



Nuclei™ N200 系列

处理器内核快速应用手册

版权声明

版权所有 © 2018–2019 芯来科技（Nuclei System Technology）有限公司。保留所有权利。

Nuclei™是芯来科技公司拥有的商标。本文件使用的所有其他商标为各持有公司所有。

本文件包含芯来科技公司的机密信息。使用此版权声明为预防作用，并不意味着公布或披露。未经芯来科技公司书面许可，不得以任何形式将本文的全部或部分信息进行复制、传播、转录、存储在检索系统中或翻译成任何语言。

本文文件描述的产品将不断发展和完善；此处的信息由芯来科技提供，但不做任何担保。

本文件仅用于帮助读者使用该产品。对于因采用本文件的任何信息或错误使用产品造成的任何损失或损害，芯来科技概不负责。

联系我们

若您有任何疑问，请通过电子邮件 support@nucleisys.com 联系芯来科技。

修订历史

| 版本号 | 修订日期 | 修订的章节 | 修订的内容 |
|-----|------------|-------|---------|
| 1.0 | 2019/05/06 | N/A | 1. 初始版本 |

目录

| | |
|---------------------------------------|-----------|
| 版权声明..... | 0 |
| 联系我们..... | 0 |
| 修订历史..... | 1 |
| 图片清单..... | 4 |
| 1. 概述 | 5 |
| 2. 背景知识——N200 系列处理器内核..... | 6 |
| 2.1. 指令集介绍..... | 6 |
| 2.2. CSR 寄存器 | 6 |
| 2.3. N200 系列内核特权架构介绍..... | 6 |
| 2.4. N200 系列内核异常机制..... | 7 |
| 2.5. N200 系列内核 NMI 机制 | 8 |
| 2.6. N200 系列内核中断机制..... | 10 |
| 2.7. N200 系列内核 TIMER 和 ECLIC 介绍 | 11 |
| 2.7.1. TIMER 简介..... | 11 |
| 2.7.2. ECLIC 简介 | 11 |
| 2.7.3. 中断屏蔽..... | 12 |
| 2.7.4. 中断级别、优先级与仲裁..... | 13 |
| 2.7.5. 中断服务程序..... | 13 |
| 2.7.6. 中断嵌套..... | 13 |
| 2.7.7. 中断咬尾..... | 16 |
| 3. 背景知识——N200 系列内核配套 SOC..... | 19 |
| 3.1. N200 系列内核配套 SoC 框图 | 19 |
| 3.2. N200 系列内核配套 SoC 储存资源..... | 19 |
| 3.3. N200 系列内核配套 SoC 外设介绍..... | 20 |
| 4. 应用实例解析——DEMO_ECLIC..... | 21 |
| 4.1. DEMO_ECLIC 简述..... | 21 |
| 4.2. DEMO_ECLIC 程序代码结构..... | 21 |
| 4.3. DEMO_ECLIC 程序解析..... | 23 |
| 4.3.1. 主程序流程图..... | 23 |
| 4.3.2. 计时器中断处理函数流程图..... | 24 |
| 4.3.3. 用户按键中断处理函数流程图..... | 25 |
| 4.3.4. 快速移植中断应用..... | 26 |

| | |
|---|-----------|
| 5. DEMO_ECLIC 软硬件快速启动 | 31 |
| 5.1. 第一步：硬件线缆连接 | 31 |
| 5.2. 第二步：设置下载器在 LINUX 系统中的 USB 权限 | 32 |
| 5.3. 第三步：连接外部中断引脚至拨码开关引脚 | 32 |
| 5.4. 第四步：将 DEMO_ECLIC 程序下载至评估板并运行 | 33 |
| 6. 快速应用——移植实时操作系统 | 35 |
| 6.1. 实时操作系统简述 | 35 |
| 6.2. 常用实时操作系统概述 | 35 |
| 6.3. FREERTOS 简介 | 37 |
| 6.4. NUCLEI N200 移植 FREERTOS | 38 |
| 6.4.1. N200-SDK 中 FreeRTOS 程序代码结构 | 38 |
| 6.4.2. FreeRTOS 原理和移植介绍 | 39 |
| 6.4.3. RTOS 操作系统的基本原理 | 39 |
| 6.4.4. FreeRTOS 源码解析和移植介绍 | 41 |
| 6.4.5. FreeRTOS 中任务与中断的关系 | 43 |
| 6.4.6. 运行 FreeRTOS | 43 |

图片清单

| | |
|---|----|
| 图 2-1 N200 系列内核特权架构图 | 7 |
| 图 2-2 进入异常处理模式图 | 8 |
| 图 2-3 退出异常处理模式图 | 8 |
| 图 2-4 进入 NMI 处理模式图 | 9 |
| 图 2-5 退出 NMI 处理模式图 | 9 |
| 图 2-6 进入中断模式图 | 10 |
| 图 2-7 退出中断模式图 | 11 |
| 图 2-8 ECLIC 关系结构图 | 12 |
| 图 2-9 中断仲裁示意图 | 13 |
| 图 2-10 中断嵌套示意图 | 14 |
| 图 2-11 计时器中断串口打印信息图 | 15 |
| 图 2-12 按键 1 中断串口打印信息图 | 15 |
| 图 2-13 按键 2 中断串口打印信息图 | 15 |
| 图 2-14 按键 1 “打断” 计时器中断串口打印信息图 | 15 |
| 图 2-15 出现三层中断嵌套串口打印信息图 | 16 |
| 图 2-16 中断咬尾示意图 | 16 |
| 图 3-1 N200 系列内核配套 SoC 结构图 | 19 |
| 图 4-1 主程序流程图 | 23 |
| 图 4-2 计时器中断处理函数流程图 | 24 |
| 图 4-3 按键中断处理函数流程图 | 25 |
| 图 5-1 评估板及下载器线缆连接图 | 32 |
| 图 5-2 连接按键至 MCU_GPIO 图 | 33 |
| 图 5-3 运行 DEMO_ECLIC 示例后于主机串口终端上显示信息 | 34 |
| 图 6-1 2017 年各种操作系统的使用数量统计图 | 38 |
| 图 6-2 裸机程序运行过程图 | 40 |
| 图 6-3 RTOS 程序运行过程图 | 40 |
| 图 6-4 程序运行串口打印信息图 | 44 |

1. 概述

N200 内核用户将通过本文，快速掌握该内核应用开发的必要知识点。包括：内核及配套 SoC 的背景知识介绍、应用实例解析及其软硬件快速启动方法。

中断应用,是 N200 内核应用中的重点和难点。本文第一个应用实例，将着重讲解 N200 内核的中断使用方法。通过对例程解析，使用户在最短的时间内，熟练掌握中断应用并移植到定制化应用项目的开发中。

实时操作系统移植，是嵌入式控制器应用开发中极其重要的一环。本文第二个应用实例，将围绕 N200 内核及其配套 SoC，通过对 FreeRTOS 实时操作系统的移植，使用户能够在了解实时操作系统移植相关知识点的同时，加深对 N200 内核运行机制的理解和掌握。

为了使用户能对以上介绍的两个应用实例，进行快速启动和实践操作。本文将基于芯来科技推出的蜂鸟 FPGA 开发板，介绍硬件快速启动方法和应用实例的调试运行流程。

2. 背景知识——N200 系列处理器内核

Nuclei N200 处理器内核（简称 N200 内核）主要面向极低功耗与极小面积的场景而设计，非常适合于替代传统的 8051 内核或者 ARM Cortex-M 系列内核应用于 IoT 或其他低功耗场景。

详情请参见《Nuclei_N200 系列简明数据手册》了解其详情。

2.1. 指令集介绍

N200 内核遵循标准的 RISC-V 指令集标准。

详情请参见《Nuclei_N200 系列指令架构手册》第 1.1 节和第 1.2 节，了解其详情。

2.2. CSR 寄存器

RISC-V 的架构中定义了一些控制和状态寄存器（Control and Status Register, CSR），用于配置或记录一些运行的状态。CSR 寄存器是处理器核内部的寄存器，使用其专有的 12 位地址编码空间。详情请参见《Nuclei_N200 系列指令架构手册》第 1.3 节了解其详情。

2.3. N200 系列内核特权架构介绍

N200 系列内核支持两个特权模式（Privilege Modes）：

- 机器模式（Machine Mode）是必须的模式，该 Privilege Mode 的编码是 0x3。
- 用户模式（User Mode）是可配置的模式，该 Privilege Mode 的编码是 0x0。
 - 如果配置了 User Mode，则也支持 PMP（Physical Memory Protection）单元对物理地址区间进行隔离和保护。

注意：N200 是一个处理器内核系列，具体每一款处理器内核（譬如 N201，N205 等）支持的 Privilege Modes 组合可能略有差异，请参见《Nuclei_N200 系列简明数据手册》了解其详情。

机器模式包含四种机器子模式（Machine Sub-Mode）。其特权模式划分，如图 2-1 所示：

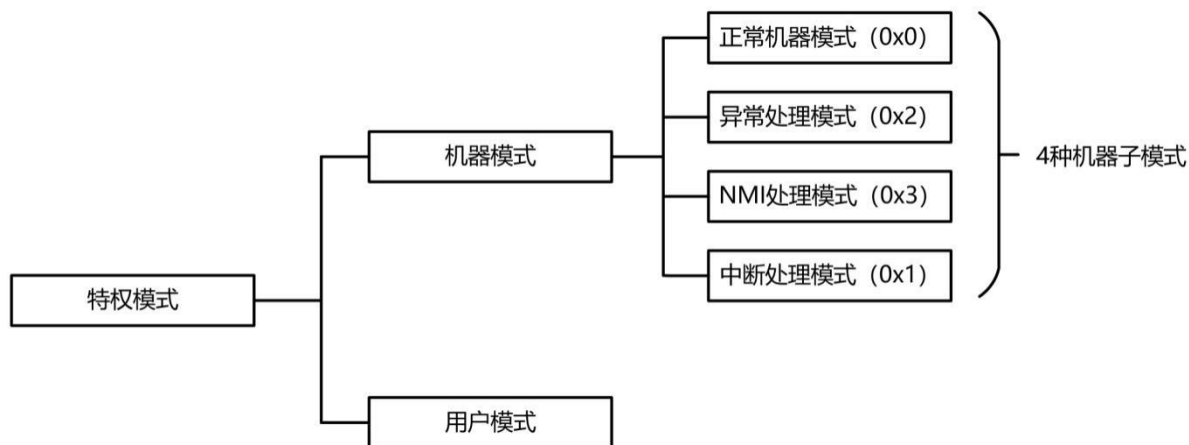


图 2-1 N200 系列内核特权架构图

详情请参见《Nuclei_N200 系列指令架构手册》第 2.2 节了解详情。

2.4. N200 系列内核异常机制

异常（Exception）机制，即处理器核在顺序执行程序指令流的过程中突然遇到了异常的事情而中止执行当前的程序，转而去处理该异常。异常发生后，处理器会进入异常服务处理程序。

图 2-2 和图 2-3 分别介绍归纳了进入和退出异常模式的相关操作，详情请参见《Nuclei_N200 系列指令架构手册》第 3 章了解其详情。

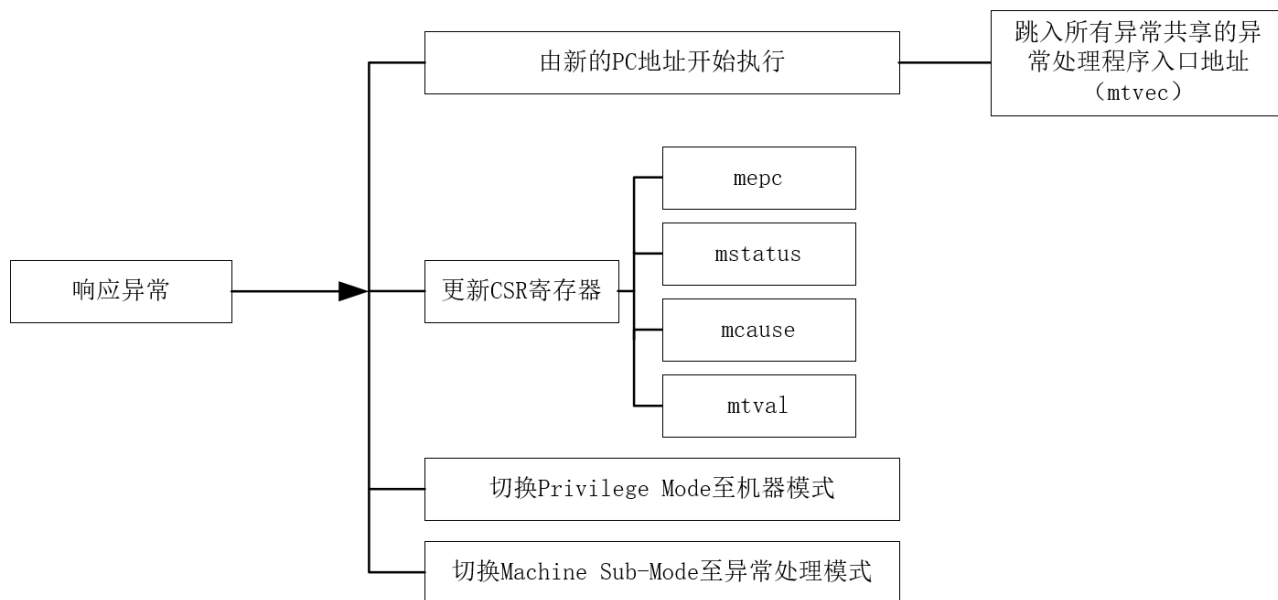


图 2-2 进入异常处理模式图

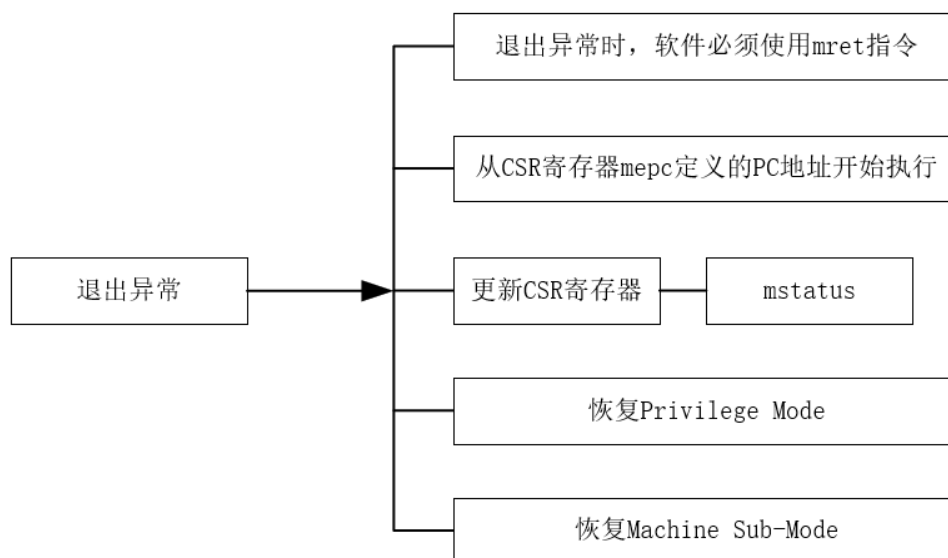


图 2-3 退出异常处理模式图

2.5. N200 系列内核 NMI 机制

NMI (Non-Maskable Interrupt) 是处理器内核的一根特殊的输入信号，往往用于指示系统层

面的紧急错误（譬如外部的硬件故障等）。在遇到 NMI 之后，处理器内核应该立即中止执行当前的程序，转而去处理该 NMI 错误。

图 2-4 和图 2-5 分别介绍归纳了进入和退出 NMI 模式的相关操作，详情请参见《Nuclei_N200 系列指令架构手册》第 4 章了解其详情。

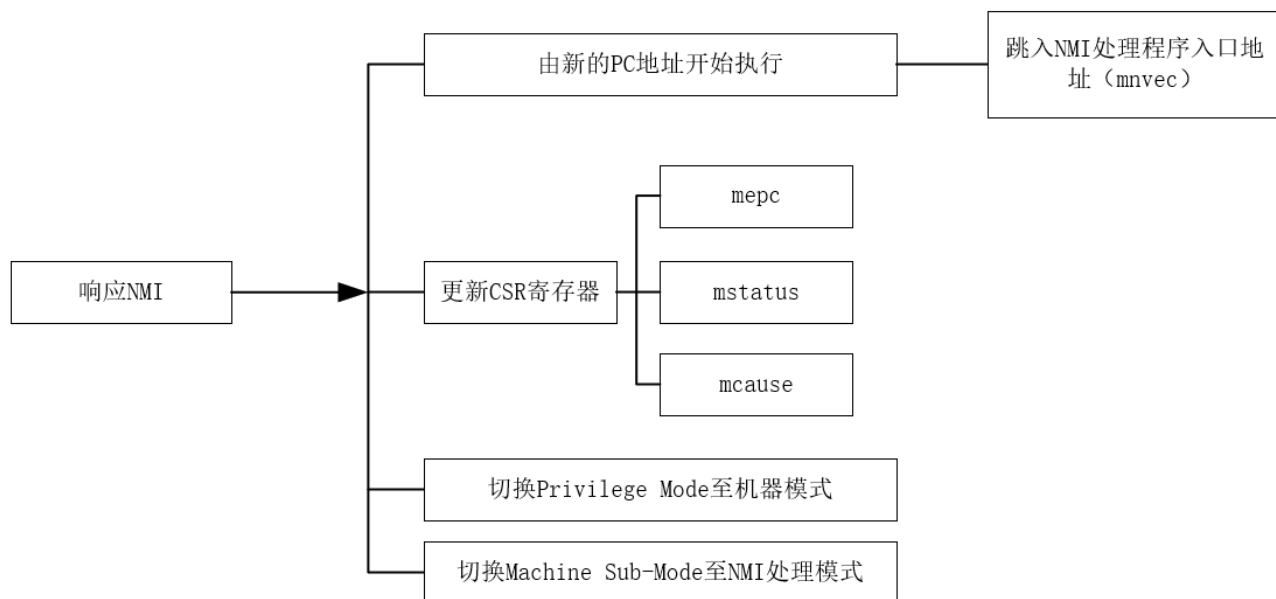


图 2-4 进入 NMI 处理模式图

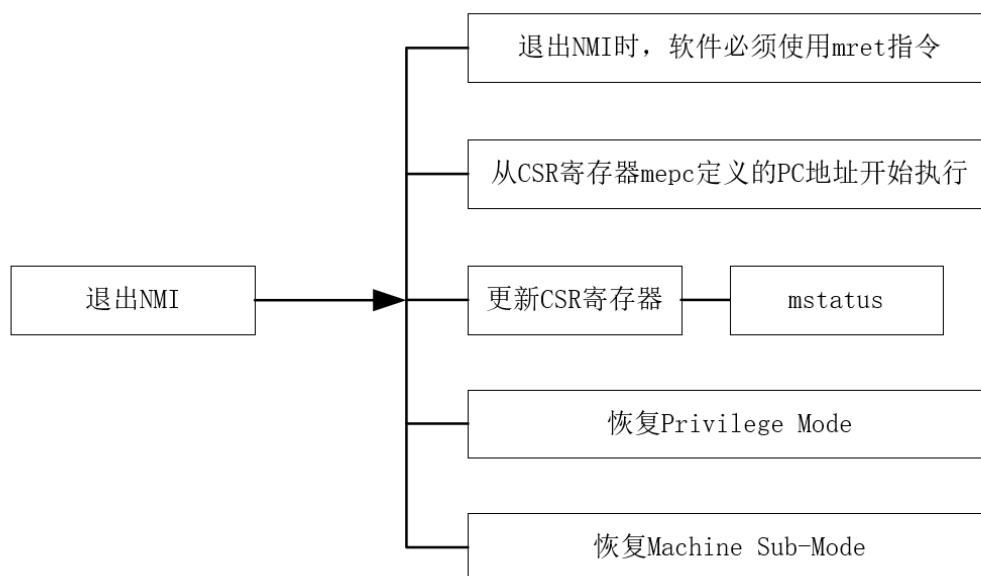


图 2-5 退出 NMI 处理模式图

2.6. N200 系列内核中断机制

中断（Interrupt）机制，即处理器核在顺序执行程序指令流的过程中突然被别请求打断而中止执行当前的程序，转而去处理别的事情，待其处理完了别的事情，然后重新回到之前程序中断的点继续执行之前的程序指令流。

图 2-6 和图 2-7 分别介绍归纳了进入和退出中断模式的相关操作，详情请参见《Nuclei_N200 系列指令架构手册》第 5 章了解其详情。

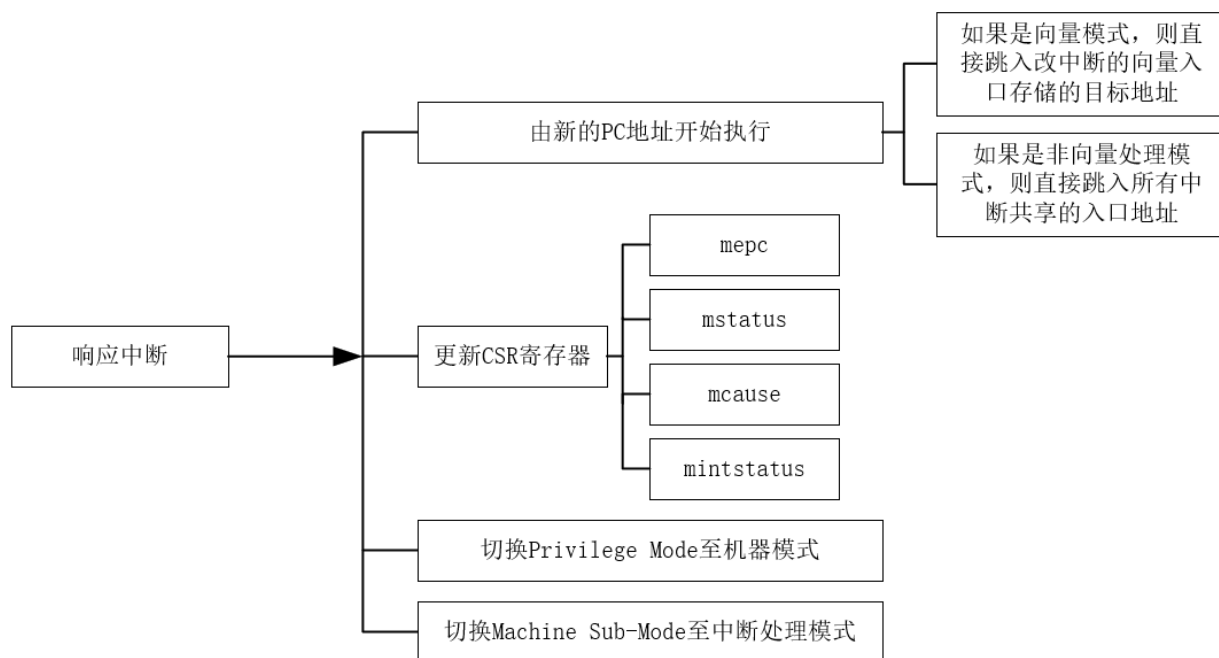


图 2-6 进入中断模式图

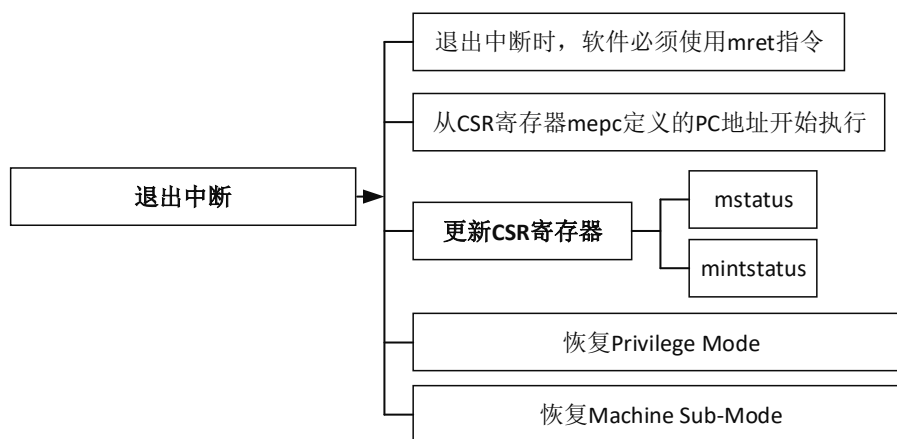


图 2-7 退出中断模式图

2.7. N200 系列内核 TIMER 和 ECLIC 介绍

2.7.1. TIMER 简介

计时器单元（Timer Unit, TIMER），在 N200 系列内核中主要用于产生计时器中断（Timer Interrupt）和软件中断（Software Interrupt）。详情请参见《Nuclei_N200 系列指令架构手册》第 6.1 节了解其详情。

2.7.2. ECLIC 简介

N200 系列内核支持在 RISC-V 标准 CLIC 基础上优化而来的“改进型内核中断控制器（Enhanced Core Local Interrupt Controller, ECLIC）”，用于管理所有的中断源。详情请参见《Nuclei_N200 系列指令架构手册》第 6.2 节了解其详情。

ECLIC 单元生成一根中断线，发送给处理器内核（作为中断目标），其关系结构如图 2-8 所示。

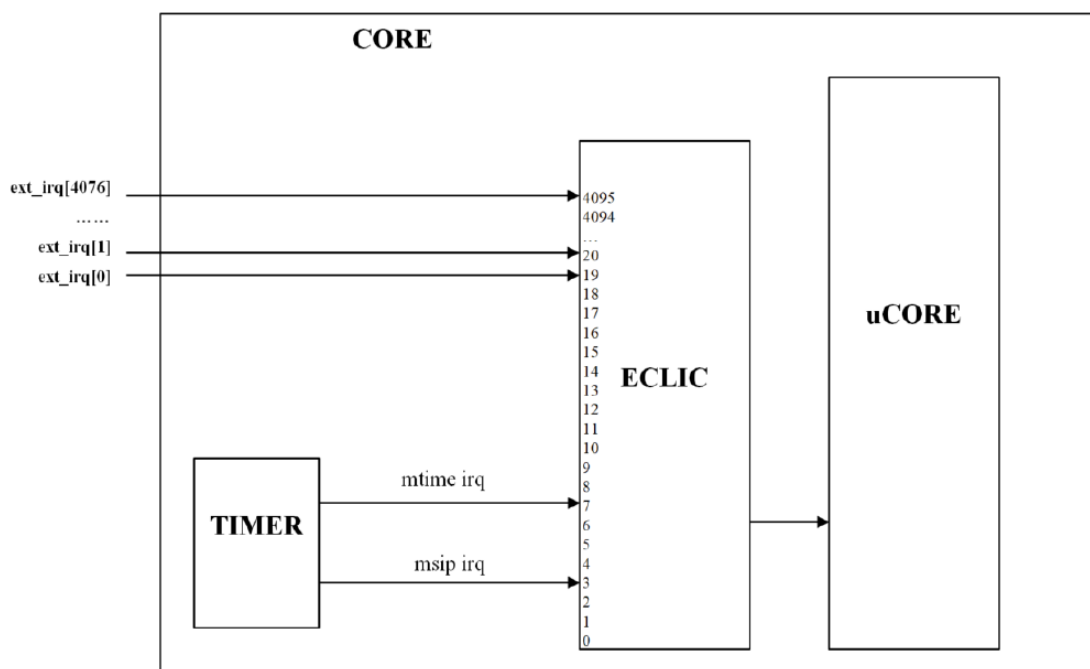


图 2-8 ECLIC 关系结构图

在下文讲解的 `Demo_eclic` 实例中，将详细讲解计时器中断和外部中断（通过用户按键实现）。

其中，计时器中断用于产生一个 **0.5s** 的周期性中断；按键 **1** 和按键 **2**，可以产生 **2** 个外部中断。（计时器中断将被设置为非向量模式，按键 **1** 中断将被设置为向量模式，而按键 **2** 中断将被设置为非向量模式。）

2.7.3. 中断屏蔽

N200 内核的中断可以被屏蔽掉，CSR 寄存器 `mstatus` 的 `MIE` 域控制中断的全局使能。

对于不同的中断类型而言，ECLIC 统一管理外部中断、软件中断以及计时器中断。ECLIC 为每个中断源分配各自的中断使能寄存器，用户可以通过配置 ECLIC 寄存器来管理各个中断源的屏蔽。

同时，用户可以通过配置 ECLIC 寄存器来管理各个中断源的屏蔽，达到使能或屏蔽某一个对应的中断。详情请参见《Nuclei_N200 系列指令架构手册》第 5.4 节了解其详情。

2.7.4. 中断级别、优先级与仲裁

当多个中断同时出现时，需要进行仲裁。对于 N200 系列内核处理器而言，ECLIC 统一管理所有的中断。ECLIC 为每个中断源分配了各自的中断级别和优先级寄存器，用户可以通过配置 ECLIC 寄存器来管理各个中断源的级别和优先级，当多个中断同时发生时，ECLIC 会仲裁出级别和优先级最高的中断，如图 2-9 中所示。详情请参见《Nuclei_N200 系列指令架构手册》第 6.2.9 节了解其详情。

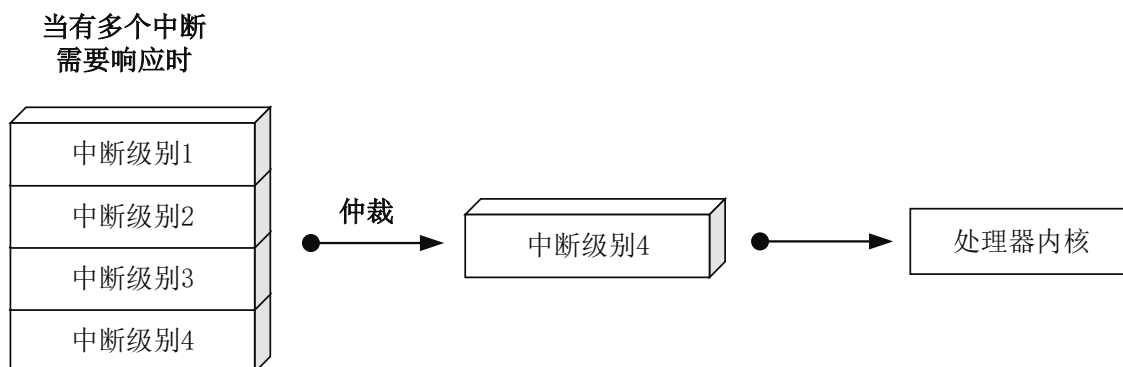


图 2-9 中断仲裁示意图

在下文讲解的 Demo_eclic 实例中，将计时器中断、按键 1 外部中断和按键 2 外部中断的中断源级别，分别设置为级别 1、级别 2 和级别 3。当这 3 个中断并发时，经过仲裁后，内核将优先响应优先级为 3 的中断，即按键 2 外部中断。

2.7.5. 中断服务程序

当处理器内核响应并进入中断后，立即根据中断源编号跳转至该中断对应的中断服务程序开始执行。譬如下文讲解的 Demo_eclic 实例中，函数 mtime_handler() 是计时器中断的服务程序，设置为非向量模式，函数 button_1_handler() 是按键 1 外部中断的服务程序，设置为向量模式，函数 button_2_handler() 是按键 2 外部中断的服务程序，设置为非向量模式。

2.7.6. 中断嵌套

处理器内核正在处理某个中断的过程中，可能有一个中断源级别更高的新中断请求到来，处理器可以中止当前的中断服务程序，转而开始响应新的中断，并执行其“中断服务程序”，如此便形成了中断嵌套（即前一个中断还没响应完，又开始响应新的中断），并且嵌套的层次可以有很多层。

在下文讲解的 Demo_eclic 实例中，演示了当有三个不同的中断源级别时，向量模式和非向量模式中断的嵌套流程。在特殊的情况下，最多会出现三层中断嵌套。详细原理可以参见

《Nuclei_N200 系列指令架构手册》第 5.11 节。其中按键 1 被设置为向量中断模式，其它两个中断被设置为非向量模式。现结合图 2-10 进行简要讲解：

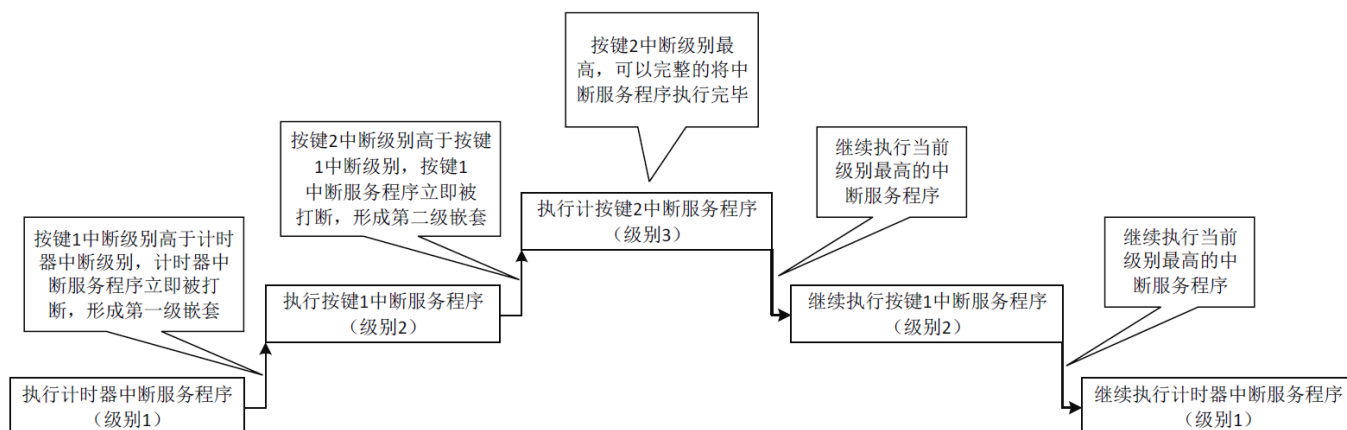


图 2-10 中断嵌套示意图

在评估板上实际运行 Demo_eclic 实例时，当出现上图所示的中断嵌套情况时，可以通过调试器的串口打印信息进行详细观察。

- 串口打印现象 1：计时器中断服务程序打印信息。如图 2-11 所示，此时计时器中断服务程序周期性的打印信息并产生延时。
- 串口打印现象 2：按键 1 中断服务程序（向量模式）打印信息。如图 2-12 所示，当用户按下按键 1 时，按键 1 对应的中断服务程序（向量模式）打印信息并产生延时。
- 串口打印现象 3：按键 2 中断服务程序打印信息。如图 2-13 所示，当用户按下按键 2 时，按键 2 对应的中断服务程序打印信息并产生延时。
- 串口打印现象 4：出现中断嵌套时的打印信息。如图 2-14 所示，当用户按下按键 1（向量模式）或按键 2 时，由于按键 1（向量模式）或按键 2 的中断级别高于计时器中断级别，计时器中断服务程序将被“打断”出现中断嵌套；如图 2-15 所示，当用户接连快速按下按键 1（向量模式）和按键 2 时，由于按键 2 的中断级别最高，按键 2 的中断服务程序将“打断”按键 1（向量模式）的中断服务程序出现中断嵌套。


```
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
```

计时器中断周期性发生。

图 2-11 计时器中断串口打印信息图

```
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Begin button1 handler----Vector mode
-----Waited 5 seconds.
-----End button1 handler
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
```

计时器中断周期性发生，
按键 1 中断发生一次。（并未出现嵌套）

图 2-12 按键 1 中断串口打印信息图

```
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
-----Begin button2 handler----NonVector mode
-----Waited 5 seconds.
-----End button2 handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
```

计时器中断周期性发生，按
键 2 中断发生一次。（并未出现嵌套）

图 2-13 按键 2 中断串口打印信息图

```
Thank you for supporting RISC-V, you will see the blink soon on the board!
Begin mtime handler----NonVector mode
-----Begin button1 handler----Vector mode
-----Waited 5 seconds.
-----End button1 handler
-----Waited 5 seconds.
End mtime handler
Begin mtime handler----NonVector mode
-----Waited 5 seconds.
End mtime handler
```

按键 1 中断“打断”正在执行的计时器中断服务程序，出现中断嵌套。

图 2-14 按键 1 “打断”计时器中断串口打印信息图

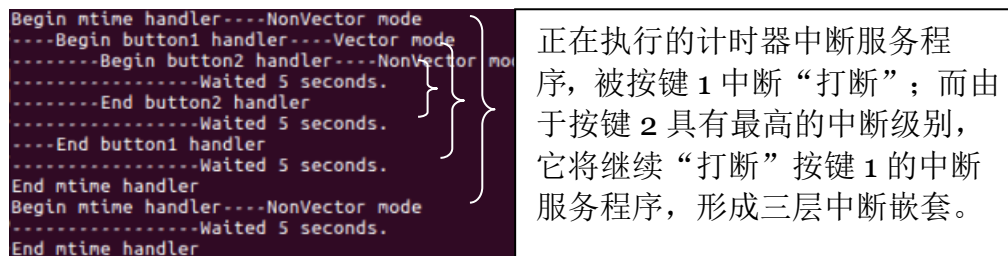


图 2-15 出现三层中断嵌套串口打印信息图

2.7.7. 中断咬尾

处理器内核正在处理某个中断的过程中，可能有新中断请求到来，但是“新中断的级别”低于或者等于“当前正在处理的中断级别”，因此，新中断不能够打断当前正在处理的中断（因此不会形成嵌套）。

当处理器完成当前中断之后，理论上需要恢复上下文，然后退出中断回到主应用程序，然后重新响应新的中断，响应新的中断又需要再次保存上下文。因此，存在着一次背靠背的“恢复上下文”和“保存上下文”操作，如果将此背靠背的“恢复上下文”和“保存上下文”省略掉，则称之为“中断咬尾”，如图 2-16 中所示，显而易见，中断咬尾可以加快多个中断的背靠背处理速度。

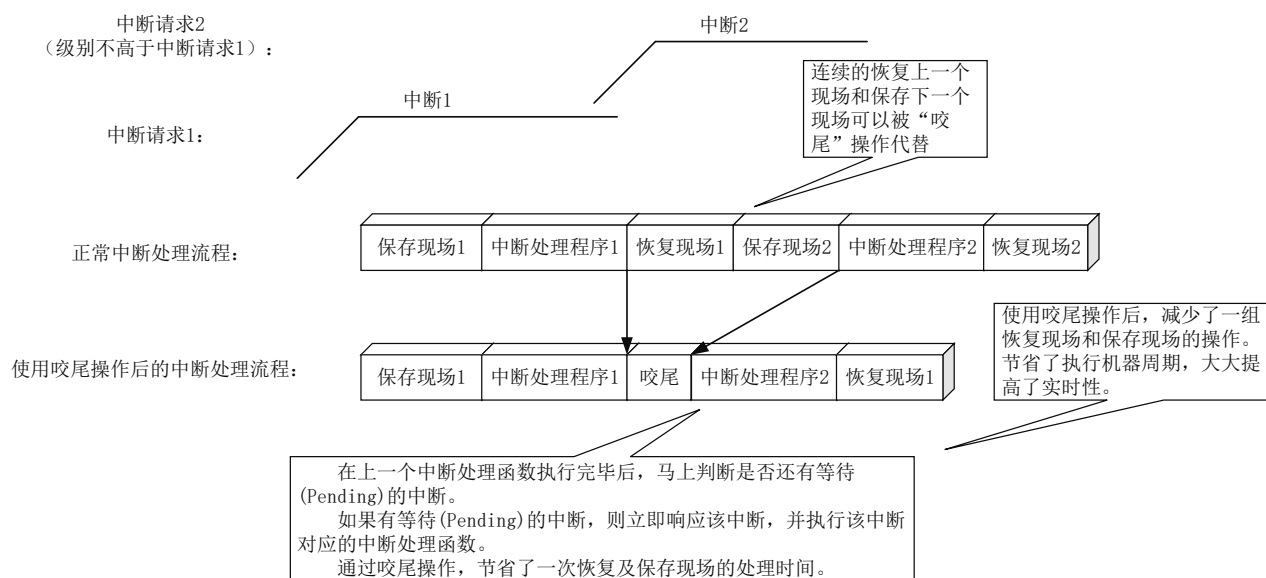


图 2-16 中断咬尾示意图

在 N200 系列内核中，只有非向量处理模式才支持中断咬尾，请参见《Nuclei_N200 系列指令架构手册》第 5.12 节和第 5.13 节了解其详情。

N200 系列内核支持“咬尾操作”机制，用户无需手动设置。现结合 N200-SDK/BSP 中的中断入口函数代码，简要讲解中断咬尾的实现原理。

irq_entry 函数即为使用汇编语言编写的在非向量模式下中断入口程序，该函数位于 bsp/nuclei-n200/n200/env/entry.S 中，主要用于上下文的保存和恢复，其代码如下：

```
// bsp/nuclei-n200/n200/env/entry.S 代码片段

//该宏用于保存 MSUBM、MEPC 和 MSUBM 寄存器进入堆栈
# Save the mepc and mstatus
.macro SAVE_EPC_STATUS
    csrr x5, CSR_MEPC
    STORE x5, 16*REGBYTES(sp)
    csrr x5, CSR_MSTATUS
    STORE x5, 17*REGBYTES(sp)
    csrr x5, CSR_MSUBM
    STORE x5, 18*REGBYTES(sp)
.endm

//该宏用于从堆栈中恢复 MSUBM、MEPC 和 MSUBM 寄存器
# Restore the mepc and mstatus
.macro RESTORE_EPC_STATUS
    LOAD x5, 16*REGBYTES(sp)
    csrw CSR_MEPC, x5
    LOAD x5, 17*REGBYTES(sp)
    csrw CSR_MSTATUS, x5
    LOAD x5, 18*REGBYTES(sp)
    csrw CSR_MSUBM, x5
.endm

.section      .text.irq
.align 2
.global irq_entry

.weak irq_entry //指定该标签为 weak 类型，标签为“弱 (weak)”属性。“弱 (weak)”属性是
// C/C++语法中定义的一种属性，一旦有具体的“非弱”性质同名函数存在，将
//会覆盖此函数。
irq_entry:    //定义标签名 irq_entry，该标签名作为函数入口

//更改堆栈指针，分配 19 个单字（32 位）的空间用于保存寄存器
    addi sp, sp, -19*REGBYTES

//进入中断处理函数之前必须先保存处理器的上下文
    SAVE_CONTEXT
//此处调用 SAVE_CONTEXT 保存 ABI 定义的“调用者应存储的寄存器 (Caller saved
//register)”进入堆栈

//特殊的 CSR 读操作，将寄存器 mcause 的数值直接作为操作数存到内存
    csrrwi x0, CSR_PUSHMCAUSE, 16
//特殊的 CSR 读操作，将寄存器 mepc 的数值直接作为操作数存到内存
    csrrwi x0, CSR_PUSHMEPC, 17
```

```
//特殊的 CSR 读操作,将寄存器 msubm 的数值直接作为操作数存到内存
csrrwi x0, CSR_PUSHMSUBM, 18

service_loop:
//特殊的 CSP 读/写操作, 通过 ECLIC 去查找当前等到的最高优先级的中断, 查询中断像量表, 进入中断向量表对应的入口地址。
csrrw ra, CSR_JALMNXTI, ra

    RESTORE_CONTEXT_EXCPT_X5

#关闭全局中断
DISABLE_MIE

//从堆栈中恢复 MEPC、MEPC 和 MSTATUS 寄存器
LOAD x5, 18*REGBYTES(sp)
csrw CSR_MSUBM, x5
LOAD x5, 17*REGBYTES(sp)
csrw CSR_MEPC, x5
LOAD x5, 16*REGBYTES(sp)
csrw CSR_MCAUSE, x5

    RESTORE_CONTEXT_ONLY_X5

//恢复寄存器之后, 更改堆栈指针, 回收 19 个单字 (32 位) 的空间
addi sp, sp, 19*REGBYTES
mret //使用 mret 指令从异常模式返回
```

详情请参见《Nuclei_N200 系列 SDK 使用说明》第 2.4.4 节了解其详情。

3. 背景知识——N200 系列内核配套 SoC

N200 系列内核配套的 MCU 级别 SoC，该 SoC 用于 N200 系列处理器内核的原型验证和用户评估。

3.1. N200 系列内核配套 SoC 框图

N200 系列内核配套 SoC 的框图如图 3-1 所示：

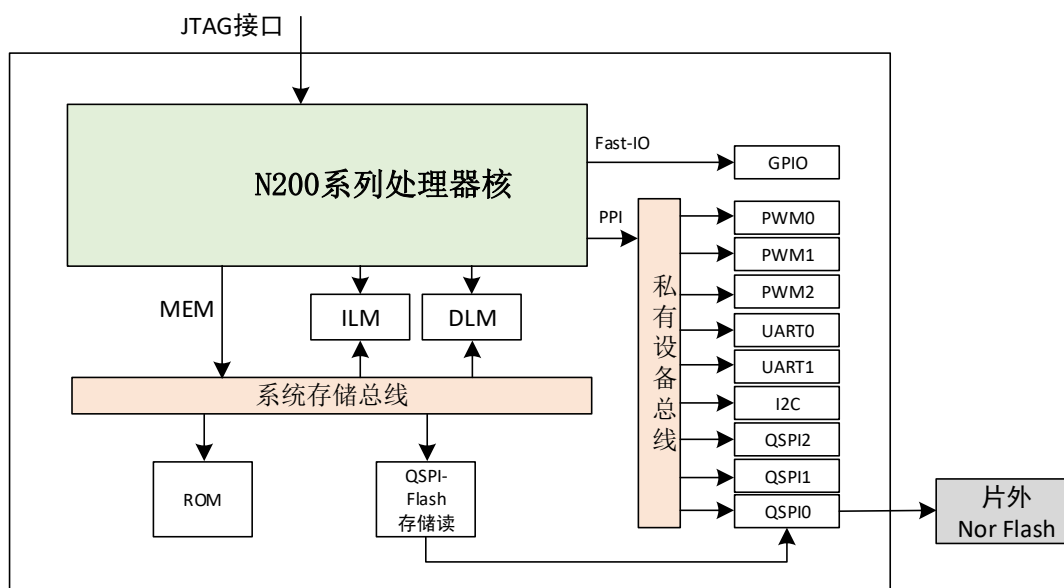


图 3-1 N200 系列内核配套 SoC 结构图

3.2. N200 系列内核配套 SoC 储存资源

如图 3-1 中所示，N200 系列内核配套 SoC 中的存储器资源分为片上存储资源和片外 Flash 存储资源。

有关 N200 系列内核配套 SoC 存储资源的详细介绍，请参见文档《Nuclei_N200 系列配套 SoC 介绍》或者中文书籍《RISC-V 架构与嵌入式开发快速入门》第 5 章。

3.3. N200 系列内核配套 SoC 外设介绍

有关 N200 系列内核配套 SoC 外设的详细介绍，请参见文档《Nuclei_N200 系列配套 SoC 介绍》或者中文书籍《RISC-V 架构与嵌入式开发快速入门》第 6 章。

4. 应用实例解析——Demo_eclic

4.1. Demo_eclic 简述

Demo_eclic 程序是一个完整的示例程序，相比 Dhrystone 和 CoreMark 这样纯粹的跑分程序，Demo_eclic 更加接近一个常见的嵌入式应用程序，它使用到了 SoC 系统中的外设，调用了中断服务程序等，其功能简述如下：

- 通过 `printf` 函数输出一串 “RISC-V” logo 的字符，`printf` 输出将会通过 UART 串口重定向至主机 PC 的屏幕上。
- 等待通过 `getc` 函数输入一个字符，然后将得到的字符通过 `printf` 输出到主机 PC 的屏幕上。
- 进入死循环不断地对 SoC 的 GPIO 13 的输出管脚进行翻转，如果使用示波器观测此 GPIO 输出管脚，可以看到其产生规律的输出方波。
- 设置 TIMER 计时器，使其先等待 5 秒钟之后开始触发计时器中断。在计时器中断中配置计时器的下一次触发时间是 0.5 秒以后，每次触发计时器中断都会在处理函数中对 GPIO 的输出管脚（对应三色灯的红灯）进行翻转。所以观察到的现象便是：刚开始等待 5 秒钟，之后开发板上红灯便开始以 1 秒钟的固定周期进行闪烁。
- 开发板上的两个用户按键，通过杜邦线连接到 SoC 的 GPIO 管脚，这两个 GPIO 管脚各自作为一个 ECLIC 的外部中断，在其中断服务程序中会将 GPIO 的输出管脚（对应三色灯的蓝灯和绿灯）进行设置，从而造成开发板上三色灯的颜色发生变化。

4.2. Demo_eclic 程序代码结构

用户可以通过 GitHub 链接：<https://github.com/nucleisys/n200-sdk>，获取完整程序代码。

在 n200-sdk 环境中，Demo_eclic 示例程序的相关代码结构如下所示。

```
n200-sdk                                     // 存放 n200-sdk 的目录
|-----software                             // 存放示例程序的源代码
|       |-----Demo_eclic                   // Demo_eclic 示例程序目录
|       |       |-----demo_eclic.c        // Demo_eclic 的示例 Demo 程序代码
|       |       |-----Makefile            // Makefile 脚本
```

Makefile 为主控制脚本，其代码片段如下：

```
//指明生成的 elf 文件名
TARGET = Demo_eclic

//指明 Demo_eclic 程序所需要的特别的 GCC 编译选项
CFLAGS += -O2

BSP_BASE = ../../bsp

//指明 Demo_eclic 程序所需要的 c 源文件
C_SRCS += Demo_eclic.c

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/env/common.mk
```


4.3. Demo_eclic 程序解析

4.3.1. 主程序流程图

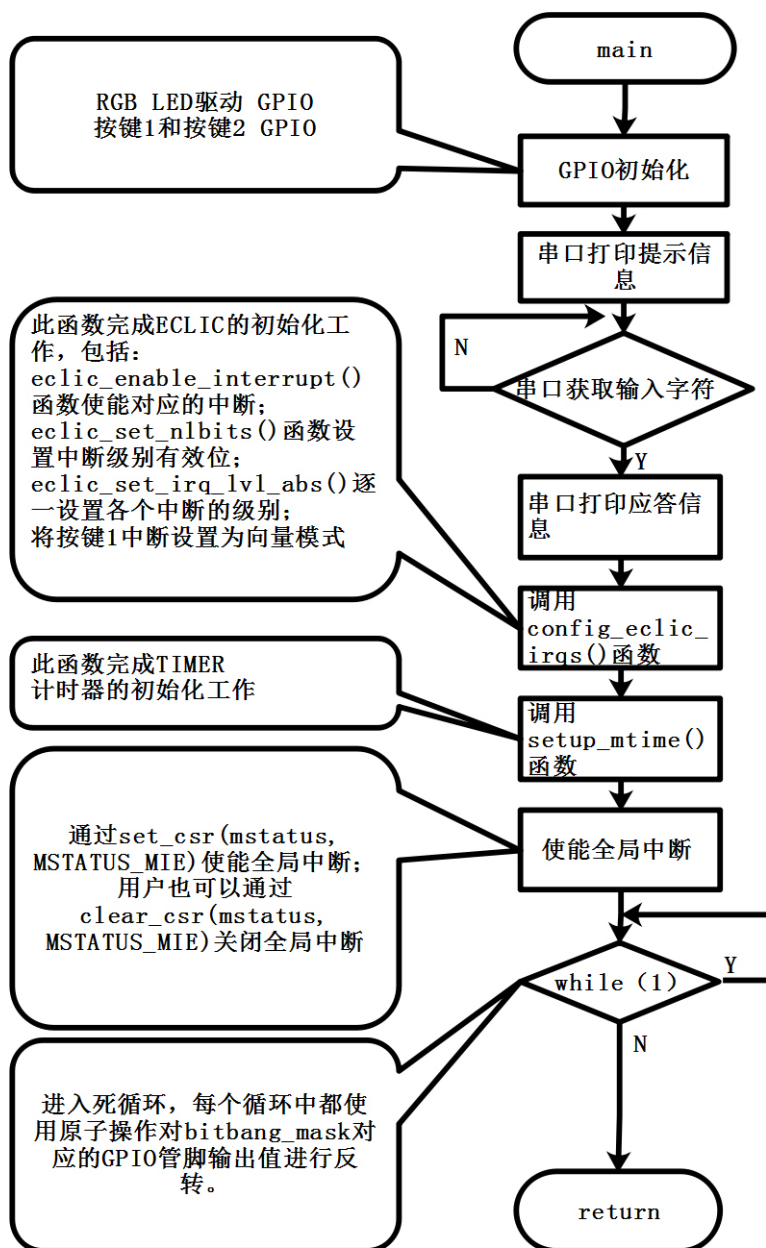


图 4-1 主程序流程图

4.3.2. 计时器中断处理函数流程图

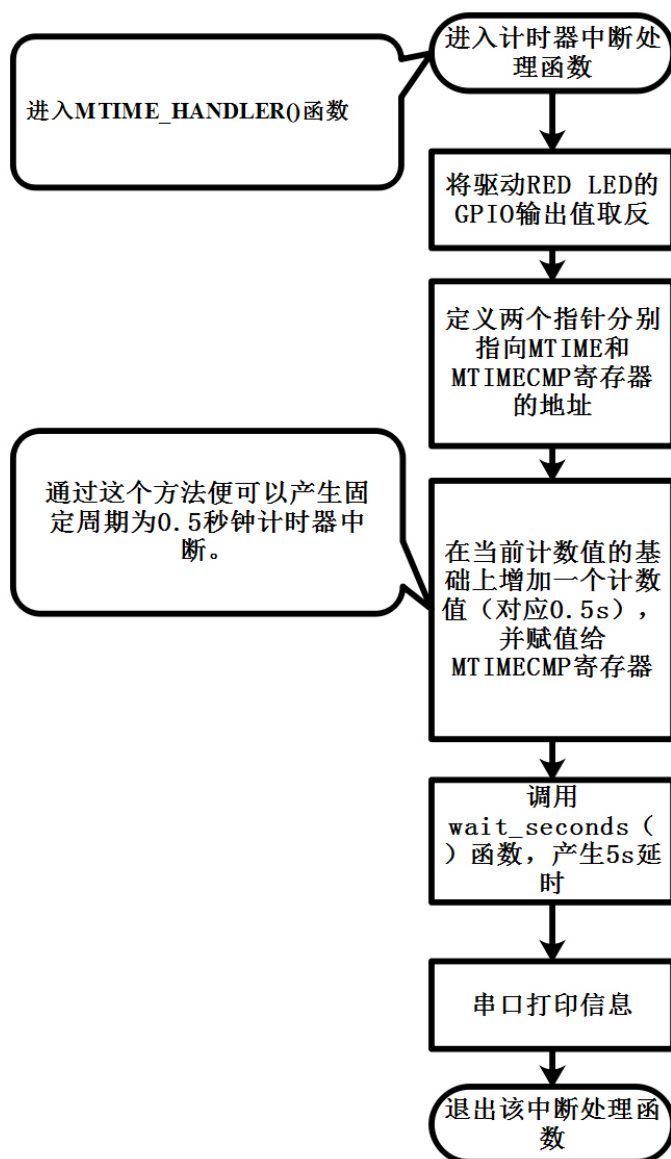


图 4-2 计时器中断处理函数流程图

4.3.3. 用户按键中断处理函数流程图

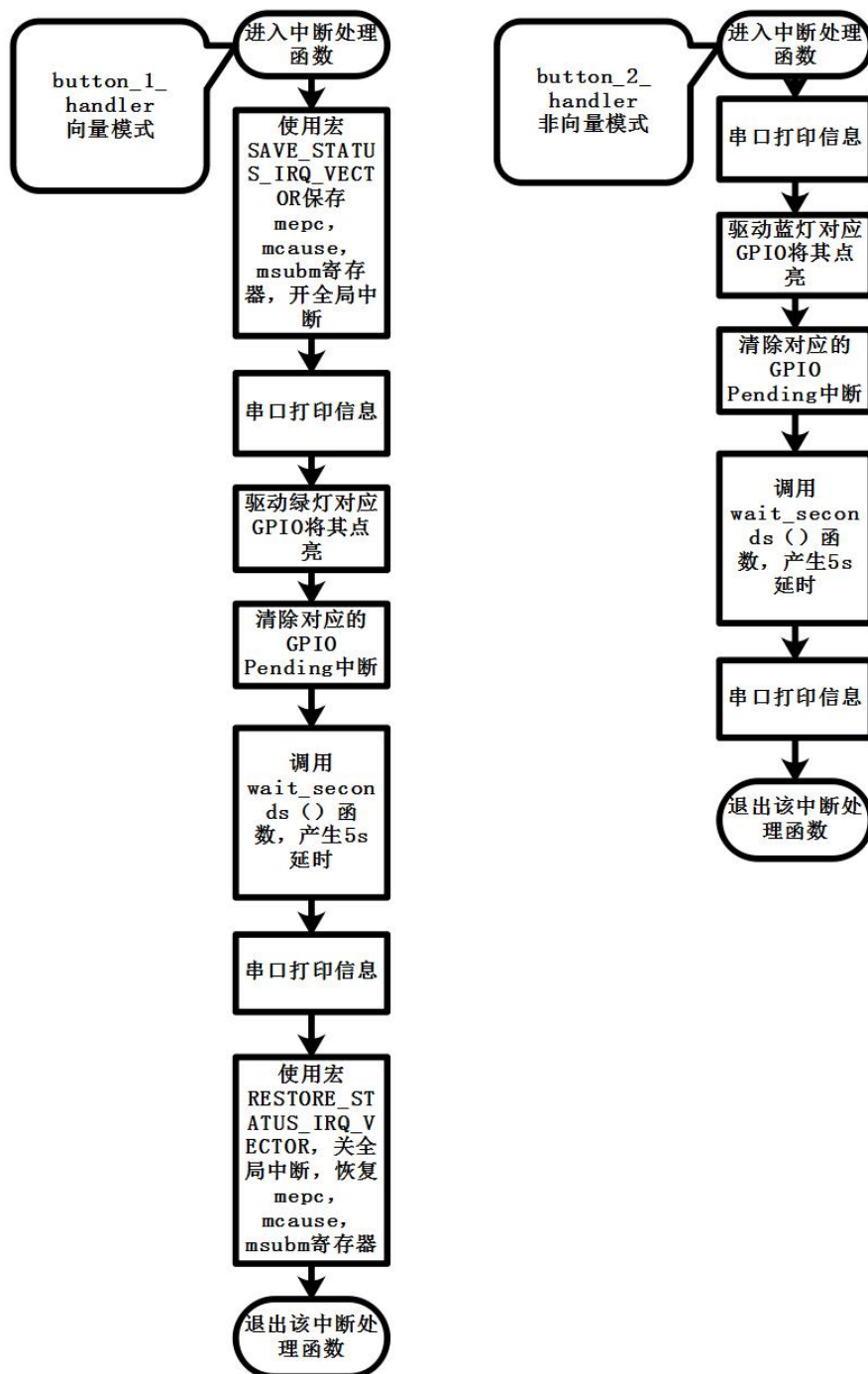


图 4-3 按键中断处理函数流程图

4.3.4. 快速移植中断应用

本节将结合 Demo_eclic 源码来详细介绍如何快速使用 N200 系列处理器内核中断。

■ 步骤一：

● 外部中断 GPIO 初始化

// n200-sdk/software/Demo_eclic/Demo_eclic.c 代码片段

```
#define BUTTON_1_GPIO_OFFSET 30
#define BUTTON_2_GPIO_OFFSET 31
//按键 1 对应管脚为 MCU_GPIO 30
//按键 2 对应管脚为 MCU_GPIO 31

#define ECLIC_INT_DEVICE_BUTTON_1 (SOC_ECLIC_INT_GPIO_BASE + BUTTON_1_GPIO_OFFSET)
#define ECLIC_INT_DEVICE_BUTTON_2 (SOC_ECLIC_INT_GPIO_BASE + BUTTON_2_GPIO_OFFSET)
//配置 ECLIC 中断源映射
```

//设置开发板上按键相关的 GPIO 寄存器

//通过“与”操作将 GPIO_OUTPUT_EN 寄存器某些位清 0，即将开发板按键对应的 GPIO 输出使能关闭

```
GPIO_REG(GPIO_OUTPUT_EN) &=
    ~(
        (0x1 << BUTTON_1_GPIO_OFFSET) |
        (0x1 << BUTTON_2_GPIO_OFFSET)
    );
```

//通过“与”操作将 GPIO_PULLUP_EN 寄存器某些位清 0，即将开发板按键对应的 GPIO 输入上拉关闭

```
GPIO_REG(GPIO_PULLUP_EN) &=
    ~(
        (0x1 << BUTTON_1_GPIO_OFFSET) |
        (0x1 << BUTTON_2_GPIO_OFFSET)
    );
```

//通过“或”操作将 GPIO_INPUT_EN 寄存器某些位设置为 1，即将开发板按键对应的 GPIO 输入使能关闭

```
GPIO_REG(GPIO_INPUT_EN) |=
    (
        (0x1 << BUTTON_1_GPIO_OFFSET) |
        (0x1 << BUTTON_2_GPIO_OFFSET)
    );
```

//通过“或”操作将 GPIO_RISE_IE 寄存器某些位设置为 1，即将开发板按键对应的 GPIO 管脚设置为上升沿触发的中断来源

```
GPIO_REG(GPIO_RISE_IE) |= (1 << BUTTON_1_GPIO_OFFSET);
GPIO_REG(GPIO_RISE_IE) |= (1 << BUTTON_2_GPIO_OFFSET);
```

● 定时器 TIMER 初始化

N200 内核 TIMER 单元中的 MTIME 寄存器和 MTIMECMP 寄存器充当计时器，这两个寄存

器均为存储器地址映射（Memory Address Mapped），可以通过配置这两个寄存器对计时器进行初始化。

在 Demo_eclic 函数中通过 setup_mtime 函数来对计时器进行配置，其源代码如下：

```
// n200-sdk/software/Demo_eclic/Demo_eclic.c 代码片段

void setup_mtime () {

    //定义两个 volatile 类型的指针，分别指向 MTIME 和 MTIMECMP 寄存器的地址
    volatile uint64_t * mtime      = (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIME);
    volatile uint64_t * mtimecmp   = (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIMECMP);

    //由于计时器默认一直在进行计数，所以需要通过读取 MTIME 寄存器得到当前的计数值
    uint64_t now = *mtime;

    //在目前的计数值基础上加上 2 秒钟的计数值，将其赋值给 MTIMECMP 寄存器，这意味着经过计时器的不断自增计数，2
    秒钟后 MTIME 的值就会大于 MTIMECMP 的值，从而产生计时器中断。
    uint64_t then = now + 2* TIMER_FREQ;
    *mtimecmp = then;

}
```

■ 步骤二：ECLIC 初始化

Demo_eclic 例程中，使用函数 config_eclic_irqs () 完成对计时器中断和两个按键中断的初始化。其代码片段如下：

```
// n200-sdk/software/Demo_eclic/Demo_eclic.c 代码片段

//设置 eclic_irq49 的中断处理函数为 BUTTON_1_HANDLER
#define BUTTON_1_HANDLER eclic_irq49_handler
//设置 eclic_irq50 的中断处理函数为 BUTTON_2_HANDLER
#define BUTTON_2_HANDLER eclic_irq50_handler
//设置 eclic_mtip (MTIME) 的中断处理函数为 MTIME_HANDLER
#define MTIME_HANDLER    eclic_mtip_handler

void config_eclic_irqs() {

    //通过配置 ECLIC 的寄存器使能“”计时器中断”和“”开发板两个按键的 GPIO 中断”。注意://ECLIC_enable_interrupt
    的函数原型定义在 bsp/nuclei-n200/n200/drivers/n200_func.c 中

    eclic_enable_interrupt (ECLIC_INT_MTIP);
    eclic_enable_interrupt (ECLIC_INT_DEVICE_BUTTON_1);
    eclic_enable_interrupt (ECLIC_INT_DEVICE_BUTTON_2);

    //设置 eclic 的中断级别有效位
    eclic_set_nbits(3);

}
```

```
//逐个设置各中断的级别
//通过配置 ECLIC 的寄存器设置 "" 计时器中断 "和" "开发板两个按键的 GPIO 中断" 的 ECLIC 中断级别, 由
//高至低依次为:

    // Button2 中断的级别为 3
    // Button1 中断的级别为 2
    // 计时器中断的级别为 1
    eclic_set_irq_lvl_abs(ECLIC_INT_MTIP, 1);
    eclic_set_irq_lvl_abs(ECLIC_INT_DEVICE_BUTTON_1, 2);
    eclic_set_irq_lvl_abs(ECLIC_INT_DEVICE_BUTTON_2, 3);

//button1 中断设置为向量模式, 其它两个中断使用非向量模式
    eclic_set_vmode(ECLIC_INT_DEVICE_BUTTON_1);
}
```

■ 步骤三：编写中断处理函数

● 计时器中断处理函数

由于 Demo_eclic 会使用到计时器中断, 所以定义了 MTIME_HANDLER()函数作为其中断处理函数, 其代码如下:

```
// n200-sdk/software/Demo_eclic/Demo_eclic.c 代码片段

void MTIME_HANDLER() {

    //将对应开发板上“三色灯中红灯”GPIO 管脚的输出值取反。由于每隔 0.5 秒进入中断处理函数后会对红灯取反(亮或者灭),
    //所以在开发板上观察到的就是每隔 1 秒钟, 红灯闪烁一次。
    GPIO_REG(GPIO_OUTPUT_VAL) ^= (0x1 << RED_LED_GPIO_OFFSET);

    //定义两个 volatile 类型的指针, 分别指向 MTIME 和 MTIMECMP 寄存器的地址
    volatile uint64_t * mtime      = (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIME);
    volatile uint64_t * mtimecmp   = (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIMECMP);

    //在目前的计数值基础上加上 0.5 秒钟的计数值, 将其赋值给 MTIMECMP 寄存器, 这意味着经过计时器的不断自增计数,
    //0.5 秒钟后 MTIME 的值就会大于 MTIMECMP 的值, 从而再次产生计时器中断。通过这个方法便可以产生固定周期为 0.5 秒钟
    //计时器中断。
    uint64_t now = *mtime;
    uint64_t then = now + 0.5 * TIMER_FREQ;
    *mtimecmp = then;
}
```

● 外部中断处理函数

在蜂鸟 FPGA 开发板上, 例程中使用到的两个用户按键, 需要使用杜邦线将 GPIO_BTN1 和 GPIO_BTN2 连接到 GPIO30 和 GPIO31 上, 这两个 GPIO 管脚各自作为一个 ECLIC 的外部中断。Demo_eclic 使用这两个外部中断, 所以定义了 button_1_handler 和 button_2_handler 分别作为

它们的中断处理函数，其代码如下：

//向量处理模式时，由于在跳入中断服务程序之前，处理器并没有进行上下文的保存，因此，理论上中断服务程序函数本身不能够进行子函数的调用（即必须是 Leaf Function）。

//此处 BUTTON_1_HANDLE 明显会调用其他的子函数，如果不加处理则会造成功能的错误。为了规避这种情形，当中断被设置为“向量模式”时，用户必须使用特殊的 `__attribute__((interrupt))` 来修饰该中断服务程序函数，那么编译器会自动的进行判断，当编译器发现该函数调用了其他子函数时，便会自动的插入一段代码进行上下文的保存。注意：这种情况下虽然保证了功能的正确性，但是由于保存上下文所造成的开销，又会事实上增大中断响应延迟（与非向量模式相当）并且造成代码尺寸（Code Size）的膨胀。

```
void __attribute__((interrupt)) BUTTON_1_HANDLER(void) {
```

//向量处理模式时，由于在跳入中断服务程序之前，处理器并没有进行任何特殊的处理，且由于处理器内核在响应中断后，mstatus 寄存器中的 MIE 域将会被硬件自动更新成为 0（意味着中断被全局关闭，从而无法响应新的中断）。因此向量处理模式默认是不支持中断嵌套的，为了达到向量处理模式且又能够中断嵌套的效果，需要在中断服务程序函数的开头处添加特殊的入栈操作，之后再重新打开中断的全局使能。

//保存 mepc, mcause, msubm, 并使能全局中断

```
SAVE_STATUS_IRQ_VECTOR;
```

```
printf ("%s", "----Begin button1 handler\n");
```

// 点亮绿灯

```
GPIO_REG(GPIO_OUTPUT_VAL) |= (1 << GREEN_LED_GPIO_OFFSET);
```

```
GPIO_REG(GPIO_RISE_IP) = (0x1 << BUTTON_1_GPIO_OFFSET);
```

// 等待 5s

```
wait_seconds(5);
```

```
printf ("%s", "----End button1 handler\n");
```

//在中断服务程序的结尾处同样需要添加对应的恢复上下文出栈操作。并且在 CSR 寄存器 mepc、mcause、msubm 出堆栈之前，需要将全局中断使能再次关闭，以保证 mepc、mcause、msubm 恢复操作的原子性（不被新的中断所打断）。

//关全局中断，恢复 mepc, mcause, msubm

```
RESTORE_STATUS_IRQ_VECTOR;
```

```
};
```

```
void BUTTON_2_HANDLER(void) {
```

```
printf ("%s", "-----Begin button2 handler\n");
```

// 点亮蓝灯

```
GPIO_REG(GPIO_OUTPUT_VAL) |= (1 << BLUE_LED_GPIO_OFFSET);
```

```
GPIO_REG(GPIO_RISE_IP) = (0x1 << BUTTON_2_GPIO_OFFSET);
```

// 等待 5s

```
wait_seconds(5);
```

```
printf ("%s", "-----End button2 handler\n");
```

```
};
```

//使用一个宏来保存 mcause, mepc, msubm 寄存器, 打开全局中断

```
#define SAVE_STATUS_IRQ_VECTOR \
uint32_t mcause = read_csr(mcause);\
uint32_t mepc = read_csr(mepc);\
uint32_t msubm = read_csr(0x7c4);\
set_csr(mstatus, MSTATUS_MIE);
```

//使用一个宏来关闭全局中断, 恢复 mcause, mepc, msubm 寄存器

```
#define RESTORE_STATUS_IRQ_VECTOR \
clear_csr(mstatus, MSTATUS_MIE);\
write_csr(0x7c4, msubm);\
write_csr(mepc, mepc);\
write_csr(mcause, mcause);
```

有关 N200 系列内核中断机制以及中断控制器的详细介绍, 请参见文档《Nuclei_N200 系列指令架构手册》第 5 章和第 6 章内容。

5. Demo_eclic 软硬件快速启动

本章将结合芯来科技公司推出的蜂鸟 FPGA 开发板和配套的蜂鸟 JTAG 下载器,进行详细讲解。该评估套件资料的 github 链接为:

https://github.com/SI-RISCV/e200_opensource/tree/master/boards, 用户可以根据实际情况选择性购买。

用户购买到的评估板,在出厂时已经预烧写了 N200 内核及其配套 SoC。用户可以把板载的 FPGA 芯片,“当作”一颗集成了 N200 系列内核的微处理器来进行开发和调试。

关于 FPGA 开发板程序的烧写方法,请用户参见文档《Nuclei_N200 系列配套 FPGA 实现》。

下文将介绍在 Linux 环境下,使用 n200-sdk 来进行软硬件快速启动。用户可以在 GitHub 上下载 n200-sdk,链接为: <https://github.com/nucleisys/n200-sdk>。

如客户需要在 Windows 环境下进行开发工作,请参见文档《Nuclei_N200 系列 IDE 使用说明》。

5.1. 第一步: 硬件线缆连接

- 连接蜂鸟 FPGA 开发板 5V 电源线;
- 将蜂鸟下载器通过排线连接至开发板;
- 将蜂鸟下载器连接至调试主机。
- 将开发板电源开关拨至“ON”档,接通电源。

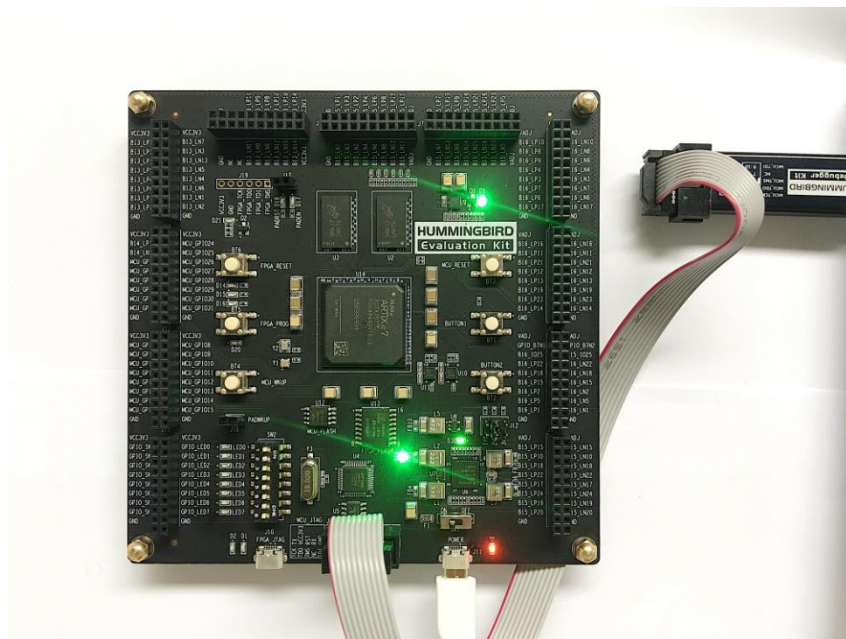


图 5-1 评估板及下载器线缆连接图

5.2. 第二步：设置下载器在 Linux 系统中的 USB 权限

如果使用 Linux 操作系统，需要按照如下步骤保证正确的设置蜂鸟下载器的 USB 权限。详细内容请参见文档《Nuclei_N200 系列 SDK 使用说明》第 2 章。

5.3. 第三步：连接外部中断引脚至拨码开关引脚

Demo_eclic 示例可以支持 2 个外部中断：按键 1 中断和按键 2 中断。在基于蜂鸟 FPGA 开发板运行该例程时，需要将 2 个外部中断引脚使用杜邦线，连接至按键引脚。

Demo_eclic 示例程序中，使用如下代码，将按键 1 和按键 2 外部中断，分别映射到 MCU_GPIO30 和 MCU_GPIO31。

```
#define BUTTON_1_GPIO_OFFSET 30
#define BUTTON_2_GPIO_OFFSET 31
```

所以如图 5-2 所示，用户需要使用两根杜邦线，将 MCU_GPIO30 连接至 GPIO_BTN1，MCU_GPIO31 连接至 GPIO_BTN2。

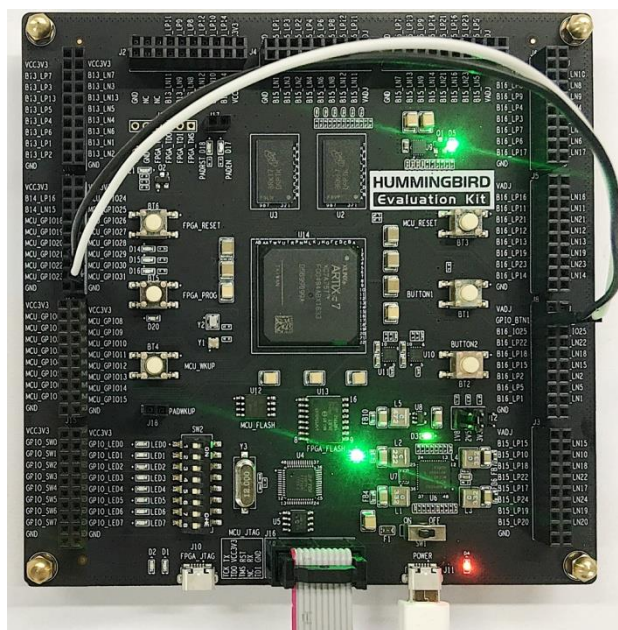


图 5-2 连接按键至 MCU_GPIO 图

5.4. 第四步：将 Demo_eclic 程序下载至评估板并运行

Demo_eclic 示例可运行于 N200 内核及配套 SoC 评估板平台中，使用基于 Linux 的 SDK 来进行演示。关于 Linux 环境下 SDK 的详细介绍，请用户参见《Nuclei_N200 系列 SDK 使用说明》。

具体操作方法按照如下步骤进行：

// 注意：确保在 n200-sdk 中正确的安装了 RISC-V GCC 工具链，请参见《Nuclei_N200 系列 SDK 使用说明》第 2.4.1 解其详情。

// 步骤一：参照《Nuclei_N200 系列 SDK 使用说明》第 2.4 节中描述的方法，编译 Demo_eclic 示例程序，使用如下命令：

```
make dasm PROGRAM=demo_eclic BOARD=nuclei-n200 CORE=n201 ODCFG=hbird
                                     DOWNLOAD=flash
```

//注意：此处指定 DOWNLOAD=flash 选项，则采用“将程序从 Flash 上载至 ILM 进行执行的方式”进行编译，请参见《Nuclei_N200 系列 SDK 使用说明》第 2.4.5 节了解更多详情。

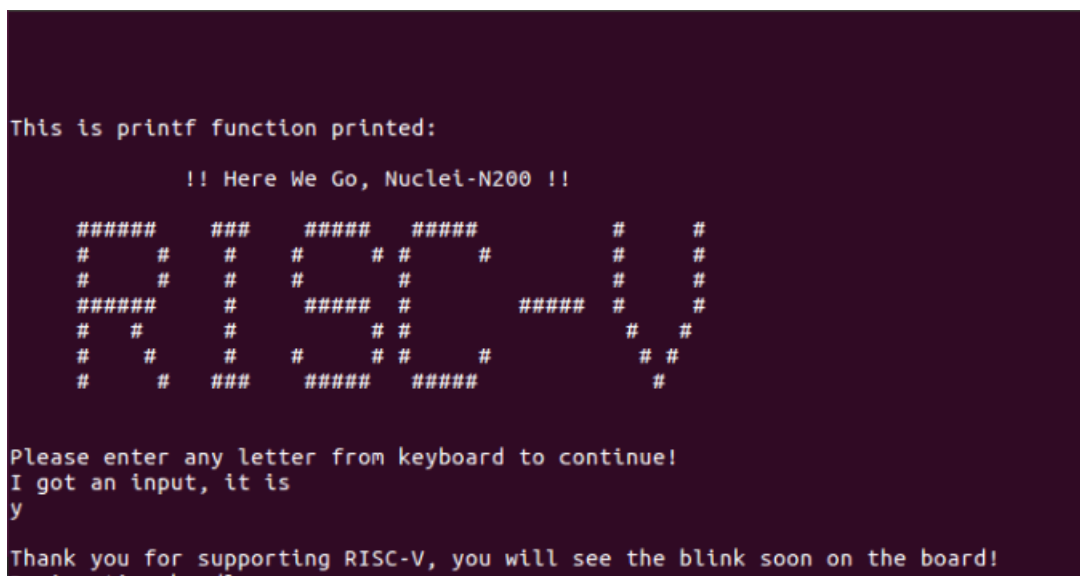
// 步骤二：参照《Nuclei_N200 系列 SDK 使用说明》第 2.5 节中描述的方法，将编译好的 Demo_eclic 程序下载至 FPGA 原型开发板中，使用如下命令：

```
make upload PROGRAM=demo_eclic BOARD=nuclei-n200 CORE=n201 ODCFG=hbird DOWNLOAD=flash
```

// 步骤三：参照《Nuclei_N200 系列 SDK 使用说明》第 2.6 节中描述的方法，在 FPGA 原型开发板上运行 Demo_eclic 程序：

// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考《Nuclei_N200 系列 SDK 使用说明》第 2.6 节中

```
// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。
//
// 由于步骤二已经将程序烧写进 FPGA 评估板的 Flash 之中，因此每次按 MCU 开发板的
// RESET 按键，则处理器复位开始执行 Demo_eclic 程序，并将 RISC-V 字符串打印至主
// 机 PC 的串口显示终端上，如图 5-3 所示，然后用户可以输入任意字符（譬如字母 y），
// 程序继续运行，开发板上将会以固定频率进行闪灯。
```



```
This is printf function printed:

      !! Here We Go, Nuclei-N200 !!

#####  ###  #####  #####  #  #
#  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #
#####  #  #####  #  #####  #  #
#  #  #  #  #  #  #  #  #  #
#  #  #  #  #  #  #  #  #  #
#  #  ###  #####  #####  #

Please enter any letter from keyboard to continue!
I got an input, it is
y

Thank you for supporting RISC-V, you will see the blink soon on the board!
```

图 5-3 运行 Demo_eclic 示例后于主机串口终端上显示信息

6. 快速应用——移植实时操作系统

6.1. 实时操作系统简述

实时操作系统（RTOS）是指当外界事件或者数据产生时，能够接受并以足够快的速度予以处理，处理的结果又能在规定的时间内来控制生产过程或对处理系统能够做出快速响应，调度一切可利用的资源完成实时任务，并控制所有实时任务协调一致运行的操作系统。主要特点是提供及时响应和可靠性。

在服务器、个人电脑、手机上运行的操作系统，譬如 Windows 和 Linux，强调在一处理器上能运行更多任务。此类操作系统的代码均具有一定规模，并且不一定能保证实时性。而对于处理器硬件资源有限，对实时性又有特殊要求的嵌入式应用领域，就需要一种代码规模适中，实时性好的操作系统。

实时性可以分为硬实时和软实时。硬实时的功能是必须在给定时间内完成操作，如果不能完成将可能导致严重后果。比如汽车安全气囊触发机制就是一个很好的硬实时的例子，在撞击后安全气囊必须在给定时间内弹出，如果响应时间超出给定时间，可能使驾驶员受到严重伤害。

对于软实时，一个典型的实例是 IPTV 数字电视机顶盒，需要实时的解码视频流，如果丢失了一个或几个视频帧，视频品质也不会相差多少。软实时系统从统计角度来说，一个任务有确定的执行时间，事件在截止时间到来之前也能得到处理，即使违反截止时间也不会带来致命的错误。

6.2. 常用实时操作系统概述

常用的实时操作系统（RTOS）有以下几种：FreeRTOS、VxWorks、uc/os-II、uclinuxeCos、RT-Thread 和 SylixOS 等。下面分别对这几种 RTOS 进行介绍说明。

■ SylixOS:

翼辉 SylixOS 实时操作系统是一款功能全面、稳定可靠、易于开发的国产实时系统平台。其解决方案覆盖网络设备、国防安全、工业自动化、轨道交通、电力、医疗、航空航天等诸多领域。SylixOS 是国内唯一一款支持 SMP 的大型实时操作系统。翼辉开发嵌入式操作系统 SylixOS 始于 2006 年，至今在军工领域已有众多项目或产品基于 SylixOS 进行开发，例如雷达、弹载飞控系统、星载任务计算机、机载火控系统、计重收费与超限检测仪、火灾报警系统、特种车辆与船用发动机状态显示器、潜艇蓄电池监控系统、轮式装甲

车实时监控系统等，其中大部分产品都要求 7*24 小时不间断运行，当前很多 SylixOS 系统节点已不间断运行超过 5 万小时（6 年）。

■ RT-Thread:

RT-Thread 是一款主要由中国开源社区主导开发的开源实时操作系统（许可证 GPLv2）。实时线程操作系统不仅是一个单一的实时操作系统内核，它也是一个完整的应用系统，包含了实时、嵌入式系统相关的各个组件：TCP/IP 协议栈、文件系统、libc 接口、图形用户界面等。RT-Thread 拥有良好的软件生态，支持市面上所有主流的编译工具如 GCC、Keil、IAR 等，工具链完善、友好，支持各类标准接口，如 POSIX、CMSIS、C++ 应用环境、Javascript 执行环境等，方便开发者移植各类应用程序。商用支持所有主流 MCU 架构，如 ARM Cortex-M/R/A、MIPS、X86、Xtensa、Andes、C-Sky、RISC-V，几乎支持市场上所有主流的 MCU 和 Wi-Fi 芯片。

■ FreeRTOS:

有关 FreeRTOS 详细介绍，请查看第 6.3 节。

■ VxWorks:

由美国 WindRiver 公司于 1983 年推出的一款实时操作系统。由于其良好的持续发展能力，高性能内核以及友好的开发环境，因此在嵌入式系统领域占有一席之地。VxWorks 由 400 多个相对独立、短小精悍的目标模块组成，用户可根据需要进行配置和裁剪，在通信、军事、航天、航空等领域应用广泛。

■ uc/os-II:

前身是 uc/os，最早由 1992 年美国嵌入式专家 Jean J.Labrosse 在《嵌入式系统编程》杂志上发表，其主要特点有公开源代码，代码结构清晰明了，注释详尽，组织有条理，可移植性好，可裁剪，可固化。

■ Uclinux:

是由 Lineo 公司主推的开放源代码的操作系统，主要针对目标处理器没有存储管理单元的嵌入式系统而设计的。Uclinux 从 Linux2.0/2.4 内核派生而来，拥有 Linux 的绝大部分特性，通常用于内存很少的嵌入式操作系统。其主要特点有体积小、稳定、良好的移植性、优秀的网络功能等。

■ eCos:

含义为嵌入式可配置操作系统，主要用于消费电子、电信、车载设备、手持设备等低成本和便携式应用。其最显著的特点为可配置性，可以在源码级别实现对系统的配置和裁剪，还可安装第三方组件扩展系统功能。

6.3. FreeRTOS 简介

由于 RTOS 需要占用一定系统资源，只有少数 RTOS 支持在小内存的 MCU 上运行，FreeRTOS 是一款迷你型实时操作系统内核，功能包括：任务管理、时间管理、信号量、消息队列、内存管理等功能，可基本满足较小系统的需要。相对于 VxWorks、uc/os-II 等商业操作系统，FreeRTOS 完全免费，具有源码公开、可移植、可裁剪、任务调度灵活等特点，可以方便地移植到各种 MCU 上运行，其突出的特性如下。

- 免费开源。完全可以放心作为商业用途。
- 文档资源齐全。在 FreeRTOS 官网上能下载到内核文件及详细的介绍资料。
- 安全性高。SafeRTOS 基于 FreeRTOS 而来，经过安全认证的 RTOS，近年来在欧美较为流行，支持抢占式和合作式任务切换模式，代码精简，核心由 3 个 C 文件组成，可支持 65536 个任务。因此其开源免费版本 FreeRTOS 在安全性方面也应该拥有一定保障。
- 市场使用率高。从 2011 年开始，FreeRTOS 市场使用率持续高速增长，根据 Eetimes 杂志市场报告显示，FreeRTOS 使用率名列前茅，如图 6-1 所示，2017 年 FreeRTOS 市场占有率为 20%，排名第二。
- 内核文件简单。内核相关文件仅由 3 个 C 文件组成，全部围绕任务调度展开，功能专一，便于理解与学习。

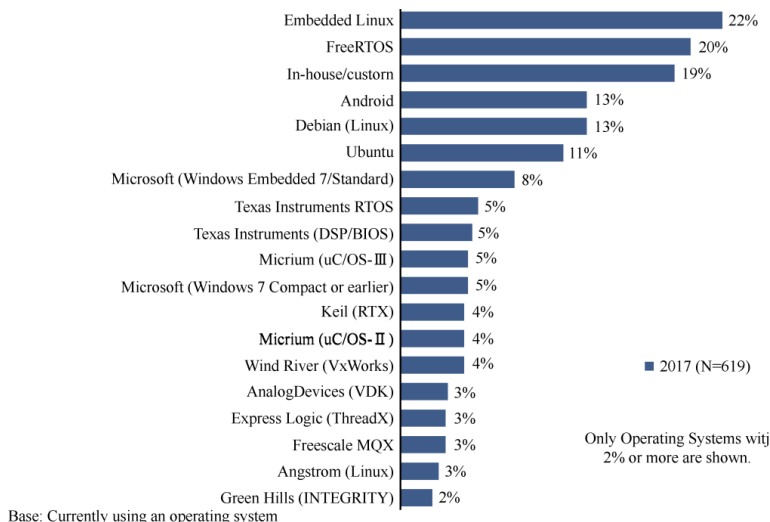


图 6-1 2017 年各种操作系统的使用数量统计图

6.4. Nuclei N200 移植 FreeRTOS

本节介绍如何在 N200-SDK 环境下移植一个简单的 FreeRTOS 示例。用户可以以此为基础进行丰富和完善，开发更多的复杂示例。

有关 n200-sdk 环境的快速上手请参见《N200 内核快速上手介绍》，有关 n200-sdk 系统性的详细介绍请参见中文书籍《RISC-V 架构与嵌入式开发入门指南》的第 11 章。

6.4.1. N200-SDK 中 FreeRTOS 程序代码结构

在 n200-sdk 环境中，FreeRTOS 示例程序的相关代码结构如下所示。

```
n200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----FreeRTOSv9.0.0 // FreeRTOS 示例程序目录
|----Source        //FreeRTOS 内核源代码
|----Demo          //FreeRTOS 的示例 Demo 程序代码
|----Makefile      //Makefile 脚本
```

Makefile 为主控制脚本，其代码片段如下：

```
//指明生成的 elf 文件名
TARGET = FreeRTOSv9.0.0
//指明 FreeRTOS 程序所需要的特别的 GCC 编译选项，默认选择 Code Size 优化 (-Os)
CFLAGS += -Os
```



```
BSP_BASE = ../../bsp

//指明 FreeRTOS 程序所需要的 c 源文件
C_SRCS += Source/croutine.c
C_SRCS += Source/list.c
C_SRCS += Source/queue.c
C_SRCS += Source/tasks.c
C_SRCS += Source/timers.c
C_SRCS += Source/event_groups.c
C_SRCS += Source/portable/MemMang/heap_4.c
C_SRCS += Source/portable/GCC/N200/port.c

C_SRCS += Demo/main.c

INCLUDES += -ISource/include
INCLUDES += -IDemo
INCLUDES += -ISource/portable/GCC/N200

ASM_SRCS += Source/portable/GCC/N200/portasm.S

//调用板级支持包 (bsp) 目录下的 common.mk
include $(BSP_BASE)/$(BOARD)/env/common.mk
```

6.4.2. FreeRTOS 原理和移植介绍

由于 RTOS 需要占用一定系统资源,只有少数 RTOS 支持在小内存的 MCU 上运行,FreeRTOS 是一款迷你型实时操作系统内核,功能包括:任务管理、时间管理、信号量、消息队列、内存管理等功能,可基本满足较小系统的需要。相对于 VxWorks、uc/os-II 等商业操作系统,FreeRTOS 完全免费,具有源码公开、可移植、可裁剪、任务调度灵活等特点、可以方便地移植到各种 MCU 上运行。

6.4.3. RTOS 操作系统的基本原理

传统裸机程序是一个大 while 循环,将所有事情看作一个任务,顺序执行代码,遇到中断发生则响应中断(可能发生中断嵌套),响应完中断后会继续之前被中断的任务,其过程如图 6-2 所示。

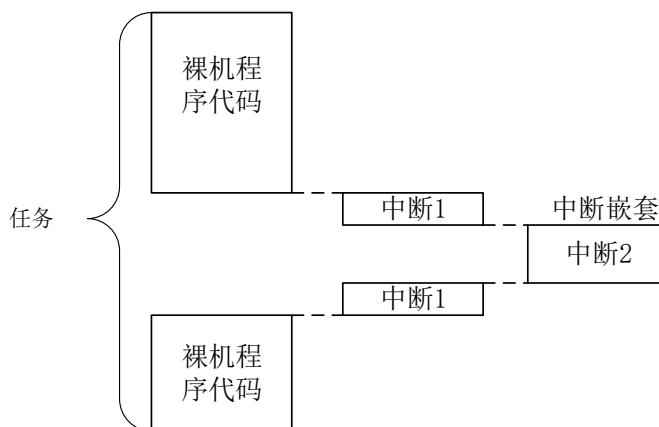


图 6-2 裸机程序运行过程图

而在 RTOS 中，将所有事情分成各个模块，每一个模块的内容看作一个任务，任务的执行顺序是灵活的，根据相应的调度算法管理任务的运行，灵活性比裸机程序强，其过程如图 6-3 所示。

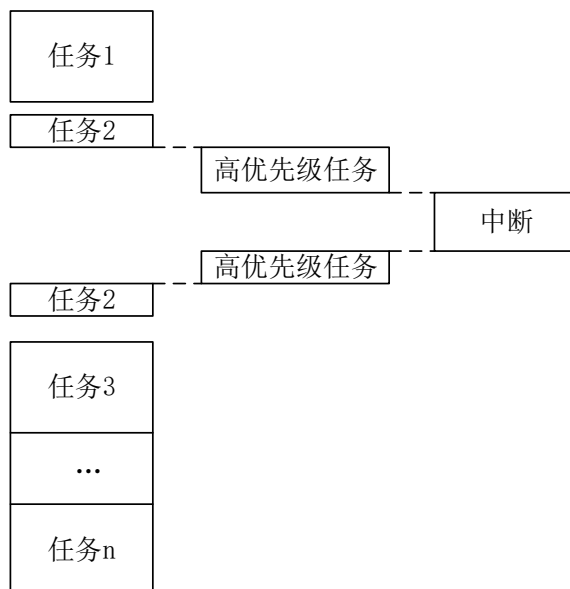


图 6-3 RTOS 程序运行过程图

FreeRTOS 中的调度算法分为时间片调度算法和抢占式调度，在 FreeRTOS 的 FreeRTOSConfig.h 文件中配置如下：

```
#define configUSE_PREEMPTION 1//使能抢占式调度器
#define configUSE_TIME_SLICING 1//使能时间片调度器
```

即使不配置 configUSE_TIME_SLICING 为 1，FreeRTOS 也会默认开启时间片调度。时间片调度算法和抢占式调度的特点简述如下：

- 时间片调度算法：每一个任务给予固定的执行时间，时间结束后进入调度器，由调度器切换到下一个任务，在默认所有任务优先级相同情况下，轮流执行所有任务。
- 抢占式调度需要设置任务优先级，在进入调度器后，调度器选择处于就绪态中优先级最高的任务作为下一个执行的任务。高优先级任务可以抢占低优先级任务，发生抢占时需要有能进入调度器的操作，调度器是任务切换的唯一实体。

6.4.4. FreeRTOS 源码解析和移植介绍

在 N200-SDK 环境中，FreeRTOS 源代码和 N200 移植相关的代码结构如下所示。

```
N200-sdk          // 存放 n200-sdk 的目录
|----software      // 存放示例程序的源代码
|----FreeRTOSv9.0.0 // FreeRTOS 示例程序目录
|----Source        //FreeRTOS 内核源代码
|----list.c//与内核相关文件
|----queue.c//与内核相关文件
|----task.c//与内核相关文件
|----include
|----FreeRTOS.h//头文件
|----list.h//头文件
|----其它头文件
|----Portable
|----GCC
|----N200 //N200 移植的相关代码，用户只需要修改此处三个
           // 代码即可完成对 FreeRTOS 的移植
|----port.c//与移植相关的代码
|----portasm.S//与移植相关的代码
|----portamacro.h//与移植相关的代码
|----MemMang
|----heap_1.c//与内存分配有关的文件
|----heap_2.c//与内存分配有关的文件
|----heap_3.c//与内存分配有关的文件
|----heap_3.c//与内存分配有关的文件
|----heap_4.c//与内存分配有关的文件
|----heap_5.c//与内存分配有关的文件
|----Demo          //FreeRTOS 的示例 Demo 程序代码
|----Makefile      //Makefile 脚本
```

如上所示，FreeRTOS 的代码层次结构分明，用户只需要修改三个文件名为“port*”的源代码，完成基本的中断和异常的底层移植，即可完成对于 FreeRTOS 的移植。

N200 移植 FreeRTOS 操作系统时，实现固定时间切换任务的操作由内核自带的 mtime 计时器中断支持，可以设置为每隔一个固定的时间段发生一次计时器中断（称之为 System Tick），在中断处理函数中进入调度器切换下一个任务。在 port.c 文件中 mtime 计时器设置代码如下：

```
// n200-sdk/software/FreeRTOSv9.0.0/Source/portable/GCC/N200/port.c 代码片段
```

```
//mtime 计时器设置函数
```

```
void vPortSetupTimer() {
```

```
uint8_t mtime_intattr;
```

```
//获取当前时间 mtime 的地址
```

```
volatile uint64_t * mtime= (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIME);
```

```
//获取 mtimecmp 的地址
```

```
volatile uint64_t * mtimecmp = (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIMECMP);
```

```
uint64_t now = *mtime;//获取当前时间
```

```
uint64_t then = now + (configRTC_CLOCK_HZ / configTICK_RATE_HZ);//计算下一次时间
```

```
//设置中断触发时间
```

```
*mtimecmp = then;
```

```
mtime_intattr=eclic_get_intattr (ECLIC_INT_MTIP);
```

```
mtime_intattr|=ECLIC_INT_ATTR_SHV;
```

```
eclic_set_intattr(ECLIC_INT_MTIP,mtime_intattr);
```

```
eclic_enable_interrupt (ECLIC_INT_MTIP); //使能 TIMER 中断
```

```
eclic_set_irq_lvl_abs(ECLIC_INT_MTIP,1); //配置 TIMER 中断级别为：级别 1
```

```
}
```

```
mtime 计时器中断处理函数如下：
```

```
void vPortSysTickHandler()
```

```
{
```

```
static uint64_t then = 0;
```

```
volatile uint64_t * mtime= (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIME); //获取当前时间的地址
```

```
volatile uint64_t * mtimecmp= (uint64_t*) (TIMER_CTRL_ADDR + TIMER_MTIMECMP); //定义设置比较值的地址
```

```
if(then != 0)
```

```
{
```

```
then += (configRTC_CLOCK_HZ / configTICK_RATE_HZ);
```

```
}
```

```
else
```

```
{
```

```
//first time setting the timer
```

```
uint64_t now = *mtime;
```

```
then = now + (configRTC_CLOCK_HZ / configTICK_RATE_HZ);
```

```
}
```

```
*mtimecmp = then;//设置下一次触发中断时间
```

```
/* Increment the RTOS tick. */
```

```
if( xTaskIncrementTick() != pdFALSE )
```

```
{
```

```
portYIELD(); //通过 ecall 异常信号, 通过异常处理机制, 进入调度器切换任务
}
}
```

更多移植代码详情, 请用户自行参见三个文件名为 “port*” 的源代码。

6.4.5. FreeRTOS 中任务与中断的关系

FreeRTOS 的任务和中断的优先级关系是移植 FreeRTOS 的难点, 需要被正确的理解, 否则程序会运行出错:

- 任务总是可以被中断打断, 任务之间具有的优先级, 但是与 “中断的优先级” 没有关系, 这两种优先级是相互独立的。
- 不调用任何 FreeRTOS API 函数的中断, 可以设置为任意的 “中断优先级”, 并且允许嵌套。
- 在 FreeRTOSConfig.h 中预先定义 configMAX_SYSCALL_INTERRUPT_PRIORITY 的值, 调用 API 函数的中断优先级只能设置为不大于该值, 支持嵌套, 但是会被内核延迟。

关于 FreeRTOS 的任务优先级和中断优先级如何设置, 以及 FreeRTOS 的更多详细信息, 请用户自行查阅相关 FreeRTOS 手册学习。

6.4.6. 运行 FreeRTOS

FreeRTOS 示例可运行于 N200-SDK 环境中, 使用《N200 系列 SDK 使用说明》中描述的运行方法按照如下步骤运行:

// 注意: 确保在 N200-SDK 中正确的安装了 RISC-V GCC 工具链, 请参见《NUCLEI_N200 系列 SDK 使用说明》了解其详情。

// 步骤一: 参照《NUCLEI_N200 系列 SDK 使用说明》中描述的方法, 编译 FreeRTOS 示例程序, 使用如下命令:

```
make dasm PROGRAM=FreeRTOSv9.0.0 BOARD=nuclei-n200 CORE=n201 OCDCFG=hbird
                                DOWNLOAD=flash
```

// 步骤二: 参照《NUCLEI_N200 系列 SDK 使用说明》中描述的方法, 将编译好的 FreeRTOS 程序下载至 FPGA 原型开发板中, 使用如下命令:

```
make upload PROGRAM=FreeRTOSv9.0.0 BOARD=nuclei-n200 CORE=n201
                                OCDCFG=hbird DOWNLOAD=flash
```

// 步骤三：参照《NUCLEI N200 系列 SDK 使用说明》中描述的方法，在 FPGA 原型开发板上运行 FreeRTOS 程序：

```
// 由于示例程序将需要通过 UART 打印结果到主机 PC 的显示屏上。参考《NUCLEI_N200 系列 SDK 使用说明》
// 所述方法将串口显示电脑屏幕设置好，使得程序的打印信息能够显示在电脑屏幕上。
//
// 由于步骤二已经将程序烧写进 FPGA 开发板的 Flash 之中，按 MCU 开发板的
// RESET 按键，则处理器复位开始执行 FreeRTOS 程序，系统会打印出目前正在执行的任务，如
//图 6-4 所示。Task1 和 Task2 交替执行。
```

[illegible]

图 6-4 程序运行串口打印信息图