LINUX

业界权威机构和专家强力推荐
多年培训、研发经验的总结

设备驱动开发详解

华清远见嵌入式培训中心 宋宝华 编著

Linux

附源代码光盘

人民邮电出版社
POSTS & TELECOM PRESS

第 19 章　Flash 设备驱动

Flash 在嵌入式系统中是必不可少的，它是 BootLoader、Linux 内核和文件系统的最佳载体。在 Linux 内核中，引入了 MTD 层为 NOR Flash 和 NAND Flash 设备提供统一的接口，从而使得 Flash 驱动的设计工作大为简化。

19.1 节讲解了 Linux Flash 驱动的结构，主要讲解了 MTD 系统的层次结构和接口。

19.2 节和 19.3 节分别讲解了 NOR Flash 和 NAND Flash 驱动的设计方法，给出了设计模板。

19.4 节和 19.5 节分别以 S3C2410 外围 NOR Flash 和 NAND Flash 为实例进一步讲解了 NOR Flash 和 NAND Flash 驱动的设计。

19.6 节讲解了如何在 Flash 上建立 cramfs、jffs/jffs2 及 yaffs/yaffs2 文件系统。

19.1 Linux Flash 驱动结构

19.1.1 Linux MTD 系统层次

在 Linux 系统中，提供了 MTD（Memory Technology Device，内存技术设备）系统来建立 Flash 针对 Linux 的统一、抽象的接口。MTD 将文件系统与底层的 Flash 存储器进行了隔离，使 Flash 驱动工程师无须关心 Flash 作为字符设备和块设备与 Linux 内核的 接口。

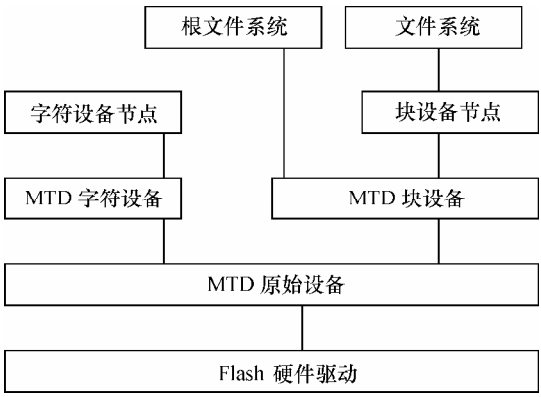


图 19.1 Linux MTD 系统

如图 19.1 所示，在引入 MTD 后，Linux 系统中的 Flash 设备驱动及接口可分为 4 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，这 4 层的作用如下。

- 硬件驱动层：Flash 硬件驱动层负责 Flash 硬件设备的读、写、擦除，Linux MTD 设备的 NOR Flash 芯片驱动位于 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。
- MTD 原始设备层：MTD 原始设备层由两部分组成，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。
- 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），构成 MTD 设备层。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、ioctl）可实现对 MTD 设备的读写和控制。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。
- 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），用户通过访问此设备节点即可访问 MTD 字符设备和块设备。

19.1.2 Linux MTD 系统接口

如图 19.2 所示，在引入 MTD 后，底层 Flash 驱动直接与 MTD 原始设备层交互，利用其提供的接口注册设备和分区。

用于描述 MTD 原始设备的数据结构是 mtd_info，这其中定义了大量关于 MTD 的

数据和操作函数，这个结构体的定义如代码清单 19.1 所示。mtd_info 是表示 MTD 原始设备的结构体，每个分区也被认为是一个 mtd_info，例如，如果有两个 MTD 原始设备，而每个上有 3 个分区，在系统中就将共有 6 个 mtd_info 结构体，这些 mtd_info 的指针被存放在名为 mtd_table 的数组里。

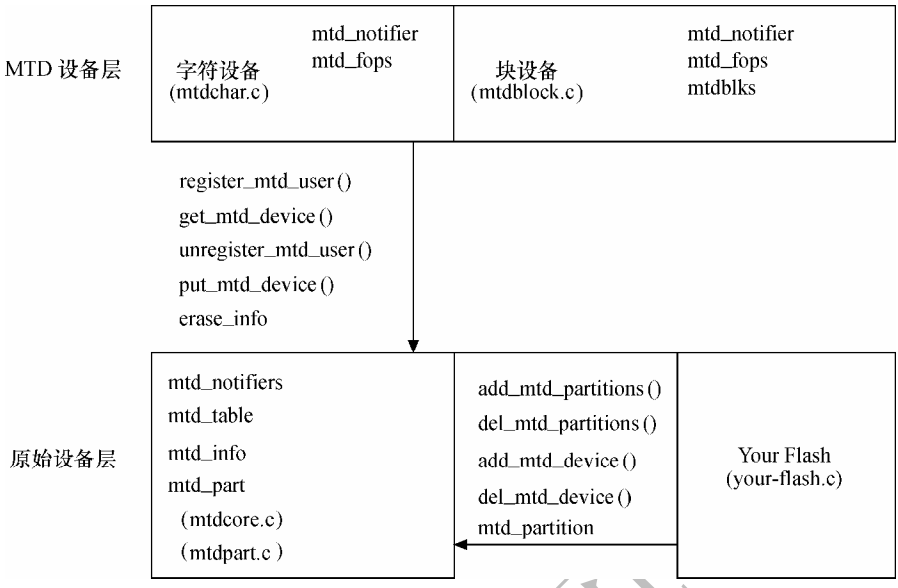


图 19.2　底层 Flash 驱动

代码清单 19.1　mtd_info 结构体

```
1  struct mtd_info
2  {
3
4    u_char type; // 内存技术的类型
5    u_int32_t flags; //标志位
6    u_int32_t size; //mtd 设备的大小
7    u_int32_t erasesize; //主要的擦除块大小（同一个
8                          //mtd 设备可能有数种不同的 erasesize）
9    u_int32_t oobblock; // oob 块大小
10   u_int32_t oobsize; // oob 数据大小
11   u_int32_t ecctype; //ecc 类型
12   u_int32_t eccsize; //ecc 工作的范围
13
14   char *name;
15   int index; //索引
16
17   //不同的 erasesize 的区域
18   int numeraseregions; //不同 erasesize 的区域的数目（通常是 1）
19   struct mtd_erase_region_info *eraseregions;
20
21   u_int32_t bank_size;
22   struct module *module;
23   int(*erase)(struct mtd_info *mtd, struct erase_info *instr);
24   //此 routine 用于将一个 erase_info 加入 erase queue
```

```
    25
    26   /*针对 eXecute-In-Place */
    27   int(*point)(struct mtd_info *mtd, loff_t from, size_t len, size_t
*retlen,
    28     u_char **mtdbuf);
    29
    30   /* 如果 unpoint 为空，不允许 XIP */
    31   void(*unpoint)(struct mtd_info *mtd, u_char *addr);
    32
    33   int(*read)(struct mtd_info *mtd, loff_t from, size_t len, size_t
*retlen,
    34     u_char *buf);  //读 Flash
    35   int(*write)(struct mtd_info *mtd, loff_t to, size_t len, size_t
*retlen,
    36     const u_char *buf); //写 Flash
    37    int(*read_ecc)(struct mtd_info *mtd, loff_t from, size_t len,
size_t *retlen,
    38     u_char *buf, u_char *eccbuf); //带 ecc 的读
    39   int(*write_ecc)(struct mtd_info *mtd, loff_t to, size_t len, size_t
*retlen,
    40     const u_char *buf, u_char *eccbuf); //带 ecc 的写
    41    int(*read_oob)(struct mtd_info *mtd, loff_t from, size_t len,
size_t *retlen,
    42     u_char *buf); //读 out-of-band
    43   int(*write_oob)(struct mtd_info *mtd, loff_t to, size_t len, size_t
*retlen,
    44     const u_char *buf); //写 out-of-band
    45   /* iovec-based 读写函数，对于 NAND Flash 需要针对它定义  */
    46   int(*readv)(struct mtd_info *mtd, struct iovec *vecs, unsigned long
count,
    47     loff_t from, size_t *retlen);
    48    int(*writev)(struct mtd_info *mtd, const struct iovec *vecs,
unsigned long
    49     count, loff_t to, size_t *retlen);
    50
    51   /* Sync */
    52   void(*sync)(struct mtd_info *mtd);
    53
    54   /* 设备锁 */
    55   int(*lock)(struct mtd_info *mtd, loff_t ofs, size_t len);
    56   int(*unlock)(struct mtd_info *mtd, loff_t ofs, size_t len);
    57   /* 能量管理函数*/
    58   int(*suspend)(struct mtd_info *mtd);
    59   void(*resume)(struct mtd_info *mtd);
    60
    61   void *priv; //私有数据
    62 }
```

mtd_info 的 type 字段给出底层物理设备的类型，包括 MTD_RAM、MTD_ROM、MTD_ NORFlash、MTD_NANDFlash、MTD_PEROM 等。

flags 字段包括 MTD_ERASEABLE（可擦除）、MTD_WRITEB_WRITEABLE（可编程）、MTD_XIP（可片内执行）、MTD_OOB（NAND 带外数据）、MTD_ECC（支持自动 ECC）等。某些内存技术支持带外数据（OOB），例如，NAND Flash 每 512 字节就会有 16 个字节的“额外数据”，用于存放纠错码或元数据。这是因为，所有 Flash 器件都受位交换现象的困扰，而 NAND 发生的概率比 NOR 大，因此 NAND 厂商推荐在使用 NAND 的时候最好要使用 ECC（Error Checking and Correcting），汉明码是最简单的 ECC。

ecctype 字段表明了 ECC 的类型，包括 MTD_ECC_NONE（不支持自动 ECC）、MTD_ECC_ RS_DiskOnChip（DiskOnChip 上自动 ECC）、MTD_ECC_SW（Toshiba & Samsung 设备 SW ECC）。

mtdcore.c 中定义了 MTD 设备数组：

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

最多可以有 MAX_MTD_DEVICES（默认定义为 16）个设备，每个 MTD 分区也算一个 MTD 设备。

mtd_info 中的 read()、write()、read_ecc()、write_ecc()、read_oob()、write_oob()是 MTD 设备驱动要实现的主要函数，后面我们将看到，在 NOR 和 NAND 的驱动代码中几乎看不到 mtd_info 的成员函数（也即这些成员函数对于 Flash 芯片驱动是透明的），这是因为 Linux 在 MTD 的下层实现了针对 NOR Hash 和 NAND Hash 的通用的 mtd_info 成员函数。

mtd_info 结构体在 Linux 2.6.18 内核发生了较大的变化，read_ecc()、write_ecc()成员函数被去掉，oobblock 等字段也被新的字段替代。

Flash 驱动中使用如下两个函数注册和注销 MTD 设备：

```
int add_mtd_device(struct mtd_info *mtd);
int del_mtd_device (struct mtd_info *mtd);
```

代码清单 19.2 所示的 mtd_part 结构体用于描述分区，其 mtd_info 结构体成员用于描述本分区，它会被加入到 mtd_table 中，其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。

代码清单 19.2　mtd_part 结构体

```
1 struct mtd_part
2 {
3   struct mtd_info mtd; //分区的信息（大部分由其 master 决定）
4   struct mtd_info *master; //该分区的主分区
5   u_int32_t offset; //该分区的偏移地址
6   int index; //分区号
7   struct list_head list;
8 };
```

mtd_partition 会在 MTD 原始设备层调用 add_mtd_partions()时传递分区信息用，这个结构体的定义如代码清单 19.3 所示。

代码清单 19.3 mtd_partition 结构体

```
1 struct mtd_partition
2 {
3   char *name;             /* 标识字符串 */
4   u_int32_t size;             /* 分区大小 */
5   u_int32_t offset;       /* 主 MTD 空间内的偏移*/
6   u_int32_t mask_flags;       /* 掩码标志 */
7 };
```

Flash 驱动中使用如下两个函数注册和注销分区：

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

add_mtd_partitions()会对每一个新建分区建立一个新的 mtd_part 结构体，将其加入 mtd_ partitions 中，并调用 add_mtd_device()将此分区作为 MTD 设备加入 mtd_table。成功时返回 0，如果分配 mtd_part 时内存不足，则返回-ENOMEM。

del_mtd_partitions()的作用是对于 mtd_partitions 上的每一个分区，如果它的主分区是 master（参数 master 是被删除分区的主分区），则将它从 mtd_partitions 和 mtd_table 中删除并释放掉，这个函数会调用 del_mtd_device()。

add_mtd_partitions()中新建的 mtd_part 需要依赖传入的 mtd_partition 参数对其进行初始化，如代码清单 19.4 所示。

代码清单 19.4 add_mtd_partitions()函数

```
1   int add_mtd_partitions(struct mtd_info *master, const struct
mtd_partition
2   *parts, int nbparts)
3  {
4   ...
5   for (i = 0; i < nbparts; i++)
6   {
7     slave = kmalloc(sizeof(*slave), GFP_KERNEL);
8     if (!slave)
9     {
10      printk("memory allocation error while creating partitions for
\"%s\"\n",
11           master->name);
12        del_mtd_partitions(master);
13        return  - ENOMEM;
14      }
15
16      memset(slave, 0, sizeof(*slave));
17      list_add(&slave->list, &mtd_partitions);
18
19      /* 设置该分区的 MTD 对象 */
20      slave->mtd.type = master->type;
21      slave->mtd.flags = master->flags &~parts[i].mask_flags;
22      slave->mtd.size = parts[i].size;
23      slave->mtd.oobblock = master->oobblock;
24      slave->mtd.oobsize = master->oobsize;
```

```
25       slave->mtd.ecctype = master->ecctype;
26       slave->mtd.eccsize = master->eccsize;
27
28      slave->mtd.name = parts[i].name;
29      slave->offset = parts[i].offset;
30
31    }
32    ...
33 }
```

最后，为了使系统能支持 MTD 字符设备与块设备及 MTD 分区，在编译内核时应该包括相应的配置选项，如下所示。

```
Memory Technology Devices (MTD)  --->
 <*> Memory Technology Device (MTD) support
  [*]   MTD partitioning support
   ……
  --- User Modules And Translation Layers
  <*> Direct char device access to MTD devices
  <*> Caching block device access to MTD devices
  ……
```

19.1.3 MTD 用户空间编程

mtdchar.c 实现了字符设备接口，通过它，用户可以直接操作 Flash 设备。通过 read()、write()系统调用可以读写 Flash，通过一系列 IOCTL 命令可以获取 Flash 设备信息、擦除 Flash、读写 NAND 的 OOB、获取 OOB layout 及检查 NAND 坏块等。

代码清单 19.5 所示为 MEMGETINFO、MEMERASE、MEMREADOOB、MEMWRITEOOB、MEMGETBADBLOCK IOCRL 的例子。

代码清单 19.5 /dev/mtdX IOCTL 演示范例

```
1  mtd_oob_buf oob;
2  erase_info_user erase;
3  mtd_info_user meminfo;
4
5  //得到 MTD 设备信息
6  if (ioctl(fd, MEMGETINFO, &meminfo) != 0)
7  {
8    perror("ioctl(MEMGETINFO)");
9  }
10
11 //擦除块
12 if (ioctl(ofd, MEMERASE, &erase) != 0)
13 {
14   perror("ioctl(MEMERASE)");
```

```
    goto error;
}

//读 OOB
if (ioctl(fd, MEMREADOOB, &oob) != 0)
{
  perror("ioctl(MEMREADOOB)");
}

//写 OOB
if (ioctl(fd, MEMWRITEOOB, &oob) != 0)
{
  fprintf(stderr, "\n%s: %s: MTD writeoob failure: %s\n", exe_name,
mtd_device,
      strerror(errno));
}

//检查坏块
if (blockstart != (ofs &(~meminfo.erasesize + 1)))
{
  blockstart = ofs &(~meminfo.erasesize + 1);
  if ((badblock = ioctl(fd, MEMGETBADBLOCK, &blockstart)) < 0)
  {
    perror("ioctl(MEMGETBADBLOCK)");
  }
  else if (badblock)//是坏块
  {
    ...
  }
  else//是好块
  ...
}
```

代码清单 19.6 所示的过程如下：它读取记录在一个映像文件中的针对 NAND 的数据信息，并把它写到 NAND Flash 中。代码中涉及大量 IOCTL 的调用及 NAND 的擦除和写入过程（含坏块检查），mtd-utils 中 nand_write、Flash_eraseall 等工具也是借助类似方法实现的。

代码清单 19.6　/dev/mtdX 用户空间编程综合范例

```
int main(int argc, char **argv)
{
  struct mtd_info_user meminfo;
  struct mtd_oob_buf oob;
  char  oobbuf[MAX_OOB_SIZE];
  ...

  memset(oobbuf, 0xff, sizeof(oobbuf));

  /* 打开/dev/mtdX */
  if ((fd = open(mtd_device, O_RDWR)) ==  - 1)
  {
```

```
13      perror("open Flash");
14      exit(1);
15    }
16
17    /* 填充 MTD 设备容量结构体 */
18    if (ioctl(fd, MEMGETINFO, &meminfo) != 0)
19    {
20      perror("MEMGETINFO");
21      close(fd);
22      exit(1);
23    }
24
25    oob.length = meminfo.oobsize;
26    oob.ptr = oobbuf;
27
28    /* 打开输入文件 */
29    if ((ifd = open(img, O_RDONLY)) ==  - 1)
30    {
31      perror("open input file");
32      goto restoreoob;
33    }
34
35
36    imglen = lseek(ifd, 0, SEEK_END);   // 得到映像长度
37    lseek(ifd, 0, SEEK_SET);
38
39    pagelen = meminfo.oobblock + meminfo.oobsize; //一页的（数据+oob）
长度
40    ...
41
42    /* 从输入文件读数据然后写入 MTD 设备 */
43    while (imglen && (mtdoffset < meminfo.size))
44    {
45      // 在擦除块之前检查是否为坏块
46      while (blockstart != (mtdoffset &(~meminfo.erasesize + 1)))
47      {
48        blockstart = mtdoffset &(~meminfo.erasesize + 1);
49        offs = blockstart;
50        baderaseblock = 0;
51        if (!quiet)
52          fprintf(stdout, "Writing data to block %x\n", blockstart);
53
54        /* 检查坏块 */
55        do
56        {
57          if ((ret = ioctl(fd, MEMGETBADBLOCK, &offs)) < 0)
58          {
59            perror("ioctl(MEMGETBADBLOCK)");
```

```
60          goto closeall;
61        }
62        if (ret == 1)
63        {
64          baderaseblock = 1;
65          if (!quiet)
66            fprintf(stderr,
67              "Bad block at %x, %u block(s) from %x will be skipped\n",
68              (int)offs, blockalign, blockstart);
69        }
70        if (baderaseblock)
71        {
72          mtdoffset = blockstart + meminfo.erasesize;
73        }
74        offs += meminfo.erasesize / blockalign;
75      }  while (offs < blockstart + meminfo.erasesize);
76    }
77
78    readlen = meminfo.oobblock;
79
80    /* 从输入文件中读 page 数据 */
81    if ((cnt = read(ifd, writebuf, readlen)) != readlen)
82    {
83      if (cnt == 0)   // EOF
84        break;
85      perror("File I/O error on input file");
86      goto closeall;
87    }
88
89    /* 从输入文件读 OOB 数据 */
90        if ((cnt = read(ifd, oobreadbuf, meminfo.oobsize)) !=
meminfo.oobsize)
91    {
92      perror("File I/O error on input file");
93      goto closeall;
94    }
95
96    /* 将 OOB 数据写入设备 */
97    oob.start = mtdoffset;
98    if (ioctl(fd, MEMWRITEOOB, &oob) != 0)
99    {
100     perror("ioctl(MEMWRITEOOB)");
101     goto closeall;
102   }
103
104   /* 写 page 数据 */
105     if (pwrite(fd, writebuf, meminfo.oobblock, mtdoffset) !=
```

```
meminfo.oobblock)
    106     {
    107       perror("pwrite");
    108       goto closeall;
    109     }
    110     imglen -= readlen;
    111     mtdoffset += meminfo.oobblock;
    112   }
    113
    114   closeall: ...
    115   return 0;
    116 }
```

19.2 NOR Flash 驱动

图 19.3　MTD、通用 NOR Flash

在 Linux 系统中，实现了针对 cfi、jedec 等接口的通用 NOR 驱动，这一层的驱动直接面向 mtd_info 的成员函数，这使得 NOR 的芯片级驱动变得十分简单，只需要定义具体的内存映射情况结构体 map_info 并使用指定接口类型调用 do_map_ probe()。

NOR Flash 驱动的核心是定义 map_info 结构体，它指定了 NOR Flash 的基址、位宽、大小等信息以及 Flash 的读写函数，该结构体对于 NOR Flash 驱动而言至为关键，甚至 NOR Flash 驱动的代码本质上可以被认作是根据 map_info 探测芯片的过程，其定义如代码清单 19.7 所示。

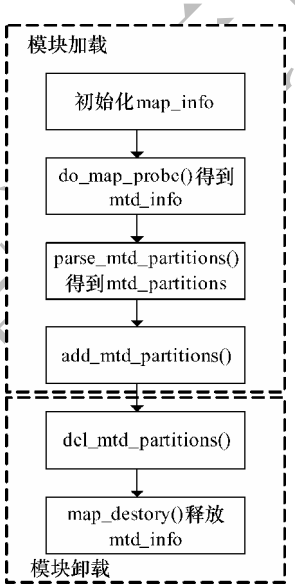
代码清单 19.7　map_info 结构体

```
1  struct map_info
2  {
3    char *name;
4    unsigned long size;
5    unsigned long phys;
6    #define NO_XIP (-1UL)
7
8    void __iomem *virt; /* 虚拟地址 */
9    void *cached;
10
11   int bankwidth; /* 总线宽度 */
12
13   #ifdef CONFIG_MTD_COMPLEX_MAPPINGS
```

```
    14    map_word(*read)(struct map_info *, unsigned long);
    15      void(*copy_from)(struct map_info *, void *, unsigned long,
ssize_t);
    16
    17    void(*write)(struct map_info *, const map_word, unsigned long);
    18    void(*copy_to)(struct map_info *, unsigned long, const void *,
ssize_t);
    19  #endif
    20  /* 缓存的虚拟地址 */
    21  void(*inval_cache)(struct map_info *, unsigned long, ssize_t);
    22
    23  void(*set_vpp)(struct map_info *, int);
    24
    25  unsigned long map_priv_1;
    26  unsigned long map_priv_2;
    27  void *fldrv_priv;
    28  struct mtd_chip_driver *fldrv;
    29 };
```

NOR Flash 驱动在 Linux 中的实现非常简单，如图 19.4 所示，主要的工作如下。

（1）定义 map_info 的实例，初始化其中的成员，根据目标板的情况为 name、size、bankwidth 和 phys 赋值。



19.4　NOR Flash 驱动

（2）如果 Flash 要分区，则定义 mtd_partition 数组，将实际电路板中 Flash 分区信息记录于其中。

（3）以 map_info 和探测的接口类型（如"cfi_probe"、"jedec_probe"等）为参数调用 do_map_ probe()，探测 Flash 得到 mtd_info。

do_map_probe()的函数原型为：

```
struct mtd_info *do_map_probe(const char *name, struct map_info *map);
```

第一个参数为探测的接口类型，常见的调用方法如下。

```
do_map_probe("cfi_probe",&xxx_map_info);
do_map_probe("jedec_probe",&xxx_map_info);
do_map_probe("map_rom",&xxx_map_info);
```

do_map_probe()会根据传入的参数 name 通过 get_mtd_chip_driver()得到具体的 MTD 驱动，调用与接口对应的 probe()函数探测设备，如代码清单 19.8 所示。

代码清单 19.8 do_map_probe()函数

```
1  struct mtd_info *do_map_probe(const char *name, struct map_info *map)
2  {
3   struct mtd_chip_driver *drv;
4   struct mtd_info *ret;
5
6   drv = get_mtd_chip_driver(name);/* 通过名称获得驱动 */
7
8   if (!drv && !request_module("%s", name))
9       drv = get_mtd_chip_driver(name);
10
11  if (!drv)
12      return NULL;
13
14  ret = drv->probe(map);/* 调用驱动的探测函数 */
15
16  module_put(drv->module);
17  if (ret)
18      return ret;
19
20  return NULL;
21 }
```

利用 map_info 中的配置，do_map_probe()可以自动识别支持 CFI 或 JEDEC（电子元件工业联合会）接口的 Flash 芯片，MTD 以后会自动采用适当的命令参数对 Flash 进行读写或擦除。

（4）在模块初始化时以 mtd_info 为参数调用 add_mtd_device()或以 mtd_info、mtd_partition 数组及分区数为参数调用 add_mtd_partitions()注册设备或分区。当然，在这之前可以调用 parse_mtd_partitions()查看 Flash 上是否已有分区信息，并将查看出的分区信息通过 add_mtd_partitions()注册。

（5）在模块卸载时调用第 4 行函数的“反函数”删除设备或分区。

代码清单 19.9 所示为一个最简单的 NOR Flash 驱动模板。

代码清单 19.9 NOR Flash 设备驱动模板

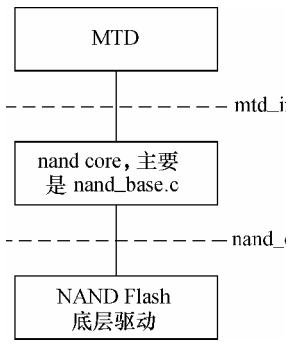
```
1  #define WINDOW_SIZE ...
2  #define WINDOW_ADDR ...
3  static struct map_info xxx_map = {  //map_info
```

```
    .name = "xxx Flash",
    .size = WINDOW_SIZE, //大小
    .bankwidth = 1,   //总线宽度
    .phys = WINDOW_ADDR //物理地址
};

static struct mtd_partition xxx_partitions[] = { // mtd_partition
 {
     .name = "Drive A",
     .offset = 0,   //分区的偏移地址
     .size = 0x0e0000 //分区大小
 },
 ...
};

#define NUM_PARTITIONS ARRAY_SIZE(xxx_partitions)

static struct mtd_info *mymtd;

static int __init init_xxx_map(void)
{
 int rc = 0;

 xxx_map.virt=ioremap_nocache(xxx_map.phys, xxx_map.size);//物理->虚拟地址

 if (!xxx_map.virt) {
     printk(KERN_ERR "Failed to ioremap_nocache\n");
     rc = -EIO;
     goto err2;
 }

 simple_map_init(&xxx_map);

 mymtd = do_map_probe("jedec_probe", &xxx_map);//探测 nor Flash
 if (!mymtd) {
     rc = -ENXIO;
     goto err1;
 }

 mymtd->owner = THIS_MODULE;
 add_mtd_partitions(mymtd, xxx_partitions, NUM_PARTITIONS);//添加分区信息

```

```
46  return 0;
47
48 err1:
49  map_destroy(mymtd);
50  iounmap(xxx_map.virt);
51 err2:
52  return rc;
53 }
54
55 static void __exit cleanup_xxx_map(void)
56 {
57  if (mymtd) {
58      del_mtd_partitions(mymtd);//删除分区
59      map_destroy(mymtd);
60  }
61
62  if (xxx_map.virt) {
63      iounmap(xxx_map.virt);
64      xxx_map.virt = NULL;
65  }
66 }
```

19.3 NAND Flash 驱动

和 NOR Flash 非常类似，如图 19.5 所示，Linux 内核在 MTD 的下层实现了通用的 NAND 驱动（主要通过 drivers/mtd/nand/nand_base.c 文件实现），因此芯片级的 NAND 驱动不再需要实现 mtd_info 中的 read()、write()、read_oob()、write_oob()等成员函数，而主体转移到了 nand_chip 数据结构。



19.5 NAND Flash 驱动

MTD 使用 nand_chip 数据结构表示一个 NAND Flash 芯片，这个结构体中包含了关于 NAND Flash 的地址信息、读写方法、ECC 模式、硬件控制等一系列底层机制，其定义如代码清单 19.10 所示。

代码清单 19.10　nand_chip 结构体

```
1  struct nand_chip
2  {
3    void __iomem *IO_ADDR_R; //读 8 根 I/O 线的地址
4    void __iomem *IO_ADDR_W; //写 8 根 I/O 线的地址
```

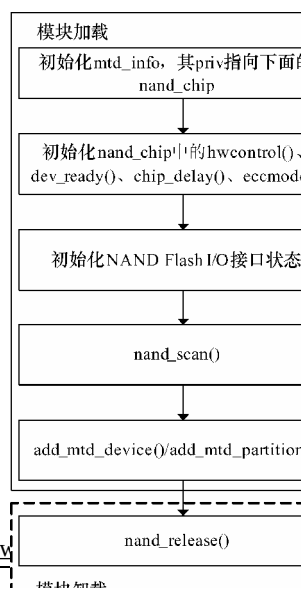
```
    5
    6    u_char(*read_byte)(struct mtd_info *mtd); //从芯片读一个字节
    7    void(*write_byte)(struct mtd_info *mtd, u_char byte); //向芯片写
一个字节
    8    u16(*read_word)(struct mtd_info *mtd); //从芯片读一个字
    9    void(*write_word)(struct mtd_info *mtd, u16 word); //向芯片写一个
字
    10
    11   void(*write_buf)(struct mtd_info *mtd, const u_char *buf, int len);
    12     //将缓冲区内容写入芯片
    13   void(*read_buf)(struct mtd_info *mtd, u_char *buf, int len);
    14     //将芯片数据读到缓冲区
    15   int(*verify_buf)(struct mtd_info *mtd, const u_char *buf, int len);
    16     //验证芯片和写入缓冲区中的数据
    17   void(*select_chip)(struct mtd_info *mtd, int chip); //控制 CE 信号
    18   int(*block_bad)(struct mtd_info *mtd, loff_t ofs, int getchip);
    19     //检查是否为坏块
    20   int(*block_markbad)(struct mtd_info *mtd, loff_t ofs); //标志坏块
    21   void(*hwcontrol)(struct mtd_info *mtd, int cmd); //板特定的硬件控
制
    22   int(*dev_ready)(struct mtd_info *mtd); //板特定的设备 ready/busy 信
息
    23   void(*cmdfunc)(struct mtd_info *mtd, unsigned command, int column,
int
    24     page_addr); //命令处理函数
    25   int(*waitfunc)(struct mtd_info *mtd, struct nand_chip *this, int
state);
    26    int(*calculate_ecc)(struct mtd_info *mtd, const u_char *dat,
u_char *ecc_code)
    27     ; //计算 ecc
    28    int(*correct_data)(struct mtd_info *mtd, u_char *dat, u_char
*read_ecc,
    29     u_char *calc_ecc); //纠正数据
    30   void(*enable_hwecc)(struct mtd_info *mtd,int mode);//板特定的硬件
ECC 使能
    31   void(*erase_cmd)(struct mtd_info *mtd, int page); //擦除命令处理
    32   int(*scan_bbt)(struct mtd_info *mtd); //扫描坏块
    33   int eccmode; //板特定的 ECC 模式
    34   int eccsize; //ECC 尺寸
    35   int eccbytes; //ECC 字节，每 eccsize 算出 eccbytes 个校验码
```

```
    int eccsteps;
    int chip_delay; //芯片特定的延迟
    spinlock_t chip_lock;
    wait_queue_head_t wq;
    nand_state_t state; //芯片状态，如输入/输出
    int page_shift;
    int phys_erase_shift;
    int bbt_erase_shift;
    int chip_shift;
    u_char *data_buf; //数据 buffer
    u_char *oob_buf; //oob buffer
    int oobdirty;
    u_char *data_poi;
    unsigned int options;
    int badblockpos;
    int numchips;
    unsigned long chipsize;
    int pagemask;
    int pagebuf;
    struct nand_oobinfo *autooob;
    uint8_t *bbt;
    struct nand_bbt_descr *bbt_td;
    struct nand_bbt_descr *bbt_md;
    struct nand_bbt_descr *badblock_pattern;
    struct nand_hw_control *controller;
    void *priv;
    int(*errstat)(struct mtd_info *mtd, struct nand_chip *this, int
state, int
      status, int page);
};
```

与 NOR Flash 类似，由于有了 MTD 层，完成一个 NAND Flash 驱动在 Linux 中的工作量也很小，如图 19.6 所示，主要的工作如下。

（1）如果 Flash 要分区，则定义 mtd_partition 数组，将实际电路板中 Flash 分区信息记录于其中。

（2）在模块加载时分配和 nand_chip 的内存，根据目标板 NAND 控制器的特殊情况初始化 nand_chip 中的 hwcontrol()、dev_ready()、calculate_ecc()、correct_data()、read_byte()、write_byte()等成员函数（如果不赋值会使用 nand_base.c 中的默认函数），注



意将 mtd_info 的 priv 置为 nand_chip。

如果使用软件 ECC，则不需要赋值 calculate_ecc()和 correct_data()成员，因为 NAND 核心层包含了相应的软件算法。如果使用 NAND 控制器的 ECC，应该自定义 calculate_ecc()函数，并在其中将硬件上产生的 ECC 字节返回到 ecc_code 参数。

（3）以 mtd_info 为参数调用 nand_scan()函数探测 NAND Flash 的存在。nand_scan()函数的原型为：

```
int nand_scan (struct mtd_info *mtd, int maxchips);
```

nand_scan()函数会读取 NAND 芯片 ID，并根据 mtd->priv 即 nand_chip 中的成员初始化 mtd_info。

（4）如果要分区，则以 mtd_info 和 mtd_partition 为参数调用 add_mtd_partitions()，添加分区信息。

代码清单 19.11 所示为一个简单的 NAND Flash 设备驱动模板。

代码清单 19.11　NAND Flash 设备驱动模板

```
1  #define CHIP_PHYSICAL_ADDRESS  ...
2  #define NUM_PARTITIONS 2
3  static struct mtd_partition partition_info[] =
4  {
5    {
6      .name = "Flash partition 1", .offset = 0, .size = 8 * 1024 * 1024
7    },
8    {
9      .name = "Flash partition 2", .offset = MTDPART_OFS_NEXT, .size
=
10       MTDPART_SIZ_FULL
11   } ,
12 };
13 int __init board_init(void)
14 {
15   struct nand_chip *this;
16   int err = 0;
17   /* 为 MTD 设备结构体和 nand_chip 分配内存 */
18     board_mtd = kmalloc(sizeof(struct mtd_info) + sizeof(struct
nand_chip),
19     GFP_KERNEL);
20   if (!board_mtd)
21   {
22     printk("Unable to allocate NAND MTD device structure.\n");
23     err =  - ENOMEM;
24     goto out;
25   }
26   /* 初始化结构体 */
```

```
    memset((char*)board_mtd,  0,  sizeof(struct  mtd_info)  +
sizeof(struct nand_chip)
    );
  /* 映射物理地址 */
  baseaddr = (unsigned long)ioremap(CHIP_PHYSICAL_ADDRESS, 1024);
  if (!baseaddr)
  {
    printk("Ioremap to access NAND chip failed\n");
    err =  - EIO;
    goto out_mtd;
  }
  /* 获得私有数据（nand_chip）指针 */
  this = (struct nand_chip*)(&board_mtd[1]);
  /* 将 nand_chip 赋予 mtd_info 私有指针 */
  board_mtd->priv = this;
  /* 设置 NAND Flash 的 I/O 基地址 */
  this->IO_ADDR_R = baseaddr;
  this->IO_ADDR_W = baseaddr;
  /* 硬件控制函数 */
  this->hwcontrol = board_hwcontrol;
  /* 从数据手册获知命令延迟时间 */
  this->chip_delay = CHIP_DEPENDEND_COMMAND_DELAY;
  /* 初始化设备 ready 函数 */
  this->dev_ready = board_dev_ready;
  this->eccmode = NAND_ECC_SOFT;
  /* 扫描以确定设备的存在 */
  if (nand_scan(board_mtd, 1))
  {
    err =  - ENXIO;
    goto out_ior;
  }
  //添加分区
  add_mtd_partitions(board_mtd, partition_info, NUM_PARTITIONS);
  goto out;
  out_ior: iounmap((void*)baseaddr);
  out_mtd: kfree(board_mtd);
  out: return err;
}
```

```
64
65 static void __exit board_cleanup(void)
66 {
67   /* 释放资源，注销设备 */
68   nand_release(board_mtd);
69   /* unmap 物理地址 */
70   iounmap((void*)baseaddr);
71   /* 释放 MTD 设备结构体 */
72   kfree(board_mtd);
73 }
74
75 /* GPIO 方式的硬件控制 */
76 static void board_hwcontrol(struct mtd_info *mtd, int cmd)
77 {
78   switch (cmd)
79   {
80     case NAND_CTL_SETCLE:
81       /* Set CLE pin high */
82       break;
83     case NAND_CTL_CLRCLE:
84       /* Set CLE pin low */
85       break;
86     case NAND_CTL_SETALE:
87       /* Set ALE pin high */
88       break;
89     case NAND_CTL_CLRALE:
90       /* Set ALE pin low */
91       break;
92     case NAND_CTL_SETNCE:
93       /* Set nCE pin low */
94       break;
95     case NAND_CTL_CLRNCE:
96       /* Set nCE pin high */
97       break;
98   }
99 }
100
101 /* 返回设备 ready 状态 */
102 static int board_dev_ready(struct mtd_info *mtd)
103 {
104 return xxx_read_ready_bit();
105 }
```

最后要强调的是，在 NAND 芯片级驱动中，如果在 nand_chip 中没有赋值，将使

用如代码清单 19.12 所示的默认分布，应该根据实际 NAND 控制器和 NAND 芯片的情况给 nand_chip 的 nand_oobinfo 成员赋值，定义 OOB 的分布。

代码清单 19.12　NAND 驱动默认的 OOB 分布

```
1  //页尺寸为 256，OOB 字节数为 8
2  static struct nand_oobinfo nand_oob_8 =
3  {
4    .useecc = MTD_NANDECC_AUTOPLACE,
5    .eccbytes = 3,
6    .eccpos = {0, 1, 2},
7    .oobfree = { {3, 2}, {6, 2} }
8  };
9
10 //页尺寸为 512，OOB 字节数为 16
11 static struct nand_oobinfo nand_oob_16 =
12 {
13  .useecc = MTD_NANDECC_AUTOPLACE,
14  .eccbytes = 6,
15  .eccpos = {0, 1, 2, 3, 6, 7},
16  .oobfree = { {8, 8} }
17 };
18
19 //页尺寸为 2048，OOB 字节数为 64
20 static struct nand_oobinfo nand_oob_64 =
21 {
22  .useecc = MTD_NANDECC_AUTOPLACE,
23  .eccbytes = 24,
24  .eccpos = {
25      40, 41, 42, 43, 44, 45, 46, 47,
26      48, 49, 50, 51, 52, 53, 54, 55,
27      56, 57, 58, 59, 60, 61, 62, 63},
28  .oobfree = { {2, 38} }
29 };
```

nand_oobinfo 中的 useecc 定义 ECC 的放置模式，包括 MTD_NANDECC_OFF（不使用 ECC）、MTD_NANDECC_AUTOPLACE（自动放置）、MTD_NANDECC_PLACE（使用结构体内指定的放置方式）等；eccbytes 定义 ECC 字节数；eccpos 是 ECC 校验码的放置位置；oobfree 中记录还可被自由使用的 OOB 区域的开始位置和长度，它是一个数组，因此，可自由使用的 OOB 区域可以是不连续的多个小片段。

内核中包含了一个 NAND 模拟器 nandsim，使用一片内存区域模拟 NAND，在没有电路板的情况下，可以使用 nandsim 模拟 NAND 芯片。YAFFS 和 YAFFS2 中分别

包含了 nandemul 和 nandemul2k（用于模拟页大小为 2KB 的 NAND），也可以模拟 NAND。

19.4 NOR Flash 驱动实例：S3C2410 外围的 NOR Flash 驱动

其实这一节的实例有非常强的适应性，虽然节名为 S3C2410 外围的 NOR Flash 驱动，实际上由于 NOR Flash 的总线接口与 SRAM 类似，没有什么特殊的控制，因此呈现出一定的平台无关性，该 NOR Flash 驱动只需稍微修改一下就可以应用在别的平台上。

代码清单 19.13　S3C2410 外围的 NOR Flash 驱动

```
1  #define WINDOW_ADDR 0x01000000      /* NOR Flash 物理地址 */
2  #define WINDOW_SIZE 0x800000        /* NOR Flash 大小 */
3  #define BUSWIDTH    2
4  /* 探测的接口类型，可以是"cfi_probe", "jedec_probe", "map_rom", NULL
*/
5  #define PROBETYPES { "cfi_probe", NULL }
6
7  #define MSG_PREFIX "S3C2410-NOR:"       /* printk 的前缀*/
8  #define MTDID      "s3c2410-nor"     /* MTD 驱动*/
9
10 static struct mtd_info *mymtd;
11
12 struct map_info s3c2410nor_map =  // map_info
13 {
14   .name = "NOR Flash on S3C2410",
15   .size = WINDOW_SIZE,
16   .bankwidth = BUSWIDTH,
17   .phys = WINDOW_ADDR,
18 };
19
20 #ifdef CONFIG_MTD_PARTITIONS
21   /* MTD 分区信息  */
22   static struct mtd_partition static_partitions[] =
23   {
24     {
25           .name  =  "BootLoader",  .size  =  0x040000,  .offset  =  0x0
//bootloader 存放的区域
```

```
    26      } ,
    27      {
    28        .name = "Kernel", .size = 0x0100000, .offset = 0x40000 //内
核映像存放的区域
    29      }
    30      ,
    31      {
    32          .name = "RamDisk", .size = 0x400000, .offset = 0x140000
//RamDisk 存放的区域
    33      }
    34      ,
    35      {
    36        .name = "cramfs(2MB)", .size = 0x200000, .offset = 0x540000 //
只读的 cramfs 区域
    37      }
    38      ,
    39      {
    40        .name = "jffs2(0.75MB)", .size = 0xc0000, .offset = 0x740000 //
可读写的 jffs2 区域
    41      }
    42      ,
    43    };
    44  #endif
    45
    46  static int mtd_parts_nb = 0;
    47  static struct mtd_partition *mtd_parts = 0;
    48
    49  int __init init_s3c2410nor(void)
    50  {
    51    static const char *rom_probe_types[] = PROBETYPES;
    52    const char **type;
    53    const char *part_type = 0;
    54
    55     printk(KERN_NOTICE MSG_PREFIX "0x%08x at 0x%08x\n", WINDOW_SIZE,
WINDOW_ADDR);
    56    s3c2410nor_map.virt = ioremap(WINDOW_ADDR, WINDOW_SIZE);//物理->
虚拟地址
    57
```

```
58   if (!s3c2410nor_map.virt)
59   {
60     printk(MSG_PREFIX "failed to ioremap\n");
61     return  - EIO;
62   }
63
64   simple_map_init(&s3c2410nor_map);
65
66   mymtd = 0;
67   type = rom_probe_types;
68   for (; !mymtd &&  *type; type++)
69   {
70     mymtd = do_map_probe(*type, &s3c2410nor_map);//探测 NOR Flash
71   }
72   if (mymtd)
73   {
74     mymtd->owner = THIS_MODULE;
75
76     #ifdef CONFIG_MTD_PARTITIONS
77        mtd_parts_nb = parse_mtd_partitions(mymtd, NULL, &mtd_parts,
MTDID);//探测分区信息
78       if (mtd_parts_nb > 0)
79         part_type = "detected";
80
81       if (mtd_parts_nb == 0) //未探测到，使用数组定义的分区信息
82       {
83         mtd_parts = static_partitions;
84         mtd_parts_nb = ARRAY_SIZE(static_partitions);
85         part_type = "static";
86       }
87     #endif
88     add_mtd_device(mymtd);
89     if (mtd_parts_nb == 0)
90              printk(KERN_NOTICE  MSG_PREFIX  "no  partition  info
available\n");
91     else
92     {
93             printk(KERN_NOTICE  MSG_PREFIX  "using  %s  partition
definition\n",
94         part_type);
```

```
    95       add_mtd_partitions(mymtd, mtd_parts, mtd_parts_nb);//添加分区
信息
    96     }
    97     return 0;
    98   }
    99
    100  iounmap((void*)s3c2410nor_map.virt);
    101  return  - ENXIO;
    102 }
    103
    104 static void __exit cleanup_s3c2410nor(void)
    105 {
    106  if (mymtd)
    107  {
    108    del_mtd_partitions(mymtd);  //删除分区
    109    del_mtd_device(mymtd);     //删除设备
    110    map_destroy(mymtd);
    111  }
    112  if (s3c2410nor_map.virt)
    113  {
    114    iounmap((void*)s3c2410nor_map.virt);
    115    s3c2410nor_map.virt = 0;
    116  }
    117 }
```

上述代码第 77 行调用的 parse_mtd_partitions()用于解析 MTD 分区信息，从第 76～95 行代码可以看出，当解析成功时，应该添加解析出来的分区表；否则，使用驱动中自定义的分区表。最常见的 MTD 分区解析器是命令行解析，即解析在 Linux 启动命令行中通过“mtdparts=”传入的 MTD 分区信息。

通过查看/proc/mtd 文件可以获知系统中包含的 MTD 设备（分区），如下所示：

```
dev :   size    erasesize  name
mtd0: 00040000  00020000  "bootloader"
mtd1: 002c0000  00020000  "kernel"
mtd2: 01000000  00020000  "rootfs"
...
```

19.5 NAND Flash 驱动实例：S3C2410 外围的 NAND Flash 驱动

19.5.1 S3C2410 NAND 控制器硬件描述

S3C2410 处理器集成了一个 NAND 控制器，它提供如下引脚。

- D[7:0]：数据/命令/地址 I/O 端口（数据、命令、地址复用）。
- CLE：命令锁存使能（数据线上 NAND Flash 命令有效，输出）。
- ALE：地址锁存使能（数据线上 NAND Flash 地址有效，输出）。
- nFCE：NAND Flash 片选（输出）。
- nFRE：NAND Flash 读使能（输出）。
- nFWE：NAND Flash 写使能（输出）。
- R/nB：NAND Flash 准备好/忙（输入）。
- NCON：输入，NAND Flash 内存地址步长选择，0：表示 3 步长地址，1：表示 4 步长地址（NAND Flash 中地址要通过 d[7:0]送多次，每送一次就为一步长）。

S3C2410 对 NAND Flash 的操作通过 NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT 和 NFECC 这 6 个寄存器来完成。

- NFCONF：NAND Flash 配置寄存器，用于使能 NAND Flash 控制器、初始化 ECC、并设置 NAND Flash 片选信号 nFCE 为 1（即不选中）。
- NFCMD：NAND Flash 命令寄存器，对于 page 大小不一样的不同型号的 NAND，其操作命令可能不一样。本节的例子芯片为 K9F1208U0M，在其数据手册中该芯片命令集的有关描述。
- NFADDR：NAND Flash 地址寄存器。
- NFDATA：NAND Flash 数据寄存器，8 位。
- NFSTAT：NAND Flash 状态寄存器，位 0 如果为 0 表示设备忙，否则表示设备准备好。
- NFECC：NAND Flash 校验寄存器。S3C2410 NAND 控制器内置的 ECC 生成模块执行以下任务：当写入数据时，ECC 生成模块产生一个 ECC 码；当读数据时，ECC 生成模块产生一个 ECC 码，驱动中可以将该 ECC 码与原先存入 OOB 中的校验码进行比较。

通过 S3C2410 的 NAND 控制器访问 NAND Flash 的一般流程如下。

（1）设置 NAND Flash 配置寄存器 NFCONF。

（2）在 NFCMD 寄存器中写入 NAND Flash 命令。

（3）在 NFADDR 寄存器中写入地址。

（4）读/写数据，通过 NFSTAT 寄存器检查 NAND Flash 状态，在读/写操作后 R/nB 信号应该被检查。

19.5.2 nand_chip 初始化和成员函数

nand_chip 是 NAND Flash 驱动的核心数据结构，这个结构体中的成员直接对应着 NAND Flash 的底层操作，针对具体的 NAND 控制器情况，本驱动中初始化了 write_buf()、read_buf()、select_chip()、chip_delay()及几个 ECC 相关的成员函数。代码清单 19.14 所示为 S3C2410 外围 NAND Flash 驱动的 nand_chip 初始化及其成员函数的实现。

代码清单 19.14 S3C2410 NAND 驱动的 nand_chip 初始化

```
1  static void s3c2410_nand_init_chip(struct s3c2410_nand_info *info,
```

```
struct
    2    s3c2410_nand_mtd *nmtd, struct s3c2410_nand_set *set)
    3  {
    4    struct nand_chip *chip = &nmtd->chip;
    5
    6    /* 初始化 nand_chip */
    7    chip->IO_ADDR_R = info->regs + S3C2410_NFDATA;
    8    chip->IO_ADDR_W = info->regs + S3C2410_NFDATA;
    9    chip->hwcontrol = s3c2410_nand_hwcontrol;
    10   chip->dev_ready = s3c2410_nand_devready;
    11   chip->write_buf = s3c2410_nand_write_buf;
    12   chip->read_buf = s3c2410_nand_read_buf;
    13   chip->select_chip = s3c2410_nand_select_chip;
    14   chip->chip_delay = 50;
    15   chip->priv = nmtd;
    16   chip->options = 0;
    17   chip->controller = &info->controller;
    18
    19   nmtd->info = info;
    20   nmtd->mtd.priv = chip;
    21   nmtd->set = set;
    22
    23   if (hardware_ecc)
    24   //如果采用硬件 ECC
    25   {
    26     chip->correct_data = s3c2410_nand_correct_data;
    27     chip->enable_hwecc = s3c2410_nand_enable_hwecc;
    28     chip->calculate_ecc = s3c2410_nand_calculate_ecc;
    29     chip->eccmode = NAND_ECC_HW3_512; //512 字节产生 3 字节的 ECC 码
    30     chip->autooob = &nand_hw_eccoob;
    31   }
    32   else
    33   //使用软件 ECC
    34   {
    35     chip->eccmode = NAND_ECC_SOFT;
    36   }
    37 }
    38
```

```
39  /* 读 NAND 中的数据到缓冲区 */
40  static void s3c2410_nand_read_buf(struct mtd_info *mtd, u_char *buf,
int len)
41  {
42    struct nand_chip *this = mtd->priv;
43    readsb(this->IO_ADDR_R, buf, len);
44  }
45
46  /* 写缓冲区的数据到 NAND Flash 中 */
47   static void s3c2410_nand_write_buf(struct mtd_info *mtd, const
u_char *buf, int
48    len)
49  {
50    struct nand_chip *this = mtd->priv;
51    writesb(this->IO_ADDR_W, buf, len);
52  }
53
54  /* 读取 ECC */
55  static int s3c2410_nand_calculate_ecc(struct mtd_info *mtd, const
u_char *dat,
56    u_char *ecc_code)
57  {
58    struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);
59
60    ecc_code[0] = readb(info->regs + S3C2410_NFECC + 0);
61    ecc_code[1] = readb(info->regs + S3C2410_NFECC + 1);
62    ecc_code[2] = readb(info->regs + S3C2410_NFECC + 2);
63
64       pr_debug("calculate_ecc:  returning  ecc  %02x,%02x,%02x\n",
ecc_code[0],
65     ecc_code[1], ecc_code[2]);
66
67    return 0;
68  }
69
70  /* 数据纠正 */
71  static int s3c2410_nand_correct_data(struct mtd_info *mtd, u_char
*dat,
72    u_char*read_ecc, u_char *calc_ecc)
73  {
```

```
    74    pr_debug("s3c2410_nand_correct_data(%p,%p,%p,%p)\n", mtd, dat,
read_ecc,
    75     calc_ecc);
    76
    77   pr_debug("eccs: read %02x,%02x,%02x vs calc %02x,%02x,%02x\n",
    78   read_ecc[0],read_ecc[1], read_ecc[2], calc_ecc[0], calc_ecc[1],
calc_ecc[2]);
    79
    80   if (read_ecc[0]== calc_ecc[0] && read_ecc[1] == calc_ecc[1] &&
read_ecc[2]
    81     == calc_ecc[2]) //ECC 码一致
    82     return 0;
    83
    84
    85   return  - 1;
    86 }
    87
    88 /* NAND Flash 准备好 */
    89 static int s3c2410_nand_devready(struct mtd_info *mtd)
    90 {
    91   struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);
    92   return readb(info->regs + S3C2410_NFSTAT) &S3C2410_NFSTAT_BUSY;
    93 }
    94
    95 /* 选中芯片 */
    96  static void s3c2410_nand_select_chip(struct mtd_info *mtd, int
chip)
    97 {
    98   struct s3c2410_nand_info *info;
    99   struct s3c2410_nand_mtd *nmtd;
    100  struct nand_chip *this = mtd->priv;
    101  void __iomem *reg;
    102  unsigned long cur;
    103  unsigned long bit;
    104
    105  nmtd = this->priv;
    106  info = nmtd->info;
    107
```

```
    bit = S3C2410_NFCONF_nFCE;
    reg = info->regs + S3C2410_NFCONF;

    cur = readl(reg);

    if (chip ==  - 1)    //不选中 NAND 芯片
    {
      cur |= bit; //nFCE 位置 1
    }
    else    //选中 1 个 NAND
    {
      //NAND 序号越界
      if (nmtd->set != NULL && chip > nmtd->set->nr_chips)
      {
        printk(KERN_ERR PFX "chip %d out of range\n", chip);
        return ;
      }

      if (info->platform != NULL)
      {
        if (info->platform->select_chip != NULL)
          (info->platform->select_chip)(nmtd->set, chip);
      }

      cur &= ~bit; //nFCE 位置 0
    }

    writel(cur, reg); //写 NFCONF 寄存器
  }

  /* NAND 硬件控制 */
  static void s3c2410_nand_hwcontrol(struct mtd_info *mtd, int cmd,
unsigned int
    ctrl)
  {
    struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);

    if (cmd == NAND_CMD_NONE) //不进行任何传输
      return ;

    if (ctrl &NAND_CLE)     //如果是发送命令
      writeb(cmd, info->regs + S3C2410_NFCMD);
    else    //传送地址
      writeb(cmd, info->regs + S3C2410_NFADDR);
  }

  /* 使能硬件 ECC 产生器 */
  static void s3c2410_nand_enable_hwecc(struct mtd_info *mtd, int
mode)
  {
    struct s3c2410_nand_info *info = s3c2410_nand_mtd_toinfo(mtd);
    unsigned long ctrl;

```

```
159   ctrl = readl(info->regs + S3C2410_NFCONF);
160   ctrl |= S3C2410_NFCONF_INITECC; //ECC 使能为置 1
161   writel(ctrl, info->regs + S3C2410_NFCONF);
162 }
```

当使用硬件 ECC 的时候，从上述代码第 30 行可以看出，NAND Flash 驱动将使用如代码清单 19.15 所示的 OOB 分布，它将 3 个字节的 ECC 码放置在 OOB 的第 0、1、2 字节处。

代码清单 19.15　S3C2410 NAND 驱动采用硬件 ECC 时的 OOB 分布

```
1 static struct nand_oobinfo nand_hw_eccoob =
2 {
3   .useecc     = MTD_NANDECC_AUTOPLACE,
4   .eccbytes   = 3,//ECC 字节数为 3
5   .eccpos     = {0, 1, 2 },//ECC 放置位置在 0、1、2
6   .oobfree    = { {8, 8} } //可自由使用的 OOB 区域
7 };
```

19.5.3　NAND 设备驱动初始化与释放

在 Linux 2.6 内核中，S3C2410 的 NAND 被注册为一个平台设置，因此，当 S3C2410 NAND 驱动模块加载调用 platform_driver_register(&s3c2410_nand_driver)注册平台驱动时，其对应的 platform_driver 结构体的 s3c24xx_nand_probe()成员函数将被调用。在 s3c24xx_nand_probe()函数中，将调用前述的 s3c2410_nand_init_chip()初始化 nand_chip，通过 nand_scan()扫描到 NAND Flash 存储器后，调用 s3c2410_nand_add_partition()添加 MTD 分区和设备信息。

同样地，当 NAND 驱动模块卸载调用 platform_driver_unregister(&s3c2410_nand_driver)移除平台驱动时，其对应的 platform_driver 结构体的 s3c2410_nand_remove 成员函数将被调用，该函数中会删除 MTD 设备和分区信息。

代码清单 19.16 所示为 NAND 设备驱动的初始化和释放的代码。经过 s3c24xx_nand_probe()的初始化之后，用户即可访问对应的 NAND 设备，经过 s3c2410_nand_remove()的释放，NAND Flash 设备和分区信息将被移除。

代码清单 19.16　S3C2410 NAND 驱动的初始化和释放函数

```
1  static int s3c24xx_nand_probe(struct platform_device *pdev, enum
s3c_cpu_type
2    cpu_type)
3  {
4    struct s3c2410_platform_nand *plat = to_nand_plat(pdev);
5    struct s3c2410_nand_info *info;
6    struct s3c2410_nand_mtd *nmtd;
```

```
7    struct s3c2410_nand_set *sets;
8    struct resource *res;
9    int err = 0;
10   int size;
11   int nr_sets;
12   int setno;
13
14   pr_debug("s3c2410_nand_probe(%p)\n", pdev);
15
16   info = kmalloc(sizeof(*info), GFP_KERNEL);
17   ...
18   platform_set_drvdata(pdev, info);
19
20   spin_lock_init(&info->controller.lock);
21   init_waitqueue_head(&info->controller.wq);
22
23   /* 获得时钟源并使能 */
24   info->clk = clk_get(&pdev->dev, "nand");
25   ...
26   clk_enable(info->clk);
27
28   /* 分配并映射资源 */
29   res = pdev->resource;
30   size = res->end - res->start + 1; //I/O 内存大小
31   info->area= request_mem_region(res->start, size, pdev->name);//
申请 I/O 内存
32   ...
33   info->device = &pdev->dev;
34   info->platform = plat;
35   info->regs = ioremap(res->start, size); //寄存器地址 ioremap
36   info->cpu_type = cpu_type;
37   ...
38   dev_dbg(&pdev->dev, "mapped registers at %p\n", info->regs);
39
40   /* 初始化硬件 */
41   err = s3c2410_nand_inithw(info, pdev);
42   if (err != 0)
43     goto exit_error;
44
45   sets = (plat != NULL) ? plat->sets: NULL;
```

```
    nr_sets = (plat != NULL) ? plat->nr_sets: 1;

    info->mtd_count = nr_sets;

    /* 分配 s3c2410 nand_info 中 s3c2410_nand_mtd 的内存*/
    size = nr_sets * sizeof(*info->mtds);
    info->mtds = kmalloc(size, GFP_KERNEL);
    ...

    /* 初始化所有可能存在的芯片 */
    nmtd = info->mtds;
    for (setno = 0; setno < nr_sets; setno++, nmtd++)
    {
      pr_debug("initialising set %d (%p, info %p)\n", setno, nmtd,
info);

      s3c2410_nand_init_chip(info, nmtd, sets);
      nmtd->scan_res = nand_scan(&nmtd->mtd, (sets) ? sets->nr_chips: 1);//
扫描 Flash,初始化 mtd_info
      if (nmtd->scan_res == 0)
      {
        s3c2410_nand_add_partition(info, nmtd, sets); //添加分区信息
      }
      if (sets != NULL)
        sets++;
    }

    if (allow_clk_stop(info))
    {
      dev_info(&pdev->dev, "clock idle support enabled\n");
      clk_disable(info->clk);
    }

    pr_debug("initialised ok\n");
    return 0;

    exit_error: s3c2410_nand_remove(pdev);

```

```
82    if (err == 0)
83      err =  - EINVAL;
84    return err;
85  }
86
87  static int s3c2410_nand_remove(struct platform_device *pdev)
88  {
89    struct s3c2410_nand_info *info = to_nand_info(pdev);
90
91    platform_set_drvdata(pdev, NULL);
92
93    if (info == NULL)
94      return 0;
95
96    /* 释放所有的 MTD 设备和分区  */
97
98    if (info->mtds != NULL)
99    {
100     struct s3c2410_nand_mtd *ptr = info->mtds;
101     int mtdno;
102     for (mtdno = 0; mtdno < info->mtd_count; mtdno++, ptr++)
103     {
104       pr_debug("releasing mtd %d (%p)\n", mtdno, ptr);
105       nand_release(&ptr->mtd);
106     }
107     kfree(info->mtds);
108   }
109
110   /* 释放公用资源 */
111   if (info->clk != NULL && !IS_ERR(info->clk))
112   {
113     if (!allow_clk_stop(info))
114       clk_disable(info->clk);
115     clk_put(info->clk);
116   }
117
118   if (info->regs != NULL)
119   {
120     iounmap(info->regs);
121     info->regs = NULL;
```

```
    122   }
    123   if (info->area != NULL)
    124   {
    125     release_resource(info->area);
    126     kfree(info->area);
    127     info->area = NULL;
    128   }
    129   kfree(info);
    130
    131   return 0;
    132 }
    133
    134 #ifdef CONFIG_MTD_PARTITIONS //如果配置了 MTD 分区支持
    135  static int s3c2410_nand_add_partition(struct s3c2410_nand_info
*info,
    136                      struct s3c2410_nand_mtd *mtd,
    137                      struct s3c2410_nand_set *set)
    138 {
    139   if (set == NULL)
    140     return add_mtd_device(&mtd->mtd); //添加 MTD 设备
    141
    142   if (set->nr_partitions > 0 && set->partitions != NULL) {
    143     return add_mtd_partitions(&mtd->mtd, set->partitions,
    144               set->nr_partitions);  //添加分区
    145   }
    146   return add_mtd_device(&mtd->mtd);
    147 }
    148 #else //内核没有配置 MTD 分区支持
    149  static int s3c2410_nand_add_partition(struct s3c2410_nand_info
*info,
    150               struct s3c2410_nand_mtd *mtd,
    151               struct s3c2410_nand_set *set)
    152 {
    153   return add_mtd_device(&mtd->mtd);//添加 MTD 设备
    154 }
    155 #endif
```

19.6 Flash 文件系统的建立

19.6.1 Flash 转换层

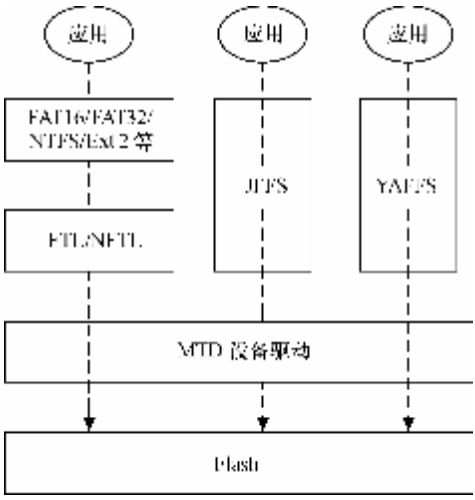


图 19.7 FTL 和 NFTL

由于无法重复地在 Flash 的同一块存储位置做写入操作（必须事先擦除该块后才能再写入），因此一般在硬盘上使用的文件系统，如 FAT16、FAT32、NTFS、Ext2 等将无法直接用在 Flash 上，为了沿用这些文件系统，则必须透过一层转换层（Translation Layer）来将逻辑块地址（Logical Block Address）对应到 Flash 存储器的物理位置，使系统能把 Flash 当作普通的硬盘一样处理，我们称这层为 FTL（Flash Translation Layer）。FTL 应用于 NOR Flash，而 NFTL 则应用于 NAND Flash，如图 19.7 所示。

一个闪存转换层的最简单的实现就是将模拟的块设备一对一地映射到闪存上。举例来说，当上层的文件系统要写一个块设备的扇区时，闪存转换层要做下面的操作来完成这个写请求。

（1）将这个扇区所在擦除块的数据读到内存中，放在缓存中。
（2）将缓存中与这个扇区对应的内容用新的内容替换。
（3）对该擦除块执行擦除操作。
（4）将缓冲中的数据写回该擦除块。

这种实现方式的缺点如下。

- 效率低，对一个扇区的更新要重写整个擦除块上的数据，造成数据带宽很大的浪费。可行的办法是只有当文件系统的写请求超过了一个擦除块的边界的时候，才去执行对闪存的擦除、写回操作（这种更新方式也叫 out-of-place）。
- 没有提供磨损平衡，那些被频繁更新的数据所在擦除块将首先变成坏块。
- 非常不安全，很容易引起数据的丢失。如果在上面的第（3）步和第（4）步之间发生了突然掉电，那么整个擦除块中的数据就全部丢失了。

为了解决上面这种实现方式的问题，闪存转换层不能只是简单地实现块设备与闪存的一一映射，它还需要将模拟块设备的扇区存储在闪存的不同位置，并且维持扇区到闪存的映射关系。而为了进行垃圾回收（Garbage Collection），闪存转换层必须能理解上层文件系统的语义。这样实现导致的最大问题就是效率不高，具体来说，闪存转换层为了能理解上层文件系统的语义，必须对文件系统的每个写请求进行解析，因此导致写操作的性能下降。另外，从软件的架构上来讲，要求文件系统下面的一层去理解文件系统的语义，也不太合理。因此，在 Flash 上，应尽可能地避免使用传统的依赖闪存转换层的文件系统，最好应采用专门的针对 Flash 的文件系统。

19.6.2 CramFS

在嵌入式 Linux 环境中，许多人会采用 RAMDISK 来储存文件系统的内容，RAMDISK 的含义是在启动时，把一部份内存虚拟成磁盘，并且把之前准备好的文件系统映像文件解压缩到该 RAMDISK 环境中。假设压缩后的文件系统映像为 8MB，存放于 Flash，解压缩后为 16MB，如果采用 RAMDISK，将需要 8MB 的 Flash 和 16MB

的 RAM 空间，而采用 CramFS 后，就不再需要消耗 16MB 的 RAM 空间。

CramFS 是 Linus Torvalds 参与开发的文件系统，在 linux/fs/cramfs 中可以找到 CramFS 的源代码。CramFS 是一种压缩的只读文件系统，当浏览 Flash 中的目录或读取文件时，CramFS 文件系统会动态地计算出压缩后的数据所储存的位置，并实时地解压缩到内存中，对于用户来说，使用 CramFS 与 RAMDISK 感觉不出使用上的差异性。

CramFS 工具的下载地址为 http://sourceforge.net/projects/cramfs/，通过如下命令可以创建 CramFS 文件系统映像：

```
mkcramfs my_cramfs/ cramfs.img （my_cramfs 是我们要创建映像的目录）
```

如下命令将生成的 cramfs.img 映像复制到 Flash 的第一个分区并 mount 到/mnt/nor 目录：

```
cp cramfs.img /dev/mtd1
mount -cramfs /dev/mtdblock1 /mnt/nor
```

很多时候，工程中需要基于已有的文件系统映像添加、删除一些文件后建立新的文件系统映像，这时候并不需要完全重新操作，可用如下的方法。

（1）将映像以 loop 方式挂载到某目录。

```
mkdir tmpdir
mount rootfs.cramfs tmpdir -o loop
cd tmpdir
```

（2）压缩被挂载的文件系统。

```
tar -cvf ../rootfs.tar ./     将 tmpdir 中的内容打包放在其父目录下
umount tmpdir
```

（3）解压缩文件系统到新目录。

```
mkdir rootfs
tar -xvf rootfs.tar -C rootfs
```

（4）修改新目录（这里是 rootfs）中的内容，以符合新的需要。

（5）重新创建映像文件。

```
mkcramfs rootfs rootfs.cramfs
```

19.6.3 JFFS/JFFS2

JFFS 是由瑞典 Axis Communications AB 公司开发的，于 1999 年年末基于 GNU GPL 发布的文件系统。最初的发布版本基于 Linux 2.0，后来 Red Hat 将它移植到 Linux 2.2，在使用的过程中，JFFS 设计中的局限被不断地暴露出来。于是在 2001 年年初，Red Hat 决定实现一个新的 JFFS2（http://www.infradead.org）。

JFFS2 是一个日志结构（log-structured）的文件系统，它在闪存上顺序地存储包含数据和原数据（meta-data）的节点。JFFS2 的日志结构存储方式使得它能对闪存进行 out-of-place 更新，而不是磁盘所采用的 in-place 更新方式。它提供的垃圾回收机制，使得我们不需要马上对擦写越界的块进行擦写，而只需要对其设置一个标志，标明为“脏”块。当可用的块数不足时，垃圾回收机制才开始回收这些节点。同时，由于 JFFS2 基于日志结构，在意外掉电后仍然可以保持数据的完整性，而不会丢失数据。因此，JFFS2 成为了目前 Flash 上应用最广泛的文件系统。

然而，JFFS2 挂载时需要扫描整块 Flash 以确定节点的合法性以及建立必要的数据结构，这使得 JFFS2 挂载时间比较长。又由于 JFFS2 将节点信息保存在内存中，使得它所占用的内存量和节点数目成正比。再者，由于 JFFS2 通过随机方式来实现磨损平衡，它不能保证磨损平衡的确定性。因此，人们提出了 JFFS3，它就是为解决 JFFS2 的这些缺陷而设计的。

和 CramFS 一样，也存在一个制作 JFFS2 文件系统的工具 mkfs.jffs2（包含在 mtd-utils 中），执行如下命令即可生成所要的映象：

```
./mkfs.jffs2 -d my_jffs2/  -o jffs2.img （my_jffs2是我们要制作映像的目录）
```

使用 mkfs.jffs2 制作映像的时候，要注意指定正确的擦除块打下和页面大小，对于 NAND Flash，应使用“-n”去掉生成映像中的 clean marker。

接下来将 jffs2.img 复制到 Flash 第 1 个分区（复制到 MTD 字符设备），如下所示：

```
cp jffs2.img /dev/mtd1
```

之后，就可以将对应的块设备 mount 到 Linux 的目录了，如下所示：

```
mount -t jffs2 /dev/mtdblock1 /mnt/nor
```

对于 NAND Flash，应使用 mtd-utils 中的 nandwrite 工具进行 jffs2 映像向 NAND Flash 的烧录，在烧录前可以使用 Flash_eraseall 擦除 Flash，并在 OOB 区域加上 JFFS2 需要的 clean marker。

mtd-utils 的下载地址为：

ftp://ftp.infradead.org/pub/mtd-utils/mtd-utils-1.0.0.tar.gz。

19.6.4 YAFFS/YAFFS2

YAFFS（Yet Another Flash File System，http://www.yaffs.net）文件系统是专门针对 NAND 闪存设计的嵌入式文件系统，目前有 YAFFS 和 YAFFS2 两个版本，两个版本的主要区别之一在于 YAFFS2 能够更好地支持大容量的 NAND Flash 芯片，而前者只针对页大小为 512 字节的 NAND。

YAFFS 文件系统有些类似于 JFFS/JFFS2 文件系统，与之不同的是 JFFS1/2 文件系统最初是针对 NOR Flash 的应用场合设计的，而 NOR Flash 和 NAND Flash 本质上有较大的区别，所以尽管 JFFS1/2 文件系统也能应用于 NAND Flash，但由于它在内存占用和启动时间方面针对 NOR 的特性做了一些取舍，所以对 NAND 来说通常并不是最优的方案。NAND 上的每一页数据都有额外的空间用来存储附加信息，YAFFS 正好利用了该空间中一部分来存储文件系统相关的内容。

YAFFS 和 JFFS 都提供了写均衡、垃圾收集等底层操作，它们的不同之处如下。

- JFFS 是一种日志文件系统，通过日志机制保证文件系统的稳定性。YAFFS 仅仅借鉴了日志系统的思想，不提供日志机能，所以稳定性不如 JFFS，但是资源占用少。
- JFFS 中使用多级链表管理需要回收的脏块，并且使用系统生成伪随机变量决定要回收的块，通过这种方法能提供较好的写均衡，在 YAFFS 中是从头到尾对块搜索，所以在垃圾收集上 JFFS 的速度慢，但是能延长 NAND 的寿命。
- JFFS 支持文件压缩，适合存储容量较小的系统；YAFFS 不支持压缩，更适合存储容量较大的系统。
- YAFFS 还带有 NAND 芯片驱动，并为嵌入式系统提供了直接访问文件系统的 API，用户可以不使用 Linux 中的 MTD 和 VFS，直接对文件进行操作（如图 19.8）。尽管如此，NAND Flash 大多还是采用 MTD+YAFFS 的模式。

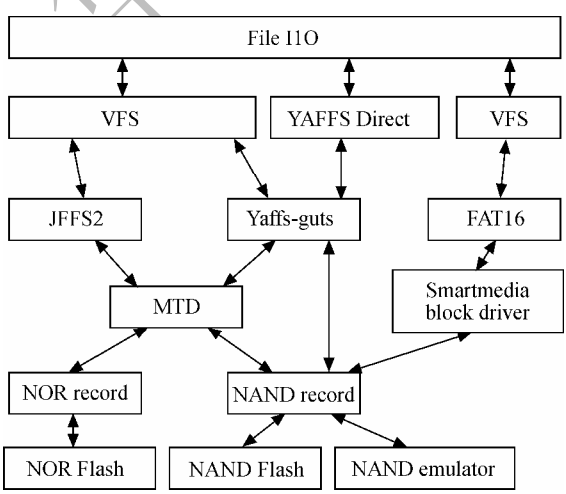


图 19.8 JFFS2 和 YAFFS

- YAFFS 使用 OOB 组织文件结构信息，而 JFFS 直接将节点信息保存在 NAND 数据区域里面，因此 YAFFS 在 mount 时只需读取 OOB，其 mount 时间远小于 JFFS。

在 Linux 2.6 内核下，YAFFS 文件系统的移植非常简单，主要包括以下工作。

（1）复制 YAFFS 源代码到 Linux 源码树。

只需要在内核中建立 YAFFS 目录 fs/yaffs，并把下载的 YAFFS 代码复制到该目录下面（YAFFS 代码包括 yaffs_ecc.c、yaffs_fileem.c、yaffs_fs.c、yaffs_guts.c、yaffs_mtdif.c、yaffs_ramem.c），下载的 YAFFS 源代码已包含了 Kconfig 和 Makefile，因此只需要修改 fs 目录下的 Kconfig 和 Makefile 并引用 fs/yaffs 中的对应文件即可，方法是：在 fs/Kconfig 中增加 source "fs/yaffs/Kconfig"；在 fs/Makefile 中增加 obj-$(CONFIG_YAFFS_FS) += yaffs/。

（2）配置内核编译选项。

在采用 make menuconfig 等方式配置内核编译选项时，除了应该选中 MTD 系统及目标板上 Flash 的驱动以外，也必须选中对 YAFFS 的支持，如下所示：

```
File systems  --->
 Miscellaneous filesystems  --->
  <*> Yet Another Flash Filing System(YAFFS) file system support
  [*]   NAND mtd support
  [*]   Use ECC functions of the generic MTD-NAND driver
  [*]   Use Linux file caching layer
  [*]   Turn off debug chunk erase check
  [*]   Cache short names in RAM
```

（3）挂载 YAFFS。

YAFFS 源代码包的 utils 目录下包含了 mkyaffs 工具，可以用它格式化 Flash，例如运行 mkyaffs /dev/mtd3 将用 YAFFS 文件系统格式化 NAND 的第 3 个分区，之后我们可以运行如下命令挂载 YAFFS：

```
mount –t yaffs /dev/mtdblock3 /mnt/nand
```

此后，对/mnt/nand 的操作就是对/dev/mtdblock3 的操作。

如果运行如下命令将根文件系统复制到/mnt/Flash0：

```
mount –t yaffs /dev/mtdblock3 /mnt/nand
cp (our_rootfs) /mnt/Flash0
umount /mnt/nand
```

则重新启动，并修改 Linux 启动参数中 root 为（仅仅是举例，具体系统的启动命令行很可能会有改变）：

```
param set linux_cmd_line "noinitrd root=/dev/mtdblock3 init=/linuxrc
console=ttyS0"
```

之后就可以直接以/dev/mtdblock3 中的根文件系统启动了。

YAFFS2 的移植方法与 YAFFS 类似，而且，目前我们已经没有必要再移植 YAFFS 了，因为 YAFFS2 的源码直接包含了对 YAFFS 的支持。

YAFFS 源代码的下载地址为：

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs.tar.gz?view=tar

YAFFS2 源代码的下载地址为：

http://www.aleph1.co.uk/cgi-bin/viewcvs.cgi/yaffs2.tar.gz?view=tar

YAFFS2 源代码包的 utils 目录下包含了 mkyaffsimage、mkyaffs2image 工具的源代码，编译即可生成 mkyaffsimage、mkyaffs2image 工具。YAFFS 不支持大页的 NAND Flash，一般只用于页大小为 512Byte 的 NAND Flash，对于页大小为 2KB 的 NAND Flash 而言，只能使用 YAFFS2。

运行“mkyaffsimage dir imagename”可以制作出 YAFFS 文件系统的镜像。需要注意的是，使用 mkyaffsimage 制作出来的 YAFFS 映像文件与通常的文件系统的映像文件不同，因为在 image 文件里除了以 512 字节为单位的一个页的数据外，同时紧跟在后还包括了 16 字节为单位的 NAND OOB 数据，因此，YAFFS 映像的下载工具必须将映像中的额外数据写入到 NAND 的 OOB 中。

nandwrite 工具“名义上”可以支持 YAFFS 映像的烧录，但是考虑到实际上 NAND ECC 模式和分布的不确定性，而 mkyaffsimage 生成的映像采用固定的 OOB 区域分布，因此，为了完全支持自适应的 YAFFS 映像烧录，必须在烧录工具中先解析 NAND 驱动的 OOB layout，再将记录在 YAFFS 映像中的 OOB 数据转换为适合于在系统中 NAND 存放的形式。

对于各种文件系统而言，如果想在启动过程中将 Flash 的 xxxfs 文件系统分区挂载到某个目录，可以通过修改启动的 rc 脚本或/etc/fstab 来完成，需在启动 rc 脚本中增加“mount -t xxxfs /dev/mtdblock3 /mnt/Flash0”类似语句，或在 fstab 中增加“/dev/mtdblock3 /mnt/Flash0 xxxfs defaults 0 0”类似语句。

19.7 总结

本章主要讲解了 Linux 系统中 MTD 系统的层次和接口，NOR 和 NAND Flash 驱动的设计方法及如何在其上建立 Flash 文件系统。

由于引入了 MTD 系统以及 MTD 下层的通用 NOR 和 NAND 驱动，Linux 中 NOR 和 NAND Flash 芯片级驱动的设计难度甚至要低于一个普通的 GPIO 字符设备。

在串口驱动部分，本章讲解了 tty_driver 到 uart_driver 的角色转换，在 Flash 驱动中，本章讲解了 mtd_info 向 map_info/nand_chip 的转移，可以说，Linux 驱动的这种分层设计思想是贯穿各种 Linux 驱动框架始终的。