U-BOOT 源码分析及移植

本文从以下几个方面粗浅地分析 u-boot 并移植到 FS2410 板上:
1、u-boot 工程的总体结构
2、u-boot 的流程、主要的数据结构、内存分配。
3、u-boot 的重要细节，主要分析流程中各函数的功能。
4、基于 FS2410 板子的 u-boot 移植。实现了 NOR Flash 和 NAND Flash 启动,网络功能。
这些认识源于自己移植 u-boot 过程中查找的资料和对源码的简单阅读。下面主要以 smdk2410 为分析对象。

一、u-boot 工程的总体结构:
1、源代码组织
对于 ARM 而言，主要的目录如下:

board	平台依赖	存放电路板相关的目录文件,每一套板子对 应一个目录。如 smdk2410(arm920t)
cpu	平台依赖	存放 CPU 相关的目录文件，每一款 CPU 对应一个目录，例如：arm920t、xscale、i386 等目录
lib_arm	平台依赖	存放对 ARM 体系结构通用的文件，主要用于实现 ARM 平台通用的函数，如软件浮点。
common	通用	通用的多功能函数实现，如环境，命令，控制台相关的函数实现。
include	通用	头文件和开发板配置文件，所有开发板的配置文件都在 configs 目录下
lib_generic	通用	通用库函数的实现
net	通用	存放网络协议的程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动，nand 驱动。

.....
2.makefile 简要分析
所有这些目录的编译连接都是由顶层目录的 makefile 来确定的。
在执行 make 之前，先要执行 make $(board)_config 对工程进行配置，以确定特定于目标板的各个子目录和头文件。
$(board)_config:是 makefile 中的一个伪目标，它传入指定的 CPU，ARCH，BOARD，SOC 参数去执行 mkconfig 脚本。
这个脚本的主要功能在于连接目标板平台相关的头文件夹，生成 config.h 文件包含板子的配置头文件。
使得 makefile 能根据目标板的这些参数去编译正确的平台相关的子目录。
以 smdk2410 板为例,执行　make smdk2410_config,
主要完成三个功能:
@在 include 文件夹下建立相应的文件（夹）软连接，

#如果是 ARM 体系将执行以下操作:
#ln -s　asm-arm　　asm
#ln -s arch-s3c24x0　asm-arm/arch
#ln -s　proc-armv　　asm-arm/proc

@生成 Makefile 包含文件 include/config.mk，内容很简单，定义了四个变量:

ARCH　= arm
CPU　= arm920t
BOARD　= smdk2410
SOC　= s3c24x0

@生成 include/config.h 头文件，只有一行:

/* Automatically generated - do not edit */
#include "config/smdk2410.h"

顶层 makefile 先调用各子目录的 makefile，生成目标文件或者目标文件库。
然后再连接所有目标文件（库）生成最终的 u-boot.bin。

连接的主要目标（库）如下：

```
OBJS  = cpu/$(CPU)/start.o
LIBS  = lib_generic/libgeneric.a
LIBS += board/$(BOARDDIR)/lib$(BOARD).a
LIBS += cpu/$(CPU)/lib$(CPU).a
ifdef SOC
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
endif
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBS += fs/cramfs/libcramfs.a fs/fat/libfat.a fs/fdos/libfdos.a fs/jffs2/libjffs2.a \
 fs/reiserfs/libreiserfs.a fs/ext2/libext2fs.a
LIBS += net/libnet.a
LIBS += disk/libdisk.a
LIBS += rtc/librtc.a
LIBS += dtt/libdtt.a
LIBS += drivers/libdrivers.a
LIBS += drivers/nand/libnand.a
LIBS += drivers/nand_legacy/libnand_legacy.a
LIBS += drivers/sk98lin/libsk98lin.a
LIBS += post/libpost.a post/cpu/libcpu.a
LIBS += common/libcommon.a
LIBS += $(BOARDLIBS)
```

显然跟平台相关的主要是：

```
cpu/$(CPU)/start.o
board/$(BOARDDIR)/lib$(BOARD).a
cpu/$(CPU)/lib$(CPU).a
cpu/$(CPU)/$(SOC)/lib$(SOC).a
lib_$(ARCH)/lib$(ARCH).a
```

这里面的四个变量定义在 include/config.mk（见上述）。
其余的均与平台无关。
所以考虑移植的时候也主要考虑这几个目标文件（库）对应的目录。

关于 u-boot 的 makefile 更详细的分析可以参照 http://blog.mcuol.com/User/lvembededsys/Article/4355_1.htm。

3、u-boot 的通用目录是怎么做到与平台无关的?
include/config/smdk2410.h
这个头文件中主要定义了两类变量。
一类是选项，前缀是 CONFIG_，用来选择处理器、设备接口、命令、属性等，主要用来 决定是否编译某些文件或者函数。

另一类是参数，前缀是 CFG_，用来定义总线频率、串口波特率、Flash 地址等参数。这些常数参量主要用来支持通用目录中的代码，定义板子资源参数。

这两类宏定义对 u-boot 的移植性非常关键，比如 drive/CS8900.c，对 cs8900 而言，很多操作都是通用的，但不是所有的板子上面都有这个芯片，即使有它在内存中映射的基地址也是平台相关的。所以对于 smdk2410 板，在 smdk2410.h 中定义了

```
#define CONFIG_DRIVER_CS8900 1              /* we have a CS8900 on-board */
#define CS8900_BASE    0x19000300           /*IO mode base address*/
```

CONFIG_DRIVER_CS8900 的定义使得 cs8900.c 可以被编译（当然还得定义 CFG_CMD_NET 才行），因为 cs8900.c 中在函数定义的前面就有编译条件判断：#ifdef CONFIG_DRIVER_CS8900　如果这个选项没有定义，整个 cs8900.c 就不会被编译了。
而常数参量 CS8900_BASE 则用在 cs8900.h 头文件中定义各个功能寄存器的地址。u-boot 的 CS8900 工作在 IO 模式下，只要给定 IO 寄存器在内存中映射的基地址，其余代码就与平台无关了。

u-boot 的命令也是通过目标板的配置头文件来配置的，比如要添加 ping 命令，就必须添加 CFG_CMD_NET 和 CFG_CMD_PING 才行。不然 common/cmd_net.c 就不会被编译了。
从这里我可以这么认为，u-boot 工程可配置性和移植性可以分为两层：
一是由 makefile 来实现，配置工程要包含的文件和文件夹上，用什么编译器。

二是由目标板的配置头文件来实现源码级的可配置性，通用性。主要使用的是#ifdef #else #endif 之类来实现的。
4、smkd2410 其余重要的文件：
include/s3c24x0.h　　定义了 s3x24x0 芯片的各个特殊功能寄存器（SFR）的地址。
cpu/arm920t/start.s　　在 flash 中执行的引导代码,也就是 bootloader 中的 stage1,负责初始化硬件环境，把 u-boot 从 flash 加载到 RAM 中去，然后跳到 lib_arm/board.c 中的 start_armboot 中去执行。
lib_arm/board.c　　u-boot 的初始化流程，尤其是 u-boot 用到的全局数据结构 gd,bd 的初始化，以及设备和控制台的初始化。
board/smdk2410/flash.c　　在 board 目录下代码的都是严重依赖目标板，对于不同的 CPU，SOC，ARCH，u-boot 都有相对通用的代码，但是板子构成却是多样的，主要是内存地址，flash 型号，外围芯片如网络。对 fs2410 来说，主要考虑从 smdk2410 板来移植，差别主要在 nor flash 上面。

二、u-boot 的流程、主要的数据结构、内存分配
1、u-boot 的启动流程：
　　从文件层面上看主要流程是在两个文件中：cpu/arm920t/start.s，lib_arm/board.c，
　　1)start.s
　在 flash 中执行的引导代码,也就是 bootloader 中的 stage1,负责初始化硬件环境，把 u-boot 从 flash 加载到 RAM 中去，然后跳到 lib_arm/board.c 中的 start_armboot 中去执行。
1.1.6 版本的 start.s 流程：
硬件环境初始化：
　　进入 svc 模式;关闭 watch dog;屏蔽所有 IRQ 掩码;设置时钟频率 FCLK、HCLK、PCLK;清 I/D cache;禁止 MMU 和 CACHE;配置 memory control;
重定位：
　　如果当前代码不在连接指定的地址上（对 smdk2410 是 0x3f000000）则需要把 u-boot 从当前位置拷贝到 RAM 指定位置中;
建立堆栈，堆栈是进入 C 函数前必须初始化的。
清.bss 区。
跳到 start_armboot 函数中执行。（lib_arm/board.c）
　2)lib_arm/board.c:
　start_armboot 是 U-Boot 执行的第一个 C 语言函数，完成系统初始化工作，进入主循环，处理用户输入的命令。这里只简要列出了主要执行的函数流程：

```
  void start_armboot (void)
  {
    //全局数据变量指针 gd 占用 r8。
      DECLARE_GLOBAL_DATA_PTR;

      /* 给全局数据变量 gd 安排空间*/
      gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));
      memset ((void*)gd, 0, sizeof (gd_t));

      /* 给板子数据变量 gd->bd 安排空间*/
      gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
      memset (gd->bd, 0, sizeof (bd_t));
      monitor_flash_len = _bss_start - _armboot_start;//取 u-boot 的长度。

      /* 顺序执行 init_sequence 数组中的初始化函数 */
      for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
            if ((*init_fnc_ptr)() != 0) {
                  hang ();
            }
      }

      /*配置可用的 Flash */
      size = flash_init ();
       ……
      /* 初始化堆空间 */
      mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);
      /* 重新定位环境变量， */
      env_relocate ();
```

```
        /* 从环境变量中获取 IP 地址 */
        gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");
        /* 以太网接口 MAC 地址 */
        ……
        devices_init ();      /* 设备初始化 */
        jumptable_init ();  //跳转表初始化
        console_init_r ();    /* 完整地初始化控制台设备 */
        enable_interrupts (); /* 使能中断处理 */
        /* 通过环境变量初始化 */
        if ((s = getenv ("loadaddr")) != NULL) {
            load_addr = simple_strtoul (s, NULL, 16);
        }
        /* main_loop()循环不断执行 */
        for (;;) {
            main_loop ();      /* 主循环函数处理执行用户命令 -- common/main.c */
        }
    }
```

初始化函数序列 init_sequence[]

init_sequence[]数组保存着基本的初始化函数指针。这些函数名称和实现的程序文件在下列注释中。

```
init_fnc_t *init_sequence[] = {
        cpu_init,            /* 基本的处理器相关配置 -- cpu/arm920t/cpu.c */
        board_init,          /* 基本的板级相关配置 -- board/smdk2410/smdk2410.c */
        interrupt_init,      /* 初始化例外处理 -- cpu/arm920t/s3c24x0/interrupt.c */
        env_init,            /* 初始化环境变量 -- common/env_flash.c */
        init_baudrate,       /* 初始化波特率设置 -- lib_arm/board.c */
        serial_init,         /* 串口通讯设置 -- cpu/arm920t/s3c24x0/serial.c */
        console_init_f,      /* 控制台初始化阶段 1 -- common/console.c */
        display_banner,      /* 打印 u-boot 信息 -- lib_arm/board.c */
        dram_init,           /* 配置可用的 RAM -- board/smdk2410/smdk2410.c */
        display_dram_config, /* 显示 RAM 的配置大小 -- lib_arm/board.c */
        NULL,
};
```

整个 u-boot 的执行就进入等待用户输入命令，解析并执行命令的死循环中。

2、u-boot 主要的数据结构

u-boot 的主要功能是用于引导 OS 的，但是本身也提供许多强大的功能，可以通过输入命令行来完成许多操作。所以它本身也是一个很完备的系统。u-boot 的大部分操作都是围绕它自身的数据结构，这些数据结构是通用的，但是不同的板子初始化这些数据就不一样了。所以 u-boot 的通用代码是依赖于这些重要的数据结构的。这里说的数据结构其实就是一些全局变量。

1）gd　全局数据变量指针，它保存了 u-boot 运行需要的全局数据，类型定义：

```
typedef struct global_data {
        bd_t  *bd;      //board data pointor 板子数据指针
        unsigned long flags;     //指示标志，如设备已经初始化标志等。
        unsigned long baudrate; //串口波特率
        unsigned long have_console; /* 串口初始化标志*/
        unsigned long reloc_off;   /* 重定位偏移，就是实际定向的位置与编译连接时指定的位置之差，一般为 0 */
        unsigned long env_addr; /* 环境参数地址*/
        unsigned long env_valid; /* 环境参数 CRC 检验有效标志 */
        unsigned long fb_base; /* base address of frame buffer */
          #ifdef CONFIG_VFD
        unsigned char vfd_type; /* display type */
          #endif
        void  **jt;  /* 跳转表，1.1.6 中用来函数调用地址登记 */
        } gd_t;
```

2)bd 板子数据指针。板子很多重要的参数。 类型定义如下：

```
typedef struct bd_info {
        int   bi_baudrate;    /* 串口波特率 */
        unsigned long bi_ip_addr;   /* IP 地址 */
        unsigned char bi_enetaddr[6]; /* MAC 地址*/
        struct environment_s       *bi_env;
        ulong        bi_arch_number; /* unique id for this board */
        ulong        bi_boot_params; /* 启动参数 */
        struct    /* RAM 配置 */
        {
       ulong start;
       ulong size;
        }bi_dram[CONFIG_NR_DRAM_BANKS];
    } bd_t;
```

3)环境变量指针 env_t *env_ptr = (env_t *)(&environment[0]);(common/env_flash.c)

env_ptr 指向环境参数区，系统启动时默认的环境参数 environment[]，定义在 common/environment.c 中。

参数解释：

bootdelay 定义执行自动启动的等候秒数
baudrate 定义串口控制台的波特率
netmask 定义以太网接口的掩码
ethaddr 定义以太网接口的 MAC 地址
bootfile 定义缺省的下载文件
bootargs 定义传递给 Linux 内核的命令行参数
bootcmd 定义自动启动时执行的几条命令
serverip 定义 tftp 服务器端的 IP 地址
ipaddr 定义本地的 IP 地址
stdin 定义标准输入设备，一般是串口
stdout 定义标准输出设备，一般是串口
stderr 定义标准出错信息输出设备，一般是串口

4）设备相关：

标准 IO 设备数组□evice_t *stdio_devices[] = { NULL, NULL, NULL };
设备列表   list_t   devlist = 0;
device_t 的定义：include\devices.h 中：

```
typedef struct {
 int flags;                    /* Device flags: input/output/system */
 int ext;                     /* Supported extensions   */
 char name[16];                  /* Device name    */
/* GENERAL functions */
 int (*start) (void);           /* To start the device   */
 int (*stop) (void);             /* To stop the device   */
/* 输出函数 */
 void (*putc) (const char c); /* To put a char   */
 void (*puts) (const char *s); /* To put a string (accelerator) */
/* 输入函数 */
 int (*tstc) (void);             /* To test if a char is ready... */
 int (*getc) (void);              /* To get that char   */
/* Other functions */
 void *priv;                       /* Private extensions   */
} device_t;
```

u-boot 把可以用为控制台输入输出的设备添加到设备列表 devlist,并把当前用作标准 IO 的设备指针加入 stdio_devices 数组中。

在调用标准 IO 函数如 printf()时将调用 stdio_devices 数组对应设备的 IO 函数如 putc()。

5)命令相关的数据结构，后面介绍。

6)与具体设备有关的数据结构，

如 flash_info_t flash_info[CFG_MAX_FLASH_BANKS];记录 nor flash 的信息。

nand_info_t nand_info[CFG_MAX_NAND_DEVICE]; nand flash 块设备信息

3、u-boot 重定位后的内存分布：

对于 smdk2410,RAM 范围从 0x30000000~0x34000000. u-boot 占用高端内存区。从高地址到低地址内存分配

如下：

显示缓冲区 (.bss_end~34000000)
u-boot(bss,data,text) (33f00000~.bss_end)
heap(for malloc)
gd(global data)
bd(board data)
stack
....
nor flash (0~2M)

三、u-boot 的重要细节。

主要分析流程中各函数的功能。按启动顺序罗列一下启动函数执行细节。按照函数 start_armboot 流程进行分析：

1)DECLARE_GLOBAL_DATA_PTR;

这个宏定义在 include/global_data.h 中：

#define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r8")

声明一个寄存器变量 gd 占用 r8。这个宏在所有需要引用全局数据指针 gd_t *gd 的源码中都有申明。

这个申明也避免编译器把 r8 分配给其它的变量. 所以 gd 就是 r8,这个指针变量不占用内存。

2）gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t));

对全局数据区进行地址分配，_armboot_start 为 0x3f000000,CFG_MALLOC_LEN 是堆大小＋环境数据区大小，config/smdk2410.h 中 CFG_MALLOC_LEN 大小定义为 192KB.

3)gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));

分配板子数据区 bd 首地址。

这样结合 start.s 中栈的分配，

stack_setup：

```
ldr r0, _TEXT_BASE  /* upper 128 KiB: relocated uboot   */
sub r0, r0, #CFG_MALLOC_LEN /* malloc area                    */
sub r0, r0, #CFG_GBL_DATA_SIZE /* bdinfoCFG_GBL_DATA_SIZE =128B */
#ifdef CONFIG_USE_IRQ
sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
sub sp, r0, #12  /* leave 3 words for abort-stack    */
```

不难得出上文所述的内存分配结构。

下面几个函数是初始化序列表 init_sequence[]中的函数:

4)cpu_init();定义于 cpu/arm920t/cpu.c

分配 IRQ，FIQ 栈底地址，由于没有定义 CONFIG_USE_IRQ,所以相当于空实现。

5)board_init；极级初始化，定义于 board/smdk2410/smdk2410.c

设置 PLL 时钟，GPIO，使能 I/D cache.

设置 bd 信息：gd->bd->bi_arch_number = MACH_TYPE_SMDK2410;//板子的 ID，没啥意义。

gd->bd->bi_boot_params = 0x30000100;//内核启动参数存放地址

6)interrupt_init;定义于 cpu/arm920t/s3c24x0/interrupt.c

初始化 2410 的 PWM timer 4,使其能自动装载计数值，恒定的产生时间中断信号，但是中断被屏蔽了用不上。

7)env_init;定义于 common/env_flash.c（搜索的时候发现别的文件也定义了这个函数，而且没有宏定义保证只有一个被编译，这是个问题，有高手知道指点一下！）

功能：指定环境区的地址。default_environment 是默认的环境参数设置。

```
gd->env_addr  = (ulong)&default_environment[0];
gd->env_valid = 0;
```

8)init_baudrate；初始化全局数据区中波特率的值

```
gd->bd->bi_baudrate = gd->baudrate =(i > 0)
 ? (int) simple_strtoul (tmp, NULL, 10)
 : CONFIG_BAUDRATE;
```

9)serial_init; 串口通讯设置 定义于 cpu/arm920t/s3c24x0/serial.c

根据 bd 中波特率值和 pclk,设置串口寄存器。

10)console_init_f;控制台前期初始化 common/console.c

由于标准设备还没有初始化（gd->flags & GD_FLG_DEVINIT=0），这时控制台使用串口作为控制台

函数只有一句：gd->have_console = 1;

10)dram_init,初始化内存 RAM 信息。board/smdk2410/smdk2410.c
其实就是给 gd->bd 中内存信息表赋值而已。
gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;
初始化序列表 init_sequence[]主要函数分析结束。
11)flash_init；定义在 board/smdk2410/flash.c
这个文件与具体平台关系密切，smdk2410 使用的 flash 与 FS2410 不一样，所以移植时这个程序就得重写。
flash_init()是必须重写的函数，它做哪些操作呢?
首先是有一个变量 flash_info_t flash_info[CFG_MAX_FLASH_BANKS]来记录 flash 的信息。flash_info_t 定义：
typedef struct {
ulong size; /* 总大小 BYTE */
ushort sector_count; /* 总的 sector 数*/
ulong flash_id; /* combined device & manufacturer code */
ulong start[CFG_MAX_FLASH_SECT]; /* 每个 sector 的起始物理地址。 */
uchar protect[CFG_MAX_FLASH_SECT]; /* 每个 sector 的保护状态，如果置 1，在执行 erase 操作的时候将跳过对应 sector*/
#ifdef CFG_FLASH_CFI //我不管 CFI 接口。
.....
#endif
} flash_info_t;
flash_init()的操作就是读取 ID 号，ID 号指明了生产商和设备号，根据这些信息设置 size,sector_count,flash_id.以及 start[]、protect[]。
12)把视频帧缓冲区设置在 bss_end 后面。
addr = (_bss_end + (PAGE_SIZE - 1)) & ~(PAGE_SIZE - 1);
size = vfd_setmem (addr);
gd->fb_base = addr;
13)mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);
设置 heap 区，供 malloc 使用。下面的变量和函数定义在 lib_arm/board.c
malloc 可用内存由 mem_malloc_start，mem_malloc_end 指定。而当前分配的位置则是 mem_malloc_brk。
mem_malloc_init 负责初始化这三个变量。malloc 则通过 sbrk 函数来使用和管理这片内存。

```
static ulong mem_malloc_start = 0;
static ulong mem_malloc_end = 0;
static ulong mem_malloc_brk = 0;

static
void mem_malloc_init (ulong dest_addr)
{
 mem_malloc_start = dest_addr;
 mem_malloc_end = dest_addr + CFG_MALLOC_LEN;
 mem_malloc_brk = mem_malloc_start;

 memset ((void *) mem_malloc_start, 0,
   mem_malloc_end - mem_malloc_start);
}
void *sbrk (ptrdiff_t increment)
{
 ulong old = mem_malloc_brk;
 ulong new = old + increment;

 if ((new < mem_malloc_start) || (new > mem_malloc_end)) {
  return (NULL);
 }
 mem_malloc_brk = new;
 return ((void *) old);
}
```

14)env_relocate() 环境参数区重定位
由于初始化了 heap 区，所以可以通过 malloc()重新分配一块环境参数区，
但是没有必要，因为默认的环境参数已经重定位到 RAM 中了。

/**这里发现个问题，ENV_IS_EMBEDDED 是否有定义还没搞清楚，而且 CFG_MALLOC_LEN 也没有定义，也就是说如果 ENV_IS_EMBEDDED 没有定义则执行 malloc,是不是应该有问题？**/

15)IP，MAC 地址的初始化。主要是从环境中读，然后赋给 gd->bd 对应域就 OK。

16）devices_init ();定义于 common/devices.c

```
int devices_init (void)//我去掉了编译选项，注释掉的是因为对应的编译选项没有定义。
 {
  devlist = ListCreate (sizeof (device_t));//创建设备列表
  i2c_init (CFG_I2C_SPEED, CFG_I2C_SLAVE);//初始化 i2c 接口，i2c 没有注册到 devlist 中去。
  //drv_lcd_init ();
  //drv_video_init ();
  //drv_keyboard_init ();
  //drv_logbuff_init ();
  drv_system_init ();       //这里其实是定义了一个串口设备，并且注册到 devlist 中。
  //serial_devices_init ();
  //drv_usbtty_init ();
  //drv_nc_init ();
 }
```

经过 devices_init()，创建了 devlist，但是只有一个串口设备注册在内。显然，devlist 中的设备都是可以做为 console 的。

16) jumptable_init ();初始化 gd->jt。1.1.6 版本的 jumptable 只起登记函数地址的作用。并没有其他作用。

17)console_init_r ();后期控制台初始化

主要过程：查看环境参数 stdin,stdout,stderr 中对标准 IO 的指定的设备名称，再按照环境指定的名称搜索 devlist，将搜到的设备指针赋给标准 IO 数组 stdio_devices[]。置 gd->flag 标志 GD_FLG_DEVINIT。这个标志影响 putc，getc 函数的实现，未定义此标志时直接由串口 serial_getc 和 serial_putc 实现，定义以后通过标准设备数组 stdio_devices[]中的 putc 和 getc 来实现 IO。

下面是相关代码：

```
 void putc (const char c)
     {
     #ifdef CONFIG_SILENT_CONSOLE
      if (gd->flags & GD_FLG_SILENT)//GD_FLG_SILENT 无输出标志
       return;
     #endif
      if (gd->flags & GD_FLG_DEVINIT) {//设备 list 已经初始化
       /* Send to the standard output */
       fputc (stdout, c);
      } else {
       /* Send directly to the handler */
       serial_putc (c);//未初始化时直接从串口输出。
      }
     }
    void fputc (int file, const char c)
     {
      if (file < MAX_FILES)
       stdio_devices[file]->putc (c);
     }
```

为什么要使用 devlist，std_device[]?

为了更灵活地实现标准 IO 重定向，任何可以作为标准 IO 的设备，如 USB 键盘，LCD 屏，串口等都可以对应一个 device_t 的结构体变量，只需要实现 getc 和 putc 等函数，就能加入到 devlist 列表中去，也就可以被 assign 为标准 IO 设备 std_device 中去。如函数

int console_assign (int file, char *devname); /* Assign the console 重定向标准输入输出*/

这个函数功能就是把名为 devname 的设备重定向为标准 IO 文件 file(stdin,stdout,stderr)。其执行过程是在 devlist 中查找 devname 的设备，返回这个设备的 device_t 指针，并把指针值赋给 std_device[file]。

18)enable_interrupts(),使能中断。由于 CONFIG_USE_IRQ 没有定义，空实现。

```
      #ifdef CONFIG_USE_IRQ
  /* enable IRQ interrupts */
  void enable_interrupts (void)
  {
   unsigned long temp;
   __asm__ __volatile__("mrs %0, cpsr\n"
        "bic %0, %0, #0x80\n"
        "msr cpsr_c, %0"
        : "=r" (temp)
        :
        : "memory");
  }
          #else
     void enable_interrupts (void)
  {
  }
```

19）设置 CS8900 的 MAC 地址。

```
cs8900_get_enetaddr (gd->bd->bi_enetaddr);
```

20）初始化以太网。

```
eth_initialize(gd->bd);//bd 中已经 IP，MAC 已经初始化
```

21）main_loop ()；定义于 common/main.c

至此所有初始化工作已经完毕。main_loop 在标准转入设备中接受命令行，然后分析，查找，执行。

关于 U-boot 中命令相关的编程:

1、命令相关的函数和定义

＠main_loop：这个函数里有太多编译选项，对于 smdk2410,去掉所有选项后等效下面的程序

```
void main_loop()
 {
  static char lastcommand[CFG_CBSIZE] = { 0, };
  int len;
  int rc = 1;
  int flag;
   char *s;
  int bootdelay;
  s = getenv ("bootdelay");  //自动启动内核等待延时
  bootdelay = s ? (int)simple_strtol(s, NULL, 10) : CONFIG_BOOTDELAY;

  debug ("### main_loop entered: bootdelay=%d\n\n", bootdelay);
  s = getenv ("bootcmd");  //取得环境中设置的启动命令行
  debug ("### main_loop: bootcmd=\"%s\"\n", s ? s : "");

  if (bootdelay >= 0 && s && !abortboot (bootdelay))
  {
   run_command (s, 0);//执行启动命令行,smdk2410.h 中没有定义 CONFIG_BOOTCOMMAND，所以没有命令执
行。
  }

  for (;;) {
  len = readline(CFG_PROMPT);//读取键入的命令行到 console_buffer

   flag = 0; /* assume no special flags for now */
   if (len > 0)
    strcpy (lastcommand, console_buffer);//拷贝命令行到 lastcommand.
   else if (len == 0)
    flag |= CMD_FLAG_REPEAT;
```

```
    if (len == -1)
    puts ("\n");
    else
    rc = run_command (lastcommand, flag);   //执行这个命令行。

    if (rc <= 0) {
     /* invalid command or not repeatable, forget it */
     lastcommand[0] = 0;
   }
  }
```

@run_comman();在命令 table 中查找匹配的命令名称，得到对应命令结构体变量指针，以解析得到的参数调用其处理函数执行命令。

@命令结构构体类型定义：command.h 中，

```
struct cmd_tbl_s {
 char  *name;                    /* 命令名   */
 int  maxargs;                   /* 最大参数个数 maximum number of arguments */
 int  repeatable; /* autorepeat allowed?  */
                                  /* Implementation function 命令执行函数*/
 int  (*cmd)(struct cmd_tbl_s *, int, int, char *[]);
 char  *usage;                   /* Usage message (short) */
#ifdef CFG_LONGHELP
 char  *help;                    /* Help  message (long) */
#endif
#ifdef CONFIG_AUTO_COMPLETE
                                 /* do auto completion on the arguments */
 int  (*complete)(int argc, char *argv[], char last_char, int maxv, char *cmdv[]);
#endif
};
typedef struct cmd_tbl_s cmd_tbl_t;


//定义 section 属性的结构体。编译的时候会单独生成一个名为.u_boot_cmd 的 section 段。
#define Struct_Section  __attribute__ ((unused,section (".u_boot_cmd")))


//这个宏定义一个命令结构体变量。并用 name,maxargs,rep,cmd,usage,help 初始化各个域。
#define U_BOOT_CMD(name,maxargs,rep,cmd,usage,help) \
cmd_tbl_t __u_boot_cmd_##name Struct_Section = {#name, maxargs, rep, cmd, usage, help}
```

2、在 u-boot 中，如何添加一个命令：

1）CFG_CMD_* 命令选项位标志。在 include/cmd_confdefs.h 中定义。

每个板子的配置文件（如 include/config/smdk2410.h）中都可以定义 u-boot 需要的命令，如果要添加一个命令，必须添加相应的命令选项。如下：

```
 #define CONFIG_COMMANDS \
(CONFIG_CMD_DFL  | \
CFG_CMD_CACHE  | \
/*CFG_CMD_NAND  |*/ \
/*CFG_CMD_EEPROM |*/ \
/*CFG_CMD_I2C  |*/ \
/*CFG_CMD_USB  |*/ \
CFG_CMD_REGINFO  | \
CFG_CMD_DATE  | \
CFG_CMD_ELF)
```

定义这个选项主要是为了编译命令需要的源文件，大部分命令都在 common 文件夹下对应一个源文件 cmd_*.c ，如 cmd_cache.c 实现 cache 命令。 文件开头就有一行编译条件：

```
#if(CONFIG_COMMANDS&CFG_CMD_CACHE)
```

也就是说，如果配置头文件中 CONFIG_COMMANDS 不或上相应命令的选项，这里就不会被编译。

2）定义命令结构体变量，如：

```
U_BOOT_CMD(
dcache, 2, 1, do_dcache,
"dcache - enable or disable data cache\n",
"[on, off]\n"
"    - enable or disable data (writethrough) cache\n"
);
```

其实就是定义了一个 cmd_tbl_t 类型的结构体变量，这个结构体变量名为__u_boot_cmd_dcache。

其中变量的五个域初始化为括号的内容。分别指明了命令名，参数个数，重复数，执行命令的函数，命令提示。

每个命令都对应这样一个变量，同时这个结构体变量的 section 属性为.u_boot_cmd.也就是说每个变量编译结束在目标文件中都会有一个.u_boot_cmd 的 section.一个 section 是连接时的一个输入段，如.text,.bss,.data 等都是 section 名。

最后由链接程序把所有的.u_boot_cmd 段连接在一起，这样就组成了一个命令结构体数组。

u-boot.lds 中相应脚本如下:

```
. = .;
__u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;
```

可以看到所有的命令结构体变量集中在__u_boot_cmd_start 开始到__u_boot_cmd_end 结束的连续地址范围内，这样形成一个 cmd_tbl_t 类型的数组，run_command 函数就是在这个数组中查找命令的。

3）实现命令处理函数。命令处理函数的格式:

```
void function (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
```

总体来说，如果要实现自己的命令，应该在 include/com_confdefs.h 中定义一个命令选项标志位。

在板子的配置文件中添加命令自己的选项。按照 u-boot 的风格，可以在 common/下面添加自己的 cmd_*.c，并且定义自己的命令结构体变量，如 U_BOOT_CMD(

```
mycommand, 2, 1, do_mycommand,
"my command!\n",
"...\n"
" ..\n"
);
```

然后实现自己的命令处理函数 do_mycommand(cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])。

四、U-boot 在 ST2410 的移植，基于 NOR FLASH 和 NAND FLASH 启动。

1、从 smdk2410 到 ST2410:

ST2410 板子的核心板与 FS2410 是一样的。我没有整到 smdk2410 的原理图，从网上得知的结论总结如下，fs2410 与 smdk2410 RAM 地址空间大小一致(0x30000000~0x34000000=64MB);

NOR FLASH 型号不一样，FS2410 用 SST39VF1601 系列的，smdk2410 用 AMD 产 LV 系列的;

网络芯片型号和在内存中映射的地址完全一致（CS8900，IO 方式基地址 0x19000300）

2、移植过程:

移植 u-boot 的基本步骤如下

(1) 在顶层 Makefile 中为开发板添加新的配置选项，使用已有的配置项目为例。

```
smdk2410_config : unconfig
@./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24×0
```

参考上面 2 行，添加下面 2 行。

```
fs2410_config : unconfig
@./mkconfig $(@:_config=) arm arm920t fs2410 NULL s3c24×0
```

(2) 创建一个新目录存放开发板相关的代码，并且添加文件。

```
board/fs2410/config.mk
board/fs2410/flash.c
board/fs2410/fs2410.c
board/fs2410/Makefile
```

board/fs2410/memsetup.S
board/fs2410/u-boot.lds
注意将 board/fs2410/Makefile 中 smdk2410.o 全部改为 fs2410.o
(3) 为开发板添加新的配置文件
可以先复制参考开发板的配置文件，再修改。例如：
$cp include/configs/smdk2410.h include/configs/fs2410.h
如果是为一颗新的 CPU 移植，还要创建一个新的目录存放 CPU 相关的代码。

(4) 配置开发板
$ make fs2410_config

3、移植要考虑的问题：
从 smdk2410 到 ST2410 移植要考虑的主要问题就是 NOR flash。从上述分析知道，u-boot 启动时要执行 flash_init() 检测 flash 的 ID 号，大小，secotor 起始地址表和保护状态表 ,这些信息全部保存在 flash_info_t flash_info[CFG_MAX_FLASH_BANKS]中。
另外，u-boot 中有一些命令如 saveenvt 需要要擦写 flash，间接调用两个函数：flash_erase 和 write_buff。在 board/smdk2410/flash.c
实现了与 smdk2410 板子相关的 nor flash 函数操作。由于 write_buffer 中调用了 write_hword 去具体写入一个字到 flash 中，这个函数本身是与硬件无关的，
所以与硬件密切相关的三个需要重写的函数是 flash_init, flash_erase,write_hword；
4、SST39VF1601：
FS2410 板 nor flash 型号是 SST39VF1601，根据 data sheet,其主要特性如下：
16bit 字为访问单位。2MBTYE 大小。
sector 大小 2kword=4KB,block 大小 32Kword=64KB;这里我按 block 为单位管理 flash,即 flash_info 结构体变量中的 sector_count 是 block 数，起始地址表保存也是所有 block 的起始地址。
SST Manufacturer ID = 00BFH ;
SST39VF1601 Device ID = 234BH;
软件命令序列如下图。

5、我实现的 flash.c 主要部分：

```
//相关定义：
 # define CFG_FLASH_WORD_SIZE unsigned short  //访问单位为 16b 字
 #define MEM_FLASH_ADDR1  (*(volatile CFG_FLASH_WORD_SIZE *)(CFG_FLASH_BASE + 0x000005555<<1 ))

//命令序列地址 1，由于 2410 地址线 A1 与 SST39VF1601 地址线 A0 连接实现按字访问，因此这个地址要左移 1 位。
 #define MEM_FLASH_ADDR2  (*(volatile CFG_FLASH_WORD_SIZE *)(CFG_FLASH_BASE + 0x000002AAA<<1 ))   //命令序列地址 2
 #define READ_ADDR0 (*(volatile CFG_FLASH_WORD_SIZE *)(CFG_FLASH_BASE + 0x0000))

//flash 信息读取地址 1，A0＝0,其余全为 0
 #define READ_ADDR1 (*(volatile CFG_FLASH_WORD_SIZE *)(CFG_FLASH_BASE + 0x0001<<1))   //flash 信息读取地址 2，A0＝1,其余全为 0
 flash_info_t flash_info[CFG_MAX_FLASH_BANKS]; /* 定义全局变量 flash_info[1]*/

 //flash_init(),我实现的比较简单，因为是与板子严重依赖的，只要检测到的信息与板子提供的已知信息符合就 OK。
 ulong flash_init (void)
 {
  int i;

  CFG_FLASH_WORD_SIZE value;
  flash_info_t *info;
  for (i = 0; i < CFG_MAX_FLASH_BANKS; i++)
  {
   flash_info[i].flash_id=FLASH_UNKNOWN;
```

```
}
info=(flash_info_t *)(&flash_info[0]);

 //进入读 ID 状态，读 MAN ID 和 device id
 MEM_FLASH_ADDR1=(CFG_FLASH_WORD_SIZE)(0x00AA);
 MEM_FLASH_ADDR2=(CFG_FLASH_WORD_SIZE)(0x0055);
 MEM_FLASH_ADDR1=(CFG_FLASH_WORD_SIZE)(0x0090);

 value=READ_ADDR0;   //read Manufacturer ID

 if(value==(CFG_FLASH_WORD_SIZE)SST_MANUFACT)
  info->flash_id = FLASH_MAN_SST;
 else
  {
   panic("NOT expected FLASH FOUND!\n");return 0;
  }
 value=READ_ADDR1;   //read device ID

 if(value==(CFG_FLASH_WORD_SIZE)SST_ID_xF1601)
  {
   info->flash_id += FLASH_SST1601;
     info->sector_count = 32;   //32 block
     info->size = 0x00200000; // 2M=32*64K
  }
 else
  {
   panic("NOT expected FLASH FOUND!\n");return 0;
  }

 //建立 sector 起始地址表。
if ((info->flash_id & FLASH_VENDMASK) == FLASH_MAN_SST )
{
  for (i = 0; i < info->sector_count; i++)
 info->start[i] = CFG_FLASH_BASE + (i * 0x00010000);
}

 //设置 sector 保护信息，对于 SST 生产的 FLASH，全部设为 0。
for (i = 0; i < info->sector_count; i++)
{
 if((info->flash_id & FLASH_VENDMASK) == FLASH_MAN_SST)
   info->protect[i] = 0;
}

 //结束读 ID 状态:
*((CFG_FLASH_WORD_SIZE *)&info->start[0])= (CFG_FLASH_WORD_SIZE)0x00F0;

//设置保护，将 u-boot 镜像和环境参数所在的 block 的 proctect 标志置 1
flash_protect (FLAG_PROTECT_SET,
       CFG_FLASH_BASE,
       CFG_FLASH_BASE + monitor_flash_len - 1,
       &flash_info[0]);

flash_protect (FLAG_PROTECT_SET,
       CFG_ENV_ADDR,
       CFG_ENV_ADDR + CFG_ENV_SIZE - 1, &flash_info[0]);
return info->size;
}
```

//flash_erase 实现

这里给出修改的部分，s_first，s_last 是要擦除的 block 的起始和终止 block 号.对于 protect[]置位的 block 不进行擦除。

擦除一个 block 命令时序按照上面图示的 Block-Erase 进行。

```
for (sect = s_first; sect<=s_last; sect++)
  {
   if (info->protect[sect] == 0)
   { /* not protected */
      addr = (CFG_FLASH_WORD_SIZE *)(info->start[sect]);
      if ((info->flash_id & FLASH_VENDMASK) == FLASH_MAN_SST)
       {
      MEM_FLASH_ADDR1 = (CFG_FLASH_WORD_SIZE)0x00AA;
      MEM_FLASH_ADDR2 = (CFG_FLASH_WORD_SIZE)0x0055;
      MEM_FLASH_ADDR1 = (CFG_FLASH_WORD_SIZE)0x0080;
      MEM_FLASH_ADDR1 = (CFG_FLASH_WORD_SIZE)0x00AA;
      MEM_FLASH_ADDR2 = (CFG_FLASH_WORD_SIZE)0x0055;
      addr[0] = (CFG_FLASH_WORD_SIZE)0x0050;  /* block erase */
      for (i=0; i<50; i++)
        udelay(1000);  /* wait 1 ms */
       }
    else
     {
      break;
     }
    }
  }
  .........
 start = get_timer (0);       //在指定时间内不能完成为超时。
 last  = start;
 addr = (CFG_FLASH_WORD_SIZE *)(info->start[l_sect]);//查询 DQ7 是否为 1，DQ7=1 表明擦除完毕
 while ((addr[0] & (CFG_FLASH_WORD_SIZE)0x0080) != (CFG_FLASH_WORD_SIZE)0x0080) {
  if ((now = get_timer(start)) > CFG_FLASH_ERASE_TOUT) {
   printf ("Timeout\n");
   return 1;
 }
 .................
```

//write_word 操作，这个函数由 write_buff 一调用，完成写入一个 word 的操作，其操作命令序列由上图中 Word-Program 指定。

```
 static int write_word (flash_info_t *info, ulong dest, ulong data)
 {
  volatile CFG_FLASH_WORD_SIZE *dest2 = (CFG_FLASH_WORD_SIZE *)dest;
  volatile CFG_FLASH_WORD_SIZE *data2 = (CFG_FLASH_WORD_SIZE *)&data;
  ulong start;
  int flag;
  int i;

  /* Check if Flash is (sufficiently) erased */
  if ((*((volatile ulong *)dest) & data) != data) {
   return (2);
  }
  /* Disable interrupts which might cause a timeout here */
  flag = disable_interrupts();

  for (i=0; i<4/sizeof(CFG_FLASH_WORD_SIZE); i++)
    {
     MEM_FLASH_ADDR1 = (CFG_FLASH_WORD_SIZE)0x00AA;
     MEM_FLASH_ADDR2 = (CFG_FLASH_WORD_SIZE)0x0055;
```

```
    MEM_FLASH_ADDR1 = (CFG_FLASH_WORD_SIZE)0x00A0;

    dest2[i] = data2[i];

    /* re-enable interrupts if necessary */
    if (flag)
      enable_interrupts();

    /* data polling for D7 */
    start = get_timer (0);
    while ((dest2[i] & (CFG_FLASH_WORD_SIZE)0x0080) !=
    (data2[i] & (CFG_FLASH_WORD_SIZE)0x0080)) {
      if (get_timer(start) > CFG_FLASH_WRITE_TOUT) {
  return (1);
      }
    }
   }
 return (0);
}
```

这些代码在与 nor flash 相关的命令中都会间接被调用。所以 u-boot 可移植性的另一个方面就是规定一些函数调用接口和全局变量，这些函数的实现是硬件相关的，移植时只需要实现这些函数。

而全局变量是具体硬件无关的。u-boot 在通用目录中实现其余与硬件无关的函数，这些函数就只与全局变量和函数接口打交道了。 通过编译选项设置来灵活控制是否需要编译通用部分。

6、增加从 Nand 启动的代码：

FS2410 板有跳线，跳线短路时从 NAND 启动，否则从 NOR 启动。根据 FS2410 BIOS 源码，我修改了 start.s 加入了可以从两种 FLASH 中启动 u-boot 的

代码。原理在于：在重定位之前先读 BWSCON 寄存器，判断 OM0 位是 0（有跳线，NAND 启动）还是 1（无跳线，NOR 启动），采取不同的重定位代码

分别从 nand 或 nor 中拷贝 u-boot 镜像到 RAM 中。这里面也有问题，比如从 Nand 启动后，nor flash 的初始化代码和与它相关的命令都是不能使用的。

这里我采用比较简单的方法，定义一个全局变量标志_boot_flash 保存当前启动 FLASH 标志，_boot_flash=0 则表明是 NOR 启动，否则是从 NAND。

在每个与 nor flash 相关的命令执行函数一开始就判断这个变量，如果为 1 立即返回。flash_init()也必须放在这个 if(!_boot_flash)条件中。

这里方法比较笨，主要是为了能在跳线处于任意状态时都能启动 u-boot。

修改后的 start.s 如下。

```
.......
 //修改 1
 .globl _boot_flash
 _boot_flash:  //定义全局标志变量，0:NOR FLASH 启动，1：NAND FLASH 启动。
 .word 0x00000000
.........

 ///修改 2:

 ldr r0,=BWSCON
 ldr r0,[r0]
 ands r0,r0,#6
 beq nand_boot   //OM0=0,有跳线，从 Nand 启动。nand_boot 在后面定义。
 ............

 //修改 4,这里在全局变量_boot_flash 中设置当前启动 flash 设备是 NOR 还是 NAND
 //这里已经完成搬运到 RAM 的工作，即将跳转到 RAM 中_start_armboot 函数中执行。
 adr r1,_boot_flash //取_boot_flash 的当前地址，这时还在 NOR FLASH 或者 NAND 4KB 缓冲中。
 ldr r2,_TEXT_BASE
```

```
 add r1,r1,r2   //得到_boot_flash 重定位后的地址，这个地址在 RAM 中。
 ldr r0,=BWSCON
 ldr r0,[r0]
 ands r0,r0,#6   //
 mov r2,#0x00000001
 streq r2,[r1]   //如果当前是从 NAND 启动，置_boot_flash 为 1

 ldr pc, _start_armboot

_start_armboot: .word start_armboot

.....

//////// 修改 4，从 NAND 拷贝 U-boot 镜像（最大 128KB），这段代码由 fs2410 BIOS 修改得来。
nand_boot:
  mov r5, #NFCONF
  ldr r0, =(1<<15)|(1<<12)|(1<<11)|(7<<8)|(7<<4)|(7)
  str r0, [r5]

  bl ReadNandID
  mov r6, #0
  ldr r0, =0xec73
  cmp r5, r0
  beq x1
  ldr r0, =0xec75
  cmp r5, r0
  beq x1
  mov r6, #1
 x1:
  bl ReadNandStatus

  mov r8, #0       //r8 是 PAGE 数变量
  ldr r9, _TEXT_BASE   //r9 指向 u-boot 在 RAM 中的起始地址。
 x2:
  ands r0, r8, #0x1f
  bne  x3      //此处意思在于页数是 32 的整数倍的时候才进行一次坏块检查  1 block=32 pages，否则直接读取页面。
  mov  r0, r8
  bl  CheckBadBlk   //检查坏块返回值非 0 表明当前块不是坏块。
  cmp  r0, #0
  addne r8, r8, #32   //如果当前块坏了，跳过读取操作。 1 block=32 pages
  bne  x4
 x3:
  mov r0, r8
  mov r1, r9
  bl ReadNandPage  //读取一页(512B)
  add r9, r9, #512
  add r8, r8, #1
 x4:
  cmp r8, #256    //一共读取 256*512=128KB。
  bcc x2

  mov r5, #NFCONF   //DsNandFlash
  ldr r0, [r5]
  and r0, r0, #~0x8000
  str r0, [r5]

  adr lr,stack_setup //注意这里直接跳转到 stack_setup 中执行
```

```
 mov pc,lr
///
/***********************************************
*
*Nand basic functions:
***********************************************
*/

//读取 Nand 的 ID 号，返回值在 r5 中
ReadNandID:
 mov     r7,#NFCONF
 ldr     r0,[r7,#0]  //NFChipEn();
 bic     r0,r0,#0x800
 str     r0,[r7,#0]
 mov     r0,#0x90  //WrNFCmd(RdIDCMD);
 strb    r0,[r7,#4]
 mov     r4,#0   //WrNFAddr(0);
 strb    r4,[r7,#8]
y1:          //while(NFIsBusy());
 ldr     r0,[r7,#0x10]
 tst     r0,#1
 beq     y1
 ldrb    r0,[r7,#0xc] //id  = RdNFDat()<<8;
 mov     r0,r0,lsl #8
 ldrb    r1,[r7,#0xc] //id |= RdNFDat();
 orr     r5,r1,r0
 ldr     r0,[r7,#0]  //NFChipDs();
 orr     r0,r0,#0x800
 str     r0,[r7,#0]
 mov   pc,lr

//读取 Nand 状态,返回值在 r1,此处没有用到返回值。

ReadNandStatus:
 mov   r7,#NFCONF
 ldr     r0,[r7,#0]   //NFChipEn();
 bic     r0,r0,#0x800
 str     r0,[r7,#0]
 mov     r0,#0x70     //WrNFCmd(QUERYCMD);
 strb    r0,[r7,#4]
 ldrb    r1,[r7,#0xc]  //r1 = RdNFDat();
 ldr     r0,[r7,#0]   //NFChipDs();
 orr     r0,r0,#0x800
 str     r0,[r7,#0]
 mov   pc,lr

//等待 Nand 内部操作完毕
WaitNandBusy:
 mov     r0,#0x70  //WrNFCmd(QUERYCMD);
 mov     r1,#NFCONF
 strb    r0,[r1,#4]
z1:             //while(!(RdNFDat()&0x40));
 ldrb    r0,[r1,#0xc]
 tst     r0,#0x40
 beq     z1
 mov     r0,#0   //WrNFCmd(READCMD0);
 strb    r0,[r1,#4]
 mov     pc,lr
```

```
//检查坏 block:
CheckBadBlk:
 mov     r7, lr
 mov     r5, #NFCONF

 bic     r0, r0, #0x1f   //addr &= ~0x1f;
 ldr     r1,[r5,#0]  //NFChipEn()
 bic     r1,r1,#0x800
 str     r1,[r5,#0]

 mov     r1,#0x50    //WrNFCmd(READCMD2)
 strb    r1,[r5,#4]
 mov     r1, #6
 strb    r1,[r5,#8]  //WrNFAddr(6)
 strb    r0,[r5,#8]  //WrNFAddr(addr)
 mov     r1,r0,lsr #8 //WrNFAddr(addr>>8)
 strb    r1,[r5,#8]
 cmp     r6,#0     //if(NandAddr)
 movne    r0,r0,lsr #16 //WrNFAddr(addr>>16)
 strneb   r0,[r5,#8]

 bl  WaitNandBusy //WaitNFBusy()

 ldrb r0, [r5,#0xc] //RdNFDat()
 sub  r0, r0, #0xff

 mov     r1,#0   //WrNFCmd(READCMD0)
 strb    r1,[r5,#4]

 ldr     r1,[r5,#0]  //NFChipDs()
 orr     r1,r1,#0x800
 str     r1,[r5,#0]

 mov  pc, r7

ReadNandPage:
 mov     r7,lr
 mov     r4,r1
 mov     r5,#NFCONF

 ldr     r1,[r5,#0]  //NFChipEn()
 bic     r1,r1,#0x800
 str     r1,[r5,#0]

 mov     r1,#0   //WrNFCmd(READCMD0)
 strb    r1,[r5,#4]
 strb    r1,[r5,#8]  //WrNFAddr(0)
 strb    r0,[r5,#8]  //WrNFAddr(addr)
 mov     r1,r0,lsr #8 //WrNFAddr(addr>>8)
 strb    r1,[r5,#8]
 cmp     r6,#0   //if(NandAddr)
 movne    r0,r0,lsr #16 //WrNFAddr(addr>>16)
 strneb   r0,[r5,#8]

 ldr     r0,[r5,#0]  //InitEcc()
 orr     r0,r0,#0x1000
 str     r0,[r5,#0]
```

```
  bl      WaitNandBusy //WaitNFBusy()

  mov     r0,#0   //for(i=0; i<512; i++)
 r1:
  ldrb    r1,[r5,#0xc] //buf[i] = RdNFDat()
  strb    r1,[r4,r0]
  add     r0,r0,#1
  bic     r0,r0,#0x10000
  cmp     r0,#0x200
  bcc     r1

  ldr     r0,[r5,#0]  //NFChipDs()
  orr     r0,r0,#0x800
  str     r0,[r5,#0]

  mov   pc,r7
```

关于 nand 命令，我尝试打开 CFG_CMD_NAND 选项，并定义

```
  #define CFG_MAX_NAND_DEVICE 1
  #define MAX_NAND_CHIPS 1
  #define CFG_NAND_BASE 0x4e000000
```

添加 boar_nand_init()定义(空实现)。但是连接时出现问题，原因是 u-boot 使用的是软浮点，而我的交叉编译 arm-linux-gcc 是硬件浮点。

看过一些解决方法，比较麻烦，还没有解决这个问题，希望好心的高手指点。不过我比较纳闷，u-boot 在 nand 部分哪里会用到浮点运算呢?

7、添加网络命令。

我尝试使用 ping 命令，其余的命令暂时不考虑。

在 common/cmd_net 中，首先有条件编译 #if (CONFIG_COMMANDS & CFG_CMD_NET)，然后在命令函数 do_ping(...)定义之前有条件编译判断

#if (CONFIG_COMMANDS & CFG_CMD_PING) 。所以在 include/cofig/fs2410.h 中必须打开这两个命令选项。

```
  #define CONFIG_COMMANDS \
   (CONFIG_CMD_DFL  | \
   CFG_CMD_CACHE  | \
   CFG_CMD_REGINFO  | \
   CFG_CMD_DATE  | \
   CFG_CMD_NET | \ //
   CFG_CMD_PING |\ //
   CFG_CMD_ELF)
```

并且设定 IP:192.168.0.12。

至此，整个移植过程已经完成。编译连接生成 u-boot.bin，烧到 nand 和 nor 上都能顺利启动 u-boot,使用 ping 命令时出现问题，

发现 ping 自己的主机竟然超时，还以为是程序出了问题，后来才发现是 windows 防火墙的问题。关闭防火墙就能 PING 通了。

总体来说，u-boot 是一个很特殊的程序，代码庞大，功能强大，自成体系。为了在不同的 CPU，ARCH，BOARD 上移植进行了很多灵活的设计。

在 u-boot 的移植过程中学到很多东西，尤其是程序设计方法方面真的是大开了眼界。u-boot 在代码级可移植性和底层程序开发技术上给人很好的启发。

很多东西没有搞明白，尤其是 u-boot 最重要的功能--引导 OS 这部分还没有涉及。linux 内核还没入门呢，路漫漫其修远兮，吾将上下而求索。

没有 IDE 环境看 u-boot 这种 makefile 工程很费劲，我用 UltraEdit 干了这件事，后来才发现可以使用 source insight 这个软件。。。。。。。。这些工作都是自己学习过程的总结，谬误之处在所难免，请高手不吝指正。。