u-boot-2010.06 在 unsp2440 上的移植

一、 步骤说明：

- 进行 u-boot 的移植，我们分成以下几步完成：
- NorFlash 启动
- 可以读写 NAND FLASH
- NAND 启动
- 支持网络、USB 下载
- 支持读写 SD 上内核及根文件系统
- 支持 USB 从下载

二、 启动流程简介

- 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：

步骤一：建立 u-boot 下的 unsp2440 开发板目录结构

在 u-boot 的目录树中默认没有 S3C2440 芯片的支持，但是其同 S3C2410 相差不多，我们根据 S3C2410 的一些配置修改得到对 2440 芯片的支持，u-boot 默认仅支持 Nor 启动，我们第一步完成 U-boot 在 NORFLASH 上的启动。

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，所以就在 board/samsung/ 下建立自己开发板的项目，取名叫 unsp2440

(2) #tar jxvf u-boot-2010.06.tar.bz2　　//解压源码

(3) #cd u-boot-2010.06/board/samsung/　　//进入目录

(4) #cp smdk2410/ unsp2440/ -R //将 2410 目录复制一份并重命名为 unsp2440

#cd unsp2440 //进入 unsp2440 目录

#mv smdk2410.c unsp2440.c //将 unsp2440 下的 smdk2410.c 改名为 unsp2440.c

#vi board/samsung/unsp2440/Makefile //修改 unsp2440 下 Makefile 的编译项，如下：

COBJS := unsp2440.o flash.o //因在 unsp2440 下我们将 smdk2410.c 改名为 unsp2440.c

(5) #cp include/configs/smdk2410.h include/configs/unsp2440.h //建立 2440 头文件

(6) 修改 u-boot 跟目录下的 Makefile 文件

查找到 smdk2410_config 的地方，在他下面按照 smdk2410_config 的格式建立 unsp2440_config 的编译选项，另外还要指定交叉编译器

smdk2410_config :unconfig //2410 编译选项格式

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

unsp2440_config :unconfig //2440 编译选项格式

@$(MKCONFIG) $(@:_config=) arm arm920t unsp2440 samsung s3c24x0

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

*说明：

- **arm：** **CPU 的架构(ARCH)**
- **arm920t：** **CPU 的类型**
- **unsp2440 ：** **对应在 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**

- **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

(7) 测试编译新建的 unsp2440 开发板项目

- **#make unsp2440_config　　//如果出现 Configuring for unsp2440 board... 则表示设置正确**
- **#make　　//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了**

到此为止，u-boot 对自己的 unsp2440 开发板还没有任何用处，以上的移植只是搭建了一个 unsp2440 开发板 u-boot 的框架，要使其功能实现，还要根据 unsp2440 开发板的具体资源情况来对 u-boot 源码进行修改。

步骤二：u-boot 支持 s3C2440 NOR 启动

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 unsp2440 开发板（注：修改或添加的地方都用红色表示）。

(1) unsp2440 开发板 u-boot 的 stage1 入口点分析

前面我们知道了程序的入口点是 arch/arm/arm920t/start.S，那么我们就打开 unsp2440 开发板 u-boot 第一个要运行的程序 arch/arm/arm920t/start.S（即 u-boot 的 stage1 部分），查找到_start 的位置如下：

```
.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

/******dec by dengwei***********/

/*bl coloured_LED_init*/

/*bl red_LED_on*/

//此处两行是对 AT91RM9200DK 开发板上的 LED 进行初始化的,作为测试使用，我们这里用不到，所以注
```

(2) unsp2440 开发板 u-boot 的 stage1 阶段的硬件设备初始化。

在 include/configs/unsp2440.h 头文件中添加 CONFIG_S3C2440 宏

#vi include/configs/unsp2440.h

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

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

#define CONFIG_S3C2410 1 /* specifically a SAMSUNG S3C2410 SoC */

#define CONFIG_SMDK2410 1 /* on a SAMSUNG SMDK2410 Board */

#define CONFIG_S3C2440 1 /* specifically a SAMSUNG S3C2440 SoC */
```

(3) 在 u-boot 中添加对 S3C2440 一些寄存器的支持、添加中断禁止部分和时钟设置部分。

1）中断寄存器

由于 2410 和 2440 的寄存器及地址大部分是一致的，所以这里就直接在 2410 的基础上再加上对 2440 的支持即可，代码如下：

#vi cpu/arm920t/start.S

```
# if defined(CONFIG_S3C2410)

ldr r1, =0x3ff

ldr r0, =INTSUBMSK

str r1, [r0]

# endif

/***********add by dengwei**********/

//关闭子中断屏蔽寄存器

# if defined(CONFIG_S3C2440)

ldr r1, =0x7ff

ldr r0, =INTSUBMSK

str r1, [r0]

# endif
```

2）时钟管理：

```
/*******add by dengwei*************************************/

# 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:HCLKCLK = 1:4:8

mov r1, #5

str r1, [r0]

ldr r0, =MPLLCON //设置系统主频为 405MHz

ldr r1, =0x7F021

str r1, [r0]

ldr r0, =UPLLCON //设置 USB 时钟频率为 48MHz

ldr r1, =0x38022

str r1, [r0]

#else

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

/* default FCLK is 120 MHz ! */

ldr r0, =CLKDIVN

mov r1, #3

str r1, [r0]

#endif
```

以上方法是在汇编中直接改变系统的时钟频率，为了实现 u-boot 修改方便同时实现同时支持 2410,2440 启动，我们也可以把时钟的初始化放在一个 C 文件中，做更加复杂的操作

```
/时钟初始化

#ifndef CONFIG_SKIP_LOWLEVEL_INIT

bl clock_init

#endif
```

S3C2440 在 start.S 中修改以上信息，只是修改了第一阶段的时钟，u-boot 在第二阶段会重新初始化系统时钟，还要分别在 board/samsung/unsp2440/unsp2440.c 和 arch/arm/cpu/arm920t/s3c24x0/speed.c 中修改或添加部分代码，如下：

因为 2410 跟 2440 的的时钟控制稍有不同，因此需要根据具体硬件选择不同的参数

```
/* support both of S3C2410 and S3C2440, by dengwei */

extern const char *mtdparts_default;

if ((gpio->GSTATUS1 == 0x32410000) || (gpio->GSTATUS1 == 0x32410002))

{

        /* arch number of SMDK2410-Board */

         gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;

         //2410 MTD 分区表信息

        //mtdparts_default = MTDPARTS_DEFAULT2410;

        //2410 启动引导参数

        //setenv("bootargs", CONFIG_BOOTARGS2410);

}

else

{
```

```
        /* arch number of UNSP2440-Board */
         gd->bd->bi_arch_number = MACH_TYPE_S3C2440;
         /*2440MTD 分区表信息*/
         //mtdparts_default = MTDPARTS_DEFAULT2440;
         //*2440 启动引导参数
         //setenv("bootargs", CONFIG_BOOTARGS2440);
}
```

#vi board/samsung/unsp2440/unsp2440.c

//设置主频和 USB 时钟频率参数与 start.S 中的一致

```
#define FCLK_SPEED 2
#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 //即默认 HCLK 等于 405M
#define M_MDIV 0x7f
```

```
#define M_PDIV 0x2

#define M_SDIV 0x1

#endif

#define USB_CLOCK 2 //USB 默认为 48M

#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

#define U_M_MDIV 0x38

#define U_M_PDIV 0x2

#define U_M_SDIV 0x2

#endif
```

#vi arch/arm/cpu/arm920t/s3c24x0/speed.c

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

```
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;
        /******add by dengwei**********/
       #if defined(CONFIG_S3C2440)
       if(pllreg == MPLL)
       {
          //参考 S3C2440 芯片手册上的公式：PLL=(2 * m * Fin)/(p * 2s)
         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)

{

         struct s3c24x0_clock_power *clk_power = s3c24x0_get_base_clock_power();

         /*******add by dengwei******/

        #if defined(CONFIG_S3C2440)

            return(get_FCLK()/4);

         #endif

           return (readl(&clk_power->CLKDIVN) & 2) ? get_FCLK() / 2 : get_FCLK();

}
```

好了！修改完毕，我们重新编译并将 u-boot.bin 使用 H-JTAG 下载到开发板的 NOR FLASH 中，观察是否会打印启动信息。

步骤三：U-boot支持NAND启动

经过前两步我们的开发板已经支持了 u-boot 的 NOR 启动，在嵌入式开发中，由于 nor FLASH 的速度及容量上的原因，经常被 NAND 所替代，下面我们研究一下怎么从 NAND 上启动我们的 u-boot

因为我们改过的代码需要在拷贝之前调用一段 C 程序，所以需要把对栈空间的初始化放在拷贝之前。

```
/* Set up the stack*/ 从后面移到前面，因为我们在后面的程序中需要调用C语言程序，所以在
化栈空间

stack_setup:
```

```
。。。。。。。

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

#ifndef CONFIG_SKIP_RELOCATE_UBOOT

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

adr r0, _start

ldr r1, _TEXT_BASE

cmp r0, r1

/*根据启动代码的实际地址与链接地址判断是从 RAM 启动或者 FLASH 启动*/

beq clear_bss /*从 ram 启动则继续执行下面的内容*/

ldr r2, _armboot_start

ldr r3, _bss_start

sub r2, r3, r2

#if 1

      bl CopyCode2Ram /*调用C语言函数，从FLASH中拷贝镜像，此函数会自动识别是NOR启动还
数的实现为boot_init.c，需放在board/Samsung/unsp_2440/目录下*/

#else /*原 U-BOOT：从 NOR 启动*/

add r2, r0, r2

copy_loop:

ldmia r0!, {r3-r10}

stmia r1!, {r3-r10}

cmp r0, r2

ble copy_loop

#endif

#endif /* CONFIG_SKIP_RELOCATE_UBOOT */
```

boot_init.c文件是我们后来增加的文件，在u-boot启动的第一个阶段调用，第一阶段代码通过 4k的SRAM运行，因此本函数必须链接到前 4k的地址中，否则会调用不到此函数。

为此我们需做以下修改：

修改board/Samsung/unsp_2440/Makefile，在第 28 行左右的位置修改成以下形式

COBJS := unsp2440.o flash.o boot_init.o

这样就可以把我们编写的boot_init.c文件编译进去。

为了将boot_init.c链接到前 4k代码中，我们还需修改以下文件：

arch/arm/cpu/arm920t/u-boot.lds u-boot 的链接文件

```
.text :
{
    arch/arm/cpu/arm920t/start.o (.text)
    board/samsung/unsp2440/lowlevel_init.o(.text)
    board/samsung/unsp2440/boot_init.o(.text)
    *(.text)
}
```

将lowlevel_init.S跟boot_init.c文件均链接到前 4K SRAM中。

我们可以重新编译并下载到开发板中，将开发板调到 NAND 启动，观察能否启动起来。

步骤四：U-boot支持NAND操作

在上一节中我们说过，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，但不管是从Nor启动或者从Nand启动，进入第二阶段以后，两者的执行流程是相同的。

当 u-boot 的 start.S 运行到“_start_armboot: .word start_armboot”时，就会调用 lib_arm/board.c 中的 start_armboot 函数，至此 u-boot 正式进入第二阶段。

此时注意：以前较早的u-boot版本进入第二阶段后，对Nand Flash的支持有新旧两套代码，新代码在drivers/nand目录下，旧代码在drivers/nand_legacy目录 下，CFG_NAND_LEGACY宏决定了使用哪套代码，如果定义了该宏就使用旧代码，否则使用新代码。

但是现在的u-boot-2010.6 版本对 Nand的初始化、读写实现是基于最近的Linux内核的MTD架构，删除了以前传统的执行方法，使移植没有以前那样复杂了，实现Nand的操作和基本命令都直接在drivers/mtd/nand目录下。

下面我们结合代码来分析一下 u-boot 在第二阶段的执行流程。

```
1. lib_arm/board.c文件中的start_armboot函数调用了drivers/mtd/nand/nand.c文件中的nand

#if defined(CONFIG_CMD_NAND)

//可以看到CONFIG_CMD_NAND宏决定了Nand的初始化

puts ("NAND: ") ;

nand_init();

#endif

2.nand_init调用了同文件下的nand_init_chip函数；

3.nand_init_chip函数调用drivers/mtd/nand/s3c2410_nand.c文件下的board_nand_init函数，
drivers/mtd/nand/nand_base.c函数中的nand_scan函数；

4.nand_scan函数调用了同文件下的nand_scan_ident函数等
```

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，代码参见S3c2440_nand.c：

s3c2440_nand.c

其次，在开发板配置文件 include/configs/my2440.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_NAND_S3C2440 1

#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
块Nand Flash，则需加上

#endif
```

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

COBJS- y + = s3c2440_nand.o

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

重新编译，将u-boot烧入NAND FLASH，并敲入help，可以看到有关NAND FLASH操作的命令已经可以使用。

在启动过程中，我们可以看到一个警告信息：“* Warning - bad CRC or NAND, using default environment”，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命令将环境变量保存在NAND FLASH中，如下：**

```
#ifdef CONFIG_CMD_NAND
      #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
      //注意这个地址不要跟bootloader等其它分区冲突
       #define CONFIG_ENV_SIZE 0x10000 /* Total Size of Environment Sector */
#else
        #define CONFIG_ENV_IS_IN_FLASH 1 //将环境变量存入norflash中
        #define CONFIG_ENV_SIZE 0x10000 /* Total Size of Environment Sector */
#endif
```

重新编译下载上述警告消失，我们可以使用 saveenv 将环境变量的值存入 NAND FLASH 中。

为了测试NAND驱动移植是否成功，我们使用loady命令配合nand命令完成相关测试。

修改include/configs/unsp2440.h中

#define CONFIG_SYS_PROMPT "unsp2440 # " /* Monitor Command Prompt */

重新编译生成 u-boot

```
#loady 0x33000000                                          //采用ymodem协议从串口下载u-b
0x33000000 中

#nand erase 0x0 0x30000                                    //擦除 0x0 到 0x30000 的FLASH地址

#nand write 0x33000000 0x0 0x30000    //将 0x33000000 中数据烧进NAND的 0x0 到 0x30000 中
```

重启开发板，观察提示符由 smdk2410 变为 unsp2440#,说明我们对 NAND FLASH 驱动的修改成功了。

原

步骤五：U-boot 支持 TFTP、nfs 网络下载

前面我们实现了使用串口下载并更新系统的功能，但是嵌入式系统的内核跟根文件系统都比较大，使用串口下载速度太慢，u-boot 提供了通过 TFTP、nfs 等网络下载的功能支持，本节我们完成此项功能的移植与修改。

1) u-boot 默认支持的是 cs8900 网卡，我们的开发板使用的是 DM9000 网卡，所以要做相应的修改。

首先要修改 include/configs/unsp2440.h 中的相关代码：

```
#define CONFIG_NET_MULTI

//#define CONFIG_CS8900 /* we have a CS8900 on-board */

//#define CONFIG_CS8900_BASE 0x19000300

//#define CONFIG_CS8900_BUS16 /* the Linux driver does accesses as shorts */

//屏蔽掉与 cs8900 有关的宏定义

#define CONFIG_DRIVER_DM9000 1

#define CONFIG_DM9000_NO_SROM 1

#define CONFIG_DM9000_BASE 0x18000300 //网卡片选地址

//网卡地址的选择：S3C2440 片外寻址分成 8 个 BLANK，每个 BLANK 的大小为 128M，

//我们的开发板的网卡接在了 BLANK3 上，因此地址等于:128M*3*1024*1024=0x18000000，由于 DM90
IO 地址需要从 0x18000300 开始，数据地址需要从 0x18000304 开始。

#define DM9000_IO 0x18000300

#define DM9000_DATA (0x18000300 + 4) //网卡数据地址

#define CONFIG_DM9000_USE_16BIT 1
```

2) 使用 TFTP、NFS 等服务必须配置相应的 IP 地址、网关等信息，我们同样需要修改 include/congifs/unsp2440.h 文件相应内容：

```
//给 u-boot 加上 ping 命令，用来测试网络通不通

#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_ETHADDR    08:00:3e:26:0a:5b   //开发板 MAC 地址

#define CONFIG_NETMASK    255.255.255.0

#define CONFIG_IPADDR     192.168.1.105       //开发板 IP 地址

#define CONFIG_SERVERIP   192.168.1.103       //Linux 主机 IP 地址
```

下面需要修改 u-boot 的源码以完成对 unsp2440 的支持：

3) 首先要修改 **dm9000 网卡的总线宽度：board/samsung/unsp2440/lowlevel_init.S**

将 56 行左右的

```
#define B3_BWSCON (DW16 + WAIT + UBLB)修改为：

#define B3_BWSCON (DW16)
```

4) 编译下载 u-boot 已测试网络情况

#ping 192.168.220.103

```
dm9000 i/o: 0x18000300, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 08:31:22:22:02:51

operating at 100M full duplex mode
```

```
Using dm9000 device

host 192.168.220.103 is alive
```

代表网络连接正确，为了验证 TFTP 是否可以使用，我们使用 tftp 更新 u-boot.bin

首先在 PC 机上开启一个 TFTP 服务器，这里我们选取 tftpd32.exe 这款小软件作为 TFTP 服务器，首先双击打开此软件，显示以下界面，将要下载的文件 u-boot.bin 文件拷到与 tftpd32.exe 同一个目录下。

打开开发板终端：

```
#tftp 0x33000000 172.20.223.63:u-boot.bin
```

- tftp:采用 tftp 协议
- 0x33000000:下载到内存的地址
- 172.20.223.63:服务器的地址，这里如果不填默认是：

```
#define CONFIG_SERVERIP   192.168.1.103      //unsp2440.h 中此宏所定义的地址
```

终端中有如下显示：

```
dm9000 i/o: 0x18000300, id: 0x90000a46

DM9000: running in 16 bit mode

MAC: 08:31:22:22:02:51

operating at 100M full duplex mode

Using dm9000 device

TFTP from server 172.20.223.63; our IP address is 172.20.223.22

Filename 'u-boot.bin'.

Load address: 0x33000000
```

```
Loading: T ##########

done

Bytes transferred = 144644 (23504 hex)
```

表示下载成功。

步骤六：U-boot 支持 TFTP 下载启动 linux 内核、根文件系统

Linux 内核的下载过程跟上节下载 u-boot.bin 基本一致，但是启动 linux 需要一些其他知识，下面我们来分析一下。

1) 首先是准备 uzImage 镜像

u-boot 采用的 linux 镜像与我们使用 make zImage 编译出的镜像稍有不同，u-boot 采用 uzImage 格式的镜像，uzImage 是由 zImage + 0x40 字节的文件头组成。

经过编译后的 u-boot 在根目录下的 tools 目录中，会有个叫做 mkimage 的工具，他可以给 zImage 添加一个 header，也就是说使得通常我们编译的内核 zImage 添加一个数据头信息部分。

```
使用：中括号括起来的是可选的

mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data_file[:data_file...] image

选项：

-A：set architecture to 'arch'        //用于指定 CPU 类型，比如 ARM

-O：set operating system to 'os'     //用于指定操作系统，比如 Linux

-T：set image type to 'type'          //用于指定 image 类型，比如 Kernel

-C：set compression type 'comp'     //指定压缩类型

-a：set load address to 'addr' (hex) //指定 image 的载入地址

-e：set entry point to 'ep' (hex)     //内核的入口地址，一般为 image 的载入地址+0x40（信息头的大小
```

```
-n：set image name to 'name'        //image 在头结构中的命名

-d：use image data from 'datafile'   //无头信息的 image 文件名

-x：set XIP (execute in place)       //设置执行位置
```

先将 u-boot 下的 tools 中的 mkimage 复制到主机的/usr/local/bin 目录下，这样就可以在主机的任何目录下使用该工具了。

现在我们进入 kernel 生成目录(一般是 arch/arm/boot 目录)，然后执行如下命令，就会在该目录下生成 一个 uImage.img 的镜像文件，把他复制到 tftp 目录下，这就是我们所说的 uImage

```
mkimage -n 'linux-2.6.34' -A arm -O linux -T kernel -C none -a 0x30008000 -e 0x30008000 uImage.img
```

2) linux 内核启动参数

前一节我们已经得到了 u-boot 所需要的 linux 镜像，下面我们设置 linux 启动所需要的参数，首先 u-boot 中机器号（match type）与内核必须统一，linux 内核中的机器号为：

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，我们找到 379 行左右：

s3c2440 ARCH_S3C2440 S3C2440 362

同时 arch/arm/mach-s3c2440/mach-smdk2440.c 文件中 smdk2440_machine_init 函数

MACHINE_START(S3C2440, "SMDK2440") 决定了当前内核的 match type 是 362.

下面我们修改 u-boot 中的 match-types 与之匹配

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_S3C2440 362
```

我们增加以下定义：

```
#define MACH_TYPE_UNSP2440 362
```

修改 board/samsung/unsp2440/unsp2440.c 文件中 board_init 函数

```
gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;

为：

gd->bd->bi_arch_number = MACH_TYPE_UNSP2440;
```

重新编译，我们就能得到支持我们开发板内核的 u-boot 了。

3) 下载 uzImage 镜像：

下载 linux 内核到 NAND FLASH 中的具体地址，由内核中的 FLASH 分区表决定：

在 arch/arm/plat-s3c24xx/common-smdk.c 文件中可找到以下信息：

```
static struct mtd_partition smdk_default_nand_part[] =

{

    [0] =  {

                .name = "bootloader",

                 .offset = 0, //boorloade 分区起始地址

                 .size = 0x30000, //bootloader 分区的大小

          },

         //bootloader 分区与 kernel 分区之间 0x30000---0x50000 的地址用作存放 linux 参数分区

          //这个地址范围要与前面我们移植 NAND 驱动时定义的 CONFIG_ENV_OFFSET 保持统一

          [1] = {
```

```
                .name = "kernel",
                .offset = 0x50000, //kernel 分区起始地址
                .size = 0x300000, //kernel 分区的大小
            },
        [2] = {
                .name = "root",
                .offset = 0x350000, //root 分区起始地址
                .size = 0x3cac000, //root 分区起始地址
            },
};
```

下面我们使用命令行更新并下载内核：

```
#tftp 0x33000000 172.20.223.63:uImage.img
#nand erase 0x50000 0x300000                    //注意这两个数据的大小一定要跟上述表中的 kern
#nand write 0x33000000 0x50000 0x300000         //将 0x33000000 中数据写入 NAND FLASH 中
启动内核：
#从 ram 中启动 kernel
        #bootm 0x33000000
#从 NAND FLASH 中启动
        #nand read 0x33000000 0x50000 0x300000
        # bootm 0x33000000
```

```
#u-boot 提供了另一个命令用来启动内核

        #nboot 0x33000000 0 0x50000        //nboot 代表从 nand 中启动内核，0x33000000 表示内核拷
一块 FLASH，0x50000 代表 kernel 镜像的起始位置，不用指定大小，这个命令会自动判断内核镜像的大

        # bootm 0x33000000
```

如果没有问题的话，在终端中可以看到内核启动信息。

4）下载 linux 根文件系统

Linux 根文件系统有各种各样的格式，有基于 norFLASH 的 jffs2，基于 NAND 的 cramfs 与 yaffs，基于网络的 nfs 等。

a) 这里首先我们先使用 nfs 根文件系统来启动我们的系统。

需要修改 u-boot 传递给 linux 内核的启动参数，我们可以修改 include/configs/unsp2440.h 文件中的宏：

```
#define CONFIG_BOOTARGS "noinitrd root=/dev/nfs
nfsroot=/home/dengwei/linux_system/root_src/rootfs_test/,rsize=1024,wsize=1024
ip=172.20.223.118:172.20.223.151:172.20.223.254:255.255.255.0::eth0:off init=/linuxrc console=tty
```

或者直接在终端下通过命令行完成：

```
setenv bootargs noinitrd root=/dev/nfs
nfsroot=/home/dengwei/linux_system/root_src/rootfs_test/,rsize=1024,wsize=1024
ip=172.20.223.118:172.20.223.151:172.20.223.254:255.255.255.0::eth0:off init=/linuxrc console=tty
```

b) 下面我们启动 NAND FLASH 中的根文件系统

NAND FLASH 中常用两种格式的根文件系统：cramfs、yaffs

cramfs 是压缩的只读根文件系统，u-boot 直接支持

yaffs/yaffs2 是未压缩的可读可写的文件系统，u-boot 需要修改之后才能支持。

这里我们先烧写 cramfs 格式的根文件系统，下节再修改、移植支持 yaffs 格式根文件系统的 u-boot。

cramfs 格式的根文件系统制作请参考其它相关文章，假设我们现在已经有了此格式的根文件系统，并且名字为：rootfs.cramfs。

#tftp 0x33000000 172.20.223.63:rootfs.cramfs

#nand erase 0x350000 0x3cac000　　//注意这两个数据的大小一定要跟上述表中的 root 分区保持一致

#nand write 0x33000000 0x350000 0x3cac000　　//将 0x33000000 中数据写入 NAND FLASH 中

启动内核，如果顺利的话可以观察到我们的根文件系统也顺利的启动起来了。

步骤七：U-boot 支持 yaffs 格式的文件下载

前面我们已经移植、修改好了基于 cramfs 格式的根文件系统，本节我们来修改 u-boot 的源码，使之支持 yaffs 格式的根文件系统。

cramfs 与 yaffs 文件系统的区别：

通常一个 Nnad 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；128MB 以上容量的 Nand Flash 芯片，一页大小通常为 2KB。

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 以页为存取单位等的特点，将文件组织成固定大小的段(Chunk)。以 528B 的页为例，Yaffs/yaffs2 文件系统使用前 512B 存储数据和 16B 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 文件系统页的结构说明：

Yaffs 页结构说明

```
= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ====

字节  用途

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ====

0 - 511             存储数据(分为两个半部)

512 - 515           系统信息

516             数据状态字

517             块状态字

518 - 519           系统信息

520 - 522           后半部 256 字节的 ECC

523 - 524           系统信息

525 - 527           前半部 256 字节的 ECC

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = ====
```

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 00B 区域数据的功能，即可以实现对 Yaffs/Yaffs2 文件系统的读写支持。

我们需要按照以下步骤修改：

1) 在 include/configs/unsp2440.h 中添加 yaffs2 烧写宏定义

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

2) 增加 yaffs 烧写命令：

#gedit common/cmd_nand.c //在 U_BOOT_CMD 中添加

```
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"

#if defined(CONFIG_MTD_NAND_YAFFS)

"nand_write[.yaffs2] -addr of | partition size - write 'size' byte yaffs image/n"

"starting at offset off'from memory address addr'(.yaffs2 for 2048+64 NAND)/n"

#endif

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

"offset 'off' (entire device if not specified)/n"

。 。 。 。

#endif

);
```

3) 在该文件中对 **nand** 操作的 **do_nand** 函数中添加 **yaffs2** 对 **nand** 的操作，如下

```
if (strncmp(cmd, "read", 4) == 0 || strncmp(cmd, "write", 5) == 0)

{
```

```
int read;

if (argc < 4)

goto usage;

addr = (ulong)simple_strtoul(argv[2], NULL, 16);

read = strncmp(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);

}

#if defined(CONFIG_MTD_NAND_YAFFS)

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"))
。 。 。 。
```

4) 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：

```
struct mtd_info
{
u_char type;
u_int32_t flags;
uint64_t size; /* Total size of the MTD */
#if defined(CONFIG_MTD_NAND_YAFFS)
u_char rw_oob;
u_char skipfirstblk;
#endif
。 。 。 。
}
```

5) 在 **nand_write_skip_bad** 函数中添加对 **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;
#if defined(CONFIG_MTD_NAND_YAFFS) //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;

}

#if !defined(CONFIG_MTD_NAND_YAFFS)

if(len_ind_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;

}

#if defined(CONFIG_MTD_NAND_YAFFS)

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;

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

if (rval != 0)

{

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

```
offset, rval);

*length -= left_to_write;

return rval;

}

left_to_write -= write_size;

offset += write_size;

//p_buffer += write_size;

#if defined(CONFIG_MTD_NAND_YAFFS)

if(nand->rw_oob ==1)

{

        p_buffer +=write_size+(write_size/nand->writesize*nand->oobsize);

}

else

{

        p_buffer +=write_size;

}

#else

        p_buffer += write_size;

#endif

}
```

```
return 0;

}
```

6) 在 nand_write_skip_bad 函数中我们看到又对 nand_write 函数进行了访问，所以这一步是到 nand_write 函数中添加对 yaffs2 的支持，如下

```
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;

#if defined(CONFIG_MTD_NAND_YAFFS)

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))*
(datapages -1) *oobsize);

memcpy((void *)(buf +(datapages) *(datasize + oobsize) -oobsize),(void *)(oobtemp),oobsize);

}

}

#endif

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;

#if defined(CONFIG_MTD_NAND_YAFFS)

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

//chip->ops.oobbuf = NULL;

ret = nand_do_write_ops(mtd, to, &chip->ops);

*retlen = chip->ops.retlen;

nand_release_device(mtd);

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

    chip->ops.mode = oldopsmode;
#endif

return ret;

}
```

OK，对 yaffs2 支持的代码已修改完毕，重新编译 u-boot 并下载到 nand 中，启动开发板，在 u-boot 的命令行输入:nand help 查看 nand 的命令，可以看到多了一个 nand write[.yaffs2]的命令，这个就是用来下载 yaffs2 文件系统到 nand 中的命令了。

7) 使用 **nand write[.yaffs2]**命令把事前制作好的 **yaffs2** 文件系统下载到 **Nand Flash** 中

```
#tftp 0x33000000 172.20.223.63:rootfs.yaffs          //用 tftp 将 yaffs2 文件系统下载到内存的 0x33000

#nand erase 0x350000 0x3cac000                       //擦除 Nand 的文件系统分区

#nand write.yaffs2 0x30000000 0x250000 0x658170

//将内存中的 yaffs2 文件系统写入 Nand 的文件系统分区，注意这里的 0x658170 是 yaffs2 文件系统的实
528 整除才可以
```

原文链接：http://emb.sunplusedu.com/answer/2013/0822/2128.html