1.1 U-Boot 工作过程

U-Boot 启动内核的过程可以分为两个阶段，两个阶段的功能如下：

（1）第一阶段的功能

- 硬件设备初始化
- 加载 U-Boot 第二阶段代码到 RAM 空间
- 设置好栈
- 跳转到第二阶段代码入口

（2）第二阶段的功能

- 初始化本阶段使用的硬件设备
- 检测系统内存映射
- 将内核从 Flash 读取到 RAM 中
- 为内核设置启动参数
- 调用内核

1.1.1 U-Boot 启动第一阶段代码分析

第一阶段对应的文件是 cpu/arm920t/start.S 和 board/samsung/mini2440/lowlevel_init.S。

U-Boot 启动第一阶段流程如下：

图 2.1 U-Boot 启动第一阶段流程

根据 cpu/arm920t/u-boot.lds 中指定的连接方式：

```
ENTRY(_start)
SECTIONS
{
    . = 0x00000000;

    . = ALIGN(4);
    .text :
    {
              cpu/arm920t/start.o    (.text)
          board/samsung/mini2440/lowlevel_init.o (.text)
           board/samsung/mini2440/nand_read.o (.text)
         *(.text)
```

```
    }
    … …
}
```

第一个链接的是 cpu/arm920t/start.o，因此 u-boot.bin 的入口代码在 cpu/arm920t/start.o 中，其源代码在 cpu/arm920t/start.S 中。下面我们来分析 cpu/arm920t/start.S 的执行。

1. 硬件设备初始化

（1）设置异常向量

cpu/arm920t/start.S 开头有如下的代码：

```
.globl _start
_start:    b     start_code                    /* 复位 */
     ldr   pc, _undefined_instruction      /* 未定义指令向量 */
     ldr   pc, _software_interrupt          /*  软件中断向量 */
     ldr   pc, _prefetch_abort              /*  预取指令异常向量 */
     ldr   pc, _data_abort                  /*  数据操作异常向量 */
     ldr   pc, _not_used                    /*  未使用  */
     ldr   pc, _irq                         /*  irq 中断向量  */
     ldr   pc, _fiq                         /*  fiq 中断向量  */
/*  中断向量表入口地址 */
_undefined_instruction:    .word undefined_instruction
_software_interrupt:  .word software_interrupt
_prefetch_abort:  .word prefetch_abort
_data_abort:       .word data_abort
_not_used:         .word not_used
_irq:               .word irq
_fiq:               .word fiq
```

```
.balignl 16,0xdeadbeef
```

以上代码设置了 ARM 异常向量表，各个异常向量介绍如下：

表 2.1 ARM 异常向量表

地址	异常	进入模式	描述
0x00000000	复位	管理模式	复位电平有效时，产生复位异常，程序跳转到复位处理程序处执行
0x00000004	未定义指令	未定义模式	遇到不能处理的指令时，产生未定义指令异常
0x00000008	软件中断	管理模式	执行 SWI 指令产生，用于用户模式下的程序调用特权操作指令
0x0000000c	预存指令	中止模式	处理器预取指令的地址不存在，或该地址不允许当前指令访问，产生指令预取中止异常
0x00000010	数据操作	中止模式	处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常
0x00000014	未使用	未使用	未使用
0x00000018	IRQ	IRQ	外部中断请求有效,且 CPSR 中的 I 位为 0 时，产生 IRQ 异常
0x0000001c	FIQ	FIQ	快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常

在 cpu/arm920t/start.S 中还有这些异常对应的异常处理程序。当一个异常产生时，CPU 根据异常号在异常向量表中找到对应的异常向量，然后执行异常向量处的跳转指令，CPU 就跳转到对应的异常处理程序执行。

其中复位异常向量的指令“b start_code”决定了 U-Boot 启动后将自动跳转到标号“start_code”处执行。

（2）**CPU 进入 SVC 模式**

```
start_code:
    /*
     * set the cpu to SVC32 mode
     */
    mrs r0, cpsr
    bic  r0, r0, #0x1f       /*工作模式位清零 */
    orr   r0, r0, #0xd3           /*工作模式位设置为“10011”（管理模式），并将中断禁止位和快中断禁止位置 1 */
    msr cpsr, r0
```

以上代码将 CPU 的工作模式位设置为管理模式，并将中断禁止位和快中断禁止位置一，从而屏蔽了 IRQ 和 FIQ 中断。

（3）设置控制寄存器地址

```
# if defined(CONFIG_S3C2400)
#  define pWTCON 0x15300000
#  define INTMSK  0x14400008
#  define CLKDIVN      0x14800014
#else      /* s3c2410 与 s3c2440 下面 4 个寄存器地址相同 */
#  define pWTCON 0x53000000            /* WATCHDOG 控制寄存器地址 */
#  define INTMSK  0x4A000008               /* INTMSK 寄存器地址  */
#  define INTSUBMSK 0x4A00001C     /* INTSUBMSK 寄存器地址 */
#  define CLKDIVN      0x4C000014              /* CLKDIVN 寄存器地址 */
# endif
```

对与 s3c2440 开发板，以上代码完成了 WATCHDOG，INTMSK，INTSUBMSK，CLKDIVN 四个寄存器的地址的设置。各个寄存器地址参见参考文献[4] 。

（4）关闭看门狗

```
    ldr   r0, =pWTCON
    mov     r1, #0x0
    str   r1, [r0]   /* 看门狗控制器的最低位为 0 时，看门狗不输出复位信号 */
```

以上代码向看门狗控制寄存器写入 0，关闭看门狗。否则在 U-Boot 启动过程中，CPU 将不断重启。

（5）屏蔽中断

```
    /*
     * mask all IRQs by setting all bits in the INTMR - default
     */
    mov     r1, #0xffffffff    /* 某位被置 1 则对应的中断被屏蔽 */
    ldr   r0, =INTMSK
    str   r1, [r0]
```

INTMSK 是主中断屏蔽寄存器，每一位对应 SRCPND（中断源引脚寄存器）中的一位，表明 SRCPND 相应位代表的中断请求是否被 CPU 所处理。

根据参考文献 4，INTMSK 寄存器是一个 32 位的寄存器，每位对应一个中断，向其中写入 0xffffffff 就将 INTMSK 寄存器全部位置一，从而屏蔽对应的中断。

```
# if defined(CONFIG_S3C2440)
      ldr  r1, =0x7fff
      ldr  r0, =INTSUBMSK
      str  r1, [r0]
# endif
```

INTSUBMSK 每一位对应 SUBSRCPND 中的一位，表明 SUBSRCPND 相应位代表的中断请求是否被 CPU 所处理。

根据参考文献 4，INTSUBMSK 寄存器是一个 32 位的寄存器，但是只使用了低 15 位。向其中写入 0x7fff 就是将 INTSUBMSK 寄存器全部有效位（低 15 位）置一，从而屏蔽对应的中断。

（6）设置 MPLLCON,UPLLCON, CLKDIVN

```
# if defined(CONFIG_S3C2440)
#define MPLLCON   0x4C000004
#define UPLLCON   0x4C000008
       ldr  r0, =CLKDIVN
       mov  r1, #5
       str  r1, [r0]

       ldr  r0, =MPLLCON
       ldr  r1, =0x7F021
       str  r1, [r0]

   ldr  r0, =UPLLCON
       ldr  r1, =0x38022
       str  r1, [r0]
# else
     /* FCLK:HCLK:PCLK = 1:2:4 */
     /* default FCLK is 120 MHz ! */
     ldr   r0, =CLKDIVN
     mov      r1, #3
     str   r1, [r0]
#endif
```

CPU 上电几毫秒后，晶振输出稳定，FCLK=Fin（晶振频率），CPU 开始执行指令。但实际上，FCLK 可以高于 Fin，为了提高系统时钟，需要用软件来启用 PLL。这就需要设置 CLKDIVN，MPLLCON，UPLLCON 这 3 个寄存器。

CLKDIVN 寄存器用于设置 FCLK，HCLK，PCLK 三者间的比例，可以根据表 2.2 来设置。

表 2.2 S3C2440 的 CLKDIVN 寄存器格式

CLKDIVN	位	说明	初始值
HDIVN	[2:1]	00 : HCLK = FCLK/1. 01 : HCLK = FCLK/2. 10 : HCLK = FCLK/4 （当 CAMDIVN[9] = 0 时） HCLK= FCLK/8 （当 CAMDIVN[9] = 1 时） 11 : HCLK = FCLK/3 （当 CAMDIVN[8] = 0 时） HCLK = FCLK/6 （当 CAMDIVN[8] = 1 时）	00
PDIVN	[0]	0: PCLK = HCLK/1 1: PCLK = HCLK/2	0

设置 CLKDIVN 为 5，就将 HDIVN 设置为二进制的 10，由于 CAMDIVN[9]没有被改变过，取默认值 0，因此 HCLK = FCLK/4。PDIVN 被设置为 1，因此 PCLK= HCLK/2。因此分频比 FCLK:HCLK:PCLK = 1:4:8 。

MPLLCON 寄存器用于设置 FCLK 与 Fin 的倍数。MPLLCON 的位[19:12]称为 MDIV，位[9:4]称为 PDIV，位[1:0]称为 SDIV。

对于 S3C2440，FCLK 与 Fin 的关系如下面公式：

MPLL(FCLK) = (2×m×Fin)/(p×)

其中： m=MDIC+8，p=PDIV+2，s=SDIV

MPLLCON 与 UPLLCON 的值可以根据参考文献 4 中“PLL VALUE SELECTION TABLE”设置。该表部分摘录如下：

表 2.3 推荐 PLL 值

输入频率	输出频率	MDIV	PDIV	SDIV
12.0000MHz	48.00 MHz	56(0x38)	2	2
12.0000MHz	405.00 MHz	127(0x7f)	2	1

当 mini2440 系统主频设置为 405MHZ，USB 时钟频率设置为 48MHZ 时，系统可以稳定运行，因此设置 MPLLCON 与 UPLLCON 为：

```
MPLLCON=(0x7f<<12) | (0x02<<4) | (0x01) = 0x7f021

UPLLCON=(0x38<<12) | (0x02<<4) | (0x02) = 0x38022
```

（7）关闭 MMU，cache

接着往下看：

```
#ifndef CONFIG_SKIP_LOWLEVEL_INIT
    bl    cpu_init_crit
#endif
```

cpu_init_crit 这段代码在 U-Boot 正常启动时才需要执行，若将 U-Boot 从 RAM 中启动则应该注释掉这段代码。

下面分析一下 cpu_init_crit 到底做了什么：

```
320  #ifndef CONFIG_SKIP_LOWLEVEL_INIT
321  cpu_init_crit:
322      /*
323       * 使数据 cache 与指令 cache 无效 */
324       */
325      mov       r0, #0
326      mcr p15, 0, r0, c7, c7, 0    /* 向 c7 写入 0 将使 ICache 与 DCache 无效*/
327      mcr p15, 0, r0, c8, c7, 0    /* 向 c8 写入 0 将使 TLB 失效 */
328
```

```
329     /*
330      * disable MMU stuff and caches
331      */
332     mrc p15, 0, r0, c1, c0, 0    /*  读出控制寄存器到 r0 中  */
333     bic  r0, r0, #0x00002300   @ clear bits 13, 9:8 (--V- --RS)
334     bic  r0, r0, #0x00000087   @ clear bits 7, 2:0 (B--- -CAM)
335     orr   r0, r0, #0x00000002   @ set bit 2 (A) Align
336     orr   r0, r0, #0x00001000   @ set bit 12 (I) I-Cache
337     mcr p15, 0, r0, c1, c0, 0    /*  保存 r0 到控制寄存器  */
338
339     /*
340      * before relocating, we have to setup RAM timing
341      * because memory timing is board-dependend, you will
342      * find a lowlevel_init.S in your board directory.
343      */
344     mov       ip, lr
345
346     bl    lowlevel_init
347
348     mov       lr, ip
349     mov       pc, lr
350  #endif /* CONFIG_SKIP_LOWLEVEL_INIT */
```

代码中的 c0，c1，c7，c8 都是 ARM920T 的协处理器 CP15 的寄存器。其中 c7 是 cache 控制寄存器，c8 是 TLB 控制寄存器。325~327 行代码将 0 写入 c7、c8，使 Cache，TLB 内容无效。

第 332~337 行代码关闭了 MMU。这是通过修改 CP15 的 c1 寄存器来实现的，先看 CP15 的 c1 寄存器的格式（仅列出代码中用到的位）：

表 2.3 CP15 的 c1 寄存器格式（部分）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
.	.	V	I	.	.	R	S	B	.	.	.	.	C	A	M

各个位的意义如下：

V： 表示异常向量表所在的位置，0：异常向量在 0x00000000；1：异常向量在 0xFFFF0000

I： 0 ：关闭 ICaches；1 ：开启 ICaches

R、S：用来与页表中的描述符一起确定内存的访问权限

B： 0 ：CPU 为小字节序；1 ： CPU 为大字节序

C： 0：关闭 DCaches；1：开启 DCaches

A： 0：数据访问时不进行地址对齐检查；1：数据访问时进行地址对齐检查

M： 0：关闭 MMU；1：开启 MMU

332~337 行代码将 c1 的 M 位置零，关闭了 MMU。

（8）初始化 RAM 控制寄存器

其中的 lowlevel_init 就完成了内存初始化的工作，由于内存初始化是依赖于开发板的，因此 lowlevel_init 的代码一般放在 board 下面相应的目录中。对于 mini2440，lowlevel_init 在 board/samsung/mini2440/lowlevel_init.S 中定义如下：

```
45  #define BWSCON   0x48000000       /* 13 个存储控制器的开始地址 */
  … …
129  _TEXT_BASE:
130      .word     TEXT_BASE
131
132  .globl lowlevel_init
133  lowlevel_init:
134      /* memory control configuration */
135      /* make r0 relative the current location so that it */
```

```
136     /* reads SMRDATA out of FLASH rather than memory ! */
137     ldr     r0, =SMRDATA
138     ldr   r1, _TEXT_BASE
139     sub  r0, r0, r1              /* SMRDATA 减 _TEXT_BASE 就是 13 个寄存器的偏移
地址 */
140     ldr   r1, =BWSCON   /* Bus Width Status Controller */
141     add     r2, r0, #13*4
142  0:
143     ldr     r3, [r0], #4    /*将 13 个寄存器的值逐一赋值给对应的寄存器*/
144     str     r3, [r1], #4
145     cmp     r2, r0
146     bne     0b
147
148     /* everything is fine now */
149     mov       pc, lr
150
151     .ltorg
152  /* the literal pools origin */
153
154  SMRDATA:            /*  下面是 13 个寄存器的值  */
155  .word  … …
156   .word  … …
      … …
```

lowlevel_init 初始化了 13 个寄存器来实现 RAM 时钟的初始化。lowlevel_init 函数对于 U-Boot 从 NAND Flash 或 NOR Flash 启动的情况都是有效的。

U-Boot.lds 链接脚本有如下代码：

```
.text：
{
        cpu/arm920t/start.o    (.text)
      board/samsung/mini2440/lowlevel_init.o (.text)
       board/samsung/mini2440/nand_read.o (.text)
     … …
}
```

board/samsung/mini2440/lowlevel_init.o 将被链接到 cpu/arm920t/start.o 后面，因此 board/samsung/mini2440/lowlevel_init.o 也在 U-Boot 的前 4KB 的代码中。

U-Boot 在 NAND Flash 启动时，lowlevel_init.o 将自动被读取到 CPU 内部 4KB 的内部 RAM 中。因此第 137~146 行的代码将从 CPU 内部 RAM 中复制寄存器的值到相应的寄存器中。

对于 U-Boot 在 NOR Flash 启动的情况，由于 U-Boot 连接时确定的地址是 U-Boot 在内存中的地址，而此时 U-Boot 还在 NOR Flash 中，因此还需要在 NOR Flash 中读取数据到 RAM 中。

由于 NOR Flash 的开始地址是 0，而 U-Boot 的加载到内存的起始地址是 TEXT_BASE，SMRDATA 标号在 Flash 的地址就是 SMRDATA－TEXT_BASE。

综上所述，lowlevel_init 的作用就是将 SMRDATA 开始的 13 个值复制给开始地址[BWSCON]的 13 个寄存器，从而完成了存储控制器的设置。

（9）复制 U-Boot 第二阶段代码到 RAM

cpu/arm920t/start.S 原来的代码是只支持从 NOR Flash 启动的，经过修改现在 U-Boot 在 NOR Flash 和 NAND Flash 上都能启动了，实现的思路是这样的：

```
    bl    bBootFrmNORFlash /*  判断 U-Boot 是在 NAND Flash 还是 NOR Flash 启动  */
    cmp       r0, #0          /*  r0 存放 bBootFrmNORFlash 函数返回值，若返回 0 表示 NAND Flash 启动，否则表示在 NOR Flash 启动  */
    beq nand_boot         /*  跳转到 NAND Flash 启动代码  */
```

```
/*  NOR Flash 启动的代码  */
     b     stack_setup        /* 跳过 NAND Flash 启动的代码 */

nand_boot:
/*  NAND Flash 启动的代码  */

stack_setup:
     /* 其他代码 */
```

其中 bBootFrmNORFlash 函数作用是判断 U-Boot 是在 NAND Flash 启动还是 NOR Flash 启动，若在 NOR Flash 启动则返回 1，否则返回 0。根据 ATPCS 规则，函数返回值会被存放在 r0 寄存器中，因此调用 bBootFrmNORFlash 函数后根据 r0 的值就可以判断 U-Boot 在 NAND Flash 启动还是 NOR Flash 启动。bBootFrmNORFlash 函数在 board/samsung/mini2440/nand_read.c 中定义如下：

```
int bBootFrmNORFlash(void)
{
   volatile unsigned int *pdw = (volatile unsigned int *)0;
   unsigned int dwVal;

   dwVal = *pdw;        /* 先记录下原来的数据 */
   *pdw = 0x12345678;
   if (*pdw != 0x12345678)      /* 写入失败，说明是在 NOR Flash 启动 */
   {
      return 1;
   }
```

```
    else                            /* 写入成功，说明是在 NAND Flash 启动 */
    {
        *pdw = dwVal;       /* 恢复原来的数据 */
        return 0;
    }
}
```

无论是从 NOR Flash 还是从 NAND Flash 启动，地址 0 处为 U-Boot 的第一条指令“ b   start_code”。

对于从 NAND Flash 启动的情况，其开始 4KB 的代码会被自动复制到 CPU 内部 4K 内存中，因此可以通过直接赋值的方法来修改。

对于从 NOR Flash 启动的情况，NOR Flash 的开始地址即为 0，必须通过一定的命令序列才能向 NOR Flash 中写数据，所以可以根据这点差别来分辨是从 NAND Flash 还是 NOR Flash 启动：向地址 0 写入一个数据，然后读出来，如果发现写入失败的就是 NOR Flash，否则就是 NAND Flash。

下面来分析 NOR Flash 启动部分代码：

```
208     adr  r0, _start             /* r0 <- current position of code   */
209     ldr   r1, _TEXT_BASE          /* test if we run from flash or RAM */


/* 判断 U-Boot 是否是下载到 RAM 中运行，若是，则不用 再复制到 RAM 中了，这种情况通常在调试 U-Boot 时才发生 */
210     cmp      r0, r1      /*_start 等于_TEXT_BASE 说明是下载到 RAM 中运行 */
211     beq stack_setup
212  /* 以下直到 nand_boot 标号前都是 NOR Flash 启动的代码 */
213     ldr   r2, _armboot_start
214     ldr   r3, _bss_start
215     sub  r2, r3, r2             /* r2 <- size of armboot          */
216     add r2, r0, r2             /* r2 <- source end address        */
```

```
217  /* 搬运 U-Boot 自身到 RAM 中*/
218  copy_loop:
219      ldmia     r0!, {r3-r10} /* 从地址为[r0]的 NOR Flash 中读入 8 个字的数据 */
220      stmia      r1!, {r3-r10} /* 将 r3 至 r10 寄存器的数据复制给地址为[r1]的内存 */
221      cmp       r0, r2                  /* until source end addreee [r2]    */
222      ble  copy_loop
223      b     stack_setup         /* 跳过 NAND Flash 启动的代码 */
```

下面再来分析 NAND Flash 启动部分代码：

```
nand_boot:
   mov r1, #NAND_CTL_BASE
   ldr r2, =( (7<<12)|(7<<8)|(7<<4)|(0<<0) )
   str r2, [r1, #oNFCONF]   /* 设置 NFCONF 寄存器 */

     /* 设置 NFCONT，初始化 ECC 编/解码器，禁止 NAND Flash 片选 */
   ldr r2, =( (1<<4)|(0<<1)|(1<<0) )
   str r2, [r1, #oNFCONT]

   ldr r2, =(0x6)          /* 设置 NFSTAT */
     str r2, [r1, #oNFSTAT]

     /* 复位命令，第一次使用 NAND Flash 前复位 */
   mov r2, #0xff
   strb r2, [r1, #oNFCMD]
   mov r3, #0
```

```
    /* 为调用 C 函数 nand_read_ll 准备堆栈 */
    ldr sp, DW_STACK_START
    mov fp, #0
    /* 下面先设置 r0 至 r2，然后调用 nand_read_ll 函数将 U-Boot 读入 RAM */
    ldr r0, =TEXT_BASE      /* 目的地址：U-Boot 在 RAM 的开始地址 */
    mov r1, #0x0            /* 源地址：U-Boot 在 NAND Flash 中的开始地址 */
    mov r2, #0x30000        /* 复制的大小,必须比 u-boot.bin 文件大,并且必须是 NAND
Flash 块大小的整数倍，这里设置为 0x30000（192KB） */
    bl  nand_read_ll            /* 跳转到 nand_read_ll 函数，开始复制 U-Boot 到 RAM
*/
     tst  r0, #0x0              /* 检查返回值是否正确 */
     beq stack_setup
bad_nand_read:
     loop2: b loop2    //infinite loop

.align 2
DW_STACK_START: .word STACK_BASE+STACK_SIZE-4
```

其中 NAND_CTL_BASE，oNFCONF 等在 include/configs/mini2440.h 中定义如下：

```
#define NAND_CTL_BASE  0x4E000000  // NAND Flash 控制寄存器基址

#define STACK_BASE  0x33F00000     //base address of stack
#define STACK_SIZE  0x8000         //size of stack

#define oNFCONF  0x00      /* NFCONF 相对于 NAND_CTL_BASE 偏移地址 */
#define oNFCONT  0x04      /* NFCONT 相对于 NAND_CTL_BASE 偏移地址*/
#define oNFADDR  0x0c     /* NFADDR 相对于 NAND_CTL_BASE 偏移地址*/
```

```
#define oNFDATA  0x10      /* NFDATA 相对于 NAND_CTL_BASE 偏移地址*/
#define oNFCMD   0x08     /* NFCMD 相对于 NAND_CTL_BASE 偏移地址*/
#define oNFSTAT  0x20       /* NFSTAT 相对于 NAND_CTL_BASE 偏移地址*/
#define oNFECC   0x2c            /* NFECC 相对于 NAND_CTL_BASE 偏移地址*/
```

NAND Flash 各个控制寄存器的设置在 S3C2440 的数据手册有详细说明，这里就不介绍了。

代码中 nand_read_ll 函数的作用是在 NAND Flash 中搬运 U-Boot 到 RAM，该函数在 board/samsung/mini2440/nand_read.c 中定义。

NAND Flash 根据 page 大小可分为 2 种：512B/page 和 2048B/page 的。这两种 NAND Flash 的读操作是不同的。因此就需要 U-Boot 识别到 NAND Flash 的类型，然后采用相应的读操作，也就是说 nand_read_ll 函数要能自动适应两种 NAND Flash。

参考 S3C2440 的数据手册可以知道：根据 NFCONF 寄存器的 Bit3（AdvFlash (Read only)）和 Bit2（PageSize (Read only)）可以判断 NAND Flash 的类型。Bit2、Bit3 与 NAND Flash 的 block 类型的关系如下表所示：

表 2.4 NFCONF 的 Bit3、Bit2 与 NAND Flash 的关系

Bit2 Bit3	0	1
0	256 B/page	512 B/page
1	1024 B/page	2048 B/page

由于的 NAND Flash 只有 512B/page 和 2048 B/page 这两种，因此根据 NFCONF 寄存器的 Bit3 即可区分这两种 NAND Flash 了。

完整代码见 board/samsung/mini2440/nand_read.c 中的 nand_read_ll 函数，这里给出伪代码：

```
int nand_read_ll(unsigned char *buf, unsigned long start_addr, int size)
{
```

```
//根据 NFCONF 寄存器的 Bit3 来区分 2 种 NAND Flash
    if( NFCONF & 0x8 )       /* Bit 是 1，表示是 2KB/page 的 NAND Flash */
    {
        ////////////////////////////////
        读取 2K block 的 NAND Flash
        ////////////////////////////////


    }
    else                    /* Bit 是 0，表示是 512B/page 的 NAND Flash */
    {
        ////////////////////////////////
        读取 512B block 的 NAND Flash
        ////////////////////////////////


    }
  return 0;
}
```

（10）设置堆栈

```
    /*  设置堆栈 */
stack_setup:
    ldr   r0, _TEXT_BASE          /* upper 128 KiB: relocated uboot   */
    sub  r0, r0, #CONFIG_SYS_MALLOC_LEN   /* malloc area              */
    sub  r0, r0, #CONFIG_SYS_GBL_DATA_SIZE /*  跳过全局数据区             */
#ifdef CONFIG_USE_IRQ
    sub  r0, r0, #(CONFIG_STACKSIZE_IRQ+CONFIG_STACKSIZE_FIQ)
```

```
#endif
    sub  sp, r0, #12          /* leave 3 words for abort-stack    */
```

只要将 sp 指针指向一段没有被使用的内存就完成栈的设置了。根据上面的代码可以知道 U-Boot 内存使用情况了，如下图所示：

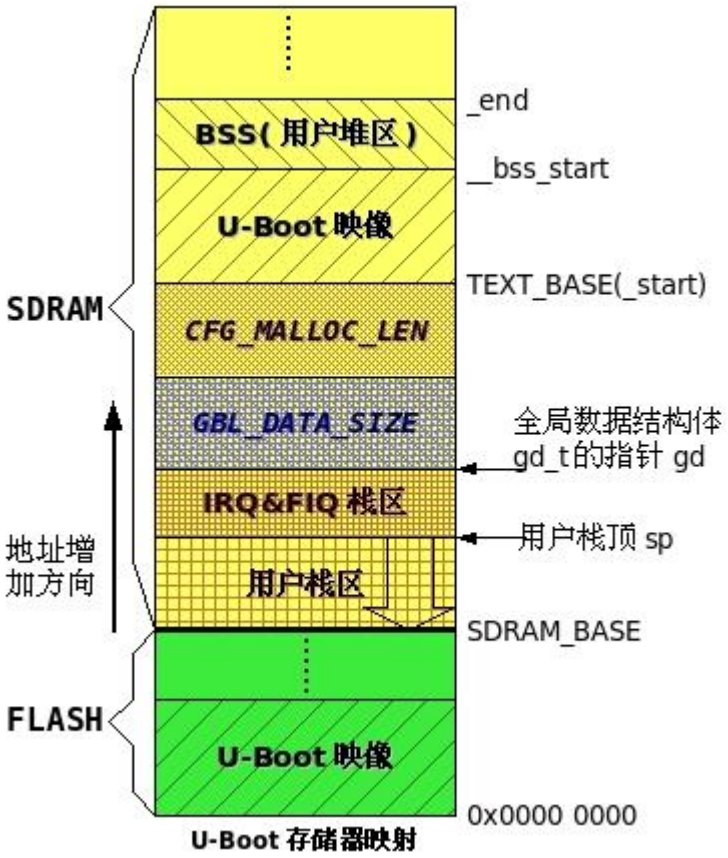

图 2.2 U-Boot 内存使用情况

（11）清除 BSS 段

```
clear_bss:
    ldr   r0, _bss_start          /* BSS 段开始地址，在 u-boot.lds 中指定*/
    ldr   r1, _bss_end            /* BSS 段结束地址，在 u-boot.lds 中指定*/
    mov      r2, #0x00000000
clbss_l:str     r2, [r0]         /* 将 bss 段清零*/
```

```
    add r0, r0, #4
    cmp     r0, r1
    ble  clbss_l
```

初始值为 0，无初始值的全局变量，静态变量将自动被放在 BSS 段。应该将这些变量的初始值赋为 0，否则这些变量的初始值将是一个随机的值，若有些程序直接使用这些没有初始化的变量将引起未知的后果。

（12）跳转到第二阶段代码入口

```
    ldr   pc, _start_armboot


_start_armboot:   .word  start_armboot
```

跳转到第二阶段代码入口 start_armboot 处。

1.1.2 U-Boot 启动第二阶段代码分析

start_armboot 函数在 lib_arm/board.c 中定义，是 U-Boot 第二阶段代码的入口。U-Boot 启动第二阶段流程如下：

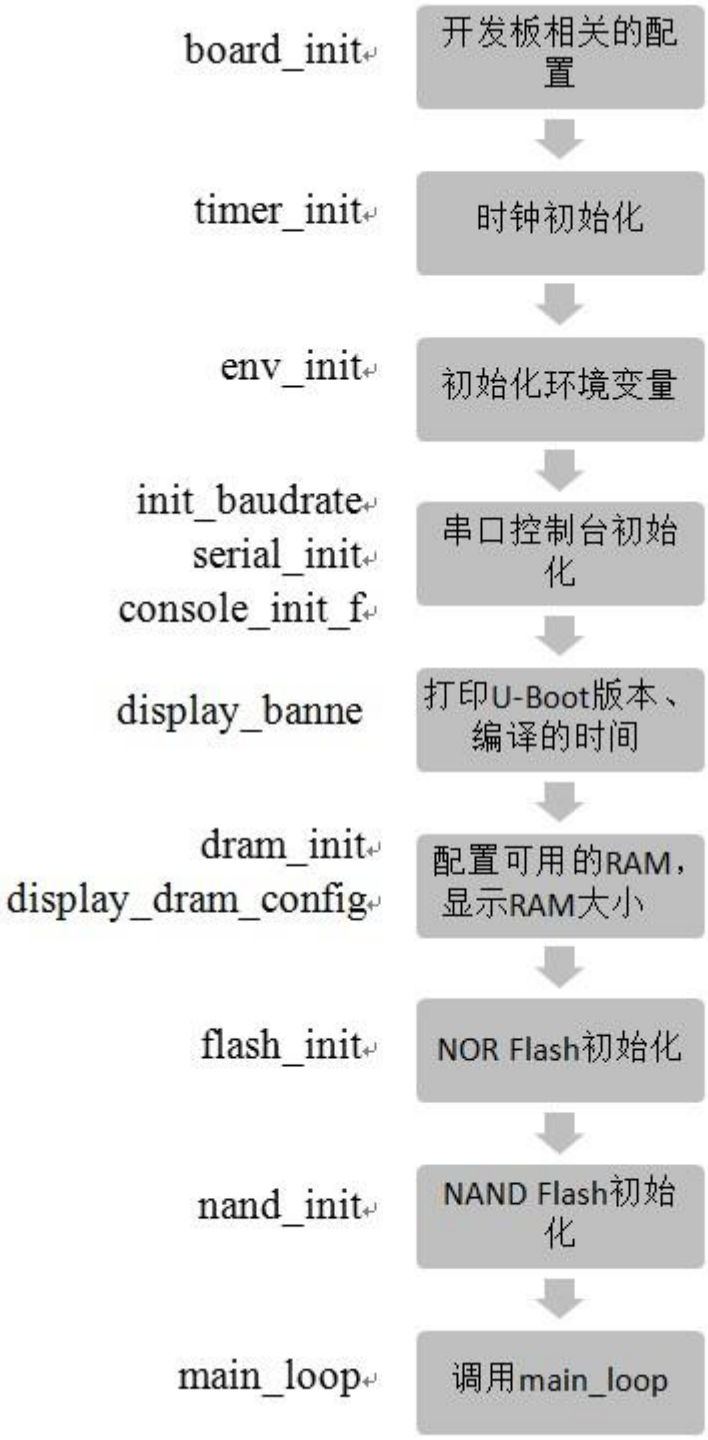

图 2.3 U-Boot 第二阶段执行流程

在分析 start_armboot 函数前先来看看一些重要的数据结构：

（1）gd_t 结构体

U-Boot 使用了一个结构体 gd_t 来存储全局数据区的数据，这个结构体在 include/asm-arm/global_data.h 中定义如下：

```
typedef  struct     global_data {
     bd_t            *bd;
     unsigned long      flags;
     unsigned long      baudrate;
     unsigned long      have_console;      /* serial_init() was called */
     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;
```

U-Boot 使用了一个存储在寄存器中的指针 gd 来记录全局数据区的地址：

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

DECLARE_GLOBAL_DATA_PTR 定义一个 gd_t 全局数据结构的指针，这个指针存放在指定的寄存器 r8 中。这个声明也避免编译器把 r8 分配给其它的变量。任何想要访问全局数据区的代码，只要代码开头加入"DECLARE_GLOBAL_DATA_PTR"一行代码，然后就可以使用 gd 指针来访问全局数据区了。

根据 U-Boot 内存使用图中可以计算 gd 的值：

```
gd = TEXT_BASE －CONFIG_SYS_MALLOC_LEN － sizeof(gd_t)
```

（2）bd_t 结构体

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

```
typedef struct bd_info {
   int              bi_baudrate;            /* 串口通讯波特率 */
   unsigned long     bi_ip_addr;         /* IP 地址*/
   struct environment_s       *bi_env;            /* 环境变量开始地址 */
```

```
    ulong           bi_arch_number;     /* 开发板的机器码 */
    ulong           bi_boot_params;     /* 内核参数的开始地址 */
    struct                      /* RAM 配置信息 */
    {
            ulong start;
            ulong size;
    }bi_dram[CONFIG_NR_DRAM_BANKS];
} bd_t;
```

U-Boot 启动内核时要给内核传递参数，这时就要使用 gd_t，bd_t 结构体中的信息来设置标记列表。

（3）init_sequence 数组

U-Boot 使用一个数组 init_sequence 来存储对于大多数开发板都要执行的初始化函数的函数指针。init_sequence 数组中有较多的编译选项，去掉编译选项后 init_sequence 数组如下所示：

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


init_fnc_t *init_sequence[] = {
      board_init,         /*开发板相关的配置--board/samsung/mini2440/mini2440.c */
      timer_init,           /* 时钟初始化-- cpu/arm920t/s3c24x0/timer.c */
      env_init,           /*初始化环境变量--common/env_flash.c 或
common/env_nand.c*/
      init_baudrate,      /*初始化波特率-- lib_arm/board.c */
      serial_init,          /* 串口初始化-- drivers/serial/serial_s3c24x0.c */
      console_init_f,    /* 控制通讯台初始化阶段 1-- common/console.c */
      display_banner,   /*打印 U-Boot 版本、编译的时间-- gedit lib_arm/board.c */
      dram_init,          /*配置可用的 RAM-- board/samsung/mini2440/mini2440.c
*/
```

```
    display_dram_config,            /* 显示 RAM 大小-- lib_arm/board.c */
    NULL,
};
```

其中的 board_init 函数在 board/samsung/mini2440/mini2440.c 中定义，该函数设置了 MPLLCOM，UPLLCON，以及一些 GPIO 寄存器的值，还设置了 U-Boot 机器码和内核启动参数地址 ：

```
    /* MINI2440 开发板的机器码 */
    gd->bd->bi_arch_number = MACH_TYPE_MINI2440;

    /* 内核启动参数地址 */
    gd->bd->bi_boot_params = 0x30000100;
```

其中的 dram_init 函数在 board/samsung/mini2440/mini2440.c 中定义如下：

```
    int dram_init (void)
    {
        /* 由于 mini2440 只有 */
        gd->bd->bi_dram[0].start = PHYS_SDRAM_1;
        gd->bd->bi_dram[0].size = PHYS_SDRAM_1_SIZE;

        return 0;
    }
```

mini2440 使用 2 片 32MB 的 SDRAM 组成了 64MB 的内存，接在存储控制器的 BANK6，地址空间是 0x30000000~0x34000000。

在 include/configs/mini2440.h 中 PHYS_SDRAM_1 和 PHYS_SDRAM_1_SIZE 分别被定义为 0x30000000 和 0x04000000（64M）。

分析完上述的数据结构，下面来分析 start_armboot 函数：

```
void start_armboot (void)
```

```
{
    init_fnc_t **init_fnc_ptr;
    char *s;
    … …
    /* 计算全局数据结构的地址 gd */
    gd = (gd_t*)(_armboot_start - CONFIG_SYS_MALLOC_LEN - sizeof(gd_t));
    … …
    memset ((void*)gd, 0, sizeof (gd_t));
    gd->bd = (bd_t*)((char*)gd - sizeof(bd_t));
    memset (gd->bd, 0, sizeof (bd_t));
    gd->flags |= GD_FLG_RELOC;

    monitor_flash_len = _bss_start - _armboot_start;

/* 逐个调用 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 ();
        }
    }

/* armboot_start 在 cpu/arm920t/start.S 中被初始化为 u-boot.lds 连接脚本中的_start
*/
    mem_malloc_init (_armboot_start - CONFIG_SYS_MALLOC_LEN,
            CONFIG_SYS_MALLOC_LEN);
```

```
/* NOR Flash 初始化 */
#ifndef CONFIG_SYS_NO_FLASH
    /* configure available FLASH banks */
    display_flash_config (flash_init ());
#endif /* CONFIG_SYS_NO_FLASH */

    … …
/* NAND Flash 初始化*/
#if defined(CONFIG_CMD_NAND)
    puts ("NAND:  ");
    nand_init();        /* go init the NAND */
#endif
    … …
    /*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");
    stdio_init (); /* get the devices list going. */
    jumptable_init ();
    … …
    console_init_r (); /* fully init console as a device */
    … …
    /* enable exceptions */
```

```
    enable_interrupts ();


#ifdef CONFIG_USB_DEVICE
    usb_init_slave();
#endif


    /* 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
    … …
    /* 网卡初始化 */
#if defined(CONFIG_CMD_NET)
#if defined(CONFIG_NET_MULTI)
    puts ("Net:   ");
#endif
    eth_initialize(gd->bd);
… …
#endif
```

```
    /* main_loop() can return to retry autoboot, if so just run it again. */
    for (;;) {
        main_loop ();
    }
    /* NOTREACHED - no way out of command loop except booting */
}
```

main_loop 函数在 common/main.c 中定义。一般情况下，进入 main_loop 函数若干秒内没有

1.1.3 U-Boot 启动 Linux 过程

U-Boot 使用标记列表（tagged list）的方式向 Linux 传递参数。标记的数据结构式是 tag，在 U-Boot 源代码目录 include/asm-arm/setup.h 中定义如下：

```
struct tag_header {
    u32 size;       /* 表示 tag 数据结构的联合 u 实质存放的数据的大小*/
    u32 tag;        /* 表示标记的类型 */
};

struct tag {
    struct tag_header hdr;
    union {
        struct tag_core          core;
        struct tag_mem32      mem;
        struct tag_videotext   videotext;
        struct tag_ramdisk     ramdisk;
        struct tag_initrd   initrd;
        struct tag_serialnr       serialnr;
```

```
        struct tag_revision      revision;
        struct tag_videolfb      videolfb;
        struct tag_cmdline       cmdline;

        /*
         * Acorn specific
         */
        struct tag_acorn   acorn;
        /*
         * DC21285 specific
         */
        struct tag_memclk       memclk;
    } u;
};
```

U-Boot 使用命令 bootm 来启动已经加载到内存中的内核。而 bootm 命令实际上调用的是 do_bootm 函数。对于 Linux 内核，do_bootm 函数会调用 do_bootm_linux 函数来设置标记列表和启动内核。do_bootm_linux 函数在 lib_arm/bootm.c 中定义如下：

```
59  int do_bootm_linux(int flag, int argc, char *argv[], bootm_headers_t *images)
60  {
61      bd_t      *bd = gd->bd;
62      char      *s;
63      int   machid = bd->bi_arch_number;
64      void      (*theKernel)(int zero, int arch, uint params);
65
66  #ifdef CONFIG_CMDLINE_TAG
67      char *commandline = getenv ("bootargs");   /* U-Boot 环境变量 bootargs */
```

```
68  #endif
    … …
73      theKernel = (void (*)(int, int, uint))images->ep; /* 获取内核入口地址 */
    … …
86  #if defined (CONFIG_SETUP_MEMORY_TAGS) || \
87      defined (CONFIG_CMDLINE_TAG) || \
88      defined (CONFIG_INITRD_TAG) || \
89      defined (CONFIG_SERIAL_TAG) || \
90      defined (CONFIG_REVISION_TAG) || \
91      defined (CONFIG_LCD) || \
92      defined (CONFIG_VFD)
93      setup_start_tag (bd);                          /* 设置 ATAG_CORE 标志 */
    … …
100 #ifdef CONFIG_SETUP_MEMORY_TAGS
101     setup_memory_tags (bd);                        /* 设置内存标记 */
102 #endif
103 #ifdef CONFIG_CMDLINE_TAG
104     setup_commandline_tag (bd, commandline);       /* 设置命令行标记 */
105 #endif
    … …
113     setup_end_tag (bd);                            /* 设置 ATAG_NONE 标
志 */
114 #endif
115
116     /* we assume that the kernel is in place */
```

```
117     printf ("\nStarting kernel ...\n\n");
      … …
126     cleanup_before_linux ();          /* 启动内核前对 CPU 作最后的设置 */
127
128     theKernel (0, machid, bd->bi_boot_params);      /* 调用内核 */
129     /* does not return */
130
131     return 1;
132 }
```

其中的 setup_start_tag，setup_memory_tags，setup_end_tag 函数在 lib_arm/bootm.c 中定义如下：

（1）setup_start_tag 函数

```
static void setup_start_tag (bd_t *bd)
{
     params = (struct tag *) bd->bi_boot_params;  /* 内核的参数的开始地址 */

     params->hdr.tag = ATAG_CORE;
     params->hdr.size = tag_size (tag_core);

     params->u.core.flags = 0;
     params->u.core.pagesize = 0;
     params->u.core.rootdev = 0;

     params = tag_next (params);
}
```

标记列表必须以 ATAG_CORE 开始，setup_start_tag 函数在内核的参数的开始地址设置了一个 ATAG_CORE 标记。

（2）setup_memory_tags 函数

```
static void setup_memory_tags (bd_t *bd)
{
    int i;
/*设置一个内存标记 */
    for (i = 0; i < CONFIG_NR_DRAM_BANKS; i++) {
        params->hdr.tag = ATAG_MEM;
        params->hdr.size = tag_size (tag_mem32);

        params->u.mem.start = bd->bi_dram[i].start;
        params->u.mem.size = bd->bi_dram[i].size;

        params = tag_next (params);
    }
}
```

setup_memory_tags 函数设置了一个 ATAG_MEM 标记，该标记包含内存起始地址，内存大小这两个参数。

（3）setup_end_tag 函数

```
static void setup_end_tag (bd_t *bd)
{
    params->hdr.tag = ATAG_NONE;
    params->hdr.size = 0;
}
```

标记列表必须以标记 ATAG_NONE 结束，setup_end_tag 函数设置了一个 ATAG_NONE 标记，表示标记列表的结束。

U-Boot 设置好标记列表后就要调用内核了。但调用内核前，CPU 必须满足下面的条件：

（1） CPU 寄存器的设置

- r0=0
- r1=机器码
- r2=内核参数标记列表在 RAM 中的起始地址

（2） CPU 工作模式

- 禁止 IRQ 与 FIQ 中断
- CPU 为 SVC 模式

（3） 使数据 Cache 与指令 Cache 失效

do_bootm_linux 中调用的 cleanup_before_linux 函数完成了禁止中断和使 Cache 失效的功能。cleanup_before_linux 函数在 cpu/arm920t/cpu.中定义：

```
int cleanup_before_linux (void)
{
    /*
     * this function is called just before we call linux
     * it prepares the processor for linux
     *
     * we turn off caches etc ...
     */

    disable_interrupts ();        /* 禁止 FIQ/IRQ 中断 */

    /* turn off I/D-cache */
    icache_disable();             /* 使指令 Cache 失效 */
```

```
    dcache_disable();           /* 使数据 Cache 失效 */
    /* flush I/D-cache */
    cache_flush();              /* 刷新 Cache */

    return 0;
}
```

由于 U-Boot 启动以来就一直工作在 SVC 模式，因此 CPU 的工作模式就无需设置了。

do_bootm_linux 中：

```
64      void    (*theKernel)(int zero, int arch, uint params);
… …
73      theKernel = (void (*)(int, int, uint))images->ep;
… …
128     theKernel (0, machid, bd->bi_boot_params);
```

第 73 行代码将内核的入口地址“images->ep”强制类型转换为函数指针。根据 ATPCS 规则，函数的参数个数不超过 4 个时，使用 r0~r3 这 4 个寄存器来传递参数。因此第 128 行的函数调用则会将 0 放入 r0，机器码 machid 放入 r1，内核参数地址 bd->bi_boot_params 放入 r2，从而完成了寄存器的设置，最后转到内核的入口地址。

到这里，U-Boot 的工作就结束了，系统跳转到 Linux 内核代码执行。

1.1.4 U-Boot 添加命令的方法及 U-Boot 命令执行过程

下面以添加 menu 命令（启动菜单）为例讲解 U-Boot 添加命令的方法。

（1） 建立 common/cmd_menu.c

习惯上通用命令源代码放在 common 目录下，与开发板专有命令源代码则放在 board/<board_dir>目录下，并且习惯以“cmd_<命令名>.c”为文件名。

（2） 定义“menu”命令

在 cmd_menu.c 中使用如下的代码定义“menu”命令：

```
_BOOT_CMD(
```

```
    menu,   3,   0,   do_menu,
    "menu - display a menu, to select the items to do something\n",
    " - display a menu, to select the items to do something"
);
```

其中 U_BOOT_CMD 命令格式如下：

```
U_BOOT_CMD(name,maxargs,rep,cmd,usage,help)
```

各个参数的意义如下：

```
name：命令名，非字符串，但在 U_BOOT_CMD 中用“#”符号转化为字符串
maxargs：命令的最大参数个数
rep：是否自动重复（按 Enter 键是否会重复执行）
cmd：该命令对应的响应函数
usage：简短的使用说明（字符串）
help：较详细的使用说明（字符串）
```

在内存中保存命令的 help 字段会占用一定的内存，通过配置 U-Boot 可以选择是否保存 help 字段。若在 include/configs/mini2440.h 中定义了 CONFIG_SYS_LONGHELP 宏，则在 U-Boot 中使用 help 命令查看某个命令的帮助信息时将显示 usage 和 help 字段的内容，否则就只显示 usage 字段的内容。

U_BOOT_CMD 宏在 include/command.h 中定义：

```
#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}
```

“##”与“#”都是预编译操作符，"##"有字符串连接的功能，“#”表示后面紧接着的是一个字符串。

其中的 cmd_tbl_t 在 include/command.h 中定义如下：

```
struct cmd_tbl_s {
    char        *name;      /* 命令名 */
    int         maxargs;    /* 最大参数个数 */
```

```
    int         repeatable;    /* 是否自动重复 */
    int         (*cmd)(struct cmd_tbl_s *, int, int, char *[]);  /*  响应函数 */
    char            *usage;         /* 简短的帮助信息 */
#ifdef    CONFIG_SYS_LONGHELP
    char            *help;          /*  较详细的帮助信息 */
#endif
#ifdef CONFIG_AUTO_COMPLETE
    /* 自动补全参数 */
    int         (*complete)(int argc, char *argv[], char last_char, int maxv, char
*cmdv[]);
#endif
};
typedef struct cmd_tbl_s  cmd_tbl_t;
```

一个 cmd_tbl_t 结构体变量包含了调用一条命令的所需要的信息。

其中 Struct_Section 在 include/command.h 中定义如下：

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

凡是带有__attribute__ ((unused,section (".u_boot_cmd"))属性声明的变量都将被存放在".u_boot_cmd"段中，并且即使该变量没有在代码中显式的使用编译器也不产生警告信息。

在 U-Boot 连接脚本 u-boot.lds 中定义了".u_boot_cmd"段：

```
    . = .;
    __u_boot_cmd_start = .;         /*将 __u_boot_cmd_start 指定为当前地址 */
    .u_boot_cmd : { *(.u_boot_cmd) }
    __u_boot_cmd_end = .;           /*  将__u_boot_cmd_end 指定为当前地址  */
```

这表明带有“.u_boot_cmd”声明的函数或变量将存储在“u_boot_cmd”段。这样只要将 U-Boot 所有命令对应的 cmd_tbl_t 变量加上“.u_boot_cmd”声明，编译器就会自动将其放在

“u_boot_cmd”段，查找 cmd_tbl_t 变量时只要在__u_boot_cmd_start 与 __u_boot_cmd_end 之间查找就可以了。

因此“menu”命令的定义经过宏展开后如下：

```
cmd_tbl_t __u_boot_cmd_menu __attribute__ ((unused,section (".u_boot_cmd"))) = {menu, 3, 0, do_menu, "menu - display a menu, to select the items to do something\n", " - display a menu, to select the items to do something"}
```

实质上就是用 U_BOOT_CMD 宏定义的信息构造了一个 cmd_tbl_t 类型的结构体。编译器将该结构体放在“u_boot_cmd”段，执行命令时就可以在“u_boot_cmd”段查找到对应的 cmd_tbl_t 类型结构体。

（3） 实现命令的函数

在 cmd_menu.c 中添加“menu”命令的响应函数的实现。具体的实现代码略：

```
int do_menu (cmd_tbl_t *cmdtp, int flag, int argc, char *argv[])
{
    /* 实现代码略 */
}
```

（4） 将 common/cmd_menu.c 编译进 u-boot.bin

在 common/Makefile 中加入如下代码：

```
COBJS-$(CONFIG_BOOT_MENU) += cmd_menu.o
```

在 include/configs/mini2440.h 加入如代码：

```
#define CONFIG_BOOT_MENU 1
```

重新编译下载 U-Boot 就可以使用 menu 命令了

（5）menu 命令执行的过程

在 U-Boot 中输入“menu”命令执行时，U-Boot 接收输入的字符串“menu”，传递给 run_command 函数。run_command 函数调用 common/command.c 中实现的 find_cmd 函数在__u_boot_cmd_start 与__u_boot_cmd_end 间查找命令，并返回 menu 命令的 cmd_tbl_t 结构。然后 run_command 函数使用返回的 cmd_tbl_t 结构中的函数指针调用 menu 命令的响应函数 do_menu，从而完成了命令的执行。