Table of Contents

Introduction

本文是运用gitbook工具来管理的笔记本，将平时积累的经验记录在相应的md文件中，执行gitbook工具命令，便能将所有md文件按照指定的方式形成一个完整的pdf文件。

> gitbook init —— 在当前目录初始化笔记，如果是在已经存在的笔记上执行此命令，则会根据最新的内容生成所需的目录及文件
> gitbook pdf ./ ./LearnNoteBook.pdf —— 将当前目录的笔记生成pdf文件
> gitbook build 书籍路径 输出路径 —— 将当前目录的笔记生成html版

Android

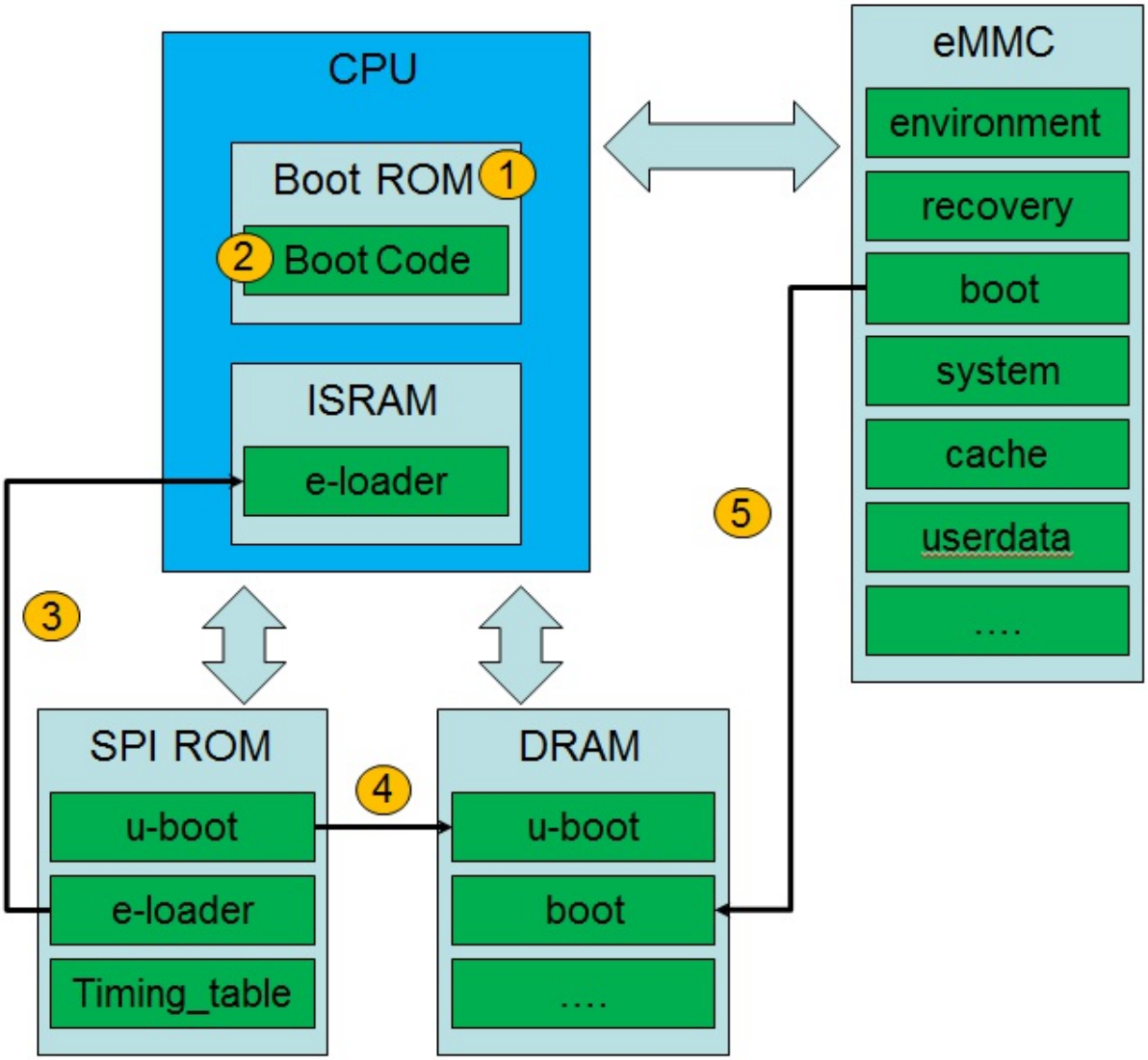
App

Framework

Android启动流程分析

1. 内核启动流程分析

> 1、设备上电后，从Boot ROM开始运行；
>
> 2、Boot ROM中的代码初始化通信端口和可引导存储设备（SD/ eMMC/NAND）；
>
> 3、Boot Code将SPI ROM中的e-loader加载到内部ISRAM，并跳转到e-loader入口执行；
>
> 4、e-loader在初始化完DRAM后，将SPI ROM中的u-boot加载到DRAM，并跳转到u-boot入口执行；
>
> 5、u-boot代码初始化eMMC后，将eMMC中的boot加载到DRAM，并启动内核；

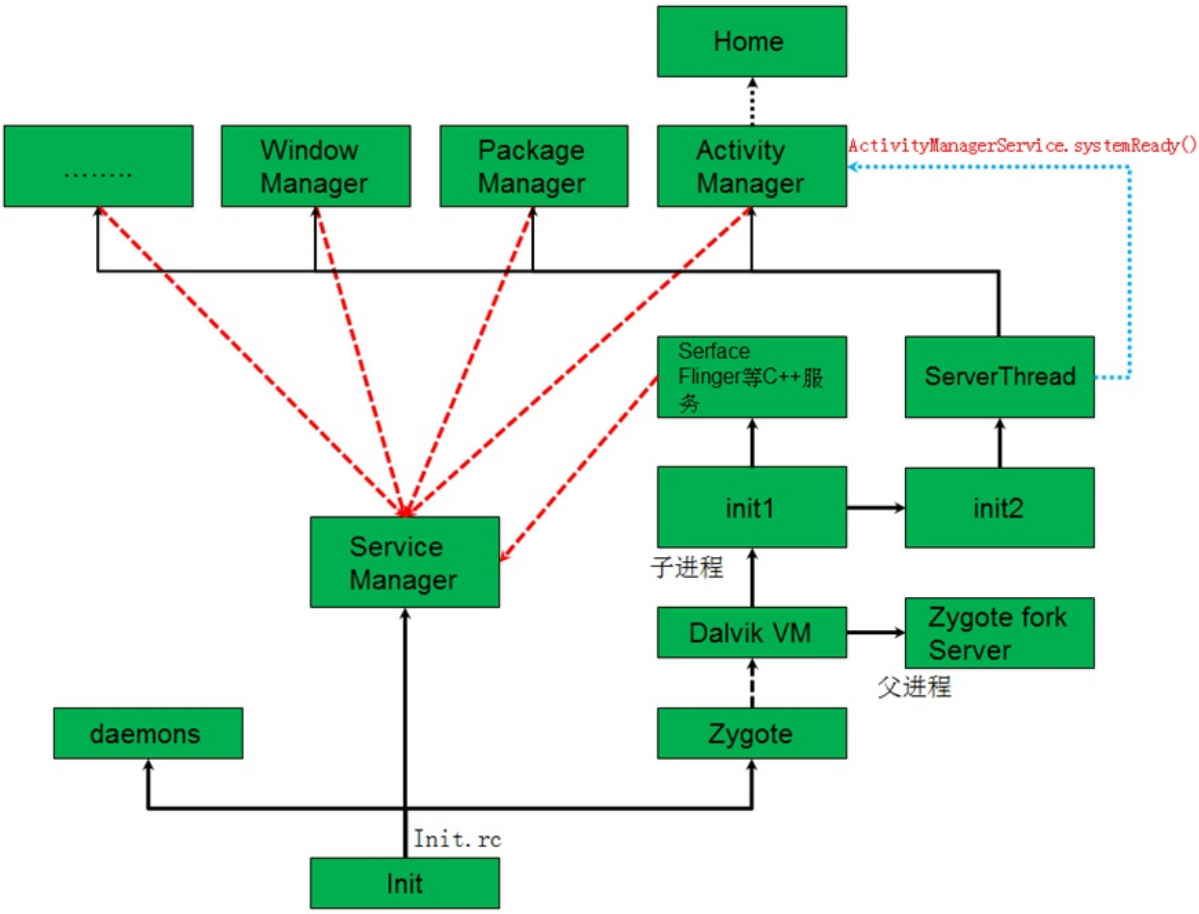


2. Android系统启动流程

> 1、内核在执行Init进程时，会分别启动守护进程、服务器管理进程和Zygote进程；

2、Zygote进程会初始化并实例化Dalvik虚拟机；接着执行ZygoteInit.java代码，打开监听，并在主进程中无线循环接收命令创建新进程；ZygoteInit还会创建子进程以执行SystemServer.java,从而启动C++服务及Java实现的管理服务，并注册到服务管理器中；

3、服务启动完后，ServerThread会给ActivityManagerService发送系统就绪通知，当ActivityManagerService收到系统就绪通知后，会启动Home应用程序；



3. Android系统启动时应用程序安装过程

Android系统在启动的过程中，Zygote进程会启动一个应用程序管理服务PackageManagerService，这个服务负责扫描系统中特定的目录，找到里面的应用程序文件，即以Apk为后缀的文件，然后对这些文件进解析，得到应用程序的相关信息，最终完成应用程序的安装过程。

1、Zygote进程在创建子进程后，执行SystemServer.java的main方法；

2、在main方法中调用init1方法，该方法通过JNI实现；

3、init1方法中会调用libsystem_server库中的system_init函数，从而初始化surfaceFlinger、sensorService、audioFlinger、mediaPlayerService、cameraService和audioPolicyService这些C++实现的服务；

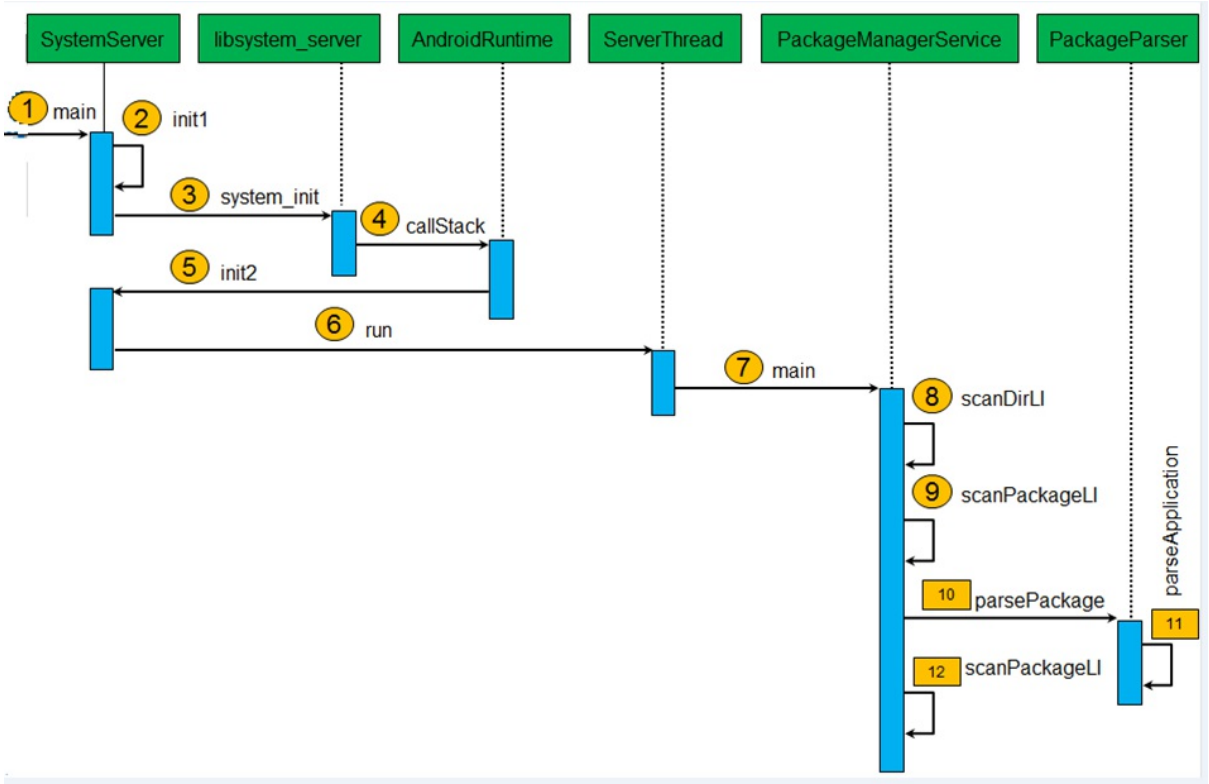
4、在system_init中，还会通过全局唯一的AndroidRuntime实例变量runtime来调用callStack；

5、在callStack中再调用SystemServer的init2方法；

6、在init2中创建ServerThread线程，并启动该线程，从而执行run方法；

> 7、在ServerThread线程中，会启动一系列Java实现的管理服务，其中就包括PackageManagerService，通过执行PackageManagerService的main方法，从而创建对应服务的实例，并将该服务添加到服务管理器中；
>
> 8、在实例化PackageManagerService时，其构造方法中会执行安装应用程序的过程，首先通过scanDirLI方法来扫面系统目录下的APK安装文件，扫描目录包括：/system/framework、/system/app、/vendor/app、/data/app和/data/app-private；
>
> 9、在scanDirLI中，对目录中的每一个APK文件，都会调用scanPackageLI对其进行解析和安装；
>
> 10、在scanPackageLI中，会对应APK创建PackageParser实例，并调用实例的parsePackage先得到APK文件中的AndroidManifest.xml文件，再调用另一个版本的parsePackage对AndroidManifest.xml文件进行解析；
>
> 11、在另一个版本的parsePackage中，会通过调用parseApplication对application标签进行解析；
>
> 12、在执行完10和11步骤后，继续回到步骤9中，并在scanPackageLI中调用另一个版本的scanPackageLI，把解析得到的package、provider、service、receiver和activity等信息保存在PackageManagerService服务中；

当以上这些流程运行完成后，Android系统启动时安装应用程序过程已近完成，此时，这些应用程序只相当于在PackageManagerService中注册完成，但还需要Home应用程序将PackageManagerService中注册好的应用程序取出来，并呈现在桌面上。



4. Android应用程序启动过程

> 1、无论是launcher启动Activity，还是通过Activity内部调用启动新的Activity，都需要通过Binder通信到ActivityManagerService进程中调用ActivityManagerService.startActivity接口；

2、ActivityManagerService调用ActivityStack.startActivityMayWait来做准备要启动的Activity相关信息；

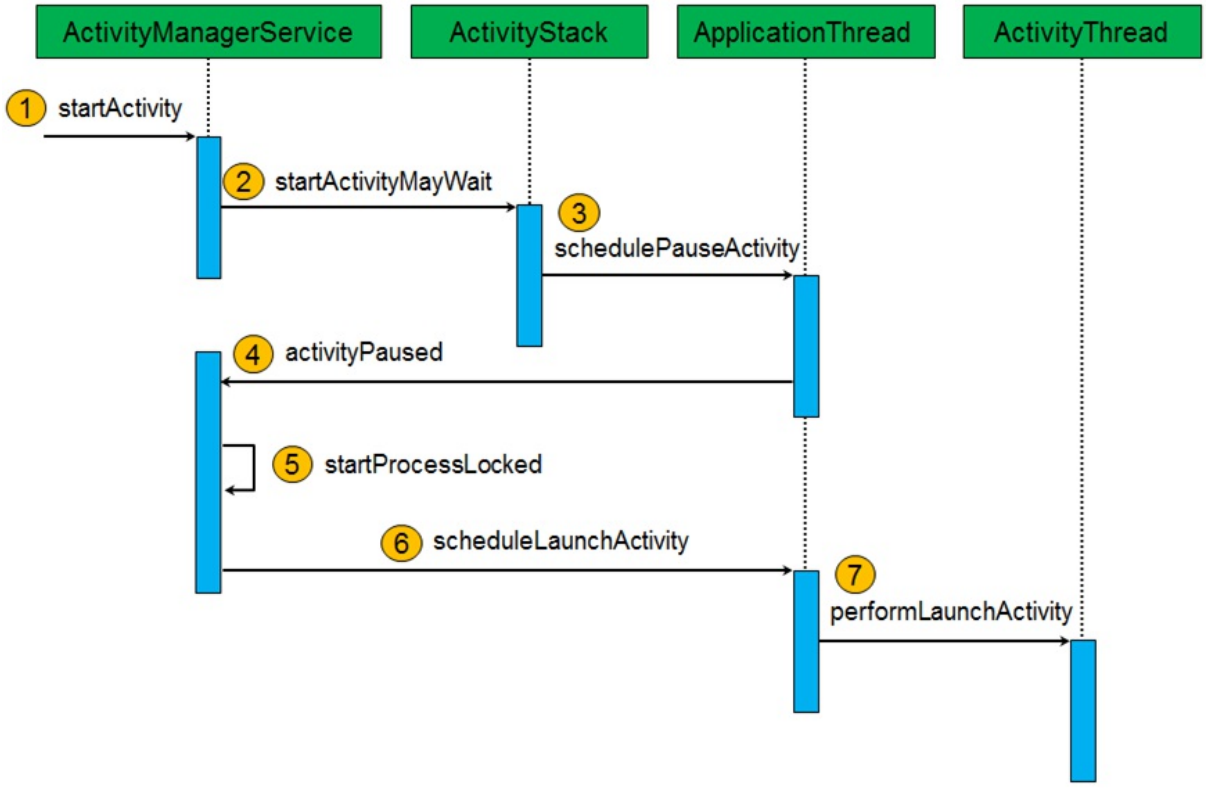
3、ActivityStack通知ApplicationThread进程要进行Activity调度了。ApplicationThread进程表示launcher进程或原Activity所在进程；

4、ApplicationThread进程不执行真正的启动操作，通过Binder通信调用ActivityManagerService.activityPaused接口进入到ActivityManagerService进程中，看是否需要创建新的进程；

5、通过点击应用图标方式启动Activity，则需要在ActivityManagerService中调用startProcessLocked创建新的进程；若是在原Activity中启动新的Activity，则不需要再创建新的进程；

6、ActivityManagerService调用ApplicationThread.scheduleLaunchActivity接口，通知相应进程执行启动Activity的操作；

7、ApplicationThread把启动Activity的操作转发给ActivityThread，ActivityThread通过ClassLoader导入相应Activity类，然后把它启动起来；



5. Launcher启动过程

Android系统的Home应用程序Launcher是由ActivityManagerService服务启动的，该服务和PackageManagerService服务一样，都是在开机时由SystemServer组件启动的

1~7跟前文《Android系统启动时应用程序安装过程》的1~7步骤一样；

9、在执行setSystemProcess时，首先将ActivityManagerService服务注册到服务管理器中，然后调用installSystemApplication将应用程序框架层下面的Android包加载进来；

10、在ServerThread将一系列服务初始化完成后，会通过调用systemReady告诉ActivityManagerService系统已经就绪；

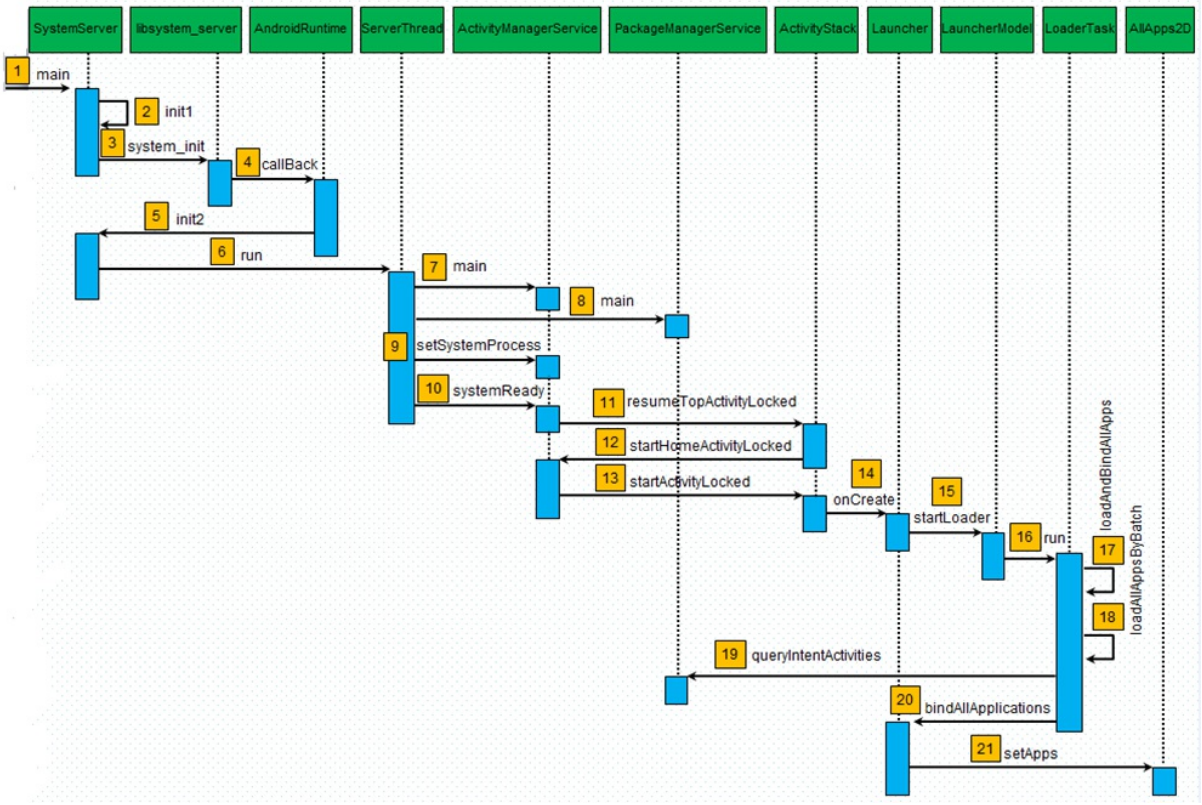
11、在systemReady中，会调用resumeTopActivityLocked来启动Home应用程序；

12、在resumeTopActivityLocked中，由于返回顶端的Activity为null，因此会调用startHomeActivityLocked，该方法会创建CATEGORY_HOME类型的Intent，然后通过Intent的resolveActivityInfo在PackageManagerService中查询同类型的Activity；

13、在startHomeActivityLocked中，因第一次启动该Activity，故会调用startActivityLocked来启动Launcher的Activity；

14、在执行startActivityLocked时，接着会调用Launcher Activity的onCreate方法；

15~21将已安装的应用加载进来，并在桌面上显示这些应用；



Android uboot传递cmdline到内核原理

1. boot.img结构

boot.img由boot header参数、kernel、ramdisk和second stage组成。boot header占用1 page，其中包含了内核加载地址、ramdisk加载地址及cmdline等参数。

```
** +-----------------+
** | boot header     | 1 page
** +-----------------+
** | kernel          | n pages
** +-----------------+
** | ramdisk         | m pages
** +-----------------+
** | second stage    | o pages
** +-----------------+
**
** n = (kernel_size + page_size - 1) / page_size
** m = (ramdisk_size + page_size - 1) / page_size
** o = (second_size + page_size - 1) / page_size
**
```

2. boot.img中的cmdline来源

boot.img中的cmdline参数是由android代码device下的BoardConfig.mk文件中变量BOARD_KERNEL_CMDLINE定义。在编译阶段，由build/core/Makefile将BOARD_KERNEL_CMDLINE变量读取出来，并传递给system/core/mkbootimg.c代码处理，在生成boot.img时写入到boot header相应的字段。

3. cmdline参数传递到内核

u-boot在启动boot.img时，执行booti boot命令，此时u-boot会跳转到do_booti函数中，将boot分区的boot.img信息读取出来，在读取cmdline后，还会将其他参数如androidboot.bootloader等参数一起追加在cmdline中，最后再将新的cmdline值写入bootargs中。启动内核时，将新的bootargs值传递给内核。

Linux

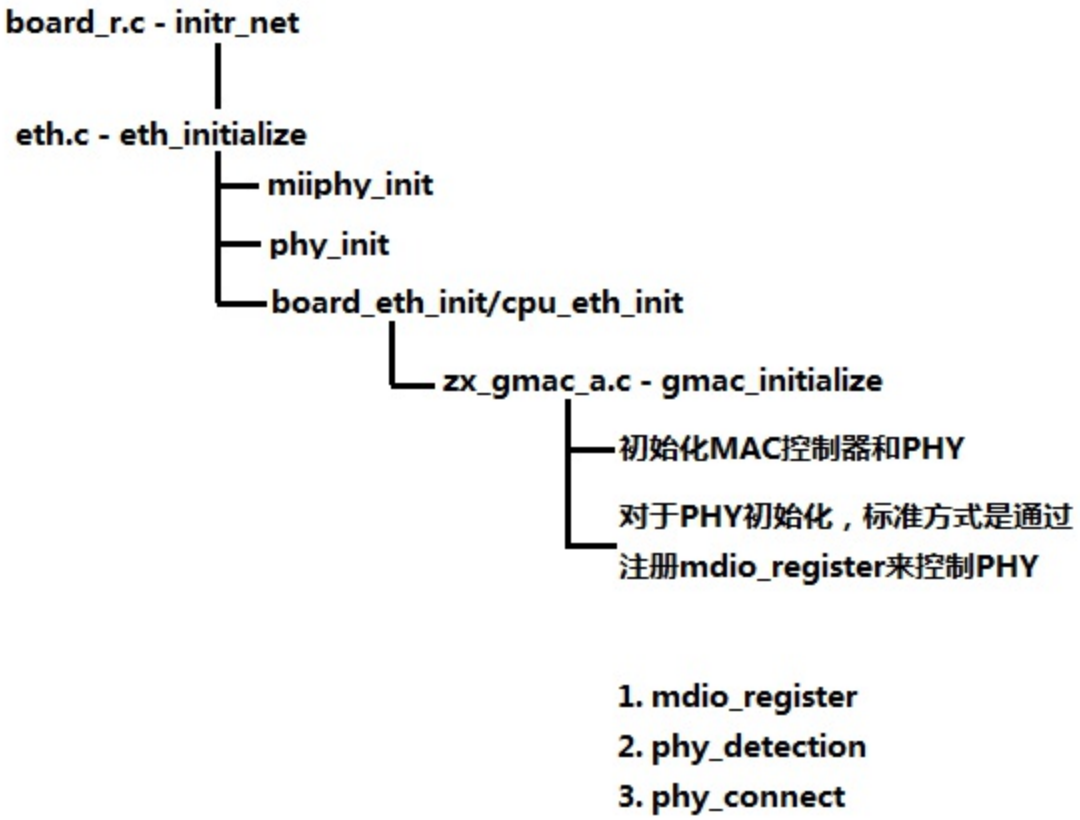
- uboot
- kernel
- Yocto

uboot

本章节主要介绍uboot相关的知识。

- uboot初始化网卡流程
- imx6 DDR配置过程

uboot初始化网卡流程



imx6 DDR配置过程

1. DDR配置选择

> uboot目录/configs/mx6qvab820_defconfig文件中，通过CONFIG_SYS_EXTRA_OPTIONS来包含配置文件vab820.cfg，vab820.cfg中会根据宏CONFIG_USE_PLUGIN来决定是使用plugin.S还是该文件中自身包含的配置内容

2. plugin.S作用

> DDR的初始化动作是在plugin.S中处理，plugin.S会根据SOC类型及DDR内存大小来include不同的DDR配置文件，配置文件路径：/board/via/vab820/中

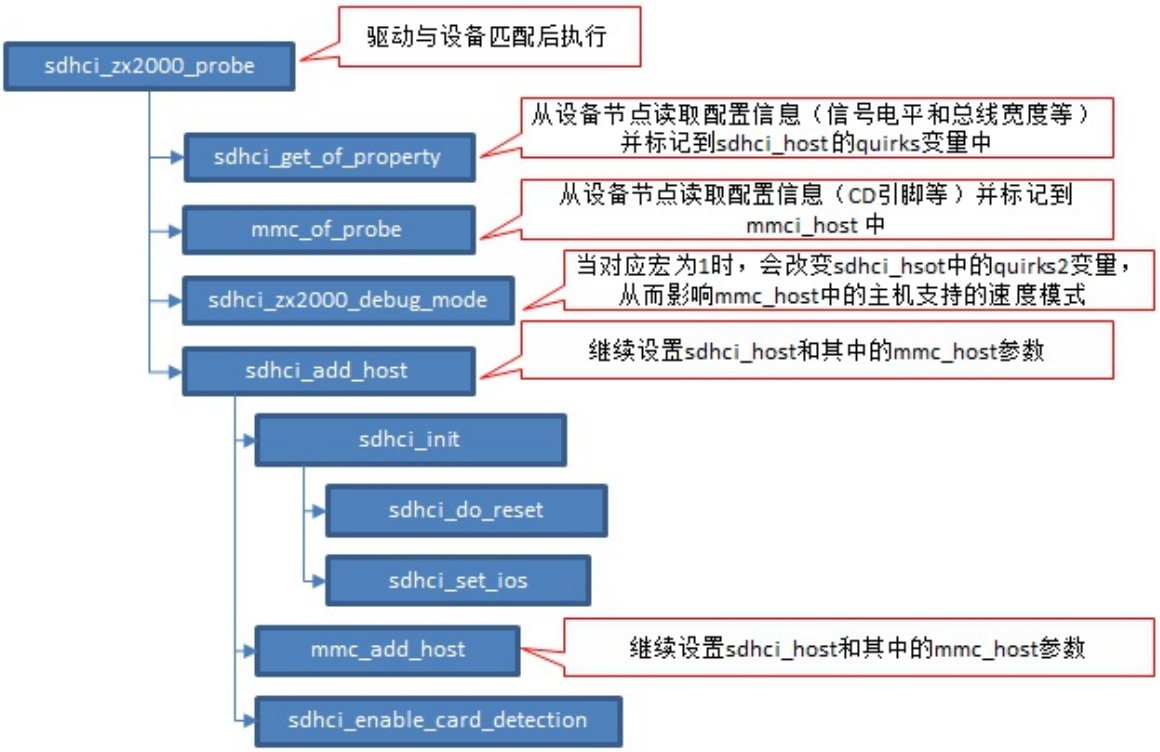
3. DDR配置文件的来源

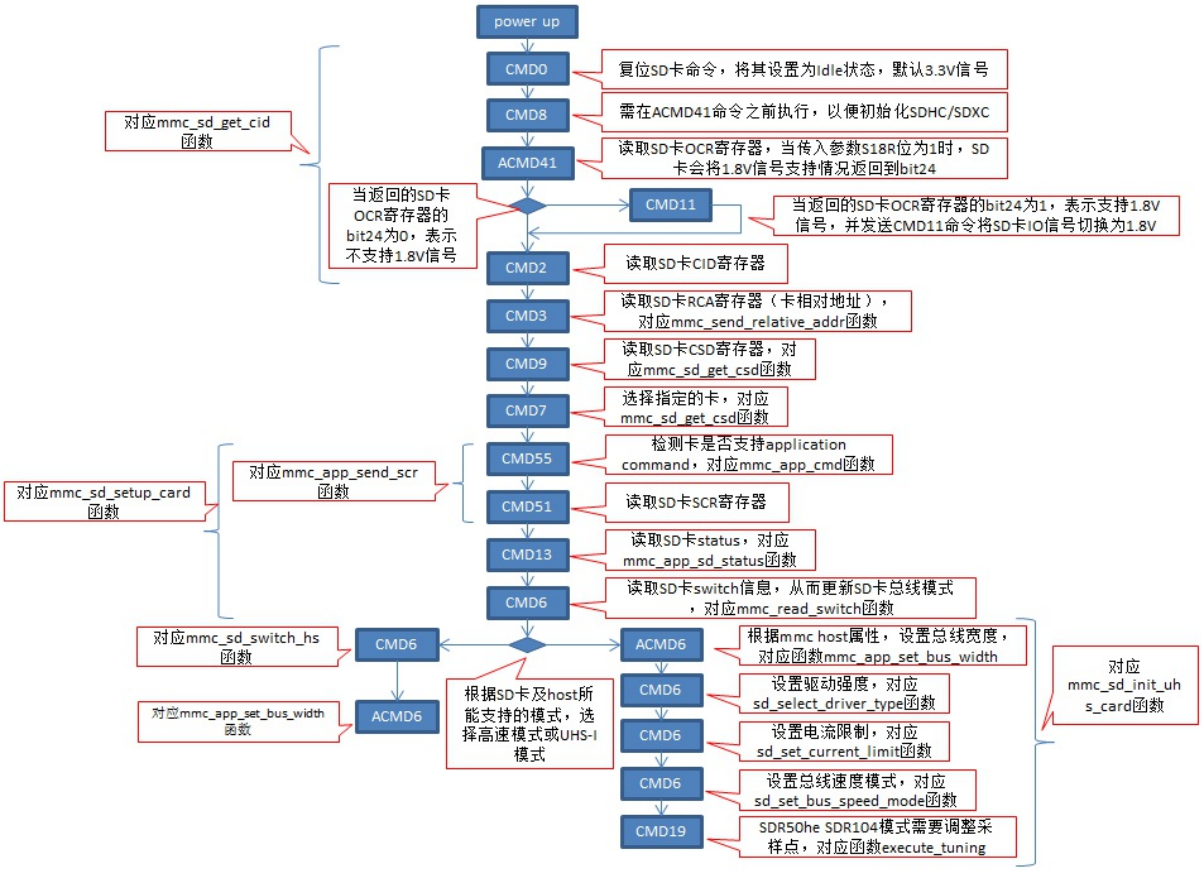
> 配置文件中来源于《IMX6DQSDL DDR3 Script Aid V0.11》文档生成的RealView.inc文件的内容
>
> 注：调试过程中，如果DDR配置不正确，uboot有时无法运行，或kernel随机panic，或Android起来后，USB鼠标无法正常使用。

kernel

SDHC流程分析


mmc_rescan
mmc_rescan_try_freq
mmc_power_up
__mmc_set_signal_voltage
设置SD host总线信号电压（默认为3.3V）
mmc_hw_reset_for_init
mmc_set_ios
ops->start_signal_voltage_switch
sdio_reset
sdhc_start_signal_voltage_switch
mmc_go_idle
sdhc_do_start_signal_voltage_switch
先对host及card进行信号电压匹配，再读取cid值
mmc_send_if_cond
regulator_set_voltage
mmc_attach_sd
mmc_sd_get_cid
signal_pad_switch
兆芯驱动中的函数，设置总线信号电平
mmc_sd_init_card
mmc_send_relative_addr
sdhc_zx2000_signal_pad_switch
mmc_add_card
mmc_sd_get_csd
mmc_select_card
读取卡信息后得到最大传输速度及支持的总线模式等
mmc_sd_setup_card
mmc_sd_switch_hs
mmc_sd_init_uhs_card
根据host及card能力设置总线宽度
mmc_app_set_bus_width
根据ocr中bit24或host当前总线信号类型决定使用哪种模式
sd_update_bus_speed
根据读取到所支持的总线模式等信息更新总线速度
sd_select_dricer_type
sd_set_current_limit
sd_set_bus_speed_mode
设置总线速度
excute_tuning


驱动与设备匹配后执行
sdhci_zx2000_probe
从设备节点读取配置信息（信号电平和总线宽度等）并标记到sdhci_host的quirks变量中
sdhci_get_of_property
从设备节点读取配置信息（CD引脚等）并标记到mmci_host 中
mmc_of_probe
当对应宏为1时，会改变sdhci_hsot中的quirks2变量，从而影响mmc_host中的主机支持的速度模式
sdhci_zx2000_debug_mode
继续设置sdhci_host和其中的mmc_host参数
sdhci_add_host
sdhci_init
sdhci_do_reset
sdhci_set_ios
mmc_add_host
继续设置sdhci_host和其中的mmc_host参数
sdhci_enable_card_detection


power up
CMD0
复位SD卡命令，将其设置为Idle状态，默认3.3V信号
CMD8
需在ACMD41命令之前执行，以便初始化SDHC/SDXC
ACMD41
读取SD卡OCR寄存器，当传入参数S18R位为1时，SD卡会将1.8V信号支持情况返回到bit24
对应mmc_sd_get_cid函数
当返回的SD卡OCR寄存器的bit24为0，表示不支持1.8V信号
CMD11
当返回的SD卡OCR寄存器的bit24为1，表示支持1.8V信号，并发送CMD11命令将SD卡IO信号切换为1.8V
CMD2
读取SD卡CID寄存器
CMD3
读取SD卡RCA寄存器（卡相对地址），对应mmc_send_relative_addr函数
CMD9
读取SD卡CSD寄存器，对应mmc_sd_get_csd函数
CMD7
选择指定的卡，对应mmc_sd_get_csd函数
CMD55
检测卡是否支持application command，对应mmc_app_cmd函数
对应mmc_app_send_scr函数
CMD51
读取SD卡SCR寄存器
对应mmc_sd_setup_card函数
CMD13
读取SD卡status，对应mmc_app_sd_status函数
CMD6
读取SD卡switch信息，从而更新SD卡总线模式，对应mmc_read_switch函数
对应mmc_sd_switch_hs函数
CMD6
根据SD卡及host所能支持的模式，选择高速模式或UHS-I模式
ACMD6
对应mmc_app_set_bus_width函数
ACMD6
根据mmc host属性，设置总线宽度，对应函数mmc_app_set_bus_width
CMD6
设置驱动强度，对应sd_select_driver_type函数
CMD6
设置电流限制，对应sd_set_current_limit函数
CMD6
设置总线速度模式，对应sd_set_bus_speed_mode函数
CMD19
SDR50he SDR104模式需要调整采样点，对应函数execute_tuning
对应mmc_sd_init_uhs_card函数

键盘LED灯消息处理流程

```
[<c0012530>] (show_stack) from [<c0abd8e8>] (dump_stack+0x7c/0xbc)
[<c0abd8e8>] (dump_stack) from [<c069d014>] (input_handle_event+0x444/0x560)
[<c069d014>] (input_handle_event) from [<c069d20c>] (input_inject_event+0x7c/0x80)
[<c069d20c>] (input_inject_event) from [<c044c0ac>] (kbd_update_leds_helper+0x54/0x70)
[<c044c0ac>] (kbd_update_leds_helper) from [<c069b1d4>] (input_handler_for_each_handle+0x2c/0x60)
[<c069b1d4>] (input_handler_for_each_handle) from [<c044c7bc>] (kbd_bh+0x78/0x8c)
[<c044c7bc>] (kbd_bh) from [<c0040014>] (tasklet_action+0x68/0xf4)
[<c0040014>] (tasklet_action) from [<c003f56c>] (__do_softirq+0xd4/0x34c)
[<c003f56c>] (__do_softirq) from [<c003faa4>] (irq_exit+0xb8/0xf4)
[<c003faa4>] (irq_exit) from [<c000ef10>] (handle_IRQ+0x44/0x90)
[<c000ef10>] (handle_IRQ) from [<c000856c>] (gic_handle_irq+0x2c/0x5c)
[<c000856c>] (gic_handle_irq) from [<c00131fc>] (__irq_usr+0x3c/0x60)
Exception stack(0xcfa01fb0 to 0xcfa01ff8)
1fa0:                                     6fbed758 12e9a970 00766982 b4cbc918
1fc0: 6fa59868 12e9a970 6fcf2b80 00000000 6f8c9ee0 9960a500 00000000 00000000
1fe0: 99d23208 993fb450 72c39d27 72c352b4 000b0030 ffffffff
---hid_irq_in: usage=10006---
/dev/input/event0: EV_MSC       MCPU: 0 PID: 1415 Comm: InputReader Not tainted 3.14.52-g7d19054-dirty #16
[<c0016954>] (unwind_backtrace) from [<c0012530>] (show_stack+0x10/0x14)
[<c0012530>] (show_stack) from [<c0abd8e8>] (dump_stack+0x7c/0xbc)
[<c0abd8e8>] (dump_stack) from [<c069d00c>] (input_handle_event+0x43c/0x560)
[<c069d00c>] (input_handle_event) from [<c069d20c>] (input_inject_event+0x7c/0x80)
[<c069d20c>] (input_inject_event) from [<c06a21e8>] (evdev_write+0x90/0xcc)
[<c06a21e8>] (evdev_write) from [<c012d688>] (vfs_write+0xa8/0x198)
[<c012d688>] (vfs_write) from [<c012dc6c>] (SyS_write+0x44/0x9c)
[<c012dc6c>] (SyS_write) from [<c000e620>] (ret_fast_syscall+0x0/0x38)
SC_SCAN             00070053
/dev/input/event0: EV_K---hid_irq_in: usage=10006---
EY       KEY_NUMLOCK          UP
/dev/input/event0: EV_SYN       SYN_REPORT           00000000
/dev/input/event0: EV_LED       LED_NUML             00000001
```

Linux自动挂载U盘和SD卡

mdev工具的简单理解

> mdev是busybox中的一个udev管理程序的一个精简版，他也可以实现设备节点的自动创建和设备的自动挂载，只是在实现的过程中有点差异，在发生热插拔时间的时候，mdev是被hotplug直接调用，这时mdev通过环境变量中的 ACTION 和 DEVPATH，来确定此次热插拔事件的动作以及影响了/sys中的那个目录。接着会看看这个目录中是否有“dev”的属性文件，如果有就利用这些信息为这个设备在/dev 下创建设备节点文件。
>
> mdev扫描/sys/block是为了实现向后兼容)和/sys/class两个目录下的dev属性文件，从该dev 属性文件中获取到设备编号(dev属性文件以"major:minor\n"形式保存设备编号)，并以包含该dev属性文件的目录名称作为设备名 device_name(即包含dev属性文件的目录称为device_name，而/sys/class和device_name之间的那部分目录称为 subsystem。也就是每个dev属性文件所在的路径都可表示为/sys/class/subsystem/device_name/dev)，在 /dev目录下创建相应的设备文件。例如，cat /sys/class/tty/tty0/dev会得到4:0，subsystem为tty,device_name为tty0。

实现方法如下：

1、在/rootfs/etc/init.d/rcS文件中添加如下命令：

```
mount -t tmpfs mdev /dev
mount -t sysfs sysfs /sys
echo /sbin/mdev > /proc/sys/kernel/hotplug
/sbin/mdev -s
```

2、在/rootfs/etc/中增加mdev.conf文件，文件内容如下：

```
mmcblk[0-9]p[0-9] 0:0 666   * (/etc/hotplug/automount "/mnt/sdcard" $MDEV $ACTION)
sd[a-z][0-9]      0:0 666   * (/etc/hotplug/automount "/mnt/udisk"  $MDEV $ACTION)
```

> 其中，*表示创建设备节点后和删除设备节点前都去执行后面的命令；如果使用@，则表示在创建设备节点后去执行后面的命令；如果使用$，则表示在删除设备节点前去执行后面的命令。

3、在/rootfs/etc/hotplug中增加automount文件，文件内容如下：

```
#!/bin/sh

if [ "$1" = "" -o "$2" = "" -o "$3" = "" ]; then
    exit 1
fi

mountPath="$1"

if [ "$3" = "add" ]; then
    blockid=`blkid /dev/$2`
    if [ "$blockid" = "" ]; then
        exit 1
    fi
```

```
    tmp='"'
    labelname="${blockid#*"LABEL=$tmp"}"
    labelname="${labelname%%"$tmp UUID="*}"

    if [ "$labelname" = "$blockid" ]; then
        echo "labelname is null" > /dev/ttyS0
        exit 1
    fi

    if [ -b /dev/$2 ]; then
        if [ ! -d $mountPath/$2 ]; then
            mkdir -p $mountPath/$2
        fi
        mount /dev/$2 $mountPath/$2
    fi
elif [ "$3" = "remove" ]; then
    sync
    umount $mountPath/$2
    rm -rf $mountPath/$2
else
    exit 1
fi
```

4、设置/rootfs/etc/hotplug/automount文件的权限，使其可以被执行

```
chmod 777 automount
```

5、制作system.img

```
sudo make_ext4fs -s -l 768M system.img rootfs/
```

Yocto

单独编译某个recipe

当需要单独编译某个recipe时，可以执行如下命令：

1. 进入build目录

```
cd build-xxx
```

2. 重新编译

```
bitbake -c cleansstate linux-imx
bitbake linux-imx
```

其中，linux-imx表示某个recipe名称，对应路径为sources/meta-xxx-bsp/meta-xxx-arm/recipes-kernel/linux/

该路径包含内容如下：
linux-imx (目录，存放源代码，如dts和patch文件等)
linux-imx_4.1.15.bbappend (配方文件）

自定义能编译出最小系统的Layer

假设已经在当前会话窗口中进行过环境设置，即执行过**MACHINE=imx6qvab820 source via-setup-release.sh -b build-vab820**，保证一些命令能正常使用

1. 进入到sources目录，并创建自定义的layer

```
cd sources
yocto-layer create my
```

在创建layer过程中，按照设置向导的提示对layer进行设置：
Please enter the layer priority you'd like to use for the layer: [default: 6] 6
Would you like to have an example recipe created? (y/n) [default: n] y
Please enter the name you'd like to use for your example recipe: [default: example] my Would you like to have an example bbappend file created? (y/n) [default: n] n

New layer created in meta-my.

Don't forget to add it to your BBLAYERS (for details see meta-my\README).

layer创建完成后，在sources目录自动创建meta-my目录，该目录中包含了基本的配方文件，源代码文件及配置文件等。

2. 适当修改配方文件以编译Linux最小系统

删除示例recipe的源代码目录：

```
rm -rf sources/meta-my/repices-example/example/my-0.1
```

修改示例recipe的配方文件my_0.1.bb内容：

```
SUMMARY = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM =
"file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

inherit core-image

IMAGE_FEATURES += " splash"

PACKAGE_ARCH = "${MACHINE_ARCH}"
```

其中inherit core-image，表示继承core-image.bbclass，其中已经设置好了一些基本配置变量及相关的功能函数，编译my recipe时，就能编译出基本的Linux系统。

3. 使能新建的layer

在via-setup-release.sh中使能meta-my层:

```
echo "BBLAYERS += \" ${BSPDIR}/sources/meta-my \"" >> $BUILD_DIR/conf/bblayers.conf
```

在via-setup-releash.sh中使能meta-my层后，在初始化环境后，会在build目录的conf/bblayer.bb中生成新增的内容

4. 初始化环境

在yocto根目录执行如下命令来初始化环境：

```
MACHINE=imx6qvab820 source via-setup-releash.sh -b build-mytest
```

5. 编译Linux最小系统

环境初始化完成后，会自动进入build-mytest目录，在该目录编译my recipe

```
bitbake my
```

编译完成后，uboot、kernel和根文件系统将会生成在如下目录中：
build-mytest/tmp/deploy/images/imx6qvab820

Yocto2.4 recovery研究

Service update package checking

1. 从服务器下载OTATimeStamp文件

通过wget命令从服务器下载OTATimeStamp文件，文件内容格式为：

```
timestamp 001.001.001
```

2.根据OTATimeStamp文件内容判断是否需要下载更新包

将下载的OTATimeStamp文件中的timestamp字段与本地OTATimeStamp文件中的timestmp字段进行比较，判断是否需要去下载更新包

3. 下载md5文件

当需要下载更新包时，首先需要下载md5文件，文件名格式为timestamp.md5，其中timestmp为OTATimeStamp文件中提取的timestamp字段值

4. 根据md5文件内容下载更新包

md5文件中列出了能从服务器下载的文件名，service程序会根据解析md5文件内容，逐一下载各个文件

eMMC partition information

分区	分区名称	文件系统类型	分区作用
p1	boot	fat	存放系统正常启动的zImage、dtb文件
p2	rootfs	ext4	存放系统正常启动的rootfs文件
p3	recovery	fat	存放recovery模式的zImage、dtb、recovery.img等
p5	image	ext4	存放OTATimeStamp和下载的zImage、dtb、rootfs文件
p6	data	ext4	临时空间，可以不需要

Create new partition by using parted tool

MBR格式的分区：

```
create_new_partitions()
{
    partprobe
    sync

    echo "unit MiB"                          >  $cmdfile
```

```
    echo "mkpart primary ext4 8 24"            >> $cmdfile
    echo "mkpart primary ext4 25 2073"         >> $cmdfile
    if [ $enable_recovery == "1" ]; then
        echo "mkpart primary ext4 2074 2090"   >> $cmdfile
        echo "mkpart primary ext4 2091 3627"   >> $cmdfile
        echo "mkpart primary ext4 3628 4652"   >> $cmdfile
        echo "mkpart primary ext4 4653 -1"     >> $cmdfile
    else
        echo "mkpart primary ext4 2074 -1"     >> $cmdfile
    fi
    echo "print"                               >> $cmdfile
    echo "quit"                                >> $cmdfile

    partprobe
    sync

    dd if=/dev/zero of=$emmc_dev bs=1k count=1 conv=notrunc
    sync

    partprobe
    sync

    # MBR type paritition
    parted -s $emmc_dev mktable msdos
    partprobe
    sync

    cat $cmdfile | parted $emmc_dev
    partprobe
    sync

    partprobe
    mdev -s
    sync

    # dump partition table
    printf "unit MiB\nprint\nquit\n" | parted $emmc_dev

    # Partprobe will notify udevd to mount partitions.
    # We have to unmount all partitions again.
    umount -f "$emmc_dev"*
    sync
}
```

GPT格式的分区：

```
create_new_partitions()
{
    partprobe
    sync

    echo "unit MiB"                            >  $cmdfile
    echo "mkpart boot ext4 8 24"               >> $cmdfile
    echo "mkpart rootfs ext4 25 2073"          >> $cmdfile
    if [ $enable_recovery == "1" ]; then
        echo "mkpart recovery ext4 2074 2090"  >> $cmdfile
        echo "mkpart extended ext4 2091 3627"  >> $cmdfile
        echo "mkpart image ext4 3628 4652"     >> $cmdfile
        echo "mkpart data ext4 4653 -1"        >> $cmdfile
    else
        echo "mkpart data ext4 2074 -1"        >> $cmdfile
    fi
    echo "print"                               >> $cmdfile
```

```
    echo "quit"                                 >> $cmdfile

    partprobe
    sync

    dd if=/dev/zero of=$emmc_dev bs=1k count=1 conv=notrunc
    sync

    partprobe
    sync

    # GPT type paritition
    parted -s $emmc_dev mktable gpt
    partprobe
    sync

    cat $cmdfile | parted $emmc_dev
    partprobe
    sync

    partprobe
    mdev -s
    sync

    # dump partition table
    printf "unit MiB\nprint\nquit\n" | parted $emmc_dev

    # Partprobe will notify udevd to mount partitions.
    # We have to unmount all partitions again.
    umount -f "$emmc_dev"*
    sync
}
```

Add support for uboot recovery mode

使能Yocto2.4 uboot 自带的recovery功能（即Android recovery功能）的patch如下：

```
diff --git a/configs/mx6qvab820_defconfig b/configs/mx6qvab820_defconfig
index c4e3e23..f7f42f9 100644
--- a/configs/mx6qvab820_defconfig
+++ b/configs/mx6qvab820_defconfig
@@ -2,7 +2,8 @@ CONFIG_ARM=y
 CONFIG_ARCH_MX6=y
 CONFIG_TARGET_MX6QVAB820=y
 CONFIG_VIDEO=y
-CONFIG_SYS_EXTRA_OPTIONS="IMX_CONFIG=board/via/mx6qvab820/vab820.cfg,MX6Q"
+CONFIG_SYS_EXTRA_OPTIONS="IMX_CONFIG=board/via/mx6qvab820/vab820.cfg,MX6Q,ANDROID_SUPPORT"
+CONFIG_EFI_PARTITION=y
 CONFIG_BOOTDELAY=3
 # CONFIG_CONSOLE_MUX is not set
 CONFIG_SYS_CONSOLE_IS_IN_ENV=y
diff --git a/drivers/usb/gadget/command.c b/drivers/usb/gadget/command.c
index e9f7d29..9a20907 100644
--- a/drivers/usb/gadget/command.c
+++ b/drivers/usb/gadget/command.c
@@ -7,7 +7,12 @@
 #include <common.h>
 #include <g_dnl.h>
 #include "bcb.h"
+#ifdef CONFIG_FASTBOOT_STORAGE_NAND
+#include <nand.h>
+#endif
+#include "bcb.h"
```

```
+#ifndef CONFIG_FASTBOOT_STORAGE_NAND
 int bcb_read_command(char *command)
 {
        int ret = 0;
@@ -60,3 +65,57 @@ int bcb_write_command(char *bcb_command)
        free(p_block);
        return 0;
 }
+#else
+#define ALIGN_BYTES 64
+#define MISC_PAGES 3
+int bcb_read_command(char *command)
+{
+       char read_cmd[128];
+       char *addr_str;
+       char *nand_str;
+       ulong misc_info_size;
+       struct mtd_info *nand = nand_info[0];
+       if (command == NULL)
+               return -1;
+       memset(read_cmd, 0, 128);
+       misc_info_size = MISC_PAGES * nand->writesize;
+       nand_str = (char *)memalign(ALIGN_BYTES, misc_info_size);
+       sprintf(read_cmd, "nand read 0x%x ${misc_nand_offset} \
+                       0x%x", nand_str, misc_info_size);
+       run_command(read_cmd, 0);
+       /* The offset of bootloader_message is 1 PAGE.
+        * The offset of bootloader_message and the size of misc info
+        * need align with user space and recovery.
+        */
+       addr_str = nand_str + nand->writesize;
+       memcpy(command, (char *)addr_str, 32);
+       free(nand_str);
+       return 0;
+}
+int bcb_write_command(char *command)
+{
+       char cmd[128];
+       char *addr_str;
+       char *nand_str;
+       ulong misc_info_size;
+       struct mtd_info *nand = nand_info[0];
+       if (command == NULL)
+               return -1;
+       memset(cmd, 0, 128);
+       misc_info_size = MISC_PAGES * nand->writesize;
+       nand_str = (char *)memalign(ALIGN_BYTES, misc_info_size);
+       sprintf(cmd, "nand read 0x%x ${misc_nand_offset} \
+                       0x%x", nand_str, misc_info_size);
+       run_command(cmd, 0);
+       /* the offset of bootloader_message is 1 PAGE*/
+       addr_str = nand_str +  nand->writesize;
+       memcpy((char *)addr_str, command, 32);
+       /* erase 3 pages which hold BCB struct.*/
+       sprintf(cmd, "nand erase ${misc_nand_offset} 0x%x",nand->erasesize);
+       run_command(cmd, 0);
+       sprintf(cmd, "nand write 0x%x ${misc_nand_offset} 0x%x",nand_str, misc_info_size);
+       run_command(cmd, 0);
+       free(nand_str);
+       return 0;
+}
+#endif
diff --git a/drivers/usb/gadget/f_fastboot.c b/drivers/usb/gadget/f_fastboot.c
index 7a4b3d7..cb67415 100755
--- a/drivers/usb/gadget/f_fastboot.c
```

```
+++ b/drivers/usb/gadget/f_fastboot.c
@@ -1422,6 +1422,7 @@ void board_fastboot_setup(void)
        u32 dev_no;
 #endif
        switch (get_boot_device()) {
+       case SPI_NOR_BOOT:
 #if defined(CONFIG_FASTBOOT_STORAGE_MMC)
        case SD1_BOOT:
        case SD2_BOOT:
@@ -1496,6 +1497,7 @@ void board_recovery_setup(void)
 #endif
        int bootdev = get_boot_device();
        switch (bootdev) {
+       case SPI_NOR_BOOT:
 #if defined(CONFIG_FASTBOOT_STORAGE_MMC)
        case SD1_BOOT:
        case SD2_BOOT:
diff --git a/include/configs/mx6qvab820_common.h b/include/configs/mx6qvab820_common.h
index 39dd5aa..3a5a028 100644
--- a/include/configs/mx6qvab820_common.h
+++ b/include/configs/mx6qvab820_common.h
@@ -447,6 +447,7 @@
 #define CONFIG_SUPPORT_RAW_INITRD
 #define CONFIG_SERIAL_TAG

+/*
 #undef CONFIG_EXTRA_ENV_SETTINGS
 #undef CONFIG_BOOTCOMMAND

@@ -474,6 +475,7 @@
        "boot_normal=run bootargs_all; run bspinst_img; boota mmc0\0" \
        "boot_recovery=run bootargs_all; run bspinst_img; boota mmc0 recovery\0" \
        "bootcmd=run boot_normal\0"
+*/

 #define CONFIG_FASTBOOT_BUF_ADDR   CONFIG_SYS_LOAD_ADDR
 #define CONFIG_FASTBOOT_BUF_SIZE   0x19000000
```

Run recovery steps in uboot

```
board_r.c initr_check_fastboot
    |-> f_fastboot.c fastboot_run_mode
            |-> fastboot_get_bootmode
            |       |-> is_recovery_key_pressing
            |       |-> command.c bcd_read_command
            |       |       |-> bcd.c bcd_rw_block // 从misc分区读取数据
            |       |-> bcd_write_command // 清除misc分区的内容，下次启动不会执行recovery
            |-> board_recovery_setup
                    |-> // 设置环境变量，启动recovery系统
```

Enter recovery mode in uboot

uboot判断是否进入recovery模式，有**读取指定寄存器值**和**读取misc分区内容**（yocto2.4 uboot默认采取此方式，与Android8.0类似）这两种方式。

1. 如果采用读取misc分区方式：

需要在reboot前往misc分区写入特定数据，数据结构格式如下：

```
/* keep same as bootable/recovery/bootloader.h */
struct bootloader_message {
    char command[32];    // such as "boot-recovery"
    char status[32];
    char recovery[768];  // such as "recovery"

    /* The 'recovery' field used to be 1024 bytes. It has only ever
     been used to store the recovery command line, so 768 bytes
     should be plenty.  We carve off the last 256 bytes to store the
     stage string (for multistage packages) and possible future
     expansion. */
    char stage[32];

    /* The 'reserved' field used to be 224 bytes when it was initially
     carved off from the 1024-byte recovery field. Bump it up to
     1184-byte so that the entire bootloader_message struct rounds up
     to 2048-byte.
     */
    char reserved[1184];
};
```

构造misc数据文件源代码：

```
#include "stdlib.h"
#include "stdio.h"
#include "string.h"

#define file_name   "misc.bin"

struct bootloader_message {
    char command[32];
    char status[32];
    char recovery[768];
    char stage[32];
    char reserved[1184];
};

struct bootloader_message boot;

int main(int argc, char *argv[])
{
    FILE *file;

    if((file = fopen(file_name, "wb")) == NULL) {
        printf("open %s failed\n", file_name);
        return -1;
    }

    memset(&boot, 0, sizeof(struct bootloader_message));
    strcpy(boot.command, "boot-recovery");
    strcpy(boot.recovery, "recovery");

    fwrite(&boot, 1, sizeof(struct bootloader_message), file);

    fclose(file);

    printf("%s has been generated successfully\n", file_name);

    return 0;
}
```

执行如下命令编译源代码，生成可执行文件misc：

```
gcc misc.c -o misc
```

执行如下命令生成misc.bin数据文件：

```
./misc
```

执行如下命令将misc.bin数据文件写入misc分区：

```
dd if=misc.bin of=/dev/mmcblk0p3  # 假如misc分区文件为/dev/mmcblk0p3
sync
```

重启系统后，uboot便能从misc分区读取recovery需要的数据，并设置环境变量，进入recovery模式：

```
Fastboot: Got Recovery key pressing or recovery commands!
setup env for recovery..
Hit any key to stop autoboot:  0
boota mmc1 recovery
```

Write misc data in Android8.0

在Android8.0中，执行reboot时（命令行执行reboot时调用reboot.c中的main函数，Android上层则是调用android_reboot），会设置sys.powerctl属性，使得init进程监听到属性变化，然后执行如下流程将command数据写入misc分区：

```
android_reboot.c android_reboot或reboot.c main
    |-> init.cpp HandlePowerctlMessage
            |-> bootloader_message.cpp write_reboot_recovery
                    |-> // 往misc分区写入数据
```

MCU

Misc

记录一些比较零散的笔记，没有具体的体系结构。

gitbook安装

GitBook 是一个基于 Node.js 的命令行工具，支持 Markdown 和 AsciiDoc 两种语法格式，可以输出 HTML、PDF、eBook 等格式的电子书，使得所记录的知识更便于管理和传播。

安装步骤如下：

1. 下载安装node.js

> 下载地址: https://nodejs.org/en/download/，如果是下载的非安装包，需要设置Path环境变量

2. 安装gitbook工具

```
npm install -g gitbook-cli
gitbook -V
```

3. 安装 Calibre

> 下载地址： https://calibre-ebook.com/download_windows，执行可执行文件安装完成后，Path环境变量会被自动设置

使用步骤如下：

1. 进入笔记本根目录，并执行如下命令初始化gitbook

```
gitbook init
```

执行初始化命令后，会在根目录生成如下两个文件

> - README.md —— 笔记的介绍写在这个文件里
> - SUMMARY.md —— 笔记的目录结构在这里配置

2. 编辑完成笔记内容后，生成电子版文档

```
gitbook init —— 根据最新笔记内容，重新生成必要的目录
gitbook pdf ./ ./mybook.pdf —— 生成pdf电子书
```

当然，还可以生成html格式的电子书

```
gitbook build [书籍路径] [输出路径]
```