# linux内核启动流程分析

## 1. linux内核自解压过程

uboot将uImage头部进行解析，并将里面的zImage搬移到指定的内存位置，然后跳转到该内存位置处开始执行。从zImage的编译过程分析，zImage实际上是由解压代码和压缩的内核镜像组成。所以zImage开始部分的代码会将压缩的内核镜像进行解压，并搬移到指定的位置。

从arch/arm /boot/compressed/vmlinux.lds中可以看到程序的入口为start。

OUTPUT\_ARCH(arm)

ENTRY(\_start)

SECTIONS

{

. = 0;

\_text = .;

.text : {

\_start = .;

\*(.start)

\*(.text)

\*(.text.\*)

\*(.fixup)

\*(.gnu.warning)

\*(.rodata)

\*(.rodata.\*)

\*(.glue\_7)

\*(.glue\_7t)

\*(.piggydata)

. = ALIGN(4);

}

在arch/arm/boot/compressed/head.S找到start入口，开始分析自解压代码。

### 1.1 DEBUG宏定义

head.S最开始部分定义了调试相关的代码。

如果开启了DEBUG宏，支持ICEDCC，使用使用内部调试协处理器CP14作为调试通道。

否则使用串口作为调试通道，一开始include <asm/arch/debug-macro.S>，这个文件和平台相关，里面定义了相关的串口操作(**addruart，senduart，waituart，busyuart**)。因这个时候系统还没有起来，所以它所用的串口配置参数是依赖于前一级bootloader所设置好的。

#ifdef DEBUG

#if defined(CONFIG\_DEBUG\_ICEDCC)

#ifdef CONFIG\_CPU\_V6

.macro loadsp, rb

.endm

.macro writeb, ch, rb

mcr p14, 0, \ch, c0, c5, 0

.endm

#else

.macro loadsp, rb

.endm

.macro writeb, ch, rb

mcr p14, 0, \ch, c0, c1, 0

.endm

#endif

#else

#include <asm/arch/debug-macro.S>

.macro writeb, ch, rb

senduart \ch, \rb

.endm

.macro loadsp, rb

mov \rb, #0x50000000

add \rb, \rb, #0x4000 \* CONFIG\_S3C2410\_LOWLEVEL\_UART\_PORT

.endm

.macro kputc,val

mov r0, \val

bl putc

.endm

.macro kphex,val,len

mov r0, \val

mov r1, #\len

bl phex

.endm

writeb宏表示写一个字节到串口。

loadsp宏用于获取串口N的寄存器参数，2440串口0的data寄存器地址为0x50000000，串口1的data寄存器地址为0x50000000+0x4000。

kputc kphex用于打印单个字符和16进制数。

还有两个宏未贴出代码：  
debug\_reloc\_start用来打印出一些代码重定位后的信息。

debug\_reloc\_end用来把解压后的内核的起始256字节的数据dump出来，查看是否正确。

### 1.2 start代码

zImage启动时，最先执行的代码从start开始。

start**:**

.type start,#function

.rept 8

mov r0, r0

.endr

首先声明start符号类型为函数，接下来重复执行mov r0,r0 8次，一共32个字节用来存放中断向量表。

b 1f

.word 0x016f2818 @ Magic numbers to help the loader

.word start @ absolute load**/**run zImage address

.word \_edata @ zImage end address

1**:** mov r7, r1 @ save architecture ID

mov r8, r2 @ save atags pointer

zImage 0x24 0x28 0x2C这三个位置用于存放魔数，start地址，zImage结束地址。

在反汇编代码中如下，其中\_edata的定义在vmlinux.lds中。

24: 016f2818 cmneq pc, r8, lsl r8

28: 00000000 andeq r0, r0, r0

2c: 001c357c andeqs r3, ip, ip, ror r5

接下来将machine id和内核启动参数地址暂存到r7和r8中，解压完了再复原。

继续下一步，读取cpsr并判断是否处理器处于supervisor模式——从u-boot进入kernel，系统已经处于SVC32模式；而利用angel进入则处于user模式。最后关闭中断。

mrs r2, cpsr @ get current mode

tst r2, #3 @ not user?

bne not\_angel

mov r0, #0x17 @ angel\_SWIreason\_EnterSVC

swi 0x123456 @ angel\_SWI\_ARM

not\_angel**:**

mrs r2, cpsr @ turn off interrupts to

orr r2, r2, #0xc0 @ prevent angel from running

msr cpsr\_c, r2

获取实际运行地址和编译时定义的差值，判断是否需要重定向。  
(1)获取LC0表的当前运行地址，adr得到的是运行地址，而不是链接地址。  
(2)把LC0表的数据分别加载到r1 r2 r3 r4 r5 r6 r12 r13中。  
通过比较r0 r1也就是比较LC0表的加载地址和链接地址是否相同，相同就调用not\_relocated函数，就不用再进行代码重定位了，否则还需要对代码在内存进行搬移重定位到链接地址。

.text

adr r0, LC0

ldmia r0, **{**r1, r2, r3, r4, r5, r6, ip, sp**}**

subs r0, r0, r1 @ calculate the delta offset

@ **if** delta is zero, we are

beq not\_relocated @ running at the address we

@ were linked at.

.type LC0, #object

LC0**:** .word LC0 @ r1

.word \_\_bss\_start @ r2

.word \_end @ r3

.word zreladdr @ r4

.word \_start @ r5

.word \_got\_start @ r6

.word \_got\_end @ ip

.word user\_stack**+**4096 @ sp

通过反汇编可以得到LC0链接地址为0x00000138，而实际运行地址则为0x30008000后面的一段地址，所以两者并不相等。

00000138 <LC0>:

138: 00000138 001c357c 001cb9b4 30008000 8...|5.........0

148: 00000000 001c3510 001c3570 001cc9b4 .....5..p5......

现在发现链接地址和运行地址不一致，需要进行重定向操作。  
\_start,\_got\_start,\_got\_end加上偏移，经过重定位后都是绝对地址了。

add r5, r5, r0

add r6, r6, r0

add ip, ip, r0

继续重定向操作。  
\_\_bss\_start(**bss段start**),\_end(**bss段end**), user\_stack+4096加上偏移。

其中user\_stack大小为4k，跟随在bss\_end后，.stack段在compressed/vmlinux.lds中定义。

add r2, r2, r0

add r3, r3, r0

add sp, sp, r0

.align

.section ".stack", "w"

user\_stack**:** .space 4096

重定向got表的入口项。  
GOT全称global offset table，GOT是一个数组，存在ELF image的数据段中，他们是一些指向objects 的指针(通常是数据objects)。在链接compressed/vmlinux时，使用了-fpic参数，该选项告诉编绎器使用GOT和PLT的方法重定位。  
在这里从got\_start地址取出4字节数，添加偏移量，一直到got\_end地址结束。

1**:** ldr r1, **[**r6, #0] @ relocate entries in the GOT

add r1, r1, r0 @ table. This fixes up the

str r1, **[**r6**]**, #4 @ C references.

cmp r6, ip

blo 1b

清空BSS段。

not\_relocated**:** mov r0, #0

1**:** str r0, **[**r2**]**, #4 @ clear bss

str r0, **[**r2**]**, #4

str r0, **[**r2**]**, #4

str r0, **[**r2**]**, #4

cmp r2, r3

blo 1b

 开启指令和数据Cache ，为了加快解压速度。

bl cache\_on

这里的 r1,r2 之间的空间为解压缩内核程序所使用，也是传给 decompress\_kernel 的第二和第三的参数。

mov r1, sp @ malloc space above stack

add r2, sp, #0x10000 @ 64k max

检查解压地址，保证解压地址和当前程序的地址不重叠, 面分配了64KB的空间来做解压时的数据缓存。

**/\***

**\***   检查是否会覆盖内核映像本身

**\***   r4 **=** 最终解压后的内核首地址

**\***   r5 **=** zImage 的运行时首地址，一般为0x30008000

**\***   r2 **=** end of malloc space分配空间的结束地址（并且处于本映像的前面）

**\*** 基本要求：r4 **>=** r2 或者 r4 **+** 映像长度 **<=** r5

**\*/**

#(1)vmlinux 的起始地址大于 zImage 运行时所需的最大地址r2,那么直接将 zImage 解压到 vmlinux 的目标地址，在这里r4=0x30008000, r2=sp+64K.

cmp r4, r2

bhs wont\_overwrite  #如果r4大于或等于r2的话

#(2)zImage的起始地址大于 vmlinux 的目标起始地址加上vmlinux 大小\*4的地址，将 zImage 直接解压到 vmlinux 的目标地址.

#r3=sp-r5(zImage的大小) r0=r4+r3\*4(0x30008000+zImage size\*4) r5=0x30008000

sub r3, sp, r5 @ **>** compressed kernel size

add r0, r4, r3, lsl #2 @ allow for 4x expansion

cmp r0, r5

bls wont\_overwrit

上述两种方案均不成立，继续往下走。开始调用解压函数，在调用函数之前需要准备所需参数。

mov r5, r2 @ decompress after malloc space

mov r0, r5

mov r3, r7

bl decompress\_kernel

decompress\_kernel需要的四个参数分别为解压的内核地址、缓存区首地址、缓存区尾地址、和芯片ID，返回解压缩代码的长度。  
r0为malloc区域后面的地址，也就将内核解压到malloc区域后。  
r1和r2为malloc区域首地址和尾地址。  
r3为arch id。

ulg

decompress\_kernel**(**ulg output\_start**,** ulg free\_mem\_ptr\_p**,** ulg free\_mem\_ptr\_end\_p**,**int arch\_id**)**

**{**

output\_data **=** **(**uch **\*)**output\_start**;** /\* Points to kernel start \*/

free\_mem\_ptr **=** free\_mem\_ptr\_p**;**

free\_mem\_ptr\_end **=** free\_mem\_ptr\_end\_p**;**

\_\_machine\_arch\_type **=** arch\_id**;**

arch\_decomp\_setup**();**

makecrc**();**

putstr**(**"Uncompressing Linux..."**);**

gunzip**();**

putstr**(**" done, booting the kernel.\n"**);**

**return** output\_ptr**;**

**}**

decompress\_kernel函数存在misc.c文件中。  
(1)首先传入参数：解压后内核地址，缓存开始地址，缓存结束地址，arch id。这些参数通过寄存器r0(r5),r1,r2,r3(r7)传入。  
(2)接着执行proc\_decomp\_setup()，它在include/asm-arm/proc-armv/uncompress.h文件中。主要刷新并起用i cache，锁住交换缓存，这是一段嵌入的arm汇编代码。  
(3)接着执行arch\_decomp\_setup()，它在include/asm-arm/arch-pxa/uncompress.h文件中，是一个空函数，用于扩展。  
(4)然后执行makecrc()，它在lib/inflate.c中，主要将产生CRC-32 table，进行循环冗余校验。  
(5)调用gunzip()解压kernel，它也在lib/inflate.c中。  
(6)返回head.S，解压后kernel的长度传给r0，解压后的内核地址预先在r5中定义了。

关于gunzip，解释一些概念。  
gunzip()函数总是用指针inbuf来表示输入数据源, 用变量insize表示输入数据源的大小(字节数), 用inptr来表示下一个待处理的字节。  
此外, 它也总是使用get\_byte()宏来从输入数据源获取数据. get\_byte()宏的定义如下:   
#define get\_byte() (inptr < insize ? inbuf[inptr++] : fill\_inbuf())   
函数fill\_inbuf()用来在一开始的时候, 设置输入数据源的位置。

int fill\_inbuf**(**void**)**

**{**

**if** **(**insize **!=** 0**)**

error**(**"ran out of input data"**);**

inbuf **=** input\_data**;**

insize **=** input\_len**;**

inptr **=** 1**;**

**return** inbuf**[**0**];**

**}**

变量input\_data, input\_data\_end表示输入数据源的起始地址和结束地址, input\_len表示输入数据原的大小(字节数). input\_len的值应该等于(input\_data\_end - input\_data)。  
input\_data和input\_data\_end在piggy.S中定义。

.section .piggydata,#alloc

.globl input\_data

input\_data**:**

.incbin "arch/arm/boot/compressed/piggy.gz"

.globl input\_data\_end

input\_data\_end**:**

解压完毕后，内核长度返回值存放在r0寄存器里。在内核末尾空出128字节的栈空间，并且使其长度128字节对齐。

add r0, r0, #127 + 128 @ alignment + stack

bic r0, r0, #127 @ align the kernel length

完成了解压缩之后，由于内核没有解压到正确的地址，最后必须通过代码搬移来搬到指定的地址0x30008000。搬运过程中有可能会覆盖掉现在运行的重定位代码，所以必须将这段代码搬运到安全的地方，这里搬运到的地址是解压缩了的代码的后面r5+r0的位置。

/\*

 \* r0     = 解压后内核长度

 \* r1-r3  = 未使用

 \* r4     = 真正内核执行地址  0x30008000

 \* r5     = 临时解压内核Image的起始地址

 \* r6     = 处理器ID

 \* r7     = 体系结构ID

 \* r8     = 参数列表               0x30000100

 \* r9-r14 = 未使用

 \*/

add r1, r5, r0  @ end of decompressed kernel 解压内核的结束地址

adr r2, reloc\_start

ldr r3, LC1              @ LC1**:** .word reloc\_end **-** reloc\_start表示reloc\_start段代码的大小

add r3, r2, r3

1**:** ldmia r2**!**, **{**r9 **-** r14**}** @ copy relocation code r2地址内容搬移到r1地址

stmia r1**!**, **{**r9 **-** r14**}**

ldmia r2**!**, **{**r9 **-** r14**}**

stmia r1**!**, **{**r9 **-** r14**}**

cmp r2, r3

blo 1b

bl cache\_clean\_flush   @清 cache

add pc, r5, r0              @ call relocation code 跳转到重定位代码开始执行

接下来重定位代码reloc\_start将Image 的代码从缓冲区r5帮运到最终的目的地r4:0x30008000处。

reloc\_start**:**

add r9, r5, r0         @r9中存放的是临时解压内核的末尾地址

sub r9, r9, #128       @不拷贝堆栈

mov r1, r4       @r1中存放的是目的地址0x30008000

1**:**

.rept 4

ldmia r5**!**, **{**r0, r2, r3, r10 **-** r14**}** @ relocate kernel

stmia r1**!**, **{**r0, r2, r3, r10 **-** r14**}** **/\***搬运内核Image的过程**\*/**

.endr

cmp r5, r9

blo 1b

mov sp, r1                            **/\***留出堆栈的位置**\*/**

add sp, sp, #128              @ relocate the stack

最后便是在0x30008000启动linux内核了。

call\_kernel**:** bl cache\_clean\_flush

bl cache\_off

mov r0, #0 @ must be zero

mov r1, r7 @ restore architecture number

mov r2, r8 @ restore atags pointer

mov pc, r4 @ call kernel

## 2.linux内核启动引导过程

该阶段主要完成的工作有，cpu ID检查，machine ID(也就是开发板ID)检查，创建初始化页表，设置C代码运行环境，跳转到内核第一个真正的C函数startkernel开始执行。  
vmlinux的链接地址为0xC0008000，在未开启mmu的情况下，有些代码必须是地址无关的。

### 2.1入口代码

入口代码如下：

\_\_HEAD   #该宏定义了下面的代码位于".head.text"段内

.type stext, %function  #声明stext为函数

ENTRY**(**stext**)**            #第二阶段的入口地址\*/

setmode PSR\_F\_BIT **|** PSR\_I\_BIT **|** SVC\_MODE, r9  #进入超级权限模式，关中断

接下来从协处理器CP15，C0读取CPU ID,然后在\_\_proc\_info\_begin开始的段中进行查找，如果找到，则返回对应处理器相关结构体在物理地址空间的首地址到r5，最后保存在r10中。

### 2.2 查找cpu id

mrc p15, 0, r9, c0, c0               #get processor id 取出cpu id

bl \_\_lookup\_processor\_type           # r5=procinfo r9=cpuid

分析\_\_lookup\_processor\_type函数前需要了解一些其他知识。  
内核所支持的每一种CPU 类型都由结构体proc\_info\_list来描述。该结构体定义如下：

#include/asm-arm/procinfo.h

struct proc\_info\_list **{**

unsigned int cpu\_val**;**

unsigned int cpu\_mask**;**

unsigned long \_\_cpu\_mm\_mmu\_flags**;** /\* used by head.S \*/

unsigned long \_\_cpu\_io\_mmu\_flags**;** /\* used by head.S \*/

unsigned long \_\_cpu\_flush**;** /\* used by head.S \*/

const char **\***arch\_name**;**

const char **\***elf\_name**;**

unsigned int elf\_hwcap**;**

const char **\***cpu\_name**;**

struct processor **\***proc**;**

struct cpu\_tlb\_fns **\***tlb**;**

struct cpu\_user\_fns **\***user**;**

struct cpu\_cache\_fns **\***cache**;**

**};**

对于 arm920 来说，其对应结构体在文件 linux/arch/arm/mm/proc-arm920.S 中初始化。 将\_\_arm920\_proc\_info存放在.proc.info.init段中。

.section ".proc.info.init", #alloc, #execinstr

.type \_\_arm920\_proc\_info,#object

\_\_arm920\_proc\_info**:**

.long 0x41009200

.long 0xff00fff0

.long PMD\_TYPE\_SECT **|** \

PMD\_SECT\_BUFFERABLE **|** \

PMD\_SECT\_CACHEABLE **|** \

PMD\_BIT4 **|** \

PMD\_SECT\_AP\_WRITE **|** \

PMD\_SECT\_AP\_READ

.long PMD\_TYPE\_SECT **|** \

PMD\_BIT4 **|** \

PMD\_SECT\_AP\_WRITE **|** \

PMD\_SECT\_AP\_READ

b \_\_arm920\_setup

.long cpu\_arch\_name

.long cpu\_elf\_name

.long HWCAP\_SWP **|** HWCAP\_HALF **|** HWCAP\_THUMB

.long cpu\_arm920\_name

.long arm920\_processor\_functions

.long v4wbi\_tlb\_fns

.long v4wb\_user\_fns

#ifndef CONFIG\_CPU\_DCACHE\_WRITETHROUGH

.long arm920\_cache\_fns

#else

.long v4wt\_cache\_fns

#endif

.size \_\_arm920\_proc\_info, . **-** \_\_arm920\_proc\_info

.proc.info.init段定义在arch/arm/kernel/vmlinux.lds中。

\_\_proc\_info\_begin = .;

\*(.proc.info.init)

\_\_proc\_info\_end = .;

有了上述知识，现在来分析\_\_lookup\_processor\_type函数。

#arch/arm/kernel/head-common.S

**.**type \_\_lookup\_processor\_type**,** **%**function

\_\_lookup\_processor\_type**:**

adr r3**,** 3f /\*取地址指令,这里的3f是向前symbol名称是3的位置\*/

/\*f是forward之意。标志3在此代码之后声明\*/

ldmda r3**,** **{**r5 **-** r7**}**/\*r5=\_\_proc\_info\_begin r6= \_\_proc\_info\_end r7=.\*/

sub r3**,** r3**,** r7 /\*r3中存储的是3f处的物理地址,而r7存储的是3f处的虚拟地址\*/

/\*这一行是计算当前程序运行的物理地址和虚拟地址的差值,将其保存到r3中\*/

add r5**,** r5**,** r3 /\*将r5存储的虚拟地址(\_\_proc\_info\_begin)转换成物理地址\*/

add r6**,** r6**,** r3 /\*将r5存储的虚拟地址(\_\_proc\_info\_begin)转换成物理地址\*/

1**:** ldmia r5**,** **{**r3**,** r4**}**/\*对照struct proc\_info\_list,可以得知,这句是将当前proc\_info的cpu\_val和cpu\_mask分别存r3, r4中\*/

**and** r4**,** r4**,** r9 /\*r9中存储了processor id,与r4的cpu\_mask进行逻辑与操作,得到我们需要的值\*/

teq r3**,** r4 /\*将上面得到的值与cpu\_val进行比较\*/

beq 2f /\*如果相等,说明我们找到了对应的processor type,跳到2处返回\*/

add r5**,** r5**,** #PROC\_INFO\_SZ /\*(如果不相等),将r5指向下一个proc\_info\*/

cmp r5**,** r6 /\*和r6比较,检查是否到了\_\_proc\_info\_end\*/

blo 1b /\*如果没有到\_\_proc\_info\_end,表明还有proc\_info配置,返回1继续查找\*/

mov r5**,** #0 /\*执行到这里,说明所有的proc\_info都匹配过了,但是没有找到匹配的,将r5设置成0(unknown processor)\*/

2**:** mov pc**,** lr

/\*

\* This provides a C-API version of the above function.

\*/

ENTRY**(**lookup\_processor\_type**)**

stmfd sp**!,** **{**r4 **-** r7**,** r9**,** lr**}**

mov r9**,** r0

bl \_\_lookup\_processor\_type

mov r0**,** r5

ldmfd sp**!,** **{**r4 **-** r7**,** r9**,** pc**}**

/\*

\* Look in include/asm-arm/procinfo.h and arch/arm/kernel/arch.[ch] for

\* more information about the \_\_proc\_info and \_\_arch\_info structures.

\*/

**.**long \_\_proc\_info\_begin

**.**long \_\_proc\_info\_end

3**:** **.**long **.**

**.**long \_\_arch\_info\_begin

**.**long \_\_arch\_info\_end

如果r5=0，跳转到\_error\_p。

movs r10, r5 @ invalid processor **(**r5**=**0**)?**

beq \_\_error\_p @ yes, error 'p'

### 2.3 查找machine id

机器 ID是由u-boot引导内核是通过thekernel第二个参数传递进来的，现在保存在r1中,在\_\_arch\_info\_begin开始的段中进行查找，如果找到，则返回machine对应相关结构体在物理地址空间的首地址到r5，最后保存在r8中。

bl \_\_lookup\_machine\_type @ r5**=**machinfo

movs r8, r5 @ invalid machine **(**r5**=**0**)?**

beq \_\_error\_a @ yes, error 'a'

同样的，先来分析下和machine相关的内容。machine\_desc为相关的结构体。

//include/asm-arm/mach/arch.h

struct machine\_desc **{**

/\*

\* Note! The first four elements are used

\* by assembler code in head-armv.S

\*/

unsigned int nr**;** /\* architecture number \*/

unsigned int phys\_io**;** /\* start of physical io \*/

unsigned int io\_pg\_offst**;** /\* byte offset for io

\* page tabe entry \*/

const char **\***name**;** /\* architecture name \*/

unsigned long boot\_params**;** /\* tagged list \*/

unsigned int video\_start**;** /\* start of video RAM \*/

unsigned int video\_end**;** /\* end of video RAM \*/

unsigned int reserve\_lp0 **:**1**;** /\* never has lp0 \*/

unsigned int reserve\_lp1 **:**1**;** /\* never has lp1 \*/

unsigned int reserve\_lp2 **:**1**;** /\* never has lp2 \*/

unsigned int soft\_reboot **:**1**;** /\* soft reboot \*/

void **(\***fixup**)(**struct machine\_desc **\*,**

struct tag **\*,** char **\*\*,**

struct meminfo **\*);**

void **(\***map\_io**)(**void**);**/\* IO mapping function \*/

void **(\***init\_irq**)(**void**);**

struct sys\_timer **\***timer**;** /\* system tick timer \*/

void **(\***init\_machine**)(**void**);**

**};**

对于平台2440，machine\_desc初始化代码。

#/arch/arm/mach-s3c2440/mach-smdk2440.c

MACHINE\_START**(**S3C2440**,** "SMDK2440"**)**

/\* Maintainer: Ben Dooks <ben@fluff.org> \*/

**.**phys\_io **=** S3C2410\_PA\_UART**,**

**.**io\_pg\_offst **=** **(((**u32**)**S3C24XX\_VA\_UART**)** **>>** 18**)** **&** 0xfffc**,**

**.**boot\_params **=** S3C2410\_SDRAM\_PA **+** 0x100**,**

**.**init\_irq **=** s3c24xx\_init\_irq**,**

**.**map\_io **=** smdk2440\_map\_io**,**

**.**init\_machine **=** smdk2440\_machine\_init**,**

**.**timer **=** **&**s3c24xx\_timer**,**

MACHINE\_END

//include/asm-arm/mach-types.h:375:

#define MACH\_TYPE\_S3C2440 362

#define MACHINE\_START(\_type,\_name) \

static const struct machine\_desc \_\_mach\_desc\_##\_type \

\_\_used \

\_\_attribute\_\_((\_\_section\_\_(".arch.info.init"))) = { \

.nr = MACH\_TYPE\_##\_type, \

.name = \_name,

#define MACHINE\_END \

}

\_\_attribute\_\_((\_\_section\_\_(".arch.info.init")))表明该结构体在并以后存放的位置。 vmlinux.lds中定义了该段的位置。

\_\_arch\_info\_begin **=** **.;**

**\*(.**arch**.**info**.**init**)**

\_\_arch\_info\_end **=** **.;**

接下来对\_\_lookup\_machine\_type函数进行分析。

**.**type \_\_lookup\_machine\_type**,** **%**function

\_\_lookup\_machine\_type**:**

adr r3**,** 3b /\*r3为标号3处的物理地址，标号3请见\_\_lookup\_processor\_type\*/

ldmia r3**,** **{**r4**,** r5**,** r6**}**/\*r4为标号3虚拟地址,r5=\_\_arch\_info\_begin r6=\_\_arch\_info\_end\*/

sub r3**,** r3**,** r4 /\*计算标号3虚拟地址和物理地址的差值\*/

add r5**,** r5**,** r3 /\*将\_\_arch\_info\_begin转换成物理地址\*/

add r6**,** r6**,** r3 /\*将\_\_arch\_info\_end转换成物理地址\*/

1**:** ldr r3**,** **[**r5**,** #MACHINFO\_TYPE**]** /\*r5是machine\_desc结构体地址\*/

                  /\*r3 = r5 + MACHINFO\_TYPE=machine\_desc结构中定义的nr成员，即机器类型ID\*/

teq r3**,** r1 /\* r1为uboot传进来的machine id，和r1进行比较\*/

beq 2f /\*如果相等，跳到标号2，返回\*/

add r5**,** r5**,** #SIZEOF\_MACHINE\_DESC /\*不相等，继续寻找\*/

cmp r5**,** r6 /\*是否到达\_\_arch\_info\_end位置\*/

blo 1b /\*未到达，继续查找\*/

mov r5**,** #0 /\*未找到,r5置0\*/

2**:** mov pc**,** lr

### 2.4 创建临时页表

bl \_\_create\_page\_tables

和页表相关的宏定义。

#define KERNEL\_RAM\_PADDR (PHYS\_OFFSET + TEXT\_OFFSET)

**.**macro pgtbl**,** rd

ldr\rd**,** **=(**KERNEL\_RAM\_PADDR **-** 0x4000**)**

**.**endm

/\*

其中：PHYS\_OFFSET在arch/arm/mach-s3c2410/include/mach/memory.h定义，为UL(0x30000000)，

而TEXT\_OFFSET在arch/arm/Makefile中定义，为内核镜像在内存中到内存开始位置的偏移（字节），为$(textofs-y)

textofs-y也在文件arch/arm/Makefile中定义，为textofs-y   := 0x00008000，r4 = 30004000为临时页表的起始地址，

首先即是初始化16K的页表，高12位虚拟地址为页表索引，每个页表索引占4个字节，所以为4K\*4 = 16K，大页表，每一个页表项，映射1MB虚拟地址.

\*/

临时页表主要建立两个映射关系：  
虚拟地址0x30000000🡪物理地址0x30000000，大小为1M。  
虚拟地址0xC0000000🡪物理地址0x30000000，大小为整个kernel。

**.**type \_\_create\_page\_tables**,** **%**function

\_\_create\_page\_tables**:**

pgtbl r4 //r4中存放的为页表的基地址，最终该地址会写入cp15的寄存器c2，这个值必须是16K对齐的

 /\*页表将4GB的地址空间分成若干个1MB的段(section)，因此页表包含4096个页表项(section entry)。

每个页表项是32bits(4 bytes)，因而页表占用4096\*4=16k的内存空间。下面的代码是将这16k的页表清\*/

mov r0**,** r4 //把页表的基地址存放到r0中

mov r3**,** #0 //把r3清0

add r6**,** r0**,** #0x4000 //r6指向16K的末尾

1**:** str r3**,** **[**r0**],** #4 //把16K的页表空间清0

str r3**,** **[**r0**],** #4

str r3**,** **[**r0**],** #4

str r3**,** **[**r0**],** #4

teq r0**,** r6

bne 1b

/\*从proc\_info\_list结构中获取字段 \_\_cpu\_mm\_mmu\_flags ，

该字段包含了存储空间访问权限等, 此处指令执行之后r7=0x00000c1e\*/

ldr r7**,** **[**r10**,** #PROCINFO\_MM\_MMUFLAGS**]**

//虚拟地址0x30000000--->物理地址0x30000000 大小为1M

mov r6**,** pc**,** lsr #20 //过PC值的高12位(右移20位),得到kernel的section,r6=0x300

orr r3**,** r7**,** r6**,** lsl #20 //得到页表需要设置的值r3=0x300<<20|MMU\_flags,r3为第一个页表描述符

//页表描述符的低20位存放各种标志位，高12位存放地址。

str r3**,** **[**r4**,** r6**,** lsl #2**]** //设置页表:将r3保存到0x3004c00=r6<<2=0x300\*4=0xc00+r4=0x30004c00中

//也就是如果需要访问0x30000000需要从0x30004c00里面找到对应的物理地址。

//虚拟地址0xc0100000--->物理地址0x30100000 直到kernel结束

add r0**,** r4**,** #**(**KERNEL\_START **&** 0xff000000**)** **>>** 18 //0x30004000+((0xc0008000&0xff000000)>>18)=0x30004000+0x3000=0x30007000=r0

str r3**,** **[**r0**,** #**(**KERNEL\_START **&** 0x00f00000**)** **>>** 18**]!** //将r3存储到r0+((0xc0008000&0x00f00000)>>18)=0x30007000+0x3000=0x3000a000

ldr r6**,** **=(**KERNEL\_END **-** 1**)** //获取内核的尾部虚拟地址存于r6中

add r0**,** r0**,** #4 //第一个地址条目存放在 0x30007004 处，以后依次递增

add r6**,** r4**,** r6**,** lsr #18 //计算最后一个地址条目存放的位置

1**:** cmp r0**,** r6 //填充这之间的地址条目

add r3**,** r3**,** #1 **<<** 20

strls r3**,** **[**r0**],** #4

bls 1b

//虚拟地址0xc0000000--->物理地址0x30000000 大小为1M

add r0**,** r4**,** #PAGE\_OFFSET **>>** 18 // r6=0x30000c1e

orr r6**,** r7**,** #**(**PHYS\_OFFSET **&** 0xff000000**)**

**.if** **(**PHYS\_OFFSET **&** 0x00f00000**)**

orr r6**,** r6**,** #**(**PHYS\_OFFSET **&** 0x00f00000**)**

**.**endif

str r6**,** **[**r0**]** //将0x30000c1e存于0x30007000处

mov pc**,** lr

**.**ltor

### 2.5 \_\_arm920\_setup

首先执行\_\_arm920\_setup。

/\*把\_\_switch\_data标号处的地址放入r13寄存器，当执行完\_\_enable\_mmu函数时会把r13寄存器的值赋值给pc，跳转到\_\_switch\_data 处执行\*/

ldr r13**,** \_\_switch\_data

/\*把\_\_enable\_mmu函数的地址值，赋值给lr寄存器，当执行完\_\_arm920\_setup时，返回后执行\_\_enable\_mmu \*/

adr lr**,** \_\_enable\_mmu

/\*查询\_\_arm920\_proc\_info,得到初始化函数为b \_\_arm920\_setup\*/

add pc**,** r10**,** #PROCINFO\_INITFUN

\_\_arm920\_setup禁止ICache/Dcache/数据Cache/指令Cache。

.type \_\_arm920\_setup, #function

\_\_arm920\_setup**:**

mov r0, #0

mcr p15, 0, r0, c7, c7 @ invalidate I,D caches on v4

mcr p15, 0, r0, c7, c10, 4 @ drain write buffer on v4

#ifdef CONFIG\_MMU

mcr p15, 0, r0, c8, c7 @ invalidate I,D TLBs on v4

#endif

adr r5, arm920\_crval

ldmia r5, **{**r5, r6**}**

mrc p15, 0, r0, c1, c0 @ get control register v4

bic r0, r0, r5

orr r0, r0, r6

mov pc, lr

.size \_\_arm920\_setup, . **-** \_\_arm920\_setup

### 2.6 打开mmu

**.**type \_\_enable\_mmu**,** **%**function

\_\_enable\_mmu**:**

#ifdef CONFIG\_ALIGNMENT\_TRAP

orr r0**,** r0**,** #CR\_A

#else

bic r0**,** r0**,** #CR\_A

#endif

#ifdef CONFIG\_CPU\_DCACHE\_DISABLE

bic r0**,** r0**,** #CR\_C

#endif

#ifdef CONFIG\_CPU\_BPREDICT\_DISABLE

bic r0**,** r0**,** #CR\_Z

#endif

#ifdef CONFIG\_CPU\_ICACHE\_DISABLE

bic r0**,** r0**,** #CR\_I

#endif

mov r5**,** #**(**domain\_val**(**DOMAIN\_USER**,** DOMAIN\_MANAGER**)** **|** \

domain\_val**(**DOMAIN\_KERNEL**,** DOMAIN\_MANAGER**)** **|** \

domain\_val**(**DOMAIN\_TABLE**,** DOMAIN\_MANAGER**)** **|** \

domain\_val**(**DOMAIN\_IO**,** DOMAIN\_CLIENT**))**

mcr p15**,** 0**,** r5**,** c3**,** c0**,** 0 @ load domain access register

mcr p15**,** 0**,** r4**,** c2**,** c0**,** 0 @ load page table pointer

b \_\_turn\_mmu\_o

**.**type \_\_turn\_mmu\_on**,** **%**function

\_\_turn\_mmu\_on**:**

mov r0**,** r0

mcr p15**,** 0**,** r0**,** c1**,** c0**,** 0 @ write control reg

mrc p15**,** 0**,** r3**,** c0**,** c0**,** 0 @ read id reg

mov r3**,** r3

mov r3**,** r3

mov pc**,** r1

### 2.7 设置c语言执行环境

\_\_mmap\_switched函数专用来设置C语言的执行环境，比如重定位工作，堆栈，以及BSS段的清零。   
\_\_switch\_data变量先定义了一系里面处量的数据，如重定位和数据段的地址，BSS段的地址，pocessor\_id和\_\_mach\_arch\_type变量的地址等。

//linux/arch/arm/kernel/head-common.S

**.**type \_\_switch\_data**,** **%**object

\_\_switch\_data**:**

**.**long \_\_mmap\_switched

**.**long \_\_data\_loc @ r4

**.**long \_\_data\_start @ r5

**.**long \_\_bss\_start @ r6

**.**long \_end @ r7

**.**long processor\_id @ r4

**.**long \_\_machine\_arch\_type @ r5

**.**long cr\_alignment @ r6

**.**long init\_thread\_union **+** THREAD\_START\_SP @ sp

/\*

\* The following fragment of code is executed with the MMU on in MMU mode,

\* and uses absolute addresses; this is not position independent.

\*

\* r0 = cp#15 control register

\* r1 = machine ID

\* r9 = processor ID

\*/

**.**type \_\_mmap\_switched**,** **%**function

\_\_mmap\_switched**:**

adr r3**,** \_\_switch\_data **+** 4 //将\_\_switch\_data + 4的物理地址赋给r3

//从r3地址读出内容

/\* r4 = \_\_data\_loc, r5 = \_data, r6 = \_bss\_start, r7 = \_end \*/

ldmia r3**!,** **{**r4**,** r5**,** r6**,** r7**}**

    /\* 下面这段代码类似于这段C代码, 即将整个数据段从\_\_data\_loc拷贝到\_data段。

    \* if (\_\_data\_loc  == \_data || \_data != \_bass\_start)

    \*    memcpy(\_data, \_\_data\_loc, \_bss\_start - \_data);

    \*/

cmp r4**,** r5 @ Copy data segment **if** needed

1**:** cmpne r5**,** r6

ldrne fp**,** **[**r4**],** #4

strne fp**,** **[**r5**],** #4

bne 1b

/\*将BSS段，也即从\_bss\_start到\_end的内存清零。\*/

mov fp**,** #0 @ Clear BSS **(and** zero fp**)**

1**:** cmp r6**,** r7

strcc fp**,** **[**r6**],**#4

bcc 1b

/\*r4 = processor\_id,

    \* r5 = \_\_machine\_arch\_type

    \* r6 = \_\_atags\_pointer

    \* r7 = cr\_alignment

    \* sp = init\_thread\_union + THREAD\_START\_SP \*/

ldmia r3**,** **{**r4**,** r5**,** r6**,** sp**}**

str r9**,** **[**r4**]** //保存处理器id到processor\_id所在的地址中

str r1**,** **[**r5**]** //保存machine id到\_\_machine\_arch\_type中

bic r4**,** r0**,** #CR\_A @ Clear 'A' bit

stmia r6**,** **{**r0**,** r4**}** @ Save control register values

b start\_kernel

最后调用start\_kernel进入下一个阶段。

## 3.linux内核启动过程

## 一些其他的基本知识：

魔数：

# arch/arm/boot/compressed/head.S

.word 0x016f2818 @ Magic numbers to help the loader

保存在zImage的36~39处，用于uboot检查。反汇编vmlinux可以得知魔数存在于0x24位置。

24: 016f2818 cmneq pc, r8, lsl r8