



# 离线烧录 开发指南

版本号: 1.0  
发布日期: 2021.04.08

## 版本历史

版本号	日期	制/修订人	内容描述
1.0	2021.04.08	AWA1669	建立初始版本



# 目 录

<b>1 概述</b>	<b>1</b>
1.1 编写目的	1
1.2 适用范围	1
1.3 相关人员	1
<b>2 总体烧写流程</b>	<b>2</b>
<b>3 逻辑/物料地址</b>	<b>3</b>
3.1 地址转换：sector 地址 -> 逻辑页地址 -> 物理页地址	3
3.2 逻辑区读写规则	4
3.3 写逻辑页规则	4
<b>4 uboot</b>	<b>5</b>
4.1 input file	5
4.2 phyinfo_buf->factory_block	7
4.3 phyinfo_buf->mbr	7
4.4 phyinfo_buf->partition	8
4.5 physical block layout	8
4.6 burn uboot + phyinfo_buf	8
<b>5 boot0</b>	<b>9</b>
5.1 input file	9
5.2 flow	9
5.3 normal/secure boot0	10
5.4 filling storage_data	12
5.5 update checksum	13
5.6 burn boot0	13
<b>6 init secure storage block</b>	<b>15</b>
6.1 input file	15
<b>7 oob 结构</b>	<b>16</b>
7.1 oob without crc	16
7.2 oob with crc	17
<b>8 mapping page</b>	<b>19</b>
<b>9 mbr/gpt partition table</b>	<b>21</b>
9.1 物理/逻辑区大小	22
9.2 input file	22
9.3 flow	22
9.4 参考文件	23
9.5 参考函数流程	23

<b>10 partition image</b>	<b>24</b>
10.1 input file . . . . .	24
10.2 计算逻辑区域 LEB 总数 . . . . .	25
10.3 动态调整 sunxi_mbr 卷 . . . . .	26
10.4 根据 sunxi_mbr 动态生成 ubi layout volume . . . . .	26
10.5 烧写逻辑卷 . . . . .	27
10.6 ubi_ec_hdr . . . . .	28
10.7 ubi_vid_hdr . . . . .	28
10.8 数据对齐 . . . . .	29



## 插 图

3-1 spinand 物理布局 . . . . .	3
4-1 boot_info-without-enable_crc . . . . .	5
4-2 boot_info-with-enable_crc . . . . .	6
5-1 boot0_head . . . . .	9
5-2 boot_head . . . . .	10
5-3 storage_data . . . . .	12
7-1 oob-structure-example . . . . .	16
7-2 Winbond-Spinand-Area-Structure . . . . .	17
7-3 Winbond-Spare-Area-Example . . . . .	17
7-4 crc-value . . . . .	18
8-1 Mapping-Page-Data . . . . .	19
8-2 oob-structure-example . . . . .	20
9-1 GPT-Scheme . . . . .	21
10-1 PEB-LEB . . . . .	25



# 1 概述

---

## 1.1 编写目的

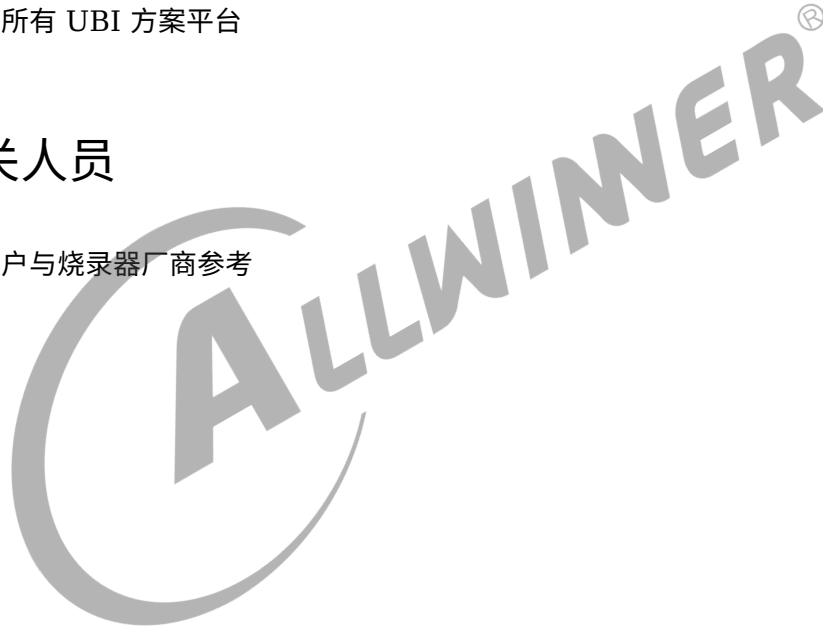
介绍 Sunxi SPINand 烧写时的数据布局

## 1.2 适用范围

本设计适用于所有 UBI 方案平台

## 1.3 相关人员

制定烧录器客户与烧录器厂商参考



## 2 总体烧写流程

- 根据 spinand device datasheet 中的块页信息确定 boot0, uboot, secure storage block 及预留物理区大小和可用逻辑区大小。

```
/* small nand:block size < 1MB; reserve 4M for uboot */
if (blksize <= SZ_128K) {
    _start = UBOOT_START_BLOCK_SMALLNAND;
    _end = _start + 32;
} else if (blksize <= SZ_256K) {
    _start = UBOOT_START_BLOCK_SMALLNAND;
    _end = _start + 16;
} else if (blksize <= SZ_512K) {
    _start = UBOOT_START_BLOCK_SMALLNAND;
    _end = _start + 8;
} else if (blksize <= SZ_1M && pagecnt <= 128) { //1M
    _start = UBOOT_START_BLOCK_SMALLNAND;
    _end = _start + 4;
}
/* big nand; reserve at least 20M for uboot */
} else if (blksize <= SZ_1M && pagecnt > 128) {
    _start = UBOOT_START_BLOCK_BIGNAND;
    _end = _start + 20;
} else if (blksize <= SZ_2M) {
    _start = UBOOT_START_BLOCK_BIGNAND;
    _end = _start + 10;
} else {
    _start = UBOOT_START_BLOCK_BIGNAND;
    _end = _start + 8;
}
}

if (CONFIG_AW_MTD_SPINAND_UBOOT_BLKES > 0)
    _end = _start + CONFIG_AW_MTD_SPINAND_UBOOT_BLKES;
```

- 根据 uboot\_start、uboot\_end 生成的 mbr, partition 信息
- 获得逻辑区起始 block 地址后，扫描出厂坏块（factory bad block），出厂坏块以逻辑块为单位
- 根据出厂坏块信息和分区表信息确定最后一个 UDISK 分区大小。
- 写 GPT 分区表和各个分区

1. 每个逻辑页的 oob 区域格式见 6 **oob 结构**
2. 每个逻辑块的尾页存放逻辑页和物理页的映射信息，其 data 区和 oob 区格式见 6 **oob 结构** 和 7 **mapping page**

- 写 uboot, 见 3
- 写 boot0, 见 4

## 3 逻辑/物料地址

### 3.1 地址转换：sector 地址 -> 逻辑页地址 -> 物理页地址

**sector:** 大小 512 bytes。

**物理页:** 大小需看 spinand device data sheet, 常见为 2K, 4 个 sector。

**逻辑页:** 大小与物理页呈倍数关系, 常见为 2 倍, 即从 2 个相邻物理块 (block[2N], block[2N+1], N 最大值为总物理块数的一半, 再减 1, 常见为 511, 即  $1024/2 - 1$ ) 各取一页, 组成逻辑页。

**boot0, uboot, secure storage block 读写单元为物理页, 不存在逻辑与物理地址转换关系**  
**分区表及各个分区读写单元为逻辑页, 存在逻辑地址与物理地址转换关系**

例如, 常见 spinand device 有 1024 个块 (编号 block0, block1, ..., block1023), 每个 block 有 64 个 page (编号 page0, page1, ..., page63), 每个 page 有 2048 字节 (4 个 sector, sector 大小为 512 字节)。

block0 到 block48 用于存放 boot0, uboot, secure storage block (见 physical block layout), 读写单元为物理页。

block49~block1023 用于存放逻辑分区, 以逻辑块为擦除单元, 逻辑页为读写单元, 逻辑块由相邻的 2 个 block 组成, 逻辑页由相邻 block 内 page 组成。例如 block66 和 block67 组成逻辑块, block66 的 page0 和 block67 的 page0 组成逻辑页 0。

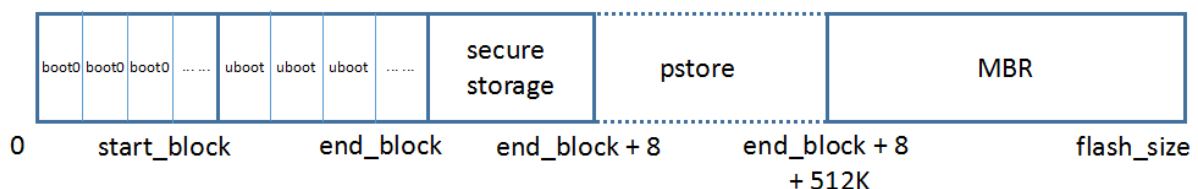


图 3-1: spinand 物理布局

以 block size = 128k 为例, start block = 8, end\_block = 32, Boot0 占 8 个 block 空间, 当有足够空间时, 会继续烧写备份 boot0 数据, Uboot 也是一样。



## 3.2 逻辑区读写规则

1. 选择逻辑块：从最大逻辑块号开始写，即倒序，（对于 1024 块的 block, 最大逻辑块号为 511）。检查是否是坏块，若是坏块，则跳过该块，选择下一个块（当前块号减 1）。
  2. 选定逻辑块后，从逻辑 page0 开始顺序写，即写入顺序 page0, page1, page2, .....
- 如果当前待写入页不是尾页，则写入逻辑 data + oob, 参考 1.6
  - 如果当前待写入页是尾页，则写入 mapping data + oob（参考 1.6 和 1.7），跳步骤 1，选择新逻辑块

## 3.3 写逻辑页规则

假如逻辑区 4k 数据 4k (top-half 2k, bottom-half 2k) + 16bytes OOB(oob 结构见 1.6), 逻辑块号 M, 页号 N, 写入过程如下

- 将 2k 上半部数据 top-half2k + 16bytes OOB 数据写入块号 2M, 页号 N
- 将 2k 下半部数据 bottom-half2k + 16bytes OOB 数据写入块号 2M+1, 页号 N

## 4 uboot

### 4.1 input file

static **boot\_package.fex/ toc1.fex** + dynamic **physic\_info** (安全方案: toc1.fex, 非安全方案: boot\_package.fex)

struct \_boot\_info physic\_info 信息在烧写 uboot 前会附加到 uboot 尾部, 该信息是动态生成的。

```
struct _boot_info{
    unsigned int magic;
    unsigned int len;
    unsigned int sum;

    unsigned int no_use_block;
    unsigned int uboot_start_block;
    unsigned int uboot_next_block;
    unsigned int logic_start_block;
    unsigned int nand_specialinfo_page;
    unsigned int nand_specialinfo_offset;
    unsigned int physic_block_reserved;
    unsigned int nand_ddrtype;
    unsigned int ddr_timing_cfg;
    unsigned int nouse[128-12];

    _MBR mbr; //4k offset 0.5k
    _PARTITION partition; //2.5k offset 4.5k
    _NAND_STORAGE_INFO storage_info; //0.5k offset 7k
    _FACTORY_BLOCK factory_block; //2k offset 7.5k
    // _UBOOT_INFO uboot_info; //0.25K
    _NAND_SPECIAL_INFO nand_special_info; //1k offset 9.5k
} ? end_boot_info ? ;
```

图 4-1: boot\_info-without-enable\_crc

```

struct _boot_info{
    unsigned int magic;
    unsigned int len;
    unsigned int sum;

    unsigned int no_use_block;
    unsigned int uboot_start_block;
    unsigned int uboot_next_block;
    unsigned int logic_start_block;
    unsigned int nand_specialinfo_page;
    unsigned int nand_specialinfo_offset;
    unsigned int physic_block_reserved;
    unsigned int nand_ddrtype;
    unsigned int ddr_timing_cfg;
    unsigned int enable_crc; // ENABLE_CRC_MAGIC
    unsigned int nouse[128-13];

    _MBR mbr; //4k offset 0.5k
    _PARTITION partition; //2.5k offset 4.5k
    _NAND_STORAGE_INFO storage_info; //0.5k offset 7k
    _FACTORY_BLOCK factory_block; //2k offset 7.5k
    // _UBOOT_INFO uboot_info; //0.25K
    _NAND_SPECIAL_INFO nand_special_info; //1k offset 9.5k
} ? end _boot_info ? ;
    
```

图 4-2: boot\_info-with-enable\_crc

attribute name	type	value	comment
magic	unsigned int	0xaa55a5a5	
len	unsigned int	32768	
sum	unsigned int	动态计算校验和	
no_use_block	unsigned int	20	值与 logic_start_block 相同
uboot_start_block	unsigned int	8	
uboot_next_block	unsigned int	32	
logic_start_block	unsigned int	20	在无坏块理想情况下
nand_specialinfo_page	unsigned int	0	
nand_specialinfo_offset	unsigned int	0	
physic_block_reserved	unsigned int	6	
nand_ddrtype	unsigned int	0	
ddr_timing_cfg	unsigned int	0	
enable_crc	unsigned int	0x63726365	如果 oob 有 crc 域，则赋值为 0x63726365
mbr	_MBR		
partition	_PARTITION		
storage_info	_NAND_STORAGE_INFO	ignore, fill 0	
factory_block	_FACTORY_BLOCK	动态扫描填充	

attribute name	type	value	comment
nand_special_info	_NAND_SPECIAL_INFO	ignore, fill 0	

## 4.2 phyinfo\_buf->factory\_block

```
struct _nand_super_block{
    unsigned short  Block_NO;
    unsigned short  Chip_NO;
};
typedef union{
    unsigned char  ndata[2048];
    struct _nand_super_block data[512];
}_FACTORY_BLOCK;
```

## 4.3 phyinfo\_buf->mbr

```
/* part info */
typedef struct _NAND_PARTITION{
    unsigned char  classname[PARTITION_NAME_SIZE];
    unsigned int   addr;
    unsigned int   len;
    unsigned int   user_type;
    unsigned int   keydata;
    unsigned int   ro;
}NAND_PARTITION;    //36bytes

/* mbr info */
typedef struct _PARTITION_MBR{
    unsigned int   CRC;
    unsigned int   PartCount;
    NAND_PARTITION array[ND_MAX_PARTITION_COUNT]; //
}PARTITION_MBR;

typedef union{
    unsigned char  ndata[4096];
    PARTITION_MBR data;
}_MBR;
```

参考函数：

- nand\_get\_mbr 函数将 sunxi\_mbr.fex 中的 sunxi\_mbr\_t 类型数据转化成 PARTITION\_MBR 类型数据
- NAND\_UbootInit -> NAND\_LogicInit -> nand\_info\_init\_4\_offline\_burn 函数将 PARTITION\_MBR 类型数据拷贝至 phyinfo\_buf->mbr

## 4.4 phyinfo\_buf->partition

```
struct _partition{
    struct _nand_disk nand_disk[MAX_PART_COUNT_PER_FTL];
    unsigned int size;
    unsigned int cross_talk;
    unsigned int attribute;
    struct _nand_super_block start;
        struct _nand_super_block end;
    //unsigned int offset;
};
typedef union{
    unsigned char ndata[2048+512];
    struct _partition data[MAX_PARTITION];
}_PARTITION;
```

## 4.5 physical block layout

- 确定 boot0, uboot, reserve block 布局, 参考 set\_uboot\_start\_and\_end\_block // boot0, uboot, reserve
- 确定 secure storage block, 参考 nand\_info\_init -> nand\_secure\_storage\_first\_build // secure storage block
- 烧写分区表, 参考 sunxi\_sprite\_download\_mbr -> download\_standard\_gpt // gpt partition table + back-up gpt
- burn partitions: 根据分区表信息烧写, sector 偏移地址 + 长度 (sector 单位)

## 4.6 burn uboot + phyinfo\_buf

写 uboot + phyinfo\_buf 的 page 时, oob 区域值为 ff 00 03 01 ff ff ff ff ff ff ff ff ff ff ff

注意事项:

- 单个备份按 block 对齐, 若写单个备份过程中遇到坏块, 则跳过该坏块, 写入下一好块, 直到将当前备份完整写入
- phyinfo\_buf 与 uboot 存储区域按 page 对齐
- 写完若干数量备份后, 若剩余 block 数量不够写一个完整备份时, 则可以空着不写

## 5 boot0

### 5.1 input file

boot0\_nand.fex (非安) /toc0.fex (安全)

### 5.2 flow

- 验证 checksum 是否准确
- 填充 storage\_data
- 重新生成 checksum 并更新 boot\_file\_head\_t 中的 check\_sum

```
typedef struct boot0_file_head_t
{
    boot_file_head_t boot_head;
    boot0_private_head_t prvt_head;
    char hash[64];
} boot0_file_head_t;
```

图 5-1: boot0\_head

```

#define BOOT0_MAGIC          "eGON.BT0"
#define SYS_PARA_LOG        0x4d415244

/*****
/*          file head of Boot          */
*****/
typedef struct _Boot_file_head
{
    __u32 jump_instruction; /* one intruction jumping to real code */
    __u8  magic[MAGIC_SIZE]; /* ="eGON.BT0" */
    __u32 check_sum;        /* generated by PC */
    __u32 length;          /* generated by PC */
    __u32 pub_head_size;   /* the size of boot_file_head_t */
    __u8  pub_head_vsn[4]; /* the version of boot_file_head_t */
    __u32 ret_addr;        /* the return value */
    __u32 run_addr;        /* run addr */
    __u32 boot_cpu;        /* eGON version */
    __u8  platform[8];     /* platform information */
}boot_file_head_t;

```

图 5-2: boot\_head

参考文件

include/private\_boot0.h

sprite/sprite\_download.c

参考函数

download\_normal\_boot0

download\_secure\_boot0

## 5.3 normal/secure boot0

非安与安全对应的文件分别是 boot0\_nand.fex/toc0.fex，非安 boot0 头部结构 \_boot0\_file\_head\_t，安全方案 boot0 头部结构 sbrom\_toc0\_config

```

typedef struct _boot0_file_head_t
{
    boot_file_head_t      boot_head;
    boot0_private_head_t  prvt_head;
    char hash[64];
    __u8                  reserved[8];
    union {
#ifdef CFG_SUNXI_SELECT_DRAM_PARA
        boot_extend_head_t  extd_head;
#endif
    };
}boot0_file_head_t;

```

```
    fes_aide_info_t fes1_res_addr;

    } fes_union_addr;
}boot0_file_head_t;
```

```
typedef struct sbrom_toc0_config
{
    unsigned char    config_vsn[4];
    unsigned int     dram_para[32];    // dram参数
    int              uart_port;        // UART控制器编号
    normal_gpio_cfg  uart_ctrl[2];     // UART控制器GPIO
    int              enable_jtag;      // JTAG使能
    normal_gpio_cfg  jtag_gpio[5];     // JTAG控制器GPIO
    normal_gpio_cfg  storage_gpio[50]; // 存储设备 GPIO信息
                                        // 0-23放nand, 24-31存放卡0, 32-39放卡2
                                        // 40-49存放spi
    char             storage_data[384]; // 0-159,存储nand信息; 160-255,存放卡信息
    unsigned int     secure_dram_mbytes; //
    unsigned int     drm_start_mbytes;   //
    unsigned int     drm_size_mbytes;    //
    unsigned int     boot_cpu;           //
    special_gpio_cfg a15_power_gpio;     //the gpio config is to a15 extern power enable
    gpio
    unsigned int     next_exe_pa;
    unsigned int     secure_without_0S;  //secure boot without semelis
    unsigned char    debug_mode;         //1:turn on printf; 0 :turn off printf
    unsigned char    power_mode;         /* 0:axp , 1: dummy pmu */
    unsigned char    rotpk_flag;
    unsigned char    reserver[1];
    unsigned int     card_work_mode;
    unsigned int     res[2];             // 总共1024字节
}
sbrom_toc0_config_t;
```



## 5.4 filling storage\_data

```

typedef struct
(
    __u8    ChipCnt;           // the count of the total nand flash chips are currently connecting on the CE pin
    __u8    ConnectMode;     // the rb connect mode
    __u8    BankCntPerChip;  // the count of the banks in one nand chip, multiple banks can support Inter-Leave
    __u8    DieCntPerChip;   // the count of the dies in one nand chip, block management is based on Die
    __u8    PlaneCntPerDie;  // the count of planes in one die, multiple planes can support multi-plane operation
    __u8    SectorCntPerPage; // the count of sectors in one single physic page, one sector is 0.5k
    __u16   ChipConnectInfo; // chip connect information, bit == 1 means there is a chip connecting on the CE pin
    __u32   PageCntPerPhyBlk; // the count of physic pages in one physic block
    __u32   BlkCntPerDie;    // the count of the physic blocks in one die, include valid block and invalid block
    __u32   OperationOpt;    // the mask of the operation types which current nand flash can support support
    __u32   FrequencePar;    // the parameter of the hardware access clock, based on 'MHz'
    __u32   SpiMode;        // spi nand mode, 0mode 0, 3mode 3
    __u8    NandChipId[8];   // the nand chip id of current connecting nand chip
    __u32   pagewithbadflag; // bad block flag was written at the first byte of spare area of this page
    __u32   MultiPlaneBlockOffset; // the value of the block number offset between the two plane block
    __u32   MaxEraseTimes;  // the max erase times of a physic block
    __u32   MaxEccBits;     // the max ecc bits that nand support
    __u32   EccLimitBits;  // the ecc limit flag for the nand
    __u32   uboot_start_block;
    __u32   uboot_next_block;
    __u32   logic_start_block;
    __u32   nand_specialinfo_page;
    __u32   nand_specialinfo_offset;
    __u32   physic_block_reserved;
    __u32   Reserved[4];
} boot_spinand_para_t;

```

图 5-3: storage\_data

下表中红色字体不能配置错，大部分值直接参考 drivers/mtd/awnand/spinand/physic/id.c

<i>*attribute name*</i>	<i>*type*</i>	<i>*value*</i>	<i>*comment*</i>
ChipCnt	unsigned char	1	
ConnectMode	unsigned char	1	忽略，可以不用理解
BankCntPerChip	unsigned char	1	忽略，可以不用理解
DieCntPerChip	unsigned char	1	
PlaneCntPerDie	unsigned char	2	忽略，可以不用理解
SectorCntPerPage	unsigned char	4	以具体物料为准，常见为 4
ChipConnectInfo	unsigned short	1	忽略，可以不用理解
PageCntPerPhyBlk	unsigned int	64	以具体物料为准，常见为 64
BlkCntPerDie	unsigned int	1024	以具体物料为准，常见为 1024，也可能为 512 或 2
OperationOpt	unsigned int	0x?	参考 id.c 各个物料配置
FrequencePar	unsigned int	100	忽略，可以不用理解
SpiMode	unsigned int	0	忽略，可以不用理解
NandChipId[8]	unsigned char	0x?	参考 id.c
pagewithbadflag	unsigned int	0	忽略，可以不用理解
MultiPlaneBlockOffset	unsigned int	1	忽略，可以不用理解
MaxEraseTimes	unsigned int		忽略，可以不用理解
EccLimitBits	unsigned int		忽略，可以不用理解
uboot_start_block	unsigned int	8	
uboot_next_block	unsigned int	58	

<i>*attribute name*</i>	<i>*type*</i>	<i>*value*</i>	<i>*comment*</i>
logic_start_block	unsigned int	24	忽略, 可以不用理解
nand_specialinfo_page	unsigned int	0	忽略, 可以不用理解
nand_specialinfo_offset	unsigned int	0	忽略, 可以不用理解
physic_block_reserved	unsigned int	6	忽略, 可以不用理解
Reserved[4]	unsigned int	0	忽略, 可以不用理解

以 GigaDevice GD5F1GQ4UBYIG spinand 为例, 其大部分信息直接来自 id.c

```
{
    .Model          = "GD5F1GQ4UBYIG",
    .NandID         = {0xc8, 0xd1, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff},
    .DieCntPerChip = 1,
    .SectCntPerPage = 4,
    .PageCntPerBlk = 64,
    .BlkCntPerDie  = 1024,
    .OobSizePerPage = 64,
    .OperationOpt  = SPINAND_QUAD_READ | SPINAND_QUAD_PROGRAM |
                    SPINAND_DUAL_READ,
    .MaxEraseTimes = 50000,
    .EccFlag       = HAS_EXT_ECC_SE01,
    .EccType       = BIT4_LIMIT5_TO_7_ERR8_LIMIT_12,
    .EccProtectedType = SIZE16_OFF4_LEN8_OFF4,
    .BadBlockFlag  = BAD_BLK_FLAG_FRIST_1_PAGE,
},
```

## 5.5 update checksum

参考文件:

sprite/sprite\_download.c

sprite/sprite\_verify.c

board/sunxi/board\_common.c

参考函数流程:

```
download_normal_boot0/download_secure_boot0 -> sunxi_sprite_generate_checksum
-> sunxi_generate_checksum
```

## 5.6 burn boot0

写 boot0 的 page 时, oob 区域值统一为 ff 00 03 01 ff ff ff ff ff ff ff ff ff ff ff

- 各个备份按 block 对齐（如果 boot0 超过 1 个 block, 单个备份起始 block 地址为偶数），若写单个备份过程中遇到坏块，则中止当前备份写过程，写下一备份即可
- boot0 的镜像文件已经包含了 boot0 header，不需额外分配组织 boot0 header 格式，只需更新 boot0 header 中的 storage\_data 部分，其他属性（比如 dram\_para）不需更新。更新后，需重新生成 boot0 header 中的校验和 check\_sum



## 6 init secure storage block

---

### 6.1 input file

无，在 uboot 的存放区域后 8 个 block 用于 secure storage block 用于存放 mac 地址等信息。需要将该块 oob 区域写入 ff aa 5c 00 00 12 34 ff ff ff ff。



## 7 oob 结构

### 7.1 oob without crc

oob area 共 16bytes, 布局如下:

oob[0]: bad block flag, 0xff 代表 good block, 否则 bad block

oob[1-4]: page feature flag, 大端存储, 逻辑页:  $\text{oob}[1] \& 0xf0 == 0xc0$ , 映射页: 0xaaaffff

oob[5-6]: erase count, 大端存储, 代表块的擦除次数, 烧录器方案中, 初始值均为 1

oob[7-10]: block used count, 大端存储, 第几次写块, 该值线性递增, 最小值为 0, 最大值为总 block 数 \* 每块可擦除次数, 例如: 从 block511 逐渐存放数据, block used count 为 0, 写 block510 时, 该值为 1, 一直顺序写到 block24, 该值为 487, 若再次写 block511, block used count 为 488。

oob[11-15]: reserved, 默认填充值 0xa5

```
[2019/3/14 14:30:08] p628b 511
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: ff c0 00 02 59 00 01 00 00 00 00 a5 a5 a5 a5
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] p638b 511
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: ff aa aa ff ff 00 01 00 00 00 00 a5 a5 a5 a5
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] dump main data
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: 00 00 00 00 bb 6d 00 00 bc 6d 00 00 bd 6d 00 00
[2019/3/14 14:30:08] 00000010: bf 6d 00 00 00 02 00 00 01 02 00 00 02 02 00 00
[2019/3/14 14:30:08] 00000020: 03 02 00 00 04 02 00 00 05 02 00 00 06 02 00 00
[2019/3/14 14:30:08] 00000030: 07 02 00 00 08 02 00 00 09 02 00 00 0a 02 00 00
[2019/3/14 14:30:08] 00000040: 0b 02 00 00 0c 02 00 00 0d 02 00 00 0e 02 00 00
[2019/3/14 14:30:08] 00000050: 0f 02 00 00 10 02 00 00 11 02 00 00 12 02 00 00
[2019/3/14 14:30:08] 00000060: 13 02 00 00 14 02 00 00 15 02 00 00 16 02 00 00
[2019/3/14 14:30:08] 00000070: 17 02 00 00 18 02 00 00 19 02 00 00 1a 02 00 00
[2019/3/14 14:30:08] 00000080: 1b 02 00 00 1c 02 00 00 1d 02 00 00 1e 02 00 00
[2019/3/14 14:30:08] 00000090: 1f 02 00 00 40 02 00 00 41 02 00 00 42 02 00 00
[2019/3/14 14:30:08] 000000a0: 43 02 00 00 44 02 00 00 45 02 00 00 46 02 00 00
[2019/3/14 14:30:08] 000000b0: 47 02 00 00 48 02 00 00 49 02 00 00 4a 02 00 00
[2019/3/14 14:30:08] 000000c0: 4b 02 00 00 4c 02 00 00 4d 02 00 00 4e 02 00 00
[2019/3/14 14:30:08] 000000d0: 4f 02 00 00 50 02 00 00 51 02 00 00 52 02 00 00
[2019/3/14 14:30:08] 000000e0: 53 02 00 00 54 02 00 00 55 02 00 00 56 02 00 00
[2019/3/14 14:30:08] 000000f0: 57 02 00 00 58 02 00 00 59 02 00 00 ff ff ff ff
[2019/3/14 14:30:08] 00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图 7-1: oob-structure-example

对于上图，逻辑块 511 的 page62，其 oob 信息表示：映射的逻辑页是 0x0000259，该块当前擦除次数 0x0001，block used count 为 0x00000000。

每款 spinand 物料存储组织方式是有差异的，具体需要参考物料 data sheet。以华邦 winbond w25n01gv 为例，其存储方式如下：

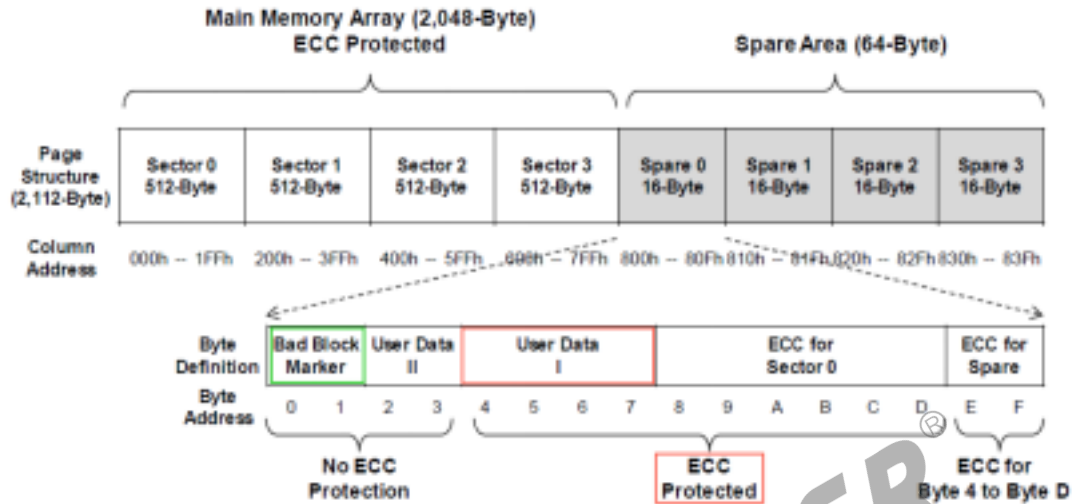


图 7-2: Winbond-Spinand-Area-Structure

如果 oob 信息 ff c0 00 04 30 00 04 00 00 23 75 a5 a5 a5 a5 a5，绿框标注坏块标记，红框区域标注着 oob 信息的存放位置（受 ecc 保护）。

最后再次说明每款物料，存放 oob 信息的位置是有差异的。

```

ff ff ff ff ff c0 00 04 75 aa 58 6c c5 56 f0 0f
ff ff ff ff 30 00 04 00 cb 4d 23 8a 58 77 84 7b
ff ff ff ff 00 23 75 a5 fd d5 5f f0 03 3f 15 15
ff ff ff ff a5 a5 a5 a5 d2 d9 62 bf 0e 14 60 60
    
```

图 7-3: Winbond-Spare-Area-Example

id.c 的 ID 表中 EccProtectedType，就是更具 Ecc Protected 区域的分布情况，选出 16byte 的数据空间，存放 oob 数据。

## 7.2 oob with crc

oob area 共 16bytes，布局如下：

oob[0]: bad block flag, 0xff 代表 good block，否则 bad block

oob[1-4]: page feature flag, 大端存储, 逻辑页: oob[1]&0xf0 == 0xc0, 映射页: 0xaaaffff

oob[5-6]: erase count, 大端存储, 代表块的擦除次数, 烧录器方案中, 初始值均为 1

oob[7-10]: block used count, 大端存储, 第几次写块, 该值线性递增, 最小值为 0, 最大值为总 block 数 \* 每块可擦除次数, 例如: 从 block511 逐渐存放数据, block used count 为 0, 写 block510 时, 该值为 1, 一直顺序写到 block24, 该值为 487, 若再次写 block511, block used count 为 488。

oob[11]: reserved, 默认填充值 0xa5

oob[12-13]: crc value, crc16 大端存储。

oob[14-15]: reserved, 默认填充值 0xa5

对于 block 的尾页 (最后一页), 根据 mapping page (page\_per\_block\*4, 对于 spinand, 大小为 256) 的数据计算 crc 值; 对于其它页, 则以写入该逻辑页的全部数据 (对于 spinand, 大小为 4k) 计算 crc 值。

```

p63@b 433
00000000: ff aa aa ff ff 0c ae 00 11 48 48 a5 e2 b3 a5 a5
p 0@b 432
00000000: ff c0 00 3e 9b 0c c1 00 11 47 6f a5 df fe a5 a5

```

图 7-4: crc-value



## 8 mapping page

mapping page 存放逻辑地址与物理地址映射信息 (mapping data)

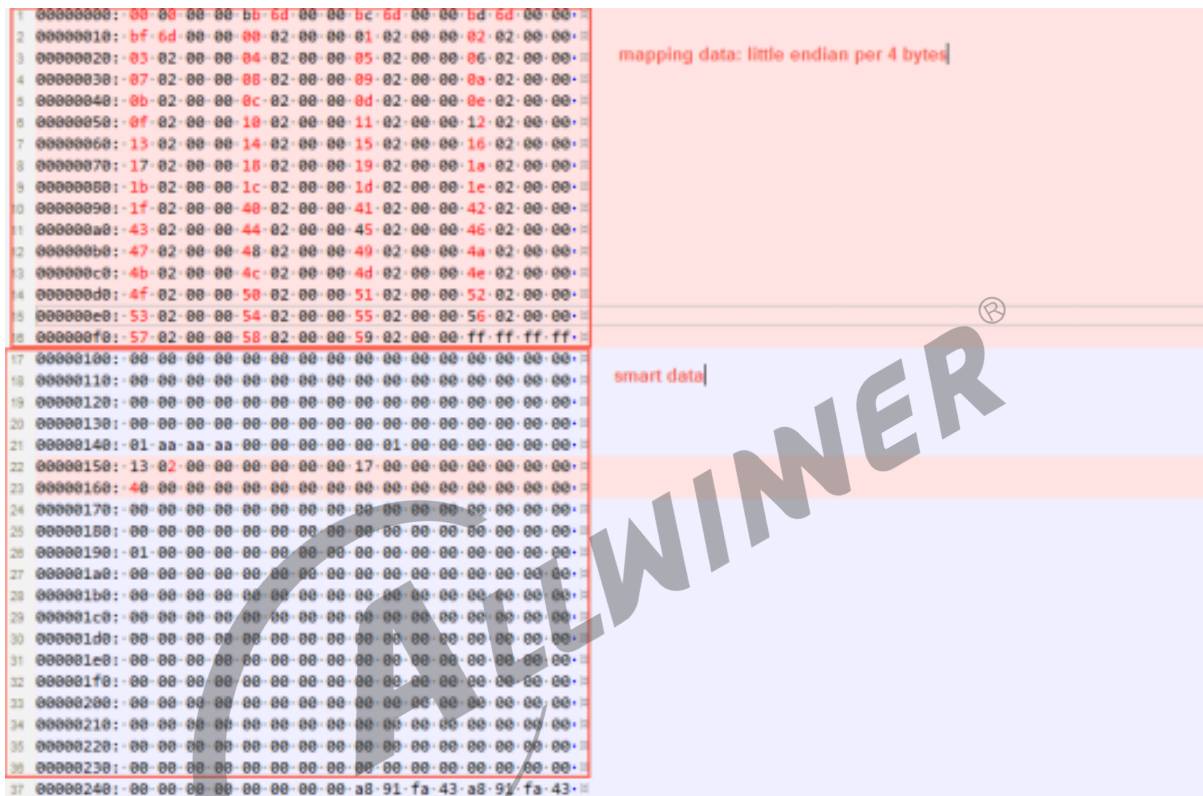


图 8-1: Mapping-Page-Data

mapping data : 每块 page 数 \*4 字节, 对于常见 spinand 一般为 256 字节, 每个索引存放该物理页映射的逻辑页信息, 比如索引 0, 即 page0 映射的逻辑页号 0x00000000 (小端格式), 其存放了 GPT 分区表信息, 对于索引 5, 即 page5 映射的逻辑页号 0x00000200, 其存放了分区 1 起始数据, page63 没有映射逻辑页, 因此存放 0xffffffff



```
[2019/3/14 14:30:08] p62@b 511
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: ff c0 00 02 59 00 01 00 00 00 00 a5 a5 a5 a5
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] p63@b 511
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: ff aa aa ff ff 00 01 00 00 00 00 a5 a5 a5 a5
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] dump main data
[2019/3/14 14:30:08]
[2019/3/14 14:30:08] 00000000: 00 00 00 00 bb 6d 00 00 bc 6d 00 00 bd 6d 00 00
[2019/3/14 14:30:08] 00000010: bf 6d 00 00 00 02 00 00 01 02 00 00 02 02 00 00
[2019/3/14 14:30:08] 00000020: 03 02 00 00 04 02 00 00 05 02 00 00 06 02 00 00
[2019/3/14 14:30:08] 00000030: 07 02 00 00 08 02 00 00 09 02 00 00 0a 02 00 00
[2019/3/14 14:30:08] 00000040: 0b 02 00 00 0c 02 00 00 0d 02 00 00 0e 02 00 00
[2019/3/14 14:30:08] 00000050: 0f 02 00 00 10 02 00 00 11 02 00 00 12 02 00 00
[2019/3/14 14:30:08] 00000060: 13 02 00 00 14 02 00 00 15 02 00 00 16 02 00 00
[2019/3/14 14:30:08] 00000070: 17 02 00 00 18 02 00 00 19 02 00 00 1a 02 00 00
[2019/3/14 14:30:08] 00000080: 1b 02 00 00 1c 02 00 00 1d 02 00 00 1e 02 00 00
[2019/3/14 14:30:08] 00000090: 1f 02 00 00 40 02 00 00 41 02 00 00 42 02 00 00
[2019/3/14 14:30:08] 000000a0: 43 02 00 00 44 02 00 00 45 02 00 00 46 02 00 00
[2019/3/14 14:30:08] 000000b0: 47 02 00 00 48 02 00 00 49 02 00 00 4a 02 00 00
[2019/3/14 14:30:08] 000000c0: 4b 02 00 00 4c 02 00 00 4d 02 00 00 4e 02 00 00
[2019/3/14 14:30:08] 000000d0: 4f 02 00 00 50 02 00 00 51 02 00 00 52 02 00 00
[2019/3/14 14:30:08] 000000e0: 53 02 00 00 54 02 00 00 55 02 00 00 56 02 00 00
[2019/3/14 14:30:08] 000000f0: 57 02 00 00 58 02 00 00 59 02 00 00 ff ff ff ff
[2019/3/14 14:30:08] 00000100: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图 8-2: oob-structure-example

对于上图，逻辑块 511 的 page62，索引 62 处存放的数据为 0x00000259，表示该块的 62 页存放的是逻辑页号是 0x00000259，索引 63 处存放的数据为 0xffffffff

## 9 mbr/gpt partition table

需要根据 sunxi\_mbr.fex 中的内容转换为上图的格式，然后写入 LBA0\_LBA33, LBA-33\_LBA-2, LBA-1 区域。LBA(logical block address) 可以理解为扇区 sector，大小 512。

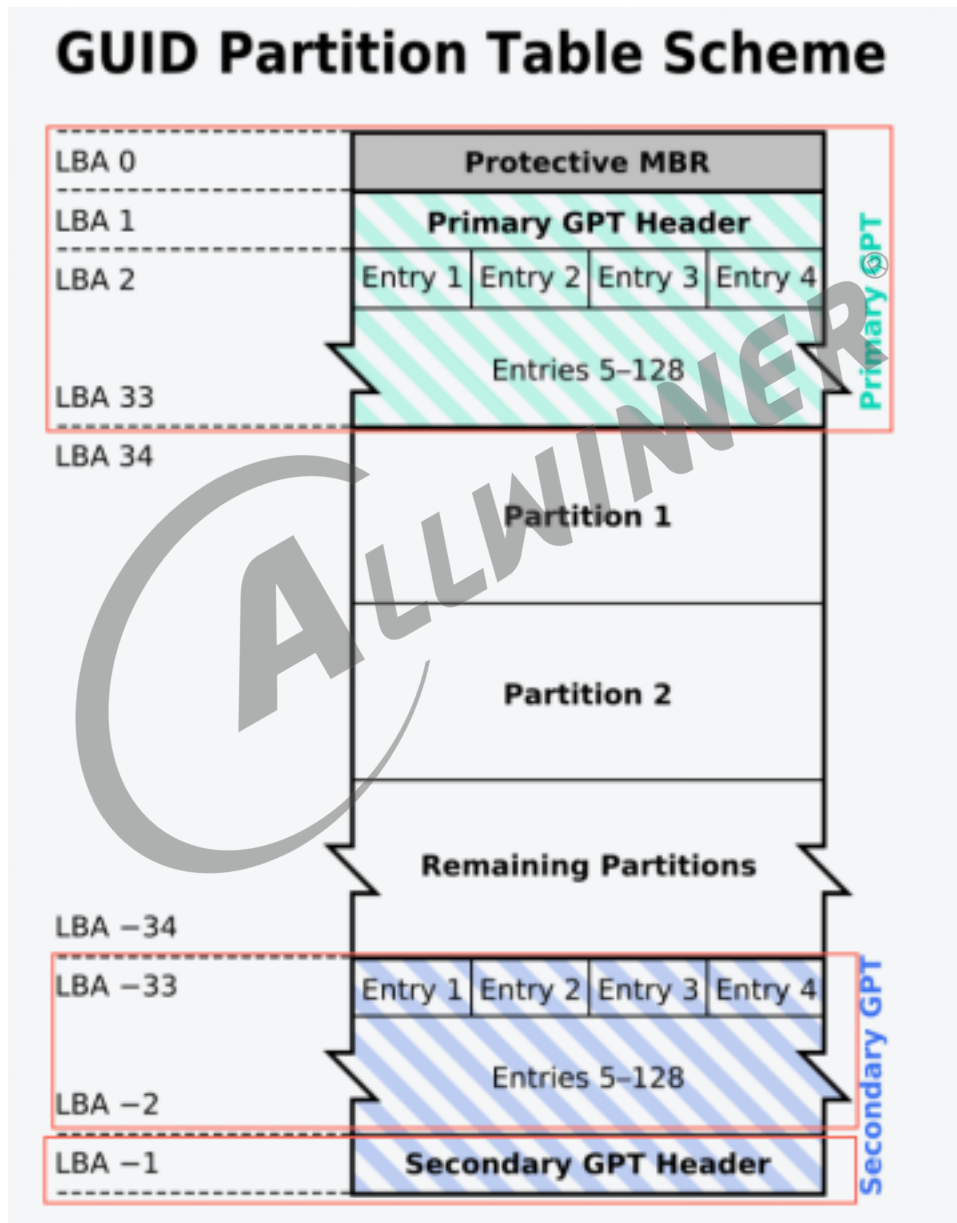


图 9-1: GPT-Scheme

## 9.1 物理/逻辑区大小

但是逻辑区到底有多大，按如下方法推算：

减 boot0 大小：8 个物理 block (block0-block7)

减 uboot 大小：32 个物理 block (block8-block31)

减 secure storage block 区域大小：理想情况下为 2 个连续物理 block (block32, block41)，如果有坏块，可能会多于 2 个，例如，若 block41 为坏块，则 secure storage block 为 block40, block42

减预留物理 block 大小：6 个，理想情况下为 block42-block47，若 secure storage block 区域中碰到坏块，导致这个物理区 block 数为奇数个，则再加一个物理 block，向上取至偶数个，例如若 block41 为坏块，则为 block43-48 + block49。

减出厂逻辑坏块数：每个 spinand 片子不一样，需要动态扫描

减去预留逻辑块数：对于有 1024 个物理 block 的 spinand 片子，预留 40 个逻辑 block，即 80 个物理 block；对于有 2048 个物理 block 的 spinand 片子，预留 85 个逻辑 block

综上所述，对于有 1024 个物理 block 的 spinand 片子，理想情况下（无出厂坏块），逻辑区大小为  $(1024 - 8 - 32 - 2 - 6 - 40 * 2) = 896$ ，每个 block128K，2 个 LBA(sector)，所以逻辑区 LBA(sector) 个数为  $896 * 128 * 2 = 229376$

因此，实际上 LBA-33, LBA-1 这 2 个 LBA 地址的实际值可能因 spinand 片子而定。

## 9.2 input file

sunxi\_mbr.fex

## 9.3 flow

1. 校验 sunxi\_mbr.fex
2. 调用 nand\_get\_mbr 将 sunxi\_mbr\_t 转换为 PARTITION\_MBR
3. erase\_flash
4. sunxi\_sprite\_download\_mbr(crc32 校验值计算请参考 util/crc32.c)

## 9.4 参考文件

drivers/sunxi\_usb/usb\_efex.c

sprite/sprite\_verify.c

drivers/sunxi\_flash/nand/nand\_for\_uboot.c

sprite/sprite\_download.c

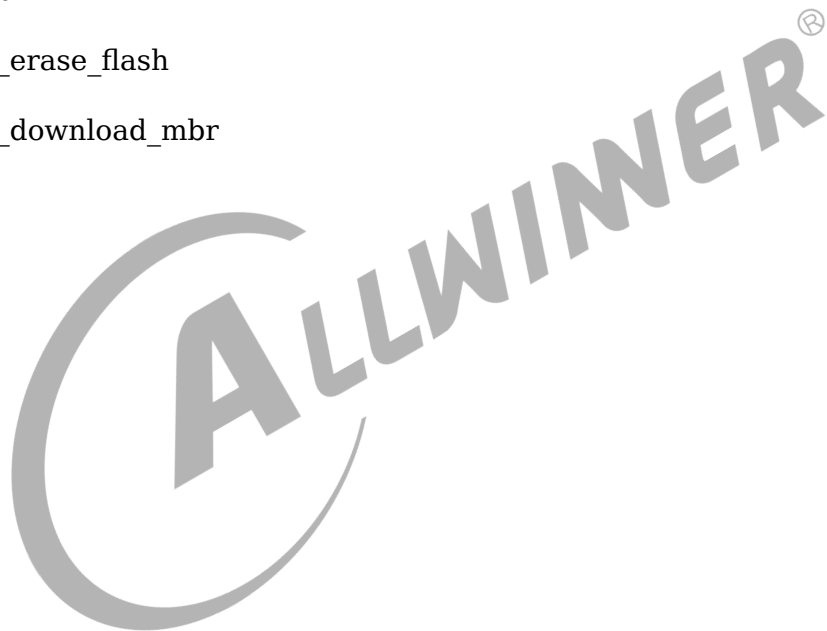
## 9.5 参考函数流程

sunxi\_sprite\_verify\_mbr

nand\_get\_mbr

sunxi\_sprite\_erase\_flash

sunxi\_sprite\_download\_mbr



## 10 partition image

描述分区信息的文件为 `sys_partition.fex`，该文件只是说明分区信息，该文件不会写入 `nand flash`。mbr 分区大小以 Kbyte 为单位，其他分区大小以 sector（512bytes, linux 块设备的读写单元为 sector）为单位。各个分区大小会按逻辑页大小对齐，这样也保证了各个分区的起始地址也是逻辑页大小对齐的。

### 10.1 input file

NO.	Name	Partitionsize(KB)	Offset(KB)	File	File size(KB)
分区表	mbr	252	0	sunxi_mbr.fex	64
分区 1	boot-resource	252	252	boot-resource.fex	
分区 2	env	252	502	env.fex	
分区 3	env-redund	252	756	env.fex	
分区 4	boot	6300	1008	boot.fex	
分区 5	rootfs	20412	7308	rootfs.fex	
分区 6	dsp0	378	27720	dsp0.fex	
分区 7	private	1008	28098	/	/
分区 8	recovery	8064	29106	recovery.fex	
分区 9	UDISK	动态计算	37170	/	/

因为 mbr 的烧写是基于 UBI 框架写入的，每个 mbr 分区信息对应一个 UBI Volume，Volume 的每个 logical eraseblock 可以被映射到任意的 physical eraseblock，映射是 UBI 管理的，并且对上层隐藏了 global wear\_leveling 机制。

#### LEB 与 PEB

block size = 128k 为例

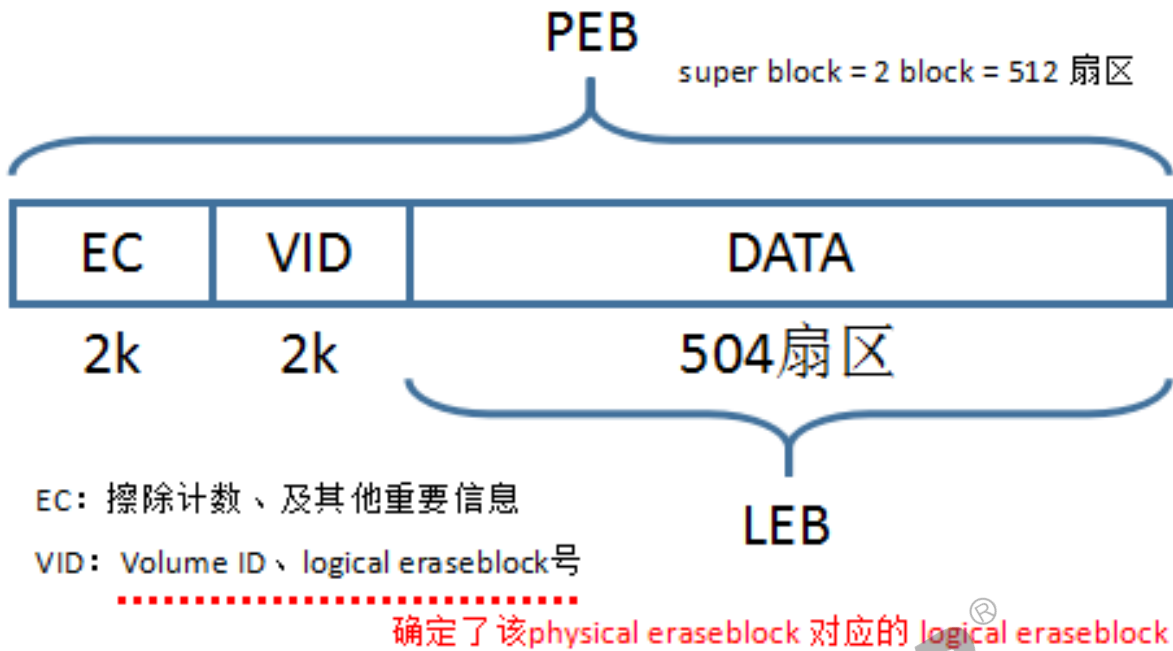


图 10-1: PEB-LEB

**PEB:** physical erase block , **LEB:** logical erase block

**PEB 和 logical block 关系**

1 PEB = 1 logical block = 2 physical blocks

## 10.2 计算逻辑区域 LEB 总数

用户可见 LEB 数 = [总物理块数 - 8 (boot0) - 24(boot1) - 8 (secure storage)] / 2 - 20 \* 总物理块数 / 1024 - 4, 规则如下:

1. 减去物理区域块数, 除 2 转化为 logical block
2. 减去坏块处理预留数 (每 1024 物理块最多 20 个物理块, 即 10 个逻辑块)
3. 减去 4 (2 个用于 ubi layout volume, 1 个用于 LEB 原子写, 1 个用于磨损均衡处理)

推算方式可以参考 u-boot-2018/cmd/ubi\_simu.c 的 ubi\_sim\_part 和 ubi\_simu\_create\_vol 函数。

正常情况下, ubi 方案 sys\_partition.fex 中各个分区的大小会按照 LEB 大小对齐。

假如一款 flash 有 1024 个 block, 每个 block 有 64 个 page, 每个 page 有 2KB, 则逻辑块大

小为 256K(642K2), 那么 PEB 大小是 256K, LEB 大小为 252K, PEB 中的首逻辑页固定用于存放 ubi\_ec\_hdr 和 ubi\_vid\_hdr。

由于预先不知道物料容量信息及预留块信息, 因此 sys\_partition.fex (sunxi\_mbr.fex) 中最后一个分区的 size 信息默认先填 0, 待 NAND 驱动初始化完成后才知道用户可见 LEB 数有多少个, 此时需要根据信息改写 sunxi\_mbr.fex 中最后一个分区的 size。

## 10.3 动态调整 sunxi\_mbr 卷

sunxi\_mbr.fex 共 64k, 共 4 个备份, 每个备份 16K

1. 计算 mbr 卷最后分区 size, 单位: 扇区 (512 字节), 计算规则如下:

- 根据 9.2 计算出的用户可见 leb 数转化出总的扇区数 total\_sector
- 依次减去分区表中各个分区占用的扇区数

2. 回填 sunxi\_mbr.fex 最后一个分区 size

3. 重新计算并回填 sunxi\_mbr 的 crc32

4. 改写其余 3 个备份

sunxi\_mbr\_t 结构体: u-boot-2018/include/sunxi\_mbr.h, 结构体各个成员均使用小端存储。

```
typedef struct sunxi_mbr
{
    unsigned int    crc32;
    unsigned int    version;
    unsigned char   magic[8];
    unsigned int    copy;
    unsigned int    index;
    unsigned int    PartCount;
    unsigned int    stamp[1];
    sunxi_partition array[SUNXI_MBR_MAX_PART_COUNT];
    unsigned int    lockflag;
    unsigned char   res[SUNXI_MBR_RESERVED];
}__attribute__((packed)) sunxi_mbr_t;
```

重新计算并回填 sunxi\_mbr crc32 的代码请参考 u-boot-2018/drivers/mtd/awnand/sunxi-ubi.c 的 adjust\_sunxi\_mbr 函数

## 10.4 根据 sunxi\_mbr 动态生成 ubi layout volume

ubi layout volume 可以理解为 UBI 模块内部用的分区信息文件, sunxi\_mbr 分区是用于全志烧写 framework 的分区信息文件。二者记录的分区信息本质上是一样的, 因此烧写时, 可以由 sunxi\_mbr 卷转化成 ubi layout volume。

ubi layout volume 由 128 个 struct ubi\_vtbl\_record (u-boot-2018/drivers/mtd/ubi/ubi-media.h) 组成, 结构体各个成员使用大端表示。

```
struct ubi_vtbl_record {
    __be32 reserved_pebs;
    __be32 alignment;
    __be32 data_pad;
    __u8 vol_type;
    __u8 upd_marker;
    __be16 name_len;
    char name[UBI_VOL_NAME_MAX+1];
    __u8 flags;
    __u8 padding[23];
    __be32 crc;
} __packed;
```

attribute name	type	value	comment
reserved_pebs	__be32		卷大小/LEB size, 对于 ubi layout volume, 固定为 2
alignment	__be32	1	
data_pad	__be32	0	
vol_type	__u8	1	动态卷: 1, 静态卷: 2, 当前方案均是动态卷
upd_marker	__u8	0	
name_len	__be16		卷名长度
name[128]	char		
flags	__u8		分区内最后一个卷 udisk, flags 为 UBI_VTBL_AUTORESIZE_FLG
padding[23]	__u8	0	
crc	__be32		crc32_le

ubi layout volume 的内容填充及烧写方法请参考 u-boot-2018/cmd/ubi\_simu.c 的 ubi\_simu\_create\_vol 和 wr\_vol\_table 函数

注意:

ubi 中 crc32\_le 算法与 sunxi\_mbr 的 crc32 算法不一样。

ubi 中 crc32\_le 参考 crc32\_le.c 用法

sunxi\_mbr 中 crc32 参考 crc32.c 用法

## 10.5 烧写逻辑卷

PEB = ubi\_ec\_hdr + ubi\_vid\_hdr + LEB

其中 ubi\_ec\_hdr 和 ubi\_vid\_hdr 存放于 PEB 的首逻辑页 (logical page0)。

- ubi\_ec\_hdr 存放于 0 字节偏移处, 大小与物理页 size 对齐



- ubi\_vid\_hdr 存放于 1 个物理页 size 偏移处，大小也与物理页 size 对齐

## 10.6 ubi\_ec\_hdr

ubi\_ec\_hdr: 主要用于存储 PEB 的擦除次数信息，需动态生成 crc32\_le 校验值。

struct ubi\_ec\_hdr 位于 u-boot-2018/drivers/mtd/ubi/ubi-media.h，结构体各个成员使用大端表示。

```
struct ubi_ec_hdr {
    __be32  magic;
    __u8    version;
    __u8    padding1[3];
    __be64  ec; /* Warning: the current limit is 31-bit anyway! */
    __be32  vid_hdr_offset;
    __be32  data_offset;
    __be32  image_seq;
    __u8    padding2[32];
    __be32  hdr_crc;
} __packed;
```

attribute name	type	value	comment
magic	__be32	0x55424923	UBI#
version	__u8	1	
padding1[3]	__u8	0	
ec	__be64	1	
vid_hdr_offset	__be32	physical page size	2048
data_offset	__be32	logical page size	4096
image_seq	__be32	0	
padding2[32]	__u8	0	
hdr_crc	__be32		crc32_le

ubi\_ec\_hdr 的填充方法请参考 u-boot-2018/cmd/ubi\_simu.c 的 fill\_ec\_hdr 函数。

## 10.7 ubi\_vid\_hdr

ubi\_vid\_hdr: 存放 PEB 和 LEB&Volume 映射信息，需动态生成 crc32\_le 校验值

struct ubi\_vid\_hdr 位于 u-boot-2018/drivers/mtd/ubi/ubi-media.h，结构体各个成员使用大端表示。

```
struct ubi_vid_hdr {
    __be32  magic;
    __u8    version;
```

```

__u8  vol_type;
__u8  copy_flag;
__u8  compat;
__be32 vol_id;
__be32 lnum;
__u8  padding1[4];
__be32 data_size;
__be32 used_ebs;
__be32 data_pad;
__be32 data_crc;
__u8  padding2[4];
__be64 sqnum;
__u8  padding3[12];
__be32 hdr_crc;
} __packed;

```

attribute name	type	value	comment
magic	__be32	0x55424921	UBI!
version	__u8	1	
vol_type	__u8	1	
copy_flag	__u8	0	
compat	__u8		默认为 0, layout volume 固定为 5
vol_id	__be32		volume id, 从 0 开始编号, layout vol 固定为 0x7ffefff
lnum	__be32		volume 内 LEB NO., 从 0 开始编号
padding1[4]	__u8	0	
data_size	__be32	0	
used_ebs	__be32	0	
data_pad	__be32	0	
data_crc	__be32	0	
padding2[4]	__u8	0	
sqnum	__be64		LEB 全局 sequence NO., 记录 LEB 的写顺序, 从 0 开始递增
padding3[12]	__u8	0	
hdr_crc	__be32		crc_le

ubi\_vid\_hdr 的填充方法请参考 u-boot-2018/cmd/ubi\_simu.c 的 fill\_vid\_hdr 函数。

## 10.8 数据对齐

有数据对齐需求时, 不能填充 0xff 数据, 可选择填充全 0。




## 著作权声明

版权所有 © 2021 珠海全志科技股份有限公司。保留一切权利。

本档及内容受著作权法保护，其著作权由珠海全志科技股份有限公司（“全志”）拥有并保留一切权利。

本档是全志的原创作品和版权财产，未经全志书面许可，任何单位和个人不得擅自摘抄、复制、修改、发表或传播本档内容的部分或全部，且不得以任何形式传播。

## 商标声明

、 **全志科技** （不完全列举）均为珠海全志科技股份有限公司的商标或者注册商标。在本档描述的产品中出现的其它商标，产品名称，和服务名称，均由其各自所有人拥有。

## 免责声明

您购买的产品、服务或特性应受您与珠海全志科技股份有限公司（“全志”）之间签署的商业合同和条款的约束。本档中描述的全部或部分产品、服务或特性可能不在您所购买或使用的范围内。使用前请认真阅读合同条款和相关说明，并严格遵循本档的使用说明。您将自行承担任何不当使用行为（包括但不限于如超压，超频，超温使用）造成的不利后果，全志概不负责。

本档作为使用指导仅供参考。由于产品版本升级或其他原因，本档内容有可能修改，如有变更，恕不另行通知。全志尽全力在本档中提供准确的信息，但并不确保内容完全没有错误，因使用本档而发生损害（包括但不限于间接的、偶然的、特殊的损失）或发生侵犯第三方权利事件，全志概不负责。本档中的所有陈述、信息和建议并不构成任何明示或暗示的保证或承诺。

本档未以明示或暗示或其他方式授予全志的任何专利或知识产权。在您实施方案或使用产品的过程中，可能需要获得第三方的权利许可。请您自行向第三方权利人获取相关的许可。全志不承担也不代为支付任何关于获取第三方许可的许可费或版税（专利税）。全志不对您所使用的第三方许可技术做出任何保证、赔偿或承担其他义务。