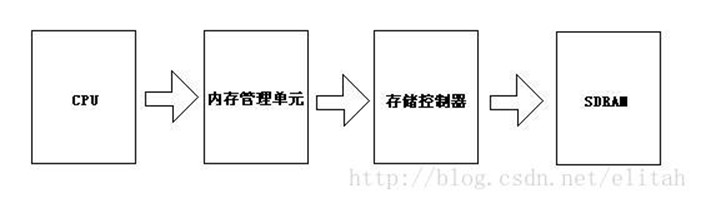
## 一、内存管理单元MMU简介



内存管理单元（Memory Management Unit）简称MMU，它是一块硬件单元，类似于存储控制器，它负责虚拟地址到物理地址的映射，并提供硬件机制的内存访问授权。

MMU可以使每个用户进程都拥有独立的地址空间，同时，内存访问权限的检查可以保护每个进程所用的内存不会被其他进程破坏。

早期的计算机由于性能低下，所用的程序是非常小的，可以全部装入内存中，随着技术的发展，物理内存无法满足应用程序的要求，所以引入了虚拟存储器。

虚拟存储器从逻辑上对内存容量进行扩充，用户看到的大容量只是一种感觉，是虚拟的，在32位的CPU系统中，这个虚拟内存地址的范围为0~0xFFFFFFFF（4GB），我们把这个地址范围称为虚拟地址空间。与虚拟地址空间、虚拟地址相对应的是物理地址空间、物理地址，它们对应实际的内存。

我们在内存中划出一小块空间，用来存储虚拟内存与物理内存之间的映射关系。

如段（Section，1MB）映射，我们使用Excel中的术语来说的话，对于32位CPU，拥有4G的虚拟地址空间，所以只需要4096个单元格，每个单元格占用4Byte，用来描述物理内存的一段（1MB），只需在内存划分出16KB即可

除此至外，还有页（Page）映射方式，页的大小有3种，大页（64KB）、小页（4KB）、极小页（1KB）。

同时，上面所说的单元格，其实在MMU中称作"描述符"，在描述符中，不但保存了段、大页、小页、极小页的其实物理地址，还保存了对应的内存访问权限。

在配置好MMU之后，MMU将开始相应CPU的请求，根据描述符中的内容，做出相应的操作。

同时，为了提供性能和程序的运行速度，还引入了TLB和Cache。

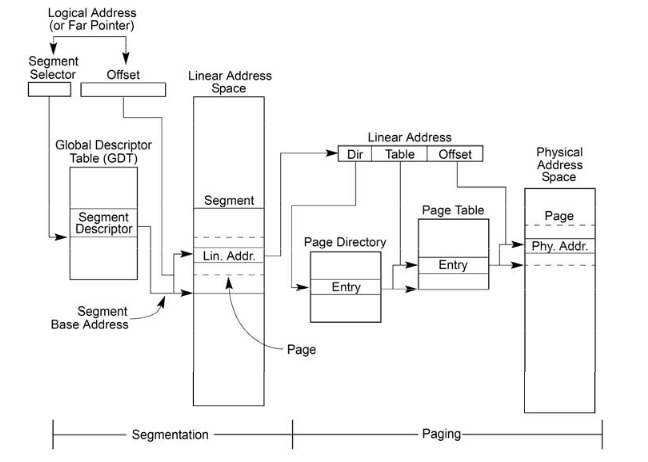
## 二、X86地址映射原理

**8086    :逻辑地址-物理地址**

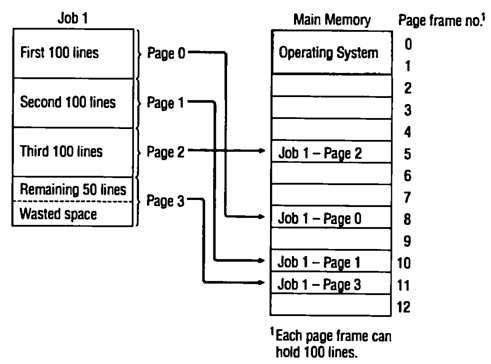
**80386    :虚拟地址-线性地址-物理地址**

### 2.1、地址变换过程

[**Linux**](http://www.madhex.com/redirect/aHR0cDovL2xpYi5jc2RuLm5ldC9iYXNlL2xpbnV4)采用页式存储管理，进程地址空间被以Page划分,Page默认4KB可以修改。物理内存被划分为相同大小的Page Frame，页帧。   
**x86**逻辑地址(VA)段转化(segment translation)后得到线性地址，页转化(page translation)后得到物理地址(PA)。也就是说程序给入一个逻辑地址，CPU拿到后经过两个映射得到数据总线的地址（物理地址）。

借用前辈的图描述IA\_32 (属于X86体系结构的32位版本)段页式地址转化(映射)的过程：   


x86 CPU\*\*段机制\*\*(Segmentation)可以将程序的代码(Code)、数据(Data)、栈(Stack)分开。使多各进程互不干扰。也就是说，分段机制把虚拟地址空间的一个逻辑地址转换为线性地址空间的一个内存地址。

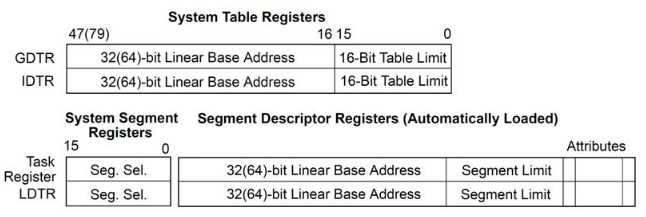
**页机制**(Paging)实现以page为需求的虚拟内存系统，在需要时才分配物理内存。页机制也可以隔离多任务。   
逻辑地址有由页号和偏移量组成。内存被分为N个page，一个Job申请了4个Page，0-3，那么内容逻辑上会以此由0排到3号page，3号page中往往或有浪费空间，内部碎片(internal fragmentation)。然后逻辑上连续的这4个Page会被映射在物理内存的不同位置，不一定顺序或者连续。如图：   


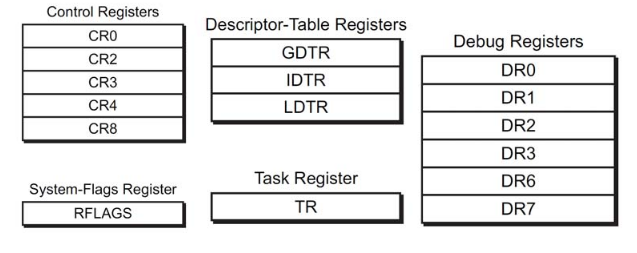
1. 具体x86地址映射说明

IA\_32提供的寄存器从功能上可分为类：CPU寄存器、系统寄存器。(笔者自己的分类)   
**CPU寄存器**   
IA\_32 (属于X86体系结构的32位版本)提供10个32bit，6个16bit寄存器，分三类：

* 通用寄存器(8\* 32bit registers)   
    
  + 数据寄存器 EAX、EBX、ECX、EDX
  + 索引(变址)寄存器 ESI、EDI - 字符串处理指令相关
  + 指针寄存器 ESP(Stack Pointer,当前栈顶)、EBP(Base Pointer,当前栈底) - 维护栈
* 控制寄存器(2\* 32bit registers)   
    
  + EIP，Instruction Pointer，跟踪下一条要执行的指令，也称程序计数寄存器
  + EFLAGS，Program Status and Control Register
* 段寄存器(6\* 16bit registers)   
    
  + CS\ES\DS\FS\GS\SS
  + 注：32bit架构中有6个段寄存器所以最多可以同时访问6个段。
  + 注：在之后介绍两种CPU运行模式下段寄存器功能的变化。

**系统寄存器**   
初始化CPU和控制系统的相关操作，用到的寄存器。与EFLAGS寄存器也有关。

* EFLAG的IOPL
* 控制寄存器   
    
  + CR0、CR2、CR3、CR4 - 系统级别，CPU特殊功能
* 系统描述表寄存器 - 只能在保护模式下使用   
    
  + GDTR(全局描述表寄存器 - GDT Entry线性基地址)
  + LDTR(局部描述表寄存器 - 进程自己LDT的  
    段描述符)
  + IDTR(中断描述表寄存器)
  +    
    *(image source:*[*http://ilinuxkernel.com/?p=1276*](http://www.madhex.com/redirect/aHR0cDovL2lsaW51eGtlcm5lbC5jb20vP3A9MTI3Ng==)*)*
* 任务寄存器 TR(Linux 没有使用)
* 调试寄存器 DR0~DR7   
    
  + 如DR7是断点控制寄存器
* 测试寄存器 TR0~TR7

如图：   


### 2.2、工作模式

IA\_32有两种工作模式：实模式，保护模式。

（有兴趣可以看一个贴，关于"寻访x86处理器"实模式"和"保护模式"的前世今生"）

实模式下：前4个段寄存器CS、DS、ES和SS与先前CPU中的所对应的段寄存器的含义完全一致，内存单元的逻辑地址仍为"段值：偏移量"(Segment:Offset)的形式。为访问某内存段内的数据，必须使用该段寄存器和存储单元的偏移量。

比如分别用CS(Code Segment), DS(Data Segment), SS(Stack Sagmet)来描述进程的代码段，数据段，堆栈段，然后用其中某个地址，如要访问堆栈数据，将SS数据左移4bit + Offset(DI)就是实际要访问的虚拟地址。

实模式由16位段寄存器做段基地址，和6位偏移地址形成20位的物理地址，最大寻址空间1MB，最大分段Limit 64KB。可以使用32位指令。在实模式下，所有的段都是可以读、写和可执行的。所以这种模式直接算出来的就是物理地址。

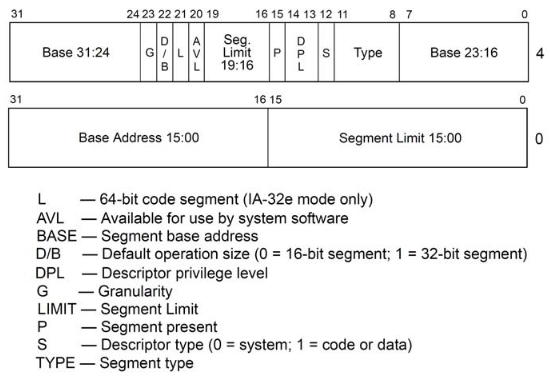
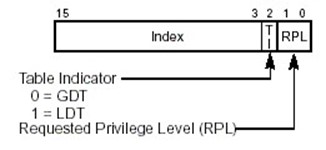
保护模式下：顾名思义要对内存空间做保护，不能像实模式那样随意访问。得到的是虚拟地址，需要再进行转化才能得到物理地址。

段寄存器装入的不是段地址。新的模式需要描述基地址、长度、权限等等来做保护，使用一64bit的数据结构 - 段描述符Selector。为了用16bit的寄存器来访问这个数据结构的信息，将这些64bit的段描述符放在一个数组中，将段寄存器的值作为下标索引间访问。

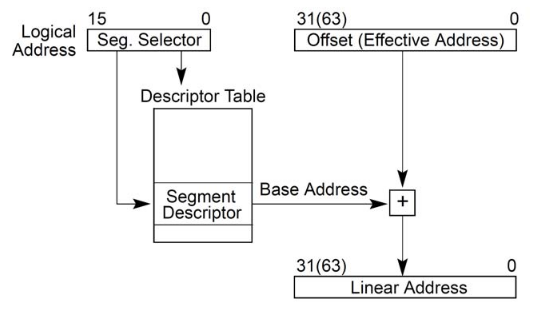
上面提到的数组就是**GDT**(Global Descriptor Table)，GDT不但存有段描述符，还有其他64bit描述符。

**GDT**(Global Descriptor Table)可以放在内存任何位置，知道入口就可以。所以Intel设计了寄存器GDTR，存放GDT入口地址，通过LGDT指令装入(32bit线性地址+16bit Limit如上描述表寄存器图)。之后CPU根据GDTR来访问GDT。GDT只有一个。

GDT需要8Byte对齐，第一个描述符全0。每个CPU一个GDT。

**LDT**(Local Descriptor Table)与GDT结构类似，不过可以存在多个，非全局可见，只对任务可见。每个任务至多有一个。LDT自身作为一个段存在，段描述符放在GDT中。进程需要通过LLDT指令(操作数是16bit Selector，GDT中所要LDT的索引)将LDT的段描述符装入LDTR，进而访问自己的LDR。   
Linux事实上没有用到LDT。   
Segment **Descriptor**: 64bit in GDT/LDT   
   
Segment **Selector**, 段选择子，高13 bit是GDT/LDT下标索引，+ 1 bit 处于GDT or LDT + 2 bit特权请求等级。   


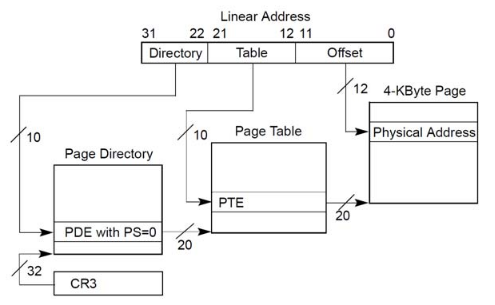
保护模式具体的还有有两种模式：段式，段页式。

1. 段式   
   IA\_32允许将段基地址用任何32bit能表示的值描述，Limit为32bit能表示的2^12的倍数的任何值(4K,Page的倍数)。在保护模式找到所需物理地址的过程是：  
     
   1. 段寄存器装入Selector
   2. 用这个Selector做索引在GDT/LDT中找到相应的段描述符
   3. 取出64bit描述符中记录目标段的的Base Address (+越界越权查看)
   4. 用Base Address + Offset得到要访问的线性地址。   
      所以可以看出，这样的访问方式适合于没有虚拟地址概念的OS，大家直接管理访问的都是线性地址。对于实现虚拟内存的，段页式更有效。   
         
      *(image source: Intel Manuals )*
2. 段页式

资料中有一种是说尽量模仿纯页模式的映射方法：事实上本身应该是强调页模式，但是IA\_32无法完全禁止段模式，但可以让其效果降低。最终效果事实上是段页式。   
方法是利用IA\_32提供的"Basic Flat Model"在GDT定义两个段描述符:Code和Data Segment，两者都包含整个线性空间(Segment Limit = 4G, Kernel Segment)。Linux用这种方式，也就是说只使用了两个段，CS，DS每个进程的六个段寄存器值都相同，只有EIP(当前指令)、ESP(栈顶指针)不同。

事实上上一步段式操作完成了虚拟到线性地址的映射，需要再做映射完成线性到物理地址的映射，完成本章节开始的第一个图的映射过程。过程是：

1. 得到的线性地址是10 bit Directory Index + 10 bit Table Index + 12 bit Offset 如下图。
2. 从CR3获取Page Directory基地址(每个进程有自己的控制块task\_strcut，里面记录了自己C3的信息)找到Page Directory的位置。
3. 以线性地址的前10bit为索引找到对应PDE(Page Directory Entry)含一个Page Table的地址。
4. 以线性地址的第二个10 bit为索引找到PTE(Page Table Entry)含有要访问4KB Page Frame的地址。
5. 根据20 bit Page Fram基地址与线性地址中Offset相加得到要访问的物理地址。

   
Two-level page table structure in x86 architecture (without PAE or PSE).

返回来再看Linux的实现(只用两个段地址CS(赋值\_\_USER\_CS)，DS/ES/SS(赋值\_\_USER\_DS),FS/GS是0)。在程序/arch/x86/include/asm/segment.h中所赋值分别是：

//文件中可以看到系统对于GDT的设计

#define GDT\_ENTRY\_DEFAULT\_USER\_CS 14

#define GDT\_ENTRY\_DEFAULT\_USER\_DS 15

#define \_\_USER\_DS (GDT\_ENTRY\_DEFAULT\_USER\_DS\*8+3)

#define \_\_USER\_CS (GDT\_ENTRY\_DEFAULT\_USER\_CS\*8+3)

//2.6.18内核的，笔者在自己给板子编译zImage的Linux 3.0.8的内核中查看有相同描述

得到的结果是：

\_\_USER\_DS = 0000000001110011//index = 14, GDT

\_\_USER\_CS = 0000000000111011//index = 15, GDT

它们被放入段寄存器中。通过5和6为Index找GDT中对应的段基址。那么GDT中的内容在哪里？段基址是多少？   
在2.6.18内核

/\*

\* The Global Descriptor Table contains 28 quadwords, per-CPU.

\*/

.align PAGE\_SIZE\_asm

ENTRY(cpu\_gdt\_table)

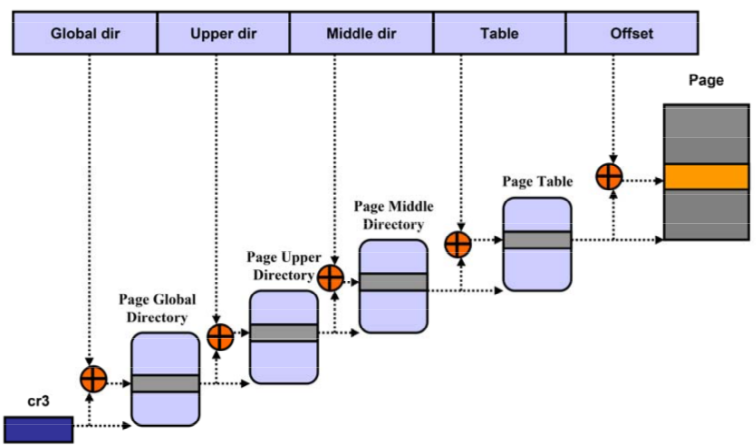
//中间略去下面是第14 15条

    .quad 0x00cffa000000ffff  
/\* 0x73 user 4GB code at 0x00000000 \*/

    .quad 0x00cff2000000ffff  
/\* 0x7b user 4GB data at 0x00000000 \*/

根据上面Segment Descriptor的图的描述，找到对应的Base Address。   
入用\_\_USER\_DS可以找到对应信息：BaseAddress是0，Limit是0xfffff。颗粒度标志G=1。表示Limit的单位是4KB。所以段的长度为0x0 ~ 0x8000 0000 -1。4G空间。代码段也是一样的。那么最后一个堆栈段上的变量的地址的效果就是 0(SS和DS段基地址一样) + Offset，逻辑地址的值就是线性地址的值。

发现一些内容找不到，在2.6.25内核后没有i386的文件夹，是x86。想问如何找到定义呢？

在得到线性地址后在利用MMU做到物理地址的转化（为了使Linux能在32位和64位CPU上运行,就要采用统一的页面地址模型。从2.6.11内核开始,页面地址模型采用了4级页面）：   


## 三、ARM地址映射原理

ARM    : VA - MVA – PA

ARM(MVA-PA) = X86(**虚拟地址-线性地址-物理地址**)

### 3.1、地址变换过程

**VA - MVA**

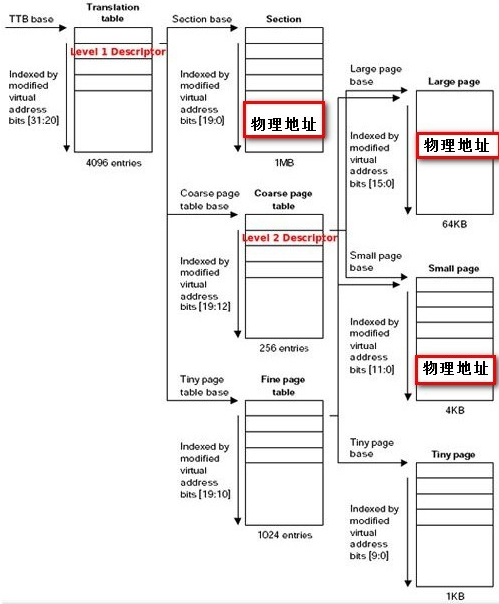
如果总是直接用VA做映射，如果两个进程所用的VA重叠，则切换进程为了把VA映射到不同PA，需要重建页表，这样引起的Cache 无效和MMU(其中的TLB)抖动带来很大开销。所以Kernel会用VA和PID(进程ID)建立MVA来方便进程切换，如果两个不同进程原本访问VA重叠，比如都用(0-32M空间)，但是经过PID的移位后就可以分开了。这样生成的MVA减少了进程切换的代价。   
芯片的CP15些处理器中register 13是识别进程的寄存器，PID写在[31:25]之后的[25:0]写为0。   
这样一来，能识别的进程最多2^7为128个，对应在4G虚拟空间就有每个进程32MB。

手册中:Addresses issued by the ARM9TDMI core in the range 0 to 32MB are translated by CP15 register 13, the ProcID register. Address A becomes A + (ProcID x 32MB). It is this translated address that is seen by both the Caches and MMU. Addresses above 32MB undergo no translation.   
表示出来，进程地址的转化就是：PA = VA + PID \* 32MB

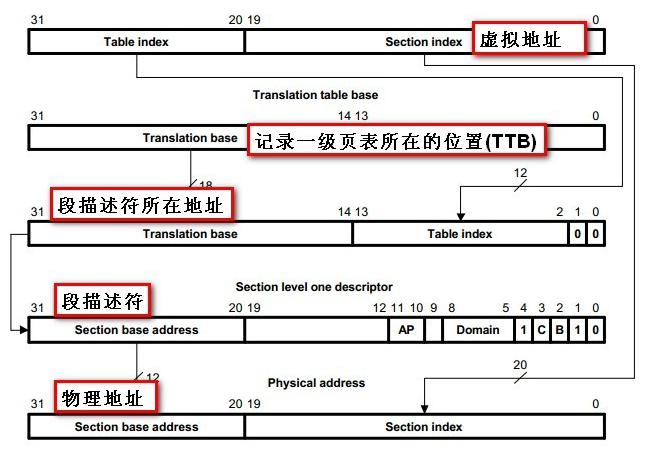
用这样方式，MMU和Cache使用MVA，就可以减少重建页表的开销。   
一些问题

ARM 用上述方式如果有大于32MB的程序呢？是错误吗。

**MVA - PA**   
得到MVA后喂到MMU做到物理地址的映射。过程如下：



中间用到的地址：



解释：

1. CP15从C2寄存器获得页基地址(**TTB**, translation table base)，一级页表所在位置。要求16K对齐。
2. MVA高12bit作为索引值，对应TTB中的一个Entry。12 bit 也就是 4K个，每个Entry可以表示1MB的地址空间。
3. TTB + MVA[31:20]得到的Entry有三种:   
     
   1. 段页(Section)描述符：指向一个1MB物理空间描述符末两位 0b10)
   2. 粗页(Coarse Page)描述符：有256个二级页表项，每个二级页表项代表4KB空间（理解：Entry指向的1MB空间中要存放32bit(4B)的描述符，可以放256个）(0b01)
   3. 细页(Fine Page)描述符：有1024个二级页表项，每个二级页表项代表1KB空间。(0b11)
   4. 还有一种是的得到的末两位为(0b00)没有对应Entry访问会产生fault
4. 如果上一步指向Section，用MVA[19:0]作为索引可以找到1MB中对应请求的物理地址。
5. 如果指向Coarse Page Table，根据MVA[19:12]做索引可以找到一个Entry，对应有两种情况：   
     
   1. 找到的是Large Page Descriptor，对应一个64KB大小的物理地址，不过由于之前提到每个粗页表项对应4KB，所以有16个Descriptors对应这一个Large Page Descriptor。再用MVA[15:0]索引这个64KB物理地址中的某个位置。
   2. 找到的是Small Page Descriptor，对应一个4KB大小的物理地址，所以是1:1。再用MVA[11:0]索引这个Page Frame中的位置。
6. 如果指向的是Fine Page Table，则用MVA[19:10]作为索引找到一个Entry，有相应三种情况，分别索引大页(Large Page)，小页(Small Page)，极小页(Tiny Page)，可以根据上图得知索引方式。

notification   
source: 《Linux设备驱动开发详解》（第二版），内容为读书笔记和网络资料，有些资料原始来源不详，分享为了方便自己和他人查阅。如有侵权请及时告知，对于带来的不便非常抱歉。转载请注明来源。个人所学有限，若有错误和不足还请不吝赐教，我会及时更正。Terrence Zhou.   
[http://blog.csdn.net/ts\_dchs](http://www.madhex.com/redirect/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvdHNfZGNocw==)

http://www.madhex.com/wp-content/uploads/2017/02/021417_0222_S3C2440M13.png

### 3.2、TLB 简介

    从虚拟地址到物理地址的转换过程可知：使用一级页表进行地址转换时，每次读/写数据需要访问两次内存，第一次访问一级页表获得物理地址，第二次才是真正的读/写数据：使用两级页表时，每次读/写数据需要访问3次内存，访问两次页表（一级页表和二级页表）获得物理地址，第三次才是真正的读/写数据。

    上述的地址转换过程大大降低了CPU的性能，有没有办法改进呢？程序执行过程中，所用到的指令、数据的地址往往集中在一个很小的范围内，其中的地址、数据经常多次使用，这称为程序访问的布局性。由此，通过使用一个高速、容量相对较小的存储器来存储近期用到的页表条目（段/大页/小页/极小页描述符），以避免每次地址转换时都到主存去查找，这样可以大幅度地提高性能。这个存储器用来帮助快速地进行地址转换，称为"转译查找缓存"（Translation Lookaside Buffers, TLB）。

### 3.3、CACHE简介

    同样基于程序访问的局部性，在主存和CPU通用寄存器之间设置一个高速的、容量相对较小的存储器，把正在执行的指令地址附近的一部分指令或数据从主存调入这个存储器，供CPU在一段时间内使用，这对提高程序的运行速度有很大的作用。这个介于主存和CPU之间的高速小容量存储器称作高速缓存存储器（Cache）。

## 四、ARM内存管理单元MMU、TLB、Cache的控制指令

### 4.1. 协处理器简介

 一个系统中最多可连接16个协处理器，每个协处理器都通过唯一的ID号标识。ARM7TDMI处理器包含两个

内部协处理器：

    CP14 通信通道协处理器

    CP15 为cache和MMU功能提供的系统控制协处理器

    因此，不能将外部协处理器的编号分配为14和15。ARM还保留了其他的协处理编号，见表：



### 4.2. 协处理器指令

    S3C2410/S3C2440中，除了有一个ARM920T的CPU核外，还有若干个协处理器。协处理器也是一个微处理器，它被用来帮助CPU完成一些特殊功能，比如浮点计算等。对MMU、TLB、Cache等的操作就涉及协处理器。CPU核与协处理器间传送数据时使用这两条指令：

    <MCR|MRC>{cond} p#,<expressionl>,Rd,cn,cm{,<expression2>}

MRC            //从协处理器获得数据，传给ARM920T CPU核的寄存器

    MCR            //数据从ARM920T CPU核的寄存器传给协处理器

    {cond}            //执行条件，省略时表示无条件执行

    p#                //协处理器序号

    <expression 1>    //一个常熟

    Rd                //ARM920T CPU核的寄存器

    cn和cm            //协处理器中的寄存器

    <expression 2>    //一个常数

    其中，<expression 1>、cn、cm、<expression 2> 仅供协处理器使用，它们的作用如何取决于具体的协处理器。

### 4.3. CP15协处理器详解

在基于ARM的嵌入式系统中，存储系统通常是通过系统控制协处理器CP15完成的。

CP15可以包含16个32位的寄存器，其编号为0-15。实际上对于某些编号的寄存器可能对应有多个物理寄存器。在指令中指定特定的标志位来区分这些物理寄存器。有些类似于ARM寄存器中，处于不同的处理器模式时，ARM某些寄存器可能不同。

**CP15 的寄存器列表如表所示：**

|  |  |  |  |
| --- | --- | --- | --- |
| 寄存器编号 | 基本作用 | 在 MMU 中的作用 | 在 PU 中的作用 |
| 0 | ID 编码（只读） | ID 编码和 cache 类型 |  |
| 1 | 控制位（可读写） | 各种控制位 |  |
| 2 | 存储保护和控制 | 地址转换表基地址 | Cachability 的控制位 |
| 3 | 存储保护和控制 | 域访问控制位 | Bufferablity 控制位 |
| 4 | 存储保护和控制 | 保留 | 保留 |
| 5 | 存储保护和控制 | 内存失效状态 | 访问权限控制位 |
| 6 | 存储保护和控制 | 内存失效地址 | 保护区域控制 |
| 7 | 高速缓存和写缓存 | 高速缓存和写缓存控制 |  |
| 8 | 存储保护和控制 | TLB 控制 | 保留 |
| 9 | 高速缓存和写缓存 | 高速缓存锁定 |  |
| 10 | 存储保护和控制 | TLB 锁定 | 保留 |
| 11 | 保留 |  |  |
| 12 | 保留 |  |  |
| 13 | 进程标识符 | 进程标识符 |  |
| 14 | 保留 |  |  |
| 15 | 因不同设计而异 | 因不同设计而异 | 因不同设计而异 |

**注：以下寄存器中相应位的含义在不同的处理器中可能不同，但总体功能不变**

#### （一）CP15 的寄存器 C0

CP15中寄存器C0对应两个标识符寄存器，由访问CP15中的寄存器指令中的指定要访问哪个具体物理寄存器，与两个标识符寄存器的对应关系如下所示：

|  |  |
| --- | --- |
| opcode2 编码 | 对应的标识符号寄存器 |
| 0b000 | 主标识符寄存器 |
| 0b001 | cache类型标识符寄存器 |
| 其他 | 保留 |

##### （1）主标识符寄存器

**指令如下：**

**MRC    P15，0，R0，C0，C0，0       #将主标示符寄存器的内容读到AMR寄存器R0中**

主标示符的编码格式对于不同的ARM处理器版本有所不同。

**对于AMR7之后的处理器，其主标示符编码格式如下 ：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 30 24 | 23 20 | 19  16 | 15 4 | 3       0 |
| 由生产商确定 | 产品子编号 | ARM体系版本号 | 产品主编号 | 处理器版本号 |

**各部分的编码详细含义如下表所示：**

|  |  |
| --- | --- |
| 位 | 说     明 |
| [3: 0] | 生产商定义的处理器版本号 |
| [15: 4] | 生产商定义的产品主编号  其中最高 4 位即位 [15:12] 可能的取值为0x0~0x7 但不能是 0x0 或 0x7  因为：  0x0表示 ARM7之前的处理器  0x7 表示ARM7处理器 |
| [19: 16] | ARM 体系的版本号，可能的取值如 下：  0x1   ARM 体系版本 4  0x2   ARM 体系版本 4T  0x3   ARM 体系版本 5  0x4   ARM 体系版本 5T  0x5   ARM 体系版本 5TE  其他   由 ARM 公司保留将来使用 |
| [23: 20] | 生产商定义的产品子编号。当产品主编号相同时，使用子编号来区分不同的产品子类，如产品中不 同的高速缓存的大小等 |
| [31: 24] | 生产厂商的编号，现在已经定义的有以下值：  0x41  =A  ARM 公司  0x44  =D  Digital Equipment 公司  0x69  =I   intel 公司 |

##### （2）cache类型标识符寄存器

**指令如下：**

**MRC    P15，0，R0，C0，C0，1       #将cache类型标识符寄存器的内容读到AMR寄存器R0中**

ARM 处理器中 cache 类型标识符寄存器的编码格式如下所示：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 31 29 | 28  25 | 24 | 23 12 | 11    0 |
| 000 | 属性字段 | S | 数据 cache 相关属性 | 指令cache 相关属性 |

各部分的编码详细含义如下表所示：

|  |  |
| --- | --- |
| 位 | 含义 |
| [28: 25] | 主要用于定义对于写回类型的cache的一些属性 |
| [24] | 定义系统中的数据 cache 和指令 cache 是分开的还是统一的：  0   系统的数据 cache 和指令 cache 是统一的；  1   系统的数据 cache 和指令 cache 是分开的 |
| [23: 12] | 定义数据 cache 的相关属性  如果位 [24] 为 0 ，本字段定义整个cache 的属性 |
| [31: 24] | 定义指令 cache 的相关属性  如果位 [24] 为 0 ，本字段定义整个cache 的属性 |

* **控制字段位 [28 ： 25] 的含义**

主要用于定义对于写回类型的cache的一些属性

**cache 类型标识符寄存器的控制字段位 [28 ： 25]：**

|  |  |  |  |
| --- | --- | --- | --- |
| 编     码 | cache 类型 | cache 内容清除方法 | cache 内容锁定方法 |
| 0b0000 | 写通类型 | 不需要内容清除 | 不支持内容锁定 |
| 0b0001 | 写回类型 | 数据块读取 | 不支持内容锁定 |
| 0b0010 | 写回类型 | 由寄存器 C7 定义 | 不支持内容锁定 |
| 0b0110 | 写回类型 | 由寄存器 C7 定义 | 支持格式 A |
| 0b0111 | 写回类型 | 由寄存器 C7 定义 | 支持格式 B |

* **控制字段位 [23 ： 12] 及控制字段位 [11 ： 0] 含义**

[23：12]用于定义数据cache的属性，[11: 0]用于定义指令cache的属性

**编码格式如下：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11      9 | 8        6 | 5       3 | 2 | 1  0 |
| 000 | cache 容量 | cache 相联特性 | M | 块大小 |

其中bits[1:0]含义如下：

|  |  |
| --- | --- |
| 编     码 | cache 块大小 |
| 0b00 | 2 个字（ 8 字节） |
| 0b01 | 4 个字（ 16 字节） |
| 0b10 | 8 个字（ 32 字节） |
| 0b11 | 16 个字（ 64 字节） |

其中bits[5:3]含义如下：

|  |  |  |
| --- | --- | --- |
| 编     码 | M=0 时含义 | M=1 时含义 |
| 0b000 | 1 路相联（直接映射） | 没有 cache |
| 0b001 | 2 路相联 | 3 路相联 |
| 0b010 | 4 路相联 | 6 路相联 |
| 0b011 | 8 路相联 | 12 路相联 |
| 0b100 | 16 路相联 | 24 路相联 |
| 0b101 | 32 路相联 | 48 路相联 |
| 0b110 | 64 路相联 | 96 路相联 |
| 0b111 | 128 路相联 | 192 路相联 |

其中bits[8:6]含义如下：

|  |  |  |
| --- | --- | --- |
| 编     码 | M=0 时含义 | M=1时含义 |
| 0b000 | 0.5KB | 0.75 KB |
| 0b001 | 1 KB | 1.5 KB |
| 0b010 | 2 KB | 3 KB |
| 0b011 | 4 KB | 6 KB |
| 0b100 | 8 KB | 12 KB |
| 0b101 | 16 KB | 24 KB |
| 0b110 | 32 KB | 48 KB |
| 0b111 | 64 KB | 96 KB |

#### （二）CP15 的寄存器 C1

CP15中的寄存器C1是一个控制寄存器，它包括以下控制功能：

* 禁止或使能MMU以及其他与存储系统相关的功能
* 配置存储系统以及ARM处理器中的相关部分的工作

**指令如下：**

**mrc p15, 0, r0, c1, c0{, 0}     ；将 CP15 的寄存器 C1 的值读到 r0 中**

**mcr p15, 0, r0, c1, c0{, 0}     ；将 r0 的值写到 CP15 的寄存器 C1 中**

**CP15 中的寄存器 C1 的编码格式及含义说明如下：**

|  |  |
| --- | --- |
| C1中的控制位 | 含义 |
| M（bit[0]） | 0 ：禁止 MMU 或者 PU  1 ：使能 MMU 或者 PU  如果系统中没有MMU及PU，读取时该位返回0，写入时忽略该位 |
| A（bit[1]） | 0 ：禁止地址对齐检查  1 ：使能地址对齐检查 |
| C（bit[2]） | 当数据cache和指令cache分开时，本控制位禁止/使能数据cache。当数据cache和指令cache统一时，该控制位禁止/使能整个cache。  0 ：禁止数据 / 整个 cache  1 ：使能数据 / 整个 cache  如果系统中不含cache，读取时该位返回0.写入时忽略  当系统中不能禁止cache 时，读取时返回1.写入时忽略 |
| W（bit[3]） | 0 ：禁止写缓冲  1 ：使能写缓冲  如果系统中不含写缓冲时，读取时该位返回0.写入时忽略  当系统中不能禁止写缓冲时，读取时返回1.写入时忽略 |
| P（bit[4]） | 对于向前兼容26位地址的ARM处理器，本控制位控制PROG32控制信号  0 ：异常中断处理程序进入 32 位地址模式  1 ：异常中断处理程序进入26 位地址模式  如果本系统中不支持向前兼容26位地址，读取该位时返回1，写入时忽略 |
| D（bit[5]） | 对于向前兼容26位地址的ARM处理器，本控制位控制DATA32控制信号  0 ：禁止 26 位地址异常检查  1 ：使能 26 位地址异常检查  如果本系统中不支持向前兼容26位地址，读取该位时返回1，写入时忽略 |
| L（bit[6]） | 对于ARMv3及以前的版本，本控制位可以控制处理器的中止模型  0 ：选择早期中止模型  1 ：选择后期中止模型 |
| B（bit[7]） | 对于存储系统同时支持big-endian和little-endian的ARM系统，本控制位配置系统的存储模式  0 ： little endian  1 ： big endian  对于只支持little-endian的系统，读取时该位返回0，写入时忽略  对于只支持big-endian的系统，读取时该位返回1，写入时忽略 |
| S（bit[8]） | 在基于 MMU 的存储系统中，本位用作系统保护 |
| R（bit[9]） | 在基于 MMU 的存储系统中，本位用作 ROM 保护 |
| F（bit[10]） | 由生产商定义 |
| Z（bit[11]） | 对于支持跳转预测的ARM系统，本控制位禁止/使能跳转预测功能  0 ：禁止跳转预测功能  1 ：使能跳转预测功能  对于不支持跳转预测的ARM系统，读取该位时返回0，写入时忽略 |
| I（bit[12]） | 当数据cache和指令cache是分开的，本控制位禁止/使能指令cache  0 ：禁止指令 cache  1 ：使能指令 cache  如果系统中使用统一的指令cache和数据cache或者系统中不含cache，读取该位时返回0，写入时忽略。当系统中的指令cache不能禁止时，读取时该位返回1，写入时忽略 |
| V（bit[13]） | 对于支持高端异常向量表的系统，本控制位控制向量表的位置  0 ：选择低端异常中断向量 0x0~0x1c  1 ：选择高端异常中断向量0xffff0000~ 0xffff001c  对于不支持高端异常向量表的系统，读取时该位返回0，写入时忽略 |
| PR（bit[14]） | 如果系统中的cache的淘汰算法可以选择的话，本控制位选择淘汰算法  0 ：常规的 cache 淘汰算法，如随机淘汰  1 ：预测性淘汰算法，如round-robin 淘汰算法  如果系统中cache的淘汰算法不可选择，写入该位时忽略。读取该位时，根据其淘汰算法是否可以比较简单地预测最坏情况返回0或者1 |
| L4（bit[15]） | 对于ARM版本5及以上的版本，本控制位可以提供兼容以前的ARM版本的功能  0 ：保持 ARMv5 以上版本的正常功能  1 ：将 ARMv5 以上版本与以前版本处理器 兼容，不根据跳转地址的 bit[0] 进行 ARM 指令和 Thumb 状态切换： bit[0] 等于 0 表示 ARM 指令，等于 1 表示 Thumb 指令 |
| Bits[31:16]） | 这些位保留将来使用，应为UNP/SBZP |

#### （三）CP15 的寄存器 C2

C2寄存器的别名：Translation table base (TTB) register

C2寄存器用来保存页表的基地址，即一级映射描述符表的基地址。其编码格如下所示：

|  |
| --- |
| 31                             0 |
| 一级映射描述符表的基地址（物理地址） |

#### （四）CP15 的寄存器 C3

CP15 中的寄存器 C3 定义了 ARM 处理器的 16 个域的访问权限。

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31                            0 | | | | | | | | | | | | | | | |
| D15 | D14 | D13 | D12 | D11 | D10 | D9 | D8 | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |

在 CP15的C3寄存器中，划分了 16个域，每个区域由两位构成，这两位说明了当前内存的检查权限：

00：当前级别下，该内存区域不允许被访问，任何的访问都会引起一个domain fault，这时 AP位无效

01：当前级别下，该内存区域的访问必须配合该内存区域的段描述符中AP位进行权检查  
10：保留状态（我们最好不要填写该值，以免引起不能确定的问题）  
11：当前级别下，对该内存区域的访问都不进行权限检查。 这时 AP位无效

所以只有当相应域的编码为 01 时，才会根据  AP位 和协处理器CP15中的C1寄存器的R,S位进行权限检查

#### （五）CP15 的寄存器 C5

CP15 中的寄存器 C5 是失效状态寄存器，分为指令状态失效和数据状态失效。

**MRC p15, 0, , c5, c0, 0 访问数据失效状态寄存器**

**MRC p15, 0, , c5, c0, 1  访问指令状态失效寄存器**

**编码格式如下所示：**

|  |  |  |  |
| --- | --- | --- | --- |
| 31            9 | 8 | 7    4 | 3       0 |
| UNP/SBZP | 0 | 域标识 | 状态标识 |

其中，域标识bit[7:4]表示存放引起存储访问失效的存储访问所属的域。

状态标识 bit[3:0] 表示放引起存储访问失效的存储访问类型，该字段含义如下表所示（优先级由上到下递减）。

|  |  |  |  |
| --- | --- | --- | --- |
| 引起访问失效的原因 | 状态标识 | 域标识 | C6 |
| 终端异常（ Terminal Exception ） | 0b0010 | 无效 | 生产商定义 |
| 中断向量访问异常  （ Vector Exception） | 0b0000 | 无效 | 有效 |
| 地址对齐 | 0b00x1 | 无效 | 有效 |
| 一级页表访问失效 | 0b1100 | 无效 | 有效 |
| 二级页表访问失效 | 0b1110 | 有效 | 有效 |
| 基于段的地址变换失效 | 0b0101 | 无效 | 有效 |
| 基于页的地址变换失效 | 0b0111 | 有效 | 有效 |
| 基于段的存储访问中域控制失效 | 0b1001 | 有效 | 有效 |
| 基于页的存储访问中域控制失效 | 0b1101 | 有效 | 有效 |
| 基于段的存储访问中访问权限控制失效 | 0b1111 | 有效 | 有效 |
| 基于页的存储访问中访问权限控制失效 | 0b0100 | 有效 | 有效 |
| 基于段的 cache 预取时外部存储系统失效 | 0b0110 | 有效 | 有效 |
| 基于页的 cache 预取时外部存储系统失效 | 0b1000 | 有效 | 有效 |
| 基于段的非 cache 预取时外部存储系统失效 | 0b1010 | 有效 | 有效 |

#### （六）CP15 的寄存器 C6

CP15 中的寄存器 C6 是失效地址寄存器，其中保存了引起存储访问失效的地址，分为数据失效地址寄存器和指令失效地址寄存器

**MRC p15, 0, , c6, c0, 0  访问数据失效地址寄存器**

**MRC p15, 0, , c6, c0, 2 访问指令失效地址寄存器**

**编码格式如下所示：**

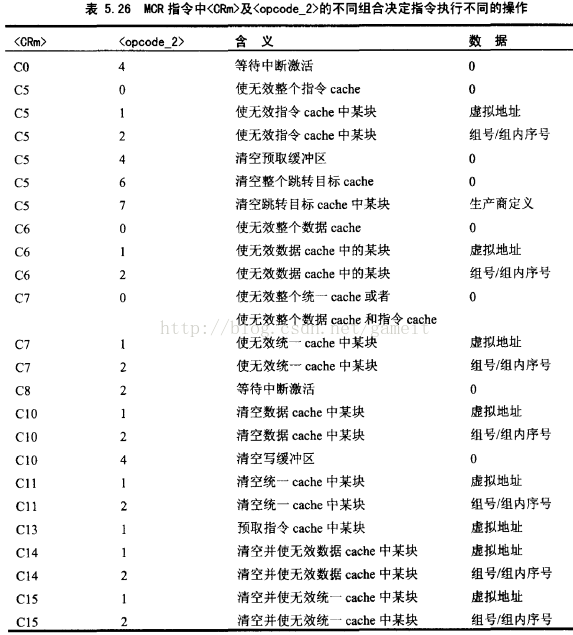
|  |
| --- |
| 31                      0 |
| 失效地址（虚拟地址） |

#### （七）CP15 的寄存器 C7

CP15 的 C7 寄存器用来控制 cache 和写缓存，它是一个只写寄存器，读操作将产生不可预知的后果。

**访问 CP15 的 C7 寄存器的指令格式如下所示：**

**mcr p15, 0, , , crm,                ；不同取值组合，实现不同功能**

**表中的数据是指Rd中的数**

#### （八）CP15 的寄存器 C8

    系统协处理器CP15的寄存器C8就是清除TLB内容的相关操作。它是一个只写的寄存器。

**MCR    p15，0，Rd，c8，CRm，opcode\_2**

        Rd中为要写入C8寄存器的内容，CRm和opcode\_2的不同组合决定指令执行的不同操作。

|  |  |  |
| --- | --- | --- |
| 指令 | Rd | 含义 |
| MCR p15, 0, Rd, c8, c5, 0 | 0 | 使无效整个指令TLB |
| MCR p15, 0, Rd, c8, c5, 1 | 虚拟地址 | 使无效指令TLB中的单个地址变换条目 |
| MCR p15, 0, Rd, c8, c6, 0 | 0 | 使无效整个数据TLB |
| MCR p15, 0, Rd, c8, c6, 1 | 虚拟地址 | 使无效数据TLB中的单个地址变换条目 |
| MCR p15, 0, , c8, c7, 0 | 0 | 使无效整个数据和指令TLB |
| MCR p15, 0, , c8, c7, 1 | 虚拟地址 | 使无效数据和指令TLB中的单个地址变换条目 |

#### （九）CP15 的寄存器 C12

CP15寄存器C12用来设置异常向量基地址，其编码格式如下所示：

**MCR p15, 0, , c12, c0, 0 ；Rd中存放要修改的异常向量基地址**

|  |  |
| --- | --- |
| 31              5 | 4         0 |
| 异常向量基地址 | Reserve |

**注：只有ARM11和cortex-a 可以任意修改异常向量基地址。arm7,ARM9,ARM10只可以在0地址或0xffff0000中**

#### （十）CP15 的寄存器 C13

CP15中的寄存器C13用于快速上下文切换。其编码格式如下所示。



访问寄存器C13的指令格式如下所示。

**MCR           p15, 0,,,c0,0**

**MRC           P15, 0,,,c0,0**

其中，  
在读操作时，结果中位[31：:25]返回PID，其他位  
的数值是不可以预知的。写操作将设置PID的值。

        当PID的值为0时，MVA = VA  |  (0（PID）<<25)，MVA=VA,相当于禁止了FCSE。系统复位后PID即为0.

        当PID的值不为0时，相当于使能了FCSE。

## 五、ARM内存管理单元MMU操作实例

关闭缓存

disable\_caches:

/\*\*\*\*\*\*\*flush I/Dcaches and mmu\*\*\*\*\*/

mcr p15, 0, r0, c7, c7, 0

mrc p15, 0, r0, c1, c0, 0

bic r0, r0, #0x00000007

mcr p15, 0, r0, c1, c0, 0

mov pc, lr

**reference**   
[[1] Linux地址地址映射，](http://www.madhex.com/redirect/aHR0cDovL2lsaW51eGtlcm5lbC5jb20vP3A9MTI3Ng==)[http://blog.chinaunix.net/uid-20528014-id-314322.html   
[2] Linux内核高端内存，](http://www.madhex.com/redirect/aHR0cDovL2Jsb2cuY2hpbmF1bml4Lm5ldC91aWQtMjA1MjgwMTQtaWQtMzE0MzIyLmh0bWw=)[http://ilinuxkernel.com/?p=1013   
[3] x86寄存器简介，](http://www.madhex.com/redirect/aHR0cDovL2lsaW51eGtlcm5lbC5jb20vP3A9MTAxMw==)[http://blog.csdn.net/shrekmu/article/details/8588341   
[4] 8086寄存器介绍，比较认真，](http://www.madhex.com/redirect/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvc2hyZWttdS9hcnRpY2xlL2RldGFpbHMvODU4ODM0MQ==)[http://www.cnblogs.com/zhaoyl/archive/2012/05/15/2501972.html   
[5] Intel® 64 and IA-32 Architectures Software Developer Manuals，](http://www.madhex.com/redirect/aHR0cDovL3d3dy5jbmJsb2dzLmNvbS96aGFveWwvYXJjaGl2ZS8yMDEyLzA1LzE1LzI1MDE5NzIuaHRtbA==)[http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html   
[6] Page Table，](http://www.madhex.com/redirect/aHR0cDovL3d3dy5pbnRlbC5jb20vY29udGVudC93d3cvdXMvZW4vcHJvY2Vzc29ycy9hcmNoaXRlY3R1cmVzLXNvZnR3YXJlLWRldmVsb3Blci1tYW51YWxzLmh0bWw=)[https://en.wikipedia.org/wiki/Page\_table   
[7] MMU，寄存器，s3c2410，](http://www.madhex.com/redirect/aHR0cHM6Ly9lbi53aWtpcGVkaWEub3JnL3dpa2kvUGFnZV90YWJsZQ==)[http://blog.csdn.net/WINITZ/article/details/4057495](http://www.madhex.com/redirect/aHR0cDovL2Jsb2cuY3Nkbi5uZXQvV0lOSVRaL2FydGljbGUvZGV0YWlscy80MDU3NDk1)