

# u-boot 在 mini2440 上的移植

uboot 版本: u-boot-2015.10

目标平台: mini2440

文档版本: 1.0

# 目录

u-boot 源码整体框架.....	3
下载解压源码.....	3
u-boot 目录结构.....	3
u-boot 组织结构.....	4
编译下载 uboot.....	5
u-boot 启动流程.....	5
stage1 代码结构.....	6
stage1 代码分析.....	7
vectors.S.....	7
start.S.....	8
lowlevel_init.S.....	11
crt0.S.....	12
board_f.c.....	17
relocate.S.....	25
stage2.....	26
u-boot 移植步骤.....	27
建立移植所需的目录结构.....	27
1.建立开发板目录.....	27
2.修改 Makefile.....	27
3.修改 Kconfig.....	27
4.建立配置文件.....	28
5.简单编译.....	29
配置文件修订.....	29
时钟修改.....	30
SDRAM 移植.....	31
向量表重定向.....	32
norflash 驱动.....	32
nandflash 驱动.....	33
DM9000 网卡移植.....	35
启动 uboot.....	36
使用 kermit 下载程序到 SDRAM 中执行.....	39
发布 uboot.....	41

# u-boot 源码整体框架

## 下载解压源码

```
下载 uboot 源码，u-boot-latest.tar.bz2
tar -jxf u-boot-latest.tar.bz2
解压后得到源码
jiaduo@jiaduo-syberos:~/temp/xx$ ls
u-boot-2015.10  u-boot-latest.tar.bz2
```

## u-boot 目录结构

Uboot 源码结构如下：

~/temp/xx/u-boot-2015.10\$ls

api

arch

board

common

config.mk

configs

disk

doc

drivers

dts

examples

fs

include

Kbuild

Kconfig

lib

Licenses

MAINTAINERS

MAKEALL

Makefile

net

post

README

scripts

snapshot.commit

test

tools

~/temp/xx/u-boot-2015.10\$

ready

ssh2: AES-256-CTR

24, 26 24 Rows, 67 Cols

VT100

CAP NUM

arch	<div>Architecture specific files</div> <div>存放和 CPU 架构有关的代码，每个子目录都会有对应的架构子目录（ARM,ATMEL,MIPS..），没个对应 CPU 目录下包含 start.S, interrupt.c, cpu.c</div> <div>start.S 是 U-boot 启动时执行的第一个文件，它主要做最早其的系统初始化，代码重定向和设置系统堆栈，为进入 U-boot 第二阶段的 C 程序奠定基础。</div> <div>interrupt.c 设置系统的各种中断和异常</div> <div>cpu.c 初始化 CPU、设置指令 Cache 和数据 Cache 等</div>
board	<div>已经支持的所有开发板相关文件，其中包含 SDRAM 初始化代码、Flash 底层驱动、板级初始化文件。也是移植需要重点进行修改的文件夹</div>

common	与处理器体系结构无关的通用代码，u-boot 的 main 函数文件 main.c U-boot 的命令解析代码/common/command.c 所有命令的上层代码 cmd_*.c Uboot 环境变量处理代码 env_*.c、等都位于该目录下
configs	存放 uboot 所支持的开发板的配置文件，uboot 的编译配置也依赖于这个目录下的文件，都是以 xxx_defconfig 结尾的文件，xxx 一般对应于 board 目录下的开发板名称。通过 make xxx_defconfig 来将配置写入到.config 中。
disk	进行磁盘分区的的代码
doc	存放针对 uboot 各个部分的说明文档，文件命名一般以 REDEME.开头，包含各个驱动模块的配置说明和对应架构的说明等等。
drivers	包含几乎所有外围芯片的驱动， 网卡、 USB、串口、 LCD、Nand Flash nor flash 等等，移植时需要针对性的修改
dtb	包含两个文件 Kconfig 和 Makefile，用来建立 uboot fdt
fs	Uboot 文件系统支持代码，yaffs2，fat32 等
net	网络协议栈，arp 协议，dhcp，域名解析 dns，网络文件系统 nfs，控制报文 icmp，简单文件传输 tftp 协议
Include	Uboot 编译使用的头文件
lib	Uboot 使用的功能函数库，RSA 加解密算法，CRC8/16/32 校验，哈希函数，随机数，字符串处理等
post test scripts example tools	存放测试用带的代码，功能测试和 power 测试、以及 uboot 执行需要的脚本文件、和应用示例代码。很少使用和查看。tools 目录存放了一些独立于 uboot 的应用软件。如图形化配置工具 mconf，linux 启动内核制作 mkimage。
Makefile MAKEALL config.mk Kconfig Kbuild README	控制整个编译过程的主 Makefile 文件和规则文件，以及帮助文档

## u-boot 组织结构

u-boot 的源码原来越像 linux，甚至很多的代码都是 linux 源码中直接拿来的。组织结构也和 linux 一致，包括配置流程 make menuconfig，编译。Kconfig 和 Makefile 来进行代码的配置和选择性的编译，和 linux 的组织结构一样。Kconfig 定义了一个配置菜单，使用 tools 下的 mconf 来读取散落在各个目

录下的 **Kconfig** 来读取整个 **uboot** 的配置，从而获取对应的宏定义，根据宏定义。**Make** 通过读取各个目录的 **Makefile** 来根据这些宏确定要编译的目标文件。

**Kconfig** 语法参考

`$less doc/README.kconfig`

**Makefile** 语法参考

**GNU Make** 使用手册

## 编译下载 uboot

# u-boot 启动流程

大多数 **bootloader** 都分为 **stage1** 和 **stage2** 两部分，**u-boot** 也不例外。依赖于 **CPU** 体系结构的代码（如设备初始化代码等）通常都放在 **stage1** 且可以用汇编语言来实现，而 **stage2** 则通常用 **C** 语言来实现，这样可以实现复杂的功能，而且有更好的可读性和移植性。

**stage1** 其主要代码功能：

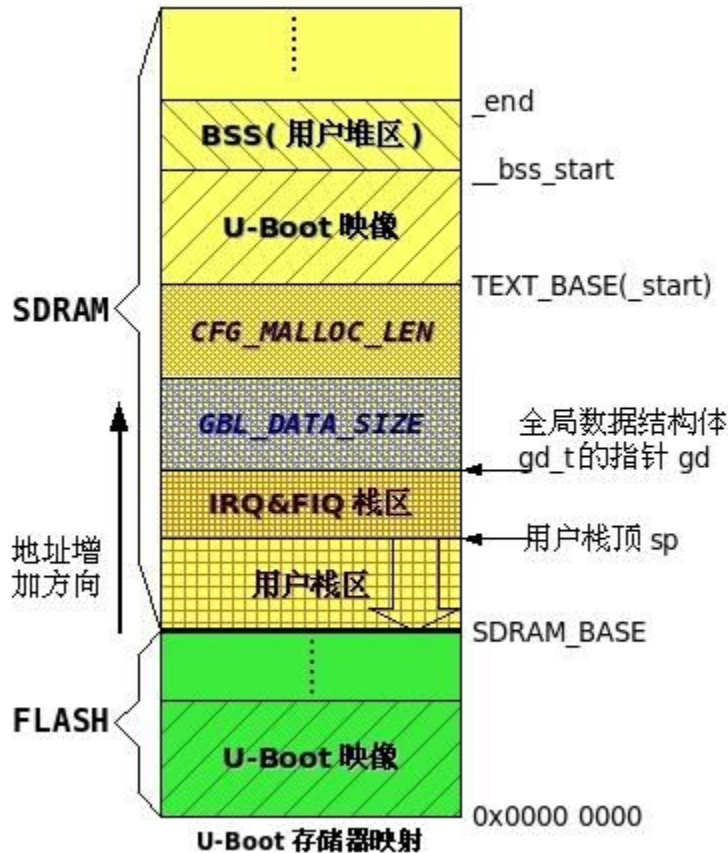
- （1）定义入口。由于一个可执行的 **Image** 必须有一个入口点，并且只能有一个全局入口，通常这个入口放在 **ROM (Flash)** 的 **0x0** 地址，因此，必须通知编译器以使其知道这个入口，该工作可通过修改连接器脚本来完成。
- （2）设置异常向量（**Exception Vector**）。
- （3）设置 **CPU** 的速度、时钟频率及终端控制寄存器。
- （4）初始化内存控制器。
- （5）将 **ROM** 中的程序复制到 **RAM** 中。
- （6）初始化堆栈。
- （7）转到 **RAM** 中执行，该工作可使用指令 **ldr pc** 来完成。

**stage2** 主要代码功能：

- （1）调用一系列的初始化函数。
- （2）初始化 **Flash** 设备。
- （3）初始化系统内存分配函数。
- （4）如果目标系统拥有 **NAND** 设备，则初始化 **NAND** 设备。

- (5) 如果目标系统有显示设备，则初始化该类设备。
- (6) 初始化相关网络设备，填写 IP、MAC 地址等。
- (7) 进去命令循环（即整个 **boot** 的工作循环），接受用户从串口输入的命令，然后进行相应的工作。

uboot 进入 stage2 后运行的内存分布：



Uboot 在现在的新版本以后都会在 stage1 阶段将自己拷贝到 SDRAM 的地址尾端，拷贝地址是 uboot 自己计算出来的，并且编译的过程中使用了 `-fpic` 选项，保证拷贝后可以重新定向到目标地址执行。

## stage1 代码结构

u-boot 的 stage1 代码包括下面几个文件：

`arch/arm/lib/vectors.S` -- 中断向量表

`arch/arm/cpu/arm920t/start.S` -- stage1 执行代码

`board/samsung/smdk2410/lowlevel_init.S` -- sdram 初始化函数

`arch/arm/lib/crt0.S` -- 被 start.S 调用的 `_main` 入口

`arch/arm/lib/relocate.S` -- 重定向代码函数

`common/board_f.c` -- stage1 初始化工作

上面几个代码的调用关系

vectors.S

```
b reset
...
```



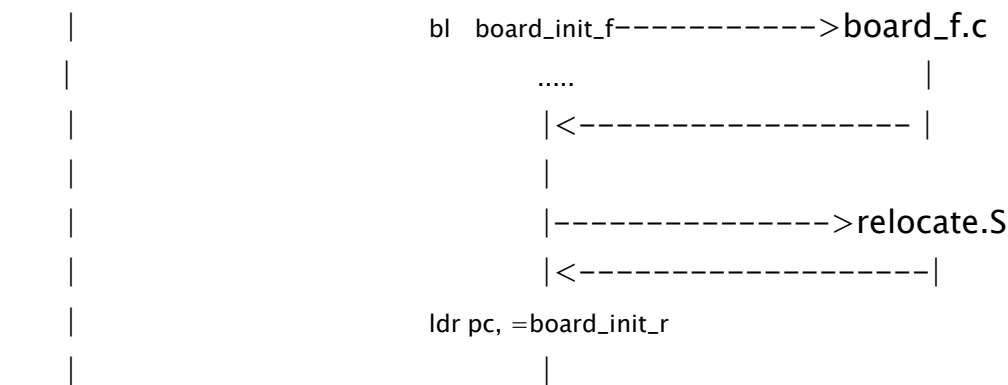
```
start.S
.globl reset
```

```
reset:
...
bl lowlevel_init----->lowlevel_init.S
```



```
...
...
```

```
bl main----->crt0.S
```



Never !!!  
Board\_r.c  
(stage1 over! , stage2 start)

stage1 代码分析

vectors.S

`_start:`

`_start` 其实是 `uboot` 第一个执行的代码段，大多数认为是 `start.S`，其实也可以这么说，因为 `_start` 存放中断向量表，向量表的第一条执行的事 `b reset (start.S)`。

```
#ifdef CONFIG_SYS_DV_NOR_BOOT_CFG
    .word    CONFIG_SYS_DV_NOR_BOOT_CFG
#endif

    b    reset    复位异常中断
    ldr pc, _undefined_instruction 未定义指令异常
    ldr pc, _software_interrupt    软中断
    ldr pc, _prefetch_abort        预取指令异常中断
    ldr pc, _data_abort            数据异常中断
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq
```

## `start.S`

```
.globl    reset

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

切换 CPU 到特权模式。特权模式可以访问所有硬件受控资源。相对于其他的模式，SVC 模式可以访问的资源更多。

`uboot` 作用，其要做的事情是初始化系统相关硬件资源，需要获取尽量多的权限，以方便操作硬件，初始化硬件。

同时也是引导 linux 内核所必需的的条件之一

<http://www.arm.linux.org.uk/developer/booting.php>



- CPU register settings
  - r0 = 0.
  - r1 = machine type number discovered in (3) above.
  - r2 = physical address of tagged list in system RAM.
- CPU mode
  - All forms of interrupts must be disabled (IRQs and FIQs.)
  - The CPU must be in SVC mode. (A special exception exists for Angel.)
- Caches, MMUs
  - The MMU must be off.
  - Instruction cache may be on or off.
  - Data cache must be off and must not contain any stale data.

```
#ifdef CONFIG_S3C24X0
```

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

关闭看门狗，有看门狗在，需要一直刷新看门狗定时器，在初始化时不需要的，也是为了保证代码不被看门狗中断

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

```
#  define pWTCON    0x15300000
```

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

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

```
#else
```

```
#  define pWTCON    0x53000000
```

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

```
#  define INTSUBMSK   0x4A00001C
```

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

```
# endif
```

```
    ldr r0, =pWTCON
```

```
    movr1, #0x0
```

```
    str r1, [r0]
```

关闭所有中断，**uboot** 代码执行都是通过查询处理任务的，不需要中断，至少现在不需要。开启中断会造成异常中断跳转，启动代码无法有效顺序执行！

```
    movr1, #0xffffffff
```

```
    ldr r0, =INTMSK
```

```

    str r1, [r0]
# if defined(CONFIG_S3C2410)
    ldr r1, =0x3ff
    ldr r0, =INTSUBMSK
    str r1, [r0]
# endif

```

设置时钟频率，使用默认的 **FCLK** 时钟，这里移植需要根据芯片类型修改

```

/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
    ldr r0, =CLKDIVN
    movr1, #3
    str r1, [r0]
#endif /* CONFIG_S3C24X0 */

```

是否跳过对 **cpu** 的一些初始化，包括禁止 **cache**、**SDRAM** 初始化，如果 **uboot** 实在 **SDRAM** 中直接运行的，需要定义 **CONFIG\_SKIP\_LOWLEVEL\_INIT**

```

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl cpu_init_crit
#endif

bl _main 这里跳转到 crt0.S

#ifndef CONFIG_SKIP_LOWLEVEL_INIT
cpu_init_crit:
/*
 * flush v4 I/D caches
 */
    movr0, #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
 */
    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.
- \*/

movip, lr **cpu\_init\_crit** 是 **BL** 跳转过来的 **lr**，存放了返回地址，如果再次使用 **BL**，**lr** 会被覆盖，所以要先保存 **lr**，至于为什么是 **ip**，没什么，只是一个通用寄存器而已

bl lowlevel\_init 跳转到 **lowlevel\_init.S** 执行函数并返回

movlr, ip

movpc, lr

#endif /\* CONFIG\_SKIP\_LOWLEVEL\_INIT \*/

## lowlevel\_init.S

.globl lowlevel\_init

lowlevel\_init:

ldr r0, =SMRDATA

ldr r1, =CONFIG\_SYS\_TEXT\_BASE

sub r0, r0, r1

ldr r1, =BWSCON /\* Bus Width Status Controller \*/

add r2, r0, #13\*4

0:

ldr r3, [r0], #4

str r3, [r1], #4

cmp r2, r0

bne 0b

**R3**: 存储着 **R0** 所指向的配置字

**R0**: 存放配置字的首地址，并每次操作后+4，一共操作 13 次，之后 **R0=R2**

**R2**: 配置字的末地址

其实是将 **SMRDATA** 相对地址处的 13 个配置字节拷贝到 **cpu** 内部对应的寄存器地址中

为什么是相对地址??? 也就是为什么要 **sub r0, r0, r1**??

因为 **SMRDATA** 地址是根据链接地址分配的，链接地址=**CONFIG\_SYS\_TEXT\_BASE**

而这段代码实际运行在 0 地址处 (**SR0M**)，所以 **SMRDATA** 的地址并不是实际的配置字地址，而是

**SDRDATA-CONFIG\_SYS\_TEXT\_BASE**

/\* everything is fine now \*/

```

movpc, lr
.ltorg
/* the literal pools origin */
SMRDATA:
.word
(0+(B1_BWSCON<<4)+(B2_BWSCON<<8)+(B3_BWSCON<<12)+(B4_BWSCON<<16)+(B5_BW
SCON<<20)+(B6_BWSCON<<24)+(B7_BWSCON<<28))
.word
((B0_Tacs<<13)+(B0_Tcos<<11)+(B0_Tacc<<8)+(B0_Tcoh<<6)+(B0_Tah<<4)+(B0_Tacp<
<2)+(B0_PMC))
.word
((B1_Tacs<<13)+(B1_Tcos<<11)+(B1_Tacc<<8)+(B1_Tcoh<<6)+(B1_Tah<<4)+(B1_Tacp<
<2)+(B1_PMC))
.word
((B2_Tacs<<13)+(B2_Tcos<<11)+(B2_Tacc<<8)+(B2_Tcoh<<6)+(B2_Tah<<4)+(B2_Tacp<
<2)+(B2_PMC))
.word
((B3_Tacs<<13)+(B3_Tcos<<11)+(B3_Tacc<<8)+(B3_Tcoh<<6)+(B3_Tah<<4)+(B3_Tacp<
<2)+(B3_PMC))
.word
((B4_Tacs<<13)+(B4_Tcos<<11)+(B4_Tacc<<8)+(B4_Tcoh<<6)+(B4_Tah<<4)+(B4_Tacp<
<2)+(B4_PMC))
.word
((B5_Tacs<<13)+(B5_Tcos<<11)+(B5_Tacc<<8)+(B5_Tcoh<<6)+(B5_Tah<<4)+(B5_Tacp<
<2)+(B5_PMC))
.word ((B6_MT<<15)+(B6_Trtd<<2)+(B6_SCAN))
.word ((B7_MT<<15)+(B7_Trtd<<2)+(B7_SCAN))
.word ((REFEN<<23)+(TREFMD<<22)+(Trp<<20)+(Trc<<18)+(Tchr<<16)+REFCNT)
.word 0x32
.word 0x30
.word 0x30

```

上面的变量是配置信息，移植会需要修改

## crt0.S

进行了一些简单的初始化后，SDRAM 已经可以使用。接着就是对 uboot 在 SDRAM 中运行做准备，重点包括：

初始化 uboot 运行结构参数 (global data 数据)

拷贝和重定向代码到 SDRAM 中

crt0.S 的任务顺序:

1. 设置堆栈指针、global data 指针。为运行 C 函数 board\_init\_f() 做准备。

这里并不是具备了 C 语言的所有运行环境, bss, rw 段的数据仍然不可用 (代码运行在 flash 中) 只能使用已经定义的常量。

2. 执行 board\_init\_f(), 为程序在 SDRAM 中运行做准备, 要完成 global data 的填充, 包括重定向地址、运行的栈指针, 运行时的 global data 地址。

3. 调用 relocate\_code, 获取 board\_init\_f 计算出的重定向地址, 进行代码拷贝和重定向

4. 跳转到 SDRAM 的 board\_init\_r 执行

ENTRY(\_main)

/\*

\* Set up initial C runtime environment and call board\_init\_f(0).

\*/

#if defined(CONFIG\_SPL\_BUILD) && defined(CONFIG\_SPL\_STACK)

ldr sp, =(CONFIG\_SPL\_STACK)

#else

ldr sp, =(CONFIG\_SYS\_INIT\_SP\_ADDR)

这里设置 SP 指向一个预留的空间, 地址为 CONFIG\_SYS\_INIT\_SP\_ADDR, 是靠 SDRAM 基地址的位置定义在配置文件中

#define CONFIG\_SYS\_INIT\_SP\_ADDR (CONFIG\_SYS\_SDRAM\_BASE + 0x1000 - \

GENERATED\_GBL\_DATA\_SIZE)

CONFIG\_SYS\_SDRAM\_BASE 在配置文件中定义为 0x30000000

#endif

#if defined(CONFIG\_CPU\_V7M) /\* v7M forbids using SP as BIC destination \*/

movr3, sp

bic r3, r3, #7

movsp, r3

#else

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

#endif

movr2, sp

sub sp, sp, #GD\_SIZE /\* allocate one GD above SP \*/

为 global data 分配 GD\_SIZE 大小的空间, 其实觉得这里 uboot 代码有些问题, 因为上面分配 SP 指针的时候已经预留出来 GENERATED\_GBL\_DATA\_SIZE 大小的空间, 现在又预留 GD\_SIZE 感觉有些重复。

```

#if defined(CONFIG_CPU_V7M)    /* v7M forbids using SP as BIC destination */
    movr3, sp
    bic r3, r3, #7
    movsp, r3

```

```

#else

```

```

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

```

```

#endif

```

```

    movr9, sp    /* GD is above SP */

```

将全局变量结构体首地址保存到 **R9** 寄存器中,此时 **R2** 存放全局变量结构体的末地址

```

    movr1, sp

```

```

    movr0, #0

```

```

clr_gd:

```

```

    cmpr1, r2    /* while not at end of GD */

```

```

#if defined(CONFIG_CPU_V7M)

```

```

    itt lo

```

```

#endif

```

```

    strlo r0, [r1]    /* clear 32-bit GD word */

```

```

    addlo r1, r1, #4    /* move to next */

```

```

    blo clr_gd

```

将 **R1-R2** 所指向的地址空间清空, 也就是 **global data** 结构

```

#if defined(CONFIG_SYS_MALLOC_F_LEN)。。 没定义

```

```

    sub sp, sp, #CONFIG_SYS_MALLOC_F_LEN

```

```

    str sp, [r9, #GD_MALLOC_BASE]

```

```

#endif

```

```

    /* mov r0, #0 not needed due to above code */

```

```

    bl board_init_f

```

跳转到 **board\_init\_f** 执行, 以 **r0** 为参数传递, **R0=0**, 具体细节会在 **board\_r.c** 中讲解

```

#if ! defined(CONFIG_SPL_BUILD)

```

```

/*

```

```

 * Set up intermediate environment (new sp and gd) and call

```

```

 * relocate_code(addr_moni). Trick here is that we'll return

```

```

 * 'here' but relocated.

```

```

 */

```

```

    ldr sp, [r9, #GD_START_ADDR_SP]    /* sp = gd->start_addr_sp */

```

**SP** 指向计算出的重定义的堆栈

```

#if defined(CONFIG_CPU_V7M)    /* v7M forbids using SP as BIC destination */

```

```

    movr3, sp

```

```

    bic r3, r3, #7

```

```

    movsp, r3

```

```

#else
    bic sp, sp, #7    /* 8-byte alignment for ABI compliance */
#endif
    ldr r9, [r9, #GD_BD]    /* r9 = gd->bd */
    sub r9, r9, #GD_SIZE    /* new GD is below bd */

```

R9 指向新计算的 **global data** 地址，因为经过 **board\_init\_f** 函数后，旧的 **global data** 已经拷贝到新的 **global data** 地址。**gd->bd** 结构体指针正好在 **global data** 的地址尾部，也就是 **GD** 在 **bd** 下面，所以 **gd->bd - GD\_SIZE = new GD**。

但是问题来了，**board\_init\_f** 执行后，新的 **GD** 指针已经保存在 **global\_data** 结构体的程序 **new\_gd** 中，为什么不直接方位这个 **new\_gd** 呢？

原因是根本没有定义这个 **NEW\_GD** 的宏啊!!!

```

    adr lr, here
    ldr r0, [r9, #GD_RELOC_OFF]    /* r0 = gd->reloc_off */
    add lr, lr, r0

```

这里为调用 **relocate\_code** 做准备，存贮 **here** 地址的返回值，保证 **relocate\_code** 代码调用返回到 **here** 处。**gd->reloc\_off** 是重定向后的地址偏移值。**Relocate\_code** 返回后就会进入真正的 **SDRAM** 中执行

```

#if defined(CONFIG_CPU_V7M)
    orr lr, #1    /* As required by Thumb-only */
#endif
    ldr r0, [r9, #GD_RELOCADDR]    /* r0 = gd->relocaddr */
    b relocate_code

```

跳转到 **relocate\_code** 执行，并将 **r0 (gd->relocaddr)** 作为参数传递

here:

```

/*
 * now relocate vectors
 */

```

```

    bl relocate_vectors

```

重定向向量表，这个函数是 **\_\_weak** 修饰的函数，可以重新定义，移植的时候根据自己的 **CPU** 类型修改，因为 **uboot** 不使用中断，所以也可以去掉

```

/* Set up final (full) environment */

```

```

    bl c_runtime_cpu_setup /* we still call old routine here */
#endif
#if !defined(CONFIG_SPL_BUILD) || defined(CONFIG_SPL_FRAMEWORK)
# ifdef CONFIG_SPL_BUILD
    /* Use a DRAM stack for the rest of SPL, if requested */
    bl spl_relocate_stack_gd
    cmpr0, #0

```

```

    movne    sp, r0
#endif
    ldr     r0, =__bss_start /* this is auto-relocated! */

#ifdef CONFIG_USE_ARCH_MEMSET
    ldr     r3, =__bss_end    /* this is auto-relocated! */
    movr1, #0x00000000      /* prepare zero to clear BSS */

    subs    r2, r3, r0      /* r2 = memset len */
    bl     memset
#else
    ldr     r1, =__bss_end    /* this is auto-relocated! */
    movr2, #0x00000000      /* prepare zero to clear BSS */

clbss_l:cmp    r0, r1      /* while not at end of BSS */
#ifdef CONFIG_CPU_V7M
    itt     lo
#endif
    strlo    r2, [r0]      /* clear 32-bit BSS word */
    addlo    r0, r0, #4     /* move to next */
    blo     clbss_l
#endif

```

以上代码清空程序的 **bss** 段，重定向后这些地址 **\_\_bss\_end** **\_\_bss\_start** 也都被重定向了

```

#ifdef ! defined(CONFIG_SPL_BUILD)
    bl coloured_LED_init
    bl red_led_on
#endif
    /* call board_init_r(gd_t *id, ulong dest_addr) */
    mov     r0, r9          /* gd_t */
    ldr     r1, [r9, #GD_RELOCADDR] /* dest_addr */
    /* call board_init_r */
    ldr     pc, =board_init_r /* this is auto-relocated! */

```

将新的 **global data** 地址，和重定向后的地址作为参数传递给 **board\_init\_r**

```

void board_init_r(gd_t *new_gd, ulong dest_addr)
    /* we should not return here. */
#endif

```

ENDPROC(\_main)

上面一直用到了几个宏 **GD\_RELOCADDR** **GD\_SIZE** **GD\_RELOC\_OFF** 等，这些在哪定义呢，这些宏如



果不编译是无法在源码中找到的，我们进行 **make** 编译后在对应的 **include/generated/generic-asm-offsets.h** 中会看到这些定义

```
#ifndef __GENERIC_ASM_OFFSETS_H__
#define __GENERIC_ASM_OFFSETS_H__
/*
 * DO NOT MODIFY.
 *
 * This file was generated by Kbuild
 */

#define GENERATED_GBL_DATA_SIZE 176 /* (sizeof(struct global_data) + 15) & ~15  @ */
#define GENERATED_BD_INFO_SIZE 80 /* (sizeof(struct bd_info) + 15) & ~15
    @ */
#define GD_SIZE 168 /* sizeof(struct global_data) @ */
#define GD_BD 0 /* offsetof(struct global_data, bd) @ */
#define GD_RELOCADDR 44 /* offsetof(struct global_data, relocaddr) @ */
#define GD_RELOC_OFF 64 /* offsetof(struct global_data, reloc_off) @ */
#define GD_START_ADDR_SP 60 /* offsetof(struct global_data, start_addr_sp)
    @ */
#endif
```

## board\_f.c

Board 其实主要是填充 **gd** 指针空间，也就是 **R9** 指针所对应的 **global data** 结构体。为后面的 **relocate\_code** 和 **board\_init\_r** 提供环境。比如计算重定向地址、新的堆栈地址、新的 **global data** 的位置。还会进行一些基础的初始化工作，时钟、串口、定时器这些基础的外设。我们马上就可以看到打印信息了

我们只关注如下的函数信息：

```
void board_init_f(ulong boot_flags)
{
```

```
.....
```

```
    gd->flags = boot_flags;
    gd->have_console = 0;
```

```
    if (initcall_run_list(init_sequence_f))
        hang();
```

这个函数实际是调用了一个函数指针数组，并按顺序执行数组中每个指针指向的函数，如果出错，就 **hang()**，让我们看看 **init\_sequence\_f** 都做了什么

```
.....
```

```
.....
```

```
}
```

init\_sequence\_f-----

发现进行了很多的函数调用，其实细看，很多函数都是有宏定义的。实际执行的函数并不多。我们把需要执行的函数留下，不需要执行的都用横线删了。。。。。重要的我都用蓝色标示了

```
static init_fnc_t init_sequence_f[] = {
```

```
#ifdef CONFIG_SANDBOX
```

```
——setup_ram_buf,
```

```
#endif
```

```
    setup_mon_len,
```

函数执行如下：

```
#if defined(__ARM__) || defined(__MICROBLAZE__)
```

```
    gd->mon_len = (ulong)&__bss_end - (ulong)_start;
```

设置 gd->mon\_len，也就是代码长度，代码包含 RO,RW,ZI，

RO 起始地址 = \_start;ZI 的尾地址 = \_\_bss\_end。所以

这里的长度就是 RO+RW+ZI

```
#ifdef CONFIG_OF_CONTROL
```

```
——fdtdec_setup,
```

```
#endif
```

```
#ifdef CONFIG_TRACE
```

```
——trace_early_init,
```

```
#endif
```

```
    initf_malloc,
```

这个函数主体也是带有宏定义的，其实对我们来说是个空函数

```
#if defined(CONFIG_MPC85xx) || defined(CONFIG_MPC86xx)
```

```
——/* TODO: can this go into arch_cpu_init()? */
```

```
——probecpu,
```

```
#endif
```

```
#if defined(CONFIG_X86) && defined(CONFIG_HAVE_FSP)
```

```
——x86_fsp_init,
```

```
#endif
```

```
    arch_cpu_init,          /* basic arch cpu dependent setup */
```

这个函数啥都没做

```
__weak int arch_cpu_init(void)
```

```
{
```

```
    return 0;
```

```
}
```

这里注意这个函数是\_\_weak 修饰过的，叫做“弱符号”表示这个函数可以被覆盖，比如我们重新定义了一个函数 int arch\_cpu\_init(void)，之前的函数就不会执行，而执行我们自己的函数，

简单介绍下“强符号”“弱符号”

- ① 同名的强符号只能有一个，否则编译器报"重复定义"错误。
- ② 允许一个强符号和多个弱符号，但定义会选择强符号的。
- ③ 当有多个弱符号相同时，链接器选择占用内存空间最大的那个。

在 C 语言中，函数和初始化的全局变量（包括显示初始化为 0）是强符号，未初始化的全局变量是弱符号。

明白了吧

mark\_bootstage,

就是 mark 了一下

initf\_dm,

这个函数受两个宏控制，CONFIG\_DM CONFIG\_SYS\_MALLOC\_F\_LEN，我们都没定义，所以也是空

arch\_cpu\_init\_dm,

也是\_\_weak 修饰的函数，里面什么都没有

#if defined(CONFIG\_BOARD\_EARLY\_INIT\_F)

board\_early\_init\_f,

#endif

这个函数对应的开发板的一些初始化，我们这里 CONFIG\_BOARD\_EARLY\_INIT\_F 定义了，所以有必要关心下，对应的函数在 board/samsung/smdk2410/smdk2410.c。不同的开发板对应的可能不同。

这个函数干嘛的呢？

是初始化 CPU 的时钟包括 FCLK PCLK HCLK USBCLK，有了合适的时钟我们才能配置串口啊！移植需要修改

/\* TODO: can any of this go into arch\_cpu\_init()? \*/

#if defined(CONFIG\_PPC) && !defined(CONFIG\_8xx\_CPUCLK\_DEFAULT)

——get\_clocks,—— /\* get CPU and bus clocks (etc.) \*/

#if defined(CONFIG\_TQM8xxL) && !defined(CONFIG\_TQM866M) \

——&& !defined(CONFIG\_TQM885D)

——adjust\_sdram\_tbs\_8xx,

#endif

——/\* TODO: can we rename this to timer\_init()? \*/

——init\_timebase,

#endif

#if defined(CONFIG\_ARM) || defined(CONFIG\_MIPS) || \

defined(CONFIG\_BLACKFIN) || defined(CONFIG\_NDS32)

timer\_init, /\* initialize timer \*/

#endif

初始化定时器，用来定时用，具体对应

arch/arm/cpu/arm920t/s3c24x0/timer.c

#ifdef CONFIG\_SYS\_ALLOC\_DPRAM

#if !defined(CONFIG\_CPM2)

```

——dpram_init,
#endif
#endif
#if defined(CONFIG_BOARD_POSTCLK_INIT)
——board_postclk_init,
#endif
#ifdef CONFIG_FSL_ESDHC
——get_clocks,
#endif
#ifdef CONFIG_M68K
——get_clocks,
#endif
    env_init,      /* initialize environment */
环境变量的初始化，比如 bootargs bootcmd bootdelay 都会在这里初始化
#if defined(CONFIG_8xx_CPUCLK_DEFAULT)
——/* get CPU and bus clocks according to the environment variable */
——get_clocks_866,
——/* adjust sdram refresh rate according to the new clock */
——sdram_adjust_866,
——init_timebase,
#endif
    init_baud_rate, /* initialize baudrate settings */
    serial_init,     /* serial communications setup */
初始化串口和对应的波特率，这些都对应 board 目录下的文件，具体：
Init_baud_rate 没有做实际的操作，只是将
gd->baudrate = getenv_ulong("baudrate", 10, CONFIG_BAUDRATE);
Serial_init 才是真正的初始化，这个函数指向了 serial_s3c24x0.c 中的 default_serial_console
通过配置 CONFIG_SERIALX 来选择对应的串口初始化函数，这里是 CONFIG_SERIAL1 对应 UART0
    console_init_f, /* stage 1 init of console */
设置一个标志位 gb->console，代表我们有终端了
#ifdef CONFIG_SANDBOX
——sandbox_early_getopt_check,
#endif
#ifdef CONFIG_OF_CONTROL
——fdtdec_prepare_fdt,
#endif
    display_options, /* say that we are here */
    display_text_info, /* show debugging info if required */
打印信息到串口，没啥解释的

```

```

#if defined(CONFIG_MPC8260)
——prt_8260_rsr,
——prt_8260_clks,
#endif /* CONFIG_MPC8260 */
#if defined(CONFIG_MPC83xx)
——prt_83xx_rsr,
#endif
#if defined(CONFIG_PPC) || defined(CONFIG_M68K)
——checkcpu,
#endif
    print_cpuinfo, /* display cpu info (and speed) */
#if defined(CONFIG_MPC5xxx)
——prt_mpc5xxx_clks,
#endif /* CONFIG_MPC5xxx */
#if defined(CONFIG_DISPLAY_BOARDINFO)
——show_board_info,
#endif

```

INIT\_FUNC\_WATCHDOG\_INIT

这个也是没有定时看门狗初始化的宏，所以也不关心

```

#if defined(CONFIG_MISC_INIT_F)
——misc_init_f,
#endif
——INIT_FUNC_WATCHDOG_RESET
#if defined(CONFIG_HARD_I2C) || defined(CONFIG_SYS_I2C)
——init_func_i2c,
#endif
#if defined(CONFIG_HARD_SPI)
——init_func_spi,
#endif

```

announce\_dram\_init,

打印“**DRAM:**”表示下面要 **DRAM** 初始化，并打印 **DRAM** 的信息

```

/* TODO: unify all these dram functions? */
#if defined(CONFIG_ARM) || defined(CONFIG_X86) || defined(CONFIG_NDS32) || \
    defined(CONFIG_MICROBLAZE) || defined(CONFIG_AVR32)
    dram_init, /* configure available RAM banks */
#endif

```

主要是获取 **DRAM** 大小并存放到

**gd->ram\_size** 中

```

#if defined(CONFIG_MIPS) || defined(CONFIG_PPC) || defined(CONFIG_M68K)

```

```

——init_func_ram,
#endif
#ifdef CONFIG_POST
——post_init_f,
#endif
——INIT_FUNC_WATCHDOG_RESET
#if defined(CONFIG_SYS_DRAM_TEST)
——testdram,
#endif /* CONFIG_SYS_DRAM_TEST */
——INIT_FUNC_WATCHDOG_RESET

#ifdef CONFIG_POST
——init_post,
#endif
    INIT_FUNC_WATCHDOG_RESET
    /*
     * Now that we have DRAM mapped and working, we can
     * relocate the code and continue running from DRAM.
     *
     * Reserve memory at end of RAM for (top down in that order):
     *  - area that won't get touched by U-Boot and Linux (optional)
     *  - kernel log buffer
     *  - protected RAM
     *  - LCD framebuffer
     *  - monitor code
     *  - board info struct
     */
    setup_dest_addr,

```

设置 **gd->ram\_top** 指向 DRAM 的有效大小，并 **gd->relocaddr = gd->ram\_top**  
 从下面开始 **gd->relocaddr** 指针会最终指向代码要重定向的地址

```

#ifdef CONFIG_BLACKFIN || defined(CONFIG_NIOS2)
——/* Blackfin u-boot monitor should be on top of the ram */
——reserve_uboot,
#endif
#ifdef CONFIG_LOGBUFFER && !defined(CONFIG_ALT_LB_ADDR)
——reserve_logbuffer,
#endif
#ifdef CONFIG_PRAM
——reserve_pram,

```

```

#endif
    reserve_round_4k,
    预留 4K bytes 的空间
    #if !(defined(CONFIG_SYS_ICACHE_OFF) && defined(CONFIG_SYS_DCACHE_OFF)) && \
    -----defined(CONFIG_ARM)
    -----reserve_mmu,
#endif
#ifdef CONFIG_LCD
    -----reserve_lcd,
#endif
    -----reserve_trace,
    -----/* TODO: Why the dependency on CONFIG_8xx? */
    #if defined(CONFIG_VIDEO) && (!defined(CONFIG_PPC) || defined(CONFIG_8xx)) && \
    -----!defined(CONFIG_ARM) && !defined(CONFIG_X86) && \
    -----!defined(CONFIG_BLACKFIN) && !defined(CONFIG_M68K)
    -----reserve_video,
#endif
    #if !defined(CONFIG_BLACKFIN) && !defined(CONFIG_NIOS2)
        reserve_uboot,
    #endif
    预留出 uboot 代码内存空间, 此时 gd->relocaddr 指向了要重定向的地址, 也就是 uboot 在内存中运
    行地址, 不在改变, 此时 gd->start_addr_sp = gd->relocaddr。之后开始设置 gd->start_addr_sp。
    #ifndef CONFIG_SPL_BUILD
        reserve_malloc,
        reserve_board,
    #endif

```

预留 malloc 所需的内存, 和 bd 结构体内存

```

    setup_machine,
    设置机器码, 需要定义 CONFIG_MACH_TYPE, 这里没有定义这个变量, 在之后的 stage2 中 board_init
    中也会设置这个值。

```

```

    reserve_global_data,
    预留 global data 结构体内存
    reserve_fdt,

```

没有使用 FDT 所以这里不关心

```

    reserve_arch,
    __weak 修饰函数, 空函数
    reserve_stacks,

```

预留 IRQ FIQ 堆栈, (如果不适用 IRQ 默认是不会分配空间的)。保证堆栈 16 字节对齐, 具体为什么 16 字节对齐。。待确定

```

    setup_dram_config,
    show_dram_config,
setup_dram_config 执行空函数
show_dram_config 显示打印 dram 信息
#ifdef CONFIG_PPC || defined(CONFIG_M68K)
—— setup_board_part1,
—— INIT_FUNC_WATCHDOG_RESET
—— setup_board_part2,
#endif
    display_new_sp,

```

通过上面的操作，将新的堆栈地址打印出来

```

#ifdef CONFIG_SYS_EXTBDINFO
—— setup_board_extra,
#endif
—— INIT_FUNC_WATCHDOG_RESET
—— reloc_fdt,
    setup_reloc,

```

这个函数设置  $gd \rightarrow reloc\_off = gd \rightarrow relocaddr - CONFIG\_SYS\_TEXT\_BASE$ ; 也就是重定向代码的偏移地址。同时将旧的 **global data** 拷贝到计算出的新的 **global data** 地址, `memcpy(gd->new_gd, (char *)gd, sizeof(gd_t));`

这样，如果 **R9** 指向新 **global data** 地址，就可以使用 **gd** 指针访问新的 **global data** 了。

```

#ifdef CONFIG_X86 || defined(CONFIG_ARC)
—— copy_uboot_to_ram,
—— clear_bss,
—— do_elf_reloc_fixups,
#endif
#ifdef !defined(CONFIG_ARM) && !defined(CONFIG_SANDBOX)
—— jump_to_copy,
#endif
—— NULL,
};

```

如果注意看上面的代码，我们知道 **R9** 寄存器存放了 **global data** 结构体地址，**board\_init\_f** 中的函数直接使用 **gd** 这个指针来方位 **global data**，**R9** 和 **gd** 指针的关系是什么？

发现 **board\_init\_f** 开头有个宏定义 **DECLARE\_GLOBAL\_DATA\_PTR**

再次查找定义

```

#ifdef CONFIG_ARM64
#define DECLARE_GLOBAL_DATA_PTR    register volatile gd_t *gd asm ("x18")
#else
#define DECLARE_GLOBAL_DATA_PTR    register volatile gd_t *gd asm ("r9")

```



其实是定义 **gd** 为一个寄存器变量。并且是 **gd\_t\*** 类型。

## relocate.S

这个文件主要是拷贝代码到 **SDRAM**，重定向代码和中断向量表。重定向代码就是对 **SDRAM** 中函数变量的地址就行修订，所依据的是 **GOT** 表，**gcc** 通过 **-pic** 编译后支持这种重定向  
这里针对 **relocate\_code**

**ENTRY(relocate\_code)**

```
ldr r1, =__image_copy_start /* r1 <- SRC &__image_copy_start */
subs r4, r0, r1             /* r4 <- relocation offset */
beq relocate_done          /* skip relocation */
ldr r2, =__image_copy_end   /* r2 <- SRC &__image_copy_end */
```

**R0** 存放了 **relocate\_addr**，**r1** 存放了要拷贝的起始地址，如果相等不需要拷贝，如果不等拷贝 **R1~R2** 的地址范围到 **R0** 地址处，具体拷贝工作是 **copy\_loop** 完成的

**copy\_loop:**

```
ldmia r1!, {r10-r11}        /* copy from source address [r1] */
stmia r0!, {r10-r11}        /* copy to target address [r0] */
cmpr1, r2                    /* until source end address [r2] */
blo copy_loop
```

拷贝完毕！

```
/*
 * fix .rel.dyn relocations
 */
ldr r2, =__rel_dyn_start /* r2 <- SRC &__rel_dyn_start */
ldr r3, =__rel_dyn_end   /* r3 <- SRC &__rel_dyn_end */
```

**fixloop:**

```
ldmia r2!, {r0-r1}          /* (r0,r1) <- (SRC location,fixup) */
and r1, r1, #0xff
cmpr1, #23                  /* relative fixup? */
bne fixnext
```

```

/* relative fix: increase location by offset */
add r0, r0, r4
ldr r1, [r0]
add r1, r1, r4
str r1, [r0]

```

fixnext:

```

cmpr2, r3
blo fixloop

```

上面的代码对 **DRAM** 中的 **uboot** 代码重定向，将 **DRAM** 中代码地址+**R4** (**offset**)。具体的细节参考

<http://blog.csdn.net/skyflying2012/article/details/37660265>

relocate\_done:

```

#ifdef __XSCALE__
/*
 * On xscale, icache must be invalidated and write buffers drained,
 * even with cache disabled – 4.2.7 of xscale core developer's manual
 */
mcr p15, 0, r0, c7, c7, 0 /* invalidate icache */
mcr p15, 0, r0, c7, c10, 4 /* drain write buffer */
#endif

/* ARMv4– don't know bx lr but the assembler fails to see that */

#ifdef __ARM_ARCH_4__
movpc, lr
#else
bx lr
#endif

ENDPROC(relocate_code)

```

## stage2

到了这一阶段，代码就没有那些汇编了，都是 C 语言完成，任务也相对变得复杂。

主要还是对外设驱动的、环境变量的初始化，然后跳转到 `main_loop` 函数  
`Main_loop` 就是接受交互命令并执行命令了。这部分很少进行修改。也没有复杂的流程，不再赘述

## u-boot 移植步骤

### 建立移植所需的目录结构

#### 1.建立开发板目录

```
cd board/samsung/  
mkdir mini2440  
cp -rf smdk2410/* mini2440/  
mv mini2440/smdk2410.c mini2440/mini2440.c
```

#### 2.修改 Makefile

```
$vim mini2440/Makefile
```

```
obj-y    := mini2440.o  
obj-y    += lowlevel_init.o
```

#### 3.修改 Kconfig

```
$vim mini2440/Kconfig  
+if TARGET_MINI2440  
+  
+config SYS_BOARD  
+  default "mini2440"  
+  
+config SYS_VENDOR
```

```
+   default "samsung"
+
+config SYS_SOC
+   default "s3c24x0"
+
+config SYS_CONFIG_NAME
+   default "mini2440"
+
+endif
```

修改为如下：

```
SYSBOARD:mini2440
SYS_VENDOR: samsung
SYS_SOC: s3c24x0
SYS_CONFIG_NAME: mini2440
```

上面几个宏确定下面几个参数：

开发板代码目录	board/samsung(SYS_VENDOR)/mini2440(SYSBOARD)/
soc 目录	arch/arm/cpu/arm920t/s3c24x0(SYS_SOC)
配置文件	include/configs/mini2440(SYS_CONFIG_NAME).h

```
$vim arch/arm/Kconfig
+config TARGET_MINI2440
+   bool "Support mini2440"
+   select CPU_ARM920T
+
+   config TARGET_ASPENITE
+       bool "Support aspenite"
+       select CPU_ARM926EJS
@@ -769,6 +773,7 @@ source "board/phytec/pcm051/Kconfig"
source "board/phytec/pcm052/Kconfig"
source "board/ppcag/bg0900/Kconfig"
source "board/samsung/smdk2410/Kconfig"
+source "board/samsung/mini2440/Kconfig"
```

## 4.建立配置文件

```
cp include/configs/smdk2410.h include/configs/mini2440.h
cp configs/smdk2410_defconfig configs/mini2440_defconfig
修改 mini2440_defconfig
```

```
+CONFIG_ARM=y
+CONFIG_TARGET_MINI2440=y
+CONFIG_SYS_PROMPT="MINI2440 # "
Mini2440.h 也需要修改，在移植过程中会介绍
```

## 5.简单编译

```
$make mini2440_defconfig
```

```
$make CROSS_COMPILE=arm-linux-
```

```
OBJCOPY u-boot.srec
```

```
OBJCOPY u-boot.bin
```

```
CFG      u-boot.cfg
```

我们会使用 u-boot.bin

## 配置文件修订

```
$vim include/configs/mini2440.h
```

Soc 定义

```
+#define CONFIG_S3C24X0      /* This is a SAMSUNG S3C24x0-type SoC */
+#define CONFIG_S3C2440      /* specifically a SAMSUNG S3C2440 SoC */ /* modified!!! */
+#define CONFIG_MINI2440      /* on a SAMSUNG MINI2440 Board */ /* modified!!! */
```

网卡支持

```
+#define CONFIG_DRIVER_DM9000
+#define CONFIG_DM9000_BASE 0x20000300 /* nGCS4 */
+#define DM9000_DATA      (CONFIG_DM9000_BASE+4)
+#define DM9000_IO CONFIG_DM9000_BASE
+#define CONFIG_DM9000_NO_SROM
+/* #define CONFIG_DM9000_DEBUG */
```

串口支持

```
+#define CONFIG_S3C24X0_SERIAL
+#define CONFIG_SERIAL1      1 /* we use SERIAL 1 on SMDK2410 */
+#define CONFIG_BAUDRATE      115200
```

网络地址

```
+
+#define CONFIG_NETMASK      255.255.255.0
+#define CONFIG_IPADDR      10.0.0.110
+#define CONFIG_SERVERIP      10.0.0.1
```

Norflash 定义

```
+ * FLASH and environment organization
+ */
+
+#define CONFIG_SYS_FLASH_CFI
+#define CONFIG_FLASH_CFI_DRIVER
+#define CONFIG_FLASH_SHOW_PROGRESS 45
+
+#define CONFIG_SYS_MAX_FLASH_BANKS 1
+#define CONFIG_SYS_FLASH_BANKS_LIST { CONFIG_SYS_FLASH_BASE }
+#define CONFIG_SYS_MAX_FLASH_SECT (35) /* modified !!!!! */
+
```

```

#define CONFIG_ENV_ADDR                (CONFIG_SYS_FLASH_BASE + 0x1E0000) /* modified !!!
*/
#define CONFIG_ENV_IS_IN_FLASH
#define CONFIG_ENV_SIZE                0x10000
/* allow to overwrite serial and ethaddr */
#define CONFIG_ENV_OVERWRITE

```

## Nand 定义

```

+ * NAND configuration
+ */
#ifdef CONFIG_CMD_NAND
#define CONFIG_NAND_S3C2410
#define CONFIG_SYS_S3C2410_NAND_HWECC
#define CONFIG_SYS_MAX_NAND_DEVICE 1
#define CONFIG_SYS_NAND_BASE        0x4E000000
#endif

```

## Autoboot

```

/* autoboot */
#define CONFIG_BOOTDELAY 5
#define CONFIG_BOOT_RETRY_TIME -1
#define CONFIG_RESET_TO_RETRY
#define CONFIG_ZERO_BOOTDELAY_CHECK
#define CONFIG_BOOTCOMMAND "version;bdfinfo"

```

只有 CONFIG\_BOOTCOMMAND 不为空的时候 CONFIG\_BOOTDELAY 才会起作用，倒计时  
这里 CONFIG\_BOOTCOMMAND 只是一个演示

## 时钟修改

### start.S

```

# if defined(CONFIG_S3C2440) /* modified !!!!! */
+   ldr r1, =0x7fff
+   ldr r0, =INTSUBMSK
+   str r1, [r0]
# endif

# if defined(CONFIG_S3C2410)
/* FCLK:HCLK:PCLK = 1:2:4 */
/* default FCLK is 120 MHz ! */
-   ldr r0, =CLKDIVN
+   ldr r0, =CLKDIVN
+   mov r1, #3
+   str r1, [r0]
# endif
+
# if defined(CONFIG_S3C2440)
+   /* FCLK:HCLK:PCLK = 1:4:8 HDIVN=2, PDIVN=1 */
+   /* set FCLK 405MHz ! */
+   ldr r0, =CLKDIVN
+   mov r1, #5
+   str r1, [r0]

```

```

+ /* HDIVN != 0 ,set asynchronous bus mode */
+ mrc p15,0,r0,c1,c0,0
+ orr r0,r0,#0xc0000000
+ mcr p15,0,r0,c1,c0,0
+ /* set FLCK in bord_early_init_f */
+ /* ... */
+## endif

```

## Mini2440.c

```

+##if defined(CONFIG_S3C2440)
+
+##define M_MDIV 0x7f          /* Fout = 405MHz, Fin = 12MHz */
+##define M_PDIV 0x2
+##define M_SDIV 0x1
+
+##else
+
+##if FCLK_SPEED==0            /* Fout = 203MHz, Fin = 12MHz for Audio */
+##define M_MDIV 0xc3
+##define M_PDIV 0x4
+##define M_SDIV 0x1
+##elif FCLK_SPEED==1         /* Fout = 202.8MHz */
+##define M_MDIV 0xa1
+##define M_PDIV 0x3
+##define M_SDIV 0x1
+
+##endif
+
+##endif /* endif CONFIG_S3C2440 */

```

S3C2440 比 S3C2410 时钟要快，设置正确的时钟分频和主频率，避免超过最大允许值

## SDRAM 移植

### Lowlevel\_init.S

```

+##ifdef CONFIG_S3C2440
+
+##define REFEN          0x1 /* Refresh enable */
+##define TREFMD         0x0 /* CBR(CAS before RAS)/Auto refresh */
+##define Trp            0x0 /* 2clk */
+##define Trc            0x3 /* 7clk */
+##define Tchr           0x2 /* 3clk */
+##define REFCNT         1260 /* period=7.8us, HCLK=101Mhz, (2048+1-7.8*101) */
+                          (2048+1-7.8*101) */
+
+##else
+
+##define REFEN          0x1 /* Refresh enable */
+##define TREFMD         0x0 /* CBR(CAS before RAS)/Auto refresh */
+##define Trp            0x0 /* 2clk */
+##define Trc            0x3 /* 7clk */
+##define Tchr           0x2 /* 3clk */
+##define REFCNT         1113 /* period=15.6us, HCLK=60Mhz, (2048+1-15.6*60) */

```

```
+
+endif
```

根据 SDRAM 设置对应的参数

## 向量表重定向

```
@@ -49,15 +49,15 @@ ENTRY(relocate_vectors)
    * CP15 c1 V bit gives us the location of the vectors:
    * 0x00000000 or 0xFFFF0000.
    */
-   ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
-   mrc p15, 0, r2, c1, c0, 0 /* V bit (bit[13]) in CP15 c1 */
-   ands r2, r2, #(1 << 13)
-   ldreq r1, =0x00000000 /* If V=0 */
-   ldrne r1, =0xFFFF0000 /* If V=1 */
-   ldmia r0!, {r2-r8,r10}
-   stmia r1!, {r2-r8,r10}
-   ldmia r0!, {r2-r8,r10}
-   stmia r1!, {r2-r8,r10}
+#   ldr r0, [r9, #GD_RELOCADDR] /* r0 = gd->relocaddr */
+#   mrc p15, 0, r2, c1, c0, 0 /* V bit (bit[13]) in CP15 c1 */
+#   ands r2, r2, #(1 << 13)
+#   ldreq r1, =0x00000000 /* If V=0 */
+#   ldrne r1, =0xFFFF0000 /* If V=1 */
```

S3C2440 不支持重定向（只支持高地址低地址），所以去掉重定向代码

## norflash 驱动

Norflash 使用的是标准的 CFI 接口芯片 Am29LV160DB，uboot 对 norflash CFI 接口支持很好。可以自动检测 norflash 的信息

在配置文件中加入定义

```
+#define CONFIG_SYS_FLASH_CFI
+#define CONFIG_FLASH_CFI_DRIVER
+#define CONFIG_FLASH_SHOW_PROGRESS 45
+
+#define CONFIG_SYS_MAX_FLASH_BANKS 1
+#define CONFIG_SYS_FLASH_BANKS_LIST { CONFIG_SYS_FLASH_BASE }
```

这里我们将环境变量存放到 norflash 中。又定义了如下参数

```
+#define CONFIG_SYS_MAX_FLASH_SECT (35) /* modified !!!! */
+
+#define CONFIG_ENV_ADDR (CONFIG_SYS_FLASH_BASE + 0x1E0000) /* modified !!! */
*/
+#define CONFIG_ENV_IS_IN_FLASH
+#define CONFIG_ENV_SIZE 0x10000
```

Am29LV160DB 一共有 35 个扇区，我们放到最后的扇区 SA33，1E0000-1EFFFF = 64Kbytes



SA33	1	1	1	1	0	X	X	X	64/32	1E0000–1EFFFF	F0000–F7FFF
SA34	1	1	1	1	1	X	X	X	64/32	1F0000–1FFFFFF	F8000–FFFFFF

## Mini2440.c

如果我们定义了

```
#define CONFIG_FLASH_CFI_LEGACY
```

那么会调用 mini2440.c 中的

```
board_flash_get_legacy
```

因为我们的 norflash 是标准 cfi 接口，可以去掉，不去也不会执行

Note: norflash 移植过程中发现开启的 DEBUG，norflash 无法正确写入数据，会出现写入超时，原因是 uboot 从 norflash 读到超时间是 ms，却当做 us 处理，又进行了 (us+999)/1000 的操作，从而导致 timeout 设置时间较短，开启后会打印信息从而消耗时间导致超时。

```
/* round up when converting to ms */
info->write_tout = (tmp + 999) / 1000;
info->flash_id = FLASH_MAN_CFI;
if ((info->interface == FLASH_CFI_X8X16) &&
    (info->chipwidth == FLASH_CFI_BY8)) {
    /* XXX - Need to test on x8/x16 in parallel. */
    info->portwidth >>= 1;
}
```

解决方法：

1. 不定义 DEBUG
2. 增加超时时间，不进行 us 到 ms 转换。不过需要修改硬件无关源码，不建议！

## nandflash 驱动

因为 S3C2440 和 S3C2410 之间的很大差别就是：S3C2410 的 Nand Flash 控制器只支持 512B+16B 的 Nand Flash，而 S3C2440 还支持 2KB+64B 的大容量 Nand Flash。所以在 Nand Flash 控制器上寄存器和控制流程上的差别很明显，底层驱动代码的修改也是必须的。具体的差别还是需要对比芯片数据手册的，下面是关于 Nand Flash 底层驱动代码的修改：

s3c24x0\_nand.c

```
#include <asm/arch/s3c24x0_cpu.h>
#include <asm/io.h>

#define S3C2410_NFCONF_EN (1<<15)
#define S3C2410_NFCONF_512BYTE (1<<14)
#define S3C2410_NFCONF_4STEP (1<<13)
#define S3C2410_NFCONF_INITECC (1<<12)
#define S3C2410_NFCONF_nFCE (1<<11)
#define S3C2410_NFCONF_TACLS(x) ((x)<<8)
#define S3C2410_NFCONF_TWRPH0(x) ((x)<<4)
#define S3C2410_NFCONF_TWRPH1(x) ((x)<<0)
#define NF_BASE CONFIG_SYS_NAND_BASE
#define S3C2410_NFCONT_EN (1<<0)
```

```

#define S3C2410_NFCONT_INITECC      (1<<4)
#define S3C2410_NFCONT_nFCE        (1<<1)
#define S3C2410_NFCONF_TACLS(x)    ((x)<<12)
#define S3C2410_NFCONF_TWRPH0(x)   ((x)<<8)
#define S3C2410_NFCONF_TWRPH1(x)   ((x)<<4)
+
#define S3C2410_ADDR_NALE 0x08
#define S3C2410_ADDR_NCLE 0x0c

-#define S3C2410_ADDR_NALE 4
-#define S3C2410_ADDR_NCLE 8

#ifdef CONFIG_NAND_SPL

@@ -38,33 +38,34 @@ static void nand_read_buf(struct mtd_info
}
#endif

+ulong IO_ADDR_W = NF_BASE;
static void s3c24x0_hwcontrol(struct mtd_info *mtd, int cmd, unsigned int ctrl)
{
- struct nand_chip *chip = mtd->priv;
+ /* struct nand_chip *chip = mtd->priv; */
struct s3c24x0_nand *nand = s3c24x0_get_base_nand();

debug("hwcontrol(): 0x%02x 0x%02x\n", cmd, ctrl);

if (ctrl & NAND_CTRL_CHANGE) {
-   ulong IO_ADDR_W = (ulong)nand;
+   IO_ADDR_W = (ulong)nand;

if (!(ctrl & NAND_CLE))
IO_ADDR_W |= S3C2410_ADDR_NCLE;
if (!(ctrl & NAND_ALE))
IO_ADDR_W |= S3C2410_ADDR_NALE;

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

if (ctrl & NAND_NCE)
-   writel(readl(&nand->nfconf) & ~S3C2410_NFCONF_nFCE,
-          &nand->nfconf);
+   writel(readl(&nand->nfconf) & ~S3C2410_NFCONF_nFCE,
+          &nand->nfconf);
else
-   writel(readl(&nand->nfconf) | S3C2410_NFCONF_nFCE,
-          &nand->nfconf);
+   writel(readl(&nand->nfconf) | S3C2410_NFCONF_nFCE,
+          &nand->nfconf);
}

if (cmd != NAND_CMD_NONE)
-   writeb(cmd, chip->IO_ADDR_W);
+   writeb(cmd, IO_ADDR_W);
}

static int s3c24x0_dev_ready(struct mtd_info *mtd)
@@ -79,7 +80,7 @@ void s3c24x0_nand_enable_hwecc(struct mt
{
struct s3c24x0_nand *nand = s3c24x0_get_base_nand();

```

```

    debug("s3c24x0_nand_enable_hwecc(%p, %d)\n", mtd, mode);
-   writel(readl(&nand->nfconf) | S3C2410_NFCONF_INITECC, &nand->nfconf);
+   writel(readl(&nand->nfcont) | S3C2410_NFCONF_INITECC, &nand->nfcont);
}

static int s3c24x0_nand_calculate_ecc(struct mtd_info *mtd, const u_char *dat,
@@ -125,16 +126,19 @@ int board_nand_init(struct nand_chip *na
    twrph0 = CONFIG_S3C24XX_TWRPH0;
    twrph1 = CONFIG_S3C24XX_TWRPH1;
#else
-   tacls = 4;
-   twrph0 = 8;
-   twrph1 = 8;
+   tacls = 1;
+   twrph0 = 4;
+   twrph1 = 2;
#endif

-   cfg = S3C2410_NFCONF_EN;
+   cfg = 0;
    cfg |= S3C2410_NFCONF_TACLS(tacls - 1);
    cfg |= S3C2410_NFCONF_TWRPH0(twrph0 - 1);
    cfg |= S3C2410_NFCONF_TWRPH1(twrph1 - 1);
    writel(cfg, &nand_reg->nfconf);
+   /* cfg = (readl(&nand_reg->nfcont) | S3C2410_NFCONF_EN); */
+   cfg = (0<<13) | (0<<12) | (0<<10) | (0<<9) | (0<<8) | (0<<6) | (0<<5) | (1<<4) | (0<<1) | (1<<0);
+   writel(cfg, &nand_reg->nfcont);

    /* initialize nand_chip data structure */
    nand->IO_ADDR_R = (void *)&nand_reg->nfddata;

```

Nand flash 的 s3c24x0\_hwcontrol 有个 bug，就是修改了 chip->IO\_ADDR\_W 的值，所以会导致写错误，这里用 IO\_ADDR\_W 全局变量来代替这个函数中的操作。

## DM9000 网卡移植

修改配置文件

```

#define CONFIG_DRIVER_DM9000
#define CONFIG_DM9000_BASE 0x20000300 /* nGCS4 */
#define DM9000_DATA (CONFIG_DM9000_BASE+4)
#define DM9000_IO CONFIG_DM9000_BASE
#define CONFIG_DM9000_NO_SROM
/* #define CONFIG_DM9000_DEBUG */
DM9000 连接在 bank4 上，通过 lowlevel_init.s 修改 bank4 的设置。设置使用 16 位数据线，并且使用 wait 信号（看电路连接）
#define B4_BWCON (DW16 + WAIT) /* DM9000 use */

```

DM9000 因为用 nGCS4 做片选，所以地址为 0x20000000。其实只用了 1 个 bit。这个地址只做片选用，只要 nGCS4 对应的 bit 是 1 就行。

又因为 DM9000 中 IO base = TXD[2:0]\*10H + 300H

也就是  $DM9000\_IO = TXD[2: 0] * 10H + 300H$

这里  $CONFIG\_DM9000\_BASE = 0x20000300$

$DM9000\_IO$  和  $DM9000\_DATA$  只有一个 bit 的区别，来区分数据线上的 16bit 是数据还是地址。

这个 bit 连接到 S3C2440 的 LADDR2 上，所以  $DM9000\_DATA = DM9000\_IO + 0x4$ ;

修改 mini2440.c 的初始化函数，增加了对 dm9000 的初始化

```
+ #ifdef CONFIG_CMD_NET
+ int board_eth_init(bd_t *bis)
+ {
+     int rc = 0;
+ /*
+ #ifdef CONFIG_CS8900
+     rc = cs8900_initialize(0, CONFIG_CS8900_BASE);
+ #endif
+ */
+     rc = dm9000_initialize(bis);
+
+     return rc;
+ }
+ #endif
```

dm9000x.c 中有一段 link 的超时代码，这段代码会造成响应慢的问题，可以去掉

```
;
- if (i == 10000) {
-     printf("could not establish link\n");
-     return 0;
+ if (i == 1000) {
+     printf("could not establish link\n");
+     return 0;
+     break;
+ }
```

## 启动 uboot

经过上面的配置我们使用

`make mini2440_defconfig`

`make CROSS_COMPILE=arm-linux-`

将得到的 `u-boot.bin` 使用 `j-flash` 下载到开发板，并按 **F9** 运行。

```
U-Boot 2015.10 (Dec 17 2015 - 15:06:25 +0800)

CPUID: 32440001
FCLK:      405 MHz
HCLK:     101.250 MHz
PCLK:      50.625 MHz
DRAM:      64 MiB
WARNING: Caches not enabled
Flash: 2 MiB
NAND:      256 MiB
In:        serial
Out:       serial
Err:       serial
Net:       dm9000
Hit any key to stop autoboot:  0
MINI2440 #
```

Ready      ssh2: AES-256-CTR      24, 12 24 Rows, 61 Cols      VT100      CAP NUM

可以看到相关的信息打印时钟、DRAM、FLASH、NAND、NET 最后进入命令交互接口 。  
具体的 **uboot** 命令可以查看官网的手册。  
我们运行几个命令测试我们的移植

## saveenv

保存我们当前变量到 norflash

```
MINI2440 # saveenv
Saving Environment to Flash...
Un-Protected 1 sectors
Erasing Flash...
. done
Erased 1 sectors
Writing to Flash... 9....8....7....6....5....4....3....2....1
....done
Protected 1 sectors
MINI2440 #
```

Ready      ssh2: AES-256-CTR      24, 12 24 Rows, 61 Cols      VT100      CAP NUM

## nand read 30000000 0 10000

从 nand 0 地址读取 4k 字节到 SDRAM 地址 30000000 处

```
MINI2440 # nand read 30000000 0 1000

NAND read: device 0 offset 0x0, size 0x1000
4096 bytes read: OK
MINI2440 #
```

Ready      ssh2: AES-256-CTR      24, 12 24 Rows, 61 Cols      VT100

```
nand write 30000000 0 1000
```

将 SDRAM 30000000 处的 4k 字节写入 nand 的 0 地址处

```
MINI2440 # nand write 30000000 0 1000

NAND write: device 0 offset 0x0, size 0x1000
4096 bytes written: OK
MINI2440 #
```

```
Ready                ssh2: AES-256-CTR    24, 12 24 Rows, 61 Cols  VT100
```

### nand erase.chip

擦除整个 nand

```
MINI2440 # nand erase.chip

NAND erase.chip: device 0 whole chip
Skipping bad block at 0x02000000

Skipping bad block at 0x07d60000

Skipping bad block at 0x0aaa0000

Erasing at 0xffe0000 -- 100% complete.
OK
MINI2440 #
```

下面我们测试一下网卡驱动

我们开发板和 PC 连接，我们的开发板地址信息是

```
#define CONFIG_NETMASK      255.255.255.0
#define CONFIG_IPADDR      10.0.0.110
#define CONFIG_SERVERIP    10.0.0.1
```

但是我们没有 mac 地址，所以要先设置 mac 地址环境变量

```
set ethaddr da:15:db:01:01:01
```

我们设置 PC 地址为 10.0.0.1 255.255.255.0

然后在开发板上执行 ping 和 tftp 命令（PC 已经配置好 tftpd-hpa）

使用 tftp 命令获取文件 test.img 并放到 SDRAM 30000000 处

```
MINI2440 # ping 10.0.0.1
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: da:15:db:01:01:01
could not establish link
Using dm9000 device
host 10.0.0.1 is alive
MINI2440 #
```

```
MINI2440 # tftp 30000000 test.img
dm9000 i/o: 0x20000300, id: 0x90000a46
DM9000: running in 16 bit mode
MAC: da:15:db:01:01:01
could not establish link
Using dm9000 device
TFTP from server 10.0.0.1; our IP address is 10.0.0.110
Filename 'test.img'.
Load address: 0x30000000
Loading: #
          29.3 KiB/s
done
Bytes transferred = 61 (3d hex)
MINI2440 #
```

---

奇怪的是，tftp 服务器只能被下载一次，再次下载需要重启 tftp 服务  
`sudo service tftpd-hpa restart`

到此，验证了我们的移植时正确的

## 使用 kermit 下载程序到 SDRAM 中执行

### 1. 安装 kermit

```
sudo apt-get install ckermit
```

配置 kermit

```
~/.kermrc
```

```
set line /dev/ttyUSB0
```

```
set speed 115200
```

```
set carrier-watch off
```

```
set handshake none
```

```
set flow-control none
```

```
robust
```

```
set file type bin
```

```
set file name lit
```

```
set rec pack 1000
```

```
set send pack 1000
```

```
set window 5
```

### 2. 设置串口权限

```
jiaduo@jiaduo-syberos:~$ ll /dev/ttyUSB0
```

```
crw-rw---- 1 root dialout 188, 0 12ææ^ 17 15:41 /dev/ttyUSB0
```

将 jiaduo 添加到 dialout 组中，保证有读写权限

```
sudo usermod -a -G dialout jiaduo
```

注销用户重新登录。

3.使用 `kermit -c` 串口

4.loadb 30000000

5.Ctrl+\ 按下 `c` 进入 `kermit` 命令行发送文件

```
send app.bin
```

成功的话会提示 `SUCCESS`

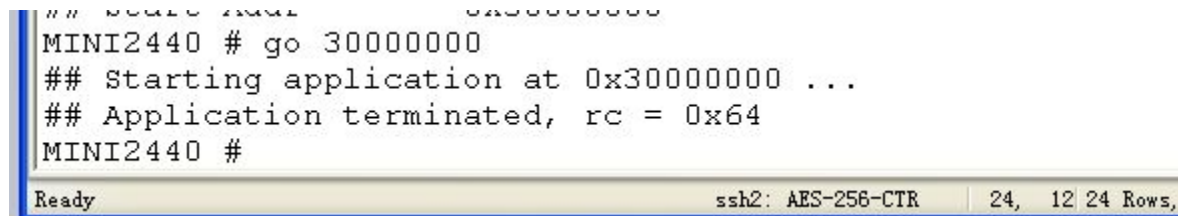
输入 `c` 重新连接如终端

6.go 30000000

运行我们下载的程序

程序现象为让 LED 闪 10 下然后熄灭。

```
MINI2440 # go 30000000
## Starting application at 0x30000000 ...
## Application terminated, rc = 0x64
MINI2440 #
```



`go` 命令是 `uboot` 中 `cmd_boot.c` 中的 `do_go` 函数来处理的。程序执行完退出，如果我们写个死循环函数，就一直不会退出，必须重启开发板。

程序如下：

`app_led.c`

```
#define GPBCON (*(volatile unsigned int *)0x56000010)
#define GPBDAT (*(volatile unsigned int *)0x56000014)
void led_init(void);
void led_on(void);
void led_off(void);
void delay(unsigned int);
void main(void) /* make sure: 'main'.test = Ttext */
{
    unsigned int i=0;
    led_init();
    for(i=0;i<10;i++)
    {
        led_on();
        delay(100);
        led_off();
        delay(100);
    }
}
void led_init(void)
{
```



```

        GPBCON &= ~(3<<10);
        GPBCON |= (1<<10);
    }
    void led_on(void)
    {
        GPBDAT &= ~(1<<5);
    }
    void led_off(void)
    {
        GPBDAT |= (1<<5);
    }
    void delay(unsigned int c)
    {
        unsigned int i=0;
        while(c--)
        {
            for(i=0;i<100000;i++);
        }
    }
}

```

## Makefile

```

CROSS_COMPILE=arm-linux-
app.bin:app.out
    $(CROSS_COMPILE)objcopy -O binary app.out app.bin
app.out:app_led.o
    $(CROSS_COMPILE)ld -e main -Ttext 0x30000000 app_led.o -o app.out
app_led.o:app_led.c
    $(CROSS_COMPILE)gcc -c app_led.c -o app_led.o

.PHONY:clean
clean:
    rm -rf *.o *.out

```

直接使用 **make** 命令生成需要的 **app.bin**。

这里注意的是没有连接文件 **.lds**，所以为了保证 **main** 函数在 30000000 处，必须让 **main** 函数在程序的开头，因为我们最后使用的是 **bin** 文件不是 **elf** 文件所以 **-e** 选项可不用

## 发布 uboot

对 **uboot** 源码修改后，可以通过创建补丁文件来记录修改，并还原和应用修改

1.通过 **diff** 命令创建补丁文件

```
diff -uprN u-boot-2015.10 u-boot-2015.10-mini2440 > u-boot-2015.10-mini2440.patch
```

u-boot-2015.10-mini2440 使我们修改后的代码  
u-boot-2015.10 是原始代码  
u-boot-2015.10-mini2440.patch 使我们需要的补丁文件

对 u-boot-2015.10 打补丁

patch -p0 < u-boot-2015.10-mini2440.patch

```
patching file u-boot-2015.10/arch/arm/cpu/arm920t/start.S
patching file u-boot-2015.10/arch/arm/Kconfig
patching file u-boot-2015.10/arch/arm/lib/relocate.S
patching file u-boot-2015.10/board/samsung/mini2440/Kconfig
patching file u-boot-2015.10/board/samsung/mini2440/lowlevel_init.S
patching file u-boot-2015.10/board/samsung/mini2440/MAINTAINERS
patching file u-boot-2015.10/board/samsung/mini2440/Makefile
patching file u-boot-2015.10/board/samsung/mini2440/mini2440.c
patching file u-boot-2015.10/configs/mini2440_defconfig
patching file u-boot-2015.10/drivers/mtd/cfi_flash.c
patching file u-boot-2015.10/drivers/mtd/mtdcore.c
patching file u-boot-2015.10/drivers/mtd/nand/s3c2410_nand.c
patching file u-boot-2015.10/drivers/net/dm9000x.c
patching file u-boot-2015.10/include/configs/mini2440.h
patching file u-boot-2015.10/include/configs/smdk2410.h
```

patch 后的 u-boot-2015.10 就和我们修改过的一样了  
编译

cd u-boot-2015.10

make mini2440\_defconfig

make CROSS\_COMPILE=arm-linux-

得到 u-boot.bin 就可以下载了

## 2.打包

tar -jcf u-boot-2015.10-mini2440.tar.bz2 u-boot-2015.10-mini2440/

