第014课 异常与中断

来自百问网嵌入式Linux wiki

目录

第001节_概念引入与处理流程

取个场景解释中断。

假设有个大房间里面有小房间，婴儿正在睡觉，他的妈妈在外面看书。

问：这个母亲怎么才能知道这个小孩醒？

1. 过一会打开一次房门，看婴儿是否睡醒，让后接着看书
2. 一直等到婴儿发出声音以后再过去查看，期间都在读书

第一种 叫做**查询方式**：

- 优点：简单
- 缺点： 累

写程序如何：

```
while(1)
{
    1 read book(读书)
    2 open door（开门）
    if(睡)
        return(read book)
    else
        照顾小孩

}
```

第二种叫**中断方式**：

- 优点：不累
- 缺点：复杂

写程序：

```
while(1)
{
    read book
    中断服务程序()//如何被调用?
    {
    处理照顾小孩
    }
}
```

我们看看母亲被小孩哭声打断如何照顾小孩？

母亲的处理过程:

1 平时看书

2 发生了各种声音，如何处理这些声音

有远处的猫叫（听而不闻，忽略）
门铃声有快递（开门收快递）
小孩哭声（打开房门，照顾小孩）

3 母亲的处理

只会处理门铃声和小孩哭声
a 现在书中放入书签，合上书(保存现场)
b 去处理 (调用对应的中断服务程序)
c 继续看书(恢复现场)

不同情况，不同处理：

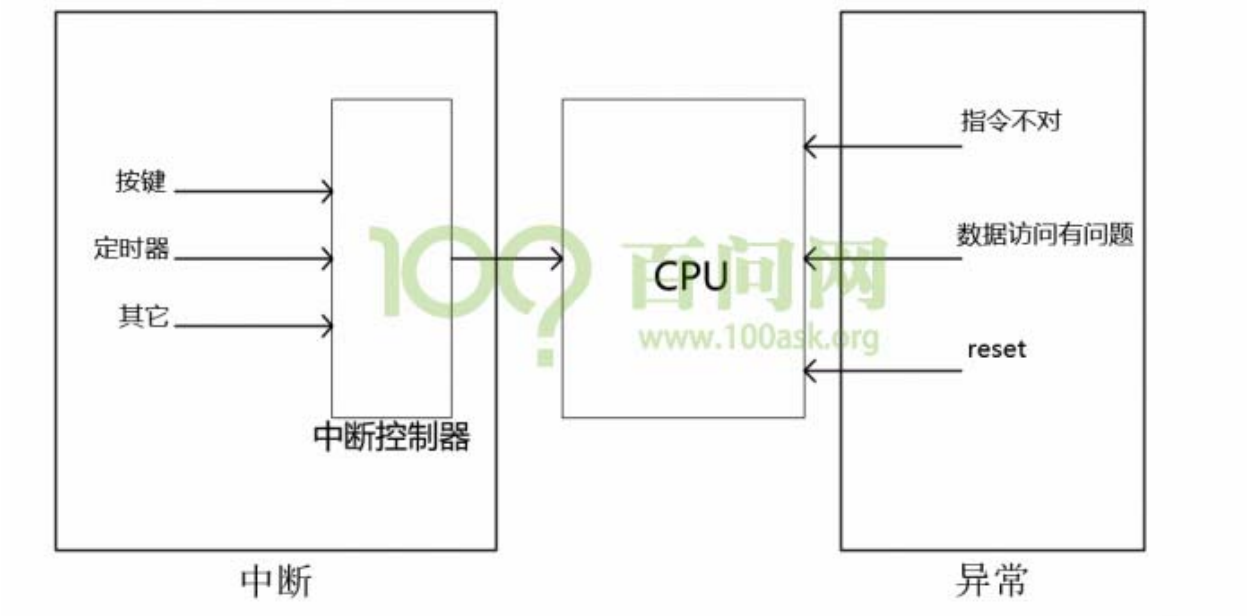
a 对于门铃：开门取快件

b 对于哭声:照顾小孩

我们将母亲的处理过程抽象化——母亲的头脑相当于CPU

耳朵听到声音会发送信号给脑袋，声音来源有很多种，有远处的猫叫，门铃声，小孩哭声。这些声音传入耳朵，再由耳朵传给大脑，除了这些可以中断母亲的看书，还有其他情况，比如身体不舒服，有只蜘蛛掉下来，对于特殊情况无法回避，必须立即处理

对比我们的arm系统



有CPU，有中断控制器。

中断控制器可以发信号给CPU告诉它发生了那些紧急情况

中断源有按键、定时器、有其它的（比如网络数据）

这些信号都可以发送信号给中断控制器，再由中断控制器发送信号给CPU表明有这些中断产生了，这些成为中断(属于一种异常)

还有什么可以中断CPU运行？

指令不对，数据访问有问题

reset信号，这些都可以中断CPU 这些成为异常中断

重点在于**保存现场以及恢复现场**

处理过程

a 保存现场(各种寄存器)

b 处理异常(中断属于一种异常)

c 恢复现场

arm对异常(中断)处理过程

1 初始化:

a 设置中断源，让它可以产生中断
b 设置中断控制器(可以屏蔽某个中断，优先级)
c 设置CPU总开关，(使能中断)

2 执行其他程序:正常程序

3 产生中断:按下按键--->中断控制器--->CPU

4 cpu每执行完一条指令都会检查有无中断/异常产生

5 发现有中断/异常产生，开始处理。对于不同的异常，跳去不同的地址执行程序。这地址上，只是一条跳转指令，跳去执行某个函数(地址)，这个就是异常向量。如下就是异常向量表,对于不同的异常都有一条跳转指令。

```
.globl _start
_start: b   reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq //发生中断时，CPU跳到这个地址执行该指令 **假设地址为0x18**
    ldr pc, _fiq
//我们先在0x18这里放 ldr pc ,__irq,于是cpu最终会跳去执行__irq代码
//保护现场，调用处理函数，恢复现场
```

(3-5都是硬件强制做的)

6 这些函数做什么事情?

软件做的:
a 保存现场(各种寄存器)

b 处理异常(中断):

分辨中断源
再调用不同的处理函数

c 恢复现场

对比母亲的处理过程来比较arm中断的处理过程。

中断处理程序怎么被调用?

CPU--->0x18 --跳转到其他函数->

做保护现场
调用函数

分辨中断源
调用对应函数

恢复现场

cpu到0x18是由硬件决定的，跳去执行更加复杂函数(由软件决定)

第002节_CPU模式(Mode)_状态(State)与寄存器

这节课我们来讲CPU的工作模式(Mode) 状态(State)寄存器

7种Mode:

```
usr/sys
undefined(und)
Supervisor(svc)
Abort(abt)
IRQ(irq)
FIQ(fiq)
```

2种State:

```
ARM state
Thumb state
```

寄存器:

```
通用寄存器
备份寄存器(banked register)
当前程序状态寄存器(Current Program Status Register);CPSR
CPSR的备份寄存器:SPSR(Save Program Status Register)
```

我们仍然以这个母亲为例讲解这个CPU模式 这个母亲无压力看书 -->(正常模式) 要考试，看书--->(兴奋模式) 生病---->(异常模式)

可以参考书籍 《ARM体系结构与编程》作者：杜春雷

对于ARM CPU有7种模式:

1 usr ： 类比 正常模式

2 sys ： 类比的话兴奋模式

3 5种异常模式：(2440用户手册72页)

3.1 und ： 未定义模式
3.2 svc ： 管理模式
3.3 abt ： 终止模式

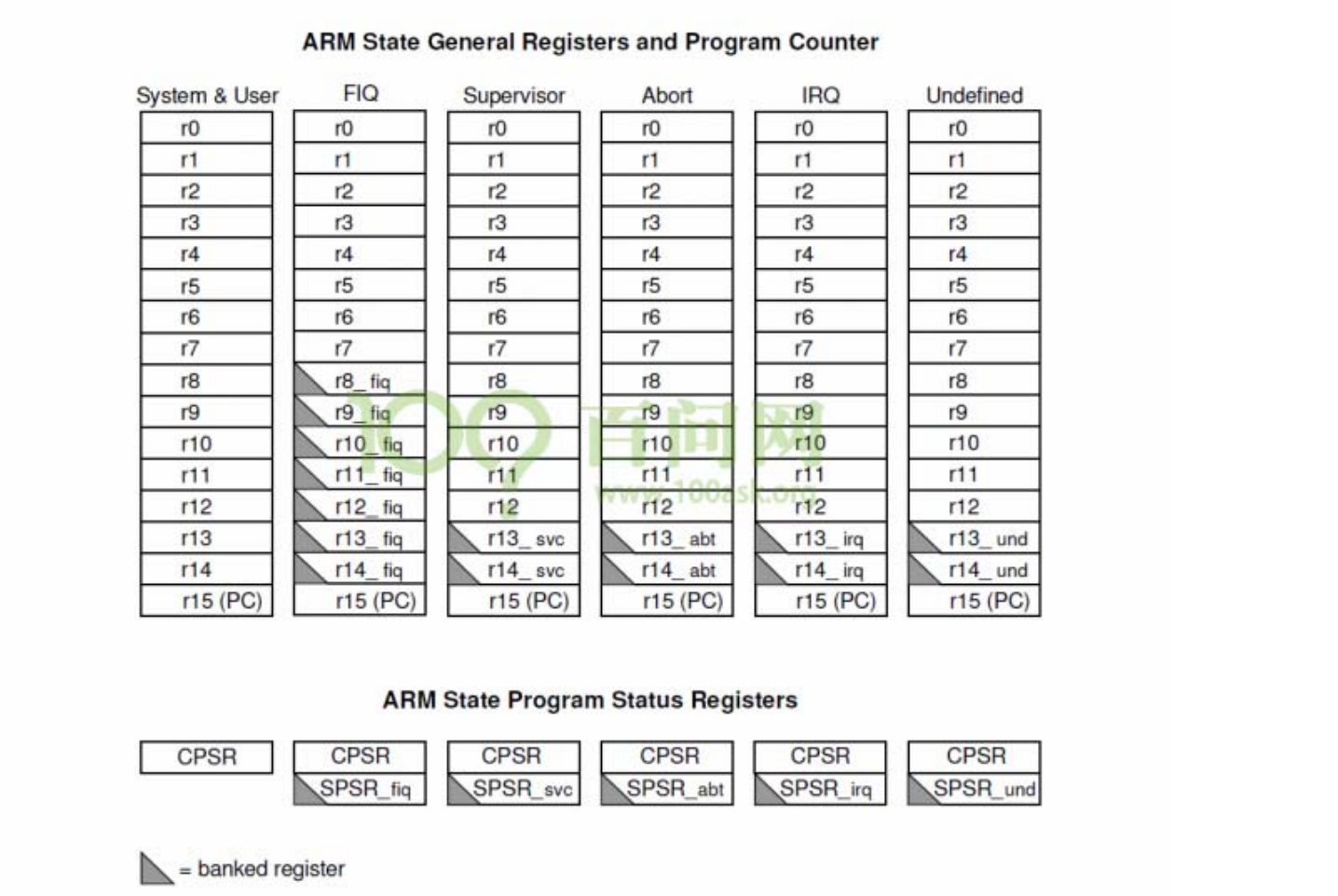
a 指令预取终止(读写某条错误的指令导致终止运行)
b 数据访问终止 (读写某个地址，这个过程出错)
都会进入终止模式

3.4 IRQ： 中断模式
3.5 FIQ： 快中断模式

我们可以称以下6种为特权模式

```
und ： 未定义模式
svc ： 管理模式
abt ： 终止模式
IRQ ： 中断模式
FIQ ： 快中断模式
sys ： 系统模式
```

usr用户模式(不可直接进入其他模式) 可以编程操作CPSR直接进入其他模式



这个图是有关各个模式下能访问寄存器的，再讲这个图之前我们先引入 2种state

CPU有两种state:

- 1 ARM state:使用ARM指令集，每个指令4byte
- 2 Thumb state:使用的是Thumb指令集，每个指令2byte

比如同样是:

mov R0, R1 编译后

对于ARM指令集要占据4个字节：机器码

对于Thumb指令集占据2个字节：机器码

引入Thumb减少存储空间

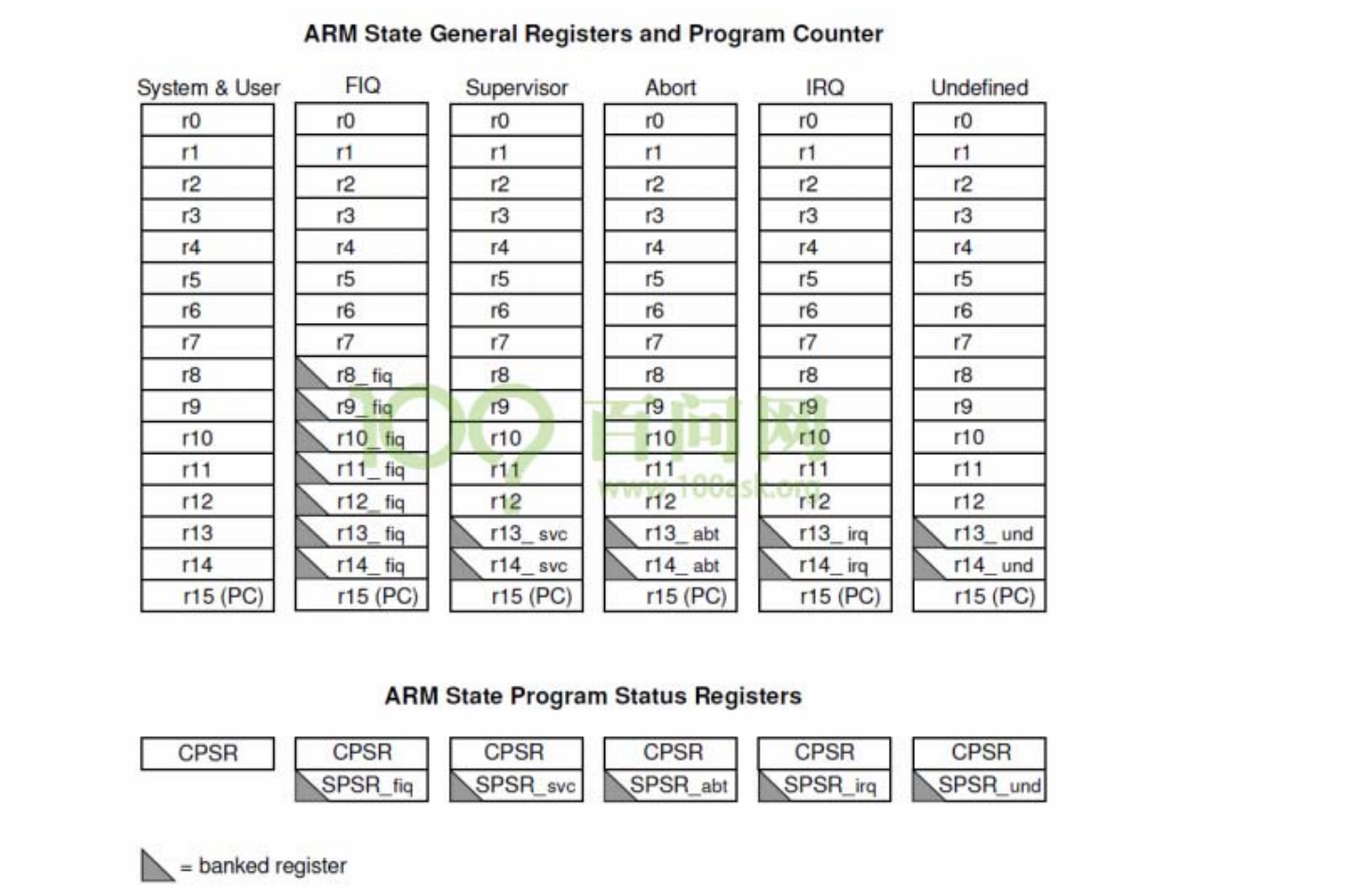
ARM指令集与Thumb指令集的区别：

Thumb 指令可以看作是 ARM 指令压缩形式的子集,是针对代码密度的问题而提出的,它具有 16 位的代码密度但是它不如ARM指令的效率高 .

Thumb 不是一个完整的体系结构,不能指望处理只执行Thumb 指令而不支持 ARM 指令集.

因此,Thumb 指令只需要支持通用功能,必要时可以借助于完善的 ARM 指令集,比如,所有异常自动进入 ARM 状态.在编写 Thumb 指令时,先要使用伪指令 CODE16 声明,而且在 ARM 指令中要使用 BX指令跳转到 Thumb 指令,以切换处理器状态.编写 ARM 指令时,则可使用伪指令 CODE32声明.

下节课会演示使用Thumb指令集编译，看是否生成的bin文件会变小很多



在每种模式下都有R0 ~ R15

在这张图注意到有些寄存器画有灰色的三角形，表示访问该模式下访问的专属寄存器 比如

```
mov R0, R8
mov R0, R8
```

在System 模式下访问的是R0 ~ R8,在所有模式下访问R0都是同一个寄存器

```
mov R0,R8_fiq
```

但是在FIQ模式下，访问R8是访问的FIQ模式专属的R8寄存器，不是同一个物理上的寄存器

在这五种异常模式中每个模式都有自己专属的R13 R14寄存器，R13用作SP(栈) R14用作LR(返回地址) LR是用来保存发生异常时的指令地址

为什么快中断(FIQ)有那么多专属寄存器，这些寄存器称为备份寄存器

回顾一下中断的处理过程

- 1 保存现场(保存被中断模式的寄存器)

就比如说我们的程序正在系统模式/用户模式下运行，当你发生中断时，需要把R0 ~ R14这些寄存器全部保存下来，让后处理异常，最后恢复这些寄存器

但如果是快中断，那么我就不需要保存 系统/用户模式下的R8 ~ R12这几个寄存器，在FIQ模式下有自己专属的R8 ~ R12寄存器，省略保存寄存器的时间，加快处理速度

但是在Linux中并不会使用FIQ模式

- 2 处理
- 3 恢复现场

CRSR当前程序状态寄存器，这是一个特别重要的寄存器

SPSR保存的程序状态寄存器，他们格式如下：

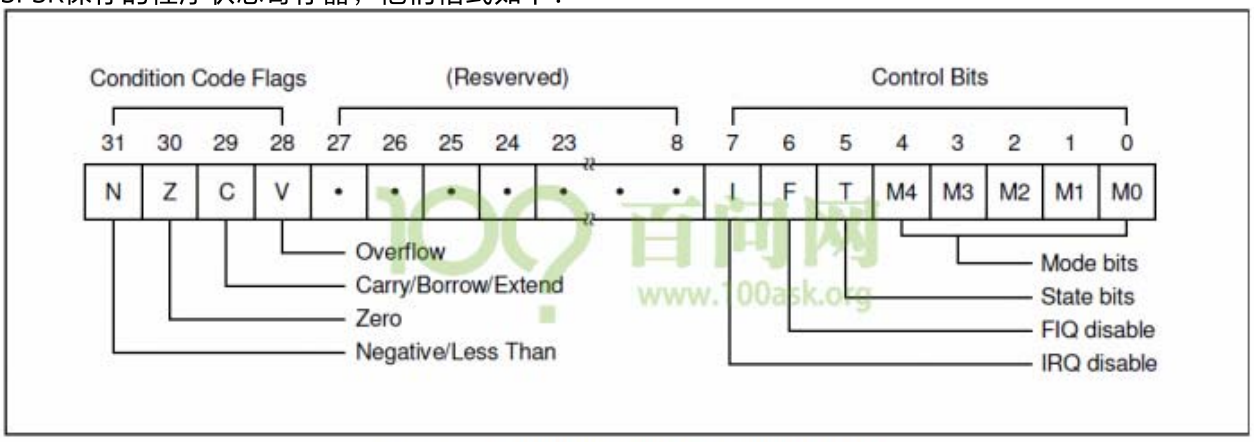


Figure 2-6. Program Status Register Format

首先 M4 ~ M0 表示当前CPU处于哪一种模式(Mode)；

我们可以读取这5位来判断CPU处于哪一种模式，也可以修改这一种模式位，让其修改这种模式；

假如你当前处于用户模式下，是没有权限修改这些位的；

M4 ~ M0对应什么值，会有说明：

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

查看其他位

```
Bit5 State bits表示CPU工作与Thumb State还是ARM State用的指令集是什么
Bit6 FIQ disable当bit6等于1时，FIQ是不工作的
Bit7 IRQ disable当bit5等于1时，禁止所有的IRQ中断，这个位是IRQ的总开关
Bit8 ~ Bit27是保留位
Bite28 ~ Bit31是状态位，
```

什么是状态位，比如说执行一条指令

```
cmp R0, R1
```

如果R0 等于 R1 那么zero位等于1，这条指令影响 Z 位，如果R0 == R1，则Z = 1

beq跳转到xxx这条指令会判断Bit30是否为1，是1的话则跳转，不是1的话则不会跳转 使用 Z 位，如果 Z 位等于1则跳转，这些指令是借助状态位实现的

SPSR保存的程序状态寄存器： 表示发生异常时这个寄存器会用来保存被中断的模式下他的CPSR

就比如我我的程序在系统模式下运行 CPSR是某个值，当发生中断时会进入irq模式，这个CPSR_irq就保存系统模式下的CPSR

我们来看看发生异常时CPU是如何协同工作的：

进入异常的处理流程(硬件)

Action on Entering an Exception

While handling an exception, the ARM920T does following activities:

1. Preserves the address of the next instruction in the appropriate Link Register. If the exception has been entered from ARM state, then the address of the next instruction is copied into the Link Register (that is, current PC + 4 or PC + 8 depending on the exception. See Table 2-2 on for details). If the exception has been entered from THUMB state, then the value written into the Link Register is the current PC offset by a value such that the program resumes from the correct place on return from the exception. This means that the exception handler need not determine which state the exception was entered from. For example, in the case of SWI, MOVS PC, R14_svc will always return to the next instruction regardless of whether the SWI was executed in ARM or THUMB state.
2. Copies the CPSR into the appropriate SPSR
3. Forces the CPSR mode bits to a value which depends on the exception
4. Forces the PC to fetch the next instruction from the relevant exception vector

我们来翻译一下：

发生异常时，我们的CPU会做什么事情

- 1把下一条指令的地址保存在LR寄存器里(某种异常模式的LR等于被中断的下一条指令的地址)

它有可能是PC + 4有可能是PC + 8,到底是那种取决于不同的情况

- 2 把CPSR保存在SPSR里面(某一种异常模式下SPSR里面的值等于CPSR)
- 3 修改CPSR的模式为进入异常模式(修改CPSR的M4 ~ M0进入异常模式)
- 4 跳到向量表

退出异常怎么做？

Action on Leaving an Exception

On completion, the exception handler:

1. Moves the Link Register, minus an offset where appropriate, to the PC. (The offset will vary depending on the type of exception.)
2. Copies the SPSR back to the CPSR
3. Clears the interrupt disable flags, if they were set on entry

- 1 让LR减去某个值，让后赋值给PC(PC = 某个异常LR寄存器减去 offset)

减去什么值呢？

也就是我们怎么返回去继续执行原来的程序，根据下面这个表来取值

Table 2-2. Exception Entry/Exit

Return Instruction		Previous State		Notes
		ARM R14_x	THUMB R14_x	
BL	MOV PC, R14	PC + 4	PC + 2	(1)
SWI	MOVS PC, R14_svc	PC + 4	PC + 2	(1)
UDEF	MOVS PC, R14_und	PC + 4	PC + 2	(1)
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4	(2)
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4	(2)
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4	(1)
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8	(3)
RESET	NA	–	–	(4)

如果发生的是SWI可以把 R14_svc复制给PC

如果发生的是IRQ可以把R14_irq的值减去4赋值给PC

- 2 把CPSR的值恢复(CPSR 值等于 某一个一场模式下的SPSR)
- 3 清中断（如果是中断的话，对于其他异常不用设置）

第003节_不重要_Thumb指令集程序示例

在上节视频里说ARMCPU有两种状态

ARM State 每条指令会占据4byte

Thumb State 每条指令占据2byte

我们说过Thumb指令集并不重要，本节演示把一个程序使用Thumb指令集来编译它

使用上一章节的重定位代码，打开Makefile和Start.S

Makefile文件

```
all:
    arm-linux-gcc -c -o led.o led.c
    arm-linux-gcc -c -o uart.o uart.c
    arm-linux-gcc -c -o init.o init.c
    arm-linux-gcc -c -o main.o main.c
    arm-linux-gcc -c -o start.o start.S
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis
```

对于使用Thumb指令集

```
all:
    arm-linux-gcc -mthumb -c -o led.o led.c//只需要在arm-linux-gcc加上 mthumb命令即可
    arm-linux-gcc -c -o uart.o uart.c
    arm-linux-gcc -c -o init.o init.c
    arm-linux-gcc -c -o main.o main.c
    arm-linux-gcc -c -o start.o start.S
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis
```

改进

```


all: led.o uart.o init.o main.o start.o //all依赖led.o uart.o init.o main.o start.o
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis

%.o : %.c
    arm-linux-gcc -mthumb -c -o $@ $< //对于所有的.c文件使用规则就可以使用thumb指令集编译 $@表示目标 $<表示第一个依赖

%.o : %.S
    arm-linux-gcc -c -o $@ $<
```

对start.S需要修改代码

原重定位章节Start.S文件

```
.text
.global _start

_start:

    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */



    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

使用thumb指令集的Start.S文件

```
.text
.global _start
.code 32 //表示后续的指令使用ARM指令集
_start:

    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */



    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    /* 怎么从ARM State切换到Thumb State? */
    adr r0, thumb_func //定义此标号的地址
    add r0, r0, #1  /* bit0=1时, bx就会切换CPU State到thumb state */
    bx r0

.code 16 //下面都使用thumb指令集
thumb_func: //需要得到这个标号的地址
    /*下面就是使用thumb指令来执行程序*/
    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr r0, =main  /* 绝对跳转, 跳到SDRAM ,先把main的地址赋值给R0 */
    mov pc, r0  /*让后再移动到PC*/

halt:
    b halt
```

上传代码编译测试

出现错误,如下

```
init.o(.text+0x6c):In function 'sdram_init2';
undefined reference to 'memcpy'
```

发现是init,o里sdram_init2使用的了memcpy函数

查看init.c

```
#include "s3c2440_soc.h"

void sdram_init(void)
{
    BWSCON = 0x22000000;

    BANKCON6 = 0x18001;
    BANKCON7 = 0x18001;

    REFRESH  = 0x8404f5;

    BANKSIZE = 0xb1;

    MRSRB6   = 0x20;
    MRSRB7   = 0x20;
}

#if 0


/**************************************************************************
* 设置控制SDRAM的13个寄存器
* 使用位置无关代码
**************************************************************************/
void memsetup(void)
{
    unsigned long *p = (unsigned long *)MEM_CTL_BASE;
    p[0] = 0x22111110;      //BWSCON
    p[1] = 0x00000700;      //BANKCON0
    p[2] = 0x00000700;      //BANKCON1
    p[3] = 0x00000700;      //BANKCON2
    p[4] = 0x00000700;      //BANKCON3
    p[5] = 0x00000700;      //BANKCON4
    p[6] = 0x00000700;      //BANKCON5
    p[7] = 0x00018005;      //BANKCON6
    p[8] = 0x00018005;      //BANKCON7
    p[9] = 0x008e07a3;      //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e04f4
    p[10] = 0x000000b2;     //BANKSIZE
    p[11] = 0x00000030;     //MRSRB6
    p[12] = 0x00000030;     //MRSRB7
}
#endif
/*下面函数使用了memcpy函数，显然是编译器的操作，使用了memcpy把数组里的值从代码段拷贝到了arr局部变量里
是否可以禁用掉memcpy*/
void sdram_init2(void)
{
    unsigned int arr[] = {
        0x22000000,     //BWSCON
        0x00000700,     //BANKCON0
        0x00000700,     //BANKCON1
        0x00000700,     //BANKCON2
        0x00000700,     //BANKCON3
        0x00000700,     //BANKCON4
        0x00000700,     //BANKCON5
        0x18001,    //BANKCON6
        0x18001,    //BANKCON7
        0x8404f5,   //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e04f4
         0xb1,  //BANKSIZE
         0x20,  //MRSRB6
         0x20,  //MRSRB7
```

```
        };
    volatile unsigned int * p = (volatile unsigned int *)0x48000000;
    int i;

    for (i = 0; i < 13; i++)
    {
        *p = arr[i];
        p++;
    }

}
```

文章说没有什么方法禁用memecpy但是可以修改这些变量

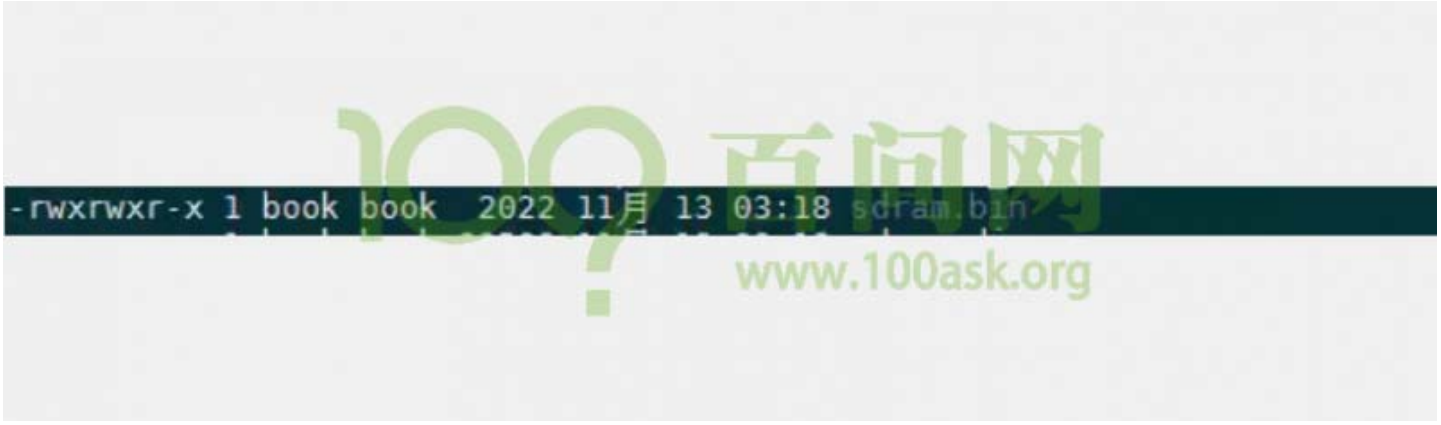
比如说将其修改为静态变量，这些数据就会放在数据段中，最终重定位时会把数据类拷贝到对应的arr地址里面去

```
void sdram_init2(void)
{
    const static unsigned int arr[] = {  //加上const 和static
        0x22000000,     //BWSCON
        0x00000700,     //BANKCON0
        0x00000700,     //BANKCON1
        0x00000700,     //BANKCON2
        0x00000700,     //BANKCON3
        0x00000700,     //BANKCON4
        0x00000700,     //BANKCON5
        0x18001,    //BANKCON6
        0x18001,    //BANKCON7
        0x8404f5,    //REFRESH,HCLK=12MHz:0x008e07a3,HCLK=100MHz:0x008e04f4
         0xb1,   //BANKSIZE
         0x20,   //MRSRB6
         0x20,   //MRSRB7

        };
    volatile unsigned int * p = (volatile unsigned int *)0x48000000;
    int i;

    for (i = 0; i < 13; i++)
    {
        *p = arr[i];
        p++;
    }

}
```

拷贝进行实验

得出bin文件有1.4k左右

查看之前的文件使用ARM指令集是2K左右



查看反汇编代码

```
sdram.elf:     file format elf32-littlearm

Disassembly of section .text:

/*前面这些ARM指令还是占用4个字节*/
30000000 <_start>:
30000000:   e3a00453    mov r0, #1392508928 ; 0x53000000
30000004:   e3a01000    mov r1, #0  ; 0x0
30000008:   e5801000    str r1, [r0]
3000000c:   e3a00313    mov r0, #1275068416 ; 0x4c000000
30000010:   e3e01000    mvn r1, #0  ; 0x0
30000014:   e5801000    str r1, [r0]
30000018:   e59f005c    ldr r0, [pc, #92]   ; 3000007c <.text+0x7c>
3000001c:   e3a01005    mov r1, #5  ; 0x5
30000020:   e5801000    str r1, [r0]
30000024:   ee110f10    mrc 15, 0, r0, cr1, cr0, {0}
30000028:   e3800103    orr r0, r0, #-1073741824    ; 0xc0000000
3000002c:   ee010f10    mcr 15, 0, r0, cr1, cr0, {0}
30000030:   e59f0048    ldr r0, [pc, #72]   ; 30000080 <.text+0x80>
30000034:   e59f1048    ldr r1, [pc, #72]   ; 30000084 <.text+0x84>
30000038:   e5801000    str r1, [r0]
3000003c:   e3a01000    mov r1, #0  ; 0x0
30000040:   e5910000    ldr r0, [r1]
30000044:   e5811000    str r1, [r1]
30000048:   e5912000    ldr r2, [r1]
3000004c:   e1510002    cmp r1, r2
30000050:   e59fd030    ldr sp, [pc, #48]   ; 30000088 <.text+0x88>
30000054:   03a0da01    moveq   sp, #4096   ; 0x1000
30000058:   05810000    streq   r0, [r1]
3000005c:   e28f0004    add r0, pc, #4  ; 0x4
30000060:   e2800001    add r0, r0, #1  ; 0x1
30000064:   e12fff10    bx  r0

30000068 <thumb_func>:
30000068:   f94ef000    bl  30000308 <sdram_init>
3000006c:   f9fef000    bl  3000046c <copy2sdram>
30000070:   fa24f000    bl  300004bc <clean_bss>
/**下面的thumb指令占据2个字节**/
30000074:   4805        ldr r0, [pc, #20]   (3000008c <.text+0x8c>)
30000076:   4687        mov pc, r0

30000078 <halt>:
30000078:   e7fe        b   30000078 <halt>
3000007a:   0000        lsl r0, r0, #0
3000007c:   0014        lsl r4, r2, #0
3000007e:   4c00        ldr r4, [pc, #0]    (30000080 <.text+0x80>)
30000080:   0004        lsl r4, r0, #0
30000082:   4c00        ldr r4, [pc, #0]    (30000084 <.text+0x84>)
30000084:   c011        stmia   r0!,{r0, r4}
30000086:   0005        lsl r5, r0, #0
30000088:   1000        asr r0, r0, #0
3000008a:   4000        and r0, r0
3000008c:   04fd        lsl r5, r7, #19
3000008e:   3000        add r0, #0
```

如果你的flash很小的话可以考虑使用Thumb指令集

烧写进去看是否可以运行

测试结果没有任何问题

Thumb指令集后面没有任何作用，只是简单作为介绍

第004节_und异常模示程序示例

写一个程序故意让其发生未定义异常，让后处理这个异常

查看uboot中源码uboot\u-boot-1.1.6\cpu\arm920t

打开start.S

```
/*code: 28 -- 72*/
#include <config.h>
#include <version.h>


/*
 *************************************************************************
 *
 * Jump vector table as in table 3.1 in [1]
 *
 *************************************************************************
 */
#define GSTATUS2    (0x560000B4)
#define GSTATUS3    (0x560000B8)
#define GSTATUS4    (0x560000BC)

#define REFRESH(0x48000024)
#define MISCCR (0x56000080)

#define LOCKTIME    0x4C000000  /* R/W, PLL lock time count register */
#define MPLLCON     0x4C000004  /* R/W, MPLL configuration register */
#define UPLLCON     0x4C000008  /* R/W, UPLL configuration register */
#define CLKCON      0x4C00000C  /* R/W, Clock generator control reg. */
#define CLKSLOW     0x4C000010  /* R/W, Slow clock control register */
#define CLKDIVN     0x4C000014  /* R/W, Clock divider control */

/******下面这些就是异常向量表*****/
.globl _start
_start: b   reset
    ldr pc, _undefined_instruction
    ldr pc, _software_interrupt
    ldr pc, _prefetch_abort
    ldr pc, _data_abort
    ldr pc, _not_used
    ldr pc, _irq
    ldr pc, _fiq

_undefined_instruction: .word undefined_instruction
_software_interrupt:    .word software_interrupt
_prefetch_abort:    .word prefetch_abort
_data_abort:        .word data_abort
_not_used:      .word not_used
_irq:           .word irq
_fiq:           .word fiq

    .balignl 16,0xdeadbeef
```

手册异常向量表定义

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

接下来我们写程序

```
    .text
    .global _start

    _start:
        b reset  /* vector 0 : reset */  //一上电复位，是从0地址开始执行，跳到reset处
        b do_und /* vector 4 : und */ //如果发生未定义指令异常，就会跳到0x04地址未定义指令异常处，执行do_und程序

    /*假设一上电从0地址开始执行，reset，做一系列初始化之后
    *故意加入一条未定义指令

    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
    当CPU发现无法执行此条指令时，就会发生未定义指令异常，就会执行do_und
        bl print2,
    */
    do_und:
        /* 执行到这里之前:
         * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
         * 2. SPSR_und保存有被中断模式的CPSR
         * 3. CPSR中的M4-M0被设置为11011, 进入到und模式
         * 4. 跳到0x4的地方执行程序
         */
//需要从新设置sp栈，指向某一块没有使用的地址
        /* sp_und未设置, 先设置它 */
        ldr sp, =0x34000000

        /* 在und异常处理函数中有可能会修改r0-r12, 所以先保存 */
        /* 发生异常时，当前被中断的地址会保存在lr寄存器中 先减后存*/
        /* lr是异常处理完后的返回地址, 也要保存 */
        stmdb sp!, {r0-r12, lr}

        /* 保存现场 */
        /* 处理und异常 */
        mrs r0, cpsr//把cpsr的值读入r0
        ldr r1, =und_string//把下面的字符串地址赋值给r1

        bl printException

        /* 这些寄存器保存在栈中，把他读取出来就可以了*/
        /* 恢复现场 */
        /* 先读后加*/
        /* 把r0 ~ r12的值从栈中都取出来，并且把原来保存的lr值，赋值到pc中去*/
        ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */
```

```


/*
*如何定义字符串，可以百度搜索 arm-linux-gcc 汇编 定义字符串
*
*官方的说明文档
*http://web.mit.edu/gnu/doc/html/as_7.html
.string "str"

Copy the characters in str to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise sp

我们使用.str会自动加上结束符
*/
    und_string:
```

```
    .string "undefined instruction exception"


reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */


    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xff123456  /* 未定义指令 */
    bl print2

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

如何处理这个异常呢？

直接print打印一句话，新建一个exception.c文件

```
#include "uart.h"

void printException(unsigned int cpsr, char *str) //cpsr打印相应的寄存器，str打印一个字符串
{
    puts("Exception! cpsr = ");\\打印cpsr
    printHex(cpsr);//输出cpsr的值
    puts(" ");//输出空格
    puts(str);//输出str值
    puts("\n\r");//回车，换行
}
```

我们打开之前编译过的程序的反汇编文件

里面一定包含了保存恢复

```
30000084 <delay>:
30000084:   e1a0c00d    mov ip, sp
30000088:   e92dd800    stmdb   sp!, {fp, ip, lr, pc} //保存 d是减 b是存
3000008c:   e24cb004    sub fp, ip, #4  ; 0x4
30000090:   e24dd004    sub sp, sp, #4  ; 0x4
30000094:   e50b0010    str r0, [fp, #-16]
30000098:   e51b3010    ldr r3, [fp, #-16]
3000009c:   e2433001    sub r3, r3, #1  ; 0x1
300000a0:   e50b3010    str r3, [fp, #-16]
300000a4:   e51b3010    ldr r3, [fp, #-16]
300000a8:   e3730001    cmn r3, #1  ; 0x1
300000ac:   0a000000    beq 300000b4 <delay+0x30>
300000b0:   eafffff8    b   30000098 <delay+0x14>
300000b4:   e89da808    ldmia   sp, {r3, fp, sp, pc}//恢复，先读后加
```

上传编译

修改makefile添加文件

```
all: start.o led.o uart.o init.o main.o exception.o
    #arm-linux-ld -Ttext 0 -Tdata 0x30000000  start.o led.o uart.o init.o main.o -o sdram.elf
    arm-linux-ld -T sdram.lds $^ -o sdram.elf
    #用$ ^来包含所有的依赖
    arm-linux-objcopy -O binary -S sdram.elf sdram.bin
    arm-linux-objdump -D sdram.elf > sdram.dis
clean:
    rm *.bin *.o *.elf *.dis

%.o : %.c
    arm-linux-gcc -c -o $@ $<

%.o : %.S
    arm-linux-gcc -c -o $@ $<
    *.dis
```

编译成功烧写

没有输出我们想要的字符串

很多同学想学会如何调试程序

这里我们演示

```
sdram:
    bl print1 //添加print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2 //添加print2,实现这两个函数，来打印

    //bl main  /* 使用BL命令相对跳转，程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转，跳到SDRAM */
```

```
halt:
    b halt
```

实现print1 print2这两个打印函数，在uart.c这个文件里

```
void print1(void)
{
    puts("abc\n\r");
}

void print2(void)
{
    puts("123\n\r");
}
```

上传代码烧写，发现print1、print2并未执行成功

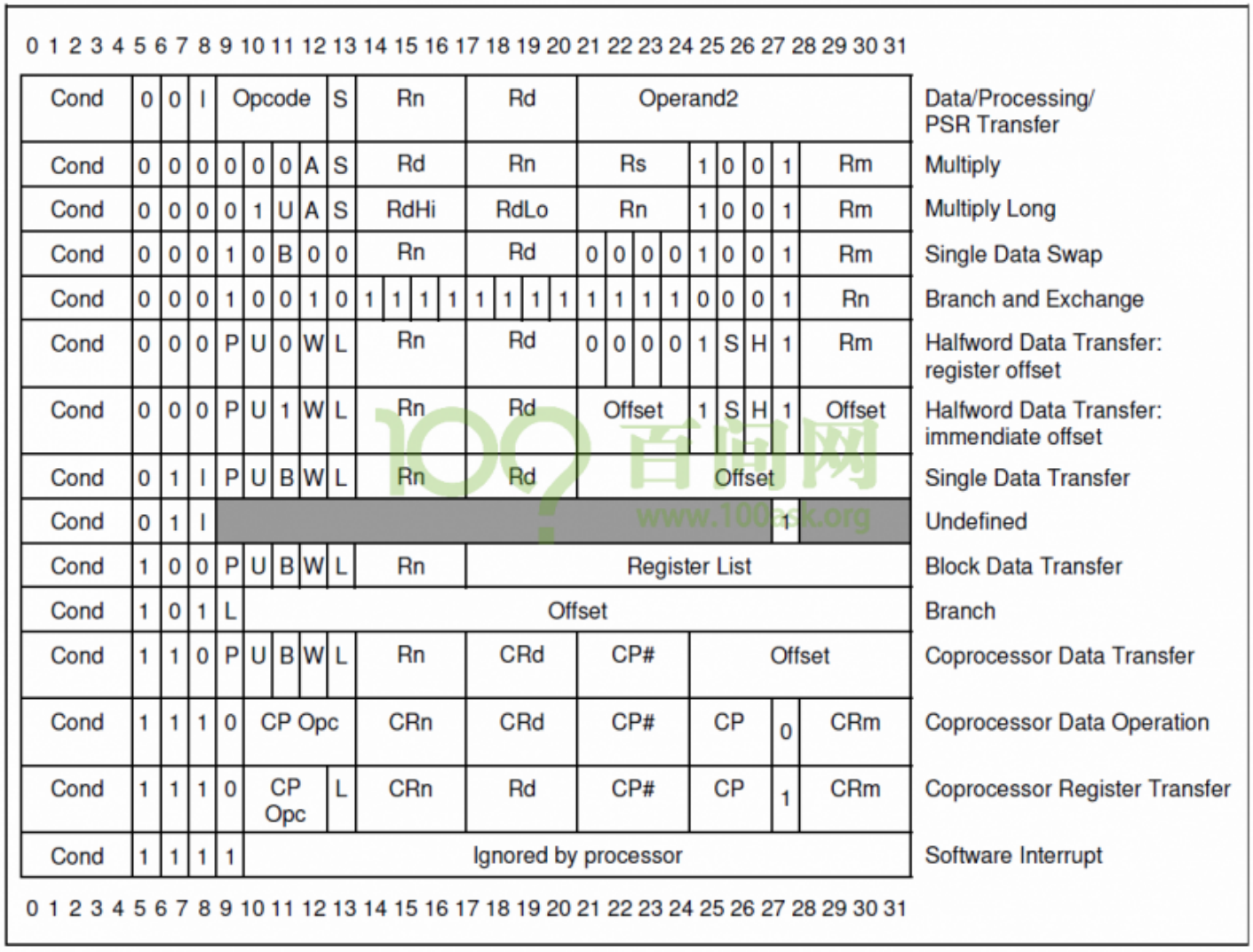
发现在start.S并未初始化 uart0_init()，删除main.c中的uart0_init()初始化函数

```
    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xff123456  /* 未定义指令 */
    bl print2

    //bl main  /* 使用BL命令相对跳转，程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转，跳到SDRAM */

halt:
    b halt
```

加上uart0_init，再次编译烧写

程序正常运行，print1 print2全部打印，表明未定义指令并未运行，难道这个地址是一个已经定义的地址

打开2440芯片手册，找到ARM指令集



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	
Cond 0 0 I Opcode S Rn Rd Operand2	Data/Processing/ PSR Transfer
Cond 0 0 0 0 0 0 A S Rd Rn Rs 1 0 0 1 Rm	Multiply
Cond 0 0 0 0 1 U A S RdHi RdLo Rn 1 0 0 1 Rm	Multiply Long
Cond 0 0 0 1 0 B 0 0 Rn Rd 0 0 0 0 1 0 0 1 Rm	Single Data Swap
Cond 0 0 0 1 0 0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 1 Rn	Branch and Exchange
Cond 0 0 0 P U 0 W L Rn Rd 0 0 0 0 1 S H 1 Rm	Halfword Data Transfer: register offset
Cond 0 0 0 P U 1 W L Rn Rd Offset 1 S H 1 Offset	Halfword Data Transfer: immendiate offset
Cond 0 1 I P U B W L Rn Rd Offset	Single Data Transfer
Cond 0 1 I 1	Undefined
Cond 1 0 0 P U B W L Rn Register List	Block Data Transfer
Cond 1 0 1 L Offset	Branch
Cond 1 1 0 P U B W L Rn CRd CP# Offset	Coprocessor Data Transfer
Cond 1 1 1 0 CP Opc CRn CRd CP# CP 0 CRm	Coprocessor Data Operation
Cond 1 1 1 0 CP Opc L CRn Rd CP# CP 1 CRm	Coprocessor Register Transfer
Cond 1 1 1 1 Ignored by processor	Software Interrupt
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31	

发现竟然是SWI指令，CPU可以识别出来，他不是一条未定义指令

我们得找到一条CPU不能识别的指令,定义为0x03000000

```
    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0x03000000  /* 未定义指令 */
    bl print2

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

编译烧写执行

打印了未定义指令异常CPSR地址,打印了字符串，最后执行main函数

```
word 0xdeadcode  /* 也是一条未定义指令 只要指令地址对不上上表就是未定义指令*/
```

我们查看下cpsr是否处于未定义模式

bit[4:0]表示CPU模式 11011，果然处于und模式

我们看看这个程序做了什么事情

```
    .text
    .global _start
/*一上电复位，从0地址开始执行
跳到   reset:
做了一系列初始化
当执行到0xdeadc0de这条指令时候，CPU根本就不知道这条指令什么意思
    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2
让后就发生未定义指令异常，他会把下一条指令的地址保存到异常模式的LR寄存器

        /* 执行到这里之前已经发生了很多事情
         * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
         * 2. SPSR_und保存有被中断模式的CPSR
         * 3. CPSR中的M4-M0被设置为11011，进入到und模式
         * 4. 跳到0x4的地方执行程序
         *
         * 设置栈 sp是指und的地址
         * sp_und未设置，先设置它
         *  /* 在und异常处理函数中有可能会修改r0-r12，所以先保存 */
         * lr是异常处理完后的返回地址，也要保存 */
         * 保存现场 */
         * 处理und异常 */
         * 恢复sp
         * cpu就会切换到之前的模式
         */
*/
.text
.global _start

_start:
    b reset  /* vector 0 : reset */
    b do_und /* vector 4 : und */

    und_addr:
        .word do_und

    do_und:
        /* 执行到这里之前:
         * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
         * 2. SPSR_und保存有被中断模式的CPSR
         * 3. CPSR中的M4-M0被设置为11011，进入到und模式
         * 4. 跳到0x4的地方执行程序
         */

        /* sp_und未设置，先设置它 */
        ldr sp, =0x34000000

        /* 在und异常处理函数中有可能会修改r0-r12，所以先保存 */
        /* lr是异常处理完后的返回地址，也要保存 */
        stmdb sp!, {r0-r12, lr}

        /* 保存现场 */
        /* 处理und异常 */
        mrs r0, cpsr
        ldr r1, =und_string
        bl printException

        /* 恢复现场 */
        ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

    und_string:
        .string "undefined instruction exception"
```

程序改进

源程序

```
.text
.global _start

_start:
    b reset  /* vector 0 : reset */
/*使用b命令跳转  相对跳转*/
```

```
    b do_und /* vector 4 : und */

do_und:
    /* 执行到这里之前:
     * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_und保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为11011, 进入到und模式
     * 4. 跳到0x4的地方执行程序
     */

    /* sp_und未设置, 先设置它 */
    ldr sp, =0x34000000

    /* 在und异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 保存现场 */
    /* 处理und异常 */
    mrs r0, cpsr
    ldr r1, =und_string

    /*这里又使用bl指令跳转，如果是nand启动，这个函数在4k之外，这个函数必定出错 为了保险，跳转到sdram中执行程序*/
    bl printException

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

    und_string:
        .string "undefined instruction exception"


reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */



    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
```

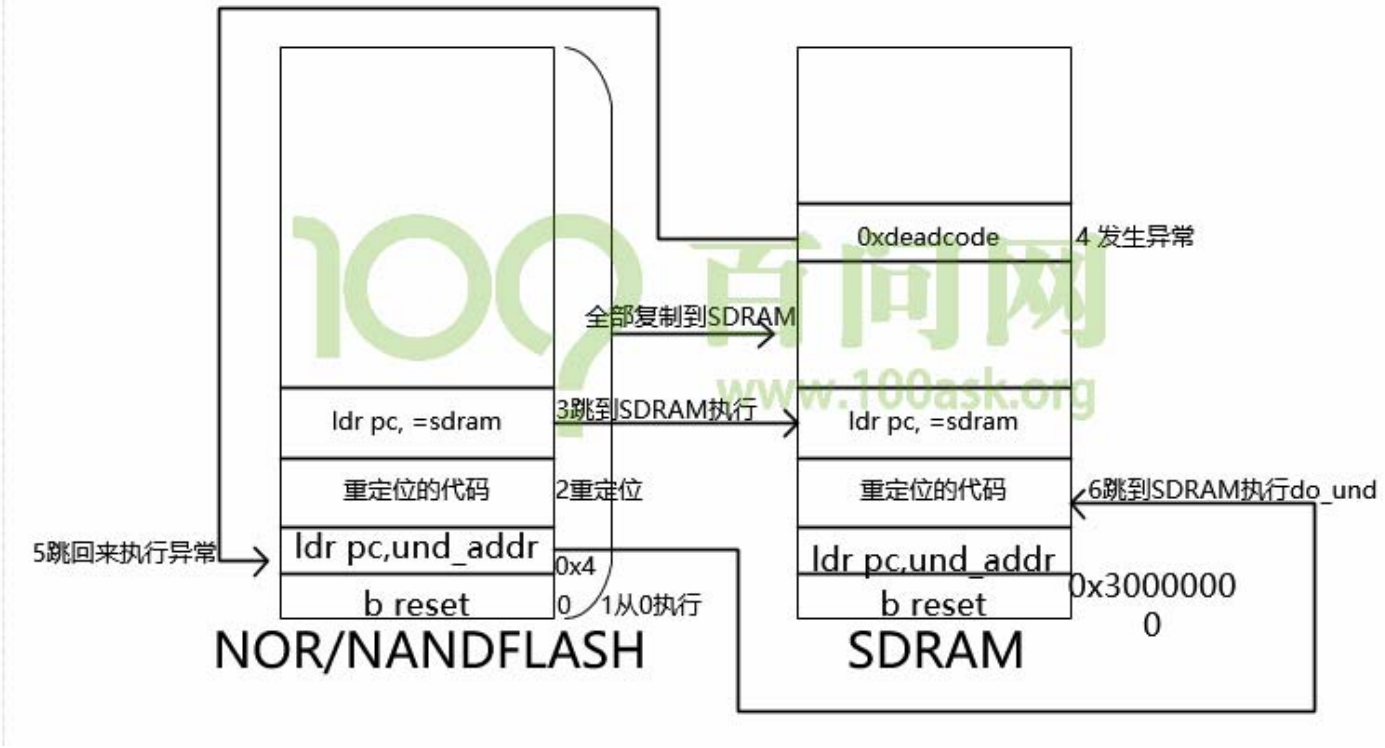
```
    streq r0, [r1]   /* 恢复原来的值 */

    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

改进后代码

```
    .text
    .global _start

    _start:
        b reset  /* vector 0 : reset */
/*跳转到sdram执行这个函数，那么这个函数一定在sdram中
我们需要指定让他去前面这块内存去读这个值，担心如果这个文件很大，超过4Knand就没法去读这个文件*/
        ldr pc, und_addr /* vector 4 : und */

/*增加如下 查看反汇编，在08的地址读让后跳到3c*/
    und_addr:
        .word do_und

    do_und:
        /* 执行到这里之前:
         * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
         * 2. SPSR_und保存有被中断模式的CPSR
         * 3. CPSR中的M4-M0被设置为11011, 进入到und模式
         * 4. 跳到0x4的地方执行程序
         */

        /* sp_und未设置, 先设置它 */
        ldr sp, =0x34000000

        /* 在und异常处理函数中有可能会修改r0-r12, 所以先保存 */
        /* lr是异常处理完后的返回地址, 也要保存 */
        stmdb sp!, {r0-r12, lr}

        /* 保存现场 */
        /* 处理und异常 */
        mrs r0, cpsr
        ldr r1, =und_string
        bl printException

        /* 恢复现场 */
        ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

    und_string:
        .string "undefined instruction exception"
    /**如果你的程序长度稍有变化，就不能保证运行
    加上 .align 4才能保证后面的程序以4字节对齐，保证程序运行
    **/
    .align 4

    reset:
        /* 关闭看门狗 */
        ldr r0, =0x53000000
        ldr r1, =0
        str r1, [r0]

        /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
        /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
        ldr r0, =0x4C000000
```

```
        ldr r1, =0xFFFFFFFF
        str r1, [r0]

        /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
        ldr r0, =0x4C000014
        ldr r1, =0x5
        str r1, [r0]

        /* 设置CPU工作于异步模式 */
        mrc p15,0,r0,c1,c0,0
        orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
        mcr p15,0,r0,c1,c0,0

        /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
         *  m = MDIV+8 = 92+8=100
         *  p = PDIV+2 = 1+2 = 3
         *  s = SDIV = 1
         *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
         */
        ldr r0, =0x4C000004
        ldr r1, =(92<<12)|(1<<4)|(1<<0)
        str r1, [r0]

        /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
         * 然后CPU工作于新的频率FCLK
         */



        /* 设置内存: sp 栈 */
        /* 分辨是nor/nand启动
         * 写0到0地址, 再读出来
         * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
         * 否则就是nor启动
         */
        mov r1, #0
        ldr r0, [r1] /* 读出原来的值备份 */
        str r1, [r1] /* 0->[0] */
        ldr r2, [r1] /* r2=[0] */
        cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
        ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
        moveq sp, #4096  /* nand启动 */
        streq r0, [r1]   /* 恢复原来的值 */

        bl sdram_init
        //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

        /* 重定位text, rodata, data段整个程序 */
        bl copy2sdram

        /* 清除BSS段 */
        bl clean_bss

/*把链接地址赋值给pc 直接就跳转到sdram中*/
        ldr pc, =sdram
    sdram:
        bl uart0_init

        bl print1
        /* 故意加入一条未定义指令 */
    und_code:
        .word 0xdeadc0de  /* 未定义指令 */
        bl print2

        //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
        ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

    halt:
        b halt
```

看一下整个程序的执行过程



第005节_swi异常模示程序示例

这节我们再来演示swi的处理流程

swi软件中断:software interrupt

在前面的视频中我们讲过ARMCPU有7中模式，除了用户模式以外，其他6种都是特权模式，这些特权模式可以直接修改CPSR进入其他模式

usr用户模式不能修改CPSR进入其他模式

Linux应用程序一般运行于用户模式

APP运行于usermode，(受限模式，不可访问硬件)

APP想访问硬件，必须切换模式，怎么切换？

发生异常3种模式

中断是一种异常
und也是
swi + 某个值(使用软中断切换模式)

现在start.S把要做的事情列出来

```
/*1
    /* 复位之后, cpu处于svc模式
     * 现在, 切换到usr模式
     * 设置栈
     * 跳转执行
 */

/*2 故意引入一条swi指令*/

/*3 需在_start这里放一条swi指令*/
```

查看异常向量表swi异常的向量地址是0x8

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

我们先切换到usr模式下

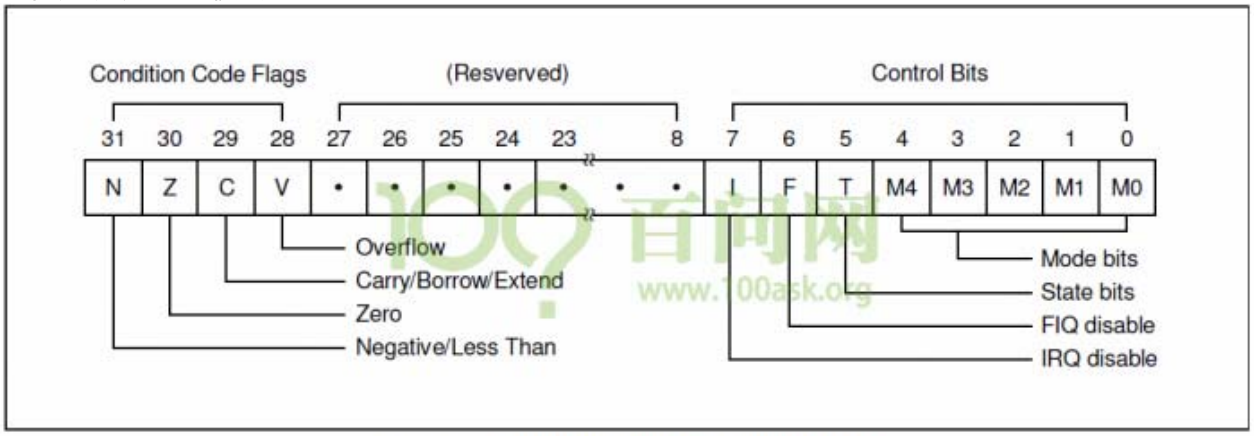


Figure 2-6. Program Status Register Format

Table 2-1. PSR Mode Bit Values

M[4:0]	Mode	Visible THUMB state registers	Visible ARM state registers
10000	User	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR
10001	FIQ	R7..R0, LR_fiq, SP_fiq PC, CPSR, SPSR_fiq	R7..R0, R14_fiq..R8_fiq, PC, CPSR, SPSR_fiq
10010	IRQ	R7..R0, LR_irq, SP_irq PC, CPSR, SPSR_irq	R12..R0, R14_irq, R13_irq, PC, CPSR, SPSR_irq
10011	Supervisor	R7..R0, LR_svc, SP_svc, PC, CPSR, SPSR_svc	R12..R0, R14_svc, R13_svc, PC, CPSR, SPSR_svc
10111	Abort	R7..R0, LR_abt, SP_abt, PC, CPSR, SPSR_abt	R12..R0, R14_abt, R13_abt, PC, CPSR, SPSR_abt
11011	Undefined	R7..R0 LR_und, SP_und, PC, CPSR, SPSR_und	R12..R0, R14_und, R13_und, PC, CPSR
11111	System	R7..R0, LR, SP PC, CPSR	R14..R0, PC, CPSR

usr模式下的 M0 ~ M4是10000

```
/**5 先进入usr模式*/
mrs r0, cpsr       /* 读出cpsr 读到r0 */
/使用bic命令 bitclean 把低4位清零/
bic r0, r0, #0xf   /* 修改M4-M0为0b10000, 进入usr模式 */
msr cpsr, r0

/*6 设置栈*/
```

```
/* 设置 sp_usr */
ldr sp, =0x33f00000
```

编译运行

发现可以处理und指令

添加 swi异常，仿照未定义指令做

```
.text
.global _start

_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
/*1 添加swi指令*/
    ldr pc, swi_addr /* vector 8 : swi */

und_addr:
    .word do_und

/*2 仿照und未定义添加指令*/
swi_addr:
    .word do_swi

do_und:
    /* 执行到这里之前:
     * 1. lr_und保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_und保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为11011, 进入到und模式
     * 4. 跳到0x4的地方执行程序
     */

    /* sp_und未设置, 先设置它 */
    ldr sp, =0x34000000

    /* 在und异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 保存现场 */
    /* 处理und异常 */
    mrs r0, cpsr
    ldr r1, =und_string
    bl printException

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

und_string:
    .string "undefined instruction exception"


/*3 复制do_und修改为swi */
do_swi:
    /* 执行到这里之前:
     * 3.1. lr_svc保存有被中断模式中的下一条即将执行的指令的地址
     * 3.2. SPSR_svc保存有被中断模式的CPSR
     * 3.3. CPSR中的M4-M0被设置为10011, 进入到svc模式
     * 3.4. 跳到0x08的地方执行程序
     */

    /* 3.5 sp_svc未设置, 先设置它 */
    ldr sp, =0x33e00000

    /* 3.6 在swi异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* 3.7 lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 3.8 保存现场 */
    /* 3.9 处理swi异常 只是打印 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    /*3.10  恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

/*swi处理函数*/
```

```
swi_string:
    .string "swi exception"
```

上传代码实验

烧写 发现没有执行

我们先把下面这些代码注释掉

```
/*3 复制do_und 修改为swi */

    /* 执行到这里之前:
     * 3.1. lr_svc保存有被中断模式中的下一条即将执行的指令的地址
     * 3.2. SPSR_svc保存有被中断模式的CPSR
     * 3.3. CPSR中的M4-M0被设置为10011, 进入到svc模式
     * 3.4. 跳到0x08的地方执行程序
     */

    /* 3.5 sp_svc未设置, 先设置它 */
    ldr sp, =0x33e00000

    /* 3.6 在swi异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* 3.7 lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 3.8 保存现场 */
    /* 3.9 处理swi异常 只是打印 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    /*3.10  恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

/* swi处理函数 */
swi_string:
    .string "swi exception"
```

上传编译

烧写执行 可以正常运行

循环打印

```
swi 0x123  /* 执行此命令, 触发SWI异常, 进入0x8执行 */
```

执行后继续执行

```
ldr pc, swi_addr /* vector 8  : swi */
```

表明问题出现在 do_swi:函数中 先把下面这句话注释掉 `.string "swi exception"`

编译烧写运行

程序可以正常运行

显然程序问题出现在`.string "swi exception"` 这句话，为什么加上这句话程序就无法执行，查看一下反汇编：

```
30000064 <swi_string>: //这里地址是64
30000064:   20697773    rsbcs   r7, r9, r3, ror r7
30000068:   65637865    strvsb  r7, [r3, #-2149]!
3000007c:   6f697470    swivs   0x00697470
30000070:   0000006e    andeq   r0, r0, lr, rrx

30000082 <reset>: //我们使用的是ARM指令集，应该是4字节对齐，发现这里并不是，问题就在这里
30000082:   e3a00453    mov r0, #1392508928 ; 0x53000000
30000086:   e3a01000    mov r1, #0  ; 0x0
3000008a:   e5801000    str r1, [r0]
```

```
3000008e:   e3a00313    mov r0, #1275068416 ; 0x4c000000
30000092:   e3e01000    mvn r1, #0  ; 0x0
```

因为这个字符串长度有问题

前面und_string 那里的字符串长度刚刚好

我们不能把问题放在运气上面

添加：

```
/*******
以4字节对齐
*/
.align 4


do_swi:
    /* 执行到这里之前:
     * 3.1. lr_svc保存有被中断模式中的下一条即将执行的指令的地址
     * 3.2. SPSR_svc保存有被中断模式的CPSR
     * 3.3. CPSR中的M4-M0被设置为10011, 进入到svc模式
     * 3.4. 跳到0x08的地方执行程序
     */

    /* 3.5 sp_svc未设置, 先设置它 */
    ldr sp, =0x33e00000

    /* 3.6 在swi异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* 3.7 lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

    /* 3.8 保存现场 */
    /* 3.9 处理swi异常 只是打印 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    /*3.10  恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */
/*****
swi处理函数
*/
swi_string:
    .string "swi exception"


.align 4
/**************
表明下面的标号要放在4字节对齐的地方
*/
```

上传代码编译运行查看反汇编：

```
30000068 <swi_string>:
30000068:   20697773    rsbcs   r7, r9, r3, ror r7
3000006c:   65637865    strvsb  r7, [r3, #-2149]!
30000070:   6f697470    swivs   0x00697470
30000074:   0000006e    andeq   r0, r0, lr, rrx
    ...

30000080 <reset>: //现在reset放在4自己对齐的地方
30000080:   e3a00453    mov r0, #1392508928 ; 0x53000000
30000084:   e3a01000    mov r1, #0  ; 0x0
30000088:   e5801000    str r1, [r0]
3000008c:   e3a00313    mov r0, #1275068416 ; 0x4c000000
30000090:   e3e01000    mvn r1, #0  ; 0x0
30000094:   e5801000    str r1, [r0]
30000098:   e59f0084    ldr r0, [pc, #132]  ; 30000124 <.text+0x124>
3000009c:   e3a01005    mov r1, #5  ; 0x5
300000a0:   e5801000    str r1, [r0]
300000a4:   ee110f10    mrc 15, 0, r0, cr1, cr0, {0}
300000a8:   e3700103    orr r0, r0, #-1073741824    ; 0xc0000000
300000ac:   ee010f10    mcr 15, 0, r0, cr1, cr0, {0}
300000b0:   e59f0070    ldr r0, [pc, #112]  ; 30000128 <.text+0x128>
```

```
300000b4:   e59f1070    ldr r1, [pc, #112]  ; 3000012c <.text+0x12c>
300000b8:   e5801000    str r1, [r0]
300000bc:   e3a01000    mov r1, #0  ; 0x0
300000c0:   e5910000    ldr r0, [r1]
```

下载烧写

程序执行完全没有问题

程序备份修改代码

swi可以根据应用程序传入的val来判断为什么调用swi指令，我们的异常处理函数能不能把这个val值读出来

```
do_swi:
    /* 执行到这里之前:
     * 1. lr_svc保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_svc保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10011, 进入到svc模式
     * 4. 跳到0x08的地方执行程序
     */

    /* sp_svc未设置, 先设置它 */
    ldr sp, =0x33e00000

    /* 保存现场 */
    /* 在swi异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr是异常处理完后的返回地址, 也要保存 */
    stmdb sp!, {r0-r12, lr}

/* 2
我们要把lr拿出来保存
因为bl printException会破坏lr

mov rX,  lr

我把lr保存在那个寄存器?

这个函数 bl printException 可能会修改某些寄存器，但是又会恢复这些寄存器，我得知道他会保护那些寄存器

我们之前讲过ATPCS规则
对于 r4 ~ r11在C函数里他都会保存这几个寄存器，如果用到的话就把他保存起来，执行完C函数再把它释放掉
我们把lr 保存在r4寄存器里，r4寄存器不会被C语言破坏
*/
    mov r4, lr

    /* 处理swi异常 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException


/*1
跳转到printSWIVal
如何才能知道swi的值呢?
我们得读出swi 0x123指令，这条指令保存在内存中，我们得找到他的内存地址
执行完0x123指令以后，会发生一次异常，那个异常模式里的lr寄存器会保存下一条指令的地址
我们把lr寄存器的地址减去4就是swi 0x123这条指令的地址
*/

/*3
我再把r4的寄存器赋给r0让后打印
我们得写出打印函数

mov r0, r4

指令地址减4才可以
swi 0x123
下一条指令bl main 减4就是指令本身
*/
    sub r0, r4, #4
    bl printSWIVal

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

swi_string:
.string "swi exception"
```

<syntaxhighlight lang="c" >
在uart.c添加printSWIVal打印函数

```
void printSWIVal(unsigned int *pSWI)
{

    puts("SWI val = ");
    printHEx(*pSWI & ~0xff000000); //高8位忽略掉
    puts("\n\r");

}
```

编译实验运行没有问题

我们再来看看这个程序是怎么跳转的

```
/*1
发生swi异常，他是在sdram中，CPU就会跳到0x8的地方
swi 0x123  /* 执行此命令，触发SWI异常，进入0x8执行 */

*/

/* 2

_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
执行这条读内存指令
    ldr pc, swi_addr /* vector 8 : swi */

读到swi_addr地址跳转到sdram执行代码 do_swi那段代码
swi_addr:
    .word do_swi

*/

/* 3
这段代码被设置栈保存现场 调用处理函数恢复现场，让后就会跳到sdram执行 swi 0x123的下一条指令
do_swi:
    /* 执行到这里之前:
     * 1. lr_svc保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_svc保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10011，进入到svc模式
     * 4. 跳到0x08的地方执行程序
     */

    /* sp_svc未设置，先设置它 */
    ldr sp, =0x33e00000

    /* 保存现场 */
    /* 在swi异常处理函数中有可能会修改r0-r12，所以先保存 */
    /* lr是异常处理完后的返回地址，也要保存 */
    stmdb sp!, {r0-r12, lr}

    mov r4, lr

    /* 处理swi异常 */
    mrs r0, cpsr
    ldr r1, =swi_string
    bl printException

    sub r0, r4, #4
    bl printSWIVal

    /* 恢复现场 */
```

```
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr的值恢复到cpsr里 */

swi_string:
    .string "swi exception"
*/
```

这节视频我们讲解了swi的处理流程

第006节_按键中断程序示例_概述与初始

在前面的视频里我们举了一个例子，母亲看书被声音打断，远处的声音来源有多种多样，声音传入耳朵，再由耳朵传入大脑，整个过程涉及声音来源耳朵大脑，为了确保这个母亲看书的过程能够被声音打断，我们必须保证声音来源可以发出声音，耳朵没有聋，脑袋没有傻。

类比嵌入式系统我们可以设置中断源，让他发出中断信号，还需要设置中断控制器，让他把这些信号发送给CPU，还需要设置CPU让他能够处理中断。

中断的处理流程： <1> 中断初始化：

1.1 我们需要设置中断源，让它能够发出中断喜好
1.2 设置中断控制器，让它能发出中断给CPU
1.3 设置CPU，CPSR有I位，是总开关
我们需要这样设置，中断源才能发送给CPU

<2> 处理完要清中断

<3> 处理时，要分辨中断源，对于不同的中断源要执行不同的处理函数

下面开始写代码

打开start.S 先做初始化工作，先做第 3 设置CPU，CPSR有I位，是总开关

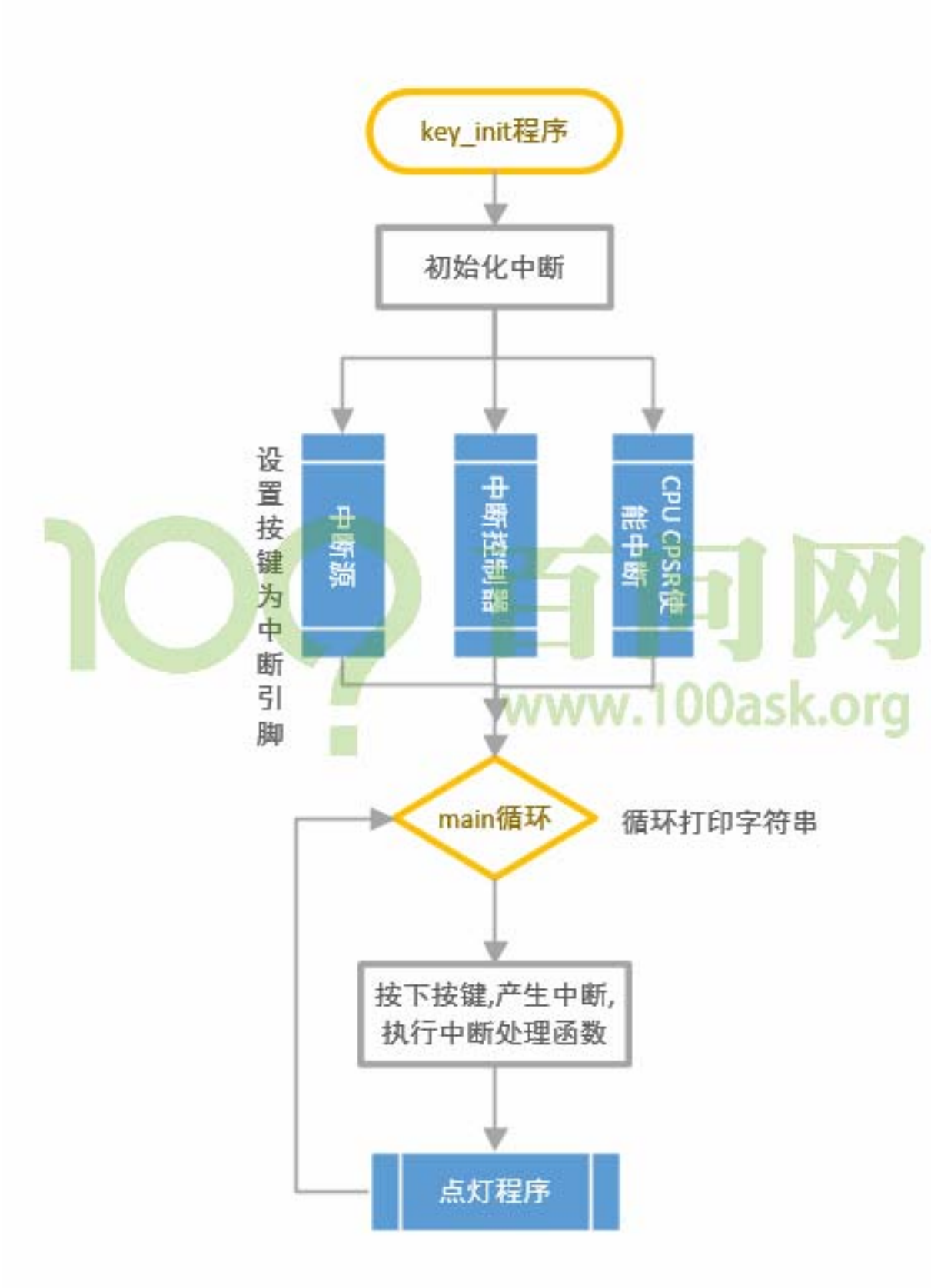
我们需要把CPSR寄存器 bit7给清零，这是中断的总开关，如果bit7设置为1，CPU无法响应任何中断

Figure 2-6. Program Status Register Format

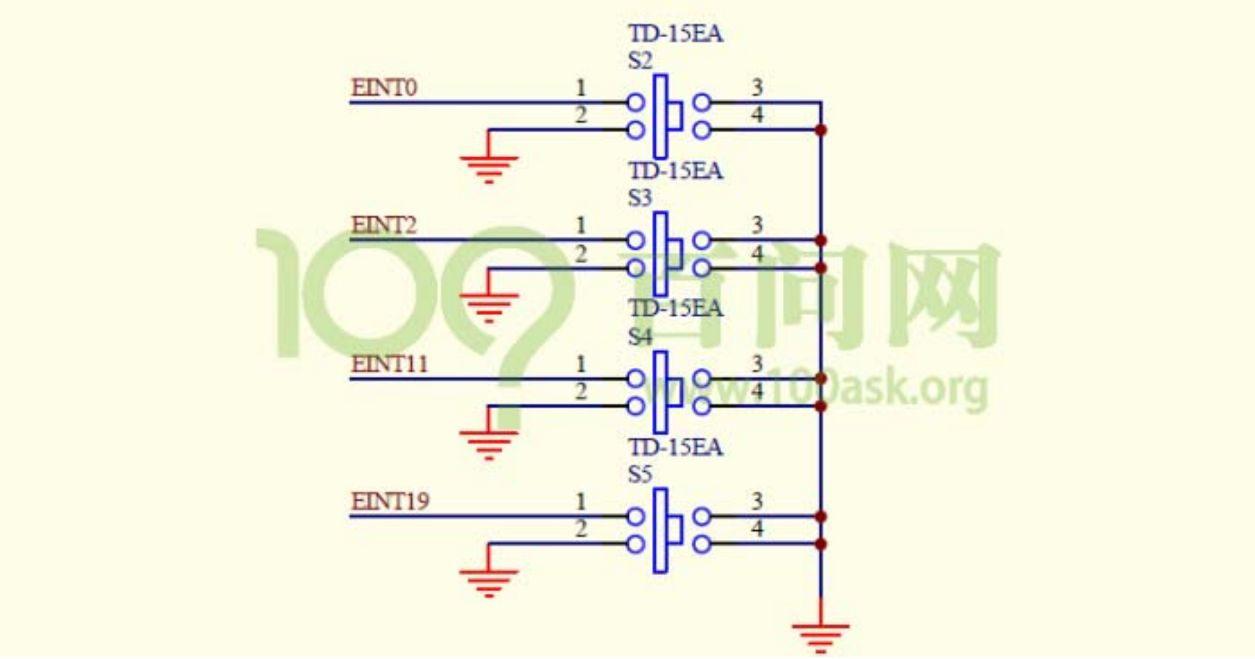
```
    /* 清除BSS段 */
    bl clean_bss

    /* 复位之后, cpu处于svc模式
     * 现在, 切换到usr模式
     */
    mrs r0, cpsr         /* 读出cpsr */
    bic r0, r0, #0xf     /* 修改M4-M0为0b10000, 进入usr模式 */

/*1
把bit7这一位清零
*/
    bic r0, r0, #(1<<7)  /* 清除I位, 使能中断 */
    msr cpsr, r0
```

```
    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    swi 0x123  /* 执行此命令，触发SWI异常，进入0x8执行 */

/*2 调用两个中断 */
    bl interrupt_init /*初始化中断控制器*/
    bl eint_init /*初始化按键，设为中断源*/

/*需要初始化上面这两个函数*/
    //bl main  /* 使用BL命令相对跳转，程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转，跳到SDRAM */

halt:
    b halt
```

添加一个 interrupt.c文件，这个程序稍微有些复杂，我们先画个流程图



我们想达到按下按键灯亮松开按键灯灭，把下面四个按键全部配置为外部中断按键



打开芯片手册找到第九章 IO ports，直接搜索“'EINT0号中断和EINT2号中断”'，找配置寄存器 GPFCON

Register	Address	R/W	Description	Reset Value
GPFCON	0x56000050	R/W	Configures the pins of port F	0x0
GPFDAT	0x56000054	R/W	The data register for port F	Undef.
GPFUP	0x56000058	R/W	Pull-up disable register for port F	0x000
Reserved	0x5600005c	–	–	–

GPFCON	Bit	Description
GPF7	[15:14]	00 = Input 01 = Output 10 = EINT[7] 11 = Reserved
GPF6	[13:12]	00 = Input 01 = Output 10 = EINT[6] 11 = Reserved
GPF5	[11:10]	00 = Input 01 = Output 10 = EINT[5] 11 = Reserved
GPF4	[9:8]	00 = Input 01 = Output 10 = EINT[4] 11 = Reserved
GPF3	[7:6]	00 = Input 01 = Output 10 = EINT[3] 11 = Reserved
GPF2	[5:4]	00 = Input 01 = Output 10 = EINT2] 11 = Reserved
GPF1	[3:2]	00 = Input 01 = Output 10 = EINT[1] 11 = Reserved
GPF0	[1:0]	00 = Input 01 = Output 10 = EINT[0] 11 = Reserved

为了简单操作

```
/* 初始化按键, 设为中断源 */
void key_eint_init(void)
{
/*1 配置GPIO为中断引脚 */
//先把eint0和eint2这两个引脚清零
GPFCON &= ~((3<<0) | (3<<4));
GPFCON |= ((2<<0) | (2<<4));    /* S2,S3被配置为中断引脚 */
```

通过电路图得知 S4 S5按键为EINT11号中断引脚和EINT19号中断引脚

Register	Address	R/W	Description	Reset Value
GPGCON	0x56000060	R/W	Configures the pins of port G	0x0
GPGDAT	0x56000064	R/W	The data register for port G	Undef.
GPGUP	0x56000068	R/W	Pull-up disable register for port G	0xfc00

GPGCON	Bit	Description
GPG15*	[31:30]	00 = Input 01 = Output 10 = EINT[23] 11 = Reserved
GPG14*	[29:28]	00 = Input 01 = Output 10 = EINT[22] 11 = Reserved
GPG13*	[27:26]	00 = Input 01 = Output 10 = EINT[21] 11 = Reserved
GPG12	[25:24]	00 = Input 01 = Output 10 = EINT[20] 11 = Reserved
GPG11	[23:22]	00 = Input 01 = Output 10 = EINT[19] 11 = TCLK[1]
GPG10	[21:20]	00 = Input 01 = Output 10 = EINT[18] 11 = nCTS1
GPG9	[19:18]	00 = Input 01 = Output 10 = EINT[17] 11 = nRTS1
GPG8	[17:16]	00 = Input 01 = Output 10 = EINT[16] 11 = Reserved
GPG7	[15:14]	00 = Input 01 = Output 10 = EINT[15] 11 = SPICLK1
GPG6	[13:12]	00 = Input 01 = Output 10 = EINT[14] 11 = SPIMOSI1
GPG5	[11:10]	00 = Input 01 = Output 10 = EINT[13] 11 = SPIMISO1
GPG4	[9:8]	00 = Input 01 = Output 10 = EINT[12] 11 = LCD_PWRDN
GPG3	[7:6]	00 = Input 01 = Output 10 = EINT[11] 11 = nSS1
GPG2	[5:4]	00 = Input 01 = Output 10 = EINT[10] 11 = nSS0
GPG1	[3:2]	00 = Input 01 = Output 10 = EINT[9] 11 = Reserved
GPG0	[1:0]	00 = Input 01 = Output 10 = EINT[8] 11 = Reserved

```
GPGCON &= ~((3<<6) | (3<<11));
GPGCON |= ((2<<6) | (2<<11));    /* S4,S5被配置为中断引脚 */
```

```
2 设置中断触发方式：(按下松开，从低电源变为高电源，或者从)双边沿触发
设置<code>EINT0 EINT2</code>为双边沿触发

 EXTINT0 |= (7<<0) | (7<<8);       /* S2,S3 */
```

Register	Address	R/W	Description	Reset Value
EXTINT0	0x56000088	R/W	External interrupt control register 0	0x000000
EXTINT1	0x5600008c	R/W	External interrupt control register 1	0x000000
EXTINT2	0x56000090	R/W	External interrupt control register 2	0x000000

EXTINT0	Bit	Description
EINT7	[30:28]	Setting the signaling method of the EINT7. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT6	[26:24]	Setting the signaling method of the EINT6. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT5	[22:20]	Setting the signaling method of the EINT5. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT4	[18:16]	Setting the signaling method of the EINT4. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT3	[14:12]	Setting the signaling method of the EINT3. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT2	[10:8]	Setting the signaling method of the EINT2. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT1	[6:4]	Setting the signaling method of the EINT1. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
EINT0	[2:0]	Setting the signaling method of the EINT0. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

设置EINT11为双边沿触发

```
EXTINT1 |= (7<<12);              /* S4 */
```

设置EINT19为双边沿触发

EXTINT2	Bit	Description	Reset Value
FLTEN23	[31]	Filter enable for EINT23 0 = Filter Disable 1= Filter Enable	0
EINT23	[30:28]	Setting the signaling method of the EINT23. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN22	[27]	Filter Enable for EINT22 0 = Filter Disable 1= Filter Enable	0
EINT22	[26:24]	Setting the signaling method of the EINT22. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN21	[23]	Filter Enable for EINT21 0 = Filter Disable 1= Filter Enable	0
EINT21	[22:20]	Setting the signaling method of the EINT21. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN20	[19]	Filter Enable for EINT20 0 = Filter Disable 1= Filter Enable	0
EINT20	[18:16]	Setting the signaling method of the EINT20. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN19	[15]	Filter enable for EINT19 0 = Filter Disable 1= Filter Enable	0
EINT19	[14:12]	Setting the signaling method of the EINT19. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000
FLTEN18	[11]	Filter enable for EINT18 0 = Filter Disable 1= Filter Enable	0
EINT18	[10:8]	Setting the signaling method of the EINT18. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered	000

```
EXTINT2 |= (7<<12);                /* S5 */
```

EXTINT1	Bit	Description
FLTEN15	[31]	Filter enable for EINT15 0 = Filter Disable 1 = Filter Enable
EINT15	[30:28]	Setting the signaling method of the EINT15. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN14	[27]	Filter enable for EINT14 0 = Filter Disable 1 = Filter Enable
EINT14	[26:24]	Setting the signaling method of the EINT14. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN13	[23]	Filter enable for EINT13 0 = Filter Disable 1 = Filter Enable
EINT13	[22:20]	Setting the signaling method of the EINT13. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN12	[19]	Filter enable for EINT12 0 = Filter Disable 1 = Filter Enable
EINT12	[18:16]	Setting the signaling method of the EINT12. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN11	[15]	Filter enable for EINT11 0 = Filter Disable 1 = Filter Enable
EINT11	[14:12]	Setting the signaling method of the EINT11. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN10	[11]	Filter enable for EINT10 0 = Filter Disable 1 = Filter Enable
EINT10	[10:8]	Setting the signaling method of the EINT10. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN9	[7]	Filter enable for EINT9 0 = Filter Disable 1 = Filter Enable
EINT9	[6:4]	Setting the signaling method of the EINT9. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered
FLTEN8	[3]	Filter enable for EINT8 0 = Filter Disable 1 = Filter Enable
EINT8	[2:0]	Setting the signaling method of the EINT8. 000 = Low level 001 = High level 01x = Falling edge triggered 10x = Rising edge triggered 11x = Both edge triggered

外部中断屏蔽寄存器EINTMASK，设置为1的话就禁止向外部发出中断信号，只有EINTMASK相应的位设置为0外部中断才能给中断控制器发信号

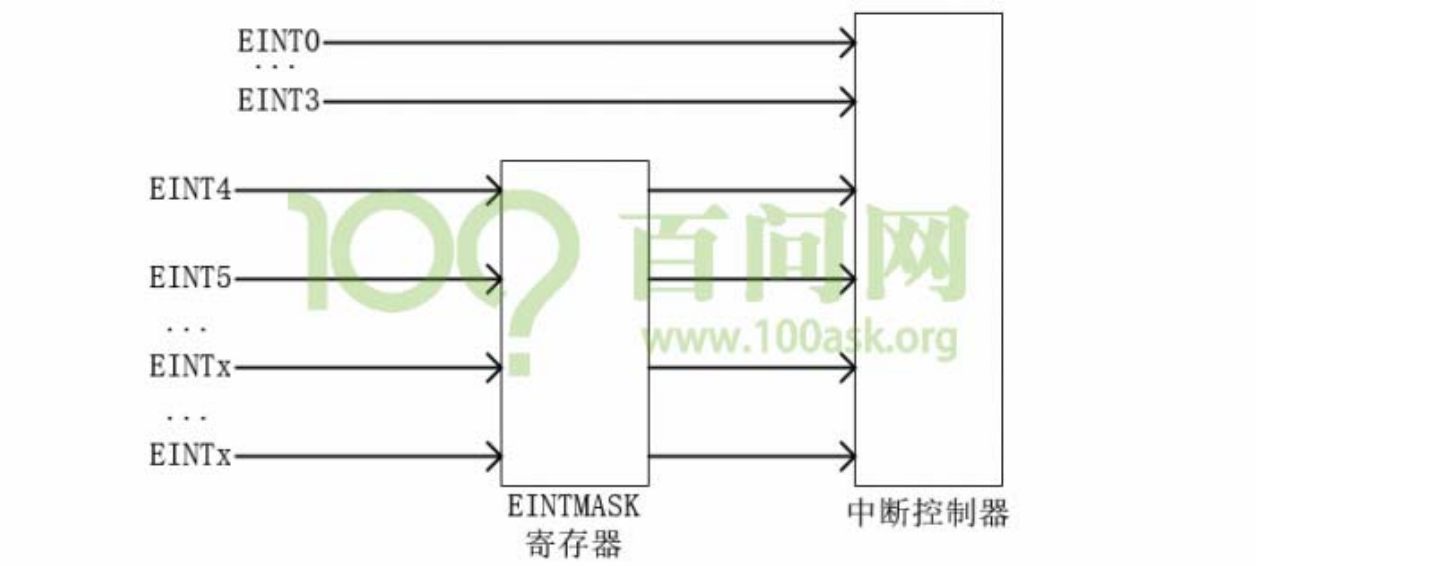
我们需要设置这个寄存器

EINTMASK (External Interrupt Mask Register)

Register	Address	R/W	Description	Reset Value
EINTMASK	0x560000a4	R/W	External interrupt mask register	0x000fffff

EINTMASK	Bit	Description
EINT23	[23]	0 = enable interrupt 1= masked
EINT22	[22]	0 = enable interrupt 1= masked
EINT21	[21]	0 = enable interrupt 1= masked
EINT20	[20]	0 = enable interrupt 1= masked
EINT19	[19]	0 = enable interrupt 1= masked
EINT18	[18]	0 = enable interrupt 1= masked
EINT17	[17]	0 = enable interrupt 1= masked
EINT16	[16]	0 = enable interrupt 1= masked
EINT15	[15]	0 = enable interrupt 1= masked
EINT14	[14]	0 = enable interrupt 1= masked
EINT13	[13]	0 = enable interrupt 1= masked
EINT12	[12]	0 = enable interrupt 1= masked
EINT11	[11]	0 = enable interrupt 1= masked
EINT10	[10]	0 = enable interrupt 1= masked
EINT9	[9]	0 = enable interrupt 1= masked
EINT8	[8]	0 = enable interrupt 1= masked
EINT7	[7]	0 = enable interrupt 1= masked
EINT6	[6]	0 = enable interrupt 1= masked
EINT5	[5]	0 = enable interrupt 1= masked
EINT4	[4]	0 = enable interrupt 1= masked
Reserved	[3:0]	Reserved

把EINT11设置为0 把EINT19设置为0对于EINT0 和EINT2显示为保留，默认时使能的，可以直接发送给中断控制器，无需设置



设置EINTMASK使能eint11,19

```
EINTMASK &= ~((1<<11) | (1<<19));
```

读EINTPEND分辨率哪个EINT产生(eint4~23)(并且要清除它)

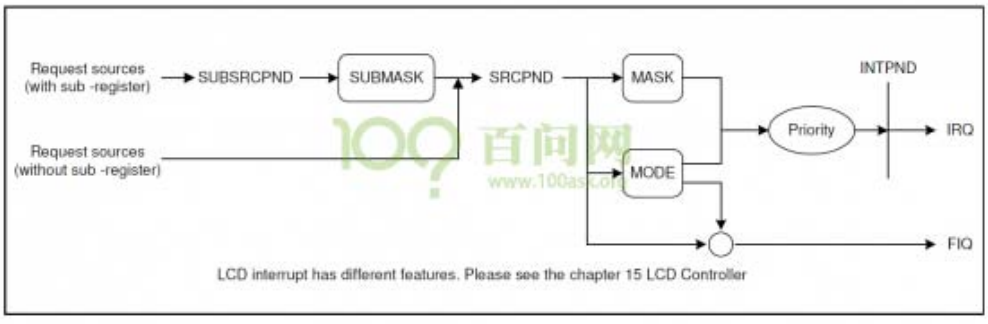
清除中断时, 写EINTPEND的相应位

EINTPEND (External Interrupt Pending Register)

Register	Address	R/W	Description	Reset Value
EINTPEND	0x560000a8	R/W	External interrupt pending register	0x00

EINTPEND	Bit	Description	Reset Value
EINT23	[23]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT22	[22]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT21	[21]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT20	[20]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT19	[19]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT18	[18]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT17	[17]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT16	[16]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT15	[15]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT14	[14]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT13	[13]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT12	[12]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT11	[11]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT10	[10]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT9	[9]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT8	[8]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT7	[7]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT6	[6]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT5	[5]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT4	[4]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
Reserved	[3:0]	Reserved	0000

我们接下来需要阅读'**第14章 Interrupt Controller章节**设置中断控制器我们只需要按照下面这张流程图设置就可以了



我们需要设置

MASK 屏蔽寄存器
INTPND 等待处理，我们可以读这个寄存器，确定是那个中断产生了
SRCPND不同的中断类型不可以直接到达这里执行

我们来看一下外部中断属于哪一种

打开芯片手册，从上往下读

INTERRUPT SOURCES

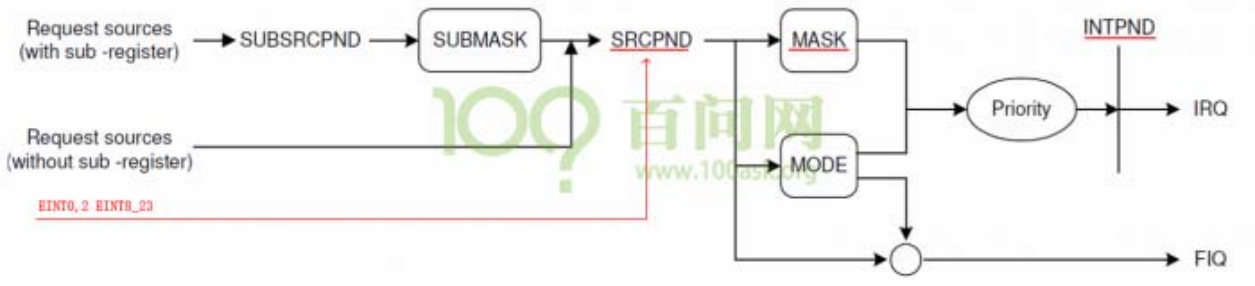
The interrupt controller supports 60 interrupt sources as shown in the table below.

Sources	Descriptions	Arbiter Group
INT_ADC	ADC EOC and Touch interrupt (INT_ADC_S/INT_TC)	ARB5
INT_RTC	RTC alarm interrupt	ARB5
INT_SPI1	SPI1 interrupt	ARB5
INT_UART0	UART0 Interrupt (ERR, RXD, and TXD)	ARB5
INT_IIC	IIC interrupt	ARB4
INT_USBH	USB Host interrupt	ARB4
INT_USBD	USB Device interrupt	ARB4
INT_NFCON	Nand Flash Control Interrupt	ARB4
INT_UART1	UART1 Interrupt (ERR, RXD, and TXD)	ARB4
INT_SPI0	SPI0 interrupt	ARB4
INT_SDI	SDI interrupt	ARB 3
INT_DMA3	DMA channel 3 interrupt	ARB3
INT_DMA2	DMA channel 2 interrupt	ARB3
INT_DMA1	DMA channel 1 interrupt	ARB3
INT_DMA0	DMA channel 0 interrupt	ARB3
INT_LCD	LCD interrupt (INT_FrSyn and INT_FiCnt)	ARB3
INT_UART2	UART2 Interrupt (ERR, RXD, and TXD)	ARB2
INT_TIMER4	Timer4 interrupt	ARB2
INT_TIMER3	Timer3 interrupt	ARB2
INT_TIMER2	Timer2 interrupt	ARB2
INT_TIMER1	Timer1 interrupt	ARB 2
INT_TIMER0	Timer0 interrupt	ARB2
INT_WDT_AC97	Watch-Dog timer interrupt(INT_WDT, INT_AC97)	ARB1
INT_TICK	RTC Time tick interrupt	ARB1
nBATT_FLT	Battery Fault interrupt	ARB1
INT_CAM	Camera Interface (INT_CAM_C, INT_CAM_P)	ARB1
EINT8_23	External interrupt 8 – 23	ARB1
EINT4_7	External interrupt 4 – 7	ARB1
EINT3	External interrupt 3	ARB0
EINT2	External interrupt 2	ARB0
EINT1	External interrupt 1	ARB0
EINT0	External interrupt 0	ARB0

由上图可得 EINT4_7 EINT8_23合用一条中断线**ARB1**

也就是可以直接到达SRCPND不需要设置SUBSRCPND和SUBMASK这两个寄存器

我们使用的外部中断源只需要设置SRCPND MASK INTPND这三个就可以




```
/* SRCPND 用来显示哪个中断产生了，需要清除对应位，我们只需要关心
 * bit0对应eint0
 * bit2对应eint2
 * bit5对应eint8_23(表明bit5等于1的时候 eint8_23中的某一个已经产生，我们需要继续分辨
 * 读EINTPEND分辨率哪个EINT产生)
 */
```

INTMOD寄存器 默认值为IRQ模式即可，不需要设置

Register	Address	R/W	Description	Reset Value
INTMOD	0X4A000004	R/W	Interrupt mode register. 0 = IRQ mode 1 = FIQ mode	0x00000000

INTMASK寄存器，需要设置为0

Register	Address	R/W	Description	Reset Value
INTMSK	0X4A000008	R/W	Determine which interrupt source is masked. The masked interrupt source will not be serviced. 0 = Interrupt service is available. 1 = Interrupt service is masked.	0xFFFFFFFF

```
/* INTMSK 用来屏蔽中断, 1对应masked屏蔽中断，我们需要设置相应位设置为0
 * bit0-eint0
 * bit2-eint2
 * bit5-eint8_23
 */
```

同时可能有多个中断产生，这么多个中断经过优先级以后，只会有一个通知CPU，是哪一个中断优先级最高，可以读INTPAD就能知道当前处理的唯 一 一个中断是那一个

Register	Address	R/W	Description	Reset Value
INTPND	0X4A000010	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

1 表示这个中断已经产生，需要配置相应的位
INTPND 用来显示当前优先级最高的、正在发生的中断, 需要清除对应位

```
bit0-eint0
bit2-eint2
bit5-eint8_23
```

INTOFFSET是用来显示INTPND寄存器中哪一位正在等待处理

Register	Address	R/W	Description	Reset Value
INTOFFSET	0x4A000014	R	Indicate the IRQ interrupt request source	0x00000000

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_AC97	9
INT_NFCON	24	INT_TICK	8
INT_UART1	23	nBATT_FLT	7
INT_SPI0	22	INT_CAM	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

```
INTPAD中bit0等于1的话INTOFFSET就等于0
INTPAD中bit1等于1的话INTOFFSET值就等于1
```

INTOFFSET ： 用来显示INTPND中哪一位被设置为1

SRCPND我们用不到

Register	Address	R/W	Description	Reset Value
SUBSRCPND	0X4A000018	R/W	Indicate the interrupt request status. 0 = The interrupt has not been requested. 1 = The interrupt source has asserted the interrupt request.	0x00000000

SUBSRCPND	Bit	Description	Initial State
Reserved	[31:15]	Not used	0
INT_AC97	[14]	0 = Not requested, 1 = Requested	0
INT_WDT	[13]	0 = Not requested, 1 = Requested	0
INT_CAM_P	[12]	0 = Not requested, 1 = Requested	0
INT_CAM_C	[11]	0 = Not requested, 1 = Requested	0
INT_ADC_S	[10]	0 = Not requested, 1 = Requested	0
INT_TC	[9]	0 = Not requested, 1 = Requested	0
INT_ERR2	[8]	0 = Not requested, 1 = Requested	0
INT_TXD2	[7]	0 = Not requested, 1 = Requested	0
INT_RXD2	[6]	0 = Not requested, 1 = Requested	0
INT_ERR1	[5]	0 = Not requested, 1 = Requested	0
INT_TXD1	[4]	0 = Not requested, 1 = Requested	0
INT_RXD1	[3]	0 = Not requested, 1 = Requested	0
INT_ERR0	[2]	0 = Not requested, 1 = Requested	0
INT_TXD0	[1]	0 = Not requested, 1 = Requested	0
INT_RXD0	[0]	0 = Not requested, 1 = Requested	0

Map To SRCPND

SRCPND	SUBSRCPND	Remark
INT_UART0	INT_RXD0,INT_TXD0,INT_ERR0	
INT_UART1	INT_RXD1,INT_TXD1,INT_ERR1	
INT_UART2	INT_RXD2,INT_TXD2,INT_ERR2	
INT_ADC	INT_ADC_S, INT_TC	
INT_CAM	INT_CAM_C, INT_CAM_P	
INT_WDT_AC97	INT_WDT, INT_AC97	

某一位等于1时INT_UART0它的来源可能有多个，这是串口0的中断，串口0的中断产生时有可能是接收到了数据(INT_RXD0),有可能是发送了数据(INT_TXD0),也有可能是产生了错误，那么到底是哪一个呢？

需要去读取SUBSRCPND下一级的源寄存器

我们只需要设置INTMSK这个寄存器

SRCPND和INTPND只有发生中断才需要设置

```
/* 初始化中断控制器 */
void interrupt_init(void)
{
    //1是屏蔽我们需要清零，外部中断0 外部中断2 外部中8_23里面还有外部中断11到19
    INTMSK &= ~((1<<0) | (1<<2) | (1<<5));

}
```

修改start.S删除

```
bl interrupt_init  /* 初始化中断控制器 */
bl key_eint_init  /* 初始化按键，设为中断源 */
```

能使用c语言就使用C语言，在main.c文件中添加调用C函数：

```
int main(void)
{
/************1/
    interrupt_init();  /* 初始化中断控制器 */
    key_eint_init();   /* 初始化按键, 设为中断源 */
******/
    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");
```

第007节_按键中断程序示例_完善

首先main.c中 我们初始化中断控制器 初始化中断源

假设按键按键就会产生中断，CPU就会跳到start.S 执行

```
_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
    ldr pc, swi_addr /* vector 8 : swi */
```

具体跳到哪里执行，我们需要看看中断向量表在哪里

Table 2-3. Exception Vectors

Address	Exception	Mode in Entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software Interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

IRQ模式的话跳到0x00000018地方

```
b halt          /* vector 0x0c : prefetch aboot */
b halt          /* vector 0x10 : data abort */
b halt          /* vector 0x14 : reserved */
ldr pc, irq_addr /* vector 0x18 : irq */
b halt          /* vector 0x1c : fiq */

/*3/
do_irq:
    /* 执行到这里之前:
     * 1. lr_irq保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10010, 进入到irq模式
     * 4. 跳到0x18的地方执行程序
     */

    /* sp_irq未设置, 先设置它 */

/* 4 分配不冲突的没有使用的内存就可以了*/
    ldr sp, =0x33d00000

/*5*/
    /* 保存现场 */

/*7
发生中断时irq返回值是R14 -4 为什么要减去4, 硬件结构让你怎么做就怎么做
*/
    /* 在irq异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr-4是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
```

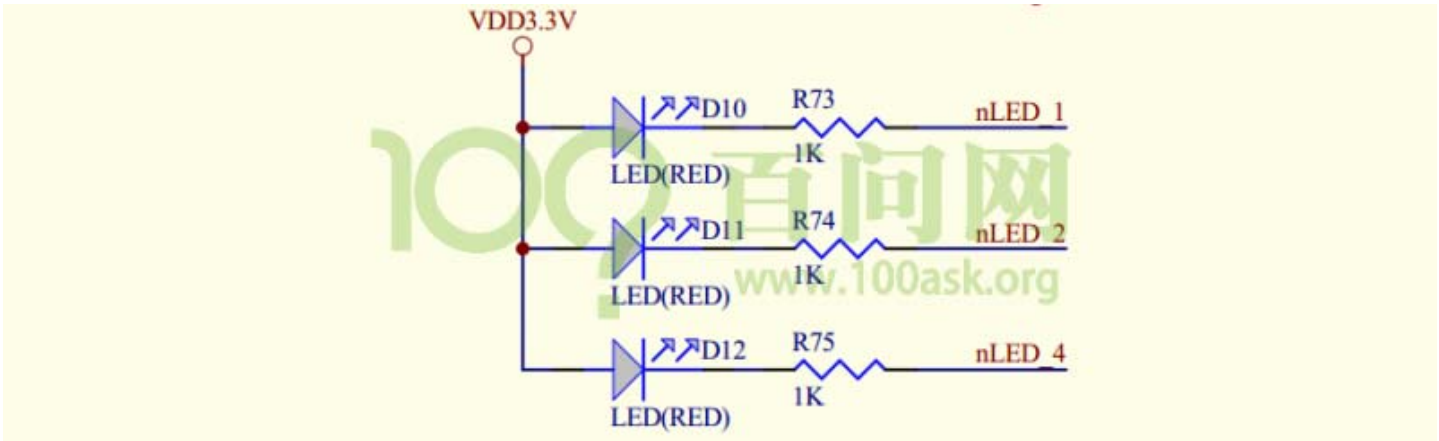
```
    stmdb sp!, {r0-r12, lr}

    /* 处理irq异常 */
/*6
在这C函数里分辨中断源，处理中断
*/
    bl handle_irq_c

/*8*/
    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr_irq的值恢复到cpsr里 */
```

接下来我们在interrupt.c中写出 handle_irq_c处理函数 这个是处理中断的C函数

```
void handle_irq_c(void)
{
    /*1 分辨中断源 */
    /*读INTOFFSET在芯片手册里找到这个寄存器，它里面的值表示INTPND中哪一位被设置成1*/
    int bit = INTOFFSET;

    /*2 调用对应的处理函数 */
    if (bit == 0 || bit == 2 || bit == 5)  /* 对应eint0,2,eint8_23 */
    {
        /*我们会调用一个按键处理函数*/
        key_eint_irq(bit); /* 处理中断, 清中断源EINTPEND */
    }

    /*3 清中断 : 从源头开始清
     *先清除掉中断源里面的某些寄存器
     *再清 SRCPND
     *再清 INTPND
     */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}
```

读EINTPEND分辨率哪个EINT产生(eint4~23)清除中断时, 写EINTPEND的相应位

```
void key_eint_irq(int irq)
{
    /**4清的时候我先把这个值读出来，清的时候我再把他写进去/
    unsigned int val = EINTPEND;

    unsigned int val1 = GPFDAT;
    unsigned int val2 = GPGDAT;

    if (irq == 0) /*1 外部中断eint0对应s2按键  */
    {
```



我们使用s2来控制那盏灯？

- 之前我们写过按键控制led灯的程序，它使用的是s2控制gpf6
- 也就是s2控制led4 D12


```
    */
    /****1.2如何知道是按下还是松开，我们需要读值*/
    if (val1 & (1<<0)) /* s2 --> gpf6 */
    {
        /*1.3 松开 */
        GPFDAT |= (1<<6);
    }
    else
    {
        /*1.4 按下 */
        GPFDAT &= ~(1<<6);
    }

}
else if (irq == 2) /*2 eint2对应s3 控制 D11  LED2 */
{
    if (val1 & (1<<2)) /* s3 --> gpf5 */
    {
        /* 松开 */
        GPFDAT |= (1<<5);
    }
    else
    {
        /* 按下 */
        GPFDAT &= ~(1<<5);
    }

}
else if (irq == 5) /*3 eint8_23, eint11对应s4 控制 D10 LED1, eint19对应s5 控制所有LED */
{
```

到底是发生哪一种中断，我们需要读取 EINTPND来判断是那个中断产生

EINTPEND (External Interrupt Pending Register)

Register	Address	R/W	Description	Reset Value
EINTPEND	0x560000a8	R/W	External interrupt pending register	0x00

EINTPEND	Bit	Description	Reset Value
EINT23	[23]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT22	[22]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT21	[21]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT20	[20]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT19	[19]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT18	[18]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT17	[17]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT16	[16]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT15	[15]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT14	[14]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT13	[13]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT12	[12]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT11	[11]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT10	[10]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT9	[9]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT8	[8]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT7	[7]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT6	[6]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT5	[5]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
EINT4	[4]	It is cleard by writing "1" 0 = Not occur 1 = Occur interrupt	0
Reserved	[3:0]	Reserved	0000

如果bit19等于1的话表明外部中断EINT19产生了，如果bit11等于1表用外部中断11产生，这里我们需要判断

```
    if (val & (1<<11)) /* 表明外部中断eint11产生 */
    {
        if (val2 & (1<<3)) /* s4 --> gpf4 */
        {
            /* 松开 */
            GPFDAT |= (1<<4);
        }
        else
        {
            /* 按下 */
            GPFDAT &= ~(1<<4);
        }
    }
    else if (val & (1<<19)) /* 表用外部中断eint19 */
    {
        if (val2 & (1<<11))
        {
            /* 松开 */
            /* 熄灭所有LED 输出高电平 */
            GPFDAT |= ((1<<4) | (1<<5) | (1<<6));
        }
        else
        {
            /* 按下：点亮所有LED */
```

```
                GPFDAT &= ~((1<<4) | (1<<5) | (1<<6));
            }
        }
    }
    /**5 再把值写进去就达到了清除的效果*/
    EINTPEND = val;
}
```

上传代码测试

我们需要包含头文件

```
#include "S3c2440_soc.h"
```

编译通过，开发板上电测试发现按键s5无法控制 查看 interrupt.c文件中的按键初始化

```
/* 初始化按键, 设为中断源 */
void key_eint_init(void)
{
    /* 配置GPIO为中断引脚 */
    GPFCON &= ~((3<<0) | (3<<4));
    GPFCON |= ((2<<0) | (2<<4));    /* S2,S3被配置为中断引脚 */
        /*发现外部中断19的bit位配置不正确应该是22*/
    GPGCON &= ~((3<<6) | (3<<22));
    GPGCON |= ((2<<6) | (2<<22));    /* S4,S5被配置为中断引脚 */
```

上传代码从新编译执行 重新烧写看是否可以使用 回顾中断处理流程

我们start.s 一上电从 _start: 运行 做一些初始化工作

```
reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */



    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
```

```
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    /* 复位之后, cpu处于svc模式
     * 现在, 切换到usr模式
     */
    mrs r0, cpsr         /* 读出cpsr */
    bic r0, r0, #0xf     /* 修改M4-M0为0b10000, 进入usr模式 */
    bic r0, r0, #(1<<7)  /* 清除I位, 使能中断 */
    msr cpsr, r0

    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    swi 0x123  /* 执行此命令, 触发SWI异常, 进入0x8执行 */

    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt

//让后设置CPSR开中断
//让后调到mian函数，做一些中断初始化

int main(void)
{
    led_init();
    interrupt_init();  /* 初始化中断控制器 */
    key_eint_init();   /* 初始化按键, 设为中断源 */

    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");
/*让后在main函数里一直循环输出串口*/
    while (1)
    {

        putchar(g_Char);
        g_Char++;

        putchar(g_Char3);
        g_Char3++;
        delay(1000000);
    }

//这个时候按下按键就会产生中断，让后进入start.s
//跳到0x18 irq模式

ldr pc, irq_addr /* vector 0x18 : irq */
```

它是一条读内存的执行，从这里读地址赋给pc

```
irq_addr:
    .word do_irq
```

就跳到sdram执行do_irq函数

```
do_irq:
```

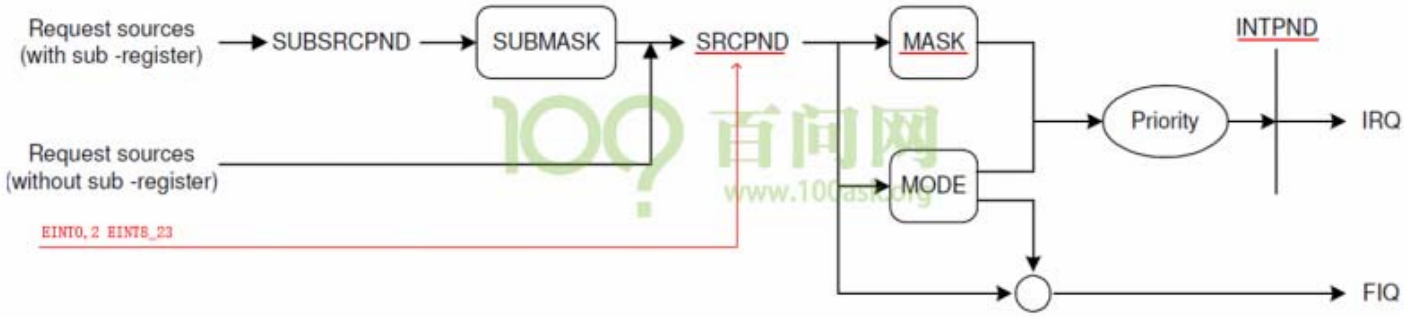
```
    /* 执行到这里之前:
     * 1. lr_irq保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10010, 进入到irq模式
     * 4. 跳到0x18的地方执行程序
     */

    /* sp_irq未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */
    /* 在irq异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr-4是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理irq异常 */
    bl handle_irq_c

    /* 恢复现场 */
```

它怎么处理

```
/* 读EINTPEND分辨率哪个EINT产生(eint4~23)
 * 清除中断时, 写EINTPEND的相应位
 */


void key_eint_irq(int irq)
{
    unsigned int val = EINTPEND;
    unsigned int val1 = GPFDAT;
    unsigned int val2 = GPGDAT;

    if (irq == 0) /* eint0 : s2 控制 D12 */
    {
        if (val1 & (1<<0)) /* s2 --> gpf6 */
        {
            /* 松开 */
            GPFDAT |= (1<<6);
        }
        else
        {
            /* 按下 */
            GPFDAT &= ~(1<<6);
        }

    }
    else if (irq == 2) /* eint2 : s3 控制 D11 */
    {
        if (val1 & (1<<2)) /* s3 --> gpf5 */
        {
            /* 松开 */
            GPFDAT |= (1<<5);
        }
        else
        {
            /* 按下 */
            GPFDAT &= ~(1<<5);
        }

    }
    else if (irq == 5) /* eint8_23, eint11--s4 控制 D10, eint19---s5 控制所有LED */
    {
        if (val & (1<<11)) /* eint11 */
        {
            if (val2 & (1<<3)) /* s4 --> gpf4 */
            {
                /* 松开 */
                GPFDAT |= (1<<4);
            }
            else
            {
                /* 按下 */
                GPFDAT &= ~(1<<4);
            }
        }
        else if (val & (1<<19)) /* eint19 */
        {
            if (val2 & (1<<11))
            {
                /* 松开 */
                /* 熄灭所有LED */
                GPFDAT |= ((1<<4) | (1<<5) | (1<<6));
```

```
            }
            else
            {
                /* 按下: 点亮所有LED */
                GPFDAT &= ~((1<<4) | (1<<5) | (1<<6));
            }
        }
    }


    EINTPEND = val;
}
```

处理完之后清中断，从源头开始清 这完全是按照中断流程操作的



第008节_定时器中断程序示例

这节课我们来写一个定时器的中断服务程序 使用定时器来实现点灯计数 查考资料就是**第10章PWM TIMER** 可以参考书籍《嵌入式Linux应用程序开发完全手册》第10章

我们先把这个结构图展示出来

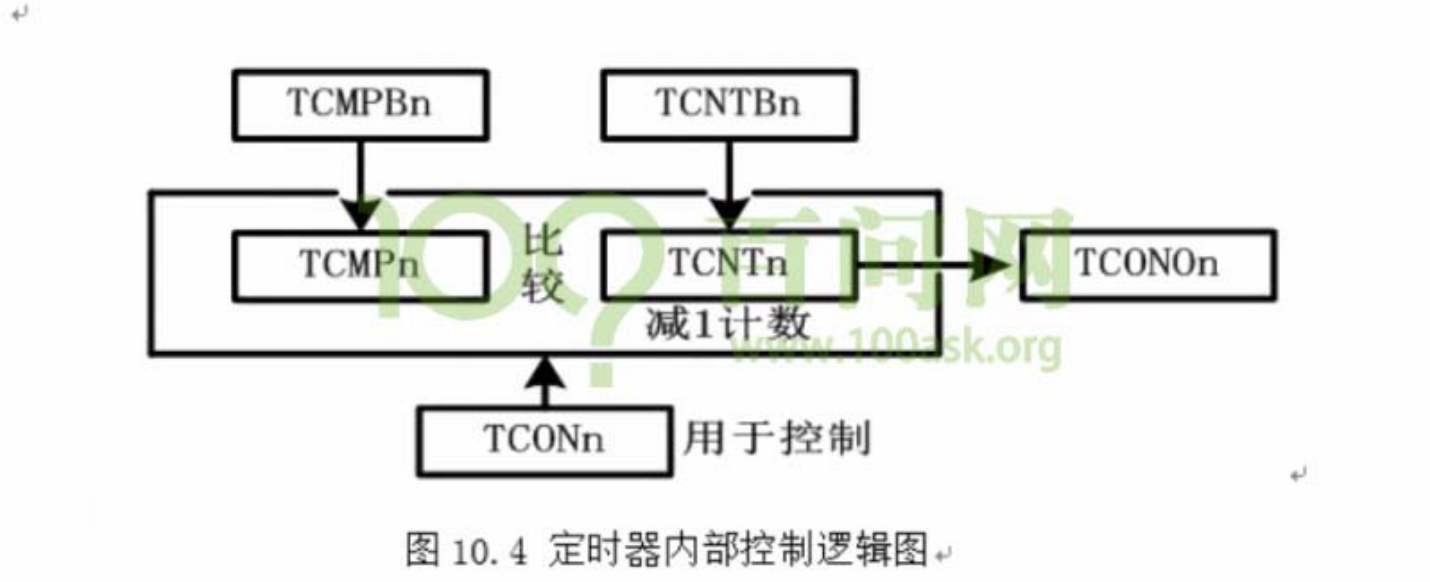


图 10.4 定时器内部控制逻辑图

这个图的结构很好

这里面肯定有一个clk(时钟)， 1 每来一个clk(时钟)这个TCNTn减去1 2 当TCNTn == TCMPn时，可以产生中断，也可以让对应的SPWM引脚反转，(比如说原来是高电平，发生之后电平转换成低电平) 3 TCNTn继续减1，当TCNTn == 0时，可以产生中断，pwm引脚再次反转 TCMPn 和 TCNTn的初始值来自 TCMPBn,TCNTBn 4 TCNTn == 0时，可自动加载初始

怎么使用定时器？

```
1 设置时钟
2 设置初值
3 加载初始，启动Timer
4 设置为自动加载
5 中断相关
```

由于2440没有引出pwm引脚，所以pwm功能无法使用，也就无法做pwm相关实验，所谓pwm是指可调制脉冲

T1高脉冲和T2低脉冲它的时间T1, T2可调整，可以输出不同频率不同占控比的波形，在控制电机时特别有用

我们这个程序只做一个实验，当TCNTn这个计数器到0的时候，就产生中断，在这个中断服务程序里我们点灯

写代码 打开我们的main函数

```
int main(void)
{
    led_init();
    interrupt_init();  /* 初始化中断控制器 */
//我们初始化了中断源，同样的，我们初始化timer
    key_eint_init();   /* 初始化按键，设为中断源 */
//初始化定时器
    timer_init();
```

我们需要实现定时器初始化函数

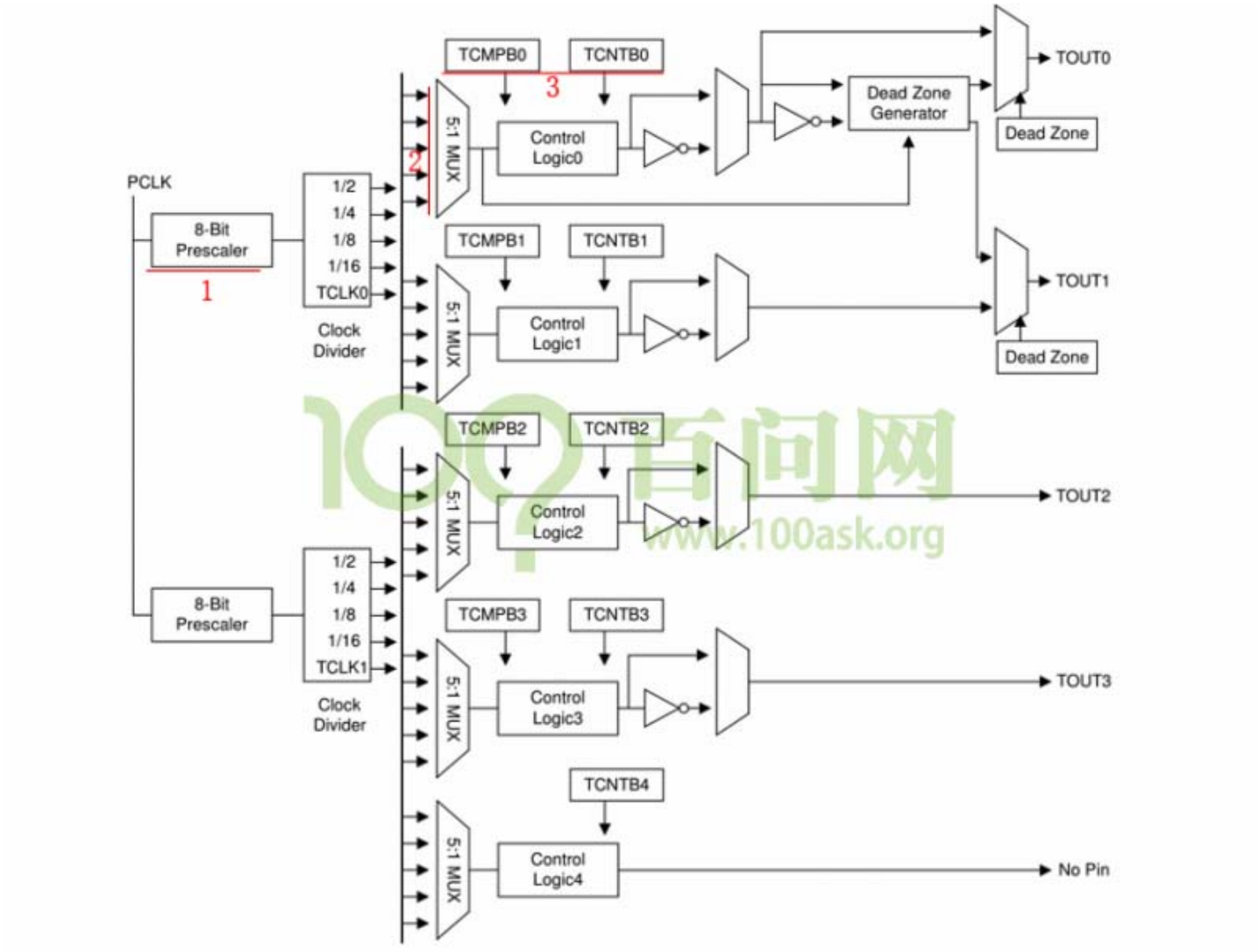
新建一个 timer.c，我们肯定需要操作一堆寄存器，添加头文件 #include "s3c2440_soc.h"

void timer_init(void)

1. 设置TIMER0的时钟
2. 设置TIMER0的初值
3. 加载初值, 启动timer0
4. 设置为自动加载并启动(值到0以后会自动加载)
5. 设置中断，显然我们需要提供一个中断处理函数void timer_irq(void)在这里面我们需要点灯

打开芯片手册，我们想设置timer0的话

1. 首先设置8-Bit Prescaler
2. 设置5.1 MUX(选择一个时钟分频)
3. 设置TCMPB 0 和TCNTB0
4. 设置TCONn寄存器



看手册上写如何初始化timer

TIMER INITIALIZATION USING MANUAL UPDATE BIT AND INVERTER BIT

An auto reload operation of the timer occurs when the down counter reaches 0. So, a starting value of the TCNTn has to be defined by the user in advance. In this case, the starting value has to be loaded by the manual update bit. The following steps describe how to start a timer:

1) Write the initial value into TCNTBn and TCMPBn.
2) Set the manual update bit of the corresponding timer. It is recommended that you configure the inverter on/off bit. (Whether use inverter or not).
3) Set start bit of the corresponding timer to start the timer (and clear the manual update bit).

If the timer is stopped by force, the TCNTn retains the counter value and is not reloaded from TCNTBn. If a new value has to be set, perform manual update.

1. 把初始值写到TCNTBn 和TCMPBn寄存器
2. 设置手动更新位
3. 设置启动位

往下看到时钟配置寄存器

TIMER CONFIGURATION REGISTER0 (TCFG0)

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

Register	Address	R/W	Description	Reset Value
TCFG0	0x51000000	R/W	Configures the two 8-bit prescalers	0x00000000

TCFG0	Bit	Description	Initial State
Reserved	[31:24]		0x00
Dead zone length	[23:16]	These 8 bits determine the dead zone length. The 1 unit time of the dead zone length is equal to that of timer 0.	0x00
Prescaler 1	[15:8]	These 8 bits determine prescaler value for Timer 2, 3 and 4.	0x00
Prescaler 0	[7:0]	These 8 bits determine prescaler value for Timer 0 and 1.	0x00

有个计算公式

```
Timer clk = PCLK / {(预分频数)prescaler value+1} / {divider value(5.1MUX值)}
```

PCLK是50M

```
= 50000000/(99+1)/16
 = 31250
```

也就是说我们得TCON是31250的话，从这个值一直减到0 Prescaler0等于99

```
TCFG0 = 99;  /* Prescaler 0 = 99, 用于timer0,1 */
```

TCFG1 MUX多路复用器的意思，他有多路输入，我们可以通过MUX选择其中一路作为输出

TIMER CONFIGURATION REGISTER1 (TCFG1)

Register	Address	R/W	Description	Reset Value
TCFG1	0x51000004	R/W	5-MUX & DMA mode selection register	0x00000000

TCFG1	Bit	Description	Initial State
Reserved	[31:24]		00000000
DMA mode	[23:20]	Select DMA request channel 0000 = No select (all interrupt) 0001 = Timer0 0010 = Timer1 0011 = Timer2 0100 = Timer3 0101 = Timer4 0110 = Reserved	0000
MUX 4	[19:16]	Select MUX input for PWM Timer4. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 3	[15:12]	Select MUX input for PWM Timer3. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 2	[11:8]	Select MUX input for PWM Timer2. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK1	0000
MUX 1	[7:4]	Select MUX input for PWM Timer1. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000
MUX 0	[3:0]	Select MUX input for PWM Timer0. 0000 = 1/2 0001 = 1/4 0010 = 1/8 0011 = 1/16 01xx = External TCLK0	0000

根据上面mux的值，我们要把MUX0 设置成0011 只需要设置这4位即可，先清零 再或上 0011 就是3

```
TCFG1 &= ~0xf;
TCFG1 |= 3;  /* MUX0  : 1/16 */
```

再来看看初始值控制寄存器

TCNTB0	Bit	Description	Initial State
Timer 0 count buffer register	[15:0]	Set count buffer value for Timer 0	0x00000000

一秒钟点灯太慢了 ，就让0.5秒

```
TCNTB0 = 15625;  /* 0.5s中断一次 */
```

这个寄存器是用来观察里面的计数值的，不需要设置

现在可以设置TCON来设置这个寄存器了

TIMER CONTROL (TCON) REGISTER

Register	Address	R/W	Description	Reset Value
TCON	0x51000008	R/W	Timer control register	0x00000000

TCON	Bit	Description	Initial state
Timer 4 auto reload on/off	[22]	Determine auto reload on/off for Timer 4. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 4 manual update (note)	[21]	Determine the manual update for Timer 4. 0 = No operation 1 = Update TCNTB4	0
Timer 4 start/stop	[20]	Determine start/stop for Timer 4. 0 = Stop 1 = Start for Timer 4	0
Timer 3 auto reload on/off	[19]	Determine auto reload on/off for Timer 3. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 3 output inverter on/off	[18]	Determine output inverter on/off for Timer 3. 0 = Inverter off 1 = Inverter on for TOUT3	0
Timer 3 manual update (note)	[17]	Determine manual update for Timer 3. 0 = No operation 1 = Update TCNTB3 & TCMPB3	0
Timer 3 start/stop	[16]	Determine start/stop for Timer 3. 0 = Stop 1 = Start for Timer 3	0
Timer 2 auto reload on/off	[15]	Determine auto reload on/off for Timer 2. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 2 output inverter on/off	[14]	Determine output inverter on/off for Timer 2. 0 = Inverter off 1 = Inverter on for TOUT2	0
Timer 2 manual update (note)	[13]	Determine the manual update for Timer 2. 0 = No operation 1 = Update TCNTB2 & TCMPB2	0
Timer 2 start/stop	[12]	Determine start/stop for Timer 2. 0 = Stop 1 = Start for Timer 2	0
Timer 1 auto reload on/off	[11]	Determine the auto reload on/off for Timer1. 0 = One-shot 1 = Interval mode (auto reload)	0
Timer 1 output inverter on/off	[10]	Determine the output inverter on/off for Timer1. 0 = Inverter off 1 = Inverter on for TOUT1	0
Timer 1 manual update (note)	[9]	Determine the manual update for Timer 1. 0 = No operation 1 = Update TCNTB1 & TCMPB1	0
Timer 1 start/stop	[8]	Determine start/stop for Timer 1. 0 = Stop 1 = Start for Timer 1	0

现在需要设置Timer0

TCON	Bit	Description	Initial state
Reserved	[7:5]	Reserved	
Dead zone enable	[4]	Determine the dead zone operation. 0 = Disable 1 = Enable	0
Timer 0 auto reload on/off	[3]	Determine auto reload on/off for Timer 0. 0 = One-shot 1 = Interval mode(auto reload)	0
Timer 0 output inverter on/off	[2]	Determine the output inverter on/off for Timer 0. 0 = Inverter off 1 = Inverter on for TOUT0	0
Timer 0 manual update (note)	[1]	Determine the manual update for Timer 0. 0 = No operation 1 = Update TCNTB0 & TCMPB0	0
Timer 0 start/stop	[0]	Determine start/stop for Timer 0. 0 = Stop 1 = Start for Timer 0	0

开始需要手工更新

```
TCON |= (1<<1);    /* Update from TCNTB0 & TCMPB0 */
```

把这两个值放到TCNTB0 和 TCMPB0中

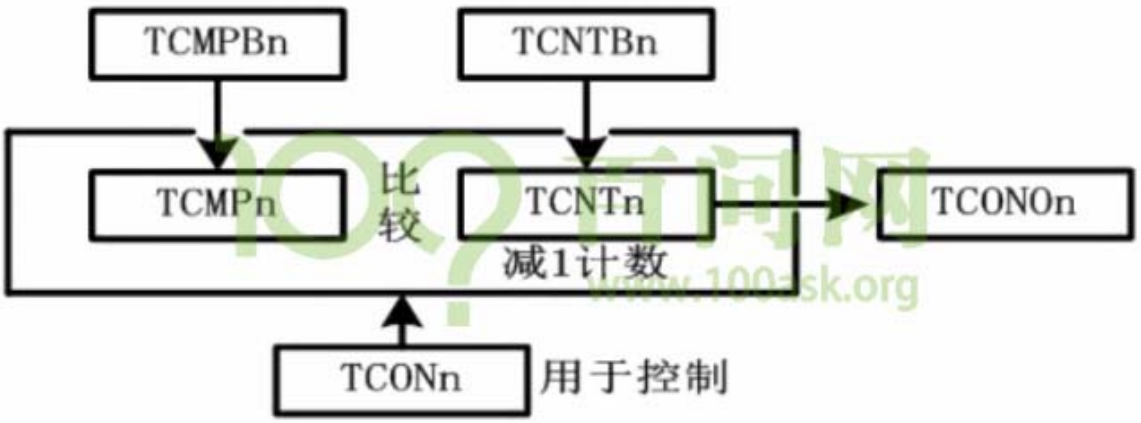


图 10.4 定时器内部控制逻辑图

注意：这一位必须清楚才能写下一位

设置为自动加载并启动，先清掉手动更新位，再或上bit0 bit3

```
TCON &= (1<<1);
TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto reload */
```

设置中断，显然我们需要提供一个中断处理函数`void timer_irq(void)` 在Timer里没有看到中断相关的控制器，我们需要回到中断章节去看看中断控制器，看看有没有定时器相关的中断 我们没有看到更加细致的Timer0寄存器

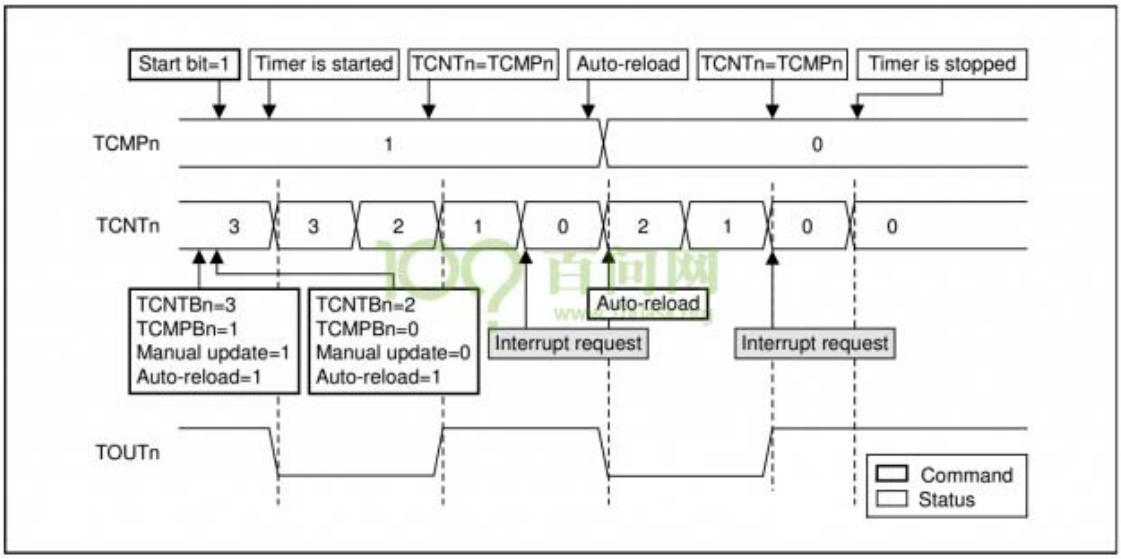


Figure 10-2. Timer Operations

当`TCNTn=TCMPn`时，他不会产生中断，只有当`TCNTn`等于0的时候才可以产生中断，我们之前以为这个定时器可以产生两种中断，那么肯定有寄存器中断或者禁止两种寄存器其中之一，那现在只有一种中断的话，就相对简单些 设置中断的话，我们只需要设置中断控制器

设置`interrupu.c`中断控制器

- 初始化中断控制器 `void interrupt_init(void)`

```
INTMSK &= ~((1<<0) | (1<<2) | (1<<5));
```

- 把定时器相应的位清零就可以了，哪一位呢？

INTPND的哪一位？ INT_TIMER0第10位即可

INTERRUPT PENDING (INTPND) REGISTER (Continued)

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
INT_NFCON	[24]	0 = Not requested, 1 = Requested	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

```
INTMSK &= ~(1<<10);  /* enable timer0 int */
```

当定时器减到0的时候就会产生中断，就会进到start.s这里一路执行do_irq

```
do_irq:
    /* 执行到这里之前:
     * 1. lr_irq保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10010, 进入到irq模式
     * 4. 跳到0x18的地方执行程序
     */

    /* sp_irq未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */
    /* 在irq异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr-4是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理irq异常 */
    bl handle_irq_c

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr_irq的值恢复到cpsr里 */


让后进入irq处理函数中处理，处理这个irq
void handle_irq_c(void)
{
    /* 分辨中断源 */
    int bit = INTOFFSET;

    /* 调用对应的处理函数 */


if(bit ==0 || bit == 2 || bit == 5)/*eint0,2,rint8_23*/
{
    key_eint_irq(bit);/*处理中断，清中断源EINTPEND*/
```

```
}else if(bit == 10)//如果等于10的话说明发生的是定时器中断，这时候就调用我们得timer_irq
{
    timer_irq();
}

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}


回到timer.c文件中，在这个定时器处理函数中我们需要点灯
void timer_irq(void)
{
    /* 点灯计数 循环点灯*/
    static int cnt = 0;
    int tmp;

    cnt++;

    tmp = ~cnt;
    tmp &= 7;
    GPFDAT &= ~(7<<4);
    GPFDAT |= (tmp<<4);
}
```

代码写完我们实验一下，上传代码，在Makefile中添加timer.o，进行编译 编译后进行烧写 现象灯没有闪烁

他不是有一个观察寄存器么?

我们不断的打印这个值，看是否有变化

TIMER 0 COUNT OBSERVATION REGISTER (TCNTO0)

Register	Address	R/W	Description	Reset Value
TCNTO0	0x51000014	R	Timer 0 count observation register	0x00000000

TCNTO0	Bit	Description	Initial State
Timer 0 observation register	[15:0]	Set count observation value for Timer 0	0x00000000

在main函数中不断打印

```
putchar(g_Char3);
g_Char3++;
delay(1000000);
printHex(TCNTO0);
```

编译实验
打印结果全都是0，发现我们的定时器根本就没有启用，在timer.c文件void timer_init(void)函数里设置为自动加载并启动，先清掉手动更新位，再或上bit0 bit3

```
TCON &= ~(1<<1);//我们没有设置取反
TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto reload */
```

再次实验
发现灯已经开始闪，就可以把调试信息去除了 对程序进行改进 进入main函数中执行 timer_init();

还需要修改interrupt.c 初始化函数 void interrupt_init(void) 还需要调用中断处理函数 void handle_irq_c(void) 每次添加一个中断我都需要修改handle_irq这个函数，这样太麻烦，我能不能保证这个interrupt文件不变，只需要在timer.c中引用即可，这里我们使用指针数组

在interrupt.c中定义函数指针数组

```
typedef void(*irq_func)(int);
```

定义一个数组，我们来卡看下这里有多少项，一共32位，我们想把每一个中断的处理函数都放在这个数组里面来，当发生中断时，我们可以得到这个中断号，让后我从数组里面调用对应的中断号就可以了

```
irq_func irq_array[32];
```

那么我们得提供一个注册函数

```
void register_irq (int irq, irq_func fp)
{
    irq_array[irq] = fp;
    INTMASK &= ~(1 << irq)
}
```

以后我直接调用对应的处理函数

```
void handle_irq_c(void)
{
    /* 分辨中断源 */
    int bit = INTOFFSET;


/* 调用对应的处理函数 */
irq_array[bit](bit);

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}

//按键中断初始化函数需要注册

    /* 初始化按键, 设为中断源 */
    void key_eint_init(void)
    {
    /* 配置GPIO为中断引脚 */
    GPFCON &= ~((3<<0) | (3<<4));
    GPFCON |= ((2<<0) | (2<<4));   /* S2,S3被配置为中断引脚 */

    GPGCON &= ~((3<<6) | (3<<22));
    GPGCON |= ((2<<6) | (2<<22));   /* S4,S5被配置为中断引脚 */


    /* 设置中断触发方式: 双边沿触发 */
    EXTINT0 |= (7<<0) | (7<<8);     /* S2,S3 */
    EXTINT1 |= (7<<12);             /* S4 */
    EXTINT2 |= (7<<12);             /* S5 */

    /* 设置EINTMASK使能eint11,19 */
    EINTMASK &= ~((1<<11) | (1<<19));

register_irq(0, key_eint_irq);
register_irq(2, key_eint_irq);
register_irq(5, key_eint_irq);
    }
```

在timer.c中也需要设置中断

```
    void timer_init(void)
    {
    /* 设置TIMER0的时钟 */
    /* Timer clk = PCLK / {prescaler value+1} / {divider value}
                 = 50000000/(99+1)/16
                 = 31250
     */
    TCFG0 = 99;  /* Prescaler 0 = 99, 用于timer0,1 */
    TCFG1 &= ~0xf;
    TCFG1 |= 3;  /* MUX0 : 1/16 */

    /* 设置TIMER0的初值 */
    TCNTB0 = 15625;  /* 0.5s中断一次 */

    /* 加载初值, 启动timer0 */
```

```
    TCON |= (1<<1);    /* Update from TCNTB0 & TCMPB0 */

    /* 设置为自动加载并启动 */
     TCON &= ~(1<<1);
     TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto reload */

       /* 设置中断 */
        register_irq(10, timer_irq);
    }
```

把interrupt.c中按键的初始化放在最后面

我们来看看我们做了什么事情，
<1>我们定义了一个指针数组

```
typedef void(*irq_func)(int);
```

这个指针数组里面放有各个指针的处理函数

```
irq_func irq_array[32];
```

当我们去初始化按键中断时，我们给这按键注册中断函数

```
register_irq(0, key_eint_irq);
register_irq(2, key_eint_irq);
register_irq(5, key_eint_irq);
```

这个注册函数会做什么事情，他会把这个数组放在注册函数里面，同时使能中断

```
void register_irq(int irq, irq_func fp)
{
    irq_array[irq] = fp;

    INTMSK &= ~(1<<irq);
}
//我们的timer.c中

timer_init();
//也会注册这个函数
    /* 设置中断 */
    register_irq(10, timer_irq);
```

把这个中断irq放在第10项里同时使能中断，以后我们只需要添加中断号，和处理函数即可，再也不需要修改函数烧写执行

我们从start.s开始看， 一上电从 b reset运行做一列初始化

```
.text
.global _start

_start:
    b reset          /* vector 0 : reset */
    ldr pc, und_addr /* vector 4 : und */
    ldr pc, swi_addr /* vector 8 : swi */
    b halt           /* vector 0x0c : prefetch aboot */
    b halt           /* vector 0x10 : data abort */
    b halt           /* vector 0x14 : reserved */


reset:
    /* 关闭看门狗 */
    ldr r0, =0x53000000
    ldr r1, =0
    str r1, [r0]

    /* 设置MPLL, FCLK : HCLK : PCLK = 400m : 100m : 50m */
    /* LOCKTIME(0x4C000000) = 0xFFFFFFFF */
    ldr r0, =0x4C000000
```

```
    ldr r1, =0xFFFFFFFF
    str r1, [r0]

    /* CLKDIVN(0x4C000014) = 0X5, tFCLK:tHCLK:tPCLK = 1:4:8  */
    ldr r0, =0x4C000014
    ldr r1, =0x5
    str r1, [r0]

    /* 设置CPU工作于异步模式 */
    mrc p15,0,r0,c1,c0,0
    orr r0,r0,#0xc0000000   //R1_nF:OR:R1_iA
    mcr p15,0,r0,c1,c0,0

    /* 设置MPLLCON(0x4C000004) = (92<<12)|(1<<4)|(1<<0)
     *  m = MDIV+8 = 92+8=100
     *  p = PDIV+2 = 1+2 = 3
     *  s = SDIV = 1
     *  FCLK = 2*m*Fin/(p*2^s) = 2*100*12/(3*2^1)=400M
     */
    ldr r0, =0x4C000004
    ldr r1, =(92<<12)|(1<<4)|(1<<0)
    str r1, [r0]

    /* 一旦设置PLL, 就会锁定lock time直到PLL输出稳定
     * 然后CPU工作于新的频率FCLK
     */



    /* 设置内存: sp 栈 */
    /* 分辨是nor/nand启动
     * 写0到0地址, 再读出来
     * 如果得到0, 表示0地址上的内容被修改了, 它对应ram, 这就是nand启动
     * 否则就是nor启动
     */
    mov r1, #0
    ldr r0, [r1] /* 读出原来的值备份 */
    str r1, [r1] /* 0->[0] */
    ldr r2, [r1] /* r2=[0] */
    cmp r1, r2   /* r1==r2? 如果相等表示是NAND启动 */
    ldr sp, =0x40000000+4096 /* 先假设是nor启动 */
    moveq sp, #4096  /* nand启动 */
    streq r0, [r1]   /* 恢复原来的值 */

    bl sdram_init
    //bl sdram_init2     /* 用到有初始值的数组, 不是位置无关码 */

    /* 重定位text, rodata, data段整个程序 */
    bl copy2sdram

    /* 清除BSS段 */
    bl clean_bss

    /* 复位之后, cpu处于svc模式
     * 现在, 切换到usr模式
     */
    mrs r0, cpsr         /* 读出cpsr */
    bic r0, r0, #0xf     /* 修改M4-M0为0b10000, 进入usr模式 */
    bic r0, r0, #(1<<7)  /* 清除I位, 使能中断 */
    msr cpsr, r0

    /* 设置 sp_usr */
    ldr sp, =0x33f00000

    ldr pc, =sdram
sdram:
    bl uart0_init

    bl print1
    /* 故意加入一条未定义指令 */
und_code:
    .word 0xdeadc0de  /* 未定义指令 */
    bl print2

    swi 0x123  /* 执行此命令, 触发SWI异常, 进入0x8执行 */

/***最后执行main函数***/
    //bl main  /* 使用BL命令相对跳转, 程序仍然在NOR/sram执行 */
    ldr pc, =main  /* 绝对跳转, 跳到SDRAM */

halt:
    b halt
```

进入main.c做一系列初始化

```
int main(void)
{
    led_init();
    //interrupt_init();  /* 初始化中断控制器 */
    key_eint_init();   /* 初始化按键, 设为中断源 */
    timer_init();

    puts("\n\rg_A = ");
    printHex(g_A);
    puts("\n\r");
```

进入按键初始化程序 interrupt.c 初始化按键,设为中断源

```
void key_eint_init(void)
{
    /* 配置GPIO为中断引脚 */
    GPFCON &= ~((3<<0) | (3<<4));
    GPFCON |= ((2<<0) | (2<<4));   /* S2,S3被配置为中断引脚 */

    GPGCON &= ~((3<<6) | (3<<22));
    GPGCON |= ((2<<6) | (2<<22));   /* S4,S5被配置为中断引脚 */


    /* 设置中断触发方式: 双边沿触发 */
    EXTINT0 |= (7<<0) | (7<<8);     /* S2,S3 */
    EXTINT1 |= (7<<12);             /* S4 */
    EXTINT2 |= (7<<12);             /* S5 */

    /* 设置EINTMASK使能eint11,19 */
     EINTMASK &= ~((1<<11) | (1<<19));

/*注册中断控制器*/
    register_irq(0, key_eint_irq);
    register_irq(2, key_eint_irq);
    register_irq(5, key_eint_irq);
}


时钟初始化程序 <code>timer_init();</code>

void timer_init(void)
{
    /* 设置TIMER0的时钟 */
    /* Timer clk = PCLK / (prescaler value+1) / (divider value)
                 = 50000000/(99+1)/16
                 = 31250
     */
    TCFG0 = 99;  /* Prescaler 0 = 99, 用于timer0,1 */
    TCFG1 &= ~0xf;
    TCFG1 |= 3;  /* MUX0 : 1/16 */

    /* 设置TIMER0的初值 */
    TCNTB0 = 15625;  /* 0.5s中断一次 */

    /* 加载初值, 启动timer0 */
    TCON |= (1<<1);   /* Update from TCNTB0 & TCMPB0 */

    /* 设置为自动加载并启动 */
    TCON &= ~(1<<1);
    TCON |= (1<<0) | (1<<3);  /* bit0: start, bit3: auto reload */

    /* 设置中断 */
    register_irq(10, timer_irq);
}
```

让后main.c函数一直循环执行 输出串口信息

```
 while (1)
 {
    putchar(g_Char);
    g_Char++;

    putchar(g_Char3);
    g_Char3++;
```

```
    delay(1000000);
    //printHex(TCNT00);
}
```

定时器减到0的时候就会产生中断，start.S 跳到 0x18的地方执行

```
    ldr pc, irq_addr /* vector 0x18 : irq */
    b halt           /* vector 0x1c : fiq */
.align 4

do_irq:
    /* 执行到这里之前:
     * 1. lr_irq保存有被中断模式中的下一条即将执行的指令的地址
     * 2. SPSR_irq保存有被中断模式的CPSR
     * 3. CPSR中的M4-M0被设置为10010, 进入到irq模式
     * 4. 跳到0x18的地方执行程序
     */

    /* sp_irq未设置, 先设置它 */
    ldr sp, =0x33d00000

    /* 保存现场 */
    /* 在irq异常处理函数中有可能会修改r0-r12, 所以先保存 */
    /* lr-4是异常处理完后的返回地址, 也要保存 */
    sub lr, lr, #4
    stmdb sp!, {r0-r12, lr}

    /* 处理irq异常 */
    bl handle_irq_c

    /* 恢复现场 */
    ldmia sp!, {r0-r12, pc}^  /* ^会把spsr_irq的值恢复到cpsr里 */
```

看看怎么处理irq

```
void handle_irq_c(void)
{
    /* 分辨中断源 */
    int bit = INTOFFSET;

    /* 调用对应的处理函数执行 */
    irq_array[bit](bit);

    /* 清中断 : 从源头开始清 */
    SRCPND = (1<<bit);
    INTPND = (1<<bit);
}
```

《《所有章节目录》》

第016课 Nand Flash
第017课 LCD
第018课 ADC和触摸屏
第019课 I2C
第20课 SPI

分类： ARM裸机加强版

- 本页面最后修改于2018年1月29日 (星期一) 15:15。