U-BOOT 启动过程分析

.U-Boot启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 6.3 所示。

看一下 board/smsk2410/u-boot.lds 这个链接脚本，可以知道目标程序的各部分链接顺序。第一个要链接的是 cpu/arm920t/start.o，那么 U-Boot 的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

1. cpu/arm920t/start.S

这个汇编程序是 U-Boot 的入口程序，开头就是复位向量的代码。

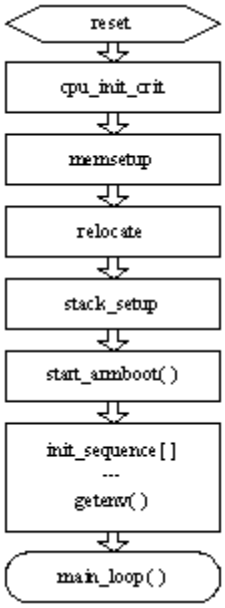


图 6.3　U-Boot 启动代码流程图

```
_start: b       reset       //复位向量

        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

…

 /* the actual reset code   */

reset:          //复位启动子程序
```

```
    /* 设置 CPU 为 SVC32 模式 */

    mrs    r0,cpsr

    bic    r0,r0,#0x1f

    orr    r0,r0,#0xd3

    msr    cpsr,r0

/* 关闭看门狗 */



/* 这些初始化代码在系统重起的时候执行，运行时热复位从 RAM 中启动不执行 */

#ifdef CONFIG_INIT_CRITICAL

    bl     cpu_init_crit

#endif



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

    adr    r0, _start          /* r0 是代码的当前位置 */

    ldr    r1, _TEXT_BASE      /* 测试判断是从 Flash 启动，还是 RAM */

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

    beq    stack_setup     /* 如果 r0 等于 r1，跳过重定位代码 */

    /* 准备重新定位代码 */

    ldr    r2, _armboot_start
```

```
        ldr    r3, _bss_start

        sub    r2, r3, r2           /* r2 得到 armboot 的大小    */

        add    r2, r0, r2           /* r2 得到要复制代码的末尾地址 */

copy_loop: /* 重新定位代码 */

        ldmia r0!, {r3-r10}   /*从源地址[r0]复制 */

        stmia r1!, {r3-r10}   /* 复制到目的地址[r1] */

        cmp    r0, r2           /* 复制数据块直到源数据末尾地址[r2] */

        ble    copy_loop


        /* 初始化堆栈等    */

stack_setup:

        ldr    r0, _TEXT_BASE               /* 上面是 128 KiB 重定位的 u-boot */

        sub    r0, r0, #CFG_MALLOC_LEN      /* 向下是内存分配空间 */

        sub    r0, r0, #CFG_GBL_DATA_SIZE /* 然后是 bdinfo 结构体地址空间   */

#ifdef CONFIG_USE_IRQ

        sub    r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)

#endif

        sub    sp, r0, #12      /* 为 abort-stack 预留 3 个字 */
```

```
clear_bss:
        ldr    r0, _bss_start       /* 找到 bss 段起始地址 */
        ldr    r1, _bss_end         /*  bss 段末尾地址   */
        mov    r2, #0x00000000      /* 清零 */
clbss_l:str r2, [r0]          /* bss 段地址空间清零循环...  */
        add    r0, r0, #4
        cmp    r0, r1
        bne    clbss_l
        /* 跳转到 start_armboot 函数入口，_start_armboot 字保存函数入口指针 */
        ldr    pc, _start_armboot
_start_armboot: .word start_armboot      //start_armboot 函数在 lib_arm/board.c 中实现
/* 关键的初始化子程序 */
cpu_init_crit:
……  //初始化 CACHE，关闭 MMU 等操作指令
        /* 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     memsetup        //memsetup 子程序在 board/smdk2410/memsetup.S 中实现

mov    lr, ip

mov    pc, lr
```

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。

```
void start_armboot (void)

{

    DECLARE_GLOBAL_DATA_PTR;

    ulong size;

    init_fnc_t **init_fnc_ptr;

    char *s;

    /* Pointer is writable since we allocated a register for it */

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

    /* compiler optimization barrier needed for GCC >= 3.4 */

    __asm__ __volatile__("": : :"memory");

    memset ((void*)gd, 0, sizeof (gd_t));

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

    memset (gd->bd, 0, sizeof (bd_t));

    monitor_flash_len = _bss_start - _armboot_start;
```

```
/* 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 ();

display_flash_config (size);

/* _armboot_start 在 u-boot.lds 链接脚本中定义 */

mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);

/* 配置环境变量，重新定位 */

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 ();       /* 获取列表中的设备 */

jumptable_init ();

console_init_r ();     /* 完整地初始化控制台设备 */
```

```
        enable_interrupts (); /* 使能例外处理 */

        /* 通过环境变量初始化 */

        if ((s = getenv ("loadaddr")) != NULL) {

                load_addr = simple_strtoul (s, NULL, 16);

        }

        /* main_loop()总是试图自动启动，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 */

        }

        /* NOTREACHED - no way out of command loop except booting */

}
```

3. 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/cmd_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,

};
```

U-BOOT start_armboot 浅析

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。

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。

从 U-Boot 源码看 C 语言对汇编代码中的符号引用

以下内容来自笔者在中国 Linux 论坛 Linux 嵌入技术讨论区的张贴：

---

aaronwong: u-boot 中代码的疑问(_armboot_start 与_start)?

我使用的是 u-boot-1.3.0-rc2。在 cpu/pxa/start.S 中，有如下的标号定义：

```
_TEXT_BASE:
.word TEXT_BASE /*uboot 映像在 SDRAM 中的重定位地址，我设置为 0xa170 0000 */

.globl _armboot_start
_armboot_start:
.word _start /*_start 是程序入口，链接完毕它的值应该是 0xa170 0000=TEXT_BASE*/
/* 这句话的意思应该是在_armboot_start 标号处，保存了_start 的值，也就是说，
_armboot_start 是存放_start 的地址，该地址对应的存储单元内容是 0xa170 0000*/
/*
* These are defined in the board-specific linker script. 下面的定义与上面应该是一个意思。
*/
.globl _bss_start
_bss_start:
.word __bss_start
```

==========

按照上面的理解，__bss_start 是 uboot 的 bss 段起始地址，那么 uboot 映像的大小就是

__bss_start - _start；在 relocate 代码段中计算 uboot 的大小时，也体现了这一点。
实际上，_armboot_start 并没有实际意义，它只是在"ldr r2, _armboot_start"中用來寻址_start 的值而已，_bss_start 也是一样的道理，真正有意义的应该是_start 和 __bss_start 本身。
但是，令我不解的是，在 C 入口函数 start_armboot()中(对应文件为 lib_arm/board.c)，有如下代码：

```
void start_armboot (void)
{
.....
gd = (gd_t*)(_armboot_start - CFG_MALLOC_LEN - sizeof(gd_t)); //第一句话
.....
monitor_flash_len = _bss_start - _armboot_start; //第二句话
.....
mem_malloc_init (_armboot_start - CFG_MALLOC_LEN); //第三句话
.....
}
```

==========
按照上面的理解，_armboot_start 与_bss_start 都是没有实际意义的，它们只是一个地址，有实际意义的是地址中的内容_start 和 __bss_start（虽然也还是地址）。象第一句话，其"意图"很明显，是把 gd 作为全局数据结构体的指针，并初始化为"SDRAM 中的 uboot 起始地址(即 TEXT_BASE)－CFG_MALLOC_LEN－全局数据结构体大小"。
要实现这个"意图"，应该是写成:gd = (gd_t*)(_start - CFG_MALLOC_LEN - sizeof(gd_t));
或者 gd = (gd_t*)(TEXT_BASE- CFG_MALLOC_LEN - sizeof(gd_t));才对阿？用 _armboot_start 来作运算应该是没有任何意义才对！？
第二句话也是一样的道理，它的意图是要计算 u-boot 映像的大小，应该写成__bss_start - _start 才对阿？
我使用 readelf 工具查看编译所得到的 uboot 映像文件得到信息如下：

```
[aaronwong@localhost build]$ readelf -s u-boot|grep _start
1018: a1700048 0 NOTYPE GLOBAL DEFAULT 1 _bss_start
1083: a1700044 0 NOTYPE GLOBAL DEFAULT 1 _armboot_start
1142: a1700000 0 NOTYPE GLOBAL DEFAULT 1 _start
1197: a171b070 0 NOTYPE GLOBAL DEFAULT ABS __bss_start
```

上面我删除了与该讨论无关的包含"_start""t 的标号信息。
显然，我前面的理解应该是正确的(_start=TEXT_BASE=0xa170 0000)。那么 u-boot 源代码中的 monitor_flash_len=_bss_start - _armboot_start=0xa1700048 - 0xa1700044 = 4，有什么意义？？
迷茫中，期盼大虾指点迷津，谢谢～！！！

---

eltshan: [Re: aaronwong]

1018: a1700048 0 NOTYPE GLOBAL DEFAULT 1 _bss_start
1083: a1700044 0 NOTYPE GLOBAL DEFAULT 1 _armboot_start
1142: a1700000 0 NOTYPE GLOBAL DEFAULT 1 _start
1197: a171b070 0 NOTYPE GLOBAL DEFAULT ABS __bss_start

我想:
_start 所在的地址是 a1700000,
_armboot_start 所在的地址是 a1700044,
那么 根据这句:
_armboot_start: .word _start
所以_armboot_start 的值应该是 a1700000

所以
monitor_flash_len = _bss_start - _armboot_start = a171b070 - a1700000 = 1b070
而不是你说的 = 4

以上个人意见.

---

aaronwong: [Re: eltshan]

谢谢，eltshan！你的理解是正确的，不过我看了之后还是没能想得很明白，因为我在想，按你所说，那么_start 的值应该是多少呢？难道是"b reset"这条指令的机器码？所以我对 ELF 格式的 u-boot 映像文件作了反汇编，分析之后终于找到了症结所在。以下是部分分析过程，首先是反汇编：
arm-iwmmxt-linux-gnueabi-objectdump -D u-boot > u-boot.s
并提取了 monitor_flash_len = _bss_start - _armboot_start;这条语句相关的反汇编代码如下：
==========
a1700044 <_armboot_start>:
a1700044: a1700000 .word 0xa1700000

a1700048 <_bss_start>:
a1700048: a171b070 .word 0xa171b070

a171b070 <monitor_flash_len>:
a171b070: 00000000 .word 0x00000000

.....
a1700f40: e59f41d0 ldr r4, [pc, #464] ; a1701118 <start_armboot+0x1dc>
//r4=[a1701118]=a1700044
.....
a1700f7c: e59f3198 ldr r3, [pc, #408] ; a170111c <start_armboot+0x1e0>
//r3=[a1700044]=a1700048
a1700f80: e5942000 ldr r2, [r4]
//r2=[a1700044]=a1700000
a1700f84: e59f4194 ldr r4, [pc, #404] ; a1701120 <start_armboot+0x1e4>
//r4=[a1701120]=a1719d24
a1700f88: e5933000 ldr r3, [r3]
//r3=[a1700048]=a171b070
a1700f8c: e0623003 rsb r3, r2, r3
//r3= r3-r2 = a171b070-a1700000 = 1b070;

a1700f90: e59f218c ldr r2, [pc, #396] ; a1701124 <start_armboot+0x1e8>
//r2=[a1701124]=a171b070
a1700f94: e5823000 str r3, [r2]
//monitor_flash_len=[r2]=r3=1b070
.....

a1701118: a1700044 .word 0xa1700044
a170111c: a1700048 .word 0xa1700048
a1701120: a1719d24 .word 0xa1719d24
a1701124: a171b070 .word 0xa171b070
==========

上面//是我自己的注释。这表明，你的理解的确是正确的。

经过这个过程之后，我终于认识到自己的误解在哪里了。原来，我是把"汇编语言中 LDR 伪指令对符号的引用"与"C 语言中对汇编程序中符号/常量/变量的引用"搞混淆了。我想说明以下几点：

(1) readelf 以及 u-boot.map 和 System.map 所给出的符号表中符号的值，实际上是表示符号所在的地址，而不是指符号本身的值。

(2) 汇编语言中没有指针的概念，因此对符号的引用是"赤裸裸"的。例如：

==========
.globl _armboot_start
_armboot_start: .word _start
ldr r2, _armboot_start
==========

实际上反汇编以后是：

==========
a1700044 <_armboot_start>:
a1700044: a1700000 .word 0xa1700000
a1700074: e51f2038 ldr r2, [pc, #-56] ; a1700044 <_armboot_start>
==========

也就是说，_armboot_start 是一个地址 0xa1700044，其中的内容是 0xa1700000，上面对 _armboot_start 的引用是直接将其替换为其表示的地址 0xa1700044，而非其中的内容 0xa1700000。这就是"赤裸裸"的引用。

(3) C 语言则不同，对变量/符号/常量的引用必须要通过地址来寻址，不管是全局变量还是局部变量，不同的是局部变量在生命期结束后，所占的地址空间会被释放而已。即使是函数调用时的参数传递，虽然是将实参的值"拷贝"给形参，但"拷贝"的过程也是通过实参和形参的地址来对两者进行访问的。

所以，在 C 语言中的 "monitor_flash_len = _bss_start - _armboot_start" 这句话中对 _armboot_star 的引用，实际上是把它用作了指针，把它作为访问对象的地址来使用，通过这个地址即 a1700044 来访问对应存储空间所存放的内容亦即 0xa1700000，_bss_start 也是同样的道理。所以这句话实际上是 monitor_flash_len =[a1700048]-[0xa1700044]=a171b070-a1700000 = 1b070，这样就得到了正确的结果。

现在，我们再**回答最前面的问题：_start 的值是什么？**_start 表示地址 0xa1700000 ，在汇编语言中，对_start 的"绝对引用"(这里是与用相对寻址进行跳转进行区别)就是将其替换为 0xa1700000，但其中存放的内容的的确确就是"b reset"这条指令的机器码，所以如果在 C 语言中引用_start，得到的结果反而就是这个指令的机器码了。其实这个问题很简单，只是和 C 语言的引用搅在一起，一些概念被偷换了而已。

以上是我的个人理解，有什么不对，还请指正。

再次感谢 eltshan 的提点。

U-Boot 1.3.1 源码阅读之 NAND FLASH 接口

U－Boot 1.3.1 是 U－Boot 目前的最新版本，和以前的版本相比，增加了对大页面 NAND FLASH 的支持。早期版本基本上只能支持到 512 字节的页面大小，而现在市面上大多是 2K 字节页面大小的 NAND FLASH，这样的 FLASH 容量大，价格低，性价比很高，很受电子设计工程师的青睐。

看源代码 board 目录下，有很多板子的配置。可以使用源代码阅读工具，搜索一下，可以看到其中有很多板子的目录下都有一个 nand.c 文件，文件中有一些 nand flash 必须的借口函数。下面逐一介绍。

board_nand_init 函数

在这个函数可以定义 NAND FLASH 命令/数据的地址，典型的可以看 board/sc3/sc3nand.c 文件：

```
        sc3_io_base = (void *) CFG_NAND_BASE;
        /* Set address of NAND IO lines (Using Linear Data Access
Region) */
        nand->IO_ADDR_R = (void __iomem *) sc3_io_base;
        nand->IO_ADDR_W = (void __iomem *) sc3_io_base;
```

还有一些接口也是必须的，如设置用户自定义的一些接口函数，比较典型的可以看 board/prodrive/pdnp3/nand.c 文件：

```
int board_nand_init(struct nand_chip *nand)
{
        pdnb3_ndfc = (struct pdnb3_ndfc_regs *)CFG_NAND_BASE;
```

```
        nand->eccmode = NAND_ECC_SOFT;

        /* Set address of NAND IO lines (Using Linear Data Access
Region) */
        nand->IO_ADDR_R = (void __iomem *) ((ulong) pdnb3_ndfc +
0x4);
        nand->IO_ADDR_W = (void __iomem *) ((ulong) pdnb3_ndfc +
0x4);
        /* Reference hardware control function */
        nand->hwcontrol = pdnb3_nand_hwcontrol;
        /* Set command delay time */
        nand->hwcontrol = pdnb3_nand_hwcontrol;
        nand->write_byte = pdnb3_nand_write_byte;
        nand->read_byte = pdnb3_nand_read_byte;
        nand->write_buf = pdnb3_nand_write_buf;
        nand->read_buf = pdnb3_nand_read_buf;
        nand->verify_buf = pdnb3_nand_verify_buf;
        nand->dev_ready = pdnb3_nand_dev_ready;
        return 0;
}
```

如果在这里定义了这些接口函数，则必须实现，用于替换 u-boot 的默认函数实现。

nand_chip 数据结构

其实，这里面最重要的是 nand_chip 数据结构，请看文件 include/linux/mtd/nand.h：

```
/**
  * struct nand_chip - NAND Private Flash Chip Data
  * @IO_ADDR_R:                                [BOARDSPECIFIC] address
```

```
to read the 8 I/O lines of the flash device
  * @IO_ADDR_W:                          [BOARDSPECIFIC] address to
write the 8 I/O lines of the flash device
  * @read_byte:                              [REPLACEABLE] read one
byte from the chip
  * @write_byte:                [REPLACEABLE] write one byte to the
chip
  * @read_word:                [REPLACEABLE] read one word from the
chip
  * @write_word:                [REPLACEABLE] write one word to the
chip
  * @write_buf:                      [REPLACEABLE] write data from the
buffer to the chip
  * @read_buf:                       [REPLACEABLE] read data from the
chip into the buffer
  * @verify_buf:                      [REPLACEABLE] verify buffer
contents against the chip data
  * @select_chip:              [REPLACEABLE] select chip nr
  * @block_bad:                      [REPLACEABLE] check, if the block
is bad
  * @block_markbad:        [REPLACEABLE] mark the block bad
  * @hwcontrol:                    [BOARDSPECIFIC] hardwarespecific
function for accesing control-lines
  * @dev_ready:                [BOARDSPECIFIC] hardwarespecific
function for accesing device ready/busy line
  *                       If set to NULL no access to ready/busy is
available and the ready/busy information
  *                       is read from the chip status register
  * @cmdfunc:                [REPLACEABLE] hardwarespecific function
```

```
for writing commands to the chip
  * @waitfunc:                [REPLACEABLE] hardwarespecific function
for wait on ready
  * @calculate_ecc:         [REPLACEABLE] function for ecc calculation
or readback from ecc hardware
  * @correct_data:          [REPLACEABLE] function for ecc correction,
matching to ecc generator (sw/hw)
  * @enable_hwecc:          [BOARDSPECIFIC] function to enable (reset)
hardware ecc generator. Must only
  *                         be provided if a hardware ECC is available
  * @erase_cmd:               [INTERN] erase command write function,
selectable due to AND support
  * @scan_bbt:               [REPLACEABLE] function to scan bad block
table
  * @eccmode:                [BOARDSPECIFIC] mode of ecc, see defines
  * @eccsize:                [INTERN] databytes used per
ecc-calculation
  * @eccbytes:               [INTERN] number of ecc bytes per
ecc-calculation step
  * @eccsteps:               [INTERN] number of ecc calculation steps
per page
  * @chip_delay:              [BOARDSPECIFIC] chip dependent delay
for transfering data from array to read regs (tR)
  * @chip_lock:              [INTERN] spinlock used to protect
access to this structure and the chip
  * @wq:                                    [INTERN] wait queue to
sleep on if a NAND operation is in progress
  * @state:                              [INTERN] the current state of
the NAND device
```

```
  * @page_shift:                [INTERN] number of address bits in a
page (column address bits)
  * @phys_erase_shift:         [INTERN] number of address bits in a
physical eraseblock
  * @bbt_erase_shift:          [INTERN] number of address bits in a bbt
entry
  * @chip_shift:                [INTERN] number of address bits in one
chip
  * @data_buf:                 [INTERN] internal buffer for one page +
oob
  * @oob_buf:                  [INTERN] oob buffer for one eraseblock
  * @oobdirty:                 [INTERN] indicates that oob_buf must be
reinitialized
  * @data_poi:                 [INTERN] pointer to a data buffer
  * @options:                  [BOARDSPECIFIC] various chip options.
They can partly be set to inform nand_scan about
  *                           special functionality. See the defines for
further explanation
  * @badblockpos:              [INTERN] position of the bad block marker in
the oob area
  * @numchips:                 [INTERN] number of physical chips
  * @chipsize:                 [INTERN] the size of one chip for
multichip arrays
  * @pagemask:                 [INTERN] page number mask = number of
(pages / chip) - 1
  * @pagebuf:                  [INTERN] holds the pagenumber which is
currently in data_buf
  * @autooob:                  [REPLACEABLE] the default
(auto)placement scheme
```

```
  * @bbt:                 [INTERN] bad block table pointer
  * @bbt_td:                [REPLACEABLE] bad block table descriptor
for flash lookup
  * @bbt_md:                [REPLACEABLE] bad block table mirror
descriptor
  * @badblock_pattern:        [REPLACEABLE] bad block scan pattern
used for initial bad block scan
  * @controller:                [OPTIONAL] a pointer to a hardware
controller structure which is shared among multiple independend devices
  * @priv:                [OPTIONAL] pointer to private chip date
  */

struct nand_chip {
        void __iomem        *IO_ADDR_R;
        void __iomem        *IO_ADDR_W;

        u_char                  (*read_byte)(struct mtd_info *mtd);
        void                        (*write_byte)(struct mtd_info
*mtd, u_char byte);
        u16                        (*read_word)(struct mtd_info *mtd);
        void                        (*write_word)(struct mtd_info
*mtd, u16 word);

        void                (*write_buf)(struct mtd_info *mtd, const
u_char *buf, int len);
        void                (*read_buf)(struct mtd_info *mtd, u_char
*buf, int len);
        int                (*verify_buf)(struct mtd_info *mtd, const
u_char *buf, int len);
```

```
        void                (*select_chip)(struct mtd_info *mtd, int
chip);
        int                 (*block_bad)(struct mtd_info *mtd, loff_t
ofs, int getchip);
        int                 (*block_markbad)(struct mtd_info *mtd,
loff_t ofs);
        void                (*hwcontrol)(struct mtd_info *mtd, int
cmd);
        int                 (*dev_ready)(struct mtd_info *mtd);
        void                (*cmdfunc)(struct mtd_info *mtd, unsigned
command, int column, int page_addr);
        int                 (*waitfunc)(struct mtd_info *mtd, struct
nand_chip *this, int state);
        int                 (*calculate_ecc)(struct mtd_info *mtd,
const u_char *dat, u_char *ecc_code);
        int                 (*correct_data)(struct mtd_info *mtd,
u_char *dat, u_char *read_ecc, u_char *calc_ecc);
        void                (*enable_hwecc)(struct mtd_info *mtd, int
mode);
        void                (*erase_cmd)(struct mtd_info *mtd, int
page);
        int                 (*scan_bbt)(struct mtd_info *mtd);
        int                 eccmode;
        int                 eccsize;
        int                 eccbytes;
        int                 eccsteps;
        int                 chip_delay;
#if 0
        spinlock_t          chip_lock;
```

```
        wait_queue_head_t wq;
        nand_state_t        state;
#endif
        int
 page_shift;
        int
 phys_erase_shift;
        int
 bbt_erase_shift;
        int
chip_shift;
        u_char                                          *da
ta_buf;
        u_char                                          *oo
b_buf;
        int
        oobdirty;
        u_char                                          *da
ta_poi;
        unsigned int                            options;
        int
               badblockpos;
        int
               numchips;
        unsigned long                                   chipsize;
        int
               pagemask;
        int
               pagebuf;
```

```
        struct nand_oobinfo             *autooob;
        uint8_t
     *bbt;
        struct nand_bbt_descr        *bbt_td;
        struct nand_bbt_descr        *bbt_md;
        struct nand_bbt_descr        *badblock_pattern;
        struct nand_hw_control        *controller;
        void
                *priv;
};
```

这个数据结构的注视非常明了，指出了那些函数可以被替换，可被替换的函数，用户可以根据自己的需求实现，一般在上面所讲的 nand.c 中实现。当然，在其他地方实现也可以，U-Boot 源代码中也有这样的例子。

nand_base.c 文件

nand_chip 数据结构定义的函数，U-Boot 的默认实现在 drivers/mtd/nand/nand_base.c 文件中。可以仔细看看源码。如果要进行替换，这些函数有一定的参考价值。

如果要为 nand_chip 数据结构增加某些接口函数或者某些成员，可以修改这两个文件。

nand Flash 和 nand_legacy

nand 目录下的文件是能够支持 2K 大页面的 NAND FLASH 的实现，而 nand_legacy 目录下的文件只能支持到 512 字节的页面，在使用的时候需要加以区分。

U-Boot Practically Porting Guide

Author: Aaron Wong aaronwong@engineer.com

U-Boot 的移植之(一)基础篇：添加新的目标板定义

本文使用最新的 U-Boot-1.3.0-rc2。

U-Boot 本身支持很多开发板，在其源代码中，每个板子都对应一个 board/目录下的文件夹(笔者注：这并不确切，因为有的文件夹是供应商名称，下面可以有多个目标板目录，这里只考虑最简单的情况)，以及 include/configs/目录下的目标板配置头文件。因此，要添加 U-Boot 对我们的目标板的支持，首先就是要建立目标板文件夹和配置头文件，并修改相关的 Makefile。

下面以实例说明为 U-Boot 添加新的目标板定义的步骤和过程。

(1)在 board/目录下建立目标板目录。

笔者的目标板是 XSBASE270，处理器是 PXA270。由于 U-Boot 中本身支持很多开发板和处理器，可以从中找出与自己处理器型号相同或相近的开发板，在此基础上再做后续修改。

adsvix 使用的也是 PXA27x 处理器，因此可以把它作为模板。

```
cd board/

cp -arv adsvix xsbase270

mv xsbase270/adsvix.c xsbase270/xsbase270.c
```

(2)在 include/configs/目录下建立目标板配置头文件。

```
cd include/configs/

cp adsvix.h xsbase270.h
```

(3)修改 Makefile。

一是要在总的 Makefile(U-Boot 源码顶层目录下)中加入目标板的编译配置选项，这也可以参考 adsvix 的进行修改，只要把目标板名称改换为 xsbase270 即可：

```
adsvix_config: unconfig

@$(MKCONFIG) $(@:_config=) arm pxa adsvix

xsbase270_config: unconfig

@$(MKCONFIG) $(@:_config=) arm pxa
xsbase270
```

这里 xsbase270 与 board/目录下目标板文件夹名称 xsbase270 一致。

另外，还需要注意，该 Makefile 中定义了 CROSS_COMPILE 的值，以在交叉编译时指定交叉编译器。缺省情况下对 ARM 的 CROSS_COMPILE 定义如下：

```
ifeq ($(ARCH),arm)

CROSS_COMPILE = arm-linux-

endif
```

即定义交叉编译器名的前缀为 arm-linux-，如果您使用的 toolchain 的名字不同，则需要作相应修改。例如笔者使用的是 arm-iwmmxt-linux-gnueabi-gcc，因此要将上面改为：CROSS_COMPILE = arm-iwmmxt-linux-gnueabi- 。

二是要修改 board/xsbase270 下的 Makefile。

```
#COBJS := adsvix.o pcmcia.o

COBJS := xsbase270.o pcmcia.o
```

这是因为前面将该目录下的源文件 adsvix.c 改为了 xsbase270.c。

至此，将新的目标板 xsbase270 的定义添加到 U-Boot 中的工作就算完成了。下面的命令可以编译得到 xsbase270 的 U-Boot：

```
# assuming you are at the top directory of u-boot

# define a build directory to keep object files during make process and also finally u-boot image

export BUILD_DIR=~/u-boot_xsbase270/build/

make xsbase270_config

# if you edit your source file and want to make again, just type “make distclean”and then call the above commands again.

make
```

当然，要使编译出来的这个 u-boot 能真正适用于我们的目标板，还有很多工作要做，包括处理器工作状态、存储器映射设置、网卡驱动的移植等等。所以，本篇的标题只是在 U-Boot 中添加对新目标板的“定义”，而非对新目标板的“支持”，这些工作需要对 U-Boot 的源代码有整体的认识，并结合自己的目标板的特性来完成。后续的篇章将继续介绍后面的内容。

作为本篇的补充内容，您也许仍有必要了解以下要点：

(1) 在 MAKEALL 文件中可以将新的目标板 xsbase270 添加到下面的 list 中：

```
#########################################################

## Xscale Systems

#########################################################

LIST_pxa=" \

adsvix \

cerf250 \

cradle \

csb226 \

delta \

innokom \

lubbock \

pleb2 \

pxa255_idp \

wepep250 \

xaeniax \

xm250 \

xsengine \

zylonite \

"
```

这并不是必须的，因为 MAKEALL 文件只用于为其中的所有目标板都编译一个 u-boot 时使用。

(2) 如何在 U-Boot 已有的目标板中找到与自己的目标板相近的目标板？

首要的是要找到与自己的目标板所用的处理器相同或统一系列的的目标板。在顶层目录下的 Makefile 中有各个板子的 config 列表，例如 XScale 系列的板子列表如下：

```
#########################################################

## XScale Systems

#########################################################

adsvix_config : unconfig

@$(MKCONFIG) $(@:_config=) arm pxa adsvix

xsbase270_config: unconfig

@$(MKCONFIG) $(@:_config=) arm pxa xsbase270

cerf250_config : unconfig

@$(MKCONFIG) $(@:_config=) arm pxa cerf250

cradle_config : unconfig

@$(MKCONFIG) $(@:_config=) arm pxa cradle

csb226_config : unconfig

@$(MKCONFIG) $(@:_config=) arm pxa csb226

delta_config :

@$(MKCONFIG) $(@:_config=) arm pxa delta

# .....  以下省略。
```

(3) 修改目标板的编译优化选项。

在 cpu/pxa/config.mk 文件中定义了目标板的编译优化选项 PLATFORM_RELFLAGS 和

PLATFORM_CPPFLAGS，您可以根据自己的需要进行修改。

笔者的交叉编译器 arm-iwmmxt-linux-gnueabi-gcc 默认有-march=iwmmxt，遵循新的 ARM EABI 标准，但仍要保留 PLATFORM_CPPFLAGS 中的"-mapcs-32,-mabi=apcs-gnu"选项，使用旧的 ABI 标准来编译，因为 u-boot 的汇编代码并非按照新的 ABI 规范编写。可使用-march=armv5te 来避免"warning: target CPU does not support interworking"警告。

如果编译过程中出现了关于 IDE 方面的错误，应修改 include/configs/xsbase270.h,注释掉″#define CONFIG_CMD_IDE”这一行，以禁止编译 IDE 的操作命令，因为在目标板启动阶段不需要对 IDE 接口进行其他操作。

U-Boot 的移植之(二)进阶篇：从源代码看系统启动过程

为什么要分析源代码？分析优秀的源代码本身就是一个学习的过程，也是进行深入研究的必经之路。不过在此我们的主要目的并非要研究 U-boot 或 Bootloader 技术本身，而仅仅是为了成功的并且恰当的将 U-Boot 移植到我们的开发板上。只有结合源代码了解了 U-boot 的系统引导过程，才能在移植和调试过程中保持清晰的思路，才能在碰到困难和问题时从根本上加以解决。

在动手分析之前，至少应该对 U-Boot 的源代码结构有基本的了解，很多参考书都有这方面的介绍，华清远见的《嵌入式 Linux 系统开发技术详解——基于 ARM》的讲解就比较清晰。

本文以 lubbock 开发板为例，以系统启动的流程为线索进行纵向分析：后续的移植工作也将以此开发板为模板。Lubbock 使用 PXA255 处理器。

首先要找到程序入口点。从 board/lubbock/u-boot.lds 可以发现，u-boot 的程序入口为_start，在 cpu/pxa/start.o 当中。因此首先要分析 start.S 程序，U-Boot 中所有的 PXA 系列的处理器都从这里开始执行第一条语句。

```
.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
```

0x0 地址开始是 ARM 异常向量表，学过 ARM 体系结构与编程的都明白，非常简单，不多废话。一上电的第一条指令是跳转到 reset 复位处理程序：

```
reset:
```

```
/* 进入 SVC 模式 */

#ifndef CONFIG_SKIP_LOWLEVEL_INIT

bl cpu_init_crit /* we do sys-critical inits */

#endif

#ifndef CONFIG_SKIP_RELOCATE_UBOOT

relocate:

......
```

一般不要定义 CONFIG_SKIP_LOWLEVEL_INIT，因此，接下来跳转到 cpu_init_crit 处开始执行：

```
cpu_init_crit:

/* 屏蔽所有中断 */

/* 设置时钟源，关闭除 FFUART,SRAM,SDRAM,FLASH 以外的外设时钟 */

......

#ifdef CFG_CPUSPEED

ldr r0, CC_BASE /* 时钟控制寄存器基址 */

ldr r1, cpuspeed

/* cpuspeed: .word CFG_CPUSPEED */

str r1, [r0, #CCCR]

mov r0, #2

mcr p14, 0, r0, c6, c0, 0



setspeed_done:

#endif /* CFG_CPUSPEED */

/* 跳转到 lowlevel_init，这里 ip 即 r12，用作暂存寄存器 */
```

```
mov ip, lr

bl lowlevel_init

mov lr, ip

/* Memory interfaces are working. Disable MMU and enable I-cache. */

ldr r0, =0x2001

......

/*  关闭 MMU，使能 I-Cache(可选) */

mov pc, lr /*  这里是从 cpu_init_crit 返回到 relocate 标号  */
```

可见，在 cpu_init_crit 中的主要工作是设置时钟，配置处理器主频(这时 CPU 的工作频率还没有改变)，调用 lowlevel_init 函数进行底层初始化(包括调整处理器工作频率、系统总线频率、存储器时钟频率以及存储系统的初始化等工作)，随后关闭 MMU 并使能 I-Cache，再返回。

lowlevel_init 函数在 board/lubbock/lowlevel_init.S 中定义，其流程都是按照 PXA27X 的开发手册来的，所以不再赘述。仅指出，其中的寄存器在 include/asm-arm/arch-pxa/pxa-regs.h 头文件中定义，寄存器初始化值在 include/configs/lubbock.h 中定义。另外，在后面的实际移植工作中，由于目标板 XSBASE270 使用的 PXA270 处理器，可使用 adsvix 开发板的 lowlevel_init.S 文件(lubbock 中没有开启 turbo 模式)。

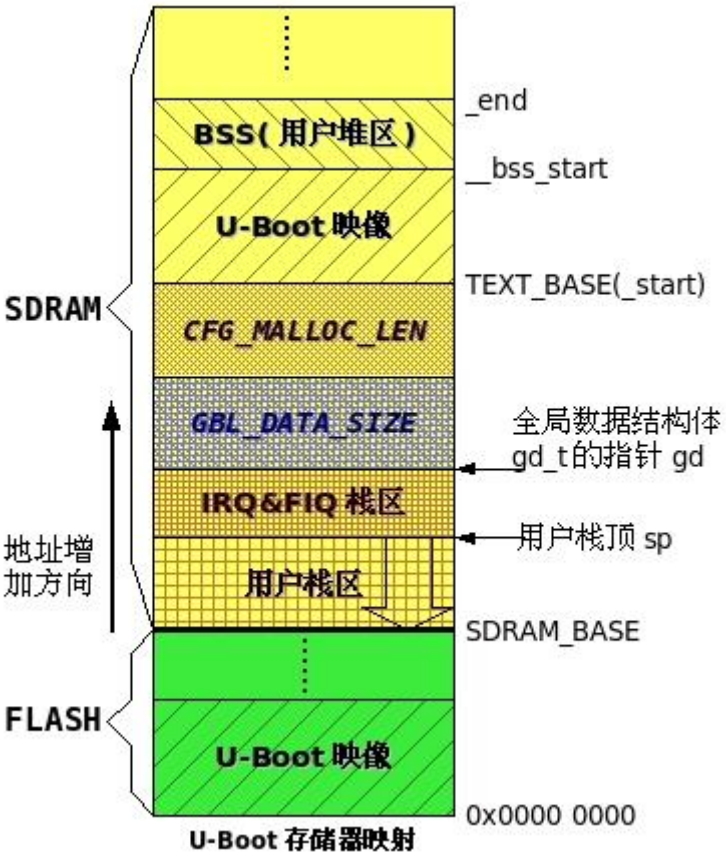
接着程序的执行线索进行分析。从 cpu_init_crit 返回后就开始 relocate(重定位)，即将 U-boot 从 FLASH 存储器搬运到 SDRAM 中 TEXT_BASE 开始的存储空间(TEXT_BASE 在 board/lubbock/config.mk 中定义)，并初始化堆栈(清零.bss 段)，以在 SDRAM 中开始进入到 Bootloader stage 2 的 C 程序入口。Relocate 部分开始的代码如下：

```
/*  之前已定义的部分变量有：

_TEXT_BASE: .word TEXT_BASE

_armboot_start: .word _start

_bss_start: .word __bss_start

_bss_end: .word _end */

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 /*  读入_start 到 r2 */

ldr r3, _bss_start /*  读入__bss_start 到 r3 */

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



/* Set up the stack */

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



ldr pc, _start_armboot



_start_armboot: .word start_armboot
```

这是很经典的一段代码，相信学习凡是过 ARM 编程的，都分析过这段代码，所以也不再赘述。之所以列出这段代码，一是为了找到 C 程序入口 start_armboot，二是为了给出 U-Boot 的一个存储器映射图：



这个图可以帮助我们更好地理解后续的 C 语言代码以及 U-Boot 对内存的分配与使用情况。

接下来进入到 Bootloader Stage 2 即 C 语言代码部分，入口是 start_armboot，对应的源文件是 lib_arm/board.c，这一文件对所有的 ARM 处理器都是通用的，因此在移植的时候不用修改。相关源代码如下：

```
DECLARE_GLOBAL_DATA_PTR;

/*  在 include/asm-arm/global_data.h 中定义的一个全局寄存器变量的声明：

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

*  用于存放全局数据结构体 gd_t 的地址。

*/

void start_armboot (void)

{
```

```
init_fnc_t **init_fnc_ptr;

char *s;

#ifndef CFG_NO_FLASH

ulong size;

#endif

#if defined(CONFIG_VFD) ||
defined(CONFIG_LCD)

/*  本次移植暂不配置 VFD 和 LCD，后面也将不考虑的部分略去  */

/*  初始化全局数据结构体指针 gd */

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

...../* memset 在 lib_generic/string.c 中定义*/

memset ((void*)gd, 0, sizeof (gd_t)); /*用 0 填充全局数据表*gd */

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

memset (gd->bd, 0, sizeof (bd_t)); /*用 0 填充(初始化) *gd->bd */



monitor_flash_len = _bss_start - _armboot_start;



for (init_fnc_ptr = init_sequence; *init_fnc_ptr;
++init_fnc_ptr) {

if ((*init_fnc_ptr)() != 0) {

hang (); /*  打印错误信息并死锁  */

}
```

```
}



#ifndef CFG_NO_FLASH

/* configure available FLASH banks */

size = flash_init (); /* drivers/cfi_flash.c 或自定义
*/

display_flash_config (size);

#endif /* CFG_NO_FLASH */



/*armboot_start is defined in the board-specific
linker script*/

mem_malloc_init (_armboot_start -
CFG_MALLOC_LEN);

......

/* initialize environment */

env_relocate ();



/* IP Address */

gd->bd->bi_ip_addr = getenv_IPaddr ("ipaddr");

/* MAC Address */

{

int i;

ulong reg;

char *s, *e;
```

```
char tmp[64];



i = getenv_r ("ethaddr", tmp, sizeof (tmp));

s = (i > 0) ? tmp : NULL;



for (reg = 0; reg < 6; ++reg) {

gd->bd->bi_enetaddr[reg] = s ? simple_strtoul (s,
&e, 16) : 0;

if (s)

s = (*e) ? e + 1 : e;

}

}



devices_init (); /* get the devices list going. */

.......

jumptable_init ();

console_init_r (); /* fully init console as a device */

enable_interrupts (); /* enable exceptions */



/* Perform network card initialisation if necessary */

#if defined(CONFIG_DRIVER_SMC91111)|
|defined (CONFIG_DRIVER_LAN91C96)

if (getenv ("ethaddr"))

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

```
#endif /* CONFIG_DRIVER_SMC91111 ||
CONFIG_DRIVER_LAN91C96 */



/* Initialize from environment */

if ((s = getenv ("loadaddr")) != NULL) {

load_addr = simple_strtoul (s, NULL, 16);

}



#if defined(CONFIG_CMD_NET)

if ((s = getenv ("bootfile")) != NULL)

copy_filename (BootFile, s, sizeof (BootFile));

#endif

#ifdef BOARD_LATE_INIT

board_late_init ();

#endif

......

/*main_loop() can return to retry autoboot, if so just
run it again.*/

for (;;) {

main_loop ();

}

}
```

gd_t 是全局数据表类型，在 include/asm-arm/global_data.h 中定义如下：

```
/*
```

```
Keep it *SMALL* and remember to set
CFG_GBL_DATA_SIZE > sizeof(gd_t)

*/

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;
```

其中，bd_t 在 include/asm-arm/u-boot.h 中定义如下：

```
typedef struct bd_info {

int bi_baudrate; /* serial console baudrate */

unsigned long bi_ip_addr; /* IP Address */

unsigned char bi_enetaddr[6]; /* Ethernet adress */

struct environment_s *bi_env;
```

```
ulong bi_arch_number; /* unique id for this board */

ulong bi_boot_params; /* where this board expects
params*/

struct /* RAM configuration */

{

ulong start;

ulong size;

}bi_dram[CONFIG_NR_DRAM_BANKS];

} bd_t;
```

jt 是函数数组指针，随后将在 jumptable_init()函数中初始化。

从 lib_arm/board.c 的源码不难分析出系统的启动流程：首先初始化全局数据表，然后顺序执行函数指针数组 init_sequence 中的一系列初始化函数——由其在本文件中的相关定义可得知初始化流程：

```
typedef int (init_fnc_t) (void);

init_fnc_t *init_sequence[] = {

cpu_init, /* basic cpu dependent setup --
cpu/pxa/cpu.c */

board_init, /* basic board setup
--board/lubbock/lubbock.c */

interrupt_init, /* set up exceptions --
cpu/pxa/interrupts.c */

env_init, /* initialize environment --
common/env_flash.c */

init_baudrate, /* initialze baudrate
settings--lib_arm/board.c */

serial_init, /* serial communications
setup--cpu/pxa/serial.c */

console_init_f, /* stage 1 init of console --
```

```
common/console.c */

display_banner, /* say that we are here --
lib_arm/board.c */

#if defined(CONFIG_DISPLAY_BOARDINFO)

checkboard, /* display board info */

#endif

dram_init, /* configure available RAM banks
--board/lubbock/lubbock.c */

display_dram_config, /* lib_arm/board.c */

NULL,

};
```

在执行这个函数序列的过程中，任何一个函数异常返回都会导致 u-boot“死锁”或说“挂起”在 hang()函数的死循环当中。

若一切顺利，接下来就调用 flash_init()函数初始化 CFI FLASH(针对 NOR 型闪存而言)，该函数在 drivers/cfi_flash.c 中定义，不过，只有在目标板头文件中”#define CFG_FLASH_CFI_DRIVER”之后该驱动才会被编译；在 lubbock 的 u-boot 实现当中，include/configs/lubbock.h 中没有定义 CFG_FLASH_CFI_DRIVER，而是在 board/lubbock/ flash.c 中实现了自己的 FLASH 驱动，包括 flash_init()在内。在移植 U-Boot 时，可以根据实际情况选择使用 U-Boot 自带的 FLASH 驱动还是自己编写新的驱动。如果配置了 NAND 闪存，还会对其进行初始化；笔者的 XSABSE270 板没有焊接 NAND FLASH，故对此不作讨论。

接下来调用 env_relocate()函数初始化环境变量，该函数在 common/env_common.c 文件中定义。在同一文件中可以发现还定义了一个字符数组 default_environment[]，用于描述缺省的环境变量，这些都要在 include/configs/lubbock.h 头文件中进行设置，包括启动命令 CONFIG_BOOTCOMMAND，波特率 CONFIG_BAUDRATE，IP 地址 CONFIG_IPADDR 等等。

然后是获取自设置的目标板的网络地址，包括 IP 地址和 MAC 地址。

再然后是调用 common/devices.c 中定义的 devices_init()函数来创建设备列表，并初始化相应的设备，主要是”stdin”,”stdout”,”stderr”以及自定义的设备如 I2C，LCD 等。这些相关代码是与平台无关的，因此从移植的角度考虑，不必作细致的研究与分析。

接着调用 common/exports.c 中定义的 jumptable_init()函数，初始化全局数据表中的跳转表 gd->jt，跳转表是一个函数指针数组，定义了 u-boot 中基本的常用的函数库；而 gd->jt 是这个函数指针数组的首指针。部分代码如下：

```
void jumptable_init (void) {

int i;

gd->jt = (void **) malloc (XF_MAX * sizeof (void
*));

for (i = 0; i < XF_MAX; i++)

gd->jt[i] = (void *) dummy;

gd->jt[XF_get_version] = (void *) get_version;

gd->jt[XF_malloc] = (void *) malloc;

gd->jt[XF_free] = (void *) free;

gd->jt[XF_getenv] = (void *) getenv;

gd->jt[XF_setenv] = (void *) setenv;

......

}
```

上面的 XF_get_version， XF_malloc， XF_free 等在 include/exports.h 的枚举变量中定义，因此，实际上是作为"Label 式整型序号"使用，即 XF_get_version=1, XF_malloc=2, XF_free=3 ...，相关代码如下：

```
enum { /* include/exports.h */

#define EXPORT_FUNC(x) XF_ ## x ,

#include <_exports.h>

#undef EXPORT_FUNC

XF_MAX

};
```

```
EXPORT_FUNC(get_version)

EXPORT_FUNC(getc)

EXPORT_FUNC(tstc)
```

```
EXPORT_FUNC(putc)

EXPORT_FUNC(puts)

EXPORT_FUNC(printf)

..... /* include/_exports.h */
```

由于这些也是平台无关的代码，因此在移植过程中也不必深究。

然后是调用 common/console.c 中定义的函数 console_init_r()初始化串口控制台，这同样是平台无关的代码，所以不必关心。

这时 U-Boot 的基本功能已经初始化完毕，便可开中断，并进行附加功能的配置与初始化，包括网卡驱动配置，目标板使用 LAN91C1111 网卡，对应 SMC91111 网卡驱动，可以根据需要配置其他的网卡驱动如 CS8900 等，这些都在 include/configs/lubbock.h 中定义。

然后是调用 board/lubbock/lubbock.c 中定义的 board_late_init()函数进行板级的后期初始化，实际上是配置 stdout 和 stderr 的硬件设备。相关源代码如下：

```
int board_late_init(void)

{

setenv("stdout", "serial");

setenv("stderr", "serial");

return 0;

}
```

最后需要注意的一个很重要的文件是 lib_arm/armlinux.c，它实现的功能包括设置内核启动参数，并负责将这些参数传递给内核，最后跳转到 Linux 内核入口函数，将控制权交给内核。

具体传递哪些参数，是通过在 include/configs/lubbock.c 中指定条件编译选项来控制的，对应于 lib_arm/armlinux.c 中的部分源代码形式如下：

```
#if defined (CONFIG_SETUP_MEMORY_TAGS)
|| \

defined (CONFIG_CMDLINE_TAG) || \

defined (CONFIG_INITRD_TAG) || \

defined (CONFIG_SERIAL_TAG) || \
```

```
defined (CONFIG_REVISION_TAG)

static void setup_start_tag (bd_t *bd);



# ifdef CONFIG_SETUP_MEMORY_TAGS

static void setup_memory_tags (bd_t *bd);

# endif

static void setup_commandline_tag (bd_t *bd, char
*commandline);



# ifdef CONFIG_INITRD_TAG

static void setup_initrd_tag (bd_t *bd, ulong
initrd_start,

ulong initrd_end);

# endif

static void setup_end_tag (bd_t *bd);

static struct tag *params;

#endif

......

void do_bootm_linux (cmd_tbl_t *cmdtp, int flag,
int argc, char *argv[],

ulong addr, ulong *len_ptr, int verify)

{

......

void (*theKernel)(int zero, int arch, uint params);
```

```
.....

#ifdef CONFIG_CMDLINE_TAG

char *commandline = getenv ("bootargs");

#endif

theKernel = (void (*)(int, int,
uint))ntohl(hdr->ih_ep);

.....

theKernel (0, bd->bi_arch_number,
bd->bi_boot_params);

}
```

关于这个参数列表中各个参数的定义及含义，以及参数列表的初始化过程，可以参考 Booting ARM Linux 一文。内核是如何找到这个参数列表在内存中的位置，以接收这些参数的呢？实际上，参数列表(tag list)在内存中的起始地址会保存在通用寄存器 R2 中，并传递给内核。而按照习惯或说惯例，通常 tag list 的首地址(物理地址)会设置为 RAM 起始地址+ 0x100 偏移量，因此 R2 的值实际上是确定不变的。另外，还要正确设置 R0 和 R1 的值，在呼叫内核时，R0 的值应为 0，R1 中则应保存机器类型(machine type)编号。R0,R1 和 R2 都会作为参数传递给内核。

在上面的代码中，定义了一个函数指针 theKernel，通过倒数第二条语句将内核入口地址赋给 theKernel(hdr 是 include/image.h 中定义的一个 image_header 结构体类型的数据，hdr->ih_ep 中保存了内核入口地址，ntohl 的功能是字节顺序的大小端转换，相关代码可以参考 tools/mkimage.c)，最后，根据 APCS 规则，将 0, bd->bi_arch_number, bd->bi_boot_params 依次作为参数通过 R0，R1 和 R2 传递给 theKernel 函数，并进入内核启动部分。

至此，我们已经从源代码入手简要分析了 U-Boot 的启动流程，在这个过程中，我们对前一篇文章"添加新的目标板定义"也有了更进一步的理解和认识：为什么要添加这些文件；哪些文件是平台相关的并且必须要根据平台特性进行修改的；哪些文件是平台无关的，是不需要修改的，只需在头文件中作适当配置即可。

下一节，我们将给出移植 U-Boot 到 XSBASE270 开发板的实例。

U-Boot 的移植之(三)实战篇：移植 U-Boot 到 XSBASE270 开发板

1. 在 U-Boot 中添加 XSBASE270 目标板的定义

具体方法可参考第一节，本篇给出部分细节和要点，假定$U-BOOT 为源码根目录。

```
#################################################################

# (1)建立目标板目录

#  其中 lowlevel_init.S 采用 adsvix 的文件，以开启 turbo mode，并注释掉

#  其中对 pxavoltage.S 文件中 initPXAvolatage 函数的调用。

#################################################################

cd board/

cp -arv lubbock xsbase270

mv xsbase270/lubbock.c xsbase270/xsbase270.c

cp adsvix/lowlevel_init.S xsbase270/

vim xsbase270/lowlevel_init.S

@setvoltage:

@ mov r10, lr

@ bl initPXAvoltage

@ mov lr, r10

#################################################################

# (2)建立目标板配置头文件

#################################################################

cd $U-BOOT/include/configs

cp lubbock.h xsbase270.h

#################################################################

# (3)修改 Makefile

#################################################################
```

```
#########在$U-BOOT/Makefile 中添加：

xsbase270_config: unconfig

@$(MKCONFIG) $(@:_config=) arm pxa xsbase270

#########在$U-BOOT/Makefile 中修改 CROSS_COMPILE：

CROSS_COMPILE = arm-iwmmxt-linux-gnueabi-

#########在$U-BOOT/board/xsbase270/Makefile 中修改：

#COBJS := lubbock.o flash.o

COBJS := xsbase270.o
```

这里去掉了 flash.c 文件，因为它是在 lubbock 板中自定义的 FLASH 存储器驱动，lubbock 不使用 U-Boot 自带的 FLASH 驱动；而在本次移植中，我们将使用 U-Boot 自带的 drivers/cfi_flash.c 作为 XSBASE270 开发板的 NOR 型闪存 28F128K18C 的驱动程序，具体过程后述。

实际移植过程中还可能要作如下改动：

```
#####################################################################

# (1) cpu/pxa/config.mk

#####################################################################

#armv5-->armv5te, modified by aaron wong

PLATFORM_CPPFLAGS += -march=armv5te -mtune=xscale



#####################################################################

# (2) include/asm-arm/mach-types.h

#####################################################################

/* added by aaron */

#define MACH_TYPE_XSBASE270 1141
```

编译 U-Boot:

```
export BUILD_DIR=~/u-boot_xsbase270/build/

make xsbase270_config

make
```

作其他必要修改，直至能正常编译通过。然后再进行后续的针对目标板的定制步骤。

2. 修改 U-Boot Stage 1(汇编级)的平台相关代码

U-Boot 第一阶段的代码包括：

(1) cpu/pxa/start.S (平台无关，处理器架构相关)

(2) board/xsbase270/lowlevel_init.S (平台与处理器型号相关)

(3) board/xsbase270/config.mk (平台相关，设置 TEXT_BASE)

(4) include/configs/xsbase270.h (平台相关，设置寄存器初值等)

lowlevel_init.S 已在第一步作了相应修改。config.mk 中设置 TEXT_BASE(U-Boot 的链接起始地址)，暂时不改动(0xa3080000)。

xsbase270.h 中定义了系统初始化时的寄存器初值(主要是 GPIO 配置，时钟与处理器频率设置，片上存储器控制器与存储系统的初始化)，这需要根据平台进行配置。下面给出部分代码示例及注释：

```
/*

* High Level Configuration Options (easy to
change)

*/

#define CONFIG_PXA27X 1 /*to keep PXA27x
specific code*/

#define CONFIG_XSBASE270 1

#define BOARD_LATE_INIT 1

#undef CONFIG_USE_IRQ /* we don't need
IRQ/FIQ stuff */

.....
```

```
/*
* Size of malloc() pool
*/
#define CFG_MALLOC_LEN (CFG_ENV_SIZE + 128*1024)
#define CFG_GBL_DATA_SIZE 128
/*
* Stack sizes
* The stack sizes are set up in start.S using the settings below
*/
#define CONFIG_STACKSIZE (128*1024) /* regular stack */
#ifdef CONFIG_USE_IRQ
#define CONFIG_STACKSIZE_IRQ (4*1024) /* IRQ stack */
#define CONFIG_STACKSIZE_FIQ (4*1024) /* FIQ stack */
#endif
/*
* Miscellaneous configurable options
*/
#define CFG_CPUSPEED 0x207 /* cpu start-up frequency,91MHz */
/*
* GPIO settings
```

```
*/

#define CFG_GPSR0_VAL 0x00003000

#define CFG_GPSR1_VAL 0x00000000

#define CFG_GPSR2_VAL 0x00010000

#define CFG_GPSR3_VAL 0x00020000

#define CFG_GPCR0_VAL 0x00000800

.....

/*

* Clock settings

*/

#define CFG_CKEN 0x00400200

#define CFG_CCCR 0x08000290 /* 520 MHz */

/*

* Memory settings

*/

#define CFG_MSC0_VAL 0x7FF82BD0

#define CFG_MSC1_VAL 0x7FF87FF8

#define CFG_MSC2_VAL 0x7FF87FF8

#define CFG_MDCNFG_VAL 0x00001AC9

#define CFG_MDREFR_VAL 0x0000001E

#define CFG_MDMRS_VAL 0x00000000

.....

/*
```

```
* PCMCIA and CF Interfaces

*/

#define CFG_MECR_VAL 0x00000001

#define CFG_MCMEM0_VAL 0x00010504

#define CFG_MCMEM1_VAL 0x00010504

.....
```

3. 修改 U-Boot Stage 2(C 语言级)的平台相关代码

U-Boot 第二阶段的大部分代码是平台无关的。从移植的角度，我们仅需要关注下面一些平台相关的代码：

(1) include/configs/xsbase270.h：通过使用定义或取消定义相关的预编译变量，用于对平台无关的代码进行平台相关的定制，包括定制 U-Boot 命令、缺省的环境变量、存储器映射、串口控制台配置、驱动程序等。

(2) board/xsbase270/xsbase270.c：板级初始化，只需进行最基本的配置，包括设置 mach-type，启动参数列表首地址，设置标准输入输出设备，获取系统 RAM 配置信息等。

(3) 驱动程序的移植。最基本的是 FLASH 存储器驱动程序和以太网卡驱动程序。对于 U-Boot 中已经支持的器件，可以进行简单移植，否则需要自己加入相关的设备驱动程序。

下面对以上三部分分别阐述。

3.1 配置 xsbase270.h

可以参考 lubbock.h，adsvix.h 等相关开发板的设置，另外也可以从 U-Boot 源码的 README 文件获取更多信息。

(1) 存储器映射配置：

```
/*

* Physical Memory Map
```

```
*/

#define CONFIG_NR_DRAM_BANKS 4 /* we
have 2 banks of DRAM*/

#define PHYS_SDRAM_1 0xa0000000 /* SDRAM
Bank #1 */

#define PHYS_SDRAM_1_SIZE 0x04000000 /* 64
MB */

#define PHYS_SDRAM_2 0xa4000000 /* SDRAM
Bank #2 */

#define PHYS_SDRAM_2_SIZE 0x00000000 /* 0
MB */

#define PHYS_SDRAM_3 0xa8000000 /* SDRAM
Bank #3 */

#define PHYS_SDRAM_3_SIZE 0x00000000 /* 0
MB */

#define PHYS_SDRAM_4 0xac000000 /* SDRAM
Bank #4 */

#define PHYS_SDRAM_4_SIZE 0x00000000 /* 0
MB */



#define PHYS_FLASH_1 0x00000000 /* Flash
Bank #1 */

#define PHYS_FLASH_2 0x04000000 /* Flash
Bank #2 */

#define PHYS_FLASH_SIZE 0x02000000 /* 32
MB */

#define PHYS_FLASH_BANK_SIZE 0x02000000
/* 32 MB Banks */

#define PHYS_FLASH_SECT_SIZE 0x00040000
```

```
/* 256 KB sectors (x2) */



#define CFG_DRAM_BASE 0xa0000000

#define CFG_DRAM_SIZE 0x04000000



#define CFG_FLASH_BASE PHYS_FLASH_1

//you can also add other IO address map here, such as a FPGA
```

(2) 定制 U-Boot 命令：

在 include/config_cmd_default.h 头文件中已经预定义了一些常用的 U-Boot 命令，我们可以在 include/configs/xsbase270.h 中包含该头文件，对于其中已定义的不需要的命令，可用 undef 去除；对于要添加的命令，使用 define 定义相关的符号即可。

```
/*

* Command line configuration.

*/

#include <config_cmd_default.h>



#define CONFIG_CMD_PING
```

(3) 控制台串口配置：

包括指定控制台所用的 PXA27X 串口，缺省的串口通信波特率等。

```
/*

* select serial console configuration

*/

#define CONFIG_FFUART 1 /* we use FFUART on XSBASE270 */

#define CONFIG_BAUDRATE 115200
```

(4) 环境变量设置

包括 BOOTP 选项设置，缺省环境变量设置，启动参数列表配置等。

```
/*
* BOOTP options
*/
#define CONFIG_BOOTP_BOOTFILESIZE
#define CONFIG_BOOTP_BOOTPATH
#define CONFIG_BOOTP_HOSTNAME


#define CONFIG_BOOTDELAY 3
#define CONFIG_ETHADDR 08:00:3e:26:0a:5b
#define CONFIG_NETMASK 255.255.255.0
#define CONFIG_IPADDR 192.168.0.21
#define CONFIG_SERVERIP 192.168.0.250
#define CONFIG_BOOTCOMMAND "bootm
80000"
#define CONFIG_BOOTARGS
"root=/dev/ram0,rw mem=64M console=ttyS0,
115200"
#define CONFIG_CMDLINE_TAG
#define CONFIG_TIMESTAMP
/* allow to overwrite serial and ethaddr */
#define CONFIG_ENV_OVERWRITE
```

其中，CONFIG_BOOTCOMMAND 和 CONFIG_BOOTARGS 在后续的引导内核实验中还需要进行修正。

(5) 网卡驱动程序配置：

```
/*

* Hardware drivers

*/

#define CONFIG_DRIVER_SMC91111

#define CONFIG_SMC91111_BASE
0x0C000000

#define CONFIG_SMC_USE_32_BIT 1
```

XSBASE270 采用的网卡是 LAN91C111，U-Boot 自带的驱动程序 drivers/smc91111.c 可支持这款网卡，因此只要在这里作相应的配置即可。CONFIG_SMC91111_BASE 要根据 PXA27X 对网卡的地址译码来决定(片选信号 CSx 和高位地址线)，CONFIG_SMC_USE_32_BIT 指定了网卡工作于 32 位数据总线模式。可以查看驱动程序源代码得到更多配置选项。

(6) NOR 型闪存驱动程序配置：

U-Boot 本身支持一系列符合 CFI(Common Flash Interface)接口规范的闪存，其缺省支持的闪存芯片信息在 include/flash.h 中定义，该头文件中还定义了 CFI 闪存驱动所必需的数据结构和其他物理及结构特性描述符。NAND 闪存驱动在 drivers/nand 目录下，这里不予考虑。CFI 是针对 NOR 型 FLASH 所提出的一种获取闪存芯片物理和结构参数的操作规程和标准。

XSBASE270 采用两片 Intel 28F128K18C 的兼容 CFI 标准的 NOR 型闪存，单片容量为 16MB，数据线宽度为 16-bit，两片并作一个 32MB 容量的数据宽度为 32-bit 的 BANK 来使用。在头文件 include/flash.h 中没有定义该芯片的相关信息，可以手动添加；这并不是必须的，如果你并不需要使用这些信息的话(例如将 CFI 驱动所检测到的 Device Id 与头文件中定义的 Device ID 进行比对与验证)。

```
/* file : include/flash.h */

#define INTEL_ID_28F128K18 0x88068806 /*
added by aaron */

#define FLASH_28F128K18 0x00BA /*Intel
28F128K18 (128M=8Mx16)*/
```

要使用 U-Boot 自带的 CFI 闪存驱动，必须要作的是在 include/configs/xsbase270.h 中添加如下定义:

```
#define CFG_FLASH_CFI
```

```
#define CFG_FLASH_CFI_DRIVER 1

/* avoid long time detection, added by aaron ,see
include/flash.h */

#define CFG_FLASH_CFI_WIDTH
FLASH_CFI_32BIT



#define CFG_MAX_FLASH_BANKS 1 /* max
number of memory banks */

#define CFG_MAX_FLASH_SECT 128 /*max
number of sectors on one chip*/



/* timeout values are in ticks */

#define CFG_FLASH_ERASE_TOUT
(25*CFG_HZ) /*Timeout for Flash Erase */

#define CFG_FLASH_WRITE_TOUT
(25*CFG_HZ) /*Timeout for Flash Write */



/* write flash less slowly */

#define CFG_FLASH_USE_BUFFER_WRITE 1
```

另外，如果把环境变量保存在 FLASH 中，还有如下相关定义：

```
/* NOTE: many default partitioning schemes
assume the kernel starts at

* the second sector, not an environment. You have
been warned!

*/

#define CFG_MONITOR_BASE 0

#define CFG_MONITOR_LEN
```

```
PHYS_FLASH_SECT_SIZE

#define CFG_ENV_IS_IN_FLASH 1

#define CFG_ENV_ADDR (PHYS_FLASH_1 +
PHYS_FLASH_SECT_SIZE)

#define CFG_ENV_SECT_SIZE
PHYS_FLASH_SECT_SIZE

#define CFG_ENV_SIZE
(PHYS_FLASH_SECT_SIZE / 16)

/* If defined, hardware flash sectors protection is
used instead of

* U-Boot software protection. */

#define CFG_FLASH_PROTECTION
```

(7) 其他配置：

```
/*

* Miscellaneous configurable options

*/

#define CFG_HUSH_PARSER 1

#define CFG_PROMPT_HUSH_PS2 "> "

#define CFG_LONGHELP /* undef to save
memory */

#ifdef CFG_HUSH_PARSER

#define CFG_PROMPT "$ " /* Monitor Command
Prompt */

#endif

#define CFG_CBSIZE 256 /* Console I/O Buffer
```

```
Size*/

/* Print Buffer Size */

#define CFG_PBSIZE
(CFG_CBSIZE+sizeof(CFG_PROMPT)+16)

#define CFG_MAXARGS 16 /* max number of
command args */

#define CFG_BARGSIZE CFG_CBSIZE /*Boot
Argument Buffer Size*/

#define CFG_DEVICE_NULLDEV 1



#define CFG_MEMTEST_START 0xa0400000 /*
memtest works on */

#define CFG_MEMTEST_END 0xa0800000 /* 4 ...
8 MB in DRAM */



#undef CFG_CLKS_IN_HZ /* everything, incl
board info, in Hz */

/*default load address */

#define CFG_LOAD_ADDR (CFG_DRAM_BASE
+ 0x8000)



#define CFG_HZ 3686400 /* incrementer freq:
3.6864 MHz */

/* valid baudrates */

#define CFG_BAUDRATE_TABLE { 9600, 19200,
38400, 57600, 115200 }
```

至此，目标板配置头文件 xsbase270.h 就完成了。

3.2 板级初始化代码 xsbase270.c

只需修改 board_init()函数即可，完整代码如下：

```
#include <common.h>

DECLARE_GLOBAL_DATA_PTR;

/* Miscelaneous platform dependent initialisations
*/



int board_init (void)

{

/* memory and cpu-speed are setup before
relocation */

/* so we do _nothing_ here */

/* arch number of XSBASE270-Board */

gd->bd->bi_arch_number =
MACH_TYPE_XSBASE270;



/* adress of boot parameters */

gd->bd->bi_boot_params = 0xa0000100;



return 0;

}



int board_late_init(void)
```

```
{

setenv("stdout", "serial");

setenv("stderr", "serial");

return 0;

}



int dram_init (void)

{

gd->bd->bi_dram[0].start = PHYS_SDRAM_1;

gd->bd->bi_dram[0].size =
PHYS_SDRAM_1_SIZE;

gd->bd->bi_dram[1].start = PHYS_SDRAM_2;

gd->bd->bi_dram[1].size =
PHYS_SDRAM_2_SIZE;

gd->bd->bi_dram[2].start = PHYS_SDRAM_3;

gd->bd->bi_dram[2].size =
PHYS_SDRAM_3_SIZE;

gd->bd->bi_dram[3].start = PHYS_SDRAM_4;

gd->bd->bi_dram[3].size =
PHYS_SDRAM_4_SIZE;



return 0;

}
```

3.3 驱动程序移植

最主要的是闪存和网卡驱动程序的移植。由于使用 U-Boot 自带的 CFI 闪存驱动程序和 SMC91111 网卡驱动程序，应此只要在头文件中进行相关配置即可完成。具体见 3.1 节。如果需要自行添加相关的设备驱动，则需要在 board/xsbase270/目录下添加驱动源文件，并将其添加到该目录下的 Makefile 中进行编译与链接。

至此，针对特定目标板的 U-Boot 软件移植工作基本完成。在下一节中，将简单讨论 U-Boot 的基本的硬件调试方法与技巧

U-Boot 的移植之(四)调试篇：下载 U-Boot 到目标板进行调试

编译完成之后，得到的几个重要文件是：

(1) u-boot.bin: 116K，原始二进制文件，用于下载到启动 ROM 进行系统引导；

(2) u-boot: 384K，ELF 格式映像文件，可加载到 SDRAM 或 SRAM 中进行调试；

(3) u-boot.srec: Motorola S-Records 格式映像。

(4) System.map: U-Boot 映像文件的符号表，各符号的链接地址。

最有效的调试方法是下载到目标板的启动闪存，使用硬件仿真器进行跟踪调试。使用 Skyeye，Qemu 等软件仿真器不能达到真实的调试效果，尤其不能真实反映第一阶段的底层初始化过程，只适合作 U-Boot 的学习与研究之用。有人提出在没有硬件仿真器的情况下，使用"点灯大法(利用目标板的 LED 指示程序运行阶段)"进行跟踪调试，这实际上无异于盲人摸象，特别是在底层初始化阶段，一条指令就可能导致异常。也有人提出注释掉 start.S 中的 lowlevel_init 调用，将 U-Boot 映像加载到 SDRAM 中进行调试，这实际上只能对 U-Boot 进行功能调试，而无法跟踪 U-Boot 的底层初始化过程。当然，如果实在嫌烧写 FLASH 的速度较慢，又心疼其擦写寿命，也可以将 U-Boot 映像加载到片上 SRAM 中调试，因为 U-Boot 的开始一部分代码是位置无关的(除了后 6 个异常向量外，不过并不构成影响)；这要求片上 SRAM 够大，因为 U-Boot 的映像大小约有 300K－400K。

笔者使用 Banyan-U ARM EMULATOR JTAG 仿真器，结合 AXD 软件平台进行调试。

首先将 u-boot.bin 下载到 FLASH 地址 0x0，连接好串口，启动 minicom 或超级终端，目标板上电后，串口控制台无任何输出。这很有可能是 lowlevel_init 那段代码出了问题，因为它牵涉到 GPIO 的配置，处理器时钟频率设置，系统总线频率与存储器的时序匹配及初始化，稍有差错就会当机。当然也有可能是串口的配置不正确，但这部分比较简单，出错的可能性比较小。

对于下载到 FLASH 存储器的原始二进制文件，只能进行汇编级的跟踪调试。先利用 objdump 工具生成 U-Boot 映像的反汇编代码：

```
arm-iwmmxt-linux-gnueabi-objdump -S u-boot > u-boot.S
```

反汇编代码 u-boot.S 和符号表 System.map 将是跟踪调试过程中的得力助手。

另一个重要的调试技巧是在 AXD 中现场修改寄存器和存储单元的内容，这样可以帮助我们找到问题所在，而不必每次改动都重新编译 u-boot，也避免了 FLASH 的频繁烧写。

例如，笔者在单步跟踪调试时，发现在地址 0xa30804b4 处，使用指令″str r1, [r0]”配置 GPDR1(GPIO 方向寄存器 1)后，存储器 0x0 地址开始的大片内容全部被更改，导致异常终止。这时可以在该处设置一个断点，复位目标板全速运行到断点处，修改寄存器 r1 的值(GPDR1 的初值)，再执行该条指令。经试验发现，对于 XSBASE270 开发板，必须要先初始化 GAFRx，再初始化 GPDRx，才不致于发生上述异常。而在 start.S 中，是先完成 GPDRx 的初始化之后，再初始化 GAFRx 的，因此需要在源代码中将这两段代码的位置互换，重新编译后，再下载到 FLASH 中。

U-Boot 的串口控制台输出如下：

```
U-Boot 1.3.0-rc2 (Oct 16 2007 - 01:57:29)

DRAM: 64 MB

Flash: 32 MB

In: serial

Out: serial

Err: serial

Hit any key to stop autoboot: 0

$
```

另一个问题是环境变量的设置与保存。将环境变量保存在 FLASH 中，使用 setenv 命令设置环境变量，再使用 saveenv 命令保存，这样在下次开机时，就会使用新的环境变量。如果使用的是 U-Boot 自带的 CFI 闪存驱动，在保存环境变量时可能会出现如下问题：

```
$ setenv ipaddr 192.168.1.21

$ saveenv

Saving Environment to Flash...
```

```
Un-Protected 1 sectors

Erasing Flash...

Flash erase error at address 40000

Block Erase Error.

Block locked.

done

Erased 1 sectors
```

这是因为缺省情况下 U-Boot 对 FLASH 有软件写保护，这时在 U-Boot 启动完毕后即使使用 jflashmm 工具也无法对 FLASH 进行烧写：

```
[aaronwong@localhost Jflash-XSBase270]$ sudo ./jflashmm u-boot.bin



JFLASH Version 5.01.007

COPYRIGHT (C) 2000 - 2003 Intel Corporation



PLATFORM SELECTION:

Processor= PXA27x

Development System= XSBase270

Data Version= 1.00.001



PXA27x revision ??

Found flash type: 28F128K18



Erasing block at address 0

Error, Block erase timed out
```

解决办法可参考 Uboot-Users 邮件列表 Erase error on dual P30(CFI) flash chips 主题讨论，具体是在 include/configs/xsbase270.h 中定义 CFG_FLASH_PROTECTION，该选项在 README 文件中的描述如下：

```
- CFG_FLASH_PROTECTION

If defined, hardware flash sectors protection is used

instead of U-Boot software protection.
```

修改完毕重新编译 U-Boot，在目标板上电后 U-Boot 启动完毕之前，使用 jflashmm 工具将新的 u-boot.bin 烧写到目标板启动闪存。这时可成功修改环境变量并保存到 FLASH 中：

```
$ setenv ipaddr 192.168.1.21

$ saveenv

Saving Environment to Flash...

. done

Un-Protected 1 sectors

Erasing Flash...

. done

Erased 1 sectors

Writing to Flash... done

. done

Protected 1 sectors
```

一旦 U-Boot 的基本功能调试通过，能正常在目标板运行，剩余的工作就是根据实际情况调整 TEXT_BASE 以及内核引导参数，使用 U-Boot 来引导 Linux 内核。在下一节中，将给出 U-Boot 引导 Linux 内核的实例。

u-boot 源码结构

从网站上下载得到 U-Boot 源码包，例如：U-Boot-1.1.2.tar.bz2

解压就可以得到全部 U-Boot 源程序。在顶层目录下有 18 个子目录，分别存放和管理不同的源程序。这些目录中所要存放的文件有其规则，可以分为 3 类。

- 第 1 类目录与处理器体系结构或者开发板硬件直接相关；
- 第 2 类目录是一些通用的函数或者驱动程序；
- 第 3 类目录是 U-Boot 的应用程序、工具或者文档。

表 6.2 列出了 U-Boot 顶层目录下各级目录存放原则。

表 6.2　　U-Boot 的源码顶层目录说明

目　录	特　性	解 释 说 明
board	平台依赖	存放电路板相关的目录文件，例如：RPXlite(mpc8xx)、smdk2410(arm920t)、sc520_cdp(x86) 等目录
cpu	平台依赖	存放 CPU 相关的目录文件，例如：mpc8xx、ppc4xx、arm720t、arm920t、 xscale、i386 等目录
lib_ppc	平台依赖	存放对 PowerPC 体系结构通用的文件，主要用于实现 PowerPC 平台通用的函数
目　录	特　性	解 释 说 明
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
lib_i386	平台依赖	存放对 X86 体系结构通用的文件，主要用于实现 X86 平台通用的函数
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
common	通用	通用的多功能函数实现
lib_generic	通用	通用库函数的实现
Net	通用	存放网络的程序
Fs	通用	存放文件系统的程序
Post	通用	存放上电自检程序
drivers	通用	通用的设备驱动程序，主要有以太网接口的驱动
Disk	通用	硬盘接口程序
Rtc	通用	RTC 的驱动程序
Dtt	通用	数字温度测量器或者传感器的驱动
examples	应用例程	一些独立运行的应用程序的例子，例如 helloworld

tools	工具	存放制作 S-Record 或者 U-Boot 格式的映像等工具，例如 mkimage
Doc	文档	开发使用文档

U-Boot 的源代码包含对几十种处理器、数百种开发板的支持。可是对于特定的开发板，配置编译过程只需要其中部分程序。这里具体以 S3C2410 arm920t 处理器为例，具体分析 S3C2410 处理器和开发板所依赖的程序，以及 U-Boot 的通用函数和工具。

u-boot 的编译

U-Boot 的源码是通过 GCC 和 Makefile 组织编译的。顶层目录下的 Makefile 首先可以设置开发板的定义，然后递归地调用各级子目录下的 Makefile，最后把编译过的程序链接成 U-Boot 映像。

1．顶层目录下的 Makefile

它负责 U-Boot 整体配置编译。按照配置的顺序阅读其中关键的几行。

每一种开发板在 Makefile 都需要有板子配置的定义。例如 smdk2410 开发板的定义如下。

```
smdk2410_config :   unconfig

    @./mkconfig $(@:_config=) arm arm920t smdk2410 NULL s3c24x0
```

执行配置 U-Boot 的命令 make smdk2410_config，通过./mkconfig 脚本生成 include/config.
mk 的配置文件。文件内容正是根据 Makefile 对开发板的配置生成的。

```
ARCH   = arm

CPU    = arm920t

BOARD  = smdk2410
```

```
SOC    = s3c24x0
```

上面的 include/config.mk 文件定义了 ARCH、CPU、BOARD、SOC 这些变量。这样硬件平台依赖的目录文件可以根据这些定义来确定。SMDK2410 平台相关目录如下。

board/smdk2410/

cpu/arm920t/

cpu/arm920t/s3c24x0/

lib_arm/

include/asm-arm/

include/configs/smdk2410.h

再回到顶层目录的 Makefile 文件开始的部分，其中下列几行包含了这些变量的定义。

```
# load ARCH, BOARD, and CPU configuration

include include/config.mk

export        ARCH CPU BOARD VENDOR SOC
```

Makefile 的编译选项和规则在顶层目录的 config.mk 文件中定义。各种体系结构通用的规则直接在这个文件中定义。通过 ARCH、CPU、BOARD、SOC 等变量为不同硬件平台定义不同选项。不同体系结构的规则分别包含在 ppc_config.mk、arm_config.mk、mips_config.mk 等文件中。

顶层目录的 Makefile 中还要定义交叉编译器，以及编译 U-Boot 所依赖的目标文件。

```
ifeq ($(ARCH),arm)

CROSS_COMPILE = arm-linux-          //交叉编译器的前缀
```

```
#endif

export   CROSS_COMPILE

…

# U-Boot objects....order is important (i.e. start must be first)

OBJS   = cpu/$(CPU)/start.o                    //处理器相关的目标文件

…

LIBS   = lib_generic/libgeneric.a              //定义依赖的目录，每个目录下先把目标文件连接成
*.a 文件。

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

…
```

然后还有 U-Boot 映像编译的依赖关系。

```
ALL = u-boot.srec u-boot.bin System.map

all:        $(ALL)

u-boot.srec:    u-boot

        $(OBJCOPY) ${OBJCFLAGS} -O srec $< $@

u-boot.bin: u-boot
```

```
        $(OBJCOPY) ${OBJCFLAGS} -O binary $< $@

……

u-boot:         depend $(SUBDIRS) $(OBJS) $(LIBS) $(LDSCRIPT)

        UNDEF_SYM='$(OBJDUMP) -x $(LIBS) \

        |sed  -n -e 's/.*\(__u_boot_cmd_.*\)/-u\1/p'|sort|uniq`;\

        $(LD) $(LDFLAGS) $$UNDEF_SYM $(OBJS) \

            --start-group $(LIBS) $(PLATFORM_LIBS) --end-group \

            -Map u-boot.map -o u-boot
```

Makefile 缺省的编译目标为 all，包括 u-boot.srec、u-boot.bin、System.map。u-boot.srec 和 u-boot.bin 又依赖于 U-Boot。U-Boot 就是通过 ld 命令按照 u-boot.map 地址表把目标文件组装成 u-boot。

其他 Makefile 内容就不再详细分析了，上述代码分析应该可以为阅读代码提供了一个线索。

2．开发板配置头文件

除了编译过程 Makefile 以外，还要在程序中为开发板定义配置选项或者参数。这个头文件是 include/configs/*<board_name>*.h。*<board_name>*用相应的 BOARD 定义代替。

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。例如：

```
#define   CONFIG_ARM920T        1

#define   CONFIG_DRIVER_CS8900  1
```

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。例如：

```
#define     CFG_FLASH_BASE     0x00000000

#define CFG_PROMPT          "=>"
```

3．编译结果

根据对 Makefile 的分析，编译分为 2 步。第 1 步配置，例如：make smdk2410_config；第 2 步编译，执行 make 就可以了。

编译完成后，可以得到 U-Boot 各种格式的映像文件和符号表，如表 6.3 所示。

表 6.3　　U-Boot 编译生成的映像文件

文 件 名 称	说　明	文 件 名 称	说　明
System.map	U-Boot 映像的符号表	u-boot.bin	U-Boot 映像原始的二进制格式
u-boot	U-Boot 映像的 ELF 格式	u-boot.srec	U-Boot 映像的 S-Record 格式

U-Boot 的 3 种映像格式都可以烧写到 Flash 中，但需要看加载器能否识别这些格式。一般 u-boot.bin 最为常用，直接按照二进制格式下载，并且按照绝对地址烧写到 Flash 中就可以了。U-Boot 和 u-boot.srec 格式映像都自带定位信息。

4．U-Boot 工具

在 tools 目录下还有些 U-Boot 的工具。这些工具有的也经常用到。表 6.4 说明了几种工具的用途。

表 6.4　　U-Boot 的工具

工 具 名 称	说　明	工 具 名 称	说　明
bmp_logo	制作标记的位图结构体	img2srec	转换 SREC 格式映像
envcrc	校验 u-boot 内部嵌入的环境变量	mkimage	转换 U-Boot 格式映像
gen_eth_addr	生成以太网接口 MAC 地址	updater	U-Boot 自动更新升级工具

这些工具都有源代码，可以参考改写其他工具。其中 mkimage 是很常用的一个工具，Linux 内核映像和 ramdisk 文件系统映像都可以转换成 U-Boot 的格式。

u-boot 的移植

U-Boot 能够支持多种体系结构的处理器，支持的开发板也越来越多。因为 Bootloader 是完全依赖硬件平台的，所以在新电路板上需要移植 U-Boot 程序。

开始移植 U-Boot 之前，先要熟悉硬件电路板和处理器。确认 U-Boot 是否已经支持新开发板的处理器和 I/O 设备。假如 U-Boot 已经支持一块非常相似的电路板，那么移植的过程将非常简单。

移植 U-Boot 工作就是添加开发板硬件相关的文件、配置选项，然后配置编译。

开始移植之前，需要先分析一下 U-Boot 已经支持的开发板，比较出硬件配置最接近的开发板。选择的原则是，首先处理器相同，其次处理器体系结构相同，然后是以太网接口等外围接口。还要验证一下这个参考开发板的 U-Boot，至少能够配置编译通过。

以 S3C2410 处理器的开发板为例,U-Boot-1.1.2 版本已经支持 SMDK2410 开发板。我们可以基于 SMDK2410 移植，那么先把 SMDK2410 编译通过。

我们以 S3C2410 开发板 fs2410 为例说明。移植的过程参考 SMDK2410 开发板，SMDK2410 在 U-Boot-1.1.2 中已经支持。

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

（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

（5）编译 U-Boot

执行 make 命令，编译成功可以得到 U-Boot 映像。有些错误是跟配置选项是有关系的，通常打开某些功能选项会带来一些错误，一开始可以尽量跟参考板配置相同。

（6）添加驱动或者功能选项

在能够编译通过的基础上，还要实现 U-Boot 的以太网接口、Flash 擦写等功能。

对于 FS2410 开发板的以太网驱动和 smdk2410 完全相同，所以可以直接使用。CS8900 驱动程序文件如下。

drivers/cs8900.c

drivers/cs8900.h

对于 Flash 的选择就麻烦多了，Flash 芯片价格或者采购方面的因素都有影响。多数开发板大小、型号不都相同。所以还需要移植 Flash 的驱动。每种开发板目录下一般都有 flash.c 这个文件，需要根据具体的 Flash 类型修改。例如：

board/fs2410/flash.c

（7）调试 U-Boot 源代码，直到 U-Boot 在开发板上能够正常启动。

调试的过程可能是很艰难的，需要借助工具，并且有些问题可能困扰很长时间。

u-boot 命令的添加

U-Boot 的命令为用户提供了交互功能，并且已经实现了几十个常用的命令。如果开发板需要很特殊的操作，可以添加新的 U-Boot 命令。

U-Boot 的每一个命令都是通过 U_Boot_CMD 宏定义的。这个宏在 include/command.h 头文件中定义，每一个命令定义一个 cmd_tbl_t 结构体。

```
#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}
```

这样每一个 U-Boot 命令有一个结构体来描述。结构体包含的成员变量：命令名称、最大参数个数、重复数、命令执行函数、用法、帮助。

从控制台输入的命令是由 common/command.c 中的程序解释执行的。find_cmd()负责匹配输入的命令，从列表中找出对应的命令结构体。

基于 U-Boot 命令的基本框架，来分析一下简单的 icache 操作命令，就可以知道添加新命令的方法。

（1）定义 CACHE 命令。在 include/cmd_confdefs.h 中定义了所有 U-Boot 命令的标志位。

```
#define CFG_CMD_CACHE       0x00000010ULL   /* icache, dcache       */
```

如果有更多的命令，也要在这里添加定义。

（2）实现 CACHE 命令的操作函数。下面是 common/cmd_cache.c 文件中 icache 命令部分的代码。

```
#if (CONFIG_COMMANDS & CFG_CMD_CACHE)
static int on_off (const char *s)
{       //这个函数解析参数，判断是打开 cache，还是关闭 cache

        if (strcmp(s, "on") == 0) {  //参数为 “on”
                return (1);
        } else if (strcmp(s, "off") == 0) {  //参数为 “off”
                return (0);
    }
    return (-1);
}

int do_icache ( cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{     //对指令 cache 的操作函数
      switch (argc) {
      case 2:                /* 参数个数为 1，则执行打开或者关闭指令 cache 操作 */
            switch (on_off(argv[1])) {
            case 0:      icache_disable();        //打开指令 cache
                 break;
            case 1:      icache_enable ();        //关闭指令 cache
                 break;
```

```
            }
            /* FALL TROUGH */
        case 1:             /* 参数个数为 0，则获取指令 cache 状态*/
            printf ("Instruction Cache is %s\n",
                    icache_status() ? "ON" : "OFF");
            return 0;
        default:  //其他缺省情况下，打印命令使用说明
            printf ("Usage:\n%s\n", cmdtp->usage);
            return 1;
        }
        return 0;
}
……
U_Boot_CMD( //通过宏定义命令
    icache,  2,  1,     do_icache,  //命令为 icache，命令执行函数为 do_icache()
    "icache   - enable or disable instruction cache\n",   //帮助信息
    "[on, off]\n"
    "     - enable or disable instruction cache\n"
);
……
#endif
```

U-Boot 的命令都是通过结构体__U_Boot_cmd_##name 来描述的。根据 U_Boot_CMD 在 include/command.h 中的两行定义可以明白。

```
#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}
```

还有，不要忘了在 common/Makefile 中添加编译的目标文件。

（3）打开 CONFIG_COMMANDS 选项的命令标志位。这个程序文件开头有#if 语句需要预处理是否包含这个命令函数。CONFIG_COMMANDS 选项在开发板的配置文件中定义。例如：SMDK2410 平台在 include/configs/smdk2410.h 中有如下定义。

```
/************************************************************

 * Command definition

 ************************************************************/

#define CONFIG_COMMANDS \

            (CONFIG_CMD_DFL  | \

            CFG_CMD_CACHE    | \

            CFG_CMD_REGINFO  | \

            CFG_CMD_DATE     | \

            CFG_CMD_ELF)
```

按照这 3 步，就可以添加新的 U-Boot 命令。

u-boot 启动过程

尽管有了调试跟踪手段，甚至也可以通过串口打印信息了，但是不一定能够判断出错原因。如果能够充分理解代码的启动流程，那么对准确地解决和分析问题很有帮助。

开发板上电后，执行 U-Boot 的第一条指令，然后顺序执行 U-Boot 启动函数。函数调用顺序如图 6.3 所示。

看一下 board/smsk2410/u-boot.lds 这个链接脚本，可以知道目标程序的各部分链接顺序。第一个要链接的是 cpu/arm920t/start.o，那么 U-Boot 的入口指令一定位于这个程序中。下面详细分析一下程序跳转和函数的调用关系以及函数实现。

1. cpu/arm920t/start.S

这个汇编程序是 U-Boot 的入口程序，开头就是复位向量的代码。

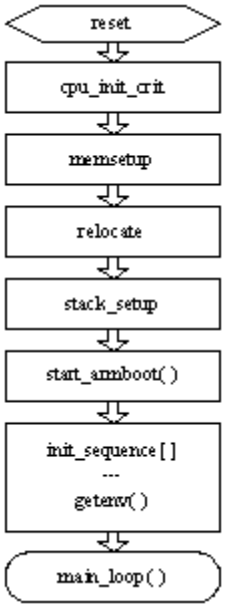


图 6.3　U-Boot 启动代码流程图

```
_start: b       reset       //复位向量

        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

…

 /* the actual reset code   */

reset:          //复位启动子程序
```

```
    /* 设置 CPU 为 SVC32 模式 */

    mrs    r0,cpsr

    bic    r0,r0,#0x1f

    orr    r0,r0,#0xd3

    msr    cpsr,r0

/* 关闭看门狗 */



/* 这些初始化代码在系统重起的时候执行，运行时热复位从 RAM 中启动不执行 */

#ifdef CONFIG_INIT_CRITICAL

    bl     cpu_init_crit

#endif



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

    adr    r0, _start          /* r0 是代码的当前位置 */

    ldr    r1, _TEXT_BASE      /* 测试判断是从 Flash 启动，还是 RAM */

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

    beq    stack_setup     /* 如果 r0 等于 r1，跳过重定位代码 */

    /* 准备重新定位代码 */

    ldr    r2, _armboot_start
```

```
        ldr    r3, _bss_start

        sub    r2, r3, r2           /* r2 得到 armboot 的大小    */

        add    r2, r0, r2           /* r2 得到要复制代码的末尾地址 */

copy_loop: /* 重新定位代码 */

        ldmia r0!, {r3-r10}   /*从源地址[r0]复制 */

        stmia r1!, {r3-r10}   /* 复制到目的地址[r1] */

        cmp    r0, r2           /* 复制数据块直到源数据末尾地址[r2] */

        ble    copy_loop


        /* 初始化堆栈等    */

stack_setup:

        ldr    r0, _TEXT_BASE               /* 上面是 128 KiB 重定位的 u-boot */

        sub    r0, r0, #CFG_MALLOC_LEN      /* 向下是内存分配空间 */

        sub    r0, r0, #CFG_GBL_DATA_SIZE /* 然后是 bdinfo 结构体地址空间   */

#ifdef CONFIG_USE_IRQ

        sub    r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)

#endif

        sub    sp, r0, #12      /* 为 abort-stack 预留 3 个字 */
```

```
clear_bss:
        ldr    r0, _bss_start       /* 找到 bss 段起始地址 */
        ldr    r1, _bss_end         /*  bss 段末尾地址   */
        mov    r2, #0x00000000      /* 清零 */
clbss_l:str r2, [r0]          /* bss 段地址空间清零循环...  */
        add    r0, r0, #4
        cmp    r0, r1
        bne    clbss_l
        /* 跳转到 start_armboot 函数入口，_start_armboot 字保存函数入口指针 */
        ldr    pc, _start_armboot
_start_armboot: .word start_armboot      //start_armboot 函数在 lib_arm/board.c 中实现
/* 关键的初始化子程序 */
cpu_init_crit:
……  //初始化 CACHE，关闭 MMU 等操作指令
        /* 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     memsetup        //memsetup 子程序在 board/smdk2410/memsetup.S 中实现

mov    lr, ip

mov    pc, lr
```

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。

```
void start_armboot (void)

{

    DECLARE_GLOBAL_DATA_PTR;

    ulong size;

    init_fnc_t **init_fnc_ptr;

    char *s;

    /* Pointer is writable since we allocated a register for it */

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

    /* compiler optimization barrier needed for GCC >= 3.4 */

    __asm__ __volatile__("": : :"memory");

    memset ((void*)gd, 0, sizeof (gd_t));

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

    memset (gd->bd, 0, sizeof (bd_t));

    monitor_flash_len = _bss_start - _armboot_start;
```

```
/* 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 ();

display_flash_config (size);

/* _armboot_start 在 u-boot.lds 链接脚本中定义 */

mem_malloc_init (_armboot_start - CFG_MALLOC_LEN);

/* 配置环境变量，重新定位 */

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 ();       /* 获取列表中的设备 */

jumptable_init ();

console_init_r ();     /* 完整地初始化控制台设备 */
```

```
        enable_interrupts (); /* 使能例外处理 */

        /* 通过环境变量初始化 */

        if ((s = getenv ("loadaddr")) != NULL) {

                load_addr = simple_strtoul (s, NULL, 16);

        }

        /* main_loop()总是试图自动启动，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 */

        }

        /* NOTREACHED - no way out of command loop except booting */

}
```

3. 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/cmd_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,

};
```

u-boot 于内核的关系

U-Boot 作为 Bootloader，具备多种引导内核启动的方式。常用的 go 和 bootm 命令可以直接引导内核映像启动。U-Boot 与内核的关系主要是内核启动过程中参数的传递。

1．go 命令的实现

```
/* common/cmd_boot.c  */

int do_go (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])

{

    ulong addr, rc;

    int     rcode = 0;
```

```
	if (argc < 2) {
		printf ("Usage:\n%s\n", cmdtp->usage);
		return 1;
	}
	addr = simple_strtoul(argv[1], NULL, 16);
	printf ("## Starting application at 0x%08lX ...\n", addr);
	/*
	 * pass address parameter as argv[0] (aka command name),
	 * and all remaining args
	 */
	rc = ((ulong (*)(int, char *[]))addr) (--argc, &argv[1]);
	if (rc != 0) rcode = 1;

	printf ("## Application terminated, rc = 0x%lX\n", rc);
	return rcode;
}
```

go 命令调用 do_go()函数，跳转到某个地址执行的。如果在这个地址准备好了自引导的内核映像，就可以启动了。尽管 go 命令可以带变参，实际使用时一般不用来传递参数。

2. bootm 命令的实现

```
/* common/cmd_bootm.c */
int do_bootm (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
```

```
ulong iflag;

ulong addr;

ulong data, len, checksum;

ulong   *len_ptr;

uint   unc_len = 0x400000;

int    i, verify;

char   *name, *s;

int    (*appl)(int, char *[]);

image_header_t *hdr = &header;


s = getenv ("verify");

verify = (s && (*s == 'n')) ? 0 : 1;

if (argc < 2) {

      addr = load_addr;

} else {

      addr = simple_strtoul(argv[1], NULL, 16);

}

SHOW_BOOT_PROGRESS (1);

printf ("## Booting image at %08lx ...\n", addr);

/* Copy header so we can blank CRC field for re-calculation */

memmove (&header, (char *)addr, sizeof(image_header_t));

if (ntohl(hdr->ih_magic) != IH_MAGIC)

{

      puts ("Bad Magic Number\n");
```

```
        SHOW_BOOT_PROGRESS (-1);
        return 1;
}
SHOW_BOOT_PROGRESS (2);
data = (ulong)&header;
len  = sizeof(image_header_t);

checksum = ntohl(hdr->ih_hcrc);
hdr->ih_hcrc = 0;

if(crc32 (0, (char *)data, len) != checksum) {
        puts ("Bad Header Checksum\n");
        SHOW_BOOT_PROGRESS (-2);
        return 1;
}
SHOW_BOOT_PROGRESS (3);
/* for multi-file images we need the data part, too */
print_image_hdr ((image_header_t *)addr);
data = addr + sizeof(image_header_t);
len  = ntohl(hdr->ih_size);
if(verify) {
        puts ("    Verifying Checksum ... ");
        if(crc32 (0, (char *)data, len) != ntohl(hdr->ih_dcrc)) {
                printf ("Bad Data CRC\n");
```

```
                SHOW_BOOT_PROGRESS (-3);
                return 1;
            }
            puts ("OK\n");
        }
        SHOW_BOOT_PROGRESS (4);
        len_ptr = (ulong *)data;
……
        switch (hdr->ih_os) {
        default:                /* handled by (original) Linux case */
        case IH_OS_LINUX:
            do_bootm_linux  (cmdtp, flag, argc, argv,
                    addr, len_ptr, verify);
            break;
        ……
}
```

bootm 命令调用 do_bootm 函数。这个函数专门用来引导各种操作系统映像，可以支持引导 Linux、vxWorks、QNX 等操作系统。引导 Linux 的时候，调用 do_bootm_linux()函数。

3. do_bootm_linux 函数的实现

```
/* lib_arm/armlinux.c */
void do_bootm_linux (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[],
                ulong addr, ulong *len_ptr, int verify)
```

```
{
	DECLARE_GLOBAL_DATA_PTR;
	ulong len = 0, checksum;
	ulong initrd_start, initrd_end;
	ulong data;
	void (*theKernel)(int zero, int arch, uint params);
	image_header_t *hdr = &header;
	bd_t *bd = gd->bd;
#ifdef CONFIG_CMDLINE_TAG
	char *commandline = getenv ("bootargs");
#endif
	theKernel = (void (*)(int, int, uint))ntohl(hdr->ih_ep);
	/* Check if there is an initrd image */
	if(argc >= 3) {
		SHOW_BOOT_PROGRESS (9);
		addr = simple_strtoul (argv[2], NULL, 16);
		printf ("## Loading Ramdisk Image at %08lx ...\n", addr);
		/* Copy header so we can blank CRC field for re-calculation */
		memcpy (&header, (char *) addr, sizeof (image_header_t));
		if (ntohl (hdr->ih_magic) != IH_MAGIC) {
			printf ("Bad Magic Number\n");
			SHOW_BOOT_PROGRESS (-10);
			do_reset (cmdtp, flag, argc, argv);
		}
```

```
data = (ulong) & header;
len = sizeof (image_header_t);
checksum = ntohl (hdr->ih_hcrc);
hdr->ih_hcrc = 0;
if(crc32 (0, (char *) data, len) != checksum) {
	printf ("Bad Header Checksum\n");
	SHOW_BOOT_PROGRESS (-11);
	do_reset (cmdtp, flag, argc, argv);
}
SHOW_BOOT_PROGRESS (10);
print_image_hdr (hdr);
data = addr + sizeof (image_header_t);
len = ntohl (hdr->ih_size);
if(verify) {
	ulong csum = 0;
	printf ("    Verifying Checksum ... ");
	csum = crc32 (0, (char *) data, len);
	if (csum != ntohl (hdr->ih_dcrc)) {
		printf ("Bad Data CRC\n");
		SHOW_BOOT_PROGRESS (-12);
		do_reset (cmdtp, flag, argc, argv);
	}
	printf ("OK\n");
}
```

```
        SHOW_BOOT_PROGRESS (11);
        if ((hdr->ih_os != IH_OS_LINUX) ||
            (hdr->ih_arch != IH_CPU_ARM) ||
            (hdr->ih_type != IH_TYPE_RAMDISK)) {
            printf ("No Linux ARM Ramdisk Image\n");
            SHOW_BOOT_PROGRESS (-13);
            do_reset (cmdtp, flag, argc, argv);
        }
        /* Now check if we have a multifile image */
    } else if ((hdr->ih_type == IH_TYPE_MULTI) && (len_ptr[1])) {
        ulong tail = ntohl (len_ptr[0]) % 4;
        int i;
        SHOW_BOOT_PROGRESS (13);
        /* skip kernel length and terminator */
        data = (ulong) (&len_ptr[2]);
        /* skip any additional image length fields */
        for (i = 1; len_ptr[i]; ++i)
            data += 4;
        /* add kernel length, and align */
        data += ntohl (len_ptr[0]);
        if (tail) {
            data += 4 - tail;
        }
        len = ntohl (len_ptr[1]);
```

```
        } else {
                /* no initrd image */
                SHOW_BOOT_PROGRESS (14);
                len = data = 0;
        }
        if (data) {
                initrd_start = data;
                initrd_end = initrd_start + len;
        } else {
                initrd_start = 0;
                initrd_end = 0;
        }
        SHOW_BOOT_PROGRESS (15);
        debug ("## Transferring control to Linux (at address %08lx) ...\n",
                (ulong) theKernel);
#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);
#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);
#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
	setup_end_tag (bd);
#endif
	/* we assume that the kernel is in place */
	printf ("\nStarting kernel ...\n\n");
	cleanup_before_linux ();

	theKernel (0, bd->bi_arch_number, bd->bi_boot_params);
}
```

do_bootm_linux()函数是专门引导 Linux 映像的函数，它还可以处理 ramdisk 文件系统的映像。这里引导的内核映像和 ramdisk 映像，必须是 U-Boot 格式的。U-Boot 格式的映像可以通过 mkimage 工具来转换，其中包含了 U-Boot 可以识别的符号。