<MMC card driver code analysis>

Author: adamchen@live.cn
Last Update:12-03-2008
Version: 0.3

ChangeHistory:
12-03-2008: 加入总结
12-02-2008: finished draft
11-30-2008: init version

${KERNEL}/drivers/mmc/card/block.c
${KERNEL}/drivers/mmc/card/queue.c

该驱动和它的名字一样，仅仅作为单纯的 MMC 存储设备（块设备）的驱动程序被使用，也就是说它只负责支持 MMC 卡的存储功能，能把 MMC 卡当作优盘使而已，而不支持基于 MMC 接口的扩展功能(SDIO)。

首先自然是驱动的入口点 init 函数。

```
662
663         res = register_blkdev(MMC_BLOCK_MAJOR, "mmc");
664         if (res)
665                 goto out;
666
667         return mmc_register_driver(&mmc_driver);
```

register_blkdev 向内核 block subsystem 申请注册一个块设备，其中参数分别是要申请设备的主设备号，以及设备名。然后进入 mmc_register_driver，开始 MMC card driver（static struct mmc_driver mmc_driver）向 mmc core 层注册的工作。

暂时进入到 mmc_register_driver。虽然它属于 mmc core 层的一个调用，不过可以先看一眼，有助于理解 card driver 和 core 之间的关系。更具体的分析以后再说。

```
156 int mmc_register_driver(struct mmc_driver *drv)
157 {
158         drv->drv.bus = &mmc_bus_type;
159         return driver_register(&drv->drv);
160 }
```

drv->drv.bus = &mmc_bus_type 把 mmc_bus 操作和 mmc card driver 关联起来，这样当 mmc_bus 做 probe 的时候，最终将会索引到 mmc card driver 的 probe 函数。所以，我们可以估计，在 mmc_bus_type 的 probe 回调里，应该有类似 drv->probe 的调用，事实也是如此。

接下来 dirver_register 则是一股脑的把 mmc card driver 所属的 device driver 相关信息像 device driver 的管理层注册。多说一句，这应该也是 linux 管理设备驱动的一种通用方式。把通用的 device driver 结构通过标准接口 driver_register 注册到内核进行管理。某个特定的 driver，如 mmc card driver，通过在自己的结构 mmc_driver 里面包含 device_driver 的结构，达到从“父类”device_driver，“继承”某些标准数据结构、信息的目的。

下面分析 mmc_driver 里面的几个重要的回调函数。

```
649 static struct mmc_driver mmc_driver = {
650         .drv            = {
651                 .name   = "mmcblk",
652         },
653         .probe          = mmc_blk_probe,
654         .remove         = mmc_blk_remove,
655         .suspend        = mmc_blk_suspend,
656         .resume         = mmc_blk_resume,
657 };
```

mmc_blk_probe：mmc card driver 的初始化函数，在 mmc_bus 进行 probe 的时候将会作为被找到的某个 mmc device driver 的回调 probe 而被调用。

mmc_blk_remove：mmc card driver 的退出函数，做清洁，回收资源。

mmc_blk_suspend/mmc_blk_resume: mmc card driver 电源管理相关的回调函数，负责相应 power manager 模块对 mmc card driver 的召唤。具体的有时间分析。

接下来就着重分析 mmc card driver 里最重要的一个回调函数 mmc_blk_probe。probe 函数如此的重要，因为在函数里面不仅仅是分配和初始化设备资源，同时也对 mmc card driver 在初始化完成以后的漫长的 run time 里面相关的回调函数做好了配置。所以，我们可以说 probe 函数已经做好了所有的准备工作，同时也定制好了设备以后的运行规则。设备以后要做的，仅仅是按照 probe 预订的规则来工作。

从代码来看 mmc_blk_probe 主要做了 3 件事情。mmc_blk_alloc 初始化和配置了 mmc card 作为 blkdev 的相关参数，如 request_queue, request 的 issue_fn 回调，blkdev 的标准回调，等等。接下来如果 mmc_blk_alloc 成功返回，则根据 SD spec，用 CMD16(MMC_SET_BLKLEN)，对 mmc 每次传输的 blk 大小进行设置。最后 mmc_set_drvdata 和 add_disk。其中 add_disk 也是 linux 块设备层的一个标准接口，表示把 mmc 所包含的标准 disk 信息向 linux filesystem 注册，在这以后，在设备文件系统目录下就可以找到 mmc card 对应的设备节点。其他程序可以通过这个节点操作该 mmc 设备。

进入 mmc_blk_alloc。首先，mmc_blk_alloc 有一个参数(struct mmc_card *card)，这个参数又是由上层调用 mmc_blk_probe 时候传入的。mmc_card 在什么时候创建，又在什么情况下被 bus prove 传递给 mmc card driver，会在后面专门解释。现在要知道的是这个 card 变量，唯一标识了一个 mmc card 的实例，mmc card driver 获取到 card 的时候，card 里面已经包含了插入的那个物理 mmc 卡的相关参数，从而 mmc card driver 可以根据 card 里面的参数对驱动程序本身进行必要的配置。而 struct mmc_card 是一个被 mmc core 层接口所支持的标准数据结构。

```
447     devidx = find_first_zero_bit(dev_use, MMC_NUM_MINORS);
448     if (devidx >= MMC_NUM_MINORS)
449         return ERR_PTR(-ENOSPC);
450     __set_bit(devidx, dev_use);
```

查询全局 bitmap dev_use，找到第一个未曾被使用的 devidx，标记该 devidx 表示已被占用。如果返回的 devidx 超过 32，返回 no space left for dev。表示 linux 当前所支持的 MMC 设备已达上限，无法添加新 card。

```
463     md->read_only = mmc_blk_readonly(card);
```

如果 card 标识为只读，可能因为 card 本身只读，或者 card 的 protect 开关在 read only。标记 driver 也为 read only.

```
470     md->block_bits = 9;
```

标记 blk size 512 btyes, 1<<9.

```
474     md->disk = alloc_disk(1 << MMC_SHIFT);
475     if (md->disk == NULL) {
476         ret = -ENOMEM;
477         goto err_kfree;
478     }
```

调用块设备标准接口，参数表示当前 card 支持的 minor 设备最大范围，目前为 8，同时也表示该 card 最多支持 8 个逻辑分区。

```
486     ret = mmc_init_queue(&md->queue, card, &md->lock);
487     if (ret)
488         goto err_putdisk;
```

mnc_init_queue 初始化 mmc card driver 中 blk dev 相关的 request queue。

进入 mnc_init_queue：

```
131     mq->queue = blk_init_queue(mmc_request, lock);
132     if (!mq->queue)
133         return -ENOMEM;
```

blk_init_queue 是 blk dev 标准接口，注册一个回调函数 mmc_request 和一把锁。每当 request queue 上的 request 需要被处理的时候，blk dev 层就会调用 mmc_request 来处理特定的 request。函数返回一个 request_queue 结构给 mmc card driver 层引用。同时，该结构绑定了刚刚注册的回调函数与同步锁。

当前的实现下，mmc_request 主要任务是唤醒提供服务的内核线程 mmc_queue_thread，让 mmc_queue_thread 来做的处理工作。接下来 mmc_queue_thread 会进入遍历 request_queue，取得每一个 request，调用在 mmc_blk_alloc 中注册的当前驱动的 request_queue.issue_fn 回调函数来处理当前 mmc card 上的每一个 r/w request。

```
138     blk_queue_prep_rq(mq->queue, mmc_prep_request);
```

blk_queue_pre_rq 向刚刚初始化成功的 queue 上再绑定一个 mmc_prep_request 回调函数。该函数的功能是在添加 request 到 queue 之前对 request 做一次类型检查，如果是不被支持的 request，则返回错误，同时打印相关信息。

接下来宏 CONFIG_MMC_BLOCK_BOUNCE 表示是否要启动 MMC Storage 优化传输的功能，基本上就是分配更大的缓冲块来完成数据传输，暂时不看。

```
181     if (!mq->bounce_buf) {
182         blk_queue_bounce_limit(mq->queue, limit);
183         blk_queue_max_sectors(mq->queue, host->max_req_size / 512);
184         blk_queue_max_phys_segments(mq->queue, host->max_phys_segs);
185         blk_queue_max_hw_segments(mq->queue, host->max_hw_segs);
186         blk_queue_max_segment_size(mq->queue, host->max_seg_size);
187
188         mq->sg = kmalloc(sizeof(struct scatterlist) *
189             host->max_phys_segs, GFP_KERNEL);
190         if (!mq->sg) {
191             ret = -ENOMEM;
192             goto cleanup_queue;
193         }
194         sg_init_table(mq->sg, host->max_phys_segs);
195     }
```

上面的代码对 request_queue 的一些参数做了配置。参考 LDDv3 中文版 473 页。

```
197     init_MUTEX(&mq->thread_sem);
198
199     //create kernel thread, which handles all io requests whenever it comes
200     //issue_fn's the actual callback
201     mq->thread = kthread_run(mmc_queue_thread, mq, "mmcqd");
202     if (IS_ERR(mq->thread)) {
203         ret = PTR_ERR(mq->thread);
204         goto free_bounce_sg;
205     }
```

创建一个内核线程，mmc_queue_thread 处理 mmc_request 请求的服务。在 mmc_queue_thread 中，通过 mq->issue_fn(mq, req)，调用 probe 时候注册在 request_queue 上的专门处理 request 的回调函数 mmc_blk_issue_rq 来处理每一个具体的 request。

mmc_queue_thread 代码主要分两步工作。

```
57          spin_lock_irq(q->queue_lock);
58          set_current_state(TASK_INTERRUPTIBLE);
59          if (!blk_queue_plugged(q))
60              //get next io request
61              //elevator algorithm
62              req = elv_next_request(q);
63          mq->req = req;
64          spin_unlock_irq(q->queue_lock);
```

第一步，进入临界区，通过电梯算法获得下一个 request。注意一点， elv_next_request 虽然返回了马上要被处理的 request 结构，但并未把该 request 结构从 request_queue 上删除。删除操作是在 issue_fn 完成该 request 以后再进行的。

```
78          mq->issue_fn(mq, req);
```

第二部，调用回调函数，处理取得的 request。如此循环，直到取得的 request 为空，然后判断是终止进程（kthread_should_stop()被标记），抑或是把自己调度，等待下一次任务处理。

退出 mnc_init_queue，回到 mmc_blk_alloc。

```
492     md->queue.issue_fn = mmc_blk_issue_rq;
493     md->queue.data = md;
```

注册 mmc_blk_issue_rq 到 md->queue，当该 request_queue 上有 request 待处理时， mmc_blk_issue_rq 将被调用。

```
496     md->disk->major = MMC_BLOCK_MAJOR;
497     md->disk->first_minor = devidx << MMC_SHIFT;
498     md->disk->fops = &mmc_bdops; // standart block dev ops
499     md->disk->private_data = md;
500     md->disk->queue = md->queue.queue;
501     md->disk->driverfs_dev = &card->dev;
```

注册相关信息到 mmc_blk_data 包含的标准块设备(gendisk)区。

```
//标准块设备回调函数，参见 ldd
md->disk->fops = &mmc_bdops

//把 mmc 的 request_queue 注册到块设备系统
//从而块设备系统可以通过该 request_queue 处理 MMC 卡上的 io 请求
md->disk->queue = md->queue.queue
```

```
518     blk_queue_hardsect_size(md->queue.queue, 1 << md->block_bits);
```

设置传输的 sector 大小为 512 字节，接口定义参见 lddv3 中文版 464 页、474 页。同 set_capacity(...)。

下面进入 mmc card driver probe 过程的最后一个子函数 mmc_blk_issue_rq。

```
215     struct mmc_blk_request brq;
```

brq 是一个栈上的数据。它从参数 mq 和 req 得到一些配置，填充，然后作为一个参数开始完成 request。一次性，次完成 request 时候都有一个 brq。

```
218     mmc_claim_host(card->host);
```

MMC 卡是插在一个 mmc host 上。所以执行 mmc card 相关操作之前要向当前所属的 mmc host 发出通。如果 host 忙，或因为其他原因无法处理，进程将会被加入 wait_queue，然后被调度。只有当 host 可用时，才被调度回来进入可执行状态，mmc_claim_host 才返回。此时，mmc 卡可以执行操作。

```
232         brq.cmd.arg = req->sector;
```

req->sector 填充到 cmd 的参数。sector 指在设备上开始扇区的索引号，以 512 字节为单位。

```
237             brq.cmd.flags = MMC_RSP_SPI_R1 | MMC_RSP_R1 | MMC_CMD_ADTC;
```

标记 flag，表示 cmd 执行完以后返回值的类型。

```
240             brq.data.blksz = 1 << md->block_bits;
```

传输数据的 blk size，512bytes。每次传送的一个 blk 大小。
block_bit 表示该设备扇区大小，9(1<<9=512bytes)表示该设备所有请求都将定位在山区开始位置，并且每个请求的大小都是扇区大小的整数倍。

```
242             brq.stop.opcode = MMC_STOP_TRANSMISSION;
243             brq.stop.arg = 0;
244             brq.stop.flags = MMC_RSP_SPI_R1B | MMC_RSP_R1B | MMC_CMD_AC;
```

标记 stop 命令的命令号 MMC_STOP_TRANSMISSION (CMD12)，返回值类型。当传输完成、传输发生异常的时候会 host 产生 CMD12 终止传输。

```
246             brq.data.blocks = req->nr_sectors >> (md->block_bits - 9);
247             if (brq.data.blocks > card->host->max_blk_count)
248                     brq.data.blocks = card->host->max_blk_count;
```

brq.data.blocks 待传送的 blk 数目。等于 request 里要求传输的扇区数目除以每一个 mmc 扇区是默认扇区 512bytes 的多少倍。现在扇区大小是 mmc 支持的默认值 9，等于默认扇区值 512bytes，所以传输的 blk 数目就等于要传输的扇区数目。
如果 brq.data.blocks 大于 host 支持一次传输的最大值，以最大值为准。剩下的丢失。

```
256             if (rq_data_dir(req) != READ &&
257                 !(card->host->caps & MMC_CAP_MULTIWRITE) &&
258                 !mmc_card_sd(card))
259                     brq.data.blocks = 1;
```

如不支持 multi-blk 传输，设置 brq.data.blocks 为 1。

```
261             if (brq.data.blocks > 1) {
262                     /* SPI multiblock writes terminate using a special
263                      * token, not a STOP_TRANSMISSION request.
264                      */
265                     if (!mmc_host_is_spi(card->host)
266                                     || rq_data_dir(req) == READ)
267                             brq.mrq.stop = &brq.stop;
268                     readcmd = MMC_READ_MULTIPLE_BLOCK;
269                     writecmd = MMC_WRITE_MULTIPLE_BLOCK;
270             } else {
271                     brq.mrq.stop = NULL;
272                     readcmd = MMC_READ_SINGLE_BLOCK;
273                     writecmd = MMC_WRITE_BLOCK;
274             }
275
276             if (rq_data_dir(req) == READ) {
277                     brq.cmd.opcode = readcmd;
278                     brq.data.flags |= MMC_DATA_READ;
279             } else {
280                     brq.cmd.opcode = writecmd;
281                     brq.data.flags |= MMC_DATA_WRITE;
282             }
```

确定传输方式，单块/多块；以及传输方向，读/写。

```
286             mmc_set_data_timeout(&brq.data, card);
```

设置传输超时限制，sdio 1s, sd 100ms 或其他。

```
295             brq.data.sg_len = mmc_queue_map_sg(mq);
```

映射数据，准备 DMA 传输。Lddv3 中文版 P482

```
305             if (brq.data.blocks !=
306                (req->nr_sectors >> (md->block_bits - 9))) {
307                   data_size = brq.data.blocks * brq.data.blksz;
308                   for (sg_pos = 0; sg_pos < brq.data.sg_len; sg_pos++) {
309                         data_size -= mq->sg[sg_pos].length;
310                         if (data_size <= 0) {
311                               mq->sg[sg_pos].length += data_size;
312                               sg_pos++;
313                               break;
314                         }
315                   }
316                   brq.data.sg_len = sg_pos;
317             }
```

如果待传输 blk 数目不是默认计算值，计算 data_size 的大小，根据 data_size 重新分陪 mq->sg[i].length 长度。

```
319             mmc_wait_for_req(card->host, &brq.mrq);
```

进入 mmc core 开始完成当前 request, mmc core 把 request 分发给 mmc host controller 驱动程序的 request 回调函数处理。当返回时，当前 request 要求的传输以完成，mmc host controller 驱动会完成三步：
1 设置寄存器
2 发出命令，然后返回到 mmc_wait_for_req，mmc card driver 此时阻塞在 complete 上
3 完成命令，产生中断，调用 completion()唤醒 mmc card driver

```
385             ret = __blk_end_request(req, 0, brq.data.bytes_xfered);
```

进入_blk_end_request

```
1938 int __blk_end_request(struct request *rq, int error, unsigned int nr_bytes)
1939 {
1940       if (blk_fs_request(rq) || blk_pc_request(rq)) {
1941             if (__end_that_request_first(rq, error, nr_bytes))
1942                   return 1;
1943       }
1944
1945       add_disk_randomness(rq->rq_disk);
1946
1947       end_that_request_last(rq, error);
1948
1949       return 0;
1950 }
```

_end_that_request_first 检查是否若有待传扇区已传完。
add_disk_randomness 为熵池做贡献。
end_that_request_last 如果已经传完，做清洁，把 request 从 request_queue 里 dequeue 出去。

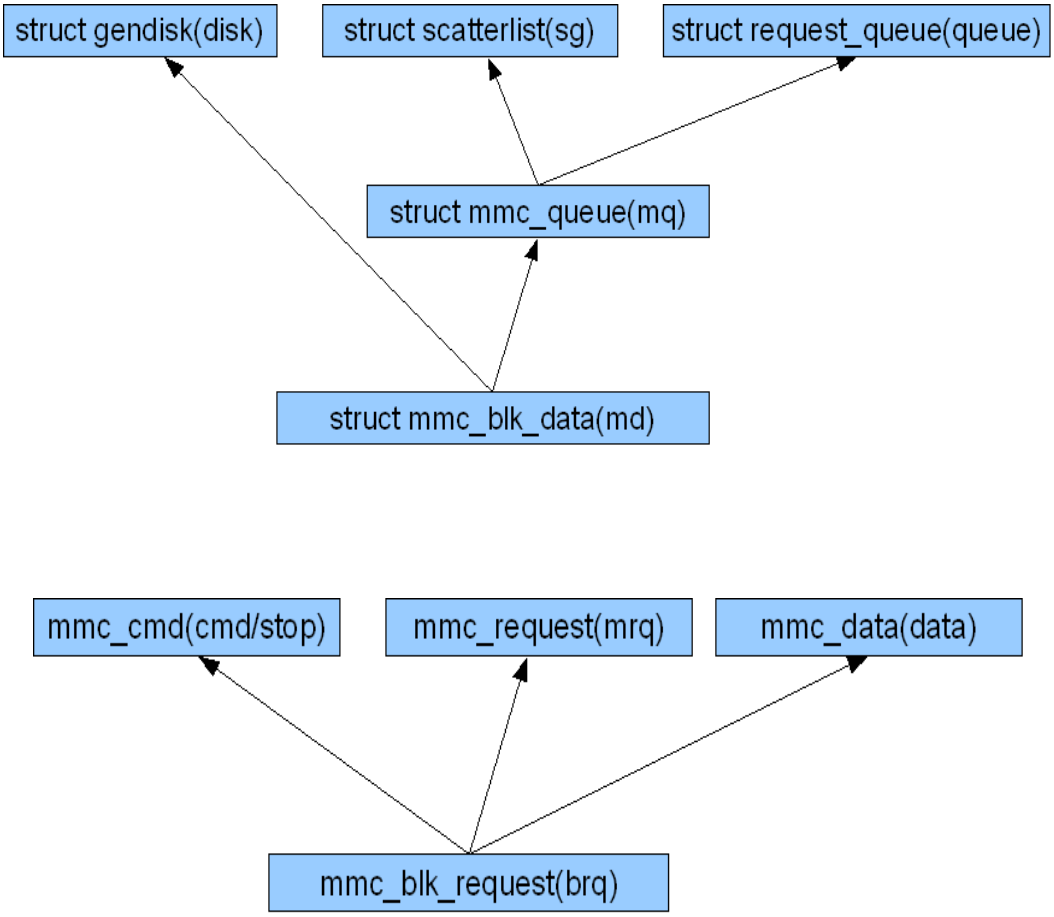
```
389       mmc_release_host(card->host);
```

释放 mmc host，其他设备可以 claim，或被从 wait_queue 唤醒。

Mmc card driver 实现基本如上所述，没有讨论 bounce 的情况，差不多也大同小异，可以自行分析。下一步将会分析 MMC host controller 驱动的实现。

总结一下，mmc card driver 的代码主要就在两个地方。第一，probe 函数，完成初始化和配置的工作。第二，runtime 的一个工作流程，分为一下几步。上层有 request 到来，绑定在设备 request_queue 上的函数 mmc_request 被调用。mmc_request 简单地唤醒内核线程 mmc_queue_thread 来处理 request。mmc_queue_thread 遍历 request_queue，取得每个 request，调用绑定在 mmc 设备上的 mmc_blk_issue_rq 来执行 card driver 部分的 io 传输操作。接下来，则会进入 host controller driver 部分，让 host 来控制真正的数据传输过程。直到完成当前 request，唤醒 complete，返回到 mmc_blk_issue_rq。数据传输的方式为，从 request 的 bio 中，通过 mmc_queue_map_sg 映射到 mmc_queue 的 scatter list 每个 entry 中，再通过 host controller driver 里的 dma_map_sg，映射到 DMA。然后进行传输。

最后 mmc driver 关键数据结构的继承图如下，作为参考：



参考文档：
ldd v3
sd simple specification