目录

[1 uboot 初步编译分析 3](#_Toc527378654)

[1.1下载uboot 3](#_Toc527378655)

[1.2编译Uboot 3](#_Toc527378656)

[1.2.1 编译uboot 3](#_Toc527378657)

[1.2.2 安装新的编译器 3](#_Toc527378658)

[1.2.3 重新编译uboot 3](#_Toc527378659)

[1.3 uboot 初步分析 4](#_Toc527378660)

[2 uboot 流程具体分析 4](#_Toc527378661)

[2.1 uboot 第一阶段流程分析 4](#_Toc527378662)

[2.2 uboot 第二阶段流程分析 5](#_Toc527378663)

[2.2.1 内核启动准备 5](#_Toc527378664)

[2.2.2 启动内核 5](#_Toc527378665)

[2.3 uboot内存分布图 6](#_Toc527378666)

[2.4 重定位代码 6](#_Toc527378667)

[3 移植uboot 9](#_Toc527378668)

[3.1 uboot-2440 初步编译 9](#_Toc527378669)

[3.1.1 新建smdk2440板文件 9](#_Toc527378670)

[3.1.2 新建配置文件 10](#_Toc527378671)

[3.1.3 添加编译smdk2440选项 10](#_Toc527378672)

[3.1.4 编译smdk2440 10](#_Toc527378673)

[3.2 分析并修改代码 10](#_Toc527378674)

[3.2.1 分析时钟和内存 10](#_Toc527378675)

[3.2.2 设置时钟 10](#_Toc527378676)

[3.2.3 修改内存 11](#_Toc527378677)

[3.2.4修改串口 12](#_Toc527378678)

[3.2.5 修改启动代码支持Nand启动，清bss，设置栈,设置机器ID 13](#_Toc527378679)

[3.2.6 修改支持Nor Flash 19](#_Toc527378680)

[3.2.7 修改代码支持nand flash 21](#_Toc527378681)

[3.2.8 设置支持网卡DM9000 23](#_Toc527378682)

[3.2.9 设置环境变量 25](#_Toc527378683)

[3.2.10 支持烧写文件系统 29](#_Toc527378684)

[3.2.11 修改uboot内部的bug 29](#_Toc527378685)

[4 uboot 命令 29](#_Toc527378686)

[5 uboot中下载程序 30](#_Toc527378687)

[5.1 使用usb 下载uboot 30](#_Toc527378688)

[5.1.1 下载程序到nor flash 30](#_Toc527378689)

[5.1.2 下载程序到nand flash 30](#_Toc527378690)

[5.2 使用tftp 下载内核 31](#_Toc527378691)

[5.3 制作补丁 31](#_Toc527378692)

Uboot 移植

# 1 uboot 初步编译分析

## 1.1下载uboot

百度搜索Uboot 找到网页web home<uboot> DENX

左侧栏：source code页面找到ftp server。找到u-boot-2012.04.01.tar.bz2下载源码。

## 1.2编译Uboot

### 1.2.1 编译uboot

tar xjf u-boot-2012.04.01.tat.bz2 //解压

cd u-boot-2012.04.01/ //进入目录

make smdk2410\_config //配置

make //编译

编译产生错误，主要是编译器版本不匹配。

arm-linux-gcc -v获得版本号

### 1.2.2 安装新的编译器

下载arm-linux-gcc-4.3.2 源码

创建目录temp

我下载的是.tgz格式文件

mkdir temp

Sudo tar –zxv –f arm-linux-gcc-4.3.2.tgz –C temp/ //解压到temp目录，查看内容，如果是bz2 使用tar xjf arm-linux-gcc-4.3.2.tar.bz2 –C temp/

要使用编译器必须解压到根目录：

1> Sudo tar –zxv –f arm-linux-gcc-4.3.2.tgz –C / //解压到根目录

2> 配置使新的编译器生效：

cd /usr/local/arm/4.3.2/bin/

echo $PATH

export PATH=/usr/local/arm/4.3.2/bin:/usr/…（路径2中除去/usr/local/arm/3.4.5相关路径，uboot不依赖于glibc，无需安装，编译lInux时需要安装）

arm-linux-gcc -v

以上方法每次使用都需要重新设置，还可以修改环境变量：

sudo vim /etc/environment 将PATH=路径修改，把3.4.5路径替换为/usr/local/arm/4.3.2/bin。重新启动系统生效。以后默认编译器为4.3.2

### 1.2.3 重新编译uboot

重新编译Uboot

1、cd /home/chao/download/u-boot-2012.04.01/

2、make disclean 清除上次编译

3、make smdk2440\_config 配置

4、make 编译

5、下载uboot.bin 一般情况启动不了

## 1.3 uboot 初步分析

编译uboot，根据编译信息分析。

u-boot.lds 指定的是程序运行的起始地址。-Ttext指定的是链接地址，是程序最终运行的地址。

uboot编译时执行的最后的链接命令

arm-linux-ld –pie –T u-boot.lds –Bstatic –Text 0x0 $UNDEF\_SYM arch /arm/cpu/arm920t/start.o … 知道链接地址为0。该地址是在Include/configs/smdk2410.h 的#define CONFIG\_SYS\_TEXT\_BASE 0x0指定的。

结合u-boot.lds中指定的起始运行地址也为0，该版本编译出的代码能够直接在norflash上运行。但是不能在sdram中运行。

关于pie选项

shell 输入arm-linux-ld –help |grep pie 获得pie的帮助信息

加上pie生成位置无关的代码，不是真正的位置无关，是要生成各符号的地址，比如变量，函数名和标号等信息，以便在程序中修改这些信息使程序可以执行。

# 2 uboot 流程具体分析

启动时执行的第一个文件是start.s。

## 2.1 uboot 第一阶段流程分析

1> b start\_code

2> 设置cpu模式为svc模式。关看门狗，屏蔽中断，（设置时钟，原版没有在此设置）

（svc模式和sys模式区别，sys模式时操作系统运行模式，svc比sys拥有更多的访问硬件的权限）

3> bl cpu\_init\_crit 关闭cache，Mmu,初始化内存

4> 设置栈，

5> bl board\_init\_f /\*位于arch/arm/lib \*/

调用init\_sequence[]里面的函数board\_early\_init\_f 设置系统时钟，GPIO端口

和dram\_init（）

函数最后调用relocate\_code(addr\_sp, id, addr);重定位代码

6> 跳转到第二阶段代码

relocate\_code:

ldr r0, \_TEXT\_BASE /\* 源地址的起始地址r0 <- Text base \*/

sub r9, r6, r0 /\* 目的地址和源地址的偏移量 r9 <- relocation offset \*/

……

ldr r0, \_board\_init\_r\_ofs //起始地址偏移量

adr r1, \_start //起始地址

add lr, r0, r1 //在flash中的地址

add lr, lr, r9 //在内存中的地址

/\* setup parameters for board\_init\_r \*/

mov r0, r5 /\* gd\_t 传递的参数保存r0\*/

mov r1, r6 /\* dest\_addr 传递的参数r1，uboot代码复制到内存的目的地址\*/

/\* jump to it ... \*/

mov pc, lr //跳转到内存执行

\_board\_init\_r\_ofs:

.word board\_init\_r - \_start/\*第二阶段函数调用\*/

## 2.2 uboot 第二阶段流程分析

### 2.2.1 内核启动准备

1>board\_init\_r（）

board\_init();

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410;

gd->bd->bi\_boot\_params = 0x30000100;

flash\_init ();

nand\_init();

env\_relocate();/\* initialize environment \*/

2>main\_loop();

3> s = getenv ("bootcmd");

run\_command(s, 0);

### 2.2.2 启动内核

cmd\_bootm.c 搜索bootm,找到命令

U\_BOOT\_CMD(

bootm, CONFIG\_SYS\_MAXARGS, 1, do\_bootm,

启动函数为do\_bootm()里面有一个boot\_os[]数组，存放启动各种操作系统的函数。

对于Linux，是

do\_bootm\_linux()

boot\_jump\_linux(images);

kernel\_entry = (void (\*)(int, int, uint))images->ep;

kernel\_entry(0, machid, r2);

## 2.3 uboot内存分布图

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)/\*sp=30000f80\*/

gd = (gd\_t \*) ((CONFIG\_SYS\_INIT\_SP\_ADDR) & ~0x07);//8字节对齐

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

所以gd=30000f80

addr=CONFIG\_SDRAM\_BASE+gd->ram\_size //addr=0x3400,0000

addr-=(4096\*4)//16k保留给TLB,addr=0x33ffc000

addr&=~(0x10000-1)//低16位清零，64k对齐，addr=0x33ff0000

addr&=(4096-1) //4k对齐，addr=0x33ff0000

addr-=gd->mon\_len

//gd->mon\_len=\_bss\_end\_ofs;\_bss\_end\_ofs定义为\_bss\_end\_ \_start, 即Uboot的大小

addr&=~(4096-1) //4k对齐

//执行完毕addr指向的是uboot在内存中的地址

addr\_sp=addr-TOTAL\_MALLOC\_LEN//堆空间大小

addr\_sp -=sizeof(bd\_t)//留出bd\_t大小，单板的相关信息，如波特率，ip等

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

gd->bd = bd;//gd 中的bd指针指向此处bd

addr\_sp -=sizeof(gd\_t) //留出gd\_t大小

id=(gd\_t \*)addr\_sp //id指向预留空间，id就代表了gd

addr\_sp-=12; //留出12B空间

addr\_sp&=~0x07 // addr\_sp 8字节对齐

memcpy（id,(void\*)gd，sizeof(gd\_t)）； //把gd拷贝到id指向的地址

gd 的值在init\_sequence[]里面的函数设置的

relocate\_code(addr\_sp, id, addr);// 堆栈，id信息，uboot在内存中的地址

0x3400,0000

16k for TLB,64K对齐

uboot重定位在这里

大小为gd->mon\_len

并4k对齐

堆空间，大小为

TOTAL\_MALLOC\_LEN

板信息bd\_t 大小

通用信息gd\_t 大小

12B，8字节对齐

栈区

剩余空间

gd\_t大小

16122

addr,uboot内存地址

bd

id

addr\_sp

0x3000,0f80

0x30000,0000

## 2.4 重定位代码

relocate\_code(addr\_sp, id, addr);

参数依次对应r0,r1,r2

1> 从flash把代码复制到内存

2> 以前程序的链接地址是0，访问全局变量。静态变量，调用函数时使用基于0地址编译得到的地址，现在把程序复制到了sdram,需要修改代码，把基于0地址编译得到的地址改为新地址。

relocate\_code:

mov r4, r0 /\* save addr\_sp \*/

mov r5, r1 /\* save addr of gd \*/

mov r6, r2 /\* save addr of destination \*/

/\* Set up the stack 重新设置栈顶地址 \*/

stack\_setup:

mov sp, r4 //sp=addr\_sp

adr r0, \_start //程序在flash中运行，\_start地址为0，否则为内存地址

cmp r0, r6 //如果\_start=内存目的地址，程序已在内存不用拷贝

beq clear\_bss /\* 程序已在内存，跳过重定位\*/

mov r1, r6 /\* r1代码在内存中目的地址 \*/

ldr r3, \_bss\_start\_ofs // r3代码+数据段的长度

add r2, r0, r3 /\* r2源代码在flash中的结束地址 \*/

copy\_loop:

ldmia r0!, {r9-r10} /\* copy from source address [r0] \*/

stmia r1!, {r9-r10} /\* copy to target address [r1] \*/

cmp r0, r2

blo copy\_loop /\*如果r0<flash结束地址，继续拷贝代码 \*/

#ifndef CONFIG\_SPL\_BUILD

/\*

\* fix .rel.dyn relocations符号地址修改

\*/

ldr r0, \_TEXT\_BASE /\* 源地址的起始地址r0就是 Text base=0 \*/

sub r9, r6, r0 /\* 目的地址和源地址的偏移量，代码内存地址-flash起始地址目的地址0x33f31000\*/

/\* r10 <- sym table ofs \_dynsym\_start\_ofs= \_\_dynsym\_start - \_start

符号表的偏移值.00073610(韦东山编译出来是73608) \*/

ldr r10, \_dynsym\_start\_ofs

/\* r10 <- sym table in FLASH 地址在反汇编中找到

00073610 <\_\_dynsym\_start>:\_\_dynsym\_start 表地址\*/

add r10, r10, r0

/\* r2 <- rel dyn start ofs .word \_\_rel\_dyn\_start - \_start

000001a0 <\_rel\_dyn\_start\_ofs>:

1a0: 0006b570 .word 0x0006b570

r2里面存储的是需要修改地址单元内容的地址，这里面的地址内部存储的是需要修改的符号地址，。修改方法：

先从r2指向的内容取出地址addr，addr=addr+0x33f41000得到内存地址。从内存地址addr中取出需要修改的符号地址然后sym\_addr=sym\_addr+0x33f41000。然后将

sym\_addr再存回addr中。

取出

\*/

ldr r2, \_rel\_dyn\_start\_ofs

add r2, r2, r0 /\* r2<- rel dyn start in FLASH \*/

ldr r3, \_rel\_dyn\_end\_ofs /\* r3 <- rel dyn end ofs \*/

add r3, r3, r0 /\* r3 <- rel dyn end in FLASH \*/

//开始修改地址

fixloop:

ldr r0, [r2]

/\* 将r2内存中的地址读到r0中，反汇编存储的为 0x00000020,而0x00000020中存储 \_undefined\_instruction 地址r0 <- location to fix up, IN FLASH! \*/

add r0, r0, r9

/\* 修改地址，地址+内存偏移0x00000020+0x33f41000

r0=0x33f41020 r0 <- location to fix up in RAM \*/

ldr r1, [r2, #4] /\*r2+4 \*/

and r7, r1, #0xff /\* r7=r1&0xff=0x17=23\*/

cmp r7, #23 /\* relative fixup? \*/

beq fixrel //相等.修改 \_rel\_dyn否则修改fixabs

cmp r7, #2 /\* absolute fixup? \*/

beq fixabs

/\* ignore unknown type of fixup \*/

b fixnext

fixabs:

/\* absolute fix: set location to (offset) symbol value \*/

mov r1, r1, LSR #4 /\* r1 <- symbol index in .dynsym \*/

add r1, r10, r1 /\* r1 <- address of symbol in table \*/

ldr r1, [r1, #4] /\* r1 <- symbol value \*/

add r1, r1, r9 /\* r1 <- relocated sym addr \*/

b fixnext

fixrel:

/\* relative fix: increase location by offset \*/

/\*r0 = 0x33f41020 和flash上0x20地址处值一样。看反汇编0x20内容

000001e0 \*/

ldr r1, [r0]

/\*r9 r1=r1+r9=0x1e0+0x33f41000=0x33f41e0

add r1, r1, r9

fixnext:

str r1, [r0] //成功修改内存中的值

add r2, r2, #8 /\* each rel.dyn entry is 8 bytes \*/

cmp r2, r3 // r3 <- rel dyn end in FLASH ，是否已经修改完毕，未修改完继续

blo fixloop //r2<r3继续修改

#endif

clear\_bss:

#ifndef CONFIG\_SPL\_BUILD //执行

ldr r0, \_bss\_start\_ofs//bss段起始位置

ldr r1, \_bss\_end\_ofs//bss段大小

mov r4, r6 /\* reloc addr \*/

add r0, r0, r4

add r1, r1, r4

mov r2, #0x00000000 /\* clear \*/

clbss\_l:str r2, [r0] /\* clear loop... \*/

add r0, r0, #4

cmp r0, r1

bne clbss\_l //不等继续清零

假设addr\_sp地址为0x3300,0000

addr=0x33f410000,

addr地址如何计算：反汇编

make smdk2410\_config

make //生成u-boot.bin在根目录下

arm-linux-objdump –D uboot >u-boot.dis

addr-=gd->mon\_len ,而gd->mon\_len=\_bss\_end\_ofs;

根据反汇编文件找到\_bss\_end\_ofs ，存储长度值为 000ae4e0

所以addr=0x33ff0000-0xae4e0=0x33f41b20

4k对齐addr&=~(4096-1) 得到 addr=0x33f41000

修改地址

假设原来的数据地址为0x100,那么拷贝到内存后，数据存放的addr+0x100=0x33f41100。

# 3 移植uboot

## 3.1 uboot-2440 初步编译

### 3.1.1 新建smdk2440板文件

1> cd board/samsung/

2> cp smdk2410 smdk2440 –rf //递归拷贝

### 3.1.2 新建配置文件

1> cd include/configs/

2> cp smdk2410.h smdk2440.h

### 3.1.3 添加编译smdk2440选项

#### grep “smdk2410” \* -nR //显示uboot源码中所有含”smdk2410” 的行

在boards.cfg 中找到smdk2410 在65行。vim boards.cfg 输入esc:65 定位到65行。将smdk2410所在行复制一份到下一行，将smdk2410 改为smdk2440。

smdk2410 arm arm920t - samsung s3c24x0

smdk2440 arm arm920t - samsung s3c24x0

### 3.1.4 编译smdk2440

1>make smdk2440\_config

2> make

添加单板成功，运行肯定不会成功，需要修改程序

## 3.2 分析并修改代码

在window上用source insight 修改代码

1. board/samsung/ smdk2410文件复制一份为smdk2440
2. include/configs/smdk2410.h 复制一份为smdk2440.h
3. 在source insight 中移除原来smdk2410文件，添加smdk2440文件

从start.s 开始查看代码修改程序

### 3.2.1 分析时钟和内存

时钟设置函数：

board\_init\_f 🡪init\_sequence[]函数数组🡪board\_early\_init\_f（）

writel((M\_MDIV << 12) + (M\_PDIV << 4) + M\_SDIV,&clk\_power->mpllcon);

内存设置函数：

cpu\_init\_crit🡪lowlevel\_init(smdk2440文件夹下)

原版Uboot先执行cpu\_init\_crit设置内存控制器，然后再设置时钟。board\_init\_f()是c函数，需要用到内存。这样时钟设置前的一段时间里，时钟内存频率不匹配。也可能导致问题。

uboot里先以60MHZ时钟计算参数来设置内存控制器，但是mpll还未设置，此时时钟还是12MHZ。

处理措施：把MPLL的设置放到start.s里面，取消board\_early\_ini\_f里面对mpll的设置。

### 3.2.2 设置时钟

1>注释掉board\_init\_f 🡪init\_sequence[]函数数组🡪board\_early\_init\_f（）里面的代码

writel(0xFFFFFF, &clk\_power->locktime);

writel((M\_MDIV << 12) + (M\_PDIV << 4) + M\_SDIV,&clk\_power->mpllcon);

2> 在start.s中注释掉以下代码

/\* FCLK:HCLK:PCLK = 1:2:4 \*/

/\* default FCLK is 120 MHz ! \*/

//ldr r0, =CLKDIVN

//mov r1, #3

//str r1, [r0]

添加如下代码

/\* 2. 设置时钟 分频系数\*/

ldr r0, =0x4c000014

mov r1, #0x05; // FCLK:HCLK:PCLK=1:4:8

str r1, [r0]

/\* 如果HDIVN非0，CPU的总线模式应该从“fast bus mode”变为“asynchronous bus mode” \*/

mrc p15, 0, r1, c1, c0, 0 /\* 读出控制寄存器 \*/

orr r1, r1, #0xc0000000 /\* 设置为“asynchronous bus mode” \*/

mcr p15, 0, r1, c1, c0, 0 /\* 写入控制寄存器 \*/

#define S3C2440\_MPLL\_400MHZ ((0x5c<<12)|(0x01<<4)|(0x01))

/\* MPLLCON = S3C2440\_MPLL\_200MHZ \*/

ldr r0, =0x4c000004

ldr r1, =S3C2440\_MPLL\_400MHZ

str r1, [r0]

/\* 启动ICACHE \*/

mrc p15, 0, r0, c1, c0, 0 @ read control reg

orr r0, r0, #(1<<12)

mcr p15, 0, r0, c1, c0, 0 @ write it back

### 3.2.3 修改内存

在lowlevel\_init.s文件里面修改内存配置

注释掉如下代码：

.word (0+(B1\_BWSCON<<4)+(B2\_BWSCON<<8)+(B3\_BWSCON<<12)+(B4\_BWSCON<<16)+(B5\_BWSCON<<20)+(B6\_BWSCON<<24)+(B7\_BWSCON<<28))

.word ((B0\_Tacs<<13)+(B0\_Tcos<<11)+(B0\_Tacc<<8)+(B0\_Tcoh<<6)+(B0\_Tah<<4)+(B0\_Tacp<<2)+(B0\_PMC))

.word ((B1\_Tacs<<13)+(B1\_Tcos<<11)+(B1\_Tacc<<8)+(B1\_Tcoh<<6)+(B1\_Tah<<4)+(B1\_Tacp<<2)+(B1\_PMC))

.word ((B2\_Tacs<<13)+(B2\_Tcos<<11)+(B2\_Tacc<<8)+(B2\_Tcoh<<6)+(B2\_Tah<<4)+(B2\_Tacp<<2)+(B2\_PMC))

.word ((B3\_Tacs<<13)+(B3\_Tcos<<11)+(B3\_Tacc<<8)+(B3\_Tcoh<<6)+(B3\_Tah<<4)+(B3\_Tacp<<2)+(B3\_PMC))

.word ((B4\_Tacs<<13)+(B4\_Tcos<<11)+(B4\_Tacc<<8)+(B4\_Tcoh<<6)+(B4\_Tah<<4)+(B4\_Tacp<<2)+(B4\_PMC))

.word ((B5\_Tacs<<13)+(B5\_Tcos<<11)+(B5\_Tacc<<8)+(B5\_Tcoh<<6)+(B5\_Tah<<4)+(B5\_Tacp<<2)+(B5\_PMC))

.word ((B6\_MT<<15)+(B6\_Trcd<<2)+(B6\_SCAN))

.word ((B7\_MT<<15)+(B7\_Trcd<<2)+(B7\_SCAN))

.word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)

.word 0x32

.word 0x30

.word 0x30

添加自己的内存设置：

SMRDATA:

.long 0x22011110 //BWSCON

.long 0x00000700 //BANKCON0

.long 0x00000700 //BANKCON1

.long 0x00000700 //BANKCON2

.long 0x00000700 //BANKCON3

.long 0x00000740 //BANKCON4

.long 0x00000700 //BANKCON5

.long 0x00018005 //BANKCON6

.long 0x00018005 //BANKCON7

.long 0x008C04F4 // REFRESH

.long 0x000000B1 //BANKSIZE

.long 0x00000030 //MRSRB6

.long 0x00000030 //MRSRB7

设置完毕下载u-boot.bin到开发板， SDRAM已经可以使用。启动发现串口有输出，有乱码。修改波特率。

### 3.2.4修改串口

uboot串口支持Ymodem协议。

board\_init\_f 🡪init\_sequence[]函数数组🡪 init\_baudrate（）函数设置了波特率是115200。波特率设置正确。

serial\_init（）🡪 serial\_init\_dev(UART\_NR);🡪 \_serial\_setbrg(dev\_index);

🡪 reg = get\_PCLK() / (16 \* gd->baudrate) - 1;🡪 get\_PCLK(void)🡪 get\_HCLK()

发现s3c2440和s3c2410中时钟操作不一样。默认使用的是s3c2410配置，要使用

s3c2440配置。需要定义宏：CONFIG\_S3C2440。

在include/configs/smdk2440.h里面

注释掉#define CONFIG\_S3C2410 /\* specifically a SAMSUNG S3C2410 SoC \*/

添加 #define CONFIG\_S3C2440

串口提示符修改为2440

将#define CONFIG\_SYS\_PROMPT "SMDK2410 # " 改为

#define CONFIG\_SYS\_PROMPT "SMDK2440 # "

重新编译uboot发现错误，错误原因是nand错误。暂且去掉对nand的支持。发现只有定义CONFIG\_NAND\_S3C2410才会编译Nand相关程序。（drivers/nand/Makefile）修改smdk2440.h 去掉#define CONFIG\_CMD\_NAND,

再次编译发现yaffscfg.c:undefined reference to “nand\_info”错误。解决方法：  
屏蔽掉yaffs。include/configs/smdk2440.h:227行注释掉

#define CONFIG\_YAFFS2

重新编译Uboot，下载到板子，串口输出正确，发现新问题：WARNING:CACHES not enableed,flash failed\*\*

### 3.2.5 修改启动代码支持Nand启动，清bss，设置栈,设置机器ID

原来的代码在链接时加了 –pie选项，使得uboot.bin里面多了\*（.rel）\*(.dymsym)等信息，不仅使得程序变大不利于从Nand启动，而且导致程序复杂。不如修改为老的

重定位方法。

参考毕业班第一课的start.s,init.c修改代码。

init.c放入board/samsung/smdk2440目录

修改board/samsung/smdk2440目录下的Makefile 在COBJS := smdk2410.o 后添加init.o。最终为COBJS := smdk2410.o init.o

init.c代码为自己编写，主要是nandflash读写和bss清除

/\* NAND FLASH控制器 \*/

#define NFCONF (\*((volatile unsigned long \*)0x4E000000))

#define NFCONT (\*((volatile unsigned long \*)0x4E000004))

#define NFCMMD (\*((volatile unsigned char \*)0x4E000008))

#define NFADDR (\*((volatile unsigned char \*)0x4E00000C))

#define NFDATA (\*((volatile unsigned char \*)0x4E000010))

#define NFSTAT (\*((volatile unsigned char \*)0x4E000020))

/\* GPIO \*/

#define GPHCON (\*(volatile unsigned long \*)0x56000070)

#define GPHUP (\*(volatile unsigned long \*)0x56000078)

/\* UART registers\*/

#define ULCON0 (\*(volatile unsigned long \*)0x50000000)

#define UCON0 (\*(volatile unsigned long \*)0x50000004)

#define UFCON0 (\*(volatile unsigned long \*)0x50000008)

#define UMCON0 (\*(volatile unsigned long \*)0x5000000c)

#define UTRSTAT0 (\*(volatile unsigned long \*)0x50000010)

#define UTXH0 (\*(volatile unsigned char \*)0x50000020)

#define URXH0 (\*(volatile unsigned char \*)0x50000024)

#define UBRDIV0 (\*(volatile unsigned long \*)0x50000028)

#define TXD0READY (1<<2)

void nand\_read\_ll(unsigned int addr, unsigned char \*buf, unsigned int len);

static int isBootFromNorFlash(void)

{

volatile int \*p = (volatile int \*)0;

int val;

val = \*p;

\*p = 0x12345678;

if (\*p == 0x12345678)

{

/\* 写成功, 是nand启动 \*/

\*p = val;

return 0;

}

else

{

/\* NOR不能像内存一样写 \*/

return 1;

}

}

void copy\_code\_to\_sdram(unsigned char \*src, unsigned char \*dest, unsigned int len)

{

int i = 0;

/\* 如果是NOR启动 \*/

if (isBootFromNorFlash())

{

while (i < len)

{

dest[i] = src[i];

i++;

}

}

else

{

//nand\_init();

nand\_read\_ll((unsigned int)src, dest, len);

}

}

void clear\_bss(void)

{

/\*\_\_bss\_start，\_\_bss\_end\_\_来源于uboot.lds文件，具体是使用\_\_bss\_end\_\_还是

\_\_bss\_end，查看uboot.lds确定

\*/

extern int \_\_bss\_start, \_\_bss\_end\_\_;

int \*p = &\_\_bss\_start;

for (; p < &\_\_bss\_end\_\_; p++)

\*p = 0;

}

void nand\_init\_ll(void)

{

#define TACLS 0

#define TWRPH0 1

#define TWRPH1 0

/\* 设置时序 \*/

NFCONF = (TACLS<<12)|(TWRPH0<<8)|(TWRPH1<<4);

/\* 使能NAND Flash控制器, 初始化ECC, 禁止片选 \*/

NFCONT = (1<<4)|(1<<1)|(1<<0);

}

static void nand\_select(void)

{

NFCONT &= ~(1<<1);

}

static void nand\_deselect(void)

{

NFCONT |= (1<<1);

}

static void nand\_cmd(unsigned char cmd)

{

volatile int i;

NFCMMD = cmd;

for (i = 0; i < 10; i++);

}

static void nand\_addr(unsigned int addr)

{

unsigned int col = addr % 2048;

unsigned int page = addr / 2048;

volatile int i;

NFADDR = col & 0xff;

for (i = 0; i < 10; i++);

NFADDR = (col >> 8) & 0xff;

for (i = 0; i < 10; i++);

NFADDR = page & 0xff;

for (i = 0; i < 10; i++);

NFADDR = (page >> 8) & 0xff;

for (i = 0; i < 10; i++);

NFADDR = (page >> 16) & 0xff;

for (i = 0; i < 10; i++);

}

static void nand\_wait\_ready(void)

{

while (!(NFSTAT & 1));

}

static unsigned char nand\_data(void)

{

return NFDATA;

}

void nand\_read\_ll(unsigned int addr, unsigned char \*buf, unsigned int len)

{

int col = addr % 2048;

int i = 0;

/\* 1. 选中 \*/

nand\_select();

while (i < len)

{

/\* 2. 发出读命令00h \*/

nand\_cmd(0x00);

/\* 3. 发出地址(分5步发出) \*/

nand\_addr(addr);

/\* 4. 发出读命令30h \*/

nand\_cmd(0x30);

/\* 5. 判断状态 \*/

nand\_wait\_ready();

/\* 6. 读数据 \*/

for (; (col < 2048) && (i < len); col++)

{

buf[i] = nand\_data();

i++;

addr++;

}

col = 0;

}

/\* 7. 取消选中 \*/

nand\_deselect();

}

在include/configs/smdk244o0.h 中 将#define CONFIG\_SYS\_TEXT\_BASE 0x0

修改为#define CONFIG\_SYS\_TEXT\_BASE 0x33f00000

修改board\_init\_f()函数最后一句relocate\_code（）注释掉，因为前面已经重定位过了。

修改start.s中代码

去掉重定位代码部分和清bss段部分。去掉relocate\_code:到cpu\_init\_crit:所有代码。并且重新设置栈。

原来的代码

call\_board\_init\_f:

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)/\*sp=30000f80\*/

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

ldr r0,=0x00000000

bl board\_init\_f /\*位于arch/arm/lib \*/

现在的代码,从自己写bootloader程序更改的

//添加保存栈地址代码

.globl base\_sp //这样就定义了全局变量，所有文件都可以使用

base\_sp:

.long 0

/\*

\* the actual start code

\*/

start\_code:

…..

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR) //初始化函数使用的栈，以后需要重新设定

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

bl nand\_init\_ll //nand初始化

mov r0, #0

/\*\_TEXT\_BASE存储地址确定，自己写bootloader代码使用的是ldr r1,=\_start,此处不能用

的原因是\_start存储地址可能超过4k\*/

ldr r1, \_TEXT\_BASE

ldr r2, \_bss\_start\_ofs

bl copy\_code\_to\_sdram //拷贝uboot代码

bl clear\_bss

ldr pc, =call\_board\_init\_f /\*跳到sdram执行call\_board\_init\_f编译出来的地址为内存地址。因此只需要将该地址加载到pc，就会跳转到内存执行\*/

/\* Set stackpointer in internal RAM to call board\_init\_f \*/

call\_board\_init\_f:

ldr r0,=0x00000000

bl board\_init\_f

/\* unsigned int的值存在r0里, 正好给board\_init\_r \*/

ldr r1, \_TEXT\_BASE

ldr sp, base\_sp //设置第二阶段代码使用的栈

/\* 调用第2阶段的代码 \*/

bl board\_init\_r

在arm/lib/board.c/board\_init\_f()中修改

开始处添加extern ulong base\_sp;声明为外部变量

返回语句之前添加

base\_sp = addr\_sp;

第二阶段代码void board\_init\_r(gd\_t \*id, ulong dest\_addr)需要id的值和Uboot在内存中的起始地址。\_TEXT\_BASE就是目的地址。执行代码ldr r1, \_TEXT\_BASE，将目的地址存入r1。

对于Id, 可以修改void board\_init\_f()函数，使其返回Id的值，这样函数返回时,id的值就存入了r0寄存器中。

修改voId board\_init\_f(gd\_t \*id, ulong dest\_addr)函数为unsigned int board\_init\_r(gd\_t \*id, ulong dest\_addr),在函数末尾添加 return （unsigned int）id;

修改include/common.h 第276行函数原型为unsigned int board\_init\_f (ulong);

定死链接地址，将board\_init\_f()函数中的372，373行

addr-=gd->mon\_len;

addr&=~(4096-1)

改为addr = CONFIG\_SYS\_TEXT\_BASE;

去掉 –pie选项

grep “\-pie” \* -nR搜索，（arch/arm/config.mk:75 LDFLAGS\_u-boot += -pie ,用#号注释掉改行）

编译生成uboot.lds,修改链接脚本uboot.lds如下

修改链接脚本

把start.s，init.c,lowlevel.s等文件放在最前面

ENTRY(\_start)

SECTIONS

{

= 0x00000000;

. = ALIGN(4);

.text :

{

\_\_image\_copy\_start = .;

CPUDIR/start.o (.text)

/\*

把start.s,init.c,lowlevel.s等文件放在最前面。board目录下文件会生成一个总的.o文件

libsmdk2440.o，把libsmdk2440.o放在最前面即可

\*/

board/samsung/smdk2440/libsmdk2440.o (.text)

\*(.text)

}

重新编译，烧写Uboot.bin到Nand flash

usb 1 30000000

nand erase 0 80000

nand write 30000000 0 80000

,修改board\_init\_r()函数board\_init()

gd->bd->bi\_arch\_number = MACH\_TYPE\_SMDK2410;改为gd->bd->bi\_arch\_number = MACH\_TYPE\_S3C2440;

### 3.2.6 修改支持Nor Flash

出现错误

Flash：\*\*failed \*\*\*\*

### ERROR ### Please RESET the board###

搜索Flash： 找到Board.c (arch\arm\lib): puts("Flash: ");

puts("Flash: ");

flash\_size = flash\_init();

if (flash\_size > 0) { //证明返回值不大于0

print\_size(flash\_size, "\n");

} else {

puts(failed);

hang(); //停止

}

void hang(void)

{

puts("### ERROR ### Please RESET the board ###\n");

for (;;);

}

因为nand启动时是没有nor flash的。需要修改程序使得如果uboot从nand启动的话可以跳过nor flash 错误继续执行。

修改程序使得如果uboot从nand启动的话可以跳过nor flash 错误提示继续执行。

puts(failed);

hang(); //停止

改为puts(“0 kB”)。

由于在nor flash启动时也会出现该错误，证明uboot不能识别nor flash，需要修改程序识别nor flash。（cfi\_flash公共闪存接口flash）

flash\_init()

//如果使用flash\_detect\_legacy检测falsh,不成功使用flash\_get\_size

🡪if (!flash\_detect\_legacy(cfi\_flash\_bank\_addr(i), i))

flash\_get\_size(cfi\_flash\_bank\_addr(i), i);

搜索cfi\_flash\_bank\_addr

Cfi\_flash.c (drivers\mtd):phys\_addr\_t cfi\_flash\_bank\_addr(int i)

flash 配置的基地址为CONFIG\_SYS\_FLASH\_BASE，没有问题。

为了输出调试信息，定义两个宏（cfi\_flash.c）

//#define DEBUG 1

//#define \_DEBUG 1

烧写到Nor flash测试，不再提示卡死，但是flash大小依然为0。

调试信息JEDEC PROBE:ID C2 2249 0证明可以检测到flash.

flash\_detect\_legacy()中有两个函数flash\_read\_jedec\_ids();和jedec\_flash\_match()。根据输出

知道读Id没有错误，是jz2440板子flash不能与已知flash匹配。

flash\_read\_jedec\_ids(info);

debug("JEDEC PROBE: ID %x %x %x\n",

info->manufacturer\_id,

info->device\_id,

info->device\_id2);

if (jedec\_flash\_match(info, info->start[0]))

break;

else

nmap\_physmem((void \*)info->start[0],

在匹配表中添加flash

jedec\_flash\_match(flash\_info\_t \*info, ulong base)

匹配表jedec\_table[]的CONFIG\_SYS\_FLASH\_LEGACY\_512Kx16区添加，

//jz2440v2使用的是mx29lv60d 1M\*16bit flash

{

.mfr\_id = (u16)MX\_MANUFACT, //厂家ID

.dev\_id = MX29LV160DB,

.name = "MXIC MX29LV160DB",

.uaddr = {

[1] = MTD\_UADDR\_0x0555\_0x02AA /\* 解锁命令 \*/

},

.DevSize = SIZE\_2MiB,

.CmdSet = P\_ID\_AMD\_STD,

.NumEraseRegions= 4, //有几种大小不同的擦除块

.regions = {//分区大小信息

ERASEINFO(8\*2\*1024, 1), //8kwords=8\*2\*1024B

ERASEINFO(4\*2\*1024, 2),

ERASEINFO(16\*2\*1024, 1),

ERASEINFO(32\*2\*1024, 31),

}

}

然后再include/configs/smdk2440.h中定义宏ONFIG\_SYS\_FLASH\_LEGACY\_512Kx16

定义位置：

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

\* FLASH and environment organization

\*/

#define CONFIG\_SYS\_FLASH\_CFI

#define CONFIG\_FLASH\_CFI\_DRIVER

#define CONFIG\_FLASH\_CFI\_LEGACY

#define CONFIG\_SYS\_FLASH\_LEGACY\_512Kx16

下载程序到nor flash.提示too many flash sectors 错误。搜索提示找到报错原因。

CONFIG\_SYS\_MAX\_FLASH\_SECT 改为128；（原来值19，我们的flash有36个块）

测试Nor flash

flinfo命令，显示flash详细信息

erase 80000 8ffff //擦除 80000到8ffff之间Flash

mw.b 30000000 12345678//写一字节数据到sdram

cp.b 30000000 80000 10000 //从sdram 30000000拷贝到80000,拷贝长度10000

md.b 80000 //从80000读一个字节

### 3.2.7 修改代码支持nand flash

修改 include/configs/smdk2440.h 开启CONGIG\_CMD\_NAND 宏定义以支持nand读写操作。

#if defined(CONFIG\_CMD\_NAND) //需要CONFIG\_CMD\_NAND才会支持nand

puts("NAND: ");

nand\_init(); /\* go init the NAND \*/

#endif

编译提示错误：error:dereferencing pointer to incmoplete tpye。原因在于我们定义的是CONFIG\_S3C2440,宏开关使用s3c2440\_nand。而代码s3c2410\_hwcontrol（）里面使用的是s3c2410\_nand。而s3c2410\_nand没有启用，所有s3c2410\_nand未定义。需要修改代码为

s3c2440\_nand.

把drivers/mtd/nand/s3c2410\_nand.c 复制成s3c2440\_nand.c。

修改smdk2440.h配置文件

/\*

\* NAND configuration

\*/

#ifdef CONFIG\_CMD\_NAND

#ifdef ConFIG\_S3C2410

#define CONFIG\_NAND\_S3C2410

#define CONFIG\_SYS\_S3C2410\_NAND\_HWECC

#else

#define CONFIG\_NAND\_S3C2440

#define CONFIG\_SYS\_S3C2440\_NAND\_HWECC

#endif

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

#define CONFIG\_SYS\_NAND\_BASE 0x4E000000

#endif

修改drivers/mtd/nand/Makefile 在CONFIG\_NAND\_S3C2410下面添加支持s3c2440\_nand。

COBJS-$(CONFIG\_NAND\_S3C2440) += s3c2440\_nand.o

将s3c2440\_nand.c中所有的s3c2410改为s3c2440。

nand 驱动被分为两层 nand 协议层和nand硬件相关层。nand硬件相关层需要根据芯片修改。主要功能：选中/取消片；选发命令/地址；R/W数据；判断状态是否忙

nand\_init(void)🡪 nand\_init\_chip(i); 🡪board\_nand\_init(nand) (s3c2440\_nand.c中)

修改时序参数我们计算的值：

tacls=0;twrph0=0;twrph1=0;

注释掉该函数原有的操作配置寄存器函数。改为如下

/\* 初始化时序 \*/

cfg = ((tacls-1)<<12)|((twrph0-1)<<8)|((twrph1-1)<<4);

writel(cfg, &nand\_reg->nfconf);

/\* 使能NAND Flash控制器, 初始化ECC, 禁止片选 \*/

writel((1<<4)|(1<<1)|(1<<0), &nand\_reg->nfcont);

修改s3c2440\_hwcontrol（）如下

由于s3c2410和s3c2440寄存器不一样

/\* ctrl : 表示做什么, 选中芯片/取消选中, 发命令还是发地址

\*

\* dat : 命令值或地址值

\*/

static void s3c2440\_hwcontrol(struct mtd\_info \*mtd, int dat, unsigned int ctrl)

{

struct s3c2440\_nand \*nand = s3c2440\_get\_base\_nand();

if (ctrl & NAND\_CLE)

{

/\* 发命令 \*/

writeb(dat, &nand->nfcmd);

}

else if(ctrl & NAND\_ALE)

{

/\* 发地址 \*/

writeb(dat, &nand->nfaddr);

}

}

ECC相关

nand 里面结构是每一个page后面跟一个oob；由于nand 容易发生页翻转，oob就是用来校验Nand里面的数据的。

写page,生成ECC,写ECC🡪oob 读page,读ECC，page数据生成ECC,比较两个ECC,相同证明数据正确。board\_nand\_init(struct nand\_chip \*nand)做如下修改，使用软件ECC。

#ifdef CONFIG\_s3C2410\_NAND\_HWECC //未定义该宏

nand->ecc.hwctl = s3c2440\_nand\_enable\_hwecc;

nand->ecc.calculate = s3c2440\_nand\_calculate\_ecc;

nand->ecc.correct = s3c2440\_nand\_correct\_data;

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

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

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

#else

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

#endif

分析nand 过程

nand\_init()

nand\_init\_chip()

board\_nand\_init() //设定nand操作函数，初始化硬件寄存器

nand\_scan() //扫描nand,里面有基本的Nand操作流程

nand\_scan\_ident()

nand\_set\_defaults(chip, busw);//设定缺省nand操作函数

chip->select\_chip = nand\_select\_chip;//不合适需要修改

chip->cmdfunc = nand\_command;

chIp->cmd\_ctrl()=s3c2440\_hwcontrol;

nand\_get\_flash\_type()//获得芯片类型

chip->select\_chip(mtd, 0);//芯片片选函数

chip->cmdfunc(mtd, NAND\_CMD\_RESET, -1, -1);//复位命令

chip->cmdfunc(mtd, NAND\_CMD\_READID, 0x00, -1);//读ID

/\* Read manufacturer and device IDs

busw = chip->options & NAND\_BUSWIDTH\_16;

chip->options没有设置，为0

chip->read\_byte = busw ? nand\_read\_byte16 : nand\_read\_byte;\*/

\*maf\_id = chip->read\_byte(mtd);

\*dev\_id = chip->read\_byte(mtd);

nand\_scan\_tail()

发现chip->select\_chip函数需要重新设置

在board\_nand\_init(struct nand\_chip \*nand)函数中将

nand->select\_chip = NULL;

修改为：

nand->select\_chip = s3c2440\_nand\_select;

编写s3c2440\_nand\_select()函数如下：

static void s3c2440\_nand\_select(struct mtd\_info \*mtd, int chipnr)

{

struct s3c2440\_nand \*nand = s3c2440\_get\_base\_nand();

switch (chipnr) {

case -1: /\* 取消选中 \*/

nand->nfcont |= (1<<1);

break;

case 0: /\* 选中 \*/

nand->nfcont &= ~(1<<1);

break;

default:

BUG();

}

}

chIp->cmd\_ctrl()就是 nand->cmd\_ctrl = s3c2440\_hwcontrol;

### 3.2.8 设置支持网卡DM9000

drivers/net/Makefile 查到只有配置CONFIG\_DRIVER\_DM9000才能将DM9000编译进Uboot。

修改smdk2440.h，去掉对CS8900的支持，改为支持DM9000

#if 0

#define CONFIG\_CS8900 /\* we have a CS8900 on-board \*/

#define CONFIG\_CS8900\_BASE 0x19000300

#define CONFIG\_CS8900\_BUS16 /\* the Linux driver does accesses as shorts \*/

#else

#define CONFIG\_DRIVER\_DM9000

#define CONFIG\_DM9000\_BASE 0x20000000

#define DM9000\_IO CONFIG\_DM9000\_BASE

#define DM9000\_DATA (CONFIG\_DM9000\_BASE + 4)

#endif

对DM9000使用类似于SDRAM

需要基地址：DM9000\_BASE.地址是由DM9000挂接在哪个bank决定的，该板子挂接在ncs4上，所以地址在0x20000000---0x28000000;

DM9000\_IO地址和DM9000\_DATA地址由LADDR2所接的CMD确定，需要看原理图。

LADDR2=0 时 发出IO地址，即DM9000\_BASE 地址

LADDR2=1时发出DATA地址，DM9000\_BASE+4;前面4个地址为配置寄存器，后面为数据空间

bit 2 1 0

H L L

4 0 0

设置内存控制器（位宽），设置时序参数，根据第二期驱动方法计算。

start.s🡪cpu\_init\_crit（low\_level\_init.s）中设置

.long 0x00000740 //BANKCON4

board\_init\_r()中eth\_initialize()进行网卡初始化，

eth\_initialize(gd->bd);

(board\_eth\_init(bis)

cs8900\_initialize()

#ifdef CONFIG\_CS8900

rc = cs8900\_initialize(0, CONFIG\_CS8900\_BASE);

#endif

添加

#ifdef CONFIG\_DRIVER\_DM9000

rc=dm9000\_initialize(bis);//可以搜索其他文件中如何使dm9000\_initialize函数

#endif

bis=gd->bd, 包含了板子初始化信息

设置板子的Ip

下载编译好的uboot

set ipaddr=192.168.0.112

set ethaddr=00:0c:29:7c:11:6d

set gatewayip=192.168.0.1

set serverip=192.168.0.111

此处不能执行save保存，因为还没有设置环境变量的保存地址。

ping 192.168.0.111 (虚拟机Ip) 显示host 192.168.0.111 is alive

此处板子ping通虚拟机，虚拟机ping不通板子的原因在于uboot不会回复主机。所以pIng不通正常的。

### 3.2.9 设置环境变量

重新编译下载uboot到nand，显示错误信息：

“bad CRC,using default envionment”

这是因为没有设置环境变量。

uboot对环境变量的处理方式：

刚开始启动时读环境变量，然后判断变量是否有效，如果无效就使用默认的参数。

搜索using default envionment；默认的环境变量在Env\_common.c中set\_default\_env(const char \*s)函数中设置。环境变量保存在default\_environment[]结构体中。

const uchar default\_environment[] = {

#ifdef CONFIG\_BOOTARGS

"bootargs=" CONFIG\_BOOTARGS "\0"

#endif

#ifdef CONFIG\_BOOTCOMMAND

"bootcmd=" CONFIG\_BOOTCOMMAND "\0"

#endif

#ifdef CONFIG\_RAMBOOTCOMMAND

"ramboot=" CONFIG\_RAMBOOTCOMMAND "\0"

#endif

#ifdef CONFIG\_NFSBOOTCOMMAND

"nfsboot=" CONFIG\_NFSBOOTCOMMAND "\0"

#endif

#if defined(CONFIG\_BOOTDELAY) && (CONFIG\_BOOTDELAY >= 0)

"bootdelay=" MK\_STR(CONFIG\_BOOTDELAY) "\0"

#endif

#if defined(CONFIG\_BAUDRATE) && (CONFIG\_BAUDRATE >= 0)

"baudrate=" MK\_STR(CONFIG\_BAUDRATE) "\0"

#endif

#ifdef CONFIG\_LOADS\_ECHO

"loads\_echo=" MK\_STR(CONFIG\_LOADS\_ECHO) "\0"

#endif

#ifdef CONFIG\_ETHADDR

"ethaddr=" MK\_STR(CONFIG\_ETHADDR) "\0"

#endif

#ifdef CONFIG\_ETH1ADDR

"eth1addr=" MK\_STR(CONFIG\_ETH1ADDR) "\0"

#endif

#ifdef CONFIG\_ETH2ADDR

"eth2addr=" MK\_STR(CONFIG\_ETH2ADDR) "\0"

#endif

#ifdef CONFIG\_ETH3ADDR

"eth3addr=" MK\_STR(CONFIG\_ETH3ADDR) "\0"

#endif

#ifdef CONFIG\_ETH4ADDR

"eth4addr=" MK\_STR(CONFIG\_ETH4ADDR) "\0"

#endif

#ifdef CONFIG\_ETH5ADDR

"eth5addr=" MK\_STR(CONFIG\_ETH5ADDR) "\0"

#endif

#ifdef CONFIG\_IPADDR

"ipaddr=" MK\_STR(CONFIG\_IPADDR) "\0"

#endif

#ifdef CONFIG\_SERVERIP

"serverip=" MK\_STR(CONFIG\_SERVERIP) "\0"

#endif

#ifdef CONFIG\_SYS\_AUTOLOAD

"autoload=" CONFIG\_SYS\_AUTOLOAD "\0"

#endif

#ifdef CONFIG\_PREBOOT

"preboot=" CONFIG\_PREBOOT "\0"

#endif

#ifdef CONFIG\_ROOTPATH

"rootpath=" CONFIG\_ROOTPATH "\0"

#endif

#ifdef CONFIG\_GATEWAYIP

"gatewayip=" MK\_STR(CONFIG\_GATEWAYIP) "\0"

#endif

#ifdef CONFIG\_NETMASK

"netmask=" MK\_STR(CONFIG\_NETMASK) "\0"

#endif

#ifdef CONFIG\_HOSTNAME

"hostname=" MK\_STR(CONFIG\_HOSTNAME) "\0"

#endif

#ifdef CONFIG\_BOOTFILE

"bootfile=" CONFIG\_BOOTFILE "\0"

#endif

#ifdef CONFIG\_LOADADDR

"loadaddr=" MK\_STR(CONFIG\_LOADADDR) "\0"

#endif

#ifdef CONFIG\_CLOCKS\_IN\_MHZ

"clocks\_in\_mhz=1\0"

#endif

#if defined(CONFIG\_PCI\_BOOTDELAY) && (CONFIG\_PCI\_BOOTDELAY > 0)

"pcidelay=" MK\_STR(CONFIG\_PCI\_BOOTDELAY) "\0"

#endif

#ifdef CONFIG\_EXTRA\_ENV\_SETTINGS

CONFIG\_EXTRA\_ENV\_SETTINGS

#endif

"\0"

};

#### 1> 在smdk2440.h中设置参数

#define CONFIG\_BOOTARGS "console=ttySAC0 root=/dev/mtdblock3"//串口0，第三个分区

set bootargs noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0,115200

#define CONFIG\_BOOTCOMMAND "nand read 30000000 kernel; bootm 30000000"

命令行设置方法：set bootcmd ‘nand read 30000000 kernel;bootm 30000000’

bootdelay 倒数计数 默认值是5

波特率 默认值是115200

ip地址

#define CONFIG\_NETMASK 255.255.255.0

#define CONFIG\_IPADDR 192.168.0.112

#define CONFIG\_SERVERIP 192.168.0.113

#define CONFIG\_ETHADDR 00:0c:29:7c:11:6b

2> 裁剪掉不需要的功能（不是必须的）

usb 的支持

//#define CONFIG\_USB\_OHCI

//#define CONFIG\_USB\_KEYBOARD

//#define CONFIG\_USB\_STORAGE

//#define CONFIG\_DOS\_PARTITION

RTC

//#define CONFIG\_RTC\_S3C24X0

/\*

\* BOOTP options

\*/

//#define CONFIG\_BOOTP\_BOOTFILESIZE

//#define CONFIG\_BOOTP\_BOOTPATH

//#define CONFIG\_BOOTP\_GATEWAY

//#define CONFIG\_BOOTP\_HOSTNAME

文件系统

/\*

\* File system

\*/

//#define CONFIG\_CMD\_FAT

//#define CONFIG\_CMD\_EXT2

//#define CONFIG\_CMD\_UBI

//#define CONFIG\_CMD\_UBIFS

//#define CONFIG\_CMD\_MTDPARTS

//#define CONFIG\_MTD\_DEVICE

//#define CONFIG\_MTD\_PARTITIONS

//#define CONFIG\_YAFFS2

//#define CONFIG\_RBTREE

//#define CONFIG\_CMD\_DATE

//#define CONFIG\_CMD\_DHCP

//#define CONFIG\_CMD\_USB

3> 重新设置分区smdk2440.h

根据uboot帮助信息找到如何配置

？ save

saveenv – save environment vaiable to persistent storage

查找saveenv 发现可以保存位置：

Env\_flash.c (common):int saveenv(void)

Env\_mmc.c (common):int saveenv(void)

Env\_nand.c (common):int saveenv(void)

我们要保存在nand中，需要编译Env\_nand.c。根据Makefile查找需要定义的宏

vim common/Makefile 发现需要定义CONFIG\_ENV\_IS\_IN\_NAND。定义该宏后

执行save命令环境变量就会保存在nand中。起始地址必须是nand块对齐的，因为

nand是块擦除的。

/\*默认环境变量在\*/

#if 0

#define CONFIG\_ENV\_ADDR (CONFIG\_SYS\_FLASH\_BASE + 0x070000)

#define CONFIG\_ENV\_IS\_IN\_FLASH

#define CONFIG\_ENV\_SIZE 0x10000

#endif

/\*保存在Nand\*/

#define CONFIG\_ENV\_IS\_IN\_NAND

#define CONFIG\_ENV\_OFFSET 0x00040000

#define CONFIG\_ENV\_SIZE 0x20000

#define CONFIG\_ENV\_RANGE CONFIG\_ENV\_SIZE// 擦除范围

4> 重新编译下载uboot依然发现提示错误

“bad CRC,using default envionment”

因为没有把设定的参数保存。需要在uboot命令行输入save命令，保存参数后重启。

5> 设置分区信息

分区信息只是为了方便用名字代替地址，并不会传输给内核。内核必须重新设定分区信息。

为了使能mtdpart相关命令，在smdk244.h中添加如下信息。

vim common/Makefile 搜索mtdparts ,只有定义了CONFIG\_CMD\_MTDPARTS才能编译cmd\_mtdparts.c。

#define CONFIG\_CMD\_MTDPARTS

#define CONFIG\_MTD\_DEVICE

#define MTDIDS\_DEFAULT "nand0=jz2440-0" /\* 哪一个设备 \*/

#define MTDPARTS\_DEFAULT "mtdparts=jz2440-0:256k(u-boot)," \

"128k(params)," \

"2m(kernel)," \

"-(rootfs)" \ /\*- 代表剩下所有\*/

在board\_init\_r()函数

run\_command("mtdparts default", 0);

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

for (;;) {

main\_loop();

}

### 3.2.10 支持烧写文件系统

烧写jffs2

1> tftp 30000000 fs\_mini\_mdev\_jffs2

2> nand erase.part rootfs

3> nand write.jffs2 30000000 0x00260000 5b89a0(文件实际长度)

4> set bootargs console=ttySCA0 root=/dev/mtdblock3 rootfstype=jffs2

支持烧写yaffs2

1> #define CONFIG\_CMD\_NAND\_YAFSS

2> tftp 30000000 fs\_mini\_mdev.yaffs2

3> nand erase.part rootfs

4> nand write.yaffs 30000000 260000 8896c0 ($filesize)

显示 no init fount 错误

### 3.2.11 修改uboot内部的bug

需要修改nand\_util.c 中nand\_write\_skip\_bad处 566行

if(!rval) 改为 if(rval) 去掉！

行 556中，ops.mode=MTD\_OOB\_AUTO

改为ops.mode=MTD\_OOB\_RAW

f (!need\_skip && !(flags & WITH\_DROP\_FFS))

修改位

f (!need\_skip && !(flags & WITH\_DROP\_FFS)&& !(flags & WITH\_YAFFS\_OOB)) {

# 4 uboot 命令

nand write 0 0 80000//从norflash 0地址拷贝到nand 0地址，长度80000.

nand read 30000000 0 80000 //从nand 0地址读取80000数据到30000000

cmp.b 0 30000000 80000//比较0地址和30000000的数据是否相同，长度80000

set machid 16a

如何知道栈地址？

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)/\*sp=30000f80\*/

反汇编，将u-boot.bin反汇编，执行命令

arm-linux-objdump –D u-boot > u-boot.dis

在反汇编文件u-boot.dis中找到<call\_board\_init\_f>这个标号。得到汇编指令：

ldr sp,[pc,#984];478<fiq+0x58> [pc,#984]=478,即数据存储在0x478地址处。在478地址处找到sp地址 sp=30000f80。

反汇编学习

源文件ldr r0=0x56000010

mov r1,#0x00000400

反汇编文件 0: e59f0014 ldr r0,[pc,#20]; 0x1c

4: e3a0b01 mov r1,#1024; 0x400

意思： 程序存储地址 机器指令 汇编指令 数据或者内存地址

第一条指令把内存单元pc+20的值加载到r0,pc的值是当前指令地址+8，即把地址28处的值加载到r0中。

# 5 uboot中下载程序

## 5.1 使用usb 下载uboot

jtag烧写uboot 1.1.6到Norflash。securCRT 串口115200连接开发板

启动uboot,输入q放弃菜单选项，输入Mtd查看分区，发现uboot分区不足以下载

2012版uboot。采用usb下载

打开DNW软件，未连接时显示usb:x,连接usb到电脑显示usb:ok

### 5.1.1 下载程序到nor flash

1> usb 1 300000000 //1一直等到下载完毕，usb是uboot命令，使用usb下载程序到0x30000000地址。

2> protect off all //解flash保护

3> erase 0 7ffff //擦除flash

4> cp.b 30000000 0 80000 //将内存30000000起始数据按字节拷贝到 0-80000地址

### 5.1.2 下载程序到nand flash

1> usb 1 300000000 //1一直等到下载完毕，usb是uboot命令，使用usb下载程序到0x30000000地址。

2> nand erase 0 80000

3> nand write 30000000 0 80000

## 5.2 使用tftp 下载内核

1> 设置好主机的IP

IP 192.168.0.113;网关：192.168.0.1；DNS:192.168.0.1

2>设置好uboot IP

set ipaddr 192.168.0.112

set ethaddr 00:0c:29:7c:11:6b(参考虚拟机)

set gatewayip 192.168.0.1

set serverip 192.168.0.113

3>启动tftpd32 v4.52 标准版（防火墙最好关闭）

设置好下载的文件目录

设置serverIP 192.168。0.113

4> secucrt 输入:

tftp 30000000 uimage;

nand erase.part kernel

nand write 30000000 kernel

bootm 30000000

### 5.3 制作补丁

1> make distclean

2> mv u-boot-2012.04.01 u-boot-2012.04.01\_100ask //修改过的uboot重命名

tar xjf u-boot-2012.04.01.tar.bz2

3>diff –urN u-boot-2012.04.01 u-boot-2012.04.01\_100ask > u-boot-2012.04.01\_100ask.patch//未修改过的uboot 和修改过的uboot对比