# 驱动框架

MTD(Memory Technology Device)指的的我们常说的flash，包括nand flash、nor flash等。Linux 的MTD子系统则就是针对flash设备设计的驱动框架。对于应用层，可通过字符设备节点或块设备节点直接访问设备，当然也可以通过挂载使用flash设备相关的文件系统。



# 设备注册

用于描述MTD设备的数据结构为mtd\_info，不仅包含了MTD设备的信息还包含了设备的操作方法：

struct mtd\_info {

u\_char type;//MTD设备类型，包括MTD\_RAM，MTD\_ROM、MTD\_NORFLASH、MTD\_NANDFLASH等

uint32\_t flags;//标志，有MTD\_WRITEABLE、MTD\_BIT\_WRITEABLE、MTD\_NO\_ERASE、MTD\_UP\_LOCK

uint64\_t size;//MTD的总大小

uint32\_t erasesize;//擦除块大小

uint32\_t writesize;//写大小，对于nor flash为1，nand flash为页、半页或者1/4页大小

uint32\_t writebufsize;//写缓冲大小，对于nand flash等于writesize，一些nor flash有写缓冲则会大于writesize

uint32\_t oobsize; //每个block的OOB数据大小，OOB(out of band),nand flash中每块后都有一个oob区域，用于存放硬件ecc校验码、坏块标记、和文件系统的组织信息，主要用于硬件纠错和坏块处理。

uint32\_t oobavail;// 每个block的可用OOB数据大小

unsigned int erasesize\_shift;

unsigned int writesize\_shift;

unsigned int erasesize\_mask;

unsigned int writesize\_mask;

unsigned int bitflip\_threshold;//位翻转阈值,即最大允许位翻转个数,超出后,读操作返回“-EUCLEAN”，不设置的话默认为ecc\_strength

const char \*name;

int index;

int (\*\_erase) (struct mtd\_info \*mtd, struct erase\_info \*instr);

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);

void \*priv;

struct module \*owner;

struct device dev;

int usecount;

};

.\_erase：擦除方法。

struct erase\_info {

struct mtd\_info \*mtd;

uint64\_t addr;//擦除开始地址

uint64\_t len;//擦除大小

uint64\_t fail\_addr;

u\_long time;

u\_long retries;

unsigned dev;

unsigned cell;

void (\*callback) (struct erase\_info \*self);//擦除完成回调

u\_long priv;

u\_char state;//擦除完成状态

struct erase\_info \*next;

};

含有callback和state成员内核，说明当前版本mtd\_erase()是异步的，所以设备驱动在擦除完成后需要更改state的状态，成功为MTD\_ERASE\_DONE，失败为MTD\_ERASE\_FAILED，然后调用mtd\_erase\_callback()执行完成回调通知擦除完成。但是高版本内核已经将mtd\_erase()改成同步的，因为mtd->\_erase()基本都使用同步方式实现，所以为了简化使用将这两个成员删掉了，直接通过mtd->\_erase()的返回状态判断擦除的成功与否。

mtd\_partition用于MTD设备的分区定义，分区信息一般在设备树中定义，注册mtd设备时内核会自动解析设备树中的分区信息，对于不支持设备树的低版本内核则是在board文件中定义：

struct mtd\_partition {

const char \*name;//分区名称

uint64\_t size;//分区大小，使用MTDPART\_SIZ\_FULL表示使用全部空间

uint64\_t offset; //分区的起始位置，如果定义为MTDPART\_OFS\_APPEND则从上一个分区的结束位置开始，如果为MTDPART\_OFS\_NXTBLK则为下一个块开始位置，如果为MTDPART\_OFS\_RETAIN则表示预留size大小的空间

uint32\_t mask\_flags;//从mtd\_info的flags中屏蔽的标志，比如设为MTD\_WRITEABLE不是代表此分区可写而是代表此分区只读

};

mtd设备的注册与注销通过如下函数完成：

int mtd\_device\_register(struct mtd\_info \*mtd, const struct mtd\_partition \*parts, int nr\_parts)

.parts/nr\_parts:如果分区信息在设备树中则为NULL和0

int mtd\_device\_unregister(struct mtd\_info \*master)

mtd\_info中的操作方法指针不能直接调用，如果需要执行擦除、读写等操作，需要调用内核提供的如下函数：

int mtd\_erase(struct mtd\_info \*mtd, struct erase\_info \*instr);

int mtd\_read(struct mtd\_info \*mtd, loff\_t from, size\_t len, size\_t \*retlen, u\_char \*buf);

int mtd\_write(struct mtd\_info \*mtd, loff\_t to, size\_t len, size\_t \*retlen, const u\_char \*buf);

# MTD字符设备

MTD字符设备的实现在mtdchar.c中，我们在注册MTD设备的时候其会调用add\_mtd\_device()，会注册mtd的字符设备，即/dev目录下的mtd0等。

mtd-utils是MTD设备的工具包，常用的命令包括flashcp、flash\_eraseall等。

mtdchar\_test.c是仿照flashcp向mtd字符设备写数据的例程，演示字符设备的读写操作以及通过ioctl读取mtd信息以及进行擦除操作。

# MTD块设备

MTD块设备的实现在mtdblock.c中，MTD块设备的注册并没有字符设备注册那么直白，下面来看一下注册过程：

同样是在add\_mtd\_device()中，其会执行mtd\_notifier链表节点结构的add:

list\_for\_each\_entry(not, &mtd\_notifiers, list)

not->add(mtd);

而向mtd\_notifier添加节点的过程如下：

register\_mtd\_blktrans(&mtdblock\_tr);

register\_mtd\_user(&blktrans\_notifier);

list\_add(&new->list, &mtd\_notifiers);

list\_add(&tr->list, &blktrans\_majors);

所以上面的not->add(mtd)中的not就是blktrans\_notifier，其add成员如下：

static void blktrans\_notify\_add(struct mtd\_info \*mtd)

{

list\_for\_each\_entry(tr, &blktrans\_majors, list)

tr->add\_mtd(tr, mtd);

}

tr指向mtdblock\_tr，所以mtd块设备的添加过程在mtdblock\_tr->add\_mtd()中完成，然后就能看到通用块设备gendisk的注册了：

static void mtdblock\_add\_mtd(struct mtd\_blktrans\_ops \*tr, struct mtd\_info \*mtd)

{

add\_mtd\_blktrans\_dev(&dev->mbd);

gd = alloc\_disk(1 << tr->part\_bits);

device\_add\_disk(&new->mtd->dev, gd);

}

# spi-nor驱动

对于spi接口的nor flash，驱动可进一步细化如下：



spi-nor.c是spi-nor驱动的核心，实现的spi\_nor\_scan函数负责扫描spi nor chip以及填充mtd\_info。m25p80.c则负责实现使用spi接口读写nor flash的寄存器和数据的方法，填充struct spi\_nor，然后调用spi\_nor\_scan得到填充完毕的mtd\_info，最后再完成mtd设备的注册。当然对于一些专用的控制器，比如Cadence QSPI，读写nor flash的寄存器和数据的方法有所不同，驱动同样也是完成spi\_nor结构的填充，然后调用spi\_nor\_scan得到填充完毕的mtd\_info，最后再完成mtd设备的注册。

目前大部分的spi-nor flash驱动肯定已经直接支持，设备树中只需添加如下类似节点即可：

flash: m25p80@0 {

#address-cells = <1>;

#size-cells = <1>;

compatible = "spansion,m25p80", "jedec,spi-nor";

reg = <0>;//片选

spi-max-frequency = <40000000>;//spi clk的最大频率

m25p,fast-read; //如果支持"fast read" opcode的flash，添加此属性在读数据时会用"fast read" opcode而不是使用常规的"read" opcode

};

上面的compatible中定义了两个。第一个由芯片制造商和名称组成的特定于某一设备的字符串，可以起到两个作用，一个是通过配置在m25p80.c的m25p\_ids中定义的设备名称从而匹配驱动，另外就是在执行spi\_nor\_scan()时通过spi\_nor\_match\_id()获取到名称一致的struct flash\_info。但是其实使用第二种"jedec,spi-nor"的方式更加合适，它是匹配驱动的通用字符串，不特定于具体的spi nor flash型号，之所以推荐使用这种方式是因为现在spi nor flash都支持通过READ ID opcode获取到ID，spi\_nor\_scan()中执行spi\_nor\_read\_id就是通过ID来获取struct flash\_info。

不同的spi nor flash的主要区别就是扇区大小、扇区个数、擦除大小等，spi-nor.c中的struct flash\_info spi\_nor\_ids定义了很多常用的spi nor flash的信息，如果其中不包含我们所使用的spi nor flash，就需要自己添加相应信息，其实这也是我们唯一所需要做的事情。

spi nor flash的信息定义如下：

struct flash\_info {

char \*name;//设备名称

u8 id[SPI\_NOR\_MAX\_ID\_LEN];// JEDEC ID + EXT ID,前三个字节是JEDEC ID，ID必须正确填写，否则驱动无法获取flash\_info

u8 id\_len;

unsigned sector\_size;//扇区大小

u16 n\_sectors;//扇区个数

u16 page\_size;

u16 addr\_width;

u16 flags;//标志

};

# nor flash驱动

# nand flash驱动

# Flash文件系统

常用的flash文件系统有JFFS2和UBIFS。

# 内核虚拟驱动

mtdram.c

虚拟MTD设备驱动，mtdram有两个参数，总大小total\_size，擦除大小erase\_size，都以KB为单位，下面会创建一个32MB大小，擦除单位为512KB的MTD设备。

insmod mtdram.ko total\_size=$((32\*1024)) erase\_size=512

block2mtd.c

使用块设备虚拟MTD设备，这里的块设备可以是真正的块设备，也可以是使用losetup模拟的块设备。

dd if=/dev/zero of=block.img bs=1M count=32//创建32MB大小的文件

losetup /dev/loop0 block.img//文件虚拟成块设备

insmod block2mtd.ko block2mtd=/dev/loop0,128KiB//参数需要设置块设备，擦除单位选设

nandsim.c

虚拟NAND flash设备，nandsim有一堆参数，默认参数下会创建一个128MB大小的flash设备。