Linux MTD 源代码分析

by Jim Zeus
Version 0.1
2002/4/29

OOB: out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的

If you got any Problem, Suggestion, Advice or Question ,
Please mail to: jimzeus@sina.com.cn
Any correction will be appreciated.

目录

专有名词：

Linux MTD介绍：
设备层和原始设备层的函数调用关系（红色部分需要我们实现）：
NOR型Flash芯片驱动与MTD原始设备
NAND和NOR的比较
源码分析
头文件分析
mtd.h
MTD_CHAR_MAJOR
MTD_BLOCK_MAJOR
MAX_MTD_DEVICES
mtd_info
mtd_info.type的取值
mtd_info.flags的取值
mtd_info.ecctype的取值
erase_info
erase_info.state的取值
mtd_notifier
get_mtd_device
put_mtd_device
partitions.h
mtd_partition
MTDPART_OFS_APPEND
MTDPART_SIZ_FULL
map.h
map_info
mtd_chip_driver
gen_probe.h
chip_probe

cfi.h
cfi_private
cfi_private.cfi_mode的取值
cfi_ident
cfi_ident.P_ID的取值
flashchip.h
flchip
/drivers/mtd/子目录
mtdcore.c
mtd_table
mtd_notifiers
add_mtd_device
del_mtd_device
register_mtd_user
unregister_mtd_user
__get_mtd_device
mtdpart.c
mtd_partitions
mtd_part
PART(x)
add_mtd_partitions
del_mtd_partitions
part_read
part_write
part_readv
part_writev
part_erase
part_lock
part_unlock
part_sync
part_suspend
part_resume
mtdblock.c
notifier
mtdblk_dev
mtdblks
erase_callback
erase_write
write_cached_data
do_cached_write
do_cached_read

mtdblock_open
mtdblock_release
handle_mtdblock_request
leaving
mtdblock_thread
mtdblock_ioctl
mtd_fops
init_mtdblock
cleanup_mtdblock
mtdchar.c
notifier
mtd_lseek
mtd_open
mtd_close
MAX_KMALLOC_SIZE
mtd_read
mtd_write
mtd_erase_callback
mtd_ioctl
mtd_fops
init_mtdchar
cleanup_mtdchar
/drivers/mtd/chips子目录
chipreg.c
chip_drvs_list
register_mtd_chip_driver
unregister_mtd_chip_driver
get_mtd_chip_driver
do_map_probe
cfi_probe.c
cfi_chipdrv
cfi_probe_init
cfi_probe_exit
cfi_probe
cfi_chip_probe
cfi_probe_chip
qry_present
cfi_chip_setup
jedec_probe.c
amd_flash_info
jedec_table

jedec_chipdrv
jedec_probe_init
jedec_probe_exit
jedec_probe
jedec_probe_chip
jedec_probe_chip
cfi_jedec_setup
gen_probe.c
mtd_do_chip_probe
genprobe_ident_chips
genprobe_new_chip
check_cmd_set
cfi_cmdset_unkown
cfi_cmdset_0002.c
cfi_amdstd_chipdrv
cfi_cmdset_0002
cfi_amdstd_setup
cfi_amdstd_erase_onesize
do_erase_oneblock
cfi_amdstd_read
do_read_onechip
cfi_amdstd_write
do_write_oneword
cfi_amdstd_sync
cfi_amdstd_suspend
cfi_amdstd_resume
cfi_amdstd_destroy
cfi_amdstd_erase_varsize
cfi_amdstd_init
cfi_amdstd_exit
cfi_cmdset_0001.c
/drivers/mtd/maps子目录
your-flash.c
WINDOW_ADDR
WINDOW_SIZE
BUSWIDTH
mymtd
your_read8
your_read16
your_read32
your_copy_from

your_write8
your_write16
your_write32
your_copy_to
your_set_vpp
your_partition
NUM_PARTITIONS
your_map
init_yourflash
cleanup_yourflash

专有名词：

1. **MTD**：Memory Technology Device，内存技术设备，
2. **JEDEC**：Joint Electron Device Engineering Council，电子电器设备联合会
3. **CFI**：Common Flash Interface，通用Flash接口，Intel发起的一个Flash的接口标准
4. **OOB**： out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的块有16个字节的extra data，用于纠错或元数据。
5. **ECC**： error correction，某些硬件不仅允许对flash的访问，也有ecc功能，所有flash器件都受位交换现象的困扰。在某些情况下，一个比特位会发生反转或被报告反转了，如果此位真的反转了，就要采用ECC算法。
6. **erasesize**： 一个erase命令可以擦除的最小块的尺寸
7. **buswidth**：MTD设备的接口总线宽度
8. **interleave**：交错数，几块芯片平行连接成一块芯片，使buswidth变大
9. **devicetype**：芯片类型，x8、x16或者x32
10. **NAND**：一种Flash技术，参看NAND和NOR的比较
11. **NOR**：一种Flash技术，参看NAND和NOR的比较

重点是mtd的定义,以及为什么要有mtd
mtd设备的分层

Linux MTD介绍：

MTD(memory technology device内存技术设备)是用于访问memory设备（ROM、flash）的Linux的子系统。MTD的主要目的是为了使新的memory设备的驱动更加简单，为此它在硬件和上层之间提供了一个抽象的接口。MTD的所有源代码在/drivers/mtd子目录下。我将CFI接口的MTD设备分为四层（从设备节点直到底层硬件驱动），这四层从上到下依次是：设备节点、MTD设备层、MTD原始设备层和硬件驱动层。

根文件系统
文件系统
  字符设备节点
MTD字符设备
MTD块设备
MTD原始设备
FLASH硬件驱动
  块设备节点

一、**Flash**硬件驱动层：硬件驱动层负责在init时驱动Flash硬件，Linux MTD设备的NOR Flash芯片驱动遵循CFI接口标准，其驱动程序位于drivers/mtd/chips子目录下。NAND型Flash的驱动程序则位于/drivers/mtd/nand子目录下
二、**MTD**原始设备：原始设备层有两部分组成，一部分是MTD原始设备的通用代码，另一部分是各个特定的Flash的数据，例如分区。
    用于描述MTD原始设备的数据结构是mtd_info，这其中定义了大量的关于MTD的数据和操作函数。

mtd_table（mtdcore.c）则是所有MTD原始设备的列表，mtd_part（mtd_part.c）是用于表示MTD原始设备分区的结构，其中包含了mtd_info，因为每一个分区都是被看成一个MTD原始设备加在mtd_table中的，mtd_part.mtd_info中的大部分数据都从该分区的主分区mtd_part->master中获得。

在drivers/mtd/maps/子目录下存放的是特定的flash的数据，每一个文件都描述了一块板子上的flash。其中调用add_mtd_device()、del_mtd_device()建立/删除mtd_info结构并将其加入/删除mtd_table（或者调用add_mtd_partition()、del_mtd_partition()（mtdpart.c）建立/删除mtd_part结构并将mtd_part.mtd_info加入/删除mtd_table 中）。

三、**MTD**设备层：基于MTD原始设备，linux系统可以定义出MTD的块设备（主设备号31）和字符设备（设备号90）。MTD字符设备的定义在mtdchar.c中实现，通过注册一系列file operation函数（lseek、open、close、read、write）。MTD块设备则是定义了一个描述MTD块设备的结构mtdblk_dev，并声明了一个名为mtdblks的指针数组，这数组中的每一个mtdblk_dev和mtd_table中的每一个mtd_info一一对应。

四、设备节点：通过mknod在/dev子目录下建立MTD字符设备节点（主设备号为90）和MTD块设备节点（主设备号为31），通过访问此设备节点即可访问MTD字符设备和块设备。

五、根文件系统：在Bootloader中将JFFS（或JFFS2）的文件系统映像jffs.image（或jffs2.img）烧到flash的某一个分区中，在/arch/arm/mach-your/arch.c文件的your_fixup函数中将该分区作为根文件系统挂载。

六、文件系统：内核启动后，通过mount 命令可以将flash中的其余分区作为文件系统挂载到mountpoint上。

设备层和原始设备层的函数调用关系（红色部分需要我们实现）：

一个MTD原始设备可以通过mtd_part分割成数个MTD原始设备注册进mtd_table，mtd_table中的每个MTD原始设备都可以被注册成一个MTD设备，其中字符设备的主设备号为90，次设备号为0、2、4、6…（奇数次设备号为只读设备），块设备的主设备号为31，次设备号为0、1、2、3…

mtd_notifier　　　　　　　　　　mtd_notifier

字符设备　　mtd_fops　　　　块设备　　　　mtd_fops

（mtdchar.c）　　　　　　　　（mtdblock.c）　　　　mtdblks

设备层

register_mtd_user()

get_mtd_device()

unregister_mtd_user()

put_mtd_device()

erase_info

OOB: out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的

mtd_notifiers
mtd_table
mtd_info
mtd_part
（mtdcore.c）
（mtdpart.c）

Your Flash

（your-flash.c）

add_mtd_partitions()
del_mtd_partitions()
add_mtd_device()
del_mtd_device()
mtd_partition

原始设备层

NOR型Flash芯片驱动与MTD原始设备

所有的NOR型Flash的驱动（探测probe）程序都放在drivers/mtd/chips下，一个MTD原始设备可以由一块或者数块相同的Flash芯片组成。假设由4块devicetype为x8的Flash，每块大小为8M，interleave为2，起始地址为0x01000000，地址相连，则构成一个MTD原始设备（0x01000000-0x03000000），其中两块interleave成一个chip，其地址从0x01000000到0x02000000，另两块interleave成一个chip，其地址从0x02000000到0x03000000。

请注意，所有组成一个MTD原始设备的Flash芯片必须是同类型的（无论是interleave还是地址相连），在描述MTD原始设备的数据结构中也只是采用了同一个结构来描述组成它的Flash芯片。

Chip#1　Chip#2

Chip#3　Chip#4
0x03000000

0x02000000

0x01000000

每个MTD原始设备都有一个**mtd_info**结构，其中的priv指针指向一个**map_info**结构，map_info结构中的fldrv_priv指向一个**cfi_private**结构，cfi_private结构的cfiq指针指向一个**cfi_ident**结构，chips指针指向一个**flchip**结构的数组。其中mtd_info、map_info和cfi_private结构用于描述MTD原始设备；因为组成MTD原始设备的NOR型Flash相同，cfi_ident结构用于描述Flash芯片的信息；而flchip结构用于描述每个Flash芯片的专有信息（比如说起始地址）

NAND和NOR的比较

NOR和NAND是现在市场上两种主要的非易失闪存技术。Intel于1988年首先开发出NOR flash技术，彻底改变了原先由EPROM和EEPROM一统天下的局面。紧接着，1989年，东芝公司发表了NAND flash结构，强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。但是经过了十多年之后，仍然有相当多的硬件工程师分不清NOR和NAND闪存。

相“flash存储器”经常可以与相“NOR存储器”互换使用。许多业内人士也搞不清楚NAND闪存技术相对于NOR技术的优越之处，因为大多数情况下闪存只是用来存储少量的代码，这时NOR闪存更适合一些。而NAND则是高数据存储密度的理想解决方案。

NOR的特点是芯片内执行(XIP, eXecute In Place)，这样应用程序可以直接在flash闪存内运行，不必再把代码读到系统RAM中。NOR的传输效率很高，在1～4MB的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。

NAND结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用NAND的困难在于flash的管理和需要特殊的系统接口。

性能比较

flash闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何flash器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除。NAND器件执行擦除操作是十分简单的，而NOR则要求在进行擦除前先要将目标块内所有的位都写为0。

由于擦除NOR器件时是以64～128KB的块进行的，执行一个写入/擦除操作的时间为5s，与此相反，擦除NAND器件是以8～32KB的块进行的，执行相同的操作最多只需要4ms。

执行擦除时块尺寸的不同进一步拉大了NOR和NADN之间的性能差距，统计表明，对于给定的一套写入操作(尤其是更新小文件时)，更多的擦除操作必须在基于NOR的单元中进行。这样，当选择存储解决方案时，设计师必须权衡以下的各项因素。

● NOR的读速度比NAND稍快一些。

● NAND的写入速度比NOR快很多。

● NAND的4ms擦除速度远比NOR的5s快。

● 大多数写入操作需要先进行擦除操作。

● NAND的擦除单元更小，相应的擦除电路更少。

接口差别

NOR flash带有SRAM接口，有足够的地址引脚来寻址，可以很容易地存取其内部的每一个字节。

NAND器件使用复杂的I/O口来串行地存取数据，各个产品或厂商的方法可能各不相同。8个引脚用来传送控制、地址和数据信息。

NAND读和写操作采用512字节的块，这一点有点像硬盘管理此类操作，很自然地，基于NAND的存储器就可以取代硬盘或其他块设备。

容量和成本

NAND flash的单元尺寸几乎是NOR器件的一半，由于生产过程更为简单，NAND结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。

NOR flash占据了容量为1～16MB闪存市场的大部分，而NAND flash只是用在8～128MB的产品当中，这也说明NOR主要应用在代码存储介质中，NAND适合于数据存储，NAND在CompactFlash、Secure Digital、PC Cards和MMC存储卡市场上所占份额最大。

可靠性和耐用性

采用flahs介质时一个需要重点考虑的问题是可靠性。对于需要扩展MTBF的系统来说，Flash是非常合适的存储方案。可以从寿命(耐用性)、位交换和坏块处理三个方面来比较NOR和NAND的可靠性。

寿命(耐用性)

在NAND闪存中每个块的最大擦写次数是一百万次，而NOR的擦写次数是十万次。NAND存储器除了具有10比1的块擦除周期优势，典型的NAND块尺寸要比NOR器件小8倍，每个NAND存储器块在给定的时间内的删除次数要少一些。

位交换

所有flash器件都受位交换现象的困扰。在某些情况下(很少见，NAND发生的次数要比NOR多)，一个比特位会发生反转或被报告反转了。

一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。

当然，如果这个位真的改变了，就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于NAND闪存，NAND的供应商建议使用NAND闪存的时候，同时使用EDC/ECC算法。

这个问题对于用NAND存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用EDC/ECC系统以确保可靠性。

坏块处理

NAND器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。

NAND器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

易于使用

可以非常直接地使用基于NOR的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。

由于需要I/O接口，NAND要复杂得多。各种NAND器件的存取方法因厂家而异。

在使用NAND器件时，必须先写入驱动程序，才能继续执行其他操作。向NAND器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在NAND器件上自始至终都必须进行虚拟映射。

软件支持

当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。

在NOR器件上运行代码不需要任何的软件支持，在NAND器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND和NOR器件在进行写入和擦除操作时都需要MTD。

使用NOR器件时所需要的MTD要相对少一些，许多厂商都提供用于NOR器件的更高级软件，这其中包括M-System的TrueFFS驱动，该驱动被Wind River System、Microsoft、QNX Software System、Symbian和Intel等厂商所采用。

驱动还用于对DiskOnChip产品进行仿真和NAND闪存的管理，包括纠错、坏块处理和损耗平衡。

——From M-system公司Arie TAL

源码分析

下面具体介绍每个文件的内容和作用：

头文件分析

mtd.h

MTD_CHAR_MAJOR

#define MTD_CHAR_MAJOR 90　　MTD字符设备的主设备号

MTD_BLOCK_MAJOR

#define MTD_BLOCK_MAJOR 31　　MTD块设备的主设备号

MAX_MTD_DEVICES

#define MAX_MTD_DEVICES 16　　最大MTD原始设备数

OOB: out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的

mtd_info

表示MTD原始设备的结构，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，在系统中就一共有6个mtd_info结构，这些mtd_info的指针被存放在名为mtd_table的数组里。

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

    /* "Major" erase size for the device. Na飑e users may take this
     * to be the only erase size available, or may use the more detailed
     * information below if they desire
     */
    u_int32_t erasesize; “ 主要的 ” erasesize（同一个mtd设备可能有数种不同的erasesize）

    u_int32_t oobblock;  // Size of OOB blocks (e.g. 512)          oob块大小
    u_int32_t oobsize;   // Amount of OOB data per block (e.g. 16)          oob特别数据大小
    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;

    /* Data for variable erase regions. If numeraseregions is zero,
     * it means that the whole device has erasesize as given above.
     */不同的erasesize的区域
    int numeraseregions; 不同erasesize的区域的数目（通常是1）
    struct mtd_erase_region_info *eraseregions;

    /* This really shouldn't be here. It can go away in 2.5 */
    u_int32_t bank_size;

    struct module *module;
    int (*erase) (struct mtd_info *mtd, struct erase_info *instr);
            此routine用于将一个erase_info加入erase queue
    /* 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
}
```

mtd_info.type的取值

```
#define MTD_ABSENT        0
#define MTD_RAM                 1
#define MTD_ROM                 2
#define MTD_NORFLASH           3
#define MTD_NANDFLASH        4
#define MTD_PEROM          5
#define MTD_OTHER         14
#define MTD_UNKNOWN          15
```

mtd_info.flags的取值

```
#define MTD_CLEAR_BITS        1       // Bits can be cleared (flash)
#define MTD_SET_BITS            2       // Bits can be set
```

```
#define MTD_ERASEABLE         4      // Has an erase function
#define MTD_WRITEB_WRITEABLE 8       // Direct IO is possible
#define MTD_VOLATILE          16     // Set for RAMs
#define MTD_XIP               32   // eXecute-In-Place possible
#define MTD_OOB                   64   // Out-of-band data (NAND flash)
#define MTD_ECC               128  // Device capable of automatic ECC

// Some common devices / combinations of capabilities
#define MTD_CAP_ROM           0
#define MTD_CAP_RAM            (MTD_CLEAR_BITS|MTD_SET_BITS|MTD_WRITEB_WRITEABLE)
#define MTD_CAP_NORFLASH       (MTD_CLEAR_BITS|MTD_ERASEABLE)
#define MTD_CAP_NANDFLASH      (MTD_CLEAR_BITS|MTD_ERASEABLE|MTD_OOB)
#define MTD_WRITEABLE          (MTD_CLEAR_BITS|MTD_SET_BITS)
```

mtd_info.ecctype的取值

```
#define MTD_ECC_NONE          0    // No automatic ECC available
#define MTD_ECC_RS_DiskOnChip   1    // Automatic ECC on DiskOnChip
#define MTD_ECC_SW        2    // SW ECC for Toshiba & Samsung devices
```

erase_info

表示erase动作的结构，由设备层调用mtd_info->erase函数传递给原始设备层

```
struct erase_info {
      struct mtd_info *mtd;    被操作的MTD原始设备
      u_int32_t addr;          被操作的地址（byte）
      u_int32_t len;           长度
      u_long time;
      u_long retries;
      u_int dev;
      u_int cell;
      void (*callback) (struct erase_info *self);
                              callback函数指针，当erase结束后该函数被调用
      u_long priv;             user模块提供的private数据
      u_char state;            当前状态
      struct erase_info *next;  erase队列中的下一个erase_info
};
```

erase_info.state的取值

```
#define MTD_ERASE_PENDING         0x01
#define MTD_ERASING                       0x02
#define MTD_ERASE_SUSPEND             0x04
#define MTD_ERASE_DONE           0x08
```

```
#define MTD_ERASE_FAILED          0x10
```

驱动模块在erase要求进入erase 队列时将state设置为MTD_ERASE_PENDING，当callback被调用时设置为MTD_ERASE_DONE或MTD_ERASE_FAILED。

mtd_notifier

MTD通知器（这个名字很古怪，但我找不到更好的词来翻译），加入/删除MTD设备和原始设备时调用的函数，在设备层，当MTD字符设备或块设备注册时，如果定义了CONFIG_DEVFS_FS，则会将一个mtd_notifier加入MTD原始设备层的mtd_notifiers链表，其中的函数会在两种情况下被调用,一是加入/删除新的MTD字符/块设备时，此时调用该MTD字符/块设备的notifier对下层所有的MTD原始设备操作一遍，二是加入/删除新的MTD原始设备时，此时调用所有的notifier对该原始设备执行一遍。

```
                        struct mtd_notifier {
                        void (*add)(struct mtd_info *mtd);      加入时调用
                        void (*remove)(struct mtd_info *mtd); 删除时调用
    struct mtd_notifier *next;      指向mtd_notifiers队列中的下一个mtd_notifier
};
```

get_mtd_device

格式：

    static inline struct mtd_info *get_mtd_device(struct mtd_info *mtd, int num)

注释：

无

功能：

获得一个MTD设备

说明：

调用__get_mtd_device()

参数：

见__get_mtd_device()

返回：

要求的MTD原始设备

调用：

__get_mtd_device()

被调用：

设备层函数，在设备层的open函数中被调用以获得一个MTD原始设备

源代码：

```
{
    struct mtd_info *ret;

    ret = __get_mtd_device(mtd, num);

    if (ret && ret->module && !try_inc_mod_count(ret->module))
        return NULL;
```

```
	return ret;
}
```

put_mtd_device

格式：

```
	static inline void put_mtd_device(struct mtd_info *mtd)
```

注释：

	无

功能：

	归还（put）一个使用完的MTD设备

说明：

	无

参数：

	该MTD设备

返回：

	无

调用：

	无

被调用：

	设备层函数，通常在设备层的release函数中调用。

源代码：

```
{
	if (mtd->module)
		__MOD_DEC_USE_COUNT(mtd->module);
}
```

partitions.h

此文件中的数据用来从/maps子目录下的文件向原始设备层传递分区信息

mtd_partition

```
/*
 * Partition definition structure:
 *
 * An array of struct partition is passed along with a MTD object to
 * add_mtd_partitions() to create them.
 *
 * For each partition, these fields are available:
 * name: string that will be used to label the partition's MTD device.
 * size: the partition size; if defined as MTDPART_SIZ_FULL, the partition
 *   will extend to the end of the master MTD device.
 * offset: absolute starting position within the master MTD device; if
 *   defined as MTDPART_OFS_APPEND, the partition will start where the
 *   previous one ended.
 * mask_flags: contains flags that have to be masked (removed) from the
 *   master MTD flag set for the corresponding MTD partition.
 *   For example, to force a read-only partition, simply adding
 *   MTD_WRITEABLE to the mask_flags will do the trick.
 *
 * Note: writeable partitions require their size and offset be
 * erasesize aligned.
 */
struct mtd_partition {    描述mtd设备分区的结构，在MTD原始设备层调用add_mtd_partions
                          时传递分区信息用
      char *name;           /* identifier string */
      u_int32_t size;        /* partition size */
      u_int32_t offset;          /* offset within the master MTD space */
      u_int32_t mask_flags;    /* master MTD flags to mask out for this partition */将被掩去的标
       志位，（例如如果在分区时置mask_flags为WRITEABLE，则此分区为只读分区）
};
```

MTDPART_OFS_APPEND

```
#define MTDPART_OFS_APPEND     (-1)       如果mtd_partition.offset取此值，表示此分区
                                        的offset紧接上一个分区的结束
```

MTDPART_SIZ_FULL

```
#define MTDPART_SIZ_FULL     (0)           如果mtd_partion.size取此值，表示此分区一
                                        直到MTD原始设备的最后结束
```

map.h

map_info

```
struct map_info {
      char *name;
      unsigned long size;
```

```
    int buswidth; /* in octets */
    __u8 (*read8)(struct map_info *, unsigned long);          Flash的读写函数，（NOR型Flash
    __u16 (*read16)(struct map_info *, unsigned long);  可以像内存一样进行读写），read8、
    __u32 (*read32)(struct map_info *, unsigned long);  read16和read32分别用于buswidth
                                                       为1、2、4字节的情况（write同理）
    /* If it returned a 'long' I'd call it readl.
     * It doesn't.
     * I won't.
     * dwmw2 */

    void (*copy_from)(struct map_info *, void *, unsigned long, ssize_t);
    void (*write8)(struct map_info *, __u8, unsigned long);
    void (*write16)(struct map_info *, __u16, unsigned long);
    void (*write32)(struct map_info *, __u32, unsigned long);
    void (*copy_to)(struct map_info *, unsigned long, const void *, ssize_t);

    void (*set_vpp)(struct map_info *, int);
    /* We put these two here rather than a single void *map_priv,
       because we want mappers to be able to have quickly-accessible
       cache for the 'currently-mapped page' without the _extra_
       redirection that would be necessary. If you need more than
       two longs, turn the second into a pointer. dwmw2 */
    unsigned long map_priv_1;
    unsigned long map_priv_2;
    void *fldrv_priv;          指向cfi_private结构
    struct mtd_chip_driver *fldrv;
};
```

mtd_chip_driver

```
struct mtd_chip_driver {
    struct mtd_info *(*probe)(struct map_info *map);     该芯片的probe程序
    void (*destroy)(struct mtd_info *);
    struct module *module;        模块
    char *name;          芯片名
    struct list_head list;
};
```

MTD芯片驱动器的结构

gen_probe.h

chip_probe

```
struct chip_probe {
	char *name;
	int (*probe_chip)(struct map_info *map, __u32 base,
			struct flchip *chips, struct cfi_private *cfi);

};
```

通用探测程序的参数结构，此结构类型将作为参数传进mtd_do_chip_probe()中，mtd_do_chip_probe()将间接调用其中的probe_chip函数

cfi.h

cfi_private

CFI私有信息

```
struct cfi_private {
	__u16 cmdset;
	void *cmdset_priv;
	int interleave;		芯片交错数，interleave片芯片交织连接成一个芯片
	int device_type;	每一片Flash的类型（1：8bits，2：16bits，4：32bits）
	int cfi_mode;		/* Are we a JEDEC device pretending to be CFI? */	CFI的模式
	int addr_unlock1;	命令地址，所谓“ unlock ”是指无论Flash处于读模式或写模式，都
	int addr_unlock2;		可向该地址写命令
	int fast_prog;		允许Fast Program模式
	struct mtd_info *(*cmdset_setup)(struct map_info *);
	struct cfi_ident *cfiq; /* For now only one. We insist that all devs	每个芯片的信息
				must be of the same type. */			（所有芯片是同类型的）
	int mfr, id;
	int numchips;		Flash芯片个数（N interleave算一块芯片）
	unsigned long chipshift; /* Because they're of the same type */
						每块芯片大小*interleave=2的chipshift次方
	const char *im_name;	/* inter_module name for cmdset_setup */
	struct flchip chips[0];  /* per-chip data structure for each chip */每一块芯片的信息
};
```

cfi_private.cfi_mode的取值

```
#define CFI_MODE_CFI	0		真的是CFI
```

OOB: out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的

```
#define CFI_MODE_JEDEC  1          JEDEC仿真CFI
```

cfi_ident

CFI接口芯片的查询结构，从芯片起始地址+0x10开始的0x2C个字节

```
/* Basic Query Structure */
struct cfi_ident {
  __u8  qry[3];          " QRY "      0x10,0x11,0x12
  __u16 P_ID;            制造商ID     Primary ID          0x13,0x14
  __u16 P_ADR;                        Primary Address     0x15,0x16
  __u16 A_ID;                         Alternate ID        0x17,0x18
  __u16 A_ADR;                        Alternate Address   0x19,0x1A
  __u8  VccMin;          逻辑供电Vcc最小电压      0x1B
  __u8  VccMax;          逻辑供电Vcc最大电压      0x1C
  __u8  VppMin;          编程/擦除供电Vpp最小电压     0x1D
  __u8  VppMax;          编程/擦除供电Vpp最大电压     0x1E
  __u8  WordWriteTimeoutTyp;      典型单字节/字写周期定时时间        0x1F
  __u8  BufWriteTimeoutTyp;                                0x20
  __u8  BlockEraseTimeoutTyp; 典型单块擦除定时时间       0x21
  __u8  ChipEraseTimeoutTyp;  典型整片擦除定时时间       0x22
  __u8  WordWriteTimeoutMax; 单字节/字写周期最大定时时间        0x23
  __u8  BufWriteTimeoutMax;                                0x24
  __u8  BlockEraseTimeoutMax; 单块擦除最大定时时间      0x25
  __u8  ChipEraseTimeoutMax; 整片擦除最大定时时间      0x26
  __u8  DevSize;                器件体积                   0x27
  __u16 InterfaceDesc;          Flash器件接口识别码ID     0x28,0x29
  __u16 MaxBufWriteSize;                                   0x2A,0x2B
  __u8  NumEraseRegions;      器件可擦除块区域个数       0x2C
  __u32 EraseRegionInfo[0]; /* Not host ordered */  擦除块区域信息：       0x2D—0x3C
} __attribute__((packed));`        bit15-0=y该擦除块区域内含同样体积擦除块的个数=y+1
                                  bit31-16=z该区域每个擦除块体积=z x 256字节
```

cfi_ident.P_ID的取值

```
#define P_ID_NONE 0
#define P_ID_INTEL_EXT 1
#define P_ID_AMD_STD 2
#define P_ID_INTEL_STD 3
#define P_ID_AMD_EXT 4
```

```
#define P_ID_MITSUBISHI_STD 256
#define P_ID_MITSUBISHI_EXT 257
#define P_ID_RESERVED 65535
```

flashchip.h

flchip

```
struct flchip {
       unsigned long start; /* Offset within the map */
       //     unsigned long len;          假设所有的Flash chip都是相同的，所以没有定义len
       /* We omit len for now, because when we group them together
          we insist that they're all of the same size, and the chip size
          is held in the next level up. If we get more versatile later,
          it'll make it a damn sight harder to find which chip we want from
          a given offset, and we'll want to add the per-chip length field
          back in.
       */
       flstate_t state;
       flstate_t oldstate;
       spinlock_t *mutex;
       spinlock_t _spinlock; /* We do it like this because sometimes they'll be shared. */
       wait_queue_head_t wq; /* Wait on here when we're waiting for the chip
                   to be ready */
       int word_write_time;
       int buffer_write_time;
       int erase_time;
};
```

/drivers/mtd/子目录

mtdcore.c

本文件主要实现了MTD原始设备层的数据结构**mtd_table**和对其的操作函数，主要包括**add_mtd_device ()**（加入MTD原始设备）、**del_mtd_device()**（删除MTD原始设备）

mtd_table

static struct mtd_info *mtd_table[MAX_MTD_DEVICES];

标示MTD设备的数组，最多可以有MAX_MTD_DEVICES个设备（每一个MTD分区也算一个MTD设备）

mtd_notifiers

static struct mtd_notifier *mtd_notifiers = NULL;

标示mtd_notifier队列的指针

add_mtd_device

格式：

int add_mtd_device(struct mtd_info *mtd)

注释**:**

```
/**
 *    add_mtd_device - register an MTD device
 *    @mtd: pointer to new MTD device info structure
 *
 *    Add a device to the list of MTD devices present in the system, and
 *    notify each currently active MTD 'user' of its arrival. Returns
 *    zero on success or 1 on failure, which currently will only happen
 *    if the number of present devices exceeds MAX_MTD_DEVICES (i.e. 16)
 */
```

功能：

加入一个MTD原始设备。

参数：

mtd：被加入的原始设备

返回：

成功：返回0

mtd_table已满：返回1

说明：

1. 加入此原始设备的mtd_info到mtd_table中，
2. notify所有的MTD设备一个mtd被加入（调用所有的notifier->add(mtd)）。

调用：

notifers->add(mtd)

被调用：

Flash相关文件（your-flash.c）初始化时（maps子目录下的文件中的init_xxx函数）

源代码：

{

```
    int i;

    down(&mtd_table_mutex);

    for (i=0; i< MAX_MTD_DEVICES; i++)
        if (!mtd_table[i])                    //找到mtd_table中的空项
        {
            struct mtd_notifier *not=mtd_notifiers;

            mtd_table[i] = mtd;              //将其指向新加入的MTD原始设备
            mtd->index = i;
            DEBUG(0, "mtd: Giving out device %d to %s\n",i, mtd->name);
            while (not)                       //如果有notifier，调用所有的notifier对新加
            {                                 //入的MTD原始设备操作一遍
                (*(not->add))(mtd);
                not = not->next;
            }
            up(&mtd_table_mutex);
            MOD_INC_USE_COUNT;
            return 0;
        }

    up(&mtd_table_mutex);
    return 1;
}
```

del_mtd_device

格式：

    int del_mtd_device (struct mtd_info *mtd)

注释：

```
/**
 *  del_mtd_device - unregister an MTD device
 *  @mtd: pointer to MTD device info structure
 *
 *  Remove a device from the list of MTD devices present in the system,
 *  and notify each currently active MTD 'user' of its departure.
 *  Returns zero on success or 1 on failure, which currently will happen
 *  if the requested device does not appear to be present in the list.
 */
```

功能：

删除一个mtd设备。

说明：

从mtd_table中删除此设备的mtd_info，通知所有的MTD设备（调用所有的notifier.remove（mtd））

参数：

　　mtd：描述此设备的mtd_info

返回：

　　成功：返回0

　　此设备不在mtd_table中：返回1

调用：

　　notifiers->remove(mtd)

被调用：

　　Flash相关文件（your-flash.c）清除设备时（drivers/mtd/maps子目录下的文件中的cleanupt_xxx函数）

源代码：

```
{
	struct mtd_notifier *not=mtd_notifiers;
	int i;

	down(&mtd_table_mutex);

	for (i=0; i < MAX_MTD_DEVICES; i++)
	{
		if (mtd_table[i] == mtd)
		{
			while (not)				//如果有notifier，调用所有的notifier对
			{					//被删除的MTD原始设备操作一遍
				(*(not->remove))(mtd);
				not = not->next;
			}
			mtd_table[i] = NULL;
			up (&mtd_table_mutex);
			MOD_DEC_USE_COUNT;
			return 0;
		}
	}

	up(&mtd_table_mutex);
	return 1;
}
```

register_mtd_user

格式：

　　void register_mtd_user (struct mtd_notifier *new)

注释：

```
/**
 *	register_mtd_user - register a 'user' of MTD devices.
 *   @new: pointer to notifier info structure
 *
 *   Registers a pair of callbacks function to be called upon addition
 *   or removal of MTD devices. Causes the 'add' callback to be immediately
 *   invoked for each MTD device currently present in the system.
 */
```

功能：

加入一个MTD原始设备的使用者（即MTD设备，MTD块设备或字符设备）。

说明：

1. 将新的mtd通知器new加入notifer队列中,
2. 增加模块的use count,
3. 对所有已有的MTD原始设备调用new->add()。

参数：

new：新MTD设备的mtd_notifier

返回：

无

调用：

new->add()

被调用：

（如果定义了CONFIG_DEVFS）MTD设备级的初始化（mtdblock.c、mtdchar.c、ftl.c中的init_xxx）

源代码：

```
{
	int i;

	down(&mtd_table_mutex);

	new->next = mtd_notifiers;
	mtd_notifiers = new;

	MOD_INC_USE_COUNT;

	for (i=0; i< MAX_MTD_DEVICES; i++)	//对所有的MTD原始设备操作一遍
		if (mtd_table[i])
			new->add(mtd_table[i]);

	up(&mtd_table_mutex);
}
```

unregister_mtd_user

格式：

```
	int unregister_mtd_user (struct mtd_notifier *old)
```

注释：

```
/**
 *   unregister_mtd_user - unregister a 'user' of MTD devices.
 *   @new: pointer to notifier info structure
 *
 *   Removes a callback function pair from the list of 'users' to be
 *   notified upon addition or removal of MTD devices. Causes the
 *   'remove' callback to be immediately invoked for each MTD device
 *   currently present in the system.
 */
```

功能：

删除一个MTD设备

说明：

1. 将被删除MTD设备的通知器old从notifer队列中删除
2. 减模块的use count,
3. 对所有已有的MTD原始设备调用old->remove()。

参数：

old：被删除MTD设备的mtd_notifier

返回：

成功：返回0

notifiers队列中无此设备的notifier：返回1

调用：

old->remove()

被调用：

（如果定义了CONFIG_DEVFS）MTD设备级的清除（mtdblock.c、mtdchar.c、ftl.c中的cleanup_xxx）

源代码：

```
{
	struct mtd_notifier **prev = &mtd_notifiers;
	struct mtd_notifier *cur;
	int i;

	down(&mtd_table_mutex);

	while ((cur = *prev)) {
		if (cur == old) {
			*prev = cur->next;

			MOD_DEC_USE_COUNT;

			for (i=0; i< MAX_MTD_DEVICES; i++)	//对所有的MTD原始设备调用一遍
				if (mtd_table[i])			//remove函数
					old->remove(mtd_table[i]);

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

```
        prev = &cur->next;
    }
    up(&mtd_table_mutex);
    return 1;
}
```

__get_mtd_device

格式：

```
    struct mtd_info *__get_mtd_device(struct mtd_info *mtd, int num)
```

注释：

```
/**
 *    __get_mtd_device - obtain a validated handle for an MTD device
 *    @mtd: last known address of the required MTD device
 *    @num: internal device number of the required MTD device
 *
 *    Given a number and NULL address, return the num'th entry in the device
 *    table, if any.    Given an address and num == -1, search the device table
 *    for a device with that address and return if it's still present. Given
 *    both, return the num'th driver only if its address matches. Return NULL
 *    if not. get_mtd_device() increases the use count, but
 *    __get_mtd_device() doesn't - you should generally use get_mtd_device().
 */
```

功能：

返回指定的MTD原始设备的handler

说明：

num==-1&mtd!=NULL：返回与mtd地址相同的设备

0<num<MAX_MTD_DEVICES&mtd==NULL：返回mtd_table中的第num个MTD设备

0<num<MAX_MTD_DEVICES&mtd!=NULL：如果mtd_table中第num个设备为mtd,
返回mtd；否则返回空。

其余情况：返回NULL。

参数：

mtd：指定MTD原始设备的指针

num：见说明

返回：

见说明

调用：

无

被调用：

get_mtd_device()

源代码：

```
{
    struct mtd_info *ret = NULL;
    int i;
```

```
	down(&mtd_table_mutex);

	if (num == -1) {
		for (i=0; i< MAX_MTD_DEVICES; i++)
			if (mtd_table[i] == mtd)
				ret = mtd_table[i];
	} else if (num < MAX_MTD_DEVICES) {
		ret = mtd_table[num];
		if (mtd && mtd != ret)
			ret = NULL;
	}

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

mtdpart.c

MTD原始设备层分区的实现，mtd_part结构是用于描述分区的，由**mtd_partitons**将mtd_part中的list成员链成一个链表。每个mtd_part结构中的mtd_info结构用于描述本分区，被加入mtd_table中，其中大部分成员由其主分区mtd_part->master决定，各种函数也指向主分区的相应函数。而主分区（其大小涵盖所有分区）则不作为一个MTD原始设备加入mtd_table。

本文件向外提供的**add_mtd_partitions()**和**del_mtd_partitions()**根据参数map_info结构将所有的分区加入mtd_table

mtdpart.c中还实现了part_read、part_write等函数，这些函数注册在每个分区中，指向主分区的read、write函数，之所以这样做而不直接将主分区的read、write函数连接到每个分区中的原因是因为函数中的参数mtd_info会被调用者置为函数所属的mtd_info，即mtd->read(mtd…)，而参数mtd_info其实应该指向主分区。

mtd_partitions

```
/* Our partition linked list */
static LIST_HEAD(mtd_partitions);			MTD原始设备分区的链表
```

mtd_part

描述分区的结构

```
/* Our partition node structure */
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;
};
```

PART(x)

```
/*
 * Given a pointer to the MTD object in the mtd_part structure, we can retrieve
 * the pointer to that structure with this macro.
 */
#define PART(x)  ((struct mtd_part *)(x))        将mtd_info结构转换为mtd_part结构
```

add_mtd_partitions

格式：

```
    int add_mtd_partitions(struct mtd_info *master, struct mtd_partition *parts, int nbparts)
```

注释：

```
/*
 * This function, given a master MTD object and a partition table, creates
 * and registers slave MTD objects which are bound to the master according to
 * the partition definitions.
 * (Q: should we register the master MTD object as well?)
 */
```

功能：

    将MTD原始设备分区

说明：

```
    打印信息（型如“ Creating 3 MTD partitions on "Your Flash map ”），
    对每一个新建分区{
     建立一个新的mtd_part结构，并将其加入mtd_partitions中，
     根据master设置mtd_part.mtd_info的成员，
        打印信息（型如“ 0x00020000-0x00800000 : "Your Kernel" ”），
        进行安全性检查，
        根据master->numeraseregions处理多erasesize区域的事务，
        调用add_mtd_device将此分区作为MTD设备加入mtd_table
    }
```

参数：

    master：指定的MTD设备
    parts：关于分区信息的指针
    nbparts：分区的数目

返回：

    成功：返回0
    分配mtd_part时内存不足：返回-ENOMEM

调用：

    add_mtd_device()
    del_mtd_partitions()

被调用：

    flash相关的初始化函数中（maps子目录下的文件的函数init_xxx中）

源代码：

```
{
    struct mtd_part *slave;
    u_int32_t cur_offset = 0;
    int i;

    printk (KERN_NOTICE "Creating %d MTD partitions on \"%s\":\n", nbparts, master->name);

    for (i = 0; i < nbparts; i++) {

        /* allocate the partition structure */
        slave = kmalloc (sizeof(*slave), GFP_KERNEL);
        if (!slave) {
            printk ("memory allocation error while creating partitions for \"%s\"\n",
                master->name);
            del_mtd_partitions(master);
            return -ENOMEM;
        }
        memset(slave, 0, sizeof(*slave));
        list_add(&slave->list, &mtd_partitions);

        /* set up the MTD object for this partition */
        slave->mtd.type = master->type;
        slave->mtd.flags = master->flags & ~parts[i].mask_flags;
        slave->mtd.size = parts[i].size;
        slave->mtd.oobblock = master->oobblock;
        slave->mtd.oobsize = master->oobsize;
        slave->mtd.ecctype = master->ecctype;
        slave->mtd.eccsize = master->eccsize;

        slave->mtd.name = parts[i].name;
        slave->mtd.bank_size = master->bank_size;

        slave->mtd.module = master->module;

        slave->mtd.read = part_read;
        slave->mtd.write = part_write;
        if (master->sync)
            slave->mtd.sync = part_sync;
        if (!i && master->suspend && master->resume) {
                slave->mtd.suspend = part_suspend;
                slave->mtd.resume = part_resume;
        }

        if (master->writev)
            slave->mtd.writev = part_writev;
        if (master->readv)
            slave->mtd.readv = part_readv;
        if (master->lock)
            slave->mtd.lock = part_lock;
        if (master->unlock)
            slave->mtd.unlock = part_unlock;
        slave->mtd.erase = part_erase;
        slave->master = master;
        slave->offset = parts[i].offset;
        slave->index = i;

        if (slave->offset == MTDPART_OFS_APPEND)
            slave->offset = cur_offset;
        if (slave->mtd.size == MTDPART_SIZ_FULL)
            slave->mtd.size = master->size - slave->offset;
        cur_offset = slave->offset + slave->mtd.size;

        printk (KERN_NOTICE "0x%08x-0x%08x : \"%s\"\n", slave->offset,
            slave->offset + slave->mtd.size, slave->mtd.name);

        /* let's do some sanity checks */          //安全检查
        if (slave->offset >= master->size) {
                /* let's register it anyway to preserve ordering */
            slave->offset = 0;
            slave->mtd.size = 0;
            printk ("mtd: partition \"%s\" is out of reach -- disabled\n",
                parts[i].name);
        }
        if (slave->offset + slave->mtd.size > master->size) {
            slave->mtd.size = master->size - slave->offset;
            printk ("mtd: partition \"%s\" extends beyond the end of device \"%s\" -- size truncated to %#x\n",
                parts[i].name, master->name, slave->mtd.size);
        }
        if (master->numeraseregions>1) {
            /* Deal with variable erase size stuff */
            int i;
            struct mtd_erase_region_info *regions = master->eraseregions;

            /* Find the first erase regions which is part of this partition. */
            for (i=0; i < master->numeraseregions && slave->offset >= regions[i].offset; i++)
                ;
                for (i--; i < master->numeraseregions && slave->offset + slave->mtd.size > regions[i].offset; i++)
{
                    if (slave->mtd.erasesize < regions[i].erasesize) {
                        slave->mtd.erasesize = regions[i].erasesize;
                    }
                }
            } else {
                /* Single erase size */
                slave->mtd.erasesize = master->erasesize;
            }

            if ((slave->mtd.flags & MTD_WRITEABLE) &&
                (slave->offset % slave->mtd.erasesize)) {
                /* Doesn't start on a boundary of major erase size */
                /* FIXME: Let it be writable if it is on a boundary of _minor_ erase size though */
                slave->mtd.flags &= ~MTD_WRITEABLE;
                printk ("mtd: partition \"%s\" doesn't start on an erase block boundary -- force read-only\n",parts[i].
name);
            }
            if ((slave->mtd.flags & MTD_WRITEABLE) &&
                (slave->mtd.size % slave->mtd.erasesize)) {
                slave->mtd.flags &= ~MTD_WRITEABLE;
                printk ("mtd: partition \"%s\" doesn't end on an erase block -- force read-only\n", parts[i].name);
            }

            /* register our partition */
            add_mtd_device(&slave->mtd);
        }

        return 0;
}
```

del_mtd_partitions

格式：

```
    int del_mtd_partitions(struct mtd_info *master)
```

注释：

```
/*
 * This function unregisters and destroy all slave MTD objects which are
 * attached to the given master MTD object.
 */
```

功能：

　　删除master上的所有分区。

说明：

对mtd_partitions上的每一个分区{
如果它的主分区是master，将它从mtd_partitions和mtd_table中删除并free掉
}
参数：
master：被删除分区的主分区
返回：
返回0
调用：
del_mtd_device()
kfree()
被调用：
flash相关的清除函数（maps子目录下的文件中的cleanup_xxx函数）
源代码：

```
{
	struct list_head *node;
	struct mtd_part *slave;

	for (node = mtd_partitions.next;
	     node != &mtd_partitions;
	     node = node->next) {
	       slave = list_entry(node, struct mtd_part, list);
	       if (slave->master == master) {
	              struct list_head *prev = node->prev;
	              __list_del(prev, node->next);        //从mtd_partitions中删除
	              del_mtd_device(&slave->mtd);     //从mtd_table中删除
	              kfree(slave);
	              node = prev;
	       }
	}

	return 0;
}
```

part_read

part_write

part_readv

part_writev

part_erase

part_lock

part_unlock

part_sync

part_suspend

part_resume

格式：
　　略
注释：
```
/*
 * MTD methods which simply translate the effective address and pass through
 * to the _real_ device.
 */
```
功能：
　　非主分区的xxxx函数。
说明：
　　进行安全性检查,
　　调用主分区的xxxx函数
参数：
　　略
返回：
　　略
调用：
　　略
被调用：
　　在add_mtd_partitions中被注册进非主分区的md_info
源代码：
　　略

mtdblock.c

　　MTD设备层的块设备的相关数据结构和程序，其中定义了MTD块设备的notifier和MTD块设备的结构mtdblk_dev类型的**mtdblks**数组，该数组中的每个成员与MTD原始设备层的mtd_table数组中的每个成员一一对应。

notifier

　　设备层的mtdblcok设备的notifier
```
static struct mtd_notifier notifier = {
        mtd_notify_add,
        mtd_notify_remove,
        NULL
};
```

mtdblk_dev

设备层的块设备的结构（and its private , I think），MTD字符设备没有相对应的结构。

```
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（与MTD设备块对齐）
      unsigned long cache_offset;             缓冲区内数据在原始设备层MTD设备内的偏移
      unsigned int cache_size;                缓冲区大小（通常被设置为MTD设备的erasesize）
      enum { STATE_EMPTY, STATE_CLEAN, STATE_DIRTY } cache_state;       缓冲区状态
}
```

mtdblks

设备层（块）设备数组（与原始设备层的MAX_MTD_DEVICES个MTD设备一一对应）

```
static struct mtdblk_dev *mtdblks[MAX_MTD_DEVICES];
```

erase_callback

格式：

```
      static void erase_callback(struct erase_info *done)
```

注释：

```
/*
 * Cache stuff...
 *
 * Since typical flash erasable sectors are much larger than what Linux's
 * buffer cache can handle, we must implement read-modify-write on flash
 * sectors for each block write requests.  To avoid over-erasing flash sectors
 * and to speed things up, we locally cache a whole flash sector while it is
 * being written to until a different sector is required.
 */
```

功能：

回调函数，当写完时被调用。

说明：

在erase_write中被赋给erase_info结构

参数：

done：

返回：

无

调用：

wake_up

被调用：

在erase_write中被注册

源代码：

```
{
	wait_queue_head_t *wait_q = (wait_queue_head_t *)done->priv;
	wake_up(wait_q);
}
```

erase_write

格式：

static int erase_write (struct mtd_info *mtd, unsigned long pos,	int len, const char *buf)

注释：

无

功能：

擦除并写flash。

说明：

1. 声明一个erase_info变量erase并给其赋值,
2. 调用原始设备层函数mtd_info->erase(mtd,erase),
3. 调用原始设备层函数mtd_info->write()

参数：

mtd：被操作的MTD原始设备

pos：MTD原始设备的起始位置

len：长度

buf：被写入的内容

返回：

FIXME

调用：

mtd_info->erase

mtd_info->write

被调用：

write_cached_data

do_cached_write

源代码：

```
{
	struct erase_info erase;
	DECLARE_WAITQUEUE(wait, current);
	wait_queue_head_t wait_q;
	size_t retlen;
```

```
    int ret;

    /*
     * First, let's erase the flash block.
     */

    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);          //调用mtd->erase()
    if (ret) {
        set_current_state(TASK_RUNNING);
        remove_wait_queue(&wait_q, &wait);
        printk (KERN_WARNING "mtdblock: erase of region [0x%lx, 0x%x] "
                     "on \"%s\" failed\n",
            pos, len, mtd->name);
        return ret;
    }

    schedule();  /* Wait for erase to finish. */
    remove_wait_queue(&wait_q, &wait);

    /*
     * Next, writhe data to flash.
     */

    ret = MTD_WRITE (mtd, pos, len, &retlen, buf);          //调用mtd->write()
    if (ret)
        return ret;
    if (retlen != len)
        return -EIO;
    return 0;
}
```

write_cached_data

格式：

```
    static int write_cached_data (struct mtdblk_dev *mtdblk)
```

注释：
　　无
功能：
　　将指定设备层MTD块设备缓冲区内的数据写入MTD设备
说明：
　　无
参数：
　　mtdblk：指定的设备层MTD块设备结构
返回：
　　成功：返回0
　　失败：FIXME
调用：
　　erase_write()写入MTD设备
被调用：
　　do_cached_write()
　　mtdblock_release()
　　mtdblock_ioctl()
源代码：

```
{
	struct mtd_info *mtd = mtdblk->mtd;
	int ret;

	if (mtdblk->cache_state != STATE_DIRTY)
		return 0;

	DEBUG(MTD_DEBUG_LEVEL2, "mtdblock: writing cached data for \"%s\" "
			"at 0x%lx, size 0x%x\n", mtd->name,
			mtdblk->cache_offset, mtdblk->cache_size);

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

	/*
	 * Here we could argably set the cache state to STATE_CLEAN.
	 * However this could lead to inconsistency since we will not
	 * be notified if this content is altered on the flash by other
	 * means.  Let's declare it empty and leave buffering tasks to
	 * the buffer cache instead.
	 */
	mtdblk->cache_state = STATE_EMPTY;
	return 0;
}
```

do_cached_write

格式：

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

注释：

无

功能：

从buf向指定的设备层MTD块设备中缓冲写

说明：

```
如果缓冲区大小为0，则调用mtd_info->write()直接向MTD设备中写，
当（len>0）{
    如果是整块block，则直接向MTD设备中写
    否则{
     先调用write_cached_data()将缓冲内的数据写入MTD设备
        在调用mtd_info->read从MTD设备中将对应块的数据读入缓冲
        将buf的数据（非整块）写入缓冲
    }
    len自减
}
```

参数：

mtdblk：指定的MTD块设备
pos：写入原始设备层MTD设备的位置
len：数据长度
buf：被写入的数据

返回：

成功：返回0
失败：返回错误码

调用：

erase_write()用于直接写MTD设备
write_cached_data()用于清空MTD块设备的缓冲区
mtd_info->read()用于从MTD设备中读数据
mtd_info->write()用于向MTD设备中写数据

被调用：

handle_mtdblock_request()

源代码：

```
{
    struct mtd_info *mtd = mtdblk->mtd;
    unsigned int sect_size = mtdblk->cache_size;
    size_t retlen;
    int ret;

    DEBUG(MTD_DEBUG_LEVEL2, "mtdblock: write on \"%s\" at 0x%lx, size 0x%x\n",
        mtd->name, pos, len);

    if (!sect_size)
```

```
		return MTD_WRITE (mtd, pos, len, &retlen, buf);

	while (len > 0) {
		unsigned long sect_start = (pos/sect_size)*sect_size;//we should write by erase block
		unsigned int offset = pos - sect_start;//real start address
		unsigned int size = sect_size - offset;
		if( size > len )
			size = len;

		if (size == sect_size) {
			/*
			 * We are covering a whole sector.  Thus there is no
			 * need to bother with the cache while it may still be
			 * useful for other partial writes.
			 */
			ret = erase_write (mtd, pos, size, buf);
			if (ret)
				return ret;
		} else {
			/* Partial sector: need to use the cache */

			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) {
				/* fill the cache with the current sector */
				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;
			}

			/* write data to our local cache */
			memcpy (mtdblk->cache_data + offset, buf, size);
			mtdblk->cache_state = STATE_DIRTY;
```

```
                }

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

        return 0;
}
```

do_cached_read

格式：

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

注释：

　　无

功能：

　　从指定的MTD块设备中缓冲读到指定位置buf

说明：

　　如果缓冲区大小为0，调用mtd_info->read()直接从原始设备层MTD设备中读取

　　否则当（len>0）{

　　　　如果缓冲区不为空且其中数据正好是所需数据，则从缓冲区拷贝到buf

　　　　否则调用mtd_info->read()从MTD中读

　　}

参数：

　　mtdblk：指定的MTD块设备

　　pos：MTD设备中指定的位置

　　len：长度

　　buf：被写入的地址

返回：

　　成功：返回0

　　失败：返回错误码

调用：

　　mtd_info->read()从指定的MTD设备中读

被调用：

　　handle_mtdblock_request()

源代码：

```
{
        struct mtd_info *mtd = mtdblk->mtd;
        unsigned int sect_size = mtdblk->cache_size;
        size_t retlen;
        int ret;
```

```
    DEBUG(MTD_DEBUG_LEVEL2, "mtdblock: read on \"%s\" at 0x%lx, size 0x%x\n",
            mtd->name, pos, len);

    if (!sect_size)
        return MTD_READ (mtd, pos, len, &retlen, buf);

    while (len > 0) {
        unsigned long sect_start = (pos/sect_size)*sect_size;
        unsigned int offset = pos - sect_start;
        unsigned int size = sect_size - offset;
        if (size > len)
            size = len;

        /*
         * Check if the requested data is already cached
         * Read the requested amount of data from our internal cache if it
         * contains what we want, otherwise we read the data directly
         * from flash.
         */
        if (mtdblk->cache_state != STATE_EMPTY &&
          mtdblk->cache_offset == sect_start) {
            memcpy (buf, mtdblk->cache_data + offset, size);
        } else {
            ret = MTD_READ (mtd, pos, size, &retlen, buf);
            if (ret)
                return ret;
            if (retlen != size)
                return -EIO;
        }

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

    return 0;
}
```

mtdblock_open

格式：

```
    static int mtdblock_open(struct inode *inode, struct file *file)
```

注释：

无

功能：

打开一个MTD块设备（设备层）

说明：

获得指定节点在原始设备层的MTD设备（mtd_table），
如果 其在设备层的对应块设备（mtdblks）被打开，返回，
否则 将其打开并初始化，
将MTD设备的大小/1024赋给对应的mtd_sizes，
将MTD设备的erasesize和PAGE_SIZE中较小的一个赋给对应的mtd_blksizes。

参数：

inode：指定的索引节点
file：没用

返回：

成功：返回0
失败：返回错误码

调用：

get_mtd_device()和put_mtd_device()获取原始设备层的MTD设备

被调用：

注册进mtd_fops结构

源代码：

```
{
	struct mtdblk_dev *mtdblk;
	struct mtd_info *mtd;
	int dev;

	DEBUG(MTD_DEBUG_LEVEL1,"mtdblock_open\n");

	if (!inode)
		return -EINVAL;

	dev = MINOR(inode->i_rdev);
	if (dev >= MAX_MTD_DEVICES)
		return -EINVAL;

	MOD_INC_USE_COUNT;

	mtd = get_mtd_device(NULL, dev);
	if (!mtd)
		return -ENODEV;
	if (MTD_ABSENT == mtd->type) {
		put_mtd_device(mtd);
		MOD_DEC_USE_COUNT;
		return -ENODEV;
	}

	spin_lock(&mtdblks_lock);

	/* If it's already open, no need to piss about. */
	if (mtdblks[dev]) {
		mtdblks[dev]->count++;
		spin_unlock(&mtdblks_lock);
		return 0;
	}

	/* OK, it's not open. Try to find it */

	/* First we have to drop the lock, because we have to
	   to things which might sleep.
	*/
	spin_unlock(&mtdblks_lock);

	mtdblk = kmalloc(sizeof(struct mtdblk_dev), GFP_KERNEL);
	if (!mtdblk) {
		put_mtd_device(mtd);
		MOD_DEC_USE_COUNT;
		return -ENOMEM;
	}
	memset(mtdblk, 0, sizeof(*mtdblk));
	mtdblk->count = 1;
	mtdblk->mtd = mtd;

	init_MUTEX (&mtdblk->cache_sem);
	mtdblk->cache_state = STATE_EMPTY;
	if ((mtdblk->mtd->flags & MTD_CAP_RAM) != MTD_CAP_RAM &&
	    mtdblk->mtd->erasesize) {
		mtdblk->cache_size = mtdblk->mtd->erasesize;
		mtdblk->cache_data = vmalloc(mtdblk->mtd->erasesize);
		if (!mtdblk->cache_data) {
			put_mtd_device(mtdblk->mtd);
			kfree(mtdblk);
			MOD_DEC_USE_COUNT;
			return -ENOMEM;
		}
	}

	/* OK, we've created a new one. Add it to the list. */

	spin_lock(&mtdblks_lock);

	if (mtdblks[dev]) {
		/* Another CPU made one at the same time as us. */
		mtdblks[dev]->count++;
		spin_unlock(&mtdblks_lock);
		put_mtd_device(mtdblk->mtd);
		vfree(mtdblk->cache_data);
		kfree(mtdblk);
		return 0;
	}

	mtdblks[dev] = mtdblk;
	mtd_sizes[dev] = mtdblk->mtd->size/1024;
	if (mtdblk->mtd->erasesize)
		mtd_blksizes[dev] = mtdblk->mtd->erasesize;
	if (mtd_blksizes[dev] > PAGE_SIZE)
		mtd_blksizes[dev] = PAGE_SIZE;
	set_device_ro (inode->i_rdev, !(mtdblk->mtd->flags & MTD_WRITEABLE));

	spin_unlock(&mtdblks_lock);

	DEBUG(MTD_DEBUG_LEVEL1, "ok\n");

	return 0;
}
```

mtdblock_release

格式：

    static release_t mtdblock_release(struct inode *inode, struct file *file)

注释：

　　无

功能：

　　释放一个设备层的MTD块设备

说明：

　　调用write_cached_data刷新缓冲区,

　　将MTD块设备使用数减1，如果变为0，释放原始设备层的MTD设备

参数：

　　inode：指定的索引节点

　　file：无用

返回：

　　成功：返回0

　　inode为NULL：返回-ENODEV

调用：

　　write_cached_data()刷新缓冲

　　put_mtd_device()释放MTD设备

被调用：

　　注册进mtd_fops结构

源代码：

```
{
        int dev;
        struct mtdblk_dev *mtdblk;
     DEBUG(MTD_DEBUG_LEVEL1, "mtdblock_release\n");

        if (inode == NULL)
                release_return(-ENODEV);

        invalidate_device(inode->i_rdev, 1);

        dev = MINOR(inode->i_rdev);
        mtdblk = mtdblks[dev];

        down(&mtdblk->cache_sem);
        write_cached_data(mtdblk);
        up(&mtdblk->cache_sem);

        spin_lock(&mtdblks_lock);
        if (!--mtdblk->count) {
                /* It was the last usage. Free the device */
                mtdblks[dev] = NULL;
                spin_unlock(&mtdblks_lock);
                if (mtdblk->mtd->sync)
                        mtdblk->mtd->sync(mtdblk->mtd);
                put_mtd_device(mtdblk->mtd);
                vfree(mtdblk->cache_data);
                kfree(mtdblk);
        } else {
                spin_unlock(&mtdblks_lock);
        }

        DEBUG(MTD_DEBUG_LEVEL1, "ok\n");

        MOD_DEC_USE_COUNT;
        release_return(0);
}
```

handle_mtdblock_request

格式：

```
        static void handle_mtdblock_request(void)
```

注释：

```
/*
 * This is a special request_fn because it is executed in a process context
 * to be able to sleep independently of the caller.  The io_request_lock
 * is held upon entry and exit.
 * The head of our request queue is considered active so there is no need
 * to dequeue requests before we are done.
 */
```

功能：
    处理对MTD块设备的请求
说明：
    获取当前请求，分情况处理
参数：
    无
返回：
    无
调用：
    do_cached_read()处理读请求
    do_cached_write()处理写请求
被调用：
    mtdblock_thread()
源代码：

```
{
    struct request *req;
    struct mtdblk_dev *mtdblk;
    unsigned int res;

    for (;;) {
        INIT_REQUEST;
        req = CURRENT;
        spin_unlock_irq(&io_request_lock);
        mtdblk = mtdblks[MINOR(req->rq_dev)];
        res = 0;

        if (MINOR(req->rq_dev) >= MAX_MTD_DEVICES)
            panic(__FUNCTION__": minor out of bound");

        if ((req->sector + req->current_nr_sectors) > (mtdblk->mtd->size >> 9))
            goto end_req;

        // Handle the request
        switch (req->cmd)
        {
            int err;

            case READ:
            down(&mtdblk->cache_sem);
            err = do_cached_read (mtdblk, req->sector << 9,
                    req->current_nr_sectors << 9,
                    req->buffer);
```

```
                up(&mtdblk->cache_sem);
                if (!err)
                        res = 1;
                break;

                case WRITE:
                // Read only device
                if ( !(mtdblk->mtd->flags & MTD_WRITEABLE) )
                        break;

                // Do the write
                down(&mtdblk->cache_sem);
                err = do_cached_write (mtdblk, req->sector << 9,
                                req->current_nr_sectors << 9,
                                req->buffer);
                up(&mtdblk->cache_sem);
                if (!err)
                        res = 1;
                break;
        }

end_req:
        spin_lock_irq(&io_request_lock);
        end_request(res);
    }
}
```

leaving

static volatile int leaving = 0;　　控制线程的变量，在mtdblock_thread作为测试条件，在cleanup_mtdblock中被赋1

mtdblock_thread

格式：

　int mtdblock_thread(void *dummy)

注释：

　无

功能：

　启动一个MTD块设备线程

说明：

　调用handle_mtdblock_request()处理对MTD块设备的读写

参数：

　dummy：FIXME

返回：

　线程结束：返回0

调用：

handle_mtdblock_request()

被调用：

init_mtdblock()

源代码：

```
{
	struct task_struct *tsk = current;
	DECLARE_WAITQUEUE(wait, tsk);

	tsk->session = 1;
	tsk->pgrp = 1;
	/* we might get involved when memory gets low, so use PF_MEMALLOC */
	tsk->flags |= PF_MEMALLOC;
	strcpy(tsk->comm, "mtdblockd");
	tsk->tty = NULL;
	spin_lock_irq(&tsk->sigmask_lock);
	sigfillset(&tsk->blocked);
	recalc_sigpending(tsk);
	spin_unlock_irq(&tsk->sigmask_lock);
	exit_mm(tsk);
	exit_files(tsk);
	exit_sighand(tsk);
	exit_fs(tsk);

	while (!leaving) {
		add_wait_queue(&thr_wq, &wait);
		set_current_state(TASK_INTERRUPTIBLE);
		spin_lock_irq(&io_request_lock);
		if (QUEUE_EMPTY || QUEUE_PLUGGED) {
			spin_unlock_irq(&io_request_lock);
			schedule();
			remove_wait_queue(&thr_wq, &wait);
		} else {
			remove_wait_queue(&thr_wq, &wait);
			set_current_state(TASK_RUNNING);
			handle_mtdblock_request();
			spin_unlock_irq(&io_request_lock);
		}
	}

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

mtdblock_ioctl

格式：

```
static int mtdblock_ioctl(struct inode * inode, struct file * file,
            unsigned int cmd, unsigned long arg)
```

注释：
    无
功能：
    对MTD块设备的IO控制
说明：
    无
参数：
    FIXME
返回：
    成功：返回0
    失败：返回错误码
调用：
    write_cached_data()
被调用：
    被注册进mtd_fops结构
源代码：

```
{
	struct mtdblk_dev *mtdblk;

	mtdblk = mtdblks[MINOR(inode->i_rdev)];

#ifdef PARANOIA
	if (!mtdblk)
		BUG();
#endif

	switch (cmd) {
	case BLKGETSIZE:   /* Return device size */
		return put_user((mtdblk->mtd->size >> 9), (long *) arg);

#ifdef BLKGETSIZE64
	case BLKGETSIZE64:
		return put_user((u64)mtdblk->mtd->size, (u64 *)arg);
#endif

	case BLKFLSBUF:
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2,2,0)
		if(!capable(CAP_SYS_ADMIN))
			return -EACCES;
#endif
		fsync_dev(inode->i_rdev);
```

```
        invalidate_buffers(inode->i_rdev);
        down(&mtdblk->cache_sem);
        write_cached_data(mtdblk);
        up(&mtdblk->cache_sem);
        if (mtdblk->mtd->sync)
            mtdblk->mtd->sync(mtdblk->mtd);
        return 0;

    default:
        return -EINVAL;
    }
}
```

mtd_fops

对MTD块设备操作的函数结构

```
static struct block_device_operations mtd_fops =
{
    open: mtdblock_open,
    release: mtdblock_release,
    ioctl: mtdblock_ioctl
};
```

init_mtdblock

格式：

```
    int __init init_mtdblock(void)
```

注释：

无

功能：

初始化MTD块设备

说明：

如果定义了CONFIG_DEVFS_FS{
    调用devfs_register_blkdev()注册块设备,
    调用register_mtd_user()将设备层MTD块设备的notifier注册进原始设备层
}
 否则调用register_blkdev()注册块设备,
初始化mtd_sizes和mtd_blksizes数组,
启动mtdblock_thread线程

参数：

无

返回：

成功：返回0
注册块设备失败：返回-EAGAIN

调用：

    kernel_thread()启动mtdblock_thread线程
    register_blkdev()注册块设备

被调用：

    module_init
    __init

源代码：

```
{
    int i;

    spin_lock_init(&mtdblks_lock);
#ifdef CONFIG_DEVFS_FS
    if (devfs_register_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME, &mtd_fops))
    {
        printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology Devices.\n",
            MTD_BLOCK_MAJOR);
        return -EAGAIN;
    }

    devfs_dir_handle = devfs_mk_dir(NULL, DEVICE_NAME, NULL);
    register_mtd_user(&notifier);
#else
    if (register_blkdev(MAJOR_NR,DEVICE_NAME,&mtd_fops)) {
        printk(KERN_NOTICE "Can't allocate major number %d for Memory Technology Devices.\n",
            MTD_BLOCK_MAJOR);
        return -EAGAIN;
    }
#endif

    /* We fill it in at open() time. */
    for (i=0; i< MAX_MTD_DEVICES; i++) {
        mtd_sizes[i] = 0;
        mtd_blksizes[i] = BLOCK_SIZE;
    }
    init_waitqueue_head(&thr_wq);
    /* Allow the block size to default to BLOCK_SIZE. */
    blksize_size[MAJOR_NR] = mtd_blksizes;
    blk_size[MAJOR_NR] = mtd_sizes;

    blk_init_queue(BLK_DEFAULT_QUEUE(MAJOR_NR), &mtdblock_request);
    kernel_thread (mtdblock_thread, NULL, CLONE_FS|CLONE_FILES|CLONE_SIGHAND);
    return 0;
}
```

cleanup_mtdblock

格式：

    static void __exit cleanup_mtdblock(void)

注释：

    无

功能：

    清除MTD块设备

说明：

    如果定义了CONFIG_DEVFS_FS{

        调用unregister_mtd_user()注销MTD块设备的notifier

        调用devfs_unregister_blkdev()注销MTD块设备

    }

否则调用unregister_blkdev()注销MTD块设备

参数：

    无

返回：

    无

调用：

    unregister_blkdev()注销MTD块设备

被调用：

    __exit

    module_exit

源代码：

```
{
	leaving = 1;
	wake_up(&thr_wq);
	down(&thread_sem);
#ifdef CONFIG_DEVFS_FS
	unregister_mtd_user(&notifier);
	devfs_unregister(devfs_dir_handle);
	devfs_unregister_blkdev(MTD_BLOCK_MAJOR, DEVICE_NAME);
#else
	unregister_blkdev(MAJOR_NR,DEVICE_NAME);
#endif
	blk_cleanup_queue(BLK_DEFAULT_QUEUE(MAJOR_NR));
	blksize_size[MAJOR_NR] = NULL;
	blk_size[MAJOR_NR] = NULL;
}
```

mtdchar.c

MTD字符设备的文件

notifier

MTD字符设备的notifier

```
static struct mtd_notifier notifier = {
	add: mtd_notify_add,
	remove:   mtd_notify_remove,
};
```

mtd_lseek

格式：

```
	static loff_t mtd_lseek (struct file *file, loff_t offset, int orig)
```

注释：

　无

功能：

　置MTD字符设备的指针

说明：

　无

参数：

　file：MTD字符设备文件

　offset：偏移

　orig：设置方式

返回：

　成功：返回当前指针位置

　设置方式无效：返回-EINVAL

调用：

　无

被调用：

　注册进mtd_fops结构

源代码：

```
{
	struct mtd_info *mtd=(struct mtd_info *)file->private_data;

	switch (orig) {
	case 0:
		/* SEEK_SET */
		file->f_pos = offset;
		break;
	case 1:
		/* SEEK_CUR */
		file->f_pos += offset;
		break;
	case 2:
		/* SEEK_END */
		file->f_pos =mtd->size + offset;
		break;
	default:
		return -EINVAL;
	}

	if (file->f_pos < 0)
		file->f_pos = 0;
	else if (file->f_pos >= mtd->size)
		file->f_pos = mtd->size - 1;

	return file->f_pos;
}
```

mtd_open

格式：

	static int mtd_open(struct inode *inode, struct file *file)

注释：

	无

功能：

	打开一个MTD字符设备

说明：

	devnum=minor>>2（参看Documentations/devices.txt），
	进行安全性检查,
	调用get_mtd_device获取MTD设备，并将file->private_data指向它

参数：

	inode：FIXME
	file：是系统提供给MTD字符设备用于传递参数的file结构，此函数中它的private_data 成员被指向原始设备层的MTD设备

返回：

	成功：返回0
	失败：返回错误码

调用：

	get_mtd_device()获得原始设备层的MTD设备

被调用：

	注册进mtd_fops结构

源代码：

```
{
	int minor = MINOR(inode->i_rdev);
	int devnum = minor >> 1;
	struct mtd_info *mtd;

	DEBUG(MTD_DEBUG_LEVEL0, "MTD_open\n");

	if (devnum >= MAX_MTD_DEVICES)
```

```
		return -ENODEV;

	/* You can't open the RO devices RW */
	if ((file->f_mode & 2) && (minor & 1))
		return -EACCES;

	mtd = get_mtd_device(NULL, devnum);

	if (!mtd)
		return -ENODEV;

	if (MTD_ABSENT == mtd->type) {
		put_mtd_device(mtd);
		return -ENODEV;
	}

	file->private_data = mtd;

	/* You can't open it RW if it's not a writeable device */
	if ((file->f_mode & 2) && !(mtd->flags & MTD_WRITEABLE)) {
		put_mtd_device(mtd);
		return -EACCES;
	}

	return 0;
} /* mtd_open */
```

mtd_close

格式：

```
	static int mtd_close(struct inode *inode, struct file *file)
```

注释：

无

功能：

关闭一个MTD字符设备

说明：

调用mtd_info->sync()同步MTD设备,

调用put_mtd_device()返还MTD设备

参数：

inode：FIXME

file：无用

返回：

    返回0
调用：
    mtd_info->sync()同步MTD设备
put_mtd_device()返还MTD设备
被调用：
    被注册进mtd_fops结构
源代码：

```
{
    struct mtd_info *mtd;

    DEBUG(MTD_DEBUG_LEVEL0, "MTD_close\n");

    mtd = (struct mtd_info *)file->private_data;

    if (mtd->sync)
        mtd->sync(mtd);

    put_mtd_device(mtd);

    return 0;
} /* mtd_close */
```

MAX_KMALLOC_SIZE

```
/* FIXME: This _really_ needs to die. In 2.5, we should lock the
   userspace buffer down and use it directly with readv/writev.
*/
#define MAX_KMALLOC_SIZE 0x20000
```

mtd_read

格式：
```
    static ssize_t mtd_read(struct file *file, char *buf, size_t count,loff_t *ppos)
```
注释：
    无
功能：
    MTD字符设备的写操作
说明：
    当count>0时{
        裁减本次操作大小len至min(MAX_KMALLOC_SIZE,count),
        申请一块大小为MAX_KMALLOC_SIZE的内核空间kbuf,
        调用mtd_info->read将MTD设备中的数据读入kbuf,
        将kbuf中的数据拷贝到用户空间buf,
        count自减

        释放kbuf
    }
参数：
    file：系统给MTD字符设备驱动程序用于传递参数的file结构，此函数通过file得到下
        层的MTD设备
    buf：用户空间的指针，用于存放读取的数据
    count：被读数据的长度
    ppos：被读数据在MTD设备中的位置
返回：
    成功：返回实际读取数据的长度
    失败：返回错误码
调用：
    mtd_info->read()用于从MTD设备中读取数据
被调用：
    被注册进mtd_fops结构
源代码：

```
{
    struct mtd_info *mtd = (struct mtd_info *)file->private_data;
    size_t retlen=0;
    size_t total_retlen=0;
    int ret=0;
    int len;
    char *kbuf;

    DEBUG(MTD_DEBUG_LEVEL0,"MTD_read\n");

    if (*ppos + count > mtd->size)
        count = mtd->size - *ppos;

    if (!count)
        return 0;

    /* FIXME: Use kiovec in 2.5 to lock down the user's buffers
      and pass them directly to the MTD functions */
    while (count) {
        if (count > MAX_KMALLOC_SIZE)
            len = MAX_KMALLOC_SIZE;
        else
            len = count;

        kbuf=kmalloc(len,GFP_KERNEL);
        if (!kbuf)
            return -ENOMEM;

        ret = MTD_READ(mtd, *ppos, len, &retlen, kbuf);
        if (!ret) {
                *ppos += retlen;
                if (copy_to_user(buf, kbuf, retlen)) {
                         kfree(kbuf);
                        return -EFAULT;
                }
                else
                        total_retlen += retlen;

                count -= retlen;
                buf += retlen;
        }
        else {
                kfree(kbuf);
                return ret;
        }

        kfree(kbuf);
    }

    return total_retlen;
} /* mtd_read */
```

mtd_write

格式：

    static ssize_t mtd_write(struct file *file, const char *buf, size_t count,loff_t *ppos)

注释：

无

功能：

对MTD字符设备的写操作

说明：

```
    当count>0时{
        裁减本次操作大小len至min(MAX_KMALLOC_SIZE,count),
        申请一块大小为MAX_KMALLOC_SIZE的内核空间kbuf,
        将用户空间buf中的数据拷贝到kbuf,
        调用mtd_info->write将kbuf中的数据读入MTD设备,
        count自减
        释放kbuf
    }
```

参数：

file：系统给MTD字符设备驱动程序用于传递参数的file结构，此函数通过file得到下层的MTD设备

buf：用户空间的指针，用于存放将要写入的数据

count：被写数据的长度
ppos：数据被写入MTD设备中的位置

返回：

成功：返回实际读取数据的长度
失败：返回错误码

调用：

mtd_info->write用于写入MTD设备

被调用：

被注册进mtd_fops结构

源代码：

```
{
	struct mtd_info *mtd = (struct mtd_info *)file->private_data;
	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);
		if (!kbuf) {
			printk("kmalloc is null\n");
			return -ENOMEM;
		}

		if (copy_from_user(kbuf, buf, len)) {
			kfree(kbuf);
			return -EFAULT;
		}
```

```
            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 */
```

mtd_erase_callback

格式：

    static void mtd_erase_callback (struct erase_info *instr)

注释：

　　无

功能：

　　唤醒进程

说明：

　　在擦除进行完后被调用

参数：

　　instr：进行的擦除的erase_info结构

返回：

　　无

调用：

　　wake_up()

被调用：

　　在mtd_ioctl中被赋给erase_info->callback

源代码：

```
{
    wake_up((wait_queue_head_t *)instr->priv);
}
```

mtd_ioctl

格式：
    static int mtd_ioctl(struct inode *inode, struct file *file, u_int cmd, u_long arg)
注释：
    无
功能：
    MTD字符设备的IO控制
说明：
    根据cmd分别处理
参数：
    inode：FIXME
    file：传递参数的结构
    cmd：IO控制的命令
    arg：IO控制的参数
返回：
    成功：返回0
    失败：返回错误码
调用：
    无
被调用：
    注册进mtd_fops结构
源代码：

```
{
    struct mtd_info *mtd = (struct mtd_info *)file->private_data;
    int ret = 0;
    u_long size;

    DEBUG(MTD_DEBUG_LEVEL0, "MTD_ioctl\n");

    size = (cmd & IOCSIZE_MASK) >> IOCSIZE_SHIFT;
    if (cmd & IOC_IN) {
        ret = verify_area(VERIFY_READ, (char *)arg, size);
        if (ret) return ret;
    }
    if (cmd & IOC_OUT) {
        ret = verify_area(VERIFY_WRITE, (char *)arg, size);
        if (ret) return ret;
    }

    switch (cmd) {
    case MEMGETREGIONCOUNT:
        if (copy_to_user((int *) arg, &(mtd->numeraseregions), sizeof(int)))
            return -EFAULT;
        break;

    case MEMGETREGIONINFO:
    {
        struct region_info_user ur;

        if (copy_from_user(     &ur,
                        (struct region_info_user *)arg,
                        sizeof(struct region_info_user))) {
            return -EFAULT;
        }

        if (ur.regionindex >= mtd->numeraseregions)
            return -EINVAL;
        if (copy_to_user((struct mtd_erase_region_info *) arg,
                &(mtd->eraseregions[ur.regionindex]),
                sizeof(struct mtd_erase_region_info)))
            return -EFAULT;
        break;
    }

    case MEMGETINFO:
        if (copy_to_user((struct mtd_info *)arg, mtd,
                 sizeof(struct mtd_info_user)))
            return -EFAULT;
        break;

    case MEMERASE:
    {
        struct erase_info *erase=kmalloc(sizeof(struct erase_info),GFP_KERNEL);
        if (!erase)
            ret = -ENOMEM;
        else {
            wait_queue_head_t waitq;
            DECLARE_WAITQUEUE(wait, current);

            init_waitqueue_head(&waitq);

            memset (erase,0,sizeof(struct erase_info));
            if (copy_from_user(&erase->addr, (u_long *)arg,
                        2 * sizeof(u_long))) {
                kfree(erase);
                return -EFAULT;
            }
            erase->mtd = mtd;
            erase->callback = mtd_erase_callback;
            erase->priv = (unsigned long)&waitq;

            /*
              FIXME: Allow INTERRUPTIBLE. Which means
```

```
                  not having the wait_queue head on the stack.

                  If the wq_head is on the stack, and we
                  leave because we got interrupted, then the
                  wq_head is no longer there when the
                  callback routine tries to wake us up.
                */
                ret = mtd->erase(mtd, erase);
                if (!ret) {
                     set_current_state(TASK_UNINTERRUPTIBLE);
                     add_wait_queue(&waitq, &wait);
                     if (erase->state != MTD_ERASE_DONE &&
                        erase->state != MTD_ERASE_FAILED)
                          schedule();
                     remove_wait_queue(&waitq, &wait);
                     set_current_state(TASK_RUNNING);

                     ret = (erase->state == MTD_ERASE_FAILED)?-EIO:0;
                }
                kfree(erase);
           }
           break;
      }

      case MEMWRITEOOB:
      {
           struct mtd_oob_buf buf;
           void *databuf;
           ssize_t retlen;

           if (copy_from_user(&buf, (struct mtd_oob_buf *)arg, sizeof(struct mtd_oob_buf)))
                return -EFAULT;

           if (buf.length > 0x4096)
                return -EINVAL;

           if (!mtd->write_oob)
                ret = -EOPNOTSUPP;
           else
                ret = verify_area(VERIFY_READ, (char *)buf.ptr, buf.length);

           if (ret)
                return ret;

           databuf = kmalloc(buf.length, GFP_KERNEL);
           if (!databuf)
```

```
                return -ENOMEM;

            if (copy_from_user(databuf, buf.ptr, buf.length)) {
                kfree(databuf);
                return -EFAULT;
            }

            ret = (mtd->write_oob)(mtd, buf.start, buf.length, &retlen, databuf);

            if (copy_to_user((void *)arg + sizeof(u_int32_t), &retlen, sizeof(u_int32_t)))
                ret = -EFAULT;

            kfree(databuf);
            break;

        }

        case MEMREADOOB:
        {
            struct mtd_oob_buf buf;
            void *databuf;
            ssize_t retlen;

            if (copy_from_user(&buf, (struct mtd_oob_buf *)arg, sizeof(struct mtd_oob_buf)))
                return -EFAULT;

            if (buf.length > 0x4096)
                return -EINVAL;

            if (!mtd->read_oob)
                ret = -EOPNOTSUPP;
            else
                ret = verify_area(VERIFY_WRITE, (char *)buf.ptr, buf.length);

            if (ret)
                return ret;

            databuf = kmalloc(buf.length, GFP_KERNEL);
            if (!databuf)
                return -ENOMEM;

            ret = (mtd->read_oob)(mtd, buf.start, buf.length, &retlen, databuf);

            if (copy_to_user((void *)arg + sizeof(u_int32_t), &retlen, sizeof(u_int32_t)))
                ret = -EFAULT;
            else if (retlen && copy_to_user(buf.ptr, databuf, retlen))
```

```
                ret = -EFAULT;

            kfree(databuf);
            break;
        }

        case MEMLOCK:
        {
            unsigned long adrs[2];

            if (copy_from_user(adrs ,(void *)arg, 2* sizeof(unsigned long)))
                return -EFAULT;

            if (!mtd->lock)
                ret = -EOPNOTSUPP;
            else
                ret = mtd->lock(mtd, adrs[0], adrs[1]);
            break;
        }

        case MEMUNLOCK:
        {
            unsigned long adrs[2];

            if (copy_from_user(adrs, (void *)arg, 2* sizeof(unsigned long)))
                return -EFAULT;

            if (!mtd->unlock)
                ret = -EOPNOTSUPP;
            else
                ret = mtd->unlock(mtd, adrs[0], adrs[1]);
            break;
        }


        default:
            DEBUG(MTD_DEBUG_LEVEL0, "Invalid ioctl %x (MEMGETINFO = %x)\n", cmd, 
MEMGETINFO);
            ret = -ENOTTY;
        }

        return ret;
} /* memory_ioctl */
```

mtd_fops

MTD字符设备的操作函数结构

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

init_mtdchar

格式：

```
	static int __init init_mtdchar(void)
```

注释：

	无

功能：

	初始化一个MTD字符设备（设备层）

说明：

	如果定义了CONFIG_DEVFS_FS{
	 调用devfs_register_chrdev()注册MTD字符设备
	 调用register_mtd_user()将设备层MTD字符设备的notifier注册进原始设备层
	}
	否则调用register_chrdev()注册MTD字符设备

参数：

	无

返回：

	成功：返回0
	注册失败：返回-EAGAIN

调用：

	devfs_register_chrdev()或register_chrdev()注册字符设备

被调用：

	__init
	module_init

源代码：

```
{
#ifdef CONFIG_DEVFS_FS
	if (devfs_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;
        }

        devfs_dir_handle = devfs_mk_dir(NULL, "mtd", NULL);

        register_mtd_user(&notifier);
#else
        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;
        }
#endif

        return 0;
}
```

cleanup_mtdchar

格式：

    static void __exit cleanup_mtdchar(void)

注释：

    无

功能：

    清除MTD字符设备

说明：

    如果定义了CONFIG_DEVFS_FS{
        调用unregister_mtd_user()注销MTD字符设备的notifier,
        调用devfs_unregister_chrdev()注销MTD字符设备
}
     否则调用unregister_chrdev()注销MTD字符设备

参数：

    无

返回：

    无

调用：

    unregister_chrdev()或devfs_unregister_chrdev()注销字符设备

被调用：

    __exit
    module_exit

源代码：

```
{
#ifdef CONFIG_DEVFS_FS
```

```
	unregister_mtd_user(&notifier);
	devfs_unregister(devfs_dir_handle);
	devfs_unregister_chrdev(MTD_CHAR_MAJOR, "mtd");
#else
	unregister_chrdev(MTD_CHAR_MAJOR, "mtd");
#endif
}
```

/drivers/mtd/chips子目录

/drivers/mtd/chips下文件的主要功能是探测MTD，该目录下文件是chipreg.c、gen_probe.c、cfi_probe.c、jedec_probe.c、cfi_cmdset_0001.c、cfi_cmdset_0002.c、map_rom.c、map_ram.c、map_absent.c、amd_flash.c、jedec.c和sharp.c，下面介绍每个文件的功能
为了确定一个Flash是否是一个CFI使能的flash memory器件，首先要往Flash的地址0x55H写入数据0x98H，然后从Flash的地址0x10H处开始连续读取3个存储单元中的内容，如果数据总线返回的3个存储单元的字符分别为'Q','R'和'Y'，那么该器件是一个CFI使能的Flash。在识别Flash为CFI使能器件后，通过查询命令来读取CFI查询结构，这些数据的地址和含义在cfi_ident.h文件中。探测CFI接口Flash设备的程序在文件**cfi_probe.c**中，这些设备的类型为“ 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“ jedec_probe ”。
CFI设备和JEDEC设备都要用到**gen_probe.c**文件。
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的扩展命令集（兼容Intel/Sharp标准命令集）两个，这两个命令集分别在**cfi_cmdset_0002.c**和**cfi_cmdset_0001.c**中实现。
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“ amd_flash ”类型的Flash的探测程序在**amd_flash.c**中。
　　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。
　　所有类型的芯片都通过**chipreg.c**中的do_map_probe()程序驱动

chipreg.c

关于MTD芯片注册的文件，此文件中定义的**chip_drvs_list**是所有芯片类型的驱动器链表，在/drivers/mtd/chips子目录下的其他文件通过调用**register_mtd_chip_driver()**和**unregister_mtd_chip_driver()**向此链表中添加或去除MTD芯片驱动器。
在/drivers/mtd/map/下的文件根据芯片类型调用**do_map_probe()**，do_map_probe()通过**get_mtd_chip_driver()**获得符合指定name的MTD芯片驱动器，再调用获得的芯片驱动器的probe程序。

chip_drvs_list

```
static LIST_HEAD(chip_drvs_list);
```

MTD芯片驱动器列表

register_mtd_chip_driver

格式：

void register_mtd_chip_driver(struct mtd_chip_driver *drv)

注释：

无

功能：

注册MTD的芯片驱动器

说明：

向chip_drvs_list里添加mtd_chip_driver

参数：

drv：被注册的MTD的芯片驱动器

返回：

无

调用：

list_add()

被调用：

各种芯片驱动的初始化程序

源代码：

```
{
	spin_lock(&chip_drvs_lock);
	list_add(&drv->list, &chip_drvs_list);
	spin_unlock(&chip_drvs_lock);
}
```

unregister_mtd_chip_driver

格式：

void unregister_mtd_chip_driver(struct mtd_chip_driver *drv)

注释：

无

功能：

删除一个MTD芯片的驱动器

说明：

从chip_drvs_list中删除一个mtd_chip_driver

参数：

drv：被删除的MTD芯片的驱动器

返回：

无

调用：

list_del()

被调用：

各种芯片驱动的清除程序
源代码：

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

get_mtd_chip_driver

格式：

```
	static struct mtd_chip_driver *get_mtd_chip_driver (char *name)
```

注释：
无
功能：
获得指定名称的MTD的芯片驱动器
说明：
根据name从chip_drvs_list中获得一个mtd_chip_driver
参数：
name：该芯片驱动器的名称
返回：
找到：返回该mtd_chip_driver
没找到：返回NULL
调用：
无
被调用：
do_map_probe
源代码：

```
{
	struct list_head *pos;
	struct mtd_chip_driver *ret = NULL, *this;

	spin_lock(&chip_drvs_lock);

	list_for_each(pos, &chip_drvs_list) {
		this = list_entry(pos, typeof(*this), list);

		if (!strcmp(this->name, name)) {
			ret = this;
			break;
		}
	}
	if (ret && !try_inc_mod_count(ret->module)) {
		/* Eep. Failed. */
		ret = NULL;
	}

	spin_unlock(&chip_drvs_lock);

	return ret;
}
```

do_map_probe

格式：

struct mtd_info *do_map_probe(char *name, struct map_info *map)

注释：

　　无

功能：

　　根据name和map的信息探测MTD并返回mtd_info结构

说明：

1. 根据name获得mtd_chip_driver
2. 调用mtd_chip_driver中的probe函数

参数：

　　name：MTD芯片类型

　　map：MTD芯片信息

返回：

　　成功：返回MTD设备的结构mtd_info

　　失败：返回NULL

调用：

　　get_mtd_chip_driver()

　　drv->probe()

被调用：

　　/drivers/mtd/maps/下的板子相关文件中的init_xxxx

源代码：

```
{
	struct mtd_chip_driver *drv;
	struct mtd_info *ret;

	drv = get_mtd_chip_driver(name);

	if (!drv && !request_module(name))
		drv = get_mtd_chip_driver(name);

	if (!drv)
		return NULL;

	ret = drv->probe(map);
#ifdef CONFIG_MODULES
```

```
	/* We decrease the use count here. It may have been a
	  probe-only module, which is no longer required from this
	  point, having given us a handle on (and increased the use
	  count of) the actual driver code.
	*/
	if(drv->module)
		__MOD_DEC_USE_COUNT(drv->module);
#endif

	if (ret)
		return ret;

	return NULL;
}
```

cfi_probe.c

“cfi_probe”型芯片的探测程序，主要由cfi_chip_probe()、cfi_probe()、cfi_chip_setup()、qry_present()、cfi_probe_init()和cfi_probe_exit()这几个函数组成。
cfi_probe()是“cfi_probe”类型芯片的探测程序，它调用通用探测程序**mtd_do_chip_probe()**，并将**cfi_chip_probe**作为参数传递给mtd_do_chip_probe()，mtd_do_chip_probe()将间接调用cfi_chip_probe的成员函数**cfi_probe_chip()**。cfi_probe()注册在“cfi_probe”芯片的驱动器**cfi_chipdrv**中。
cfi_probe_chip()将调用**qry_present()**和**cfi_chip_setup()**初始化cfi_private结构，qry_presetn()负责验证该MTD设备支持CFI接口，cfi_chip_setup()则读出CFI查询结构中的数据（见cfi.h）
cfi_probe_init()和**cfi_probe_exit()**是“cfi_prbe”型芯片驱动器的注册程序和清除程序。

cfi_chipdrv

```
static struct mtd_chip_driver cfi_chipdrv = {
	probe: cfi_probe,		芯片的探测程序
	name: "cfi_probe",		芯片名称
	module: THIS_MODULE
};
```

“cfi_probe”类型MTD芯片的驱动器

cfi_probe_init

格式：

```
	int __init cfi_probe_init(void)
```

注释：

　　无

功能：

初始化“cfi_probe”类型的MTD芯片

说明：

调用register_mtd_chip_driver()将cfi_chipdrv加入MTD驱动器列表chip_drvs_list

参数：

无

返回：

0

调用：

register_mtd_chip_driver()

被调用：

__init

module_init

源代码：

```
{
	register_mtd_chip_driver(&cfi_chipdrv);
	return 0;
}
```

cfi_probe_exit

格式：

```
static void __exit cfi_probe_exit(void)
```

注释：

无

功能：

清除“cfi_probe”MTD芯片驱动

说明：

调用unregister_mtd_chip_driver从MTD芯片驱动器列表chip_drvs_list中删除cfi_chipdrv

参数：

无

返回：

无

调用：

unregister_mtd_chip_driver()

被调用：

__exit

module_exit

源代码：

```
{
	unregister_mtd_chip_driver(&cfi_chipdrv);
}
```

cfi_probe

格式：

struct mtd_info *cfi_probe(struct map_info *map)

注释：

无

功能：

“cfi_probe”类型MTD芯片的探测程序

说明：

调用通用的探测程序mtd_do_chip_probe()，并将cfi_chip_probe作为参数传进去

参数：

map：芯片的相关信息

返回：

MTD设备信息结构mtd_info

调用：

mtd_do_chip_probe

被调用：

注册在cfi_chipdrv中，根据芯片类型被do_map_probe()调用

源代码：

```
{
	/*
	 * Just use the generic probe stuff to call our CFI-specific
	 * chip_probe routine in all the possible permutations, etc.
	 */
	return mtd_do_chip_probe(map, &cfi_chip_probe);
}
```

cfi_chip_probe

```
static struct chip_probe cfi_chip_probe = {
	name: "CFI",
	probe_chip: cfi_probe_chip
};
```

cfi_probe传递给通用探测程序mtd_do_chip_probe的参数

cfi_probe_chip

格式：

```
static int cfi_probe_chip(struct map_info *map, __u32 base,
			struct flchip *chips, struct cfi_private *cfi)
```

注释：

无

功能：

“cfi_probe”类型MTD芯片驱动程序

说明：

1. 调用qry_present()检查是否CFI接口的MTD

2. 如果cfi->numchips=0，调用cfi_chip_setup()设置；（搜索新芯片）
3. 否则探测此芯片是否为原芯片的别名，如果不是，此芯片作为同类芯片加入

参数：

FIXME

返回：

成功：返回1

失败：返回0或-1

调用：

qry_present()

cfi_chip_setup()

被调用：

注册在cfi_chip_probe中

源代码：

```
{
	int i;

	cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);   //reset，进入读模式
	cfi_send_gen_cmd(0x98, 0x55, base, map, cfi, cfi->device_type, NULL);      //发送查询命令

	if (!qry_present(map,base,cfi))          //如果不存在QRY，则不是CFI接口
		return 0;

	if (!cfi->numchips) {          //如果是MTD原始设备中的第一块芯片
		/* This is the first time we're called. Set up the CFI
		   stuff accordingly and return */
		return cfi_chip_setup(map, cfi);
	}

	/* Check each previous chip to see if it's an alias */
	for (i=0; i<cfi->numchips; i++) {
		/* This chip should be in read mode if it's one
		   we've already touched. */
		if (qry_present(map,chips[i].start,cfi)) {
			/* Eep. This chip also had the QRY marker.
			 * Is it an alias for the new one? */
			cfi_send_gen_cmd(0xF0, 0, chips[i].start, map, cfi, cfi->device_type, NULL);

			/* If the QRY marker goes away, it's an alias */
			if (!qry_present(map, chips[i].start, cfi)) {
				printk(KERN_DEBUG "%s: Found an alias at 0x%x for the chip at 0x%lx\n",
					map->name, base, chips[i].start);
				return 0;
			}
			/* Yes, it's actually got QRY for data. Most
			 * unfortunate. Stick the new chip in read mode
			 * too and if it's the same, assume it's an alias. */
```

```
                /* FIXME: Use other modes to do a proper check */
                cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);

                if (qry_present(map, base, cfi)) {
                        printk(KERN_DEBUG "%s: Found an alias at 0x%x for the chip at 0x%lx\n",
                                map->name, base, chips[i].start);
                        return 0;
                }
        }
    }

    /* OK, if we got to here, then none of the previous chips appear to
       be aliases for the current one. */
    if (cfi->numchips == MAX_CFI_CHIPS) {
        printk(KERN_WARNING"%s: Too many flash chips detected. Increase MAX_CFI_CHIPS from %d.
\n", map->name, MAX_CFI_CHIPS);
        /* Doesn't matter about resetting it to Read Mode - we're not going to talk to it anyway */
        return -1;
    }
    chips[cfi->numchips].start = base;
    chips[cfi->numchips].state = FL_READY;
    cfi->numchips++;

    /* Put it back into Read Mode */
    cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);

    printk(KERN_INFO "%s: Found %d x%d devices at 0x%x in %d-bit mode\n",
        map->name, cfi->interleave, cfi->device_type*8, base,
        map->buswidth*8);

    return 1;
}
```

qry_present

格式：

```
    static inline int qry_present(struct map_info *map, __u32 base,
                        struct cfi_private *cfi)
```

注释：

```
/* check for QRY, or search for jedec id.
  in: interleave,type,mode
  ret: table index, <0 for error
 */
```

功能：

　　检查QRY，

说明：

读出从0x10开始的三个字节，如果为”QRY”，则说明是CFI接口的MTD

参数：

map：芯片信息

base：基地址

cfi：CFI私有信息

返回：

成功：返回1

失败：返回0

调用：

无

被调用：

cfi_probe_chip()

源代码：

```
{
    int osf = cfi->interleave * cfi->device_type; // scale factor

    if (cfi_read(map,base+osf*0x10)==cfi_build_cmd('Q',map,cfi) &&
        cfi_read(map,base+osf*0x11)==cfi_build_cmd('R',map,cfi) &&
        cfi_read(map,base+osf*0x12)==cfi_build_cmd('Y',map,cfi))
          return 1;  // ok !

    return 0;       // nothing found
}
```

cfi_chip_setup

格式：

```
    static int cfi_chip_setup(struct map_info *map,
            struct cfi_private *cfi)
```

注释：

无

功能：

设置cfi_private结构变量cfi的成员cfiq（cfi_ident结构）

说明：

1. 读取器件可擦除块区域个数
2. 分配并清零内存块
3. 调用cfi_read_query()读取CFI数据结构cfi_ident

参数：

map：芯片信息

cfi：被设置的cfi_private结构

返回：

成功：返回1

失败：返回0

调用：

　无
被调用：
　cfi_chip_probe()
源代码：

```
{
	int ofs_factor = cfi->interleave*cfi->device_type;
	__u32 base = 0;
	int num_erase_regions = cfi_read_query(map, base + (0x10 + 28)*ofs_factor);
	int i;

#ifdef DEBUG_CFI
	printk("Number of erase regions: %d\n", num_erase_regions);
#endif
	if (!num_erase_regions)
		return 0;

	cfi->cfiq = kmalloc(sizeof(struct cfi_ident) + num_erase_regions * 4, GFP_KERNEL);
	if (!cfi->cfiq) {
		printk(KERN_WARNING "%s: kmalloc failed for CFI ident structure\n", map->name);
		return 0;
	}

	memset(cfi->cfiq,0,sizeof(struct cfi_ident));

	cfi->cfi_mode = 1;		//JEDEC仿真模式
	cfi->fast_prog=1;		/* CFI supports fast programming *///CFI支持Fast Program模式

	/* Read the CFI info structure */
	for (i=0; i<(sizeof(struct cfi_ident) + num_erase_regions * 4); i++) {
		((unsigned char *)cfi->cfiq)[i] = cfi_read_query(map,base + (0x10 + i)*ofs_factor);
	}

	/* Do any necessary byteswapping */
	cfi->cfiq->P_ID = le16_to_cpu(cfi->cfiq->P_ID);

	cfi->cfiq->P_ADR = le16_to_cpu(cfi->cfiq->P_ADR);
	cfi->cfiq->A_ID = le16_to_cpu(cfi->cfiq->A_ID);
	cfi->cfiq->A_ADR = le16_to_cpu(cfi->cfiq->A_ADR);
	cfi->cfiq->InterfaceDesc = le16_to_cpu(cfi->cfiq->InterfaceDesc);
	cfi->cfiq->MaxBufWriteSize = le16_to_cpu(cfi->cfiq->MaxBufWriteSize);

#ifdef DEBUG_CFI
	/* Dump the information therein */
	print_cfi_ident(cfi->cfiq);
#endif
```

```
	for (i=0; i<cfi->cfiq->NumEraseRegions; i++) {
		cfi->cfiq->EraseRegionInfo[i] = le32_to_cpu(cfi->cfiq->EraseRegionInfo[i]);

#ifdef DEBUG_CFI
		printk("  Erase Region #%d: BlockSize 0x%4.4X bytes, %d blocks\n",
			i, (cfi->cfiq->EraseRegionInfo[i] >> 8) & ~0xff,
			(cfi->cfiq->EraseRegionInfo[i] & 0xffff) + 1);
#endif
	}
	/* Put it back into Read Mode */
	cfi_send_gen_cmd(0xF0, 0, base, map, cfi, cfi->device_type, NULL);

	return 1;
}
```

jedec_probe.c

“jedec_probe”型芯片的探测程序，主要由jedec_probe()、jedec_probe_chip()、cfi_jedec_setup()、jedec_probe_init()和jedec_probe_exit()这几个函数组成。
jedec_probe()是“jedec_probe”类型芯片的探测程序，它调用通用探测程序**mtd_do_chip_probe()**，并将**jedec_chip_probe**作为参数传递给mtd_do_chip_probe()，mtd_do_chip_probe()将间接调用jedec_chip_probe的成员函数**jedec_probe_chip()**。jedec_probe()注册在“jedec_probe”芯片的驱动器**jedec_chipdrv**中。
jedec_probe_chip()调用**cfi_jedec_setup()**初始化cfi_private结构，cfi_jedec_setup()根据
jedec_probe_init()和**jedec_probe_exit()**是“cfi_prbe”型芯片驱动器的注册程序和清除程序

amd_flash_info

```
struct amd_flash_info {
	const __u16 mfr_id;
	const __u16 dev_id;
	const char *name;
	const int DevSize;
	const int InterfaceDesc;
	const int NumEraseRegions;
	const int CmdSet;
	const ulong regions[4];
};
```

AMD Flash芯片的信息结构

jedec_table

static const struct amd_flash_info jedec_table[] = {}
包含各种jedec_probe类型芯片信息的结构

jedec_chipdrv

```
static struct mtd_chip_driver jedec_chipdrv = {
	probe: jedec_probe,		芯片的探测程序
	name: "jedec_probe",	芯片名称
	module: THIS_MODULE
};
```

“jedec_probe”型芯片的驱动器

jedec_probe_init

格式：
	int __init jedec_probe_init(void)
注释：
	无
功能：
	初始化“jedec_probe”类型的MTD芯片
说明：
	调用register_mtd_chip_driver()将jedec_chipdrv加入MTD驱动器列表chip_drvs_list
参数：
	无
返回：
	0
调用：
	register_mtd_chip_driver()
被调用：
	__init
	module_init
源代码：

```
{
	register_mtd_chip_driver(&jedec_chipdrv);
	return 0;
}
```

jedec_probe_exit

格式：
	static void __exit jedec_probe_exit(void)

注释：

无

功能：

清除“jedec_probe”MTD芯片驱动

说明：

调用unregister_mtd_chip_driver从MTD芯片驱动器列表chip_drvs_list中删除jedec_chipdrv

参数：

无

返回：

无

调用：

unregister_chip_driver

被调用：

__exit

module_exit

源代码：

```
{
	unregister_mtd_chip_driver(&jedec_chipdrv);
}
```

jedec_probe

格式：

struct mtd_info *jedec_probe(struct map_info *map)

注释：

无

功能：

“jedec_probe”型MTD芯片的探测程序

说明：

调用通用探测程序mtd_do_chip_probe()，并将jedec_chip_probe作为参数传递给mtd_do_chp_probe()

参数：

map：芯片信息

返回：

MTD设备信息结构mtd_info

调用：

mtd_do_chip_probe()

被调用：

注册在jedec_chipdrv中，根据芯片类型被do_map_probe()调用

源代码：

```
{
	/*
	 * Just use the generic probe stuff to call our CFI-specific
	 * chip_probe routine in all the possible permutations, etc.
	 */
	return mtd_do_chip_probe(map, &jedec_chip_probe);
}
```

jedec_probe_chip

```
static struct chip_probe jedec_chip_probe = {
	name: "JEDEC",
	probe_chip: jedec_probe_chip
};
```

jedec_probe传递给通用探测程序mtd_do_chip_probe的参数

jedec_probe_chip

格式：

```
	static int jedec_probe_chip(struct map_info *map, __u32 base,
			struct flchip *chips, struct cfi_private *cfi)
```

注释：

　　无

功能：

　　“jedec_probe”类型MTD芯片驱动程序

说明：

　　主要工作是设置传进的cfi_private型参数cfi。

参数：

　　FIXME

返回：

　　成功：返回1

　　失败：返回0

调用：

　　cfi_jedec_stup()

被调用：

　　注册在jedec_chip_probe中

源代码：

　　无

cfi_jedec_setup

格式：

```
static int cfi_jedec_setup(struct cfi_private *p_cfi, int index)
```

注释：

　　无

功能：

　　根据index从jedec_table中选择对应的信息赋给p_cfi

说明：

　　无

参数：

　　p_cfi：cfi_private结构的CFI私有信息

　　index：芯片在jedec_table中的索引

返回：

　　成功：返回1

　　失败：返回0

调用：

　　无

被调用：

　　jedec_probe_chip()

源代码：

　　无

gen_probe.c

通用芯片探测程序，由mtd_do_chip_probe()、genprobe_ident_chips()、genprobe_new_chip()、check_cmd_set()和cfi_cmdset_unknown()组成

cfi_probe()或jedec_probe()调用**mtd_do_chip_probe()**，mtd_do_chip_probe()调用**genprobe_ident_chips()**，genprobe_ident_chips()调用**genprobe_new_chip()**，genprobe_new_chip()则调用mtd_do_chip_probe()的参数chip_probe->probe_chip()。

mtd_do_chip_probe

格式：

　　struct mtd_info *mtd_do_chip_probe(struct map_info *map, struct chip_probe *cp)

注释：

　　无

功能：

　　根据MTD芯片的信息map和参数cp返回MTD设备mtd_info结构

说明：

　　获得cfi_private结构

　　调用check_cmd_set()返回mtd_info结构

参数：

　　map：MTD芯片信息

　　cp：由cfi_probe()或jedec_probe()传进来的信息

返回：

　　MTD设备信息

调用：

　　genprobe_ident_chips()

check_cmd_set()

被调用：

cfi_probe()

jedec_probe()

源代码：

```
{
	struct mtd_info *mtd = NULL;
	struct cfi_private *cfi;

	/* First probe the map to see if we have CFI stuff there. */
	cfi = genprobe_ident_chips(map, cp);

	if (!cfi)
		return NULL;

	map->fldrv_priv = cfi;
	/* OK we liked it. Now find a driver for the command set it talks */

	mtd = check_cmd_set(map, 1); /* First the primary cmdset */
	if (!mtd)
		mtd = check_cmd_set(map, 0); /* Then the secondary */

	if (mtd)
		return mtd;

	printk(KERN_WARNING"cfi_probe: No supported Vendor Command Set found\n");

	kfree(cfi->cfiq);
	kfree(cfi);
	map->fldrv_priv = NULL;
	return NULL;
}
```

genprobe_ident_chips

格式：

```
	struct cfi_private *genprobe_ident_chips(struct map_info *map, struct chip_probe *cp)
```

注释：

无

功能：

生成cfi_private结构并返回

说明：

1. 调用genprobe_new_chip搜索第一块flash芯片

2. 设置cfi.chipshift，并将cfi.numchips设置为1
3. 循环调用cp->probe_chip()搜索所有的flash芯片

参数：

map：芯片信息

cp：chip_probe结构参数

返回：

cfi_private结构

调用：

genprobe_new_chip()

被调用：

mtd_do_chip_probe()

源代码：

无

genprobe_new_chip

格式：

```
static int genprobe_new_chip(struct map_info *map, struct chip_probe *cp,
                struct cfi_private *cfi)
```

注释：

无

功能：

设置cfi

说明：

调用cp->probe_chip()

参数：

map：芯片信息

cp：chip_probe结构参数

cfi：被设置的cfi_private结构

返回：

成功：返回1

失败：返回0

调用：

cfi-> probe_chip()

被调用：

genprobe_ident_chips()

源代码：

无

check_cmd_set

格式：

```
static struct mtd_info *check_cmd_set(struct map_info *map, int primary)
```

注释：

　　无

功能：

　　根据map_info中的厂商信息调用不同的命令集

说明：

　　根据厂商类型调用两个不同的命令集，cfi_cmdset_0001()和cfi_cmdset_0002()，如果符合的类型没有则调用cfi_cmdset_unkown

参数：

　　map：MTD芯片信息

　　primary：如果取1，则芯片厂商信息为map中的primary，否则为auxillary

返回：

　　MTD设备信息mtd_info

调用：

　　cfi_cmdset_0001()

　　cfi_cmdset_0002()

　　cfi_cmdset_unknown()

被调用：

　　mtd_do_chip_probe()

源代码：

```
{
	struct cfi_private *cfi = map->fldrv_priv;
	__u16 type = primary?cfi->cfiq->P_ID:cfi->cfiq->A_ID;

	if (type == P_ID_NONE || type == P_ID_RESERVED)
		return NULL;

	switch(type){
		/* Urgh. Ifdefs. The version with weak symbols was
		 * _much_ nicer. Shame it didn't seem to work on
		 * anything but x86, really.
		 * But we can't rely in inter_module_get() because
		 * that'd mean we depend on link order.
		 */
#ifdef CONFIG_MTD_CFI_INTELEXT
	case 0x0001:
	case 0x0003:
		return cfi_cmdset_0001(map, primary);
#endif
#ifdef CONFIG_MTD_CFI_AMDSTD
	case 0x0002:
		return cfi_cmdset_0002(map, primary);
#endif
	}

	return cfi_cmdset_unknown(map, primary);
}
```

cfi_cmdset_unkown

格式：

```
    static inline struct mtd_info *cfi_cmdset_unknown(struct map_info *map, int primary)
```

注释：

无

功能：

调用未知类型的命令集

说明：

如果存在返回mtd_info，如果不存在返回NULL

参数：

map：MTD芯片信息

primary：FIXME

返回：

存在命令集：返回mtd_info

否则：返回NULL

调用：

无

被调用：

check_cmd_set()

源代码：

```
{
    struct cfi_private *cfi = map->fldrv_priv;
    __u16 type = primary?cfi->cfiq->P_ID:cfi->cfiq->A_ID;
#if defined(CONFIG_MODULES) && defined(HAVE_INTER_MODULE)
    char probename[32];
    cfi_cmdset_fn_t *probe_function;

    sprintf(probename, "cfi_cmdset_%4.4X", type);

    probe_function = inter_module_get_request(probename, probename);

    if (probe_function) {
        struct mtd_info *mtd;

        mtd = (*probe_function)(map, primary);
        /* If it was happy, it'll have increased its own use count */
        inter_module_put(probename);
        return mtd;
    }
#endif
    printk(KERN_NOTICE "Support for command set %04X not present\n",
        type);
    return NULL;
}
```

cfi_cmdset_0002.c

CFI的AMD标准命令集，**cfi_cmdset_0002()**调用**cfi_amdstd_setup()**建立mtd_info，**cfi_amdstd_read()**、**cfi_amdstd_write()**、**cfi_amdstd_sync()**、**cfi_amdstd_resume()**、**cfi_amdstd_erase_onesize()**（或**cfi_amdstd_erase_varsize()**）、**cfi_amdstd_suspend()**被注册在mtd_info中。其中cfi_amdstd_read()调用**do_read_onechip()**，cfi_amdstd_write()调用**do_write_oneword()**，cfi_amdstd_erase_onesize()（或cfi_amdstd_erase_varsize()）调用**do_erase_oneblock()**。
cfi_amdstd_chipdrv是AMD标准命令集的驱动，它被连接在map->fldrv上，其中的destroy函数指针指向**cfi_amdstd_destroy()**。
    cfi_amdstd_init()和**cfi_amdstd_exit()**负责AMD标准命令集的初始化和清除工作。

cfi_amdstd_chipdrv

AMD标准命令集的驱动，不被加入chip_drvs_list，而被注册在map->fldrv上

```
static struct mtd_chip_driver cfi_amdstd_chipdrv = {
    probe: NULL, /* Not usable directly */
    destroy: cfi_amdstd_destroy,
    name: "cfi_cmdset_0002",
    module: THIS_MODULE
};
```

cfi_cmdset_0002

格式：

```
    struct mtd_info *cfi_cmdset_0002(struct map_info *map, int primary)
```

注释：
    无
功能：
    建立mtd_info结构
说明：
    进行一些初始化设置后调用cfi_amdstd_setup()
参数：
    map：芯片信息
    primary：等于1：primary，等于0：alternate
返回：
    mtd_info结构
调用：
    cfi_amdstd_setup()

被调用：

check_cmd_set()

源代码：

```
{
	struct cfi_private *cfi = map->fldrv_priv;
	unsigned char bootloc;
	int ofs_factor = cfi->interleave * cfi->device_type;		//offset factor
	int i;
	__u8 major, minor;
	__u32 base = cfi->chips[0].start;

	if (cfi->cfi_mode==1){
		__u16 adr = primary?cfi->cfiq->P_ADR:cfi->cfiq->A_ADR;

		cfi_send_gen_cmd(0x98, 0x55, base, map, cfi, cfi->device_type, NULL);
							//发送查询命令
		major = cfi_read_query(map, base + (adr+3)*ofs_factor);
		minor = cfi_read_query(map, base + (adr+4)*ofs_factor);

		printk(KERN_NOTICE " Amd/Fujitsu Extended Query Table v%c.%c at 0x%4.4X\n",
			major, minor, adr);
				cfi_send_gen_cmd(0xf0, 0x55, base, map, cfi, cfi->device_type, NULL);
							//发送reset命令使 Flash进入read模式
		cfi_send_gen_cmd(0xaa, 0x555, base, map, cfi, cfi->device_type, NULL);
		cfi_send_gen_cmd(0x55, 0x2aa, base, map, cfi, cfi->device_type, NULL);
		cfi_send_gen_cmd(0x90, 0x555, base, map, cfi, cfi->device_type, NULL);
							//发送Autoselect命令
		cfi->mfr = cfi_read_query(map, base);
		cfi->id = cfi_read_query(map, base + ofs_factor);

		/* Wheee. Bring me the head of someone at AMD. */
#ifdef AMD_BOOTLOC_BUG
		if (((major << 8) | minor) < 0x3131) {
			/* CFI version 1.0 => don't trust bootloc */
			if (cfi->id & 0x80) {
				printk(KERN_WARNING "%s: JEDEC Device ID is 0x%02X. Assuming broken CFI table.
\n", map->name, cfi->id);
				bootloc = 3;	/* top boot */
			} else {
				bootloc = 2;	/* bottom boot */
			}
		} else
#endif
			{
				cfi_send_gen_cmd(0x98, 0x55, base, map, cfi, cfi->device_type, NULL);
				bootloc = cfi_read_query(map, base + (adr+15)*ofs_factor);
```

```
                }
            if (bootloc == 3 && cfi->cfiq->NumEraseRegions > 1) {
                printk(KERN_WARNING "%s: Swapping erase regions for broken CFI table.\n", map->name);

                for (i=0; i<cfi->cfiq->NumEraseRegions / 2; i++) {
                    int j = (cfi->cfiq->NumEraseRegions-1)-i;
                    __u32 swap;

                    swap = cfi->cfiq->EraseRegionInfo[i];
                    cfi->cfiq->EraseRegionInfo[i] = cfi->cfiq->EraseRegionInfo[j];
                    cfi->cfiq->EraseRegionInfo[j] = swap;
                }
            }
            switch (cfi->device_type) {        //设置unlock地址的值
            case CFI_DEVICETYPE_X8:
                cfi->addr_unlock1 = 0x555;
                cfi->addr_unlock2 = 0x2aa;
                break;
            case CFI_DEVICETYPE_X16:
                cfi->addr_unlock1 = 0xaaa;
                if (map->buswidth == cfi->interleave) {
                    /* X16 chip(s) in X8 mode */
                    cfi->addr_unlock2 = 0x555;
                } else {
                    cfi->addr_unlock2 = 0x554;
                }
                break;
            case CFI_DEVICETYPE_X32:
                cfi->addr_unlock1 = 0x1555;
                cfi->addr_unlock2 = 0xaaa;
                break;
            default:
                printk(KERN_NOTICE "Eep. Unknown cfi_cmdset_0002 device type %d\n", cfi->device_type);
                return NULL;
            }
        } /* CFI mode */

        for (i=0; i< cfi->numchips; i++) {
            cfi->chips[i].word_write_time = 1<<cfi->cfiq->WordWriteTimeoutTyp;
            cfi->chips[i].buffer_write_time = 1<<cfi->cfiq->BufWriteTimeoutTyp;
            cfi->chips[i].erase_time = 1<<cfi->cfiq->BlockEraseTimeoutTyp;
        }

        map->fldrv = &cfi_amdstd_chipdrv;
        MOD_INC_USE_COUNT;
```

```
    cfi_send_gen_cmd(0xf0, 0x55, base, map, cfi, cfi->device_type, NULL);
    return cfi_amdstd_setup(map);          //调用cfi_amdstd_setup()
}
```

cfi_amdstd_setup

格式：

    static struct mtd_info *cfi_amdstd_setup(struct map_info *map)

注释：

无

功能：

构建mtd_info结构

说明：

1. 内存分配mtd_info并清零
2. mtd->priv=map，mtd->type=NORFLASH
3. 根据cfi->cfiq中的数据设置mtd->eraseregions
4. 设置mtd->erase，read，write，sync，suspend，resume
5. 返回mtd

参数：

map：MTD芯片信息

返回：

成功：返回mtd_info

失败：返回NULL

调用：

无

被调用：

cfi_cmdset_0002()

源代码：

```
{
    struct cfi_private *cfi = map->fldrv_priv;
    struct mtd_info *mtd;
    unsigned long devsize = (1<<cfi->cfiq->DevSize) * cfi->interleave;

    mtd = kmalloc(sizeof(*mtd), GFP_KERNEL);
    printk(KERN_NOTICE "number of %s chips: %d\n", (cfi->cfi_mode)?"CFI":"JEDEC",cfi->numchips);

    if (!mtd) {
      printk(KERN_WARNING "Failed to allocate memory for MTD device\n");
      kfree(cfi->cmdset_priv);
      return NULL;
    }

    memset(mtd, 0, sizeof(*mtd));
```

```
	mtd->priv = map;
	mtd->type = MTD_NORFLASH;
	/* Also select the correct geometry setup too */
	mtd->size = devsize * cfi->numchips;

	if (cfi->cfiq->NumEraseRegions == 1) {
		/* No need to muck about with multiple erase sizes */
		mtd->erasesize = ((cfi->cfiq->EraseRegionInfo[0] >> 8) & ~0xff) * cfi->interleave;
	} else {
		unsigned long offset = 0;
		int i,j;

		mtd->numeraseregions = cfi->cfiq->NumEraseRegions * cfi->numchips;
		mtd->eraseregions = kmalloc(sizeof(struct mtd_erase_region_info) * mtd->numeraseregions,
GFP_KERNEL);
		if (!mtd->eraseregions) {
			printk(KERN_WARNING "Failed to allocate memory for MTD erase region info\n");
			kfree(cfi->cmdset_priv);
			return NULL;
		}

		for (i=0; i<cfi->cfiq->NumEraseRegions; i++) {
			unsigned long ernum, ersize;
			ersize = ((cfi->cfiq->EraseRegionInfo[i] >> 8) & ~0xff) * cfi->interleave;
			ernum = (cfi->cfiq->EraseRegionInfo[i] & 0xffff) + 1;

			if (mtd->erasesize < ersize) {
				mtd->erasesize = ersize;
			}
			for (j=0; j<cfi->numchips; j++) {
				mtd->eraseregions[(j*cfi->cfiq->NumEraseRegions)+i].offset = (j*devsize)+offset;
				mtd->eraseregions[(j*cfi->cfiq->NumEraseRegions)+i].erasesize = ersize;
				mtd->eraseregions[(j*cfi->cfiq->NumEraseRegions)+i].numblocks = ernum;
			}
			offset += (ersize * ernum);
		}
		if (offset != devsize) {
			/* Argh */
			printk(KERN_WARNING "Sum of regions (%lx) != total size of set of interleaved chips (%lx)\n",
offset, devsize);
			kfree(mtd->eraseregions);
			kfree(cfi->cmdset_priv);
			return NULL;
		}
#if 0
		// debug
```

```
          for (i=0; i<mtd->numeraseregions;i++){
               printk("%d: offset=0x%x,size=0x%x,blocks=%d\n",
                    i,mtd->eraseregions[i].offset,
                    mtd->eraseregions[i].erasesize,
                    mtd->eraseregions[i].numblocks);
          }
#endif
     }

     switch (CFIDEV_BUSWIDTH)
     {
     case 1:
     case 2:
     case 4:
#if 1
          if (mtd->numeraseregions > 1)
               mtd->erase = cfi_amdstd_erase_varsize;
          else
#endif
               mtd->erase = cfi_amdstd_erase_onesize;
          mtd->read = cfi_amdstd_read;
          mtd->write = cfi_amdstd_write;
          break;

     default:
           printk(KERN_WARNING "Unsupported buswidth\n");
          kfree(mtd);
          kfree(cfi->cmdset_priv);
          return NULL;
          break;
     }
     mtd->sync = cfi_amdstd_sync;
     mtd->suspend = cfi_amdstd_suspend;
     mtd->resume = cfi_amdstd_resume;
     mtd->flags = MTD_CAP_NORFLASH;
     map->fldrv = &cfi_amdstd_chipdrv;
     mtd->name = map->name;
     MOD_INC_USE_COUNT;
     return mtd;
}
```

cfi_amdstd_erase_onesize

格式：

```
     static int cfi_amdstd_erase_onesize(struct mtd_info *mtd, struct erase_info *instr)
```

注释：

无

功能：

擦除（一个erase size region）

说明：

1. 安全检查
2. 选择正确的chip后，循环调用do_erase_oneblock()擦除单个块
3. 调用erase_info->callback函数

参数：

mtd：被擦除的MTD设备

instr：擦除命令，由设备层传入，包含擦除的地址和长度等信息

返回：

成功：返回0

失败：返回错误码

调用：

do_erase_oneblock()

被调用：

在cfi_amdstd_setup()中被注册在mtd_info->erase中

源代码：

```
{
	struct map_info *map = mtd->priv;
	struct cfi_private *cfi = map->fldrv_priv;
	unsigned long adr, len;
	int chipnum, ret = 0;

	if (instr->addr & (mtd->erasesize - 1))
		return -EINVAL;

	if (instr->len & (mtd->erasesize -1))
		return -EINVAL;

	if ((instr->len + instr->addr) > mtd->size)
		return -EINVAL;

	chipnum = instr->addr >> cfi->chipshift;
	adr = instr->addr - (chipnum << cfi->chipshift);
	len = instr->len;

	while(len) {
		ret = do_erase_oneblock(map, &cfi->chips[chipnum], adr);

		if (ret)
			return ret;

		adr += mtd->erasesize;
		len -= mtd->erasesize;
```

```
          if (adr >> cfi->chipshift) {
               adr = 0;
               chipnum++;

               if (chipnum >= cfi->numchips)
               break;
          }
     }

     instr->state = MTD_ERASE_DONE;
     if (instr->callback)
          instr->callback(instr);

     return 0;
}
```

do_erase_oneblock

格式：

static inline int do_erase_oneblock(struct map_info *map, struct flchip *chip, unsigned long adr)

注释：

无

功能：

使用AMD标准命令集擦除一个块

说明：

1. 如果状态不是FL_READY，等待
2. 发送命令序列擦除一个块

参数：

map：芯片信息
chip：被擦除块所在的芯片
adr：地址

返回：

成功：返回0
失败：返回错误码

调用：

无

被调用：

cfi_amdstd_erase_onesize()

源代码：

```
{
     unsigned int status;
     unsigned long timeo = jiffies + HZ;
```

```
	struct cfi_private *cfi = map->fldrv_priv;
	unsigned int rdy_mask;
	DECLARE_WAITQUEUE(wait, current);

 retry:
	cfi_spin_lock(chip->mutex);

	if (chip->state != FL_READY){
		set_current_state(TASK_UNINTERRUPTIBLE);
		add_wait_queue(&chip->wq, &wait);

		cfi_spin_unlock(chip->mutex);

		schedule();
		remove_wait_queue(&chip->wq, &wait);
#if 0
		if(signal_pending(current))
			return -EINTR;
#endif
		timeo = jiffies + HZ;

		goto retry;
	}

	chip->state = FL_ERASING;

	adr += chip->start;
	ENABLE_VPP(map);
	cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
	cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
	cfi_send_gen_cmd(0x80, cfi->addr_unlock1, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
	cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
	cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
	cfi_write(map, CMD(0x30), adr);		//擦除一个block的命令序列

	timeo = jiffies + (HZ*20);

	cfi_spin_unlock(chip->mutex);
	schedule_timeout(HZ);
	cfi_spin_lock(chip->mutex);

	rdy_mask = CMD(0x80);

	/* FIXME. Use a timer to check this, and return immediately. */
	/* Once the state machine's known to be working I'll do that */
	while ( ( (status = cfi_read(map,adr)) & rdy_mask ) != rdy_mask ) {
		static int z=0;

		if (chip->state != FL_ERASING) {
			/* Someone's suspended the erase. Sleep */
			set_current_state(TASK_UNINTERRUPTIBLE);
			add_wait_queue(&chip->wq, &wait);

			cfi_spin_unlock(chip->mutex);
			printk(KERN_DEBUG "erase suspended. Sleeping\n");

			schedule();
			remove_wait_queue(&chip->wq, &wait);
#if 0
			if (signal_pending(current))
				return -EINTR;
#endif
			timeo = jiffies + (HZ*2); /* FIXME */
			cfi_spin_lock(chip->mutex);
			continue;
		}

		/* OK Still waiting */
		if (time_after(jiffies, timeo)) {
			chip->state = FL_READY;
			cfi_spin_unlock(chip->mutex);
			printk(KERN_WARNING "waiting for erase to complete timed out.");
			DISABLE_VPP(map);
			return -EIO;
		}

		/* Latency issues. Drop the lock, wait a while and retry */
		cfi_spin_unlock(chip->mutex);

		z++;
		if ( 0 && !(z % 100 ))
			printk(KERN_WARNING "chip not ready yet after erase. looping\n");

		cfi_udelay(1);

		cfi_spin_lock(chip->mutex);
		continue;
	}

	/* Done and happy. */
	DISABLE_VPP(map);
```

```
    chip->state = FL_READY;
    wake_up(&chip->wq);
    cfi_spin_unlock(chip->mutex);
    return 0;
}
```

cfi_amdstd_read

格式：

```
     static int cfi_amdstd_read (struct mtd_info *mtd, loff_t from, size_t len, size_t *retlen, u_char *buf)
```

注释：
    无

功能：
    读取

说明：
    无

参数：
    mtd：MTD设备信息
    from：起始地址
    len：长度
    retlen：读取的长度
    buf：目标缓冲区

返回：
    成功：返回0
    失败：返回错误码

调用：
    do_read_onechip()

被调用：
    在cfi_amdstd_setup()中被注册给mtd->read()

源代码：

```
{
    struct map_info *map = mtd->priv;
    struct cfi_private *cfi = map->fldrv_priv;
    unsigned long ofs;
    int chipnum;
    int ret = 0;

    /* ofs: offset within the first chip that the first read should start */

    chipnum = (from >> cfi->chipshift);
    ofs = from - (chipnum <<  cfi->chipshift);


    *retlen = 0;
    while (len) {
        unsigned long thislen;

        if (chipnum >= cfi->numchips)
            break;

        if ((len + ofs -1) >> cfi->chipshift)
            thislen = (1<<cfi->chipshift) - ofs;
        else
            thislen = len;

        ret = do_read_onechip(map, &cfi->chips[chipnum], ofs, thislen, buf);
        if (ret)
            break;

        *retlen += thislen;
        len -= thislen;
        buf += thislen;

        ofs = 0;
        chipnum++;
    }
    return ret;
}
```

do_read_onechip

格式：

    static inline int do_read_onechip(struct map_info *map, struct flchip *chip, loff_t adr, size_t len, u_char *buf)

注释：

无

功能：

从flash中读

说明：

调用map->copy_from()读

参数：

map：芯片信息

chip：被读数据所在芯片

adr：起始地址

len：长度

buf：目标缓冲区

返回：

成功：返回0

失败：返回错误码

调用：

map->copy_from()

被调用：

cfi_amdstd_read()

源代码：

```
{
	DECLARE_WAITQUEUE(wait, current);
	unsigned long timeo = jiffies + HZ;

 retry:
	cfi_spin_lock(chip->mutex);

	if (chip->state != FL_READY){
#if 0
		printk(KERN_DEBUG "Waiting for chip to read, status = %d\n", chip->state);
#endif
		set_current_state(TASK_UNINTERRUPTIBLE);
		add_wait_queue(&chip->wq, &wait);

		cfi_spin_unlock(chip->mutex);

		schedule();
		remove_wait_queue(&chip->wq, &wait);
#if 0
		if(signal_pending(current))
			return -EINTR;
#endif
		timeo = jiffies + HZ;

		goto retry;
	}

	adr += chip->start;

	chip->state = FL_READY;

	map->copy_from(map, buf, adr, len);

	wake_up(&chip->wq);
	cfi_spin_unlock(chip->mutex);

	return 0;
}
```

OOB: out of band，某些内存技术支持out-of-band数据——例如，NAND flash每512字节的

cfi_amdstd_write

格式：

static int cfi_amdstd_write (struct mtd_info *mtd, loff_t to , size_t len, size_t *retlen, const u_char *buf)

注释：

无

功能：

向flash中写

说明：

1. 如果起始地址没有buswidth对齐，写第一个字节
2. 以buswidth为单位写
3. 如果最后一个字节没够buswidth对齐，写最后一个字节

参数：

mtd：被写的MTD设备信息
to：目标地址
len：长度
retlen：成功写入的长度
buf：源缓冲区

返回：

成功：返回0
失败：返回错误码

调用：

do_write_oneword()

被调用：

在cfi_amdstd_setup中被注册为mtd->write

源代码：

```
{
	struct map_info *map = mtd->priv;
	struct cfi_private *cfi = map->fldrv_priv;
	int ret = 0;
	int chipnum;
	unsigned long ofs, chipstart;

	*retlen = 0;
	if (!len)
		return 0;

	chipnum = to >> cfi->chipshift;
	ofs = to  - (chipnum << cfi->chipshift);
	chipstart = cfi->chips[chipnum].start;

	/* If it's not bus-aligned, do the first byte write */
	if (ofs & (CFIDEV_BUSWIDTH-1)) {		//如果没有buswidth对齐，写第一个字节
		unsigned long bus_ofs = ofs & ~(CFIDEV_BUSWIDTH-1);
		int i = ofs - bus_ofs;
		int n = 0;
```

```
        u_char tmp_buf[4];
        __u32 datum;

        map->copy_from(map, tmp_buf, bus_ofs + cfi->chips[chipnum].start,
CFIDEV_BUSWIDTH);          //首先从flash中读出第一个字节所在的字（buswidth大小）
                           //到tmp_buf
        while (len && i < CFIDEV_BUSWIDTH)
            tmp_buf[i++] = buf[n++], len--;   //将源缓冲区中没对齐的部分覆盖tmp_buf

        if (cfi_buswidth_is_2()) {
            datum = *(__u16*)tmp_buf;
        } else if (cfi_buswidth_is_4()) {
            datum = *(__u32*)tmp_buf;
        } else {
            return -EINVAL;  /* should never happen, but be safe */
        }

        ret = do_write_oneword(map, &cfi->chips[chipnum],
                bus_ofs, datum, 0);           //调用do_write_oneword()将该字写入flash
        if (ret)
            return ret;

        ofs += n;
        buf += n;
        (*retlen) += n;

        if (ofs >> cfi->chipshift) {          //如果该地址在下一个芯片中
            chipnum ++;
            ofs = 0;
            if (chipnum == cfi->numchips)          //chipnum芯片号越界
                return 0;
        }
    }

    /* Go into unlock bypass mode */       //使用CFI命令进入Fast Program模式
    cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chipstart, map, cfi, CFI_DEVICETYPE_X8, NULL);
    cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chipstart, map, cfi, CFI_DEVICETYPE_X8, NULL);
    cfi_send_gen_cmd(0x20, cfi->addr_unlock1, chipstart, map, cfi, CFI_DEVICETYPE_X8, NULL);

    /* We are now aligned, write as much as possible */   //此时已buswidth对齐
    while(len >= CFIDEV_BUSWIDTH) {
        __u32 datum;

        if (cfi_buswidth_is_1()) {
            datum = *(__u8*)buf;
        } else if (cfi_buswidth_is_2()) {
```

```
		datum = *(__u16*)buf;
	} else if (cfi_buswidth_is_4()) {
		datum = *(__u32*)buf;
	} else {
		return -EINVAL;
	}
	ret = do_write_oneword(map, &cfi->chips[chipnum],		//以字（buswidth大小）
				ofs, datum, cfi->fast_prog);		//为单位向flash中写
	if (ret) {		//如果出错
		if (cfi->fast_prog){		//而且允许Fast Program模式
			/* Get out of unlock bypass mode */	//退出Fast Program模式
			cfi_send_gen_cmd(0x90, 0, chipstart, map, cfi, cfi->device_type, NULL);
			cfi_send_gen_cmd(0x00, 0, chipstart, map, cfi, cfi->device_type, NULL);
		}
		return ret;
	}

	ofs += CFIDEV_BUSWIDTH;
	buf += CFIDEV_BUSWIDTH;
	(*retlen) += CFIDEV_BUSWIDTH;
	len -= CFIDEV_BUSWIDTH;

	if (ofs >> cfi->chipshift) {		//如果地址进入下一个芯片
		if (cfi->fast_prog){			//而且允许Fast Program模式
			/* Get out of unlock bypass mode */	//则退出Fast Program模式
			cfi_send_gen_cmd(0x90, 0, chipstart, map, cfi, cfi->device_type, NULL);
			cfi_send_gen_cmd(0x00, 0, chipstart, map, cfi, cfi->device_type, NULL);
		}

		chipnum ++;		//芯片号自增
		ofs = 0;		//芯片偏移置零
		if (chipnum == cfi->numchips)
			return 0;
		chipstart = cfi->chips[chipnum].start;
		if (cfi->fast_prog){		//进入Fast Program 模式
			/* Go into unlock bypass mode for next set of chips */
			cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chipstart, map, cfi, CFI_DEVICETYPE_X8,
NULL);
			cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chipstart, map, cfi, CFI_DEVICETYPE_X8,
NULL);
			cfi_send_gen_cmd(0x20, cfi->addr_unlock1, chipstart, map, cfi, CFI_DEVICETYPE_X8,
NULL);
		}
	}
}
```

```
	if (cfi->fast_prog){		//如果允许Fast Program模式
		/* Get out of unlock bypass mode */	//退出Fast Program模式
		cfi_send_gen_cmd(0x90, 0, chipstart, map, cfi, cfi->device_type, NULL);
		cfi_send_gen_cmd(0x00, 0, chipstart, map, cfi, cfi->device_type, NULL);
	}

	if (len & (CFIDEV_BUSWIDTH-1)) {	//如果结束地址没buswidth对齐
		int i = 0, n = 0;
		u_char tmp_buf[4];
		__u32 datum;

		map->copy_from(map, tmp_buf, ofs + cfi->chips[chipnum].start, CFIDEV_BUSWIDTH);
		while (len--)
			tmp_buf[i++] = buf[n++];

		if (cfi_buswidth_is_2()) {
			datum = *(__u16*)tmp_buf;
		} else if (cfi_buswidth_is_4()) {
			datum = *(__u32*)tmp_buf;
		} else {
			return -EINVAL;  /* should never happen, but be safe */
		}

		ret = do_write_oneword(map, &cfi->chips[chipnum],
				ofs, datum, 0);
		if (ret)
			return ret;

		(*retlen) += n;
	}

	return 0;
}
```

do_write_oneword

格式：

```
	static int do_write_oneword(struct map_info *map, struct flchip *chip, unsigned long adr, __u32 datum, int fast)
```

注释：

	无

功能：

向flash中写入一个字（buswidth个字节）

说明：

1. 调用CFI命令和cfi_write()向flash中写
2. 调用cfi_read()验证

参数：

map：芯片信息
chip：被写入的芯片
adr：地址
datum：被写入的字
fast：是否为Fast Program模式

返回：

成功：返回0
失败：返回错误码

调用：

cfi_write()

被调用：

cfi_amdstd_write()

源代码：

```
{
	unsigned long timeo = jiffies + HZ;
	unsigned int Last[4];			//验证数据
	unsigned long Count = 0;
	struct cfi_private *cfi = map->fldrv_priv;
	DECLARE_WAITQUEUE(wait, current);
	int ret = 0;

 retry:
	cfi_spin_lock(chip->mutex);

	if (chip->state != FL_READY){		//如果被写入的芯片的状态不是FL_READY，等待
#if 0
		printk(KERN_DEBUG "Waiting for chip to write, status = %d\n", chip->state);
#endif
		set_current_state(TASK_UNINTERRUPTIBLE);
		add_wait_queue(&chip->wq, &wait);

		cfi_spin_unlock(chip->mutex);

		schedule();
		remove_wait_queue(&chip->wq, &wait);
#if 0
		printk(KERN_DEBUG "Wake up to write:\n");
		if(signal_pending(current))
			return -EINTR;
#endif
		timeo = jiffies + HZ;
```

```
        goto retry;
    }

    chip->state = FL_WRITING;

    adr += chip->start;
    ENABLE_VPP(map);
    if (fast) { /* Unlock bypass */          //如果允许Fast Program模式
        cfi_send_gen_cmd(0xA0, 0, chip->start, map, cfi, cfi->device_type, NULL);
    }         //进入Fast Program模式
    else {     //否则进入Program模式
         cfi_send_gen_cmd(0xAA, cfi->addr_unlock1, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
         cfi_send_gen_cmd(0x55, cfi->addr_unlock2, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
         cfi_send_gen_cmd(0xA0, cfi->addr_unlock1, chip->start, map, cfi, CFI_DEVICETYPE_X8, NULL);
    }

    cfi_write(map, datum, adr);        //向flash中写

    cfi_spin_unlock(chip->mutex);
    cfi_udelay(chip->word_write_time);          //延迟等待
    cfi_spin_lock(chip->mutex);
//下面是验证过程
    Last[0] = cfi_read(map, adr);          //从刚才写入的位置读出数据
    //    printk("Last[0] is %x\n", Last[0]);
    Last[1] = cfi_read(map, adr);
    //    printk("Last[1] is %x\n", Last[1]);
    Last[2] = cfi_read(map, adr);
    //    printk("Last[2] is %x\n", Last[2]);

    for (Count = 3; Last[(Count - 1) % 4] != Last[(Count - 2) % 4] && Count < 10000; Count++){         //反复
读，直到连续两次读出的数据相同
        cfi_spin_unlock(chip->mutex);
        cfi_udelay(10);
        cfi_spin_lock(chip->mutex);

         Last[Count % 4] = cfi_read(map, adr);
        //          printk("Last[%d%%4] is %x\n", Count, Last[Count%4]);
    }

    if (Last[(Count - 1) % 4] != datum){    //如果读出的数据和写入的数据不同，出错返回
        printk(KERN_WARNING "Last[%ld] is %x, datum is %x\n",(Count - 1) % 4,Last[(Count - 1) % 4],
datum);
         cfi_send_gen_cmd(0xF0, 0, chip->start, map, cfi, cfi->device_type, NULL);
        DISABLE_VPP(map);
        ret = -EIO;
    }
    DISABLE_VPP(map);
    chip->state = FL_READY;
    wake_up(&chip->wq);
    cfi_spin_unlock(chip->mutex);

    return ret;
}
```

cfi_amdstd_sync

N/A

cfi_amdstd_suspend

N/A

cfi_amdstd_resume

N/A

cfi_amdstd_destroy

N/A

cfi_amdstd_erase_varsize

N/A

cfi_amdstd_init

格式：

```
    int __init cfi_amdstd_init(void)
```

注释：

    无

功能：

    初始化AMD标准命令集

说明：

    无

参数：

    无

返回：

    0

调用：

    inter_module_register()（该函数被定义为空）

被调用：

    __init
    module_init

源代码：

```
{
    inter_module_register(im_name, THIS_MODULE, &cfi_cmdset_0002);
    return 0;
}
```

cfi_amdstd_exit

格式：

    static void __exit cfi_amdstd_exit(void)

注释：

    无

功能：

    清除AMD标准命令集

说明：

    无

参数：

    无

返回：

    无

调用：

    inter_module_unregister()（该函数被定义为空）

被调用：

    __exit
    module_exit

源代码：

```
{
    inter_module_unregister(im_name);
}
```

cfi_cmdset_0001.c

N/A

/drivers/mtd/maps子目录

此目录下的每一个文件为一个具体的MTD原始设备的相关信息，包括该MTD原始设备的起始物理地址、大小、分区情况、读写函数、初始化和清除程序。每一个MTD的驱动工程师的代码都应该写成一个文件放在这里，下面我们以your-flash.c为例解释一下

your-flash.c

假设这个MTD原始设备是一个NOR型的Flash，起始物理地址是0x1000000，大小为16M，总线宽度为16bits

WINDOW_ADDR

MTD原始设备的起始地址（PA）
#define WINDOW_ADDR 0x01000000

WINDOW_SIZE

MTD原始设备的大小
#define WINDOW_SIZE 0x02000000
16M

BUSWIDTH

总线宽度
#define BUSWIDTH 2
16bits

mymtd

由do_mtd_probe搜索出的MTD设备主分区
static struct mtd_info *mymtd;

your_read8

格式：
static __u8 your_read8(struct map_info *map,unsigned long ofs)
注释：
　　无
功能：
　　读Flash
说明：
　　NOR Flash可以和内存一样读写，但是需要先进行一些设置，然后调用此函数进行读操作（写函数

相同）
参数：
　　略
返回：
　　略
调用：
　　略
被调用：
　　注册进your_map
源代码：

```
{
	return __raw_readb(map->map_priv_1 + ofs);
}
```

your_read16

your_read32

your_copy_from

your_write8

your_write16

your_write32

your_copy_to

格式：
　　略
注释：
　　无
功能：
　　略
说明：
　　略
参数：
　　略
返回：
　　略
调用：
　　略
被调用：

注册进your_map

源代码：

略

your_set_vpp

格式：

```
static void your_set_vpp(struct map_info *map,int set)
```

注释：

无

功能：

设置flash的读写状态

说明：

向寄存器SOME_REGISTER写值，打开或去掉flash的写保护（Write Protection）

参数：

map：flash芯片信息

set：为1设置为写状态，为0设置为读状态

返回：

无

调用：

__raw_writel()

__raw_readl()

被调用：

注册进your_map

源代码：

```
{
    if(set)                  //Disable Write Protection
        __raw_writel(__raw_readl(SOME_REGISTER)|0x00000001, SOME_REGISTER)
    else
        __raw_writel(__raw_readl(SOMEREGISTER)&0xfffffffe, SOME_REGISTER);
}
```

your_partition

假设你的板子flash的分区结构如下，16MB大小的flash被分为3个区：

0—8K：“ Your Bootloarder ”，用于装载Bootloader；

8K—8M：“ Your Kernel ”，用于放置内核映像

8M—16M：“ Your Rootfs ”，用于存放根文件系统（JFFS或JFFS2）的映像,

```
static struct mtd_partition your_partition[]={
   {
      name: "Your Bootloader",
      offset: 0,
        size:   0x20000,
        mask_flags: MTD_WRITEABLE,  /* force read-only *///只读
    },
    {
       name:"Your Kernel",
       offset:0x20000,
       size:0x7e0000,
       mask_flags:  MTD_WRITEABLE,  /* force read-only *///只读
    },
    {
        name: "Your Rootfs",
        offset: 0x800000,
        size:0x800000,
    }
};
```

NUM_PARTITIONS

分区的个数（3）

```
#define NUM_PARTITIONS (sizeof(your_partition)/sizeof(your_partition[0]))
```

your_map

```
struct map_info your_map = {
      name: "Your Flash map",
      size: WINDOW_SIZE,
      buswidth: BUSWIDTH,
      read8: your_read8,
      read16: your_read16,
      read32: your_read32,
      copy_from: your_copy_from,
      write8: your_write8,
      write16: your_write16,
      write32: your_write32,
      copy_to: your_copy_to,
      set_vpp:your_set_vpp,
};
```

init_yourflash

格式：

static int __init init_yourflash(void)

注释：

无

功能：

初始化flash

说明：

1. 调用do_map_probe()搜索MTD设备并将其赋给mymtd,
2. 调用add_mtd_partitions()将your_partiton的各个分区加入mtd_table

参数：

无

返回：

成功：返回0

失败：返回错误码

调用：

do_map_probe()搜索MTD设备

add_mtd_partitions()将分区加入mtd_table

被调用：

__init

module_init

源代码：

```
{
    printk(KERN_NOTICE "Your flash mapping:size %x at %x\n", WINDOW_SIZE, WINDOW_ADDR);
    your_map.map_priv_1 = (unsigned long)ioremap(WINDOW_ADDR, WINDOW_SIZE);
    if (!your_map.map_priv_1) {
        return -EIO;
    }

    mymtd = do_map_probe("cfi_probe", &your_map);
    if (mymtd) {
        mymtd->module = THIS_MODULE;
        add_mtd_partitions( mymtd, your_partition, NUM_PARTITIONS );

        //add_mtd_device(mymtd);

        return 0;
    }

    iounmap((void *)your_map.map_priv_1);
    return -ENXIO;

}
```

cleanup_yourflash

格式：

```
    mod_exit_t cleanup_yourflash(void)
```

注释：

　　无

功能：

　　清除flash

说明：

　　调用del_mtd_partitions删除mtd_table中的MTD设备,

参数：

　　无

返回：

　　无

调用：

　　del_mtd_partitions()删除mtd_table中的MTD设备

　　map_destroy()

被调用：

　　__exit

　　module_exit

源代码：

```
{
    if (mymtd) {
        del_mtd_partitions(mymtd);
        map_destroy(mymtd);
    }
    if (your_map.map_priv_1) {
        iounmap((void *)your_map.map_priv_1);
        your_map.map_priv_1 = 0;
    }
}
```