U-Boot启动过程

一个嵌入式的存储设备通过通常包括四个分区：

第一分区：存放的当然是u-boot

第二个分区：存放着u-boot要传给系统内核的参数

第三个分区：是系统内核（kernel）

第四个分区：则是根文件系统

如下图所示：

嵌入式系统中，系统在上电或复位时通常都从地址0x00000000处开始执行，而在这个地址处安排的通常就是系统的Boot Loader程序。一个可执行的image必须有一个入口点，并且只能有一个**全局入口点**，所以要通知编译器这个入口在哪里。由此我们可以找到程序的入口点是在**u-boot.lds**中指定的，其中ENTRY(\_start)说明程序从\_start开始运行，而他指向的是start.o文件。

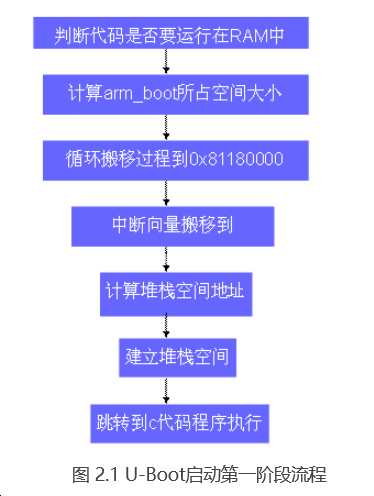
U-Boot启动内核的过程可以分为两个阶段，两个阶段的功能如下：

       （1）第一阶段的功能

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

      （2）第二阶段的功能

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



 根据cpu/arm920t/u-boot.lds中指定的**连接方式**：

ENTRY(\_start)  
SECTIONS  
{  
. = 0x00000000; //起始地址  
. = ALIGN(4); //4字节对齐  
.text : //test指代码段，上面3行标识是不占用任何空间的  
{  
cpu/arm920t/start.o (.text) //这里把start.o放在第一位就表示把start.s编  
译时放到最开始，这就是为什么把uboot烧到起始地址上它肯定运行的是start.s  
\*(.text)  
}  
. = ALIGN(4); //前面的 “.” 代表当前值，是计算一个当前的值，是计算上  
面占用的整个空间，再加一个单元就表示它现在的位置。

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**

globl \_startglobal                                    /\*声明一个符号可被其它文件引用，相当于声明了一个全局变量，.globl与.global相同\*/

\_start:    b     start\_code                    /\* 复位 \*/b是不带返回的跳转(bl是带返回的跳转)，意思是无条件直接跳转到start\_code标号出执行程序

**ldr   pc, \_undefined\_instruction**      /\* 未定义指令向量 l---dr相当于mov操作\*/

       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  /\*就是在当前地址，即\_undefined\_instruction 处存放 undefined\_instruction\*/

\_software\_interrupt:  .word software\_interrupt

\_prefetch\_abort:  .word prefetch\_abort

\_data\_abort:        .word data\_abort

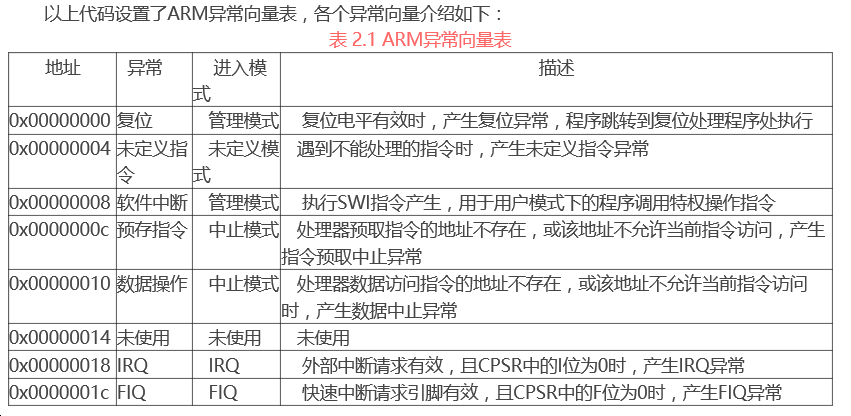
\_not\_used:          .word not\_used

\_irq:                     .word irq

\_fiq:                     .word fiq

.word 用法解释：

他们是**系统定义的异常**，一上电程序跳转到start\_code异常处执行相应的汇编指令，下面定义出的都是不同的异常，比如软件发生软中断时，CPU就会去执行软中断的指令，这些异常中断在CUP中地址是从0开始，每个异常占4个字节。



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，即设置相应的CPSR程序状态字，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。

       操作系统先注册一个总的中断，然后去查是由哪个中断源产生的中断，再去查用户注册的中断表，查出来后就去执行用户定义的用户中断处理函数。

**（3）设置控制寄存器地址**

# if defined(CONFIG\_S3C2400)        /\*关闭看门狗\*/

#  define pWTCON 0x15300000       /\*;看门狗寄存器\*/

#  define INTMSK  0x14400008        /\*;中断屏蔽寄存器\*/

#  define CLKDIVN      0x14800014 /\*;时钟分频寄存器\*/

#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 次级中断屏蔽寄存器\*/

#  define CLKDIVN      0x4C000014                   /\* CLKDIVN寄存器地址 ;时钟分频寄存器\*/

# 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   /\*将pwtcon寄存器地址赋给R0\*/

       mov       r1, #0x0      /\*r1的内容为0\*/

       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 \*/

关闭看门狗。**否则在U-Boot启动过程中，CPU将不断重启。**

**为什么要关看门狗？**

         就是防止不同得两个以上得CPU，进行喂狗的时间间隔问题：**说白了**，就是你运行的代码如果超出喂狗时间，而你不关狗，就会导致，你代码还没运行完又得去喂狗，就这样反复得重启CPU，那你代码永远也运行不完，所以，得先关看门狗得原因，就是这样。

关狗---详细的原因：

      关闭看门狗，关闭中断，所谓的**喂狗**是每隔一段时间给某个寄存器置位而已，在实际中会专门启动一个线程或进程会专门喂狗，当上层软件出现故障时就会停止喂狗，

      停止喂狗之后，cpu会自动复位，一般都在**外部专门有一个看门狗**，做一个外部的电路，不在cpu内部使用看门狗，cpu内部的看门狗是复位的cpu

       当开发板很复杂时，有好几个cpu时，就不能完全让板子复位，但我们通常都让整个板子复位。看门狗每隔短时间就会喂狗，问题是在两次喂狗之间的时间间隔内，运行的代码的时间是否够用，两次喂狗之间的代码是否在两次喂狗的时间延迟之内，如果在延迟之外的话，代码还没改完就又进行喂狗，代码永远也改不完

**（5）屏蔽中断**

       /\*

        \* mask all IRQs by setting all bits in the INTMR - default

        \*/

       mov       r1, #0xffffffff     /\*屏蔽所有中断， 某位被置1则对应的中断被屏蔽 \*/ /\*寄存器中的值\*/

       ldr   r0, =INTMSK       /\*将管理中断的寄存器地址赋给ro\*/

       str   r1, [r0]                  /\*将全r1的值赋给ro地址中的内容\*/

       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。

**（6）设置MPLLCON,UPLLCON, CLKDIVN**

# if defined(CONFIG\_S3C2440)

#define MPLLCON   0x4C000004

#define UPLLCON   0x4C000008

          ldr  r0, =CLKDIVN   ;设置时钟

          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。

**设置时钟分频，为什么要设置时钟？**

起始可以不设，系统能不能跑起来和频率没有任何关系，频率的设置是要让外围的设备能承受所设置的频率，如果频率过高则会导致cpu操作外围设备失败

说白了：**设置频率，就为了CPU能去操作外围设备**

**（7）关闭MMU，cache  ------（也就是做bank的设置）**

       接着往下看：

#ifndef CONFIG\_SKIP\_LOWLEVEL\_INIT

       bl    cpu\_init\_crit  /\* ;跳转并把转移后面紧接的一条指令地址保存到链接寄存器LR(R14)中，以此来完成子程序的调用\*/

#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**

**为什么要关闭catch和MMU呢？catch和MMU是做什么用的？**

**MMU**是Memory Management Unit的缩写，中文名是[内存管理](http://baike.baidu.com/view/4541016.htm" \t "_blank)单元，它是中央处理器（CPU）中用来管理虚拟存储器、物理存储器的控制线路

    同时也负责**[虚拟地址](http://baike.baidu.com/view/1499823.htm" \t "_blank)**映射为**物理地址**，以及提供硬件机制的内存访问授权

**概述：**

**一，关catch**

       catch和MMU是通过CP15管理的，刚上电的时候，CPU还不能管理他们

       上电的时候MMU必须关闭，指令catch可关闭，可不关闭，但数据catch一定要关闭

       否则可能导致刚开始的代码里面，去取数据的时候，从catch里面取，而这时候RAM中数据还没有catch过来，导致数据预取异常

**二：关MMU**

      因为MMU是;把虚拟地址转化为物理地址得作用

      而目的是设置控制寄存器，而控制寄存器本来就是实地址（物理地址），再使能MMU，不就是多此一举了吗？

**详细分析---**

      Catch是cpu内部的一个**2级**缓存，它的作用是将常用的数据和指令放在cpu内部，MMU是用来把虚实地址转换为物理地址用的

      我们的目的:是**设置控制的寄存器**，寄存器都是实地址（物理地址），如果既要开启MMU又要做虚实地址转换的话，中间还多一步，多此一举了嘛?  
      先要把实地址转换成虚地址，然后再做设置，但对uboot而言就是起到一个简单的初始化的作用和引导操作系统，如果开启MMU的话，很麻烦，也没必要，所以关闭MMU.

       说到catch就必须提到一个关键字**Volatile**，以后在设置寄存器时会经常遇到，**他的本质**：是告诉编译器不要对我的代码进行优化，作用是让编写者感觉不倒变量的变化情况（也就是说，让它执行速度加快吧）

**优化的过程**：是将**常用的代码取出来放到catch中**，它**没有**从实际的**物理地址去取**，它直接从**cpu的缓**存中**去取**，但常用的代码就是为了感觉一些常用变量的变化

**优化原因：**如果正在取数据的时候发生跳变，那么就感觉不到变量的变化了，所以在这种情况下要用Volatile关键字告诉编译器不要做优化，每次从实际的物理地址中去取指令，**这就是为什么关闭catch关闭MMU**。

         但在C语言中是不会关闭catch和MMU的，会打开，如果编写者要感觉外界变化，或变化太快，从catch中取数据会有误差，就加一个关键字Volatile。

**（8）初始化RAM控制寄存器**

                     bl lowlevel\_init下来初始化各个bank，把各个bank设置必须搞清楚，对以后移植复杂的uboot有很大帮助  
                     设置完毕后拷贝uboot代码到4k空间，拷贝完毕后执行内存中的uboot代码

      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       0x33F80000, board/config.mk中这段话表示，用户告诉编译器编译地址的起始地址

131

132  .globl lowlevel\_init

133  lowlevel\_init:

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。

**问题一：如果换一块开发板有可能改哪些东西？**  
**首先，cpu的运行模式，如果需要对cpu进行设置那就设置，管看门狗，关中断不用改，时钟有可能要改，如果能正常使用则不用改，关闭catch和MMU不用改，设置bank有可能要改。最后一步拷贝时看地址会不会变，如果变化也要改，执行内存中代码，地址有可能要改。**  
**问题二：Nor Flash和Nand Flash本质区别：**

**就在于是否进行代码拷贝，也就是下面代码所表述：无论是Nor Flash还是Nand Flash，核心思想就是将uboot代码搬运到内存中去运行，但是没有拷贝bss后面这段代码，只拷贝bss前面的代码，bss代码是放置全局变量的。Bss段代码是为了清零，拷贝过去再清零重复操作**

**（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 \*/

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。

**（10）设置堆栈**

       /\*  设置堆栈 \*/

stack\_setup:

**（11）清除BSS段**

clear\_bss:

**（12）跳转到第二阶段代码入口**

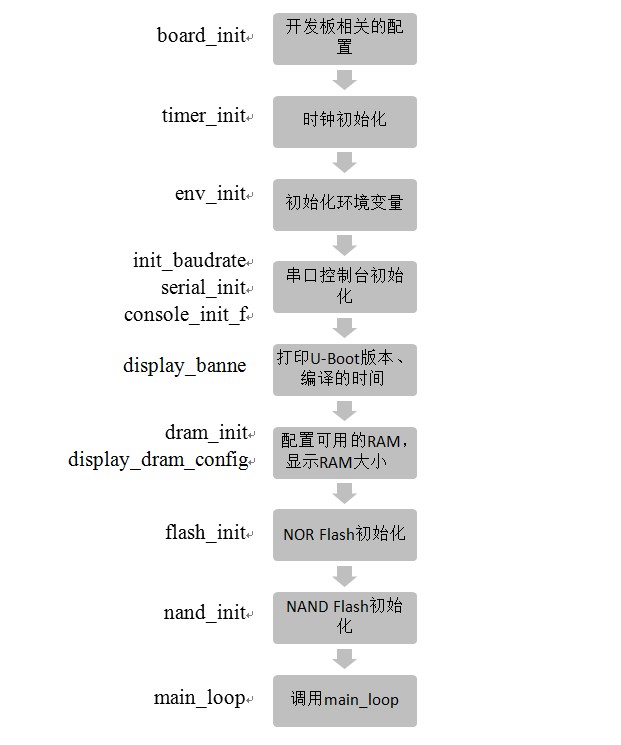
       ldr   pc, **\_start\_armboot**

\_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：



  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

**（2）bd\_t结构体**

       bd\_t在include/asm-arm.u/u-boot.h

       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。

       第一阶段调用start\_armboot指向C语言执行代码区，首先它要从内存上的重定位数据获得不完全配置的全局数据表格和板级信息表格，即获得**gd\_t**和**bd\_t**，

这两个类型变量记录了**刚启动时**的信息，并将要记录作为引导内核和文件系统的参数，如bootargs等等，并且将来还会在启动内核时，由uboot交由kernel时会有所用。

**（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

Ø  r0=0

Ø  r1=机器码

Ø  r2=内核参数标记列表在RAM中的起始地址

（2）    CPU工作模式

Ø  禁止IRQ与FIQ中断

Ø  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。