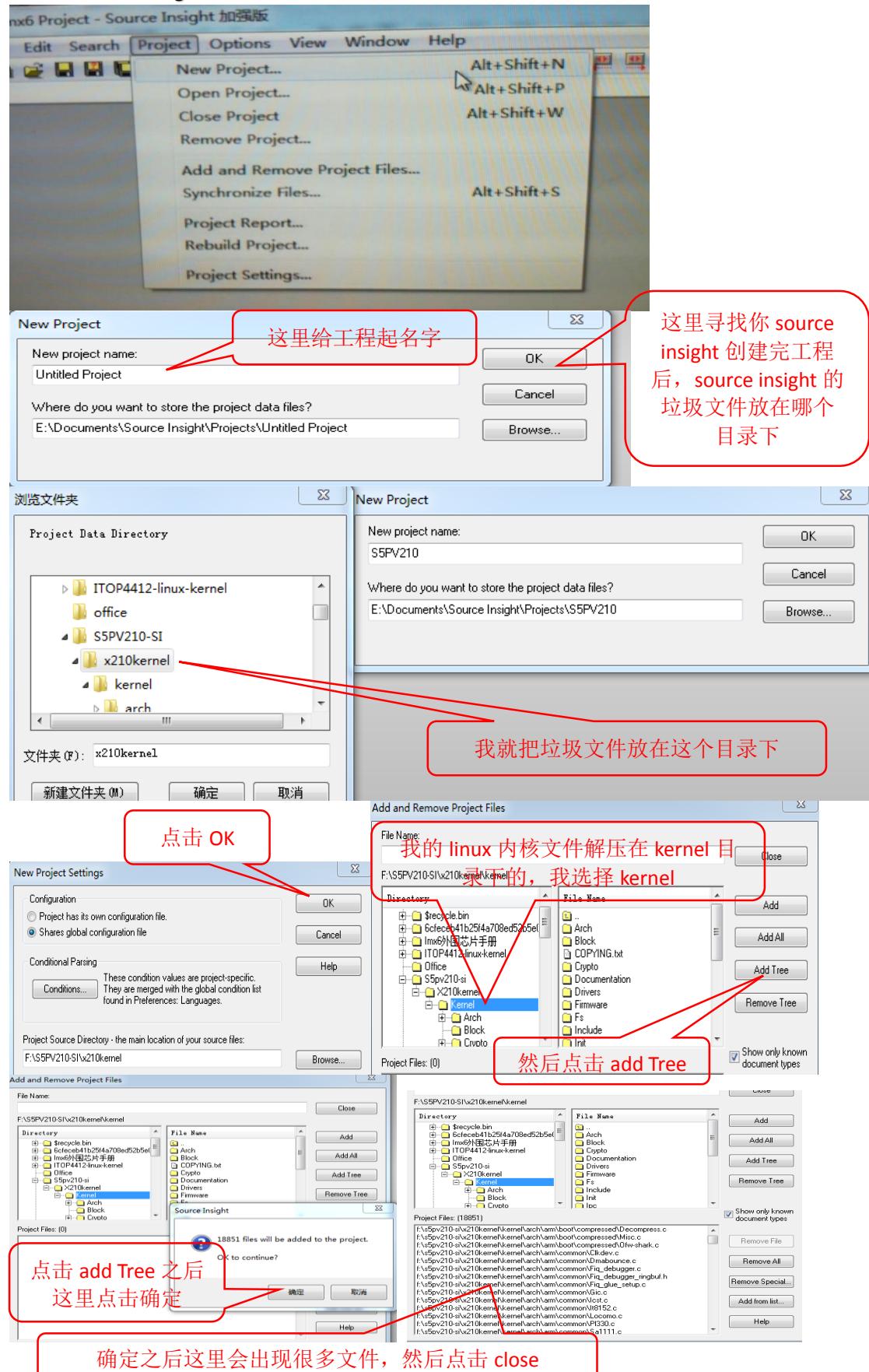

S5PV210 Linux 系统移植

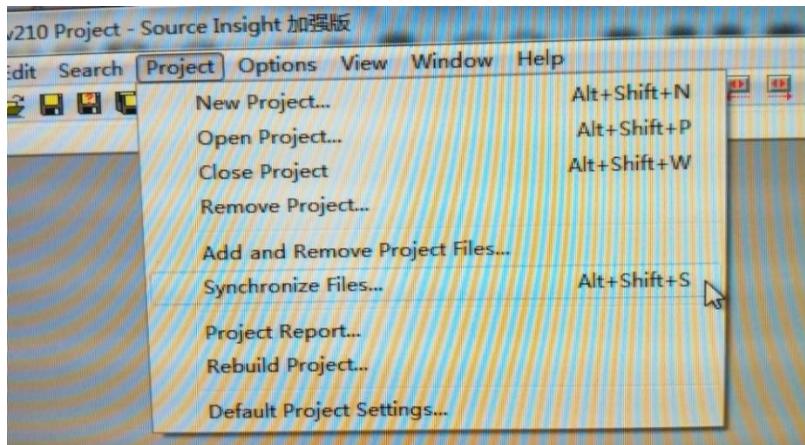
作者:向仔州

Source insight 工程创建.....	2
Uboot 使用.....	6
Uboot 常用命令.....	6
Uboot ping 设置功能: 支持 tftp 和 nfs.....	7
Uboot 系统移植	10
Uboot 如何编译出来.....	11
Uboot 重点目录结构概览.....	11
Uboot 的主 Makefile 分析.....	14
make x210_sd_config 配置的时候细节分析.....	21
Uboot 链接脚本, 下载 uboot 到开发板, uboot 启动的第一个程序就在那里指定.....	24
Uboot 启动分析.....	23
Uboot 启动 kernel(内核).....	53
移植不同型号 INAND/SD 卡启动 Uboot 和 kernel 内核,	54
从三星官方移植 uboot 实验.....	60
Inand 驱动问题修改.....	65
网卡芯片在 Uboot 上移植.....	66
UBOOT 启动时 LCD 显示开机界面.....	70
S5PV210 官方内核移植,编译.....	72
Linux 内核移植原理.....	73
Kconfig 文件的格式详解.....	73
Linux 内核启动过程.....	77
Linux 内核打补丁.....	79
根文件系统制作成镜像.....	82
ext3 文件系统制作.....	82
移植 busybox 文件系统.....	85
NFS 网络文件系统.....	90
busybox 文件系统原理.....	96
下面我们来建立 rcs 文件.....	97
文件系统用户登录密码设置.....	99
文件系统实现链接动态链接库.....	101
增加开机自动启动功能.....	102
将 nfs 移植好的文件系统 rootfs 制作成镜像烧录进开发板.....	103
TFTP 下载大容量内核和文件系统.....	105
Linux 系统使用 SCSI 命令访问 SCSI 总线上硬盘, CD-ROM 信息的方法.....	111
ARM Linux 动态库编译链接和指定新的默认动态库存放位置.....	115
1.在 X86 的 linux(ubuntu)平台下编译动态链接库, 然后使用.....	115
2.如果我们将.so 库程序编译到 ARM linux 开发板上怎么做呢?	116

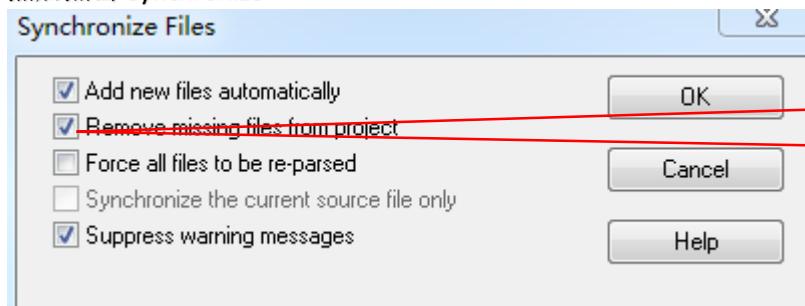
Source insight 工程创建

我们用 source insight 创建 S5PV210 的工程，以便以后查阅代码

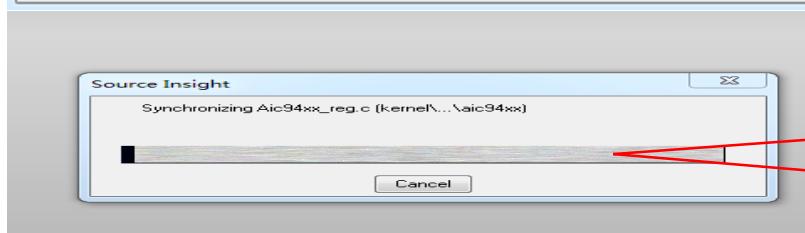




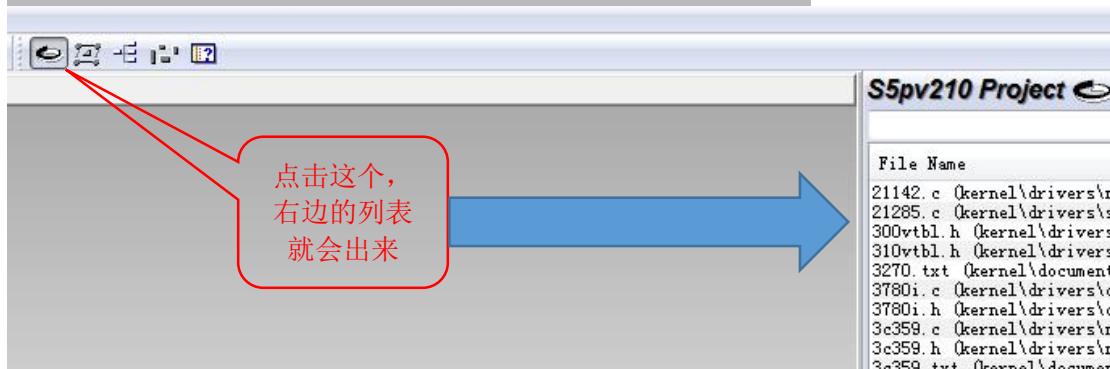
然后点击 Synchronize



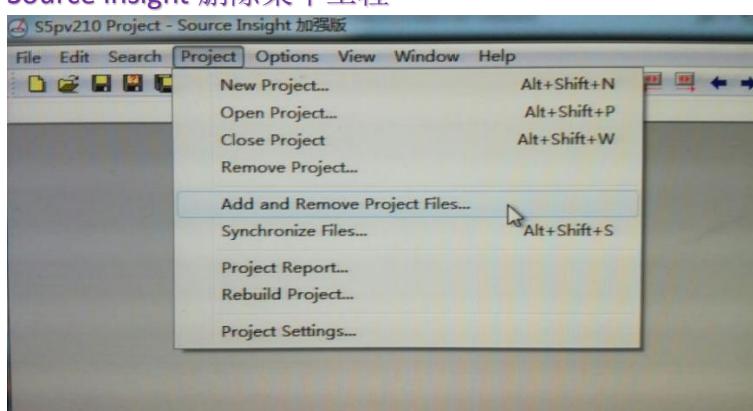
把勾选上，
然后 OK

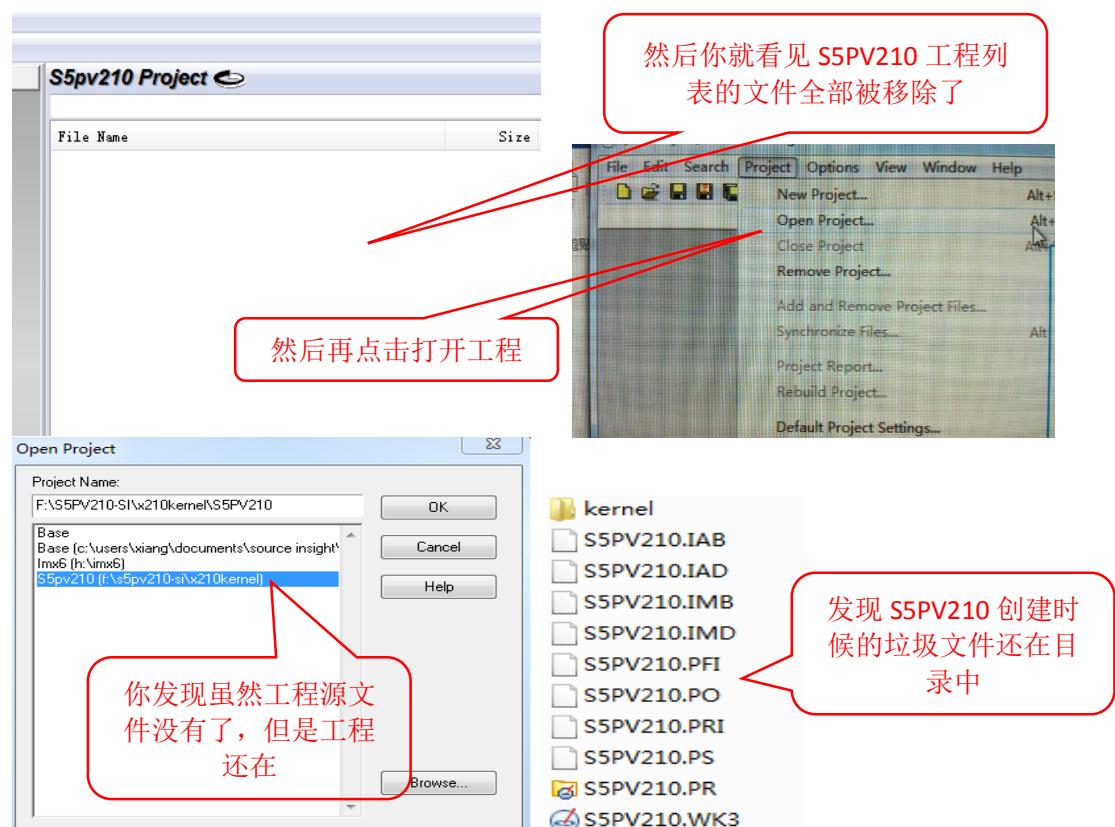
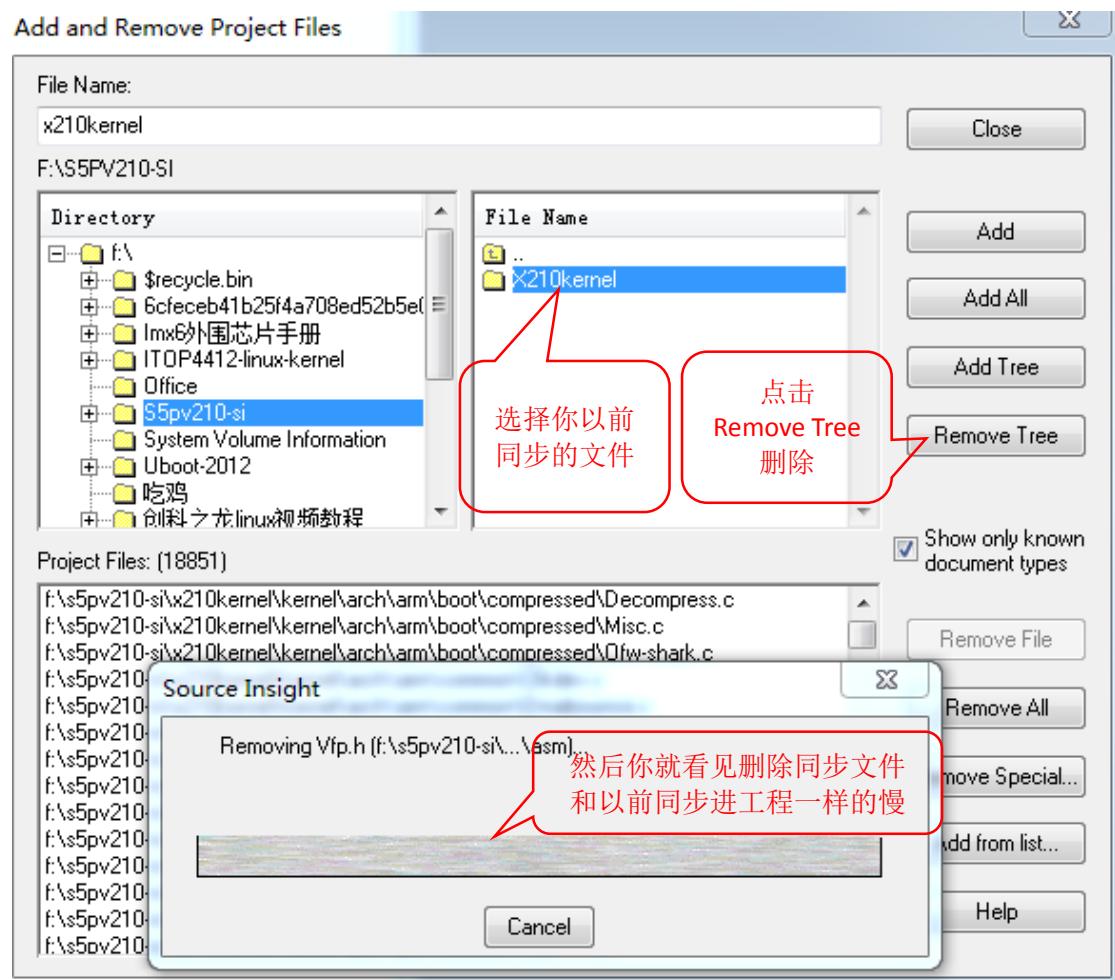


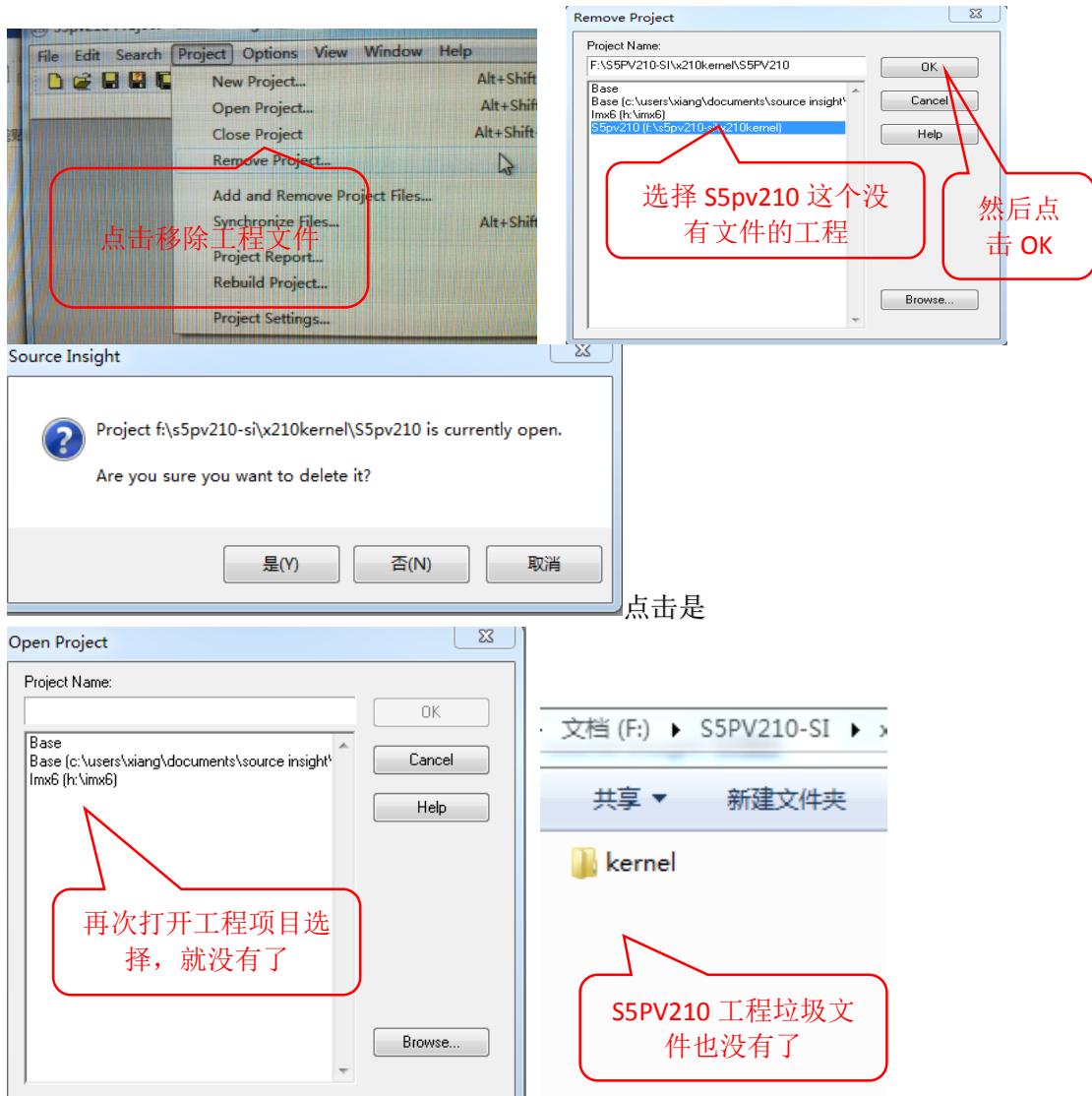
有个很长时
间的同步，
耐心等待



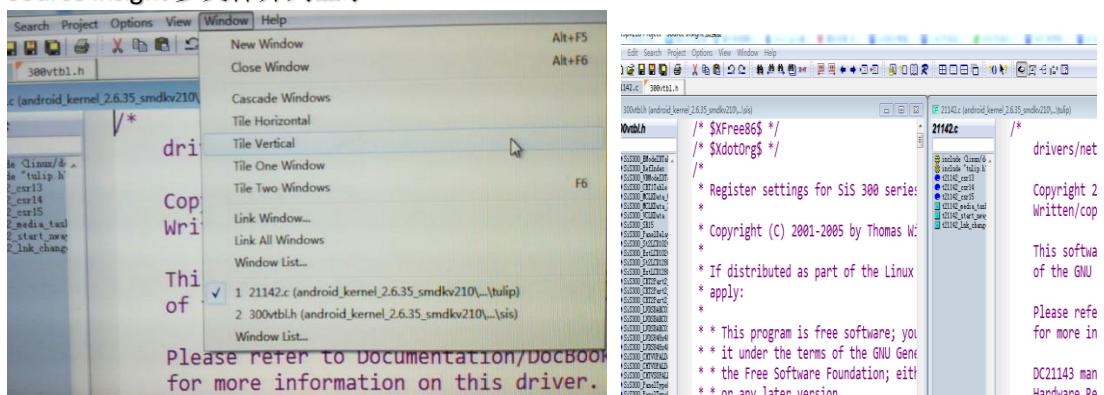
Source insight 删除某个工程



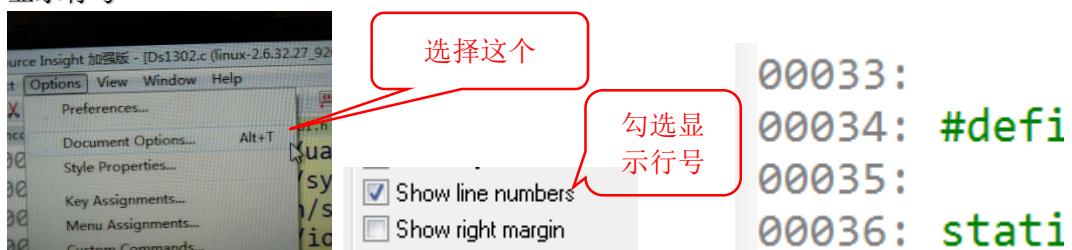




Source insight 多文件并列显示



显示行号



Uboot 使用

Uboot 常用命令

Print : 打印 Uboot 系统环境变量

```
x210 # print
bootcmd=movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000
mtdpart=80000 400000 300000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192.168.0.1
rc console=ttySAC2,115200

Environment size: 440/16380 bytes
```

环境变量设置后，就会存储在 **EMMC/Inand** 里面的一块区域，下次开机环境变量设置后的值还是存在

set: 设置 Uboot 里面的环境变量

格式 **set name value**

```
x210 # print
bootcmd=movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000
mtdpart=80000 400000 300000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192.168.0.1
rc console=ttySAC2,115200

Environment size: 440/16380 bytes
```

这些=左边的值都是
环境变量

bootdelay=3 表示 Uboot 启动后有 3 秒你敲键盘的时间进入 Uboot，如果 3 秒过了你还没有敲键盘，就进入 **kernel** 启动了。

set bootdelay 10 我将 boot 启动时间改为 10 秒

```
x210 # set bootdelay 10
x210 # print
bootcmd=movi read kernel 300080
mtdpart=80000 400000 3000000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=
rc console=ttySAC2,115200
otdelay=10
```

设置成 10 秒

Print 查看环境变
量变成 10 秒了

但是这样改了，重启开发板还是发现没有更改成功，你必须要 **save** 保存，**save** 后修改值才会进入 **EMMC** 保存

```
x210 # set bootdelay 10
x210 # save
Saving Environment to SMDK bootable device...
done
x210 #
```

我再重启开发板

```
Board: XZ10
DRAM: 512 MB
Flash: 8 MB
SD/MMC: 3728MB
In: serial
Out: serial
Err: serial
[LEFT UP] boot mode
checking mode for fastboot ...
Hit any key to stop autoboot: 8
```

你看这里时间就
变了

Uboot ping 设置功能：支持 tftp 和 nfs
先测试开发板和 PC 机网络连接是否正常
没有基本网络后忘记直接连接

The screenshot shows the Windows Control Panel's Network and Sharing Center. It displays a network diagram with three nodes: 'XIANG-PC (此计算机)' (Local Computer), '网络 5' (Network 5), and 'Internet'. A connection line goes from the local computer to the network, and another from the network to the Internet. Below this, a '本地连接' (Local Connection) icon is shown, connected to 'Realtek PCIe GBE Family Controller'. A tooltip for this connection indicates it is a '家庭网络' (Home Network) with '访问类型: Internet', '家庭组: 已加入' (Joined Home Group), and '连接: 本地连接' (Connection: Local Connection). To the right, there is a diagram illustrating a direct physical connection between a green 'PC 主机' (Host PC) and an orange '开发板' (Development Board) with a red double-headed arrow labeled '网线' (Network Cable). A speech bubble points to the IP address field in the network configuration window, stating '这是主机的网段设置 192.168.1.....' (This is the host's segment setting 192.168.1....).

本地连接 属性

网络 共享

连接时使用：
Realtek PCIe GBE Family Controller

配置(C)...
此连接使用下列项目(0)：

- Microsoft 网络客户端
- VMware Bridge Protocol
- QoS 数据包计划程序
- Microsoft 网络的文件和打印机共享
- Internet 协议版本 6 (TCP/IPv6)
- Internet 协议版本 4 (TCP/IPv4)

IP 地址(I): 192 . 168 . 1 . 10
子网掩码(U): 255 . 255 . 255 . 0
默认网关(D): . . .

设置开发板 uboot 网络地址 ipaddr

```
x210 # set ipaddr 192.168.1.20
```

然后我 print 查看下开发板地址设置是否正确

```
serverip=192.168.0.141  
bootdelay=3  
ipaddr=192.168.1.20  
gatewayip=192.168.1.1
```

ipaddr 网段和主机相同
gatewayip 网关的网段也要和主机在一个网段

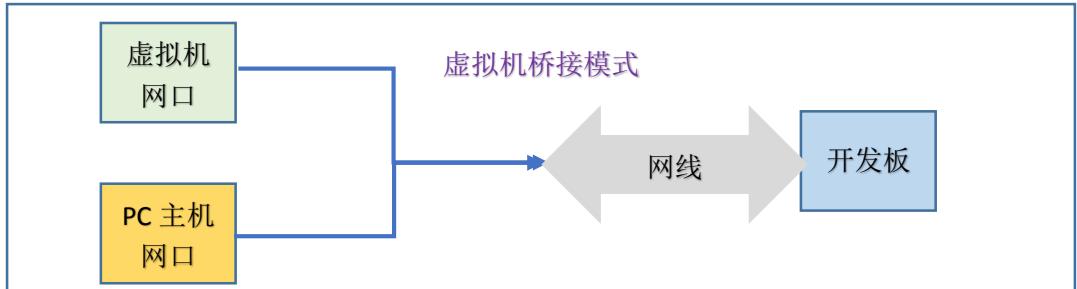
这样 PC 机器和主机就能 ping 通。然后我们开发板和 PC 机各自接入路由器，从新设置连接



首先确保你的 PC 主机里面的 ubuntu 虚拟机是桥接模式



开发板经过网线只能看到 windows 主机的网卡 ip 地址，看不到 ubuntu 虚拟机 IP 地址
所以开发板无法直接 ping 虚拟机 IP 地址，得经过 windows 主机软件转换才行



开发板可以直接 ping 虚拟机的 IP 地址，因为虚拟机和 PC 主机的是并行用一个网卡
所以为了方便我们虚拟机都设置成桥接模式

先下载 linux+QT 系统给开发板，然后给开发板插入网线。

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux# ifconfig
eth0      Link encap:Ethernet HWaddr 00:0c:29:c4:2e
          inet addr:192.168.14.12  Bcast:192.168.14.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe2e/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:28498 errors:0 dropped:0 overruns:0 frame:0
            TX packets:4041 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:9361648 (9.3 MB)  TX bytes:344964 (344.9 KB)

lo       Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:65536  Metric:1
            RX packets:54 errors:0 dropped:0 overruns:0 frame:0
            TX packets:54 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:4506 (4.5 KB)  TX bytes:4506 (4.5 KB)

root@ubuntu:/home/xiang/S5PV210/S5PV210-linux# ping 192.168.14.36
PING 192.168.14.36 (192.168.14.36) 56(84) bytes of data.
64 bytes from 192.168.14.36: icmp_seq=1 ttl=64 time=1.26 ms
64 bytes from 192.168.14.36: icmp_seq=2 ttl=64 time=1.01 ms
64 bytes from 192.168.14.36: icmp_seq=3 ttl=64 time=0.992 ms
^C
--- 192.168.14.36 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2004ms
rtt min/avg/max/mdev = 0.992/1.091/1.268/0.128 ms

[root@x210v3 ~]# ifconfig
Welcome to Buildroot
x210v3 login: root
Password:
[root@x210v3 ~]# ls
[root@x210v3 ~]# ifconfig
eth0      Link encap:Ethernet HWaddr 00:09:C0:FF:EC:48
          inet addr:192.168.14.36  Bcast:192.168.14.255
          inet6 addr: fe80::209:c0ff:feec48/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:35 errors:0 dropped:0 overruns:0 frame:0
            TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:3278 (3.2 Kib)  TX bytes:1124 (1.0 KiB)
Interrupt:42 Base address:0x4300

lo       Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:16436  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

[root@x210v3 ~]# ping 192.168.14.12
PING 192.168.14.12 (192.168.14.12) 56 data bytes
64 bytes from 192.168.14.12: seq=0 ttl=64 time=10.765 ms
64 bytes from 192.168.14.12: seq=1 ttl=64 time=0.473 ms
64 bytes from 192.168.14.12: seq=2 ttl=64 time=1.036 ms
64 bytes from 192.168.14.12: seq=3 ttl=64 time=0.504 ms
64 bytes from 192.168.14.12: seq=4 ttl=64 time=1.067 ms

--- 192.168.14.12 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss,
round-trip min/avg/max = 0.473/2.769/10.765 ms
[root@x210v3 ~]#
```

开发板的 linux+QT 系统和电脑虚拟机互相 ping 是通的，证明开发板和电脑网口正常。

现在我们看看在 uboot 环境下，能不能 ping 通虚拟机

```
x210 # set ipaddr 192.168.14.10
```

用 set 命令，将 uboot 的 ipaddr 地址修改成和我虚拟机同一个网段。

IP 地址=192.168.14.10，前面三个红色的数字要和虚拟机 IP 地址一样，这三个红色的数字就是网段。最后一个数字不能和虚拟机一样，但是也不要和家里或者公司其他设备的地址一样。

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:c4:2e
          inet addr:192.168.14.12  Bcast:192.168.14.255  Mask:255.255.255.0
              inet6 addr: fe80::20c:29ff:fe2e:2e%eth0  Brd:fe80::ff:ffff:fe2e:2e
              brd:fe80::ff:ffff:fe2e:2e
```

这是虚拟机 IP 地址

用 print 查看 uboot 环境变量

```
x210 # print
bootcmd=movi read kernel 30008000; movi read roo
mtdpart=80000 400000 3000000
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootdelay=3
bootargs=root=/dev/mmcblk0p2 rootfstype=ext3 ini
ipaddr=192.168.14.10
```

ipaddr 修改成功

```
x210 # ping 192.168.14.12
dm9000 i/o: 0x88000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
host 192.168.14.12 is alive
```

显示 is alive 表示
ping 通了

证明 uboot 是支持网卡驱动的。

如果显示一下结果就表示没有 ping 通

```
x210 # ping 192.168.1.10
dm9000 i/o: 0x88000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
operating at 100M full duplex mode
ping failed; host 192.168.1.10 is not alive
```

显示 host....is not
alive 就表示没有
ping 通

这就是没有 ping 通，这样你就要查看 uboot 网卡驱动有没有问题，因为 linux+QT 我们测试网卡驱动没有问题。那么问题就在 uboot 网卡驱动这里。

Uboot 支持网络 ping 虚拟机的用处是可以用来做 tftp 和 nfs 网络传输。

```
x210 # set serverip 192.168.14.12
x210 # print
bootcmd=movi read kernel 30008000;
mtdpart=80000 400000 3000000
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
netmask=255.255.0.0
bootdelay=3
bootargs=root=/dev/mmcblk0p2 rootfs
ipaddr=192.168.14.10
gatewayip=192.168.14.1
serverip=192.168.14.12
```

将 serverip 改成
你虚拟机的 IP

```
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
serverip=192.168.1.102
netmask=255.255.0.0
bootdelay=3
```

设置 serverip

设置成功

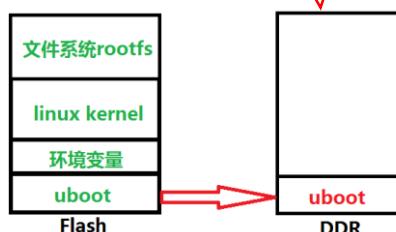
最后 save

Uboot 系统移植



Flash 分区内容

1. 开发板启动的时候 irom 会自动将 flash 的 uboot 代码拷贝到 ddr 下



环境变量

2. DDR 的 Uboot 开始运行了， DDR 的 Uboot 会执行代码将 flash 的环境变量拷贝一份到内存

DDR

```
x210 # print
bootcmd=movi read kernel 300080
mtdpart=8000000 4000000 3000000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=
rc console=ttySAC2,115200
otdelay=10
```

这就是我们前边将 uboot 延时改成 10 秒，但是重启后还是 3 秒的原因，为什么用 save，就是将内存里面改好的环境变量拷贝进 flash



环境变量

3. 用 save 将内存修改后的环境变量拷贝进 flash，掉电存储

DDR

Uboot 如何编译出来

uboot 找到 uboot 目录

```
root@ubuntu:/home/xiang/S5PV210/u-boot/u-boot# ls
api           CHANGELOG          config.mk   disk      fs        libfdt     Makefile   nand_spl    post      tools
api_examples  CHANEGLOG-before-U-Boot-1.1.5  COPYING    doc       image_split lib_generic  mk        net       README
arm_config.mk  Changelog_Samsung      cpu       drivers   include    MAINTAINERS  mkconfig  onenand_b1  rules.mk
board         common             CREDITS   examples lib_arm   MAKEALL    mkmovi   onenand_ip1 sd_fusing
```

这是 uboot 文件结构

```
root@ubuntu:/home/xiang/S5PV210/u-boot/u-boot# make x210_sd_config
Configuring for x210_sd board...
```

配置 x210_sd_config 版本的 uboot, 配置 uboot 成功

```
Makefile  n
mk      n
n      n
```

查看 uboot 根目录下的 Makefile 交叉编译器是否设置正确

```
146 #CROSS_COMPILE = /usr/local/arm/4.2.2-eabi/usr/bin/arm-linux-
147 CROSS_COMPILE = /usr/local/arm/arm-2009q3/bin/arm-none-linux-gnueabi-
148 endif
```

这就是我使用的 S5PV210 官方交叉编译器存放的路径, 确保正确无误.

然后 make 编译

也可以 make -j1, make -j2, make -j4, 这种-j 几就是指定几核编译. 看电脑是几核. 编译速度快慢而已

/uboot# make 我虚拟机是单核, 所以直接 make

```
u-boot
u-boot.bin
u-boot.dis
u-boot.map
u-boot.srec
```

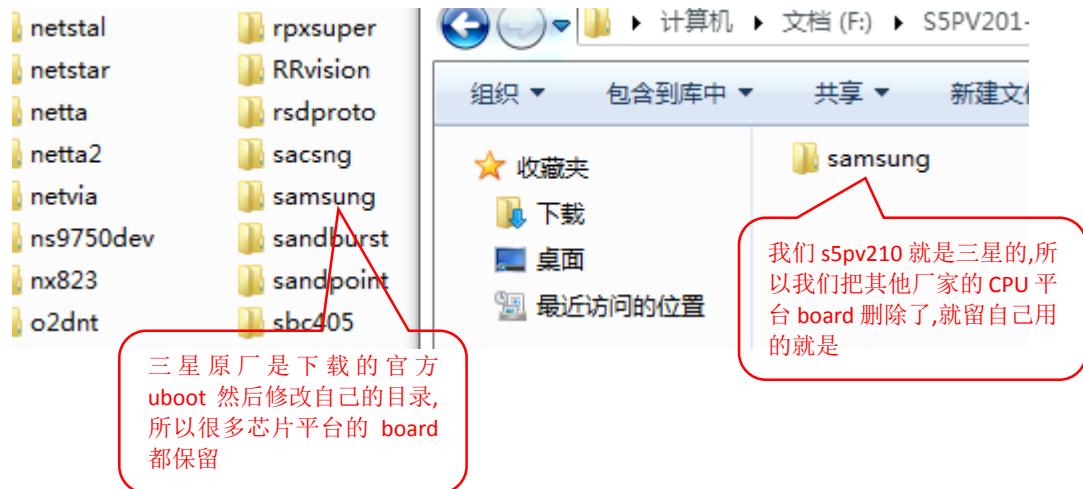
这就是编译成功后的 uboot, 大小是 384k, 下载进开发板就可以了

```
root@ubuntu:/home/xiang/S5PV210/u-boot/u-boot# du -ha u-boot.bin
384K    u-boot.bin
```

Uboot 重点目录结构概览

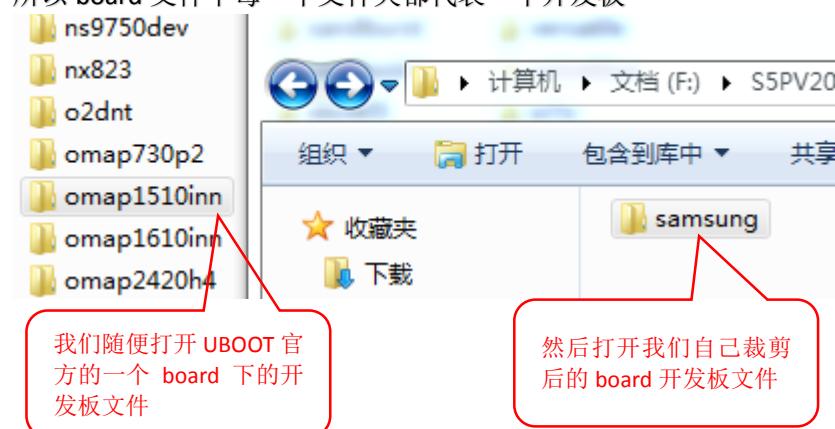


我们打开对比一下



我们 s5pv210 就是三星的, 所以我们把其他厂家的 CPU 平台 board 删除了, 就留自己用的就是

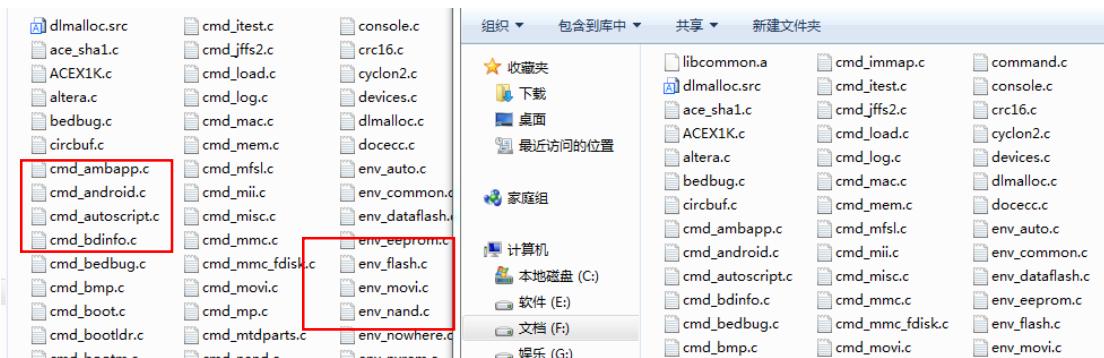
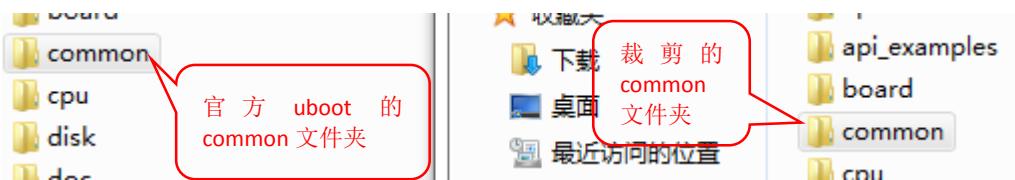
所以 board 文件下每一个文件夹都代表一个开发板



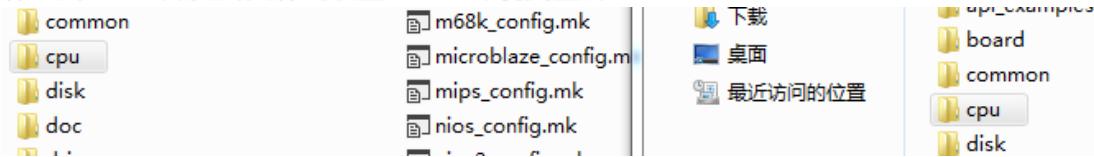
然后打开我们自己裁剪后的 board 开发板文件



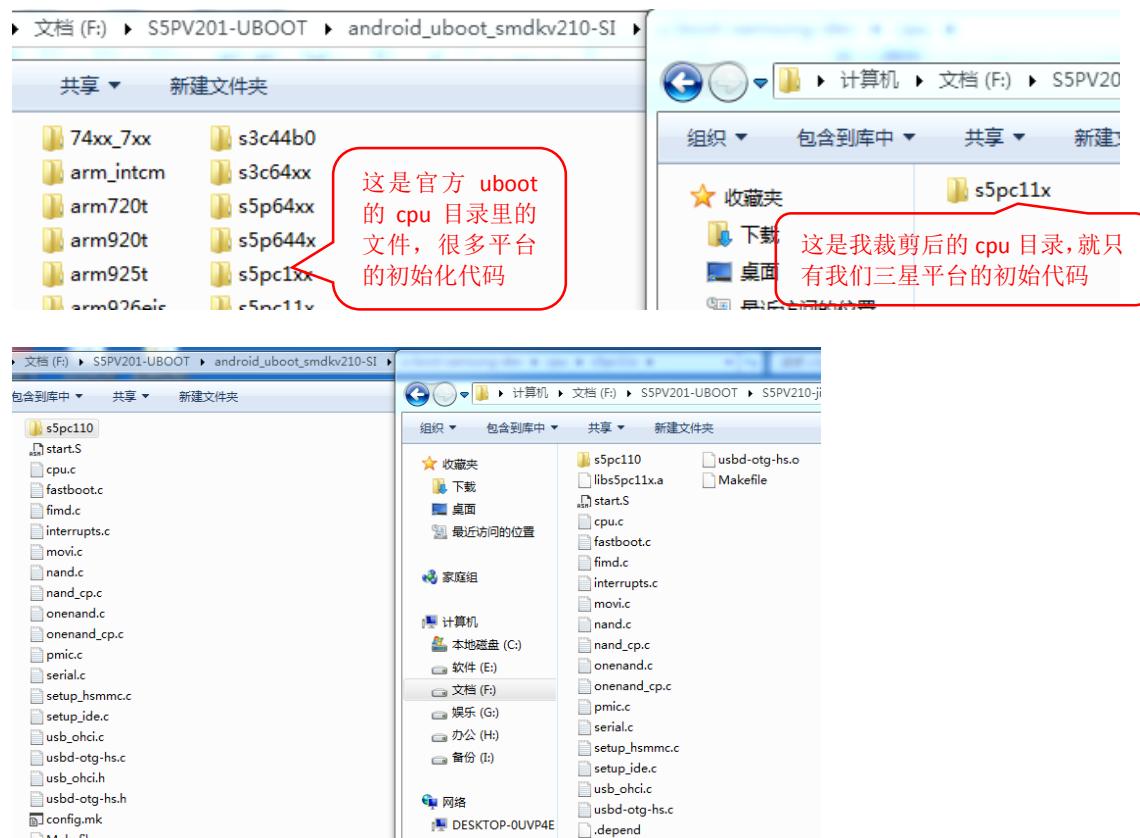
所以 uboot 可移植性是因为 uboot 目录下有很多套代码, 对应不同的开发板



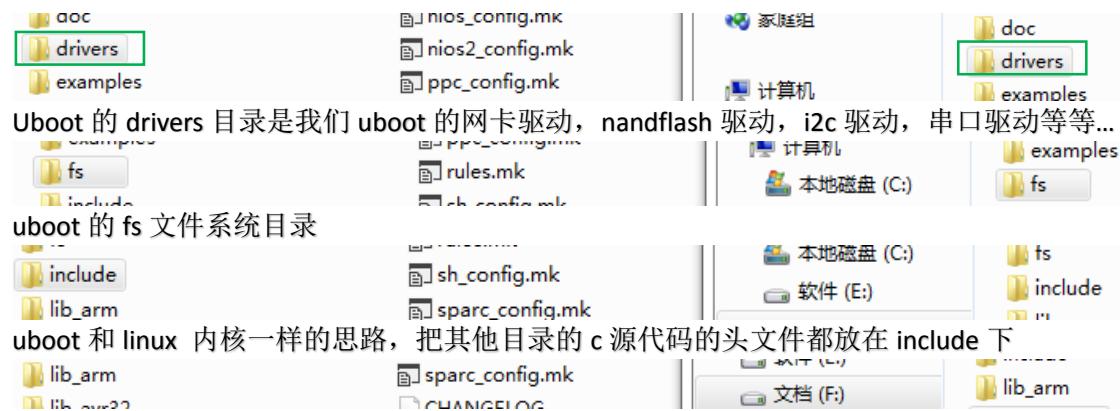
两边 common 文件内容差不多， cmd 开头的文件是实现 uboot 命令的，比如 movi 啊， print 什么的， env 开头的文件是设置 uboot 环境变量的



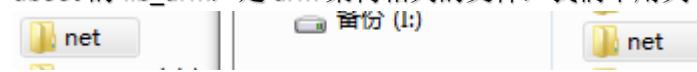
cpu 目录很关键，是和芯片型号有关的初始化和控制代码，比如芯片的 cpu 初始化，中断初始化，串口初始化， iic 之类的。包括启动代码 start.S 也放在这里。



cpu 目录官方的和我裁剪的文件内容是一样的



uboot 的 lib_arm, 是 arm 架构相关的文件, 我们不用关心。

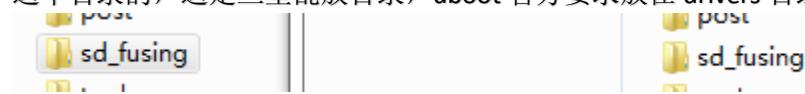


uboot 的 net 目录是包含支持的网络传输协议

这就是 uboot 的 net 目录能支持的网络协议, 比如 tftp 下载, nfs 下载, bootp 自动获取 ip 地址, net 网络接口 ping 命令



uboot 的 onenand 目录就是三星官方自己加的 onenand flash 驱动程序, 官方 uboot 是没有这个目录的, 这是三星乱放目录, uboot 官方要求放在 drivers 目录下。



uboot 的 sd_fusing 就是 sd 卡烧写的驱动

Uboot 的主 Makefile 分析

```
24 VERSION = 1
25 PATCHLEVEL = 3
26 SUBLEVEL = 4
27 EXTRAVERSION =
28 U_BOOT_VERSION = $(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
```

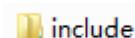
这是 uboot 的版本号

版本号拼接起来放入这个变量

```
29 VERSION_FILE = $(obj)include/version autogenerated.h
```

这个 obj 变量是用=号, 所以有可能定义在 Makefile 后面几行

这个 include 是 uboot 根目录下的 include



Version.... 这个文件在 include 下找不到, 因为要 Makefile 编译过程中才在 include 下生成 version autogenerated.h

```
/uboot/include# vim version autogenerated.h
```

```
#define U_BOOT_VERSION "U-Boot 1.3.4"
```

这就是 virsion....里面的东西

这个字符串就是用上面 U_BOOT_VERSION= ... 生成的

```
U-Boot 1.3.4 (Mar  
CPU: S5PV210@1000  
APLL = 100  
MPLL = 667
```

这个开机启动打印的 u-boot 版本就是我这里定义的

```
1 HOSTARCH := $(shell uname -m | \  
2 | sed -e s/i.86/i386/ \  
3 | -e s/sun4u/sparc64/ \  
4 | -e s/arm.*/arm/ \  
5 | -e s/sa110/arm/ \  
6 | -e s/powerpc/ppc/ \  
7 | -e s/ppc64/ppc/ \  
8 | -e s/macppc/ppc/)  
9  
10 HOSTOS := $(shell uname -s | tr '[[:upper:]]' '[[:lower:]]' | \  
11 | sed -e 's/^(cygwin\).*$/cygwin/')
```

Shell uname 得到当前 ubuntu 使用的 cpu 系统架构, 我这里是 X86_64, 所以得到 X86_64, 然后经过 | 符号, 将 X86_64 字符交给 sed 去替换, 得到 i386 字符给 HOSTARCH 变量 sed 使用看我的 shell 文档

```
3 export| HOSTARCH HOSTOS
```

uname -s 得到 PC 机的操作系统名称 Linux, 然后转换成小写 linux 赋值给变量 HOSTOS

所以现在 HOSTARCH 是 i686, HOSTOS 是 linux

然后用 export 将 HOSTARCH, HOSTOS 导出成环境变量, 给下面的子 Makefile 使用。 (export 导出的环境变量给谁用, 看我的 Makefile 基础文档)

```
9 ifeq (,$(findstring s,$(MAKEFLAGS)))  
10 XECHO = echo  
11 else  
12 XECHO = :  
13 endif
```

make -s 选择这个, 编译的时候就不会打印 Makefile 的信息

```
14 ifdef O  
15 ifeq ("$(origin O)", "command line")  
16 BUILD_DIR := $(O)  
17 endif  
18 endif
```

make 是否带了 O 参数, 带了就执行 ifeq

这个 O 参数有的话就会返回 command line, origin 功能去看我的 Makefile 基础

make O=/tmp

这就是 make 的时候先把路径传给 O, 然后再把路径传给 BUILD_DIR

```

3
4 ifneq ($(BUILD_DIR),)
5 saved-output := $(BUILD_DIR)
6
7 # Attempt to create a output directory.
8 $(shell [ -d ${BUILD_DIR} ] || mkdir -p ${BUILD_DIR})
9
10 # Verify if it was successful.
11 BUILD_DIR := $(shell cd ${BUILD_DIR} && /bin/pwd)
12 $(if ${BUILD_DIR},,$(error output directory "${saved-output}" does not exist))
13 endif # ifneq ($(BUILD_DIR),)
14
15 OBJTREE|| := $(if ${BUILD_DIR} ${BUILD_DIR}, ${CURDIR})
16 SRCTREE|| := ${CURDIR}
17 TOPDIR|| := ${SRCTREE}
18 LNDIR|| := ${OBJTREE}
19 export| TOPDIR SRCTREE OBJTREE
20
21 MKCONFIG| := ${SRCTREE}/mkconfig
22 export MKCONFIG
23
24 ifneq (${OBJTREE}, ${SRCTREE})
25 REMOTE_BUILD| := 1
26 export REMOTE_BUILD
27 endif
28
29 # we also need them before config.mk is included which is the case for
30 # some targets like unconfig, clean, clobber, distclean, etc.
31 ifneq (${OBJTREE}, ${SRCTREE})
32 obj := ${OBJTREE}/
33 src := ${SRCTREE}/
34 else
35 obj :=
36 src :=
37 endif
38 export obj src
39
40 # Make sure CDPATH settings don't interfere
41 unexport CDPATH
42
43 #####
44
45 ifeq ($(ARCH), powerpc)
46 ARCH = ppc
47 endif
48
49 ifeq ($obj) include/config.mk, $(wildcard ${obj}include/config.mk)
50
51 # load ARCH, BOARD, and CPU configuration
52 include ${obj}include/config.mk
53 export| ARCH CPU BOARD VENDOR SOC
54
55 ifndef CROSS_COMPILE
56 ifeq ($(HOSTARCH), $(ARCH))
57 CROSS_COMPILE =
58 else
59 ifeq ($(ARCH), ppc)
60 CROSS_COMPILE = ppc_8xx-
61 endif
62 endif
63
64 image_split
65 include
66 config.mk
67 config.mk_configs

```

这些代码就是指定你编译出来的.o 文件放在指定的目录下，我不用这种，我喜欢.c 和.o 在一个文件下，所以这部分代码不需要分析太深

SRCTREE 指定的是 uboot 源码根目录

指定 uboot 源码根目录下的 mkconfig 路径赋值给 MKCONFIG，然后 export 导出

lib_generic.mk
MAINTAINERS.mkconfig.mk
MAKEFILE.mkmovi

指定 uboot 根目录路径给 obj

Obj 是 uboot 根目录，就是包含根目录下 include 目录里面的 config.mk
config.mk 不是 uboot 自带的，是配置过程中 make_x210_sd_config 生成的 config.mk 文件

include 可以将 config.mk 文件里面的内容展开，ARCH CPU BOARD VENDOR SOC
然后用 export 导出给子 makefile

ARCH	= arm
CPU	= s5pc11x
BOARD	= x210
VENDOR	= samsung
SOC	= s5pc110

```

ARCH = arm
CPU = s5pc11x
BOARD = x210
VENDOR = samsung
SOC = s5pc110

```

这几个值怎么传入 config.mk 的？

在主 Makefile 的第 2589 行，我们在 make 之前执行 make x210_sd_config 配置时执行的

```

2588 x210_sd_config :|      unconfig
2589 |      @$(MKCONFIG) $(@:_config=) arm s5pc11x x210 samsung s5pc110
2590 |      @echo TEXT_BASE = 0xc3e00000" > ${obj}board/samsung/x210/config.mk
2591
2592

```

配置时执行 MKCONFIG 脚本，也就是我前面指定的 mkconfig shell 脚本

然后在执行过程中就把 arm s5pc11x.....参数传入进 mkconfig 脚本了

下面我们来接着分析

```

129
130 ifeq ($obj)include/config.mk,$(wildcard $(obj)include/config.mk)
131
132 # load ARCH, BOARD, and CPU configuration
133 include $(obj)include/config.mk
134 export ARCH CPU BOARD VENDOR SOC
135
136 ifndef CROSS_COMPILE
137 ifeq ($(HOSTARCH),$(ARCH))
138   CROSS_COMPILE =
139 else
140   ifeq ($(ARCH),ppc)
141     CROSS_COMPILE = ppc_8xx-
142   endif
143   ifeq ($(ARCH),arm)
144     #CROSS_COMPILE = arm-linux-
145     #CROSS_COMPILE = /usr/local/arm/4.4.1386-eabi/bin/arm-linux-
146     #CROSS_COMPILE = /usr/local/arm/4.2.2-eabi/usr/bin/arm-linux-
147     CROSS_COMPILE = /opt/arm-2009q3/bin/arm-none-linux-gnueabi-
148   endif
149   ifeq ($(ARCH),i386)
150     CROSS_COMPILE = i386-linux-
151   endif
152   ifeq ($(ARCH),mips)
153     CROSS_COMPILE = mips_4KC-
154   endif
155   ifeq ($(ARCH),nios)
156     CROSS_COMPILE = nios-elf-
157   endif
158   ifeq ($(ARCH),nios2)
159     CROSS_COMPILE = nios2-elf-
160   endif
161   ifeq ($(ARCH),m68k)
162     CROSS_COMPILE = m68k-elf-
163   endif
164   ifeq ($(ARCH),microblaze)
165     CROSS_COMPILE = mb-

```

如果 make CROSS_COMPILE 就执行这一段，否则不执行这一段，我们不执行

这里很多代码都用到了我们上面 export 的 ARCH 变量，所以 ARCH 的值是不能随便改的，只能在 mkconfig 里去选一个值

因为 ARCH=arm，所以选择这一项，
ARCH 就是定义我现在的 uboot 代码
是给哪一个 CPU 工作的，如果
ARCH=i386 那么 uboot 就会去编译
i386 的代码 cortex-a8/usr/bin/arm-l

然后 CROSS_COMPILE 变量会被调用

```
32 export| CROSS_COMPILE  
33  
34 # load other configuration  
35 include $(TOPDIR)/config.mk  
36
```

导出我们设置好的交叉编译器路径变量

TOPDIR 该变量是 uboot 根目录路径

证明了我们下面要分析的是根目录 uboot 的 config.mk，而不是 include/config.mk

```
t/u-boot# ls  
          config.mk  disk  
bt-1.1.5    COPYING  doc  
          cpu       drive
```

```
97 AS | = $(CROSS_COMPILE)as  
98 LD | = $(CROSS_COMPILE)ld  
99 CC | = $(CROSS_COMPILE)gcc  
100 CPP | = $(CC) -E  
101 AR | = $(CROSS_COMPILE)ar  
102 NM | = $(CROSS_COMPILE)nm  
103 LDR | = $(CROSS_COMPILE)ldr  
104 STRIP | = $(CROSS_COMPILE)strip  
105 OBJCOPY = $(CROSS_COMPILE)objcopy  
106 OBJDUMP = $(CROSS_COMPILE)objdump  
107 RANLIB | = $(CROSS_COMPILE)RANLIB  
108
```

我们发现 uboot 根目录 config.mk 就是将主 Makefile 导出的 CROSS_COMPILE 用起来，因为主 Makefile 指定了交叉工具链路径了，这里我们就在工具链后面增加具体编译器就行了

```
111 # Load generated board configuration  
112 sinclude $(OBJTREE)/include/autoconf.mk  
113
```

执行 autoconfig.mk

autoconfig.mk 不是 uboot 源码本身的，是配置 uboot 自动生成的

```
CONFIG_CMD_FLASH=y  
CONFIG_BOOTARGS="console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3"  
CONFIG_CMD_MEMORY=y  
CONFIG_CMD_RUN=y  
CONFIG_IPADDR="192.168.1.88"  
CONFIG_BOOTP_HOSTNAME=y  
CONFIG_MX_LV640EB=y  
CONFIG_SUPPORT_VFAT=y  
CONFIG_DRIVER_SMC911X_16_BIT=y  
CONFIG_CMDLINE_EDITING=y  
CONFIG_CMD_USB=y  
CONFIG_CMD_EXT2=y  
CONFIG_BOOTCOMMAND="movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000"  
CONFIG_CMD_SETGETDCR=y  
CONFIG_FLASH_CET_LEGACY=y
```

这个 autoconfig.mk 是配置后的确认文件，确认无误，uboot 按照这个 autoconfig.mk 规则进行编译。但是这个文件的配置是怎么来的？

这个 autoconfig.mk 配置来源于 uboot 源码自带的平台配置 h 文件，在 include/configs/xxx.h

```
/u-boot/include/configs# vim x210_sd.h
```

```
#define CONFIG_BOOTARGS      "console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3"  
//#define CONFIG_BOOTARGS      "console=ttySAC0,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3"  
#define CONFIG_ETHADDR      00:40:5c:26:0a:5b  
#define CONETC_NETMASK     255.255.0.0
```

这个 h 文件设置的是 x210 的 INAND 版本

这个 x210_sd.h 每个配置项都很重要，这就是设置串口，文件系统之类

还有设置 nand flash 版本的

```
root@ubuntu:/home/xiang/S5PV210/uboot/uboot/include/configs# ls  
x210_nand.h x210_sd.h
```

这个 x210_nand.h 是 nand flash 版本的 u-boot

```
smdkv210onenand.h  
smdkv210single.h  
smdkv210single_ubi.h  
smdkv210vogue.h
```

三星官方还有 4 个 u-boot 版本的配置文件，针对 4 种开发板

下面我们继续分析 u-boot 根目录的 config.mk

```
109 #####  
110  
111 # Load generated board configuration  
112 sinclude $(OBJTREE)/include/autoconf.mk  
113  
114 ifdef| ARCH  
115 sinclude $(TOPDIR)/$(ARCH)_config.mk| # include arch specific rules  
116 endif  
117 ifdef| CPU  
118 sinclude $(TOPDIR)/cpu/$(CPU)/config.mk| # include CPU specific rules  
119 endif  
120 ifdef| SOC  
121 sinclude $(TOPDIR)/cpu/$(CPU)/$(SOC)/config.mk| # include SoC specific rules  
122 endif  
123 ifdef| VENDOR  
124 BOARD_DIR = $(VENDOR)/$(BOARD)  
125 else  
126 BOARD_DIR = $(BOARD)  
127 endif  
128 ifdef| BOARD  
129 sinclude $(TOPDIR)/board/$(BOARD_DIR)/config.mk| # include board specific rules  
130 endif  
131  
132 #####  
142 ifndef LDSCRIPT  
143 #LDSCRIPT := $(TOPDIR)/board/$(BOARD_DIR)/u-boot.lds  
144 ifeq ($(CONFIG_NAND_U_BOOT),y) 如果定义的是 NAND flash 就等于 y,  
如果不是就执行 else 下的参数 debug  
145 LDSCRIPT := $(TOPDIR)/board/$(BOARD_DIR)/u-boot-nand.lds  
146 else  
147 LDSCRIPT := $(TOPDIR)/board/$(BOARD_DIR)/u-boot.lds  
148 endif
```

这个 CONFIG_NAND_U_BOOT 定义在 include/autoconf 下

```
CONFIG_NAND=y  
CONFIG_NAND_BL1_8BIT_ECC=y
```

这个 ARCH 就是我们主 Makefile 导出的环境变量，这里是 ARCH=arm，那么就是配置 u-boot 根目录下的 arm_config.mk
PLATFORM_CPPFLAGS += -DCONFIG_ARM -D_ARM_ 这就是这个参数，我们不管它

这些都是配置编译属性的东西，我们不看了，继续看后面

如果定义的是 NAND flash 就等于 y，如果不是就执行 else 下的参数 debug

看来是要执行这个，这个是指定 u-boot 根目录/board/samsung/x210

u-boot.lds

分析 u-boot 链接就得看这个

下面我们继续分析 u-boot 根目录的 config.mk

```
156 ifneq ($(TEXT_BASE),)  
157 CPPFLAGS += -DTEXT_BASE=$(TEXT_BASE)  
158 endif  
  
2589 x210_sd_config :| unconfig  
2590 | @$(MKCONFIG) $(@:_config=) arm s5pc11x x210 samsung s5pc110  
2591 | @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/x210/config.mk  
2592
```

这个 TEXT_BASE 来自主 Makefile

所以 TEXT_BASE 变量的值是 0xc3e00000

这个 TEXT_BASE 是 uboot 链接时指定的链接地址？？？

下面我们继续分析 uboot 根目录的 config.mk

```
199 ifneq ($(TEXT_BASE),)
200 LDFLAGS += -Ttext $(TEXT_BASE)
201 endif
202
```

其实就是这里-Ttext 指定的
链接脚本，-Ttext 可以看我的
(S5PV210 基础编程文档)

我们回到主 Makefile 继续分析

```
187 #####
188 # U-Boot objects....order is important (i.e. start must be first)
189
190 OBJS = cpu/$(CPU)/start.o
191 ifeq ($(CPU),i386)
192 OBJS += cpu/$(CPU)/start16.o
193 OBJS += cpu/$(CPU)/reset.o
194 endif
195 ifeq ($(CPU),ppc4xx)
196 OBJS += cpu/$(CPU)/resetvec.o
197 endif
198 ifeq ($(CPU),mpc85xx)
199 OBJS += cpu/$(CPU)/resetvec.o
200 endif
201
202 OBJS := $(addprefix $(obj),$(OBJS))
```

指定 uboot/cpu/你用的 CPU 型号/start.S 文件

/uboot/cpu/s5pc11x# ls

start.S

就是这个文件给 OBJS 变量

因为用的不是这些 CPU 平台所以不会执行

这里 OBJS 其实就是集合，把所有的.o 文件集合起来

这里就是把我们需要的.a 静态库放进 LIBS 变量集合
一起编译

```
5 ALL += $(obj)u-boot.srec $(obj)u-boot.bin $(obj)System.map $(U_BOOT_NAND) $(U_BOOT_ONENAND) $(obj)u-boot.dis
6 ifeq ($(ARCH),blackfin)
7 ALL += $(obj)u-boot.ldr
8 endif
9
10 all:| | $(ALL)
11
12 $(obj)u-boot.hex:| $(obj)u-boot
13 | | $(OBJCOPY) ${OBJCFLAGS} -O ihex $< $@
14
15 $(obj)u-boot.srec:| $(obj)u-boot
16 | | $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@
17
18 $(obj)u-boot.bin:| $(obj)u-boot
```

我们 make 之后，就从这里开始编译了

下面有很多都是编译的，我们就不列举了。uboot 编译大框架已经介绍完了。

make x210_sd_config 配置的时候细节分析

再来看主 Makefile

```
x210_sd_config :|      unconfig
|  @$(MKCONFIG) ${@:_config=} arm s5pc11x x210 samsung s5pc110
|  @echo "TEXT_BASE = 0x38000000" > $(obj)board/samsung/x210/config.mk
```

这里就是将 x210_sd_config 替换成 X210_sd,(这个参数去看我的 Makefile 基础的 Makefile 下执行 shell)

然后将参数传进 mkconfig, 执行 mkconfig

```
mkconfig 这个是 mkconfig 执行路径, 主 Makefile 的 MKCONFIG 变量指定了的
$#=6, $1=x210_sd, $2=arm, $3=s5pc11x, $4=x210, $5=samsung, $6=s5pc110
```

```
1 APPEND=no|      # Default: Create new config file
2 BOARD_NAME=""|  # Name to print in make output
3
4 while [ $# -gt 0 ] ; do
5   case "$1" in
6     --) shift ; break ;;
7     -a) shift ; APPEND=yes ;;
8     -n) shift ; BOARD_NAME="${1%_config}" ; shift ;;
9     *) break ;;
10    esac
11 done
12
13 [ "${BOARD_NAME}" ] || BOARD_NAME="$1"
14
15 [ $# -lt 4 ] && exit 1
16 [ $# -gt 6 ] && exit 1
17
18 echo "Configuring for ${BOARD_NAME} board..."
19
20
21 # Create link to architecture specific headers
22 #
23 if [ "$SRCTREE" != "$OBJTREE" ] ; then
24   mkdir -p ${OBJTREE}/include
25   mkdir -p ${OBJTREE}/include2
26   cd ${OBJTREE}/include2
27   rm -f asm
28   ln -s ${SRCTREE}/include/asm-$2 asm
29   LNPREFIX="../../.include2/asm/"
30   cd ..../include
31   rm -rf asm-$2
32   rm -f asm
33   mkdir asm-$2
34   ln -s asm-$2 asm
35
36 else
37   cd ./include
38   rm -f asm
39   ln -s asm-$2 asm
40
41 fi
```

\$#>0 所以执行 while 循环

\$1=x210_sd, 所以 case 没有配上

跳出 while 循环

把 x210_sd 赋值给 BOARD_NAME

如果 \$# < 4 就执行 exit, 退出 shell

如果 \$# > 6 就执行 exit, 退出 shell

如果是在当前目录下编译, .o 文件都生成在.c 文件下, 这两个是相等的, 如果是指定了.o 文件的输出目录这两个就不相等

如果是指定了输出目录, 就会去输出目录的路径下创建 include, include2 文件夹, 然后进入 include2

在指定目录的 include 目录下创建 符号链接

我没有指定目录路径, .o 文件都编译在.c 文件下。所以直接在 uboot/include 目录下创建 符号链接

asm
asm-arm

```
51 rm -f asm-$2/arch  
52  
53 if [ -z "$6" -o "$6" = "NULL" ] ; then  
54 |     ln -s ${LNPREFIX}arch-$3 asm-$2/arch  
55 else  
56 |     ln -s ${LNPREFIX}arch-$6 asm-$2/arch  
57 fi  
58
```

如果以前配置

/uboot/include/asm-arm#

生成了 arch 就删除掉，重新生成

root@uboot:~/arch#

如果 if 成立，在 include/arm-arm

下创建 arch 文件，arch 文件指向

/uboot/include/asm-arm/arch-s5pc11x#

/uboot/include/asm-arm# ls -l arch
arch -> arch-s5pc11x

如果 if 不成立 arch 就软连接 arch-s5pc110，

arch-s5p64xx
arch-s5pc11x
arch-s5pc1xx
arch-s5pc110

因为没有

s5pc110 目录，所以不会创建软链接

```
82 # create link for s5pc11x SoC  
83 if [ "$3" = "s5pc11x" ] ; then  
84     rm -f regs.h  
85     ln -s $6.h regs.h  
86     rm -f asm-$2/arch  
87     ln -s arch-$3 asm-$2/arch  
88  
89 fi
```

在 uboot/include
目录下创建 regs.h
文件

在 uboot/include 下
asm-arm/arch 软连
接 arch-s5pc11x

```
107 if [ "$2" = "arm" ] ; then  
108 |     rm -f asm-$2/proc  
109 |     ln -s ${LNPREFIX}proc-armv asm-$2/proc  
110 fi  
111
```

在 include/asm-arm 下创建一个 proc 文件，这个 proc
文件软连接/include/asm-arm/proc-armv

```

120 #
121 # Create include file for Make
122 #
123 echo "ARCH = $2" > config.mk
124 echo "CPU = $3" >> config.mk
125 echo "BOARD = $4" >> config.mk
126
127 [ "$5" ] && [ "$5" != "NULL" ] && echo "VENDOR = $5" >> config.mk
128
129 [ "$6" ] && [ "$6" != "NULL" ] && echo "SOC = $6" >> config.mk
130
131 #
132 # Create board specific header file
133 #
134 if [ "$APPEND" = "yes" ]|      # Append to existing config file
135 then
136 |     echo >> config.h
137 else
138 |     > config.h|      # Create new config file
139 fi
140 echo "/* Automatically generated - do not edit */" >>config.h
141 echo "#include <configs/$1.h>" >>config.h
142
143 exit 0

```

因为现在我们在 uboot/include 目录下，所以创建的 config.mk 在 include 下，

1	ARCH	= arm
2	CPU	= s5pc11x
3	BOARD	= x210

追加内容给 config.mk

在 include 目录下创建 config.h 文件

```

1 ARCH = arm
2 CPU = s5pc11x
3 BOARD = x210
4 VENDOR = samsung
5 SOC = s5pc110

```

这个 config.mk 创建成功

这个 config.mk 用来干什么，我们前面主 Makefile 分析过了。

```

132 # load ARCH, BOARD, and CPU configuration
133 include $(obj)include/config.mk
134 export| ARCH CPU BOARD VENDOR SOC
135

```

主 Makefile 执行的 config.mk

```
/u-boot, the open source u-boot project
/u-boot/include# vim config.h
```

这个文件很关键

```

1 /* Automatically generated - do not edit */
2 #include <configs/x210_sd.h>

```

x210_sd.h 这个文件会生成一个 autoconfig.mk 文件，这个文件会被 Makefile 引入

Uboot 链接脚本，下载 uboot 到开发板，uboot 启动的第一个程序就在这里指定

/uboot/board/samsung/x210#

我用的是 S5PV210 开发板，所以 uboot 链接脚本在这里

u-boot.lds

```
4 OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
5 /*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
6 OUTPUT_ARCH(arm)
7 ENTRY(_start) _start 就是整个程序的入口
8
9 SECTIONS
10 {
11     . = 0x00000000;
12     这里
13     ALIGN(4),
14     让开始地址
15     4 字节对齐
16     . = ALIGN(4);
17     .text : {
18         cpu/s5pc11x/start.o (.text)
19         cpu/s5pc11x/s5pc110/cpu_init.o (.text)
20         board/samsung/x210/lowlevel_init.o (.text)
21         cpu/s5pc11x/onenand_cp.o (.text)
22         cpu/s5pc11x/nand_cp.o (.text)
23         cpu/s5pc11x/movi.o (.text)
24         common/secure_boot.o (.text)
25         common/ace_sha1.o (.text)
26         cpu/s5pc11x/pmic.o (.text)
27         *(.text)
28     }
29     . = ALIGN(4);
30     .rodata : { *(.rodata) }
31
32     这些 text 代码段的文件
33     排列顺序很
34     重要，因这些
35     代码都是放
36     在 BL1 16K 存
37     储区去执行
38     的，是去初始化
39     外设和
40     CPU 的
41     . = ALIGN(4);
42     .data : { *(.data) }
43
44     . = ALIGN(4);
45     .got : { *(.got) }
46
47     __u_boot_cmd_start = .;
48     __u_boot_cmd : { *(.u_boot_cmd) }
49     __u_boot_cmd_end = .;
50
51     . = ALIGN(4);
52     .mmudata : { *(.mmudata) }
53
54     . = ALIGN(4);
55     __bss_start = .;
56     .bss : { *(.bss) }
57     _end = .;
58 }
```

这是指定程序下载进 CPU 在 sram 运行的开始地址，
这是在链接脚本中指定。

第 2 种方法就是在 Makefile 编译中用-Ttext 指定地址，
这样这个 -Ttext 的地址会覆盖链接脚本这里的. =
0x0000 地址，使-Ttext 指定的地址为程序启动的开始地
址，这两种方法都可以使用，看自己喜好。这个-Ttext
的地址是多少呢？

```
[2585] 210_nand_config.S! unconfig
[2586] @S(MKCONFIG) ${@:_config=) arm s5pc11x x210 samsung
[2587] @echo "TEXT_BASE = @c3e00000" > ${obj}board/samsung
```

-Ttext 指定的地址在主
Makefile TEXT_BASE 指定的

其余都是一样，就
是 这 里
的 .uboot_cmd 段
是我们自定义段

Uboot 启动分析

```
.text      :  
{  
    cpu/s5pc11x/start.o| (.text)  
u-boot.lds 第 1 个程序 start.S  
28 #include <config.h>  
29 #include <version.h>  
30 #if defined(CONFIG_ENABLE_MMU)  
31 #include <asm/proc/domain.h>  
32 #endif  
33 #include <regs.h>  
34  
35 #ifndef CONFIG_ENABLE_MMU  
36 #ifndef CFG_PHY_UBOOT_BASE  
37 #define CFG_PHY_UBOOT_BASE |     CFG_UBOOT_BASE  
38 #endif  
39 #endif
```

1 /* Automatically generated -
2 #include <configs/x210_sd.h>
config.h 包含了 x210_sd.h

判断 x210_sd.h 的
CONFIG_ENABLE_MMU 宏是否定义

start.S 有很多跟宏定义有关的代码段，这些宏都是在 x210_sd.h 下定义的

```
229 #ifdef CONFIG_VOGUES  
230 |     /* PS_HOLD(GPH0_0) set to output high */  
231 |     ldr |     r0, =ELFIN_GPIO_BASE  
232 |     ldr |     r1, =0x00000001
```

比如 CONFIG_VOGUES 等等.....

所以 include<config.h> 将 x210_sd.h 与 start.S 关联了起来

```
root@ubuntu:/home/xiang/S5PV210/uboot/uboot/include/configs# ls  
x210_nand.h  x210_sd.h
```

我们主 Makefile 的 MKCONFIG 为什么配置的是 x210_sd.h 而不是 x210_nand.h 呢？因为 x210_sd.h 里面的宏都是用来设置 INAND flash 的，而 x210_nand.h 里面的宏是来配置 nand flash 的。我现在开发板用的 INAND 存储芯片，所以配置 x210_sd.h

下面我们接着分析 start.S

```
28 #include <config.h>  
29 #include <version.h>  
30 #if defined(CONFIG_ENABLE_MMU)  
31 #include <asm/proc/domain.h>  
32 #endif  
33 #include <regs.h>  
34  
35 #ifndef CONFIG_ENABLE_MMU  
36 #ifndef CFG_PHY_UBOOT_BASE  
37 #define CFG_PHY_UBOOT_BASE |     CFG_UBOOT_BASE  
38 #endif  
39 #endif
```

因为 x210_sd.h 定义了 MMU
97 #define CONFIG_ENABLE_MMU
所以要包含这个 h 文件

因为我们定义了
ENABLE_MMU, 所以下面两句不执行

```

49 #if defined(CONFIG_EVT1) && !defined(CONFIG_FUSED)
50     .word 0x2000
51     .word 0x0
52     .word 0x0
53     .word 0x0
54#endif

```

.word 定义每一行相当于 1 个 int, 这里定义了 16 个字节

X210.h 定义了 EVT1, 没有定义 FUSED 所以满足条件, 执行这句

这 16 个字节就是校验头, 我 S5PV210 基础说过 SD 卡启动或者 INAND 启动, 存储介质里面存放的镜像程序前 16 个字节存放的是校验头。

```

/uboot/sd_fusing# vim C110-EVT1-mkbl1.c
14 int main (int argc, char *argv[])
15 {
16     FILE|      *fp;
17     char|      *Buf, *a;
18     int|      BufLen;
19     int|      nbytes, fileLen;
20     unsigned int|  checksum;
21     int|      i;
22
23 ///////////////////////////////////////////////////////////////////
24     if (argc != 4)
25     {
26         printf("Usage: mkbl1 <source file> <destination file> <size> \n");
27         return -1;
28     }
29
30 ///////////////////////////////////////////////////////////////////
31     BufLen = atoi(argv[3]);
32     Buf = (char *)malloc(BufLen); argv[1]这是我们交叉编译出来的 uboot.bin
33     memset(Buf, 0x00, BufLen);
34
35 ///////////////////////////////////////////////////////////////////
36     fp = fopen(argv[1], "rb"); 打开 uboot.bin
37     if( fp == NULL)
38     {
39         printf("source file open error\n");
40         free(Buf);
41         return -1;
42     }
43
44     fseek(fp, 0L, SEEK_END);
45     fileLen = ftell(fp); 计算 uboot.bin 长度
46     fseek(fp, 0L, SEEK_SET);
47
48     if ( BufLen > fileLen )
49     {
50         printf("Usage: unsupported size\n");
51         free(Buf);
52         fclose(fp);
53         return -1;
54     }
55
56     nbytes = fread(Buf, 1, BufLen, fp); 看看长度是否超标
57     if ( nbytes != BufLen )
58     {
59         printf("source file read error\n");
60         free(Buf);
61         fclose(fp);
62         return -1;
63     }
64
65     fclose(fp);

```

把 uboot.bin 内容读出来

```

68 ///////////////////////////////////////////////////////////////////
69 |     a = Buf + 16; +16
70 |     for(i = 0, checksum = 0; i < BufLen - 16; i++)
71 |     |         checksum += (0x000000FF) & *a++;
72 |
73 |     a = Buf + 8; | buf+8
74 |     *( (unsigned int *)a ) = checksum; buf+8
75 |

76 ///////////////////////////////////////////////////////////////////
77 |     fp = fopen(argv[2], "wb");
78 |     if (fp == NULL)
79 |     {
80 |         printf("destination file open error\n");
81 |         free(Buf);
82 |         return -1;
83 |     }
84 |
85 |     a| = Buf;
86 |     nbytes| = fwrite( a, 1, BufLen, fp); 将前 16 字节校验和， uboot.bin 写入文件，形成新的烧写文件
87 |
88 |     if ( nbytes != BufLen )
89 |     {
90 |         printf("destination file write error\n");
91 |         free(Buf);
92 |         fclose(fp);
93 |         return -1;
94 |     }
95 |
96 |     free(Buf);
97 |     fclose(fp);
98 |

```

下面我们继续接着分析 start.S

```

56 .globl _start
57 _start: b|      reset 板子发生复位跑到 b 来执行
58 |      ldr| pc, _undefined_instruction
59 |      ldr| pc, _software_interrupt
60 |      ldr| pc, _prefetch_abort
61 |      ldr| pc, _data_abort
62 |      ldr| pc, _not_used
63 |      ldr| pc, _irq
64 |      ldr| pc, _fiq 这是异常向量表
65

99 _TEXT_BASE:
100 |      .word| TEXT_BASE
101

```

```

107 _TEXT_PHY_BASE: 定义物理地址的地址
108     .word| CFG_PHY_UBOOT_BASE
109

```

这个 CFG_PHY_UBOOT_BASE 就是在 include/configs/x210_sd.h 里面配置的

```

557 #define CFG_PHY_UBOOT_BASE| MEMORY_BASE_ADDRESS + 0x3e000000
558 #define CFG_PHY_KERNEL_BASE| MEMORY_BASE_ADDRESS + 0x8000

```

MEMORY_BASE_ADDRESS 地址是 0x30000000, 这个地址是 DDR 的起始地址

CFG_PHY_UBOOT_BASE 就是 Uboot 在内存中的物理地址

下面我们继续分析 start.S

```

141 reset:
142     /*
143     * set the cpu to SVC32 mode and IRQ & FIQ disable
144     */
145     @;mrs| r0,cpsr
146     @;bic| r0,r0,#0x1f
147     @;orr| r0,r0,#0xd3
148     @;msr| cpsr,r0
149     msr| cpsr_c, #0xd3| @ I & F disable, Mode: 0x13 - SVC
150

```

所以我们 CPU 上电，将 IRQ,FIQ 中断状态关闭。然后进入 ARM 状态 SVC 模式

```

168 #ifndef CONFIG_EVT1
169 #if 0|
170     bl| v7_flush_dcache_all
171 #else
172     bl| disable_l2cache
173
174     mov| r0, #0x0| @
175     mov| r1, #0x0| @ i|
176     mov| r3, #0x0
177     mov| r4, #0x0
178 lp1:
179     mov| r2, #0x0| @ j
180 lp2:
181     mov| r3, r1, LSL #29|| @ r3 = r1(i) <<29
182     mov| r4, r2, LSL #6|| @ r4 = r2(j) <<6
183     orr| r4, r4, #0x2|| @ r3 = (i<<29)|(j<<6)|(1<<1)
184     orr| r3, r3, r4
185     mov| r0, r3|| @ r0 = r3
186     bl| CoInvalidateDCacheIndex
187     add| r2, #0x1|| @ r2(j)++
188     cmp| r2, #1024|| @ r2 < 1024
189     bne| lp2|| @ jump to lp2
190     add| r1, #0x1|| @ r1(i)++
191     cmp| r1, #8|| @ r1(i) < 8
192     bne| lp1|| @ jump to lp1
193
194     bl| set_l2cache_auxctrl
195
196     bl| enable_l2cache
197 #endif
198 #if 1|
199     bl| disable_l2cache
200     bl| set_l2cache_auxctrl_cycle
201     bl| enable_l2cache
202
203
204
205

```

先禁止 L2cache 关闭 CPU 缓存

初始化 L2cache

再打开 L2cache

```

206
207     * Invalidate L1 I/D
208     */
209     mov r0, #0          @ set up for MCR
210     mcr p15, 0, r0, c8, c7, 0  @ invalidate TLBs
211     mcr p15, 0, r0, c7, c5, 0  @ invalidate icache
212
213     /*
214      * disable MMU stuff and caches
215      */
216     mrc p15, 0, r0, c1, c0, 0
217     bic r0, r0, #0x00002000    @ clear bits 13 (--V--)
218     bic r0, r0, #0x00000007    @ clear bits 2:0 (-CAM)
219     orr r0, r0, #0x00000002    @ set bit 1 (--A-) Align
220     orr r0, r0, #0x00000800    @ set bit 12 (Z--) BTB
221     mcr | p15, 0, r0, c1, c0, 0
222
223     /* Read booting information */
224     ldr r0, =PRO_ID_BASE
225     ldr r1, [r0,#OMR_OFFSET]
226     bic r2, r1, #0xfffffff1

```

刷新 L1cache

关闭 MMU，因为现在没有做虚拟地址和物理地址映射，所以先关闭 MMU

这是获取启动模式，是 SD 启动还是 INAND 启动

PRO_ID_BASE 是个地址值，这个值在 regs.h 里面包含的 s5pc110.h 定义，地址值赋值给 r0 变量

我们将 0xe000 0004 地址里面的值读出来，就知道是 sd 启动还是 inand 启动

还要将 r0 里的地址偏移 4 字节，等于 0xe0000004，这个地址寄存器就是 OMR0~OM5，启动介质选择引脚，是 sd 启动还是 inand 启动

所以这个 r2 的值就决定了 OM 引脚的接法，启动方式，后面代码就判断 r2 的值来决定启动方式

```

242     /* NAND BOOT */
243     cmp r2, #0x0|           @ 512B 4-cycle
244     moveq r3, #BOOT_NAND
245
246     cmp r2, #0x2|           @ 2KB 5-cycle
247     moveq r3, #BOOT_NAND
248
249     cmp r2, #0x4|           @ 4KB 5-cycle| 8-bit ECC
250     moveq r3, #BOOT_NAND
251
252     cmp r2, #0x6|           @ 4KB 5-cycle| 16-bit ECC
253     moveq r3, #BOOT_NAND
254
255     cmp r2, #0x8|           @ OneNAND Mux
256     moveq r3, #BOOT_ONENAND
257
258     /* SD/MMC BOOT */
259     cmp r2, #0xc
260     moveq r3, #BOOT_MMCSD
261
262     /* NOR BOOT */
263     cmp r2, #0x14
264     moveq r3, #BOOT_NOR
265

```

这里就开始判断 r2 了，r2=0x0，就是 NAND flash 启动

唯一的区别在这里，比如这个 nand 是支持页大小 512 字节，4-cycle 的 nand

r2=0x2，也是 NAND flash 启动

r2=0x4，也是 NAND flash 启动

r2=0x6，也是 NAND flash 启动

r2=0x8，是 onenand 启动

r2=0xc，是 sd 卡启动

r2=0x14，是 nor flash 卡启动

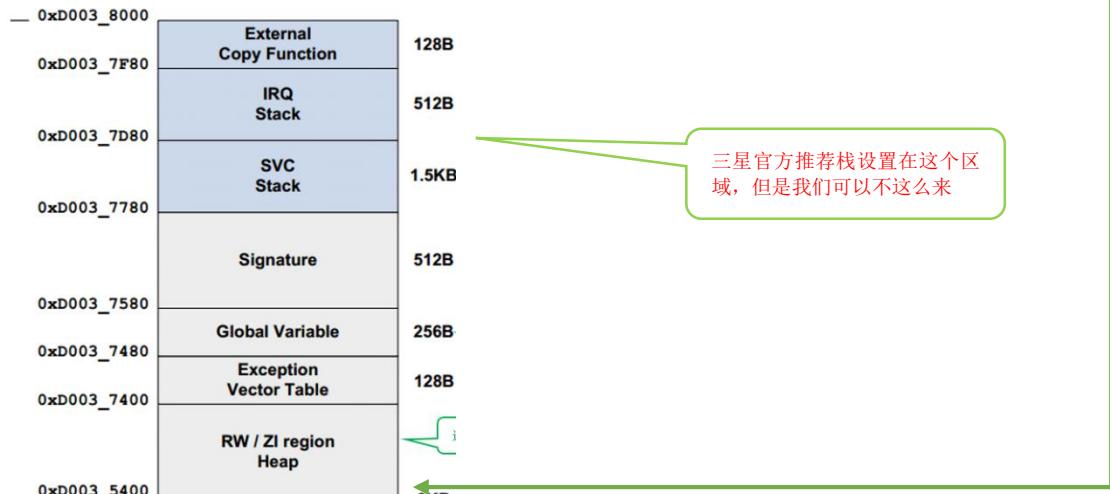
这里的 r3 得到的#BOOT 值，后面会有对应的程序用到

r3 可能等于 BOOT_NAND, BOOT_ONENAND, BOOT_NOR，根据 r2 来确定

```

282 |
283 |     ldr |     sp, =0xd0036000 /* end of sram dedicated to u-boot */
284 |     sub |     sp, sp, #12 | /* set stack */
285 |     mov |     fp, #0
286 |
287 |     bl |     lowlevel_init /* go setup pll,mux,memory */
288 |
289

```



为什么这里要设置栈，因为后面我们要调用函数，还有就是 DDR 现在还没有初始化，所以代码只能在 CPU 内部 sram 用

```

283 |
284 |     ldr |     sp, =0xd0036000 /* end of sram dedicated to u-boot */
285 |     sub |     sp, sp, #12 | /* set stack */
286 |     mov |     fp, #0
287 |
288 |     bl |     lowlevel_init /* go setup pll,mux,memory */
289

```

调用 x210 目录下的 lowlevel_init.S 文件

```
/uboot/board/samsung/x210# vi lowlevel_init.S
```

```

40 |         .globl lowlevel_init
41 lowlevel_init:
42 |         push | {lr} ← 现将 lr 压栈
43 |

```

Makefile 将 DDR 的起始地址传给 _TEXT_BASE

```

106 |     /* when we already run in ram, we don't need to relocate U-Boot.
107 |     * and actually, memory controller must be configured before U-Boot
108 |     * is running in ram.
109 |     */
110 |     ldr |     r0, =0xff000fff
111 |     bic |     r1, pc, r0 | /* r0 <- current base addr of code */
112 |     ldr |     r2, _TEXT_BASE | /* r1 <- original base addr in ram */
113 |     bic |     r2, r2, r0 | /* r0 <- current base addr of code */
114 |     cmp |     r1, r2 | /* compare r0, r1 */
115 |     beq |     1f | /* r0 == r1 then skip sdram init */
116 |
117 |     /* init system clock */
118 |     bl |     system_clock_init
119 |
120 |     /* Memory initialize */
121 |     bl |     mem_ctrl_asm_init
122 |
123 |     1:
124 |     /* for UART */
125 |     bl |     uart_asm_init
126 |
127 |     bl |     tzpc_init
128

```

```

205     system_clock_init:
206
207         ldr|    r0, =ELFIN_CLOCK_POWER_BASE| @0xe0100000
208
209         /* Set Mux to FIN */
210         ldr|    r1, =0x0
211         str|    r1, [r0, #CLK_SRC0_OFFSET]
212
213         ldr|    r1, | =APLL_LOCKTIME_VAL
214         str|    r1, | [r0, #APLL_LOCK_OFFSET]
215
216         /*****lxg added*****
217         ldr|    r0, =ELFIN_CLOCK_POWER_BASE| @0xe0100000
218
219         ldr|    r1, | =MPLL_LOCKTIME_VAL
220         str|    r1, | [r0, #MPLL_LOCK_OFFSET]
221         /*****end*****/

```

这个初始化时钟的代码从 205 行~305 行都是，

3.7.1 REGISTER MAP

Register	Address	R/W	Description
APLL_LOCK	0xE010_0000	R/W	Control PLL locking period for APLL.
Reserved	0xE010_0004	-	Reserved
MPLL_LOCK	0xE010_0008	R/W	Control PLL locking period for MPLL.
Reserved	0xE010_000C	-	Reserved
CLK_SRC0	0xE010_0200	R/W	Select clock source 0 (Main)

一直到 305 行都是差不多的设置，照着手册对比就是

→ 内存初始化文件在 uboot/cpu/s5pc11x/s5pc110/cpu_init.S 里面

```

4 | .globl mem_ctrl_asm_init
5 mem_ctrl_asm_init:
6
7 #ifndef CONFIG_EVT1
8
9 |     ldr    r0, =ASYNC_MSYS_DMC0_BASE
10
11|     ldr    r1, =0x0
12|     str    r1, [r0, #0x0]
13
14|     /* This register is removed at EVT1 of C110. */
15|     ldr    r1, =0x0
16|     str    r1, [r0, #0xC]
17
18#endif
19
20#endif CONFIG_MCP_SINGLE
21
22/* DMC0 Drive Strength (Setting 2X) */
23
24|     ldr    r0, =ELFIN_GPIO_BASE
25
26|     ldr    r1, =0x0000AAAA
27|     str    r1, [r0, #MP1_ODRV_SR_OFFSET]
28
29|     ldr    r1, =0x0000AAAA
30|     str    r1, [r0, #MP1_ODRV_SR_OFFSET]
121|     bl    mem_ctrl_asm_init
122
123 1:
124|     /* for UART */
125|     bl    uart_asm_init
126
127|     bl    tzpc_init
128
129#endif CONFIG_ONENAND
130|     bl    onenandcon_init

```

这就是内存初始化开始位置

这些宏的值也是寄存器地址，所以这些宏定义在 x210_sd.h，

我们 DDR 初始化完成之后，再次回到 uboot/board/samsung/x210/lowlevel_init.S 文件接着 mem_ctrl_asm_init 之后，初始化串口

串口初始化成功之后，终端会打印 OK

lowlevel_init.S 分析到串口初始化就暂时完结了，后面的代码暂时不分析，现在回到 uboot/cpu/s5pc11x/start.S 接着 lowlevel_init 下面继续分析

```
288 |     bl    lowlevel_init /* go setup pll,mux,memory */
289 |     /* To hold max8698 output before releasing power on switch,
290 |      * set PS_HOLD signal to high
291 |      */
292 |     ldr   r0, =0xE010E81C /* PS_HOLD_CONTROL register */
293 |     ldr   r1, =0x00005301 /* PS HOLD output high*/
294 |     str   r1, [r0]
295 |
296 |     /* get ready to call C functions */
297 |     ldr   sp, _TEXT_PHY_BASE /* setup temp stack pointer */
298 |     sub   sp, sp, #12
299 |     mov   fp, #0 |           /* no previous frame, so fp=0 */
300 |
301 |     /* when we already run in ram, we don't need to relocate U-Boot.
302 |      * and actually, memory controller must be configured before U-Boot.
```

以后 uboot 就加载到 ddr 的 0x33e00000 地址上

```
107 | .text _TEXT_PHY_BASE:
108 | .word  CFG_PHY_UBOOT_BASE
```

这个 _TEXT_PHY_BASE 就在 start.S 的 107 行

.word 指针指向 CFG_PHY_UBOOT_BASE 宏

这个 CFG_PHY_UBOOT_BASE 宏在 uboot/include/configs/x210_sd.h 里

```
00557: #define CFG_PHY_UBOOT_BASE MEMORY_BASE_ADDRESS + 0x3e00000
00558: #define CFG_PHY_KERNEL_BASE MEMORY_BASE_ADDRESS + 0x8000
```

这时候外部 ddr 已经初始化完成，所以要把 CPU 内部 SRAM 分配的栈，移植到 DDR 中去，在 DDR 中设置这个栈，这个 _TEXT_PHY_BASE 里面的地址就是 uboot 加载到 ddr 的基地址，这里是 33e00000

```
305 |     ldr   r0, =0xff000fff
306 |     bic   r1, pc, r0 |           /* r0 <- current base addr of code */
307 |     ldr   r2, _TEXT_BASE |       /* r1 <- original base addr in ram */
308 |     bic   r2, r2, r0 |           /* r0 <- current base addr of code */
309 |     cmp   r1, r2 |             /* compare r0, r1
310 |     beq   after_copy |         /* r0 == r1 then skip flash copy
```

对比地址决定 uboot 要不要重定位

```
312 #if defined(CONFIG_EVT1)
313 |     /* If BL1 was copied from SD/MMC CH2 */
314 |     ldr   r0, =0xD0037488
315 |     ldr   r1, [r0]
316 |     ldr   r2, =0xEB200000
317 |     cmp   r1, r2
318 |     beq   mmcisd_boot
```

Uboot 第 1 部分从 SD 卡那个通道启动

比如从 sd0 通道启动，那么 r2 地址为 EB000000,如果从 SD2 通道启动 r2 地址为 EB200000

如果 r1 不等于 r2 就跳过 mmcisd_boot 执行下面的代码

如果相等就是从 SD2 通道启动的，就是外部 SD 卡，如果从外部的 SD 卡启动，我们就执行 mmcisd_boot 段代码

```

321 |     ldr    r0, =INF_REG_BASE
322 |     ldr    r1, [r0, #INF_REG3_OFFSET]
323 |     cmp    r1, #BOOT_NAND|          /* 0x0 => boot device is nand */
324 |     beq    nand_boot
325 |     cmp    r1, #BOOT_ONENAND|        /* 0x1 => boot device is onenand */
326 |     beq    onenand_boot
327 |     cmp    r1, #BOOT_MMCSD
328 |     beq    mmcisd_boot
329 |     cmp    r1, #BOOT_NOR
330 |     beq    nor_boot
331 |     cmp    r1, #BOOT_SEC_DEV
332 |     beq    mmcisd_boot

```

取出 INF_REG3_OFFSET 寄存器里面的值,然后和下面的#BOOT_MMCSD 比较,如果相同就再次确认是从 SD 卡启动,执行 mmcisd_boot

```

343 mmcisd_boot:
344 #if DELETE
345 |     ldr    sp, _TEXT_PHY_BASE
346 |     sub    sp, sp, #12
347 |     mov    fp, #0
348#endif
349 |     bl     movi_b12_copy
350 |     b      after_copy
351

```

这段代码删除了

真正执行的代码就是
movi_b12_copy

在 uboot/cpu/s5pc11x/movi.c 文件中

```

00014: extern raw_area_t raw_area_control;
00015:
00016: typedef u32(*copy_sd_mmc_to_mem)
00017: (32 channel, 32 start_block, 16 block_size, 32 *trg, 32
00018:
00019: void movi_b12_copy(void)
00020: {
00021:     ulong ch;
00022: #if defined(CONFIG_EVT1)
00023:     ch = *(volatile u32 *)0xD0037488;
00024:     copy_sd_mmc_to_mem copy_b12 =
00025:         (copy_sd_mmc_to_mem) (*(u32 *) (0xD0037F98));
00026:
00027: #if defined(CONFIG_SECURE_BOOT)
00028:     ulong rv;
00029: #endif
00030: #else
00031:     ch = *(volatile u32 *)0xD003A508;
00032:     copy_sd_mmc_to_mem copy_b12 =
00033:         (copy_sd_mmc_to_mem) (*(u32 *) (0xD003E008));
00034:     u32 ret;
00035:     if (ch == 0xEB000000) {
00036:         ret = copy_b12(0, MOVI_BL2_POS, MOVI_BL2_BLKCNT,
00037:             CFG_PHY_UBOOT_BASE, 0);
00038:     }
00039: #if defined(CONFIG_SECURE_BOOT)
00040:     /* do security check */
00041:     rv = Check_Signature((SecureBoot_CTX *)SECURE_BOOT_CONTEXT,
00042:                          (unsigned char *)CFG_PHY_UBOOT_BASE, (1024*512),
00043:                          (unsigned char *)CFG_PHY_UBOOT_BASE+(1024*512));
00044:     if (rv != 0){
00045:         while(1);
00046:     }
00047: #endif
00048:     else if (ch == 0xEB200000) {
00049:         ret = copy_b12(2, MOVI_BL2_POS, MOVI_BL2_BLKCNT,
00050:             CFG_PHY_UBOOT_BASE, 0);
00051:     }
00052: #if defined(CONFIG_SECURE_BOOT)

```

读取 0xD0037488 里面的数据决定是 SD 卡启动还是什么启动

判断内部 INAND 启动

还是外部 SD 卡启动

```

348 #endif
349 |     bl      movi_bl2_copy
350 |     b       after_copy

```

Uboot 之虚拟地址映射

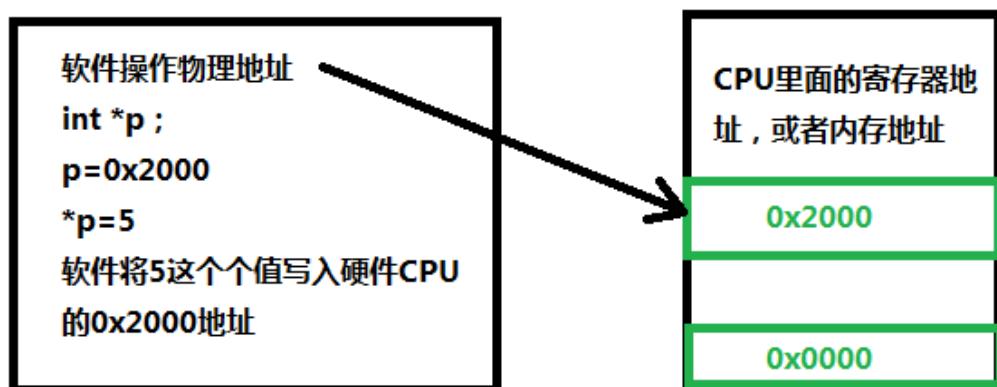
```

357 after_copy:
358
359 #if defined(CONFIG_ENABLE_MMU)
360 enable_mmu:
361     /* enable domain access */
362     ldr| r5, =0x0000ffff
363     mcr| p15, 0, r5, c3, c0, 0| @load domain access register
364
365     /* Set the TTB register */
366     ldr| r0, _mmu_table_base
367     ldr| r1, =CFG_PHY_UBOOT_BASE
368     ldr| r2, =0xffff00000
369     bic| r0, r0, r2
370     orr| r1, r0, r1
371     mcr| p15, 0, r1, c2, c0, 0
372
373     /* Enable the MMU */

```

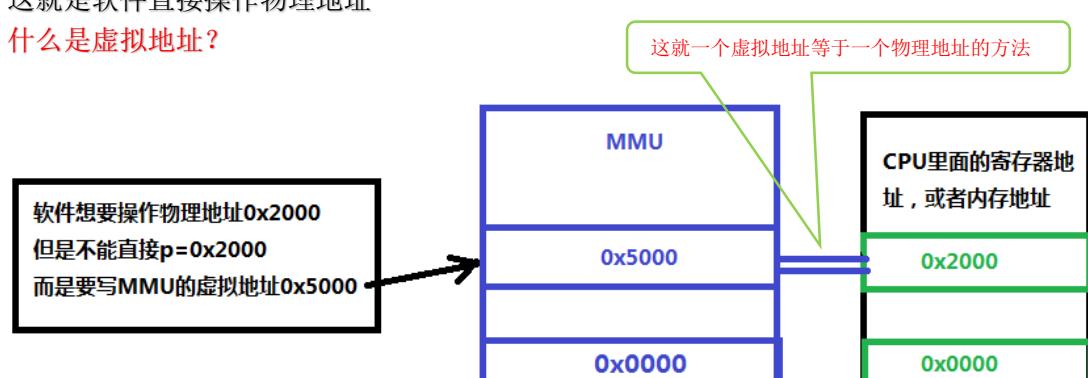
什么是物理地址？

CPU 的寄存器地址就是物理地址，每个寄存器的地址值是不能软件修改的。



这就是软件直接操作物理地址

什么是虚拟地址？



所以我们要建立一个映射表，来映射虚拟地址和物理地址的对应关系

用硬件的 mmu 单元来建立虚拟物理映射才行，操作 CP15 协处理器。

地址映射好处 1：可以对内存进行分块，然后对分块的每一块单独映射。

地址映射好处 2：把 DDR 常用的数据放入 cache，CPU 先去 cache 里面找数据，如果 cache

里面没有 CPU 需要的数据，才去 DDR 里面找，cache 读写速度比 DDR 快 N 倍

进入 uboot/cpu/s5pc11x/start.S

```
357 after_copy:  
358  
359 #if defined(CONFIG_ENABLE_MMU)  
360 enable_mmub:  
361     /* enable domain access */  
362     ldr|    r5, =0x0000ffff  
363     mcr|    p15, 0, r5, c3, c0, 0|      @load domain access register  
364  
365     mcr 用于  
366     cp15 协  
367     处理器里  
368     面的寄存  
369     器  
370  
371     /* Set the TTB register */  
372     ldr|    r0, _mmu_table_base  
373     ldr|    r1, =CFG_PHY_UBOOT_BASE  
374     ldr|    r2, =0xffff0000  
375     bic|    r0, r0, r2  
376     orr|    r1, r0, r1  
377     mcr|    p15, 0, r1, c2, c0, 0  
378  
379     /* Enable the MMU */
```

访问 CP15 寄存器的指令

访问 CP15 寄存器指令的编码格式及语法说明如下：

31 28	27 24	23 21	2	19 16	15 12	11 8	7 5	4	3 0
cond	1 1 1 0	opcode	L	cr n	rd	1 1 1 1	opcode	1	crm

<opcode_1>：协处理器行为操作码，对于 CP15 来说，<opcode_1>永远为 0b000，否则结果未知。

<rd>：不能是 r15/pc，否则，结果未知。

<crn>：作为目标寄存器的协处理器寄存器，编号为 C0~C15。

<crm>：附加的目标寄存器或源操作数寄存器，如果不设置附加信息，将 crm 设置为 c0，否则结果未知。

<opcode_2>：提供附加信息比如寄存器的版本号或者访问类型，用于区分同一个编号的不同物理寄存器，可以省略<opcode_2>或者将其设置为 0，否则结果未知。

0的二进制

Mcr 指令使用：

下面的指令从 ARM 寄存器 R4 中将数据传送到协处理器 CP15 的寄存器 C1 中。其中 R4 为 ARM 寄存器，存放源操作数，C1、C0 为协处理器寄存器，为目标寄存器，opcode_1 为 0，opcode_2 为 0。

MCR p15, 0, R4, C1, C0, 0

mcr| p15, 0, r5, c3, c0, 0|

4.1.2 CP15 寄存器介绍

CP15 的寄存器列表如表 4-1 所示。

表 4-1 ARM 处理器中 CP15 协处理器的寄存器

我们 CPU 执行这句意思就是：将 ARM 寄存器 R4 中取出的数据写入 CP15 寄存器 C3 中

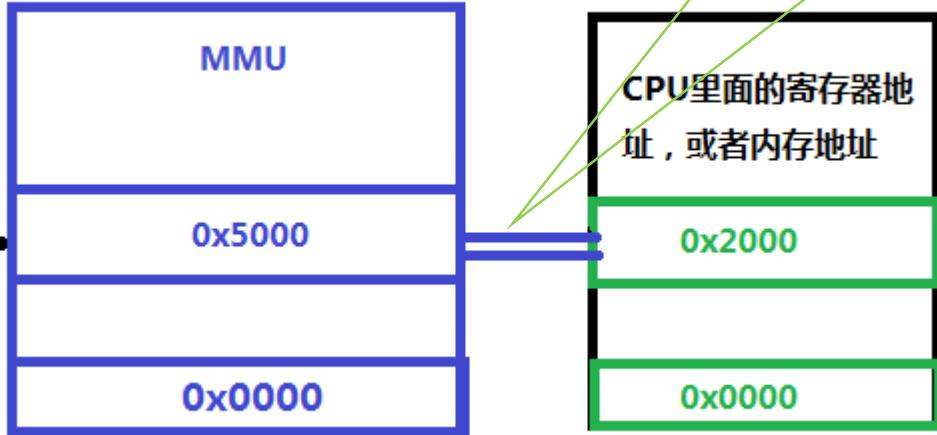
寄存器编号	基本作用	在 MMU 中的作用	在 PU 中的作用
0	ID 编码（只读）	ID 编码和 cache 类型	
1	控制位（可读写）	各种控制位	
2	存储保护和控制	地址转换表基址	Cachability 的控制位
3	存储保护和控制	域访问控制位	Bufferability 控制位
4	暂无具体功能	暂无具体功能	暂无具体功能

CP15 的 C3 寄存器就是域访问控制位

```

356
357 after_copy:
358
359 #if defined(CONFIG_ENABLE_MMU)
360 enable_mmu:
361     /* enable domain access */
362     ldr|    r5, =0x0000ffff
363     mcr|    p15, 0, r5, c3, c0, 0| |      @load domain access register
364
365     /* Set the TTB register */
366     ldr|    r0, _mmu_table_base
367     ldr|    r1, =CFG_PHY_UBOOT_BASE
368     ldr|    r2, =0xffff00000
369     bic|    r0, r0, r2
370     orr|    r1, r0, r1
371     mcr|    p15, 0, r1, c2, c0, 0
372
373     /* Enable the MMU */

```



转换表最容易理解的解释是：

数字代表表项	字母代表表项里面的值
1	a
2	b
3	c
.	.
.	.
255	z

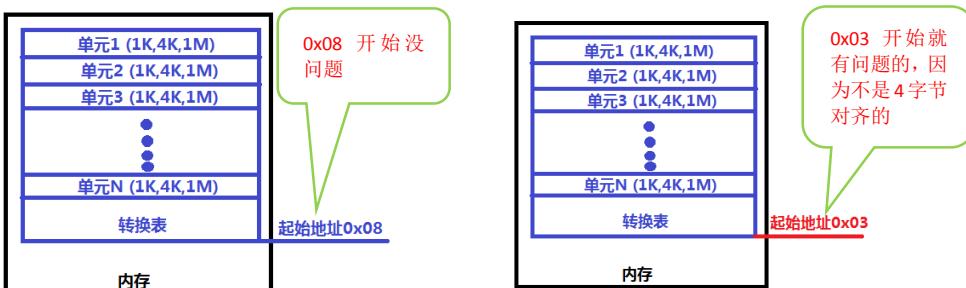
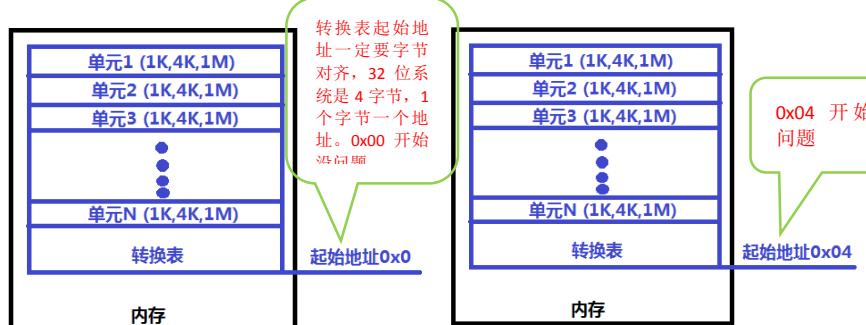
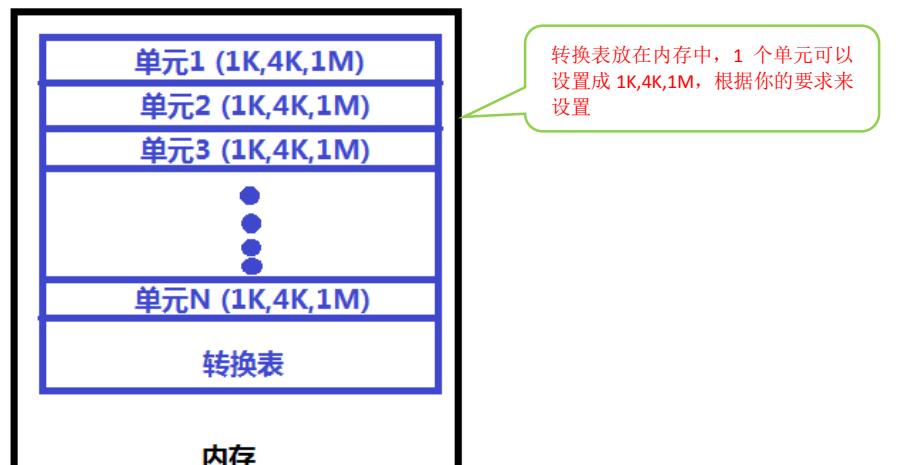
表项 1, 2, 3.....代表虚拟地址, a, b, c....代表物理地址, 每个表项存放一个物理地址

我们建立好这个转换表之后，我们去操作虚拟地址，MMU 会自动帮我们把虚拟地址转换到物理地址去，所以你实际就是在操作物理地址。

ARM 支持 3 中表大小，1KB(细表)，4KB(粗表)，1MB(段式表)

我们一般用段式表。

真正的转换表，由很多个单元构成的，每个单元负责一个内存块，所以是很多内存块构成，所以转换表在 32 位系统下是哦 0~4G 的范围



所以 32 位系统转换表是 4 字节对齐

```

357 after_copy:
358
359 #if defined(CONFIG_ENABLE_MMU)
360 enable_mmu:
361     /* enable domain access */
362     ldr |    r5, =0x0000ffff
363     mcr |    p15, 0, r5, c3, c0, 0|    @load domain access register
364
365     /* Set the TTB register */
366     ldr |    r0, _mmu_table_base
367     ldr |    r1, =CFG_PHY_UBOOT_BASE
368     ldr |    r2, =0xffff00000
369     bic |    r0, r0, r2
370     orr |    r1, r0, r1
371     mcr |    p15, 0, r1, c2, c0, 0
372
373     /* Enable the MMU */

```

```

417 _mmu_table_base:
418     .word mmu_table
419 #endif

```

这个就是转换表基址，
_mmu_table_base 就是
mmu_table

mmu_table 在 uboot/board/samsung/x210/lowlevel_init.S

```
593 mmu_table:  
594     .set __base,0  
595     // Access for iRAM  
596     .rept 0x100  
597     FL_SECTION_ENTRY __base,3,0,0,0  
598     .set __base,__base+1  
599     .endr  
600  
601     // Not Allowed  
602     .rept 0x200 - 0x100  
603     .word 0x00000000  
604     .endr  
605  
606     .set __base,0x200  
607     // should be accessed  
608     .rept 0x600 - 0x200  
609     FL_SECTION_ENTRY __base,3,0,1,1  
610     .set __base,__base+1  
611     .endr  
612  
613     .rept 0x800 - 0x600  
614     .word 0x00000000  
615     .endr  
616  
617     .set __base,0x800  
618     // should be accessed  
619     .rept 0xb00 - 0x800  
620     FL_SECTION_ENTRY __base,3,0,0,0  
621     .set __base,__base+1  
622     .endr  
623  
584     /* form a first-level section entry */  
585 .macro FL_SECTION_ENTRY base,ap,d,c,b  
586     .word (\base << 20) | (\ap << 10) | \  
587     (\d << 5) | (1<<4) | (\c << 3) | (\b << 2) | (1<<1)  
588 .endm  
589 section_mmudata ".n"
```

意思是 0x100 就是十进制的 256，我循环建立 256 个单元

这个 FL_SECTION_ENTRY 是宏

.rept 和 endr 是一个循环

.rept cnt //cnt 表示循环次数 执行的汇编内容.....

.endr //endr 循环结束

例如: .rept 100
 i++
 endr
以上就是 i 加 100 次

这个.word 后面括号计算出来的最终数值就代表一个表项

.macro 是用来定义宏的，类似 C 语言的#define

.endm 是用来终止这个宏，和 C 语言不一样，C 语言宏不需要终止符号。

FL_SECTION_ENTRY 这个是宏名，自己定义的

FL_SECTION_ENTRY 后面跟的 base, ap, d, c, b 是宏的 5 个参数变量，类似 C 语言的宏变量加参数#define (a,b) a+b

.word 定义 4 字节 unsigned int 的数

```

593 mmu_table:
594     .set __base, 0
595     // Access for iRAM
596     .rept 0x100
597     FL_SECTION_ENTRY __base, 3, 0, 0, 0
598     .set __base, __base+1
599     .endr
600
601     // Not Allowed
602     .rept 0x200 - 0x100
603     .word 0x00000000
604     .endr
605
606     .set __base, 0x200
607     // should be accessed
608     .rept 0x600 - 0x200
609     FL_SECTION_ENTRY __base, 3, 0, 1, 1
610     .set __base, __base+1
611     .endr
612
613     .rept 0x800 - 0x600
614     .word 0x00000000
615     .endr
616
617     .set __base, 0x800
618     // should be accessed
619     .rept 0xb00 - 0x800
620     FL_SECTION_ENTRY __base, 3, 0, 0, 0
621     .set __base, __base+1
622     .endr
623

```

1. 所以每循环一次就建立一个表项

2. 最先 base 值设置为 0

3. base=0 时，建立了一个表项 0

4. base+1 等于 base=1

5. 第二次循环，base=1 时建立表项 1，以此类推

6. 最后出来这么一张表，因为我们 ARM 是按照段式映射来设置的，前面说了 1 段代表 1M 大小，也就是一个单元(表项)，所以这个表是 256MB 大小的映射单元

00000000	base=0
00100000	base=1
00200000	base=2
...	
10000000	base=256

所以我们把 0~4G 的映射表列出来。

```

593 mmu_table:
594     .set __base,0
595     // Access for iRAM
596     .rept 0x100
597     FL_SECTION_ENTRY __base,3,0,0,0
598     .set __base,__base+1
599     .endr
600
601     // Not Allowed
602     .rept 0x200 - 0x100
603     .word 0x00000000
604     .endr
605
606     .set __base,0x200
607     // should be accessed
608     .rept 0x600 - 0x200
609     FL_SECTION_ENTRY __base,3,0,1,1
610     .set __base,__base+1
611     .endr
612
613     .rept 0x800 - 0x600
614     .word 0x00000000
615     .endr
616
617     .set __base,0x800
618     // should be accessed
619     .rept 0xb00 - 0x800
620     FL_SECTION_ENTRY __base,3,0,0,0
621     .set __base,__base+1
622     .endr
623
624 /* */
625     .rept 0xc00 - 0xb00
626     .word 0x00000000
627     .endr */
628     .set __base,0xB00
629     .rept 0xc00 - 0xb00
630     FL_SECTION_ENTRY __base,3,0,0,0
631     .set __base,__base+1
632     .endr
633
634     // 0xC00_0000映射到0x2000_0000
635     .set __base,0x300
636     // set __base,0x200
637     // 256MB for SDRAM with cacheable
638     .rept 0xD00 - 0xC00
639     FL_SECTION_ENTRY __base,3,0,1,1
640     .set __base,__base+1
641     .endr
642
643     // access is not allowed.
644     @.rept 0xD00 - 0xC80
645     @.word 0x00000000
646     @.endr
647
648     .set __base,0xD00
649     // 1:1 mapping for debugging with non-cacheable
650     .rept 0x1000 - 0xD00
651     FL_SECTION_ENTRY __base,3,0,0,0
652     .set __base,__base+1
653     .endr
654
655 #else // CONFIG_MCP_AC, CONFIG_MCP_H, CONFIG_MCP_B

```

总结一下内存范围

VA虚拟地址	PA物理地址
00000000	00000000
00000001	00000001
00000010	00000010
	length=256M
10000000	10000000

VA虚拟地址	PA物理地址
10000000	0
20000000	0

VA虚拟地址	PA物理地址
20000000	20000000
60000000	60000000

VA虚拟地址	PA物理地址
60000000	0
80000000	0

VA虚拟地址	PA物理地址
80000000	80000000
b0000000	b0000000

VA虚拟地址	PA物理地址
b0000000	b0000000
c0000000	c0000000

VA虚拟地址	PA物理地址
c0000000	30000000
d0000000	40000000

只有这块虚拟地址和物理地址映射不一样，这块也是重点区域

总结一下 uboot 虚拟地址映射内存范围表

VA	PA	长度	占用范围
0~10000000	0~10000000	256M	//虚拟地址物理地址没变可用
10000000~20000000	0	256M	//这段虚拟地址不可用
20000000~60000000	20000000~60000000	1G	512M~1.5G //可用
60000000~80000000	0	512M	1.5G~2G //不可用
80000000~b0000000	80000000~b0000000	768M	2G~2.75G //可用
b0000000~c0000000	b0000000~c0000000	256M	2.75G~3G //可用
c0000000~d0000000	30000000~40000000	256M	3G~3.25G //这里虚拟地址映射
不一样，因为这段就是匹配 DDR 的地址段			
d-完	d-完	768M	3.25G~4G

S5PV210 的 DRAM 控制器有两个：

DMC0: 0x30000000~0x3FFFFFF //正好是 30000000~40000000 这一段，这一段就是给 DDR 用的
DMC1: 0x40000000~0x4FFFFFF

总结：虚拟地址映射就是把虚拟地址的 c0000000 开头的 256MB 映射给了 DMC0 的 0x30000000 的 256MB，也就是 DDR。其余地址都是一一对应的，所以重点关注 DMC0 这一段。

```
2588  
2589 x210_sd_config :|      unconfig  
2590 |      @$(MKCONFIG) $(@:_config=) arm s5pc11x x210 samsung s5pc110  
2591 |      @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/x210/config.mk  
2592
```

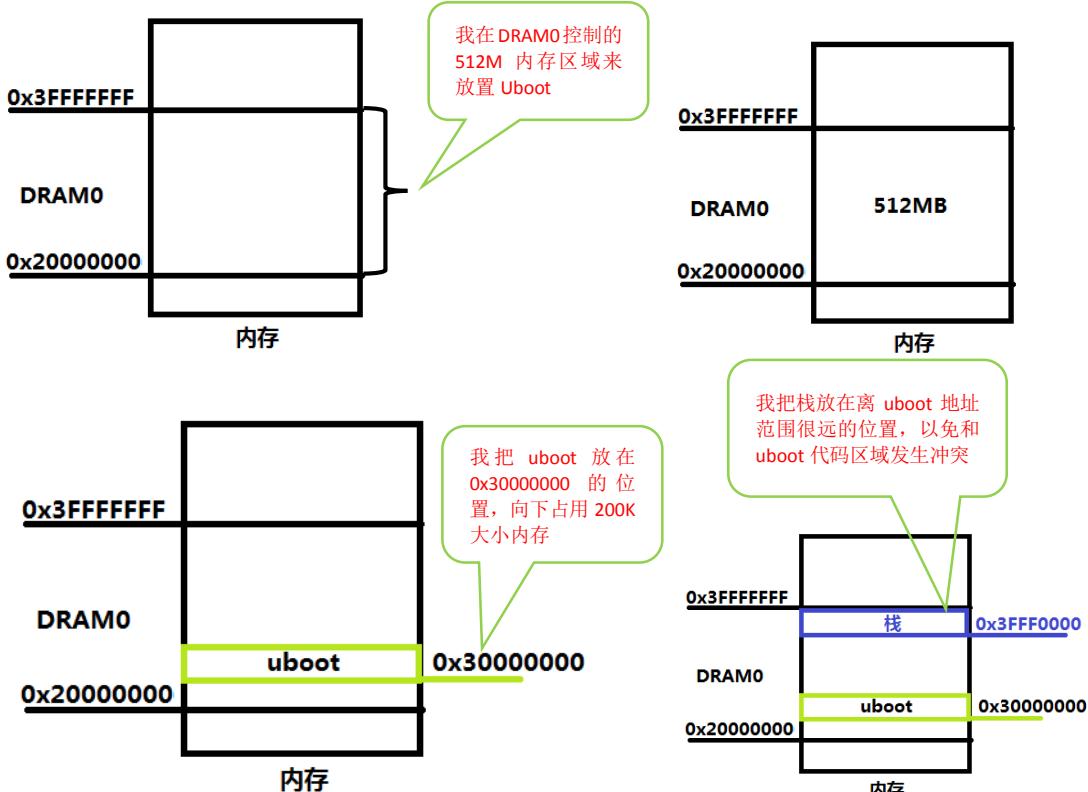
现在又回到/u-boot/cpu/s5pc11x/start.S 文件

```
360 enable_mmu:
361     /* enable domain access */
362     ldr| r5, =0x000fffff
363     mcr| p15, 0, r5, c3, c0, 0 |      @load domain access register
364
365     /* Set the TTB register */
366     ldr| r0, _mmu_table_base
367     ldr| r1, =CFG_PHY_UBOOT_BASE
368     ldr| r2, =0xffff0000
369     bic| r0, r0, r2
370     orr| r1, r0, r1
371     mcr| p15, 0, r1, c2, c0, 0
372
373     /* Enable the MMU */
374 mmu_on:
375     mrc| p15, 0, r0, c1, c0, 0
376     orr| r0, r0, #1
377     mcr| p15, 0, r0, c1, c0, 0
378     nop
379     nop
380     nop
381     nop
382
383 skip_hw_init:
384     /* Set up the stack| | | | |
385
386 stack_setup:
387 #if defined(CONFIG_MEMORY_UPPER_CODE)
388     ldr| sp, =(CFG_UBOOT_BASE + CFG_UBOOT_SIZE - 64)
389 #else
390     ldr| r0, _TEXT_BASE| |           /* upper 128 KiB
391     sub| r0, r0, #CFG_MALLOC_LEN/* malloc area
392     sub| r0, r0, #CFG_GBL_DATA_SIZE /* bdinfo
393 #if defined(CONFIG_USE_IRQ)
394     sub| r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STA
395 #endif
396     sub| sp, r0, #12| |           /* leave 3 words
397
398 #endif
399
```

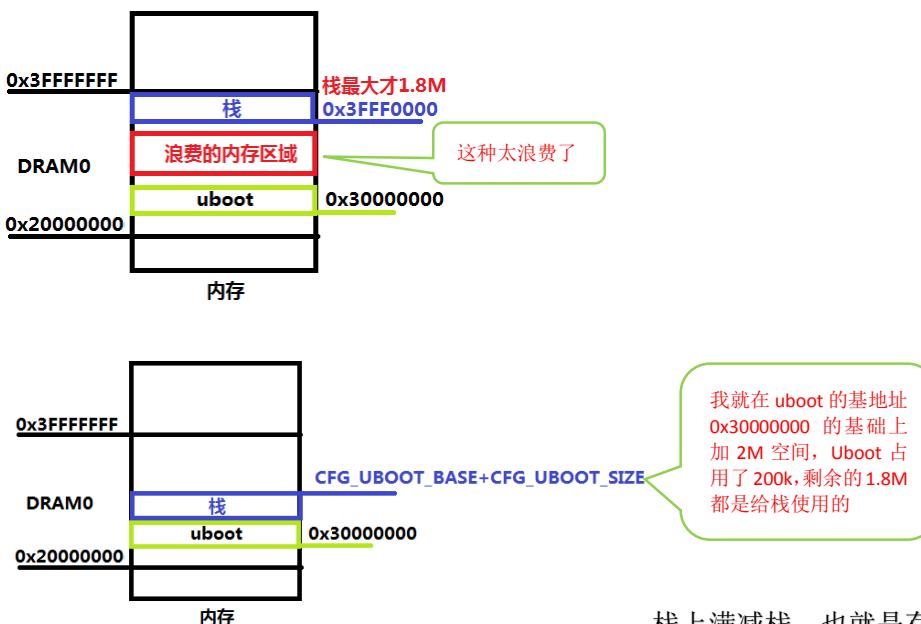
`CFG_UBOOT_BASE = 0x33e00000 //这是 Uboot 的地址, 但是我设置的栈空间不能把 Uboot 代码区域覆盖了`

所以 `CFG_UBOOT_BASE + CFG_UBOOT_SIZE` (`CFG_UBOOT_SIZE` 是 2M), 因为 Uboot 最大才 200 多 KB, 所以还有 1.848M 可以给栈用。

这次设置栈是让 Uboot 和栈的位置相对安全, 怎么理解呢?



这种把栈放在离 uboot 代码区域很远的位置, 没有问题。只是栈和 Uboot 之间的区域就不能被别人使用。但是这个区域很大, 你看我 uboot 和栈都在 512M 内存的下边和上边, 但是 uboot 和栈用不到这么多区域, 那么中间的区域是否被浪费了?



栈上满减栈, 也就是存一个数据栈地址向下减, 但是你不可能存 1.8M 的零时数据散, 所以不会冲掉 uboot 区域。

栈已经设置完毕，下面来清除 bss 段

```
400 clear_bss:  
401 | ldr | r0, _bss_start | /* find start of bss segment */  
402 | ldr | r1, _bss_end | /* stop here */  
403 | mov | r2, #0x00000000 | /* clear */  
404
```

再次清理 bss 段代码，操作方法和裸机逻辑一样，看前面资料

/* find start of bss segment */

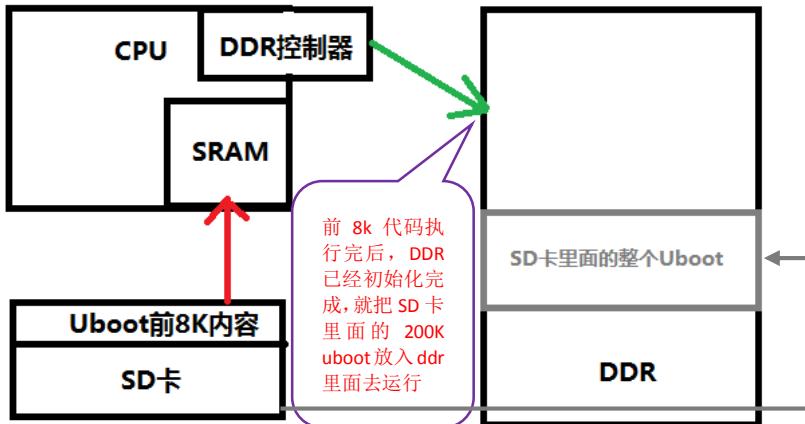
/* stop here */

/* clear */

```
410  
411 | ldr | pc, _start_armboot  
412
```

执行这一句进入 uboot
第二阶段，这句话的意思
就是 uboot 第二阶段 C 语
言的函数地址传递给
PC，实现长跳转

这个长跳转可以从 CPU 的 SRAM 初始化 uboot 第一阶段，跳转到 DDR 执行 uboot 初始化第二阶段



这样长跳转之后，虽然 DDR 里面有 uboot 第一阶段和第二阶段的代码，但第一阶段永远得不到运行，直接从第二阶段开始运行，因为第一阶段在 SRAM 里面已经运行过了。

上面的 uboot 汇编代码初始化的就是些看门狗，时钟，DDR 控制器，并且完成重定位(把 uboot 复制到 DDR 去)。然后我们下面要做的是在 DDR 上用 C 语言初始化其他部分的外设，比如 INAND，网卡，串口，uboot 命令，uboot 环境变量。

进入 uboot/lib_arm/board.c

```

444 void start_armboot (void)
445 {
446     init_fnc_t **init_fnc_ptr;
447     char *s;
448     int mmc_exist = 0;
449 #if !defined(CONFIG_NO_FLASH) || defined(CONFIG_VFD) || defined(CONFIG_LCD)
450     ulong size;
451 #endif
452
453 #if defined(CONFIG_VFD) || defined(CONFIG_LCD)
454     unsigned long addr;
455 #endif
456
457 #if defined(CONFIG_BOOT_MOVINAND)
458     uint *magic = (uint *) (PHYS_SDRAM_1);
459 #endif
460
461     /* Pointer is writable since we allocated a register for it */
462 #ifdef CONFIG_MEMORY_UPPER_CODE /* by scsuh */
463     ulong gd_base;
464
465     gd_base = CFG_UBOOT_BASE + CFG_UBOOT_SIZE - CFG_MALLOC_LEN - CFG_STACK_SIZE - sizeof(gd_t);
466 #ifdef CONFIG_USE_IRQ
467     gd_base -= (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ);
468 #endif
469     gd = (gd_t*)gd_base;
470 #else
471     gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
472 #endif
473

```

Start.S 直接跳转到这里

410 411 ldr pc, _start_armboot

这是一个二级函数指针

412 typedef int (init_fnc_t) (void);

这是函数类型

什么是函数类型?, 看我的 C 语言文档

这个 gd 是个全局变量

70 DECLARE_GLOBAL_DATA_PTR;

DECLARE_GLOBAL_DATA_PTR 这个宏在 uboot/include/Asm-arm/Global_data.h 下面定义的

00067: #define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r8")

用 volatile 修饰表示编译器不要优化, 用 register 修饰, 是要求 gd 这个变量不要放在全局区, 而是放在寄存器 r8 上, 但是也同时拥有全局变量功能。提高访问速度, 因为代码很多地方都会用这个全局变量

uboot/include/Asm-arm/Global_data.h 定义了 gd 变量的类型叫做 gd_t

```

00036: typedef struct global_data {
00037:     bd_t      *bd;           存放串口波特率
00038:     unsigned long flags;    这是布尔的表示, 串
00039:     unsigned long baudrate; 口控制台有没有
00040:     unsigned long have_console; /* serial_init() was called */
00041:     unsigned long reloc_off;  /* Relocation Offset */
00042:     unsigned long env_addr;   /* Address of Environment struct */
00043:     unsigned long env_valid;  /* Checksum of Environment valid? */
00044:     unsigned long fb_base;    /* base address of frame buffer */
00045: #ifdef CONFIG_VFD
00046:     unsigned char vfd_type;  /* display type */
00047: #endif
00048: #if 0
00049:     unsigned long cpu_clk;   /* CPU clock in Hz! */
00050:     unsigned long bus_clk;
00051:     phys_size_t ram_size;   /* RAM size */
00052:     unsigned long reset_status; /* reset status register at boot */
00053: #endif
00054:     void        **jt;        /* jump table */
00055: } ? end global_data ? gd_t;

```

在内存中的环境变量是否可以使用。这是因为环境变量是在 SD 卡里面的, 我们在使用环境变量时, 是使用的 SD 卡拷贝到内存的环境变量, 这里就是判断 SD 卡拷贝环境变量到内存是否完成

uboot/include/Asm-arm/U-boot.h

```

00039: typedef struct bd_info {
00040:     int       bi_baudrate; /* serial console baudrate */
00041:     unsigned long bi_ip_addr; /* IP Address */
00042:     unsigned char bi_enetaddr[6]; /* Ethernet address */
00043:     struct environment_s *bi_env;
00044:     ulong    bi_arch_number; /* unique id for this board */
00045:     ulong    bi_boot_params; /* where this board expects params */
00046:     struct    /* RAM configuration */
00047:     {
00048:         ulong start;
00049:         ulong size;
00050:     } bi_dram[CONFIG_NR_DRAM_BANKS];
00051: #ifdef CONFIG_HAS_ETH1
00052:     /* second onboard ethernet port */
00053:     unsigned char bi_enet1addr[6];
00054: #endif
00055: } bd_t;

```

这里和 gd_t 的串口比特率一致

板子网卡 IP 地址

板子网卡 MAC 地址

板子的全球唯一 ID

```

444 void start_armboot (void)
445 {
446     init_fnc_t **init_fnc_ptr;
447     char *s;
448     int mmc_exist = 0;
449 #if !defined(CONFIG_NO_FLASH) || defined(CONFIG_MMC)
450     ulong size;
451 #endif
452 #if defined(CONFIG_VFD) || defined(CONFIG_LCD)
453     unsigned long addr;
454 #endif
455 #endif
456
457 #if defined(CONFIG_BOOT_MOVINAND)
458     uint *magic = (uint *) (PHYS_SDR
459 #endif
460
461 /* Pointer is writable since we allocated a register for it */
462 #ifndef CONFIG_MEMORY_UPPER_CODE /* by scsuh */
463     ulong gd_base;
464
465     gd_base = CFG_UBOOT_BASE + CFG_UBOOT_SIZE - CFG_MALLOC_LEN - CFG_STACK_SIZE - sizeof(gd_t);
466 #ifndef CONFIG_USE_IRQ
467     gd_base -= (CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ);
468 #endif
469     gd = (gd_t*)gd_base; // 把分配好的 gd 首地址赋值给 gd 指针
470 #else
471     gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
472 #endif

```

存放Uboot的第一个地址是从CFG_UBOOT_BASE + CFG_UBOOT_SIZE - CFG_MALLOC_LEN -CFG_STACK_SIZE- gd_t大小
(uboot预留2M大小) (uboot要使用的堆区) (uboot要使用的栈区)
但是uboot只占用其中
200K

```

473
474     /* compiler optimization barrier needed for GCC >= 3.4 */
475     __asm__ __volatile__(": : :memory");
476
477     memset ((void*)gd, 0, sizeof (gd_t));
478     gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
479     memset (gd->bd, 0, sizeof (bd_t));
480
481     monitor_flash_len = _bss_start - _armboot_start;
482
483     for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
484         if ((*init_fnc_ptr)() != 0) {
485             hang ();
486         }
487     }

```

gd 分配后怕内存不干净，这里先清 0

这里为什么是 char*, 这和指针到底是-1, 还是-4, 还是-8 有关
系, 如果指针类型是 char* 那么指针-1 就是-1, 如果指针类型
是 int* 那么指针-1 就是-4

```

473
474     /* compiler optimization barrier needed for GCC >= 3.4 */
475     __asm__ __volatile__(": : :memory");
476
477     memset ((void*)gd, 0, sizeof (gd_t));
478     gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
479     memset (gd->bd, 0, sizeof (bd_t));
480
481     monitor_flash_len = _bss_start - _armboot_start;
482
483     for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
484         if ((*init_fnc_ptr)() != 0) {
485             hang ();
486         }
487     }

```

这里减
bd 正好
bd 结构
的首地址
在这里

```

473     /* compiler optimization barrier needed for GCC >= 3.4 */
474     __asm__ __volatile__(": : :memory");
475
476
477     memset ((void*)gd, 0, sizeof (gd_t));
478     gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
479     memset (gd->bd, 0, sizeof (bd_t));
480
481     monitor_flash_len = _bss_start - _armboot_start;
482
483     for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
484         if ((*init_fnc_ptr)() != 0) {
485             |         hang ();
486         }
487     }

```

init_sequence 这是一个函数指针数组，我们需要执行的各类函数都放在这个数组里面

```

416 init_fnc_t *init_sequence[] = {
417     cpu_init, /* basic CPU dependent setup */
418 #if defined(CONFIG_SKIP_RELOCATE_UBOOT)
419     reloc_init, /* Set the relocation address
420                  do this AFTER cpuid
421                  as possible */
422 #endif
423     board_init, /* basic Board dependent setup */
424     interrupt_init, /* set up exceptions */
425     env_init, /* initialize environment */
426     init_baudrate, /* initialize baudrate settings */
427     serial_init, /* serial communications setup */
428     console_init_f, /* stage 1 init of console */
429     display_banner, /* say that we are here */
430 #if defined(CONFIG_DISPLAY_CPUINFO)
431     print_cpuinfo, /* display CPU info (and speed) */
432 #endif
433 #if defined(CONFIG_DISPLAY_BOARDINFO)
434     checkboard, /* display board info */
435 #endif
436 #if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
437     init_func_i2c,
438 #endif
439     dram_init, /* configure available RAM banks */
440     display_dram_config,

```

init_fnc_t *init_sequence[] 第一个先看中括号，代表这是一个数组，然后在看 *，代表这是 0 个指针。然后这个指针数组是存放什么类型的東西呢？看最前面的 init_fnc_t 是放函数类型

里面每一个函数都是 int 函数名 (void)
因为 init_fnc_t 类型是 typedef int (init_fnc_t)(void)

```

473     /* compiler optimization barrier needed for GCC >= 3.4 */
474     __asm__ __volatile__(": : :memory");
475
476
477     memset ((void*)gd, 0, sizeof (gd_t));
478     gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
479     memset (gd->bd, 0, sizeof (bd_t));
480
481     monitor_flash_len = _bss_start - _armboot_start;
482
483     for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
484         if ((*init_fnc_ptr)() != 0) {
485             |         hang ();
486         }
487     }

```

*init_fnc_ptr 完全等价于
*init_fnc_ptr != NULL
取出数组里面的元素看看是否停止循环

现在我们来看 init_sequence 数组里面的每个函数是用来干什么的

```

416 init_fnc_t *init_sequence[] = {
417     cpu_init,           /* basic CPU dependent setup */
418 #if defined(CONFIG_SKIP_RELocate_UBOOT)
419     reloc_init,         /* Set the relocation done flag, must
420                         do this AFTER cpu_init(), but as soon
421                         as possible */
422 #endif
423     board_init,          /* basic board specific setup */
424     interrupt_init,      /* set up exceptions */
425     env_init,            /* initialize environment */
426     init_baudrate,       /* initialize baudrate settings */
427     serial_init,          /* serial communications setup */
428     console_init_f,      /* stage 1 init of console */
429     display_banner,      /* say that we are here */
430 #if defined(CONFIG_DISPLAY_CPUINFO)
431     print_cpuinfo,        /* display CPU info (and speed) */
432 #endif
433 #if defined(CONFIG_DISPLAY_BOARDINFO)
434     checkboard,          /* display board info */
435 #endif
436 #if defined(CONFIG_HARD_I2C) || defined(CONFIG_SOFT_I2C)
437     init_func_i2c,
438 #endif
439     dram_init,            /* configure available RAM banks */
440     display_dram_config,

```

cpu_init 在 uboot/cpu/s5pc11x/cpu.c

```

00088: int cpu_init (void) ←
00089: {
00090:     /*
00091:      * setup up stacks if necessary
00092:      */
00093: #ifdef CONFIG_USE_IRQ
00094:     IRQ_STACK_START = _armboot_start - CFG_MALLOC_LEN - CFG_GBL_DATA_SIZE - 4;
00095:     FIQ_STACK_START = IRQ_STACK_START - CONFIG_STACKSIZE_IRQ;
00096: #endif
00097:     return 0;
00098: }

```

我们 board.c 是用来初始化 CPU 外接的一些芯片内容，所以这里的 cpu_init 没做什么事

board_init 在 uboot/board/x210/x210.c

```

00085: int board_init(void) ←
00086: {
00087:     DECLARE_GLOBAL_DATA_PTR;
00088: #ifdef CONFIG_DRIVER_SMC911X
00089:     smc9115_pre_init();
00090: #endif
00091:
00092: #ifdef CONFIG_DRIVER_DM9000
00093:     dm9000_pre_init();
00094: #endif
00095:
00096:     gd->bd->bi_arch_number = MACH_TYPE;
00097:     gd->bd->bi_boot_params = (PHYS_SDRAM_1+0x100);
00098:
00099:     return 0;
00100: }

```

这里声明是因为我们下面用到了 gd 全局变量，当然也可以用 include，看写代码的喜好

00065: static void dm9000_pre_init(void)
00066: {
00067: unsigned int tmp;
00068:
00069: #if defined(CN9000_1GBIT_DATA)
00070: SROM_Bw_REG |= ~0x0f << 4);
00071: SROM_Bw_REG |= (1<<7) | (1<<6) | (1<<5) | (1<<4);
00072: #else
00073: SROM_Bw_REG |= ~0x0f << 4);
00074: SROM_Bw_REG |= (0<<6) | (0<<5) | (0<<4);
00075: #endif
00076: SROM_B1C_REG = ((0<<28)|(1<<23)|(5<<16)|(1<<1);

初始化 CPU 接入 DM9000 网卡芯片的 IO 口

这里得到 uboot 机器码，是自己写的，不是 CPU 全球唯一 ID 哦！

uboot 获取 bi_boot_params 参数，实现将 bootargs 变量放在内存的 bi_boot_params 指定的地址处

bootargs=root=/dev/mmcblk0p2 rootfstype=ext3 initrd=/init
内核启动后就知道在内存哪个地址段找到 bootargs 变量，这样才能找到文件系统。

```

X210_sd.h
00498: #endif
00499:
00500: #define CONFIG_NR_DRAM_BANKS 2 /* we have 2 bank of DRAM */
00501: #define SDRAM_BANK_SIZE 0x10000000 /* 512 MB 1qm */
00502: // #define SDRAM_BANK_SIZE 0x20000000 /* 1GB 1qm */
00503: #define PHYS_SDRAM_1 MEMORY_BASE_ADDRESS /* SDRAM Bank #1 */
00504: #define PHYS_SDRAM_1_SIZE SDRAM_BANK_SIZE
00505: #define PHYS_SDRAM_2 MEMORY_BASE_ADDRESS2 /* SDRAM Bank #2 */
00506: #define PHYS_SDRAM_2_SIZE SDRAM_BANK_SIZE

```

这个表示有 2 片 DDR 芯片

每一片 DDR 大小 512MB

第 1 片 DDR 的首地址

这个 `bi_arch_number` 机器码会在 `uboot` 启动 `kernel` 内核的时候，`uboot` 自动把机器码传递给 `kernel`，`kernel` 拿到这个机器码就会和自己配置内核时候设置的机器码比对，如果相同就启动内核，如果不同就不启动。

`interrupt_init` 在 `uboot/cpu/s5pc11x/Interrupts.c`

```
00178: int interrupt_init(void)
00179: {
00180:
00181:     S5PC11X_TIMERS *const timers = S5PC11X_GetBase_TIMERS();
00182:
00183:     /* use PWM Timer 4 because it has no output */
00184:     /* prescaler for Timer 4 is 16 */
00185:     timers->TCFG0 = 0x0f00;
00186:     if (timer_load_val == 0) {
00187:         /*
00188:          * for 10 ms clock period @ PCLK with 4 bit divider = 1/2
00189:          * (default) and prescaler = 16. Should be 10390
00190:          * @33.25MHz and @ 66 MHz
00191:         */
00192:         timer_load_val = get_PCLK() / (16 * 100);
00193:     }
00194:
00195:     /* load value for 10 ms timeout */
00196:     Lastdec = timers->TCNTB4 = timer_load_val;
00197:     /* auto load. manual update of Timer 4 */
```

这个 time4 定时器不支持中断，不能用于 linux 多任务切换，这个只是用在 uboot 下面，uboot 定时。就是我们在启动内核前敲打键盘进入 uboot 的定时时间

`env_init` 在 `uboot/Common/Env_movi.c` 下，这个 `env_init` 函数在很多文件都有

```
env_dataflash.c
env_dataflash.o
env_eeprom.c
env_eeprom.o
env_flash.c
env_flash.o
environment.c
environment.o
env_movi.c
```

`env_init` 函数是根据你存储介质使用的类型不同，`env_init` 函数实现方法也不同，nandflash，EMMC，INAND，eeprom 这些介质不同对应的 `env_init` 也不同

```
00037: int env_init(void)
00038: {
00039: #if defined(ENV_IS_EMBEDDED)
00040:     ulong total;
00041:     int crc1_ok = 0, crc2_ok = 0;
00042:     env_t *tmp_env1, *tmp_env2;
00043:
00044:     total = CFG_ENV_SIZE;
00045:
00046:     tmp_env1 = env_ptr;
00047:     tmp_env2 = (env_t *)((ulong)env_ptr + CFG_ENV_SIZE);
00048:
00049:     crc1_ok = (crc32(0, tmp_env1->data, ENV_SIZE) == tmp_env1->crc);
00050:     crc2_ok = (crc32(0, tmp_env2->data, ENV_SIZE) == tmp_env2->crc);
00051:
00052:     if (!crc1_ok && !crc2_ok)
00053:         gd->env_valid = 0;
00054:     else if(crc1_ok && !crc2_ok)
00055:         gd->env_valid = 1;
00056:     else if(!crc1_ok && crc2_ok)
```

初次判断 INAND 拷贝进 DDR 的这一份 env 环境变量能不能用

当前的环境变量还没有从 INAND 拷贝进内存 DDR，所以现在环境变量 `env` 是不能用的

init_baudrate 设置串口波特率，在 uboot/lib_arm/board.c 里

```
178 static int init_baudrate (void)
179 {
180     char tmp[64]; /* long enough for environment variables */
181     int i = getenv_r ("baudrate", tmp, sizeof (tmp));
182     gd->bd->bi_baudrate = gd->baudrate = (i > 0)
183         ? (int) simple_strtoul (tmp, NULL, 10)
184         : CONFIG_BAUDRATE;
185
186     return (0);
187 }
```

这里是获取环境变量的波特率值，获取终端 uboot 设置的环境变量值给 tmp 变量，这个波特率值是字符串
baudrate=115200
ethaddr=00:10:5e:26:e8

如果 getenv_r 设置失败，就执行 CONFIG_BAUDRATE，这个宏默认为 115200

因为这个波特率值是字符串，所以这里要把字符串转换成 int 的数给 tmp

Bd->bi_baudrate 得到了波特率值。这是个全局变量，其他地方绝对会使用它。

serial_init 在 uboot/cpu/s5pc11x/serial.c

```
00054: int serial_init(void)
00055: {
00056:     serial_setbrg();
00057:
00058:     return (0);
00059: }
```

```
00041: void serial_setbrg(void)
00042: {
00043:     DECLARE_GLOBAL_DATA_PTR;
00044:
00045:     int i;
00046:     for (i = 0; i < 100; i++);
00047: }
```

我们发现其实 serial_init 函数什么都没有做，这是为什么呢？这是因为我们在汇编阶段已经对串口的寄存器初始化过了。这里就不需要再初始化了，但是其它平台可能会在这里进行串口初始化，毕竟 C 语言容易理解些。

console_init_f 在 uboot/common/console.c

```
00362: int console_init_f (void)
00363: {
00364:     gd->have_console = 1;
00365:
00366: #ifdef CONFIG_SILENT_CONSOLE
00367:     if (getenv("silent") != NULL)
00368:         gd->flags |= GD_FLG_SILENT;
00369: #endif
00370:
00371:     return (0);
00372: }
```

其实这里面没有做什么，只是把 have_console 置 1 了而已

display_banner 在 uboot/lib_arm/board.c 里

```
304 static int display_banner (void)
305 {
306     printf ("\n\n%s\n", version_string);
307     debug ("U-Boot code: %08lx -> %08lx BSS: -> %08lx\n",
308            _armboot_start, _bss_start, _bss_end);
309 #ifdef CONFIG_MEMORY_UPPER_CODE/* by scsuh */
310     debug ("\t\tbMalloc and Stack is above the U-Boot Code.\n");
311 #else
312     debug ("\t\tbMalloc and Stack is below the U-Boot Code.\n");
313 #endif
314 #ifdef CONFIG_MODEM_SUPPORT
315     debug ("Modem Support enabled\n");
316 #endif
317 #ifdef CONFIG_USE_IRQ
318     debug ("IRQ Stack: %08lx\n", IRQ_STACK_START);
319     debug ("FIQ Stack: %08lx\n", FIQ_STACK_START);
320 #endif
321     open_backlight(); //lqm.
322     //open_gprs();
323
324     return (0);
325 }
```

这句打印有用

这是初始化打开 LCD 的背光，但是加在这个地方就是有点不合理

print_cpuinfo 在 uboot/cpu/s5pc11x/s5pc110/speed.c

```

00228: int print_cpuinfo(void)
00229: {
00230:     uint set_speed;
00231:     uint tmp;
00232:     uchar result_set;
00233:
00234: #if defined(CONFIG_CLK_533_133_100_100)
00235:     set_speed = 53300;
00236: #elif defined(CONFIG_CLK_667_166_166_133)
00237:     set_speed = 66700;
00238: #elif defined(CONFIG_CLK_800_200_166_133)
00239:     set_speed = 80000;
00240: #elif defined(CONFIG_CLK_1000_200_166_133)
00241:     set_speed = 100000;
00242: #elif defined(CONFIG_CLK_1200_200_166_133)
00243:     set_speed = 120000;
00244: #else
00245:     set_speed = 100000;
00246:     printf("Any CONFIG_CLK_XXX is not enabled\n");
00247: #endif
00248:     tmp = (set_speed / (get_ARMCLK() / 1000000));
00249:
00250:     if((tmp < 105) && (tmp > 95)){
00251:         result_set = 1;
00252:     } else {
00253:         result_set = 0;
00254:     }
00255:
00256: #ifdef CONFIG_MCP_SINGLE
00257:     printf("\nCPU: S5PV210@%ldMHz(%s)\n", get_ARMCLK() / 1000000, ((result_set == 1) ? "C
00258: #else
00259:     printf("\nCPU: S5PC110@%ldMHz(%s)\n", get_ARMCLK() / 1000000, ((result_set == 1) ? "C
00260: #endif
00261:     printf("      APPLL = %ldMHz, HclkMsys = %ldMHz, PclkMsys = %ldMHz\n",
00262:            get_FCLK() / 1000000, get_HCLK() / 1000000, get_PCLK() / 1000000);
00263: #if 1
00264:     printf("      MPLL = %ldMHz, EPPLL = %ldMHz\n",

```

```

U-Boot 1.3.4 (Nov 19 2015 - 17:31:41) for x210

CPU: S5PV210@1000MHz(OK)
      APPLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPPLL = 96MHz
      HclkDsys = 166MHz, PclkDsys = 83MHz
      HclkPsys = 133MHz, PclkPsys = 66MHz
      SCLKA2M = 200MHz

Serial = CLKUART

```

这就是 uboot 启动时，打印 CPU 的时钟信息

到底 是 用 那 个 宏 参 数 是 在
uboot/include/configs/x210_sd.h 里 面 定 义 的

```

00301: #define CONFIG_CLK_667_166_166_133
00302: #define CONFIG_CLK_533_133_100_100
00303: #define CONFIG_CLK_800_200_166_133
00304: #define CONFIG_CLK_800_100_166_133
00305: #define CONFIG_CLK_1000_200_166_133
00306: #define CONFIG_CLK_400_200_166_133
00307: #define CONFIG_CLK_400_100_166_133

```

我选择的是 CONFIG_CLK_1000.....这个参数

dram_init 在 uboot/board/X210/X210.c

```

00102: int dram_init(void)
00103: {
00104:     DECLARE_GLOBAL_DATA_PTR;
00105:
00106:     gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
00107:     gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;
00108:
00109: #if defined(PHYS_SDRAM_2)
00110:     gd->bd->bi_dram[1].start = PHYS_SDRAM_2;
00111:     gd->bd->bi_dram[1].size = PHYS_SDRAM_2_SIZE;
00112: #endif
00113:
00114: #if defined(PHYS_SDRAM_3)
00115:     gd->bd->bi_dram[2].start = PHYS_SDRAM_3;
00116:     gd->bd->bi_dram[2].size = PHYS_SDRAM_3_SIZE;
00117: #endif
00118:
00119:     return 0;
00120: }

```

第 1 片内存起始地址

这里不是 初始化 硬件
DDR，硬件 DDR 在前面汇
编就初始化了，这里只是
将硬件 ddr 信息赋值给 gd
全局变量

第 1 片内存大小

第 2 片内存

display_dram_config 在 uboot/lib_arm/board.c 里

```

334 static int display_dram_config (void)
335 {
336     int i;
337
338 #ifdef DEBUG
339     puts ("RAM Configuration:\n");
340
341     for(i=0; i<CONFIG_NR_DRAM_BANKS; i++) {
342         printf ("Bank %d: %08lx ", i, gd->bd->bi_dram[i].start);
343         print_size(gd->bd->bi_dram[i].size, "\n");
344     }
345 #else
346     ulong size = 0;
347
348     for (i=0; i<CONFIG_NR_DRAM_BANKS; i++) {
349         size += gd->bd->bi_dram[i].size;
350     }
351
352     puts("DRAM:   ");
353     print_size(size, "\n");
354 #endif
355
356     return (0);
357 }

```

这些 内存的 打印 信息 就是 从
dram_init 函数 里面 先 赋 值 给
全局 变量 gd 的， 这里 再 来 赋
值 gd 参数， 打印 出 来

所以 init_sequence 就是 初始化 一 些 信 息

现在回到 board.c

```
488 #ifndef CFG_NO_FLASH  
489 |     /* configure available FLASH banks */  
490 |     size = flash_init ();  
491 |     display_flash_config (size);  
492 #endif /* CFG_NO_FLASH */  
493  
494 #ifdef CONFG_VED
```

这是初始化 norflash 我没用到 norflash, 这段代码没用

```
598 599 #if defined(CONFIG_X210)  
600 |     #if defined(CONFIG_GENERIC_MMC)  
601 |         puts ("SD/MMC: ");  
602 |         mmc_exist = mmc_initialize(gd->bd);  
603 |         if (mmc_exist != 0)  
604 |             {  
605 |                 puts ("0 MB\n");  
606 |             }  
607 #ifdef CONFIG_CHECK_X210CV3  
608 |             check_flash_flag=0;//check inand  
609 #endif  
610 |         }  
611 #ifdef CONFIG_CHECK_X210CV3  
612 |         else  
613 |             {  
614 |                 check_flash_flag=1;//check inand ok!  
615 |             }  
616 #endif  
617 |     #endif  
618  
619 |     #if defined(CONFIG_MTD_ONENAND)  
620 |         puts("OneNAND: ");  
621 |         onenand_init();  
622 |         /*setenv("bootcmd", "onenand read c0008000 80000 380000;bootm");  
623 |     #else  
624 |         //puts("OneNAND: (FSR layer enabled)\n");  
625 |     #endif  
626  
627 |     #if defined(CONFIG_CMD_NAND)  
628 |         puts("NAND: ");  
629 |         nand_init();
```

在 uboot/drivers/mmc/mmc.c 文件下
01177: int mmc_initialize(bd_t *bis)
01178: {
01179: struct mmc *mmc;
01180: int err;
01181: INIT_LIST_HEAD(&mmc_devices);
01182: cur_dev_num = 0;
01183: if (board_mmc_init(bis) < 0)
01184: cpu_mmc_init(bis);
01185: #if defined(DEBUG_S3C_HSMMC)
01186: print_mmc_devices(',');
01187: #endif
01188: #ifdef CONFIG_CHECK_X210CV3
01189: mmc = find_mmc_device(1);//Lqm
01190: #endif
01191: #else
01192: mmc = find_mmc_device(0);
01193: #endif
01194: #endif
01195: if (mmc) {
01196: if (mmc) {
01197: Uboot 初始化硬件的代码都在 drivers 目录下, 比如 SD 卡, 网卡

```
01177: int mmc_initialize(bd_t *bis)  
01178: {  
01179:     struct mmc *mmc;  
01180:     int err;  
01181:     INIT_LIST_HEAD(&mmc_devices);  
01182:     cur_dev_num = 0;  
01183:     if (board_mmc_init(bis) < 0)  
01184:         cpu_mmc_init(bis);  
01185: #if defined(DEBUG_S3C_HSMMC)  
01186:     print_mmc_devices(',');  
01187: #endif  
01188: #ifdef CONFIG_CHECK_X210CV3  
01189:     mmc = find_mmc_device(1);//Lqm  
01190: #endif  
01191: #else  
01192:     mmc = find_mmc_device(0);  
01193: #endif  
01194: #endif  
01195:     if (mmc) {  
01196:         if (mmc) {  
01197:             board_mmc_init 这个是板级初始化, 但是里面没有实现什么东西  
而且我们发现三星平台的 cpu, SSPV210,S3C2440,S3C6410 的 SD 卡控制器都是一模一样的, 都是 S3C_HSMMC, 所以我们可以拷贝相同的代码  
在 uboot/cpu/s5pc11x/s5pc110/cpu.c 文件下  
00232: int cpu_mmc_init(bd_t *bis)  
00233: {  
00234: #ifdef CONFIG_S3C_HSMMC  
00235:     setup_hsmmc_clock();  
00236:     setup_hsmmc_cfg_gpio();  
00237:     return smdk_s3c_hsmmc_init();  
00238: #else  
00239:     return 0;  
00240: #endif  
00241: }  
Sd 卡时钟初始化  
Sd 卡 GPIO 初始化  
发现串口, NAND, 芯片上面的 soc 外设硬件初始化都是放在 cpu 目录下的
```

```

598 #if defined(CONFIG_X210)
599     #if defined(CONFIG_GENERIC_MMC)
600         puts ("SD/MMC: ");
601         mmc_exist = mmc_initialize(gd->bd);
602         if (mmc_exist != 0)
603         {
604             puts ("0 MB\n");
605         }
606     #ifdef CONFIG_CHECK_X210CV3
607         |     check_flash_flag=0;//check inand error!
608     #endif
609     #endif
610     }
611 #ifdef CONFIG_CHECK_X210CV3
612     else
613     {
614         check_flash_flag=1;//check inand ok!
615     }
616 #endif
617     #endif
618
619     #if defined(CONFIG_MTD_ONENAND)
620         puts("OneNAND: ");
621         onenand_init();
622         /*setenv("bootcmd", "onenand read c0008000 80000 380000;bootr
623     #else
624         //puts("OneNAND: (FSR layer enabled)\n");
625     #endif
626
627     #if defined(CONFIG_CMD_NAND)
628         puts("NAND: ");
629         nand_init();
630     #endif
631
632     /* get IP address */
633     gd->bd->bi_ip_addr = getenv_IPAddr ("ipaddr");
634
635     /* MAC Address */
636     {
637         int i;
638         ulong reg;
639         char *s, *e;
640         char tmp[64];
641
642         i = getenv_r ("ethaddr", tmp, sizeof (tmp));
643         s = (i > 0) ? tmp : NULL;
644
645         for (reg = 0; reg < 6; ++reg)
646         {
647             gd->bd->bi_enetaddr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
648             if (s)
649                 s = (*e) ? e + 1 : e;
650         }
651
652     #ifdef CONFIG_HAS_ETH1
653         i = getenv_r ("eth1addr", tmp, sizeof (tmp));
654         s = (i > 0) ? tmp : NULL;
655
656         for (reg = 0; reg < 6; ++reg)
657         {
658             gd->bd->bi_enet1addr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
659             if (s)
660                 s = (*e) ? e + 1 : e;
661         }
662     #endif
663     }
664
665     devices_init ();/* get the devices list going. */
666
667
668 /*****lxg added*****
669 #ifdef CONFIG_MPAD
670     extern int x210_preboot_init(void);
671     x210_preboot_init();
672 #endif
673
674     . . .
675     extern void update_all(void);
676     if(check_menu_update_from_sd()==0)//update mode
677     {
678         puts ("[LEFT DOWN] update mode\n");
679         run_command("fdisk -c 0",0);
680         update_all();
681     }
682     else
683         puts ("[LEFT UP] boot mode\n");
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701

```

MMC 初始化失败

MMC 初始化成功

如果有 ONENAND flash 那么就在 x210_sd.h 里面定义这个宏，然后这里代码就会去初始化 onenand

这是初始化 nandflash，也是在 x210_sd.h 里面定义宏

获取 IP 地址给 bi_ip_addr

这是获取网卡的 MAC 地址

把获取的 MAC 地址放入全局变量 gd

这个 ip 地址和 MAC 地址的默认值在哪里设置的呢？其实是在 include/configs/x210_sd.h 文件下

00232: #define CONFIG_BOOTARGS "console=ttySAC2,11
00233: /*#define CONFIG_BOOTARGS "console=ttySAC
00234: #define CONFIG_ETHADDR 00:40:5c:26:0a:5b
00235: #define CONFIG_NETMASK 255.255.0.0
00236: #define CONFIG_GATEWAYIP 192.168.1.188
00237: #define CONFIG_SERVERIP 192.168.1.102
00238: #define CONFIG_GATEWAYIP 192.168.0.1

这是 LCD 硬件初始化

这是 uboot 的自动更新，板子启动后 uboot 会检查更新按键是否按下，按键按下，uboot 从 SD 卡槽读取 uboot 和内核转入到 INAND 中，这样实现了量产时快速烧写 INAND，实现批量生产

```

903     for (;;) {
904         main_loop ();
905     }

```

最后 uboot 进入死循环，uboot 在这里等待你在终端敲命令行

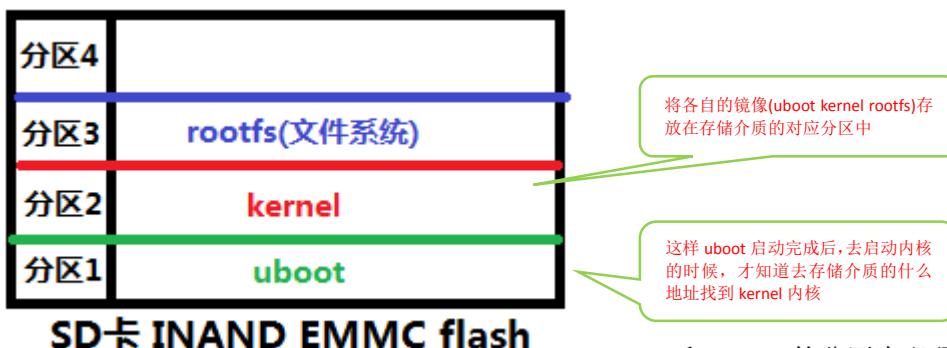
总结：uboot 第 1 阶段为汇编阶段，第 2 阶段为 C 语言阶段

uboot 第 1 阶段代码在 CPU 内部 SRAM 中运行，第 2 阶段代码在 DDR 中运行

uboot 第 1 阶段是初始化 CPU 内部的 SOC，如看门狗，串口等。第 2 阶段初始化 CPU 外部，如网卡，LCD，INAND 这些。

Uboot 启动 kernel(内核)

第 1 点：uboot，kernel，rootfs 不是随便放在 SD 卡、EMMC、INAND、存储介质的任意位置的。必须对 SD 卡或者 INAND、EMMC 进行分区。



SD 卡 INAND EMMC flash

uboot 和 kernel 的分区表必须一致。

同时 uboot 和 kernel 要和 SD 卡的实际分区要一直。

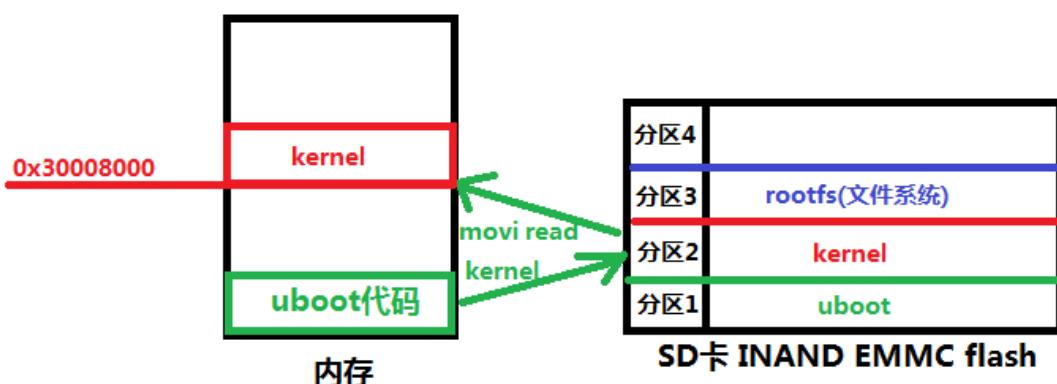
比如我们这个内核使用的链接地址是 0x30008000。

```

x210 # print
bootcmd='movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000
mtdpart=80000 400000 3000000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192.168.1.102
rc console=ttySAC2,115200
Environment size: 440/16380 bytes
```

```

Uboot 命令 movi read kernel 30008000; 就是将 SD/INAND 卡的里面的 kernel 内核读出来放在 DDR 的 0x30008000 地址处



现在只是将 kernel 读进内存，但是还没有启动内核。

```

x210 # print
bootcmd='movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000
mtdpart=80000 400000 3000000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192.168.1.102
rc console=ttySAC2,115200
Environment size: 440/16380 bytes
```

```

紧跟着 uboot 命令 bootm 30008000 去 DDR 的 0x30008000 地址处启动内核

你可以手敲 movi read，bootm 试一下

movi 命令读取内核到 DDR 是 INAND 介质使用的。如果是 nandflash 存储介质，要用 nand 命令读取内核

movi read kernel , 这个 kernel 字符是 uboot 规定 SD 卡划分一个区域给内核存放。这种叫 raw 分区方式。这个内核在内存的地址 0x30008000 不是我随便取的，而是根据内核的链接脚本来设置的。

zImage 镜像和 uImage 镜像的区别

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel# ls
arch      CREDITS   firmware initrd.img.cpio lib      mm      net      scripts  tools    vmlinux.o
block     crypto     fs       ipc      MAINTAINERS modules.builtin README  security  usr
COPYING   Documentation include Kbuild  Makefile  modules.order REPORTING-BUGS sound   virt
COPYING.txt drivers   init     kernel   mk       Module.synvers samples System.map vmlinuz
```

这个 vmlinuz 其实就是内核编译出来的镜像，但是是 elf 格式的。elf 格式就是 X86 编译出来的镜像就只能在 x86 上面跑，开发板自己编译出来的镜像可以在开发板上跑，但是把 x86 编译的镜像烧录进开发板是不行的。

arm-none-linux-gnueabi-objcopy

所以我们用 arm-none-linux-gnueabi-objcopy 这个交叉编译工具将 vmlinuz 制作成 zImage 镜像，这种 zImage 镜像就是可以烧录的。而且注意 vmlinuz 镜像有 78M 这么大，制作成 zImage 就变成了 7.8M，符合嵌入式这种小巧的存储空间。

zImage 就可以烧录进开发板使用了。所以原则上是没有 uImage 这个东西。

但是 uboot 自己做了一个工具，这个工具可以把 zImage 制作成 uImage。



所以 uboot 启动内核时，内核应该是 uImage，但是 uboot 中也是支持 zImage 启动的。

至于到底 uboot 支持不支持 zImage，我们要查看 x210_sd.h 里面的宏 LINUX_ZIMAGE_MAGIC

移植不同型号 INAND/SD 卡启动 Uboot 和 kernel 内核，

首先在启动文件 uboot/uboot/lib_arm/board.c 下面找到 staert_armboot 启动函数

```
444 void start_armboot (void)
445 {
446     init_fnc_t **init_fnc_ptr;
447     char *s;
448     int mmc_exist = 0;
449 #if !defined(CFG_NO_FLASH) || defined(CONFIG_VFD)
450     ulong size;
451 #endif
452
453 #if defined(CONFIG_VFD) || defined(CONFIG_LCD)
454 #if defined(CONFIG_X210)
455
456     #if defined(CONFIG_GENERIC_MMC)
457         puts ("SD/MMC: ");
458         mmc_exist = mmc_initialize(gd->bd);
459     #endif
460
461 #endif
462 }
```

mmc_initialize 函数在 uboot/drivers/mmc/mmc.c 中

然后重点关注 mmc_initialize

```

01177: int mmc_initialize(bd_t *bis)
01178: {
01179:     struct mmc *mmc;
01180:     int err;
01181:
01182:     INIT_LIST_HEAD(&mmc_devices);
01183:     cur_dev_num = 0;
01184:
01185:     if (board_mmc_init(bis) < 0)
01186:         cpu_mmc_init(bis); // 执行 cpu_mmc_init
01187:
01188: #if defined(DEBUG_S3C_HSMMC)
01189:     print_mmc_devices(' ', ' ');
01190: #endif
01191:
01192: #ifdef CONFIG_CHECK_X210CV3
01193:     mmc = find_mmc_device(1); // lqm
01194: #else
01195:     mmc = find_mmc_device(0);
01196: #endif
01197:     if (mmc) {

```

在 uboot/cpu/s5pc11x/cpu.c 执行 cpu_mmc_init

```

00232: int cpu_mmc_init(bd_t *bis)
00233: {
00234: #ifdef CONFIG_S3C_HSMMC
00235:     setup_hsmmc_clock();
00236:     setup_hsmmc_cfg_gpio();
00237:     return smdk_s3c_hsmmc_init();
00238: #else
00239:     return 0;
00240: #endif
00241: }

```

初始化 mmc_devices 链表的头节点，当 SD 卡插进板子的时候，系统就会生成一个新的 mmc_devices 结构体，插入初始化的这个链表 mmc_devices 结构体里面的指针

执行 cpu_mmc_init

初始化 mmc 控制器时钟和 gpio 管脚，其实这两个函数写在这里并不太合适

这个函数就很重要了

在 uboot/drivers/mmc/s3c_hsmmc.c 文件中

```
00440: int smdk_s3c_hsmmc_init(void)
00441: {
00442: #ifdef OM_PIN
00443:     if(OM_PIN == SDMMC_CHANNEL0) {
00444:         int err;
00445:         printk("SD/MMC channel0 is selected for booting device.\n");
00446:         err = s3c_hsmmc_initialize(0);
00447:         return err;
00448:     } else if (OM_PIN == SDMMC_CHANNEL1) {
00449:         int err;
00450:         printk("SD/MMC channel1 is selected for booting device.\n");
00451:         err = s3c_hsmmc_initialize(1);
00452:         return err;
00453:     } else
00454:         printk("SD/MMC isn't selected for booting device.\n");
00455: #endif
00456:
00457:     int err;
00458:
00459: #ifdef USE_MMC0
00460:     err = s3c_hsmmc_initialize(0);
00461:     if(err)
00462:         return err;
00463: #endif
00464:
00465: #ifdef USE_MMC1
00466:     err = s3c_hsmmc_initialize(1);
00467:     if(err)
00468:         return err;
00469: #endif
00470:
00471: #ifdef USE_MMC2
00472:     err = s3c_hsmmc_initialize(2);
00473:     if(err)
00474:         return err;
00475: #endif
00476:
00477: #ifdef USE_MMC3
00478:     err = s3c_hsmmc_initialize(3);
00479:     if(err)
00480:         return err;
00481: #endif
00482:     return -1;
00483: } ? end smdk_s3c_hsmmc_init ?
```

因为在某个文件定义了
USE_MMC0 和 USE_MMC1，所
以这个 s3c_hsmmc_initialize 会
执行 2 次，传入两个通道号

```

00395: static int s3c_hsmmc_initialize(int channel)
00396: {
00397:     struct mmc *mmc;
00398:     这个推测句是CPU接卡数据的函数
00399:     mmc = &mmc_channel[channel];
00400:     sprintf(mmc->name, "S3C_HSMMC%d", channel);
00401:     mmc->priv = &mmc_host[channel];
00402:     mmc->send_cmd = s3c_hsmmc_send_command;
00403:     mmc->set_ios = s3c_hsmmc_set_ios;
00404:     mmc->init = s3c_hsmmc_init;
00405:     mmc->voltages = MMC_VDD_32_33 | MMC_VDD_33_34;
00406:     mmc->host_caps = MMC_MODE_4BIT | MMC_NODE_HS_52MHz | MMC_MODE_HS;
00407:     #if defined(USE_MMCO_8BIT) || defined(USE_MMCC2_8BIT)
00408:     mmc->host_caps |= MMC_MODE_8BIT;
00409:     #endif
00410:     mmc->f_min = 400000;
00411:     mmc->f_max = 52000000;
00412:
00413:     mmc_host[channel].clock = 0;
00414:
00415:     switch(channel) {
00416:         case 0:
00417:             mmc_host[channel].ioaddr =
00418:                 break;
00419:         case 1:
00420:             mmc_host[channel].ioaddr =
00421:                 (void *)ELFIN_HSMMC_1_BASE;
00422:                 break;
00423:         case 2:
00424:             mmc_host[channel].ioaddr =
00425:                 (void *)ELFIN_HSMMC_2_BASE;
00426:                 break;
00427:     #ifdef USE_MMCC3
00428:         case 3:
00429:             mmc_host[channel].ioaddr =
00430:                 (void *)ELFIN_HSMMC_3_BASE;
00431:                 break;
00432:     #endif
00433:         default:
00434:             printk("mmc err: not supported channel %d\n", channel);
00435:     }
00436:     return mmc_register(mmc);
00437: } ? end s3c_hsmmc_initialize ?

```

这个通道号,确定是CPU的哪一个SD卡控制器

这就是SD控制器的对象,一个SD控制器通道,对应一个*mmc,所以每执行一次s3c_hsmmc_initialize函数就会创建一个*mmc

但是*mmc是指针啊,没有对应的内存,怎么能叫创建sd控制器对象呢?你看这个数组就是有意义的内存,分配给了mmc

这个函数就是CPU的SD卡控制器向SD卡写数据的函数

你看这里面就有操作MMC控制器寄存器部分

将这次分配给SD控制器通道的内存地址注册进mmc_devices链表

在/uboot/drivers/mmc/mmc.c 执行 mmc_register 函数

```

01089: int mmc_register(struct mmc *mmc)
01090: {
01091:     /* Setup the universal parts of the block interface just once */
01092:     mmc->block_dev.if_type = IF_TYPE_MMC;
01093:     mmc->block_dev.dev = cur_dev_num++;
01094:     mmc->block_dev.removable = 1;
01095:     mmc->block_dev.block_read = mmc_bread;
01096:     mmc->block_dev.block_write = mmc_bwrite;
01097:
01098:     INIT_LIST_HEAD(&mmc->link);
01099:
01100:     list_add_tail(&mmc->link, &mmc_devices);
01101:
01102:     return 0;
01103: }

```

这里是继续给这次分配的SD卡通的写参数

这里就很重要了,把这次分配的sd卡控制器通道的内存地址挂接在mmc_devices链表的后面

这就说明了mmc_devices是一个链表,里面的节点就是给sd卡/MMC通道挂接的。

我们继续回过头来分析 mmc_initialize 函数

```
01177: int mmc_initialize(bd_t *bis)
01178: {
01179:     struct mmc *mmc;
01180:     int err;
01181:
01182:     INIT_LIST_HEAD(&mmc_devices);
01183:     cur_dev_num = 0;
01184:
01185:     if (board_mmc_init(bis) < 0)
01186:         cpu_mmc_init(bis);
01187:
01188: #if defined(DEBUG_S3C_HSMMC)
01189:     print_mmc_devices(',');
01190: #endif
01191:
01192: #ifdef CONFIG_CHECK_X210CV3
01193:     mmc = find_mmc_device(1); //Lqm
01194: #else
01195:     mmc = find_mmc_device(0);
01196: #endif
01197:     if (mmc) {
01198:         if (mmc_init(mmc));
01199:             if (err)
01200:                 err = mmc_init(mmc);
01201:             if (err) {
01202:                 printf("Card init fail!\n");
01203:                 return err;
01204:             }
01205:     }
01206 }
```

这个函数工作原理就是循环搜索 mmc_devices 链表，去寻找刚才我们 mmc_register 注册的 MMC/SD 卡控制器通道号

如果找到了我们注册的 MMC 控制器通道号，就去初始化这个通道的 MMC 控制器

mmc_init 还是在 uboot/drivers/mmc/mmc.c

```
01112: int mmc_init(struct mmc *host) ←
01113: {
01114:     int err;
01115:
01116:     err = host->init(host);
01117:
01118:     if (err)
01119:         return err;
01120:
01121: /* Reset the Card */
01122: err = mmc_go_idle(host);
01123:
01124: if (err)
01125:     return err;
01126:
01127: /* Test for SD version 2 */
01128: err = mmc_send_if_cond(host);
01129:
01130: /* Now try to get the SD card's operating condition */
01131: err = mmc_send_app_op_cond(host);
01132: /* If the command timed out, we check for an MMC card */
01133: if (err == TIMEOUT) {
01134:     err = mmc_send_op_cond(host);
01135:
01136:     if (err)
01137:         return UNUSABLE_ERR;
01138: } else
01139:     if (err)
01140:         return UNUSABLE_ERR;
01141:
01142: return mmc_startup(host);
01143: } ? end mmc_init ?
```

所以 mmc_init 函数是用来初始化外部 SD 卡/INAND 芯片，cpu_mmc_init 是用来初始化 CPU 芯片内部的 MMC 控制器。

为什么 MMC 控制器要分，mmc.c 和 s3c_hsmmc.c，不能写在一起吗？

```
/uboot/uboot/drivers/mmc# ls
c.a  Makefile  mmc.c  mmc.o  s3c_hsmmc.c  s3c_hsmmc.o
/uboot/uboot/drivers/mmc#
```

下面我们来分析 mmc.c

mmc.c 里面的发送函数会去调用 s3c_hsmmc.c 的函数来执行

mmc.c 的数据发送/接受函数

s3c_hsmmc.c 的数据发送/接受函数

```

mmc.c
00051: int mmc_set_blocklen(struct mmc *mmc, int len)
00052: {
00053:     struct mmc_cmd cmd;
00054:     cmd.opcode = MMC_CMD_SET_BLOCKLEN;
00055:     cmd.resp_type = MMC_RSP_R1;
00056:     cmd.arg = len;
00057:     cmd.flags = 0;
00058:
00059:
00060:     return mmc_send_cmd(mmc, &cmd, NULL);
00061: }
00200: int mmc_go_idle(struct mmc *host)
00201: {
00202:     struct mmc_cmd cmd;
00203:     int err;
00204:
00205:     udelay(1000);
00206:
00207:     cmd.opcode = MMC_CMD_GO_IDLE_STATE;
00208:     cmd.arg = 0;
00209:     cmd.resp_type = MMC_RSP_NONE;
00210:     cmd.flags = 0;
00211:
00212:     err = mmc_send_cmd(host, &cmd, NULL);
0046: int mmc_send_cmd(struct mmc *mmc, struct mmc_cmd *cmd, struct mmc_data *data)
0047: {
0048:     return mmc->send_cmd(mmc, cmd, data);
0049: }

s3c_hsmmc.c
00395: static int s3c_hsmmc_initialize(int channel)
00396: {
00397:     struct mmc *mmc;
00398:     mmc = &mmc_channel[channel];
00399:     sprintf(mmc->name, "S3C_HSMMC%d", channel);
00400:     mmc->priv = &mmc_host[channel];
00401:     mmc->send_cmd = s3c_hsmmc_send_command;
00402:     mmc->set_ios = s3c_hsmmc_set_ios;
00403:     mmc->init = s3c_hsmmc_init;
00404:     mmc->voltage =
00405:         mmc->host_caps = MMC_VDD_32_33 | MMC_VDD_33_34;
00406:     mmc->host_caps |= MMC_MODE_4BIT | MMC_MODE_HS_52MHz | MMC_MODE_HS;
00407: #if defined(USE_MMC2_8BIT) || defined(USE_MMC2_8BIT)
00408:     mmc->host_caps |= MMC_MODE_8BIT;
00409: #endif
00410: }

s3c_hsmmc_send_command.c
00100: s3c_hsmmc_send_command(struct mmc *mmc, struct mmc_cmd *cmd,
00101:                           struct mmc_data *data)
00102: {
00103:     struct sdhci_host *host = mmc->priv;
00104:
00105:     int flags, i;
00106:     u32 mask;
00107:
00108:
00109:     /* Clear Error Interrupt Status Register before issuing cmd */
00110:     writew(readw(host->iodev + SDHCI_ERRINT_STATUS),
00111:           host->iodev + SDHCI_ERRINT_STATUS);
00112:
00113:     /* Clear Normal Interrupt Status Register before issuing cmd */
00114:     writew(readw(host->iodev + SDHCI_INT_STATUS),
00115:           host->iodev + SDHCI_INT_STATUS);

```

从以上分析我们确定了 mmc.c 和 s3c_hsmmc_send_command.c 是两个工程师写的。

写 mmc.c 的工程师只是负责写 SD 卡的时序操作。s3c_hsmmc_send_command.c 的工程师只负责写 S5PV210 的 CPU 端 MMC 寄存器函数，然后发送数据给 SD 卡。

这样的好处就是 SD 卡有 2G,4G,16G~256G。mmc.c 的工程师把这些规格的代码都写好。如果你从三星平台换到飞思卡尔平台了，你不需要改 mmc.c，你只需要修改 s3c_hsmmc_send_command.c 这个里面的代码就 OK 了

如果你增加了新容量的 SD 卡比如 512G，你才去修改 mmc.c。换 CPU 是不用修改 mmc.c.

从三星官方移植 uboot 实验

```
root@ubuntu:/home/xiang/S5PV210/samsung-uboot/u-boot-samsung-dev# ls
api           Changelog_Samsung  drivers    lib_blackfin  lib_nios2      microblaze_config.mk  onenand_b1     sparc_config.mk
api_examples   common            examples   lib_fdt       lib_ppc       mips_config.mk   onenand_ipl   tools
arm_config.mk config.mk        COPYING    lib_generic   lib_sh       mkconfig        post         u-boot.lds
avr32_config.mk          COPYING    i386_config.mk lib_i386     lib_sparc   m68k_config.mk  nand_spl    README
blackfin_config.mk          credits   image_split  lib_m68k    lib_microblaze MAINTAINERS   net        rules.mk
board          disk              lib_arm     lib_mips    MAKEALL    nios2_config.mk  sd_fusing   sh_config.mk
CHANGELOG      doc               lib_avr32   lib_nios   Makefile    nios_config.mk
CHANGELOG-before-U-Boot-1.1.5          lib_nios2
```

这是三星原厂的 uboot

```
143 ifeq ($(_ARCH), arm)
144 #CROSS_COMPILE = arm-linux-
145 #CROSS_COMPILE = /usr/local/arm/4.4.1-eabi-cortex-a8/usr/bin/arm-linux-
146 #CROSS_COMPILE = /usr/local/arm/4.2.2-eabi/usr/bin/arm-linux-
147 CROSS_COMPILE = /opt/arm-2009q3/bin/arm-none-linux-gnueabi-
```

修改 Makefile 交叉编译工具链

```
root@ubuntu:/home/xiang/S5PV210/samsung-uboot/u-boot-samsung-dev# make x210_sd_config
make: *** No rule to make target `x210_sd_config'. Stop.
```

在配置 x210 的时候出现 stop 问题，这是因为 makefile 文件里面没有 x210_sd_config 选项

```
2581 smdkv210single_config :| unconfig
2582     @$(MKCONFIG) $(@:_config)= arm s5pc11x smdkc110 samsung s5pc110
2583     @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/smdkc110/config.mk
2584
2585 smdkv210single_rev02_config :| unconfig
2586     @$(MKCONFIG) $(@:_config)= arm s5pc11x smdkc110 samsung s5pc110
2587     @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/smdkc110/config.mk
2588
2589 smdkv210single_ubt_config :| unconfig
2590     @$(MKCONFIG) $(@:_config)= arm s5pc11x smdkc110 samsung s5pc110
2591     @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/smdkc110/config.mk
2592
2593 smdkv210vogue_config :| unconfig
2594     @$(MKCONFIG) $(@:_config)= arm s5pc11x smdkc110 samsung s5pc110
2595     @echo "TEXT_BASE = 0xc3e00000" > $(obj)board/samsung/smdkc110/config.mk
```

我们发现 Makefile 只有 smdkv210single 系列选项，这里面每个配置项也就是(_config)对应 include/configs/ 目录下的文件，我们选择 smdkv210single_config

这样配置 smdkv210single 选项就修改了对应在 include/configs/ 下 smdkv210single.h 头文件

```
root@ubuntu:/home/xiang/S5PV210/samsung-uboot/u-boot-samsung-dev# make smdkv210single_config
Configuring for smdkv210single board...
```

配置成功

```
Configuring for smdkv210single board...
root@ubuntu:/home/xiang/S5PV210/samsung-uboot/u-boot-samsung-dev# make
Generating include/autoconf.mk.dep
for dir in tools examples api_examples ; do make -C $dir _depend ; done
make[1]: Entering directory `/home/xiang/S5PV210/samsung-uboot/u-boot-samsung-dev/tools'
```

然后直接 make 就可以编译出 uboot 镜像了

u-boot
u-boot.bin

将 uboot.bin 烧录进开发板

SD checksum Error

我们启动开发板发现没有进入 uboot 初始化，这是为什么呢？

这个 SD checksum Error 是 CPU 自身启动后会自动去检查 SDIO0 通道有没有插入 SD 卡/INAND。发现是接入的 INAND，那么程序就会自动再去检测 SDIO1 通道的位置有没有 SD 卡。这里并不是检查没检查到 SD 卡，而是根本没有执行到 SDIO1 检查程序位置。

我们记得串口第一个输出的字符应该是“OK”，但是这里没有打印 OK，那么程序应该出现在 lowlevel_init.S 程序里。

开发板程序运行的第一个位置就在 uboot/cpu/s5pc11x/start.S

```
284 |     ldr |     sp, =0xd0036000 /* end of sram dedicated to u-boot
285 |     sub |     sp, sp, #12| /* set stack */
286 |     mov |     fp, #0
287 |
288 |     bl |     lowlevel_init| /* go setup pll,mux,memory */
289 |     /* To hold max8698 output before releasing power on switch
```

根据“OK”字符的理解，lowlevel_init 程序里面才执行 OK，所以问题定位在这里，进去看看

所以我们进入 uboot/board/samsung/smdkc110/lowlevel_init.S

```

110 |         ldr |    r0, =0xff000fff
111 |         bic |    r1, pc, r0|
112 |         ldr |    r2, _TEXT_BASE|
113 |         bic |    r2, r2, r0|
114 |         cmp |    r1, r2
115 |         beq |    1f|      |
116 |
117 |     /* init PMIC chip */
118 |     bl PMIC_InitIp ←
119 |
120 |     /* init system clock */
121 |     bl system_clock_init ← 这里是初始化时钟
122 |
123 |     /* Memory initialize */
124 |     bl mem_ctrl_asm_init ← 这里是初始化 DDR
125 |
126 1:   /* for UART */
127 |     bl uart_asm_init ← 这里是初始化串口，也是打印“OK”的地方，这里没有执行，证明前面的汇编有地方死循环了
128 |
129 |     bl tzpc_init
130 |
131 |
132 #if defined(CONFIG_ONENAND)
133     bl oneandcon_init

```

最后发现是 PMIC_InitIp 汇编代码出问题了，因为我们没有用 PMIC 电源管理 IC 给 CPU 供电，所以这里的 PMIC_InitIp 执行就进入死循环，始终找不到电源管理 IC。所以我们把这段代码屏蔽就好了

```

117 |         /* init PMIC chip */
118 //|     bl PMIC_InitIp
119

```

屏蔽 PMIC_InitIp 就好了

U-Boot 1.3.4 (Dec 7 2015 - 14:54:04) for SMDKV210

CPU: S5PV210@1000MHz(OK)
APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
MPPLL = 667MHz, EPLL = 80MHz
HclkDsys = 166MHz, PclkDsys = 83MHz
HclkPsys = 133MHz, PclkPsys = 66MHz
SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 1 GB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment

In: serial

我们再次启动开发板，uboot 就启动起来了

U-Boot 1.3.4 (Dec 7 2015 - 14:54:04) for SMDKV210

CPU: S5PV210@1000MHz(OK)
APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
MPPLL = 667MHz, EPLL = 80MHz
HclkDsys = 166MHz, PclkDsys = 83MHz
HclkPsys = 133MHz, PclkPsys = 66MHz
SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 1 GB → 我发现 DRAM 不对，怎么会是 1GB 呢？我明明只接了两片 DDR，一共才 512M
Flash: 8 MB → 而且 SD 卡初始化失败

SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!

0 MB
NAND: 0 MB → NANDflash 没有接 CPU 所以是 0

The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment

In: serial

现在结论是 uboot 可以运行了

但是很多功能无法使用

```

U-Boot 1.3.4 (Dec 7 2015 - 14:54:04) for SMDKV210

CPU: S5PV210@1000MHz (OK)
      APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPLL = 80MHz
      HclkSys = 166MHz, PclkSys = 83MHz
      HclkPsys = 133MHz, PclkPsys = 66MHz
      SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 1 GB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment

In: serial

```

我们先修改 SMDKV210 名字，改成自己的

因为我们编译 uboot 的时候，配置的是 make smdkv210single_config, 所以我们修改 uboot/include/configs 对应的 smdkv210single.h 文件

/include/configs# vim smdkv210single.h 进入头文件

```

510
511 #define CONFIG_IDENT_STRING| " for SMDKV210"
512
511 #define CONFIG_IDENT_STRING| " for XZZ"
512

```

修改这个名字就是
改成 xzz

```

U-Boot 1.3.4 (Dec 7 2015 - 14:54:04) for SMDKV210

CPU: S5PV210@1000MHz (OK)
      APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPLL = 80MHz
      HclkSys = 166MHz, PclkSys = 83MHz
      HclkPsys = 133MHz, PclkPsys = 66MHz
      SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 1 GB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment

In: serial

```

我们来修改时钟，但是时钟默认是对的

```

280 // #define CONFIG_CLK_667_166_166_133
281 // #define CONFIG_CLK_533_133_100_100
282 // #define CONFIG_CLK_800_200_166_133
283 // #define CONFIG_CLK_800_100_166_133
284 #define CONFIG_CLK_1000_200_166_133
285 // #define CONFIG_CLK_400_200_166_133
286 // #define CONFIG_CLK_400_100_166_133

```

这就是修改时钟的地方，我们现在选择的是 1000_200.....

```

280 // #define CONFIG_CLK_667_166_166_133
281 // #define CONFIG_CLK_533_133_100_100
282 // #define CONFIG_CLK_800_200_166_133
283 // #define CONFIG_CLK_800_100_166_133
284 #define CONFIG_CLK_1000_200_166_133
285 // #define CONFIG_CLK_400_200_166_133
286 // #define CONFIG_CLK_400_100_166_133

```

如果我改成 800_200..... 可以不呢？其实是可以的，CPU 降频工作没有问题，但是你的串口波特率计算方式和 DDR 的时钟有可能要修改。但是这些串口和 DDR 时钟都是在另外一个 c 文件写死了的，所以可能会出问题

DDR 配置信息更改

```
U-Boot 1.3.4 (Dec 7 2015 - 14:54:04) for SMDKV210

CPU: S5PV210@1000MHz(OK)
      APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPLL = 80MHz
      HclkDsys = 166MHz, PclkDsys = 83MHz
      HclkPsys = 133MHz, PclkPsys = 66MHz
      SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 1 GB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment
In: serial
```

执行 bdinfo 查看一下

```
SMDKV210 # bdinfo
arch_number = 0x00000998
env_t = 0x00000000
boot_params = 0x20000100
DRAM bank = 0x00000000
-> start = 0x20000000
-> size = 0x20000000
DRAM bank = 0x00000001
-> start = 0x40000000
-> size = 0x20000000
ethaddr = 00:40:5C:26:0A:5B
ip_addr = 192.168.0.20
baudrate = 115200 bps
SMDKV210 #
```

我们是 512M 的 DDR，这里怎么成为 1G 了？

这里 DRAM0 起始地址不应该 0x2000_0000

0x7FFF_FFFF

0x6000_0000

0x5FFF_FFFF

0x4000_0000

0x3FFF_FFFF

0x2000_0000

DRAM1

DRAM0

而且 DRAM0 接的 DDR 大小也设置错了

因为我们 uboot text_base 地址是从 0x3000_0000 开始的



这是 DDR 和 CPU DRAM 的接线关系

用 md 命令查看 0x20000000 地址发现有数据正常输出，证明内存初始化没有问题

在修改 DDR 信息之前，我们看看内存初始化代码有没有问题

```
SMDKV210 # md 20000000 10
20000000: 20588816 40046180 18611951 30000800
20000010: 8200c808 50900014 00290200 d0120000
20000020: 84001640 c8950400 80061042 40004219
20000030: 80120800 44089580 08911800 10420000
SMDKV210 # md 40000000 10
40000000: 0a201043 51100620 18000804 38240200
40000010: 10011201 00344a00 10040000 08008000
40000020: b3148002 08001800 21000044 41520c00
40000030: 09000820 70a20400 28000000 50000000
SMDKV210 #
```

```

467
468 #define CONFIG_NR_DRAM_BANKS 2 /* we have 2 bank of DRAM */
469 #define SDRAM_BANK_SIZE 0x20000000 /* 512 MB */
470 #define PHYS_SDRAM_1 MEMORY_BASE_ADDRESS /* SDRAM Bank #1 */
471 #define PHYS_SDRAM_1_SIZE SDRAM_BANK_SIZE
472 #define PHYS_SDRAM_2 (MEMORY_BASE_ADDRESS + SDRAM_BANK_SIZE) /* SDRAM Bank #2 */
473 #define PHYS_SDRAM_2_SIZE SDRAM_BANK_SIZE
474

```

```

467
468 #define CONFIG_NR_DRAM_BANKS 2 /* we have 2 bank of DRAM */
469 #define SDRAM_BANK_SIZE 0x20000000 /* 512 MB */
470 #define PHYS_SDRAM_1 MEMORY_BASE_ADDRESS /* SDRAM Bank #1 */
471 #define PHYS_SDRAM_1_SIZE SDRAM_BANK_SIZE
472 #define PHYS_SDRAM_2 (MEMORY_BASE_ADDRESS + SDRAM_BANK_SIZE) /* SDRAM Bank #2 */
473 #define PHYS_SDRAM_2_SIZE SDRAM_BANK_SIZE
474

```

```

71
72 #define MEMORY_BASE_ADDRESS 0x20000000
73

```

我将其改成我们要的 0x30000000

全部修改后

```

72 #define MEMORY_BASE_ADDRESS 0x30000000
468 #define CONFIG_NR_DRAM_BANKS 2 /* we have 2 bank of DRAM */
469 #define SDRAM_BANK_SIZE 0x10000000 /* 512 MB */
470 #define PHYS_SDRAM_1 MEMORY_BASE_ADDRESS /* SDRAM Bank #1 */
471 #define PHYS_SDRAM_1_SIZE SDRAM_BANK_SIZE
472 #define PHYS_SDRAM_2 0x40000000
473 #define PHYS_SDRAM_2_SIZE SDRAM_BANK_SIZE

```

这样就完全正确了，每片 DDR 大小为 0x10000000(256M)，两片加起来 512M

```

U-Boot 1.3.4 (Dec 8 2015 - 11:07:19) for ASTON210

CPU: S5PV210@1000MHz (OK)
      APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPLL = 80MHz
                  HclkDsys = 166MHz, PclkDsys = 83MHz
                  HclkPsys = 133MHz, PclkPsys = 66MHz
                  SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 512 MB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment

```

```

SMDKV210 # bdfinfo
arch_number = 0x00000998
env_t = 0x00000000
boot_params = 0x20000100
DRAM bank = 0x00000000
-> start = 0x20000000
-> size = 0x10000000
DRAM bank = 0x00000001
-> start = 0x40000000
-> size = 0x10000000
ethaddr = 00:40:5C:26:0A:5B
ip_addr = 192.168.0.20
baudrate = 115200 bps
SMDKV210 #

```

板子启动后 DDR 的起始地址和大小都正确了

Inand 驱动问题修改

```
U-Boot 1.3.4 (Dec 8 2015 - 11:07:19) for ASTON210

CPU: S5PV210@1000MHz (OK)
      APLL = 1000MHz, HclkMsys = 200MHz, PclkMsys = 100MHz
      MPLL = 667MHz, EPLL = 80MHz
      HclkDsys = 166MHz, PclkDsys = 83MHz
      HclkPsys = 133MHz, PclkPsys = 66MHz
      SCLKA2M = 200MHz

Serial = CLKUART
Board: SMDKV210
DRAM: 512 MB
Flash: 8 MB
SD/MMC: unrecognised EXT_CSD structure version 6
unrecognised EXT_CSD structure version 6
Card init fail!
0 MB
NAND: 0 MB
The input address don't need a virtual-to-physical translation : 23e9c008
*** Warning - using default environment
```

SD 卡驱动启动失败，导致接的 INAND 存储器也启动失败，因为 INAND 也是 SDIO 接口。

根据打印信息查找发现问题出现在 uboot/drivers/mmc/mmc.c 中

```
00817: if (ext_csd_struct > 5) {
00818:     printf("unrecognised EXT_CSD structure "
00819:             "version %d\n", ext_csd_struct);
00820:     err = -1;
00821:     goto ↓out;
00822: }
```

向上看我们发现 if(ext...) 代码是在 mmc_read_ext_csd 函数中

```
/*
 * Read and decode extended CSD.
 */
static int mmc_read_ext_csd(struct mmc *host)
{
    int err;
    u8 *ext_csd;
    unsigned int ext_csd_struct;

    if (host->version < (MMC_VERSION_4 | MMC_VERSION_MMC))
        return 0;

    /*
     * As the ext_csd is so large and mostly unused, we do
     * raw block in mmc_card.
     */

    err = mmc_send_ext_csd(host, ext_csd);
    if (err)
```

mmc_send_ext_csd 函数发送命令给 SD 卡/Inand，然后同时 err 接受 SD/inand 返回的数据

```
ext_csd_struct = ext_csd[EXT_CSD_REV];
if (ext_csd_struct > 5) {
    printf("unrecognised EXT_CSD structure "
        "version %d\n", ext_csd_struct);
    err = -1;
    goto ↓out;
}
```

问题是我们的 Inand 版本号大于 5，但是我手上的 uboot 代码只支持小于 5 的版本号。小于 5 的版本号只有 SD 卡才支持。

根据网上的一些搜索结果将 ext_csd_struct > 5 改成 ext_csd_struct > 6 或者更高值就可以解决

```
ext_csd_struct = ext_csd[EXT_CSD_REV];
if (ext_csd_struct > 5) {
    printf("unrecognised EXT_CSD structure "
        "version %d\n", ext_csd_struct);
    err = -1;
    goto ↓out;
}
```

修改前

```
00820:     ext_csd_struct = ext_csd[EXT_CSD_REV];
00821:     if (ext_csd_struct > 6) {
00822:         /* //lqm masked.
00823:         printf("unrecognised EXT_CSD structure "
00824:             "version %d\n", ext_csd_struct);
00825:         err = -1;
00826:         goto out;
00827:         */
00828:         ext_csd_struct = 5; //lqm changed.
00829:     }
```

修改后就可以了，但是这只是在软

件上骗过了 Inand 驱动，不知道运行的时候会不会出现 Inand 版本不同造成时序也不同，这种就只有调试的时候再确定了。但是一般 Inand 版本不同不会影响 Inand/SD 卡的驱动。这种只有尝试了才知道。

Uboot 串口节点修改

比如我以前是串口 1 用来输出终端调试信息，我现在想改成串口 2 输出终端调试信息。

修改 Uboot/include/configs/smdkv210single.h 配置文件

```
00145: #define CONFIG_SERIAL3           1 /* we use UART1 on SMDK110 */
00146:
00147: #define CFG_HUSH_PARSER          /* use "hush" command parser */
```

这个 CONFIG_SERIAL3 定义的是 UART2 他这里注释错了

如果要改成串口 0 来输出终端信息，只需要添加 CONFIG_SERIAL0
这个宏，把 CONFIG_SERIAL3 屏蔽就是了。

CONFIG_SERIAL 的宏定义在 s5pc110.h 定义了很多组

就是修改这一行，来选择哪一路串口输出终端调试信息

这是修改网卡 mac 地址

修改板子上 uboot 默认网卡地址

在 smdkv210single.h 配置文件

00213: #define CONFIG_ETHADDR	00:40:5c:26:0a:5b
00214: #define CONFIG_NETMASK	255.255.255.0
00215: #define CONFIG_IPADDR	192.168.0.20
00216: #define CONFIG_SERVERIP	192.168.0.10
00217: #define CONFIG_GATEWAYIP	192.168.0.1

修改掩码,网卡 IP 地址,服务器 IP 地址, 网关地址

网卡芯片在 Uboot 上移植



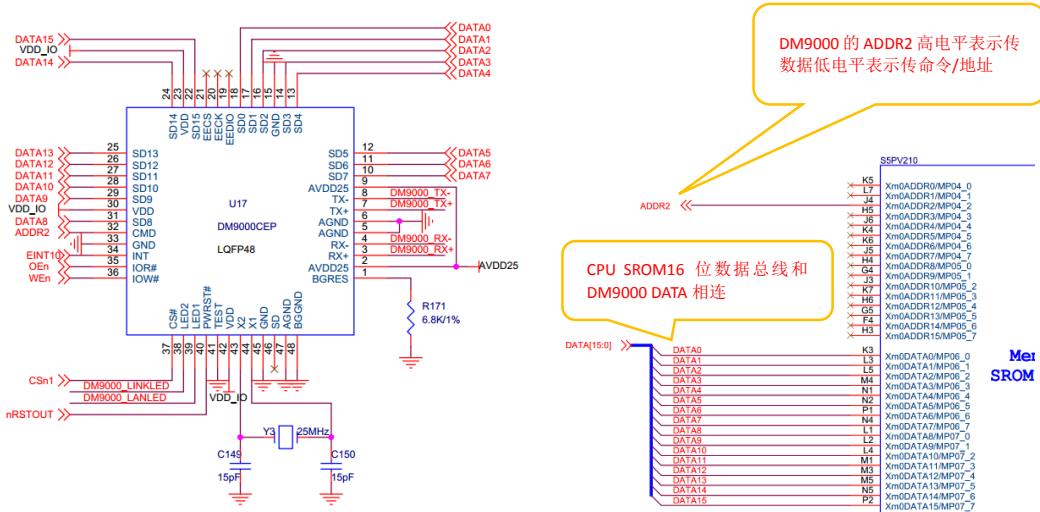
这就是 CPU 和网卡的导线连接方式

2.1.1 OVERVIEW OF SROM CONTROLLER

S5PV210 SROM Controller (SROMC) support external 8 / 16-bit NOR Flash/ PROM/ SRAM memory.

S5PV210 SROM Controller supports 6-bank memory up to maximum 16Mbyte per bank.

S5PV210 的 SROM 可以用 8 位/16 位总线连接外部 SRAM,nor falso 等 8/16 位总线存储芯片
恰好 DM9000 把数据引脚伪装成了 8 位/16 位的存储芯片模式。



Uboot 网卡初始化我们得先从 uboot 启动文件开始找
启动文件在 uboot/uboot/cpu/s5pc11x/start.S

```
411 |     ldr |     pc, _start_armboot
412 |
413 _start_armboot:
414 |     .word start_armboot
```

在 start.S 找到启动 start_armboot 函数

start_armboot 在 uboot/lib_arm/board.c 中

```
414 int print_cpuinfo (void); /* test-only */
415
416 init_fnc_t *init_sequence[] = {
417 |     cpu_init, |           /* basic cpu dependent setup */
418 #if defined(CONFIG_SKIP_RELOCATE_UBOOT)
419 |     reloc_init, |          /* Set the relocation done flag, must
420 |     |           |          do this AFTER cpu_init(), but as soon
421 |     |           |          as possible */
422 #endif
423 |     board_init, |          /* basic board dependent setup */
424 |     interrupt_init, |       /* set up exceptions */
425 |     env_init, |           /* initialize environment */
```

执行 board_init 函数

这个 board_init 函数在 uboot/board/Samsung/smdkc110.c

这个配置宏在 smdkv210single.h 里面定义了

```
int board_init(void)
{
    DECLARE_GLOBAL_DATA_PTR;
#ifdef CONFIG_DRIVER_SMC911X
    smc9115_pre_init();
#endif
#ifdef CONFIG_DRIVER_DM9000
    dm9000_pre_init();
#endif
    gd->bd->bi_arch_number = MACH_TYPE;
    gd->bd->bi_boot_params = (PHYS_SDRAM_1+0x100);

    return 0;
}
```

```
static void dm9000_pre_init(void)
{
    unsigned int tmp;

    #if defined(DM9000_16BIT_DATA)
    SROM_BW_REG &= ~(0xf << 20);
    SROM_BW_REG |= (0<<23) | (0<<22) | (0<<21) | (1<<20);
    #else
    SROM_BW_REG &= ~(0xf << 20);
    SROM_BW_REG |= (0<<19) | (0<<18) | (0<<16);
    #endif
    SROM_BCS_REG = ((0<<28) | (1<<24) | (5<<16) | (1<<12) | (4<<8) | (6<<4) | (0<<0));

    tmp = MP01CON_REG;
    tmp &=~(0xf<<20);
    tmp |=(2<<20);
    MP01CON_REG = tmp;
```

这个 dm9000_pre_init 函数就是初始化 CPU 里面的 SROM 接口支持 DM9000 网卡芯片
主要操作 SROM_BW_REG, SROM_BCS_REG, MP01CON_REG 三个寄存器
下面修改 CPU 连接 DM9000 的数据引脚基地址

在 uboot/include/configs/smdkv210single.h 文件中

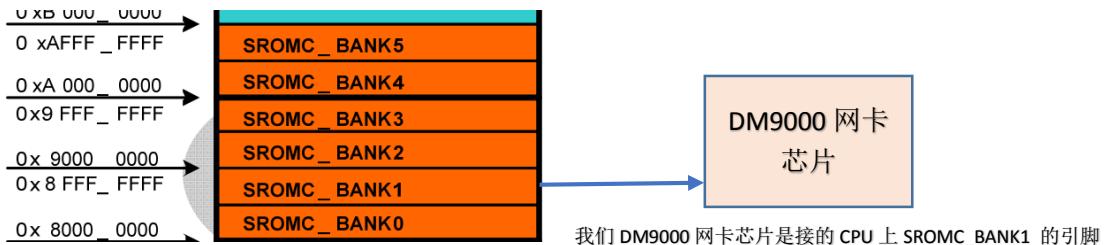
```

00128: #define DM9000_16BIT_DATA
00129:
00130: #define CONFIG_DRIVER_DM9000      1
00131:
00132: #ifdef CONFIG_DRIVER_DM9000
00133: #define CONFIG_DM9000_BASE          (0xA8000000)
00134: #define DM9000_IO                 (CONFIG_DM9000_BASE)
00135: #if defined(DM9000_16BIT_DATA)
00136: #define DM9000_DATA              (CONFIG_DM9000_BASE+2)
00137: #else
00138: #define DM9000_DATA              (CONFIG_DM9000_BASE+1)
00139: #endif
00140: #endif

```

因为 DM9000 是数据和地址/命令都复用 DATA 管脚，所以 CPU 发命令 / 地址依靠 DM9000_IO 这个地址操作

DM9000 和 CPU 交互数据靠 DM9000_DATA 管脚



0x8800_0000	0x8FFF_FFFF	128MB	SROM Bank 1
0x9000_0000	0x97FF_FFFF	128MB	SROM Bank 2

所以 SROM_Bank1 起始地址在 0x8800_0000

```

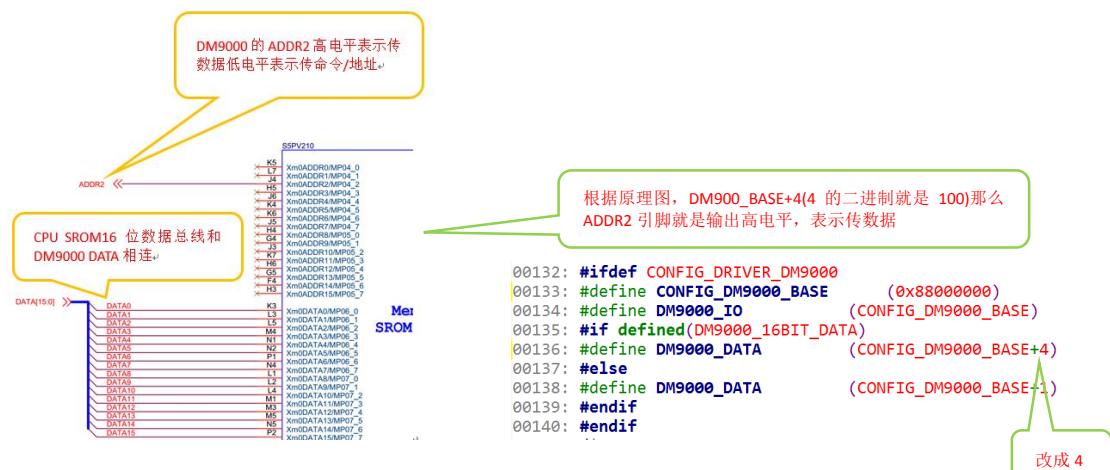
00132: #ifdef CONFIG_DRIVER_DM9000
00133: #define CONFIG_DM9000_BASE          (0x88000000)
00134: #define DM9000_IO                 (CONFIG_DM9000_BASE)
00135: #if defined(DM9000_16BIT_DATA)
00136: #define DM9000_DATA              (CONFIG_DM9000_BASE+2)

```

基址修改成 0x88000000 没问题

这个 DM9000 和 CPU 数据总线就必须改成 CONFIG_DM9000_BASE+4

为什么数据总线要+4 呢？



修改完配置文件后编译 uboot 下载金开发板

```
ASTON210 # ping 192.168.1.141
ERROR: resetting DM9000 -> not responding
dm9000 i/o: 0x88000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:57:5c:26:0a:5b
operating at 100M full duplex mode
host 192.168.1.141 is alive
```

在开发板启动后 uboot 模式下 ping 主机 IP，记得 uboot CONFIG_SERVERIP 环境变量要改成主机 ip

打印 host Is alive 表示网卡正常工作了。

用 tftp 去下载一个主机文件，下载成功没有问题，网卡移植完成。但是这种移植是建立在 DM9000 网卡芯片支持 S5PV210 驱动的情况下。

UBOOT 启动时 LCD 显示开机界面

Uboot 的 LCD 驱动程序在 uboot/drivers/video/Mpadfb.c 文件中

```
00877: void mpadfb_init()
00878: {
00879: //  unsigned short int *pFB;//这里一定要用short类型!
00880: //  int i;
00881:     fb_init();//Lqm masked for test
00882:     lcd_port_init();
00883:     lcd_reg_init();
00884: #ifdef CONFIG_CHECK_X210CV3
00885:     init_logo();
00886: #endif
00887:     display_logo(&s5pv210_fb);
00888: #if DISP_MODE == TRULY043)
00889:     backlight_brightness_init(0);
00890: #else//AT070TN92
00891:     backlight_brightness_init(1);
00892: #endif
```

```
00712: void fb_init()←
00713: {
00714:     get_HCLKD();
00715:     info.bitmap.info.width = Lcd.width;
00716:     info.bitmap.info.height = Lcd.height;
00717:     info.bitmap.info.bpp = Lcd.bpp;
00718:     info.bitmap.info.bytes_per_pixel = Lcd.bpp / 8;
00719:     info.bitmap.info.pitch = Lcd.width * Lcd.bpp / 8;
00720:
00721:     info.bitmap.info.red_mask_size = Lcd.rgba.r_mask;
00722:     info.bitmap.info.red_field_pos = Lcd.rgba.r_field;
00723:     info.bitmap.info.green_mask_size = Lcd.rgba.g_mask;
00724:     info.bitmap.info.green_field_pos = Lcd.rgba.g_field;
00725:     info.bitmap.info.blue_mask_size = Lcd.rgba.b_mask;
00726:     info.bitmap.info.blue_field_pos = Lcd.rgba.b_field;
00727:     info.bitmap.info.alpha_mask_size = Lcd.rgba.a_mask;
00728:     info.bitmap.info.alpha_field_pos = Lcd.rgba.a_field;
```

fb_init 主要是将你当前使用的 LCD 屏幕各种参数赋值给全局变量 info，方便其它 c 文件调用

```
00752: void lcd_port_init()←
00753: {
00754:     writel(0x22222222, GPF0CON);
00755:     writel(0xffffffff, GPF0DRV);
00756:     writel(0x0, GPF0PUD);
00757:     writel(0x22222222, GPF1CON);
00758:     writel(0xffffffff, GPF1DRV);
```

Lcd_port_init LCD 相关的引脚初始化

```
00796: void lcd_reg_init()←
00797: {
00798:     //display path selection
00799:
00800:     writel((readl(S5PV210_DISPLAY_CONTROL) & ~(0x3<<0)) | (0x2<<0));
00801:
00802:     //turn all windows off
00803:
00804:     writel((readl(S5PV210_WINCON0) & ~0x1), S5PV210_WINCON0);
00805:     writel((readl(S5PV210_WTNCON1) & ~0x1), S5PV210_WTNCON1);
```

Lcd_reg_init S5PV210 内部 LCD 控制器初始化

```
00887:     display_logo(&s5pv210_fb);
00888: display_log 就是用来显示 logo 图片的，也就是 uboot 启动时开机图片，这个 display_log 函数是在 uboot/drivers/video/Logo.c 中
```

```
00888: #if(DISP_MODE == TRULY043)  
00889:     backlight brigness_init(0);  
00890: #else//AT070TN92  
00891:     backlight brigness_init(1);  
00892: #endif
```

这是打开/关闭 LCD 背光

现在我们来处理如果我的 LCD 屏幕从 800X480 分辨率换成了 1024X600 的屏幕该怎么修改驱动呢？

```
00712: void fb_init()
00713: {
00714:     get_HCLKD();
00715:     info.bitmap.info.width = Lcd.width; ——————
00716:     info.bitmap.info.height = Lcd.height;
00717:     info.bitmap.info.bpp = Lcd.bpp;
00718:     info.bitmap.info.bytes_per_pixel = Lcd.bpp / 8;
00719:     info.bitmap.info.pitch = Lcd.width * Lcd.bpp / 8;
00720:
00721:     info.bitmap.info.red_mask_size = Lcd.rgb444.r_mask;
00722:     info.bitmap.info.red_field_pos = Lcd.rgb444.r_field;
```

其实主要就是修改传入 info 全

局变量的 width 和 height 变量值

```
00024: #if(DISP_MODE == AT070TN92)
00025: static struct s5pv210fb_lcd lcd = {  
00026:     .width      = 800,  
00027:     .height     = 480,  
00028:     .bpp        = 32,  
00029:     .freq       = 60,  
00030:  
00031:     .output     = S5PV210FB_OUTPUT_RGB,  
00032:     .rgb_mode   = S5PV210FB_MODE_BGR_P,  
00033:     .bpp_mode   = S5PV210FB_BPP_MODE_32BPP,  
00034:     .swap        = S5PV210FB_SWAP_WORD,  
00035:  
00036:     .rgba = {  
00037:         .r_mask    = 8,  
00038:         .r_field   = 0  
00039:  
00040: #if(DISP_MODE == AT070TN92)
00041: static struct s5pv210fb_lcd lcd =  
00042:     .width      = 1024,  
00043:     .height     = 600,  
00044:     .bpp        = 32
```

LCD 驱动参数变量官方一般都给写好了，你只需要修改 width 和 height 值就是了，如果遇到时序差异就要修改 `s5pv210fb_lcd` 变量里面的时序参数变量

修改后就支持 1024x600 了

而且LCD驱动还给出了其它屏幕参数

```
而且LCD驱动还输出了以下所有常数  
00110: #elif(DISP_MODE == TRULY043)  
00111: static struct s5pv210fb_lcd lcd = {  
00112:     .width        = 480,  
00113:     .height       = 272,  
00114:     .bpp          = 32,  
00115:     .freq         = 60,  
00116:  
00117:     .output       = S5PV210FR_OUTPUT_RGR,  
  
00153: #elif(DISP_MODE==VGA_1024X768)  
00154: static struct s5pv210fb_lcd lcd = {  
00155:     .width        = 1024,  
00156:     .height       = 768,  
00157:     .bpp          = 32,  
00158:     .freq         = 60,  
00159:  
00160:     .output       = S5PV210FR_OUTPUT_SVGA,
```

还有 VGA 输出到显示的参数

下面说说用 image2lcd 软件制作出的位图 logo 数据放在哪里？

下面既使用 Image2Icd 软件制作，在 u-boot/drivers/video/Logo.c 中

主要是修改这个数组，写入 logo 图片的长宽和图像数据

S5PV210 官方内核移植,编译

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel# ls
arch  COPYING.txt  Documentation  fs      initrd.img.cpio  kernel  Makefile  net      samples  sound  virt
block  CREDITS     drivers       include ipc      lib      mk      README    scripts  tools
COPYING  crypto     firmware     init      Kbuild   MAINTAINERS  MM      REPORTING-BUGS  security  usr
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel#
```

这就是 S5PV210 内核。

先去 Makefile 指定交叉编译工具链

```
ARCH      ?= arm
CROSS_COMPILE ?= /opt/arm-2009q3/bin/arm-none-linux-gnueabi-
```

然后 make defconfig, 但是我们不知道要 make 那一个 defconfig, 所以要去找

一般厂家都把 defconfig 放在 arch/arm/configs

```
: cd arch/arm/configs/
```

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel/arch/arm/configs# ls
210ii_initrd_defconfig  x210ii_qt_defconfig
```

我们发现我们要配置的是 x210ii_qt_defconfig 配置文件

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel# make x210ii_qt_defconfig
#
# configuration written to .config
#
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel#
```

只要出现 configuration written 这句话,就证明配置成功

```
..          crypto
arch      Documentation
block     drivers
.config   firmware
.config.old  fs
COPYING   .gitignore
```

在顶层目录下会出现 .config 文件,有了这个文件才能进行下一步

执行 make menuconfig 出现配置菜单

menuconfig 配置菜单要依赖 ncurses 库, 所以要按照 ncurses 库。

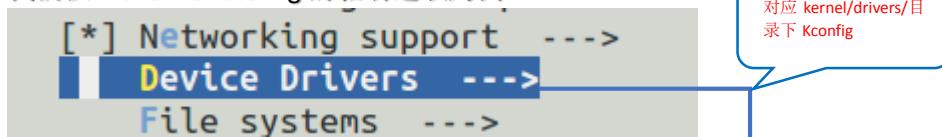
```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/kernel/arch/arm/boot# ls
bootp  compressed  Image  install.sh  Makefile  test  test2  zImage
```

有些平台编译出来的内核 zImage 文件在当前目录下, 有些平台编译出的 zImage 存放的目录不一样, 我这个 S5PV210 指定的 zImage 存放在 arch/arm/boot 目录下。

Linux 内核移植原理

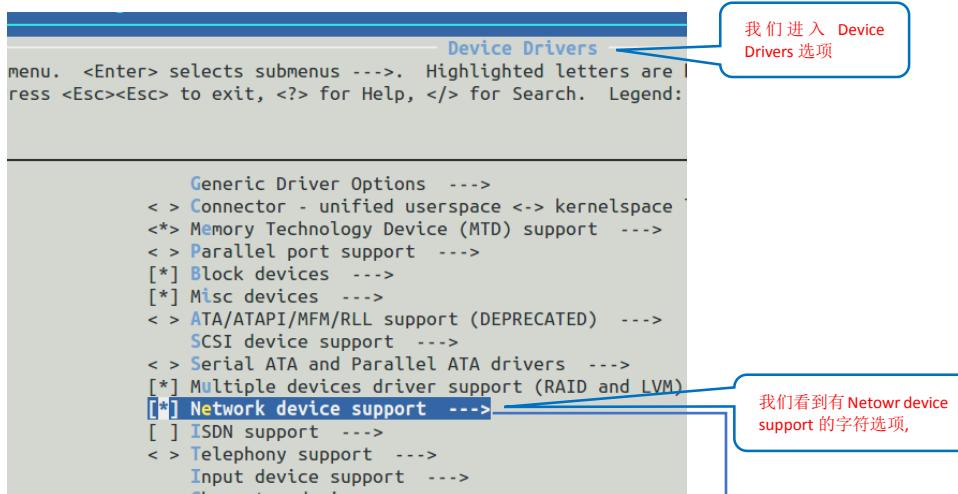
Kconfig 文件的格式详解

我们以 make menuconfig 的驱动选项为例



```
1 menu "Device Drivers"
2
3 source "drivers/base/Kconfig"
4
5 source "drivers/connector/Kconfig"
6
7 source "drivers/mtd/Kconfig"
```

Source 表示点击了 Device Drivers 选项后要引入哪些目录下的 Kconfig 文件



我们点击 Device Drivers 后引入了其它目录的 Kconfig 文件里面的 menuconfig 定义的选项
这个字符选项就是来自 kernel/drivers/net 目录下的 Kconfig

```
5 menuconfig NETDEVICES
6   default y if UML
7   depends on NET
8   bool "Network device support" ←
9   ---help---
10  You can say N here if you don't intend to connect your Linux box to
11    any other computer at all.
```

kernel/drivers/net 目录下的 Kconfig 文件会实现 menuconfig 很多的层级选项，而不是上面的一个选项代表一个目录，而是一个目录下有很多选项，每个选项代表一个 c 文件。

menuconfig 关键字表示定义的选项是一个大菜单

```

5 menuconfig NETDEVICES
6 |     default y if UML
7 |     depends on NET
8 |     bool "Network device support"
9 |     ---help---
10 |         You can say N here if you don't intend to connect your Linux box to
11 |         any other computer at all.
12 |
13 |         You'll have to say Y if your computer contains a network card that
14 |         you want to use under Linux. If you are going to run SLIP or PPP over
15 |         telephone line or null modem cable you need say Y here. Connecting
16 |         two machines with parallel ports using PLIP needs this, as well as
17 |         AX.25/KISS for sending Internet traffic over amateur radio links.
18 |
19 |         See also "The Linux Network Administrator's Guide" by Olaf Kirch and
20 |         Terry Dawson. Available at <http://www.tldp.org/guides.html>.
21 |
22 |         If unsure, say Y.
23 |
24 # All the following symbols are dependent on NETDEVICES - do not repeat
25 # that for each of the symbols.
26 if NETDEVICES
27
28 config IFB
29 |     tristate "Intermediate Functional Block support"
30 |     depends on NET_CLS_ACT
31 |     ---help---
32 |         This is an intermediate driver that allows sharing of

```

这就是菜单

config 关键字表示菜单中的配置项

```

Network device support
er> selects submenus --->. Highlighted letters ar
Esc to exit, <?> for Help, </> for Search. Legen

--- Network device support
<*>  Intermediate Functional Block support
< >  Dummy net driver support
< >  Bonding driver support
< >  MAC-VLAN support (EXPERIMENTAL)
< >  EQL (serial line load balancing) support
< >  Universal TUN/TAP device driver support
< >  Virtual ethernet pair device
< >  PHY Device support and infrastructure ...
[*]  Ethernet (10 or 100Mbit) --->
[*]  Ethernet (1000 Mbit) --->

```

这就表示在 Network device support 菜单下还有一个子菜单

Network device support 主菜单和 Ethernet(10 or 100Mbit)子菜单都在同一个 Kconfig 文件下

```

menuconfig NET_ETHERNET
187   bool "Ethernet (10 or 100Mbit)"
188   depends on !UML
189   ---help---
190     Ethernet (also called IEEE 802.3 or ISO 8802-2) is the most common
191     type of Local Area Network (LAN) in universities and companies.
192
193     Common varieties of Ethernet are: 10BASE-2 or Thinnet (10 Mbps over
194     coaxial cable, linking computers in a chain), 10BASE-T or twisted
195     pair (10 Mbps over twisted pair cable, linking computers to central
196     hubs), 10BASE-F (10 Mbps over optical fiber links, using hubs),
197     100BASE-TX (100 Mbps over two twisted pair cables, using hubs),
198     100BASE-T4 (100 Mbps over 4 standard voice-grade twisted pair
199     cables, using hubs), 100BASE-FX (100 Mbps over optical fiber links)
200     [the 100Base varieties are also known as Fast Ethernet], and Gigabit
201     Ethernet (1 Gbps over optical fiber or short copper links).
202
203     If your Linux machine will be connected to an Ethernet and you have
204     an Ethernet network interface card (NIC) installed in your computer,
205     say Y here and read the Ethernet-HOWTO, available from
206     <http://www.tldp.org/docs.html#howto>. You will then also have
207     to say Y to the driver for your particular NIC.
208
209     Note that the answer to this question won't directly affect the
210     kernel: saying N will just cause the configurator to skip all
211     the questions about Ethernet network cards. If unsure, say N.
212
213 if NET_ETHERNET
214
215 config MII
216 |     tristate "Generic Media Independent Interface device support"
217 |     help
218 |         Most ethernet controllers have MII transceiver either as an external
|           Ethernet (10 or 100Mbit)
|           ---> Generic Media Independent Interface device support
|               ASTX AX88796 NE2000 clone support

```

在同一个 Kconfig 文件下就可以实现菜单和配置项的关系，只需要将 config 关键字定义的字符串写在 menuconfig 关键字下面就行

在菜单下就是这样显示的

这里就告诉我们如果我们要自己创建了一个文件夹，那么要在这个文件夹下创建 Kconfig 文件，然后这个文件夹上一级目录里面的 Kconfig 文件要定义 source 关键字指定这个文件夹路径，这样你在 menuconfig 的时候才能在界面上看到你这个文件夹菜单。

下面我们来说说一个奇怪的菜单选择现象

```
Ethernet (10 or 100Mbit)
<Enter> selects submenus --->. Highlighted letters are hotkeys. Pre
sc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in

--- Ethernet (10 or 100Mbit)
-* Generic Media Independent Interface device support
< > ASIX AX88796 NE2000 clone support
< > SMC 91C9x/91C1xxx support
<*> DM9000 support
[*] DM9000 16-bit
(4) DM9000 maximum debug level
[ ] Force simple NSR based PHY polling
< > ENC28J60 support
< > OpenCores 10/100 Mbps Ethernet MAC support
< > SMSC LAN911[5678] support
< > SMSC LAN911x/LAN921x families embedded ethernet support
< > Dave ethernet support (DNET)
< > Broadcom 440x/47xx ethernet support
< > Micrel KSZ8842
< > Micrel KS8851 SPI
< > Micrel KS8851 MLL
```

```
185
186 menuconfig NET_ETHERNET
187 |     bool "Ethernet (10 or 100Mbit)"
188 |     depends on !UML
189 |     ---help---
190 |         Ethernet (also called IEEE 802.3 or ISO 8802-2) is the most common
191 |         type of local area network. It is a standard for physically connecting
192 |         computer systems together.
193 if NET_ETHERNET
194
195 config MII
196 |     tristate "Generic Media Independent Interface device support"
197 |     help
198
199 config MACB
200 |     tristate "Atmel MACB support"           这个选项在菜单里面怎么没有
201 |     depends on AVR32 || ARCH_AT91SAM9260 || ARCH_AT91SAM9263 || ARC
202 |     select PHYLIB
203 |     help
204
205 config MACE_AAUI_PORT
206 |     bool "Use AAUI port instead of TP by default"    这个选项在菜单里面怎么没有
207 |     depends on MACE
208 |     help
209
210 config DM9000
211 |     tristate "DM9000 support"
212 |     depends on ARM || BLACKFIN || MIPS
213 |     select CRC32
214 |     select MII
215 |     ---help---
216 |         Support for DM9000 chipset.
217 |
218 |         To compile this driver as a module, choose M here.
219 |         The module will be called dm9000.
220
221 config DM9000_16BIT           这个选项怎么
222 |     bool "DM9000 16-bit"                     在菜单就有了
223 |     depends on DM9000
224 |     default n
225 |     help
```

tristate 关键字,如果定义了这个关键字, 然后下面的什么都不定义, 那么菜单选项绝对是有的

depends 关键字, 在定义了 **tristate** 关键字后, 后面跟着定义了 **depends** 关键字, 就会造成这个菜单选项有可能有有可能没有。这是因为(**depends on**)表示该菜单选项依赖于其它菜单选项, 所以必须先把其它菜单选项选择了, 当前菜单选项才能显示出来。

下面以 DM9000 菜单选项为例

```

950 config DM9000
951     tristate "DM9000 support"
952     depends on ARM || BLACKFIN || MIPS
953     select CRC32
954     select MII
955     ---help---
956         Support for DM9000 chipset.
957
958         To compile this driver as a module, choose M here.
959         This will be called dm9000.
960
961 config DM9000_16BIT
962     bool "DM9000 16-bit"
963     depends on DM9000
964     default n
965     help

```

要选择 DM9000 16-bit
(depends on) 就要求先把
DM9000 support 先选择上

```

<--> ENC28J60 support
<-> DM9000 support
<-> OpenCores 10/100 Mbps Ethernet MAC support

```

我没有选择 DM9000 support, 导致 DM9000 16-bit 没法选上。

```

<*> DM9000 support
[*] DM9000 16-bit
(*) DM9000 maximum debug level

```

一旦我选择上 DM9000 support, 那么 DM9000 16-bit 就可以选择上了。

这就是 **depends on** 关键字的作用, 让当前选项依赖其它选项

如何根据 Kconfig 选项实现 C 文件被编译呢?

我们还是进入 kernel/drivers/net 目录, 找到 dm9000.c

```

dm9000.c
dm9000.h

```

如何编译进内核或者编译成模块

查看 Makefile

```

248 obj-$(CONFIG_SMC91X) += smc91x.o
249 obj-$(CONFIG_BFIN_MAC) += bfin_mac.o
250 obj-$(CONFIG_DM9000) += dm9000.o
251 obj-$(CONFIG_PASEMI_MAC) += psemi_mac.o

```

发现这一行有个 dm9000.o, 一个 c 文件放入 Makefile 编译就要用.o 来表示, 意思就是把它编译成.o 中间连接文件, 供其它程序调用。CONFIG_DM9000 的值=y, 那么编译器就会把 dm9000.c 编译成.o。怎么让 CONFIG_DM9000 = y 呢?, 看下面!!

Makefile 和 Kconfig 是怎么关联 dm9000 编译的

```

950 config DM9000
951     tristate "DM9000 support"
952     depends on ARM || BLACKFIN || MIPS
953     select CRC32
954     select MII
955     ---help---
956         Support for DM9000 chipset.

```

现在 Kconfig 定义一个
config DM9000, 这段话在
Makefile 中就会写成大写
的 CONFIG_DM9000,

所以 Kconfig 的小写 config DM9000 和 Makefile 中的 CONFIG_DM9000 是一一对应的。我在 Kconfig 的 menuconfig 菜单中选中 DM9000 support, 那么就会导致当前目录下对应的 Makefile 中的 CONFIG_DM9000 = y, 从而编译 dm9000.c 文件。

Linux 内核启动过程

内核顶层 Makefile 分析

```
1 VERSION = 2  
2 PATCHLEVEL = 6  
3 SUBLEVEL = 35  
4 EXTRAVERSION = .7  
5 NAME = yokohama
```

这是打印内核版本号，你可以任意修改，也可以就按标准来不修改

```
82 # 1) O=  
83 # Use "make O=dir/to/store/output/files/"  
84 #
```

你如果不直接 make 编译内核，而是 make O=... 传入内核参数，就是让内核编译出来的文件放在指定文件下，而不是放在内核目录默认文件夹下

```
191 ARCH | - ?= arm  
192 #CROSS_COMPILE | ?= /usr/local/arm/arm-none-linux-gnueabi-2010-09-50  
193 CROSS_COMPILE | ?= /opt/arm-2009q3/bin/arm-none-linux-gnueabi-  
194 # Architecture as present in compile.h
```

ARCH 设置平台名，CROSS_COMPILE 指定该 CPU 交叉编译器路径，前面经常用，这样才能编译。

Makefile 暂时分析完毕

链接脚本分析

链接脚本在 arch/arm/kernel 目录下 vmlinux.lds.S 文件

```
vmlinux.lds  
vmlinux.lds.S
```

```
11 OUTPUT_ARCH(arm)  
12 ENTRY(stext)  
13  
14 #ifndef __ARMEB__  
15 jiffies = jiffies_64;  
16 #else  
17 jiffies = jiffies_64 + 4;  
18 #endif  
19  
20 SECTIONS  
21 {  
22 #ifdef CONFIG_XIP_KERNEL  
23 . = XTR_VIRT_ADDR/CONTEXTR_PHYS_OFFSET
```

ENTRY 指定的 stext 就是入口的符号

这个 stext 入口就是指的在 arch/arm/kernel/ 目录下的 head.S 文件

```
14 #include <linux/linkage.h>  
15 #include <linux/init.h>  
16  
17 #include <asm/assembler.h>  
18 #include <asm/domain.h>  
19 #include <asm/ptrace.h>  
20 #include <asm/asm-offsets.h>  
21 #include <asm/memory.h>  
22 #include <asm/thread_info.h>  
23 #include <asm/system.h>  
24  
25 #if (PHYS_OFFSET & 0x001fffff)  
26 #error "PHYS_OFFSET must be at an even 2MiB boundary!"  
27 #endif  
28  
29 #define KERNEL_RAM_VADDR | (PAGE_OFFSET + TEXT_OFFSET)  
30 #define KERNEL_RAM_PADDR | (PHYS_OFFSET + TEXT_OFFSET)
```

KERNEL_RAM_VADDR 表示内核运行时的虚拟地址

KERNEL_RAM_PADDR 表示内核运行时的物理地址

```
#define KERNEL_RAM_VADDR (PAGE_OFFSET + TEXT_OFFSET)
```

这个虚拟地址的 PAGE_OFFSET 值在什么地方呢？

其实是在 arch/arm/include/asm 目录下的 Memory.h 文件

```
00027: #ifdef CONFIG_MMU
00028:
00029: /*
00030: * PAGE_OFFSET - the virtual address of the start of the kernel image
00031: * TASK_SIZE - the maximum size of a user space task.
00032: * TASK_UNMAPPED_BASE - the lower boundary of the mmap VM area
00033: */
00034: #define PAGE_OFFSET UL(CONFIG_PAGE_OFFSET)
00035: #define TASK_SIZE (UL(CONFIG_PAGE_OFFSET) - UL(0x01000000))
00036: #define TASK_UNMAPPED_BASE (UL(CONFIG_PAGE_OFFSET) / 3)

这个宏是在.config 里面定义的 CINFIG_MMU=y
这个 CONFIG_PAGE_OFFSET 也是在.config 里面定义的
371 CONFIG_PAGE_OFFSET=0xC0000000
372 # CONFIG_PREEMPT_NONE is not set
```

这个 TEXT_OFFSET 宏是在什么地方定义的呢？

经过搜索发现 TEXT_OFFSET = 0x0000 8000

所以最后虚拟地址是 KERNEL_RAM_VADDR = 0xC000 8000

```
#define KERNEL_RAM_PADDR (PHYS_OFFSET + TEXT_OFFSET)
```

这个 KERNEL_RAM_PADDR 物理地址是多少呢？

```
PHYS_OFFSET - 51 locations
# 1 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 2 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 3 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 4 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 5 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 6 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 7 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 8 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 9 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 10 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma
# 11 PHYS_OFFSET - Constant in Memory.h [android_kernel_2.6.35_smdkv210...\ma

00003: */
00004:
00005: #ifndef __ASM_ARCH_MEMORY_H
00006: #define __ASM_ARCH_MEMORY_H
00007:
00008: #define PHYS_OFFSET UL(0x00008000)

我们搜索发现很多平台都有这个物理地址宏，那我们就找我们平台目录下的这个 PHYS_OFFSET 宏就是了
```

在 arch/arm/mach-s5pv210/include/mach 目录下 memory.h 文件中找到

```
00016: #if defined(CONFIG_MACH_SMDKV210)
00017: #define PHYS_OFFSET UL(0x30000000)
00018: #else
00019: #define PHYS_OFFSET UL(0x30000000)
00020: #endif
```

在.config 文件中定义了 CONFIG_MACH_SMDKV210=y

那么 KERNEL_RAM_PADDR 物理地址就是 0x3000 8000

```
x210 # print
bootcmd=movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000
mtddpart=80000 400000 3000000
bootdelay=3
baudrate=115200
ethaddr=00:40:5c:26:0a:5b
ipaddr=192.168.1.88
serverip=192.168.1.102
gatewayip=192.168.0.1
netmask=255.255.0.0
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192.168.0.1
rc console=ttySAC2,115200

Environment size: 440/16380 bytes
```

所以我们 uboot 加载内核是加载在 0x30008000，启动内核也是从 0x30008000 地址启动

```

76 |
77 |     _HEAD
78 ENTRY(stext)
79 |     setmode|PSR_F_BIT | PSR_I_BIT | SVC_MODE, r9 @ ensure svc mode
80 |             |         |         |         |         |         |         |
81 |             mrc | p15, 0, r9, c0, c0|         |         |         |         |
82 |             bl  | _lookup_processor_type|         |         |         |         |
83 |             movs| r10, r5|         |         |         |         |
84 |             beq | _error_p|         |         |         |         |
85 |             bl  | _lookup_machine_type|         |         |         |
86 |             movs| r8, r5|         |         |         |         |
87 |             beq | _error_a|         |         |         |         |
88 |             bl  | __vet_atags|         |         |         |         |
89 |             bl  | __create_page_tables|         |         |         |
90 |
91 |     /*
92 |     * The following calls CPU specific code in a position independent

```

这就是内核起始代码入口

调用另一个汇编文件的函数，函数名叫 look up....

读取 CPU 的 ID 号码

其实内核一定要使用 CPU 官方的版本，如果要更新内核也要确认 CPU 官方是否更新了内核。因为你去开源社区移植的内核到官方 CPU，会出现很多驱动不一致的问题。

所以内核移植暂时不讲，就用官方的就是了。

Linux 内核打补丁

内核补丁是用来解决现在内核版本的漏洞问题。比如现在开发板有个内核，但是该内核的网卡驱动有漏洞，或者网络协议有漏洞，那么在不修改内核版本的情况下，就给内核打补丁来解决。

打补丁说白了就是修改内核中的 C 文件

单个文件补丁制作

```

root@ubuntu:/home
main1.c  main2.c
root@ubuntu:/home

```

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("main1111111\n");
6     return 0;
7 }

```

创建 main1.c 文件

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("main1111111\n");
6     printf("main2222222\n");
7     return 0;
8 }

```

创建 main2.c 文件,main2.c 文件的代码是在

main1.c 文件上面修改的，增加了新的语句。

这时候我不想破坏 main1.c 的原本代码，那么我就将 main2.c 的代码做出补丁打入 main1.c

```

root@ubuntu:/home/xiang/xpatch# ls
main1.c  main2.c
root@ubuntu:/home/xiang/xpatch#

```

直接用 diff 命令制作补丁。diff 格式(diff -uN 原本的文件 修改后的文件 > 补丁名称.patch)

```

root@ubuntu:/home/xiang/xpatch# ls
main1.c  main2.c  main.patch

```

main.patch 就是补丁文件，打开看看

```

1 --- main1.c      2019-04-02 23:32:42.934993306 -0700
2 +++ main2.c      2019-04-02 23:33:27.546992298 -0700
3 @@ -3,5 +3,6 @@
4     int main()
5     {
6         printf("main11111111\n");
7 +        printf("main22222222\n");
8         return 0;
9     }

```

```
root@ubuntu:/home/xiang/xpatch# patch -p0 < main.patch
patching file main1.c
```

patch 命令格式 `patch p0 < 补丁文件名.patch`。p0 可以是 p1, p2, p3.....

以上是在 `main1.c` 和 `main2.c` 同一级目录下面执行的 `patch`, 操作正确

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("main11111111\n");
6     printf("main22222222\n");
7     return 0;
8 }
```

你看我的 `main1.c` 文件被修改了增加了

`printf("main22222222\n")`, 这个 `main22222222` 是 `main2.c` 文件里面的。

打补丁的好处就是如果你觉得你的代码有问题可以用`-R`取消, 让你有后悔的机会

```
root@ubuntu:/home/xiang/xpatch# patch -RE -p0 < main.patch
patching file main1.c
```

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("main11111111\n");
6     return 0;
7 }
```

你看 `main1.c` 又回到原本代码了

这就是打补丁的好处, 让你修改源文件的同时在 `patch` 里面帮你备份了一份, 这样比直接修改 C 代码原文件安全多了, 当然你觉得牛逼你可以直接去修改 C 文件源代码, 或者复制粘贴一部分出来再修改。其实打补丁方式和直接修改 C 代码方式, 目的都是一样的。

下面介绍下多个目录里面打补丁的方法, 这个比单文件打补丁麻烦一点

`prj1 prj2`

root@ubuntu:~# 创建两个目录

```
root@ubuntu:/home/xiang/xpatch/prj1# ls
test1.c
```

prj1 目录创建文件 test1

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("test111111\n");
6     return 0;
7 }
```

test1 文件内容

```
root@ubuntu:/home/xiang/xpatch/prj2# ls
test2.c
```

prj1 目录创建文件 test2

```

1 #include <stdio.h>
2
3 int main()
4 {
5     printf("test222222\n");
6     return 0;
7 }
```

test2 文件内容

现在用 **diff** 创建补丁，一定要注意 **diff** 创建补丁所在的目录。因为这个涉及到用 **patch** 打补丁时 **p** 的参数设置

```
root@ubuntu:/home/xiang/xpatch# ls  
prj1 prj2  
root@ubuntu:/home/xiang/xpatch# diff -uN prj1/test1.c prj2/test2.c > xzz.patch  
1 --- prj1/test1.c| 2019-04-03 01:43:34.242824350 -0700  
2 +++ prj2/test2.c| 2019-04-03 01:43:34.242824350 -0700  
3 @@ -2,7 +2,7 @@  
4  
5 int main()  
6 {  
7 -    printf("test11111\n");  
8 +    printf("test22222\n");  
9 |    return 0;  
10 }  
11  
root@ubuntu:/home/xiang/xpatch# ls  
prj1 prj2 xzz.patch  
root@ubuntu:/home/xiang/xpatch# vim xzz.patch  
root@ubuntu:/home/xiang/xpatch# patch -p0 < xzz.patch  
patching file prj1/test1.c  
  
1 #include <stdio.h>  
2  
3 int main()  
4 {  
5     printf("test22222\n");  
6     return 0;  
7 }  
8
```

test1 文件的代码修改成功。

做补丁都需要指定具体文件名，所以在多目录情况下要把做补丁的文件路径写全

删除 test1 的 test11111 代码，添加 test2 的 test22222 代码，但是为什么会出现删除 test1 的代码呢？我单文件测试都没有遇到这个情况，这是因为 test1 和 test2 的 printf 代码都在 c 文件同一行，所以一定要注意

这个 **patch p0,p1,p2** 的选项问题是和在哪一级目录创建的 **patch** 补丁文件有关，自行百度。

根文件系统制作成镜像

根文件系统格式有， ext2, ext3, ext4, jffs2, fat32。

ext3 文件系统制作

Ubuntu 默认就有一个 mke2fs 软件，这个软件就是给文件系统制作 ext2, ext3, ext4 镜像的。比如制作 ext2 就要用 mkfs.ext2 软件

```
root@ubuntu:/home/xiang/S5PV210/noos_driver/s_driver# MKFS.  
mkfs.bfs  mkfs.cramfs  mkfs.ext2  mkfs.ext3  mkfs.ext4  mkfs.ext4dev  mkfs.fat  mkfs.minix  mkfs.msdos  mkfs.ntfs  mkfs.vfat
```

你看 ubuntu 下面有很多文件系统制作工具。既然有 mkfs.ext3, 为什么还要用 mke2fs 呢？

```
noOS_driver/s_driver# ls -l /sbin/mkfs.ext3  
2016 /sbin/mkfs.ext3 -> mke2fs
```

因为 mkfs.ext3 是 mke2fs 的软链接

我们来创建文件系统

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# dd if=/dev/zero of=rootfs.ext3 bs=1024 count=2048  
2048+0 records in  
2048+0 records out  
2097152 bytes (2.1 MB) copied, 0.0178431 s, 118 MB/s
```

先用 dd 命令创建一个 ext3 的文件系统，该系统里面全部为 0

```
untu:/home/xiang/S5PV210/rootfsext3# ls  
rootfs.ext3
```

这个 rootfs.ext3 就是创建的文件系统镜像，我们知道不能直接向镜像文件里面写数据，或者创建目录什么的，所以我们要用另外一种方法间接的向镜像里面创建文件，写数据

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# ls  
rootfs->rootfs.ext3
```

我们创建一个 rootfs 文件夹，我们向 rootfs 文件夹写东西，就等于向 rootfs.ext3 写东西。

现在我们要将 rootfs 文件夹和 rootfs.ext3 关联起来，这样 rootfs.ext3 才知道我们写入的东西从哪个目录来。

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# losetup /dev/loop1 rootfs.ext3
```

使用 losetup 将 rootfs.ext3 磁盘镜像文件虚拟成快设备。用 /dev/loop1 来模拟该块设备

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mke2fs -m 0 /dev/loop1 2048  
mke2fs 1.42.9 (4-Feb-2014)  
Discarding device blocks: done  
Filesystem label=  
OS type: Linux  
Block size=1024 (log=0)  
Fragment size=1024 (log=0)  
Stride=0 blocks, Stripe width=0 blocks  
256 inodes, 2048 blocks  
0 blocks (0.00%) reserved for the super user  
First data block=1  
Maximum filesystem blocks=2097152  
1 block group  
8192 blocks per group, 8192 fragments per group  
256 inodes per group  
  
Allocating group tables: done  
Writing inode tables: done  
Writing superblocks and filesystem accounting information: done
```

这里必须是三个
done 才算创建成功

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mount -t ext3 /dev/loop1 ./rootfs
mount: wrong fs type, bad option, bad superblock on /dev/loop1,
      missing codepage or helper program, or other error
      In some cases useful info is found in syslog - try
      dmesg | tail or so
```

将代表 rootfs.ext3 的虚拟文件系统 loop1 挂载到 rootfs 文件夹上失败。

失败的原因在于前面创建文件系统用的 mke2fs 这里

这个命令写法有问题

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mke2fs -m 0 /dev/loop1 2048
mke2fs 1.42.9 (4-Feb-2014)
```

我们将 mke2fs 命令换成 mkfs.ext3

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mkfs.ext3 /dev/loop1
mke2fs 1.42.9 (4-Feb-2014)
Discarding device blocks: done
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
Stride=0 blocks, Stripe width=0 blocks
256 inodes, 2048 blocks
102 blocks (4.98%) reserved for the super user
First data block=1
Maximum filesystem blocks=2097152
1 block group
8192 blocks per group, 8192 fragments per group
256 inodes per group

Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done

root@ubuntu:/home/xiang/S5PV210/rootfsext3# mount -t ext3 /dev/loop1 ./rootfs
```

像这样不加参数直接创建指定的 loop1 为 ext3 文件系统

将代表 rootfs.ext3 的虚拟文件系统 loop1 挂载到 rootfs 文件夹上，成功

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mount -t ext3 /dev/loop1 ./rootfs
root@ubuntu:/home/xiang/S5PV210/rootfsext3# ls
rootfs  rootfs.ext3
root@ubuntu:/home/xiang/S5PV210/rootfsext3# cd rootfs
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# ls
```

你进入 rootfs 就能看到 ext3 文件系统

其实就把 rootfs.ext3 放到了 rootfs 文件夹中，这样你可以在 rootfs 文件夹创建任何目录文件，系统会自动帮你把你创建的目录和文件赋值给 rootfs.ext3

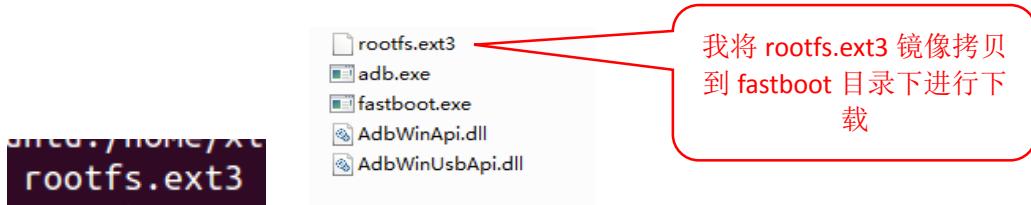
```
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# ls
linuxrc  lost+found
```

我在 rootfs 文件中创建了一个 linuxrc 的文件，这个文件没有任何东西。

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# umount /dev/loop1
root@ubuntu:/home/xiang/S5PV210/rootfsext3# losetup -d /dev/loop1
root@ubuntu:/home/xiang/S5PV210/rootfsext3# cd rootfs
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# ls
```

然后我在 rootfs 上级目录下，取消 rootfs.ext3 和 rootfs 文件的相互挂载

我再进入 rootfs 文件里面，发现什么都没有了，但是 linuxrc 已经在 rootfs.ext3 文件里面了



```
E:\Desktop\fastboot>fastboot devices
SMDKC110-01      fastboot

E:\Desktop\fastboot>fastboot flash system rootfs.ext3
sending 'system' <2048 KB>... OKAY
writing 'system'... OKAY
```

在 fastboot 目录下用我们 fastboot 命令下载文件系统到 Inand，这样就完成了文件系统下载然后重启开发板

```
[ 5.588266] Kernel panic - not syncing: No init found. Try passing init= option to kernel. See Linux Documentation/init.txt for guidance.
[ 5.600604] Backtrace:
[ 5.603015] [<c0034fb8>] (dump_backtrace+0x0/0x110) from [<c0501498>] (dump_stack+0x18/0x1c)
[ 5.611557] r6:c0058894 r5:c00083d4 r4:c06f5b6c r3:00000002
[ 5.617040] [<c0501480>] (dump_stack+0x0/0x1c) from [<c0501514>] (panic+0x78/0xf8)
[ 5.624606] [<c050149c>] (panic+0x0/0xf8) from [<c00305e4>] (init_post+0xb4/0xdc)
[ 5.632046] r3:dfc34000 r2:dff2f2f00 r1:00000000 r0:c0646222
[ 5.637666] [<c0030530>] (init_post+0x0/0xdc) from [<c0008500>] (kernel_init+0x12c/0x170)
[ 5.645824] r4:c0731500 r3:df801090
[ 5.649368] [<c00083d4>] (kernel_init+0x0/0x170) from [<c0058894>] (do_exit+0x0/0x5f0)
[ 5.657262] r4:00000000 r3:00000000
[ 5.660801] Rebooting in 5 seconds... 发现在这里文件系统崩溃了
```

因为我们创建的 ext3 文件系统什么东西没有向系统里面加，我们只是验证内核是否能正常挂载我们创建的 ext3 文件系统。

```
[ 0.000000] Kernel command line: console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3
```

先查看打印的 kernel command 输出。

Log 输出用串口 2

内核支持文件系统格式为 ext3

文件系统烧写在 Inand 的设备节点

VFS 输出这句话证明文件系统 ext3 挂载没有问题

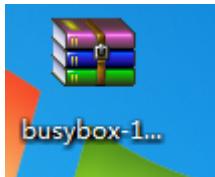
但是执行 ext3 文件系统里面的 linuxrc 出现错误

出现错误很正常，我们的 linuxrc 建立的是一个空文件。

证明我们创建 ext3 文件系统是没有问题的，只是我们还没有完善文件系统。

移植 busybox 文件系统

在 <https://busybox.net/downloads/> 地址下载 busybox



我下载的是 busybox-1.24.1.tar 版本

解压 busybox-1.24.1

```
root@ubuntu:/home/xiang/SSPV210/SSPV210-linux/rootfs/busybox-1.24.1# ls
applets      AUTHORS      coreutils    editors    int      LICENSE      Makefile.custom  modutils   qemu_multiarch_testing  selinux      TODO
applets_sh    Config.in    debianutils  examples   INSTALL   logutils     Makefile.flags  networking  README       shell       TODO_unicode
arch         config      docs        findutils  libbb    mailutils   Makefile.help   printutils  runit        sysklog     util-linux
archival     console-tools e2fsprogs  include   libpwdgrp Makefile   miscutils     procps     scripts    testsuite
```

1.先修改 Makefile

```
190 ARCH ?= $(SUBARCH)
191
192 # Architecture as present in compile.h
193 UTS_MACHINE := $(ARCH)
```

和 linux 内核一样修改这个 ARCH 改为
ARCH=arm

```
190 ARCH ?= $(SUBARCH)
191 ARCH = arm
192 # Architecture as present in compile.h
193 UTS_MACHINE := $(ARCH)
```

修改后 ARCH = arm

```
164 CROSS_COMPILE ?=
165 # bbox: we may have CONFIG CROSS_COMPILER_PREFIX in .config,
166 # and it has not been included yet... thus using an awkward syntax.
167 ifeq ($(CROSS_COMPILE),)
168 CROSS_COMPILE := $(shell grep ^CONFIG CROSS_COMPILER_PREFIX .config 2>/dev/null)
169 CROSS_COMPILE := $(subst CONFIG CROSS_COMPILER_PREFIX=,,$(CROSS_COMPILE))
170 CROSS_COMPILE := $(subst ",,$(CROSS_COMPILE))
171 #")
172 endif
173
174 # SUBARCH tells the usermode build what the underlying arch is. That is set
```

和 linux 内核一样指定交叉编译器路径

```
169 CROSS_COMPILE := $(subst CONFIG CROSS_COMPILER_PREFIX=,,$(CROSS_COMPILE))
170 CROSS_COMPILE := $(subst ",,$(CROSS_COMPILE))
171 #")
172 endif
173
174 CROSS_COMPILE = /opt/arm-2009q3/bin/arm-none-linux-gnueabi-
```

指定了交叉编译器路径

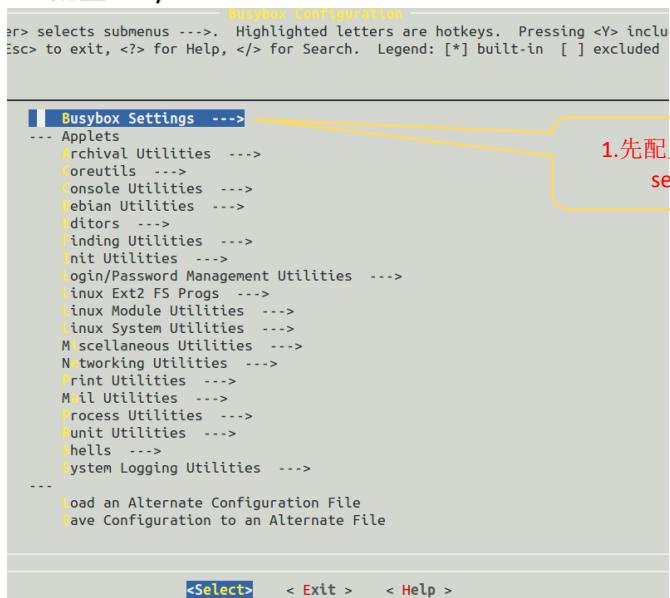
然后保存退出 Makefile

2.执行 make menuconfig 配置 busybox 文件系统的功能

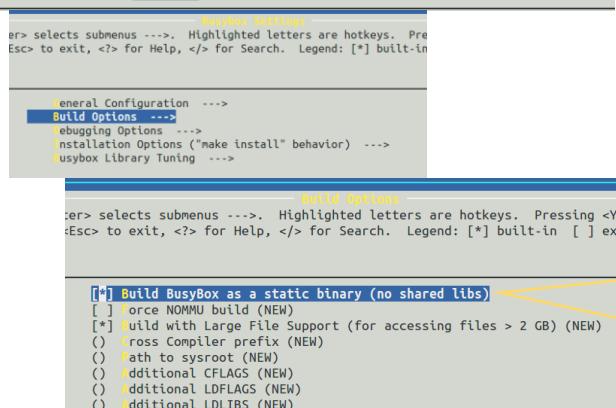


这就是 busybox 的配置菜单

2.1 配置 busybox

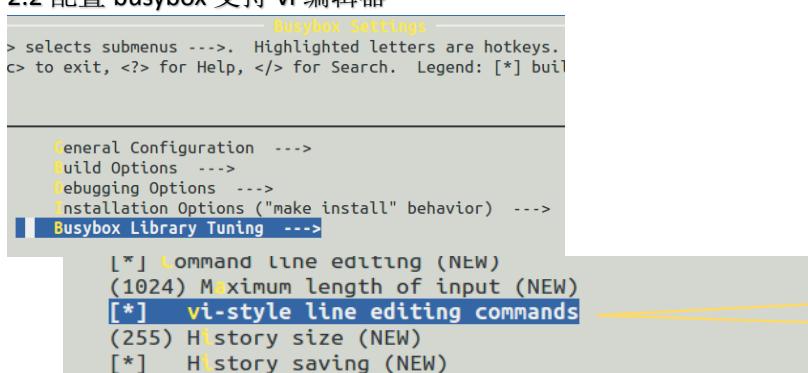


1.先配置 busybox settings



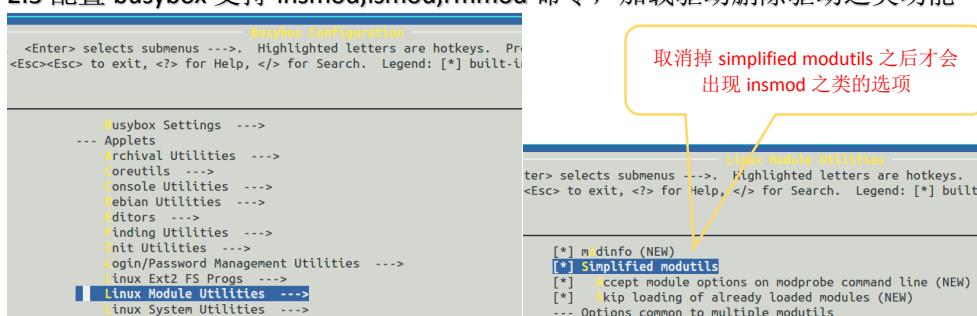
把 busybox 配置成静态的库文件系统，因为你开发板启动的时候，启动完 kernel，然后再启动 busybox 文件系统，在 busybox 还没启动起来的时候，哪里来的动态库，所以要配置成静态库。动态库是文件系统 busybox/Qt 启动完了才有的

2.2 配置 busybox 支持 vi 编辑器



支持 Vi 编辑器

2.3 配置 busybox 支持 insmod,lsmod,rmmmod 命令，加载驱动删除驱动之类功能



取消掉 simplified modutils 之后才会出现 insmod 之类的选项

```

Linux Module Utilities
<Enter> selects submenus -->. Highlighted letters are hotkeys. I
sc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] built
[*] modinfo (NEW)
[ ] Simplified modutils
[*] insmod
[*] rmmod
[*] lsmod
[*] pretty output
[*] modprobe
[*] blacklist support
[*] depmod
Options common to multiple modutils

```

把这几项全部选中就可以了

2.4 配置 busybox 支持 mdev,也就是 insmod 驱动后在/dev下自动生成设备节点

```

Busybox Configuration
<Enter> selects submenus -->. Highlighted letters are hotkeys. I
sc><Esc> to exit, <?> for Help, </> for Search. Legend: [*]
[*] Busybox Settings --->
--- Appslets
  Archival Utilities --->
  Coreutils --->
  Console Utilities --->
  Debian Utilities --->
  Editors --->
  Finding Utilities --->
  Init Utilities --->
  Login/Password Management Utilities --->
  Linux Ext2 FS Progs --->
  Linux Module Utilities --->
  Linux System Utilities --->
  [ ] Linux System Utilities --->

```

确认 mdev 选中就是了

```

Linux System Utilities
<Enter> selects submenus -->. Highlighted letters are
><Esc> to exit, <?> for Help, </> for Search. Legend:
[*] blockdev (NEW)
[*] fstatattr (NEW)
[*] fstrim (NEW)
[*] mdev (NEW)
[*] Support /etc/mdev.conf (NEW)
[*] Support subdirs/symlinks (NEW)

```

3.开始编译 busybox

保存配置好的菜单，直接 make

```

coreutils/lib.a(sync.o): In function `sync_main':
sync.c:(.text.sync_main+0x78): undefined reference to `syncfs'
collect2: ld returned 1 exit status
make: *** [busybox_unstripped] Error 1

```

编译过程中发现 sync.c 里面的 syncfs 函数没找到，但是我发现 sync 这个功能我现在用不到，那么我们找到 sync.c 文件修改它。

```

root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1# find -name "sync.c"
./coreutils/sync.c

```

find 找到了这个文件

我们在 coreutils 目录下没有找到 Makefile 文件，那么我们去 Kbuild 文件下去找

```

5 #
6 # Licensed under GPLv2, see file LICENSE in this source tree.
7
8 lib-y| | | += libcoreutils/
9
10 lib-y:=
11
12 lib-$(CONFIG_BASENAME) += basename.o
13 lib-$(CONFIG_CAT) += cat.o
14 lib-$(CONFIG_MORE) += cat.o # more uses it if stdout isn't a tty
15 lib-$(CONFIG_LESS) += cat.o # less too
16 lib-$(CONFIG_CRONTAB) += cat.o # crontab -l
17 lib-$(CONFIG_DATE) += date.o
18 lib-$(CONFIG_HEAD) += head.o
19 lib-$(CONFIG_HOSTID) += hostid.o
20 lib-$(CONFIG_GROUPS) += id.o
21 lib-$(CONFIG_ID) += id.o
22 lib-$(CONFIG_SHUF) += shuf.o
23 lib-$(CONFIG_SYNC) += sync.o

```

我在 Kbuild 文件下找到了 sync.o,
对应的配置项是 CONFIG_SYNC.
那我们去配置菜单修改
CONFIG_SYNC 选项

这个 Kbuild 也有点类似 Makefile 的功能，选择配置编译文件

执行 make menuconfig

```
Symbol: SYNC [=y]
Prompt: sync
Defined at coreutils/Config.in:126
Location:
-> Coreutils

Symbol: FEATURE_SYNC_FANCY [=y]
Prompt: Enable -d and -f flags (requires syncfs(2) in libc)
Defined at coreutils/Config.in:126
Location:
-> Coreutils
```

Coreutils

inter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in []

- [*] basename
- [*] cat
- [*] date
- [*] tnable ISO date format output (-I)
- [] Support %[num]N nanosecond format specifier
- [*] Support weird 'date MMDDhhmm[[YY]YY][.ss]' format
- [*] dd
- [*] tnable signal handling for status reporting
- [*] Enable the third status line upon signal
- [*] Enable ibs, obs and conv options
- [*] Enable status display options
- [*] hostname
- [*] id
- [*] groups
- [*] shuf
- [*] sync**

保存配置选项再次 make

```
Final link with: m
DOC    busybox.pod
DOC    BusyBox.txt
DOC    busybox.1
DOC    BusyBox.html
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1#
```

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1# ls
applets    busybox          configs      e2fsprogs  init       loginut
applets_sh  busybox_unstripped  console-tools editors   INSTALL  mailutil
arch        busybox_unstripped.map coreutils   examples libbb     Makefile
archival    busybox_unstripped.out debianutils findutils libpwdgrp Makefile
```

4. 执行 make install (busybox 默认安装在当前目录下的./install 目录中，但是有些软件必须要指定 prefix 安装目录，不然不小心把 ubuntu 根文件系统覆盖掉就完了)

```
inter> selects submenus --->. Highlighted letters are hotkeys. Pressing <Esc> to exit, <?> for Help, </> for Search. Legend: [*] built-in [ ]
```

```
What kind of applet links to install (as soft-links) --->
(../_install) busyBox installation prefix
```

```
./_install//usr/sbin/ubilinuxupdatevol -> ../../bin/busybox
./_install//usr/sbin/udhcpd -> ../../bin/busybox
```

```
You will probably need to make your busybox binary
setuid root to ensure all configured applets will
work properly.
```

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1# ls
bin  linuxrc  sbin  usr
```

我还是确认了一下 busybox 安装的位置。

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1/_install# ls
bin  linuxrc  sbin  usr
```

这就表示安装完成

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/busybox-1.24.1/_install#
```

在_install 目录下看到了最小的文件系统内容。但是没有动态库 lib 目录。

现在我这个文件系统在 `_install` 目录没有什么用，我要做文件系统的目录是我自己创建的 `rootfs` 目录，所以我要重新安装一次 `busybox`

我创建了一个做文件系统的目录。

再次 `make install`

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/xzzrootfs# ls  
bin  linuxrc  sbin  usr
```

这就对了

你可以把这个文件系统做出镜像下载进开发板调试，但是这种每次向文件系统加载一些功能很强大的程序库时，每次都要下载很麻烦。

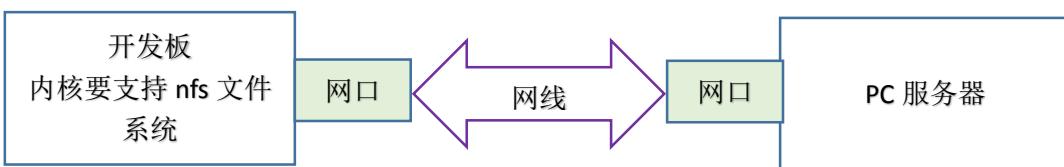
所以可以用 `tftp` 或者 `nfs` 来将文件系统挂载到板子上，这样你修改 PC 虚拟机 `ubuntu` 系统的 `nfs` 文件系统目录时同时就修改了开发板上的文件系统内容。记住 `nfs` 文件系统是在开发板的 `RAM` 中运行的。如果你要做成产品，还得把这个文件系统做出镜像烧写进开发板才行

下面我们先讲解 `nfs` 的使用方法，然后在 `nfs` 后面章节再讲解文件系统的细节部分。

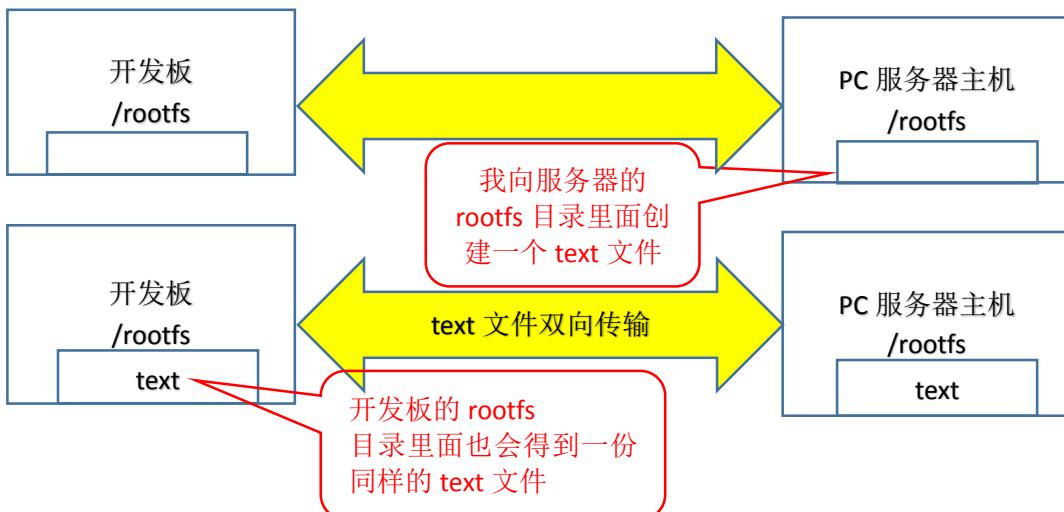
NFS 网络文件系统

NFS 有客户端和服务器组成

我们用开发板做 NFS 客户端，用电脑主机 ubuntu 系统来做服务器



nfs 文件系统的好处是什么？



这样我们就可以把驱动直接放在 `rootfs` 文件里面，然后开发板在 `rootfs` 文件里面调试，不用用 U 盘或者 SD 卡去插入开发板调试方便很多，而且也可以直接在 `rootfs` 里面调试文件系统的 lib 库或者一些程序，不用再去使用 fastboot 去下载调试后的文件系统镜像。

注意：nfs 文件系统只是适合在开发调试阶段使用，因为驱动和文件系统并没有存在在 emmc/iNAND 里面，所以开发板断点后就没有了，还有就是在调试的时候一定要保证你的网线网络正常。当我们调试完成觉得文件系统没有问题了，就要把该 nfs 的目录做出 ext3 下载进 emmc。

安装 PC 端 nfs 软件

```
sudo apt-get install nfs-kernel-server
```

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# sudo apt-get install nfs-kernel-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  keyutils libgssglue1 libnfsidmap2 libtirpc1 nfs-common rpcbind
Suggested packages:
  open-iscsi watchdog
The following NEW packages will be installed:
```

```
sudo apt-get install nfs-common
```

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# sudo apt-get install nfs-common
Reading package lists... Done
Building dependency tree
Reading state information... Done
nfs-common is already the newest version.
nfs-common set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 548 not upgraded.
root@ubuntu:/home/xiang/S5PV210/nfsrootfs#
```

nfsrootfs

我们在 ubuntu 上随便找个地方创建个目录来做 nfs 文件系统共享目录

然后我们要给 nfs 服务器指定这个 nfsrootfs 目录是我们要和开发板共享的目录

修改/etc(exports 文件，指定 nfsrootfs 目录路径

```
root@ubuntu:/# vim /etc/exports
```

添加这一行/home/xiang/S5PV210/nfsrootfs *(rw,sync,no_root_squash,no_subtree_check)

```
8 # /srv/nfs4      gss/krb5i(rw, sync, fsid=0, crossmnt, no_subtree_check)
9 # /srv/nfs4/homes gss/krb5i(rw, sync, no_subtree_check)
10 #
11 /home/xiang/S5PV210/nfsrootfs *(rw, sync, no_root_squash, no_subtree_check)
```

这就是指定我刚才建立的 nfsrootfs 目录路径

然后保存退出

```
# chmod 777 nfsrootfs
```

给这个 nfs 和开发板共享的目录最高权限

```
root@ubuntu:/home/xiang/S5PV210# sudo showmount -e
clnt_create: RPC: Program not registered
```

用 sudo showmount -e 发现 NFS 服务器没有启动，造成了 RPC 不能注册

```
root@ubuntu:/home/xiang/S5PV210# /etc/init.d/nfs-kernel-server status
nfsd not running
```

检查 init.d/nfs-kernel-server status 发现 nfs 服务器没有运行起来

```
root@ubuntu:/home/xiang/S5PV210# /etc/init.d/nfs-kernel-server start
* Exporting directories for NFS kernel daemon...
* Starting NFS kernel daemon
```

启动 nfs 服务器

```
root@ubuntu:/home/xiang/S5PV210# /etc/init.d/nfs-kernel-server status
nfsd running
```

检查 nfs 服务器启动起来了

```
root@ubuntu:/home/xiang/S5PV210# sudo exportfs -r
root@ubuntu:/home/xiang/S5PV210# sudo showmount localhost -e
Export list for localhost:
/home/xiang/S5PV210/nfsrootfs *
```

再次用 sudo exportfs -r 更新/etc/exportfs 文件

然后就看到 nfsrootfs 目录注册到 nfs 服务器了

```
root@ubuntu:/home/xiang/S5PV210# /etc/init.d/nfs-kernel-server restart
* Stopping NFS kernel daemon
* Unexporting directories for NFS kernel daemon...
* Exporting directories for NFS kernel daemon...
* Starting NFS kernel daemon
root@ubuntu:/home/xiang/S5PV210#
```

再次重启 nfs 服务器，PC 端 nfs 服务器搭建完成

下面我们在 PC 下建立另外一个目录来测试下 nfs 服务器是否能挂载上 nfsrootfs 目录

在本地任意地方创建一个目录

```
root@ubuntu:/home/xiang/S5PV210# ls
dnw  nfsrootfs  nfstest  noOS_driver  rootfsext3
root@ubuntu:/home/xiang/S5PV210#
```

我们将 nfstest 目录挂载到 nfsrootfs 目录下，看两个目录是不是共享的

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ifconfig  
eth0      Link encap:Ethernet HWaddr 00:0c:29:c4:2e:  
          inet addr:192.168.14.12 Bcast:192.168.14.255 M:  
          inet6 addr: fe80::20c:29ff:fe42:e/64 Scope:Link  
          brd fe80::ff0c:29ff:fe42:e
```

Ifconfig 查看下 PC 端主机 IP 地址

```
root@ubuntu:/home/xiang/S5PV210# mount -t nfs -o noblock 192.168.14.12:/home/xiang/S5PV210/nfsrootfs nfstest/
```

```
mount -t nfs -o noblock 192.168.14.12:/home/xiang/S5PV210/nfsrootfs nfstest/
```

我 PC 服务器的 IP 地址
和共享的文件目录

这是本机上另外一个目
录，只是为了测试下 nfs
链接是否正常，开发板
不是这个目录

注意不要去挂载/mnt 目录，因为这个目录有其他用处
挂载/mnt 目录容易出问题

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls  
linuxrc  text1  text2  text3
```

我在 NFS 服务器目录下创建了三个 text 文件

我们看看另外一个 nfstest 目录是否也有这三个文件

```
root@ubuntu:/home/xiang/S5PV210/nfstest# ls  
linuxrc  text1  text2  text3
```

你看 nfstest 目录也同时出现了三个文件，证明 nfs 服务器挂载成功。

下面我们把 nfstest 目录改成在开发板上使用的目录

```
root@ubuntu:/home/xiang/S5PV210# umount nfstest
```

我们挂载开发板目录之前我们要先用 umount 把 PC 端做挂载实验的 nfstest 目录卸载掉
这样我们的 nfsrootfs 目录才可以去挂载开发板上的目录

```
Serial = CLKUART  
Board: X210  
DRAM: 512 MB  
Flash: 8 MB  
SD/MMC: 3728MB  
*** Warning - using default environment  
  
In:     serial  
Out:    serial  
Err:    serial  
[LEFT UP] boot mode  
checking mode for fastboot ...  
Hit any key to stop autoboot: 0  
x210 #
```

开发板要在 uboot 下面设置 PC 端的 nfs 服务器地址和路径

```
[ 0.000000] Kernel command line: console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/li  
nuxrc rootfstype=ext3
```

修改开发板上的
bootargs

调试串口不变接
口不变

nfs 文件系统是网络文件系统，不是存储在
EMMC 里面的 ext3 文件系统，所以这里要变

```
[ 0.000000] Kernel command line: console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext3
```

Linuxrc 是在 nfs 网络文件系统里面的，不是在 ext3 文件系统里面，所以这里要变

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.141:/xiang/home/nfsrootfs  
ip=192.168.1.10:192.168.1.141:192.168.1.1:255.255.255.0::eth0:off init=/linuxrc  
console=ttySAC2,115200
```

uboot 指定设备为 nfs 设备

串口不变

PC 端主机网卡节点

关闭开发板 dhcp 自动分配 IP 功能

指定 PC 机的主机 IP 和主机上面我建立的 nfsrootfs 目录路径，这个要改
给开发板设定死 IP 地址

PC 端主机 IP 地址

设置网段

子网掩码

文件系统第一个进程执行 linuxrc 文件

```
root@ubuntu:/home/xiang/S5PV210# ifconfig  
eth0      Link encap:Ethernet HWaddr 00:0c:29:cf:c4:2e  
          inet addr:192.168.14.12 Bcast:192.168.14.255 Mask:255.255.255.0  
          inet6 addr: fe80::20c:29ff:fecc:c42e/64 Scope:Link
```

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls  
linuxrc
```

根据我主机的网段和文件路径，uboot 的 bootargs 该如何设置

```
setenv bootargs root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs  
ip=192.168.14.25:192.168.14.12:192.168.14.1:255.255.255.0::eth0:off init=/linuxrc  
console=ttySAC2,115200
```

注意以上字段的空格和分割符号

```
*** Warning - using default environment  
  
In:      serial  
Out:     serial  
Err:     serial  
[LEFT UP] boot mode  
checking mode for fastboot ...  
Hit any key to stop autoboot: 0  
x210 # setenv bootargs root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192  
.168.14.25:192.168.14.12:192.168.14.1:255.255.255.0::eth0:off init=/linuxrc console=ttySAC2,11  
5200  
x210 #
```

我们把 bootargs 写进去了

```
x210 # print  
bootcmd=movi read kernel 30008000; movi read rootfs 30B00000 300000; bootm 30008000 30B00000  
mtddpart=80000 400000 3000000  
bootdelay=3  
baudrate=115200  
ethaddr=00:40:5c:26:0a:5b  
ipaddr=192.168.1.88  
serverip=192.168.1.102  
gatewayip=192.168.0.1  
netmask=255.255.0.0  
bootargs=root=/dev/nfs nfsroot=192.168.14.12:/home/xiang/SPV210/nfsrootfs ip=192.168.14.25:192  
.168.14.12:192.168.14.1:255.255.255.0::eth0:off init=/linuxrc console=ttySAC2,115200  
  
Environment size: 440/16380 bytes  
x210 # save  
Saving Environment to SMDK bootable device...  
done  
x210 #
```

我们用 print 看看不是写进去了

用 save 保存

显示 done 保存成功

再次确认 bootargs 配置正确

```
x210 # print
mtddpart=80000 400000 3000000
baudrate=115200
ethaddr=00:40:5c:26:a5:b
netmask=255.255.0.0
bootdelay=3
gatewayip=192.168.1.14
serverip=192.168.1.12
bootcmd=movi read kernel 0x30008000;bootm 0x30008000
bootargs=root=/dev/nfs nfsroot=192.168.1.103:/home/xiang/S5PV210/nfsrootfs ip=192.168.1.10:192.168.1.103:192.168.1.1:255.255.255.0::eth0:off init=/sbin/init
linuxrc console=ttySAC2,115200
ipaddr=192.168.1.10
Environment size: 400/16380 bytes
x210 # setenv serverip 192.168.1.103
x210 # setenv gatewayip 192.168.1.1
x210 # save
Saving Environment to SMDK bootable device...
done
x210 #

x210 # print
mtddpart=80000 400000 3000000
baudrate=115200
ethaddr=00:40:5c:26:a5:b
netmask=255.255.0.0
bootdelay=3
bootcmd=movi read kernel 0x30008000;bootm 0x30008000
bootargs=root=/dev/nfs nfsroot=192.168.1.103:/home/xiang/S5PV210/nfsrootfs ip=192.168.1.10:192.168.1.103:192.168.1.1:255.255.255.0::eth0:off init=/sbin/init
linuxrc console=ttySAC2,115200
ipaddr=192.168.1.10
serverip=192.168.1.103
gatewayip=192.168.1.1
Environment size: 399/16380 bytes
x210 #
```

重启开发板

```
[ 5.579355] VFS: Cannot open root device "nfs" or unknown-block(0,255)
```

发现挂载 nfs 文件系统失败

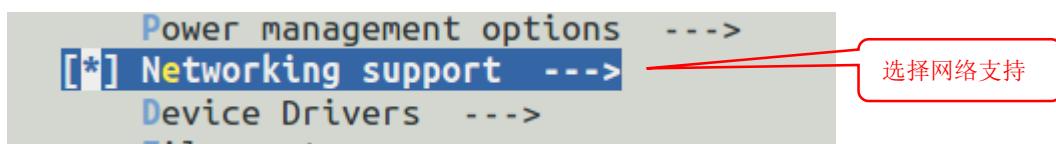
```
[ 5.579355] VFS: Cannot open root device "nfs" or unknown-block(0,255)
[ 5.584497] Please append a correct "root=" boot option; here are the available partitions:
```

而且后面输出会打印内核输出报错，这是为什么呢？

它要求你给一个正确的 root 路径，你这个 setenv bootargs root=/dev/nfs 系统根本不认识 /dev/nfs，这个问题就是内核没有配置支持 nfs 方式启动的功能

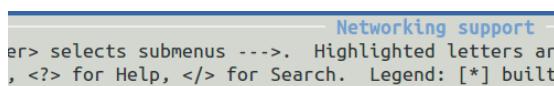
```
lmg.cpio lib
MAINTAINERS modu
Makefile modu
mk Modu
/kernle# make menuconfig
```

回到内核顶层目录，执行菜单选项



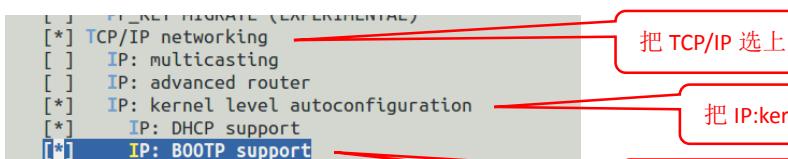
```
Power management options --->
[*] Networking support --->
Device Drivers --->
```

选择网络支持



```
Networking support
--- Networking support --->
[*] Networking options --->
[ ] Amateur Radio support --->
```

选择网络选项



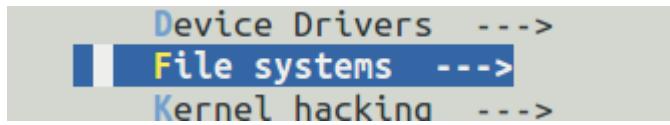
```
[ ] *-NET_MIGRATE (EXPERIMENTAL)
[*] TCP/IP networking
[ ] IP: multicasting
[ ] IP: advanced router
[*] IP: kernel level autoconfiguration
[*] IP: DHCP support
[*] IP: BOOTP support
```

把 TCP/IP 选上

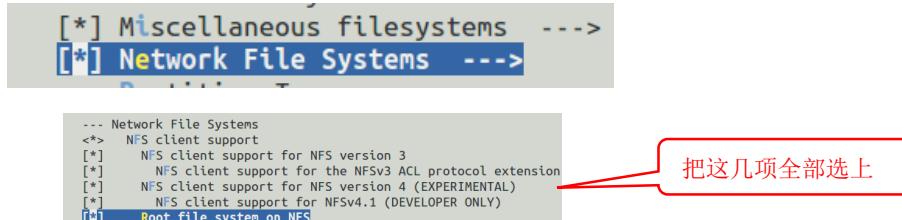
把 IP:kernel....选上

把 DHCP 和 BOOTP 选上

第一步内核网络配置完成



第二步配置文件系统支持



第二步文件系统支持 NFS 配置完成

直接 make 编译，将内核下载进开发板

重新启动板子

```
[ 7.713004] eth0: link up, 100Mbps, full-duplex, lpa 0x4DE1
[ 7.717222] ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
[ 8.218904] Looking up port of RPC 100005/1 on 192.168.1.103
```

发现板子启动后，卡在了 Looking 这里，这是因为你的 linuxrc 是手动随意创建的，不是用 busybox 生成的，所以你的 linuxrc 内容没对

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls
linuxrc
```

这就是我放在 PC 虚拟机里面 nfs 文件系统目录下的 linuxrc 是自己创建了。

```
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/xzzrootfs# ls
bin  linuxrc  sbin  usr
root@ubuntu:/home/xiang/S5PV210/S5PV210-linux/rootfs/xzzrootfs# cp -r * /home/xiang/S5PV210/nfsrootfs
```

下面我将 busybox 工具生成的正规的 linuxrc 和其它目录拷贝到 nfs 文件系统 nfsrootfs 目录

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls
bin  linuxrc  sbin  usr
```

拷贝成功。检查开发板和 PC 机网络连接正常，确认主机 nfs 文件系统启动正常，确认开发板 ip 地址网卡配置正确。

重启开发板

```
[ 8.218904] Looking up port of RPC 100005/1 on 192.168.1.103
[ 8.355985] VFS: Mounted root (nfs filesystem) on device 0:12.
[ 8.360402] Freeing init memory: 172K
can't run '/etc/init.d/rcS': No such file or directory
can't open /dev/tty2: No such file or directory
can't open /dev/tty3: No such file or directory
can't open /dev/tty4: No such file or directory
can't open /dev/tty2: No such file or directory
ls
bin  linuxrc  sbin  usr
can't open /dev/tty3: No such file or directory
can't open /dev/tty4: No such file or directory
can't open /dev/tty2: No such file or directory
```

这就是 VFS 挂载成功

虽然终端在不停的打印找不到/dev 目录，但是我可以执行 ls 命令了，证明 busybox 生成的文件系统挂载成功了

busybox 文件系统原理

先来解决文件系统挂起之后找不到启动文件的问题

```
can't open /dev/tty4: No such file or directory  
can't open /dev/tty2: No such file or directory  
ls  
bin  linuxrc  sbin  usr  
can't open /dev/tty3: No such file or directory  
can't open /dev/tty4: No such file or directory  
can't open /dev/tty2: No such file or directory
```

这种情况就是没有找到启动文件

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# mkdir etc  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls  
bin  etc  linuxrc  sbin  usr
```

去 nfs 文件系统下建立个 etc 目录

其实 nfs 文件系统里面的内容都是要最后做出 ext2,ext3 镜像的文件系统内容

: vim inittab 在 etc 目录下创建 inittab 文件

```
1 #first:run the system script file  
2 ::sysinit:/etc/init.d/rcS  
3 ::askfirst:-/bin/sh  
4 ::ctrlaltdel:-/sbin/reboot  
5 #umount all filesystem  
6 ::shutdown:/bin/umount -a -r  
7 #restart init process  
8 ::restart:/sbin/init
```

在 inittab 文件写上这些内容

重新启动开发板

```
[    7.175330] VFS: Mounted root (nfs filesystem) on device 0:12.  
[    7.179744] Freeing init memory: 172K  
can't run '/etc/init.d/rcS': No such file or directory  
  
Please press Enter to activate this console.  
/ #
```

文件系统挂起成功，找到了第一个执行程序 inittab，所以终端不会循环打印 can't open... 了

我们来分析下 inittab 文件

```
#first:run the system script file //打#号就是注释的意思  
::sysinit:/etc/init.d/rcS //sysinit 命令就是控制台在执行之前要执行什么程序，这里是执行 rcS 脚本  
::askfirst:-/bin/sh //askfirst 就是在执行/bin/sh 控制台程序之前，输出 please press.....字符串，然后你按回车，才会  
执行/bin/sh 程序，控制台程序就是支持你输入命令行的功能  
::ctrlaltdel:-/sbin/reboot //ctrlaltdel 命令是键盘按下 ctrl+alt+del 就执行/sbin/reboot 命令重启开发板要有界面系统才行  
#umount all filesystem  
::shutdown:/bin/umount -a -r //shutdown 点击关机，就会去执行/bin/umount 这里 umount 书写错误，就是取消挂载  
#restart init process  
::restart:/sbin/init //restart 命令，当你点击重启功能的时候执行/sbin/init
```

inittab 文件就是在板子文件系统挂载后，文件系统第一个程序 linuxrc 会去无限循环调用 inittab 文件，反复循环执行 inittab 里面的每个程序

下面我们来建立 rcS 文件

在/etc/目录下创建 init.d 目录，因为这个路径是在上一页 inittab 里面指定的

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# mkdir init.d  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# ls  
init.d  inittab
```

在 init.d 目录下创建 rcS 文件

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# vim rcS  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# ls  
rcS  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# chmod 755 rcS  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# ls  
rcS
```

记住一定要给 rcS 文件加权限，因为 rcS 是运行脚本，shell 脚本不一定要在文件名后面.sh。只需要对文件加了运行权限该文件就属于 shell 脚本，就算写了.sh 也要加运行权限
下面就是 rcS 文件的内容

```
1 #!/bin/sh  
2 PATH=/sbin:/bin:/usr/sbin:/usr/bin  
3  
4 runlevel=S  
5 prelevel=N  
6  
7 umask 022  
8  
9 export PATH runlevel prelevel  
10  
11 mount -a  
12  
13 echo /sbin/mdev > /proc/sys/kernel/hotplug  
14 mdev -s  
15  
16 /bin/hostname -F /etc/sysconfig/HOSTNAME  
17  
18 ifconfig eth0 192.168.1.12  
19
```

runlevel 有 6 中模式，我们这里的 S 是单用户模式，其余模式自行百度

这句就是自己定义个 PATH 环境变量，然后其它的应用程序，比如 ls 命令会自动去找 PATH 定义的路径里面有没 ls 程序。（像 PATH=\$PATH:/usr/sbin/arm-linux/ ...这种就是在原有 PATH 变量路径的基础上增加新的路径）

umask 022 这个 022 就是表示在文件系统下创建的文件默认权限是多少

将环境变量导出到全局，linux 其它应用程序就可以使用这些变量定义路径，或者内容了

mount -a busybox 会自动去/etc/目录下寻找 fstab 文件

让 mdev 自动挂载驱动设备节点在/dev 目录下

设置当前系统主机名

其实网卡 IP 可以 dhcp 获取，这里 IP 可以不设置死

因为我没有建立 fstab 文件，这个文件去网上下载个标准版的拷贝进来

启动开发板

```
mount: can't read '/etc/fstab': No such file or directory  
/etc/init.d/rcS: line 13: can't create /proc/sys/kernel/hotplug: nonexistent directory  
mdev: can't change directory to '/dev': No such file or directory  
hostname: can't open '/etc/sysconfig/HOSTNAME': No such file or directory  
  
Please press Enter to activate this console.  
/ # ls  
bin      etc      linuxrc  sbin      usr  
/ #
```

创建 fstab 文件，普通文件文件就行

<file system>	<dir>	<type>	<options>	<dump>	<pass>
proc	/proc	proc	defaults	0	0
sysfs	/sys	sysfs	defaults	0	0
tmpfs	/var	tmpfs	defaults	0	0
tmpfs	/tmp	tmpfs	defaults	0	0
tmpfs	/dev	tmpfs	defaults	0	0

重启开发板

```
mount: mounting proc on /proc failed: No such file or directory
mount: mounting sysfs on /sys failed: No such file or directory
mount: mounting tmpfs on /var failed: No such file or directory
mount: mounting tmpfs on /tmp failed: No such file or directory
mount: mounting tmpfs on /dev failed: No such file or directory
/etc/init.d/rcS: line 13: can't create /proc/sys/kernel/hotplug: nonexistent directory
mdev: can't change directory to '/dev': No such file or directory
hostname: can't open '/etc/sysconfig/HOSTNAME': No such file or directory

Please press Enter to activate this console.
```

fstab 文件找到了，但是无法挂载这些目录，这是因为你要自己在文件系统创建好这些目录，然后 mount -a 只负责自动挂载

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls
bin dev etc linuxrc proc sbin sys tmp usr var
```

目录创建完毕，重启开发板

```
[ 8.171775] Freeing init memory: 172K
hostname: can't open '/etc/sysconfig/HOSTNAME': No such file or directory
```

Please press Enter to activate this console.

```
/ # ls
bin etc proc sys usr
dev linuxrc sbin tmp var
/ #
```

主机名的文件还没有创建

但是文件系统的目录应该挂载成功了

如何确认文件系统设备节点和目录挂载成功了

```
/ # ls proc/
1          4          bus          kallsyms      self
10         41         cgroups      kmsq          softirqs
11         42         cmdline      kpagecount    stat
12         44         cpu          kpageflags   sys
13         45         cpufreq     loadavg      sysrq-trigger
14         46         crypto      locks        sysvipc
15         49         devices    meminfo     timer_list
```

查看开发板 proc 目录有内容

```
/ # ls /dev/
CEC          ptyr6          sequencer2      ttyq5
HDP          ptyr7          snd           ttyq6
adc          ptyr8          tty           ttyq7
alarm        ptyr9          tty0          ttyq8
apm_bios    ptyra          tty1          ttyq9
ashmem       ptyrb          tty10         ttyqa
```

查看/dev 目录有内容

证明挂载成功。

下面我们来解决 hostname: can't open '/etc/sysconfig/HOSTNAME': No such file or directory

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# mkdir sysconfig
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# ls
fstab init.d inittab sysconfig
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# cd sysconfig/
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/sysconfig# vim HOSTNAME
```

1 xzz_Linux

在 HOSTNAME 文件写上你自己喜欢的名字

```
[ 8.179281] Freeing init memory: 172K
Please press Enter to activate this console.
/ # ls
bin etc proc sys usr
dev linuxrc sbin tmp var
/ # echo $HOSTNAME
xzz_Linux
/ # hostname
xzz_Linux
/ #
```

系统启动完成，没有打印错误，主机名也变成自己的了，最小文件系统移植完成。

文件系统用户登录密码设置

以上几页操作方式都是开发板启动后直接就进入了 root 用户下敲打命令行。

现在要做一个开放板启动后必须输入用户名和密码之后，才能进入 root 用户下敲打命令行。

```
1 #first:run the system script file
2 ::sysinit:/etc/init.d/rcS
3 ::askfirst:-/bin/sh
4 ::ctrlaltdel:-/sbin/reboot
5 #umount all filesystem
6 ::shutdown:/bin/umount -a -r
7 #restart init process
8 ::restart:/sbin/init
```

程序开机执行 inittab 时，直接执行了~/bin/sh 程序，这个 sh 程序就是直接进入 root 用户，而不需要用户密码设置

修改 inittab 文件

```
#first:run the system script file
::sysinit:/etc/init.d/rcS
#::askfirst:-/bin/sh
::sysinit:/bin/login
::ctrlaltdel:-/sbin/reboot
#umount all filesystem
::shutdown:/bin/umount -a -r
#restart init process
::restart:/sbin/init
```

1. 屏蔽掉直接进入 root 的 sh 程序
2. 用 sysinit 程序执行/bin 目录下的 login 程序

```
[    8.299591] Freeing init memory: 172K
xzz_Linux login: root
Password:
```

^CLogin incorrect

xzz_Linux login:

开发板重启要求你输入用户名和密码

但是我没有设置用户密码啊!!!

设置用户名和密码的文件有两个，一个是 passwd 文件，一个是 shadow 文件

这两个文件都存放在/etc 目录下

下面我们看看 ubuntu 系统的 passwd 文件

```
1 root:x:0:0:root:/root:/bin/bash
2 daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
3 bin:x:2:2:bin:/bin:/usr/sbin/nologin
4 sys:x:3:3:sys:/dev:/usr/sbin/nologin
5 sync:x:4:65534:sync:/bin:/bin/sync
6 games:x:5:60:games:/usr/games:/usr/sbin/nologin
7 man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
8 lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
9 mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
10 news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

这行代表 root 用户的密码设置

这行代表 daemon 用户密码设置

下面依次类推，所以不同用户登录密码设置不同

这里 x 表示这行用户密码是加密了的

绿色字符表示登录之后默认的命令行目录在哪里

橙色表示用户登录后自动去执行哪一种脚本程序

下面我们来分析 ubuntu 的 shadow 文件

```
root@ubuntu: /etc
1 root:$6$jLCWtk20$ArOuk..hODRSqeJwrMt1.D1HckxHCixYgDd3t
2 daemon:*:16652:0:99999:7:::
3 bin:*:16652:0:99999:7:::
4 sys:*:16652:0:99999:7:::
5 sync.*:16652:0:99999:7:::
```

这就是 root 用户加密后的登录密码

我们嵌入式系统的 passwd 和 shadow 文件直接复制 ubuntu 的来用

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc# ls
fstab init.d inittab passwd shadow sysconfig
```

已经把 ubuntu 的 passwd 和 shadow 文件复制到开发板的 busybox 文件系统了

修改 passwd 文件

```
1 root:x:0:0:root:/:/bin/sh
```

设置用户登录之后默认在根目录/ 敲命令行

Busybox 文件系统只支持 sh 的 shell 脚本，所以把 bash 改成 sh

删除 passwd 文件的其它行，只保留 root 行，记住删除的时候一定要确认该 passwd 文件是我们开发板文件系统的文件，而不是 ubuntu 的系统的 passwd 文件

修改 shadow 文件

```
1 root:$6$jLCWtk20$ArOuk..hODRSqeJwrMt1.D1HckxHCixYgDd3t00afzz0oP7r3mx6j0s1hpZYG.enwtJ1JAFAzDupKvsCgdKwqJ:16877:0:99999:7:::
```

删除 shadow 文件的其它行，只保留 root 行，记住删除的时候一定要确认该 shadow 文件是我们开发板文件系统的文件，而不是 ubuntu 的系统的 shadow 文件

```
1 root:$6$jLCWtk20$ArOuk..hODRSqeJwrMt1.D1HckxHCixYgDd3t00afzz0oP7r3mx6j0s1hpZYG.enwtJ1JAFAzDupKvsCgdKwqJ:16877:0:99999:7:::
```

修改这段黄色框的加密密码，我 ubuntu 密码是 675....，所以你下载到开发板密码也是 675...

如果你不想用 ubuntu 设置好的 shadow 加密密码，你可以用 passwd 命令来重新设置密码。

```
xzz_Linux login: root
Password:
login[55]: root login on 'console'
-sh: can't access tty; job control turned off
~ # ls
bin      etc      proc      sys      usr
dev      linuxrc   sbin      tmp      var
```

这就是用 ubuntu 的 shadow 密码 675...，移植到开发板成功登录的现象

login[55]: root login on 'console'

-sh: can't access tty; job control turned off 问题解决

这个问题是我在 inittab 文件的::sysinit:/bin/login 中，登录系统里面没有指定我的 root 用户使用的是哪一个串口设备节点。我们终端使用的是串口 2。

所以 ls /dev 发现 `s3c2410_serial2` 串口后是这个名字。

```
1 #first:run the system script file
2 ::sysinit:/etc/init.d/rcS
3 #:::askfirst:-/bin/sh
4 s3c2410_serial2::sysinit:/bin/login
5 ::ctrlaltdel:-/sbin/reboot
6 #umount all filesystem
```

将 s3c2410_serial2 写进 inittab 文件对应的用户登录脚本位置即可

```
xzz_Linux login: root
Password:
login[55]: root login on 'console'
-sh: can't access tty; job control turned off
~ # ls
bin      etc      proc      sys      usr
dev      linuxrc   sbin      tmp      var
```

板子终端 root 登录正常

文件系统实现链接动态链接库

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls  
bin dev etc lib linuxrc proc sbin sys tmp usr var
```

在文件系统根目录下建立个 lib 目录来存放动态链接库文件。

有些基础动态库文件官方的交叉编译器帮你做好了的，你去开发板使用的交叉编译工具链目录里面去找，拷贝交叉编译器 lib 里面的库文件到文件系统 lib 目录下就行了。不需要自己再去编译制作基础的库文件。

```
root@ubuntu:/opt/arm-2009q3/arm-none-linux-gnueabi/libc/lib# ls  
ld-2.10.1.so          libcidn-2.10.1.so    libgcc_s.so           libnss_compat-2.10.1.so  libnss_hesiod.so.2  
ld-linux.so.3          libcidn.so.1       libgcc_s.so.1        libnss_compat.so.2      libnss_nis-2.10.1.  
libanl-2.10.1.so      libcrypt-2.10.1.so   libm-2.10.1.so       libnss_dns-2.10.1.so   libnss_nisplus-2.10.1.  
libanl.so.1           libcrypt.so.1     libmemusage.so     libnss_files-2.10.1.so  libnss_nis.so.2  
libBrokenLocale-2.10.1.so libc.so.6        libm.so.6          libnss_files.so.2      libpcprofile.so  
libBrokenLocale.so.1  libdl-2.10.1.so    libnsl-2.10.1.so    libnss_hesiod-2.10.1.so libpthread-2.10.1.  
libc-2.10.1.so         libdl.so.2       libnsl.so.1        libnss_hesiod-2.10.1.so libpthread-2.10.1.
```

一定是交叉编译工具链里面的 libc 目录下的 lib 目录，将这些库文件拷贝进开发板文件系统的 lib。千万不要把 ubuntu 里面的 lib 文件拷到开发板了，因为那时 x86 编译器编译的库。不是 arm-linux 编译编译的库。

```
libBrokenLocale.so.1      libdl-2.10.1.so    libnsl-2.10.1.so  libnss_files.so.2      libpcprofile.so  
libc-2.10.1.so           libdl.so.2       libnsl.so.1       libnss_hesiod-2.10.1.so  libpthread-2.10.1.so  
root@ubuntu:/opt/arm-2009q3/arm-none-linux-gnueabi/libc/lib# cp * -rfd /home/xiang/S5PV210/nfsrootfs/lib
```

复制交叉编译器的 lib 库到文件系统 lib，记住一定要加-rfd，因为不加-rfd 的话，lib 里面有符号链接的文件复制完之后符号链接的文件会变成真实的文件，导致 lib 目录容量变大，所以加-rfd 是为了保证 lib 目录的内容复制之后还是保持原来的符号链接

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/lib# ls  
ld-2.10.1.so          libcidn-2.10.1.so    libgcc_s.so           libnss_compat-2.10.1.  
ld-linux.so.3          libcidn.so.1       libgcc_s.so.1        libnss_compat.so.2      libnss_nis-2.10.1.  
libanl-2.10.1.so      libcrypt-2.10.1.so   libm-2.10.1.so       libnss_dns-2.10.1.so   libnss_nisplus-2.10.1.  
libanl.so.1           libcrypt.so.1     libmemusage.so     libnss_files-2.10.1.so  libnss_nis.so.2  
libBrokenLocale-2.10.1.so libc.so.6        libm.so.6          libnss_files.so.2      libpcprofile.so  
libBrokenLocale.so.1  libdl-2.10.1.so    libnsl-2.10.1.so    libnss_hesiod-2.10.1.so libpthread-2.10.1.  
libc-2.10.1.so         libdl.so.2       libnsl.so.1        libnss_hesiod-2.10.1.so libpthread-2.10.1.  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/lib# ls -l  
total 3872  
-rwxr-xr-x 1 root root 155919 Apr  1 23:35 ld-2.10.1.so  
lrwxrwxrwx 1 root root      12 Apr  1 23:35 ld-linux.so.3 -> ld-2.10.1.so  
-rwxr-xr-x 1 root root 26523 Apr  1 23:35 libanl-2.10.1.so  
lrwxrwxrwx 1 root root      16 Apr  1 23:35 libanl.so.1 -> libanl-2.10.1.so  
-rwxr-xr-x 1 root root 19152 Apr  1 23:35 libBrokenLocale-2.10.1.so
```

复制后文件系统 lib 库有动态库内容了

而且符
合链接
海存在

```
~ # ls  
bin      etc      linuxrc    sbin      tmp      var  
dev      lib      proc      sys      usr  
~ # cd lib/  
/lib # ls  
ld-2.10.1.so          libgcc_s.so           libnss_nis-2.10.1.so  
ld-linux.so.3          libgcc_s.so.1        libnss_nis.so.2  
libBrokenLocale-2.10.1.so libm-2.10.1.so     libnss_nisplus-2.10.1.so  
libBrokenLocale.so.1  libm.so.6          libnss_nisplus.so.2  
libSegFault.so         libmemusage.so     libpcprofile.so  
libanl-2.10.1.so      libnsl-2.10.1.so    libpthread-2.10.1.so  
libanl.so.1            libnsl.so.1        libpthread.so.0  
libc-2.10.1.so         libnss_compat-2.10.1.so libresolv-2.10.1.so
```

开发板上 lib 有这些动态库了，至于程序能不能链接上动态库呢？用 helloworld 程序测试下

```
#include <stdio.h>  
int main(void)  
{  
    printf("xxxxxxxxxxxxzzzzzzzz\n");  
    return 0;  
}
```

这个 printf 函数是在 libc.so 库文
件 C 程序基础库中实现的

```
root@ubuntu:/home/xiang/S5PV210/noOS_driver# arm-none-linux-gnueabi-gcc-4.4.1 -o helloworld helloworld.c
```

直接-o 编译，这样就是动态链接 printf 函数的库文件

```

~ # ls
bin          etc        lib        proc
dev          helloworld linuxrc   sbin
~ # ./helloworld
xxxxxxxxxxxxzzzzzzzzz
~ #

```

执行成功，证明动态库移植正确。

为了系统完善我们可以把交叉编译器 libc 中的 usr/lib/里面的库也移植到开发板

```

root@ubuntu:/home/xiang/S5PV210/nfsrootfs/usr# mkdir lib
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/usr# ls
bin  lib  sbin

```

文件系统 usr 目录下创建 lib 目录

```
root@ubuntu:/opt/arm-2009q3/arm-none-linux-gnueabi/libc/usr/lib# cp -rfd -r /home/xiang/S5PV210/nfsrootfs/usr/lib
```

复制到开发板

```

root@ubuntu:/home/xiang/S5PV210/nfsrootfs/usr/lib# ls
bin  libanl.a      libcidn.so    libdl.so   libnsl.a      libnss_nisplus.
crti.o libanl.so    libc_nonshared.a libg.a     libnsl.so    libnss_nis.so
crti.o libBrokenLocale.a libcrypt.a    libieee.a  libnss_compat.so libpthread.a
crti.o libBrokenLocale.so libcrypt.so   libm.a     libnss_dns.so libpthread_nons
gconv  libbsd-compat.a libc.so       libncheck.a libnss_files.so libpthread.so
gcrti.o libc.a       libdl.a      libn.so     libnss_hesiod.so libresolv.a
gcrti.o libc.a       libdl.a      libn.so     libnss_nisplus.so libresolv.a

```

开发板 usr/lib 移植成功

```

~ # ls /usr/lib/
Mcrt1.o          libcrypt.a           libpthread.a
Scrt1.o          libcrypt.so          libpthread.so
bin              libdl.a              libpthread_nonshared.a
crt1.o          libdl.so             libresolv.a
crti.o          libg.a               libresolv.so
crti.o          libieee.a            librrocsvc.a

```

这样你的 C 程序在运行时加载动态库的时候，会先去 lib 目录下面找有没有动态库，如果 lib 下面没有就回去 usr/lib 下面找有没有动态库，如果 usr/lib 下面没有就会程序运行报错。

这个文件系统我没有增加动态库的环境变量，如果你不想自己做的库文件在 lib 目录和 usr/lib 目录下，想把库文件放在自己定义的目录下，那么就要在开机脚本里面增加动态库环境变量。

7 export LD_LIBRARY_PATH=/lib:/usr/lib

这就是增加的动态库

环境变量放，你还可以打冒号后面接着加动态库路径。

增加开机自动启动功能

主要修改 rcS 文件实现开机自动启动自己做的程序

```

~ # ls
bin          etc        lib
dev          helloworld linuxrc   我们开机直接运行根/目录下的 helloworld 程序

```

```

root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# ls
rcS
root@ubuntu:/home/xiang/S5PV210/nfsrootfs/etc/init.d# vim rcS

```

直接进入文件系统/etc/init.d 目录修改 rcS

```

1 #!/bin/sh
2 PATH=/sbin:/bin:/usr/sbin:/usr/bin
3
4 runlevel=$
5 prelevel=$N
6
7 umask 022
8
9 export PATH runlevel prelevel
10
11 mount -a
12
13 echo /sbin/mdev > /proc/sys/kernel/hotplug
14 mdev -s
15
16 /bin/hostname -F /etc/sysconfig/HOSTNAME
17
18 ifconfig eth0 192.168.1.12
19
20 ./helloworld

```

因为 helloworld 在根目录下，所以不用 cd 什么的，也不用 ../../ 什么的，直接执行 ./helloworld 就行了

程序开机自动执行了

```

[ 0.115000] booting init memory. 1/4M
xxxxxxxxxxxxzzzzzzzzz
xzz_Linux login: root
Password:

```

将 nfs 移植好的文件系统 rootfs 制作成镜像烧录进开发板

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# mkdir mnt  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls  
bin dev etc helloworld lib linuxrc mnt proc sbin sys tmp usr var
```

将文件系统挂载 sd 卡的 mnt 目录创建好，整个文件系统就做好了。

按照前面有一个章节<<根文件系统制作成镜像>>的方法，把现在完整的 busybox 文件系统制作成 ext2 镜像，烧录进开发板

```
128M nfsrootfs  
root@ubuntu:/home/xiang/S5PV210# du -h nfsrootfs 执行发现该文件系统有 128M 大小
```

第 1 步: dd if=/dev/zero of=rootfs.ext2 bs=1024 count=2048

这里要修改 count 大小，根据我现在制作出来的 nfsrootfs 大小来确定。

bs=1024 是指的每一块是 1024 字节，count=2048 意思是该文件系统有 2048 个 bs 容量也就是 $1024 \times 2048 = 2M$ 。但是我这里 nfsrootfs 是 128M，那么我就要修改 count=204800 大小。

也就是 $1024 \times 204800 = 209715200(200M)$ ，这 200M 就是用来放文件系统空间。

修改为 dd if=/dev/zero of=rootfs.ext2 bs=1024 count=204800

第 2 步: losetup /dev/loop1 rootfs.ext2 //先做好镜像

第 3 步: mke2fs -m 0 /dev/loop1 204800 //这里要和 count 一样，就是设置镜像大小

第 4 步: 创建一个 rootfs 空目录来挂载

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# ls  
rootfs rootfs.ext2  
root@ubuntu:/home/xiang/S5PV210/rootfsext3#
```

到时候 nfsrootfs 文件系统内容拷贝进 rootfs，就相当于 rootfs 内容向 rootfs.ext2 镜像里面写

第 5 步: mount -t ext2 /dev/loop1 ./rootfs //挂载 rootfs，这个 rootfs 是创建的空目录

第 6 步:

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# cp * -rfd .. /rootfsext3/rootfs  
root@ubuntu:/home/xiang/S5PV210/nfsrootfs#
```

然后将 nfsrootfs 里面的文件系统拷贝给 rootfs

第 7 步: umount /dev/loop1

第 8 步: losetup -d /dev/loop1

卸载后就发现 rootfs.ext2 镜像就有了文件系统内容了。

重复看一下执行步骤

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# dd if=/dev/zero of=rootfs.ext2 bs=1024 count=204800  
204800+0 records in  
204800+0 records out  
209715200 bytes (210 MB) copied, 0.864895 s, 242 MB/s
```

创建了 210M 的空间

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# losetup /dev/loop1 rootfs.ext2
```

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mke2fs -m 0 /dev/loop1 204800  
mke2fs 1.42.9 (4-Feb-2014)  
Discarding device blocks: done  
Filesystem label=  
OS type: Linux  
Block size=1024 (log=0)  
Fragment size=1024 (log=0)  
Stride=0 blocks, Stripe width=0 blocks  
51200 inodes, 204800 blocks  
0 blocks (0.00%) reserved for the super user  
First data block=1  
Maximum filesystem blocks=67371068  
25 block groups  
8192 blocks per group, 8192 fragments per group  
2048 inodes per group  
Superblock backups stored on blocks:  
    8193, 24577, 40961, 57345, 73729  
Allocating group tables: done  
Writing inode tables: done  
Writing superblocks and filesystem accounting information: done
```

出现 write....done
就是正确

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3# mount -t ext2 /dev/loop1 ./rootfs
root@ubuntu:/home/xiang/S5PV210/rootfsext3# cd rootfs
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# ls
lost+found
```

rootfs 目录没挂载到
rootfs.ext2 镜像了

```
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# ls
bin dev etc helloworld lib linuxrc mnt proc sbin sys tmp usr var
root@ubuntu:/home/xiang/S5PV210/nfsrootfs# cp * -rfd /home/xiang/S5PV210/rootfsext3/rootfs
root@ubuntu:/home/xiang/S5PV210/nfsrootfs#
```

拷贝做好的文件系统
给镜像

```
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# ls
bin dev etc helloworld lib linuxrc lost+found mnt proc sbin
root@ubuntu:/home/xiang/S5PV210/rootfsext3/rootfs# cd ..
root@ubuntu:/home/xiang/S5PV210/rootfsext3# ls
rootfs rootfs.ext2
root@ubuntu:/home/xiang/S5PV210/rootfsext3# umount /dev/loop1
root@ubuntu:/home/xiang/S5PV210/rootfsext3# losetup -d /dev/loop1
```

执行取消挂载，
rootfs 的内容就会自动
给 rootfs.ext2 同时
rootfs 目录会被清除

将用 fastboot 将 rootfs.ext2 烧写进开发板

```
E:\Desktop\fastboot>fastboot flash system rootfs.ext2
sending 'system' (204800 KB)...
```

开发板文件
系统在 flash
的 system 区

```
x210 # fastboot
[Partition table on MoviNAND]
ptn 0 name='bootloader' start=0x0 len=N/A (use hard-coded info. (cmd: movi))
ptn 1 name='kernel' start=N/A len=N/A (use hard-coded info. (cmd: movi))
ptn 2 name='ramdisk' start=N/A len=0x300000 (-3072KB) (use hard-coded info. (cmd: movi))
ptn 3 name='config' start=0xAECC00 len=0x1028DC00 (-264759KB)
ptn 4 name='system' start=0x10D7A800 len=0x1028DC00 (-264759KB)
ptn 5 name='cache' start=0x21008400 len=0x65F7000 (-104412KB)
ptn 6 name='userdata' start=0x275FF400 len=0xC0C6FC00 (-3158463KB)
Received 17 bytes: download:0c800000
Starting download of 209715200 bytes
.......
```

文件系统有
128M 下载
比较慢

```
.....
downloading of 209715200 bytes finished
Received 12 bytes: flash:system
flashing 'system'

MMC write: dev # 0, block # 551892, count 529518 ...
```

```
E:\Desktop\fastboot>fastboot flash system rootfs.ext2
sending 'system' (204800 KB)... OKAY
writing 'system'... OKAY
```

这种就表示
下载成功

程序下载完成后重启开发板

进入 uboot 模式

```
x210 # set bootargs console=ttySAC2,115200 root=/dev/mmcblk0p2 rw init=/linuxrc rootfstype=ext2
```

开发板 uboot 的 bootargs 要从 nfs 改成 ext2 启动，这样才会去 INAND 启动文件系统

然后 save 保存

然后再次重启开发板

```
[ 5.573020] Freeing init memory: 172K
xxxxxxxxxxxxxxxxxxxx
xzz_Linux login: root
Password:
login[56]: root login on 's3c2410_serial2'
~ # ls
bin      etc      lib      lost+found  proc      sys      usr
dev      helloworld  linuxrc   mnt       sbin      tmp      var
```

你看文件系统烧录进 INAND 存储器了。以上下载方式可以做出 shell 脚本方便执行。

TFTP 下载大容量内核和文件系统



因为 `uboot` 担任了开发板 `tftp` 客户端的角色，所以主机就只能给开发板传送，`kernel` 内核镜像和文件系统镜像进行下载。

准备工作：

1. 确保开发板网线连接正常
2. 确保开发板 `uboot` 模式下 `ping` 虚拟机是 `host....is alive`，证明开发板能 `ping` 通虚拟机

虚拟机先安装 tftp 程序

```
sudo apt-get install tftp-hpa tftpd-hpa
```

```
root@ubuntu:/home/xiang# sudo apt-get install tftp-hpa tftpd-hpa
Reading package lists... Done
Building dependency tree... 63%
```

```
sudo apt-get install xinetd
```

```
root@ubuntu:/home/xiang# sudo apt-get install xinetd
Reading package lists... Done
Building dependency tree
Reading state information... Done
```

vim /etc/xinetd.conf 查看 `xinetd.conf` 文件

```
5 defaults
6 {
7
8 # Please note that you need a log_type line to be able to use log_on_success
9 # and log_on_failure. The default is the following :
10 # log_type = SYSLOG daemon info
11
12 }
13
14 includedir /etc/xinetd.d
```

确保 `xinetd.conf` 文件的内容是上面这样。一般都是上面这样。

g# vim /etc/default/tftpd-hpa 修改 `tftpd-hpa` 里面的内容

```
2
3 TFTP_USERNAME="tftp" 这个不用改
4 TFTP_DIRECTORY="/var/lib/tftpboot" 这个路径要改成我们以后放内核镜像的 tftp 目录位置,然后开发板去这个目录位置下载内核镜像
5 TFTP_ADDRESS=":69" 这里要改成 0.0.0.0: 69
6 TFTP_OPTIONS="--secure" 这里要改成 -l -c -s
```

我们随便找个位置创建 tftpboot 目录

```
root@ubuntu:/# ls  
bin  cdrom  etc  initrd.img  lib32  libx32    media  opt  root  sbin  sys      tmp  var  
boot dev   home  lib       lib64  lost+found  mnt   proc  run   srv   tftpboot  usr  vmlinuz  
root@ubuntu:/#
```

为了方便我在根目录下创建了 tftpboot 目录

```
TFTP_USERNAME="tftp"  
TFTP_DIRECTORY="/tftpboot"  
TFTP_ADDRESS="0.0.0.0:69"  
TFTP_OPTIONS="-l -c -s"
```

tftpd-hpa 文件修改完成，保存退出

```
root@ubuntu:/# chmod 777 tftpboot
```

给 tftpboot 目录最高权限

```
opt  root  sbin  sys      tmp  var  
proc run   srv   tftpboot  usr  vmlinuz
```

去/etc/xinetd.d/目录下创建 tftp 文件

```
:/# vim /etc/xinetd.d/tftp
```

```
service tftp  
{  
    socket_type = dgram  
    wait = yes  
    disable=no  
    user=root  
    protocol=udp  
    server=/usr/sbin/in.tftpd  
    server_args=-s /tftpboot  
    #log_on_success+=PID HOST DURATION  
    #log_on_failure+=HOST  
    per_source=11  
    cps=100 2  
    flags=IPv4  
}
```

这个路径必须和
你创建的 tftpboot
文件路径一致

```
service tftp
{
    socket_type = dgram
    wait = yes
    disable=no
    user=root
    protocol=udp
    server=/usr/sbin/in.tftpd
    server_args=-s /tftpboot
    #log_on_success+=PID HOST DURATION
    #log_on_failure+=HOST
    per_source=11
    cps=100 2
    flags=IPv4
}
```

```
root@ubuntu:/# sudo service tftpd-hpa restart
tftpd-hpa stop/waiting
tftpd-hpa start/running, process 4651
root@ubuntu:/#
```

重启 tftp 服务

```
root@ubuntu:/home/xiang# sudo service tftpd-hpa status
tftpd-hpa start/running
root@ubuntu:/home/xiang#
```

查看虚拟机 tftp 服务器是否开启，以后打开虚拟机都可以查看下

```
root@ubuntu:/# sudo /etc/init.d/xinetd reload
 * Reloading internet superserver configuration xinetd
root@ubuntu:/# sudo /etc/init.d/xinetd restart
 * Stopping internet superserver xinetd
 * Starting internet superserver xinetd
root@ubuntu:/#
```

我们用虚拟机先本地测试下看 tftp 服务行不行

```
root@ubuntu:/home/xiang# cd /tftpboot/
root@ubuntu:/tftpboot# ls
root@ubuntu:/tftpboot# vim a.c
root@ubuntu:/tftpboot# vim xzz.c
root@ubuntu:/tftpboot# ls
xzz.c
root@ubuntu:/tftpboot#
```

先去 tftpboot 目录下创建个 xzz.c 文件

```
root@ubuntu:/# tftp localhost
tftp> get a.c
Error code 1: File not found
tftp> get xzz.c
tftp> q
root@ubuntu:/#
```

在虚拟机另外一个终端 get a.c 发现报错，这是因为你 tftpboot 目录里面根本就没有 a.c

get xzz.c 就没有问题了，因为 tftpboot 目录有 xzz.c 文件，然后按 q 退出

证明虚拟机 tftp 搭建完成

然后我们在开发板上面来使用 tftp 下载虚拟机的内容

```
gatewayip=192.168.14.1  
netmask=255.255.0.0  
ipaddr=192.168.14.10  
serverip=192.168.14.12
```

首先在 uboot 下用 print 确定 ipaddr 是和虚拟机在一个网段，然后 serverip=虚拟机的 ip 地址

```
Environment size: 373/16380 bytes
```

```
root@ubuntu:/mnt/hgfs/win+linux# ifconfig  
eth0      Link encap:Ethernet  HWaddr 00:0c:29:cf:c4:2e  
          inet addr:192.168.14.12  Bcast:192.168.14.255  Mask:255.255.255.0  
          inet6 addr: fe80::20c:29ff:fecc:c42e/64 Scope:Link  
             UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
```

```
root@ubuntu:/tftpboot# ls  
XZZ.C
```

然后下载虚拟机 tftpboot 目录里面的 xzz.c 文件到开发板上

执行 Uboot 命令 `tftp 0x30008000 xzz.c` //这句话的意思就是把 xzz.c 文件下载到 `0x30008000` 内存地址上

```
x210 # tftp 0x30008000 xzz.c  
dm9000 i/o: 0x88000300, id: 0x90000a46  
DM9000: running in 16 bit mode  
MAC: 00:40:5c:26:0a:5b  
operating at 100M full duplex mode  
TFTP from server 192.168.14.12; our IP address is 192.168.14.10  
Filename 'xzz.c'.  
Load address: 0x30008000  
Loading: #  
done  
Bytes transferred = 5 (0x5)  
v210 *
```

这里执行

显示 done 证明下载成功

```
x210 # tftp 0x30008000 ad.c  
dm9000 i/o: 0x88000300, id: 0x90000a46  
DM9000: running in 16 bit mode  
MAC: 00:40:5c:26:0a:5b  
operating at 100M full duplex mode  
TFTP from server 192.168.14.12; our IP address is 192.168.14.10  
Filename 'ad.c'.  
Load address: 0x30008000  
Loading: *  
TFTP error: 'File not found' (1)  
Starting again
```

如果你开发板下载的文件虚拟机 tftpboot 目录里面没有

这里就会显示找不到文件

```

x210 # tftp 0x30008000 xzz.c
dm9000 i/o: 0x88000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:40:5c:26:0a:5b
could not establish link
TFTP from server 192.168.14.12; our IP address is 192.168.14.10
Filename 'xzz.c'.
Load address: 0x30008000
Loading: T T T

```

如果开发板 tftp 下载的时候
网线断了，就没有 done 输出，这里还会显示没有连接

上面使用的是 uboot 下的 tftp 软件下载内核或者文件系统。但是我们在调试驱动的时候可以直接从虚拟机(服务器)上下载 hello.c 这种文件到开发板的文件系统上调试。所以我们下面要用一个和 uboot 不一样的 tftp，就 linux 的 tftp。

首先确保 uboot 的 tftp 和虚拟机的通讯正常，确保 busybox 文件系统里面安装了 tftp 软件，或者 QT 系统里面安装了 tftp 软件。

然后在开发板上直接使用 tftp 下载虚拟机/tftpboot 目录下的文件

```
[root@x210v3 etc]# tftp -g -r xzz.c 192.168.14.12
tftp: timeout
```

-g: 获取要下载的文件名
-r: 从服务器下载到开发板

要下载虚拟机(服务器) /tftpboot
目录下文件的名字

虚拟机(服务器)IP，这样开发板才知道去那台服务器下载 xzz.c 文件

发现出现 time out。

检查发现，虚拟机和开发板相互可以 ping 通，证明网络没有问题，问题就出在虚拟机 tftp 进程上。

```
root@ubuntu:/home/xiang# tftp localhost
tftp> get xzz.c
Transfer timed out.

tftp> q
```

测试主机自己的 tftp 发现无法通信，看来是主机自己的 tftp 服务程序出现问题了。

```
tftp>
root@ubuntu:/home/xiang# sudo service tftpd-hpa restart
tftpd-hpa stop/waiting
tftpd-hpa start/running, process 3409
```

重新启动 tftp 服务

```
root@ubuntu:/home/xiang# sudo /etc/init.d/xinetd reload
 * Reloading internet superserver configuration xinetd
root@ubuntu:/home/xiang# sudo /etc/init.d/xinetd restart
 * Stopping internet superserver xinetd
 * Starting internet superserver xinetd
root@ubuntu:/home/xiang# tftp localhost
tftp> get xzz.c
tftp> q
```

重新启动 xinetd，然主机自己测试 tftp 传输成功。

```
[root@x210v3 /]# tftp -g -r xzz.c 192.168.14.12
[root@x210v3 /]# ls
Settings/  etc/      linuxrc@  mnt/      root/      sys/      var/
bin/        home/     lost+found/ opt/      run@      tmp/      xzz.c
dev/        lib/      media/     proc/      sbin/
[root@x210v3 /]#
开发板再次测试 tftp 下载没有问题
```

我们下载个可以运行的文件试试

```
root@ubuntu:/tftpboot# arm-linux-gcc -o hello hello.c
root@ubuntu:/tftpboot# ls
hello  hello.c
root@ubuntu:/tftpboot# [REDACTED]
[REDACTED]
[root@x210v3 /]# tftp -g -r hello 192.168.14.12
[root@x210v3 /]# ls
Settings/  etc/      lib/      media/      proc/      sbin/      usr/
bin/        hello     linuxrc@  mnt/      root/      sys/      var/
dev/        home/     lost+found/ opt/      run@      tmp/
[root@x210v3 /]# chmod 777 hello
[root@x210v3 /]# ./hello
hello C .....
[root@x210v3 /]#
```

下载下来后记得修改执行文件 hello 的权限，这样就可以正常运行了。

Linux 系统使用 SCSI 命令访问 SCSI 总线上硬盘、CD-ROM 信息的方法

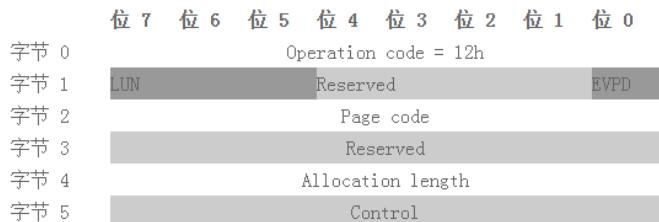
Linux 系统扫描 scsi 磁盘有几种方法？

第 1 种，应用程序通过定义的“SG_IO ioctl”可以像 scsi 磁盘等设备发送自定义的 scsi 指令集

表 1. 最常用的 SCSI 命令

命令	描述
Inquiry	请求目标设备的摘要信息
Test/Unit/Ready	检测目标设备是否准备好进行传输
READ	从 SCSI 目标设备传输数据
WRITE	向 SCSI 目标设备传输数据
Request Sense	请求最后一个命令的检测数据
Read Capacity	请求存储容量信息

表 2. inquiry 命令格式定义



如果 EVPD 参数位（用于启用关键产品数据）为 0 并且 Page Code 参数字节为 0，那么目标将返回标准 inquiry 数据。如果 EVPD 参数为 1，那么目标将返回对应 page code 字段的特定于供应商的数据。

```
typedef struct sg_io_hdr_t
{
    int interface_id; //一般应该设置为 's'
    int dxfer_direction; //用于确定数据传输的方向;
    unsigned char cmd_len; //指向 SCSI 命令的 cmdp 的字节长度
    unsigned char mx_sb_len; //当 sense_buffer 为输出时，可以写回到 sbp 的最大大小
    unsigned short iovec_count;
    unsigned int dxfer_len; //数据传输的用户内存的长度
    void * dxferp; //指向数据传输时长度至少为 dxfer len 字节的用户内存的指针。
    unsigned char * cmdp; //指向将要执行的 SCSI 命令的指针
    unsigned char * sbp; //缓冲检测指针
    unsigned int timeout; //用于使特定命令超时
    unsigned int flags;
    int pack_id;
    void * usr_ptr;
    unsigned char status; //由 SCSI 标准定义的 SCSI 状态字节。
    unsigned char masked_status;
    unsigned char msg_status;
    unsigned char sb_len_wr;
    unsigned short host_status;
    unsigned short driver_status;
    int resid;
    unsigned int duration;
    unsigned int info;
}
```

这个结构体清单就在头文件#include<scsi/sg.h>里面

表 1. 最常用的 SCSI 命令

命令	描述
Inquiry	请求目标设备的摘要信息
Test/Unit/Ready	检测目标设备是否准备好进行传输
READ	从 SCSI 目标设备传输数据
WRITE	向 SCSI 目标设备传输数据
Request Sense	请求最后一个命令的检测数据
Read Capacity	请求存储容量信息

表 2. inquiry 命令格式定义



如果 EVPD 参数位（用于启用关键产品数据）为 0 并且 Page Code 参数字节为 0，那么目标将返回标准 inquiry 数据。如果 EVPD 参数为 1，那么目标将返回对应 page code 字段的特定于供应商的数据。

```
#include<stdio.h>
#include<malloc.h>
#include<unistd.h>
#include<string.h>
#include<fcntl.h>
#include<sys/ioctl.h>
#include<scsi/sg.h>/SCSI的结构体和宏定义都在这个头文件里面
unsigned char sense_buffer[254];
unsigned char data_buffer[254];

struct sg_io_hdr* init_hdr(int page_code, int evpd)
{
    unsigned char cdb[6];
    struct sg_io_hdr * p_scsi_hdr = (struct sg_io_hdr *)malloc(sizeof(struct sg_io_hdr));//给scsi结构体分配一个缓存，因为写入磁盘的命令和读取磁盘结果的数据都在这个结构体里面
    memset(p_scsi_hdr, 0, sizeof(struct sg_io_hdr));//给结构体初始化为0
    if (p_scsi_hdr)
    {
        p_scsi_hdr->interface_id = 'S';
        p_scsi_hdr->flags = SG_FLAG_LUN_INHIBIT;
    }
    p_scsi_hdr->dxfer = data_buffer;
    p_scsi_hdr->dxfer_len = sizeof(data_buffer);
    p_scsi_hdr->sbp = sense_buffer;
    p_scsi_hdr->mx_sb_len = sizeof(sense_buffer);
    cdb[0] = 0x12; /*This is for Inquiry*/
    cdb[1] = evpd & 1;
    cdb[2] = page_code & 0xff;
    cdb[3] = 0;
    cdb[4] = 0xff;
    cdb[5] = 0; /*For control filed, just use 0 */
    p_scsi_hdr->dxfer_direction = SG_DXFER_FROM_DEV;
    p_scsi_hdr->cmdp = cdb; //将数组的cdb命令写入结构体
    p_scsi_hdr->cmd_len = 6; //写入cdb命令数组长度6字节
    int fd = open("/dev/sg1", O_RDWR);
    if(fd<0)
        printf("open sg0 error..\n");
    int ret = ioctl(fd, SG_IO, p_scsi_hdr);
    if (ret<0)
        printf("Sending SCSI Command failed.\n");
    close(fd);
}
return p_scsi_hdr;
}
```

给 dxfer 一个缓存，硬盘经过 ioctl
后返回的数据就放在这里

读取硬盘设备基本信息

- " dxfer_direction : 用于确定数据传输的方向；常常使用以下值之一：
- " SG_DXFER_NONE : 不需要传输数据。比如 SCSI Test Unit Ready 命令。
- " SG_DXFER_TO_DEV : 将数据传输到设备。使用 SCSI WRITE 命令。
- " SG_DXFER_FROM_DEV : 从设备输出数据。使用 SCSI READ 命令。
- " SG_DXFER_TO_FROM_DEV : 双向传输数据。
- " SG_DXFER_UNKNOWN : 数据的传输方向未知。

把 cdb 命令写入硬盘，硬盘返回
数据给 dxferp

```

5 void show_vendor(struct sg_io_hdr* hdr1) {
6     unsigned char* buffer = hdr1->dxferp;
7     int i;
8     printf("vendor id:");
9     for (i=8; i<16; ++i) {
10         putchar(buffer[i]);
11     }
12     putchar('\n');
13 } // 我挨着 dxferp 数组的顺序把供应商名字读出来

14 void show_product(struct sg_io_hdr* hdr) {
15     unsigned char *buffer = hdr->dxferp;
16     int i;
17     printf("product id:");
18     for (i = 16; i<32; ++i) {
19         putchar(buffer[i]);
20     }
21     putchar('\n');
22 } // 我挨着 dxferp 数组的顺序把产品类型读出来

23 void show_product_rev(struct sg_io_hdr * hdr) {
24     unsigned char * buffer = hdr->dxferp;
25     int i;
26     printf("product ver:");
27     for (i = 32; i<36; ++i) {
28         putchar(buffer[i]);
29     }
30     putchar('\n');
31 } // 我挨着 dxferp 数组的顺序把产品版本号读出来
32 }
```

```

int main()
{
    struct sg_io_hdr *p = init_hdr(0,0);
    show_vendor(p);
    show_product(p);
    show_product_rev(p);
    free(p);
    return 0;
}
```

sg0
sg1

在/dev 下面有两个 scsi 设备，sg0, sg1

```
int fd = open("/dev/sg0", O_RDWR);
```

下面是 scsi 读取 sg0 的结果

```
root@ubuntu:/home/xzz/code/C/scsi# cat /proc/scsi/scsi
Attached devices:
Host: scsi3 Channel: 00 Id: 00 Lun: 00
  Vendor: NECVMWar Model: VMware SATA CD01 Rev: 1.00
  Type:  CD-ROM                         ANSI  SCSI revision: 05
Host: scsi32 Channel: 00 Id: 00 Lun: 00
  Vendor: VMware, Model: VMware Virtual S Rev: 1.0
  Type:  Direct-Access                   ANSI  SCSI revision: 02
root@ubuntu:/home/xzz/code/C/scsi# ./scsi
vendor id:NECVMWar
product id:VMware SATA CD01
product ver:1.00
root@ubuntu:/home/xzz/code/C/scsi#
```

```
int fd = open("/dev/sg1", O_RDWR);
```

下面这是 sg1 的结果

```
root@ubuntu:/home/xzz/code/C/scsi# cat /proc/scsi/scsi
Attached devices:
Host: scsi3 Channel: 00 Id: 00 Lun: 00
  Vendor: NECVMWar Model: VMware SATA CD01 Rev: 1.00
  Type:  CD-ROM                         ANSI  SCSI revision: 05
Host: scsi32 Channel: 00 Id: 00 Lun: 00
  Vendor: VMware, Model: VMware Virtual S Rev: 1.0
  Type:  Direct-Access                   ANSI  SCSI revision: 02
root@ubuntu:/home/xzz/code/C/scsi# ./scsi
vendor id:VMware,
product id:VMware Virtual S
product ver:1.0
```

这就是 SCSI 命令取读取 SCSI 总线上设备信息的方法

实际使用方法查看我的 linux USB 3G_4 模块配置使用教程

ARM Linux 动态库编译链接和指定新的默认动态库存放位置

1. 在 X86 的 linux(ubuntu) 平台下编译动态链接库，然后使用

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# ls  
apptest.c test1.c test2.c test.h
```

apptest.c 程序调用 test1.c 和 test2.c 里面的 test1() 函数和 test2() 函数

我们将 test1.c 和 test2.c 编译成.so 的动态库，然后供其 apptest.c 程序来调用

```
gcc test1.c test2.c -fPIC -shared -o libtest.so
```

将 test1.c 和 test2.c 里面程序全部编译进 libtest.so 库文件里面，这样 apptest.c 调用 test.so 就是了，前面的 lib 是 linux 系统要求加的。

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# ls  
apptest.c libtest.so test1.c test2.c test.h
```

你看 libtest.so 库文件编译出来了。

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# gcc -o apptest apptest.c -ltest  
/usr/bin/ld: cannot find -ltest  
collect2: error: ld returned 1 exit status
```

编译的时候 apptest 应用程序需要调用 libtest.so 里面的函数，所以 gcc 编译的时候直接-ltest 就是了。因为 test 就是 libtest.so 的简写，必须是-ltest 简写，不能用全称(libtest.so)
哦！！，这是 linux 的规矩，很扯淡

发现编译找不到-ltest 的库文件

这是因为 linux 的应用程序需要调用库文件的函数，默认是去 /usr/lib, /lib 这两个目录找。

```
root@ubuntu:/etc/ld.so.conf.d# ls  
fakeroot-x86_64-linux-gnu.conf  vmware-tools-libraries.conf  x86_64-linux-gnu_EGL.conf  zz_i386-biarch-compat.conf  
libc.conf                      x86_64-linux-gnu.conf          x86_64-linux-gnu_GL.conf  zz_x32-biarch-compat.conf  
root@ubuntu:/etc/ld.so.conf.d#
```

这个开机默认寻找库文件路径是在 /etc/ld.so.conf.d 里面的某个文件设置的

```
libtest.so  test1.c  test2.c  test.h  
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# mv libtest.so /usr/lib
```

我们将 libtest.so 文件拷贝到 /usr/lib 或者 /lib 目录下就是了。

```
solib# gcc -o apptest apptest.c -ltest  
solib#
```

再次编译就成功了

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# ./apptest  
test11111  
test22222
```

你看 apptest 调用了 libtest.so 库文件里面的 test1.c 和 test2.c。

其实整个过程我们发现和头文件 test.h 关系不大。这是 C 语言规则，因为主程序调用其他 C 文件是不需要头文件的，头文件只是帮助你方便查询其他 C 文件的函数而已

2.如果我们将.so 库程序编译到 ARM linux 开发板上怎么做呢?

Arm linux 开发板编译.so 文件,要在 X86 上用交叉编译器的 gcc 才行

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# arm9260t-gcc-4.3.2 test1.c test2.c -fPIC -shared -o libtest.so
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# ls
apptest.c libtest.so test1.c test2.c test.h
root@ubuntu:/home/xiang/ASM9260T/apptest/solib#
```

这个 libtest.so 文件放在 X86 linux 的/usr/lib 下面是不行的,

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# arm9260t-gcc-4.3.2 -o apptest apptest.c -ltest
/opt/ASM9260T_CROSS/arm-2008q3-linux/bin/../lib/gcc/arm-none-linux-gnueabi/4.3.2/../../../../arm-none-linux-gnueabi/bin/ld: cannot find -ltest
collect2: ld returned 1 exit status
```

apptest 编译的时候找不到。libtest.so 要放在交叉编译器的目录下。因为 apptest 应用程序交叉编译需要找库文件才能编译出来,然而交叉编译器指定的库文件在自己交叉编译要求的目录下。

```
cp libtest.so /opt/ASM9260T_CROSS/arm-2008q3-linux/arm-none-linux-gnueabi/libc/usr/lib/
```

所以将库文件复制到交叉编译 lib 目录下,记住是复制到 libc 目录下的/usr/lib,一定要是 libc

```
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# arm9260t-gcc-4.3.2 -o apptest apptest.c -ltest
root@ubuntu:/home/xiang/ASM9260T/apptest/solib# ls
apptest apptest.c test1.c test2.c test.h
```

将 apptest 程序放入开发板测试

```
# ./apptest
./apptest: error while loading shared libraries: libtest.so: cannot open shared object file: No such file or directory
#
```

这是因为我们没有把 libtest.so 拷贝到开发板的/usr/lib 目录下。

1.libtest.so 拷贝到 X86 的交叉编译目录下是为了编译应用程序

2.libtest.so 再次拷贝到开发板文件系统/usr/lib 下,是开发板运行 apptest 时候能找到库文件

```
# cp libtest.so /usr/lib/
# ./apptest
test11111
test2222
*
```

我们将库文件拷贝到开发板/usr/lib 下,你看运行成功

第二种方法:现在我不想把库文件拷贝到/usr/lib 目录下,我想自己建立个目录让 apptest 应用程序默认去找我指定目录下的 libtest.so。

```
# /etc/profile: system-wide .profile file for the Bourne shells
echo "Processing /etc/profile..."
# no-op
# Set search library path
echo "Set search library path in /etc/profile"
export LD_LIBRARY_PATH=/lib:/usr/lib:/mnt/yaffs2/Cdatabox/lib
# Set user path
echo "Set user path in /etc/profile"
```

/etc/profile 启动文件里面 LD_LIBRARY_PATH 默认指定的是/lib, /usr/lib 目录,现在我在文件系统里给 profile 文件新增加一个默认路径,叫 /mnt/yaffs2/Cdatabox/lib

这样 apptest 就会去/lib 目录下找.so 文件,找不到的话就会跑到/usr/lib 目录下找,再找不到的话就跑到/mnt/yaffs2/Cdatabox/lib 目录下找。

```
# 
/mnt/yaffs2/Cdatabox/lib
# 我们在目录/mnt/yaffs2/Cdatabox 下建立了 lib 文件
# cp libtest.so /mnt/yaffs2/Cdatabox/lib/
# ./apptest
test11111
test2222
*
```

你看成功找到库文件