### 一. 阅读发布的Um\_s3c2410.pdf开发手册，回答如下问题

#### 对于I/O端口操作需求，假设GPB0端口连接了一个按键，需要读取按键的值。

1）至少需要操作的两个寄存器及其地址是：

- GPBDAT（General Purpose I/O Port B Data Register），用于读取或设置GPB端口的数据。该寄存器的地址通常是0x56000000 + 0x04。

- GPBCON（General Purpose I/O Port B Control Register），用于配置端口的功能（输入、输出等）。该寄存器的地址通常是0x56000000 + 0x00。

2）需要操作的寄存器位及操作方法：

- 在GPBCON寄存器中，需要配置与GPB0相对应的位，确保该位设置为输入模式。

- 对于GPBCON，需要进行GPBCON[0:1]=10操作，按键值=GPBDAT[10:0]

#### 解释下列寄存器的含义以及地址

1）\*\*ADCCON (Analog to Digital Converter Control Register)\*\*

- \*\*含义\*\*: 这个寄存器用于控制ADC（模数转换器）的工作模式，包括启动转换、选择通道、设置采样时间等。

- \*\*地址\*\*: 0x58000000

2）\*\*WTCON (Watchdog Timer Control Register)\*\*

- \*\*含义\*\*: WTCON寄存器控制看门狗定时器的功能，包括启用/禁用看门狗、设置工作模式（计数器或看门狗）、以及复位控制等。

- \*\*地址\*\*: 0x53000000

3）\*\*SPPIN0 (Serial Peripheral Interface Pin Control Register 0)\*\*

- \*\*含义\*\*: SPI接口是用于与外设通信的一种同步串行接口。SPPIN0寄存器控制SPI接口的一些引脚配置，如主/从模式选择、时钟极性与相位等。

- \*\*地址\*\*: 0x59000008,28

4）\*\*BWSCON (Bank Width & Wait Status Control Register)\*\*

- \*\*含义\*\*: 这个寄存器控制内存银行（Bank）的宽度（16位或32位访问）和等待状态，对于优化内存访问性能至关重要。它允许配置不同内存区域的访问方式，以适应不同速度的内存器件。

- \*\*地址\*\*: 0x48000000

5）\*\*BANKSIZE\*\*

- \*\*含义\*\*: 这通常指的是内存银行的大小配置，但它不是一个寄存器名称，而是描述系统内存组织的概念。在S3C2410上下文中，这可能是指通过寄存器（如BWSCON和其他相关寄存器）设置的每个内存银行的容量或地址范围。

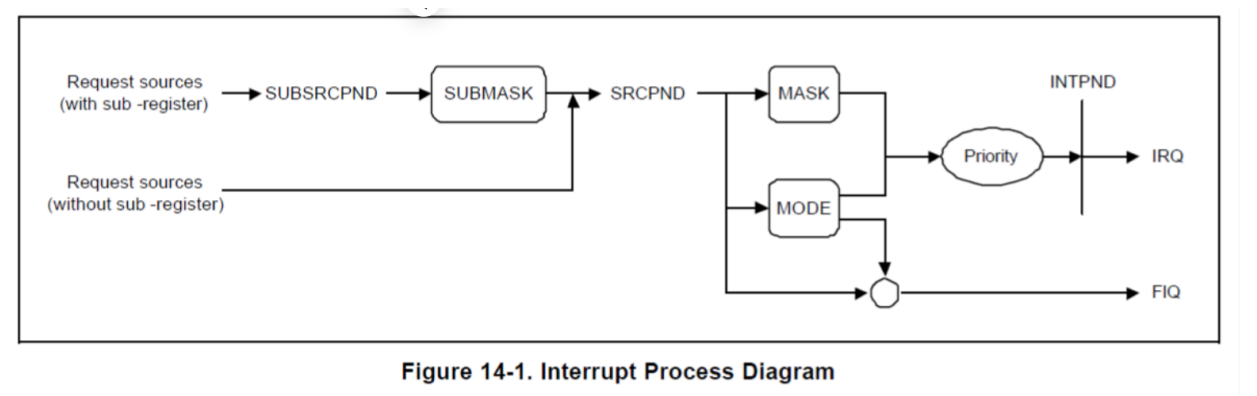
- \*\*地址\*\*:0x48000028

#### 结合STM32F746\_Experiment\_v1.1\01\_汇编\common\src\2410init.s和sys\_init.c\interrupts\_init()。进行程序分析。

1）分析sys\_init.c\debug\_swi()函数的执行效果。代码+注释的方式。

2） 分析2410init.s和2410init.s，并结合课上内容，以EINT3中断（设定为IRQ）为例，说清楚跳中断是如何传入ARM，并最终被执行的，可截取关键代码进行注释描述。

\*\*翻译下图，并说明中断信号的行进路线\*\*



图中是中断处理流程图，描述了中断请求的处理过程，中断信号的行进路线是从SUBSRCND和SRCPND端口进入，经过SUBMASK和MASK筛选，然后经过MODE分类，最后经过INTPND待处理，最终发出FIQ信号表示中断处理完成。

中断信号的行进路线如下：

1. 中断请求首先通过SUBSRCND（子源中断）和SRCPND（源中断）两个端口进入系统。

2. 然后，中断请求会经过SUBMASK（子屏蔽）和MASK（屏蔽）两个端口进行筛选，只有被允许的中断请求才会继续传递下去。

3. 接着，中断请求会经过MODE（模式）端口进行分类，根据不同的模式进行处理。

4. 最后，经过优先级排序后的中断请求会被发送到INTPND（中断待处理）端口，等待处理。

5. 如果中断请求被处理，那么系统会发出一个FIQ（快速中断）信号，表示中断已经完成处理。

##### \*\*分析sys\_init.c\debug\_swi()函数的执行效果。代码+注释的方式。\*\*

`debug\_swi()`函数是作为处理软件中断（SWI）异常的回调，当系统遇到SWI指令时被调用。它首先通过`uart\_printf`函数打印一条信息，指出SWI异常发生，并递增全局变量`nCNT`以跟踪异常发生的次数。随后调用`break\_point()`函数，用于提供调试时的视觉和听觉反馈。

```c++

void debug\_swi(void)

{

    // 记录进入SWI异常的信息，并使全局计数器'nCNT'递增，以追踪异常发生的次数。

    // 输出信息包含当前异常计数。

    uart\_printf("!!!进入 SWI 异常. %d\r\n", nCNT+=1);

    // 调用break\_point函数，进行调试指示操作，如LED闪烁和蜂鸣。

    break\_point();

}

void break\_point(void)

{

    int i;

    char m = 200; // 控制闪烁循环次数的计数器

    // 配置PORTF的GPIO设置：

    // - 将GPFCON寄存器的7至4位置为输出模式

    // - 禁用PORTF的上拉电阻，避免与外部电路冲突

    rGPFCON=0x5500;

    rGPFUP=0;

    // 闪烁循环：交替改变GPFDAT（PORTF的数据寄存器）的值，使得连接到PORTF的LED（假设7至4位代表LED）

    // 闪烁，同时在LED亮时发出蜂鸣声，以达到诊断目的的视觉和听觉提示。

    while(m) {

        rGPFDAT=0;                   // 关闭LED（假定低电平有效）

        beep(1);                   // 发出蜂鸣声

        for(i=0;i<20000;i++);      // 短暂延时以维持蜂鸣时长

        rGPFDAT=0xF0;               // 打开LED（假定高电平有效）

        beep(0);                   // 停止蜂鸣

        for(i=0;i<20000;i++);      // 延时以保持LED点亮状态，然后准备切换

        m--;                       // 循环计数减一

    }

}

```

#####  分析2410init.s和2410init.s，并结合课上内容，以EINT3中断（设定为IRQ）为例，说清楚跳中断是如何传入ARM，并最终被执行的，可截取关键代码进行注释描述。

1. \*\*中断初始化阶段\*\*

- \*\*中断控制器配置\*\*：需要在系统的中断控制器（比如Advanced Interrupt Controller, AIC）中配置EINT3对应的中断源，包括中断优先级、触发方式（上升沿、下降沿等）以及中断服务例程（ISR）的入口地址。

- \*\*全局中断使能\*\*：确保全局中断是开启的，通常通过写入特定寄存器（如INTMSK）来完成。

- \*\*IRQ模式设置\*\*：确保CPU的IRQ中断模式是使能的，这通常涉及到修改CPSR寄存器的相应位。

2. \*\*中断触发\*\*

当EINT3中断源发生时，中断控制器会向CPU发送中断请求信号。

3. \*\*中断响应\*\*

- \*\*CPU响应\*\*：CPU检测到中断请求后，如果当前允许中断并且不在更高级别的中断处理中（如FIQ），则保存当前状态（包括PC和其他寄存器）到堆栈，然后切换到IRQ模式。

- 模式切换：使用宏定义HANDLER来简化异常处理程序的设置，比如对于IRQ模式：

  ```

  1HandlerIRQ: HANDLER HandleIRQ

  ```

  这里，HANDLER宏负责设置IRQ模式下的堆栈，并跳转到HandleIRQ

  地址处的中断服务例程。

4. \*\*中断服务例程（ISR）执行\*\*

- \*\*中断向量表\*\*：IRQ模式下，CPU会自动跳转到中断向量表中IRQ模式对应的地址（通常是0x18）执行，该地址存放了一条跳转指令指向实际的IRQ处理程序。

- ISR入口：定义了 IsrIRQ作为IRQ中断处理的起点，其执行流程如下：

  ```

  1IsrIRQ:

  2sub  sp,sp,#4 @ 保留空间存放返回地址

  3stmfd  sp!,{r8-r9} @ 保存工作寄存器

  4ldr  r9,=INTOFFSET @ 获取中断偏移地址

  5ldr  r9,[r9] @ 读取偏移值

  6ldr  r8,=HandleEINT0 @ 获取EINT0的处理函数地址

  7add  r8,r8,r9,lsl #2 @ 根据偏移计算EINT3的处理函数地址

  8ldr  r8,[r8] @ 读取EINT3的ISR地址

  9str  r8,[sp,#8] @ 存储ISR地址

  10ldmfd sp!,{r8-r9,pc} @ 恢复寄存器并跳转到ISR

  ```

  上述代码段展示了如何动态确定具体的中断源（EINT3），并根据偏移量计算出正确的中断处理函数地址，最后执行该ISR。

5. \*\*中断返回\*\*

- 当ISR执行完毕，会执行`ldmfd sp!,{r8-r9,pc}`恢复之前保存的寄存器状态和PC值，从而返回到中断前的执行点

### 二. 以实验包中“STM32F746\_Experiment\_v1.1\02\_GPIO”例程为例子，描述STM32系列嵌入式系统的启动过程（如何调用到main函数的，之前经过了哪些初始化工作）。

STM32系列的启动过程涉及到复位、系统初始化、栈和堆的设置，最后跳转到`main`函数开始执行用户程序。

1. \*\*上电复位（Power-on Reset, POR）\*\*：当STM32微控制器上电时，会经历一个复位过程，此时系统会从地址`0x00000000`开始执行代码。

2. \*\*向量表（Vector Table）\*\*：在地址`0x00000000`处，系统会找到向量表。向量表是一个特殊的数据结构，它包含了一系列的函数指针，这些指针指向了中断处理程序和复位处理程序。

3. \*\*复位处理程序（Reset Handler）\*\*：向量表的第一个条目是复位处理程序的地址。在提供的代码中，复位处理程序是`Reset\_Handler`，它被设计为执行一些必要的初始化工作。

4. \*\*系统初始化（SystemInit）\*\*：在`Reset\_Handler`中，首先调用了`SystemInit`函数，这个函数执行了一些底层的硬件初始化，比如设置系统时钟、配置内存等。

5. \*\*调用C语言入口点（C Main Function）\*\*：`SystemInit`之后，代码通过`LDR R0, =\_\_main`获取C语言程序的入口点地址，然后使用`BX R0`跳转到`\_\_main`函数开始执行用户代码。

6. \*\*堆和栈的初始化\*\*：在启动代码中，还包含了对栈（Stack）和堆（Heap）的初始化。栈用于存储函数调用时的局部变量和返回地址，而堆用于动态内存分配。

7. \*\*中断处理程序（Interrupt Handlers）\*\*：向量表还包含了中断处理程序的入口地址，这些处理程序在相应的中断事件发生时被调用。

8. 最终 \_\_main 函数会调用用户的 main 函数，进入到用户程序