# 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搬运到最终的目的地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内核启动之start\_kernel

asmlinkage void \_\_init start\_kernel**(**void**)**

**{**

char **\*** command\_line**;**

extern struct kernel\_param \_\_start\_\_\_param**[],** \_\_stop\_\_\_param**[];**

//获取当前正在执行初始化的处理器ID

//这里为空

smp\_setup\_processor\_id**();**

//提供调试和错误的初始化，打印函数调用栈

//这里为空

unwind\_init**();**

//lockdep是linux内核的一个调试模块，用来检查内核互斥机制尤其是自旋锁潜在的死锁问题

//这里未打开该功能

lockdep\_init**();**

//屏蔽当前CPU上的所有中断

local\_irq\_disable**();**

//修改标记early\_boot\_irqs\_enabled;

//通过一个静态全局变量 early\_boot\_irqs\_enabled来帮助我们调试代码，

//通过这个标记可以帮助我们知道是否在”early bootup code”

early\_boot\_irqs\_off**();**

//每一个中断都有一个IRQ描述符（struct irq\_desc）来进行描述。

//这个函数的主要作用是设置所有的 IRQ描述符（struct irq\_desc）的锁是统一的锁，

//还是每一个IRQ描述符（struct irq\_desc）都有一个小锁。

early\_init\_irq\_lock\_class**();**

//大内核锁（BKL--Big Kernel Lock）

lock\_kernel**();**

//调用 clockevents\_register\_notifier 函数向 clockevents\_chain 通知链注册元素

tick\_init**();**

//用以启动的CPU进行初始化。也就是初始化CPU0

boot\_cpu\_init**();**

//初始化页地址，对于arm这里是个空函数

page\_address\_init**();**

//打印linux版本信息，这里还未初始化console，只是将打印信息存入

//到buf中，一旦console初始化，便会输出到console

printk**(**KERN\_NOTICE**);**

printk**(**linux\_banner**);**

//cpu架构相关设置

setup\_arch**(&**command\_line**);**

//保存未处理的和最原始的command\_line

setup\_command\_line**(**command\_line**);**

//这里为空

unwind\_setup**();**

//如果定义了CONFIG\_SMP宏，则这个setup\_per\_cpu\_areas()函数给每个CPU分配内存，

//并拷贝.data.percpu段的数据。

setup\_per\_cpu\_areas**();**

//如果是SMP环境，则设置boot CPU的一些数据。在引导过程中使用的CPU称为boot CPU

smp\_prepare\_boot\_cpu**();** /\* arch-specific boot-cpu hooks \*/

//核心进程调度器初始化，调度器的初始化的优先级要高于任何中断的建立，

//并且初始化进程0，即idle进程，但是并没有设置idle进程的NEED\_RESCHED标志，

//所以还会继续完成内核初始化剩下的事情。

//这里仅仅为进程调度程序的执行做准备。

//它所做的具体工作是调用init\_bh函数(kernel/softirq.c)把timer,tqueue,immediate三个任务队列加入下半部分的数组

sched\_init**();**

//抢占计数器加1

preempt\_disable**();**

//结点和内存域的初始化

//在linux系统中，内存划分结点，每个结点关联到系统中的一个处理器。

//各个结点又划分内存域，主要包括ZONE\_DMA、ZONE\_NORMAL、ZONE\_HIGHMEM。

build\_all\_zonelists**();**

//设置内存分页分配通知器

page\_alloc\_init**();**

//打印启动命令

printk**(**KERN\_NOTICE "Kernel command line: %s\n"**,** boot\_command\_line**);**

//处理early\_param定义的参数，这里并没有任何操作

parse\_early\_param**();**

//处理uboot传进来的root init console选项

parse\_args**(**"Booting kernel"**,** static\_command\_line**,** \_\_start\_\_\_param**,**

\_\_stop\_\_\_param **-** \_\_start\_\_\_param**,**

**&**unknown\_bootoption**);**

//确认中断是否关闭，若没有关闭，则再次关闭

**if** **(!**irqs\_disabled**())** **{**

printk**(**KERN\_WARNING "start\_kernel(): bug: interrupts were "

"enabled \*very\* early, fixing it\n"**);**

local\_irq\_disable**();**

**}**

//对内核异常表进行排序

sort\_main\_extable**();**

//空函数

trap\_init**();**

//初始化直接读拷贝更新的锁机制，RCU主要提供在读取数据机会比较多，

//但更新比较少的场合，这样减少读取数据锁性能低下的问题

rcu\_init**();**

//架构相关中断初始化,调用init\_arch\_irq-->s3c24xx\_init\_irq完成初始化

init\_IRQ**();**

//进程hash表的初始化

pidhash\_init**();**

//初始化引导CPU的时钟相关的数据结构

init\_timers**();**

//对高精度时钟进行初始化

hrtimers\_init**();**

//软中断初始化

softirq\_init**();**

//初始化时钟源

timekeeping\_init**();**

//初始化系统时间，

//检查系统定时器描述结构struct sys\_timer全局变量system\_timer是否为空，

//如果为空将其指向dummy\_gettimeoffset()函数。

//同时要调用system\_timer->init()-->s3c24xx\_timer.init初始化TIMER4并建立中断;

//在这里使用TIMER4作为常规的定时器

time\_init**();**

//profile只是内核的一个调试性能的工具，

//这个可以通过menuconfig中的Instrumentation Support->profile打开

profile\_init**();**

**if** **(!**irqs\_disabled**())**

printk**(**"start\_kernel(): bug: interrupts were enabled early\n"**);**

//与开始的early\_boot\_irqs\_off相对应

early\_boot\_irqs\_on**();**

//与local\_irq\_disbale相对应，开中断

local\_irq\_enable**();**

//console串口初始化

console\_init**();**

**if** **(**panic\_later**)**

panic**(**panic\_later**,** panic\_param**);**

//如果定义了CONFIG\_LOCKDEP宏，那么就打印锁依赖信息，否则什么也不做

lockdep\_info**();**

//如果定义CONFIG\_DEBUG\_LOCKING\_API\_SELFTESTS宏

//则locking\_selftest()是一个空函数，否则执行锁自测

locking\_selftest**();**

#ifdef CONFIG\_BLK\_DEV\_INITRD

**if** **(**initrd\_start **&&** **!**initrd\_below\_start\_ok **&&**

initrd\_start **<** min\_low\_pfn **<<** PAGE\_SHIFT**)** **{**

printk**(**KERN\_CRIT "initrd overwritten (0x%08lx < 0x%08lx) - "

"disabling it.\n"**,**initrd\_start**,**min\_low\_pfn **<<** PAGE\_SHIFT**);**

initrd\_start **=** 0**;**

**}**

#endif

//初始化VFS的两个重要数据结构dcache和inode的缓存。

vfs\_caches\_init\_early**();**

cpuset\_init\_early**();**

//bootmem迁移至伙伴系统

mem\_init**();**

//初始化slab分配器

kmem\_cache\_init**();**

//创建每个CPU的高速缓存集合数组

setup\_per\_cpu\_pageset**();**

//初始化NUMA的内存访问策略

numa\_policy\_init**();**

//这里为空

**if** **(**late\_time\_init**)**

late\_time\_init**();**

//计算CPU需要校准的时间

calibrate\_delay**();**

//PID分配映射初始化

pidmap\_init**();**

pgtable\_cache\_init**();**

//prio\_tree在linux内核中被应用于反向内存映射

prio\_tree\_init**();**

//匿名虚拟内存域初始化

anon\_vma\_init**();**

//进程创建机制初始化

fork\_init**(**num\_physpages**);**

//进程缓存初始化

proc\_caches\_init**();**

//文件系统的缓存区初始化

buffer\_init**();**

unnamed\_dev\_init**();**

//内核密钥管理系统初始化

key\_init**();**

//内核密钥管理系统初始化

security\_init**();**

//内核调试系统后期初始化

vfs\_caches\_init**(**num\_physpages**);**

//初始化radix树算法初始化

radix\_tree\_init**();**

//信号管理系统初始化

signals\_init**();**

//页回写机制初始化

page\_writeback\_init**();**

//proc文件系统初始化

#ifdef CONFIG\_PROC\_FS

proc\_root\_init**();**

#endif

//CPUSET初始化

cpuset\_init**();**

//任务状态早期初始化

taskstats\_init\_early**();**

//任务延迟机制初始化

delayacct\_init**();**

//检查CPU配置等是否非法使用不具备的功能

check\_bugs**();**

//初始化ACPI电源管理

acpi\_early\_init**();** /\* before LAPIC and SMP init \*/

//后继初始化，在init/main.c中实现，创建并启动内核线程

rest\_init**();**

**}**

### 3.1 setup\_processor

检查处理器的类型是否匹配，并获取处理器信息来设置处理器的相关底层参数。

static void \_\_init setup\_processor**(**void**)**

**{**

struct proc\_info\_list **\***list**;**

/\*lookup\_processor\_type定义在arch/arm/kernel/head-common.S中

\*它本身是一个汇编函数，通过一个C语言API包装而成的

\*通过该函数返回cpu信息

\*/

list **=** lookup\_processor\_type**(**processor\_id**);**

**if** **(!**list**)** **{**

printk**(**"CPU configuration botched (ID %08x), unable "

"to continue.\n"**,** processor\_id**);**

**while** **(**1**);**

**}**

/\*

\*cpu\_arm920\_name:

\*.asciz "ARM920T"

\*/

cpu\_name **=** list**->**cpu\_name**;**

/\*

\*定义和cpu相关的处理函数

\*/

#ifdef MULTI\_CPU

processor **=** **\***list**->**proc**;**

#endif

#ifdef MULTI\_TLB

cpu\_tlb **=** **\***list**->**tlb**;**

#endif

#ifdef MULTI\_USER

cpu\_user **=** **\***list**->**user**;**

#endif

#ifdef MULTI\_CACHE

cpu\_cache **=** **\***list**->**cache**;**

#endif

//打印cpu的相关信息

printk**(**"CPU: %s [%08x] revision %d (ARMv%s), cr=%08lx\n"**,**

cpu\_name**,** processor\_id**,** **(**int**)**processor\_id **&** 15**,**

proc\_arch**[**cpu\_architecture**()],** cr\_alignment**);**

//设置一些和名字相关的字符串

sprintf**(**init\_utsname**()->**machine**,** "%s%c"**,** list**->**arch\_name**,** ENDIANNESS**);**

sprintf**(**elf\_platform**,** "%s%c"**,** list**->**elf\_name**,** ENDIANNESS**);**

//针对特定的ARM核软件屏蔽一些功能

elf\_hwcap **=** list**->**elf\_hwcap**;**

#ifndef CONFIG\_ARM\_THUMB

elf\_hwcap **&=** **~**HWCAP\_THUMB**;**

#endif

/\*调用cpu\_arm920\_proc\_init函数，好像没干什么事情

\*ENTRY(cpu\_arm920\_proc\_init)

\*mov pc, lr

\*/

cpu\_proc\_init**();**

**}**

### 3.2 setup\_arch

进行与体系结构相关的第一个初始化工作；对不同的体系结构来说该函数有不同的定义，对于ARM平台而言，该函数定义在arch/arm/kernel/Setup.c。

void \_\_init setup\_arch**(**char **\*\***cmdline\_p**)**

**{**

struct tag **\***tags **=** **(**struct tag **\*)&**init\_tags**;**

struct machine\_desc **\***mdesc**;**

char **\***from **=** default\_command\_line**;**

//通过cpu id获取对应的cpu结构体并设置对应的底层函数

setup\_processor**();**

//通过machine id获取对应的machine结构体

mdesc **=** setup\_machine**(**machine\_arch\_type**);**

machine\_name **=** mdesc**->**name**;**

/\*通过struct machine\_desc中的soft\_reboot数据来设置重启类型：

\*如果存在就为“s”：softreset；如果不存在就为“h”：hardreset

\*/

**if** **(**mdesc**->**soft\_reboot**)**

reboot\_setup**(**"s"**);**

//获取tag的位置，这里为虚拟地址0xc0000100

**if** **(**mdesc**->**boot\_params**)**

tags **=** phys\_to\_virt**(**mdesc**->**boot\_params**);**

/\*首先判断是不是正确的atag格式，

\*如果是以前老版本的param\_struct格式会首先将其转换成tag格式

\*/

**if** **(**tags**->**hdr**.**tag **!=** ATAG\_CORE**)**

convert\_to\_tag\_list**(**tags**);**

//如果转换以后还是不对，则使用默认的init\_tags。

**if** **(**tags**->**hdr**.**tag **!=** ATAG\_CORE**)**

tags **=** **(**struct tag **\*)&**init\_tags**;**

/\*用内核参数列表填充meminfo，fixup函数出现在注册machine\_desc中，

\*在2440中没有定义这个函数

\*/

**if** **(**mdesc**->**fixup**)**

mdesc**->**fixup**(**mdesc**,** tags**,** **&**from**,** **&**meminfo**);**

/\*如果内存被初始化过，清除tags中关于MEM的参数，以免再次被初始化

\*否则对传递进来的tag进行分析

\*/

**if** **(**tags**->**hdr**.**tag **==** ATAG\_CORE**)** **{**

**if** **(**meminfo**.**nr\_banks **!=** 0**)**

squash\_mem\_tags**(**tags**);**

parse\_tags**(**tags**);**

**}**

/\*每一个任务都有一个mm\_struct结构管理任务内存空间，init\_mm

\*是内核的mm\_struct，其中设置成员变量\* mmap指向自己，

\*意味着内核只有一个内存管理结构，设置\* pgd=swapper\_pg\_dir，

\*swapper\_pg\_dir是内核的页目录，在arm体系结构有16k，

\*所以init\_mm定义了整个kernel的内存空间，下面我们会碰到内核

\*线程，所有的内核线程都使用内核空间，拥有和内核同样的访问

\*权限。

\*/

//内核代码段开始

init\_mm**.**start\_code **=** **(**unsigned long**)** **&**\_text**;**

//内核代码段结束

init\_mm**.**end\_code **=** **(**unsigned long**)** **&**\_etext**;**

//内核数据段开始

init\_mm**.**end\_data **=** **(**unsigned long**)** **&**\_edata**;**

//内核数据段结束

init\_mm**.**brk **=** **(**unsigned long**)** **&**\_end**;**

//from指向default\_command\_line前者已经保存了uboot传递进来的参数

memcpy**(**boot\_command\_line**,** from**,** COMMAND\_LINE\_SIZE**);**

boot\_command\_line**[**COMMAND\_LINE\_SIZE**-**1**]** **=** '\0'**;**

//分析uboot传进来的cmdline

parse\_cmdline**(**cmdline\_p**,** from**);**

//创建内核页表，映射所有的物理内存和IO空间

paging\_init**(&**meminfo**,** mdesc**);**

//完成对从内核启动到此时,cpu使用到的所有外围总线设备实体的注册登记工作

request\_standard\_resources**(&**meminfo**,** mdesc**);**

#ifdef CONFIG\_SMP

smp\_init\_cpus**();**

#endif

//对用到的IRQ, ABT, UND模式的SP都作了初始化

cpu\_init**();**

//设置machine对应的初始化函数

init\_arch\_irq **=** mdesc**->**init\_irq**;**

system\_timer **=** mdesc**->**timer**;**

init\_machine **=** mdesc**->**init\_machine**;**

#ifdef CONFIG\_VT

#if defined(CONFIG\_VGA\_CONSOLE)

conswitchp **=** **&**vga\_con**;**

#elif defined(CONFIG\_DUMMY\_CONSOLE)

conswitchp **=** **&**dummy\_con**;**

#endif

#endif

//拷贝中断向量表到0x0xFFFF\_0000

#if defined(CONFIG\_KGDB)

extern void \_\_init early\_trap\_init**(**void**);**

early\_trap\_init**();**

#endif

**}**

### 3.3 parse\_tags

分析uboot传递过来的tags参数。

//遍历tags列表调用parse\_tag进行解析

static void \_\_init parse\_tags**(**const struct tag **\***t**)**

**{**

**for** **(;** t**->**hdr**.**size**;** t **=** tag\_next**(**t**))**

**if** **(!**parse\_tag**(**t**))**

printk**(**KERN\_WARNING

"Ignoring unrecognised tag 0x%08x\n"**,**

t**->**hdr**.**tag**);**

**}**

//从tagtable里面寻找合适的解析函数

static int \_\_init parse\_tag**(**const struct tag **\***tag**)**

**{**

extern struct tagtable \_\_tagtable\_begin**,** \_\_tagtable\_end**;**

struct tagtable **\***t**;**

**for** **(**t **=** **&**\_\_tagtable\_begin**;** t **<** **&**\_\_tagtable\_end**;** t**++)**

**if** **(**tag**->**hdr**.**tag **==** t**->**tag**)** **{**

t**->**parse**(**tag**);**

**break;**

**}**

**return** t **<** **&**\_\_tagtable\_end**;**

**}**

处理tag参数的函数存放在tagtable列表里，tagtable列表存放在.taglist.init中相关定义如下。

\_\_tagtable\_begin **=** **.;**

**\*(.**taglist**.**init**)**

\_\_tagtable\_end **=** **.;**

#define \_\_tag \_\_used \_\_attribute\_\_((\_\_section\_\_(".taglist.init")))

#define \_\_tagtable(tag, fn) \

static struct tagtable \_\_tagtable\_##fn \_\_tag = { tag, fn }

调用\_\_tagtable宏进行tagtable初始化。

static int \_\_init parse\_tag\_core**(**struct tag **\***tag**)**

**{**

**if** **(**tag**->**hdr**.**size **>** 2**)** **{**

**if** **((**tag**->**u**.**core**.**flags **&** 1**)** **==** 0**)**

root\_mountflags **&=** **~**MS\_RDONLY**;**

ROOT\_DEV **=** new\_decode\_dev**(**tag**->**u**.**core**.**rootdev**);**

**}**

**return** 0**;**

**}**

\_\_tagtable**(**ATAG\_CORE**,** parse\_tag\_core**);**

static int \_\_init parse\_tag\_mem**(**struct tag **\***tag**)**

**{**

unsigned long start**,** end**;**

**if** **(**tag**->**u**.**mem\_range**.**size **==** 0**)**

**return** 0**;**

start **=** tag**->**u**.**mem\_range**.**addr**;**

end **=** tag**->**u**.**mem\_range**.**addr **+** tag**->**u**.**mem\_range**.**size **-** 1**;**

add\_physical\_memory**(**start**,** end**);**

**return** 0**;**

**}**

\_\_tagtable**(**ATAG\_MEM**,** parse\_tag\_mem**);**

static int \_\_init parse\_tag\_cmdline**(**struct tag **\***tag**)**

**{**

strlcpy**(**boot\_command\_line**,** tag**->**u**.**cmdline**.**cmdline**,** COMMAND\_LINE\_SIZE**);**

**return** 0**;**

**}**

\_\_tagtable**(**ATAG\_CMDLINE**,** parse\_tag\_cmdline**);**

### 3.4 parse\_cmdline

parse\_cmdline做了三件事，首先它解析了from所指向的完整的内核参数中关于内存的部分，其次它将没有解析的部分复制到command\_line中，最后它将start\_kernel()传进来的内核参数指针指向command\_line。这里处理的命令行参数有mem initrd ecc cachepolicy nowrite nocache这六个,当前uboot传进来的参数有root init console,所以这里并未处理任何参数。

static void \_\_init parse\_cmdline**(**char **\*\***cmdline\_p**,** char **\***from**)**

**{**

char c **=** ' '**,** **\***to **=** command\_line**;**

int len **=** 0**;**

**for** **(;;)** **{**

**if** **(**c **==** ' '**)** **{**

extern struct early\_params \_\_early\_begin**,** \_\_early\_end**;**

struct early\_params **\***p**;**

**for** **(**p **=** **&**\_\_early\_begin**;** p **<** **&**\_\_early\_end**;** p**++)** **{**

int len **=** strlen**(**p**->**arg**);**

**if** **(**memcmp**(**from**,** p**->**arg**,** len**)** **==** 0**)** **{**

**if** **(**to **!=** command\_line**)**

to **-=** 1**;**

from **+=** len**;**

p**->**fn**(&**from**);**

**while** **(\***from **!=** ' ' **&&** **\***from **!=** '\0'**)**

from**++;**

**break;**

**}**

**}**

**}**

c **=** **\***from**++;**

**if** **(!**c**)**

**break;**

**if** **(**COMMAND\_LINE\_SIZE **<=** **++**len**)**

**break;**

**\***to**++** **=** c**;**

**}**

**\***to **=** '\0'**;**

**\***cmdline\_p **=** command\_line**;**

**}**

其中\_\_early\_begin \_\_early\_end定义在vmlinux.lds中。

\_\_early\_begin **=** **.;**

**\*(.**early\_param**.**init**)**

\_\_early\_end **=** **.;**

相关初始化代码如下：

**./**arch**/**arm**/**kernel**/**setup**.**c**:**\_\_early\_param**(**"mem="**,** early\_mem**);**

**./**arch**/**arm**/**mm**/**init**.**c**:**\_\_early\_param**(**"initrd="**,** early\_initrd**);**

**./**arch**/**arm**/**mm**/**mmu**.**c**:**\_\_early\_param**(**"cachepolicy="**,** early\_cachepolicy**);**

**./**arch**/**arm**/**mm**/**mmu**.**c**:**\_\_early\_param**(**"nocache"**,** early\_nocache**);**

**./**arch**/**arm**/**mm**/**mmu**.**c**:**\_\_early\_param**(**"nowb"**,** early\_nowrite**);**

**./**arch**/**arm**/**mm**/**mmu**.**c**:**\_\_early\_param**(**"ecc="**,** early\_ecc**);**

### 3.5 paging\_init

void \_\_init paging\_init**(**struct meminfo **\***mi**,** struct machine\_desc **\***mdesc**)**

**{**

void **\***zero\_page**;**

/\*建立各种类型的页表选项。该函数是为了给mem\_types数组中的

\*各种类型的页表参数添加上我们的要求，主要是一级页表，二级页表，

\*访问权限控制等标志位

\*/

build\_mem\_type\_table**();**

/\*清除在内核代码执行到start\_kernel之前时创建的大部分临时内存页表\*/

prepare\_page\_table**(**mi**);**

/\*初始化bootmem分配信息

\*实际上是初始化引导阶段所使用的内存，当内核启动完成，

\*将使用SLUB来管理内存，届时管理方法将和boot阶段会不一样。

\*/

bootmem\_init**(**mi**);**

/\*映射中断向量表(一、二级)

\*调用mdesc->map\_io进行内存映射，

\*和2440相关的寄存器映射

\*/

devicemaps\_init**(**mdesc**);**

/\*获取中断向量的一级页表\*/

top\_pmd **=** pmd\_off\_k**(**0xffff0000**);**

/\*在低端内存中申请一页内存并清0，注意返回的是虚拟地址\*/

zero\_page **=** alloc\_bootmem\_low\_pages**(**PAGE\_SIZE**);**

memzero**(**zero\_page**,** PAGE\_SIZE**);**

/\*获取该虚拟地址对应的页描述符，注意empty\_zero\_page是全局变量\*/

empty\_zero\_page **=** virt\_to\_page**(**zero\_page**);**

/\*刷新D-CAHCE内容进RAM中\*/

flush\_dcache\_page**(**empty\_zero\_page**);**

**}**

### 3.6 request\_standard\_resources

一个独立的挂接在cpu总线上的设备单元,一般都需要一段线性的地址空间来描述设备自身, linux是怎么管理所有的这些外部"物理地址范围段",进而给用户和linux自身一个比较好的观察4G总线上挂接的一个个设备实体的简洁、统一级联视图的呢?

linux采用struct resource结构体来描述一个挂接在cpu总线上的设备实体

struct resource **{**

    resource\_size\_t start**;**

    resource\_size\_t end**;**

    const char **\***name**;**

    unsigned long flags**;**

    struct resource **\***parent**,** **\***sibling**,** **\***child**;**

**};**

resource->name描述这个设备实体的名称,这个名字开发人员可以随意起,但最好贴切;  
resource->start描述设备实体在cpu总线上的线性起始物理地址;  
resource->end描述设备实体在cpu总线上的线性结尾物理地址;  
resource->flag描述这个设备实体的一些共性和特性的标志位;

只需要了解一个设备实体的以上4项,linux就能够知晓这个挂接在cpu总线的上的设备实体的基本使用情况,也就是[resource->start,resource->end]这段物理地址现在是空闲着呢,还是被什么设备占用着呢?  
linux会坚决避免将一个已经被一个设备实体使用的总线物理地址区间段[resource->start,resource->end],再分配给另一个后来的也需要这个区间段或者区间段内部分地址的设备实体,进而避免设备之间出现对同一总线物理地址段的重复引用,而造成对唯一物理地址的设备实体二义性.  
以上的4个属性仅仅用来描述一个设备实体自身,或者是设备实体可以用来自治的单元,但是这不是linux所想的,linux需要管理4G物理总线的所有空间,所以挂接到总线上的形形色色的各种设备实体,这就需要链在一起,因此resource结构体提供了另外3个成员。

resource->parent描述管理本设备实体的父类resource指针  
resource->sibling描述比本设备实体物理地址大的下一个设备实体的resource指针[这是一个单向链表]  
resource->child描述本设备实体可能还要更详细的描述本设备实体内部的一些物理地址区段,来让linux能够更清楚本设备实体的内部的级联关系,甚至可以将本设备实体内部的具有特色的区段的给它取个名字标示一下。

通过request\_resource()函数,我们可以将打算占用[start,end]总线地址空间的设备实体,登记到供linux内核观察4G总线上设备占用情况的单向观察链表resource中,进而避免对已经被占用的总线地址空间段,再分配给另一个新来的设备实体。

static void \_\_init

request\_standard\_resources**(**struct meminfo **\***mi**,** struct machine\_desc **\***mdesc**)**

**{**

struct resource **\***res**;**

int i**;**

//内核代码段开始

kernel\_code**.**start **=** virt\_to\_phys**(&**\_text**);**

kernel\_code**.**end **=** virt\_to\_phys**(&**\_etext **-** 1**);**

//内核数据段开始

kernel\_data**.**start **=** virt\_to\_phys**(&**\_\_data\_start**);**

kernel\_data**.**end **=** virt\_to\_phys**(&**\_end **-** 1**);**

**for** **(**i **=** 0**;** i **<** mi**->**nr\_banks**;** i**++)** **{**

unsigned long virt\_start**,** virt\_end**;**

**if** **(**mi**->**bank**[**i**].**size **==** 0**)**

**continue;**

//总线上的所有memory

virt\_start **=** \_\_phys\_to\_virt**(**mi**->**bank**[**i**].**start**);**

virt\_end **=** virt\_start **+** mi**->**bank**[**i**].**size **-** 1**;**

//申请res存储空间

res **=** alloc\_bootmem\_low**(sizeof(\***res**));**

//该设备实体的名称

res**->**name **=** "System RAM"**;**

//该设备实体的名称

res**->**start **=** \_\_virt\_to\_phys**(**virt\_start**);**

res**->**end **=** \_\_virt\_to\_phys**(**virt\_end**);**

res**->**flags **=** IORESOURCE\_MEM **|** IORESOURCE\_BUSY**;**

//将该设备实体登记注册到总线空间链表

request\_resource**(&**iomem\_resource**,** res**);**

//如果kernel\_code在这个memory内,那么将kernel\_code这个比较有特性的设备实体登记注册到

//res[该memory实体]设备实体下,来详细描述res内部地址空间中一些比较有特色的设备实体

//的分布情况，进而使linux能够给用户提供一个更加详细,充实的cpu外围总线设备的级联图解情况.

**if** **(**kernel\_code**.**start **>=** res**->**start **&&**

kernel\_code**.**end **<=** res**->**end**)**

request\_resource**(**res**,** **&**kernel\_code**);**

**if** **(**kernel\_data**.**start **>=** res**->**start **&&**

kernel\_data**.**end **<=** res**->**end**)**

request\_resource**(**res**,** **&**kernel\_data**);**

**}**

//到这里所有总线上的外部内存都已经登记注册了,这些空间已经被告知linux---占用了.

//                树根

//    .............................res..........................

//    |                          |

//    |                          [..kernel\_data..]//又生出来一根枝儿

//    [..kernel\_code..]//生出来一根枝儿

**if** **(**mdesc**->**video\_start**)** **{**

video\_ram**.**start **=** mdesc**->**video\_start**;**

video\_ram**.**end **=** mdesc**->**video\_end**;**

request\_resource**(&**iomem\_resource**,** **&**video\_ram**);**

**}**

/\*

\* Some machines don't have the possibility of ever

\* possessing lp0, lp1 or lp2

\*/

**if** **(**mdesc**->**reserve\_lp0**)**

request\_resource**(&**ioport\_resource**,** **&**lp0**);**

**if** **(**mdesc**->**reserve\_lp1**)**

request\_resource**(&**ioport\_resource**,** **&**lp1**);**

**if** **(**mdesc**->**reserve\_lp2**)**

request\_resource**(&**ioport\_resource**,** **&**lp2**);**

**}**

### 3.7 cpu\_init

void cpu\_init**(**void**)**

**{**

unsigned int cpu **=** smp\_processor\_id**();**

struct stack **\***stk **=** **&**stacks**[**cpu**];**

**if** **(**cpu **>=** NR\_CPUS**)** **{**

printk**(**KERN\_CRIT "CPU%u: bad primary CPU number\n"**,** cpu**);**

BUG**();**

**}**

**if** **(**system\_state **==** SYSTEM\_BOOTING**)**

dump\_cpu\_info**(**cpu**);**

/\*

\* setup stacks for re-entrant exception handlers

\*/

\_\_asm\_\_ **(**

"msr cpsr\_c, %1\n\t" //切换到IRQ\_MODE

"add sp, %0, %2\n\t" //设置r13\_irq

"msr cpsr\_c, %3\n\t" //切换到ABT\_MODE

"add sp, %0, %4\n\t" //设置r13\_abt

"msr cpsr\_c, %5\n\t" //切换到UND\_MODE

"add sp, %0, %6\n\t" //设置r13\_und

"msr cpsr\_c, %7" //切换回SVC\_MODE

**:**

**:** "r" **(**stk**),**

"I" **(**PSR\_F\_BIT **|** PSR\_I\_BIT **|** IRQ\_MODE**),**

"I" **(**offsetof**(**struct stack**,** irq**[**0**])),**

"I" **(**PSR\_F\_BIT **|** PSR\_I\_BIT **|** ABT\_MODE**),**

"I" **(**offsetof**(**struct stack**,** abt**[**0**])),**

"I" **(**PSR\_F\_BIT **|** PSR\_I\_BIT **|** UND\_MODE**),**

"I" **(**offsetof**(**struct stack**,** und**[**0**])),**

"I" **(**PSR\_F\_BIT **|** PSR\_I\_BIT **|** SVC\_MODE**)**

**:** "r14"**);**

**}**

cpu\_init就是对用到的IRQ, ABT, UND模式的SP都作了初始化可以看到,IRQ, ABT, UND模式的堆栈大小都只有3个int,你可能会问,进入这三个模式后, 怎么保存大量的寄存器?呵呵,进入这三种模式后,linux会把模式切换到SVC\_MODE,使用的是SVC\_MODE的堆栈。

### 3.8 early\_trap\_init

这个函数把定义在 arch/arm/kernel/entry-armv.S 中的异常向量表和异常处理程序的 stub进行重定位：异常向量表拷贝到 0xFFFF\_0000，异常向量处理程序的 stub 拷贝到 0xFFFF\_0200。然后调用 modify\_domain()修改了异常向量表所占据的页面的访问权限，这使得用户态无法访问该页，只有核心态才可以访问。

void \_\_init early\_trap\_init**(**void**)**

**{**

unsigned long vectors **=** CONFIG\_VECTORS\_BASE**;**

extern char \_\_stubs\_start**[],** \_\_stubs\_end**[];**

extern char \_\_vectors\_start**[],** \_\_vectors\_end**[];**

extern char \_\_kuser\_helper\_start**[],** \_\_kuser\_helper\_end**[];**

int kuser\_sz **=** \_\_kuser\_helper\_end **-** \_\_kuser\_helper\_start**;**

//异常向量表拷贝到0x0000\_0000(或 0xFFFF\_0000)，

    //异常处理程序的stub拷贝到0x0000\_0200(或0xFFFF\_0200)

memcpy**((**void **\*)**vectors**,** \_\_vectors\_start**,** \_\_vectors\_end **-** \_\_vectors\_start**);**

memcpy**((**void **\*)**vectors **+** 0x200**,** \_\_stubs\_start**,** \_\_stubs\_end **-** \_\_stubs\_start**);**

memcpy**((**void **\*)**vectors **+** 0x1000 **-** kuser\_sz**,** \_\_kuser\_helper\_start**,** kuser\_sz**);**

//拷贝信号处理函数

memcpy**((**void **\*)**KERN\_SIGRETURN\_CODE**,** sigreturn\_codes**,**

**sizeof(**sigreturn\_codes**));**

//刷新 Cache，修改异常向量表占据的页面的访问权限

flush\_icache\_range**(**vectors**,** vectors **+** PAGE\_SIZE**);**

modify\_domain**(**DOMAIN\_USER**,** DOMAIN\_CLIENT**);**

**}**

### 3.9 parse\_early\_param

void \_\_init parse\_early\_param**(**void**)**

**{**

static \_\_initdata int done **=** 0**;**

static \_\_initdata char tmp\_cmdline**[**COMMAND\_LINE\_SIZE**];**

**if** **(**done**)**

**return;**

/\* All fall through to do\_early\_param. \*/

strlcpy**(**tmp\_cmdline**,** boot\_command\_line**,** COMMAND\_LINE\_SIZE**);**

parse\_args**(**"early options"**,** tmp\_cmdline**,** **NULL,** 0**,** do\_early\_param**);**

done **=** 1**;**

**}**

调用do\_early\_param分析从command\_line解析的参数，并和.init.setup段中的内容进行对比。

static int \_\_init do\_early\_param**(**char **\***param**,** char **\***val**)**

**{**

struct obs\_kernel\_param **\***p**;**

**for** **(**p **=** \_\_setup\_start**;** p **<** \_\_setup\_end**;** p**++)** **{**

**if** **(**p**->**early **&&** strcmp**(**param**,** p**->**str**)** **==** 0**)** **{**

**if** **(**p**->**setup\_func**(**val**)** **!=** 0**)**

printk**(**KERN\_WARNING

"Malformed early option '%s'\n"**,** param**);**

**}**

**}**

/\* We accept everything at this stage. \*/

**return** 0**;**

**}**

.init.setup定义如下

\_\_setup\_start **=** **.;**

**\*(.**init**.**setup**)**

\_\_setup\_end **=** **.;**

相关设置宏如下，请注意第四个参数early, do\_early\_param只处理early\_param声明的内容，也就是early=1。

#define \_\_setup\_param(str, unique\_id, fn, early) \

static char \_\_setup\_str\_##unique\_id[] \_\_initdata = str; \

static struct obs\_kernel\_param \_\_setup\_##unique\_id \

\_\_attribute\_used\_\_ \

\_\_attribute\_\_((\_\_section\_\_(".init.setup"))) \

\_\_attribute\_\_((aligned((sizeof(long))))) \

= { \_\_setup\_str\_##unique\_id, fn, early }

#define early\_param(str, fn) \

\_\_setup\_param(str, fn, fn, 1)

#define \_\_setup(str, fn) \

\_\_setup\_param(str, fn, fn, 0)

在2440中传进来root init console三个选项，都不在early\_param中定义。

### 3.10 unknown\_bootoption

unknown\_bootoption用于分析init root console等uboot传进来的参数。

首先将param val重新组合为param=val形式，Obsolete\_checksetup则遍历init\_setup段所有param，如有匹配，调用处理函数。  
这里需要注意的一点前面将param重新拼成了param=val形式。后面遍历匹配都是匹配的”param=”而不是“param”。  
例子：\_\_setup(“console=”, console\_setup)。

static int \_\_init unknown\_bootoption**(**char **\***param**,** char **\***val**)**

**{**

/\* Change NUL term back to "=", to make "param" the whole string. \*/

**if** **(**val**)** **{**

/\* param=val or param="val"? \*/

**if** **(**val **==** param**+**strlen**(**param**)+**1**)**

val**[-**1**]** **=** '='**;**

**else** **if** **(**val **==** param**+**strlen**(**param**)+**2**)** **{**

val**[-**2**]** **=** '='**;**

memmove**(**val**-**1**,** val**,** strlen**(**val**)+**1**);**

val**--;**

**}** **else**

BUG**();**

**}**

/\* Handle obsolete-style parameters \*/

**if** **(**obsolete\_checksetup**(**param**))**

**return** 0**;**

/\*

\* Preemptive maintenance for "why didn't my mispelled command

\* line work?"

\*/

**if** **(**strchr**(**param**,** '.'**)** **&&** **(!**val **||** strchr**(**param**,** '.'**)** **<** val**))** **{**

printk**(**KERN\_ERR "Unknown boot option `%s': ignoring\n"**,** param**);**

**return** 0**;**

**}**

**if** **(**panic\_later**)**

**return** 0**;**

**if** **(**val**)** **{**

/\* Environment option \*/

unsigned int i**;**

**for** **(**i **=** 0**;** envp\_init**[**i**];** i**++)** **{**

**if** **(**i **==** MAX\_INIT\_ENVS**)** **{**

panic\_later **=** "Too many boot env vars at `%s'"**;**

panic\_param **=** param**;**

**}**

**if** **(!**strncmp**(**param**,** envp\_init**[**i**],** val **-** param**))**

**break;**

**}**

envp\_init**[**i**]** **=** param**;**

**}** **else** **{**

/\* Command line option \*/

unsigned int i**;**

**for** **(**i **=** 0**;** argv\_init**[**i**];** i**++)** **{**

**if** **(**i **==** MAX\_INIT\_ARGS**)** **{**

panic\_later **=** "Too many boot init vars at `%s'"**;**

panic\_param **=** param**;**

**}**

**}**

argv\_init**[**i**]** **=** param**;**

**}**

**return** 0**;**

**}**

uboot传进来root init console三个选项，对应的初始化和声明如下：

static int \_\_init root\_dev\_setup**(**char **\***line**)**

**{**

strlcpy**(**saved\_root\_name**,** line**,** **sizeof(**saved\_root\_name**));**

**return** 1**;**

**}**

\_\_setup**(**"root="**,** root\_dev\_setup**);**

static int \_\_init init\_setup**(**char **\***str**)**

**{**

unsigned int i**;**

execute\_command **=** str**;**

**for** **(**i **=** 1**;** i **<** MAX\_INIT\_ARGS**;** i**++)**

argv\_init**[**i**]** **=** **NULL;**

**return** 1**;**

**}**

\_\_setup**(**"init="**,** init\_setup**);**

static int \_\_init console\_setup**(**char **\***str**)**

**{**

char name**[sizeof(**console\_cmdline**[**0**].**name**)];**

char **\***s**,** **\***options**;**

int idx**;**

/\*

\* Decode str into name, index, options.

\*/

**if** **(**str**[**0**]** **>=** '0' **&&** str**[**0**]** **<=** '9'**)** **{**

strcpy**(**name**,** "ttyS"**);**

strncpy**(**name **+** 4**,** str**,** **sizeof(**name**)** **-** 5**);**

**}** **else** **{**

strncpy**(**name**,** str**,** **sizeof(**name**)** **-** 1**);**

**}**

name**[sizeof(**name**)** **-** 1**]** **=** 0**;**

**if** **((**options **=** strchr**(**str**,** ','**))** **!=** **NULL)**

**\*(**options**++)** **=** 0**;**

#ifdef \_\_sparc\_\_

**if** **(!**strcmp**(**str**,** "ttya"**))**

strcpy**(**name**,** "ttyS0"**);**

**if** **(!**strcmp**(**str**,** "ttyb"**))**

strcpy**(**name**,** "ttyS1"**);**

#endif

**for** **(**s **=** name**;** **\***s**;** s**++)**

**if** **((\***s **>=** '0' **&&** **\***s **<=** '9'**)** **||** **\***s **==** ','**)**

**break;**

idx **=** simple\_strtoul**(**s**,** **NULL,** 10**);**

**\***s **=** 0**;**

add\_preferred\_console**(**name**,** idx**,** options**);**

**return** 1**;**

**}**

\_\_setup**(**"console="**,** console\_setup**);**

### 3.11 console\_init

控制台初始化，这里主要初始化串口。

//从\_\_con\_initcall\_start~\_\_con\_initcall\_end寻找初始化函数

void \_\_init console\_init**(**void**)**

**{**

initcall\_t **\***call**;**

/\* Setup the default TTY line discipline. \*/

**(**void**)** tty\_register\_ldisc**(**N\_TTY**,** **&**tty\_ldisc\_N\_TTY**);**

call **=** \_\_con\_initcall\_start**;**

**while** **(**call **<** \_\_con\_initcall\_end**)** **{**

**(\***call**)();**

call**++;**

**}**

**}**

\_\_con\_initcall\_start和\_\_con\_initcall\_end声明如下：

\_\_con\_initcall\_start **=** **.;**

**\*(.**con\_initcall**.**init**)**

\_\_con\_initcall\_end **=** **.;**

定义con\_initcall**.**init段宏如下：

#define console\_initcall(fn) /

static initcall\_t \_\_initcall\_##fn **/**

\_\_attribute\_used\_\_ \_\_attribute\_\_**((**\_\_section\_\_**(**".con\_initcall.init"**)))=**fn

2440中有如下定义：

console\_initcall**(**s3c24xx\_serial\_initconsole**);**

## 4.linux内核启动之rest\_init

内核创建了两个内核线程，一个是内核线程的管理者，另一个是内核初始化线程init，上面两个线程就是我们平时在Linux系统中用ps命令看到：

PID TTY TIME CMD

1 **?** 00**:**00**:**00 init

2 **?** 00**:**00**:**00 kthreadd

函数分析如下：

static void noinline \_\_init\_refok rest\_init**(**void**)**

\_\_releases**(**kernel\_lock**)**

**{**

int pid**;**

//我们必须先创建init内核线程，这样它就可以获得pid为1。

//尽管如此init线程将会挂起来等待创建kthreads线程。

//如果我们在创建kthreadd线程前调度它，就将会出现OOPS。

//创建kernel\_init内核线程，内核的1号进程！！！！！

kernel\_thread**(**kernel\_init**,** **NULL,** CLONE\_FS **|** CLONE\_SIGHAND**);**

//设定NUMA系统的内存访问策略为默认

numa\_default\_policy**();**

//创建kthreadd内核线程，它的作用是管理和调度其它内核线程。

//它循环运行一个叫做kthreadd的函数，该函数的作用是运行kthread\_create\_list全局链表中维护的内核线程。

//调用kthread\_create创建一个kthread，它会被加入到kthread\_create\_list 链表中；

//被执行过的kthread会从kthread\_create\_list链表中删除；

//且kthreadd会不断调用scheduler函数让出CPU。此线程不可关闭。

pid **=** kernel\_thread**(**kthreadd**,** **NULL,** CLONE\_FS **|** CLONE\_FILES**);**

kthreadd\_task **=** find\_task\_by\_pid**(**pid**);**

unlock\_kernel**();**

//使能抢占，但不重新调度

preempt\_enable\_no\_resched**();**

//执行调度，切换进程。

schedule**();**

//进程调度完成，回到这里，禁用抢占。

preempt\_disable**();**

//此时内核本体进入了idle状态，用循环消耗空闲的CPU时间片，

//该函数从不返回。在有其他进程需要工作的时候，该函数就会被抢占！这个函数因构架不同而异。

cpu\_idle**();**

**}**

### 4.1 kernel\_init

static int \_\_init kernel\_init**(**void **\*** unused**)**

**{**

lock\_kernel**();**

set\_cpus\_allowed**(**current**,** CPU\_MASK\_ALL**);**

init\_pid\_ns**.**child\_reaper **=** current**;**

\_\_set\_special\_pids**(**1**,** 1**);**

cad\_pid **=** task\_pid**(**current**);**

//在SMP系统做准备，激活所有CPU，并开始SMP系统的调度。

smp\_prepare\_cpus**(**max\_cpus**);**

do\_pre\_smp\_initcalls**();**

smp\_init**();**

sched\_init\_smp**();**

cpuset\_init\_smp**();**

//与构架相关的部分已经初始化完成了，do\_basic\_setup函数主要是初始化设备驱动，

//完成其他驱动程序（直接编译进内核的模块）的初始化

do\_basic\_setup**();**

//检查是否有早期用户空间的init程序。如果有，让其执行

**if** **(!**ramdisk\_execute\_command**)**

ramdisk\_execute\_command **=** "/init"**;**

//如果没有检测到/init，调用prepare\_namespace挂载根文件系统

//saved\_root\_name

**if** **(**sys\_access**((**const char \_\_user **\*)** ramdisk\_execute\_command**,** 0**)** **!=** 0**)** **{**

ramdisk\_execute\_command **=** **NULL;**

prepare\_namespace**();**

**}**

//Ok, 我们已经完成了启动初始化, and

//且我们本质上已经在运行。释放初始化用的内存（initmem）段

//并开始用户空间的程序..

//真正启动了用户空间进程init

init\_post**();**

**return** 0**;**

**}**

### 4.2 init\_post

执行execute\_command指定的init进程，也就是/linuxrc。

在该函数中会尝试打开/dev/console设备文件，并将描述符0复制给1、2，所以标准输入，标准输出，标准错误都对应同一设备。

static int noinline init\_post**(**void**)**

**{**

free\_initmem**();**

unlock\_kernel**();**

mark\_rodata\_ro**();**

system\_state **=** SYSTEM\_RUNNING**;**

numa\_default\_policy**();**

**if** **(**sys\_open**((**const char \_\_user **\*)** "/dev/console"**,** O\_RDWR**,** 0**)** **<** 0**)**

printk**(**KERN\_WARNING "Warning: unable to open an initial console.\n"**);**

**(**void**)** sys\_dup**(**0**);**

**(**void**)** sys\_dup**(**0**);**

**if** **(**ramdisk\_execute\_command**)** **{**

run\_init\_process**(**ramdisk\_execute\_command**);**

printk**(**KERN\_WARNING "Failed to execute %s\n"**,**

ramdisk\_execute\_command**);**

**}**

**if** **(**execute\_command**)** **{**

run\_init\_process**(**execute\_command**);**

printk**(**KERN\_WARNING "Failed to execute %s. Attempting "

"defaults...\n"**,** execute\_command**);**

**}**

run\_init\_process**(**"/sbin/init"**);**

run\_init\_process**(**"/etc/init"**);**

run\_init\_process**(**"/bin/init"**);**

run\_init\_process**(**"/bin/sh"**);**

panic**(**"No init found. Try passing init= option to kernel."**);**

**}**

### 4.3 do\_basic\_setup

static void \_\_init do\_basic\_setup**(**void**)**

**{**

//Workqueue 是内核里面很重要的一个机制，特别是内核驱动，

//一般的小型任务 (work) 都不会自己起一个线程来处理，

//而是扔到 Workqueue 中处理。

//Workqueue 的主要工作就是用进程上下文来处理内核中大量的小任务。

init\_workqueues**();**

//创建一个单线程工作队列khelper。

//运行的系统中只有一个，主要作用是指定用户空间的程序路径和环境变量,

//最终运行指定的user space的程序，属于关键线程，不能关闭。

usermodehelper\_init**();**

//初始化驱动模型中的各子系统，可见的现象是在/sys中出现的目录和文件

driver\_init**();**

//在proc文件系统中创建irq目录，并在其中初始化系统中所有中断对应的目录。

init\_irq\_proc**();**

//调用所有编译内核的驱动模块中的初始化函数。

//这里就是驱动程序员需要关心的步骤，其中按照各个内核模块初始化函数所自定义的启动级别（1～7），按顺序调用器初始化函数。

//对于同一级别的初始化函数，安装编译是链接的顺序调用，也就是和内核Makefile的编写有关。

//在编写内核模块的时候需要知道这方面的知识，比如你编写的模块使用的是I2C的API，

//那你的模块的初始化函数的级别必须低于I2C子系统初始化函数的级别（也就是级别数（1~7）要大于I2C子系统）。

//如果编写的模块必须和依赖的模块在同一级，那就必须注意内核Makefile的修改了。

do\_initcalls**();**

**}**

### 4.4 driver\_init

驱动模型子系统的初始化.

void \_\_init driver\_init**(**void**)**

**{**

//初始化驱动模型中的部分子系统和kobject：

//devices

//dev

//dev/block

//dev/char

devices\_init**();**

//初始化驱动模型中的bus子系统

buses\_init**();**

//初始化驱动模型中的class子系统

classes\_init**();**

//初始化驱动模型中的firmware子系统

firmware\_init**();**

//初始化驱动模型中的hypervisor子系统

hypervisor\_init**();**

//初始化驱动模型中的bus/platform子系统

platform\_bus\_init**();**

//初始化驱动模型中的devices/system子系统

system\_bus\_init**();**

//初始化驱动模型中的devices/system/cpu子系统

cpu\_dev\_init**();**

//初始化驱动模型中的devices/system/memory子系统

memory\_dev\_init**();**

attribute\_container\_init**();**

**}**

### 4.5 do\_initcalls

do\_initcall函数通过for循环，由\_\_initcall\_start开始，直到\_\_initcall\_end结束，依次调用识别到的初始化函数。而位于\_\_initcall\_start和\_\_initcall\_end之间的区域组成了.initcall.init节，其中保存了由xxx\_initcall形式的宏标记的函数地址，do\_initcall函数可以很轻松地取得函数地址并执行其指向的函数。

static void \_\_init do\_initcalls**(**void**)**

**{**

initcall\_t **\***call**;**

int count **=** preempt\_count**();**

**for** **(**call **=** \_\_initcall\_start**;** call **<** \_\_initcall\_end**;** call**++)** **{**

ktime\_t t0**,** t1**,** delta**;**

char **\***msg **=** **NULL;**

char msgbuf**[**40**];**

int result**;**

**if** **(**initcall\_debug**)** **{**

printk**(**"Calling initcall 0x%p"**,** **\***call**);**

print\_fn\_descriptor\_symbol**(**": %s()"**,**

**(**unsigned long**)** **\***call**);**

printk**(**"\n"**);**

t0 **=** ktime\_get**();**

**}**

result **=** **(\***call**)();**

**if** **(**initcall\_debug**)** **{**

t1 **=** ktime\_get**();**

delta **=** ktime\_sub**(**t1**,** t0**);**

printk**(**"initcall 0x%p"**,** **\***call**);**

print\_fn\_descriptor\_symbol**(**": %s()"**,**

**(**unsigned long**)** **\***call**);**

printk**(**" returned %d.\n"**,** result**);**

printk**(**"initcall 0x%p ran for %Ld msecs: "**,**

**\***call**,** **(**unsigned long long**)**delta**.**tv64 **>>** 20**);**

print\_fn\_descriptor\_symbol**(**"%s()\n"**,**

**(**unsigned long**)** **\***call**);**

**}**

**if** **(**result **&&** result **!=** **-**ENODEV **&&** initcall\_debug**)** **{**

sprintf**(**msgbuf**,** "error code %d"**,** result**);**

msg **=** msgbuf**;**

**}**

**if** **(**preempt\_count**()** **!=** count**)** **{**

msg **=** "preemption imbalance"**;**

preempt\_count**()** **=** count**;**

**}**

**if** **(**irqs\_disabled**())** **{**

msg **=** "disabled interrupts"**;**

local\_irq\_enable**();**

**}**

**if** **(**msg**)** **{**

printk**(**KERN\_WARNING "initcall at 0x%p"**,** **\***call**);**

print\_fn\_descriptor\_symbol**(**": %s()"**,**

**(**unsigned long**)** **\***call**);**

printk**(**": returned with %s\n"**,** msg**);**

**}**

**}**

/\* Make sure there is no pending stuff from the initcall sequence \*/

flush\_scheduled\_work**();**

**}**

\_\_initcall\_start，\_\_initcall\_end定义如下：

\_\_initcall\_start **=** **.;**

**\*(.**initcall0**.**init**)** **\*(.**initcall0s**.**init**)** **\*(.**initcall1**.**init**)** **\*(.**initcall1s**.**init**)** **\*(.**initcall2**.**init**)** **\*(.**initcall2s**.**init**)** **\*(.**initcall3**.**init**)** **\*(.**initcall3s**.**init**)** **\*(.**initcall4**.**init**)** **\*(.**initcall4s**.**init**)** **\*(.**initcall5**.**init**)** **\*(.**initcall5s**.**init**)** **\*(.**initcallrootfs**.**init**)** **\*(.**initcall6**.**init**)** **\*(.**initcall6s**.**init**)** **\*(.**initcall7**.**init**)** **\*(.**initcall7s**.**init**)**

\_\_initcall\_end **=** **.;**

initcallx**.**init段的定义如下：

#define \_\_define\_initcall(level,fn,id) \

static initcall\_t \_\_initcall\_##fn##id \_\_attribute\_used\_\_ \

\_\_attribute\_\_((\_\_section\_\_(".initcall" level ".init"))) = fn

#define pure\_initcall(fn) \_\_define\_initcall("0",fn,1)

#define core\_initcall(fn) \_\_define\_initcall("1",fn,1)

#define postcore\_initcall(fn) \_\_define\_initcall("2",fn,2)

#define arch\_initcall(fn) \_\_define\_initcall("3",fn,3)

#define subsys\_initcall(fn) \_\_define\_initcall("4",fn,4)

#define fs\_initcall(fn) \_\_define\_initcall("5",fn,5)

#define rootfs\_initcall(fn) \_\_define\_initcall("rootfs",fn,rootfs)

#define device\_initcall(fn) \_\_define\_initcall("6",fn,6)

#define late\_initcall(fn) \_\_define\_initcall("7",fn,7)

其中和machine相关的初始化如下：

static int \_\_init customize\_machine**(**void**)**

**{**

/\* customizes platform devices, or adds new ones \*/

**if** **(**init\_machine**)**

init\_machine**();**

**return** 0**;**

**}**

arch\_initcall**(**customize\_machine**);**

static void \_\_init smdk2440\_machine\_init**(**void**)**

**{**

s3c24xx\_fb\_set\_platdata**(&**smdk2440\_lcd\_cfg**);**

platform\_add\_devices**(**smdk2440\_devices**,** ARRAY\_SIZE**(**smdk2440\_devices**));**

smdk\_machine\_init**();**

**}**

static struct platform\_device **\***smdk2440\_devices**[]** \_\_initdata **=** **{**

**&**s3c\_device\_usb**,**

**&**s3c\_device\_lcd**,**

**&**s3c\_device\_wdt**,**

**&**s3c\_device\_i2c**,**

**&**s3c\_device\_iis**,**

**&**s3c2440\_device\_sdi**,**

**};**

## 5.一些其他的基本知识：

### 5.1魔数

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

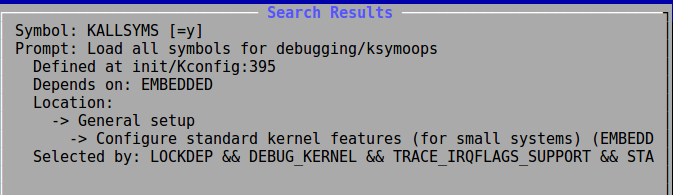
.word 0x016f2818 @ Magic numbers to help the loader

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

### 5.2 KALLSYMS

KALLSYMS开关配置



### 5.3 寄存器地址虚实映射

Linux内核访问外设I/O资源的方式有两种：动态映射和静态映射(map\_desc).

#### 5.3.1静态映射(map\_desc)方式：

静态内存映射分3个层次。  
1.开发板的层次（如：声卡，网卡和开发板相关的部分）。  
2.其他系统的层次（不影响开机的部分，如：usb，lcd，adc）。  
3.最小系统的层次（系统必需的几个，如GPIO,IRQ,MEMCTRL,UART)。

静态映射方式调用iotable\_init建立映射表。以2440为例，内存映射的入口函数如下：

static void \_\_init smdk2440\_map\_io**(**void**)**

**{**

s3c24xx\_init\_io**(**smdk2440\_iodesc**,** ARRAY\_SIZE**(**smdk2440\_iodesc**));**

s3c24xx\_init\_clocks**(**12000000**);**

s3c24xx\_init\_uarts**(**smdk2440\_uartcfgs**,** ARRAY\_SIZE**(**smdk2440\_uartcfgs**));**

**}**

s3c24xx\_init\_io中会调用如下两个函数建立映射表，在这里看到第一个iotable\_init函数了。

iotable\_init**(**s3c\_iodesc**,** ARRAY\_SIZE**(**s3c\_iodesc**));**

**(**cpu**->**map\_io**)(**mach\_desc**,** size**);**

cpu->map\_io在这里为s3c244x\_map\_io，该函数调用如下函数建立映射表。

iotable\_init**(**s3c244x\_iodesc**,** ARRAY\_SIZE**(**s3c244x\_iodesc**));**

iotable\_init**(**mach\_desc**,** size**);**

这样可以看出一共做了三次iotable\_init，对如下内容进行了映射。

tatic struct map\_desc s3c\_iodesc**[]** \_\_initdata **=** **{**

IODESC\_ENT**(**GPIO**),**

IODESC\_ENT**(**IRQ**),**

IODESC\_ENT**(**MEMCTRL**),**

IODESC\_ENT**(**UART**),**

IODESC\_ENT**(**IIS**)** // for IIS, thiswa.diy@163.com, www.100ask.net

**};**

static struct map\_desc s3c244x\_iodesc**[]** \_\_initdata **=** **{**

IODESC\_ENT**(**CLKPWR**),**

IODESC\_ENT**(**TIMER**),**

IODESC\_ENT**(**WATCHDOG**),**

IODESC\_ENT**(**LCD**),**

**};**

static struct map\_desc smdk2440\_iodesc**[]** \_\_initdata **=** **{**

/\* ISA IO Space map (memory space selected by A24) \*/

**{**

**.**virtual **=** **(**u32**)**S3C24XX\_VA\_ISA\_WORD**,**

**.**pfn **=** \_\_phys\_to\_pfn**(**S3C2410\_CS2**),**

**.**length **=** 0x10000**,**

**.**type **=** MT\_DEVICE**,**

**},** **{**

**.**virtual **=** **(**u32**)**S3C24XX\_VA\_ISA\_WORD **+** 0x10000**,**

**.**pfn **=** \_\_phys\_to\_pfn**(**S3C2410\_CS2 **+** **(**1**<<**24**)),**

**.**length **=** SZ\_4M**,**

**.**type **=** MT\_DEVICE**,**

**},** **{**

**.**virtual **=** **(**u32**)**S3C24XX\_VA\_ISA\_BYTE**,**

**.**pfn **=** \_\_phys\_to\_pfn**(**S3C2410\_CS2**),**

**.**length **=** 0x10000**,**

**.**type **=** MT\_DEVICE**,**

**},** **{**

**.**virtual **=** **(**u32**)**S3C24XX\_VA\_ISA\_BYTE **+** 0x10000**,**

**.**pfn **=** \_\_phys\_to\_pfn**(**S3C2410\_CS2 **+** **(**1**<<**24**)),**

**.**length **=** SZ\_4M**,**

**.**type **=** MT\_DEVICE**,**

**}**

**};**

其中IODESC\_ENT宏定义如下：

#define IODESC\_ENT(x) { (unsigned long)S3C24XX\_VA\_##x, \_\_phys\_to\_pfn(S3C24XX\_PA\_##x), S3C24XX\_SZ\_##x, MT\_DEVICE }

以LCD为例展开后等价为：

#define S3C2410\_ADDR(x) (0xF0000000 + (x))

#define S3C24XX\_VA\_LCD S3C2410\_ADDR(0x00300000)

#define S3C2410\_PA\_LCD (0x4D000000)

#define S3C24XX\_SZ\_LCD SZ\_1M

static struct map\_desc s3c2410\_iodesc**[]** \_\_initdata **=** **{**

**{**

**.**virtual  **=**     **(**unsigned long**)**S3C24XX\_VA\_LCD**),**

**.**pfn      **=**     \_\_phys\_to\_pfn**(**S3C24XX\_PA\_LCD**),**

**.**length   **=**     S3C24XX\_SZ\_LCD**,**

**.**type     **=**     MT\_DEVICE

**},**

     ……

**};**

virtual为需要映射的虚拟地址，pfn为物理地址。

一旦建立好映射后，可以使用虚拟地址进行操作了。

#define S3C2410\_LCDREG(x) ((x) + S3C24XX\_VA\_LCD)

#define S3C2410\_LCDCON1 S3C2410\_LCDREG(0x00)

#### 5.3.2动态映射(ioremap)方式：

动态映射通过内核提供的ioremap函数动态创建一段外设I/O内存资源到内核虚拟地址的映射表，从而可以在内核空间中访问这段I/O资源。

ioremap宏定义在asm/io.h内：

#define ioremap(cookie,size)           \_\_ioremap(cookie,size,0)

void \_\_iomem **\*** \_\_ioremap**(**unsigned long phys\_addr**,** size\_t size**,** unsigned long flags**);**

phys\_addr：要映射的起始的IO地址

size：要映射的空间的大小

flags：要映射的IO空间和权限有关的标志

该函数返回映射后的内核虚拟地址**(**3G**-**4G**).** 接着便可以通过读写该返回的内核虚拟地址去访问之这段I**/**O内存资源。

比如我们要访问s3c2440平台上的I2S寄存器, 查看datasheet 知道IIS物理地址为0x55000000，我们把它定义为宏S3C2410\_PA\_IIS，如下：

#define S3C2410\_PA\_IIS    (0x55000000)

若要在内核空间(iis驱动)中访问这段I/O寄存器(IIS)资源需要先建立到内核地址空间的映射。

our\_card**->**regs **=** ioremap**(**S3C2410\_PA\_IIS**,** 0x100**);**

**if** **(**our\_card**->**regs **==** **NULL)** **{**

         err **=** **-**ENXIO**;**

**goto** exit\_err**;**

**}**

创建好了之后，我们就可以通过readl(our\_card->regs )或writel(value, our\_card->regs)等IO接口函数去访问它。

使用完后需调用ioumap解除映射关系。  
另外在ioremap之前，一般会用request\_mem\_region函数标记该物理内存。表示这块物理内存被占用了，不能被别人使用，不能swap。  
每个驱动模块在使用io内存之前最好都要先使request\_mem\_region申请一下，这样就利用到了内核提供的锁保护机制，不用在自己的驱动中再写代码去防止已经申请到的资源被其他驱动有意无意的伤害到

### 5.4 linux内存管理的基本知识

1.Linux将物理内存完全一一映射到内核空间，这样很方便管理内存，任何页面的虚拟地址减去一个0xc0000000的偏移就可以得到物理地址。

2.内核还需要动态管理一些内存用于vmalloc或者设备临时映射等，因此不能将1G的虚拟空间完全一一映射物理内存，因此权衡了一个896M的大小，0xc0000000到0xc0000000+896M的虚拟地址空间一一映射物理内存，从0xc0000000+896M到0xffffffff的地址空间作为动态映射的需要。

3.因此早期的Linux内核为了简单起见认为只有一种方式管理物理内存，那就是一一映射，因此大于896M的物理内存将不可用。

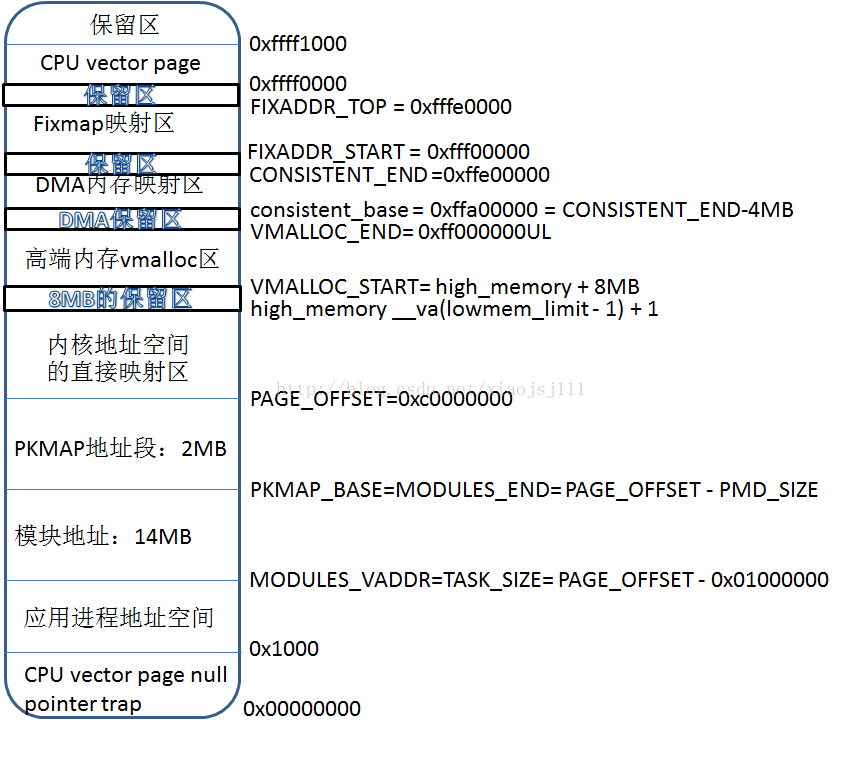
4.虽然内核无法直接一一映射896M以上的物理内存，然而内核可以动态映射它们到vmalloc空间，另外也可以将这些大于896M的内存页面分配给用户态，然后将之映射到用户进程地址空间的0xc0000000以下。

5.增加了high-memory的概念，使得Linux可以使用896之上的物理内存。所有大于896M的物理内存成为高端内存。增加了high-memory支持的内核可以使用4G的物理内存。

6.大于4G的物理内存还是无法使用，因为cpu的地址总线只有32位，地址总线直接作用于物理内存。

7. Linux的每个进程都有单独的页表，对于普通的进程来说，都有一个叫做mm\_struct的结构体，它的成员pgd会指向内存中这个进程对应的页表。内核页表其实在刚开机的时候就已经初始化好了，它存放的地址是swapper\_pg\_dir，也就是0XC0004000.这个地址里的每一项会描述1M的内存，0xc0007000到0xc0008000这段描述内核态的地址。每个用户态进程创建的时候，内核都会将这个页表复制到进程的页表中。而对于内核线程来说，由于一直工作在内核态，使用这个内核页表也就足够了。

8.虚拟内存分布



内核逻辑地址空间：是指从PAGE\_OFFSET(3G)到high\_memory(物理内存的大小，最大896)之间的线性地址空间，是系统物理内存映射区，它映射了全部或部分（如果系统包含高端内存）物理内存。内核逻辑地址空间与系统RAM内存物理地址空间是一一对应的（包括内存孔洞也是一一对应的），内核逻辑地址空间中的地址与RAM内存物理地址空间中对应的地址只差一个固定偏移量（3G），如果RAM内存物理地址空间从0x00000000地址编址，那么这个偏移量就是PAGE\_OFFSET。

9.vmalloc区用于ioremap和vmalloc操作。

### 5.5 时钟初始化

s3c2440的总线时钟是在bootloader中就设置好的。linux中只是读取了MPLL以及CLKDIVN寄存器中的值，通过这些数值来算出s3c2440当前运行的频率，而并不会对其修改！

### 5.6 文件系统挂载

为什么挂载yaffs2文件系统的时候不用指定rootfstype呢？而jffs2需要增加：rootfstype=jffs2。

FS\_REQUIRES\_DEV标志：表示该文件系统不是虚文件系统，必须要有实际的块设备。

static struct file\_system\_type jffs2\_fs\_type **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**name **=** "jffs2"**,**

**.**get\_sb **=** jffs2\_get\_sb**,**

**.**kill\_sb **=** jffs2\_kill\_sb**,**

**};**

static struct file\_system\_type yaffs2\_fs\_type **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**name **=** "yaffs2"**,**

**.**get\_sb **=** yaffs2\_read\_super**,**

**.**kill\_sb **=** kill\_block\_super**,**

**.**fs\_flags **=** FS\_REQUIRES\_DEV**,**

**};**

从代码中可以看到jffs2文件系统的结构体没有FS\_REQUIRES\_DEV标志。  
通过cat /proc/filesystems， 如果没有'nodev'这个字符串，就说明带了这个标志。

# cat /proc/filesystems

nodev sysfs

nodev rootfs

nodev bdev

nodev proc

nodev sockfs

nodev usbfs

nodev pipefs

nodev anon\_inodefs

nodev futexfs

nodev tmpfs

nodev inotifyfs

nodev devpts

ext3

ext2

cramfs

nodev ramfs

vfat

iso9660

nodev nfs

nodev nfs4

nodev jffs2

romfs

udf

yaffs

yaffs2

nodev rpc\_pipefs

内核在挂载根文件系统时，如果没有命令行参数指定根文件系统类型，会从注册的所有文件系统中挑选有这个标志的文件系统一一尝试。

### 5.7 linux内存分配器

内存分配器：

内存管理是内核是最复杂同时也是最重要的一部分，其中就涉及到了多种内存分配器，如果内核初始化阶段使用的bootmem分配器，分配大块内存的伙伴系统，以及其分配较小块内存的slab、slub和slob分配器。

(1)bootmem分配器

bootmem分配器用于在启动阶段早期分配内存。该分配器用一个位图来管理页，位图比特位的数目与系统中物理内存页的数目相同。比特位为1表示已用页，比特位为0，表示空闲页。在需要分配内存时，分配器逐位扫描位图，直至找到一个能提供足够连续页的位置，即所谓的最先最佳或最先适配位置。

该分配提供了如下内核接口：

|  |  |
| --- | --- |
| 内核接口 | 说明 |
| alloc\_bootmem  alloc\_bootmem\_pages(size) | 按指定大小在ZONE\_NORMAL内存域分配内存 |
| alloc\_bootmem\_low  alloc\_bootmem\_low\_pages(size) | 功能同上，只是从ZONE\_DMA内存域分配内存。 |
| free\_bootmem |  |

(2)slab分配器

功能：提供小的内存块，也可用作一个缓存。

slab分配器接口

|  |  |
| --- | --- |
| 接口名称 | 说明 |
| kmem\_cache\_create | 分配一个cache |
| kmem\_cache\_destroy | 销毁一个cache |
| kmem\_cache\_alloc | 从一个cache中分配一个对象空间 |
| kmem\_cache\_free | 释放一个对象空间到cache中 |

**kmalloc和vmalloc的区别**

1、kmalloc保证分配的内存在物理上是连续的,vmalloc保证的是在虚拟地址空间上的连续  
2、kmalloc能分配的大小有限,vmalloc能分配的大小相对较大·  
3、vmalloc比kmalloc要慢  
4、kmallloc使用的是slab[内存分配](https://www.baidu.com/s?wd=%E5%86%85%E5%AD%98%E5%88%86%E9%85%8D&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rHcsnAF9njuhPhfduA7W0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3Erj04n10Ln1RYPHRdPW6vrHnz)机制，而vmalloc使用的是[伙伴系统](https://www.baidu.com/s?wd=%E4%BC%99%E4%BC%B4%E7%B3%BB%E7%BB%9F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1Y3rHcsnAF9njuhPhfduA7W0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3Erj04n10Ln1RYPHRdPW6vrHnz)分配机制，这也是造成它们区别的根本所在

### 5.8 linux内存页表的创建

#### 5.8.1 内存初始化

内核中内存管理的三个阶段。

| 阶段 | 起点 | 终点 | 描述 |
| --- | --- | --- | --- |
| 第一阶段 | 系统启动 | bootmem或者memblock初始化完成 | 此阶段只能使用memblock\_reserve函数分配内存， 早期内核中使用init\_bootmem\_done = 1标识此阶段结束 |
| 第二阶段 | bootmem或者memblock初始化完 | buddy完成前 | 引导内存分配器bootmem或者memblock接受内存的管理工作, 早期内核中使用mem\_init\_done = 1标记此阶段的结束 |
| 第三阶段 | buddy初始化完成 | 系统停止运行 | 可以用cache和buddy分配内存 |

(1)bootmem分配器创建。

在启动过程中，尽管内存管理模块尚未初始化完成，但内核仍然需要分配内存以创建各种数据结构。bootmem分配器用于在启动阶段分配内存

(2)build\_all\_zonelist用于结点和内存域的初始化。

在linux系统中，内存划分结点，每个结点关联到系统中的一个处理器。各个结点又划分内存域，主要包括ZONE\_DMA、ZONE\_NORMAL、ZONE\_HIGHMEM。

内核定义了内存的一个层次结构关系，首先试图分配廉价的内存，如果失败，则根据访问速度和容量，逐渐尝试分配更昂贵的内存。   
高端内存最廉价，因为内核没有任何部分依赖于从该内存域分配的内存，如果高端内存用尽，对内核没有副作用，所以优先分配高端内存。   
普通内存域的情况有所不同，许多内核数据结构必须保存在该内存域，而不能放置到高端内存域，因此如果普通内存域用尽，那么内核会面临内存紧张的情况。   
DMA内存域最昂贵，因为它用于外设和系统之间的数据传输。

(3)mem\_init用于停用bootmem分配器并迁移到伙伴系统。

在系统初始化进行到伙伴系统分配器能够承担内存管理的责任后，必须停用bootmem分配器。该函数遍历所有的内存域结点，对每个结点分别调用free\_all\_bootmem\_node函数，停用bootmem分配器。该函数调用free\_all\_bootmem\_core，首先扫描bootmem分配器位图，释放每个未用的页。到伙伴系统的接口是\_\_free\_pages\_bootmem函数，该函数对每个空闲内存页调用，该函数内部依赖于标准函数\_\_free\_page。它使得这些页并入到伙伴系统的数据结构，在其中作为空闲页，用于分配管理。在页位图已经完全扫描后，它占据的内存空间也必须释放掉，此后，只有伙伴系统可用于内存分配。

(4)初始化slab分配器。

kmem\_cache\_init初始化内核用于小块内存区的分配器（slab分配器）。他在内核初始化阶段、伙伴系统启用之后调用。

kmem\_cache\_init创建系统中的第一个slab缓存，以便为kmem\_cache的实例提供内存，为此，内核使用一个在编译时创建的静态数据(cache\_cache)。该函数接下来初始化一般性的缓存，用作kmalloc内存的来源。为此，针对所需的各个缓存长度，分别调用kmem\_cache\_create函数。

#### 5.8.2 准备工作

Linux的内存(正式)页表是在内核代码执行到start\_kernel函数后执行paging \_init函数建立的，这里要注意一个事情就是说，这里paging\_init函数可以正常创建内存页表的条件有两个：

1.meminfo已初始化：即初始化物理内存各个node的各个bank，一般对于小型arm嵌入式设备，不涉及多个内存就是一个node和一个bank；这部分初始化是在paging\_init函数前面的对uboot所传参数的解析中完成的(可在内核的arm\_add\_memory函数中加入打印信息验证)；

2.全局变量init\_mm的代码段首尾、数据段首尾四个成员已初始化：在paging\_init前面有对这四个成员的初始化，它们规定了内核镜像的代码段起始、代码段结尾、数据段起始、数据段结尾(数据段结尾也是整个内核镜像的结尾)，给这四个成员赋的地址值都是在vmlinux.lds.S链接脚本中规定的(即虚拟地址)，界定它们的意义在于能够正确的界定内核镜像的运行需要在虚拟地址占用的空间位置及大小，以利于其他内容在内核空间位置的确定。

#### 5.8.3为物理内存创建内存映射

函数调用关系bootmem\_init-->bootmem\_init\_node-->map\_memory\_bank。

建立0xC0000000--->0x30000000的映射关系

static inline void map\_memory\_bank**(**struct membank **\***bank**)**

**{**

#ifdef CONFIG\_MMU

struct map\_desc map**;**

map**.**pfn **=** \_\_phys\_to\_pfn**(**bank**->**start**);**

map**.**virtual **=** \_\_phys\_to\_virt**(**bank**->**start**);**

map**.**length **=** bank**->**size**;**

map**.**type **=** MT\_MEMORY**;**

create\_mapping**(&**map**);**

#endif

**}**

#### 5.8.4为中断向量创建内存映射

首先调用bootmem分配器分配一页空间，和0xffff0000建立映射，该区域用于存储中断向量表

vectors **=** alloc\_bootmem\_low\_pages**(**PAGE\_SIZE**);**

map**.**pfn **=** \_\_phys\_to\_pfn**(**virt\_to\_phys**(**vectors**));**

map**.**virtual **=** 0xffff0000**;**

map**.**length **=** PAGE\_SIZE**;**

map**.**type **=** MT\_HIGH\_VECTORS**;**

create\_mapping**(&**map**);**

#### 5.8.5 为相关寄存器建立内存映射

见5.3

