目录

[说明 2](#_Toc372792572)

[准备工作 2](#_Toc372792573)

[增加对S3C2440平台的支持 4](#_Toc372792574)

[增加NorFlash驱动支持 12](#_Toc372792575)

[添加NAND驱动 14](#_Toc372792576)

[增加对NAND启动方式支持 20](#_Toc372792577)

[加入网卡DM9000驱动支持 31](#_Toc372792578)

[加入MTD（NAND）分区 36](#_Toc372792579)

[启动zImage内核的支持 38](#_Toc372792580)

[USB设备驱动支持 45](#_Toc372792581)

[烧写YAFFS2文件系统支持 52](#_Toc372792582)

[使U-BOOT支持jffs2文件系统 56](#_Toc372792583)

[实现Tab键命令自动补齐、上下箭头键调用历史命令 57](#_Toc372792584)

[修改命令提示符 57](#_Toc372792585)

[添加启动菜单 57](#_Toc372792586)

[添加LCD驱动和LOGO制作 61](#_Toc372792587)

[附录一深度分析NandFlash—控制器参数TACLS、TWRPH0和TWRPH1的确定 71](#_Toc372792588)

[附录二CONFIG\_DM9000\_BASE值的确定 74](#_Toc372792590)

[附录三全局配置选项说明 76](#_Toc372792592)

[附录四Uboot常用命令 78](#_Toc372792594)

# 说明

Uboot版本：2010.06

交叉编译器版本：arm-linux-gcc 4.3.2

操作系统版本：Ubuntu 13.10

开发板平台：

ARM920T—TX2440

NOR FLASH:EN29LV160BB-70TIP (2M) x8bit

NAND FLASH: K9F2G08U0B (256M+8M)x8bit

SDRAM:H57V2562GTR-75C x 2 (32M) x 2

网卡: DM9000E

显示器:AT070TN90(7’ TFTLCD)

本文档主要参考了XC2440和TX2440的移植手册，并结合自己的移植经验，还有一些帮助资料，更好地解释某些移植步骤的原因。

源代码下载地址:

<http://yun.baidu.com/s/1Geq0b>

移植**uboot**步骤说明：

以下是根据长期的开发经验总结出的uboot移植开发的步骤，供学习者参考，本手册也是大体按照这个步骤进行的。

做uboot移植开发，首先要保证uboot能适应自己的硬件平台，然后要能实现在flash上烧写系统和引导系统的功能，这些基本的功能做好以后，就可以给uboot加入一些辅助的功能，使uboot功能更强更方便使用。

# 准备工作

解压源码：

tar xjvf u-boot-2010.06.tar.bz2

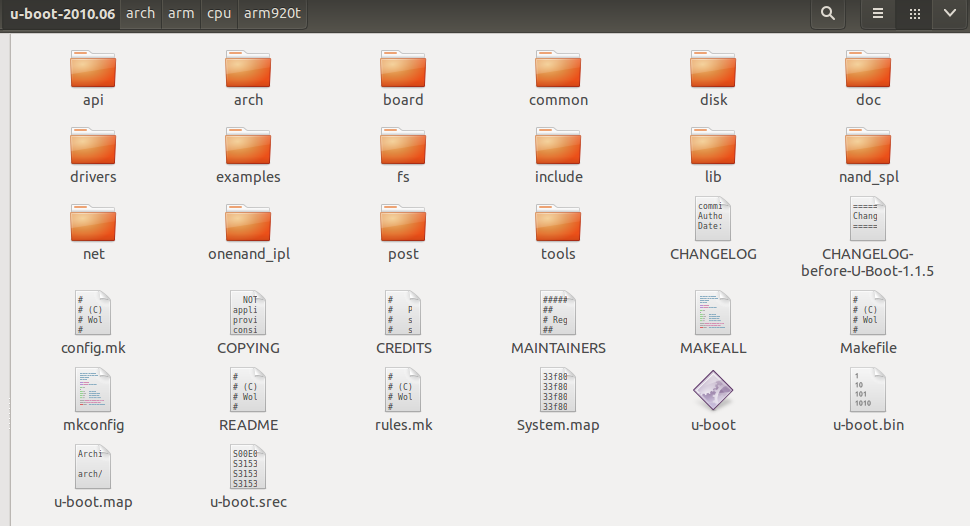
源码目录加上所有权限

sudo chmod 777 u-boot-2010.06 –R（参数-R表示递归地改变每个子目录下文件的权限）

进入源码目录：

cd u-boot-2010.06

源码结构说明：



**arch**：各种架构CPU相关的文件，类似linux的arch目录

arch/arm/cpu：ARM架构的各种CPU的初始化文件，我们用到的是arm920t/s3c24xx

arch/arm/include：ARM架构各种CPU相关的头文件，我们用到的是asm/arch-s3c24x0

arch/arm/lib：ARM架构相关的库文件

**board**：常用的主板BSP文件，每一种板对应一个目录，如smdk2410

**common**：与架构无关的通用函数的实现，包括main函数，各种命令的实现

**drivers**：通用的设备驱动程序

**fs**：文件系统的实现，包括yaffs2、cramfs、ext2、jffs2等

**include**：全局的头文件和主板的配置文件

**lib**：平台无关的通用库文件

**net**：网络协议相关的文件

**post**：开机自检相关的文件

**tools**：uboot自带的工具

我们移植的时候，只需要编译和我们目标硬件平台相关的文件，这就需要对编译选项进行配置，告诉系统，需要编译到哪些目录下编译哪些文件，这些都是通过Makefile文件来实现的。

首先，打开主目录下的**Makefile**文件，找到如下内容：

smdk2410\_config **:**unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t smdk2410 samsung s3c24x0

这个是针对三星smdk2410开发板的配置选项，这个开发板使用的S3C2410处理器，和我们使用的S3C2440处理器是类似的，大部分配置可以通用，但是在某些硬件方面也有一些差别，需要我们自己编写代码，以支持S3C2440处理器。

配置各项含义如下：

**arm**：CPU的架构(ARCH) **arm920t**：CPU的类型(CPU)

**smdk2410**：开发板的型号(BOARD)

**samsung**：开发者/或经销商(VENDER)

**s3c24x0**：片上系统（SOC）

配置选项所代表的目录关系说明：

**ARCH**：对应于主目录下的arch/arm目录

**CPU**：对应于arch/arm/cpu目录下的子目录，此值为arm920t，就使用arch/arm/cpu/arm920t目录下的代码

**BOARD**：开发板的主目录，位于board目录下，要和下面的VENDER字段配合使用

**VENDER**：对应于board目录下的子目录，此值为samsung，那么BOARD目录就包含在board/samsung子目录下，BOARD的值为smdk2410，那对应的开发板目录即为：board/samsung/smdk2410；如果VENDER的值为NULL，开发板目录即为：board/smdk2410

**SOC**：对应于是CPU目录下的子目录，此值为s3c24x0，CPU为arm920t，对应的目录为：arch/arm/cpu/arm920t/s3c24x0

# 增加对S3C2440平台的支持

**修改主目录下的Makefile文件**

在smdk2410之后仿照smdk2410的配置添加针对smdk2440开发板的配置：

smdk2410\_config **:**unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t smdk2410 samsung s3c24x0

smdk2440\_config **:**unconfig

@$(MKCONFIG) $(@:\_config=) arm arm920t smdk2440 samsung s3c24x0

（注意：第二行前面是**TAB**键，而不是空格！！）

指定交叉编译器，修改第159行为：

CROSS\_COMPILE ?=arm-linux-

**建立自己的目标板**

在**board/samsung**目录下，新建自己的开发板目录**smdk2440**，把**board/samsung/smdk2410**目录下的所有文件拷到**smdk2440**目录下，把**smdk2410.c**改名为**smdk2440.c**。修改该目录下的**Makefile**（第28行），把smdk2410.o改为smdk2440.o。

COBJS **:=**smdk2440.o flash.o

SOBJS **:=**lowlevel\_init.o

删除board目录下的所有文件夹，只保留samsung目录。

删除samsung目录下的所有文件夹，只保留smdk2440目录。

创建配置文件

在**include/configs**目录下创建开发板的配置头文件，把**smdk2410.h**复制为**smdk2440.h**，我们的uboot只需要用到这一个配置文件，其他的文件都没有用了。

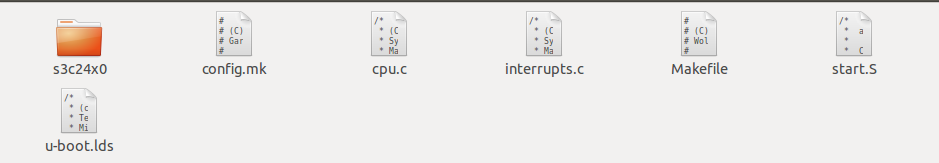
删除include/configs目录下所有文件，只保留smdk2440.h

**删除arch目录中多余文件**

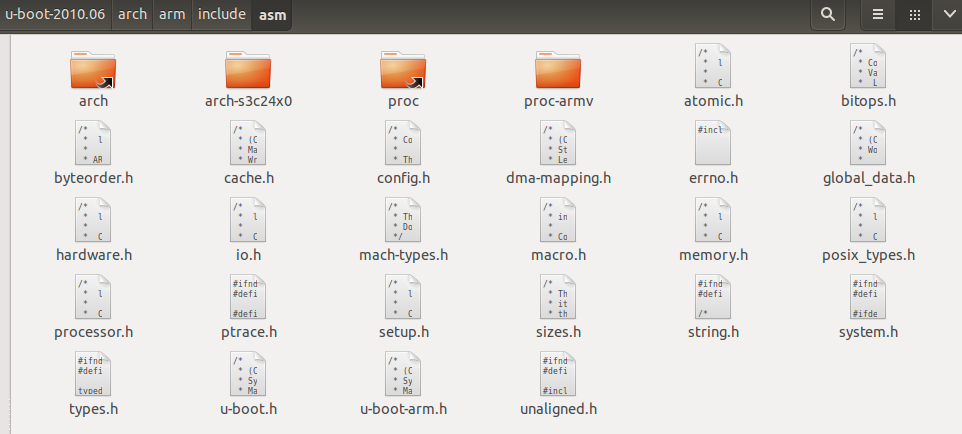
进入arch目录，删除所有文件夹，只保留arm目录。

进入arm/cpu目录，删除所有文件夹，只保留arm920t目录

进入arm920t目录，删除其他cpu文件夹，只保留s3c24x0



进入arch/cpu/include/asm目录，删除类似arch-xxx的文件夹，只保留s3c2440架构的



开灯

修改arch/arm/cpu/arm920t/start.S文件中的LED部分

mrs r0, cpsr

bic r0, r0, #0x1f

orr r0, r0, #0xd3

msr cpsr, r0

/\*bl coloured\_LED\_init

bl red\_LED\_on\*/

@ All LED on

#define GPIO\_CTL\_BASE 0x56000000

#define oGPIO\_F 0x50

#define oGPIO\_CON 0x0

#define oGPIO\_DAT 0x4

#define oGPIO\_UP 0x8

mov r1, #GPIO\_CTL\_BASE

add r1, r1, #oGPIO\_F

ldr r2,=0xff55

str r2, [r1, #oGPIO\_CON]

mov r2, #0xff

str r2, [r1, #oGPIO\_UP]

mov r2, #0x00

str r2, [r1, #oGPIO\_DAT]

关闭中断

修改arch/arm/cpu/arm920t/start.S文件中的中断关闭部分：

# if defined(CONFIG\_S3C2410)

ldr r1, =0x3ff

ldr r0, =INTSUBMSK

str r1, [r0]

# endif

# if defined(CONFIG\_S3C2440)

ldr r1, =0x7fff

ldr r0, =INTSUBMSK

str r1, [r0]

# endif

由于需要定义CONFIG\_S3C2440，所以在include/configs/smdk2440.h中添加CONFIG\_S3C2440的定义。

#define CONFIG\_ARM920T 1 /\* This is an ARM920T Core \*/

#define CONFIG\_S3C24X0 1 /\* in a SAMSUNG S3C24x0-type SoC \*/

#define CONFIG\_S3C2410 1 /\* specifically a SAMSUNG S3C2410 SoC \*/

#define CONFIG\_SMDK2410 1 /\* on a SAMSUNG SMDK2410 Board \*/

#define CONFIG\_S3C2440 1

修改时钟

因为我们使用的代码是2410平台的，2410和2440处理器的时钟设置有差别，需要修改以适应于2440处理器，否则系统不能运行。

修改arch/arm/cpu/arm920t/start.S文件中的时钟初始化部分：

添加MPLLCON的地址

# define INTSUBMSK 0x4A00001C

# define CLKDIVN 0x4C000014 /\* clock divisor register \*/

# define MPLLCON 0x4C000004

# endif

添加时钟设置

**先**屏蔽掉2410的**CLKDIVN**的设置

**#if 0**

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

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

ldr r0**, =**CLKDIVN

**mov**r1**,** #3

**str**r1**, [**r0**]**

**#endif**

/\* FCLK:HCLK:PCLK = 1:4:8 UCLK = UPLL\*/

ldr r0, =CLKDIVN

mov r1, #5

str r1, [r0]

mrc p15, 0, r1, c1, c0, 0

orr r1, r1, #0xc0000000

mcr p15, 0, r1, c1, c0, 0

ldr r1, =MPLLCON

ldr r2, =0x5c011

str r2, [r1] @400mhz

说明：**mrc p15, 0, r1, c1, c0, 0**、**orr r1, r1, #0xc0000000**、**mcr p15, 0, r1, c1, c0, 0**这**3**句程序非常重要，对于**S3C2440**，**CLKDIVN**的第【**2:1**】位为**HDIVN**，如果**HDIVN**非**0**，**CPU**总线模式应该从“**快速总线模式**”转换为“**异步总线模式**”**,**可以通过上面的**3**句程序实现。否则**CPU**的工作频率将自动变成**HCLK**，不再是**FCLK**。具体详见《S3C2440用户手册》第7章FCLK、HCLK和PCLK部分。这对**SDRAM**的初始化至关重要。

同时把smdk2440.c中的clock初始化代码部分屏蔽掉

**#if 0**

#define FCLK\_SPEED 1

#if FCLK\_SPEED==0 /\* Fout = 203MHz, Fin = 12MHz for Audio \*/

#define M\_MDIV 0xC3

#define M\_PDIV 0x4

#define M\_SDIV 0x1

#elif FCLK\_SPEED==1 /\* Fout = 202.8MHz \*/

#define M\_MDIV 0xA1

#define M\_PDIV 0x3

#define M\_SDIV 0x1

#endif

#define USB\_CLOCK 1

#if USB\_CLOCK==0

#define U\_M\_MDIV 0xA1

#define U\_M\_PDIV 0x3

#define U\_M\_SDIV 0x1

#elif USB\_CLOCK==1

#define U\_M\_MDIV 0x48

#define U\_M\_PDIV 0x3

#define U\_M\_SDIV 0x2

**#endif**

添加以下宏定义

#define S3C2440\_MPLL\_400MHZ ((0x5c<<12)|(0x01<<4)|(0x01)) // 400MHz

#define S3C2440\_UPLL\_48MHZ ((0x38<<12)|(0x02<<4)|(0x02)) // 48MHz

修改board\_init函数中时钟部分：

/\* to reduce PLL lock time, adjust the LOCKTIME register \*/

clk\_power**->**LOCKTIME **=** 0xFFFFFF**;**

/\* configure UPLL \*/

clk\_power**->**UPLLCON **=** S3C2440\_UPLL\_48MHZ**;** //fin=12.000MHz

/\* some delay between MPLL and UPLL \*/

delay**(**4000**);**

/\* configure MPLL \*/

clk\_power**->**MPLLCON **=** S3C2440\_MPLL\_400MHZ**;** //fin=12.000MHz

/\* some delay between MPLL and UPLL \*/

delay**(**8000**);**

参照《S3C2440用户手册》：当你设置MPLL 和UPLL 的值时，你必须首先设置UPLL 值再设置MPLL 值。（大约需要7 个NOP 的间隔）

修改获得时钟接口函数

现在已经完成了S3C2440平台的时钟设置，但是串口还不能正常打印信息。因为串口的波特率是要根据PCLK来计算的，在drivers/serial/serial\_s3c24x0.c的\_serial\_setbrg函数中使用get\_PCLK函数获得前面设置的PCLK时钟计算串口波特率，get\_PCLK的实现文件为：arch/arm/cpu/arm920t/s3c24x0/speed.c

这个文件里实现几个获得时钟频率的接口函数，供驱动中调用，但是适应于S3C2410平台的，和S3C2440平台的有差别，我们也要改成适应于S3C2440平台的。

修改get\_PLLCLK函数

static ulong get\_PLLCLK**(**int pllreg**)**

**{**

……………………

//return (CONFIG\_SYS\_CLK\_FREQ \*m) / (p << s);

**return (**CONFIG\_SYS\_CLK\_FREQ **\* m \* 2) / (**p **<<**s**);**

**}**

CONFIG\_SYS\_CLK\_FREQ在include/configs/smdk2440.h中定义，为12000000

加入关于时钟分频的宏定义，在下面的get\_HCLK函数中要用到：

#define S3C2440\_CLKDIVN\_HDIVN\_MASK (3<<1)

#define S3C2440\_CLKDIVN\_HDIVN\_1 (0<<1)

#define S3C2440\_CLKDIVN\_HDIVN\_2 (1<<1)

#define S3C2440\_CLKDIVN\_HDIVN\_4\_8 (2<<1)

#define S3C2440\_CLKDIVN\_HDIVN\_3\_6 (3<<1)

#define S3C2440\_CAMDIVN\_HCLK3\_HALF (1<<8)

#define S3C2440\_CAMDIVN\_HCLK4\_HALF (1<<9)

下面需要分别修改get\_HCLK函数，大致思路是：根据CLKDIVN和CAMDIVN的值计算出时钟分频值，再用FCLK除以分频值得到HCLK和PCLK。

因为要用到CAMDIVN寄存器，这是S3C2440才带有的，S3C2410的代码中没有此寄存器，我们需要在寄存器声明的头文件中加入。

寄存器声明头文件路径为：arch/arm/include/asm/arch-s3c24x0/s3c24x0.h

structs3c24x0\_clock\_power **{**

……………

u32 CLKDIVN**;**

u32 CAMDIVN**;**

**};**

回到speed.c文件，修改get\_HCLK函数：

ulong get\_HCLK(void)

{

struct s3c24x0\_clock\_power \*clk\_power = s3c24x0\_get\_base\_clock\_power();

unsigned long clkdiv;

unsigned long camdiv;

int hdiv;

clkdiv = readl(&clk\_power->CLKDIVN);

camdiv = readl(&clk\_power->CAMDIVN);

switch (clkdiv & S3C2440\_CLKDIVN\_HDIVN\_MASK) {

case S3C2440\_CLKDIVN\_HDIVN\_1:

hdiv = 1;

break;

case S3C2440\_CLKDIVN\_HDIVN\_2:

hdiv = 2;

break;

case S3C2440\_CLKDIVN\_HDIVN\_4\_8:

hdiv = (camdiv & S3C2440\_CAMDIVN\_HCLK4\_HALF) ? 8 : 4;

break;

case S3C2440\_CLKDIVN\_HDIVN\_3\_6:

hdiv = (camdiv & S3C2440\_CAMDIVN\_HCLK3\_HALF) ? 6 : 3;

break;

}

//return (readl(&clk\_power->CLKDIVN) & 2) ? get\_FCLK() / 2 : get\_FCLK();

return get\_FCLK() / hdiv;

}

**get\_PCLK**函数不用修改。

设置内存参数

board/samsung/smdk2440/lowlevel\_init.S是用于初始化S3C2440的存储控制器，我们主要看SDRAM所接的Bank6的设置，Bank7没有用到

#define B6\_BWSCON (DW32) /\* Bandk6的位宽是32位\*/

#define B6\_MT 0x3 /\* SDRAM \*/

#define B6\_Trcd 0x1

#define B6\_SCAN 0x1 /\* 9bit \*/

/\* REFRESH parameter \*/

#define REFEN 0x1 /\* Refresh enable \*/

#define TREFMD 0x0 /\* CBR(CAS before RAS)/Auto refresh \*/

#define Trp 0x0 /\* 2clk \*/

#define Trc 0x3 /\* 7clk \*/

#define Tchr 0x2 /\* 3clk \*/

//#define REFCNT 1113 /\* period=15.6us, HCLK=60Mhz,(2048+1-15.6\*60) \*/

#define REFCNT 0x4f5

REFCNT是SDRAM的刷新率，这个值要根据总线频率和所使用的SDRAM做相应修改.

计算公式：刷新周期 = (211 - REFCNT + 1) / HCLK

怎么确定所用SDRAM的刷新周期呢？

目前公认的标准是，存储体中电容的数据有效保存期上限是64ms（毫秒，1/1000秒），也就是说每一行刷新的循环周期是64ms。这样每一行的刷新所用时间就是64ms/行数，即刷新周期。我们在看内存规格时，经常会看到4096 Refresh Cycles/64ms或8K Refresh Cycles/64ms的标识，这里的4096与8K就代表这个芯片中每个L-Bank的行数。刷新命令一次对一行有效，刷新周期也是随总行数而变化，4096行时为15.625us，8K行时就为7.8125us

修改**u-boot.lds**：

路径为arch/arm/cpu/arm920t/u-boot.lds，把lowlevel\_init.S的目标文件加入到.text中：

.text :

{

arch/arm/cpu/arm920t/start.o (.text)

board/samsung/smdk2440/lowlevel\_init.o (.text)

\*(.text)

}

u-boot.lds决定了u-boot可执行映像的连接方式，以及各个段的装载地址（装载域）和执行地址（运行域），防止这个目标文件被放在 4K“起步石”之后。

指定**MACH\_TYPE**

在**board/samsung/smdk2440/smdk2440.c**文件中的board\_init()函数中，查看gd->bd->bi\_arch\_number变量的值，即为MACH\_TYPE：

/\* arch number of S3C2440-Board \*/

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

MACH\_TYPE在**arch/arm/include/asm/mach-types.h**文件定义，其中MACH\_TYPE\_S3C2440的值是362，这个要和内核中的mach-types一致，如果使用我们提供的内核，此值不用改。



MACH\_TYPE只是在启动linux内核时用到，不影响uboot的运行。

到此，uboot在S3C2440平台的基础移植工作已完成，接下来编译u-boot，然后在开发板上运行一下，验证是否可以正常启动。

执行

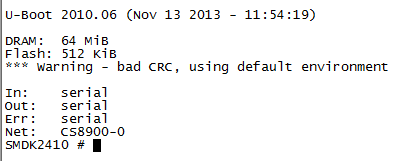
make smdk2440\_config

成功后显示：



执行make all，生成的uboot.bin烧入开发板，暂时只能烧写到NorFlash中。

成功后显示：



# 增加NorFlash驱动支持

根据上一节打印出的信息可以看到，现在检测到的Flash容量是512kB的，因为uboot默认的NorFlash配置是AM29LV400，TX2440上使用的NorFlash型号为EN29LV160BB（容量2M），我们需要修改成2M NorFlash的配置。

（注意：以下仅仅是修改了**NorFlash**的**Size**和**Sectors Number**的定义，驱动都是一样的。EN29LV160B是bottom boot sector，容量16Mbit，这与include/flash.h中AM29LV160B定义类似，所以我们可以定义**NorFlash**型号为**AM29LV160**）

修改**include/configs/smdk2440.h**

注释掉AM29LV400和AM29LV800的定义，加入AM29LV160的定义：

#if 0

#define CONFIG\_AMD\_LV400 1 /\* uncomment this if you have a LV400 flash \*/

#define CONFIG\_AMD\_LV800 1 /\* uncomment this if you have a LV800 flash \*/

#endif

**#define CONFIG\_AMD\_LV160 1**

**……**

#ifdef CONFIG\_AMD\_LV400

#define PHYS\_FLASH\_SIZE 0x00080000 /\* 512KB \*/

#define CONFIG\_SYS\_MAX\_FLASH\_SECT (11) /\* max number of sectors on one chip \*/

#define CONFIG\_ENV\_ADDR (CONFIG\_SYS\_FLASH\_BASE + 0x070000) /\* addr of environment \*/

#endif

**#ifdef CONFIG\_AMD\_LV160**

**#define PHYS\_FLASH\_SIZE 0x00200000 /\* 2MB \*/**

**#define CONFIG\_SYS\_MAX\_FLASH\_SECT (35) /\* max number of sectors on one chip \*/**

**#define CONFIG\_ENV\_ADDR (CONFIG\_SYS\_FLASH\_BASE + CONFIG\_ENV\_OFFSET) /\* addr of environment \*/**

**#endif**

注释掉在nor flash中保存环境变量的宏，定义在nand flash中保存环境变量的宏，否则编译时会出错：

#define CONFIG\_ENV\_IS\_IN\_FLASH 1 //暂时未注释，因为目前还没有NAND的驱动

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

#define CONFIG\_ENV\_OFFSET 0x100000

#define CONFIG\_ENV\_SIZE 0x20000 /\* Total Size of Environment Sector \*/

sector（扇区）的数量根据芯片手册而得，35个。

CONFIG\_ENV\_ADDR 是环境变量的保存地址。

CONFIG\_SYS\_FLASH\_BASE是FLASH的起始地址，即0。

CONFIG\_ENV\_OFFSET是环境变量的偏移地址，即Uboot分区中params分区（这在后面NAND FLASH移植部分）的起始地址，所以要统一。环境变量最好保存在NAND FLASH中，目前只修改了NOR FLASH部分，还未添加NAND FLASH驱动，所以暂时保留了CONFIG\_ENV\_IS\_IN\_FLASH的定义，将环境变量保存至NOR FLASH，后面添加了NAND FLASH驱动后再注释掉。

CONFIG\_ENV\_SIZE是环境变量大小，即params分区大小。

修改**board/samsung/smdk2440/flash.c**

在flash\_init函数中加入flash\_id:

#elif defined(CONFIG\_AMD\_LV800)

**(**AMD\_MANUFACT **&**FLASH\_VENDMASK**) |**

**(**AMD\_ID\_LV800B **&**FLASH\_TYPEMASK**);**

**#elif defined(CONFIG\_AMD\_LV160)**

**(AMD\_MANUFACT & FLASH\_VENDMASK) |**

**(AMD\_ID\_LV160B & FLASH\_TYPEMASK);**

#else

flash\_print\_info函数加入显示flash\_info语句：

**case(**AMD\_ID\_LV800B **&**FLASH\_TYPEMASK**):**

printf**(**"1x Amd29LV800BB (8Mbit)\n"**);**

**break;**

**case (AMD\_ID\_LV160B & FLASH\_TYPEMASK):**

**printf ("1x Amd29LV160B (2M)\n");**

**break;**

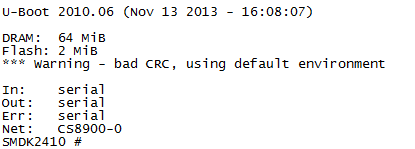
**再此编译，**，建议每次编译前都清除一下上次编译留下的中间代码：

#make clean

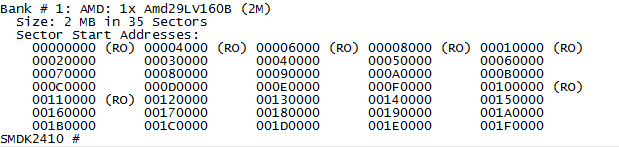
#make smdk2440\_config

#make all

**生成u-boot.bin，烧入开发板。成功后，会发现Flash已经变成2MB了。**



在开发板中输入命令：flinfo，会显示：



RO表示该扇区处于写保护状态，是只读的

在对该扇区擦除、写入之前，要先解除写保护，命令为：protect off all

擦除Flash操作：

erase all 擦除整片Flash

erase start end 擦除sectors 如：erase 0x20000 0x2ffff

erase start +len 擦除指定大小如：erase 0x40000 + 0x12345

写入Flash操作：

cp.b 内存地址Flash地址文件大小

如：cp.b 0x30000000 0 0x12345

作用是：将内存地址0x30000000处的内容写入到Norflash的0地址开始处，写入的数据大小为0x12345

# 添加NAND驱动

在uboot中已经带有完善的MTD/NAND驱动，支持S3C2410的NAND控制器驱动，驱动文件为：**drivers/mtd/nand/s3c2410\_nand.c**

我们使用的S3C2440的NAND控制器和S3C2410的有很大差别，需要修改驱动。

修改**include/configs/smdk2440.h**，加入NAND的定义：

/\*NAND flash settings\*/

#define CONFIG\_CMD\_NAND

#if defined(CONFIG\_CMD\_NAND)

#define CONFIG\_NAND\_S3C2440

#define CONFIG\_SYS\_NAND\_BASE 0x4E000000 //Nand配置寄存器基地址，nand.c要用到

#define CONFIG\_SYS\_MAX\_NAND\_DEVICE 1 //nand个数，nand.c要用到

#endif

修改cpu的寄存器声明的头文件，**arch/arm/include/asm/arch-s3c24x0/s3c24x0.h**

在结构体s3c2410\_nand之后添加S3C2440 NAND控制器寄存器声明：

struct s3c2440\_nand **{**

u32 NFCONF**;**

u32 NFCONT**;**

u32 NFCMMD**;**

u32 NFADDR**;**

u32 NFDATA**;**

u32 NFMECCD0**;**

u32 NFMECCD1**;**

u32 NFSECCD**;**

u32 NFSTAT**;**

u32 NFESTAT0**;**

u32 NFESTAT1**;**

u32 NFMECC0**;**

u32 NFMECC1**;**

u32 NFSECC**;**

u32 NFSBLK**;**

u32 NFEBLK**;**

**};**

修改cpu架构头文件：

**在arch/arm/include/asm/arch-s3c24x0/目录中**复制s3c2410.h为s3c2440.h，替换内容中s3c2410为s3c2440，替换内容中S3C2410为S3C2440(注意大小写)。

修改s3c24x0\_cpu.h，将s3c2410部分改为s3c2440

#elif defined CONFIG\_S3C2400

#include <asm/arch/s3c2400.h>

#elif defined CONFIG\_S3C2440

#include <asm/arch/s3c2440.h>

#else

在**drivers/mtd/nand**目录中新增s3c2440\_nand.c(参照s3c2410\_nand.c修改)

#include <common.h>

#include <nand.h>

#include <asm/arch/s3c24x0\_cpu.h>

#include <asm/io.h>

define S3C2440\_NFCONF\_TACLS(x) ((x)<<12)

#define S3C2440\_NFCONF\_TWRPH0(x) ((x)<<8)

#define S3C2440\_NFCONF\_TWRPH1(x) ((x)<<4)

#define S3C2440\_NFCONT\_SECCL (1<<6)

#define S3C2440\_NFCONT\_MECCL (1<<5)

#define S3C2440\_NFCONT\_INITECC (1<<4)

#define S3C2440\_NFCONT\_nCE (1<<1)

#define S3C2440\_NFCONT\_MODE (1<<0)

#define S3C2440\_ADDR\_NALE 0x08

#define S3C2440\_ADDR\_NCLE 0x0c

#ifdef CONFIG\_NAND\_SPL

/\* in the early stage of NAND flash booting, printf() is not available \*/

#define printf(fmt, args...)

static void nand\_read\_buf(struct mtd\_info \*mtd, u\_char \*buf, int len)

{

int i;

struct nand\_chip \*this = mtd->priv;

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

buf[i] = readb(this->IO\_ADDR\_R);

}

#endif

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

{

struct nand\_chip \*chip = mtd->priv;

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

debugX(1, "hwcontrol(): 0x%02x 0x%02x\n", cmd, ctrl);

if (ctrl & NAND\_CTRL\_CHANGE) {

ulong IO\_ADDR\_W = (ulong)nand; //NAND控制寄存器基地址

if (!(ctrl & NAND\_CLE)) //要写的是地址

IO\_ADDR\_W |= S3C2440\_ADDR\_NCLE;

if (!(ctrl & NAND\_ALE)) //要写的是命令

IO\_ADDR\_W |= S3C2440\_ADDR\_NALE;

if (cmd == NAND\_CMD\_NONE)

IO\_ADDR\_W = &nand->NFDATA;//在写完命令和地址后，一定还要把IO端口的地址重新设置为寄存器NFDATA

chip->IO\_ADDR\_W = (void \*)IO\_ADDR\_W;

if (ctrl & NAND\_NCE)

writel(readl(&nand->NFCONF) &~S3C2440\_NFCONT\_nCE,

&nand->NFCONF); //使能nand flash

else

writel(readl(&nand->NFCONF) | S3C2440\_NFCONT\_nCE,

&nand->NFCONF); //禁止nand flash

}

if (cmd != NAND\_CMD\_NONE)

writeb(cmd, chip->IO\_ADDR\_W);

}

static int s3c2440\_dev\_ready(struct mtd\_info \*mtd)

{

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

debugX(1, "dev\_ready\n");

return readl(&nand->NFSTAT) & 0x01;

}

#ifdef CONFIG\_S3C2440\_NAND\_HWECC

void s3c2440\_nand\_enable\_hwecc(struct mtd\_info \*mtd, int mode)

{

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

debugX(1, "s3c2440\_nand\_enable\_hwecc(%p, %d)\n", mtd, mode);

writel(readl(&nand->NFCONF) | S3C2440\_NFCONF\_INITECC, &nand->NFCONF);

}

static int s3c2440\_nand\_calculate\_ecc(struct mtd\_info \*mtd, const u\_char \*dat,

u\_char \*ecc\_code)

{

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

ecc\_code[0] = readb(&nand->NFECC);

ecc\_code[1] = readb(&nand->NFECC + 1);

ecc\_code[2] = readb(&nand->NFECC + 2);

debugX(1, "s3c2440\_nand\_calculate\_hwecc(%p,): 0x%02x 0x%02x 0x%02x\n",

mtd , ecc\_code[0], ecc\_code[1], ecc\_code[2]);

return 0;

}

static int s3c2440\_nand\_correct\_data(struct mtd\_info \*mtd, u\_char \*dat,

u\_char \*read\_ecc, u\_char \*calc\_ecc)

{

if (read\_ecc[0] == calc\_ecc[0] &&

read\_ecc[1] == calc\_ecc[1] &&

read\_ecc[2] == calc\_ecc[2])

return 0;

printf("s3c2440\_nand\_correct\_data: not implemented\n");

return -1;

}

#endif

int board\_nand\_init(struct nand\_chip \*nand)

{

u\_int32\_t cfg;

u\_int8\_t tacls, twrph0, twrph1;

struct s3c24x0\_clock\_power \*clk\_power = s3c24x0\_get\_base\_clock\_power();

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

debugX(1, "board\_nand\_init()\n");

writel(readl(&clk\_power->CLKCON) | (1 << 4), &clk\_power->CLKCON);

/\* initialize hardware \*/

tacls = 2;

twrph0 = 1;

twrph1 = 0;

cfg = 0;

cfg |= S3C2440\_NFCONF\_TACLS(tacls);

cfg |= S3C2440\_NFCONF\_TWRPH0(twrph0);

cfg |= S3C2440\_NFCONF\_TWRPH1(twrph1);

writel(cfg, &nand\_reg->NFCONF); //配置NFCONF

cfg = S3C2440\_NFCONT\_SECCL;

cfg |= S3C2440\_NFCONT\_MECCL;

cfg |= S3C2440\_NFCONT\_INITECC;

cfg |= S3C2440\_NFCONT\_MODE;

writel(cfg, &nand\_reg->NFCONT); //配置NFCONT

/\* initialize nand\_chip data structure \*/

nand->IO\_ADDR\_R = nand->IO\_ADDR\_W = (void \*)&nand\_reg->NFDATA;

nand->select\_chip = NULL;

/\* read\_buf and write\_buf are default \*/

/\* read\_byte and write\_byte are default \*/

#ifdef CONFIG\_NAND\_SPL

nand->read\_buf = nand\_read\_buf;

#endif

/\* hwcontrol always must be implemented \*/

nand->cmd\_ctrl = s3c2440\_hwcontrol;

nand->dev\_ready = s3c2440\_dev\_ready;

#ifdef CONFIG\_S3C2440\_NAND\_HWECC

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

#ifdef CONFIG\_S3C2440\_NAND\_BBT

nand->options = NAND\_USE\_FLASH\_BBT;

#else

nand->options = 0;

#endif

debugX(1, "end of nand\_init\n");

return 0;

}

tacls、twrph0、twrph1怎么确定呢？

[详见附录一“深度分析NandFlash-控制器参数TACLS、TWRPH0和TWRPH1的确定”。](#_附录一)

修改drivers/mtd/nand/目录下Makefile

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

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

COBJS-$(CONFIG\_NAND\_S3C64XX) += s3c64xx.o

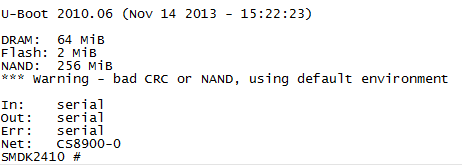
上面已完成对S3C2440的NAND驱动的支持，上一节的环境变量就可以保存在NAND FLASH了，修改include/configs/smdk2440.h

//#define CONFIG\_ENV\_IS\_IN\_FLASH 1 //暂时未注释，因为目前还没有NAND的驱动

#define CONFIG\_ENV\_IS\_IN\_NAND 1//common目录Makefile，连接时要用

#define CONFIG\_ENV\_OFFSET 0x100000

再次编译uboot，烧入到Nor Flash中，就会检测到NANDFLASH。

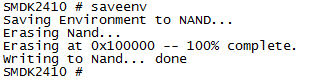


使用nand命令：nand info

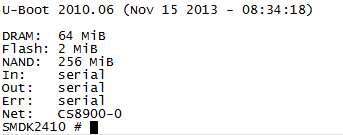


说明：现在我们的**uboot**中只是支持了**NandFlash**驱动，可以探测到**Nand**，但是还不支持从**Nand**启动，需要修改底层的启动代码以支持**Nand**启动方式。

使用命令：saveenv，保存环境变量



保存后，重启开发板，警告“\*\*\* Warning – bad CRC or NAND,using default environment”不出现了。



# 增加对NAND启动方式支持

**Nand**启动原理：

如果将S3C2440配置成从NANDFLASH启动（将开发板的启动跳线帽插上，此时OM0管脚拉低）S3C2440的Nand控制器会自动把Nandflash中的前4K代码数据搬到内部SRAM中(地址为0x40000000)，同时还把这块SRAM地址映射到了0x00000000地址，即0x40000000~0x40001000会被映射到0x00000000~0x00001000。CPU从0x00000000位置开始运行程序。

如果将S3C2440配置成从Norflash启动（将开发的启动跳线帽拔掉，此时OM0管脚拉高），0x00000000就是Norflash的物理起始地址，Norflash中的程序就从这里开始运行，不涉及到数据拷贝和地址映射

地址映射图如下所示：



如何自动识别从**NAND Flash**启动还是从**Nor flash**启动？

方法一：

代码原理：在启动的时候，用程序将0x40000000～0x40001000中的某些位置清零，然后回读0x00000000～0x00001000中的相应位置，为零说明是NAND boot，如果是原来的数据就是Nor boot。判断完后如果是NAND boot，还要恢复被改动的数据，再进入自拷贝阶段。

只要检测的位置合理，这方法是可行的。现在的关键是选什么位置的数据。经过查看源码，选择了在start.S文件开头，全局中断向量之后第57行的变量：.balignl 16,0xdeadbeef

选这个数据作为检测位置的理由如下：

（1）他是非零数，而且数据是确定的：0xdeadbeef

（2）他的位置是固定的：0x0000003c（0x4000003c），在.balignl之前，标号\_start:中每条指令占4个字节，之后的每个.word expressions（定义一个字，并分配空间）占4个字节，所以在.balignl 16,0xdeadbeef指令之前，一共占了4\*15=60个字节的空间。.balignl 16,0xdeadbeef的意思是从此开始计算（0x4000003c或0x0000003c，即60），对齐到地址为16的整数倍为止（0x40000040或0x00000040，即64），如果移动4个字节空间，则存储0xdeadbeef；如果移动0~3个字节空间，则存储值不确定。

（3）他在检测程序之前，不会影响程序的向下运行

（4）他不属于程序，他是一个程序中的魔数（Magic Number），用魔数来检测也比较合理。

所以最后的检测步骤是：在启动的时候，将0x4000003c位置开始的四个字节清零，然后读取0x0000003c位置开始的四个字节。如果回读的结果为零，说明是Nand boot，否则就是Nor boot（为了保险还可以判断是否为0xdeadbeef，不是的话就说明有未知错误，死循环!）。但是最后有一点很重要：如果是Nand boot，必须要复原清零的数据。原因是：在Nand boot过后，会核对内部SRAM中的4K程序，和从Nand中拷贝到SDRAM的前4K程序是否一致，如果不一致会进入死循。

方法二：

类似方法一，向地址0x00000000写一个数，再从0地址中把数据读回来，然后判断读回来的数与写入的数是否相等。向norflash中写数据需要特定的命令时序，而向内存中写数据可以直接向内存地址赋值。nor启动时,0地址是norflash的起始址，而向norflash写数据需要特定的命令时序，所以直接给地址赋值是不起做用的；在nand启动时，0地址是内存的地址映射，可以用直接赋值的方法向内存中写入数据。

但是此方法有个弊端，如果写入的数与原来的数本来就相等，就会出错。

intBootFrmNORflash**(**void**)**

**{**

volatile unsigned int **\***pdw **= (**volatile unsigned int **\*)**0**;**

unsigned int dwVal**;**

dwVal **= \***pdw**; //保存0地址原数据**

**\***pdw **=** 0x12345678**; //向0地址写数**

**if(\***pdw **!=** 0x12345678**) //读出0地址的数，并判断**

**{**

**return**1**; //nor boot**

**}**

**else**

**{**

**\***pdw **=** dwVal**; //恢复0地址数据**

**return**0**; //nand boot**

**}**

**}**

方法三：

通过读取BWSCON 的DW0，即OM1和OM0，当DW0=00时，nandflash启动，其他为从norflash启动

#define BWSCON 0x48000000

ldr r0,=BWSCON

ldr r0,[r0]

ands r0,r0,#6

beqnandboot //如果是00，跳转到nandflash启动代码

我们选择第一种方法来做：

判断当前代码位置，如果在内存，直接跳到**stack\_setup**

relocate: /\* relocate U-Boot to RAM \*/

adr r0, \_start /\* r0 <- current position of code \*/

ldr r1, \_TEXT\_BASE /\* test if we run from flash or RAM \*/

cmp r0, r1 /\* don't reloc during debug \*/

beq stack\_setup

如果代码当前位置不在内存中，就判断启动方式为**Nand Flash**或者**Nor Flash，**在上面代码后添加方法一的判断语句。

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CHECK\_BOOT\_FLASH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ldr r1, =0x4000003C /\* address of Internal SRAM 0x4000003C\*/

mov r0, #0 /\* r0 = 0 \*/

str r0, [r1]

mov r1, #0x3c /\* address of 0x0000003C\*/

ldr r0, [r1]

cmp r0, #0 /\*判断0x0000003C所指向的4个字节是否被清0\*/

bne norboot /\*没有被清0，就是在Nor Flash中启动，则跳到Nor Flash的启动代码处norboot\*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* CHECK\_BOOT\_FLASH \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

如果是从**Nand Flash**启动的话，那么**Nand Flash**搬移代码如下：

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* NAND\_BOOT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

nandboot:

/\* recovery \*/

ldr r0, =0xdeadbeef /\*被清0,就是在Nand Flash中启动\*/

ldr r1, =0x4000003C /\*恢复0x4000003C指向的4字节的数据0xdeadbeef\*/

str r0, [r1]

#define LENGTH\_UBOOT 0x100000

#define NAND\_CTL\_BASE 0x4E000000

#ifdef CONFIG\_S3C2440

/\* Offset \*/

#define oNFCONF 0x00

#define oNFCONT 0x04

#define oNFCMD 0x08

#define oNFSTAT 0x20

@ reset NAND

mov r1, #NAND\_CTL\_BASE

ldr r2, =( (2<<12)|(1<<8)|(0<<4)|(0<<0) )

str r2, [r1, #oNFCONF]

ldr r2, [r1, #oNFCONF]

ldr r2, =( (1<<4)|(0<<1)|(1<<0) ) @ Active low CE Control

str r2, [r1, #oNFCONT]

ldr r2, [r1, #oNFCONT]

ldr r2, =(0x6) @ RnB Clear

str r2, [r1, #oNFSTAT]

ldr r2, [r1, #oNFSTAT]

mov r2, #0xff @ RESET command

strb r2, [r1, #oNFCMD]

mov r3, #0 @ wait

nand1:

add r3, r3, #0x1

cmp r3, #0xa

blt nand1

nand2:

ldr r2, [r1, #oNFSTAT] @ wait ready

tst r2, #0x4

beq nand2

ldr r2, [r1, #oNFCONT]

orr r2, r2, #0x2 @ Flash Memory Chip Disable

str r2, [r1, #oNFCONT]

@ get read to call C functions (for nand\_read())

ldr sp, DW\_STACK\_START @ setup stack pointer

mov fp, #0 @ no previous frame, so fp=0

@ copy U-Boot to RAM

ldr r0, =TEXT\_BASE

mov r1, #0x0

mov r2, #LENGTH\_UBOOT

bl nand\_read\_ll

tst r0, #0x0

beq ok\_nand\_read

bad\_nand\_read:

loop2:

b loop2 @ infinite loop

ok\_nand\_read:

@ verify

mov r0, #0

ldr r1, =TEXT\_BASE

mov r2, #0x400 @ 4 bytes \* 1024 = 4K-bytes

go\_next:

ldr r3, [r0], #4

ldr r4, [r1], #4

teq r3, r4

bne notmatch

subs r2, r2, #4

beq stack\_setup

bne go\_next

notmatch:

loop3:

b loop3 @ infinite loop

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* NAND\_BOOT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

如果是从**Nor Flash**启动的话，那么**Nor Flash**搬移代码和原来的一样，为保险起见，先检查0x0000003C位置的魔数是否改变，并添加norboot标号

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* NOR\_BOOT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

norboot:

/\*\*\*\*\*\*\*\*\*\*\* CHECK\_FOR\_MAGIC\_NUMBER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

ldr r1, =0xdeadbeef

cmp r0, r1

bne loop3

/\*\*\*\*\*\*\*\*\*\*\* CHECK\_FOR\_MAGIC\_NUMBER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

adr r0, \_start /\* r0 <- current position of code \*/

ldr r1, \_TEXT\_BASE /\* test if we run from flash or RAM \*/

ldr r2, \_armboot\_start

ldr r3, \_bss\_start

sub r2, r3, r2 /\* r2 <- size of armboot \*/

add r2, r0, r2 /\* r2 <- source end address \*/

copy\_loop:

ldmia r0!, {r3-r10} /\* copy from source address [r0] \*/

stmia r1!, {r3-r10} /\* copy to target address [r1] \*/

cmp r0, r2 /\* until source end addreee [r2] \*/

ble copy\_loop

添加nandboot中DW\_STACK\_START的定义：

\_start\_armboot: .word start\_armboot

#define STACK\_BASE 0x33f00000

#define STACK\_SIZE 0x10000

.align 2

DW\_STACK\_START: .word STACK\_BASE+STACK\_SIZE-4

添加一个Flash启动标志（BootFlag）

为方便后面启动菜单移植时使用，从**Nand**启动时将其设置为**0**，从**Nor**启动时将其设置为**1**，默认将BootFlag设为0：

.globl \_bss\_end

\_bss\_end:

.word \_end

.globl BootFlag

BootFlag:

.word 0

在norboot代码中修改BootFlag的值为1

copy\_loop:

ldmia r0!, {r3-r10} /\* copy from source address [r0] \*/

stmia r1!, {r3-r10} /\* copy to target address [r1] \*/

cmp r0, r2 /\* until source end addreee [r2] \*/

ble copy\_loop

SetBootFlag:

ldr r0, =BootFlag

mov r1, #1 /\*从Nor启动，将标志设置为1\*/

str r1, [r0]

#endif /\* CONFIG\_SKIP\_RELOCATE\_UBOOT \*/

在**board/samsung/smdk2440**文件夹中创建**nand\_read.c**文件，实现nand read功能函数，代码如下：

/\*

\*nand\_read.c: Simple NAND read functions for booting from NAND

\*

\*This is used by cpu/arm920/start.S assembler code,

\*and the board-specific linker script must make sure this

\*file is linked within the first 4kB of NAND flash.

\*

\*Taken from GPLv2 licensed vivi bootloader,

\*Copyright (C) 2002 MIZI Research, Inc.

\*

\*Author: Hwang, Chideok <hwang@mizi.com>

\*Date : $Date: 2004/02/04 10:37:37 $

\*

\*u-boot integration and bad-block skipping (C) 2006 by OpenMoko, Inc.

\*Author: Harald Welte <laforge@openmoko.org>

\*/

#include <common.h>

#include <linux/mtd/nand.h>

#define \_\_REGb(x) (\*(volatile unsigned char \*)(x))

#define \_\_REGw(x) (\*(volatile unsigned short \*)(x))

#define \_\_REGi(x) (\*(volatile unsigned int \*)(x))

#define NF\_BASE 0x4e000000

#define NFCONF \_\_REGi(NF\_BASE + 0x0)

#define NFCONT \_\_REGi(NF\_BASE + 0x4)

#define NFCMD \_\_REGb(NF\_BASE + 0x8)

#define NFADDR \_\_REGb(NF\_BASE + 0xc)

#define NFDATA \_\_REGb(NF\_BASE + 0x10)

#define NFDATA16 \_\_REGw(NF\_BASE + 0x10)

#define NFSTAT \_\_REGb(NF\_BASE + 0x20)

#define NFSTAT\_BUSY 1

#define nand\_select() (NFCONT &= ~(1 << 1))

#define nand\_deselect() (NFCONT |= (1 << 1))

#define nand\_clear\_RnB() (NFSTAT |= (1 << 2))

static inline void nand\_wait(void)

{

int i;

while (!(NFSTAT & NFSTAT\_BUSY))

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

}

struct boot\_nand\_t {

int page\_size;

int block\_size;

int bad\_block\_offset;

// unsigned long size;

};

static int is\_bad\_block(struct boot\_nand\_t \* nand, unsigned long i)

{

unsigned char data;

unsigned long page\_num;

nand\_clear\_RnB();

if (nand->page\_size == 512) {

NFCMD = NAND\_CMD\_READOOB; /\* 0x50 \*/

NFADDR = nand->bad\_block\_offset & 0xf;

NFADDR = (i >> 9) & 0xff;

NFADDR = (i >> 17) & 0xff;

NFADDR = (i >> 25) & 0xff;

} else if (nand->page\_size == 2048) {

page\_num = i >> 11; /\* addr / 2048 \*/

NFCMD = NAND\_CMD\_READ0;

NFADDR = nand->bad\_block\_offset & 0xff;

NFADDR = (nand->bad\_block\_offset >> 8) & 0xff;

NFADDR = page\_num & 0xff;

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

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

NFCMD = NAND\_CMD\_READSTART;

} else {

return -1;

}

nand\_wait();

data = (NFDATA & 0xff);

if (data != 0xff)

return 1;

return 0;

}

static int nand\_read\_page\_ll(struct boot\_nand\_t \* nand, unsigned char \*buf, unsigned long addr)

{

unsigned short \*ptr16 = (unsigned short \*)buf;

unsigned int i, page\_num;

nand\_clear\_RnB();

NFCMD = NAND\_CMD\_READ0;

if (nand->page\_size == 512) {

/\* Write Address \*/

NFADDR = addr & 0xff;

NFADDR = (addr >> 9) & 0xff;

NFADDR = (addr >> 17) & 0xff;

NFADDR = (addr >> 25) & 0xff;

} else if (nand->page\_size == 2048) {

page\_num = addr >> 11; /\* addr / 2048 \*/

/\* Write Address \*/

NFADDR = 0;

NFADDR = 0;

NFADDR = page\_num & 0xff;

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

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

NFCMD = NAND\_CMD\_READSTART;

} else {

return -1;

}

nand\_wait();

for (i = 0; i < (nand->page\_size>>1); i++) {

\*ptr16 = NFDATA16;

ptr16++;

}

return nand->page\_size;

}

static unsigned short nand\_read\_id()

{

unsigned short res = 0;

NFCMD = NAND\_CMD\_READID;

NFADDR = 0;

res = NFDATA;

res = (res << 8) | NFDATA;

return res;

}

extern unsigned int dynpart\_size[];

/\* low level nand read function \*/

int nand\_read\_ll(unsigned char \*buf, unsigned long start\_addr, int size)

{

int i, j;

unsigned short nand\_id;

struct boot\_nand\_t nand;

/\* chip Enable \*/

nand\_select();

nand\_clear\_RnB();

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

;

nand\_id = nand\_read\_id();

if (0) { /\* dirty little hack to detect if nand id is misread \*/

unsigned short \* nid = (unsigned short \*)0x31fffff0;

\*nid = nand\_id;

}

if (nand\_id == 0xec76 || /\* Samsung K91208 \*/

nand\_id == 0xad76 ) { /\*Hynix HY27US08121A\*/

nand.page\_size = 512;

nand.block\_size = 16 \* 1024;

nand.bad\_block\_offset = 5;

// nand.size = 0x4000000;

} else if (nand\_id == 0xecf1 || /\* Samsung K9F1G08U0B \*/

nand\_id == 0xecda || /\* Samsung K9F2G08U0B \*/

nand\_id == 0xecd3 ) { /\* Samsung K9K8G08 \*/

nand.page\_size = 2048;

nand.block\_size = 128 \* 1024;

nand.bad\_block\_offset = nand.page\_size;

// nand.size = 0x8000000;

} else {

return -1; // hang

}

if ((start\_addr & (nand.block\_size-1)) || (size & ((nand.block\_size-1))))

return -1; /\* invalid alignment \*/

for (i=start\_addr; i < (start\_addr + size);) {

#ifdef CONFIG\_S3C2410\_NAND\_SKIP\_BAD

if (i & (nand.block\_size-1)== 0) {

if (is\_bad\_block(&nand, i) ||

is\_bad\_block(&nand, i + nand.page\_size)) {

/\* Bad block \*/

i += nand.block\_size;

size += nand.block\_size;

continue;

}

}

#endif

j = nand\_read\_page\_ll(&nand, buf, i);

i += j;

buf += j;

}

/\* chip Disable \*/

nand\_deselect();

return 0;

}

修改board/samsung/smdk2440目录下**Makefile**文件，加入对它的编译选项支持：

COBJS **:=**smdk2440.o flash.onand\_read.o

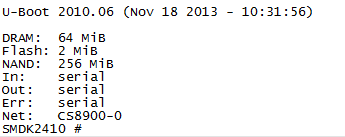
修改**arch/arm/cpu/arm920t/u-boot.lds**文件，加入：

arch/arm/cpu/arm920t/start.o (.text)

board/samsung/smdk2440/lowlevel\_init.o (.text)

board/Samsung/smdk2440/nand\_read.o (.text)

再次编译，将生成的u-boot.bin下载到Nor Flash或者是Nand Flash中，运行效果如下图：



# 加入网卡DM9000驱动支持

U-BOOT-2010.06版本已经对CS8900和DM9000X网卡有比较完善的代码支持(代码drivers/net/目录下)，在S3C24XX系列中默认对CS8900网卡进行配置使用。现在修改对我们开发板上DM9000网卡的支持,只要在个别地方根据开发板的具体网卡片选进行设置，就可以支持对S3C24XX系列中DM9000网卡的使用。

drivers/net/目录下有关DM9000的代码，发现**dm9000x.h**中对CONFIG\_DRIVER\_DM9000宏的依赖，**dm9000x.c**中对CONFIG\_DM9000\_BASE宏、DM9000\_IO宏、DM9000\_DATA等宏的依赖，所以我们修改代码如下：

在**include/configs/smdk2440.h**中取消CS8900网卡的配置，加入DM9000网卡的配置：

**#if 0**

#define CONFIG\_NET\_MULTI

#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 \*/

**#endif**

#define CONFIG\_DRIVER\_DM9000 1

#define CONFIG\_NET\_MULTI 1

#define CONFIG\_DM9000\_NO\_SROM 1

#define CONFIG\_DM9000\_BASE 0x20000300 //网卡片选地址

#define DM9000\_IO CONFIG\_DM9000\_BASE //传输IO数据的地址，即DM9000的CMD引脚为低电平时的地址

#define DM9000\_DATA (CONFIG\_DM9000\_BASE + 4) //传输DATA数据的地址，即DM9000的CMD引脚为高电平时的地址

如何确定CONFIG\_DM9000\_BASE的值？

[详见附录二“CONFIG\_DM9000\_BASE值的确定”。](#_附录二)

定义了CONFIG\_DRIVER\_DM9000之后，uboot就会自动编译drivers/net/dm9000x.c这个文件。

给**u-boot**加上**ping**命令，用来测试网络通不通

#define CONFIG\_CMD\_CACHE

#define CONFIG\_CMD\_DATE

#define CONFIG\_CMD\_ELF

#define CONFIG\_CMD\_PING//来自cmd\_net.c

修改网络参数配置：

#define CONFIG\_ETHADDR 1a:2b:3c:4d:5e:6f

#define CONFIG\_NETMASK 255.255.255.0

#define CONFIG\_IPADDR 192.168.1.10

#define CONFIG\_SERVERIP 192.168.1.30

说明：**CONFIG\_IPADDR**指的是开发板的**IP**地址，**CONFIG\_SERVERIP**指的是服务器的**IP**地址（即如果使用tftp命令时用的是本机下载，可以设置为自己电脑的IP），这两个**IP**地址必须在同一网段上，在使用网络时，必须保证电脑服务器端的**IP**地址和**CONFIG\_SERVERIP**的值保持一致。

修改**board/samsung/smdk2440/smdk2440.c**，在board\_eth\_init函数中加入dm9000初始化函数：

#ifdef CONFIG\_CS8900

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

#endif

#ifdef CONFIG\_DRIVER\_DM9000

rc **=** dm9000\_initialize**(**bis**); //来自dm9000x.c**

#endif

修改网卡驱动文件drivers/net/dm9000x.c，在dm9000\_init函数中加入设置MAC地址的操作：

**char \*tmp = getenv("ethaddr");**

**char \*end;**

**for(i = 0;i < 6; i++)**

**{**

**dev->enetaddr[i] = tmp ? simple\_strtoul(tmp, &end, 16) : 0;**

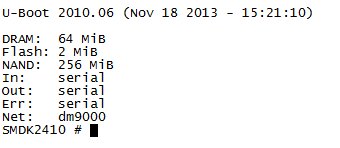
**if(tmp)**

**tmp = (\*end)? end + 1 :end;**

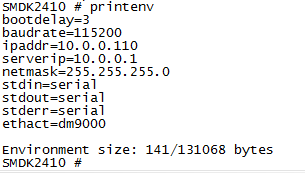
**}**

printf**(**"MAC: %pM\n"**,** dev**->**enetaddr**);**

**再次编译，将u-boot.bin烧入开发板**



使用命令printenv，发现配置仍然没有改变



这是由于环境变量在nandflash中，现在读取的还是以前的环境变量，新修改的环境变量还未重新保存。

设置新的IP地址：

setenv ethaddr 1a:2b:3c:4d:5e:6f

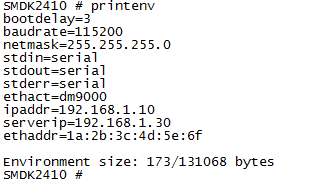
setenv ipaddr 192.168.1.10

setenv serverip 192.168.1.30

saveenv

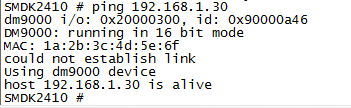
或者直接擦除环境变量分区nand erase params，再重启开发板，使用saveenv命令保存。

修改后



测试网络：

ping 192.168.1.30 出现host 192.168.1.30 is alive表示网络已通。



从上图可以看出，我们的网络已连通了，但是报了个错误“could not establish link”。解决此问题的方法：

修改**drivers/net/dm9000x.c**

屏蔽掉dm9000\_init函数中的这一部分：

#if 0

i = 0;

while (!(phy\_read(1) & 0x20)) { /\* autonegation complete bit \*/

udelay(1000);

i++;

if (i == 10000) {

printf("could not establish link\n");

return 0;

}

}

#endif

再次编译，将生成的u-boot.bin下载到Nor Flash或者是Nand Flash中。

现在就可以用TFTP下载镜像到开发板中了，我们来做个小测试。

使用tftp下载流水灯的裸机程序到内存，并运行，代码如下：

#define GPFCON (\*(volatile unsigned long \*) 0x56000050)

#define GPFDAT (\*(volatile unsigned long \*) 0x56000054)

#define GPFUP (\*(volatile unsigned long \*) 0x56000058)

#define LED1ON 0xFE //LED1点亮值为0xFE(低电平点亮)

#define LED2ON 0xFD //LED2点亮值为LED1左移1位

#define LED3ON 0xFB //LED3点亮值为LED2左移1位

#define LED4ON 0xF7 //LED4点亮值为LED3左移1位

void delay(int k);

int main()

{

GPFCON = (GPFCON | 0xFFFF) & 0xFFFFFF55; //GPF0--GPF3设置为output

GPFUP = GPFUP & 0xFFFF; //使能GPF上拉电阻

GPFDAT = 0x0F; //GPF低4位初始化为1

while(1)

{

GPFDAT = LED1ON; //点亮LED1

delay(100);

GPFDAT = LED2ON; //点亮LED2

delay(100);

GPFDAT = LED3ON; //点亮LED3

delay(100);

GPFDAT = LED4ON; //点亮LED4

delay(100);

}

}

void delay(int k)

{

int i, j;

for(i = 0; i < 10000; i ++)

for(j = 0; j < k; j ++);

}

程序注意要点：

1、不能包含库函数。

2、程序入口函数必须在最前面，例如delay函数的定义不能放在main函数之前。

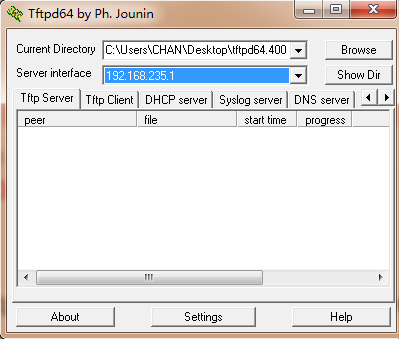
使用arm-linux-gcc-4.3.2编译，生成可执行文件：

arm-linux-gcc -nostdlib -o LED LED.c -Ttext 0x30000000

再把可执行文件生成二进制文件

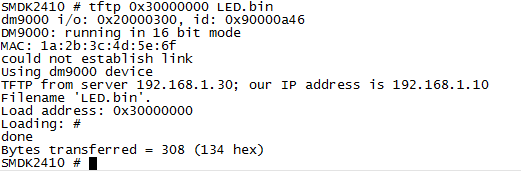
arm-linux-objcopy -O binary LED LED.bin

把LED.bin通过tftp下载到开发板，先打开tftpd32或tftpd64这个软件，server interface选择自己电脑的IP



将LED.bin文件拷贝至tftpd32这个软件同目录下

在终端上使用命令tftp 0x30000000 LED.bin，将程序下载至内存0x30000000处。



运行流水灯程序：

go 0x30000000



# 加入MTD（NAND）分区

MTD分区是uboot中一个非常重要的概念，因为uboot的主要作用就是引导操作系统烧写程序，那么我们的程序（指bootloader，内核，文件系统这些）是如何存储在MTD设备中的也是我们比较关心的。

我们开发板上使用的MTD设备就是Nandflash，我们在uboot中需要对nandflash进行分区，这和我们电脑上硬盘的分区很相似。其实就是把nandflash划分为几个地址区域，每个分区都有分区名，分区的大小和偏移地址。

有了分区，我们就可以规定，哪个分区存放什么程序，烧写nandflash的时候，就可以直接指定分区名，而不用再计算要烧写的地址了。

在uboot中已经集成了MTD分区的相关操作，我们只需要加入MTD分区的配置和分区表的定义。

在**include/configs/smdk2440.h**文件中加入MTDPARTS命令：

/\* MTDPARTS settings \*/

#define CONFIG\_CMD\_MTDPARTS 1

#define CONFIG\_MTD\_DEVICE 1

#define CONFIG\_MTD\_PARTITIONS1

加入MTD分区信息：

#define MTDIDS\_DEFAULT "nand0=nandflash0"

#define MTDPARTS\_DEFAULT "mtdparts=nandflash0:1m@0(bios),"\

"128k(params),"\

"4m(kernel),"\

"-(root)"

分区说明：

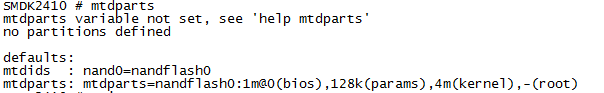
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 序号 | 分区名 | 分区大小 | 地址偏移量 | 作用 |
| 0 | bios | 0x00100000（1M） | 0x00000000 | 存放uboot |
| 1 | params | 0x00020000（128K） | 0x00100000 | 存放环境变量 |
| 2 | kernel | 0x00400000（4M） | 0x00120000 | 存放内核 |
| 3 | root | 0x0fae0000（其余） | 0x00520000 | 存放文件系统 |

如果定义了CONFIG\_CMD\_MTDPARTS，就会编译common目录下的cmd\_mtdparts.c文件，这需要用到drivers/mtd/mtdcore.c中的函数。就需要定义CONFIG\_MTD\_DEVICE和CONFIG\_MTD\_PARTITIONS这两项，才能编译mtdcore.c和mtdpart.c

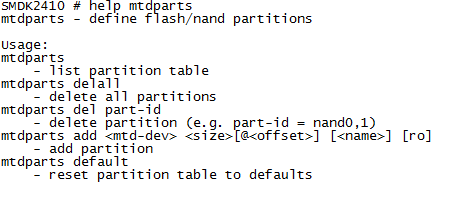
编译uboot，烧入FLASH中，输入命令：help，即可看到mtdparts命令

**mtdparts - define flash/nand partitions**

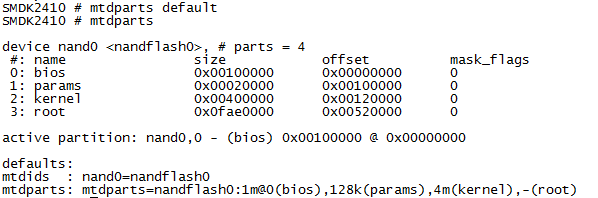
使用命令mtdparts试一下，但并没有打印出我们刚才设置的分区信息，使用printenv命令也查不到分区信息。



但是我们看到mtd defaults中的信息和我们设置的一样，使用上面的提示命令helpmtdparts，查看这个命令的帮助，我们会看到这一项：



使用命令mtdparts default，然后再用mtdparts，便可打印出分区信息



因此，我们需要在程序中执行一次mtdparts default命令，才能使分区自动生效。

修改**common/main.c**文件中main\_loop函数，加入mtdparts\_init函数的调用：

#ifdef CONFIG\_AUTO\_COMPLETE

install\_auto\_complete**();**

#endif

**#ifdef CONFIG\_CMD\_MTDPARTS**

**extern int mtdparts\_init(void);**

**if (!getenv("mtdparts"))**

**{**

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

**}**

**else**

**{**

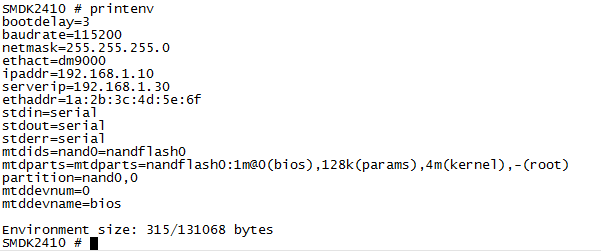
**mtdparts\_init();**

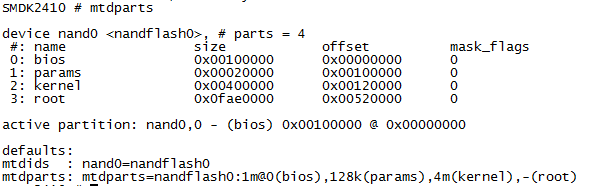
**}**

**#endif**

重新编译uboot，下到FLASH中，这样我们每次运行uboot，就可以使用我们设置的分区了。

再使用printenv和mtdparts查看一下，可以看到分区信息：





# 启动zImage内核的支持

Uboot默认只支持启动uImage镜像，还不支持启动zImage镜像。

在**arch/arm/lib**目录下创建**boot\_zImage.c**文件，代码如下：

#include <common.h>

#include <command.h>

#include <image.h>

#include <u-boot/zlib.h>

#include <asm/byteorder.h>

#include <asm/arch-s3c24x0/s3c2410.h>

#include "asm/mach-types.h"

#define LINUX\_KERNEL\_OFFSET 0x8000

#define LINUX\_PARAM\_OFFSET 0x100

#define LINUX\_PAGE\_SIZE 0x00001000

#define LINUX\_PAGE\_SHIFT 12

#define LINUX\_ZIMAGE\_MAGIC 0x016f2818

#define DRAM\_SIZE 0x04000000

extern int nand\_read\_ll(unsigned char\*, unsigned long, int);

/\*

\*Disable IRQs

\*/

#define local\_irq\_disable() \

({ \

unsigned long temp; \

\_\_asm\_\_ \_\_volatile\_\_( \

"mrs %0, cpsr @ local\_irq\_disable\n" \

" orr %0, %0, #128\n" \

" msr cpsr\_c, %0" \

: "=r" (temp) \

: \

: "memory", "cc"); \

})

static inline void cpu\_arm920\_cache\_clean\_invalidate\_all(void)

{

\_\_asm\_\_(

" mov r1, #0\n"

" mov r1, #7 << 5\n" /\* 8 segments \*/

"1: orr r3, r1, #63 << 26\n" /\* 64 entries \*/

"2: mcr p15, 0, r3, c7, c14, 2\n" /\* clean & invalidate D index \*/

" subs r3, r3, #1 << 26\n"

" bcs 2b\n" /\* entries 64 to 0 \*/

" subs r1, r1, #1 << 5\n"

" bcs 1b\n" /\* segments 7 to 0 \*/

" mcr p15, 0, r1, c7, c5, 0\n" /\* invalidate I cache \*/

" mcr p15, 0, r1, c7, c10, 4\n" /\* drain WB \*/

);

}

void cache\_clean\_invalidate(void)

{

cpu\_arm920\_cache\_clean\_invalidate\_all();

}

static inline void cpu\_arm920\_tlb\_invalidate\_all(void)

{

\_\_asm\_\_(

"mov r0, #0\n"

"mcr p15, 0, r0, c7, c10, 4\n" /\* drain WB \*/

"mcr p15, 0, r0, c8, c7, 0\n" /\* invalidate I & D TLBs \*/

);

}

void tlb\_invalidate(void)

{

cpu\_arm920\_tlb\_invalidate\_all();

}

void call\_linux(long a0, long a1, long a2)

{

local\_irq\_disable();

cache\_clean\_invalidate();

tlb\_invalidate();

\_\_asm\_\_(

"mov r0, %0\n"

"mov r1, %1\n"

"mov r2, %2\n"

"mov ip, #0\n"

"mcr p15, 0, ip, c13, c0, 0\n" /\* zero PID \*/

"mcr p15, 0, ip, c7, c7, 0\n" /\* invalidate I,D caches \*/

"mcr p15, 0, ip, c7, c10, 4\n" /\* drain write buffer \*/

"mcr p15, 0, ip, c8, c7, 0\n" /\* invalidate I,D TLBs \*/

"mrc p15, 0, ip, c1, c0, 0\n" /\* get control register \*/

"bic ip, ip, #0x0001\n" /\* disable MMU \*/

"mcr p15, 0, ip, c1, c0, 0\n" /\* write control register \*/

"mov pc, r2\n"

"nop\n"

"nop\n"

: /\* no outpus \*/

: "r" (a0), "r" (a1), "r" (a2)

: "r0","r1","r2","ip"

);

}

/\*

\* pram\_base: base address of linux paramter

\*/

static void setup\_linux\_param(ulong param\_base)

{

struct param\_struct \*params = (struct param\_struct \*)param\_base;

char \*linux\_cmd;

// printf("Setup linux parameters at 0x%08lx\n", param\_base);

memset(params, 0, sizeof(struct param\_struct));

params->u1.s.page\_size = LINUX\_PAGE\_SIZE;

params->u1.s.nr\_pages = (DRAM\_SIZE >> LINUX\_PAGE\_SHIFT);

/\* set linux command line \*/

linux\_cmd = getenv ("bootargs");

if (linux\_cmd == NULL) {

printf("Wrong magic: could not found linux command line\n");

} else {

memcpy(params->commandline, linux\_cmd, strlen(linux\_cmd) + 1);

// printf("linux command line is: \"%s\"\n", linux\_cmd);

}

}

/\*

\* dst: destination address

\* src: source

\* size: size to copy

\* mt: type of storage device

\*/

static inline int copy\_kernel\_img(ulong dst, const char \*src, size\_t size)

{

int ret = 0;

ret = nand\_read\_ll((unsigned char \*)dst,

(unsigned long)src, (int)size);

return ret;

}

int boot\_zImage(ulong from, size\_t size)

{

int ret;

ulong boot\_mem\_base; /\* base address of bootable memory \*/

ulong to;

ulong mach\_type;

boot\_mem\_base = 0x30000000;

/\* copy kerne image \*/

to = boot\_mem\_base + LINUX\_KERNEL\_OFFSET;

printf("Copy linux kernel from 0x%08lx to 0x%08lx, size = 0x%08lx ... ",

from, to, size);

ret = copy\_kernel\_img(to, (char \*)from, size);

if (ret) {

printf("failed\n");

return -1;

} else {

printf("Copy Kernel to SDRAM done,");

}

if (\*(ulong \*)(to + 9\*4) != LINUX\_ZIMAGE\_MAGIC) {

printf("Warning: this binary is not compressed linux kernel image\n");

printf("zImage magic = 0x%08lx\n", \*(ulong \*)(to + 9\*4));

} else {

// printf("zImage magic = 0x%08lx\n", \*(ulong \*)(to + 9\*4));

;

}

/\* Setup linux parameters and linux command line \*/

setup\_linux\_param(boot\_mem\_base + LINUX\_PARAM\_OFFSET);

/\* Get machine type \*/

mach\_type = MACH\_TYPE\_S3C2440;

// printf("MACH\_TYPE = %d\n", mach\_type);

/\* Go Go Go \*/

printf("NOW, Booting Linux......\n");

call\_linux(0, mach\_type, to);

return 0;

}

int do\_boot\_zImage (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

boot\_zImage(0x120000,0x400000);

return 0;

}

U\_BOOT\_CMD(

boot\_zImage, 3, 0, do\_boot\_zImage,

"boot\_zImage - boot Linux 's zImage\n",

" - boot Linux 's zImage"

);

int boot\_noos(ulong from, size\_t size)

{

int ret;

ulong boot\_mem\_base; /\* base address of bootable memory \*/

ulong to;

boot\_mem\_base = 0x30000000;

/\* copy kerne image \*/

to = boot\_mem\_base;

printf("Copy code from 0x%08lx to 0x%08lx, size = 0x%08lx ... ",

from, to, size);

ret = copy\_kernel\_img(to, (char \*)from, size);

if (ret) {

printf("failed\n");

return -1;

} else {

printf("Copy code to SDRAM done,");

}

/\* Go Go Go \*/

printf("NOW, Booting code......\n");

run\_command("go 0x30000000", 0);

return 0;

}

int do\_boot\_noos (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

boot\_noos(0x200000,0x200000);

return 0;

}

U\_BOOT\_CMD(

boot\_noos, 3, 0, do\_boot\_noos,

"boot\_noos - boot User Program\n",

" - boot User Program"

);

在同目录下**Makefile**文件中加入编译选项：

COBJS-y += interrupts.o

COBJS-y += reset.o

COBJS-y += boot\_zImage.o

分析一下代码：

intdo\_boot\_zImage **(**cmd\_tbl\_t **\***cmdtp**,** int flag**,** int argc**,** char **\***argv**[])**

**{**

boot\_zImage**(**0x120000**,**0x400000**);**

**return**0**;**

**}**

U\_BOOT\_CMD**(**

boot\_zImage**,** 3**,** 0**,** do\_boot\_zImage**,**

"boot\_zImage - boot Linux 's zImage\n"**,**

" - boot Linux 's zImage"

**);**

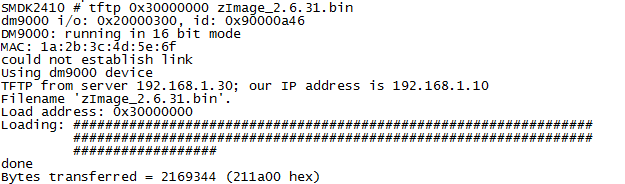
boot\_zImage函数调用copy\_kernel\_img函数，再调用nand\_read\_ll函数，读取nandflash中0x120000地址处开始的内核镜像（大小为0x400000）拷贝到内存中0x30008000地址处，接下来设置内核启动参数，得到mach\_type，最后再调用call\_linux函数，启动内核。

nand\_read\_ll函数位于board/samsung/smdk2440/nand\_read.c中，在“添加NAND启动支持”这部分已经加入了nand\_read.c的编译选项。

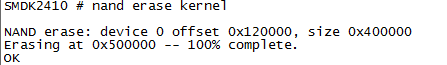
再次编译，将生成的u-boot.bin下载到Nand Flash中，可以用tftp下载内核zImage文件到开发板中，然后将其写到Nand Flash中，就可以启动LINUX了。

下载步骤：

1、使用命令：tftp 0x30000000 zImage\_2.6.31.bin，将zImage镜像下载到内存0x30000000处。



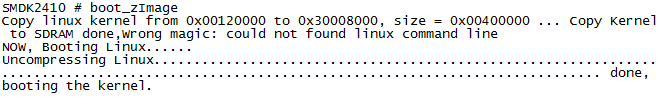
2、使用命令：nand erase kernel，擦除nand中kernel分区



3、使用命令：nand write 0x30000000 kernel，将内存从0x30000000开始的数据下载到nand的kernel分区，大小为kernel分区大小。



在uboot中使用boot\_zImage命令，启动内核。



但是现在有一个错误“could not found linux command line”，导致内核启动不起来。这是由于uboot没有向内核传递参数。

在**include/configs/smdk2440.h**中添加代码

#define CONFIG\_BOOTARGS "noinitrd root=/dev/mtdblock2 init=/linuxrc console=ttySAC0 mem=64M"

#define CONFIG\_SETUP\_MEMORY\_TAGS 1

#define CONFIG\_INITRD\_TAG 1

#define CONFIG\_CMDLINE\_TAG 1

如果内核镜像是**uImage**格式，**U-BOOT**添加自启动命令，在**include/configs/smdk2440.h**中添加代码

#define CONFIG\_BOOTCOMMAND "nboot 0x30008000 kernel; bootm 0x30008000"

或者

#define CONFIG\_BOOTCOMMAND"nand read 0x30008000 kernel; bootm 0x30008000"

自启动时（开机3秒内无输入），首先执行nboot 0x30008000 kernel，将NAND Flash中kernel区的映像文件复制到内存0x30008000上，然后执行bootm 0x30008000启动内存中的映像。

如果内核镜像是**zImage**格式，可以在**common/main.c**中添加：

debug ("### main\_loop: bootcmd=\"%s\"\n", s ? s : "<UNDEFINED>");

if (bootdelay >= 0 && s && !abortboot (bootdelay)) {

# ifdef CONFIG\_AUTOBOOT\_KEYED

int prev = disable\_ctrlc(1); /\* disable Control C checking \*/

# endif

boot\_zImage(0x120000,0x400000);

# ifndef CONFIG\_SYS\_HUSH\_PARSER

run\_command (s, 0);

# else

parse\_string\_outer(s, FLAG\_PARSE\_SEMICOLON |

FLAG\_EXIT\_FROM\_LOOP);

# endif

# ifdef CONFIG\_AUTOBOOT\_KEYED

disable\_ctrlc(prev); /\* restore Control C checking \*/

# endif

}

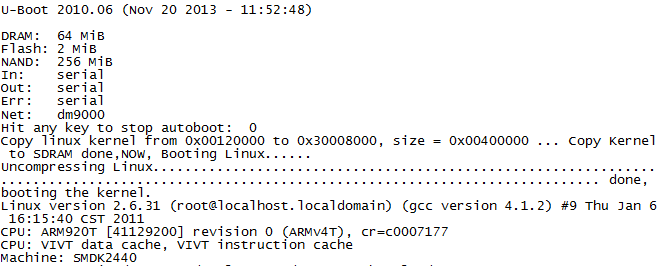
# ifdef CONFIG\_MENUKEY

if (menukey == CONFIG\_MENUKEY) {

s = getenv("menucmd");

当然我们可以两样都添加上。

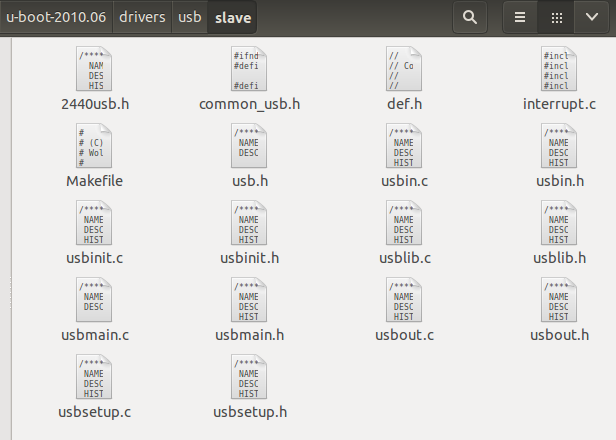
再次编译，将u-boot.bin烧入开发板，会显示如下信息：



# USB设备驱动支持

uboot中暂时没有USB Slave驱动，需要自己添加，这个驱动用于开发板连接PC机用USB接口下载程序。移植USB Slave驱动的过程比较复杂，每一步都要细心处理，否则很容易出现检测不到USB连接的情况。请按下面几个步骤移植。

1.向**drivers/usb**目录下加入**slave**目录，包括以下文件（还有一个隐藏文件.depend）：



2.查看**drivers/usb/slave/Makefile**文件，第26行

LIB := $(obj)libusb\_slave.a 🡨这个是生成的静态库的名称

在根目录下的**Makefile**文件中添加usb\_slave的静态库：

在第240行左右添加代码：

LIBS += drivers/usb/phy/libusb\_phy.a

LIBS += drivers/usb/slave/libusb\_slave.a

LIBS += drivers/video/libvideo.a

3.修改**arch/arm/include/asm/arch-s3c24x0/s3c24x0.h**文件：

修改s3c24x0\_usb\_device结构体

在第300行左右

#else /\* little endian \*/

u8 FUNC\_ADDR\_REG;

u8 res1[3];

u8 PWR\_REG;

u8 res2[3];

u8 EP\_INT\_REG;

u8 res3[15];

u8 USB\_INT\_REG;

u8 res4[3];

u8 EP\_INT\_EN\_REG;

u8 res5[15];

u8 USB\_INT\_EN\_REG;

u8 res6[3];

u8 FRAME\_NUM1\_REG;

u8 res7[3];

u8 FRAME\_NUM2\_REG;

u8 res8[3];

u8 INDEX\_REG;

u8 res9[7];

u8 MAXP\_REG;

u8 res10[3];

u8 EP0\_CSR\_IN\_CSR1\_REG;

u8 res11[3];

u8 IN\_CSR2\_REG;

u8 res12[7];

u8 OUT\_CSR1\_REG;

u8 res13[3];

u8 OUT\_CSR2\_REG;

u8 res14[3];

u8 OUT\_FIFO\_CNT1\_REG;

u8 res15[3];

u8 OUT\_FIFO\_CNT2\_REG;

u8 res16[3];

u32 res17[8];

struct s3c24x0\_usb\_dev\_fifos fifo[5];

u32 res18[11];

struct s3c24x0\_usb\_dev\_dmas ep1;

struct s3c24x0\_usb\_dev\_dmas ep2;

u8 res19[16];

struct s3c24x0\_usb\_dev\_dmas ep3;

struct s3c24x0\_usb\_dev\_dmas ep4;

#endif /\* \_\_BIG\_ENDIAN \*/

// struct s3c24x0\_usb\_dev\_fifos fifo[5];

// struct s3c24x0\_usb\_dev\_dmas dma[5];

};

4.修改**smdk2440.h**，加入USB Slave设备的配置：

//#undef CONFIG\_USE\_IRQ /\* we don't need IRQ/FIQ stuff \*/

#define CONFIG\_USB\_DEVICE 1

#ifdef CONFIG\_USB\_DEVICE

#define CONFIG\_USE\_IRQ 1

#endif

5.修改**arch/arm/cpu/arm920t/start.S**文件，加入USB中断相关操作：

#ifdef CONFIG\_USE\_IRQ

.align 5

irq**:**

**/\***

get\_irq\_stack

irq\_save\_user\_regs

bl do\_irq

irq\_restore\_user\_regs

**\*/**

/\* use IRQ for USB and DMA \*/

sub lr, lr, #4 @ the return address

ldr sp, IRQ\_STACK\_START @ the stack for irq

stmdb sp!, { r0-r12,lr } @ save registers

ldr lr, =int\_return @ set the return addr

ldr pc, =IRQ\_Handle @ call the isr

int\_return:

ldmia sp!, { r0-r12,pc }^ @ return from interrupt

.align 5

6.修改**arch/arm/lib/board.c**，加入usb slave初始化函数：

/\* enable exceptions \*/

enable\_interrupts **();**

**#ifdef CONFIG\_USB\_DEVICE**

**usb\_init\_slave();**

**#endif**

/\* Perform network card initialisation if necessary \*/

**#ifdef CONFIG\_DRIVER\_TI\_EMAC**

/\* XXX: this needs to be moved to board init \*/

extern void davinci\_eth\_set\_mac\_addr (const u\_int8\_t \*addr);

7.在**common**目录中加入usbslave命令文件(cmd\_usbslave.c)，将写好的cmd\_usbslave.c文件放到common目录下，修改同目录下Makefile文件，加入编译选项：

COBJS-$(CONFIG\_MODEM\_SUPPORT) += modem.o

COBJS-$(CONFIG\_UPDATE\_TFTP) += update.o

COBJS-$(CONFIG\_USB\_KEYBOARD) += usb\_kbd.o

COBJS-$(CONFIG\_USB\_DEVICE) += cmd\_usbslave.o

cmd\_usbslave.c代码如下：

#include <common.h>

#include <command.h>

#include <asm/byteorder.h>

#ifdef CONFIG\_USB\_DEVICE

#ifdef CONFIG\_USE\_IRQ

#define IRQ\_STACK\_START (\_armboot\_start - CONFIG\_SYS\_MALLOC\_LEN - CONFIG\_SYS\_GBL\_DATA\_SIZE - 4)

#define FIQ\_STACK\_START (IRQ\_STACK\_START - CONFIG\_STACKSIZE\_IRQ)

#define FREE\_RAM\_END (FIQ\_STACK\_START - CONFIG\_STACKSIZE\_FIQ - CONFIG\_STACKSIZE)

#define FREE\_RAM\_SIZE (FREE\_RAM\_END - PHYS\_SDRAM\_1)

#else

#define FREE\_RAM\_END (\_armboot\_start - CONFIG\_SYS\_MALLOC\_LEN - CONFIG\_SYS\_GBL\_DATA\_SIZE - 4 - CONFIG\_STACKSIZE)

#define FREE\_RAM\_SIZE (FREE\_RAM\_END - PHYS\_SDRAM\_1)

#endif

int g\_bUSBWait = 1;

u32 g\_dwDownloadLen = 0;

extern int download\_run;

extern volatile unsigned int dwUSBBufBase;

extern volatile unsigned int dwUSBBufSize;

extern \_\_u32 usb\_receive(char \*buf, size\_t len, unsigned int wait);

int do\_usbslave (cmd\_tbl\_t \*cmdtp, int flag, int argc, char \*argv[])

{

int i;

size\_t len = ~0UL;

char buf[32];

/\* download\_run为1时表示将文件保存在USB Host发送工具dnw指定的位置

\* download\_run为0时表示将文件保存在参数argv[2]指定的位置

\* 要下载程序到内存，然后直接运行时，要设置download\_run=1，这也是这个参数名字的来由

\*/

download\_run = 1;

switch (argc) {

case 1:

{

break;

}

case 2:

{

g\_bUSBWait = (int)simple\_strtoul(argv[1], NULL, 16);

break;

}

case 3:

{

g\_bUSBWait = (int)simple\_strtoul(argv[1], NULL, 16);

load\_addr = simple\_strtoul(argv[2], NULL, 16);

download\_run = 0;

break;

}

default:

{

printf ("Usage:\n%s\n", cmdtp->usage);

return 1;

}

}

dwUSBBufBase = load\_addr;

dwUSBBufSize = (FREE\_RAM\_SIZE&(~(0x80000-1)));

if (g\_bUSBWait)

len = FREE\_RAM\_SIZE;

g\_dwDownloadLen = usb\_receive(dwUSBBufBase, len, g\_bUSBWait);

sprintf(buf, "%X", g\_dwDownloadLen);

setenv("filesize", buf);

return 0;

}

U\_BOOT\_CMD(

usbslave, 3, 0, do\_usbslave,

"usbslave - get file from host(PC)\n",

"[wait] [loadAddress]\n"

"\"wait\" is 0 or 1, 0 means for return immediately, not waits for the finish of transferring\n"

);

#endif

8.初始化USB中断管脚：

在TX2440开发板上USB DEVICE中断管脚接的是S3C2440处理器的GPG12，应把这个管脚设为输出，初始电平为0，在初始化usbslave时，再将它拉高。

修改**board/samsung/smdk2440/smdk2440.c**文件，在board\_init函数中修改：

gpio**->**GPGCON **=** 0xFD95FFBA**;**

gpio**->**GPGUP **=** 0x0000FFFF**;**

**gpio->GPGDAT &= ~(1<<12);**

在**drivers/usb/slave/usbinit.c**中的usb\_slave\_init()函数中将GPG12管脚拉高：

writel**((**readl**(&**gpioregs**->**GPGDAT**) | (**1**<<**12**)), &**gpioregs**->**GPGDAT**);**

由于在smdk2440.h中定义了CONFIG\_USE\_IRQ选项，在board.c文件会调用interrupt\_init函数（位于arch/arm/lib/interrupts.c），用于初始化平台中断，其中返回arch\_interrupt\_init函数。但我们的代码里没有实现此函数，需要自己实现。

修改S3C2440平台的interrupt文件，路径为：arch/arm/cpu/arm920t/s3c24x0/interrupts.c，创建一个返回0的arch\_interrupt\_init函数

#include <asm/arch/s3c24x0\_cpu.h>

#include <asm/proc-armv/ptrace.h>

#include <asm/io.h>

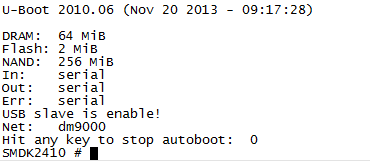
int arch\_interrupt\_init **(**void**)**

**{**

**return**0**;**

**}**

再次编译，将镜像烧入开发板，显示：

****

使用命令：usbslave或usbslave 1 0x30000000（下载地址可省略，使用DNW所设置的地址）

如果usb成功连接，会提示：

****

如果连接失败会提示：

USB host is not connected yet

用DNW软件选择要下载的文件，然后就可以对下载的内容进行操作了，比如写入nandflash中。

# 烧写YAFFS2文件系统支持

uboot默认不支持烧写YAFFS2文件系统镜像，我们只需要加入烧写yaffs文件系统的命令，并修改nand的驱动中写操作的处理方式。

1.在**include/configs/smdk2440.h**中加入支持YAFFS的配置：

/\* YAFFS \*/

#define ENABLE\_CMD\_NAND\_YAFFS 1

#define ENABLE\_CMD\_NAND\_YAFFS\_SKIPFB 1

2.加入烧写YAFFS的命令支持，修改**common/cmd\_nand.c**文件：

在do\_nand函数中的读写操作中加入yaffs的选项

if (read)

ret = nand\_read\_skip\_bad(nand, off, &size, (u\_char \*)addr);

else

ret = nand\_write\_skip\_bad(nand, off, &size,(u\_char \*)addr);

#if defined(ENABLE\_CMD\_NAND\_YAFFS)

}else if ( s != NULL && (!strcmp(s, ".yaffs") || !strcmp(s, ".yaffs2"))){

if(read) {

printf("nand read.yaffs[2] is not provide temporarily!");

} else {

nand->rw\_oob = 1;

#if defined(ENABLE\_CMD\_NAND\_YAFFS\_SKIPFB)

nand->skipfirstblk = 1;

#else

nand->skipfirstblk = 0;

#endif

ret = nand\_write\_skip\_bad(nand,off,&size,(u\_char \*)addr);

#if defined(ENABLE\_CMD\_NAND\_YAFFS\_SKIPFB)

nand->skipfirstblk = 0;

#endif

nand->rw\_oob = 0;

}

#endif

} else if (!strcmp(s, ".oob")) {

/\* out-of-band data \*/

mtd\_oob\_ops\_t ops = {

.oobbuf = (u8 \*)addr,

.ooblen = size,

.mode = MTD\_OOB\_RAW

};

在U\_BOOT\_CMD中加入yaffs的选项

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

"nand read[.yaffs[2]] is not provide temporarily!\n"

"nand write[.yaffs[2]] addr off size - write the 'size' byte yaffs image starting\n"

" at offset 'off' from memory address 'addr' (.yaffs2 for 2048 + 16 NAND)\n"

**#endif**

加入了烧写yaffs文件系统命令后，我们即可执行nand write.yaffs命令来烧写yaffs文件系统镜像。

3.修改**include/linux/mtd/mtd.h**文件：

在mtd\_info结构体中加入nand驱动中用到的两个变量：

structmtd\_info **{**

**u\_char rw\_oob;**

**u\_char skipfirstblk;**

u\_char type**;**

4.在**drivers/mtd/nand/nand\_util.c**文件中修改nand\_write\_skip\_bad()函数，添加对OOB操作的支持

size\_t len\_incl\_bad**;**

u\_char **\***p\_buffer **=** buffer**;**

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

**if(nand->rw\_oob==1) {**

**size\_t oobsize = nand->oobsize;**

**size\_t datasize = nand->writesize;**

**int datapages = 0;**

**if (((\*length)%(nand->oobsize+nand->writesize)) != 0) {**

**printf("Attempt to write error length data!\n");**

**return -EINVAL;**

**}**

**datapages = \*length/(datasize+oobsize);**

**\*length = datapages\*datasize;**

**left\_to\_write = \*length;**

**}**

**#endif**

**……**

if ((offset + len\_incl\_bad) > nand->size) {

printf ("Attempt to write outside the flash area\n");

return -EINVAL;

}

#if !defined(ENABLE\_CMD\_NAND\_YAFFS)

if (len\_incl\_bad == \*length) {

rval = nand\_write (nand, offset, length, buffer);

if (rval != 0)

printf ("NAND write to offset %llx failed %d\n",

offset, rval);

return rval;

}

#endif

while (left\_to\_write > 0) {

size\_t block\_offset = offset & (nand->erasesize - 1);

size\_t write\_size;

**……**

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

**if(nand->skipfirstblk == 1) {**

**nand->skipfirstblk = 0;**

**printf("Skip the first good block %llx\n",offset & ~(nand->erasesize - 1));**

**offset += nand->erasesize - block\_offset;**

**continue;**

**}**

**#endif**

**if(**left\_to\_write **< (**nand**->**erasesize **-** block\_offset**))**

write\_size **=** left\_to\_write**;**

**else**

write\_size **=** nand**->**erasesize **-** block\_offset**;**

**printf("\rWriting at 0x%08lx -- ",offset);**

**……**

left\_to\_write **-=** write\_size**;**

offset**+=** write\_size**;**

//p\_buffer += write\_size;

**#ifdefined(ENABLE\_CMD\_NAND\_YAFFS)**

**if(nand->rw\_oob==1) {**

**p\_buffer += write\_size+(write\_size/nand->writesize\*nand->oobsize);**

**} else {**

**p\_buffer += write\_size;**

**}**

**#else**

p\_buffer += write\_size;

**#endif**

**}**

**return0;**

**}**

5.修改**drivers/mtd/nand/nand\_base.c**文件：

在nand\_write函数中加入：

static int nand\_write**(**struct mtd\_info **\***mtd**,** loff\_t to**,** size\_t len**,**

size\_t **\***retlen**,** const uint8\_t **\***buf**)**

**{**

structnand\_chip **\***chip **=** mtd**->**priv**;**

intret**;**

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

**int oldopsmode = 0;**

**u\_char data\_buffer[len] ,oob\_buffer[len/(mtd->writesize)\*mtd->oobsize];**

**if(mtd->rw\_oob==1) {**

**size\_t oobsize = mtd->oobsize;**

**size\_t datasize = mtd->writesize;**

**int i = 0;**

**uint8\_t oobtemp[oobsize];**

**int datapages = 0;**

**datapages = len/(datasize);**

**for(i=0;i<(datapages);i++) {**

**memcpy((void \*)(data\_buffer+i\*datasize),(void \*)(buf+datasize\*i+i\*oobsize),datasize);**

**memcpy((void \*)(oob\_buffer+i\*oobsize),(void \*)(buf+datasize\*(i+1)+i\*oobsize),oobsize);**

**}**

**}**

**#endif**

/\* Do not allow reads past end of device \*/

if ((to + len) > mtd->size)

return -EINVAL;

if (!len)

return 0;

nand\_get\_device(chip, mtd, FL\_WRITING);

chip->ops.len = len;

**//chip->ops.datbuf = (uint8\_t \*)buf;**

**//chip->ops.oobbuf = NULL;**

**if(mtd->rw\_oob!=1) {**

**chip->ops.datbuf = (uint8\_t \*)buf;**

**}else{**

**chip->ops.datbuf = (uint8\_t \*)data\_buffer;**

**}**

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

**if(mtd->rw\_oob!=1) {**

**chip->ops.oobbuf = NULL;**

**} else {**

**chip->ops.oobbuf = (uint8\_t \*)oob\_buffer;**

**chip->ops.ooblen = mtd->oobsize;**

**oldopsmode = chip->ops.mode;**

**chip->ops.mode = MTD\_OOB\_RAW;**

**}**

**#else**

**chip->ops.oobbuf = NULL;**

**#endif**

ret = nand\_do\_write\_ops(mtd, to, &chip->ops);

\*retlen = chip->ops.retlen;

nand\_release\_device(mtd);

**#if defined(ENABLE\_CMD\_NAND\_YAFFS)**

**chip->ops.mode = oldopsmode;**

**#endif**

return ret;

}

通过以上的修改，uboot就可以支持烧写yaffs2文件系统镜像了。

# 使U-BOOT支持jffs2文件系统

在**include/configs/smdk2440.h**中添加代码

/\* JFFS2 \*/

#define CONFIG\_CMD\_JFFS2

#define CONFIG\_JFFS2\_NAND 1

#define CONFIG\_JFFS2\_CMDLINE 1

#define CONFIG\_JFFS2\_DEV "nand0"

# 实现Tab键命令自动补齐、上下箭头键调用历史命令

**在include/configs/smdk2440.h**中添加代码

/\* auto complete \*/

#define CONFIG\_CMDLINE\_EDITING

#define CONFIG\_AUTO\_COMPLETE

# 修改命令提示符

**在include/configs/smdk2440.h**中添加代码

将#define CONFIG\_SYS\_PROMPT “SMDK2410 # ”宏定义中SMDK2410改为SMDK2440

# 添加启动菜单

在command目录中新建cmd\_menu.c，代码如下：

#include <common.h>

#include <command.h>

static char awaitkey(unsigned long delay, int \*error\_p)

{

int i;

if(delay == -1)

{

while(1)

{

if(tstc()) //we got a key press

return getc();

}

}

else

{

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

{

if(tstc()) //we got a key press

return getc();

udelay(10 \* 1000);

}

}

if(error\_p)

\*error\_p = -1;

return 0;

}

void main\_menu\_usage(void)

{

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n");

if(BootFlag == 1)

{

printf("\*\*\*\*\*\*\*\* S3C2440 NOR启动UBOOT下载模式\*\*\*\*\*\*\*\*\r\n");

}

else

{

printf("\*\*\*\*\*\*\*\* S3C2440 NAND启动UBOOT下载模式\*\*\*\*\*\*\*\*\r\n");

}

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n");

printf("[1] 下载u-boot 写入到Nand Flash\r\n");

printf("[2] 下载zImage内核镜像写入到Nand Flash\r\n");

printf("[3] 下载yaffs2文件系统写入到Nand Flash\r\n");

printf("[4] 重新启动\r\n");

printf("[ESC] 退出菜单\r\n");

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\r\n");

if(BootFlag == 1)

{

printf("[o] 下载u-boot 写入到Nor Flash\r\n");

}

printf("输入选择：");

}

void menu\_shell(void)

{

char c;

char cmd\_buf[200];

while(1)

{

main\_menu\_usage();

c = awaitkey(-1, NULL);

printf("%c\n",c);

switch(c)

{

case '1':

{

strcpy(cmd\_buf,"usbslave 1 0x30000000; nand erase bios; nand write 0x30000000 bios");

run\_command(cmd\_buf,0);

break;

}

case '2':

{

strcpy(cmd\_buf,"usbslave 1 0x30000000; nand erase kernel; nand write 0x30000000 kernel");

run\_command(cmd\_buf,0);

break;

}

case '3':

{

strcpy(cmd\_buf,"usbslave 1 0x30000000; nand erase root; nand write.yaffs 0x30000000 root $(filesize)");

run\_command(cmd\_buf,0);

break;

}

case '4':

{

strcpy(cmd\_buf,"reset");

run\_command(cmd\_buf,0);

break;

}

case0x1b:

{

return;

break;

}

case 'o':

{

strcpy(cmd\_buf,"usbslave 1 0x30000000; protect off all; erase 0 +0x100000;cp.b 0x30000000 0 0x100000");

run\_command(cmd\_buf,0);

break;

}

}

}

}

int do\_menu( cmd\_tbl\_t \*cmdtp, /\*bd\_t \*bd,\*/ int flag, int argc, char \*argv[] )

{

/\*printf("<NOT YET IMPLEMENTED>\n"); \*/

menu\_shell();

return 0;

}

U\_BOOT\_CMD(

menu, 1, 1, do\_menu,

"Download Menu",

""

);

修改command目录Makefile，添加cmd\_menu.o

COBJS-y += main.o

COBJS-y += cmd\_menu.o

由于cmd\_menu.c中使用了**arch/arm/cpu/arm920t/start.S**定义的变量BootFlag，所以在**include/asm/ u-boot-arm.h**中添加这个变量的声明。

extern ulong IRQ\_STACK\_START; /\* top of IRQ stack \*/

extern ulong FIQ\_STACK\_START; /\* top of FIQ stack \*/

extern ulong BootFlag;

/\* cpu/.../cpu.c \*/

int cpu\_init(void);

int cleanup\_before\_linux(void);

在main.c中调用menu命令：

main\_loop函数

# ifdef CONFIG\_AUTOBOOT\_KEYED

disable\_ctrlc(prev); /\* restore Control C checking \*/

# endif

}

run\_command("menu",0);

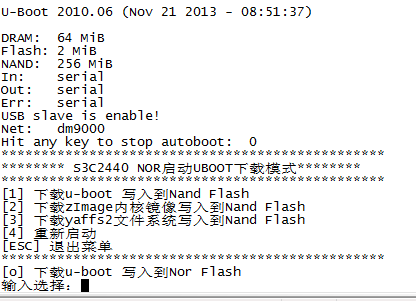
# ifdef CONFIG\_MENUKEY

if (menukey == CONFIG\_MENUKEY) {

s = getenv("menucmd");

if (s) {

编译生成u-boot.bin，效果如下：

****

# 添加LCD驱动和LOGO制作

Uboot中LCD的驱动位置为：drivers/video/，核心驱动文件是videomodes.c和cfb\_console.c，其他的是各平台的LCD硬件驱动，我们需要加入2440的LCD控制器驱动。

1．在drivers/video目录中新增s3c2440\_fb.c，代码如下：

#include <common.h>

#if defined(CONFIG\_VIDEO\_S3C2440)

#include <video\_fb.h>

#include "videomodes.h"

#include <asm/arch/s3c24x0\_cpu.h>

#include <asm/io.h>

/\*

\* Export Graphic Device

\*/

GraphicDevice smi;

#define VIDEO\_MEM\_SIZE 0x200000

#define MVAL (0)

#define MVAL\_USED (0) //0=each frame 1=rate by MVAL

#define INVVDEN (1) //0=normal 1=inverted

#define BSWP (0) //Byte swap control

#define HWSWP (1) //Half word swap control

#define VBPD (12)

#define VFPD (21)

#define VSPW (9)

#define HBPD (35)

#define HFPD (209)

#define HSPW (9)

#define CLKVAL\_TFT (0)

void lcd\_enable(void)

{

struct s3c24x0\_gpio \* const gpio = s3c24x0\_get\_base\_gpio();

struct s3c24x0\_lcd \* const lcd = s3c24x0\_get\_base\_lcd();

gpio->GPGUP = gpio->GPGUP & ((~(1 << 4)) | (1 << 4));

gpio->GPGCON = gpio->GPGCON & ((~( 3 << 8)) | ( 3 << 8));

gpio->GPGDAT = gpio->GPGDAT | (1 << 4 );

lcd->LCDCON5 = lcd->LCDCON5 & ((~( 1 << 3)) | (1 << 3)); // PWREN

lcd->LCDCON5 = lcd->LCDCON5 & ((~( 1 << 5)) | (0 << 5)); // INVPWREN

lcd->LCDCON1 |= 1;

gpio->GPBUP = gpio->GPBUP & ((~(1 << 1)) | (1 << 1));

gpio->GPBCON = gpio->GPBCON & ((~( 3 << 2)) | ( 3 << 2));

gpio->GPBDAT = gpio->GPBDAT | (1 << 1 );

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*

\* Init video chip with common Linux graphic modes (lilo)

\*/

void \*video\_hw\_init (void)

{

struct s3c24x0\_lcd \* const lcd = s3c24x0\_get\_base\_lcd();

GraphicDevice \*pGD = (GraphicDevice \*)&smi;

int videomode;

int bppmode;

unsigned long t1, hsynch, vsynch;

char \*penv;

int tmp, i, bits\_per\_pixel;

struct ctfb\_res\_modes \*res\_mode;

struct ctfb\_res\_modes var\_mode;

tmp = 0;

videomode = CONFIG\_SYS\_DEFAULT\_VIDEO\_MODE;

/\* get video mode via environment \*/

if ((penv = getenv ("videomode")) != NULL) {

/\* deceide if it is a string \*/

if (penv[0] <= '9') {

videomode = (int) simple\_strtoul (penv, NULL, 16);

tmp = 1;

}

}

else

{

tmp = 1;

}

if (tmp) {

/\* parameter are vesa modes \*/

/\* search params \*/

for (i = 0; i < VESA\_MODES\_COUNT; i++) {

if (vesa\_modes[i].vesanr == videomode)

break;

}

if (i == VESA\_MODES\_COUNT) {

printf ("no VESA Mode found, switching to mode 0x%x ", CONFIG\_SYS\_DEFAULT\_VIDEO\_MODE);

i = 0;

}

res\_mode = (struct ctfb\_res\_modes \*) &res\_mode\_init[vesa\_modes[i].resindex];

bits\_per\_pixel = vesa\_modes[i].bits\_per\_pixel;

}

else

{

res\_mode = (struct ctfb\_res\_modes \*) &var\_mode;

bits\_per\_pixel = video\_get\_params (res\_mode, penv);

}

/\* calculate hsynch and vsynch freq (info only) \*/

t1 = (res\_mode->left\_margin + res\_mode->xres + res\_mode->right\_margin + res\_mode->hsync\_len) / 8;

t1 \*= 8;

t1 \*= res\_mode->pixclock;

t1 /= 1000;

hsynch = 1000000000L / t1;

t1 \*= (res\_mode->upper\_margin + res\_mode->yres + res\_mode->lower\_margin + res\_mode->vsync\_len);

t1 /= 1000;

vsynch = 1000000000L / t1;

/\* fill in Graphic device struct \*/

sprintf (pGD->modeIdent, "%dx%dx%d %ldkHz %ldHz", res\_mode->xres, res\_mode->yres, bits\_per\_pixel, (hsynch / 1000), (vsynch / 1000));

pGD->winSizeX = res\_mode->xres;

pGD->winSizeY = res\_mode->yres;

pGD->plnSizeX = res\_mode->xres;

pGD->plnSizeY = res\_mode->yres;

switch (bits\_per\_pixel)

{

case 8:

pGD->gdfBytesPP = 1;

pGD->gdfIndex = GDF\_\_8BIT\_INDEX;

bppmode = 11;

break;

case 16:

pGD->gdfBytesPP = 2;

pGD->gdfIndex = GDF\_16BIT\_565RGB;

bppmode = 12;

break;

case 24:

pGD->gdfBytesPP = 3;

pGD->gdfIndex = GDF\_24BIT\_888RGB;

bppmode = 13;

break;

}

pGD->frameAdrs = LCD\_VIDEO\_ADDR;

pGD->memSize = VIDEO\_MEM\_SIZE;

lcd->LCDCON1 = (CLKVAL\_TFT << 8) | (MVAL\_USED << 7) | (3 << 5) | (bppmode << 1) | 0;

lcd->LCDCON2 = (VBPD << 24) | (pGD->winSizeY-1 << 14) | (VFPD << 6) | (VSPW);

lcd->LCDCON3 = (HBPD << 19) | (pGD->winSizeX-1 << 8) | (HFPD);

lcd->LCDCON4 = (MVAL << 8) | (HSPW);

lcd->LCDCON5 = (1 << 11) | (0 << 10) | (1 << 9) | (1 << 8) | (0 << 7) | (0 << 6) | (1 << 3) | (BSWP << 1) | (HWSWP);

lcd->LCDINTMSK |= (3);

lcd->LPCSEL &= (~7);

lcd->TPAL = 0x0;

writel((pGD->frameAdrs >> 1), &lcd->LCDSADDR1);

/\* This marks the end of the frame buffer. \*/

writel((((readl(&lcd->LCDSADDR1))&0x1fffff) + (pGD->winSizeX+0) \* pGD->winSizeY), &lcd->LCDSADDR2);

writel((pGD->winSizeX & 0x7ff), &lcd->LCDSADDR3);

/\* Clear video memory \*/

memset((void \*)pGD->frameAdrs, 0, pGD->memSize);

lcd\_enable();

return ((void\*)&smi);

}

void video\_set\_lut (unsigned int index, /\* color number \*/

unsigned char r, /\* red \*/

unsigned char g, /\* green \*/

unsigned char b /\* blue \*/

)

{

}

#endif /\* CONFIG\_VIDEO\_S3C2440 \*/

修改video目录下Makefile文件

COBJS-$(CONFIG\_VIDEO\_VCXK) += bus\_vcxk.o

COBJS-$(CONFIG\_VIDEO\_S3C2440) += s3c2440\_fb.o videomodes.o cfb\_console.o

COBJS := $(COBJS-y)

2.修改include/configs/smdk2440.h，添加下面的宏定义

/\* LCD settings \*/

#define CONFIG\_VIDEO

//使用video lcd驱动，可以调用cfb\_console.c中的drv\_video\_init函数

#define CONFIG\_VIDEO\_S3C2440

//使用2440的lcd驱动，可以编译s3c2440\_fb.c文件

#define CONFIG\_VIDEO\_LOGO

//开启显示LOGO功能，默认显示linux logo

#define CONFIG\_VIDEO\_BMP\_LOGO

//如果定义了BMP LOGO，则显示bmp logo，否则显示linux logo

#define VIDEO\_FB\_16BPP\_WORD\_SWAP

//是否开启16位字节交换功能

#define CONFIG\_SYS\_VIDEO\_LOGO\_MAX\_SIZE (800\*480+1024+100)

//支持的最大LOGO尺寸

#define LCD\_VIDEO\_ADDR 0x33d00000

//显存的地址

#define CONFIG\_CMD\_BMP

//使用BMP命令

#define CONFIG\_CFB\_CONSOLE

//同CONFIG\_VIDEO

#define CONFIG\_SYS\_CONSOLE\_INFO\_QUIET

//不显示console info

/\* for PC-keyboard \*/

#define VIDEO\_KBD\_INIT\_FCT 0

#define VIDEO\_TSTC\_FCT serial\_tstc

#define VIDEO\_GETC\_FCT serial\_getc

3.修改videomodes.c和videomodes.h文件，加入LCD参数列表：

在videomodes.h中有两个结构体，ctfb\_vesa\_modes描述了所使用的显示器的Vesa number、resolution index和BPP；ctfb\_res\_modes描述了显示器的属性参数：resolution、pixclock、Timing value等。

在videomodes.c中创建的vesa\_modes和res\_mode\_init结构体，列出了所支持的显示器，我们需要仿照这两个结构体中的内容加入对我们的LCD设备的支持。

修改drivers/video/videomodes.c

{0x31A, RES\_MODE\_1280x1024, 16},

{0x31B, RES\_MODE\_1280x1024, 24},

{0x213, RES\_MODE\_800x480, 16},

};

RES\_MODE\_800x480前的数据0x213与前面的不同即可。

const struct ctfb\_res\_modes res\_mode\_init[RES\_MODES\_COUNT] = {

/\* x y pixclk le ri up lo hs vs s vmode \*/

{640, 480, 39721, 40, 24, 32, 11, 96, 2, 0, FB\_VMODE\_NONINTERLACED},

{800, 600, 27778, 64, 24, 22, 1, 72, 2, 0, FB\_VMODE\_NONINTERLACED},

{1024, 768, 15384, 168, 8, 29, 3, 144, 4, 0, FB\_VMODE\_NONINTERLACED},

{960, 720, 13100, 160, 40, 32, 8, 80, 4, 0, FB\_VMODE\_NONINTERLACED},

{1152, 864, 12004, 200, 64, 32, 16, 80, 4, 0, FB\_VMODE\_NONINTERLACED},

{1280, 1024, 9090, 200, 48, 26, 1, 184, 3, 0, FB\_VMODE\_NONINTERLACED},

{800, 480, 18890, 35, 209, 12, 21, 9, 9, 0, FB\_VMODE\_NONINTERLACED},

};

此处的参数是参照LCD手册修改的。

我们先来看一下各参数的含义：

struct ctfb\_res\_modes {

int xres; /\* 可见分辨率 \*/

int yres;

/\* 时序:所有值都以像素时钟为单位（当然除了像素时钟本身） \*/

int pixclock; /\*像素时钟（单位：皮秒） \*/

int left\_margin; /\*从行同步到图像左边沿的像素时钟数 \*/

int right\_margin; /\*从行同步到图像右边沿的像素时钟数 \*/

int upper\_margin; /\*从场同步到图像上边沿的行数 \*/

int lower\_margin; /\*从场同步到图像下边沿的行数 \*/

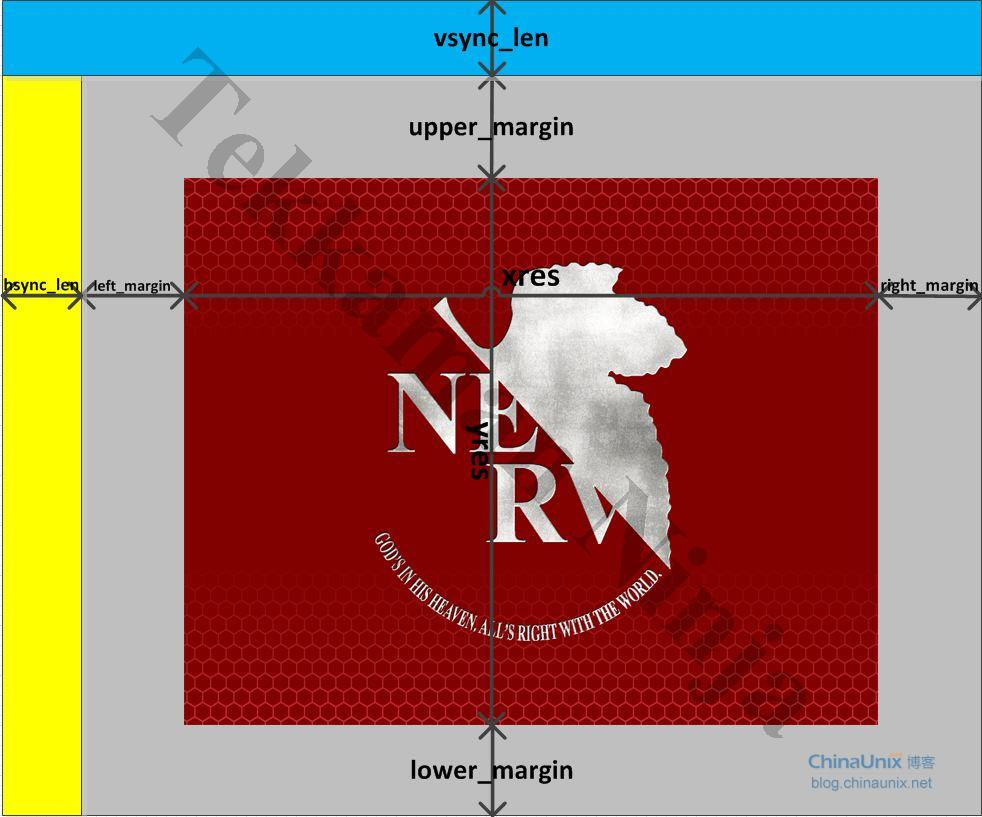
int hsync\_len; /\*行同步时间长度（像素时钟数VCLK） \*/

int vsync\_len; /\*场同步时间长度（行数） \*/

int sync; /\* see FB\_SYNC\_\* \*/

int vmode; /\* see FB\_VMODE\_\* \*/

};



VBPD(vertical back porch):表示在一帧图像开始时，垂直同步信号以后的无效行数，对应upper\_margin;

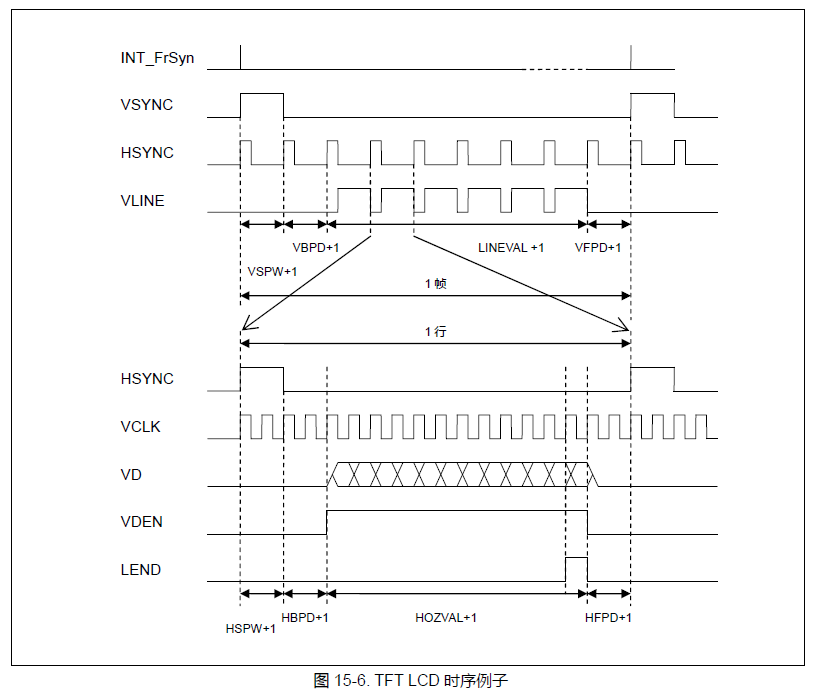
VFBD(vertical front porch):表示在一帧图像结束后，垂直同步信号以前的无效行数，对应lower\_margin；

VSPW(vertical sync pulse width):表示垂直同步脉冲的宽度，用行数计算，对应vsync\_len;

HBPD(horizontal back porch):表示从一个水平同步信号脉冲的结束 到 一行的有效数据开始之间的VCLK的个数，对应left\_margin;

HFPD(horizontal front porch):表示一行的有效数据结束 到 下一个水平同步信号脉冲的开始之间的VCLK的个数，对应right\_margin;

HSPW(horizontal sync pulse width):表示水平同步信号的宽度，用VCLK计算，对应hsync\_len



扫描一帧所需的时间：

=((VSPW+1)+(VBPD+1)+(LINEVAL+1)+(VFPD+1))个行时间

扫描一行所需的时间：

=((HSPW+1)+(HBPD+1)+(HOZVAL+1)+(HFPD+1))个VCLK时间

1个VCLK时间由LCDCON1内的CLKVAL决定：

=HCLK/[2\*(CLKVAL+1)]

sync=0说明水平同步信号（HSYNC）和垂直同步信号（VSYNC）都是下降沿有效。

pixclock的计算公式：

pixclock=1/dotclock（单位：皮秒），其中dotclock的视频硬件在显示器上绘制像素的速率。dotclock = ( x + HBPD + HFPD + HSPW) \* (y + VBPD + VFPD + VSPW) \* 整屏的刷新率。

整屏的刷新率计算方法如下：

我们以X35LCD为例，通过查文档知道，fclk=6.34MHz，那么画一个像素的时间就是 1/6.34us，如果屏的大小是240\*320，那么显示一行需要的时间就是240/6.34us，每条扫描线是240，但水平回扫（HFPD）和水平同步（HSPW）也需要时间，如果水平回扫和水平同步需要29个像素时钟，因此，画一条扫描线完整的时间就是(240+29)/6.34us。完整的屏有320根线，但垂直回扫（VFPD）和垂直同步（VSPW）也需要时间，如果垂直回扫和垂直同步需要13个像素时钟，那么画一个完整的屏需要(240+29)\*(320+13)/6.34us，所以整屏的刷新率就是6.34/((240+29)\*(320+13))MHz。

修改drives/video/videomodes.h

设定显示器的vesa number

#ifndef CONFIG\_SYS\_DEFAULT\_VIDEO\_MODE

#define CONFIG\_SYS\_DEFAULT\_VIDEO\_MODE 0x213

#endif

添加屏幕分辨率，增加一个自己的分辨率，COUNT加1

#define RES\_MODE\_1152x864 4

#define RES\_MODE\_1280x1024 5

#define RES\_MODE\_800x480 6

#define RES\_MODES\_COUNT 7

#define VESA\_MODES\_COUNT 20

编译生成u-boot.bin，显示器就可以显示uboot信息和logo了，三星平台logo默认是denx的图标。这时汉字显示是乱码，启动菜单建议使用英文。

制作自己的**LOGO**

在头文件中已经定义了CONFIG\_VIDEO\_LOGO和CONFIG\_VIDEO\_BMP\_LOGO，即使用bmp图片的方式显示logo。

所有的开机logo都放在了/tools/logos 之下，三星默认使用denx.bmp作为启动logo。

我们可以制作一个bmp图片，替换denx.bmp。

或者修改tools/Makefile文件，自定义logo的文件名

LOGO\_H = $(OBJTREE)/include/bmp\_logo.h

LOGO-$(CONFIG\_LCD\_LOGO) += $(LOGO\_H)

LOGO-$(CONFIG\_VIDEO\_LOGO) += $(LOGO\_H)

ifeq ($(LOGO\_BMP),)

LOGO\_BMP= logos/denx.bmp //修改红色部分

endif

ifeq ($(VENDOR),atmel)

LOGO\_BMP= logos/atmel.bmp

endif

注意：bmp图片位深度必须为8位。

在ubuntu系统下制作bmp图片的方法：

需要安装Netpbm工具包。

准备一张jpeg图片，通过命令行处理为8bit BMP图片。

处理脚本如下：

#!/bin/sh

#install Netpbm first

jpegtopnm $1 | ppmquant 31 | ppmtobmp -bpp 8 > $2

使用方法：（脚本名）（待处理的JPEG图片名）（输出文件名）

或者使用命令：

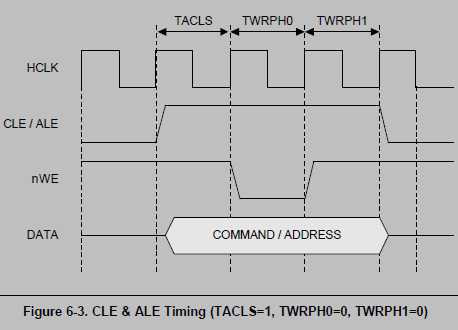
jpegtopnm xxx.jpg| ppmquant 31 | ppmtobmp -bpp 8 > xxx.bmp

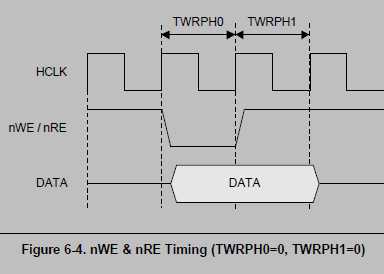
本次移植到此结束。

# 附录一

[深度分析NandFlash—控制器参数TACLS、TWRPH0和TWRPH1的确定](http://blog.chinaunix.net/uid-27097876-id-3310796.html)

这三个参数对于NandFlash的读写操作来说是比较重要的，没搞清楚这三个参数，后面的事就不用提了，那这三个参数到底是干什么的呢，我们怎么样去配置他们呢，今天我花了点时间研究了一下这三个参数。在datasheet中对他们的时序有下面两种图示，一种是写命令或者地址的情况，一种是读写数据的情况：



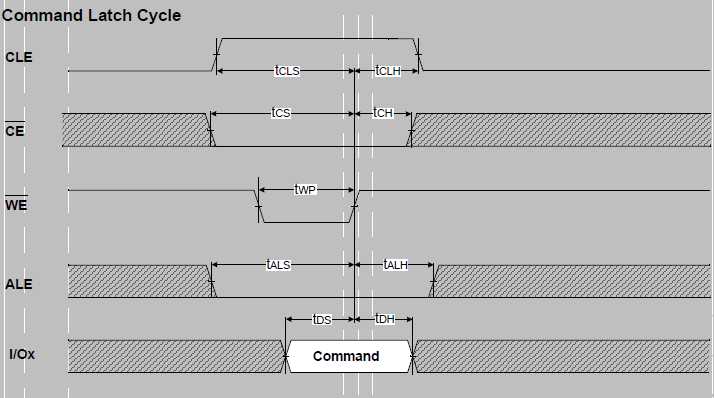


由上图可知，这三个参数控制的是Nand Flash 信号线CLE/ALE与写控制信号nWE的时序关系。

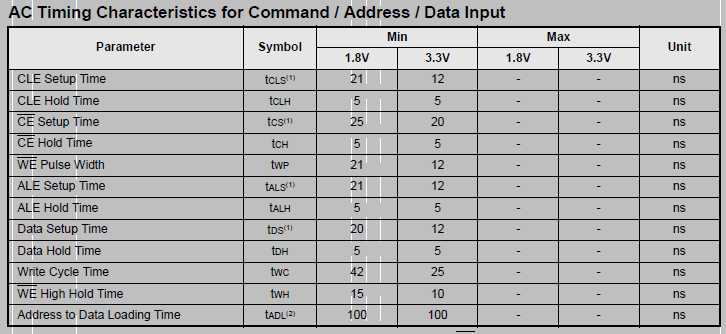
TACLS：表示CLT/ALE的建立时间(setup time)。

TWRPH0：表示nWE/nRE持续的时间。

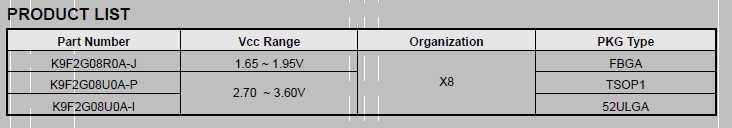
TWRPH1：表示写进去的数据起作用的时间(hold time)。



结合上面的图，就可以看出TACLS就相当于tCLS和tALS参数，TWRPH0就相当于tWP，而TWRPH1就相当于tCLH和TALH

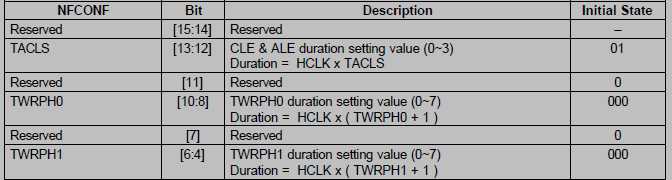


上图中有两种电压，1.8和3.3，我们的K9F2G08U0A工作时的电压时3.3V，所以应该选用3.3V对应的时序要求，NandFlash的工作电压在哪里看呢，还是datasheet，如下图所示：



从上图我们可以很容易得看出K9F2G08U0A工作时的电压是3.3V。

那这三个参数在哪里确定呢，它们的值又是怎么算呢？在NFCON这个寄存器中，还是datasheet：



由于我们的HCLK是100MHz，周期也就是10ns，可以设

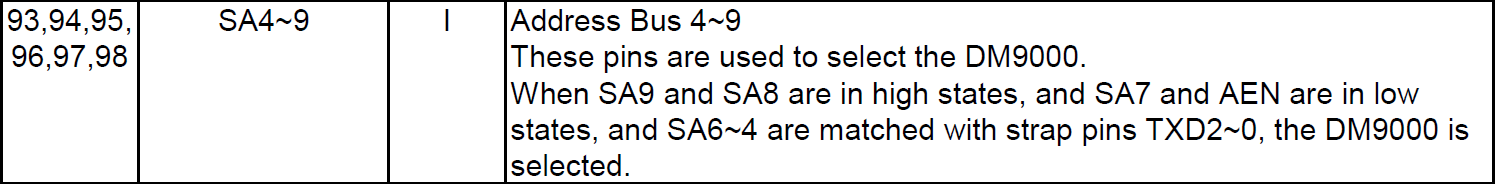
TACLS=2，即2\*10ns=20ns > 12ns；TWRPH0=1，及（1+1）\*10ns=20ns > 12ns；TWRPH1=0，即（0+1）\*10ns=10ns > 5ns

符合时序要求

# 附录二

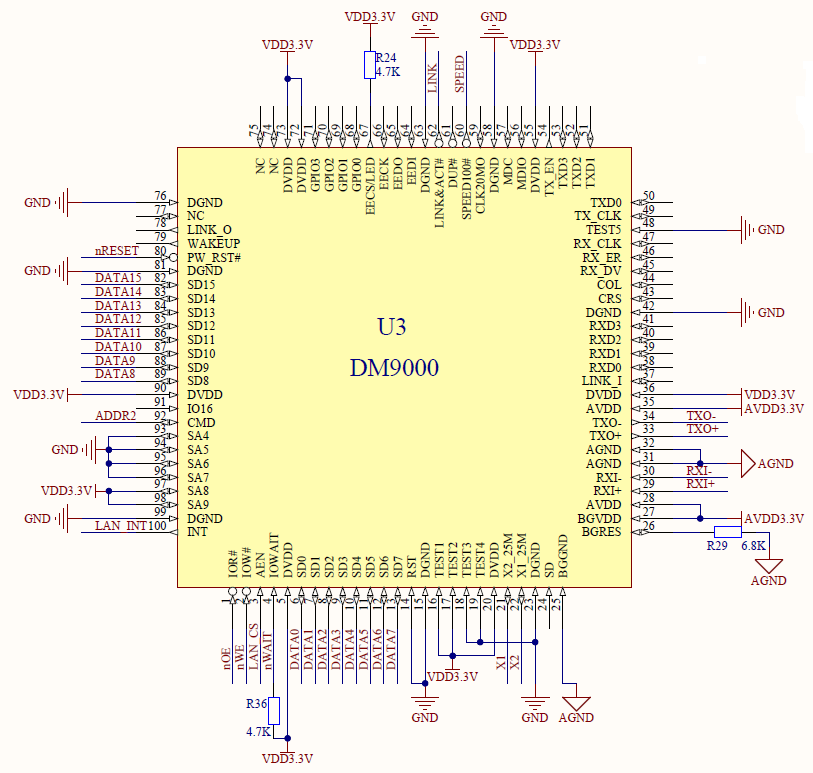
CONFIG\_DM9000\_BASE值的确定

对DM9000读写操作,首先对DM9000正确寻址。AEN(地址允许)是输入引脚片选信号。SA4～SA9是地址总线4～9位。由数据手册可以得出：

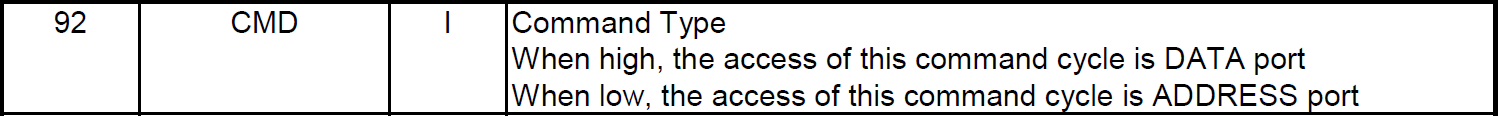


当AEN低且SA9和SA8高,而SA7、SA6、SA5、SA4为低时,则DM9000被选中。DM9000默认I/0基地址为300H。

我们先看看网卡原理图：



AEN连接到nGCS4，SA9、SA8接高电平，SA7~SA4接低电平，CMD接ADDR2。



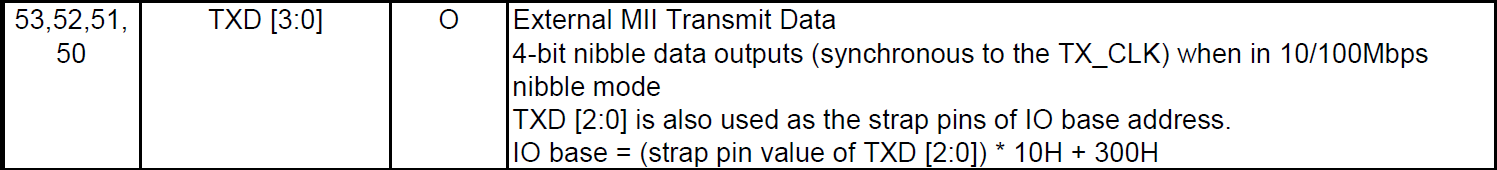
CMD引脚用于设置COMMAND模式,CMD为高时,选择数据端口。CMD为低时,选地址端口。

数据端口和地址端口的地址码由下式决定:

DM9000地址端口=高位片选地址+IO base+0H

DM9000数据端口=高位片选地址+IO base+4H

其中,高位片选地址由S3C2440的nGCS4提供,即为:0x20000000H。



由TXD[2:0]决定IO base，由于TXD[2:0]悬空未接，所以IO base = 300H。

所以address地址是0x20000300 (ADDR2 = 0), data地址是0x20000304 (ADDR2 = 1)。

访问0x20000304时，ADDR2为1，也就是CMD为1，为DATA输入输出。

# 附录三

全局配置选项说明

#define CONFIG\_ARM920T 1 架构为ARM920T

#define CONFIG\_S3C24X0 1 使用三星S3C24X0的SOC

#define CONFIG\_S3C2410 1 使用三星S3C2410的CPU（实际上我们使用S3C2440）

#define CONFIG\_SMDK2410 1 使用三星SMDK2410开发板

#define CONFIG\_SYS\_CLK\_FREQ 12000000 系统输入时钟为12MHz

#define USE\_920T\_MMU 1 使能MMU功能

#define CONFIG\_USB\_DEVICE 1 使用USB Device

#define CONFIG\_USE\_IRQ 1 使能USB中断

#define CONFIG\_SYS\_MALLOC\_LEN (CONFIG\_ENV\_SIZE + 128\*1024)

#define CONFIG\_SYS\_GBL\_DATA\_SIZE 128

#define CONFIG\_S3C24X0\_SERIAL 使用S3C24X0的串口设备

#define CONFIG\_SERIAL1 1 串口设备为UART1(对应着硬件的UART0)

#define CONFIG\_RTC\_S3C24X0 1 使用RTC设备

#define CONFIG\_ENV\_OVERWRITE

#define CONFIG\_BAUDRATE 115200 串口波特率115200

#define CONFIG\_BOOTDELAY 3 启动延时为3秒（按任意键进命令行模式等待的时间）

#define CONFIG\_BOOTARGS "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0 mem=64M" 命令行参数，在引导系统时传递给内核，设置root节点，init脚本，终端设备节点，内存容量等

#define CONFIG\_BOOTCOMMAND "nboot 0x32000000 kernel; bootm 0x32000000"

启动系统的命令，此语句赋给“bootcmd”变量

#define CONFIG\_ETHADDR 1a:2b:3c:4d:5e:6f 网卡的MAC地址

#define CONFIG\_NETMASK 255.255.255.0 网卡的子网掩码

#define CONFIG\_IPADDR 192.168.1.10 网卡的IP地址

#define CONFIG\_SERVERIP 192.168.1.99 服务器的IP地址

#define CONFIG\_SYS\_LONGHELP

#define CONFIG\_SYS\_PROMPT "[u-boot@XC2440]# " 命令行提示符

#define CONFIG\_SYS\_CBSIZE 256 Console I/O Buffer的大小

#define CONFIG\_SYS\_PBSIZE (CONFIG\_SYS\_CBSIZE+sizeof(CONFIG\_SYS\_PROMPT)+16)

打印缓冲区的大小，即串口输出调试信息的size

#define CONFIG\_SYS\_MAXARGS 16 最大的命令参数的个数

#define CONFIG\_SYS\_BARGSIZE CONFIG\_SYS\_CBSIZE 启动参数的缓冲区大小

#define CONFIG\_AMD\_LV160 1 使用的Norflash型号

#define PHYS\_FLASH\_SIZE 0x00200000 Norflash的容量

#define CONFIG\_SYS\_MAX\_FLASH\_SECT (35) Norflash的最大sectors数量

#define CONFIG\_STACKSIZE\_FIQ (4\*1024)

#define CONFIG\_NR\_DRAM\_BANKS 1 SDRAM使用的BANK数，我们仅使用BANK6

#define PHYS\_SDRAM\_1 0x30000000 内存的起始地址

#define PHYS\_SDRAM\_1\_SIZE 0x04000000 内存的大小

#define PHYS\_FLASH\_1 0x00000000 Norflash的起始地址

#define CONFIG\_SYS\_MEMTEST\_START 0x30000000 用于memtest的起始内存地址

#define CONFIG\_SYS\_MEMTEST\_END 0x33F00000 用于memtest的结束内存地址

#define CONFIG\_SYS\_LOAD\_ADDR 0x32000000 默认下载程序到内存中的起始地址

#define CONFIG\_SYS\_HZ 1000 用于计算定时时间等，此值应保持为1000

#define CONFIG\_STACKSIZE (128\*1024) 系统的栈的大小

#define CONFIG\_STACKSIZE\_IRQ (4\*1024)

#define CONFIG\_SYS\_FLASH\_BASE PHYS\_FLASH\_1 同上

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

环境变量的存储地址

#define CONFIG\_ENV\_IS\_IN\_NAND 1 环境变量存储在Nandflash中而不是Norflash

#define CONFIG\_ENV\_OFFSET 0x100000 存储环境变量地址的偏移量，用于计算ENV\_ADDR

#define CONFIG\_ENV\_SIZE 0x20000 存储环境变量地址的大小

#define CONFIG\_NAND\_S3C2410 1 使用三星NAND控制器

#define CONFIG\_SYS\_NAND\_BASE 0x4e000000 NAND控制器的寄存器基地址

#define CONFIG\_SYS\_MAX\_NAND\_DEVICE 1 使用的最大NAND器件数量

#define CONFIG\_MTD\_NAND\_VERIFY\_WRITE 1 使能写校验功能

#define CONFIG\_MTD\_DEVICE 1 使用MTD设备

#define MTDIDS\_DEFAULT "nand0=nandflash0" 用于设置mtdids变量

#define MTDPARTS\_DEFAULT 默认的MTD分区表，形式如下：

"mtdparts=nandflash0:1m@0(bios),"\

"128k(params),"\

"4m(kernel),"\

"-(root)"

# 附录四

Uboot常用命令

boot\_zImage 引导zImage内核镜像

用法：boot\_zImage

启动Nandflash中0x120000地址处的zImage内核，（此命令在代码中固定了启动地址，用户可修改）

bmp 在LCD上显示内存中的BMP图片

用法：bmp display <imageAddr> [x y]

例：bmp display 0x30000000

先将bmp图片下载到内存30000000地址中，再显示

cp 内存拷贝

用法：cp[.b .w .l] source target count

例：cp.b 0x30000000 0 0x100000

将内存30000000地址处开始的0x10000长度的数据拷贝到0地址

erase 擦除Norflash

用法：

erase start end 指定起始和结束地址

erase start +len 指定起始地址和擦除的长度

erase bank N 指定要擦除的BANK

erase all 擦除整个Flash

例：erase 0 +0x100000

擦除Norflash的从0地址处开始的长度为0x100000的区域

flinfo 显示Norflash memory banks信息

用法：flinfo

go 从内存中指定地址处运行程序

用法：go ram\_addr

例：go 0x30000000

将程序下载到内存中，再使用go命令运行

help 列出某个命令的描述和用法

用法：help commnd

例：help nand

可以列出此命令下的子命令的用法

loadx 通过串口xmodem协议下载二进制文件

用法：loadx [offset] [baud]

例：loadx 0x30000000 115200

此命令需要串口终端的xmodem协议支持

loady 通过串口ymodem协议下载二进制文件（同上）

menu 显示快捷菜单

用法：menu

mtdparts: 显示mtd分区表信息

用法：

mtdparts delall 删除所有MTD分区

mtdparts default （复位）使用默认分区表

ping：测试网络连接

用法：ping ip\_addr

例：ping 192.168.1.100

printenv 列出环境参数

用法：

printenv 列出所有参数的值

printenv name 列出指定参数name的值

例：printenv ipaddr

查看本机IP地址

protect：使能或禁止Norflash写保护

用法：protect on/off（参数较多，请使用help protect查看）

例：protect off all

使所有Norflash Banks可写

reset 重启uboot

用法：reset

saveenv 保存环境参数到flash

用法：saveenv

setenv 设置环境参数

用法：setenv name value

例：setenv bootdelay 2

设置启动延时为2秒

usbslave 使用usb device，从PC机下载程序到内存中

用法：usbslave 1 ram\_addr

例：usbslave 1 0x30000000

**nand Nandflash**操作子命令合集

使用help nand可以列出命令列表

nand erase 擦除nand

用法：nand erase [clean] [offset size]

例：

nand erase 擦除整个NAND

nand erase kernel 擦除kernel分区

nand erase 0x120000 0x400000 擦除指定地址和大小

nand read 从nand中读取数据

用法：nand read ram\_addr offset size

例：nand read 0x30000000 0x100000 0x200000

从NAND指定地址读取数据到内存中

nand write 向nand中写入数据

用法: nand write(.jffs2) ram\_addr flash\_addr filesize

例nane write 0x30000000 0x0 0x1000000

适用于烧写uboot、linux\_kernel和jffs2文件系统

烧写yaffs2文件系统

用法：nand write.yaffs ram\_addr flash\_addr filesize

例：nand write 0x30000000 rootfs $(filesize)

烧写yaffs2文件系统到rootfs分区$(filesize)表示下载到内存中的文件长度

nand bad 显示坏块信息

用法：nand bad

**USB**设备操作子命令合集

使用help usb可以列出命令列表

usb start 初始化USB设备，同usb reset

usb dev 查看检测到的usb设备信息

usb storage 查看USB存储设备的信息

usb tree 查看USB设备树

usb info 查看USB HOST设备信息

usb stop 停止（释放）USB设备

usb part 查看USB存储设备的分区

**FAT**文件系统命令合集

fatls：查看设备上的目录和文件

用法：fatls<interface><dev[:part]> [directory]

例：fatls usb 0

fatinfo：查看设备上的文件系统的信息

用法：fatinfo<interface><dev[:part]>

例：fatinfo usb 0

fatload：从文件系统中读取二进制文件加载到内存中

用法：fatload<interface><dev[:part]><addr><filename> [bytes]

例：fatload usb 0 0x30000000 u-boot.bin

从U盘上加载名为u-boot.bin的文件到内存的0x30000000地址处