Linux内核MTD驱动程序与SD卡驱动程序

flash闪存设备和SD插卡设备是嵌入式设备用到的主要存储设备它们相当于PC机的硬盘在嵌入设备特别是手持设备中flash
嵌入设备的存储设备的空间划分及所有逻辑设备和文件系统示例列出如下图

图嵌入设备的存储空间划分及文件系统示例图

在嵌入设备上的flash芯片上blob和zImage直接按内存线性地址存储管理对于flash芯片上留出的供用户使用的存储空间使用
在嵌入设备上的MMC/SD插卡则由MMCBLOCK驱动程序和VFAT文件系统进行存储管理本章分析了MTD设备和MMC/SD驱

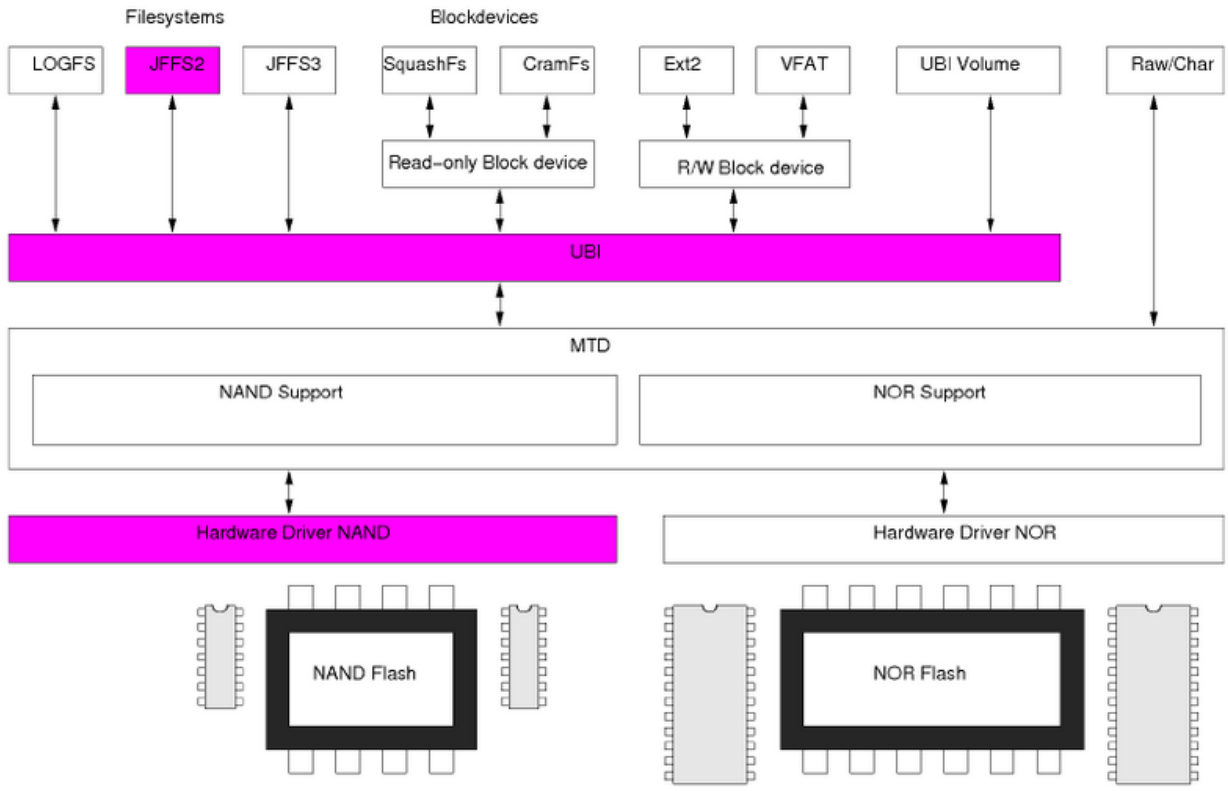

Figure 3-1. UBI/MTD Integration

MTD内存技术设备

Linux中MTD子系统在系统的硬件驱动程序和文件系统之间提供通用接口在MTD上常用的文件文件系统是JFFS2日志闪存文
Flash File
System JFFS2用于微型嵌入式设备的原始闪存芯片的文件系统JFFS2文件系统是日志结构化的这意味着它基本上是一长列
— 可能是文件的名称也许是一些数据与Ext2文件系统相比JFFS2因为有以下这些优点

JFFS2在扇区级别上执行闪存擦除写读操作要比Ext2文件系统好JFFS2提供了比Ext2fs更好的崩溃掉电安全保护当需要更
KB执行读擦除写例程
这样做的效率非常低JFFS2是附加文件而不是重写整个扇区并且具有崩溃掉电安全保护这一功能

JFFS2是是为FLASH定制的文件系统JFFS1实现了日志功能JFFS2实现了压缩功能它的整个设计提供了更好的闪存管理JF
缺点很少主要是当文件系统已满或接近满时JFFS2会大大放慢运行速度这是因为垃圾收集的问题

MTD驱动程序是专门为基于闪存的设备所设计的它提供了基于扇区的擦除和读写操作的更好的接口MTD子系统支持众多

MTD子系统提供了对字符设备MTD_CHAR和块设备MTD_BLOCK的支持MTD_CHAR提供对闪存的原始字符访问象通常
/dev/mtd0mtd1mtd2等MTD_BLOCK块设备文件是 /dev/mtdblock0mtdblock1等等

NAND和NOR是制作Flash的工艺CFI和JEDEC是flash硬件提供的接口linux通过这些用通用接口抽象出MTD设备JFFS2文件

NOR flash带有SRAM接口□可以直接存取内部的每一个字节□NAND器件使用串行I/O口来存取数据□8个引脚用来传送控制□地址和数据信息□NAND读和写操作用512字节的块□

MTD内存技术设备层次结构

MTD(memory technology device内存技术设备)在硬件和文件系统层之间的提供了一个抽象的接口□MTD是用来访问内存设备□如□ROM□flash□的中间层□它将内存设备的共有

MTD中间层细分为四层□按从上到下依次为□设备节点□MTD设备层□MTD原始设备层和硬件驱动层□MTD中间层层次结构图如

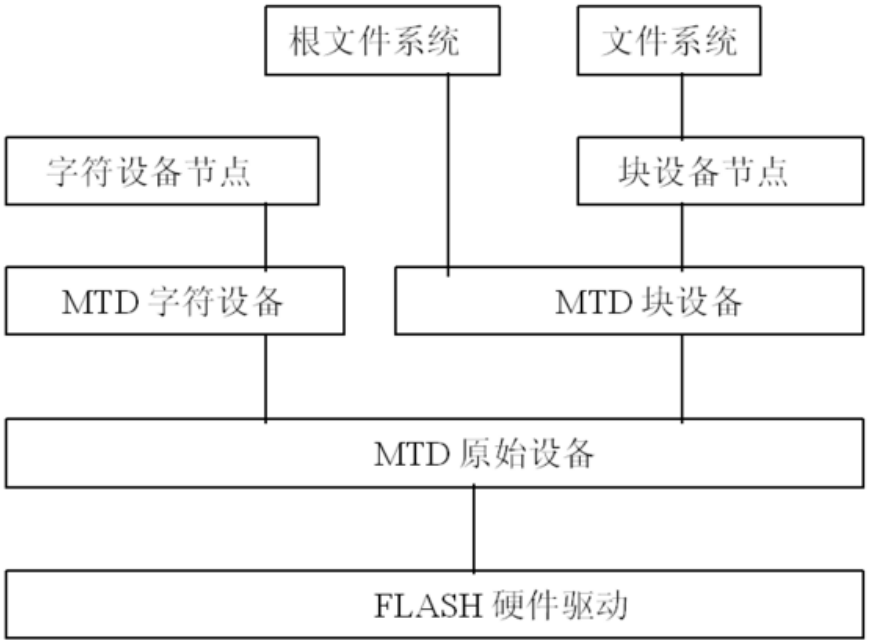

图1 MTD中间层层次结构图

Flash硬件驱动层对应的是不同硬件的驱动程序□它负责驱动具体的硬件□例如□符合CFI接口标准的Flash芯片驱动驱动程序在c

在原始设备层中□各种内存设备抽象化为原始设备□原始设备实际上是一种块设备□MTD字符设备的读写函数也调用原始设备

```
struct mtd_info *mtd_table[MAX_MTD_DEVICES];
```

每个原始设备可能分成多个设备分区□设备分区是将一个内存分成多个块□每个设备分区用一个结构mtd_part来描述□所有的分

```
/* Our partition linked list */
static LIST_HEAD(mtd_partitions);
```

MTD原始设备到具体设备之间存在的一些映射关系数据在drivers/mtd/maps/目录下的对应文件中□这些映射数据包括分区信息

在MTD设备层中□MTD字符设备通过注册的file operation函数集来操作设备□而这些函数是通过原始设备层的操作函数来实现的□即调用了块设备的操作函数□MTD块设备实

```
static struct mtdblk_dev {
    struct mtd_info *mtd;
    int count;
    struct semaphore cache_sem;
    unsigned char *cache_data;
    unsigned long cache_offset;
    unsigned int cache_size;
    enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;
} *mtdblks[MAX_MTD_DEVICES];
```

由于flash设备种类的多样性，MTD用MTD翻译层将三大类flash设备进行的封装，每大类设备有自己的操作函数集，它们的mtdb

图 MTD设备在内核中的层次图

MTD原始设备层中封装了三大类设备，分别是Inverse Flash，NAND Flash和MTD，它们的上体读写方法不一样，这里只分析了MTD，因为它是最常用的。

设备层和原始设备层的函数调用关系

原始设备层主要是通过mtd_info结构来管理设备，函数add_mtd_partitions()和del_mtd_partitions()将的设备分区的mtd_info结构

设备层和原始设备层的函数调用关系图如图2，MTD各种结构之间的关系图如图3。

图2 设备层和原始设备层的函数调用关系

图3 MTD各种结构之间的关系

MTD相关结构

MTD块设备的结构mtdblk_dev代表了一个闪存块设备，MTD字符设备没有相对应的结构，结构mtdblk_dev列出如下：

```
struct mtdblk_dev {
    struct mtd_info mtd; / Locked */        下层原始设备层的MTD设备结构
    int count;
    struct semaphore cache_sem;
    unsigned char *cache_data;       //缓冲区数据地址
    unsigned long cache_offset;//在缓冲区中读写位置偏移
    //缓冲区中的读写数据大小，通常被设置为MTD设备的erasesize。
    unsigned int cache_size;
    enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;//缓冲区状态
}
```

结构mtd_info描述了一个MTD原始设备，每个分区也被实现为一个mtd_info，如果有两个MTD原始设备，每个上有三个分区，在

```
struct mtd_info {
      u_char type;              //内存技术的类型
      u_int32_t flags;       //标志位
      u_int32_t size;           // mtd设备的大小

        //“主要的”erasesize，同一个mtd设备可能有数种不同的erasesize。
        u_int32_t erasesize;
        u_int32_t oobblock;  // oob块大小，例如，512
      u_int32_t oobsize;    //每个块oob数据量，例如16
      u_int32_t ecctype;       //ecc类型
      u_int32_t eccsize;          //自动ecc可以工作的范围

         // Kernel-only stuff starts here.
        char *name;
        int index;

         //可变擦除区域的数据，如果是0，意味着整个设备为erasesize
        int numeraseregions; //不同erasesize的区域的数目，通常是1。
        struct mtd_erase_region_info *eraseregions;
        u_int32_t bank_size;
```

```
        struct module *module;
        //此routine用于将一个erase_info加入erase queue
        int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
        /* This stuff for eXecute-In-Place */
        int (*point) (struct mtd_info *mtd, loff_t from, size_t len,
                size_t *retlen, u_char **mtdbuf);
        /* We probably shouldn't allow XIP if the unpoint isn't a NULL
*/
         void (*unpoint) (struct mtd_info *mtd, u_char * addr);
        int (*read) (struct mtd_info *mtd, loff_t from, size_t len,
                size_t *retlen, u_char *buf);
        int (*write) (struct mtd_info *mtd, loff_t to, size_t len,
                size_t *retlen, const u_char *buf);
        int (*read_ecc) (struct mtd_info *mtd, loff_t from,
                size_t len, size_t *retlen, u_char *buf, u_char
*eccbuf);
        int (*write_ecc) (struct mtd_info *mtd, loff_t to, size_t len,
                size_t *retlen, const u_char *buf, u_char *eccbuf);
        int (*read_oob) (struct mtd_info *mtd, loff_t from, size_t len,
                size_t *retlen, u_char *buf);
        int (*write_oob) (struct mtd_info *mtd, loff_t to, size_t len,
                size_t *retlen, const u_char *buf);
        /* iovec-based read/write methods. We need these especially for
 NAND flash,
        with its limited number of write cycles per erase.
        NB: The 'count'parameter is the number of vectors, each of
        which contains an (ofs, len) tuple.
        */
        int (*readv) (struct mtd_info *mtd, struct iovec *vecs,
                unsigned long count, loff_t from, size_t *retlen);
        int (*writev) (struct mtd_info *mtd, const struct iovec *vecs,
                unsigned long count, loff_t to, size_t *retlen);
        /* Sync */
        void (*sync) (struct mtd_info *mtd);

        /* Chip-supported device locking */
        int (*lock) (struct mtd_info *mtd, loff_t ofs, size_t len);
        int (*unlock) (struct mtd_info *mtd, loff_t ofs, size_t len);
        /* Power Management functions */
        int (*suspend) (struct mtd_info *mtd);
        void (*resume) (struct mtd_info *mtd);

        void *priv;                     //指向map_info结构
}
```

设备层的mtdblcok设备的notifier声明如下：

```
static struct mtd_notifier notifier = {
    mtd_notify_add,
    mtd_notify_remove,
    NULL
};
```

mtd_part结构是用于描述MTD原始设备分区的结构mtd_part中的list成员链成一个链表mtd_partitons每个mtd_part结构中的m
结构mtd_part列出如下

```
/* Our partition linked list */
static LIST_HEAD(mtd_partitions);                    MTD原始设备分区的链表
struct mtd_part {
      struct mtd_info mtd;              //分区的信息大部分由其master决定
      struct mtd_info *master;        //该分区的主分区
      u_int32_t offset;                    //该分区的偏移地址
      int index;                                 //分区号
    struct list_head list;
};
```

结构mtd_partition描述mtd设备分区的结构在MTD原始设备层调用函数add_mtd_partions时传递分区信息使用结构列出如下

```
struct mtd_partition {
      char *name;              //分区名
      u_int32_t size;             //分区大小
      u_int32_t offset;             //在主MTD空间的偏移
        u_int32_t mask_flags;
};
```

MTD块设备初始化

图 函数init_mtdblock调用层次图

在具体的设备驱动程序初始化时它会添加一个MTD设备结构到mtd_table数组中MTD翻译层通过查找这个数组可访问到各
函数init_mtdblock注册一个MTD翻译层设备初始化处理请求的线程赋上MTD翻译层设备操作函数集实例注册这个设备的
mtd块设备驱动程序利用一个线程当有读写请求时从缓冲区将数据写入块设备或从块设备读入到缓冲区中
函数init_mtdblock分析如下在drivers/mtd/mtdblock.c中

```
static int __init init_mtdblock(void)
{
    return register_mtd_blktrans(&mtdblock_tr);
}
```

MTD翻译层设备操作函数集实例列出如下

```
static struct mtd_blktrans_ops mtdblock_tr = {
      .name             = “mtdblock”,
      .major             = 31,
      .part_bits       = 0,
      .open              = mtdblock_open,
      .flush             = mtdblock_flush,
      .release       = mtdblock_release,
      .readsect       = mtdblock_readsect,
```

```
        .writesect      = mtdblock_writesect,
        .add_mtd        = mtdblock_add_mtd,
        .remove_dev     = mtdblock_remove_dev,
        .owner          = THIS_MODULE,
};

static LIST_HEAD(blktrans_majors);
int register_mtd_blktrans(struct mtd_blktrans_ops *tr)
{
        int ret, i;
        //如果第一个设备类型被注册了，注册notifier来阻止
        /* Register the notifier if/when the first device type is
        registered, to prevent the link/init ordering from fucking
        us over. */
        if (!blktrans_notifier.list.next)//如果不存在
                //注册MTD翻译层块设备，创建通用硬盘结构并注册
                register_mtd_user(&blktrans_notifier);
        tr->blkcore_priv = kmalloc(sizeof(*tr->blkcore_priv),
GFP_KERNEL);

        if (!tr->blkcore_priv)
                return -ENOMEM;
        memset(tr->blkcore_priv, 0, sizeof(*tr->blkcore_priv));
        down(&mtd_table_mutex);

        //创建blk_major_name结构初始化后加到&major_names[]数组中
        ret = register_blkdev(tr->major, tr->name);
      …
        spin_lock_init(&tr->blkcore_priv->queue_lock);
        init_completion(&tr->blkcore_priv->thread_dead);
        init_waitqueue_head(&tr->blkcore_priv->thread_wq);

        //创建请求队列并初始化，赋上块设备特定的请求处理函数mtd_blktrans_request
        tr->blkcore_priv->rq = blk_init_queue(mtd_blktrans_request,
                &tr->blkcore_priv->queue_lock);
      …
        tr->blkcore_priv->rq->queuedata = tr;//赋上MTD翻译层块设备操作函数集
        //创建线程mtd_blktrans_thread
        ret = kernel_thread(mtd_blktrans_thread, tr, CLONE_KERNEL);
      …
        //在devfs文件系统中创建设备的目录名
        devfs_mk_dir(tr->name);

        INIT_LIST_HEAD(&tr->devs);//初始化设备的链表
        list_add(&tr->list, &blktrans_majors);
        for (i=0; i<MAX_MTD_DEVICES; i++) {
                if (mtd_table[i] && mtd_table[i]->type != MTD_ABSENT)
```

```
                        //创建MTD翻译层设备结构并初始化，然后到MTD设备链表中
                        tr->add_mtd(tr, mtd_table[i]);
        }

        up(&mtd_table_mutex);
         return 0;
 }
```

函数mtd_blktrans_request是MTD设备的请求处理函数，当请求队列中的请求需要设备处理时调用这个函数，在MTD设备中，函

```
static void mtd_blktrans_request(struct request_queue *rq)
{
    struct mtd_blktrans_ops *tr = rq->queuedata;
    wake_up(&tr->blkcore_priv->thread_wq);
}
```

线程函数mtd_blktrans_thread处理块设备的读写请求，函数mtd_blktrans_thread列出如下：

```
static int mtd_blktrans_thread(void *arg)
{
    struct mtd_blktrans_ops *tr = arg;
    struct request_queue *rq = tr->blkcore_priv->rq;
    /* we might get involved when memory gets low, so use PF_MEMALLOC
*/ current->flags |= PF_MEMALLOC | PF_NOFREEZE;
    //变成以init为父进程的后台进程
    daemonize("%sd", tr->name);

    //因为一些内核线程实际上要与信号打交道，daemonize()没有做后台化工作，
    //我们不能仅调用exit_sighand函数，
    //因为当最终退出时这样将可能引起oop，对象指针溢出错误，
    spin_lock_irq(&current->sighand->siglock);
    sigfillset(&current->blocked);

    // 重新分析是否有挂起信号并设置或清除TIF_SIGPENDING标识给当前进程
    recalc_sigpending();
    spin_unlock_irq(&current->sighand->siglock);
    spin_lock_irq(rq->queue_lock);

    while (!tr->blkcore_priv->exiting) {
        struct request *req;
        struct mtd_blktrans_dev *dev;
        int res = 0;
        DECLARE_WAITQUEUE(wait, current);，//声明当前进程的等待队列

        req = elv_next_request(rq);//从块设备的请求队列中得到下一个请求

        if (!req) {//如果请求不存在
            //将设备的等待线程加到等待队列中
            add_wait_queue(&tr->blkcore_priv->thread_wq, &wait);
```

```
            set_current_state(TASK_INTERRUPTIBLE);
            spin_unlock_irq(rq->queue_lock);
            schedule();//调度让CPU有机会执行等待的线程
             remove_wait_queue(&tr->blkcore_priv->thread_wq, &wait);
            spin_lock_irq(rq->queue_lock);
            continue;
        }

        //如果请求存在
        dev = req->rq_disk->private_data;//得到请求的设备
        tr = dev->tr;//得到MTD翻译层设备操作函数集实例
        spin_unlock_irq(rq->queue_lock);
        down(&dev->sem); res = do_blktrans_request(tr, dev, req);//处理请求
up(&dev->sem);
        spin_lock_irq(rq->queue_lock);
        end_request(req, res); //从请求队列中删除请求并更新统计信息
    }
    spin_unlock_irq(rq->queue_lock);
    //调用所有请求处理完的回调函数并调用do_exit函数退出线程
     complete_and_exit(&tr->blkcore_priv->thread_dead, 0);
}
```

函数do_blktrans_request完成请求的具体操作它调用MTD翻译层设备操作函数集实例中的具体函数来进行处理函数do_blktr

```
static int do_blktrans_request(struct mtd_blktrans_ops *tr,
        struct mtd_blktrans_dev *dev,
        struct request *req)
{
    unsigned long block, nsect;
    char *buf;

    block = req->sector;
    nsect = req->current_nr_sectors;
    buf = req->buffer;

    if (!(req->flags & REQ_CMD))
        return 0;
    //如果读写的扇区数超出了块设备的容量返回
    if (block + nsect > get_capacity(req->rq_disk))
        return 0;

    //根据(rq)->flags & 1标识来判断操作方式调用具体的设备操作函数
    switch(rq_data_dir(req)) {
        case READ:
            for (; nsect > 0; nsect--, block++, buf += 512)
                if (tr->readsect(dev, block, buf))
                    return 0;
            return 1;
```

```
        case WRITE:
            if (!tr->writesect)
                return 0;
            for (; nsect > 0; nsect--, block++, buf += 512)
                if (tr->writesect(dev, block, buf))
                    return 0;
            return 1;

        default:
            printk(KERN_NOTICE “Unknown request %ld\n”,
rq_data_dir(req));
        return 0;
    }
}
```

图 函数register_mtd_user调用层次图

结构mtd_notifier是用于通知加上和去掉MTD原始设备对于块设备来说这个结构实例blktrans_notifier用来通知翻译层加上和

```
static struct mtd_notifier blktrans_notifier = {
    .add = blktrans_notify_add,
    .remove = blktrans_notify_remove,
};
```

函数register_mtd_user注册MTD设备通过分配通盘硬盘结构来激活每个MTD设备使其出现在系统中函数register_mtd_user

函数register_mtd_user分析如下在drivers/mtd/mtdcore.c中

```
static LIST_HEAD(mtd_notifiers);
void register_mtd_user (struct mtd_notifier *new)
{
    int i;
    down(&mtd_table_mutex);
    //将MTD块设备的通知结构实例blktrans_notifier加入
    //到全局链表mtd_notifiers上
    list_add(&new->list, &mtd_notifiers);
    //模块引用计数加1
__module_get(THIS_MODULE);

    //对每个MTD块设备调用MTD通知结构实例的加设备函数
    for (i=0; i< MAX_MTD_DEVICES; i++)
        if (mtd_table[i])
            new->add(mtd_table[i]);
    up(&mtd_table_mutex);
}
```

函数blktrans_notify_add通知MTD翻译层将设备加入到链表blktrans_majors中并分配处理每个MTD分区对应的通用硬盘结构

函数blktrans_notify_add分析如下在drivers/mtd/mtd_blkdevs.c中

```
static LIST_HEAD(blktrans_majors);
static void blktrans_notify_add(struct mtd_info *mtd)
```

```
{
    struct list_head *this;
    if (mtd->type == MTD_ABSENT)//设备不存在
        return;

    //遍历每个MTD主块设备
    list_for_each(this, &blktrans_majors) {
        struct mtd_blktrans_ops *tr = list_entry(this,
        struct mtd_blktrans_ops, list);
        tr->add_mtd(tr, mtd);
    }

}
```

函数mtdblock_add_mtd分配了MTD翻译层块设备结构□初始化后加到MTD翻译层块设备链表中□函数mtdblock_add_mtd分析如

```
static void mtdblock_add_mtd(struct mtd_blktrans_ops *tr,
struct mtd_info *mtd)
{
    struct mtd_blktrans_dev *dev = kmalloc(sizeof(*dev), GFP_KERNEL);
    if (!dev)
        return;
    memset(dev, 0, sizeof(*dev));
    dev->mtd = mtd;
    dev->devnum = mtd->index;
    dev->blksize = 512;
    dev->size = mtd->size >> 9;
    dev->tr = tr;

    if (!(mtd->flags & MTD_WRITEABLE))
        dev->readonly = 1;
    add_mtd_blktrans_dev(dev);
}
```

函数add_mtd_blktrans_dev给每个MTD主设备分配设备号□并加到MTD设备链表对应位置上□然后给每个MTD设备分区分配一

函数add_mtd_blktrans_dev分析如下□在drivers/mtd/mtd_blkdevs.c中□□

```
int add_mtd_blktrans_dev(struct mtd_blktrans_dev *new)
{
    struct mtd_blktrans_ops *tr = new->tr;
    struct list_head *this;
    int last_devnum = -1;
    struct gendisk *gd;

    if (!down_trylock(&mtd_table_mutex)) {
        up(&mtd_table_mutex);
        BUG();
    }
    //遍历MTD每个主块设备
```

```
    list_for_each(this, &tr->devs) {
        struct mtd_blktrans_dev *d = list_entry(this,
         struct mtd_blktrans_dev,list);

        if (new->devnum == -1) {//如果没有设备号
            //使用第一个空闲的设备号
            if (d->devnum != last_devnum+1) {
                //找到空闲设备号，并把设备加到链表的尾部
                new->devnum = last_devnum+1;
                list_add_tail(&new->list, &d->list);
                goto added;
            }
        } else if (d->devnum == new->devnum) {//设备号已被使用
            /* Required number taken */
            return -EBUSY;
        } else if (d->devnum > new->devnum) {
            //申请的设备号是空闲的，加到链表的尾部
            list_add_tail(&new->list, &d->list);
            goto added;
        }
        last_devnum = d->devnum;
    }

    if (new->devnum == -1)//如果新设备的设备号为-1，就赋上(最后一个设备号+1)
        new->devnum = last_devnum+1;
    //所有的设备号*分区数 > 256
    if ((new->devnum << tr->part_bits) > 256) {
        return -EBUSY;
    }

    init_MUTEX(&new->sem);
    list_add_tail(&new->list, &tr->devs);//加到链表尾部

added:
    if (!tr->writesect)
        new->readonly = 1;
    //分配通知硬盘结构gendisk，每分区一个
    gd = alloc_disk(1 << tr->part_bits);
    if (!gd) {
        list_del(&new->list);
        return -ENOMEM;
    }

    //初始化通用硬盘结构
    gd->major = tr->major;
    gd->first_minor = (new->devnum) << tr->part_bits;
    gd->fops = &mtd_blktrans_ops;
```

```
    snprintf(gd->disk_name, sizeof(gd->disk_name),
        "%s%c", tr->name, (tr->part_bits?'a':'0') + new->devnum);
    snprintf(gd->devfs_name, sizeof(gd->devfs_name),
        "%s/%c", tr->name, (tr->part_bits?'a':'0') + new->devnum);

    /* 2.5 has capacity in units of 512 bytes while still
    having BLOCK_SIZE_BITS set to 10. Just to keep us amused. */
    set_capacity(gd, (new->size * new->blksize) >> 9);
    gd->private_data = new; //通用硬盘结构的私有数据指向翻译层的MTD设备
    new->blkcore_priv = gd;
    gd->queue = tr->blkcore_priv->rq; //设置请求队列

    if (new->readonly)
        set_disk_ro(gd, 1);//设置硬盘读写模式
    add_disk(gd);//加通用硬盘结构到全局链表中
    return 0;
}
```

MTD块设备的读写操作

函数mtdblock_writesect调用层次图

MTD翻译层设备操作函数集实例mtdblock_tr有对MTD设备的各种操作函数这些操作函数调用了mtd_info结构中的操作函数

函数mtdblock_writesect将数据写入到flash设备中函数分析如下

```
static int mtdblock_writesect(struct mtd_blktrans_dev *dev,
            unsigned long block, char *buf)
{
    //从MTD块设备数组中得到块设备结构
    struct mtdblk_dev *mtdblk = mtdblks[dev->devnum];
    if (unlikely(!mtdblk->cache_data && mtdblk->cache_size)) {
        //分配块设备用于擦除的缓存空间
        mtdblk->cache_data = vmalloc(mtdblk->mtd->erasesize);
        if (!mtdblk->cache_data)
            return -EINTR;

    }
    //从位置block开始写一个扇区(512字节)
    return do_cached_write(mtdblk, block<<9, 512, buf);
}
```

函数do_cached_write将数据写入到设备由于flash设备需要先擦除再才能写入因而在数据块大小不是正好扇区大小需要通

函数do_cached_write分析如下

```
static int do_cached_write (struct mtdblk_dev *mtdblk, unsigned long
pos,
        int len, const char *buf)
{
    struct mtd_info *mtd = mtdblk->mtd;
    //得到擦除缓冲区大小
```

```
    unsigned int sect_size = mtdblk->cache_size;
    size_t retlen;
    int ret;

    if (!sect_size)//如果块设备的缓冲大小为0，直接写设备
        return MTD_WRITE (mtd, pos, len, &retlen, buf);

    while (len > 0) {
        //将要写的在设备上的位置pos地址处，长度为len
        //             |<-offset-->|<-size-->|
        // ----------sect_start---|pos-----len-|
        //             |<-   sect_size     ->|
        //计算扇区开始位置
        unsigned long sect_start = (pos/sect_size)*sect_size;
        //计算出相对扇区开始位置的偏移
        unsigned int offset = pos - sect_start;
        //计算出所写的大小
        unsigned int size = sect_size - offset;
        if( size > len )
            size = len;

        if (size == sect_size) {//正好是擦除缓冲区大小
            //直接写入，不需要通过缓冲区，
            ret = erase_write (mtd, pos, size, buf);
            if (ret)
                return ret;
        } else {
            //只有部分扇区大小的数据，需通过缓冲区补充成扇区大小，
            //方法是，先从设备中读出数据到缓冲区，再将buf中数据拷贝到缓冲区，
            //这样，凑合成一个扇区大小的数据，再把缓冲区数据写入设备，
            //如果缓冲区数据是脏的，把缓冲区数据写设备
            if (mtdblk->cache_state == STATE_DIRTY &&
                    mtdblk->cache_offset != sect_start) {
                ret = write_cached_data(mtdblk);
                if (ret)
                    return ret;
            }

            if (mtdblk->cache_state == STATE_EMPTY ||
                    mtdblk->cache_offset != sect_start) {
                //把当前的扇区数据填充缓冲区，
                mtdblk->cache_state = STATE_EMPTY;
                ret = MTD_READ(mtd, sect_start, sect_size, &retlen,
                mtdblk->cache_data);
                if (ret)
                    return ret;
                if (retlen != sect_size)
```

```
                    return -EIO;
                mtdblk->cache_offset = sect_start;
                mtdblk->cache_size = sect_size;
                mtdblk->cache_state = STATE_CLEAN;
            }

            //将数据从buf中拷贝到缓冲区中
            memcpy (mtdblk->cache_data + offset, buf, size);
            mtdblk->cache_state = STATE_DIRTY;
        }

        buf += size;
        pos += size;
        len -= size;
    }

    return 0;
}
```

函数write_cached_data将设备缓存中的数据写入到设备□在写完缓存中数据时□缓存的状态发生变化□函数write_cached_data列

```
static int write_cached_data (struct mtdblk_dev *mtdblk)
{
    struct mtd_info *mtd = mtdblk->mtd;
    int ret;

    if (mtdblk->cache_state != STATE_DIRTY)
        return 0;
    ret = erase_write (mtd, mtdblk->cache_offset,
    mtdblk->cache_size, mtdblk->cache_data);

    if (ret)
        return ret;
    mtdblk->cache_state = STATE_EMPTY;
    return 0;
}
```

函数erase_write写一扇区数据到设备中□写的方法是□先擦除对应扇区□擦除完成后□再写数据□函数erase_write分析如下□

```
static int erase_write (struct mtd_info *mtd, unsigned long pos,
        int len, const char *buf)
{
    struct erase_info erase;
    DECLARE_WAITQUEUE(wait, current);
    wait_queue_head_t wait_q;
    size_t retlen;
    int ret;

    //首先□擦除flash闪存块
```

```
    init_waitqueue_head(&wait_q);
    erase.mtd = mtd;
    erase.callback = erase_callback;
    erase.addr = pos;
    erase.len = len;
    erase.priv = (u_long)&wait_q;

    set_current_state(TASK_INTERRUPTIBLE);
    add_wait_queue(&wait_q, &wait);

    ret = MTD_ERASE(mtd, &erase);
    if (ret) {//如果擦除完成
        set_current_state(TASK_RUNNING);//运行当前进程
        remove_wait_queue(&wait_q, &wait);//清除等待队列
        return ret;
    }

    schedule();  //调度来等待擦除工作的完成
    remove_wait_queue(&wait_q, &wait); //清除等待队列
    //第二步写数据到flash设备 ret = MTD_WRITE (mtd, pos, len, &retlen, buf);
    if (ret)
        return ret;
    if (retlen != len)
        return -EIO;
    return 0;
}
```

函数mtdblock_readsect调用了函数do_cached_read从flash设备中读数据到指定位置的buf中如果数据在设备的缓存中就直接

```
static int do_cached_read (struct mtdblk_dev *mtdblk, unsigned long
pos,
               int len, char *buf)
```

其中参数mtdblk是指定的MTD块设备pos是MTD设备中指定的位置len是长度buf是被写入的地址调用成功时返回0失败时

MTD核心初始化

MTD核心主要工作是进行电源管理及在/proc文件系统中输出MTD设备的信息函数init_mtd初始化proc文件系统函数注册电

函数init_mtd分析如下在linux/drivers/mtd/mtd_core.c中

```
int __init init_mtd(void)
{
    if ((proc_mtd = create_proc_entry( “mtd”, 0, 0 )))
        proc_mtd->read_proc = mtd_read_proc;
    mtd_pm_dev = pm_register(PM_UNKNOWN_DEV, 0, mtd_pm_callback);
    return 0;
}

static void __exit cleanup_mtd(void)
{
```

```
    if (mtd_pm_dev) {
        pm_unregister(mtd_pm_dev);
        mtd_pm_dev = NULL;
    }

    if (proc_mtd)
        remove_proc_entry( “mtd”, 0);
}
```

mtd_read_proc函数是proc系统调用到的最终读函数，它以字符形式读出结构struct mtd_info相关信息。

mtd_pm_callback函数通过各个设备的MTD设备结构mtd_info将电源管理请求传给具体的设备驱动程序，mtd_pm_callback函数

```
static int mtd_pm_callback(struct pm_dev *dev, pm_request_t rqst, void
*data)
{
    int ret = 0, i;
    if (down_trylock(&mtd_table_mutex))
        return -EAGAIN;

    if (rqst == PM_SUSPEND) {//电源挂起状态
        for (i = 0; ret == 0 && i < MAX_MTD_DEVICES; i++) {
            if (mtd_table[i] && mtd_table[i]->suspend)
                ret = mtd_table[i]->suspend(mtd_table[i]);
            }
            } else i = MAX_MTD_DEVICES-1;

            if (rqst == PM_RESUME || ret) {//电源恢复
            for ( ; i >= 0; i--) {
                if (mtd_table[i] && mtd_table[i]->resume)
                    mtd_table[i]->resume(mtd_table[i]);
            }
        }
    up(&mtd_table_mutex);
    return ret;
}
```

MTD字符设备

当系统打开flash设备上的文件，它建立好了文件的操作函数集实例，当对文件操作时，就调用了这个文件操作函数集实例中的函

函数init_mtdchar注册了一个字符设备，列出如下，在drivers/mtd/mtdchar.c中：

```
static int __init init_mtdchar(void)
{
    if (register_chrdev(MTD_CHAR_MAJOR, “mtd”, &mtd_fops)) {
        printk(KERN_NOTICE “Can’t allocate major number %d
                for Memory Technology Devices.\n”,
                MTD_CHAR_MAJOR);
            return -EAGAIN;
    }
```

```
    mtdchar_devfs_init();
    return 0;
}
```

MTD字符设备的操作函数结构mtd_fops列出如下：

```
static struct file_operations mtd_fops = {
      .owner            = THIS_MODULE,
      .llseek            = mtd_lseek,
      .read            = mtd_read,
      .write            = mtd_write,
      .ioctl            = mtd_ioctl,
      .open            = mtd_open,
      .release      = mtd_close,
};
```

这里只分析了mtd_write函数，函数mtd_write完成此函数是对MTD字符设备的写操作，其中参数file是系统给MTD字符设备驱动函数mtd_write分析如下：

```
static ssize_t mtd_write(struct file *file, const char __user *buf,
        size_t count,loff_t *ppos)
{
    struct mtd_info *mtd = file->private_data;//得到MTD设备结构
    char *kbuf;
    size_t retlen;
    size_t total_retlen=0;
    int ret=0;
    int len;

    DEBUG(MTD_DEBUG_LEVEL0,"MTD_write\n");
    if (*ppos == mtd->size)
        return -ENOSPC;
    if (*ppos + count > mtd->size)
        count = mtd->size - *ppos;

    if (!count)
        return 0;
    while (count) {
        if (count > MAX_KMALLOC_SIZE)
            len = MAX_KMALLOC_SIZE;
        else
            len = count;

        kbuf=kmalloc(len,GFP_KERNEL);//分配buffer
        if (!kbuf) {
            printk("kmalloc is null\n");
            return -ENOMEM;
        }
```

```
        //从用户空间buf拷贝数据到内核空间kbuf
        if (copy_from_user(kbuf, buf, len)) {
            kfree(kbuf);
            return -EFAULT;
        }

        //调用设备的写函数
        ret = (*(mtd->write))(mtd, *ppos, len, &retlen, kbuf);
        if (!ret) {
            *ppos += retlen;
            total_retlen += retlen;
            count -= retlen;
            buf += retlen;
        }
        else {
            kfree(kbuf);
            return ret;
        }

        kfree(kbuf);
    }

    return total_retlen;
} /* mtd_write */
```

具体**flash**芯片的探测及映射

（1）flash芯片映射信息结构

flash芯片映射信息结构map_info描述了每一排闪存的映射信息，如：每一排闪存的驱动程序，物理地址，读写操作函数和映射地

flash芯片映射信息结构map_info列出如下（在include/linux/mtd/mtd.h中）：

```
struct map_info {
    char *name;
    unsigned long size;  //flash大小
    unsigned long phys;    //起始物理地址

#define NO_XIP (-1UL)
    void __iomem *virt;  //I/O映射的虚拟地址
     void *cached; //8位的字节，它不是实际总线的必要宽度，
    //在再次与第一个芯片通信之前，它是字节上重复的间隔
    int bankwidth;
#ifdef CONFIG_MTD_COMPLEX_MAPPINGS

    map_word (*read)(struct map_info *, unsigned long);
    void (*copy_from)(struct map_info *, void *, unsigned long,
ssize_t);
    void (*write)(struct map_info *, const map_word, unsigned long);
```

```
     void (*copy_to)(struct map_info *, unsigned long, const void *,
ssize_t);
    /* We can perhaps put in 'point' and 'unpoint' methods, if we
really
        want to enable XIP for non-linear mappings. Not yet though. */
#endif


/*在映射驱动程序的copy_from应用中□映射驱动程序使用缓存是可能的□然而□当芯片驱动程序知道一些flash区域已改变
    void (*inval_cache)(struct map_info *, unsigned long, ssize_t);
    /* set_vpp() must handle being reentered—enable, enable, disable
        must leave it enabled. */
    void (*set_vpp)(struct map_info *, int);
    unsigned long map_priv_1;
    unsigned long map_priv_2;
    void *fldrv_priv;
    struct mtd_chip_driver *fldrv; //flash芯片驱动程序
};
```

结构mtd_chip_driver是flash芯片驱动程序的描述□列出如下□

```
struct mtd_chip_driver {
    struct mtd_info *(*probe)(struct map_info *map);//探测函数
    void (*destroy)(struct mtd_info *);
    struct module *module;□//驱动程序的模块结构
    char *name;□//驱动程序名
    struct list_head list;
};
```

□2□flash芯片探测方法及接口标准

每种flash控制芯片可控制多种闪存□这个控制芯片的驱动程序有自己的读写和探测操作函数或者使用通用的操作函数□它注册

在/drivers/mtd/chips目录下有各种flash控制芯片的驱动程序及芯片探测程序□这些文件有chipreg.c□gen_probe.c□cfi_probe.c□jede

确定flash闪存芯片是否支持CFI接口的方法是□向flash闪存的地址0x55H写入数据0x98H□再从flash闪存的地址0x10H处开始□读

"cfi_probe"□

也可以用JEDEC□电子电器设备联合会□标准设备模仿CFI接口□探测JEDEC设备的程序在jedec_probe.c中□JEDEC设备的类型为

对于flash芯片□不同的制造商使用不同的命令集□目前Linux的MTD实现的命令集有AMD/Fujitsu的标准命令集和Intel/Sharp的扩

此外还有一些非CFI标准的Flash□其中"jedec"类型的Flash的探测程序在jedec.c中,"sharp"类型的Flash的探测程序在sharp.c中□"a

最后□还有一些非Flash的MTD□比如ROM或absent□无□设备□这些设备的探测程序在map_rom.c□map_ram.c和map_absent.c中□

chip_drvs_list是所有芯片类型的驱动器链表,flash控制芯片的驱动程序通过调用register_mtd_chip_driver()和unregister_mtd_chi

```
void register_mtd_chip_driver(struct mtd_chip_driver *drv)
{
    spin_lock(&chip_drvs_lock);
    list_add(&drv->list, &chip_drvs_list);
    spin_unlock(&chip_drvs_lock);
}
void unregister_mtd_chip_driver(struct mtd_chip_driver *drv)
{
```

```
    spin_lock(&chip_drvs_lock);
    list_del(&drv->list);
    spin_unlock(&chip_drvs_lock);
}
```

映射驱动程序调用函数do_map_probe来查找对应的控制芯片驱动程序函数中参数name是控制芯片类型名称参数是映射驱动

函数do_map_probe分析如下在drivers/mtd/chips/chipreg.c中

```
struct mtd_info *do_map_probe(const char *name, struct map_info *map)
{
    struct mtd_chip_driver *drv;
    struct mtd_info *ret;
    //查找得到name类型的控制芯片驱动程序结构
    drv = get_mtd_chip_driver(name);
    if (!drv && !request_module("%s", name))
        drv = get_mtd_chip_driver(name);
    if (!drv)
        return NULL;

    ret = drv->probe(map); //具体控制芯片驱动程序的探测函数
    //使用计数减1它可能已是一个探测过的模块在这不需要再探测
    //而在实际的驱动程序中已做处理
    module_put(drv->module);
    if (ret)
        return ret;
    return NULL;
}
```

驱动程序实例分析

1CFI控制芯片驱动程序

CFI控制芯片驱动程序cfi_probe在drivers/mtd/chip/cfi_probe.c中这里只做了简单的说明

```
static struct mtd_chip_driver cfi_chipdrv = {
    .probe              = cfi_probe,
    .name               = "cfi_probe",
    .module             = THIS_MODULE
};
```

函数cfi_probe_init注册驱动程序cfi_chipdrv到全局链表中函数列出如下

```
int __init cfi_probe_init(void)
{
    register_mtd_chip_driver(&cfi_chipdrv);
    return 0;
};
static void __exit cfi_probe_exit(void)
{
    unregister_mtd_chip_driver(&cfi_chipdrv);
};
```

函数cfi_probe调用mtd_do_chip_probe函数来完成了探测操作□在函数cfi_chip_probe中□它调用qry_present来查询是否是CFI接口
函数cfi_probe列出如下□

```
struct mtd_info *cfi_probe(struct map_info *map)
{
    return mtd_do_chip_probe(map, &cfi_chip_probe);
}

static struct chip_probe cfi_chip_probe = {
      .name            = “CFI”,
      .probe_chip      = cfi_probe_chip
};
```

□2□映射驱动程序
用户可设置flash空间映射信息填充在映射驱动程序中□包括该MTD原始设备的起始物理地址□大小□分区情况等□映射驱动程序
flagadm_map是映射信息结构□它含有flash存储空间的配置信息□列出如下□

```
struct map_info flagadm_map = {
            .name =            “FlagaDM flash device”,
            .size =            FLASH_SIZE,
            .bankwidth =      2,
};
```

flagadm_parts是flash存储空间的分区□列出如下□

```
struct mtd_partition flagadm_parts[] = {
   {
        .name =            “Bootloader”,
        .offset      =      FLASH_PARTITION0_ADDR,
        .size =            FLASH_PARTITION0_SIZE
    },
    {
        .name =            “Kernel image”,
        .offset =      FLASH_PARTITION1_ADDR,
        .size =            FLASH_PARTITION1_SIZE
    },
    {
        .name =            “Initial ramdisk image”,
        .offset =      FLASH_PARTITION2_ADDR,
        .size =            FLASH_PARTITION2_SIZE
    },
    {
        .name =            “Persistant storage”,
        .offset =      FLASH_PARTITION3_ADDR,
        .size =            FLASH_PARTITION3_SIZE
    }
};

#define PARTITION_COUNT (sizeof(flagadm_parts)/sizeof(struct
```

```
mtd_partition))
        static struct mtd_info *mymtd;
```

函数init_flagadm是映射驱动程序的初始化，它得到了端口映射地址，初始化了操作函数，通过探测函数得到MTD设备结构，函数

```
int __init init_flagadm(void)
{
    printk(KERN_NOTICE "FlagaDM flash device: %x at %x\n",
        FLASH_SIZE, FLASH_PHYS_ADDR);
    flagadm_map.phys = FLASH_PHYS_ADDR;

    //端口映射
    flagadm_map.virt = ioremap(FLASH_PHYS_ADDR,FLASH_SIZE);
    if (!flagadm_map.virt) {
        printk("Failed to ioremap\n");
        return -EIO;
    }

    //赋上通用的读写操作函数，如__raw_writeb等
    simple_map_init(&flagadm_map);
    //探测CFI类型接口得到MTD设备结构
    mymtd = do_map_probe("cfi_probe", &flagadm_map);
    if (mymtd) {
        mymtd->owner = THIS_MODULE;
        //将分区信息加到MTD设备结构实例mymtd中
        add_mtd_partitions(mymtd, flagadm_parts, PARTITION_COUNT);
        printk(KERN_NOTICE "FlagaDM flash device initialized\n");
        return 0;
    }

    iounmap((void *)flagadm_map.virt);//取消端口映射
    return -ENXIO;
}

static void __exit cleanup_flagadm(void)
{
    if (mymtd) {
        del_mtd_partitions(mymtd);
        map_destroy(mymtd);
    }

    if (flagadm_map.virt) {
        iounmap((void *)flagadm_map.virt);
        flagadm_map.virt = 0;
    }
}
```

SD/MMC卡块设备驱动程序

SD/MMC卡组成的存储系统是许多嵌入设备的主要存储设备□相当于PC机的硬盘□在嵌入设备上的SD/MMC卡控制器通过MM
MMC驱动程序以分通用设备层□MMC抽象设备层□MMC协议层和具体设备层四层来构建□上一层抽象出下一层的共有特性□每
MMC卡的请求管理□电源管理等□MMC协议层将MMC操作分解成标准的MMC协议□具体设备层则负责具体物理设备的寄存器
MMC驱动程序主要处理两部分的内容□一是创建通用硬盘结构向系统注册□以便系统对MMC设备的管理□另一方面□要完成系

图 MMC驱动程序的层次结构

MMC抽象设备层相关结构

□1□设备描述结构

图 MMC卡设备相关结构关系图

MMC设备由控制器及插卡组成□对应的设备结构为mmc_host结构和mmc_card结构□MMC卡设备相关结构关系图如上图□下面
每个卡的插槽对应一个块的数据结构mmc_blk_data□结构列出如下□在drivers/mmc/mmc_block.c中□□

```
struct mmc_blk_data {
      spinlock_t      lock;
      struct gendisk      *disk;□//通用硬盘结构
      struct mmc_queue queue;□//MMC请求队列结构

      unsigned int      usage;
      unsigned int      block_bits;□//卡每一块大小所占的bit位
};
```

结构mmc_card是一个插卡的特性描述结构□它代有了一个插卡□列出如下□在include/linux/mmc/card.h中□□

```
struct mmc_card {
      struct list_head      node;             //在主设备链表中的节点
      struct mmc_host            *host;             // 卡所属的控制器
      struct device            dev;             //通用设备结构
      unsigned int            rca;             //设备的相对本地系统的地址
      unsigned int            state;             //卡的状态
#define MMC_STATE_PRESENT      (1<<0)             //卡出现在sysfs文件系统中
#define MMC_STATE_DEAD            (1<<1)             //卡不在工作状态
#define MMC_STATE_BAD            (1<<2)             //不认识的设备
      u32                  raw_cid[4];       /* raw card CID */
      u32                  raw_csd[4];       /* raw card CSD */
      struct mmc_cid            cid;             //卡的身份鉴别□值来自卡的CID寄存器
struct mmc_csd            csd;             //卡特定信息□值来自卡的CSD寄存器
};
```

结构mmc_host描述了一个MMC卡控制器的特性及操作等□结构mmc_host列出如下□在include/linux/mmc/host.h中□□

```
struct mmc_host {
      struct device            *dev;□//通用设备结构
      struct mmc_host_ops      *ops;□//控制器操作函数集结构
      unsigned int            f_min;
      unsigned int            f_max;
      u32                  ocr_avail;□□□//卡可用的OCR寄存器值
      char                  host_name[8];□//控制器名字
```

```
//主控制器中与块层请求队列相关数据
      unsigned int            max_seg_size;       //最大片断的尺寸
      unsigned short            max_hw_segs;       //最大硬件片断数
      unsigned short            max_phys_segs;       //最大物理片断数
      unsigned short            max_sectors;       //最大扇区数
      unsigned short            unused;

      //私有数据
      struct mmc_ios            ios;             //当前i/o总线设置
      u32                  ocr;             //当前的OCR设置

      struct list_head      cards;             //接在这个主控制器上的设备

      wait_queue_head_t       wq;//等待队列
      spinlock_t            lock;                //卡忙时的锁
      struct mmc_card            *card_busy;       //正与主控制器通信的卡
      struct mmc_card            *card_selected;       //选择的MMC卡

      struct work_struct      detect;//工作结构
};
```

结构mmc_host_ops是控制器的操作函数集它包括请求处理函数指针和控制器对卡I/O的状态的设置函数指针结构mmc_host

```
struct mmc_host_ops {
      void      (*request)(struct mmc_host *host, struct mmc_request
*req);
      void      (*set_ios)(struct mmc_host *host, struct mmc_ios *ios);
}
```

结构mmc_ios描述了控制器对卡的I/O状态列出如下

```
struct mmc_ios {
      unsigned int      clock;                  //时钟频率
      unsigned short      vdd;
      unsigned char       bus_mode;             //命令输出模式
      unsigned char       power_mode;             //电源供应模式
};
```

结构mmc_driver是MMC设备驱动程序结构列出如下

```
struct mmc_driver {
      struct device_driver drv;
      int  (*probe)(struct mmc_card *);
      void (*remove)(struct mmc_card *);
      int  (*suspend)(struct mmc_card *, u32);
      int  (*resume)(struct mmc_card *);
};
```

□2□ 读写请求相关结构

图 MMC卡读写请求结构示意图

对于MMC卡的操作是通过MMC请求结构mmc_request的传递来完成的□来自系统块层的读写请求到达MMC设备抽象层时□用结构mmc_request描述了读写MMC卡的请求□它包括命令□数据及请求完成后的回调函数□结构mmc_request列出如下□在include

```
struct mmc_request {
      struct mmc_command      *cmd;
      struct mmc_data            *data;
      struct mmc_command      *stop;

      void                    *done_data;       //回调函数的参数
      void                    (*done)(struct mmc_request *);//请求完成的回调函数
};
```

结构mmc_queue是MMC的请求队列结构□它封装了通用请求队列结构□加入了MMC卡相关结构□结构mmc_queue列出如下□在c

```
struct mmc_queue {
      struct mmc_card            *card;□□//MMC卡结构
      struct completion       thread_complete;□//线程完成结构
      wait_queue_head_t       thread_wq;□//等待队列
      struct semaphore       thread_sem;
      unsigned int             flags;
      struct request             *req;□□//通用请求结构
      int                    (*prep_fn)(struct mmc_queue *, struct
request *);
□□    //发出读写请求函数
      int                    (*issue_fn)(struct mmc_queue *, struct
request *);□
      void                    *data;
      struct request_queue       *queue;□//块层通用请求队列
      struct scatterlist       *sg;□//碎片链表
};
```

结构mmc_data描述了MMC卡读写的数据相关信息□如□请求□操作命令□数据及状态等□结构mmc_data列出如下□在include/linux/

```
struct mmc_data {
      unsigned int             timeout_ns;       //数据超时( ns,最大80ms)
      unsigned int             timeout_clks;       //数据超时(以时钟计数)
      unsigned int             blksz_bits;       //数据块大小的bit位
      unsigned int             blocks;             //块数
      unsigned int             error;              //数据错误
      unsigned int             flags;□□□//数据操作标识

#define MMC_DATA_WRITE       (1 << 8)
#define MMC_DATA_READ       (1 << 9)
#define MMC_DATA_STREAM       (1 << 10)

      unsigned int             bytes_xfered;
```

```
      struct mmc_command      *stop;              //停止命令
      struct mmc_request      *mrq;              //相关的请求

      unsigned int            sg_len;             //碎片链表的长度
      struct scatterlist      *sg;              // I/O碎片链表指针
};
```

结构mmc_command描述了MMC卡操作相关命令及数据状态信息等结构列出如下

```
struct mmc_command {
      u32                  opcode;
      u32                  arg;
      u32                  resp[4];
      unsigned int            flags;             //期望的反应类型
#define MMC_RSP_NONE      (0 << 0)
#define MMC_RSP_SHORT      (1 << 0)
#define MMC_RSP_LONG      (2 << 0)
#define MMC_RSP_MASK      (3 << 0)
#define MMC_RSP_CRC       (1 << 3)             /* expect valid crc */
#define MMC_RSP_BUSY      (1 << 4)              /* card may send busy */

/*
 * These are the response types, and correspond to valid bit
 * patterns of the above flags.  One additional valid pattern
 * is all zeros, which means we don't expect a response.
 */
#define MMC_RSP_R1       (MMC_RSP_SHORT|MMC_RSP_CRC)
#define MMC_RSP_R1B       (MMC_RSP_SHORT|MMC_RSP_CRC|MMC_RSP_BUSY)
#define MMC_RSP_R2       (MMC_RSP_LONG|MMC_RSP_CRC)
#define MMC_RSP_R3       (MMC_RSP_SHORT)

      unsigned int            retries;       /* max number of retries */
      unsigned int            error;             /* command error */

#define MMC_ERR_NONE      0
#define MMC_ERR_TIMEOUT      1
#define MMC_ERR_BADCRC      2
#define MMC_ERR_FIFO      3
#define MMC_ERR_FAILED      4
#define MMC_ERR_INVALID      5

      struct mmc_data            *data;             //与命令相关的数据片断
      struct mmc_request      *mrq;             //与命令相关的请求
};
```

MMC抽象设备层MMC块设备驱动程序

1、MMC块设备驱动程序初始化

函数mmc_blk_init注册一个MMC块设备驱动程序，它先将MMC块设备名注册到名称数组major_names中，然后，还把驱动程序注

函数mmc_blk_init分析如下，在drivers/mmc/mmc_block.c中：

```
static int __init mmc_blk_init(void)
{
       int res = -ENOMEM;

         //将卡名字mmc和major注册到块设备的名称数组major_names中
       res = register_blkdev(major, "mmc");
       if (res < 0) {
             printk(KERN_WARNING "Unable to get major %d for MMC media:
%d\n",
                    major, res);
             goto out;
       }
       if (major == 0)
             major = res;
       //在devfs文件系统中创建mmc目录
       devfs_mk_dir("mmc");
       return mmc_register_driver(&mmc_driver);

 out:
       return res;
}
```

mmc_driver驱动程序实例声明如下：

```
static struct mmc_driver mmc_driver = {
       .drv            = {
             .name      = "mmcblk",
       },
       .probe            = mmc_blk_probe, // MMC块设备驱动程序探测函数
       .remove            = mmc_blk_remove,
       .suspend      = mmc_blk_suspend,
       .resume            = mmc_blk_resume,
};
```

函数mmc_register_driver 注册一个媒介层驱动程序，其中参数drv是MMC媒介层驱动程序结构。

函数mmc_register_driver分析如下，在drivers/mmc/mmc_sysfs.c中：

```
int mmc_register_driver(struct mmc_driver *drv)
{
       drv->drv.bus = &mmc_bus_type;
       drv->drv.probe = mmc_drv_probe;
       drv->drv.remove = mmc_drv_remove;
     //把块设备注册到sysfs文件系统中对应的总线及设备目录下
       return driver_register(&drv->drv);
```

```
}
```

2、MMC块设备驱动程序探测函数

图 函数mmc_blk_probe调用层次图

函数mmc_blk_probe是MMC控制器探测函数，它探测MMC控制器是否存在，并初始化控制器的结构，同时，还探测MMC卡的状

函数mmc_blk_probe列出如下：

```
static int mmc_blk_probe(struct mmc_card *card)
{
	struct mmc_blk_data *md; //每个插槽一个结构mmc_blk_data
	int err;

	if (card->csd.cmdclass & ~0x1ff)
		return -ENODEV;

	if (card->csd.read_blkbits < 9) {//所读的块小于1扇区
		printk(KERN_WARNING "%s: read blocksize too small (%u)\n",
			mmc_card_id(card), 1 << card->csd.read_blkbits);
		return -ENODEV;
	}
	//分配每个插槽的卡的块数据结构，初始化了通用硬盘及请求队列
	md = mmc_blk_alloc(card);
	if (IS_ERR(md))
		return PTR_ERR(md);
	//设置块大小，发命令设置卡为选中状态
	err = mmc_blk_set_blksize(md, card);
	if (err)
		goto out;

	printk(KERN_INFO "%s: %s %s %dKiB\n",
		md->disk->disk_name, mmc_card_id(card), mmc_card_name(card),
		(card->csd.capacity << card->csd.read_blkbits) / 1024);

	mmc_set_drvdata(card, md);//即card ->driver_data = md
	add_disk(md->disk);	//向系统注册通用硬盘结构，它包括每分区信息
	return 0;

 out:
	mmc_blk_put(md);

	return err;
}
```

函数*mmc_blk_alloc给每一插槽分配一个结构mmc_blk_data，并分配设置通用硬盘结构和初始了请求队列结构，

函数*mmc_blk_alloc分析如下，在drivers/mmc/mmc_block.c中：

```
//最大支持8个控制器，每个控制器可控制4个卡，每个卡最大8个分区，
#define MMC_SHIFT	3 //表示每个卡最大支持8个分区
```

```
#define MMC_NUM_MINORS      (256 >> MMC_SHIFT)□//为256/8=32
//即定义dev_use[32/(8*4)] = devuse[1]□一个控制器用32位表示使用情况
static unsigned long dev_use[MMC_NUM_MINORS/(8*sizeof(unsigned long))];
static struct mmc_blk_data *mmc_blk_alloc(struct mmc_card *card)
{
      struct mmc_blk_data *md;
      int devidx, ret;
//查找dev_use中第一个bit为0的位序号□在MMC_NUM_MINORS位以内□
//找到第个空闲的分区
      devidx = find_first_zero_bit(dev_use, MMC_NUM_MINORS);
      if (devidx >= MMC_NUM_MINORS)
            return ERR_PTR(-ENOSPC);
      __set_bit(devidx, dev_use);//将分区对应的位设置为1□表示使用□

      md = kmalloc(sizeof(struct mmc_blk_data), GFP_KERNEL);//分配对象空间
      if (md) {
            memset(md, 0, sizeof(struct mmc_blk_data));
□□□//分配gendisk结构及通用硬盘的分区hd_struct结构□并初始化内核对象
            md->disk = alloc_disk(1 << MMC_SHIFT);
            if (md->disk == NULL) {
                  kfree(md);
                  md = ERR_PTR(-ENOMEM);
                  goto out;
            }

            spin_lock_init(&md->lock);
            md->usage = 1;
□□□□//初始化请求队列
            ret = mmc_init_queue(&md->queue, card, &md->lock);
            if (ret) {
                  put_disk(md->disk);
                  kfree(md);
                  md = ERR_PTR(ret);
                  goto out;
            }
         //赋上各种请求队列处理函数
            md->queue.prep_fn = mmc_blk_prep_rq;//准备请求
            md->queue.issue_fn = mmc_blk_issue_rq;//发出请求让设备开始处理
            md->queue.data = md;
        //初始化通用硬盘
            md->disk->major      = major;
            md->disk->first_minor = devidx << MMC_SHIFT;
            md->disk->fops = &mmc_bdops;□//块设备操作函数集
            md->disk->private_data = md;
            md->disk->queue = md->queue.queue;
            md->disk->driverfs_dev = &card->dev;
```

```
/*带有永久的块设备可移去的介质应被设置GENHD_FL_REMOVABLE标识□对于永久的介质可移去的块设备不应设置GENHD

            sprintf(md->disk->disk_name, "mmcblk%d", devidx);
            sprintf(md->disk->devfs_name, "mmc/blk%d", devidx);

            md->block_bits = card->csd.read_blkbits;
□□□□//为请求队列设置硬件扇区大小
            blk_queue_hardsect_size(md->queue.queue, 1 << md->block_bits);
            set_capacity(md->disk, card->csd.capacity);//设置卡的容量
      }
 out:
      return md;
}
```

□3□MMC卡请求的处理

图 函数mmc_init_queue调用层次图

函数mmc_init_queue初始化一个MMC卡请求队列结构□其中参数mq是mmc请求队列□参数card是加在这个队列里的mmc卡□参

函数mmc_init_queue分析如下□在drivers/mmc/mmc_queue.c中□□

```
int mmc_init_queue(struct mmc_queue *mq, struct mmc_card *card,
□□□□□□□□□□□□□□□□□□□□spinlock_t *lock)
{
      struct mmc_host *host = card->host;
      u64 limit = BLK_BOUNCE_HIGH;
      int ret;

      if (host->dev->dma_mask && *host->dev->dma_mask)
            limit = *host->dev->dma_mask;

      mq->card = card;
□□//初始化块层的请求队列□请求合并策略□赋上请求处理函数mmc_request□
      mq->queue = blk_init_queue(mmc_request, lock);□
      if (!mq->queue)
            return -ENOMEM;
□□//初始化请求队列的扇区及片断限制
      blk_queue_prep_rq(mq->queue, mmc_prep_request);//赋上准备请求函数
      blk_queue_bounce_limit(mq->queue, limit);
      blk_queue_max_sectors(mq->queue, host->max_sectors);
      blk_queue_max_phys_segments(mq->queue, host->max_phys_segs);
      blk_queue_max_hw_segments(mq->queue, host->max_hw_segs);
      blk_queue_max_segment_size(mq->queue, host->max_seg_size);

      mq->queue->queuedata = mq;
      mq->req = NULL;

      mq->sg = kmalloc(sizeof(struct scatterlist) * host->max_phys_segs,
```

```
                    GFP_KERNEL);
      if (!mq->sg) {
            ret = -ENOMEM;
            goto cleanup;
      }

      init_completion(&mq->thread_complete);
      init_waitqueue_head(&mq->thread_wq);
      init_MUTEX(&mq->thread_sem);

   //创建请求队列处理线程mmc_queue_thread
      ret = kernel_thread(mmc_queue_thread, mq, CLONE_KERNEL);
      if (ret >= 0) {
            wait_for_completion(&mq->thread_complete);
            init_completion(&mq->thread_complete);
            ret = 0;
            goto out;
      }

 cleanup:
      kfree(mq->sg);
      mq->sg = NULL;

      blk_cleanup_queue(mq->queue);
 out:
      return ret;
}
```

函数mmc_prep_request　在准备一个MMC请求时做一些状态转移及保护操作函数列出如下在drivers/mmc/mmc_queue.c中

```
static int mmc_prep_request(struct request_queue *q, struct request
*req)
{
      struct mmc_queue *mq = q->queuedata;
      int ret = BLKPREP_KILL;

      if (req->flags & REQ_SPECIAL) {
            //在req->special 中已建立命令块表示请求已准备好
            BUG_ON(!req->special);
            ret = BLKPREP_OK;
      } else if (req->flags & (REQ_CMD | REQ_BLOCK_PC)) {
            //块I/O请求需要按照协议进行翻译
            ret = mq->prep_fn(mq, req);
      } else {
            //无效的请求
            blk_dump_rq_flags(req, "MMC bad request");
      }
```

```
	if (ret == BLKPREP_OK)//请求已准备好□不需再准备了
		req->flags |= REQ_DONTPREP;

	return ret;
}
```

函数mmc_blk_prep_rq是准备请求时调用的函数□这里仅做了简单的保护处理□列出如下□在drivers/mmc/mmc_block.c中□□

```
static int mmc_blk_prep_rq(struct mmc_queue *mq, struct request *req)
{
	struct mmc_blk_data *md = mq->data;
	int stat = BLKPREP_OK;
□□//如果没有设备□没法完成初始化
	if (!md || !mq->card) {
		printk(KERN_ERR "%s: killing request - no device/host\n",
			req->rq_disk->disk_name);
		stat = BLKPREP_KILL;
	}

	return stat;
}
```

函数mmc_request是通用MMC请求处理函数□它唤醒请求队列处理线程□它在特定的主控制器上被任何请求队列调用□当主控制

函数mmc_request分析如下□在driver/mmd/mmc_queue.c中□□

```
static void mmc_request(request_queue_t *q)
{
	struct mmc_queue *mq = q->queuedata;

	if (!mq->req)//如果有请求□唤醒请求队列处理线程
		wake_up(&mq->thread_wq);
}
```

线程函数mmc_queue_thread调用了具体设备的请求处理函数□利用线程机制来处理请求□函数mmc_queue_thread分析如下□

```
static int mmc_queue_thread(void *d)
{
	struct mmc_queue *mq = d;
	struct request_queue *q = mq->queue;
	DECLARE_WAITQUEUE(wait, current);□//声明一个当前进程的等待队列

	//设置当前进程状态□来让线程自己来处理挂起
	current->flags |= PF_MEMALLOC|PF_NOFREEZE;
□□//让线程继承init进程□从而不会使用用户进程资源
	daemonize("mmcqd");□

	complete(&mq->thread_complete);□//设置线程完成时的回调函数

	down(&mq->thread_sem);
```

```
	add_wait_queue(&mq->thread_wq, &wait);□//加线程到等待队列
	do {
		struct request *req = NULL;

		spin_lock_irq(q->queue_lock);
		set_current_state(TASK_INTERRUPTIBLE);
		if (!blk_queue_plugged(q))□//如果队列是非堵塞状态□得到下一个请求
			mq->req = req = elv_next_request(q);
		spin_unlock_irq(q->queue_lock);

		if (!req) {//如果请求为空
			if (mq->flags & MMC_QUEUE_EXIT)
				break;
			up(&mq->thread_sem);
			schedule();
			down(&mq->thread_sem);
			continue;
		}
		set_current_state(TASK_RUNNING);
□□□□//这里调用了mmc_blk_issue_rq开始处理请求
		mq->issue_fn(mq, req);
	} while (1);
	remove_wait_queue(&mq->thread_wq, &wait);
	up(&mq->thread_sem);
□□//调用请求处理完后的回调函数
	complete_and_exit(&mq->thread_complete, 0);
	return 0;
}
```

函数mmc_blk_issue_rq初始化MMC块请求结构后□向卡发出请求命令□并等待请求的完成□函数分析如下□

```
static int mmc_blk_issue_rq(struct mmc_queue *mq, struct request *req)
{
	struct mmc_blk_data *md = mq->data;
	struct mmc_card *card = md->queue.card;
	int ret;
□□//认领控制器□发命令到卡设置card为选中状态
	if (mmc_card_claim_host(card))
		goto cmd_err;

	do {
		struct mmc_blk_request brq;
		struct mmc_command cmd;
□□□□
□□□□//初始化MMC块请求结构
		memset(&brq, 0, sizeof(struct mmc_blk_request));
		brq.mrq.cmd = &brq.cmd;
		brq.mrq.data = &brq.data;
```

```
            brq.cmd.arg = req->sector << 9;
            brq.cmd.flags = MMC_RSP_R1;
            brq.data.timeout_ns = card->csd.tacc_ns * 10;
            brq.data.timeout_clks = card->csd.tacc_clks * 10;
            brq.data.blksz_bits = md->block_bits;
            brq.data.blocks = req->nr_sectors >> (md->block_bits - 9);
            brq.stop.opcode = MMC_STOP_TRANSMISSION;
            brq.stop.arg = 0;
            brq.stop.flags = MMC_RSP_R1B;

            if (rq_data_dir(req) == READ) {//读请求
                  brq.cmd.opcode = brq.data.blocks > 1 ?
MMC_READ_MULTIPLE_BLOCK : MMC_READ_SINGLE_BLOCK;
                  brq.data.flags |= MMC_DATA_READ;
            } else {//写
                  brq.cmd.opcode = MMC_WRITE_BLOCK;
                  brq.cmd.flags = MMC_RSP_R1B;
                  brq.data.flags |= MMC_DATA_WRITE;
                  brq.data.blocks = 1;
            }
            brq.mrq.stop = brq.data.blocks > 1 ? &brq.stop : NULL;

            brq.data.sg = mq->sg;
            brq.data.sg_len = blk_rq_map_sg(req->q, req, brq.data.sg);
□□□□//等待请求完成
            mmc_wait_for_req(card->host, &brq.mrq);
            ......
            do {
                  int err;

                  cmd.opcode = MMC_SEND_STATUS;
                  cmd.arg = card->rca << 16;
                  cmd.flags = MMC_RSP_R1;
                  err = mmc_wait_for_cmd(card->host, &cmd, 5);
                  if (err) {
                        printk(KERN_ERR "%s: error %d requesting
status\n",
                                    req->rq_disk->disk_name, err);
                        goto cmd_err;
                  }
            } while (!(cmd.resp[0] & R1_READY_FOR_DATA));

            //一个块被成功传输
            spin_lock_irq(&md->lock);
            ret = end_that_request_chunk(req, 1,
brq.data.bytes_xfered);
```

```
            if (!ret) {
                  //整个请求完全成功完成
                  add_disk_randomness(req->rq_disk);
                  blkdev_dequeue_request(req);//从队列中删除请求
                  end_that_request_last(req);//写一些更新信息
            }
            spin_unlock_irq(&md->lock);
      } while (ret);

      mmc_card_release_host(card);

      return 1;

 cmd_err:
      mmc_card_release_host(card);

      spin_lock_irq(&md->lock);
      do {
□□□□//结束请求req上的I/O□操作成功时返回0
            ret = end_that_request_chunk(req, 0,
                        □□□□req->current_nr_sectors << 9);
      } while (ret);

      add_disk_randomness(req->rq_disk);
      blkdev_dequeue_request(req);
      end_that_request_last(req);
      spin_unlock_irq(&md->lock);

      return 0;
}
```

函数mmc_card_claim_host发出命令选择这个卡card□函数列出如下□在 include/linuc/mmc/card.h中□□

```
static inline int mmc_card_claim_host(struct mmc_card *card)
{
      return __mmc_claim_host(card->host, card);
}
```

函数__mmc_claim_host专有地认领一个控制器□参数host是认领的mmc控制器□参数card是去认领控制器的卡□函数__mmc_clai

函数__mmc_claim_host分析如下□在drivers/mmc/mmc.c中□□

```
int __mmc_claim_host(struct mmc_host *host, struct mmc_card *card)
{
      DECLARE_WAITQUEUE(wait, current);//给当前进程声明一个等待队列
      unsigned long flags;
      int err = 0;

      add_wait_queue(&host->wq, &wait);//加host->wq到等待队列中
      spin_lock_irqsave(&host->lock, flags);
```

```
        while (1) {
                set_current_state(TASK_UNINTERRUPTIBLE);//设置当前进程不可中断状态
                if (host->card_busy == NULL)//如果没有忙的卡跳出循环
                        break;
                spin_unlock_irqrestore(&host->lock, flags);
                schedule();//如果有忙的卡去调度执行
                spin_lock_irqsave(&host->lock, flags);
        }
        set_current_state(TASK_RUNNING);//设置当前进程为运行状态
        host->card_busy = card;//指定当前忙的卡
        spin_unlock_irqrestore(&host->lock, flags);
        remove_wait_queue(&host->wq, &wait);//从等待队列中移去host->wq
    //如果卡不是选择状态发出命令到卡设置为选择状态
        if (card != (void *)-1 && host->card_selected != card) {
                struct mmc_command cmd;

                host->card_selected = card;

                cmd.opcode = MMC_SELECT_CARD;
                cmd.arg = card->rca << 16;
                cmd.flags = MMC_RSP_R1;
//////等待命令完成
                err = mmc_wait_for_cmd(host, &cmd, CMD_RETRIES);
        }

        return err;
}
```

函数mmc_wait_for_req开始执行一个请求并等待请求完成函数分析如下在drivers/mmc/mmc.c中

```
int mmc_wait_for_req(struct mmc_host *host, struct mmc_request *mrq)
{
        DECLARE_COMPLETION(complete);

        mrq->done_data = &complete;
        mrq->done = mmc_wait_done;

        mmc_start_request(host, mrq);

        wait_for_completion(&complete);

        return 0;
}
```

函数mmc_start_request开始排队执行一个在控制器上的命令参数host是执行命令的控制器参数mrq是将要开始执行的请求

函数mmc_start_request分析如下在drivers/mmc/mmc.c中

```
void mmc_start_request(struct mmc_host *host, struct mmc_request *mrq)
{
```

```
        DBG("MMC: starting cmd %02x arg %08x flags %08x\n",
            mrq->cmd->opcode, mrq->cmd->arg, mrq->cmd->flags);

        WARN_ON(host->card_busy == NULL);

        mrq->cmd->error = 0;
        mrq->cmd->mrq = mrq;
        if (mrq->data) {
                mrq->cmd->data = mrq->data;
                mrq->data->error = 0;
                mrq->data->mrq = mrq;
                if (mrq->stop) {
                        mrq->data->stop = mrq->stop;
                        mrq->stop->error = 0;
                        mrq->stop->mrq = mrq;
                }
        }
    //调用请求处理函数，对于amba主控制器来说就是mmci_request函数
        host->ops->request(host, mrq);
}
```

具体MMC控制器驱动程序示例

下面以amba控制器为例分析MMC卡驱动程序，amba控制器是集成在ARM处理器中的MMC卡控制器。

（1）amba控制器驱动程序相关结构

amba控制器驱动程序相关结构在include/asm-arm/hardware/amba.h中，分别说明如下：

结构amba_device

是amba控制器设备结构，它在通用设备结构的基础上封装了amba控制器选定的信息数据，列出如下：

```
struct amba_device {
        struct device             dev;　//通用设备结构
        struct resource             res;　//设备资源结构
        u64                     dma_mask;
        unsigned int              periphid;
        unsigned int              irq[AMBA_NR_IRQS];　
};

struct amba_id {
        unsigned int              id;
        unsigned int              mask;
        void                     *data;
};
```

结构amba_driver是amba控制器驱动程序描述结构，列出如下：

```
struct amba_driver {
        struct device_driver        drv;　//通用驱动程序结构
        int                     (*probe)(struct amba_device *, void *);
```

```
        int                     (*remove)(struct amba_device *);
        void            (*shutdown)(struct amba_device *);
        int                     (*suspend)(struct amba_device *, u32);
        int                     (*resume)(struct amba_device *);
        struct amba_id              *id_table;
};
```

J

amba控制器的描述结构封装了MMC通用的主控制器结构、MMC请求结构、MMC命令结构、MMC数据结构等。

```
struct mmci_host {
    //下面mmc_*结构是MMC设备抽象层的各种设备结构
        void __iomem            *base;
        struct mmc_request      *mrq;  //MMC读写请求
        struct mmc_command      *cmd;
        struct mmc_data             *data;
        struct mmc_host             *mmc;
        struct clk              *clk;

        unsigned int              data_xfered;

        spinlock_t              lock;

        unsigned int              mclk;
        unsigned int              cclk;
        u32                     pwr;
        struct mmc_platform_data *plat;  //平台中与MMC相关的数据

        struct timer_list       timer;  //定时器
        unsigned int              oldstat;

        unsigned int              sg_len;

        /* pio stuff */
        struct scatterlist        *sg_ptr;  //碎片链表指针
        unsigned int              sg_off;
        unsigned int              size;
}
```

（2）**amba**控制器的初始化

函数mmci_init 是amba控制器的初始化，它注册了amba控制器驱动程序结构mmci_driver。

amba控制器驱动程序的初始化函数和模块清除函数分别列出如下：

```
static int __init mmci_init(void)
{
        return amba_driver_register(&mmci_driver);
}

static void __exit mmci_exit(void)
```

```
{
	amba_driver_unregister(&mmci_driver);
}
```

amba控制器驱动程序结构实例mmci_driver列出如下：

```
#define DRIVER_NAME "mmci-pl18x"
static struct amba_driver mmci_driver = {
	.drv		= {
		.name	= DRIVER_NAME,
	},
	.probe		= mmci_probe,　　//设备探测及初始化函数
	.remove		= mmci_remove,　//移去设备处理函数
	.suspend	= mmci_suspend,　//电源挂起函数
	.resume		= mmci_resume,　//电源恢复函数
	.id_table	= mmci_ids,
};
```

（3）设备探测函数**mmci_probe**

函数mmci_probe探测设备并初始化设备，它的工作包括：申请设备I/O内存并进行I/O映射，初始化控制器结构，激活设备时钟，申

函数mmci_probe分析如下：在drivers/mmc/mmci.c中：

```
static int mmci_probe(struct amba_device *dev, void *id)
{
	struct mmc_platform_data *plat = dev->dev.platform_data;
	struct mmci_host *host;
	struct mmc_host *mmc;
	int ret;

	/* must have platform data */
	if (!plat) {
		ret = -EINVAL;
		goto out;
	}
    //申请所有与设备相关的I/O内存区域：
	ret = amba_request_regions(dev, DRIVER_NAME);
	if (ret)
		goto out;

    //分配并初始化主控制器结构mmc_host：
	mmc = mmc_alloc_host(sizeof(struct mmci_host), &dev->dev);
	if (!mmc) {
		ret = -ENOMEM;
		goto rel_regions;
	}

	// 得到mmc对应有mmci结构，即host = ((void *)((mmc) + 1))
host = mmc_priv(mmc);
//得到名为“MCLK”的时钟结构，它描述了一个硬件时钟的信息
```

```
	host->clk = clk_get(&dev->dev, "MCLK");
	if (IS_ERR(host->clk)) {
		ret = PTR_ERR(host->clk);
		host->clk = NULL;
		goto host_free;
	}
□□//时钟是否使用
	ret = clk_use(host->clk);
	if (ret)
		goto clk_free;
□□
□□//激活时钟
	ret = clk_enable(host->clk);
	if (ret)
		goto clk_unuse;

	host->plat = plat;
	host->mclk = clk_get_rate(host->clk);//得到时钟频率
	host->mmc = mmc;
	host->base = ioremap(dev->res.start, SZ_4K); //I/O映射的虚拟地址
	if (!host->base) {
		ret = -ENOMEM;
		goto clk_disable;
	}

□□//具体设备的控制器操作函数
	mmc->ops = &mmci_ops;
	mmc->f_min = (host->mclk + 511) / 512;
	mmc->f_max = min(host->mclk, fmax);
	mmc->ocr_avail = plat->ocr_mask;

	/*
	 * We can do SGIO
	 */
	mmc->max_hw_segs = 16;
	mmc->max_phys_segs = NR_SG;//定义为16

	//因为仅有一个16位数据长度寄存器□我们必须保证一个请求中不能超过 -1□
//选择64□512字节□扇区作为限制□
	mmc->max_sectors = 64;

	//设置最大片断尺寸□因为不做DMA□仅受限制于数据长度寄存器*
	mmc->max_seg_size = mmc->max_sectors << 9;

	spin_lock_init(&host->lock);
□□//写寄存器
	writel(0, host->base + MMCIMASK0);
```

```
        writel(0, host->base + MMCIMASK1);
        writel(0xfff, host->base + MMCICLEAR);
//分配共享中断□中断函数是mmci_irq处理命令及传输数据传输完成时的中断处理函数□中断号是dev->irq[0]□
        ret = request_irq(dev->irq[0], mmci_irq, SA_SHIRQ,
DRIVER_NAME " (cmd)", host);
        if (ret)
                goto unmap;
    //中断mmci_pio_irq是PIO数据传输的中断处理函数
        ret = request_irq(dev->irq[1], mmci_pio_irq, SA_SHIRQ,
DRIVER_NAME " (pio)", host);
        if (ret)
                goto irq0_free;
□□//将中断使能写入寄存器
        writel(MCI_IRQENABLE, host->base + MMCIMASK0);

□□//设置dev->driver_data = mmc□
        amba_set_drvdata(dev, mmc);
□□//初始化主控制器硬件□关控制器电源□检测控制器插槽有否卡□
        mmc_add_host(mmc);

        printk(KERN_INFO "%s: MMCI rev %x cfg %02x at 0x%08lx irq
%d,%d\n",
                mmc->host_name, amba_rev(dev), amba_config(dev),
                dev->res.start, dev->irq[0], dev->irq[1]);
□□//初始化定义器
        init_timer(&host->timer);
        host->timer.data = (unsigned long)host;
        host->timer.function = mmci_check_status;//设置定时函数为检测状态函数
        host->timer.expires = jiffies + HZ;
        add_timer(&host->timer);

        return 0;
......
}
```

函数amba_request_regions申请所有与设备相关的I/O内存区域□其参数dev是设备结构amba_device□参数name若为NULL□表示

函数amba_request_regions分析如下□在arch/arm/commom/amba.c中□□

```
int amba_request_regions(struct amba_device *dev, const char *name)
{
        int ret = 0;

        if (!name)//如果为NULL□表示是设备驱动程序的名字□
                name = dev->dev.driver->name;
          //填写资源结构□I/O空间大小为SZ_4K□并分析是否在
          //全局资源结构iomem_resource所控制的地址范围内□将资源加到资源树中□
        if (!request_mem_region(dev->res.start, SZ_4K, name))
                ret = -EBUSY;
```

```
        return ret;
}
```

函数mmc_alloc_host
分配并初始化主控制器结构mmc_host□其参数extra是私有数据结构的大小□参数dev是主控制器设备模型结构的指针□

```
struct mmc_host *mmc_alloc_host(int extra, struct device *dev)
{
        struct mmc_host *host;
□       □//分配结构对象空间
        host = kmalloc(sizeof(struct mmc_host) + extra, GFP_KERNEL);
        if (host) {
                memset(host, 0, sizeof(struct mmc_host) + extra);//清0

                spin_lock_init(&host->lock);
                init_waitqueue_head(&host->wq);//初始化等待队列
                INIT_LIST_HEAD(&host->cards);□//初始化链表
□□□             □//设置工作的函数及定时器□函数是(&host->detect)-> mmc_rescan(host)
                INIT_WORK(&host->detect, mmc_rescan, host);

                host->dev = dev;

                //缺省时不支持SGIO□多个片断□或大请求□
                   //如需支持必须按控制器的能力设备下面的参数□
                host->max_hw_segs = 1;
                host->max_phys_segs = 1;
                    //计数每页的扇区数=页大小/512
                host->max_sectors = 1 << (PAGE_CACHE_SHIFT - 9);
                host->max_seg_size = PAGE_CACHE_SIZE;
        }

        return host;
}
```

□**4□amba**控制器操作函数

amba控制器操作函数集实例列出如下□

```
static struct mmc_host_ops mmci_ops = {
        .request        = mmci_request,□//amba控制器请求处理函数
        .set_ios        = mmci_set_ios,□//控制设备状态□将控制值写入设备寄存器□
};
```

函数mmci_request把在MMC抽象设备层封装好的MMC协议命令写入具体的amba控制器完成MMC卡的读写操作□

函数mmci_request列出如下□

```
static void mmci_request(struct mmc_host *mmc, struct mmc_request *mrq)
{
        struct mmci_host *host = mmc_priv(mmc);
```

```
        WARN_ON(host->mrq != NULL);

        spin_lock_irq(&host->lock);

        host->mrq = mrq;

        if (mrq->data && mrq->data->flags & MMC_DATA_READ)
              mmci_start_data(host, mrq->data);
□□      //开始执行命令□即向寄存器写入操作指令来执行读写操作□
        mmci_start_command(host, mrq->cmd, 0);

        spin_unlock_irq(&host->lock);
}
```

Article Sources and Contributors

Linux内核MTD驱动程序与SD卡驱动程序 *Source*: http://www.shangshuwu.cn/index.php?oldid=783 *Contributors*: Njlts

Image Sources, Licenses and Contributors

Image:Linux kernel mtd mmc sd driver 13.png *Source*: http://www.shangshuwu.cn/index.php?title=文件:Linux_kernel_mtd_mmc_sd_driver_13.png *License*: unknown *Contributors*: Njlts

Image:Linux kernel mtd mmc sd driver 14 1024.png *Source*: http://www.shangshuwu.cn/index.php?title=文件:Linux_kernel_mtd_mmc_sd_driver_14_1024.png *License*: unknown *Contributors*: Njlts