1. **Shell脚本语法---在Makefile等文件中常见**

**表1.1. 测试命令**

|  |  |
| --- | --- |
| [ -d DIR ] | 如果DIR存在并且是一个目录则为真 |
| [ -f FILE ] | 如果FILE存在且是一个普通文件则为真 |
| [ -z STRING ] | 如果STRING的长度为零则为真 |
| [ -n STRING ] | 如果STRING的长度非零则为真 |
| [ STRING1 = STRING2 ] | 如果两个字符串相同则为真 |
| [ STRING1 != STRING2 ] | 如果字符串不相同则为真 |
| [ ARG1 OP ARG2 ] | ARG1和ARG2应该是整数或者取值为整数的变量，OP是-eq（等于）-ne（不等于）-lt（小于）-le（小于等于）-gt（大于）-ge（大于等于）之中的一个 |

表 1.2. 带与、或、非的测试命令

|  |  |
| --- | --- |
| [ ! EXPR ] | EXPR可以是上表中的任意一种测试条件，!表示逻辑反 |
| [ EXPR1 -a EXPR2 ] | EXPR1和EXPR2可以是上表中的任意一种测试条件，-a表示逻辑与 |
| [ EXPR1 -o EXPR2 ] | EXPR1和EXPR2可以是上表中的任意一种测试条件，-o表示逻辑或 |

makefile中参数：

$0 这个程式的执行名字

$n 这个程式的第n个参数值，n=1..9

$\* 这个程式的所有参数,此选项参数可超过9个。

$# 这个程式的参数个数

$$ 这个程式的PID(脚本运行的当前进程ID号)

$! 执行上一个背景指令的PID(后台运行的最后一个进程的进程ID号)

$? 执行上一个指令的返回值 (显示最后命令的退出状态。0表示没有错误，其他任何值表明有错误)

$- 显示shell使用的当前选项，与set命令功能相同

$@ 跟$\*类似，但是可以当作数组用

**2.u-boot 配置过程分析**

拿到任何一个工程源码，首先是配置，然后是编译，有的需要安装，有的不需要， *u-boot* 也不例外，  
在 *u-boot-2014.04* 源码顶层目录下的 *README* 文件介绍了 *u-boot* 的历史，版本命名规则，目录组织架构，软件配置，如何添加一个新的板子等。目录组织结构在 *README* 中 *Directory Hierarchy* 一节有详细说明，下面列出一些常用的：  
*arch*：存放 *CPU* 架构相关的文件，里面的每个目录代表一种架构。  
*arch/arm/cpu*：存放 *arm CPU* 相关的文件，里面每个目录代表一种 *arm cpu  
board*：存放单板相关的文件  
*board/samsung*：存放三星公司生产的单板  
*include/configs*：存放每个单板的配置文件

软件配置在*README* 中 *Software Configuration* 一节有详细说明， *README* 告诉我们：要针对某个单  
板进行配置，需要执行： *make NAME\_config*，其中 *NAME* 在 *boards.cfg* 中列出。

*S5PV210*和*S5PC100* 都是 *Cortex-A8* 架构，比较类似，我以 *S5PC100* 为例进行说明。

myroot@myroot:~/TQ210N/u-boot-2014.04$ make smdkc100\_config

Configuring for smdkc100 board...

通过执行 *make smdkc100\_config*，其中*smdkc100* 是*smdkc100* 这个单板的一个目标名称，因为有的  
单板有多个配置目标，比如 *atmel* 公司的 *at91sam9263ek* 单板就有 *5* 个目标， *make* 命令首先会从顶层*Makefile* 中搜索匹配目标 *smdkc100\_config*，找到

%\_config:: outputmakefile

@$(MKCONFIG) -A $(@:\_config=) // MKCONFIG := $(srctree)/mkconfig

*%\_config* 前面的*%*是通配符，会匹配所有以*\_config* 为后缀的目标  
*::*是 *Makefile* 的中的多目标规则，可以同时跟多个目标，举例：

myroot@myroot:~/TQ210N/u-boot-2014.04$ make smdkc100\_config smdk2410\_config

Configuring for smdkc100 board...

Configuring for smdk2410 board...

*$(MKCONFIG)*是顶层目录下的一个可执行 *shell* 脚本文件， *$(@:\_config=)*会将所有目标中的后缀  
*\_config*去掉，得到*smdkc100*，然后 *make* 会执行命令：  
*u-boot-2014.04/mkconfig –A smdkc100*下面分析 *mkconfig*首先定义了几个变量：  
*TARGETS*： *Makefile* 的目标  
*arch*：体系架构，比如 *arm*、 *x86*、 *mips* 等  
*cpu*： *cpu* 类型，比如 *arm920t*、 *arm11* 等  
*board*：单板名称，比如 *smdk2410*、 *smdkc100* 等  
*vendor*：厂商名称，比如 *samsung*、 *freescale* 等  
*soc*：片上系统，比如 *s3c2410*、 *s3c2440*、 *s5pv210* 等

if [ \( $# -eq 2 \) -a \( "$1" = "-A" \) ] ; then

# Automatic mode

line=`awk '($0 !~ /^#/ && $7 ~ /^'"$2"'$/) { print $1, $2, $3, $4, $5, $6, $7, $8 }' $srctree/boards.cfg`

if [ -z "$line" ] ; then

echo "make: \*\*\* No rule to make target \`$2\_config'. Stop." >&2

exit 1

fi

set ${line}

# add default board name if needed

[ $# = 3 ] && set ${line} ${1}

fi

如果参数个数等于*2*，而且第 *1* 个参数等于“*-A*”，则执行：  
*line=`awk '($0 !~ /^#/ && $7 ~ /^’"$2"’$/) { print $1, $2, $3, $4, $5, $6, $7, $8 }' $srctree/boards.cfg`  
awk* 是一个非常强大的文本处理工具， *$srctree* 被替换为 *u-boot-2014.04*，即 *u-boot* 源码树目录， *awk*会读取 *u-boot-2014.04/boards.cfg* 中的每一行，如果和前面的表达式*($0 !~ /^#/ && $7 ~ /^'"$2"'$/)*相匹配，则执行*{ print $1, $2, $3, $4, $5, $6, $7, $8 }*。  
*boards.cfg* 保存了各种单板的相关信息，其格式为：  
*# Status, Arch, CPU:SPLCPU, SoC, Vendor, Board name, Target, Options, Maintainers  
################################################################################**Active arm armv7 s5pc1xx samsung smdkc100 smdc100 - Minkyu Kang<mk7.kang@samsung.com>  
awk* 在执行过程中， *$0* 代表当前整行， 同时将第一个字段存入*$1*，将第 *2* 个字段存入*$2*， 依此类推，*awk* 缺省按空格分段，可以通过*-F* 指定分隔符。

若果当前行不以*#*开头，而且第 *7* 个字段和 *mkconfig* 传进来的第 *2* 个参数（*smdkc100*） 相等，则分别将字段 *1~*字段 *8* 输出到 *line* 中保存，最终得到：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *line=Active* | *arm* | *armv7* | *s5pc1xx* | *samsung* | *smdkc100* | *smdkc100* | *–* |
| *$#=8 $1=Active* | 状态 |  |  |  |  |  |  |
| *$2=arm* | 架构 |  |  |  |  |  |  |
| *$3=armv7* | *CPU* |  |  |  |  |  |  |
| *$4=s5pc1xx* | *SOC* |  |  |  |  |  |  |
| *$5=samsung* | 厂商 |  |  |  |  |  |  |
| *$6=smdkc100* | 单板名称 |  |  |  |  |  |  |
| *$7=smdkc100* | 配置目标 |  |  |  |  |  |  |
| *$8=-* | 选项 |  |  |  |  |  |  |

# Strip all options and/or \_config suffixes

CONFIG\_NAME="${7%\_config}"

[ "${BOARD\_NAME}" ] || BOARD\_NAME="${7%\_config}"

*${7%\_config}*表示去掉*${7}*的后缀*\_config*，这里*${7}=smdkc100*，得到  
*CONFIG\_NAME=smdkc100  
BOARD\_NAME=smdkc100*

arch="$2"

cpu=`echo $3 | awk 'BEGIN {FS = ":"} ; {print $1}'`

spl\_cpu=`echo $3 | awk 'BEGIN {FS = ":"} ; {print $2}'`

if [ "$6" = "<none>" ] ; then

board=

elif [ "$6" = "-" ] ; then

board=${BOARD\_NAME}

else

board="$6"

fi

[ "$5" != "-" ] && vendor="$5"

[ "$4" != "-" ] && soc="$4"

*arch=arm  
cpu=armv7  
spl\_cpu=*

*board=smdkc100  
vendor=samsung*

*soc=s5pc1xx*

if [ "$options" ] ; then

echo "Configuring for ${BOARD\_NAME} - Board: ${CONFIG\_NAME}, Options: ${options}"

else

echo "Configuring for ${BOARD\_NAME} board..."

fi

*$8=- 所以*

*Configuring for smdkc100 board...*

#

# Create link to architecture specific headers

#

if [ -n "$KBUILD\_SRC" ] ; then @@KBUILD\_SRC=

mkdir -p ${objtree}/include

LNPREFIX=${srctree}/arch/${arch}/include/asm/

cd ${objtree}/include

mkdir -p asm

else

cd arch/${arch}/include

fi

rm -f asm/arch

if [ -z "${soc}" ] ; then

ln -s ${LNPREFIX}arch-${cpu} asm/arch

else

ln -s ${LNPREFIX}arch-${soc} asm/arch

fi

if [ "${arch}" = "arm" ] ; then

rm -f asm/proc

ln -s ${LNPREFIX}proc-armv asm/proc

fi

if [ -z "$KBUILD\_SRC" ] ; then

cd ${srctree}/include

fi

KBUILD\_SRC=

创建架构相关的头文件的链接  
如果 *KBUILD\_SRC* 不为空，则进入到 *arch/arm/include* 目录，删除里面的 *asm/arch* 这个链接文件。  
如果*${soc}*为空，则执行 *ln -s ${LNPREFIX}arch-${cpu} asm/arch*，否则执行*ln -s ${LNPREFIX}arch-${soc} asm/arch*， 这里 *soc=s5pc1xx*， *LNPREFIX* 为空，所以会执行*ln –s arch-s5pc1xx asm/arch*

这里给*u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx* 创建了一个软连接，如下所示：

myroot@myroot:~/TQ210N/u-boot-2014.04/arch/arm/include/asm$ ll arch

lrwxrwxrwx 1 myroot myroot 12 12月 16 22:54 arch -> arch-s5pc1xx/

紧接着，如果*${arch} = “arm”*， 则删除 *asm/proc*，同时给 *proc-armv* 创建一个软链接，如下所示：

myroot@myroot:~/TQ210N/u-boot-2014.04/arch/arm/include/asm$ ll proc

lrwxrwxrwx 1 myroot myroot 9 12月 16 22:54 proc -> proc-armv/

如果 *KBUILD\_SRC* 为空，则进入到 *u-boot-2014.04/include*，这里 *KBUILD\_SRC* 为空，所以会进入到  
*u-boot-2014.04/include* 目录。

#

# Create include file for Make

#

( echo "ARCH = ${arch}"

if [ ! -z "$spl\_cpu" ] ; then

echo 'ifeq ($(CONFIG\_SPL\_BUILD),y)'

echo "CPU = ${spl\_cpu}"

echo "else"

echo "CPU = ${cpu}"

echo "endif"

else

echo "CPU = ${cpu}"

fi

echo "BOARD = ${board}"

[ "${vendor}" ] && echo "VENDOR = ${vendor}"

[ "${soc}" ] && echo "SOC = ${soc}"

exit 0 ) > config.mk

# Assign board directory to BOARDIR variable

if [ -z "${vendor}" ] ; then

BOARDDIR=${board}

else

BOARDDIR=${vendor}/${board}

fi

为 *make* 创建头文件  
*ARCH=arm  
CPU=arm  
BOARD=smdkc100  
VENDOR=samsung  
SOC=s5pc1xx*

使用 *echo* 将这些信息一行一行的输出到 *u-boot-2014.04/include/config.mk* 中：

然后给变量 *BOARDDIR* 赋值为 *samsung/smdkc100*

#

# Create board specific header file

#

if [ "$APPEND" = "yes" ] # Append to existing config file

then

echo >> config.h

else

> config.h # Create new config file

fi

echo "/\* Automatically generated - do not edit \*/" >>config.h

for i in ${TARGETS} ; do

i="`echo ${i} | sed '/=/ {s/=/ /;q; } ; { s/$/ 1/; }'`"

echo "#define CONFIG\_${i}" >>config.h ;

done

echo "#define CONFIG\_SYS\_ARCH \"${arch}\"" >> config.h

echo "#define CONFIG\_SYS\_CPU \"${cpu}\"" >> config.h

echo "#define CONFIG\_SYS\_BOARD \"${board}\"" >> config.h

[ "${vendor}" ] && echo "#define CONFIG\_SYS\_VENDOR \"${vendor}\"" >> config.h

[ "${soc}" ] && echo "#define CONFIG\_SYS\_SOC \"${soc}\"" >> config.h

[ "${board}" ] && echo "#define CONFIG\_BOARDDIR board/$BOARDDIR" >> config.h

cat << EOF >> config.h

#include <config\_cmd\_defaults.h>

#include <config\_defaults.h>

#include <configs/${CONFIG\_NAME}.h>

#include <asm/config.h>

#include <config\_fallbacks.h>

#include <config\_uncmd\_spl.h>

EOF

创建单板相关的头文件

*APPEND=no*，所以会创建一个新的 *u-boot-2014.04/include/config.h*

/\* Automatically generated - do not edit \*/

#define CONFIG\_SYS\_ARCH "arm"

#define CONFIG\_SYS\_CPU "armv7"

#define CONFIG\_SYS\_BOARD "smdkc100"

#define CONFIG\_SYS\_VENDOR "samsung"

#define CONFIG\_SYS\_SOC "s5pc1xx"

#define CONFIG\_BOARDDIR board/samsung/smdkc100

#include <config\_cmd\_defaults.h>

#include <config\_defaults.h>

#include <configs/smdkc100.h>

#include <asm/config.h>

#include <config\_fallbacks.h>

#include <config\_uncmd\_spl.h>

在这个头文件中定义了单板相关的宏，包含了单板的配置头文件 *configs/smdkc100.h*，我们需要针对  
自己的单板配置这个头文件。

**总结： 执行 *make smdkc100\_config* 主要做了 *3* 件事：**

1. 创建架构相关的头文件的链接

myroot@myroot:~/TQ210N/u-boot-2014.04/arch/arm/include/asm$ ll arch

lrwxrwxrwx 1 myroot myroot 12 12月 16 22:54 arch -> arch-s5pc1xx/

myroot@myroot:~/TQ210N/u-boot-2014.04/arch/arm/include/asm$ ll proc

lrwxrwxrwx 1 myroot myroot 9 12月 16 22:54 proc -> proc-armv/

1. 为 *make* 创建头文件 *u-boot-2014.04/include/config.mk*

ARCH=arm  
CPU=arm  
BOARD=smdkc100  
VENDOR=samsung  
SOC=s5pc1xx

*3)* 创建单板相关的头文件 *u-boot-2014.04/include/config.h*

**3.u-boot编译过程分析**

**4.SPL**

参考 *doc/README.SPL*

我们知道， *S5PV210*启动过程需要 *BL1*， *BL2*，其中 *BL1* 进行一些基本初始化（时钟、内存），加载*BL2* 到内存。 *U-boot*为此专门实现了*SPL* 框架。  
 *SPL* 的全称为 *secondary program loader*，即第 *2* 阶段程序加载器，即我们要实现的 *BL1*。  
要支持*SPL*，我们需要在单板配置文件 *u-boot-2014.04/include/configs/smdkc100.h* 定义宏 *CONFIG\_SPL  
#define CONFIG\_SPL*这样顶层 *Makefile* 在包含 *u-boot-2014.04/include/configs/smdkc100.h* 时，会得到*ALL-y += spl/u-boot-spl.bin*

这样在执行 *make all* 时， *make* 会编译 *spl/u-boot-spl.bin* 这个目标

spl/u-boot-spl.bin: spl/u-boot-spl

@:

spl/u-boot-spl: tools prepare

$(Q)$(MAKE) obj=spl -f $(srctree)/spl/Makefile all

从这得知， *spl/u-boot-spl.bin* 依赖于 *spl/u-boot-spl*，然后 *make* 找到目标 *spl/u-boot-spl*，然后执行指定的 *Makefile* 文件： *u-boot-2014.04/spl/Makefile*。  
 *u-boot-2014.04/spl/Makefile* 中首先导出环境变量 *CONFIG\_SPL\_BUILD=y*， 这个宏在各个源代码文件中用来控制代码的走向。

CONFIG\_SPL\_BUILD := y

export CONFIG\_SPL\_BUILD  
 这样编译后，最终会在 *u-boot-2014.04/spl* 下生成 *u-boot-spl.bin*， 同时在 *u-boot-2014.04* 下生成  
*u-boot.bin  
 u-boot.bin* 使用的链接脚本为 *u-boot-2014.04/arch/arm/cpu/u-boot.lds  
 u-boot-spl.bin* 使用的链接脚本为 *u-boot-2014.04/arch/arm/cpu/u-boot-spl.lds*

1. **添加自己的单板**

在 *u-boot* 的帮助文档 *README* 中有介绍如何添加自己的单板：

If the system board that you have is not listed, then you will need

to port U-Boot to your hardware platform. To do this, follow these

steps:

1. Add a new configuration option for your board to the toplevel

"boards.cfg" file, using the existing entries as examples.

Follow the instructions there to keep the boards in order.

2. Create a new directory to hold your board specific code. Add any

files you need. In your board directory, you will need at least

the "Makefile", a "<board>.c", "flash.c" and "u-boot.lds".

3. Create a new configuration file "include/configs/<board>.h" for

your board

3. If you're porting U-Boot to a new CPU, then also create a new

directory to hold your CPU specific code. Add any files you need.

4. Run "make <board>\_config" with your new name.

5. Type "make", and you should get a working "u-boot.srec" file

to be installed on your target system.

6. Debug and solve any problems that might arise.

[Of course, this last step is much harder than it sounds.]

1. 首先在顶层目录下的 *boards.cfg* 中添加自己的单板的硬件配置信息，我们仿照 *smdkc100*

Active arm armv7 s5pc1xx samsung smdkv210 smdkv210 - cumtzwf [<cumtzwf@163.com>](mailto:<cumtzwf@163.com>)

1. 在顶层目录下的 *board* 目录下创建一个用于存储单板相关代码文件的目录，我们直接拷贝  
   *board/samsung/smdkc100/*目录为 *board/samsung/smdkv210*，然后修改里面的文件名

myroot@myroot:~/TQ210N/u-boot-2014.04/board/samsung/smdkv210$ mv smdkc100.c smdkv210.c

|  |  |
| --- | --- |
| Makefile修改前：  obj-y := smdkc100.o  obj-$(CONFIG\_SAMSUNG\_ONENAND) += onenand.o  obj-y += lowlevel\_init.o | Makefile修改后：  obj-y := smdkv210.o  obj-$(CONFIG\_SAMSUNG\_ONENAND) += onenand.o  obj-y += lowlevel\_init.o |

1. 为自己的单板创建配置文件 *include/configs/<board>.h* ，也是直接拷贝 *smdkc100* 的配置文件

[myroot@myroot:~/TQ210N/u-boot-2014.04$](mailto:myroot@myroot:~/TQ210N/u-boot-2014.04$) cp include/configs/smdkc100.h include/configs/smdkv210.h

现在可以测试一下：

编写编译脚本：build-uboot.sh

myroot@myroot:~/TQ210N/u-boot-2014.04$ vim build-uboot.sh

make smdkv210\_config

make ARCH=arm CROSS\_COMPILE=arm-linux-

一切 *ok* 的话，最终在顶层目录下生成 *u-boot.bin*

***u-boot* 源码包含很多 *CPU* 架构和单板相关的文件，我们只关心 *S5PV210*，其它的都用不到，可以把  
不用的删掉：**

*u-boot-2014.04/arch* 目录下只保留 *arm* 目录

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls arch/

arm

*u-boot-2014.04/arch/arm* 目录下保留内容如下：

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls arch/arm/

config.mk cpu include lib

*u-boot-2014.04/arch/arm/cpu* 目录下保留内容如下：

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls arch/arm/cpu/

armv7 Makefile u-boot.lds u-boot-spl.lds

*u-boot-2014.04/arch/arm/include/asm* 目录下以 *arch-*开头的目录只保留 *arch-s5pc1xx*

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls arch/arm/include/asm/

arch-s5pc1xx byteorder.h errno.h kona-common omap\_gpio.h ptrace.h u-boot-arm.h

armv7.h cache.h gic.h linkage.h omap\_mmc.h sections.h u-boot.h

armv8 config.h global\_data.h mach-types.h omap\_musb.h setup.h unaligned.h

assembler.h davinci\_rtc.h gpio.h macro.h pl310.h spl.h utils.h

atomic.h dma-mapping.h hardware.h memory.h posix\_types.h string.h

bitops.h ehci-omap.h imx-common omap\_boot.h proc-armv system.h

bootm.h emif.h io.h omap\_common.h processor.h types.h

*u-boot-2014.04/board* 目录下只保留 *samsung*

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls board/

samsung

*u-boot-2014.04/board/samsung* 目录下只保留如下内容

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls board/samsung/

common smdkv210

*u-boot-2014.04/include/configs* 目录下只保留 *smdkv210.h*

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls include/configs/

smdkv210.h

把多余的文件删除后，同时执行 *make distclean* 清楚编译产生的目标文件后，经过压缩后，大小只  
有 *4MB*，原来有 *9M* 多

1. **移植 u-boot-spl.bin**

首先大致分析一下 *u-boot* 的代码走向， 通过看链接脚本 *u-boot-2014.04/arch/arm/cpu/u-boot.lds* 和*u-boot-2014.04/arch/arm/cpu/u-boot-spl.lds* 得知，不管是 *u-boot.bin* 还是 *u-boot-spl.bin* 都以  
*u-boot-2014.04/arch/arm/cpu/armv7/start.S* 开始，同时在链接脚本文件中指定了程序的入口函数  
*ENTRY(\_start)*

.globl \_start

\_start: b reset

ldr pc, \_undefined\_instruction

ldr pc, \_software\_interrupt

ldr pc, \_prefetch\_abort

ldr pc, \_data\_abort

ldr pc, \_not\_used

ldr pc, \_irq

ldr pc, \_fiq

*start.S* 首先安装了异常向量表， 执行第 *1* 条指令，然后跳转到这里：

reset:

bl save\_boot\_params

/\*

\* disable interrupts (FIQ and IRQ), also set the cpu to SVC32 mode,

\* except if in HYP mode already

\*/

mrs r0, cpsr

and r1, r0, #0x1f @ mask mode bits

teq r1, #0x1a @ test for HYP mode

bicne r0, r0, #0x1f @ clear all mode bits

orrne r0, r0, #0x13 @ set SVC mode

orr r0, r0, #0xc0 @ disable FIQ and IRQ

msr cpsr,r0

接着往下执行到：

/\* the mask ROM code should have PLL and others stable \*/

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

bl cpu\_init\_cp15

bl cpu\_init\_crit

#endif

这里有个判断，如果没有定义 *CONFIG\_SKIP\_LOWLEVEL\_INIT* 这个宏，则进行一些底层的初始化（不  
跳过底层初始化），其中 *cpu\_init\_crit* 又调用了 *lowlevel\_init*，这个函数在  
*u-boot-2014.04/board/samsung/smdkv210/lowlevel\_init.S* 中定义，我们可以在里面进行时钟、内存等  
底层的初始化。然后跳转到函数*\_main*， 这个函数在 *u-boot-2014.04/arch/arm/lib/crt0.S* 中定义， *crt* 即 *C Run-time*

/\*

\* entry point of crt0 sequence

\*/

ENTRY(\_main)

/\*

\* Set up initial C runtime environment and call board\_init\_f(0).

\*/

#if defined(CONFIG\_SPL\_BUILD) && defined(CONFIG\_SPL\_STACK)

ldr sp, =(CONFIG\_SPL\_STACK)

#else

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

#endif

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

sub sp, sp, #GD\_SIZE /\* allocate one GD above SP \*/

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

mov r9, sp /\* GD is above SP \*/

mov r0, #0

bl board\_init\_f

*ENTRY(\_main)*展开即为：  
*.globl \_main;  
ALIGN;  
\_main:*

接着调用函数 *board\_init\_f*，这个函数在*u-boot-2014.04/arch/arm/lib/board.c* 中定义， 进行一些基本  
的硬件初始化，为进入 *DRAM* 内存运行做准备，然后继续往下执行。

/\*

\* Set up intermediate environment (new sp and gd) and call

\* relocate\_code(addr\_moni). Trick here is that we'll return

\* 'here' but relocated.

\*/

ldr sp, [r9, #GD\_START\_ADDR\_SP] /\* sp = gd->start\_addr\_sp \*/

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

ldr r9, [r9, #GD\_BD] /\* r9 = gd->bd \*/

sub r9, r9, #GD\_SIZE /\* new GD is below bd \*/

adr lr, here

ldr r0, [r9, #GD\_RELOC\_OFF] /\* r0 = gd->reloc\_off \*/

add lr, lr, r0

ldr r0, [r9, #GD\_RELOCADDR] /\* r0 = gd->relocaddr \*/

b relocate\_code

这里调用 *relocate\_code*，重定位，将 *u-boot* 搬运到 *DRAM* 内存

/\* Set up final (full) environment \*/

bl c\_runtime\_cpu\_setup /\* we still call old routine here \*/

ldr r0, =\_\_bss\_start /\* this is auto-relocated! \*/

ldr r1, =\_\_bss\_end /\* this is auto-relocated! \*/

mov r2, #0x00000000 /\* prepare zero to clear BSS \*/

clbss\_l:cmp r0, r1 /\* while not at end of BSS \*/

strlo r2, [r0] /\* clear 32-bit BSS word \*/

addlo r0, r0, #4 /\* move to next \*/

blo clbss\_l

bl coloured\_LED\_init

bl red\_led\_on

/\* call board\_init\_r(gd\_t \*id, ulong dest\_addr) \*/

mov r0, r9 /\* gd\_t \*/

ldr r1, [r9, #GD\_RELOCADDR] /\* dest\_addr \*/

/\* call board\_init\_r \*/

ldr pc, =board\_init\_r /\* this is auto-relocated! \*/

/\* we should not return here. \*/

这里执行清 *BSS* 操作，然后把函数 *board\_init\_r* 的链接地址赋给 *PC* 指针，直接跳转到 *DRAM* 内存运行，在 *board\_init\_r* 继续执行一些硬件初始化，最终执行到这里

/\* main\_loop() can return to retry autoboot, if so just run it again \*/

for (;;)

main\_loop();

*main\_loop* 在*u-boot-2014.04/common/main.c* 中定义，在*u-boot-2014.04/common/board\_r.c中调用*， 在 *main\_loop* 中会检测用户的输入，如果在指定的延时时间内，没有用户按下按钮，则根据环境变量，从 *SD* 卡或 *FLASH* 拷贝内核镜像到 *DRAM*，然后跳转到 *OS* 起始地址执行 *OS*。 这里只是大致分析了一下代码的走向，并没有详细分析。 具体可以根据代码走向一步一步分析代码。

**首先移植 *u-boot-spl.bin***

参照u-boot-2014.04/doc/README.SPL

The building of SPL images can be with:

#define CONFIG\_SPL

我们采用*SPL* 方式，因此需要在单板配置文件 *u-boot-2014.04/include/configs/smdkv210.h* 中添加宏

#define CONFIG\_SPL

#endif /\* \_\_CONFIG\_H \*/

我们可以进行如下操作，只编译*u-boot-spl.bin*

myroot@myroot:~/TQ210N/u-boot-2014.04$ make smdkv210\_config

Configuring for smdkv210 board...

myroot@myroot:~/TQ210N/u-boot-2014.04$ make ARCH=arm CROSS\_COMPILE=arm-linux- spl/u-boot-spl.bin

编译出错：

LD spl/arch/arm/cpu/built-in.o

LDS spl/u-boot-spl.lds

LD spl/u-boot-spl

arch/arm/lib/built-in.o: In function `\_main':

/home/myroot/TQ210N/u-boot-2014.04/arch/arm/lib/crt0.S:74: undefined reference to `board\_init\_f'

/home/myroot/TQ210N/u-boot-2014.04/spl/Makefile:225: recipe for target 'spl/u-boot-spl' failed

make[1]: \*\*\* [spl/u-boot-spl] Error 1

Makefile:1082: recipe for target 'spl/u-boot-spl' failed

make: \*\*\* [spl/u-boot-spl] Error 2

我们前面分析得知在 *u-boot-2014.04/arch/arm/lib/board.c* 中定义了一个函数*board\_init\_f*，这个函数  
进行了非常多的初始化操作，由于 *u-boot-spl.bin* 的最终目的只是把 *BL2* 从外部存储器（*SD* 卡、 *NAND*）拷贝到 *SDRAM*，所以 *u-boot-spl.bin* 只需初始化时钟、 *SDRAM*、 *NAND*，然后调用一个拷贝函数，拷贝完成后直接跳转到 *SDRAM* 执行*BL2*，就完事了，我们可以修改 *crt0.S*，不让其调用 *board\_init\_f*。

首先在 *u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/cpu.h* 中添加与 *S5PV210* 相关的寄存器定  
义，后面会用到

/\* S5PV210 add by zwf \*/

#define S5PV210\_PRO\_ID 0xE0000000

#define S5PV210\_CLOCK\_BASE 0xE0100000

#define S5PV210\_GPIO\_BASE 0xE0200000

#define S5PV210\_PWMTIMER\_BASE 0xE2500000

#define S5PV210\_WATCHDOG\_BASE 0xE2700000

#define S5PV210\_UART\_BASE 0xE2900000

#define S5PV210\_SROMC\_BASE 0xE8000000

#define S5PV210\_MMC\_BASE 0xEB000000

#define S5PV210\_DMC0\_BASE 0xF0000000

#define S5PV210\_DMC1\_BASE 0xF1400000

#define S5PV210\_VIC0\_BASE 0xF2000000

#define S5PV210\_VIC1\_BASE 0xF2100000

#define S5PV210\_VIC2\_BASE 0xF2200000

#define S5PV210\_VIC3\_BASE 0xF2300000

在 *u-boot-2014.04/board/samsung/smdkv210/lowlevel\_init.S* 中的初始化都是汇编代码， 我还是喜欢用  
*C* 语言，再说 *S5PV210* 的性能如此之高，也不在乎损耗一点性能，将其修改为：

/\* modied by zwf \*/

.globl lowlevel\_init

lowlevel\_init:

mov r9, lr

#ifdef CONFIG\_SPL\_BUILD

bl clock\_init /\* 时钟初始化 \*/

bl ddr\_init /\* DDR初始化 \*/

#endif

mov pc, r9 /\* 返回 \*/

由于这些操作只需要在 *u-boot-spl.bin* 中进行，因此这里使用 *CONFIG\_SPL\_BUILD* 宏来控制，当编译*u-boot-spl.bin* 才会将这些初始化代码编译进 *u-boot-spl.bin*，而编译 *u-boot.bin* 时就不会。  
这里调用了两个函数， 我们在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中实现它，参照  
第 *4* 章裸机编程。由于这两个函数只需要在 *u-boot-spl.bin* 中实现，而 *u-boot.bin* 不需要，同时 *smdkv210.c*中的其他函数只需要在 *u-boot.bin* 中实现，前面分析过，编译 *u-boot-spl.bin* 时， *spl/Makefile* 会导出一个宏 *CONFIG\_SPL\_BUILD*，我们通过这个宏来控制代码是否被编译，下面列出修改后的框架：

*u-boot-2014.04/board/samsung/smdkv210/smdkv210.c*

#ifndef CONFIG\_SPL\_BUILD /\* add by zwf \*/

#else /\* CONFIG\_SPL\_BUILD (add by zwf) \*/

void clock\_init(void)

#endif /\* CONFIG\_SPL\_BUILD (add by zwf) \*/

实现 *clock\_init* 会用到时钟相关的寄存器，需要在  
*u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/clock.h* 中添加 *S5PV210* 的时钟相关寄存器的定义

/\* add by zwf \*/

struct s5pv210\_clock {

unsigned int apll\_lock;

unsigned char res1[0x04];

unsigned int mpll\_lock;

unsigned char res2[0x04];

unsigned int epll\_lock;

unsigned char res3[0x0C];

unsigned int vpll\_lock;

unsigned char res4[0xdc];

unsigned int apll\_con0;

unsigned int apll\_con1;

unsigned int mpll\_con;

unsigned char res5[0x04];

unsigned int epll\_con0;

unsigned int epll\_con1;

unsigned char res6[0x08];

unsigned int vpll\_con;

unsigned char res7[0xdc];

unsigned int src0;

unsigned int src1;

unsigned int src2;

unsigned int src3;

unsigned int src4;

unsigned int src5;

unsigned int src6;

unsigned char res8[0x64];

unsigned int mask0;

unsigned int mask1;

unsigned char res9[0x78];

unsigned int div0;

unsigned int div1;

unsigned int div2;

unsigned int div3;

unsigned int div4;

unsigned int div5;

unsigned int div6;

unsigned int div7;

};

在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中添加头文件

#include <asm/arch/clock.h> /\* add by zwf \*/

这里的 *arch* 链接到 *arch-s5pc1xx*，前面分析过在*clock\_init* 函数中使用到了一个宏 *samsung\_get\_base\_clock*struct s5pv210\_clock \*const clock = (struct s5pv210\_clock \*)samsung\_get\_base\_clock();  
这个宏在 *u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/cpu.h* 中定义

/\* modied by zwf \*/ \

#define SAMSUNG\_BASE(device, base) \

static inline unsigned int samsung\_get\_base\_##device(void) \

{ \

return S5PV210\_##base; \

}

SAMSUNG\_BASE(clock, CLOCK\_BASE)

SAMSUNG\_BASE(gpio, GPIO\_BASE)

SAMSUNG\_BASE(pro\_id, PRO\_ID)

SAMSUNG\_BASE(mmc, MMC\_BASE)

SAMSUNG\_BASE(sromc, SROMC\_BASE)

SAMSUNG\_BASE(timer, PWMTIMER\_BASE)

SAMSUNG\_BASE(uart, UART\_BASE)

SAMSUNG\_BASE(watchdog, WATCHDOG\_BASE)

/\* add by zwf \*/

SAMSUNG\_BASE(dmc0, DMC0\_BASE)

SAMSUNG\_BASE(dmc1, DMC1\_BASE)

这个宏展开即为：

SAMSUNG\_BASE(clock, CLOCK\_BASE)

static inline unsigned int samsung\_get\_base\_clock(void)\

{\

if(cpu\_is\_s5pc100()) \

return S5PC100\_CLOCK\_BASE; \

else if(cpu\_is\_s5pc110()) \

return S5PC110\_CLOCK\_BASE; \

else \

return 0; \

}

这里首先判断 *CPU* 类型，这里只判断了 *S5PC100*和 *S5PC110*，这里判断 *CPU* 类型同样使用的是宏

#define IS\_SAMSUNG\_TYPE(type, id) \

static inline int cpu\_is\_##type(void) \

{ \

return s5p\_cpu\_id == id ? 1 : 0; \

}

IS\_SAMSUNG\_TYPE(s5pc100, 0xc100)

IS\_SAMSUNG\_TYPE(s5pc110, 0xc110)

这个宏展开即为：

#define IS\_SAMSUNG\_TYPE(s5pc100, 0xc100) \

static inline int cpu\_is\_s5pc100(void) \

{ \

return s5p\_cpu\_id == 0xc100 ? 1 : 0; \

}

这里判断了一个变量 *s5p\_cpu\_id*，这个变量通过 *s5p\_set\_cpu\_id* 函数读取寄存器 *PRO\_ID* 来设置  
我们不用那么麻烦，直接修改 *SAMSUNG\_BASE* 这个宏，让它直接返回 *S5PV210\_##base*

/\* modied by zwf \*/ \

#define SAMSUNG\_BASE(device, base) \

static inline unsigned int samsung\_get\_base\_##device(void) \

{ \

return S5PV210\_##base; \

}

*clock\_init* 函数完成了，接下来需要实现 *ddr\_init*，同样首先需要定义寄存器，仿照 *clock.h*，编写  
*u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/dmc.h*

/\* add by zwf \*/

#ifndef \_\_ASM\_ARM\_ARCH\_DRAM\_H\_

#define \_\_ASM\_ARM\_ARCH\_DRAM\_H\_

#ifndef \_\_ASSEMBLY\_\_

struct s5pv210\_dmc0 {

unsigned int concontrol;

unsigned int memcontrol;

在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中添加头文件

#include <asm/arch/dmc.h> /\* add by zwf \*/

同时在 *u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/cpu.h* 中添加宏

/\* add by zwf \*/

SAMSUNG\_BASE(dmc0, DMC0\_BASE)

SAMSUNG\_BASE(dmc1, DMC1\_BASE)

现在就可以仿照 *clock\_init* 实现 *ddr\_init* 了，具体请看源码

接着执行到 *u-boot-2014.04/arch/arm/lib/crt0.S* 中的*\_main* 函数

|  |  |
| --- | --- |
| 修改前：  #if defined(CONFIG\_SPL\_BUILD) && defined(CONFIG\_SPL\_STACK)  ldr sp, =(CONFIG\_SPL\_STACK)  #else  ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)  #endif | 修改后：  /\* modied by zwf \*/  #if !defined(CONFIG\_SPL\_BUILD)  ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR) |

这里在设置栈指针， 由于 *u-boot-spl.bin* 相当小，使用 *SRAM* 为其提供栈内存就够了，所以可以将这  
里的栈指针设置代码屏蔽掉（为 *u-boot-spl.bin* 屏蔽，对 *u-boot.bin* 没有影响），如下：

#ifdef CONFIG\_SPL\_BUILD

bl copy\_bl2\_to\_ram /\* 拷贝BL2到DDR \*/

ldr pc, =CONFIG\_SYS\_SDRAM\_BASE /\* 跳转到DDR的起始地址执行BL2 \*/

#else

bl board\_init\_f

#endif

这里调用了一个函数 *copy\_bl2\_to\_ram*， 用于从 *SD* 卡或 *NAND* 拷贝 *BL2* 到 *SDRAM*在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中实现它，拷贝完成后，直接跳转到 *DDR*的起始地址执行 *BL2*。*CONFIG\_SYS\_SDRAM\_BASE* 在 *u-boot-2014.04/include/configs/smdkv210.h* 中定义，我们将其修改为我们实际*DDR* 的起始地址 *0x20000000*

/\* DRAM Base \*/

#define CONFIG\_SYS\_SDRAM\_BASE 0x20000000 /\* modied by zwf \*/

在 *copy\_bl2\_to\_ram* 中我们实现从 *SD* 卡的扇区 *32* 拷贝 *BL2* 到 *DDR* 的起始地址 *0x20000000*

问：为什么是 *32*？

答：三星手册明确规定 *BL1* 的大小最大为 *16KB*，而每一个扇区为 *512B*， *16KB* 刚好在扇区 *32*。 这不是强制要求的， 只要这个数字乘以 *512* 小于你的 *u-boot-spl.bin* 就可以了。

void copy\_bl2\_to\_ram(void)

{

/\*

\*\* ch: 通道

\*\* sb: 起始块

\*\* bs: 块大小

\*\* dst: 目的地

\*\* i: 是否初始化

\*/

#define CopySDMMCtoMem(ch, sb, bs, dst, i) \

(((unsigned char(\*)(int, unsigned int, unsigned short, unsigned int\*, unsigned char))\

(\*((unsigned int \*)0xD0037F98)))(ch, sb, bs, dst, i))

unsigned int V210\_SDMMC\_BASE = \*(volatile unsigned int \*)(0xD0037488); // V210\_SDMMC\_BASE

unsigned char ch = 0;

/\* 参考S5PV210手册7.9.1 SD/MMC REGISTER MAP \*/

if (V210\_SDMMC\_BASE == 0xEB000000) // 通道0

ch = 0;

else if (V210\_SDMMC\_BASE == 0xEB200000) // 通道2

ch = 2;

CopySDMMCtoMem(ch, 32, 10, (unsigned int \*)CONFIG\_SYS\_SDRAM\_BASE, 0);

}

由于现在我们还不知道 *BL2* 有多大，所以拷贝多大先暂时随便写个值。

现在执行 *make spl/u-boot-spl.bin*，出现错误

myroot@myroot:~/TQ210N/u-boot-2014.04$ make ARCH=arm CROSS\_COMPILE=arm-linux- spl/u-boot-spl.bin

OBJCOPY spl/u-boot-spl.bin

/home/myroot/TQ210N/u-boot-2014.04/tools/mkexynosspl spl/u-boot-spl.bin spl/smdkv210-spl.bin

make[1]: /home/myroot/TQ210N/u-boot-2014.04/tools/mkexynosspl: Command not found

/home/myroot/TQ210N/u-boot-2014.04/spl/Makefile:200: recipe for target 'spl/smdkv210-spl.bin' failed

make[1]: \*\*\* [spl/smdkv210-spl.bin] Error 127

Makefile:1082: recipe for target 'spl/u-boot-spl' failed

make: \*\*\* [spl/u-boot-spl] Error 2

分析 *u-boot-2014.04/spl/Makefile*

ifdef CONFIG\_SAMSUNG

ALL-y += $(obj)/$(BOARD)-spl.bin

endif

all: $(ALL-y)

ifdef CONFIG\_SAMSUNG

ifdef CONFIG\_VAR\_SIZE\_SPL

VAR\_SIZE\_PARAM = --vs

else

VAR\_SIZE\_PARAM =

endif

$(obj)/$(BOARD)-spl.bin: $(obj)/u-boot-spl.bin

$(if $(wildcard $(objtree)/spl/board/samsung/$(BOARD)/tools/mk$(BOARD)spl),\

$(objtree)/spl/board/samsung/$(BOARD)/tools/mk$(BOARD)spl,\

$(objtree)/tools/mkexynosspl) $(VAR\_SIZE\_PARAM) $< $@

endif

如果定义了*CONFIG\_SAMSUNG*，就添加目标*$(obj)/$(BOARD)-spl.bin*

*CONFIG\_SAMSUNG* 在*u-boot-2014.04/include/configs/smdkv210.h* 中定义  
*$(obj)/$(BOARD)-spl.bin* 展开为 *spl/smdkv210-spl.bin*

这里需要调用一个工具用来给 *u-boot-spl.bin* 添加头信息，我们将之前制作头信息的工具（*addheader*）  
拷贝到*u-boot-2014.04/tools*里面，并更名为*mksmdkv210spl*

myroot@myroot:~/TQ210N/u-boot-2014.04$ ls tools/| grep mks

mksmdkv210spl

然后修改 *u-boot-2014.04/spl/Makefile*

$(obj)/$(BOARD)-spl.bin: $(obj)/u-boot-spl.bin

$(objtree)/tools/mk$(BOARD)spl $< $@

endif

现在再次执行 *make spl/u-boot-spl.bin* 编译，最终生成 *spl/smdkv210-spl.bin*，已经添加了 *16B* 头信息

OBJCOPY spl/u-boot-spl.bin

/home/myroot/TQ210N/u-boot-2014.04/tools/mksmdkv210spl spl/u-boot-spl.bin spl/smdkv210-spl.bin

测试一下 *spl/smdkv210-spl.bin*使用 *DD\_For\_Windows.exe* 将 *spl/smdkv210-spl.bin* 烧写到 *SD* 的扇区 *1*，将裸机程序  
*tq210\_NoOS/1-led\_on\_c* 中的 *led.bin* 烧写到 *SD* 卡的扇区 *32*



注意： 这里要取消“添加头信息”，因为 *spl/smdkv210-spl.bin* 已经添加了头信息， 如果烧写  
*spl/u-boot-spl.bin* 就需要选中“添加头信息”



然后取下 *SD* 卡，将其插入开发板，可以看到 *LED* 流水灯效果。

1. **u-boot.bin** 内存布局分析

*u-boot.bin* 从 *start.S* 中的 *reset* 开始执行，执行一些 *CPU* 底层初始化，然后跳转到 *crt0.S* 中的*\_main*函数。

/\* modied by zwf \*/

#if !defined(CONFIG\_SPL\_BUILD)

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

sub sp, sp, #GD\_SIZE /\* allocate one GD above SP \*/

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

mov r9, sp /\* GD is above SP \*/

mov r0, #0

#endif

/\* modied by zwf \*/

#ifdef CONFIG\_SPL\_BUILD

bl copy\_bl2\_to\_ram /\* 拷贝BL2到DDR \*/

ldr pc, =CONFIG\_SYS\_SDRAM\_BASE /\* 跳转到DDR的起始地址执行BL2 \*/

#else

bl board\_init\_f

#endif

这里仅为执行 *board\_init\_f* 函数设置栈，以及为全局变量 *gd* 预留一块存储空间。 在 *u-boot-spl.bin* 中  
调用了 *copy\_bl2\_to\_ram*，将 *u-boot.bin* 拷贝到 *DDR* 的基地址 *0x20000000*，然后给 *PC* 指针赋值为 *DDR*的基地址直接跳转到 *0x20000000* 地址取指执行。 *gd* 保存了一些重要的全局的变量，其定义在  
*u-boot-2014.04/arch/arm/include/asm/global\_data.h* 头文件中。

#ifdef CONFIG\_ARM64

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm ("x18")

#else

#define DECLARE\_GLOBAL\_DATA\_PTR register volatile gd\_t \*gd asm ("r9")

#endif

从这得知全局变量 *gd* 是一个寄存器变量， 保存在 *r9* 寄存器中， *gd\_t* 结构体定义在

*u-boot-2014.04/include/asm-generic/global\_data.h*

typedef struct global\_data {

bd\_t \*bd;

unsigned long flags;

unsigned int baudrate;

unsigned long cpu\_clk; /\* CPU clock in Hz! \*/

unsigned long bus\_clk;

/\* We cannot bracket this with CONFIG\_PCI due to mpc5xxx \*/

unsigned long pci\_clk;

unsigned long mem\_clk;

#if defined(CONFIG\_LCD) || defined(CONFIG\_VIDEO)

unsigned long fb\_base; /\* Base address of framebuffer mem \*/

#endif

#if defined(CONFIG\_POST) || defined(CONFIG\_LOGBUFFER)

unsigned long post\_log\_word; /\* Record POST activities \*/

unsigned long post\_log\_res; /\* success of POST test \*/

unsigned long post\_init\_f\_time; /\* When post\_init\_f started \*/

#endif

#ifdef CONFIG\_BOARD\_TYPES

unsigned long board\_type;

#endif

unsigned long have\_console; /\* serial\_init() was called \*/

#ifdef CONFIG\_PRE\_CONSOLE\_BUFFER

unsigned long precon\_buf\_idx; /\* Pre-Console buffer index \*/

#endif

#ifdef CONFIG\_MODEM\_SUPPORT

unsigned long do\_mdm\_init;

unsigned long be\_quiet;

#endif

unsigned long env\_addr; /\* Address of Environment struct \*/

unsigned long env\_valid; /\* Checksum of Environment valid? \*/

unsigned long ram\_top; /\* Top address of RAM used by U-Boot \*/

unsigned long relocaddr; /\* Start address of U-Boot in RAM \*/

phys\_size\_t ram\_size; /\* RAM size \*/

unsigned long mon\_len; /\* monitor len \*/

unsigned long irq\_sp; /\* irq stack pointer \*/

unsigned long start\_addr\_sp; /\* start\_addr\_stackpointer \*/

unsigned long reloc\_off;

struct global\_data \*new\_gd; /\* relocated global data \*/

#ifdef CONFIG\_DM

struct device \*dm\_root; /\* Root instance for Driver Model \*/

struct list\_head uclass\_root; /\* Head of core tree \*/

#endif

const void \*fdt\_blob; /\* Our device tree, NULL if none \*/

void \*new\_fdt; /\* Relocated FDT \*/

unsigned long fdt\_size; /\* Space reserved for relocated FDT \*/

void \*\*jt; /\* jump table \*/

char env\_buf[32]; /\* buffer for getenv() before reloc. \*/

#ifdef CONFIG\_TRACE

void \*trace\_buff; /\* The trace buffer \*/

#endif

#if defined(CONFIG\_SYS\_I2C)

int cur\_i2c\_bus; /\* current used i2c bus \*/

#endif

unsigned long timebase\_h;

unsigned long timebase\_l;

struct arch\_global\_data arch; /\* architecture-specific data \*/

} gd\_t;

*CONFIG\_SYS\_INIT\_SP\_ADDR* 在单板配置文件 *smdkv210.h* 中定义。这个可以随便设置，但要保证能够  
供 *board\_init\_f* 使用，以及存储 *GD ('global data')*，我们可以直接设置为 *DDR* 的最高地址 *0x60000000*

u-boot-2014.04/include/configs/*smdkv210.h*

#define CONFIG\_SYS\_INIT\_SP\_ADDR (CONFIG\_SYS\_SDRAM\_BASE + PHYS\_SDRAM\_1\_SIZE) /\* modied by zwf \*/

/\* SMDKC100 has 1 banks of DRAM, we use only one in U-Boot \*/

#define CONFIG\_NR\_DRAM\_BANKS 1

#define PHYS\_SDRAM\_1 CONFIG\_SYS\_SDRAM\_BASE /\* SDRAM Bank #1 \*/

#define PHYS\_SDRAM\_1\_SIZE (1024 << 20) /\* 0x40000000, 1024 MB Bank #1 \*//\*(modied by zwf)\*/

*GD\_SIZE* 在 *u-boot-2014.04/include/generated/generic-asm-offsets.h* 中定义，其值为 *160*

#define GENERATED\_GBL\_DATA\_SIZE 160 /\* (sizeof(struct global\_data) + 15) & ~15 @ \*/

#define GENERATED\_BD\_INFO\_SIZE 32 /\* (sizeof(struct bd\_info) + 15) & ~15 @ \*/

#define GD\_SIZE 160 /\* sizeof(struct global\_data) @ \*/

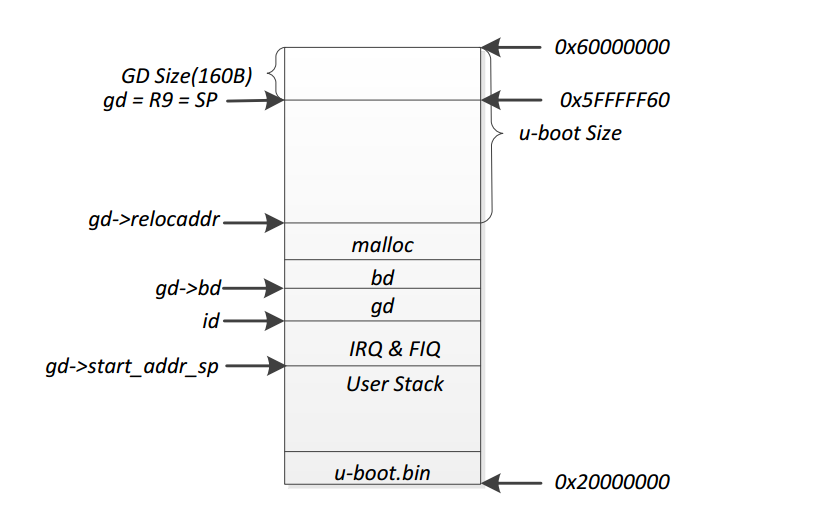
#define GD\_BD 0 /\* offsetof(struct global\_data, bd) @ \*/

#define GD\_RELOCADDR 44 /\* offsetof(struct global\_data, relocaddr) @ \*/

#define GD\_RELOC\_OFF 64 /\* offsetof(struct global\_data, reloc\_off) @ \*/

#define GD\_START\_ADDR\_SP 60 /\* offsetof(struct global\_data, start\_addr\_sp) @ \*/

设置后的内存布局如下图：



接着调用 *board.c* 中的 *board\_init\_f* 函数，给它传了一个参数 *r0=0*，事实上没有用到。这个函数使用

到了一些全局变量，因此我们需要让 *u-boot.bin* 位于其链接地址，通过修改 *smdkv210.h* 中的宏  
*CONFIG\_SYS\_TEXT\_BASE* 指定其链接地址。 我们将其指定为 *u-boot.bin* 在 *DDR* 中的起始地址

/\* DRAM Base \*/

#define CONFIG\_SYS\_SDRAM\_BASE 0x20000000 /\* modied by zwf \*/

/\* Text Base \*/

#define CONFIG\_SYS\_TEXT\_BASE 0x20000000 /\* modied by zwf \*/

继续分析 *board\_init\_f*

(/u-boot-2014.04/arch/arm/lib/board.c)

void board\_init\_f(ulong bootflag)

{

bd\_t \*bd;

init\_fnc\_t \*\*init\_fnc\_ptr;

gd\_t \*id;

ulong addr, addr\_sp;

首先定义了几个重要的变量： *addr* 最终为重定位地址， *addr\_sp* 为最终的用户栈指针地址。

memset((void \*)gd, 0, sizeof(gd\_t));

gd->mon\_len = (ulong)&\_\_bss\_end - (ulong)\_start;

初始化全局变量 *gd*，同时计算出 *u-boot.bin* 的大小，保存到 *gd->mon\_len*

for (init\_fnc\_ptr = init\_sequence; \*init\_fnc\_ptr; ++init\_fnc\_ptr) {

if ((\*init\_fnc\_ptr)() != 0) {

hang ();

}

}

依次调用数组 *init\_sequence* 中的每个函数，进行一系列初始化操作。

init\_fnc\_t \*init\_sequence[] = {

arch\_cpu\_init, /\* basic arch cpu dependent setup \*/

mark\_bootstage,

#ifdef CONFIG\_OF\_CONTROL

fdtdec\_check\_fdt,

#endif

#if defined(CONFIG\_BOARD\_EARLY\_INIT\_F)

board\_early\_init\_f,

#endif

timer\_init, /\* initialize timer \*/

#ifdef CONFIG\_BOARD\_POSTCLK\_INIT

board\_postclk\_init,

#endif

#ifdef CONFIG\_FSL\_ESDHC

get\_clocks,

#endif

env\_init, /\* initialize environment \*/

init\_baudrate, /\* initialze baudrate settings \*/

serial\_init, /\* serial communications setup \*/

console\_init\_f, /\* stage 1 init of console \*/

display\_banner, /\* say that we are here \*/

print\_cpuinfo, /\* display cpu info (and speed) \*/

#if defined(CONFIG\_DISPLAY\_BOARDINFO)

checkboard, /\* display board info \*/

#endif

#if defined(CONFIG\_HARD\_I2C) || defined(CONFIG\_SYS\_I2C)

init\_func\_i2c,

#endif

dram\_init, /\* configure available RAM banks \*/

NULL,

};

*arch\_cpu\_init* 在 *u-boot-2014.04/arch/arm/cpu/armv7/s5p-common/cpu\_info.c* 中定义，它调用了  
*s5p\_set\_cpu\_id* 读取 *CPU* 版本和 *ID*，保存到 *s5p\_cpu\_rev* 和 *s5p\_cpu\_id* 中。接着调用 *u-boot-2014.04/arch/arm/cpu/armv7/s5p-common/timer.c* 中的 *timer\_init* 初始化 *PWM* 定时器接着调用 *serial\_init* 初始化串口，通过跟踪代码发现在单板配置文件 *smdkv210.h* 中通过宏*CONFIG\_SERIAL0* 指定使用哪个串口

/\*

\* select serial console configuration

\*/

#define CONFIG\_SERIAL0 1 /\* use SERIAL 0 on SMDKC100 \*/

默认使用的是串口 *0*， 我们还需要为串口 *0* 配置 *GPIO* 端口，在  
*u-boot-2014.04/board/samsung/smdkv210/lowlevel\_init.S* 中添加代码

/\* modied by zwf \*/

.globl lowlevel\_init

lowlevel\_init:

mov r9, lr

#ifdef CONFIG\_SPL\_BUILD

bl clock\_init /\* 时钟初始化 \*/

bl ddr\_init /\* DDR初始化 \*/

/\*add by zwf\*/

ldr r0,=0xE0200000 @ GPA0CON UART 0/1

bic r1,r1,0x0000FFFF

eor r1,r1,0x00002222

str r1,[r0]

#endif

mov pc, r9 /\* 返回 \*/

接着调用 *display\_banner*，显示 *u-boot* 版本信息，我们可以在这个函数中添加显示自己的 *log*，例如：(u-boot-2014.04/arch/arm/lib/board.c )

static int display\_banner(void)

{

printf("\n\n%s\n\n", version\_string);

debug("U-Boot code: %08lX -> %08lX BSS: -> %08lX\n",

(ulong)&\_start,

(ulong)&\_\_bss\_start, (ulong)&\_\_bss\_end);

#ifdef CONFIG\_MODEM\_SUPPORT

debug("Modem Support enabled\n");

#endif

#ifdef CONFIG\_USE\_IRQ

debug("IRQ Stack: %08lx\n", IRQ\_STACK\_START);

debug("FIQ Stack: %08lx\n", FIQ\_STACK\_START);

#endif

return (0);

}

接着调用 *u-boot-2014.04/arch/arm/cpu/armv7/s5p-common/cpu\_info.c* 中的 *print\_cpuinfo*，打印 *CPU*名称和时钟

#ifdef CONFIG\_DISPLAY\_CPUINFO

int print\_cpuinfo(void)

{

char buf[32];

printf("CPU:\t%s%X@%sMHz\n",

s5p\_get\_cpu\_name(), s5p\_cpu\_id,

strmhz(buf, get\_arm\_clk()));

return 0;

}

#endif

*s5p\_get\_cpu\_name* 得到 *CPU* 的名称， 在*u-boot-2014.04/arch/arm/include/asm/arch-s5pc1xx/cpu.h* 中定义为：

#define \_S5PC1XX\_CPU\_H

#define S5P\_CPU\_NAME "S5P"

其中 *s5p\_cpu\_id* 为一个整数，对于 *S5PC100* 和 *S5PC110* 可以通过处理得到 *0xc100* 和 *0xc110*， 打印出来就是 *S5PC100* 和 *S5PC110*，但对于 *S5PV210* 就没法将那个 *v* 用整数表示了。 我们直接修改为打印*S5PV210*。

printf("CPU:\t%sV210$sMHz\n",s5p\_get\_cpu\_name(),strmhz(buf,get\_arm\_clk()));

*get\_arm\_clk* 根据 *CPU* 的 *ID* 决定调用那个函数来获取时钟，其定义在 *u-boot-2014.04/arch/arm/cpu/armv7/s5pc1xx/clock.c* 中

unsigned long get\_arm\_clk(void)

{

if (cpu\_is\_s5pc110())

return s5pc110\_get\_arm\_clk();

else

return s5pc100\_get\_arm\_clk();

}

我们仿照 *s5pc110\_get\_arm\_clk* 为 *S5PV210* 实现一个函数 *s5pv210\_get\_arm\_clk*， *S5PC110* 和 *S5PV210*比较相近，然后将 *get\_arm\_clk* 修改为:

unsigned long get\_arm\_clk(void)

{

/\*modied by zwf\*/

return s5pv210\_get\_arm\_clk();

}

/\* s5pv210: return ARM clock frequency (add by zwf)\*/

static unsigned long s5pv210\_get\_arm\_clk(void)

{

struct s5pv210\_clock \*clk =

(struct s5pv210\_clock \*)samsung\_get\_base\_clock();

unsigned long div;

unsigned long dout\_apll, armclk;

unsigned int apll\_ratio, arm\_ratio;

div = readl(&clk->div0);

/\* APLL\_RATIO: [2:0] \*/

apll\_ratio = div & 0x7;

dout\_apll = get\_pll\_clk(APLL) / (apll\_ratio + 1);

armclk = dout\_apll;

return armclk;

}

在 *s5pv210\_get\_arm\_clk* 中调用了 *get\_pll\_clk(APLL)*来获得 *APLL* 输出时钟，其定义为：

unsigned long get\_pll\_clk(int pllreg)

{

if (cpu\_is\_s5pc110())

return s5pc110\_get\_pll\_clk(pllreg);

else

return s5pc100\_get\_pll\_clk(pllreg);

}

我们将其修改为:

unsigned long get\_pll\_clk(int pllreg)

{

return s5pv210\_get\_pll\_clk(pllreg);

}

同样仿照 *s5pc110\_get\_pll\_clk* 为 *S5PV210* 实现一个函数 *s5pv210\_get\_pll\_clk*

/\* s5pv210: return pll clock frequency(add by zwf) \*/

static unsigned long s5pv210\_get\_pll\_clk(int pllreg)

{

struct s5pv210\_clock \*clk =

(struct s5pv210\_clock \*)samsung\_get\_base\_clock();

unsigned long r, m, p, s, mask, fout;

unsigned int freq;

switch (pllreg) {

case APLL:

r = readl(&clk->apll\_con);

break;

case MPLL:

r = readl(&clk->mpll\_con);

break;

case EPLL:

r = readl(&clk->epll\_con);

break;

case VPLL:

r = readl(&clk->vpll\_con);

break;

default:

printf("Unsupported PLL (%d)\n", pllreg);

return 0;

}

/\*

\* APLL\_CON: MIDV [25:16]

\* MPLL\_CON: MIDV [25:16]

\* EPLL\_CON: MIDV [24:16]

\* VPLL\_CON: MIDV [24:16]

\*/

if (pllreg == APLL || pllreg == MPLL)

mask = 0x3ff;

else

mask = 0x1ff;

m = (r >> 16) & mask;

/\* PDIV [13:8] \*/

p = (r >> 8) & 0x3f;

/\* SDIV [2:0] \*/

s = r & 0x7;

freq = CONFIG\_SYS\_CLK\_FREQ\_V210;

if (pllreg == APLL) {

if (s < 1)

s = 1;

/\* FOUT = MDIV \* FIN / (PDIV \* 2^(SDIV - 1)) \*/

fout = m \* (freq / (p \* (1 << (s - 1))));

} else

/\* FOUT = MDIV \* FIN / (PDIV \* 2^SDIV) \*/

fout = m \* (freq / (p \* (1 << s)));

return fout;

}

这里用到一个宏 *CONFIG\_SYS\_CLK\_FREQ\_V210*，在单板配置文件 *smdkv210.h* 中定义

/\* input clock of PLL: SMDKC100 has 12MHz input clock \*/

#define CONFIG\_SYS\_CLK\_FREQ 12000000

/\* input clock of PLL: SMDKV210 has 24MHz input clock(add by zwf) \*/

#define CONFIG\_SYS\_CLK\_FREQ\_V210 24000000

其实为 *S5PV210* 添的这 *2* 个函数和 *S5PC110* 的基本一样。

继续回到 *board.c* 中的 *init\_sequence*，接着调用*u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中的 *dram\_init*，其定义为 :

int dram\_init(void)

{

gd->ram\_size = get\_ram\_size((long \*)PHYS\_SDRAM\_1, PHYS\_SDRAM\_1\_SIZE);

return 0;

}

这里用到的宏 *PHYS\_SDRAM\_1* 和 *PHYS\_SDRAM\_1\_SIZE* 在 *smdkv210.h* 中定义，分别为 *SDRAM* 的基地址和大小。 这里得到内存大小，赋值给全局变量 *gd->ram\_size*。

继续往下执行

(u-boot-2014.04/arch/arm/lib/board.c)void board\_init\_f(ulong bootflag)

addr = CONFIG\_SYS\_SDRAM\_BASE + get\_effective\_memsize();

这句执行后得到： *addr=0x60000000*，即内存的最高地址

/\*

\* reserve memory for U-Boot code, data & bss

\* round down to next 4 kB limit

\*/

addr -= gd->mon\_len;

addr &= ~(4096 - 1);

debug("Reserving %ldk for U-Boot at: %08lx\n", gd->mon\_len >> 10, addr);

这里为重定位的 *u-boot* 预留一块内存空间， *gd->mon\_len* 为 *u-boot* 的大小

/\*

\* reserve memory for malloc() arena

\*/

addr\_sp = addr - TOTAL\_MALLOC\_LEN;

debug("Reserving %dk for malloc() at: %08lx\n",

TOTAL\_MALLOC\_LEN >> 10, addr\_sp);

这里开始计算栈指针地址， 为 *malloc* 预留一块内存空间，作为堆内存

/\*

\* (permanently) allocate a Board Info struct

\* and a permanent copy of the "global" data

\*/

addr\_sp -= sizeof (bd\_t);

bd = (bd\_t \*) addr\_sp;

gd->bd = bd;

debug("Reserving %zu Bytes for Board Info at: %08lx\n",

sizeof (bd\_t), addr\_sp);

这里为 *bd* 预留一块内存空间，同时使 *gd->bd* 指向现在的 *addr\_sp* 所在地址， *bd* 变量保存了单板的一些信息，比如机器码

#ifdef CONFIG\_MACH\_TYPE

gd->bd->bi\_arch\_number = CONFIG\_MACH\_TYPE; /\* board id for Linux \*/

#endif

这里将配置的机器码保存到 *gd->bd->bi\_arch\_number*， 这个机器码必须和内核的机器码相同，否则启动不了内核，我们可以在 *smdkv210.h* 中定义这个机器码。

addr\_sp -= sizeof (gd\_t);

id = (gd\_t \*) addr\_sp;

debug("Reserving %zu Bytes for Global Data at: %08lx\n",

sizeof (gd\_t), addr\_sp);

这里为 *gd* 预留一块内存空间，同时让临时变量 *id* 指向现在 *addr\_sp* 所在地址

#ifdef CONFIG\_USE\_IRQ

addr\_sp -= (CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ);

debug("Reserving %zu Bytes for IRQ stack at: %08lx\n",

CONFIG\_STACKSIZE\_IRQ+CONFIG\_STACKSIZE\_FIQ, addr\_sp);

#endif

这里为 *IRQ* 和 *FIQ* 预留一块内存空间

gd->bd->bi\_baudrate = gd->baudrate;

/\* Ram ist board specific, so move it to board code ... \*/

dram\_init\_banksize();

display\_dram\_config(); /\* and display it \*/

gd->relocaddr = addr;

gd->start\_addr\_sp = addr\_sp;

gd->reloc\_off = addr - (ulong)&\_start;

debug("relocation Offset is: %08lx\n", gd->reloc\_off);

if (new\_fdt) {

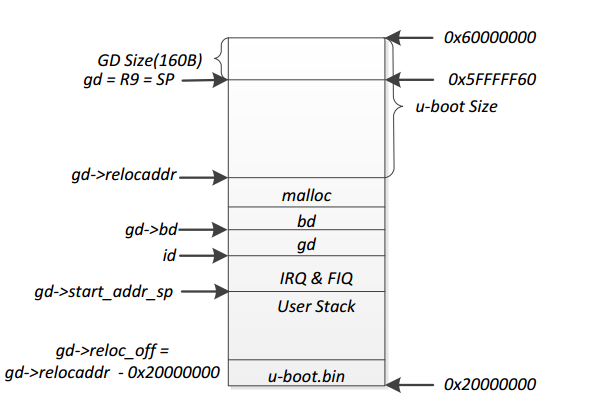
memcpy(new\_fdt, gd->fdt\_blob, fdt\_size);

gd->fdt\_blob = new\_fdt;

}

memcpy(id, (void \*)gd, sizeof(gd\_t));

将一些关键变量的值保持到全局变量 *gd* 中，这里的*\_start* 的地址为 *0x20000000*，因为在前面分析中，我们通过宏 *CONFIG\_SYS\_TEXT\_BASE* 将代码段的基地址设置为 *0x20000000*， *addr - (ulong)&\_start* 刚好得到重定位地址相对 *u-boot* 当前所处的地址（*0x20000000*） 的偏移地址。 然后将全局变量 *gd* 的内容复制到 *id* 所指向的那块内存，后面的代码会将 *gd* 指向这块内存。 现在的内存布局如下图所示：



然后返回到 *crt0.S* 的*\_main* 函数

/\*

\* Set up intermediate environment (new sp and gd) and call

\* relocate\_code(addr\_moni). Trick here is that we'll return

\* 'here' but relocated.

\*/

ldr sp, [r9, #GD\_START\_ADDR\_SP] /\* sp = gd->start\_addr\_sp \*/

bic sp, sp, #7 /\* 8-byte alignment for ABI compliance \*/

ldr r9, [r9, #GD\_BD] /\* r9 = gd->bd \*/

sub r9, r9, #GD\_SIZE /\* new GD is below bd \*/

adr lr, here

ldr r0, [r9, #GD\_RELOC\_OFF] /\* r0 = gd->reloc\_off \*/

add lr, lr, r0

ldr r0, [r9, #GD\_RELOCADDR] /\* r0 = gd->relocaddr \*/

b relocate\_code

here:

*r9* 即 *gd*，其中 *GD\_START\_ADDR\_SP*、 *GD\_BD*、 *GD\_SIZE*、 *GD\_RELOC\_OFF* 和 *GD\_RELOCADDR* 均在*u-boot-2014.04/include/generated/generic-asm-offsets.h* 中定义

#define GENERATED\_GBL\_DATA\_SIZE 160 /\* (sizeof(struct global\_data) + 15) & ~15 @ \*/

#define GENERATED\_BD\_INFO\_SIZE 32 /\* (sizeof(struct bd\_info) + 15) & ~15 @ \*/

#define GD\_SIZE 160 /\* sizeof(struct global\_data) @ \*/

#define GD\_BD 0 /\* offsetof(struct global\_data, bd) @ \*/

#define GD\_RELOCADDR 44 /\* offsetof(struct global\_data, relocaddr) @ \*/

#define GD\_RELOC\_OFF 64 /\* offsetof(struct global\_data, reloc\_off) @ \*/

#define GD\_START\_ADDR\_SP 60 /\* offsetof(struct global\_data, start\_addr\_sp) @ \*/

只要知道这两点，再结合 *u-boot* 的注释，这段代码就很清楚了。 执行这段代码后的内存布局为：



同时将标号 *here*的相对地址赋值给 *lr*，然后将 *lr*减去将要重定位的地址相对 *u-boot*当前地址的偏移，结果是 *lr* 保存了 *u-boot* 重定位后的地址， 调用 *relocate\_code* 重定位完成后，返回跳转到 *lr* 地址执行（现在已经位于重定位后的区域）。

1. **u-boot 重定位**

在 *u-boot* 的帮助文档 *doc/README.arm-relocation* 中对重定位有说明。 *u-boot* 为了生成位置无关码，  
在链接时指定了*-pie* 选项， 这个选项在 *u-boot-2014.04/arch/arm/config.mk* 中指定

# needed for relocation

LDFLAGS\_u-boot += -pie

当使用*-pie* 选项后，链接器会生成一个修正表（*fixup tables*），在最终的二进制文件 *u-boot.bin* 中表  
现为多了 *2* 个段 *.rel.dyn* 和 *.dynsym*， 还需要在链接脚本文件中增加这 *2* 个段， *u-boot.bin* 的链接脚本文件为 *u-boot-2014.04/arch/arm/cpu/u-boot.lds*，有了这 *2* 个段， *u-boot* 可以根据这 *2* 个段的信息将重定位后的代码的链接地址修正为其运行地址，这样 *u-boot* 就可以重定位到任何地址。

在 *crt0.S* 中调用 *relocate\_code* 完成重定位，并进行地址修正，返回后进行清 *BSS* 操作，然后跳转到*board.c* 中的 *board\_init\_r* 函数，进行更进一步的初始化，比如网卡， 然后进入 *main\_loop* 循环。  
重定位后的 *u-boot* 内存布局如下图：



现在执行 *make all* 编译 *u-boot.bin* 和 *u-boot-spl.bin*，最终生成了 *u-boot.bin* 和 *spl/smdkv210-spl.bin*查看一下 *u-boot.bin* 的大小

myroot@myroot:~/TQ210N/u-boot-2014.04$ ll -h u-boot.bin

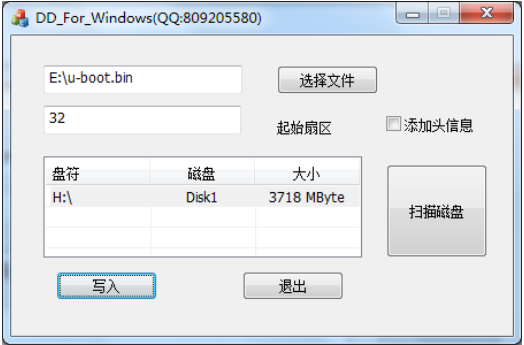
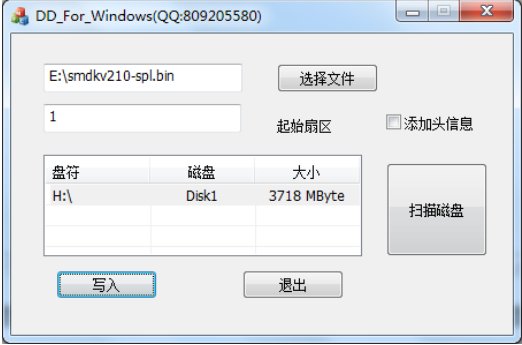
-rw-rw-r-- 1 myroot myroot 197K 12月 18 21:57 u-boot.bin

有 *197K*，换算成块大小为 *197 \* 1024 / 512 = 394* 块， 我们需要修改 *u-boot-spl.bin* 中的 *BL2* 拷贝函数*copy\_bl2\_to\_ram*，其定义在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c*

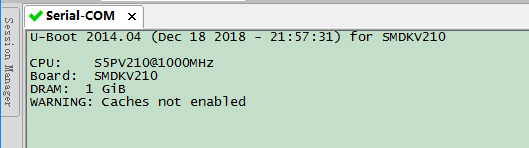
CopySDMMCtoMem(ch, 32, 400, (unsigned int \*)CONFIG\_SYS\_SDRAM\_BASE, 0);

将拷贝大小改为 *400*，拷贝 *400* 块，保存后再次执行 *make all*

使用 *DD\_For\_Windows.exe* 烧写 *smdkv210-spl.bin* 到 *SD* 卡的扇区 *1*，烧写 *u-boot.bin* 到 *SD* 卡扇区 *32*



然后将 *SD* 卡插入开发板，从 *SD* 卡启动。



卡在这里不走了， 通过搜索“*WARNING: Caches not enabled*”，发现是在

*u-boot-2014.04/arch/arm/lib/cache.c* 中的*\_\_enable\_caches* 函数

/\*

\* Default implementation of enable\_caches()

\* Real implementation should be in platform code

\*/

void \_\_enable\_caches(void)

{

puts("WARNING: Caches not enabled\n");

}

void enable\_caches(void)

\_\_attribute\_\_((weak, alias("\_\_enable\_caches")));

这里给*\_\_enable\_caches* 设置了一个别名 *enable\_caches*，当调用 *enable\_caches* 就相当于调用  
*\_\_enable\_caches*， *weak* 表示这是一个弱符号， 当其他地方有相同的函数定义时（普通函数），编译器会忽略掉这个弱符号， 这样可以避免重定义。 因此我们可以自己定义 *enable\_caches* 函数。  
通过跟踪代码发现，在 *board.c* 中的 *board\_init\_r* 函数调用了 *enable\_caches*。现在已经可以使用 *puts*、*printf* 等函数了，我们可以通过打印信息来判断程序卡在哪里了。 一路调试下去，最终发现程序卡在  
*onenand\_init* 里

#if defined(CONFIG\_CMD\_ONENAND)

onenand\_init();

#endif

这个宏 *CONFIG\_CMD\_ONENAND* 在 *smdkv210.h* 中定义，我们的 *TQ210* 开发板没有 *onenand*，所以将这个宏屏蔽掉

#define CONFIG\_CMD\_CACHE

#define CONFIG\_CMD\_REGINFO

/\*masked by zwf\*/

//#define CONFIG\_CMD\_ONENAND

另外，还有一些和单板相关的信息，我们可以修改一下

修改 *smdkv210.h*

#define CONFIG\_SYS\_LONGHELP /\* undef to save memory \*/

#define CONFIG\_SYS\_HUSH\_PARSER /\* use "hush" command parser \*/

#define CONFIG\_SYS\_PROMPT "SMDKV210 # " /\* modied by zwf \*/

这里修改 *u-boot* 命令行下的命令提示符

#define CONFIG\_SYS\_MONITOR\_LEN (256 << 10) /\* 256 KiB \*/

#define CONFIG\_IDENT\_STRING " for SMDKV210" /\* modied by zwf \*/

修改 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中的 *checkboard* 函数

#ifdef CONFIG\_DISPLAY\_BOARDINFO

int checkboard(void)

{

/\* modied by zwf \*/

printf("Board:\tSMDKV210\n");

return 0;

}

#endif

再次执行 *make all*，出错

myroot@myroot:~/TQ210N/u-boot-2014.04$ ./build-uboot.sh

common/built-in.o: In function `saveenv':

/home/myroot/TQ210N/u-boot-2014.04/common/env\_onenand.c:112: undefined reference to `onenand\_mtd'

common/built-in.o: In function `env\_relocate\_spec':

/home/myroot/TQ210N/u-boot-2014.04/common/env\_onenand.c:64: undefined reference to `onenand\_mtd'

Makefile:919: recipe for target 'u-boot' failed

make: \*\*\* [u-boot] Error 1

与 *onenand* 相关的，我们查看 *u-boot-2014.04/common/Makefile*，看下 *env\_onenand.c* 是怎么被编译进*u-boot.bin* 的

obj-$(CONFIG\_ENV\_IS\_IN\_ONENAND) += env\_onenand.o

是由于定义了 *CONFIG\_ENV\_IS\_IN\_ONENAND* 这个宏， 这个宏在 *smdkv210.h* 中定义， 我们将它屏蔽掉

/\*-----------------------------------------------------------------------

\* Boot configuration

\*/

/\*masked by zwf\*/

//#define CONFIG\_ENV\_IS\_IN\_ONENAND 1

再次编译，还是出错

myroot@myroot:~/TQ210N/u-boot-2014.04$ ./build-uboot.sh

common/cmd\_nvedit.c:51:3: error: #error Define one of CONFIG\_ENV\_IS\_IN\_{EEPROM|FLASH|DATAFLASH|ONENAND|SPI\_FLASH|NVRAM|MMC|FAT|REMOTE|UBI} or CONFIG\_ENV\_IS\_NOWHERE

scripts/Makefile.build:335: recipe for target 'common/cmd\_nvedit.o' failed

make[1]: \*\*\* [common/cmd\_nvedit.o] Error 1

Makefile:940: recipe for target 'common' failed

make: \*\*\* [common] Error 2

我们查看 *u-boot-2014.04/common/cmd\_nvedit.c*

#if !defined(CONFIG\_ENV\_IS\_IN\_EEPROM) && \

!defined(CONFIG\_ENV\_IS\_IN\_FLASH) && \

!defined(CONFIG\_ENV\_IS\_IN\_DATAFLASH) && \

!defined(CONFIG\_ENV\_IS\_IN\_MMC) && \

!defined(CONFIG\_ENV\_IS\_IN\_FAT) && \

!defined(CONFIG\_ENV\_IS\_IN\_NAND) && \

!defined(CONFIG\_ENV\_IS\_IN\_NVRAM) && \

!defined(CONFIG\_ENV\_IS\_IN\_ONENAND) && \

!defined(CONFIG\_ENV\_IS\_IN\_SPI\_FLASH) && \

!defined(CONFIG\_ENV\_IS\_IN\_REMOTE) && \

!defined(CONFIG\_ENV\_IS\_IN\_UBI) && \

!defined(CONFIG\_ENV\_IS\_NOWHERE)

# error Define one of CONFIG\_ENV\_IS\_IN\_{EEPROM|FLASH|DATAFLASH|ONENAND|\

SPI\_FLASH|NVRAM|MMC|FAT|REMOTE|UBI} or CONFIG\_ENV\_IS\_NOWHERE

#endif

说明必须指定环境变量保存位置，我们可以将环境变量保存到 *SD* 卡或 *NAND*，但是现在还没移植好，所以先暂时将 *CONFIG\_CMD\_ONENAND* 和 *CONFIG\_ENV\_IS\_IN\_ONENAND* 这 *2* 个宏打开，然后在 *board.c* 中屏蔽 *onenand* 初始化代码。

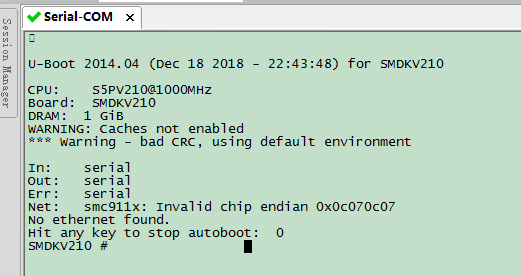
#if defined(CONFIG\_CMD\_ONENAND)

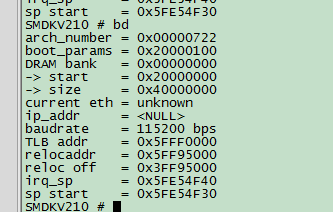
/\*masked by zwf\*/

//onenand\_init();

#endif

再次编译，成功。  
由于这次的修改只针对 *u-boot.bin*，对 *u-boot-spl.bin* 没有影响，因此可以只烧写 *u-boot.bin* 到 *SD* 卡的扇区 *32*。





通过执行 *bd* 可以查看单板信息，实际上是执行 *bdinfo*， *u-boot* 的命令可以缩写（只取前面几个字母），  
只要不和其它命令发送混淆， *u-boot* 都能解析出来。

1. **u-boot 启动流程总结**

本总结基于本文前面移植的方式，因为 *u-boot* 的启动方式不只这一种。  
首先我们移植了 *u-boot-spl.bin* 和 *u-boot.bin*。  
*1) S5PV210* 上电运行 *iROM* 里的代码（*BL0*），执行一系列初始化，然后从 *SD* 卡的扇区 *1* 拷贝  
*u-boot-spl.bin*（*BL1*）到 *iRAM*，然后跳转到 *u-boot-spl.bin* 的起始地址 *0xD0020010* 执行 *u-boot-spl.bin  
2) u-boot-spl.bin* 初始化时钟、 *DDR*，然后从 *SD* 卡的扇区 *32* 拷贝 *u-boot.bin*（*BL2*）到 *DDR* 的起始  
地址 *0x20000000*，然后跳转到 *0x20000000* 执行 *u-boot.bin  
3) u-boot.bin* 执行一系列初始化（定时器、串口等），然后从 *DDR* 的最高地址 *0x60000000* 开始计算  
需要重定位的地址和用户栈指针地址（*SP*），然后把自己（*u-boot.bin*）拷贝到前面计算出的重定  
位地址，再然后根据修正表（*.rel.dyn* 和 *.dynsym*）修正重定位后的 *u-boot.bin* 的链接地址为其运  
行地址，最后跳转到重定位后的 *u-boot.bin* 去执行。

1. **支持命令补全和历史命令**  
   我们知道 Linux 的 shell 支持按 Tab 键自动补全命令，以及历史命令。使用非常方便，我们也想让 u-boot支持这些功能，怎么办？  
   最简单的办法就是百度或者 Google，不过 Google 现在打不开了，就使用百度吧



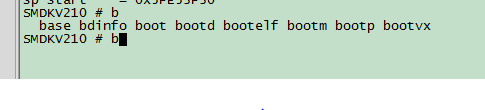
“让 *u-boot shell* 支持 *tab*、命令历史*-jdzwl-ChinaUnix* 博客”一文有说明，需要配置 *2* 个宏  
*CONFIG\_CMDLINE\_EDITING* 和 *CONFIG\_AUTO\_COMPLETE*，我们在 *smdkv210.h* 中增加这 *2* 个宏

#include <config\_cmd\_default.h>

#define CONFIG\_CMDLINE\_EDITING /\*add by zwf\*/

#define CONFIG\_AUTO\_COMPLETE /\*add by zwf\*/

执行 *make all* 编译成功，把 *u-boot.bin* 下载到 *SD* 卡扇区 *32*，然后将 *SD* 卡插入开发板，从 *SD* 卡启动



输入b，按tab键会提示需要输入的命令，如图：

另外，在 *u-boot* 的帮助文档 *README* 中对历史命令和 *Tab* 补全都有说明

- Command Interpreter:

CONFIG\_AUTO\_COMPLETE

Enable auto completion of commands using TAB.

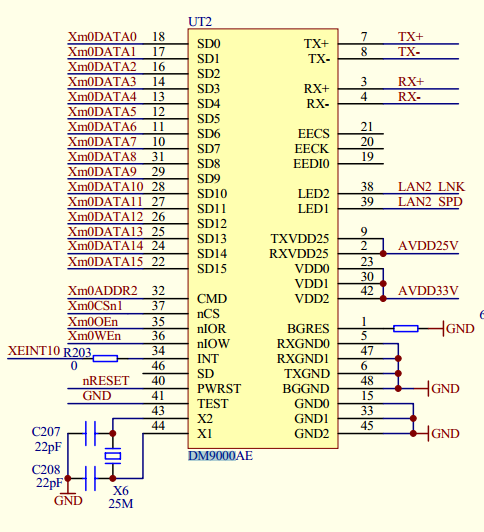
- Commandline Editing and History:

CONFIG\_CMDLINE\_EDITING

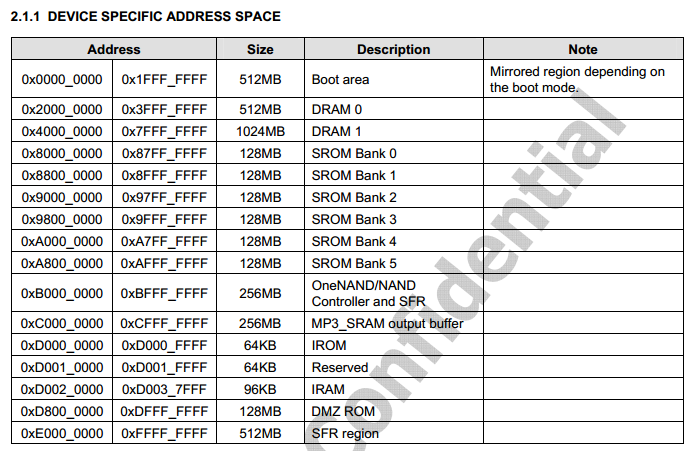
Enable editing and History functions for interactive

commandline input operations

1. **移植网卡** **DM9000**



*TQ210* 板载的网卡芯片是 *DM9000A*，接在 *S5PV210* 的 *SROM* 控制器的 *BANK1* 上。*S5PV210* 的 *SROM* 控制器支持 *8/16* 位 *NOR Flash/PROM/SRAM* 内存。 分为 *6* 个 *bank*，每个 *bank* 寻址空间达 *128MB*。 每个 *bank* 有一个片选信号 *nGCS[5:0]*， 用来选通外接的内存芯片。 当发的地址在*bank1* 的寻址范围类（*0x88000000~0x8FFFFFFF*）时，表示在访问 *bank1*， 此时 *nGCS1* 信号被拉低，这就选中了接在 *bank1* 上的 *DM9000A* 网卡芯片。 *S5PV210* 手册给出了每个 *bank* 的地址空间.



当对某个地址进行读操作时， *S5PV210* 的 *Xm0OEn* 信号被自动拉低，这个信号引脚接在 *DM9000A* 的  
*nIOR* 上， 这就对 *DM9000A* 进行了读使能，可以从 *DM9000A* 读数据； 当对某个地址进行写操作时，  
*S5PV210* 的 *Xm0WEn* 信号被自动拉低，这个信号引脚接在 *DM9000A* 的 *nIOW* 上，这就对 *DM9000A*进行了写使能，可以向 *DM9000A* 写数据。 *DM9000A* 的 *16* 根数据线全部接在 *S5PV210* 的数据线上。

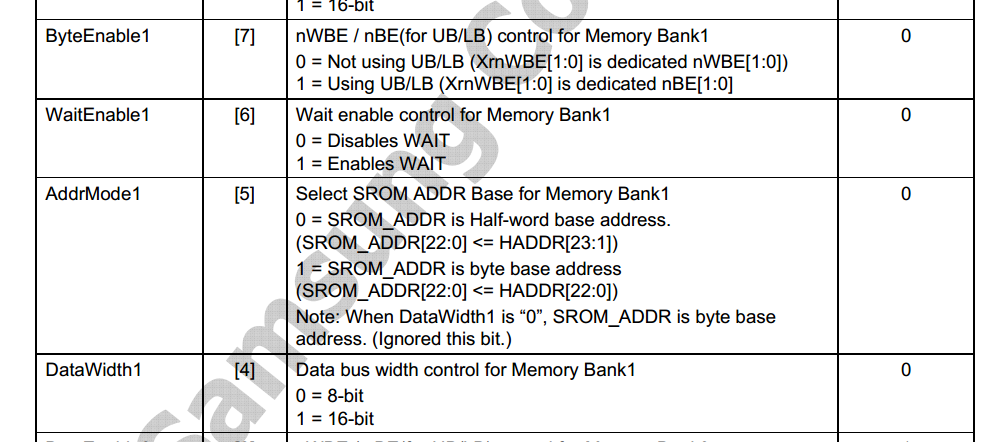
*DM9000A* 包含一系列可被访问的控制状态寄存器，这些寄存器是字节对齐的，他们在硬件或软件复  
位时被设置成初始值。 *DM9000A* 有 *2* 个端口： *DATA* 和 *INDEX*（即地址）。 *DM9000A* 的地址和数据线复用，当 *CMD* 引脚为低电平时，操作的是 *INDEX* 端口，当 *CMD* 引脚为高电平时操作的是 *DATA* 端口。*CMD* 引脚接在 *S5PV210* 的地址线 *Xm0ADDR2* 上。  
假设要读取 *DM9000A* 的寄存器 *RSR*（*RX Status Register*），需要分 *2* 步：

1. 向 *INDEX* 端口写入 *RSR* 寄存器的地址（*0x06*）  
   条件： *nGCS1* 信号拉低、 *Xm0WEn* 信号拉低、 *Xm0ADDR2* 拉低， 或者说向下面的地址写数据 *0x06  
   (0x88000000~0x8FFFFFFF*） *& ~(1 << 2)  
   2)* 从 *DATA* 端口读取 *RSR* 寄存器的值  
   条件： *nGCS1* 信号拉低、 *Xm0OEn* 信号拉低、 *Xm0ADDR2* 拉高， 或者说从下面的地址读数据  
   *(0x88000000~0x8FFFFFFF*） *| (1 << 2)*

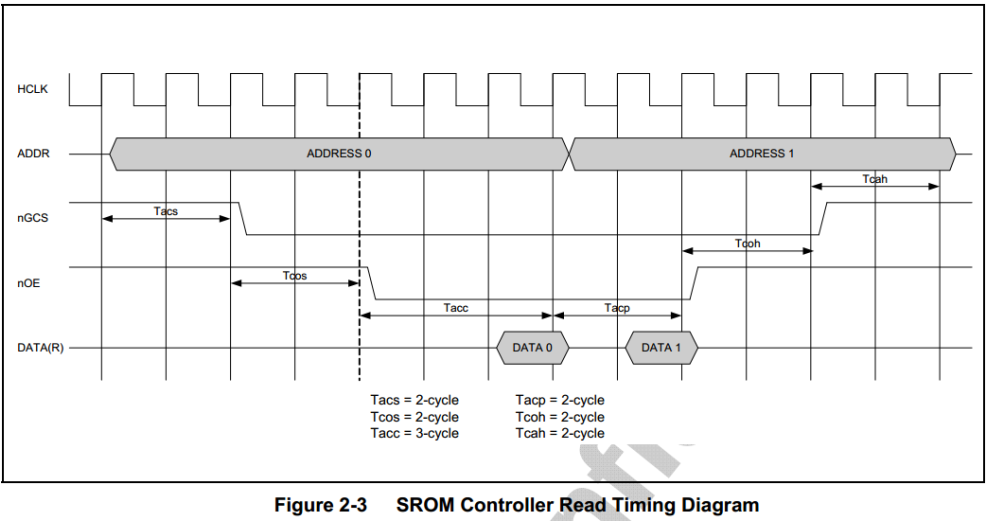
假设要向 *DM9000A* 的寄存器 *TCR*（*TX Control Register*）写数据 *0x56*，同样需要 *2* 步  
*1)* 向 *INDEX* 端口写入 *TCR* 寄存器的地址（*0x02*）  
条件： *nGCS1* 信号拉低、 *Xm0WEn* 信号拉低、 *Xm0ADDR2* 拉低， 或者说向下面的地址写数据 *0x02  
(0x88000000~0x8FFFFFFF*） *& ~(1 << 2)*

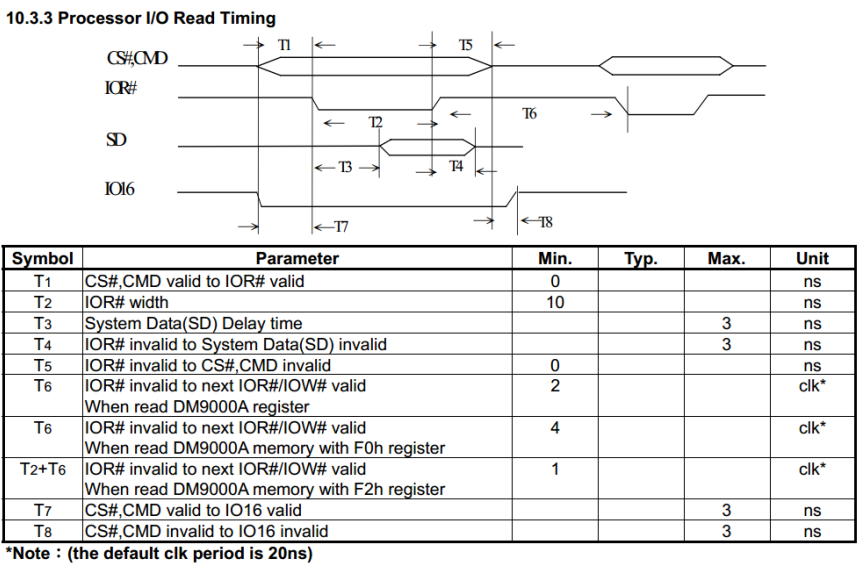
1. 将要写入的数据（*0x56*） 写入 *DATA* 端口  
   条件： *nGCS1* 信号拉低、 *Xm0WEn* 信号拉低、 *Xm0ADDR2* 拉高，或者说向下面的地址写数据 *0x56  
   (0x88000000~0x8FFFFFFF*） *| (1 << 2)*有了这些理论知识就可以确定 *DM9000A* 的基地址了，在后面会用到。

下面开始移植  
首先我们要初始化 *S5PV210* 的 *SROM* 控制器，设置相关时序。 针对 *SROM* 控制器的每一个 *bank* 只有*2* 个寄存器： *SROM\_BW* 和 *SROM\_BC*，我要配置的是 *bank1*，所以只需配置 *SROM\_BW* 和 *SROM\_BC1*在 *SROM\_BW* 寄存器中，我们只关心与 *bank1* 相关的域



上面分析过， *DM9000A* 的 *16* 根数据线全部接在 *S5PV210* 的数据线上，所以 *DataWidth1* 设置为 *1。  
DM9000A* 的地址是按字节存取的，所以 *AddrMode1* 设置为 *1*，通过查看原理图，没有使用 *Xm0WAITn*和 *Xm0BEn* 引脚，所以 *WaitEnable1* 和 *ByteEnable1* 均设置为 *0  
SROM\_BW[7:4]=0x3*下面分析时序，设置 *SROM\_BC1* 寄存器。





DM9000A读时序

上面 *2* 个时序图分别为 *S5PV210* 的 *SROM* 控制器读时序和 *DM9000A* 的读时序  
*Tacs*：地址发出后等多长时间发片选， *DM9000A* 中 *CS* 和 *CMD*（地址）同时发出，所以 *Tacs=0ns  
Tcos*：发出片选信号后等多长时间发出读使能信号（*nOW*、 *IOR*），在 *DM9000A* 的时序图上对应 *T1*，  
最小为 *0*，我们稍微设置大点 *Tcos=5ns  
Tacc*：读使能信号持续时间，在 *DM9000A* 的时序图上对应 *T2*，我们设置为设置 *Tacc=15ns  
Tcoh*：读使能信号结束后，片选信号保持时间，在 *DM9000A* 的时序图中对应 *T5*，所以可以设置  
*Tcoh=5ns  
Tcah*：片选结束后，地址保存时间， *DM9000A* 中片选和地址同时结束，所以 *Tcah=0  
Tacp*：页模式，不管  
*PMC*：页模式，不管

*S5PV210* 的 *SROM* 控制器使用 *MSYS* 域提供 *HCLK* 时钟，为 *200MHz*，一个 *clock* 为 *5ns*。

*u-boot* 已经自带了 *DM9000A* 的驱动，另外，在 *u-boot* 源码中的 *dm9000x.c* 的注释中有一段话

06/03/2008 Remy Bohmer <linux@bohmer.net>

- Fixed the driver to work with DM9000A.

专门为 *DM9000A* 升级了该驱动。  
通过跟踪代码进入到 *board.c* 中的 *board\_init\_r* 函数

#if defined(CONFIG\_CMD\_NET)

puts("Net: ");

eth\_initialize(gd->bd);

#if defined(CONFIG\_RESET\_PHY\_R)

debug("Reset Ethernet PHY\n");

reset\_phy();

#endif

#endif

如果定义了 *CONFIG\_CMD\_NET*，就调用 *eth\_initialize(gd->bd)*进行网卡初始化。这个宏在  
*config\_cmd\_default.h* 中定义，这个头文件又被单板配置文件 *smdkv210.h* 所包含。

/\*

\* Ethernet Contoller driver

\*/

#ifdef CONFIG\_CMD\_NET

#define CONFIG\_SMC911X 1 /\* we have a SMC9115 on-board \*/

#define CONFIG\_SMC911X\_16\_BIT 1 /\* SMC911X\_16\_BIT Mode \*/

#define CONFIG\_SMC911X\_BASE 0x98800300 /\* SMC911X Drive Base \*/

#define CONFIG\_ENV\_SROM\_BANK 3 /\* Select SROM Bank-3 for Ethernet\*/

#endif /\* CONFIG\_CMD\_NET \*/

*eth\_initialize* 函数在 *u-boot-2014.04/net/eth.c* 中定义

/\*

\* If board-specific initialization exists, call it.

\* If not, call a CPU-specific one

\*/

if (board\_eth\_init != \_\_def\_eth\_init) {

if (board\_eth\_init(bis) < 0)

printf("Board Net Initialization Failed\n");

} else if (cpu\_eth\_init != \_\_def\_eth\_init) {

if (cpu\_eth\_init(bis) < 0)

printf("CPU Net Initialization Failed\n");

} else

printf("Net Initialization Skipped\n");

这里已经有注释，如果定义了单板相关的初始化函数就调用它，否则调用 *CPU* 相关的初始化函数。  
我们继续看下*\_\_def\_eth\_init* 函数，同样在 *u-boot-2014.04/net/eth.c* 中定义

/\*

\* CPU and board-specific Ethernet initializations. Aliased function

\* signals caller to move on

\*/

static int \_\_def\_eth\_init(bd\_t \*bis)

{

return -1;

}

int cpu\_eth\_init(bd\_t \*bis) \_\_attribute\_\_((weak, alias("\_\_def\_eth\_init")));

int board\_eth\_init(bd\_t \*bis) \_\_attribute\_\_((weak, alias("\_\_def\_eth\_init")));

这里用到了 *gcc* 的弱符号和别名属性， 前面分析过。 如果我们没有定义自己的 *board\_eth\_init* 函数，  
则 *board\_eth\_init* 就和*\_\_def\_eth\_init* 相同，调用 *board\_eth\_init* 就相当于调用*\_\_def\_eth\_init*，现在就  
能明白上面的 *if* 判断语句了。

*board\_eth\_init* 在 *u-boot-2014.04/board/samsung/smdkv210/smdkv210.c* 中定义

nt board\_eth\_init(bd\_t \*bis)

{

int rc = 0;

#ifdef CONFIG\_SMC911X

rc = smc911x\_initialize(0, CONFIG\_SMC911X\_BASE);

#endif

return rc;

}

这里通过配置宏来决定调用哪个网卡初始化函数，同时在 *smdkv210.c* 中还有个函数 *smc9115\_pre\_init*这个函数用来设置 *SROM* 控制器。  
我们使用的是 *DM9000A*，我们先查看下 *DM9000A* 的源码，看下使用哪个初始化函数  
*u-boot-2014.04/drivers/net/DM9000x.c*

int dm9000\_initialize(bd\_t \*bis)

{

struct eth\_device \*dev = &(dm9000\_info.netdev);

/\* Load MAC address from EEPROM \*/

dm9000\_get\_enetaddr(dev);

dev->init = dm9000\_init;

dev->halt = dm9000\_halt;

dev->send = dm9000\_send;

dev->recv = dm9000\_rx;

sprintf(dev->name, "dm9000");

eth\_register(dev);

return 0;

}

看到这个函数比较像，并且它需要一个 *bd\_t \**类型参数，而 *board\_eth\_init* 也传进来一个 *bd\_t \**的参  
数，因此就调用这个 *DM9000A* 的初始化函数，在这个函数中 *dm9000\_get\_enetaddr* 从 *EEPROM* 加载  
*MAC*地址，这个函数根据宏 *CONFIG\_DM9000\_NO\_SROM* 来决定是否从 *EEPROM* 加载 *MAC*地址，*TQ210*上的 *DM9000A* 没有接 *EEPROM*，我们将在 *smdkv210.h* 中定义这个宏，表示不从 *EEPROM* 加载 *MAC*地址。 另外，我们还需要知道如何才能把 *DM9000A* 的驱动编译进 *u-boo.bin*，我们查看  
*u-boot-2014.04/drivers/net/Makefile*

obj-$(CONFIG\_DRIVER\_DM9000) += dm9000x.o

从 *Makefile* 得知，需要定义 *CONFIG\_DRIVER\_DM9000* 这个宏。

修改 *board\_eth\_init* 函数

int board\_eth\_init(bd\_t \*bis)

{

int rc = 0;

#ifdef CONFIG\_SMC911X

rc = smc911x\_initialize(0, CONFIG\_SMC911X\_BASE);

/\* add by zwf \*/

#elif defined(CONFIG\_DRIVER\_DM9000)

rc = dm9000\_initialize(bis);

#endif

return rc;

}

同时我们看到在 *smdkv210.c* 中的 *board\_init* 函数还调用了 *smc9115* 的初始化，我们不需要，把它屏  
蔽掉，然后仿照实现 *dm9000\_pre\_init* 函数，初始化 *SROM*

/\*

\* Miscellaneous platform dependent initialisations

\*/

/\* masked by zwf \*/

#if 0

static void smc9115\_pre\_init(void)

{

u32 smc\_bw\_conf, smc\_bc\_conf;

struct s5pc100\_gpio \*const gpio =

(struct s5pc100\_gpio \*)samsung\_get\_base\_gpio();

/\* gpio configuration GPK0CON \*/

s5p\_gpio\_cfg\_pin(&gpio->k0, CONFIG\_ENV\_SROM\_BANK, GPIO\_FUNC(2));

/\* Ethernet needs bus width of 16 bits \*/

smc\_bw\_conf = SMC\_DATA16\_WIDTH(CONFIG\_ENV\_SROM\_BANK);

smc\_bc\_conf = SMC\_BC\_TACS(0x0) | SMC\_BC\_TCOS(0x4) | SMC\_BC\_TACC(0xe)

| SMC\_BC\_TCOH(0x1) | SMC\_BC\_TAH(0x4)

| SMC\_BC\_TACP(0x6) | SMC\_BC\_PMC(0x0);

/\* Select and configure the SROMC bank \*/

s5p\_config\_sromc(CONFIG\_ENV\_SROM\_BANK, smc\_bw\_conf, smc\_bc\_conf);

}

#endif

/\* add by zwf \*/

static void dm9000\_pre\_init(void)

{

u32 smc\_bw\_conf, smc\_bc\_conf;

/\* Ethernet needs bus width of 16 bits \*/

smc\_bw\_conf = SMC\_DATA16\_WIDTH(CONFIG\_ENV\_SROM\_BANK)

| SMC\_BYTE\_ADDR\_MODE(CONFIG\_ENV\_SROM\_BANK);

smc\_bc\_conf = SMC\_BC\_TACS(0) | SMC\_BC\_TCOS(1) | SMC\_BC\_TACC(2)

| SMC\_BC\_TCOH(1) | SMC\_BC\_TAH(0) | SMC\_BC\_TACP(0) | SMC\_BC\_PMC(0);

/\* Select and configure the SROMC bank \*/

s5p\_config\_sromc(CONFIG\_ENV\_SROM\_BANK, smc\_bw\_conf, smc\_bc\_conf);

}

int board\_init(void)

{

/\* masked by zwf \*/

//smc9115\_pre\_init();

dm9000\_pre\_init();

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDKC100;

gd->bd->bi\_boot\_params = PHYS\_SDRAM\_1 + 0x100;

return 0;

}

然后在配置文件 *smdkv210.h* 中添加宏 *CONFIG\_DRIVER\_DM9000A*，同时屏蔽掉 *smc9111* 的相关宏

/\* masked by zwf \*/

#if 0

#define CONFIG\_SMC911X 1 /\* we have a SMC9115 on-board \*/

#define CONFIG\_SMC911X\_16\_BIT 1 /\* SMC911X\_16\_BIT Mode \*/

#define CONFIG\_SMC911X\_BASE 0x98800300 /\* SMC911X Drive Base \*/

#endif

/\* modied by zwf \*/

#define CONFIG\_ENV\_SROM\_BANK 1 /\* Select SROM Bank-1 for Ethernet\*/

在*smdkv210.h* 定义 *CONFIG\_DM9000A\_BASE*、 *DM9000A\_DATA* 和 *DM9000A\_IO  
CONFIG\_DM9000A\_BASE* 为 *DM9000A* 的基地址  
*DM9000A\_DATA* 为 *DM9000A* 的 *DATA* 端口地址  
*DM9000A\_IO* 为 *DM9000A* 的 *INDEX* 端口地址  
#define CONFIG\_DM9000\_BASE 0x88000000

#define DM9000\_IO (CONFIG\_DM9000\_BASE)

#define DM9000\_DATA (CONFIG\_DM9000\_BASE + 0x4)

这里的 *DM9000A\_DATA* 定义为基地址*+0x4*，刚好把 *Xm0ADDR2* 拉高，即把 *CMD* 拉高。  
网上很多文章说什么基地址要加 *0x300*，全是无稽之谈，还解释得头头是道。

编译，成功生成 *u-boot.bin*，将 *u-boot.bin* 烧尽 *SD* 卡，从 *SD* 卡启动开发板

在前面分析过如何确定这些地址。另外，我们还需要添加一些 *u-boot* 的命令，比如 *ping*、 *tftpboot*。  
*ping* 命令用来检查网络是否通畅， *tftpboot* 用来从 *tftp* 服务器下载文件。

如何知道 *u-boot* 支持哪些命令，查看头文件： *u-boot-2014.04/include/config\_cmd\_all.h*

#ifndef \_CONFIG\_CMD\_ALL\_H

#define \_CONFIG\_CMD\_ALL\_H

/\*

\* Alphabetical list of all possible commands.

\*/

#define CONFIG\_CMD\_AMBAPP /\* AMBA Plug & Play Bus print utility \*/

#define CONFIG\_CMD\_ASKENV /\* ask for env variable \*/

该头文件的注释也说明， 该文件列出了所有可用的命令。

#define CONFIG\_CMD\_NET /\* bootp, tftpboot, rarpboot \*/

#define CONFIG\_CMD\_PING /\* ping support \*/

另外在 *smdkv210.h* 中包含了头文件 *config\_cmd\_default.h*，里面定义了一些命令，这是 *u-boot* 默认  
配置的命令，我们看下里面有没有我们需要配置的，如果没有，我们就需要自己配置。

#define CONFIG\_CMD\_BDI /\* bdinfo \*/

#define CONFIG\_CMD\_BOOTD /\* bootd \*/

#define CONFIG\_CMD\_CONSOLE /\* coninfo \*/

#define CONFIG\_CMD\_ECHO /\* echo arguments \*/

#define CONFIG\_CMD\_EDITENV /\* editenv \*/

#define CONFIG\_CMD\_FPGA /\* FPGA configuration Support \*/

#define CONFIG\_CMD\_IMI /\* iminfo \*/

#define CONFIG\_CMD\_ITEST /\* Integer (and string) test \*/

#ifndef CONFIG\_SYS\_NO\_FLASH

#define CONFIG\_CMD\_FLASH /\* flinfo, erase, protect \*/

#define CONFIG\_CMD\_IMLS /\* List all found images \*/

#endif

#define CONFIG\_CMD\_LOADB /\* loadb \*/

#define CONFIG\_CMD\_LOADS /\* loads \*/

#define CONFIG\_CMD\_MEMORY /\* md mm nm mw cp cmp crc base loop \*/

#define CONFIG\_CMD\_MISC /\* Misc functions like sleep etc\*/

#define CONFIG\_CMD\_NET /\* bootp, tftpboot, rarpboot \*/

#define CONFIG\_CMD\_NFS /\* NFS support \*/

#define CONFIG\_CMD\_RUN /\* run command in env variable \*/

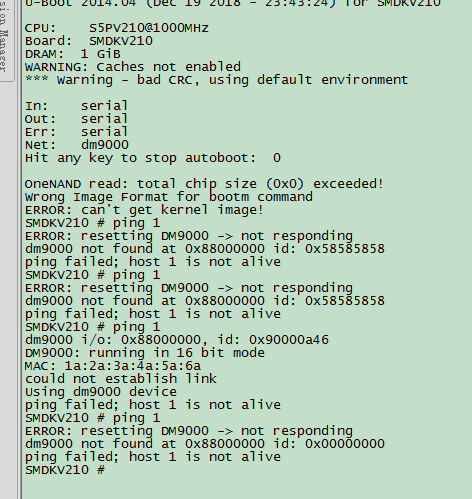
#define CONFIG\_CMD\_SAVEENV /\* saveenv \*/

#define CONFIG\_CMD\_SETGETDCR /\* DCR support on 4xx \*/

#define CONFIG\_CMD\_SOURCE /\* "source" command support \*/

#define CONFIG\_CMD\_XIMG /\* Load part of Multi Image \*/

可以看到 *CONFIG\_CMD\_NET* 已经配置了，所以我们只需要配置 *CONFIG\_CMD\_PING* 命令。



有的时候能读取到 *DM9000A* 的 *ID*，连续操作就能读取到 *DM9000A* 的 *ID*，但间隔一会操作就读取不  
到 *DM9000A* 的 *ID*，通过调试，在 *dm9000x.c* 的*dm9000\_reset* 函数中加一句延时操作，就可以正  
常读取 *DM9000A* 的 *ID* 了。

do {

DM9000\_DBG("resetting the DM9000, 2nd reset\n");

udelay(25); /\* Wait at least 20 us \*/

} while (DM9000\_ior(DM9000\_NCR) & 1);

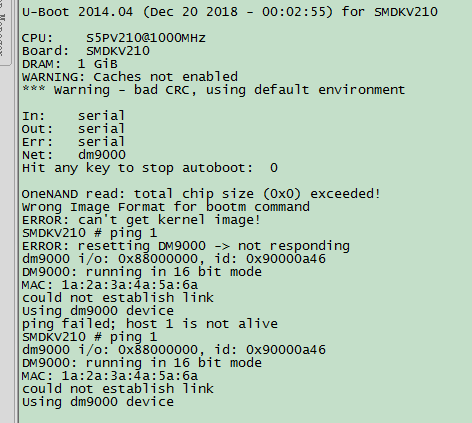
udelay(150);/\*add by zwf\*/

/\* Check whether the ethernet controller is present \*/

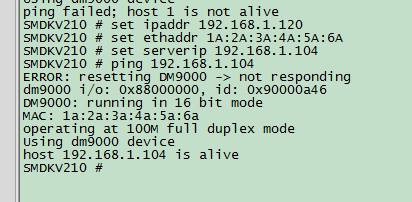
if ((DM9000\_ior(DM9000\_PIDL) != 0x0) ||

(DM9000\_ior(DM9000\_PIDH) != 0x90))

printf("ERROR: resetting DM9000 -> not responding\n");



首先设置环境变量  
*ipaddr*： *u-boot* 的 *IP  
ethaddr*： *u-boot* 的 *MAC* 地址  
*serverip*： *u-boot* 通过 *tftpboot* 从哪里下载文件的 *PC* 机的 *IP*我的电脑的 *IP* 为 *192.168.1.104*，进行如下设置



可以 *ping* 通

下面使用 *tftpboot* 下载文件到内存，首先打开 *tftp* 服务器软件 *tftpd32.exe*

**配置ubuntu环境下的tftp：**

*1.tftp服务器安装*

*sudo apt-get install tftpd-hpa（是服务器端）*

*sudo apt-get install tftp-hpa（如果不需要客户端可以不安装）*

*2.配置TFTP服务器*

*sudo vim /etc/default/tftpd-hpa*

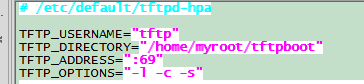
*将原来的内容改为:*

*TFTP\_USERNAME="tftp"*

*TFTP\_ADDRESS="0.0.0.0:69"*

*TFTP\_DIRECTORY="tftp根目录" #服务器目录,需要设置权限为777,chomd 777*

*TFTP\_OPTIONS="-l -c -s"*



*3.重新启动TFTP服务*

*sudo service tftpd-hpa restart*

*4.关闭安全相关的权限*

*$sudo ufw disable //关闭防火墙*

*SELinux 宽容模式(permissive) 强制模式(enforcing) 关闭(disabled) 几种模式之间的转换*

*$setenforce permissive //SELinux 宽容模式(permissive)，否则tftp不可用*

*可能需要先安装selinux-utils*

*$sudo apt install selinux-utils*

1. 移植 **NAND FLASH**

通过查看帮助文档 *doc/README.nand*，要支持 *NAND*，需要配置 *CONFIG\_CMD\_NAND*我们修改 *smdkv210.h*，把 *CONFIG\_CMD\_ONENAND* 屏蔽掉，同时加上 *CONFIG\_CMD\_NAND*

#if 0

#define CONFIG\_CMD\_ONENAND

#endif

/\*add by zwf\*/

#define CONFIG\_CMD\_NAND

编译出错

myroot@myroot:~/TQ210N/u-boot-2014.04$ ./build-uboot.sh

CC common/cmd\_nand.o

common/cmd\_nand.c: In function 'set\_dev':

common/cmd\_nand.c:116: error: 'CONFIG\_SYS\_MAX\_NAND\_DEVICE' undeclared (first use in this function)

common/cmd\_nand.c:116: error: (Each undeclared identifier is reported only once

common/cmd\_nand.c:116: error: for each function it appears in.)

common/cmd\_nand.c: In function 'do\_nand':

common/cmd\_nand.c:497: error: 'CONFIG\_SYS\_MAX\_NAND\_DEVICE' undeclared (first use in this function)

common/cmd\_nand.c: In function 'do\_nandboot':

common/cmd\_nand.c:1053: error: 'CONFIG\_SYS\_MAX\_NAND\_DEVICE' undeclared (first use in this function)

scripts/Makefile.build:335: recipe for target 'common/cmd\_nand.o' failed

make[1]: \*\*\* [common/cmd\_nand.o] Error 1

Makefile:940: recipe for target 'common' failed

make: \*\*\* [common] Error 2

没有定义 *CONFIG\_SYS\_MAX\_NAND\_DEVICE*，最大 *NAND* 设备数，我们的板子只有 *1* 个 *NAND*，将其  
定义为 *1,在smdkv210.h中定义*

/\*add by zwf\*/

#define CONFIG\_SYS\_MAX\_NAND\_DEVICE 1

再次编译，出错

myroot@myroot:~/TQ210N/u-boot-2014.04$ ./build-uboot.sh

LD drivers/mtd/built-in.o

CC drivers/mtd/nand/nand.o

drivers/mtd/nand/nand.c:41: error: 'CONFIG\_SYS\_NAND\_BASE' undeclared here (not in a function)

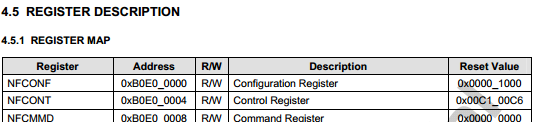
scripts/Makefile.build:335: recipe for target 'drivers/mtd/nand/nand.o' failed

make[1]: \*\*\* [drivers/mtd/nand/nand.o] Error 1

Makefile:940: recipe for target 'drivers/mtd/nand' failed

make: \*\*\* [drivers/mtd/nand] Error 2

未定义 *NAND* 的基地址，查看 *S5PV210* 手册， *NAND* 的基地址为 *0xB0E00000*，在 *smdkv210.h* 中定义



再次编译，还是出错

myroot@myroot:~/TQ210N/u-boot-2014.04$ ./build-uboot.sh

LD u-boot

common/built-in.o: In function `saveenv':

/home/myroot/TQ210N/u-boot-2014.04/common/env\_onenand.c:112: undefined reference to `onenand\_mtd'

common/built-in.o: In function `env\_relocate\_spec':

/home/myroot/TQ210N/u-boot-2014.04/common/env\_onenand.c:64: undefined reference to `onenand\_mtd'

drivers/mtd/nand/built-in.o: In function `nand\_init\_chip':

/home/myroot/TQ210N/u-boot-2014.04/drivers/mtd/nand/nand.c:91: undefined reference to `board\_nand\_init'

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9537

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9771

Segmentation fault (core dumped)

Makefile:919: recipe for target 'u-boot' failed

make: \*\*\* [u-boot] Error 139

*env\_onenand.c* 与 *onenand* 相关的，我们不需要编译，查看 *u-boot-2014.04/common/Makefile*

obj-$(CONFIG\_ENV\_IS\_IN\_NAND) += env\_nand.o

obj-$(CONFIG\_ENV\_IS\_IN\_NVRAM) += env\_nvram.o

obj-$(CONFIG\_ENV\_IS\_IN\_ONENAND) += env\_onenand.o

有 *env\_nand.o* 和 *env\_onenand.o*，我们需要把环境变量保存到 *NAND*，因此我们需要在 *smdkv210.h*定义*CONFIG\_ENV\_IS\_IN\_NAND*，同时屏蔽掉 *CONFIG\_ENV\_IS\_IN\_ONENAND*

/\*-----------------------------------------------------------------------

\* Boot configuration

\*/

/\*maske by zwf\*/

#if 0

#define CONFIG\_ENV\_IS\_IN\_ONENAND 1

#endif

/\*add by zwf\*/

#define CONFIG\_ENV\_IS\_IN\_NAND 1

再次编译

LD u-boot

drivers/mtd/nand/built-in.o: In function `nand\_init\_chip':

/home/myroot/TQ210N/u-boot-2014.04/drivers/mtd/nand/nand.c:91: undefined reference to `board\_nand\_init'

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9537

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9771

Segmentation fault (core dumped)

Makefile:919: recipe for target 'u-boot' failed

make: \*\*\* [u-boot] Error 139

没有定义 *board\_nand\_init*， 看名字就知道这是单板相关的 *NAND* 初始化函数。 通过搜索 *u-boot* 源码发现在 *u-boot-2014.04/drivers/mtd/nand/s3c2410\_nand.c* 中定义了这个函数，我们仿照  
*s3c2410\_nand.c* 编写 *s5pv210\_nand.c*，拷贝 *s3c2410\_nand.c* 为 *s5pv210\_nand.c*

myroot@myroot:~/TQ210N/u-boot-2014.04$ cp drivers/mtd/nand/s3c2410\_nand.c drivers/mtd/nand/s5pv210\_nand.c

同时查看 *u-boot-2014.04drivers/mtd/nand/Makefile*，看下怎么将 *s5pv210\_nand.c* 编译进 *u-boot*

obj-$(CONFIG\_NAND\_S3C2410) += s3c2410\_nand.o

只有定义了 *CONFIG\_NAND\_S3C2410* 就会编译 *s3c2410\_nand.c*

我们仿照添加一行

obj-$(CONFIG\_NAND\_S3C2410) += s3c2410\_nand.o

/\* add by zwf \*/

obj-$(CONFIG\_NAND\_S5PV210) += s5pv210\_nand.o

并且在 *smdkv210.h* 中定义 *CONFIG\_NAND\_S5PV210*

/\*add by zwf\*/

#define CONFIG\_CMD\_NAND

/\*add by zwf\*/

#define CONFIG\_SYS\_MAX\_NAND\_DEVICE 1

/\*add by zwf\*/

#define CONFIG\_SYS\_NAND\_BASE 0xB0E00000

/\*add by zwf\*/

#define CONFIG\_NAND\_S5PV210

将 *s5pv210\_nand.c* 中的 *s3c2410* 全部替换为 *s5pv210*，然后我们参考裸机编程中的 *NAND* 一节来修改*S5pv210\_nand.c*首先我们需要用到 *nand*相关的寄存器操作，在 *arch/arm/include/asm/arch-s5pc1xx/cpu.h*中添加 *NAND*的基地址

#define S5PV210\_VIC2\_BASE 0xF2200000

#define S5PV210\_VIC3\_BASE 0xF2300000

#define S5PV210\_NAND\_BASE 0xB0E00000

紧接着在下面添加宏

SAMSUNG\_BASE(nand, NAND\_BASE)

然后在 *arch/arm/include/asm/arch-s5pc1xx/*下创建文件 *nand\_reg.h*，定义 *NAND* 的寄存器结构体

/\* add by zwf\*/

#ifndef \_\_ASM\_ARM\_ARCH\_NAND\_REG\_H\_

#define \_\_ASM\_ARM\_ARCH\_NAND\_REG\_H\_

#ifndef \_\_ASSEMBLY\_\_

struct s5pv210\_nand {

u32 nfconf;

u32 nfcont;

u32 nfcmmd;

然后在 *s5pv210\_nand.c* 中添加头文件  
*#include <asm/arch/nand\_reg.h>*

参考前面裸机编程中 *NAND* 操作，在 *board\_nand\_init* 进行相关的初始化， 增加函数  
*s5pv210\_nand\_select\_chip*，这个片选函数最终将调用 *s5pv210\_hwcontrol*， 修改 *s5pv210\_hwcontrol*函数，这个函数主要做一些硬件相关的操作，比如发命令、发地址、片选。 具体请看代码。  
编译成功， 由于增加了 *NAND* 驱动， *u-boot.bin* 的大小已经变为 *215KB*

myroot@myroot:~/TQ210N/u-boot-2014.04$ ll -h u-boot.bin

-rw-rw-r-- 1 myroot myroot 220K 12月 22 11:29 u-boot.bin

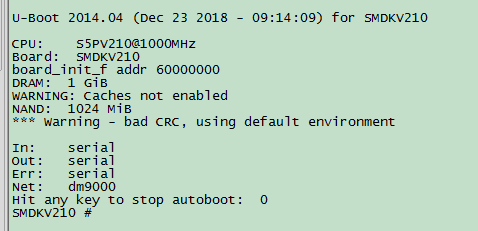
我们需要修改 *u-boot-spl.bin* 中的 *copy\_bl2\_to\_ram* 中的拷贝大小  
*220K / 512 = 220 x 1024 / 512 = 440*我们之间将拷贝大小修改为 *500* ，拷贝 *250KB*。

smdkv210.c (u-boot-2014.04/board/samsung/smdkv210/smdkv210.c)

CopySDMMCtoMem(ch, 32, 500, (unsigned int \*)CONFIG\_SYS\_SDRAM\_BASE, 0);

此处修改涉及到spl

再次编译， 将 *smdkv210-spl.bin* 和 *u-boot.bin* 烧写的 *SD* 卡，从 *SD* 卡启动开发板



打印警告： *Warning - bad CRC, using default environment*这是由于我们在 *smdkv210.h* 中指定了将环境变量保存到 *NAND*， *u-boot* 启动的时候会从指定的地址  
读取环境变量

board.c (u-boot-2014.04/arch/arm/lib/board.c)

/\* initialize environment \*/

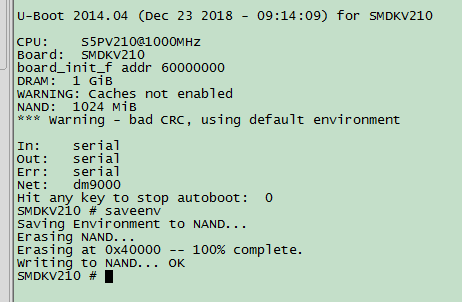
if (should\_load\_env())

env\_relocate();

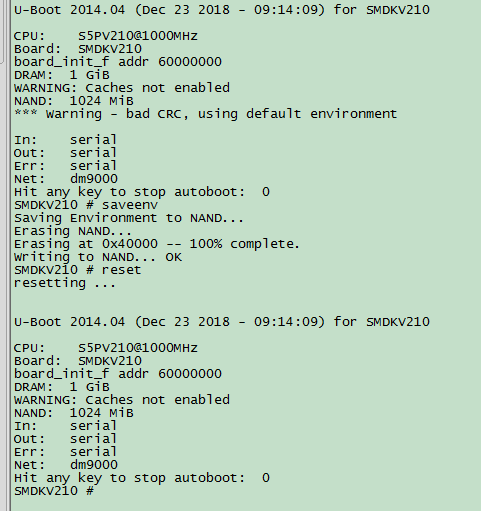
else

set\_default\_env(NULL);

如果读取失败，则使用默认的环境变量，默认的环境变量就是我们通过 *CONFIG\_*这样的宏在单板配  
置文件中的定义的。 比如之前定义的 *CONFIG\_IPADDR*。我们执行 *saveenv* 将环境变量保存到 *NAND* 中，下次启动就不会有那样的警告了

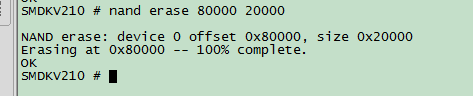


执行 *reset* 复位

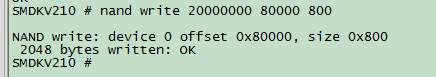


现在来测试一下 *NAND* 读写是否正确无误  
我们先通过 *tftp* 下载一个文件到内存

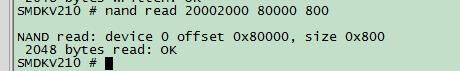
然后擦除 *NAND* 的 *0x80000* 地址开始的一块



然后将内存 *0x20000000* 开始的 *0x800* 字节的数据写入 *NAND* 的 *0x80000* 地址



然后从 *NAND* 的 *0x80000* 地址读取 *0x800* 字节数据到内存的 *0x20002000* 地址

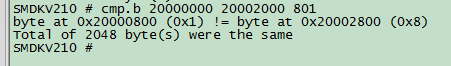


然后使用 *cmp* 命令，比较 *0x20000000* 和 *0x20002000* 开始的 *0x800* 字节数据是否相同



*Total of 2048 byte(s) were the same*我们读写了 *0x800* 个字节，即 *2048* 个字节，全部正确无误。

验证不相等。



**13.给 NAND 添加分区**

这里的分区只是为了操作的方便。 假设有个分区 *kernel*，其地址区间为 *0x20000~0x320000*，其大小  
为 *0x300000*在没有分区的情况下，对这块区间的操作如下：  
*nand erase 20000 300000* 擦除操作  
*nand write 20000000 20000 300000* 写操作  
*nand read 20000000 20000 300000* 读操作  
而有了分区的情况下，操作将变得非常简单，如下（其中 *kernel* 为分区名）  
*nand erase.part kernel* 擦除操作

|  |  |
| --- | --- |
| *nand write 20000000 kerne* | 写操作 *l* |
| *nand read 20000000 kernel* | 读操作 |

在 *smdkv210.h* 中，已经有 *NAND* 分区的例子，我们直接修改

|  |  |
| --- | --- |
| 修改前：  #define MTDIDS\_DEFAULT "onenand0=s3c-onenand"  #define MTDPARTS\_DEFAULT "mtdparts=s3c-onenand:256k(bootloader)"\  ",128k@0x40000(params)"\  ",3m@0x60000(kernel)"\  ",16m@0x360000(test)"\  ",-(UBI)" | 修改后：  #define MTDIDS\_DEFAULT "nand0=s5p-nand"  #define MTDPARTS\_DEFAULT "mtdparts=s5p-nand:256k(bootloader)"\  ",128k@0x40000(params)"\  ",3m@0x60000(kernel)"\  ",-(rootfs)" |

*256K* 给 *u-boot  
128K* 给 *params*，即环境变量  
*3M* 给内核  
剩余的全部给根文件系统  
这里指定了环境变量在 *NAND* 的分区所在的地址空间，在 *smdkv210.h* 中定义了其所在的地址和大小

/\*add by zwf\*/

#define CONFIG\_ENV\_IS\_IN\_NAND

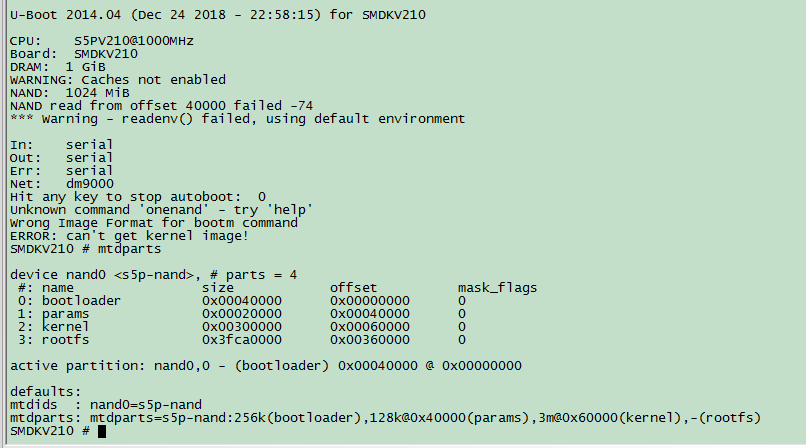
#define CONFIG\_ENV\_SIZE (128 << 10) /\* 128KiB, 0x20000 \*/

#define CONFIG\_ENV\_ADDR (256 << 10) /\* 256KiB, 0x40000 \*/

#define CONFIG\_ENV\_OFFSET (256 << 10) /\* 256KiB, 0x40000 \*/

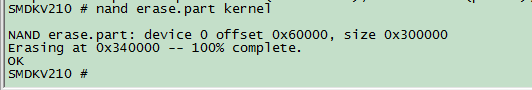
两者需一致。

效果如下：



注意：这里执行了 *mtdparts default* 初始化分析信息，还必须执行 *saveenv* 将其保持到 *NAND*，然后  
重启再次查看分区信息，一切正常

现在可以使用分区操作 *nand* 了



**14.命令行解释器**

参考 *u-boot* 的帮助文档 *README* 中 *Command Interpreter  
CONFIG\_AUTO\_COMPLETE*定义后，可以通过 *Tab* 键进行命令的自动补全，前面 ***10***节已经配置过了。  
*CONFIG\_SYS\_HUSH\_PARSER*定义后，可以使*"hush" shell*（来自于 *Busybox*）作为一个命令行解释器。 这样可以使用一些强大的命令行语法，比如： *if...then...else...fi conditionals or `&&' and '||' constructs ("shell scripts")*这个配置在 *smdkv210.h* 中已经配置了。

*CONFIG\_SYS\_PROMPT\_HUSH\_PS2*定义第 *2* 阶段的命令提示符，当命令行解释器需要获得用户输入时，将打印这个提示符，在*common/hush.c* 中默认定义为*"> "*

#ifndef CONFIG\_SYS\_PROMPT\_HUSH\_PS2

#define CONFIG\_SYS\_PROMPT\_HUSH\_PS2 "> "

#endif

我们可以在 *smdkv210.h* 中定义这个配置。

下面有段注意：

Note:

In the current implementation, the local variables

space and global environment variables space are

separated. Local variables are those you define by

simply typing `name=value'. To access a local

variable later on, you have write `$name' or

`${name}'; to execute the contents of a variable

directly type `$name' at the command prompt.

Global environment variables are those you use

setenv/printenv to work with. To run a command stored

in such a variable, you need to use the run command,

and you must not use the '$' sign to access them.

To store commands and special characters in a

variable, please use double quotation marks

surrounding the whole text of the variable, instead

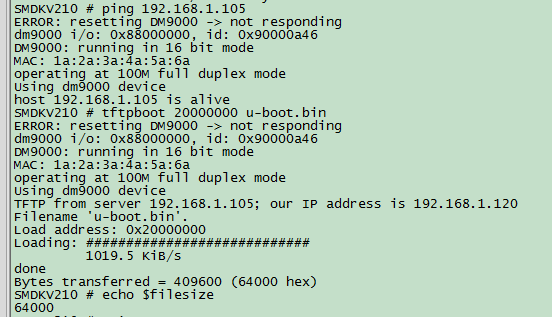
of the backslashes before semicolons and special

symbols.

*u-boot* 中实现了局部变量， 它和环境变量不同，环境变量可以通过 *saveenv* 写入 *FLASH*。  
通过“*name=value*”这样的语法定义一个局部变量， 通过*$ name* 或者*${name}*这样的方式来读取，  
直接在命令行键入*$name* 来执行这个变量包含的内容。



在使用 *tftpboot* 命令下载文件后， *tftpboot* 会将下载的文件大小以十六进制格式保存到局部变量  
*filesize* 中，我们在接下来的操作中可用引用这个变量的值。



说明：此处由于本机的nfs配置一直启动不了，所以换了另外的虚拟机

**15.实现 NAND 8位硬件 ECC**

在 *smdkv210.h* 中定义宏 *CONFIG\_S5PV210\_NAND\_HWECC*、 *CONFIG\_SYS\_NAND\_ECCSIZE*、  
*CONFIG\_SYS\_NAND\_ECCBYTES*

/\*add by zwf\*/

#define CONFIG\_S5PV210\_NAND\_HWECC

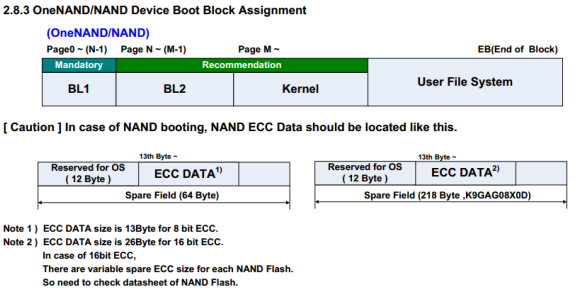
#define CONFIG\_SYS\_NAND\_ECCSIZE 512

#define CONFIG\_SYS\_NAND\_ECCBYTES 13

*CONFIG\_SYS\_NAND\_ECCSIZE* 定义了消息长度，即每多少字节进行 *1* 次 *ECC* 校验  
*CONFIG\_SYS\_NAND\_ECCBYTES* 定义为 *13Byte*，参考裸机编程 *4.10* 节

将 *drivers/mtd/nand/s5pv210\_nand.c* 中的 *CONFIG\_S3C2410\_NAND\_HWECC* 替换为  
*CONFIG\_S5PV210\_NAND\_HWECC*

我们只进行 *ECC* 校验写， *ECC* 校验读使用三星提供的函数，我们必须按照三星手册规定的 *ECC* 校验  
码在 *Spare Field* 中的存储格式进行存储。



因此我们需要自定义 *nand\_ecclayout* 结构体，这个结构体描述了如何存储 *ECC* 数据，同时将这个结  
构体赋值给 *nand->ecc.layout*

*s5pv210\_nand.c (u-boot-2014.04/drivers/mtd/nand/s5pv210\_nand.c)*

#ifdef CONFIG\_S5PV210\_NAND\_HWECC

nand->ecc.hwctl = s5pv210\_nand\_enable\_hwecc;

nand->ecc.calculate = s5pv210\_nand\_calculate\_ecc;

nand->ecc.correct = s5pv210\_nand\_correct\_data;

nand->ecc.mode = NAND\_ECC\_HW;

nand->ecc.size = CONFIG\_SYS\_NAND\_ECCSIZE;

nand->ecc.bytes = CONFIG\_SYS\_NAND\_ECCBYTES;

nand->ecc.strength = 1;

/\* add by zwf \*/

nand->ecc.layout = &nand\_oob\_64;

nand->ecc.read\_page = s5pv210\_nand\_read\_page\_hwecc;

#else

nand->ecc.mode = NAND\_ECC\_SOFT;

#endif

/\* add by zwf \*/

static struct nand\_ecclayout nand\_oob\_64 = {

.eccbytes = 52, /\* 2048 / 512 \* 13 \*/

.eccpos = { 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,

22, 23, 24, 25, 26, 27, 28, 29, 30, 31,

32, 33, 34, 35, 36, 37, 38, 39, 40, 41,

42, 43, 44, 45, 46, 47, 48, 49, 50, 51,

52, 53, 54, 55, 56, 57, 58, 59, 60, 61,

62, 63},

/\* 0和1用于保存坏块标记，12~63保存ecc，剩余2~11为free \*/

.oobfree = {

{.offset = 2,

.length = 10}

}

};

参考裸机编程修改 *s5pv210\_nand\_enable\_hwecc* 和 *s5pv210\_nand\_calculate\_ecc* 函数，同时，我们还  
要使用三星提供的 *8* 位硬件 *ECC* 拷贝函数来自定义一个读页函数， 将它赋值给 *nand->ecc.read\_page*

#ifdef CONFIG\_S5PV210\_NAND\_HWECC

nand->ecc.hwctl = s5pv210\_nand\_enable\_hwecc;

nand->ecc.calculate = s5pv210\_nand\_calculate\_ecc;

nand->ecc.correct = s5pv210\_nand\_correct\_data;

nand->ecc.mode = NAND\_ECC\_HW;

nand->ecc.size = CONFIG\_SYS\_NAND\_ECCSIZE;

nand->ecc.bytes = CONFIG\_SYS\_NAND\_ECCBYTES;

nand->ecc.strength = 1;

/\* add by zwf \*/

nand->ecc.layout = &nand\_oob\_64;

nand->ecc.read\_page = s5pv210\_nand\_read\_page\_hwecc;

#else

nand->ecc.mode = NAND\_ECC\_SOFT;

#endif

/\* add by zwf \*/

#define NF8\_ReadPage\_Adv(a,b,c) (((int(\*)(u32, u32, u8\*))(\*((u32 \*)0xD0037F90)))(a,b,c))

static int s5pv210\_nand\_read\_page\_hwecc(struct mtd\_info \*mtd, struct nand\_chip \*chip,

uint8\_t \*buf, int oob\_required, int page)

{

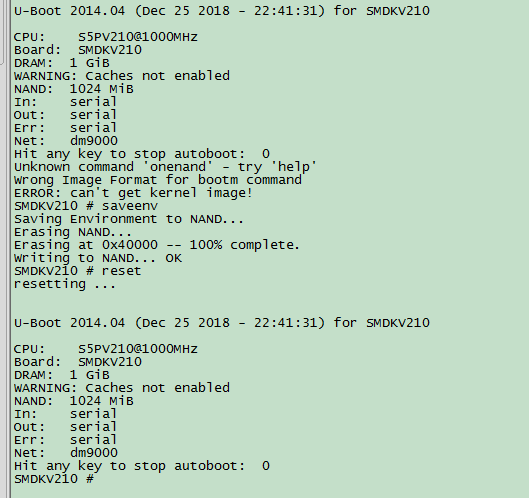
/\* TQ210使用的NAND FLASH一个块64页 \*/

return NF8\_ReadPage\_Adv(page / 64, page % 64, buf);

}

其他的代码请看源码。

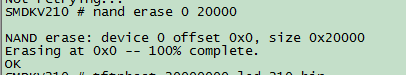
重新编译，成功生成 *u-boot.bin*，将它烧写到 *SD* 卡的扇区 *32*，从 *SD* 卡启动开发板

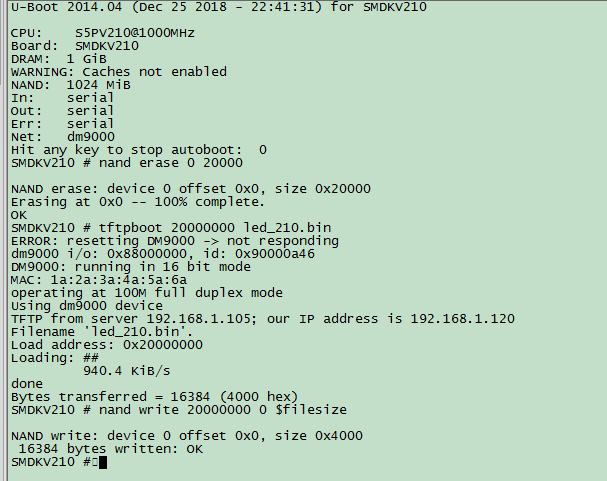


首先保存环境变量，然后重启

下面进行 *NAND* 启动试验，将裸机编程中的 *1-led\_on\_c* 程序烧写到 *NAND*，然后从 *NAND* 启动，可  
以看到 *LED* 流水灯效果。

首先我们先编译 *1-led\_on\_c*，生成 *led.bin*，然后使用 *addheader* 添加 *16B* 的头信息生成 *210.bin*，然  
后将 *210.bin* 拷贝到 *tftp* 服务器目录。





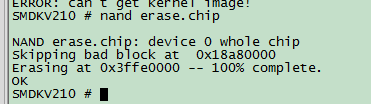
从 *NAND* 启动可以看到 *2* 个 *LED* 交替闪烁

**16.实现 NAND 启动 u-boot**

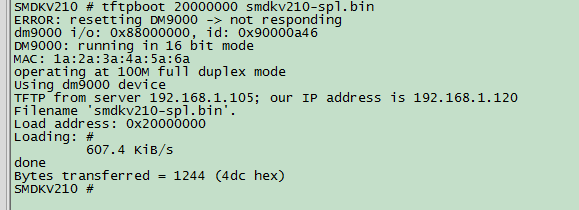
从 *NAND* 启动 *u-boot*，需要 *BL1* 初始化 *NAND* 控制器，然后从 *NAND* 拷贝 *BL2* 到 *DDR* 内存。这里的*BL1* 即我们移植的 *u-boot-spl.bin*， *BL2* 即我们移植的 *u-boot.bin*。

在 *u-boot.bin* 中的 *NAND* 驱动比较大，它包含了很多功能，而 *u-boot-spl.bin* 中只需要从 *NAND* 拷贝*BL2* 到 *DDR* 就行了，因此我们可以在 *u-boot-spl.bin* 进行简单的 *NAND* 控制器初始化，然后使用三星提供的带 *8* 位硬件 *ECC* 的 *NAND* 拷贝函数来拷贝 *u-boot.bin* 到 *DDR* 内存。  
 我们修改 *board/samsung/smdkv210/smdkv210.c* 中的 *copy\_bl2\_to\_ram* 函数，在这个函数中首先判断当前是从 *SD* 卡启动还是从 *NAND* 启动，如果是从 *NAND* 启动则进行 *NAND* 初始化，然后从 *NAND* 拷贝*BL2* 到 *DDR* 内存，如果是从 *SD* 卡启动，则从 *SD* 卡拷贝 *BL2* 到 *DDR* 内存。  
 我们通过读取 *OMR* 寄存器来判断 *S5PV210* 当前是从哪个设备启动的。 参考手册 *Table 6-3*，手册上并没有说 *OM* 寄存器的地址，我是参考的三星原厂的 *u-boot* 代码。  
 代码中都有详细注释，具体请看代码。  
 重新编译，成功生成 *spl/smdkv210-spl.bin* 和 *u-boot.bin*，将它们全部拷贝到 *tftp* 服务器目录下，然后就可以使用上一节移植的 *u-boot* 来烧写最新的 *u-boot* 到 *NAND FLASH*。 首先从 *SD* 卡启动开发板

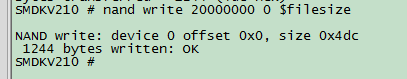
擦除整个 *NAND FLASH*



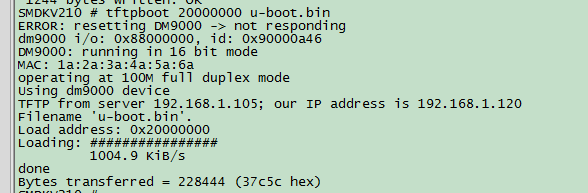
使用 *tftpboot* 下载 *smdkv210-spl.bin* 到 *DDR* 的起始地址 *0x20000000*



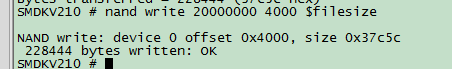
烧写 *smdkv210-spl.bin* 到 *NAND* 的 *0* 地址



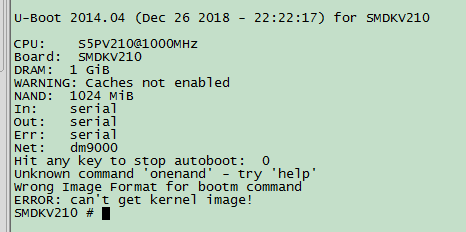
使用 *tftpboot* 下载 *u-boot.bin* 到 *DDR* 的起始地址 *0x20000000*



烧写 *u-boot.bin* 到 *NAND FLASH* 的 *0x4000* 地址（*0x0~0x3FFF* 预留给 *smdkv210-spl.bin*）



拨动拨码开关，从 *NAND* 启动开发板



可以正常从nand flash中启动了

**17 合并 smdkv210-spl.bin 和 u-boot.bin 为一个文件**

之前的烧写都是将 *smdkv210-spl.bin* 和 *u-boot.bin* 分开单独烧写的，这样太麻烦了，我们将这 *2* 个*bin* 文件合并为一个 *bin* 文件——*u-boot.bin*

**myroot@myroot:~/TQ210N/u-boot-2014.04$ cp spl/smdkv210-spl.bin tmp.bin**

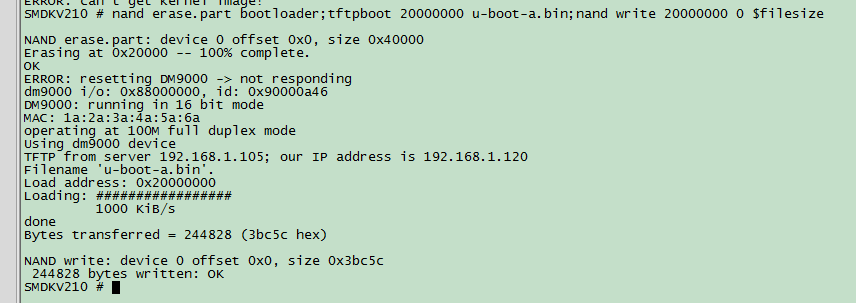
**myroot@myroot:~/TQ210N/u-boot-2014.04$ truncate tmp.bin -c -s 16K**

**myroot@myroot:~/TQ210N/u-boot-2014.04$ cat u-boot.bin >> tmp.bin**

**myroot@myroot:~/TQ210N/u-boot-2014.04$ cp tmp.bin u-boot-a.bin**

首先拷贝 *spl/smdkv210-spl.bin* 为一个临时文件 *tmp.bin*然后使用 *truncate* 命令将 *tmp.bin* 的大小扩展为 *16KB*

*truncate*后面跟的参数 *-c* 表示不创建文件， *-s* 指定扩展的大小。  
然后使用 *cat* 命令读取 *u-boot.bin* 的内容追加到 *tmp.bin* 后面  
然后拷贝 *tmp.bin* 为 *u-boot-a.bin*现在我们就可以将这个 *u-boot-a.bin* 烧写到 *NAND FLASH* 了。  
使用 *DD\_For\_Windows.exe* 烧写到 *SD* 卡启动不了，还在调试中。  
我们还是按照原来的方式使用 *DD\_For\_Windows.exe* 烧写 *smdkv210-spl.bin* 到 *SD* 卡的扇区 *1*，  
烧写 *u-boot.bin* 到 *SD* 卡的扇区 *32*，从 *SD* 卡启动开发板， 现在可以使用 *u-boot* 烧写 *u-boot-a.bin* 到*NAND FLASH* 了



首先使用 *nand erase.part bootloader* 命令擦除 *bootloader* 分区，然后拨动拨码开关从 *NAND* 启动。  
我们可以将上面合并单文件的那一系列命令写入 *Makefile*，这样比较方便。  
首先在 *ALL* 中添加一个目标 *combine*

# Always append ALL so that arch config.mk's can add custom ones

# combine add by zwf

ALL-y += u-boot.srec u-boot.bin System.map combine

然后定义 *combine* 的规则

#add by zwf

combine:u-boot.bin spl/u-boot-spl.bin FORCE

cp $(objtree)/spl/smdkv210-spl.bin $(objtree)/tmp.bin

truncate $(objtree)/tmp.bin -c -s 16K

cat $(objtree)/u-boot.bin >> $(objtree)/tmp.bin

cp $(objtree)/tmp.bin $(objtree)/u-boot-a.bin

我们在最后将合并的文件 *u-boot-a.bin* 拷贝到 *tftp* 服务器目录，这个路径根据自己的实际环境修改  
现在我们再次编译一下

OBJCOPY spl/u-boot-spl.bin

/home/myroot/TQ210N/u-boot-2014.04/tools/mksmdkv210spl spl/u-boot-spl.bin spl/smdkv210-spl.bin

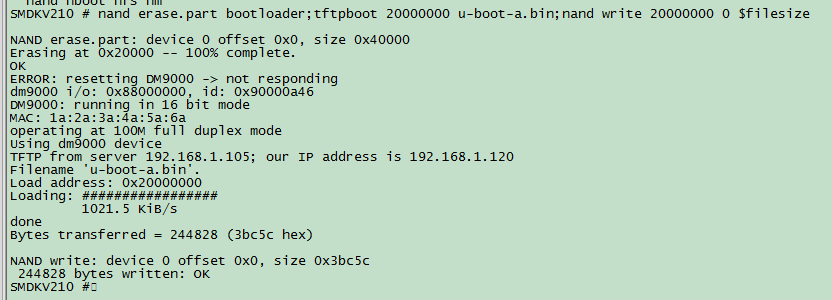
cp /home/myroot/TQ210N/u-boot-2014.04/spl/smdkv210-spl.bin /home/myroot/TQ210N/u-boot-2014.04/tmp.bin

truncate /home/myroot/TQ210N/u-boot-2014.04/tmp.bin -c -s 16K

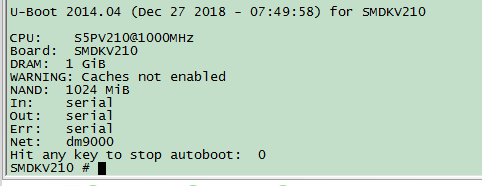
cat /home/myroot/TQ210N/u-boot-2014.04/u-boot.bin >> /home/myroot/TQ210N/u-boot-2014.04/tmp.bin

cp /home/myroot/TQ210N/u-boot-2014.04/tmp.bin /home/myroot/TQ210N/u-boot-2014.04/u-boot-a.bin

可以看到已经成功执行这些命令了，这样每次 *make all* 后，就可以立即烧写了。



然后选择从NAND启动，可以正常启动了



**18 添加启动菜单**

参考帮助文档 *doc/README.bootmenu*首先我们需要在 *smdkv210.h* 中增加 *4* 个宏

/\*add by zwf for bootmenu follow 4\*/

#define CONFIG\_CMD\_BOOTMENU

#define CONFIG\_MENU

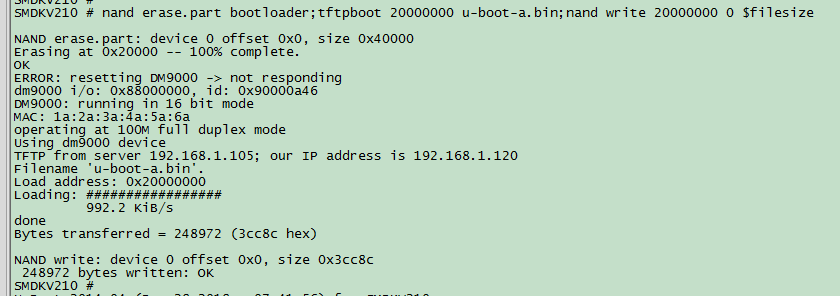
#define CONFIG\_AUTOBOOT\_KEYED

#define CONFIG\_MENU\_SHOW

另外，帮助文档还提到要定义 *CONFIG\_BOOTDELAY*，表示显示菜单多久，这个宏已经定义了，默认  
为 *3s*

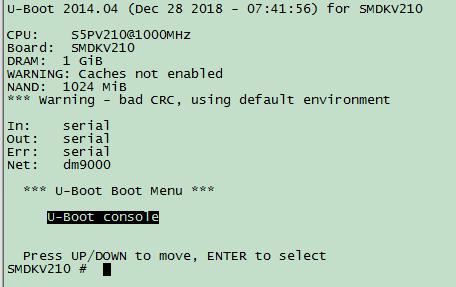
#define CONFIG\_BOOTDELAY 3

现在重新编译 *u-boot*， 从 *NAND FLASH* 启动开发板， 将 *u-boot-a.bin* 烧写到 *NAND FLASH*



我这里将所有命令写在一行命令之间使用“；”隔开，将这一行命令拷贝到一个记事本，以后操作  
时直接复制过来，一个回车就搞定了。 然后执行 *reset* 复位

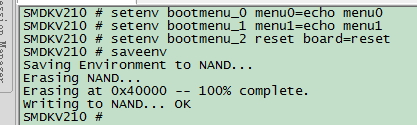
nand erase.part bootloader;tftpboot 20000000 u-boot-a.bin;nand write 20000000 0 $filesize



可以看到， *u-boot* 显示了默认的菜单界面， 这时只有一条默认菜单， *3s* 延迟后， *u-boot* 就执行第 *1*条菜单进入到命令行模式了。  
我们通过设置环境变量来添加菜单。  
**使用 *setenv* 添加菜单的格式为***bootmenu\_<num>="<title>=<commands>  
<num>* 菜单编号，从 *0* 开始  
*<title>* 菜单标题

*<commands>* 当一个菜单被选中后要执行的命令  
标题和命令以第一个“*=*” 为分界点  
延迟结束后 *u-boot* 立即执行第一个菜单的命令  
按上下键来选择菜单，被选中的菜单以高亮显示，按回车键执行被选中的菜单的命令  
按 *Esc* 键或者 *Q* 键退出延迟计数，一直显示菜单，直到用户选择某个菜单执行

下面来添加几个菜单，添加完成后，使用 *saveenv* 保存环境变量到 *NAND FLASH*



SMDKV210 # setenv bootmenu\_0 menu0=echo menu0

SMDKV210 # setenv bootmenu\_1 menu1=echo menu1

SMDKV210 # setenv bootmenu\_2 reset board=reset

SMDKV210 # saveenv

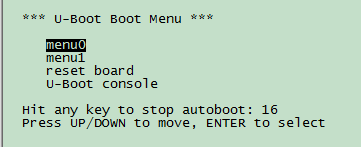
现在执行 *reset* 命令重启开发板

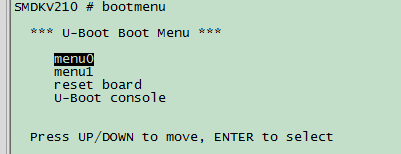


延迟结束后，执行了第 *1* 个菜单的命令，打印了 *menu0*。

我们还可以使用 *bootmenu* 跟一个延迟参数来启动菜单，不跟参数的话，以环境变量 *bootdelay* 为延  
迟时间







我们可以按 *Q* 键或者 *Esc* 键退出延迟计数，一直显示菜单直到用户选择某个菜单。

**19 实现 LCD 显示**

*arch/arm/lib/board.c* 中的 *board\_init\_f* 函数

#ifdef CONFIG\_LCD

#ifdef CONFIG\_FB\_ADDR

gd->fb\_base = CONFIG\_FB\_ADDR;

#else

/\* reserve memory for LCD display (always full pages) \*/

addr = lcd\_setmem(addr);

gd->fb\_base = addr;

#endif /\* CONFIG\_FB\_ADDR \*/

#endif /\* CONFIG\_LCD \*/

如果定义了 *CONFIG\_LCD*，就会调用了 *lcd\_setmem* 函数，这个函数用来分配帧缓存地址， 其定义在  
*common/lcd.c*

ulong lcd\_setmem(ulong addr)

{

ulong size;

int line\_length;

debug("LCD panel info: %d x %d, %d bit/pix\n", panel\_info.vl\_col,

panel\_info.vl\_row, NBITS(panel\_info.vl\_bpix));

size = lcd\_get\_size(&line\_length);

/\* Round up to nearest full page, or MMU section if defined \*/

size = ALIGN(size, CONFIG\_LCD\_ALIGNMENT);

addr = ALIGN(addr - CONFIG\_LCD\_ALIGNMENT + 1, CONFIG\_LCD\_ALIGNMENT);

/\* Allocate pages for the frame buffer. \*/

addr -= size;

debug("Reserving %ldk for LCD Framebuffer at: %08lx\n",

size >> 10, addr);

return addr;

}

这个函数调用了 *panel\_info* 变量，这个变量在每个具体的 *lcd* 驱动中定义，定义了具体 *LCD* 的相关信  
息，比如这里通过 *debug* 打印了它的成员变量： *vl\_col*（列）、 *vl\_row*（行）和 *vl\_bpix*（*BPP*）。 然后调用*lcd\_get\_size* 函数来获得帧缓存大小。 *lcd\_get\_size* 在 *common/lcd.c* 中定义为一个弱符号，我们可以在自己的 *LCD* 驱动中重定义这个函数，也可以使用默认的。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* \*\* GENERIC Initialization Routines \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* With most lcd drivers the line length is set up

\* by calculating it from panel\_info parameters. Some

\* drivers need to calculate the line length differently,

\* so make the function weak to allow overriding it.

\*/

\_\_weak int lcd\_get\_size(int \*line\_length)

{

\*line\_length = (panel\_info.vl\_col \* NBITS(panel\_info.vl\_bpix)) / 8;

return \*line\_length \* panel\_info.vl\_row;

}

*arch/arm/lib/board.c* 中的 *board\_init\_r* 函数

stdio\_init(); /\* get the devices list going. \*/

这里调用了 *stdio\_init*，初始化标准输入、标准输出、标准错误输出等 *IO*。 它在 *common/stdio.c* 中定  
义

*common/stdio.c* 中的 *stdio\_init* 函数

#ifdef CONFIG\_LCD

drv\_lcd\_init ();

#endif

如果定义了 *CONFIG\_LCD*，就调用 *drv\_lcd\_init*，其定义在 *common/lcd.c*

int drv\_lcd\_init(void)

{

struct stdio\_dev lcddev;

int rc;

lcd\_base = map\_sysmem(gd->fb\_base, 0);

lcd\_init(lcd\_base); /\* LCD initialization \*/

/\* Device initialization \*/

memset(&lcddev, 0, sizeof(lcddev));

strcpy(lcddev.name, "lcd");

lcddev.ext = 0; /\* No extensions \*/

lcddev.flags = DEV\_FLAGS\_OUTPUT; /\* Output only \*/

lcddev.putc = lcd\_putc; /\* 'putc' function \*/

lcddev.puts = lcd\_puts; /\* 'puts' function \*/

rc = stdio\_register(&lcddev);

return (rc == 0) ? 1 : rc;

}

这个函数又调用了当前文件中的 *lcd\_init*，初始化 *LCD*

static int lcd\_init(void \*lcdbase)

{

/\* Initialize the lcd controller \*/

debug("[LCD] Initializing LCD frambuffer at %p\n", lcdbase);

lcd\_ctrl\_init(lcdbase);

/\*

\* lcd\_ctrl\_init() of some drivers (i.e. bcm2835 on rpi\_b) ignores

\* the 'lcdbase' argument and uses custom lcd base address

\* by setting up gd->fb\_base. Check for this condition and fixup

\* 'lcd\_base' address.

\*/

if (map\_to\_sysmem(lcdbase) != gd->fb\_base)

lcd\_base = map\_sysmem(gd->fb\_base, 0);

debug("[LCD] Using LCD frambuffer at %p\n", lcd\_base);

lcd\_get\_size(&lcd\_line\_length);

lcd\_is\_enabled = 1;

lcd\_clear();

lcd\_enable();

/\* Initialize the console \*/

console\_col = 0;

#ifdef CONFIG\_LCD\_INFO\_BELOW\_LOGO

console\_row = 7 + BMP\_LOGO\_HEIGHT / VIDEO\_FONT\_HEIGHT;

#else

console\_row = 1; /\* leave 1 blank line below logo \*/

#endif

return 0;

}

这个函数调用了 *lcd\_ctrl\_init*，这个函数在具体的 *LCD* 驱动中实现，我们需要自己实现这个函数，它进行具体的硬件初始化，下面调用 *lcd\_clear* 清屏， *lcd\_enable* 使能 *LCD* 控制器，这个函数也需要我们自己实现。 分析到这里，我们就可以开始移植 *LCD* 驱动了。  
 首先在*drivers/video*目录下创建文件*s5pv210\_fb.c*，同时修改 *drivers/video/Makefile*，把 *s5pv210\_fb.c*编译进 *u-boot.bin*

# add by zwf

obj-$(CONFIG\_S5PV210\_LCD) += s5pv210\_fb.o

然后在 *smdkv210.h* 中定义相关宏

/\*add by zwf\*/

#define CONFIG\_LCD

#define CONFIG\_S5PV210\_LCD

按照惯例，我们需要在 *s5pv210\_fb.c* 中操作 *LCD* 控制器的寄存器，因此需要在  
*arch/arm/include/asm/arch-s5pc1xx/*下创建 *lcd\_reg.h*，定义 *LCD* 控制器的寄存器结构体

/\* add by zwf \*/

#ifndef \_\_asm\_arm\_arch\_dram\_h\_

#define \_\_asm\_arm\_arch\_dram\_h\_

#ifndef \_\_assembly\_\_

struct s5pv210\_lcd {

unsigned int vidcon0;

unsigned int vidcon1;

unsigned int vidcon2;

同样需要修改 *arch/arm/include/asm/arch-s5pc1xx/cpu.h*

#define S5PV210\_NAND\_BASE 0xB0E00000

#define S5PV210\_LCD\_BASE 0xF8000000

定义 *LCD* 控制器的基地址

/\* add by zwf \*/

SAMSUNG\_BASE(dmc0, DMC0\_BASE)

SAMSUNG\_BASE(dmc1, DMC1\_BASE)

SAMSUNG\_BASE(nand, NAND\_BASE)

SAMSUNG\_BASE(lcd, LCD\_BASE)

定义获得 *LCD* 控制器基地址的宏

然后仿照裸机编程中 *LCD* 一节的代码，在 *s5pv210\_fb.c* 中实现 *lcd\_ctrl\_init* 函数和 *lcd\_enable* 函数。  
还要定义一个用来描述 *LCD* 相关信息的结构体

vidinfo\_t panel\_info = {

.vl\_col = 800,

.vl\_row = 480,

.vl\_bpix = LCD\_COLOR24,

};

这个结构体中使用了一个宏 *LCD\_COLOR24*，表示使用 *24BPP*，在 *include/lcd.h* 中只定义了下面几个  
表示颜色位深的宏

#define LCD\_MONOCHROME 0

#define LCD\_COLOR2 1

#define LCD\_COLOR4 2

#define LCD\_COLOR8 3

#define LCD\_COLOR16 4

我们的 *LCD* 使用的是 *24BPP*，因此需要在其后面添加一个 *LCD\_COLOR24*

#define LCD\_COLOR16 4

/\* add by zwf \*/

#define LCD\_COLOR24 5

另外，在下面定义了默认的 *LCD\_BPP* 为 *LCD\_COLOR8*

/\* Default to 8bpp if bit depth not specified \*/

#ifndef LCD\_BPP

# define LCD\_BPP LCD\_COLOR8

#endif

我们使用的是 *24BPP*，因此需要在 *smdkv210.h* 中定义 *LCD\_BPP* 为 *24BPP*

/\*add by zwf\*/

#define CONFIG\_LCD

#define CONFIG\_S5PV210\_LCD

#define LCD\_BPP LCD\_COLOR24

下面开始编译，出错

CC common/lcd.o

common/lcd.c:106:3: error: #error Unsupported LCD BPP.

common/lcd.c: In function 'console\_scrollup':

common/lcd.c:181: warning: implicit declaration of function 'COLOR\_MASK'

common/lcd.c: In function 'lcd\_drawchars':

common/lcd.c:309: warning: unused variable 'd'

scripts/Makefile.build:335: recipe for target 'common/lcd.o' failed

make[1]: \*\*\* [common/lcd.o] Error 1

Makefile:947: recipe for target 'common' failed

make: \*\*\* [common] Error 2

找到 *common/lcd.c* 第 *106* 行

#if LCD\_BPP == LCD\_MONOCHROME

# define COLOR\_MASK(c) ((c) | (c) << 1 | (c) << 2 | (c) << 3 | \

(c) << 4 | (c) << 5 | (c) << 6 | (c) << 7)

#elif (LCD\_BPP == LCD\_COLOR8) || (LCD\_BPP == LCD\_COLOR16)

# define COLOR\_MASK(c) (c)

#else

# error Unsupported LCD BPP.

#endif

如果 *LCD\_BPP == LCD\_COLOR8* 或者 *LCD\_BPP == LCD\_COLOR16* ，就定义 *COLOR\_MASK*，否则就输出  
错误信息 *Unsupported LCD BPP*，我们的 *BPP* 为 *24*，因此需要修改这段代码

/\* modied by zwf \*/

#elif (LCD\_BPP == LCD\_COLOR8) || (LCD\_BPP == LCD\_COLOR16 || (LCD\_BPP == LCD\_COLOR24)

# define COLOR\_MASK(c) (c)

我们还需要修改 *lcd\_drawchars* 函数，也是与 *BPP* 相关的，否则屏幕没输出

#if LCD\_BPP == LCD\_COLOR16

ushort \*d = (ushort \*)dest;

/\* add by zwf \*/

#elif LCD\_BPP == LCD\_COLOR24

uint\*d = (uint \*)dest;

#else

uchar \*d = dest;

#endif

我们使用 *24BPP*，一个像素在内存中占 *32* 位，有一个字节被浪费掉了

#elif LCD\_BPP == LCD\_COLOR16

for (c = 0; c < 8; ++c) {

\*d++ = (bits & 0x80) ?

lcd\_color\_fg : lcd\_color\_bg;

bits <<= 1;

}

/\* add by zwf \*/

#elif LCD\_BPP == LCD\_COLOR24

for (c = 0; c < 8; ++c) {

\*d++ = (bits & 0x80) ?

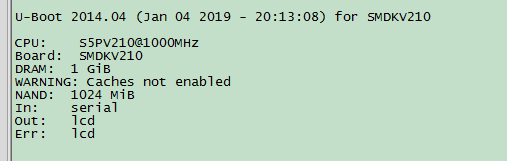
lcd\_color\_fg : lcd\_color\_bg;

bits <<= 1;

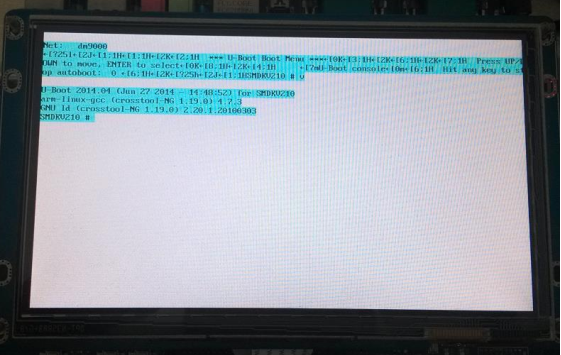
}

#endif

再次编译，成功将文字输出到 *LCD* 上了



*stdio* 为串口，*stdou* 和 *stderr* 已经定向到 *lcd* 了，我们可以在串口终端输入东西，但是我们只能在 *LCD*上看到我们输入了什么。



我们可以设置环境变量 *stdout=serial*，将标准输出重新定向到串口，我们在串口终端输入  
 *set stdout serial*注意：我们的输入没法在串口终端显示出来，但可以在 *LCD* 上看到我们的输入，然后按回车执行后，  
就将标准输出切换到串口终端了。  
 在 *u-boot* 中，将标准输出（*stdou*）定向到 *LCD* 是没什么意义的， 我希望的是在 *u-boot* 中显示开机画面，这样避免了在 *u-boot* 启动到内核启动期间屏幕一直黑屏。

在 *common/lcd.c* 中的 *lcd\_clear* 函数中调用了当前文件中的 *lcd\_logo*

static void \*lcd\_logo(void)

{

#ifdef CONFIG\_SPLASH\_SCREEN

char \*s;

ulong addr;

static int do\_splash = 1;

if (do\_splash && (s = getenv("splashimage")) != NULL) {

int x = 0, y = 0;

do\_splash = 0;

if (splash\_screen\_prepare())

return (void \*)lcd\_base;

addr = simple\_strtoul (s, NULL, 16);

splash\_get\_pos(&x, &y);

if (bmp\_display(addr, x, y) == 0)

return (void \*)lcd\_base;

}

#endif /\* CONFIG\_SPLASH\_SCREEN \*/

*splash screen*即启动画面的意思。  
 如果定义了*CONFIG\_SPLASH\_SCREEN*，就会执行下面这段代码。首先获得环境变量 *splashimage* 的值保存到 *s* 中，然后调用 *simple\_strtoul* 将它转换为 *16* 进制的整数保存到 *addr* 中。  
 环境变量 *splashimage* 保存了要显示的图片的地址，然后调用 *splash\_screen\_prepare* 做一些准备工作，比如：我们的图片实际保存在 *SD* 卡或者 *FLASH* 中， *splash\_screen\_prepare* 这个函数就可以从 *SD* 卡或者 *FLASH* 中将图片读取到环境变量 *splashimage* 指定的地址，接着调用 *bmp\_display* 函数显示图片，给它传了一个参数 *addr*， *addr* 保存了要显示的图片在内存中的地址。 因此，我们需要准备一张 *BMP* 格式的图片。

首先在 *smdkv210.h* 中定义宏 *CONFIG\_SPLASH\_SCREEN*

/\*add by zwf\*/

#define CONFIG\_LCD

#define CONFIG\_S5PV210\_LCD

#define LCD\_BPP LCD\_COLOR24

#define CONFIG\_SPLASH\_SCREEN

我们还需要修改分区信息，增加一个分区用来存储 *BMP* 图片，我制作了一张 *800x480* 的 *BMP* 格式  
的图片，大约 *1M* 多点。我们就给它分配 *2M* 的空间。

/\*modied by zwf\*/

#define MTDIDS\_DEFAULT "nand0=s5p-nand"

#define MTDPARTS\_DEFAULT "mtdparts=s5p-nand:256k(bootloader)"\

",128k@0x40000(params)"\

",2m@0x60000(log)"\

",3m@0x260000(kernel)"\

",-(rootfs)"

*CONFIG\_EXTRA\_ENV\_SETTINGS*中定义一下额外的环境变量，将默认的删除掉，然后定义*splashimage* 为 *0x23000000*，我们将 *BMP* 图片从 *FLASH* 中加载到这个地址。

#define CONFIG\_ENV\_OVERWRITE

/\* modied by zwf \*/

#define CONFIG\_EXTRA\_ENV\_SETTINGS \

"splashimage=0x23000000" \

另外，我们还需要实现 *splash\_screen\_prepare* 函数，用来从 *FLASH* 中拷贝 *BMP* 图片到环境变量  
*splashimage* 指定的地址。 我们在 *s5pv210\_fb.c* 中定义 *splash\_screen\_prepare*

/\*

\*\* 将NAND FLASH的log分区的数据读取到环境变量

\*\* splashimage指定的内存地址

\*/

int splash\_screen\_prepare(void)

{

char \*s = NULL;

char cmd[100];

if ((s = getenv("splashimage")) == NULL)

{

printf("Not set environable: splashimage\n");

return 1;

}

sprintf(cmd, "nand read %s log", s);

return run\_command\_list(cmd, -1, 0);

}

这里我们使用了*run\_command\_list* 来执行命令，这样比较方便。

编译出错

common/built-in.o: In function `lcd\_logo':

/home/myroot/TQ210N/u-boot-2014.04/common/lcd.c:1105: undefined reference to `bmp\_display'

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9537

arm-linux-ld: BFD (Sourcery G++ Lite 2008q3-72) 2.18.50.20080215 assertion fail /scratch/julian/lite-respin/linux/obj/binutils-src-2008q3-72-arm-none-linux-gnueabi-i686-pc-linux-gnu/bfd/elf32-arm.c:9771

Segmentation fault (core dumped)

Makefile:926: recipe for target 'u-boot' failed

make: \*\*\* [u-boot] Error 139

myroot@myroot:~/TQ210N/u-boot-2014.04$

没有定义 *bmp\_display*，通过 *SourceInght* 跟踪代码发现 *bmp\_display* 在 *common/cmd\_bmp.c* 中定义，  
我们看下 *common* 下的 *Makefile*

obj-$(CONFIG\_CMD\_BMP) += cmd\_bmp.o

发现需要定义 *CONFIG\_CMD\_BMP* 这个宏，我们在 *smdkv210.h* 中来定义它

/\*add by zwf\*/

#define CONFIG\_LCD

#define CONFIG\_S5PV210\_LCD

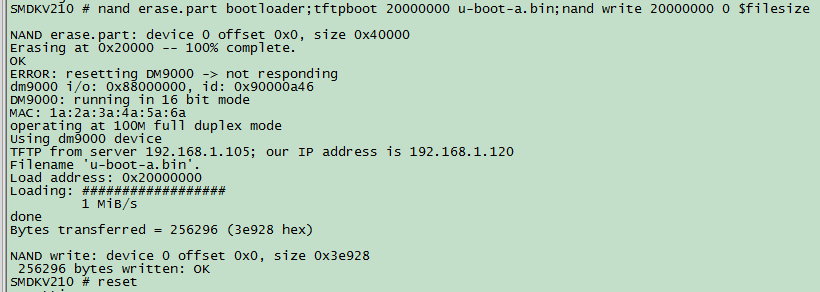
#define LCD\_BPP LCD\_COLOR24

#define CONFIG\_SPLASH\_SCREEN

#define CONFIG\_CMD\_BMP

#endif /\* \_\_CONFIG\_H \*/

再次编译，成功生成 *u-boot-a.bin*，如果还是出错，需要执行 *make distclean*，重新配置、编译。  
 前面移植了 *u-boot* 的 *LCD* 驱动，把输出定向到了 *LCD*，我们通过设置环境变量 *stdout* 重新将标准输出定向到串口（前面讲过），然后烧写 *u-boot-a.bin* 到 *NAND FLASH*



然后重启  
我们发现，添加启动画面后， *u-boot* 已经不再把标准输出定向到 *LCD* 了，看 *common/console.c* 中的  
*console\_init\_r* 就明白了

/\* Called after the relocation - use desired console functions \*/

int console\_init\_r(void)

{

struct stdio\_dev \*inputdev = NULL, \*outputdev = NULL;

int i;

struct list\_head \*list = stdio\_get\_list();

struct list\_head \*pos;

struct stdio\_dev \*dev;

#ifdef CONFIG\_SPLASH\_SCREEN

/\*

\* suppress all output if splash screen is enabled and we have

\* a bmp to display. We redirect the output from frame buffer

\* console to serial console in this case or suppress it if

\* "silent" mode was requested.

\*/

if (getenv("splashimage") != NULL) {

if (!(gd->flags & GD\_FLG\_SILENT))

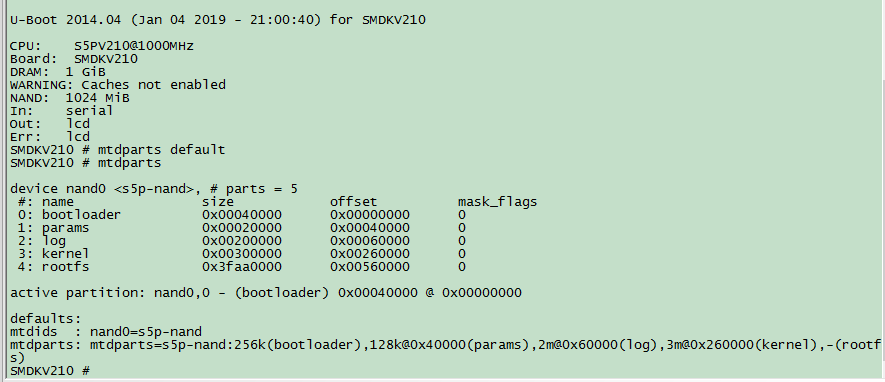
outputdev = search\_device (DEV\_FLAGS\_OUTPUT, "serial");

}

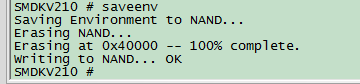
#endif

如果定义了环境变量 *splashimage*，就把标准输出定向到了 *serial*。

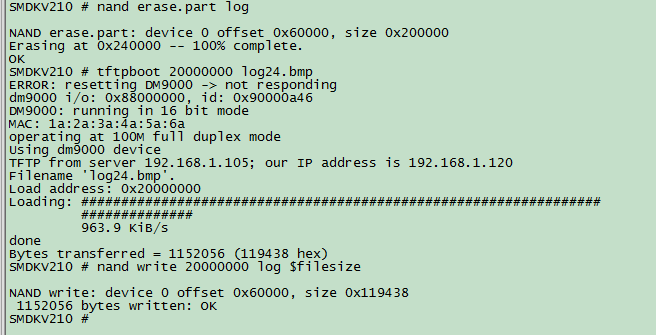
由于现在还没有烧写图片到 *NAND FLASH*，所以 *LCD* 没有显示，下面烧写图片到 *NAND* 的 *log* 分区首先需要加载默认的分区，然后查看一下分区，看是否有 *log* 分区



执行 *saveenv* 将新的分区信息保存到 *NAND*



将制作好的 *bmp* 图片写入 *NAND*



然后重启， *LCD*没有显示图片，串口输出一条错误信息



提示需要 *32* 位的图片，我之前制作了 *2* 张 *bmp* 图片，一张 *24* 位，一张 *32* 位，现在我烧写 *32* 位的图片。

首先我们添加几个菜单，方便以后操作

SMDKV210 # set bootmenu\_0 start kernel=echo

SMDKV210 # set bootmenu\_1 update u-boot(u-boot-a.bin)=nand erase.part bootloader\;tftpboot 20000000 u-boot-a.bin\;nand write 20000000 0 $filesize

SMDKV210 # set bootmenu\_2 update log(log32.bmp)=nand erase.part log\;tftpboot 20000000 log32.bmp\;nand write 20000000 log $filesize

SMDKV210 # saveenv

Saving Environment to NAND...

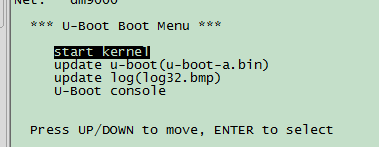
Erasing NAND...

Erasing at 0x40000 -- 100% complete.

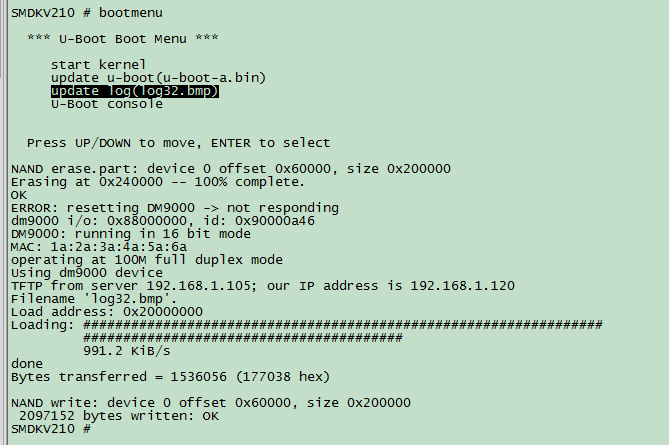
Writing to NAND... OK

SMDKV210 #

我们添加了 *3* 个菜单， 菜单 *0* 通常用来执行 *bootcmd* 指定的命令，用来启动内核，默认执行菜单 *0*，  
由于现在我们还没移植内核，所以用 *echo* 命令打印空，什么也不做。 另外添加了更新 *u-boot* 和 *log*的菜单。



现在我们可以使用上下键选择菜单 *download log* 来下载图片了，将 *32* 位图片下载到 *NAND* 后重启



没有提示错误了，但 *LCD* 还是没有显示，跟踪代码发现，在 *bmp\_display* 中调用的  
*lcd\_display\_bitmap* 函数是根据 *CONFIG\_BMP\_xx* 来决定如何显示图片的，我们没有定义这个宏，导  
致 *lcd\_display\_bitmap* 没有显示任何东西。我们需要定义的是 *CONFIG\_BMP\_32BPP*，在 *smdkv210.h* 中定义

/\*add by zwf\*/

#define CONFIG\_LCD

#define CONFIG\_S5PV210\_LCD

#define LCD\_BPP LCD\_COLOR24

#define CONFIG\_SPLASH\_SCREEN

#define CONFIG\_CMD\_BMP

#define CONFIG\_BMP\_32BPP

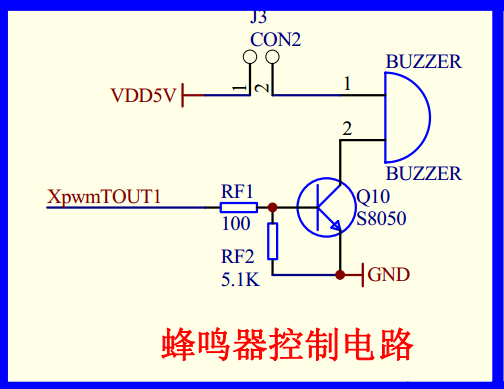
重新编译，把 *u-boot-a.bin*烧写到 *NAND*，重启开发板，选择 *update u-boot(u-boot-a.bin)*菜单更新 *u-boot*

重启开发板，已经显示图片了

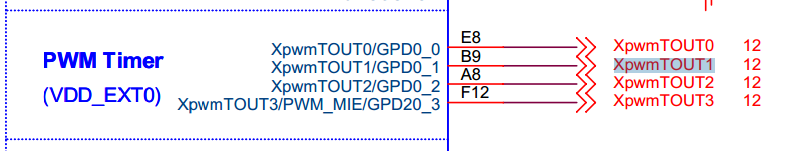


**20 实现开机提示音**

*TQ210* 开发板使用的是有源蜂鸣器，只要给它通电，蜂鸣器就会发声，实现比较简单。如果是无源  
蜂鸣器的话，那就稍微麻烦些，需要给它输入一定的频率，蜂鸣器才会发声。  
 首先查看 *TQ210* 开发板底板原理图



由原理图可以得知，给 *XpwmTOUT1* 一个高电平，蜂鸣器就会发声  
我们再查看 *TQ210* 核心板原理图，看 *XpwmTOUT1* 接在哪个引脚



*XpwmTOUT1* 接在 *GPD0\_1* 引脚，我们在 *smdkv210.c* 中增加蜂鸣器控制函数

/\* add by zwf \*/

void beeper\_ctrl(u8 on)

{

u32 val = 0;

u32 \*gpd0con=(u32 \*)0xE02000A0; //GPD0CON

u32 \*gpd0dat=(u32 \*)0xE02000A4; //GPD0DAT

/\*配置GPD0[1]为输出\*/

val=readl(gpd0con);

val=(val & ~(0xF << 4)) | (1 << 4);

writel(val,gpd0con);

if(on){// 开启蜂鸣器

writel(readl(gpd0dat) |(1<< 1),gpd0dat);

}

else{//关闭蜂鸣器

writel(readl(gpd0dat) &~(1<< 1),gpd0dat);

}

}

#else /\* CONFIG\_SPL\_BUILD (add by zwf) \*/

在 *board.c* 中开启蜂鸣器发声，延迟 *1000ms* 后关闭蜂鸣器

/u-boot-2014.04/arch/arm/lib/*board.c*

void board\_init\_r(gd\_t \*id, ulong dest\_addr)

{

ulong malloc\_start;

#if !defined(CONFIG\_SYS\_NO\_FLASH)

ulong flash\_size;

#endif

/\*add by zwf\*/

beeper\_ctrl(1); //开启蜂鸣器

mdelay(1000); //延迟1000ms

beeper\_ctrl(0); //关闭蜂鸣器

编译：烧写到开发板，重新上电。蜂鸣器响1s后关闭。