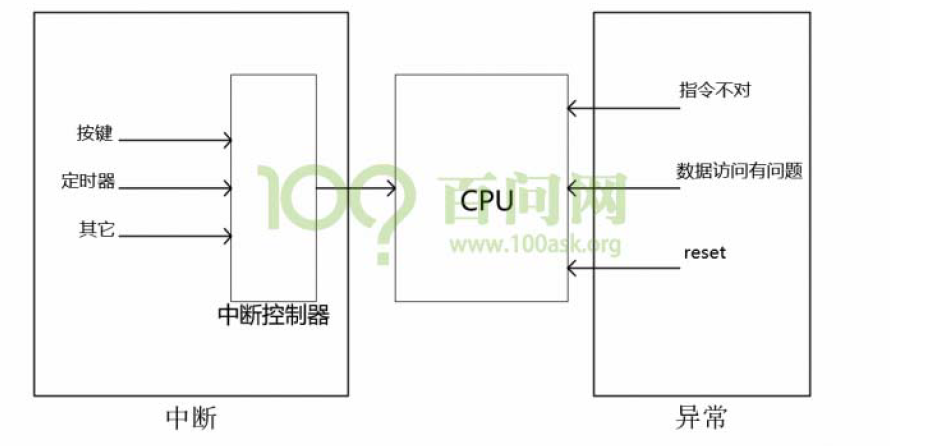
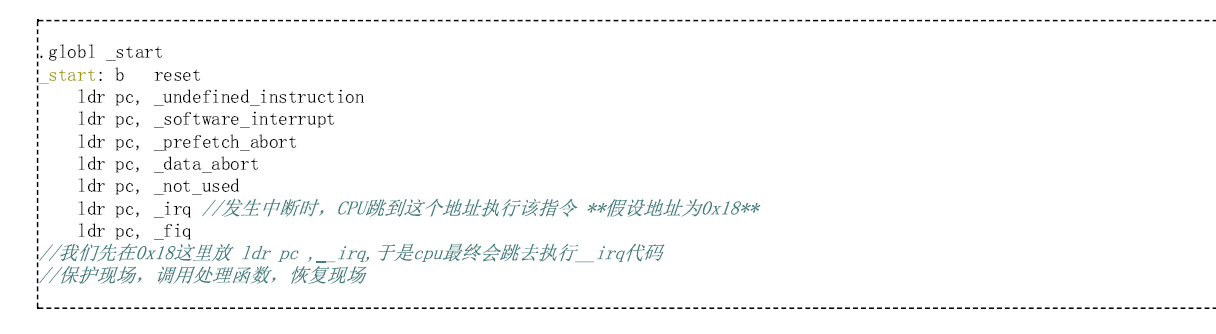
## 01概念与处理流程



arm对异常，中断处理guochemg：

1. 初始化：
   1. 设置中断源，使能产生中断
   2. 设置中断控制器（可以屏蔽某个中断，优先级）
   3. 设置CPU总开关（使能中断）
2. 执行其他程序（后台程序）
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的产生
5. 发现有中断异常产生，开始处理，对于不同的异常跳到不同的地址去执行程序，这些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，下图为异常向量表，对于不同的异常都有一条跳转指令。

（3—5都是硬件强制做的）



1. 这些函数做了什么事

（软件做的）

1. 保存现场（各种寄存器）
2. 处理异常中断：

分辨中断源

再调用不同的处理函数

1. 恢复现场

（CPU到0x18是硬件决定的，跳去执行处理函数是软件决定的）

## 02CPU模式（mode）状态（state）与寄存器

7种mode

usr/sys：正常模式/兴奋模式

undefined(und)：未定义模式

supervisor(svc)：管理模式

abort(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）

IRQ(irq) ：中断模式

FIQ(fiq) ：快中断模式

2种state

arm state：使用ARM指令集，每个指令4byte

thumb state：使用的thumb指令集，每个指令2byte

引入thumb减少存储空间，但是不如arm指令的效率高

寄存器：

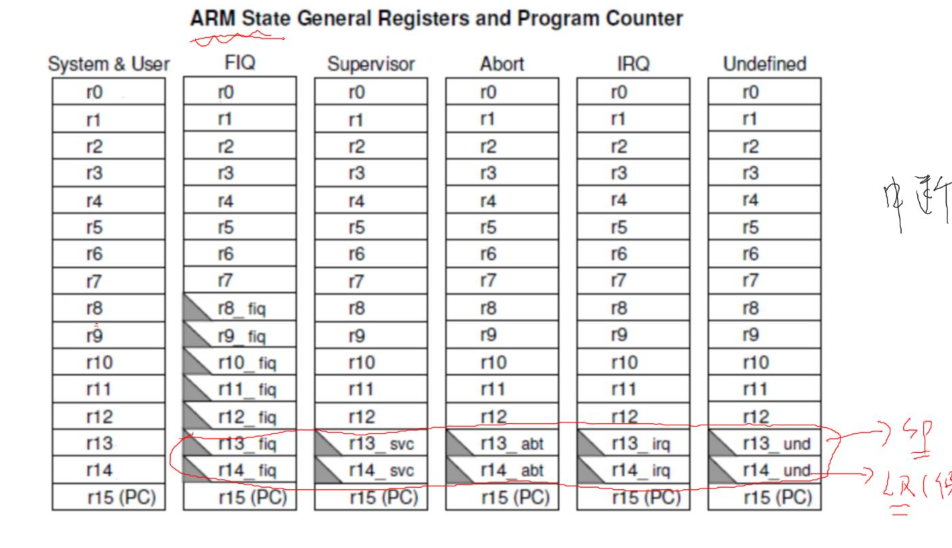
通用寄存器

备份寄存器 （banked register）指fiq的专属寄存器

link register （lr）？？：返回地址，保存发生异常时的指令地址

当前程序状态寄存器cpsr （current program status register）

cpsr的备份寄存器spsr （save program status register）



mov R0,R8

在其他模式下访问的是同个R8寄存器，但在FIQ模式下访问的是专属的R8寄存器，

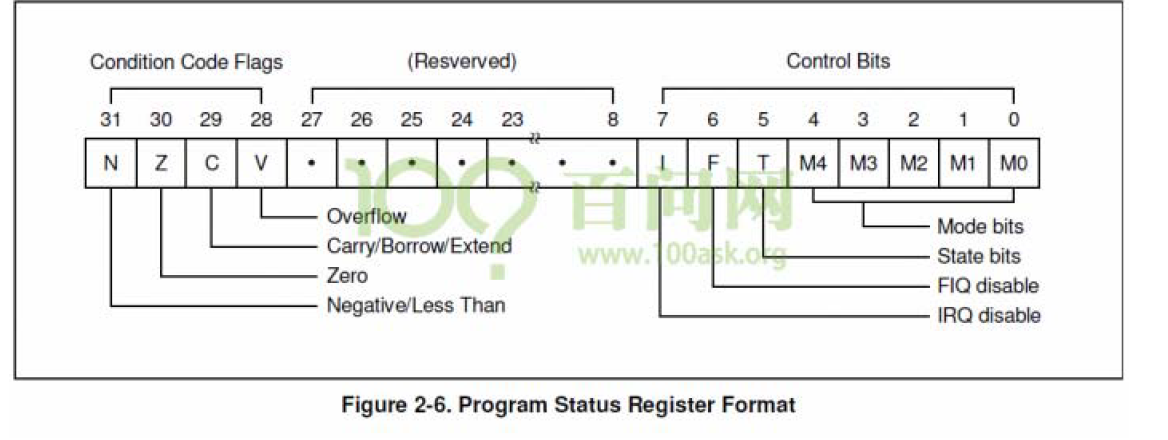
这五种异常模式中每个模式都有自己专属的R13（用于sp栈），R14（用于ls返回地址）寄存器

中断的处理过程：

保存现场（保存被中断模式的寄存器）

比如说程序正在系统模式下运行，当你发生中断时，需要把r0—r14这些寄存器全部保存下来，再处理异常，最后恢复这些寄存器

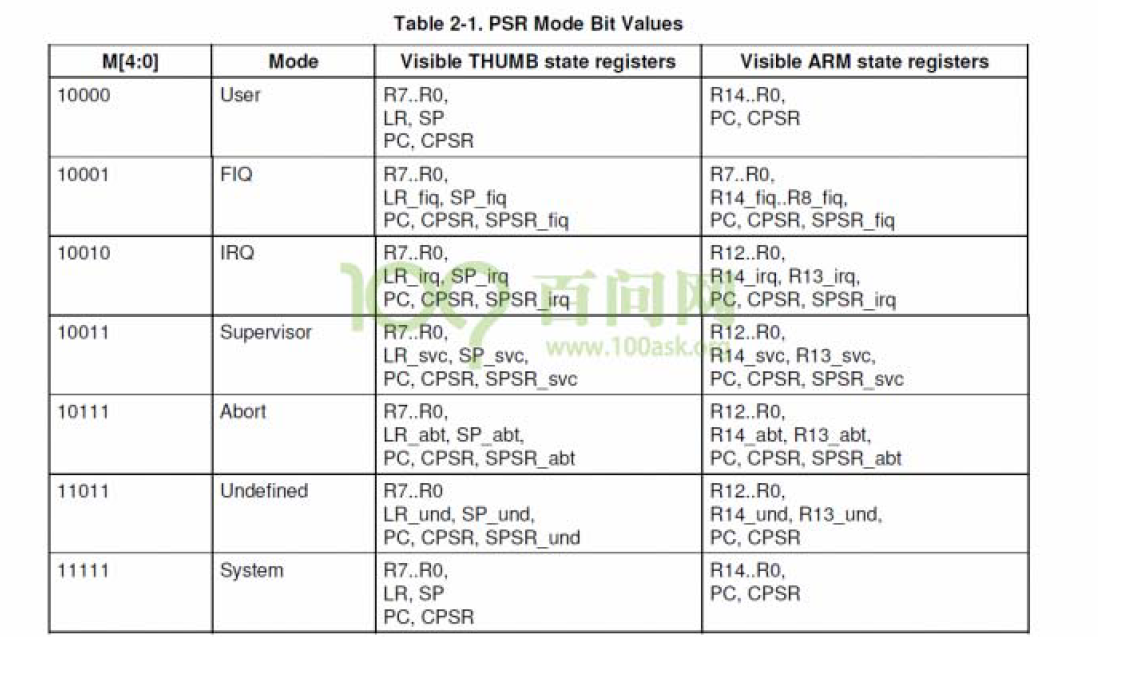
如果是fiq，就不需要保存系统模式下的r8—r12寄存器，fiq有专属的，节省保存时间加快处理速度。但是在linux中不会使用fiq



cpsr是当前程序状态寄存器，重要！上图所示，spsr和cpsr格式相同是

首先M4—M0表示当前CPU处于哪一种模式（mode）

可以读取这5位来判断处于那一模式，也可以修改进入模式，用户模式下没权限修改

M4—M0对应一下的值

再看其他位：

bit 5 ：state bit

bit 6 ：1禁止fiq模式

bit 7 ：1禁止irq，即所有中断

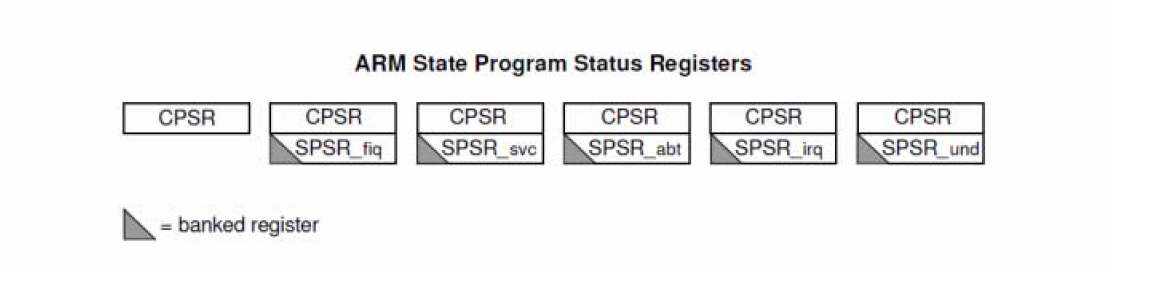
bit8 – bit27 ：保留位

bit 28—bit31 ：状态位

什么是状态位，比如说 cmp R0, R1

如果R0==R1那么zero==1，这条指令影响Z位，

beq跳转到xx指令，会判断z位是否为1，是才1跳转



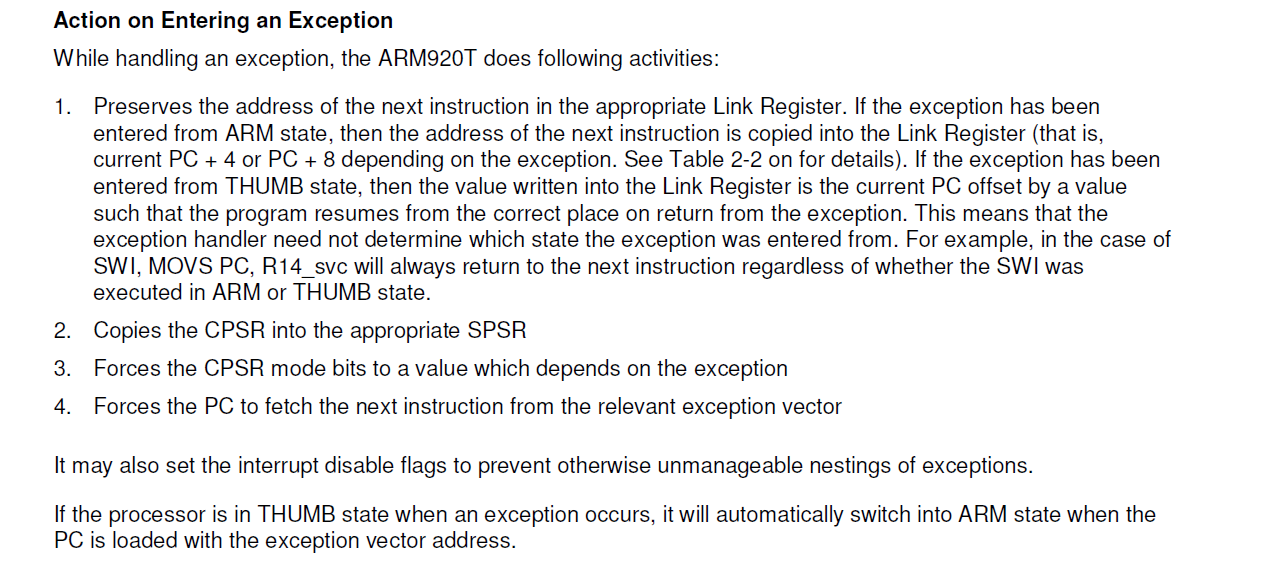
cpsr保存当下的mode、spsr保存中断前的mode

比如程序在系统模式下运行cpsr M[4:0] = 11111 ，发生中断进入irq模式 M[4:0] = 10010

那么在进入irq后，cpsr = 11111，spsr=110010

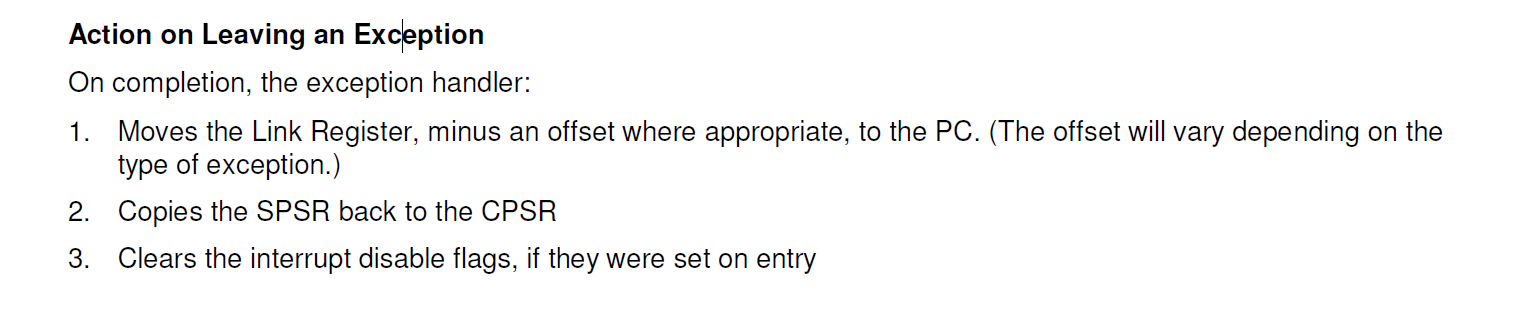
## 03芯片手册中断异常处理详细流程

发生中断，CPU做什么：



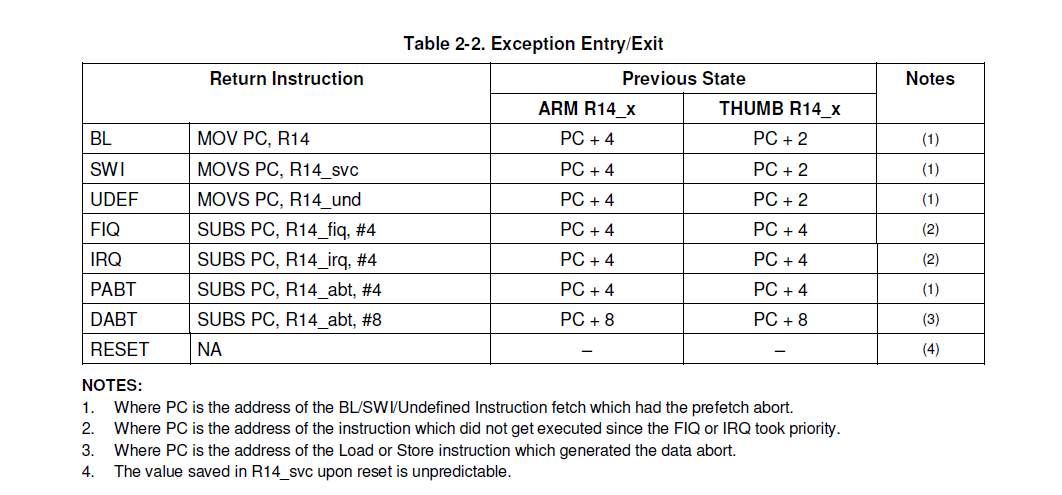
1. 把下一条指令的地址保存在lr\_异常 寄存器，可能是pc+4 或者pc+8，取决于不同情况
2. 把cpsr保存到spsr里
3. 修改cpsr的mode进入相应的模式
4. 跳到向量表

怎么退出：



1. pc = lr\_异常 – offset

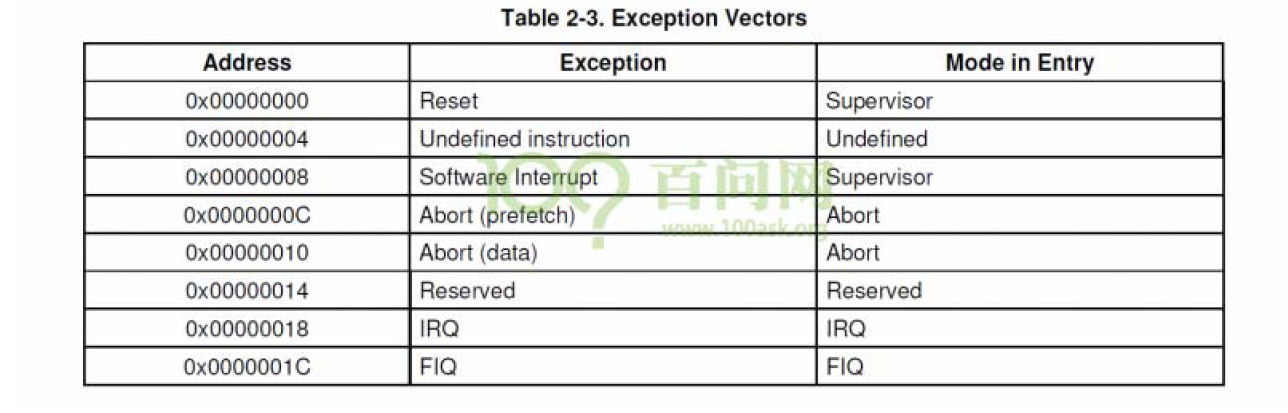
offset的值根据下表取值



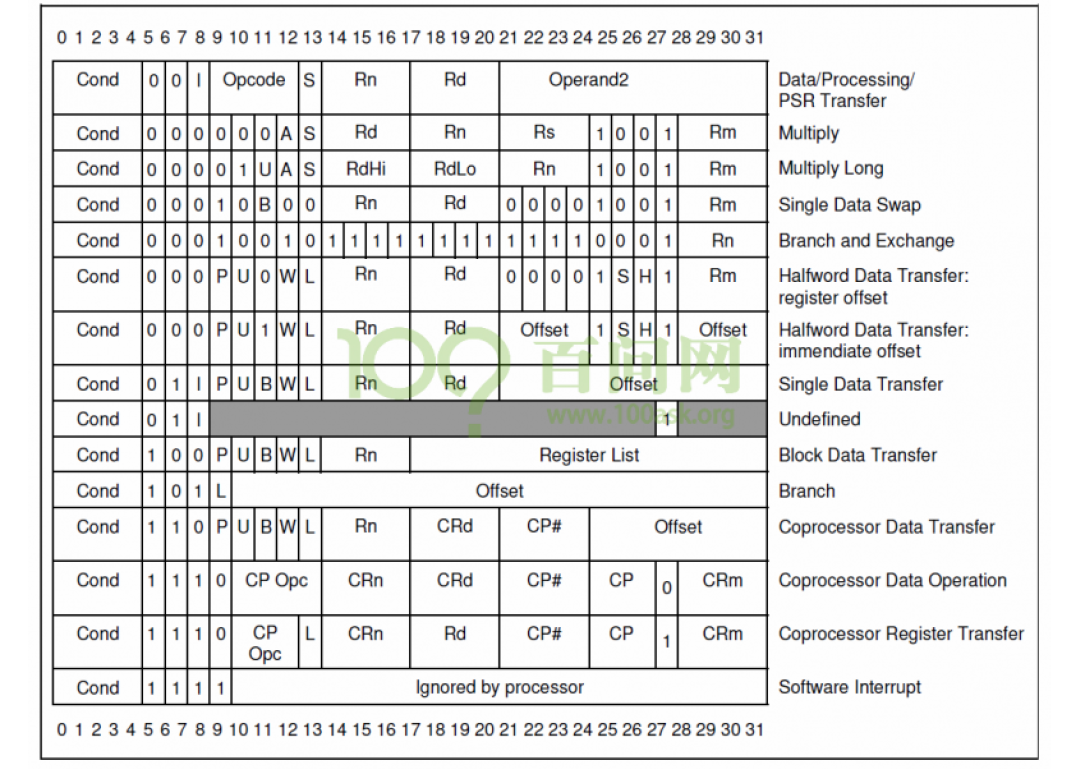
1. 把cpsr的值恢复（cpsr 等于 spsr）
2. 清中断

## 04und异常模式程序示例

手册异常向量表：

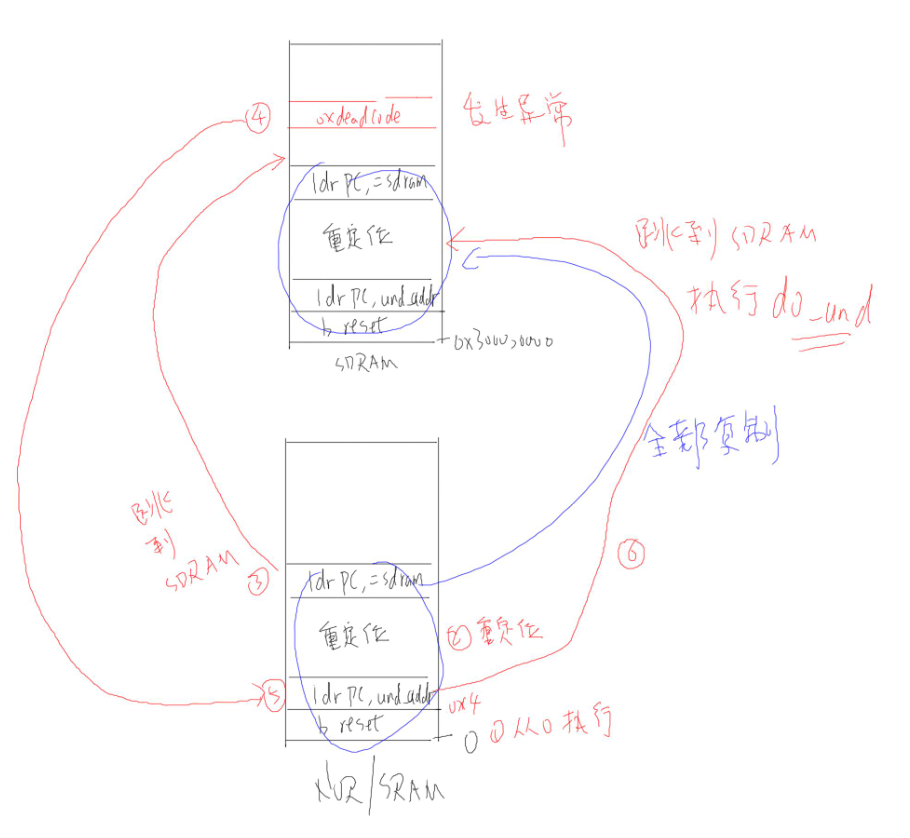


arm指令集，用来构造一条未定义指令：



见程序001

程序流程图



## 05swi异常程序示例

在上个程序的基础上，做些改动，异常处理函数把产生异常的val值打印出来

linux程序一般运行于用户模式，用户模式下不能修改cpsr进入其他模式，

可以通过发生异常再切换模式，

首先引入用户模式

见程序002

## 06按键中断程序示例

中断处理流程

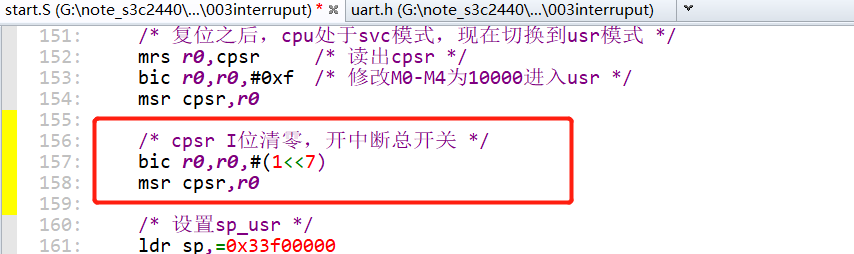
1. 中断初始化

a 设置中断源，使能发出中断信号

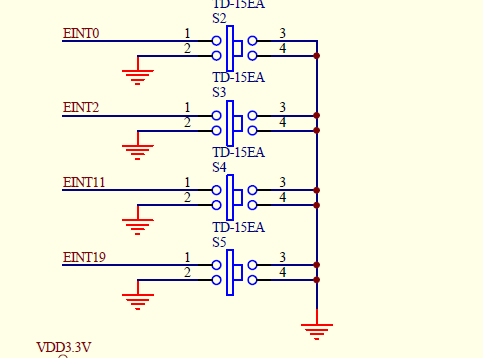
b 设置中断控制器，使能发中断给cpu

c 设置cpu，cpsr中I位，总开关

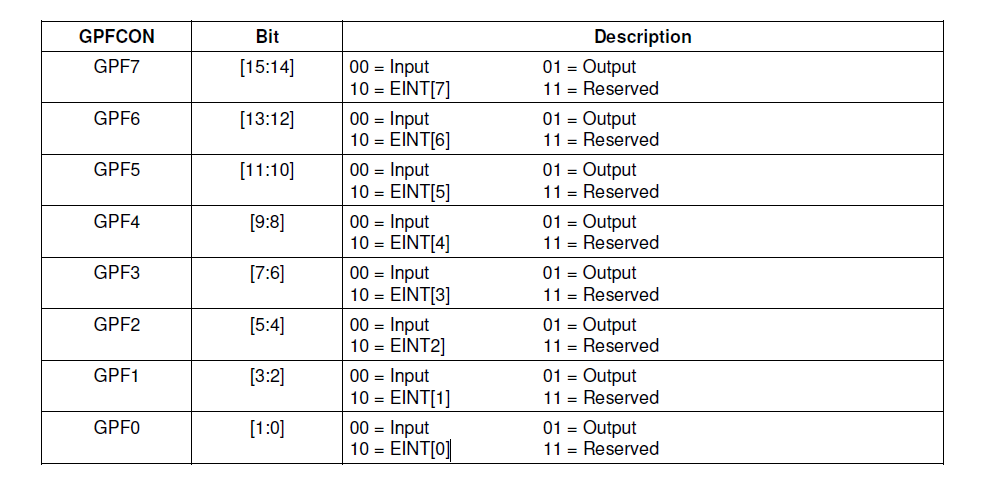
1. 处理完中断要清中断
2. 处理时要分辨中断源，对于不同的中断源执行不同的处理函数

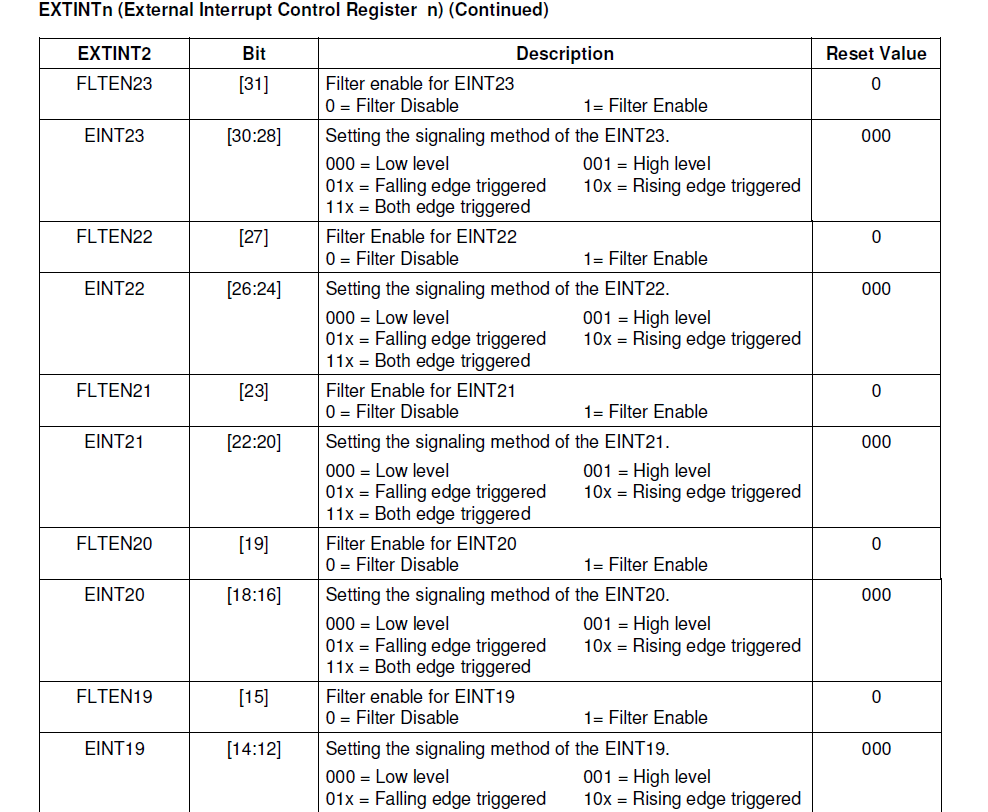
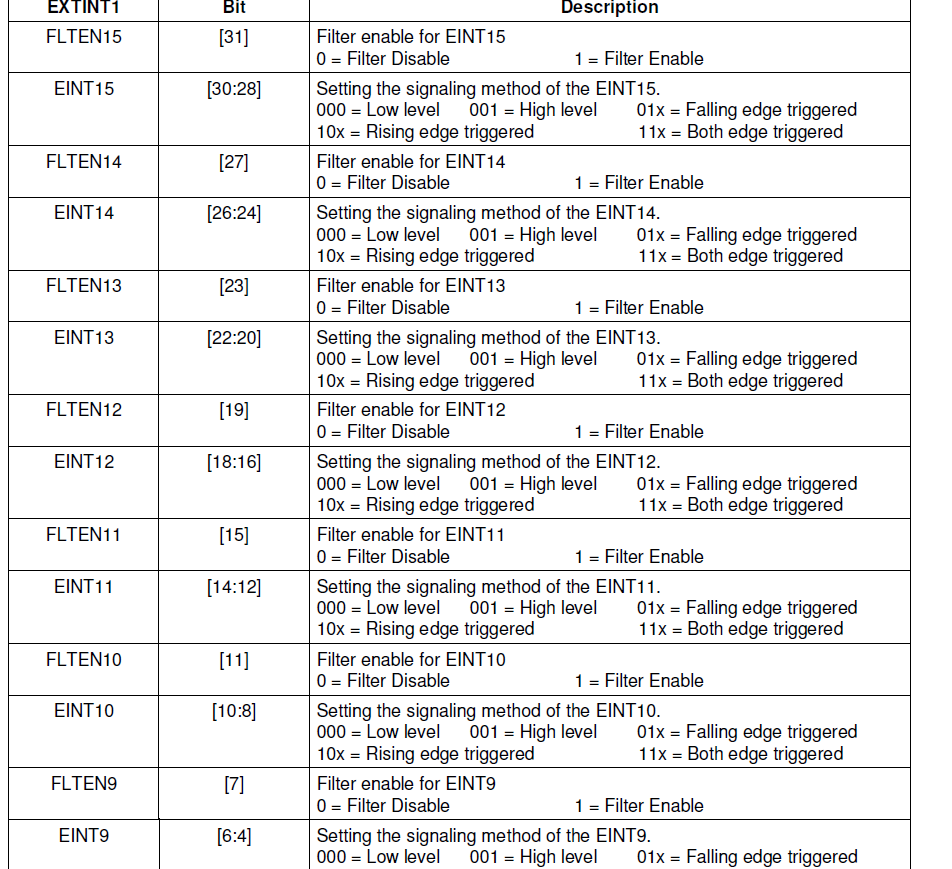
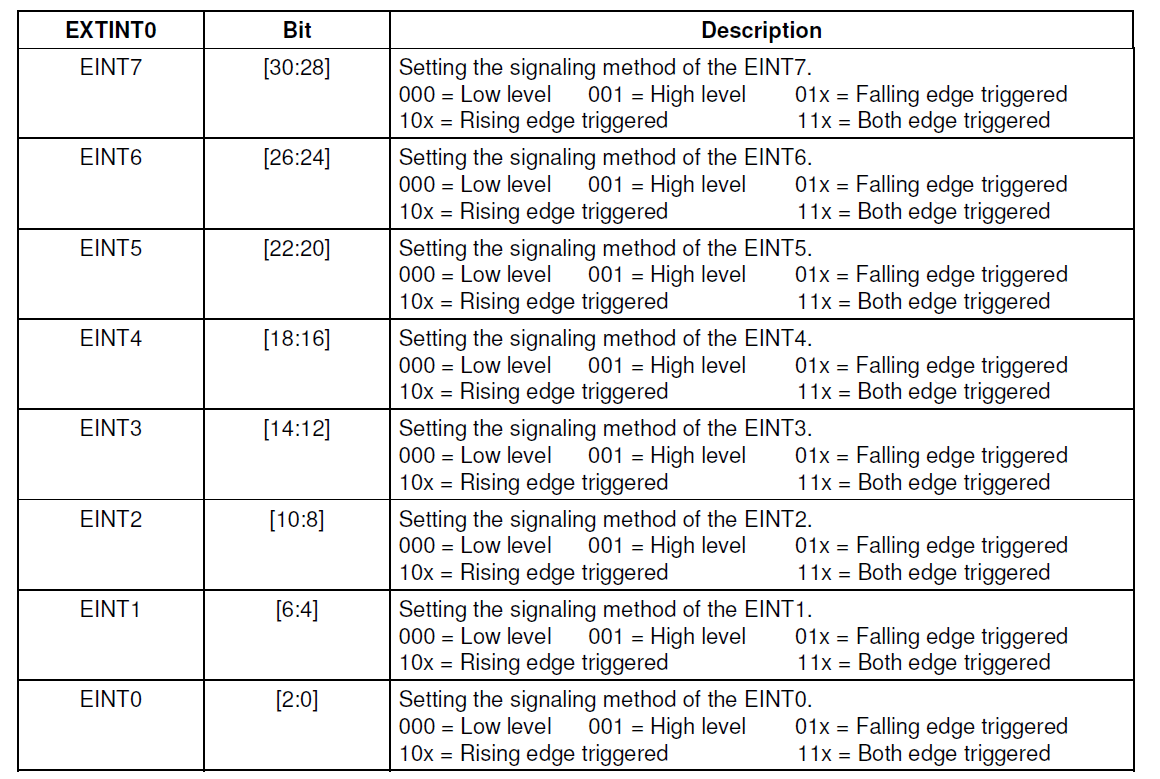
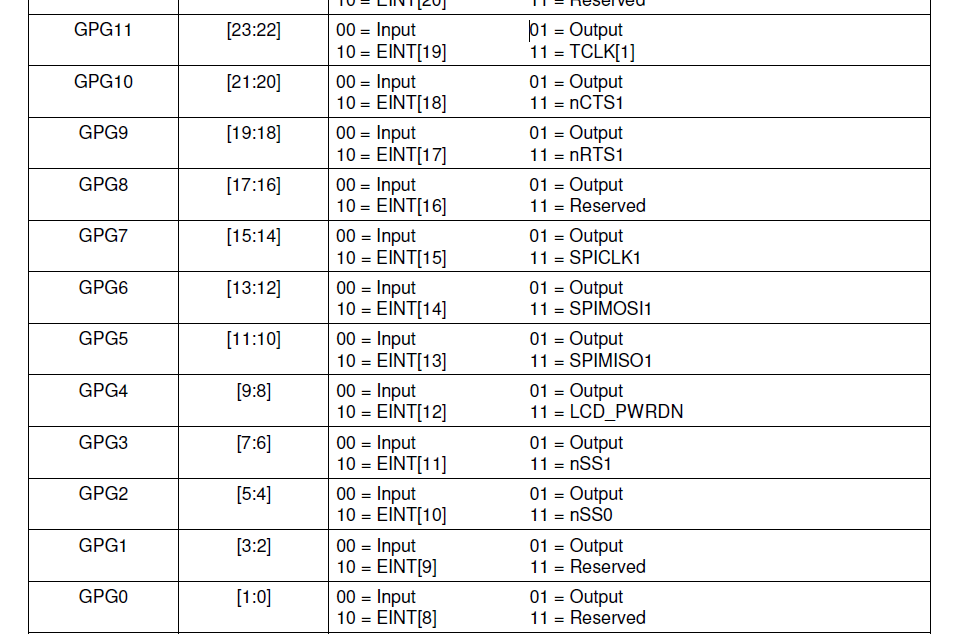


### 看原理图选择设置这几个按键为中断源

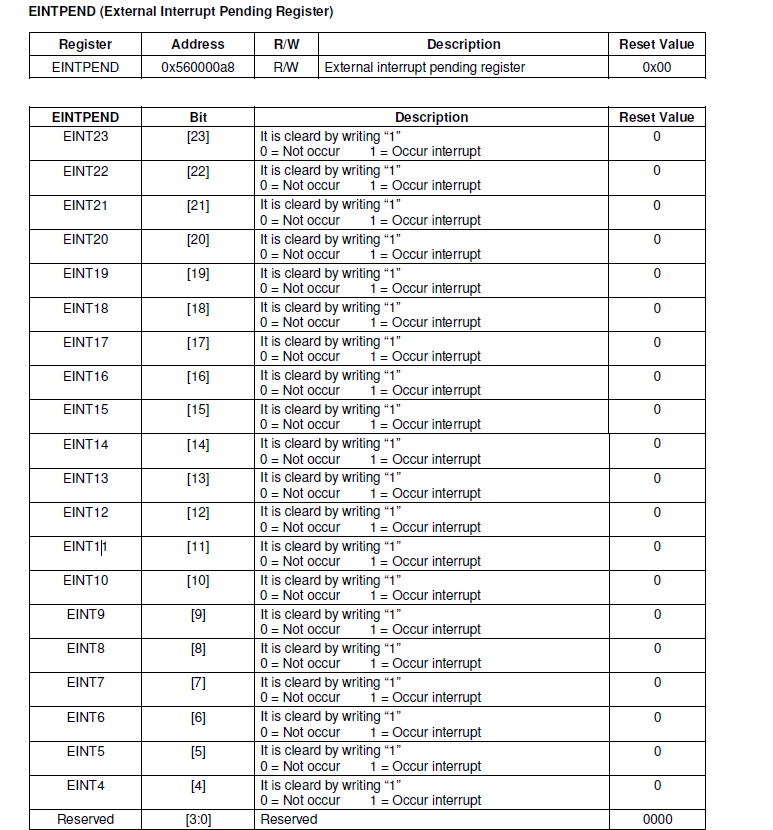
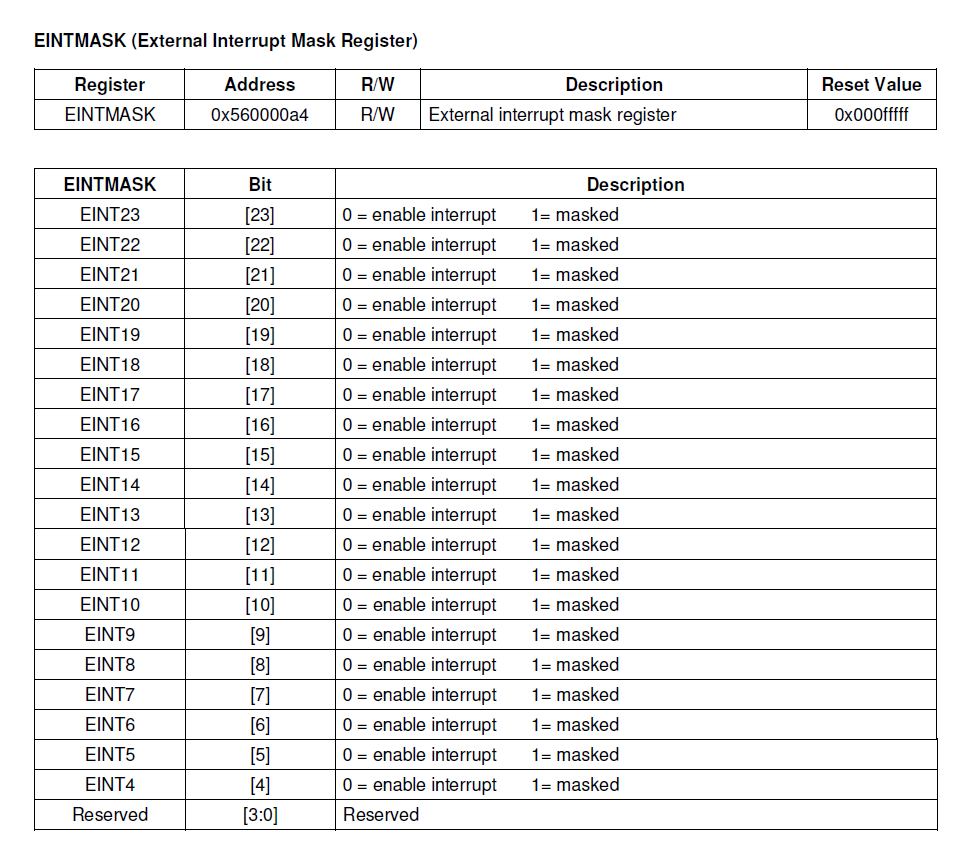


芯片手册找到相应寄存器，设置gpio为中断引脚，中断触发方式为双边沿触发

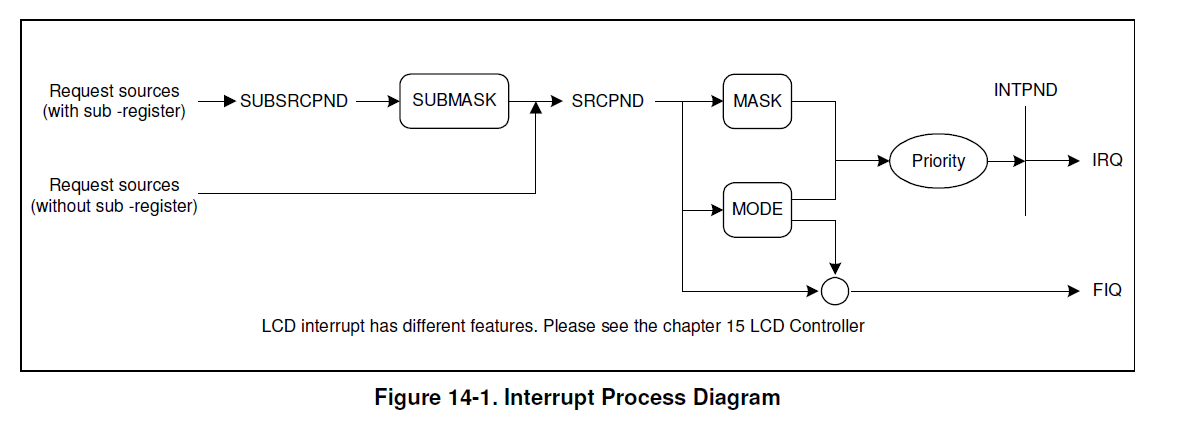




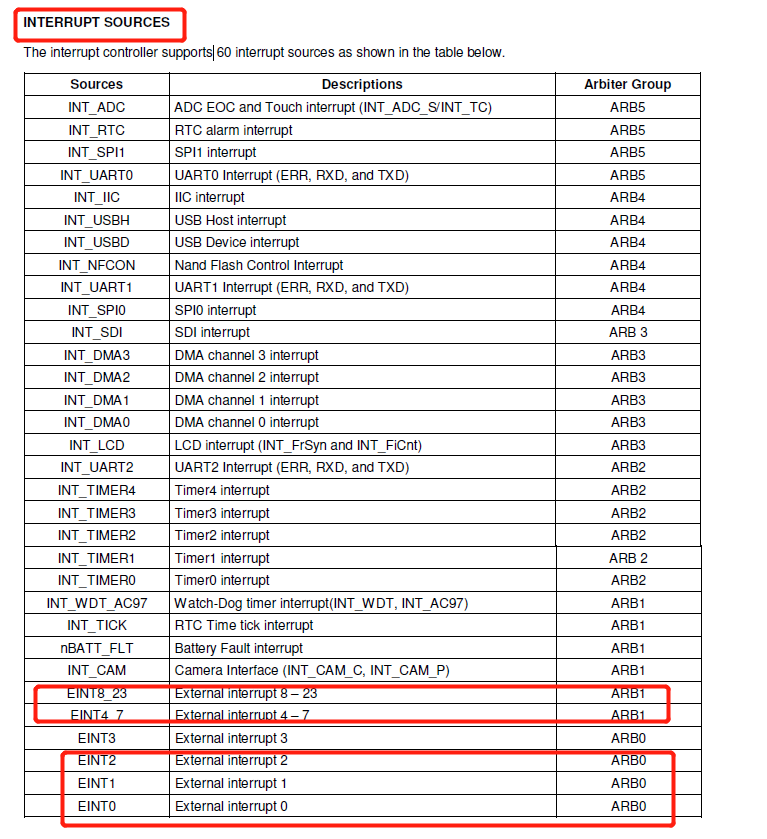
外部中断屏蔽寄存器，设置为1禁止向外部发出信号中断，所以这里设置0



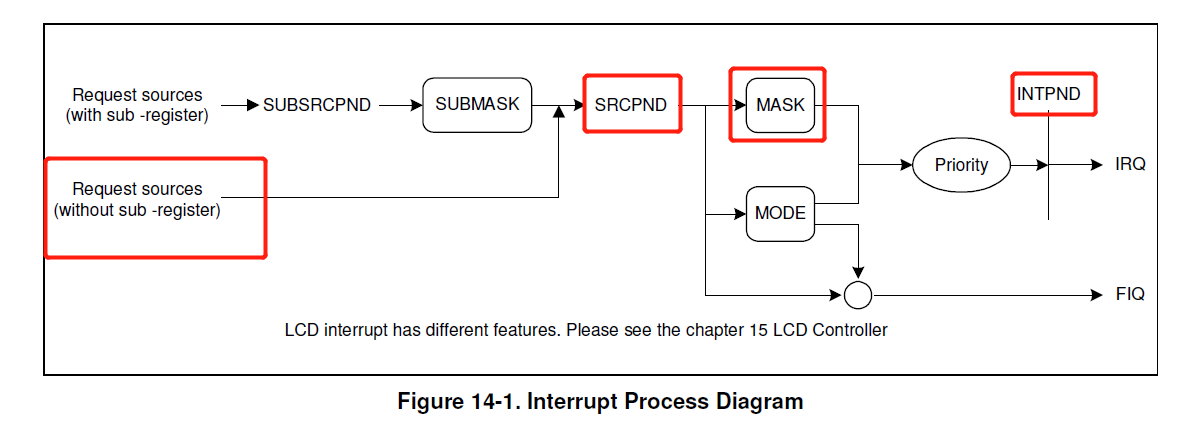
### 上面设置完后要来设置中断控制器



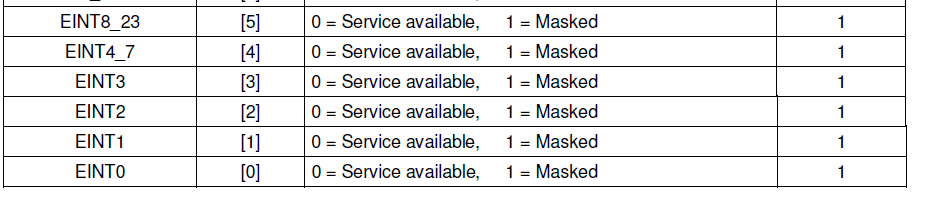
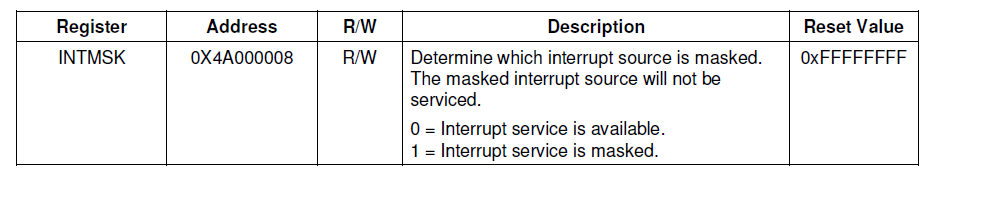
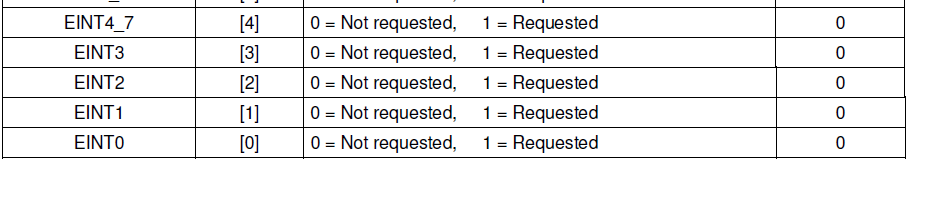
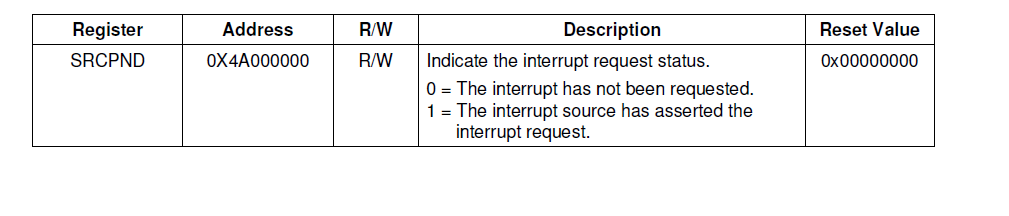
查看eint0 2 11 19

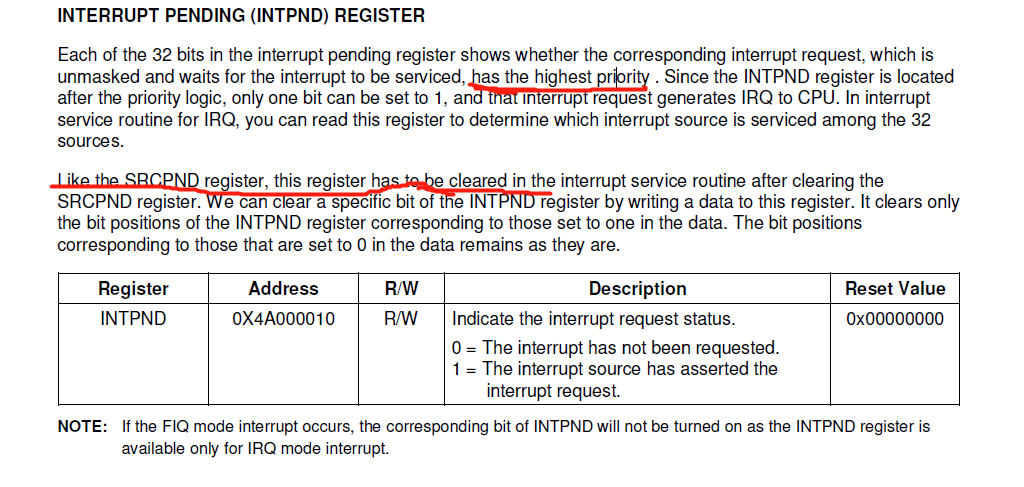


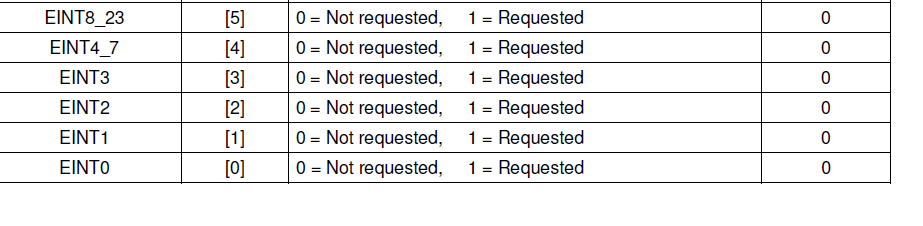
所以只需要设置 srcpnd mask intpnd 三个寄存器

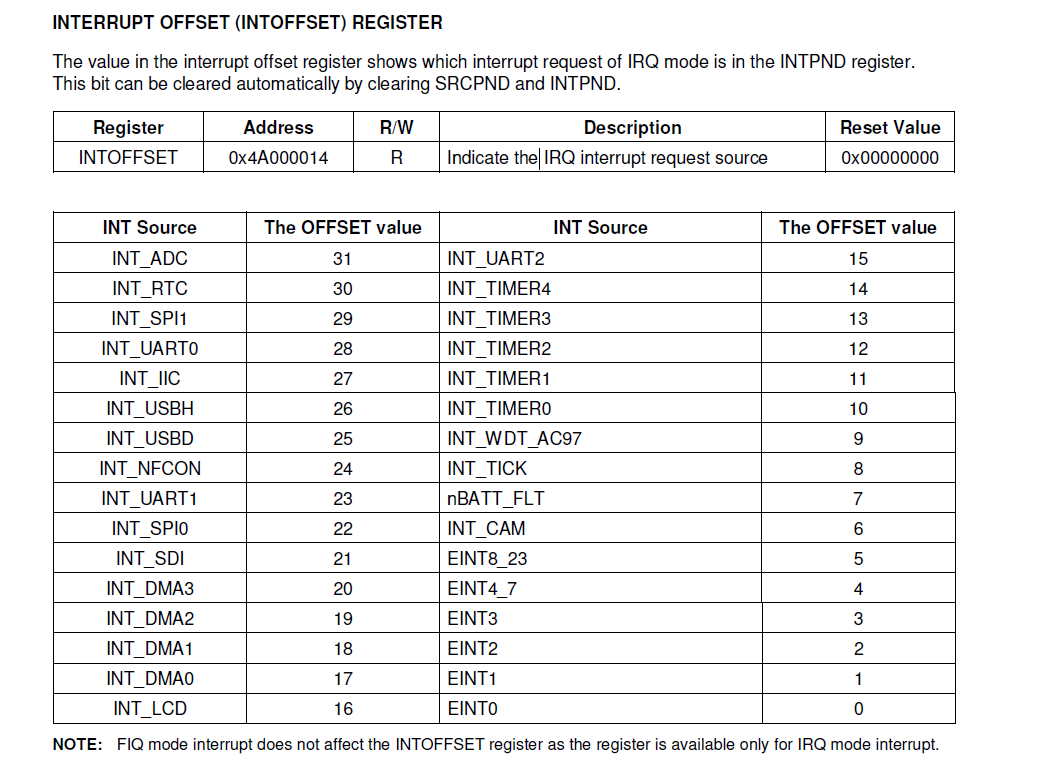


接下来查看三个寄存器









这里只设置intmsk，另外两个寄存器等发生中断的时候设置

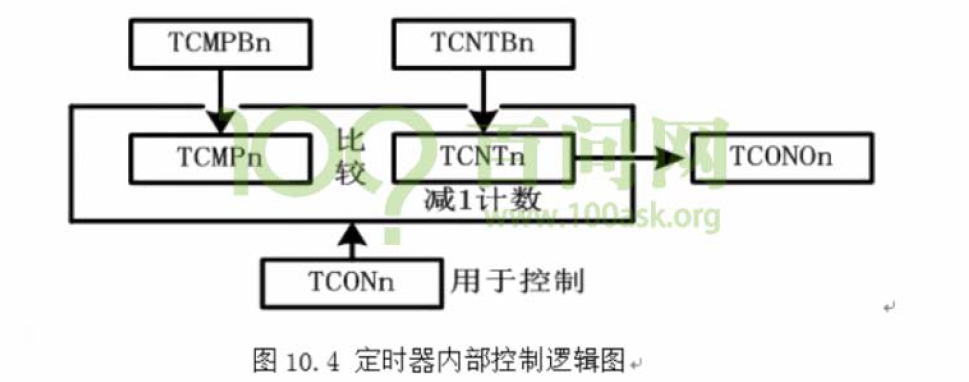
设置完成在main中添加初始化函数

然后程序发生中断，根据中断向量表在start.S添加代码

然后写中断处理函数

看程序003

## 07定时器中断程序示例



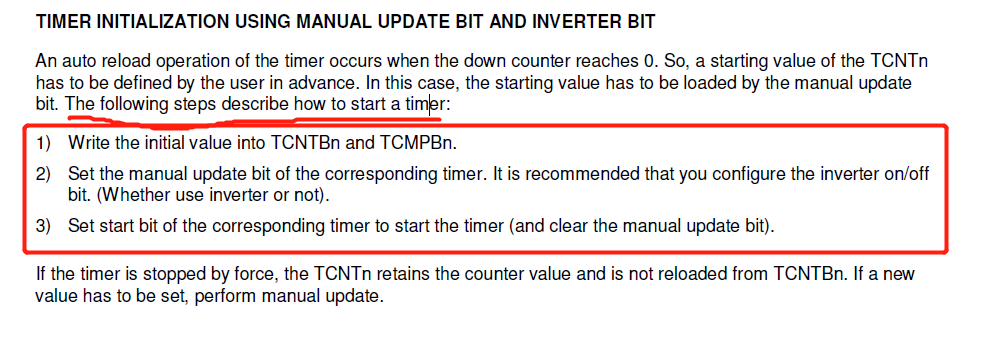
1. 每来一个clk，TCNTn – 1
2. 当TCNTn == 0时，可以再次产生中断
3. 自动加载初值，TCMOn和TCNTn的初值来自TCMPBn，TCNTBn

怎么使用定时器：

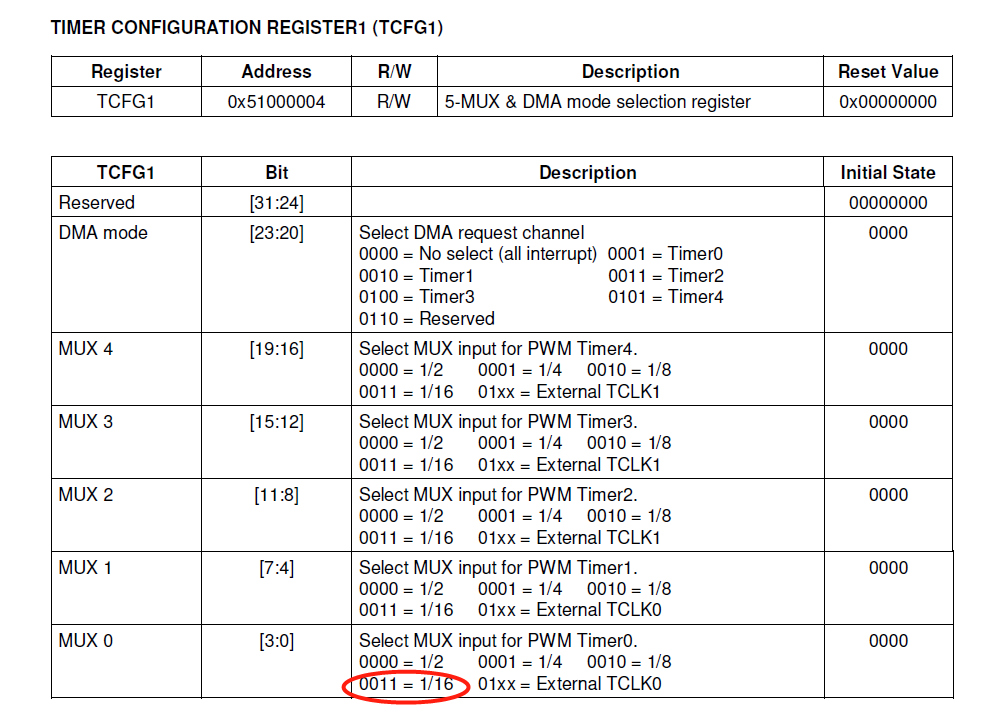
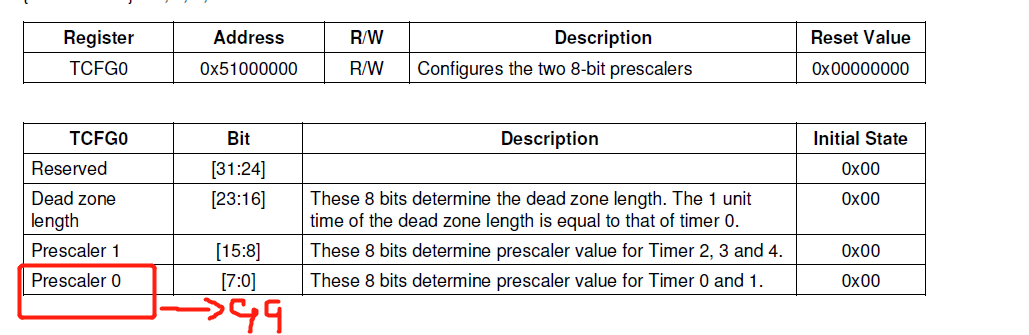
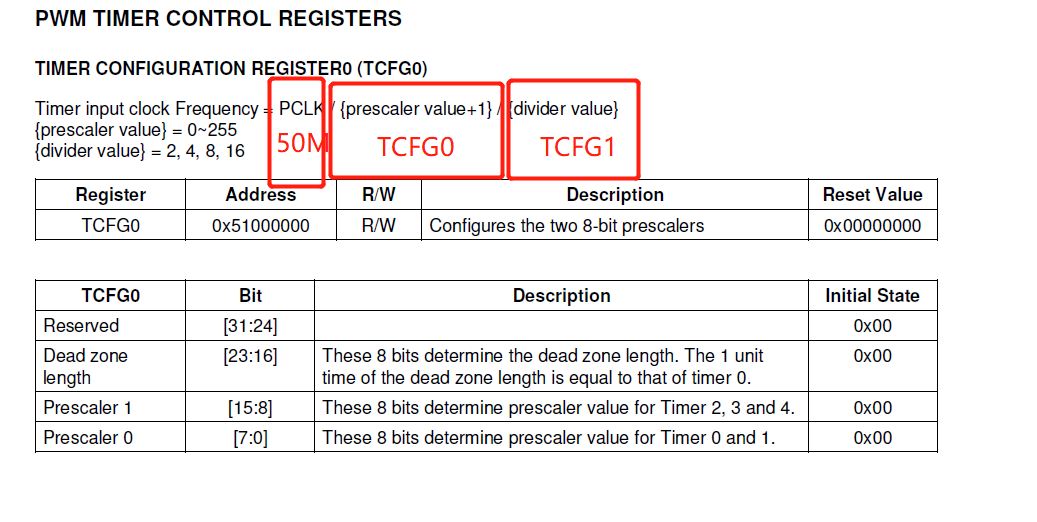
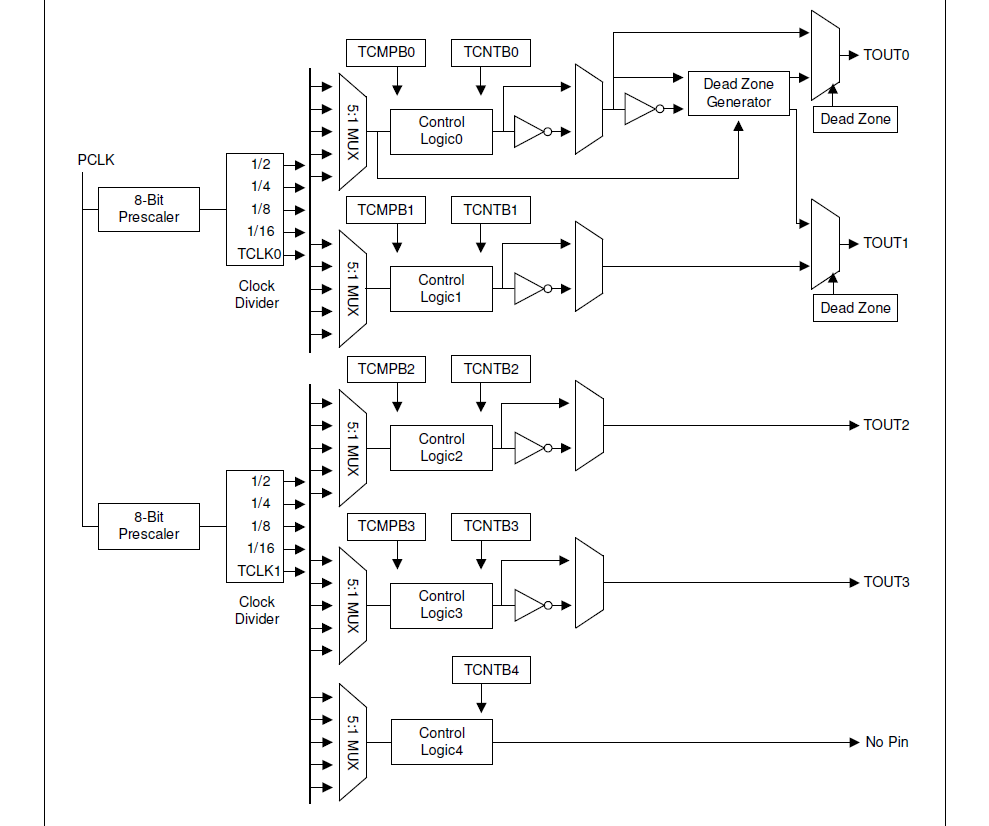
1. 设置时钟
2. 设置初值
3. 加载设置，启动Timer
4. 设置为自动加载初值
5. 中断相关

写程序实现TCNTn == 0产生中断，点灯

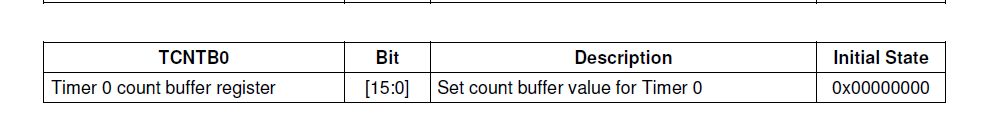
创建timer.c写timer\_init()

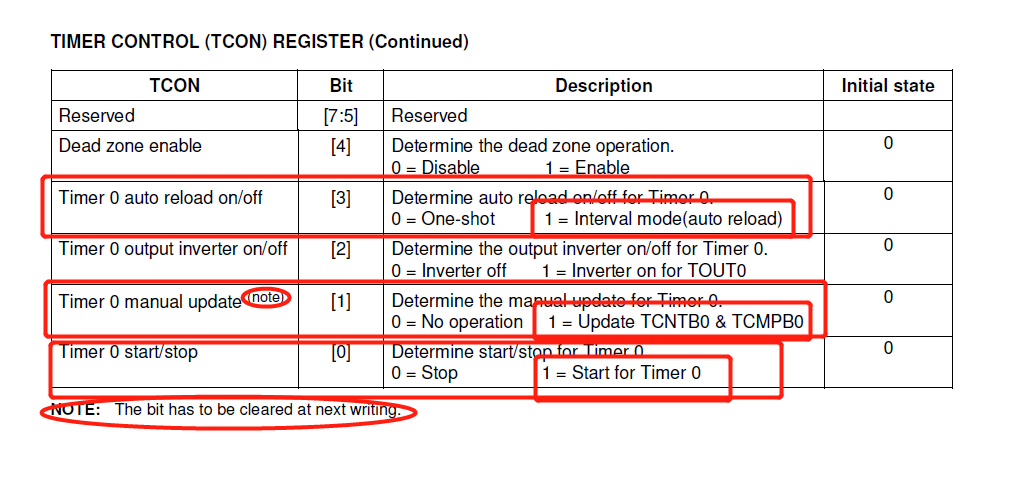
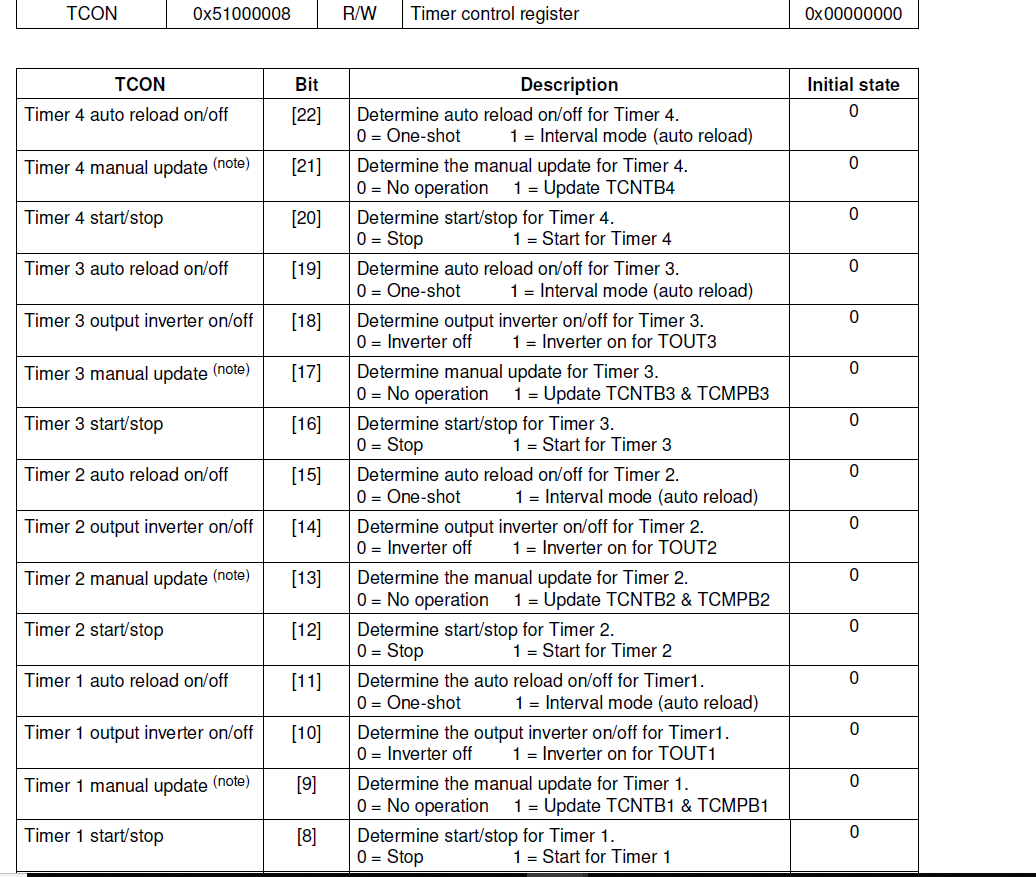


1. 初值写到TCNTBn和TCMPBn
2. 设置手动更新位
3. 设置启动位

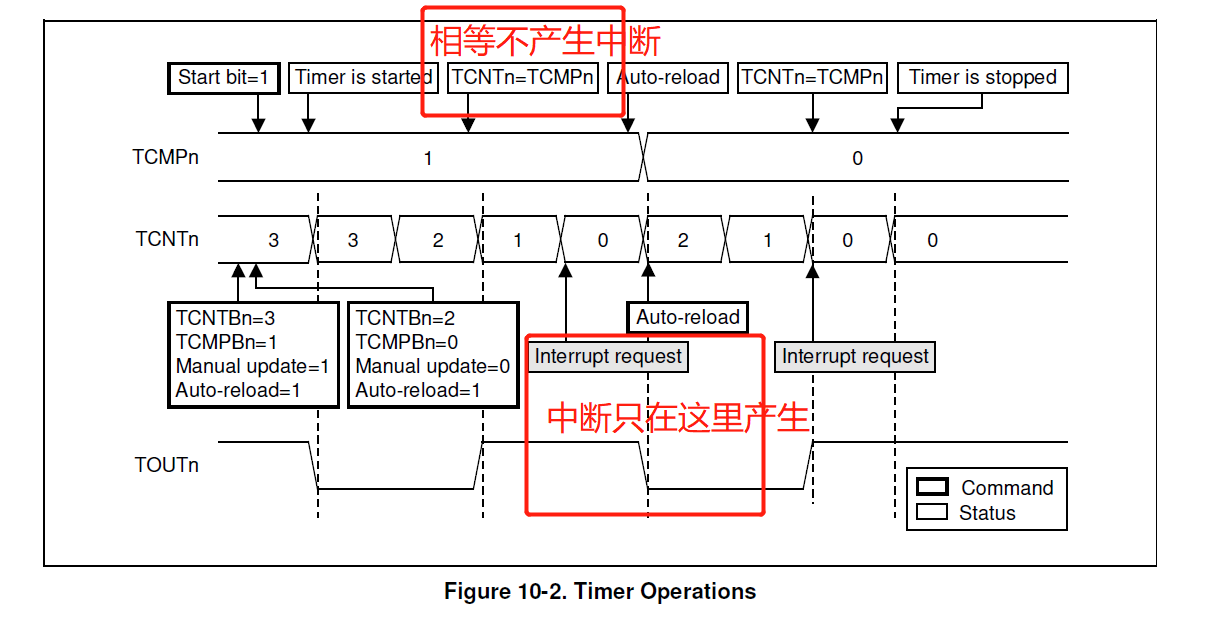


再来看看初始值控制器，用来查看计数值

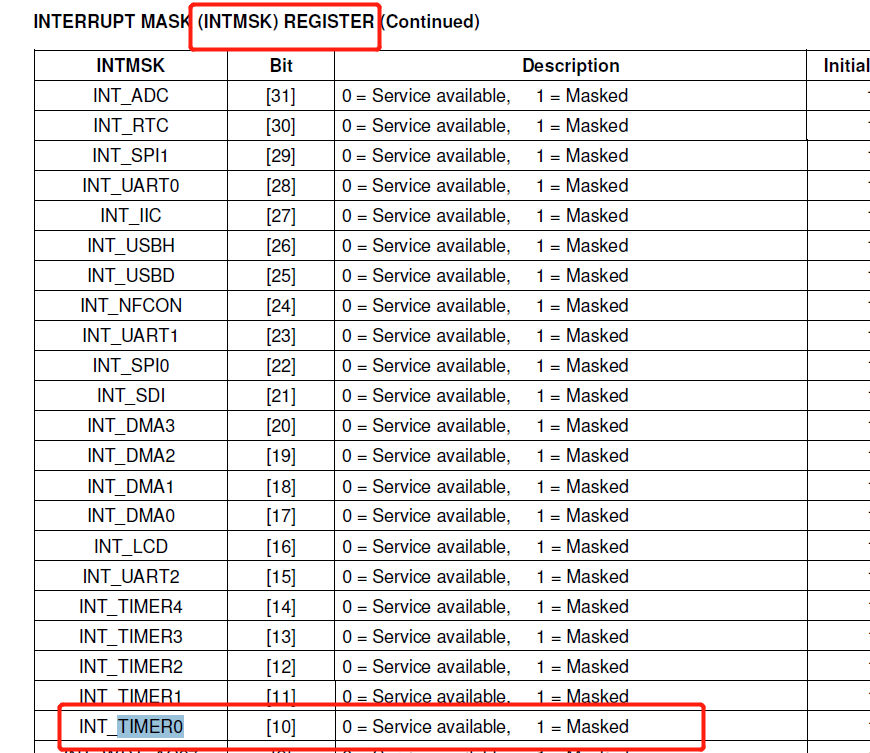
现在可以设置TCON了



这边设置完了，再提供一个中断处理函数 void timer\_irq(void)



初始化中断控制器



链接编译没有问题，没添加一个中断函数，就得修改handle\_irq，太麻烦了，提供一个注册函数，下面进行完善程序

首先定义一个指针数组

type void (\*irq\_func)(int)

这个指针数组放有各个指针的处理函数

irq\_func irq\_array[32]

当初始化按键中断时，给按键中断注册函数

register\_irq(irq,key\_eint\_key)

在handle\_irq里面，直接调用对应处理函数

irq\_array[bit](bit)