一、Bootloader 的启动

Linux 系统是通过 Bootloader 引导启动的。一上电，就要执行 Bootloader 来初始化系统。可以通过第 4 章的 Linux 启动过程框图回顾一下。

系统加电或复位后，所有 CPU 都会从某个地址开始执行，这是由处理器设计决定的。比如，X86 的复位向量在高地址端，ARM 处理器在复位时从地址 0x00000000 取第一条指令。嵌入式系统的开发板都要把板上 ROM 或 Flash 映射到这个地址。因此，必须把 Bootloader 程序存储在相应的 Flash 位置。系统加电后，CPU 将首先执行它。

主机和目标机之间一般有串口可以连接，Bootloader 软件通常会通过串口来输入输出。例如：输出出错或者执行结果信息到串口终端，从串口终端读取用户控制命令等

Bootloader 启动过程通常是多阶段的，这样既能提供复杂的功能，又有很好的可移植性。例如：从 Flash 启动的 Bootloader 多数是两阶段的启动过程。从后面 U-Boot 的内容可以详细分析这个特性。

大多数 Bootloader 都包含 2 种不同的操作模式：本地加载模式和远程下载模式。这 2 种操作模式的区别仅对于开发人员才有意义，也就是不同启动方式的使用。从最终用户的角度看，Bootloader 的作用就是用来加载操作系统，而并不存在所谓的本地加载模式与远程下载模式的区别。

因为 Bootloader 的主要功能是引导操作系统启动，所以我们详细讨论一下各种启动方式的特点。

1．网络启动方式

这种方式开发板不需要配置较大的存储介质，跟无盘工作站有点类似。但是使用这种启动方式之前，需要把 Bootloader 安装到板上的 EPROM 或者 Flash 中。Bootloader 通过以太网接口远程下载 Linux 内核映像或者文件系统。第 4 章介绍的交叉开发环境就是以网络启动方式建立的。这种方式对于嵌入式系统开发来说非常重要。

使用这种方式也有前提条件，就是目标板有串口、以太网接口或者其他连接方式。串口一般可以作为控制台，同时可以用来下载内核影像和 RAMDISK 文件系统。串口通信传输速率过低，不适合用来挂接 NFS 文件系统。所以以太网接口成为通用的互连设备，一般的开发板都可以配置 10M 以太网接口。

对于 PDA 等手持设备来说，以太网的 RJ-45 接口显得大了些，而 USB 接口，特别是 USB 的迷你接口，尺寸非常小。对于开发的嵌入式系统，可以把 USB 接口虚拟成以太网接口来通讯。这种方式在开发主机和开发板两端都需要驱动程序。

另外，还要在服务器上配置启动相关网络服务。Bootloader 下载文件一般都使用 TFTP 网络协议，还可以通过 DHCP 的方式动态配置 IP 地址。

DHCP/BOOTP 服务为 Bootloader 分配 IP 地址，配置网络参数，然后才能够支持网络传输功能。如果 Bootloader 可以直接设置网络参数，就可以不使用 DHCP。

TFTP 服务为 Bootloader 客户端提供文件下载功能，把内核映像和其他文件放在/tftpboot 目录下。这样 Bootloader 可以通过简单的 TFTP 协议远程下载内核映像到内存。如图 6.1 所示。

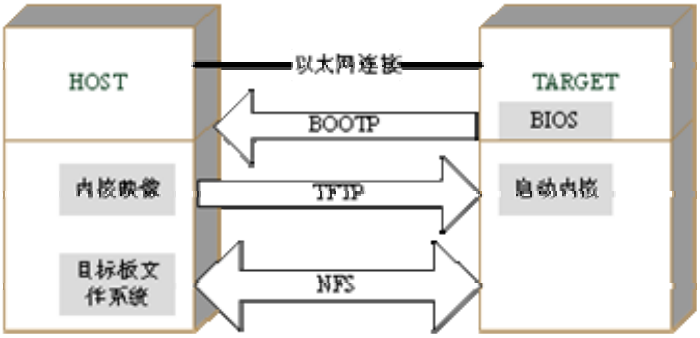


图 1.1　网络启动示意图

大部分引导程序都能够支持网络启动方式。例如：BIOS 的 PXE（Preboot Execution Environment）功能就是网络启动方式；U-Boot 也支持网络启动功能。

2．磁盘启动方式

传统的 Linux 系统运行在台式机或者服务器上，这些计算机一般都使用 BIOS 引导，并且使用磁盘作为存储介质。如果进入 BIOS 设置菜单，可以探测处理器、内存、硬盘等设备，可以设置 BIOS 从软盘、光盘或者某块硬盘启动。也就是说，BIOS 并不直接引导操作系统。那么在硬盘的主引导区，还需要一个 Bootloader。这个 Bootloader 可以从磁盘文件系统中把操作系统引导起来。

Linux 传统上是通过 LILO（LInux LOader）引导的，后来又出现了 GNU 的软件 GRUB（GRand Unified Bootloader）。这 2 种 Bootloader 广泛应用在 X86 的 Linux 系统上。你的开发主机可能就使用了其中一种，熟悉它们有助于配置多种系统引导功能。

LILO软件工程是由Werner Almesberger创建，专门为引导Linux开发的。现在LILO的维护者是John Coffman，最新版本下载站点：http://lilo.go.dyndns.org。LILO有详细的文档，例如LILO套件中附带使用手册和参考手册。此外，还可以在LDP的“LILO mini-HOWTO”中找到LILO的使用指南。

GRUB 是 GNU 计划的主要 bootloader。GRUB 最初是由 Erich Boleyn 为 GNU Mach 操作系统撰写的引导程序。后来有 Gordon Matzigkeit 和 Okuji Yoshinori 接替 Erich 的工作，继续维护和开发 GRUB。GRUB 的网站 http://www.gnu.org/software/grub/上有对套件使用的说明文件，叫作《GRUB manual》。GRUB 能够使用 TFTP 和 BOOTP 或者 DHCP 通过网络启动，这种功能对于系统开发过程很有用。

除了传统的 Linux 系统上的引导程序以外，还有其他一些引导程序，也可以支持磁盘引导启动。例如：LoadLin 可以从 DOS 下启动 Linux；还有 ROLO、LinuxBIOS，U-Boot 也支持这种功能。

3．Flash 启动方式

大多数嵌入式系统上都使用 Flash 存储介质。Flash 有很多类型，包括 NOR Flash、NAND Flash 和其他半导体盘。其中，NOR Flash（也就是线性 Flash）使用最为普遍。

NOR Flash 可以支持随机访问，所以代码是可以直接在 Flash 上执行的。Bootloader 一般是存储在 Flash 芯片上的。另外，Linux 内核映像和 RAMDISK 也可以存储在 Flash 上。通常需要把 Flash 分区使用，每个区的大小应该是 Flash 擦除块大小的整数倍。图 6.2 是 Bootloader 和内核映像以及文件系统的分区表。

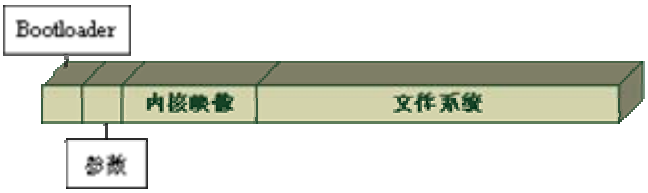


图 1.2　Flash 存储示意图

Bootloader 一般放在 Flash 的底端或者顶端，这要根据处理器的复位向量设置。要使 Bootloader 的入口位于处理器上电执行第一条指令的位置。

接下来分配参数区，这里可以作为 Bootloader 的参数保存区域。

再下来内核映像区。Bootloader 引导 Linux 内核，就是要从这个地方把内核映像解压到 RAM 中去，然后跳转到内核映像入口执行。

然后是文件系统区。如果使用 Ramdisk 文件系统，则需要 Bootloader 把它解压到 RAM 中。如果使用 JFFS2 文件系统，将直接挂接为根文件系统。这两种文件系统将在第 12 章详细讲解。

最后还可以分出一些数据区，这要根据实际需要和 Flash 大小来考虑了。

这些分区是开发者定义的，Bootloader 一般直接读写对应的偏移地址。到了 Linux 内核空间，可以配置成 MTD 设备来访问 Flash 分区。但是，有的 Bootloader 也支持分区的功能，例如：Redboot 可以创建 Flash 分区表，并且内核 MTD 驱动可以解析出 redboot 的分区表。

除了 NOR Flash，还有 NAND Flash、Compact Flash、DiskOnChip 等。这些 Flash 具有芯片价格低，存储容量大的特点。但是这些芯片一般通过专用控制器的 I/O 方式来访问，不能随机访问，因此引导方式跟 NOR Flash 也不同。在这些芯片上，需要配置专用的引导程序。通常，这种引导程序起始的一段代码就把整个引导程序复制到 RAM 中运行，从而实现自举启动，这跟从磁盘上启动有些相似。

二、U-BOOT 的目录结构

u-boot 目录下有 18 个子目录，分别存放管理不同的源程序。这些目录中所要存放的文件有其规则，可以分成三类。

- 第一类目录与处理器体系结构或者开发板硬件直接相关；
- 第二类目录是一些通用的函数或者驱动程序；
- 第三类目录是 u-boot 的应用程序、工具或者文档。

Board：和一些已有开发板相关的文件，比如 Makefile 和 u-boot.lds 等都和具体开发板的硬件和地址分配有关。

Common：与体系结构无关的文件，实现各种命令的 C 文件。

CPU：CPU 相关文件，其中的子目录都是以 u-boot 所支持的 CPU 为名，比如有子目录 arm926ejs、mips、mpc8260 和 nios 等，每个特定的子目 录中都包括 cpu.c 和 interrupt.c 和 start.S。其中 cpu.c 初始化 cpu、设置指令 cache 和数据 cache 等；interrupt.c 设置系统的各种终端和异常，比如快速中断，开关中断、时钟中断、软件中断、预取中止和未定义指令等；start.S 是 u- boot 启动时执行的第一个文件，他主要是设置系统堆栈和工作发式，为进入 C 程序奠定基础。

Disk：disk 驱动的分区处理代码、

Doc：文档。

Drivers：通用设备驱动程序，比如各种网卡、支持 CFI 的 flash、串口和 USB 总线等。

Dtt：数字温度测量器或者传感器的驱动

Examples：一些独立运行的应用程序的例子。

Fs：支持文件系统的文件，u-boot 现在支持 cramfs、fat、fdos、jffs2、yaffs 和 registerfs。

Include：头文件，还有对各种硬件平台支持的会变文件，系统的配置文件和对文件系统支持的文件。

Net：与网络有关的代码，BOOTP 协议、TFTP 协议 RARP 协议和 NFS 文件系统的实现。

Lib_ppc：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，与 PowerPC 体系结构相关的代码。

Lib_i386：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，与 PowerPc 体系结

构相关的代码。

Lib_arm：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，与 ARM 体系结构相关的代码。

Lib_generic：通用的多功能函数实现。

Post：上电自检。

Rtc： 实时时钟驱动。

Tools：创建 S-Record 格式文件和 U-BOOT images 的工具。

三、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 首先可以设置板子的定义，然后递归地调用各级目录下的 Makefile，最后把编译过的程序链接成 u-boot 的映像。

顶层目录下的 Makefile，它是负责 U-Boot 整体配置编译。每一种开发板在 Makefile 都需要有板子配置的定义，如 smdk2442 定义如下：

```
smdk2442_config: unconfig
@./mkconfig $(@:_config=) arm arm920t smdk2442
```

执行配置 U-Boot 的命令 make smdk2442_config，通过./mkconfig 脚本生成 include/config.mk 的配置文件。文件内容是根据 Makefile 对板子的配置生成的。

配置环境和编译过程如下所述,U-boot 的编译环境配置需要：cross-2.95.3.tar.bz2 和 s3c24x0_uboot_rel_0_0_1_061002.tar.bz2，将文件拷贝到/home/amoi/working/下，（chenpx@chenpx:/mnt/hgfs/share$ cp cross-2.95.3.tar.bz2 /home/amoi/working 和chenpx@chenpx:/mnt/hgfs/share$ cp s3c-u-boot-1.1.6.tar.bz2 /home/amoi/working），然后对对文件进行解压（chenpx@chenpx:/home/chenpx/working$ tar jxvf cross-2.95.3.tar.bz2 和 chenpx@chenpx:/home/chenpx/working$ tar jxvf s3c24x0_uboot_rel_0_0_1_061002.tar.bz2 ），在/usr/local/目录下建立一个arm文件夹（mkdir -p /usr/local/arm （-p 是需要时创建上层目录，如目录早已存在则不当作错误））将cross-2.95.3.tar.bz2 解压出来的移动到/usr/local/arm/下 （mv 2.95.3 /usr/local/arm/）移动后添加环境变量export PATH=$PATH:/usr/local/2.95.3/bin/

修改 s3c24x0_uboot_dev 中的 makefile，修改 CROSS_COMPILE = /usr/local/arm/2.95.3/bin/arm-linux-其他的用#注释掉。

接下来就是加载配置：

最后进行编译：make，最终在 s3c24x0_uboot-dev 目录下生成 u-boot、u-boot.bin、u-boot.map、u-boot.srec 四个文件。

四、u-boot 系统启动流程

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。

1、Stage1 start.S 代码结构

u-boot 的 stage1 代码通常放在 start.S 文件中，他用汇编语言写成，其主要代码部分如下：

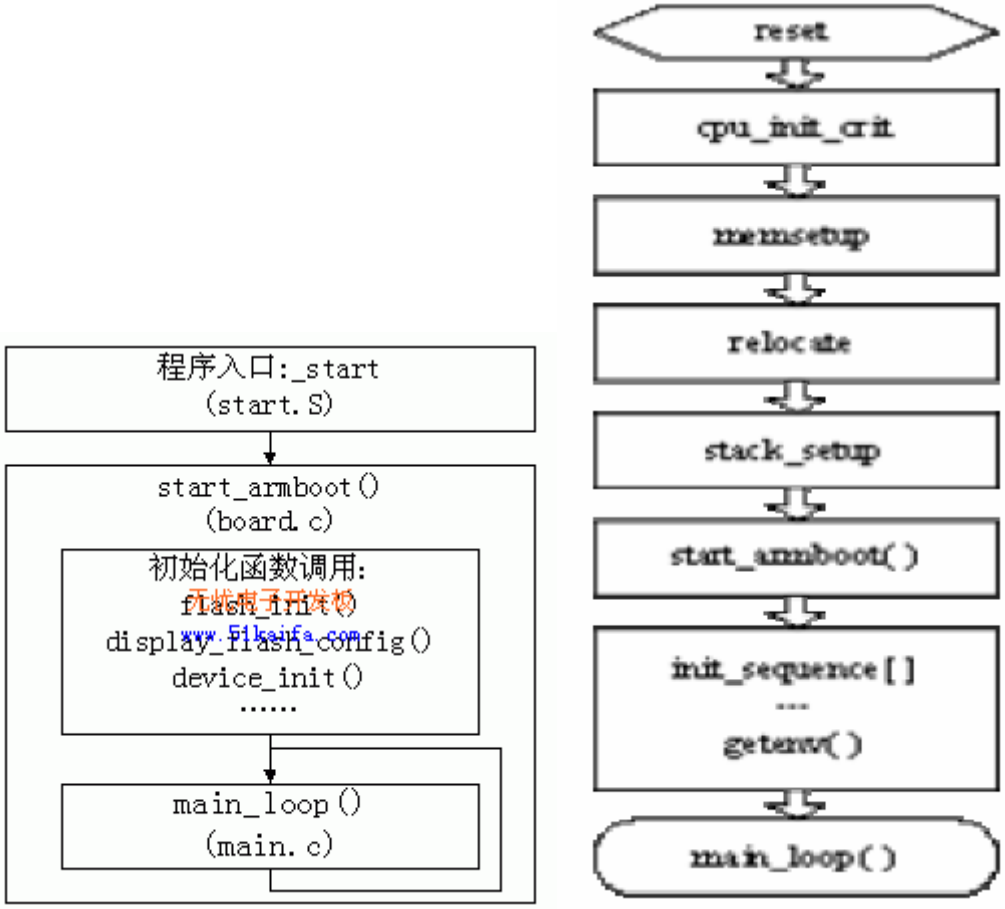
（1）定义入口。由于一个可执行的 Image 必须有一个入口点，并且只能有一个全局入口，通常这个入口放在 ROM（Flash）的 0x0 地址，因此，必须通知编译器以使其知道这个入口，该工作可通过修改连接器脚本来完成。
（2）设置异常向量（Exception Vector）。
（3）设置 CPU 的速度、时钟频率及终端控制寄存器。
（4）初始化内存控制器。
（5）将 ROM 中的程序复制到 RAM 中。
（6）初始化堆栈。
（7）转到 RAM 中执行，该工作可使用指令 ldr pc 来完成。

2、Stage2　C 语言代码部分

lib_arm/board.c 中的 start arm boot 是 C 语言开始的函数也是整个启动代码中 C 语言的主函数，同时还是整个 u-boot（armboot）的主函数，该函数只要完成如下操作：
（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。

3、U-Boot 的启动顺序

主要顺序如下图所示



图为 U-Boot 顺序

下面就根据 Start.S 代码进行解释：

```
/************************ 中断向量 ************************/
.globl _start                          //u-boot 启动入口
_start: b       reset                 //复位向量并且跳转到 reset
    ldr    pc, _undefined_instruction
    ldr    pc, _software_interrupt
    ldr    pc, _prefetch_abort
    ldr    pc, _data_abort
    ldr    pc, _not_used
    ldr    pc, _irq                       //中断向量
    ldr    pc, _fiq                       //中断向量
    b      sleep_setting                      //跳转到 sleep_setting
并通过下段代码拷贝到内存里
relocate:                              //把 uboot 重新定位到 RAM
    adr    r0, _start                     // r0 是代码的当前位置
    ldr    r2, _armboot_start               //r2 是 armboot 的开始地址
    ldr    r3, _armboot_end                 //r3 是 armboot 的结束地址
    sub    r2, r3, r2                       // r2 得到 armboot 的大小
    ldr    r1, _TEXT_BASE                     // r1 得到目标地址
    add    r2, r0, r2                       // r2 得到源结束地址
copy_loop:                             //重新定位代码
    ldmia  r0!, {r3-r10}                  //将 r0 单元中的数据读出到 r3-r10，r0 自动减 1
    stmia  r1!, {r3-r10}                  //将 r3-r10 的数据保存到 r1 指向的地址，r1 自动加 1
    cmp    r0, r2                          //复制数据块直到源数据末尾地址[r2]
    ble    copy_loop

系统上电或 reset 后，cpu 的 PC 一般都指向 0x0 地址，在 0x0 地址上的指令是
reset:                                    //复位启动子程序
    /******** 设置 CPU 为 SVC32 模式***********/
    mrs    r0, cpsr                        //将 CPSR 状态寄存器读取，保存到 R0 中
    bic    r0, r0, #0x1f                   //清楚 r0 的 0-4 位，结果写入 r0 中
    orr    r0, r0, #0xd3                   //将 r0 的值与立即数 0xd3 或并写入 r0 中
    msr    cpsr, r0                        //填充 cpsr 寄存器，将 R0 写入状态寄存器中

/*************** 关闭看门狗 ******************/
    ldr     r0, =pWTCON                    //
    mov     r1, #0x0
    str     r1, [r0]                  //将 r1 存储到 r0 包含的有效地址,将看门狗寄存器值清零

/*************** 关闭所有中断 *****************/
    mov     r1, #0xffffffff
    ldr     r0, =INTMSK
    str     r1, [r0]                       //
    ldr     r2, =0x7ff
    ldr     r0, =INTSUBMSK
    str     r2, [r0]                       //清 INTMSK、INTSUBMSK 某些位，关闭所有中断
```

```
/************** 初始化系统时钟 *****************/
    ldr     r0, =LOCKTIME
    ldr     r1, =0xffffff
    str     r1, [r0]                    //关闭定时器
clear_bss:
    ldr      r0, _bss_start             //找到 bss 的起始地址
    add      r0, r0, #4                 //从 bss 的第一个字开始
    ldr      r1, _bss_end               // bss 末尾地址
    mov      r2, #0x00000000            //清零
clbss_l:
    str      r2, [r0]                   // bss 段空间地址清零循环
    add      r0, r0, #4                 //从 bss 的第一个字开始
    cmp      r0, r1                     //循环比较 r0 是否等 r1
    bne      clbss_l

/***************** 关键的初始化子程序 ***********************/
/ * cpu 初始化关键寄存器
 * 设置重要寄存器
 * 设置内存时钟
* /
cpu_init_crit:
 /** flush v4 I/D caches*/
    mov     r0, #0
    mcr     p15, 0, r0, c7, c7, 0 /* flush v3/v4 cache 将 r0 中数据传到协处理器 p15 的寄存器 c7*/
    mcr     p15, 0, r0, c8, c7, 0 /* flush v4 TLB */
/************** disable MMU stuff and caches ****************/
    mrc     p15, 0, r0, c1, c0, 0           //将协处理器 p15 寄存器中的数据传送到寄存器 r0 中.
    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
/******* 在重新定位前，我们要设置 RAM 的时间，因为内存时钟依赖开发板硬件的，你将会找到 board 目录底下的 memsetup.S。*******/
    mov     ip, lr
#ifndef CONFIG_S3C2440A_JTAG_BOOT
    bl      memsetup                    //调用 memsetup 子程序（在 board/smdk2442memsetup.S）
#endif
    mov     lr, ip
    mov     pc, lr                        //子程序返回

memsetup:
/*************** 初始化内存 **************/
     mov     r1, #MEM_CTL_BASE
     adrl    r2, mem_cfg_val
     add     r3, r1, #52
1:   ldr     r4, [r2], #4
```

```
    str     r4, [r1], #4
    cmp     r1, r3
    bne     1b
/*********** 跳转到原来进来的下一个指令（start.S 文件里） **************/
    mov     pc, lr                      //子程序返回
/****************** 建立堆栈 ******************/
    ldr     r0, _armboot_end            //armboot_end 重定位
    add     r0, r0, #CONFIG_STACKSIZE   //向下配置堆栈空间
    sub     sp, r0, #12                 //为 abort-stack 预留个 3 字
/*************** 跳转到 C 代码去 *************/
    ldr     pc, _start_armboot          /*跳转到 start_armboot 函数入口，start_armboot
字保存函数入口指针*/
_start_armboot: .word start_armboot    //start_armboot 函数在 lib_arm/board.c 中实现
从此进入第二阶段 C 语言代码部分
/*************** 异常处理程序 ******************/
.align  5
undefined_instruction:              //未定义指令
    get_bad_stack
    bad_save_user_regs
    bl  do_undefined_instruction
.align 5
software_interrupt:                  //软件中断
    get_bad_stack
    bad_save_user_regs
    bl  do_software_interrupt
.align 5
prefetch_abort:                     //预取异常中止
    get_bad_stack
    bad_save_user_regs
    bl  do_prefetch_abort
.align 5
data_abort:                         //数据异常中止
    get_bad_stack
    bad_save_user_regs
    bl  do_data_abort
.align 5
not_used:                           //未利用
    get_bad_stack
    bad_save_user_regs
    bl  do_not_used
.align 5
irq:                                  //中断请求
    get_irq_stack
    irq_save_user_regs
    bl  do_irq
    irq_restore_user_regs
.align 5
```

```
fiq:                                    //快速中断请求
    get_fiq_stack
    /* someone ought to write a more effiction fiq_save_user_regs */
    irq_save_user_regs
    bl  do_fiq
    irq_restore_user_regs
sleep_setting:                          //休眠设置
@ prepare the SDRAM self-refresh mode
    ldr     r0, =0x48000024 @ REFRESH Register
    ldr     r1, [r0]
    orr     r1, r1,#(1<<22) @ self-refresh bit set
@ prepare MISCCR[19:17]=111b to make SDRAM signals(SCLK0,SCLK1,SCKE) protected
    ldr     r2, =0x56000080 @ MISCCR Register
    ldr     r3, [r2]                // MISCCR Register 的有效值放入寄存器 r3 中
    orr     r3, r3, #((1<<17)|(1<<18)|(1<<19))   //设置寄存器 r3 的某些位

@ prepare the Power_Off mode bit in CLKCON Register
    ldr     r4, =0x4c00000c @ CLKCON Register
    ldr     r5, =(1<<3)             //设置寄存器 r5 的第 3 位为 1
    b       set_sdram_refresh       //调到 set_sdram_refresh 段处理
.align 5
set_sdram_refresh:
    str     r1, [r0]            @ SDRAM self-refresh enable 将 r1 的有效地址放入 REFRESH Register
@ wait until SDRAM into self-refresh
    mov     r1, #64
1:  subs    r1, r1, #1
    bne     1b                  //指定次数的循环操作，每次循环减 1，判断结果是否为 0，为 0 退出

@ set the MISCCR & CLKCON register for power off
    str     r3, [r2]
    str     r5, [r4]
    nop                 @ waiting for power off
    nop
    nop
    b   .
```

第二阶段进入 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。进入 start_armboot 函数里，先对硬件资源进行初始化如下：

```
    init_fnc_t *init_sequence[] = {
        cpu_init,               /*基本的处理器相关配置 -- cpu/arm920t/cpu.c*/
        board_init,             /* 基本的开发板相关配置—board/smdk2442/smdk2442.c*/
        interrupt_init,         /* 初始化例外处理---cpu/arm920t/ 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/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—borad/smdk2442/smdk2442.c */
        display_dram_config,    /*显示 RAM 的配置大小---lib_arm/board.c */
#if defined(CONFIG_VCMA9) || defined (CONFIG_CMC_PU2)
        checkboard,             /*检测开发板---*.c */
#endif
        NULL,
};
```

使用以下语句调用执行

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

start_armboot 的主要过程如下：

```
    void start_armboot (void){
        DECLARE_GLOBAL_DATA_PTR;
        ulong size;
        gd_t gd_data;
        bd_t bd_data;
        init_fnc_t **init_fnc_ptr;
        char *s;
#if defined(CONFIG_VFD)
        unsigned long addr;
#endif
        /* Pointer is writable since we allocated a register for it */
        gd = &gd_data;
        memset ((void *)gd, 0, sizeof (gd_t));
        gd->bd = &bd_data;
        memset (gd->bd, 0, sizeof (bd_t));
        monitor_flash_len = _armboot_end_data - _armboot_start;
        /*** 调用执行 init_sequence 数组按顺序执行初始化 ***/
        for (init_fnc_ptr = init_sequence; *init_fnc_ptr; ++init_fnc_ptr) {
            if ((*init_fnc_ptr)() != 0) {
                hang ();
            }
        }
#if 0
 /*************** 配置可用的 flash 单元 *************/
        size = flash_init ();               //初始化 flash
        display_flash_config (size);        //显示 flash 的大小
/******** _arm_boot 在 armboot.lds 链接脚本中定义 ********/
#endif
#ifdef CONFIG_VFD
#  ifndef PAGE_SIZE
```

```
#  define PAGE_SIZE 4096        /*定义页面大小*/
#  endif
 /********** 为 VFD 显示预留内存(整个页面)  **********/
/******** armboot_real_end 在 board-specific 链接脚本中定义********/
        addr = (_armboot_real_end + (PAGE_SIZE - 1)) & ~(PAGE_SIZE - 1);
        size = vfd_setmem (addr);
        gd->fb_base = addr;
 /******* 进入下一个界面 ********/
        addr += size;
        addr = (addr + (PAGE_SIZE - 1)) & ~(PAGE_SIZE - 1);
        mem_malloc_init (addr);
#else
/********  armboot_real_end 在 board-specific 链接脚本中定义 *******/
        mem_malloc_init (_armboot_real_end);
#endif    /* CONFIG_VFD */
#if (CONFIG_COMMANDS & CFG_CMD_NAND)
        puts ("NAND:");
        nand_init();  /* NAND 初始化 */
#endif
#ifdef CONFIG_HAS_DATAFLASH
        AT91F_DataflashInit();
        dataflash_print_info();
#endif
/********** 初始化环境 **********/
        env_relocate ();
/********** 配置环境变量，重新定位 **********/
#ifdef CONFIG_VFD
 /* must do this after the framebuffer is allocated */
        drv_vfd_init();
#endif
 /* 从环境中得到 IP 地址 */
        bd_data.bi_ip_addr = getenv_IPaddr ("ipaddr");
 /*以太网接口 MAC 地址*/
    {
        int i;
        ulong reg;
        char *s, *e;
        uchar tmp[64];
        i = getenv_r ("ethaddr", tmp, sizeof (tmp));
        s = (i > 0) ? tmp : NULL;
        for (reg = 0; reg < 6; ++reg) {
            bd_data.bi_enetaddr[reg] = s ? simple_strtoul (s, &e, 16) : 0;
            if (s)
            s = (*e) ? e + 1 : e;
        }
    }
        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*/
#if defined(CONFIG_MISC_INIT_R)
 /* 其他平台由初始化决定*/
        misc_init_r ();
#endif
 /* 启用异常处理 */
        enable_interrupts ();
#ifdef CONFIG_DRIVER_CS8900
        cs8900_get_enetaddr (gd->bd->bi_enetaddr);
#endif
#ifdef CONFIG_DRIVER_LAN91C96
        if (getenv ("ethaddr")) {
            smc_set_mac_addr(gd->bd->bi_enetaddr);
        }
 /* eth_hw_init(); */
#endif /* CONFIG_DRIVER_LAN91C96 */
 /* 通过环境变量初始化*/
        if ((s = getenv ("loadaddr")) != NULL) {
            load_addr = simple_strtoul (s, NULL, 16);
        }
#if (CONFIG_COMMANDS & CFG_CMD_NET)
        if ((s = getenv ("bootfile")) != NULL) {
            copy_filename (BootFile, s, sizeof (BootFile));
        }
#endif /* CFG_CMD_NET */
#ifdef BOARD_POST_INIT
        board_post_init ();
#endif
 /* 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 */
}
```

lowlevel.S 代码分析

上面说到了 start.S 这个文件，现在说说 lowlevel.S 这个代码的作用以及代码的相关注释！

```
    #include <config.h>
    #include <version.h>
    #include <s3c2440.h>
    #include "smdk2440_val.h"

_TEXT_BASE:
    .word TEXT_BASE
    .globl lowlevel_init            /* lowlevel_init 的入口 */
```

```
lowlevel_init:
    mov     r12, lr                 /* lr 内存时钟值，依赖于硬件 */
    /* init system clock */
    bl      system_clock_init        /* 跳转到 system_clock_init */
    /* for UART */
    bl      uart_asm_init            /* 跳转到 uart_asm_init */
    /* simple init for NAND */
    bl      nand_asm_init            /* 跳转到 nand_asm_init */
    /* when we already run in ram, we don't need to relocate U-Boot.
     * and actually, memory controller must be configured before U-Boot
     * is running in ram.*/
    ldr     r0, =0xf0000fff         /* r0=0xf0000fff                    */
    bic     r1, pc, r0              /* r0 <- current base addr of code */
    ldr     r2, _TEXT_BASE          /* r1 <- original base addr in ram */
    bic     r2, r2, r0              /* r0 <- current base addr of code */
    cmp     r1, r2                  /* compare r0, r1                   */
    beq     1f                      /* r0 == r1 then skip sdram init   */
    adrl    r0, mem_cfg_val         /* 取得 mem_cfg_val 子程序的入口 */
    bl      mem_con_init    /* 跳转到mem_con_init,文件在\cpu\s3c24xx\s3c2440\cpu_init.S下 */
    ldr     r0, =ELFIN_UART_BASE     /* 加载 ELFIN_UART_BASE 地址到 r0 ，ELFIN_UART_BASE 是 UART
                                        的初始地址 */
    ldr     r1, =0x4b4b4b4b          /* 加载立即数 0x4b4b4b4b 到 r1     */
    str     r1, [r0, #0x20]          /* 地址（ELFIN_UART_BASE+0x20）=0x4b4b4b4b，用来设置串口
传输缓冲寄存器，输出 K 字母 */
1:  mov     lr, r12
    mov     pc, lr                      /* 子程序返回 */
/*
 * system_clock_init: Initialize core clock and bus clock.void system_clock_init(void)
 */
system_clock_init:
    /* Disable Watchdog */
    ldr     r0, =ELFIN_WATCHDOG_BASE      /* 加载 ELFIN_WATCHDOG_BASE 地址到 r0     */
    mov     r1, #0                        /* r1=0x0                              */
    str     r1, [r0]                      /* ELFIN_WATCHDOG_BASE=0，关闭看门狗  */

    /* mask all IRQs by setting all bits in the INTMR - default */
    ldr     r0,  =ELFIN_INTERRUPT_BASE        /*  加 载  ELFIN_INTERRUPT_BASE  地 址 到  r0 ，
ELFIN_INTERRUPT_BASE 是 SRCPND 寄存器地址 */
    mov     r1, #0xffffffff                   /* r1=0xffffffff */
    str     r1, [r0, #INTMSK_OFFSET]          /* (ELFIN_INTERRUPT_BASE+INTMSK_OFFSET)
                                                 =0xffffffff 关闭所有中断*/

 /*****************************************************************************
  *NOTE:INTERRUPT，中断掩码寄存器，决定那个中断源被屏蔽，某位为 1 则屏蔽中断源，初始值为
0xFFFFFFFF，屏蔽所有中断
   *****************************************************************************/
    ldr     r1, =0x00001fff           /* 加载立即数 0x000007ff*/    /*modify by chenpx 7ff */
```

```
    str     r1, [r0, #INTSUBMSK_OFFSET]      /* (ELFIN_INTERRUPT_BASE+INTSUBMSK_OFFSET)
                                                =0x000001fff 屏蔽中断源*/

 /*****************************************************************************
  * NOTE:INTSUBMSK，中断子掩码寄存器，该寄存器只能屏蔽 13 个中断源，因此其仅低 11 位有效，初始
  值为 0x7FF
  *****************************************************************************/

 /* Initialize System Clock */
    ldr     r0, =ELFIN_CLOCK_POWER_BASE         /* 加载 ELFIN_CLOCK_POWER_BASE 地址到 r0 中 */
    ldr     r1, =0x00ffffff                     /* 加载立即数 0x00ffffff 到 r1 中 */
    str     r1, [r0, #LOCKTIME_OFFSET]          /* Set Clock Divider            */
#ifndef S3C2440_PLL_OFF
    /* FCLK:HCLK:PCLK */ /* FCLK 是帧同步时钟，HCLK 是行同步时钟，PCLK 是 LCD 的扫描时钟 */
    mov     r1, #0x00000000                 /* r1=0x00000000                */
    str     r1,[r0, #CAMDIVN_OFFSET]      /* CAMDIVN=0x00000000,初始化照相机时钟分配寄存器 */
    ldr     r1, =CLKDIVN_VAL                /* 加载 CLKDIVN_VAL 地址到 r1 中 */
    str     r1, [r0, #CLKDIVN_OFFSET]       /* CLKDIVN=CLKDIVN_VAL，设置时钟分配控制寄存器 */

    /*****************************************************************
     * NOTE:设置时钟寄存器，CLKDIVN 第 0 位为 PDIVN，为 0 则 PCLK=HCLK，      *
     * 为 1 则 PCLK=HCLK/2 ;第 1 位为 HDIVN，为 0 则 HCLK=FCLK,为 1 则 HCLK=FCLK/2 *
     *****************************************************************/
#endif
 /* if HDIVN is not ZERO, the CPU bus mode has to be changed from the fast
  * bus mode to the async bus mode.
  */
     mrc     p15, 0, r1, c1, c0, 0          @ read ctrl register
     orr     r1, r1, #0xc0000000            @ Asynchronous 置某位
     mcr     p15, 0, r1, c1, c0, 0          @ write ctrl register
#ifndef S3C2440_PLL_OFF
    /* UPLL setup */
     ldr    r1, =UPLL_VAL                   /* 读取 UPLL_VAL 地址的数据加载到 r1 中 */
     str    r1, [r0, #UPLLCON_OFFSET]       /* r1 的值保存到 UPLLCON 中， */
#endif
    nop
    nop
    nop
    nop
    nop     @ wait until upll has the effect
    nop
    nop
    nop
#ifndef S3C2440_PLL_OFF
    /* PLL setup */
    ldr     r1, =MPLL_VAL                   /* 读取 MPLL_VAL 地址的数据并加载到 r1 中 */
    str     r1, [r0, #MPLLCON_OFFSET]       /* 将 r1 的数据保存到 MPLLCON 中， */
```

```
#endif
 /* wait at least 200us to stablize all clock */
    mov     r2, #0x10000
1:  subs    r1, r1, #1
    bne     1b              /*指定次数的循环操作，每次循环减 1，判断结果是否为 0，为 0 退出*/
    mov     pc, lr                          /* 子程序返回 */
/*
 * uart_basic_init: Initialize UART in asm mode, 115200bps fixed.void uart_asm_init(void)
 */
uart_asm_init:
    /* set GPIO to enable UART */
    ldr     r0, =ELFIN_GPIO_BASE          /* ELFIN_GPIO_BASE 为 0x56000000 */
    ldr     r1, =0x0000aaaa               /* 加载立即数到 0x0000aaaa 到 r1 中 */
    str     r1, [r0, #GPHCON_OFFSET]      /* 将 r1 的数据保存到 GPHCON 中，设置 GPHCON 寄存器，
                设置 GPH0~7 分别为 nCTS0，nRTS0，TXD0,RXD0,TXD1,RXD1,TXD2,RXD2 其他为输入 */
    ldr     r0, =ELFIN_UART_BASE          /* ELFIN_UART_BASE 的地址为 0x56000000 */
    mov     r1, #0x0                      /* 加载立即数 0x0 到 r1 中      */
    str     r1, [r0, #0x8]                /* 初始化 UART FIFO 控制寄存器 UFCON0 */
    str     r1, [r0, #0xC]                /* 初始化 UART MODEM 控制寄存器的 UMCON0 */
    mov     r1, #0x3                      /* r1=0x3 */
    str     r1,[r0, #0x0]           /* 设置 UART 专用寄存器 UNCON0，并设置 Word Length 为 8bits */
    ldr     r1, =0x245                    /* 加载立即数 0x245 到 r1 中 */
    str     r1, [r0, #0x4]          /* 设置 UART 控制寄存器 UCON0，设置其发送和接收模式为中断
                                          请求或轮询模式并使它能接收错误中断的电平 */
#if (CONFIG_SYS_CLK_FREQ == 16934400)
  #if defined (CONFIG_SMDK2440)
    ldr     r1, =0x23
  #elif defined (CONFIG_SMDK2442)
    ldr     r1, =0x19
  #endif
#endif
    str     r1, [r0, #UBRDIV0_OFFSET]       /* 设置 UBRDIV0 波特率寄存器      */
    ldr     r1, =0x4f4f4f4f                 /* 加载立即数 0x4f4f4f4f 到 r1 中 */
    str     r1, [r0, #0x20]                 /* 将 0x4f4f4f4f 保存到 0x56000020 中,输出 0 字母 */
    mov     pc, lr                          /* 子程序返回 */

/*
 * Nand Interface Init for smdk2440
 */
nand_asm_init:
    ldr     r0, =ELFIN_NAND_BASE    /* 加载 ELFIN_NAND_BASE 地址到 r0 中,NAND flash 的起始地址 */
    ldr     r1, [r0, #NFCONF_OFFSET]        /* 加载 NFCONF 的地址到 r1 中             */
    orr     r1, r1, #0xf0                   /* 将 r1 的第四 4~8 位置 1                 */
    orr     r1, r1, #0xff00                 /* r1 高八位置 1                          */
    str     r1, [r0]                        /* 将地址 r1 的值保存到 r0 中             */
#if 0
    ldr     r1, [r0, #NFCONT_OFFSET]
```

```
    orr     r1, r1, #0x01
#else
    mov     r1, #0x03                       /* r1=0x03                                  */
#endif
    str     r1, [r0, #NFCONT_OFFSET]        /* 设置NFCONT的NAND flash不工作并禁用芯片选择 */
    mov     pc, lr                          /* 子程序返回 */
    .align 4
mem_cfg_val:
    .long   vBSWCON  /* 0x00 */
    .long   vBANKCON0 /* 0x04 */
    .long   vBANKCON1 /* 0x08 */
    .long   vBANKCON2 /* 0x0c */
    .long   vBANKCON3 /* 0x10 */
    .long   vBANKCON4 /* 0x14 */
    .long   vBANKCON5 /* 0x18 */
    .long   vBANKCON6 /* 0x1c */
    .long   vBANKCON7 /* 0x20 */
    .long   vREFRESH /* 0x24 */
    .long   vBANKSIZE /* 0x28 */
    .long   vMRSRB6  /* 0x2c */
    .long   vMRSRB7  /* 0x30 */
var_in_lowlevel_init:
    .ltorg
#ifdef CONFIG_ENABLE_MMU
/*
 * MMU Table for SMDK2440
 */
/* these macro should match with pci window macros */
#define UNCACHED_SDRAM_START 1
#define UNCACHED_SDRAM_SZ 1
 /* form a first-level section entry */
    .macro FL_SECTION_ENTRY base,ap,d,c,b
    .word (\base << 20) | (\ap << 10) | (\d << 5) | (1<<4) | (\c << 3) | (\b << 2) | (1<<1)
    .endm
    .section .mmudata, "a"
    .align 14
/* the following alignment creates the mmu table at address 0x4000. */
    .globl mmu_table
mmu_table:
    .set __base,0
 /* 1:1 mapping for debugging */
    .rept 0x600
    FL_SECTION_ENTRY __base,3,0,0,0
    .set __base,__base+1
    .endr
 /* access is not allowed. */
    .rept 0xC00 - 0x600
```

```
    .word 0x00000000
    .endr
 /* 128MB for SDRAM 0xC0000000 -> 0x30000000 */
    .set __base, 0x300
    .rept 0xC80 - 0xC00
    FL_SECTION_ENTRY __base,3,0,1,1
    .set __base,__base+1
    .endr
 /* access is not allowed. */
    .rept 0x1000 - 0xc80
    .word 0x00000000
    .endr
#endif
```

五、U-Boot 的调试

新移植的 U-Boot 不能正常工作，这时就需要调试了。调试 U-Boot 离不开工具，只有理解 U-Boot 启动过程，才能正确地调试 U-Boot 源码。

5.1 硬件调试器

硬件电路板制作完成以后，这时上面还没有任何程序，就叫作裸板。首要的工作是把程序或者固件加载到裸板上，这就要通过硬件工具来完成。习惯上，这种硬件工具叫作仿真器。

仿真器可以通过处理器的 JTAG 等接口控制板子，直接把程序下载到目标板内存，或者进行 Flash 编程。如果板上的 Flash 是可以拔插的，就可以通过专用的 Flash 烧写器来完成。在前面章节介绍过目标板跟主机之间的连接，其中 JTAG 等接口就是专门用来连接仿真器的。

仿真器还有一个重要的功能就是在线调试程序，这对于调试 Bootloader 和硬件测试程序很有用。从最简单的 JTAG 电缆，到 ICE 仿真器，再到可以调试 Linux 内核的仿真器。

复杂的仿真器可以支持与计算机间的以太网或者 USB 接口通信。

对于 U-Boot 的调试，可以采用 BDI2000。BDI2000 完全可以反汇编地跟踪 Flash 中的程序，也可以进行源码级的调试。

使用 BDI2000 调试 U-boot 的方法如下。

（1）配置 BDI2000 和目标板初始化程序，连接目标板。

（2）添加 U-Boot 的调试编译选项，重新编译。

U-Boot 的程序代码是位置相关的，调试的时候尽量在内存中调试，可以修改连接定位地址 TEXT_BASE。TEXT_BASE 在 board/<board_name>/config.mk 中定义。

另外，如果有复位向量也需要先从链接脚本中去掉。链接脚本是 board/<board_name>/u-boot.lds。

添加调试选项，在 config.mk 文件中查找，DBGFLAGS，加上-g 选项。然后重新编译 U-Boot。

（3）下载 U-Boot 到目标板内存。

通过 BDI2000 的下载命令 LOAD，把程序加载到目标板内存中。然后跳转到 U-Boot 入口。

（4）启动 GDB 调试。

启动 GDB 调试，这里是交叉调试的 GDB。GDB 与 BDI2000 建立链接，然后就可以设置断点执行了。

```
$ arm-linux-gdb u-boot
(gdb)target remote 192.168.1.100:2001
(gdb)stepi
(gdb)b start_armboot
(gdb)c
```

5.2 软件跟踪

假如 U-Boot 没有任何串口打印信息，手头又没有硬件调试工具，那样怎么知道 U-Boot 执行到什么地方了呢？可以通过开发板上的 LED 指示灯判断。

开发板上最好设计安装八段数码管等 LED，可以用来显示数字或者数字位。

U-Boot 可以定义函数 show_boot_progress (int status)，用来指示当前启动进度。在 include/common.h 头文件中声明这个函数。

```
#ifdef CONFIG_SHOW_BOOT_PROGRESS
    void    show_boot_progress (int status);
#endif
```

CONFIG_SHOW_BOOT_PROGRESS 是需要定义的。这个在板子配置的头文件中定义。CSB226 开发板对这项功能有完整实现，可以参考。在头文件 include/configs/csb226.h 中，有下列一行。

```
#define CONFIG_SHOW_BOOT_PROGRESS       1
```

函数 show_boot_progress (int status)的实现跟开发板关系密切，所以一般在 board 目录下的文件中实现。看一下 CSB226 在 board/csb226/csb226.c 中的实现函数。

```
/** 设置 CSB226 板的 0、1、2 三个指示灯的开关状态
 * csb226_set_led: - switch LEDs on or off
 * @param led:   LED to switch (0,1,2)
 * @param state: switch on (1) or off (0)
 */
void csb226_set_led(int led, int state){
      switch(led) {
             case 0: if (state==1) {
                            GPCR0 |= CSB226_USER_LED0;
                    } else if (state==0) {
                            GPSR0 |= CSB226_USER_LED0;
                    }
                    break;
             case 1: if (state==1) {
                            GPCR0 |= CSB226_USER_LED1;
                    } else if (state==0) {
                            GPSR0 |= CSB226_USER_LED1;
                    }
                    break;
             case 2: if (state==1) {
                            GPCR0 |= CSB226_USER_LED2;
                  } else if (state==0) {
```

```
                        GPSR0 |= CSB226_USER_LED2;
                }
                break;
        }
        return;
}
/** 显示启动进度函数，在比较重要的阶段，设置三个灯为亮的状态（1, 5, 15）*/
void show_boot_progress (int status){
        switch(status) {
                case  1: csb226_set_led(0,1); break;
                case  5: csb226_set_led(1,1); break;
                case 15: csb226_set_led(2,1); break;
        }
        return;
}
```

这样，在 U-Boot 启动过程中就可以通过 show_boot_progresss 指示执行进度。比如 hang()函数是系统出错时调用的函数，这里需要根据特定的开发板给定显示的参数值。

```
void hang (void){
        puts ("### ERROR ### Please RESET the board ###\n");
#ifdef CONFIG_SHOW_BOOT_PROGRESS
        show_boot_progress(-30);
#endif
        for (;;);
}
```

六、U-Boot 与内核的关系

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);         /*将字符转换为长整形，用 16 进制转换*/
       printf ("## Starting application at 0x%08lX ...\n", addr);
       /*
        * pass address parameter as argv[0] (aka command name),
        * and all remaining args 执行所有转换参数
```

```
     */
    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。

七、U-Boot 的常用命令

U-Boot 上电启动后，敲任意键可以退出自动启动状态，进入命令行。

```
U-Boot 1.1.2 (Apr 26 2005 - 12:27:13)
U-Boot code: 11080000 -> 1109614C  BSS: -> 1109A91C
RAM Configuration:
Bank #0: 10000000 32 MB
Micron StrataFlash MT28F128J3 device initialized
Flash: 32 MB
In:    serial
Out:   serial
Err:   serial
Hit any key to stop autoboot:  0
U-Boot>
```

在命令行提示符下，可以输入 U-Boot 的命令并执行。U-Boot 可以支持几十个常用命令，通过这些命令，可以对开发板进行调试，可以引导 Linux 内核，还可以擦写 Flash 完成系统部署等功能。掌握这些命令的使用，才能够顺利地进行嵌入式系统的开发。输入 help 命令，可以得到当前 U-Boot 的所有命令列表。每一条命令后面是简单的命令说明。

```
=> help
?       - alias for 'help'
    autoscr - run script from memory
```

```
    base    - print or set address offset
    bdinfo  - print Board Info structure
    boot    - boot default, i.e., run 'bootcmd'
    bootd   - boot default, i.e., run 'bootcmd'
    bootm   - boot application image from memory
    bootp   - boot image via network using BootP/TFTP protocol
    cmp     - memory compare
    coninfo  - print console devices and information
    cp      - memory copy
    crc32   - checksum calculation
    dhcp    - invoke DHCP client to obtain IP/boot params
    echo    - echo args to console
    erase   - erase FLASH memory
    flinfo  - print FLASH memory information
    go      - start application at address 'addr'
    help    - print online help
    iminfo  - print header information for application image
    imls    - list all images found in flash
    itest    - return true/false on integer compare
    loadb   - load binary file over serial line (kermit mode)
    loads   - load S-Record file over serial line
    loop   - infinite loop on address range
    md    - memory display
    mm    - memory modify (auto-incrementing)
    mtest   - simple RAM test
    mw      - memory write (fill)
    nfs     - boot image via network using NFS protocol
    nm      - memory modify (constant address)
    printenv - print environment variables
    protect - enable or disable FLASH write protection
    rarpboot - boot image via network using RARP/TFTP protocol
    reset   - Perform RESET of the CPU
    run     - run commands in an environment variable
    saveenv - save environment variables to persistent storage
    setenv  - set environment variables
    sleep   - delay execution for some time
    tftpboot - boot image via network using TFTP protocol
    version - print monitor version
=>
```

U-Boot 还提供了更加详细的命令帮助，通过 help 命令还可以查看每个命令的参数说明。由于开发过程的需要，有必要先把 U-Boot 命令的用法弄清楚。接下来，根据每一条命令的帮助信息，解释一下这些命令的功能和参数。

```
=> help bootm
bootm [addr [arg ...]]
    - boot application image stored in memory passing arguments 'arg ...'; when booting a Linux
```

```
kernel, 'arg' can be the address of an initrd image
```

bootm 命令可以引导启动存储在内存中的程序映像。这些内存包括 RAM 和可以永久保存的 Flash。

第 1 个参数 addr 是程序映像的地址，这个程序映像必须转换成 U-Boot 的格式。

第 2 个参数对于引导 Linux 内核有用，通常作为 U-Boot 格式的 RAMDISK 映像存储地址；也可以是传递给 Linux 内核的参数（缺省情况下传递 bootargs 环境变量给内核）。

```
=> help bootp
bootp [loadAddress] [bootfilename]
```

bootp 命令通过 bootp 请求，要求 DHCP 服务器分配 IP 地址，然后通过 TFTP 协议下载指定的文件到内存。

第 1 个参数是下载文件存放的内存地址。

第 2 个参数是要下载的文件名称，这个文件应该在开发主机上准备好。

```
=> help cmp
cmp [.b, .w, .l] addr1 addr2 count
     - compare memory
```

cmp 命令可以比较 2 块内存中的内容。.b 以字节为单位；.w 以字为单位；.l 以长字为单位。注意：cmp.b 中间不能保留空格，需要连续敲入命令。

第 1 个参数 addr1 是第一块内存的起始地址。

第 2 个参数 addr2 是第二块内存的起始地址。

第 3 个参数 count 是要比较的数目，单位按照字节、字或者长字。

```
=> help cp
cp [.b, .w, .l] source target count
       - copy memory
```

cp 命令可以在内存中复制数据块，包括对 Flash 的读写操作。

第 1 个参数 source 是要复制的数据块起始地址。

第 2 个参数 target 是数据块要复制到的地址。这个地址如果在 Flash 中，那么会直接调用写 Flash 的函数操作。所以 U-Boot 写 Flash 就使用这个命令，当然需要先把对应 Flash 区域擦干净。

第 3 个参数 count 是要复制的数目，根据 cp.b cp.w cp.l 分别以字节、字、长字为单位。

```
=> help crc32
crc32 address count [addr]
     - compute CRC32 checksum [save at addr]
```

crc32 命令可以计算存储数据的校验和。

第 1 个参数 address 是需要校验的数据起始地址。

第 2 个参数 count 是要校验的数据字节数。

第 3 个参数 addr 用来指定保存结果的地址。

```
=> help echo
echo [args..]
      - echo args to console; \c suppresses newline
```

echo 命令回显参数。

```
=> help erase
erase start end
      - erase FLASH from addr 'start' to addr 'end'
erase N:SF[-SL]
      - erase sectors SF-SL in FLASH bank # N
erase bank N
      - erase FLASH bank # N
erase all
      - erase all FLASH banks
```

erase 命令可以擦 Flash。

参数必须指定 Flash 擦除的范围。

按照起始地址和结束地址，start 必须是擦除块的起始地址；end 必须是擦除末尾块的结束地址。这种方式最常用。举例说明：擦除 0x20000 - 0x3ffff 区域命令为 erase 20000 3ffff。

按照组和扇区，N 表示 Flash 的组号，SF 表示擦除起始扇区号，SL 表示擦除结束扇区号。另外，还可以擦除整个组，擦除组号为 N 的整个 Flash 组。擦除全部 Flash 只要给出一个 all 的参数即可。

```
=> help flinfo
flinfo
       - print information for all FLASH memory banks
flinfo N
       - print information for FLASH memory bank # N
```

flinfo 命令打印全部 Flash 组的信息，也可以只打印其中某个组。一般嵌入式系统的 Flash 只有一个组。

```
=> help go
go addr [arg ...]
      - start application at address 'addr'
        passing 'arg' as arguments
```

go 命令可以执行应用程序。

第 1 个参数是要执行程序的入口地址。

第 2 个可选参数是传递给程序的参数，可以不用。

```
=> help iminfo
iminfo addr [addr ...]
      - print header information for application image starting at address 'addr' in memory; this
includes verification of the image contents (magic number, header and payload checksums)
```

iminfo 可以打印程序映像的开头信息，包含了映像内容的校验（序列号、头和校验和）。

第 1 个参数指定映像的起始地址。

可选的参数是指定更多的映像地址。

```
=> help loadb
loadb [ off ] [ baud ]
```

```
     - load binary file over serial line with offset 'off' and baudrate 'baud'
```

loadb 命令可以通过串口线下载二进制格式文件。

```
=> help loads
loads [ off ]
    - load S-Record file over serial line with offset 'off'
```

loads 命令可以通过串口线下载 S-Record 格式文件。

```
=> help mw
mw [.b, .w, .l] address value [count]
     - write memory
```

mw 命令可以按照字节、字、长字写内存，.b .w .l 的用法与 cp 命令相同。
    第 1 个参数 address 是要写的内存地址。
    第 2 个参数 value 是要写的值。
    第 3 个可选参数 count 是要写单位值的数目。

```
=> help nfs
nfs [loadAddress] [host ip addr:bootfilename]
```

nfs 命令可以使用 NFS 网络协议通过网络启动映像。

```
=> help nm
nm [.b, .w, .l] address
     - memory modify, read and keep address
```

nm 命令可以修改内存，可以按照字节、字、长字操作。
    参数 address 是要读出并且修改的内存地址。

```
=> help printenv
printenv
      - print values of all environment variables
printenv name ...
      - print value of environment variable 'name'
```

printenv 命令打印环境变量。
    可以打印全部环境变量，也可以只打印参数中列出的环境变量。

```
=> help protect
protect on  start end
      - protect Flash from addr 'start' to addr 'end'
protect on  N:SF[-SL]
      - protect sectors SF-SL in Flash bank # N
protect on  bank N
      - protect Flash bank # N
```

```
protect on  all
      - protect all Flash banks
protect off start end
      - make Flash from addr 'start' to addr 'end' writable
protect off N:SF[-SL]
     - make sectors SF-SL writable in Flash bank # N
protect off bank N
     - make Flash bank # N writable
protect off all
     - make all Flash banks writable
```

protect 命令是对 Flash 写保护的操作，可以使能和解除写保护。

第 1 个参数 on 代表使能写保护；off 代表解除写保护。

第 2、3 参数是指定 Flash 写保护操作范围，跟擦除的方式相同。

```
=> help rarpboot
rarpboot [loadAddress] [bootfilename]
```

rarboot 命令可以使用 TFTP 协议通过网络启动映像。也就是把指定的文件下载到指定地址，然后执行。

第 1 个参数是映像文件下载到的内存地址。

第 2 个参数是要下载执行的映像文件。

```
=> help run
run var [...]
      - run the commands in the environment variable(s) 'var'
```

run 命令可以执行环境变量中的命令，后面参数可以跟几个环境变量名。

```
=> help setenv
setenv name value ...
      - set environment variable 'name' to 'value ...'
setenv name
      - delete environment variable 'name'
```

setenv 命令可以设置环境变量。

第 1 个参数是环境变量的名称。

第 2 个参数是要设置的值，如果没有第 2 个参数，表示删除这个环境变量。

```
=> help sleep
sleep N
      - delay execution for N seconds (N is _decimal_ !!!)
```

sleep 命令可以延迟 N 秒钟执行，N 为十进制数。

```
=> help tftpboot
tftpboot [loadAddress] [bootfilename]
```

tftpboot 命令可以使用 TFTP 协议通过网络下载文件。按照二进制文件格式下载。另外使用这个命令，必须配置好相关的环境变量。例如 serverip 和 ipaddr。

第 1 个参数 loadAddress 是下载到的内存地址。

第 2 个参数是要下载的文件名称，必须放在 TFTP 服务器相应的目录下。

这些 U-Boot 命令为嵌入式系统提供了丰富的开发和调试功能。在 Linux 内核启动和调试过程中，都可以用到 U-Boot 的命令。但是一般情况下，不需要使用全部命令。比如已经支持以太网接口，可以通过 tftpboot 命令来下载文件，那么还有必要使用串口下载的 loadb 吗？反过来，如果开发板需要特殊的调试功能，也可以添加新的命令。

在建立交叉开发环境和调试 Linux 内核等章节时，在 ARM 平台上移植了 U-Boot，并且提供了具体 U-Boot 的操作步骤。

八、U-Boot 的环境变量

有点类似 Shell，U-Boot 也使用环境变量。可以通过 printenv 命令查看环境变量的设置。

```
U-Boot> printenv
    bootdelay=3
    baudrate=115200
    netmask=255.255.0.0
    ethaddr=12:34:56:78:90:ab
    bootfile=uImage
    bootargs=console=ttyS0,115200 root=/dev/ram rw initrd=0x30800000,8M
    bootcmd=tftp 0x30008000 zImage;go 0x30008000
    serverip=192.168.1.1
    ipaddr=192.168.1.100
    stdin=serial
    stdout=serial
    stderr=serial
    Environment size: 337/131068 bytes
U-Boot>
```

表 8.1 是常用环境变量的含义解释。通过 printenv 命令可以打印出这些变量的值。

表 8.1　　U-Boot 环境变量的解释说明

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

U-Boot 的环境变量都可以有缺省值，也可以修改并且保存在参数区。U-Boot 的参数区一般有 EEPROM 和 Flash 两种设备。

环境变量的设置命令为 setenv，在上面命令的解释。

举例说明环境变量的使用。

```
=>setenv serverip  192.168.1.1
=>setenv ipaddr  192.168.1.100
=>setenv rootpath  "/usr/local/arm/3.3.2/rootfs"
=>setenv bootargs  "root=/dev/nfs rw nfsroot=\$(serverip):\$(rootpath) ip=
\$(ipaddr) "
=>setenv kernel_addr 30000000
=>setenv nfscmd  "tftp \$(kernel_addr) uImage; bootm \$(kernel_addr) "
=>run nfscmd
```

上面定义的环境变量有 serverip ipaddr rootpath bootargs kernel_addr。环境变量 bootargs 中还使用了环境变量，bootargs 定义命令行参数，通过 bootm 命令传递给内核。环境变量 nfscmd 中也使用了环境变量，功能是把 uImage 下载到指定的地址并且引导起来。可以通过 run 命令执行 nfscmd 脚本。