# ROM 分区配置

Version: 1.0

Date: 2018-01-01

| | | | |
|---|---|---|---|
| **Document Number:** | | **Document Version:** | |
| **Owner:** | | **Date:** | |
| **Document Type:** | | | |
| **NOTE:** | ALL MATERIALS INCLUDED HEREIN ARE COPYRIGHTED AND CONFIDENTIAL UNLESS OTHERWISE INDICATED. The information is intended only for the person or entity to which it is addressed and may contain confidential and/or privileged material. Any review, retransmission, dissemination, or other use of or taking of any action in reliance upon this information by persons or entities other than the intended recipient is prohibited.<br><br>This document is subject to change without notice. Please verify that your company has the most recent specification.<br><br>Copyright © 2013 Spreadtrum Communications Inc. | | |



SPREADTRUM

www.spreadtrum.com

# 目录

本文涉及的专有名词，定义和缩写等：

| | |
|---|---|
| MTD | memory technology device |
| UBI | Unsorted Block Images |
| UBIFS | Unsorted Block Image File System |
| eMMC | Embedded Multi Media Card |
| | |
| | |
| | |
| | |

# 1. 分区结构概述

本章节旨在初步介绍展讯平台 ROM 空间的分配情况，详细内容还需要从代码层面去深入的理解。

## 1.1. 分区介绍

首先看看我们有哪些分区，如图 1 所示：

| Partition Number | Partition Name | PartitionType | PartitionSize/单位 | PartitionPurpose |
|---|---|---|---|---|
| 1 | splloader | RAW/MTD | BOOT0,大小由 eMMC 厂家定义 | 这个分区是 spl-16k.bin，主要功能是完成 DDR 初始化、加载 u-boot.bin 到 0x8f800000 这个地址，并且跳转到该地址运行。 |
| 2 | ubootloader | RAW/MTD | BOOT1,大小由 eMMC 厂家定义 | 这个分区用来保存 u-boot.bin，主要功能是初始化一些硬件相关、完成将 flash 上的数据加载到 RAM 中 |
| 3 | prodnv | Ext4/ubifs | size="5" /MBytes> | 这个分区是 prodnv.bin，该分区就是开机后系统中的 productinfo 分区，保存 adc 校准参数、eng.db 数据库。 |
| 4 | miscdata | RAW/ubi | size="1" /MBytes> | 这个分区用来保存 ota、recovery 时的一些数据。 |
| 5 | wmodem | RAW/ubi | size="8" /MBytes> | 这个分区用来存放 modem.bin，通信协议栈相关。 |
| 6 | wdsp | RAW/ubi | size="1" /MBytes> | 这个分区用来存放 dsp.bin，数字处理等。 |
| 7 | wfixnv1 | RAW/ubi | size="1" /MBytes> | 这个分区用来存放 fixnv.bin，射频参数相关。 |

| | | | | |
|---|---|---|---|---|
| 8 | wfixnv2" | RAW/ubi | size="1" /MBytes> | 这分区是 fixnv1 的备份，防止 fixnv 破坏导致系统无法开机。 |
| 9 | wruntimenv1 | RAW/ubi | size="1" /MBytes> | 这个分区是运行时由 modem 生成，是 fixnv 的一份复制。 |
| 10 | wruntimenv | RAW/ubi | size="1" /MBytes> | 这个分区是 wruntimenv1 的备份分区，起到掉电保护作用。 |
| 11 | wcnmodem | RAW/ubi | size="1" /MBytes> | 这个分区用来存放 wcnmodem.bin，是 Connectivity 芯片的的协议栈相关。 |
| 12 | wcnfixnv1 | RAW/ubi | size="1" /MBytes> | 这个分区用来存放 wcnfixnv.bin，是 Connectivity 芯片的射频参数相关。 |
| 13 | wcnfixnv2 | RAW/ubi | size="1" /MBytes> | 这个分区数据是运行时生成，是 wcnfixnv 的一份复制。 |
| 14 | wcnruntimenv1 | RAW/ubi | size="1" /MBytes> | 这个分区是 wcnruntimenv1 的备份，起到掉电保护的作用。 |
| 15 | logo | RAW/ubi | size="1" /MBytes> | 这个分区用来存放开机 logo 图片。 |
| 16 | fbootlogo | RAW/ubi | size="1" /MBytes> | 这个分区用来存放 fastboot 模式的 logo 图片。 |
| 17 | boot | RAW/ubi | size="11" /MBytes> | 这个分区用来存放 boot.img，kernel 驱动相关。 |
| 18 | system | Ext4/ubifs | size="440" /MBytes> | 这个分区用来存放 system.img，android 系统相关。 |
| 29 | cache | Ext4/ubifs | size="150" /MBytes> | 这个分区用来存放 cache.img，在 CTS 测试，恢复出厂设置是需要使用。 |

| | | | | |
|---|---|---|---|---|
| 20 | recovery | RAW/ubi | size="12" /MBytes> | 这个分区用来存放 recovery.img， 恢复出厂设置相关。 |
| 21 | misc | RAW/ubi | size="1" /MBytes> | 此分区包含杂项系统关闭开关上的窗体中的设置相关。 |
| 22 | userdata | Ext4/ubifs | size="1500" /MBytes> | 这个分区用来存放 userdata.img， 包含用户的数据。 |
| 23 | internalsd | FAT(eMMC 方案专有分区) | size="0xFFFFFFFF"/> | 内置 sd 卡分区 |

图 1

图 1 是的展讯平台 ROM 空间划分情况以及分区格式、分区大小，分区功能的的初步描述，先对展讯平台的分区情况有了一个宏观上的认识，接下来我们分别对 NAND 方案和 eMMC 方案的分区做进一步的介绍。

## 1.2. eMMC 方案分区形式

在 eMMC 方案中我们可以通过查看对应的 pac 包中的 Productname.xml 文件看到分区的详细信息。具体如下：

```
<Partitions>
        <!-- size unit is MBytes -->
        <!--
        <Partition id="splloader"       size="0"/>
        <Partition id="ubootloader"     size="0"/>
        -->
        <Partition id="prodnv" size="5"/>
        <Partition id="miscdata" size="1"/>
        <Partition id="wmodem" size="8"/>
        <Partition id="wdsp" size="2"/>
        <Partition id="wfixnv1" size="1"/>
        <Partition id="wfixnv2" size="1"/>
        <Partition id="wruntimenv1" size="1"/>
        <Partition id="wruntimenv2" size="1"/>
        <Partition id="wcnmodem" size="1"/>
        <Partition id="wcnfixnv1" size="1"/>
        <Partition id="wcnfixnv2" size="1"/>
        <Partition id="wcnruntimenv1" size="1"/>
        <Partition id="wcnruntimenv2" size="1"/>
        <Partition id="logo" size="1"/>
        <Partition id="fbootlogo" size="1"/>
        <Partition id="boot" size="11"/>
        <Partition id="system" size="400"/>
        <Partition id="cache" size="150"/>
        <Partition id="recovery" size="12"/>
        <Partition id="misc" size="1"/>
        <Partition id="userdata" size="1500"/>
        <Partition id="internalsd" size="0xFFFFFFFF"/>
</Partitions>
```

## 1.3. Nand 方案分区形式

Nand 分区与 emmc 方案的几乎一致，但不同点要特别注意，如下所示为一个 nand 工程常用的 xml 分区配置：

```
-<Partitions>
<!-- size unit is MBytes -->
<Partition size="5" id="prodnv"/>
<Partition size="1" id="miscdata"/>
<Partition size="20" id="recovery"/>
<Partition size="1" id="misc"/>
<Partition size="1" id="logo"/>
<Partition size="1" id="fbootlogo"/>
<Partition size="1" id="l_fixnv1"/>
<Partition size="1" id="l_fixnv2"/>
```

<Partition size="1" id="l_runtimenv1"/>

<Partition size="1" id="l_runtimenv2"/>

<Partition size="1" id="gpsgl"/>

<Partition size="1" id="gpsbd"/>

<Partition size="10" id="wcnmodem"/>

<Partition size="25" id="l_modem"/>

<Partition size="10" id="l_gdsp"/>

<Partition size="20" id="l_ldsp"/>

<Partition size="1" id="pm_sys"/>

<Partition size="20" id="boot"/>

<Partition size="200" id="system"/>

<Partition size="25" id="cache"/>

<Partition size="0xFFFFFFFF" id="userdata"/> </Partitions>

但不同的两个地方需要特别注意：

1.Nand 方案的分区是在 mtd 分区的基础上实现的，mtd 分区由 fdl2 中对应的 board configs 文件 include/configs/xx.h 设置

#define MTDPARTS_DEFAULT "mtdparts=sprd-nand:256k(splloader),1280k(uboot),-(ubipac)

如上的设置的意思是 将 nand 的全部 block 分为三个 mtd 分区 第一个是 splloader 分区 大小为 256KB 第二个是 uboot 分区 大小为 1280KB 第三个是 ubipac 分区 大小为剩余的部分。

因此，对于 uboot 和 splloader 的分区大小调整，需要在对应的 fdl2 中对应的 board configs 文件 include/configs/xx.h 设置。

由于 nand 厂商仅承诺第一个 block 一定是好的并且为了坏块的处理，因此对于 mtd 的分区大小调整，必须以 block 大小的整数倍划分，并且最好给每个 mtd 分区预留几个 block，一旦发现有怀块，还可以跳过该 block。一般 nand 有两种 block 大小 128KB 和 256KB 。

2.对于三个分区文件 prodnv system userdata cache 由于需要在 ubifs 文件系统中挂载，所以需要对应使用的 nand 器件的性质，包括 block 大小和 page 大小。目前市面上主要有两种 nand 器件 ，一种是每个 page 4kB ，每个 block 256KB；另一种是每个 page 2KB，每个 block 128KB。

因此在 nand 工程的 pac 包中，对于以上 4 个镜像文件，都会有两个，例如 system_p4K_b256K.img 和 system_p2K_b128K.img ,xml 中 则需要设置 selbyflash =1

的字样 reacherdownloader 下载工具会在 fdl2 阶段查询该机器使用的 nand 类型 以便选择不同的 img 下载。

```
<File selbyflash="1">
 <ID>System</ID>
 <IDAlias>System</IDAlias>
 <Type>YAFFS_IMG2</Type>
 <Block id="system">
   <Base>0x0</Base>
   <Size>0x0</Size>
 </Block>
<Flag>1</Flag>
<CheckFlag>0</CheckFlag>
<Description>System image file</Description>
 </File>
```

# 2. 如何增加分区

增加一个分区请参考如下方法进行，这个不区分 eMMC 和 nand ，修改方法是一致的只需要修改对应工程的 xml 文件就可以完成分区的增加工作。

例如：增加一个 fast_logo 分区；

第一步：增加一行 <Partition id="**fast_logo**" size="1"/>

对于我们增加的分区 size 一般是依照实际要写入的 bin 文件的大小来定义，大小以 M 为单位。Size 设置的过大比较浪费空间，设置太小下载时就会报错，详细可以参考 2.2 的详细描述。

第二步：增加 file 定义

```
<File>
        <ID>Fast_Logo</ID>
        <IDAlias> Fast_Logo </IDAlias>
```

<Type>**CODE2**</Type>//还有一种 Type **EraseFlash2** Type 决定是否会有文件写入到给分区，如果是 CODE2 就可以写入数据到该分区，如果是 EraseFlash2 就对该分区只进行格式化操作。

<Block id="**fast_logo**">

<Base>0x0</Base>

<Size>0x0</Size>

</Block>

<Flag>1</Flag> //如果不置 1 是无法选择文件写入的。

<CheckFlag>0</CheckFlag>

<Description>**fast logo image file**</Description>//这个描述可以自定义

</File>

修改好对应的 xml 文件之后重新制作 pac 包使用工具加载新生成的 pac 包或者重新在工具中选择对应的 Product 就可以看到增加的这个分区了。

如果想增加一个 sd 分区和上边的修改方案是一样的，不过需要将 Type 修改成 EraseFlash2。这样 eMMC 方案的 fdl2 如果检测到分区 id 是 sd 会将其格式化成 FAT 文件系统，写入 FAT 文件系统信息。

## 2.1. 如何修改分区大小

这个操作可以说成是增加一个分区的一部分，因为新增加一个分区也要配置分区的大小。随着项目的进行起初配置的分区大小已经无法满足当前的需求的时候就需要重新配置分区的大小，那么如何配置分区大小那？操作很简单举例如下：

第一、修改 system 分区

修改 system 分区大小涉及以下几点：

1）修改/device/sprd/“project”/Boardconfig.mk 文件：

BOARD_SYSTEMIMAGE_PARTITION_SIZE := 7000000000 #这里配置的是文件系统镜像大小

BOARD_USERDATAIMAGE_PARTITION_SIZE := 1950000000 #这里配置的是文件系统镜像大小

如果是增大 system 分区镜像那么就需要相应的减小 userdata 分区镜像，不然开机会报加密失败。

2）修改工具工程配置 project.xml 文件中对应的 system 分区的大小，如下：

<Partition id="system" size="235"/> :这里配置的是物理的分区大小，即 system 分区大小，这个值必须大于等于 Boardconfig.mk 中配置的文件系统镜像大小。

<Partition id="userdata" size="0xFFFFFFFF"/> ：data 分区是根据 flash 总大小减去其他空间总大小的差值，因此这里不需要修改。

修改好 xml 文件之后请重新制作 pac 包，以确保修改成功。

第二、修改 cache 分区

1）修改/device/sprd/”project“/Boardconfig.mk 文件

BOARD_CACHEIMAGE_PARTITION_SIZE := 150000000 ：这里配置的是文件系统镜像大小

如果需要将这个修改的更大的话就需要相应的减少 data 分区 img 大小，参考 system 分区修改。

2）修改工具工程配置 project.xml 文件中对应的 cache 分区的大小，如下：

<Partition id="cache" size="150"/> ：这里配置的是物理的分区大小，请修改成需要配置的大小,物理分区必须大于等于 Boardconfig.mk 中配置的文件系统镜像大小。

修改好 xml 文件之后请重新制作 pac 包，以确保修改成功。

第三、修改 prodnv 分区

1）修改/device/sprd/”project“/Boardconfig.mk 文件

BOARD_PRODNVIMAGE_PARTITION_SIZE := 5242880 ：这里配置的是文件系统镜像大小

如果需要将这个该的更大的话就需要相应的减少 data 分区 img 大小，参考 system 分区修改。

2）修改工具工程配置 project.xml 文件中对应的 prodnv 分区的大小，如下：

<Partition id="prodnv" size="5"/> :这里配置的是物理分区大小，请修改成需要配置的大小，物理分区必须大于等于 Boardconfig.mk 中配置的文件系统镜像大小。

注意：修改好这个之后还需要修改备份 prodnv 的大小修改如下：

```
<File backup="1">
  <ID>ProdNV</ID>
  <IDAlias>ProdNV</IDAlias>
  <Type>CODE2</Type>
  <Block id="prodnv">
    <Base>0x0</Base>
    <Size>0x500000</Size> ：这里是备份 prodnv 的大小 5M，请修改成需要配置的大小。
  </Block>
  <Flag>1</Flag>
```

```
    <CheckFlag>0</CheckFlag>
    <Description>Download prodnv section   operation</Description>
  </File>
```

修改好 xml 文件之后请重新制作 pac 包，以确保修改成功。

第四、修改其他分区大小

修改除 system、cache、prodnv、data 之外的分区只需要修改工具中 project.xml 文件即可，修改好 xml 文件之后请重新制作 pac 包，以确保修改成功。

# 3. 如何删除分区

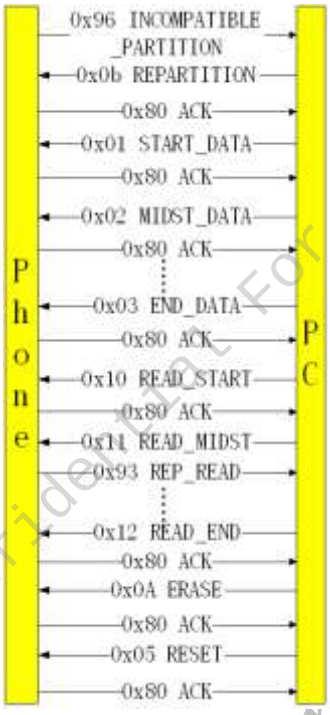
删除一个分区的方法和增加一个分区就是一个对立的过程，进行相反的操作操作就可以完成这个工作，即删除 xml 文件中的 id 项和对应的 file 项就可以了，修改好了之后同样需要重新制作 pac 包使用工具加载新生成的 pac 或者在工具中重新选择修改好的 Product 即可检查修改是否生效。

# 4. repartition 分区生效流程

这一节从软件流程上说明增删改物理分区之后，是如何生效的，即软件上如何将新的分区信息写入到手机内部的存储器件上（nand/eMMC）。

整个下载过程采用的是简单的问答式协议（如下图），由 PC（下载工具）主控，相当于 server，手机则相当于 client。从下图协议握手过程可以看到，下载过程第一步就是重分区。

重分区 repartition：我们在下载 pac 的时候，fdl2 会向 PC 下载工具发送分区不一致的信号（BSL_INCOMPATIBLE_PARTITION），然后下载工具会下发重分区命令（BSL_REPARTITION），fdl2 收到重分区命令后，会执行相应的注册函数，解析 pac 中的 xml 分区文件，进行重新分区工作。




以下以 uboot15 代码为例说明：

u-boot15\common\ cmd_download.c

```
int do_download(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
{
… …
        /* register all cmd process functions */
        dl_cmd_register(BSL_CMD_START_DATA, dl_cmd_write_start);
        dl_cmd_register(BSL_CMD_MIDST_DATA, dl_cmd_write_midst);
        dl_cmd_register(BSL_CMD_END_DATA, dl_cmd_write_end);
        dl_cmd_register(BSL_CMD_READ_FLASH_START, dl_cmd_read_start);
        dl_cmd_register(BSL_CMD_READ_FLASH_MIDST, dl_cmd_read_midst);
        dl_cmd_register(BSL_CMD_READ_FLASH_END, dl_cmd_read_end);
        dl_cmd_register(BSL_ERASE_FLASH, dl_cmd_erase);
        dl_cmd_register(BSL_REPARTITION, dl_cmd_repartition);
        dl_cmd_register(BSL_CMD_NORMAL_RESET, dl_cmd_reboot);
        dl_cmd_register(BSL_CMD_POWER_DOWN_TYPE, dl_powerdown_device);
```

```
… …
        /* uart download doesn't supoort disable hdlc, so need check it */
        if (FDL_get_DisableHDLC() == NULL)
                dl_send_ack (BSL_INCOMPATIBLE_PARTITION);
        else {
                Da_Info.dwVersion = 1;
                Da_Info.bDisableHDLC = 1;
                ack_packet.body.type = BSL_INCOMPATIBLE_PARTITION;
                memcpy((unchar *)ack_packet.body.content, (unchar
*)&Da_Info,sizeof(Da_Info));
                ack_packet.body.size = sizeof(Da_Info);
                dl_send_packet(&ack_packet);
        }
        /* enter command handler */
        dl_cmd_handler();
         return 0;
}
```

在 do_download 函数中进行完一系列命令的注册后，uboot 会向下载工具发送 BSL_INCOMPATIBLE_PARTITION 的信号，之后进入 dl_cmd_handler 函数,等待工具发的 command。

工具在收到手机发送的 BSL_INCOMPATIBLE_PARTITION 信号后，会向手机端发送 BSL_REPARTITION 的 command。手机侧执行 dl_cmd_repartition 函数，进行重分区工作。

Uboot15/common/dloader/dl_cmd_proc.c

```
int dl_cmd_repartition(dl_packet_t *pakcet, void *arg)
{
… …
        /*接收工具发送的新的 XML 物理分区信息*/
        _parse_repartition_header(raw_data, &rp_info, &p_part_list);
        if (0 == rp_info.version) {
```

```
        part_cell_length = REPARTITION_UNIT_LENGTH;
    } else {
        part_cell_length = REPARTITION_UNIT_LENGTH_V1;
        size = rp_info.table_size;
    }
    if (0 != (size % part_cell_length)) {
        printf("%s:recvd packet size(%d) error \n", __FUNCTION__,size);
        dl_send_ack(BSL_INCOMPATIBLE_PARTITION);
        return 0;
    }
    total_partition_num = size / part_cell_length;
    debugf("Partition total num:%d \n", total_partition_num);
    op_res = dl_repartition(p_part_list, total_partition_num, rp_info.version,
rp_info.unit);
    _send_reply(op_res);
    return 0;
}
```

从上面的函数可知，在接收到 PAC 包中 XML 分区信息，并做了一些解析后，执行 dl_repartition 函数，以下**以 emmc 中的处理函数为例**：

Uboot15/common/dl_operate.c

```
OPERATE_STATUS dl_repartition(uchar * partition_cfg, uint16_t total_partition_num,
                                                    uchar version, uchar size_unit)
{
… …
    res = _parser_repartition_cfg(partition_info, partition_cfg, total_partition_num,
version, size_unit);
    if (res < 0) {
        free(partition_info);
```

```
        return OPERATE_SYSTEM_ERROR;
    }
    res = common_repartition(partition_info, (int)total_partition_num);
    free(partition_info);
    if (0 == res)
        return OPERATE_SUCCESS;
    else
        return OPERATE_SYSTEM_ERROR;
}
```

其中：_parser_repartition_cfg：解析工具传输的分区信息。

common_repartition:执行分区动作函数。

uboot15/common/sprd_common_rw.c

```
int common_repartition(disk_partition_t *partitions, int parts_count)
{ … …
    memset(&local_part_info, 0 , sizeof(disk_partition_t));
    dev_desc = get_dev_hwpart("mmc", 0, PARTITION_USER);
    if (NULL == dev_desc) {
        errorf("get mmc device hardware part(%d) fail\n", PARTITION_USER);
        return -1;
    }
    while (counter < 3) {
        ret = gpt_restore(dev_desc, SPRD_DISK_GUID_STR, partitions, parts_count);
        if (0 == ret)
            break;
        counter++;
    }
```

```
    if (3 == counter) {
            return -1;
    } else {
            init_part(dev_desc);
            return 0;
    }
```

由上可知，执行 **gpt_restore**，将新的分区信息写入。这里在写入时会判断写入是否成功，如果不成功会重复尝试三次。

# 5. 芯片工程存储（Emmc Nand）相关代码概述

对于使用 emmc 和 nand 两种不同存储器件的芯片工程而言，主要的区别在于底层驱动的不同和对应使用的文件系统的不同，以及由此带来的 fstab 的不同和分区结构的不同。文件系统屏蔽了不同存储器件的差异，向上提供统一的接口，因此 framwork 层和应用层不需要做任何修改。

本章节主要从 device splloader uboot kernel 四个方面介绍关于 emmc 和 nand 的不同配置。

## 5.1. device

芯片工程的 device 中 ，与存储相关的配置主要包括以下几个：

**xml 文件** 作用：负责 rom 分区,第二章 rom 分区会重点介绍，这里不再赘述。

fstab 文件 作用：文件系统挂载

recovery.fstab 作用：同 fstab 用于恢复出厂设置的文件系统挂

emmc fstab:

```
/dev/block/platform/sdio_emmc/by-name/system      /system           ext4  ro,barrier=1 wait
/dev/block/platform/sdio_emmc/by-name/userdata    /data                           ext4 noatime,nosuid,nodev,nomblk_io_submit,noauto_da_alloc,discard wait,encryptable=footer,check
/dev/block/platform/sdio_emmc/by-name/cache    /cache                             ext4 noatime,nosuid,nodev,nomblk_io_submit,noauto_da_alloc,discard wait,check,formattable
/dev/block/platform/sdio_emmc/by-name/prodnv                    /productinfo      ext4 noatime,nosuid,nodev,nomblk_io_submit,noauto_da_alloc,discard wait,check
```

nand fstab:

```
/dev/ubi0_system                          /system          ubifs   ro,compr=lzo
wait
/dev/ubi0_userdata                /data          ubifs  noatime,nosuid,nodev,compr=lzo
wait,encryptable=footer
/dev/ubi0_cache                   /cache         ubifs  noatime,nosuid,nodev,compr=lzo
wait
/dev/ubi0_prodnv                  /productinfo   ubifs  noatime,nosuid,nodev,compr=lzo
wait
```

Recovery fstab 与此类似。

BoardPartitionconfig.mk

该文件控制了与文件系统相关的镜像文件的格式和大小配置 例如 system.img cache.img userdata.img prodnv.img

Emmc 版本 ：主要配置 ubifs 文件系统相关镜像的 size 和文件系统格式

```
TARGET_USERIMAGES_USE_EXT4 := true
BOARD_CACHEIMAGE_PARTITION_SIZE := 150000000
BOARD_PRODNVIMAGE_PARTITION_SIZE := 5242880
BOARD_PERSISTIMAGE_PARTITION_SIZE := 2097152
BOARD_SYSINFOIMAGE_PARTITION_SIZE := 5242880
BOARD_FLASH_BLOCK_SIZE := 4096
BOARD_CACHEIMAGE_FILE_SYSTEM_TYPE := ext4
BOARD_PRODNVIMAGE_FILE_SYSTEM_TYPE := ext4
BOARD_SYSINFOIMAGE_FILE_SYSTEM_TYPE := ext4

TARGET_SYSTEMIMAGES_SPARSE_EXT_DISABLED := true
TARGET_USERIMAGES_SPARSE_EXT_DISABLED := false
TARGET_CACHEIMAGES_SPARSE_EXT_DISABLED := false
TARGET_PRODNVIMAGES_SPARSE_EXT_DISABLED := true
TARGET_SYSINFOIMAGES_SPARSE_EXT_DISABLED := true

ifeq ($(strip $(BOARD_HAVE_OEM_PARTITION)),true)
BOARD_OEMIMAGE_PARTITION_SIZE := 524288000
```

```
BOARD_OEMIMAGE_FILE_SYSTEM_TYPE := ext4
TARGET_OEMIMAGES_SPARSE_EXT_DISABLED := true
Endif
```

Nand 版本：主要配置 ubifs 文件系统相关镜像的 size 和文件系统格式

```
# UBIFS partition layout

BOARD_FLASH_BLOCK_SIZE := 4096
TARGET_USERIMAGES_USE_UBIFS := true
BOARD_PAGE_SIZE := 4096
BOARD_SECT_SIZE := 4096
BOARD_BLOCK_SIZE := 262144
BOARD_ERASE_SIZE := 253952
BOARD_SYSTEMIMAGE_PARTITION_SIZE := 350000000
BOARD_USERDATAIMAGE_PARTITION_SIZE := 300000000
BOARD_CACHEIMAGE_PARTITION_SIZE := 50000000
BOARD_PRODNVIMAGE_PARTITION_SIZE := 10000000
BOARD_CACHEIMAGE_FILE_SYSTEM_TYPE := ubifs
BOARD_PRODNVIMAGE_FILE_SYSTEM_TYPE := ubifs
BOARD_SYSINFOIMAGE_FILE_SYSTEM_TYPE := ubifs
```

## 5.2. splloader

编译配置 ：根据使用 emmc 和 nand 的不同，splloader 分为两种配置。通过在 Makefile 使用 CONFIG_EMMC_BOOT 和 CONFIG_NAND_BOOT 区分编译不同的文件。

Emmc 代码配置： emmc 工程使用 emmc_boot.c 读取 emmc 中的 uboot 分区中的镜像并 load 到内存中相应的地址中。代码如下：

```
Emmc_Read(PARTITION_BOOT2,0,CONFIG_SYS_EMMC_U_BOOT_SECTOR_NUM, (uint8 *) CONFIG_SYS_NAND_U_BOOT_DST);}
```

Nand 代码配置：nand 工程使用 nand_boot.c 读取 nand 中的 mtd1 分区中的镜像并 load 到内存中的相应地址中代码如下：

```
Nand_load(&nand_info,CONFIG_SYS_NAND_U_BOOT_OFFS,CONFIG_SYS_NAND_U_BOOT_SIZE,(uchar*) CONFIG_SYS_NAND_U_BOOT_DST-SECURE_HEADER_OFF);
```

## 5.3. uboot / fdl2

fdl2 承担了芯片的几乎所有的下载任务，主要包括存储芯片包括 emmc 和 nand 片初始化，文件分区系统的初始化和镜像的写入任务，本章节重点介绍分区系统和镜像的下载。

Emmc 方案：使用 GPT 分区系统。

gpt 分区的重分区

在 u-boot15/common/sprd_common_rw.c 中 关于 gpt 分区的重分区代码如下：

```
int common_repartition(disk_partition_t *partitions, int parts_count)
{
    block_dev_desc_t *dev_desc;
    int counter = 0;
    int ret = 0;

    memset(&local_part_info, 0 , sizeof(disk_partition_t));
    dev_desc = get_dev_hwpart("mmc", 0, PARTITION_USER);
    if (NULL == dev_desc) {
        errorf("get mmc device hardware part(%d) fail\n", PARTITION_USER);
        return -1;
    }
    while (counter < 3) {
        ret = gpt_restore(dev_desc, SPRD_DISK_GUID_STR, partitions, parts_count);
        if (0 == ret)
            break;
        counter++;
    }
    if (3 == counter) {
        return -1;
    } else {
        init_part(dev_desc);
        return 0;
    }
}
```

其中关键函数 gpt_restore 以及子函数 gpt_fill_header 和 gpt_fill_pte 以及 write_gpt_table 都在目录 u-boot15/disk/part_efi.c 中 。

函数 write_gpt_table 为重分区的重点函数，涉及到包括各个分区名称以及起始地址信息的 gpt 分区头写入 mmc 的 user 分区的前面几个 block 中。

```
int write_gpt_table(block_dev_desc_t *dev_desc,
          gpt_header *gpt_h, gpt_entry *gpt_e)
{
    const int pte_blk_cnt = BLOCK_CNT((gpt_h->num_partition_entries
                                * sizeof(gpt_entry)), dev_desc);
    u32 calc_crc32;

    debug("max lba: %x\n", (u32) dev_desc->lba);
    /* Setup the Protective MBR */
    if (set_protective_mbr(dev_desc) < 0)
          goto err;

    /* Generate CRC for the Primary GPT Header */
    calc_crc32 = efi_crc32((const unsigned char *)gpt_e,
                     le32_to_cpu(gpt_h->num_partition_entries) *
                     le32_to_cpu(gpt_h->sizeof_partition_entry));
    gpt_h->partition_entry_array_crc32 = cpu_to_le32(calc_crc32);

    calc_crc32 = efi_crc32((const unsigned char *)gpt_h,
                     le32_to_cpu(gpt_h->header_size));
    gpt_h->header_crc32 = cpu_to_le32(calc_crc32);

    /* Write the First GPT to the block right after the Legacy MBR */
    if (dev_desc->block_write(dev_desc->dev, 1, 1, gpt_h) != 1)
          goto err;

    if (dev_desc->block_write(dev_desc->dev, 2, pte_blk_cnt, gpt_e)
        != pte_blk_cnt)
          goto err;

    prepare_backup_gpt_header(gpt_h);

    if (dev_desc->block_write(dev_desc->dev,
                        (lbaint_t)le64_to_cpu(gpt_h->last_usable_lba)
```

```
                        + 1,
                        pte_blk_cnt, gpt_e) != pte_blk_cnt)
                goto err;

        if (dev_desc->block_write(dev_desc->dev,
                        (lbaint_t)le64_to_cpu(gpt_h->my_lba), 1,
                        gpt_h) != 1)
                goto err;

        debug("GPT successfully written to block device!\n");
        return 0;

     err:
        printf("** Can't write to device %d **\n", dev_desc->dev);
        return -1;
    }
```

分区寻找以及镜像写入

在建立好 gpt 分区后，接下来的工作就是将接收上位机发送的各个分区的镜像写入对应的分区中。具体代码实现在路径 u-boot15/common/sprd_common_rw.c 的函数 common_raw_write 中

```
dev_desc = get_dev_hwpart("mmc", 0, PARTITION_USER);
get_partition_info_by_name(dev_desc, part_name, &part_info);
```

详细过程函数：遍历 gpt 分区表，寻找对应的分区名和起始地址，并返回。

```
int get_partition_info_by_name_efi(block_dev_desc_t * dev_desc, uchar *partition_name,
disk_partition_t *info)
{
    ALLOC_CACHE_ALIGN_BUFFER_PAD(gpt_header, gpt_head, 1, dev_desc->blksz);
    gpt_entry *pgpt_pte = NULL;
    int ret=-1;
    unsigned int i,j,partition_nums=0;
    uchar disk_partition[PARTNAME_SZ];

    if (!dev_desc || !info || !partition_name) {
        printf("%s: Invalid Argument(s)\n", __func__);
        return -1;
    }
```

```
        /* This function validates AND fills in the GPT header and PTE */
        if (is_gpt_valid(dev_desc, GPT_PRIMARY_PARTITION_TABLE_LBA,
                        gpt_head, &pgpt_pte) != 1) {
                printf("%s: *** ERROR: Invalid Main GPT ***\n", __func__);
                if (is_gpt_valid(dev_desc, (dev_desc->lba - 1),
                                gpt_head, &pgpt_pte) != 1) {
                        printf("%s: *** ERROR: Invalid alternate GPT ***\n",
                                __func__);
                        return -1;
                } else {
                        /* Rewrite Primary GPT partition table with the value of Backup GPT
*/
                        write_primary_gpt_table(dev_desc, gpt_head, pgpt_pte);
                }
        }
        /*Get the partition info*/
        partition_nums=le32_to_cpu(gpt_head->num_partition_entries);
           for(i=0;i<partition_nums;i++)
        {
                for(j=0;j<PARTNAME_SZ;j++)
        {
                disk_partition[j]=pgpt_pte[i].partition_name[j]&0xFF;
        }
                if(0==strcmp(disk_partition,partition_name))
        {
        /* The ulong casting  limits the maximum disk size to 2 TB */
        info->start = (ulong)le64_to_cpu(pgpt_pte[i].starting_lba);
        /* The ending LBA is inclusive, to calculate size, add 1 to it */
        info->size = (ulong)le64_to_cpu((pgpt_pte[i].ending_lba) + 1)
                        - info->start;
        info->blksz = dev_desc->blksz;

        sprintf((char *)info->name, "%s",print_efiname(&((pgpt_pte)[i])));
        sprintf((char *)info->type, "U-Boot");

        /*
```

```
        debug("%s: start 0x" LBAF ", size 0x" LBAF ", name %s\n", __func__,
              info->start, info->size, info->name);
        */
        ret =0;
        break;
        }
    }
        /* Remember to free pte */
        if(pgpt_pte!=NULL){
        free(pgpt_pte);
    }
        return ret;
    }
```

Nand 分区方案：使用 mtd 分区和 ubifs 分区相结合的分区格式

1.分区挂载和初始化：不同于块设备 emmc 使用 gpt 分区，nand 方案的芯片分区使用 mtd 分区和 ubifs 分区相结合的分区格式。

mtd 分区由 fdl2 中对应的 board configs 文件 include/configs/xx.h 设置

```
#define MTDPARTS_DEFAULT "mtdparts=sprd-nand:256k(splloader),1280k(uboot),-(ubipac)
```

如上的设置的意思是 将 nand 的全部 block 分为三个 mtd 分区 第一个是 splloader 分区 大小为 256KB 第二个是 uboot 分区 大小为 1280KB 第三个是 ubipac 分区 大小为剩余的部分。

因此我们将除了 splloader 和 uboot 的其他镜像放置在第三个 mtd 分区中，我们首先应该将第三个 mtd 分区 ubipac 初始化为 ubifs 分区系统，在函数 fdl_ubi_dev_init 中。

```
 int fdl_ubi_dev_init(void)
{
    int ret;
    if(cur_ubi.ubi_initialized){
        ubi_detach_mtd_dev(cur_ubi.dev_num, 1);
        cur_ubi.dev_num = fdl_ubi_dev_attach(UBIPAC_PART);
    }else{
        cur_ubi.dev_num = nand_ubi_dev_init();
    }
    cur_ubi.ubi_initialized =1;
    if(cur_ubi.dev_num<0){
```

```
        printf("%s:ubi dev attach failure!\n",__FUNCTION__);
        return 0;
    }
    cur_ubi.dev = ubi_get_device(cur_ubi.dev_num);
    if(!cur_ubi.dev){
        printf("%s:ubi get device failure!\n",__FUNCTION__);
        return 0;
    }
    return 1;
```

ubifs 文件系统的初始化比较复杂，主要函数为 u-boot15/drive/mtd/ubi 目录下。

```
int ubi_attach(struct ubi_device *ubi, int force_scan)
{
    int err;
    struct ubi_attach_info *ai;

    ai = alloc_ai("ubi_aeb_slab_cache");
    if (!ai)
        return -ENOMEM;

#ifdef CONFIG_MTD_UBI_FASTMAP
    /* On small flash devices we disable fastmap in any case. */
    if ((int)mtd_div_by_eb(ubi->mtd->size, ubi->mtd) <= UBI_FM_MAX_START) {
        ubi->fm_disabled = 1;
        force_scan = 1;
    }

    if (force_scan)
        err = scan_all(ubi, ai, 0);
    else {
        err = scan_fast(ubi, ai);
        if (err > 0) {
            if (err != UBI_NO_FASTMAP) {
                destroy_ai(ai);
                ai = alloc_ai("ubi_aeb_slab_cache2");
                if (!ai)
```

```
                        return -ENOMEM;

                    err = scan_all(ubi, ai, 0);
                } else {
                    err = scan_all(ubi, ai, UBI_FM_MAX_START);
                }
            }
        }
    #else
        err = scan_all(ubi, ai, 0);
    #endif
        if (err)
            goto out_ai;

        ubi->bad_peb_count = ai->bad_peb_count;
        ubi->good_peb_count = ubi->peb_count - ubi->bad_peb_count;
        ubi->corr_peb_count = ai->corr_peb_count;
        ubi->max_ec = ai->max_ec;
        ubi->mean_ec = ai->mean_ec;
        dbg_gen("max. sequence number:       %llu", ai->max_sqnum);

        err = ubi_read_volume_table(ubi, ai);
        if (err)
            goto out_ai;

        err = ubi_wl_init(ubi, ai);
        if (err)
            goto out_vtbl;

        err = ubi_eba_init(ubi, ai);
        if (err)
            goto out_wl;

    #ifdef CONFIG_MTD_UBI_FASTMAP
        if (ubi->fm && ubi_dbg_chk_gen(ubi)) {
            struct ubi_attach_info *scan_ai;
```

```
        scan_ai = alloc_ai("ubi_ckh_aeb_slab_cache");
        if (!scan_ai) {
            err = -ENOMEM;
            goto out_wl;
        }

        err = scan_all(ubi, scan_ai, 0);
        if (err) {
            destroy_ai(scan_ai);
            goto out_wl;
        }

        err = self_check_eba(ubi, ai, scan_ai);
        destroy_ai(scan_ai);

        if (err)
            goto out_wl;
    }
#endif

    destroy_ai(ai);
    return 0;

out_wl:
    ubi_wl_close(ubi);
out_vtbl:
    ubi_free_internal_volumes(ubi);
    vfree(ubi->vtbl);
out_ai:
    destroy_ai(ai);
    return err;
}
```

以上函数概括起来主要是包括 scan 所有 nand block ，获取或者写入 ubi magic number 收集所有块的信息，如收集坏块信息建立坏块管理，收集可用块信息建立文件系统。

2 镜像写入

Nand 工程的下载全部过程主要在 u-boot15/common/dloader/dl_nand_operate.c 中。

dl_nand_write 函数将区分镜像在 mtd 分区 spl 和 uboot 或者是第三个 mtd 的 ubifs 分区。

```
static void _dl_nand_write(nand_info_t * nand, unsigned long long offset, unsigned long length, char *buffer)
{
    int ret;

    printf("_dl_nand_write:partname:%s,offset:0x%llx,len:0x%x\n",    dl_stat.mtd.name, offset, length);

    //TODO:temp here for step 1 debug
    if (strcmp(dl_stat.mtd.name, "splloader") == 0) {
        sprd_nand_write_spl(buffer, dl_stat.nand);
        printf("write spl\n");
        dl_stat.wp = 0;
        dl_stat.unsv_size = 0;
        return;
    }
    switch (dl_stat.part_type) {
    case PART_TYPE_MTD:
        ret = nand_write_skip_bad(nand, offset, &length, NULL, dl_stat.mtd.size, buffer, 0);
        dl_stat.unsv_size -= length;
        memmove(dl_stat.buf, dl_stat.buf + length, dl_stat.unsv_size);
        dl_stat.wp -= length;
        dl_stat.mtd.rw_point += length;
        if (ret) {
            /*mark a block as badblock */
            printf("nand write error %d, mark bad block 0x%llx\n", ret, dl_stat.mtd.rw_point & ~(nand->erasesize - 1));
            nand->_block_markbad(nand, dl_stat.mtd.rw_point & ~(nand->erasesize - 1));
        }
        break;
    case PART_TYPE_UBI:
        debugf("write volume %s\n", dl_stat.ubi.cur_volnm);
```

```
            ret = fdl_ubi_volume_write(dl_stat.ubi.dev, dl_stat.ubi.cur_volnm, buffer,
length);
            if (ret) {
                  printf("ubi volume write error %d!\n", ret);
                  return;
            }
            dl_stat.unsv_size -= length;
            memmove(dl_stat.buf, dl_stat.buf + length, dl_stat.unsv_size);
            dl_stat.wp -= length;
            break;
      default:
            debugf("part type error!\n");
            return;
      }

      return;
}
```

## 5.4. kernel

对于 kernel 配置，Emmc 和 Nand 版本的工程应包含两部分，一个是 dts 的配置，一个是 defconfig 的配置。

Emmc 版本配置：

Emmc dts：

```
sdio3: sdio@20600000{
            compatible = "sprd,sdhost-3.0";
            reg = <0 0x20600000 0 0x1000>;
            interrupts = <0 60 0x0>;
            sprd,name = "sdio_emmc";
            /*detect_gpio = <-1>; */
            SD_Pwr_Name = "vddemmccore";
            _1_8V_signal_Name = "vddgen0";
            signal_default_Voltage = <1800000>;
            ocr_avail = <0x00040000>;
            clocks = <&clk_emmc>, <&clk_384m>;
```

```
                base_clk = <384000000>;
                bus-width = <8>;
                caps = <0xC00F8D47>;
                caps2 = <0x202>;
                pm_caps = <0x4>;
                writeDelay = <0x34>;
                readPosDelay = <0x8>;
                readNegDelay = <0x8>;
        };
```

Emmc Defconfig:

```
# MMC/SD/SDIO Card Drivers#
CONFIG_MMC_BLOCK=y
CONFIG_MMC_BLOCK_MINORS=32
CONFIG_MMC_BLOCK_BOUNCE=y
CONFIG_MMC_BLOCK_DEFERRED_RESUME=y
CONFIG_SDIO_CARD=y
```

Nand 版本配置：

Dts:

```
sprd-nand@20c00000 {
    compatible = "sprd,sprd-nand";
    reg = <0x20c00000 0x1000>;
    vddnandio = "vddemmcio";
    vddnandcore = "vddemmccore";
    clocks = <&clk_nandc_2x>,<&clk_nandc_ecc>,<&clk_192m>,<&clk_256m>;
    version = <1>;
    };
```

Nand defconfig: 除了包含 emmc defconfig 以外，还需要增加 mtd 和 ubifs 的配置

```
# #MTD/nand Controller Drivers #
CONFIG_MTD_NAND_SPRD=y
CONFIG_MTD_OF_PARTS=y
CONFIG_MTD_CMDLINE_PARTS=y
CONFIG_MTD_UBI=y
CONFIG_MTD_UBI_WL_THRESHOLD=4096
CONFIG_MTD_UBI_BEB_LIMIT=20
# # File systems #
```

```
CONFIG_UBIFS_FS=y
CONFIG_UBIFS_FS_ADVANCED_COMPR=y
CONFIG_UBIFS_FS_LZO=y
CONFIG_UBIFS_FS_ZLIB=y
CONFIG_UBIFS_FS_DEBUG=y
```