基于ARM的Linux的启动分析报告

摘要：本文主要分析基于ARM的Linux－2.2.26内核启动过程。将首先从/arch/arm/Makefile着手，介绍三种不同的启动方案，再剖析典型的压缩内核zImage启动方案的代码结构，最后将详细分析这种方案的启动过程，直到调用start_kernel()为止。

1、Linux内核的启动方案：

由/arch/arm/Makefile的代码可以看出，主要有三种启动方案,分别是:

```
   echo '* zImage - Compressed kernel image (arch/$
(ARCH)/boot/zImage)'
   echo ' Image - Uncompressed kernel image (arch/$
(ARCH)/boot/Image)'
   echo ' bootpImage - Combined zImage and initial RAM disk'
   echo ' (supply initrd image via make variable INITRD=<path>)'
```

Linux内核有两种映像：一种是非压缩内核，叫 Image，另一种是它的压缩版本，叫zImage。根据内核映像的不同，Linux内核的启动在开始阶段也有所不同。zImage是Image经过压缩形成的，所以它的大小比 Image小。但为了能使用zImage，必须在它的开头加上解压缩的代码，将 zImage解压缩之后才能执行，因此它的执行速度比Image要慢。但考虑到嵌入式系统的存储空容量一般比较小，采用zImage可以占用较少的存储空间，因此牺牲一点性能上的代价也是值得的。所以一般的嵌入式系统均采用压缩内核的方式(另外bootpImage是编译包含zImage和initrd的映像，可以通过make变量INITRD=<path>提供initrd映像)。

2、基于 zImage 的启动方案。

1、 zImage 的生成过程

1、编译链接 vmlinux

2、生成 vmlinux.lds 链接脚本

3、链接生成 zImage

2、 zImage 的代码结构

在内核编译完成后会在 arch/arm/boot/下生成 zImage。

```
#arch/arm/boot/Makefile:

$(obj)/zImage:    $(obj)/compressed/vmlinux FORCE

    $(call if_changed,objcopy)

    @echo '  Kernel: $@ is ready'
```

由此可见，zImage 的是 elf 格式的，由内核顶层目录下的 arch/arm/boot /compressed/vmlinux 二进制化得到的：

```
#arch/armboot/compressed/Makefile:

$(obj)/vmlinux: $(obj)/vmlinux.lds $(obj)/$(HEAD) $(obj)/piggy.o \

       $(addprefix $(obj)/, $(OBJS)) FORCE

    $(call if_changed,ld)

    @:


$(obj)/piggy.gz: $(obj)/../Image FORCE

    $(call if_changed,gzip)


$(obj)/piggy.o:  $(obj)/piggy.gz FORCE
```

总结一下 zImage 的组成，它是由一个压缩后的内核 piggy.o，连接上一段初始化及解压功能的代码（head.o misc.o）组成的。

3、 zImage 的启动过程

1．Linux 内核的一般启动过程：

1)对于 ARM 系列处理器来说，zImage 的入口程序即为 arch/arm/boot/compressed/head.S。它依次完成以下工作：开启 MMU 和 Cache，调用 decompress_kernel()解压内核，最后通过调用 call_kernel()进入非压缩内核 Image 的启动。

Linux 非压缩内核的入口位于文件/arch/arm/kernel/head-armv.S 中的 stext 段。该段的基地址就是压缩内核解压后的跳转地址。如果系统中加载的内核是非压缩的 Image，那么 bootloader 将内核从 Flash 中拷贝到 RAM 后将直接跳到该地址处，从而启动 Linux 内核。

2）执行镜像：解压後/非压缩镜像直接执行（linux/arch/arm/kernel/head-armv.S：ENTRY(stext)-> __entry->__ret->__switch_data->__mmap_switched->）

3）该程序通过查找处理器内核类型和处理器类型调用相应的初始化函数，再建立页表，最后跳转到 start_kernel()函数开始内核的初始化工作。（linux/init/main.c：start_kernel()）

2、zImage 的启动过程

1） 内核启动地址的确定

1、#/arch/arm/Makefile 文件中，设置内核启动的虚拟地址

```
textaddr-y   := 0xC0008000     这个是内核启动的虚拟地址
TEXTADDR := $(textaddr-y)
```

2、#/arch/arm/boot/Makefile 文件中，设置内核启动的物理地址

```
ZRELADDR    := $(zreladdr-y)
PARAMS_PHYS := $(params_phys-y)
```

3、 #/arch/arm/boot/compressed/Makefile 文件中，

```
  SEDFLAGS    =
s/TEXT_START/$(ZTEXTADDR)/;s/LOAD_ADDR/$(ZRELADDR)/;s/BSS_START/$
(ZBSSADDR)/
```

使得 TEXT_START = ZTEXTADDR(从 flash 中启动时)，LOAD_ADDR = ZRELADDR

其中 TEXT_START 是内核 ram 启动的偏移地址，这个地址是物理地址

ZTEXTADDR 就是解压缩代码的 ram 偏移地址，

LOAD_ADDR 就是 zImage 中解压缩代码的 ram 偏移地址，

ZRELADDR 是内核 ram 启动的偏移地址，

zImage 的入口点由# /arch/arm/boot/compressed/vmlinux.lds.in 决定：

```
OUTPUT_ARCH(arm)
ENTRY(_start)
SECTIONS
{
  . = TEXT_START;
```

```
_text = .;


.text : {
  _start = .;
  *(.start)
  *(.text)
  ……
}
```

2) 内核解压缩过程

内核压缩和解压缩代码都在目录#/arch/arm/boot/compressed，编译完成后将产生 vmlinux、head.o、misc.o、head-xscale.o、piggy.o 这几个文件，其中

head.o：内核的头部文件，负责初始设置；

misc.o：主要负责内核的解压工作，它在 head.o 之后；

head-xscale.o：主要针对 Xscale 的初始化，将在链接时与 head.o 合并；

piggy.o：一个中间文件，其实是一个压缩的内核(kernel/vmlinux)，只不过没有和
初始化文件及解压文件链接而已；

vmlinux：没有(zImage 是压缩过的内核)压缩过的内核，就是由 piggy.o、
head.o、misc.o、head-xscale.o 组成的。

3) 在 BootLoader 完成系统的引导以后并将 Linux 内核调入内存之后，调用

bootLinux()，这个函数将跳转到 kernel 的起始位置。如果 kernel 没有压缩，就可以启动了。

如果 kernel 压缩过，则要进行解压，在压缩过的 kernel 头部有解压程序。

压缩过的kernel入口第一个文件源码位置

arch/arm/boot/compressed/head.S。

它将调用函数decompress_kernel()，这个函数在arch/arm/boot/compressed/misc.c 中，decompress_kernel()又调用

proc_decomp_setup()，arch_decomp_ setup()进行设置，然后使用在打印出信息“Uncompressing Linux...”后，调用gunzip()。将内核放于指定的位置。

4) 以下分析#/arch/arm/boot/compressed/head.S文件：

(1) 对于各种Arm CPU的DEBUG输出设定，通过定义宏来统一操作。

(2) 设置kernel开始和结束地址，保存architecture ID。

(3) 如果在ARM2以上的CPU中，用的是普通用户模式，则升到超级用户模式，然后关中断。

(4) 分析LC0结构delta offset，判断是否需要重载内核地址(r0存入偏移量，判断

r0是否为零)。

接下来要把内核镜像的相对地址转化为内存的物理地址，即重载内核地址：

(5) 需要重载内核地址，将r0的偏移量加到BSS region和GOT table中。

(6) 清空bss堆栈空间r2－r3。

(7) 建立C程序运行需要的缓存，并赋于64K的栈空间。

(8) 这时r2是缓存的结束地址，r4是kernel的最后执行地址，r5是kernel境象文件的开始地址。检查是否地址有冲突。将r5等于r2，使decompress后的kernel地址就在64K的栈之后。

(9) 调用文件 misc.c 的函数 decompress_kernel()，解压内核于缓存结束的地方(r2 地址之后)。此时各寄存器值有如下变化：

r0 为解压后 kernel 的大小

r4 为 kernel 执行时的地址

r5 为解压后 kernel 的起始地址

r6 为 CPU 类型值(processor ID)

r7 为系统类型值(architecture ID)

(10) 将 reloc_start 代码拷贝之 kernel 之后(r5+r0 之后)，首先清除缓存，而后执行 reloc_start。

(11) reloc_start 将 r5 开始的 kernel 重载于 r4 地址处。

(12) 清除 cache 内容，关闭 cache，将 r7 中 architecture ID 赋于 r1，执行 r4 开始的 kernel 代码。

5) 我们在内核启动的开始都会看到这样的输出

Uncompressing Linux...done, booting the kernel.

这也是由 decompress_kernel 函数内部输出的，它调用了 putc()输出字符串，putc 是在#/include/asm-arm/arch-pxa/uncompress.h 中实现的。执行完解压过程，再返回到#/arch/arm/boot/compressed/head.S 中，启动内核：

```
call_kernel:    bl  cache_clean_flush
        bl  cache_off
        mov r0, #0
        mov r1, r7          @ restore architecture number
```

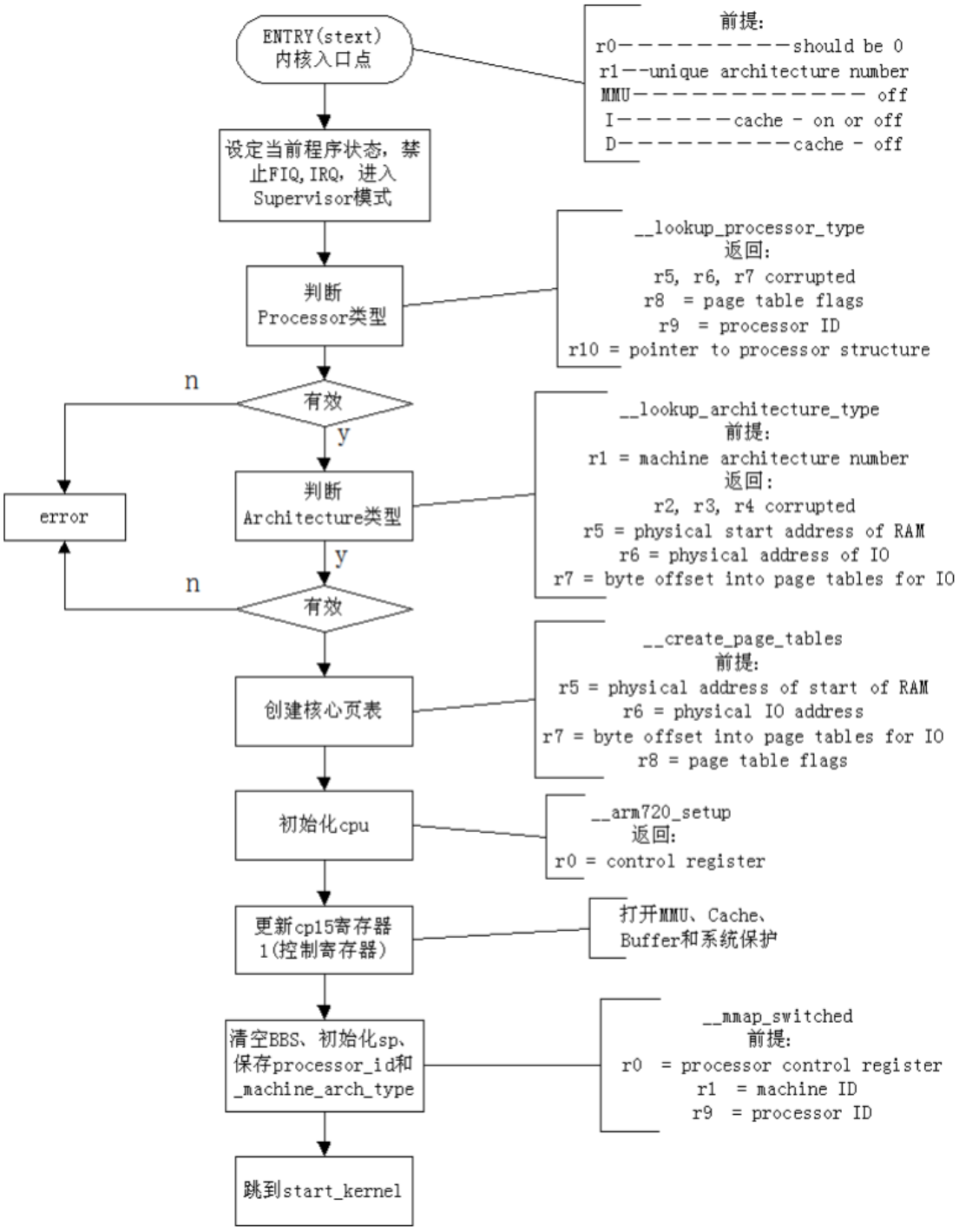
```
        mov pc, r4          @ call kernel
```

6）执行 zImage 镜像，到 start_kernel( )

整个 arm linux 内核的启动可分为三个阶段：第一阶段主要是进行 cpu 和体系结构的检查、cpu 本身的初始化以及页表的建立等；第一阶段的初始化是从内核入口（ENTRY(stext)）开始到 start_kernel 前结束。这一阶段的代码在/arch/arm/kernel/head.S 中。/arch/arm/kernel/head.S 用汇编代码完成，是内核最先执行的一个文件。这一段汇编代码的主要作用，是检查 cpu id，architecture number，初始化页表、cpu、bbs 等操作，并跳到 start_kernel 函数。它在执行前，处理器的状态应满足：

- r0 - should be 0
- r1 - unique architecture number
- MMU - off
- I-cache - on or off
- D-cache — off

a）流程图



b） 代码详细注释

```
#/arch/arm/kernel/head.S

/*
```

```
* swapper_pg_dir is the virtual address of the initial page table.
* We place the page tables 16K below KERNEL_RAM_VADDR.  Therefore, we
* must make sure that KERNEL_RAM_VADDR is correctly set.  Currently,
we *expect the least significant 16 bits to be 0x8000, but we could
probably relax this *restriction to KERNEL_RAM_VADDR >= PAGE_OFFSET +
0x4000.
*/
#if (KERNEL_RAM_VADDR & 0xffff) != 0x8000
#error KERNEL_RAM_VADDR must start at 0xXXXX8000
#endif

    .globl swapper_pg_dir
    .equ   swapper_pg_dir, KERNEL_RAM_VADDR - 0x4000

    .macro pgtbl, rd
    ldr \rd, =(KERNEL_RAM_PADDR - 0x4000)
    .endm
```

```
/*
 * Since the page table is closely related to the kernel start
address, we
 * can convert the page table base address to the base address of the
section
```

```
 * containing both.
 */
      .macro   krnladr, rd, pgtable, rambase
      bic   \rd, \pgtable, #0x000ff000
      .endm
/*
```

```
 /*
 * Kernel startup entry point.
 * ---------------------------
 *
 * This is normally called from the decompressor code.  The
requirements
 * are: MMU = off, D-cache = off, I-cache = dont care, r0 = 0,
 * r1 = machine nr, r2 = atags pointer.
 *
* See linux/arch/arm/tools/mach-types for the complete list of
machine
* numbers for r1.
*/
   .section ".text.head", "ax"
```

```
    .type   stext, %function

   ENTRY(stext)                          //内核入口点

   msr cpsr_c, #PSR_F_BIT | PSR_I_BIT | SVC_MODE

    //程序状态，禁止 FIQ、IRQ，设定 Supervisor 模式。0b11010011

    mrc p15, 0, r9, c0, c0                @ get processor id

    bl  __lookup_processor_type      @ r5=procinfo  r9=cupid  //跳转到
判断

   //cpu 类型，查找运行的 cpu 的 id 值和此 linux 编译支持的 id 值是否有相
等

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

    beq __error_p                   @ yes, error 'p'

    bl  __lookup_machine_type         @ r5=machinfo

   //跳转到判断体系类型，看 r1 寄存器的 architecture number 值是否支持。

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

    beq __error_a                    @ yes, error 'a'

    bl  __vet_atags

    bl  __create_page_tables         //创建核心页表
```

```
/*

* The following calls CPU specific code in a position independent
```

```
* manner.  See arch/arm/mm/proc-*.S for details.  r10 = base of
* xxx_proc_info structure selected by __lookup_machine_type
* above.  On return, the CPU will be ready for the MMU to be
* turned on, and r0 will hold the CPU control register value.
*/
    ldr r13, __switch_data      @ address to jump to after
                                @ mmu has been enabled
    adr lr, __enable_mmu        @ return (PIC) address //lr=0xc0028054
    add pc, r10, #PROCINFO_INITFUNC
              @ initialise processor  //r10: pointer to processor
structure
```

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

```
*/
#define ATAG_CORE 0x54410001
#define ATAG_CORE_SIZE ((2*4 + 3*4) >> 2)

    .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   __atags_pointer           @ r6
    .long   cr_alignment          @ r7
    .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
 *  r2  = atags pointer
 *  r9  = processor ID
 */
    .type   __mmap_switched, %function
__mmap_switched:
//把 sp 指针指向 init_task_union+8192（include/linux/sched.h）处，即第
//一个进程的 task_struct 和系统堆栈的地址；清空 BSS 段；保存 processor
ID
//和 machine type 到全局变量 processor_id 和__machine_arch_type，这些值
//以后要用到；r0 为"A"置位的 control register 值，r2 为"A"清空的
```

```
//control register 值，即对齐检查（Alignment fault checking）位，并保
//存到cr_alignment，和cr_no_alignment（在文件entry-armv.S中）。最
//后跳转到start_kernel（init/main.c）
    adr r3, __switch_data + 4

    ldmia  r3!, {r4, r5, r6, r7} @ r2 = compat//r2=0xc0000000
    cmp   r4, r5 @ Copy data segment if needed  //r4=0xc00c04e0;
__bss_start
1:  cmpne  r5, r6     //r5=0xc00e02a8; _end  //r6=0xc00c0934;
processor_id
    ldrne  fp, [r4], #4         //r7=0xc00c0930; __machine_arch_type
    strne  fp, [r5], #4         //r8=0xc00bcb88; cr_alignment
    bne   1b                  //sp=0xc00bc000;(init_task_union)+8192

    mov fp, #0            @ Clear BSS (and zero fp)
1:  cmp r6, r7
    strcc  fp, [r6],#4
    bcc 1b

    ldmia  r3, {r4, r5, r6, r7, sp}
    str r9, [r4]          @ Save processor ID
    str r1, [r5]          @ Save machine type
    str r2, [r6]          @ Save atags pointer
```

```
   bic r4, r0, #CR_A        @ Clear 'A' bit
   stmia  r7, {r0, r4}         @ Save control register values
   b   start_kernel            //下面就开始真正的内核了：）
```

```
/*
 * Enable the MMU.  This completely changes the structure of the
visible
 * memory space.  You will not be able to trace execution through
this.
 * If you have an enquiry about this, *please* check the linux-arm-
kernel
 * mailing list archives BEFORE sending another post to the list.
 *
 *  r0  = cp#15 control register
 *  r13 = *virtual* address to jump to upon completion
 *
 * other registers depend on the function called upon completion
 */
   .align 5
   .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, r13
```

```
/*
 * Setup the initial page tables.  We only setup the barest
 * amount which are required to get the kernel running, which
 * generally means mapping in the kernel code.
 *
 * r8  = machinfo
 * r9  = cpuid
 * r10 = procinfo
 *
 * Returns:
 *  r0, r3, r6, r7 corrupted
 *  r4 = physical page table address
 */
   .type  __create_page_tables, %function
__create_page_tables:
   pgtbl  r4            @ page table address
```

```
                        //调用宏 pgtbl，r4＝0xc0024000：页表基址
    /*
     * Clear the 16K level 1 swapper page table
     */
    mov r0, r4              //r0＝0xc0024000
    mov r3, #0
    add r6, r0, #0x4000       //r6=0xc0028000
1:  str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
    str r3, [r0], #4
    teq r0, r6
    bne 1b                  //将地址 0xc0024000～0xc0028000 清 0

    ldr r7, [r10, #PROCINFO_MM_MMUFLAGS] @ mm_mmuflags

    /*
     * Create identity mapping for first MB of kernel to
     * cater for the MMU enable.  This identity mapping
     * will be removed by paging_init().  We use our current program
     * counter to determine corresponding section base address.
     */
    mov r6, pc, lsr #20   @ start of kernel section
```

```
    orr r3, r7, r6, lsl #20      @ flags + kernel base
    str r3, [r4, r6, lsl #2]     @ identity mapping

    /*
     * Now setup the pagetables for our kernel direct
     * mapped region.
     */
    add r0, r4,  #(KERNEL_START & 0xff000000) >> 18
    str r3, [r0, #(KERNEL_START & 0x00f00000) >> 18]!
    ldr r6, =(KERNEL_END - 1)
    add r0, r0, #4
    add r6, r4, r6, lsr #18
1:  cmp r0, r6
    add r3, r3, #1 << 20
    strls  r3, [r0], #4
    bls 1b
```

```
/*
 * Read processor ID register (CP#15, CR0), and look up in the
linker-built
 * supported processor list.  Note that we can't use the absolute
addresses
 * for the __proc_info lists since we aren't running with the MMU on
```

```
 * (and therefore, we are not in the correct address space).  We have
to
 * calculate the offset.
 *
 *  r9 = cpuid
 * Returns:
 *  r3, r4, r6 corrupted
 *  r5 = proc_info pointer in physical address space
 *  r9 = cpuid (preserved)
 */
    .type   __lookup_processor_type, %function
__lookup_processor_type:   //判断cpu类型
    adr r3, 3f              //取标号3的地址
    ldmda   r3, {r5 - r7}
    sub r3, r3, r7        @ get offset between virt&phys
    add r5, r5, r3        @ convert virt addresses to
    add r6, r6, r3        @ physical address space
1:  ldmia   r5, {r3, r4}  @ value, mask       //读取arm linux中cpu信
息
    and r4, r4, r9        @ mask wanted bits  //屏蔽cpu id的低8位
    teq r3, r4              //寄存器0的cpu id与arm linux中cpu id比较
    beq 2f
```

```
   add r5, r5, #PROC_INFO_SZ     @ sizeof(proc_info_list) //否则寻找
下一块
                                                    //proc_info
   cmp r5, r6
   blo 1b
   mov r5, #0         @ unknown processor  //没有匹配信息，r5＝0
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
```

```
/*
 * Lookup machine architecture in the linker-build list of
architectures.
 * Note that we can't use the absolute addresses for the __arch_info
 * lists since we aren't running with the MMU on (and therefore, we
are
 * not in the correct address space).  We have to calculate the
offset.
 *
 *  r1 = machine architecture number
 * Returns:
 *  r3, r4, r6 corrupted
 *  r5 = mach_info pointer in physical address space
 */
   .type  __lookup_machine_type, %function
__lookup_machine_type:       //判断体系类型
   adr r3, 3b              //取上面标号 2 的地址
```

```
    ldmia   r3, {r4, r5, r6}
    sub r3, r3, r4          @ get offset between virt&phys
    add r5, r5, r3          @ convert virt addresses to
    add r6, r6, r3          @ physical address space
1:  ldr r3, [r5, #MACHINFO_TYPE] @ get machine type
    teq r3, r1              @ matches loader number?
    beq 2f              @ found
    add r5, r5, #SIZEOF_MACHINE_DESC    @ 不匹配，查找 next
machine_desc
    cmp r5, r6
    blo 1b
    mov r5, #0              @ unknown machine
2:  mov pc, lr

/*
 * This provides a C-API version of the above function.
 */
ENTRY(lookup_machine_type)
    stmfd   sp!, {r4 - r6, lr}
    mov r1, r0
    bl  __lookup_machine_type
    mov r0, r5
```

```
ldmfd  sp!, {r4 - r6, pc}
```

因水平有限，如有不妥之处或者更好的方法，敬请指出。hdm125@126.com

dianming_hu

2009-3-11