# 《嵌入式系统》

## （第十二讲）

厦门大学信息学院软件工程系 曾文华

2024年12月3日

- 第1章：嵌入式系统概述
- 第2章：ARM处理器和指令集
- 第3章：嵌入式Linux操作系统
- 第4章：嵌入式软件编程技术
- 第5章：开发环境和调试技术
- 第6章：Boot Loader技术
- 第7章：ARM Linux内核
- 第8章：文件系统
- 第9章：设备驱动程序设计基础
- 第10章：字符设备和驱动程序设计
- 第11章：Android操作系统（增加）
- 第12章：块设备和驱动程序设计
- 第13章：网络设备驱动程序开发
- 第14章：嵌入式GUI及应用程序设计

# 第12章　块设备和驱动程序设计

- 12.1　块设备驱动程序设计概要
- 12.2　Linux块设备驱动相关数据结构与函数
- 12.3　块设备的注册与注销
- 12.4　块设备初始化与卸载
- 12.5　块设备操作
- 12.6　请求处理
- 12.7　MMC卡驱动

- 块设备是Linux三大设备之一（另外两种是字符设备，网络设备），块设备也是通过/dev下的文件系统节点访问。

- 块设备的数据存储单位是块，块的大小通常为512B至32KB不等。

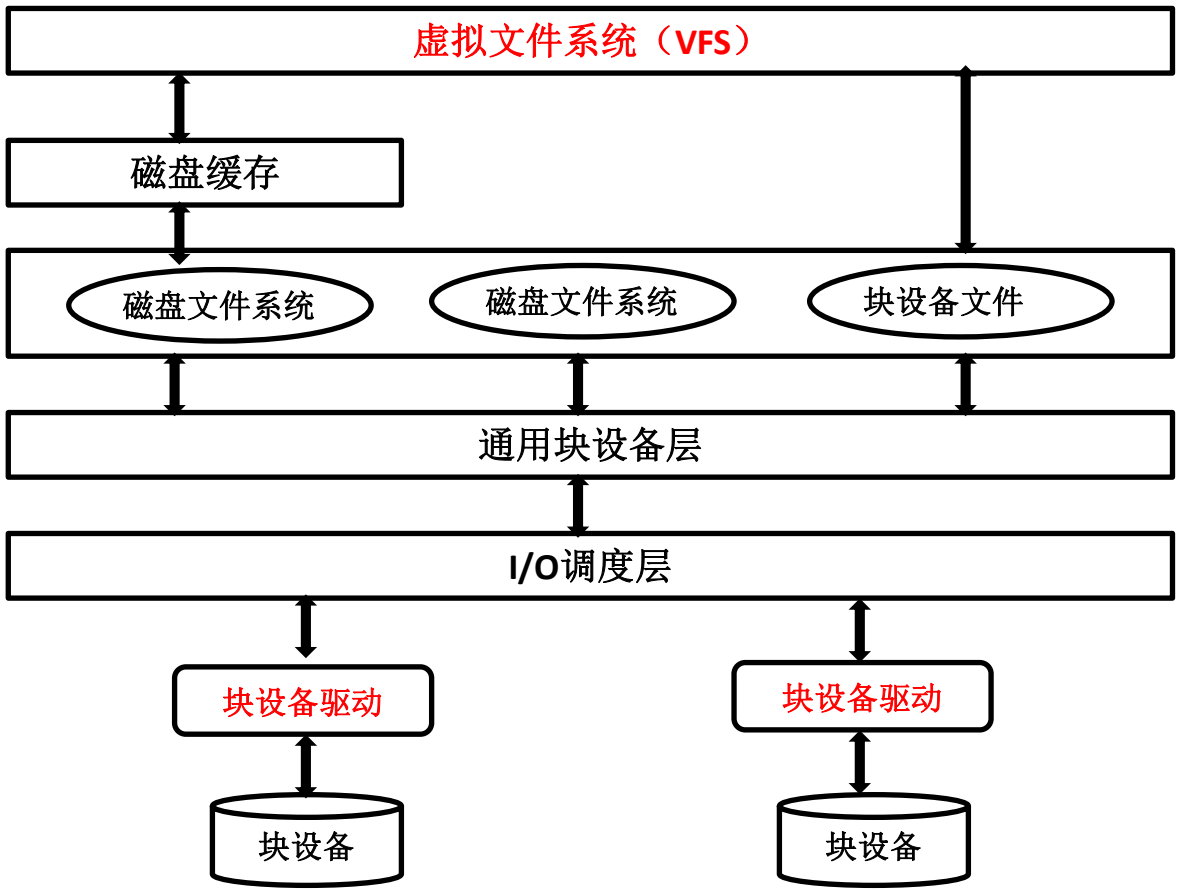
- 块设备每次能传输一个或多个块，支持随机访问，并且采用了缓存技术。

- 常见的块设备包括IDE硬盘、SCSI硬盘、CD-ROM等等。
  - IDE：Integrated Device Electronics，集成磁盘电子接口
  - SCSI：Small Computer System Interface，小型计算机系统接口

# 12.1 块设备驱动程序设计概要

- 块设备驱动在虚拟文件系统（VFS）中的位置：

VFS（Virtual File System，虚拟文件系统）的作用就是采用标准的Unix系统调用读写位于不同物理介质上的不同文件系统，即为各类文件系统提供了一个统一的操作界面和应用编程接口。VFS是一个可以让open()、read()、write()等系统调用不用关心底层的存储介质和文件系统类型就可以工作的粘合层。



- **12.1.1 块设备的数据交换方式**

  - 块设备以块（512B至32KB）为单位进行读写；字符设备以字节为单位进行读写。

  - 块设备有对应的缓冲区，并使用了请求队列对I/O请求进行管理，块设备支持随机访问；字符设备只能顺序访问。

- **11.1.2 块设备读写请求**

  - 对块设备的读写都是通过请求实现的。

  - Linux中每一个块设备都有一个I/O请求队列，每个请求队列都有调度器的插口，调度器可以实现对请求队列里请求的合理组织，如合并临近请求，调整请求完成顺序等。

  - Linux 2.6内核有4个I/O调度器（Scheduler）：
    ① No-op I/O scheduler：实现了一个简单的FIFO队列；
    ② Anticipatory I/O scheduler：是目前内核中默认的I/O调度器；
    ③ Deadline I/O scheduler：是针对Anticipatory I/O scheduler的缺点进行改善而来的；
    ④ CFQ I/O schedule：为系统内的所有任务分配相同的带宽，提供一个公平的工作环境，它比较适合桌面环境。

# 12.2 Linux块设备驱动相关数据结构与函数

## • 12.2.1 gendisk结构

– gendisk（通用磁盘）数据结构：struct gendisk。在Linux内核中，gendisk数据结构表示是一个独立磁盘设备或者一个分区。

– Linux提供了一组函数接口来操作gendisk数据结构：

① 分配gendisk
- struct gendisk *alloc_disk(int minors);

② 增加（注册）gendisk
- void add_disk(struct gendisk *disk);

③ 释放（删除）gendisk
- void del_gendisk(struct gendisk *gd);

④ 引用计数
- 减少引用计数：get_disk();
- 增加引用计数：put_disk();

⑤ 设置和查看磁盘容量
- 设置磁盘容量：void set_capacity(struct gendisk *disk, sector_t size);
- 查看磁盘容量：sector_t get_capacity(struct gendisk *disk);

```
struct gendisk {
        int major;                              /* 主设备号 */
        int first_minor;                        /*第1 个次设备号*/
        int minors;                             /* 最大的次设备数，如果不能分区，则为1*/
        char disk_name[32];                     /* 设备名称 */
        struct hd_struct **part;                /* 磁盘上的分区信息 */
        struct block_device_operations *fops;   /*块设备操作结构体*/
        struct request_queue *queue;            /*请求队列*/
        void *private_data;                     /*私有数据*/
        sector_t capacity;                      /*扇区数，512 字节为1 个扇区*/
        int flags;
        char devfs_name[64];
        int number;
        struct device *driverfs_dev;
        struct kobject kobj;
        struct timer_rand_state *random;
        int policy;
        atomic_t sync_io;                       /* RAID */
        unsigned long stamp;
        int in_flight;
#ifdef CONFIG_SMP
        struct disk_stats *dkstats;
#else
        struct disk_stats dkstats;
#endif
};
```

**struct gendisk（通用磁盘）结构体**

- **12.2.2 request结构**
  - 块设备的读写都是通过请求实现的。
  - 请求数据结构： **struct request**

```
struct request {
        struct list_head queue;
        int elevator_sequence;
        volatile int rq_status;
        #define RQ_INACTIVE             (-1)
        #define RQ_ACTIVE               1
        #define RQ_SCSI_BUSY            0xffff
        #define RQ_SCSI_DONE            0xfffe
        #define RQ_SCSI_DISCONNECTING   0xffe0
        kdev_t rq_dev;
        int cmd;
        int errors;
        unsigned long sector;
        unsigned long nr_sectors;
        unsigned long hard_sector, hard_nr_sectors;
        unsigned int nr_segments;
        unsigned int nr_hw_segments;
        unsigned long current_nr_sectors;
        void * special;
        char * buffer;
        struct completion * waiting;
        struct buffer_head * bh;
        struct buffer_head * bhtail;
        request_queue_t *q;
};
```

**struct request（请求）结构体**

- **12.2.3 request_queue队列**

  – 每一个块设备都有一个I/O请求队列。

  – 请求队列数据结构：struct request_queue

  – 请求队列数据结构包括：

    ① 请求队列的初始化和清除；
    ② 提取和删除请求；
    ③ 队列的参数设置；
    ④ 内核通告。

**struct request_queue**
（请求队列）结构体

```
struct request_queue
{
        struct request_list     rq[2];
        struct list_head        queue_head;
        elevator_t              elevator;
        request_fn_proc         * request_fn;
        merge_request_fn        * back_merge_fn;
        merge_request_fn        * front_merge_fn;
        merge_requests_fn       * merge_requests_fn;
        make_request_fn         * make_request_fn;
        plug_device_fn          * plug_device_fn;
        void                    * queuedata;
        struct tq_struct        plug_tq;
        char                    plugged;
        char                    head_active;
        spinlock_t              queue_lock;
        wait_queue_head_t       wait_for_request;
};
```

- **12.2.4 bio结构**

  - bio（block I/O，块I/O）是Linux内核中通用块层的一个核心数据结构，它描述了块设备的I/O操作，联系了内存缓冲区与块设备。

  - bio是底层对部分块设备的I/O请求描述，其包含驱动程序执行请求所需的全部信息。

```
struct bio {
        sector_t                bi_sector;
        struct bio              *bi_next;
        struct block_device     *bi_bdev;
        unsigned long           bi_flags;
        unsigned long           bi_rw;
        unsigned short          bi_vcnt;
        unsigned short          bi_idx;
        unsigned short          bi_phys_segments;
        unsigned short          bi_hw_segments;
        unsigned int            bi_size;
        unsigned int            bi_hw_front_size;
        unsigned int            bi_hw_back_size;
        unsigned int            bi_max_vecs;
        struct bio_vec          *bi_io_vec;
        bio_end_io_t            *bi_end_io;
        atomic_t                bi_cnt;
        void                    *bi_private;
        bio_destructor_t        *bi_destructor;
};
```

**struct bio（块I/O）结构体**

# 12.3 块设备的注册和注销

- 块设备的注册：
  - int register_blkdev(unsigned int major, const char *name);
    - major：主设备号
    - name：设备名

- 块设备的注销
  - int unregister_blkdev(unsigned int major, const char* name);
    - major：主设备号
    - name：设备名

# 12.4 块设备的初始化和卸载

- 块设备的初始化过程主要完成以下的工作：
  ① 注册块设备及块设备驱动程序；
  ② 分配、初始化、绑定请求队列（如果使用请求队列的话）；
  ③ 分配、初始化gendisk，为相应的成员赋值并添加gendisk；
  ④ 其他初始化工作，如申请缓存区，设置硬件尺寸（不同设备，有不同的处理）。

- 块设备的卸载过程刚好与初始化过程相反：
  ① 删除请求队列；
  ② 撤销gendisk的引用，并删除gendisk；
  ③ 释放缓冲区，撤销对块设备的应用，注销块设备驱动。

# 12.5 块设备操作

- 块设备操作数据结构：**struct block_device_operations**
  - 字符设备文件操作数据结构： **struct file_operations**

**struct block_device_operations**
块设备操作结构体

```
struct block_device_operations {
        int (*open) (struct block_device *, fmode_t);
        int (*release) (struct gendisk *, fmode_t);
        int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
        int (*direct_access) (struct block_device *, sector_t,  void **, unsigned long *);
        int (*media_changed) (struct gendisk *);
        int (*revalidate_disk) (struct gendisk *);
        int (*getgeo)(struct block_device *, struct hd_geometry *);
        struct module *owner;
};
```

## ① 打开和释放

- int (*open) (struct block_device *, fmode_t);
- int (*release) (struct gendisk *, fmode_t);

## ② I/O操作

- int (*ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
- int (*locked_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);
- int (*compat_ioctl) (struct block_device *, fmode_t, unsigned, unsigned long);

## ③ 介质改变

- int (*media_changed) (struct gendisk *);

## ④ 使介质有效

- int (*revalidate_disk) (struct gendisk *);

## ⑤ 获得驱动器信息

- int (*getgeo)(struct block_device *, struct hd_geometry *);

## ⑥ 模块指针

- struct module *owner;

# 12.6 请求处理

- 块设备没有read和write操作函数。
- 对块设备的读写是通过请求函数完成的。
- 请求处理分为两种情况：
  - （1）使用请求队列
    ① 请求函数
    ② 通告内核
    ③ 屏障请求和不可重试请求
  - （2）不使用请求队列

# 12.7 MMC卡驱动

- ## 12.7.1 MMC/SD芯片介绍

  - MMC卡（Multi-Media Card，多媒体卡）：1997年由西门子公司和SanDisk公司共同开发，基于东芝公司的NAND Flash技术。

  - SD卡（Secure Digital Memory Card，安全数码卡）：SD卡是由松下电器、东芝和SanDisk联合推出，1999年8月发布。

  - SD卡的数据传送和物理规范由MMC卡发展而来，大小和MMC卡（32mm×24mm×1.4mm）差不多，尺寸为32mm x 24mm x 2.1mm，长宽和MMC卡一样，只是比MMC卡厚了0.7mm，以容纳更大容量的存贮单元。

MMC/SD卡正面


MMC卡反面

MMC卡

SD卡反面

SD卡

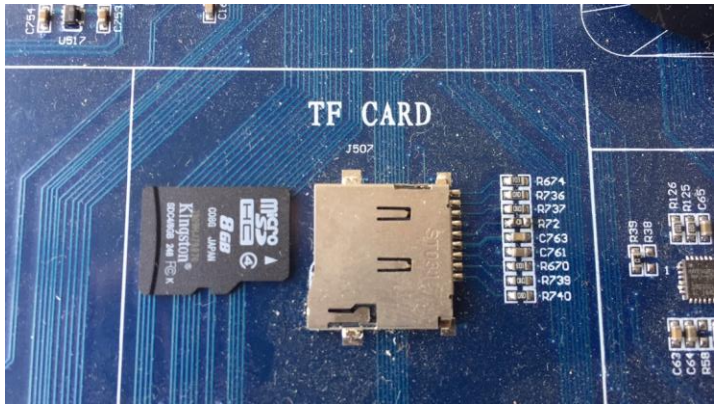
- **MMC卡、SD卡的管脚定义**

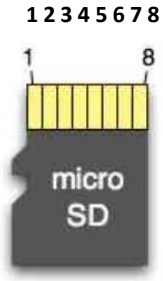

MMC卡


SD卡

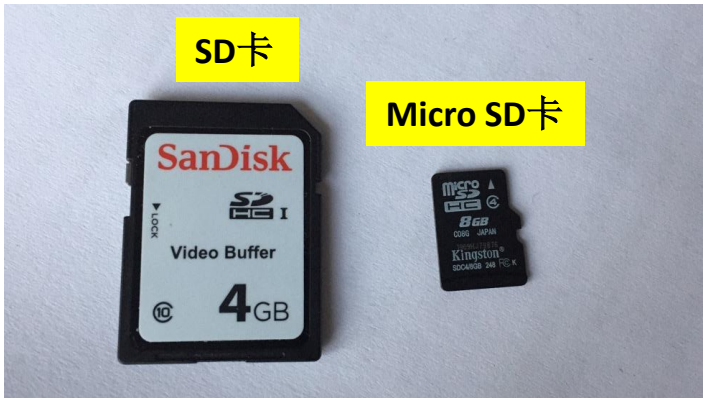
1.CD DAT3 I/O/PP 卡监测数据位 3
2.CMD PP 命令/回复
3. Vss S 地
4.Vcc S 供电电压
5.CLK I 时钟
6.Css2 S 地
7.DAT0 I/O/PP 数据位 0
8.DAT1 I/O/PP 数据位 1
9.DAT2 I/O/PP 数据位 2

- **Micro SD卡（TF卡）**
  - **Micro SD Card，原名Trans-flash Card（TF卡），2004年正式更名为Micro SD Card，由SanDisk（闪迪）公司发明，主要用于移动电话。**




| Pin | SD | SPI |
|---|---|---|
| 1 | DAT2 | X |
| 2 | CD/DAT3 | CS |
| 3 | CMD | DI |
| 4 | VDD | VDD |
| 5 | CLK | SCLK |
| 6 | VSS | VSS |
| 7 | DAT0 | DO |
| 8 | DAT1 | X |




- **MMC卡的工作模式：**

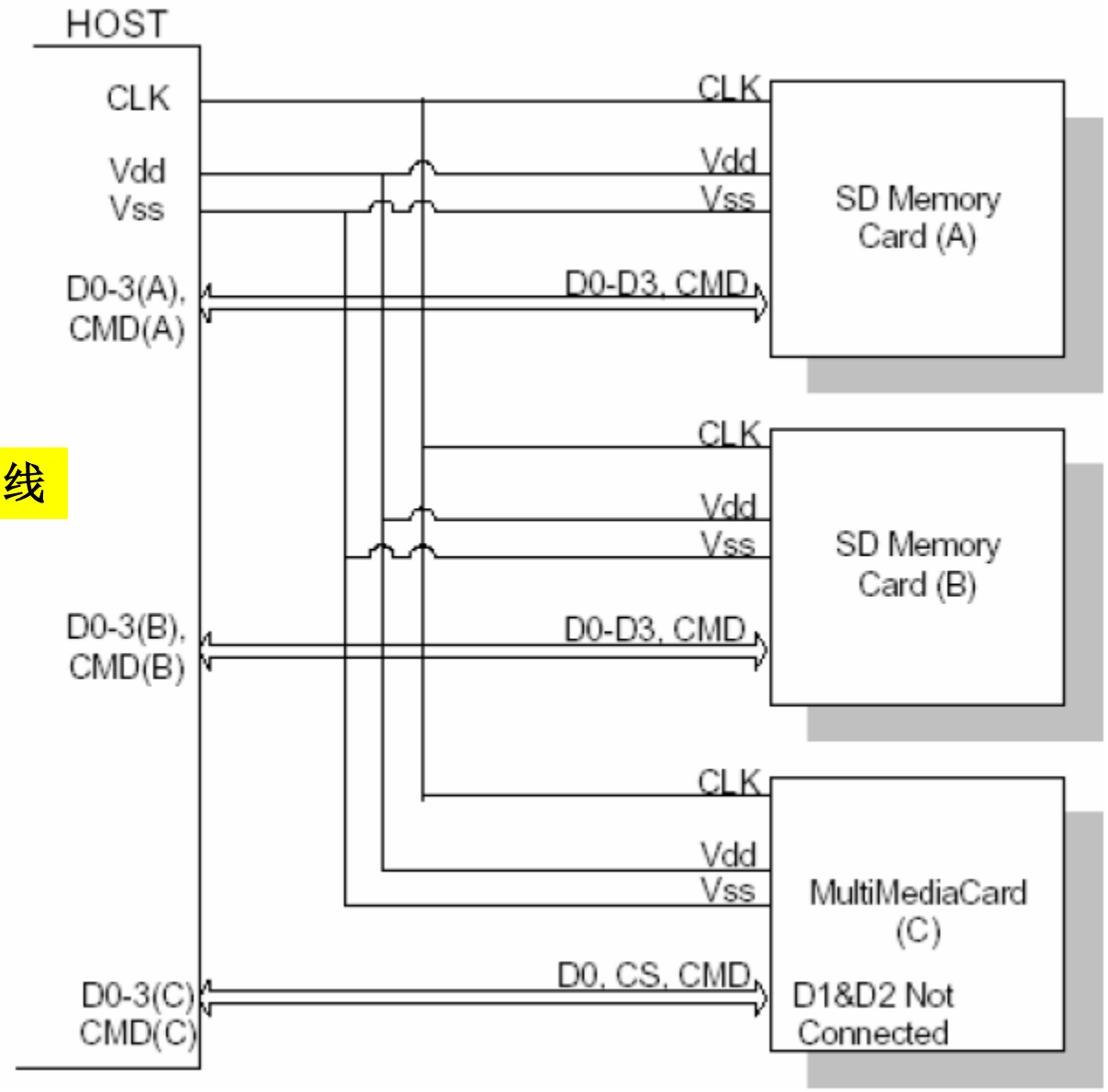
  ① **MMC模式**：标准的默认模式。

  ② **SPI模式**（Serial Peripheral Interface，串行外设接口）：用于只需要小数量的卡（通常是一个）和低数据传输率。

- **SD卡的工作模式：**

  ① **SD模式**：9根信号线：CLK、CMD、DAT0-DAT3、Vcc（+5V）、Vss（GND）、Css2（GND）。

  ② **SPI模式**（Serial Peripheral Interface，串行外设接口）：7根信号线：CS、CLK、MISO（DATAOUT）、MOSI（DATAIN）、 、Vcc（+5V）、Vss（GND）、Css2（GND）。




MMC/SD总线与MMC卡/SD卡的连接

# • 12.7.2 MMC/SD卡驱动结构

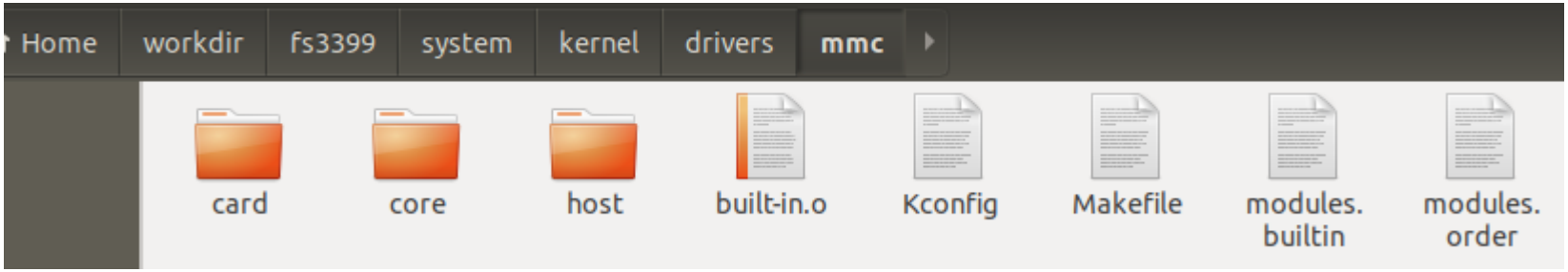
– MMC/SD驱动层次：

① 块设备驱动层（drivers/mmc/card）：该层实现块设备驱动，为上层提供块设备的操作功能。

② MMC/SD核心层（drivers/mmc/core）：该层主要完成MMC/SD规范和协议的实现。

③ MMC/SD接口层（drivers/mmc/host）：该层主要实现Host接口的驱动，并为上层提供操作接口。

– 块设备驱动层、MMC/SD核心层，与具体的硬件平台无关；MMC/SD接口层根据不同的硬件和不同的控制器有不同的实现。




MMC/SD卡驱动程序位于：/home/linux/workdir/fs3399/system/kernel/drivers/mmc

- **12.7.3 MMC卡块设备驱动分析**

  - drivers/mmc/card/block.c
  - drivers/mmc/card/queue.c
  - 主要完成：
    ① 注册与注销
    ② 设备加载与卸载
    ③ 设备的打开与释放
    ④ MMC驱动的请求处理函数

    block.c

    queue.c

**/home/linux/workdir/fs3399/system/kernel/drivers/mmc/card/block.c**

```
/*
 * Block driver for media (i.e., flash cards)
 *
 * Copyright 2002 Hewlett-Packard Company
 * Copyright 2005-2008 Pierre Ossman
 *
 * Use consistent with the GNU GPL is permitted,
 * provided that this copyright notice is
 * preserved in its entirety in all copies and derived works.
 *
 * HEWLETT-PACKARD COMPANY MAKES NO WARRANTIES, EXPRESSED OR IMPLIED,
 * AS TO THE USEFULNESS OR CORRECTNESS OF THIS CODE OR ITS
 * FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 * Many thanks to Alessandro Rubini and Jonathan Corbet!
 *
 * Author:  Andrew Christian
 *          28 May 2002
 */
#include <linux/moduleparam.h>
#include <linux/module.h>
#include <linux/init.h>

#include <linux/kernel.h>
#include <linux/fs.h>
#include <linux/slab.h>
#include <linux/errno.h>
#include <linux/hdreg.h>
#include <linux/kdev_t.h>
#include <linux/blkdev.h>
#include <linux/mutex.h>
#include <linux/scatterlist.h>
#include <linux/string_helpers.h>
#include <linux/delay.h>
#include <linux/capability.h>
#include <linux/compat.h>
#include <linux/pm_runtime.h>

#include <trace/events/mmc.h>

#include <linux/mmc/ioctl.h>
#include <linux/mmc/card.h>
#include <linux/mmc/host.h>
#include <linux/mmc/mmc.h>
#include <linux/mmc/sd.h>

#include <asm/uaccess.h>

#include "queue.h"

MODULE_ALIAS("mmc:block");
#ifdef MODULE_PARAM_PREFIX
#undef MODULE_PARAM_PREFIX
#endif
#define MODULE_PARAM_PREFIX "mmcblk."

#define INAND_CMD38_ARG_EXT_CSD  113
#define INAND_CMD38_ARG_ERASE    0x00
#define INAND_CMD38_ARG_TRIM     0x01
#define INAND_CMD38_ARG_SECERASE 0x80
#define INAND_CMD38_ARG_SECTRIM1 0x81
#define INAND_CMD38_ARG_SECTRIM2 0x88
#define MMC_BLK_TIMEOUT_MS  (10 * 60 * 1000)        /* 10 minute timeout */
#define MMC_SANITIZE_REQ_TIMEOUT 240000
#define MMC_EXTRACT_INDEX_FROM_ARG(x) ((x & 0x00FF0000) >> 16)

#define mmc_req_rel_wr(req)     ((req->cmd_flags & REQ_FUA) && \
                                  (rq_data_dir(req) == WRITE))
#define PACKED_CMD_VER  0x01
#define PACKED_CMD_WR   0x02

static DEFINE_MUTEX(block_mutex);

/*
 * The defaults come from config options but can be overriden by module
 * or bootarg options.
 */
static int perdev_minors = CONFIG_MMC_BLOCK_MINORS;

/*
```

## struct block_device_operations 块设备操作结构体

```
static const struct block_device_operations mmc_bdops = {
        .open                   = mmc_blk_open,
        .release                = mmc_blk_release,
        .getgeo                 = mmc_blk_getgeo,
        .owner                  = THIS_MODULE,
        .ioctl                  = mmc_blk_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl           = mmc_blk_compat_ioctl,
#endif
};
```

# mmc_blk_open 打开设备

```
static int mmc_blk_open(struct block_device *bdev, fmode_t mode)
{
        struct mmc_blk_data *md = mmc_blk_get(bdev->bd_disk);
        int ret = -ENXIO;

        mutex_lock(&block_mutex);
        if (md) {
                if (md->usage == 2)
                        check_disk_change(bdev);
                ret = 0;

                if ((mode & FMODE_WRITE) && md->read_only) {
                        mmc_blk_put(md);
                        ret = -EROFS;
                }
        }
        mutex_unlock(&block_mutex);

        return ret;
}
```

## mmc_blk_release 释放设备

```
static void mmc_blk_release(struct gendisk *disk, fmode_t mode)
{
        struct mmc_blk_data *md = disk->private_data;

        mutex_lock(&block_mutex);
        mmc_blk_put(md);
        mutex_unlock(&block_mutex);
}
```

# mmc_blk_ioctl 设备控制

```
static int mmc_blk_ioctl(struct block_device *bdev, fmode_t mode,
        unsigned int cmd, unsigned long arg)
{
        switch (cmd)
        {
                case MMC_IOC_CMD:
                        return mmc_blk_ioctl_cmd(bdev, (struct mmc_ioc_cmd __user *)arg);

                case MMC_IOC_MULTI_CMD:
                        return mmc_blk_ioctl_multi_cmd(bdev, (struct mmc_ioc_multi_cmd __user *)arg);

                default:
                        return -EINVAL;
        }
}
```

# mmc_blk_compat_ioctl 设备控制

```
#ifdef CONFIG_COMPAT

static int mmc_blk_compat_ioctl(struct block_device *bdev, fmode_t mode,
        unsigned int cmd, unsigned long arg)
{
        return mmc_blk_ioctl(bdev, mode, cmd, (unsigned long)
compat_ptr(arg));
}

#endif
```

# mmc_blk_init 模块初始化

```
static int __init mmc_blk_init(void)
{
        int res;

        if (perdev_minors != CONFIG_MMC_BLOCK_MINORS)
                pr_info("mmcblk: using %d minors per device\n", perdev_minors);

        max_devices = 256 / perdev_minors;

        res = register_blkdev(MMC_BLOCK_MAJOR, "mmc");
        if (res)
                goto out;

        res = mmc_register_driver(&mmc_driver);
        if (res)
                goto out2;

        return 0;
 out2:
        unregister_blkdev(MMC_BLOCK_MAJOR, "mmc");
 out:
        return res;
}
```

## mmc_blk_exit 模块退出

```
static void __exit mmc_blk_exit(void)
{
        mmc_unregister_driver(&mmc_driver);
        unregister_blkdev(MMC_BLOCK_MAJOR, "mmc");
}
```

```
module_init(mmc_blk_init);
module_exit(mmc_blk_exit);
```

/home/linux/workdir/fs3399/system/kernel/drivers/mmc/card/queue.c

MMC驱动的请求处理函数

```
/*
 *  linux/drivers/mmc/card/queue.c
 *
 *  Copyright (C) 2003 Russell King, All Rights Reserved.
 *  Copyright 2006-2007 Pierre Ossman
 *
 */
#include <linux/slab.h>
#include <linux/module.h>
#include <linux/blkdev.h>
#include <linux/freezer.h>
#include <linux/kthread.h>
#include <linux/scatterlist.h>
#include <linux/dma-mapping.h>

#include <linux/mmc/card.h>
#include <linux/mmc/host.h>
#include <linux/sched/rt.h>
#include "queue.h"

#define MMC_QUEUE_BOUNCESZ	65536

/*
 * Prepare a MMC request. This just filters out odd stuff.
 */
static int mmc_prep_request(struct request_queue *q, struct request *req)
{
	struct mmc_queue *mq = q->queuedata;

	/*
	 * We only like normal block requests and discards.
	 */
	if (req->cmd_type != REQ_TYPE_FS && !(req->cmd_flags & REQ_DISCARD)) {
		blk_dump_rq_flags(req, "MMC bad request");
		return BLKPREP_KILL;
	}

	if (mq && (mmc_card_removed(mq->card) || mmc_access_rpmb(mq)))
		return BLKPREP_KILL;

	req->cmd_flags |= REQ_DONTPREP;

	return BLKPREP_OK;
}

static int mmc_queue_thread(void *d)
{
	struct mmc_queue *mq = d;
	struct request_queue *q = mq->queue;
	struct sched_param scheduler_params = {0};

	scheduler_params.sched_priority = 1;

	sched_setscheduler(current, SCHED_FIFO, &scheduler_params);

	current->flags |= PF_MEMALLOC;

	down(&mq->thread_sem);
	do {
		struct request *req = NULL;
		unsigned int cmd_flags = 0;

		spin_lock_irq(q->queue_lock);
		set_current_state(TASK_INTERRUPTIBLE);
		req = blk_fetch_request(q);
		mq->mqrq_cur->req = req;
		spin_unlock_irq(q->queue_lock);

		if (req || mq->mqrq_prev->req) {
			set_current_state(TASK_RUNNING);
			cmd_flags = req ? req->cmd_flags : 0;
			mq->issue_fn(mq, req);
			cond_resched();
			if (mq->flags & MMC_QUEUE_NEW_REQUEST) {
				mq->flags &= ~MMC_QUEUE_NEW_REQUEST;
				continue; /* fetch again */
```

# 小结

- 主要介绍嵌入式系统中块设备驱动的开发。
- 在块设备的I/O操作中，始终围绕着请求来进行的。

# 进一步探索

- 字符设备和块设备之间有什么主要区别？
- 块设备中请求处理函数的作用？

# Thanks