GEC2440&U-BOOT-2009.11 移植实验

u-boot简介

u-boot是德国DENX小组的开发用于多种嵌入式CPU的bootloader程序，u-boot不仅仅支持嵌入式Linux系统的引导，当前，它还支持NetBSD, VxWorks, QNX, RTEMS, ARTOS, LynxOS嵌入式操作系统。u-boot除了支持PowerPC系列的处理器外，还能支持MIPS、x86、ARM、NIOS、XScale等诸多常用系列的处理器。

u-boot 源码目录介绍

目录	内容
board	和一些已有开发板有关的文件. 每一个开发板都以一个子目录出现在当前目录中，比如说:SMDK2410,子目录中存放与开发板相关的配置文件.
common	实现uboot命令行下支持的命令，每一条命令都对应一个文件。例如go命令对应就是cmd_boot.c
cpu	与特定CPU架构相关目录，每一款uboot下支持的CPU在该目录下对应一个子目录，比如有子目录arm920t等。
disk	对磁盘的支持
doc	文档目录。uboot有非常完善的文档，推荐大家参考阅读。
drivers	uboot支持的设备驱动程序都放在该目录，比如各种网卡、支持CFI的Flash、串口和USB等。
fs	支持的文件系统，uboot现在支持cramfs、fat、fdos、jffs2和registerfs。
include	uboot使用的头文件，还有对各种硬件平台支持的汇编文件，系统的配置文件和对文件系统支持的文件。该目录下configs目录有与开发板相关的配置头文件，如smdk2410.h。该目录下的asm目录有与CPU体系结构相关的头文件，asm对应的是asmarm.
lib_xxxx	与体系结构相关的库文件。如与ARM相关的库放在lib_arm中。
net	与网络协议栈相关的代码，BOOTP协议、TFTP协议、RARP协议和NFS文件系统的实现。
tools	uboot的工具，如：mkimage, crc等等。

u-boot 的启动过程

启动流程

我们一般把bootloader都分为阶段1(stage1)和阶段2(stage2)两大部分，依赖于CPU体系结构的代码（如CPU初始化代码等）通常都放在阶段1中且通常用汇编语言实现，而阶段2则通常用C语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

阶段1，汇编代码，对于s3c2440是cpu/arm920t/start.s文件。

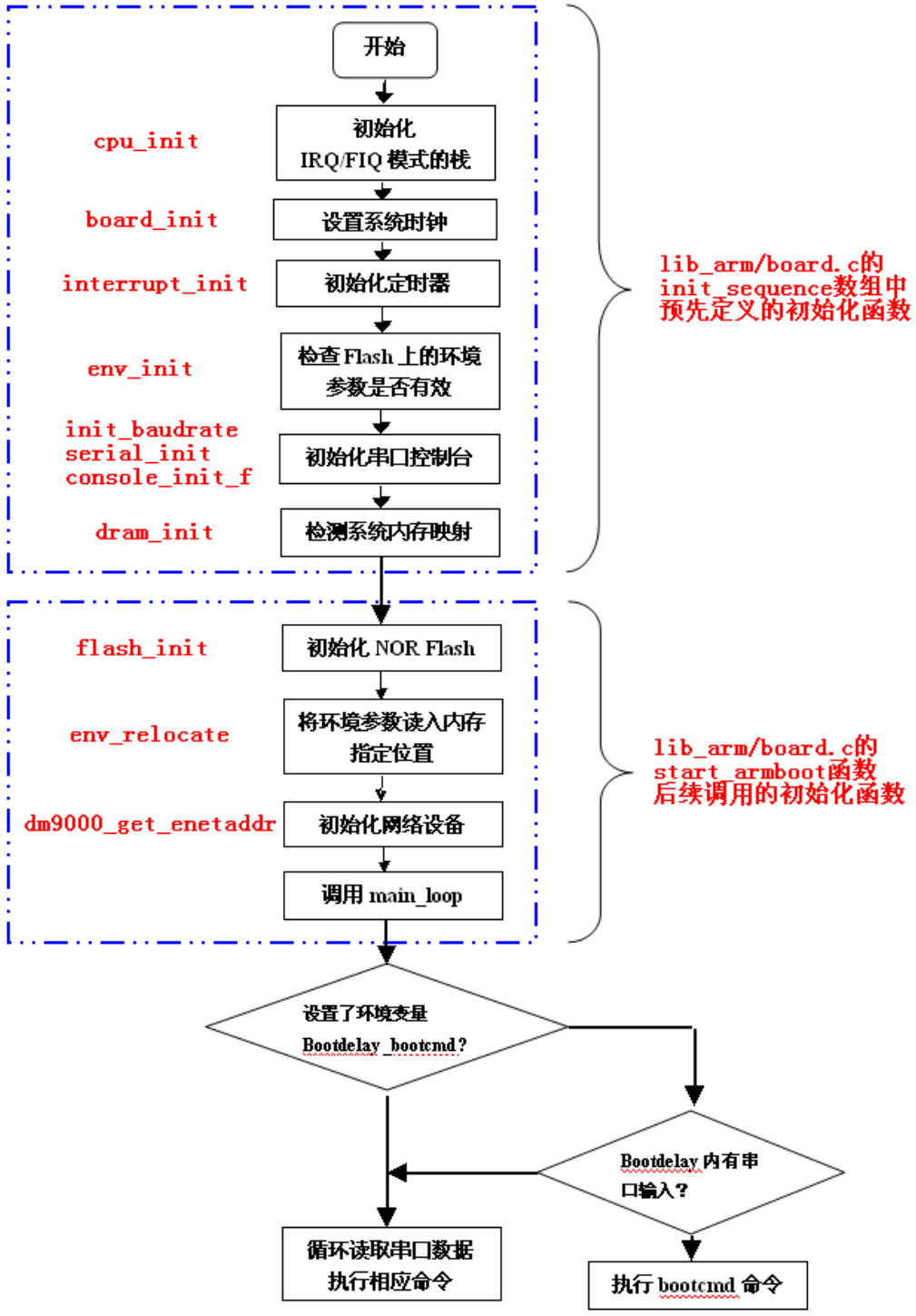
主要流程如下：

设置*CPU*的模式为*SVC*模式
关闭看门狗
禁掉所有中断
设置以*CPU*的频率
把自己拷贝到RAM

配置内存区控制寄存器
配置的栈空间
进入C代码部分

阶段2是C语言代码，在lib_arm/board.c中的start_armboot是C语言开始的函数，也是整个启动代码中C语言的主函数。这个函数调用一系列的初始化函数，然后进入主UBOOT命令行，进入命令循环（即整个boot的工作循环），接受用户从串口输入的命令，然后进行相应的工作，如下图所示。

当用户输入启动linux的命令的时候，u-boot会将 kernel 映像（zImage）和从 nand flash 上读到 RAM 空间中，为内核设置启动参数，调用内核，从而启动linux。



u-boot移植要点

我们可以总结出bootloader的两大功能：

1 是下载功能，既通过网口、串口或者USB口下载文件到RAM中。

2 是对flash芯片的读写功能。

u-boot对S3C2440已经有了很好的支持，在移植过程中主要是完善u-boot对nand flash的读写功能。

u-boot移植前的准备工作

1.下载源码

Uboot的源码可以从以下网址下载：

http://downloads.sourceforge.net/u-boot/u-boot-2009.11.tar.bz2?modtime=1134752480&big_mirror=0

我们这里下载的是u-boot-2009.11.tar.bz2

工具链使用cross-4.1.2

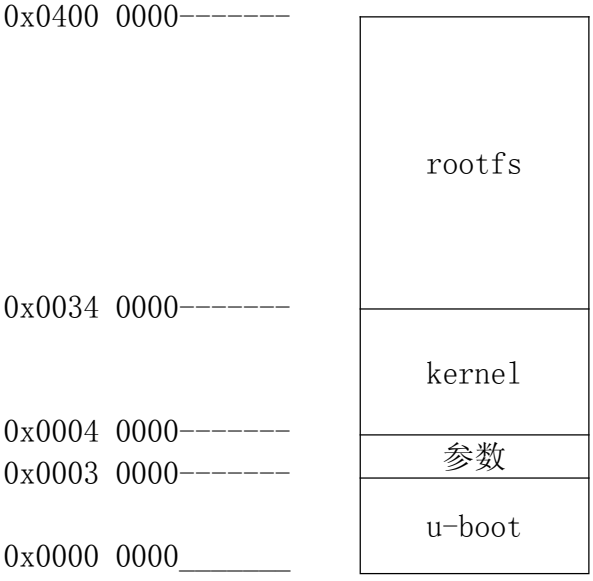
2.建立工作目录：

```
#vi mkdir /root/build_uboot
#vi cd /root/build_uboot
```

把下载的源码拷贝到该目录,解压；

```
#vi tar jxvf u-boot-2009.11.tar.bz2
#vi mv u-boot-2009.11 u-boot
```

3.确定分区：



我们可以根据以上的分区信息来配置我们的系统.

u-boot移植的具体步骤(详细记录)

1、修改顶层 Makefile(vi Makefile)

1)、创建目标板信息

在

```
smdk2410_config : unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

后面添加

```
gec2440_config : unconfig
        @$(MKCONFIG) $(@:_config=) arm arm920t gec2440 samsung s3c24x0
```

各项的意思如下:

arm: CPU的架构(ARCH)

arm920t: CPU的类型(CPU)，其对应于cpu/arm920t子目录。

gec2440: 开发板的型号(BOARD)，对应于board/gec2440目录。

samsung: 开发者/或经销商(vender)。

s3c24x0: 片上系统(SOC)。

板子起名叫gec2440，可以依自己的喜好修改

2)、修改 Makefile 规则

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

2、建立主代码与测试

1)、复制 board/samsung/smdk2410 目录为 board/samsung/gec2440：

```
#cp  –arf  board/samsung/smdk2410/  board/samsung/gec2440/
```

并修改该目录下的 Makefile(vi board/samsung/gec2440/Makefile)：

```
COBJS :=smdk2410.o flash.o
    修改为：
COBJS := gec2440.o flash.o
```

并将复制后目录下的 smdk2410.c 改名为 gec2440.c：

```
#mv  board/samsung/gec2440/smdk2410.c  board/samsung/gec2440/gec2440.c
```

2)、复制 include/configs/smdk2410.h 为 include/configs/gec2440.h

```
#cp  include/configs/smdk2410.h  include/configs/gec2440.h
```

3)、测试是否能配置和编译成功

```
#make gec2440_config
#make
```

编译完成时最后两句如下：

```
arm-linux-objcopy -O srec u-boot u-boot.srec
arm-linux-objcopy --gap-fill=0xff -O binary u-boot u-boot.bin
```

表示编译成功。

3、初始化 stage1 阶段的硬件设备

1）、修改 cpu/arm920t/start.S：

在 start_code 函数中：

修改

```
    bl coloured_LED_init
    bl red_LED_on
```

如下

```
    //这两行是 AT91RM9200DK 开发板的 LED 初始化和控制函数，注释
    //bl coloured_LED_init
    //bl red_LED_on
```

添加下面的代码，以取代上面功能：

```
    #if defined(CONFIG_S3C2440)         //区别与其他开发板
    #define GPBCON    0x56000010
    #define GPBDAT    0x56000014
    #define GPBUP     0x56000018
                ldr         r0,=GPBUP
                ldr         r1,=0xff
                str         r1,[r0]
                ldr         r0,=GPBCON
                ldr         r1,=0x557ff
                str         r1,[r0]

                ldr         r0,=GPBDAT
                ldr         r1,=0x7df
                str         r1,[r0]
    #endif
```

2）、在 include/configs/gec2440.h 头文件中添加 CONFIG_S3C2440 宏（红色一行）

```
    #define CONFIG_ARM920T       1    /* This is an ARM920T Core     */
    #define CONFIG_S3C2410       1    /* in a SAMSUNG S3C2410 SoC    */
    #define CONFIG_SMDK2410      1    /* on a SAMSUNG SMDK2410 Board */
    #define CONFIG_S3C2440       1    /* in a SAMSUNG S3C2440 SoC    */
```

4、修改时钟

因为 S3C2410 与 S3C2440 的时钟及 Nand、SDRAM 等配置不同，故接下来需要进行修改相关配置。

1）、修改 cpu/arm920t/start.S：

```
    #if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410) || defined(CONFIG_S3C2440)
    /* turn off the watchdog */
    # if defined(CONFIG_S3C2400)
    # define pWTCON     0x15300000
    # define INTMSK     0x14400008    /* Interupt-Controller base addresses */
    # define CLKDIVN    0x14800014    /* clock divisor register */
    #else  //下面 2410 和 2440 的寄存器地址是一致的
    # define pWTCON     0x53000000
```

```
# define INTMSK     0x4A000008    /* Interupt-Controller base addresses */
# define INTSUBMSK  0x4A00001C
# define CLKDIVN    0x4C000014    /* clock divisor register */
# endif
    ldr  r0, =pWTCON
    mov  r1, #0x0
    str  r1, [r0]
    /*
     * 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     //0x3ff，根据 2410 芯片手册，INTSUBMSK 有 11 位可用，vivi 也是 0x7ff，
u-boot 则是 0x3ff，不过芯片复位后所有中断都被屏蔽，故这无影响
    ldr  r0, =INTSUBMSK
    str  r1, [r0]
# endif
# if defined(CONFIG_S3C2440) //添加 s3c2440 的中断禁止部分
    ldr  r1, =0x7fff         //根据 2440 芯片手册，INTSUBMSK 寄存器有 15 位可用
    ldr  r0, =INTSUBMSK
    str  r1, [r0]
# endif

# if defined(CONFIG_S3C2440)    //添加 s3c2440 的时钟部分
#define MPLLCON   0x4C000004    //系统主频配置寄存器基地址
#define UPLLCON   0x4C000008    //USB 时钟频率配置寄存器基地址
    ldr  r0, =CLKDIVN          //设置分频系数 FCLK:HCLK:PCLK = 1:4:8
    mov  r1, #5
    str  r1, [r0]
    ldr  r0, =MPLLCON  //设置系统主频为 405MHz
    ldr  r1, =0x7F021  //这个值参考芯片手册“PLL VALUE SELECTION TABLE”部分
    str  r1, [r0]
    ldr  r0, =UPLLCON  //设置 USB 时钟频率为 48MHz
    ldr  r1, =0x38022  //这个值参考芯片手册“PLL VALUE SELECTION TABLE”部分
    str  r1, [r0]
# else //其他开发板的时钟部分，这里就不用管了，我们现在是做 2440 的
    /* FCLK:HCLK:PCLK = 1:2:4 */
    /* default FCLK is 120 MHz ! */
    ldr  r0, =CLKDIVN
    mov  r1, #3
    str  r1, [r0]
```

```
# endif
#endif    /* CONFIG_S3C2400 || CONFIG_S3C2410 || CONFIG_S3C2440 */
```

S3C2440 的时钟部分除了在 cpu/arm920t/start.S 中添加外，还要分别在 board/samsung/gec2440/gec2440.c 和 cpu/arm920t/s3c24x0/speed.c 中修改或添加部分代码，如下：

2)、修改 board/samsung/gec2440/gec2440.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“PLLVALUE SELECTION TABLE”部分进行设
#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
```

3)、修改 cpu/arm920t/s3c24x0/speed.c

//根据设置的分频系数 FCLK:HCLK:PCLK = 1:4:8 修改获取时钟频率的函数

```
static ulong get_PLLCLK(int pllreg){
     S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
     ulong r, m, p, s;
     if (pllreg == MPLL)
          r = clk_power->MPLLCON;
```

```
        else if (pllreg == UPLL)
            r = clk_power->UPLLCON;
        else
            hang();
        m = ((r & 0xFF000) >> 12) + 8;
        p = ((r & 0x003F0) >> 4) + 2;
        s = r & 0x3;
    #if defined(CONFIG_S3C2440)
        if(pllreg == MPLL)
        {   //参考 S3C2440 芯片手册上的公式：PLL=(2 * m * Fin)/(p * 2^s)
            return((CONFIG_SYS_CLK_FREQ * m * 2) / (p << s));
        }
    #endif
    return((CONFIG_SYS_CLK_FREQ * m) / (p << s));
}
    /* return HCLK frequency */
    ulong get_HCLK(void)
    {
        S3C24X0_CLOCK_POWER * const clk_power = S3C24X0_GetBase_CLOCK_POWER();
    #if defined(CONFIG_S3C2440)
        return(get_FCLK()/4);
    #endif
        return((clk_power->CLKDIVN & 0x2) ? get_FCLK()/2 : get_FCLK());
    }
```

5、下载 SDRAM 测试(利用 GEC2440 启动时的 bootloader 把 u-boot.bin 下载到 RAM 中运行测试)

1)、屏蔽 u-boot 中再次对 CPU、RAM 的初始化

```
#vi cpu/arm920t/start.S：
```

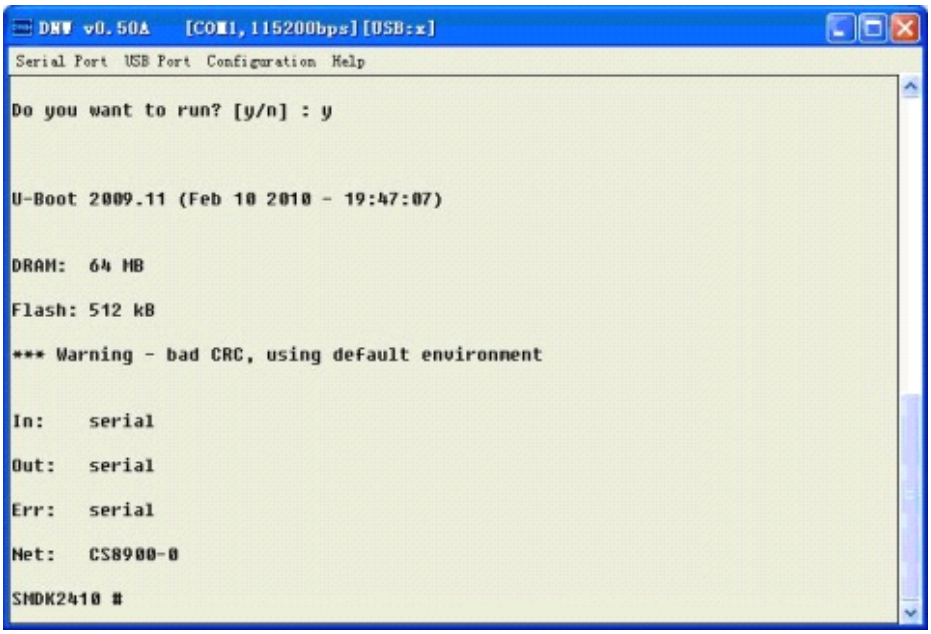
修改如下

```
#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
```

2）、编译

```
#make gec2440_config
#make
```

下载到 SDRAM（BANK6）中运行结果是：板上的 LED1 灯亮，其它全灭；同时，终端有输出信息并且出现类似 Shell 的命令行，这说明这一部分移植完成。示意图如下：



3)、修改支持 Nor Flash 启动及自定义命令符

通常，在嵌入式 bootloader 中，有两种方式来引导启动内核：从 Nor Flash 或 Nand Flash 启动。u-boot 默认从 Nor Flash 启动，从 6 中运行结果中发现几个问题：

第一， 开发板的 Nor Flash 是 1M 的，而显示的是 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自己的 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 AMD AM29LV800DB。另外怎样将命令行提示符前面的 SMDK2410 变成我自己定义的呢？下面一一解决这些问题，让 u-boot 完全对 Nor Flash 的支持。

```
#vi  include/configs/gec2440.h
```

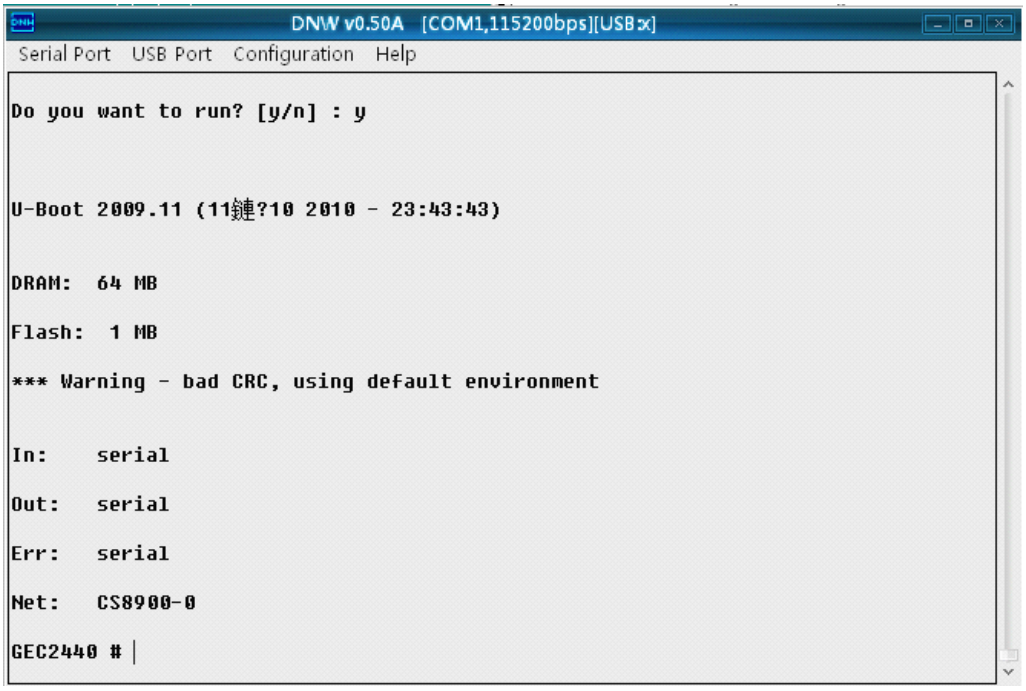
修改如下

```
#define CONFIG_SYS_PROMPT   "GEC2440 #" //"SMDK2410 # "
/* Monitor Command Prompt */
```

Nor Flash 是 Am29LV800B，为 1MB

```
//#define CONFIG_AMD_LV400      1
//#if 0
#define CONFIG_AMD_LV800        1
//#endif
```

下载到 SDRAM 运行结果如下：



现在运行显示 Flash 容量正确了。

6、添加 Nand Flash 支持（stage1：将 flash 上代码定位到 sdram 中）

目前 u-boot 还没对 2440 中的 Nand Flash 支持，也就是说要想 u-boot 从 Nand Flash 上启动得自己去实现了。

1)、在 include/configs/gec2440.h 头文件中定义 Nand 要用到的宏和寄存器，如下：

//在文件末尾加入以下 Nand Flash 相关定义（在最后一句#endif /* __CONFIG_H */之前）：

```
/*
 * Nand flash register and envionment variables
 */
#define CONFIG_S3C2440_NAND_BOOT  1
#define NAND_CTL_BASE  0x4E000000  //Nand Flash 配置寄存器基地址，查 2440 手册可得知
#define STACK_BASE  0x33F00000     //定义堆栈的地址
#define STACK_SIZE  0x8000         //堆栈的长度大小
#define oNFCONF  0x00 //相对 Nand 配置寄存器基地址的偏移量，还是配置寄存器的基地址
#define oNFCONT  0x04 //相对 Nand 配置寄存器基地址的偏移量，可得到控制寄存器的基地址(0x4E000004)
#define oNFADDR  0x0c //相对 Nand 配置寄存器基地址的偏移量，可得到地址寄存器的基地址 0x4E00000c
#define oNFDATA  0x10 //相对 Nand 配置寄存器基地址的偏移量，可得到数据寄存器的基地址(0x4E000010
#define oNFCMD   0x08 //相对 Nand 配置寄存器基地址的偏移量，可得到指令寄存器的基地址(0x4E000008
#define oNFSTAT  0x20 //相对 Nand 配置寄存器基地址的偏移量，可得到状态寄存器的基地址(0x4E000020)
#define oNFECC   0x2c //相对 Nand 配置寄存器基地址的偏移量，可得到 ECC 寄存器的基地址(0x4E00002c)
```

接下来，修改 cpu/arm920t/start.S 这个文件，使 u-boot 从 Nand Flash 启动，前面提过，u-boot 默认是从 Nor Flash 启动的。修改部分如下：

2）、修改 cpu/arm920t/start.S

//注意：在上面 Nor Flash 启动中，我们为了把 u-boot 用 bootloader 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，把其注释去掉。

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif

#if 0 //屏蔽掉 u-boot 中的从 Nor Flash 启动部分
#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 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
#endif /* CONFIG_SKIP_RELOCATE_UBOOT */
#endif

//下面添加 2440 中 u-boot 从 Nand Flash 启动
#ifdef CONFIG_S3C2440_NAND_BOOT
    mov r1, #NAND_CTL_BASE   //复位 Nand Flash
    ldr r2, =( (7<<12)|(7<<8)|(7<<4)|(0<<0) )
    str r2, [r1, #oNFCONF]   //设置配置寄存器的初始值，参考 s3c2440 手册
    ldr r2, [r1, #oNFCONF]
    ldr r2, =( (1<<4)|(0<<1)|(1<<0) )
    str r2, [r1, #oNFCONT]   //设置控制寄存器
    ldr r2, [r1, #oNFCONT]
    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, #oNFCONT]
    orr r2, r2, #0x2          //取消片选
    str r2, [r1, #oNFCONT]
    //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, #0x30000  //传递给 C 代码的第三个参数：u-boot 的长度大小(128k)
    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:
    //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 //CONFIG_S3C2440_NAND_BOOT
_start_armboot: .word start_armboot //在这一句的下面加上 DW_STACK_START 的定义
.align 2
DW_STACK_START: .word STACK_BASE+STACK_SIZE-4
```

3）、在 board/samsung/gec2440/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：

```
#include <config.h>
#define NF_BASE   0x4E000000  //Nand Flash 配置寄存器基地址
#define __REGb(x) (*(volatile unsigned char *)(x))
#define __REGi(x) (*(volatile unsigned int  *)(x))
#define NFCONF __REGi(NF_BASE + 0x0 )  //通过偏移量还是得到配置寄存器基地址
```

```
#define NFCONT  __REGi(NF_BASE + 0x4 )  //通过偏移量得到控制寄存器基地址
#define NFCMD   __REGb(NF_BASE + 0x8 )  //通过偏移量得到指令寄存器基地址
#define NFADDR  __REGb(NF_BASE + 0xC )  //通过偏移量得到地址寄存器基地址
#define NFDATA  __REGb(NF_BASE + 0x10)  //通过偏移量得到数据寄存器基地址
#define NFSTAT  __REGb(NF_BASE + 0x20)  //通过偏移量得到状态寄存器基地址
#define NAND_CHIP_ENABLE  (NFCONT &= ~(1<<1))  //Nand 片选使能
#define NAND_CHIP_DISABLE (NFCONT |= (1<<1))   //取消 Nand 片选
#define NAND_CLEAR_RB     (NFSTAT |= (1<<2))
#define NAND_DETECT_RB    { while(! (NFSTAT&(1<<2)) );}
#define NAND_SECTOR_SIZE 512
#define NAND_BLOCK_MASK (NAND_SECTOR_SIZE - 1)
/* low level nand read function */
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; //地址或长度不对齐
    }
    NAND_CHIP_ENABLE; //选中 Nand 片选
    for(i=start_addr; i < (start_addr + size);)
    {
        //发出 READ0 指令
        NAND_CLEAR_RB;
        NFCMD = 0;
        //对 Nand 进行寻址
        NFADDR = i & 0xFF;
        NFADDR = (i >> 9) & 0xFF;
        NFADDR = (i >> 17) & 0xFF;
        NFADDR = (i >> 25) & 0xFF;
        NAND_DETECT_RB;
        for(j=0; j < NAND_SECTOR_SIZE; j++, i++)
        {
            *buf = (NFDATA & 0xFF);
            buf++;
        }
    }
    NAND_CHIP_DISABLE; //取消片选信号
    return 0;
}
```

4)、修改 board/samsung/gec2440/Makefile

在 board/samsung/gec2440/Makefile 中添加 nand_read.c 的编译选项，使他编译到 u-boot 中，如下：

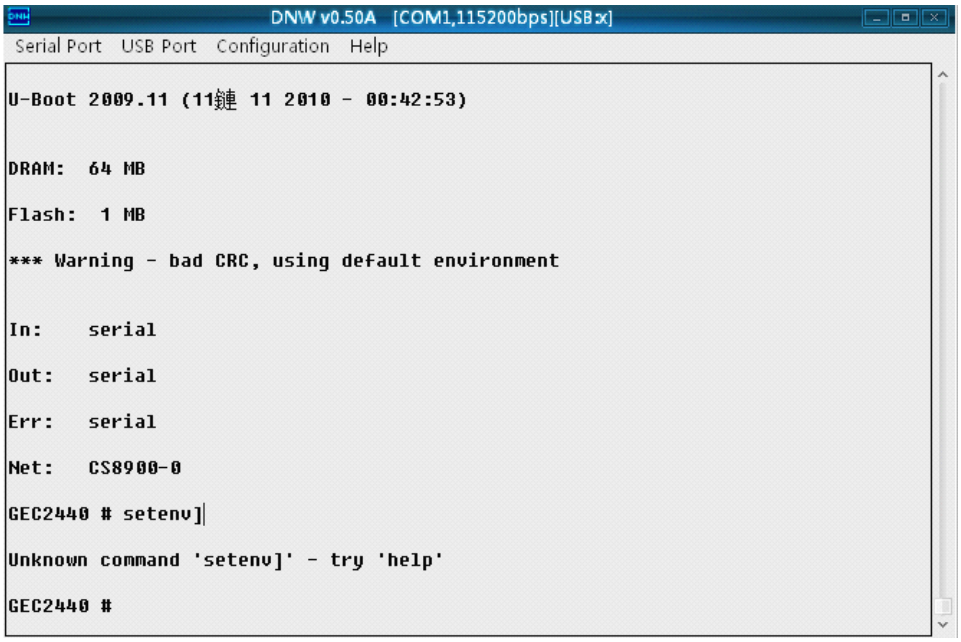
```
COBJS   := gec2440.o flash.o nand_read.o
```

5)、修改 cpu/arm920t/u-boot.lds

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/gec2440/lowlevel_init.o (.text)
    board/samsung/gec2440/nand_read.o (.text)
    *(.text)
}
```

最后编译 u-boot，生成 u-boot.bin 文件，并将其烧到 Nand Flash 中，启动结果如下：



从上面的运行图看，显然现在的 Nand 还不能做任何事情（环境变量也不能保存），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(K9F1208U0C)的 Stage2 部分的有关操作支持。

7、添加 Nand Flash 支持（stage2：添加对 flash 读写操作支持）

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（也是在此目录下）文件，在这里面来实现对 nand 的操作，代码如下：

1）、新建 drivers/mtd/nand/s3c2440_nand.c

```
#include <common.h>
#if 0
#define DEBUGN    printf
#else
```

```
#define DEBUGN(x, args ...) {}
#endif
#include <nand.h>
#include <s3c2410.h>
#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 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 S3C2440_NFCONT_nCE  (1<<1)
#define S3C2440_ADDR_NALE   0x0c
#define S3C2440_ADDR_NCLE   0x08
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_NALE;
        if (!(ctrl & NAND_ALE))                 //要写的是命令
        IO_ADDR_W |= S3C2440_ADDR_NCLE;
        if (ctrl & NAND_NCE)
            NFCONT &= ~S3C2440_NFCONT_nCE;     //使能 nand flash
        else
            NFCONT |= S3C2440_NFCONT_nCE;      //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 tacls, twrph0, twrph1;
        struct s3c24x0_clock_power * const clk_power = s3c24x0_get_base_clock_power();
        DEBUGN("board_nand_init()\n");
        clk_power->CLKCON |= (1 << 4);
        twrph0 = 4; twrph1 = 2; tacls = 0;
        cfg = (tacls<<12)|(twrph0<<8)|(twrph1<<4);
        NFCONF = cfg;
        cfg = (1<<6)|(1<<4)|(0<<1)|(1<<0);
        NFCONT = cfg;
        /* initialize nand_chip data structure */
        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;
        return 0;
    }
```

2)、修改 include/configs/gec2440.h

接下来，在开发板配置文件 include/configs/gec2440.h 文件中定义支持 Nand 操作的相关宏，如下：

```
    /* Command line configuration. */
    #define CONFIG_CMD_NAND
    #define CONFIG_CMDLINE_EDITING
    #ifdef CONFIG_CMDLINE_EDITING
    #undef CONFIG_AUTO_COMPLETE
    #else
    #define CONFIG_AUTO_COMPLETE
    #endif
    /* NAND flash settings */
    #if defined(CONFIG_CMD_NAND)
    #define CONFIG_SYS_NAND_BASE            0x4E000000 //Nand 配置寄存器基地址
    #define CONFIG_SYS_MAX_NAND_DEVICE      1
```

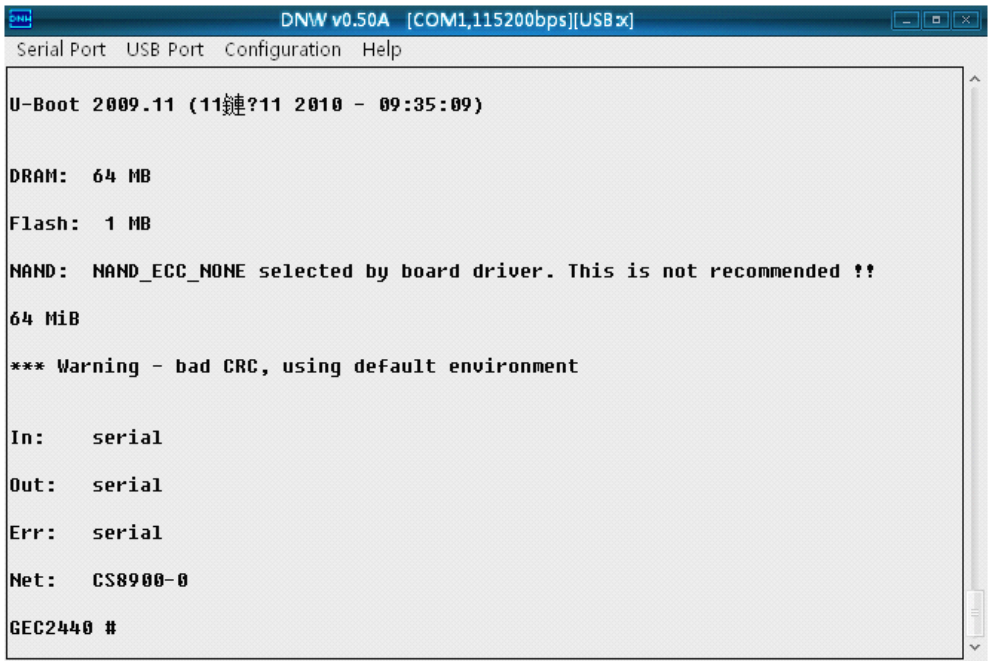
```
    #define CONFIG_MTD_NAND_VERIFY_WRITE    1
    //#define NAND_SAMSUNG_LP_OPTIONS       1 //注意：我们这里是 64M 的 Nand Flash，所以不用，如果是 128M 的大块 Nand Flash，则需加上
    #endif
```

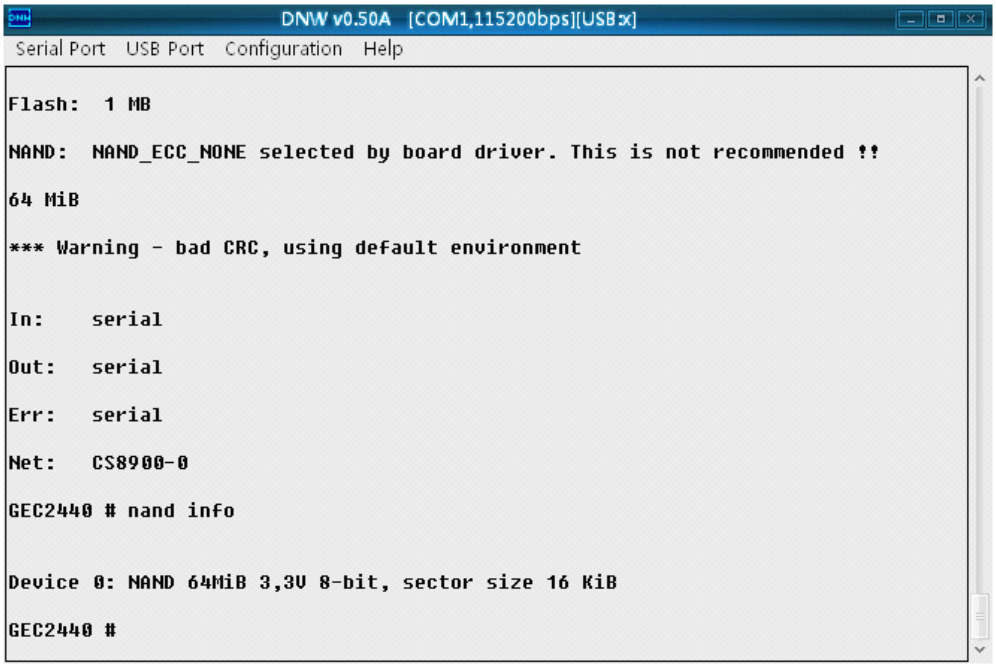
3）、修改 drivers/mtd/nand/Makefile

接下来，在 drivers/mtd/nand/Makefile 文件中添加 s3c2440_nand.c 的编译项，如下：

```
    COBJS-y += s3c2440_nand.o
    COBJS-$(CONFIG_NAND_S3C2440) += s3c2440_nand.o
```

4）、重新编译 u-boot，把生成的 u-boot.bin 烧写到 Nand Flash 中，并从 Nand Flash 启动，结果如下：





从上图可以看出，现在 u-boot 已经对我们开发板上 64M 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。

5)、更改环境变量存储路径

从以上的启动信息看，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，如下：

```
GEC2440 # saveenv

Saving Environment to Flash...

Un-Protected 1 sectors

Erasing Flash...Erasing sector 18 ... Erased 1 sectors

GEC2440 #
```

从上图可以看到保存环境变量并没有成功，而且从信息看他将把环境变量保存到 Flash 中，显然这不正确，我们是要保存到 Nand 中。原来，u-boot 在默认的情况下把环境变量都是保存到 Nor Flash 中的，所以我们要修改代码，让他保存到 Nand 中，如下：

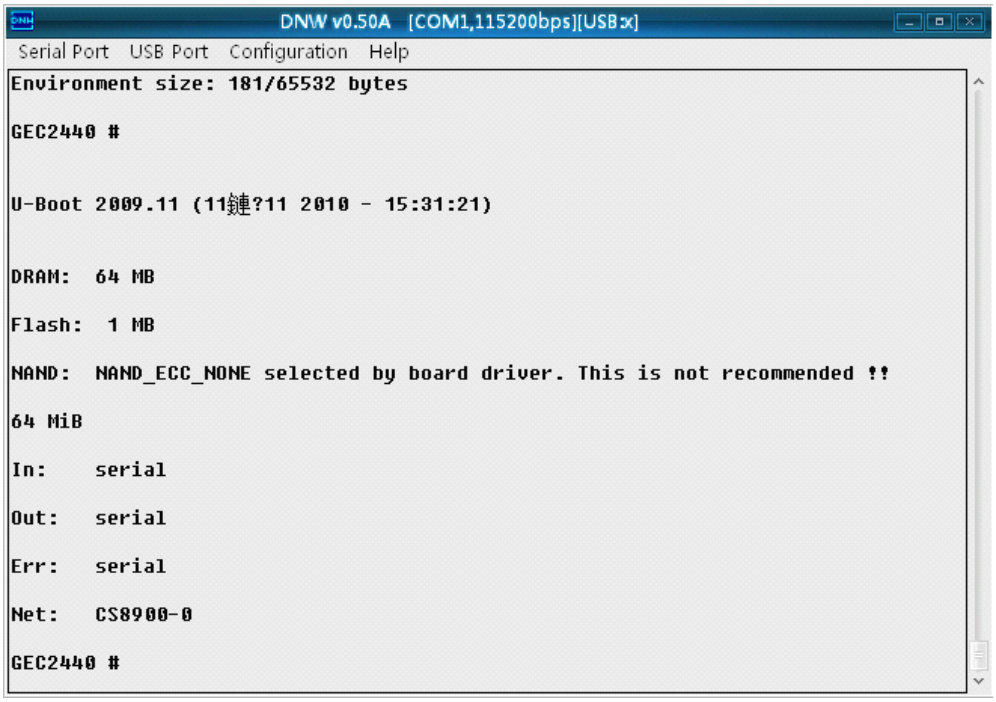
修改 include/configs/gec2440.h：

```
    //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      0x30000 //将环境变量保存到 nand 中的 0x30000 位置
    #define CONFIG_ENV_SIZE        0x10000 /* 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：

```
GEC2440 # saveenv

Saving Environment to NAND...

Erasing Nand...


Erasing at 0x0 -- 196608te.
Erasing at 0x0 -- 212992te.
Erasing at 0x0 -- 229376te.
Erasing at 0x0 -- 245760 complete.

Writing to Nand... done

GEC2440 #
```

重启后，警告信息没有了，如下：



8、支持网络

u-boot-2009.11 版本已经对 CS8900 和 DM9000X 网卡有比较完善的代码支持（代码在 drivers/net/目录下），但由于在 S3C24XX 系列中默认对 CS8900 网卡进行配置使用，而开发板的网卡是 DM9000，因此需要做少量修改，U-Boot-2009.11 里面没有开启 PING 命令，也没有初始化网卡的 MAC 地址，故需要修改以下内容：

1）、删除默认配置文件 include/autoconf.mk (如果前面没有编译过，则此步可略)

2）、修改 include/configs/gec2440.h：

```
添加：
#define CONFIG_CMD_PING
#define CONFIG_CMD_NET

注销 CS8900 的默认配置
//#define CONFIG_CS8900 1                    /* we have a CS8900 on-board */
//#define CONFIG_CS8900_BASE      0x19000300
//#define CONFIG_CS8900_BUS16     1          /* the Linux driver does accesses as shorts */……

添加 DM9000 的配置
 #define CONFIG_DRIVER_DM9000 1
#define CONFIG_DM9000_BASE 0x10000000        //片选 nCS2  地址 0x10000000
#define DM9000_IO CONFIG_DM9000_BASE
#define DM9000_DATA (CONFIG_DM9000_BASE+0x1000000)   //地址线 ADDR24  偏移 0x1000000
#define CONFIG_DM9000_USE_16BIT
```

3）、修改 board/samsung/gec2440/gec2440.c

将下面程序

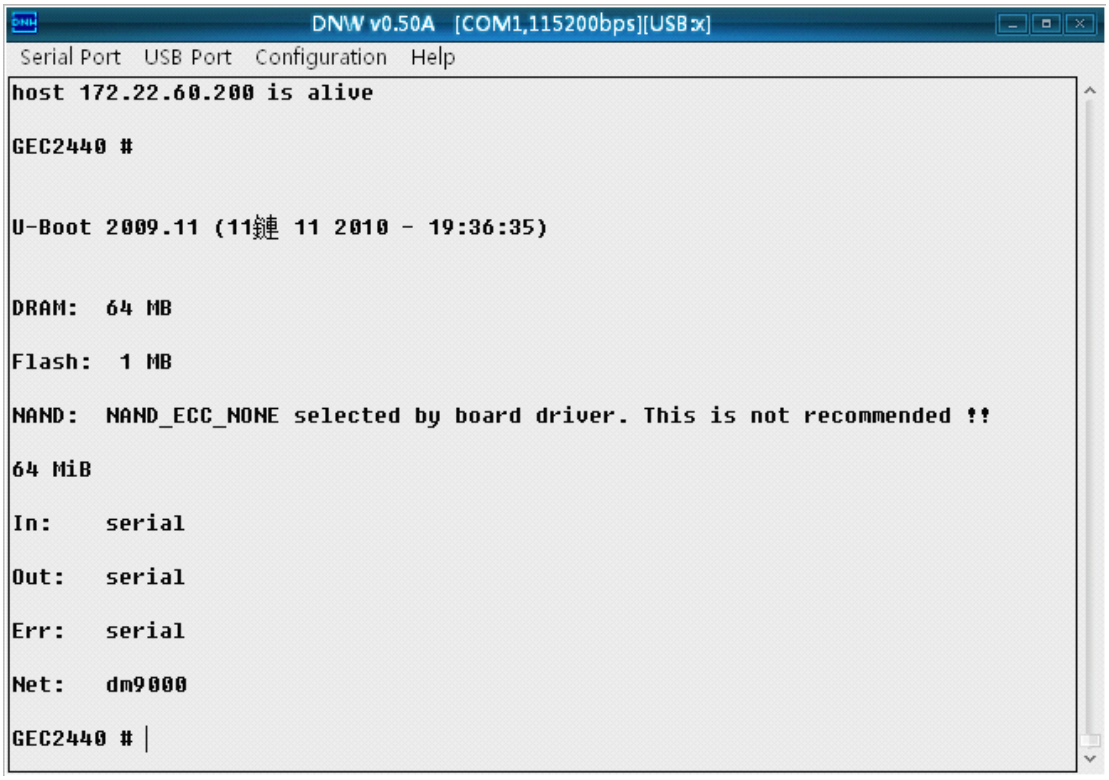
```
#ifdef CONFIG_CMD_NET
int board_eth_init(bd_t *bis)
```

```
{
        int rc = 0;
#ifdef CONFIG_CS8900
        rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
#endif
        return rc;
}
#endif
```

改为：

```
#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
#ifdef CONFIG_DRIVER_DM9000
     rc = dm9000_initialize(bis);
#endif
    return rc;
}
#endif
```

运行结果如下：

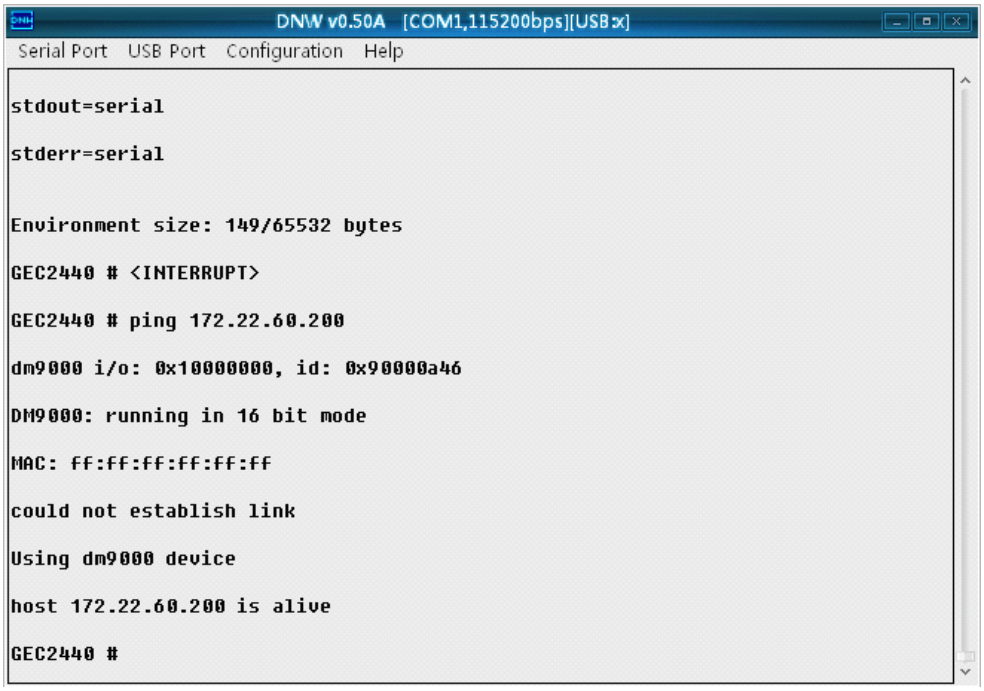


9、U-boot 命令应用

1)、IP 设置

```
GEC2440 # setenv   ipaddr   172.22.60.100      （开发板）
GEC2440 # setenv   serverip 172.22.60.200      （主机）
GEC2440 # saveenv
GEC2440 # printenv                              (查看变量内容)
GEC2440 # ping   172.22.60.200
```

可见 ping 通如下所示：



可能出现的错误

```
GEC2440 # ping 172.22.60.200

dm9000 i/o: 0x10000000, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 00:00:00:00:00:00

could not establish link

*** ERROR: `ethaddr' not set

dm9000 i/o: 0x10000000, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 00:00:00:00:00:00

could not establish link

ping failed; host 172.22.60.200 is not alive
```

网络物理地址没有设置，可采用下列方法修改

```
GEC2440 # setenv  ethaddr  12:23:34:45:56:67
GEC2440 # saveenv
```

2)、tftp 下载与烧写

U-boot：

```
GEC2440 # tftp  0x30008000  u-boot.bin
GEC2440 # nand  erase  0x0  0x40000
GEC2440 # nand  write  0x30008000  0x0  0x40000
```

Kernel：

```
GEC2440 # tftp  0x30008000  zImage
GEC2440 # nand  erase  0x40000  0x300000
GEC2440 # nand  write  0x30008000  0x40000  0x300000
```

Jffs2：

```
GEC2440 # tftp  0x30008000  rootfs.jffs2
GEC2440 # nand  erase  0x340000  0x3cc0000
GEC2440 # nand  write.jffs2  0x30008000  0x340000  0x1e00000
// GEC2440 # nand  write.jffs2  0x30008000  0x340000  xxxx（根据文件系统的大小）
```

3)、传参

```
GEC2440 # setenv bootargs root=/dev/mtdblock2 init=/linuxrc console=ttySAC0,115200
          rootfstype=jffs2 rw
GEC2440 # saveenv
```

4)、引导系统

```
GEC2440 # setenv  bootcmd  nand  read  0x30008000  0x40000 0x300000 \;go  0x30008000
GEC2440 # saveenv
GEC2440 # boot
```

10、重新编译时可能出现问题

错误现象 1

```
arm-linux-ld -g -Ttext 0xc100000 \
      -o hello_world -e hello_world hello_world.o libstubs.a \
      -L/usr/local/arm/4.1.2/bin/../lib/gcc/arm-angstrom-linux-gnueabi/4.1.2 -lgcc
arm-linux-objcopy -O srec hello_world hello_world.srec 2>/dev/null
make[1]: *** [hello_world.srec] 错误 127
make[1]: Leaving directory `/root/build_uboot/u-boot-2009.11/examples/standalone'
make: *** [examples/standalone] 错误 2
```

解决方法

进到 u-boot 目录

```
[root@localhost u-boot-2009.11]#vi Makefile
```

修改

```
142 SUBDIRS = tools \
143 #         examples/standalone \
144 #         examples/api
```

重新编译

错误现象 2

```
/bin/sh: arm-linux-objdump: command not found
arm-linux-objcopy -O srec u-boot u-boot.srec
make: arm-linux-objcopy：命令未找到
make: *** [u-boot.srec] 错误 127
```

解决方法

进去工具链目录

```
[root@localhost bin]# ln -s arm-angstrom-linux-gnueabi-objcopy arm-linux-objcopy
[root@localhost bin]# ln -s arm-angstrom-linux-gnueabi-objdump arm-linux-objdump
```

附录1

printenv　　打印环境变量

setenv　　设置环境变量

```
如：  setenv ipaddr *.*.*.*    （开发板ip）
      Setenv serverip *.*.*.*  （服务端即PC端ip）
```

saveenv　保存设定的环境变量，每次修改完环境变量后需要保存数据

我们经常要设置的环境变量有ipaddr,serverip，bootcmd，bootargs。

Tftp　即将内核镜像文件从PC中下载到SDRAM的指定地址，然后通过bootm来引导内核，前提是所用PC要安装设置tftp服务。

```
如: tftp 30008000 zImage
```

nand erase　擦除nand flash中数据块

```
如：nand erase 0x40000     0x300000
               起始地址    擦写大小
```

nand write　把RAM中的数据写到Nand Flash中，如果NandFlash 相应的区域有坏块，则会有错误提示。

```
如：nand write 0x30008000 0x40000    0x300000
               内存地址   起始地址   擦写大小
```

nand read　从nand flash中读取数据到RAM，如果NandFlash 相应的区域有坏块，则会有错误提示。

```
如：nand read 0x30008000 0x40000 0x300000
注释：将flash中 起始地址为0x40000 大小为0x300000 的数据拷贝到内存起始地址为0x30008000
      的地址段上，然后去读取
```

go　直接跳转到可执行文件的入口地址，执行可执行文件。

```
如:  go 30008000
```

boot　启动uboot引导

nand write.jffs2：向Nand Flash 写入数据（烧写jffs2文件系统使用），如果NandFlash 相应的区域有坏块，可以跳过坏块。

nand read.jffs2s：读取Nand Flash 相应区域的数据（烧写jffs2文件系统使用），如果NandFlash 相应的区域有坏块，将对应坏块区域的缓冲填充0xff，然后跳过此坏块继续读取。

nand read.jffs2：　读取Nand Flash 相应区域的数据（烧写jffs2文件系统使用），如果NandFlash 相应的区域有坏块，直接跳过坏块。

附录 2

Linux系统下配置tftp服务器（虚拟机网络连接方式：桥连）

A：在服务器端（PC）设置(设置前确保已安装TFTP服务器软件)

```
#vi /etc/xinetd.d/tftp
```

```
service tftp
{
disable = no        //默认为yes，配置为no，开启tftp服务器
socket_type = dgram
protocol = udp
wait = yes
user = root
server = /usr/sbin/in.tftpd
server_args = -s /test   //  “test”为tftp服务器的目录，pc上必须存在该目录
per_source = 11
cps = 100 2
flags = IPv4
}
```

B、启动tftp服务

```
#service xinetd restart
```

重启xinetd服务，因为TFTP服务受控于xinetd, xinetd是管服务的服务

C、配置PC与开发板U-boot的ip信息，同一网段。

```
虚拟机：      ifconfig eth0 192.168.1.111    （注意不要与PC上的windows冲突）
U-boot :      setenv  ipaddr 192.168.1.100
              setenv  serverip 192.168.1.111
              saveenv
```

D、开发板上电然后通过网络从虚拟机中的linux下载文件。

```
GEC2440 # tftp 30008000 zImage
......
```

Windows下使用tftp32下载工具

A、 打开tftp32.exe
B、 配置tftp服务器的ip为192.168.1.222（注意与虚拟机的tftp服务网址不要冲突）
C、 配置提供下载服务的目录
D、 开发板上电进入Bootloader引导界面，使用tftp下载即可

操作步骤

1 用网线连接开发板和PC机
2 启动U-BOOT并设置环境变量

```
    setenv  ipaddr 192.168.1.100  //设置开发板的IP
    setenv  serverip 192.168.1.222  //设置PC机windows的IP
    setenv ethaddr 11.22.33.44.55.66 //设置开发板的物理地址
    saveenv     //保存
```

3 PC机端打开TFTP服务器，并且把要下载的文件拷贝到tftp服务器程序所在的目录下

4 下载和烧写

在u-boot下用以下命令

```
tftp 30008000 zImage
nand erase 40000 300000
nand write 30008000 40000 300000
……
```

设置开机自启动内核挂载功能：

```
setenv bootcmd nand read 30008000 40000 300000 \;go 30008000
```

设置uboot引导参数：

```
setenv bootargs root=/dev/mtdblock2 init=/linuxrc console=ttySAC0,115200
      rootfstype=jffs2 rw
```

5 设置文件系统从网络引导（挂载NFS文件系统）

A.在虚拟机的linux中设置NFS 共享目录

```
vi /etc/exports
```

B、添加共享目录（将根文件系统目录设置为共享目录）：

```
/root/rootfs *(rw,sync,no_root_squash)
```

C、设置ip，重启NFS 服务

```
ifconfig eth0 192.168.1.111 up
/etc/init.d/nfs reboot  或者  service nfs restart
```

D、在开发板的uboot 上设置命令行参数：

```
setenv bootargs root=/dev/nfs nfsroot=192.168.1.111:/root/rootfs ip=192.168.1.100
init=/linuxrc  console=ttySAC0,115200
```

其中:

root 指定根文件系统为”/dev/nfs”网络根文件系统

nfsroot 指定网络根文件系统的路径是ip 地址为192.168.1.222的主机上的“/root/rootfs”目录，

ip 指定开发板的IP 地址，需与rootfs目录中的rc.local脚本配置的开发板ip一致

```
saveenv
```