实战演练之FreeRTOS

一、概述

本教程将介绍如何在 T-Core 开发板上实现 RISC-V 设计——FreeRTOS：将 FreeRTOS 移植到 T-Core RISC-V，配置FreeRTOSConfig.h 文件，创建工程实现任务管理。

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 软件创建 FreeRTOS_10.3.1 工程、编译并运行 FreeRTOS_10.3.1 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 FreeRTOS 系统；
- 熟悉 FreeRTOS 系统的任务管理。

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-V1.2.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。）

    - 硬件相关的三个移植文件： port.c、portasm.S 和 portmacro.h

    （注：这三个文件可从 RISC-V on T-Core 第十二讲 下载。）

三、设计思路

3.1 FreeRTOS

嵌入式实时操作系统（Embedded Real-time Operation System，RTOS），是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其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作出快速响应，并控制所有实时任务协调一致运行的嵌入式操作系统。

FreeRTOS 是一个迷你的实时操作系统内核，作为一个轻量级的操作系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能、软件定时器、协程等，可基本满足较小系统的需要。

由于 RTOS 需占用一定的系统资源（尤其是 RAM 资源），只有 μC/OS-II、embOS、salvo、FreeRTOS 等少数实时操作系统能在小 RAM 单片机上运行。相对 μC/OS-II、embOS 等商业操作系统，FreeRTOS 操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行，目前的最新版本为 10.3.1 版。

3.2 FreeRTOS 系统配置

在使用 FreeRTOS 时需要根据自己的需求来配置 FreeRTOS，FreeRTOS 的系统配置文件为 FreeRTOSConfig.h，在此文件中可以完成 FreeRTOS 的裁剪和配置。

FreeRTOS 的配置是通过在 FreeRTOSConfig.h 中使用 "#define" 语句定义宏定义实现的。在 FreeRTOS 的官方 demo 中，每个工程都有一个 FreeRTOSConfig.h 文件，我们在进行配置时可以参考这个文件，甚至可以直接复制使用。由于篇幅的限制，在这里仅对本教程中使用到的宏定义进行介绍，其他定义请读者自行学习。

1. 命名以 **"INCLUDE_"** 开始的宏
    - INCLUDE_uxTaskPriorityGet

      如果要使用 uxTaskPriorityGet() 函数，需要将 INCLUDE_uxTaskPriorityGet 定义为 1。
    - INCLUDE_vTaskDelay

      如果要使用 vTaskDelay() 函数，需要将 INCLUDE_vTaskDelay 定义为 1。
2. 命名以 **"config"** 开始的宏
    - configSUPPORT_DYNAMIC_ALLOCATION

      若定义为 1，在创建 FreeRTOS 的内核对象时所需要的 RAM 就会从 FreeRTOS 的堆中动态的获取内存，若定义为 0，则所需的 RAM 需要用户自行提供。默认情况下宏 configSUPPORT_DYNAMIC_ALLOCATION 为 1。
    - configMAX_PRIORITIES

      设置任务的优先级数量，设置好以后任务就可以设置为从 0 到 configMAX_PRIORITIES-1 中的任意优先级，其中 0 是最低优先级，configMAX_PRIORITIES-1 是最高优先级。
    - configMINIMAL_STACK_SIZE

      设置空闲任务的最小任务堆栈大小，注意这里是以字为单位，而不是字节。
    - configTICK_RATE_HZ

      设置 FreeRTOS 的系统时钟节拍频率，单位为 Hz，此频率就是滴答定时器的中断频率， 需要使用此宏来配置滴答定时器的中断。
    - configTOTAL_HEAP_SIZE

      设置堆大小，如果使用了动态内存管理的话，FreeRTOS 在创建任务、信号量、队列等时，就会使用 heap_x.c（x 为 1-5）中的内存申请函数来申请内存。
    - configUSE_16_BIT_TICKS

      设置系统节拍计数器变量数据类型，系统节拍计数器变量类型为 TickType_t，当 configUSE_16_BIT_TICKS 为 1 时，TickType_t 为 16-bit，当 configUSE_16_BIT_TICKS 为 0 时，TickType_t 为 32-bit。
    - configUSE_IDLE_HOOK

      为 1 时使用空闲任务钩子函数，用户需要实现空闲任务钩子函数，函数原型如下：

```
1 void vApplicationIdleHook(void)
```

- configUSE_MALLOC_FAILED_HOOK

  为 1 时使用内存分配失败钩子函数，用户需要实现内存申请失败钩子函数，函数原型如下：

```
1 void vApplicationMallocFailedHook(void)
```

- configUSE_PREEMPTION

  为 1 时使用抢占式调度，为 0 时使用协作式调度。若使用抢占式调度，内核会在每个时钟节拍中断时进行任务切换，若使用协作式调度，会在如下地方进行任务切换：一个任务调用了函数 taskYIELD()；一个任务调用了可以使任务进入阻塞态的 API 函数；应用程序明确定义了在中断中执行上下文切换。

- configUSE_TICK_HOOK

  为 1 时使能时间片钩子函数，用户需要实现时间片钩子函数，函数原型如下：

```
1 void vApplicationTickHook(void)
```

3.3 FreeRTOS 任务管理

在 FreeRTOS 里面每一个实时的应用可以作为一个独立的任务，任何时间只有一个任务在运行，具体运行哪个任务由调度器决定，每个任务都有自己的堆栈，用于任务切换时，上下文的保存与恢复。

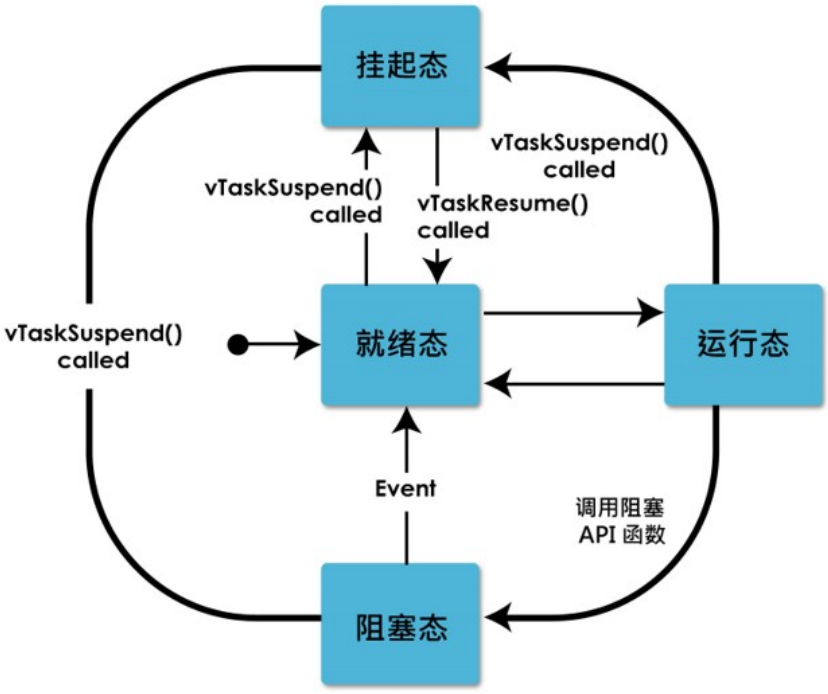

图3.2 FreeRTOS 任务管理

FreeRTOS 最基本的功能就是任务管理，FreeRTOS 的任务永远处于这四种状态：

- **运行态**：一个任务在运行时的状态。
- **就绪态**：已经准备就绪的、可以运行的任务，不过此时就绪态的任务还没有运行，因为此时还有同一优先级或优先级更高的任务在运行。
- **阻塞态**：任务正在等待某个外部事件或调用了延时函数而进入阻塞状态，任务在等待队列、信号量、事件组、通知、互斥量的时候也会进入阻塞状态，任务进入阻塞状态会有个超时时间，当超过

这个时间，即使等待的事件没有发生也会退出阻塞态。

- **挂起态**：像阻塞态一样，任务进入挂起态以后不能被调度器调用进入运行态，但是进入挂起态的任务没有超时时间，任务挂起调用 API 是 vTaskSuspend()，恢复就绪态调用 API vTaskResume()。

表 3.1 列出了 FreeRTOS 任务管理相关的 API。

表3.1 FreeRTOS 任务管理相关 API

函数	描述
xTaskCreate()	使用动态方法创建一个任务
xTaskCreateStatic()	使用静态方法创建一个任务
vTaskDelete()	删除一个任务
vTaskSuspend()	挂起一个任务
vTaskResume()	恢复一个任务的运行
vTaskResumeFromISR()	中断服务函数中恢复一个任务的运行
vTaskPrioritySet()	改变任务优先级

3.4 系统内核控制函数

FreeRTOS 中有一些函数只供系统内核使用，用户应用程序一般不允许使用，这些 API 函数就是系统内核控制函数。

表3.2 内核控制函数

函数	描述
taskYIELD()	任务切换
taskENTER_CRITICAL()	进入临界区，用于任务中
taskEXIT_CRITICAL()	退出临界区，用于任务中
taskENTER_CRITICAL_FROM_ISR()	进入临界区，用于中断服务函数中
taskEXIT_CRITICAL_FROM_ISR()	退出临界区，用于终端服务函数中
taskDISABLE_INTERRUPTS()	关闭中断
taskENABLE_INTERRUPTS()	打开中断
vTaskStartScheduler()	开启任务调度器
vTaskEndScheduler()	关闭任务调度器
xTaskSuspendAll()	挂起任务调度器
xTaskResumeAll()	恢复任务调度器
xTaskStepTick()	设置系统节拍值

3.5 FreeRTOS 调度算法

FreeRTOS 使用相关的调度算法来决定当前需要执行那个任务，任务调度算法是 FreeRTOS 操作系统的核心， FreeRTOS 支持时间片、协作式和抢占式三种任务调度。实际应用主要是抢占式调度和时间片调度，协作式调度用到的很少。

1. 时间片调度

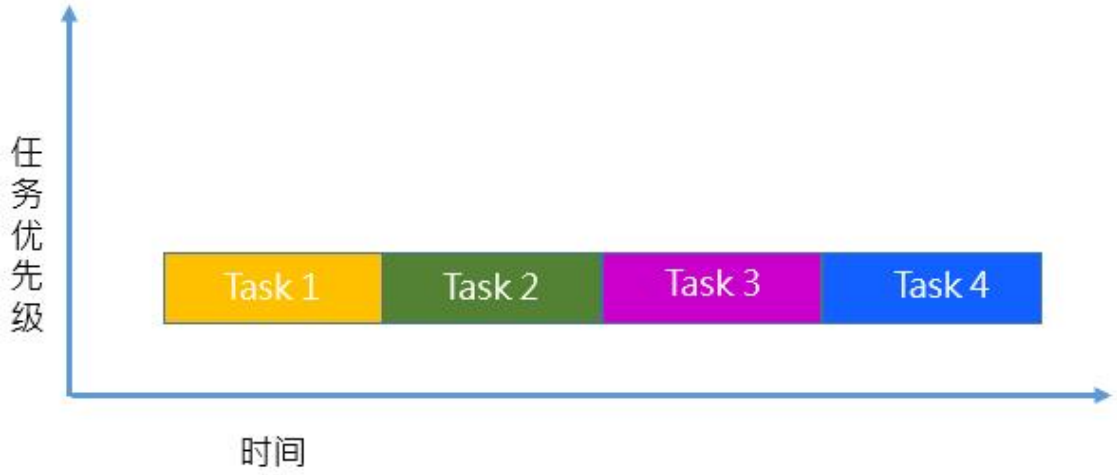

图3.3 时间片调度

每个任务都有相同的优先级，相同优先级的任务会顺序执行，遇到阻塞式的 API 函数，比如遇到 vTaskDelay() 函数，才会执行同优先级任务之间的任务切换。

2. 协作式调度

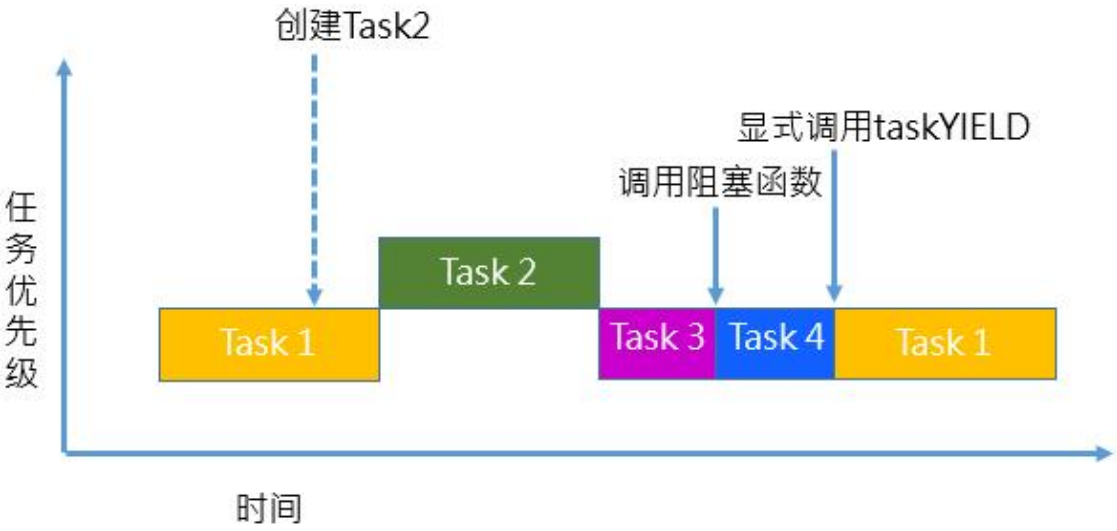

图3.4 协调式调度

协作式调度主要用于资源有限的设备，永远不会发生抢占，优先级高的任务就绪以后也需要等待当前的任务执行完才能进入运行态。

当然也有两个例外，比如当前任务调用了阻塞函数或者显式调用 taskYIELD() 函数。此时会立即发生任务切换。

3. 优先级抢占式调度

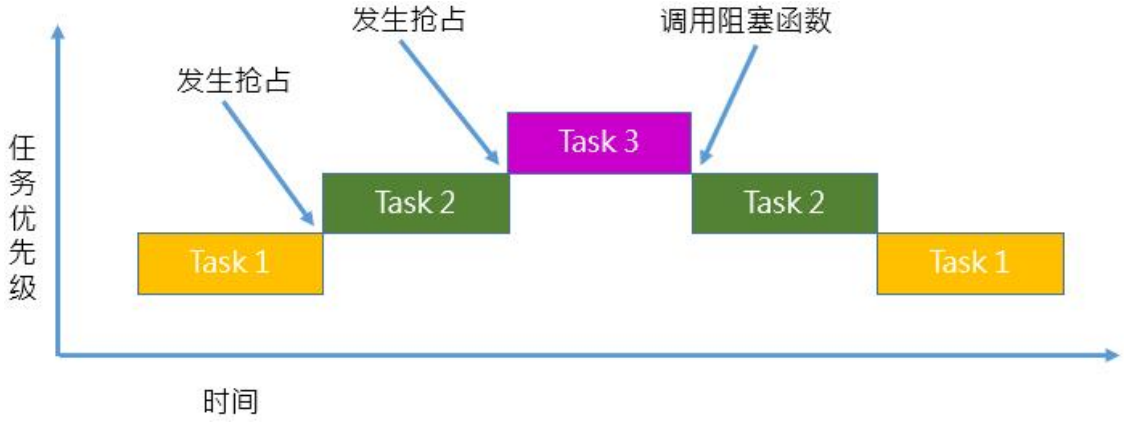

图3.5 优先级抢占式调度

在优先级抢占式调度中，每个任务都有不同的优先级，任务会一直运行直到被高优先级任务抢占或者遇到阻塞式的 API 函数。

比如 task1 正在执行的时候，比它优先级高的 task2 处于就绪态，就会切换 task2 到运行态，如果后面 task2 还没有执行完，又有优先级更高的 task3 处于就绪态，那么此时调度器会切换 task3 进入运行态，如果 task3 调用了阻塞函数， 这时就会切换 task2 到运行态，恢复 task2 的执行。

在有多个不同优先级和多个相同优先级的任务系统，可以时间片调度和抢占式调度混合使用。

3.6 空闲任务

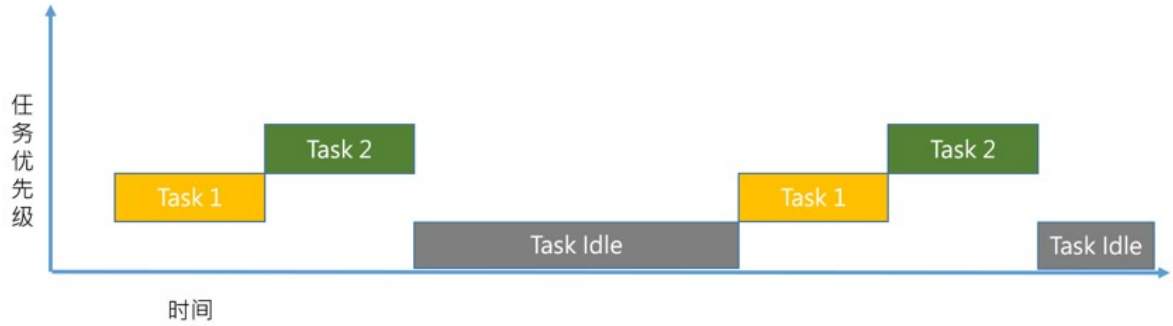

图3.6 空闲任务

FreeRTOS 的调度器启动后，会自动创建一个空闲任务，这样就可以确保至少有一个任务处于运行态。但是这个空闲任务使用最低优先级，如果应用中有其他高优先级任务处于就绪态的话，这个空闲任务就不会跟高优先级的任务抢占 CPU 资源。

空闲任务还有另外一个重要的职责，如果某个任务要调用函数 vTaskDelete() 删除本身，那么这个任务的任务控制块 TCB 和任务堆栈等这些由 FreeRTOS 系统自动分配的内存需要在空闲任务中释放掉；如果删除的是别的任务，那么相应的内存就会被直接释放掉，不需要在空闲任务中释放。

可以根据实际情况减少空闲任务使用 CPU 的时间（比如，当 CPU 运行空闲任务的时候使处理器进入低功耗模式）。

用户可以创建与空闲任务优先级相同的应用任务，当宏 configIDLE_SHOULD_YIELD 为 1 时，应用任务就可以使用空闲任务的时间片，也就是说空闲任务会让出时间片给同优先级的应用任务。

1. 空闲任务函数的创建

   当调用函数 vTaskStartScheduler() 启动任务调度器的时候，此函数就会自动创建空闲任务，代码如下：

```
1 void vTaskStartScheduler(void)
2 {
```

```
    BaseType_t xReturn;

    // 创建空闲任务
    #if(configSUPPORT_STATIC_ALLOCATION == 1)       // 使用静态方法创建空闲任务
    {
        StaticTask_t *pxIdleTaskTCBBuffer = NULL;
        StaticTask_t *pxIdleTaskStackBuffer = NULL;
        uint32_t ulIdleTaskStackSize;

        vApplicationGetIdleTaskMemory(&pxIdleTaskTCBBuffer,
                                      &pxIdleTaskStackBuffer,
                                      &ulIdleTaskStackSize);
        xIdleTaskHandle = xTaskVreateStatic(prvIdleTask,
                                            "IDLE",
                                            ulIdleTaskStackSize,
                                            (void*)NULL,
                                            (tskIDLE_PRIORITY|portPRIVILEGE_BIT),
                                            pxIdleTaskStackBuffer,
                                            pxIdleTaskTCBBuffer);
        if(xIdleTaskHandle != NULL)
        {
            xReturn = pdPASS;
        }
        else
        {
            xReturn = pdFAIL;
        }
    }

    /* 使用动态方法创建空闲任务，空闲任务函数为 prvIdleTask(),
    任务堆栈大小为 configMINIMAL_STACK_SIZE，任务堆栈大小可以在 FreeRTOSConfig.h 中修改。
    任务优先级为 tskIDLE_PRIORITY，宏 tskIDLE_PRIORITY 为 0，说明空闲任务优先级最低。
    用户不能随意修改空闲任务的优先级 */
    #else
    {
        xReturn = xTaskCreate(prvIdleTask,
                              "IDLE",
                              configMINIMAL_STATIC_SIZE,
                              (void*)NULL,
                              (tskIDLE_PRIORITY|portPRIVILEGE_BIT),
                              &xIDleTaskHandle);
    }
    #endif /*configSUPPORT_STATIC_ALLOCATION*/

    /*************************************************/
    /******************其他代码省略*******************/
    /*************************************************/
}
```

2. 空闲任务函数

空闲任务函数为 prvIdleTask()，但是实际上是找不到这个函数的，因为它是通过宏定义来实现的，在文件 portmacro.h 中有如下宏定义：

```
#define portTASK_FUNCTION(vFunction,pvParameters) void vFunction(void *pvParameters)
```

空闲任务的任务函数 portTASK_FUNCTION() 在 tasks.c 文件中定义。

3.7 空闲任务钩子函数

1. 钩子函数

FreeRTOS 中有多个钩子函数，钩子函数类似回调函数，当某个功能（函数）执行的时候就会调用钩子函数，钩子函数的具体内容需由用户自行编写。钩子函数是一个可选功能，可以通过宏定义来选择使用哪个钩子函数，可选的钩子函数如表 3.3 所示。

表3.3 钩子函数使能宏

宏定义	描述
configUSE_IDLE_HOOK	空闲任务钩子函数，空闲任务会调用此钩子函数
configUSE_TICK_HOOK	时间片钩子函数，xTaskIncrementTick() 会调用此钩子函数。此钩子函数最终会被节拍中断服务函数调用，对于 RISCV 来说就是滴答定时器中断服务函数
configUSE_MALLOC_FAILED_HOOK	内存申请失败钩子函数，当使用函数 pvPortMalloc() 申请内存失败的时候就会调用此钩子函数
configUSE_DAEMON_TASK_STARTUP_HOOK	守护（Daemon）任务启动钩子函数，守护任务也就是定时器服务函数

2. 空闲任务钩子函数

在每个空闲任务运行周期都会调用空闲任务钩子函数，如果想在空闲任务优先级下处理某个任务，有以下两种方式：

- 在空闲任务钩子函数中处理任务

  不管什么时候都要保证系统中至少有一个任务可以运行，因此不能在空闲任务钩子函数中调用任何会阻塞空闲任务的 API 函数，比如 vTaskDelay()，或者其他带有阻塞时间的信号量或队列操作函数。

- 创建一个与空闲任务优先级相同的任务

  创建一个任务是比较好的解决方法，但是这种方法会消耗更多的 RAM。要使用空闲任务钩子函数首先要在 FreeRTOSConfig.h 中将宏 configUSE_IDLE_HOOK 改为 1，然后编写空闲任务钩子函数 vApplicationIdleHook()。通常在空闲任务钩子函数中将处理器设置为低功耗模式来节省功耗。

四、操作步骤

4.1 FreeRTOS 移植

1. 进入 Home 文件夹，右键打开 Terminal 终端，使用如下命令从 github 下载 FreeRTOS 源码。（注：读者可以自行选择路径下载 FreeRTOS 源码）

```
git clone https://github.com/FreeRTOS/FreeRTOS.git --recurse-submodules
```

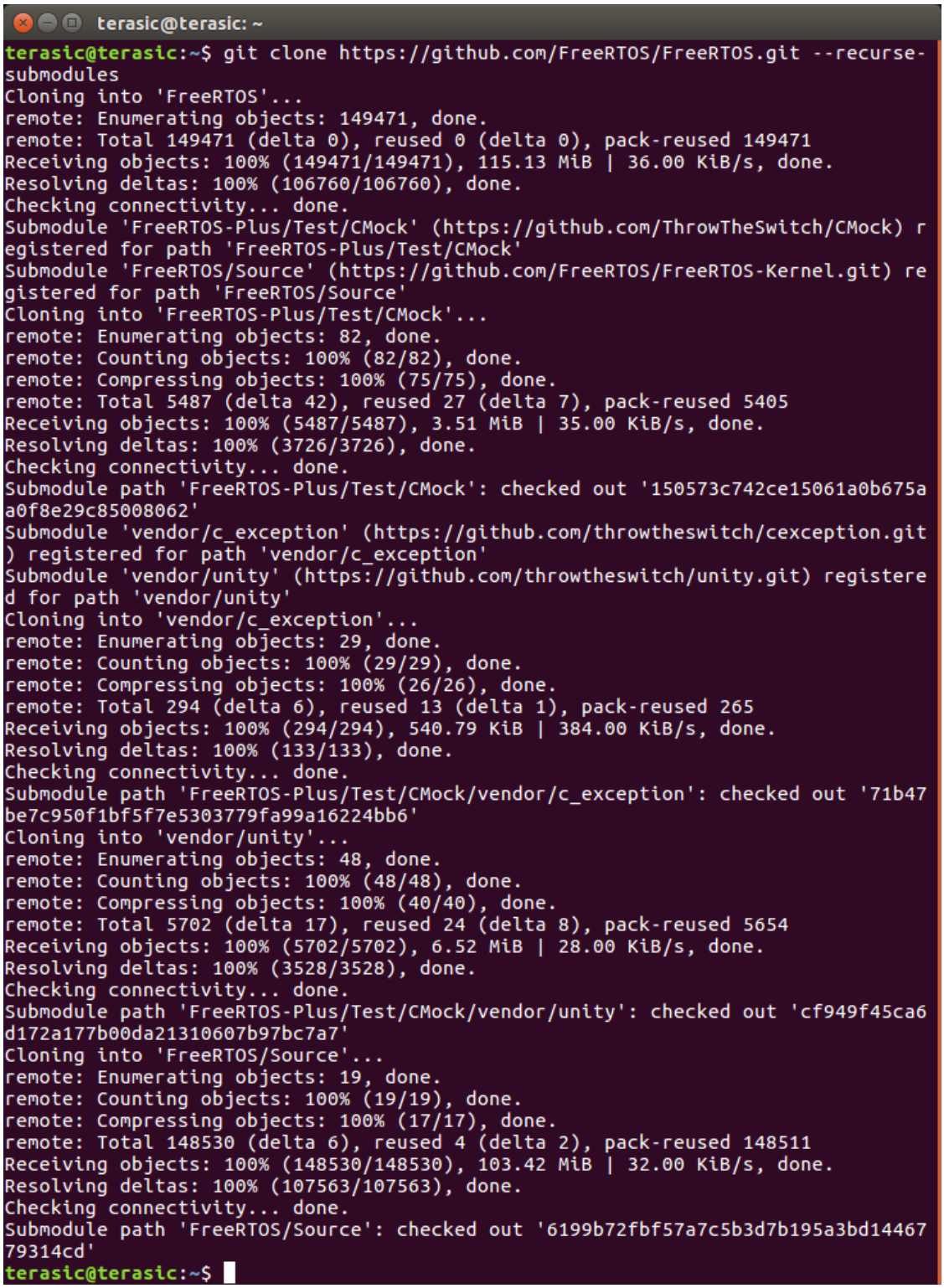

图4.1.1 下载 FreeRTOS 源码

下载完成后会生成一个名为 FreeRTOS 的文件夹。

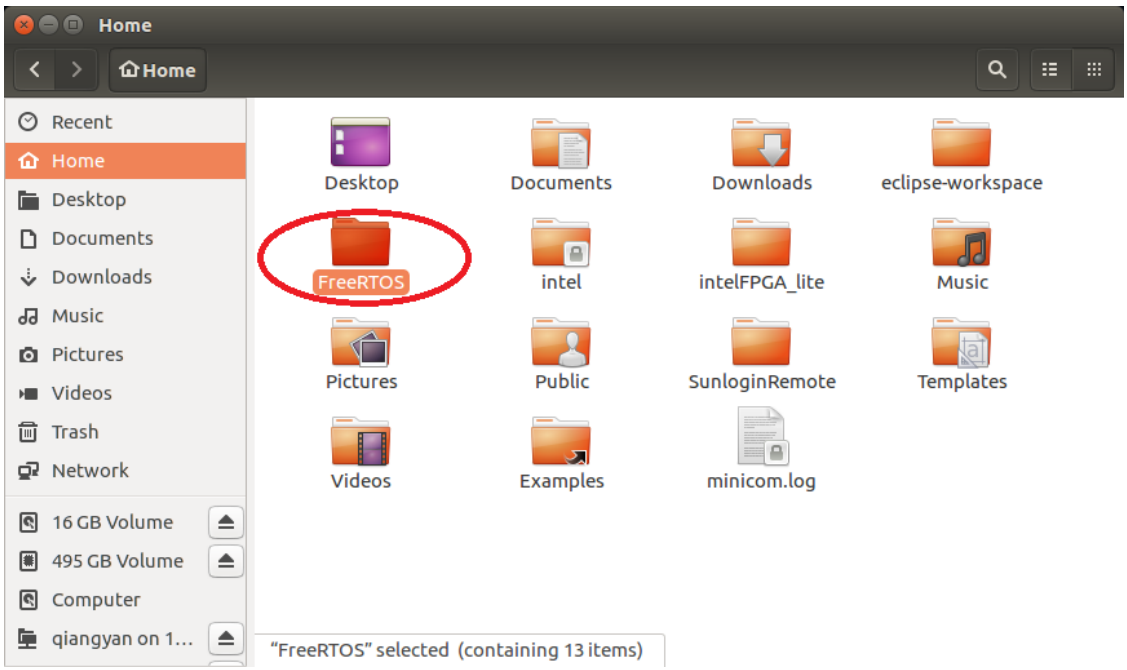

图4.1.2 生成的 FreeRTOS 文件夹

2. 进入 FreeRTOS 文件夹，使用如下命令切换版本为 FreeRTOS-10.3.1。

```
git checkout V10.3.1
```

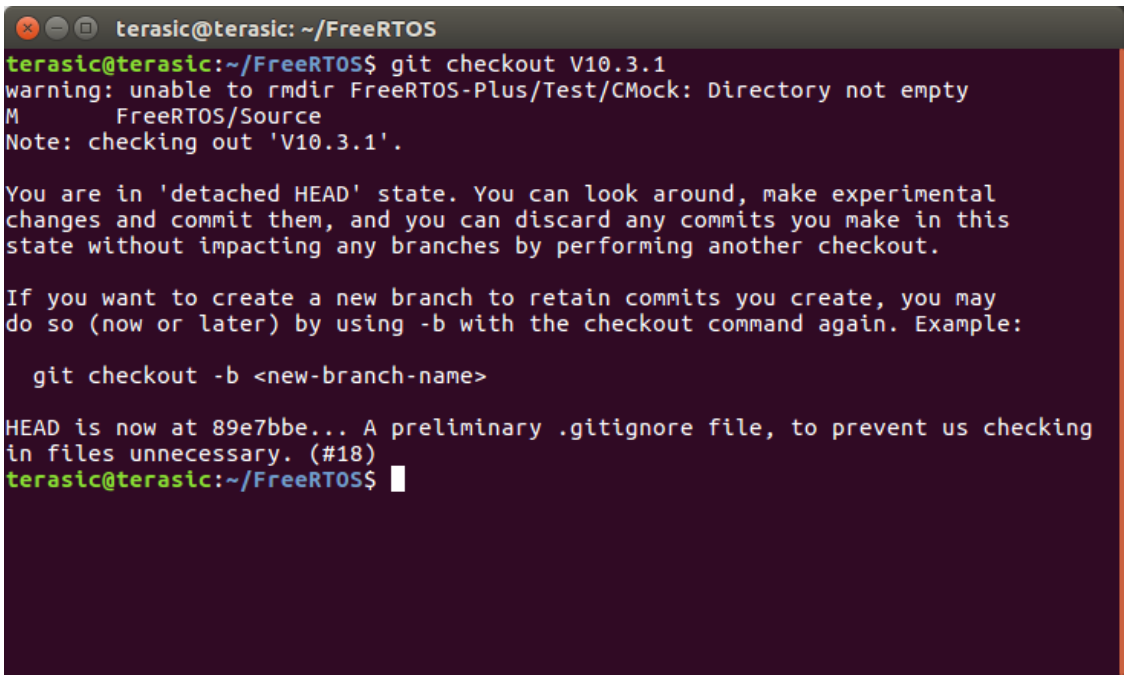

图4.1.3 切换版本为 FreeRTOS-10.3.1

切换完成后，输入如下命令获取文件（文件夹）在工作区、暂存区的状态，确认版本是否切换成功。

```
git status
```

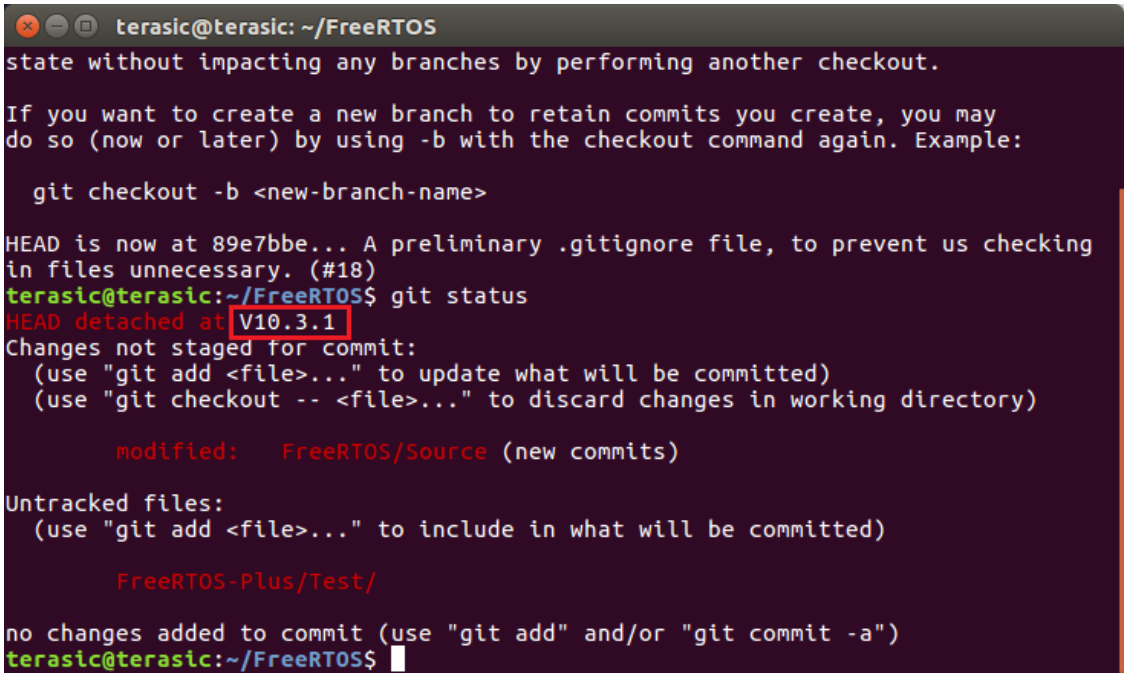

图4.1.4 查看版本号

3. 复制 FreeRTOS 目录下的 FreeRTOS 文件夹到 "~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software"，重命名为 FreeRTOS_10.3.1。

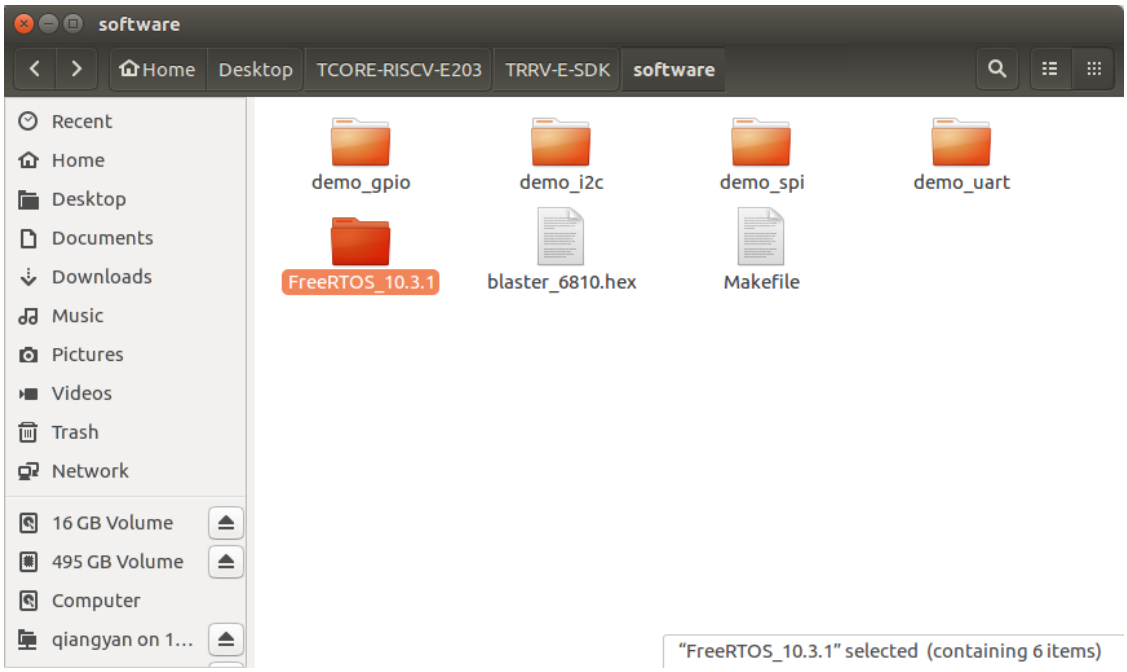

图4.1.5 复制 FreeRTOS 文件夹到 software 文件夹

4. 删除 Demo 目录下所有内容，并新建文件夹，命名为 RISCV_TCORE_GCC。

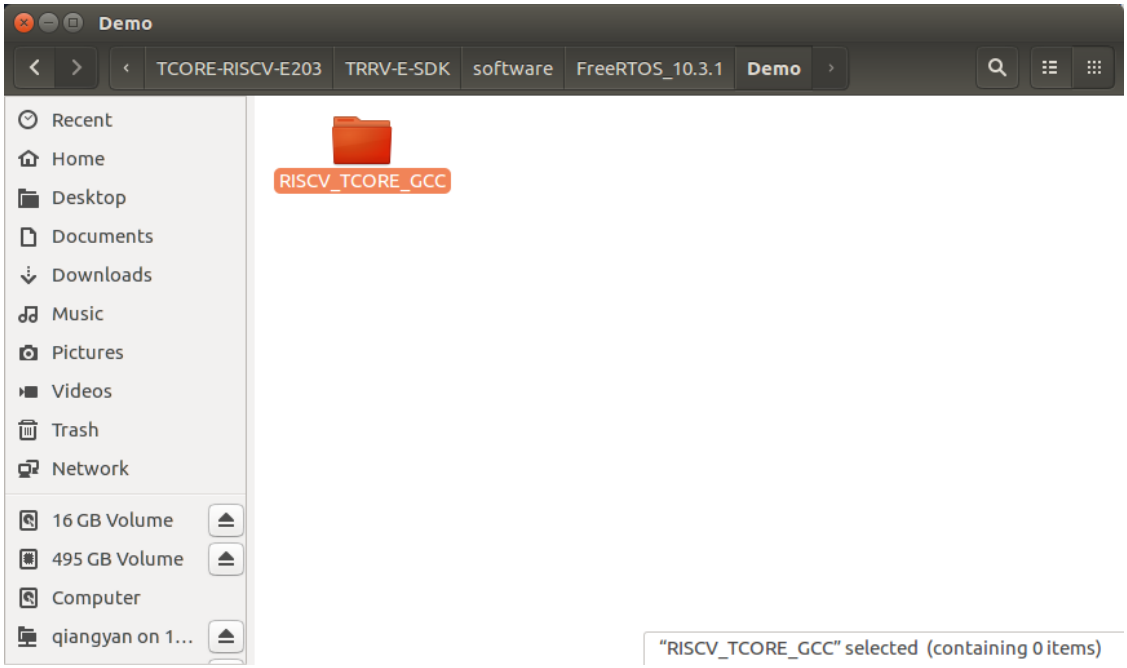

图4.1.6 新建 RISCV_TCORE_GCC 文件夹

5. 删除 Source/portable 目录下除 MemMang 目录的其他所有内容。

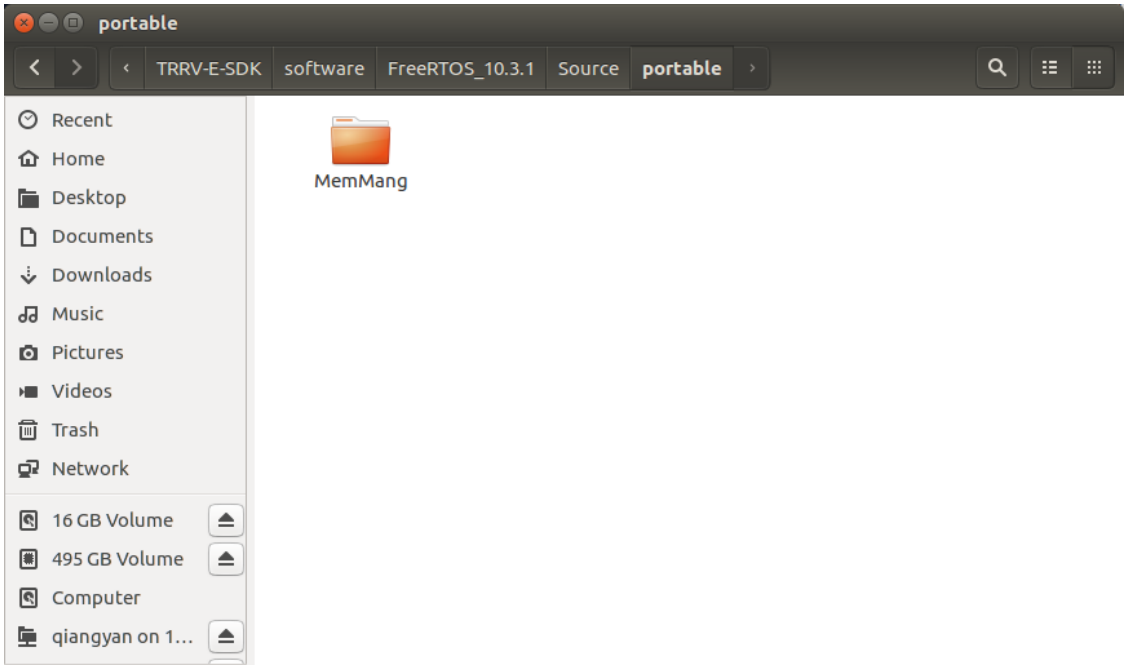

图4.1.7 portable 目录

6. 在 Source/portable 目录下新建 GCC 目录及 GCC/T-Core 子目录

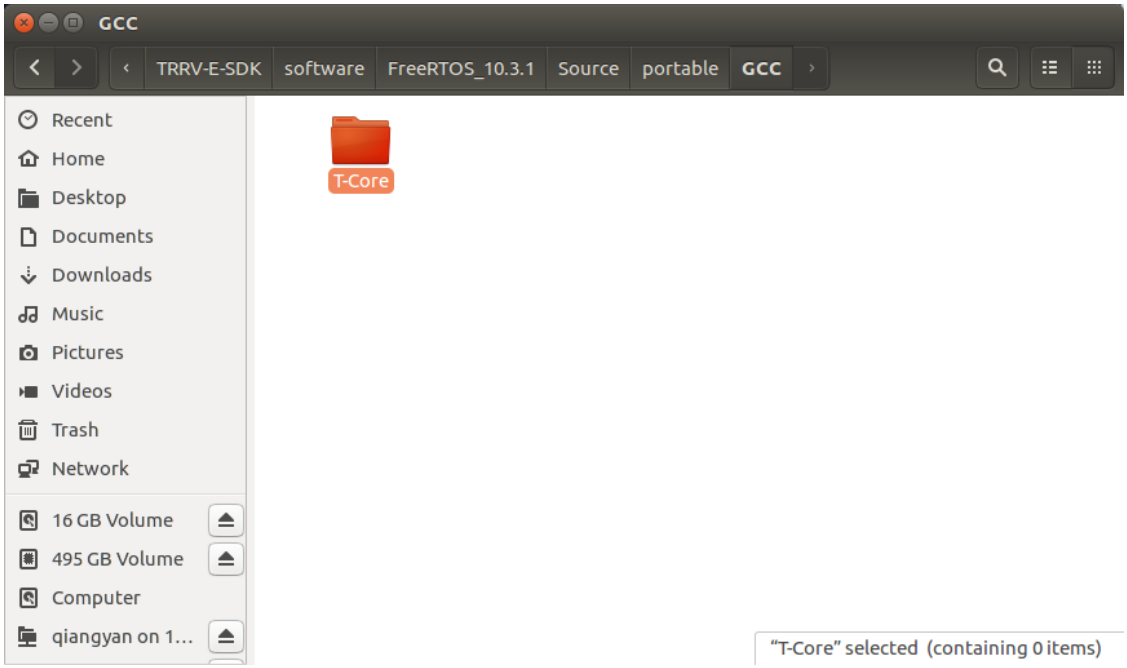

图4.1.8 新建 GCC 和 T-Core 文件夹

7. 将移植好的三个文件 port.c、portasm.S 和 portmacro.h 复制到 Source/portable/GCC/T-Core 目录下。

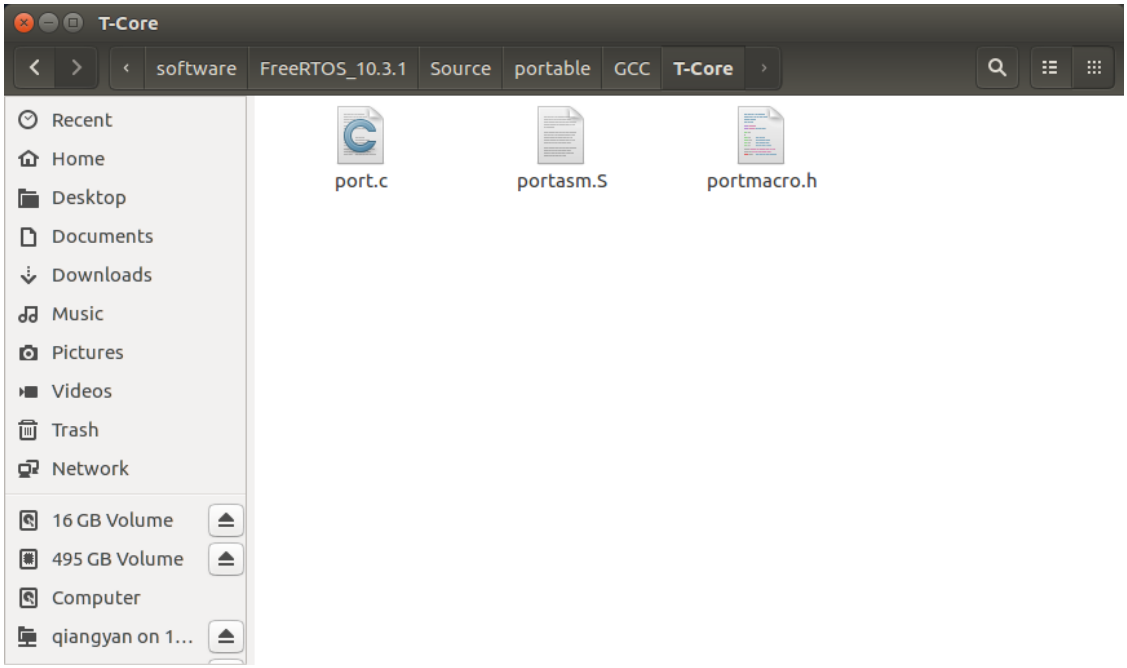

图4.1.9 复制 port.c、portasm.S 和 portmacro.h 文件

4.2 使用 Makefile 编译和下载应用程序

4.2.1 构建工程

1. **FreeRTOS 系统配置**

   在 "RISCV_TCORE_GCC" 文件夹下创建一个空白文本文档并命名为 "FreeRTOSConfig.h"。

   定义 configRTC_CLOCK_HZ，设置 RTC 的时钟频率为 32.768kHz，用作定时器中断的时钟。

```
1 #define configRTC_CLOCK_HZ                    32768
```

   定义 "INCLUDE_" 开始的宏。

```
/* Optional functions - most linkers will remove unused functions
anyway. */
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelay                  1
```

定义 "config" 开始的宏。

```
/* Here is a good place to include header files that are required
across your application. */
#define configUSE_16_BIT_TICKS              0
#define configUSE_PREEMPTION                1
#define configMAX_PRIORITIES                3
#define configTICK_RATE_HZ                  80
#define configMINIMAL_STACK_SIZE            450

/* Memory allocation related definitions. */
#define configSUPPORT_DYNAMIC_ALLOCATION    1
#define configTOTAL_HEAP_SIZE               11*1024

/* Hook function related definitions. */
#define configUSE_IDLE_HOOK                 1
#define configUSE_TICK_HOOK                 1
#define configISE_MALLOC_FAILED_HOOK        1
```

2. 创建程序文件（.c文件）

在 "RISCV_TCORE_GCC" 文件夹下创建一个 "main.c" 的文本文档。

包含如下头文件。

```
/* Kernel includes. */
#include "FreeRTOS.h" /* Must come first. */
#include "task.h"     /* RTOS task related API prototypes. */
#include "timers.h"   /* Software timer related API prototypes. */
#include "semphr.h"   /* Semaphore related API prototypes. */

/* TODO Add any manufacture supplied header files can be included here.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "platform.h"
#include "encoding.h"
#include "plic/plic_driver.h"
```

定义 LED 掩码及每个 LED 对应的偏移位数。

```
#define TERASIC_LED_MASK  0x0000000F // led mask
#define LED0_GPIO_OFFSET  0
#define LED1_GPIO_OFFSET  1
#define LED2_GPIO_OFFSET  2
#define LED3_GPIO_OFFSET  3
```

创建 Task1、Task2 任务句柄 Task1_Handler 和 Task2_Handler，FreeRTOS 使用 TaskHandle_t 标识任务句柄。

```
TaskHandle_t Task1_Handler;
TaskHandle_t Task2_Handler;
```

定义 Task1 实现，当 Task1 处于运行态时，打印 "do Task1: priority is [优先级数]"。同时，LED0 点亮一段时间后熄灭，一段时间后再次点亮，循环交替。这里采用 delay() 函数，为非阻塞的延时函数，执行非阻塞的函数不会阻塞当前任务的运行。

```
static void prvTask1( void *pvParameters )
{
    for( ;; ){
        printf("do Task1: priority is %d\n",uxTaskPriorityGet(Task1_Handler)); // print task priority
        GPIO_REG(GPIO_OUTPUT_VAL)  ^=   (0x1 << LED0_GPIO_OFFSET) ; // change led value
        delay(500); // non-block delay
        GPIO_REG(GPIO_OUTPUT_VAL)  ^=   (0x1 << LED0_GPIO_OFFSET) ; // change led value again
        delay(500); // non-block delay
    }
}
```

定义 Task2 实现，当 Task2 处于运行态时，打印 "do Task2: priority is [优先级数]"。同时，LED1 点亮一段时间后熄灭，一段时间后再次点亮，循环交替。且定义了 task2_count 进行计数，每执行一次 Task2，就打印一次 "Task2 count:[执行次数]"。当 Task2 执行 5 次后，调用 vTaskDelay() 函数，为阻塞的延时函数，执行阻塞的函数会阻塞当前任务的运行，在这里，Task2 会被阻塞 3s。

```
static void prvTask2( void *pvParameters )
{
    int task2_count=0;
    for( ;; ){
        task2_count++;
        printf("do Task2: priority is %d\n",uxTaskPriorityGet(Task2_Handler)); // print task priority
        GPIO_REG(GPIO_OUTPUT_VAL)  ^=   (0x1 << LED1_GPIO_OFFSET) ; // change led value
        delay(500); // non-block delay
        GPIO_REG(GPIO_OUTPUT_VAL)  ^=   (0x1 << LED1_GPIO_OFFSET) ; // change led value again
        delay(500); // non-block delay
        printf("Task2 count:%d\n",task2_count); // print count value

        if(task2_count==5){ // when count=5,block delay task2,so task1 run
            vTaskDelay(3000/portTICK_PERIOD_MS);
            task2_count=0; // set task2_count 0
        }
    }
}
```

配置硬件，在本教程中，只使用了 LED，在 led_init() 函数中，配置 GPIO 输出使能，LED 初始状态为全灭。

```
void led_init(){
    GPIO_REG(GPIO_OUTPUT_EN)   |=  TERASIC_LED_MASK;
    GPIO_REG(GPIO_OUTPUT_VAL)  &= ~TERASIC_LED_MASK;
}

static void prvSetupHardware( void ){
    led_init();
}
```

定义 delay 延时函数，用于不精确的计数延时。

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

在 main 函数中，首先配置硬件系统。

```
int main(void)
{
    /* Configure the system ready to run the demo. */
    prvSetupHardware();
```

创建 Task1，任务实现函数为 prvTask1()，命名为 "Task1"，任务堆栈大小为 configMINIMAL_STACK_SIZE（任务堆栈大小可以在 FreeRTOSConfig.h 中修改）。任务优先级为 tskIDLE_PRIORITY+1，宏 tskIDLE_PRIORITY 为 0，说明 Task1 任务优先级为 1，Task1 任务句柄为 Task1_Handler。

```
    /* Create the task1 as described in the comments at the top of this file. */
    xTaskCreate(    prvTask1,                   // The function that implements the task.
                    ( const char * ) "Task1",   // Text name for the task, just to help debugging.
                    configMINIMAL_STACK_SIZE,   // The size (in words) of the stack that should be created for the task.
                    NULL,                       // A parameter that can be passed into the task.  Not used in this simple demo.
                    tskIDLE_PRIORITY+1,         // The priority to assign to the task.  tskIDLE_PRIORITY (which is 0) is the lowest priority.
                    &Task1_Handler );           // Used to obtain a handle to the created task.

```

创建 Task2，任务实现函数为 prvTask2()，命名为 "Task2"，任务堆栈大小为 configMINIMAL_STACK_SIZE（任务堆栈大小可以在 FreeRTOSConfig.h 中修改）。任务优先级为 tskIDLE_PRIORITY+2，宏 tskIDLE_PRIORITY 为 0，说明 Task2 任务优先级为 2，Task2 任务句柄为 Task2_Handler。

```
    // Create the task2 in exactly the same way.
    xTaskCreate(    prvTask2,
                    ( const char * ) "Task2",
                    configMINIMAL_STACK_SIZE,
                    NULL,
                    tskIDLE_PRIORITY+2,
                    &Task2_Handler );
```

开启任务调度器。

```
    /* Start the tasks and timer running. */
    vTaskStartScheduler();
}
```

定义空闲任务钩子函数。

```
void vApplicationTickHook( void )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    static uint32_t ulCount = 0;

    ulCount++;
    if( ulCount >= 500UL )
    {
        ulCount = 0UL;
        GPIO_REG(GPIO_OUTPUT_VAL)  ^=   (0x1 << LED2_GPIO_OFFSET) ; //
change led value
    }

}

void vApplicationMallocFailedHook( void )
{
    write(1,"malloc failed\n", 14);
    for( ;; );
}

extern UBaseType_t uxCriticalNesting;
void vApplicationIdleHook( void ){
    volatile size_t xFreeStackSpace;

    xFreeStackSpace = xPortGetFreeHeapSize();

    if( xFreeStackSpace > 100 )
    {

    }
}
```

3. 创建 **Makefile** 文件

在 "FreeRTOS_10.3.1" 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 = FreeRTOS_10.3.1
CFLAGS += -Os

BSP_BASE = ../../bsp

C_SRCS += Source/list.c
C_SRCS += Source/queue.c
C_SRCS += Source/tasks.c
C_SRCS += Source/timers.c
C_SRCS += Source/portable/MemMang/heap_4.c
C_SRCS += Source/portable/GCC/T-Core/port.c

C_SRCS += Demo/RISCV_TCORE_GCC/main.c
C_SRCS += $(BSP_BASE)/$(BOARD)/drivers/plic/plic_driver.c

INCLUDES += -ISource/include
INCLUDES += -IDemo/RISCV_TCORE_GCC
INCLUDES += -ISource/portable/GCC/T-Core

ASM_SRCS += Source/portable/GCC/T-Core/portasm.S

include $(BSP_BASE)/$(BOARD)/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 "FreeRTOS_10.3.1"。

4.2.2 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=FreeRTOS_10.3.1" 命令编译应用程序。如图 4.2.1 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=FreeRTOS_10.3.1                # 编译应用程序
```

图4.2.1 编译应用程序

2. 工程编译完成之后，可以看到在 "FreeRTOS_10.3.1" 文件夹下生成了可执行文件 "FreeRTOS_10.3.1"，如图 4.2.2 所示。

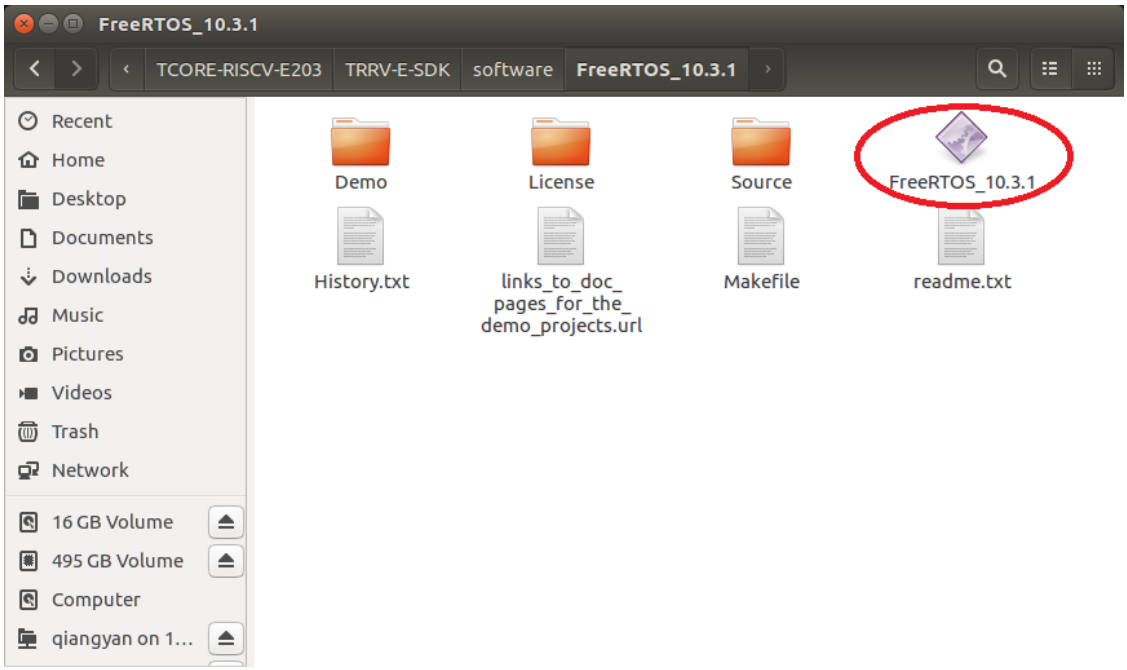

图4.2.2 编译生成二进制文件

4.2.3 执行工程

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

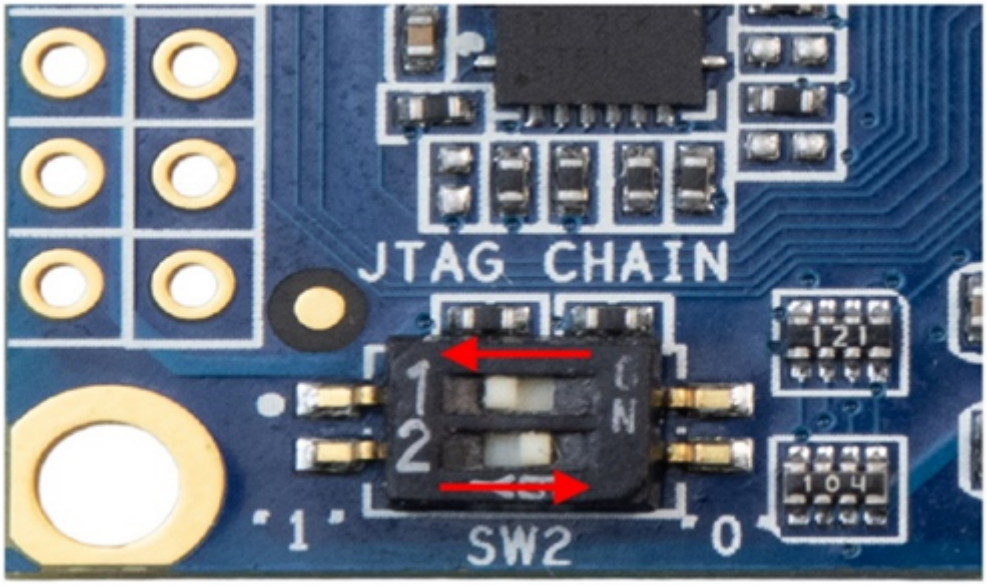

图4.2.3 设置 SW2 开关

2. 将 SIF 子卡连接到 T-Core 的 TMD 2×6 header （JP6），并使用 USB miniB 线缆将 SIF 子卡与 PC 连接。

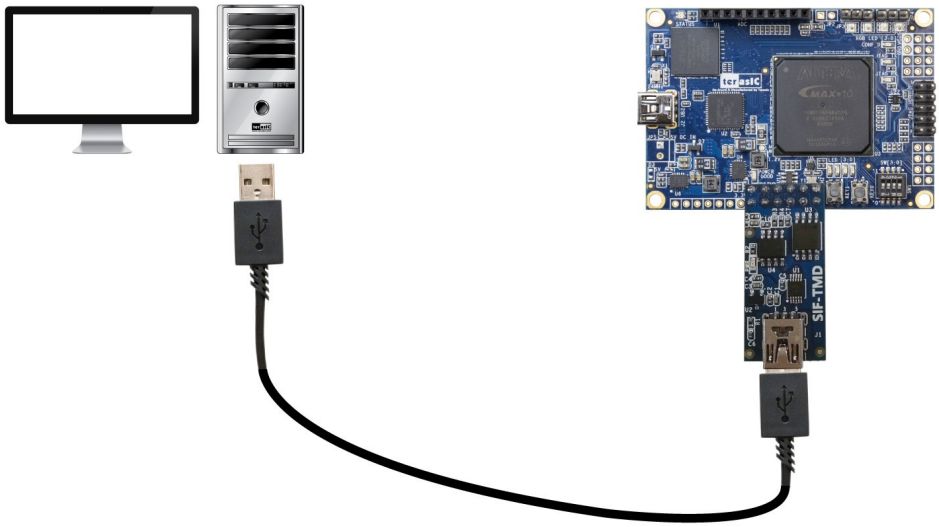

图4.2.4 连接 SIF 子卡到 T-CORE 和 PC

3. 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。

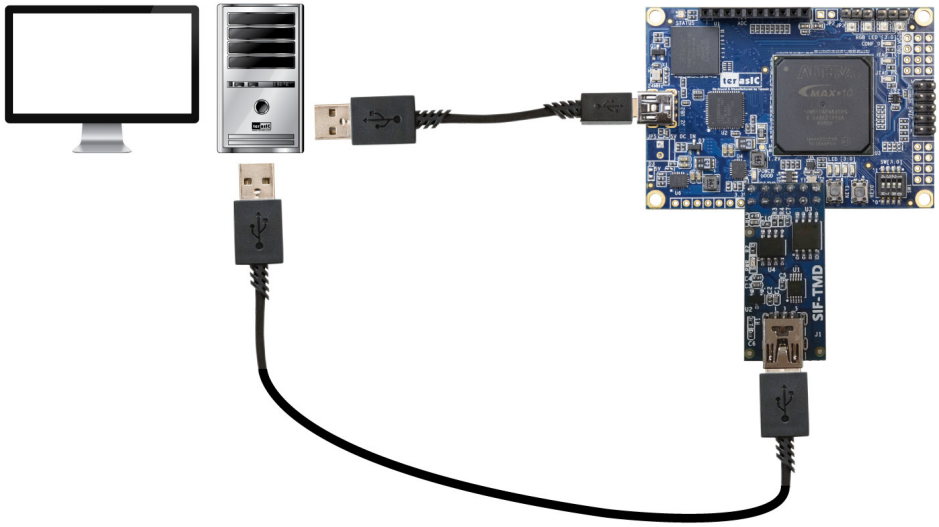

图4.2.5 连接开发板和 PC

4. 使用 "sudo minicom" 命令打开 linux 系统的串口调试工具。

```
sudo minicom
```

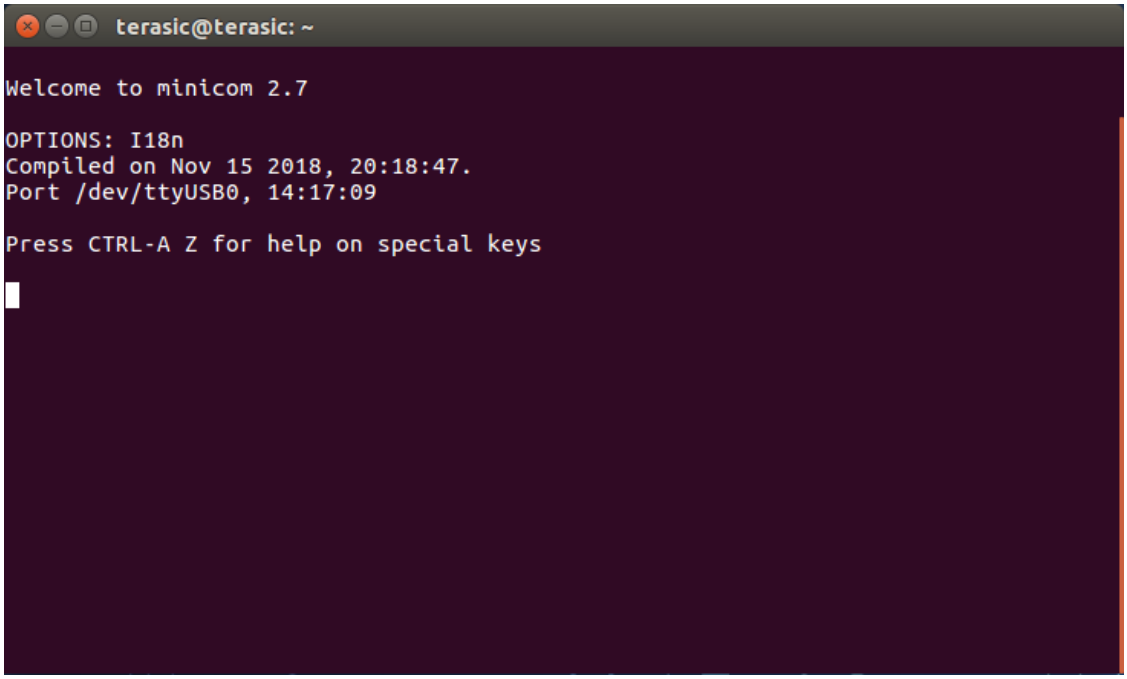

图4.2.6 打开串口调试工具

5. 使用 "make upload PROGRAM=FreeRTOS_10.3.1" 将可执行文件 "FreeRTOS_10.3.1" 下载到 T-Core 开发板的 QSPI Flash 中。

```
make upload PROGRAM=FreeRTOS_10.3.1
```

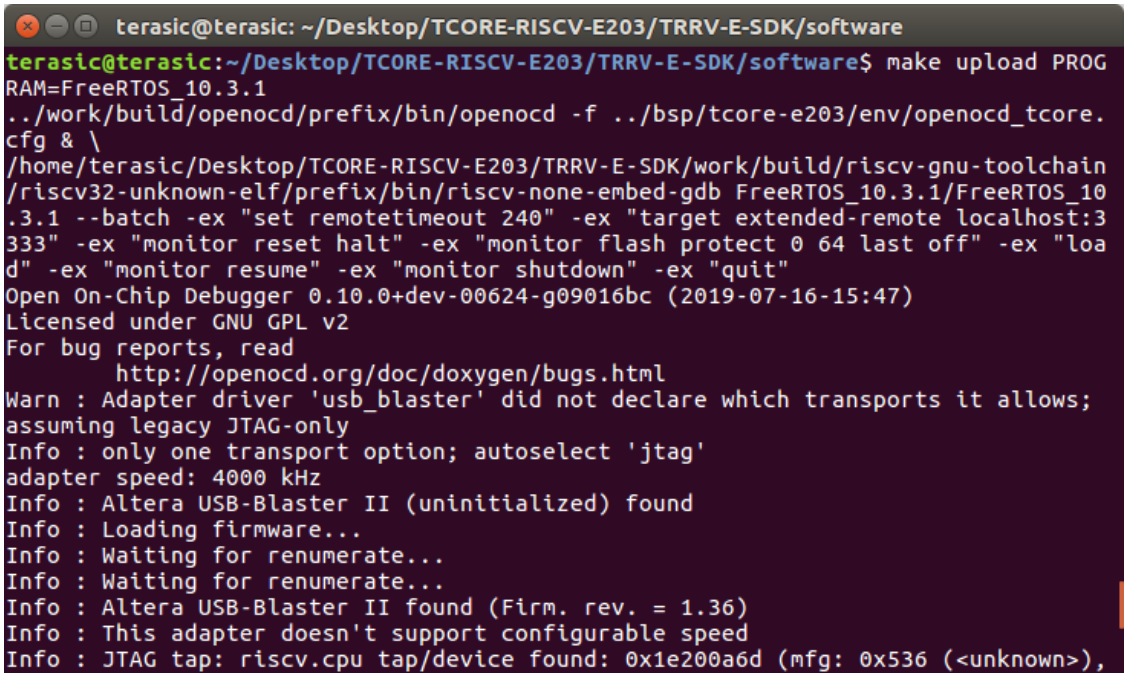

图4.2.7 下载可执行文件

4.2.4 运行结果

程序下载完成后，串口通信工具窗口会打印出执行 5 次 Task2 的信息，然后 Task2 被阻塞，此时执行 Task1，3s 后，继续执行 Task2。

```
terasic@terasic: ~
terasic@terasic:~$ sudo minicom
[sudo] password for terasic:


Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Nov 15 2018, 20:18:47.
Port /dev/ttyUSB0, 20:08:15

Press CTRL-A Z for help on special keys

Malloc aCore freq at 16001597 Hz
do Task2: priority is 2
Task2 count:1
do Task2: priority is 2
Task2 count:2
do Task2: priority is 2
Task2 count:3
do Task2: priority is 2
Task2 count:4
do Task2: priority is 2
Task2 count:5
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task1: priority is 1
do Task2: priority is 2
Task2 count:1
do Task2: priority is 2
```

图4.2.8 运行结果

接着将 main.c 文件中 Task1 和 Task2 的优先级更改为图 4.2.9 所示的代码并保存。

```
105 int main(void)
106 {
107     /* Configure the system ready to run the demo. */
108     prvSetupHardware();
109
110     /* Create the task1 as described in the comments at the top of this file.
111     xTaskCreate(    prvTask1,                                   // The functic
112                     ( const char * ) "Task1",   // Text name for the task, jus
113                                     configMINIMAL_STACK_SIZE,       // The
114                     NULL,                                           // A p
115                     tskIDLE_PRIORITY+2,                 // The priority to ass
116                     &Task1_Handler );                   // Used to obtain a ha
117
118     // Create the task2 in exactly the same way.
119     xTaskCreate(     prvTask2,
120                     ( const char * ) "Task2",
121                                     configMINIMAL_STACK_SIZE,
122                     NULL,
123                     tskIDLE_PRIORITY+1,
124                     &Task2_Handler );
125
126     /* Start the tasks and timer running. */
127     vTaskStartScheduler();
128 }
```

图4.2.9 修改 main.c

再次执行 4.2.2 和 4.2.3 节中的操作，编译和下载应用程序，可以观察到：串口通信工具窗口会一直打印任务 1 的相关信息（一直执行 Task1）。

```
terasic@terasic: ~
Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Nov 15 2018, 20:18:47.
Port /dev/ttyUSB0, 09:49:59

Press CTRL-A Z for help on special keys

do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
do Task1: priority is 2
```

图4.2.10 运行结果

4.3 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）

4.3.1 创建 FreeRTOS_10.3.1 工程

1. 将文件夹命名由 "blinking_LED" 修改为 "FreeRTOS_10.3.1"，如图 4.3.1 所示。

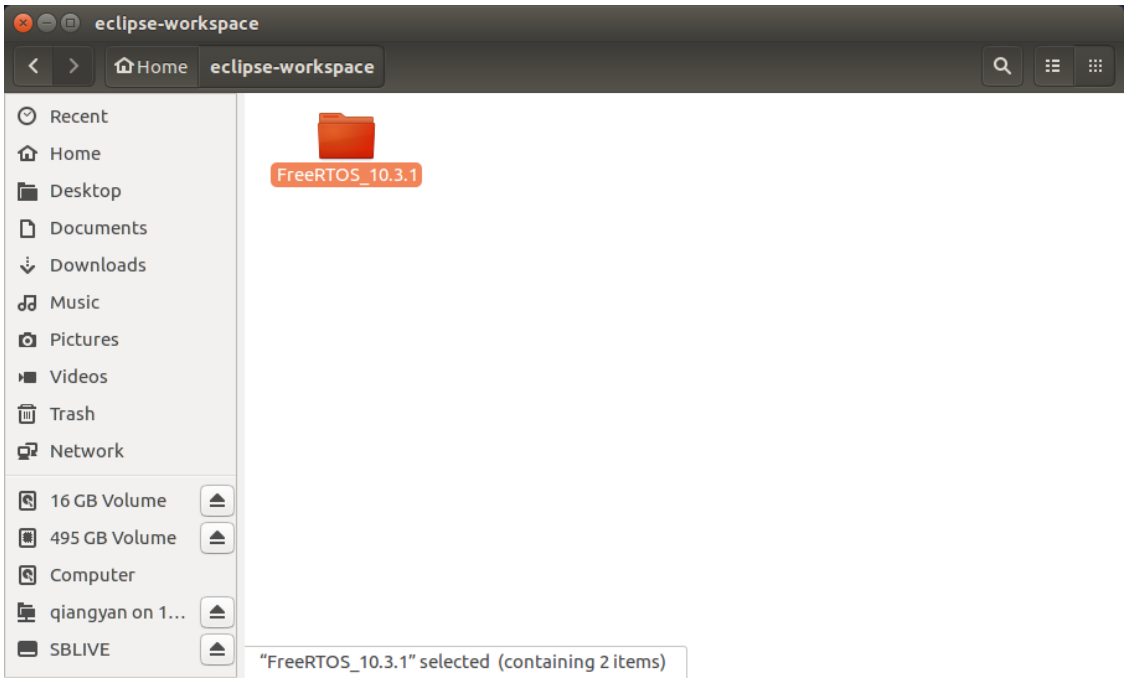

图4.3.1 修改文件名为 "FreeRTOS_10.3.1"

2. 将 4.2 节中创建的 software 路径下的 FreeRTOS_10.3.1 文件夹复制到当前文件夹路径下的 src 文件夹中，删除 src 文件夹中的 main.c 文件。

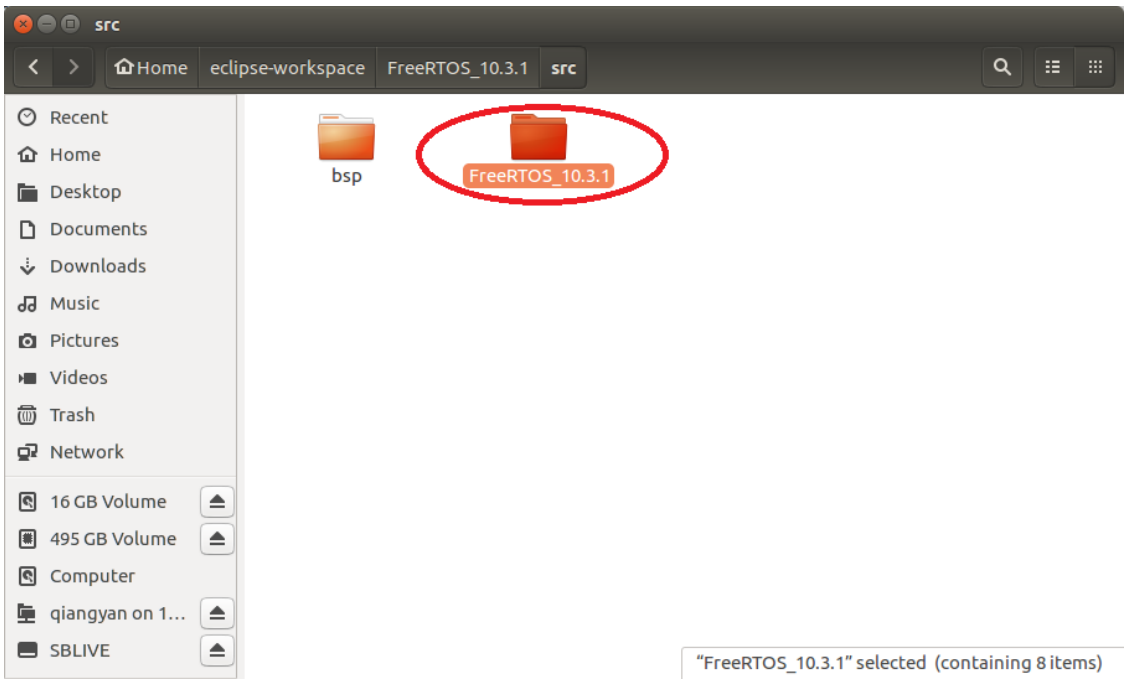

图4.3.2 复制 "FreeRTOS_10.3.1" 到 src 文件夹

3. 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。

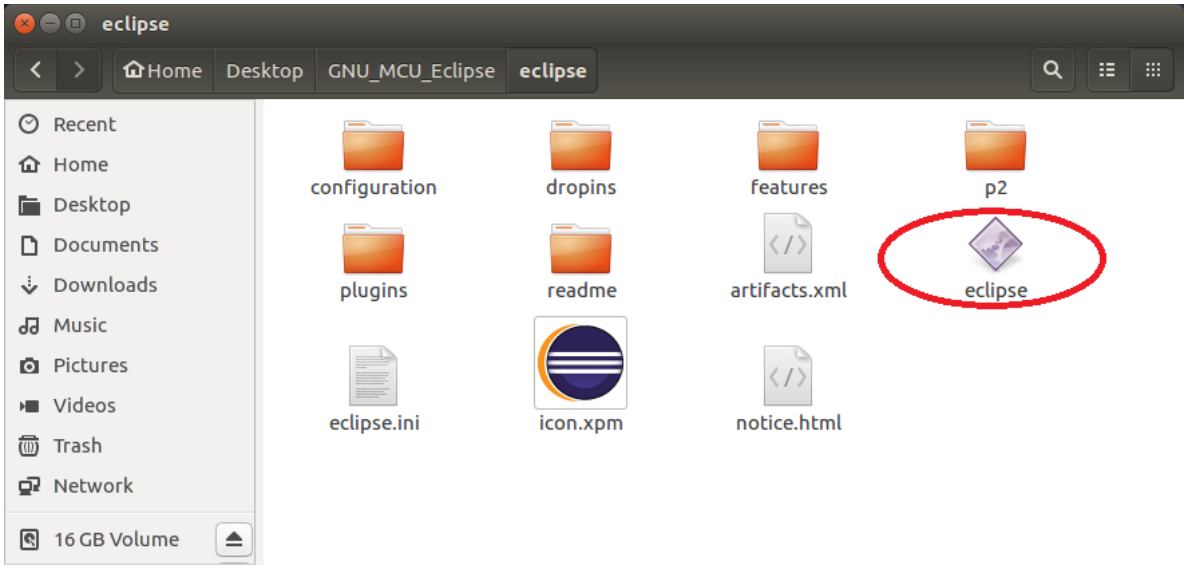

图4.3.3 启动 Eclipse

3. 启动 Eclipse 后，弹出设置 Workspace 的对话框，如图 4.3.4 所示，默认为 home 下的 eclipse-workspace（可根据需要自行设置）。

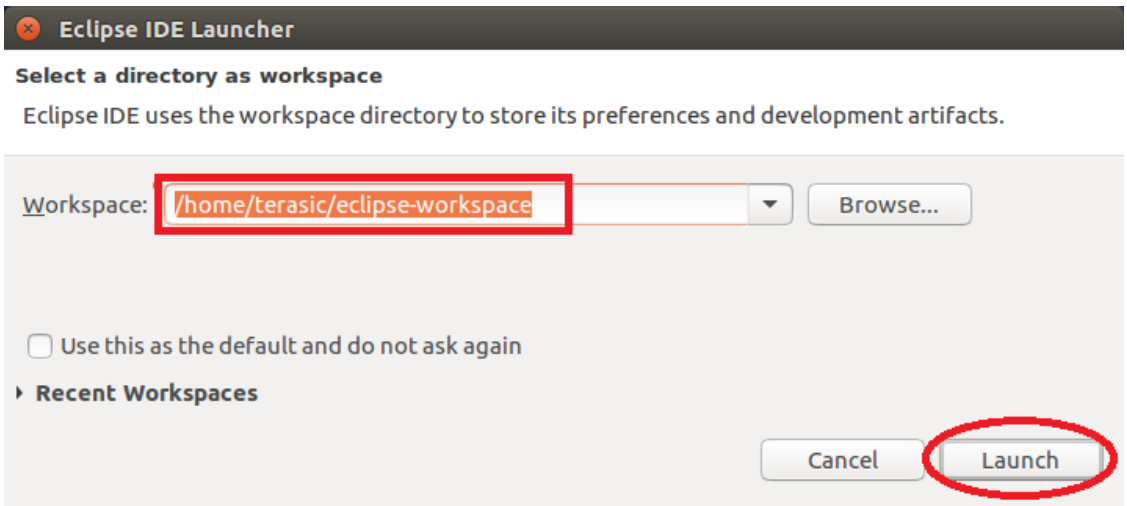

图4.3.4 设置 Workspace

4. 设置好 Workspace 目录后，单击 Launch，将会启动 Eclipse，进入 Welcome 界面，如图 4.3.5 所示。

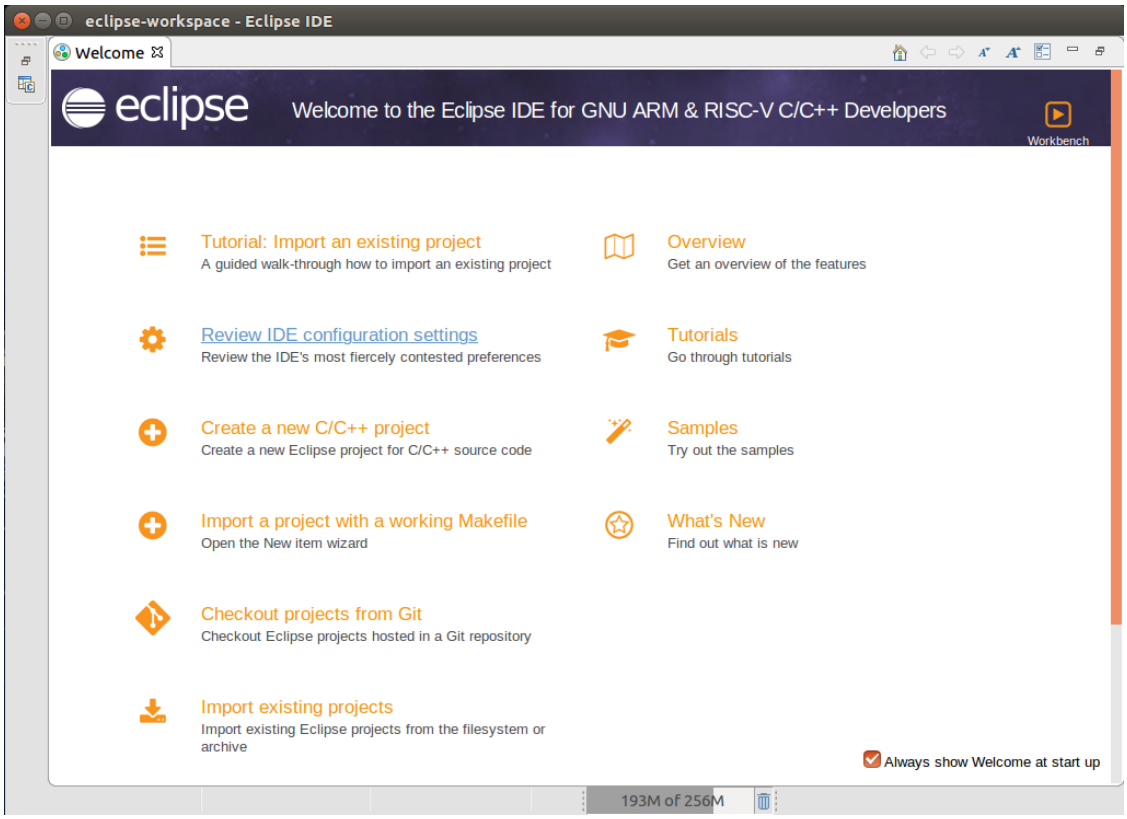

图4.3.5 进入 Eclipse 界面

5. 点击 Welcome 处的叉号，关闭 Welcome 界面，如图 4.3.6 所示。

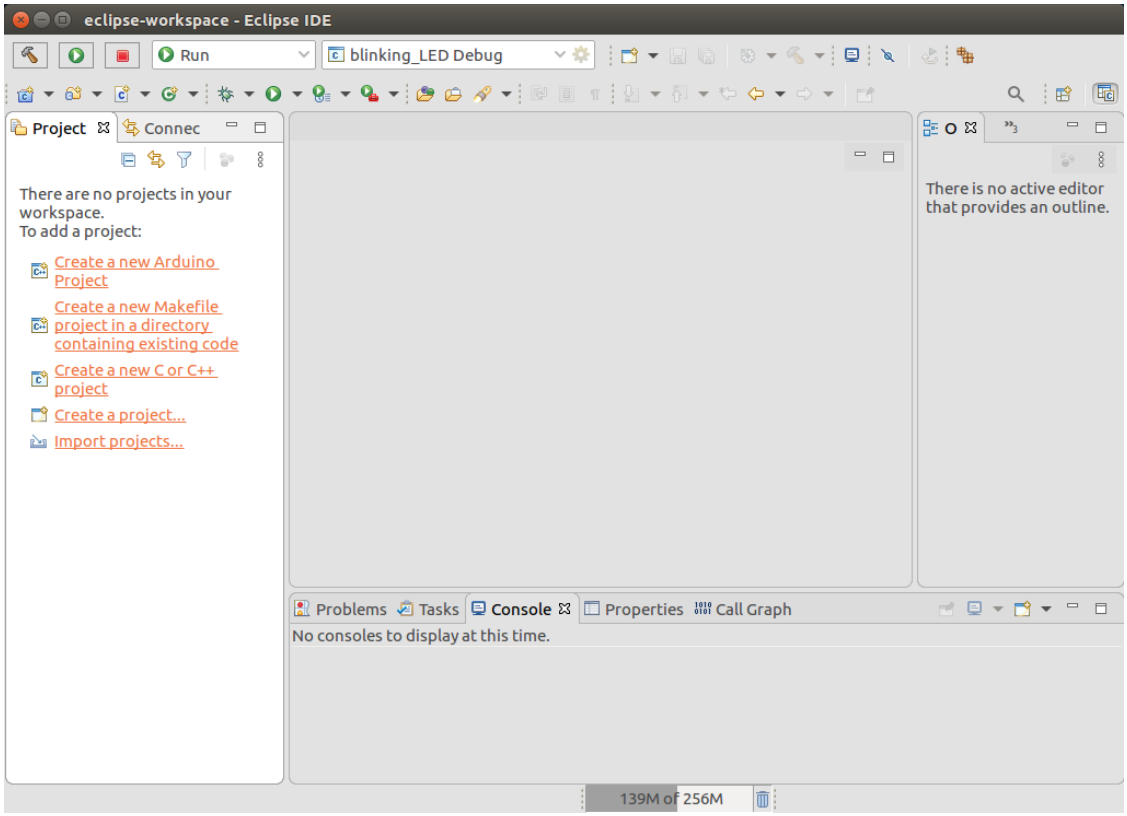

图4.3.6 进入 Eclipse 界面

6. 点击菜单栏 File -> Import... 导入工程，出现如图 4.3.7 所示界面，选择 "Existing Projects into Workspace"，点击 Next。（注：把鼠标移动到顶部菜单栏就会看到 File 选项）

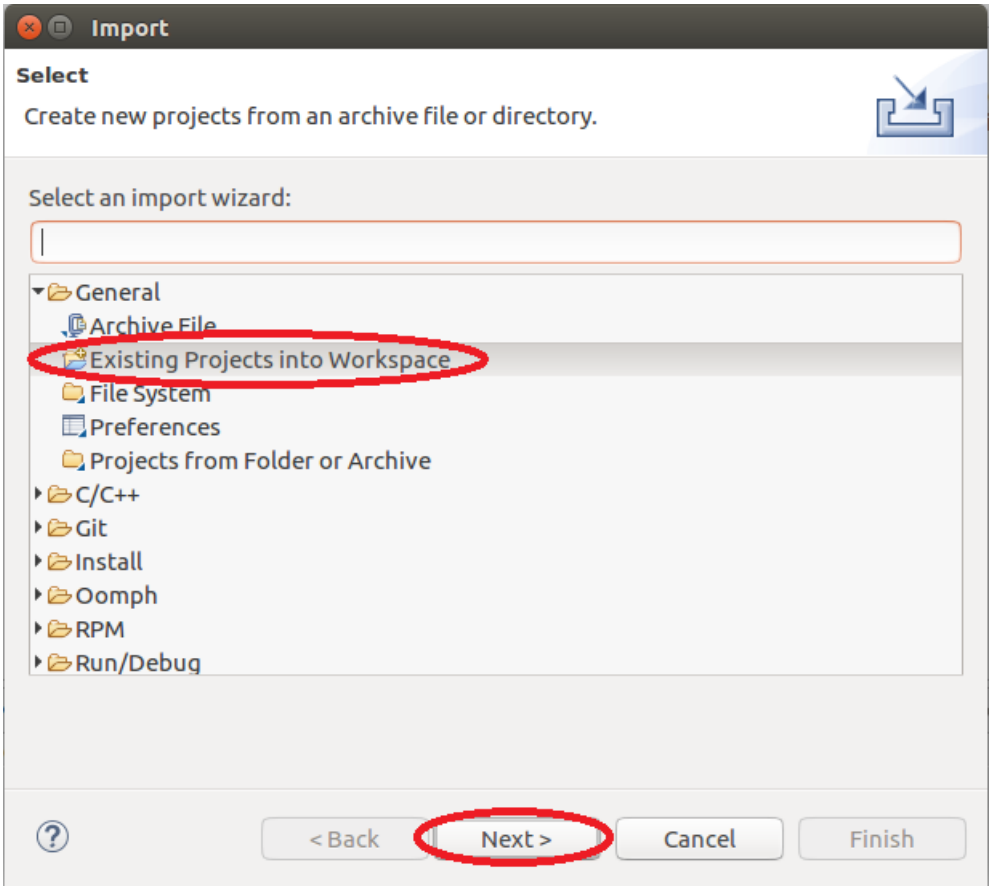

图4.3.7 选择导入工程类型

7. 点击 Browse 导入已有的工程。

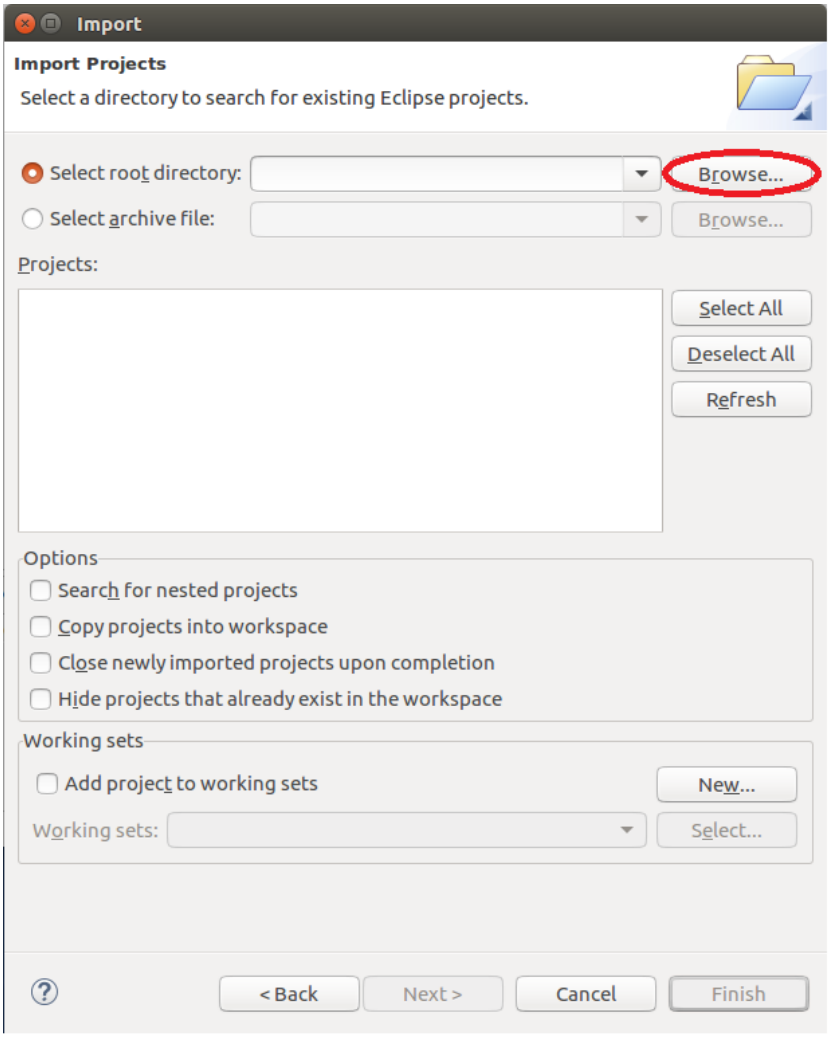

图4.3.8 点击 Browse

8. 选择要添加的 FreeRTOS_10.3.1 工程，点击 OK。

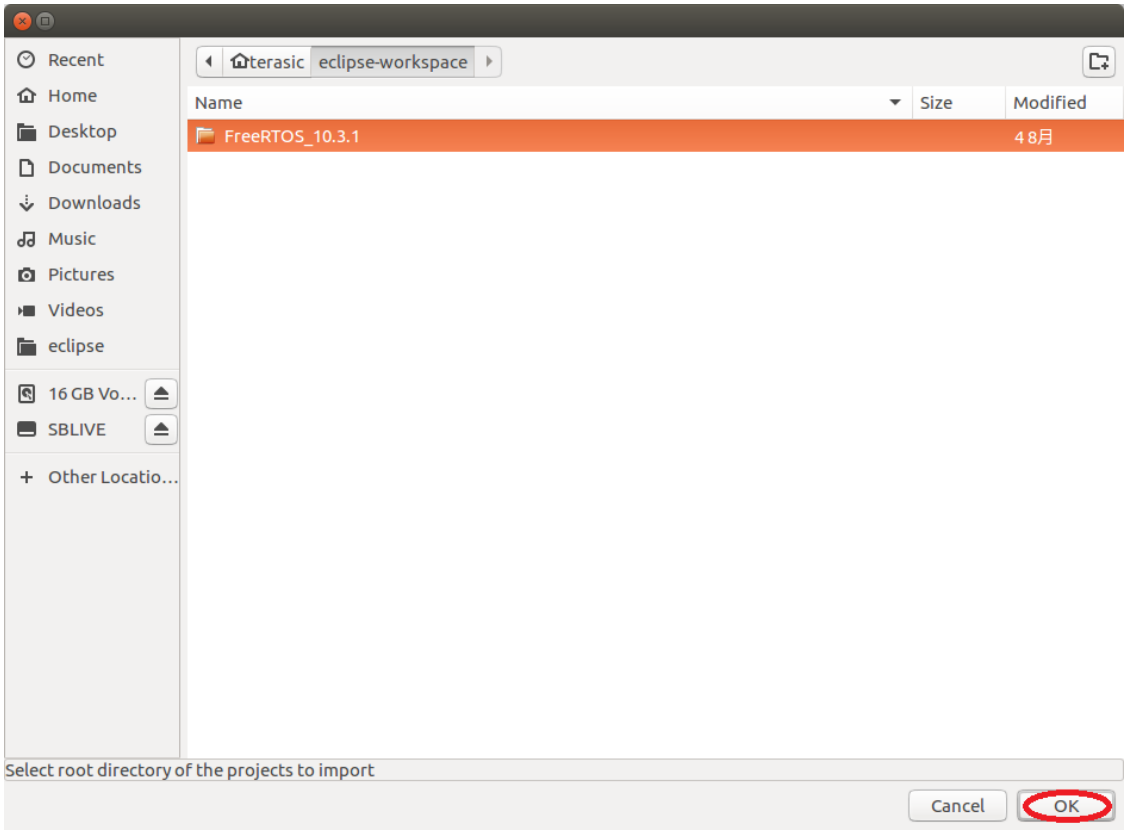

图4.3.9 添加 FreeRTOS_10.3.1 工程

9. 勾选 "Add projects to working sets" 将 FreeRTOS_10.3.1 工程添加到当前工作空间，点击 Finish。

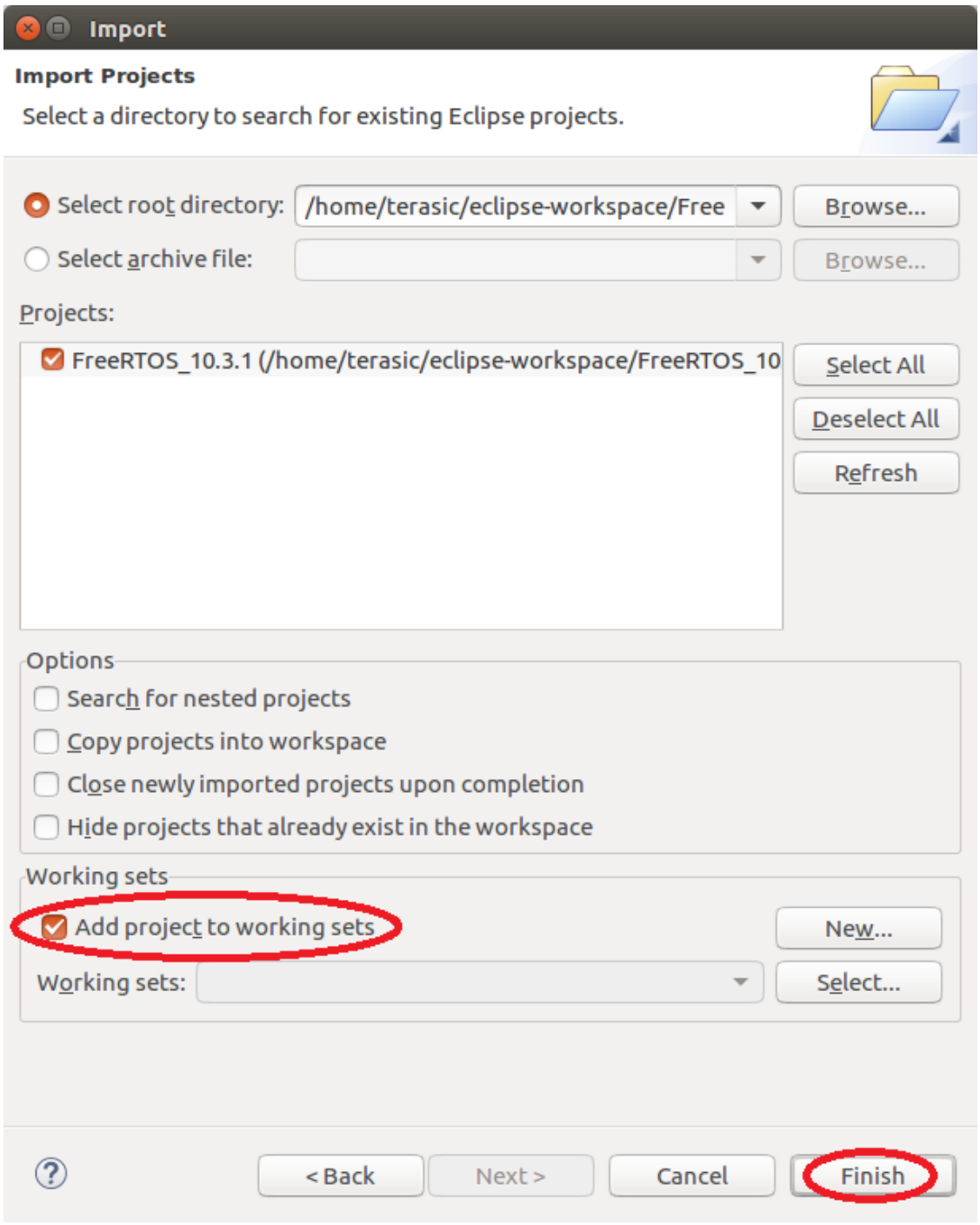

图4.3.10 添加 FreeRTOS_10.3.1 工程到工作空间

10. 导入后的工程界面如图 4.3.11 所示。

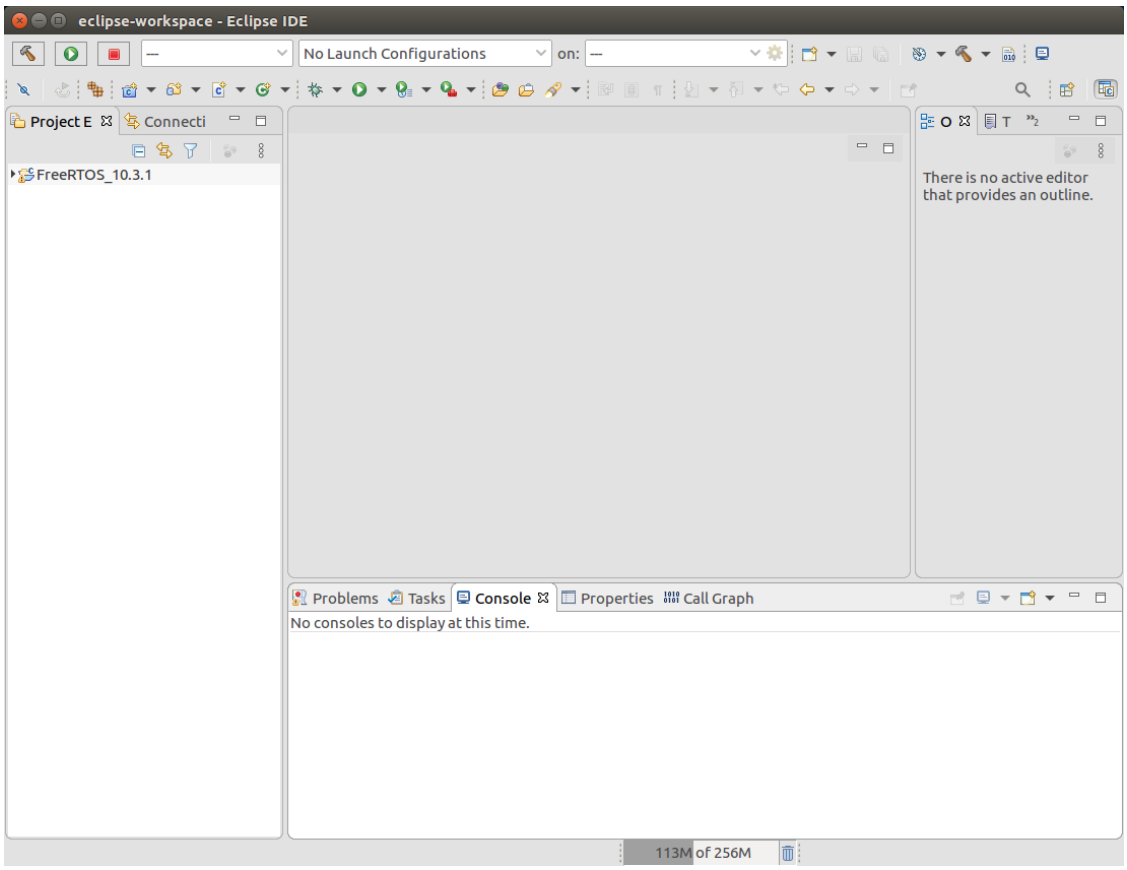

图4.3.11 导入后的工程界面

4.3.2 编译 FreeRTOS_10.3.1 工程

1. 点击 FreeRTOS_10.3.1 --> src --> FreeRTOS_10.3.1 --> Source --> portable --> MemMang 下拉框，右键 Delete 删除除 heap_4 外的其他文件。eclipse 会把所有的 c 文件都包含到 makefile，但是我们只需要 heap_4 相关文件，所以要删除掉其他的 heap，否则编译时会报错。

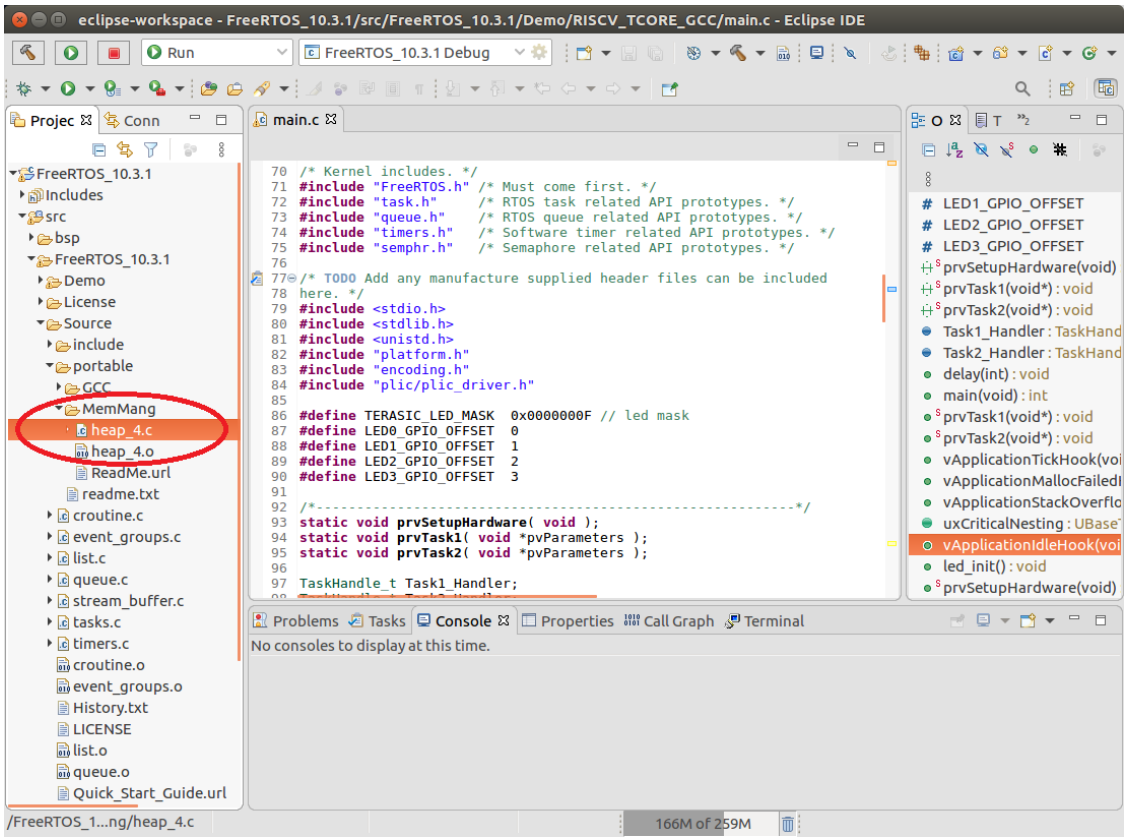

图4.3.12 删除多余 heap 文件

2. 在 Eclipse 主界面中，选中 FreeRTOS_10.3.1 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 4.3.13 所示。

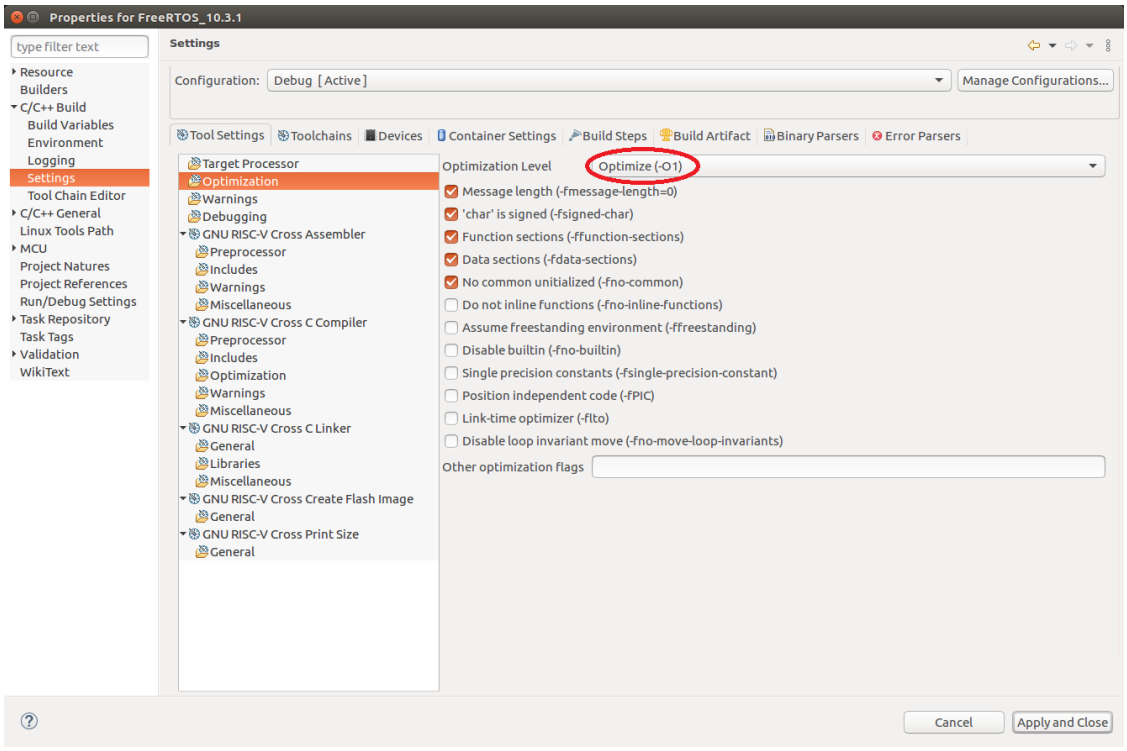

图4.3.13 修改 Optimization level

3. 点击 GNU RISC-V Cross C Compiler 下拉下的 Includes，在 Include paths 下添加如图 4.3.14 所示的路径。

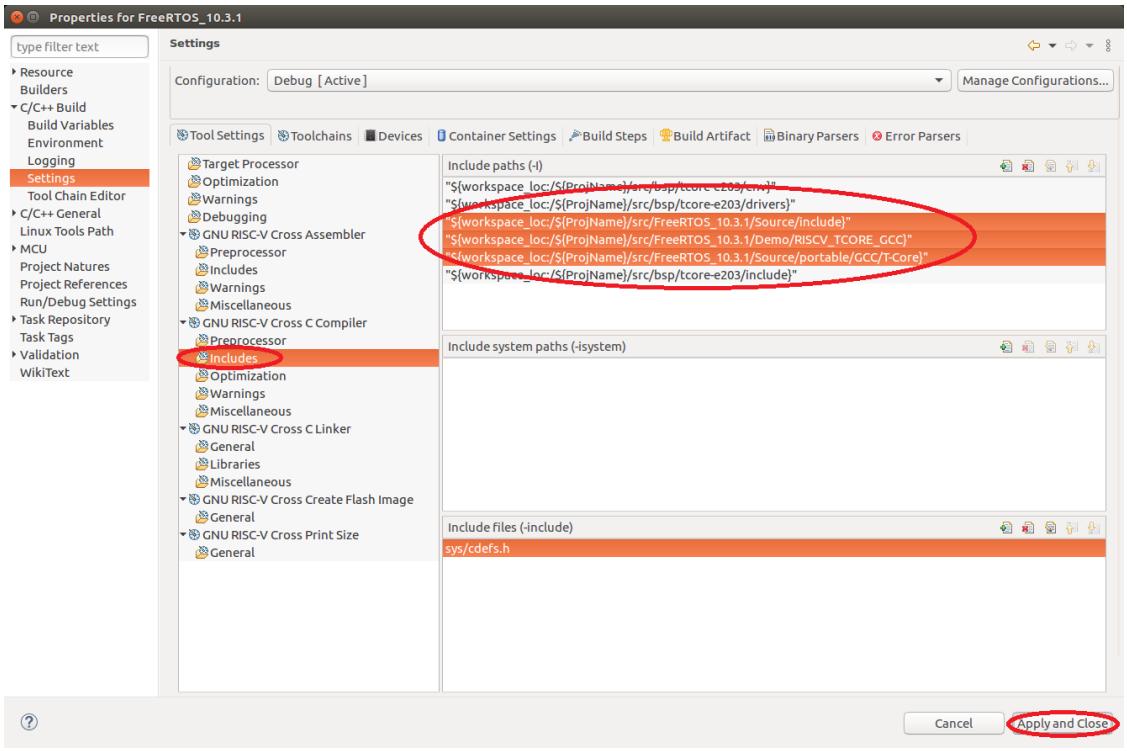

图4.3.14 添加 Include paths

4. 选中 FreeRTOS_10.3.1 工程，右键点击 Clean Project；再次选中 FreeRTOS_10.3.1 工程，右键点击 Build Project，若 FreeRTOS_10.3.1 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 4.3.15 所示。需要右键点击 Refresh 在 Debug 下拉项中看到生成的 .elf 文件。

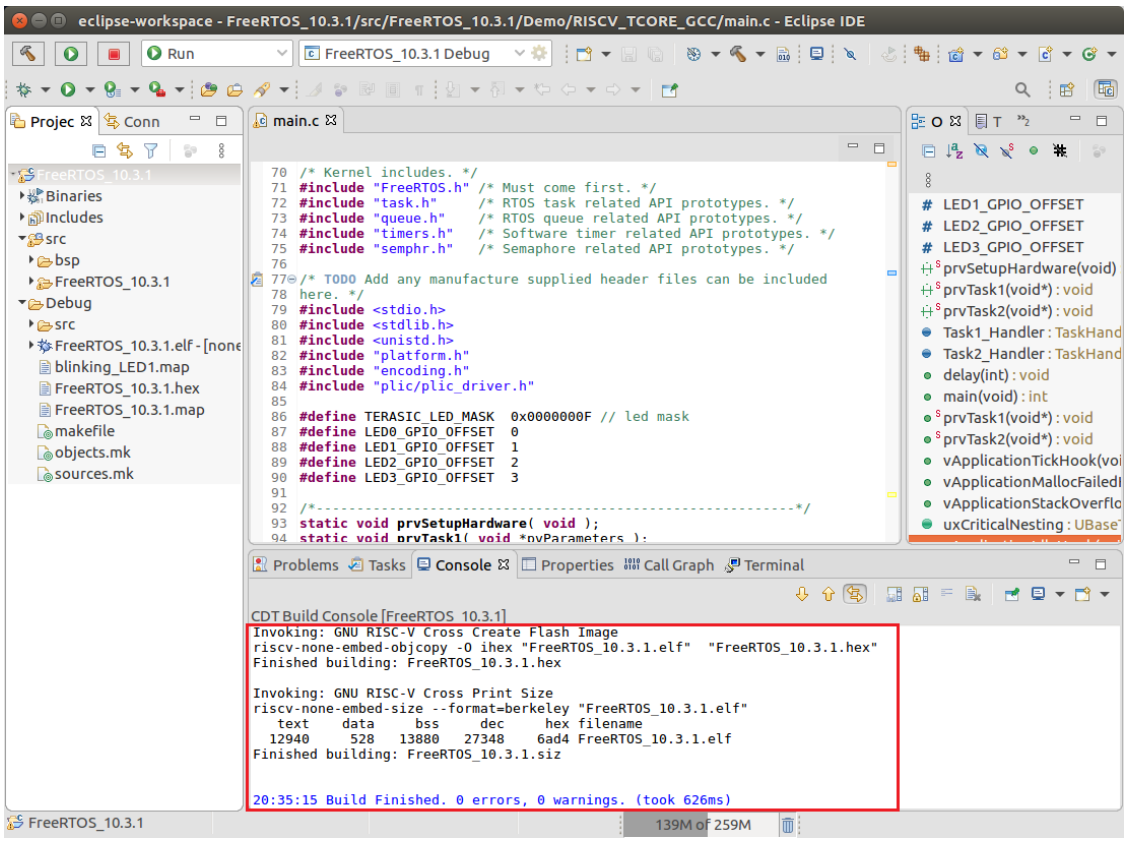

图4.3.15 编译成功

5. 右键 Debug 下拉选项中的 "blinking_LED.map"，点击 Delete，删除完成后如图 4.3.16 所示。

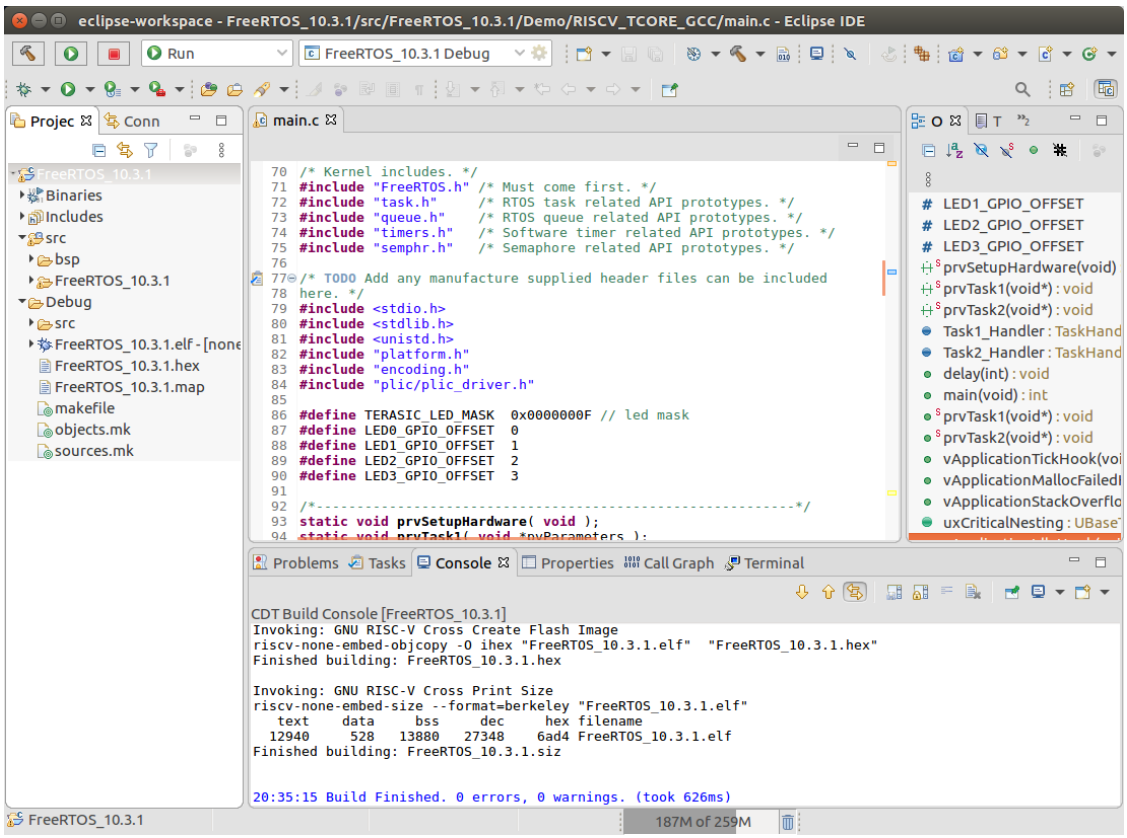

图4.3.16 删除 blinking_LED.map 文件

4.3.3 运行 FreeRTOS_10.3.1 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 设置为 SW2.1=1，SW2.2=0，选择 RISC-V JTAG 链路。

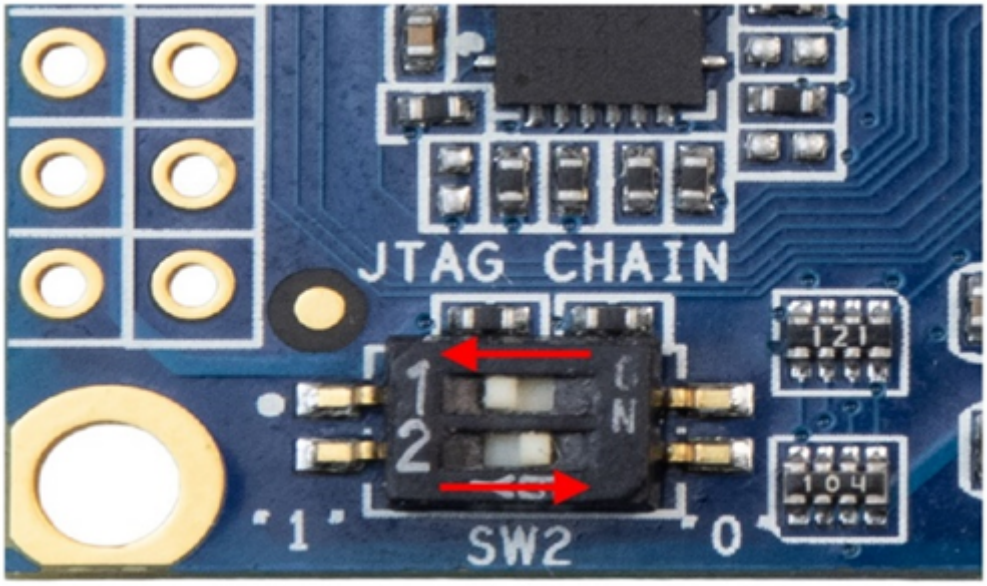

图4.3.17 设置 SW2 开关

- 将 SIF 子卡连接到 T-Core 的 TMD 2×6 header （JP6），并使用 USB miniB 线缆将 SIF 子卡与 PC 连接。
- 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. 在运行工程前，还是要先打开串口调试工具，在 Eclipse 软件中自带有 Terminal 终端，依次点击菜单栏中 Window --> Show View --> Terminal，使 Terminal 显示出来，然后点击右侧蓝色图标。（注：把鼠标移动到顶部菜单栏就会看到 Window 选项）

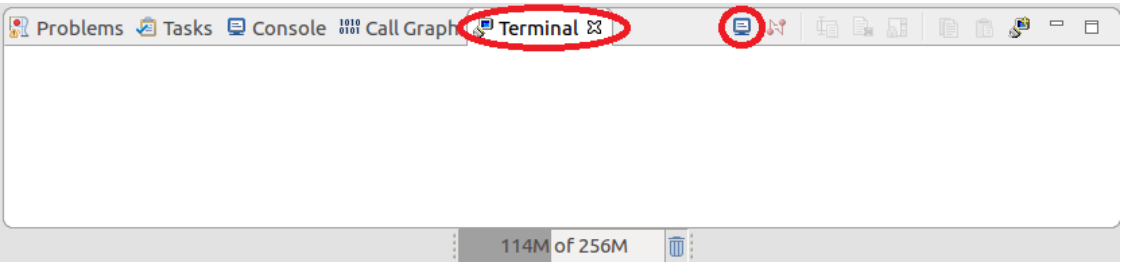

图4.3.18 显示 Terminal 终端

3. 选择 Loacal Terminal，编码方式选择 UTF-8，打开一个 Terminal 终端。

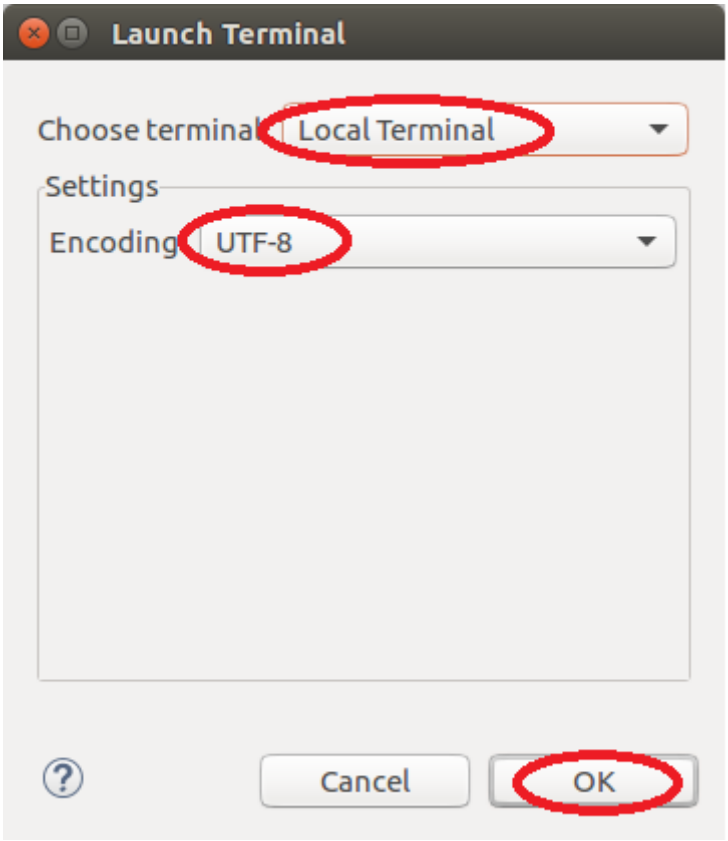

图4.3.19 打开 Terminal 终端

4. 打开后的 Terminal 如图 4.3.20 所示。

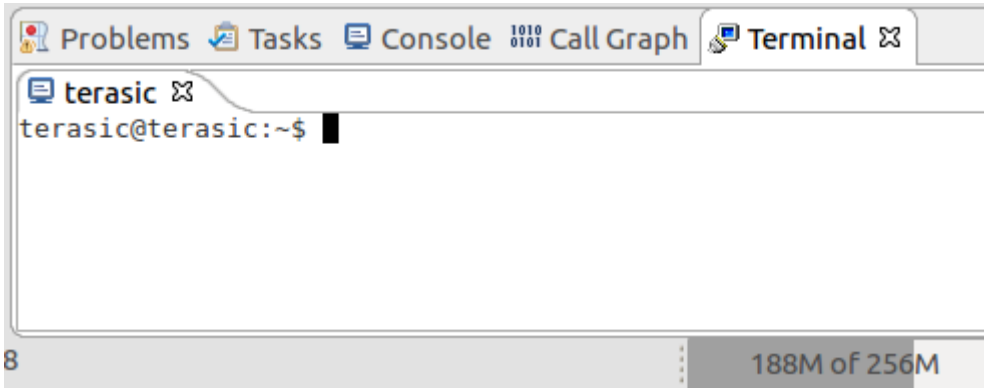

图4.3.20 Terminal 终端

5. 使用 "sudo minicom" 命令打开 linux 系统的串口调试工具。

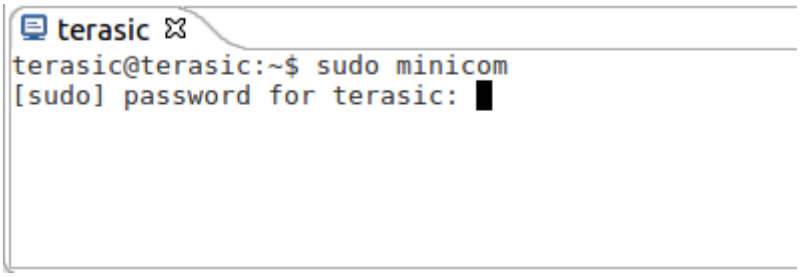

图4.3.21 打开串口调试工具

6. 选中 FreeRTOS_10.3.1 工程，右键点击 Run As -> Run Configurations...，双击 GDB OpenOCD Debugging 会出现如图 4.3.22 所示的 FreeRTOS_10.3.1 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 FreeRTOS_10.3.1 工程。

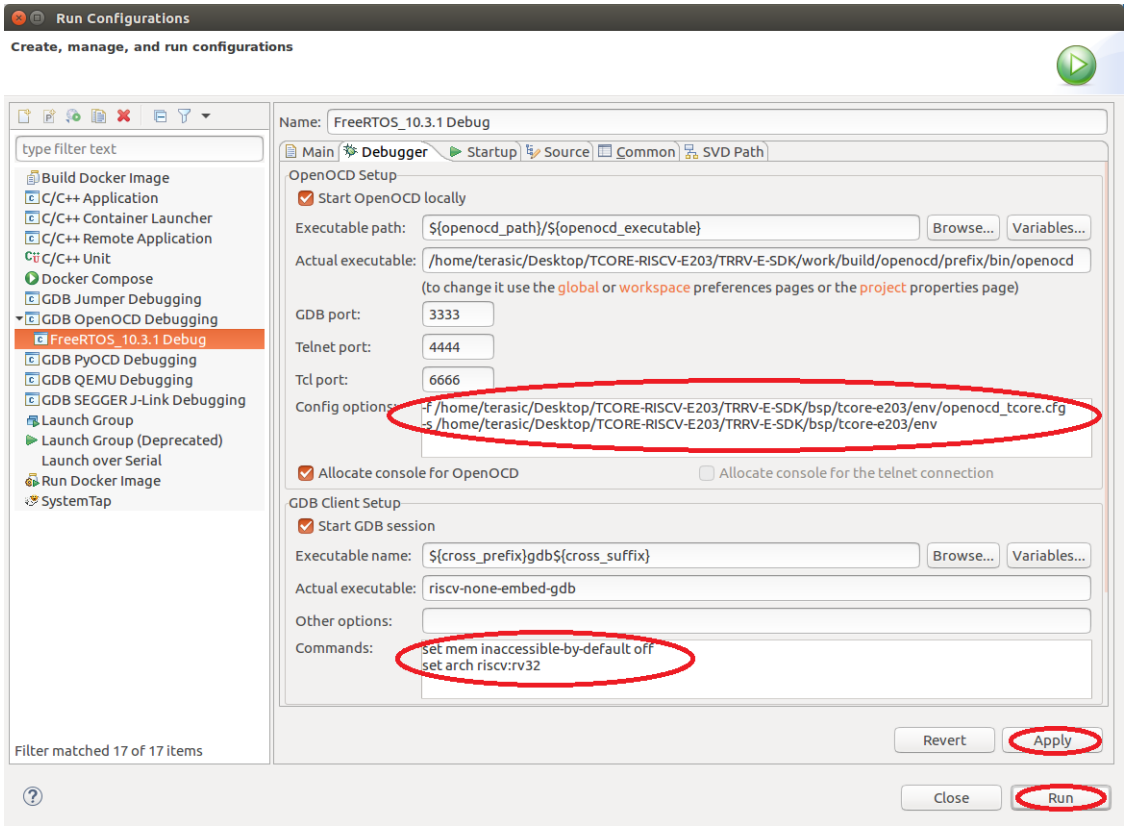

图4.3.22 运行配置

7. 程序下载成功后，如图 4.3.23 所示。

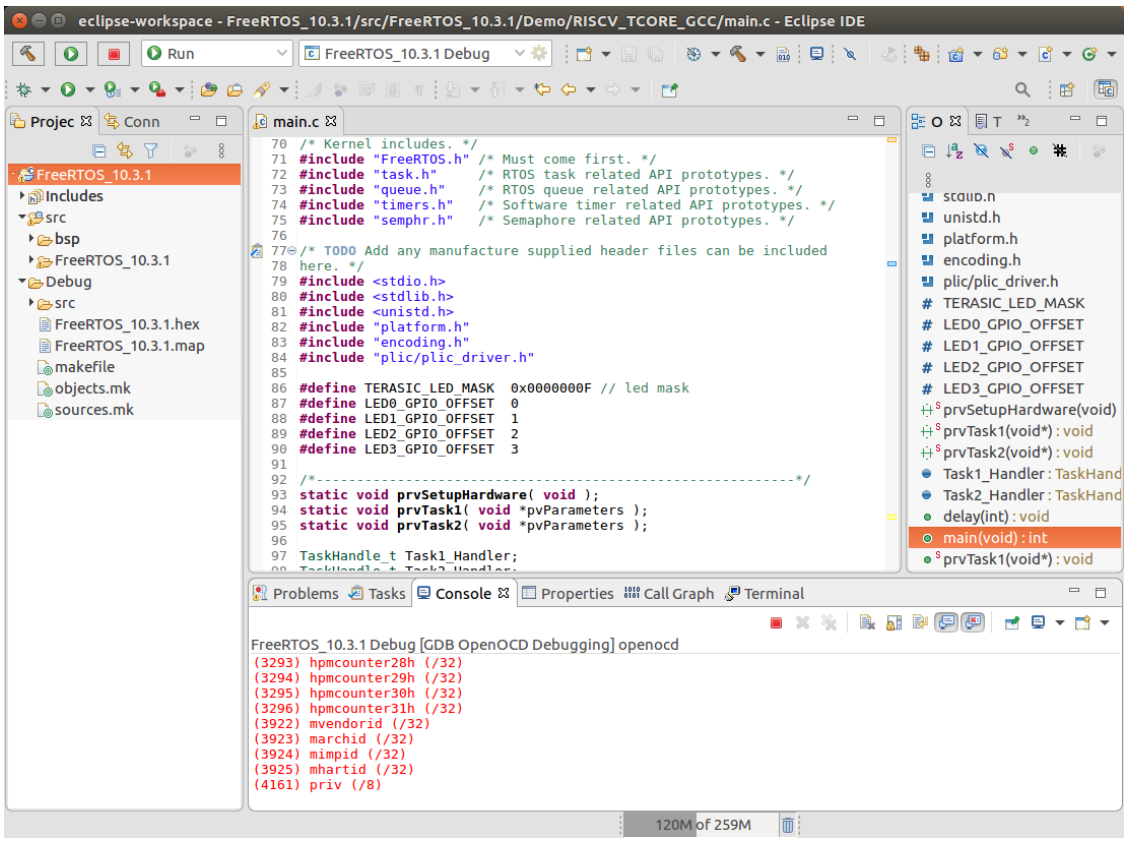

图4.3.23 运行 FreeRTOS_10.3.1 工程

4.3.4 运行结果

程序下载完成后，先按 KEY0 键复位。串口通信工具窗口会打印出执行 5 次 Task2 的信息，然后 Task2 被阻塞，此时执行 Task1，3s 后，继续执行 Task2。

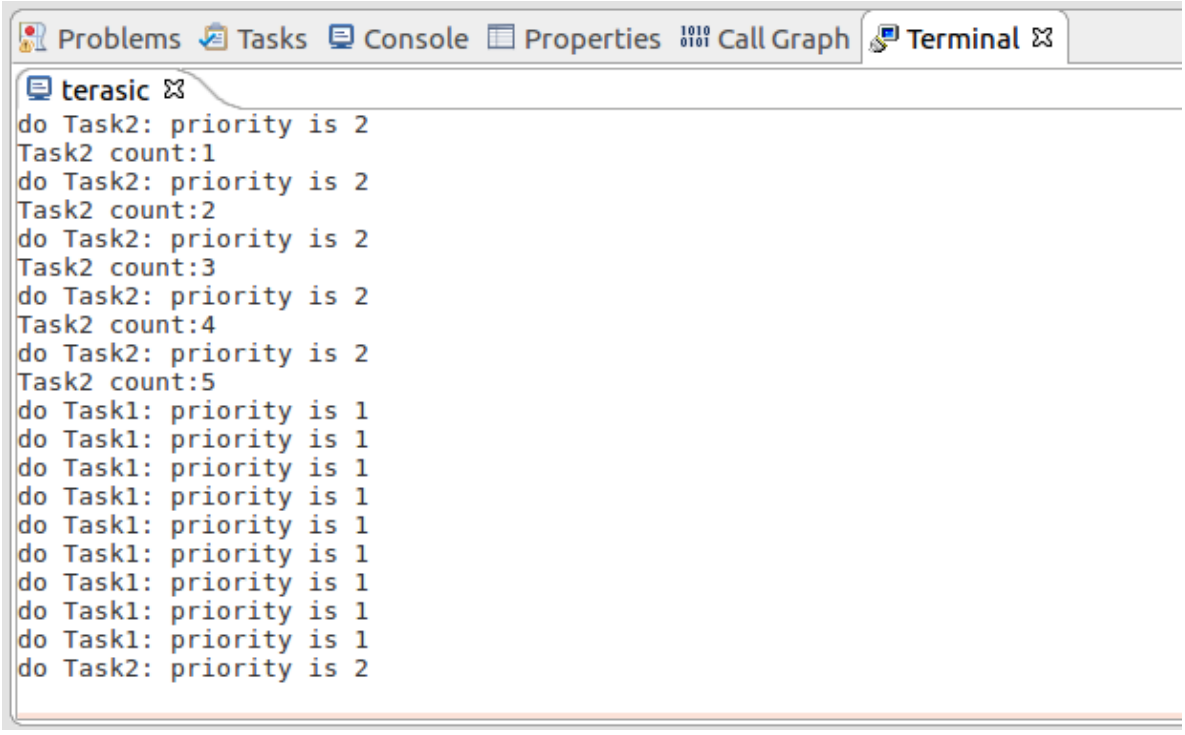

图4.3.24 运行结果

同样地，修改 Task1 和 Task2 的优先级，再次编译和下载应用程序，可以观察到：串口通信工具窗口会一直打印任务 1 的相关信息（一直执行 Task1）。

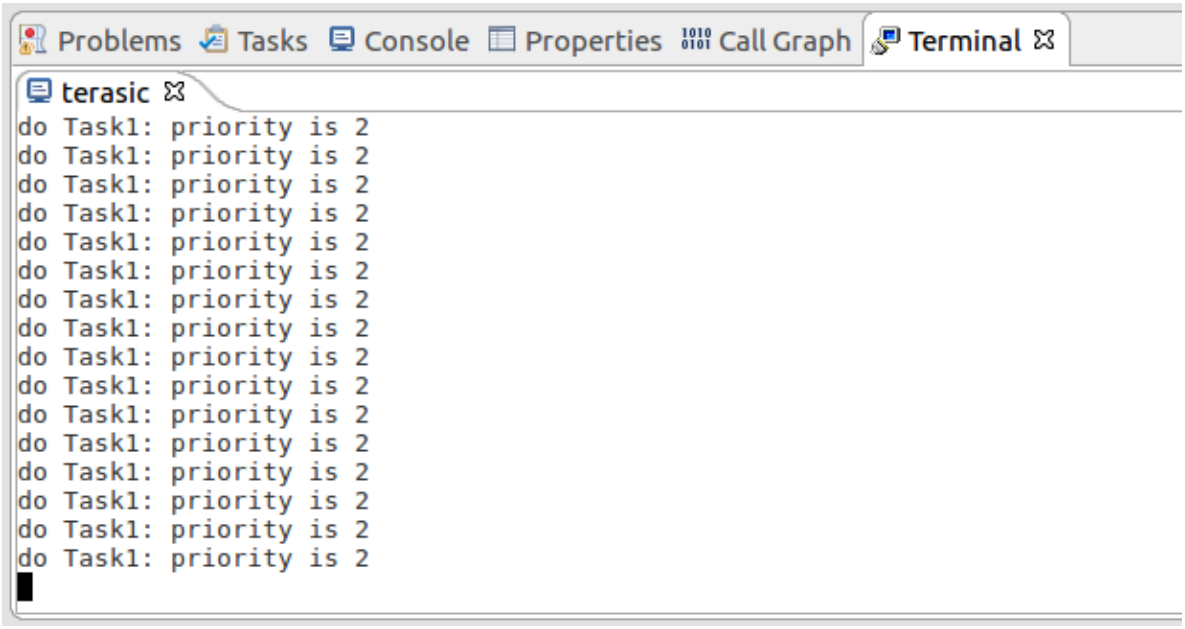

图4.3.25 运行结果

附录

1. 修订历史

版本	时间	修改记录
V1.0	2020.08.19	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