fastboot使用参考

fastboot主要原理是把点屏相关的必要模块和客户app放到ramdisk提前启动，达到快速亮屏的功能。

优点： 可以实现快速亮屏

缺点：制作ramdisk需要占用额外内存，app size越大，制作的ramdisk就越大，吃的内存越多。

所以在确定使用fastboot前，先要评估内存是否够用。

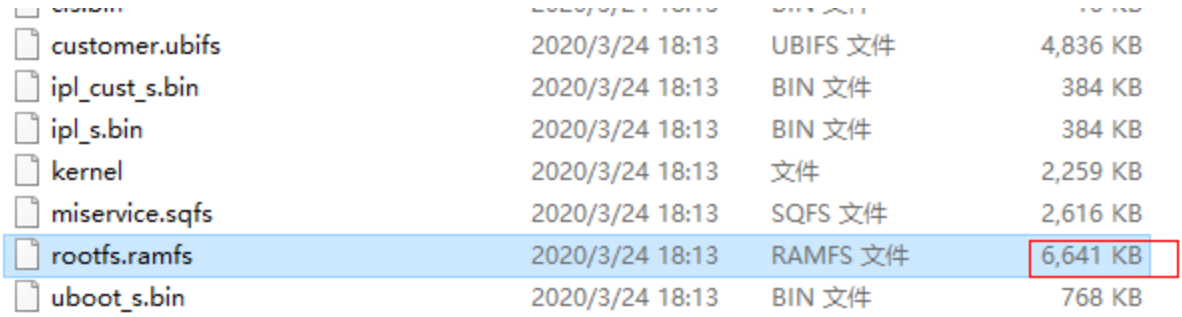
原则上客户app如果很大，不建议使用fastboot。

公版fastboot内存统计

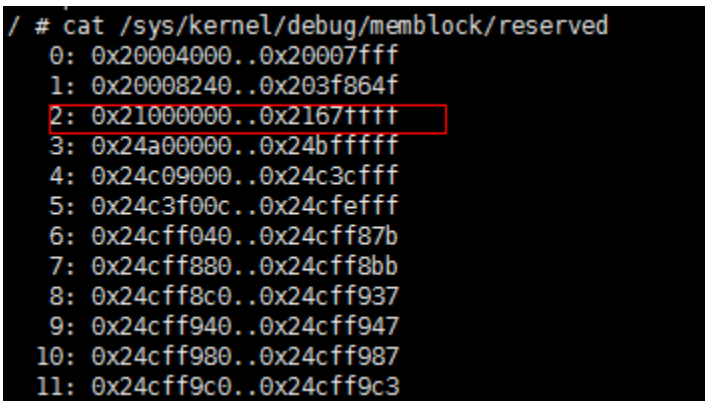
以zk_mini_fastboot为例统计公版fastboot内存使用情况

ramdisk占用内存：

ramdisk生成的size为6.49M



对应占用的内存差不多也是0x67ffff = 6.66M



所以linux可用内存 = MemTotal - Ramdisk Memory

注：DDR SIZE = MMA SIZE + MemTotal + kernel txt + MMAP IP

打开fastboot配置

目前公版ssd202 &ssd201 nand和nor flash全部共用一套app来demo，app路径如下：

sdk\verify\application\zk_mini_fastboot

源码路径：

https://github.com/aaron201912/SSD_PLAYER/tree/master/zk_mini_fastboot

公版默认release的config已经把这个app打包到image：

project\release\customer_tailor\nvr_i2m_display_fastboot_glibc_tailor.mk

```
interface_shadow:=disable
interface_uac:=disable
interface_vdf:=disable
interface_vdisp:=disable
#interface_vdec:=disable
interface_venc:=disable
interface_wlan:=enable

misc_fbdev:=enable
#verify_jpeg2disp:=enable
verify_zk_mini_fastboot:=enable
#mhal
#mhal_aio:=disable
mhal_csi:=disable
#mhal_disp:=disable
#mhal_divp:=disable
```

fastboot的编译config只有kernel和project需要单独配置，boot跟正常启动的config一样。具体如下：

Nand flash

project:

configs/nvr/i2m/8.2.1/spinand.ram-glibc-squashfs.011a.128 **//ssd202**

configs/nvr/i2m/8.2.1/spinand.ram-glibc-squashfs.011a.64 **//ssd201**

kernel:

infinity2m_spinand_ssc011a_s01a_minigui_fastboot_defconfig

Nor flash

project:

configs/nvr/i2m/8.2.1/nor.glibc-ramfs.011a.64 **//ssd201**

configs/nvr/i2m/8.2.1/nor.glibc-ramfs.011a.128 **//ssd202**

kernel:

infinity2m_ssc011a_s01a_fastboot_defconfig

注意：编译完kernel，需要先把kernel对应的lib release到project下面，再编译project，release步骤如下：

首先进入到project/kbuild/4.9.84 目录

nand flash

```
./release.sh -k ${RELEASEDIR}/kernel -b 011A-fastboot -p nvr -f spinand -c i2m -l glibc -v 8.2.1
```

nor flash

```
./release.sh -k ${RELEASEDIR}/kernel -b 011A-fastboot -p nvr -f nor -c i2m -l glibc -v 8.2.1
```

fastboot主要实现流程

fastboot实现的方式主要体现在project\image\configs\i2m\rootfs_fastboot.mk这个makefile里面，这个makefile把开机的行为全部放到/etc/profile下面，大概流程如下：

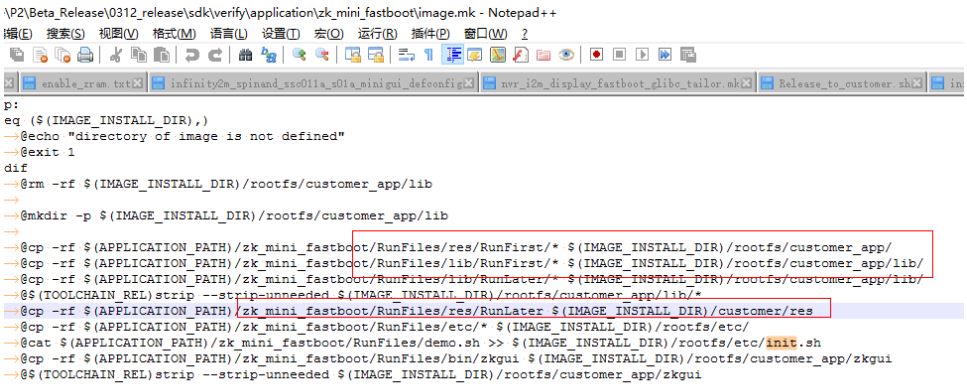
1. 将开机必须用到的模块放到/etc/init.sh

```
Line 67:    chmod 755 $(LATE_INIT_FILE)
Line 97:            sed 's#\(.*\).ko#insmod /lib/modules/$(KERNEL_VERSION)/\1.ko#' > $(INIT_FILE); \
Line 101:       echo -e "# kernel_mod_list\n" >> $(INIT_FILE); \
Line 109:           sed 's#\(.*\).ko#insmod /config/modules/$(KERNEL_VERSION)/\1.ko#' >> $(LATE_INIT_FILE); \
Line 113:       echo -e "# kernel_mod_list\n" >> $(LATE_INIT_FILE); \
Line 120:           sed 's#\(.*\).ko#insmod /lib/modules/$(KERNEL_VERSION)/\1.ko#' >> $(INIT_FILE); \
Line 124:       echo -e "# misc_mod_list\n" >> $(INIT_FILE); \
Line 130:           sed '2,20s#\(.*\)#insmod /lib/modules/$(KERNEL_VERSION)/\1.ko\nif [ $$? -eq 0 ]; then\n  busybox mknod
Line 134:       echo "#mi module" >> $(INIT_FILE); \
Line 139:           sed 's#\(.*\).ko#insmod /lib/modules/$(KERNEL_VERSION)/\1.ko#' >> $(INIT_FILE); \
Line 143:       echo -e "# misc_mod_list_late\n" >> $(INIT_FILE); \
Line 155:   sed -i 's/mi_common/insmod \/lib\/modules\/$(KERNEL_VERSION)\/mi_common.ko\nmajor=\`cat \/proc\/devices \| busy
Line 156:   echo "#mi module" >> $(INIT_FILE);
Line 157:   sed -i '/#mi module/a   major=`cat /proc/devices | busybox awk "\\\\$$2==\\""mi_poll"\\" {print \\\\$$1}"`\nbus
Line 174:   echo mdev -s >> $(INIT_FILE);
Line 177:       sed -i 's/mi_sys.ko/mi_sys.ko cmdQBufSize=128 logBufSize=0 default_config_path=\/sstar_configs/g' $(INIT_FI
Line 178:       sed -i 's/fbdev.ko/fbdev.ko default config path file=\/sstar configs\/fbdev.ini/g' $(INIT FILE); \
```

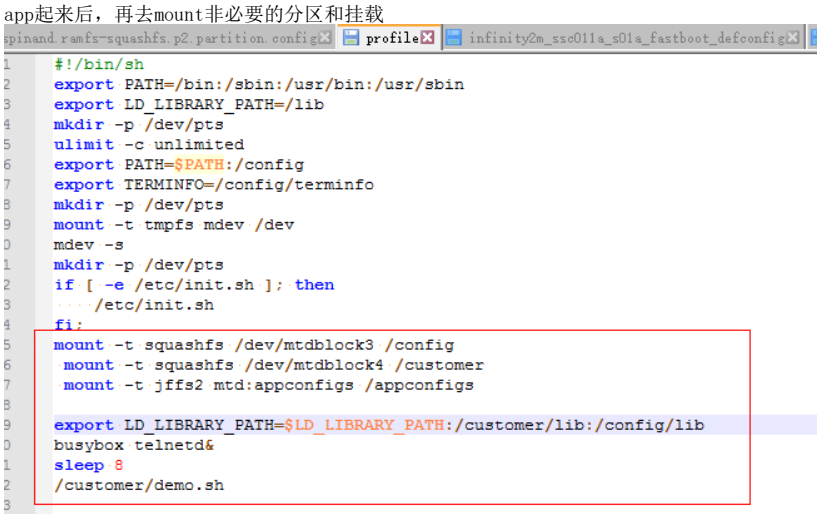
2. 然后在init.sh最后启动app，sdk\verify\application\zk_mini_fastboot\image.mk

```
endif
→@rm -rf $(IMAGE_INSTALL_DIR)/rootfs/customer_app/lib
→
→@mkdir -p $(IMAGE_INSTALL_DIR)/rootfs/customer_app/lib
→
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/res/RunFirst/* $(IMAGE_INSTALL_DIR)/rootfs/customer_app/
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/lib/RunFirst/* $(IMAGE_INSTALL_DIR)/rootfs/customer_app/lib/
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/lib/RunLater/* $(IMAGE_INSTALL_DIR)/rootfs/customer_app/lib/
→@$(TOOLCHAIN_REL)strip --strip-unneeded $(IMAGE_INSTALL_DIR)/rootfs/customer_app/lib/*
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/res/RunLater $(IMAGE_INSTALL_DIR)/customer/res
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/etc/* $(IMAGE_INSTALL_DIR)/rootfs/etc/
→@cat $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/demo.sh >> $(IMAGE_INSTALL_DIR)/rootfs/etc/init.sh
→@cp -rf $(APPLICATION_PATH)/zk_mini_fastboot/RunFiles/bin/zkgui $(IMAGE_INSTALL_DIR)/rootfs/customer_app/zkgui
→@$(TOOLCHAIN_REL)strip --strip-unneeded $(IMAGE_INSTALL_DIR)/rootfs/customer_app/zkgui
```

注：app启动显示第一张画面必须的res放在rootfs的customer_app中，其他的res放在customer下面：
这样做的好处是可以减少rootfs的大小，加快启动速度



3. app起来后，再去mount非必要的分区和挂载



fastboot客制化需要注意的地方

客户在导入fastboot时，主要需要考虑以下几个地方：

- 客户的app大小跟公版不同
- 客户的app内存使用跟公版不同
- 客户的app行为跟公版不同

客户app大小跟公版不同

app的大小对fastboot至关重要，因为app越大，制作出来的ramdisk越大，rootfs的分区越大，使用的内存也越大，同时解压ramdisk花的时间也就越长。**所以如果客户的app很大，fastboot并不能发挥它的优势**。

目前app是放在rootfs下面，所以如果客户的app大小跟公版不同，可以通过修改如下分区文件来更改rootfs分区的大小：

nand：

project\image\configs\i2m\spinand.ramfs-squashfs.p2.partition.config

```
rootfs$(RESOUCE)    = $(OUTPUTDIR)/rootfs
rootfs$(FSTYPE)     = ramfs
rootfs$(PATSIZE)    = 0x00680000
rootfs$(BOOTENV)    = console=ttyS0,115200 ubi.mtd=UBI,2048 rootfstype=ramfs initrd=$(INITRAMFSLOADADDR),$(rootfs$(PATSIZE)) loglevel=0
rootfs$(BOOTCMD)    = nand read.e $(INITRAMFSLOADADDR) rootfs $(rootfs$(PATSIZE))\;
rootfs$(MTDPART)    = $(rootfs$(PATSIZE))(rootfs)
```

nor：

project\image\configs\i2m\nor.ramfs.partition.config

```
kernel$(RESOUCE)    = $(PROJ_ROOT)/release/$(PRODUCT)/$(CHIP)/$(BOARD)/$(TOOLCHAIN)/$(TOOLCHAIN_VERSION)/bin/kerr
kernel$(PATSIZE)    = 0x00210000
kernel$(BOOTENV)    = $(KERNEL_BOOT_ENV) loglevel=0
kernel$(BOOTCMD)    = sf read $(KERNELBOOTADDR) \$${sf_kernel_start} \$${sf_kernel_size}\;

rootfs$(RESOUCE)    = $(OUTPUTDIR)/rootfs
rootfs$(FSTYPE)     = ramfs
rootfs$(PATSIZE)    = 0x00680000
rootfs$(BOOTENV)    = rootfstype=ramfs initrd=$(INITRAMFSLOADADDR),$(rootfs$(PATSIZE))
rootfs$(BOOTCMD)    = mxp r.info rootfs\; sf read $(INITRAMFSLOADADDR) \$${sf_part_start} \$${sf_part_size}\;

miservice$(RESOUCE)    = $(OUTPUTDIR)/miservice/config
miservice$(FSTYPE)     = squashfs
miservice$(PATSIZE)    = 0x00300000
miservice$(MOUNTTG)    = /config
miservice$(MOUNTPT)    = /dev/mtdblock3
```

注意分区总大小不能超过flash的总大小，所以rootfs增大后，需要从其他分区相应的减少；同时分区大小必须64kb对齐

客户app内存使用跟公版不同

目前我们DDR的内存主要由mma和linux memory组成：
DDR SIZE ~= MMA SIZE + linux MemTotal

所以客户可以根据实际mma的使用情况，增大或减少mma的大小来调整内存使用，实际mma的使用可以在最大使用场景通过如下命令dump出来：

```
/ # cat /proc/mi_modules/mi_sys_mma/mma_heap_name0
          mma heap name       heap_base_cpu_bus_addr                  length            chunk_mgr_avail
         mma_heap_name0                    2299a000                 1566000                    a3c000
chunk_mgr info:
          offset            length             avail
               0           1566000            a3c000

each chunk info:
          offset            length         used_flag          task_name
               0             20000                 1             CMDMEM
           20000              1000                 1      sys-logConfig
           21000            400000                 1          fb_device
          421000            177000                 1                 gui
          598000              4000                 1       gui_readbmp1
          59c000              4000                 1       gui_readbmp1
          5a0000            258000                 1       gui_readbmp1
          7f8000             10000                 1       gui_readbmp1
          808000             10000                 1       gui_readbmp1
          818000             10000                 1       gui_readbmp1
          828000             10000                 1       gui_readbmp1
          838000             10000                 1       gui_readbmp1
          848000             10000                 1       gui_readbmp1
          858000             10000                 1       gui_readbmp1
```

客户app行为跟公版不同

fastboot跟正常启动最大的区别在于fastboot有些非必要的模块是延迟启动的，所以画面刚出来不能马上使用usb或者wifi等没有启动的模块，需要在app的ui做等待处理。

fastboot打开bootlogo

目前bootlogo在uboot做的，所以开bootlogo必须要跑uboot，具体改动如下：

1. setenv ota_upgrade_status 1 ，通过ota_upgrade_status控制要不要跑uboot；
2. setenv autoestart 0，关闭uboot网络功能，加快开机时间；
3. boot\drivers\mstar\spinand\inc\config\infinity2m\drvSPINAND_uboot.h中打开quadmode read，提升load img速度
   #define SUPPORT_SPINAND_QUAD (1)