目录

[1.1 U-Boot工作过程 1](#_Toc482556093)

[1.1.1 U-Boot启动第一阶段代码分析 1](#_Toc482556094)

[1.1.2 U-Boot启动第二阶段代码分析 18](#_Toc482556095)

[1.1.3 U-Boot启动Linux过程 25](#_Toc482556096)

[1.1.4 U-Boot添加命令的方法及U-Boot命令执行过程 30](#_Toc482556097)

## 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/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。