



Imagination大学计划

RVfpga实验9

中断驱动I/O

1. 简介

在本实验中，我们介绍中断的概念并展示如何在RVfpga上使用这些中断。中断可能由软件或硬件产生。在本实验中，我们重点介绍由物理引脚值变化而触发的硬件中断。具体来说，我们从第2部分开始，描述**编程I/O**和**中断驱动I/O**之间的区别。然后，我们说明RVfpga系统的中断控制器的操作，该控制器是SweRV EH1内核的一部分（第3部分）。在第4部分中，我们介绍如何使用Western Digital的外设支持包（Peripherals Support Package, PSP）和开发板支持包（Board Support Package, BSP）来配置外部中断，PSP和BSP是包含硬件外设驱动程序的软件。最后，我们列出了几个示例程序（第5部分），并提供了一些使用和扩展RVfpga系统硬件中断的练习（第6部分）。

2. 编程I/O与中断驱动I/O

可通过以下几种方法与外设交互：编程I/O、中断驱动I/O和直接存储器访问（Direct Memory Access, DMA）。在实验2-8中，我们使用**编程I/O**与外设交互。使用编程I/O时，用户程序会连续轮询I/O接口，并根据其状态相应地做出响应。例如，实验6中的**基本练习**使用编程I/O连续轮询（读取）开关0和1，以控制由四个点亮LED组成的块的速度和方向，这些点亮的LED重复地从一侧移到另一侧。编程I/O很容易实现，并且只需要很少的硬件支持，但是对I/O接口的连续轮询会使处理器忙于处理无用的工作。

中断驱动I/O可避免此问题，使程序仅在外设上发生事件时才做出响应。在此方法中，外设负责在某些事件发生时向处理器发送信号（称为**中断**），例如定时器溢出、在UART接口上接收到字符、按钮切换等事件。当没有事件发生（即没有中断）时，处理器继续执行有用的工作。当处理器接收到中断时，它将停止正在运行的程序，并调用**中断服务程序**（Interrupt Service Routine, ISR），也称作**中断处理程序**。ISR本质上是一个包含void参数的函数，可用于处理中断 - 即，读取按钮的新值，执行一些与定时器溢出相关的操作等。处理器通常支持单向量和多向量模式。在单向量模式下（图1），所有中断都调用相同的ISR。因此，当发生中断时，处理器将暂停主程序并跳转到通用ISR，后者首先确定中断源，然后执行与所确定的中断原因相对应的特定ISR代码。在多向量模式下（图2），每个中断都会调用不同的ISR。因此，当生成中断时，会首先确定中断原因，然后程序跳转到与所确定的原因相对应的ISR。

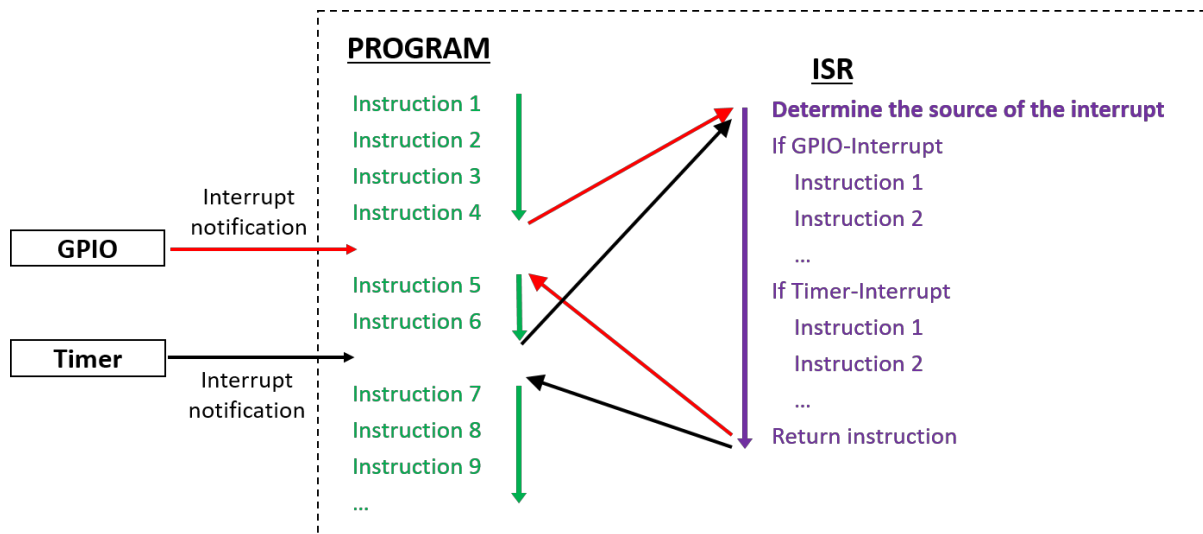


图1. 单向量模式下2个中断的示例

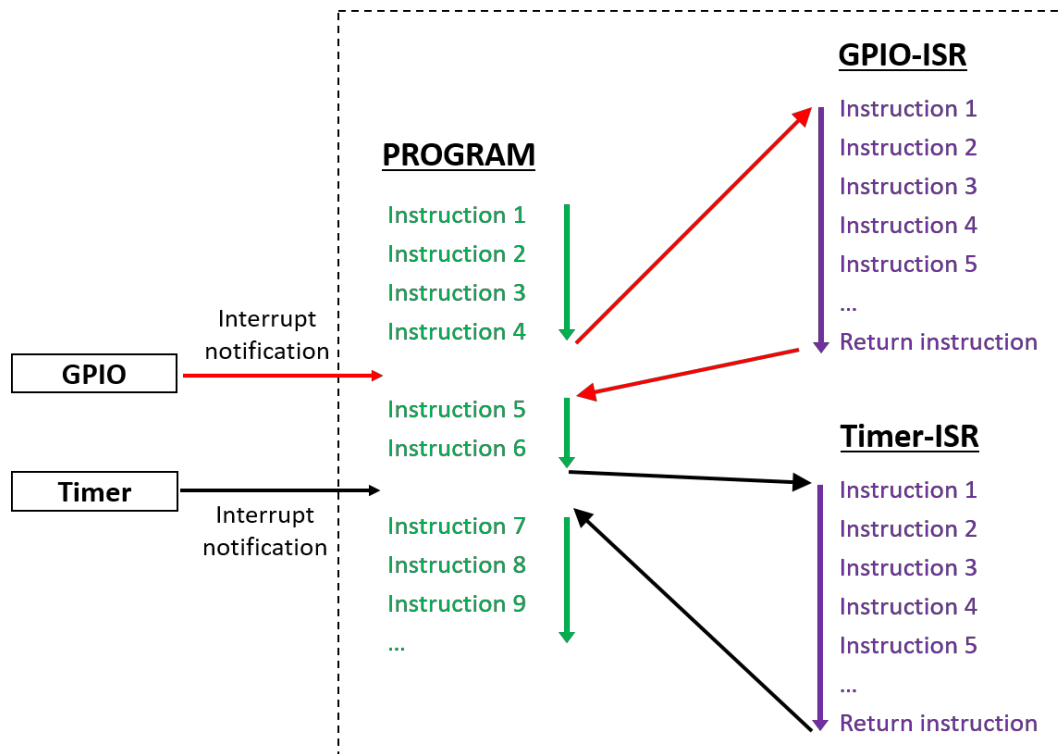


图2. 多向量模式下2个中断的示例

处理器通常允许对中断进行优先处理。不仅将优先处理较高优先级的中断，而且较高优先级的中断将抢占正在处理的较低优先级的中断。例如，假设将按钮中断优先级设置为5，将定时器中断优先级设置为7，并将阈值设置为4（因此两个优先级都高于阈值）。如果程序正在执行其正常流程，此时按下了按钮，则将发生中断，并且处理器调用ISR，ISR将从按钮读取数据

并加以处理。如果在按钮ISR激活时定时器溢出，则ISR本身将被中断，以便处理器可以立即处理定时器溢出。处理完成后，它将返回以完成按钮中断，然后返回主程序¹。

3. SWERV EH1提供的可编程中断控制器

SweRV EH1内核支持中断，如以下参考资料中所述，总结如下：

- **[PRM v1.7]** 1.7版本（2020年6月25日），《RISC-V SweRV EH1程序员参考手册》第6章，下载地址：https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf
- **[ISM v1.11]** 1.11版本初稿（2018年12月1日），《RISC-V指令集手册 - 第II卷：特权架构》第7章，下载地址：<https://github.com/riscv/riscv-isa-manual/releases/tag/draft-20181201-2650e2a>

SweRV EH1内核（参见[PRM v1.7]）的外部中断在很大程度上是按照RISC-V PLIC（平台级中断控制器）规范建模的。但是，中断控制器与内核而不是平台相关联。因此，更通用的术语PIC（可编程中断控制器）用于指代SweRV EH1内核中可用的控制器。PIC提供以下主要特性：

- 支持多达255个外部中断源（从1（最高优先级）到255（最低优先级））；每个中断源都有其自己的使能端。
- 除源编号外，还额外提供15个优先级；有两种优先方案可用：1-15（其中1是最低优先级）或0-14（其中14是最低优先级）。可以为每个源分配一个优先级。
- 提供对可编程优先级阈值的支持，以禁用较低优先级的中断。
- 支持向量外部中断、中断链和嵌套中断。

图3说明了RVfpga系统的中断系统的简化版本。所有产生中断的功能单元都称为**外部中断源**。外部中断源通过向PIC发送异步信号来指示中断请求，并且信号以_irq（中断请求的缩写）结尾。在本实验中，我们演示如何使用定时器和GPIO的中断；这些单元分别使用信号ptc_irq和gpio_irq产生中断。

每个外部中断源都连接到专用网关（位于PIC内部），该专用网关是一种硬件结构，负责将中断请求与内核的时钟域同步，并将请求信号转换为PIC的通用中断请求格式（即，高电平/低电平有效或电平触发）。PIC一次只能为每个中断源处理一个中断请求。它会评估所有待处理的和允许的中断请求，并选择具有最低源ID的优先级最高的中断。然后，将此优先级与可编程的优先级阈值进行比较，并且为了支持嵌套中断，还会与中断处理程序（如果当前正在运行）的优先级进行比较。如果所选请求的优先级高于这两个阈值，则PIC将向内核发送中断通知，内核会停止主程序的执行并跳转到相应的ISR，如图1（单向量模式）和图2（多向量模式）所示。

¹ D. Harris 和 S. Harris, 《数字设计和计算机体系结构》，第二版 - 2012, Morgan Kaufmann 出版社（美国加州旧金山），ISBN:978-0-12-394424-5。

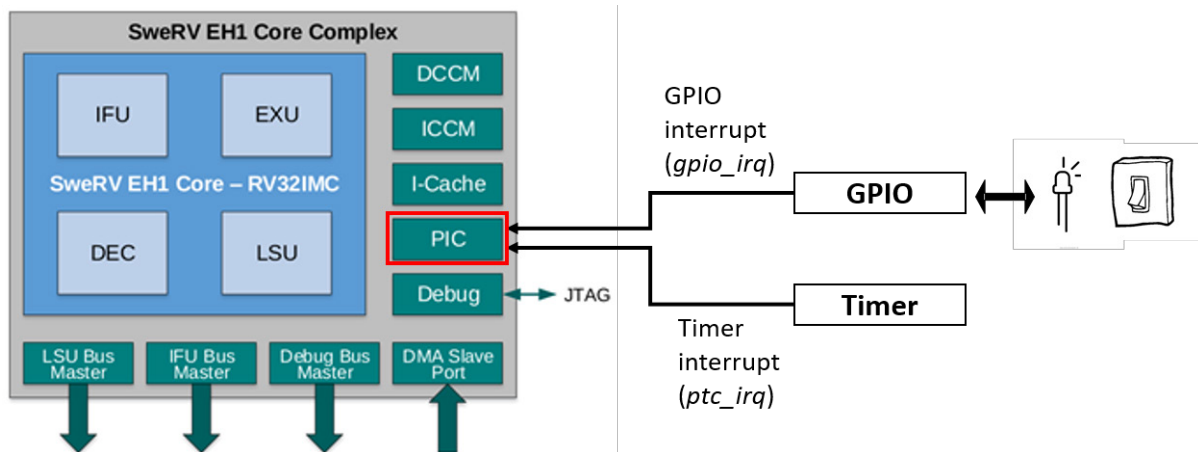


图3. RVfpga系统的中断系统

PIC的主要功能概括为以下几个基本步骤：

- 1) **允许/禁止**：PIC能够允许/禁止外部中断
- 2) **配置**：可以将PIC配置为监听具有不同极性（高电平有效/低电平有效）或类型（边沿触发/电平触发）的外部中断。PIC还允许将ISR分配给不同的存储器地址。
- 3) **过滤和优先级分配**：PIC允许为中断分配优先级。当主程序运行时，PIC选择已允许的优先级最高的触发中断。
- 4) **通知**：PIC选择了优先级最高的中断后，它将通知内核停止执行主程序，以便跳转到为所选中断服务的程序。
- 5) **抢占**：如果允许嵌套中断，则可以抢占由另一个具有更高优先级的中断服务的中断。

4. 在SweRV EH1中配置外部中断

与任何其他外设类似，PIC使用存储器映射寄存器进行配置，用户可通过装载/存储指令访问这些寄存器。可以在寄存器级别使用中断系统，但该过程非常复杂；幸运的是，WD的处理器支持包（Processor Support Package, PSP）和开发板支持包（BSP）

（<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>）包括多个函数，这些函数提供了一种更简单的方法来使用中断实现程序。表1介绍了配置外部中断所需的主要函数和宏。为了完整起见，本文档末尾的附录对可用的不同寄存器以及在寄存器级配置和使用PIC的步骤进行了说明。

表1. 用于配置外部中断的基本函数和宏

插座	说明
void pspInterruptsSetVectorTableAddress (void* pVectTable);	准备向量表地址
void pspExternalInterruptSetVectorTableAddress (void* pExtIntVectTable);	准备外部中断向量表地址
void bspInitializeGenerationRegister (u32_t uiExtInterruptPolarity)	将生成寄存器置于其初始状态
void bspClearExtInterrupt (u32_t uiExtInterruptNumber)	清除生成外部中断的触发器
void pspExtInterruptSetPriorityOrder (u32_t uiPriorityOrder);	设置优先级顺序（标准或预留）
void pspExtInterruptsSetThreshold (u32_t uiThreshold);	设置PIC中外部中断的优先级阈值
void pspExtInterruptsSetNestingPriorityThreshold (u32_t uiNestingPriorityThreshold);	设置PIC中外部中断的嵌套优先级阈值
void pspExtInterruptSetPolarity (u32_t uiIntNum, u32_t uiPolarity);	设置指定中断线路的极性（高电平有效或低电平有效）
void pspExtInterruptSetType (u32_t uiIntNum, u32_t uiIntType);	设置指定中断线路的类型（电平触发或边沿触发）
void pspExtInterruptClearPendingInt (u32_t uiIntNum);	清除指定中断线路的待处理中断的指示
void pspExtInterruptSetPriority (u32_t uiIntNum, u32_t uiPriority);	设置指定中断线路的优先级
void pspExternalInterruptEnableNumber (u32_t uiIntNum);	使能PIC中的指定中断线路
void pspInterruptsEnable (void);	无论其先前状态如何，都允许中断（在所有特权级别中）
void pspInterruptsDisable (u32_t *pOutPrevIntState);	在每个特权级别中禁止中断并返回当前中断状态

示例中断服务程序（ISR）将在本实验的后面部分给出。这些程序根据表1中的函数按照以下步骤配置RVfpga系统中断。注意，除了配置PIC外，还必须配置生成外部中断的外设（稍后将为示例和练习中使用的每个外设对此进行描述）。

中断系统的默认初始化:

1. 在多向量模式下，设置外部向量中断地址表的基址。使用函数 `pspInterruptsSetVectorTableAddress` 和 `pspExternalInterruptSetVectorTableAddress`。
2. 将生成寄存器置于其初始状态。使用函数 `bspInitializeGenerationRegister`。
3. 确保清除了外部中断触发器。使用函数 `bspClearExtInterrupt`。
4. 设置优先级顺序（函数 `pspExtInterruptSetPriorityOrder`）、阈值（函数 `pspExtInterruptsSetThreshold`）和嵌套优先级阈值（函数 `pspExtInterruptsSetNestingPriorityThreshold`）的默认值。

每个中断源的初始化:

1. 对于每个中断源，使用函数`pspExtInterruptSetPolarity`和`pspExtInterruptSetType`设置极性（高电平有效/低电平有效）和类型（电平触发/边沿触发）
2. 使用函数`pspExtInterruptClearPendingInt`清除所有待处理的中断。
3. 使用函数`pspExtInterruptSetPriority`设置每个外部中断源的优先级。
4. 使用函数`pspExternalInterruptEnableNumber`为适当的外部中断源允许中断。
5. 在多向量模式下，对于每个外部中断源，将相应处理程序的地址写入外部向量中断地址表中。

高级任务：为了更深入地了解这些基本函数，请查看位于`.platformio/packages/framework-wd-riscv-sdk/psp`下的PSP代码和位于`.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_ah1/bsp`下的BSP代码。需要特别注意的是下面列出的文件，其中一些文件包含在`api_inc`子文件夹中。

- `bsp_external_interrupts.h`: 在RVfpga中产生外部中断
- `psp_interrupts_ah1.h`: 为AH1内核上的ISR提供信息和注册API
- `psp_ext_interrupts_ah1.h`: 为SweRV AH1定义psp外部中断接口
- `psp_macros_ah1.h`: 为SweRV AH1定义psp宏
- `psp_csrs_ah1.h`: SweRV AH1 CSR的定义

此外，还建议对其中至少一个函数进行低至寄存器级别的分析。为此，可以使用附录中提供的信息，这些信息描述了SweRV AH1内核的PIC如何在寄存器级配置和管理外部中断。

高级任务：我们还建议分析并执行Western Digital提供的外部中断演示（网址为<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>），并且可以作为PlatformIO项目在以下位置获得：`[RVfpgaPath]/RVfpga/Labs/Lab9/WD_demo_external_int_Original`。如果一切正常，则应在串行控制台中看到以下消息：

```
Hello from SweRV core running on NexysA7
Core list:
    EH1 = 11
    EL2 = 16
Running demo on core 11...
-----
SweRVolf version 255.255255 (SHA 000000ef) (dirty 128)
-----
External Interrupts tests passed successfully
```

5. 示例

在本部分中，我们提供了将已编程I/O程序转换为中断驱动I/O程序的示例。我们给出了三个示例，说明了编程I/O固有的不同问题（前两个示例），然后指出了如何使用中断驱动I/O方案轻松解决这些问题（第三个示例）。

A. LED-Switch C-Lang程序

每次最右侧的开关上发生0→1跳变时，*LED-Switch_C-Lang*程序（参见图4）都会反转最右侧LED的状态。该程序位于：

[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_C-Lang.c

外设初始化之后，程序进入一个无限循环，该循环会将当前开关状态与先前开关状态进行比较，如果检测到0→1跳变，则会反转LED状态（请注意，当发生1→0跳变时，不会有任何变化）。

在之前的示例和练习中，我们用C语言编写了用于访问I/O寄存器（`READ_GPIO`、`READ_Reg`、`WRITE_GPIO`、`WRITE_Reg`等）的宏。在本示例中，我们改为使用PSP中定义的两个宏来实现相同的目的：`M_PSP_READ_REGISTER_32`（该宏读取作为参数提供的32位寄存器）和`M_PSP_WRITE_REGISTER_32`（该宏向32位寄存器写入第二个参数中提供的值）。请记住，为了能够使用这些宏，必须在文件*platformio.ini*中添加行`framework = wd-riscv-sdk`（以RVfpga为目标创建项目时，这是为默认操作），以及在程序的开头添加行`#include "psp_api.h"`（图4，第1行）。

```

1  #include "psp_api.h"
2
3  #define GPIO_SWs    0x80001400
4  #define GPIO_LEDs   0x80001404
5  #define GPIO_INOUT  0x80001408
6
7  int main ( void )
8  {
9      int LED_state, Sw_current_state, Sw_previous_state;
10
11      /* Configure LEDs and Switches */
12      M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
13
14      /* Init states */
15      LED_state = 0;
16      M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
17      Sw_previous_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
18
19      while (1) {
20          /* Invert LED-0 when SW-0 goes high */
21          Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
22          if(Sw_current_state==1 && Sw_previous_state==0){
23              LED_state = !LED_state;
24              M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
25          }
26          Sw_previous_state = Sw_current_state;
27      }
28
29      return(0);
30  }

```

图4. *LED-Switch_C-Lang*程序

任务：分析*LED-Switch_C-Lang*程序以了解其细节。如果需要，可以使用调试器来逐步分析该程序。

该程序可以正常工作，但是效率非常低，因为处理器只能读/写开关/LED。显然，我们希望处理器不仅仅能够与I/O设备通信，而且还要执行其他操作。

B. LED-Switch 7SegDispl C-Lang程序

在第二个示例中，*LED-Switch_7SegDispl_C-Lang*程序在*LED-Switch_C-Lang*的基础上扩展了第二个外设：7段显示屏。该程序执行两项任务：

- 与第一个示例相同，每次最右侧的开关上发生0→1跳变时，该程序都会反转最右侧的LED。
- 它在8位7段显示屏中显示升序计数，该计数大约每秒递增一次。请注意，为简便起见，我们使用for循环生成一秒延迟（在练习1中，将使用实验8的定时器来实现此目的）。

可以在图5中查看此程序，此程序位于：

[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_C-Lang.c

一些初始化之后，程序进入一个无限循环，该循环会将当前开关状态与先前开关状态进行比较，如果检测到0→1跳变，则会反转LED状态。然后，8位7段显示屏上显示的值递增，并产生延迟。请参见图5中的红框部分。

```

1  #include "psp_api.h"
2
3  #define SegEn_ADDR    0x80001038
4  #define SegDig_ADDR  0x8000103C
5
6  #define GPIO_SWs      0x80001400
7  #define GPIO_LEDs     0x80001404
8  #define GPIO_INOUT    0x80001408
9
10 int main ( void )
11 {
12     int i, LED_state, Sw_current_state, Sw_next_state, count=0;
13
14     /* Configure LEDs and Switches */
15     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF);
16
17     /* Configure 7-Seg Displays */
18     M_PSP_WRITE_REGISTER_32(0x80001038, 0x0);
19
20     /* Init states */
21     LED_state = 0;
22     M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
23     Sw_current_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
24
25     while (1) {
26         /* Invert LED-0 when SW-0 goes high */
27         Sw_next_state = (M_PSP_READ_REGISTER_32(GPIO_SWs) >> 16) & 0x1;
28         if(Sw_current_state==0 && Sw_next_state==1){
29             LED_state = !LED_state;
30             M_PSP_WRITE_REGISTER_32(GPIO_LEDs, LED_state);
31         }
32         Sw_current_state = Sw_next_state;
33
34         /* Increase 7-Seg Displays */
35         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
36         count++;
37
38         /* Delay */
39         for(i=0;i<1000000;i++);
40     }
41
42     return(0);
43 }

```

图5. *LED-Switch_7SegDispl_C-Lang*程序

任务：分析*LED-Switch_7SegDispl_C-Lang*程序以了解其详细信息。如果需要，可以使用调试器来逐步分析该程序。

请注意，在这种情况下，该程序在某些情况下甚至无法正常运行。例如，将永远不会检测到延迟循环内发生的0→1→0开关跳变。此外，我们仍然遇到与上一个示例相同的问题：处理器始终在忙于读/写设备或产生延迟。

我们如何改善这些情况？解决方案是使用**中断驱动I/O**。在下一部分提供的示例和练习中，我们将展示如何解决所有这些问题，以及如何实现效率更高且在所有情况下都能正常运行的程序。

C. LED-Switch 7SegDispl Interrupts C-Lang程序

这是最后一个示例，在本示例（`[RVfpgaPath]/RVfpga/Labs/Lab9/LED-Switch_7SegDispl_Interrupts_C-Lang.c`）中，我们展示如何使用中断驱动I/O来读取最右侧开关的状态。使用此策略可解决程序缺少延迟循环期间发生的开关跳变的问题。但请注意，使处理器忙于延迟循环的问题仍然存在。（将在练习1中解决此问题。）

新的**main**函数（如图7所示）执行以下任务：

- 初始化中断系统：
 - 中断的默认初始化：调用函数DefaultInitialization（第119行），如图8所示。
 - 通过调用函数pspExtInterruptsSetThreshold(5)（第120行）设置特定的阈值。优先级不超过此阈值的外部中断将被忽略。
- 初始化外部中断线路IRQ4：
 - 初始化线路IRQ4：调用函数ExternalIntLine_Initialization（第123行），用于中断线路4，优先级为6，GPIO_ISR作为中断服务程序。我们在图9中分析此函数。
 - 将IRQ4与GPIO中断线路（第124行）连接。具体方法是设置字0x80001018的位0（在本示例中标记为Select_INT）。该系统控制器存储器映射寄存器包含2位（参见图6）：位0，称为irq_gpio_enable，设置为1时用于将GPIO中断线路与IRQ4连接；位1，称为irq_ptc_enable，设置为1时用于将定时器中断线路与IRQ3连接。目前，了解这种高级功能已足够；稍后，在练习2中，我们将详细说明Verilog实现，以便可以在练习中对其进行修改。

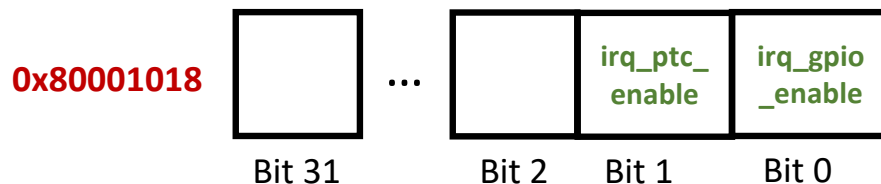


图6. RVfpga系统的寄存器0x80001018

- 初始化外设（在本示例中，为GPIO和7段显示屏）：
 - 调用函数GPIO_Initialization（第127行）。我们在图10中分析此函数。
 - 使能八个7段显示屏（第128行）。
- 允许中断：
 - 调用函数pspInterruptsEnable（第131行）和宏M_PSP_SET_CSR（第132行）。常量D_PSP_MIE_NUM和D_PSP_MIE_MEIE_MASK由WD的PSP定义。
- 最后，写入7段显示屏，并在永久重复的循环内建立延迟（第134-141行）。

```

114 int main(void)
115 {
116     int count=0, i;
117
118     /* INITIALIZE THE INTERRUPT SYSTEM */
119     DefaultInitialization(); /* Default initialization */
120     pspExtInterruptsSetThreshold(5); /* Set interrupts threshold to 5 */
121
122     /* INITIALIZE INTERRUPT LINE IRQ4 */
123     ExternalIntLine_Initialization(4, 6, GPIO_ISR); /* Initialize line IRQ4 with a priority of 6. Set GPIO_ISR as the Interrupt Service Routine */
124     M_PSP_WRITE_REGISTER_32(Select_INT, 0x1); /* Connect the GPIO interrupt to the IRQ4 interrupt line */
125
126     /* INITIALIZE THE PERIPHERALS */
127     GPIO_Initialization(); /* Initialize the GPIO */
128     M_PSP_WRITE_REGISTER_32(SegEn_ADDR, 0x0); /* Initialize the 7-Seg Displays */
129
130     /* ENABLE INTERRUPTS */
131     pspInterruptsEnable(); /* Enable all interrupts in mstatus CSR */
132     M_PSP_SET_CSR(D_PSP_MIE_NUM, D_PSP_MIE_MEIE_MASK); /* Enable external interrupts in mie CSR */
133
134     while (1) {
135         /* Increase 7-Seg Displays */
136         M_PSP_WRITE_REGISTER_32(SegDig_ADDR, count);
137         count++;
138
139         /* Delay */
140         for(i=0; i<50000000; i++);
141     }
142 }
143

```

图7. *main*函数

DefaultInitialization函数（如图8所示）执行第4部分的项目“中断系统的默认初始化”中所述的步骤：

- 配置向量表（第53和56行）。请注意，在本示例中，数组 `G_Ext_Interrupt_Handlers` 存储向量表。
- 初始化用于触发IRQ的寄存器（第59行）。
- 清除第61-65行的所有外部中断（在本示例中为IRQ3和IRQ4）。常量 `D_BSP_FIRST_IRQ_NUM` 和 `D_BSP_LAST_IRQ_NUM` 分别由WD的BSP定义为3和4。
- 建立默认阈值和优先级（第68、71和74行）。同样，这些函数使用的常量由WD的PSP定义。

```

48 void DefaultInitialization(void)
49 {
50     u32_t uiSourceId;
51
52     /* Register interrupt vector */
53     pspInterruptsSetVectorTableAddress(&M_PSP_VECT_TABLE);
54
55     /* Set external-interrupts vector-table address in MEIVT CSR */
56     pspExternalInterruptSetVectorTableAddress(G_Ext_Interrupt_Handlers);
57
58     /* Put the Generation-Register in its initial state (no external interrupts are generated) */
59     bspInitializeGenerationRegister(D_PSP_EXT_INT_ACTIVE_HIGH);
60
61     for (uiSourceId = D_BSP_FIRST_IRQ_NUM; uiSourceId <= D_BSP_LAST_IRQ_NUM; uiSourceId++)
62     {
63         /* Make sure the external-interrupt triggers are cleared */
64         bspClearExtInterrupt(uiSourceId);
65     }
66
67     /* Set Standard priority order */
68     pspExtInterruptSetPriorityOrder(D_PSP_EXT_INT_STANDARD_PRIORITY);
69
70     /* Set interrupts threshold to minimal (== all interrupts should be served) */
71     pspExtInterruptsSetThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
72
73     /* Set the nesting priority threshold to minimal (== all interrupts should be served) */
74     pspExtInterruptsSetNestingPriorityThreshold(M_PSP_EXT_INT_THRESHOLD_UNMASK_ALL_VALUE);
75 }

```

图8. *DefaultInitialization*函数

ExternalIntLine_Initialization函数（如图9所示）执行第4部分的项目“每个中断源的初始化”中所述的步骤：

- 配置IRQ4中断的类型和极性（这些函数使用的常量由WD的PSP定义），并在相应的网关处清除任何潜在的待处理中断（第81、84和87行）。

- 设置IRQ4的优先级（第90行）。
- 在PIC中允许IRQ4中断（第93行）。
- 将GPIO中断服务程序（GPIO_ISR）记录在向量表中（第96行），该向量表存储在数组G_Ext_Interrupt_Handlers中。

```

78 void ExternalIntLine_Initialization(u32_t uiSourceId, u32_t priority, pspInterruptHandler_t pTestIsr)
79 {
80     /* Set Gateway Interrupt type (Level) */
81     pspExtInterruptSetType(uiSourceId, D_PSP_EXT_INT_LEVEL_TRIG_TYPE);
82
83     /* Set gateway Polarity (Active high) */
84     pspExtInterruptSetPolarity(uiSourceId, D_PSP_EXT_INT_ACTIVE_HIGH);
85
86     /* Clear the gateway */
87     pspExtInterruptClearPendingInt(uiSourceId);
88
89     /* Set IRQ4 priority */
90     pspExtInterruptSetPriority(uiSourceId, priority);
91
92     /* Enable IRQ4 interrupts in the PIC */
93     pspExternalInterruptEnableNumber(uiSourceId);
94
95     /* Register ISR */
96     G_Ext_Interrupt_Handlers[uiSourceId] = pTestIsr;
97 }

```

图9. *ExternalIntLine_Initialization*函数

GPIO_Initialization函数（如图10所示）执行以下任务：

- 将GPIO引脚配置为输入/输出，并将LED初始化为0（第103和104行）。
- 配置GPIO中断。（要进一步了解每个GPIO寄存器的功能，请使用GPIO内核规范，可从以下位置获得该规范：

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/Peripherals/gpio/docs/gpio_spec.pdf。）

- o **RGPIO_INTE**：确定哪些通用引脚生成中断（第107行）。
- o **RGPIO_PTRIG**：确定生成中断的边沿（第108行）。
- o **RGPIO_INTS**：清除所有引脚的中断（第109行）。
- o **RGPIO_CTRL**：该寄存器的最低有效位允许生成中断（第110行）。

```

100 void GPIO_Initialization(void)
101 {
102     /* Configure LEDs and Switches */
103     M_PSP_WRITE_REGISTER_32(GPIO_INOUT, 0xFFFF); /* GPIO_INOUT */
104     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, 0x0); /* GPIO_LEDS */
105
106     /* Configure GPIO interrupts */
107     M_PSP_WRITE_REGISTER_32(RGPIO_INTE, 0x10000); /* RGPIO_INTE */
108     M_PSP_WRITE_REGISTER_32(RGPIO_PTRIG, 0x10000); /* RGPIO_PTRIG */
109     M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0); /* RGPIO_INTS */
110     M_PSP_WRITE_REGISTER_32(RGPIO_CTRL, 0x1); /* RGPIO_CTRL */
111 }

```

图10. *GPIO_Initialization*函数

最后，在GPIO上触发中断时调用ISR（即图11所示的**GPIO_ISR**函数）。此ISR（中断服务程序）执行以下任务：

- 读取LED的当前状态（第35行）。

- 反转和屏蔽LED（第36-37行）。
- 向LED写入新值（第38行）。
- 清除GPIO中断（第41行）。
- 清除IRQ4外部中断（第44行）。

```

30 void GPIO_ISR(void)
31 {
32     unsigned int i;
33
34     /* Write the LED */
35     i = M_PSP_READ_REGISTER_32(GPIO_LEDS);          /* RGPIO_OUT */
36     i = !i;                                           /* Invert the LEDs */
37     i = i & 0x1;                                     /* Keep only the right-most LED */
38     M_PSP_WRITE_REGISTER_32(GPIO_LEDS, i);          /* RGPIO_OUT */
39
40     /* Clear GPIO interrupt */
41     M_PSP_WRITE_REGISTER_32(RGPIO_INTS, 0x0);        /* RGPIO_INTS */
42
43     /* Stop the generation of the specific external interrupt */
44     bspClearExtInterrupt(4);
45 }

```

图11. GPIO_ISR函数

任务：分析LED-Switch_7SegDispl_Interrupts_C-Lang程序以了解其详细信息。可以将实现与第4部分的说明进行比较，并根据需要使用调试器来逐步分析程序。

6. 练习

练习1. 修改LED-Switch_7SegDispl_Interrupts_C-Lang程序以包括第二个中断源，在本例中是由定时器的中断源。请记住，定时器可以用作PWM发生器、定时器或计数器，因此通常称为PTC单元。

- 在RVfpga系统中，通过设置字0x80001018的位1（`irq_ptc_enable`）可以将定时器中断连接到IRQ3（参见图6）。
- 创建一个初始化PTC中断的函数，类似于前面示例中的GPIO_Initialization。
- 创建第二个ISR，称为PTC_ISR。它应类似于LED-Switch_7SegDispl_Interrupts_C-Lang程序中的GPIO_ISR，但应使用IRQ3来调用。PTC_ISR应处理并清除定时器中断。

实现并调试程序后，使用PSP函数`pspExtInterruptsSetThreshold(threshold)`和`pspExtInterruptSetPriority(interrupt_source, priority)`来分析优先级和阈值的不同组合。请注意，甚至可以在执行时更改优先级；例如，可以在7段显示屏上显示计数到10，然后通过修改相应外部中断源的优先级停止计数。

练习2. 修改RVfpgaNexys以包含第三个中断源，该中断源来自实验6中设计用于控制板上按钮的第二个GPIO（GPIO2）。可通过两种方法完成此练习：

- 可以将GPIO2中断连接到未使用的外部中断源。SweRV EH1最多提供255条不同的中断线路，目前为止，我们仅使用了其中的2条。这种方法的缺点在于需要修改WD的库。

- 可以将GPIO2中断连接到IRQ4，以便GPIO模块（连接到LED和开关）和GPIO2（连接到按钮）使用单向量中断模式。尽管在某些情况下多向量模式更为可取，但这种方法的优点是可以重复使用BSP。

通过提供有关RVfpga系统中的中断低级实现的一些详细信息，我们为第二种方法提供了一些指导。

图12给出了电路，该电路将各种中断源（GPIO中断、定时器中断和SweRVolf内核中最初可用的中断源，我们在此不分析和使用这些中断源）与IRQ4和IRQ3连接起来。具体来说，当`irq_gpio_enable = 1`（图6）时，IRQ4与GPIO连接；而`irq_ptc_enable = 1`（图6）时，IRQ3与定时器连接。当`irq_gpio_enable = irq_ptc_enable = 0`时，IRQ4和IRQ3与SweRVolf原始中断源连接，在本实验中，我们不使用这些中断源（如果您有兴趣使用这些中断源，请访问<https://github.com/chipsalliance/Cores-SweRVolf>来查看更多信息）。

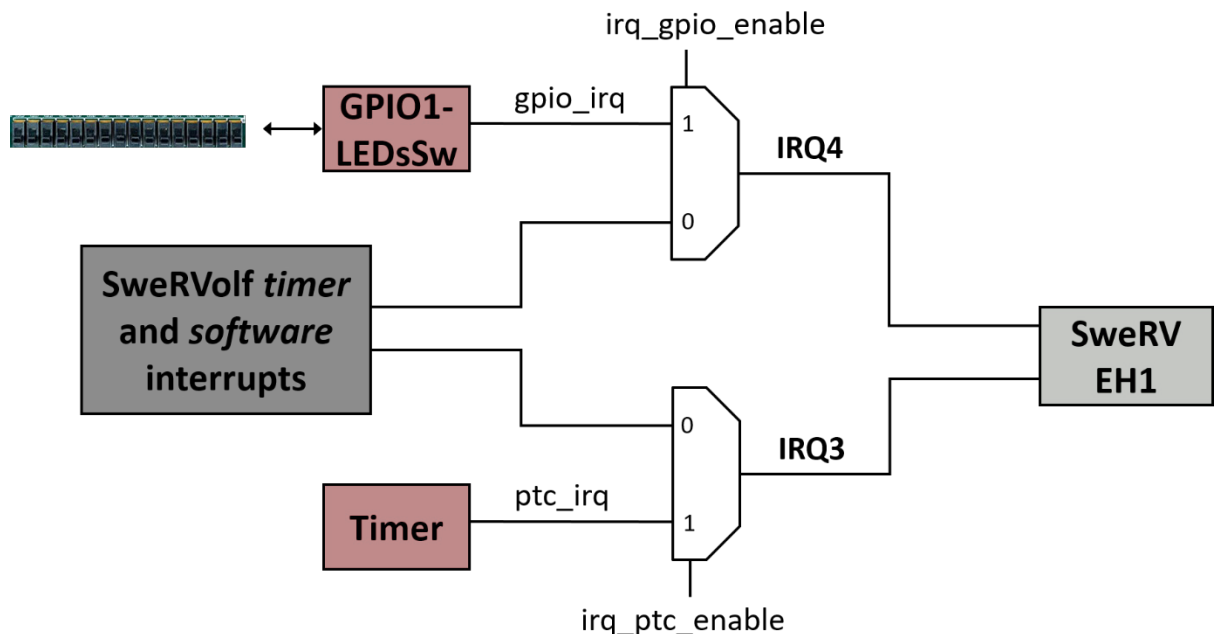


图12. 逻辑实现：GPIO和定时器中断分别与IRQ4和IRQ3的连接

图13给出了模块`swervolf_core`的Verilog区域，该模块实现中断源与IRQ4和IRQ3之间的连接。当信号`irq_gpio_enable`为1时，GPIO中断与IRQ4连接（红框上部）。当信号`irq_ptc_enable`为1时，定时器中断与IRQ3连接（红框下部）。当两个信号均为0（图中未突出显示的代码）时，在SweRVolfX中实现的中断源将连接到IRQ3和IRQ4。


```

123 always @(posedge i_clk) begin
124     o_wb_ack <= i_wb_cyc & !o_wb_ack;
125
126     nmi_int    <= 1'b0;
127     nmi_int_r  <= nmi_int;
128
129     // GPIO Interrupt through IRQ4. Enable by setting bit 0 of word 0x80001018
130     if (irq_gpio_enable & gpio_irq) begin
131         sw_irq4 <= 1'b1;
132     end
133
134     // Timer (PTC) Interrupt through IRQ3. Enable by setting bit 1 of word 0x80001018
135     if (irq_ptc_enable & ptc_irq) begin
136         sw_irq3 <= 1'b1;
137     end
138
139     // SweRVolf simple timer and software interrupts. Enable by resetting bits 0 and 1 of word 0x80001018
140     if (!irq_gpio_enable & !irq_ptc_enable) begin
141
142         if (sw_irq3_edge)
143             sw_irq3 <= 1'b0;
144         if (sw_irq4_edge)
145             sw_irq4 <= 1'b0;
146
147         if (irq_timer_en)
148             irq_timer_cnt <= irq_timer_cnt - 1;
149
150         if (irq_timer_cnt == 32'd1) begin
151             irq_timer_en <= 1'b0;
152             if (sw_irq3_timer)
153                 sw_irq3 <= 1'b1;
154             if (sw_irq4_timer)
155                 sw_irq4 <= 1'b1;
156             if (!(sw_irq3_timer | sw_irq4_timer))
157                 nmi_int <= 1'b1;
158         end
159     end
160 end

```

图13. Verilog实现：GPIO和定时器中断分别与IRQ4和IRQ3的连接，如红色突出显示部分。

在本练习中，必须扩展先前的实现（图12）以包括一个新的中断源，该中断源连接到IRQ4，如图14所示。

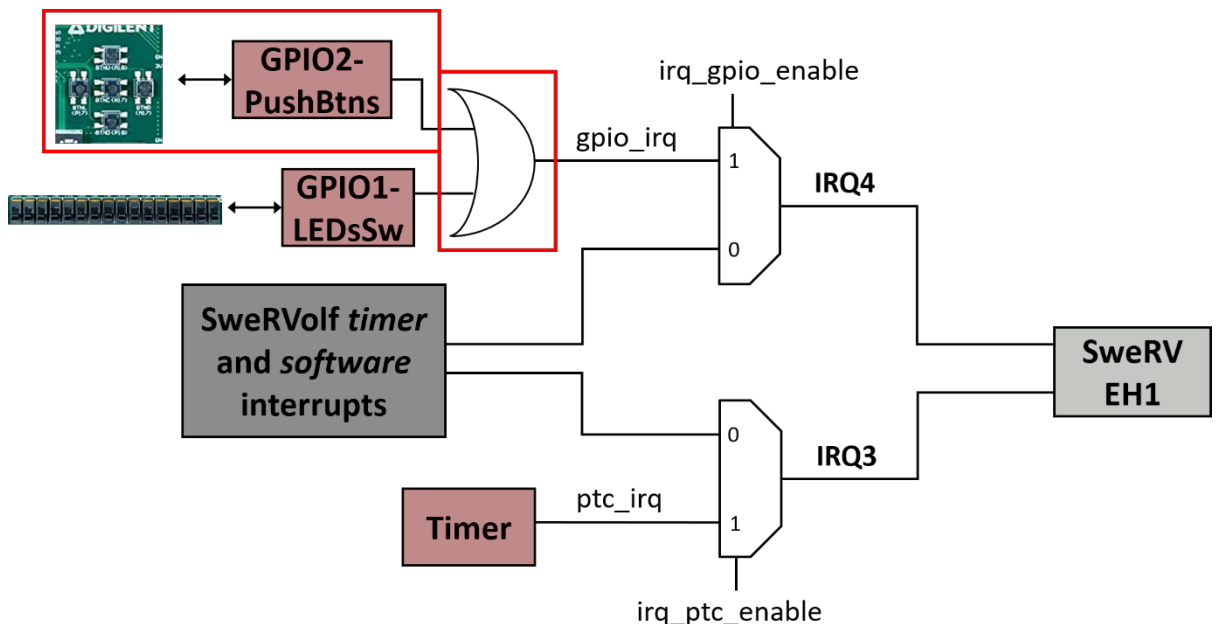


图14. 逻辑实现：第二个中断源（由读取按钮的GPIO提供）与IRQ4连接

我们突出显示了其他一些也需要了解的Verilog区域，但在本示例中无需对其进行修改。

- 在`swervolf_core`模块的第600行（图15）将中断源插入SweRV处理器。尽管有四个中断源可用，但在本实验中，我们仅关注中断源`sw_irq4`和`sw_irq3`。

```
600      .extintsrc_req ({4'd0, sw_irq4, sw_irq3, spi0_irq, uart_irq}),
```

图15. 中断源发送到SweRV

- 在`swervolf_syscon`模块的第192-196行（图16），通过内核写入使能信号`irq_gpio_enable`和`irq_ptc_enable`（可通过地址`0x80001018`访问，请参见图6）。

```
192      6: begin //0x18-0x1B
193          if (i_wb_sel[0])
194              irq_gpio_enable <= i_wb_dat[0];
195              irq_ptc_enable <= i_wb_dat[1];
196      end
```

图16. 从SweRV内核写入寄存器`0x80001018`

在`swervolf_syscon`模块的第248-249行，从内核读取使能信号`irq_gpio_enable`和`irq_ptc_enable`（参见图17）。

```
248      //0x18-0x1B
249      6 : o_wb_rdt <= {30'd0, irq_ptc_enable, irq_gpio_enable};
```

图17. 读取寄存器`0x80001018`到SweRV内核

练习3. 使用在前面练习中设计的扩展RVfpgaNexys版本来实现C程序，该程序在LED上从1开始显示逐渐递增的二进制计数。将中断与定时器结合使用来生成延迟，以在各个递增值的显示之间留出等待时间，这样人眼便可看到这些值。读取BTNC并使用它来更改计数速度，读取Switch[0]并使用它重新启动计数（按下时）。

对于练习2中的扩展RVfpgaNexys，现在有三个可能的中断源：

- GPIO**（来自开关的中断）
- GPIO2**（来自在练习2中设计的按钮的中断）
- PTC**（定时器）

假定练习2中的扩展RVfpgaNexys实现有两个中断源共享同一行（*IRQ4*），则相应的中断服务程序（GPIO_ISR）必须识别生成中断的设备。可以从GPIO寄存器中获取相关信息。

附录

本附录介绍SweRV EH1内核的可编程中断控制器（Programmable Interrupt Controller, PIC）如何在寄存器级管理外部中断。PIC使用表2所示的存储器映射寄存器。必须注意的是，PIC存储器空间的起始地址为0xF00C0000；此地址称为RV_PIC_BASE。地址是相对于该基址给出的。

表2. PIC存储器映射寄存器地址映射

名称	地址（相对于RV_PIC_BASE）	说明	在手册中的位置
meipIS	0x0004 - 0x0004+S _{max} *4-1	外部中断优先级寄存器	[PRM v1.7]的表6-2
meipX	0x1000 - 0x1000+(X _{max} +1)*4-1	外部中断待处理寄存器	[PRM v1.7]的表6-3
meieS	0x2000 - 0x2000+S _{max} *4-1	外部中断使能寄存器	[PRM v1.7]的表6-4
mpiccfg	0x3000 - 0x3003	外部中断PIC配置寄存器	[PRM v1.7]的表6-1
meigwctrlS	0x4004 - 0x4004+S _{max} *4-1	外部中断网关配置寄存器（仅适用于可配置的网关）	[PRM v1.7]的表6-11
meigwclrS	0x5004 - 0x5004+S _{max} *4-1	外部中断网关清除寄存器（仅适用于可配置的网关）	[PRM v1.7]的表6-12

所有寄存器均为32位宽，并且可通过装载和存储指令访问，与存储器映射I/O一样。访问类型取决于我们要访问的特定位（可以在[PRM v1.7]中进行查看）。

一些寄存器具有以S或X结尾的参数化名称。这些寄存器可能存在多个实例。参数S是指外部中断源的数量，在SweRV EH1中等于网关的数量。因此，以“S”结尾的寄存器具有1到255个可用的寄存器实例。在本实验中，我们仅使用2个外部中断源：IRQ3（与定时器相关）和IRQ4（与GPIO相关）。参数X是指一组32个网关。这并不意味着对网关进行了分组，但是对其进行分组可以减少某些32位寄存器所需的存储器大小，其中1个位便足以对一组外部中断源执行操作。外部中断待处理寄存器就是这种情况，其中1个位便足以区分是否已处理该中断。为了获得有关这些寄存器的更多信息，在表1的最右列指出了[PRM v1.7]中包含位级别（特定中断）说明的位置。

除了表2中所示的寄存器外，PIC还包含控制和状态寄存器（CSR）。标准RISC-V ISA建立了12位编码空间（csr[11:0]），最多支持4,096个CSR。按照惯例，CSR地址的高4位（csr[11:8]）用于根据特权级别对CSR的读写可访问性进行编码。前两位（csr[11:10]）指示寄存器是读/写（00、01或10）还是只读（11）。后两位（csr[9:8]）对可以访问CSR的最低特权级别进行编码。有关CSR的更多信息，请参见[PRM v1.7]和[ISM v1.11]。表3列出了有助于管理SweRV EH1内核中的外部中断的CSR。可通过csrrw或csrrs（CSR读/写和CSR读/设置）等专用装载和存储指令访问这些CSR。

表3. PIC非标准RISC-V CSR地址映射

名称	编号	说明	位置
meivt	0xBC8	外部中断向量表寄存器	[PRM v1.7]的表6-6
meipt	0xBC9	外部中断优先级阈值寄存器	[PRM v1.7]的表6-5
meicpct	0xBCA	外部中断声明ID/优先级捕捉触发寄存器	[PRM v1.7]的表6-8
meicidpl	0xBCB	外部中断声明ID的优先级寄存器	[PRM v1.7]的表6-9
meicurpl	0xBCC	外部中断当前优先级寄存器	[PRM v1.7]的表6-10
meihap	0xFC8	外部中断处理程序地址指针寄存器	[PRM v1.7]的表6-7
mie	0x304	机器中断使能寄存器	[PRM v1.7]的表11-1
mstatus	0x300	机器状态寄存器	[ISM v1.11]的图3.7

表3的最右列指出了[PRM v1.7]或[ISM v1.11]中说明给定CSR的位级别信息的位置（注意，[PRM v1.7]中未提供*mstatus*位说明，而[ISM v1.11]中提供了此说明）。

A. 外部中断配置

在本小节中，我们总结了使用上述寄存器配置外部中断所需的基本步骤：

1. 通过将*mie* CSR中的位*miep*清零，禁止所有外部中断。
2. 通过写入*mpiccfg*寄存器的*priord*位来配置优先级顺序。
3. 在多向量模式下，如果未配置，则通过写入*meivt*寄存器的基址字段来设置外部向量中断地址表的基址。
4. 通过写入*meipt*寄存器的*prithresh*字段来设置优先级阈值。
5. 通过向*meicidpl*寄存器的*clidpri*字段和*meicurpl*寄存器的*currpri*字段写入“0”（或对于反转的优先级顺序写入“15”）来初始化嵌套优先级阈值。
6. 对于每个可配置的网关*S*，在*meigwctrlS*寄存器中设置极性（高电平有效/低电平有效）和类型（电平触发/边沿触发），并通过写入网关的*meigwclrS*寄存器将IP位清零。
7. 在多向量模式下，对于每个外部中断源*S*，将相应处理程序的地址写入外部向量中断地址表中。
8. 通过写入*meiplS*寄存器的相应优先级字段，设置每个外部中断源*S*的优先级。
9. 通过设置每个中断源*S*的*meieS*寄存器的*inten*位，为适当的外部中断源允许中断。
10. 激活*mstatus* CSR中的*mei*位。
11. 通过设置*mie* CSR中的位*miep*，允许所有外部中断。

以上是用于*S*网关的常规步骤。但是，在RVfpga系统中，我们仅使用2个中断源（IRQ3和IRQ4），每个中断源都有其自己的网关。而且，需要注意的是，不必非要遵循上述顺序，因为一些操作是可以互换的（例如，可以在步骤2之前完成步骤4）。此外，由于每个函数在输入时都会调用*psplInterruptsDisable*，因此不一定要执行步骤1。

B. 外部中断工作模式

在本小节中，我们介绍触发外部中断后PIC将如何工作。一旦在外部中断线路（导线）上发生所需事件，就会执行以下操作：

1. PIC决定哪个待处理中断具有最高优先级。
2. 当目标hart（硬件线程）采用外部中断时，它会禁止所有中断（即将RISC-V hart的 *mstatus* 寄存器中的 *mie* 位清零），并跳转到外部中断处理程序。
3. 外部中断处理程序写入 *meicpct* 寄存器，以触发对待处理的最高优先级外部中断的中断源ID（在 *meihap* 寄存器中）及其相应优先级（在 *meicidpl* 寄存器中）的捕捉操作。
4. 然后，处理程序读取 *meihap* 寄存器以获取在 *claimid* 字段中提供的中断源ID。根据 *meihap* 寄存器的内容，外部中断处理程序跳转到特定于该外部中断源的处理程序。在图18中可以观察到此过程。
5. 对于特殊中断源的中断处理程序（ISR）：
 - a. 对于电平触发的中断源，中断处理程序会清除SoC IP中发起中断请求的状态。
 - b. 对于边沿触发的中断源，中断处理程序通过写入 *meigwclrS* 寄存器将源网关中的IP位清零。
 这会将源的中断请求置为无效。
6. 同时，PIC在后台继续评估待处理的中断。

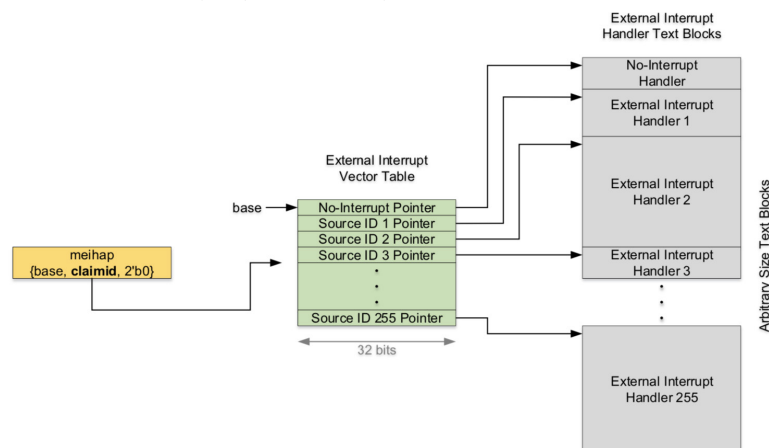


图18. 向量化外部中断（取自[PRM v1.7]）

必须注意的是，这是一种常规的工作模式。此外，SweRV EH1内核还支持嵌套中断（最多15个）。有关更多信息，请查阅[PRM v1.7]。