s3c2410 watchdog (编辑完毕)

作者：蔡于清
www.another-prj.com

1．什么是watchdog?
watchdog,中文名称叫做“看门狗”，全称watchdog timer,从字面上我们可以知道其实它属于一种定时器。然而它与我们平常所接触的定时器在作用上又有所不同。普通的定时器一般起记时作用，记时超时(Timer Out)则引起一个中断，例如触发一个系统时钟中断。熟悉windows开发的朋友应该用过windows的Timer，windows Timer的作用与方才所讨论的定时器在功能上是相同的，只是windows Timer属于软件定时器,当windows Timer记时超时则引起App向System发送一条消息从而触发某个事件的发生。我们从以上的描述可知不论软件定时器或硬件定时器他们的作用都是在某个时间点上引起一个事件的发生，对于硬件定时器来说这个事件可能是通过中断的形式得以表现，对于软件定时器，这个事件则可以是以系统消息的形式得以表现。正如本文开头所讲的，watchdog本质上是一种定时器，那么普通定时器所拥有的特征它也应该具备，是的，当它记时超时时也会引起事件的发生，只是这个事件除了可以是系统中断外，它也可以是一个系统重起信号(Reset Signal)，可以这么说吧，**能发送系统重起信号的定时器我们就叫它watchdog.**

2．watchdog的工作描述
当一个硬件系统开启了watchdog 功能，那么运行在这个硬件系统之上的软件必须在规定的时间间隔内向watchdog发送一个信号.这个行为简称为"喂狗"(feed dog)，以免watchdog记时超时引发系统重起。

3．watchdog存在的意义是什么？
你可能会问watchdog存在的意义是什么？开启了watchdog之后软件必须定时向它发信息，这不是麻烦又耗费资源的行为吗？其实这个行为很重要，这个行为是软件向硬件报告自身运行状态的一中手法。一个软件运行良好，那么它应该可以在规定的时间间隔内向watchdog发送信息，这等同于软件每隔一段时间就告诉硬件：“嘿，哥们，我在好好的跑着呢，你放心吧。”，若软件由于某个不当的操作而进入死循环(也就是俗称的死机)，则他无法向watchdog发送信息了，watchdog将发生记时超时，从而引起硬件重起。如果没有watchdog的存在，程序已经死掉了，但我们的用户还一头雾水，以为系统正在进行大规模的运算而进行耐心的等待。。。这一等可就是天荒地老啊。。。-_-!!

4．s3c2410 watchdog的操作
对于s3c2410 的watchdog来说,**PCLK**是它唯一的时钟信号源。(不知道PCLK的朋友可以上网搜搜或看我下一篇文章)
s3c2410 用了 3 个寄存器对watchdog进行操作，3 个寄存器分别为：WTCON，WTDAT，WTCNT。
WTCON：watchdog控制寄存器
WTDAT：watchdog数据寄存器
WTCNT：watchdog记数寄存器

以上各个寄存器的详细信息请参考s3c2410 数据手册上关于watchdog部分
5．s3c2410 watchdog 工作描述：
在开启watchdog之前，我们必需在寄存器WTDAT里面存有一个值，在watchdog开启之后这个值会被自动加载进寄存器WTCNT中，WTCNT的作用将在下面进行讲解，现在你只需要知道WTDAT必须有一个值，这

个值将被自动装进WTCNT中(注 1)

Watchdog根据PCLK，Prescaler Value，Clock Select会产生一个watchdog自己的工作周期，我们把这个工作周期记为t_watchdog（注 2），watchdog在一个t_watchdog周期结束时会产生一个记数递减信号，每当这个信号产生时，WTCNT中的值便减 1，若在WTCNT递减为 0(Timer Out)的时候软件层还没有重新往WTCNT中写入数值(这个行为便是我上文提到的喂狗)，则watchdog触发Reset Signal，系统重起。
根据上述的描述，我们可以更形象地描述watchdog的工作原理和 3 个寄存器之间的相互关系：

WTCNT通过WTDAT得到一个值，watchdog在每个t_watchdog周期里向WTCNT发送一个递减信号，当WTCNT的值递减到 0 的时候则发生time out，重而重起系统。

下面我帖出一段设置watchdog并开启watchdog的程序

```
1：void enable_watchdog()
2：{
3：                rWTCON=0x7F81;
4：                rWTDAT=0x8000;
5：                rWTCON|=1<<5;
6：}
```

rWTCON，rWTDAT分别为寄存器WTCON，WTDAT的地址解引用，我如下定义他们

```
#define     rWTCON          (*(volatile unsigned int *)0x53000000)
#define     rWTDAT          (*(volatile unsigned int *)0x53000004)
```

从上面的设置我们可知寄存器WTCON的值为 0x7F81，分解出来得：

```
Prescaler Value             =255
Division_factor             =16(Clock Select=16)
Interrupt Generation        =0(不产生中断)
Reset                       =1(开启Reset Signal)
```

第 4 行设置寄存器WTDAT的值为 0x8000。
第 5 行开启watchdog

当调用上面的函数之后，你的系统已经开启了watchdog，所以你必须在WTCNT中的值递减到 0 之前重新往该寄存器写入一个非 0 值(feed dog)，否则将引起系统重起,以下是feed_dog函数

```
void feed_dog()
{
    rWTCNT=0x8000;
}
```

下面是void enable_watchdog()和feed_dog()配合使用的一个例子

```
void main()
{
        init_system();
        .
        ...
        .....
        enable_watchdog();
        .
        ...
        .....
        while(1)
        {
                feed_dog();
        }
}
```

在这个例子中我假设了main函数是系统的主函数，在做完一系列系统初始化之后enable_watchdog()函数被调用，此时watchdog被启动，下面的while循环则是不断的进行feed_dog，使系统不发生重起。当然在实际应用中不可能采取这种架构来对watchdog进行操作，一般来说feed_dog函数的调用是被安插在定时器的中断服务例程中，当然，定时器的time out（注意是定时器的time out,不是watchdog的 time out）时间长度必须合适，否则在定时器还没来得及发生中断调用feed_dog函数之前，watchdog已经time out了，那也将引起系统重起.

注 1：事实上，WTDAT和WTCNT这两个寄存器在系统上电之后会被硬件自动的填入两个初始值 0x8000，开启watchdog之后，WTCNT并没有马上就把WTDAT中的值装入，而是使用初始值 0x8000。**在发生第一次time out之后，WTDAT寄存器中的值才会被真正的装载进WTCNT寄存器中。**

注 2：t_watchdog可根据公式对其进行计算：

t_watchdog= 1/ (PCLK / (Prescaler value + 1) / Division_factor)

Prescaler Value位于寄存器WTCON的 8 至 15 位，其值为 0~255

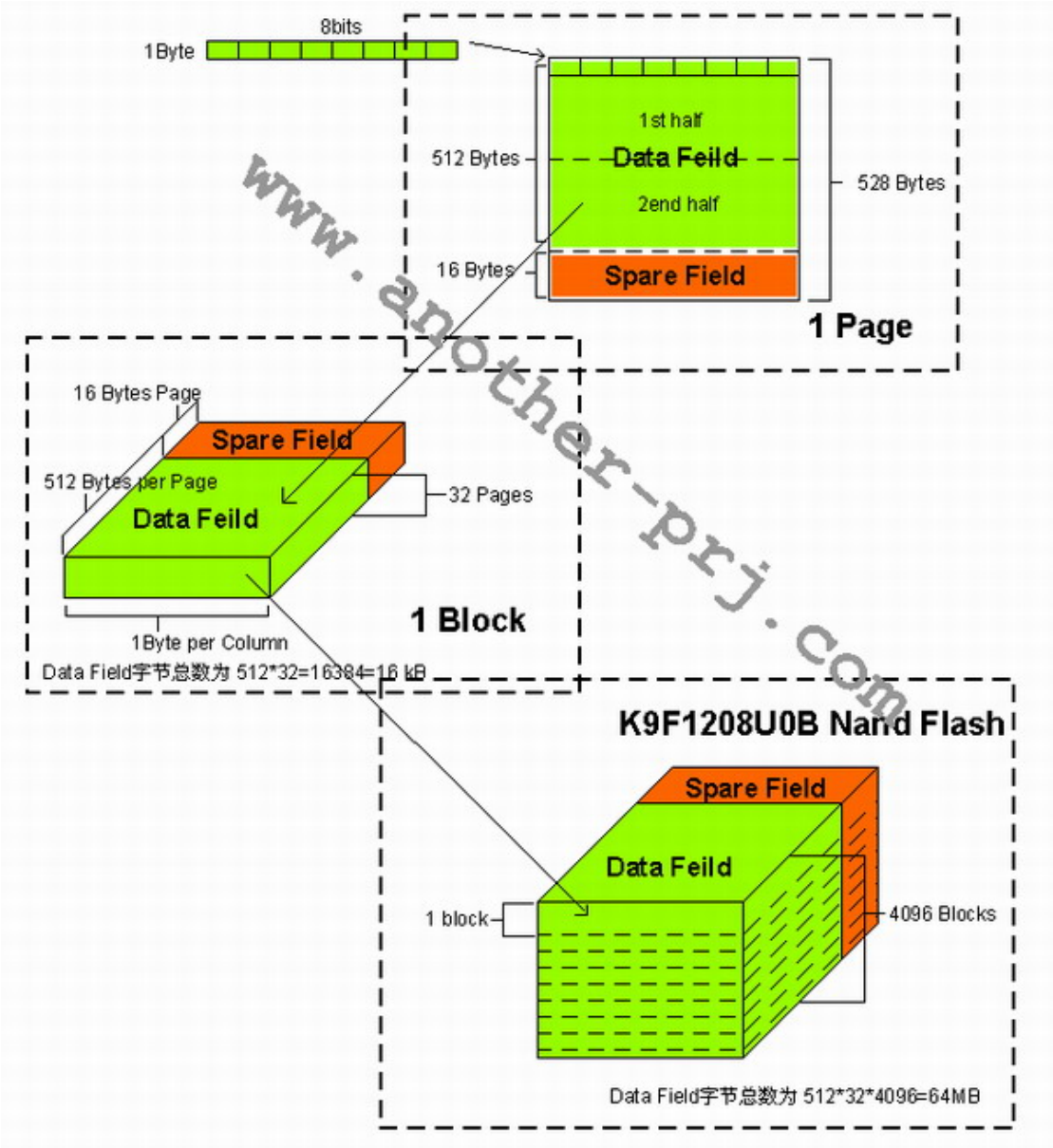
Division_factor由寄存器WTCON中的 3~4 位(Clock Select)决定，其值可以为 00，01，10，11 分别代表 Division_factor的值为 16，32，64，128

关于各个寄存器的详细信息请参考s3c2410 的操作手册

s3c2410 NandFlash K9F1208U0A/ K9F1208U0B 的读取操作(编辑完毕)

作者：蔡于清
www.another-prj.com

我的板子上使用的是SAMSUNG的K9F1208U0B，下面我将对此型号的NandFlash读取操作做一个讲解。
首先我们先从物理结构上来了解这颗芯片，结构图如下所示



正如硬盘的盘片被分为磁道，每个磁道又被分为若干扇区，一块 Nand Flash 被分为若干 Block,每个 Block 又被分为若干 Page。由上图我们可以知道 flash 中

Byte(字节)，Page(页),Block（块）3 个单位之间的关系为 ：

1 Page =512 Bytes Data Field+ 16 Bytes Spare Field
1 Blcok=32 Pages

我们讨论的 K9F1208U0B 总共有 **4096 个 Blocks**，故我们可以知道这块 flash 的容量为 4096 *(32 *528)= 69206016 Bytes = 66 MB
但事实上**每个 Page 上的最后 16Bytes 是用于存贮检验码用的，并不能存放实际的数据，**所以实际上我们可以操作的芯片容量为 4096 *(32 *512) = 67108864 Bytes = 64 MB 由上图所示，1 个 Page 总共由 528 Bytes 组成，这 528 个字节按顺序由上而下以列为单位进行排列（1 列代表一个 Byte。第 0 行为第 0 Byte ，第 1 行为第 1 Byte，以此类推，每个行又由 8 个位组成，每个位表示 1 个 Byte 里面的 1bit）。这 528Bytes 按功能分为两大部分，分别是 Data Field 和 Spare Field，其中 Spare Field 占 528Bytes 里的 16Bytes,这 16Bytes 是用于在读写操作的时候存放校验码用的，一般不用做普通数据的存储区，除去这 16Bytes,剩下的 512Bytes 便是我们用于存放数据用的 Data Field，所以一个 Page 上虽然有 528 个 Bytes，但我们只按 512Bytes 进行容量的计算。
Data Field 按位置关系又可分为两个部分，分别称为 **1st half 与 2nd half，**每个 half 各占 256 个 bytes。或许你会感到纳闷，为什么要把 DataField 分为两个部分？把他们看做一个整体进行操作不就好了吗？呵呵，凡事都有因果，这么分块自然有它的道理所在，但现在还不是告诉你答案的时候。我们还是先讨论一下它的操作吧。对 K9F1208U0B 的操作是通过向 Nand Flash 命令寄存器（对于 s3c2410 来说此寄存器为 NFCMD，内存映射地址为 0x4e000004）发送命令队列进行的，为什么说是命令队列？就是因为要完成某个操作的时候发送的不是一条命令，而是连续几条命令或是一条命令加几个参数
下面是 K9F1208U0B 的操作命令集：

Table 1. Command Sets

Function	1st. Cycle	2nd. Cycle	3rd. Cycle	Acceptable Command during Busy
Read 1	00h/01h(1)	-	-	
Read 2	50h	-	-	
Read ID	90h	-	-	
Reset	FFh	-	-	O
Page Program (True)(2)	80h	10h	-	
Page Program (Dummy)(2)	80h	11h	-	
Copy-Back Program(True)(2)	00h	8Ah	10h	
Copy-Back Program(Dummy)(2)	03h	8Ah	11h	
Block Erase	60h	D0h	-	
Multi-Plane Block Erase	60h----60h	D0h	-	
Read Status	70h	-	-	O
Read Multi-Plane Status	71h(3)	-	-	O

NOTE : 1. The 00h command defines starting address of the 1st half of registers.
The 01h command defines starting address of the 2nd half of registers.
After data access on the 2nd half of register by the 01h command, the status pointer is automatically moved to the 1st half register(00h) on the next cycle.
2. Page Program(True) and Copy-Back Program(True) are available on 1 plane operation.
Page Program(Dummy) and Copy-Back Program(Dummy) are available on the 2nd,3rd,4th plane of multi plane operation.
3. The 71h command should be used for read status of Multi Plane operation.
Caution : Any undefined command inputs are prohibited except for above command set of Table 1.

读命令有两个，分别是 Read1,Read2 其中 Read1 用于读取 Data Field 的数据，而 Read2 则是用于读取 Spare Field 的数据。对于 Nand Flash 来说，读操作的最小操作单位为 Page，也就是说当我们给定了读取的起始位置后，读操作将从该位置开始，连续读取到本 Page 的最后一个 Byte 为止（可以包括 Spare Field）**Nand**

Flash 的寻址 Nand Flash 的地址寄存器把一个完整的 Nand Flash 地址分解成 Column Address 与 Page Address.进行寻址

Column Address：列地址。Column Address 其实就是指定 Page 上的某个 Byte，指定这个 Byte 其实也就是指定此页的读写起始地址。

Paage Address：页地址。由于页地址总是以 512Bytes 对齐的，所以它的低 9 位总是 0。确定读写操作是在 Flash 上的哪个页进行的。

Read1 命令

当我们得到一个 Nand Flash 地址 src_addr 时我们可以这样分解出 Column Address 和 Page Address

```
column_addr=src_addr%512;                    // column address
page_address=(src_addr>>9);                  // page address
```

也可以这么认为，一个 Nand Flash 地址的 A0~A7 是它的 column_addr，A9~A25 是它的 Page Address。(注意地址位 A8 并没有出现，也就**是 A8 被忽略**，在下面你将了解到这是什么原因)

Read1 命令的操作分为 4 个 Cycle，发送完读命令 00h 或 01h（00h 与 01h 的区别请见下文描述）之后将分 4 个 Cycle 发送参数，1st.Cycle 是发送 Column Address。2nd.Cycle ,3rd.Cycle 和 4th.Cycle 则是指定 Page Address（每次向地址寄存器发送的数据只能是 8 位，所以 17 位的 Page Address 必须分成 3 次进行发送.4 个 Cycle 见下图所示

	I/O 0	I/O 1	I/O 2	I/O 3	I/O 4	I/O 5	I/O 6	I/O 7	I/O8 to 15
1st Cycle	A0	A1	A2	A3	A4	A5	A6	A7	L*
2nd Cycle	A9	A10	A11	A12	A13	A14	A15	A16	L*
3rd Cycle	A17	A18	A19	A20	A21	A22	A23	A24	L*
4th Cycle	A25	L*	L*	L*	L*	L*	L*	L*	L*

Column Address

Row Address (Page Address)

NOTE : Column Address : Starting Address of the Register.

* L must be set to "Low".

* The device ignores any additional input of address cycles than reguired.

你是否还记得我上文提到过的 Data Field 被分为 1st half 和 2end half 两个部分？而从上面的命令集我们看到 Read1 的命令里面出现了两个命令选项，分别是 00h 和 01h。这里出现了两个读命是否令你意识到什么呢？是的，**00h 是用于读写 1st half 的命令，而 01h 是用于读取 2nd half 的命令**。现在我可以结合上图给你说明为什么 K9F1208U0B 的 DataField 被分为 2 个 half 了。

如上文我所提及的，Read1 的 1st.Cycle 是发送 Column Address，假设我现在指定的 Column Address 是 0，那么读操作将从此页的第 0 号 Byte 开始一直读取到此页的最后一个 Byte(包括 Spare Field)，如果我指定的 Column Address 是 127，情况也与前面一样，但不知道你发现没有，用于传递 Column Address 的数据线有 8 条（I/O0~I/O7，对应 A0~A7，这也是 A8 为什么不出现在我们传递的地址位中），也就是说我们能够指定的 Column Address 范围为 0~255，但不要忘了，1 个 Page 的 DataField 是由 512 个 Byte 组成的，假设现在我要指定读命令从第 256 个字节处开始读取此页，那将会发生什么情景？我必须把 Column Address 设置为 256，但 Column Address 最大只能是 255，这就造成数据溢出。。。正是因为这个原因我们才把 Data Field 分为两个半区，当要读取的起始地址（Column Address）在 0~255 内时我们用 00h 命令，当读取的起始地址是在 256~511 时，则使用 01h 命令.假设现在我要指定从第 256 个 byte 开始读取此页，那么我将这样发送命令串

```
column_addr=256;
NF_CMD=0x01;                                 从 2nd half 开始读取⇓
NF_ADDR=column_addr&0xff;                    1st Cycle
NF_ADDR=page_address&0xff;                   2nd.Cycle
NF_ADDR=(page_address>>8)&0xff;              3rd.Cycle
```

NF_ADDR=(page_address>>16)&0xff;　　4th.Cycle

其中 NF_CMD 和 NF_ADDR 分别是 NandFlash 的命令寄存器和地址寄存器的地址解引用，我一般这样定义它们，

#define rNFCMD　　(*(volatile unsigned char *)0x4e000004)　　//NADD Flash command

#define rNFADDR　　(*(volatile unsigned char *)0x4e000008)　　//NAND Flash address

事实上，当 NF_CMD=0x01 时，地址寄存器中的第 8 位（A8）将被设置为 1（如上文分析，**A8 位不在我们传递的地址中，这个位其实就是硬件电路根据 01h 或是 00h 这两个命令来置高位或是置低位**），这样我们传递 column_addr 的值 256 虽然由于数据溢出变为 1，但 A8 位已经由于 NF_CMD=0x01 的关系被置为 1 了，所以我们传到地址寄存器里的值变成了

A0 A1 A2 A3 A4 A5 A6 A7 A8

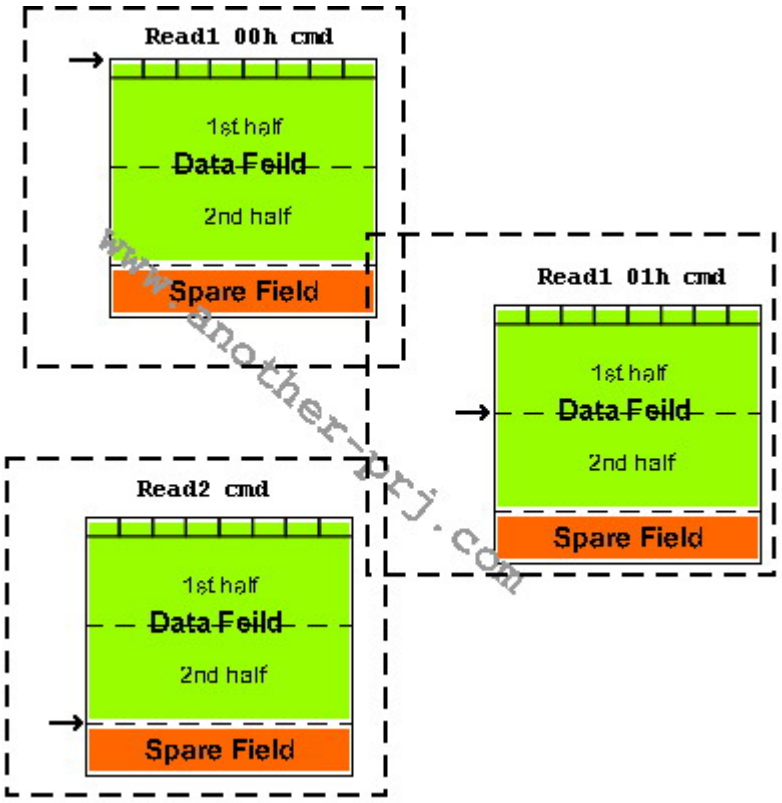
1　0　0　0　0　0　0　0　1

这 8 个位所表示的正好是 256，这样读操作将从此页的第 256 号 byte（2nd half 的第 0 号 byte）开始读取数据。

Read2

Read2 则是指定读取 Spare Field 的内容

其实 Read1 和 Read2 都是读命令，他们的区别相当于对一个读指针进行不同区域的定位。如图所示



nand_flash.c 中包含 3 个函数

void nf_reset(void);

void nf_init(void);

void nf_read(unsigned int src_addr,unsigned char *desc_addr,int size);

nf_reset()将被 nf_init()调用。nf_init()是 nand_flash 的初始化函数，在对 nand flash 进行任何操作之前，nf_init()必须被调用。

nf_read(unsigned int src_addr,unsigned char *desc_addr,int size);为读函数，src_addr 是 nand flash 上的地址，desc_addr 是内存地址，size 是读取文件的长度。

在 nf_reset 和 nf_read 函数中存在两个宏

```
NF_nFCE_L();
NF_nFCE_H();
```

你可以看到当每次对 Nand Flash 进行操作之前 NF_nFCE_L()必定被调用，操作结束之时 NF_nFCE_H()必定被调用。这两个宏用于启动和关闭 Flash 芯片的工作（片选/取消片选）。至于 nf_reset()中的

```
rNFCONF=(1<<15)|(1<<14)|(1<<13)|(1<<12)|(1<<11)|(TACLS<<8)|(TWRPH0<<4)|(TWRPH1<<0);
```

这一行代码是对 NandFlash 的控制寄存器进行初始化配置,rNFCONF 是 Nand Flash 的配置寄存器，各个位的具体功能请参阅 s3c2410 数据手册。

现在举一个例子，假设我要从 Nand Flash 中的第 5000 字节处开始读取 1024 个字节到内存的 0x30000000 处，我们这样调用 read 函数

```
nf_read(5000, 0x30000000,1024);
```

我们来分析 5000 这个 src_addr.

根据

```
column_addr=src_addr%512;
page_address=(src_addr>>9);
```

我们可得出 column_addr=5000%512=392

```
page_address=(5000>>9)=9
```

于是我们可以知道 5000 这个地址是在第 9 页的第 392 个字节处，于是我们的 nf_read 函数将这样发送命令和参数

```
column_addr=5000%512;
>page_address=(5000>>9);
NF_CMD=0x01;                                从 2nd half 开始读取
NF_ADDR= column_addr &0xff;                 1st Cycle
NF_ADDR=page_address&0xff;                  2nd.Cycle
NF_ADDR=(page_address>>8)&0xff;             3rd.Cycle
NF_ADDR=(page_address>>16)&0xff;            4th.Cycle
```

向 NandFlash 的命令寄存器和地址寄存器发送完以上命令和参数之后,我们就可以从 rNFDATA 寄存器(NandFlash 数据寄存器)读取数据了.

我用下面的代码进行数据的读取.

```
for(i=column_addr;i<512;i++)
{
        *buf++=NF_RDDATA();
}
```

每当读取完一个 Page 之后,数据指针会落在下一个 Page 的 0 号 Column(0 号 Byte).

附件里是完整的 NandFlash 读取操作代码.请配合本文使用.

s3c2410 中断异常处理(编辑完毕)

作者：蔡于清
www.another-prj.com

在进入正题之前，我想先把ARM920T的异常向量表（Exception Vectors）做一个简短的介绍。：]
ARM920T的异常向量表有两种存放方式，一种是低端存放（从 0x00000000 处开始存放），另一种是高端存放（从 0xfff000000 处开始存放）。关于为什么要分两种方式进行存放这点我将在介绍MMU的文章中进行说明,本文采用低端模式。ARM920T能处理有 8 个异常，他们分别是：
Reset，Undefined instruction，Software Interrupt，Abort (prefetch)，Abort (data)，Reserved，IRQ，FIQ
下面是某个采用低端模式的系统源码片段：

```
/*************************************************************************
_start:
b       Handle_Reset
b       HandleUndef
b       HandleSWI
b       HandlePrefetchAbort
b       HandleDataAbort
b       HandleNotUsed
b       HandleIRQ
b       HandleFIQ
.....
...
..
other codes
...
..
.
*************************************************************************/
```

上面这部分片段一般出现在一个名叫“head.s”的汇编文件的里，“b Handle_Reset”这条语句就是系统上电之后运行的第一条语句。也就是说这部分代码的二进制码必须位于内存的最开始部分（这正是低端存放模式），因为上电后CPU会从SDRAM的 0x00000000 处取第一条指令并执行。

```
Address             Instruct
0x00000000:         b       Handle_Reset
0x00000004:         b       HandleUndef
0x00000008:         b       HandleSWI
0x0000000C:         b       HandlePrefetchAbort
0x00000010:         b       HandleDataAbort
0x00000014:         b       HandleNotUsed
0x00000018:         b       HandleIRQ
```

0x0000001C:　　b　　HandleFIQ

上面是该程序段在系统上电后加载到内存后的分布情况，我们可以看到每条指令占用了 4 个字节。

上电后，PC指针会跳转到Handle_Reset处开始运行。以后系统每当有异常出现，则CPU会根据异常号，从内存的 0x00000000 处开始查表做相应的处理，比如系统触发了一个IRQ异常，IRQ为第 6 号异常，则CPU将把PC指向 0x00000018 地址（4*6=24=0x00000018）处运行，该地址的指令是跳转到“中断异常服务例程”（HandleIRQ）处运行。以上就是我对异常向量表的一个简单介绍。现在可以进入我们文章的主题“中断异常处理”，s3c2410 的中断分快中断(FIQ)和普通中断（IRQ）,我们讨论的重点是普通中断（IRQ）。

s3c2410 的中断异常处理模块总共由以下寄存器构成

SRCPND(SOURCE PENDING REGISTER)

INTMOD(INTERRUPT MODE REGISTER)

INTMSK(INTERRUPT MASK REGISTER)

PRIORITY(PRIORITY REGISTER)

INTPND(INTERRUPT PENDING REGISTER)

INTOFFSET(INTERRUPT OFFSET REGISTER)

SUBSRCPND (INTERRUPT SUB SOURCE PENDING)

INTSUBMSK (INTERRUPT SUB MASK REGISTER）

下面我将讲解每个寄存器在一个中断处理流程中所扮演的角色

SRCPND / SUBSRCPND这两个寄存器在功能上是相同的，它们是中断源引脚寄存器，在一个中断异常处理流程中，中断信号传进中断异常处理模块后首先遇到的就是**SRCPND / SUBSRCPND**,这两个寄存器的作用是用于标示出哪个中断请求被触发。**SRCPND**的有效位为 32，**SUBSRCPND** 的有效位为 11，它们中的每一位分别代表一个中断源。**SRCPND**为主中断源引脚寄存器，**SUBSRCPND**为副中断源引脚寄存器。这里列举出**SRCPND**的各个位信息：

SOURCE PENDING (SRCPND) REGISTER (Continued)

SRCPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0
INT_RTC	[30]	0 = Not requested, 1 = Requested	0
INT_SPI1	[29]	0 = Not requested, 1 = Requested	0
INT_UART0	[28]	0 = Not requested, 1 = Requested	0
INT_IIC	[27]	0 = Not requested, 1 = Requested	0
INT_USBH	[26]	0 = Not requested, 1 = Requested	0
INT_USBD	[25]	0 = Not requested, 1 = Requested	0
Reserved	[24]	Not used	0
INT_UART1	[23]	0 = Not requested, 1 = Requested	0
INT_SPI0	[22]	0 = Not requested, 1 = Requested	0
INT_SDI	[21]	0 = Not requested, 1 = Requested	0
INT_DMA3	[20]	0 = Not requested, 1 = Requested	0
INT_DMA2	[19]	0 = Not requested, 1 = Requested	0
INT_DMA1	[18]	0 = Not requested, 1 = Requested	0
INT_DMA0	[17]	0 = Not requested, 1 = Requested	0
INT_LCD	[16]	0 = Not requested, 1 = Requested	0
INT_UART2	[15]	0 = Not requested, 1 = Requested	0
INT_TIMER4	[14]	0 = Not requested, 1 = Requested	0
INT_TIMER3	[13]	0 = Not requested, 1 = Requested	0
INT_TIMER2	[12]	0 = Not requested, 1 = Requested	0
INT_TIMER1	[11]	0 = Not requested, 1 = Requested	0
INT_TIMER0	[10]	0 = Not requested, 1 = Requested	0
INT_WDT	[9]	0 = Not requested, 1 = Requested	0
INT_TICK	[8]	0 = Not requested, 1 = Requested	0
nBATT_FLT	[7]	0 = Not requested, 1 = Requested	0
Reserved	[6]	Not used	0
EINT8_23	[5]	0 = Not requested, 1 = Requested	0
EINT4_7	[4]	0 = Not requested, 1 = Requested	0
EINT3	[3]	0 = Not requested, 1 = Requested	0
EINT2	[2]	0 = Not requested, 1 = Requested	0
EINT1	[1]	0 = Not requested, 1 = Requested	0
EINT0	[0]	0 = Not requested, 1 = Requested	0

每个位的初始值皆为 0。假设现在系统触发了 TIMER0 中断，则第 10bit 将被置 1，代表 TIMER0 中断被触发，该中断请求即将被处理（若该中断没有被屏蔽的话）。**SUBSRCPND** 情况与 **SRCPND** 相同，这里就不多讲了。

INTMOD 寄存器有效位为 32 位，每一位与 **SRCPND** 中各位相对应，它的作用是指定该位相应的中断源处理模式（IRQ 还是 FIQ）。若某位为 0，则该位相对应的中断按 IRQ 模式处理，为 1 则以 FIQ 模式进行处理，该寄存器初始化值为 0x00000000,即所有中断皆以 IRQ 模式进行处理。（详细请参考 s3c2410 操作手册）。

INTMSK/ INTSUBMSK 寄存器为中断屏蔽寄存器 ， **INTMSK** 为主中断屏蔽寄存器，**INTSUBMSK** 为副中断屏蔽寄存器。**INTMSK** 有效位为 32，**INTSUBMSK** 有效位为 11，这两个寄存器各个位与 **SRCPND** 和 **SUBSRCPND** 分别对应。它们的作用是决定该位相应的中断请求是否被处理。若某位被设置为 1，则该位相对应的中断产生后将被忽略（CPU 不处理该中断请求），设置为 0 则对其进行处理。这两个寄存器初始化后的值是 0xFFFFFFFF 和 0x7FF，既默认情况下所有的中断都是被屏蔽的。

到目前为止我们总共讲解了 **SRCPND，INTMOD，INTMSK，SUBSRCPND，INTSUBMSK** 五个寄存器，在继续讲解 PRIORITY 寄存器之前我们先来看一张图。

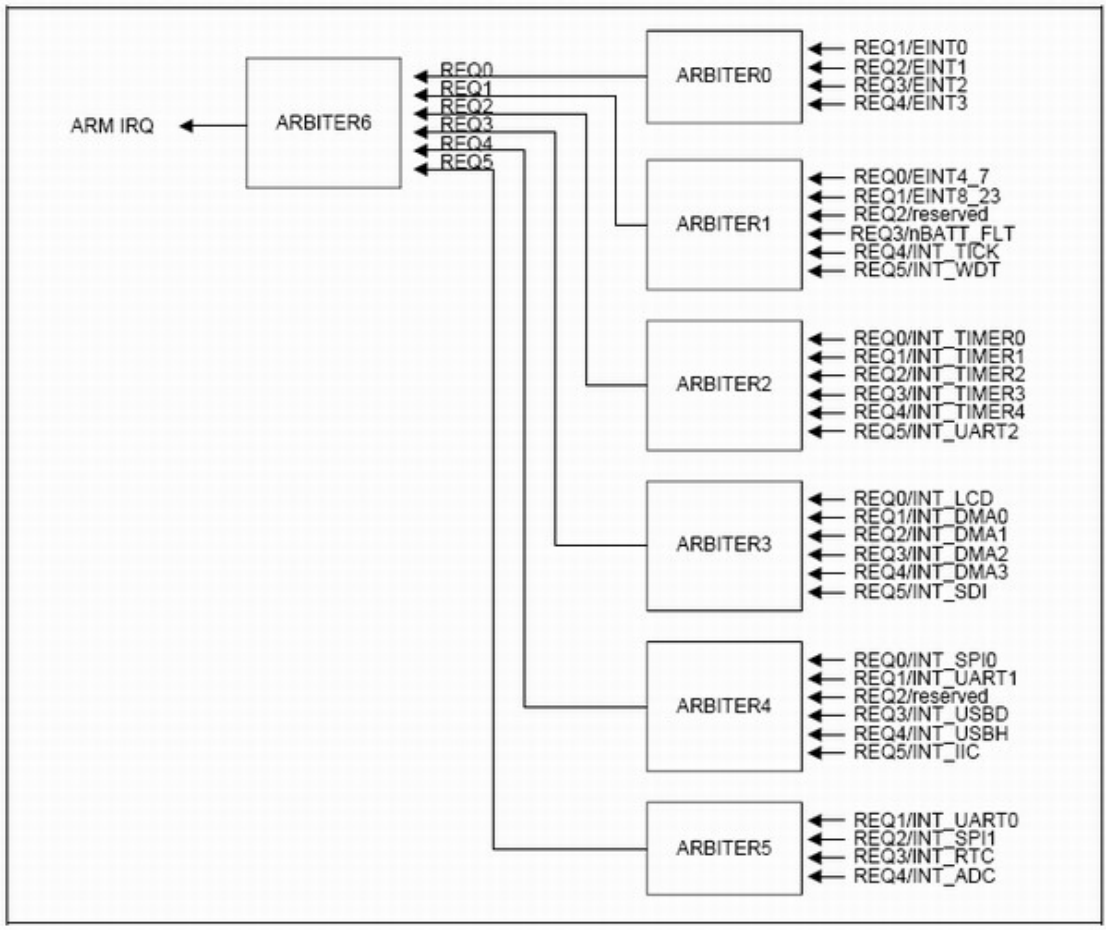


Figure 14-2. Priority Generating Block

先弄清楚一点，现在要讨论的是一个中断优先级的判断问题。为什么会有中断有先级的问题呢？我们知道 CPU 某个时刻只能对一个中断源进行中断处理，如果现在有 3 个中断同时发生了，那 CPU 要按什么顺序处理这个 3 个中断呢？这正是引入优先级判断的原因所在，通过优先级判断，CPU 可以按某种顺序逐个处理中断请求。3sc2410 的优先级判断分为两级。

如上图所示，**SRCPND** 寄存器对应的 32 个中断源总共被分为 6 个组，每个组由一个 **ARBITER（0~5）** 寄存器对其进行管理。中断必须先由所属组的 **ARBITER（0~5）** 进行第一次优先级判断（第一级判断）后再发往 **ARBITER6** 进行最终的判断（第二级判断）。**ARBITER（0~5）** 这六个组的优先级已经固定，我们无法改变，也就是说由 **ARBITER0** 控制的该组中断优先级最高（该组产生的中断进行第一级判断后永远会以 REQ0 向 **ARBITER6** 传递过去）其次是 **ARBITER1, ARBITER2, ARBITER4, ARBITER4, ARBITER5**.我们能够控制的是某个组里面各个中断的优先级顺序。怎么控制？通过 **PRIORITY** 寄存器进行控制：]

以下是 PRIORITY 寄存器各个位的参数表

PRIORITY	Bit	Description	Initial State
ARB_SEL6	[20:19]	Arbiter 6 group priority order set 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5	0
ARB_SEL5	[18:17]	Arbiter 5 group priority order set 00 = REQ 1-2-3-4 01 = REQ 2-3-4-1 10 = REQ 3-4-1-2 11 = REQ 4-1-2-3	0
ARB_SEL4	[16:15]	Arbiter 4 group priority order set 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5	0
ARB_SEL3	[14:13]	Arbiter 3 group priority order set 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5	0
ARB_SEL2	[12:11]	Arbiter 2 group priority order set 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5	0
ARB_SEL1	[10:9]	Arbiter 1 group priority order set 00 = REQ 0-1-2-3-4-5 01 = REQ 0-2-3-4-1-5 10 = REQ 0-3-4-1-2-5 11 = REQ 0-4-1-2-3-5	0
ARB_SEL0	[8:7]	Arbiter 0 group priority order set 00 = REQ 1-2-3-4 01 = REQ 2-3-4-1 10 = REQ 3-4-1-2 11 = REQ 4-1-2-3	0
ARB_MODE6	[6]	Arbiter 6 group priority rotate enable 0 = Priority does not rotate, 1 = Priority rotate enable	1
ARB_MODE5	[5]	Arbiter 5 group priority rotate enable 0 = Priority does not rotate, 1 = Priority rotate enable	1
ARB_MODE4	[4]	Arbiter 4 group priority rotate enable 0 = Priority does not rotate, 1 = Priority rotate enable	1
ARB_MODE3	[3]	Arbiter 3 group priority rotate enable 0 = Priority does not rotate, 1 = Priority rotate enable	1
ARB_MODE2	[2]	Arbiter 2 group priority rotate enable 0 = Priority does not rotate, 1 = Priority rotate enable	1

从表上我们可以知道 **PRIORITY** 寄存器内部各个位被分为两种类型，一种是 **ARB_MODE**,另一种为 **ARB_SEL, ARB_MODE** 类型有 5 组对应 **ARBITER（2~6）,ARB_SEL** 类型有 7 组对应 **ARBITER（0~6）。** 现在我将以 **ARBITER2** 为例，讲解中断组与 **PRIORITY** 寄存器中 **ARB_SEL, ARB_MODE** 之间的相互关系。

首先我们看到 ARBITER2 寄存器管理的该组中断里包括了 6 个中断，分别是 INT_TIMER0，INT_TIMER1，INT_TIMER2，INT_TIMER3，INT_TIMER4，INT_UART2，她们的默认中断请求号分别为 REQ0，REQ1，REQ2，REQ3，REQ4，REQ5。我们先看 **PRIORITY** 寄存器中的 **ARB_SEL2**，该参数由两个位组成，初始值为 00。从该表可以看出 00 定义了一个顺序 0-1-2-3-4-5 ，这个顺序就是这组中断组的优先级排列，这个顺序指明了以中断请求号为 0（REQ0）的 INT_TIMER0 具有最高的中断优先级，其次是 INT_TIMER1，INT_TIMER2...。假设现在 **ARB_SEL2** 的值被我们设置为 01。则一个新的优先级次序将被使用，01 对应的优先级次序为 0-2-3-4-1-5，从中可以看出优先级最高和最低的中断请求和之前没有变化，但本来处于第 2 优先级的 INT_TIMER1 中断现在变成了第 5 优先级。从 **ARB_SEL2** 被设置为 00,01,10,11 各个值所出现的情况我们可以看出，除最高和最低的优先级不变以外，其他各个中断的优先级其实是在做一个旋转排列（rotate）。为了达到对各个中断平等对待这一目标，我们可以让优先级次序在每个中断请求被处理完之后自动进行一次旋转，如何自动让它旋转呢？我们可以通过 **ARB_MODE2** 达到这个目的，该参数只有 1 个 bit，置 1 代表开启对应中断组的优先级次序旋转，0 则为关闭。事实上当该位置为 1 之后，每处里完某个组的一个中断后，该组的 **ARB_SEL** 便递增在 1（达到 11 后恢复为 00）。

现在我们另 **ARB_MODE2=1，ARB_SEL2=00**

则当前 **ARBITER2** 的优先级顺序为 0-1-2-3-4-5，假设现在该组的 1 号中断请求 INT_TIMER1 和 2 号中断请求 INT_TIMER2 被同时触发，CPU 根据优先级判断后决定先把 INT_TIMER1 中断向 **ARBITER6** 进行发送（在 ARBITER6 做第最终优先级判断），接着再向 **ARBITER6** 发送 INT_TIMER2 中断。请注意，在 INT_TIMER1 被处理完毕后，该组中段的优先级次序被自动做了一次旋转，旋转后 **ARBITER2** 的优先级顺序变为 0-2-3-4-1-5。假设之后某个时刻该组的 INT_TIMER1 和 INT_TIMER2 又被同时触发，则此时 CPU 优先处理的会是 INT_TIMER2。若我们另 ARB_MODE2=0，则改组的中断优先级次序在任何情况下都不做任何改变，除非我们人为地重新设置了 ARB_SEL2 的值。

呼。。。好累。。。终于说完了麻烦的优先级-_-...

继续。。。

INTPND 寄存器可能是整个中断处理过程中我们要特别注意的一个寄存器了，他的操作比较特别，怎么特别？请听我慢慢道来.：]

先看一下该寄存器各位详细功能列表

INTPND	Bit	Description	Initial State
INT_ADC	[31]	0 = Not requested, 1 = Requested	0
INT_RTC	[30]	0 = Not requested, 1 = Requested	0
INT_SPI1	[29]	0 = Not requested, 1 = Requested	0
INT_UART0	[28]	0 = Not requested, 1 = Requested	0
INT_IIC	[27]	0 = Not requested, 1 = Requested	0
INT_USBH	[26]	0 = Not requested, 1 = Requested	0
INT_USBD	[25]	0 = Not requested, 1 = Requested	0
Reserved	[24]	Not used	0
INT_UART1	[23]	0 = Not requested, 1 = Requested	0
INT_SPI0	[22]	0 = Not requested, 1 = Requested	0
INT_SDI	[21]	0 = Not requested, 1 = Requested	0
INT_DMA3	[20]	0 = Not requested, 1 = Requested	0
INT_DMA2	[19]	0 = Not requested, 1 = Requested	0
INT_DMA1	[18]	0 = Not requested, 1 = Requested	0
INT_DMA0	[17]	0 = Not requested, 1 = Requested	0
INT_LCD	[16]	0 = Not requested, 1 = Requested	0
INT_UART2	[15]	0 = Not requested, 1 = Requested	0
INT_TIMER4	[14]	0 = Not requested, 1 = Requested	0
INT_TIMER3	[13]	0 = Not requested, 1 = Requested	0
INT_TIMER2	[12]	0 = Not requested, 1 = Requested	0
INT_TIMER1	[11]	0 = Not requested, 1 = Requested	0
INT_TIMER0	[10]	0 = Not requested, 1 = Requested	0
INT_WDT	[9]	0 = Not requested, 1 = Requested	0
INT_TICK	[8]	0 = Not requested, 1 = Requested	0
nBATT_FLT	[7]	0 = Not requested, 1 = Requested	0
Reserved	[6]	Not used	0
EINT8_23	[5]	0 = Not requested, 1 = Requested	0
EINT4_7	[4]	0 = Not requested, 1 = Requested	0
EINT3	[3]	0 = Not requested, 1 = Requested	0
EINT2	[2]	0 = Not requested, 1 = Requested	0
EINT1	[1]	0 = Not requested, 1 = Requested	0
EINT0	[0]	0 = Not requested, 1 = Requested	0

正如你所见的，**INTPND** 寄存器与 **SRCPND** 长得一模一样，但他们在中断异常处理中却扮演着不同的角色，如果说 **SRCPND** 是中断信号进入中断处理模块后所经过的第一个场所的话，那么 **INTPND** 则是中断信号在中断处理模块里经历的最后一个寄存器。它的每个位对应一个中断请求，若该位被置 1，则表示相应的中断请求被触发，描述到这里你可能会发现它不仅和 **SRCPND** 长得一模一样，就连功能都一样，其

实不然，他们在功能上有着重大的区别。SRCPND 是中断源引脚寄存器，某个位被置 1 表示相应的中断被触发，但我们知道在同一时刻内系统可以触发若干个中断，只要中断被触发了，**SRCPND** 的相应位便被置 1，也就是说 **SRCPND** 在同一时刻可以有若干位同时被置 1，然而 **INTPND** 则不同，他在某一时刻只能有 1 个位被置 1，INTPND 某个位被置 1（该位对应的中断在所有已触发的中断里具有最高优先级且该中断没有被屏蔽），则表示 CPU 即将或已经在对该位相应的中断进行处理。于是我们可以有一个总结：SRCPND 说明了有什么中断被触发了，INTPND 说明了 CPU 即将或已经在对某一个中断进行处理。

特别注意：每当某一个中断被处理完之后，我们必须手动地把 **SRCPND/SUBSRCPND，INTPND** 三个寄存器中与该中断相应的位由 1 设置为 0，刚才我说 **INTPND** 的操作很特别，它的特别之处就在于对当我们要把该寄存器中某个值为 1 的位设置为 0 时，我们不是往该位置 0，而是往该位置 1。假设 **SRCPND=0x00000003，INTPND=0x00000001**,该值说明当前 0 号中断和 1 号中断被触发，但当前正在被处理的是 0 号中断，处理完毕后我们应该这样设置 **INTPND 和 SRCPND**：

SRCPND=0x00000002　　//位 0 被置为 0

INTPND =0x00000001　　//位 0 被置为 0（方法是往该位写入 1）

INTOFFSET 寄存器的功能则很简单，它的作用只是用于表明哪个中断正在被处理。下面是该寄存器各位详细功能列表

INT Source	The OFFSET Value	INT Source	The OFFSET Value
INT_ADC	31	INT_UART2	15
INT_RTC	30	INT_TIMER4	14
INT_SPI1	29	INT_TIMER3	13
INT_UART0	28	INT_TIMER2	12
INT_IIC	27	INT_TIMER1	11
INT_USBH	26	INT_TIMER0	10
INT_USBD	25	INT_WDT	9
Reserved	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	Reserved	6
INT_SDI	21	EINT8_23	5
INT_DMA3	20	EINT4_7	4
INT_DMA2	19	EINT3	3
INT_DMA1	18	EINT2	2
INT_DMA0	17	EINT1	1
INT_LCD	16	EINT0	0

若当前 INT_TIMER0 被触发了，则该寄存器的值为 10，以此类推。

现在我把整个中断流程用一个图加以说明

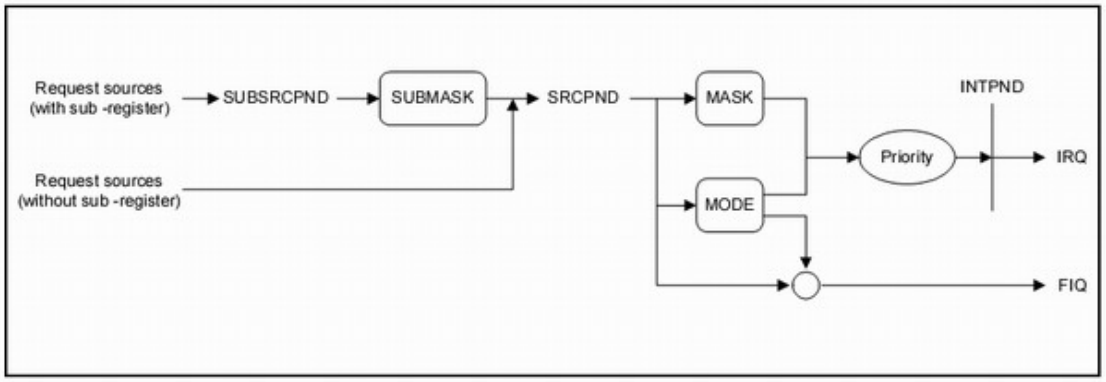


Figure 14-1. Interrupt Process Diagram

[attach]21[/attach]
以上这个图清楚地说明了一个中断异常处理流程。
下面我用 INT_TIMER0, INT_TIMER2 和 INT_UART0 三个中断完整地介绍一次中断异常处理。首先我们得做几个假设：
假设 1：这三个中断的屏蔽被取消。
假设 2：**PRIORITY** 寄存器中 ARB_MODE2，ARB_MODE5 皆为 0，既不进行优先级的自动旋转排序，任何时候
ARBITER2，ARBITER5 控制的中断组优先级次序分别为 0-1-2-3-4-5 和 1-2-3-4。
假设 3：这三个中断皆为 IRQ 类型。
假设 4：这三个中断同时被触发。
INT_TIMER0, INT_TIMER2 和 INT_UART0 三个中断被同时触发，此时三个中断信号流向 **SRCPND** 寄存器，使该寄存器中的第 10 位，12 位，28 位被置为 1，中断信号继续向前流经 **INTMASK** 寄存器，这三个中断都没有被屏蔽，于是信号进一步流经 **INTMODE** 寄存器，这三个中断皆为 IRQ 类型，故中断信号继续向前流向 **PRIORITY** 寄存器，经过优先级判断，INT_TIMER0 中断信号使 **INTPND** 寄存器的第 10 位置 1（INT_TIMER0 优先级最高），此时 **INTOFFSET** 寄存器的值为 10，CPU 转向相应的中断服务例程进行处理。处理完毕后，我们的程序将 **INTPND** 和 **SRCPND** 的第 10 置为 0，至此 INT_TIMER0 中断处理完毕。此时 **SRCPND** 的第 12 位，28 位仍为 1（这两个中断请求未被处理），故他们会继续被 CPU 已刚才描述的方式进行处理。
中断异常处理就先讲到这吧 ：]

s3c2410 MMU (编辑完毕)

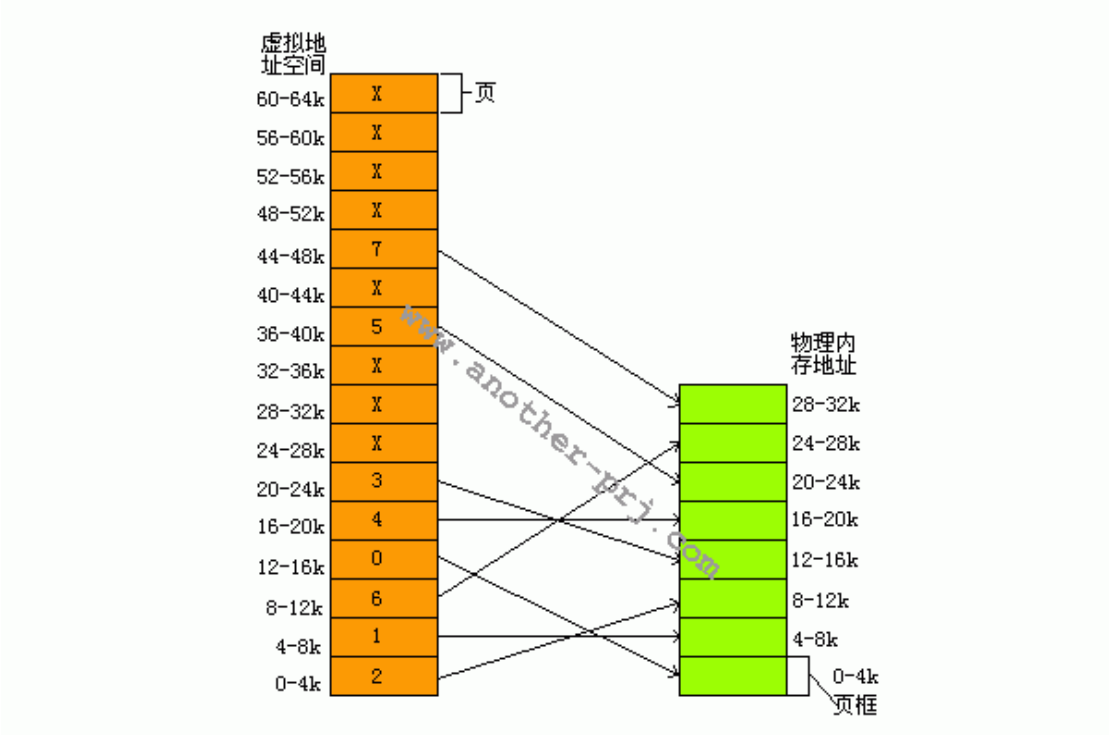
作者：蔡于清
www.another-prj.com

MMU,全称Memory Manage Unit，中文名——存储器管理单元。
许多年以前，当人们还在使用DOS或是更古老的操作系统的时候，计算机的内存还非常小，一般都是以K为单位进行计算，相应的，当时的程序规模也不大，所以内存容量虽然小，但还是可以容纳当时的程序。但随着图形界面的兴起还用用户需求的不断增大，应用程序的规模也随之膨胀起来，终于一个难题出现在程序员的面前，那就是应用程序太大以至于内存容纳不下该程序，通常解决的办法是把程序分割成许多称为**覆盖块（overlay）**的片段。覆盖块 0 首先运行，结束时他将调用另一个覆盖块。虽然覆盖块的交换是由OS完成的，但是必须先由程序员把程序先进行分割，这是一个费时费力的工作，而且相当枯燥。人们必须找到更好的办法从根本上解决这个问题。不久人们找到了一个办法，这就是**虚拟存储器(virtual memory).**虚拟存储器的基本思想是程序，数据，堆栈的总的大小可以超过物理存储器的大小，操作系统把当前使用的部分保留在内存中，而把其他未被使用的部分保存在磁盘上。比如对一个 16MB的程序和一

个内存只有 4MB的机器，OS通过选择，可以决定各个时刻将哪 4M的内容保留在内存中，并在需要时在内存和磁盘间交换程序片段，这样就可以把这个 16M的程序运行在一个只具有 4M内存机器上了。而这个 16M的程序在运行前不必由程序员进行分割。

任何时候，计算机上都存在一个程序能够产生的地址集合，我们称之为**地址范围**。这个范围的大小由CPU的位数决定，例如一个 32 位的CPU，它的地址范围是 0~0xFFFFFFFF (4G),而对于一个 64 位的CPU，它的地址范围为 0~0xFFFFFFFFFFFFFFFF (64T).这个范围就是我们的程序能够产生的地址范围，我们把这个地址范围称为**虚拟地址空间**，该空间中的某一个地址我们称之为**虚拟地址**。与虚拟地址空间和虚拟地址相对应的则是**物理地址空间**和**物理地址**，大多数时候我们的系统所具备的物理地址空间只是虚拟地址空间的一个子集，这里举一个最简单的例子直观地说明这两者，对于一台内存为 256MB的 32bit x86 主机来说，它的虚拟地址空间范围是 0~0xFFFFFFFF（4G),而物理地址空间范围是 0x000000000~0x0FFFFFFF（256MB）。在没有使用虚拟存储器的机器上，虚拟地址被直接送到内存总线上，使具有相同地址的物理存储器被读写。而在使用了虚拟存储器的情况下，虚拟地址不是被直接送到内存地址总线上，而是送到内存管理单元——MMU（主角终于出现了：]）。他由一个或一组芯片组成，一般存在与协处理器中，其功能是把虚拟地址映射为物理地址。

大多数使用虚拟存储器的系统都使用一种称为**分页（paging）**。虚拟地址空间划分成称为**页（page）**的单位,而相应的物理地址空间也被进行划分，单位是页**框(frame).**页和页框的大小必须相同。接下来配合图片我以一个例子说明页与页框之间在MMU的调度下是如何进行映射的



在这个例子中我们有一台可以生成 16 位地址的机器，它的虚拟地址范围从 0x0000~0xFFFF(64K),而这台机器只有 32K 的物理地址，因此他可以运行 64K 的程序，但该程序不能一次性调入内存运行。这台机器必须有一个达到可以存放 64K 程序的外部存储器（例如磁盘或是 FLASH）,以保证程序片段在需要时可以被调用。在这个例子中，页的大小为 4K,页框大小与页相同（这点是必须保证的，内存和外围存储器之间的传输总是以页为单位的），对应 64K 的虚拟地址和 32K 的物理存储器，他们分别包含了 16 个页和 8 个页框。我们先根据上图解释一下分页后要用到的几个术语，在上面我们已经接触了**页**和**页框**，上图中绿色部分是物理空间，其中每一格表示一个物理页框。橘黄色部分是虚拟空间，每一格表示一个页，它由两部分组成，分别是 **Frame Index(页框索引)**和**位 p（present 存在位）**，Frame Index 的意义很明显，它指出本页

是往哪个物理页框进行映射的，位 p 的意义则是指出本页的映射是否有效，如上图，当某个页并没有被映射时（或称“映射无效”，Frame Index 部分为 X），该位为 0，映射有效则该位为 1。

我们执行下面这些指令（本例子的指令不针对任何特定机型，都是伪指令）

例 1：

MOVE REG,0 //将 0 号地址的值传递进寄存器 REG.

虚拟地址 0 将被送往 MMU,MMU 看到该虚地址落在页 0 范围内（页 0 范围是 0 到 4095），从上图我们看到页 0 所对应（映射）的页框为 2（页框 2 的地址范围是 8192 到 12287），因此 MMU 将该虚拟地址转化为物理地址 8192，并把地址 8192 送到地址总线上。内存对 MMU 的映射一无所知，它只看到一个对地址 8192 的读请求并执行它。MMU 从而把 0 到 4096 的虚拟地址映射到 8192 到 12287 的物理地址。

例 2：

MOVE REG,8192

被转换为

MOVE REG,24576

因为虚拟地址 8192 在页 2 中，而页 2 被映射到页框 6（物理地址从 24576 到 28671）

例 3：

MOVE REG,20500

被转换为

MOVE REG,12308

虚拟地址 20500 在虚页 5（虚拟地址范围是 20480 到 24575）距开头 20 个字节处，虚页 5 映射到页框 3（页框 3 的地址范围是 12288 到 16383），于是被映射到物理地址 12288+20=12308。

通过适当的设置 MMU，可以把 16 个虚页隐射到 8 个页框中的任何一个，但是这个方法并没有有效的解决虚拟地址空间比物理地址空间大的问题。从上图中我们可以看到，我们只有 8 个页框（物理地址），但我们有 16 个页（虚拟地址），所以我们只能把 16 个页中的 8 个进行有效的映射。我们看看例 4 会发生什么情况

MOV REG,32780

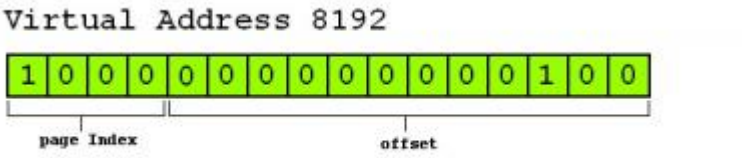
虚拟地址 32780 落在页 8 的范围内，从上图总我们看到页 8 没有被有效的进行映射（该页被打上 X），这是又会发生什么？MMU 注意到这个页没有被映射，于是通知 CPU 发生一个**缺页故障（page fault）.**这种情况下操作系统必须处理这个页故障，它必须从 8 个物理页框中找到 1 个当前很少被使用的页框并把该页框的内容写入外围存储器（这个动作被称为 **page copy**），随后把需要引用的页（例 4 中是页 8）映射到刚才释放的页框中（这个动作称为修改映射关系），然后从新执行产生故障的指令（MOV REG,32780）。假设操作系统决定释放页框 1，那么它将把虚页 8 装入物理地址的 4-8K,并做两处修改：首先把标记虚页 1 未被映射（原来虚页 1 是被影射到页框 1 的），以使以后任何对虚拟地址 4K 到 8K 的访问都引起页故障而使操作系统做出适当的动作（这个动作正是我们现在在讨论的），其次他把虚页 8 对应的页框号由 X 变为 1，因此重新执行 MOV REG,32780 时，MMU 将把 32780 映射为 4108。

我们大致了解了 MMU 在我们的机器中扮演了什么角色以及它基本的工作内容是什么，下面我们将举例子说明它究竟是如何工作的（注意，本例中的 MMU 并无针对某种特定的机型，它是所有 MMU 工作的一个抽象）。

首先明确一点，MMU 的主要工作只有一个，就是把虚拟地址映射到物理地址。

我们已经知道，大多数使用虚拟存储器的系统都使用一种称为分页（paging）的技术，就象我们刚才所举的例子，虚拟地址空间被分成大小相同的一组页，每个页有一个用来标示它的**页号**（这个页号一般是它在该组中的索引，这点和 C/C++中的数组相似）。在上面的例子中 0~4K 的页号为 0，4~8K 的页号为 1，8~12K 的页号为 2，以此类推。而虚拟地址（注意：是一个确定的地址，不是一个空间）被 MMU 分为 2 个部分，第一部分是**页号索引（page Index），**第二部分则是相对该页首地址的**偏移量（offset）.** 。我们还是以刚才那个 16 位机器结合下图进行一个实例说明，该实例中，虚拟地址 8196 被送进 MMU,MMU 把它映射成

物理地址。16 位的 CPU 总共能产生的地址范围是 0~64K,按每页 4K 的大小计算，该空间必须被分成 16 个页。而我们的虚拟地址第一部分所能够表达的范围也必须等于 16（这样才能索引到该页组中的每一个页），也就是说这个部分至少需要 4 个 bit。一个页的大小是 4K(4096),也就是说偏移部分必须使用 12 个 bit 来表示(2^12=4096，这样才能访问到一个页中的所有地址),8196 的二进制码如下图所示：



该地址的页号索引为 0010（二进制码），既索引的页为页 2，第二部分为 000000000100（二进制），偏移量为 4。页 2 中的页框号为 6（页 2 映射在页框 6，见上图），我们看到页框 6 的物理地址是 24~28K。于是 MMU 计算出虚拟地址 8196 应该被映射成物理地址 24580（页框首地址+偏移量=24576+4=24580）。同样的，若我们对虚拟地址 1026 进行读取，1026 的二进制码为 0000010000000010，page index=0000=0,offset=010000000010=1026。页号为 0，该页映射的页框号为 2，页框 2 的物理地址范围是 8192~12287，故 MMU 将虚拟地址 1026 映射为物理地址 9218（页框首地址+偏移量=8192+1026=9218）以上就是 MMU 的工作过程。

下面我们针对 s3c2410 的 MMU(注 1)进行讲解。

S3c2410 总共有 4 种内存映射方式，分别是：

1．Fault (无映射)

2．Coarse Page (粗表)

3．Section (段)

4．Fine Page (细表)

我们以 **Section(段)**进行说明。

ARM920T 是一个 32bit 的 CPU,它的虚拟地址空间为 2^32=4G。而在 **Section 模式**，这 4G 的虚拟空间被分成一个一个称为**段（Section）**的单位(与我们上面讲的页在本质上其实是一致的),每个段的长度是 1M (而我们之前所使用的页的长度是 4K)。4G 的虚拟内存总共可以被分成 4096 个段（1M*4096=4G）,因此我们必须用 4096 个描述符来对这组段进行描述，每个描述符占用 4 个 Byte,故这组描述符的大小为 16KB (4K*4096),这 4096 个描述符构为一个表格，我们称其为 **Tralaton Table.**

上图是描述符的结构

Section base address:段基地址（相当于页框号首地址）

AP: 访问控制位 Access Permission

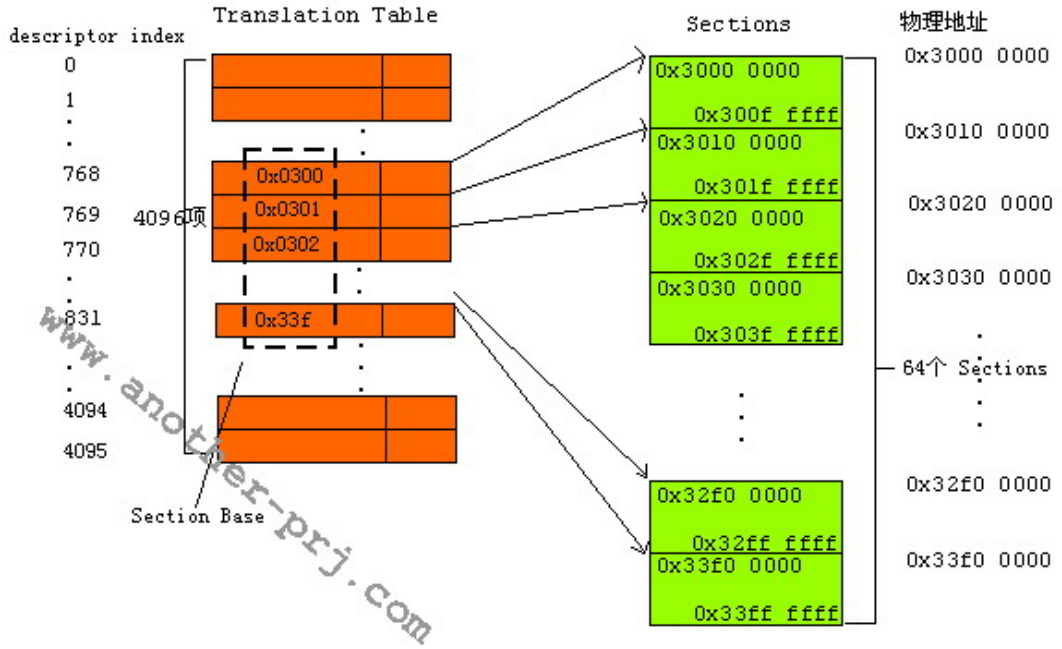
Domain: 访问控制寄存器的索引。Domain 与 AP 配合使用，对访问权限进行检查

C:当 C 被置 1 时为 write-through (WT)模式

B: 当 B 被置 1 时为 write-back (WB)模式

（C,B 两个位在同一时刻只能有一个被置 1）

下面是 s3c2410 内存映射后的一个示意图：



我的 s3c2410 上配置的 SDRSAM 大小为 64M,该 SDRAM 的物理地址范围是 0x3000 0000~0x33FF FFFF(属于 Bank 6)，由于 1 个 Section 的大小是 1M,所以该物理空间可以被分成 64 个物理段(页框).

在 Section 模式下，送进 MMU 的虚拟地址**(注 1)**被分为两部分（这点和我们上面举的例子是一样的），这两部分为 **Descriptor Index**(相当于上面例子的 Page Index)和 **Offset**,descript index 长度为 12bit(2^12=4096,从这个关系式你能看出什么？:)),Offset 长度为 20bit(2^20=1M,你又能看出什么？:)).

观察一下一个描述符（Descriptor）中的 Section Base Address 部分，它长度为 12 bit，里面的值是该虚拟段（页）映射成的物理段（页框）的物理地址前 12bit，由于每一个物理段的长度都是 1M，所以物理段首地址的后 20bit 总是为 0x00000(每个 Section 都是以 1M 对齐)，确定一个物理地址的方法是 物理页框基地址+虚拟地址中的偏移部分=Section Base Address<<20+Offset ,呵呵，可能你有点糊涂了，还是举一个实际例子说明吧。假设现在执行指令

MOV REG, 0x30000012

虚拟地址的二进制码为 00110000 00000000 00000000 00010010

前 12 位是 Descriptor Index= 00110000 0000=768,故在 Translation Table 里面找到第 768 号描述符，该描述的 Section Base Address=0x0300,也就是说描述符所描述的虚拟段（页）所映射的物理段（页框）的首地址为 0x3000 0000（物理段（页框）的基地址=Section Base Address 左移 20bit=0x0300<<20=0x3000 0000），而 Offset=000000 00000000 00010010=0x12,故虚拟地址 0x30000012 映射成的物理地址=0x3000 0000+0x12=0x3000 0012（物理页框基地址+虚拟地址中的偏移）。你可能会问怎么这个虚拟地址和映射后的物理地址一样？这是由我们定义的映射规则所决定的。在这个例子中我们定义的映射规则是把虚拟地址映射成和他相等的物理地址。我们这样书写映射关系的代码：

```
void mem_mapping_linear(void)
{
    unsigned long descriptor_index, section_base, sdram_base, sdram_size;
    sdram_base=0x30000000;
    sdram_size=0x 4000000;
    for (section _base= sdram_base,descriptor_index = section _base>>20;
```

```
        section _base < sdram_base+ sdram_size;
        descriptor_index+=1;section _base +=0x100000)
    {
        *(mmu_tlb_base + (descriptor_index)) = (section _base>>20) | MMU_OTHER_SECDESC;
    }
}
```

上面的这段段代码把虚拟空间 0x3000 0000~0x33FF FFFF 映射到物理空间 0x3000 0000~0x33FF FFFF，由于虚拟空间与物理空间空间相吻合，所以虚拟地址与他们各自对应的物理地址在值上是一致的。当初始完 Translation Table 之后，记得要把 Translation Table 的首地址(第 0 号描述符的地址)加载进协处理器 CP15 的 Control Register2(2 号控制寄存器)中,该控制寄存器的名称叫做 Translation table base (TTB) register。以上讨论的是 descriptor 中的 Section Base Address 以及虚拟地址和物理地址的映射关系,然而 MMU 还有一个重要的功能，那就是**访问控制机制(Access Permission)。**

简单说访问控制机制就是 CPU 通过某种方法判断当前程序对内存的访问是否合法（是否有权限对该内存进行访问），如果当前的程序并没有权限对即将访问的内存区域进行操作，则 CPU 将引发一个异常，s3c2410 称该异常为 **Permission fault**，x86 架构则把这种异常称之为**通用保护异常（General Protection），**什么情况会引起 Permission fault 呢？比如处于 User 级别的程序要对一个 System 级别的内存区域进行写操作，这种操作是越权的，应该引起一个 Permission fault，搞过 x86 架构的朋友应该听过**保护模式（Protection Mode）,**保护模式就是基于这种思想进行工作的，于是我们也可以这么说：s3c2410 的访问控制机制其实就是一种保护机制。那 s3c2410 的访问控制机制到底是由什么元素去参与完成的呢？它们间是怎么协调工作的呢？这些元素总共有：

1．协处理器 CP15 中 Control Register3：DOMAIN ACCESS CONTROL REGISTER

2．段描述符中的 AP 位和 Domain 位

3．协处理器 CP15 中 Control Register1(控制寄存器 1)中的 S bit 和 R bit

4．协处理器 CP15 中 Control Register5(控制寄存器 5)

5．协处理器 CP15 中 Control Register6(控制寄存器 6)

DOMAIN ACCESS CONTROL REGISTER 是**访问控制寄存器**，该寄存器有效位为 32，被分成 16 个区域，每个区域由两个位组成，他们说明了当前内存的访问权限检查的级别，如下图所示：

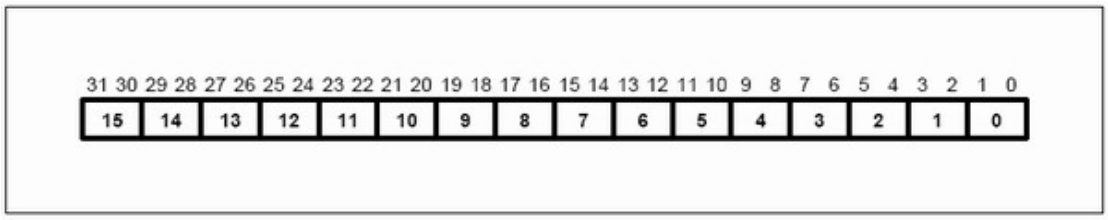


Figure 3-10. Domain Access Control Register Format

每区域可以填写的值有 4 个，分别为 00,01,10,11(二进制)，他们的意义如下所示：

Table 3-5. Interpreting Access Control Bits in Domain Access Control Register

Value	Meaning	Notes
00	No Access	Any access will generate a domain fault.
01	Client	Accesses are checked against the access permission bits in the section or page descriptor.
10	Reserved	Reserved. Currently behaves like the no access mode.
11	Manager	Accesses are *not* checked against the access permission bits so a permission fault cannot be generated.

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

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。

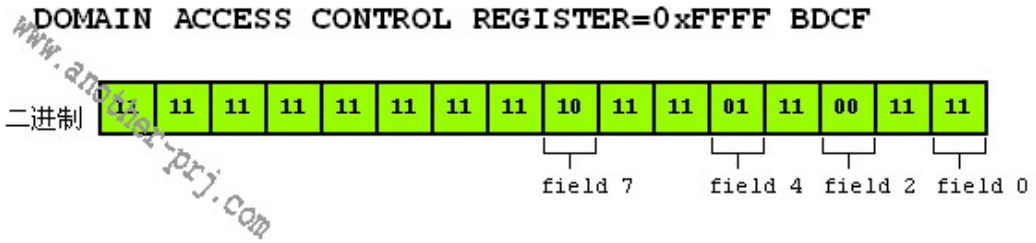
我们再来看看 discriptor 中的 Domain 区域，该区域总共有 4 个 bit,里面的值是对 DOMAIN ACCESS CONTROL REGISTER 中 16 个区域的索引.而 **AP 位配合 S bit 和 A bit** 对当前描述符描述的内存区域被访问权限的说明,他们的配合关系如下图所示:

Table 3-6. Interpreting Access Permission (AP) Bits

AP	S	R	Supervisor Permissions	User Permissions	Notes
00	0	0	No access	No access	Any access generates a permission fault
00	1	0	Read only	No access	Supervisor read only permitted
00	0	1	Read only	Read only	Any write generates a permission fault
00	1	1	Reserved		
01	x	x	Read/write	No access	Access allowed only in supervisor mode
10	x	x	Read/write	Read only	Writes in user mode cause permission fault
11	x	x	Read/write	Read/write	All access types permitted in both modes.
xx	1	1	Reserved		

AP 位也是有四个值,我结合实例对其进行说明.

在下面的例子中，我们的 DOMAIN ACCESS CONTROL REGISTER 都被初始化成 0xFFFF BDCF,如下图所示：



例 1：

Discriptor 中的 domain=4,AP=10(这种情况下 S bit ,A bit 被忽略)

假设现在我要对该描述符描述的内存区域进行访问：

由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。

假设当前 CPU 处于 Supervisor 模式下，则程序可以对该描述符描述的内存区域进行读写操作。

假设当前 CPU 处于 User 模式下，则程序可以对该描述符描述的内存进行读访问，若对其进行写操作则引起一个 permission fault.

例 2：

Discriptor 中的 domain=0,AP=10(这种情况下 S bit ,A bit 被忽略)

domain=0,而 DOMAIN ACCESS CONTROL REGISTER 中 field 0 的值是 11，系统对任何内存区域的访问都不进行访问权限的检查。
由于统对任何内存区域的访问都不进行访问权限的检查，所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都可以顺利地进行读写操作

例 3：Discriptor 中的 domain=4,AP=11(这种情况下 S bit ,A bit 被忽略)
由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。
由于 AP=11，所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都可以顺利地进行读写操作

例 4：
Discriptor 中的 domain=4,AP=00, S bit=0,A bit=0
由于 domain=4,而 DOMAIN ACCESS CONTROL REGISTER 中 field 4 的值是 01，系统会对该访问进行访问权限的检查。
由于 AP=00，S bit=0,A bit=0,所以无论 CPU 处于合种模式下（Supervisor 模式或是 User 模式），程序对该描述符描述的内存都只能进行读操作，否则引起 permission fault.
通过以上 4 个例子我们得出两个结论：
1．对某个内存区域的访问是否需要进行权限检查是由该内存区域的描述符中的 Domain 域决定的。
2．某个内存区域的访问权限是由该内存区域的描述符中的 AP 位和协处理器 CP15 中 Control Register1(控制寄存器 1)中的 S bit 和 R bit 所决定的。
关于访问控制机制我们就讲到这里.

注 1:对于 s3c2410 来说,MMU 是以 Modify Visual Address(MVA) 进行寻址的,这个地址是 Virtual Address 的一个变换,我将在以后谈论到进程切换的时候中向大家介绍 MVA

下一篇文档我将介绍 s3c2410 的 Caches,Write Buffer 并给出开启 MMU 的源代码,请把本文档和下一篇文档配合阅读。

s3c2410 CACHES, WRITE BUFFER (编辑完毕)

作者：蔡于清
www.another-prj.com

在上一篇文档中我向大家介绍MMU的工作原理和对s3c2410 MMU部分操作进行了讲解。我们知道MMU存在的原因是为了支持虚拟存储技术，但不知道你发现了没有，虚拟存储技术的使用会降低整个系统的效率，因为与传统的存储技术相比，虚拟存储技术对内存的访问操作多了一步，就是对地址进行查表（查找映射关系），必须先从虚拟地址中分解出页号和页内偏移，根据页号对描述符进行索引（这就是一个查表过程）得到物理空间的首地址，这样做的代价是巨大的（其实这也正是时间效率与空间效率之间矛盾的一个体现），对某些嵌入式系统来说这简直就是恶梦。那么在引入了虚拟存储技术之后有没有方法在时间效率与空间效率这个矛盾之间取得一个平衡点呢？答案是有，我们可以通过一种技术从最大限度上降低这两者的矛盾，这种技术是Caches(缓存)。也是我们本文要介绍的。

以下内容转载自中计报

Cache的工作原理

Cache的工作原理是基于程序访问的局部性。

对大量典型程序运行情况的分析结果表明，在一个较短的时间间隔内，由程序产生的地址往往集中在存储器逻辑地址空间的很小范围内。指令地址的分布本来就是连续的，再加上循环程序段和子程序段要重复执行多次。因此，对这些地址的访问就自然地具有时间上集中分布的倾向。

数据分布的这种集中倾向不如指令明显，但对数组的存储和访问以及工作单元的选择都可以使存储器地址相对集中。这种对局部范围的存储器地址频繁访问，而对此范围以外的地址则访问甚少的现象，就称为程序访问的局部性。

根据程序的局部性原理，可以在主存和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冲存储器(Cache)。

系统正是依据此原理，不断地将与当前指令集相关联的一个不太大的后继指令集从内存读到Cache，然后再与CPU高速传送，从而达到速度匹配。

CPU对存储器进行数据请求时，通常先访问Cache。由于局部性原理不能保证所请求的数据百分之百地在Cache中，这里便存在一个命中率。即CPU在任一时刻从Cache中可靠获取数据的几率。

命中率越高，正确获取数据的可靠性就越大。一般来说，Cache的存储容量比主存的容量小得多，但不能太小，太小会使命中率太低；也没有必要过大，过大不仅会增加成本，而且当容量超过一定值后，命中率随容量的增加将不会有明显地增长。

只要Cache的空间与主存空间在一定范围内保持适当比例的映射关系，Cache的命中率还是相当高的。

一般规定Cache与内存的空间比为 4：1000，即 128kB Cache可映射 32MB内存；256kB Cache可映射 64MB内存。在这种情况下，命中率都在 90％以上。至于没有命中的数据，CPU只好直接从内存获取。获取的同时，也把它拷进Cache，以备下次访问。

Cache的基本结构

Cache通常由相联存储器实现。相联存储器的每一个存储块都具有额外的存储信息，称为标签(Tag)。当访问相联存储器时，将地址和每一个标签同时进行比较，从而对标签相同的存储块进行访问。Cache的 3 种基本结构如下：

全相联Cache

在全相联Cache中，存储的块与块之间，以及存储顺序或保存的存储器地址之间没有直接的关系。程序可以访问很多的子程序、堆栈和段，而它们是位于主存储器的不同部位上。

因此，Cache保存着很多互不相关的数据块，Cache必须对每个块和块自身的地址加以存储。当请求数据时，Cache控制器要把请求地址同所有地址加以比较，进行确认。

这种Cache结构的主要优点是，它能够在给定的时间内去存储主存器中的不同的块，命中率高；缺点是每一次请求数据同Cache中的地址进行比较需要相当的时间，速度较慢。

直接映像Cache

直接映像Cache不同于全相联Cache，地址仅需比较一次。

在直接映像Cache中，由于每个主存储器的块在Cache中仅存在一个位置，因而把地址的比较次数减少为一次。其做法是，为Cache中的每个块位置分配一个索引字段，用Tag字段区分存放在Cache位置上的不同的块。

单路直接映像把主存储器分成若干页，主存储器的每一页与Cache存储器的大小相同，匹配的主存储器的偏移量可以直接映像为Cache偏移量。Cache的Tag存储器(偏移量)保存着主存储器的页地址(页号)。

以上可以看出，直接映像Cache优于全相联Cache，能进行快速查找，其缺点是当主存储器的组之间做频繁调用时，Cache控制器必须做多次转换。

组相联Cache

组相联Cache是介于全相联Cache和直接映像Cache之间的一种结构。这种类型的Cache使用了几组直接映像的块，对于某一个给定的索引号，可以允许有几个块位置，因而可以增加命中率和系统效率。

Cache与DRAM存取的一致性

在CPU与主存之间增加了Cache之后，便存在数据在CPU和Cache及主存之间如何存取的问题。读写各有 2 种方式。

贯穿读出式(Look Through)

该方式将Cache隔在CPU与主存之间，CPU对主存的所有数据请求都首先送到Cache，由Cache自行在自身查找。如果命中，则切断CPU对主存的请求，并将数据送出；不命中，则将数据请求传给主存。

该方法的优点是降低了CPU对主存的请求次数，缺点是延迟了CPU对主存的访问时间。

旁路读出式(Look Aside)

在这种方式中，CPU发出数据请求时，并不是单通道地穿过Cache，而是向Cache和主存同时发出请求。由于Cache速度更快，如果命中，则Cache在将数据回送给CPU的同时，还来得及中断CPU对主存的请求；不命中，则Cache不做任何动作，由CPU直接访问主存。

它的优点是没有时间延迟，缺点是每次CPU对主存的访问都存在，这样，就占用了一部分总线时间。

写穿式(Write Through)

任一从CPU发出的写信号送到Cache的同时，也写入主存，以保证主存的数据能同步地更新。

它的优点是操作简单，但由于主存的慢速，降低了系统的写速度并占用了总线的时间。

回写式(Copy Back)

为了克服贯穿式中每次数据写入时都要访问主存，从而导致系统写速度降低并占用总线时间的弊病，尽量减少对主存的访问次数，又有了回写式。

它是这样工作的：数据一般只写到Cache，这样有可能出现Cache中的数据得到更新而主存中的数据不变(数据陈旧)的情况。但此时可在Cache 中设一标志地址及数据陈旧的信息，只有当Cache中的数据被再次更改时，才将原更新的数据写入主存相应的单元中，然后再接受再次更新的数据。这样保证了Cache和主存中的数据不致产生冲突。

...

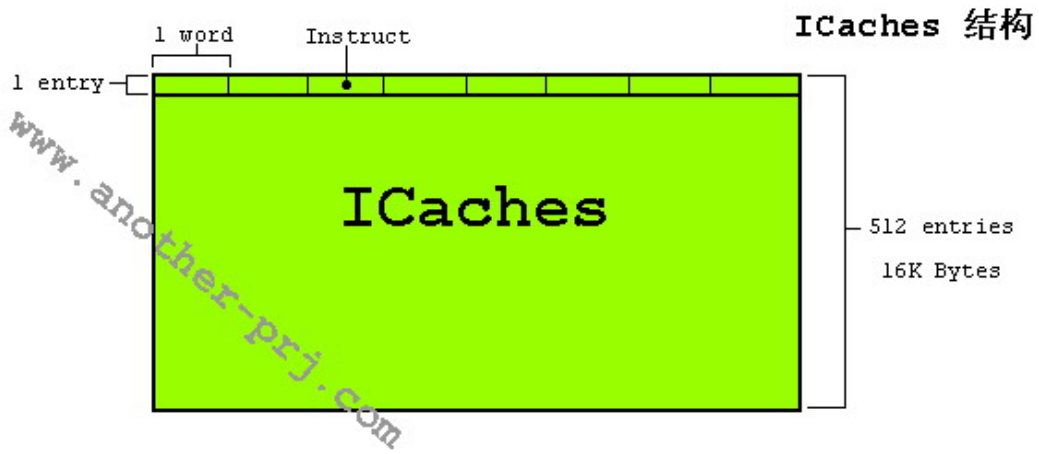
.....

你可以通过http://www.chinaunix.net/jh/45/180390.html阅读完全文

s3c2410 内置了**指令缓存（ICaches）,数据缓存（DCaches）,写缓存（write buffer），物理地址标志读写区（Physical Address TAG RAM),**CPU将通过它们来提高内存访问效率。

我们先讨论**指令缓存（ICaches）。**

ICaches使用的是虚拟地址，它的大小是 16KB,它被分成 512 行(entry)，每行 8 个字（8 words,32Bits）。



当系统上电或重起（Reset）的时候，ICaches 功能是被关闭的，我们必须往 Icr bit 置 1 去开启它，Icr bit 在 CP15 协处理器中控制寄存器 1 的第 12 位（关闭 ICaches 功能则是往该位置 0）。ICaches 功能一般是在 MMU 开启之后被使用的（为了降低 MMU 查表带来的开销）,但有一点需要注意，并不是说 MMU 被开启了 ICaches 才会被开启，正如本段刚开始讲的，ICaches 的开启与关闭是由 Icr bit 所决定的，无论 MMU 是否被开启，只要 Icr bit 被置 1 了，ICaches 就会发挥它的作用。

大家是否还记得 discriptor（描述符）中有一个 C bit 我们称之为 **Ctt**,它是指明该描述符描述的内存区域内的内容（可以是指令也可以是数据）是否可以被 Cache，若 Ctt=1,则允许 Cache,否则不允许被 Cache。于是 CPU 读取指令出现了下面这些情况：

1. 如果 CPU 从 Caches 中读取到所要的一条指令（**cache hit**）且这条指令所在的内存区域是 Cacheble 的（该区域

所属描述符中 Ctt=1）,则 CPU 执行这条指令并从 Caches 中返回（不需要从内存中读取）。

2. 若 CPU 从 Caches 中读取不到所要的指令（**cache miss**）而这条指令所在的内存区域是 Cacheble 的（同第 1 点），则 CPU 将从内存中

读取这条指令，同时，一个称为**“8-word linefill”的动作将发生**，这个动作是把该指令所处区域的 8 个 word 写进

ICaches 的某个 entry 中，这个 entry 必须是没有被锁定的（对锁定这个操作感兴趣的朋友可以找相关的资料进行了解）

3. 若 CPU 从 Caches 中读取不到所要的指令（cache miss）而这条指令所在的内存区域是 UnCacheble 的（该区域所属描

述符中 Ctt=0），则 CPU 将从内存读取这条指令并执行后返回（不发生 linefill）

通过以上的说明，我们可以了解到 CPU 是怎么通过 ICaches 执行指令的。你可能会有这个疑问，ICaches 总共只有 512 个条目（entry），当 512 个条目都被填充完之后，CPU 要把新读取近来的指令放到哪个条目

上呢？答案是 CPU 会把新读取近来的 8 个 word 从 512 个条目中选择一个对其进行写入，那 CPU 是怎么选出一个条目来的呢？这就关系到 ICaches 的**替换法则（replacemnet algorithm）**了。ICaches 的 replacemnet algorithm 有两种，一种是 **Random 模式**另一种 **Round-Robin 模式**，我们可以通过 CP15 协处理器中寄存器 1 的 RR bit 对其进行指定（0 = Random replacement 1 = Round robin replacement），如果有需要你还可以进行指令锁定（INSTRUCTION CACHE LOCKDOWN）。
关于替换法则和指令锁定我就不做详细的讲解，感兴趣的朋友可以找相关的资料进行了解。

接下来我们谈**数据缓存（DCaches）**和 **写缓存（write buffer）**
DCaches 使用的是虚拟地址，它的大小是 16KB,它被分成 512 行（entry），每行 8 个字（8 words,32Bits）。每行有两个修改标志位（dirty bits），第一个标志位标识前 4 个字，第二个标志位标识后 4 个字，同时每行中还有一个 TAG 地址（标签地址）和一个 valid bit。
与 ICaches 一样，系统上电或重起（Reset）的时候，DCaches 功能是被关闭的，我们必须往 Ccr bit 置 1 去开启它，Ccr bit 在 CP15 协处理器中控制寄存器 1 的第 2 位（关闭 DCaches 功能则是往该位置 0）。与 ICaches 不同，DCaches 功能是必须在 MMU 开启之后才能被使用的。
我们现在讨论的都是 **DCaches**,你可能会问那 **Write Buffer** 呢？他和 **DCaches** 区别是什么呢？ 其实 **DCaches** 和 **Write Buffer** 两者间的操作有着非常紧密的联系，很抱歉，到目前为止我无法说出他们之间有什么根本上的区别（-_-!!!），但我能告诉你什么时候使用的是 DCaches,什么时候使用的是 Write Buffer.
系统可以通过 Ccr bit 对 Dcaches 的功能进行开启与关闭的设定，但是在 s3c2410 中却没有确定的某个 bit 可以来开启或关闭 Write Buffer...你可能有点懵...我们还是来看一张表吧，这张表说明了 DCaches,Write Buffer 和 CCr,Ctt (descriptor 中的 C bit),Btt(descriptor 中的 B bit)之间的关系，其中“Ctt and Ccr”一项里面的值是 Ctt 与 Ccr 进行逻辑与之后的值（Ctt&&Ccr）.

Table 4-1. Data Cache and Write Buffer Configuration

Ctt and Ccr	Btt	Data cache, write buffer and memory access behavior
0 (1)	0	Non-cached, non-buffered (NCNB) Reads and writes are not cached and always perform accesses on the ASB and may be externally aborted. Writes are not buffered. The CPU halts until the write is completed on the ASB. Cache hits should never occur. (2)
0	1	Non-cached buffered (NCB) Reads and writes are not cached, and always perform accesses on the ASB. Cache hits should never occur. Writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Reads may be externally aborted. Writes can not be externally aborted.
1	0	Cached, write-through mode (WT) Reads which hit in the cache will read the data from the cache and do not perform an access on the ASB. Reads which miss in the cache cause a linefill. All writes are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache. Writes cannot be externally aborted.
1	1	Cached, write-back mode (WB) Reads which hit in the cache will read the data from the cache and do not perform an ASB access. Reads which miss in the cache cause a linefill. Writes which miss in the cache are placed in the write buffer and will appear on the ASB. The CPU continues execution as soon as the write is placed in the write buffer. Writes which hit in the cache update the cache and mark the appropriate half of the cache line as dirty, and do not cause an ASB access. Cache write-backs are buffered. Writes (Cache write-misses and cache write-backs) cannot be externally aborted.

从上面的表格中我们可以清楚的知道系统什么时候使用的是 DCaches,什么时候使用的是 Write Buffer，我们也可以看到 DCaches 的写回方式是怎么决定的（write-back or write-througth）。

在这里我要对 Ctt and Ccr=0 进行说明，能够使 Ctt and Ccr=0 的共有三种情况，分别是

Ctt =0， Ccr=0

Ctt =1， Ccr=0

Ctt =0， Ccr=1

我们分别对其进行说明。

情况 1（Ctt =0， Ccr=0）：这种情况下 CPU 的 DCaches 功能是关闭的（Ccr=0），所以 CPU 存取数据的时候不会从 DCaches 里进行数据地查询，CPU 直接去内存存取数据。

情况 2（Ctt =1， Ccr=0）：与情况 1 相同。

情况 3（Ctt =0， Ccr=1）：这种情况下 DCaches 功能是开启的，CPU 读取数据的时候会先从 DCaches 里进行数据地查询，若 DCaches 中没有合适的数据，则 CPU 会去内存进行读取，但此时由于 Ctt =0（Ctt 是 descriptor 中的 C bit,该 bit 决定该 descriptor 所描述的内存区域是否可以被 Cache）,所以 CPU 不会把读取到的数据 Cache 到 DCaches(不发生 linefill).

到此为止我们用两句话总结一下 DCaches 与 Write Buffer 的开启和使用：

1. DCaches 与 Write Buffer 的开启由 Ccr 决定。
2. DCaches 与 Write Buffer 的使用规则由 Ctt 和 Btt 决定。

DCaches 与 **ICaches** 有一个最大的不同，ICaches 存放的是指令，DCaches 存放的是数据。程序在运行期间指令的内容是不会改变的，所以 ICaches 中指令所对应的内存空间中的内容不需要更新。但是数据是随着程序的运行而改变的，所以 DCaches 中数据必须被及时的更新到内存（这也是为什么 ICaches 没有写回操作而 DCaches 提供了写回操作的原因）。提到写回操作，就不得不提到 **PA TAG 地址（物理标签地址）**这个固件，它也是整个 Caches 模块的重要组成部分。
简单说 PA TAG 地址（物理标签地址）的功能是指明了写回操作必须把 DCaches 中待写回内容写到物理内存的哪个位置。不知道你还记不记得，DCaches 中每个 entry 中都有一个 PA TAG 地址（物理标签地址），当一个 linefill 发生时，被 Cache 的内容被写进了 DCaches,同时被 Cache 的物理地址则被写入了 PA TAG 地址（物理标签地址）。除了 TAG 地址（标签地址），还有两个称为 **dirty bit（修改标志位）**的位出现在 DCaches 的每一个 entry 中，它们指明了当前 entry 中的数据是否已经发生了改变（发生改变简称为变“脏”，所以叫 dirty bit,老外取名称可真有意思 -_-!!!）。如果某个 entry 中的 dirty bit 置位了，说明该 entry 已经变脏，于是一个写回操作将被执行，写回操作的目的地址则是由 PA TAG 地址（物理标签地址）索引到的物理地址。

关于 Caches，Write Buffer 更详细的内容请大家阅读 s3c2410 的操作手册：]

s3c2410 Timer

作者：蔡于清
www.another-prj.com

s3c2410提供了5个**16位**的Timer（Timer0~Timer4），其中Timer0~Timer3支持Pulse Width Modulation——PWM（脉宽调制 ）。Timer4 是一个内部定时器（internal timer），他没有输出引脚（output pins）。
下面是Timer的工作原理图。

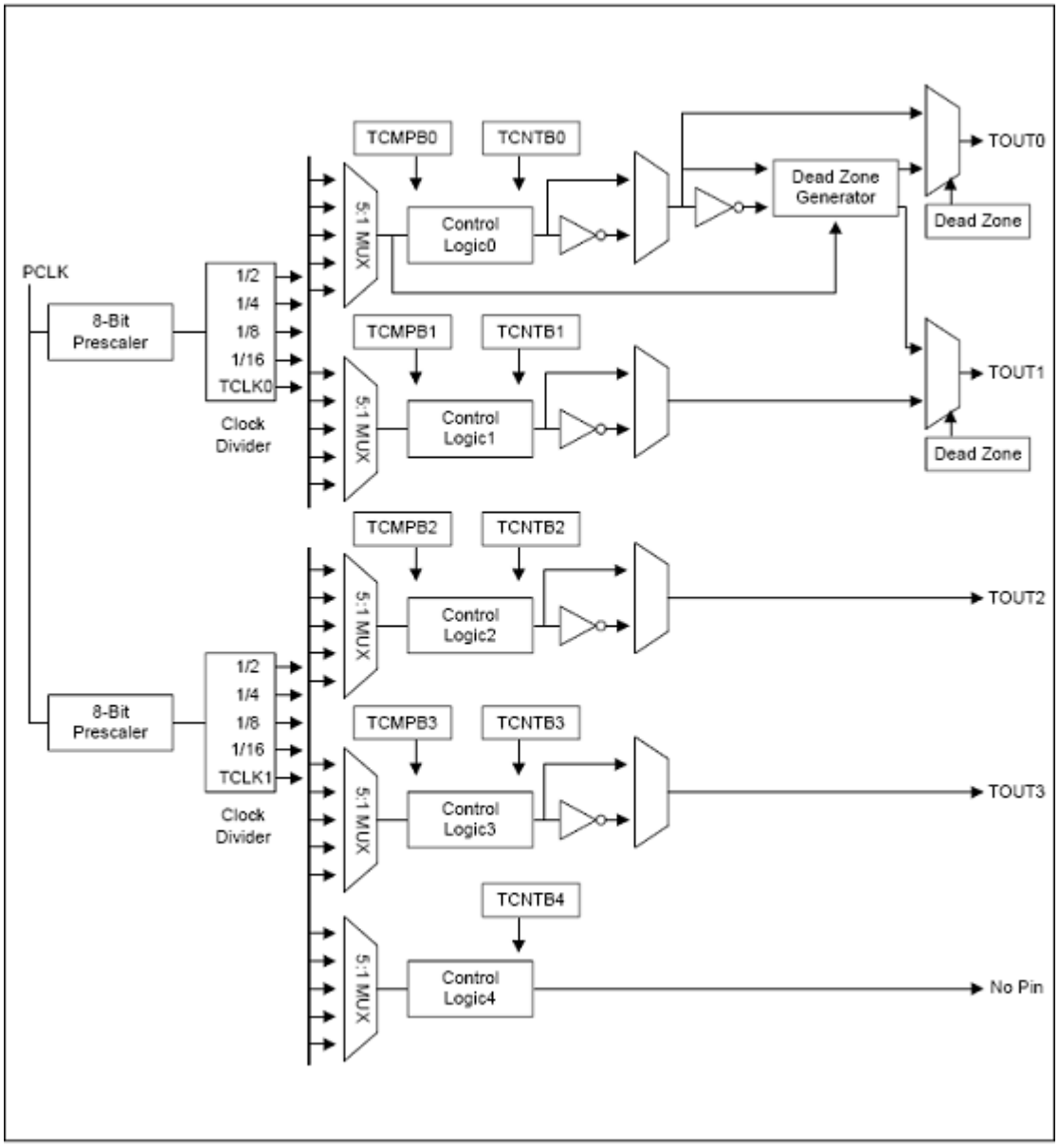


Figure 10-1. 16-bit PWM Timer Block Diagram

[attach]60[/attach]

如上图所示，**PCLK是Timer的信号源**，我们通过设置每个Timer相应的**Prescaler**和**Clock Divider**把PCLK转换成输入时钟信号传送给各个Timer的逻辑控制单元(Control Logic),事实上每个Timer都有一个称为**输入时钟频率（Timer input clock Frequency）**的参数，这个频率就是通过PCLK，Prescaler和Clock Divider确定下来的，每个Timer 的逻辑控制单元就是以这个频率在工作。下面给出输入时钟频率的公式：

Timer input clock Frequency = PCLK / {prescaler value+1} / {clock divider }
{prescaler value} = 0~255
{ clock divider } = 2, 4, 8, 16

然而并不是每一个Timer都有对应的Prescaler和Clock Divider，从上面的原理图我们可以看到Timer0,Timer1 共用一对Prescaler和Clock Divider,Timer2,Timer3,Timer4 共用另一对Prescaler和Clock Divider，s3c2410 的整个时钟系统模块只存在两对Prescaler和Clock Divider。

我曾经在讨论watchdog的文章中提到，watchdog也是一种定时器，他的工作就是在一个单位时间内对一个给定的数值进行递减和比较的操作，而我们这篇文章讨论的定时器他的工作内容和watchdog在本质上是一样的。定时器在一个**工作周期（Timer input clock cycle）**内的具体工作内容主要有 3 个。分别是：

1. **对一个数值进行递减操作**
2. **把递减后的数值和另一个数值进行比较操作**
3. **产生中断或执行DMA操作**

在启用Timer之前我们会对Timer进行一系列初始化操作，这些操作包括上面提到的设置Prescaler和Clock Divider，其中还有一个非常重要的就是要给Timer两个数值，我们分别称之为**Counter**(变量,用于递减)和**Comparer**（定值,用于比较）,Counter会被Timer 加载到**COUNT BUFFER REGISTER（TCNTB）,**而Comparer会被Timer 加载到和**COMPARE BUFFER REGISTER（TCMPB）**，每个Timer都有这样两个寄存器。当我们设置完毕启动Timer之后，Timer在一个工作周期内所做的就是先把**TCNTB**中的数值（Counter）减 1，之后把**TCNTB**中的数值和**TCMPB**中的数值(Comparer)进行对比，若Counter已经被递减到**等于**Comparer,**发生计数超出**，则Timer产生中断信号（或是执行DMA操作）并自动把Counter重新装入TCNTB（刷新TCNTB以重新进行递减）。以上就是Timer的工作原理。

下面我们结合代码具体说明如何对Timer0 进行初始化并开启它。
首先我假设我的PCLK是 50700000Hz

```
// define Timer register
#define rTCFG0 (*(volatile unsigned int *)0x51000000)
#define rTCFG1 (*(volatile unsigned int *)0x51000004)
#define rTCON (*(volatile unsigned int *)0x51000008)
#define rTCNTB0 (*(volatile unsigned int *)0x5100000C)
#define rTCMPB0 (*(volatile unsigned int *)0x51000010)
#define rTCNTO0 (*(volatile unsigned int *)0x51000014)
#define rTCNTB1 (*(volatile unsigned int *)0x51000018)
#define rTCMPB1 (*(volatile unsigned int *)0x5100001C)
#define rTCNTO1 (*(volatile unsigned int *)0x51000020)
#define rTCNTB2 (*(volatile unsigned int *)0x51000024)
#define rTCMPB2 (*(volatile unsigned int *)0x51000028)
#define rTCNTO2 (*(volatile unsigned int *)0x5100002C)
#define rTCNTB3 (*(volatile unsigned int *)0x51000030)
#define rTCMPB3 (*(volatile unsigned int *)0x51000034)
#define rTCNTO3 (*(volatile unsigned int *)0x51000038)
#define rTCNTB4 (*(volatile unsigned int *)0x5100003C)
#define rTCNTO4 (*(volatile unsigned int *)0x51000040)

void timer0_config()
{
/*
        Timer0 的prescaler由rTCFG0 的 0~7 bit决定
```

```
                Prescaler=119
*/
                rTCFG0=119
/*
                Timer0 的divider value由TCFG1 的  0~3 bit决定,设置为 3 表示divider value = 1/16
                rTCFG1 的第 20~23bit用于决定Timer计数超出后所采取的响应，我们使用了中断模式
（20~23bit全部为 0），
                即计数超出后产生中断
*/
                rTCFG1=3;

                rTCNTB0=26406;
                rTCMPB0=0;
}
```

由于我们的PCLK是50700000Hz，根据Timer input clock Frequency的计算公式我们如下计算Timer0的时钟输入频率：

prescaler value = 119
divider value = 1/16
PCLK= 50700000
Timer input clock Frequency =50700000/ (119+1)/(1/16)=26406

也就是说通过设置prescaler和divider value之后，**Timer0 的工作频率为 26406**，也就是说一秒内Timer0会进行 26406 次递减和比较操作，假设我们现在是要让Timer0 每 1 秒产生一次中断的话，我们应该设置Counter=26406 和Camparer=0,既：
rTCNTB0=26406;
rTCMPB0=0;

如果我们要让Timer0 每 0.5 秒产生一次中断，则我们应该设置Counter=26406/2 和Camparer=0,既：
rTCNTB0=13203;
rTCMPB0=0;

如果我们要让Timer0 每 0.25 秒产生一次中断，则我们应该设置Counter=26406/4 和Camparer=0,既：
rTCNTB0=6601;
rTCMPB0=0;

初始化完Timer后我们要开启它。

```
void timer0_start()
{
/*
                Update TCNTB0 & TCMPB0
                rTCON寄存器的第 1 位是刷新Timer0 的COUNT BUFFER REGISTER（TCNTB）和
                 COMPARE BUFFER REGISTER（TCMPB），由于是第一次加载Counter和Comparer，
```

```
                所以我们需要手动刷新它们
*/
        rTCON|=1<<1;
/*
        置rTCON第 0 位为 1，开启Timer0
        把rTCON第 1 位置为 0，停止刷新TCNTB0 和 TCMPB0
        置rTCON第 3 位为 1，设置Counter的加载模式为自动加载（auto reload），这样每当
        Timer计数超出之后（此时TCNTB的值等于TCMPB的值），Timer会自动把原来我们给
        定的Counter重新加载到TCNTB中
*/
    rTCON=0x09;
}
```

要使你的Timer能够正常的工作,除了调用timer0_config()和timer0_start()之外,我们还必须设置Timer的中断服务例程并取消对Timer的中断的屏蔽.这些操作可以参考<<s3c2410 中断异常处理>>一文.
另外OS实验区提供的源代码中有更为详细的代码,需要的朋友可以参考,关于Timer的代码存在于timer.c timer.h文件中