

u-boot-2010.03 在 Mini2440 上的移植

一.移植环境

(1) Ubuntu9.04。

(2) 使用的开发板是友善之臂的 Mini2440，配有 Nor Flash (SST39VF1601) 大小为 2MB，NandFlash (K9F1GU08B) 128MB，每页的大小是 2KB。

(3) 交叉编译器的版本是 arm-linux-gcc-4.4.3。

(4) 移植的 u-boot 版本号为 u-boot-2010.03 其官方下载地址 <ftp://ftp.denx.de/pub/u-boot/>。
现在的用的这个版本，仍然不支持 2440 的处理器，我们必须以 smdk2410 为原型，在此基础上进行 U-Boot 的移植工作。

二.本次移植的功能特点：

- 支持 Nand Flash 读写
- 支持从 Nor/Nand Flash 启动
- 支持 DM9000 网卡
- 支持 Yaffs 文件系统
- 支持 tftp 下载

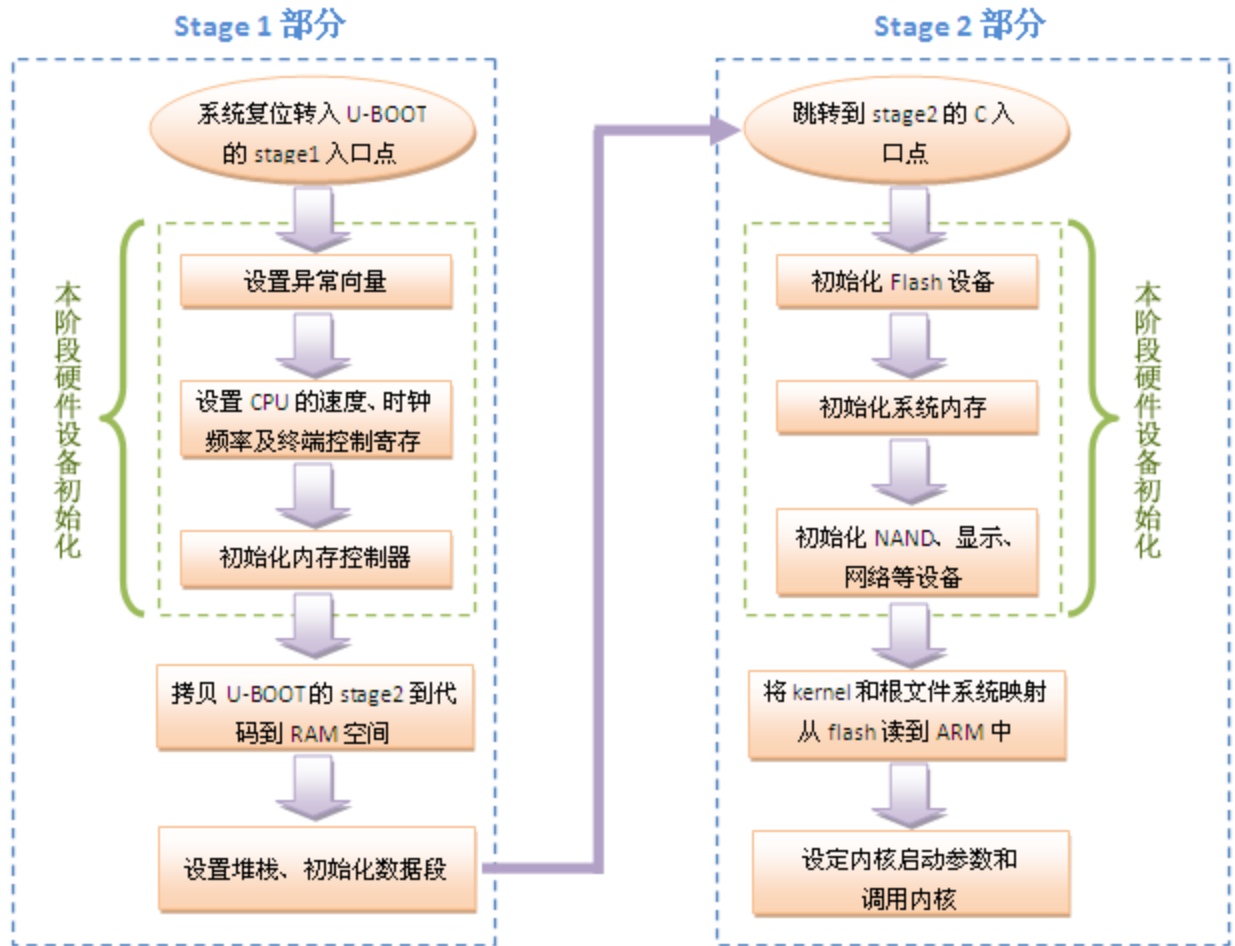
三.u-boot 主要的目录结构和启动流程

具体如下图：

u-boot-2009.08 主要目录及说明		
+	board	一些已经支持的开发板相关文件，主要包含 SDRAM、FLASH、网卡驱动等
	common	与处理器体系结构无关的通用代码，如：内存大小探测与故障检测
+	cpu	与处理器相关的文件。每个子目录中都包括 cpu.c、interrupt.c 和 start.S
	disk	Disk 驱动分区相关信息代码
	doc	U-boot 的说明文档
+	drivers	通用设备驱动程序，如：网卡、FLASH、串口、USB 总线等
+	examples	可在 U-boot 下运行的示例程序，如：hello_world.c、timer.c 等
+	fs	支持文件系统的文件，如：cramfs、fat、fdos、jffs2、registerfs 等
+	include	U-boot 头文件，还有各硬件平台的汇编文件，系统配置文件和对文件系统支持的文件
	lib_XXX	处理器体系结构相关文件，如：lib_arm 包含的是 ARM 体系结构的文件
	net	与网络功能相关的文件，如：bootp、nfs、tftp 等
+	post	上电自检文件目录
+	tools	用于创建 U-boot、S-Record 和 Bin 镜像文件的工具

U-BOOT 主要目录说明图

u-boot 的 stage1 代码通常放在 cpu/xxxx/start.S 文件中，他用汇编语言写成；u-boot 的 stage2 代码通常放在 lib_XXXX/board.c 文件中，他用 C 语言写成。各个部分的流程图如下：



U-BOOT 启动流程图

四、u-boot 的移植

（一）中断、时钟等设置

1、建立自己的开发板目录并测试编译环境

目前 u-boot 对很多 CPU 直接支持，可以查看 board 目录的一些子目录，如：board/samsung/目录下就是对三星一些 ARM 处理器的支持，有 smdk2400、smdk2410 和 smdk6400，但没有 2440，所以我们就在这里建立自己的开发板项目。

1) 因 2440 和 2410 的资源差不多，主频和外设有点差别，所以我们就在 board/samsung/下建立自己开发板的项目，取名叫 mini2440

```
#tar -jxvf u-boot-2010.03.tar.bz2    //解压源码
#cd u-boot-2010.03/board/samsung/    //进入目录
#mkdir mini2440                      //创建 mini2440 文件夹
```

2) 因 2440 和 2410 的资源差不多，所以就以 2410 项目的代码作为模板，以后再修改

```
#cp -rf smdk2410/* mini2440/    //将 2410 下所有的代码复制到 2440 下
#cd mini2440                    //进入 mini2440 目录
#mv smdk2410.c mini2440.c      //将 mini2440 下的 smdk2410.c 改名为 mini2440.c
#cd ../../..                    //回到 u-boot 根目录
#cp include/configs/smdk2410.h include/configs/mini2440.h //建立 2440 头文件
#gedit board/samsung/mini2440/Makefile //修改 mini2440 下 Makefile 的编译项，如下：
COBJS = mini2440.o flash.o //因在 mini2440 下我们将 smdk2410.c 改名为 mini2440.c
```

3) 修改 u-boot 根目录下的 Makefile 文件。查找到 smdk2410_config 的地方，在下面按照 smdk2410_config 的格式建立 mini2440_config 的编译选项，另外还要指定交叉编译器

```
#gedit Makefile
```

```
CROSS_COMPILE ?= arm-linux-    //指定交叉编译器为 arm-linux-gcc

smdk2410_config : unconfig    //2410 编译选项格式
```

```

@$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 samsung s3c24x0
mini2440_config      :      unconfig      //2440 编译选项格式
@$(MKCONFIG) $(@:_config=) arm arm920t mini2440 samsung s3c24x0

```

*说明: arm: CPU 的架构(ARCH)

arm920t: CPU 的类型

mini2440 : 对应 board 目录下建立新的开发板项目的目录

samsung: 新开发板项目目录的上级目录, 如直接在 board 下建立新的开发板项目的目录, 则这里就为 NULL

s3c24x0: CPU 型号

*注意: 编译选项格式的第二行要用 Tab 键开始, 否则编译会出错

4) 测试编译新建的 mini2440 开发板项目

```

#make distclean      //清除 u-boot 里之前的痕迹
#make mini2440_config //如果出现 Configuring for mini2440 board...则表示设置正确
#make all//编译后在根目录下会出现 u-boot.bin 文件, 则 u-boot 移植的第一步就算完成

```

到此为止, u-boot 对自己的 mini2440 开发板还没有任何用处, 以上的移植只是搭建了一个 mini2440 开发板 u-boot 的框架, 要使其功能实现, 还要根据 mini2440 开发板的具体资源情况来对 u-boot 源码进行修改。

2、根据 u-boot 启动流程图的步骤来分析或者修改添加 u-boot 源码, 使之适合 mini2440 开发板 (注: 修改或添加的地方都用红色表示)。

1) mini2440 开发板 u-boot 的 stage1 入口点分析。

一般在嵌入式系统软件开发中, 在所有源码文件编译完成之后, 链接器要读取一个链接分配文件, 在该文件中定义了程序的入口点, 代码段、数据段等分配情况等。那么我们的 mini2440 开发板 u-boot 的这个链接文件就是 cpu/arm920t/u-boot.lds, 打开该文件部分代码如下:

```
#gedit cpu/arm920t/u-boot.lds
```

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)      //定义生成文件的目标平台是 arm
ENTRY(_start)         //定义程序的入口点是_start
SECTIONS
{
    //其他一些代码段、数据段等分配
    . = 0x00000000;
    . = ALIGN(4);
    .text :
    {
        cpu/arm920t/start.o    (.text)
        *(.text)
    }
    .....
}
```

知道了程序的入口点是_start，那么我们就打开 mini2440 开发板 u-boot 第一个要运行的程序 cpu/arm920t/start.S（即 u-boot 的 stage1 部分），查找到_start 的位置如下：

```
#gedit cpu/arm920t/start.S
```

```
.globl _start
_start: b    start_code    //将程序的执行跳转到 start_code 处
```

从这个汇编代码可以看到程序又跳转到 start_code 处开始执行，那么再查找到 start_code 处的代码如下：

```
/*
the actual start code
```

```

*/
start_code:

/*
    * set the cpu to SVC32 mode
*/

    mrs    r0,cpsr
    bic    r0,r0,#0x1f
    orr    r0,r0,#0xd3
    msr    cpsr,r0

    bl     coloured_LED_init //此处两行是对 AT91RM9200DK 开发板上的 LED 进行初
始化的
    bl     red_LED_on

```

由此可以看到，start_code 处才是 u-boot 启动代码的真正开始处。以上就是 u-boot 的 stage1 入口的过程。

2) mini2440 开发板 u-boot 的 stage1 阶段的硬件设备初始化。

a. 由于在 u-boot 启动代码处有两行是 AT91RM9200DK 的 LED 初始代码，但我们 mini2440 上的 LED 资源与该开发板的不一致，所以我们要删除或屏蔽该处代码，再加上 mini2440 的 LED 驱动代码（注：添加 mini2440 LED 功能只是用于表示 u-boot 运行的状态，给调试带来方便，可将该段代码放到任何你想调试的地方），修改如下：

```

#gedit include/configs/mini2440.h

#define CONFIG_ARM920T    1    /* This is an ARM920T Core*/

#define CONFIG_S3C24X0    1    /* in a SAMSUNG S3C24x0-type SoC*/

//注释掉有关 S3C2410 的定义
//#define    CONFIG_S3C2410    1    /* Specifically in a SAMSUNG S3C2410 SoC */
#define CONFIG_SMDK2410    1    /* on a SAMSUNG SMDK2410 Board */

//添加下面两个定义

```

```

#define CONFIG_S3C2440 1    /* Specifically in a SAMSUNG S3C2440 SoC */
#define CONFIG_mini2440_LED 1 //启动时点亮 LED

#gedit cpu/arm920t/start.S

/*bl coloured_LED_init    //这两行是 AT91RM9200DK 开发板的 LED 初始化，注释掉
   bl red_LED_on*/

//当 u-boot 进入第一阶段时 LED1 灯点亮

#if defined(CONFIG_S3C2440) //区别与其他开发板

//根据 mini2440 原理图可知 LED 分别由 S3C2440 的 PB5、6、7、8 口来控制，以下是
PB 端口寄存器基地址(查 2440 的 DataSheet 得知)

#define GPBCON  0x56000010
#define GPBDAT  0x56000014
#define GPBUP   0x56000018

    ldr r0, =GPBUP

    ldr r1, =0x7FF    //即：二进制 1111111111，关闭 PB 口上拉
    str r1, [r0]

    ldr r0, =GPBCON

    ldr r1, =0x15400 //即：二进制 00010101010000000000，配置 PB5、6、7、8 为输出
    str r1, [r0]

    ldr r0, =GPBDAT

    ldr r1, =0x1C0    //即：二进制 111000000，PB5 设为低电平，6、7、8 为高电平
    str r1, [r0]

#endif

#gedit board/samsung/mini2440/mini2440.c

int board_init (void)
{

```



```

S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
S3C24X0_GPIO * const gpio = S3C24X0_GetBase_GPIO();.....
/* set up the I/O ports */
    gpio->GPACON = 0x007FFFFFFF;
#ifdef CONFIG_mini2440_LED
    gpio->GPBCON = (1<<(5*2)) |(1<<(6*2)) | (1<<(7*2)) | (1<<(8*2));

//即：二进制 00010101010000000000，配置 PB5、6、7、8 为输出口
#else
    gpio->GPBCON = 0x00044556;
#endif

    gpio->GPBUP = 0x000007FF;
    gpio->GPCCON = 0xAAAAAAAA;
    gpio->GPCUP = 0x0000FFFF;
    gpio->GPDCON = 0xAAAAAAAA;
    gpio->GPDUP = 0x0000FFFF;
    gpio->GPECON = 0xAAAAAAAA;
    gpio->GPEUP = 0x0000FFFF;
    gpio->GPFCON = 0x000055AA;
    gpio->GPFUP = 0x000000FF;
    gpio->GPGCON = 0xFF95FFBA;
    gpio->GPGUP = 0x0000FFFF;
    gpio->GPHCON = 0x002AFAAA;
    gpio->GPHUP = 0x000007FF;

/* arch number of SMDK2410-Board */
    gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;

/* adress of boot parameters */
    gd->bd->bi_boot_params = 0x30000100;

    icache_enable();
    dcache_enable();

```

```

#if defined(CONFIG_mini2440_LED)

    gpio->GPBDAT &=~(1<<6); //当 u-boot 进入第二阶段时 LED2 灯点亮
#endif

return 0;

}

```

b. 修改中断禁止

```

#gedit cpu/arm920t/start.S

/*
 * mask all IRQs by setting all bits in the INTMR – default
 */

mov    r1, #0xffffffff
ldr    r0, =INTMSK
str    r1, [r0]
# if defined(CONFIG_S3C2410)

    ldr    r1, =0x7ff
    ldr    r0, =INTSUBMSK
    str    r1, [r0]

# endif

#if defined(CONFIG_S3C2440)    //添加 s3c2440 的中断禁止部分

    ldr r1,=0x7fff    //INTSUBMSK 寄存器有 15 位可用,置 1 屏蔽中断
    ldr r0,=INTSUBMSK

    str r1,[r0]

#endif

#gedit include/asm-arm/arch-s3c24x0/s3c24x0_cpu.h

//添加对 S3C2440 的定义, 在调用 cpu/arm920t/s3c24x0/interrupts.c 用得到

#elif defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)

```

c.修改时钟设置（2440 的主频为 405MHz）

```
#gedit cpu/arm920t/start.S
```

```
//添加对 S3C2440 的定义
```

```
#ifdef CONFIG_S3C24X0
```

```
/* turn off the watchdog */
```

```
# if defined(CONFIG_S3C2400)
```

```
# define pWTCON    0x15300000
```

```
# define INTMSK    0x14400008    /* Interrupt-Controller base addresses */
```

```
# define CLKDIVN    0x14800014    /* clock divisor register */
```

```
#else
```

```
# define pWTCON    0x53000000
```

```
# define INTMSK    0x4A000008    /* Interrupt-Controller base addresses */
```

```
# define INTSUBMSK    0x4A00001C
```

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

```
# endif
```

```
//设置时钟寄存器
```

```
#define    CLK_CTL_BASE    0x4C000000
```

```
#define    MDIV_405    0x7f<<12
```

```
#define    PSDIV_405    0x21
```

```
#define    MDIV_200    0xa1<<12
```

```
#define    PSDIV_200    0x31
```

```
.....
```

```
#if defined(CONFIG_S3C2440)
```

```
    ldr r0,=CLKDIVN    //设置分频系数 FCLK:HCLK:PCLK = 1:4:8
```

```
    mov r1,#5
```

```
    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


mov r1,#CLK_CTL_BASE    //设置 FCLK=405MHz
mov r2,#MDIV_405
add r2,r2,#PSDIV_405
str r2,[r1,#0x04]
#else
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */

ldr r0,=CLKDIVN
mov r1,#3
str r1,[r0]


mrc p15,0,r1,c1,c0,0
orr r1,r1,#0xc0000000
mcr p15,0,r1,c1,c0,0


mov r1,#CLK_CTL_BASE
mov r2,#MDIV_200
add r2,r2,#PSDIV_200
str r2,[r2,#0x04]
#endif
#endif    /* CONFIG_S3C2400 || CONFIG_S3C2410 */

#gedit board/samsung/mini2440/mini2440.c //设置主频和 USB 时钟频率参数与 start.S 中
的一致

#define FCLK_SPEED 2    //设置默认等于 2，即下面红色代码部分有效
#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
#elif FCLK_SPEED==2 /* Fout = 405MHz */
#define M_MDIV 0x7F //这三个值根据 S3C2440 芯片手册“PLL VALUE
SELECTION TABLE”部分进行设置
#define M_PDIV 0x2
#define M_SDIV 0x1
#endif

#define USB_CLOCK 2 //设置默认等于 2，即下面红色代码部分有效

#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
#elif USB_CLOCK==2 /* Fout = 48MHz */
#define U_M_MDIV 0x38 //这三个值根据 S3C2440 芯片手册“PLL VALUE
SELECTION TABLE”部分进行设置
#define U_M_PDIV 0x2
#define U_M_SDIV 0x2
#endif

#gedit include/asm-arm/arch-s3c24x0/s3c24x0.h

分别在第 81 91 95 106 144 400 行将

#ifdef CONFIG_S3C2410 改为

```

```

#if defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)

.....

struct s3c24x0_clock_power {
    u32 LOCKTIME;
    u32 MPLLCON;
    u32 UPLLCON;
    u32 CLKCON;
    u32 CLKSLOW;
    u32 CLKDIVN;

#ifdef CONFIG_S3C2440    //定义 s3c2440 的 CAMDIVN 寄存器
    u32 CAMDIVN;
#endif
};

#gedit cpu/arm920t/s3c24x0/speed.c

static ulong get_PLLCLK(int pllreg)
{
    struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();
    ulong r, m, p, s;
    if (pllreg == MPLL)
        r = readl(&clk_power->MPLLCON);
    else if (pllreg == UPLL)
        r = readl(&clk_power->UPLLCON);
    else
        hang();
    m = ((r & 0xFF000) >> 12) + 8;
    p = ((r & 0x003F0) >> 4) + 2;
    s = r & 0x3;

#ifdef CONFIG_S3C2440
    if (pllreg == MPLL)

```

```

        return ((CONFIG_SYS_CLK_FREQ * m * 2) / (p << s));

        else if (pllreg == UPLL)
#endif

        return (CONFIG_SYS_CLK_FREQ * m) / (p << s);
}

/* return FCLK frequency */
ulong get_FCLK(void)
{
    return get_PLLCLK(MPLL);
}

/* return HCLK frequency */
ulong get_HCLK(void)
{
    struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();

    //return (readl(&clk_power->CLKDIVN) & 2) ? get_FCLK() / 2 : get_FCLK();
#ifdef CONFIG_S3C2440
    if (clk_power->CLKDIVN & 0x6)
    {
        if ((clk_power->CLKDIVN & 0x6)==2) return(get_FCLK()/2);
        if ((clk_power->CLKDIVN & 0x6)==6) return((clk_power->CAMDIVN &
0x100) ? get_FCLK()/6 : get_FCLK()/3);
        if ((clk_power->CLKDIVN & 0x6)==4) return((clk_power->CAMDIVN &
0x200) ? get_FCLK()/8 : get_FCLK()/4);
        return(get_FCLK());
    }
    else
        return(get_FCLK());
#endif
}

```

d.修改 SDRAM 的设置

```
#gedit board/samsung/mini2440/lowlevel_init.S

/* 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 */
#if defined(CONFIG_S3C2440)
#define Trp        0x2
#define REFCNT     1012     /*period=64ms/8192=7.8125us*/
#else
#define REFCNT      1113     /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */
#define Trp 0x0
#endif
```

e.修改串口的设置

```
#gedit drivers/serial/serial_s3c24x0.c

static int serial_init_dev(const int dev_index)
{
    struct s3c24x0_uart *uart = s3c24x0_get_base_uart(dev_index);

#ifdef CONFIG_HWFLOW
    hwflow = 0; /* turned off by default */
#endif

    /* FIFO enable, Tx/Rx FIFO clear */
    //writel(0x07, &uart->UFCON);
    writel(0x00, &uart->UFCON);
    writel(0x0, &uart->UMCON);
```



```

/* Normal,No parity,1 stop,8 bit */
writel(0x3, &uart->ULCON);

/*
 * tx=level,rx=edge,disable timeout int.,enable rx error int.,
 * normal,interrupt or polling
 */
writel(0x245, &uart->UCON);

```

3、测试

现在编译 u-boot，在根目录下会生成一个 u-boot.bin 文件。然后我们利用 mini2440 原有的 supervivi 把 u-boot.bin 下载到 RAM 中运行测试(注意：我们使用 supervivi 进行下载时已经对 CPU、RAM 进行了初始化，所以我们在 u-boot 中要屏蔽掉 cpu/arm920t/start.S 中对 CPU、RAM 的初始化)，如下：

```

/*#ifndef CONFIG_SKIP_LOWLEVEL_INIT //在 start.S 文件中屏蔽 u-boot 对 CPU、RAM
的初始化
    bl cpu_init_crit
#endif*/

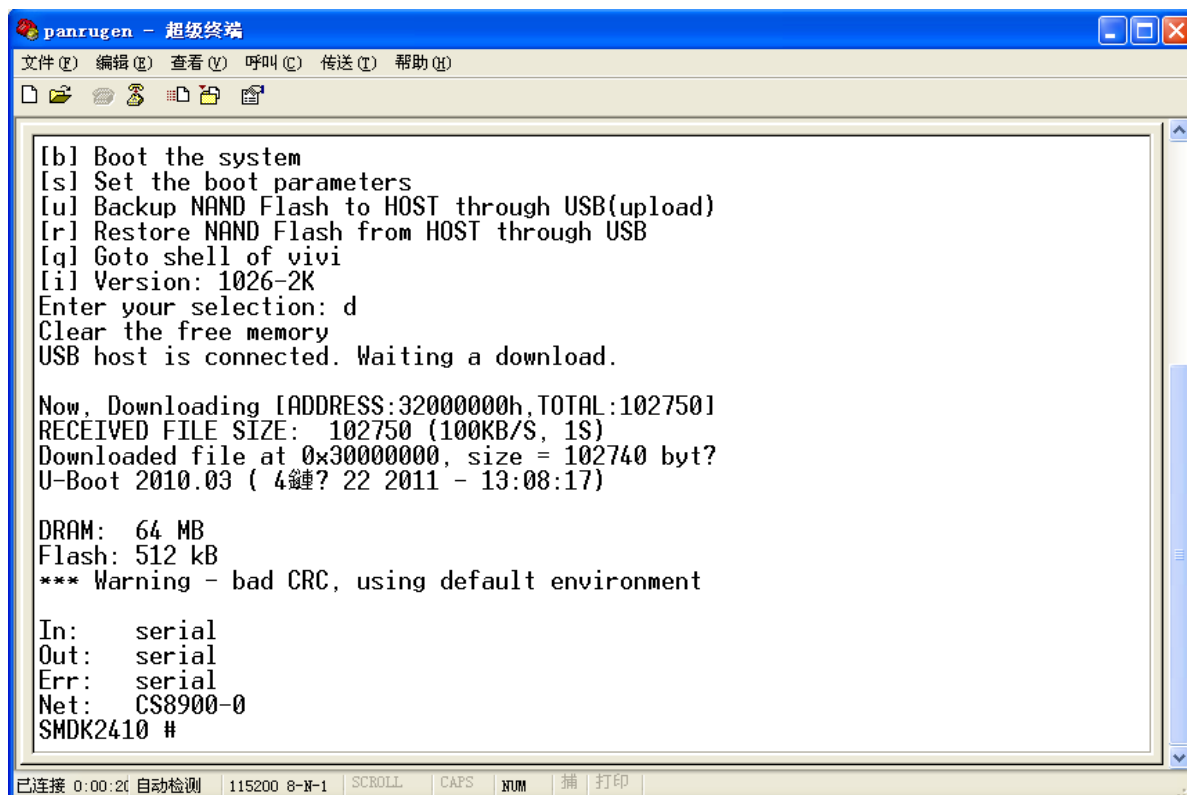
#make distclean

#make mini2440_config

#make all

```

下载运行后可以看到开发板上的 LED 灯第一了亮了，其他三个熄灭，终端有输出信息并且出现类似 Shell 的命令，这说明这一部分移植完成。终端运行结果如下：



```
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[il] Version: 1026-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:32000000h,TOTAL:102750]
RECEIVED FILE SIZE: 102750 (100KB/S, 1S)
Downloaded file at 0x30000000, size = 102740 byt?
U-Boot 2010.03 ( 4鏈? 22 2011 - 13:08:17)

DRAM: 64 MB
Flash: 512 kB
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Net: CS8900-0
SMDK2410 #
```

已连接 0:00:20 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

(二) 支持 NorFlash

通常，在嵌入式 bootloader 中，有两种方式来引导启动内核：从 Nor Flash 启动和从 Nand Flash 启动。u-boot 中默认是从 Nor Flash 启动，再从上一节这个运行结果图中看，还发现几个问题：第一，我开发板的 Nor Flash 是 2M 的，而这里显示的是 512kB；第二，出现 Warning - bad CRC, using default environment 的警告信息。不是 u-boot 默认是从 Nor Flash 启动的吗？为什么会有这些错误信息呢？这是因为我们还没有添加对我们的 Nor Flash 的支持，u-boot 默认的是其他型号的 Nor Flash，而我们的 Nor Flash 的

型号是 SST39VF1601。另外怎样将命令行提示符前面的 SMDK2410 变成我自己定义的呢？

下面我们一一来解决这些问题：

1、让 u-boot 完全对我们 Nor Flash 的支持。首先我们修改头文件代码如下：

```
#gedit include/configs/mini2440.h //修改命令行前的名字和 Nor Flash 参数部分的定义

#define CONFIG_SYS_PROMPT "[PAN_MINI2440]# " //将命令行前的名字改成 [MINI2440]

/*-----
* FLASH and environment organization
*/
/*
#if 0 //注释掉下面两个类型的 Nor Flash 设置，因为不是我们所使用的型号
#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_SYS_MAX_FLASH_BANKS 1 /* max number of memory banks */

#ifdef CONFIG_AMD_LV800
#define PHYS_FLASH_SIZE 0x00100000 /* 1MB */
#define CONFIG_SYS_MAX_FLASH_SECT (19) /* max number of sectors on one chip */
/*
#define CONFIG_ENV_ADDR (CONFIG_SYS_FLASH_BASE + 0x0F0000) /* addr of environment */
#endif
#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
#define CONFIG_SST_39VF1601 1 //添加 mini2440 开发板 Nor Flash 设置
#define PHYS_FLASH_SIZE 0x200000//我们开发板的 Nor Flash 是 2M
#define CONFIG_SYS_MAX_FLASH_SECT (512) //根据 SST39VF1601 的芯片手册描述，对其进行操作有两种方式：块方式和扇区方式。现采用扇区方式(sector)，1 sector = 2Kword = 4Kbyte(bit：通常指一个二进制位；byte：通常包含 8bit；word：与总线、cpu
```

命令字位数等有关，如数据总线为 32 位，则 1word 为 4byte。SST39VF1601 的芯片有 16 位的数据线，则 1word 为 2byte。存储容量通常用 byte 表示，因为与系统硬件无关。)，所以 2M 的 Nor Flash 共有 512 个 sector

```
#define CONFIG_ENV_ADDR (CONFIG_SYS_FLASH_BASE + 0x040000) //暂设置环境变量的首地址为 0x040000(即：256Kb)
```

2、然后添加对我们 mini2440 开发板上 2M 的 Nor Flash(型号为 SST39VF1601)的支持。

在 u-boot 中对 Nor Flash 的操作分别有初始化、擦除和写入，所以我们主要修改与硬件密切相关的三个函数 flash_init、flash_erase、write_hword，修改代码如下：

```
#gedit board/samsung/mini2440/flash.c
```

//修改定义部分如下：

```
//#define MAIN_SECT_SIZE 0x10000
```

```
#define MAIN_SECT_SIZE 0x1000 //定义为 4k，刚好是一个扇区的大小
```

```
#ifdef CONFIG_SST_39VF1601 //根据数据手册添加 SST39VF1601NorFlash 芯片
```

```
#define MEM_FLASH_ADDR1 (*(volatile u16 *) (CONFIG_SYS_FLASH_BASE  
+ (0x000005555<<1)))
```

```
#define MEM_FLASH_ADDR2 (*(volatile u16 *) (CONFIG_SYS_FLASH_BASE + (0x000002AAA  
<<1)))
```

```
#else
```

```
#define MEM_FLASH_ADDR1 (*(volatile u16 *) (CONFIG_SYS_FLASH_BASE + (0x00000555  
<< 1)))
```

```
#define MEM_FLASH_ADDR2 (*(volatile u16 *) (CONFIG_SYS_FLASH_BASE + (0x000002AA  
<< 1)))
```

```
#endif
```

//修改 flash_init 函数如下：

```
#elif defined(CONFIG_AMD_LV800)
```

```
(AMD_MANUFACT & FLASH_VENDMASK) |
```

```
(AMD_ID_LV800B & FLASH_TPEMASK);
```

```

#elif defined(CONFIG_SST_39VF1601) // 在 CONFIG_AMD_LV800 后面添加
CONFIG_SST_39VF1601

    (SST_MANUFACT & FLASH_VENDMASK) |

    (SST_ID_xF1601 & FLASH_TYPEMASK);

.....

for (j = 0; j < flash_info[i].sector_count; j++) {
#ifdef CONFIG_SST_39VF1601 //ifdef 的作用是如果 CONFIG_SST_39VF1601 定义了
执行#else 部分代码，如果没定义执行下面的代码，这相当于注释掉下面的代码
    if (j <= 3) {
        /* 1st one is 16 KB */
        if (j == 0) {
            flash_info[i].start[j] =flashbase + 0;
        }
        /* 2nd and 3rd are both 8 KB */
        if ((j == 1) || (j == 2)) {
            flash_info[i].start[j]=flashbase +0x4000+(j-1) *0x2000;
        }
        /* 4th 32 KB */
        if (j == 3) {
            flash_info[i].start[j] =flashbase + 0x8000;
        }
    } else {
        flash_info[i].start[j] =flashbase + (j - 3) * MAIN_SECT_SIZE;
    }
#else
    flash_info[i].start[j] = flashbase + (j) * MAIN_SECT_SIZE;
#endif
}

```

//修改 flash_print_info 函数如下:

```
switch (info->flash_id & FLASH_VENDMASK) {
    case (AMD_MANUFACT & FLASH_VENDMASK):
        printf ("AMD: ");
        break;
    case (SST_MANUFACT & FLASH_VENDMASK): //添加 SST39VF1601 支持
        printf("SST: ");
        break;
    default:
        printf ("Unknown Vendor ");
        break;
}

switch (info->flash_id & FLASH_TYPEMASK) {
    case (AMD_ID_LV400B & FLASH_TYPEMASK):
        printf ("1x Amd29LV400BB (4Mbit)\n");
        break;
    case (AMD_ID_LV800B & FLASH_TYPEMASK):
        printf ("1x Amd29LV800BB (8Mbit)\n");
        break;
    case (SST_ID_xF1601 & FLASH_TYPEMASK): //添加 SST39VF1601 支持
        printf("1x SST39VF1601 (2MB)\n");
        break;
    default:
        printf ("Unknown Chip Type\n");
        goto Done;
        break;
}
```

//修改 flash_erase 函数如下:

```
ushort result;
```

```
int iflag, cflag, prot, sect;
```

```
int rc = ERR_OK;
```

```
//int chip; //注释掉 chip
```

```
.....
```

```
#ifdef CONFIG_SST_39VF1601
```

```
if((info->flash_id & FLASH_VENDMASK) != (SST_MANUFACT & FLASH_VENDMASK))
```

```
{
```

```
    return ERR_UNKNOWN_FLASH_VENDOR;
```

```
}
```

```
#else
```

```
    if ((info->flash_id & FLASH_VENDMASK) != (AMD_MANUFACT & FLASH_VENDMASK))
```

```
    {
```

```
        return ERR_UNKNOWN_FLASH_VENDOR;
```

```
    }
```

```
#endif
```

```
.....
```

```
#if 0 //注释掉 wait until flash is ready
```

```
/* wait until flash is ready */
```

```
chip = 0;
```

```
do {
```

```
    result = *addr;
```

```
    /* check timeout */
```

```
    if (get_timer_masked () > CONFIG_SYS_FLASH_ERASE_TOUT) {
```

```
        MEM_FLASH_ADDR1 = CMD_READ_ARRAY;
```

```
        chip = TMO;
```

```
        break;
```

```

    }

    if (!chip && (result & 0xFFFF) & BIT_ERASE_DONE)
        chip = READY;

    if (!chip && (result & 0xFFFF) & BIT_PROGRAM_ERROR)
        chip = ERR;
} while (!chip);
MEM_FLASH_ADDR1 = CMD_READ_ARRAY;
if (chip == ERR) {
    rc = ERR_PROG_ERROR;
    goto outahere;
}
if (chip == TMO) {
    rc = ERR_TIMEOUT;
    goto outahere;
}
#endif

// 添加下面的代码

/* wait until flash is ready */
while(1){
    unsigned short i;
    i = *((volatile unsigned short *)addr) & 0x40;
    if(i != *((volatile unsigned short *)addr) & 0x40))
        continue;
    if(((volatile unsigned short *)addr) & 0x80)
        break;
}

//修改 write_hword 函数如下:
vu_short *addr = (vu_short *) dest;

```



```
ushort result;

int rc = ERR_OK;

int cflag, iflag;

//int chip;    //注释掉 chip
.....

MEM_FLASH_ADDR1 = CMD_UNLOCK1;
MEM_FLASH_ADDR2 = CMD_UNLOCK2;
//MEM_FLASH_ADDR1 = CMD_UNLOCK_BYPASS;
/*addr = CMD_PROGRAM;
MEM_FLASH_ADDR1 = CMD_PROGRAM;
*addr = data;
/* arm simple, non interrupt dependent timer */
reset_timer_masked ();

#if 0    //注释掉 wait until flash is ready
    /* wait until flash is ready */
    chip = 0;
    do {
        result = *addr;
        /* check timeout */
        if (get_timer_masked () > CONFIG_SYS_FLASH_ERASE_TOUT) {
            chip = ERR | TMO;
            break;
        }
        if (!chip && ((result & 0x80) == (data & 0x80)))
            chip = READY;
        if (!chip && ((result & 0xFFFF) & BIT_PROGRAM_ERROR)) {
            result = *addr;
            if ((result & 0x80) == (data & 0x80))
                chip = READY;
```

```
else
    chip = ERR;
}
} while (!chip);
*addr = CMD_READ_ARRAY;
if (chip == ERR || *addr != data)
    rc = ERR_PROG_ERROR;
#endif
// 添加下面的代码
/* wait until flash is ready */
while(1){
    unsigned short i = *(volatile unsigned short *)addr & 0x40;
    if(i != *(volatile unsigned short *)addr & 0x40)    //D6 == D6
        continue;
    if(*(volatile unsigned short *)addr & 0x80) == (data & 0x80)){
        rc = ERR_OK;
        break;    //D7 == D7
    }
}
```

3、测试

修改完后重新编译 u-boot，下载到 RAM 中运行结果如下图：

```
U-Boot 2010.03 (Apr 17 2010 - 17:20:34)

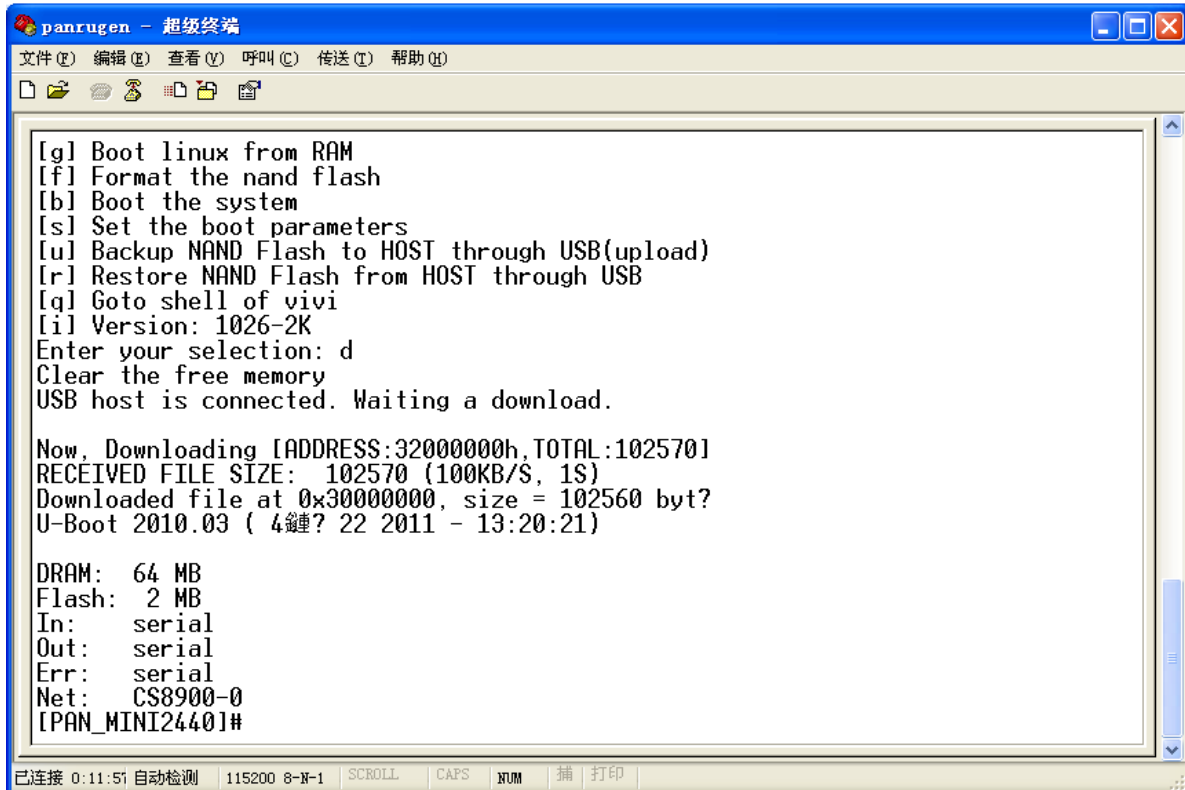
DRAM: 64 MB
Flash: 2 MB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   CS8900-A
```

从运行结果图看，Nor Flash 的大小可以正确检测到了，命令行前面的名字也由原来的 SMDK2410 改成我自己定义的[MINI2440]了，但是还会出现 bad CRC 的警告信息，其实这并不是什么问题，只是还没有将环境变量设置到 Nor Flash 中，我们执行一下 u-boot 的：saveenv 命令就可以了。如下图：

```
Saving Environment to Flash...
Un-Protected 16 sectors
Erasing Flash...Erasing sector 64 ... ok.
Erasing sector 65 ... ok.
Erasing sector 66 ... ok.
Erasing sector 67 ... ok.
Erasing sector 68 ... ok.
Erasing sector 69 ... ok.
Erasing sector 70 ... ok.
Erasing sector 71 ... ok.
Erasing sector 72 ... ok.
Erasing sector 73 ... ok.
Erasing sector 74 ... ok.
Erasing sector 75 ... ok.
Erasing sector 76 ... ok.
Erasing sector 77 ... ok.
Erasing sector 78 ... ok.
Erasing sector 79 ... ok.
Erased 16 sectors
Writing to Flash... done
Protected 16 sectors
```

再重新下载 u-boot.bin 文件到 RAM 中运行，可以观察到不会出现警告信息了，这时候 u-boot 已经对我们开发板上的 Nor Flash 完全支持了。如下：



```
panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1026-2K
Enter your selection: d
Clear the free memory
USB host is connected. Waiting a download.

Now, Downloading [ADDRESS:32000000h,TOTAL:102570]
RECEIVED FILE SIZE: 102570 (100KB/S, 1S)
Downloaded file at 0x30000000, size = 102560 byt?
U-Boot 2010.03 ( 4鏈? 22 2011 - 13:20:21)

DRAM: 64 MB
Flash: 2 MB
In: serial
Out: serial
Err: serial
Net: CS8900-0
[PAN_MINI2440]#
```

已连接 0:11:51 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

（三）支持 NandFlash

1、在 u-boot 中添加对我们开发板上 Nand Flash 的支持

目前 u-boot 中还没有对 2440 上 Nand Flash 的支持，也就是说要想 u-boot 从 Nand Flash 上启动得自己去实现了。

1)首先，在 include/configs/mini2440.h 头文件中定义 NandFlash 要用到的宏，如下：

```
#gedit include/configs/mini2440.h //在文件末尾加入以下 Nand Flash 相关定义

#define CONFIG_ARM920T 1 /* This is an ARM920T Core*/

#define CONFIG_S3C24X0 1 /* in a SAMSUNG S3C24x0-type SoC*/
//#define CONFIG_S3C2410 1 /* Specifically in a SAMSUNG S3C2410 SoC */
#define CONFIG_SMDK2410 1 /* on a SAMSUNG SMDK2410 Board */
#define CONFIG_S3C2440 1 /* Specifically in a SAMSUNG S3C2440 SoC */
#define CONFIG_mini2440_LED 1

#define CONFIG_S3C2440_NAND_BOOT 1 //添加对 NandFlash 的定义
```

2)其次，修改 cpu/arm920t/start.S 这个文件，使 u-boot 从 Nand Flash 启动，在上一节中提过，u-boot 默认是从 Nor Flash 启动的。修改部分如下：

```
#gedit cpu/arm920t/start.S

//注意：在上一篇 Nor Flash 启动中，我们为了把 u-boot 用 supervivi 下载到内存中运行而屏蔽掉这段有关 CPU 初始化的代码。而现在我们要把 u-boot 下载到 Nand Flash 中，从 Nand Flash 启动，所以现在要恢复这段代码。

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif

.....

#If 0 //注释掉 NorFlash 启动
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
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

    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 {r3-r10}    /* copy from source address [r0] */

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

    cmp r0, r2    /* until source end address [r2] */

    ble copy_loop

#endif    /* CONFIG_SKIP_RELOCATE_UBOOT */

#endif

//下面添加对 Nand Flash 的支持，下面的这段代码主要实现的功能就把，4kb 以后的程序可以复制到 RAM 中，调用了用 C 语言写的函数 nand_read_ll(), 128MB 的 NandFlash 的读命令和 64MB 小页的读操作有所不同，必须还要发送一个 0x30 的指令

#define NAND_CTL_BASE 0x4e000000

#ifdef CONFIG_S3C2440_NAND_BOOT

    @reset NAND    //复位 Nand Flash

    #define oNFCNF 0x00

    #define oNFCNT 0x04

    #define oNFSTAT 0x08

    #define oNFCMD 0x20

    mov r1, #NAND_CTL_BASE

    ldr r2, =( (7<<12)|(7<<8)|(7<<4)|(0<<0))

    str r2, [r1, #oNFCNF]    //设置配置寄存器的初始值

    ldr r2, [r1, #oNFCNF]

    ldr r2, =((1<<4)|(0<<1)|(1<<0))

```

```

str r2,[r1,#oNFCNT]      //设置控制寄存器
ldr r2,[r1,#oNFCNT]

ldr r2,=(0x6)            //RnB Clear
str r2,[r1,#oNFSTAT]
ldr r2,[r1,#oNFSTAT]

mov r2,#0xff             //复位 command
strb r2,[r1,#oNFCMD]
mov r3,#0                //等待
nand1:
    add r3,r3,#0x1
    cmp r3,#0xa
    blt nand1
nand2:
    ldr r2,[r1,#oNFSTAT]    //等待就绪
    tst r2,#0x4
    beq nand2

    ldr r2,[r1,#oNFCNT]
    orr r2,r2,#0x2          //取消片选
    str r2,[r1,#oNFCNT]

    @get read to call C functions (for nand_read())
    ldr sp,DW_STACK_START    //为 C 代码准备堆栈, DW_STACK_START 定义在下面
    mov fp,#0
    @copy U-Boot to Ram
    ldr r0,=TEXT_BASE        //传递给 C 代码的第一个参数: u-boot 在 RAM 中的起始地址
    mov r1,#0x0              //传递给 C 代码的第二个参数: Nand Flash 的起始地址
    mov r2,#0x60000          //传递给 C 代码的第三个参数: u-boot 的长度大小

```

```

    bl nand_read_ll    //此处调用 C 代码中读 Nand 的函数，现在还没有要自己编写实现
    tst r0,#0x0
    beq ok_nand_read
bad_nand_read:
    loop2:  b loop2    //infinite loop
ok_nand_read:
    @verify    //检查搬移后的数据，如果前 4k 完全相同，表示搬移成功
    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          //添加结束
.....
//_start_armboot:    .word start_armboot 之后添加以下代码
_start_armboot:    .word start_armboot
#define STACK_BASE 0x33f00000
#define STACK_SIZE 0x10000
.align 2
DW_STACK_START:.word STACK_BASE+STACK_SIZE - 4

```

3) 再次，在 board/samsung/mini2440/目录下新建一个 nand_read.c 文件，在该文件中

来实现上面汇编中要调用的 `nand_read_ll` 函数，代码如下：

```
#gedit board/samsung/mini2440/nand_read.c //新建一个 nand_read.c 文件，记得保存
#include <config.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
#if defined(CONFIG_S3C2410)
#define NFCONF __REGi(NF_BASE + 0x0)
#define NFCMD __REGb(NF_BASE + 0x4)
#define NFADDR __REGb(NF_BASE + 0x8)
#define NFDATA __REGb(NF_BASE + 0xc)
#define NFSTAT __REGb(NF_BASE + 0x10)
#define NFSTAT_BUSY 1
#define nand_select() (NFCONF &= ~0x800)
#define nand_deselect() (NFCONF |= 0x800)
#define nand_clear_RnB() do { } while (0)
#elif defined(CONFIG_S3C2440)
#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 << 2)
#define nand_select() (NFCONT &= ~(1 << 1))
#define nand_deselect() (NFCONT |= (1 << 1))
#define nand_clear_RnB() (NFSTAT |= NFSTAT_BUSY)
#endif

static inline void nand_wait(void)
```

```
{  
    int i;  
    while (!(NFSTAT & NFSTAT_BUSY))  
        for (i=0; i<10; i++);  
}  
  
#if defined(CONFIG_S3C2410)  
#define NAND_PAGE_SIZE 512  
#define BAD_BLOCK_OFFSET 517  
#define NAND_BLOCK_MASK (NAND_PAGE_SIZE - 1)  
#define NAND_BLOCK_SIZE 0x4000  
#else  
#define NAND_5_ADDR_CYCLE  
#define NAND_PAGE_SIZE 2048  
#define BAD_BLOCK_OFFSET NAND_PAGE_SIZE  
#define NAND_BLOCK_MASK (NAND_PAGE_SIZE - 1)  
#define NAND_BLOCK_SIZE (NAND_PAGE_SIZE * 64)  
#endif  
  
#if defined(CONFIG_S3C2410) && (NAND_PAGE_SIZE != 512)  
#error "S3C2410 does not support nand page size != 512"  
#endif  
  
static int is_bad_block(unsigned long i)  
{  
    unsigned char data;  
    unsigned long page_num;  
    nand_clear_RnB();  
#if (NAND_PAGE_SIZE == 512)  
    NFCMD = NAND_CMD_READOOB;
```

```
NFADDR = BAD_BLOCK_OFFSET & 0xf;

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

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

NFADDR = (i >> 25) & 0xff;
#elif (NAND_PAGE_SIZE == 2048)
    page_num = i >> 11;

    NFCMD = NAND_CMD_READ0;

    NFADDR = BAD_BLOCK_OFFSET & 0xff;

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

    NFADDR = page_num & 0xff;

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

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

    NFCMD = NAND_CMD_READSTART;
#endif

    nand_wait();

    data = (NFDATA & 0xff);

    if (data != 0xff)

        return 1;

    return 0;
}

static int nand_read_page_ll(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)
    NFADDR = addr & 0xff;

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

```
NFADDR = (addr >> 17) & 0xff;
NFADDR = (addr >> 25) & 0xff;
#elif (NAND_PAGE_SIZE == 2048)
    page_num = addr >> 11;
    NFADDR = 0;
    NFADDR = 0;
    NFADDR = page_num & 0xff;
    NFADDR = (page_num >> 8) & 0xff;
    NFADDR = (page_num >> 16) & 0xff;
    NFCMD = NAND_CMD_READSTART;
#else
#error "unsupported nand page size"
#endif

    nand_wait();
    for (i = 0; i < NAND_PAGE_SIZE; i++)
    {
        *buf = (NFDATA & 0xff);
        buf++;
    }
    return NAND_PAGE_SIZE;
}

int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
    int i, j;
    if ((start_addr & NAND_BLOCK_MASK) || (size & NAND_BLOCK_MASK))
    {
        return -1;
    }
}
```

```

nand_select();

nand_clear_RnB();

for (i=0; i<10; i++);
for (i=start_addr; i < (start_addr + size);)
{
    j = nand_read_page_ll(buf, i);
    i += j;
    buf += j;
}

nand_deselect();

return 0;
}

```

4) 在/include/asm-arm/arch-s3c24x0/s3c24x0.h 中添加 2440Nand Flash 的定义

```

#gedit include/asm-arm/arch-s3c24x0/s3c24x0.h
/* NAND FLASH (see S3C2410 manual chapter 6) */

struct s3c2410_nand {
    u32  NFCONF;
    u32  NFCMD;
    u32  NFADDR;
    u32  NFDATA;
    u32  NFSTAT;
    u32  NFECC;
};

struct s3c2440_nand {
    u32 NFCONF;
    u32 NFCONT;
    u32 NFCMD;
    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;

};

```

5) 然后，在 board/samsung/mini2440/Makefile 中添加 nand_read.c 的编译选项，使他编译到 u-boot 中，如下：

```
COBJS := mini2440.o flash.o nand_read.o
```

6) 特别注意：在测试之前还要改一个文件，由于 Makefile 版本与先前的有一定的变化，使得对于 24x0 处理器从 nand 启动的遇到问题。也就是网上有人说的：无法运行过 lowlevel_init。其实这个问题是由于编译器将我们自己添加的用于 nandboot 的子函数 nand_read_ll 放到了 4K 之后造成的（到这不理解的话，请仔细看看 24x0 处理器 nandboot 原理）。u-boot 根本没有完成自我拷贝，你可以看 uboot 根目录下的 System.map 文件就可知道原因。

解决办法其实很简单：

将 278 行的 `__LIBS := $(subst $(obj),,$(LIBS)) $(subst $(obj),,$(LIBBOARD))`

改为 `__LIBS := $(subst $(obj),,$(LIBBOARD)) $(subst $(obj),,$(LIBS))`

不过这里还有另外一种方法就是就是修改在 cpu/arm920t/u-boot.lds 中，这个 u-boot 启动连接脚本文件决定了 u-boot 运行的入口地址，以及各个段的存储位置，这也是链接定位的作用。添加下面两行代码的主要目的也是防止编译器把我们自己添加的用于 nandboot 的子函数放到 4K 之后，否则是无法启动的。如下：

```

.text :
{
    cpu/arm920t/start.o (.text)
}

```

```
board/samsung/mini2440/lowlevel_init.o (.text)
board/samsung/mini2440/nand_read.o (.text)
*(.text)
}
```

7) 为了防止不停重启, 需关闭看门狗, 即修改 cpu/arm920t/start.S 这个文件

//在调用 `cpu_init_crit` 前, 添加以下代码

```
#define WTCON 0x53000000

ldr r1, =WTCON

mov r0, #0

str r0, [r1]
```

8) 最后编译 u-boot, 生成 u-boot.bin 文件。然后先将 mini2440 开发板调到 Nor 启动档, 利用 `supervivi` 的 `a` 命令将 u-boot.bin 下载到开发板的 Nand Flash 中, 再把开发板调到 Nand 启动档, 打开电源就从 Nand Flash 启动了, 启动结果图如下:

```
panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1026-2K
Enter your selection:

U-Boot 2010.03 ( 4鏈? 22 2011 - 13:42:02)

DRAM: 64 MB
Flash: 2 MB
*** Warning - bad CRC, using default environment

In: serial
Out: serial
Err: serial
Net: CS8900-0
[PAN_MINI2440]# nand info
Unknown command 'nand' - try 'help'
[PAN_MINI2440]# nand help
Unknown command 'nand' - try 'help'
[PAN_MINI2440]#
```

已连接 0:33:46 自动检测 115200 8-N-1 SCROLL CAPS NUM 辅 打印

从上面的运行图看, 显然现在的 Nand 还不能做任何事情, 而且也没有显示有关 Nand 的任何信息, 所以只能说明上面的这些步骤只是完成了 Nand 移植的 Stage1 部分。

下面我们来添加我们开发板上的 Nand Flash(K9F1GU08B)的 Stage2 部分的有关操作支持。

2、添加 Nand Flash(K9F1GU08B)的有关操作支持

上次移植的 u-boot-2010.03 虽然可以从 NandFlash 启动，但是，在 u-boot 还是没有办法进行操作，主要是没有 NandFlash 的驱动。我的 NandFlash 的型号是三星公司的 K9F1G08U，大小是 128MB，页的大小是 2KB，也就是大页 NandFlash 的读写。在以前的版本中，U-boot 对 NAND Flash 支持新旧两套代码，而我们使用的版本中已经不在支持旧的代码，新代码移植于 Linux 内核 2.6，它更加的智能，可以自动识别更多的 NandFlash 型号。NandFlash 的驱动代码在 drivers/mtd/nand/目录下，由于，这个版本的 u-boot 还没有支持 S3C2440，所以我们需要自己写驱动，不过，我们可以根据 2410 进行改写，只需要加上对 S3C2440 的支持。

1) 在进行移植前，我们必须理解，在对 NandFlash 编写驱动时，各个函数的调用过程是怎么样的，它的入口函数是哪一个？

首先，要让 U-Boot 支持 NAND Flash，要在配置文件 include/configs/mini2440.h 中，定义 CFG_CMD_NAND；我们知道，在 U-Boot 第一阶段完成后，就会跳转到 C 函数 void start_armboot (void) {} 运行，这个函数在 lib_arm/board.c 中进行定义的。在这个函数中我们可以看到，下面的一句代码，这句代码说明只有定义了 CONFIG_CMD_NAND 后，才会调用 nand_init() 函数，对 Nand Flash 进行相应的初始化和识别的过程。

```
void start_armboot (void)
{
    .....
    #if defined(CONFIG_CMD_NAND)
        puts ("NAND ");
        nand_init();      /* go init the NAND */
    #endif
    .....
    /* main_loop() can return to retry autoboot, if so just run it again. */
    for (;;) {
        main_loop ();
    }
```

(1)CONFIG_CMD_NAND 后，就会调用 nand_init() 初始化函数。该函数位于 drivers/mtd/nand/nand.c 中，

```
void nand_init(void)
{
```



```

int i;
unsigned int size = 0;
for (i = 0; i < CONFIG_SYS_MAX_NAND_DEVICE; i++) {
    nand_init_chip(&nand_info[i], &nand_chip[i], base_address[i]);
    size += nand_info[i].size / 1024;
    if (nand_curr_device == -1)
        nand_curr_device = i;
}
printf("NAND: %u MiB\n", size / 1024);

```

(2)在 nand_init()函数中调用同一个文件中的 nand_init_chip()函数。

```

static void nand_init_chip(struct mtd_info *mtd, struct nand_chip *nand, ulong base_addr)
{

```

```

    .....
    int maxchips = CONFIG_SYS_NAND_MAX_CHIPS;
    nand->IO_ADDR_R = nand->IO_ADDR_W = (void __iomem *)base_addr;
    if (board_nand_init(nand) == 0) {
    if (nand_scan(mtd, maxchips) == 0) {
    .....

```

(3)而在 nand_init_chip()函数中调用，我们自己写的 board_nand_init()函数，该函数在我们要改写的文件 s3c2410_nand.c 中，

(4)然后，调用 drivers/mtd/nand/nand_base.c 文件中的 nand_scan()函数。

(5)在 nand_scan()函数中调用同一个文件中的 nand_scan_ident()函数，nand_scan_ident()函数又调用 nand_get_flash_type()函数，得到 NandFlash 芯片的厂家 ID 和芯片 ID。

(6)最后调用函数同一个文件中的 nand_scan_tail()函数，进行对芯片中各个函数操作的初始化。NandFlash 的初始化，驱动也就完成了。

我们所要做的工作就是改写其中的一个 board_nand_init()函数。在改写这个函数之前，我们应该，先配置好，各个变量。

2) 先在 include/configs/mini2440.h 文件中，添加下面的定义和一些设置的参数。

```

/*
* Command line configuration.
*/
#include <config_cmd_default.h>

#define CONFIG_CMD_CACHE
#define CONFIG_CMD_DATE

```

```

#define CONFIG_CMD_ELF

#define CONFIG_CMD_NAND //添加的定义

.....

/*-----
Nand Flash Setting
-----*/

#if defined (CONFIG_CMD_NAND)
#define CONFIG_NAND_S3C2440 //为了使 s3c2440_nand.c 文件编译进去，必须要定
义。
#define CONFIG_SYS_NAND_BASE 0x4e000000 //NandFlash 中的各个寄存器的地址，
基址
#define CONFIG_SYS_MAX_NAND_DEVICE 1 //NandFlash 设备的数目为 1
#define CONFIG_MTD_NAND_VERIFY_WRITE 1
//#define NAND_SAMSUNG_LP_OPTIONS 1
#define NAND_MAX_CHIPS 1 //每个 NandFlash 设备，由 1 个 NandFlash 芯片组成。
#define CONFIG_SYS_64BIT_VSPRINTF
#endif

#define CONFIG_INITRD_TAG
#define CONFIG_CMDLINE_TAG

#define CONFIG_SYS_HUSH_PARSER
#define CONFIG_SYS_PROMPT_HUSH_PS2 ">"
#define CONFIG_CMDLINE_EDITING

#define CONFIG_AUTO_COMPLETE

```

3)因为 2440 和 2410 对 nand 控制器的操作有很大的不同，所以 s3c2410_nand.c 下对 nand 操作的函数就是我们做移植需要实现的部分了，他与具体的 Nand Flash 硬件密切相关。为了区别与 2410，这里我们就重新建立一个 s3c2440_nand.c 文件，在这里面来

实现对 nand 的操作，代码如下：

```
#gedit drivers/mtd/nand/s3c2440_nand.c //新建 s3c2440_nand.c 文件
#include <common.h>

#if 0
#define DEBUGN printf
#else
#define DEBUGN(x, args ...) {}
#endif

#include <nand.h>

#include <asm/arch/s3c24x0_cpu.h> //注意 s3c2410.h 的位置与老版本不一样
#include <asm/io.h>

#define __REGb(x) (*(volatile unsigned char *)(x))
#define __REGi(x) (*(volatile unsigned int *)(x))

#define NF_BASE 0x4e000000 //Nand 配置寄存器基地址
#define S3C2440_ADDR_NALE 0x08
#define S3C2440_ADDR_NCLE 0x0c

#define NFCONF __REGi(NF_BASE + 0x0) //偏移后还是得到配置寄存器基地址
#define NFCONT __REGi(NF_BASE + 0x4) //偏移后得到 Nand 控制寄存器基地址
#define NFCMD __REGb(NF_BASE + 0x8) //偏移后得到 Nand 指令寄存器基地址
#define NFADDR __REGb(NF_BASE + 0xc) //偏移后得到 Nand 地址寄存器基地址
#define NFDATA __REGb(NF_BASE + 0x10) //偏移后得到 Nand 数据寄存器基地址
#define NFMECCD0 __REGi(NF_BASE + 0x14) //偏移后得到 Nand 主数据区域 ECC0
寄存器基地址
#define NFMECCD1 __REGi(NF_BASE + 0x18) //偏移后得到 Nand 主数据区域 ECC1
寄存器基地址
#define NFSECCD __REGi(NF_BASE + 0x1C) //偏移后得到 Nand 空闲区域 ECC 寄存器
基地址
```

```

#define NFSTAT    __REGb(NF_BASE + 0x20) //偏移后得到 Nand 状态寄存器基地址
#define NFSTAT0   __REGi(NF_BASE + 0x24) //偏移后得到 Nand ECC0 状态寄存器基地址
#define NFSTAT1   __REGi(NF_BASE + 0x28) //偏移后得到 Nand ECC1 状态寄存器基地址
#define NFMECC0   __REGi(NF_BASE + 0x2C) //偏移后得到 Nand 主数据区域 ECC0 状态寄存器基地址
#define NFMECC1   __REGi(NF_BASE + 0x30) //偏移后得到 Nand 主数据区域 ECC1 状态寄存器基地址
#define NFSECC    __REGi(NF_BASE + 0x34) //偏移后得到 Nand 空闲区域 ECC 状态寄存器基地址
#define NFSBLK    __REGi(NF_BASE + 0x38) //偏移后得到 Nand 块开始地址
#define NFEBLK    __REGi(NF_BASE + 0x3c) //偏移后得到 Nand 块结束地址

#define NFECC0 __REGb(NF_BASE + 0x2c)
#define NFECC1 __REGb(NF_BASE + 0x2d)
#define NFECC2 __REGb(NF_BASE + 0x2e)

#define S3C2440_NFCONT_nFCE    (1<<1)
#define S3C2440_NFCONT_EN     (1<<0)
#define S3C2440_NFCONT_INTECC (1<<4)
#define S3C2440_NFCONT_MAINECCLOCK (1<<5)

#define S3C2440_NFCONF_TACLS(x) ((x)<<12)
#define S3C2440_NFCONF_TWRPH0(x) ((x)<<8)
#define S3C2440_NFCONF_TWRPH1(x) ((x)<<4)

ulong IO_ADDR_W = NF_BASE;

```

```

static void s3c2440_hwcontrol(struct mtd_info *mtd, int cmd, unsigned int ctrl)
{
    //struct nand_chip *chip = mtd->priv;
    DEBUGN("hwcontrol(): 0x%02x 0x%02x\n", cmd, ctrl);
    if (ctrl & NAND_CTRL_CHANGE) {
        IO_ADDR_W = NF_BASE;

        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;

        //chip->IO_ADDR_W = (void *)IO_ADDR_W;

        if (ctrl & NAND_NCE)
            NFCONT &= ~S3C2440_NFCONT_nFCE;    //使能 nand flash
        else
            NFCONT |= S3C2440_NFCONT_nFCE;    //禁止 nand flash
    }

    if (cmd != NAND_CMD_NONE)
        writeb(cmd, (void *)IO_ADDR_W);
}

static int s3c2440_dev_ready(struct mtd_info *mtd)
{
    DEBUGN("dev_ready\n");
    return (NFSTAT & 0x01);
}

int board_nand_init(struct nand_chip *nand)
{
    u_int32_t cfg;

```

```

u_int8_t tacs, twrph0, twrph1;

struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();

struct s3c2440_nand *nand_reg = s3c2410_get_base_nand();
//s3c2410_get_base_nand()在 s3c2410.h 中定义

DEBUGN("board_nand_init()\n");

clk_power->CLKCON |= (1 << 4);

/* initialize hardware */
twrph0 = 4;twrph1 = 2;tacs = 0;

cfg = 0;

cfg |= S3C2440_NFCONF_TACLS(tacs-1);

cfg |= S3C2440_NFCONF_TWRPH0(twrph0-1);

cfg |= S3C2440_NFCONF_TWRPH1(twrph1-1);

NFCONF = cfg;

NFCONT= (0<<13) | (0<<12) | (0<<10) | (0<<9) | (0<<8) | (1<<6) | (0<<5) | (1<<4)
|(0<<1) |(1<<0);

/* initialize nand_chip data structure 这个地址就是 NFDATA 寄存器的地址* */
nand->IO_ADDR_R = nand->IO_ADDR_W = (void *)0x4e000010;

/* read_buf and write_buf are default */

/* read_byte and write_byte are default */

/* hwcontrol always must be implemented */

nand->cmd_ctrl = s3c2440_hwcontrol;

nand->dev_ready = s3c2440_dev_ready;

DEBUGN("end of nand_init\n");

return 0;
}

```

4) 然后, 在 drivers/mtd/nand/Makefile 文件中添加 s3c2440_nand.c 的编译项, 如下:

```
# gedit drivers/mtd/nand/Makefile

include $(TOPDIR)/config.mk

LIB := $(obj)libnand.a

ifdef CONFIG_CMD_NAND // 如果定义了 CONFIG_CMD_NAND，下面的六个文件都会被

COBJS-y += nand.o //编译进去，而下面的必须在定义了相应的配置常量以后才能
COBJS-y += nand_base.o //编译进去
COBJS-y += nand_bbt.o
COBJS-y += nand_ecc.o
COBJS-y += nand_ids.o
COBJS-y += nand_util.o

COBJS-$(CONFIG_NAND_ATMEL) += atmel_nand.o
COBJS-$(CONFIG_DRIVER_NAND_BFIN) += bfin_nand.o
COBJS-$(CONFIG_NAND_DAVINCI) += davinci_nand.o
COBJS-$(CONFIG_NAND_FSL_ELBC) += fsl_elbc_nand.o
COBJS-$(CONFIG_NAND_FSL_UPM) += fsl_upm.o
COBJS-$(CONFIG_NAND_KIRKWOOD) += kirkwood_nand.o
COBJS-$(CONFIG_NAND_MPC5121_NFC) += mpc5121_nfc.o
COBJS-$(CONFIG_NAND_NDFC) += ndfc.o
COBJS-$(CONFIG_NAND_NOMADIK) += nomadik.o

//而只有定义了 CONFIG_NAND_S3C2440 后，s3c2440_nand.c 文件才能被编译进去。
COBJS-$(CONFIG_NAND_S3C2440) += s3c2440_nand.o

COBJS-$(CONFIG_NAND_S3C64XX) += s3c64xx.o
COBJS-$(CONFIG_NAND_OMAP_GPMC) += omap_gpmc.o
COBJS-$(CONFIG_NAND_PLAT) += nand_plat.o
endif
```

5) 最后，重新编译 u-boot 并使用 `supervivi` 的 `a` 命令下载到 Nand Flash 中，把开发板调到 Nand 档从 Nand 启动，启动结果图如下：

```

panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

*** Warning - bad CRC or NAND, using default environment

In:    serial
Out:   serial
Err:   serial
Net:   CS8900-0
[PAN_MINI2440]# nand help
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
nand read/write 'size' bytes starting at offset 'off'
to/from memory address 'addr', skipping bad blocks.
nand erase [clean] [off size] - erase 'size' bytes from
offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand dump [l.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
[PAN_MINI2440]#

```

从上图可以看出，现在 u-boot 已经对我们开发板上 128M 的 Nand Flash 完全支持了。Nand 相关的基本命令也都可以正常使用了，但有一个警告信息“*** Warning - bad CRC or NAND, using default environment”，我们知道，这是因为我们还没有将 u-boot 的环境变量保存 nand 中的缘故，那现在我们就用 u-boot 的 saveenv 命令来保存环境变量，结果却失败了。这是由于 u-boot 在默认的情况下把环境变量都是保存到 Nor Flash 中的，而我们是要保存到 Nand 中，所以我们要修改代码，让他保存到 Nand 中，如下：

```

#gedit include/configs/mini2440.h

//注释掉环境变量保存到 Flash 的宏(注意：如果你要使用上一篇中的从 Nor 启动的
saveenv 命令，则要恢复这些 Flash 宏定义)

//#define CONFIG_ENV_IS_IN_FLASH 1

//#define CONFIG_ENV_SIZE      0x10000 /* Total Size of Environment Sector */

//添加环境变量保存到 Nand 的宏(注意：如果你要使用上一篇中的从 Nor 启动的 saveenv
命令，则不要这些 Nand 宏定义)

#define CONFIG_ENV_IS_IN_NAND 1

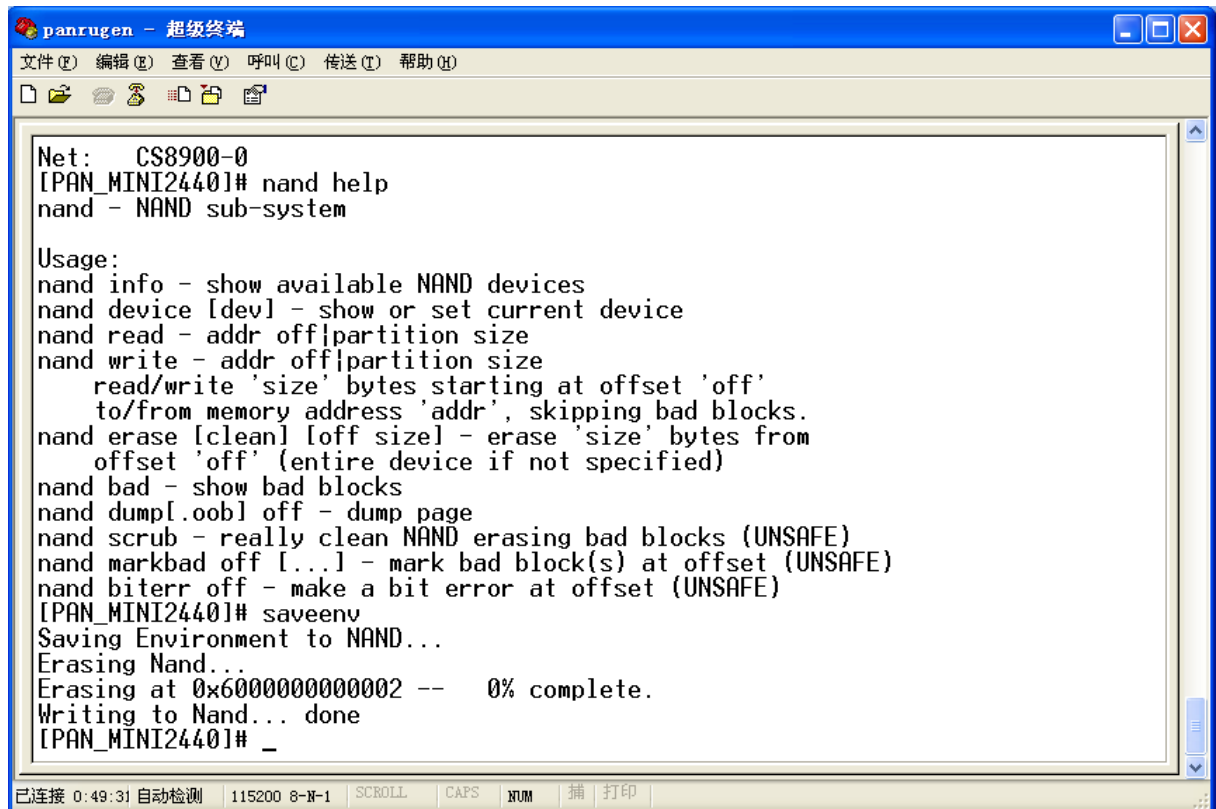
#define CONFIG_ENV_OFFSET      0x60000 //将环境变量保存到 nand 中的 0x60000 位

```


置

```
#define CONFIG_ENV_SIZE      0x20000 /* Total Size of Environment Sector */
```

重新编译 u-boot，下载到 nand 中，启动开发板再来保存环境变量，如下：



```
panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

Net: CS8900-0
[PAN_MINI2440]# nand help
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
               read/write 'size' bytes starting at offset 'off'
               to/from memory address 'addr', skipping bad blocks.
nand erase [clean] [off size] - erase 'size' bytes from
               offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
[PAN_MINI2440]# saveenv
Saving Environment to NAND...
Erasing Nand...
Erasing at 0x600000000000002 -- 0% complete.
Writing to Nand... done
[PAN_MINI2440]# _
```

已连接 0:49:31 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

可以看到，现在成功保存到 Nand 中了，为了验证，我们重新启动开发板，那条警告信息现在没有了，如下：

```

[w] Download WinCE NK.bin
[d] Download & Run
[z] Download zImage into RAM
[g] Boot linux from RAM
[f] Format the nand flash
[b] Boot the system
[s] Set the boot parameters
[u] Backup NAND Flash to HOST through USB(upload)
[r] Restore NAND Flash from HOST through USB
[q] Goto shell of vivi
[i] Version: 1026-2K
Enter your selection:

U-Boot 2010.03 ( 4鏈? 22 2011 - 13:55:27)

DRAM: 64 MB
Flash: 2 MB
NAND: NAND_ECC_NONE selected by board driver. This is not recommended !!
128 MiB
In: serial
Out: serial
Err: serial
Net: CS8900-0
[PAN_MINI2440]#

```

(四) 支持写 xmodem 协议

这次的移植修改主要增加对 xmodem 协议的支持。xmodem 协议是一种文件传输协议（常用于串口通信），因其简单性和较好的性能而被广泛应用。xmodem 协议通过 console 口传输文件，支持 128 字节和 1k 字节两种类型的数据包，并且支持一般校验和 crc 两种校验方式，在出现数据包错误的情况下支持多次重传（一般为 10 次）。

xmodem 协议传输由接收程序和发送程序完成。先由接收程序发送协商字符，协商校验方式，协商通过之后发送程序就开始发送数据包，接收程序接收到完整的一个数据包之后按照协商的方式对数据包进行校验。校验通过之后发送确认字符，然后发送程序继续发送下一包；如果校验失败，则发送否认字符，发送程序重传此数据包。

1、我们主要是依照 loady 的实现过程编写代码。首先使用 U_BOOT_CMD 宏来增加 loadx 命令。修改文件 common/cmd_load.c 文件（1072 行左右），仿照 loadx 的编写如下代码。

```

#gedit common/cmd_load.c

#if defined(CONFIG_CMD_LOADB)

U_BOOT_CMD(

    loadb, 3, 0, do_load_serial_bin,

```

```

"load binary file over serial line (kermit mode)",

"[ off ] [ baud ]\n"

"    - load binary file over serial line"

" with offset 'off' and baudrate 'baud'"

);

#ifdef(ENABLE_CMD_LOADB_X)

U_BOOT_CMD(

    loadx,3,0,do_load_serial_bin,

    "loadx - load binary file over serial line (xmodem mode)\n",

    "[off][baud]\n"

    "- load binary file over serial line"

    "with offset 'off' and baudrate 'baud'\n"

);

#endif

U_BOOT_CMD(

loady,3,0,do_load_serial_bin,

"load binary file over serial line (ymodem mode)",

"[ off ] [ baud ]\n"

"- load binary file over serial line"

" with offset 'off' and baudrate 'baud'"

);

```

2、然后，在 do_load_serial_bin 函数中增加对 loadx 命令的处理分支。也是按照 Loady 的实现来修改。在 478 行附近的位置，添加下面的代码。

```

#ifdef ENABLE_CMD_LOADB_X

if(strcmp(argv[0],"loadx")==0){

    printf("## Read for binary (xmodem) download"

```

```

        "to 0x%08IX at %d bps...\n",
        offset,
        load_baudrate);

    addr = load_serial_xmodem(offset);
} else if(strcmp(argv[0], "loady")==0){
#else
    if (strcmp(argv[0], "loady")==0){
#endif

    printf ("## Ready for binary (ymodem) download "
            "to 0x%08IX at %d bps...\n",
            offset,
            load_baudrate);

    addr = load_serial_ymodem (offset);

    } else {

    printf ("## Ready for binary (kermit) download "
            "to 0x%08IX at %d bps...\n",
            offset,
            load_baudrate);

    addr = load_serial_bin (offset);

    if (addr == ~0) {

        load_addr = 0;

        printf ("## Binary (kermit) download aborted\n");

        rcode = 1;

    } else {

        printf ("## Start Addr      = 0x%08IX\n", addr);

```

```

        load_addr = addr;

    }

}

```

3、接着就是添加 load_serial_xmodem()函数

//先在 36 行附近添加声明。

```

#if defined(CONFIG_CMD_LOADB)
static ulong load_serial_ymodem (ulong offset);
static ulong load_serial_xmodem (ulong offset);
#endif

//最后在 977 行，仿照 load_serial_ymodem() 函数，编写 load_serial_xmodem() 函数。
这里主要就将局部数组 ymodemBuf 改名为 xmodemBuf, 并在后面使用到的地方统一修改,
info.mode 的值从 xyzModem_ymodem 改为 xyzModem_xmodem.

#if defined ENABLE_CMD_LOADB_X
static ulong load_serial_xmodem(ulong offset)
{
    int size;
    char buf[32];
    int err;
    int res;
    connection_info_t info;
    char xmodemBuf[1024];
    ulong store_addr = ~0;
    ulong addr = 0;
    size = 0;
    info.mode = xyzModem_xmodem;
    res = xyzModem_stream_open(&info, &err);
    if (!res) {
        while ((res =

```

```
xyzModem_stream_read(xmodemBuf, 1024, &err) > 0) {  
    store_addr = addr + offset;  
    size += res;  
    addr += res;  
#ifndef CONFIG_SYS_NO_FLASH  
    if (addr2info (store_addr)) {  
        int rc;  
        rc = flash_write ((char *) xmodemBuf,  
                           store_addr, res);  
        if (rc != 0) {  
            flash_perror (rc);  
            return (~0);  
        }  
    } else  
#endif  
    {  
        memcpy ((char *) (store_addr), xmodemBuf, res);  
    }  
}  
} else {  
    printf ("%s\n", xyzModem_error (err));  
}  
xyzModem_stream_close (&err);  
xyzModem_stream_terminate (false, &getcxdem);  
flush_cache (offset, size);  
printf ("## Total Size = 0x%08x = %d Bytes\n", size, size);
```

```

    sprintf(buf, "%X", size);

    setenv("filesize", buf);

    return offset;
}

#endif

```

4、最后，不要忘了在 mini2440.h 文件中定义 ENABLE_CMD_LOADB_X 宏。重新编译，下载到 NANDFlash 中，就可以使用 loadx 命令，利用串口来下载文件了。效率要比 JTAG 高很多。

```
#gedit include/configs/mini2440.h
```

```
//最后一行加上 xmodem 的定义
```

```
#define ENABLE_CMD_LOADB_X 1
```

（五）支持 DM9000 网卡

在这一篇中，我们首先让开发板对 CS8900 或者 DM9000X 网卡的支持，然后再分析实现 u-boot 怎样来引导 Linux 内核启动。因为测试 u-boot 引导内核我们要用到网络下载功能。

1、u-boot 对 DM9000 网卡的支持。

u-boot-2010.03 版本已经对 CS8900 和 DM9000 网卡有比较完善的代码支持(代码在 drivers/net/目录下)，不过在配置的时候，默认的是配置的 CS8900 的网卡，所以还是需要稍微的修改就可以。首先在/include/configs/mini2440.h 文件中（62 行左右），添加关于 DM9000 网卡的配置变量。

```
#gedit include/configs/mini2440.h
```

```
/*
```

```
* Hard
```

```
ware drivers
```

```
*/
```

```
#define CONFIG_NET_MULTI 1
```

```
//屏蔽掉 u-boot 默认对 CS8900 网卡的支持
```

```
//#define CONFIG_DRIVER_CS8900 1 /* we have a CS8900 on-board */
```

```
//#define CS8900_BASE 0x19000300
```

```
//#define CS8900_BUS16 1 /* the Linux driver does accesses as shorts */
```

//添加 u-boot 对 DM9000X 网卡的支持，其中 CONFIG_DM9000_BASE 宏是最重要的，因为这个就网卡的地址，不同的网卡有不同的地址，DM9000 访问的基址为 0x20000000, 之所以再偏移 0x300 是由它的特性决定的。一般情况下，只有配正确这个地址，网卡的移植就会很顺利。mini2440 的 BANK4 连接的外设就是网卡 DM9000，BANK4 的基地址就是 0x20000000 和 0x20000004

```
#define CONFIG_DRIVER_DM9000 1
```

```
#define CONFIG_DM9000_NO_SROM 1
```

```
#define CONFIG_DM9000_BASE 0x20000300 //网卡片选地址
```

```
#define DM9000_IO CONFIG_DM9000_BASE
```

```
#define DM9000_DATA (CONFIG_DM9000_BASE + 4) //网卡数据地址
```

```
//#define CONFIG_DM9000_USE_16BIT 1
```

注意：

u-boot-2010.03 可以自动检测 DM9000 网卡的位数，根据开发板原理图可知网卡的数据位为 16 位，并且网卡位于 CPU 的 BANK4 上，所以只需在 board/samsung/mini2440/lowlevel_init.S 中设置#define B4_BWSCON (DW16)即可，不需要此处的#define CONFIG_DM9000_USE_16BIT 1

```
//给 u-boot 加上 ping 命令，用来测试网络通不通
```

```
/*
```

```
* Command line configuration.
```

```
*/
```

```
#include <config_cmd_default.h>
```



```

#define CONFIG_CMD_CACHE

#define CONFIG_CMD_DATE

#define CONFIG_CMD_ELF

#define CONFIG_CMD_NAND

#define CONFIG_CMD_PING

//恢复被注释掉的网卡 MAC 地址和修改你合适的开发板 IP 地址

#define CONFIG_BOOTDELAY 3

/*#define CONFIG_BOOTARGS "root=ramfs devfs=mount console=ttySA0,9600" */

#define CONFIG_ETHADDR      08:90:90:90:90:90  //开发板 MAC 地址

#define CONFIG_NETMASK      255.255.255.0

#define CONFIG_IPADDR       192.168.1.226 //开发板 IP 地址

#define CONFIG_SERVERIP     192.168.1.200 //Linux 主机 IP 地址

/*#define CONFIG_BOOTFILE   "elinos-lart" */

/*#define CONFIG_BOOTCOMMAND "tftp; bootm" */

```

添加板载 DM9000 网卡初始化代码，如下：

```

#gedit board/samsung/mini2440/mini2440.c

#include <net.h>

#include <netdev.h>

#if 0

#ifdef CONFIG_CMD_NET

int board_eth_init(bd_t *bis)

{int rc = 0;

#ifdef CONFIG_CS8900

rc = cs8900_initialize(0, CONFIG_CS8900_BASE);

```

```

#endif

return rc;

}

#endif

#endif

#ifdef CONFIG_DRIVER_DM9000

int board_eth_init(bd_t *bis)

{

    return dm9000_initialize(bis);

}

#endif

```

修改 MD9000 网卡驱动代码，如下：

```
#gedit drivers/net/dm9000x.c
```

#if 0 //屏蔽掉 dm9000_init 函数中的这一部分（363 行左右），不然使用网卡的时候会报“could not establish link”的错误

```

i = 0;

while (!(phy_read(1) & 0x20)) { /* autonegation complete bit */

    udelay(1000);

    i++;

    if (i == 10000) {

        printf("could not establish link ");

        return 0;

    }

}

#endif

```

然后重新编译 u-boot，下载到 Nand 中从 Nand 启动，查看启动信息和环境变量并使用 ping 命令测试网卡，操作如下：

```

panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)

Enter your selection:
U-Boot 2010.03 ( 4鏈? 22 2011 - 14:14:13)

DRAM: 64 MB
Flash: 2 MB
NAND: NAND_ECC_NONE selected by board driver. This is not recommended !!
128 MiB
In: serial
Out: serial
Err: serial
Net: dm9000
IPAN_MINI24401# ping 192.168.2.200
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:00:00:00:00:00
operating at 100M full duplex mode
*** ERROR: 'ethaddr' not set
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:00:00:00:00:00
operating at 100M full duplex mode
ping failed: host 192.168.2.200 is not alive
IPAN_MINI24401# _
已连接 1:10:34 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

```

2、u-boot 对 ping 功能的支持

可以看到，启动信息里面显示了 Net: dm9000，printenv 查看的环境变量也和 include/configs/mini2440.h 中设置的一致。但是现在有个问题就是 ping 不能通过。出现问题的地方可能是 DM9000 网卡驱动中关闭网卡的地方，如是就试着修改代码如下：

```

#gedit drivers/net/dm9000x.c //屏蔽掉 dm9000_halt 函数中的内容

static void dm9000_halt(struct eth_device *netdev)
{
    //DM9000_DBG("%sn", __func__);

    /* RESET devie */

    //phy_write(0, 0x8000); /* PHY RESET */

    //DM9000_iow(DM9000_GPR, 0x01); /* Power-Down PHY */

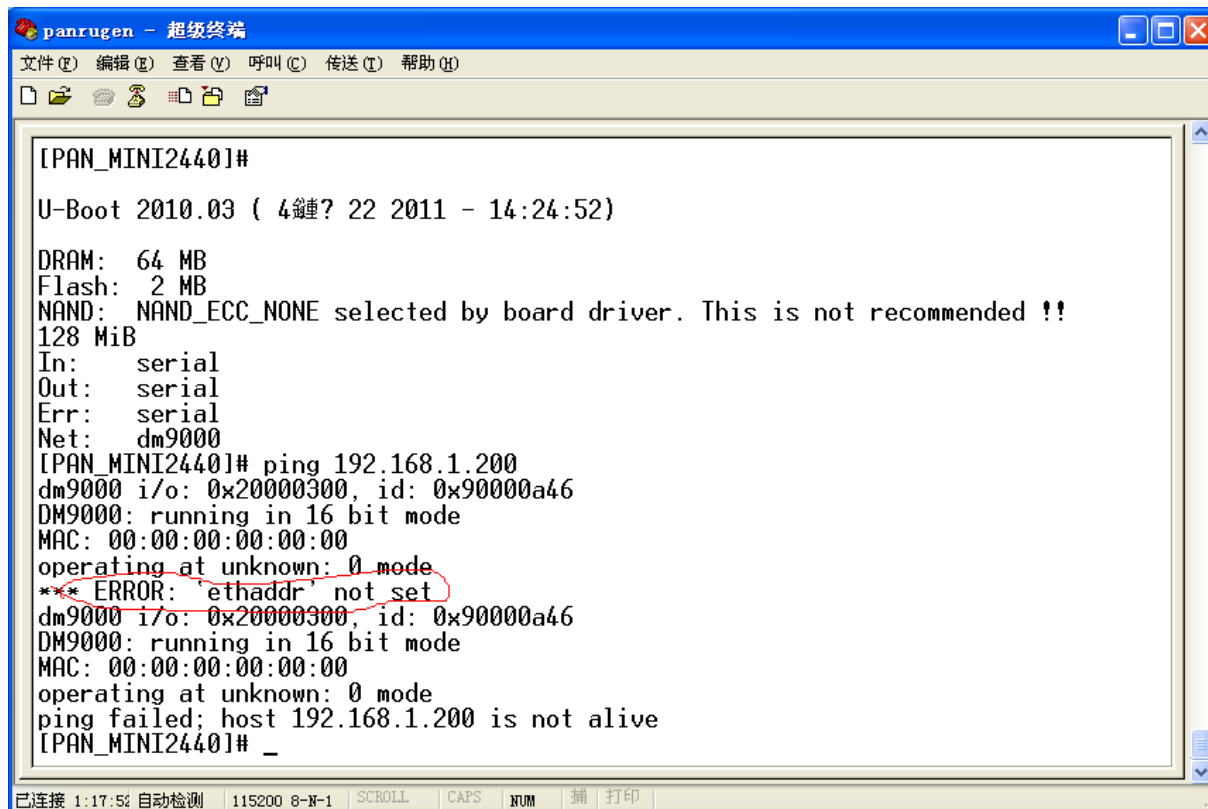
    //DM9000_iow(DM9000_IMR, 0x80); /* Disable all interrupt */

```

```
//DM9000_iow(DM9000_RCR, 0x00); /* Disable RX */
}
```

重新编译 u-boot，下载到 Nand 中从 Nand 启动。

注：若提示错误“ERROR: `ethaddr' not set”，如下图：



```
panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[PAN_MINI2440]#
U-Boot 2010.03 ( 4鏈? 22 2011 - 14:24:52)
DRAM: 64 MB
Flash: 2 MB
NAND: NAND_ECC_NONE selected by board driver. This is not recommended !!
128 MiB
In: serial
Out: serial
Err: serial
Net: dm9000
[PAN_MINI2440]# ping 192.168.1.200
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:00:00:00:00:00
operating at unknown: 0 mode
*** ERROR: `ethaddr' not set ***
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 00:00:00:00:00:00
operating at unknown: 0 mode
ping failed; host 192.168.1.200 is not alive
[PAN_MINI2440]# _
```

则通过 NOR flash 中的菜单 x，擦除原来 NAND flash 中的内容，然后重新移植 u-boot.bin 即可，这时就可以 ping 通了。

```

[q] Goto shell of vivi
[il Version: 1026-2K
Enter your selection:

U-Boot 2010.03 ( 4鏈? 22 2011 - 14:24:52)

DRAM: 64 MB
Flash: 2 MB
NAND: NAND_ECC_NONE selected by board driver. This is not recommended !!
128 MiB
*** Warning - bad CRC or NAND, using default environment

In: serial
Out: serial
Err: serial
Net: dm9000
[PAN_MINI2440]# ping 192.168.1.200
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:90:90:90:90:90
operating at unknown: 0 mode
Using dm9000 device
host 192.168.1.200 is alive
[PAN_MINI2440]#

```

已连接 1:22:24 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

3、tftp 下载功能测试

先在 Linux 主机上安装 tftp，然后将 uImage 复制到/tftpboot/中，最后在开发板上使用 tftp 进行下载，如下图所示：

```

Err: serial
Net: dm9000
[PAN_MINI2440]# ping 192.168.1.200
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:90:90:90:90:90
operating at unknown: 0 mode
Using dm9000 device
host 192.168.1.200 is alive
[PAN_MINI2440]# tftp 0x30008000 uImage
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: 08:90:90:90:90:90
operating at unknown: 0 mode
Using dm9000 device
TFTP from server 192.168.1.200; our IP address is 192.168.1.226
Filename 'uImage'.
Load address: 0x30008000
Loading: #####
done
Bytes transferred = 2286852 (22e504 hex)
[PAN_MINI2440]#

```

已连接 1:25:24 自动检测 115200 8-N-1 SCROLL CAPS NUM 捕 打印

（六）u-boot 对 yaffs/yaffs2 文件系统下载的支持

通常一个 Nand Flash 存储设备由若干块组成，1 个块由若干页组成。一般 128MB 以下容量的 Nand Flash 芯片，一页大小为 528B，被依次分为 2 个 256B 的主数据区和 16B 的额外空间，yaffs 文件系统对其支持的较好；128MB 以上容量的 Nand Flash 芯片，一页大小通常为 2KB+64B，2KB 的主数据区和 64B 的额外空间，yaffs2 文件系统对其支持的较好。由于 Nand Flash 出现位反转的概率较大，一般在读写时需要使用 ECC 进行错误检验和恢复。

Yaffs/yaffs2 文件系统的设计充分考虑到 Nand Flash 以页为存取单位等特点，将文件组织成固定大小的段(Chunk)。以 2KB 的页为例，Yaffs/yaffs2 文件系统使用前 2KB 存储数据和 64B 的额外空间存放数据的 ECC 和文件系统的组织信息等(称为 OOB 数据)。通过 OOB 数据，不但能实现错误检测和坏块处理，同时还可以避免加载时对整个存储介质的扫描，加快了文件系统的加载速度。以下是 Yaffs/yaffs2 文件系统 OOB 数据使用：

域	描述	512B 页	2KB 页
chunkId	chunk 编号	20bit	4B
block sequenceNumber	块序列号	无	4B
serialNumber	数据状态跟踪	2bit	无
byteCount	页已用字节数	10bit	4B
objectId	索引节点号	18bit	4B
blockStatus	块状态	1B	无
pageStatus	页状态	1B	无
tagsEcc	标签的ecc	12bit	12B
ecc	数据的ecc	2*3B	8*3B
total	和	16B	52B

好了，在了解 Nand Flash 组成和 Yaffs/yaffs2 文件系统结构后，我们再回到 u-boot 中。目前，在 u-boot 中已经有对 Cramfs、Jffs2 等文件系统的读写支持，但与带有数据校验等功能的 OOB 区的 Yaffs/Yaffs2 文件系统相比，他们是将所有文件数据简单的以线性表形式组织的。所以，我们只要在此基础上通过修改 u-boot 的 Nand Flash 读写命令，增加处理 OOB 区域数据的功能，即可以实现对 Yaffs/Yaffs2 文件系统的读写支持。

实现对 Yaffs 或者 Yaffs2 文件系统的读写支持步骤如下：

1)在 include/configs/mini2440.h 头文件中定义一个管理对 Yaffs2 支持的宏和开启 u-boot 中对 Nand Flash 默认分区的宏，如下：

```
#gedit include/configs/mini2440.h //添加到文件末尾即可

#define CONFIG_MTD_NAND_YAFFS2 1 //定义一个管理对 Yaffs2 支持的宏

//开启 Nand Flash 默认分区，注意此处的分区要和你的内核中的分区保持一致

#define MTDIDS_DEFAULT "nand0=nandflash0"

#define MTDPARTS_DEFAULT "mtdparts=nandflash0:384k(bootloader), "\
    "128k(params), "\
    "5m(kernel), "\
    "-(root)" //这里要和 Linux 内核中的分区信息相一致
```

2)在原来对 Nand 操作的命令集列表中添加 Yaffs2 对 Nand 的写命令，如下：

```
#gedit common/cmd_nand.c //在 U_BOOT_CMD 中（478 行左右）添加

U_BOOT_CMD(nand, CONFIG_SYS_MAXARGS, 1, do_nand,
    "NAND sub-system",
    "info - show available NAND devices\n"
    "nand device [dev] - show or set current device\n"
    "nand read - addr off|partition size\n"
    "nand write - addr off|partition size\n"
    " read/write 'size' bytes starting at offset 'off'\n"
    " to/from memory address 'addr', skipping bad blocks.\n"

//注意：这里只添加了 yaffs2 的写命令，因为我们只用 u-boot 下载(即写)功能，所以我们
//没有添加 yaffs2 读的命令

#if defined(CONFIG_MTD_NAND_YAFFS2)
    /*"nand read[.yaffs[2]] addr off size - read the `size' byte yaffs image starting\n"
    /*      at offset `off' to memory address `addr' (.yaffs2 for 512+16 NAND)\n"
    "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 512+16 NAND)\n"
#endif

"nand erase [clean] [off size] - erase 'size' bytes from\n"
```

```

"offset 'off' (entire device if not specified)\n"
"nand bad - show bad blocks\n"
"nand dump[.oob] off - dump page\n"
"nand scrub - really clean NAND erasing bad blocks (UNSAFE)\n"
"nand markbad off [...] - mark bad block(s) at offset (UNSAFE)\n"
"nand biterr off - make a bit error at offset (UNSAFE)"
#ifdef CONFIG_CMD_NAND_LOCK_UNLOCK
    "\n"
    "nand lock [tight] [status]\n"
    "bring nand to lock state or display locked pages\n"
    "nand unlock [offset] [size] - unlock section"
#endif
);

```

接着，在该文件中对 nand 操作的 do_nand 函数中（381 行左右）添加 yaffs2 对 nand 的操作，如下：

```

if (strcmp(cmd, "read", 4) == 0 || strcmp(cmd, "write", 5) == 0)
{
    int read;
    if (argc < 4)
        goto usage;
    addr = (ulong)simple_strtoul(argv[2], NULL, 16);
    read = strcmp(cmd, "read", 4) == 0; /* 1 = read, 0 = write */
    printf("\nNAND %s: ", read ? "read" : "write");
    if (arg_off_size(argc - 3, argv + 3, nand, &off, &size) != 0)
        return 1;
    s = strchr(cmd, '.');
    if (!s || !strcmp(s, ".jffs2") || !strcmp(s, ".e") || !strcmp(s, ".i"))
    {
        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);

}

//添加 yaffs2 相关操作，注意该处又关联到 nand_write_skip_bad 函数
#ifdef CONFIG_MTD_NAND_YAFFS2

    else if (s != NULL && (!strcmp(s, ".yaffs2")))

    {

        nand->rw_oob = 1;

        nand->skipfirstblk = 1;

        //写入 yaffs，不支持读入

        ret = nand_write_skip_bad(nand, off, &size, (u_char *)addr);

        nand->skipfirstblk = 0;

        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

    };

    if (read)

        ret = nand->read_oob(nand, off, &ops);

    else

        ret = nand->write_oob(nand, off, &ops);

}

```

```

else
{
    printf("Unknown nand command suffix '%s'.\n", s);
    return 1;
}

printf(" %zu bytes %s: %s\n", size, read ? "read" : "written", ret ? "ERROR" : "OK");
return ret == 0 ? 0 : 1;
}

```

3) 在 include/linux/mtd/mtd.h 头文件的 mtd_info 结构体中添加上面用到 rw_oob 和 skipfirstblk 数据成员，如下：

#gedit include/linux/mtd/mtd.h //在 mtd_info 结构体中添加

```

struct mtd_info {
    u_char type;
    u_int32_t flags;
    uint64_t size;    /* Total size of the MTD */

    #if defined(CONFIG_MTD_NAND_YAFFS2)
        u_char rw_oob;
        u_char skipfirstblk;
    #endif

    /* "Major" erase size for the device. Naive users may take this
     * to be the only erase size available, or may use the more detailed
     * information below if they desire
     */
    u_int32_t erasesize;

    /* Minimal writable flash unit size. In case of NOR flash it is 1 (even
     * though individual bits can be cleared), in case of NAND flash it is
     * one NAND page (or half, or one-fourths of it), in case of ECC-ed NOR
     * it is of ECC block size, etc. It is illegal to have writesize = 0.
     * Any driver registering a struct mtd_info must ensure a writesize of

```

```

* 1 or larger.

*/

u_int32_t writesize;

```

4) 在第二步关联的 `nand_write_skip_bad` 函数中（472 行左右）添加对 Nand OOB 的相关操作，如下：

```

#gedit drivers/mtd/nand/nand_util.c //在 nand_write_skip_bad 函数中添加

int nand_write_skip_bad(nand_info_t *nand, loff_t offset, size_t *length, u_char *buffer)
{
    int rval;
    size_t left_to_write = *length;
    size_t len_incl_bad;
    u_char *p_buffer = buffer;

//这段程序主要是从 yaffs 映像中提取要写入的正常数据的长度，即不包括 oob 数据的数据长度

#ifdef CONFIG_MTD_NAND_YAFFS2 //addr yaffs2 file system support
    if(nand->rw_oob == 1)
    {
        size_t oobsize = nand->oobsize; //定义 oobsize 的大小
        size_t datasize = nand->writesize; //可用的数据的大小
        int datapages = 0;

//长度不是 528 整数倍，认为数据出错。文件大小必须要是（512+16）的整数倍
        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

/* Reject writes, which are not page aligned */

    if ((offset & (nand->writesize - 1)) != 0 || (*length & (nand->writesize - 1)) != 0)
    {
        printf ("Attempt to write non page aligned data\n");
        return -EINVAL;
    }

    len_incl_bad = get_len_incl_bad (nand, offset, *length);

    if ((offset + len_incl_bad) >= nand->size) {
        printf ("Attempt to write outside the flash area\n");
        return -EINVAL;
    }

    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;
    }

#ifdef CONFIG_MTD_NAND_YAFFS2 //add yaffs2 file system support
    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;

        WATCHDOG_RESET ();

        if (nand_block_isbad (nand, offset & ~(nand->erasesize - 1))) {

            printf ("Skip bad block 0x%08llx\n",offset & ~(nand->erasesize - 1));

            offset += nand->erasesize - block_offset;

            continue;

        }

        //如需跳过第一个好块，则跳过第一个好块。

        #if defined(CONFIG_MTD_NAND_YAFFS2) //add yaffs2 file system support

            if(nand->skipfirstblk==1)

            {

                nand->skipfirstblk=0;

                printf("skip 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("\r Writing at 0x%lx --",offset);

    rval = nand_write (nand, offset, &write_size, p_buffer);

    if (rval != 0) {

        printf ("NAND write to offset %lx failed %d\n",offset, rval);

        *length -= left_to_write;

        return rval;

    }

    left_to_write -= write_size;

    printf("%d%% is complete",100-(left_to_write/(*length/100)));

    offset += write_size;

#ifdef CONFIG_MTD_NAND_YAFFS2 //add yaffs2 file system support

    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

}
```

```
return 0;

}
```

5)在第四步 nand_write_skip_bad 函数中看到又对 nand_write 函数（2095 行左右）进行了访问，所以这一步是到 nand_write 函数中添加对 yaffs2 的支持，如下：

```
#gedit drivers/mtd/nand/nand_base.c    //在 nand_write 函数中添加
static int nand_write(struct mtd_info *mtd, loff_t to, size_t len, size_t *retlen, const uint8_t
*buf)
{
    struct nand_chip *chip = mtd->priv;
    int ret;

    //此段代码是将要写入的数据中的正常数据移到 buf 中的前段，把 oob 数据移到后段。
    #if defined(CONFIG_MTD_NAND_YAFFS2) //add yaffs2 file system support

        int oldopsmode = 0;
        if(mtd->rw_oob==1)
        {
            int i = 0;
            int datapages = 0;
            size_t oobsize = mtd->oobsize;//定义 oobsize 的大小
            size_t datasize = mtd->>writesize;//定义正常的数据区的大小
            uint8_t oobtemp[oobsize];
            datapages = len / (datasize);//传进来的 len 是没有包括 oob 的数据长度
            for(i = 0; i < (datapages); i++)
            {
                memcpy((void *)oobtemp, (void *)(buf + datasize * (i + 1)), oobsize);
                memmove((void *)(buf + datasize * (i + 1)), (void *)(buf + datasize * (i + 1) +
oobsize), (datapages - (i + 1)) * (datasize) + (datapages - 1) * oobsize);
                memcpy((void *) (buf+(datapages) * (datasize + oobsize) - oobsize), (void
*)(oobtemp), 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;
#ifdef CONFIG_MTD_NAND_YAFFS2
    if(mtd->rw_oob!=1)
    {
        chip->ops.oobbuf = NULL;
    }
    else
    {
        chip->ops.oobbuf = (uint8_t *) (buf+len); //将 oob 缓存的指针指向 buf 的
后段，即 oob 数据区的起始地址。
        chip->ops.ooblen = mtd->oobsize;
        oldopsmode = chip->ops.mode;
        chip->ops.mode = MTD_OOB_RAW; //将写入模式改为直接书写 oob 区，即写入数
据时，不进行 ECC 校验的计算和写入。（yaffs 映像的 oob 数据中，本身就带有 ECC 校验）
    }
#else
    chip->ops.oobbuf = NULL;
#endif
    ret = nand_do_write_ops(mtd, to, &chip->ops);
    *retlen = chip->ops.retlen;

```



```

nand_release_device(mtd);

#ifdef CONFIG_MTD_NAND_YAFFS2

    chip->ops.mode = oldopsmode;//恢复原模式

#endif

return ret;
}

```

到此，U-Boot 对 YAFFS 文件系统的支持就已经完成了，编译以后，下载到 NANDFlash 中启动，输入 `nand help` 命令就可以看到有 `nand write[.yaffs2]` 的命令选项，现在就可以使用该命令就行烧写 yaffs 文件系统的映像了。

```

panrugen - 超级终端
文件(F) 编辑(E) 查看(V) 呼叫(C) 传送(T) 帮助(H)
[IPAN_MINI2440]# nand help
nand - NAND sub-system

Usage:
nand info - show available NAND devices
nand device [dev] - show or set current device
nand read - addr off|partition size
nand write - addr off|partition size
               read/write 'size' bytes starting at offset 'off'
               to/from memory address 'addr', skipping bad blocks.
nand read[.yaffs2]] is not provide temporarily!
nand writel.yaffs2]] addr off size - write the 'size' byte yaffs image start
ing
               at offset 'off' from memory address 'addr' (.yaffs2 for 512*16 NAND)
nand erase [clean] [off size] - erase 'size' bytes from
               offset 'off' (entire device if not specified)
nand bad - show bad blocks
nand dump[.oob] off - dump page
nand scrub - really clean NAND erasing bad blocks (UNSAFE)
nand markbad off [...] - mark bad block(s) at offset (UNSAFE)
nand biterr off - make a bit error at offset (UNSAFE)
[IPAN_MINI2440]#

```

(七) 引导 Linux 内核启动

在前面我们讲了 u-boot 对 Nor Flash 和 Nand Flash 的启动支持，那现在我们就再来探讨一下 u-boot 怎样来引导 Linux 内核的启动。

1、机器码的确定

通常，在 u-boot 和 kernel 中都会有一个机器码(即：MACH_TYPE)，只有这两个机器码一致时才能引导内核，否则运行到“done.booting the kernel”就停止了，如下所示：

首先，确定 u-boot 中的 MACH_TYPE。在 u-boot 的 include/asm-arm/mach-types.h 文件中针对不同的 CPU 定义了非常多的 MACH_TYPE,可以找到下面这个定义：

```
#define MACH_TYPE_Q2440 1997
#define MACH_TYPE_QQ2440 1998
#define MACH_TYPE_MINI2440 1999 //mini2440 的机器码
#define MACH_TYPE_COLIBRI300 2000
#define MACH_TYPE_JADES 2001
```

那么我们就修改 u-boot 的 MACH_TYPE 代码引用部分，确定 u-boot 的 MACH_TYPE。如下：

```
#gedit board/samsung/mini2440/mini2440.c //修改 board_init 函数

gpio->GPHUP = 0x000007FF;

/* arch number of SMDK2410-Board */

#if defined(CONFIG_S3C2440)
    gd->bd->bi_arch_number = MACH_TYPE_MINI2440; // 为 u-boot 添加机器码
    //gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;
#endif
```

其次，确定 kernel 中的 MACH_TYPE。在 kernel 的 arch/arm/tools/mach-types 文件中针对不同的 CPU 定义了非常多的 MACH_TYPE,也可以找到下面这个定义：

f5d8231_4_v2	MACH_F5D8231_4_V2	F5D8231_4_V2	1996
q2440	MACH_Q2440	Q2440	1997
qq2440	MACH_QQ2440	QQ2440	1998
mini2440	MACH_MINI2440	MINI2440	1999//Linux 中的 MINI2440 的机器码
colibri300	MACH_COLIBRI300	COLIBRI300	2000
jades	MACH_JADES	JADES	2001

那么我们就修改 kernel 的 MACH_TYPE 代码引用部分，确定 kernel 的 MACH_TYPE 如下：

```
#gedit arch/arm/mach-s3c2440/mach-mini2440.c

MACHINE_START(MINI2440, "MINI2440")

改为：
```

```
MACHINE_START(MINI2440, "FriendlyARM MINI2440 development board")
```

现在可以编译内核，然后制作出 uImage，将 uImage 下载到开发板上，以后复制到 Nand Flash 中，不过 u-boot 还是无法引导内核，还是需要在 u-boot 中一些配置。

在 u-boot-2010.03/include/configs/mini2440.h 文件中

```
#define CONFIG_BOOTDELAY 3 //自动启动前延时 3 秒，不过这个要和下面的另一个配置一起定义后才会起作用
```

配置命令行的参数

```
#define CONFIG_BOOTARGS "noinitrd root=/dev/mtdblock3 init=/linuxrc console=ttySAC0,115200 mem=64M"
```

//只有这个宏定义了以后，上面的那个宏的定义才会生效，否则还是会直接的出现命令行的提示符，不会引导内核的。root=/dev/mtdblock3 这是由我们 Linux 中的 nand Flash 分区所决定的，我的 NandFlash 的第四个分区为根文件系统所以是 mtdblock3,如果不是的话请修改

我的 NandFlash 的情况是这样的

```
"U-boot", 0x00000000 -- 0x00040000,
"param", 0x00040000--0x00060000,
"Kernel", 0x00060000---0x00560000,
"root", 0x00560000---结束
```

还要在该文件中增加下面两个宏的定义:

```
#define CONFIG_INITRD_TAG
```

```
#define CONFIG_CMDLINE_TAG 1 //向内核传递命令行参数
```

```
#define CONFIG_SETUP_MEMORY_TAGS 1 //向内核传递内存分布信息
```

为了使 u-boot 可以自行引导 Linux 内核，我们还需要添加下面的这句话。

```
#define CONFIG_BOOTCOMMAND "nand read 0x30008000 0x60000 0x500000;bootm 0x30008000"
```

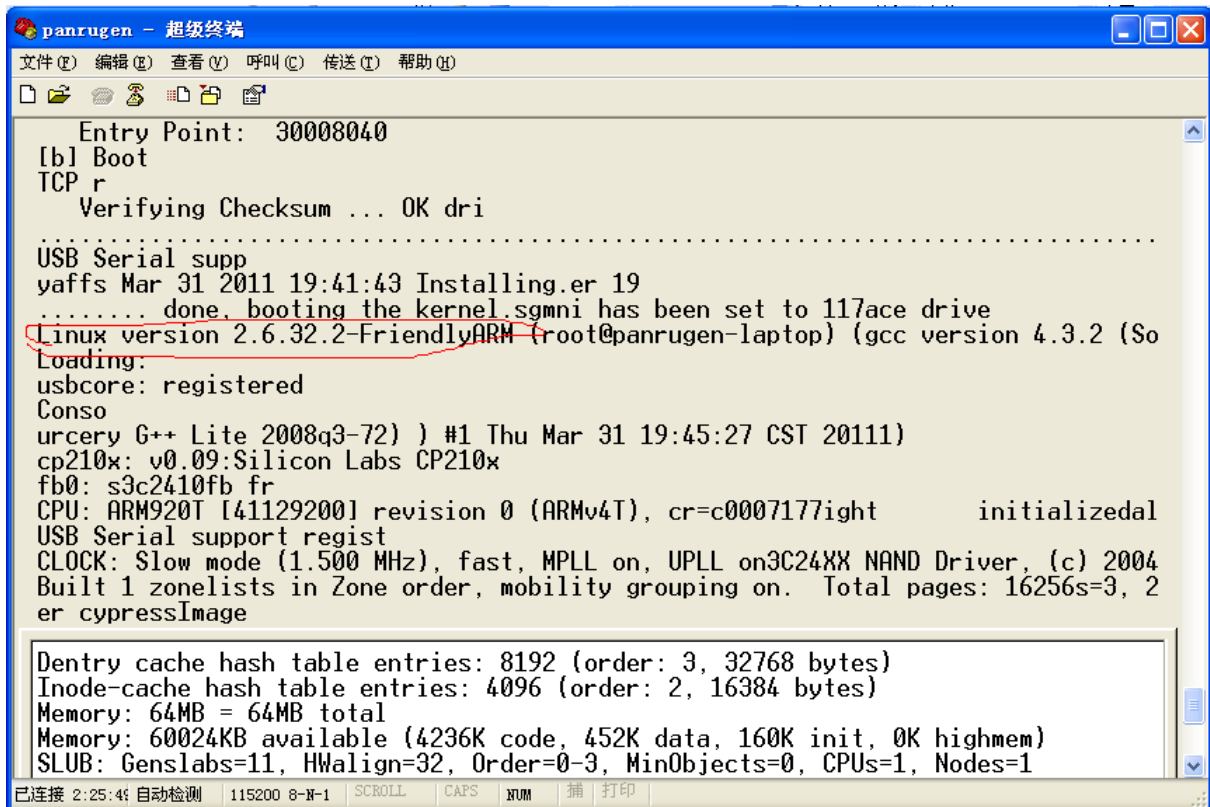
//这个宏定义的意思是将 nand 中 0x60000-0x500000(和 kernel 分区一致)的内容读到内存 0x30008000 中，然后用 bootm 命令来执行

分别重新编译 u-boot，生成 u-boot.bin，重新移植到开发板上，记得要 saveenv，然后将 uImage 用 tftp 下载到内存后使用 bootm 命令来测试引导内核，结果可以引导了，有如下命令：

```
tftp 0x30008000 uImage
```

```
bootm 0x30008000
```

如下图:



2、下载固化

在前面我们看到了 NandFlash 的分区信息。下面我们使用命令，把我们的 Linux 内核和制作的根文件系统下载到 NandFlash 中，以便以开始启动就可以引导 Linux 内核。

	起始地址	结束地址	大小空间
uboot	0x00000000	0x00040000	0x40000
param	0x00040000	0x00060000	0x20000
Kernel	0x00060000	0x00560000	0x500000
Root	0x00560000		

这是 NandFlash 的分区情况;

在 u-boot 命令行下:

输入 `#tftp 0x30008000 uImage` , 稍等即可下载 uImage 到内存中。

接着执行 `#nand erase 60000 500000` 删除掉 kernel 空间原有的数据。

执行 `#nand write 0x30008000 60000 500000` , 将内存中的 kernel 烧入 nand

flash。

接下来，输入 `#tftp 0x30008000 rootfs.image`，将根文件系统镜像下载到内存中，需设置 `rootfs.img` 为可执行文件。

再输入 `#nand erase 0x560000 $filesize` 将 root 空间内原有数据删除。

再输入 `#nand write.yaffs2 0x30008000 0x560000 $filesize`

现在可以从开发板输入命令 `boot`，从 NandFlash 启动可以看到启动的信息了。