目录

u-boot-1.1.6 之 cpu/arm920t/start.s 分析

```
/*
 *  armboot - Startup Code for ARM920 CPU-core
 *
 *  Copyright (c) 2001 Marius Gr 鰓 er <mag@sysgo.de>
 *  Copyright (c) 2002 Alex Z 黳 ke <azu@sysgo.de>
 *  Copyright (c) 2002 Gary Jennejohn <gj@denx.de>
 *
 * See file CREDITS for list of people who contributed to this
 * project.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of
 * the License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
```

```
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston,
 * MA 02111-1307 USA
 */

#include <config.h>
#include <version.h>

/*
 *************************************************************************
 *
 * Jump vector table as in table 3.1 in [1]
 *
 *************************************************************************
 */
 //global 声明一个符号可被其他文档引用，相当于声明了一个全局变量，.globl 和.global 相同。
 //该部分为处理器的异常处理向量表。地址范围为 0x0000 0000 ~ 0x0000 0020,刚好 8 条指令。
.globl _start      //u-boot 启动入口
_start: b       reset    //复位向量并且跳转到 reset
 ldr pc, _undefined_instruction
 ldr pc, _software_interrupt
 ldr pc, _prefetch_abort
 ldr pc, _data_abort
 ldr pc, _not_used
 ldr pc, _irq     //中断向量
 ldr pc, _fiq     //中断向量

 // .word 伪操作用于分配一段字内存单元（分配的单元都是字对齐的），并用伪操作中的 expr 初始化。.long 和.int 作用与之相同。
_undefined_instruction: .word undefined_instruction
_software_interrupt: .word software_interrupt
_prefetch_abort: .word prefetch_abort
_data_abort:  .word data_abort
_not_used:  .word not_used
_irq:   .word irq
_fiq:   .word fiq
 // .align 伪操作用于表示对齐方式：通过添加填充字节使当前位置满足一定的对齐方式。.balign 的作用同.align。
 // .align {alignment} {,fill} {,max}
 //  其中：alignment 用于指定对齐方式，可能的取值为 2 的次幂，缺省为 4。fill 是填充内容，缺省用 0 填充。max 是填充字节数最大值，假如填充字节数超过 max,
 //  就不进行对齐,例如：
```

```
// .align 4 /* 指定对齐方式为字对齐 */
.balignl 16,0xdeadbeef

/*
 *************************************************************************
*************
 *
 * Startup Code (reset vector)
 *
 * do important init only if we don't start from memory!
 * relocate armboot to ram
 * setup stack
 * jump to second stage
 *
 *************************************************************************
*************
 */

// TEXT_BASE 在研发板相关的目录中的 config.mk 文档中定义，他定义了
// 代码在运行时所在的地址，那么_TEXT_BASE 中保存了这个地址
_TEXT_BASE:
 .word TEXT_BASE

// 声明 _armboot_start 并用 _start 来进行初始化，在 board/u-boot.lds 中定义。
.globl _armboot_start
_armboot_start:
 .word _start
/*
 * These are defined in the board-specific linker script.
 */
 // 声明_bss_start 并用__bss_start 来初始化，其中__bss_start 定义在和板相关的 u-boot.lds 中。
// _bss_start 保存的是__bss_start 这个标号所在的地址，这里涉及到当前代码所在
// 的地址不是编译时的地址的情况，这里直接取得该标号对应的地址，不受编译时
// 地址的影响. _bss_end 也是同样的道理.
.globl _bss_start
_bss_start:
 .word __bss_start
 // 同上
.globl _bss_end
_bss_end:
 .word _end
#ifdef CONFIG_USE_IRQ
/* IRQ stack memory (calculated at run-time) */
.globl IRQ_STACK_START
```

```
IRQ_STACK_START:
 .word 0x0badc0de
/* IRQ stack memory (calculated at run-time) */
.globl FIQ_STACK_START
FIQ_STACK_START:
 .word 0x0badc0de
#endif

/*
 * the actual reset code
 */
//  MRS {} Rd,CPSR|SPSR 将 CPSR|SPSR 传送到 Rd
//  使用这两条指令将状态寄存器传送到一般寄存器，只修改必要的位，再将结果传送回状态寄存器，这样能够最好地完成对 CRSP 或 SPSR 的修改
//  MSR {} CPSR_|SPSR_,Rm 或是 MSR {} CPSR_f|SPSR_f,#
//  MRS 和 MSR 配合使用，作为更新 PSR 的“读取——修改——写回”序列的一部分
//   bic r0,r1,r2  ;r0:=r1 and not r2
//   orr ro,r1,r2  ;r0:=r1 or r2
//  这几条指令执行完毕后，进入 SVC32 模式，该模式主要用来处理软件中断(SWI)
reset:
 /*
  * set the cpu to SVC32 mode
  */
 mrs r0,cpsr    //将 CPSR 状态寄存器读取，保存到 R0 中
 bic r0,r0,#0x1f
 orr r0,r0,#0xd3
 msr cpsr,r0    //将 R0 写入状态寄存器中
/* turn off the watchdog */
 //关闭看门狗
#if defined(CONFIG_S3C2400)
# define pWTCON  0x15300000
# define INTMSK  0x14400008 /* Interupt-Controller base addresses */
# define CLKDIVN 0x14800014 /* clock divisor register */
#elif defined(CONFIG_S3C2410)
# define pWTCON  0x53000000
# define INTMSK  0x4A000008 /* Interupt-Controller base addresses */
# define INTSUBMSK 0x4A00001C
# define CLKDIVN 0x4C000014 /* clock divisor register */
#endif
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
 ldr     r0, =pWTCON
 mov     r1, #0x0
 str     r1, [r0]
 /*
```

```
 * mask all IRQs by setting all bits in the INTMR - default
 */
//关闭所有中断
 mov r1, #0xffffffff
 ldr r0, =INTMSK
 str r1, [r0]
# if defined(CONFIG_S3C2410)
 ldr r1, =0x3ff
 ldr r0, =INTSUBMSK
 str r1, [r0]
# endif
 /* FCLK:HCLK:PCLK = 1:2:4 */
 /* default FCLK is 120 MHz ! */
 ldr r0, =CLKDIVN
 mov r1, #3
 str r1, [r0]
#endif /* CONFIG_S3C2400 || CONFIG_S3C2410 */
 /*
 * we do sys-critical inits only at reboot,
 * not when booting from ram!
 */
 // B 转移指令，跳转到指令中指定的目的地址
 // BL 带链接的转移指令，像 B 相同跳转并把转移后面紧接的一条指令地址保存到链接寄存器 LR（R14）中，以此来完成子程式的调用
 // 该语句首先调用 cpu_init_crit 进行 CPU 的初始化，并把下一条指令的地址保存在 LR 中，以使得执行完后能够正常返回。
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
 bl cpu_init_crit
#endif
#ifndef CONFIG_SKIP_RELOCATE_UBOOT
 //调试阶段的代码是直接在 RAM 中运行的，而最后需要把这些代码固化到 Flash 中，因此 U-Boot 需要自己从 Flash 转移到
 //RAM 中运行，这也是重定向的目的所在。
 //通过 adr 指令得到当前代码的地址信息：假如 U-boot 是从 RAM 开始运行，则从 adr,r0,_start 得到的地址信息为
 //r0=_start=_TEXT_BASE=TEXT_BASE=0xa3000000;假如 U-boot 从 Flash 开始运行，即从处理器对应的地址运行，
 //则 r0=0x0000,这时将会执行 copy_loop 标识的那段代码了。
 // _TEXT_BASE 定义在 board/smdk2410/config.mk 中
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
```

```
 //重新定位代码
//声明_bss_start 并用__bss_start 来初始化，其中__bss_start 定义在和板相关的 u-boot.lds 中
 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 */
 /* Set up the stack          */
 //初始化堆栈
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  /* bdinfo                      */
#ifdef CONFIG_USE_IRQ
 sub r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
#endif
 sub sp, r0, #12     /* leave 3 words for abort-stack    */
clear_bss:
 ldr r0, _bss_start     /* find start of bss segment       */
 ldr r1, _bss_end     /* stop here                     */
 mov  r2, #0x00000000   /* clear                         */
clbss_l:str r2, [r0]     /* clear loop...                 */
 add r0, r0, #4
 cmp r0, r1
 ble clbss_l
#if 0
 /* try doing this stuff after the relocation */
 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, =INTMR
 str r1, [r0]
 /* FCLK:HCLK:PCLK = 1:2:4 */
 /* default FCLK is 120 MHz ! */
 ldr r0, =CLKDIVN
```

```
 mov r1, #3
 str r1, [r0]
 /* END stuff after relocation */
#endif
 //跳转到 start_armboot 函数入口，_start_armboot 字保存函数入口指针
//start_armboot 函数在 lib_arm/board.c 中实现
ldr pc, _start_armboot
_start_armboot: .word start_armboot

/*
 ************************************************************************
*************
 *
 * CPU_init_critical registers
 *
 * setup important registers
 * setup memory timing
 *
 ************************************************************************
*************
 */

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
 /*
  * flush v4 I/D caches
  */
 //初始化 CACHE
 mov r0, #0
 mcr p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache */
 mcr p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
 /*
  * disable MMU stuff and caches
  */
  //关闭 MMU 和 CACHE
 mrc p15, 0, r0, c1, c0, 0
 bic r0, r0, #0x00002300 @ clear bits 13, 9:8 (--V- --RS)
 bic r0, r0, #0x00000087 @ clear bits 7, 2:0 (B--- -CAM)
 orr r0, r0, #0x00000002 @ set bit 2 (A) Align
 orr r0, r0, #0x00001000 @ set bit 12 (I) I-Cache
 mcr p15, 0, r0, c1, c0, 0
 /*
  * before relocating, we have to setup RAM timing
  * because memory timing is board-dependend, you will
```

```
 * find a lowlevel_init.S in your board directory.
 */
//初始化 RAM 时钟。因为内存时钟是依赖开发板硬件的，所以在 board 的相应目录下可以找到 memsetup.s 文件
 mov ip, lr
 bl lowlevel_init  //lowlevel_init 子程序在 board/smdk2410/memsetup.s 中实现
 mov lr, ip
 mov pc, lr
#endif /* CONFIG_SKIP_LOWLEVEL_INIT */
/*
 *************************************************************************
 *
 * Interrupt handling
 *
 *************************************************************************
 */
@
@ IRQ stack frame.
@
#define S_FRAME_SIZE 72
#define S_OLD_R0 68
#define S_PSR  64
#define S_PC  60
#define S_LR  56
#define S_SP  52
#define S_IP  48
#define S_FP  44
#define S_R10  40
#define S_R9  36
#define S_R8  32
#define S_R7  28
#define S_R6  24
#define S_R5  20
#define S_R4  16
#define S_R3  12
#define S_R2  8
#define S_R1  4
#define S_R0  0
#define MODE_SVC 0x13
#define I_BIT  0x80
/*
 * use bad_save_user_regs for abort/prefetch/undef/swi ...
```

```
 * use irq_save_user_regs / irq_restore_user_regs for IRQ/FIQ handling
 */
.macro bad_save_user_regs
sub sp, sp, #S_FRAME_SIZE
stmia sp, {r0 - r12}   @ Calling r0-r12
ldr r2, _armboot_start
sub r2, r2, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
sub r2, r2, #(CFG_GBL_DATA_SIZE+8)  @ set base 2 words into abort stack
ldmia r2, {r2 - r3}   @ get pc, cpsr
add r0, sp, #S_FRAME_SIZE  @ restore sp_SVC
add r5, sp, #S_SP
mov r1, lr
stmia r5, {r0 - r3}   @ save sp_SVC, lr_SVC, pc, cpsr
mov r0, sp
.endm
.macro irq_save_user_regs
sub sp, sp, #S_FRAME_SIZE
stmia sp, {r0 - r12}   @ Calling r0-r12
add     r8, sp, #S_PC
stmdb   r8, {sp, lr}^                  @ Calling SP, LR
str     lr, [r8, #0]                  @ Save calling PC
mrs     r6, spsr
str     r6, [r8, #4]                   @ Save CPSR
str     r0, [r8, #8]                   @ Save OLD_R0
mov r0, sp
.endm
.macro irq_restore_user_regs
ldmia sp, {r0 - lr}^   @ Calling r0 - lr
mov r0, r0
ldr lr, [sp, #S_PC]   @ Get PC
add sp, sp, #S_FRAME_SIZE
subs pc, lr, #4   @ return & move spsr_svc into cpsr
.endm
.macro get_bad_stack
ldr r13, _armboot_start  @ setup our mode stack
sub r13, r13, #(CONFIG_STACKSIZE+CFG_MALLOC_LEN)
sub r13, r13, #(CFG_GBL_DATA_SIZE+8) @ reserved a couple spots in abort
stack
str lr, [r13]   @ save caller lr / spsr
mrs lr, spsr
str     lr, [r13, #4]
mov r13, #MODE_SVC   @ prepare SVC-Mode
@ msr spsr_c, r13
msr spsr, r13
```

```
 mov lr, pc
 movs pc, lr
 .endm
 .macro get_irq_stack   @ setup IRQ stack
 ldr sp, IRQ_STACK_START
 .endm
 .macro get_fiq_stack   @ setup FIQ stack
 ldr sp, FIQ_STACK_START
 .endm
/*
 * exception handlers
 */
//以下都是中断处理函数,具体实现在 lib_arm 目录下 interrupts.c
 .align  5
undefined_instruction:
 get_bad_stack
 bad_save_user_regs
 bl  do_undefined_instruction
 .align 5
software_interrupt:
 get_bad_stack
 bad_save_user_regs
 bl  do_software_interrupt
 .align 5
prefetch_abort:
 get_bad_stack
 bad_save_user_regs
 bl  do_prefetch_abort
 .align 5
data_abort:
 get_bad_stack
 bad_save_user_regs
 bl  do_data_abort
 .align 5
not_used:
 get_bad_stack
 bad_save_user_regs
 bl  do_not_used
#ifdef CONFIG_USE_IRQ
 .align 5
irq:
 get_irq_stack
 irq_save_user_regs
 bl  do_irq
```

```
 irq_restore_user_regs
 .align 5
fiq:
 get_fiq_stack
 /* someone ought to write a more effiction fiq_save_user_regs */
 irq_save_user_regs
 bl  do_fiq
 irq_restore_user_regs
#else
 .align 5
irq:
 get_bad_stack
 bad_save_user_regs
 bl  do_irq
 .align 5
fiq:
 get_bad_stack
 bad_save_user_regs
 bl  do_fiq
#endif
```

u-boot 中.lds 连接脚本文件的分析

对于.lds **文件，它定义了整个程序编译之后的连接过程，决定了一个可执行程序的各个段的存储位置。虽然现在我还没怎么用它，但感觉还是挺重要的，有必要了解一下。**

先看一下 GNU **官方网站上对**.lds **文件形式的完整描述：**

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
  { contents } >region :phdr =fill
...
}
```

secname和 contents是必须的，其他的都是可选的。下面挑几个常用的看看：

1、secname：**段名**

2、contents：决定哪些内容放在本段，可以是整个目标文件，也可以是目标文件中的某段（代码段、数据段等）

3、start：本段连接（运行）的地址，如果没有使用 AT（ldadr），本段存储的地址也是 start。GNU 网站上说 start 可以用任意一种描述地址的符号来描述。

4、AT（ldadr）：定义本段存储（加载）的地址。

看一个简单的例子：（摘自《2410 完全开发》）

```
/* nand.lds */

SECTIONS {

firtst 0x00000000 : { head.o init.o }

second 0x30000000 : AT(4096) { main.o }

}
```

以上，head.o 放在 0x00000000 地址开始处，init.o 放在 head.o 后面，他们的运行地址也是 0x00000000，即连接和存储地址相同（没有 AT 指定）；main.o 放在 4096（0x1000，是 AT 指定的，存储地址）开始处，但是它的运行地址在 0x30000000，运行之前需要从 0x1000（加载处）复制到 0x30000000（运行处），此过程也就用到了读取 Nand flash。

这就是存储地址和连接（运行）地址的不同，称为加载时域和运行时域，可以在.lds 连接脚本文件中分别指定。

编写好的.lds 文件，在用 arm-linux-ld 连接命令时带-Tfilename 来调用执行，如 arm-linux-ld –Tnand.lds x.o y.o –o xy.o。也用-Ttext 参数直接指定连接地址，如 arm-linux-ld –Ttext 0x30000000 x.o y.o –o xy.o。

既然程序有了两种地址，就涉及到一些跳转指令的区别，这里正好写下来，以后万一忘记了也可查看，以前不少东西没记下来现在忘得差不多了。。。

ARM 汇编中，常有两种跳转方法：b 跳转指令、ldr 指令向 PC 赋值。

我自己经过归纳如下：

（1） b step1 ：b 跳转指令是相对跳转，依赖当前 PC 的值，偏移量是通过该指令本身的 bit[23:0]算出来的，这使得使用 b 指令的程序不依赖于要跳到的代码的位置，只看指令本身。

（2） ldr pc, =step1 ：该指令是从内存中的某个位置（step1）读出数据并赋给 PC，同样依赖当前 PC 的值，但是偏移量是那个位置（step1）的连接地址（运行时的地址），所以可以用它实现从 Flash 到 RAM 的程序跳转。

（3） 此外，有必要回味一下 adr 伪指令，U-boot 中那段 relocate 代码就是通过 adr 实现当前程序是在 RAM 中还是 flash 中。仍然用我当时的注释：

```
relocate: /* 把 U-Boot 重新定位到 RAM */

    adr r0, _start /* r0 是代码的当前位置 */

/* adr 伪指令，汇编器自动通过当前 PC 的值算出 如果执行到_start 时 PC 的值，

放到 r0 中：
```

```
当此段在 flash 中执行时 r0 = _start = 0；当此段在 RAM 中执行时_start = _TEXT_BASE(在 board/smdk2410/config.mk 中指定的值为 0x33F80000，即 u-boot 在把代码拷贝到 RAM 中去执行的代码段的开始) */
    ldr r1, _TEXT_BASE /* 测试判断是从 Flash 启动，还是 RAM */
/* 此句执行的结果 r1 始终是 0x33FF80000，因为此值是又编译器指定的(ads 中设置，或-D 设置编译器参数) */
   cmp r0, r1 /* 比较 r0 和 r1，调试的时候不要执行重定位 */
```

下面，结合 u-boot.lds 看看一个正式的连接脚本文件。这个文件的基本功能还能看明白，虽然上面分析了好多，但其中那些 GNU 风格的符号还是着实让我感到迷惑，好菜啊，怪不得连被 3 家公司鄙视，自己鄙视自己。。。

```
OUTPUT_FORMAT("elf32&shy;littlearm", "elf32&shy;littlearm", "elf32&shy;littlearm")
  ;指定输出可执行文件是 elf 格式,32 位 ARM 指令,小端
OUTPUT_ARCH(arm)
  ;指定输出可执行文件的平台为 ARM
ENTRY(_start)
  ;指定输出可执行文件的起始代码段为_start.
SECTIONS
{
      . = 0x00000000 ; 从 0x0 位置开始
      . = ALIGN(4) ; 代码以 4 字节对齐
      .text : ;指定代码段
      {
        cpu/arm920t/start.o (.text) ; 代码的第一个代码部分
        *(.text) ;其它代码部分
      }
      . = ALIGN(4)
```

```
        .rodata : { *(.rodata) } ;指定只读数据段
        . = ALIGN(4);
        .data : { *(.data) } ;指定读/写数据段
        . = ALIGN(4);
        .got : { *(.got) } ;指定 got 段, got 段式是 uboot 自定义的一个段, 非标准段
        __u_boot_cmd_start = . ;把__u_boot_cmd_start 赋值为当前位置, 即起始位
置
        .u_boot_cmd : { *(.u_boot_cmd) } ;指定 u_boot_cmd 段, uboot 把所有的 u
boot 命令放在该段.
        __u_boot_cmd_end = .;把__u_boot_cmd_end 赋值为当前位置,即结束位置
        . = ALIGN(4);
        __bss_start = .; 把__bss_start 赋值为当前位置,即 bss 段的开始位置
        .bss : { *(.bss) }; 指定 bss 段
        _end = .; 把_end 赋值为当前位置,即 bss 段的结束位置
}
```

分享一篇我总结的 uboot 学习笔记（转）

1. 下面代码是系统启动后 U-boot 上电后运行的第一段代码，他是什么意思？

```
.globl _start
_start:     b       reset
    ldr     pc, _undefined_instruction
    ldr     pc, _software_interrupt
    ldr     pc, _prefetch_abort
    ldr     pc, _data_abort
    ldr     pc, _not_used
    ldr     pc, _irq
    ldr     pc, _fiq
```

```
_undefined_instruction:     .word undefined_instruction
_software_interrupt:     .word software_interrupt
_prefetch_abort:     .word prefetch_abort
_data_abort:         .word data_abort
_not_used:         .word not_used
_irq:              .word irq
_fiq:              .word fiq
    .balignl 16,0xdeadbeef
```

他们是系统定义的异常，一上电程序跳转到 reset **异常处执行相应的汇编指令，下面定义出的都是不同的异常，比如软件发生软中断时，**CPU **就会去执行软中断的指令，这些异常中断在** CUP **中地址是从** 0 **开始，每个异常占** 4 **个字节。**

```
reset:
    /*
    * set the cpu to SVC32 mode
    */
    mrs     r0,cpsr
    bic     r0,r0,#0x1f
    orr     r0,r0,#0xd3
    msr     cpsr,r0
```

操作系统先注册一个总的中断，然后去查是由哪个中断源产生的中断，再去查用户注册的中断表，查出来后就去执行用户定义的用户中断处理函数。

ldr pc, _undefined_instruction **表示把**_undefined_instruction **存放的数值存放到** pc **指针上，**_undefined_instruction: .word undefined_instruction **表示未定义的这个异常是由**.word **来定义的，它表示定义一个字，一个** 32 **位的数，**.word **后面的数表示把该标识的编译地址写入当前地址，标识是不占用任何指令的。把标识存放的数值** copy **到指针** pc **上面，那么标识上存放的值是什么？是由**.word undefined_instruction **来指定的，**pc **就代表你运行代码的地址，她就实现了** CPU **要做一次跳转时的工作。**

什么是编译地址？什么是运行地址？

32 位的处理器，它的每一条指令是 4 个字节，以 4 个字节存储顺序，进行顺序执行，CPU 是顺序执行的，只要没发生什么跳转，它会顺序进行执行，编译器会对每一条指令分配一个编译地址，这是编译器分配的，在编译过程中分配的地址，我们称之为编译地址。

运行地址是指，程序指令真正运行的地址，是由用户指定的，用户将运行地址烧录到哪里，哪里就是运行的地址。比如有一个指令的编译地址是 0x5，实际运行的地址是 0x200，如果

用户将指令烧到 0x200 上，那么这条指令的运行地址就是 0x200，当编译地址和运行地址不同的时候会出现什么结果？结果是不能跳转，编译后会产生跳转地址，如果实际地址和编译后产生的地址不相等，那么就不能跳转。C 语言编译地址都希望把编译地址和实际运行地址放在一起的，但是汇编代码因为不需要做 C 语言到汇编的转换，可以人为的去写地址，所以直接写的就是他的运行地址，这就是为什么任何 bootloader 刚开始会有一段汇编代码，因为起始代码编译地址和实际地址不相等，这段代码和汇编无关，跳转用的运行地址。编译地址和运行地址如何来算呢？假如有两个编译地址 a=0x10，b=0x7，b 的运行地址是 0x300，那么 a 的运行地址就是 b 的运行地址加上两者编译地址的差值，a-b=0x10-0x7=0x3，a 的运行地址就是 0x300+0x3=0x303。

假设 uboot 上两条指令的编译地址为 a=0x33000007 和 b=0x33000001，这两条指令都落在 bank6 上，现在要计算出他们对应的运行地址，要找出运行地址的始地址，这个是由用户烧录进去的，假设运行地址的首地址是 0x0，则 a 的运行地址
为 0x7，b 为 0x1，就是这样算出来的。

为什么要分配编译地址？这样做有什么好处，有什么作用？

比如在函数 a 中定义了函数 b，当执行到函数 b 时要进行指令跳转，要跳转到 b 函数所对应的起始地址上去，编译时，编译器给每条指令都分配了编译地址，如果编译器已经给分配了地址就可以直接进行跳转，查找 b 函数跳转指令所对应的表，进行直接跳转，因为有个编译地址和指令对应的一个表，如果没有分配，编译器就查找不到这个跳转地址，要进行计算，非常麻烦。

什么是相对地址？

以 NOR Flash 为例，NOR FaIsh 是映射到 bank0 上面，SDRAM 是映射到 bank6 上面，uboot 和内核最终是在 SDRAM 上面运行，最开始我们是从 Nor Flash 的零地址开始往后烧录，uboot 中至少有一段代码编译地址和运行地址是不一样的，编译 uboot 或内核时，都会将编译地址放入到 SDRAM 中，他们最终都会在 SDRAM 中执行，刚开始 uboot 在 Nor Flash 中运行，运行地址是一个低端地址，是 bank0 中的一个地址，但编译地址是 bank6 中的地址，这样就会导致绝对跳转指令执行的失败，所以就引出了相对地址的概念。那么什么是相对地址呢？至少在 bank0 中 uboot 这段代码要知道不能用 b+编译地址这样的方法去跳转指令，因为这段代码的编译地址和运行地址不一样，那如何去做呢？要去计算这个指令运行的真实地址，计算出来后再做跳转，应该是 b+运行地址，不能出现 b+编译地址，而是 b+运行地址，而运行地址是算出来的。

```
_TEXT_BASE:
   .word      TEXT_BASE  //0x33F80000,在 board/config.mk 中
```

这段话表示，用户告诉编译器编译地址的起始地址

Uboot 启动流程

1. 设置 CPU 的启动模式

reset:

//设置 CPU 进入管理模式 即设置相应的 CPSR 程序状态字

```
    /* * set the cpu to SVC32 mode*/
    mrs     r0,cpsr
    bic     r0,r0,#0x1f
    orr     r0,r0,#0xd3
    msr     cpsr,r0
```

2. 关闭看门狗，关闭中断，所谓的喂狗是每隔一段时间给某个寄存器置位而已，在实际中会专门启动一个线程或进程会专门喂狗，当上层软件出现故障时就会停止喂狗，停止喂狗之后，cpu 会自动复位，一般都在外部专门有一个看门狗，做一个外部的电路，不在 cpu 内部使用看门狗，cpu 内部的看门狗是复位的 cpu，当开发板很复杂时，有好几个 cpu 时，就不能完全让板子复位，但我们通常都让整个板子复位。看门狗每隔短时间就会喂狗，问题是在两次喂狗之间的时间间隔内，运行的代码的时间是否够用，两次喂狗之间的代码是否在两次喂狗的时间延迟之内，如果在延迟之外的话，代码还没改完就又进行喂狗，代码永远也改不完。

//关闭看门狗的实际代码

```
#if defined(CONFIG_S3C2400) || defined(CONFIG_S3C2410)
    ldr     r0, =pWTCON //将 pwtcon 寄存器地址赋给 R0
    mov     r1, #0x0 //r1 的内容为 0
    str     r1, [r0]  //将 R1 的内容送到 Ro 寄存器中去
```

3. 屏蔽所有中断，为什么要关中断？中断处理中 ldr pc 是将代码的编译地址放在了指针上，而这段时间还没有搬移代码，所以编译地址上面没有这个代码，如果进行跳转就会跳转到空指针上面

```
    /* * mask all IRQs by setting all bits in the INTMR - default*/
    mov     r1, #0xffffffff  //寄存器中的值全为 11111111111111111111111111111111
    ldr     r0, =INTMSK    //将管理中断的寄存器地址赋给 ro
    str     r1, [r0]       //将全 1 的值赋给 ro 地址中的内容
#if defined(CONFIG_S3C2410)
    ldr     r1, =0x3ff
    ldr     r0, =INTSUBMSK
    str     r1, [r0]
#endif
```

3. 设置时钟分频，为什么要设置时钟？起始可以不设，系统能不能跑起来和频率没有任何关系，频率的设置是要让外围的设备能承受所设置的频率，如果频率过高则会导致 cpu 操作外围设备失败

//设置 CPU 的频率

```
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
ldr     r0, =CLKDIVN
mov     r1, #3
str     r1, [r0]
```

4. 做 bank 的设置

cpu_init_crit:

```
/* flush v4 I/D caches，关闭 catch*/
mov     r0, #0
mcr     p15, 0, r0, c7, c7, 0     /* flush v3/v4 cache */
mcr     p15, 0, r0, c8, c7, 0     /* flush v4 TLB *///协处理器
```

//禁止 MMU

```
/** disable MMU stuff and caches*/
mrc     p15, 0, r0, c1, c0, 0
bic     r0, r0, #0x00002300     @ clear bits 13, 9:8 (--V- --RS)
bic     r0, r0, #0x00000087     @ clear bits 7, 2:0 (B--- -CAM)
orr     r0, r0, #0x00000002     @ set bit 2 (A) Align
orr     r0, r0, #0x00001000     @ set bit 12 (I) I-Cache
mcr     p15, 0, r0, c1, c0, 0  //关闭
```

为什么要关闭 catch 和 MMU 呢？catch 和 MMU 是做什么用的？

Catch 是 cpu 内部的一个 2 级缓存，她的作用是将常用的数据和指令放在 cpu 内部，MMU 是用来做虚实地址转换用的，我们的目的是设置控制的寄存器，寄存器都是实地址，如果既要开启 MMU 又要做虚实地址转换的话，中间还多一步，

先要把实地址转换成虚地址，然后再做设置，但对 uboot 而言就是起到一个简单的初始化的作用和引导操作系统，如果开启 MMU 的话，很麻烦，也没必要，所以关闭 MMU.

说道 catch 就必须提到一个关键字 Volatile，以后在设置寄存器时会经常遇到，他的本质是告诉编译器不要对我的代码进行优化，优化的过程是将常用的代码取出来放到 catch 中，它没有从实际的物理地址去取，它直接从 cpu 的缓存中去取，但常用的代码就是为了感知一

些常用变量的变化，如果正在取数据的时候发生跳变，那么就感觉不到变量的变化了，所以在这种情况下要用 Volatile 关键字告诉编译器不要做优化，每次从实际的物理地址中去取指令，这就是为什么关闭 catch 关闭 MMU。但在 C 语言中是不会关闭 catch 和 MMU 的，会打开，如果编写者要感知外界变化，或变化太快，从 catch 中取数据会有误差，就加一个关键字 Volatile。

5. bl lowlevel_init 下来初始化各个 bank，把各个 bank 设置必须搞清楚，对以后移植复杂的 uboot 有很大帮助

6.设置完毕后拷贝 uboot 代码到 4k 空间，拷贝完毕后执行内存中的 uboot 代码

以上流程基本上适用于所有的 bootloader，这就是 step1 阶段

7. 问题：如果换一块开发板有可能改哪些东西？

首先，cpu 的运行模式，如果需要对 cpu 进行设置那就设置，管看门狗，关中断不用改，时钟有可能要改，如果能正常使用则不用改，关闭 catch 和 MMU 不用改，设置 bank 有可能要改。最后一步拷贝时看地址会不会变，如果变化也要改，执行内存中代码，地址有可能要改。

8. Nor Flash 和 Nand Flash 本质区别就在于是否进行代码拷贝，也就是下面代码所表述：无论是 Nor Flash 还是 Nand Flash，核心思想就是将 uboot 代码搬运到内存中去运行，但是没有拷贝 bss 后面这段代码，只拷贝 bss 前面的代码，bss 代码是放置全局变量的。Bss 段代码是为了清零，拷贝过去再清零重复操作

```
//uboot 代码搬运到 RAM 中去
#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  //flash 中 armboot_start 的起始地址
    ldr     r3, _bss_start  //uboot_bss 的起始地址
    sub     r2, r3, r2      /* r2 <- size of armboot//uboot 实际程序代码的大小          */
    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 */
```

9. 看一下 uboot.lds文件，在 board/smdk2410目录下面，uboot.lds是告诉编译器这些段改怎么划分，GUN编译过的段，最基本的三个段是 RO, RW, ZI，RO表示只读，对应于具体的指代码段，RW是数据段，ZI是归零段，就是全局变量的那段。Uboot代码这么多，如何保证 start.s会第一个执行，编译在最开始呢？就是通过 uboot.lds链接文件进行

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
/*OUTPUT_FORMAT("elf32-arm", "elf32-arm", "elf32-arm")*/
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;  /起始地址

    . = ALIGN(4);   //4字节对齐
    .text      :      //test指代码段，上面 3行标识是不占用任何空间的
    {
      cpu/arm920t/start.o    (.text) /这里把 start.o放在第一位就表示把 start.s
编
译时放到最开始，这就是为什么把 uboot烧到起始地址上它肯定运行的是 start.s
      *(.text)
    }

    . = ALIGN(4);  /前面的 “ .” 代表当前值，是计算一个当前的值，是计算上
面占用的整个空间，再加一个单元就表示它现在的位置
    .rodata : { *(.rodata) }

    . = ALIGN(4);
    .data : { *(.data) }

    . = ALIGN(4);
    .got : { *(.got) }

    . = .;
    __u_boot_cmd_start = .;
```

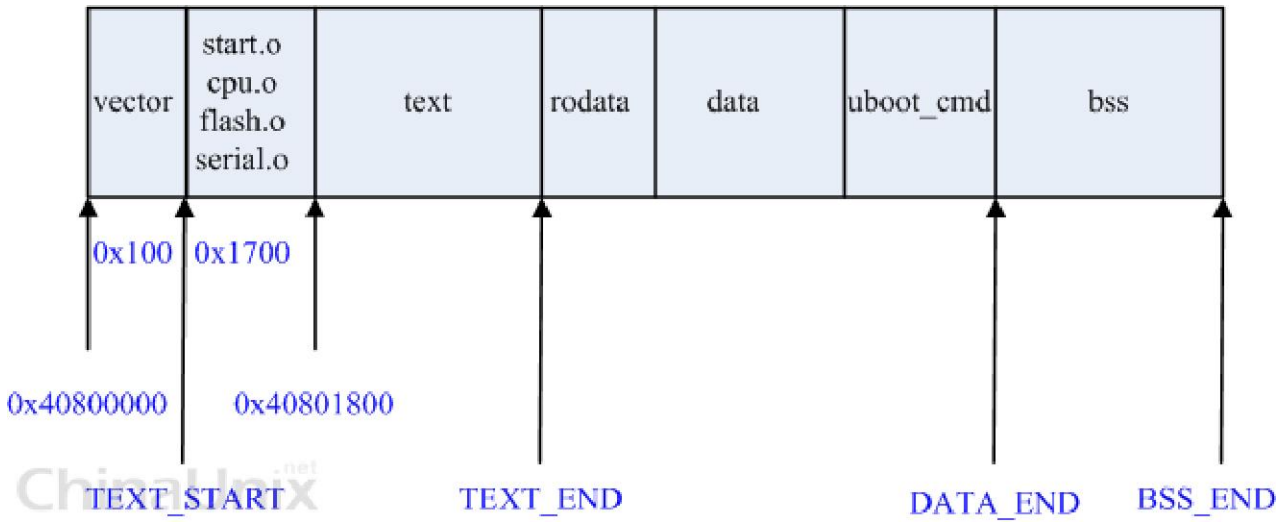
```
.u_boot_cmd : { *(.u_boot_cmd) }
__u_boot_cmd_end = .;

. = ALIGN(4);
__bss_start = .;     //bss表示归零段
.bss : { *(.bss) }
_end = .;
```

回忆一下 GUN在编译代码时的四个步骤，1.预处理，2.编译，3.汇编，4.链接，链接就做的是这个文件的动作，就是将这些文件重新 map一下分配地址。
最后运行的是 _start_armboot:　　.word start_armboot函数跳转到 step2的阶段，这个函数是 uboot中第一个 C代码，也是第一个在内存中运行的代码

U-BOOT 内存布局及启动过程浅析

本文以 ARC600平台的某一实现为例，对 U-BOOT的内存布局和启动方式进行简要的分析。

【内存布局】

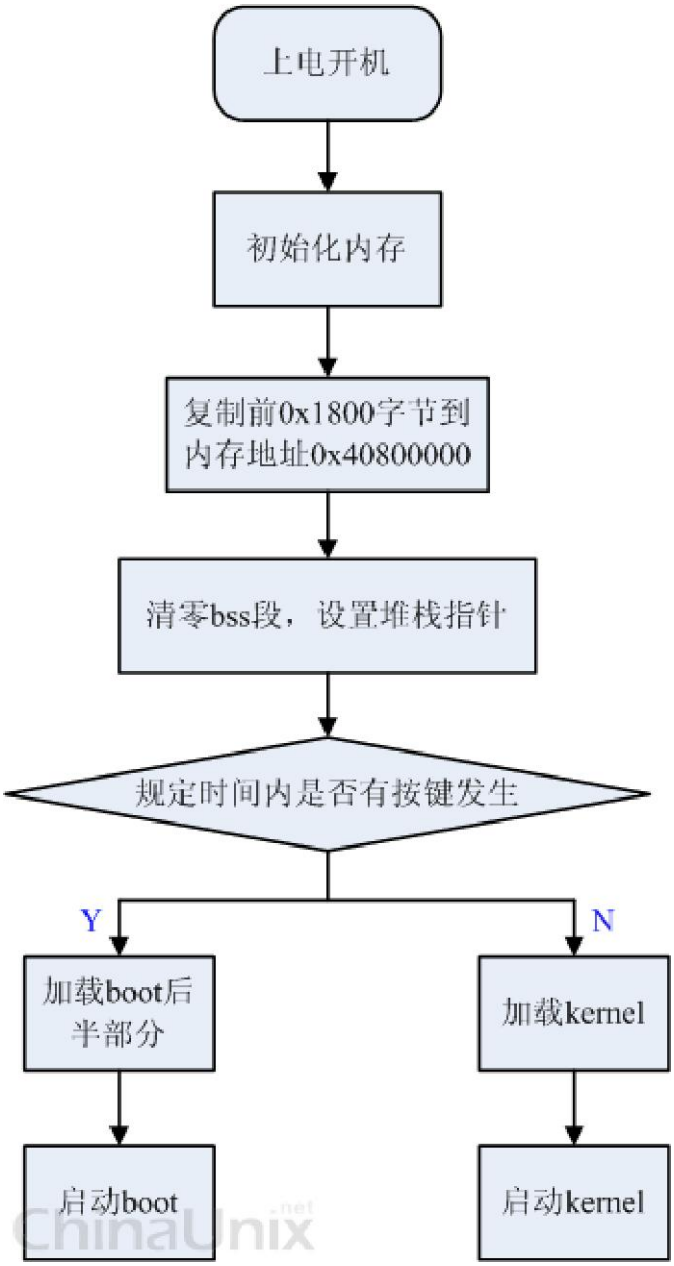


在 ARC600平台，U-BOOT的内存布局图 1所示。

该布局由 board/arc600/u-boot.lds文件定义,在链接的时候生成相应的二进制映像。首先，定义起始地址为 0x40800000，接下来是中断向量表，大小为 256字节，按每个中断向量占用 4个字节的跳转地址算，最多可以有 64个中断向量；第二部分是一些基础性的代码段，它为下一步加载 boot或者 kernel做准备，其大小为 0x1700字节；第三部分是代码段的后半部分，代码段的大部分代码都在这里；第四部分只读数据区；第五部分为可读写数据区；第六部分为 U-BOOT命令代码区；最后一部分为未初始化数据段。

有一点比较疑惑的就是 U-BOOT命令代码区存放的分明是代码，但它却在数据段。内核中会把一些初始化代码放在数据区，因为这些代码只运行一次，放在数据区可以在内核启动后回收该区域内存。但显然 U-BOOT命令不可能只运行一次，为何要把它放在数据段？不解！

【启动过程】



众所周知，U-BOOT是存放在 FLASH上的。系统启动时，CPU会映射 FLASH到它的内存空间（映射一部分、还是全部 FLASH空间？），然后执行 FLASH上的代码。首先，进入 cpu/arc600/start.S中的入口 _start，进行内存初始化，接着把 U-BOOT的前 0x1800字节从 FLASH复制到内存的 0x40800000处，也就是链接时的地址；然后对 bss段进行清零，设置堆栈指针，为运行 C函数做准备；下一步，运行 C函数检测在规定时间内是否有按键发生，如有则加载 boot的后半部分 (0x40801800——DATA_END)并启动 boot，无则加载 kernel并启动 kernel。U-BOOT启动的前半部分流程如图 2所示。

U-BOOT启动的后半部分，会进行 heap、环境变量（env）的初始化，PHY驱动的加载等工作，然后进入一个无限循环开始 shell的运行，shell运行过程中的内存示意如图 3所示。其中，heap和 stack依次排列在 bss段的后面，图中所示的 free area则为 U-BOOT未用到的内存。

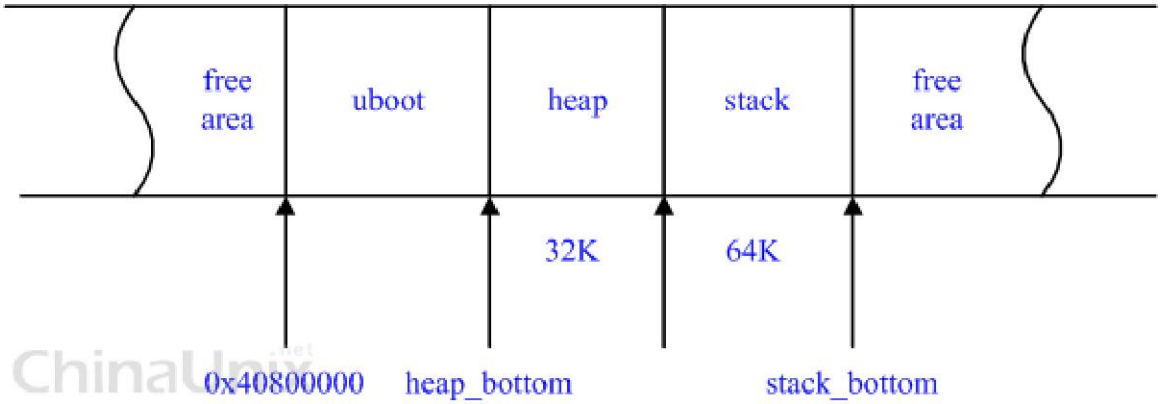


图 3中，heap区域为 malloc()提供内存。在 uClib库中，malloc()是通过 sbrk()或者 mmap()实现的，而 sbrk()和 mmap()是在内核中实现的。U-BOOT作为系统最早运行的程序，没有内核的支持。为了实现 malloc()，它定义一个 32K的 heap区域，在此区域的基础上实现了简化版的 sbrk()。

图 3中，stack区域是在 U-BOOT启动的前半部分中第三步设置的。它首先根据 BSS_END、heap大小和 stack大小算出 stack_bottom的值，然后设置堆栈指针 SP和帧指针 FP为 stack_bottom - 4。

u-boot 中的命令实现

软件平台：u-boot-1.1.6, gcc for blackfin, visual dsp 5.0

我们知道，u-boot的运行过程是首先进行一些初始化化工作，然后在一个死循环中不断接收串口的命令并进行解释执行，下面我们就看看执行部分代码的实现，见 common/main.c中的 run_command：

```
int run_command (const char *cmd, int flag)
{
...
    while (*str) {
      ...
        /* find macros in this token and replace them */
        process_macros (token, finaltoken);
        /* Extract arguments */
        if ((argc = parse_line (finaltoken, argv)) == 0) {
            rc = -1;    /* no command at all */
```

```
            continue;
        }
        /* Look up command in command table */
        if ((cmdtp = find_cmd(argv[0])) == NULL) {
            printf ("Unknown command '%s' - try 'help'\n", argv[0]);
            rc = -1;    /* give up after bad command */
            continue;
        }
        /* found - check max args */
        if (argc > cmdtp->maxargs) {
            printf ("Usage:\n%s\n", cmdtp->usage);
            rc = -1;
            continue;
        }
    ...
        /* OK - call function to do the command */
        if ((cmdtp->cmd) (cmdtp, flag, argc, argv) != 0) {
            rc = -1;
        }
        repeatable &= cmdtp->repeatable;
        /* Did the user stop this? */
        if (had_ctrlc ())
            return 0; /* if stopped then not repeatable */
    }
    return rc ? rc : repeatable;
}
```

很简单的一个过程，扩展宏定义 -> 分析命令及其参数 -> 查找命令 -> 执行命令，有意思的地方在查找命令上（common/command.c）：

```
cmd_tbl_t *find_cmd (const char *cmd)
{
    cmd_tbl_t *cmdtp;
    cmd_tbl_t *cmdtp_temp = &__u_boot_cmd_start;    /*Init value */
    const char *p;
    int len;
    int n_found = 0;
    /*
    * Some commands allow length modifiers (like "cp.b");
    * compare command name only until first dot.
    */
    len = ((p = strchr(cmd, '.')) == NULL) ? strlen (cmd) : (p - cmd);
    for (cmdtp = &__u_boot_cmd_start;
        cmdtp != &__u_boot_cmd_end;
        cmdtp++) {
        if (strncmp (cmd, cmdtp->name, len) == 0) {
```

```
            if (len == strlen (cmdtp->name))
                return cmdtp; /* full match */
            cmdtp_temp = cmdtp;    /* abbreviated command ? */
            n_found++;
        }
    }
    if (n_found == 1) {                 /* exactly one match */
        return cmdtp_temp;
    }
    return NULL; /* not found or ambiguous command */
}
```

看起来还是很简单的一个过程，在一个命令数组中查找是否有指定名称的命令。问题是，在这里使用的两个符号 __u_boot_cmd_start和 __u_boot_cmd_end，在所有的 C文件中都找不到它们的定义，那么它们的空间从哪里来呢？这些分散在不同文件中的结构体又是如何能够放在同一个数组中呢？

答案就在 board/bf561-ezkit/u-boot.lds.s中，这个文件其实就是一个链接文件，类似于 VDSP中的 LDF文件， see see：

```
___u_boot_cmd_start = .;
.u_boot_cmd : { *(.u_boot_cmd) }
___u_boot_cmd_end = .;
```

这几句话的意思其实就是指示链接器将所有 .u_boot_cmd数据段中的内容全部放在一起，而且 ___u_boot_cmd_start和 ___u_boot_cmd_end是不会占用任何存储空间的，它们只是用来指示地址的两个符号而已。那么数据段的定义在哪里呢？看看 U_BOOT_CMD的宏定义吧（ include/command.h）：

```
#define Struct_Section __attribute__ ((unused,section (".u_boot_cmd")))
#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}
__attribute__ ((unused,section (".u_boot_cmd")))
```

就指示编译器将这些用 U_BOOT_CMD定义的结构体放在 .u_boot_cmd这个数据段中。

如果要在 VDSP中编译 u-boot，那么就需要在 LDF文件中也定义这样一个数据段：

```
        .u_boot_cmd
    {
        ___u_boot_cmd_start = .;
        INPUT_SECTIONS(common.dlb(.u_boot_cmd) common.dlb(__u_boot_cmd))
        ___u_boot_cmd_end = .;
    } > MEM_SDRAM_U_BOOT
```

不过让人郁闷的是：如果在一个定义命令的 C文件中没有一个函数被其它文件引用， VDSP在链接时将认为这是一个多余的文件，从而不会将这个文件中的函数链接进来，当然就无法使用其中定义的这些命令，如 cmd_load.c。

解决的办法可以是在这些文件中添加一个空函数，并在主函数中调用它们，这样 VDSP就会把这个文件链接进来了。

U-BOOT 环境变量实现

(基于 smdk2410)

1.相关文件

common/env_common.c
供 u-boot 调用的通用函数接口，它们隐藏了 env 的不同实现方式，比如 dataflash, epprom, flash 等

common/env_dataflash.c
env 存储在 dataflash 中的实现

common/env_epprom.c
env 存储在 epprom 中的实现

common/env_flash.c
env 存储在 flash 中的实现

common/env_nand.c
env 存储在 nand 中的实现

common/env_nvedit.c
实现 u-boot 对环境变量的操作命令

environment.c
环境变量以及一些宏定义

env 如果存储在 Flash 中还需要 Flash 的支持。

2.数据结构

env 在 u-boot 中通常有两种存在方式，在永久性存储介质中（ Flash NVRAM 等 ）在 SDRAM，可以配置不使用 env 的永久存储方式，但这不常用。u-boot 在启动的时候会将存储在永久性存储介质中的 env 重新定位到 RAM 中，这样可以快速访问，同时可以通过 saveenv 将 RAM 中的 env 保存到永久性存储介质中。

在 include/environment.h 中定义了表示 env 的数据结构

```
typedef struct environment_s
{
    unsigned long crc;    /* CRC32 over data bytes */
#ifdef CFG_REDUNDAND_ENVIRONMENT
```

```
    unsigned char flags;    /* active/obsolete flags */
#endif
    unsigned char data[ENV_SIZE]; /* Environment data */
} env_t;
```

关于以上结构的说明:

crc **是** u-boot **在保存** env **的时候加上去的校验头，在第一次启动时一般** crc **校验会出错，这很正常，因为这时** Flash **中的数据无效。**

data **字段保存实际的环境变量。**u-boot **的** env **按** name=value"\0"**的方式存储，在所有** env **的最后以**"\0\0"**表示整个** env **的结束。新的** name=value **对总是被添加到** env **数据块的末尾，当删除一个** name=value **对时，后面的环境变量将前移，对一个已经存在的环境变量的修改实际上先删除再插入。**

env **可以保存在** u-boot **的** TEXT **段中，这样** env **就可以同** u-boot **一同加载入** RAM **中，这种方法没有测试过。**

上文提到 u-boot **会将** env **从** flash **等存储设备重定位到** RAM **中，在** env **的不同实现版本（** env_xxx.c **）中定义了** env_ptr, **它指向** env **在** RAM **中的位置。**u-boot **在重定位** env **后对环境变量的操作都是针对** env_ptr**。**

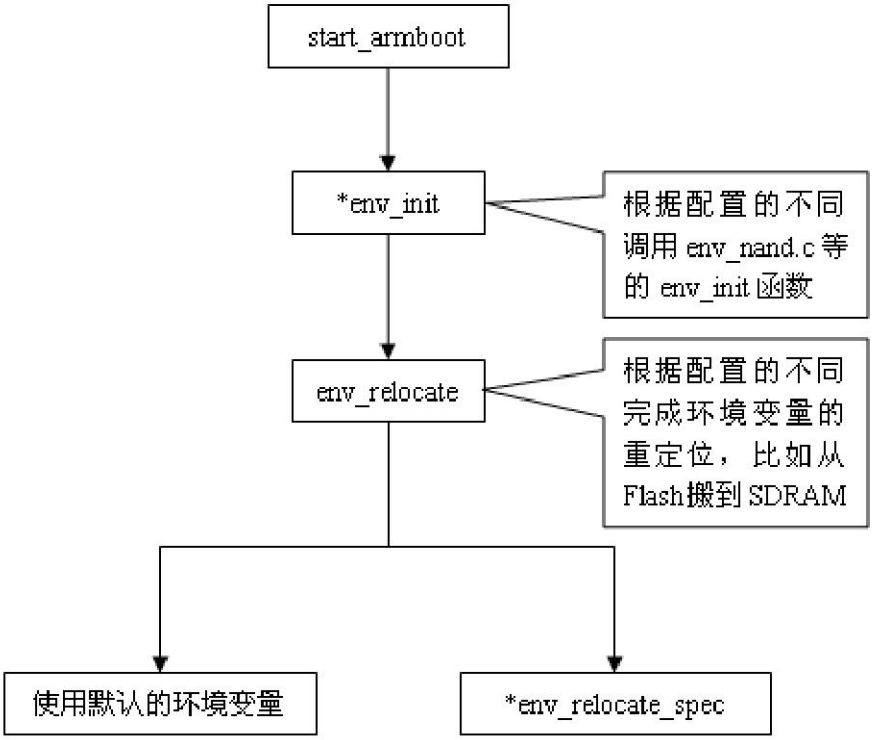
env_t **中除了数据之外还包含校验头，**u-boot **把** env_t **的数据指针有保存在了另外一个地方,这就是** gd_t **结构（ 不同平台有不同的** gd_t **结构 ）,这里以** ARM **为例仅列出和** env **相关的部分**

```
typedef struct global_data
{
    ...
    unsigned long env_off;          /* Relocation Offset */
    unsigned long env_addr;         /* Address of Environment struct ??? */
    unsigned long env_valid         /* Checksum of Environment valid */
    ...
} gd_t;
```

<include/asm-arm/Global_data.h>

gd_t.env_addr **即指向** env_ptr->data**。**

3.ENV 的初始化



start_armboot：（lib_arm/board.c）

*env_init：env_xxx.c（xxx = nand | flash | epprom …）

env_relocate：env_common.c

*env_relocate_spec：env_xxx.c（xxx=nand | flash | eporom…）

3.1env_init

实现 env 的第一次初始化，对于 nand env **（非** embedded **方式）**：

Env_nand.c：env_init

gd->env_addr = (ulong)&default_environment[0]; //**先使** gd->env_addr **指向默认的环境变量**

gd->env_valid = 1;// env **有效位置** 1

3.2 env_relocate

#ifdefine ENV_IS_EMBEDDED

…**（略）**

#else

env_ptr = (env_t *)malloc (CFG_ENV_SIZE);

#endif

if(gd->env_valid == 0) // **在** Env_annd.c：env_init **中已经将** gd->env_valid **置** 1

```
{
    ...
}
else
    env_relocate_spec ();// 调用具体的 env_relocate_spec 函数
gd->env_addr = (ulong)&(env_ptr->data);// 最终完成将环境变量搬移到内存
```

这里涉及到两个和环境变量有关的宏

ENV_IS_EMBEDDED：env 是否存在于 u-boot TEXT 段中

CFG_ENV_SIZE：env 块的大小

实际上还需要几个宏来控制 u-boot 对环境变量的处理

CFG_ENV_IS_IN_NAND：env 块是否存在于 Nand Flash 中

CFG_ENV_OFFSET：env 块在 Flash 中偏移地址

3.3*env_relocate_spec

这里仅分析 Nand Flash 的 env_relocate_spec 实现

如果未设置 CFG_ENV_OFFSET_REDUND，env_relocate_spec 的实现如下：

```
void env_relocate_spec (void)
{
#if !defined(ENV_IS_EMBEDDED)
    ulong total;
    int ret;

    total = CFG_ENV_SIZE;
    ret = nand_read(&nand_info[0], CFG_ENV_OFFSET, &total, (u_char*)env_ptr);
    if (ret || total != CFG_ENV_SIZE)
        return use_default();

    if (crc32(0, env_ptr->data, ENV_SIZE) != env_ptr->crc)
        return use_default();
#endif /* ! ENV_IS_EMBEDDED */
}
```

上面的代码很清楚的表明了 env_relocate_spec 的意图，调用 nand_read 将环境变量从 CFG_ENV_OFFSET 处读出，环境变量的大小为 CFG_ENV_SIZE 注意 CFG_ENV_OFFSET 和 CFG_ENV_SIZE 要和 Nand Flash 的块/页边界对齐。读出数据后再调用 crc32 对 env_ptr->data 进行校验并与保存在 env_ptr->crc 的校验码对比，看数据是否出错，从这里也可以看出在系统第一次启动时，Nand Flash 里面没有存储任何环境变量,crc 校验肯定回出错，当我们保存环境变量后，接下来再启动板子 u-boot 就不会再报 crc32 出错了。

4. ENV 的保存

由上问的论述得知, env 将从永久性存储介质中搬到 RAM 里面，以后对 env 的操作，比如修改环境变量

的值，删除环境变量的值都是对这个 env 在 RAM 中的拷贝进行操作，由于 RAM 的特性，下次启动时所做的修改将全部消失，u-boot 提供了将 env 写回 永久性存储介质的命令支持：saveenv，不同版本的 env（nand flash，flash ...）实现方式不同，以 Nand Flash 的实现（未定义 CFG_ENV_OFFSET_REDUND）为例

Env_nand.c：saveenv

```
int saveenv(void)
{
    ulong total;
    int ret = 0;

    puts ("Erasing Nand...");
    if (nand_erase(&nand_info[0], CFG_ENV_OFFSET, CFG_ENV_SIZE))
        return 1;

    puts ("Writing to Nand... ");
    total = CFG_ENV_SIZE;
    ret = nand_write(&nand_info[0], CFG_ENV_OFFSET, &total, (u_char*)env_ptr);
    if (ret || total != CFG_ENV_SIZE)
        return 1;

    puts ("done\n");
    return ret;
}
```

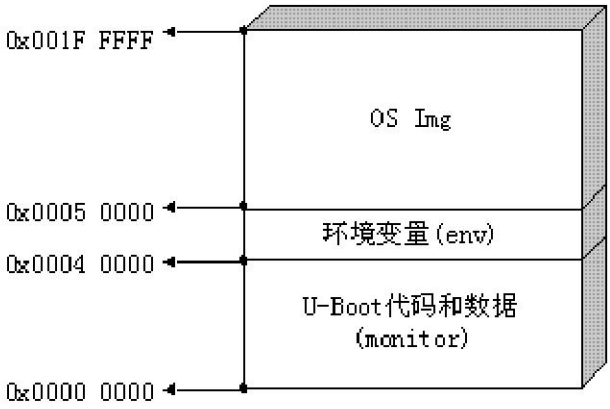
Nand Flash 的 saveenv 命令实现很简单，调用 nand_erase 和 nand_write 进行 Nand Flash 的 erase, write。nand_write/erase 使用的是 u-boot 的 nand 驱动框架，我在做开发的过程中使用的是 nand_legacy 驱动，所以可以把 nand_erase 和 nand_write 改成 nand_legacy_erase 和 nand_legacy_rw 就可实现 nand_legacy 驱动的保存环境变量版本。

==========

U-Boot 环境变量

U-Boot 通过环境变量（env）为用户提供一定程度的可配置性，这些环境变量包括串口终端所使用的波特率(baudrate)、启动操作系统内核的参数(bootargs)、本地 IP 地址(ipaddr)、网卡 MAC 地址(ethaddr)等等。环境变量可以固化到非易失性存储介质中，使用 printenv / saveenv 命令来查看和修改。本例中，环境变量固化到 Flash 中（AM29LV160DB，2MB)。

可配置性意味着环境变量中的项目是可以被添加、删除和修改的，即环境变量的内容可能会频繁变化。为了不让这种变化对 U-Boot 的代码和数据造成破坏，通常的选择是在 Flash 中准备一个专用的 sector 来存储环境变量。简化的 ROM 分配模型如下图所示，monitor 占用 Flash 前 256KB，env 置于其后，Flash 的最后一部分用来存放压缩的操作系统内核。



AM29LV160DB 分为 35 个 sector，地址范围分配如下：

Sector	Size (KB)	Address Range (Hex)
SA0	16	000000 ~ 003FFF
SA1	8	004000 ~ 005FFF
SA2	8	006000 ~ 007FFF
SA3	16	008000 ~ 00FFFF
SA4	32	010000 ~ 01FFFF
SA5	64	020000 ~ 02FFFF
…	64	…
SA34	64	1F0000 ~ 1FFFFF

由于 U-Boot 代码通常达到 100KB 左右，且必须从地址 0 处开始，按照这样的分配方式，我们将不得不为 env 分配一块 64KB 的 sector，而实际中使用到的可能只是其中的几百字节！U-Boot 还会为 env 在 RAM 中保持一块同样大小的空间，这就造成 ROM 和 RAM 空间不必要的浪费。
为了尽可能地减少资源浪费，同时保证系统的健壮性，我们可以把 env 放置在 Flash 中容量最小的 sector 里。这样，env 嵌入（embed）到 U-Boot 的代码段。在 common/environment.h 中会比较 env 和 monitor 的范围，如果确定 env 位于 monitor 内，则定义 ENV_IS_EMBEDDED。

```
# if (CFG_ENV_ADDR >= CFG_MONITOR_BASE) && \
      (CFG_ENV_ADDR+CFG_ENV_SIZE) <= (CFG_MONITOR_BASE + CFG_MONITOR_LEN)
#   define ENV_IS_EMBEDDED   1
# endif
```

修改 board/buf/EVB44B0/u-boot.lds：

```
/*------------------------------------------------------------
 * Environment Variable setup
 */
```

```
#define CFG_ENV_IS_IN_FLASH        1  /* 使用 Flash 存储 env   */
#define CFG_ENV_SIZE           0x2000   /* 容量 8KB (SA1)       */
#define CFG_ENV_OFFSET  0x4000   /* 偏移地址 (SA1)       */
```

$(LD)将一系列的 obj 文件连接成 elf 格式文件，其输出文件的内存布局由 linker script 决定。修改 board/buf/EVB44B0/u-boot.lds：

```
SECTIONS
{
    . = 0x00000000;
    . = ALIGN(4);

    .text      :
    {
      cpu/s3c44b0/start.o (.text)
       board/buf/EVB44B0/lowlevel_init.o (.text)
      lib_generic/string.o (.text)
   lib_generic/zlib.o (.text)

      . = env_offset;
      common/environment.o (.text)

      *(.text)
    }

    /* other sections ... */
}
```

从 u-boot.map 选择那些 U-Boot 运行必须的，且不易受 CFG_*宏影响的 obj 文件，填充到 start.o 后面。可以参考 board/trab/u-boot.lds。

env_offset 定义在 common/environment.c 中：

```
#define GEN_SYMNAME(str) SYM_CHAR #str
#define GEN_VALUE(str) #str
#define GEN_ABS(name, value) \
          asm (".globl " GEN_SYMNAME(name)); \
          asm (GEN_SYMNAME(name) " = " GEN_VALUE(value))
GEN_ABS(env_offset, CFG_ENV_OFFSET);
```

u-boot 代码链接的问题

环境和配置：u-boot-1.1.2, arm-linux-gcc(v3.2), redhat linux9.0, cpu(s3c44b0), board(B2)

在 /board/dave/B2/u-boot.lds有

```
. = 0x00000000;


. = ALIGN(4);

.text :

{

cpu/s3c44b0/start.o (.text)

*(.text)

}
```

而 /board/dave/B2/config.mk中有

```
TEXT_BASE = 0x0C100000
```

最后编译出来的代码反汇编得到

```
0c100000 <_start>:

c100000: ea00000a b c100030 <reset>

c100004: e28ff303 add pc, pc, #201326592 ; 0xc000000

c100008: e28ff303 add pc, pc, #201326592 ; 0xc000000

c10000c: e28ff303 add pc, pc, #201326592 ; 0xc000000

c100010: e28ff303 add pc, pc, #201326592 ; 0xc000000

c100014: e28ff303 add pc, pc, #201326592 ; 0xc000000
```

```
c100018: e28ff303 add pc, pc, #201326592 ; 0xc000000

c10001c: e28ff303 add pc, pc, #201326592 ; 0xc000000



0c100020 <_TEXT_BASE>:

c100020: 0c100000 ldceq 0, cr0, [r0]


0c100024 <_armboot_start>:

c100024: 0c100000 ldceq 0, cr0, [r0]


0c100028 <_bss_start>:

c100028: 0c115694 ldceq 6, cr5, [r1], -#592


0c10002c <_bss_end>:

c10002c: 0c1198a4 ldceq 8, cr9, [r1], -#656


0c100030 <reset>:

c100030: e10f0000 mrs r0, CPSR
```

链接得到的起始地址为什么是 TEXT_BASE，而不是 0呢，所以现在只能够下载到ram中运行，但是无法烧写道 flash中跑，这是怎么回事呢？u-boot中应该是start.S中的这段代码在 flash中运行吧，后面就把自身拷贝到 ram中 TEXT_BASE地址处，为什么在链接文件中指定的 _start的起始地址为 0x00000000呢？

blob中把整个代码分为两部分，所以有两个连接文件，前面 1k在 flash中运行，

链接起始地址为 0x0，ram中运行的，然后通过工具把两部分组合到一起，但是u-boot是怎么做的呢？

迷惑中，请帮忙解答，谢先！

Re: u-boot代码链接的问题

好象与这个地址没关系的
u-boot既可以在 SDRAM中，
也可在 Flash中运行

Re: u-boot代码链接的问题

兄弟，首先需要知道 CPU的启动方式，一般来说有 BOOTROM, SPI, FLASH这三种方式；一般来说也不会存在从 SDRAM/DRAM上启动机器，因为它们不可能作为永久存储。
然后就是需要知道，外设地址映射关系，对于 CPU来说总是有统一的 IO地址映射，并且在不同的访问模式下，地址映射关系不一定相同。
还有，代码可以运行在不同的介质上，如：上述三种再包括 RAM/ROM。
另外：还要知道代码段的链接地址，这个你应该是知道的。

最后，就可以考虑你的 FLASH启动问题了。在 Uboot里面的 -text的参数，应该是真对某个硬件系统的配置，这个就是 FLASH的高端地址或者低端地址。
一般来说，bootloader的代码的第一段总是运行在永久介质上，例如：FLASH，然后才将代码搬移到 DRAM中，这个时候在 dram中执行第二段 BOOTLOADER的代码段，也就是你说下载到内存中的说法，实际上他都已经运行了一个阶段了。在内存中的运行入口地址，按照系统情况可以自行安排了。

因此，这个最开始的 FLASH地址，肯定不是 0，是什么看看你的单板的 DATASHEET和编程参考

Re: u-boot代码链接的问题

谢谢，我的板子 cpu是 s3c44b0，不支持 memory remap，flash为 4M，从 0x00000000到 0x00400000，
sdram为 8M，地址从 0x0c000000到 0x0c800000，我不清楚的问题时链接脚本中指

定的 _start的入口地址为 0x00000000，为何编译出来的代码。链接地址是从 0x0c100000开始的，即把 TEXT_BASE作为 _start的入口地址了，如果我把 TEXT_BASE改为 0x00000000，那么后面的 rellocate代码还有意义吗？

Re: u-boot代码链接的问题

我猜测，问题可能有可能，我说的仅仅是 may be:
和我前面说的一样，boot loader一般有两段代码段（独立的），在连接的时候也是分别连接，故此有两个代码连接文件，和编译过程共产生的两个 MAP文件，可以查一下你看的连接文件和编译文件是不是对应的。

另外有两个建议：
1: 这个也是疑惑，我看了你的问题后，作了这样的前提假设（个人的理解）：在 dram里的 loader stage起始地址是定义在 sdram的最低端。从这个假设出来，我觉得这个链接也是有问题的，因为一般来说 sdram的最低端是做向量表、和模式栈顶区、参数区来用的，这个肯定有问题

2: 分析这个问题，首先要确定你的编译过程，可以把编译过程定向到一个文件中仔细的分析，可以看到编译链接各个细节。不妨发给我一份
hls780204cn@vip.sina.com

Re: u-boot代码链接的问题

TEXTADDR 是内核的虚拟起始地址，并且在 arch/<target>/ 下的 Makefile 中指定它的值。这个地址必须与引导装载程序使用的地址相匹配。 一旦引导装载程序将内核复制到闪存或 DRAM 中，内核就被重新定位到 TEXTADDR — 它通常在 DRAM 中，然后，引导装载程序将控制转给这个地址，以便内核能开始执行。

Re: u-boot代码链接的问题

我已经理解了：
“链接得到的起始地址为什么是 TEXT_BASE，而不是 0呢，”
因为 u_boot如果从 flash运行的话，那么它会将自己的代码拷贝到 RAM中，然后运行。 u-boot开始部分代码与编译的入口没有关系，而主要的代码是在 RAM中运行，因此编译的入口地址是 TEXT_BASE.因此 u-boot既可以 flash运行，也可以 ram运行。

“为什么在链接文件中指定的 _start的起始地址为 0x00000000呢？”
lds文件中的起始地址为 0x00000000是不起作用的，由 -TTEXT_BASE参数替代的。

刚开始比较疑惑的原因是对：
126 relocate: /* relocate U-Boot to RAM */
127 adr r0, _start /* r0 <- current position of code */
adr这条指令没有理解正确，因为把它想成 mv r0,_start了，实际上 adr这里的 _start是相对的，如果从 flash运行的话，r0就是 0， 如果从 ram运行的话，r0就是 C100000。

我现在可以运行 u-boot了，串口可以显示内容并且可以使用命令。但网卡驱动和 flash驱动还有问题。慢慢搞就可以搞定，因为可以用 printf调试的。

Re: u-boot代码链接的问题

楼上是不是说 因为 adr指令是小范围地址读取指令，所以在不同的运行环境下，_start的值不同啊。

针对 44box中的代码
```
adr r0, real_vectors
add r2, r0, #1024
ldr r1, =0x0c000000
add r1, r1, #0x08
vector_copy_loop:
ldmia r0!, {r3-r10}
stmia r1!, {r3-r10}
cmp r0, r2
ble vector_copy_loop
```
复制向量中断，为什么要复制 1024个字节呢？

问得比较弱，见笑了。

Re: u-boot代码链接的问题

多一点是没有关系的

Re: u-boot代码链接的问题

在 FLASH中以相对地址运行，然后一个绝对跳转完成从 FLASH到 RAM的切换，应该制定为 UBOOT在 RAM中的起始地址！

看 makefile 文件
```
$(obj)u-boot.img:    $(obj)u-boot.bin
```

```
./tools/mkimage -A $(ARCH) -T firmware -C none \
-a $(TEXT_BASE) -e 0 \
-n $(shell sed -n -e 's/.*U_BOOT_VERSION//p' $(VERSION_FILE) | \
    sed -e 's/"[   ]*$$/ for $(BOARD) board"/') \
-d $< $@
```

ldr 和 adr 在使用标号表达式作为操作数的区别

ARM汇编有 ldr指令以及 ldr、 adr伪指令，他门都可以将标号表达式作为操作数，下面通过分析一段代码以及对应的反汇编结果来说明它们的区别。

```
        ldr     r0, _start

        adr     r0, _start

        ldr     r0, =_start

_start:

        b  _start
```

编译的时候设置 RO 为 0x30000000, 下面是反汇编的结果：

```
0x00000000: e59f0004  ldr r0, [pc, #4] ; 0xc

0x00000004: e28f0000  add r0, pc, #0 ; 0x0

0x00000008: e59f0000  ldr r0, [pc, #0] ; 0x10

0x0000000c: eafffffe  b 0xc

0x00000010: 3000000c  andcc r0, r0, ip
```

1. ldr   r0, _start

   这是一条指令，从内存地址 _start 的位置把值读入。

在这里 _start是一个标号（是一个相对程序的表达式），汇编程序计算相对于 PC 的偏移量，并生成相对于 PC的前索引的指令： ldr r0, [pc, #4]。执行指令后， r0 = 0xeaffffe。

ldr r0, _start是根据 _start对当前 PC的相对位置读取其所在地址的值，因此可以在和 _start标号的相对位置不变的情况下移动。

2. adr r0, _start

这是一条伪指令，总是会被汇编程序汇编为一个指令。汇编程序尝试产生单个 ADD 或 SUB 指令来装载该地址。如果不能在一个指令中构造该地址，则生成一个错误，并且汇编失败。

在这里是取得标号 _start 的地址到 r0, 因为地址是相对程序的，因此 ADR产生依赖于位置的代码，在此例中被汇编成： add r0, pc, #0。因此该代码可以在和标号相对位置不变的情况下移动；

假如这段代码在 0x30000000 运行，那么 adr r0, _start 得到 r0 = 0x3000000c；如果在地址 0 运行，就是 0x0000000c 了。

通过这一点可以判断程序在什么地方运行。U-boot中那段 relocate代码就是通过 adr实现当前程序是在 RAM中还是 flash中，下面进行简要分析。

```
relocate: /* 把 U-Boot重新定位到 RAM */
    adr r0, _start /* r0是代码的当前位置 */
/* adr伪指令，汇编器自动通过当前 PC的值算出 如果执行到 _start时 PC的值，放到 r0中：
当此段在 flash中执行时 r0 = _start = 0; 当此段在 RAM中执行时 _start = _TEXT_BASE(在 board/smdk2410/config.mk中指定的值为 0x30000000, 即 u-boot在把代码拷贝到 RAM中去执行的代码段的开始） */
    ldr r1, _TEXT_BASE /* 测试判断是从 Flash启动，还是 RAM */
/* 此句执行的结果 r1始终是 0x30000000, 因为此值是又编译器指定的 (ads中设置，或 -D设置编译器参数） */
```

```
        cmp r0, r1 /* 比较 r0和 r1, 调试的时候不要执行重定位 */
```

3. ldr r0, =_start

这是一条伪指令，是一个相对程序的或外部的表达式。汇编程序将相对程序的标号表达式 label-expr 的值放在一个文字池中，并生成一个相对程序的 LDR 指令来从文字池中装载该值，在此例中生成的指令为：ldr r0, [pc, #0]，对应文字池中的地址以及值为：0x00000010: 3000000c。如果 label-expr 是一个外部表达式，或者未包含于当前段内，则汇编程序在目标文件中放置一个链接程序重定位命令。链接程序在链接时生成地址。

因此取得的是标号 _start 的绝对地址，这个绝对地址（运行地址）是在连接的时候确定的。它要占用 2 个 32bit 的空间，一条是指令，另一条是文字池中存放 _start 的绝对地址。因此可以看出，不管这段代码将来在什么地方运行，它的结果都是 r0 = 0x3000000c。由于 ldr r0, =_start取得的是 _start的绝对地址，这句代码可以在 _start 标号的绝对位置不变的情况下移动；如果使用寄存器 pc在程序中可以实现绝对转移。

参考资料：

1. ARM DUI 0204BSC, RealView编译工具 2.0 版 汇编程序指南，http://infocenter.arm.com/help/index.jsp

2. GNU汇编使用经验，http://blog.chinaunix.net/u1/37614/showart_390095.html

3. 对 .lds连接脚本文件的分析，http://blog.chinaunix.net/u1/58780/showart.php?id=462971

start_armboot 浅析

ARM920t 架构的 CPU 在完成基本的初始化后（ARM 汇编代码），就进入它的 C 语言代码，而 C 语言代码的入口就是 start_armboot, start_armboot 在 lib_arm/board.c 中。start_armboot 将完成以下工作。

1.全局数据结构的初始化

比如 gd_t 结构的初始化:

```
251         gd = (gd_t*)(_armboot_start – CFG_MALLOC_LEN – sizeof(gd_t));
```

_armboot_start 是 u-boot 在 RAM 中的开始地址（对于 u-boot 最终搬移到 RAM 中运行的情

况）,CFG_MALLOC_LEN **在** include/configs/<board name>.h **中定义。**

bd_t **结构的初始化**:
272 gd->bd = (bd_t*)((char*)gd-sizeof(bd_t));
u-boot **把** bd_t **结构紧接着** gd_t **结构存放。**

内存分配的初始化
316 mem_malloc_init(_armboot_start-CFG_MALLOC_LEN);
经过以上的初始化后，u-boot **在内存中的布局为（在底端为低地址）**

BSS

U-BOOT TEXT/DATA

CFG_MALLOC_LEN
---
gd_t
---
bd_t

STACK

2.调用通用初始化函数

```
for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
        if ((*init_fnc_ptr)() != 0) {
            hang ();
        }
    }
```

init_sequence[]**是** init_fnc_t **函数指针数组，这个数组包含了众多初始化函数，比如** cpu_init**，** board_init **等。**

```
//init_fnc_t *init_sequence[] = {
    //      cpu_init,           /* basic cpu dependent setup */
    //      board_init,         /* basic board dependent setup */
    //      interrupt_init,         /* set up exceptions */
    //      env_init,           /* initialize environment */
    //      init_baudrate,          /* initialze baudrate settings */
    //      serial_init,            /* serial communications setup */
    //      display_banner,
    //      dram_init,          /* configure available RAM banks */
    //      display_dram_config,
    //      NULL,
    //      };
```

3.初始化具体设备

这一部分包括对 Flash，LCD，网络的初始化等，例如

```
318  #if (CONFIG_COMMANDS & CFG_CMD_NAND)
     puts ("NAND:    ");
     nand_init();            /* go init the NAND */
     #endif

367  devices_init();

386  #ifdef CONFIG_DRIVER_CS8900
     cs8900_get_enetaddr (gd->bd->bi_enetaddr);
     #endif
```

4.初始化环境变量

环境变量在通用初始化函数里面，已经初始化一次（env_init），这里调用 env_relocate 对环境变量进行重新定位。在我的另一篇文章"U-BOOT ENV 实现"中有对环境变量实现的讨论。

5.进入主循环

当然 start_armboot 除了以上工作外，还完成其它的初始化工作，具体参考 lib_arm/board.c，在一切准备就绪之后，就进入 u-boot 的主循环:

```
416  for (;;) {
          main_loop ();
     }
```

main_loop 的代码比较长，基本是就是执行用户的输入命令。

u-boot 编译过程

现在介绍一下 u-boot 的编译过程，这里用的 uboot 版本是 U-Boot 2008.10，硬件用 smdk2410，这个板子用得比较普遍，uboot 已经有对其的支持。通过我们对编译过程和代码的了解，我们也容易用 uboot 支持我们自己需要的硬件。

编译命令非常简单：

make smdk2410_config （生成配置）

make all （生成最终文件）

当然，更好的做法是把编译出的文件生成到另外一个目录，并 make clean 如：

```
export BUILD_DIR=../tmp
make distclean
make smdk2410_config
make all
```

现在，我们可以来看看 Makefile，u-boot 的 Makefile 文件非常大。但是，其结构却并不复杂。

u-boot 已经支持了很多硬件，前半部分是共用部分，编译出最终的 uboot 可执行文件。

而后半部分，是为各种不同的硬件进行配置，每种硬件有一个目标，每个的做法都非常类似，我们用到的是：

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

这里的 MKCONFIG := $(SRCTREE)/mkconfig

实际上是调用脚本 mkconfig，而这个脚本做的工作简单如下：

建立 include/config.mk 文件

```
    echo "ARCH   = $2" >  config.mk
    echo "CPU    = $3" >> config.mk
    echo "BOARD  = $4" >> config.mk
    echo "VENDOR = $5" >> config.mk
    echo "SOC    = $6" >> config.mk
```

建立 include/config.h

```
    echo "#include <configs/$1.h>" >>config.h
```

在这里$1-$6 的值分别是：smdk2410 arm arm920t smdk2410 NULL s3c24x0

而执行了 make smdk2410_config 之后，就生成了相应的 config.mk，config.h 两个文件。

在 config.mk 文件中，定义了相应硬件信息 ： ARCH CPU BOARD VENDOR SOC

在 config.h 文件中，包含了相应硬件的头文件 smdk2410.h ，位于 include\configs 目录下。

如果新建自己的硬件项目，那么也需要建立相应的头文件在这个地方。

这样，uboot 的配置已经生成，下一次介绍 make all 的过程。

接着上次，这次介绍 make all 的过程。
首先，介绍一下生成的 config.mk 和 config.h 如何使用，得到正确配置的。

config.mk 直接被 include 到 Makefile 来,并使用其定义如下：

```
include $(obj)include/config.mk
export ARCH CPU BOARD VENDOR SOC
```

这样可以直接选择需要编译的模块，例如：

```
LIBS += cpu/$(CPU)/$(SOC)/lib$(SOC).a
LIBS += lib_$(ARCH)/lib$(ARCH).a
LIBBOARD = board/$(BOARDDIR)/lib$(BOARD).a
```

config.h 被 include/common.h 所包含，而它有包含了相应硬件的头文件。

```
common.h <--- config.h <--- smdk2410.h
```

除了在源程序中，使用这些头文件的定义之外，uboot 还有一个脚本通过解析 common.h 以及其包含的所有头文件信息，来生成配置信息如下形式：

```
CONFIG_BAUDRATE=115200
CONFIG_NETMASK="255.255.255.0"
CONFIG_DRIVER_CS8900=y
CONFIG_ARM920T=y
CONFIG_RTC_S3C24X0=y
CONFIG_CMD_ELF=y
```

而头文件的定义的形式如下，对比可以看出脚本的工作原理。

```
#define CONFIG_BAUDRATE 115200
#define CONFIG_NETMASK 255.255.255.0
#define CONFIG_DRIVER_CS8900 1 /* we have a CS8900 on-board */
#define CONFIG_ARM920T 1 /* This is an ARM920T Core */
#define CONFIG_RTC_S3C24X0 1
#define CONFIG_CMD_ELF
```

通过这样，uboot 可以自动得到一个模块选择的配置功能。如果我们需要添加什么定义或者功能，也可以在相应的头文件中加入定义实现。

现在，配置已经得到，就看最后的编译流程。
编译分为五大部分，分别如下：

1. $(SUBDIRS) 工具，例子等，包括目录：tools examples api_examples
2. $(OBJS) 启动模块 cpu/arm920t/start.o

3. $(LIBBOARD) **板子支持模块** board/smdk2410/libsmdk2410.a
4. $(LIBS) **其他模块，有诸如以下模块：**

   cpu/arm920t/libarm920t.a
   cpu/arm920t/s3c24x0/libs3c24x0.a
   lib_arm/libarm.a
   fs/jffs2/libjffs2.a
   fs/yaffs2/libyaffs2.a
   net/libnet.a
   disk/libdisk.a
   drivers/bios_emulator/libatibiosemu.a
   drivers/mtd/libmtd.a
   drivers/net/libnet.a
   drivers/net/phy/libphy.a
   drivers/net/sk98lin/libsk98lin.a
   drivers/pci/libpci.a
   common/libcommon.a /

5. $(LDSCRIPT) **链接脚本**

编译完成这五部分，链接成 elf **格式的** u-boot **文件，最后通过** objcopy -O binary **命令将** elf **格式转换成为** raw binary **格式的文件** u-boot.bin **就可以烧到板子上使用了。**

mkconfig 文件的分析

http://niutao.org/blog/?p=50

在编译 u-boot **之前都要执行**"make XXX_config"**命令，笼统的说是配置** u-boot**，使其编译出适合目标板的** bootloader**。那么该命令都做了那些工作，具体的执行过程是怎样的？**

我们首先从 u-boot **的** Makefile **文件看起，例如我们首先执行**"make smd2410_config"**命令，则在** Makefile **中会执行：**

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

也就是：

```
./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24×0
```

也就是说执行”make XXX_config”之后，实际上执行的是mkconfig脚本，下面是对mkconfig文件的分析：

```
#!/bin/sh -e

APPEND=no    # Default: Create new config file
BOARD_NAME=""    # Name to print in make output
#如果命令行参数中有--,-a,-n等参数，则执行以下循环。
#如果有-n XXX_config或者-n XXX，则取出XXX作为目标板的名字
while [ $# -gt 0 ] ; do
    case "$1" in
    --) shift ; break ;;
    -a) shift ; APPEND=yes ;;
    -n) shift ; BOARD_NAME="${1%%_config}" ; shift ;;
    *)  break ;;
    esac
done
#如果传递给该脚本的参数小于4个或者大于6个，则退出
[ "${BOARD_NAME}" ] || BOARD_NAME="$1"

#如果BOARD_NAME为空，则BOARD_NAME等于传递给该脚本的第一个参数
[ $# -lt 4 ] && exit 1
[ $# -gt 6 ] && exit 1

echo "Configuring for ${BOARD_NAME} board..."
#OBJTREE和SRCTREE都是在Makefile中导出的变量，分别为编译目录和源码目录
#如果编译目录和源码目录不为同一目录，则执行一下命令，创建$(OBJTREE)/include等目录
if [ "$SRCTREE" != "$OBJTREE" ] ; then
    mkdir -p ${OBJTREE}/include
    mkdir -p ${OBJTREE}/include2
    cd ${OBJTREE}/include2
    rm -f asm
    ln -s ${SRCTREE}/include/asm-$2 asm
    LNPREFIX="../../include2/asm/"
    cd ../include
    rm -rf asm-$2
    rm -f asm
    mkdir asm-$2
    ln -s asm-$2 asm
else
#如果编译目录和源码目录为同一个目录，则进入include目录，删除旧的asm链接，创建目
#标板到asm的链接，例如如果目标体系结构为arm，则创建asm软链接指向asm-arm。
```

```
    cd ./include
    rm -f asm
    ln -s asm-$2 asm
fi
#删除目标平台下的旧的 arch 链接
rm -f asm-$2/arch
#如果第 6 个参数长度为 0 或者其等于 NULL，则创建 arch-$3 到 asm-$2 的链接，对于命令
#"./mkconfig smdk2410 arm arm920t smdk2410 NULL s3c24x0"
# $6=s3c24x0 不为空
# $3=arm920t $2=arm
#则最终执行的命令为"ln -s arch-s3c24x0 asm-arm/arch"
if [ -z "$6" -o "$6" = "NULL" ] ; then
    ln -s ${LNPREFIX}arch-$3 asm-$2/arch
else
    ln -s ${LNPREFIX}arch-$6 asm-$2/arch
fi
#如果目标平台为 ARM，则删除建立的 asm-arm 链接，重新建立从 proc-armv 到 asm-arm 的链接
if [ "$2" = "arm" ] ; then
    rm -f asm-$2/proc
    ln -s ${LNPREFIX}proc-armv asm-$2/proc
fi

#
# Create include file for Make
#
echo "ARCH   = $2" >  config.mk
echo "CPU    = $3" >> config.mk
echo "BOARD  = $4" >> config.mk
#如果参数 5 存在并且不为 NULL，则将 VENDOR = $5 追加在 config.mk 文件中
[ "$5" ] && [ "$5" != "NULL" ] && echo "VENDOR = $5" >> config.mk
#如果参数 6 存在并且不为 NULL，则将 SOC = $6 追加在 config.mk 文件中
[ "$6" ] && [ "$6" != "NULL" ] && echo "SOC    = $6" >> config.mk

#创建 config.h 文件
if [ "$APPEND" = "yes" ]    # Append to existing config file
then
    echo >> config.h
else
    > config.h       # Create new config file
fi
echo "/* Automatically generated - do not edit */" >>config.h
```

```
echo "#include <configs/$1.h>" >>config.h

exit 0
```

总结一下，”make XXX_config”总共做了一下工作：

(1)确定开发板名称为 BOARD_NAME = $1。
(2)创建一些链接文件，为编译 u-boot 做准备：

```
    ln -s asm-$2 asm
    ln -s arch-$6 asm-$2/arch
    ln -s proc-armv asm-$2/proc #仅在目标平台为 arm 的时候才执行。
```

(3)创建顶层 Makefile 包含的文件 include/config.mk。

```
ARCH   = $2
CPU    = $3
BOARD  = $4
VENDOR = $5
SOC    = $6
```

(4)创建开发板相关的头文件 include/config.h。

```
/* Automatically generated - do not edit */
#include
```

从 NAND 闪存中启动 U-BOOT 的设计

南昌大学信息工程学院 刘晔 汪灿 2007-02-12 21:29:20 电子设计应用 /

引言

随着嵌入式系统的日趋复杂，它对大容量数据存储的需求越来越紧迫。而嵌入式设备低功耗、小体积以及低成本的要求，使硬盘无法得到广泛的应用。NAND闪存设备就是为了满足这种需求而迅速发展起来的。目前关于 U-BOOT的移植解决方案主要面向的是微处理器中的 NOR 闪存，如果能在微处理器上的 NAND 闪存中实现 U-BOOT的启动，则会给实际应用带来极大的方便。

U-BOOT简介

U-BOOT 支持 ARM、PowerPC等多种架构的处理器，也支持 Linux、NetBSD和 VxWorks等多种操作系统，主要用来开发嵌入式系统初始化代码 bootloader。bootloader是芯片复位后进入操作系统之前执行的一段代码，完成由硬件启动到操作系统启动的过渡，为运行操作系统提供基本的运行环境，如初始化 CPU、堆栈、初始化存储器系统等，其功能类似于 PC机的 BIOS。U-BOOT执行流程图如图 1所示。

图 1 U-BOOT启动流程图

NAND 闪存工作原理

S3C2410开发板的 NAND闪存由 NAND闪存控制器 (集成在 S3C2410 CPU中)和 NAND闪存芯片 (K9F1208U0A)两大部分组成。当要访问 NAND闪存芯片中的数据时 ,必须通过 NAND闪存控制器发送命令才能完成。所以 ,NAND闪存相当于 S3C2410的一个外设 ,而不位于它的内存地址区。

NAND闪存 (K9F1208U0A)的数据存储结构分层为 : 1设备 (Device) = 4096 块 (Block);1块 = 32页 /行 (Page/row);1页 = 528B = 数据块 (512B) + OOB块 (16B)在每一页中,最后 16个字节 (又称 OOB)在 NAND闪存命令执行完毕后设置状态,剩余 512个字节又分为前半部分和后半部分。可以通过 NAND闪存命令 00h/01h/50h分别对前半部、后半部、 OOB进行定位,通过 NAND闪存内置的指针指向各自的首地址。

NAND闪存的操作特点为 :擦除操作的最小单位是块 ; NAND闪存芯片每一位只能从 1变为 0, 而不能从 0变为 1, 所以在对其进行写入操作之前一定要将相应块擦除 ; OOB部分的第 6字节为坏快标志,即如果不是坏块该值为 FF, 否则为坏块 ;除 OOB第 6字节外,通常用 OOB的前 3个字节存放 NAND闪存的硬件 ECC(校验寄存器)码 ;

从 NAND 闪存启动 U-BOOT 的设计思路

如果 S3C2410被配置成从 NAND闪存启动 ,上电后, S3C2410的 NAND闪存控制器会自动把 NAND闪存中的前 4K数据搬移到内部 RAM中 ,并把 0x00000000设置为内部 RAM的起始地址 , CPU从内部 RAM的 0x00000000位置开始启动。因此要把最核心的启动程序放在 NAND闪存的前 4K中。

由于 NAND闪存控制器从 NAND闪存中搬移到内部 RAM的代码是有限的 ,所以 ,在启动代码的前 4K里 ,必须完成 S3C2410的核心配置,并把启动代码的剩余部分搬到 RAM中运行。在 U-BOOT中 ,前 4K完成的主要工作就是 U-BOOT启动的第一个阶段 (stage1)。

根据 U-BOOT的执行流程图,可知要实现从 NAND闪存中启动 U-BOOT, 首先需要初始化 NAND闪存 ,并从 NAND闪存中把 U-BOOT搬移到 RAM中,最后需要让 U-BOOT支持 NAND闪存的命令操作。 2开发环境

本设计中目标板硬件环境如下 :CPU为 S3C2410, SDRAM为 HY57V561620, NAND闪存为 64MB的 K9F1208U0A, 主机软件环境为 Redhat9.0、 U-BOOT-1.1.3、 gcc 2.95.3。修改 U-BOOT的 Makefile, 加入 :

```
wch2410_config : unconfig
@./mkconfig $(@:_config=) arm arm920t wch2410 NULL s3c24x0
```

即将开发板起名为 wch2410, 接下来依次进行如下操作 :

```
mkdir board/wch2410
cp board/smdk2410 board/wch2410
mv smdk2410.c wch2410.c
cp include/configs/smdk2410.h include/configs/wch2410.h
export PATH="/usr/local/arm/2".95.3/bin:$PATH
```

最后执行 :

```
make wch2410_config
make all ARCH="arm"
```

生成 u-boot.bin, 即通过了测试编译。

具体设计

支持 NAND 闪存的启动程序设计

因为 U-BOOT的入口程序是 /cpu/arm920t/start.S,故需在该程序中添加 NAND闪存的复位程序,以及实现从 NAND闪存中把 U-BOOT搬移到 RAM中的功能程序。

首先在 /include/configs/wch2410.h中加入 CONFIG_S3C2410_NAND_BOOT,如下：

```
#define CONFIG_S3C2410_NAND_BOOT 1 @支持从 NAND 闪存中启动
```

然后在 /cpu/arm920t/start.S中添加

```
#ifdef CONFIG_S3C2410_NAND_BOOT
copy_myself:
mov r10, lr
ldr sp, DW_STACK_START @安装栈的起始地址
mov fp, #0 @初始化帧指针寄存器
bl nand_reset @跳到复位 C函数去执行，执行 NAND闪存复位
.......
/*从 NAND闪存中把 U-BOOT拷贝到 RAM*/
ldr r0, =UBOOT_RAM_BASE@设置第 1个参数：UBOOT在 RAM中的起始地址
mov r1, #0x0 @设置第 2个参数 :NAND闪存的起始地址
mov r2, #0x20000 @设置第 3个参数：U-BOOT的长度 (128KB)
bl nand_read_whole @调用 nand_read_whole(),把 NAND闪存中的数据读入到 RAM中
tst r0, #0x0 @如果函数的返回值为 0,表示执行成功
beq ok_nand_read @执行内存比较，把 RAM中的前 4K内容与 NAND闪存中的前 4K内容进行比较，如果完全相同，则表示搬移成功
```

其中，nand_reset (), nand_read_whole()被加在 /board/wch2410/wch2410.c中。

支持 U-BOOT 命令设计

在 U-BOOT下对 nand闪存的支持主要是在命令行下实现对 nand闪存的操作。对 nand闪存实现的命令为：nand info(打印 nand Flash信息)、nand device(显示某个 nand闪存设备)、nand read(读取 nand闪存)、nand write(写 nand闪存)、nand erease(擦除 nand闪存)、nand bad(显示坏块)等。

用到的主要数据结构有：struct nand_flash_dev、struct nand_chip。前者包括主要的芯片型号、存储容量、设备 ID、I/O总线宽度等信息；后者是具体对 NAND闪存进行操作时用到的信息。

a. 设置配置选项

修改 /include/configs/wch2410.h,主要是在 CONFIG_COMMANDS中打开 CFG_CMD_NAND选项。定义 NAND闪存控制器在 SFR区中的起始寄存器地址、页面大小，定义 NAND闪存命令层的底层接口函数等。

b. 加入 NAND闪存芯片型号

在 /include/linux/mtd/ nand_ids.h中对如下结构体赋值进行修改：

```
static struct nand_flash_dev nand_flash_ids[] = {
......
{"Samsung K9F1208U0A", NAND_MFR_SAMSUNG, 0x76, 26, 0, 3, 0x4000, 0},
.......
}
```

这样对于该款 NAND闪存芯片的操作才能正确执行。

c. 编写 NAND闪存初始化函数

在 /board/wch2410/wch2410.c中加入 nand_init()函数。

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

```
/* 初始化 NAND闪存控制器，以及 NAND闪存芯片 */
nand_reset();
/* 调用 nand_probe()来检测芯片类型 */
printf ("%4lu MB\n", nand_probe(CFG_NAND_BASE) >> 20);
}
```

该函数在启动时被 start_armboot()调用。

最后重新编译 U-BOOT并将生成的 u-boot.bin烧入 NAND闪存中，目标板上电后从串口输出如下信息：

```
U-Boot 1.1.3 (Nov 14 2006 - 11:29:50)
U-Boot code: 33F80000 -> 33F9C9E4 BSS: -> 33FA0B28
RAM Configuration:
Bank #0: 30000000 64 MB
## Unknown Flash on Bank 0: ID 0xffff, Size = 0x00000000 = 0 MB
Flash: 0 kB
NAND: 64 MB
In:serial
Out: serial
Err: serial
Hit any key to stop autoboot: 0
wch2410 #
```

结语

以往将 U-BOOT移植到 ARM9平台中的解决方案主要针对的是 ARM9中的 NOR闪存，因为 NOR闪存的结构特点致使应用程序可以直接在其内部运行，不用把代码读到 RAM中，移植过程相对简单。从 NAND闪存中启动 U-BOOT的设计难点在于 NAND闪存需要把 U-BOOT的代码搬移到 RAM中，并要让 U-BOOT支持 NAND闪存的命令操作。本文介绍了实现这一设计的思路及具体程序。移植后，U-BOOT在嵌入式系统中运行良好。

参考文献

1 杜春雷．ARM体系结构与编程 [M]．北京 :清华大学出版社，2003

2 S3C2410 User’s Mannual[Z].Samsung

U-boot 给 kernel 传参数和 kernel 读取参数—struct tag (以及补充)

U-boot 会给 Linux Kernel 传递很多参数，如：串口， RAM ， videofb 等。而 Linux kernel 也会读取和处理这些参数。两者之间通过 struct tag 来传递参数。 U-boot 把要传递给 kernel 的东西保存在 struct tag 数据结构中，启动 kernel 时，把这个结构体的物理地址传给 kernel ；Linux kernel 通过这个地址，用 parse_tags 分析出传递过来的参数。

本文主要以 U-boot 传递 RAM 和 Linux kernel 读取 RAM 参数为例进行说明。

1 、u-boot 给 kernel 传 RAM 参数

./common/cmd_bootm.c 文件中， bootm 命令对应的 do_bootm 函数，当分析 uImage 中信息发现 OS 是 Linux 时 ， 调用 ./lib_arm/bootm.c 文件中的 do_bootm_linux 函数来启动 Linux kernel 。

在 do_bootm_linux 函数中：

```
void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
                ulong addr, ulong *len_ptr, int verify)
{
……
#if defined (CONFIG_SETUP_MEMORY_TAGS) || \
    defined (CONFIG_CMDLINE_TAG) || \
    defined (CONFIG_INITRD_TAG) || \
    defined (CONFIG_SERIAL_TAG) || \
    defined (CONFIG_REVISION_TAG) || \
    defined (CONFIG_LCD) || \
    defined (CONFIG_VFD)
      setup_start_tag (bd);      // 初始化 tag 结构体开始
#ifdef CONFIG_SERIAL_TAG
      setup_serial_tag (&params);
#endif
#ifdef CONFIG_REVISION_TAG
      setup_revision_tag (&params);
#endif
#ifdef CONFIG_SETUP_MEMORY_TAGS
      setup_memory_tags (bd);      // 设置 RAM 参数
#endif
#ifdef CONFIG_CMDLINE_TAG
      setup_commandline_tag (bd, commandline);
#endif
#ifdef CONFIG_INITRD_TAG
      if (initrd_start && initrd_end)
           setup_initrd_tag (bd, initrd_start, initrd_end);
#endif
#if defined (CONFIG_VFD) || defined (CONFIG_LCD)
      setup_videolfb_tag ((gd_t *) gd);
```

```
#endif
    setup_end_tag (bd);          // 初始化 tag 结构体结束
#endif
……
……
    theKernel (0, machid, bd->bi_boot_params);
// 传给 Kernel 的参数= (struct tag *) 型的 bd->bi_boot_params
//bd->bi_boot_params 在 board_init 函数中初始化如对于 at91rm9200 ，初始化在 at91rm9200dk.c 的 board_init 中进行： bd->bi_boot_params =PHYS_SDRAM + 0x100;
// 这个地址也是所有 taglist 的首地址，见下面的 setup_start_tag 函数
}
```

对于 setup_start_tag **和** setup_memory_tags **函数说明如下。**

函数 setup_start_tag **也在此文件中定义，如下：**

```
static void setup_start_tag (bd_t *bd)
{
    params = (struct tag *) bd->bi_boot_params;
// 初始化 (struct tag *) 型的全局变量 params 为 bd->bi_boot_params 的地址，之后的 setup tags 相关函数如下面的 setup_memory_tags 就把其它 tag 的数据放在此地址的偏移地址上。

    params->hdr.tag = ATAG_CORE;
    params->hdr.size = tag_size (tag_core);
    params->u.core.flags = 0;
    params->u.core.pagesize = 0;
    params->u.core.rootdev = 0;
    params = tag_next (params);
}
```

RAM **相关参数在** bootm.c **中的函数** setup_memory_tags **中初始化：**

```
static void setup_memory_tags (bd_t *bd)
{
    int i;
    for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
        params->hdr.tag = ATAG_MEM;
        params->hdr.size = tag_size (tag_mem32);
        params->u.mem.start = bd->bi_dram[i].start;
        params->u.mem.size = bd->bi_dram[i].size;
        params = tag_next (params);
    }              // 初始化内存相关 tag
```

```
}
```

2 、Kernel 读取 U-boot 传递的相关参数

对于 Linux Kernel ， ARM 平台启动时，先执行 arch/arm/kernel/head.S ，此文件会调用 arch/arm/kernel/head-common.S 中的函数，并最后调用 start_kernel ：

```
……
b     start_kernel
……
```

init/main.c 中的 start_kernel 函数中会调用 setup_arch 函数来处理各种平台相关的动作，包括了 u-boot 传递过来参数的分析和保存：

```
start_kernel()
{
……
      setup_arch(&command_line);
……
}
```

其中， setup_arch 函数在 arch/arm/kernel/setup.c 文件中实现，如下：

```
void __init setup_arch(char **cmdline_p)
{
      struct tag *tags = (struct tag *)&init_tags;
      struct machine_desc *mdesc;
      char *from = default_command_line;
      setup_processor();
      mdesc = setup_machine(machine_arch_type);
      machine_name = mdesc->name;
      if (mdesc->soft_reboot)
            reboot_setup("s");
      if (__atags_pointer)
// 指向各种 tag 起始位置的指针，定义如下：
//unsigned int __atags_pointer __initdata;
// 此指针指向 __initdata 段，各种 tag 的信息保存在这个段中。
            tags = phys_to_virt(__atags_pointer);
      else if (mdesc->boot_params)
            tags = phys_to_virt(mdesc->boot_params);
```

```
    if (tags->hdr.tag != ATAG_CORE)
        convert_to_tag_list(tags);
    if (tags->hdr.tag != ATAG_CORE)
        tags = (struct tag *)&init_tags;
    if (mdesc->fixup)
        mdesc->fixup(mdesc, tags, &from, &meminfo);
    if (tags->hdr.tag == ATAG_CORE) {
        if (meminfo.nr_banks != 0)
            squash_mem_tags(tags);
        save_atags(tags);
        parse_tags(tags);
// 处理各种 tags ，其中包括了 RAM 参数的处理。
// 这个函数处理如下 tags ：
__tagtable(ATAG_MEM, parse_tag_mem32);
__tagtable(ATAG_VIDEOTEXT, parse_tag_videotext);
__tagtable(ATAG_RAMDISK, parse_tag_ramdisk);
__tagtable(ATAG_SERIAL, parse_tag_serialnr);
__tagtable(ATAG_REVISION, parse_tag_revision);
__tagtable(ATAG_CMDLINE, parse_tag_cmdline);
    }
    init_mm.start_code = (unsigned long) &_text;
    init_mm.end_code  = (unsigned long) &_etext;
    init_mm.end_data  = (unsigned long) &_edata;
    init_mm.brk       = (unsigned long) &_end;
    memcpy(boot_command_line, from, COMMAND_LINE_SIZE);
    boot_command_line[COMMAND_LINE_SIZE-1] = '\0';
    parse_cmdline(cmdline_p, from);  // 处理编译内核时指定的 cmdline 或 u-boot 传递的 cmdline
    paging_init(&meminfo, mdesc);
    request_standard_resources(&meminfo, mdesc);
#ifdef CONFIG_SMP
    smp_init_cpus();
#endif
    cpu_init();
    init_arch_irq = mdesc->init_irq;
    system_timer = mdesc->timer;
    init_machine = mdesc->init_machine;
#ifdef CONFIG_VT
#if defined(CONFIG_VGA_CONSOLE)
```

```
        conswitchp = &vga_con;
#elif defined(CONFIG_DUMMY_CONSOLE)
        conswitchp = &dummy_con;
#endif
#endif
        early_trap_init();
}
```

对于处理 RAM **的** tag **，调用了** parse_tag_mem32 **函数：**

```
static int __init parse_tag_mem32(const struct tag *tag)
{
……
        arm_add_memory(tag->u.mem.start, tag->u.mem.size);
……
}
__tagtable(ATAG_MEM, parse_tag_mem32);
```

上述的 arm_add_memory **函数定义如下：**

```
static void __init arm_add_memory(unsigned long start, unsigned long size)
{
        struct membank *bank;
        size -= start & ~PAGE_MASK;

        bank = &meminfo.bank[meminfo.nr_banks++];
        bank->start = PAGE_ALIGN(start);
        bank->size = size & PAGE_MASK;
        bank->node = PHYS_TO_NID(start);
}
```

如上可见， parse_tag_mem32 **函数调用** arm_add_memory **函数把** RAM **的** start **和** size **等参数保存到了** meminfo **结构的** meminfo **结构体中。最后，在** setup_arch **中执行下面语句：**

```
        paging_init(&meminfo, mdesc);
```

对有 MMU **的平台上调用** arch/arm/mm/nommu.c **中的** paging_init **，否则调用** arch/arm/mm/mmu.c **中的** paging_init **函数。这里暂不分析** mmu.c **中的** paging_init **函数。**

3 、关于 U-boot 中的 bd 和 gd

U-boot 中有一个用来保存很多有用信息的全局结构体－－ gd_t （ global data 缩写），其中包括了 bd 变量，可以说 gd_t 结构体包括了 u-boot 中所有重要全局变量。最后传递给内核的参数，都是从 gd 和 bd 中来的，如上述的 setup_memory_tags 函数作用就是用 bd 中的值来初始化 RAM 相应的 tag 。

对于 ARM 平台这个结构体的定义大致如下：

include/asm-arm/global_data.h

```
typedef    struct      global_data  {
      bd_t        *bd;
      unsigned  long   flags;
      unsigned  long   baudrate;
      unsigned  long   have_console;  /*  serial_init()  was  called  */
      unsigned  long   reloc_off;       /*  Relocation  Offset  */
      unsigned  long   env_addr;       /*  Address    of  Environment  struct  */
      unsigned  long   env_valid;       /*  Checksum  of  Environment  valid?  */
      unsigned  long   fb_base;   /*  base  address  of  frame  buffer  */
      void        **jt;        /*  jump  table  */
}  gd_t;
```

在 U-boot 中使用 gd 结构之前要用先用宏 DECLARE_GLOBAL_DATA_PTR 来声明。这个宏的定义如下：

include/asm-arm/global_data.h

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

从这个宏的定义可以看出， gd 是一个保存在 ARM 的 r8 寄存器中的 gd_t 结构体的指针。

说明：本文的版本为 U-boot-1.3.4 、Linux-2.6.28 ，平台是 ARM 。

/补充一下：

来自 :http://hi.baidu.com/armfans/blog/item/306cd5035f24ff084afb514b.html

bootloader巧妙地利用函数指针及传参规范将 R0:0x0, R1:机器号， R2:参数地址传递给内核．由于 R0, R1比较简单，不需要再作说明．需要花点时间了解的是 R2寄存器．

R2寄存器传递的是一个指针，这个指针指向一个 TAG区域．UBOOT和 Linux内核之间正是通过这个扩展了的 TAG区域来进行复杂参数的传递，如 command line, 文件系统信息等等，用户也可以扩展这个 TAG来进行更多参数的传递． TAG区域存放的地址，也就是 R2的值,是在 /board /yourboard/youboard.c里的 board_init函数中初始化的，

如在 UB4020中初始化为： gd->bd->bi_boot_params = 0x30000100;，这是一个绝对地址.

TAG区的结构比较简单，可以视为一个一个 TAG的排列（数组？），每一个 TAG传递一种特定类型的参数．各种系统 TAG的定义可以参考 ./include/asm-arm/setup.h.

下面是一个 TAG区的例子：

0x30000100 00000005 54410001 00000000 00000000
0x30000110 00000000 0000000F 54410009 746F6F72
0x30000120 65642F3D 61722F76 7220306D 6F632077
0x30000130 6C6F736E 74743D65 2C305379 30303639
0x30000140 696E6920 6C2F3D74 78756E69 EA006372
0x30000150 00000004 54420005 30300040 00200000
0x30000160 00000000 00000000

我们可以看到一共有三个 TAG:

第一个 TAG的长度是 5个字，类型是 ATAG_CORE（54410001），有三个元素，均为全零． TAG区必须以这个 TAG开头．

第二个 TAG的长度是 F个字，类型是 ATAG_CMDLINE（54410009），这是一个字符串，是向内核传递的 kernel command line

第三个 TAG的长度是 4个字，类型是 ATAG_INITRD2（54410005），有两个元素，第一个是 start:30300040（30300000+40），第二个是 size:200000（2M）

如果说还有第四个 TAG，那就是末尾的两个全零，这是 TAG结束的标志．

这些 TAG是在 ./lib_arm/arm_linux.c中的 do_bootm_linux函数中建立起来的．具体建立哪些 TAG，由相应的控制宏决定．具体可以参考相应代码．例子中第一个 TAG是起始 TAG，如果环境变量中有 bootargs，则建立第二个 TAG，如果 bootm有两个参数（引导文件系统），则会读取文件系统头部的必要信息，建立第三个 TAG.

内核启动后，将根据 R2寄存器的值找到这些 TAG，并根据 TAG类型，调用相应的处理函数进行处理，从而获取内核运行的必要信息．

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 s3c24x0
```

参考上面 2 行，添加下面 2 行。

```
fs2410_config    :    unconfig
@./mkconfig $(@:_config=) arm arm920t fs2410 NULL s3c24x0
```

(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。

TABLE 6: SOFTWARE COMMAND SEQUENCE

Command Sequence	1st Bus Write Cycle		2nd Bus Write Cycle		3rd Bus Write Cycle		4th Bus Write Cycle		5th Bus Write Cycle		6th Bus Write Cycle	
	Addr[1]	Data[2]	Addr[1]	Data[2]	Addr[1]	Data[2]	Addr[1]	Data[2]	Addr[1]	Data[2]	Addr[1]	Data[2]
Word-Program	5555H	AAH	2AAAH	55H	5555H	A0H	WA[3]	Data				
Sector-Erase	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	SA_X[4]	30H
Block-Erase	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	BA_X[4]	50H
Chip-Erase	5555H	AAH	2AAAH	55H	5555H	80H	5555H	AAH	2AAAH	55H	5555H	10H
Erase-Suspend	XXXXH	B0H										
Erase-Resume	XXXXH	30H										
Query Sec ID[5]	5555H	AAH	2AAAH	55H	5555H	88H						
User Security ID Word-Program	5555H	AAH	2AAAH	55H	5555H	A5H	WA[6]	Data				
User Security ID Program Lock-Out	5555H	AAH	2AAAH	55H	5555H	85H	XXH[6]	0000H				
Software ID Entry[7,8]	5555H	AAH	2AAAH	55H	5555H	90H						
CFI Query Entry	5555H	AAH	2AAAH	55H	5555H	98H						
Software ID Exit[9,10] /CFI Exit/Sec ID Exit	5555H	AAH	2AAAH	55H	5555H	F0H						
Software ID Exit[9,10] /CFI Exit/Sec ID Exit	XXH	F0H										

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。