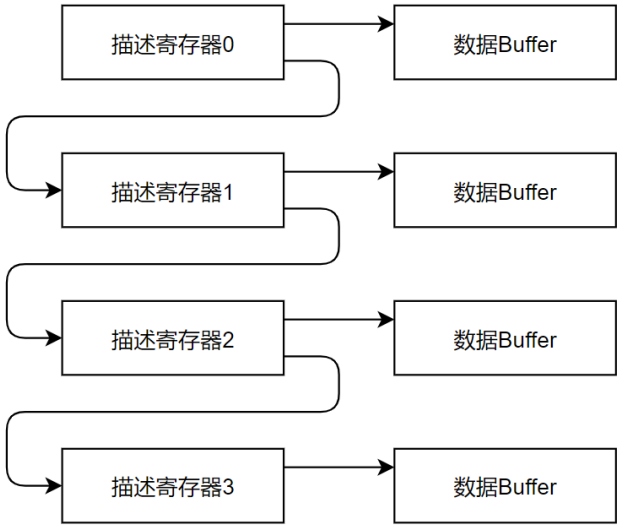


图 3-16 DMAC 结构图

该 DMAC 由多个描述寄存器组成，是一个链式结构，每一个描述寄存器都记录着一个数据 Buffer 的地址，同时还记录着下一个描述寄存器的地址。

3.1.3 控制器操作流程

上面我们已经简单了解了下数据传输类型、控制器说明。其中控制器部分我们已经知道了有两个非常重要的概念，就是命令路径和数据路径，其中命令路径负责传输令牌控制，数据路径负责传输数据包；命令路径由命令寄存器（CMD REG）和命令参数寄存器（CMD ARG REG）来控制，数据路径由 FIFO 来发送和接收数据（对于有些公司集成了 DMAC 单元，用来加快数据的传输）。

命令路径和数据路径的所有状态变化都不需要驱动程序来完成，而是全部由硬件控制器来完成，因此我们只需要对命令寄存器和命令参数寄存器以及数据 FIFO 进行操作即可。

3.1.3.1 命令寄存器

SDIO 控制器通过配置命令寄存器来发送不同的命令，我们来看下全志的 V3s 的 SDIO 控制器的命令寄存器说明：

表 3-5 V3s 命令寄存器说明

Offset: 0x0018			Register Name: SD_CMD
bit	R/W	Default/Hex	Descriptions
31	R/W	0	CMD_LOAD Start Command 当改为被置 1 后，命令会发送出去，同时该位会自动清 0
30	/	/	/
29	R/W	0	Use Hold Register 0：不适用 Hold 寄存器 1：使用 Hold 寄存器
28	R/W	0	VOL_SW Voltage Switch 0：一般命令 1：切换电压命令，仅仅对于 CMD11 有效
27	R/W	0	BOOT_ABT 设置该位会终止启动操作
26	R/W	0	EXP_BOOT_ACK Expect Boot Acknowledge
25:24	R/W	0	BOOT_MOD Boot Mode 00：一般命令 01：强制启动操作 10：可选启动操作 11：保留
23	/	/	/
22	/	/	/
21	R/W	0	PRG_CLK Change Clock 0：一般命令 1：更改设备时钟
20:16	/	/	/
15	R/W	0	SEND_INIT_SEQ 0：一般命令 1：在发送命令之前会先发送一个初始化序列
14	R/W	0	STOP_ABT__CMD

			Stop Abort Command 0：一般命令 1：发送停止命令或者终止数据传输
13	R/W	0	WAIT_PRE_OVER Wait Data Transfer Over 0：立刻发送命令，不关心是否有数据传输 1：在发送命令之前，等待之前的数据传输结束
12	R/W	0	STOP_CMD_FLAG Send Stop Command Automaticaly(CMD12) 0：在数据传输结束时不需要发送停止命令 1：在数据传输时发送停止命令
11	R/W	0	TRANS_MOD Transfer Mode 0：块数据传输 1：数据流传输
10	R/W	0	TRANS_DIR Transfer Direction 0：读操作 1：写操作
9	R/W	0	DATA_TRANS Data Transfer 0：没有数据传输命令 1：有数据传输命令
8	R/W	0	CHK_RESP_CRC Check Response CRC 0：不进行 CRC 校验 1：进行 CRC 校验
7	R/W	0	LONG_RESP Response Type 0：短响应（48 位） 1：长响应（136 位）
6	R/W	0	RESP_RCV Response Receive 0：没有响应传输命令 1：有响应传输命令
5:0	R/W	0	CMD_IDX Command Index 命令索引值，例如 CMD0 此值为 0，CMD1 此值为 1

可以看到，CMD 索引占了 6 位，因此 SDIO 协议总共有$2^6 = 64$条命令。在发送命令之前，我们需要清除 31:6 位，即只保留命令索引，我们再根据具体的命令对该寄存器的相应位进行设置。

3.1.3.2 命令参数寄存器

命令参数寄存器顾名思义就是在发送命令的时候添加参数，当然，一般来说，大部分命令本身就带有参数，只有少部分命令需要额外添加参数，例如读写块的数据时需要传入地址参数

表 3-6 V3s 命令参数寄存器

Offset: 0x001C			Register Name: SD_CMDARG
bit	R/W	Default/Hex	Descriptions
31:0	R/W	0	CMD_ARG：Command Argument

3.1.3.3 中断屏蔽寄存器

在主机发送命令之后，主机可能需要等待很久才能收到设备的响应，这个时候，主机不可能停下来等待响应到达，此时主机会去做其他的事情，而响应是通过中断的方式告诉主机的，当设备发送响应给主机后，SDIO 控制器就会将相应的中断位置 1，表示此时有中断请求，同时通知 CPU 此时会进入中断服务函数中进行特定的操作，因此，我们在写 Host 驱动程序的时候一般会采用中断的方式来接收设备端的响应，而响应的数据会保存在响应寄存器中。

表 3-7 V3 中断屏蔽寄存器

Offset: 0x0030			Register Name: SD_INTMASK
bit	R/W	Default/Hex	Descriptions
31:0	R/W	0	0 - interrupt masked 1 - interrupt enabled Bit field defined as following: bit 31- card removed bit 30 - card inserted bit 17~29 - reserved bit 16 - SDIO interrupt bit 15 - Data End-bit error bit 14 - Auto Stop Command done bit 13 - Data Start Error bit 12 - Command Busy and illegal write bit 11 - FIFO under run/overflow bit 10 - Data starvation timeout /V1 .8 Switch Done bit 9 - Data timeout/Boot data start bit 8 - Response timeout/Boot ACK received bit 7 - Data CRC error bit 6 - Response CRC error bit 5 - Data Receive Request bit 4 Data Transmit Request bit 3- Data Transfer Complete bit 2 - Command Complete bit 1 - Response Error (no response or response CRC error) bit 0 - Reserved

上表为中断屏蔽寄存器的各个位说明，我们只需要按照具体的方式清除相关的位即可。

3.2 MMC Host 驱动

主机驱动也就是我们所说的控制器驱动，该部分是提供给 MMC Core 层回调的，也就是 MMC Core 实现了 MMC 的所有的协议操作，但最终的读写只是一个钩子函数，具体的需要主机控制器驱动来实现。对于 SoC 厂家来说，Host Driver 是需要 BSP 驱动开发工程师编写的，该部分，需要做的事情就是将一些读写、中断、时钟等一些操作实现并注册到驱动框架中即可。下面我们就来详细分析下主机控制器驱动。

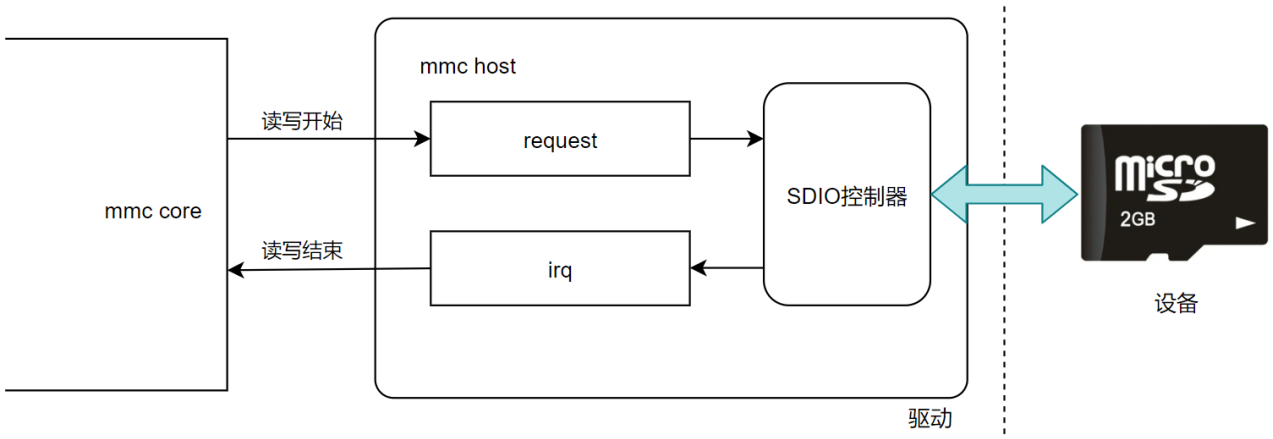


图 3-17 控制器与设备数据交互

mmc 中的一个完成的请求流程：

1. mmc core 调用 request 函数，即 mmc core 发送操作码给 host 层。
2. 在 request 函数中 host 层会根据操作码来控制 SDIO 控制器，从而控制命令路径和数据路径发送相关操作。
3. SDIO 控制器通过命令路径和数据路径将相关信息发送给设备端。
4. 设备端收到命令后做出响应，并返回数据给 SDIO 控制器。
5. SDIO 控制器收到返回数据，并当数据接收完毕后会触发 host 层的 irq 函数。
6. host 层中的 irq 函数执行请求完成实现完整的一个请求。

因此我们在写 SDIO 的 Host 层驱动时，需要完成两个与 mmc core 层请求相关的函数，分别是 request 和 irq 函数。

3.2.1 mmc_host 结构体

```
struct mmc_host {
    struct device       *parent;
    struct device       class_dev;
    int         index;
    const struct mmc_host_ops *ops;
    struct mmc_pwrseq   *pwrseq;
    unsigned int        f_min;
    unsigned int        f_max;
```

```
    unsigned int        f_init;
    u32         ocr_avail;
    u32         ocr_avail_sdio; /* SDIO-specific OCR */
    u32         ocr_avail_sd;   /* SD-specific OCR */
    u32         ocr_avail_mmc;  /* MMC-specific OCR */
    struct wakeup_source    *ws;        /* Enable consume of uevents */
    u32         max_current_330;
    u32         max_current_300;
    u32         max_current_180;
    ...
};
```

该结构体非常庞大，这里仅仅列举一部分，从该结构体可以看到，其中有一个 struct mmc_host_ops *ops 成员，该成员是 mmc_host 结构体中最重要的成员之一，也就是上层对底层的操作接口。下面我们来重点看这个结构体。

由于 mmc 的容量不同，因此后面出现的大容量 mmc 可能无法在老的驱动上运行。mmc 中分三种不同容量的卡，分别是标准容量、大容量和超大容量：

- 标准容量 SD 卡：SDSC，0~2GB
- 大容量 SD 卡：SDHC，2GB~32GB
- 超大容量 SD 卡：32GB~

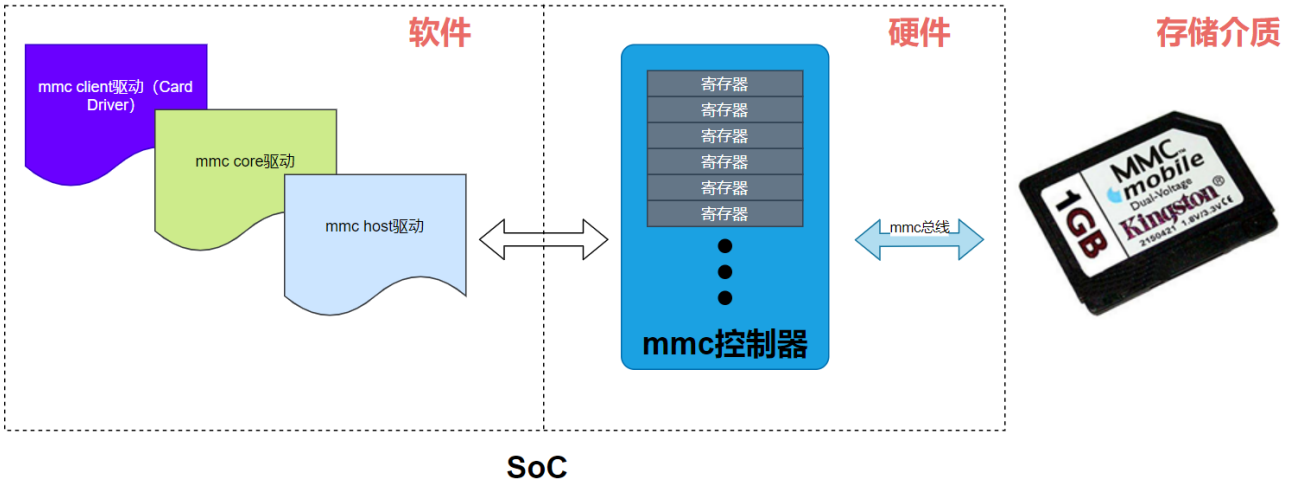


图 3-18 mmc 子系统通过读写 mmc 控制器来实现存储介质的读写

3.2.2 mmc_host_ops 结构体

MMC 子系统将主机驱动中的读写抽象为 request，其具体的结构体如下所示：

```
struct mmc_host_ops {
    void    (*post_req)(struct mmc_host *host, struct mmc_request *req,int
             err);
    void    (*pre_req)(struct mmc_host *host, struct mmc_request *req);
    void    (*request)(struct mmc_host *host, struct mmc_request *req);
```

```
    /* Submit one request to host in atomic context. */
    int (*request_atomic)(struct mmc_host *host,struct mmc_request *req);
    void    (*set_ios)(struct mmc_host *host, struct mmc_ios *ios);
    int (*get_ro)(struct mmc_host *host);
    int (*get_cd)(struct mmc_host *host);
    void    (*enable_sdio_irq)(struct mmc_host *host, int enable);
    /* Mandatory callback when using MMC_CAP2_SDIO_IRQ_NOTHREAD. */
    void    (*ack_sdio_irq)(struct mmc_host *host);
    /* optional callback for HC quirks */
    void    (*init_card)(struct mmc_host *host, struct mmc_card *card);
    int (*start_signal_voltage_switch)(struct mmc_host *host, struct mmc_ios
         *ios);
    /* Check if the card is pulling dat[0:3] low */
    int (*card_busy)(struct mmc_host *host);
    /* The tuning command opcode value is different for SD and eMMC cards */
    int (*execute_tuning)(struct mmc_host *host, u32 opcode);
    /* Prepare HS400 target operating frequency depending host driver */
    int (*prepare_hs400_tuning)(struct mmc_host *host, struct mmc_ios *ios);
    /* Prepare switch to DDR during the HS400 init sequence */
    int (*hs400_prepare_ddr)(struct mmc_host *host);
    /* Prepare for switching from HS400 to HS200 */
    void    (*hs400_downgrade)(struct mmc_host *host);
    /* Complete selection of HS400 */
    void    (*hs400_complete)(struct mmc_host *host);
    /* Prepare enhanced strobe depending host driver */
    void    (*hs400_enhanced_strobe)(struct mmc_host *host,struct mmc_ios
         *ios);
    int (*select_drive_strength)(struct mmc_card *card,unsigned int max_dtr,
         int host_drv,int card_drv, int *drv_type);
    /* Reset the eMMC card via RST_n */
    void    (*hw_reset)(struct mmc_host *host);
    void    (*card_event)(struct mmc_host *host);
    int (*multi_io_quirk)(struct mmc_card *card,
         unsigned int direction, int blk_size);
    /* Initialize an SD express card, mandatory for MMC_CAP2_SD_EXP. */
    int (*init_sd_express)(struct mmc_host *host, struct mmc_ios *ios);
};
```

下面我们简单介绍下其中常用的成员。

- post_req，pre_req：这两个成员用来提供 Linux 中 request 的双缓冲机制，是可选项。
- request：该成员实现了 mmc 的读写操作请求，是 ops 中最重要的成员之一。
- request_atomic：原子操作请求。
- set_ios：该函数是 core 层调用寄存器的时候会最开始执行的函数，在该函数中一

般需要实现 mmc 的识别、电源开启、时钟频率、总线宽度等必要的操作。

- get_ro：返回卡的读写属性，可以使用内核默认的 mmc_gpio_get_ro 来检测卡的读写属性。
- get_cd：返回卡是否存在，可以使用内核默认的 mmc_gpio_get_cd 函数。
- enable_sdio_irq：开启 mmc 或者 SD 卡的中断
- start_signal_voltage_switch：切换卡的电压，大部分卡都默认 3.3V，因此直接返回 3.3V 电压即可。
- hw_reset：硬件复位，复位 SD 卡或者 MMC 卡。
- card_busy：检测 SD 卡或者 MMC 卡是否为忙状态。

上面的函数中，request 和 set_ios 是必须要实现的，因此我们在写控制器驱动的时候必须实现这两个函数。

3.2.2.1 set_ios 函数指针

该函数较为简单，当 Linux 内核中的 core 层调用 mmc 驱动时，此时最开始会调用此函数，用来初始化 mmc 卡。

1. **设置电源**

   我们检测 mmc 卡的电源是否开启，mmc_ios 结构体中的 power_mode 保存者上层的电源状态，我们可以根据这个状态来做相应的设置，当然，我们也可以认为我们的 mmc 电源一直都是开启的，直接返回对应的状态即可。

2. **设置总线宽度**

   mmc 支持三种总线位宽，分别是 1bit、4bit、8bit。一般 mmc 控制器都会有一个寄存器用来设置 mmc 数据传输时的总线宽度。因此，我们直接对寄存器写如相应的值即可，例如下面这个代码：

```
static void sunxi_mmc_set_bus_width(struct sunxi_mmc_host *host,
                                    unsigned char width)
{
    switch (width) {
    case MMC_BUS_WIDTH_1:
        mmc_writel(host, REG_WIDTH, SDXC_WIDTH1);
        break;
    case MMC_BUS_WIDTH_4:
        mmc_writel(host, REG_WIDTH, SDXC_WIDTH4);
        break;
    case MMC_BUS_WIDTH_8:
        mmc_writel(host, REG_WIDTH, SDXC_WIDTH8);
        break;
    }
}
```

上面的代码时全志的 SoC 中设置总线位宽的函数，这个函数实际上就是对相应的寄存

器进行设置，具体的可以看寄存器手册。

3. 设置时钟频率

在数据传输之前，我们必须设置好 mmc 控制器读写 mmc 卡的时钟频率。实际上也是设置 mmc 的寄存器，例如，下面时全志 SoC 设置时钟的函数：

```
static void sunxi_mmc_set_clk(struct sunxi_mmc_host *host, struct mmc_ios
*ios)
{
    u32 rval;
    /* set ddr mode */
    rval = mmc_readl(host, REG_GCTRL);
    if (ios->timing == MMC_TIMING_UHS_DDR50 ||
        ios->timing == MMC_TIMING_MMC_DDR52)
        rval |= SDXC_DDR_MODE;
    else
        rval &= ~SDXC_DDR_MODE;
    mmc_writel(host, REG_GCTRL, rval);
    host->ferror = sunxi_mmc_clk_set_rate(host, ios);
    /* Android code had a usleep_range(50000, 55000); here */
}
```

为了不影响其他位的设置，上面的代码最开始先读取了下原来寄存器中值，然后再设置相应的位，这种操作方法在 Linux 中较为常见。

3.2.2.2 request 函数指针

这个函数指针是 mmc_host_ops 结构体中最重要的成员，mmc core 层对 mmc 卡的读写都是通过该函数来完成的，该函数实现了对 mmc 的命令处理，数据传输等功能。

mmc 协议将对 mmc 卡之间的通信分为命令路径传输和数据路径传输两种，mmc 子系统将这命令传输抽象为 struct mmc_command，将数据传输抽象为 struct mmc_data。

我们先来看命令路径传输，struct mmc_command 结构体虽然看起来非常庞大，但较为简单，其定义如下：

```
struct mmc_command {
    u32         opcode;
    u32         arg;
#define MMC_CMD23_ARG_REL_WR    (1 << 31)
#define MMC_CMD23_ARG_PACKED    ((0 << 31) | (1 << 30))
#define MMC_CMD23_ARG_TAG_REQ   (1 << 29)
    u32         resp[4];
    unsigned int        flags;      /* expected response type */
#define MMC_RSP_PRESENT (1 << 0)
#define MMC_RSP_136 (1 << 1)        /* 136 bit response */
```

```
#define MMC_RSP_CRC (1 << 2)        /* expect valid crc */
#define MMC_RSP_BUSY    (1 << 3)        /* card may send busy */
    #define MMC_RSP_OPCODE  (1 << 4)        /* response contains opcode */
    ...
unsigned int        sg_len;     /* size of scatter list */
    int         sg_count;   /* mapped sg entries */
    struct scatterlist  *sg;        /* I/O scatter list */
     s32         host_cookie;    /* host private data */
}
```

从结构体中可以看到，mmc_command 有操作码、参数、以及响应、标志、段信息（数据位置）等等。这些记录了一个命令的所有信息，我们可以根据 mmc_command 来查看当前 core 层发送下来的命令信息，并执行相应的操作。

我们来看下操作码的定义，在 include/linux/mmc/mmc.h 中定义了所有的操作码，如下所示（部分）：

```
/* Standard MMC commands (4.1)           type  argument     response */
   /* class 1 */
#define MMC_GO_IDLE_STATE         0   /* bc                          */
#define MMC_SEND_OP_COND          1   /* bcr  [31:0] OCR         R3  */
#define MMC_ALL_SEND_CID          2   /* bcr                     R2  */
#define MMC_SET_RELATIVE_ADDR     3   /* ac   [31:16] RCA        R1  */
#define MMC_SET_DSR               4   /* bc   [31:16] RCA            */
#define MMC_SLEEP_AWAKE           5   /* ac   [31:16] RCA 15:flg R1b */
#define MMC_SWITCH                6   /* ac   [31:0] See below   R1b */
#define MMC_SELECT_CARD           7   /* ac   [31:16] RCA        R1  */
#define MMC_SEND_EXT_CSD          8   /* adtc                    R1  */
#define MMC_SEND_CSD              9   /* ac   [31:16] RCA        R2  */
#define MMC_SEND_CID             10   /* ac   [31:16] RCA        R2  */
#define MMC_READ_DAT_UNTIL_STOP  11   /* adtc [31:0] dadr        R1  */
#define MMC_STOP_TRANSMISSION    12   /* ac                      R1b */
#define MMC_SEND_STATUS          13   /* ac   [31:16] RCA        R1  */
#define MMC_BUS_TEST_R           14   /* adtc                    R1  */
#define MMC_GO_INACTIVE_STATE    15   /* ac   [31:16] RCA            */
#define MMC_BUS_TEST_W           19   /* adtc                    R1  */
#define MMC_SPI_READ_OCR         58   /* spi                 spi_R3  */
#define MMC_SPI_CRC_ON_OFF       59   /* spi  [0:0] flag     spi_R1  */
```

这些操作都是 Linux 内核标准的操作，每一个操作都对应这一个命令路径中的状态，我们可以根据命令路径来完成所有的操作。

结构体中的 flags 表示需要期待返回的数据类型，该值对应着 CMD 寄存器中的一些标志位，例如命令是否有响应，响应为长响应还是短响应等等。

下面我们再来看 struct mmc_data 结构体，该结构体记录了 core 层发送下来的数据信息，这些信息并没有直接保存数据，而是将数据存放的段信息保存下来，即 sg 信息。

```
struct mmc_data {
    unsigned int        timeout_ns; /* data timeout (in ns, max 80ms) */
    unsigned int        timeout_clks;   /* data timeout (in clocks) */
    unsigned int        blksz;      /* data block size */
    unsigned int        blocks;     /* number of blocks */
    unsigned int        blk_addr;   /* block address */
    int         error;      /* data error */
    unsigned int        flags;
#define MMC_DATA_WRITE      BIT(8)
#define MMC_DATA_READ       BIT(9)
/* Extra flags used by CQE */
#define MMC_DATA_QBR        BIT(10)     /* CQE queue barrier*/
#define MMC_DATA_PRIO       BIT(11)     /* CQE high priority */
#define MMC_DATA_REL_WR     BIT(12)     /* Reliable write */
#define MMC_DATA_DAT_TAG    BIT(13)     /* Tag request */
#define MMC_DATA_FORCED_PRG BIT(14)     /* Forced programming */
    unsigned int        bytes_xfered;
    struct mmc_command  *stop;      /* stop command */
    struct mmc_request  *mrq;       /* associated request */
    unsigned int        sg_len;     /* size of scatter list */
    int         sg_count;   /* mapped sg entries */
    struct scatterlist  *sg;        /* I/O scatter list */
    s32         host_cookie;    /* host private data */
};
```

和 command 路径传输一样，在数据路径传输也保存了数据的段信息。因此，core 层发送数据时，并没有直接将数据发送给 Host 驱动层，而是将数据的地址告诉 host，这样做的目的是几乎所有的 mmc 控制器都是支持 DMA 数据传输方式，如果不适应 DMA 会非常耗 CPU 的资源，因此 core 层发送数据给 host 层驱动时，只需要告诉数据的位置在哪，然后 host 层就会利用 DMA 将数据传输给 FIFO。

现在我们大致知道了 core 层通过发送 commond 路径传输和 data 路径传输给 host 层，实际上这个过程就是 core 调用 request 函数。但具体怎么做我们还需要熟悉 mmc 协议，mmc 的读写流程比较复杂，涉及到很多的命令，这么多的命令并不是所有的命令都和底层相关，因此，core 层完成了绝大部分的命令和几乎所有的协议，但涉及到寄存器相关的必须有 host 层来完成，这样我们就需要实现一些与寄存器相关的命令。

在分析 request 函数之前，我们需要对 mmc 或者 SDIO 的读写流程有一定的了解，该部分请详见 3.1 章节。

request 的需要实现的步骤如下：

1. 根据 mmc_request 中的 data 来判断此命令是否有数据传输，如果有数据传输，则准备进行 DMA 映射，为数据搬移做准备。
2. 根据 mmc_request 中的 cmd->flag 标志来设置 CMD 中的各个标志位。
3. 设置好 CMD 寄存器相关的标志位后，开始数据搬移。

3.2.3 irq 中断函数

从图 3-17 可以看到，mmc core 层通过调用 host 层的 request 来实现读写请求，但对于大量数据而言，对于 CPU 而言，读写肯定是一件非常耗时的一件事，因此，调用 request 函数后应该立刻返回，而不是等待读写完毕再返回。这样就需要由一个机制来通知控制器读写完毕操作，这个地方就引入中断方式来实现。对于 mmc 子系统而言，mmc core 层调用 request 函数来发送命令，然后由 irq 中断方式来告诉 mmc 子系统读写完毕。

在中断函数中，做的事情比较简单，主要是清除中断标志位并通知 mmc core 请求执行完毕。

3.2.3 mmc host 驱动编写流程

下面我们详细讲解下如何注册 mmc host 驱动，大致步骤如下：

1. 在 probe 中调用 `mmc_alloc_host` 函数动态分配一个 `mmc` 主机结构体：

   我们先看下该函数的原型，如下：

   ```
   struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
   ```

   - `extra`：分配内存的大小
   - `dev`：平台设备

2. 为 mmc 中断申请资源：

   使用 `request_irq` 或者 `request_threaded_irq` 或者 dev_request_threaded_irq 函数申请一个中断资源。一般来说建议使用 `dev_request_threaded_irq` 来申请线程化中断服务函数，其主要原因是我们希望将比较重要的操作在上半部分执行，而不是那么紧急的事情在下半部分执行，即线程函数中执行。

3. 为 `mmc` host 驱动申请数据缓存：

   用来存放 mmc core 中读写数据时的缓存，该缓存需要使用 DMA 方式申请，因为对于 mmc 控制器而言，搬运数据都是使用 DMA 方式的。

4. 初始化 `mmc` 控制器：

   由于 mmc 控制器复位之后需要进行初始化，例如初始化时钟、初始化 DMA，初始化屏蔽中断等。

5. 使用 `mmc_add_host` 函数将 `mmc` 注册到子系统中：

   该函数原型如下。

   ```
   int mmc_add_host(struct mmc_host *host)
   ```

   和其他子系统一样，这里需要将设备添加到该子系统中，这样子系统在完成读写操作时就可以实现最终的介质读写。

6. 定义 `mmc_host_ops` 操作集合结构体，同时实现该操作集合中的重要成员，例如

request、set_ios、get_ro(非必须)、get_cd(非必须)等：

request 请求处理函数用来处理 mmc core 层下发的命令；set_ios 函数用来实现 mmc core 调用 host 最开始会先调用 set_ios 一次，用来初始化一些设备属性，例如时钟频率、片选电平、总线宽度、电压、驱动类型、输出模式等等。

3.2.4 mmc 驱动示例

上面我们详细讲解了 mmc 子系统的相关结构，现在我们来实现一个简单的 mmc 驱动程序。Linux 内核中已经对全志的大部分芯片的 mmc 控制器进行了支持（该部分由全志公司编写的），读者可以仔细结合上述章节配合来阅读源码。由于官方的驱动考虑的过去周到，导致其整个逻辑变得不易察觉，因此笔者将其进行了简化，一步一步为 SoC 实现 mmc host 驱动。

这里使用的平台为笔者自己设计的开发板，该开发板的 SoC 为 F1C200S，内部集成了 64Mbyte DDR 内存，主频默认 400MHz，自带一个 SDIO 控制器，详细参数请参考相关数据手册。首先，我们还是需要在 dts 文件中添加设备节点，如下，实际上就是 Linux 中默认的 mmc0 设备节点。下面我们在 dtsi 文件中添加设备如下设备节点，注意，由于 pinctrl 节点中的引脚描述可能不完全，因此我们还需要在 pinctrl 节点中添加 mmc0 引脚说名，如下：

```
mmc0_pins: mmc0-pins {
    pins = "PF0", "PF1", "PF2", "PF3", "PF4", "PF5";
    function = "mmc0";
};
```

上面的 mmc0-pins 节点中的 pins 属性的值没有先后顺序，只要将 mmc0 引脚添加进入即可，这是因为 pins 仅仅指定了 mmc0 将要使用这些引脚，这些引脚将不能再被其他驱动申请，而真正对每个引脚初始化的是 function 属性指定的，在 pinctrl 子系统中，通过 function 属性的值 mmc0 来对每个引脚进行具体的初始化，其 CLK、CMD、D0~D7 引脚都已经在 pinctrl 子系统中定义了。然后再定义 mmc@1c0f000 设备节点，并添加 mmc0 pandle 引用，如下：

```
mmc0: mmc@1c0f000 {
    compatible = "allwinner,f1c200s-mmc-test ";
    reg     = <0x01c0f000 0x1000>;
    clocks =  <&ccu CLK_BUS_MMC0>,<&ccu CLK_MMC0>,
              <&ccu CLK_MMC0_OUTPUT>,<&ccu CLK_MMC0_SAMPLE>;
    clock-names = "ahb","mmc","output","sample";
    resets = <&ccu RST_BUS_MMC0>;
    reset-names = "ahb";
    interrupts = <23>;
    pinctrl-names = "default";
    pinctrl-0 = <&mmc0_pins>;
    status = "disabled";
    #address-cells = <1>;
    #size-cells = <0>;
};
```

我们对上面的设备节点进行简单的说明下，首先 compatible 属性用来匹配驱动，reg 属性指定

了该设备的地址，通过数据手册，我们可以看到，mmc0 的起始地址为 0x01C0F000：

表 3-8 F1C200S 芯片的 mmc 控制器基地址

Module Name	Base Address
SDC0	0x01C0F000
SDC1	0x01C10000

clocks 属性指定了 mmc0 的时钟源，clock-name 属性对 clocks 属性进行了字段说明，这里指定了四个时钟，第一个时钟为 AHB 时钟；第二个时钟为 mmc0 时钟，第三个时钟为输出

和 MCU 开发过程一样，所有的外设在使用之前必须开启相应的时钟，因此在驱动挂载时候，我们需要开启 clocks 属性中的所有时钟。

resets 属性指定了外设的复位时钟信号，reset-names 属性对 resets 属性进行了说明，在驱动中可以通过 reset-names 属性值来直接获取 reset 的属性值，这样更容易阅读。interrupts 属性指定了该外设的中断号，通过中断控制器属性可以看到，中断号的属性 cell 只有一个。从数据手册可以看到 mmc0 的中断号为 23，需要注意的是，这里的中断号和 Cortex A7 以及之后的带有 GIC 控制器的 SoC 有点区别，在 GIC 控制器中，中断有偏移，因此在 GIC 控制器中的中断的属性值=物理中断号-偏移基数。pinctrl-names 属性是 pinctrl 子系统中的最常用的属性，这里值为 default。而 pinctrl-0 属性指定了 pinctrl-names 属性的第一个值，因此这里的引脚只有一种模式，就是映射为 mmc 引脚，实际上很多引脚可能会复用为其他功能引脚，这时我们需要添加 pinctrl-1 属性了，用来说明当 mmc 睡眠时，此引脚的功能。status 属性指定了给节点并未开启，我们可以在 dts 文件中引用 mmc0 节点并修改该 status 属性为“okay”。 #address-cells 和#size-cells 这两个属性相信读者非常熟悉了，该属性指定了该设备节点的子节点的 reg 属性的 cell 单元。

上面我们已经编写好了设备节点，剩下的就是开始编写驱动代码了。首先我们最开始需要实现 probe 函数，该函数中，我们主要的事情是需要完成 host 内存分配以及相关参数设置。

3.3 SDHCI 驱动

SDHCI（SDIO Host Controller Interface）,即 SDIO 主机控制器接口。SD 卡对读者来说可能并不陌生，在大部分的数码相机中，存储数据所用的介质正是 SD 卡，其总线协议就是 SDIO 协议（当然，SD 卡也支持 SPI 协议）。随着 SD 卡的普及以及该协议的优越性，越来越多的外围芯片采用 SDIO 协议来传输数据，例如 ESP8089 WIFI 芯片所用的接口便是 SDIO 协议。因此 SDHCI 驱动并不仅仅针对 SD 卡，同时还针对所有采用 SDIO 协议的设备。

第四章 SCSI 子系统

SCSI（Small Computer System Interface）即小型计算机系统接口，该接口是一种用于计算机及其周边设备之间（硬盘、软驱、光驱、打印机、扫描仪等）系统级接口的独立处理器标准。可以说 SCSI 是目前众多子系统中最复杂的子系统之一了，目前该子系统几乎包含了所有的块设备驱动。

第五章 UFS 设备

什么是 UFS 呢？在目前的中高端手机中，使用的外存正是 UFS 芯片，而在一些比较低端的手机中和一些嵌入式设备中使用的是 eMMC 作为外存。中高端手机中使用 UFS 的主要原因是 UFS 的速度非常快，下面是 UFS 芯片和 eMMC 芯片的读写性能对比：

表 5-1 eMMC 与 UFS 性能对比

闪存版本	eMMC 4.5	eMMC 5.0	eMMC 5.1	UFS 2.0	UFS 3.1
理论带宽	200MB/s	400MB/s	600MB/s	1450MB/s	2900MB/s
顺序读取速度	140MB/s	250MB/s	250MB/s	350MB/s	1821MB/s
顺序写入速度	50MB/s	90MB/s	125MB/s	150MB/s	760MB/s
随机读取速度	7K IOPS[8]	7K IOPS	11K IOPS	19K IOPS	60K IOPS
随机写入速度	2K IOPS	13K IOPS	13K IOPS	14K IOPS	70K IOPS

可以看到，UFS 的性能已经远远高于 eMMC 的性能，随着 UFS 的普及，越来越多的手机厂商开始使用 UFS。为何 UFS 有如此快的速度呢？这是因为 UFS 采用的是差分信号，而 eMMC 使用的还是 SDIO 总线。

5.1 UFS 简介

UFS 是目前高端手机中最常见的存储设备，如下图所示：

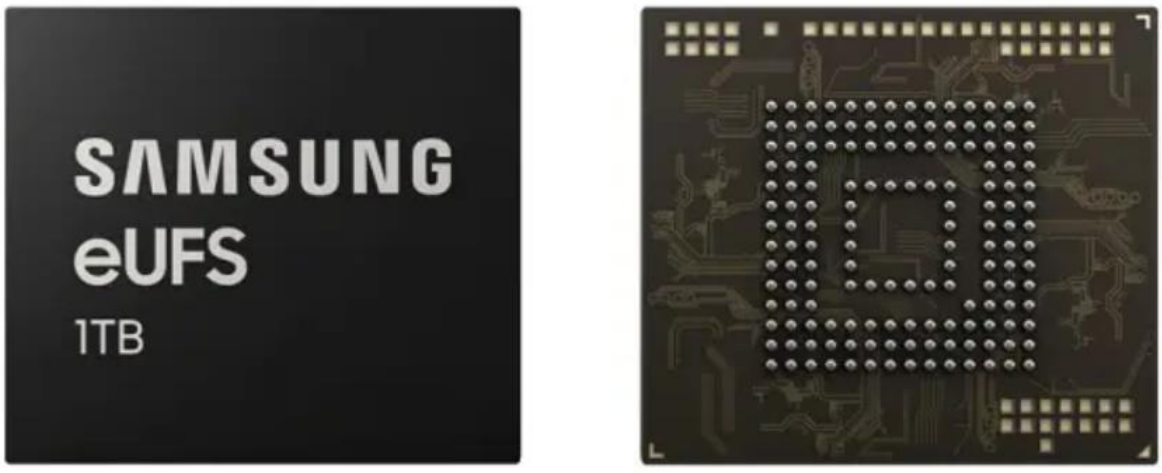


图 5-1 UFS 芯片

目前主要的 UFS 厂商有三星、海力士、美光、铠侠，这些厂商几乎垄断了市场的 90%的市场，UFS 芯片的引脚不是很多，最常见的封装便是 uMCP[9]。

搭载 UFS 手机的 SoC 芯片内部都会集成 UFS 控制器，就像 eMMC 一样，SoC 需要自带控制器才能对 eMMC 芯片进行控制和数据传输。

[8] IOPS: Input/Output Operations Per Second，即 Input/Output Operations Per Second

[9] uMCP 基于 eMCP 的方向发展而来，封装尺寸基本相同：11.5mmx13mm（其它尺寸有 12x16；14x18；16x20；11x10mm）

图 5-2 手机主板上的 UFS 芯片

下面我们来看下 UFS 的内部结构，如下图所示：

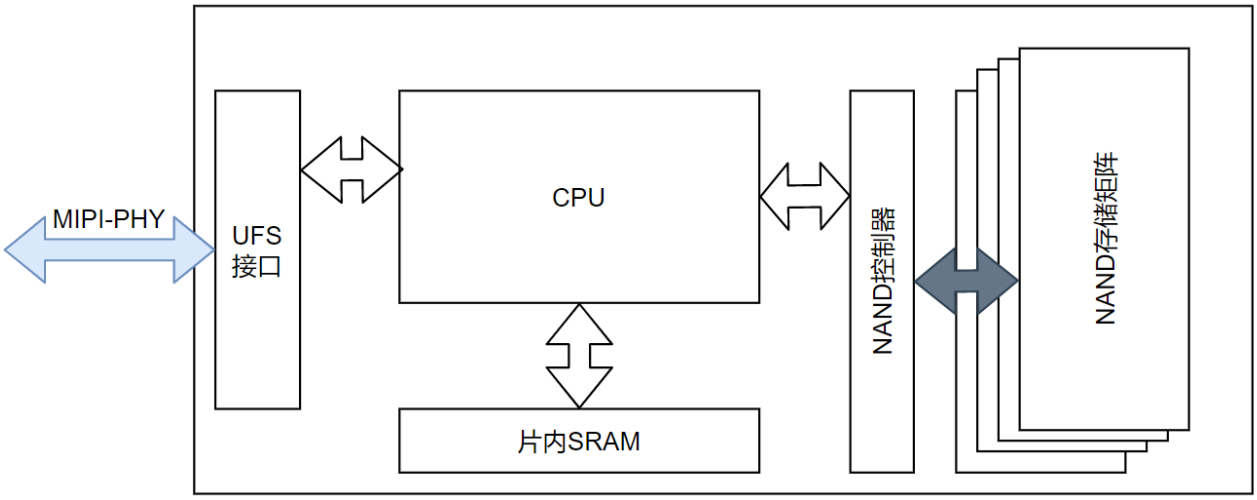


图 5-3 UFS 结构图

UFS 芯片由 UFS 接口、UFS 控制器（CPU）、片内 SRAM、NAND 控制器、NAND 存储矩阵构成。其中 UFS 接口使用的是 MIPI-PHY 总线接口，一般来说，NAND 控制器会直接集成在 CPU 中，而由于 UFS 芯片体积很小，无法像 SSD 那样做成 PCB（SSD 一般会外挂 DDR 作为内存），因此 UFS 内部的 SRAM 非常小，这就使其内存资源非常紧张，这也是 SSD 与 UFS 的最大区别，后面我们会详细讲解为何内存的容量不同导致其软件架构上的差异较大。

5.1.1 UFS 固件框架

UFS 中固件的框架主要分三个部分，分别是前端、中端、后端。前端部分负责处理与手机 SoC 之间的命令处理和数据传输；中断负责算法实现、后端负责将数据写入到实际的 NAND 存储矩阵中。其中中端部分是整个固件最重要的部分，下面我们来详细了解下中端部分的具体细节。

中端部分最重要的便是负责将逻辑地址转换为物理地址，因此也称为 FTL（Flash Translation Layer），即闪存转换层。为何需要有闪存转换层呢？这是由于 NAND 的读写特性造成的，对于 NAND 而言，写之前必须先擦除，但是对于大容量的 NAND 而言，擦除是以块为单位的，这样就存在一个问题，即使写入 1 个字节的数据（实际上不可能写一个字节的数据），那么需要先擦除一个块，然后再写入，那么问题来了，假设又要写一个字节的数据呢，那么此时又要擦除一个块，然后再写入，如果不采取任何措施，这样是不是显得非常荒诞。是的，对于用户（SoC）而言，它是不会关心这些的，它认为写到哪个地址上就写哪个地址上了，即使写入一个字节也得写入，因此在 UFS 内部需要完成这个工作，这便是 FTL 的最重要的功能。

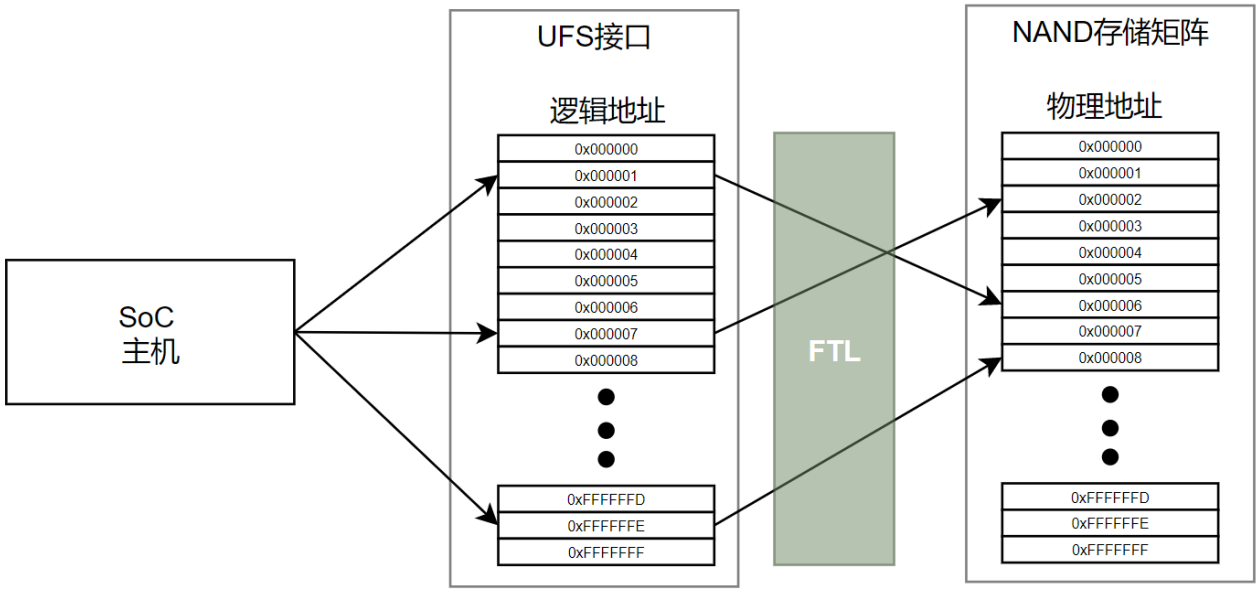


图 5-4 逻辑地址到物理地址转换

对于 SoC 而言，其访问的都是逻辑地址，例如将数据写到地址 0x00 上，而实际上数据不一定在 NAND 的地址 0x00 上。我们将逻辑地址称为 LBA（Logical Block Address），物理地址称为 PPA（Page Physical Address），这样 FTL 的主要任务就是实现 LBA 到 PPA 的转换。当然由于 NAND 的写不能直接写入到已经写过的块上，因此在 FTL 中还需要实现 GC（Garbage Collection）即垃圾回收。由于 NAND 非常容易损坏，因此我们不能总是让几个块一直来回擦除，而是将读写的次数分摊到各个不同的块上，也就是 WL（Wear Leveling）即磨损平衡。每个块由很多个页组成，但有些页上的数据已经被删除了，对于 NAND 而言不能直接擦除（因为擦除是以块为单位的，而该块很有可能存在有效的数据），而是标记该数据为垃圾，这样我们就需要标记一个块中的哪些数据是有效的，哪些数据是无效的，这种非常类似于动态内存分配，因此也需要引入一个表来记录没一个块中页的数据有效性，我们称该表为 MB（Mapping BitMap），即位图映射。当然上面只是 FTL 中最重要的一些算法，实际的 FTL 中还包含这其他的算法，例如 APL（异常掉电恢复）、后台垃圾回收（BKOPS）等，下面我们详细来讲解下上面的各个算法。

5.1.1.1 L2P（Logical To Physical）

图 5-4 描述了逻辑地址到物理地址的转换，我们现在来具体讲解下 L2P 的实现过程，首先 L2P 的本质是逻辑地址到物理地址的转换，因此我们需要有一个表来记录两个转换的关系。很显然，如果我们的地址可以单个字节访问，那么我们的表的大小就是存储大小，然而实际上

NAND 的最小读写是 Page，即页，对于 UFS 而言，其页一般为 16Kbyte，但由于历史原因，当前 UFS 大部分的逻辑地址以一个 4KByte 为单位。这样我们的需要转换的最小单位不是一个字节了，而是 4Kbyte，现在以一个 128GByte 的存储容量来计算，那么我们的需要的映射表的大小就是 128/4K*4 = 128Mbyte[10]。

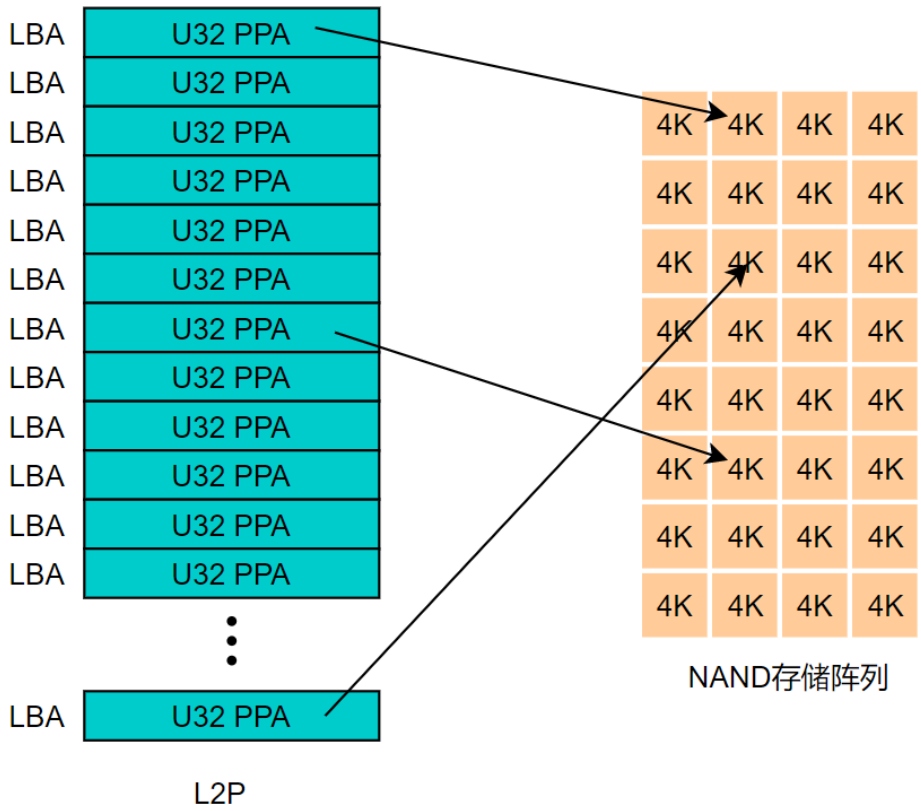


图 5-5 L2P 实现过程

L2P 可以做是一个一维数组，下标表示的是逻辑地址，内容表示的是物理地址，这样就实现了逻辑到物理地址的转换，显然 128GByte 的存储需要的 L2P 大小就是 128Mbyte，那每次查找这个表的时间太长了，为了缩短查找时间，就出现了二级 L2P，这一点非常类似于内存管理的段页式机制。同理，我们还可以实现三级 L2P 表来加快查找过程。

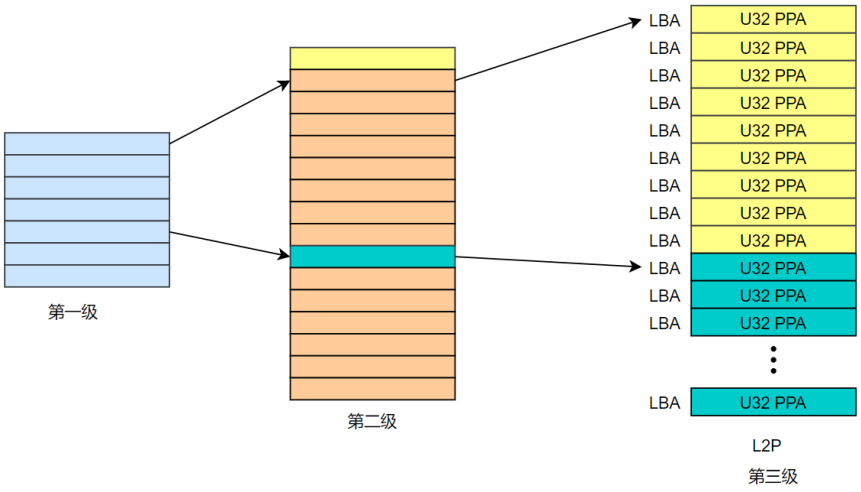


图 5-6 三级 L2P 示意图

[10] 乘以 4 是因为我们的表每个都是 32 位的

5.1.1.2 GC（垃圾回收）

上面提到过，GC 负责垃圾回收，当一个块中的页被擦除了，那么此页会被当做垃圾，当垃圾的数量非常多时，此时就需要开始做 GC，将有效数据搬移出来，然后存放到新的块中，再将老的块全部擦除掉。

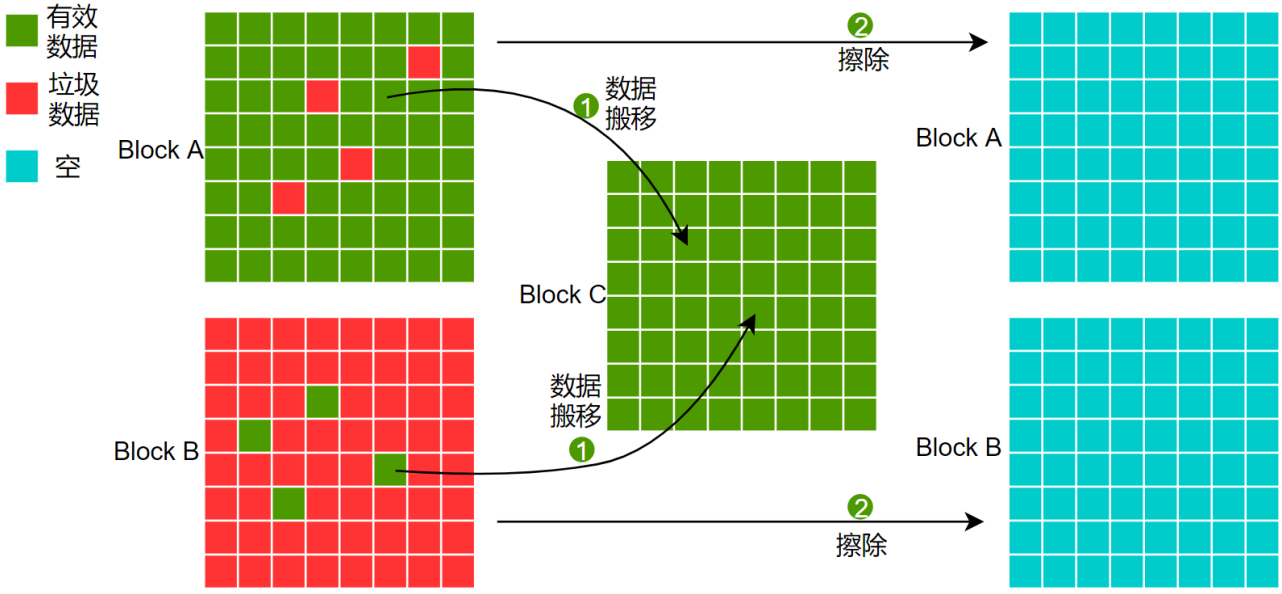


图 5-5 垃圾回收策略

上面的过程中需要先将有效数据放到缓存中，然后再将缓存中的数据写到一个新的块中，最后将原来的块整个擦除掉。由于 NAND 由寿命，因此 FTL 中还会对每个块的擦除次数做记录，每擦除一次加一。很显然，我们需要对每个块中的每个页情况进行记录，因此我们引入了 MB，下面我们来详细讲解 MB 的实现。

5.1.1.3 MB（位图映射）

像内存动态分配一样，我们需要标记该块中所有的有效数据，这样在做 GC 的时候就可以知道哪些数据是有效的，哪些数据是无效的。

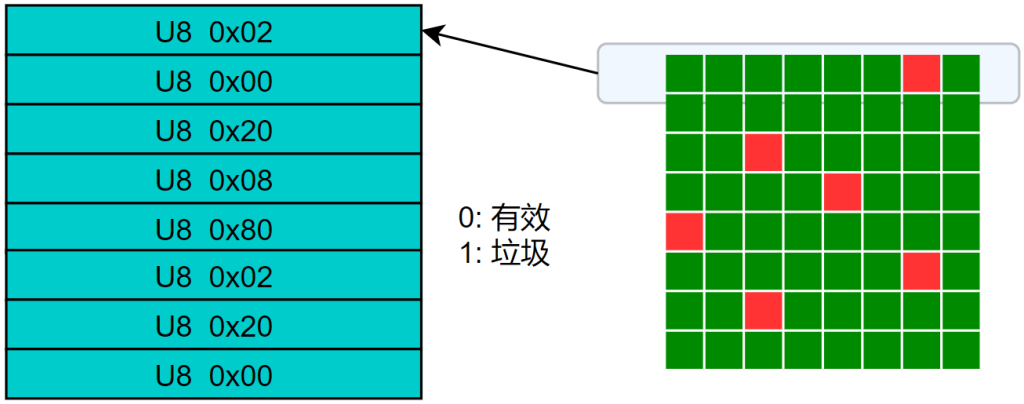


图 5-6 MB 实现策略

如上图所示，左边是一个一维数组，每个元素的每个 bit 位记录着 MB 的页情况，上图中 0 表示有效数据，1 表示垃圾数据，这种关系是一一对应的，这样，我们只需要去查找这个一维数组便可知道该块中页的垃圾数据情况了。

5.1.2 UFS 协议

SoC 与 UFS 通信需要通过 UFS 协议来实现，UFS 协议栈是由应用层、传输协议层、链路层（内连层）、物理层（接口层）。应用层为最上层，主要是各种 UFS 命令，其中目前大部分用的都是 SCSI 命令，该命令由 INCITS T10 制定；传输层由 JEDEC 制定，该层主要负责实现上层应用层的各个命令的处理；链路层主要负责将解析后的命令与下层进行对接；物理层负责将最后的数据发送给 Host 端；链路层和物理层使用的都是 MIPI 协议。除了应用层，还有一个设备管理接口，该设备管理器接口可以直接访问链路层，同时在应用层中，还有一类特殊的接口，就是任务管理，该接口主要对 UFS 中的任务进行管理。

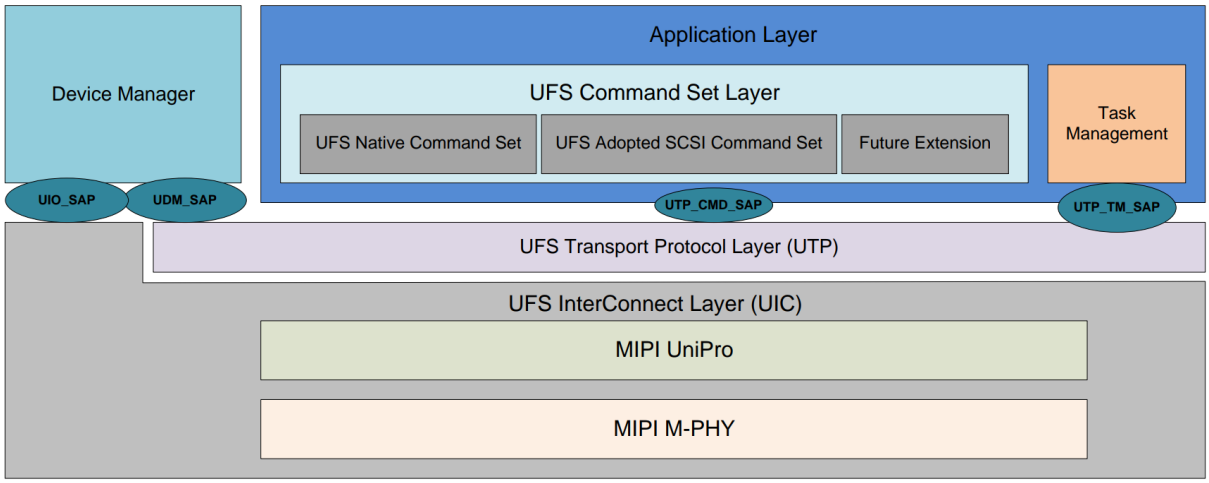


图 5-7 UFS 协议栈

对于 Linux 或者安卓驱动工程师最关心的便是应用层，因此，我们有必要 UFS 命令、设备管理层以及任务管理进行详细的了解。

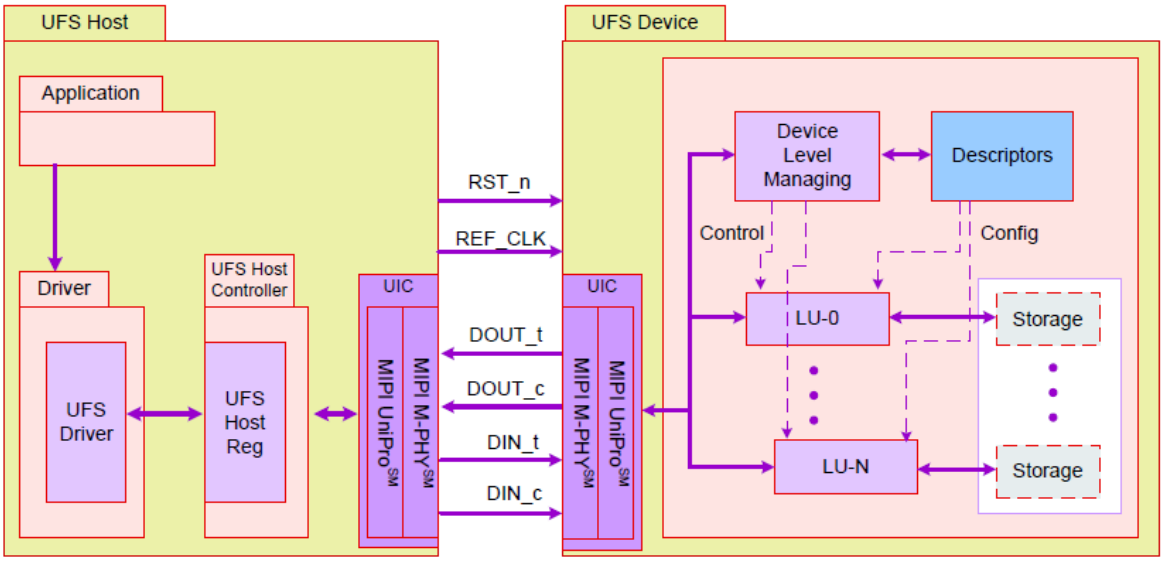


图 5-7 UFS 协议栈

5.1.2.1 LUN（Logical Unit Number）[11]

由于 UFS 的内存容量很大，因此我们将其拆分成了很多个组，每个组由很多个 block 构成，我们将该组命名为逻辑单元号，即 LUN。

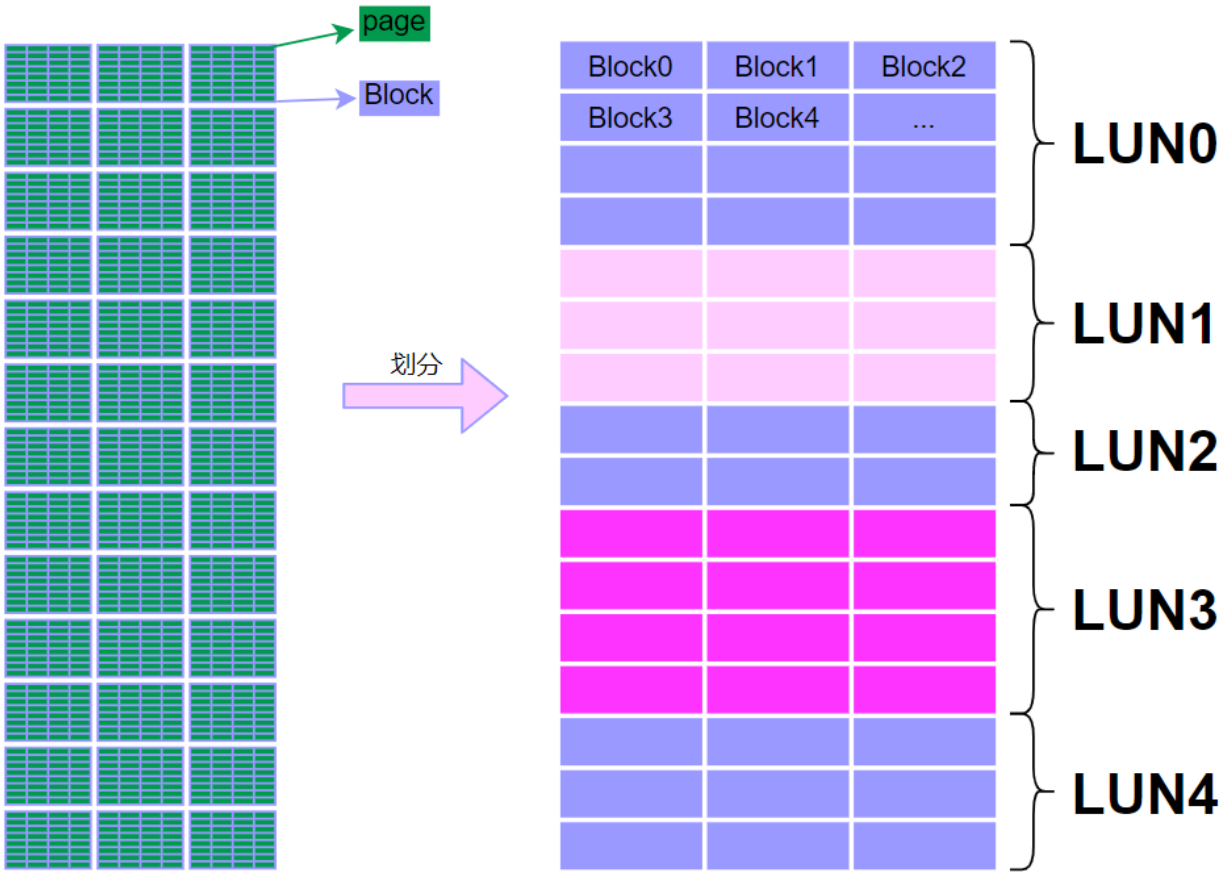


图 5-8 UFS 中的 LUN

每个 LUN 的大小是不固定的，在 UFS 芯片初始化的时候会进行配置，例如上图中，LUN0 的大小为 12 个 Block，LUN1 的大小为 9 个 LUN。而且这些 LUN 的大小并不是固定了就不再变化的，随着使用时间的延长，有些 Block 就会损坏，这样 LUN 的大小也会随之改变，当然，这些都是 UFS 内部的固件程序来完成的。

5.1.2.2 UFS 与 HOST 通信

UFS 归根结底还是属于一个存储设备，因此必然存在与 SoC 之间的数据通信，下图给出了 UFS 与 SoC（主机）之间的通信流程。在 SDIO 中有令牌包、数据包和响应包；同理，在 UFS 中数据通信也是靠传输包来实现的，其最小的数据包定义为 UPIU（UFS Protocol Interface Uint）。UPIU 由被分为多种类型，主要有 Command UPIU、RTT UPIU、DATA UPIU、Response UPIU。

[11] LUN 是对存储设备而言的，volume 是对主机而言的

- Command UPIU

  主机向 UFS 发送命令的 UPIU，该命令有读、写等。

- RTT UPIU

  设备发送给主机写数据传输中的数据响应，来告诉主机设备能够接收数据的能力，该 RTT UPIU 只存在于写数据时。

- DATA UPIU

  当主机与设备之间进行数据传输时，此时数据通过 DATA UPIU 来进行传输。

- Response UPIU

  每发送一笔命令传输结束后，设备都会向主机发送一个 Response UPIU 来结束本次传输会话。

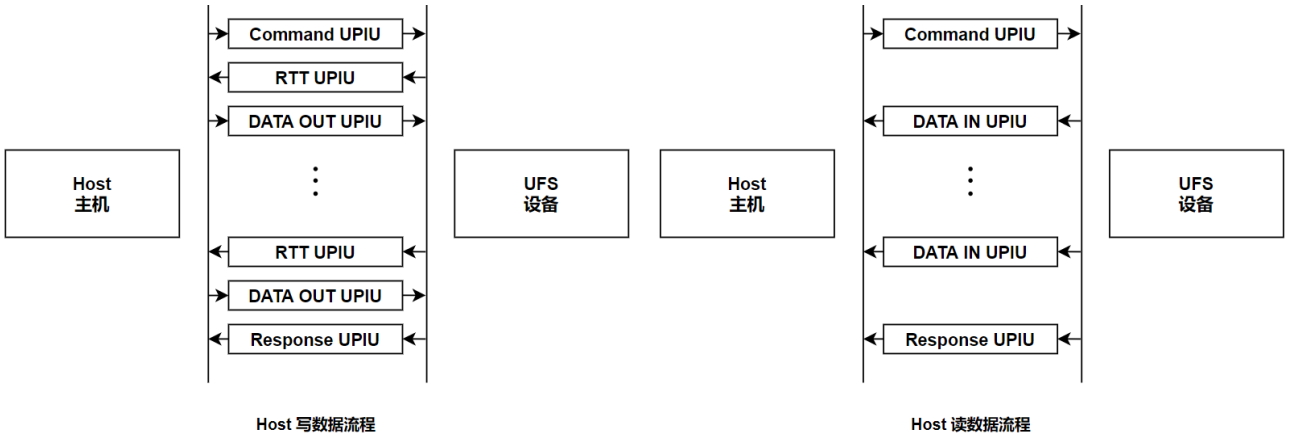


图 5-9 主机与设备之间的数据传输

➢ **写数据过程**

首先主机发送一个写命令包给设备，设备收到命令包后会发送一个 RTT 包给主机，告诉主机，此时设备能够接收数据的大小，此时主机就开始将该大小的数据发送给 UFS 设备，然后设备再次发送 RTT 告诉主机最多能够发送多少数据给设备，随后主机开始发送数据，依次反复，知道所有数据全部传输完毕，最后设备将返回一个 Response UPIU 来结束本次数据传输。

➢ **读数据过程**

首先主机发送一个读命令给设备，此时设备直接返回 DATA UPIU 将数据返回给主机，重复操作，最后设备将返回一个 Response UPIU 来结束本次数据传输。

5.1.2.3 SCSI 命令简介

上面提到过，UFS 协议中的应用程中有三种命令，实际上 UFS 并没有自己的命令，而是直接使用简化后的 SCSI 命令，因此我们有必要了解下 SCSI 命令。SCSI 是专门为硬盘设备定义的一套命令，因此 SCSI 命令并不是专门某一个设备命令，而是一类设备的标准命令。

SCSI 命令由操作码、命令特定参数、控制字节三部分组成，如下表所示，该表描述了一个完整的 SCSI 命令的格式，很显然这个命令并没有具体到各个事务，因此各个厂家可以根据自己的设备需求来制定自己的命令集合。

表 5-2 SCSI 命令描述块

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
0	操作码							
1	命令特定参数							
2								
…								
n-1								
n	控制字节							

表 5-3 SCSI 命令组

组	操作码	说明
0	00H~1FH	6 字节命令
1	20H~3FH	10 字节命令
2	40H~5FH	10 字节命令
3	60H~7FH	保留
4	80H~9FH	16 字节命令
5	A0H~BFH	12 字节命令
6	C0H~DFH	厂家自定
7	E0H~FFH	厂家自定

下面我们以一个 10 字节命令描述块来具体说明，如下表所示，从表中可以看到，命令操作码占据了一个字节，LUN 占了三个 bit 位，因此做多为 8 个 LUN，OPO 为可选的，FUA 为

表 5-4 10 字节命令描述块格式

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
0	READ（28H）或 WRITE（2AH）							
1	LUN			OPO	FUA	保留		Rel
2	(MSB) 逻辑块的起始地址							
4								
5								(LSB)
6	保留							
7	(MSB) 传输长度							
8								(LSB)
9	控制字节							

Force Unit Acess，0 表示从 cache 中获取数据，1 表示旁路 cache。紧接着有三个字节来描述逻辑块地址的起始地址，这样最大的逻辑块地址为 2 的 24 次方就是 16M，注意，这里的 16M 地址是指 LUN 指定的位置，并不是整个块设备，也就是说如果采样 10 字节命令来传输，则块设备的一个 LUN 的最大地址为 16M，如果这是 UFS，由于 UFS 的一个逻辑地址对应的是 4Kbyte，因此 UFS 的一个 LUN 的最大空间为 16*4KByte = 64GByte。

命令有多种不同的类型，如下表所示，M 表示必须指定的命令，其中 V 和 Z 是 SCSI 新标

准提出的。

表 5-5 命令类型说明

表示字母	说明
M	必须执行命令
O	选择命令，如果执行，必须依照标准
V	制造商指定的命令（仅针对 SCSI-2）
Z	某些设备类型必须实现该命令，其余的可选（SCSI-3 新标准）
R	保留命令（未定义）

这里我们先以一个简单的硬盘中的 SCSI 命令来说明下，可以看到，这些 SCSI 命令无非就是实现数据的读写和其他的一些特定操作。

表 5-6 硬盘设备的部分命令举例

操作码	名称	类型	说明
00H	TEST UNIT READY	M	检查 LUN 是否准备好接受命令
03H	REQUEST SENSE	M	请求传送详细检测数据
04H	FORMAT UNIT	M	格式化磁盘
07H	REASSIGN BLOCK	O	缺陷块分配到保留区
08H	READ(6)	M	6 字节读数据
0AH	WRITE(6)	M	6 字节写数据
0BH	SEEK(6)	O	6 字节寻找指定逻辑块
…	…		
28H	READ(10)	M	10 字节读数据
2AH	WRITE(10)	M	10 字节写数据
2BH	SEEK(10)	O	10 字节需寻找指定逻辑块
…	…		
5AH	MODE SENSE（10Byte）	O	读取设备参数

5.1.2.4 UFS 命令

对于 UFS 中的 SCSI 命也是类似的，我们来看下 UFS 中的 SCSI 命令，大致有四类命令，分别是查询类、读写类、管理类和杂项类。

- **查询类**

1. Inquiry 命令
2. Report 命令
3. Read Capacity 命令
4. Request Sense 命令
5. Test Unit Ready 命令
6. Send Diagnostic 命令

- **读写类**
  1. Pre-Fetch 命令
  2. Read 命令
  3. Write 命令
  4. Unmap 命令
  5. Format Uint 命令
  6. Synchronize Cache 命令
  7. Verify 命令
  8. Security Protocol Out 命令
  9. Security Protocol In 命令
- **管理类**
  1. Start Stop Unit 命令
  2. Mode Sense 命令
  3. Mode Select 命令
- **杂项类**
  1. Read Buffer 命令
  2. Write Buffer 命令

5.1.3 UFS HCI

编写 Linux 中 UFS 的驱动最重要的是了解 UFS HCI，即主机控制器。

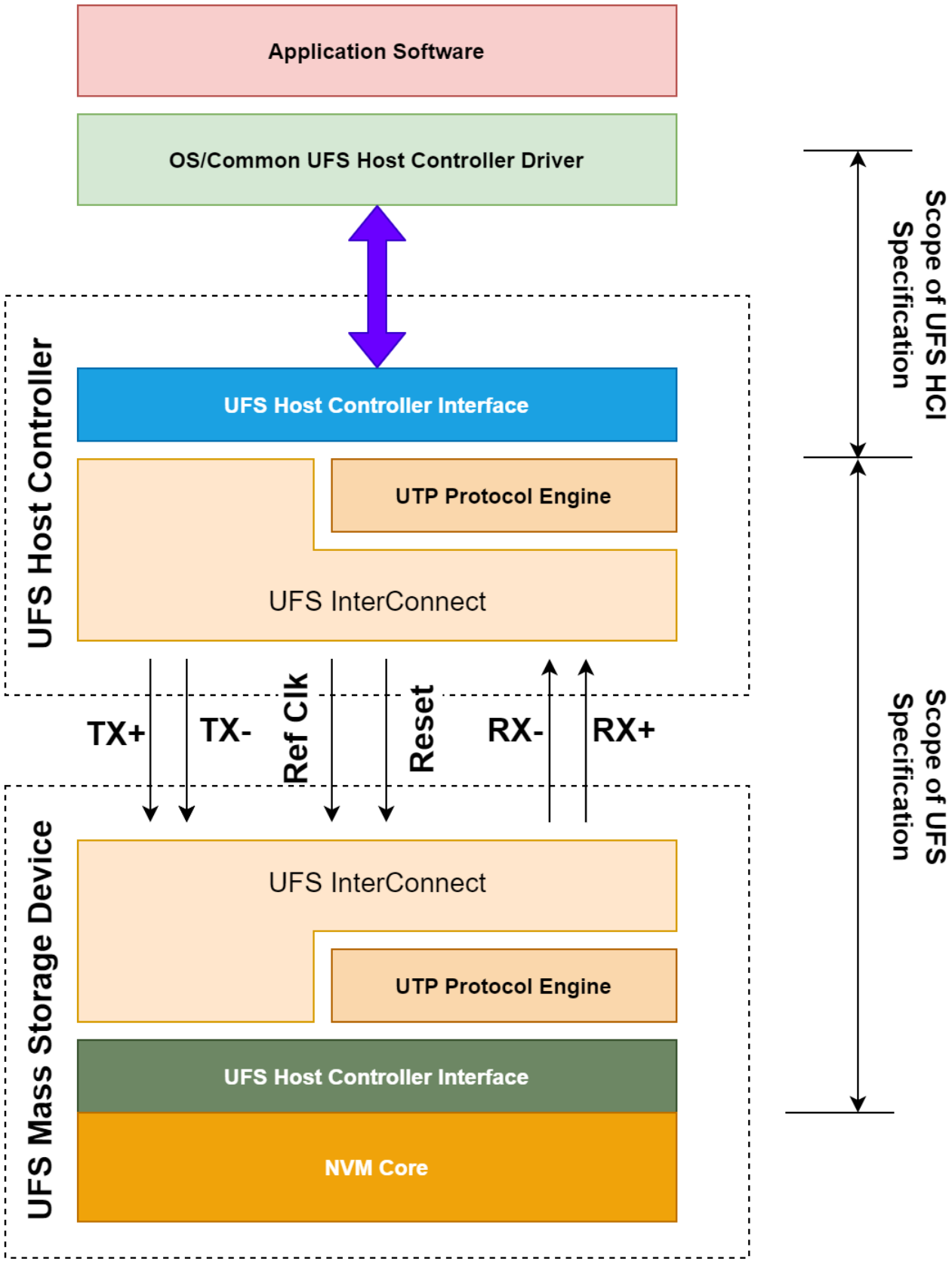


图 5-10 UFS HCI 与 UFS Device 连接

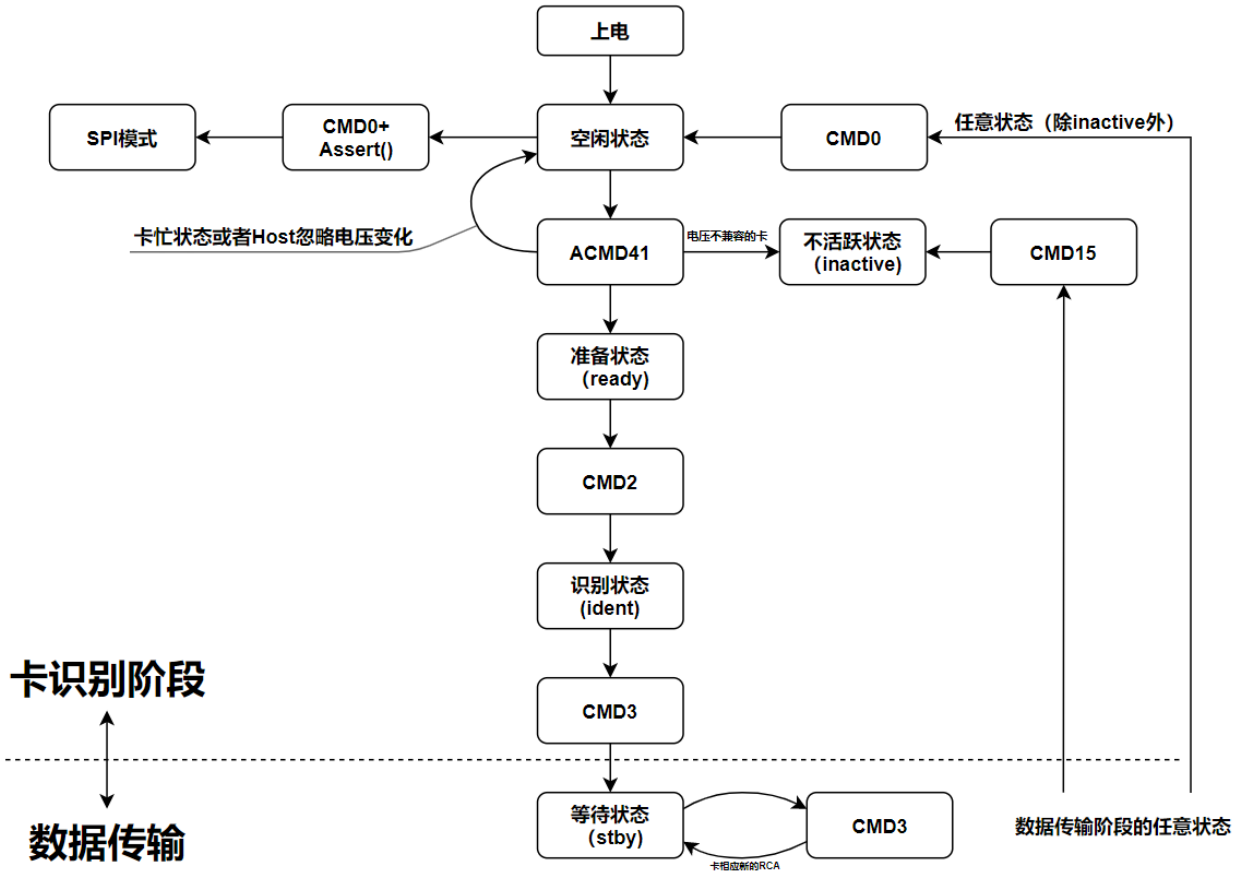


图 3-4 SD 卡的识别阶段

第六章 内存管理

对于任何人一个操作系统而言，内存管理非常重要，那什么是内存管理呢，内存管理的意义何在，带着这两个问题，我们来开始探索内存管理。

6.1 C 语言中的内存管理

在 C 语言的应用开发过程中，我们总是使用 malloc 函数来分配一个内存，