S3C2440完全开发流程

作者：dreamer2006@163.com
2009-11-3

一．简介

本书面向由传统51单片机转向ARM嵌入式开发的硬件工程师、由硬件转嵌入式软件开发的工程师、没有嵌入式开发经验的软件工程师，本书开发板基于天嵌科技的TQ S3C2440开发板，其官方网站为：http://www.embedsky.net/，共分9个部分：

1、开发环境建立
2、S3C2440功能部件介绍与实验(含实验代码)
3、Bootloader vivi详细注释
4、Linux移植
5、Linux驱动
6、Yaffs文件系统详解
7、调试工具
8、GUI开发简介
9、UC/OS移植

通过学习第二部分，即可了解基于ARM CPU的嵌入式开发所需要的外围器件及其接口。对应的实验代码实现了对这些接口的操作，这可以让硬件工程师形成一个嵌入式硬件开发的概念。这部分也可以当作S3C2440的数据手册来使用。

一个完整的嵌入式linux系统包含4部分内容：Bootloader、Parameters、Kernel、Root File System。3、4、5、6部分详细介绍了这4部分的内容，这是Linux底层软件开发人员应该掌握的。通过学习这些章节，您可以详细了解到如何在一个裸板上裁减、移植Linux，如何构造自己的根文件系统，如何编写适合客户需求的驱动程序——驱动程序这章将结合几个经典的驱动程序进行讲解。您还可以了解到在用在nand flash上的非常流行的yaffs文件系统是如何工作的，本书将结合yaffs代码详细介绍yaffs文件系统。

第7部分介绍了嵌入式Linux开发中使用gdb进行调试的详细过程。

此文档目前完成了1、2、3部分，后面部分将陆续完成。希望能对各位在嵌入式开发方面献上棉力。

欢迎来信指出文中的不足与错误，欢迎来信探讨技术问题。

Email： dreamer2006@163.com

QQ ： 389658188

二．建立开发环境

1、 编译器arm-linux-gcc-3.4.1

下载地址：

ftp://ftp.handhelds.org/projects/toolchain/arm-linux-gcc-3.4.1.tar.bz2

执行如下命令安装：

```
bunzip2 arm-linux-gcc-3.4.1.tar.bz2
tar xvf arm-linux-gcc-3.4.1.tar -C /
```

生成的编译工具在目录/usr/local/arm/3.4.1/bin下，修改/etc/profile，增加如下一行。这可以让我们直接运行arm-linux-gcc，而不必将其绝对路径都写出来，不过这得重新启动后才生效：

```
pathmunge /usr/local/arm/3.4.1/bin
```

这个交叉编译器已经比较老旧，对于新版本的编译器我们有很多选择，如下表：

表1.常用交叉编译器

编译器	地址	备注
Crosstool-chain-0.43	http://kegel.com/crosstool	ARM开源项目，自己编译
天嵌编译器(推荐)	http://www.embedsky.net/	EABI_4.3.3_EmbedSky_20090812
CodeSourcery	http://www.codesourcery.com/	Sourcery arm-none-linux-gnueabi
Ronetix-arm-linux	http://www.ronetix.at/software.html	主要针对AT91芯片，可参考
友善编译器(推荐)	http://www.arm9.net/download.asp	内容较全，推荐！！

※注：ronetix-arm-linux-4.3.3.tar.bz2---没有太多研究

CodeSourcery的c编译器，这是ARM官方指定的编译器，可以下载免费版本CodeSourcery的编译器对于新手来说可能不知如何配置，那么没有关系，可以到下载友善已经制作好的基于CodeSourcery的编译器。

2、 Jflash-S3C2440：S3C2440芯片的JTAG工具

我们的第一个程序就是通过它下载到开发板上的nor flash或者nand flash上去的。把它放到/usr/local/bin目录下。

下载地址：

ftp://ftp.mizi.com/pub/linuette/SDK/1.5/target/box/Jflash/Jflash-S3C2440

注意：步骤3您现在不必理会，可以等进行到“调试”部分时再回过头来看。

Jlink-v7

如果手头有这个调试工具，那么使用起来会非常方便，而且支持ADS,MDK,IAR，下载速度远远高于并口JTAG小板，由于Jlink使用的是USB接口，对于没有并口的本本来说无疑又是一大优点。

3、安装**gdb**调试工具

下载地址：

http://www.gnu.org/software/gdb/download/

http://ftp.gnu.org/gnu/gdb/gdb-6.3.tar.gz

A、执行如下命令安装：

a.安装在主机上运行的arm-linux-gdb工具：

```
tar xvzf gdb-6.3.tar.gz
cd gdb6.3
./configure --target=arm-linux
make
make install
```

此时，在/usr/local/bin中生成arm-linux-gdb等工具

b.继续上面的步骤，安装gdbserver。需要将此工具下载到开发板上运行，这在后面会详细描述：

```
cd gdbserver
export CC=/usr/local/arm/3.4.1/bin/arm-linux-gcc
./configure arm-linux
make
```

此时在当前目录中生成了gdbserver工具，当我们讲到如何调试时，会把这个文件下载到开发板上去。

B、Linux程序的调试也可以使用图形界面，如果用的是ubuntu，那么可以先安装ddd如下命令：

```
sudo apt-get install ddd
```

编译程序时加入-g参数，然后启动ddd 生成目标文件进行图形程序调试，非常方

便。

4、USB下载工具

Win平台：

除了通过串口下载程序外，还可以使用USB来下载，速度比串口要快上百倍。常用的也是三星公司推荐的工具是DNW，目前各大开发板商已经推出自己的升级版本DNW，比如友善推出了增加备份功能的DNW，天嵌推出了汉化版的DNW，用起来会更方便。这里要注意的是USB驱动，如果用三星官方提供的驱动，那么会经常出现死机蓝屏现象，但是友善已经将驱动做了改进，蓝屏现象得到修改。

Linux平台：

对于Linux平台，好多人抱怨无法像在WIN下面那样下载方便。但是，国内的高手们已经将DNW移植到了Linux平台上，常见的有两个版本，一个是字符模式版本，一个是GUI版本（基于QT），后者要自己编译，当然也可以使用编译好的二进制文件，前提是主机安装了QT，并用要在root用户下运行程序。

对于调度工具，Linux下也可以用Jlink，相关驱动可以去Segger官方下载驱动，不过只有字符模式。

5、ubuntu开发环境建立

系统安装：

将分区作如下调整，一个根目录，一个交换分区，一个工作目录

```
/    /work    /swap
5G       10G        1G
```

更改升级源：

```
sudo gedit /etc/apt/source.list
```

一般情况下，国内比较好的源有重庆电子，北京交通，上海交通这些教育网资源，另外台湾那边速度也不错。

安装必要工具：

(1)vsftpd

```
sudo apt-get install vsftpd
```

配置：将write_enable=no改为write_enable=yes

将local_enable=no改为local_enbale=yes
然后将服务配置重新启动一下：

```
sudo /etc/init.d/vsftpd restart
```

(2)ssh

```
sudo apt-get install openssh-server
```

(3)NFS service

```
sudo apt-get install nfs-kernel-server portmap
```

配置：打开/etc/exports，添加如下内容：

```
/work/nfs_root *(rw,sync,no_root_squash)
```

然后将服务重新启动一下：

```
sudo /etc/init.d/nfs-kernel-server restart
```

(4)安装二进制工具包

```
sudo apt-get install build-essential      //语法词法分析器
sudo apt-get install bison flex           //同上
sudo apt-get install manpages-dev         //man手册
```

(5)将下载好的交叉编译器安装到系统中

一般情况下，要将编译器放到/usr/local/arm/4.3.2/目录下，然后在~/.bashrc中将编译器的环境变量加入其中，如果在终端中输入arm-none-linux-gcc –v出现相应的版本信息，那么说明已经安装成功。

(6)安装ncurses

官方下载地址：http://www.gnu.org/software/ncurses/

下载源码，然后手工编译安装：

```
cd /work/tools
tar xzf ncurses.tar.gz
cd ncurses-5.6
./configure –with-shared –prefix=/usr
make
sudo make install
```

三．S3C2440基础实验

本章将逐一介绍S3C2440各功能模块，并结合简单的程序进行上机实验。您不必将本章各节都看完，完全可以看了一、两节，得到一个大概的印象之后，就开始下一章。本章可以当作手册来用。

注意：了解S3C2440各部件最好的参考资料是它的数据手册。本文不打算翻译该手册，在进行必要的讲解后，进行实际实验——这才是本文的重点。

1、实验一：LED_ON

led_on.s只有7条指令，它只是简单地点亮发光二极管LED1。本实验的目的是让您对开发流程有个基本概念。

实验步骤：

a．把PC并口和开发板JTAG接口连起来、确保插上“BOOT SEL”跳线、上电(呵呵，废话，如果以后实验步骤中未特别指出，则本步骤省略)

b．进入LED_ON目录后，执行如下命令生成可执行文件led_on：make

c．执行如下命令将led_on写入nand flash：

i. Jflash-S3C2440 led_on /t=5

ii. 当出现如下提示时，输入0并回车：

```
K9S1208 NAND Flash JTAG Programmer Ver 0.0
 0:K9S1208 Program      1:K9S1208 Pr BlkPage   2: Exit

 Select the function to test :
```

iii. 当出现如下提示时，输入0并回车：

```
Available target block number: 0~4095
Input target block number:
```

iv. 当再次出现与步骤ii相同的提示时，输入2并回车

d．按开发板上reset键后可看见LED1被点亮了

实验步骤总地来说分3类：编写源程序、编译/连接程序、烧写代码。

先看看源程序led_on.s：

```
.text
.global _start
_start:
    LDR R0,=0x56000010      @R0设为GPBCON寄存器。此寄存器
                            @用于选择端口B各引脚的功能：
                            @是输出、是输入、还是其他
    MOV R1,#0x00004000
```

```
    STR R1,[R0]             @设置GPB7为输出口
    LDR R0,=0x56000014      @R0设为GPBDAT寄存器。此寄存器
                            @用于读/写端口B各引脚的数据
    MOV R1,#0x00000000      @此值改为0x00000080,
                            @可让LED1熄灭
    STR R1,[R0]             @GPB7输出0，LED1点亮
MAIN_LOOP:
    B MAIN_LOOP
```

对于程序中用到的寄存器GPBCON、GPBDAT，我稍作描述，具体寄存器的操作可看实验三：I/O PORTS。GPBCON用于选择PORT B的11根引脚的功能：输出、输入还是其他特殊功能。每根引脚用2位来设置：00表示输入、01表示输出、10表示特殊功能、11保留不用。LED1-3的引脚是GPB7-GPB10，使用GPBCON中位[12:13]、[13:14]、[15:16]、[17:18]来进行功能设置。GPBDAT用来读/写引脚：GPB0对应位0、GPB1对应位1，诸如此类。当引脚设为输出时，写入0或1可使相应引脚输出低电平或高电平。

程序很简单，第4、5、6行3条指令用于将LED1对应的引脚设成输出引脚；第7、8、9行3条指令让这条引脚输出0；第11行指令是个死循环。

实验步骤b中，指令“make”的作用就是编译、连接led_on.s源程序。Makefile的内容如下：

```
led_on:led_on.s
    arm-linux-gcc -g -c -o led_on.o led_on.s
    arm-linux-ld -Ttext 0x0000000 -g led_on.o -o led_on_tmp.o
    arm-linux-objcopy -O binary -S led_on_tmp.o led_on
clean:
    rm -f led_on
    rm -f led_on.o
    rm -f led_on_tmp.o
```

make指令比较第1行中文件led_on和文件led_on.s的时间，如果led_on的时间比led_on.s的时间旧(led_on未生成时，此条件默认成立)，则执行第2、3、4行的指令更新led_on。您也可以不用指令make，而直接一条一条地执行2、3、4行的指令——但是这样多累啊。第2行的指令是预编译，第3行是连接，第4行是把ELF格式的可执行文件led_on_tmp.o转换成二进制格式文件led_on。执行“make clean”时强制执行6、7、8行的删除命令。

注意：Makefile文件中相应的命令行前一定有一个制表符(TAB)

汇编语言可读性太差，现在请开始实验二，我用C语言来实现了同样的功能，而以后的实验，我也尽可能用C语言实现。

2、 实验二： LED_ON_C

C语言程序执行的第一条指令，并不在main函数中。当我们生成一个C程序的可执行文件时，编译器总是在我们的代码前加一段固定的代码——crt0.o，它是编译器自带的一个文件。此段代码设置C程序的堆栈等，然后调用main函数。很可惜，在我们的裸板上，这段代码无法执行，所以我们得自己写一个。这段代码很简单，只有3条指令。

crt0.s代码：

```
.text
.global _start
_start:
    ldr sp, =1024*4          @设置堆栈，注意：不能大于4k
                             @nand flash中的代码在复位后会
                             @移到内部ram中，它只有4k
    bl main                  @调用C程序中的main函数
halt_loop:
    b halt_loop
```

现在，我们可以很容易写出控制LED的程序了，led_on_c.c代码如下：

```
#define GPBCON (*(volatile unsigned long *)0x56000010)
#define GPBDAT (*(volatile unsigned long *)0x56000014)
int main()
{
    GPBCON = 0x00004000;     //设置GPB7为输出口
    GPBDAT = 0x00000000;     //令GPB7输出0
    return 0;
}
```

最后，我们来看看Makefile:

```
led_on_c : crt0.s led_on_c.c
    arm-linux-gcc -g -c -o crt0.o crt0.s
    arm-linux-gcc -g -c -o led_on_c.o led_on_c.c
    arm-linux-ld -Ttext 0x0000000 -g crt0.o led_on_c.o -o led_on_c_tmp.o
    arm-linux-objcopy -O binary -S led_on_c_tmp.o led_on_c
clean:
    rm -f led_on_c
    rm -f led_on_c.o
    rm -f led_on_c_tmp.o
    rm -f crt0.o
```

第2、3行分别对源程序crt0.s、led_on_c.c进行预编译，第4行将预编译得到的结果连接起来，第5行把连接得到的ELF格式可执行文件led_on_c_tmp.o转换成二进制格式文

件led_on_c。

好了，可以开始上机实验了：

实验步骤：

a．进入LED_ON_C目录后，执行如下命令生成可执行文件led_on_c：
make
b．执行如下命令将led_on_c写入nand flash：
i. Jflash-S3C2440 led_on_c /t=5
ii.当出现如下提示时，输入0并回车：
K9S1208 NAND Flash JTAG Programmer Ver 0.0
0:K9S1208 Program 1:K9S1208 Pr BlkPage 2: Exit
Select the function to test :
iii.当出现如下提示时，输入0并回车：
Input target block number:
iv.当出现与步骤ii相同的提示时，输入2并回车
c．按开发板上reset键后可看见LED1被点亮了

目录LEDS中的程序是使用4个LED从0到15轮流计数，您可以试试：

a．进入目录后make
b．Jflash-S3C2440 leds /t=5
c．reset运行

另外，如果您有兴趣，可以使用如下命令看看二进制可执行文件的反汇编码：

arm-linux-objdump -D -b binary -m arm xxxxx(二进制可执行文件名)

注意：本文的所有程序均在SOURCE目录中，各程序所在目录均为大写，其可执行文件名为相应目录名的小写，比如LEDS目录下的可执行文件为leds。以后不再赘述如何烧写程序：直接运行Jflash-S3C2440即可看到提示。

3、实验三：I/O PORTS

请打开S3C2440数据手册第9章IO/ PORTS，I/O PORTS含GPA、GPB、……、GPH八个端口。它们的寄存器是相似的：GPxCON用于选择引脚功能，GPxDAT用于读/写引脚数据，GPxUP用于确定是否使用内部上拉电阻(x为A、B、……、H，没有GPAUP寄存器)。

1、PORT A与PORT B-H在功能选择方面有所不同，GPACON中每一位对应一根引脚(共23根引脚)。当某位设为0时，相应引脚为输出引脚，此时我们可以在GPADAT中相应位写入0或1让此引脚输出低电平或高电平；当某位设为1时，相应引脚为地址线或用于地址控制，此时GPADAT无用。一般而言GPACON通常设为全1，以便访问外部存储器件。PORT A我们暂时不必理会。

2、PORT B-H在寄存器操作方面完全相同。GPxCON中每两位控制一根引脚：00表示输入、01表示输出、10表示特殊功能、11保留不用。GPxDAT用于读/写引脚：当引脚设为输入时，读此寄存器可知相应引脚的状态是高是低；当引脚设为输出时写此寄存器相应位可令此引脚输出低电平或高电平。GpxUP：某位为0时，相应引脚无内部上拉；为1时，相应引脚使用内部上拉。

其他寄存器的操作在后续相关章节使用到时再描述；PORT A-H中引脚的特殊功能比如串口引脚、中断引脚等，也在做相关实验时再描述。

目录KEY_LED中的程序功能为：当K1-K4中某个按键按下时，LED1-LED4中相应LED点亮。

key_led.c代码：

```
#define GPBCON (*(volatile unsigned long *)0x56000010)
#define GPBDAT (*(volatile unsigned long *)0x56000014)
#define GPFCON (*(volatile unsigned long *)0x56000050)
#define GPFDAT (*(volatile unsigned long *)0x56000054)

/* LED1-4对应GPB7-10 */
#define GPB7_out (1<<(7*2))
#define GPB8_out (1<<(8*2))
#define GPB9_out (1<<(9*2))
#define GPB10_out (1<<(10*2))

/* K1-K3对应GPF1-3 K4对应GPF7 */
#define GPF1_in ~(3<<(1*2))
#define GPF2_in ~(3<<(2*2))
#define GPF3_in ~(3<<(3*2))
#define GPF7_in ~(3<<(7*2))
int main()
{
    //LED1-LED4对应的4根引脚设为输出
    GPBCON =GPB7_out | GPB8_out | GPB9_out | GPB10_out ;
    //K1-K4对应的4根引脚设为输入
    GPFCON &= GPF1_in & GPF2_in & GPF3_in & GPF7_in ;
    while(1){
    //若Kn为0(表示按下)，则令LEDn为0(表示点亮)
        GPBDAT = ((GPFDAT & 0x0e)<<6) | ((GPFDAT & 0x80)<<3); }
    return 0;
}
```

实验步骤：

a．进入目录KEY_LED，运行make命令生成key_led
b．烧写key_led

4、实验四：arm-linux-ld

在开始后续实验之前，我们得了解一下arm-linux-ld连接命令的使用。在上述实验中，我们一直使用类似如下的命令进行连接：

```
arm-linux-ld -Ttext 0x00000000 crt0.o led_on_c.o -o led_on_c_tmp.o
```

我们看看它是什么意思：-o选项设置输出文件的名字为led_on_c_tmp.o；“--Ttext 0x00000000”设置代码段的起始地址为0x00000000；这条指令的作用就是将crt0.o和led_on_c.o连接成led_on_c_mp.o可执行文件，此可执行文件的代码段起始地址为0x00000000。

我们感兴趣的就是“—Ttext”选项！进入LINK目录，link.s代码如下：

```
.text
.global _start
_start:
    b step1
step1:
    ldr pc, =step2
step2:
  b  step2
```

Makefile如下：

```
link:link.s
    arm-linux-gcc -c -o link.o link.s
    arm-linux-ld -Ttext 0x00000000 link.o -o link_tmp.o
    # arm-linux-ld -Ttext 0x30000000 link.o -o link_tmp.o
    arm-linux-objcopy -O binary -S link_tmp.o link
    arm-linux-objdump -D -b binary -m arm link >ttt.s
    # arm-linux-objdump -D -b binary -m arm link >ttt2.s
clean:
    rm -f link
    rm -f link.o
    rm -f link_tmp.o
```

实验步骤：

1．进入目录LINK，运行make生成arm-linux-ld选项为“-Ttext 0x00000000”的反汇编

码ttt.s
2．make clean
3．修改Makefile：将第4、7行的“#”去掉，在第3、6行前加上“#”
4.运行make生成arm-linux-ld选项为“-Ttext 0x30000000”的反汇编码ttt2.s

link.s程序中用到两种跳转方法：b跳转指令、直接向pc寄存器赋值。我们先把在不同“—Ttext”选项下，生成的可执行文件的反汇编码列出来，再详细分析这两种不同指令带来的差异。

ttt.s：	ttt2.s
0: eafffffff b 0x4	0: eafffffff b 0x4
4: e59ff000 ldr pc, [pc, #0] ; 0xc	4: e59ff000 ldr pc, [pc, #0] ; 0xc
8: eafffffe b 0x8	8: eafffffe b 0x8
c: 00000008 andeq r0, r0, r8	c: 30000008 tsteq r0, #8 ; 0x8

先看看b跳转指令：它是个相对跳转指令，其机器码格式如下：

[31:28]位是条件码；[27:24]位为“1010”时，表示B跳转指令，为“1011”时，表示BL跳转指令；[23:0]表示偏移地址。使用B或BL跳转时，下一条指令的地址是这样计算的：将指令中24位带符号的补码立即数扩展为32(扩展其符号位)；将此32位数左移两位；将得到的值加到pc寄存器中，即得到跳转的目标地址。我们看看第一条指令“b step1”的机器码eafffffff：

1．24位带符号的补码为0xffffff，将它扩展为32得到：0xffffffff
2．将此32位数左移两位得到：0xfffffffc，其值就是-4
3．pc的值是当前指令的下两条指令的地址，加上步骤2得到的-4，这恰好是第二条指令step1的地址

各位不要被被反汇编代码中的“b 0x4”给迷惑了，它可不是说跳到绝对地址0x4处执行，绝对地址得像上述3个步骤那样计算。您可以看到b跳转指令是依赖于当前pc寄存器的值的，这个特性使得使用b指令的程序不依赖于代码存储的位置——即不管我们连接命令中“--Ttext”为何，都可正确运行。

再看看第二条指令ldr pc, =step2：从反汇编码“ldr pc, [pc, #0]”可以看出，这条指令从内存中某个位置读出数据，并赋给pc寄存器。这个位置的地址是当前pc寄存器的值加上偏移值0，其中存放的值依赖于连接命令中的“--Ttext”选项。执行这条指令后，对于ttt.s，pc=0x00000008；对于ttt2.s， pc=0x30000008。于是执行第三条指令“b step2”时，它的绝对地址就不同了：对于ttt.s，绝对地址为0x00000008；对于ttt.s，绝对地址为0x30000008。

ttt2.s上电后存放的位置也是0，但是它连接的地址是0x30000000。我们以后会经常用到“存储地址和连接地址不同”(术语上称为加载时域和运行时域)的特性：大多

机器上电时是从地址0开始运行的，但是从地址0运行程序在性能方面总有很多限制，所以一般在开始的时候，使用与位置无关的指令将程序本身复制到它的连接地址处，然后使用向pc寄存器赋值的方法跳到连接地址开始的内存上去执行剩下的代码。在实验5、6中，我们将会作进一步介绍。

arm-linux-ld命令中选项“-Ttext”也可以使用选项“-Tfilexxx”来代替，在文件filexxx中，我们可以写出更复杂的参数来使用arm-linux-ld命令——在实验6中，我们就是使用这种方法来指定连接参数的。

5、实验五：MEMORY CONTROLLER

S3C2440提供了外接ROM、SRAM、SDRAM、NOR Flash、NAND Flash的接口 。S3C2440外接存储器的空间被分为8 BANKS，每BANK容量为128M：当访问BANKx(x从0到7)所对应的地址范围(x*128M到(x+1)*128M-1，BANK6、7有稍微差别，请参考下面第5点BANKSIZE寄存器的说明)时，片选信号nGCSx有效。本文所用的开发板，使用了64M的NAND Flash和64M的SDRAM：NAND Flash不对应任何BANK，它是通过几组寄存器来访问的，在上电后，NAND Flash开始的4k数据被自动地复制到芯片内部一个被称为“Steppingstone”的RAM上。Steppingstone被映射为地址0，上面的4k程序完成必要的初始化；SDRAM使用BANK6，它的物理起始地址为6*128M=0x30000000。请您打开S3C2440数据手册，第5章的图“Figure 5-1. S3C2440X Memory Map after Reset”可让您一目了然。

在开始下面内容前，如果您对SDRAM没什么概念，建议先看看这篇文章《高手进阶，终极内存技术指南——完整/进阶版》。当然，不看也没关系，照着做就行了。此文链接地址：

http://bbs.cpcw.com/viewthread.php?tid=196978&fpage=1&highlight=

本实验介绍如何使用SDRAM，这需要设置13个寄存器。呵呵，别担心，这些寄存器很多是类似的，并且由于我们只使用了BANK6，大部分的寄存器我们不必理会：

1．BWSCON：对应BANK0-BANK7，每BANK使用4位。这4位分别表示：
   a．STx：启动/禁止SDRAM的数据掩码引脚，对于SDRAM，此位为0；对于SRAM，此位为1。
   b．WSx：是否使用存储器的WAIT信号，通常设为0
   c．DWx：使用两位来设置存储器的位宽：00-8位，01-16位，10-32位，11-保留。
   d．比较特殊的是BANK0对应的4位，它们由硬件跳线决定，只读。

对于本开发板，使用两片容量为32Mbyte、位宽为16的SDRAM组成容量为64Mbyte、位宽为32的存储器，所以其BWSCON相应位为：0010。对于本开发板，BWSCON可设为0x22111110：其实我们只需要将BANK6对应的4位设为0010即可，其它的是什么值没什么影响，这个值是参考手册上给出的。

2．BANKCON0-BANKCON5：我们没用到，使用默认值0x00000700即可

3．BANKCON6-BANKCON7：设为0x00018005
在8个BANK中，只有BANK6和BANK7可以使用SRAM或SDRAM，所以BANKCON6-7与BANKCON0-5有点不同：

a．MT([16:15])：用于设置本BANK外接的是SRAM还是SDRAM：SRAM-0b00，SDRAM-0b11

b．当MT=0b11时，还需要设置两个参数：
Trcd([3:2])：RAS to CAS delay，设为推荐值0b01
SCAN([1:0])：SDRAM的列地址位数，对于本开发板使用的SDRAM HY57V561620CT-H，列地址位数为9，所以SCAN=0b01。如果使用其他型号的SDRAM，您需要查看它的数据手册来决定SCAN的取值：00-8位，01-9位，10-10位

4．REFRESH(SDRAM refresh control register)：设为0x008e0000+ R_CNT
其中R_CNT用于控制SDRAM的刷新周期，占用REFRESH寄存器的[10:0]位，它的取值可如下计算(SDRAM时钟频率就是HCLK)：

```
R_CNT = 2^11 + 1 – SDRAM时钟频率(MHz) * SDRAM刷新周期(uS)
```

在未使用PLL时，SDRAM时钟频率等于晶振频率12MHz；SDRAM的刷新周期在SDRAM的数据手册上有标明，在本开发板使用的SDRAM HY57V561620CT-H的数据手册上，可看见这么一行“8192 refresh cycles / 64ms”：所以，刷新周期=64ms/8192 = 7.8125 uS。

对于本实验，R_CNT = 2^11 + 1 – 12 * 7.8125 = 1955, REFRESH=0x008e0000 + 1955 = 0x008e07a3

5．BANKSIZE：0x000000b2
位[7]=1：Enable burst operation
位[5]=1：SDRAM power down mode enable
位[4]=1：SCLK is active only during the access (recommended)
位[2:1]=010：BANK6、BANK7对应的地址空间与BANK0-5不同。

BANK0-5的地址空间都是固定的128M，地址范围是(x*128M)到(x+1)*128M-1，x表示0到5。但是BANK7的起始地址是可变的，您可以从S3C2440数据手册第5章“Table 5-1. Bank 6/7 Addresses”中了解到BANK6、7的地址范围与地址空间的关系。本开发板仅使用BANK6的64M空间，我们可以令位[2:1]=010(128M/128M)或001(64M/64M)：这没关系，多出来的空间程序会检测出来，不会发生使用不存在的内存的情况——后面介绍到的bootloader和linux内核都会作内存检测。位[6]、位[3]没有使用

6．MRSRB6、MRSRB7：0x00000030
能让我们修改的只有位[6:4](CL)，SDRAM HY57V561620CT-H不支持CL=1的情况，所以位[6:4]取值为010(CL=2)或011(CL=3)。

只要我们设置好了上述13个寄存器，往后SDRAM的使用就很简单了。本实验先使用汇编语言设置好SDRAM，然后把程序本身从Steppingstone(还记得吗？本节开始的时候提到过，复位之后NAND Flash开头的4k代码会被自动地复制到这里)复制到SDRAM处，然后跳到SDRAM中执行。

本实验源代码在SDRAM目录中， head.s开头的代码如下：

```
bl disable_watch_dog
```

```
    bl memsetup
    bl copy_steppingstone_to_sdram
    ldr pc, =set_sp              @跳到SDRAM中继续执行
set_sp:
    ldr sp, =0x34000000          @设置堆栈
    bl main                      @跳转到C程序main函数
halt_loop:
    b halt_loop
```

为了让程序结构简单一点，我都使用函数调用的方式。第一条指令是禁止WATCH DOG，您如果细心的话，一定会发现程序LEDS运行得有些不正常，那是因为WATCH DOG在不断地重启系统。以前为了程序简单，我没有把这段程序加上去。往WTCON寄存器(地址0x53000000)写入0即可禁止WATCH DOG。第二条指令设置本节开头所描述的13个寄存器，以便使用SDRAM。请您翻看实验四最后一段文字，往下程序做的事情就是：将Steppingstone中的代码复制到SDRAM中(起始地址为0x30000000)，然后向pc寄存器直接赋值跳到SDRAM中执行下一条指令“ldr sp, =0x34000000”。再往后的代码就和实验二、三一样了。

最后我们来看看SDRAM目录下的Makefile：

```
sdram : head.s sdram.c
    arm-linux-gcc -c -o head.o head.s
    arm-linux-gcc -c -o sdram.o sdram.c
    arm-linux-ld -Ttext 0x30000000 head.o sdram.o -o sdram_tmp.o
    arm-linux-objcopy -O binary -S sdram_tmp.o sdram
```

请看第4句，是否和实验四联系起来了呢？忘记的人请回头复习，不再罗嗦。

在目录SDRAM下执行make指令生成可执行文件sdram后，下载到板子上运行，可以发现与LEDS程序相比，LED闪烁得更慢：这就对了，外部SDRAM的性能比起内部SRAM来说性能是差些。

把程序从性能更好的内部SRAM移到外部SDRAM中去，是否多此一举呢？内部SRAM只有4k大小，如果我们的程序大于4k，那么就不能指望利用内部SRAM来运行了。所以得想办法把存储在NAND Flash中的代码，复制到SDRAM中去。对于NAND Flash中的前4k，芯片自动把它复制到内部SRAM中，我们可以很轻松地再把它复制到SDRAM中(实验五中函数copy_steppingstone_to_sdram就做这事)。但是对于4k之后的代码，复制它就不那么轻松了，这正是实验六的内容：

6、实验六：NAND FLASH CONTROLLER

当OM1、OM0都是低电平(请看数据手册198页)——即开发板插上BOOT SEL跳线时，S3C2440从NAND Flash启动：NAND Flash的开始4k代码会被自动地复制到内部SRAM中。我们需要使用这4k代码来把更多的代码从NAND Flash中读到SDRAM中

去。NAND Flash的操作通过NFCONF、NFCMD、NFADDR、NFDATA、NFSTAT和NFECC六个寄存器来完成。在开始下面内容前，请打开S3C2440数据手册和NAND Flash K9F1208U0M的数据手册。

在S3C2440数据手册218页，我们可以看到读写NAND Flash的操作次序：

1). Set NAND flash configuration by NFCONF register.
2). Write NAND flash command onto NFCMD register.
3). Write NAND flash address onto NFADDR register.
4). Read/Write data while checking NAND flash status by NFSTAT register. R/nB signal should be checked before read operation or after program operation.

下面依次介绍：

1)、NFCONF：设为0xf830——使能NAND Flash控制器、初始化ECC、NAND Flash片选信号nFCE=1(inactive，真正使用时再让它等于0)、设置TACLS、TWRPH0、TWRPH1。需要指出的是TACLS、TWRPH0和TWRPH1，请打开S3C2440数据手册218页，可以看到这三个参数控制的是NAND Flash信号线CLE/ALE与写控制信号nWE的时序关系。我们设的值为TACLS=0，TWRPH0=3，TWRPH1=0，其含义为：TACLS=1个HCLK时钟，TWRPH0=4个HCLK时钟，TWRPH1=1个HCLK时钟。请打开K9F1208U0M数据手册第13页，在表“**AC Timing Characteristics for Command / Address / Data Input**”中可以看到：

CLE setup Time = 0 ns，CLE Hold Time = 10 ns，
ALE setup Time = 0 ns，ALE Hold Time = 10 ns，
WE Pulse Width = 25 ns

可以计算，即使在HCLK=100MHz的情况下，TACLS+TWRPH0+TWRPH1=6/100 uS=60 ns，也是可以满足NAND Flash K9F1208U0M的时序要求的。

2)、NFCMD：

对于不同型号的Flash，操作命令一般不一样。对于本板使用的K9F1208U0M，请打开其数据手册第8页“**Table 1. Command Sets**”，上面列得一清二楚。

3)、NFADDR：无话可说

4)、NFDATA：只用到低8位

5)、NFSTAT：只用到位0，0-busy，1-ready

6)、NFECC：待补

现在来看一下如何从NAND Flash中读出数据，请打开K9F1208U0M数据手册第29页“**PAGE READ**”，跟本节的第2段是遥相呼应啊，提炼出来罗列如下(设读地址为addr)：

1)、NFCONF = 0xf830
2)、在第一次操作NAND Flash前，通常复位一下：
NFCONF &= ~0x800 (使能NAND Flash)
NFCMD = 0xff (reset命令)
循环查询NFSTAT位0，直到它等于1
3)、NFCMD = 0 (读命令)
4)、这步得稍微注意一下，请打开K9F1208U0M数据手册第7页，那个表格列出了在地址操作的4个步骤对应的地址线，A8没用到：
NFADDR = addr & 0xff
NFADDR = (addr>>9) & 0xff (注意了，左移9位，不是8位)
NFADDR = (addr>>17) & 0xff (左移17位，不是16位)
NFADDR = (addr>>25) & 0xff (左移25位，不是24位)
5)、循环查询NFSTAT位0，直到它等于1
6)、连续读NFDATA寄存器512次，得到一页数据(512字节)
7)、NFCONF |= 0x800 (禁止NAND Flash)

本实验代码在NAND目录下，源代码为head.s、init.c和main.c。head.s调用init.c中的函数来关WATCH DOG、初始化SDRAM、初始化NAND Flash，然后将main.c中的代码从NAND Flash地址4096开始处复制到SDRAM中，最后，跳到main.c中的main函数继续执行。代码本身没什么难度，与前面程序最大的差别就是“连接脚本”的引入：实验4最后一段提到过在arm-linux-ld命令中，选项“-Ttext”可以使用选项“-Tfilexxx”来代替。在本实验中，使用“-Tnand.lds”。nand.lds内容如下：

```
SECTIONS {
    firtst 0x00000000 : { head.o init.o }
    second 0x30000000 : AT(4096) { main.o }
}
```

完整的连接脚本文件形式如下：

```
SECTIONS {
...
secname start BLOCK(align) (NOLOAD) : AT ( ldadr )
{ contents } >region :phdr =fill
...
}
```

并非每个选项都是必须的，仅介绍nand.lds用到的：

1)、secname：段名，对于nand.lds，段名为first和second

2)、start：本段运行时的地址，如果没有使用AT(xxx)，则本段存储的地址也是start

3)、AT(ldadr)：定义本段存储(加载)的地址

4)、{ contents }：决定哪些内容放在本段，可以是整个目标文件，也可以是目标文件中的某段(代码段、数据段等)

nand.lds的含义是：head.o放在0x00000000地址开始处，init.o放在hean.o后面，它们的运行地址是0x00000000；main.o放在地址4096(0x1000)开始处，但是它的运行地址在0x30000000，在运行前需要从4096处复制到0x30000000处。为了更形象一点，您可以打开反汇编文件ttt.s，现摘取部分内容如下：

```
00000000 <.data>:
1 0: e3a0da01 mov sp, #4096 ; 0x1000
2 4: eb00000b bl 0x38
3 8: eb000011 bl 0x54
4 c: eb000042 bl 0x11c
...
5 1000: e1a0c00d mov ip, sp
6 1004: e92dd800 stmdb sp!, {fp, ip, lr, pc}
7 1008: e24cb004 sub fp, ip, #4 ; 0x4
8 100c: e59f1058 ldr r1, [pc, #88] ; 0x106c
...
```

上面的第1-4行与head.s中的前面4行代码对应，第2-4行调用init.c中的函数disable_watch_dog、memsetup、init_nand；再看看第5行，“1000”的得来正是由于设置了“AT(4096)”，这行开始的是main.c中的第一个函数Rand()。

如果您想进一步了解连接脚本如何编写，请参考《Using ld The GNU linker》(在目录“参考资料”下)。

上面的几个程序都是在摆弄那几个LED，现在来玩点有意思的：

7、实验七：UART

UART的寄存器有11X3个(3个UART)之多，我选最简单的方法来进行本实验，用到的寄存器也有8个。不过初始化就用去了5个寄存器，剩下的3个用于接收、发送数据。如此一来，操作UART倒也不复杂。本板使用UART0：

1、初始化：

a、把使用到的引脚GPH2、GPH3定义为TXD0、RXD0：

GPHCON |= 0xa0

GPHUP |= 0x0c (上拉)

b．ULCON0 (UART channel 0 line control register)：设为0x03

此值含义为：8个数据位，1个停止位，无校验，正常操作模式(与之相对的是Infra-Red Mode，此模式表示0、1的方式比较特殊)。

c．UCON0 (UART channel 0 control register)：设为0x05

除了位[3:0]，其他位都使用默认值。位[3:0]=0b0101表示：发送、接收都使用“中断或查询方式”——本实验使用查询查询方式。

d．UFCON0 (UART channel 0 FIFO control register)：设为0x00

每个UART内部都有一个16字节的发送FIFO和接收FIFO，但是本实验不使用FIFO，设为默认值0

e．UMCON0 (UART channel 0 Modem control register)：设为0x00

本实验不使用流控，设为默认值0

f．UBRDIV0 (R/W Baud rate divisior register 0)：设为12

本实验未使用PLL， PCLK=12MHz，设置波特率为57600，则由公式

UBRDIVn = (int)(PCLK / (bps x 16)) –1

可以计算得UBRDIV0 = 12，请使用S3C2440数据手册第314页的误差公式验算一下此波特率是否在可容忍的误差范围之内，如果不在，则需要更换另一个波特率(本实验使用的57600是符合的)。

2、发送数据：

a．UTRSTAT0 (UART channel 0 Tx/Rx status register)：

位[2]：无数据发送时，自动设为1。当我们要使用串口发送数据时，先读此位以判断是否有数据正在占用发送口。

位[1]：发送FIFO是否为空，本实验未用此位

位[0]：接收缓冲区是否有数据，若有，此位设为1。本实验中，需要不断查询此位一判断是否有数据已经被接收。

b．UTXH0 (UART channel 0 transmit buffer register)： 把要发送的数据写入此寄存器。

3、接收数据：

a．UTRSTAT0：如同上述“2、发送数据”所列，我们用到位[0]

b．URXH0 (UART channel 0 receive buffer register)：

当查询到UTRSTAT0 位[0]=1时，读此寄存器获得串口接收到的数据。

串口代码在UART目录下的serial.c文件中，包含三个函数：init_uart，putc，getc。代码如下：

```
void init_uart( )
{//初始化UART
    GPHCON |= 0xa0;          //GPH2,GPH3 used as TXD0,RXD0
    GPHUP = 0x0c;            //GPH2,GPH3内部上拉
    ULCON0 = 0x03;           //8N1(8个数据位，无校验位，1个停止位)
    UCON0 = 0x05;            //查询方式
    UFCON0 = 0x00;           //不使用FIFO
    UMCON0 = 0x00;           //不使用流控
    UBRDIV0 = 12;            //波特率为57600
}
void putc(unsigned char c)
{                            //不断查询，直到可以发送数据
    while(!(UTRSTAT0 & TXD0READY));
    UTXH0 = c;               //发送数据
}
unsigned char getc()
{                            //不断查询，直到接收到了数据
    while(!(UTRSTAT0 & RXD0READY));
     return URXH0;           //返回接收到的数据
}
```

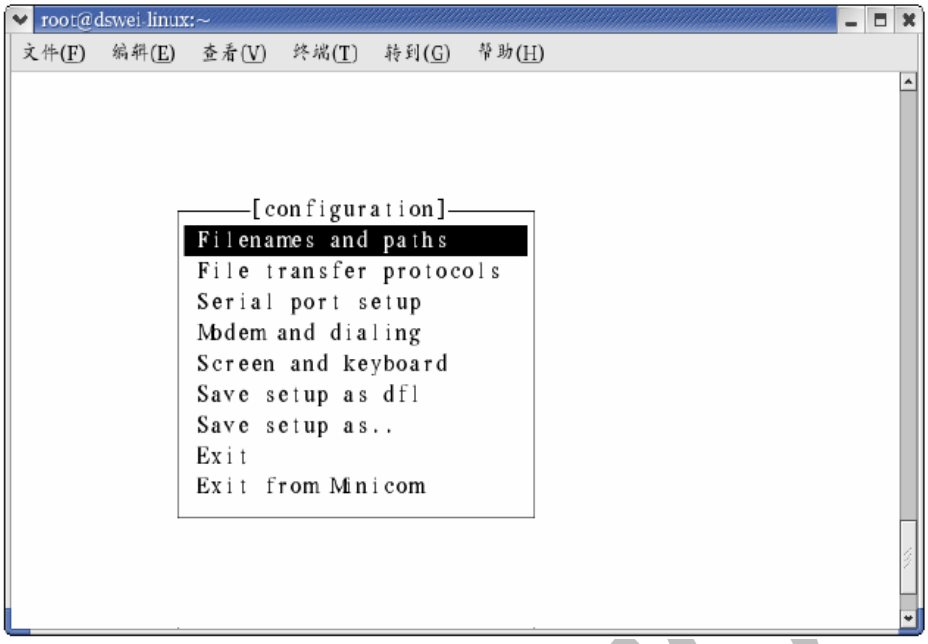
本实验将串口输入的数字、字母加1后再从串口输出，比如输入A就输出B。

实验步骤：

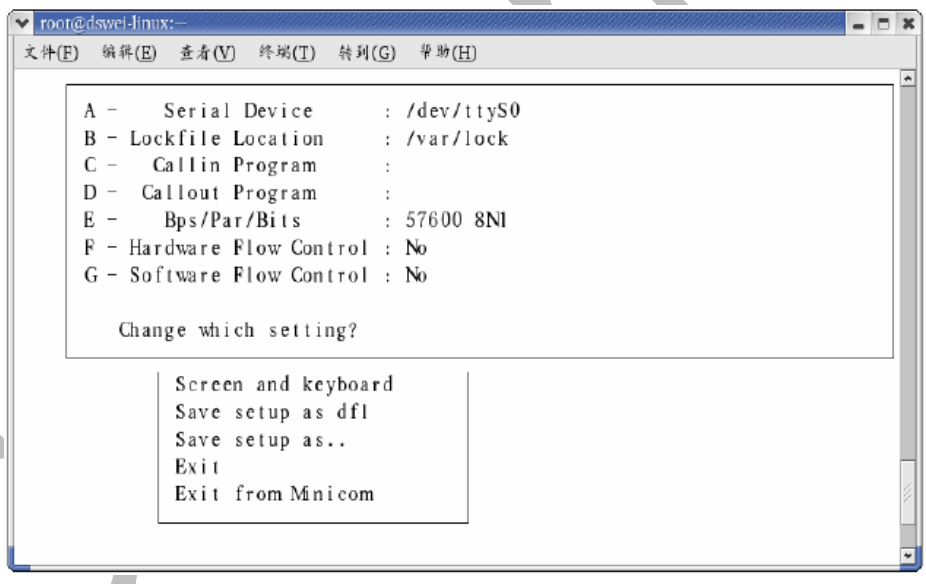
1、进入UART目录运行“make”命令生成可执行文件uart，下载到开发板上

2、在主机上运行串口工具minicom：

a．在终端上运行“minicom –s”启动minicom，出来如下界面：



b．进入“Serial port setup”：



键入相应字母设置各项，比如：我用的是串口1，所以在A项设置为“/dev/ttyS0”；按“E”，设置波特率为57600，8N1；按F、G，设置无流控。最后回车退出，回到步骤a所示的界面。

c．可以选择“Save setup as dfl”，这样下次启动minicom时可以不再进行步骤b的设置。

d．选择“Exit”退出设置界面。

3、复位开发板后，您可以在minicom上体验一下本程序了。

8、实验八：printf、scanf

本实验利用串口实现两个很常用的函数：printf和scanf。

试验代码存放在stdio目录下，其中lib目录中包含了实现printf和scanf函数的主要文件。大部分文件摘自linux2.6内核，本试验主要在vsprintf.c文件的基础上，封装了printf和scanf函数(print.c文件中)。在vsprintf.c文件中，需要用到一些乘法和除法操作，文件lib1funcs.S实现除法、求模操作；div64.h和div64.S实现64位的除法操作；muldi3.c实现乘法操作。另外，stdio目录下的serial.c文件实现了putc和getc函数，这两个函数与具体板子的情况相关，所以我没把它们放入lib目录下。

此试验的Makefile文件将lib目录里的文件生成静态库文件libc.a，现在，你把stdio.h文件包含进你的代码后，就可以非常方便地实现输入、输出了——请参考本试验代码。

实验：与试验7类似，以波特率57600 8N1打开串口，将make后生成的可执行文件exe烧入开发板，可在minicom上观察到结果：程序从minicom中接收字串，从这些字串中检出数字，分别以10进制和16进制方式打印出来。

另外，您可以参考stdio_test_lib目录下的代码，写出自己的测试程序。这个目录里面的文件基本与本试验一样，只是在lib目录下仅仅保留了libc.a库文件。

最后，sys/lib/stdio目录中存放的是标准输入、输出的代码，其中的Makefile可以生成libc.a文件并存放在sys/lib目录下，以后本人编写的库函数都也将存放在此目录下。

9、实验九：INTERRUPT CONTROLLER

S3C2440数据手册354页“**Figure 14-1. Interrupt Process Diagram**”非常简洁地概括了中断处理的流程，我把这个图搬过来，然后结合用到的寄存器用文字解释一下。

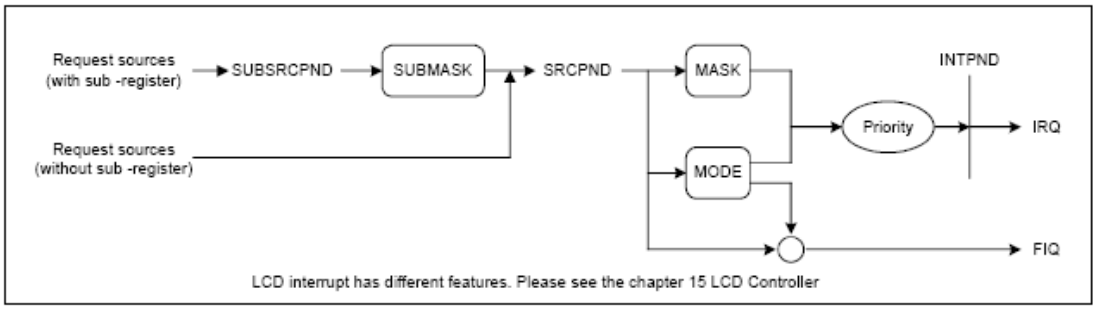


图1 Interrupt Process Diagram

SUBSRCPND和SRCPND寄存器表明有哪些中断被触发了，正在等待处理(pending)；SUBMASK(INTSUBMSK寄存器)和MASK(INTMSK寄存器)用于屏蔽某些中断。图中的“Request sources(with sub -register)”表示的是INT_RXD0、INT_TXD0等11个中断源，它们不同于“Request sources(without sub -register)”：

1、“Request sources(without sub -register)”中的中断源被触发之后，SRCPND寄存器中相应位被置1，如果此中断没有被INTMSK寄存器屏蔽、或者是快中断(FIQ)的话，它将被进一步处理

2、对于“Request sources(with sub -register)”中的中断源被触发之后，SUBSRCPND寄存器中的相应位被置1，如果此中断没有被INTSUBMSK寄存器屏蔽的话，它在SRCPND寄存器中的相应位也被置1，之后的处理过程就和“Request sources(without sub -register)”一样了。

继续沿着图1前进：在SRCPND寄存器中，被触发的中断的相应位被置1，等待处理：

1、如果被触发的中断中有快中断(FIQ)——MODE(INTMOD寄存器)中为1的位对应的中断是FIQ，则CPU的FIQ中断函数被调用。注意：FIQ只能分配一个，即INTMOD中只能有一位设为1。

2、对于一般中断IRQ，可能同时有几个中断被触发，未被INTMSK寄存器屏蔽的中断经过比较后，选出优先级最高的中断——此中断在INTPND寄存器中的相应位被置1，然后CPU调用IRQ中断处理函数。中断处理函数可以通过读取INTPND寄存器来确定中断源是哪个，也可以读INTOFFSET寄存器来确定中断源。

请打开S3C2440数据手册357页，“**Figure 14-2. Priority Generating Block**”显示了各中断源先经过6个一级优先级仲裁器选出各自优先级最高的中断，然后再经过二级优先级仲裁器选从中选出优先级最高的中断。IRQ的中断优先级由RIORITY寄存器设定，请参考数据手册365页，RIORITY寄存器中ARB_SELn(n从0到6)用于设定仲裁器n各输入信号的中断优先级，例如ARB_SEL6[20:19](0最高，其后各项依次降低)：

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

RIORITY寄存器还有一项比较特殊的功能，如果ARB_MODEn设为1，则仲裁器n中输入的中断信号的优先级别将会轮换。例如ARB_MODE6设为1，则仲裁器6的6个输入信号的优先级将如下轮换(见数据手册358页):

If REQ0 or REQ5 is serviced, ARB_SEL bits are not changed at all.
If REQ1 is serviced, ARB_SEL bits are changed to 01b.
If REQ2 is serviced, ARB_SEL bits are changed to 10b.
If REQ3 is serviced, ARB_SEL bits are changed to 11b.
If REQ4 is serviced, ARB_SEL bits are changed to 00b.

意思即是：

REQ0和REQ5的优先级不会改变
当REQ1中断被处理后，ARB_SEL6 = 0b01，即REQ1的优先级变成本仲裁器中最低的(除去REQ5)

当REQ2中断被处理后，ARB_SEL6 = 0b10，即REQ2的优先级变成本仲裁器中最低的(除去REQ5)
当REQ3中断被处理后，ARB_SEL6 = 0b11，即REQ3的优先级变成本仲裁器中最低的(除去REQ5)
当REQ4中断被处理后，ARB_SEL6 = 0b00，即REQ4的优先级变成本仲裁器中最低的(除去REQ5)

现在来总结一下使用中断的步骤：

1、当发生中断IRQ时，CPU进入“中断模式”，这时使用“中断模式”下的堆栈；当发生快中断FIQ时，CPU进入“快中断模式”，这时使用“快中断模式”下的堆栈。所以在使用中断前，先设置好相应模式下的堆栈。

2、对于“Request sources(without sub -register)”中的中断，将INTSUBMSK寄存器中相应位设为0

3、将INTMSK寄存器中相应位设为0

4、确定使用此的方式：是FIQ还是IRQ。

a．如果是FIQ，则在INTMOD寄存器设置相应位为1

b．如果是IRQ，则在RIORITY寄存器中设置优先级

5、准备好中断处理函数，

a．中断向量：

在中断向量设置好当FIQ或IRQ被触发时的跳转函数， IRQ、FIQ的中断向量地址分别为0x00000018、0x0000001c(数据手册79页“**Table 2-3. Exception Vectors**”)

b．对于IRQ，在跳转函数中读取INTPND寄存器或INTOFFSET寄存器的值来确定中断源，然后调用具体的处理函数

c．对于FIQ，因为只有一个中断可以设为FIQ，无须判断中断源

d．中断处理函数进入和返回时需要花点心思：

i．对于IRQ，进入和返回的代码如下：

```
sub lr, lr, #4                  @计算返回地址
stmdb sp!, { r0-r12,lr }        @保存使用到的寄存器
… …
ldmia sp!, { r0-r12,pc }^       @中断返回
                                @^表示将spsr的值赋给cpsr
```

ii．对于FIQ，进入和返回的代码如下：

```
sub lr, lr, #4                    @计算返回地址
stmdb sp!, { r0-r7,lr }           @保存使用到的寄存器
… …
ldmia sp!, { r0-r7,pc }^          @快中断返回，
                                   @^表示将spsr的值赋给cpsr
```

iii. 中断返回之前需要清中断：往SUBSRCPND(用到的话)、SRCPND 、INTPND中相应位写1即可。对于INTPND，最简单的方法就是“INTPND=INTPND”。

6、设置CPSR寄存器中的F-bit(对于FIQ)或I-bit(对于IRQ)为0，开中断

本实验使用按键K1-K4作为4个外部中断——EINT1-3、EINT7，当Kn按下时，通过串口输出“EINTn,Kn pressed!”，主程序让4个LED轮流从0到15计数。对于外部中断，除了上面说的几个寄存器外，还有几个寄存器需要设置，请打开数据手册276页“EXTERNAL INTERRUPT CONTROL REGISTER (EXTINTn)”：

1、EXTINT0-2：它们用于设置EINT0-24共25个外部中断分别是低电平触发、高电平触发、上升沿触发、下降沿触发或者“上升/下降沿触发”(不知道确切的术语)。对于本实验，使用默认值：低电平触发。

2、EINTFLT0-3：未用

3、EINTMASK、EINTPEND：呵呵，这对寄存器和上面的INTMSK、INTPND实在相似。EINTMASK用于设置是否评比EINT4-23；EINTPEND表明有几个外部中断已经发生，正在等待处理(pending)，对某位写入1可以让此位清零。对于本实验，EINTMASK位[7]设为0——使用EINT7。

4、GSTATUS0-4：未用

本实验代码在INT目录下，下面摘取与中断相关的代码：

(1)、head.s中：

```
… …
    msr cpsr_c, #0xd2             @进入中断模式
    ldr sp, =0x33000000           @设置中断模式堆栈
    msr cpsr_c, #0xdf             @进入系统模式
    ldr sp, =0x34000000           @设置系统模式堆栈
    bl init_irq                   @调用中断初始化函数，在init.c中
    msr cpsr_c, #0x5f             @设置I-bit=0，开IRQ中断
… …
HandleIRQ:                        @IRQ中断向量跳转函数
```

```
    sub lr, lr, #4                  @计算返回地址
    stmdb sp!, { r0-r12,lr }        @保存使用到的寄存器
    ldr lr, =int_return             @设置返回地址
    ldr pc, =EINT_Handle            @调用中断处理函数EINT_Handle，
                                    @在interrupt.c中
int_return:
    ldmia sp!, { r0-r12,pc }^       @中断返回,
                                    @^表示将spsr的值复制到cpsr
```

(2)、init.c的中断初始化函数init_irq：

```
#define EINT1 (2<<(1*2))
#define EINT2 (2<<(2*2))
#define EINT3 (2<<(3*2))
#define EINT7 (2<<(7*2))
void init_irq( )
{
    GPFCON |= EINT1 | EINT2 | EINT3 | EINT7;    //K1-K4对应
    //EINT1-3和EINT7
    GPFUP |= (1<<1) | (1<<2) | (1<<3) | (1<<7); //上拉
    EINTMASK &= (~0x80);        //EINT7使能，对于外部中断EINT4-23，
    //除下面的INTMSK外，还须设置EINTMASK
    INTMSK &= (~0x1e);              //EINT1-3、4-7使能(EINT4-7共用INTMSK[4])
    PRIORITY &= (~0x03);            //设定优先级
}
```

(3)、interrupt.c：

```
void EINT_Handle()
{
    unsigned long oft = INTOFFSET;
    switch( oft )
    {
        case 1: printk("EINT1, K1 pressed!\n\r"); break;
        case 2: printk("EINT2, K2 pressed!\n\r"); break;
        case 3: printk("EINT3, K3 pressed!\n\r"); break;
        case 4: printk("EINT7, K4 pressed!\n\r"); break;
        default: printk("Interrupt unknown!\n\r"); break;
    }
    //以下清中断
    if( oft == 4 ) EINTPEND = 1<<7;   //EINT4-7合用IRQ4，
    //注意：EINTPEND[3:0]保留未用，
    //向这些位写入1可能导致未知结果
    SRCPND = 1<<oft;                //向当前处理的中断对应的位写1清0
    INTPND = INTPND;            //清0
}
```

实验步骤：

1、进入INT目录运行“make”命令生成可执行文件int，下载到板子上

2、打开minicom(拨特率：57600,8N1)

3、按下按钮即可看见串口输出

4、您可以同时按住几个按钮观察串口输出，可以发现K1优先级最高，K4最低

10、实验十：TIMER

本实验利用TIMER0触发中断，使4个LED 1秒钟闪一次。先简述一下TIMER的工作过程。请打开数据手册284页，我在“**Figure 10-1. 16-bit PWM Timer Block Diagram**”上进行解说。摘录此图，见下面图3。

对于TIMER0，PCLK被“8-Bit Prescaler”(在TCFG0中定义)分频后，进入“Clock Divider”，“Clock Divider”有5种频率输出：从“8-Bit Prescaler”进来的时钟的2分频、4分频、8分频、16分频，或者外部时钟TCLK0。后面的“5:1 MUX”(5选1的选择器)使用TCFG1来确定使用哪种频率。“Control Logic0”在“5:1 MUX”的输出的频率下工作。对于“Control Logic0”，我得画图说明：

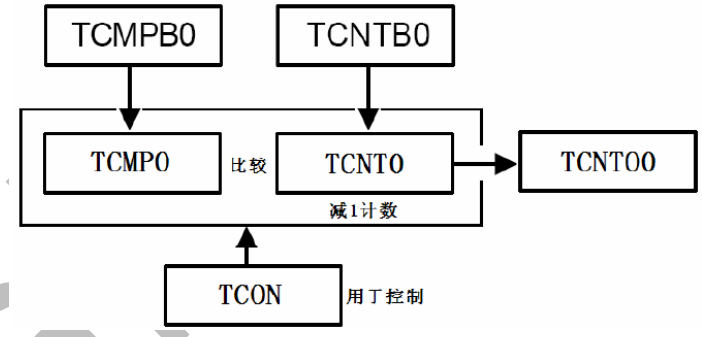


图2 **Control Logic0**内部结构

“Control Logic0”的工作流程为(请参考数据手册287页“**TIMER INITIALIZATION USING MANUAL UPDATE BIT AND INVERTER BIT**”)：

1、上电之后，设置TCMPB0和TCNTB0寄存器

2、设置TCON寄存器bit[1]为1，这样，TCMPB0和TCNTB0寄存器的值就被装入TCMP0和TCNT0寄存器中；设置bit[2]确定是否使用“Timer 0 output inverter”

3、设置TCON寄存器bit[0]启动TIMER0，设置bit[3]确定是否使用“Timer 0 auto reload”

4、当TIMER0启动后，TCNT0在“5:1 MUX”输出的时钟下，进行减1计数。当

TCNT0等于TCMP0时：TOUT0输出翻转；当TCNT0等于0时：TOUT0输出翻转，TIMER0中断被触发(如果此中断使能了的话)，TCMPB0和TCNTB0寄存器的值被自动装入TCMP0和TCNT0寄存器中(如果TCON寄存器bit[3]等于1的话)——此时下个计数流程开始

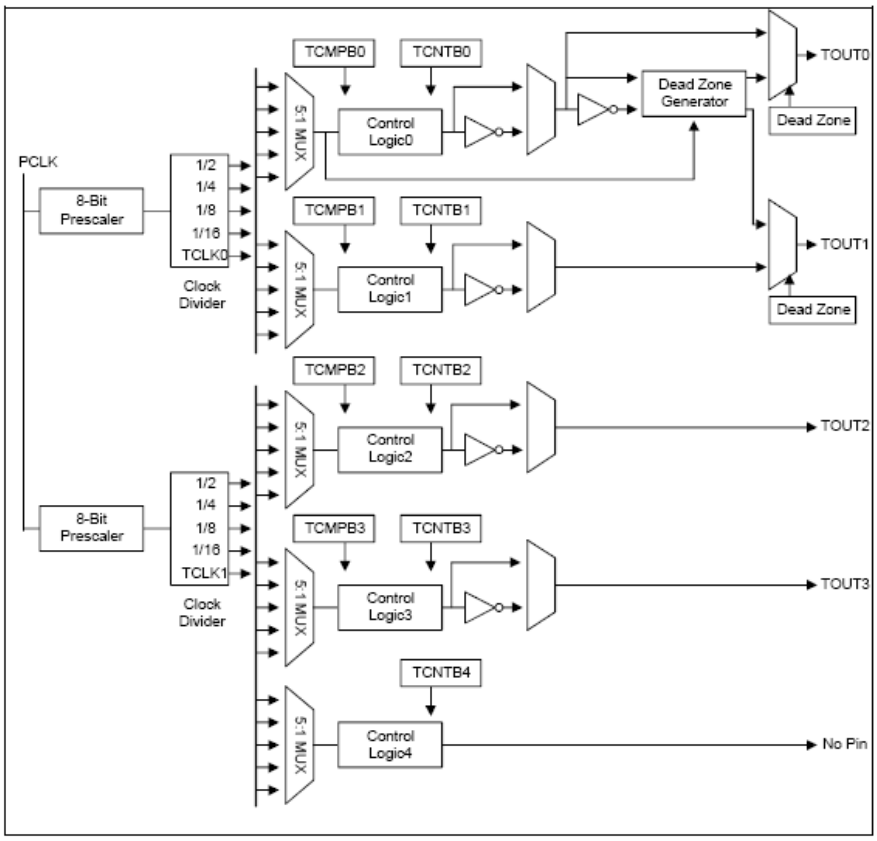


图3 16-bit PWM Timer Block Diagram

现在介绍一下上面说到的几个寄存器：

1、TCFG0和TCFG1：分别设为119和0x03

这连个寄存器用于设置“Control Logic”的时钟，计算公式如下：

Timer input clock Frequency = PCLK / {prescaler value+1} / {divider value}

对于TIMER0，prescaler value = TCFG0[7:0]，divider value由TCFG[3:0]确定(0b000：2，0b001：4，0b010：8，0b0011：16，0b01xx：使用外部TCLK0)。

对于本实验，TIMER0时钟 = 12MHz/(119+1)/(16) = 6250Hz

2、TCNTB0：设为3125，在6250Hz的频率下，此值对应的时间为0.5S

3、TCON：

TIMER0对应bit[3:0]：

bit[3]用于确定在TCNT0计数到0时，是否自动将TCMPB0和TCNTB0寄存器的值装入TCMP0和TCNT0寄存器中

bit[2]用于确定TOUT0是否反转输出(本实验未用)

bit[1]用于手动更新TCMP0和TCNT0寄存器：在第一次使用定时器前，此位需要设为1，此时TCMPB0和TCNTB0寄存器的值装入TCMP0和TCNT0寄存器中

bit[0]用于启动TIMER0

4、TCONO0：只读寄存器，用于读取当前TCON0寄存器的值，本实验未用

本实验的代码在TIMER目录下，init.c中的Timer0_init函数初始化并启动TIMER0：

```
void Timer0_init()
{
    TCFG0 = 119;              //Prescaler0 = 119
    TCFG1 = 0x03;             //Select MUX input for PWM Timer0:divider=16
    TCNTB0 = 3125;            //0.5秒钟触发一次中断
    TCON |= (1<<1);           //Timer 0 manual update
    TCON = 0x09;              /*Timer 0 auto reload on Timer 0 output inverter off
    清"Timer 0 manual update" Timer 0 start */
}
```

init.c中的init_irq函数使能TIEMR0中断：

```
void init_irq( )
{
    INTMSK &= (~(1<<10)); //INT_TIMER0中断使能
}
```

interrupt.c中的Timer0_Handle函数用于处理TIMER0中断：每0.5s发生一次中断，中断发生时将4个LED的状态反转，即1s闪一次：

```
void Timer0_Handle()
{
    if(INTOFFSET == 10){
    GPBDAT = ~(GPBDAT & (0xf << 7));
    }
    //清中断
    SRCPND = 1 << INTOFFSET;
    INTPND = INTPND;
}
```

11、实验十一：MMU

在理论上概括或解释MMU，这不是我能胜任的。我仅基于为了理解本实验中操作MMU的代码而对MMU做些说明，现在先简单地描述虚拟地址(VA)、变换后的虚拟地址(MVA)、物理地址(PA)之间的关系：

启动MMU后，S3C2440的CPU核看到的、用到的只是虚拟地址VA，至于VA如何最终落实到物理地址PA上，CPU是不理会的。而caches和MMU也是看不见VA的，它们利用VA变换得来的MVA去进行后续操作——转换成PA去读/写实际内存芯片，MVA是除CPU外的其他部分看见的虚拟地址。对于VA与MVA之间的变换关系，请打开数据手册551页，我摘取了“**Figure 2-8. Address Mapping Using CP15 Register 13**”：

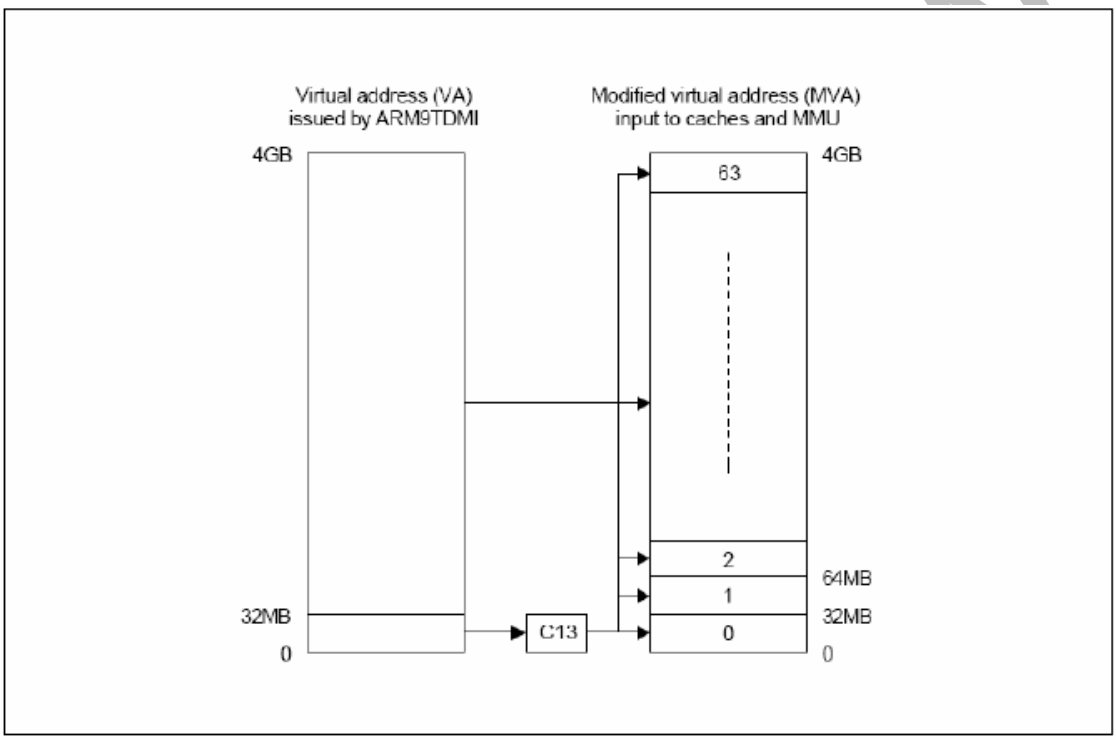


图4 VA与MVA的关系

如果VA<32M，需要使用进程标识号PID(通过读CP15的C13获得)来转换为MVA。VA与MVA的转换方法如下(这是硬件自动完成的)：

```
if(VA < 32M) then
      MVA = VA | (PID << 25) //VA < 32M
else
     MVA = VA //VA >= 32M
```

利用PID生成MVA的目的是为了减少切换进程时的代价：如果两个进程占用的虚拟地址空间(VA)有重叠，不进行上述处理的话，当进行进程切换时必须进行虚拟地址到物理地址的重新影射，这需要重建页表、使无效caches和TLBS等等，代价非

常大。但是如果像上述那样处理的话，进程切换就省事多了：假设两个进程1、2运行时的VA都是0-32M，则它们的MVA分别是(0x02000000-0x03ffffff)、(0x04000000-0x05ffffff)——前面说过MMU、Caches使用MVA而不使用VA，这样就不必进行重建页表等工作了。

现在来讲讲MVA到PA的变换过程：请打开数据手册557页，“**Figure 3-1. Translating Page Tables**”(见下述图5)非常精练地概括了对于不同类型的页表，MVA是如何转换为PA的。图中的页表“Translation table”起始地址为“TTB base”，在建立页表后，写入CP15的寄存器C2。

使用MVA[31:20]检索页表“Translation table”得到一个页表项(entry，4字节)，根据此entry的低2位，可分为以下4种：

1、0b00：无效
2、0b01：粗表(Coarse page)

entry[31:10]为粗表基址(Coarse page table base address)，据此可以确定一块1K大小的内存——称为粗页表(Coarse page table,见图5)。

粗页表含256个页表项，每个页表项对应一块4K大小的内存，每个页表项又可以分为大页描述符、小页描述符。MVA[19:12]用来确定页表项。一个大页(64K)对应16个大页描述符，这16个大页描述符相邻且完全相同，entry[31:16]为大页基址(Large page base)。MVA[15:0]是大页内的偏移地址。一个小页(4K)对应1个小页描述符，entry[31:12]为小页基址(Small page base)。MVA[11:0]是小页内的偏移地址。

3、0b10：段(Section)

段的操作最为简单，entry[31:20]为段基址(Section base)，据此可以确定一块1M大小的内存(Section，见图5)，而MVA[19:0]则是块内偏移地址

4、0b11：细表(Fine page)

entry[31:12]为细表基址(Fine page table base address)，据此可以确定一块4K大小的内存——称为细页表(Fine page table,见图5)。

细页表含1024个页表项，每个页表项对应一块1K大小的内存，每个页表项又可以分为大页描述符、小页描述符、极小页描述符。MVA[19:10]用来确定页表项。一个大页(64K)对应64个大页描述符，这64个大页描述符相邻且完全相同，entry[31:16]为大页基址(Large page base)。MVA[15:0]是大页内的偏移地址。一个小页(4K)对应4个小页描述符，entry[31:12]为小页基址(Small page base)。MVA[11:0]是小页内的偏移地址。极小页(1K)对应1个极小页描述符，entry[31:10]为极小页基址(Tiny page base) 。MVA[9:0]是极小页内的偏移地址。

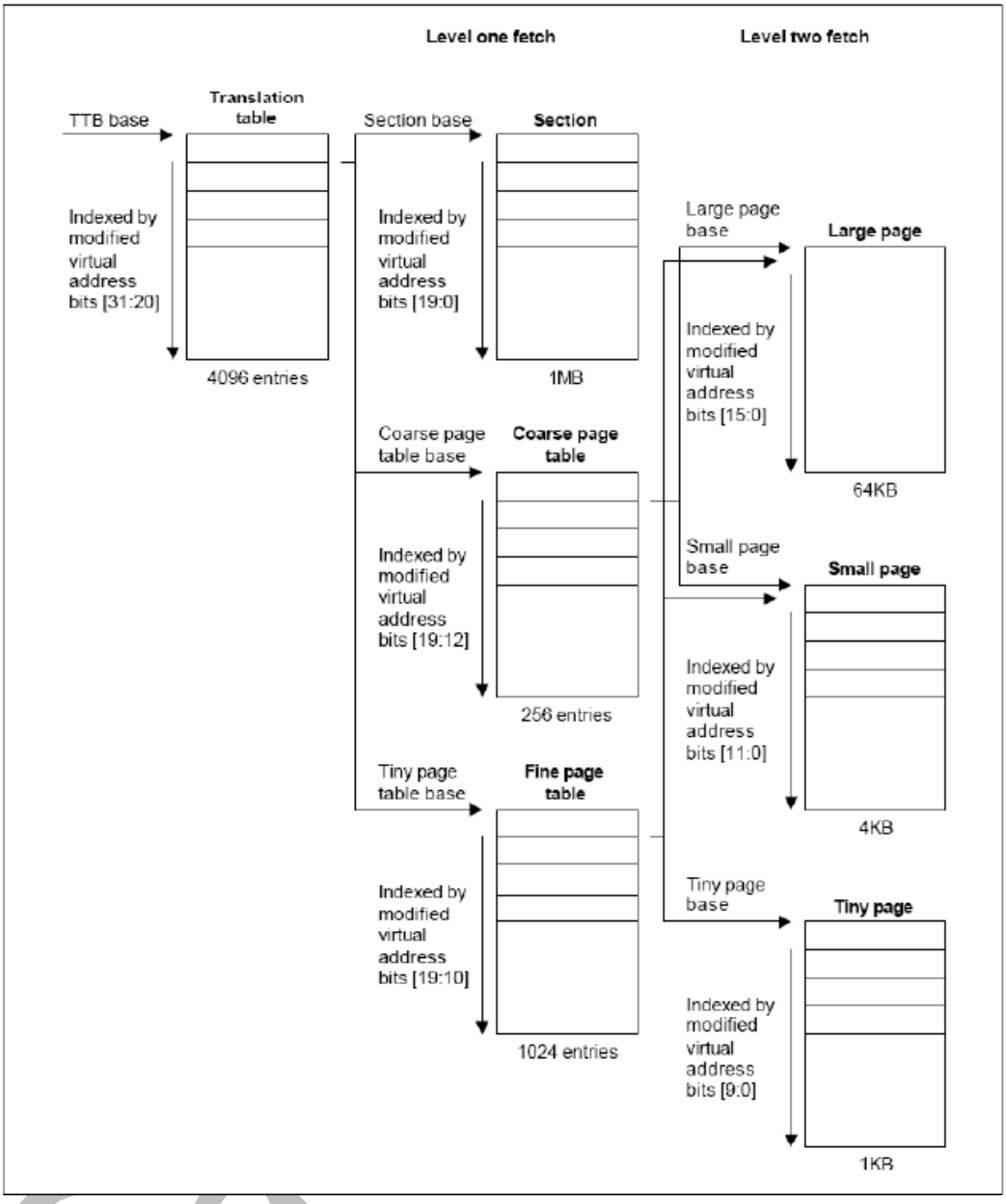


图5 Translating Page Tables

访问权限的检查是MMU主要功能之一，它由描述符的AP和domain、CP15寄存器C1的R/S/A位、CP15寄存器C3(域访问控制)等联合作用。本实验不使用权限检查(令C3为全1)。

下面简单介绍一下使用Cache和Write buffer：

1、“清空”(clean)的意思是把Cache或Write buffer中已经脏的(修改过，但未写入主存)数据写入主存

2、“使无效”(invalidate)：使之不能再使用，并不将脏的数据写入主存

3、对于I/O影射的地址空间，不使用Cache和Write buffer

4、在使用MMU前，使无效Cache和drain write buffer

与cache类似，在使用MMU前，使无效TLB。

上面有些部分讲得很简略，除了作者水平不足之外，还在于本书的侧重点——实验。理论部分就麻烦各位自己想办法了。不过这些内容也足以了解本实验的代码了。本实验与实验9完成同样的功能：使用按键K1-K4作为4个外部中断——EINT1-3、EINT7，当Kn按下时，通过串口输出“EINTn,Kn pressed!”，主程序让4个LED轮流从0到15计数。代码在目录MMU下。下面摘取与MMU相关的代码详细说明。

先看看head.s代码(将一些注释去掉了)：

```
    b Reset
HandleUndef:
    b HandleUndef
HandleSWI:
    b HandleSWI
HandlePrefetchAbort:
    b HandlePrefetchAbort
HandleDataAbort:
    b HandleDataAbort
HandleNotUsed:
    b HandleNotUsed
    ldr pc, HandleIRQAddr
HandleFIQ:
    b HandleFIQ
HandleIRQAddr:
    .long HandleIRQ
Reset:                                  @函数disable_watch_dog, memsetup, init_nand,
                                        @nand_read_ll在init.c中定义
    ldr sp, =4096                       @设置堆栈
    bl disable_watch_dog            @关WATCH DOG
    bl memsetup_2                       @初始化SDRAM
    bl init_nand                            @初始化NAND Flash
    bl copy_vectors_from_nand_to_sdram  @在init.c中
    bl copy_process_from_nand_to_sdram  @在init.c中
    ldr sp, =0x30100000                 @重新设置堆栈
                                        @(因为下面就要跳到SDRAM中执行了)
    ldr pc, =run_on_sdram               @跳到SDRAM中
run_on_sdram:
    bl mmu_tlb_init                     @调用C函数mmu_tlb_init(mmu.c中)，建立页表
```

```
    bl mmu_init                 @调用C函数mmu_init(mmu.c中)，使能MMU
    msr cpsr_c, #0xd2           @进入中断模式
    ldr sp, =0x33000000         @设置中断模式堆栈
    msr cpsr_c, #0xdf           @进入系统模式
    ldr sp, =0x30100000         @设置系统模式堆栈
    bl init_irq                 @调用中断初始化函数，在init.c中
    msr cpsr_c, #0x5f           @设置I-bit=0，开IRQ中断
    ldr lr, =halt_loop          @设置返回地址
    ldr pc, =main               @b指令和bl指令只能前后跳转32M的范围，
                                @所以这里使用向pc赋值的方法进行跳转
halt_loop:
    b halt_loop
HandleIRQ:
    sub lr, lr, #4              @计算返回地址
    stmdb sp!, { r0-r12,lr }    @保存使用到的寄存器
    ldr lr, =int_return         @设置返回地址
    ldr pc,=EINT_Handle         @调用中断处理函数，在interrupt.c中
int_return:
    ldmia sp!, { r0-r12,pc }^   @中断返回,
                                @^表示将spsr的值复制到cpsr
```

请注意第12、15行，我们将IRQ中断向量由以前的“b HandleIRQ”换成了：

```
    12 ldr pc, HandleIRQAddr
    15 HandleIRQAddr:
    16 .long HandleIRQ
```

这是因为b跳转指令只能前后跳转32M的范围，而本实验中中断向量将重新放在VA=0xffff0000开始处(而不是通常的0x00000000)，到HandleIRQAddr的距离远远超过了32M。将中断向量重新定位在0xffff0000处，是因为MMU使能后，中断发生时：

1、如果中断向量放在0x00000000处，则对于不同的进程(PID)，中断向量的MVA将不同

2、如果中断向量放在0xffff0000处，则对于不同的进程(PID)，中断向量的MVA也相同

显然，如果使用1，则带来的麻烦非常大——对于每个进程，都得设置自己的中断向量。所以MMU使能后，处理中断的方法应该是2。

第22行copy_vectors_from_nand_to_sdram函数将中断向量复制到内存物理地址0x33ff0000处，在mmu_tlb_init函数中会把0x33ff0000影射为虚拟地址0xffff0000。

第23行copy_process_from_nand_to_sdram函数将存在nand flash开头的4K代码全部复

制到0x30004000处(本实验的连接地址为0x30004000)。请注意SDRAM起始地址为0x30000000，前面的16K空间用来存放一级页表(在mmu_tlb_init中设置)。

第27行mmu_tlb_init函数设置页表。本实验以段的方式使用内存，所以仅使用一级页表，且页表中所有页表项均为段描述符。

mmu_tlb_init代码(在mmu.c中)如下：

```
void mmu_tlb_init()
{
    unsigned long entry_index;
    /*SDRAM*/
    for(entry_index = 0x30000000; entry_index < 0x34000000; entry_index+=0x100000)
    {
    /*section table's entry:AP=0b11,domain=0,Cached,write-through mode(WT)*/
    *(mmu_tlb_base+(entry_index>>20)) =
        entry_index |(0x03<<10)|(0<<5)|(1<<4)|(1<<3)|0x02;
    }
    /*SFR*/
    for(entry_index = 0x48000000; entry_index < 0x60000000; entry_index += 0x100000)
    {
    /*section table's entry:AP=0b11,domain=0,NCNB*/
    *(mmu_tlb_base+(entry_index>>20)) =
        entry_index |(0x03<<10)|(0<<5)|(1<<4)| 0x02;
    }
    /*exception vector*/
    /*section table's entry:AP=0b11,domain=0,Cached,write-through mode(WT)*/
    *(mmu_tlb_base+(0xffff0000>>20)) =
    (VECTORS_PHY_BASE) |(0x03<<10)|(0<<5)|(1<<4)|(1<<3)|0x02;
}
```

第4-8行令64M SDRAM的虚拟地址和物理地址相等——从0x30000000到0x33ffffff，这样可以使得在head.s中第28行调用mmu_init使能MMU前后的地址一致。

第9-13行设置特殊功能寄存器的虚拟地址，也让它们的虚拟地址和物理地址相等——从0x48000000到0x5fffffff。并且不使用cache和write buffer。

第14-17行设置中断向量的虚拟地址，虚拟地址0xfff00000对应物理地址0x33f00000。

回到head.s中第28行，调用mmu.c中的mmu_init函数使能MMU，此函数代码入下：

```
void mmu_init()
```

```
{
    unsigned long ttb = MMU_TABLE_BASE;
    __asm__(
        "mov r0, #0\n"
    /* invalidate I,D caches on v4 */
        "mcr p15, 0, r0, c7, c7, 0\n"
    /* drain write buffer on v4 */
        "mcr p15, 0, r0, c7, c10, 4\n"
    /* invalidate I,D TLBs on v4 */
        "mcr p15, 0, r0, c8, c7, 0\n"
    /* Load page table pointer */
        "mov r4, %0\n"
        "mcr p15, 0, r4, c2, c0, 0\n"
    /* Write domain id (cp15_r3) */
        "mvn r0, #0\n" /*0b11=Manager，不进行权限检查*/
        "mcr p15, 0, r0, c3, c0, 0\n"
    /* Set control register v4 */
        mrc p15, 0, r0, c1, c0, 0\n"
    /* Clear out 'unwanted' bits */
        "ldr r1, =0x1384\n"
        "bic r0, r0, r1\n"
    /* Turn on what we want */
    /*Base location of exceptions = 0xffff0000*/
        "orr r0, r0, #0x2000\n"
    /* Fault checking enabled */
        "orr r0, r0, #0x0002\n"
        #ifdef CONFIG_CPU_D_CACHE_ON /*is not set*/
        "orr r0, r0, #0x0004\n"
        #endif
       #ifdef CONFIG_CPU_I_CACHE_ON /*is not set*/
        "orr r0, r0, #0x1000\n"
        #endif
    /* MMU enabled */
        "orr r0, r0, #0x0001\n"
    /* write control register *//*write control register P545*/
        "mcr p15, 0, r0, c1, c0, 0\n"
        : /* no outputs */
        : "r" (ttb) );
}
```

此函数使用嵌入汇编的方式，第29行的"r" (ttb)表示变量ttb的值赋给一个寄存起作为输入参数，这个寄存器由编译器自动分配；第13行的“%0”表示这个寄存器 。MMU控制寄存器C1中各位的含义(第18-37行)，可以参考539页“**Table 2-10. Control Register 1-bit Functions**”。如果想详细了解本函数用到的操作协处理器的指令，可以参

考数据手册529页“**Appendix 2 PROGRAMMER'S MODEL**”。

本实验代码在CLOCK目录下，运行make命令后将可执行文件mmu下载、运行。然后将mmu.h文件中如下两行的注释去掉，重新make后下载运行mmu，可以发现LED闪烁的速度变快了很多——这是因为使用了cache(见上面代码28-33行)：

```
#define CONFIG_CPU_D_CACHE_ON 1
#define CONFIG_CPU_I_CACHE_ON 1
```

12、实验十二：CLOCK

S3C2440 CPU主频可以达到266MHz，前面的实验都没有使用PLL，CPU的频率只有12MHz。本实验在实验10的基础上启动PLL，使得FCLK=200MHz，HCLK=100MHz，PCLK=50MHz。

请打开数据手册219页第7章“CLOCK & POWER MANAGEMENT”。S3C2440有两个PLL：MPLL和UPLL，UPLL专用与USB设备，本实验介绍的是MPLL——用于设置FCLK、HCLK、PLCK。FCLK用于CPU核，HCLK用于AHB总线的设备(比如SDRAM)，PCLK用于APB总线的设备(比如UART)。请打开数据手册224页，“**Figure 7-4. Power-On Reset Sequence (when the external clock source is a crystal oscillator)**”展示了上电后， MPLL启动的过程，摘录此图如下：

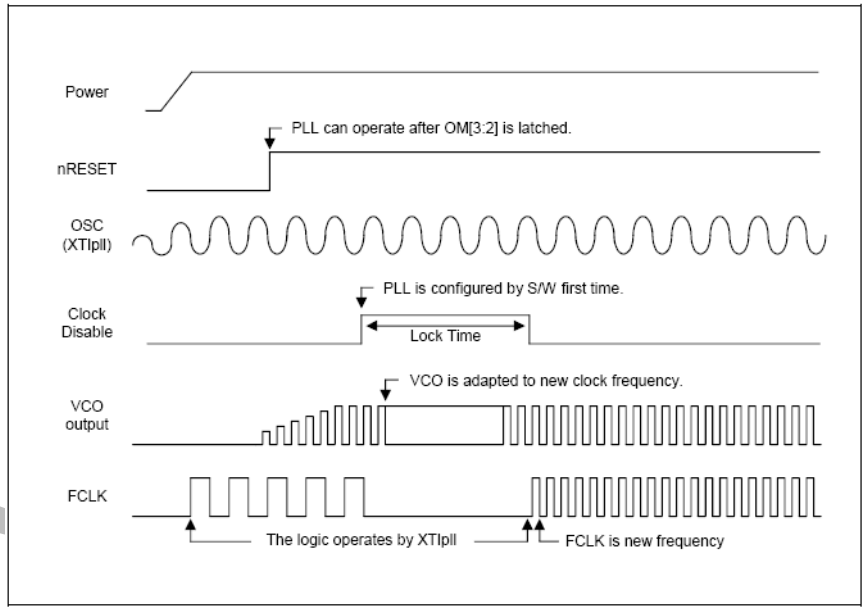


图6 上电后**MPLL**的启动过程

请跟随FCLK的图像了解启动过程：

1、上电几毫秒后，晶振输出稳定，FCLK=晶振频率，nRESET信号恢复高电平后，CPU开始执行指令。

2、我们可以在程序开头启动MPLL，在设置MPLL的几个寄存器后，需要等待一段时间(Lock Time)，MPLL的输出才稳定。在这段时间(Lock Time)内，FCLK停振CPU停止工作。Lock Time的长短由寄存器LOCKTIME设定。

3、Lock Time之后，MPLL输出正常，CPU工作在新的FCLK下。

本实验的工作就是设置MPLL的几个寄存器：

1)、LOCKTIME：设为0x00ffffff

前面说过，MPLL启动后需要等待一段时间(Lock Time)，使得其输出稳定。位[23:12]用于UPLL，位[11:0]用于MPLL。本实验使用确省值0x00ffffff。

2)、CLKDIVN：设为0x03,用于设置FCLK、HCLK、PCLK三者的比例：

bit[2]——HDIVN1，若为1，则bit[1:0]必须设为0b00，此时FCLK:HCLK:PCLK=1:1/4:1/4；若为0，三者比例由bit[1:0]确定
bit[1]——HDIVN，0：HCLK=FCLK；1：HCLK=FCLK/2
bit[0]——PDIVN，0：PCLK=HCLK；1：PCLK=HCLK/2

本实验设为0x03，则FCLK:HCLK:PCLK=1:1/2:1/4

3)、请翻到数据手册226页，有这么一段：

```
If HDIVN = 1, the CPU bus mode has to be changed from the fast bus mode to the
asynchronous bus mode using following instructions：
MMU_SetAsyncBusMode
mrc p15, 0, r0, c1, c0, 0
orr r0, r0, #R1_nF:OR:R1_iA
mcr p15,0, r0, c1, c0, 0
```

其中的“R1_nF:OR:R1_iA”等于0xc0000000。意思就是说，当HDIVN = 1时，CPU bus mode需要从原来的“fast bus mode”改为“asynchronous bus mode”。

4)、MPLLCON：设为(0x5c << 12)|(0x04 << 4)|(0x00)，即0x5c0040

对于MPLLCON寄存器，[19:12]为MDIV，[9:4]为PDIV，[1:0]为SDIV。有如下计算公式：

```
MPLL(FCLK) = (m * Fin)/(p * 2^s)
```

其中: m = MDIV + 8, p = PDIV + 2

对于本开发板，Fin = 12MHz,MPLLCON设为0x5c0040，可以计算出FCLK=200MHz，再由CLKDIVN的设置可知：HCLK=100MHz，PCLK=50MHz。

当设置MPLLCON之后——相当于图6中的“PLL is configured by S/W first time”，Lock Time就被自动插入，Lock Time之后，MPLL输出稳定，CPU工作在200MHz频率下。

上面的4个步骤的工作在init.c中的clock_init函数中完成，代码如下：

```
void clock_init()
{
    LOCKTIME = 0x00ffffff;
    CLKDIVN = 0x03; /*FCLK:HCLK:PCLK=1:2:4, HDIVN1=0,HDIVN=1,PDIVN=1
*/
/*If HDIVN = 1,the CPU bus mode has to be changed from the fast bus mode to the
asynchronous bus mod using following instructions.*/
__asm__(
    "mrc p15, 0, r1, c1, c0, 0\n"          /* read ctrl register */
    "orr r1, r1, #0xc0000000\n"            /* Asynchronous */
    "mcr p15, 0, r1, c1, c0, 0\n"          /* write ctrl register */
    );
    MPLLCON = MPLL_200MHz; /*CLK=200MHz,HCLK=100MHz,PCLK=50MHz*/
}
```

在head.s中，在memsetup_2函数之前调用clock_init函数。

本实验是在实验10的基础上修改得来，当工作频率改变后，一些设备的初始化参数需要调整，很幸运，只需要修改两个地方：

1、SDRAM控制器的REFRESH寄存器(在init.c中)：

请翻看实验五：MEMORY CONTROLLER中关于REFRESH寄存器的介绍，本实验HCLK=100MHz，REFRESH寄存器取值如下：

```
R_CNT = 2^11 + 1 – 100 * 7.8125 = 1268,
REFRESH=0x008e0000 + 1268 = 0x008e04f4
```

在init.c中的memsetup_2函数，将原来的0x008e07a3换成0x008e04f4即可。

2、UART0的UBRDIV0寄存器(在serial.c中)：

请翻看实验七：UART中关于UBRDIV0的介绍，本实验PCLK=50MHz，设置波特率为57600时，UART0寄存器取值可由下式计算得53：

```
UBRDIVn = (int)(PCLK / (bps x 16) ) –1
```

在serial.c中的init_uart函数，将UART0由原来的12换成53即可。

本实验代码在CLOCK目录下，运行make命令后将可执行文件clock下载、运行，可以发现LED闪烁的速度变快了很多。然后将mmu.h文件中如下两行的注释去掉，重新make后下载运行clock，可以发现LED闪烁得更快了，简直分辨不出来了——这是因为使用了cache：

```
#define CONFIG_CPU_D_CACHE_ON      1
#define CONFIG_CPU_I_CACHE_ON 1
```

四． Bootloader vivi

为了将linux移植到ARM上，碰到的第一个程序就是bootloader，我选用韩国mizi公司的vivi。您可以在以下地址下载：

http://www.mizi.com/developer/S3C2440x/download/vivi.html

如果您对bootloader没有什么概念，在学习VIVI的代码之前，建议您阅读一篇文章《嵌入式系统 Boot Loader 技术内幕》(詹荣开著)。链接地址如下：

http://www-128.ibm.com/developerworks/cn/linux/l-btloader/

当您阅读了上述文章后，我再企图在理论上罗嗦什么就不合适了(这篇文章实在太好了)。vivi也可以分为2个阶段，阶段1的代码在arch/S3C2440/head.S中，阶段2的代码从init/main.c的main函数开始。您可以跳到实验部分先感受一下vivi。

1、阶段1：arch/S3C2440/head.S

沿着代码执行的顺序，head.S完成如下几件事情：

1)、关WATCH DOG：上电后，WATCH DOG默认是开着的

2)、禁止所有中断：vivi中没用到中断(不过这段代码实在多余，上电后中断默认是关闭的)

3)、初始化系统时钟：启动MPLL，FCLK=400MHz，HCLK=100MHz，PCLK=50MHz，“CPU bus mode”改为“Asynchronous bus mode”。请参考实验十一：CLOCK

4)、初始化内存控制寄存器：还记得那13个寄存器吗？请复习实验五：MEMORY CONTROLLER

5)、检查是否从掉电模式唤醒，若是，则调用WakeupStart函数进行处理——这是一段没用上的代码，vivi不可能进入掉电模式

6)、点亮所有LED

7)、初始化UART0：

a．设置GPIO，选择UART0使用的引脚

b．初始化UART0，设置工作方式(不使用FIFO)、波特率115200 8N1、无流控等，请参考实验七：UART

8)、将vivi所有代码(包括阶段1和阶段2)从nand flash复制到SDRAM中：

a．设置nand flash控制寄存器

b．设置堆栈指针——调用C函数时必须先设置堆栈

c．设置即将调用的函数nand_read_ll的参数：r0=目的地址(SDRAM的地址)，r1=源地址(nand flash的地址)，r2=复制的长度(以字节为单位)

d．调用nand_read_ll进行复制

e．进行一些检查工作：上电后nand flash最开始的4K代码被自动复制到一个称为“Steppingstone”的内部RAM中(地址为0x00000000-0x00001000)；在执行nand_read_ll之后，这4K代码同样被复制到SDRAM中(地址为0x33f00000-0x33f01000)。比较这两处的4K代码，如果不同则表示出错

9)、跳到bootloader的阶段2运行——就是调用init/main.c中的main函数：

a．重新设置堆栈

b．设置main函数的参数

c．调用main函数

如果您做了第二章的各个实验，理解head.S就不会有任何困难——上面说到的几个步骤，都有相应的实验。head.S有900多行，把它搬到这篇文章上来就太不厚道了——白白浪费页数而已。在vivi/arch/S3C2440/head.S上，我做了比较详细的注释(其中的乱码可能是韩文，不去管它)。

当执行完head.S的代码后，内存的使用情况如下：

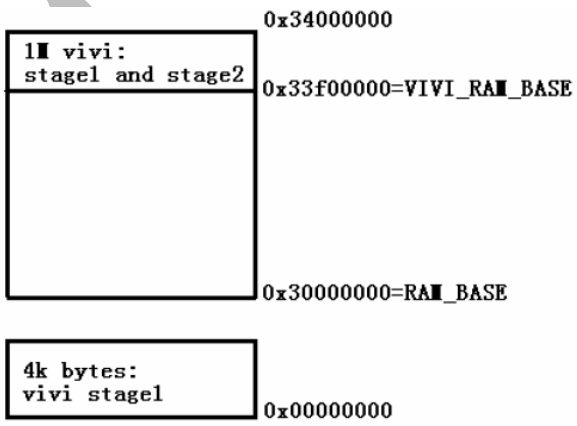


图7 执行vivi stage1后内存的划分情况

2、阶段2：init/main.c

本阶段从init/main.c中的main函数开始执行，它可以分为8个步骤。我先把main函数的代码罗列如下(去掉了乱码——可能是韩文，现在没留下多少有用的注释了)，然后逐个分析：

```
int main(int argc, char *argv[])
{
	int ret;
/*Step 1*/
	putstr("\r\n");
	putstr(vivi_banner);
	reset_handler();
/*Step 2*/
	ret = board_init();
	if (ret) {
		putstr("Failed a board_init() procedure\r\n");
		error();
	}
/*Step 3*/
	mem_map_init();
	mmu_init();
	putstr("Succeed memory mapping.\r\n");
/* Now, vivi is running on the ram. MMU is enabled.*/
/*Step 4*/
/* initialize the heap area */
	ret = heap_init();
	if (ret) {
		putstr("Failed initailizing heap region\r\n");
		error();
	}
/*Step 5*/
	ret = mtd_dev_init();
/*Step 6*/
	init_priv_data();
/*Step 7*/
	misc();
	init_builtin_cmds();
/*Step 8*/
	boot_or_vivi();
	return 0;
}
```


1)、Step 1：reset_handler()

reset_handler用于将内存清零，代码在lib/reset_handle.c中。

[main(int argc, char *argv[]) > reset_handler()]

```
void
reset_handler(void)
{
	int pressed;
	pressed = is_pressed_pw_btn(); /*判断是硬件复位还是软件复位*/
	if (pressed == PWBT_PRESS_LEVEL) {
		DPRINTK("HARD RESET\r\n");
		hard_reset_handle(); /*调用clear_mem对SDRAM清0*/
	} else {
		DPRINTK("SOFT RESET\r\n");
		soft_reset_handle(); /*此函数为空*/
	}
}
```

在上电后，reset_handler调用第8行的hard_reset_handle()，此函数在lib/reset_handle.c中：

[main(int argc, char *argv[]) > reset_handler() > hard_reset_handle()]

```
static void
hard_reset_handle(void)
{
	#if 0
	clear_mem((unsigned long)(DRAM_BASE + VIVI_RAM_ABS_POS), \
	(unsigned long)(DRAM_SIZE - VIVI_RAM_ABS_POS));
	#endif
/*lib/memory.c,将起始地址为USER_RAM_BASE，长度为USER_RAM_SIZE的内存清0*/
	clear_mem((unsigned long)USER_RAM_BASE, (unsigned long)USER_RAM_SIZE);
}
```

2)、Step 2：board_init()

board_init调用2个函数用于初始化定时器和设置各GPIO引脚功能，代码在arch/S3C2440/smdk.c中：

[main(int argc, char *argv[]) > board_init()]

```
int board_init(void)
{
	init_time(); /*arch/S3C2440/proc.c*/
```

```
    set_gpios(); /*arch/S3C2440/smdk.c*/
    return 0;
}
```

init_time()只是简单的令寄存器TCFG0 = 0xf00，vivi未使用定时器，这个函数可以忽略。

set_gpios()用于选择GPA-GPH端口各引脚的功能及是否使用各引脚的内部上拉电阻，并设置外部中断源寄存器EXTINT0-2(vivi中未使用外部中断)。

3)、Step 3：建立页表和启动MMU

mem_map_init函数用于建立页表，vivi使用段式页表，只需要一级页表。它调用3个函数，代码在arch/S3C2440/mmu.c中：

[main(int argc, char *argv[]) > mem_map_init(void)]

```
void mem_map_init(void)
{
    #ifdef  CONFIG_S3C2440_NAND_BOOT  /* CONFIG_S3C2440_NAND_BOOT=y
*/
    mem_map_nand_boot(); /* 最终调用mem_mepping_linear,建立页表 */
    #else
    mem_map_nor();
    #endif
    cache_clean_invalidate(); /* 清空cache,使无效cache */
    tlb_invalidate(); /* 使无效快表TLB */
}
```

第9、10行的两个函数可以不用管它，他们做的事情在下面的mmu_init函数里又重复了一遍。对于本开发板，在.config中定义了CONFIG_S3C2440_NAND_BOOT。mem_map_nand_boot()函数调用mem_mapping_linear()函数来最终完成建立页表的工作。页表存放在SDRAM物理地址0x33dfc000开始处，共16K：一个页表项4字节，共有4096个页表项；每个页表项对应1M地址空间，共4G。mem_map_init先将4G虚拟地址映射到相同的物理地址上NCNB(不使用cache，不使用write buffer)——这样，对寄存器的操作跟未启动MMU时是一样的；再将SDRAM对应的64M空间的页表项修改为使用cache。mem_mapping_linear函数的代码在arch/S3C2440/mmu.c中：

[main(int argc, char *argv[]) > mem_map_init(void) > mem_map_nand_boot( ) > mem_mapping_linear(void)]

```
static inline void mem_mapping_linear(void)
{
```

```
    unsigned long pageoffset, sectionNumber;
    putstr_hex("MMU table base address = 0x", (unsigned long)mmu_tlb_base);
/* 4G 虚拟地址映射到相同的物理地址. not cacacheable, not bufferable */
/* mmu_tlb_base = 0x33dfc000*/
    for (sectionNumber = 0; sectionNumber < 4096; sectionNumber++) {
        pageoffset = (sectionNumber << 20);
        *(mmu_tlb_base + (pageoffset >> 20)) = pageoffset | MMU_SECDESC;
    }
/* make dram cacheable */
/* SDRAM物理地址0x3000000-0x33ffffff,
    DRAM_BASE=0x30000000,DRAM_SIZE=64M
*/
    for (pageoffset = DRAM_BASE; pageoffset < (DRAM_BASE+DRAM_SIZE); \
    pageoffset += SZ_1M) {
    //DPRINTK(3, "Make DRAM section cacheable: 0x%08lx\n", pageoffset);
    *(mmu_tlb_base + (pageoffset >> 20)) = \
    pageoffset | MMU_SECDESC | MMU_CACHEABLE;
    }
}
```

mmu_init()函数用于启动MMU，它直接调用arm920_setup()函数。arm920_setup()的代码在arch/S3C2440/mmu.c中：

[main(int argc, char *argv[]) > mmu_init( ) > arm920_setup( )]

```
static inline void arm920_setup(void)
{
    unsigned long ttb = MMU_TABLE_BASE; /* MMU_TABLE_BASE = 0x 0x33dfc000 */
    __asm__(
/* Invalidate caches */
    "mov r0, #0\n"
    "mcr p15, 0, r0, c7, c7, 0\n" /* invalidate I,D caches on v4 */
    "mcr p15, 0, r0, c7, c10, 4\n" /* drain write buffer on v4 */
    "mcr p15, 0, r0, c8, c7, 0\n" /* invalidate I,D TLBs on v4 */
/* Load page table pointer */
    "mov r4, %0\n"
    "mcr p15, 0, r4, c2, c0, 0\n" /* load page table pointer */
/* Write domain id (cp15_r3) */
    "mvn r0, #0\n" /* Domains 0b01 = client, 0b11=Manager*/
    "mcr p15, 0, r0, c3, c0, 0\n"    /* load domain access register，
    write domain 15:0, 数据手册P548(access permissions)*/
/* Set control register v4 */
    "mrc p15, 0, r0, c1, c0, 0\n"    /* get control register v4 */
/*数据手册P545：read control register */
```

```
/* Clear out 'unwanted' bits (then put them in if we need them) */
/* ..VI ..RS B... .CAM */ /*这些位的含义在数据手册P546*/
        "bic r0, r0, #0x3000\n"          /* ..11 .... .... .... */
/*I(bit[12])=0 = Instruction cache disabled*/
/*V[bit[13]](Base location of exception registers)=0 = Low addresses =
0x0000 0000*/
        "bic r0, r0, #0x0300\n"          /* .... ..11 .... .... */
/*R(ROM protection bit[9])=0*/
/*S(System protection bit[8])=0*/
/*由于TTB中AP=0b11(line141)，所以RS位不使用(P579)*/
        "bic r0, r0, #0x0087\n"          /* .... .... 1... .111 */
/*M(bit[0])=0 = MMU disabled*/
/*A(bit[1])=0 =Data address
alignment fault checking disable*/
/*C(bit[2])=0 = Data cache disabled*/
/*B(bit[7])=0= Little-endian operation*/
/* Turn on what we want */
/* Fault checking enabled */
        "orr r0, r0, #0x0002\n"          /* .... .... .... ..1. */
/*A(bit[1])=1 = Data address
alignment fault checking enable*/
        #ifdef CONFIG_CPU_D_CACHE_ON        /*is not set*/
        "orr r0, r0, #0x0004\n"          /* .... .... .... .1.. */
/*C(bit[2])=1 = Data cache enabled*/
        #endif
        #ifdef CONFIG_CPU_I_CACHE_ON  /*is not set*/
        "orr r0, r0, #0x1000\n"          /* ...1 .... .... .... */
/*I(bit[12])=1 = Instruction cache enabled*/
        #endif
/* MMU enabled */
        "orr r0, r0, #0x0001\n"          /* .... .... .... ...1 */
/*M(bit[0])=1 = MMU enabled*/
        "mcr p15, 0, r0, c1, c0, 0\n"    /* write control register */
/*数据手册P545*/
        :                                           /* no outputs */
        : "r" (ttb) );
}
```

至此，内存如下划分：

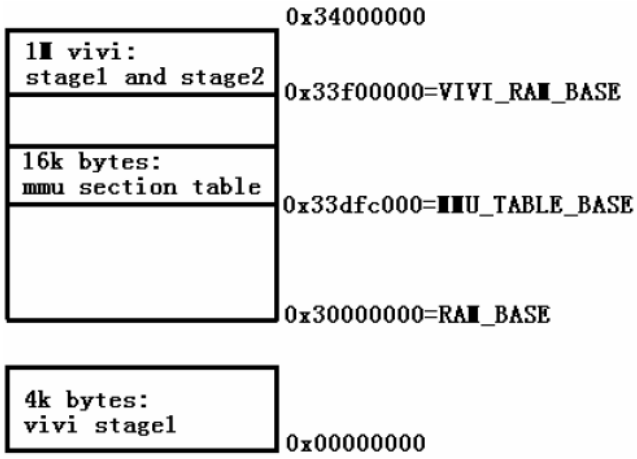


图8 创建页表后内存的划分情况

4)、Step 4：heap_init()

heap——堆，内存动态分配函数mmalloc就是从heap中划出一块空闲内存的，mfree则将动态分配的某块内存释放回heap中。

heap_init函数在SDRAM中指定了一块1M大小的内存作为heap(起始地址HEAP_BASE = 0x33e00000)，并在heap的开头定义了一个数据结构blockhead——事实上，heap就是使用一系列的blockhead数据结构来描述和操作的。每个blockhead数据结构对应着一块heap内存，假设一个blockhead数据结构的存放位置为A，则它对应的可分配内存地址为“A + sizeof(blockhead)”到“A + sizeof(blockhead) + size - 1”。blockhead数据结构在lib/heap.c中定义：

```
typedef struct blockhead_t {
    Int32 signature;                //固定为BLOCKHEAD_SIGNATURE
    Bool allocated;                 //此区域是否已经分配出去：0-N，1-Y
    unsigned long size;             //此区域大小
    struct blockhead_t *next;   //链表指针
    struct blockhead_t *prev;   //链表指针
} blockhead;
```

现在来看看heap是如何运作的(如果您不关心heap实现的细节，这段可以跳过)。vivi 对heap的操作比较简单，vivi中有一个全局变量static blockhead *gHeapBase，它是heap的链表头指针，通过它可以遍历所有blockhead数据结构。假设需要动态申请一块sizeA大小的内存，则mmalloc函数从gHeapBase开始搜索blockhead数据结构，如果发现某个blockhead满足：

a. allocated = 0 //表示未分配

b. size > sizeA，

则找到了合适的blockhead，于是进行如下操作：

a．allocated设为1

b．如果size – sizeA > sizeof(blockhead)，则将剩下的内存组织成一个新的blockhead，放入链表中

c．返回分配的内存的首地址

释放内存的操作更简单，直接将要释放的内存对应的blockhead数据结构的allocated设为0即可。

下面用图来简单演示先分配1K内存，再分配2K内存的过程：

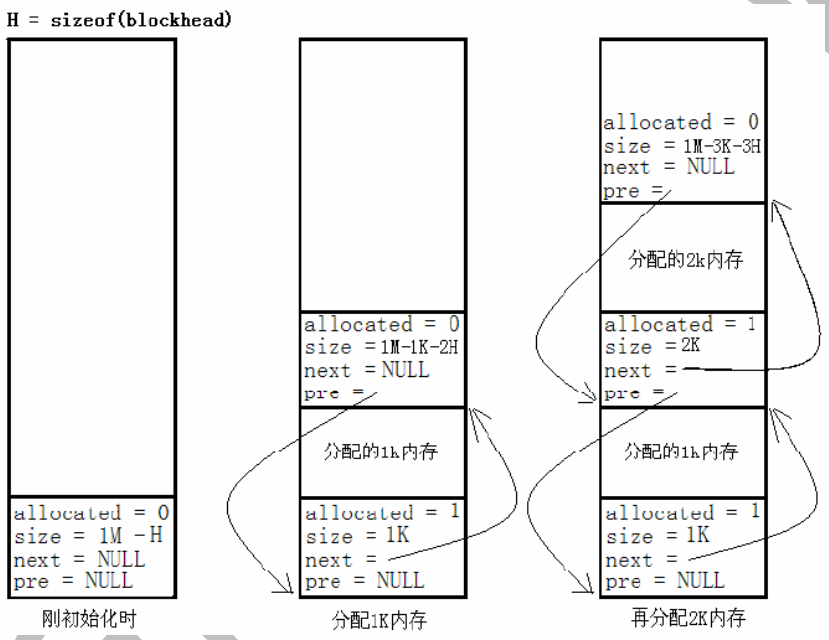


图9 在heap中连续分配1K和2K内存的示意图

heap_init函数直接调用mmalloc_init函数进行初始化，此函数代码在lib/heap.c中比较简单，初始化gHeapBase即可：

[main(int argc, char *argv[]) > heap_init(void) > mmalloc_init(unsigned char *heap, unsigned long size)]

```
static inline int mmalloc_init(unsigned char *heap, unsigned long size)
{
    if (gHeapBase != NULL) return -1;
    DPRINTK("malloc_init(): initialize heap area at 0x%08lx, size = 0x%08lx\n", heap, size);
    gHeapBase = (blockhead *)(heap);
```

```
    gHeapBase->allocated=FALSE;
    gHeapBase->signature=BLOCKHEAD_SIGNATURE;
    gHeapBase->next=NULL;
    gHeapBase->prev=NULL;
    gHeapBase->size = size - sizeof(blockhead);
    return 0;
}
```

分配heap区域后，内存划分情况如下：

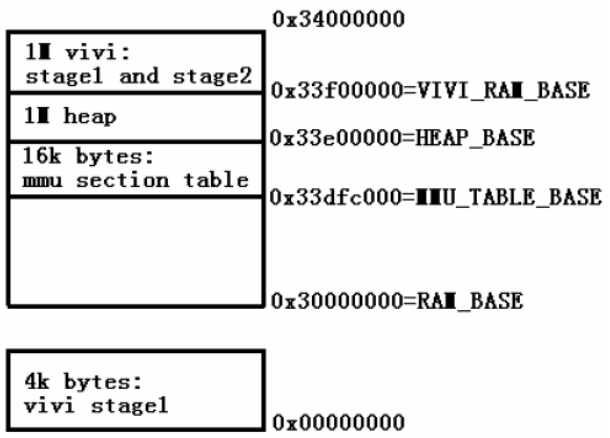


图10 执行heap_init后内存的划分情况

5)、Step 5：mtd_dev_init()

在分析代码前先介绍一下MTD(Memory Technology Device)相关的技术。在linux系统中，我们通常会用到不同的存储设备，特别是FLASH设备。为了在使用新的存储设备时，我们能更简便地提供它的驱动程序，在上层应用和硬件驱动的中间，抽象出MTD设备层。驱动层不必关心存储的数据格式如何，比如是FAT32、ETX2还是FFS2或其它。它仅仅提供一些简单的接口，比如读写、擦除及查询。如何组织数据，则是上层应用的事情。MTD层将驱动层提供的函数封装起来，向上层提供统一的接口。这样，上层即可专注于文件系统的实现，而不必关心存储设备的具体操作。

在我们即将看到的代码中，使用mtd_info数据结构表示一个MTD设备，使用nand_chip数据结构表示一个nand flash芯片。在mtd_info结构中，对nand_flash结构作了封装，向上层提供统一的接口。比如，它根据nand_flash提供的read_data(读一个字节)、read_addr(发送要读的扇区的地址)等函数，构造了一个通用的读函数read，将此函数的指针作为自己的一个成员。而上层要读写flash时，执行mtd_info中的read、write函数即可。

下面分析代码：

mtd_dev_init()用来扫描所使用的NAND Flash的型号，构造MTD设备，即构造一个

mtd_info的数据结构。对于本开发板，它直接调用mtd_init()，mtd_init又调用smc_init()，此函数在drivers/mtd/maps/S3C2440_flash.c中：

[main(int argc, char *argv[]) > mtd_dev_init() > mtd_init() > smc_init()]

```
static int
smc_init(void)
{
/*struct mtd_info *mymtd，数据类型在include/mtd/mtd.h*/
/*strcut nand_chip在include/mtd/nand.h中定义*/
    struct nand_chip *this;
    u_int16_t nfconf;
/* Allocate memory for MTD device structure and private data */
    mymtd = mmalloc(sizeof(struct mtd_info) + sizeof(struct nand_chip));
    if (!mymtd) {
        printk("Unable to allocate S3C2440 NAND MTD device structure.\n");
        return -ENOMEM;
    }
/* Get pointer to private data */
    this = (struct nand_chip *)(&mymtd[1]);
/* Initialize structures */
    memset((char *)mymtd, 0, sizeof(struct mtd_info));
    memset((char *)this, 0, sizeof(struct nand_chip));
/* Link the private data with the MTD structure */
    mymtd->priv = this; /* set NAND Flash controller */
    nfconf = NFCONF;
/* NAND Flash controller enable */
    nfconf |= NFCONF_FCTRL_EN;
/* Set flash memory timing */
    nfconf &= ~NFCONF_TWRPH1; /* 0x0 */
    nfconf |= NFCONF_TWRPH0_3; /* 0x3 */
    nfconf &= ~NFCONF_TACLS; /* 0x0 */
    NFCONF = nfconf;
/* Set address of NAND IO lines */
    this->hwcontrol = smc_hwcontrol;
    this->write_cmd = write_cmd;
    this->write_addr = write_addr;
    this->read_data = read_data;
    this->write_data = write_data;
    this->wait_for_ready = wait_for_ready;
/* Chip Enable -> RESET -> Wait for Ready -> Chip Disable */
    this->hwcontrol(NAND_CTL_SETNCE);
    this->write_cmd(NAND_CMD_RESET);
    this->wait_for_ready();
```

```
    this->hwcontrol(NAND_CTL_CLRNCE);
    smc_insert(this);
    return 0;
}
```

6)、－14行构造了一个mtd_info结构和nand_flash结构，前者对应MTD设备，后者对应nand flash芯片(如果您用的是其他类型的存储器件，比如nor flash，这里的nand_flash结构应该换为其他类型的数据结构)。MTD设备是具体存储器件的抽象，那么在这些代码中这种关系如何体现呢——第14行的代码把两者连结在一起了。事实上，mtd_info结构中各成员的实现(比如read、write函数)，正是由priv变量所指向的nand_flash的各类操作函数(比如read_addr、read_data等)来实现的。

15－20行是初始化S3C2440上的NAND FLASH控制器，和我们在实验六“NAND FLASH CONTROLLER”里面做的一样。

前面分配的nand_flash结构还是空的，现在当然就是填满它的各类成员了，这正是21-26行做的事情。27-30行对这块nand flash作了一下复位操作。

最后，也是最复杂的部分，根据刚才填充的nand_flash结构，构造mtd_info结构，这由31行的smc_insert函数调用smc_scan完成。先看看smc_insert函数：

[main(int argc, char *argv[]) > mtd_dev_init() > mtd_init() > smc_init() > smc_insert(struct nand_chip *this)]

```
inline int
smc_insert(struct nand_chip *this) {
/* Scan to find existance of the device */
/*smc_scan defined at drivers/mtd/nand/smc_core.c*/
    if (smc_scan(mymtd)) {
        return -ENXIO;
    }
/* Allocate memory for internal data buffer */
    this->data_buf = mmalloc(sizeof(u_char) *  (mymtd->oobblock + mymtd->oobsize));
    if (!this->data_buf) {
        printk("Unable to allocate NAND data buffer for S3C2440.\n");
        this->data_buf = NULL;
        return -ENOMEM;
    }
    return 0;
}
```

后面的7－13行，我也没弄清楚，待查。

现在，终于到重中之重了，请看smc_scan函数的代码：

[main(int argc, char *argv[]) > mtd_dev_init() > mtd_init() > smc_init() > smc_insert(struct nand_chip *this) > smc_scan(struct mtd_info *mtd)]

```
int smc_scan(struct mtd_info *mtd)
{
    int i, nand_maf_id, nand_dev_id;
    struct nand_chip *this = mtd->priv;
/* Select the device */
    nand_select();
/* Send the command for reading device ID */
    nand_command(mtd, NAND_CMD_READID, 0x00, -1);
    this->wait_for_ready();
/* Read manufacturer and device IDs */
    nand_maf_id = this->read_data();
    nand_dev_id = this->read_data();
/* Print and sotre flash device information */
    for (i = 0; nand_flash_ids[i].name != NULL; i++) {
        if (nand_maf_id == nand_flash_ids[i].manufacture_id &&
        nand_dev_id == nand_flash_ids[i].model_id) {
        if (!(mtd->size) && !(nand_flash_ids[i].page256)) {
            mtd->name = nand_flash_ids[i].name;
            mtd->erasesize = nand_flash_ids[i].erasesize;
            mtd->size = (1 << nand_flash_ids[i].chipshift);
            mtd->eccsize = 256;
            mtd->oobblock = 512;
            mtd->oobsize = 16;
            this->page_shift = 9;
            this->dev = &nand_smc_info[GET_DI_NUM(nand_flash_ids[i].chipshift)];
        }
      printk("NAND device: Manufacture ID:" \
      " 0x%02x, Chip ID: 0x%02x (%s)\n", nand_maf_id, nand_dev_id, mtd->name);
      break;
      }
    }
/* De-select the device */
    nand_deselect();
/* Print warning message for no device */
    if (!mtd->size) {
        printk("No NAND device found!!!\n");
        return 1;
    }
/* Fill in remaining MTD driver data */
    mtd->type = MTD_NANDFLASH;
    mtd->flags = MTD_CAP_NANDFLASH | MTD_ECC;
```

```
    mtd->module = NULL;
    mtd->ecctype = MTD_ECC_SW;
    mtd->erase = nand_erase;
    mtd->point = NULL;
    mtd->unpoint = NULL;
    mtd->read = nand_read;
    mtd->write = nand_write;
    mtd->read_ecc = nand_read_ecc;
    mtd->write_ecc = nand_write_ecc;
    mtd->read_oob = nand_read_oob;
    mtd->write_oob = nand_write_oob;
    mtd->lock = NULL;
    mtd->unlock = NULL;
/* Return happy */
    return 0;
}
```

5-9行，读取nand flash得厂家ID和设备ID，下面将用这两个ID号，在一个预设的数组“nand_flash_ids”里查找到与之相符的项。nand_flash_ids是nand_flash_dev结构的数组，里面存放的是世界上比较常用的nand flash型号的一些特性(见下面的定义)：

```
struct nand_flash_dev {
char * name;
int manufacture_id;
int model_id;
int chipshift;
char page256;
char pageadrlen;
unsigned long erasesize;
};
static struct nand_flash_dev nand_flash_ids[] = {
{"Toshiba TC5816BDC", NAND_MFR_TOSHIBA, 0x64, 21, 1, 2, 0x1000},
……
{NULL,}
};
```

10－28行根据找到的数组项，构造前面分配的mtd_info结构。

35行之后的代码，填充mtd_info结构的剩余部分，大多是一些函数指针，比如read、write函数等。

执行完mtd_dev_init后，我们得到了一个mtd_info结构的全局变量(mymtd指向它)，以后对nand flash的操作，直接通过mymtd提供的接口进行。

6)、Step 6：init_priv_data()

此函数将启动内核的命令参数取出，存放在内存特定的位置中。这些参数来源有两个：vivi预设的默认参数，用户设置的参数(存放在nand flash上)。init_priv_data先读出默认参数，存放在“VIVI_PRIV_RAM_BASE”开始的内存上；然后读取用户参数，若成功则用用户参数覆盖默认参数，否则使用默认参数。

init_priv_data函数分别调用get_default_priv_data函数和load_saved_priv_data函数来读取默认参数和用户参数。这些参数分为3类：

a．vivi自身使用的一些参数，比如传输文件时的使用的协议等

b．linux启动命令

c．nand flash的分区参数

get_default_priv_data函数比较简单，它将vivi中存储这些默认参数的变量，复制到指定内存中。执行完后，此函数执行完毕后，内存使用情况如下：

图11 执行init_priv_data后内存的划分情况

load_saved_priv_data函数读取上述内存图中分区参数的数据，找到“param”分区，然后从中读出上述3类参数，覆盖掉默认参数(如果能成功读出的话)。

7)、Step 7：misc()和init_builtin_cmds()

这两个函数都是简单地调用add_command函数，给一些命令增加相应的处理函数。在vivi启动后，可以进去操作界面，这些命令，就是供用户使用的。它们增加了如下命令：

```
a. add_command(&cpu_cmd)
b. add_command(&bon_cmd)
c. add_command(&reset_cmd)
d. add_command(&param_cmd)
```

```
e. add_command(&part_cmd)
f. add_command(&mem_cmd)
g. add_command(&load_cmd)
h. add_command(&go_cmd)
i. add_command(&dump_cmd)
j. add_command(&call_cmd)
k. add_command(&boot_cmd)
l. add_command(&help_cmd)
```

现在来看看add_command函数是如何实现得：

```
void add_command(user_command_t *cmd)
{
	if (head_cmd == NULL) {
		head_cmd = tail_cmd = cmd;
	} else {
		tail_cmd->next_cmd = cmd;
		tail_cmd = cmd;
	}
/*printk("Registered '%s' command\n", cmd->name);*/
}
```

很简单！把user_command_t结构放入一个链表即可。user_command_t结构定义如下，呵呵，name是命令名称，cmdfunc是这个命令的处理函数，helstr是帮助信息。

```
typedef struct user_command {
	const char *name;
	void (*cmdfunc)(int argc, const char **);
	struct user_command *next_cmd;
	const char *helpstr;
} user_command_t;
```

8)、Step 8：boot_or_vivi()

此函数根据情况，或者启动“vivi_shell”，进入与用户进行交互的界面，或者直接启动linux内核。代码如下：

[main(int argc, char *argv[]) > void boot_or_vivi(void)]

```
void boot_or_vivi(void)
{
	char c;
	int ret;
	ulong boot_delay;
```

```
    boot_delay = get_param_value("boot_delay", &ret);
    if (ret) boot_delay = DEFAULT_BOOT_DELAY;
/* If a value of boot_delay is zero, unconditionally call vivi shell*/
    if (boot_delay == 0) vivi_shell();
/*wait for a keystroke (or a button press if you want.) */
    printk("Press Return to start the LINUX now, any other key for vivi\n");
    c = awaitkey(boot_delay, NULL);
    if (((c != '\r') && (c != '\n') && (c != '\0'))) {
        printk("type \"help\" for help.\n");
        vivi_shell();
    }
    run_autoboot();
    return;
}
```

第10行等待键盘输入，如果在一段时间内键盘无输入，或者输入了回车键，则调用run_autoboot启动内核；否则调用vivi_shell进入交互界面。

vivi_shell等待用户输入命令(等待串口数据)，然后根据命令查找“Step 7：misc()和init_builtin_cmds()”设置的命令链表，运行找到的命令函数。vivi_shell函数是通过调用serial_term函数来实现的，serial_term代码如下：

[main(int argc, char *argv[]) > void boot_or_vivi(void) > vivi_shell(void) > serial_term]

```
void serial_term(void)
{
    char cmd_buf[MAX_CMDBUF_SIZE];
    for (;;) {
        printk("%s> ", prompt); /*prompt is defined upside*/
        getcmd(cmd_buf, MAX_CMDBUF_SIZE);
/* execute a user command */
    if (cmd_buf[0])
    exec_string(cmd_buf);
    }
}
```

第6行getcmd函数读取串口数据，将返回的字符串作为参数调用exec_string函数 。exec_string代码如下：

[main(int argc, char *argv[]) > void boot_or_vivi(void) > vivi_shell(void) > serial_term > void exec_string(char *buf)]

```
void exec_string(char *buf)
{
```

```
    int argc;
    char *argv[128];
    char *resid;
    while (*buf) {
        memset(argv, 0, sizeof(argv));
        parseargs(buf, &argc, argv, &resid);
        if (argc > 0)
        execcmd(argc, (const char **)argv);
        buf = resid;
    }
}
```

它首先调用parseargs函数分析所传入的字符串，确定参数个数及将这些参数分别保存。比如对于字符串“abcd efgh ijklm 123”，parseargs函数返回的结果是：argc = 4，argv[0]指向“abcd”字符串，argv[1]指向“efgh”字符串，argv[2]指向“ijklm”字符串，argv[3]指向“123”字符串。然后，调用execmd函数：

[main(int argc, char *argv[]) > void boot_or_vivi(void) > vivi_shell(void) > serial_term > void execcmd(int argc, const char **argv)]

```
 void execcmd(int argc, const char **argv)
{
    user_command_t *cmd = find_cmd(argv[0]);
    if (cmd == NULL) {
        printk("Could not found '%s' command\n", argv[0]);
        printk("If you want to konw available commands, type 'help'\n");
        return;
    }
/*printk("execcmd: cmd=%s, argc=%d\n", argv[0], argc);*/
    cmd->cmdfunc(argc, argv);
}
```

首先，第3行的find_cmd函数根据命令(argv[0])找到对应的处理函数，然后执行它(第10行)。命令名字和对应的处理函数指针，在上述“Step 7：misc()和init_builtin_cmds()”中，使用一个命令链表存起来了。具体的命令处理函数，这里就不细说了，各位有兴趣的可以自行阅读源代码。

run_autoboot用于启动内核，起始就是运行“boot”命令的处理函数command_boot。我们看看它是如何启动内核的，下面我们只列出用到的代码：

[main(int argc, char *argv[]) > void boot_or_vivi(void) > void run_autoboot(void) > exec_string("boot") > command_boot(1, 0)]

```
void command_boot(int argc, const char **argv)
```

```
{
……
    media_type = get_param_value("media_type", &ret);
    if (ret) {
        printk("Can't get default 'media_type'\n");
    return;
    }
    kernel_part = get_mtd_partition("kernel");
    if (kernel_part == NULL) {
        printk("Can't find default 'kernel' partition\n");
        return;
    }
    from = kernel_part->offset;
    size = kernel_part->size;
……
    boot_kernel(from, size, media_type);
……
}
```

第3行，获得存储内核代码的器件类型；第8行获取“kernel”分区信息，第13、14行由此分区信息获得内核存放的位置及大小；第15行的boot_kernel函数将内核从其存储的器件上(flash)复制到内存，然后执行它。boot_kernel代码如下：

[main(int argc, char *argv[]) > void boot_or_vivi(void) > void run_autoboot(void) > exec_string("boot") > command_boot(1, 0) > int boot_kernel(ulong from, size_t size, int media_type)]

```
int boot_kernel(ulong from, size_t size, int media_type)
{
    int ret;
    ulong boot_mem_base; /* base address of bootable memory */
    ulong to;
    ulong mach_type;
    boot_mem_base = get_param_value("boot_mem_base", &ret);
    if (ret) {
        printk("Can't get base address of bootable memory\n");
        printk("Get default DRAM address. (0x%08lx\n", DRAM_BASE);
        boot_mem_base = DRAM_BASE;
    }
/* copy kerne image LINUX_KERNEL_OFFSET = 0x8000 */
    to = boot_mem_base + LINUX_KERNEL_OFFSET;
    printk("Copy linux kernel from 0x%08lx to 0x%08lx, size = 0x%08lx ... ", from, to, size);
    ret = copy_kernel_img(to, (char *)from, size, media_type);
```

```
    if (ret) {
        printk("failed\n");
        return -1;
    } else {
        printk("done\n");
    }
    if (*(ulong *)(to + 9*4) != LINUX_ZIMAGE_MAGIC) {
        printk("Warning: this binary is not compressed linux kernel image\n");
        printk("zImage magic = 0x%08lx\n", *(ulong *)(to + 9*4));
    } else {
        printk("zImage magic = 0x%08lx\n", *(ulong *)(to + 9*4));
    }
/* Setup linux parameters and linux command line LINUX_PARAM_OFFSET = 0x100 */
    setup_linux_param(boot_mem_base + LINUX_PARAM_OFFSET);
/* Get machine type */
    mach_type = get_param_value("mach_type", &ret);
    printk("MACH_TYPE = %d\n", mach_type);
/* Go Go Go */
    printk("NOW, Booting Linux.....\n");
    call_linux(0, mach_type, to);
    return 0;
}
```

第7－13行用来确定在内核将存放在内存中的位置；15行调用nand_read_ll函数(详细介绍请看“实验六：NAND FLASH CONTROLLER”)从flash芯片上将内核读出；28行设置内核启动的参数，比如命令行等；29行获取处理器型号；32行的函数call_linux汇编代码，跳到内核存放的地址处执行。值得细看的有两个函数：setup_linux_param和call_linux。

setup_linux_param函数用于设置linux启动时用到的各类参数：

[main( ) > boot_or_vivi( ) > run_autoboot( ) > exec_string("boot") > command_boot(1, 0) > boot_kernel( ) > setup_linux_param( )]

```
static void setup_linux_param(ulong param_base)
{
    struct param_struct *params = (struct param_struct *)param_base;
    char *linux_cmd;
    printk("Setup linux parameters at 0x%08lx\n", param_base);
    memset(params, 0, sizeof(struct param_struct));
    params->u1.s.page_size = LINUX_PAGE_SIZE;
    params->u1.s.nr_pages = (DRAM_SIZE >> LINUX_PAGE_SHIFT);
/* set linux command line */
    linux_cmd = get_linux_cmd_line(); /* lib/priv_data/param.c */
```

```
    if (linux_cmd == NULL) {
        printk("Wrong magic: could not found linux command line\n");
    } else {
        memcpy(params->commandline, linux_cmd, strlen(linux_cmd) + 1);
        printk("linux command line is: \"%s\"\n", linux_cmd);
    }
}
```

此函数在param_base地址存放linux启动时用到的参数。对于本开发板，此地址为RAM_BASE + LINUX_PARAM_OFFSET = 0x3000_0000 + 0x100。第9行将命令行参数从原来位置(上述Step 6：init_priv_data中将命令行等存储在内存中)读出，复制到相应位置(第13行)。

call_linux函数先对cache和tlb进行一些设置，然后将内核存放的位置直接赋给pc寄存器，从而执行内核。call_linux函数代码如下：

[main( ) > boot_or_vivi( ) > run_autoboot( ) > exec_string("boot") > command_boot(1, 0) > boot_kernel( ) > call_linux( )]

```
void call_linux(long a0, long a1, long a2)
{
    cache_clean_invalidate();
    tlb_invalidate();
    __asm__(
        "mov r0, %0\n"
        "mov r1, %1\n"
        "mov r2, %2\n"
        "mov ip, #0\n"
        "mcr p15, 0, ip, c13, c0, 0\n"    /* zero PID */
        "mcr p15, 0, ip, c7, c7, 0\n"     /* invalidate I,D caches */
        "mcr p15, 0, ip, c7, c10, 4\n"    /* drain write buffer */
        "mcr p15, 0, ip, c8, c7, 0\n"     /* invalidate I,D TLBs */
        "mrc p15, 0, ip, c1, c0, 0\n"     /* get control register */
        "bic ip, ip, #0x0001\n" /* disable MMU */
        "mcr p15, 0, ip, c1, c0, 0\n"     /* write control register */
        "mov pc, r2\n"
        "nop\n"
        "nop\n"
        :                                 /* no outpus */
        : "r" (a0), "r" (a1), "r" (a2)
    );
}
```

对cache、tlb等的操作，您可以参考“实验十一：MMU”。6、7、8行中的%0、%1、%2

对应3个寄存器，它们存放的值分别为参数a0、a1和a2。这样，执行完这3条语句后，r0、r1、r2对应的值分别为：0, mach_type, 0x3000_8000(内核在内存中的存放地址)。第17行将跳到去执行内核。至此，linux内核终于开始运行了！

vivi执行完毕后，内存使用情况如下：

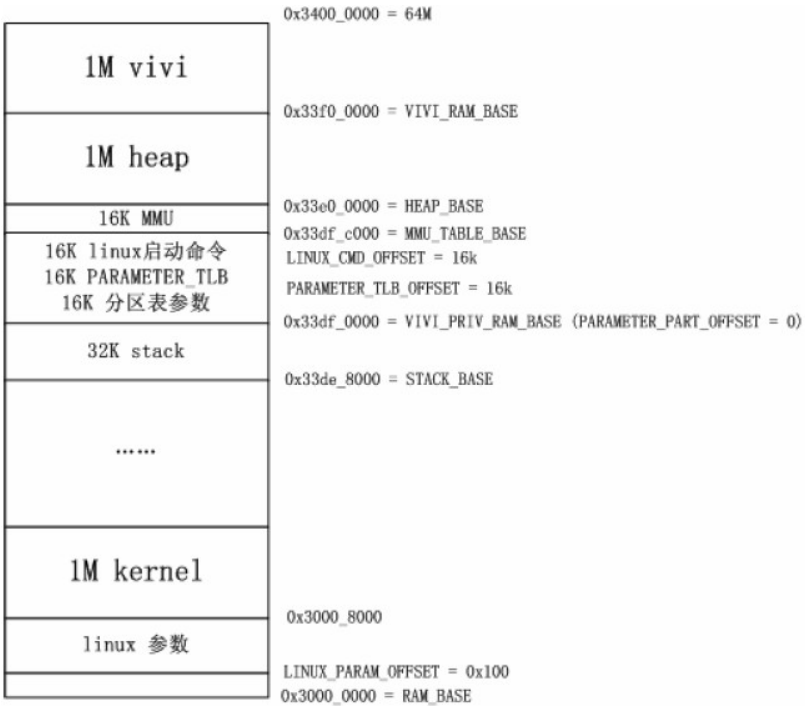


图 12 vivi 启动内核后内存的划分情况

五． 附录一VI命令解释

1、 help命令

给目标板(开发板)加电/重启，即可进入vivi命令界面

```
vivi> help
Usage:
   cpu [{cmds}]                    -- Manage cpu clocks
   bon [{cmds}]                    -- Manage the bon file system
   reset                           -- Reset the system
   param [set|show|save|reset]             -- set/get parameter
   part [add|del|show|reset]               -- Manage MTD partitions
   mem [{cmds}]                    -- Manage Memory
   loadyaffs {...}                 -- Load a yaffs image to Flash
   eboot                           -- Run Wince Ethernet Bootloader(eboot)
   wince                           -- Run Wince
   load {...}                      -- Load a file to RAM/Flash
   go <addr> <a0> <a1> <a2> <a3>           -- jump to <addr>
   dump <addr> <length>                    -- Display (hex dump) a range of memory.
   call <addr> <a0> <a1> <a2> <a3>         -- jump_with_return to <addr>
   boot [{cmds}]                   -- Booting linux kernel
   help [{cmds}]                   -- Help about help?
```

2、 mem命令

mem系列命令用于对系统的内存进行操作

```
vivi> mem
invalid 'mem' command: wrong argumets
Usage:
   compare <dst> <src> <length> -- compare
   mem copy <dst> <src> <length>
   mem info
   mem reset -- reset memory control register
   mem serach <start_addr> <end_addr> <value> -- serach memory address that contain value
vivi>mem info
RAM Information:
Default ram size     : 64M
Real ram size        : 64M
Free memory          : 63M
RAM mapped to            : 0x30000000 - 0x34000000          (SDRAM映射的地址范围- 64M)
```

```
Flash memory mapped to      : 0x10000000 - 0x12000000      (Flash映射的地址范围 32M)
Available memory region     : 0x30000000 - 0x33f80000   (用户可以使用的有效的内存区域地址范围,约为63.5M)
Stack base address          : 0x33fafffc                (栈的基地址)
Current stack pointer       : 0x33fafc7c                (当前栈指针的值)
Memory control register vlaues                          (S3C2440的内存控制寄存器的当前值)
  BWSCON   = 0x2211d120
  BANKCON0     = 0x00000700
  BANKCON1     = 0x00000700
  BANKCON2     = 0x00000700
  BANKCON3     = 0x00000700
  BANKCON4     = 0x00000700
  BANKCON5     = 0x00000700
  BANKCON6     = 0x00018005
  BANKCON7     = 0x00018005
  REFRESH  = 0x008e0459
  BANKSIZE     = 0x000000b2
  MRSRB6       = 0x00000030
  MRSRB7       = 0x00000030
```

3、 load命令

load命令下载程序到存储器中(Flash或者RAM中)

```
vivi> load help
Usage:
   load <flash|ram> [ <partname> | <addr> <size> ] <x|y|z|t>
   <flash|ram>
```

关键字参数flash和ram用于选择目标介质是Flash还是RAM。

如果选择下载到Flash中，其实还是先要下载到RAM中(临时下载到SDRAM的起始地址处0x30000000保存一下，然后再转写入 FLASH)，然后再通过Flash驱动程序提供的写操作，将数据写入到Flash中。如果选择了flash参数，那么到底是将"数据"写入NOR Flash还是NAND Flash，取决于boot loader编译的过程中，所进行的配置，这就要看配置的时候将MTD设备配置成NOR Flash还是NAND Flash。

[<partname> | <addr> <size>]

partname是vivi的MTD分区表中的分区名,MTD分区的起始地址；addr和size是让用户自己选择下载的目标存储区域，而不是使用vivi的MTD分区。addr表示下载的目标地址，size表示下载的文件大小，单位字节，size参数不一定非要指定得和待下载的文件大小一样大，但是一定要大于等于待下载的文件的字节数。

<x|y|z|t>

关键字参数x y和z分别表示从 PC主机上下载文件到ARM9系统中，采用哪种串行文件传送协议，x表示采用xmodem协议，y表示采用ymodem协议，z表示采用zmodem协议

请注意目前该bootloader -- vivi还没有实现zmodem协议，所以该参数只能选择x和y

关键字参数t应该是开发板vivi增加的，是tftp下载！很好用的！速度比jtag要快多了！

开发板的vivi eboot烧写都要通过load命令

如：要烧写eboot.nb0到flash的eboot分区

```
vivi> load    flash   eboot    t
```

使用交叉网线连好PC与开发板，把eboot.nb0拷贝到于mtftp.exe同一目录下，在windows命令行输入

```
mytftp    -i     192.168.0.15    PUT    eboot.nb0
```

等待烧写完成即可

4、 param命令

param 命令用于对bootloader的参数进行操作

```
vivi> param help
Usage:
param help                      -- Help aout 'param' command
param reset                     -- Reset parameter table to default table
param save                      -- Save parameter table to flash memeory
param set <name> <value>                -- Reset value of parameter
param set linux_cmd_line "..."          -- set boot parameter
param set wince_part_name "..."         -- set the name of partition wich wincewill be stored in
param show                      -- Display parameter table
vivi> param show
Number of parameters: 19
name            :       hex     integer
```

//(1)类型，168表示 S3C2440的开发系统

//(2)媒介类型，即指示了bootloader从哪个媒介启动起来的

//(3)引导 linux 内核启动的基地址映像将被从 Flash 中拷贝到boot_mem_base + 0x8000 的地址处，内核参数将被建立在boot_mem_base+0x100的地址处

//(4)bootloader启动时，默认设置的串口波特率

```
mach_type       :   000000c1    193     //(1)
media_type      :   00000003    3       //(2)
boot_mem_base   :   30000000  805306368         //(3)
baudrate        :   0001c200    115200      //(4)
```

//(5) 以下三个参数和xmodem文件传送协议相关:

xmodem_one_nak表示接收端(即ARM9系统这端)发起第一个NAK信号给发送端(即PC主机这端)到启动；

xmodem_initial_timeout表示接收端(即ARM9系统这端)启动xmodem协议后的初始超时时间，第一次接收超时按照这个参数的值来设置，但是超时一次后，后面的超时时间就不再是这个参数的值了，而是xmodem_timeout的值；

xmodem_timeout表示在接收端(即ARM9系统这端)等待接受发送端(即PC主机这端)送来的数据字节过程中，如果发生了一次超时，那么后面的超时时间就设置成参数xmodem_timeout的值了这三个参数不需要修改，系统默认的值就可以了，不建议用户去修改这几个参数值

```
xmodem_one_nak      :   00000000        0
xmodem_initial_timeout  :   000493e0        300000
xmodem_timeout      :   000f4240        1000000
```

//(6) ymodem_initial_timeout

表示接收端(即 ARM9 系统这端)在启动了ymodem协议后的初始超时时间，这个参数不需要修改，系统默认的值就可以了，不建议用户去修改这几个参数值

//(7) boot_delay是bootloader自动引导linux kernel功能的延时时间

```
ymodem_initial_timeout      :   0016e360        1500000     //(6)
boot_delay          :   00300000        3145728     //(7)
os                  :   WINCE
display             :   VGA 640X480
ip                  :   192.168.0.15
host                :   192.168.0.1
gw                  :   192.168.0.1
mask                :   255.255.255.0
wincesource         :   00000001        1
wincedeploy         :   00000000        0
mac                 :   00:00:c0:ff:ee:08
wince part name     :   wince
Linux command line          :   noinitrd root=/dev/mtdblock/3 init=/linuxrc console=ttyS0
```

//(8) Linux command line不是bootloader的参数，而是kernel启动的时候，kernel不能自动检测到的必要的参数些参数需要bootloader传递给linux kernel, Linux command line就是设置linux kernel启动时，需要手工

传给 kernel的参数

```
    param reset              将bootloader 参数值复位成系统默认值。
    param set paramname value      设置参数值
    param save                    保存参数设置
    param set linux_cmd_line ”linux bootparam”
                                设置linux 启动参数，参数linux bootparam表示要设置的linux kernel命令行参数。
```

修改boot_delay(延长等待时间)

```
vivi> param set boot_delay 0x05000000
vivi> param save
```

5、 part命令

part命令用于对MTD分区进行操作

```
vivi> part show
mtdpart info. (5 partitions)
name                offset          size            flag
-------------------------------------------------------------------------------------------------------------------
-
vivi            : 0x00000000  0x00020000   0   128k
eboot           : 0x00020000  0x00020000   0   128k
param           : 0x00040000  0x00010000   0   64k
kernel          : 0x00050000  0x00100000   0   1M
root            : 0x00150000  0x03eac000   0    62M+688k
```

MTD分区是针对Flash(NOR Flash或者NAND Flash)的分区，以便于对bootloader对Flash进行管理

part add　　命令用于添加一个MTD分区

```
    part add    name    offset    size    flag
```

参数 name是要添加的分区的分区名

参数 offset是要添加的分区的偏移(相对于整个MTD设备的起始地址的偏移，在 ARM9系统中不论配置的是NOR Flash，还是NAND Flash，都只注册了一个mtd_info结构，也就是说逻辑上只有一个MTD设备，这个MTD设备的起始地址为0x00000000)；

参数 size是要添加的分区的大小，单位为字节；

参数 flag是要添加的分区的标志，参数flag的取值只能为以下字符串(请注意必须为大写)或者通过连接符|

这个标志表示了这个分区的用途

```
“BONFS”    作为BONFS文件系统的分区；
“JFFS2”    作为JFFS2文件系统的分区；
“LOCK”     该分区被锁定了；
“RAM”      该分区作为RAM使用
```

例如，添加新的 MTD分区mypart

```
vivi> part add mypart 0x500000 0x100000 JFFS2

mypart: offset = 0x00500000, size = 0x00100000, flag = 8
```

```
part del        命令用于删除一个 MTD分区
part del        name
```

参数name是要删除的MTD分区的分区名

```
part save       保存part分区信息
part reset      恢复为系统默认part分区
```

6、 boot命令

boot命令用于引导linux kernel启动

```
vivi>boot help
Usage:
    boot <media_type> -- booting kernel
        value of media_type (location of kernel image)
        1 = RAM
        2 = NOR Flash Memory
        3 = SMC (On S3C2410)
    boot <media_type> <mtd_part> -- boot from specific mtd partition
    boot <media_type> <addr> <size>
    boot help -- help about 'boot' command
```

<media_type>

boot关键字后面media_type必须指定媒介类型，因为boot命令对不同媒介的处理方式是不同的，例如如果kernel在 SDRAM中，那么boot执行的过程中就可以跳过拷贝kernel映像到

SDRAM中这一步骤了

Boot命令识别的媒介类型有以下三种：

ram 表示从RAM(在ARM9系统中即为SDRAM)中启动linux kernel，linux kernel必须要放在RAM中

nor 表示从NOR Flash中启动linux kernel，linux kernel必须已经被烧写到了NOR Flash中

smc 表示从NAND Flash中启动linux kernel，linux kernel必须已经被烧写到了NAND Flash中

<mtd_part>

参数mtd_part是MTD分区的名，MTD设备的一个分区中启动linux kernel，kernel映像必须被放到这个分区中；

<addr> <size> 分别表示linux kernel起始地址和kernel的大小。为什么要指定kernel大小呢?因为kernel首先要被copy到boot_mem_base + 0x8000的地方，然后在boot_mem_base + 0x100开始的地方设置内核启动参数，要拷贝 kernel，当然需要知道kernel的大小啦，这个大小不一定非要和kernel实际大小一样，但是必须许大于等于kernel的大小，单位字节

7、 bon命令

bon命令用于对bon分区进行操作。通过bon help可以显示系统对bon系列命令的帮助提示 。bon分区是nand flash设备的一种简单的分区管理方式。bootloader支持bon分区，同时Samsung提供的针对S3C2410移植的linux版本中也支持了bon分区，这样就可以利用bon分区来加载linux的根文件系统.

MTD分区和BON分区，当ARM9系统配置了NAND Flash作为MTD设备，那么MTD分区和BON分区都在同一片NAND Flash上

```
vivi> bon part info    命令用于显示系统中bon分区的信息
BON info. (3 partitions)
No: offset      size          flags       bad
0: 0x00000000  0x00030000      00000000      0     192k
1: 0x00030000  0x00100000      00000000      0   1M
2: 0x00130000  0x03ec8000      00000000      1     62M+800k
```

bon 分 区 表 被 保 存 到 nand flash 的 最 后 0x4000 个 字 节 中 ， 即 在 nand flash 的 0x03FFC000~0x33FFFFFF范围内，分区表起始于0x03FFC000。

```
vivi> bon part       命令用于建立系统的bon分区表
vivi> bon part 0 192k 1M
doing partition
size = 0
```

```
size = 196608
size = 1048576
check bad block
part = 0 end = 196608
```

go命令

go命令用于跳转到指定地址处执行该地址处的代码。

go addr跳转到指定地址运行该处程序。

未完成，待续.....