CKT

高通 EFS Introduction

kaiyu.wang@ck-telecom.com
2014/10/24

目录

什么是 EFS

Embedded File System，高通为 modem 端实现的一套文件系统，用以维护所有 NV 参数和一些系统配置。

这套文件系统只可被 modem 访问，而对 AP 端加密，但可以通过固定接口或者工具在 AP 端访问。

高通在 modem 端实现了类 posix 的 API 做操作接口。这些接口提供了和一般 fs 同样的操作接口，例如 efs_open,efs_write，efs_read 等。我们几乎不会有机会通过代码去直接操作存在于 EFS 中的文件。

EFS 的物理存储仍然是在传统的 emmc 中，高通开辟了单独的三个分区给 EFS，用以实现一些机制。但是 modem 并没有办法直接去访问 emmc，所以 modem 会将参数保存在 RAM 中，并在合适的时间通过一些通讯接口将数据写回到 AP 端的 EMMC 中。

同样，为了实现对 AP 保密的机制，RAM 中开辟了两块 BUFFER,一块给 Modem 放正确的 EFS，一块给 AP 放加密后的 EFS，用以写回 EMMC。

EFS 和 QCN 及 NV

NV 是 nonvolatile memory，用以存放手机设备的一些参数和配置，例如 RF 参数，硬件软件配置，各种地址等等。当手机启动后，modem 会通过读取这些 NV 项的值配合软件的逻辑对系统进行配置。

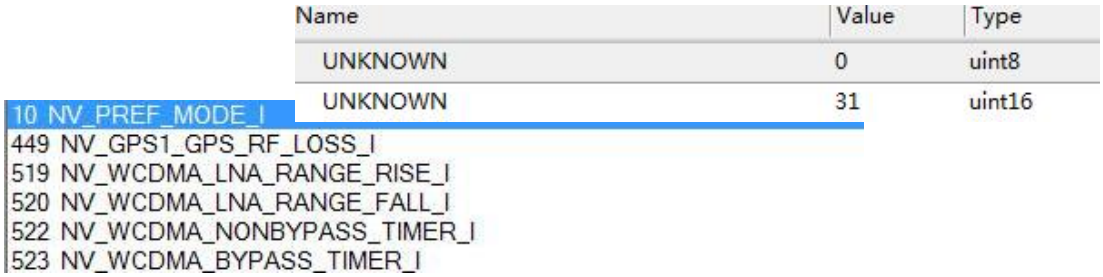
NV 的定义

```
<NvItem id="10" name="NV_PREF_MODE_I" permission="readwrite">
    <Member type="uint8" sizeOf="1" name=""/>
    <Member type="uint16" sizeOf="1" name=""/>
</NvItem>
```

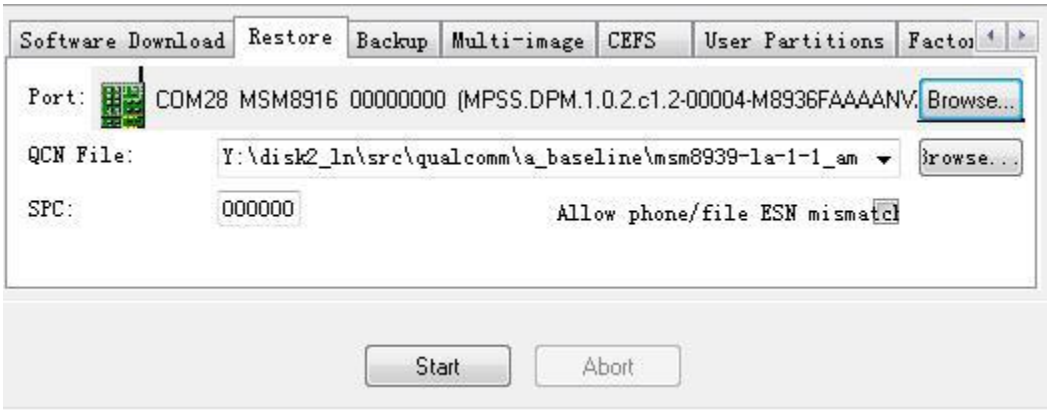
NV 的配置

```
<NvItem id="10" subscriptionid="0" name="NV_PREF_MODE_I"
mapping="direct" encoding="dec" index="0">0,31</NvItem>
```

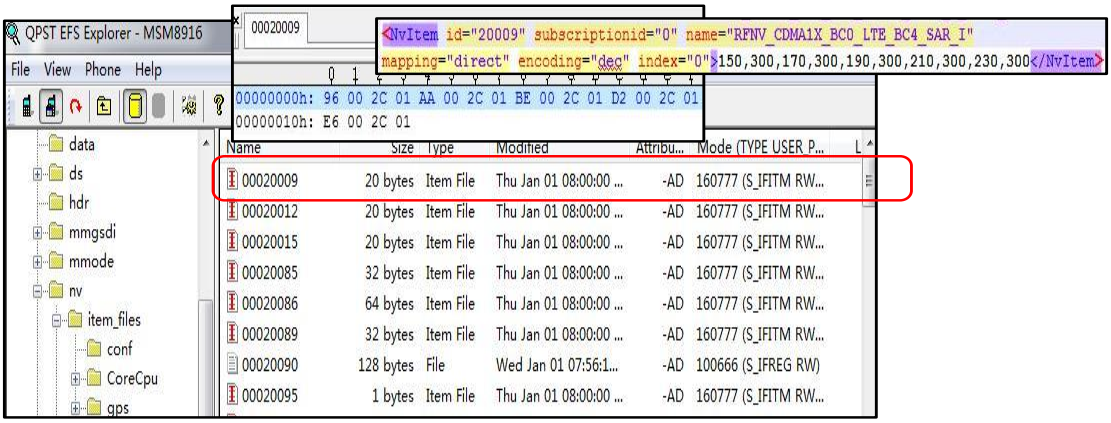
QCN 是高通存储 NV 数据的一种文件格式，可以理解为大量的 NV 项和对应数据结构及它们的值被打包在一起。QCN 工具可以通过 QRCT 的 NV manager 打开，也可以通过 QRCT 的 NV Tools 转换成可以被文本编辑的 XML 文件。



QCN 文件本身不便于直接编辑，多用于工具下载，如 QPST 中的 software download 工具可以方便的进行下载和备份 QCN 文件。例如当手机中已有校准数据时，可以用工具将 QCN 进行备份：



每一个 NV 项在 EFS 中都以一个文件的形式存在，烧写一个 QCN 到手机中，最终会将 QCN 中的所有 NV 数据导入到 EFS 分区中，可以通过 QPST 的 EFS Explorer 进行查看。

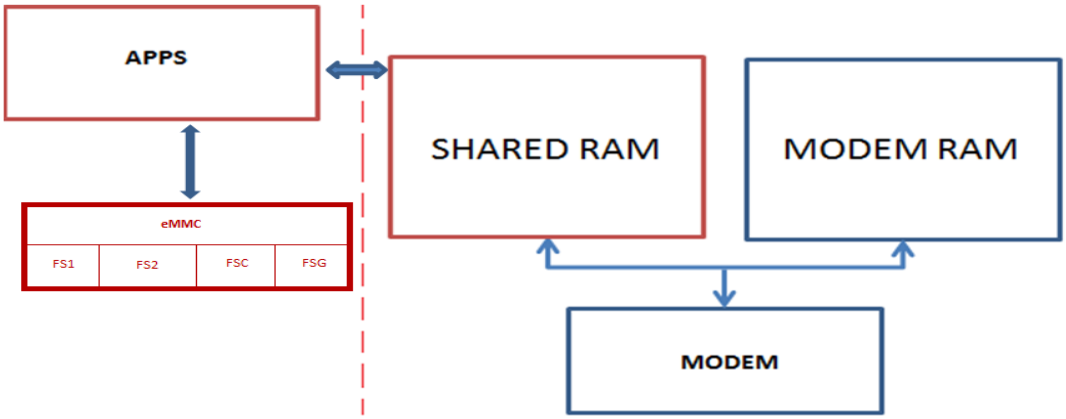


由此，QCN 是 NV 的集合和载体，EFS 是实际存储和操作 NV 的文件系统。当然 EFS 中不仅仅包括 NV 的数据，它的用处主要还是给 modem 提供文件系统，NV 数据只是其中非常重要的一部分。

EFS 的机制

1. EFS 的整体框架

EFS 中的数据对于设备来说非常重要，因此高通为了保证数据安全以及防止数据损坏，做了一系列的措施，这些措施包括双 BUFFER，多分区。

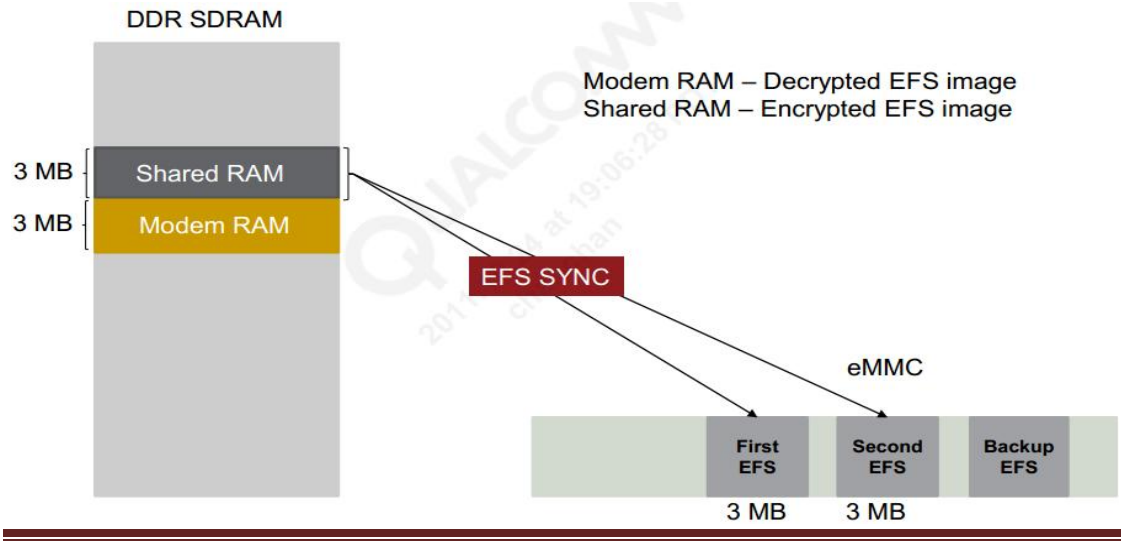


图中红色部分加密，AP 访问；蓝色部分是解密后的内容，modem 访问。

2. EFS 的 sync 机制

所有的 EFS 数据由 modem 管理，并存放在 modem ram 的 buffer 中。modem 会周期性的将 buffer 中数据进行加密并拷入 shared ram buffer 供 AP 访问，AP 接收到消息后，会根据 modem 的指令，将数据写入 EMMC 中 first 或者 second EFS 分区。

具体写入哪个分区，通常如果这一次写入 first，那么下一次就写入 second。以这种机制来保证两个分区的数据尽可能的相同。



modem 的 fs_task 会在执行中会等待系统发给它 FS_EFS_SYNC_SIG：

```
if ((task_sigs & FS_EFS_SYNC_SIG) != 0)
{
    FS_MSG_MED ("EFS sync timer sig received", 0, 0, 0);
    (void) fs_os_clr_sigs (fs_os_self (), FS_EFS_SYNC_SIG);
    fs_sync_timer_signal_received ();
}
```

在 fs_sync_timer_signal_received ()中进行处理：

```
void fs_sync_timer_signal_received (void)
{
  int result;
  /* The only error codes accepted are ENODEV or ETIMEDOUT*/
  result = efs_sync ("/"); //调用 efs_sync 接口，"/"指定为根目录
  if (result != 0)
  {
    FS_MSG_HIGH ("%d error for implicit efs_sync", efs_errno, 0 , 0);
    ASSERT ((efs_errno == ENODEV) || (efs_errno == ETIMEDOUT) ||
            (efs_errno == EPERM));
  }
}
```

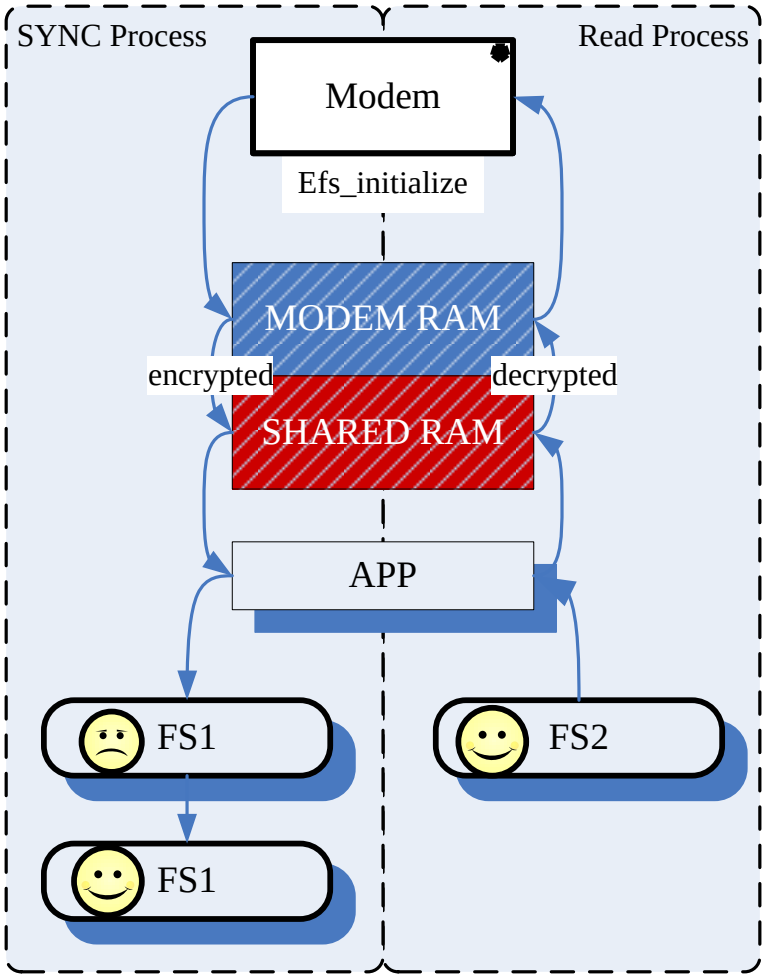
efs_sync 调用 rmts 的接口和 AP 进行通信，这里不做展开。

2. EFS 的分区

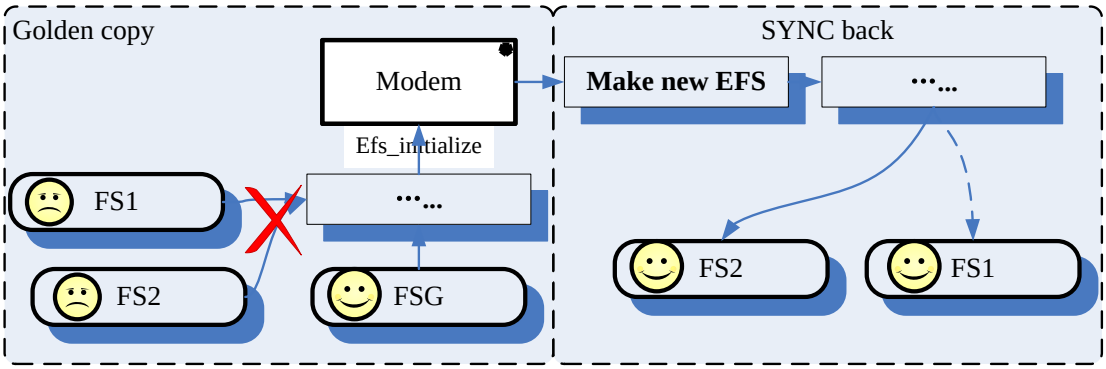
高通为 EFS 的功能开辟了三个分区，或者更准确说是四个：

```
<partition label="modemst1" size_in_kb="1536"
<partition label="modemst2" size_in_kb="1536"
<partition label="fsg" size_in_kb="1536"
<partition label="fsc" size_in_kb="1"
```

这四个分区存在于 EMMC 中，modemst1 和 modemst2 是两个互为备份的分区。如果写入时突然掉电或者其他意外情况损坏其中一个分区的数据，还有另外一个分区数据可用。因此在启动过程中 modem 会去依次检查 fs1 和 fs2 的分区数据，如果 fs1 损坏就使用 fs2，并在下一次 sync 发生时将数据写回 fs1。



如果 fs1 和 fs2 都损坏，就会检查 fsg 分区的数据，如果可用就会将 fsg 分区数据提取到 modem ram 中，并在 sync 中写回 fs1 和 fs2。我们所采用的 golden copy 就是利用了这种机制。



还有一种情况，当 fs1,fs2,fsg 三个分区的数据都不可用，通常空板或者擦除了 EFS 分区的手机就是这样。为了保证 Modem 启动后有可用的文件系统，Modem 的 fs_task 在初始化 efs 时会在 modem ram 中制作一个全新的 EFS，并写回到 EMMC 中。

fsc 分区只有一个 1kb，实际上类似于 android 端 misc 分区，它是制作 golden copy 的一部分，在工具端通过 diag 接口对 modem 发送命令：EFS2_DIAG_MAKE_GOLDEN_COPY，modem 会将一个 backup_cookie 写入到 fsc 分区中，下一次启动 sbl 检查到这个 cookie，会将 fs1 和 fs2 的数据考入 FSG 分区。但实际上我们并没有这样用。

4. EFS 的 RAM 机制

modem ram

SDRAM 中开辟了一块 3MB 大小的 buffer 用于 modem ram,每一个 page 的大小为 512byte,总共有 6114 个 page。

EFS 的所有文件和目录数据首先保存在 modem ram 中，且只供 modem 访问，AP 不能访问。在适当的时机，再通过固定的接口写回 EMMC。

这些数据的节点在 modem ram 中被直接映射到 page 上，因此不需要坏块管理和垃圾收集等机制，这也符合 NV 本身的机制，即只能被覆盖而无法被删除。

modem ram 的第一和最后一个 page 是 super block，super block 中声明了整个 modem ram 的时间，用以检查哪个分区数据是当前最新的。super block 是 EFS 镜像中很重要的一个 page，EFS 是通过检查此 page 的数据是否有效来确定整个镜像是否可用。

```
PACKED struct fs_ramfs_superblock_data_type
{
  uint32 header;
  uint32 version;
  uint32 magic1;
  uint32 magic2;
  uint32 age; //当前 super block 的更新时间
  uint32 upper_data[FS_UPPER_DATA_COUNT];
  uint8  pad[FS_RAMFS_SUPERBLOCK_PAD_SIZE];//数据填充
}PACKED_POST;
```

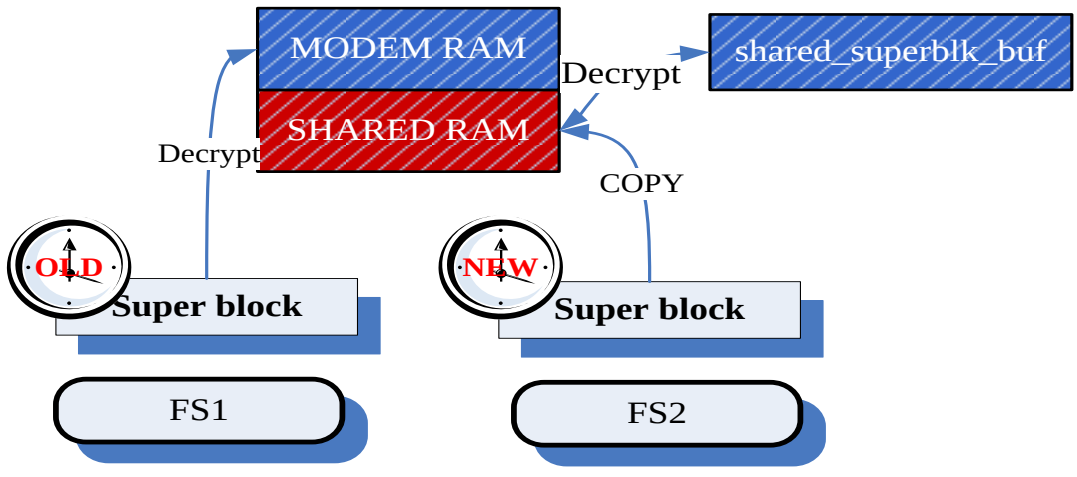
shared ram

大小同为 3MB，主要作用是存放加密后的 modem ram 内容供 AP 写回 EMMC。在 EFS 初始化过程中也作为临时的 buffer 进行使用。

同理，shared ram 的第一个 block 和最后一个 block 也是 super block。

boot up

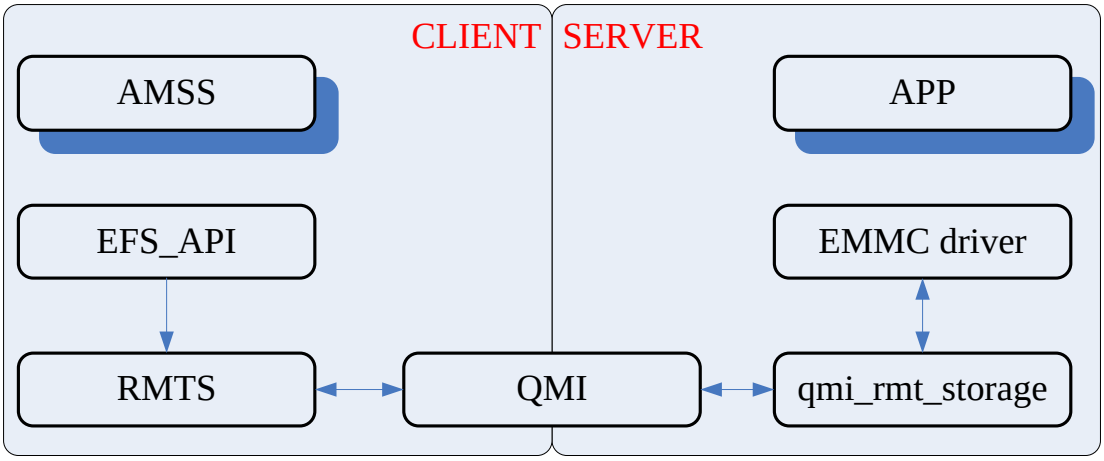
modem 初始化时将 fs1 和 fs2 的分区数据分别 copy 到 modem ram 和 shared ram 中，分别进行解密并读取 super block 数据，查找最新的那个分区数据，并将分区数据 copy 到 modem ram 中。



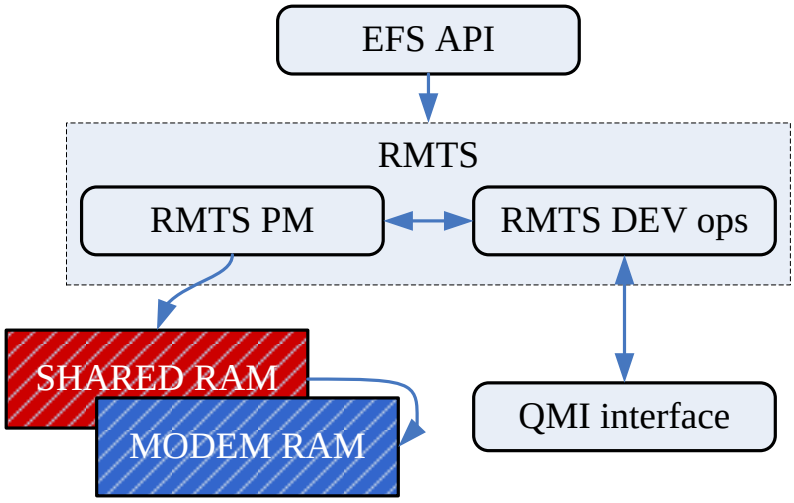
EFS 的接口

RMTS

RMTS 的全称是 Remote Storage Client Interface。
一方面 RMTS 掌管着 modem ram 和 shared ram 中的数据，modem ram 中的数据就是 modem 事实上使用的 efs copy；另一方面 RMTS 还要兼顾着和 QMI 接口通信，以便于对 APP 端的 EMMC 进行读写等操作。
rmts 的框架



RMTS 从结构上可以分成两个部分，page manger 和 DEV。



RMTS page manager

page manager 的职责是对两块 buffer 进行管理，执行 EFS 的 API 时按照需要从两块 buffer 存

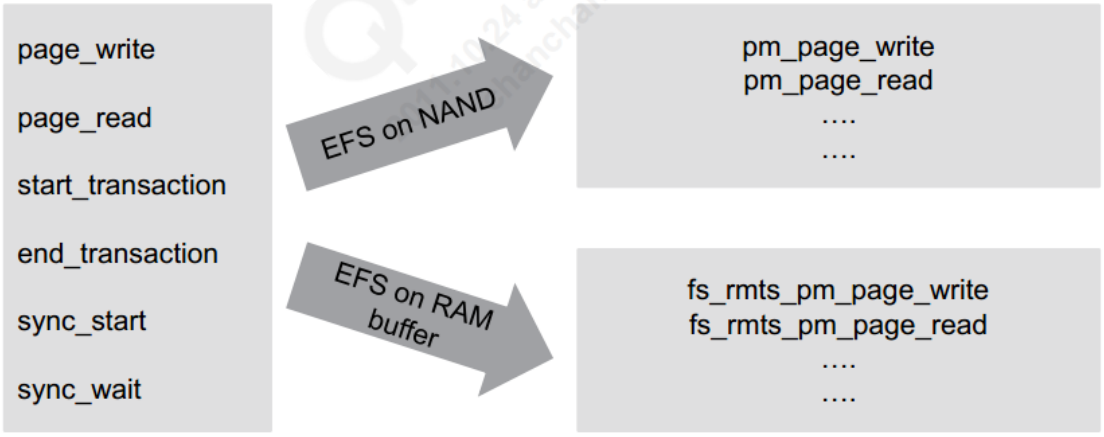
取数据，并将数据交给 DEV 的 ops 去处理。
从 APP 的 emmc 读到的数据会先放到 shared ram 中，再解密并验证后再 copy 到 efs copy 既 modem ram 的指定地址上。

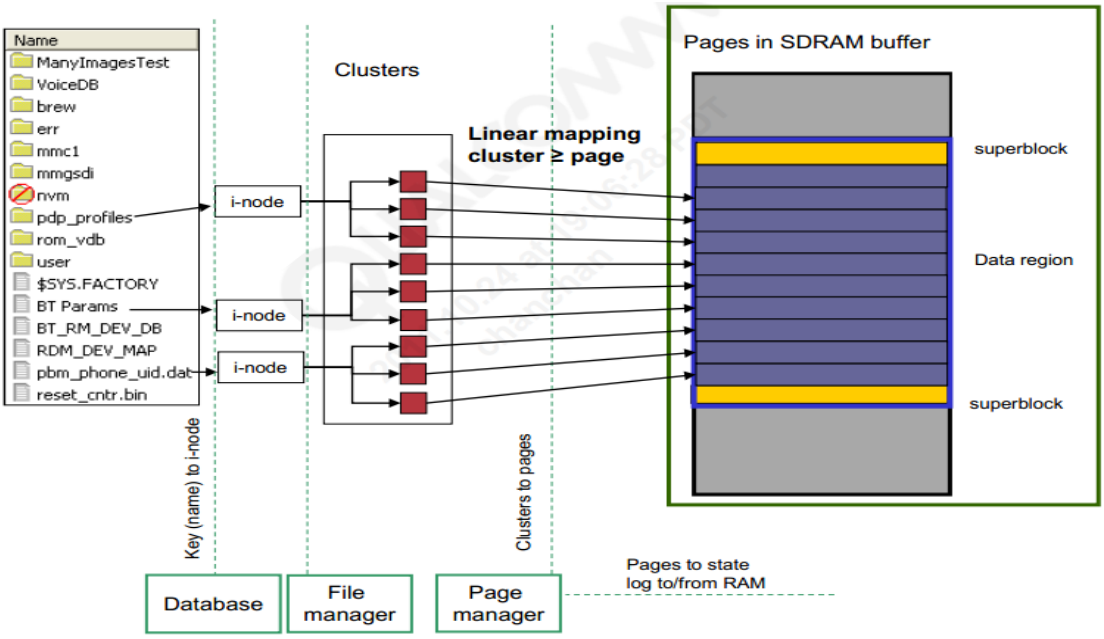
RMTS 的接口主要位于：Fs_rmts_pm.c (core\storage\efs\src)

```
static void
fs_rmts_pm_init_ops (fs_pm_t pm_data)
{
  ASSERT (pm_data != NULL);
  pm_data->ops.page_write = fs_rmts_pm_page_write;
  pm_data->ops.page_read = fs_rmts_pm_page_read;
  pm_data->ops.store_info = fs_rmts_pm_store_info;
  pm_data->ops.get_info = fs_rmts_pm_get_info;
  pm_data->ops.start_transaction = fs_rmts_pm_start_transaction;
  pm_data->ops.end_transaction = fs_rmts_pm_end_transaction;
  pm_data->ops.space_limit = fs_rmts_pm_space_limit;
  pm_data->ops.register_free_check_cb =
fs_rmts_pm_register_free_check_cb;
  pm_data->ops.register_alloc_count_cb =
fs_rmts_pm_register_alloc_count_cb;
  pm_data->ops.shred = NULL;
  pm_data->ops.scrub = NULL;
  pm_data->ops.read_iovec = NULL;
  pm_data->ops.write_iovec = NULL;
  pm_data->ops.sync_start = fs_rmts_pm_sync_start;
  pm_data->ops.sync_wait  = fs_rmts_pm_sync_wait;
  pm_data->ops.sync_end   = fs_rmts_pm_sync_end;
  pm_data->ops.sync_no_wait    = fs_rmts_pm_sync_no_wait;
  pm_data->ops.sync_get_status = fs_rmts_pm_sync_get_status;
  pm_data->ops.make_golden_copy = fs_rmts_pm_make_golden_copy;
  pm_data->ops.fs_init_complete = fs_rmts_pm_fs_init_complete;
  pm_data->ops.lock_efs = fs_rmts_pm_lock_efs;
  pm_data->ops.borrow_ram = fs_rmts_pm_fs_borrow_ram;
  pm_data->ops.store_data_in_sb = fs_rmts_pm_store_data_in_sb;
}
```

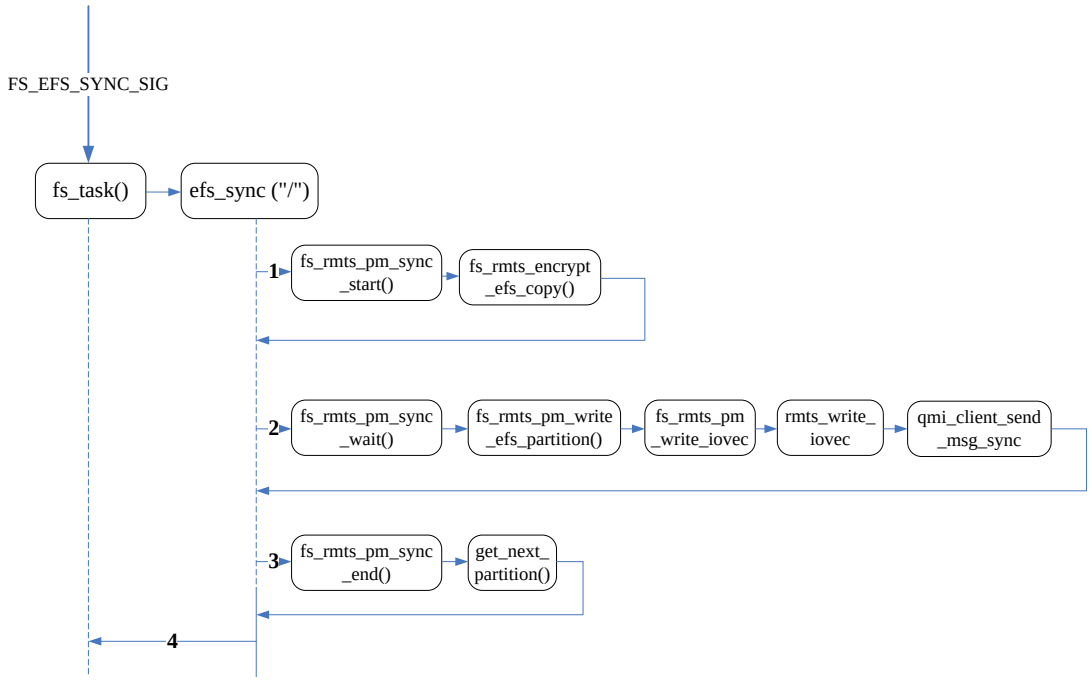
可以看到 ops 中的函数接口，跟 page 相关的为 ram 操作函数，和 sync 相关的为 APP 通信函数.另外，page manager 还需要跟踪两个 EMMC 分区中哪一个是目前激活的分区，在 sync 时就将数据写入激活的分区。激活的概念是指，fs1 分区在被 sync 之后，会指定 fs2 为激活分区；反之，fs2 会指定 fs1。

rmts page manager 对于 ram 是直接读写，ram 的簇号和 EFS 的 page 号是线性的映射，所以基本无需做垃圾收集等内存管理。


page_write
page_read
start_transaction
end_transaction
sync_start
sync_wait
EFS on NAND
EFS on RAM buffer
pm_page_write
pm_page_read
....
....
fs_rmts_pm_page_write
fs_rmts_pm_page_read
....
....


Name
ManyImagesTest
VoiceDB
brew
err
mmc1
mmgsdi
nvm
pdp_profiles
rom_vdb
user
$SYS.FACTORY
BT Params
BT_RM_DEV_DB
RDM_DEV_MAP
pbm_phone_uid.dat
reset_cntr.bin
i-node
i-node
i-node
Clusters
Linear mapping
cluster ≥ page
Pages in SDRAM buffer
superblock
Data region
superblock
Key (name) to i-node
Clusters to pages
Pages to state
log to/from RAM
Database
File manager
Page manager

rmts 接口 sync 的流程



fs_task 接收到 FS_EFS_SYNC_SIG 后，开始 efs sync 流程

1. 先将 modem ram 中的 efs 进行加密
2. 通过 rmts 接口调用 qmi 和 APP 进行通讯，modem 的 rmts 作为体系中的 client，APP 作为 server。
3. 将下一个分区指定为 active 分区
4. 等待下一次的 sync 信号

```
    fs_task.c  00158  EFS sync timer sig received
fs_rmts_pm.c  02030  47,66, EFS: Encrypt complete for sync [time(ms),age]
fs_rmts_pm.c  01329  0,917504,2,EFS: RMTS Write_iovec start[parti,bytes,sync_count
  rmts_api.c  00922  rmts: EFS Client write
  rmts_api.c  01038  rmts: write iovec server is up
  rmts_api.c  01047  rmts: write iovec partition found
  rmts_api.c  01072  rmts: write populating iovec
  rmts_api.c  01078  rmts: sector_addr= 0 buffer = -2039480320 num sec = 1792
  rmts_api.c  01097  QMI WRITE prmts_init_info SERVICE INFO[0] = 100
  rmts_api.c  01098  QMI WRITE prmts_init_info SERVICE INFO[1] = 1
  rmts_api.c  01099  QMI WRITE prmts_init_info SERVICE INFO[2] = 6
  rmts_api.c  01100  QMI WRITE prmts_init_info->service_object = c1f4e6f8
  rmts_api.c  01101  QMI WRITE pclient_data->write_client = 20
  rmts_api.c  01102  QMI WRITE pclient_data->write_os_params.sig = 1073741824
  rmts_api.c  01103  QMI WRITE pclient_data->write_os_params.timer_sig = 536870912
  rmts_api.c  01104  QMI WRITE pclient_data->write_os_params.tcb = -1015919532
  rmts_api.c  01105  QMI WRITE CLINET init rmts:rc = 0
  rmts_api.c  01120  rmts: start qmi
  rmts_api.c  01128  rmts:rc = 0 end qmi
  rmts_api.c  01153  rmtfs_resp->resp.result = QMI_RESULT_SUCCESS_V01
  rmts_api.c  01163  rmts_before_done
  rmts_api.c  01173  Write done
fs_rmts_pm.c  02059  [0,88], EFS: write_iovec for sync [parti,time (ms)]
 fs_public.c  03533  135,0, EFS: EFS-Sync complete (ms)
```

qmi_rmt_storage

qmi_rmt_storage 是 APP 端的一个 native service:

```
service rmt_storage /system/bin/rmt_storage
    class core
    user root
    disabled
on property:ro.boot.emmc=true
    start rmt_storage
```

启动后会执行一直循环等待 modem 端发来的事件，并进行处理

```
    while(1) {
        fds = os_params.fds;
        select(os_params.max_fd+1, &fds, NULL, NULL, NULL);
        os_params_in.fds = fds;
        qmi_csi_handle_event(svc.service_handle, &os_params_in);
    }
```

EFS 的 golden copy

所谓的 golden copy，将已经预设好的 NV 和系统配置制成一个 EFS 镜像，这个镜像可以帮助其他手机快速的导入 NV 和系统配置，而不需要再通过工具独自下载 QCN。这个镜像就是 golden copy。
高通提供了一套制作 golden copy 的方法。

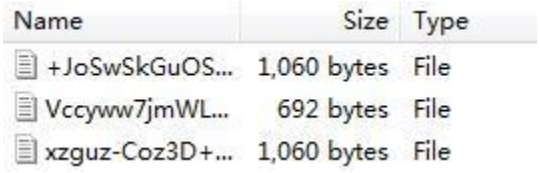
1. modem 宏设置：

(1). 为 efs 开后门

已知我们无法通过 AP 端去直接访问 modem ram 中解密后的 efs 数据，而使用工具就必须通过 AP 访问，所以普通的 modem 镜像无法获取解密后的 efs 数据。高通开了个后门：
#define FEATURE_EFS_ENABLE_FACTORY_IMAGE_SECURITY_HOLE

(2). 高通 Secure File System

高通在 modem 的代码中做了一些保护机制，使得被读出的 RF 相关 NV 在其他手机上无法正常工作，这套机制就是 SFS。
SFS 使用一种叫 MSM Device key 的特殊 key 来对一些关键存储文件来进行加密，这些被加密的文件在 efs 中查看以乱码形式存在。MSM_DEV_KEY 是从一个硬件寄存器中读出的一个 128 位唯一值，这个值在芯片封装时就已经决定，只能被 DSP 芯片读出。MSM_DEV_KEY 只在 Secure Boot 打开后才起作用。它的作用就是为每个芯片做标识，因此在 SFS 工作的时候，我们的 EFS 不能正常工作。
制作 EFS，必须先关闭 SFS，好在目前 SFS 对我们的用处并不大，所以直接把它注释掉了：
modem_proc/cust/inc/custuim.h 255
//#define FEATURE_MMGSDI_PERSO_SFS
此前显示乱码的文件实际上就是我们的 perso 文件。



以下两个目录，正常显示的是 SFS 关闭的状态，乱码是 SFS 打开的状态

设置宏之后，重新编译 nonhlos 镜像并烧入 modem 分区。

2. 擦除手机中 efs

已知 modemst1,modemst2,fsg 是用来存储 efs 数据的分区，在制作 golden copy 前需要先清除之前的数据，避免手机中已有的 NV 造成干扰。
擦除的方法就是直接用 fastboot 将这三个分区的数据写全 0 镜像。

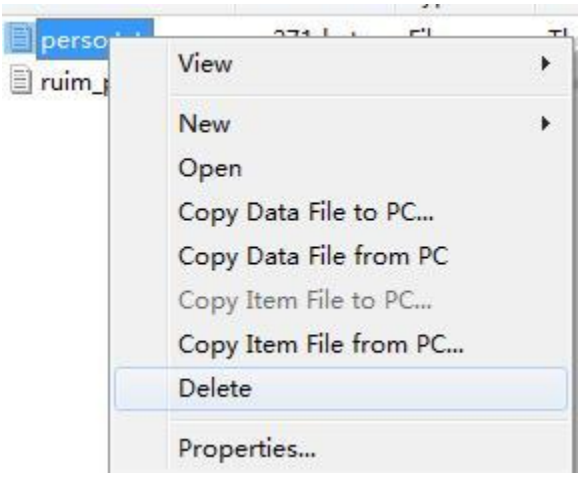
3. 下载 QCN

因为手机中的三个 NV 相关分区已经被清空，手机开机后，modem 做初始化，会通过 RMTS 自动生成一套全新的 EFS 文件系统。此时将需要制成 golden copy 的 QCN 通过工具写入，就可以在手机中生成一套带备份的 efs golden copy。

4. 导入 perso 文件

当前的手机已经实现了 simlock 功能，所以需要替换手机中原有的 perso 文件。
(1). 将手机连接 QPST EFS Explorer 工具
(2). 找到 perso 的 folder:

(3). 右键点击，将 perso 目录下的两个文件删除



(4). 将需要替换的连个 perso 文件，perso.txt 和 ruim_perso.txt 通过 new 导入到 efs。

5. 将 efs 读出来

高通提供了一个 perl 脚本来读取手机中的 efs 镜像，这个脚本路径：
modem_proc/core/storage/tools/ efsreadimage.pl
执行它需要以下环境：
（1）windows 下执行，脚本中需要调用 use Win32::OLE 的库
（2）安装 perl 环境 v5.10.1。这个高通没有明确说，但是目前这个版本是可用的。
（3）保持 QPST server 后台运行，可以是任意的调试工具，运行时可以看到以下图标：

这个脚本本质上是通过 diag 的命令通讯，通过 modem 端的 diag_task 将数据读出。具体的 diag 命令执行格式可以参考这个文档：
80-V1294-11_Y_FS_Subsys_ICD.pdf
执行命令，参数“-z”是将数据打包成 tar：
perl efsreadimage.pl -z
如果没有另外命名，生成的包：fs_image.tar.gz

6. 将 TAR 包签名

高通提供了一套给 TAR 包签名的工具，这套工具路径在：
modem_proc/core/storage/tools/qdst
高通默认提供了一套虚拟的证书，这套证书可以由 OEM 厂商自己进行修改添加比如 SW_ID 等信息，当然这是完全没必要的。
签名工具需要 OpenSSL 0.9.8m 以上版本，和 python 环境，版本没有具体说明，实际上可以编译 modem 镜像的 python 版本就可以，如 2.7.4
执行方法：
python QDSTMBN.py fs_image.tar.gz
签名成功会显示这个：

```
************************************************************
------------------------------------------------------------
        fs_image.tar.gz.mbn needs to be created
------------------------------------------------------------
************************************************************

QDSTMBN: Adding mbn_fs_image.tar.gz of size 17960 bytes.
QDSTMBN: Adding QDST_DefaultPart_SBL-signature.bin of size 256 bytes (0x4628).
QDSTMBN: Adding QDST_DefaultPart_SBL-attestation_cert.cer of size 1237 bytes (0x4728).
QDSTMBN: Adding QDST_DefaultPart_SBL-attestation_ca_cert.cer of size 1075 bytes (0x4BFD).
QDSTMBN: Adding QDST_DefaultPart_SBL-root_cert.cer of size 1099 bytes (0x5030).
QDSTMBN: Adding padding of size 2733 bytes (0x547B).

************************************************************
------------------------------------------------------------
QDSTMBN:        fs_image.tar.gz.mbn created
------------------------------------------------------------
************************************************************
NOTE: Used user supplied ZIP file 'cert/QDST_DefaultPart_SBL.zip'
```

生成 fs_image.tar.gz.mbn

7. 生成 EFS 镜像

要生成一个可以 flash 到其他设备的 EFS 还差最后一步，给签名后的 TAR 包加上 efs 头
头文件<MODEM BUILD>\build\ms\bin\<BUILD_ID>\ efs_image_meta.bin
脚本<MODEM BUILD>\modem_proc\core\bsp\efs_image_header\tools\
脚本需要python环境。
执行方法：
python efs_image_create.py [options] <efs_meta_file> < input_file>
最终会生成fs_image.tar.gz.mbn.img，这个镜像就是我们要的efs golden copy

8.烧写 EFS 其他手机

(1). 基本原理：modem启动后，会先检查FS1和FS2的分区数据是否有效，如果没有找到有效数据，modem会尝试将fsg分区中签名过后的TAR包解压并将数据导出并制成EFS，在适当的时机，分别写回FS1和FS2。
所以要我们的golden copy生效的前提就是让FS1和FS2在启动前变成无效数据。
如果一旦FS1或者FS2中存在有效数据，FSG的数据就没有用了。
(2). 我们现在的服务器脚本已经集成了自动下载efs镜像的功能：
rawprogram_unsparse_with_QCN.xml
在QFIL下载时使用这个脚本，会自动擦除fs1和fs2，并将fsg.img下载到fsg分区。正常情况下开机就能生成正确的EFS。

Reference

80-NF891-1_A_EFS_Prepopulate_Feature_B-Family_Chipsets
80-V1294-11_Y_FS_Subsys_ICD
SP80-NL239-5_D_MSM8916_SW_Debug_Manual_SPD
Code in : (modem_proc\core\storage\efs\src) and (core\storage\remotefs)