**ARCH32-Cache-Tlb-Memlayout技术研究**

**说明：本文参考的平台为ARMV7 + LINUX4.4**

目录

[2 cache 3](#_Toc523941044)

[3 Translation tables 3](#_Toc523941045)

[3.1 转换条目格式 3](#_Toc523941046)

[3.1.1 Level1描述符 3](#_Toc523941047)

[3.1.2 Level2描述符 4](#_Toc523941048)

[3.2 寄存器 4](#_Toc523941049)

[3.3 Translation table walks 5](#_Toc523941050)

[3.3.1 读取一级表 5](#_Toc523941051)

[3.3.2 读取二级表 5](#_Toc523941052)

[3.3.3 4K页面的转化流程 6](#_Toc523941053)

[3.4 更改转换表的属性 6](#_Toc523941054)

[4 TLB 6](#_Toc523941055)

[4.1 Global and non-global 6](#_Toc523941056)

[4.2 TLB matching 7](#_Toc523941057)

[4.3 TLB maintenance 7](#_Toc523941058)

[5 fcse 7](#_Toc523941059)

[6 memlayout 7](#_Toc523941060)

[6.1 基础知识介绍 7](#_Toc523941061)

[6.2 \_\_create\_page\_tables 7](#_Toc523941062)

[6.3 \_\_v7\_ca7mp\_setup 10](#_Toc523941063)

[6.4 paging\_init 10](#_Toc523941064)

[6.4.1 build\_mem\_type\_table 10](#_Toc523941065)

[6.4.2 prepare\_page\_table 10](#_Toc523941066)

[6.4.3 map\_lowmem 10](#_Toc523941067)

[6.4.4 dma\_contiguous\_remap 12](#_Toc523941068)

[6.4.5 early\_fixmap\_shutdown 12](#_Toc523941069)

[6.4.6 devicemaps\_init 12](#_Toc523941070)

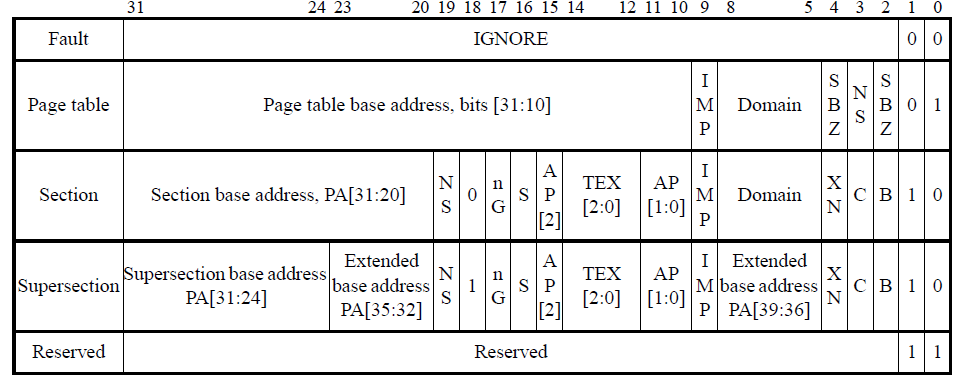
# cache

# Translation tables

## 转换条目格式

### Level1描述符

一级描述符中的每个条目描述了1M的内存空间，描述符[1:0]位的标志一级描述符的类型:



**图：VMSAv7 first-level descriptor formats**

0b00：fault格式，表示该范围的VA没有映射到PA。

0b01：页描述符。描述符给出了二级转换表的物理地址，该表指定了相关的1MByte MVA范围的映射方式。 二级转换表需要1KB内存，可以映射大页面和小页面。

0b10：段或supersection描述符。Bit[18]决定了该描述符是段还是supersection描述符。

0b11：无效。

一级页表项中部分bit的意义：

**TEX[2:0], C, B**：内存区域的属性。

**XN：**确定该区域的文件是否具有可执行权限。

**NS：**Non-secure 位。

**Domain：**

**AP[2], AP[1:0]：**访问权限位。

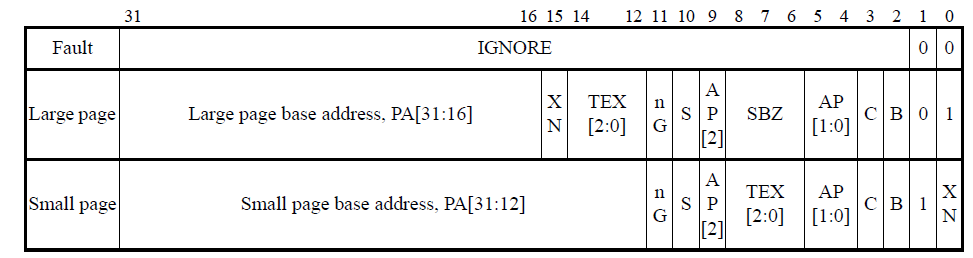
**S：**确定该片内存是否可以共享。

**nG：**确定该片区域属于global或者non-global。四三三

**Bit[18]，bits[1:0]：**0，描述符为段（section）；1，描述符为supersection

### Level2描述符

同理，二级描述符bits[1:0]标志了描述符的类型。



**图：second-level descriptor formats**

0b00：fault格式，表示该范围的VA没有映射到PA。

0b01：大页描述符。

0b1x：小页描述符。

二级页表项中部分bit的意义：

**XN：**确定该片内存是否可以执行。

**TEX[2:0], C, B：**内存范围属性位。

**AP[2], AP[1:0]：**权限访问位。

**S：**共享位，确定该片内存区域是否可以共享。

**nG：**确定该片内存是否为global属性。

## 寄存器

（1）TTBCR作用： 在TLB中未找到MVA时，TTBCR确定使用哪个转换表基址寄存器（TTBR0或TTBR1）查找转换的基地址。

TTBCR.N == 0，选择TTBR0

TTBCR.N > 0，选择TTBR1，若MVA中的[31:32-N]位均为0，则使用TTBR0，否则使用TTBR1。

（2）TTBR0作用：存放页表的基地址，根据TTBCR.N值的不同，表的范围是128byte至16Kbytes。

系统中各线程维护一张first-level表，上下文切换时：

* TTBR0保存了下个process的first-level转换表。
* 如果更改了转换表的大小，则TTBCR需要更新。
* 更新CONTEXTIDR

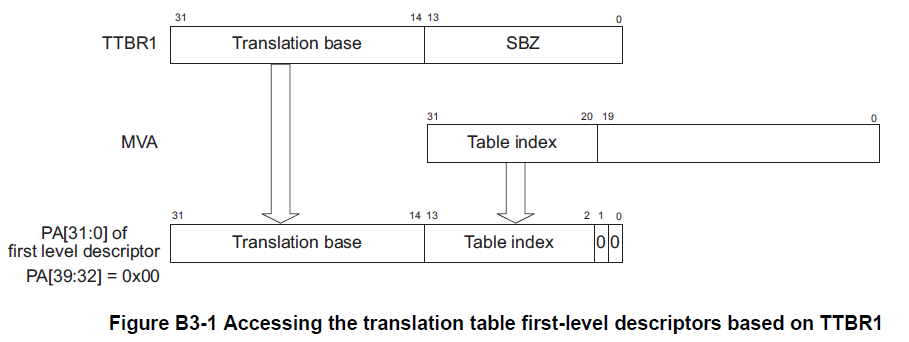
（3）TTBR1作用

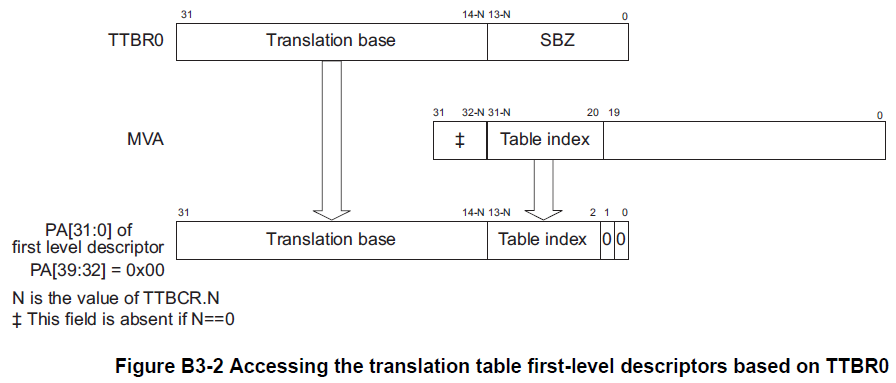
## Translation table walks

当TLB miss发生时，系统发起一次Translation table walk。

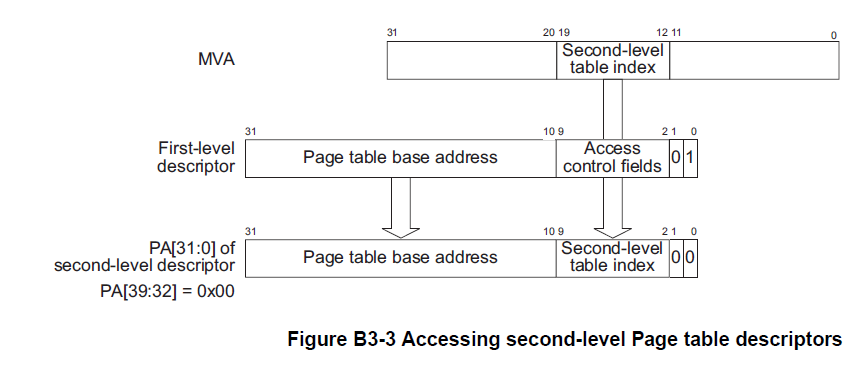
* section-mapped（段映射）只需访问一级页表。
* page-mapped（页映射）需要访问一级和二级页表。

### 读取一级表

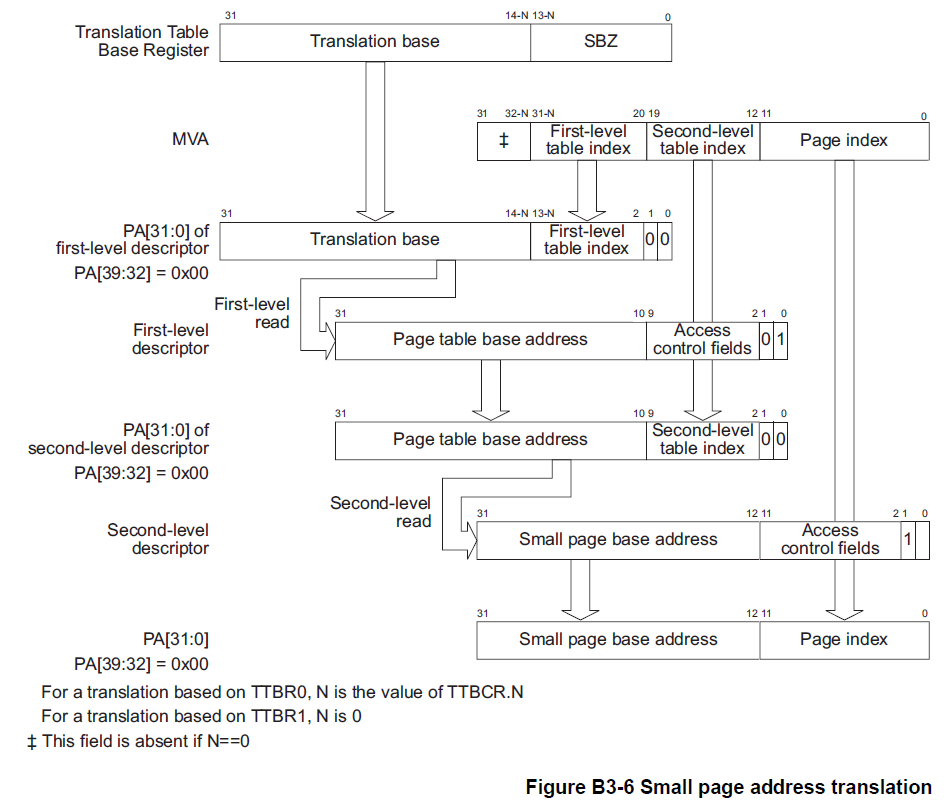




### 读取二级表



### 4K页面的转化流程



## 更改转换表的属性

# TLB

TLB是对内存页表项的缓存，避免系统每次进行VA-PA时都需要经过translation table walk。

## Global and non-global

nG == 0：转换是全局的

Ng == 1：转换是进程独占的，跟**contexidr**寄存器定义的当前的ASID（*Address Space Identifier*）相关。

**ASID**：地址空间标识符（ASID）标识与特定进程关联的页面，并提供用于更改进程特定表的机制，而无需对TLB结构执行维护。

**CONTEXTIDR寄存器**：Context ID Register，标志两部分内容：

Bits[31：8]：current Process Identifier (PROCID)

Bits[7：0]：current Address Space Identifier (ASID).

## TLB matching

## TLB maintenance

# fcse

# memlayout

## 基础知识介绍

（1）由于CPU要开启MMU进入虚地址执行模式，因此必须先通过\_\_create\_page\_tables建立一个临时的page table(将来这个table会被抛弃，重新建立)。

（2）函数中出现的宏及其解释

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | 宏 | | |  | | --- | | 默认值 | | |  | | --- | | 定义 | |
| KERNEL\_RAM\_VADDR | 0xC0008000 | 内核在内存中的虚拟地址 |
| PAGE\_OFFSET | 0xC0000000 | 内核虚拟地址空间的起始地址 |
| TEXT\_OFFSET | 0x00008000 | 内核起始位置相对于内存起始位置的偏移 |
| PHYS\_OFFSET | 构架相关 | 物理内存的起始地址（sunxi：0x40000000） |
| PG\_DIR\_SIZE | 0x4000 | 一级页表的大小 |
| SECTION\_SHIFT | 20 | 一级页表段内偏移 |

（3）小记

（1）1f代码forward的意思 1b表示before

（2）PGD: Page Global Directory (页目录)

PMD: Page Middle Directory (页目录)

PTE: Page Table Entry (页表项)

## \_\_create\_page\_tables

（1）进入函数

r8 = phys\_offset, r9 = cpuid, r10 = procinfo





R4 = phys\_offset（0x40000000）+ TEXT\_OFFSET（0x8000）= 0x40008000，内核代码段的起始物理地址；

R4 = 0x40008000 - PG\_DIR\_SIZE（0x4000）= 0x30004000

运行完后内存布局图：



（2）清理页表空间地址（0x40004000--0x40008000）



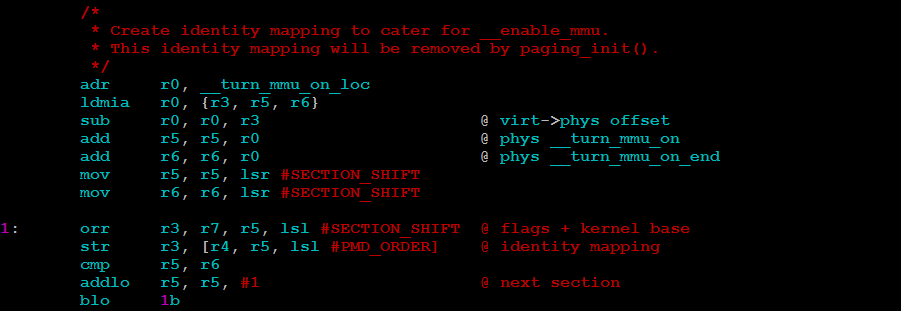
清理内存地址0x40004000---0x40008000地址段的空间，地址的高12位虚拟地址为页表索引，共4k个页表项，每一个页表项映射1MB虚拟地址，每个页表项占4个字节，页表的空间大小为为4K\*4 = 16K。

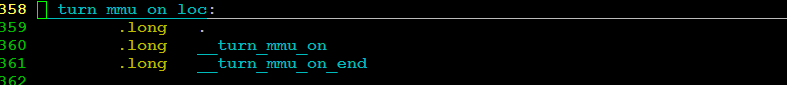
（3）获取mm\_mmuflags



R10寄存器保存的指针指向是我们前面找到的proc\_info\_list结构，将mm\_mmuflags标志取出存于r7中，也就是页表项的标志位，可参考一、二级页表的描述符。

（4）创建\_\_enable\_mmu映射





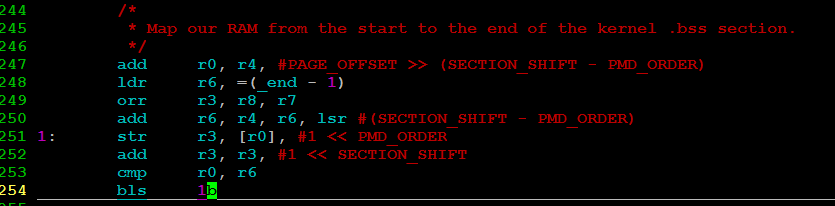
* \_\_turn\_mmu\_on\_loc为\_\_enabel\_mmu函数的地址，用adr指令赋值给r0, SECTION\_SHIFT页内偏移（值为20）
* 执行上述代码后r5，r6分别保存了 \_\_turn\_mmu\_on和\_\_turn\_mmu\_on\_end的高12bit，也就是页内偏移地址。

结合mmu\_flags和r5/r6的值，生成一级页表项，保存在R3中，并且把R3的值存入物理内存中[R4 + R5 << 20] = R3，此时R4的地址为0x30004000，这个页表映射的是\_\_enable\_mmu的物理地址。

运行完后内存布局图：



（5）映射kernelss

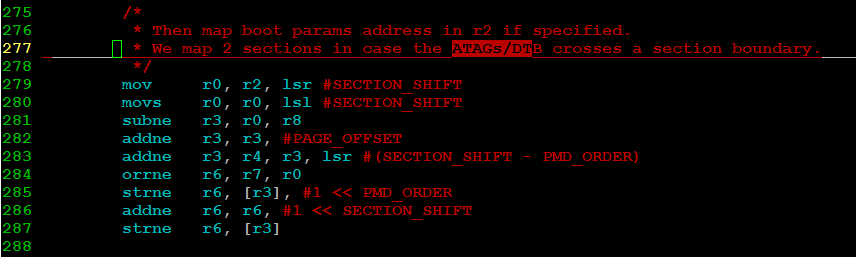


* R0 = R4 + #PAGE\_OFFSET >> (SECTION\_SHIFT - PMD\_ORDER)，PAGE\_OFFSET为0xc0000000，R4 = 0x40004000，因此R0 = 0x40004000 + 0xc0000000>>18 = 0x40007000
* R6 = 内核地址的末尾
* R8保存的是phys\_offset（0x40000000），R7保存的是mmu\_flags值，R3 = phys\_offset | mmu\_flags
* 将r3的值保存如物理地址，[r0] = r3，按照1M的映射关系，递归写入物理内存

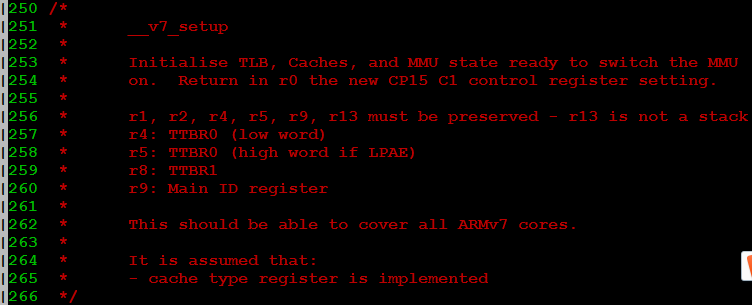
运行完后内存布局图：



（6）映射ATAGs/DT



## \_\_v7\_ca7mp\_setup



上图内核解释了该函数的作用：初始化TLB、Caches、MMU，为开启MMU做准备

## paging\_init

### build\_mem\_type\_table

作用：根据CPU类型，设置mem\_types全局数组，mem\_types数组保存了页目录和页表的属性，将来创建页目录和页表时，会用到mem\_types。

### prepare\_page\_table

请空页目录，有两块地址空间区域是不需要清除的，一个是kernel image，另外一个是kernel线性地址映射区。

### map\_lowmem

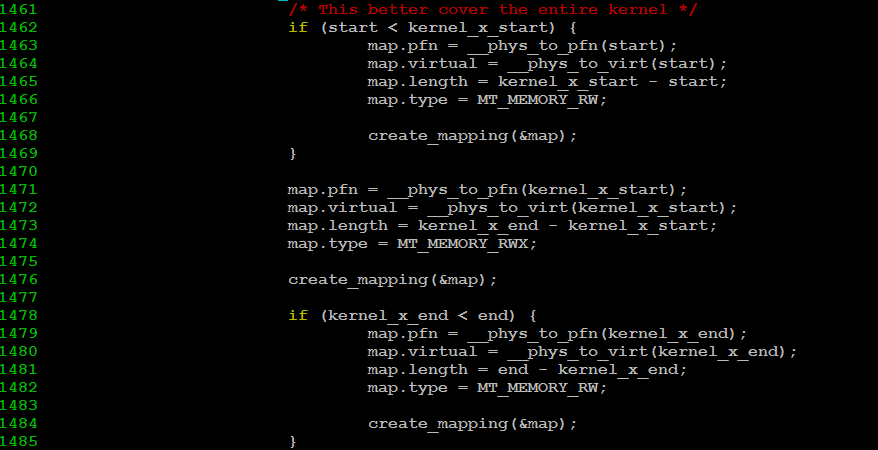
建立低端内存的所有页目录和页表：遍历memory bank，映射那些没有highmem标记的内存bank。

分析依据平台SUNXI AARCH32平台，物理内存大小为1G，物理地址为0x40000000---0x80000000。

（1）变量的意义

* kernel\_x\_start：phys\_addr\_t kernel\_x\_start = round\_down(\_\_pa(\_sdata), SECTION\_SIZE);根据分析值0x40100000，为内核的起始物理地址。（至于为什么从偏移1M的地方开始，需要追下lds文件）
* kernel\_x\_end：phys\_addr\_t kernel\_x\_end = round\_up(\_\_pa(\_\_init\_end), SECTION\_SIZE); 根据分析值为0x40800000，为内核的结束物理地址
* start：phys\_addr\_t start = reg->base; SUNX平台值为0x40000000，为物理内存起始地址。
* end ：phys\_addr\_t end = start + reg->size; if (end > arm\_lowmem\_limit) end = arm\_lowmem\_limit; end值为低端内存区域的最大值，值为0x70000000 （768M）

（2）分块映射

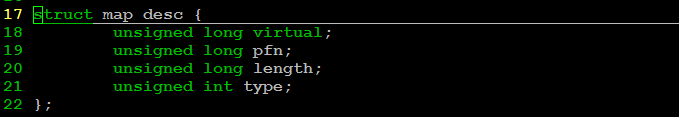


上述代码为map\_lowmem的主要映射部分代码，分为3部分通过函数create\_mapping进行映射。

第一段：0x40000000 --------0x40100000，类型为MT\_MEMORY\_RW

第二段：0x40100000---------0x40800000，类型为MT\_MEMORY\_RWX

第三段：0x40800000---------0x70000000，类型为MT\_MEMORY\_RW



pfn：页帧号

（3）create\_mapping

* 通过pgd\_offset\_k获取页面目录项PGD，从下面的推论可以看到页表的基地址为0xc0004000。

#define pgd\_offset\_k(addr) pgd\_offset(&init\_mm, addr)

#define pgd\_offset(mm, addr) ((mm)->pgd + pgd\_index(addr))

#define pgd\_index(addr) ((addr) >> PGDIR\_SHIFT)

#define PMD\_SHIFT 21

#define PGDIR\_SHIFT 21

上述代码可以推出 pgd\_offset\_k(addr)等价于 init\_mm-> pgd + (addr >> PGDIR\_SHIFT)

init\_mm-> pgd值大小为swapper\_pg\_dir，该值得定义如下

.globl swapper\_pg\_dir

.equ swapper\_pg\_dir, KERNEL\_RAM\_VADDR - PG\_DIR\_SIZE

#define KERNEL\_RAM\_VADDR (PAGE\_OFFSET + TEXT\_OFFSET)

#define PG\_DIR\_SIZE 0x4000

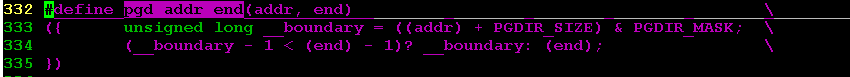
swapper\_pg\_dir = 0xc0004000

pgd\_offset\_k(addr) = swapper\_pg\_dir + ((addr) >> PGDIR\_SHIFT)

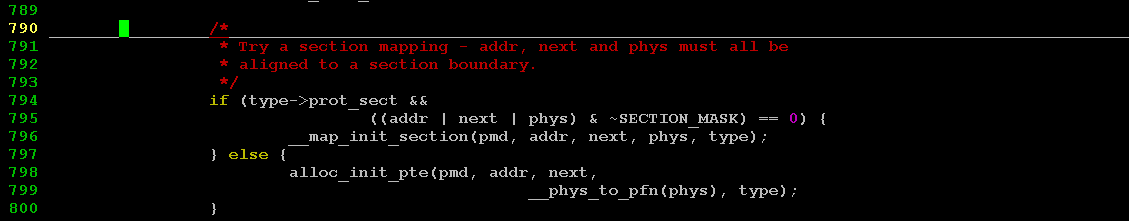
= 0xc0004000 + addr >> 21

* pgd\_addr\_end宏获取本次映射的结束位置，其中PGDIR\_SIZE=(1UL << PGDIR\_SHIFT)=2M，一次处理2个连续的一级表项:2\*1M = 2M。#define PGDIR\_SHIFT 21

代码如下：

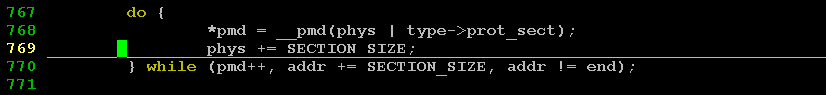


* alloc\_init\_pmds



由于当前平台使用的内存大小为1G，Map\_lowmem阶段的内存映射分为3块（详情将上（2）描述），类型为类型为MT\_MEMORY\_RW或类型为MT\_MEMORY\_RWX，都包含prot\_sect属性（section映射），并且三块内存区域是1M对齐的，因此根据上图代码的条件，map\_lowmem应该使用段映射，运行 \_\_map\_init\_section函数。

* \_\_map\_init\_section



section映射函数的主要部分，主要功能是对\*pmd进行赋值。

### dma\_contiguous\_remap

1. CMA区域位于高端内存

if (end > arm\_lowmem\_limit)

end = arm\_lowmem\_limit;

if (start >= end)

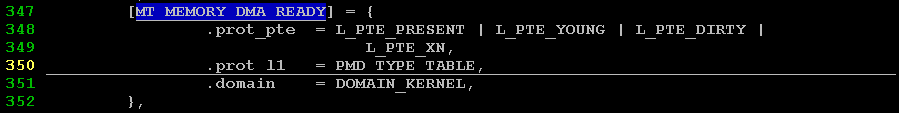
continue;

根据代码条件，当cma申请的内存位于高端内存时，推出映射。

1. CMA区域位于低端内存

当前验证平台内存大小为1G，设置CMA为512M。根据调试，确认系统分配CMA区域为start:50000000----end:70000000。

* 填充map\_desc结构体，重点关注下type为T\_MEMORY\_DMA\_READY
* 调用pmd\_clear清空之前低端内存映射的该片区一级页表项（map\_lowmem中申请的是section页表项），并且刷TLB，防止无效页表
* alloc\_init\_pte



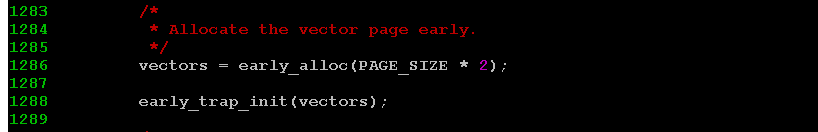
由于CMA预留内存type为T\_MEMORY\_DMA\_READY如上图，因此create\_mapping最后调用alloc\_init\_pte初始化页表。

在原来的section页表区域设置1级页表，另外申请空间生成2级页表。

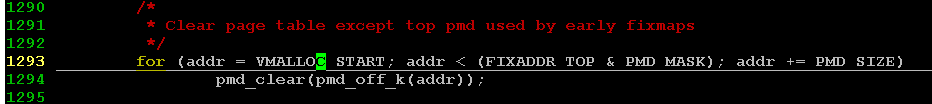
### early\_fixmap\_shutdown

### devicemaps\_init

（1）初始化异常向量表

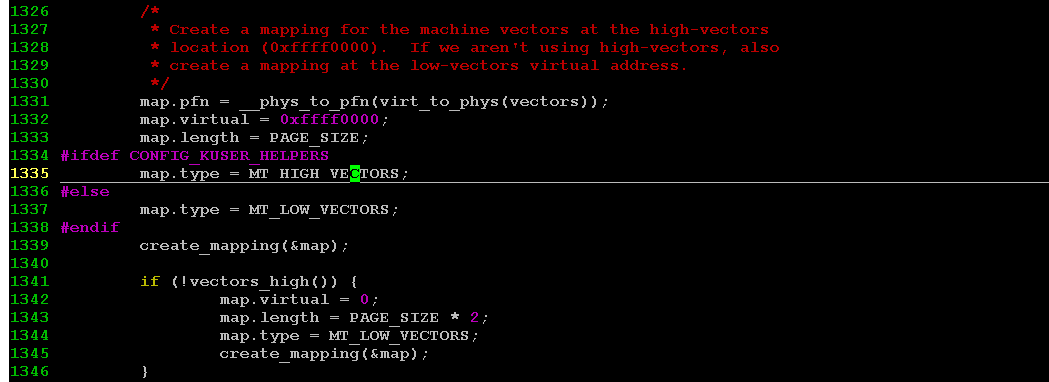


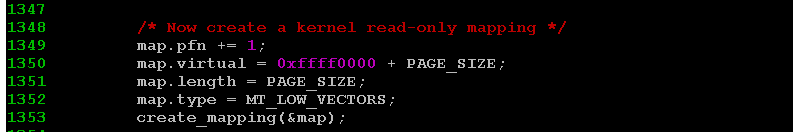
（2）清理VMALLOC区域页表



（3）在高端位置（0xffff0000）创建异常向量

步骤1中申请了两页物理页给异常向量，通过early\_trap\_init函数将向量拷到该区域。这一步重新创建映射关系，将这两个物理页面强制映射到虚拟地址到0xffff0000。





（4）mdesc->map\_io

执行架构相关的io资源映射。

### Kmap\_init