**SD卡驱动学习笔记**

以TI DM365为例，内核版本2.6.18，he\_liangbin 2012-7-8

1. **MMC/SD简介**
2. **概念**

（1）MMC：(Multi Media Card)

（2）SD：(Secure Digital Memory Card)，完全兼容 MMC 标准。

（3）SDIO：(Secure Digital Input and Output Card)安全数字输入输出卡。SDIO 是在SD 标准上定义了一种外设接口，通过 SD 的 I/O 接脚来连接外围设备，并且通过 SD

上的 I/O 数据接位与这些外围设备进行数据传输。

1. **工作模式**

工作模式是针对主机控制器来说的，SDI 控制器可以在符合 MMC 的标准下工作，或者可以在符合 SD 的标准下工作，或者可以在符合SDIO 的标准下工作。故就分别简称为：MMC 模式、SD 模式和 SDIO 模式。

1. **传输模式**

传输模式也是针对主机控制器来说的，指控制器与卡之间数据的传输模式， SDI 控制器可支持 SPI、1 位和 4 位的三种传输模式

1. **分类**

按体积大小，普通SD卡，mini-SD卡，micro-SD卡（TF卡）

按存储大小，普通SD卡（<=2GB，支持FAT12/FAT16），HCSD卡(>2GB,<=32GB，支持FAT32)。

1. **MMC/SD卡协议**
2. **总线协议**

SD总线通信是基于命令和数据位流方式的，由一个起始位开始，以一个停止位结束。

命令——命令是开始开始操作的标记。命令从主机发送一个卡（寻址命令）或所有

接的卡（广播命令）。命令在CMD线上串行传送。

响应——响应是从寻址卡或所有连接的卡（同步）发送给主机用来响应接受到的命

的标记。命令在CMD线上串行传送。

数据——数据可以通过数据线在卡和主机间双向传送。

1. **传输方式：空操作、写操作、读操作**

首先由主机向SD卡发送命令command，等待SD卡的回复response，如果成功收到回复，则进行数据传输。其中，指令线和数据线上传输的指令和数据都要遵循相应的协议格式。

1. 空操作



1. 写操作

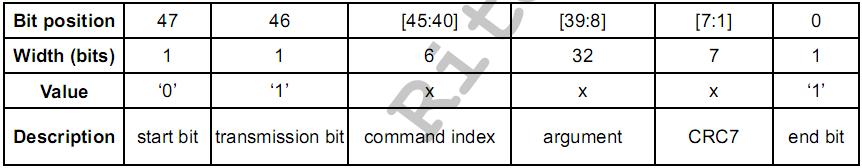


1. 读操作



1. **命令格式**

一条指令command共48位，其中command index指代这条具体的指令名称，argument为该指令的参数。格式如图：



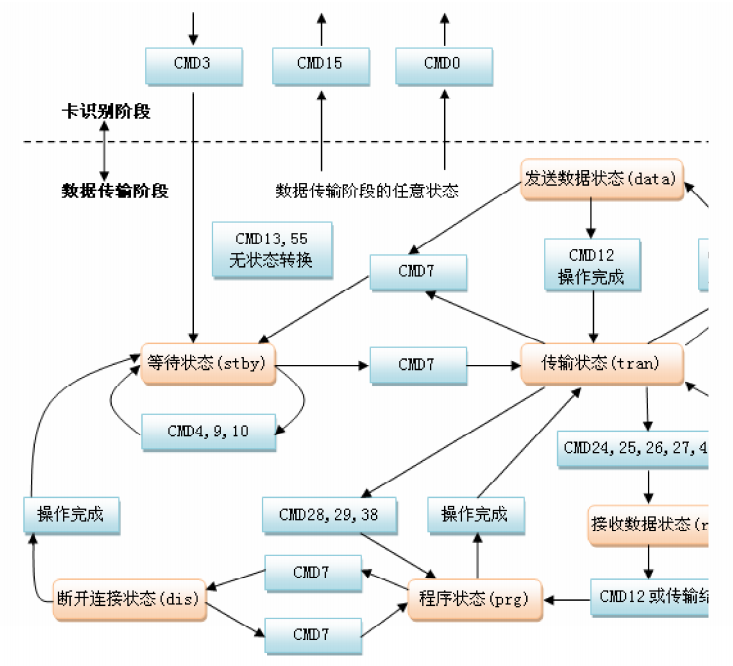
1. **MMC/SD卡驱动阶段：卡识别阶段和数据传输阶段**
2. 卡识别阶段

在卡识别阶段通过命令使 MMC/SD 处于：空闲(idle)、准备(ready)、识别(ident)、等待(stby)、不活动(ina)几种不同的状态。



1. 数据传输阶段

在数据传输阶段通过命令使 MMC/SD 处于：发送(data)、传输(tran)、接(rcv)、程序(prg)、断开连接(dis)几种不同的状态



1. **MMC/SD子系统代码结构**
2. **子系统源码目录**

在 Linux 中 MMC/SD 卡的记忆体都当作块设备。MMC/SD 设备驱动代码在kernel\drivers\mmc 下面，主要包含如下：davinci-mmc.c 、mmc.c、 mmc-block.c、 mmc-queue.c、 mmc-sysfs.c。

1. **子系统划分**

MMC子系统可以划分为三个部分：card、 core、 host层，其中前两个层内核提供，host层需要动手实现跟设备关系密切。

1. card层

（mmc-block.c、 mmc-queue.c、 mmc-sysfs.c）：要把操作的数据以块设备的处理方式写到记忆体上或从记忆体上读取，使SD 卡实现为块设备的。

1. core层（

（mmc.c）：将数据以何种格式，何种方式在 MMC/SD 主机控制器与 MMC/SD 卡的记忆体(即块设备)之间进行传递。实现不同规范和协议，为host层的驱动层提供接口函数。

1. host层

（davinci-mmc.c）：要动手实现的具体 MMC/SD 设备驱动了，包括 RAM 芯片中的SDI控制器(支持对 MMC/SD 卡的控制，俗称MMC/SD 主机控制器)和 SDI控制器 与MMC/SD 卡的硬件接口电路。

1. **层次关系**

子系统三个部分的层次关系如图所示:



1. **MMC/SD驱动程序分析**
2. **重要数据结构**
3. host主控制器struct mmc\_davinci\_host

该结构体位于davinci-mmc.c中，用于存储SD相关的所有数据信息。

struct mmc\_davinci\_host {

. . . .

struct mmc\_request \*que\_mmc\_request;

struct mmc\_command \*cmd;

struct mmc\_data \*data;

struct mmc\_host \*mmc;//贯穿三个层的核心结构体

struct device \*dev;

struct clk \*clk;

. . . .

}

1. MMC核心结构体struct mmc\_host

该结构体位于host.h，可以认为是linux为SD卡控制器专门准备的一个类，该类里面的成员是所有SD卡控制器都需要的，放之四海而皆准的数据结构，而在PXA芯片控制器的驱动程序pxamci.c中，则为该类具体化了一个对象struct mmc\_host \*mmc。

struct mmc\_host {

struct device \*dev;

struct class\_device class\_dev;

const struct mmc\_host\_ops \*ops; // SD卡主控制器的操作函数，即该控制器所具备的驱动能力

struct mmc\_ios  ios;  // 配置时钟、总线、电源、片选、时序

struct mmc\_card \*card\_selected;//连接到此主控制器的SD卡设备

. . . .

struct work\_struct detect;

unsigned long private[0] \_\_\_\_cacheline\_aligned;

}

1. struct mmc\_host\_ops实现主控制器的基本操作

定义了对 host 主机进行操作的各种方法，其定义在 Core 核心层的,也就是Core核心层对Host主机层提供的接口函数, 他是Core层与Host层进行数据交换的载体。

struct mmc\_host\_ops {

//核心函数，完成主控器与SD卡设备之间的数据通信

void (\*request)(struct mmc\_host \*host, struct mmc\_request \*req);

//配置时钟、总线、电源、片选、时序

void (\*set\_ios)(struct mmc\_host \*host, struct mmc\_ios \*ios);

int (\*get\_ro)(struct mmc\_host \*host);

};

1. struct mmc\_request

struct mmc\_request {

struct mmc\_command \*cmd;

struct mmc\_data \*data;

struct mmc\_command \*stop;

void \*done\_data; /\* completion data \*/

void (\*done)(struct mmc\_request \*);/\* completion function \*/

};

1. struct mmc\_command

struct mmc\_command {

u32 opcode;//对应命令格式中的command index

u32 arg; //对应argument

u32 resp[4]; //对应response

. . . .

struct mmc\_data \*data; /\* data segment associated with cmd \*/

struct mmc\_request \*mrq; /\* associated request \*/

};

1. **host层驱动**
2. 添加MMC/SD平台设备

在 Linux 中，MMC/SD 设备是被作为平台设备添加到系统的，可以查看内核代码：/arch/arm/mach-davinci/board\_dm365\_evm.c中为MMC/SD主机控制器SDI定义了平台设备和平台设备资源，然后在系统初始化的时候添加到系统中.

static struct resource mmc1\_resources[] = {

[0] = { /\* registers \*/

.start = DM365\_MMC\_SD1\_BASE,

.end = DM365\_MMC\_SD1\_BASE + SZ\_1K - 1,

.flags = IORESOURCE\_MEM,

},

[1] = { /\* interrupt \*/

.start = IRQ\_DM3XX\_MMCINT1,

.end = IRQ\_DM3XX\_MMCINT1,

.flags = IORESOURCE\_IRQ,

},

[2] = { /\* dma rx \*/

.start = DM365\_DMA\_MMC1RXEVT,

.end = DM365\_DMA\_MMC1RXEVT,

.flags = IORESOURCE\_DMA | IORESOURCE\_DMA\_RX\_CHAN,

},

[3] = { /\* dma tx \*/

.start = DM365\_DMA\_MMC1TXEVT,

.end = DM365\_DMA\_MMC1TXEVT,

.flags = IORESOURCE\_DMA | IORESOURCE\_DMA\_TX\_CHAN,

},

[4] = { /\* event queue \*/

.start = EVENTQ\_3,

.end = EVENTQ\_3,

.flags = IORESOURCE\_DMA | IORESOURCE\_DMA\_EVENT\_Q,

},

};

static struct davinci\_mmc\_platform\_data mmc1\_platform\_data = {

.mmc\_clk = "MMCSDCLK1",

.rw\_threshold = 64,

.use\_4bit\_mode = 1,

.use\_8bit\_mode = 0,

.max\_frq = 50000000,

.pio\_set\_dmatrig = 1,

};

static struct platform\_device mmc1\_device = {

.name = "davinci-mmc",

.id = 1,

.dev = {

.platform\_data = &mmc1\_platform\_data,

},

.num\_resources = ARRAY\_SIZE(mmc1\_resources),

.resource = mmc1\_resources,

};

static struct platform\_device \*dm365\_evm\_devices[] \_\_initdata = {

&serial\_device,

&mmc0\_device,

&mmc1\_device,

&nor\_device,

&rtc\_device,

&dm365\_kp\_device,

};

static \_\_init void dm365\_evm\_init(void)

{

. . . .

platform\_add\_devices(dm365\_evm\_devices,ARRAY\_SIZE(dm365\_evm\_devices));

}

1. 注册设备

相应结构体初始化

static struct platform\_driver davinci\_mmcsd\_driver = {

.probe = davinci\_mmc\_probe,

.remove = davinci\_mmcsd\_remove,

.suspend = davinci\_mmcsd\_suspend,

.resume = davinci\_mmcsd\_resume,

.driver = {

.name = DRIVER\_NAME,

},

};

static int davinci\_mmcsd\_init(void)

{

return platform\_driver\_register(&davinci\_mmcsd\_driver);

}

static void \_\_exit davinci\_mmcsd\_exit(void)

{

platform\_driver\_unregister(&davinci\_mmcsd\_driver);

}

1. 探针函数davinci\_mmc\_probe

在驱动的接口函数中davinci\_mmc\_probe() 函数，用于分配 mmc\_host ，mmc\_davinci\_host结构体，并对结构体进行设置，在 SDI 主机控制器操作接口函数 mmc\_host\_ops 中会调用 mmc\_davinci\_host结构体，申请中断并设置中断服务函数，将结构体 mmc\_host添加到主机。函数原型及部分关键代码如下：

static int davinci\_mmc\_probe(struct platform\_device \*pdev)

{

. . . .

/\* 获取资源，该资源在MMC/SD平台添加时定义好 \*/

res = platform\_get\_resource(pdev, IORESOURCE\_MEM, 0);

irq = platform\_get\_irq(pdev, 0);

. . .

/\* 分配 mmc\_host ，mmc\_davinci\_host结构体, 里面包含mmc\_rescan()，该函数很重要\*/

mmc = mmc\_alloc\_host(sizeof(struct mmc\_davinci\_host), &pdev->dev);

. . . .

init\_mmcsd\_host(host);

. . . .

mmc->ops = &mmc\_davinci\_ops;

. . . .

ret = request\_irq(host->irq, mmc\_davinci\_irq, 0, DRIVER\_NAME, host);

platform\_set\_drvdata(pdev, host);

mmc\_add\_host(mmc);

. . . .

}

1. MMC操作接口函数

static struct mmc\_host\_ops mmc\_davinci\_ops = {

.request = mmc\_davinci\_request,

.set\_ios = mmc\_davinci\_set\_ios,

.get\_ro = mmc\_davinci\_get\_ro

};

1. 申请中断

davinci\_mmc\_probe (struct platform\_device \*pdev)中有个中断，为SD主控制器芯片内电路固有的内部中断。当调用(\*request)，即host->ops->request(host, mrq)，即上文中的mmc\_davinci\_request ()后，控制器与SD卡之间开始进行一次指令或数据传输，通信完毕后，主控芯片将产生一个内部中断，以告知此次指令或数据传输完毕。

1. **core层驱动**

core层提供了一系列SD卡的接口服务函数，其中，最为核心的一个函数便是之前提到的位于mmc.c的mmc\_rescan()。

1. mmc\_rescan()

mmc\_rescan扫描SD总线上是否存在SD卡，具体的实现方法就是通过向SD卡上电，看是否能成功;如果上电成功，则返回0，进行总线与SD卡的绑定。如果上电失败，则返回非0值，尝试其他上电的方法。

函数原型及部分关键代码：

static void mmc\_rescan(void \*data)

{

. . . .

/\* 扫描SD卡 \*/

mmc\_setup(host);

. . . .

/\* 注册SD卡, 即进行总线与SD卡的绑定\*/

list\_for\_each\_safe(l, n, &host->cards) {

struct mmc\_card \*card = mmc\_list\_to\_card(l);

/\*

\* If this is a new and good card, register it.

\*/

if (!mmc\_card\_present(card) && !mmc\_card\_dead(card)) {

if (mmc\_register\_card(card))

mmc\_card\_set\_dead(card);

else

mmc\_card\_set\_present(card);

}

/\*

\* If this card is dead, destroy it.

\*/

if (mmc\_card\_dead(card)) {

list\_del(&card->node);

mmc\_remove\_card(card);

}

}

. . . .

}

1. 识别卡函数mmc\_setup()

函数原型及关键代码：

static void mmc\_setup(struct mmc\_host \*host)

{

. . .

/\* 将总线和host联系起来 \*/

mmc\_discover\_cards(host);

. . .

}

1. 注册SD卡mmc\_register\_card()

函数原型及关键代码：

int mmc\_register\_card(struct mmc\_card \*card)

{

. . . .

ret = device\_add(&card->dev);

. . . .

}

1. **card层驱动**
2. 总线驱动数据结构struct bus\_type

该结构体主要是包含总线属性，挂载到该总线的设备属性和驱动属性，及总线操作接口。其部分代码如下：

struct bus\_type {

. . . .

struct bus\_attribute \* bus\_attrs;

struct device\_attribute \* dev\_attrs;

struct driver\_attribute \* drv\_attrs;

int (\*match)(struct device \* dev, struct device\_driver \* drv);

int (\*uevent)(struct device \*dev, char \*\*envp,

int num\_envp, char \*buffer, int buffer\_size);

int (\*probe)(struct device \* dev);

int (\*remove)(struct device \* dev);

void (\*shutdown)(struct device \* dev);

int (\*suspend)(struct device \* dev, pm\_message\_t state);

int (\*resume)(struct device \* dev);

};

1. 总线驱动注册

相应结构体初始化：

static struct bus\_type mmc\_bus\_type = {

.name = "mmc",

.dev\_attrs = mmc\_dev\_attrs,

.match = mmc\_bus\_match,

.uevent = mmc\_bus\_uevent,

.probe = mmc\_bus\_probe,

.remove = mmc\_bus\_remove,

.suspend = mmc\_bus\_suspend,

.resume = mmc\_bus\_resume,

};

static struct class mmc\_host\_class = {

.name = "mmc\_host",

.release = mmc\_host\_classdev\_release,

};

注册和注销总线：

static int \_\_init mmc\_init(void)

{

int ret;

workqueue = create\_singlethread\_workqueue("kmmcd");

if(!workqueue)

return -ENOMEM;

ret = bus\_register(&mmc\_bus\_type);

if (ret == 0) {

ret = class\_register(&mmc\_host\_class);

if (ret)

bus\_unregister(&mmc\_bus\_type);

}

return ret;

}

static void \_\_exit mmc\_exit(void)

{

class\_unregister(&mmc\_host\_class);

bus\_unregister(&mmc\_bus\_type);

destroy\_workqueue(workqueue);

}

1. 块驱动结构体

struct mmc\_driver为MMC设备驱动结构体，它提供了MMC块设备的基本操作接口。其代码原型如下：

struct mmc\_driver {

struct device\_driver drv;

int (\*probe)(struct mmc\_card \*);

void (\*remove)(struct mmc\_card \*);

int (\*suspend)(struct mmc\_card \*, pm\_message\_t);

int (\*resume)(struct mmc\_card \*);

};

struct block\_device\_operations为通用块设备基本操作接口，其代码原型如下：struct block\_device\_operations {

int (\*open) (struct inode \*, struct file \*);

int (\*release) (struct inode \*, struct file \*);

int (\*ioctl) (struct inode \*, struct file \*, unsigned, unsigned long);

long (\*unlocked\_ioctl) (struct file \*, unsigned, unsigned long);

long (\*compat\_ioctl) (struct file \*, unsigned, unsigned long);

int (\*direct\_access) (struct block\_device \*, sector\_t, unsigned long \*);

int (\*media\_changed) (struct gendisk \*);

int (\*revalidate\_disk) (struct gendisk \*);

int (\*getgeo)(struct block\_device \*, struct hd\_geometry \*);

struct module \*owner;

};

1. 块驱动注册

相应结构体初始化：

static struct mmc\_driver mmc\_driver = {

.drv = {

.name = "mmcblk",

},

.probe = mmc\_blk\_probe,

.remove = mmc\_blk\_remove,

.suspend = mmc\_blk\_suspend,

.resume = mmc\_blk\_resume,

};

static struct block\_device\_operations mmc\_bdops = {

.open = mmc\_blk\_open,

.release = mmc\_blk\_release,

.getgeo = mmc\_blk\_getgeo,

.owner = THIS\_MODULE,

};

其中接口函数mmc\_blk\_probe()中的mmc\_blk\_alloc()调用了mmc\_bdops。

注册和注销设备：

static int \_\_init mmc\_blk\_init(void)

{

int res = -ENOMEM;

res = register\_blkdev(major, "mmc");

. . . .

return mmc\_register\_driver(&mmc\_driver);

}

static void \_\_exit mmc\_blk\_exit(void)

{

mmc\_unregister\_driver(&mmc\_driver);

unregister\_blkdev(major, "mmc");

}

其中的mmc\_register\_driver()将块设备和总线联系在一起：

int mmc\_register\_driver(struct mmc\_driver \*drv)

{

drv->drv.bus = &mmc\_bus\_type;

return driver\_register(&drv->drv);

}

1. **各个驱动联系流程图**
2. host底层驱动流程图

平台设备初始化，平台驱动初始化如图所示。





1. core层mmc\_rescan()函数执行流程



1. 块和总线驱动注册



1. request及数据传输实现



1. **总体架构**
2. **总体启动架构**

kernel启动时，先后执行mmc\_init()及mmc\_blk\_init()，以对mmc设备及mmc块模块进行初始化。然后在挂载mmc设备驱动时，执行驱动程序中的davinci\_mmc\_probe ()，检测host设备中挂载的sd设备。此时probe函数会创建一个host设备，然后开启一个延时任务mmc\_rescan()。驱动挂载成功后，mmc\_rescan()函数被执行，然后对卡进行初始化（步骤后面详细讲述）。确定当前插入的卡是一张有效、可识别的存储卡。然后调用mmc\_add\_host ()把存储卡加到系统中。正式与系统驱动连接在一起。

卡设备加到系统中后，通知mmc块设备驱动。块设备驱动此时调用probe函数，即mmc\_blk\_probe()函数，mmc\_blk\_probe()首先分配一个新的mmc\_blk\_data结构变量，然后调用mmc\_init\_queue，初始化blk队列。然后建立一个线程 mmc\_queue\_thread()。

1. **卡初始化流程**

a、发送CMD0使卡进入IDLE状态

b、发送CMD8，检查卡是否SD2.0。SD1.1是不支持CMD8的，因此在SD2.0 Spec中提出了先发送CMD8，如响应为无效命令，则卡为SD1.1，否则就是SD2.0（请参考SD2.0 Spec）。

c、发送CMD5读取OCR寄存器。

d、发送ACMD55、CMD41，使卡进入工作状态。MMC卡并不支持ACMD55、CMD41，如果这步通过了，则证明这张卡是SD卡。

e、如果d步骤错误，则发送CMD1判断卡是否为MMC。SD卡不支持CMD1，而MMC卡支持，这就是SD和MMC类型的判断依据。

f、如果ACMD41和CMD1都不能通过，那这张卡恐怕就是无效卡了，初始化失败。

1. **核心任务**

MMC/SD卡的驱动整个架构由三个文件夹组成，其实一共就做两件事：

卡的检测和卡数据的读取。

1. 卡的检测涉及到的函数

xxx\_mmc\_probe()(host/xxx\_mmc.c底层驱动)

mmc\_alloc\_host()(core/core.c)

mmc\_rescan()(core/core.c)

mmc\_attach\_mmc()(core/mmc.c)

mmc\_init\_card()(core/mmc.c)

mmc\_add\_card()(core/bus.c)

device\_add()

mmc\_bus\_match()(core/bus.c)

mmc\_bus\_probe()(core/bus.c)

alloc\_disk/add\_disk()

mmc\_blk\_probe()(card/block.c)

1. 卡中数据读写涉及到的函数

mmc\_blk\_issue\_rq()(card/block.c)

mmc\_wait\_for\_req()(core/core.c)

mmc\_start\_request()(core/core.c)

host->ops->request(host,mrq)(底层驱动host中的xxx\_mmc\_request)。