一、实验名称

时钟Tick

二、实验目的

（1）深刻理解中断的原理和机制；
（2）掌握CPU访问中断控制器的方法；
（3）掌握Arm体系结构的中断机制和规范；
（4）实现时钟中断服务和部分异常处理等。

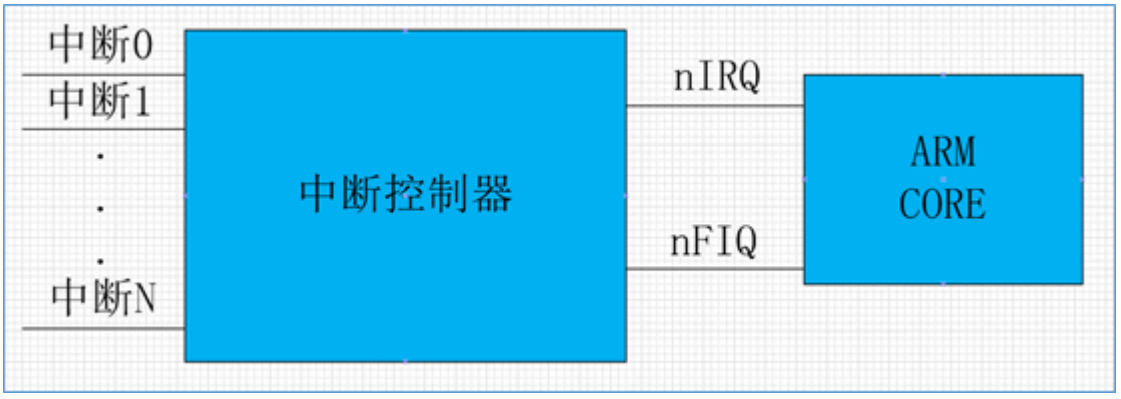
三、写在前面

前面实验四实现了异常处理。其中包括Sync（Synchronous exceptions，同步异常）、IRQ（Interrupt requests，中断请求）、FIQ （Fast Interrupt Requests，快速中断请求）、SError（System Error，系统错误）。实验五对IRQ和FIQ进行了考虑，重点实现了处理IRQ的例程。为了更好地理解这份实验报告，先让我们总体地概述一下它实现思路。
GIC是Arm采用的中断控制器。我们的实验采用CICv2版本。GIC通过**分发器（Distributor）**管理所有中断源，通过**CPU接口（CPU Interface）**完成中断向CPU的传递。而控制Distributor的寄存器是GICD*开头的，控制CPU Interface是GICC*开头的。我们首先初始化GIC，更改异常向量表，重写中断处理函数。对定时器进行配置，实现时钟中断函数。最后呢，我们可以看到硬件发出的周期性时钟中断的结果。

四、实验任务

1. Arm的中断系统

中断是一种硬件机制。借助于中断，CPU可以不必再采用轮询这种低效的方式访问外部设备。将所有的外部设备与CPU直接相连是不现实的，外部设备的中断请求一般经由中断控制器，由中断控制器仲裁后再转发给CPU。如下图所示Arm的中断系统。



其中nIRQ是普通中断，nFIQ是快速中断。 Arm采用的中断控制器叫做GIC，即general interrupt controller。gic包括多个版本，如GICv1（已弃用），GICv2，GICv3，GICv4。简单

起见，我们实验将选用GICv2版本。

为了配置好gicv2中断控制器，与pl011串口一样，我们需要阅读其技术参考手册。访问Arm官网在 这里 下载ARM Generic Interrupt Controller Architecture Specification - version 2.0 的pdf版本。

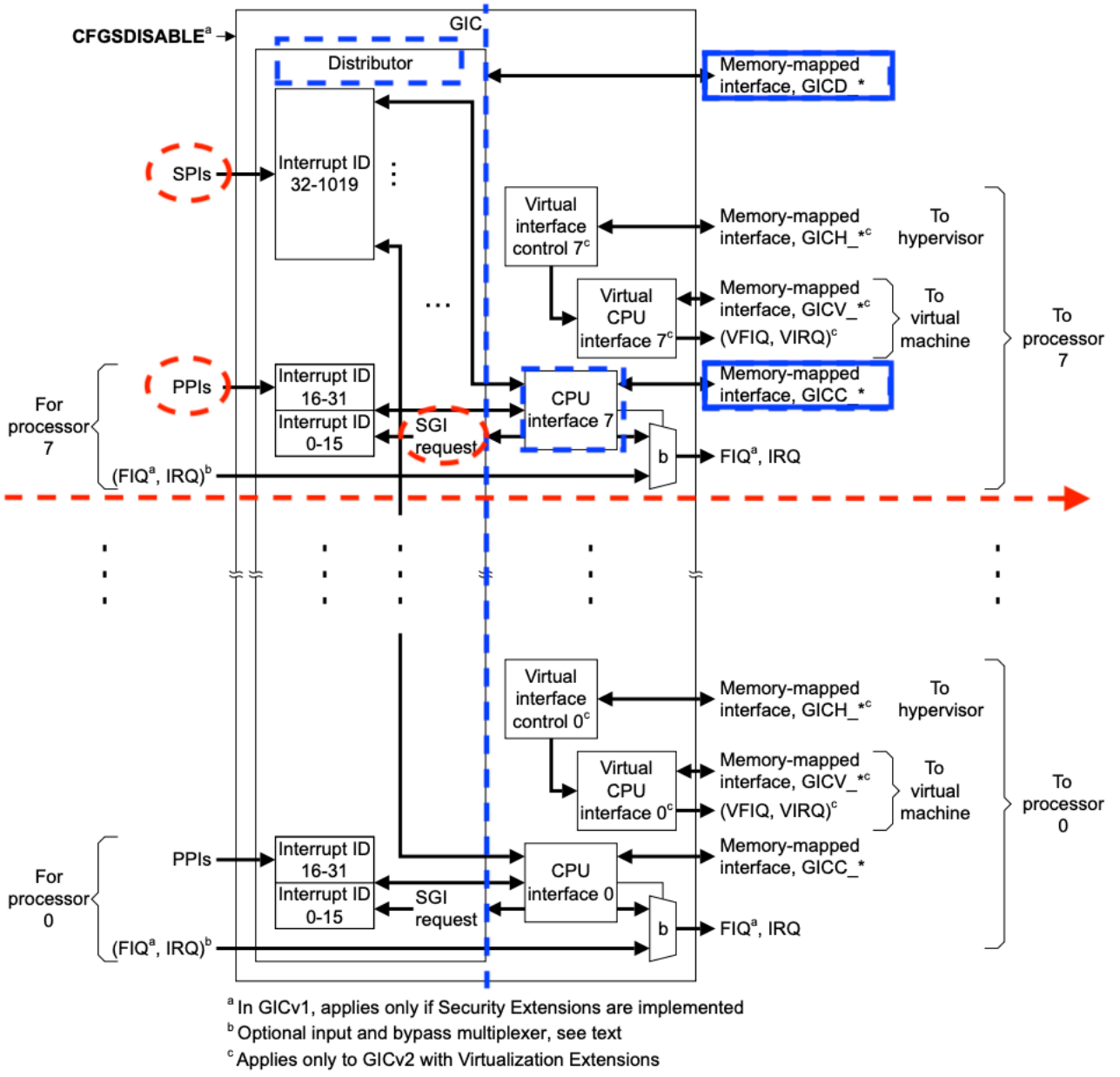


Figure 2-1 GIC logical partitioning

从上图（来源于ARM Generic Interrupt Controller Architecture Specification - version 2.0中的Chapter 2 GIC Partitioning）可以看出：

- GICv2 最多支持8个核的中断管理。
- GIC包括两大主要部分（由图中蓝色虚竖线分隔，Distributor和CPU Interface由蓝色虚矩形框标示），分别是：
  - Distributor，其通过GICD_开头的寄存器进行控制（蓝色实矩形框标示）
  - CPU Interface，其通过GICC_开头的寄存器进行控制（蓝色实矩形框标示）
- 中断类型分为以下几类（由图中红色虚线椭圆标示）：

- SPI：（shared peripheral interrupt），共享外设中断。该中断来源于外设，通过Distributor分发给特定的core，其中断编号为32-1019。从图中可以看到所有核共享SPI。
- PPI：（private peripheral interrupt），私有外设中断。该中断来源于外设，但只对指定的core有效，中断信号只会发送给指定的core，其中断编号为16-31。从图中可以看到每个core都有自己的PPI。
- SGI：（software-generated interrupt），软中断。软件产生的中断，用于给其他的core发送中断信号，其中断编号为0-15。
- virtual interrupt，虚拟中断，用于支持虚拟机。图中也可以看到，因为我们暂时不关心，所以没有标注。
- 此外可以看到(FIQ, IRQ)可通过b进行旁路，我们也不关心。如感兴趣可以查看技术手册了解细节。

此外，由ARM Generic Interrupt Controller Architecture Specification - version 2.0 (section 1.4.2)可知，外设中断可由两种方式触发：

- edge-triggered: 边沿触发，当检测到中断信号上升沿时中断有效。
- level-sensitive：电平触发，当中断源为指定电平时中断有效。

因为soc中，中断有很多，为了方便对中断的管理，对每个中断，附加了中断优先级。在中断仲裁时，高优先级的中断，会优于低优先级的中断，发送给cpu处理。当cpu在响应低优先级中断时，如果此时来了高优先级中断，那么高优先级中断会抢占低优先级中断，而被处理器响应。

由ARM Generic Interrupt Controller Architecture Specification - version 2.0 (section 3.3)可知，GICv2最多支持256个中断优先级。GICv2中规定，所支持的中断优先级别数与GIC的具体实现有关，如果支持的中断优先级数比256少（最少为16），则8位优先级的低位为0，且遵循RAZ/WI（Read-As-Zero, Writes Ignored）原则。

2.GICv2初始化

由下图中virt.dts中intc和timer的部分

```
intc@8000000 {
        phandle = <0x8001>;
        reg = <0x00 0x8000000 0x00 0x10000 0x00 0x8010000 0x00 0x10000>;
        compatible = "arm,cortex-a15-gic";
        ranges;
        #size-cells = <0x02>;
        #address-cells = <0x02>;
        interrupt-controller;
        #interrupt-cells = <0x03>;

        v2m@8020000 {
```

```
                phandle = <0x8002>;
                reg = <0x00 0x8020000 0x00 0x1000>;
                msi-controller;
                compatible = "arm,gic-v2m-frame";
        };
};

timer {
        interrupts = <0x01 0x0d 0x104 0x01 0x0e 0x104 0x01 0x0b 0x104 0x01
0x0a 0x104>;
        always-on;
        compatible = "arm,armv8-timer\0arm,armv7-timer";
};
```

并结合kernel.org中关于 ARM Generic Interrupt Controller 和 ARM architected timer 的devicetree的说明可知：

- intc中的 `reg` 指明GICD寄存器映射到内存的位置为0x8000000，长度为0x10000， GICC寄存器映射到内存的位置为0x8010000，长度为0x10000
- intc中的 `#interrupt-cells` 指明 interrupts 包括3个cells。第一个文档 指明：第一个cell为中断类型，0表示SPI，1表示PPI；第二个cell为中断号，SPI范围为[0-987]，PPI为[0-15]；第三个cell为flags，其中[3:0]位表示触发类型，4表示高电平触发，[15:8]为PPI的cpu中断掩码，每1位对应一个cpu，为1表示该中断会连接到对应的cpu。
- 以timer设备为例，其中包括4个中断。以第二个中断的参数 `0x01 0x0e 0x104` 为例，其指明该中断为PPI类型的中断，中断号14， 路由到第一个cpu，且高电平触发。但注意到PPI的起始中断号为16，所以实际上该中断在GICv2中的中断号应为16 + 14 = 30。

阅读ARM Generic Interrupt Controller Architecture Specification - version 2.0，在其Chapter 4 Programmers' Model部分有关于GICD和GICC寄存器的描述，以及如何使能Distributor和CPU Interfaces的方法。

在 ARM 架构中，GIC（Generic Interrupt Controller）用于管理和处理中断。GIC 主要由两个部分组成：GIC Distributor（GICD）和 GIC CPU Interface（GICC）。
逐个介绍：
1.GIC Distributor（GICD）：

- *GICD 用于分配中断至处理器核心，并提供中断的控制和配置。*
- *在 ARM 架构中，GICD 通常包含一系列寄存器，用于配置中断的使能、优先级、触发方式等。*
- *一些常见的 GICD 寄存器包括：*
- *-GICD_CTLR：用于控制 GICD 的全局使能状态。*
- *GICD_IGROUPR：用于配置中断的组，控制中断是否支持组。*
- *GICD_IPRIORITYR：用于配置中断的优先级。*

- *GICD_ISENABLER：用于使能中断。*
- *GICD_ICENABLER：用于禁用中断。*
- *通过设置 GICD_CTLR 寄存器中的使能位，可以启用或禁用 GICD。*
  2.GIC CPU Interface（GICC）：
- *GICC 用于处理接收到的中断信号，并将其分派给相应的处理器核心。*
- *GICC 包含一系列寄存器，用于中断的控制、优先级处理和中断处理状态的管理。*
- *一些常见的 GICC 寄存器包括：*
- *GICC_CTLR：用于控制 GICC 的全局使能状态。*
- *GICC_PMR：中断优先级屏蔽寄存器，用于控制中断优先级的屏蔽。*
- *GICC_BPR：中断二级优先级屏蔽寄存器，用于控制中断二级优先级的屏蔽。*
- *GICC_IAR：中断挂起寄存器，用于确认中断和获取中断 ID。*
- *GICC_EOIR：中断结束寄存器，用于标记中断处理完成。*
  使能Distributor和CPU Interfaces
  要使能 GIC Distributor 和 GIC CPU Interface，可以分别设置其相应的控制寄存器。
- 对于 GIC Distributor，通过设置 GICD_CTLR 寄存器的使能位来启用或禁用 GICD。
- 对于 GIC CPU Interface，通过设置 GICC_CTLR 寄存器的使能位来启用或禁用 GICC。
  例如，在初始化过程中，可以使用类似如下的代码来使能 GIC Distributor 和GIC CPU Interface：

```
// 初始化 GICv2 的 distributor 和 cpu interface

// 禁用 distributor 和 cpu interface 后进行相应配置

GIC_REG_WRITE(GICD_CTLR, GICD_CTLR_DISABLE);

GIC_REG_WRITE(GICC_CTLR, GICC_CTLR_DISABLE);

// 进行相应的配置，如设置中断优先级、触发方式等

// 启用 distributor 和 cpu interface
GIC_REG_WRITE(GICD_CTLR, GICD_CTLR_ENABLE);

GIC_REG_WRITE(GICC_CTLR, GICC_CTLR_ENABLE);
```

通过配置这些寄存器，可以有效地控制中断的分配和处理，并使系统能够响应和处理各种中断事件。

新建 src/bsp/hwi_init.c 文件，初始化 GIC

```
#include "prt_typedef.h"
#include "os_attr_armv8_external.h"
```

```
#define OS_GIC_VER          2  // 使用GICv2架构

// GIC寄存器基地址定义
#define GIC_DIST_BASE              0x08000000  // 分发器（Distributor）基地址
#define GIC_CPU_BASE               0x08010000  // CPU接口（CPU Interface）基
地址

// 分发器寄存器偏移定义
#define GICD_CTLR                  (GIC_DIST_BASE + 0x0000U)  // 控制寄存器
#define GICD_TYPER                 (GIC_DIST_BASE + 0x0004U)  // 类型寄存器
#define GICD_IIDR                  (GIC_DIST_BASE + 0x0008U)  // 实现标识寄存
器
#define GICD_IGROUPRn              (GIC_DIST_BASE + 0x0080U)  // 中断组寄存器
#define GICD_ISENABLERn            (GIC_DIST_BASE + 0x0100U)  // 中断使能寄存
器（设置）
#define GICD_ICENABLERn            (GIC_DIST_BASE + 0x0180U)  // 中断禁用寄存
器（清除）
#define GICD_ISPENDRn              (GIC_DIST_BASE + 0x0200U)  // 中断挂起寄存
器（设置）
#define GICD_ICPENDRn              (GIC_DIST_BASE + 0x0280U)  // 中断清除挂起
寄存器（清除）
#define GICD_ISACTIVERn            (GIC_DIST_BASE + 0x0300U)  // 中断激活状态
寄存器（读）
#define GICD_ICACTIVERn            (GIC_DIST_BASE + 0x0380U)  // 中断清除激活
寄存器（写）
#define GICD_IPRIORITYn            (GIC_DIST_BASE + 0x0400U)  // 中断优先级寄
存器
#define GICD_ICFGR                 (GIC_DIST_BASE + 0x0C00U)  // 中断配置寄存
器

// CPU接口寄存器偏移定义
#define GICC_CTLR                  (GIC_CPU_BASE  + 0x0000U)  // CPU接口控制
寄存器
#define GICC_PMR                   (GIC_CPU_BASE  + 0x0004U)  // 优先级屏蔽寄
存器
#define GICC_BPR                   (GIC_CPU_BASE  + 0x0008U)  // 二进制点寄存
器
#define GICC_IAR                   (GIC_CPU_BASE + 0x000CU)  // 中断确认寄存器
（读）
#define GICC_EOIR                  (GIC_CPU_BASE + 0x0010U)  // 结束中断寄存器
（写）
#define GICD_SGIR                  (GIC_DIST_BASE + 0x0F00U)  // 软件生成中断
寄存器
```

```
// 宏定义
#define BIT(n)                          (1 << (n))  // 生成位掩码

// 寄存器位定义
#define GICC_CTLR_ENABLEGRP0        BIT(0)  // 使能组0中断
#define GICC_CTLR_ENABLEGRP1        BIT(1)  // 使能组1中断
#define GICC_CTLR_FIQBYPDISGRP0     BIT(5)  // 禁用组0 FIQ旁路
#define GICC_CTLR_IRQBYPDISGRP0     BIT(6)  // 禁用组0 IRQ旁路
#define GICC_CTLR_FIQBYPDISGRP1     BIT(7)  // 禁用组1 FIQ旁路
#define GICC_CTLR_IRQBYPDISGRP1     BIT(8)  // 禁用组1 IRQ旁路

// 寄存器操作宏
#define GIC_REG_READ(addr)          (*(volatile U32 *)((uintptr_t)(addr)))
// 读寄存器
#define GIC_REG_WRITE(addr, data)  (*(volatile U32 *)((uintptr_t)(addr)) =
(U32)(data))  // 写寄存器

// 函数实现

/*
 * 禁用指定中断
 * 参数: intId - 中断号
 * 实现: 清除对应中断在GICD_ICENABLERn寄存器中的位
 */
OS_SEC_L4_TEXT void OsGicDisableInt(U32 intId)
{
    // 计算寄存器索引: 每个寄存器控制32个中断
    U32 regIndex = intId / GICD_ICENABLER_SIZE;
    // 计算寄存器内位偏移
    U32 bitPos = intId % GICD_ICENABLER_SIZE;
    // 构造寄存器地址
    volatile U32 *regAddr = (volatile U32 *)(GICD_ICENABLERn + regIndex *
sizeof(U32));
    // 清除对应位（禁用中断）
    GIC_REG_WRITE(regAddr, 1 << bitPos);
}
```

> [补充说明]
> 在GIC架构中，`GICD_ICENABLERn`寄存器的每个位对应一个中断。写入`1`会禁止该中断的转发
> 该寄存器仅用于控制 **SPI（32~1019）和PPI（16~31）** 的禁用操作。SGI的位映射位于独

> 立的寄存器中。
> `

```
/*
 * 使能指定中断
 * 参数: intId - 中断号
 * 实现: 设置对应中断在GICD_ISENABLERn寄存器中的位
 */
OS_SEC_L4_TEXT void OsGicEnableInt(U32 intId)
{
    U32 regIndex = intId / GICD_ISENABLER_SIZE;
    U32 bitPos = intId % GICD_ISENABLER_SIZE;
    volatile U32 *regAddr = (volatile U32 *)(GICD_ISENABLERn + regIndex *
sizeof(U32));
    GIC_REG_WRITE(regAddr, 1 << bitPos);
}
```

> [补充说明]
> 使能中断时需操作`GICD_ISENABLERn`寄存器（写入`1`），而禁用中断操作`GICD_ICENABLERn`寄存器。两者的寄存器地址和位映射规则类似，但功能相反。

```
/*
 * 直接清除中断的pending状态（触发但未处理）
 * 参数: interrupt - 中断号
 * 实现: 向GICD_ICPENDRn寄存器对应位写入1
 */
OS_SEC_L4_TEXT void OsGicClearIntPending(uint32_t interrupt)
{
    U32 regIndex = interrupt / GICD_ICPENDR_SIZE;
    U32 bitPos = interrupt % GICD_ICPENDR_SIZE;
    GIC_REG_WRITE(GICD_ICPENDRn + regIndex * sizeof(U32), 1 << bitPos);
}

/*
 * 设置中断优先级
 * 参数:
 *   interrupt - 中断号
 *   priority  - 优先级值（0=最高，255=最低）
 * 实现: 将优先级写入GICD_IPRIORITYn寄存器对应字节
 */
```

```
OS_SEC_L4_TEXT void OsGicIntSetPriority(uint32_t interrupt, uint32_t
priority) {
    U32 shift = (interrupt % GICD_IPRIORITY_SIZE) * GICD_IPRIORITY_BITS;
    volatile U32* addr = (volatile U32*)(GICD_IPRIORITYn + (interrupt /
GICD_IPRIORITY_SIZE) * sizeof(U32));
    U32 value = GIC_REG_READ(addr);
    value = (value & ~(0xFF << shift)) | (priority << shift);  // 更新优先级
字段
    GIC_REG_WRITE(addr, value);
}
```

```
/*
 * 设置中断配置
 * 参数:
 *   interrupt - 中断号
 *   config    - 配置值（0=电平敏感，1=边沿触发）
 * 实现: 配置GICD_ICFGR寄存器的对应位
 */
OS_SEC_L4_TEXT void OsGicIntSetConfig(uint32_t interrupt, uint32_t config) {
    U32 shift = (interrupt % GICD_ICFGR_SIZE) * GICD_ICFGR_BITS;
    volatile U32* addr = (volatile U32*)(GICD_ICFGR + (interrupt /
GICD_ICFGR_SIZE)*sizeof(U32));
    U32 value = GIC_REG_READ(addr);
    value = (value & ~(0x03 << shift)) | (config << shift);  // 更新配置字段
    GIC_REG_WRITE(addr, value);
}

/*
 * 中断确认
 * 返回: 中断号（通过读取GICC_IAR获得）
 * 实现: 读取IAR寄存器，返回中断ID
 */
OS_SEC_L4_TEXT U32 OsGicIntAcknowledge(void)
{
    return GIC_REG_READ(GICC_IAR);  // 读取即确认中断
}

/*
 * 标记中断完成
 * 参数: value - 从IAR获得的中断ID
 * 实现: 向EOIR寄存器写入中断ID，清除中断状态
 */
```

```
OS_SEC_L4_TEXT void OsGicIntClear(U32 value)
{
    GIC_REG_WRITE(GICC_EOIR, value);  // 写入中断ID结束处理
}
```

```
/*
 * GIC初始化函数
 * 返回: OS_OK 表示成功
 * 实现流程:
 * 1. 禁用分发器和CPU接口
 * 2. 配置优先级和二进制点
 * 3. 重新启用分发器和CPU接口
 */
U32 OsHwiInit(void)
{
    // 1. 禁用阶段：确保配置期间无中断干扰
    GIC_REG_WRITE(GICD_CTLR, GICD_CTLR_DISABLE);  // 关闭分发器
    GIC_REG_WRITE(GICC_CTLR, GICC_CTLR_DISABLE);  // 关闭CPU接口

    // 2. 配置优先级过滤和二进制点
    GIC_REG_WRITE(GICC_PMR, GICC_PMR_PRIO_LOW);   // 允许所有优先级中断
    GIC_REG_WRITE(GICC_BPR, GICC_BPR_NO_GROUP);   // 无组优先级分割

    // 3. 重新启用阶段
    GIC_REG_WRITE(GICD_CTLR, GICD_CTLR_ENABLE);   // 开启分发器
    GIC_REG_WRITE(GICC_CTLR, GICC_CTLR_ENABLE);   // 开启CPU接口

    return OS_OK;
}
```

在 hwi_init.c 中 OsHwiInit 函数实现 GIC 的初始化，此外还提供了其他函数实现开关指定中断、设置中断属性、确认中断和标记中断完成等功能。

注意：
你需要参照 OsGicIntSetPriority 等函数实现 OsGicEnableInt 和 OsGicDisableInt 函数。

3.使能时钟中断

（1）新建 src/include/prt_config.h

```
/* Tick中断时间间隔，tick处理时间不能超过1/OS_TICK_PER_SECOND(s) */
```

```
#define OS_TICK_PER_SECOND                                  1000
```

> [补充说明]
> **Tick 间隔**：由 `OS_TICK_PER_SECOND` 直接决定，频率为1000HZ，每个Tick的时间间隔为1ms。

（2）新建 src/include/os_cpu_armv8.h

```
// 防止头文件被重复包含的宏定义
#ifndef OS_CPU_ARMV8_H
#define OS_CPU_ARMV8_H

// 包含项目自定义的类型定义头文件
#include "prt_typedef.h"


// ARMv8 异常等级定义

// 当前异常等级（Current Exception Level）掩码值
// ARMv8架构有4个异常等级（EL0-EL3），这里定义了常用的三个：
#define CURRENT_EL_2      0x8    // EL2（Hypervisor特权级，用于虚拟化）
#define CURRENT_EL_1      0x4    // EL1（操作系统内核特权级）
#define CURRENT_EL_0      0x0    // EL0（用户态特权级）


// DAIF 异常屏蔽位定义

// DAIF寄存器控制调试异常（Debug）、中止异常（Abort）、IRQ和FIQ的屏蔽
// 每个位对应一种异常类型的屏蔽状态（1=屏蔽，0=允许）
#define DAIF_DBG_BIT      (1U << 3)  // 调试异常屏蔽位（位3）
#define DAIF_ABT_BIT      (1U << 2)  // 中止异常屏蔽位（位2）
#define DAIF_IRQ_BIT      (1U << 1)  // IRQ中断屏蔽位（位1）
#define DAIF_FIQ_BIT      (1U << 0)  // FIQ快速中断屏蔽位（位0）


// 中断相关定义

// 通用中断屏蔽位

#define INT_MASK          (1U << 7)   // 通用中断屏蔽位
```

```
// 内存屏障指令宏定义

// DSB（Data Synchronization Barrier）：确保所有内存操作完成
// "sy"表示系统级屏障，影响所有处理器核心
#define PRT_DSB()         OS_EMBED_ASM("DSB sy" : : : "memory")

// DMB（Data Memory Barrier）：保证内存访问顺序
#define PRT_DMB()         OS_EMBED_ASM("DMB sy" : : : "memory")

// ISB（Instruction Synchronization Barrier）：刷新指令流水线
#define PRT_ISB()         OS_EMBED_ASM("ISB" : : : "memory")

#endif /* OS_CPU_ARMV8_H */
```

（3）新建 src/bsp/timer.c 文件，对定时器和对应的中断进行配置

```
#include "prt_typedef.h"

#include "prt_config.h"

#include "os_cpu_armv8.h"



U64 g_timerFrequency;

extern void OsGicIntSetConfig(uint32_t interrupt, uint32_t config);

extern void OsGicIntSetPriority(uint32_t interrupt, uint32_t priority);

extern void OsGicEnableInt(U32 intId);

extern void OsGicClearIntPending(uint32_t interrupt);



// 核心定时器初始化函数
void CoreTimerInit(void)
{
    // 配置中断控制器（GIC）相关设置
    OsGicIntSetConfig(30, 0);        // 配置中断号30为电平触发模式（0b00）
    OsGicIntSetPriority(30, 0);      // 设置中断30的优先级为0（最高优先级）
    OsGicClearIntPending(30);        // 清除中断30的挂起状态
```

```
    OsGicEnableInt(30);                 // 使能中断号30（定时器中断）

    // 读取系统定时器频率（ARMv8架构的CNTFRQ_EL0寄存器）
    OS_EMBED_ASM("MRS %0, CNTFRQ_EL0" : "=r"(g_timerFrequency) : : "memory",
"cc");
    // 解释：将系统定时器频率读取到全局变量g_timerFrequency
    // CNTFRQ_EL0: 计数器频率寄存器，返回CPU主频值（单位：Hz）

    // 配置定时器周期（单位：时钟周期数）
    U32 cfgMask = 0x0;                  // 初始化配置掩码
    U64 cycle = g_timerFrequency / OS_TICK_PER_SECOND;
    // 计算定时器周期：总频率 / 每秒时钟节拍数 = 单个时钟节拍的周期数

    // 禁用定时器前先清空控制寄存器（安全操作）
    OS_EMBED_ASM("MSR CNTP_CTL_EL0, %0" : : "r"(cfgMask) : "memory");
    // CNTP_CTL_EL0: 定时器控制寄存器
    // 0x0配置：禁用定时器（enable=0）、屏蔽中断（imask=0）、清除状态（istatus=0）

    PRT_ISB();                          // 插入指令序列屏障，确保内存操作完成
    // ISB（Instruction Synchronization Barrier）保证之前的所有指令执行完毕

    // 设置定时器重载值（定时周期）
    OS_EMBED_ASM("MSR CNTP_TVAL_EL0, %0" : : "r"(cycle) : "memory", "cc");
    // CNTP_TVAL_EL0: 定时器当前值寄存器
    // 当计数器值减到0时触发中断

    // 启用定时器并配置中断使能
    cfgMask = 0x1;                      // 0x1配置：使能定时器（enable=1）
    OS_EMBED_ASM("MSR CNTP_CTL_EL0, %0" : : "r"(cfgMask) : "memory");
    // 此时定时器开始运行，当cycle值递减到0时触发中断

    // 清除DAIF中断屏蔽寄存器中的D位（IRQ屏蔽）
    OS_EMBED_ASM("MSR DAIFCLR, #2");
    // DAIF寄存器：异常屏蔽位
    // #2对应二进制0010，清除D位（IRQ屏蔽），允许外部中断触发
}
```

4.时钟中断处理

（1）-将 prt_vector.S 中的 EXC_HANDLE 5 OsExcDispatch 改为 EXC_HANDLE 5 OsHwiDispatcher，表明我们将对 IRQ 类型的异常（即中断）使用 OsHwiDispatcher 处理。
提示：
需修改为 EXC_HANDLE 5 OsHwiDispatcher ，否则还是 OsExcDispatch 函数处理，仅会输

出 “Catch a exception.” 信息

```
163   .org (VBAR + 0x280)                    // IRQ/vIRQ, Current EL with SP_ELx
164   |    EXC_HANDLE  5 OsHwiDispatcher
```

（2）在 prt_vector.S 中加入 OsHwiDispatcher 处理代码，其类似于之前的 OsExcDispatch，因此不再说明。

```
    .globl OsHwiDispatcher
     .type OsHwiDispatcher, @function
     .align 4
OsHwiDispatcher:
     mrs    x5, esr_el1
     mrs    x4, far_el1
     mrs    x3, spsr_el1
     mrs    x2, elr_el1
     stp    x4, x5, [sp,#-16]!
     stp    x2, x3, [sp,#-16]!

     mov    x0, x1  // 将异常类型（x1）传递给x0（第一个参数）
     mov    x1, sp  // 将当前栈指针（sp）传递给x1（第二个参数）


     bl     OsHwiDispatch // 调用实际的中断处理函数OsHwiDispatch


     ldp    x2, x3, [sp],#16
     add    sp, sp, #16         // 跳过far, esr, HCR_EL2.TRVM==1的时候，EL1不能写far, esr
     msr    spsr_el1, x3        //恢复spsr_el1、elr_el1
     msr    elr_el1, x2
     dsb    sy                   //数据同步屏障，确保内存操作完成
     isb                         //指令同步屏障，确保后续指令从新上下文执行

     RESTORE_EXC_REGS // 恢复上下文

     eret //从异常返回
```

> [补充说明]
> ESR_EL1：记录异常类型（如中断、系统错误）和错误信息。
> FAR_EL1：记录触发异常的内存地址（如数据访问中止）。
> SPSR_EL1：保存异常发生时的CPU状态（如条件标志、中断屏蔽位）。

> ELR_EL1：保存异常返回地址（下一条指令地址）。
> stp：将寄存器对保存到栈中，!表示栈指针自动调整。

（3）在 prt_exc.c 中引用头文件 os_attr_armv8_external.h ， os_cpu_armv8.h ，OsHwiDispatch 处理 IRQ 类型的中断。

```
// 声明一个外部函数，用于处理系统滴答定时器中断
extern void OsTickDispatcher(void);

// 定义内联函数处理活动的中断（强制内联优化）
OS_SEC_ALW_INLINE INLINE void OsHwiHandleActive(U32 irqNum)
{
    switch(irqNum){          // 根据中断号进行分支处理
        case 30:             // 假设30号中断对应系统滴答定时器
            OsTickDispatcher();// 调用系统滴答处理函数
            // PRT_Printf("."); // 调试打印（注释状态）
            break;           // 跳出switch
        default:             // 其他中断号
            break;           // 无操作直接返回
    }
}

// 声明外部函数：读取中断控制器确认的中断号
extern  U32 OsGicIntAcknowledge(void);
// 声明外部函数：向中断控制器写入清除命令
extern void OsGicIntClear(U32 value);

/*
中断处理入口函数（L0级内核文本段）
描述：
- 此处已通过外部代码关闭全局中断
- 主要完成中断确认->分发->清除的标准流程
参数说明：
- excType: 异常类型（当前未使用）
- excRegs: 异常寄存器上下文（当前未使用）
*/
OS_SEC_L0_TEXT void OsHwiDispatch( U32 excType, struct ExcRegInfo *excRegs)
{
    // 1. 中断确认操作（获取中断号）
    U32 value = OsGicIntAcknowledge();  // 从GIC控制器获取中断信息

    // 2. 解析中断号和核心号
    U32 irq_num = value & 0x1ff;         // 低9位为中断号（GIC标准定义）
```

```
    U32 core_num = value & 0xe00;        // 高3位为目标CPU核心编号

    // 3. 调用中断处理调度函数
    OsHwiHandleActive(irq_num);          // 处理具体中断业务

    // 4. 清除中断挂起状态（需同时指定中断号和目标核心）
    OsGicIntClear(irq_num | core_num);  // 组合中断号+核心号生成清除命令
}
```

src/bsp/os_attr_armv8_external.h 头文件可以在 此处 下载。

（4）新建 src/kernel/tick/prt_tick.c 文件，提供 OsTickDispatcher 时钟中断处理函数。

```
#include "os_attr_armv8_external.h"   // 包含ARMv8架构相关属性定义（内存模型/异常处理等）
#include "prt_typedef.h"              // 包含项目自定义类型定义（如U64等基础类型）
#include "prt_config.h"               // 包含系统配置参数（如OS_TICK_PER_SECOND）
#include "os_cpu_armv8_external.h"    // 包含ARMv8 CPU相关操作接口

// 声明外部定时器频率变量（由硬件抽象层提供）
extern U64 g_timerFrequency;

/* Tick计数器（BSS段存储 - 未初始化全局变量）
 * 用于记录系统启动后经历的时钟节拍总数
 * 属于核心内核系统（位于src/core/kernel/sys/prt_sys.c） */
OS_SEC_BSS U64 g_uniTicks;

/*
* Tick中断处理函数
* 功能：
* 1. 维护系统时钟节拍计数
* 2. 更新定时器中断周期
* 3. 触发任务超时检查、软件定时器扫描等周期性操作
*/
OS_SEC_TEXT void OsTickDispatcher(void)
{
    uintptr_t intSave;  // 中断保存变量（用于保存中断状态）

    // 原子操作开始：关闭中断保证计数操作的原子性
    intSave = OsIntLock();

    g_uniTicks++;        // 递增系统tick计数器
```

```
    // 恢复中断状态（原子操作结束）
    OsIntRestore(intSave);

    // 计算定时器重载值：
    // 用定时器频率除以每秒tick数，得到每个tick周期对应的计时器计数值
    U64 cycle = g_timerFrequency / OS_TICK_PER_SECOND;

    /* ARMv8汇编指令：
       将cycle值写入CNTP_TVAL_EL0寄存器
       该寄存器控制物理定时器（EL0特权级）的下次触发时间
       "r"(cycle) 表示将cycle值作为输入操作数
       clobber列表声明内存和条件码寄存器可能被修改 */
    OS_EMBED_ASM("MSR CNTP_TVAL_EL0, %0" : : "r"(cycle) : "memory", "cc");
}

/*
* 获取当前系统tick计数
* 返回值：
*   U64类型值，表示系统启动后经历的时钟节拍总数
* 应用场景：
*   任务调度统计
*   超时检测
*   性能分析 */
OS_SEC_L2_TEXT U64 PRT_TickGetCount(void)
{
    return g_uniTicks;  // 直接返回原子计数器值
}
```

关键点补充：

1. *原子操作：通过 `OsIntLock()/OsIntRestore()` 实现临界区保护，防止多核环境下计数器竞争*
2. *定时器配置：`CNTP_TVAL_EL0` 是ARMv8的物理定时器寄存器，写入值代表下次中断前需要经过的计时周期数*
3. *时间计算关系：`cycle = 定时器频率 / 每秒tick数`，例如若定时器频率10MHz且每秒10000tick，则每个tick周期1ms*
4. *内存屏障：汇编中的"memory" clobber会阻止编译器优化内存访问顺序，确保内存操作可见性*
5. *权限级别：EL0表示异常级别0（用户态）

注意需将 hwi_init.c timer.c prt_tick.c 等文件加入构建系统。

> [提示]
> src/kernel， src/kernel/tick 目录下均需加入 CMakeLists.txt， src/ 和 src/bsp/ 下的 CMakeLists.txt 需修改。其中，
> src/kernel/tick/CMakeLists.txt 类似 src/bsp/CMakeLists.txt
> src/kernel/CMakeLists.txt 内容为： add_subdirectory(tick)
> src/CMakeLists.txt 需修改增加include目录、包含子目录和编译目标：

```
 include_directories(

    ${CMAKE_CURRENT_SOURCE_DIR}/include   # 增加 src/include 目录

    ${CMAKE_CURRENT_SOURCE_DIR}/bsp

)



add_subdirectory(bsp)

add_subdirectory(kernel) # 增加 kernel 子目录



list(APPEND OBJS $<TARGET_OBJECTS:bsp> $<TARGET_OBJECTS:tick>) # 增加
$<TARGET_OBJECTS:tick> 目标

add_executable(${APP} main.c ${OBJS})
```

> 后续实验中若新增文件加入构建系统不再赘述，请参照此处。

（5）在 OsTickDispatcher 中调用了 OsIntLock 和 OsIntRestore 函数，这两个函数用于关中断和开中断。简单起见，将其放入 prt_exc.c 中。

```
/*
* 描述：开启全局可屏蔽中断。
*/
OS_SEC_L0_TEXT uintptr_t PRT_HwiUnLock(void) // 位于安全级别L0的代码段，用于解锁
（开启）全局中断
{
    uintptr_t state = 0;  // 用于保存当前中断状态

    OS_EMBED_ASM(
        "mrs %0, DAIF      \n"   // 将DAIF寄存器的值读取到state变量（DAIF：异常
```

```
屏蔽位寄存器）
        "msr DAIFClr, %1   \n"   // 清除DAIF寄存器中的IRQ位（通过DAIFClr寄存器，参数为DAIF_IRQ_BIT）
        : "=r"(state)            // 输出操作数：state变量接收DAIF原值
        : "i"(DAIF_IRQ_BIT)      // 输入操作数：固定输入值为DAIF_IRQ_BIT（用于清除IRQ位）
        : "memory", "cc");       // 告知编译器内存和条件码寄存器可能被修改

    return state & INT_MASK;     // 返回原始中断状态（仅保留INT_MASK定义的位，屏蔽其他无关位）
}

/*
* 描述：关闭全局可屏蔽中断。
*/
OS_SEC_L0_TEXT uintptr_t PRT_HwiLock(void) // 位于安全级别L0的代码段，用于加锁（关闭）全局中断
{
    uintptr_t state = 0;
    OS_EMBED_ASM(
        "mrs %0, DAIF      \n"   // 读取当前DAIF寄存器值到state
        "msr DAIFSet, %1   \n"   // 设置DAIF寄存器中的IRQ位（通过DAIFSet寄存器，参数为DAIF_IRQ_BIT）
        : "=r"(state)            // 输出state变量保存修改前的DAIF值
        : "i"(DAIF_IRQ_BIT)      // 输入固定值DAIF_IRQ_BIT（用于设置IRQ位）
        : "memory", "cc");       // 声明内存和条件码寄存器可能被修改

    return state & INT_MASK;     // 返回原始中断状态（仅保留关键位）
}

/*
* 描述：恢复原中断状态寄存器。
*/
OS_SEC_L0_TEXT void PRT_HwiRestore(uintptr_t intSave) // 参数intSave为之前保存的中断状态
{
    if ((intSave & INT_MASK) == 0) {  // 判断原始状态：若INT_MASK位未置位，表示此前中断是开启的
        OS_EMBED_ASM(
            "msr DAIFClr, %0\n"       // 若原状态为开启，则再次清除IRQ位（恢复开启状态）
            :
            : "i"(DAIF_IRQ_BIT)       // 输入DAIF_IRQ_BIT以清除IRQ位
```

```
            : "memory", "cc");
    } else {
        OS_EMBED_ASM(
            "msr DAIFSet, %0\n"       // 若原状态为关闭，则设置IRQ位（恢复关闭状
态）
            :
            : "i"(DAIF_IRQ_BIT)       // 输入DAIF_IRQ_BIT以设置IRQ位
            : "memory", "cc");
    }
    return;
}
```

头文件 src/bsp/os_cpu_armv8_external.h

```
// 防止头文件被重复包含的预处理指令
#ifndef OS_CPU_ARMV8_EXTERNAL_H
// 如果未定义过该标识符，则定义它并继续编译
#define OS_CPU_ARMV8_EXTERNAL_H

// 声明中断解锁函数（返回值用于保存中断状态）
extern uintptr_t PRT_HwiUnLock(void);
/*
extern 表示函数实现在其他文件
uintptr_t 是足够大的无符号整型，用于存储指针值
返回值可能用于保存中断状态以便后续恢复
*/

// 声明中断加锁函数（返回值用于保存中断状态）
extern uintptr_t PRT_HwiLock(void);
/*
典型用途：通过禁用中断实现临界区保护
返回值通常用于后续恢复中断状态
*/

// 声明中断状态恢复函数
extern void PRT_HwiRestore(uintptr_t intSave);
/*
参数 intSave 应是由 PRT_HwiLock 返回的状态值
用于精确恢复中断使能状态
*/
```

```
// 定义用户级中断解锁宏
#define OsIntUnLock() PRT_HwiUnLock()
/*
封装底层函数为更语义化的接口
用户调用 OsIntUnLock() 实际执行 PRT_HwiUnLock()
*/

// 定义用户级中断加锁宏
#define OsIntLock() PRT_HwiLock()
/*
保持接口一致性
用户通过这个宏进行中断加锁操作
*/

// 定义中断状态恢复宏
#define OsIntRestore(intSave) PRT_HwiRestore(intSave)
/*
参数传递机制：将保存的中断状态原样传给底层函数
保证中断恢复操作的原子性
*/

#endif  // 结束头文件保护
```

5.读取系统Tick值

（1）新建 prt_tick.h，声明 Tick 相关的接口函数.

```
#ifndef PRT_TICK_H

#define PRT_TICK_H

#include "prt_typedef.h"

extern U64 PRT_TickGetCount(void);

#endif /* PRT_TICK_H */
```

（2）main.c 修改为：

```
#include "prt_typedef.h"
```

```
#include "prt_tick.h"



extern U32 PRT_Printf(const char *format, ...);

extern void PRT_UartInit(void);

extern void CoreTimerInit(void);

extern U32 OsHwiInit(void);

U64 delay_time = 10000;

S32 main(void)
{

    // 初始化GIC

    OsHwiInit();

    // 启用Timer

    CoreTimerInit();
    PRT_UartInit();



    PRT_Printf("            _       _ _____      _             _
_   _ _   _ _   _           \n");

    PRT_Printf("  _ __ ___ (_)_ __ (_) ____|   _| | ___ _ __  | |__  _   _
 | | | | \\ | | | | | ___ _ __ \n");

    PRT_Printf(" | '_ ` _ \\| | '_ \\| |  _|| | | | |/ _ \\ '__| | '_ \\| |
| | | |_| |  \\| | | | |/ _ \\ '__|\n");

    PRT_Printf(" | | | | | | | | | | | |__| |_| | |  __/ |    | |_) | |_| |
|  _  | |\\  | |_| |  __/ |   \n");

    PRT_Printf(" |_| |_| |_|_|_| |_|_|_____\\__,_|_|\\___|_|    |_.__/ \\__,
| |_| |_|_| \\_|\\___/ \\___|_|   \n");
```

```
    PRT_Printf("                                                    |___/
                            \n");



    PRT_Printf("ctr-a h: print help of qemu emulator. ctr-a x: quit
emulator.\n\n");
    for(int i = 0; i < 10; i++)

    {



        U32 tick = PRT_TickGetCount();

        PRT_Printf("[%d] current tick: %u\n", i, tick);



        //delay

        int delay_time = 10000000;  // 根据自己机器计算能力不同调整该值

        while(delay_time>0){

            PRT_TickGetCount();  //消耗时间，防止延时代码被编译器优化

            delay_time--;

        }
    }
    while(1);

    return 0;

}
```

输出如下：tick都为0 ，暂时还没有实现hwi_init.c 中的两个函数。

```
(base) xiaoye@localhost:~/OSlab/lab5$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

[0] current tick: 0
[1] current tick: 0
[2] current tick: 0
[3] current tick: 0
[4] current tick: 0
[5] current tick: 0
[6] current tick: 0
[7] current tick: 0
[8] current tick: 0
[9] current tick: 0
```

五、lab5 作业

作业1

实现 hwi_init.c 中缺失的 OsGicEnableInt 和 OsGicDisableInt 函数。
我们可以从 ARM Generic Interrupt Controller Architecture Specification - version 2.0 中找到有关使能中断和清除中断的寄存器，如下图：

GICv2 suggests replacement register names for GICv1 registers. Table B-1 shows the GICv1 names and the GICv2 suggested replacement names for the registers in the Distributor.

Table B-1 Replacement names for the registers in the Distributor

Register	GICv2 name	GICv1 name
Distributor Control	GICD_CTLR	ICDDCR
Interrupt Controller Type	GICD_TYPER	ICDICTR
Distributor Implementer Identification	GICD_IIDR	ICDIIDR
Interrupt Group	GICD_IGROUPRn	ICDISRn
Interrupt Set-Active	GICD_ISACTIVERn	ICDABRn
Interrupt Set-Enable	GICD_ISENABLERn	ICDISERn
Interrupt Clear-Enable	GICD_ICENABLERn	ICDICERn
Interrupt Set-Pending	GICD_ISPENDRn	ICDISPRn
Interrupt Clear-Pending	GICD_ICPENDRn	ICDICPRn
Interrupt Priority	GICD_IPRIORITYRn	ICDIPRn
Interrupt Processor Targets	GICD_ITARGETSRn	ICDIPTRn
Interrupt Configuration	GICD_ICFGRn	ICDICRn
Software Generated Interrupt	GICD_SGIR	ICDSGIR
Identification	-	-

GICD_ISENABLERn（GICD_ISENABLER为GIC支持的每个中断提供一个设置使能位。向一个设置使能位写入1会从分发器向CPU接口转发相应中断。）

4.3.5 Interrupt Set-Enable Registers, GICD_ISENABLERn

The GICD_ISENABLER characteristics are:

Purpose

The GICD_ISENABLERs provide a Set-enable bit for each interrupt supported by the GIC. Writing 1 to a Set-enable bit enables forwarding of the corresponding interrupt from the Distributor to the CPU interfaces. Reading a bit identifies whether the interrupt is enabled.

Usage constraints

A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Set-enable bits for the lockable SPIs that are configured as Group 0, see *Configuration lockdown* on page 4-82.

Whether implemented SGIs are permanently enabled, or can be enabled and disabled by writes to GICD_ISENABLER0 and GICD_ICENABLER0, is IMPLEMENTATION DEFINED.

Configurations

These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented GICD_ISENABLERs is (GICD_TYPER.ITLinesNumber+1). The implemented GICD_ISENABLERs number upwards from GICD_ISENABLER0.

In a multiprocessor implementation, GICD_ISENABLER0 is banked for each connected processor. This register holds the Set-enable bits for interrupts 0-31.

Attributes

See the register summary in Table 4-1 on page 4-75.

Figure 4-6 shows the GICD_ISENABLER bit assignments.

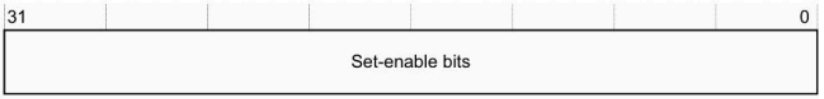


Figure 4-6 GICD_ISENABLER bit assignments

GICD_ICPENDRn（GICD_ICPENDR为GIC支持的每个中断提供一个清除挂起位。向一个清除挂起位写入1会清除相应外设中断的挂起状态。）

4.3.8 Interrupt Clear-Pending Registers, GICD_ICPENDRn

The GICD_ICPENDR characteristics are:

Purpose

The GICD_ICPENDRs provide a Clear-pending bit for each interrupt supported by the GIC. Writing 1 to a Clear-pending bit clears the pending state of the corresponding peripheral interrupt. Reading a bit identifies whether the interrupt is pending.

Usage constraints

A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Clear-pending bits for the lockable SPIs that are configured as Group 0, see *Configuration lockdown* on page 4-82.

Clear-pending bits for SGIs are read-only and ignore writes.

Configurations

These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented GICD_ICPENDRs is (GICD_TYPER.ITLinesNumber+1). The implemented GICD_ICPENDRs number upwards from GICD_ICPENDR0.

In a multiprocessor implementation, GICD_ICPENDR0 is banked for each connected processor. This register holds the Clear-pending bits for interrupts 0-31.

Attributes

See the register summary in Table 4-1 on page 4-75.

Figure 4-9 shows the GICD_ICPENDR bit assignments.

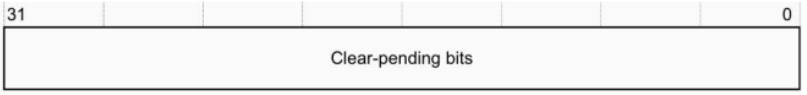


Figure 4-9 GICD_ICPENDR bit assignments

Table 4-12 on page 4-100 shows the GICD_ICPENDR bit assignments.

整体思路

（1）确定是哪个寄存器（同种类型的寄存器可能有多个），并构造相应的地址；
（2）确定是该中断对应的寄存器中的位；
（3）清除对应的位。

```
    // 计算寄存器索引：每个寄存器控制32个中断
    U32 regIndex = intId / GICD_ICENABLER_SIZE;
    // 计算寄存器内位偏移
    U32 bitPos = intId % GICD_ICENABLER_SIZE;
    // 构造寄存器地址
    volatile U32 *regAddr = (volatile U32 *)(GICD_ICENABLERn + regIndex *
sizeof(U32));
    // 清除对应位（禁用中断）
    GIC_REG_WRITE(regAddr, 1 << bitPos);
```

OsGicEnableInt 和 OsGicDisableInt 函数实现如下：

```
13   #define GICD_ISENABLERn             (GIC_DIST_BASE + 0x0100U)
25   #define GICD_ISENABLER_SIZE                  32
```

```
/**
 * 使能指定中断
 * 参数：intId - 中断号（0-1019 for GICv2）
 * 实现：设置对应中断在GICD_ISENABLERn寄存器中的位
 */
OS_SEC_L4_TEXT void OsGicEnableInt (U32 intId)
{
    // 计算寄存器索引
    U32 regIndex = intId / GICD_ISENABLER_SIZE;
    // 计算寄存器内位偏移
    U32 bitPos = intId %GICD_ISENABLER_SIZE;

    // 获取目标寄存器地址（GICD_ISENABLERn基址 + 寄存器偏移）
    volatile U32 *regAddr = (volatile U32 *)(GICD_ISENABLERn + regIndex *
sizeof(U32));

    // 通过位或操作设置对应中断使能位（避免影响其他中断位）
    *regAddr |= (1U << bitPos);
}
```

```
14   #define GICD_ICENABLERn              (GIC_DIST_BASE + 0x0180U)
26   #define GICD_ICENABLER_SIZE                  32
```

```
/**
 * 禁用指定中断
 * 参数: interrupt - 中断号
 * 实现: 在对应中断的GICD_ICENABLERn寄存器中设置位（置1即禁用中断）
 */
OS_SEC_L4_TEXT void OsGicDisableInt(U32 interrupt)
{
    /* 计算寄存器索引：每个ICENABLER寄存器控制32个中断 */
    U32 regIndex = interrupt / GICD_ICENABLER_SIZE;  // GIC标准每个ICENABLER
控制32个中断

    /* 计算寄存器内位偏移：对应中断在寄存器中的位位置 */
    U32 bitPos = interrupt % GICD_ICENABLER_SIZE;

    /* 计算寄存器物理地址：ICENABLER寄存器组基址 + 寄存器索引偏移 */
    volatile U32 *regAddr = (volatile U32 *)(GICD_ICENABLERn + regIndex *
4);

    /* 通过写1清除对应位（GIC规范要求写1禁用中断） */
    GIC_REG_WRITE(regAddr, 1U << bitPos);  // 使用1的移位更直观
}
```

想要实现一个比较慢的输出，故我将延时设置成了250000000。可以看到平均tick间隔为4633 ticks：

```
(base) xiaoye@localhost:~/OSlab/lab5$ sh runMiniEuler.sh
qemu-system-aarch64 -machine virt,gic-version=2 -m 1024M -cpu cortex-a53 -nographic -kernel build/miniEuler -s
ctr-a h: print help of qemu emulator. ctr-a x: quit emulator.

[0] current tick: 7
[1] current tick: 4884
[2] current tick: 9802
[3] current tick: 14651
[4] current tick: 19582
[5] current tick: 24476
[6] current tick: 29185
[7] current tick: 34139
[8] current tick: 39416
[9] current tick: 44337
```