**1.什么是中断？**

比如你正在工作，老板打电话突然叫你，这个时候你得放下手上的工作，先去老板办公室做事情。回来之后就又继续工作。你去老板办公室处理事情的过程就是中断服务过程，老板叫你，你听到了，那就是中断。你听到了，你可以不去，那就是你把老板的话屏蔽掉了，这就是可屏蔽中断，如果你得必须去老板办公室，那这就是不可屏蔽的中断了。那么如果你去老板办公室做事情去了，回来之后忘了之前想要干的事情，这下就可怕了，你可能会不再做那个事情了，所以去老板办公室之前先将要干的事情保存在脑子里，或者写成文档保存下来，回来之后回想或者看文档你就知道之前进行到哪了，该做什么。这就对应中断响应之前就要保存现场，响应完中断就要恢复现场，然后从之前停下来的地方继续。

**2.什么是异常？**

对SoC来说，发生复位、软中断、中断、快速中断、取指令异常、数据异常等，都是异常，我们可以发现中断其实是异常的一种。

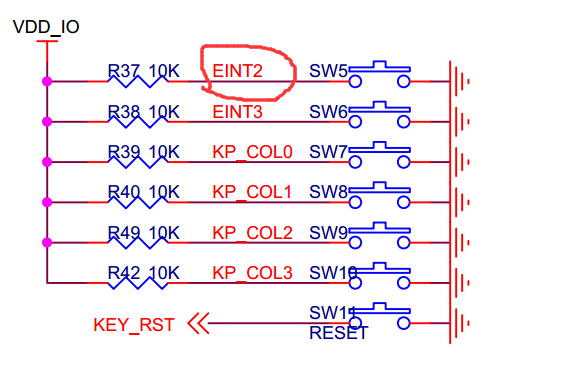
**3.什么是按键？**

按键的物理特性

(1)、平时没人按的时候，弹簧把按键按钮弹开。此时内部断开的。

(2)、有人按下的时候，手的力量克服弹簧的弹力，将按钮按下，此时内部保持接通（闭合）状态；如果手拿开，则弹簧作用下按钮又弹开，同时内部又断开。

按键的电学原理



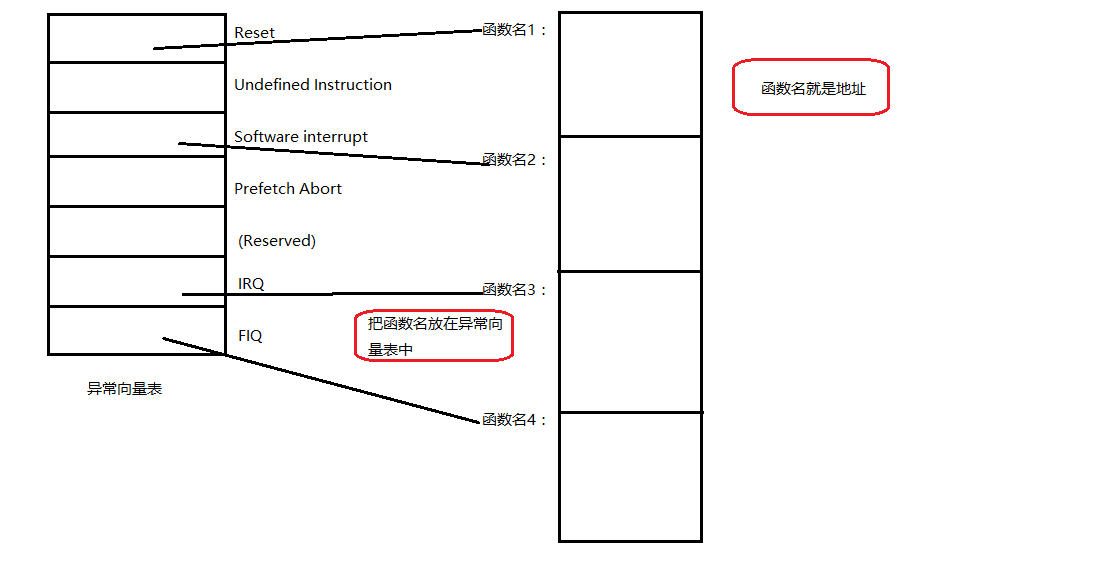
1)硬件接法： SW5:GPH0\_2 SW6:GPH0\_3 SW7:GPH2\_0……

(2)按键的电路连接分析：平时按钮没有按下时，按钮内部断开，GPIO引脚处电压为高电平；当有人按下按钮时，按钮内部导通，外部VDD经过电阻和按钮连接到地，形成回路，此时GPIO引脚处电压就变成了低电平。此时VDD电压全部分压在了电阻上（这个电阻就叫分压电阻，这个电阻不能太小，因为电阻的功率是U\*U/R）

(3)总结：按键的工作方法：其实就是按键的按下与弹开，分别对应GPIO的两种电平状态（按下则GPIO为低电平，弹开则GPIO为高电平）。此时SoC内部可以通过检测这个GPIO的电平高低来判断按键有没有被按下，这个判断结果即可作为SoC的输入信号。

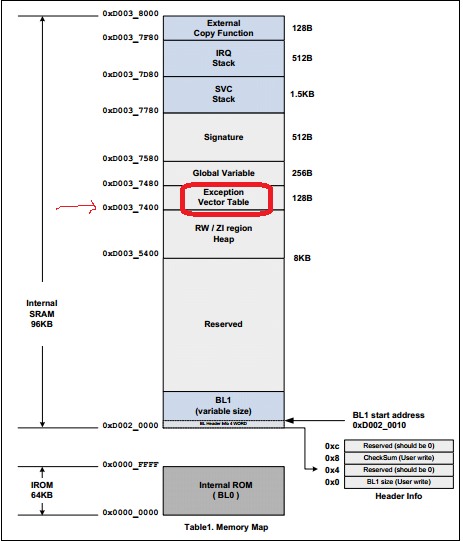
**4.异常向量表**

向量我们在数学中学过，知道就是有方向，可以指向，所以异常向量表就是说一块连续的地址处存放异常函数服务的程序的入口地址。也就是说指向我们异常服务函数的入口地址。 所以我们第一步通常是把我们的服务函数的入口地址绑到向量表上的，这样我们就可以在随意的地方写异常服务程序，只需要将入口地址绑定到异常向量表处就行。如下图：

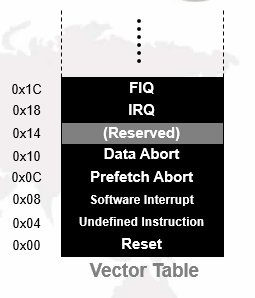


这样之后，我们会想CPU怎么一下就找到相应的异常向量处，执行其对应的中断呢？

其实中断向量表的地址CPU设计人员开始就设计好了，可以这样理解，设计人员设计CPU时候就已经确定把中断向量表的地址定义在哪里。所以发生中断后，CPU 的PC指针自动会指向中断向量表处，然后找到你绑定的异常服务程序的入口地址，进而跳转到异常服务程序处执行我们自己写的服务程序。注意：这个过程都是硬件自动完成的（是由硬件决定的）。查看手册可以看到中断向量表地址在0xD0037400处。如下图：



我们也可以知道那些异常的相对位置如下：

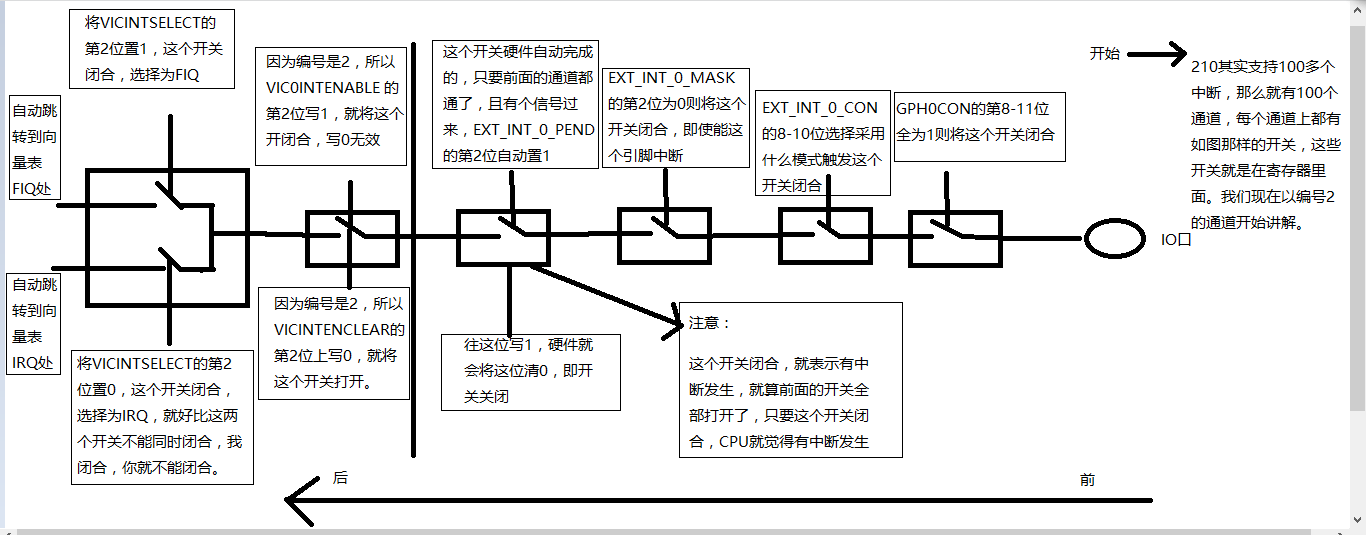


知道这些后，我们首先将我们自己写中断的服务程序绑定在IRQ向量处。int.c内容如下：

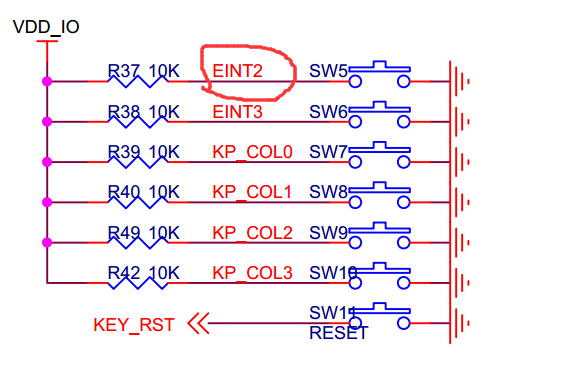


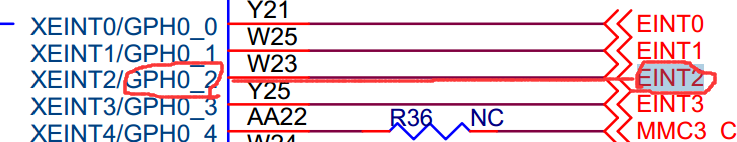
这样在main函数调用system\_init\_exception()就完成绑定了。然后我们编译连接之后烧到板子里，实验发现并没有执行。大家可以想想为什么？那就是我们怎么就让中断发生呢？

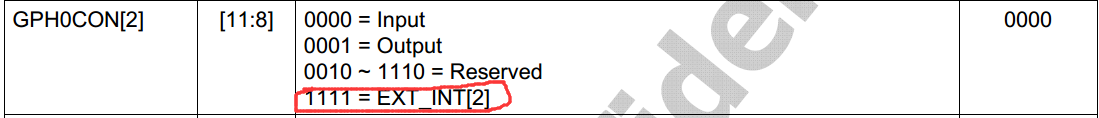
我画一个图可以让大家明白中断怎么就可以发生:

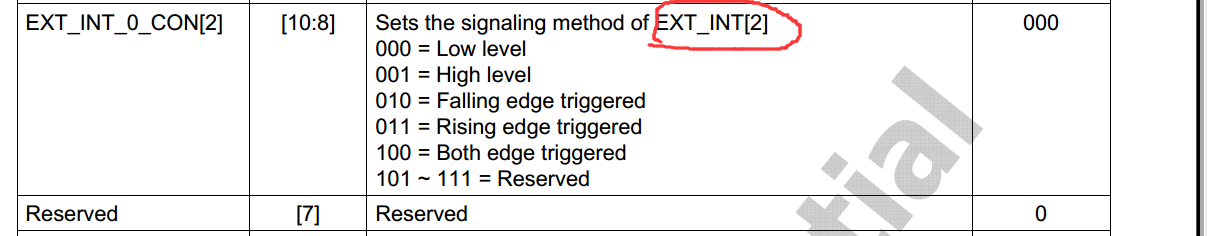


关于图上的IO口和寄存器每个位的详细说明如下：

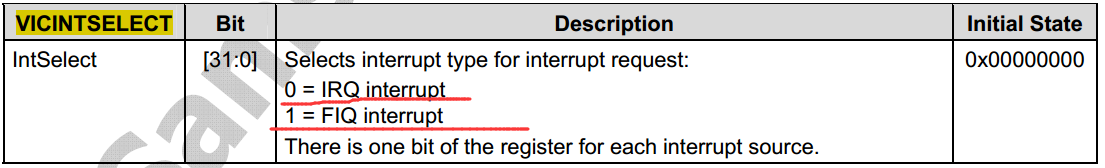
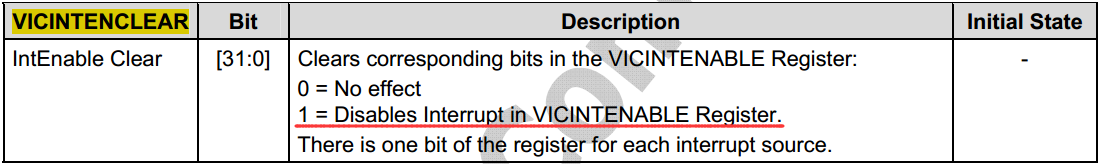
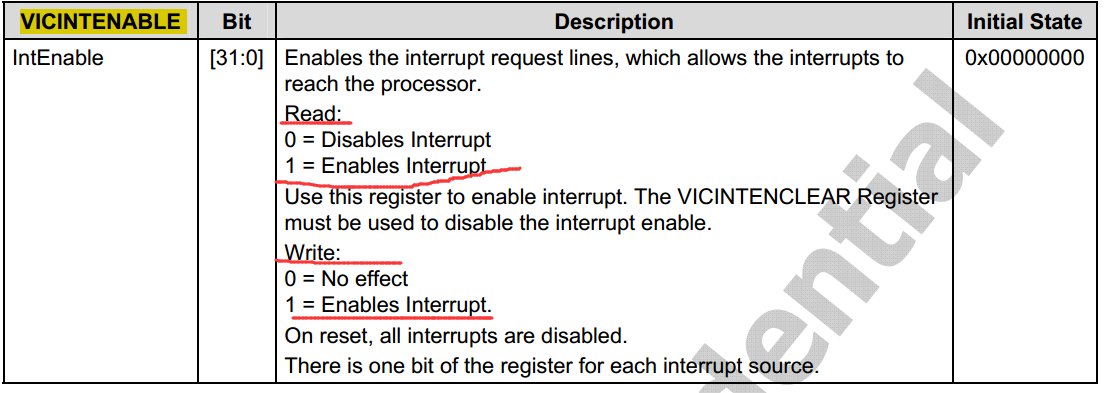
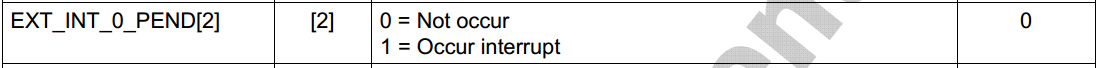




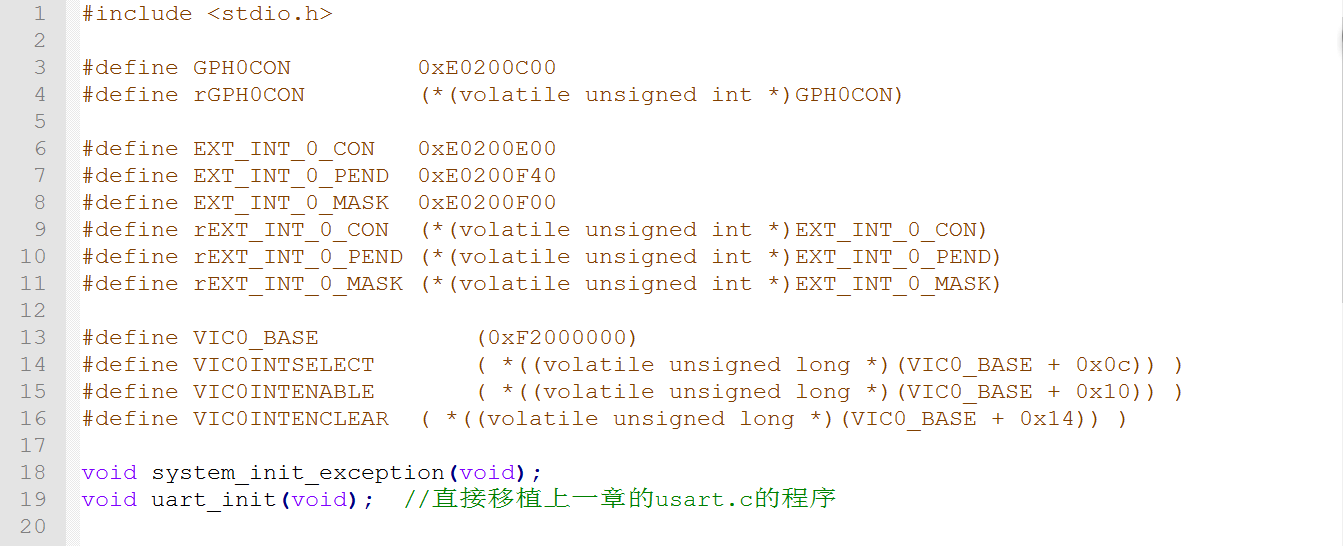


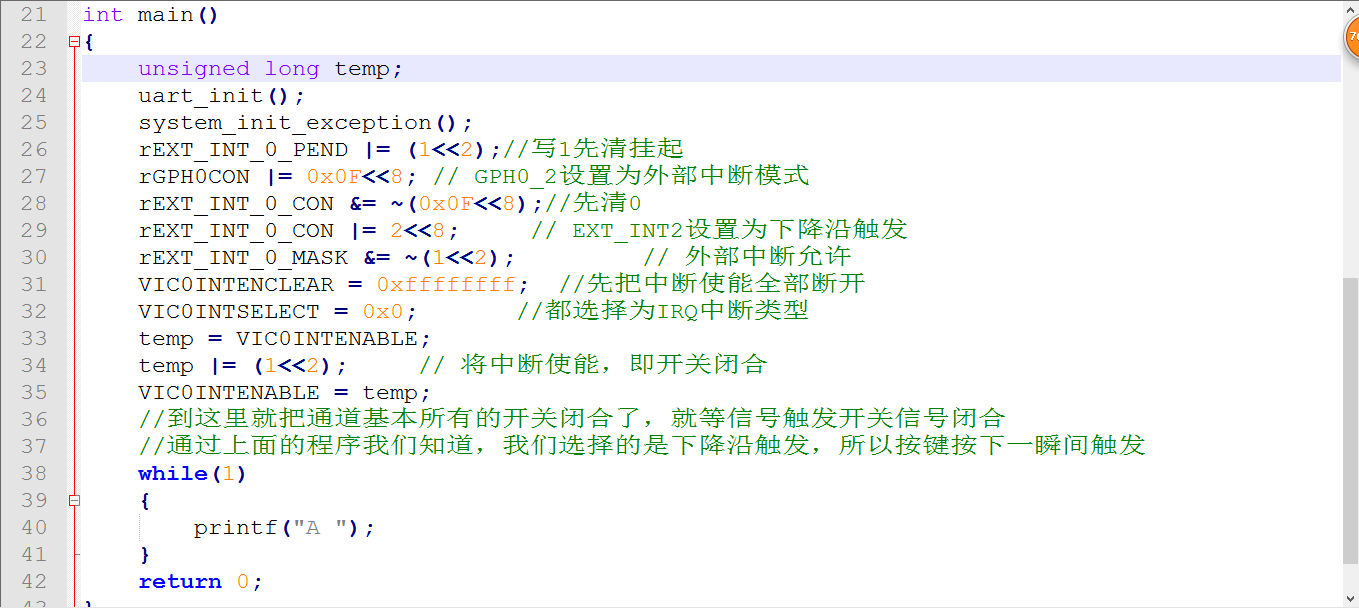




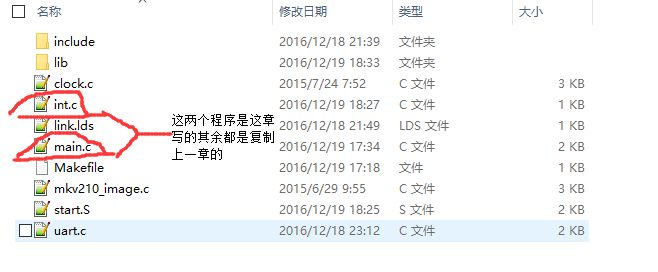


现在就来实现下中断：main.c文件内容如下：

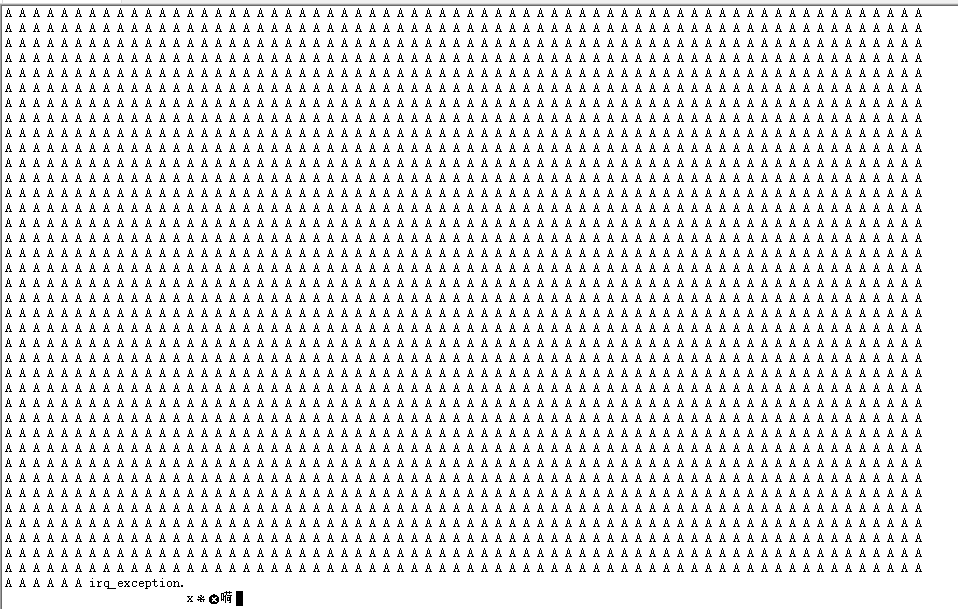




在这说明下这里面用到usart.c，clock.c，start.S，Makefile都是直接使用前面一章的程序，而int.c就是开始绑定的那个程序，mian.c就是上面那个程序。给大家截下目录下的文件如下：



这个时候下载验证下结果如下图：



正常的时候一直打印A,然后我将按键按下的时候，打印irq\_exception.\n。从图上也可以看到，但是问题来了，那就是打印完就不动了，没有返回回去正常打印A。其实这就要涉及到中断的时候保存现场，中断完之后恢复现场等机制了。

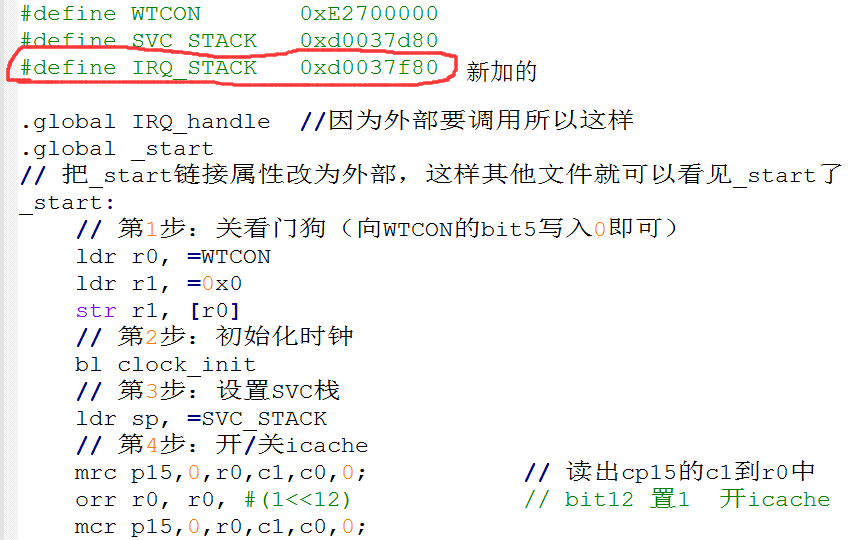
* 当异常产生时, ARM core:（硬件自动完成）
  + 拷贝 CPSR 到 SPSR\_<mode>
  + 设置适当的 CPSR 位：
    - 改变处理器状态进入 ARM 态
    - 改变处理器模式进入相应的异常模式
    - 设置中断禁止位禁止相应中断 (如果需要)
  + 保存返回地址到 LR\_<mode>
  + 设置 PC 为相应的异常向量

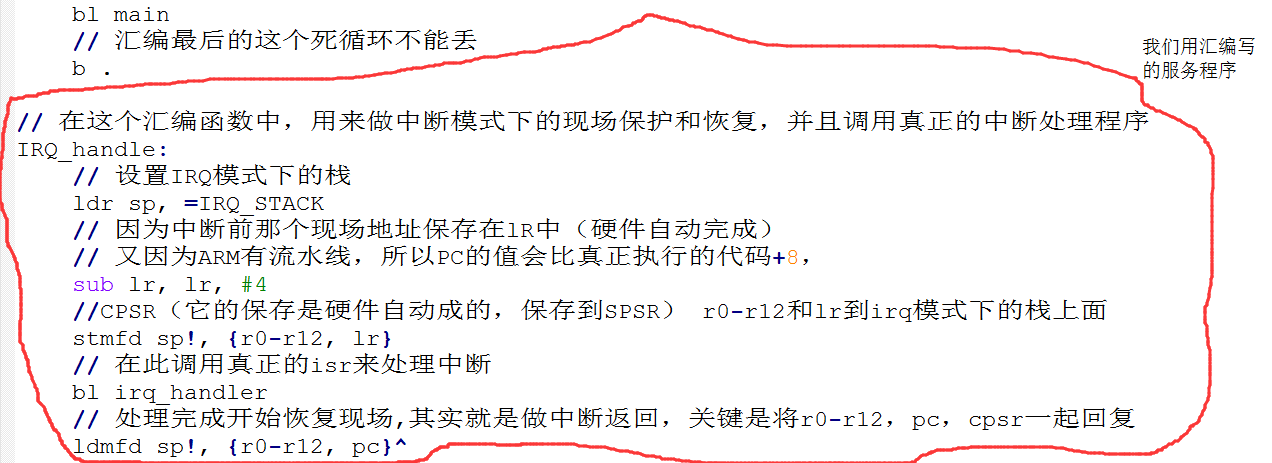
/\*这句话其实我们应该就纠正前面的观点了，其实发生中断响应了，不是说我们PC指针跑到异常向量表处，从而跳转到我们服务程序入口处。而是中断响应到来，硬件直接把相应的中断向量处存放的入口地址直接赋给PC指针，从而跳转到我们服务程序处，前面那样说只是让大家好理解。所以看到这里就重新把思路改回来。还有就是前面那种思路在有些单片机就是采用的，51就是一个例子。\*/

* 返回时, 异常处理需要:（这是需要我们程序自己处理）
  + 从 SPSR\_<mode>恢复CPSR
  + 从LR\_<mode>恢复PC
  + Note:这些操作只能在 ARM 态执行.

因为我们要保存现场，所以我们需要使用汇编来写我的IRQ的服务程序

具体修改移植朱老师的。如下：



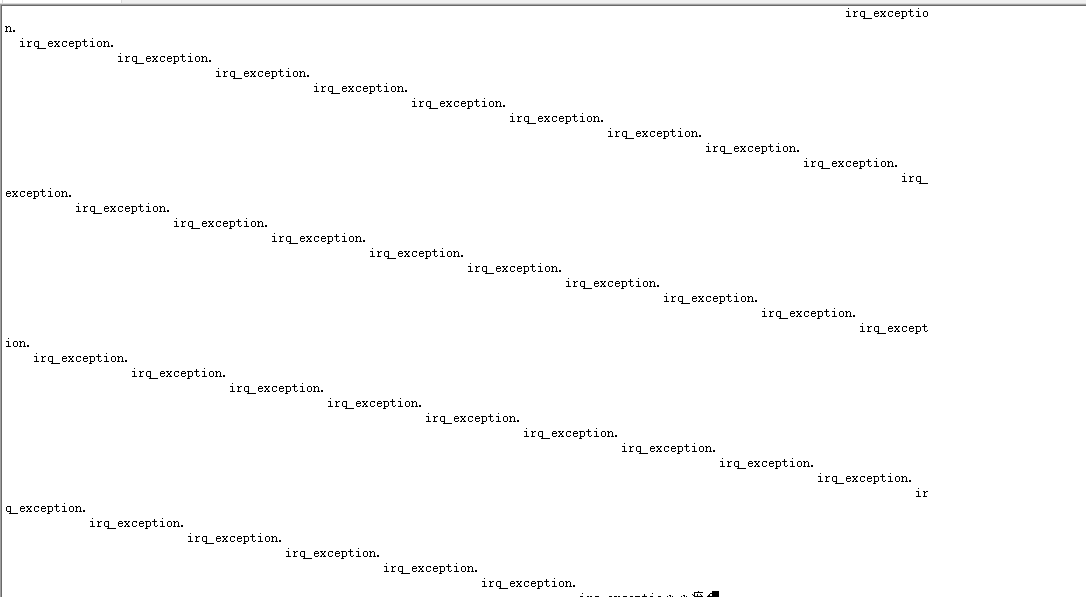


接下来修改int.c中的程序：



所以接下来程序应该这样跑，上电之后就一直打A,然后我按下按键，PC就跑到我们汇编写的服务程序处，然后保存现场，调用irq\_handler()函数，打印irq\_exception.\n然后返回到我们汇编写的服务程序，接着就是恢复现场，PC指针又跑到主程序里面不断的打印A。

好，现在下载验证一下：

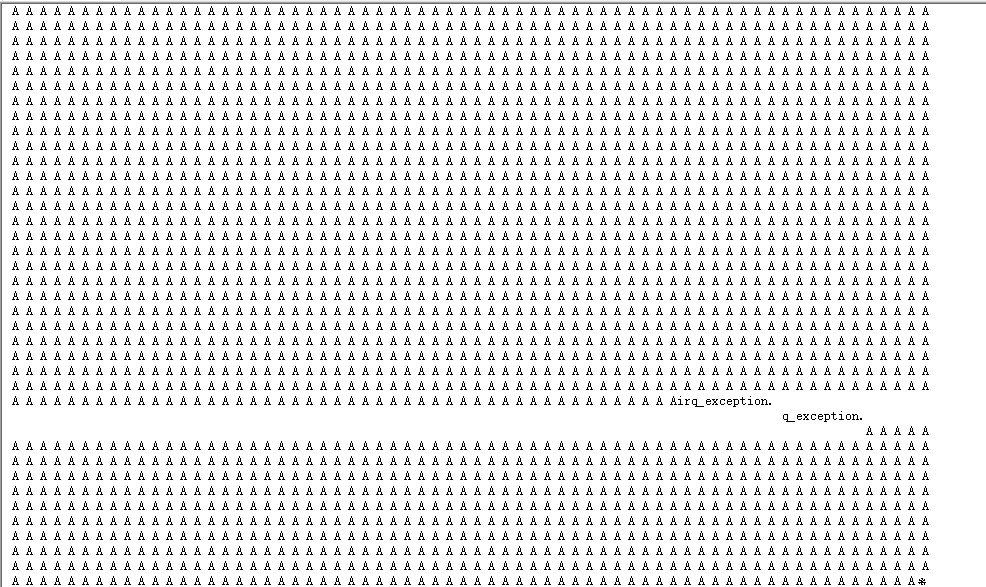


从图上我们可以看到我们的中断一直在发生，这是为什么呢？

其实仔细瞧瞧上面我画的那个图就知道，那个EXT\_INT\_0\_PEND寄存器每位都是硬件自动置1，可恶的是硬件不会自动清0，必须我们自己清0，所以说我们响应一次中断如果不清零，CPU还会认为你触发了中断，然后不断的执行中断服务程序。所以我现在需要修改，在int.c中修改如下：

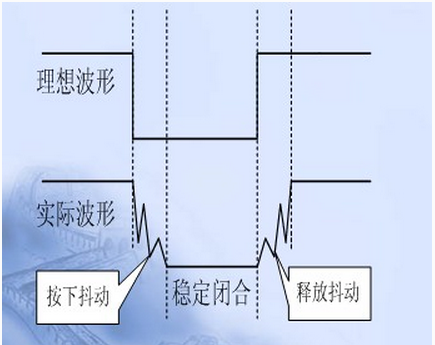


下载验证结果如下图：



从图上我们发现这下恢复正常了，但是又有一个问题出现，就是我按了一下，CPU怎么进入了两次中断呢？

其实这就是机械按键有抖动的缘故,具体请看下图：

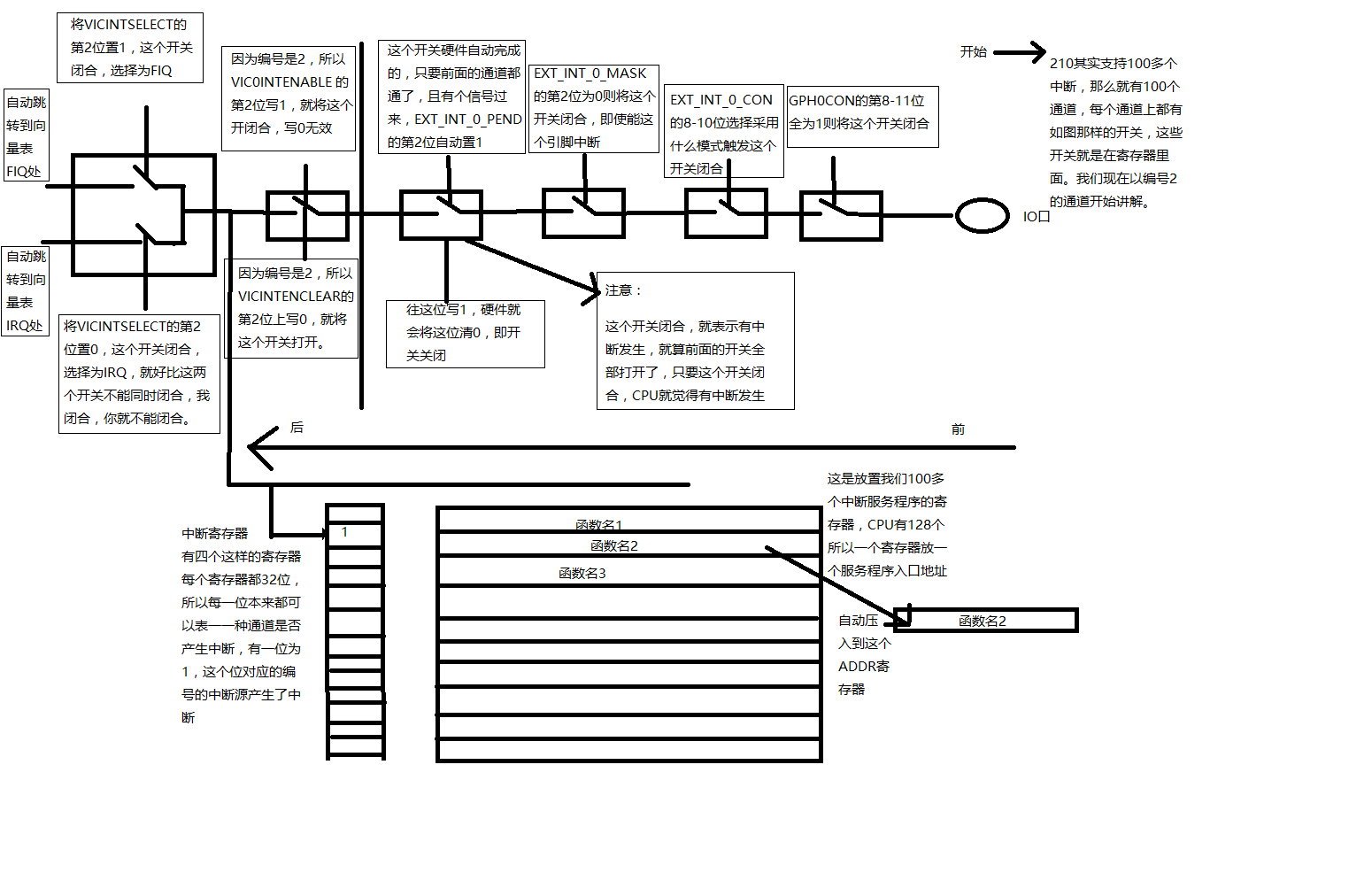


所以现在出现两种消抖的方法，第一种就是硬件消抖，第二种就是软件消抖

关于这两种消抖的具体实现就不说了，在网上很容易的找到关于消抖的讲解。

到现在你可能问我，210不是支持100多个中断吗？现在看来不就一个吗？

其实CPU设计者是这样设计的，如下：



图上我大致说明了一下，大家可以结合接下来的文字来综合理解。每个中断通道都和上图相似，也就是说每个通道的中断源发生了相应的中断都会让PC的值=IRQ异常向量表处放的地址，也就是说都要执行我们用汇编写的服务程序，我们在服务程序内部调用一个函数，这个函数找到对应的VICnADDR寄存器（在图上就是那个ADDR寄存器）VICnADDR中的n其实就是0,1,2,3。因为CPU设计者将状态寄存器（VICnIRQSTATUS）分为四个，那个放入口地址的寄存器（VICnVECTADDR0～VICnVECTADDR31）有32\*4个，32个为一组。所以分为四组。VIC0IRQSTATUS状态寄存器对应VIC0VECTADDR0～VIC0VECTADDR31这组，这组对应VIC0ADDR

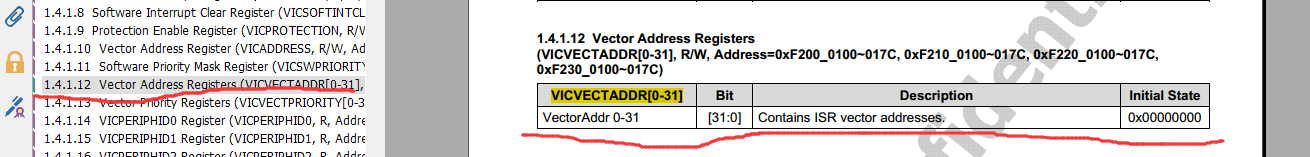
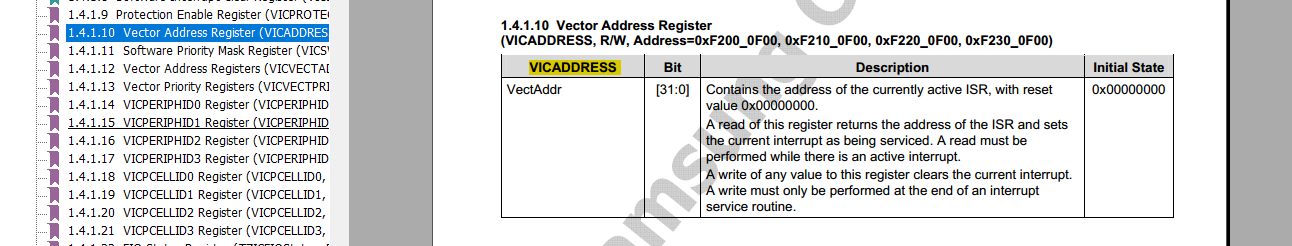
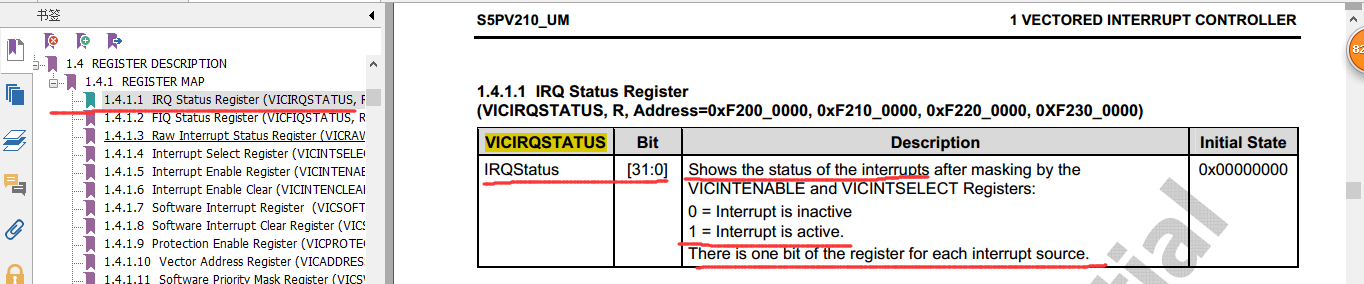
VIC1IRQSTATUS状态寄存器对应VIC1VECTADDR0～VIC1VECTADDR31这组，这组对应VIC1ADDR

VIC2IRQSTATUS状态寄存器对应VIC2VECTADDR0～VIC2VECTADDR31这组，这组对应VIC2ADDR

VIC3IRQSTATUS状态寄存器对应VIC3VECTADDR0～VIC3VECTADDR31这组，这组对应VIC3ADDR

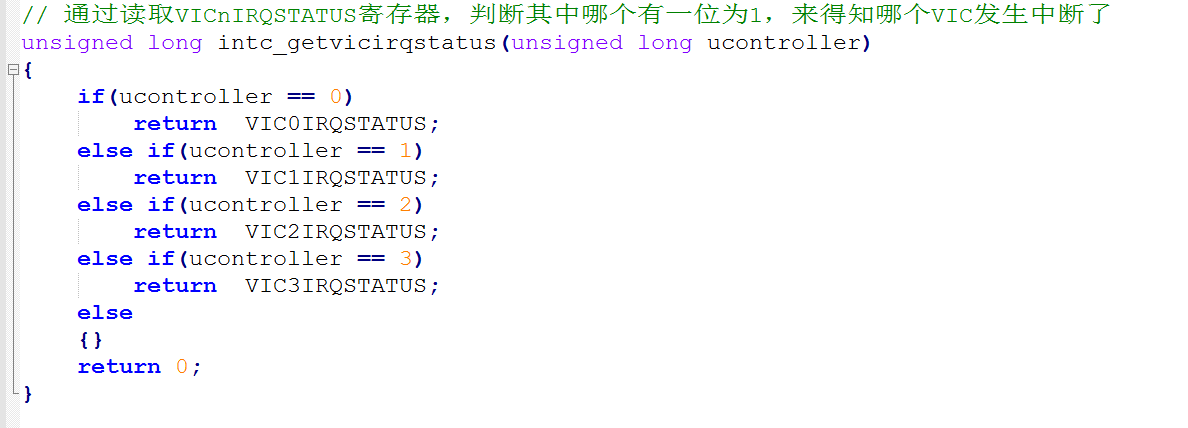
第一组对应编号0-31，第二组对应编号32-63，第三组对应编号 64-95，第五组对应编号96-127。所以我们会把相应编号的服务程序入口地址放到相应的放入口地址程序的寄存器中，只要发生中断，就会在相应编号的状态寄存器那位置1，将对应的入口地址压入VICnADDR寄存器中。这一过程是自动的，硬件决定的。

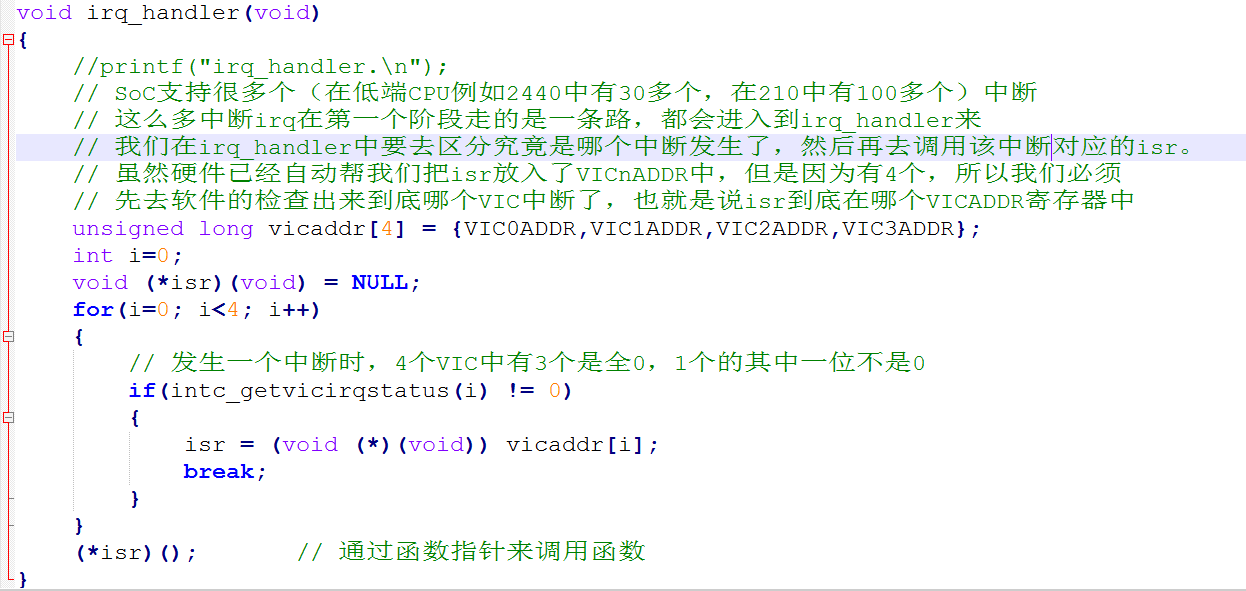
这些寄存器的在S5PV210\_UM\_REV1.1手册找到，如下：



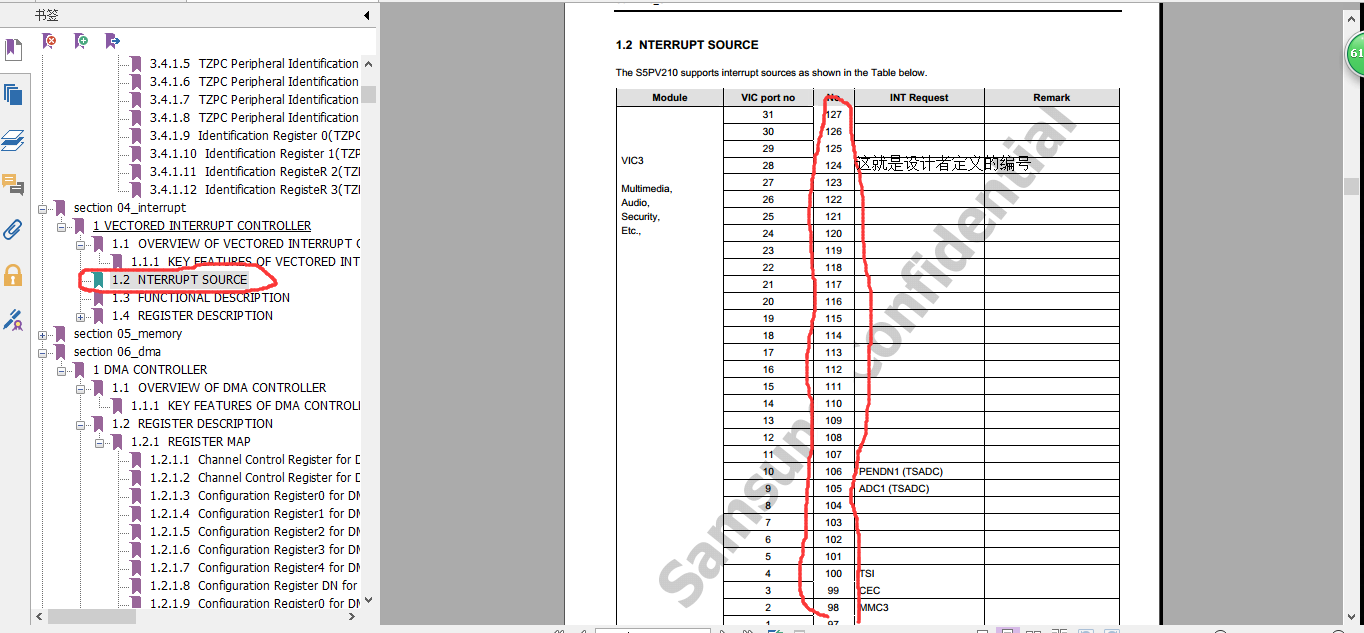
所以我们利用这样的机制，我们在那个函数里可以这样找到对应的VICnADDR寄存器

如下：





其实CPU设计者已经将这些中断源了号，查看S5PV210\_UM\_REV1.1手册，可以清楚看到他们编的号，如下：



我只列出一部分，读者可以需要自行查看。

为了方便把各中断源真正的服务程序的入口地址按号放置，所以定义很多编号的宏。

所以贴出完整的int.c程序，根据编号就能打开相应的中断，根据编号将服务程序入口地址放入相应的寄存器中。程序如下：

**int.c文件**

#include "int.h"

#include "stdio.h"

// 主要功能：绑定第一阶段异常向量表；禁止所有中断；选择所有中断类型为IRQ；

// 清除VICnADDR为0

void system\_init\_exception(void)

{

// 第一阶段处理，绑定异常向量表

r\_exception\_irq = (unsigned int)IRQ\_handle;

// 初始化中断控制器的基本寄存器

intc\_init();

}

// 清除需要处理的中断的中断处理函数的地址

void intc\_clearvectaddr(void)

{

// VICxADDR:当前正在处理的中断的中断处理函数的地址

VIC0ADDR = 0;

VIC1ADDR = 0;

VIC2ADDR = 0;

VIC3ADDR = 0;

}

// 初始化中断控制器

void intc\_init(void)

{

// 禁止所有中断

// 为什么在中断初始化之初要禁止所有中断？

// 因为中断一旦打开，因为外部或者硬件自己的原因产生中断后一定就会寻找isr

// 而我们可能认为自己用不到这个中断就没有提供isr，这时它自动拿到的就是乱码

// 则程序很可能跑飞，所以不用的中断一定要关掉。

// 一般的做法是先全部关掉，然后再逐一打开自己感兴趣的中断。一旦打开就必须

// 给这个中断提供相应的isr并绑定好。

VIC0INTENCLEAR = 0xffffffff;

VIC1INTENCLEAR = 0xffffffff;

VIC2INTENCLEAR = 0xffffffff;

VIC3INTENCLEAR = 0xffffffff;

// 选择中断类型为IRQ

VIC0INTSELECT = 0x0;

VIC1INTSELECT = 0x0;

VIC2INTSELECT = 0x0;

VIC3INTSELECT = 0x0;

// 清VICxADDR

intc\_clearvectaddr();

}

// 绑定我们写的isr到VICnVECTADDR寄存器

// 绑定过之后我们就把isr地址交给硬件了，剩下的我们不用管了，硬件自己会处理

// 等发生相应中断的时候，我们直接到相应的VICnADDR中去取isr地址即可。

// 参数：intnum是int.h定义的物理中断号，handler是函数指针，就是我们写的isr

// VIC0VECTADDR定义为VIC0VECTADDR0寄存器的地址，就相当于是VIC0VECTADDR0～31这//个 数组（这个数组就是一个函数指针数组）的首地址，然后具体计算每一个中断的时候

// 只需要首地址+偏移量即可。

void intc\_setvectaddr(unsigned long intnum, void (\*handler)(void))

{

//VIC0

if(intnum<32)

{

\*( (volatile unsigned long \*)(VIC0VECTADDR + 4\*(intnum-0)) ) = (unsigned)handler;

}

//VIC1

else if(intnum<64)

{

\*( (volatile unsigned long \*)(VIC1VECTADDR + 4\*(intnum-32)) ) = (unsigned)handler;

}

//VIC2

else if(intnum<96)

{

\*( (volatile unsigned long \*)(VIC2VECTADDR + 4\*(intnum-64)) ) = (unsigned)handler;

}

//VIC3

else

{

\*( (volatile unsigned long \*)(VIC3VECTADDR + 4\*(intnum-96)) ) = (unsigned)handler;

}

return;

}

// 使能中断

// 通过传参的intnum来使能某个具体的中断源，中断号在int.h中定义，是物理中断号

void intc\_enable(unsigned long intnum)

{

unsigned long temp;

// 确定intnum在哪个寄存器的哪一位

// <32就是0～31，必然在VIC0

if(intnum<32)

{

temp = VIC0INTENABLE;

temp |= (1<<intnum); // 如果是第一种设计则必须位操作，第二种设计可以

// 直接写。

VIC0INTENABLE = temp;

}

else if(intnum<64)

{

temp = VIC1INTENABLE;

temp |= (1<<(intnum-32));

VIC1INTENABLE = temp;

}

else if(intnum<96)

{

temp = VIC2INTENABLE;

temp |= (1<<(intnum-64));

VIC2INTENABLE = temp;

}

else if(intnum<NUM\_ALL)

{

temp = VIC3INTENABLE;

temp |= (1<<(intnum-96));

VIC3INTENABLE = temp;

}

// NUM\_ALL : enable all interrupt

else

{

VIC0INTENABLE = 0xFFFFFFFF;

VIC1INTENABLE = 0xFFFFFFFF;

VIC2INTENABLE = 0xFFFFFFFF;

VIC3INTENABLE = 0xFFFFFFFF;

}

}

// 禁止中断

// 通过传参的intnum来禁止某个具体的中断源，中断号在int.h中定义，是物理中断号

void intc\_disable(unsigned long intnum)

{

unsigned long temp;

if(intnum<32)

{

temp = VIC0INTENCLEAR;

temp |= (1<<intnum);

VIC0INTENCLEAR = temp;

}

else if(intnum<64)

{

temp = VIC1INTENCLEAR;

temp |= (1<<(intnum-32));

VIC1INTENCLEAR = temp;

}

else if(intnum<96)

{

temp = VIC2INTENCLEAR;

temp |= (1<<(intnum-64));

VIC2INTENCLEAR = temp;

}

else if(intnum<NUM\_ALL)

{

temp = VIC3INTENCLEAR;

temp |= (1<<(intnum-96));

VIC3INTENCLEAR = temp;

}

// NUM\_ALL : disable all interrupt

else

{

VIC0INTENCLEAR = 0xFFFFFFFF;

VIC1INTENCLEAR = 0xFFFFFFFF;

VIC2INTENCLEAR = 0xFFFFFFFF;

VIC3INTENCLEAR = 0xFFFFFFFF;

}

return;

}

// 通过读取VICnIRQSTATUS寄存器，判断其中哪个有一位为1，来得知哪个VIC发生中断了

unsigned long intc\_getvicirqstatus(unsigned long ucontroller)

{

if(ucontroller == 0)

return VIC0IRQSTATUS;

else if(ucontroller == 1)

return VIC1IRQSTATUS;

else if(ucontroller == 2)

return VIC2IRQSTATUS;

else if(ucontroller == 3)

return VIC3IRQSTATUS;

else

{}

return 0;

}

// 真正的中断处理程序。意思就是说这里只考虑中断处理，不考虑保护/恢复现场

void irq\_handler(void)

{

// SoC支持很多个（在低端CPU例如2440中有30多个，在210中有100多个）中断

// 这么多中断irq在第一个阶段走的是一条路，都会进入到irq\_handler来

// 我们在irq\_handler中要去区分究竟是哪个中断发生了，然后再去调用该中断

// 对应的isr。

// 虽然硬件已经自动帮我们把isr放入了VICnADDR中，但是因为有4个，所以我们必

//须先去软件的检查出来到底哪个VIC中断了，也就是说isr到底在哪个VICADDR寄存//器中

unsigned long vicaddr[4] = {VIC0ADDR,VIC1ADDR,VIC2ADDR,VIC3ADDR};

int i=0;

void (\*isr)(void) = NULL;

for(i=0; i<4; i++)

{

// 发生一个中断时，4个VIC中有3个是全0，1个的其中一位不是0

if(intc\_getvicirqstatus(i) != 0)

{

isr = (void (\*)(void)) vicaddr[i];

break;

}

}

(\*isr)(); // 通过函数指针来调用函数

}

**int.h文件放声明,各寄存器空间宏定义和硬件编号宏**

#ifndef \_\_INT\_H\_\_

#define \_\_INT\_H\_\_

void intc\_init(void);

void intc\_enable(unsigned long intnum);

void intc\_disable(unsigned long intnum);

void intc\_setvectaddr(unsigned long intnum, void (\*handler)(void));

void intc\_clearvectaddr(void);

unsigned long intc\_getvicirqstatus(unsigned long ucontroller);

void irq\_handler(void);

void IRQ\_handle(void);

void system\_init\_exception(void);

//// Interrupt

#define VIC0\_BASE (0xF2000000)

#define VIC1\_BASE (0xF2100000)

#define VIC2\_BASE (0xF2200000)

#define VIC3\_BASE (0xF2300000)

// VIC0

#define VIC0IRQSTATUS ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x00)) )

#define VIC0FIQSTATUS ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x04)) )

#define VIC0INTSELECT ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x0c)) )

#define VIC0INTENABLE ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x10)) )

#define VIC0INTENCLEAR ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x14)) )

#define VIC0VECTADDR (VIC0\_BASE + 0x100)

#define VIC0ADDR ( \*((volatile unsigned long \*)(VIC0\_BASE + 0xf00)) )

// VIC1

#define VIC1IRQSTATUS ( \*((volatile unsigned long \*)(VIC1\_BASE + 0x00)) )

#define VIC1FIQSTATUS ( \*((volatile unsigned long \*)(VIC1\_BASE + 0x04)) )

#define VIC1INTSELECT ( \*((volatile unsigned long \*)(VIC1\_BASE + 0x0c)) )

#define VIC1INTENABLE ( \*((volatile unsigned long \*)(VIC1\_BASE + 0x10)) )

#define VIC1INTENCLEAR ( \*((volatile unsigned long \*)(VIC1\_BASE + 0x14)) )

#define VIC1VECTADDR (VIC1\_BASE + 0x100)

#define VIC1ADDR ( \*((volatile unsigned long \*)(VIC1\_BASE + 0xf00)) )

// VIC2

#define VIC2IRQSTATUS ( \*((volatile unsigned long \*)(VIC2\_BASE + 0x00)) )

#define VIC2FIQSTATUS ( \*((volatile unsigned long \*)(VIC2\_BASE + 0x04)) )

#define VIC2INTSELECT ( \*((volatile unsigned long \*)(VIC2\_BASE + 0x0c)) )

#define VIC2INTENABLE ( \*((volatile unsigned long \*)(VIC2\_BASE + 0x10)) )

#define VIC2INTENCLEAR ( \*((volatile unsigned long \*)(VIC2\_BASE + 0x14)) )

#define VIC2VECTADDR (VIC2\_BASE + 0x100)

#define VIC2ADDR ( \*((volatile unsigned long \*)(VIC2\_BASE + 0xf00)) )

// VIC3

#define VIC3IRQSTATUS ( \*((volatile unsigned long \*)(VIC3\_BASE + 0x00)) )

#define VIC3FIQSTATUS ( \*((volatile unsigned long \*)(VIC3\_BASE + 0x04)) )

#define VIC3INTSELECT ( \*((volatile unsigned long \*)(VIC3\_BASE + 0x0c)) )

#define VIC3INTENABLE ( \*((volatile unsigned long \*)(VIC3\_BASE + 0x10)) )

#define VIC3INTENCLEAR ( \*((volatile unsigned long \*)(VIC3\_BASE + 0x14)) )

#define VIC3VECTADDR (VIC3\_BASE + 0x100)

#define VIC3ADDR ( \*((volatile unsigned long \*)(VIC3\_BASE + 0xf00)) )

#define exception\_vector\_table\_base 0xD0037400

#define exception\_reset (exception\_vector\_table\_base + 0x00)

#define exception\_undef (exception\_vector\_table\_base + 0x04)

#define exception\_sotf\_int (exception\_vector\_table\_base + 0x08)

#define exception\_prefetch (exception\_vector\_table\_base + 0x0C)

#define exception\_data (exception\_vector\_table\_base + 0x10)

#define exception\_irq (exception\_vector\_table\_base + 0x18)

#define exception\_fiq (exception\_vector\_table\_base + 0x1C)

#define r\_exception\_reset (\*(volatile unsigned int \*)exception\_reset)

#define r\_exception\_undef (\*(volatile unsigned int \*)exception\_undef)

#define r\_exception\_sotf\_int (\*(volatile unsigned int \*)exception\_sotf\_int)

#define r\_exception\_prefetch (\*(volatile unsigned int \*)exception\_prefetch)

#define r\_exception\_data (\*(volatile unsigned int \*)exception\_data)

#define r\_exception\_irq (\*(volatile unsigned int \*)exception\_irq)

#define r\_exception\_fiq (\*(volatile unsigned int \*)exception\_fiq)

// 中断源编号

#define INT\_LIMIT (96)

//INT NUM - VIC0

#define NUM\_EINT0 (0)

#define NUM\_EINT1 (1)

#define NUM\_EINT2 (2)

#define NUM\_EINT3 (3)

#define NUM\_EINT4 (4)

#define NUM\_EINT5 (5)

#define NUM\_EINT6 (6)

#define NUM\_EINT7 (7)

#define NUM\_EINT8 (8)

#define NUM\_EINT9 (9)

#define NUM\_EINT10 (10)

#define NUM\_EINT11 (11)

#define NUM\_EINT12 (12)

#define NUM\_EINT13 (13)

#define NUM\_EINT14 (14)

#define NUM\_EINT15 (15)

#define NUM\_EINT16\_31 (16)

#define NUM\_Reserved17 (17)

#define NUM\_MDMA (18)

#define NUM\_PDMA0 (19)

#define NUM\_PDMA1 (20)

#define NUM\_TIMER0 (21)

#define NUM\_TIMER1 (22)

#define NUM\_TIMER2 (23)

#define NUM\_TIMER3 (24)

#define NUM\_TIMER4 (25)

#define NUM\_SYSTIMER (26)

#define NUM\_WDT (27)

#define NUM\_RTC\_ALARM (28)

#define NUM\_RTC\_TICK (29)

#define NUM\_GPIOINT (30)

#define NUM\_FIMC3 (31)

//INT NUM - VIC1

#define NUM\_CORTEX0 (32+0)

#define NUM\_CORTEX1 (32+1)

#define NUM\_CORTEX2 (32+2)

#define NUM\_CORTEX3 (32+3)

#define NUM\_CORTEX4 (32+4)

#define NUM\_IEM\_APC (32+5)

#define NUM\_IEM\_IEC (32+6)

#define NUM\_Reserved39 (32+7)

#define NUM\_NFC (32+8)

#define NUM\_CFC (32+9)

#define NUM\_UART0 (32+10)

#define NUM\_UART1 (32+11)

#define NUM\_UART2 (32+12)

#define NUM\_UART3 (32+13)

#define NUM\_I2C (32+14)

#define NUM\_SPI0 (32+15)

#define NUM\_SPI1 (32+16)

#define NUM\_SPI2 (32+17)

#define NUM\_AUDIO (32+18)

#define NUM\_I2C\_PMIC (32+19)

#define NUM\_I2C\_HDMI (32+20)

#define NUM\_HSIRX (32+21)

#define NUM\_HSITX (32+22)

#define NUM\_UHOST (32+23)

#define NUM\_OTG (32+24)

#define NUM\_MSM (32+25)

#define NUM\_HSMMC0 (32+26)

#define NUM\_HSMMC1 (32+27)

#define NUM\_HSMMC2 (32+28)

#define NUM\_MIPI\_CSI (32+29)

#define NUM\_MIPI\_DSI (32+30)

#define NUM\_ONENAND\_AUDI (32+31)

//INT NUM - VIC2

#define NUM\_LCD0 (64+0)

#define NUM\_LCD1 (64+1)

#define NUM\_LCD2 (64+2)

#define NUM\_LCD3 (64+3)

#define NUM\_ROTATOR (64+4)

#define NUM\_FIMC\_A (64+5)

#define NUM\_FIMC\_B (64+6)

#define NUM\_FIMC\_C (64+7)

#define NUM\_JPEG (64+8)

#define NUM\_2D (64+9)

#define NUM\_3D (64+10)

#define NUM\_MIXER (64+11)

#define NUM\_HDMI (64+12)

#define NUM\_HDMI\_I2C (64+13)

#define NUM\_MFC (64+14)

#define NUM\_TVENC (64+15)

#define NUM\_I2S0 (64+16)

#define NUM\_I2S1 (64+17)

#define NUM\_I2S2 (64+18)

#define NUM\_AC97 (64+19)

#define NUM\_PCM0 (64+20)

#define NUM\_PCM1 (64+21)

#define NUM\_SPDIF (64+22)

#define NUM\_ADC (64+23)

#define NUM\_PENDN (64+24)

#define NUM\_KEYPAD (64+25)

#define NUM\_Reserved90 (64+26)

#define NUM\_HASH (64+27)

#define NUM\_FEEDCTRL (64+28)

#define NUM\_PCM2 (64+29)

#define NUM\_SDM\_IRQ (64+30)

#define NUM\_SMD\_FIQ (64+31)

//INT NUM - VIC3

#define NUM\_IPC (96+0)

#define NUM\_HOSTIF (96+1)

#define NUM\_HSMMC3 (96+2)

#define NUM\_CEC (96+3)

#define NUM\_TSI (96+4)

#define NUM\_MDNIE0 (96+5)

#define NUM\_MDNIE1 (96+6)

#define NUM\_MDNIE2 (96+7)

#define NUM\_MDNIE3 (96+8)

#define NUM\_ADC1 (96+9)

#define NUM\_PENDN1 (96+10)

#define NUM\_ALL (200)

#endif

**接下来就是修改main.c了，main..c文件程序如下：**

#include <stdio.h>

#include "int.h"

#define GPH0CON 0xE0200C00

#define rGPH0CON (\*(volatile unsigned int \*)GPH0CON)

#define EXT\_INT\_0\_CON 0xE0200E00

#define EXT\_INT\_0\_PEND 0xE0200F40

#define EXT\_INT\_0\_MASK 0xE0200F00

#define rEXT\_INT\_0\_CON (\*(volatile unsigned int \*)EXT\_INT\_0\_CON)

#define rEXT\_INT\_0\_PEND (\*(volatile unsigned int \*)EXT\_INT\_0\_PEND)

#define rEXT\_INT\_0\_MASK (\*(volatile unsigned int \*)EXT\_INT\_0\_MASK)

//下面四行的宏定义在int.h中已经定义了，所以不需要再定义了。

/\* #define VIC0\_BASE (0xF2000000)

#define VIC0INTSELECT ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x0c)) )

#define VIC0INTENABLE ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x10)) )

#define VIC0INTENCLEAR ( \*((volatile unsigned long \*)(VIC0\_BASE + 0x14)) ) \*/

void system\_init\_exception(void);

void uart\_init(void); //直接移植上一章的usart.c的程序

void isr\_eint2()

{

printf("irq\_exception.\n");

rEXT\_INT\_0\_PEND |= (1<<2);//写1清挂起

}

int main()

{

unsigned long temp;

uart\_init();

//下面这个函数就是IRQ汇编服务程序绑定到异常向量表，关闭所有中断，所有中断都为IRQ中断类型

//将VICnADDR四个寄存器都清空。这样初始化完之后就很干净了嘛。

system\_init\_exception();

rEXT\_INT\_0\_PEND |= (1<<2);//写1先清挂起

rGPH0CON |= 0x0F<<8; // GPH0\_2设置为外部中断模式

rEXT\_INT\_0\_CON &= ~(0x0F<<8);//先清0

rEXT\_INT\_0\_CON |= 2<<8; // EXT\_INT2设置为下降沿触发

rEXT\_INT\_0\_MASK &= ~(1<<2); // 外部中断允许

//上面初始化已经把下面两行程序做的事情已经做了，所以不需要再做了

// VIC0INTENCLEAR = 0xffffffff; //先把中断使能全部断开

// VIC0INTSELECT = 0x0; //都选择为IRQ中断类型

//现在我们需要把每个中断源对应的真正服务程序放到对应编号的寄存器中

intc\_setvectaddr(NUM\_EINT2, isr\_eint2);//可以看到根据编号把函数名放到对应寄存器中

//下面三行的程序我们用intc\_enable(unsigned long intnum);就代替了，所以注释掉

intc\_enable(NUM\_EINT2);

// temp = VIC0INTENABLE;

// temp |= (1<<2); // 将中断使能，即开关闭合

// VIC0INTENABLE = temp;

//到这里我们根据物理编号就把对应中断通道设置好了

while(1)

{

printf("A ");

}

return 0;

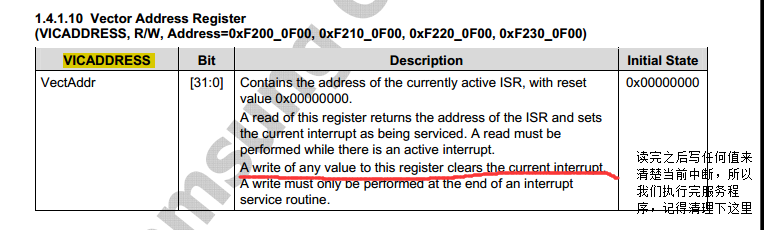
}

其他文件和上面保持一致就可以了。目录说明如下：



接下来我们就编译连接下载到板子。对了忘了说了，我们其实是下载到SD卡，破破坏iNand中的bootloader以从SD2启动。把卡插进SD2卡槽然后上电观察实验结果，发现就只能响应一次中断，第一次按响应了，后面再按就不响应了。怎么回事呢？

我们可以再仔细瞧瞧VICADDR寄存器，如下：



所以在main修改isr\_eint2函数如下：

void isr\_eint2()

{

printf("irq\_exception.\n");

rEXT\_INT\_0\_PEND |= (1<<2);//写1清挂起

VIC0ADDR = 0; //写任何值清楚，我们就写0来清楚

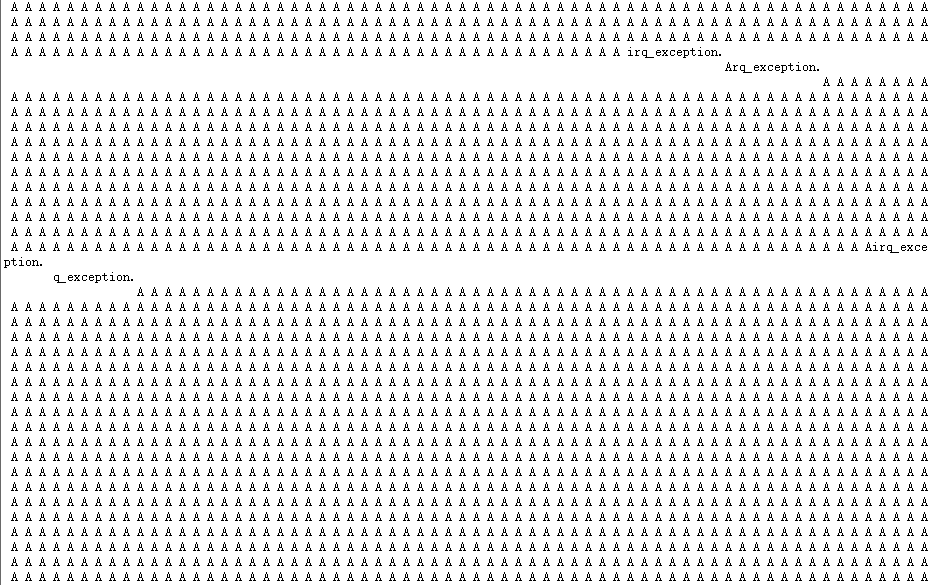
VIC1ADDR = 0;

VIC2ADDR = 0;

VIC3ADDR = 0;

}

这样修改后下载进去的结果就正常了（我们设计消抖），如下：



没按下一次按键，就会响应一下。

即将结束这章，那我顺便说说IRQ和FIQ究竟有何区别。210中支持2种中断，irq和fiq。irq是普通中断，fiq是快速中断。快速中断提供一种更快响应处理的中断通道，用于对实时性要求很高的中断源。fiq在CPU设计时预先提供了一些机制保证fiq可以被快速处理，从而保证实时性。fiq的限制就是只能有一个中断源被设置为fiq，其他都是irq。

CPU如何保证fiq比irq快？

有2个原因：第一，fiq模式有专用的r8～r12，因此在fiq的isr中可以直接使用r8-r12而不用保存，这就能节省时间；第二，异常向量表中fiq是最后一个异常向量入口。因此fiq的isr不需要跳转，可以直接写在原地，这样就比其他异常少跳转三次，因为IRQ都是先跑到公共的那个用汇编写服务程序处，然后跳转到功能为找到自己专用服务程序入口地址的函数里，进而跳转到自己真正的服务程序处，对应上面所说的就是先跳转到IRQ\_handle，然后跳转到irq\_handler 进而跳转到isr\_eint2。所以这样的机制为FIQ省了些时间。

最后在这里声明下：我这篇文章基本上都是以朱有鹏老师的记录和程序为原本而写的。在这里谢谢朱老师的为我们的努力和教育！