实战演练之GPIO中断

一、概述

本教程将介绍如何在 T-Core 开发板上实现 RISC-V 设计——GPIO 中断：通过将开发板上的拨码开关 SW[1:0] 设置为 "00"、"01"、"10" 或 "11" 来选择流水灯的模式，然后再通过按键 KEY[1] 触发中断，控制 LED 实现不同的流水灯。

整个程序的实现主要包括：设计思路/原理的分析，使用 Makefile 编译和下载应用程序，或使用 Eclipse 软件打开 Hello World 工程、修改 main.c 主函数、编译并运行 Hello World 工程，在开发板上验证实验结果。

通过本教程，您将会掌握以下知识：

- 巩固学习使用 Eclipse 软件对 T-CORE RISC-V 的应用程序进行开发；
- 巩固学习使用 Makefile 编译和下载应用程序；
- 了解拨码开关（SW）、按键（Button）、发光二极管（LED）的工作原理及驱动方法；
- 掌握 RISC-V GPIO 中断原理；
- 学习 RISC-V 架构 PLIC 工作原理。

二、设备

1. 硬件
    - PC 主机
    - T-Core 开发套件

    （注：T-Core 是一款基于 Intel® MAX 10 FPGA 的开发套件，支持 RISC-V CPU 的板载 JTAG 调试，是学习 RISC-V CPU 设计或嵌入式系统设计的理想平台。如需了解该套件的详情，请访问 Terasic T-Core 官网。）
2. 软件
    - Quartus Prime 19.1 Lite Edition（已安装好 USB Blaster II 驱动）

    （注：Quartus Prime 软件的下载和安装（USB Blaster II 驱动的安装）可参考 "第八讲 RISC-V on T-Core 的开发流程" 文档。）

    - TCORE-RISCV-E203

    （注：TCORE-RISCV-E203-V1.0.tar.gz 可在 Terasic T-Core 官网设计资源 下载，安装可参考 "第八讲 RISC-V on T-Core 的开发流程" 文档。）

三、设计思路

3.1 T-Core 开发板外设工作原理

T-Core 开发板上有四个连接到 FPGA 端的拨码开关，这些开关都未去抖动，可在电路中用作电平触发的数据输入。每个拨码开关都直接单独地连接到 MAX 10 FPGA，当某个拨码开关拨到向上的位置时，会产生一个高电平到 FPGA；当拨到向下的位置时，会产生一个低电平到 FPGA。图 3.1 为 T-Core 开发板拨码开关和 FPGA 之间的连接示意图。

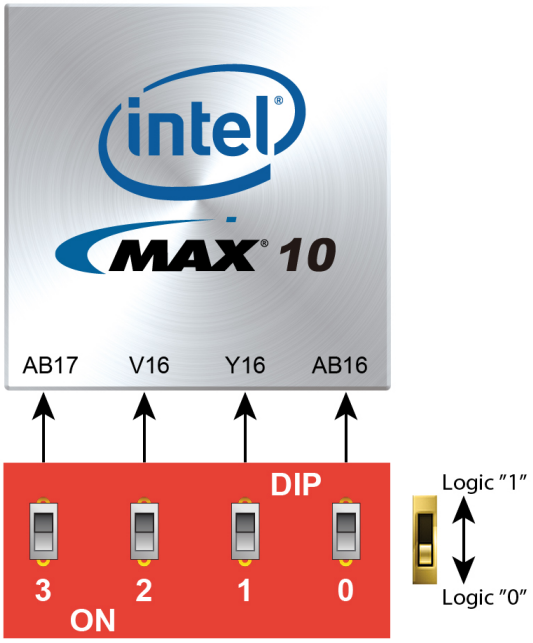

图3.1 T-Core 开发板拨码开关和 FPGA 之间的连接

T-Core 开发板上有两个连接到 FPGA 端的按键，这两个按键都利用斯密特触发器（Schmitt trigger） 电路实现了去抖动，可在电路中用作时钟或者复位输入。当按下按键时，会产生下降沿；当按键释放时，会产生上升沿，如图 3.2 所示。

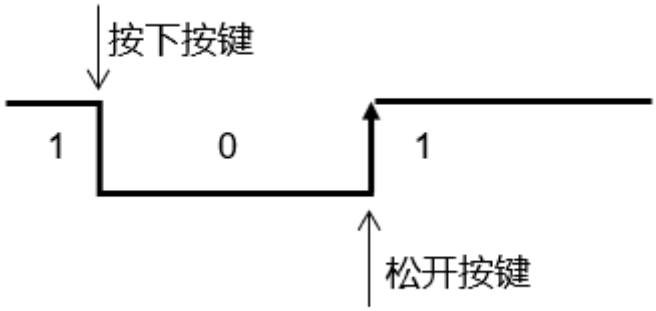

图3.2 T-Core 开发板按键工作原理

图 3.3 为 T-Core 开发板按键和 FPGA 之间的连接示意图。

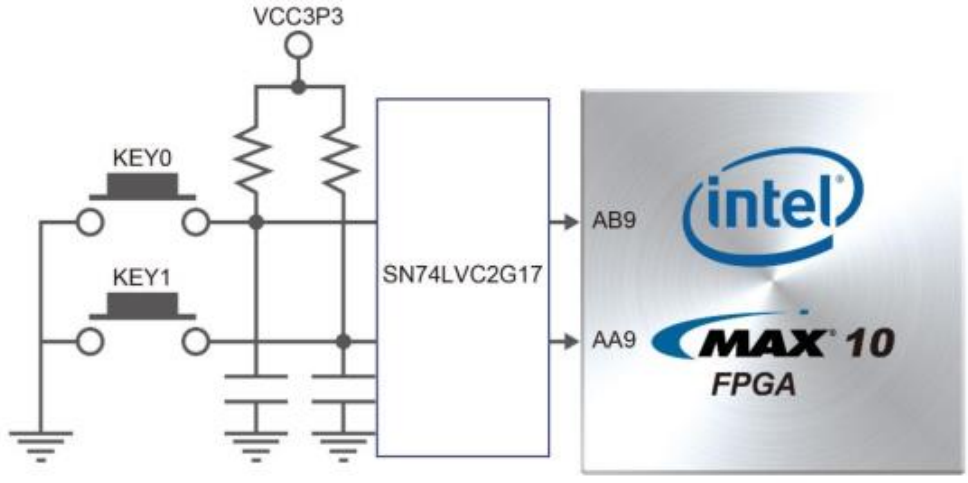

图3.3 T-Core 开发板按键和 FPGA 之间的连接

T-Core 开发板上有四个连接到 FPGA 端的、用户可控的 LED 灯。每个 LED 灯 由 MAX 10 FPGA 直接单独驱动，当 FPGA 输出高电平时，对应 LED 灯点亮；当 FPGA 输出低电平时，对应 LED 灯熄灭。图 3.4 为 T-Core 开发板 LED 灯和 FPGA 之间的连接示意图。

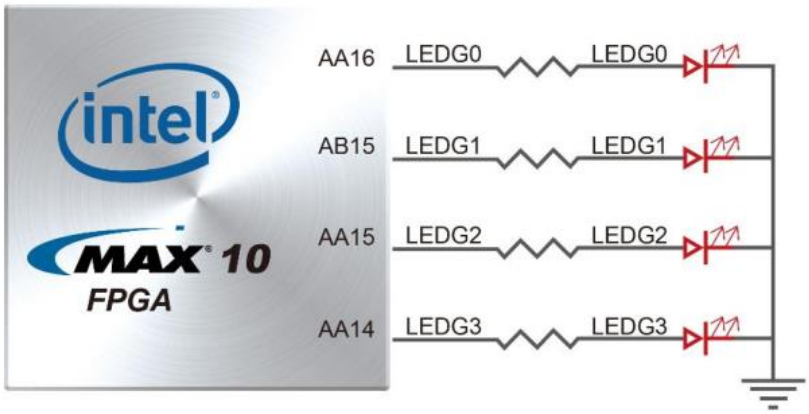

图3.4 T-Core 开发板 LED 灯和 FPGA 之间的连接

在本教程中，按键 KEY0 用于系统复位，按键 KEY1 用于提供中断信号。

3.2 GPIO 结构

GPIO的32个I/O结构完全相同，每个独立 I/O 的结构如图 3.5 所示。

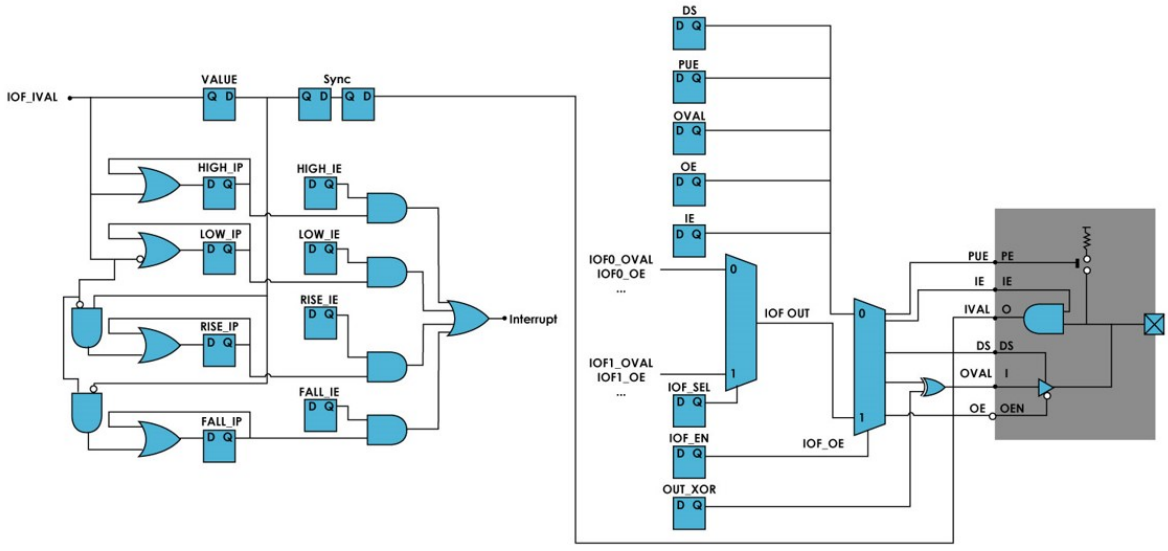

图3.5 GPIO I/O 结构图

1. 每个**I/O**的**Pad**有如下控制信号
    - PUE：上拉使能
    - IE：输入使能
    - IVAL：输入值
    - DS：输出驱动强度
    - OVAL：输出值
    - OE：输出使能
2. 每个I/O具有两种模式，软件控制模式和 IOF（H/W IO Function）控制模式，本教程采用 IOF 控制模式来实现用按键控制流水灯的流向。
3. 不管是软件控制模式，还是 IOF 控制模式，都有：

- GPIO_VALUE 输入值寄存器对应此 I/O 的比特位的值直接来自此 I/O Pad 的 IVAL 控制信号值。
- 此 I/O Pad 的 PUE 内部上拉控制信号值直接来自 GPIO_PUE 寄存器对应此 I/O 的比特位。
- 此 I/O Pad 的 DS 驱动强度控制信号值直接来自 GPIO_DS 寄存器对应此 I/O 的比特位。

3.3 GPIO 寄存器列表

本教程主要用到以下 GPIO 寄存器：GPIO_INPUT_VAL 寄存器用于反映 GPIO 的输入值，GPIO_INPUT_EN 寄存器用于在软件控制模式下配置 GPIO 的输入使能，GPIO_OUTPUT_EN 寄存器用于在软件控制模式下配置 GPIO 的输出使能，GPIO_OUTPUT_VAL 寄存器用于在软件控制模式下配置 GPIO 的输出值，GPIO_RISE_IE 寄存器用于控制上升沿中断使能，GPIO_RISE_IP 寄存器是上升沿中断等待标志。

表3.1 GPIO 寄存器列表

寄存器名称	偏移地址	复位默认值	描述
GPIO_INPUT_VAL	0x000	0x0	Pin 的输入值
GPIO_INPUT_EN	0x004	0x0	Pin 的输入使能
GPIO_OUTPUT_EN	0x008	0x0	Pin 的输出使能
GPIO_OUTPUT_VAL	0x00C	0x0	Pin 的输出值
GPIO_RISE_IE	0x018	0x0	上升沿中断使能
GPIO_RISE_IP	0x01C	0x0	上升沿中断等待标志（Pending）

3.4 GPIO 中断

GPIO 的每个 I/O 都可以根据 I/O Pad 的 IVAL 输入信号产生不同类型的中断，包括上升沿触发、下降沿触发、高电平触发和低电平触发。本实验只使用了上升沿触发，在这里仅对上升沿触发进行说明。

- 如果 GPIO_RISE_IE 寄存器对应此 I/O 的比特位被配置成 1，这表示对此 I/O Pad 的 IVAL 输入信号进行上升沿检测。
- 一旦检测到上升沿，则产生中断。产生的中断会反映在 GPIO_RISE_IP 寄存器对应此 I/O 的比特位中，该中断会一直保持，直到软件向 GPIO_RISE_IP 寄存器对应此 I/O 的比特位中写入 1 值。

3.5 GPIO 映射关系

关于 T-Core 的 RISC-V 处理器的 GPIO 与 T-Core 外设 LED 和 KEY 的映射关系可参考表 3.2。根据 T-Core 的外设与 E203 的 GPIO 映射关系可找到对应的寄存器并进行读写操作。

表3.2 T-Core 外设与 E203 的 GPIO 映射关系

T-Core 的 GPIO 外设	映射到 E203
LED0-3	GPIO0-3
SW0-3	GPIO4-7
KEY0-1	GPIO8-9

3.6 PLIC

PLIC 全称平台级别中断控制器（Platform Level Interrupt Controller），它是 RISC-V 架构定义的系统中断控制器，主要用于多个外部中断源的优先级仲裁，最后产生一根外部中断信号通给 RISC-V 处理器核。

1. PLIC 理论上可以支持高达 1024 个外部中断源，在具体的 RISC-V 核中连接的中断源个数可以不不同。在 T-CORE RISC-V 中，PLIC 连接了 GPIO、UART、I2C、PWM 等多个外部中断源，其中断分配如表 3.3 所示。

表3.3 PLIC 的中断分配

PLIC 中断源	来源
0	预留为表示没有中断
1	wdogcmp
2	rtccmp
3	uart0
4	uart1
5 — 7	qspi0 — qspi2
8 — 39	gpio0 — gpio31
40 — 43	pwm0cmp0 — pwm0cmp3
44 — 47	pwm1cmp0 — pwm1cmp3
48 — 51	pwm2cmp0 — pwm2cmp3
52	i2c

2. PLIC 将多个外部中断源仲裁为一个单比特的中断信号，送入处理器核作为机器模式外部中断，处理器核收到中断进入异常服务程序后，可以通过读 PLIC 的相关寄存器查看中断源的编号和信息。
3. 处理器核在处理完相应的中断服务程序后，可以通过写 PLIC 的相关寄存器和具体的外部中断源的寄存器来清除中断源（假设中断来源为 GPIO，则可以通过GPIO模块的中断相关寄存器清除该中断）。
4. PLIC 寄存器

   PLIC 寄存器是一个存储器地址映射的模块，寄存器地址区间如表 3.4 所示。

表3.4 PLIC 的中断分配

地址	寄存器英文名称	寄存器中文名称	复位默认值
0x0C00_0004	Source 1 priority	中断源 1 的优先级	0x0
0x0C00_0008	Source 2 priority	中断源 2 的优先级	0x0
... ...	... ...	... ...	... ...
0x0C00_0FFC	Start of pending array (read-only)	中断源 1023 的优先级	0x0
... ...	... ...	... ...	... ...
0x0C00_1000	Start of pending array	中断等待标志的起始地址	0x0
... ...	... ...	... ...	... ...
0x0C00_107C	End of pending array	中断等待标志的结束地址	0x0
... ...	... ...	... ...	... ...
0x0C00_2000	Target 0 enables	中断目标 0 的使能位	0x0
... ...	... ...	... ...	... ...
0x0C20_0000	Target 0 priority threshold	中断目标 0 的优先级门槛	0x0
0x0C20_0004	Target 0 claim/complete	中断目标 0 的响应/完成	0x0

PLIC 理论上支持 1024 个中断源，所以这里有 1024 个优先级寄存器。优先级寄存器为 32 位，但是优先级寄存器有效位数与优先级数量相关，假设需要实现 0~7 这八个优先级，则有效位为 3 位，其余位都为 0。

每个中断等待标志 IP 是 1 位宽，每个 IP 等待寄存器为 32 位宽，即每个 IP 等待寄存器可包含 32 个中断等待标志 IP，那么 1024 个中断源就需要 32 个 32 位的 IP 等待寄存器，也就是从 0xC00_1000 到 0xC00_107C 的 32 个地址。

每个中断使能 IE 是 1 位宽，每个 IE 使能寄存器是 32 位宽，即每个 IE 使能寄存器可包含 32 个中断使能 IE，那么1024 个中断源就需要 32 个 32 位的 IE 寄存器，也就是从 0xC00_2000 到 0xC00_207C 的 32 个地址。

Target 0 priority threshold 对应 target 0 的阈值寄存器，虽然它也是 32 位，但阈值寄存器的有效位数应该与中断优先级寄存器的有效位数相同。

PLIC 的中断响应寄存器可读，中断完成寄存器可写，就组合成一个可读可写的寄存器 Target 0 claim/complete。

3.7 中断相关寄存器

1. **mtvec**

   mtvec 寄存器用于配置异常的入口地址。

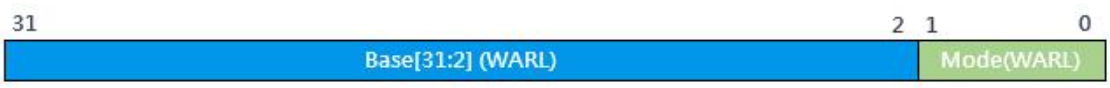

图3.6 mtvec 寄存器格式

- 若 MODE 的值为0，则所有的异常响应时处理器均跳转到 BASE 值指示的 PC 地址。
- 若 MODE 的值为1，则侠义的异常发生，处理器跳转到 BASE 值指示的 PC 地址；侠义的中断发生时，处理器跳转到 BASE+4×CAUSE 值指示的 PC 地址。CAUSE 的值表示中断对应的异常编号。

2. **mcause**

mcause 寄存器，用于保存进入异常之前的出错原因，以便对异常原因进行诊断和调试。

图3.7 mcause 寄存器格式

最高 1 位为 Interrupt 域，低 31 位为异常编号域，这两个域的组合用于指示 RISC-V 架构定义的 12 种中断类型和 16 种异常类型。

3. **mepc**

mepc 寄存器用于保存进入异常之前指令的 PC 值，作为异常的返回地址。

图3.8 mepc 寄存器格式

虽然 mepc 寄存器会在异常发生时自动被硬件更新，但是 mepc 寄存器本身也是一个可读可写的寄存器，因此软件也可以直接写该寄存器以修改其值。

对于侠义的中断和侠义的异常而言，RISC-V 架构定义其返回地址（更新的 mepc）有些细微差别。

- 出现中断时，中断返回地址 mepc 的值被更新为下一条尚未执行的指令。
- 出现异常时，中断返回地址 mepc 的值被更新为当前发生异常的指令 PC。

4. **mstatus**

mstatus 寄存器是机器模式下的状态寄存器。如图 3.7 所示，该寄存器包含若干不同的功能域，其中 MIE 域表示全局中断使能。

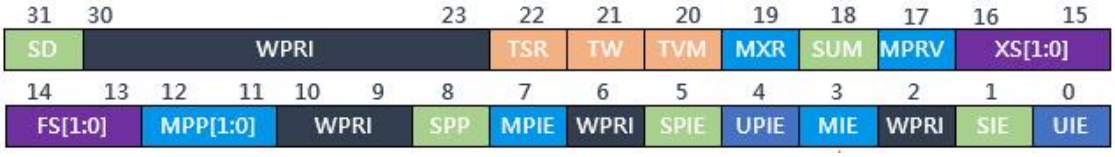

图3.9 mstatus 寄存器格式

- 当 MIE 域的值为 1 时，表示所有中断的全局开关打开。
- 当 MIE 域的值为 0 时，表示全局关闭所有的中断。

5. **mie**

mie 寄存器用于控制不同中断类型的局部屏蔽。之所以称为局部屏蔽，是因为相对而言，mstatus 寄存器中的 MIE 域提供了全局中断使能。

图3.10 mie 寄存器格式

- MEIE 域控制机器模式下外部中断的屏蔽。当 MEIE 域为0，屏蔽外部中断；当 MEIE 域为1，使能外部中断。
- MTIE 域控制机器模式下计时器中断的屏蔽。
- MSIE 域控制机器模式下软件中断的屏蔽。

3.4 程序流程图

本教程是在 GPIO 的 IOF 控制模式下，实现用拨码开关 SW[1:0] 来选择流水灯的模式，然后再通过按键 KEY[1] 触发中断，控制 LED 实现不同的流水灯。

程序开始时，先对 GPIO 进行初始化，再对中断进行初始化，然后在 while(1) 中，按下按键 KEY1 时触发中断后，在中断处理函数中读取拨码开关 SW[1:0] 输入的值，得到流水灯模式 led_module，然后 LED 按照拨码开关选择的模式呈现对应的流水灯闪烁。

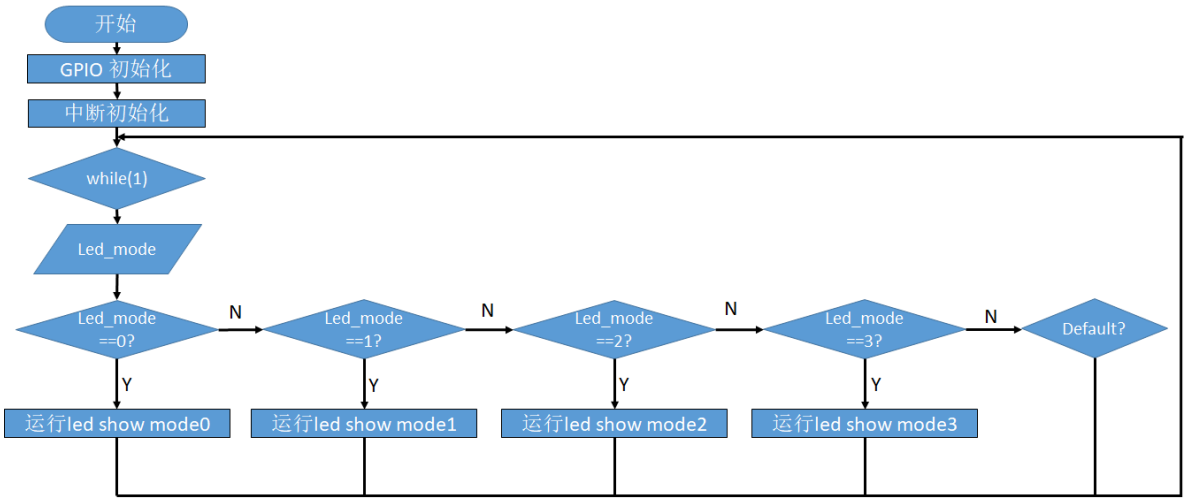

图3.11 GPIO 中断主程序框图

下面对中断的整个响应等待过程进行分析，在中断初始化后，中断被使能，此时系统会对中断信号进行监控，当发生中断时，程序从主程序跳转到相应的中断入口，对中断源进行判断，当中断源为 KEY1 对应的 GPIO 的 IO 时，调用 KEY1 中断处理函数，读取拨码开关 SW[1:0] 输入的值，得到流水灯模式 led_module，然后退出当前中断，系统继续监控等待下次中断。当退出中断后，程序回到图 3.11 所示的主循环中。

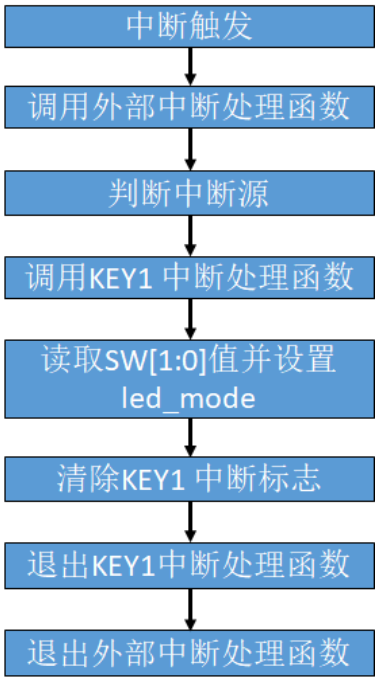

图3.12 GPIO 中断响应程序框图

四、操作步骤

4.1 使用 Makefile 编译和下载应用程序

在本小节中，我们将使用直接操作寄存器的方式，实现用拨码开关 SW[1:0] 来选择流水灯的模式，然后再通过按键 KEY[1] 触发中断，控制 LED 实现不同的流水灯。

4.1.1 创建并构建工程

1. 创建工程文件夹

   工程通常包含很多例如 .c/.h 或 Makefile 等的设计文件，这些文件通常被存储在同一文件夹下，因此，需要创建一个工程文件夹来存储设计文件和生成文件。

   可以在 "~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK" 的 software 文件夹下创建一个 "demo_interrupt" 文件夹，所以这个文件夹的绝对路径为：

   "~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software/demo_interrupt"

2. 创建程序文件（.c文件）

   首先，在 "demo_interrupt" 文件夹下创建一个 "demo_interrupt.c" 的文本文档。

   包含需要的头文件，在本实验中使用的宏定义和函数均可以在这些头文件中找到。

```c
#include <stdio.h>
#include <stdlib.h>
#include "platform.h"
#include <string.h>
#include "plic/plic_driver.h"
#include "encoding.h"
#include <unistd.h>
#include "stdatomic.h"
```

   定义 LED 和按键的掩码，单独定义按键 KEY1 的偏移，根据 GPIO 的 PLIC 基地址定义按键 KEY1 对应 GPIO 的 IO PLIC 地址，用于 PLIC 对 KEY1 的中断进行管理。

```c
// LED0-3: bit 0-3
// SW0-3:  bit 4-7
// KEY0-1: bit 8-9

#define TERASIC_LED_MASK  0x0000000F // led mask
#define TERASIC_SW01_MASK 0x00000030 // switch 0 and 1 mask
#define TERASIC_KEY_MASK  0x00000300 // key mask, key0 use for reset
#define KEY_1_GPIO_OFFSET 9          // key1 offset

#define PLIC_INT_DEVICE_KEY_1 (PLIC_INT_GPIO_BASE + KEY_1_GPIO_OFFSET)
```

   定义存放外部中断处理的指针的指针数组，当有多个中断信号时，用该数组进保存。

```c
// Structures for registering different interrupt handlers
// for different parts of the application.
typedef void (*function_ptr_t) (void);
function_ptr_t g_ext_interrupt_handlers[PLIC_NUM_INTERRUPTS];
```

   定义一个空的中断处理函数，后面会用于清除其他外部中断信号的响应。

```c
// do nothing interrupt handler
void no_interrupt_handler (void) {};
```

使用结构体 plic_instance_t 实例化一个 PLIC 的数据对象 g_plic。

```
// Instance data for the PLIC.
plic_instance_t g_plic;
```

其中结构体 pilc_instance_t 的定义在 plic_driver.h 中，可以看到在结构体中定义了基地址，中断信号源，中断优先级。（注意：此处仅仅是进行说明，不需要将下面的代码写入 demo_interrupt.c）。

```
typedef struct __plic_instance_t
{
    uintptr_t base_addr;

    uint32_t num_sources;
    uint32_t num_priorities;

} plic_instance_t;
```

定义全局变量 led_mode（流水灯模式）、led_value（LED 赋值）、gpio_value（ GPIO 输出值）。

```
// blinking led direction flag
volatile int led_mode=0;
volatile uint8_t led_value=0x1;
volatile uint32_t gpio_value=0;
```

定义外部中断处理函数，handle_m_ext_interrupt 函数。首先获取中断源编号，如果中断源编号在 1 和 52 之间时，执行中断响应函数，当中断完成时，进行相应处理退出中断。

需要说明的是在整个 demo_interrupt.c 没有直接对这个外部中断处理函数进行调用，但是这里是不能去掉的，因为在系统的初始化文件 init.c 中调用了该函数，即系统初始化时将这个外部中断处理函数进行了关联注册，当有外部中断产生时，对于整个主程序来说，这个 handle_m_ext_interrupt 是中断处理的入口，当进行编译的时候，demo_interrupt.c 是会和一些系统的初始化或者配置文件一起编译的，比如这里的 init.c。

```
//external interrupt handle
void handle_m_ext_interrupt()
{
    // get interrupt number
    plic_source int_num  = PLIC_claim_interrupt(&g_plic);
    if ((int_num >=1 ) && (int_num <= PLIC_NUM_INTERRUPTS)){
        g_ext_interrupt_handlers[int_num]();// run interrupt service function
    }
    else
    {
        exit(1 + (uintptr_t) int_num);
    }
    // interrupt complete
    PLIC_complete_interrupt(&g_plic, int_num);
}
```

定义 KEY1 的中断处理函数 key1_irq_handler，读取拨码开关 SW[1:0] 来对流水灯模式进行设置，设置 LED 初值为 0x1，将 GPIO_RISE_IP 对应比特位写1，清除当前中断信号。

```c
void key1_irq_handler(void) {
    // Read SW[1:0] value to set led show mode
    led_mode=(GPIO_REG(GPIO_INPUT_VAL)&TERASIC_SW01_MASK)>>4;

    // set default value
    led_value=0x1;

    // Clear the GPIO Pending interrupt by writing 1.
    GPIO_REG(GPIO_RISE_IP) |= (0x1 << KEY_1_GPIO_OFFSET);
};
```

定义 PLIC 中断注册函数 register_plic_irqs，首先初始化 PLIC，再将所有中断处理函数都初始为空，接着先使能WDOGCMP，设置其优先级为 1，然后将 key1_irq_handler 存入对应中断处理函数的指针数组，最后再将 key1 对应的GPIO 的 IO 的中断进行使能，并设置优先级为 1。

```c
void register_plic_irqs (){
    // init plic

PLIC_init(&g_plic,PLIC_CTRL_ADDR,PLIC_NUM_INTERRUPTS,PLIC_NUM_PRIORITIES);

    // init interrupt handlers as null
    for (int i=0; i<PLIC_NUM_INTERRUPTS; i++){
        g_ext_interrupt_handlers[i] = no_interrupt_handler;
    }

    // to enable KEY1. must enable WDOGCMP first
    PLIC_enable_interrupt (&g_plic, PLIC_INT_WDOGCMP);
    PLIC_set_priority(&g_plic, PLIC_INT_WDOGCMP, 1);

    g_ext_interrupt_handlers[PLIC_INT_DEVICE_KEY_1] = key1_irq_handler;

    // Have to enable the interrupt both at the GPIO level,
    // and at the PLIC level.
    PLIC_enable_interrupt (&g_plic, PLIC_INT_DEVICE_KEY_1);

    // Priority must be set > 0 to trigger the interrupt.
    PLIC_set_priority(&g_plic, PLIC_INT_DEVICE_KEY_1, 1);

}
```

定义 GPIO 的初始化函数 gpio_init，使能 LED0-3、KEY0-1、SW0-1 对应的 GPIO 输出。

```c
void gpio_init(){
    // Set LED0-3 output
    GPIO_REG(GPIO_OUTPUT_EN) |= TERASIC_LED_MASK;

    // Set KEY0-1 input
    GPIO_REG(GPIO_INPUT_EN)  |= TERASIC_KEY_MASK;

    // Set SW0-1 input
    GPIO_REG(GPIO_INPUT_EN)  |= TERASIC_SW01_MASK;
}
```

定义中断初始化函数 interrupt_init，首先将 GPIO_RISE_IE 中与 KE1_GPIO 对应的比特位设置成 1，开启 GPIO 对应位的中断使能，再调用 CSR 的清除函数，清除 CSR 寄存器组中 mie 寄存器中 MEIE 的对应比特位，屏蔽外部中断，再进行 PLIC 的外部中断注册，在注册完成后，再将 mie 的 MEIE 域设置为 1，使能外部中断，最后将 mstatus 寄存器的 MIE 域设置为 1，所有中断的全局打开。（注意：当全局打开时，被使能的中断才可以中断，当全局关闭时，所有中断均不能中断）

```
void interrupt_init(){
    // Enable KEY1 Interrupt
    GPIO_REG(GPIO_RISE_IE) |= (0x1<< KEY_1_GPIO_OFFSET);

    // clear external interrupt
    clear_csr(mie, MIE_MEIE);

    // init external interrupt
    register_plic_irqs();

    // Enable external interrupt and global interrupt
    set_csr(mie, MIE_MEIE);
    set_csr(mstatus, MSTATUS_MIE);
}
```

定义 delay 延时函数，用于不精确的计数延时。

```
void delay(int s){
    volatile int i=s*1000;
    while(i--);
}
```

定义 led_show_mode0、led_show_mode1、led_show_mode2、led_show_mode3 四种流水灯模式，led_show_mode0 为从右向左闪烁，led_show_mode1 为从左向右闪烁，led_show_mode2 为从右向左循环点亮再从右向左循环熄灭，led_show_mode3 为从左向右循环点亮再从左向右循环熄灭。

```
void led_show_mode0(){
    // led 0001->0010->0100->1000->0001, left loop
    if(led_value != 0x08)
        led_value = led_value << 1;
    else
        led_value = 0x01;
}

void led_show_mode1(){
    // led 1000->0100->0010->0001->1000, right loop
    if(led_value != 0x01)
        led_value = led_value >> 1;
    else
        led_value = 0x08;
}

void led_show_mode2(){
    // led 0001->0011->0111->1111->1110->1100->1000->0000->0001, left
loop
    if((led_value&0x1)&&(led_value!=0xf)) // 0001->0011->0111->1111
        led_value = ((led_value << 1)+1)&0xf;
    else if(led_value&0x8) //1111->1110->1100->1000->0000
        led_value = (led_value<<1)&0xf;
```

```
    else
        led_value=0x1;
}

void led_show_mode3(){
    // led 1000->1100->1110->1111->0111->0011->0001->0000->1000, right
loop
    if((led_value&0x8)&&(led_value!=0xf)) // 1000->1100->1110->1111
        led_value = ((led_value >> 1)+0x8)&0xf;
    else if(led_value&0x1) //1111->0111->0011->0001->0000
        led_value = (led_value>>1)&0xf;
    else
        led_value=0x8;
}
```

定义 update_led_value 函数，用于将 GPIO_OUTPUT_VAL 寄存器输出更新为 led_value。

```
void update_led_value(){
    // update gpio_value
    gpio_value |= (uint32_t)(led_value << 0) & TERASIC_LED_MASK;
    //set gpio reg
    GPIO_REG(GPIO_OUTPUT_VAL) = (GPIO_REG(GPIO_OUTPUT_VAL)&
(~TERASIC_LED_MASK))|gpio_value;
    // clear gpio_value after setting gpio reg
    gpio_value=0;
}
```

最后定义程序主入口 main 函数，首先对 GPIO 进行初始化，再对中断进行初始化，接着读取 SW[1:0] 的输入值获取 led_module，给 led_value 设置初值，用于不同模式下的条件判断，在 while 循环中，switch 条件语句用于选择流水灯模式，再使用 update_led_value 函数将 GPIO_OUTPUT_VAL 寄存器输出更新为 led_value。

```
int main(int argc, char **argv)
{
    // configure gpio
    gpio_init();

    // initialize interrupt
    interrupt_init();

    // start blinking led
    led_mode=(GPIO_REG(GPIO_INPUT_VAL)&TERASIC_SW01_MASK)>>4;    //
initial mode
    led_value=0x1;  // default led value

    while(1){
        switch(led_mode){
            case 0: //sw[1:0]=00
                led_show_mode0();
                break;
            case 1: //sw[1:0]=01
                led_show_mode1();
                break;
            case 2: //sw[1:0]=10
                led_show_mode2();
                break;
```

```
            case 3: //sw[1:0]=11
                led_show_mode3();
                break;
            default:
                break;
        }

        // update leds' value
        update_led_value();

        // delay
        delay(1000);
    }   //end of while

    return 0;
}
```

3. 创建 **Makefile** 文件

在 "demo_interrupt" 文件夹下创建一个空白文本文档并命名为 "Makefile"，然后在文档中写入如下所示内容。Makefile 文件中制定了 Linux 编译工程的一系列规则，最后编译生成可执行文件。

```
TARGET = demo_interrupt
CFLAGS += -O1

BSP_BASE = ../../bsp

C_SRCS += demo_interrupt.c
C_SRCS += $(BSP_BASE)/tcore-e203/drivers/plic/plic_driver.c

include $(BSP_BASE)/tcore-e203/env/common.mk
```

在 Makefile 中："TARGET" 定义了生成的可执行文件名字，这个例子中生成的可执行文件名将为 "demo_interrupt"。

4.1.2 编译工程

1. 使用 Linux 命令 "cd" 切换当前目录至工程路径 "~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software"，然后，执行 "make software PROGRAM=demo_interrupt" 命令编译应用程序。如图 4.1.1 所示。

```
cd Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software      # 切换当前目录至工程路径
make software PROGRAM=demo_interrupt                 # 编译应用程序
```

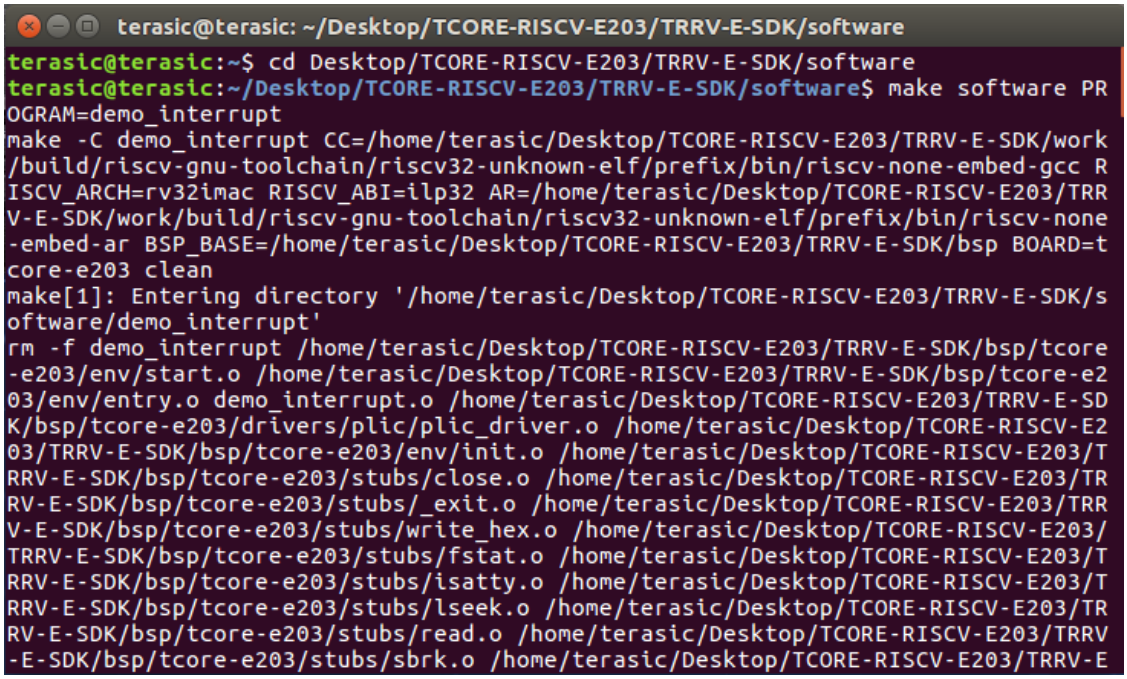

图4.1.1 编译应用程序

2. 工程编译完成之后，可以看到在 "demo_interrupt" 文件夹下生成了可执行文件 "demo_interrupt"，如图 4.1.2 所示。

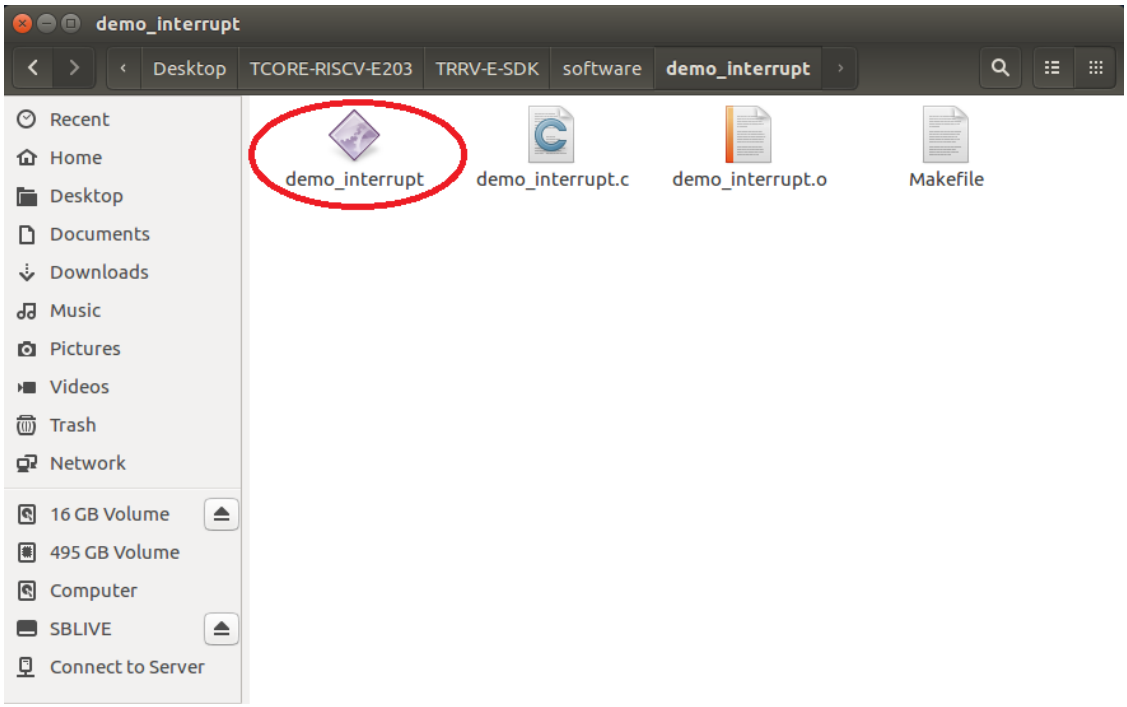

图4.1.2 编译生成二进制文件

4.1.3 执行工程

1. 关闭 T-Core 开发板电源后，将开发板上的 SW2: SW2.1=1，SW2.2=0，选择 RISC-V JTAG 链路。

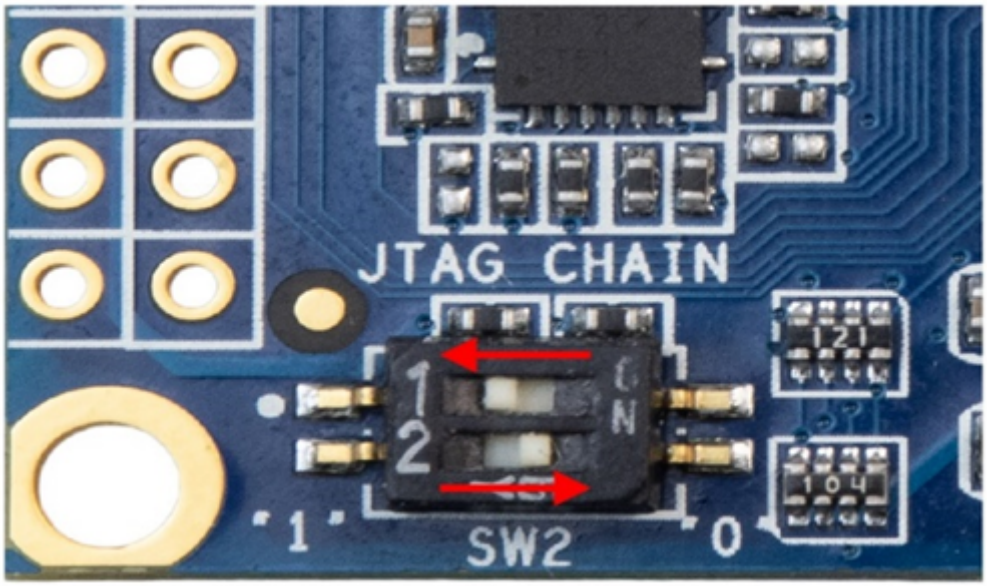

图4.1.3 设置 SW2 开关

2. 将 USB Blaster II 线缆的一端连接到开发板的 USB Blaster 接口（J2），另一端连接至 PC 主机的 USB 接口。

图4.1.4 连接开发板和 PC

3. 使用 "make upload PROGRAM=demo_interrupt" 将可执行文件 "demo_interrupt" 下载到 T-Core 开发板的 QSPI Flash 中。

```
make upload PROGRAM=demo_interrupt
```

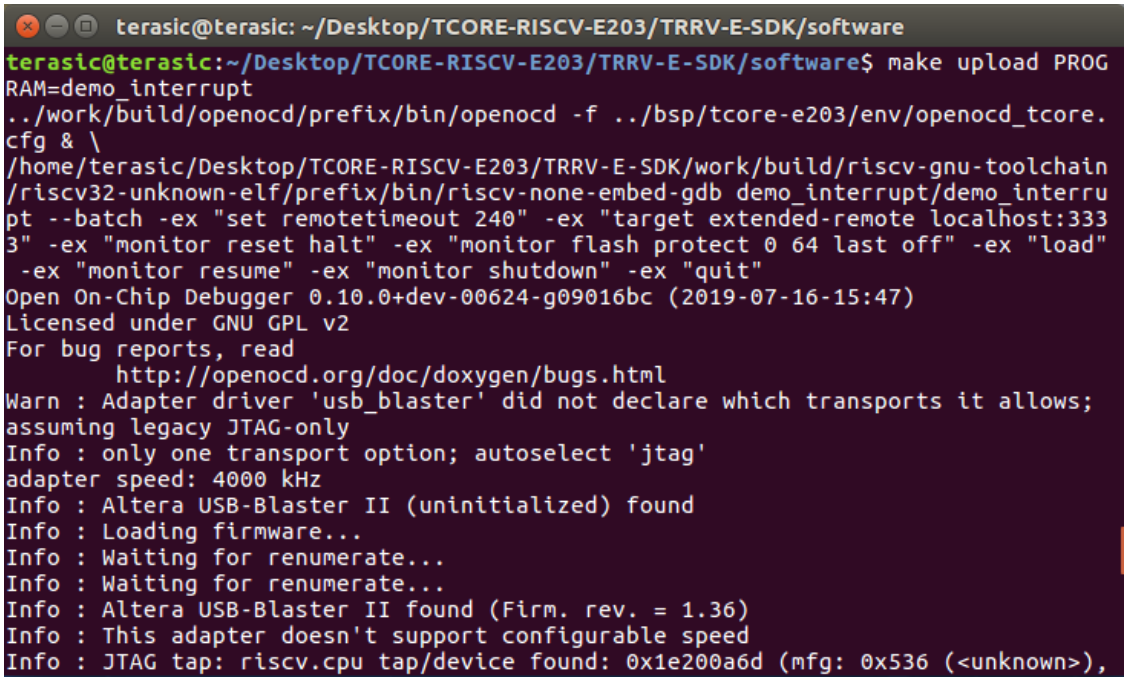

图4.1.5 下载可执行文件

4.1.4 运行结果

程序下载完成后，若 SW[1:0] 为 "00"，按下 KEY1，LED 从右向左呈现流水灯闪烁；若 SW[1:0] 为 "01"，再次按下 KEY1，LED 从左向右呈现流水灯闪烁；若 SW[1:0] 为 "10"，再次按下 KEY1，LED 从右向左循环点亮再从右向左循环熄灭；若 SW[1:0] 为 "11"，再次按下 KEY1，LED 从左向右循环点亮再从左向右循环熄灭。

4.2 使用 Eclipse 软件编译和下载应用程序

在进行下面的操作前，请先将在第八讲中创建的 blinking_LED 工程复制到 "~/eclipse-workspace" 文件夹。（注：请使用依据 v1.1 及以上版本的第八讲手册创建的 blinking_LED 工程）

4.2.1 打开 demo_interrupt 工程

1. 将文件夹命名由 "blinking_LED" 修改为 "demo_interrupt"，如图 4.2.1 所示。

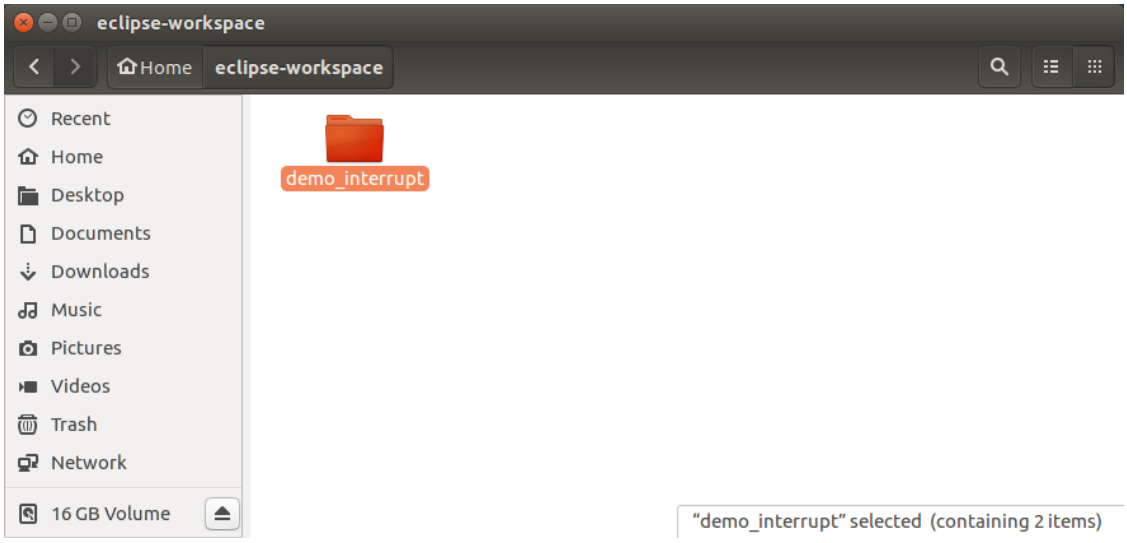

图4.2.1 修改文件名为 "demo_interrupt"

2. 双击 GNU_MCU_Eclipse 文件夹中的 eclipse 文件夹下的可执行文件 eclipse，启动 Eclipse 软件。

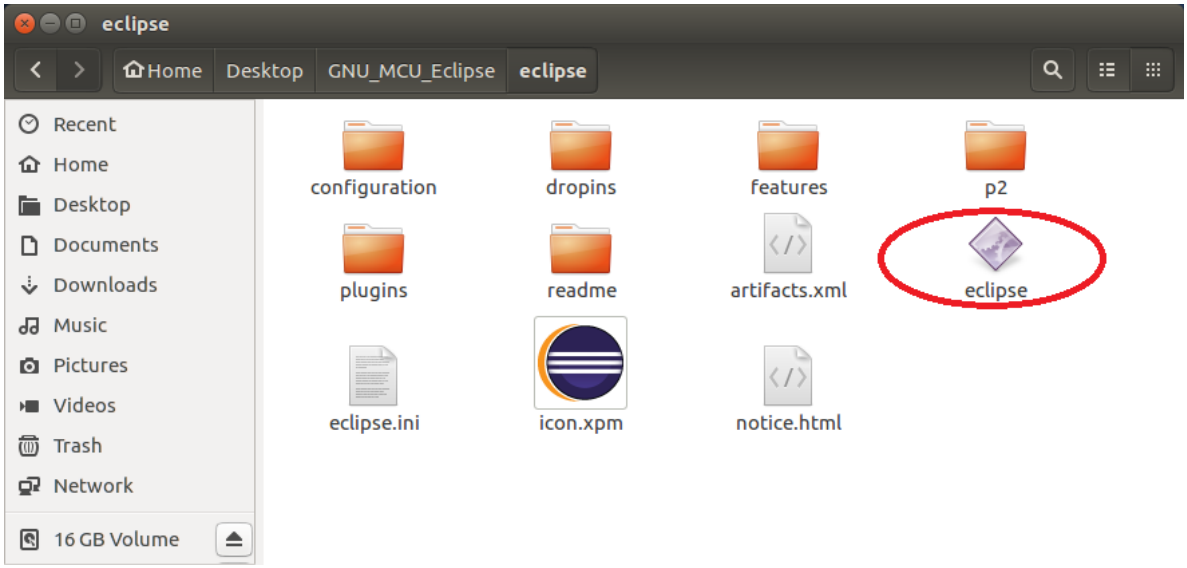

图4.2.2 启动 Eclipse

3. 启动 Eclipse 后，弹出设置 Workspace 的对话框，如图 4.2.3 所示，默认为 home 下的 eclipse-workspace（可根据需要自行设置）。

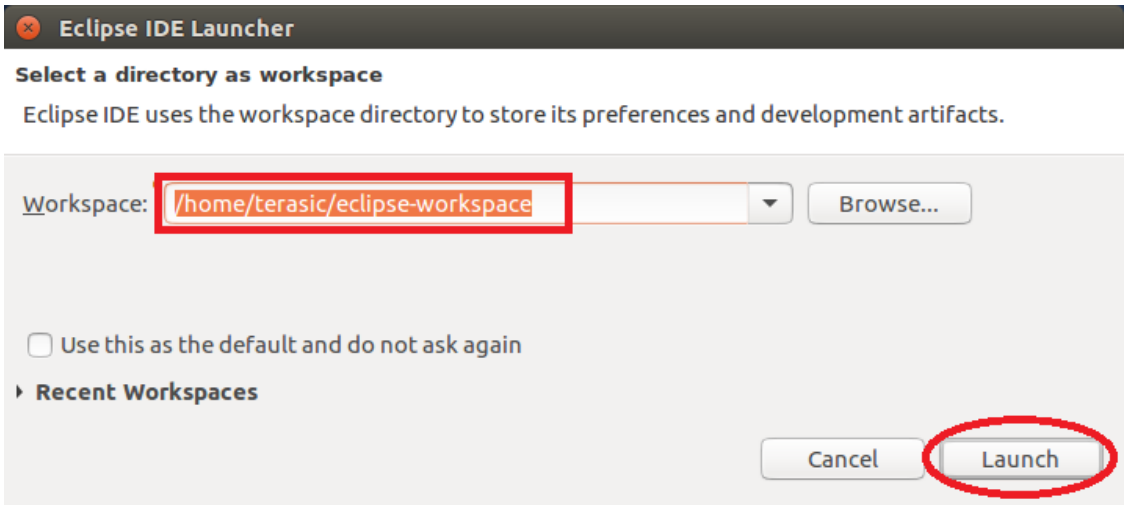

图4.2.3 设置 Workspace

4. 设置好 Workspace 目录后，单击 Launch，将会启动 Eclipse，进入 Welcome 界面，如图 4.2.4 所示。

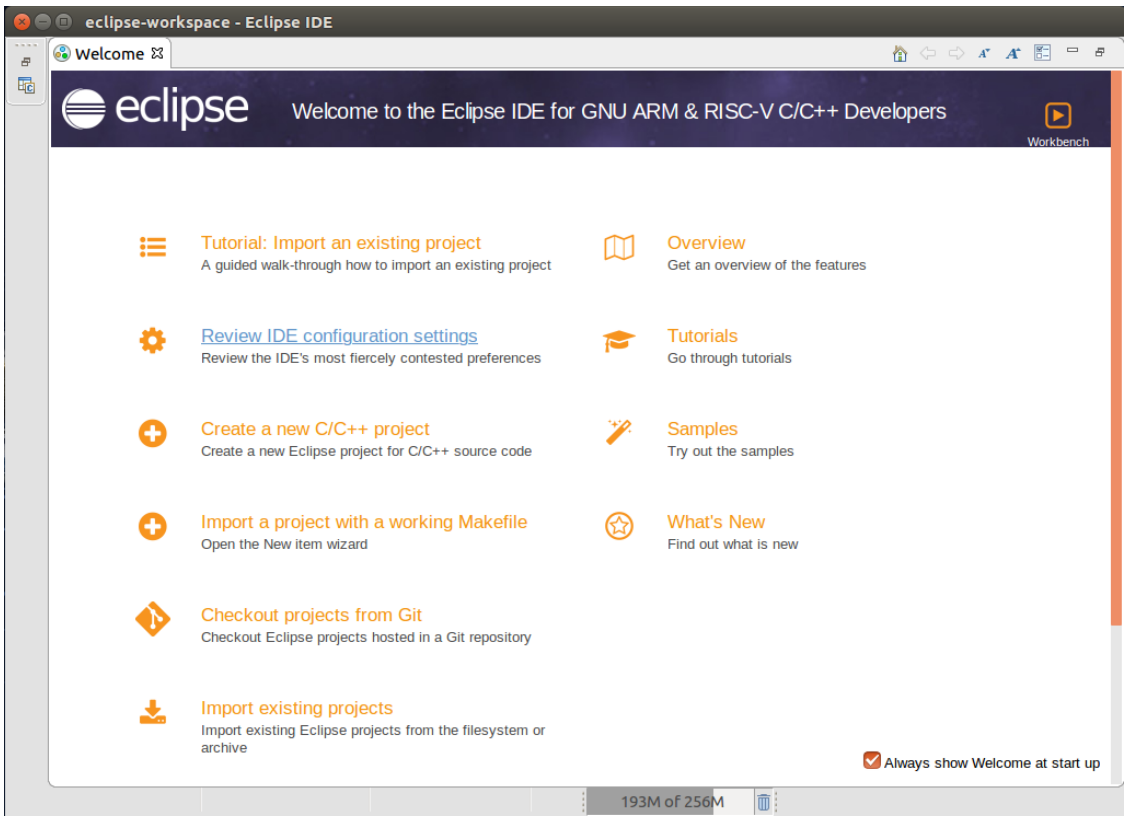

图4.2.4 进入 Eclipse 界面

5. 点击 Welcome 处的叉号，关闭 Welcome 界面，如图 4.2.5 所示。

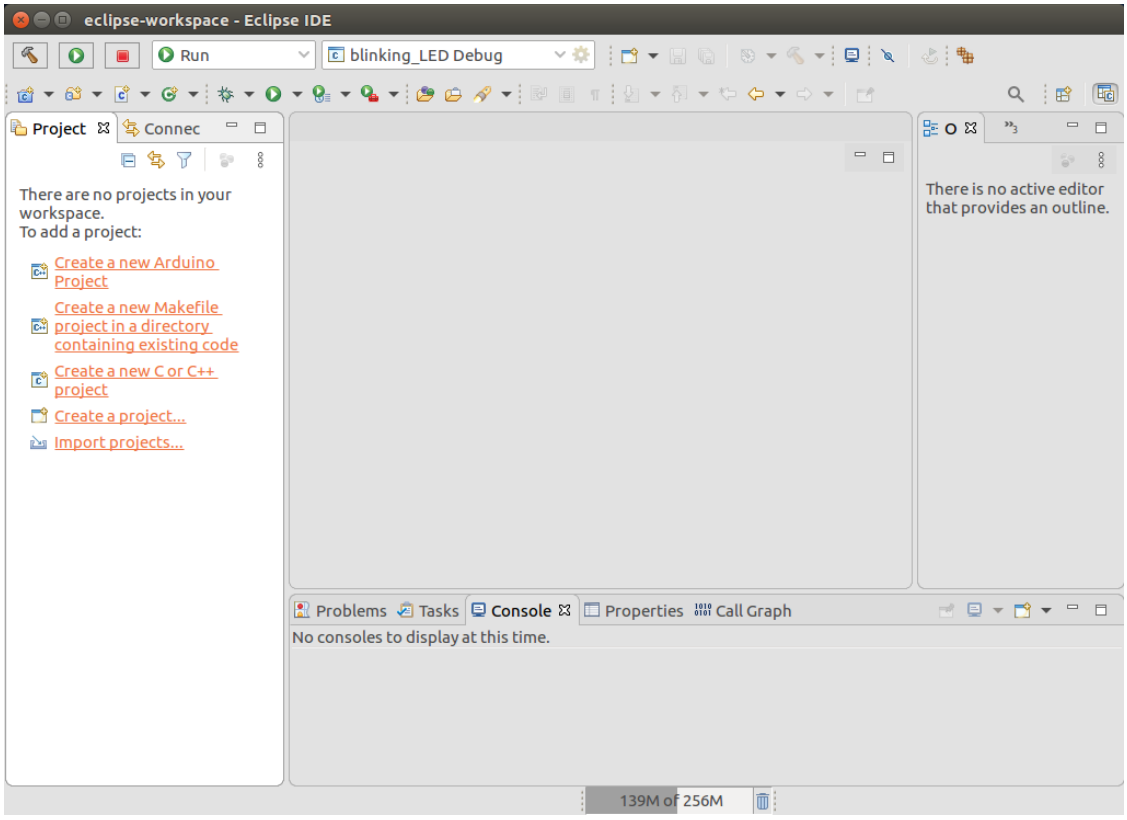

图4.2.5 进入 Eclipse 界面

6. 点击菜单栏 File -> Import... 导入工程，出现如图 4.2.6 所示界面，选择 "Existing Projects into Workspace"，点击 Next。

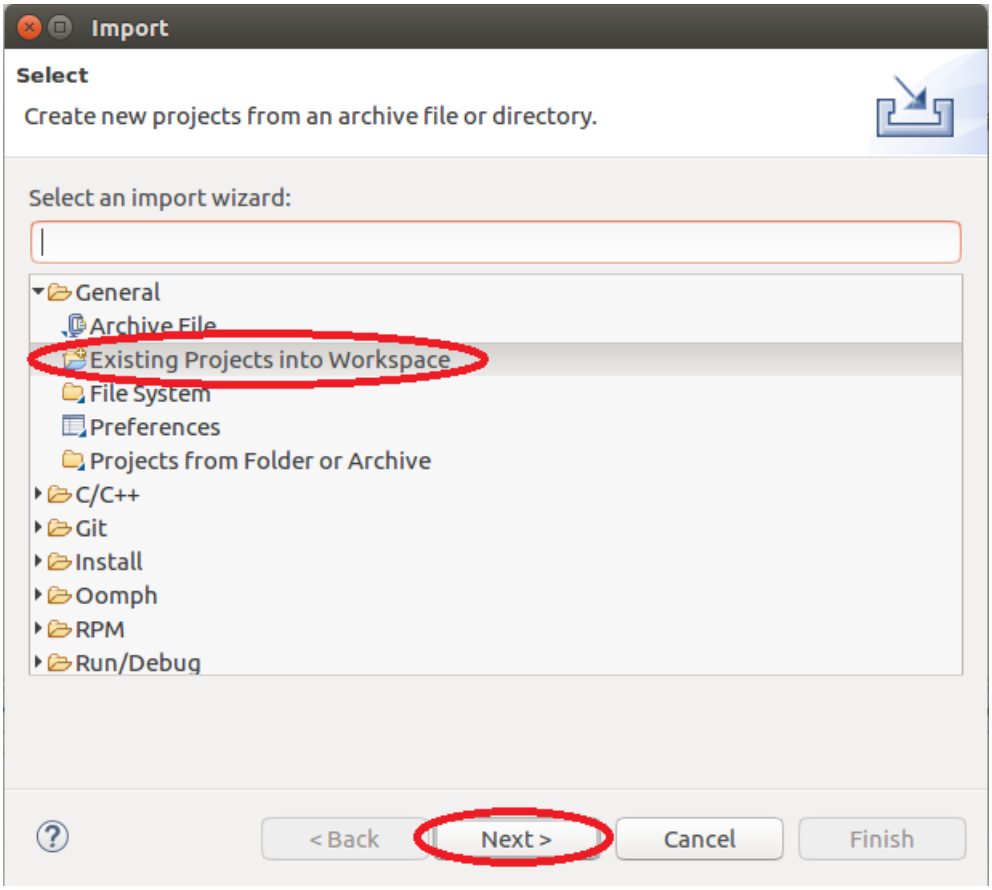

图4.2.6 选择导入工程类型

7. 点击 Browse 导入已有的工程。

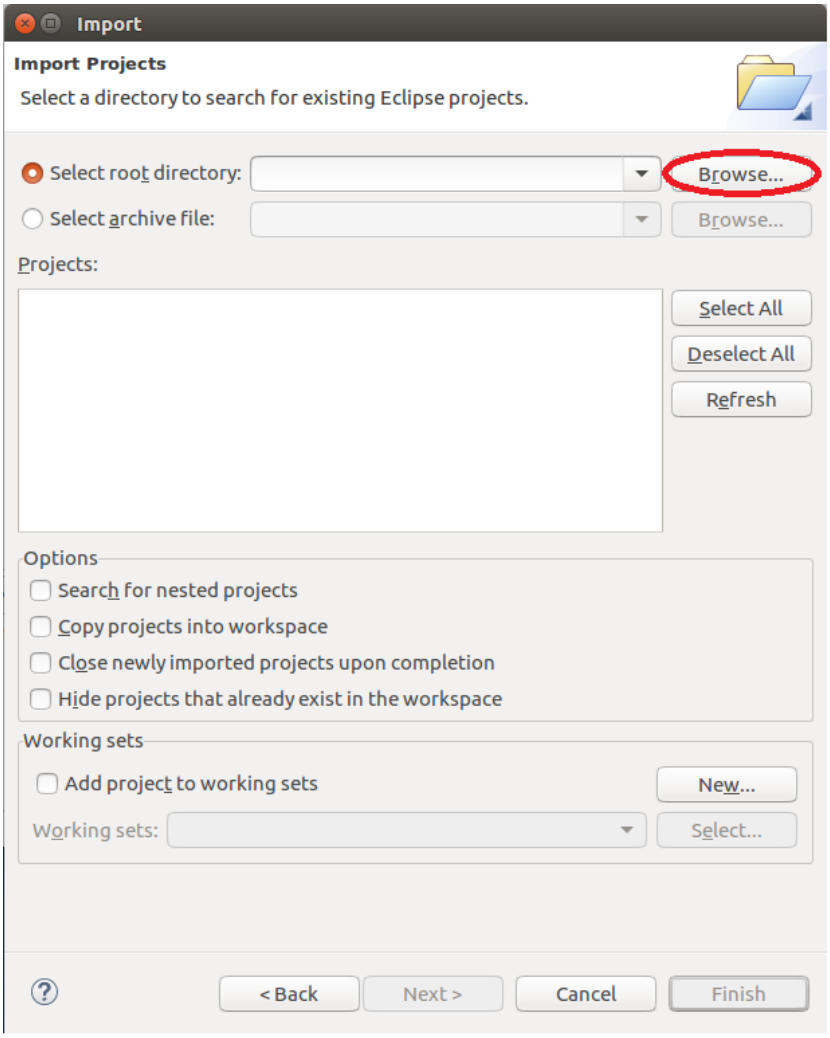

图4.2.7 点击 Browse

8. 选择要添加的 demo_interrupt 工程，点击 OK。

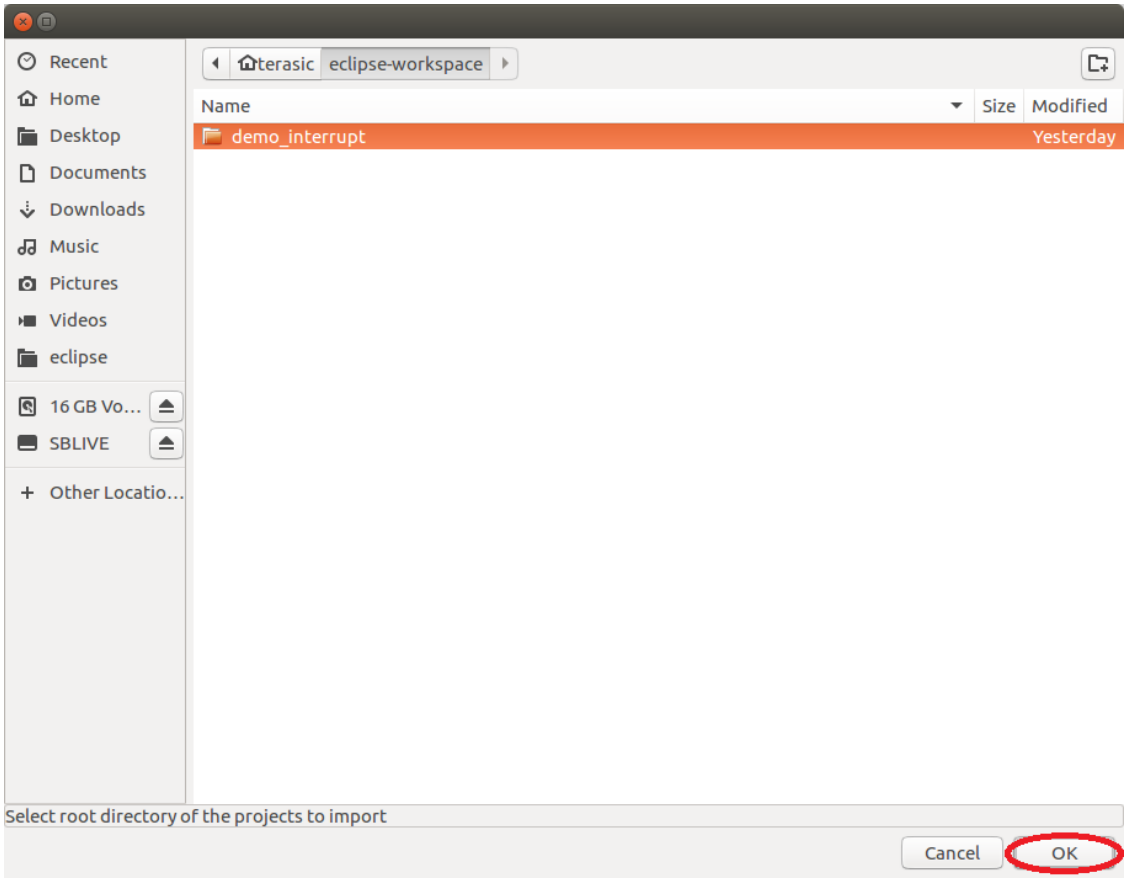

图4.2.8 添加 demo_interrupt 工程

9. 勾选 "Add projects to working sets" 将 demo_interrupt 工程添加到当前工作空间，点击 Finish。

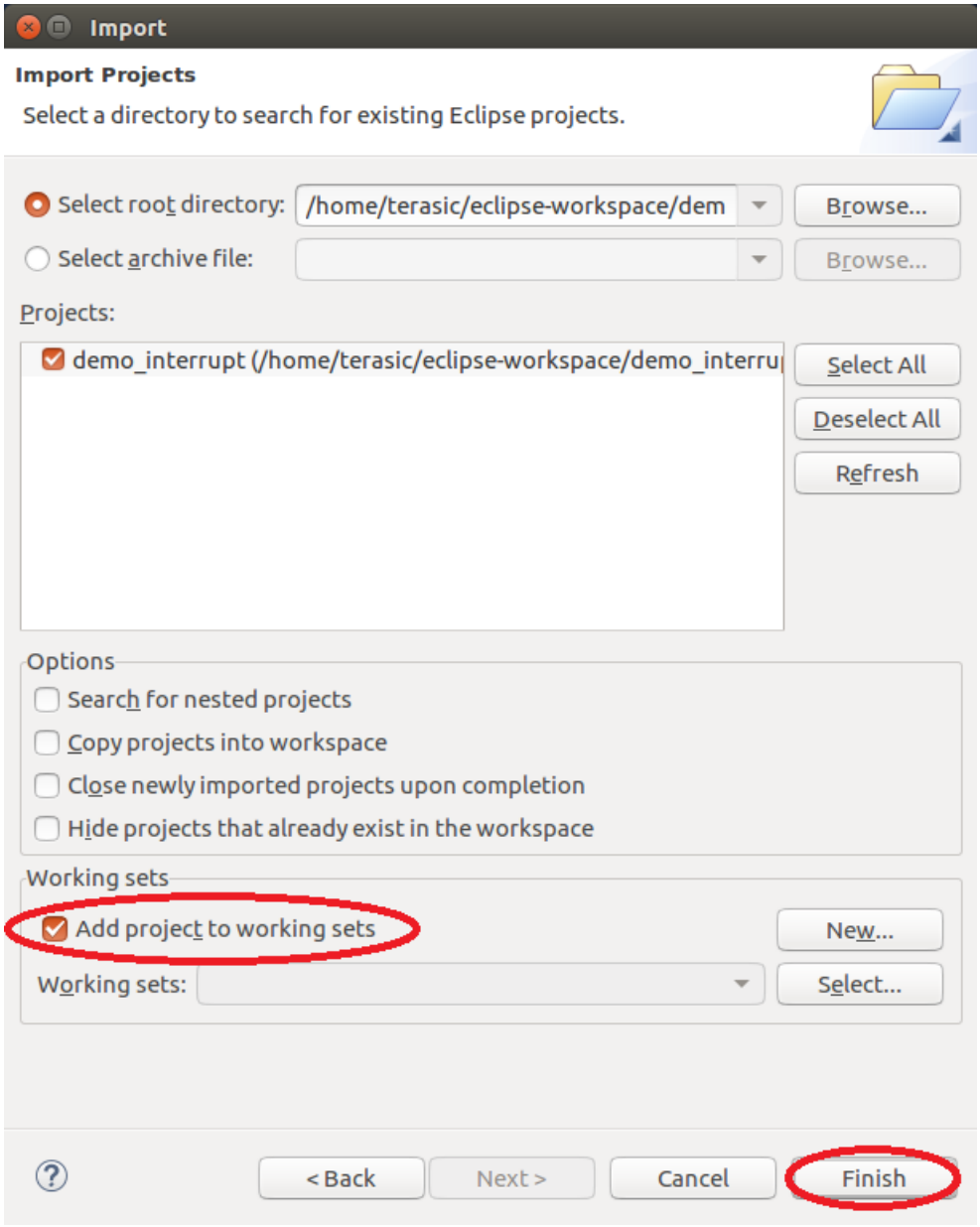

图4.2.9 添加 demo_interrupt 工程到工作空间

10. 导入后的工程界面如图 4.2.10 所示。

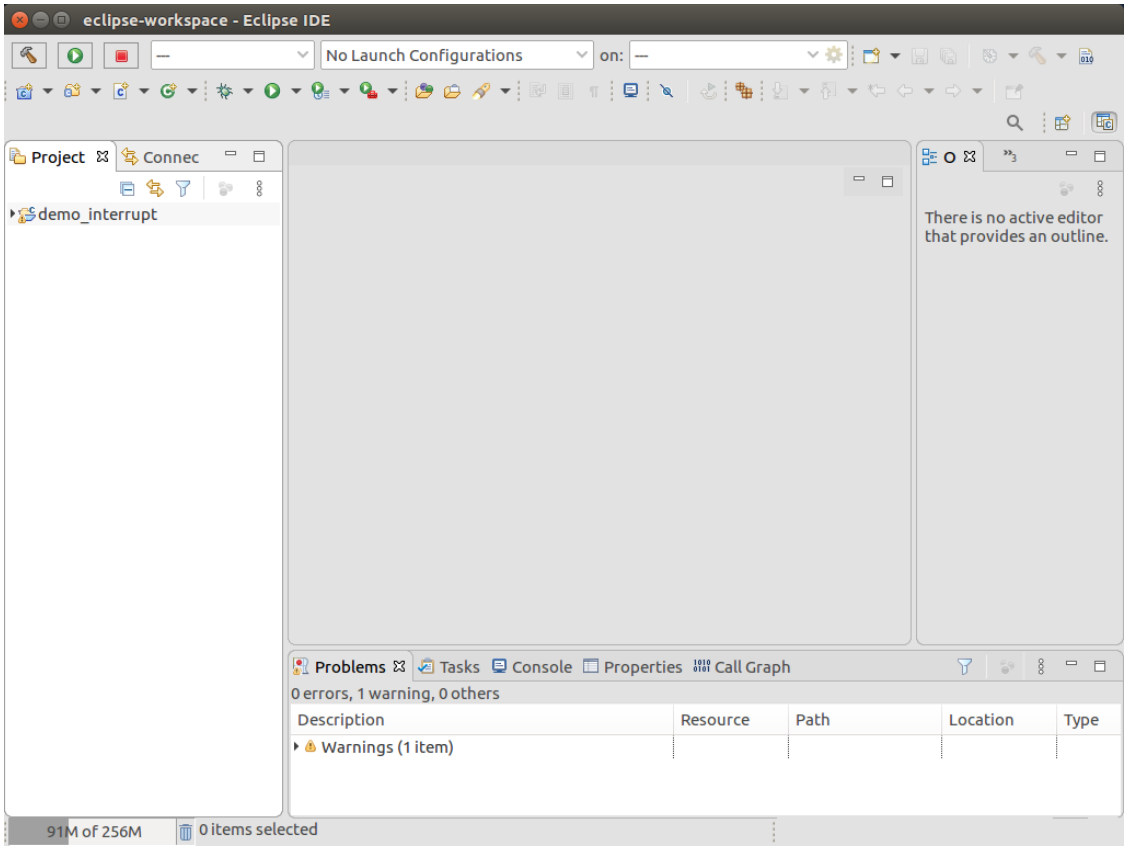

图4.2.10 导入后的工程界面

4.2.2 修改 main.c 文件

点击 demo_interrupt --> src --> bsp 下拉框，双击打开 main.c 文件，复制 4.1.1 节中的 main.c 文件中的代码替换掉当前 "main.c" 的代码并保存，如图 4.2.11 所示。

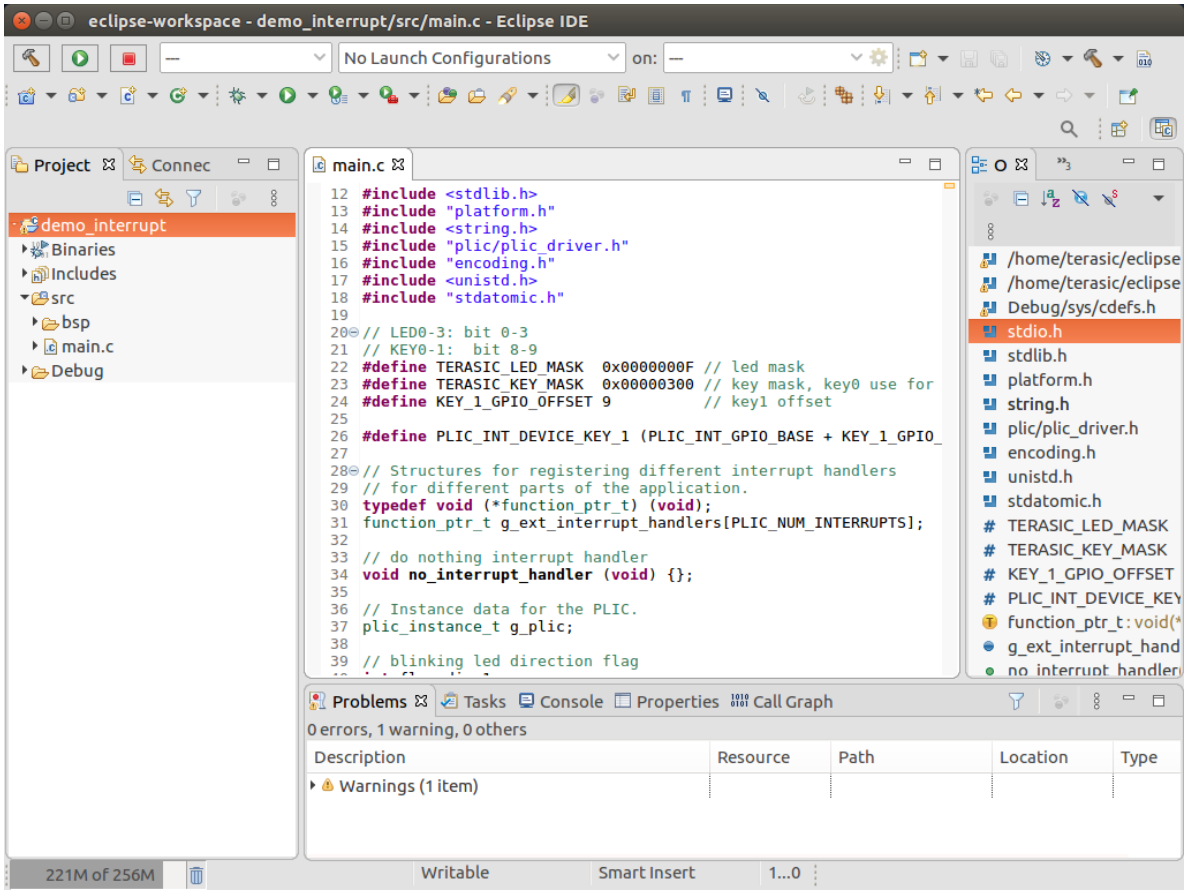

图4.2.11 修改 main.c 文件

4.2.3 编译 demo_interrupt工程

1. 在 Eclipse 主界面中，选中 demo_interrupt工程，右键点击 Clean Project；再次选中 demo_interrupt 工程，右键点击 Build Project，若 demo_interrupt 工程参照之前的步骤设置正确，则在这一步会编译成功，如图 4.2.12 所示。（注：需要右键点击 Refresh 才可以在 Debug 下拉项中看到生成的 .elf 文件）

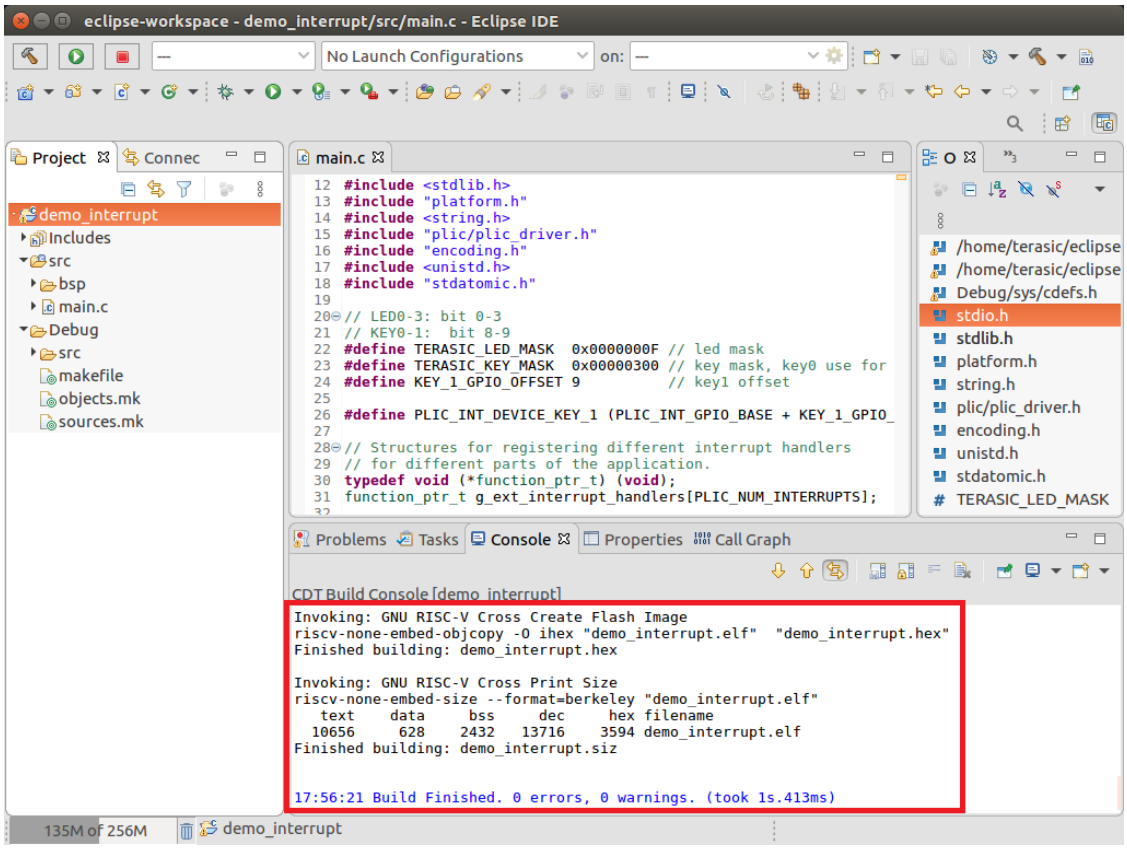

图4.2.12 编译成功

2. 右键 Debug 下拉选项中的 "blinking_LED.map"，点击 Delete，删除完成后如图 4.2.13 所示。

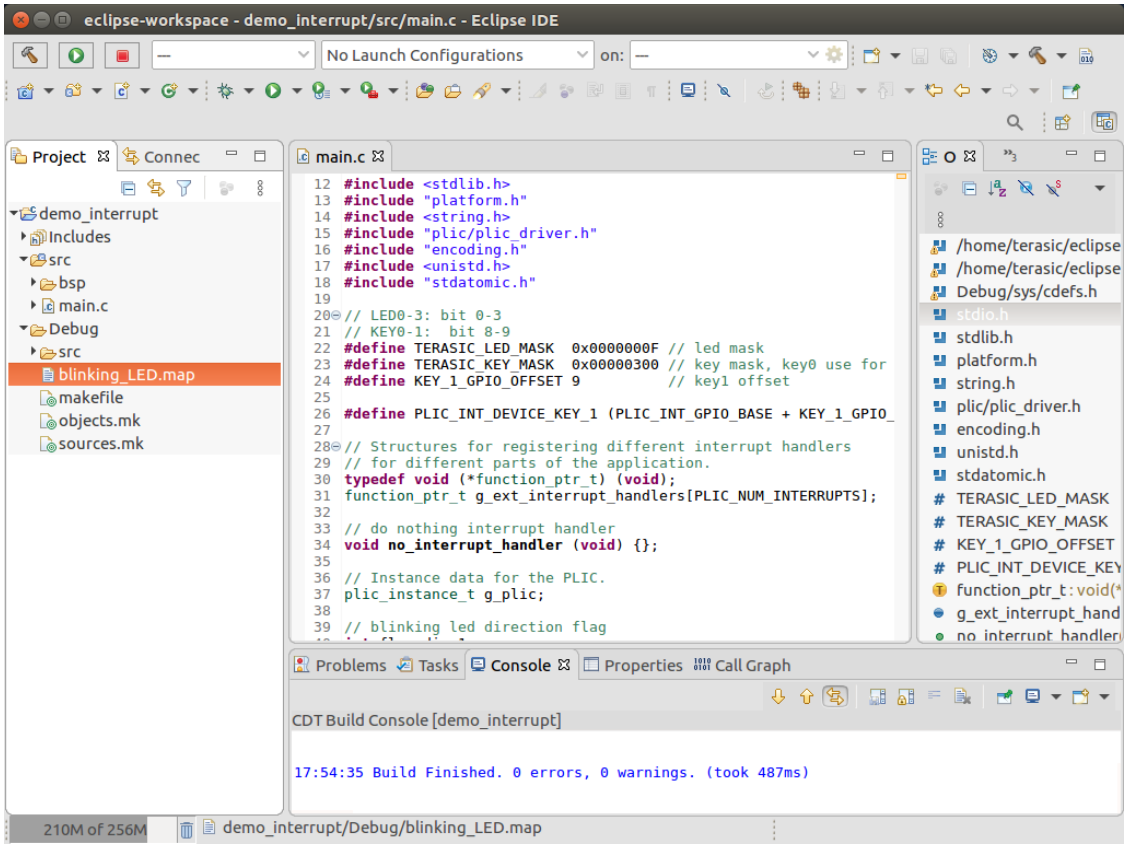

图4.2.13 删除 blinking_LED.map 文件

4.2.4 运行 demo_interrupt工程

1. 使用 USB Cable 将 T-Core 开发板与 PC 电脑进行连接来烧录应用程序。具体操作如下：
   - 关闭 T-Core 开发板电源后，将开发板上的 SW2: SW2.1=1，SW2.2=0，选择 RISC-V JTAG 链路。

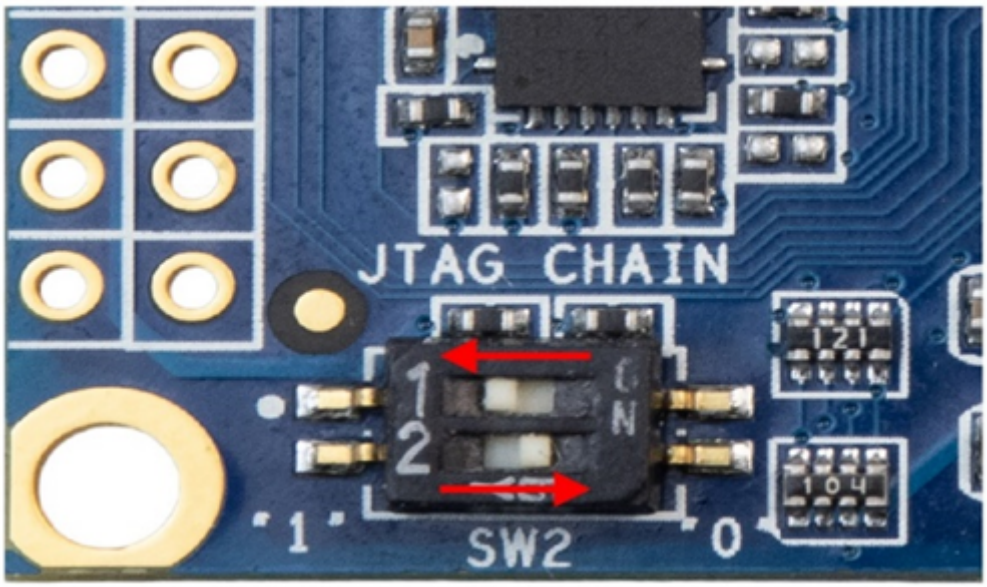

图4.2.14 设置 SW2 开关

   - 将 USB Blaster II 线缆的一端连接到开发板的 USB Blaster 接口（J2），另一端连接至 PC 主机的 USB 接口。
2. 选中 demo_interrupt 工程，右键点击 Properties，点击 C/C++ Build 下拉选择 Settings，点击 Tool Settings 选项卡下的 Optimization，修改 Optimization level 为 "Optimize(-O1)"，如图 4.3.15 所示。

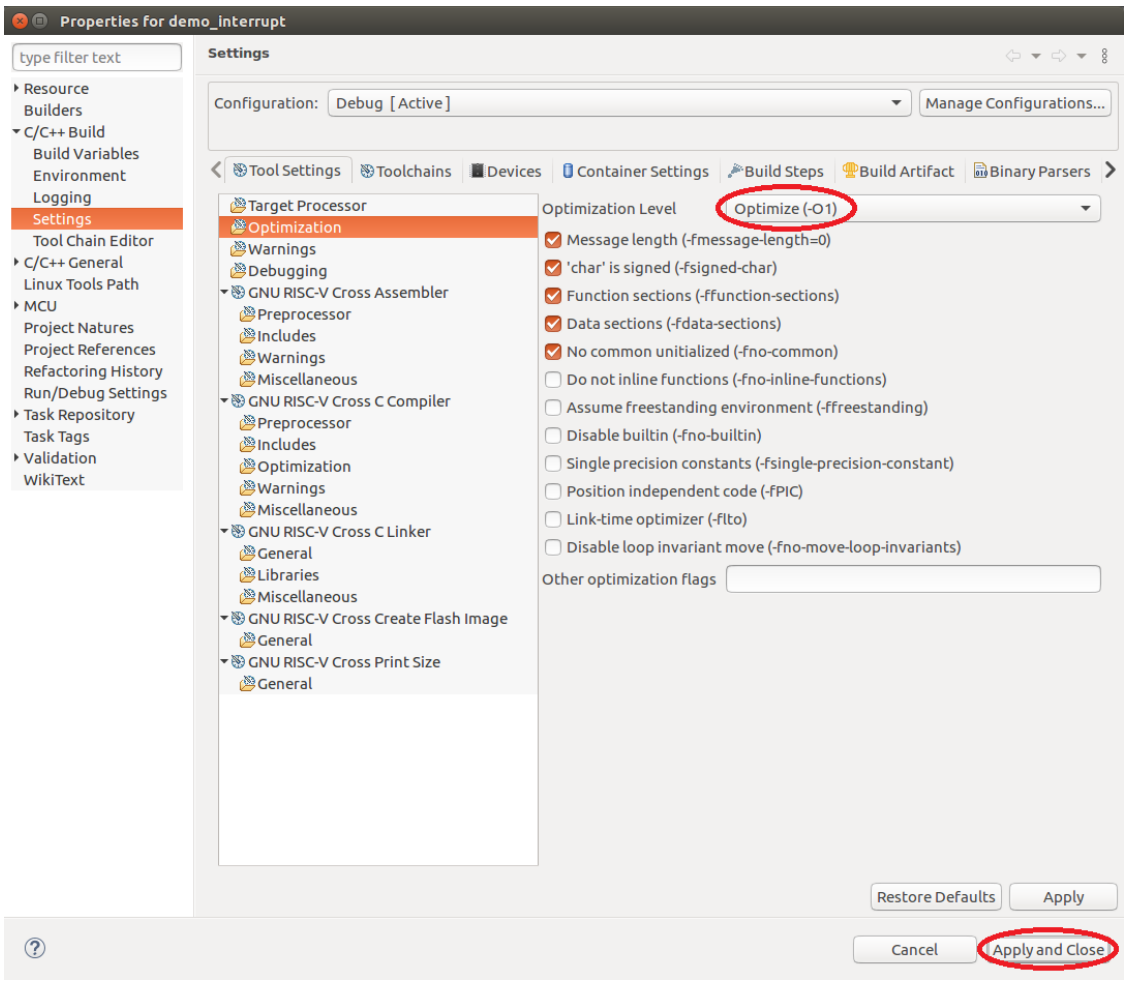

图4.3.15 修改 Optimization level

3. 选中 demo_interrupt 工程，右键点击 Run As -> Run Configurations...，双击 GDB OpenOCD Debugging 会出现如图 4.3.16 所示的 demo_interrupt Debug 界面，在 Config options 中添加 "-f /home/terasic/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/bsp/tcore-e203/env/openocd_tcore.cfg" 和 "-s /home/terasic/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/bsp/tcore-e203/env/"，在 Commands 中添加 "set arch riscv:rv32"，点击 Run 运行 demo_interrupt 工程。

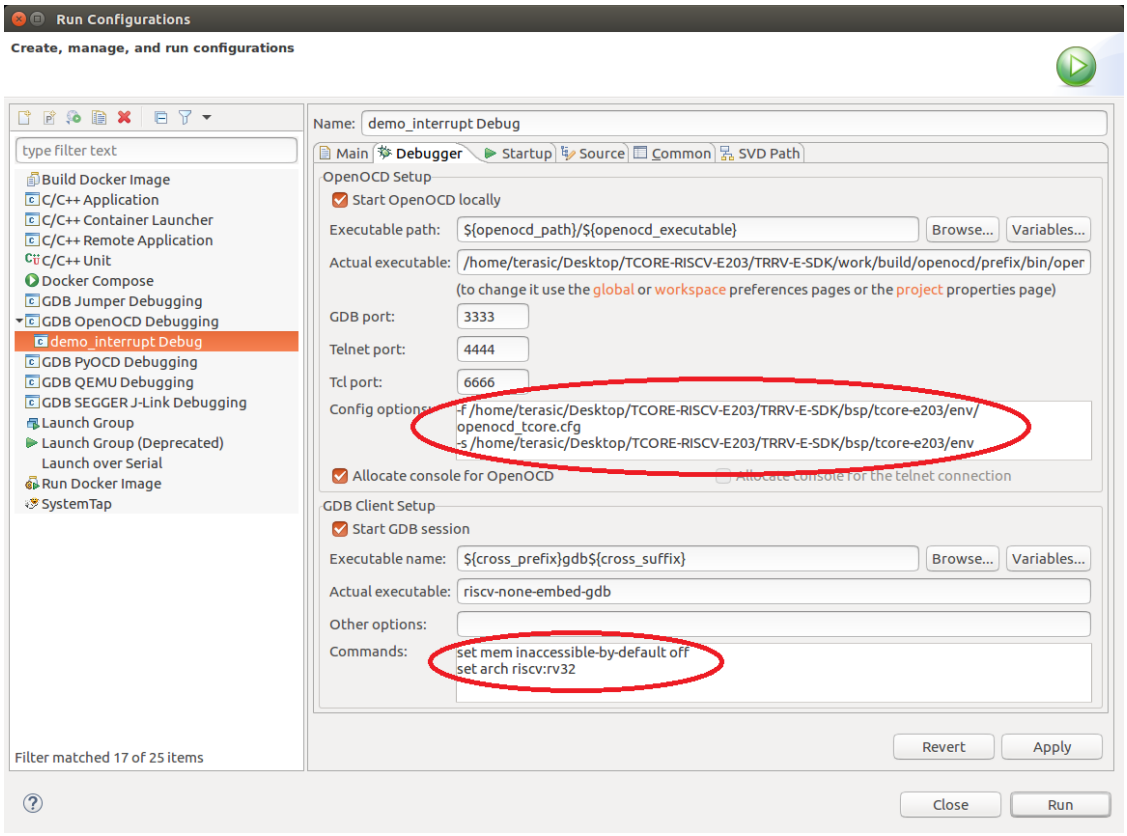

图4.2.16 运行配置

4. 程序下载成功后，如图 4.2.17 所示。

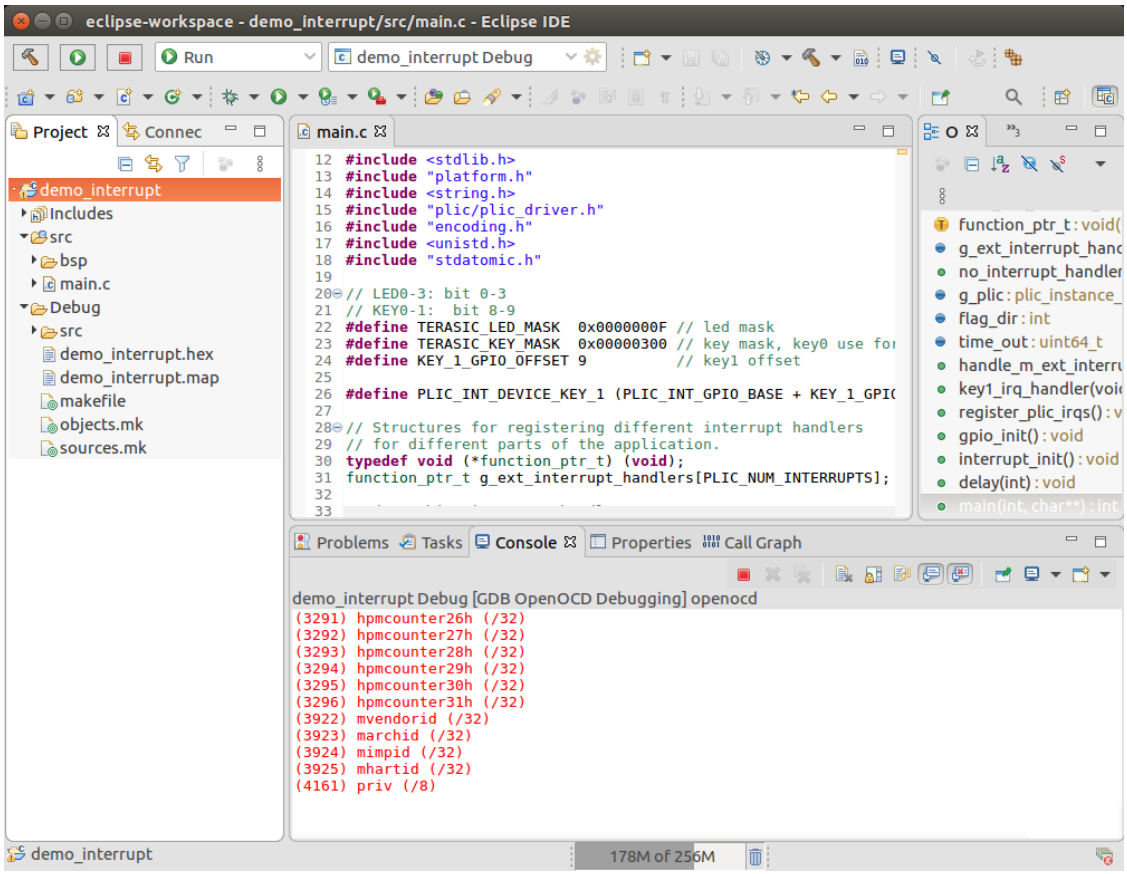

图4.2.17 运行 demo_interrupt 工程

4.2.5 运行结果

程序下载完成后，先按 KEY0 键复位。若 SW[1:0] 为 "00"，按下 KEY1，LED 从右向左呈现流水灯闪烁；若 SW[1:0] 为 "01"，再次按下 KEY1，LED 从左向右呈现流水灯闪烁；若 SW[1:0] 为 "10"，再次按下 KEY1，LED 从右向左循环点亮再从右向左循环熄灭；若 SW[1:0] 为 "11"，再次按下 KEY1，LED 从左向右循环点亮再从左向右循环熄灭。

附录

1. 修订历史

版本	时间	修改记录
V1.0	2020.08.01	初始版本

2. 版权声明

3. 获取帮助

如遇到任何问题，可通过以下方式联系我们：

电话：027-87745390

地址：武汉市东湖新技术开发区金融港四路18号光谷汇金中心7C

网址：www.terasic.com.cn

邮箱：support@terasic.com.cn

微信公众号：

订阅号

服务号