交换机

刷机和升级指南

--- 1.1

1.0

修订记录

|  |  |  |  |
| --- | --- | --- | --- |
| **版本** | **日期** | **修订人** | **修订内容** |
| 1.1 | 2022/6/18 | 曾毅 | 更新nand 分区烧写 |
| 1.0 | 2022/5/23 | 曾毅 | 初稿 |
|  |  |  |  |

目录

[交换机 1](#_Toc208)

[刷机和升级指南 1](#_Toc30003)

[--- 1.1 1](#_Toc13268)

[1. 目的 4](#_Toc24961)

[2. 分区 5](#_Toc9465)

[2.1 通用分区 5](#_Toc6535)

[2.1.1 Nor flash 5](#_Toc20909)

[2.1.2 Nand flash 6](#_Toc12216)

[2.1.3 Nor+Nand 6](#_Toc27220)

[2.1.4 其他 6](#_Toc17342)

[2.2 我司交换机分区设计 6](#_Toc17778)

[2.2.1 设计原则 6](#_Toc22890)

[2.2.2 楠菲微分区设计 7](#_Toc18808)

[2.2.3 我司分区设计 7](#_Toc20450)

[2.3 其他问题 9](#_Toc27029)

[3. 楠菲微交换机刷机 10](#_Toc17301)

[3.1 楠菲微方案启动流程 10](#_Toc30332)

[3.1.1 bootloader刷机 10](#_Toc14461)

[3.1.2 uimage刷机 11](#_Toc18951)

[3.2 楠菲微交换芯片调试shell启动 12](#_Toc6065)

[4. 交换机升级 13](#_Toc19725)

[4.1 Connect挂载方案 13](#_Toc8796)

[4.1.1 Linux创建UBI FS分区 13](#_Toc4340)

[4.1.2 Uboot系统烧写image.ubifs 13](#_Toc19857)

[4.1.3 分区启动挂载 14](#_Toc17087)

[4.2 Linux Shell升级 15](#_Toc21907)

[4.3 命令行升级 16](#_Toc11181)

# 目的

本文档基于总体设计，介绍我司系统的分区设计原则，以及我司设备以及楠菲微样机的刷机方法。

# 分区

嵌入式系统启动有多种方案，比如从nor flash启动，从spi nor/nand flash启动，从nand flash启动。而启动程序，按照通用嵌入式开发方法（基于Linux系统）大体可以分为boot和image，根据flash介质的不同boot和image会放置到不同的flash分区。

## 通用分区

本章节介绍行业通常的分区方案，

### Nor flash

不管是spi nor flash还是并行nor flash都属于nor flash，它们之间的差别只是控制器读取flash的接口不同，但nor flash的介质属性是一样的。通常小的嵌入式设备会使用spi nor flash因为外部接口总线少，成本低，而对于主流CPU厂家比如NXP则喜欢用并行nor flash接口(elbc等)，这种接口地址线和数据线比较多，但是CPU可以直接寻址，性能很高，不管哪种接口，程序在flash上的存放都是一样的。

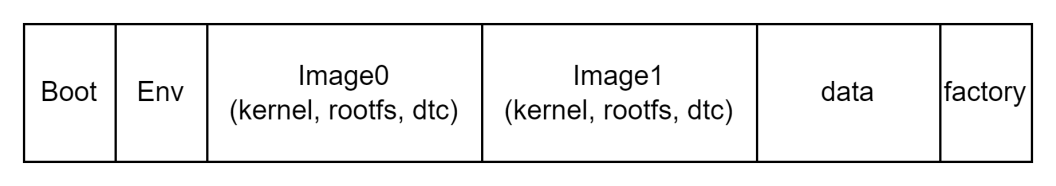
Nor flash有自身的介质属性，比如页和块的概念需要自行学习。

对于只有nor flash的嵌入式方案，通常的分区如下，

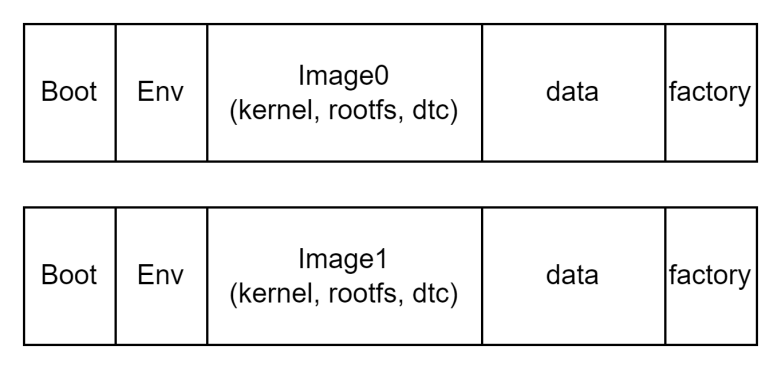


boot为启动代码，Env存放启动参数，Image通常是linux kernel，rootfs和dtc（设备树）结合体，有的方案也分开存放，data分区用于存放设备的配置文件，有些系统也可以将其挂载为jffs2文件系统，最后一部分一般被设计存放factory生产过程的参数或者预留。

这种分区有一些少量的变种，比如单芯片双分区，



双芯片双分区，



双芯片双分区要么CPU要支持，要么需要CPLD或者FPGA协助切换片选。

### Nand flash

Nand flash因为容量比较大，所以一般都是串行接口（地址线太多），不过串行接口也分为spi或者标准Nand flash接口。如果是单Nand flash方案通常CPU要支持从Nand flash启动模式且第一阶段boot不能大于4K字节（保证第一页不是坏块）。

Nand分区通常如下，



和nor flash最大的区别在于有一个用于能够检查坏块的boot。

其他的情况和nor flash类似，比如双分区等。

### Nor+Nand

nand flash启动比nor flash复杂，但是nor flash的容量和价格都比不上nand flash，所以将两者结合也是一种主流方案，一般的做法是将boot放到nor flash而将image放nand flash，这种方案的变种多种多样，有的把boot和kernel放nor flash，有的只将boot放nor flash，总的来说应该根据产品方案的特点以及flash的特性具体分析。

### 其他

除了nor和nand还有其他启动方案，但是在通信嵌入式行业用的比较少，比如sdcard（sdcard和nand的区别需要了解），固态硬盘启动等。

## 我司交换机分区设计

我司是设备集成厂商，所以可以预见会面对不同芯片厂商的不同方案设计，在这种情况下保证我司业务在不同芯片方案上的兼容性是我们的主要设计方向（这样可以减少维护工作量，减少不同方案之间的调整带来的不稳定性）。

### 设计原则

基于Linux方案的嵌入式系统最理想的方案是将业务功能按照标准的Linux目录进行放置，比如业务进程存放在/sbin或者/usr/sbin目录，业务动态库存放在/lib或者/usr/lib目录，如果方案要这样实施需要将业务功能和rootfs打包在一起，如果设备还要支持ISSU升级那么rootfs要么不压缩直接装到flash分区，要么支持squashfs+overlayfs，这些做法和有的芯片厂家提供的方案存在一些差异，会带来额外的工作量，比如楠菲微默认将kernel和rootfs打包成为一个uImage安装包。

基于以上原因，我们将业务代码按照Linux标准目录安装到/package目录（目录结构和Linux保持一致），根据芯片方案的不同，选择将”/package”目录下的内容覆盖”/”目录并打包成为rootfs或者将/package挂载为独立目录，通过增加/package/bin，/package/sbin，/package/lib等搜索路径支持业务开展。

### 楠菲微分区设计

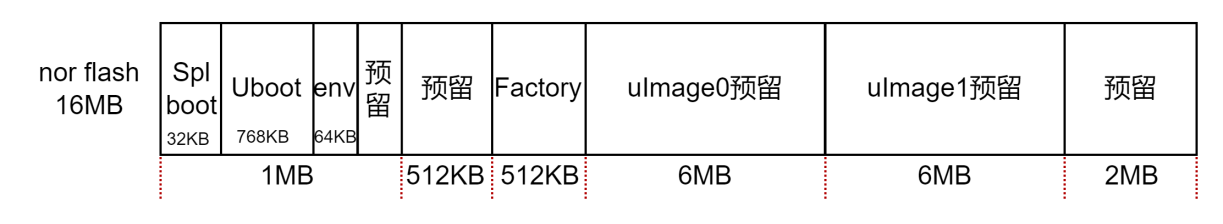
楠菲微的交换机方案为nor+nand的方式，在Demo机上nor flash只存放boot，而nand flash存放uImage。楠菲微Demo机分区如下，



很明显楠菲微为以后单nand方案做了预留，另外后面两个分区是楠菲微自己取的名字和实际的SDCARD一点关系也没有。

### 我司分区设计

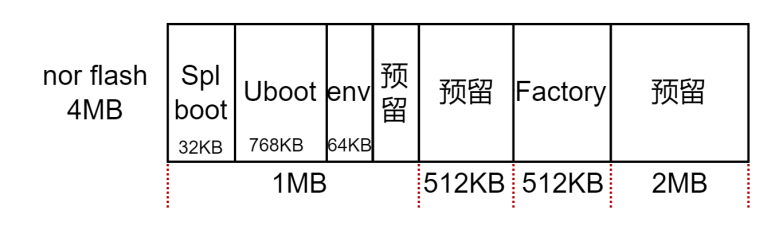
考虑到目前楠菲微的方案特点，我司对其方案的分区划分考虑如下，



首先是16MB nor flash的分区情况，spl boot和uboot预留1MB大小（目前已经看到楠菲微的uboot size在增长，目前大概有600KB左右），Env主要存放uboot的环境变量，Factory存放生产的一些配置数据，uimage0和uimage1做预留以后考虑将uimage放到nor flash中增加系统的稳定性，最后2MB空间预留作为未知处理。16MB分区地址划分，

|  |  |  |  |
| --- | --- | --- | --- |
| **分区** | **起始地址(0x)** | **长度(0x)** | **说明** |
| 预留 | 0x0 | 0x4000 | 楠菲微要求的预留部分，主要是为XIP预留 |
| Spl+Uboot | 0x4000 | 0xc8000 | 实际的boot是由spl boot和uboot两个部分组成，spl boot不依赖DDR直接在SRAM中执行，spl boot完成DDR初始化然后把uboot载入DDR再跳转执行 |
| Env | 0xc8000 | 0x10000 | 存放环境变量 |
| 预留 | 0xd8000 | 0x24000 | 预留 |
| 预留 | 0x100000 | 0x80000 | 预留 |
| Factory | 0x180000 | 0x80000 | 存放生产数据 |
| uImage0 | 0x200000 | 0x600000 | uImage0 |
| uImage1 | 0x800000 | 0x600000 | uImage1 |
| 预留 | 0xd00000 | 0x200000 | 预留 |

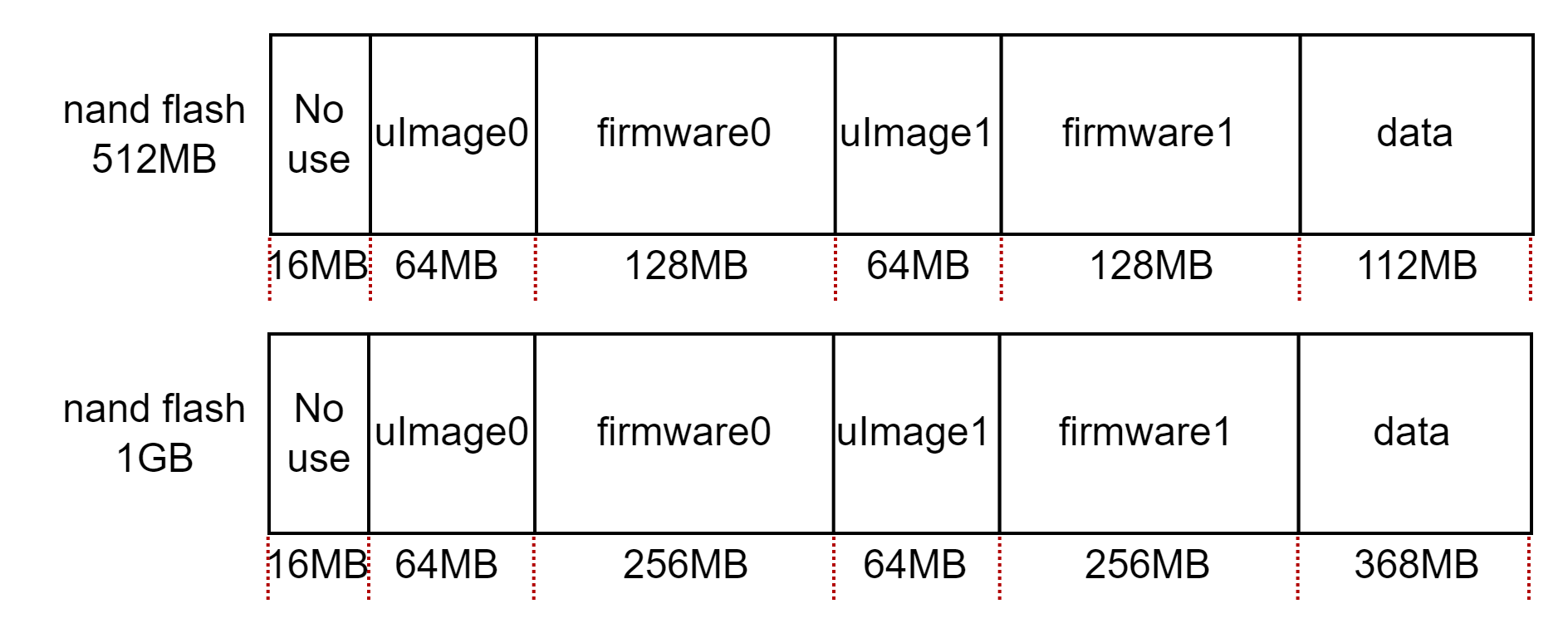
如果后续考虑降成本nor flash可能会使用4MB，那么分区如下，



4MB分区地址划分，

|  |  |  |  |
| --- | --- | --- | --- |
| **分区** | **起始地址(0x)** | **长度(0x)** | **说明** |
| 预留 | 0x0 | 0x4000 | 楠菲微要求的预留部分，应该是CPU内部还有ROM会使用这部分地址空间 |
| Spl+Uboot | 0x4000 | 0xc8000 | 实际的boot是由spl boot和uboot两个部分组成，spl boot不依赖DDR直接在SRAM中执行，spl boot完成DDR初始化然后把uboot载入DDR再跳转执行 |
| Env | 0xc8000 | 0x10000 | 存放环境变量 |
| 预留 | 0xd8000 | 0x24000 | 预留 |
| 预留 | 0x100000 | 0x80000 | 预留 |
| Factory | 0x180000 | 0x80000 | 存放生产数据 |
| 预留 | 0xd00000 | 0x200000 | 预留 |

对于Nand flash分区，我们在前面部分的预留和uImage分区和楠菲微保持一致，分区如下，



Nand flash考虑到成本和产品差异（接入和汇聚）可能支持512MB或者1GB大小的容量，首先我们和楠菲微的Demo机保持一致预留16MB空间，然后我们支持uImage0+Firmware0和uImage1+Firmware1双分区，最后的data分区用于存放设备的配置文件，日志等数据信息。其中需要注意的是uImage和firmware共同组成单个分区，不能允许只升级uImage或者firmware的情况发生（虽然有可能也可以正常工作）。

512MB分区地址划分，

|  |  |  |  |
| --- | --- | --- | --- |
| **分区** | **起始地址(0x)** | **长度(0x)** | **说明** |
| 预留 | 0x0 | 0x1000000 | 楠菲微Demo机的预留部分，我们最好保留 |
| uImage0 | 0x1000000 | 0x4000000 | uImage0 64MB |
| Firmware0 | 0x5000000 | 0x8000000 | Firmware0 128MB |
| uImage1 | 0xd000000 | 0x4000000 | uImage1 64MB |
| Firmware1 | 0x12000000 | 0x8000000 | Firmware1 128MB |
| data | 0x1a000000 | 0x7000000 | date分区112MB |

1GB分区地址划分，

|  |  |  |  |
| --- | --- | --- | --- |
| **分区** | **起始地址(0x)** | **长度(0x)** | **说明** |
| 预留 | 0x0 | 0x1000000 | 楠菲微Demo机的预留部分，我们最好保留 |
| uImage0 | 0x1000000 | 0x4000000 | uImage0 64MB |
| Firmware0 | 0x5000000 | 0x10000000 | Firmware0 128MB |
| uImage1 | 0x15000000 | 0x4000000 | uImage1 64MB |
| Firmware1 | 0x19000000 | 0x10000000 | Firmware1 128MB |
| data | 0x29000000 | 0x17000000 | date分区112MB |

## 其他问题

分区比较容易遇到的吐血问题是分区空间不够的情况，这种情况，

第一要防止破坏系统已有的数据，首先业务在Linux的文件系统的框架下是安全的，其次在升级的时候软件要进行分区大小和文件大小的检查；

第二如果容量不够，怎么进行分区重新划分或者相同型号前后不同容量（比如前期设备用512MB后面发现容量不够，后续设备采用1GB大小容量）如何识别，目前来看自适应应该是比较困难的，通过在eeprom或者nor flash前面的数据区中标识分区信息是具备实操的方法。

# 楠菲微交换机刷机

楠菲微SOC内部有ROM可以在nor flash无法启动的情况下通过硬件跳线启动，并通过x-modem下载boot重新烧写nor flash。

## 楠菲微方案启动流程

这里简单描述nor flash启动流程，详细可以参考《SF90xx\_SOC平台软件使用手册》。

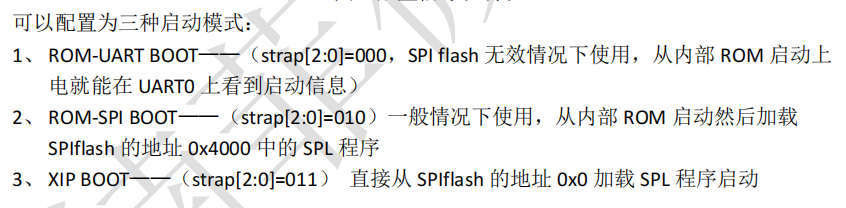
1. 上电CPU加载nor flash 0x4000偏移开始的16kB spl boot程序到sram中执行，这部分代码主要初始化CPU寄存器和DDR控制器，然后将接下来的uboot加载到DDR中；
2. 跳转到DDR uboot执行，uboot主要初始化CPU外部控制器，楠菲微的uboot把交换芯片也做了一定程度的初始化；
3. uboot根据启动命令加载nand flash中的uImage，解压并启动；
4. uImage中的内核启动并挂载rootfs，然后根据启动参数挂载其他分区。

### bootloader刷机

一般情况下spl boot和uboot会被打包成为bootloader.bin进行统一烧写，我们应该较少可能会单独升级spl，bootloader在uboot下通过命令升级，

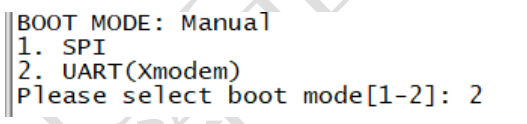
|  |
| --- |
| //在上电开始spl boot启动时可以通过ctrl+c暂停，通过y-modem下载uboot，  //进入uboot以后可以通过以下命令刷bootloader.bin  uboot#setenv ipaddr 172.31.4.53  uboot#setenv serverip 172.31.4.52  uboot#ping 172.31.4.52  uboot#swwrite 0x2f 0x17 0x403 //调整带外口RGMII delay，防止错包  uboot#swwrite 0x2f 0x16 0x7 //调整带外口RGMII delay，防止错包  //PC 端启动 tftp 服务，把 bootloader.bin 放到 tftp 下载目录  uboot#updateboot  //下载刷新完毕，重启 |

如果设备变砖，只能先通过硬件跳线到第一种模式(000)刷新nor flash恢复，默认我们是(010)，



在模式0的情况下，可以通过以下过程把Uboot加载运行起来，然后在Uboot下通过tftp下载烧写文件到DDR，在使用Flash命令把烧写文件写到Flash上。

1. 调整硬件引脚 strap[2:0]=000，Cpu 上电进入到 shell 界面



2. 通过输入对应的编号 2，选择 UART 启动 SPL

3. 通过 X-modem 方式下载 PC 上的 SPL 到片上 SRAM 启动运行

4. 启动 SPL 过程中按组合键 ctrl + C 进入到 SPL shell

5. 执行命令 ymodem2ddr，从 PC 通过 Y-modem 方式下载 uboot 到 ddr 运行

6. Uboot 启动 shell 后，使用 ping 192.168.0.2 方式 ping 通 PC 机器；

7. PC 端启动 tftp 服务，把 bootloader.bin 放到 tftp 下载目录

8. 在 Uboot shell 下执行 updateuboot 命令，该命令会下载 192.168.0.2 的机器上的

bootloader.bin 文件，然后烧写 spi-flash。正常情况下应该能看到 tftp 下载过程以及 spi falsh 烧写过程，等待烧写结束，断电重启。

### uimage刷机

我们的image分为两部分：uImage和upgrade，uImage是内核和rootfs的打包文件会被直接烧写到nand flash的image0和image1分区中，而upgrade在firmware分区中，由于firmware分区采用ubi文件系统所以片烧upgrade文件是通过mkfs.ubi制作的ubi文件，或者采用upgrade.tar.gz升级压缩包，通过解压缩覆盖对应分区。

片烧的方式可以在uboot和linux系统两个场景下完成升级，而升级压缩包只能在linux系统场景下升级。

#### uboot场景刷新uImage

uboot场景下刷新uImage步骤如下，

|  |
| --- |
| //在上电开始spl boot启动时可以通过ctrl+c暂停，通过y-modem下载uboot，  uboot#setenv ipaddr 172.31.4.53  uboot#setenv serverip 172.31.4.52  uboot#ping 172.31.4.52  uboot#swwrite 0x2f 0x17 0x403 //调整带外口RGMII delay，防止错包  uboot#swwrite 0x2f 0x16 0x7 //调整带外口RGMII delay，防止错包  tftpboot 0x83000000 uImage  nand erase 0x1000000 0x4000000 //uImage0  nand write 0x83000000 0x1000000 0x4000000 //uImage0  //刷完重启 |

#### linux场景刷新uImage

Linux场景下刷新uImage步骤，

|  |
| --- |
| ifconfig eth0 172.31.4.53 netmask 255.255.255.0  cd /tmp  tftp -g -l uImage 172.31.4.52  flash\_eraseall /dev/mtd0 //通过cat /proc/mtd查看正确的分区  nandwrite -s 0x0 /dev/mtd0 uImage -p |

## 楠菲微交换芯片调试shell启动

3.1章节启动以后就是一个裸的linux环境，这时可以启动我司的业务代码（详见第4章节），也可以启动楠菲微自带的sdk调试shell。

可以通过以下命令挂载楠菲微调试shell，

|  |
| --- |
| #我在demo机的/data目录下存了一个user\_sdk.tar.gz压缩包，如果没有则需要编译楠菲微sdk将sf\_core.ko和diag\_shell-strip下载到系统中。  cp /data/user\_sdk.tar.gz /tmp  cd /tmp  tar xzf user\_sdk.tar.gz  ls /tmp/user\_sdk  #插入sdk内核组件，插入前可以通过lsmod先看，如果已经插入就不用再插入了  cd /tmp/usr\_sdk  insmod sf\_core.ko  #启动楠菲微shell  diag\_shell-strip |

# 交换机升级

上一章节主要介绍楠菲微的目标文件升级，本章主要介绍connect系统的文件升级。

## Connect挂载方案

### Linux创建UBI FS分区

设备因为各种原因导致NAND Flash被擦除会导致业务分区全部丢失，最简单的方法就是手动重新分区，并通过4.2章节将业务程序升级。

首先NAND Flash是被MTD子系统管理，我们通过Uboot的环境变量将MTD分区告诉了内核，而分区的信息是根据2.2.3章节我司的分区设计决定，由于no use/uimage0/uimage1分区都不是ubi fs所以不用创建UBI分区，只有firm0/fimr1/data三个分区需要创建UBI分区。

首先查看firm0/firm1/data对应的MTD分区，

|  |
| --- |
| / # cat /proc/mtd  dev: size erasesize name  mtd0: 01000000 00020000 "resv"  mtd1: 04000000 00020000 "image0" #kernel uImage  mtd2: 08000000 00020000 "firm0" #connect 业务  mtd3: 04000000 00020000 "image1" #kernel uImage  mtd4: 08000000 00020000 "firm1" #connect 业务  mtd5: 07000000 00020000 "data" #数据区，主要放运行过程中的数据  / # |

可以看到firm0对应mtd2，firm1对应mtd4，data对应mtd5。通用以下命令挂载UBI分区，

|  |
| --- |
| #如果ubifs已经被挂载，需要通过命令卸载，查看ls /dev  / # ls /dev  ...  mtd0 network\_throughput ram3 ubi0  mtd0ro null ram4 ubi\_ctrl  ...  /#ubidetach -d 0 /dev/ubi\_ctrl  #挂载firm0  / # flash\_eraseall /dev/mtd2  Erasing 128 Kibyte @ 8000000 - 100% complete.  / #ubiformat /dev/mtd2  ubiformat: mtd2 (nand), size 134217728 bytes (128.0 MiB), 1024 eraseblocks of 131072 bytes (128.0 KiB), min. I/O size 2048 bytes  libscan: scanning eraseblock 1023 -- 100 % complete  ubiformat: 1024 eraseblocks have valid erase counter, mean value is 1  ubiformat: formatting eraseblock 1023 -- 100 % complete  / #ubiattach /dev/ubi\_ctrl -m 2  [ 724.807066] UBI: attaching mtd2 to ubi0  [ 724.913689] UBI: scanning is finished  [ 724.917373] UBI: empty MTD device detected  [ 724.938244] UBI: attached mtd2 (name "firm0", size 128 MiB) to ubi0  [ 724.951007] UBI: PEB size: 131072 bytes (128 KiB), LEB size: 126976 bytes  [ 724.959266] UBI: min./max. I/O unit sizes: 2048/2048, sub-page size 2048  [ 724.966031] UBI: VID header offset: 2048 (aligned 2048), data offset: 4096  [ 724.972897] UBI: good PEBs: 1024, bad PEBs: 0, corrupted PEBs: 0  [ 724.978912] UBI: user volume: 0, internal volumes: 1, max. volumes count: 128  [ 724.986036] UBI: max/mean erase counter: 0/0, WL threshold: 4096, image sequence number: 266387135  [ 724.994974] UBI: available PEBs: 940, total reserved PEBs: 84, PEBs reserved for bad PEB handling: 80  [ 725.004173] UBI: background thread "ubi\_bgt0d" started, PID 69  / # ls /dev  ...  mtd0 network\_throughput ram3 ubi0  mtd0ro null ram4 ubi\_ctrl  ...  / #ubimkvol /dev/ubi0 -N firm0 -m  / # ls /dev  ...  i2c-1 net ram2 ubi0  kmsg network\_latency ram3 ubi0\_0  ...  / #mount -t ubifs /dev/ubi0\_0 /firm0  [ 1078.895152] UBIFS: default file-system created  [ 1078.903391] UBIFS: background thread "ubifs\_bgt0\_0" started, PID 94  [ 1078.932526] UBIFS: mounted UBI device 0, volume 0, name "firm0"  [ 1078.938457] UBIFS: LEB size: 126976 bytes (124 KiB), min./max. I/O unit sizes: 2048 bytes/2048 bytes  [ 1078.947554] UBIFS: FS size: 117960704 bytes (112 MiB, 929 LEBs), journal size 5967872 bytes (5 MiB, 47 LEBs)  [ 1078.957336] UBIFS: reserved for root: 4952683 bytes (4836 KiB)  [ 1078.963152] UBIFS: media format: w4/r0 (latest is w4/r0), UUID 144985D7-45D7-4ECB-890D-9AC5D079CD62, small LPT model  #firm1  / # ubiformat /dev/mtd4 #用flash\_eraseall /dev/mtd4也可以  / # ubiattach /dev/ubi\_ctrl -m 4  / # ubimkvol /dev/ubi1 -N firm1 -m  / # mount -t ubifs /dev/ubi1\_0 /firm1  #data  / # ubiformat /dev/mtd5 #用flash\_eraseall /dev/mtd5也可以  / # ubiattach /dev/ubi\_ctrl -m 5  / # ubimkvol /dev/ubi2 -N data -m  / # mount -t ubifs /dev/ubi2\_0 /data |

### Uboot系统烧写image.ubifs

注意：本章节在UBOOT烧写时还有一些问题，暂时不要使用

Connect系统编译完成后生成imge.tar.gz文件，通过解压缩得到image目录（内含所有所有业务数据），使用mkfs.ubi可以将image目录制作成image.ubifs。

|  |
| --- |
| ./mkfs.ubi –r ./image –m 2048 –e 128KiB –c 1024 –o image.ubifs |

Mkfs.ubi文件在connect仓库的tools目录下。#确认真假

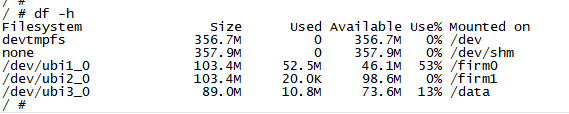
在uboot烧写image.ubifs的流程如下，首先介入uboot系统，然后下载image.ubifs文件，最后烧写到对应分区，

|  |
| --- |
| setenv ipaddr 172.31.4.53  setenv serverip 172.31.4.52  ping 172.31.4.52  swwrite 0x2f 0x17 0x403  swwrite 0x2f 0x16 0x7  tftpboot 0x83000000 image.img  nand erase 0x5000000 0x8000000  ubi create firm0  ubi part firm0  ubi write 0x83000000 firm0 0x3200000 #0x3200000和最后image.ubifs的大小有关 |

### 分区启动挂载

前面讲了不同的芯片公司提供了不同的uimage方案，很不幸楠菲微提供的方案比较保守，他将kernel和rootfs打包压缩成为一个文件，在uboot启动阶段将对应文件解压，这种方式无法支持ISSU升级（组件化升级），基于这个情况我们采取挂载独立/package目录的方式运行整个业务。

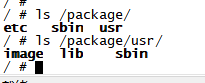
具体的方案为，首先按照2.2.3章节挂载对应分区，



可以从上图看到/firm0和/firm1业务分区，以及/data数据分区（主要放配置文件和log）。分区的挂载通过在/etc/init.d/rcS启动脚本中进行，

|  |
| --- |
| / # cat /etc/init.d/rcS  # rmmod of newer version of busybox needs this direcotry to exist  mkdir -p /lib/modules/`uname -r`  #这里有些其他动作...略过  #挂载/firm0分区  ubiattach -m 2 -d 1 /dev/ubi\_ctrl  if ! [ -d /firm0 ]; then  mkdir -p /firm0  fi  mount -t ubifs /dev/ubi1\_0 /firm0  #挂载/firm1分区  ubiattach -m 4 -d 2 /dev/ubi\_ctrl  if ! [ -d /firm1 ]; then  mkdir -p /firm1  fi  mount -t ubifs /dev/ubi2\_0 /firm1  #挂载/data分区  ubiattach -m 5 -d 3 /dev/ubi\_ctrl  if ! [ -d /data ]; then  mkdir -p /data  fi  mount -t ubifs /dev/ubi3\_0 /data  #将当前执行分区firm0或者firm1建立软链接/package  将当前执行分区的lib目录建立软链接为/usr/lib，这样大多数进程  可以通过标准目录/usr/lib搜索到相关库。  选择当前执行分区的算法由升级组件决定，目前还未调试默认挂载firm0。  #do some check  ln -sf /firm0 /package  if [ -d /package/usr/lib ]; then  ln -sf /package/usr/lib /usr/lib  fi  #启动业务进程  /package/etc/autorun start |

业务进程的启动过程要参考service-app-mgmt组件的设计文档（现在还没写），这里进行简单介绍，可以看到/package目录和linux目录基本保持一致，



所以业务组件的启动脚本在/package/etc/rcS.d/，



启动脚本Sxx会按照数字从小到大执行，S52用于挂载交换芯片内核ko模块，S60启动业务的主要进程app-monitor，其它业务进程的启动是由app-monitor进程中的service-app-mgmt组件通过/package/etc/monitor.config配置文件决定。

## Linux Shell升级

当前阶段由于升级组件还未调试，所以只有通过在Linux shell下手动刷新业务程序，如果要刷新整个业务分区执行以下动作，

|  |
| --- |
| #杀掉当前运行的业务进程  cp /package/usr/sbin/kill.sh /data/  cd /data  ./kill.sh  cd -  #下载image，删除分区业务进程，拷贝新进程  cd /tmp  rm -f image.tar.gz  rm -rf image  ifconfig eth0 172.31.4.53  tftp -g -l image.tar.gz 172.31.4.52  tar xzf image.tar.gz  rm -rf /package/\*#如果只是替换其中的一些库和进程就不用删除整个目录  cp -rf ./image/\* /package/  #手动拉起app-monitor  /package/usr/sbin/app-monitor &  #待系统启动到sync....100%，可以启动命令行  /package/usr/sbin/mgmt-cmd |

## 命令行升级

//待升级组件调试完成后再完善