

# 实战演练之PWM呼吸灯

---

## 一、概述

---

本教程将介绍如何在 T-Core 开发板上实现 RISC-V 设计——PWM 呼吸灯：通过配置 PWM 相关寄存器，产生 PWM 波，输出到 LED 实现呼吸灯的效果。

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 demo\_pwm 工程、修改 main.c 主函数、编译并运行 demo\_pwm 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；
- 了解发光二极管（LED）的工作原理及驱动方法；
- 学习 RISC-V PWM 外设的结构和工作原理；
- 掌握 RISC-V PWM 的实现原理。

## 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-V1.2.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。）

## 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、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 3.1 为 T-Core 开发板 LED 灯和 FPGA 之间的连接示意图。

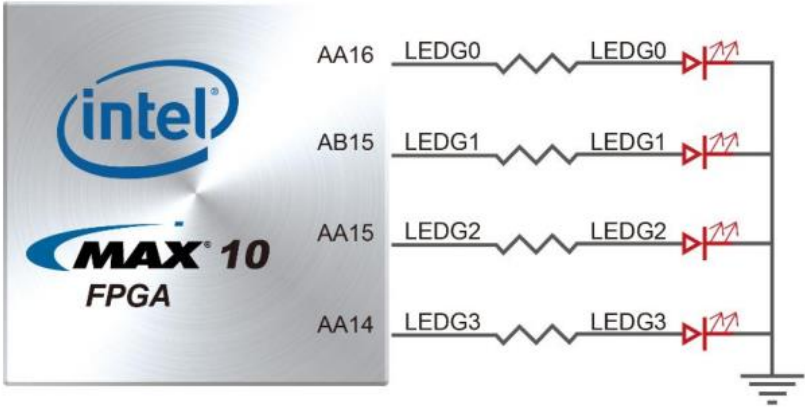


图3.1 T-Core 开发板 LED 灯和 FPGA 之间的连接

### 3.2 GPIO 寄存器列表

本教程主要用到以下 GPIO 寄存器：GPIO\_IOF\_EN 寄存器用于使能 H/W IO Function，GPIO\_IOF\_SEL 寄存器用于选择 H/W IO 的来源。

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_IOF_EN	0x038	0x0	H/W IO Function 使能
GPIO_IOF_SEL	0x03C	0x0	选择 H/W IO 的来源

### 3.3 GPIO 映射关系

关于 T-Core 的 RISC-V 处理器的 GPIO 与 T-Core 外设 LED 和其外设 PWM0 的映射关系可参考表 3.2。根据表 3.2 可知，当 GPIO0-3 复用为 PWM0 时，改变 PWM0 的输出值，即可控制 LED0-3 的亮灭状态。

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
PWM0_0-3	GPIO0-3

### 3.4 PWM 特性

PWM 全称 Pulse-Width Modulation（脉冲宽度调制），是MCU中常用的模块。PWM 是利用 MCU 的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。

PWM 逻辑结构如图 3.2 所示。

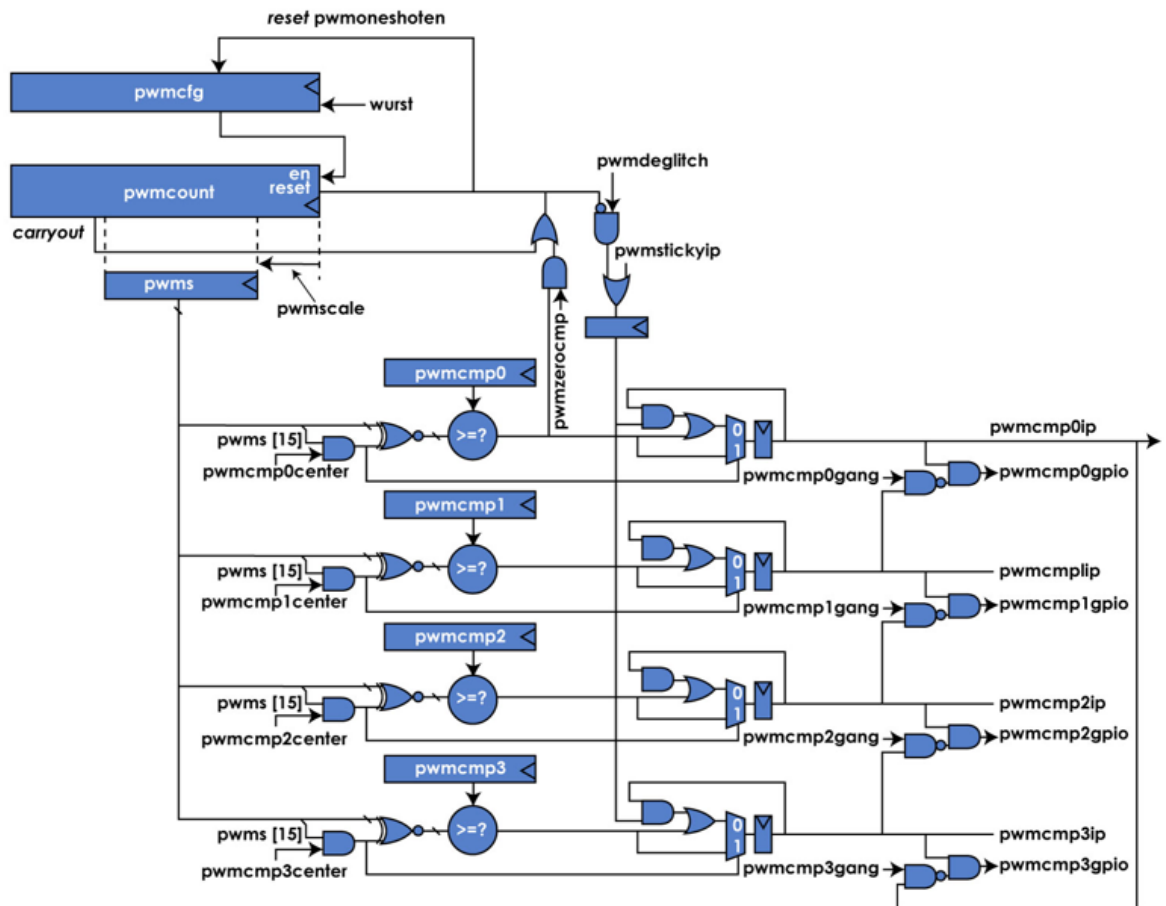


图3.2 PWM 结构图

RISC-V 核支持 3 个 PWM 模块，分别为 PWM0、PWM1 和 PWM2。其中，PWM0 是宽度为 8bit 的比较器，而 PWM1 和 PWM2 是宽度为 16bit 的比较器。除了计数器宽度的差别之外，3 个 PWM 的工作原理和特性完全相同，简述如下。

- 每个 PWM 支持 4 个可编程比较器（称为 pwmcmp0-pwmcmp3），每一个比较器可以产生对应的一路 PWM 输出（信号名为 pwmcmpgpio）和中断（信号名为 pwmcmpip），如图 3.2 所示。为了对 PWM 的原理进行统一地介绍，本教程将使用 pwmcmpwidth 作为参数表示 PWM 比较器的宽度。
- PWM 每一路输出均可以产生靠左对齐或者靠右对齐的脉冲信号。
- PWM 每一路输出均可以产生居中对齐的脉冲信号。
- PWM 每一路输出均可以产生任意形状的脉冲信号。
- PWM 每一路输出均可以产生周期性的脉冲信号或者一次性的脉冲信号。
- PWM 每一路输出均可以作为周期精确的中断发生器

### 3.5 PWM 寄存器列表

PWM 的可配置寄存器为存储器地址映射寄存器，PWM 作为从模块挂载在 SoC 的私有设备总线上。每个 PWM 的可配置寄存器列表及其偏移地址如表 3.3 所示。

表3.3 PWM 可配置寄存器列表

寄存器名称	偏移地址	复位默认值	功能描述
PWMCFG	0x000	0x0	PWM配置寄存器
PWMCOUNT	0x008	0x0	PWM计数器计数值寄存器
PWMS	0x010	0x0	PWM计数器比较值寄存器
PWMCMP0	0x020	0x0	PWM比较器寄存器0
PWMCMP1	0x024	0x0	PWM比较器寄存器1
PWMCMP2	0x028	0x0	PWM比较器寄存器2
PWMCMP3	0x02C	0x0	PWM比较器寄存器3

### 3.3 PWM 寄存器配置

#### 1. PWMCFG 寄存器

PWMCFG 寄存器可以用于对 PWM 进行配置，PWMCFG 寄存器的格式如图 3.3 所示。

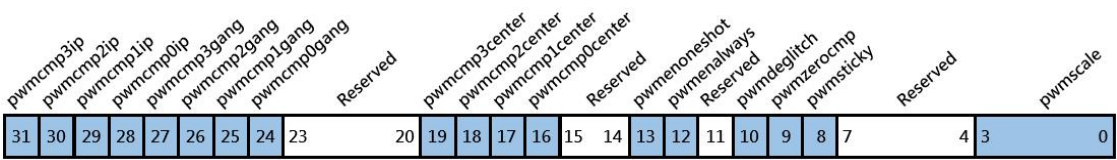


图3.3 PWMCFG 寄存器格式

表 3.4 中给出了在本教程中会用到的 PWMCFG 寄存器的部分比特域。

表3.4 PWMCFG 寄存器部分比特域

域名	比特域	读写属性	复位默认值	描述
pwmcmp3center	19	可读可写	0x0	配置 PWM 第 4 路产生居中对齐的脉冲信号
pwmcmp2center	18	可读可写	0x0	配置 PWM 第 3 路产生居中对齐的脉冲信号
pwmcmp1center	17	可读可写	0x0	配置 PWM 第 2 路产生居中对齐的脉冲信号
pwmcmp0center	16	可读可写	0x0	配置 PWM 第 1 路产生居中对齐的脉冲信号
pwmnoneshot	13	可读可写	0x0	配置 PWM 输出一次性的脉冲信号
pwmalways	12	可读可写	0x0	配置 PWM 输出周期性的脉冲信号
pwmdeglitch	10	可读可写	0x0	如果该域的值被配置为 1，则可以保证即使在软件修改 PWMCMP_X 寄存器时，PWM 的输出也不会产生毛刺反之，如果该域的值被配置为 0，则无法保证
pwmzerocmp	9	可读可写	0x0	如果该域的值被配置为 1，则表示 PWM 计数器的计数值达到比较阈值之后，将会被清零。通过此特性，还可以用于产生精确的周期性中断如果该域的值被配置为 0，则表示 PWM 计数器的计数值达到比较阈值之后，不会被清零，计数器会一直自增加一直到溢出归零

pwmscale	3:0	可读可写	0x0	该域用于指定从 PWMCOUNT 寄存器中取出 pwmcmpwidth 位（作为 PWMS 寄存器的值）的低位起始位置
----------	-----	------	-----	---

## 2. PWMCOUNT 寄存器

PWMCOUNT 是一个可读可写的寄存器，宽度为  $(15 + \text{pwmcmpwidth})$ 。该寄存器反映的是 PWM 计数器的值，譬如 PWM0 的比较器宽度为 8bit，则计数器宽度为 23bit；PWM1 和 PWM2 的比较器宽度为 16bit，则计数器宽度为 31 位。

如果 PWMCFG 寄存器的 pwmenalways 域和 pwmenoneshot 域均没有被配置为 1，则 PWM 计数器处于未被使能状态，计数器的值不会自增。注意：系统上电复位后 PWM 处于未被使能状态。

PWM 计数器在被使能后会每个时钟周期自增加一，达到预定的条件后会归零。从 PWM 开始计数到归零之间的这段时间称为一个 PWM 周期，需要注意以下几点：

- PWM 计数器归零的预定条件如下：
  - 条件一：如果 PWMCFG 寄存器的 pwmzerocmp 域被配置为 1，则当 "PWM 计数器比较值 PWMS 寄存器或其取反的值" 大于或者等于 "PWMCMP0 寄存器设定的比较阈值时"，计数器归零。
  - 条件二：如果 PWMCFG 寄存器的 pwmzerocmp 域被配置为 0，则 PWM 计数器一直自增，直到 PWMS 寄存器反映的值达到最大值（全为1）溢出后归零。
- PWM 计数器归零之后，可以重新开始自增计数，或者停止计数。
  - 如果 PWMCFG 寄存器的 pwmenalways 域被配置为 1，则 PWM 计数器归零后会重新计数。因此软件可以利用配置 PWMCFG 寄存器的 pwmenalways 域来达到产生周期性脉冲信号的效果。
  - 如果 PWMCFG 寄存器的 pwmenoneshot 域被配置为1，则 PWM 计数器归零后硬件也将 pwmenoneshot 域的值清零，并不再重新计数。因此软件可以利用配置 PWMCFG 寄存器的 pwmenoneshot 域来达到只产生一次性脉冲信号的效果。由于产生一次性脉冲信号之后，PWMCFG 寄存器 pwmenoneshot 域的值会被硬件清零。软件可以重新配置 PWMCFG 寄存器 pwmenoneshot 域为 1，再次发送一次性的脉冲信号。

PWMCOUNT 寄存器在系统复位后被清零，且由于 PWMCOUNT 寄存器是可读可写的寄存器，因此软件还可以直接写入此寄存器以改变 PWM 计数器的值。

## 3. PWMS 寄存器

PWMS 寄存器的值来自 PWMCOUNT 寄存器中取出的 pwmcmpwidth 位，如图 3.2 所示，PWMCFG 寄存器 pwmscale 域的值指定了 PWMCOUNT 寄存器中取出 pwmcmpwidth 位的低位起始位置。因此，PWMS 寄存器只是一个只读的影子寄存器，软件对其进行写操作将会被忽略。

如果 PWMCFG 寄存器 pwmscale 域的值0，则意味着直接取出 PWMCOUNT 寄存器低 pwmcmpwidth 位作为 PWMS 寄存器的值；如果 PWMCFG 寄存器 pwmscale 域的值0为最大值 15，则意味着将 PWMCOUNT 寄存器的值除以  $2^{15}$  作为 PWMS 寄存器的值，在这种情况下：

由于 PWM 处于主域的时钟域。譬如，假设主域的时钟频率为 16MHz，按照此频率计算，那么 PWMS 寄存器自增的频率约为 488.3Hz。计算过程为：

$$16000000/2^{15} = 488.28125$$

## 4. PWM 接口数据线

每个 PWM 模块有 4 路输出信号，PWM0、PWM1、PWM2 的输出信号线均通过 GPIO 的 IOF 功能复用 GPIO 引脚，分配情况如表 3.5 所示。

表3.5 GPIO 外设复用分配表

GPIO Pad 编号	IOF0	IOF1		GPIO Pad 编号	IOF0	IOF1
0	-	PWM0_0		16	UART0:RX	-
1	-	PWM0_1		17	UART0:TX	-
2	QSPI1:SS0	PWM0_2		18	-	-
3	QSPI1:SD0/MOSI	PWM0_3		19	-	PWM1_1
4	QSPI1:SD1/MISO	-		20	-	PWM1_0
5	QSPI1:SCK	-		21	-	PWM1_2
6	QSPI1:SD2	-		22	-	PWM1_3
7	QSPI1:SD3	-		23	-	-
8	QSPI1:SS1	-		24	UART1:RX	-
9	QSPI1:SS2	-		25	UART1:TX	-
10	QSPI1:SS3	PWM2_0		26	QSPI:SS	-
11	-	PWM2_1		27	QSPI2:SD0/MOSI	-
12	I2C:SDA	PWM2_2		28	QSPI2:SD1/MISO	-
13	I2C:SCL	PWM2_3		29	QSPI2:SCK	-
14	-	-		30	QSPI2:SD2	-
15	-	-		31	QSPI2:SD3	-

5. 产生左对齐或者右对齐的脉冲信号

如果 PWMCFG 寄存器的 pwmcmpcenter 域被配置为 0，则 PWM 会产生左对齐的脉冲信号波形。  
左对齐脉冲信号波形示例图如图 3.4 所示。

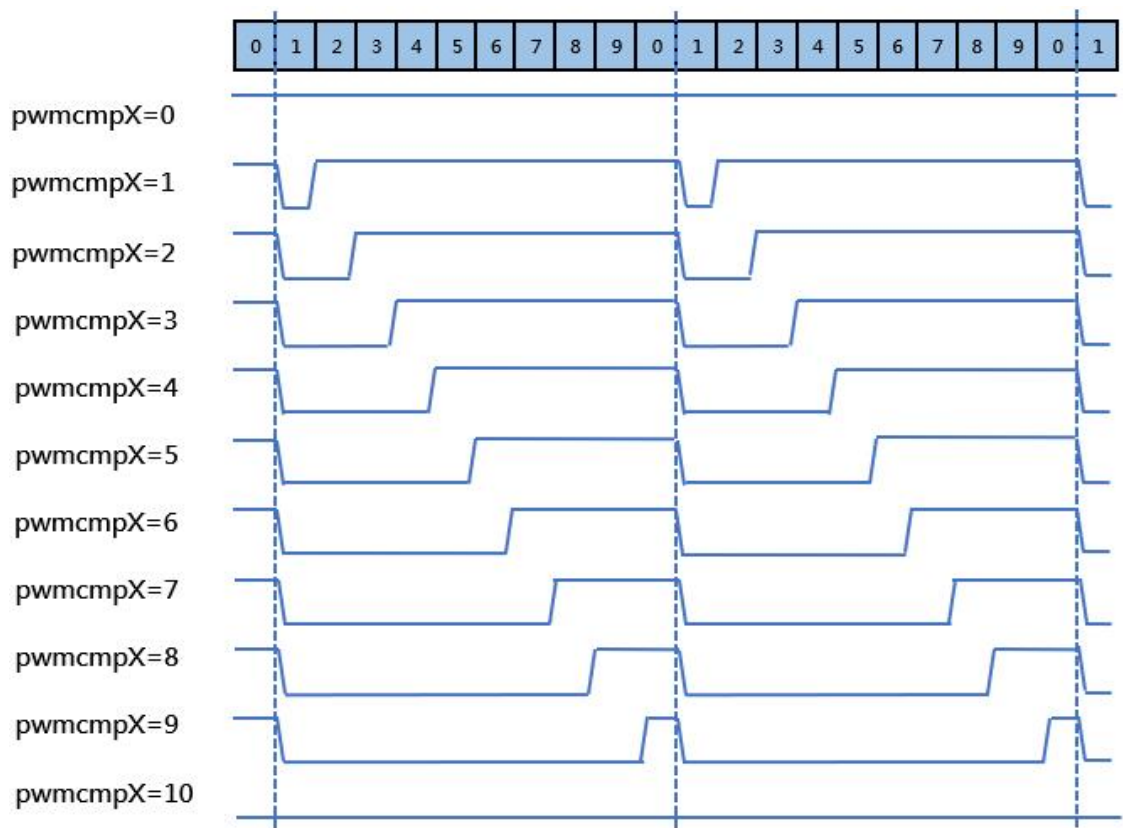


图3.4 PWM 输出左对齐的脉冲波形

此示例要点如下：

- PWMCFG 寄存器的 pwmscale 域的值被配置为 0，因此 PWMS 寄存器的值每个周期自增一。
- PWMCMP0 寄存器被配置为 9，因此一个 PWM 周期为 10 个时钟。
- PWMCFG 寄存器的 pwmenalways 域被配置为 1，因此 PWMS 寄存器归零后 PWM 计数器重新开始计数，输出周期性的脉冲波形信号。
- PWMCFG 寄存器的 pwmzerocmp 域被配置为 1，因此 PWMS 寄存器的值到 9 之后便归零。
- 图中列举了 PWM 的 PWMCMP 寄存器被配置不同值时输出的左对齐脉冲信号波形。该波形相对 PWMS 寄存器值进行比较的结果有一个周期的延迟错位，这是因为在图 3.2 中，PWM 的输出（pwmcmpip 的值）是将比较器输出的结果经过一级寄存器寄存后的结果。

注意：PWMCMP0 已经被配置为 9（用于产生 10 个时钟的周期）。只有 PWMCMP1、PWMCMP2、PWMCMP3 寄存器有可能被配置为其他不同的值。

由于 PWM 的输出信号通过 GPIO 的 IOF 功能复用 GPIO 引脚，而 GPIO 的每个 I/O 可以通过配置 GPIO\_OUTPUT\_XOR 寄存器使得输出信号取反。因此，可以通过配置 GPIO 将左对齐的脉冲信号转换成右对齐的脉冲信号。

## 6. 产生居中对齐的脉冲信号

如果 PWMCFG 寄存器的 pwmcmpcenter 域被配置为 1，则 PWM 会产生居中对齐的脉冲信号波形。

## 7. 配置 pwmdeglitch 防止输出毛刺

如图 3.2 所示，由于 PWM 的输出信号来自 "PWM 的计数器比较值（PWMS 寄存器的值）" 与 "PWMCMP 寄存器设定的比较阈值" 进行比较的结果，因此假设软件再运行的过程中改变 PWMCMP 寄存器的值，那么就可能造成 PWM 的输出信号产生毛刺。为了防止此种情况下产生毛刺，软件可以配置 PWMCFG 寄存器的 pwmdeglitch 域为 1。假设 PWMCFG 寄存器的 pwmdeglitch 域被配置为 1，则：



- 如果是产生居中对齐的脉冲信号，则可以保证在前半个周期只产生一个上升沿，在后半个周期只产生一个下降沿。
- 如果是产生左对齐的脉冲信号，一旦 PWM 的输出（pwmcmpip的值）产生高电平，pwmcmpip 的值会一直保持，直到该轮 PWM 周期结束。

## 3.6 程序流程图

本教材是通过配置 PWM 相关寄存器，产生 PWM 波，输出到 LED 实现呼吸灯的效果。

程序开始时，先对 PWM0 对应的 GPIO 进行初始化，再对 PWM 控制器进行初始化，进入 while 循环，循环修改 PWMCMP1，PWMCMP2，PWMCMP3 的值，改变输出到 LED1-3 的 PWM 波的占空比，使 LED1-3 呈现呼吸灯的效果。

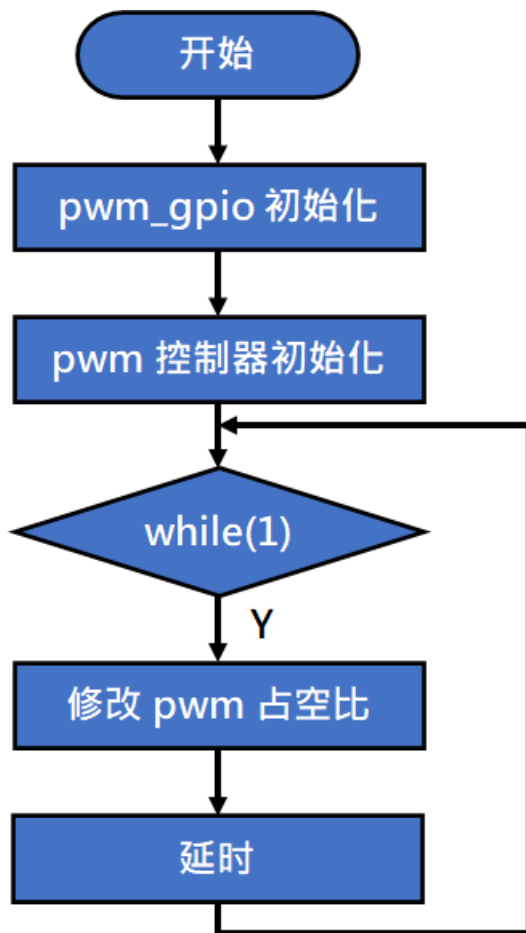


图3.4 PWM 呼吸灯程序流程图

## 四、操作步骤

### 4.1 使用 Makefile 编译和下载应用程序

#### 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\_pwm" 文件夹，所以这个文件夹的绝对路径为：

## 2. 创建程序文件（.c文件）

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

包含相应的头文件，在头文件 platform.h 中，定义了一些外设基地址和外设寄存器的操作函数，如 PWM0\_REG() 和 GPIO\_REG() 等，还包含了外设的寄存器头文件定义，比如 pwm.h，在 pwm.h 中对 PWM 外设的寄存器的偏移地址进行了定义，如 PWM\_CFG、PWM\_COUNT 等。

```
1 | #include <platform.h>
```

定义 PWM0 对应 GPIO 的初始化函数 pwm\_gpio\_init，配置 GPIO\_IOF\_EN 和 GPIO\_IOF\_SEL 寄存器，将 GPIO0-3 复用为 IOF1 模式下的 PWM0\_0-3，这 2 个寄存器在 UART 通讯教程中已经详细讲解过。

```
1 | void pwm_gpio_init(){
2 |     /* Configure PWM0 IO*/
3 |     GPIO_REG(GPIO_IOF_EN) |= IOF1_PWM0_MASK; // IOF En = 1
4 |     GPIO_REG(GPIO_IOF_SEL) |= IOF1_PWM0_MASK; // IOF Sel = 1
5 | }
```

定义 PWM 控制器函数 pwm\_controller\_init，首先将 PWM0 的 PWM\_CFG 寄存器中的 deglitch、enalways、zerocmp 位设置为 1；再将 PWM0 的 CMP0 设置为 9，CMP1、CMP2、CMP3 初值都设置为 0。

```
1 | void pwm_controller_init(void){
2 |     // set pwm deglitch,enalways,zerocmp bits
3 |     PWM0_REG(PWM_CFG) = ( PWM_CFG_DEGLITCH | PWM_CFG_ENALWAYS |
PWM_CFG_ZEROCMP);
4 |
5 |     // pmws LBS increment at 488.3Hz about 2ms
6 |     // set pwm_count value
7 |     PWM0_REG(PWM_COUNT) = 0;
8 |
9 |     // set pwm_cmp0-3 value
10 |    PWM0_REG(PWM_CMP0) = 9;
11 |    PWM0_REG(PWM_CMP1) = 0;
12 |    PWM0_REG(PWM_CMP2) = 0;
13 |    PWM0_REG(PWM_CMP3) = 0;
14 | }
```

定义延时函数 delay。

```
1 | void delay(int s){
2 |     volatile int i=s*1000;
3 |     while(i--);
4 | }
```

定义功能函数 set\_pwm\_cmp123，设置 PWM 的 CMP1，CMP2，CMP3 值为 cmp。

```

1 void set_pwm_cmp123(int cmp){
2     PWM0_REG(PWM_CMP1) = cmp;
3     PWM0_REG(PWM_CMP2) = cmp;
4     PWM0_REG(PWM_CMP3) = cmp;
5 }

```

最后定义程序的主函数，在 main 函数中，首先进行 PWM0 对应 GPIO 的初始化，接着初始化 PWM 控制器，然后定义 cmp\_value 数组，用于设置 PWM 的 CMP1, CMP2, CMP3 值，定义 time 变量，用于 delay 的函数延时。在 while(1) 循环中，将 cmp\_value 的值赋给 PWM 的 CMP1, CMP2, CMP3 寄存器，从而实现 LED1-3 从亮到暗再到亮，循环交替，形成呼吸灯的效果。

```

1 int main(void){
2     // initialize pwm gpio
3     pwm_gpio_init();
4
5     // initialize pwm controller
6     pwm_controller_init();
7
8     // define cmp value for breathing light
9     volatile int cmp_value[21]=
10    {0,1,2,3,4,5,6,7,8,9,10,9,8,7,6,5,4,3,2,1,0};
11
12    int time=500;
13    while(1){
14        // set cmp 0-3 value
15        for (int i=0;i<21;i++){
16            set_pwm_cmp123(cmp_value[i]);
17            delay(time);
18        }
19    }
20    return 0;
21 }

```

### 3. 创建 Makefile 文件

在 "demo\_pwm" 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。

```

1 TARGET = demo_pwm
2 CFLAGS += -O1
3
4 BSP_BASE = ../../bsp
5
6 C_SRCS += demo_pwm.c
7
8 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\_pwm"。

## 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\_pwm" 命令编译应用程序。如图 4.1.1 所示。

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

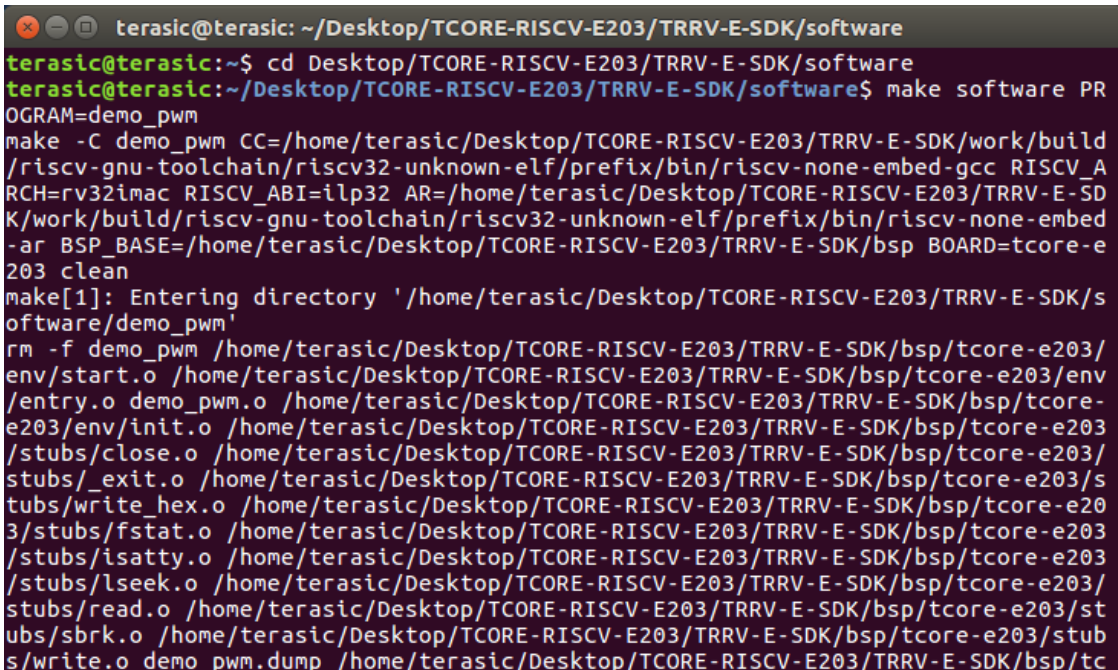


图4.1.1 编译应用程序

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

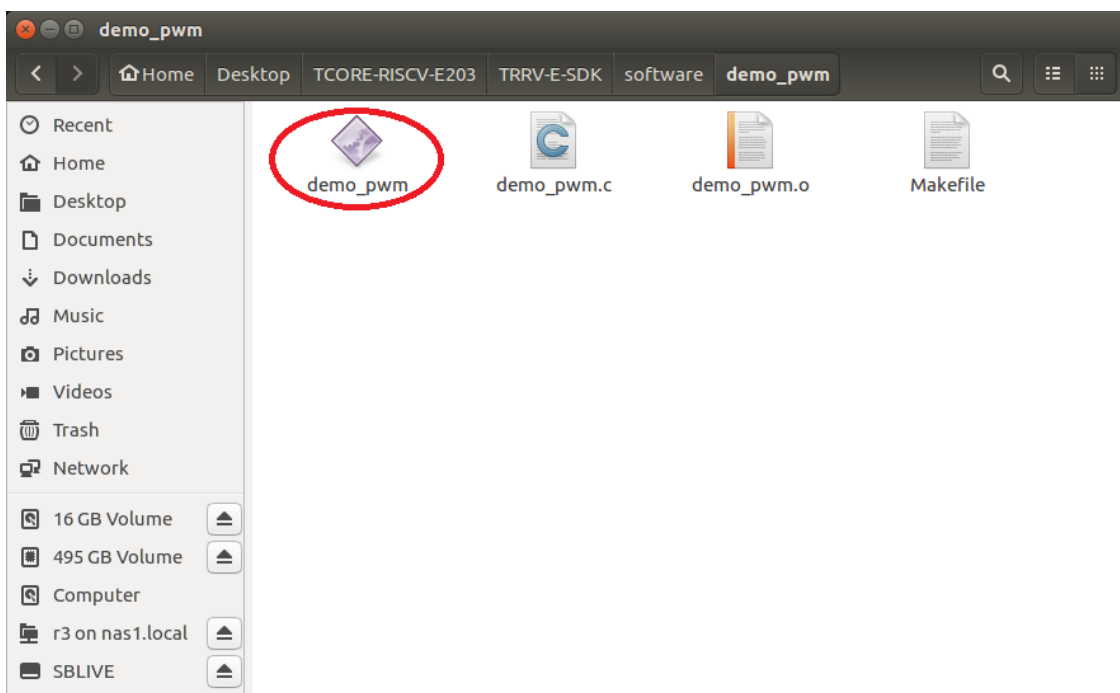


图4.1.2 编译生成二进制文件

### 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。

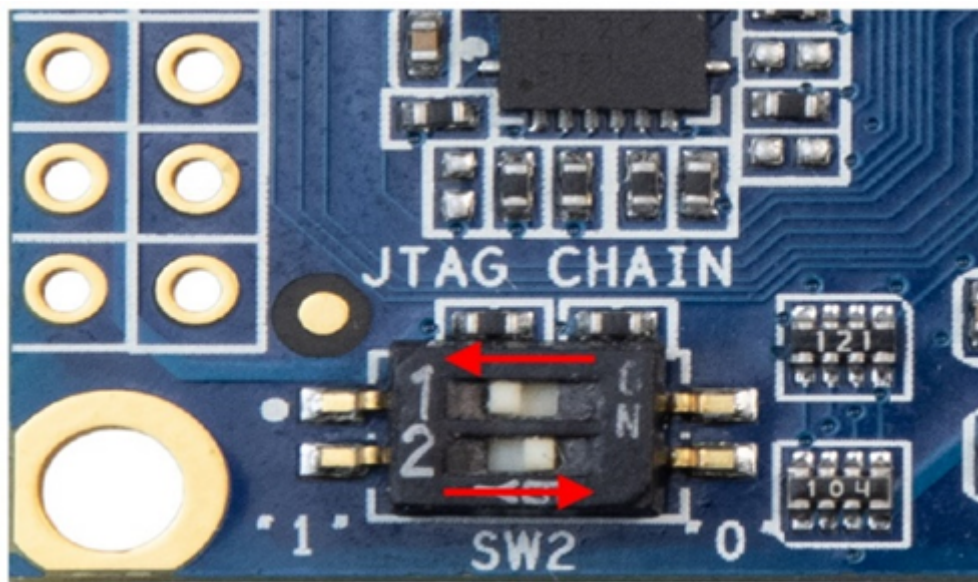


图4.1.3 设置 SW2 开关

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\_pwm" 将可执行文件 "demo\_pwm" 下载到 T-Core 开发板的 QSPI Flash 中。

```
1 | make upload PROGRAM=demo_pwm
```

```
terasic@terasic: ~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software
terasic@terasic:~/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/software$ make upload PROG
RAM=demo_pwm
../work/build/openocd/prefix/bin/openocd -f ../bsp/tcore-e203/env/openocd_tcore.
cfg & \
/home/terasic/Desktop/TCORE-RISCV-E203/TRRV-E-SDK/work/build/riscv-gnu-toolchain
/riscv32-unknown-elf/prefix/bin/riscv-none-embed-gdb demo_pwm/demo_pwm --batch -
ex "set remotetimeout 240" -ex "target extended-remote localhost:3333" -ex "moni
tor reset halt" -ex "monitor flash protect 0 64 last off" -ex "load" -ex "monito
r resume" -ex "monitor shutdown" -ex "quit"
Open On-Chip Debugger 0.10.0+dev-00624-g09016bc (2019-07-16-15:47)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Warn : Adapter driver 'usb_blaster' did not declare which transports it allows;
assuming legacy JTAG-only
Info : only one transport option; autoselect 'jtag'
adapter speed: 4000 kHz
Info : Altera USB-Blaster II found (Firm. rev. = 1.36)
Info : This adapter doesn't support configurable speed
Info : JTAG tap: riscv.cpu tap/device found: 0x1e200a6d (mfg: 0x536 (<unknown>),
part: 0xe200, ver: 0x1)
Info : Examined RISC-V core; XLEN=32, misa=0x40001105
Info : Listening on port 3333 for gdb connections
Info : [0] Found 1 triggers
```

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，LED1、LED2、LED3 亮度一致且由亮变暗再变亮，循环交替，形成呼吸灯的效果。

## 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 创建 demo\_pwm 工程

1. 将文件夹命名由 "blinking\_LED" 修改为 "demo\_pwm"，如图 4.2.1 所示。

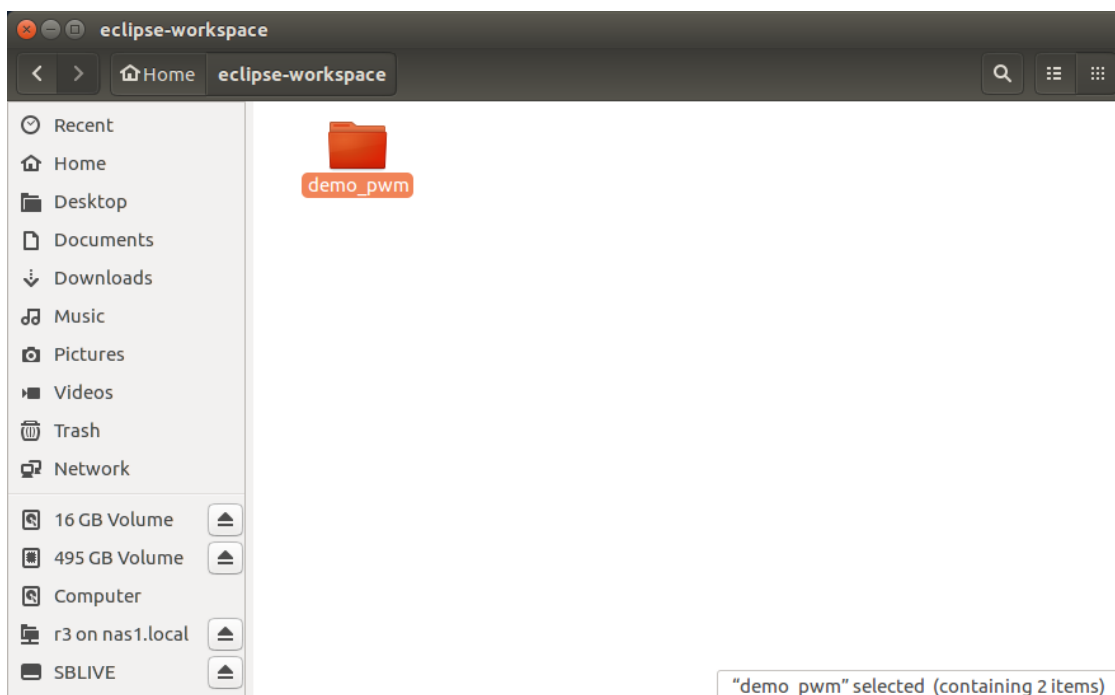


图4.2.1 修改文件名为 "demo\_pwm"



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。

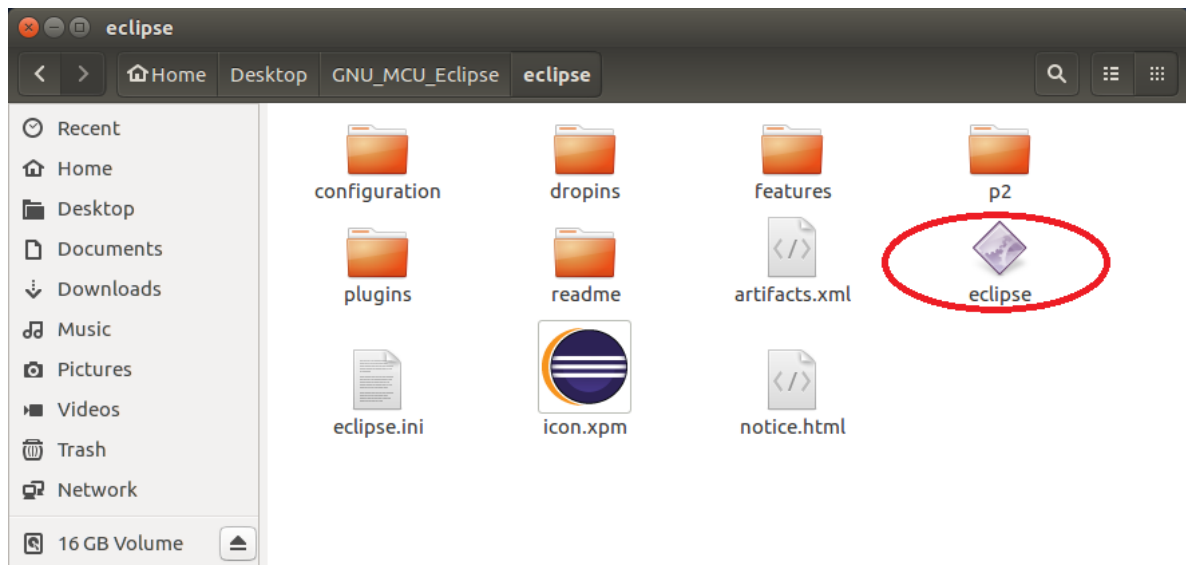


图4.2.2 启动 Eclipse

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）。

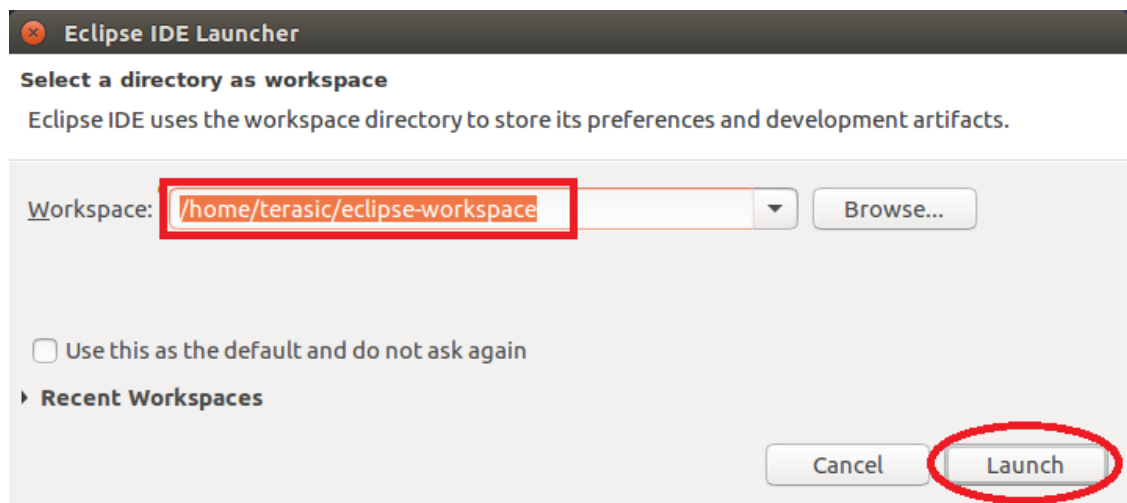


图4.2.3 设置 Workspace

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。

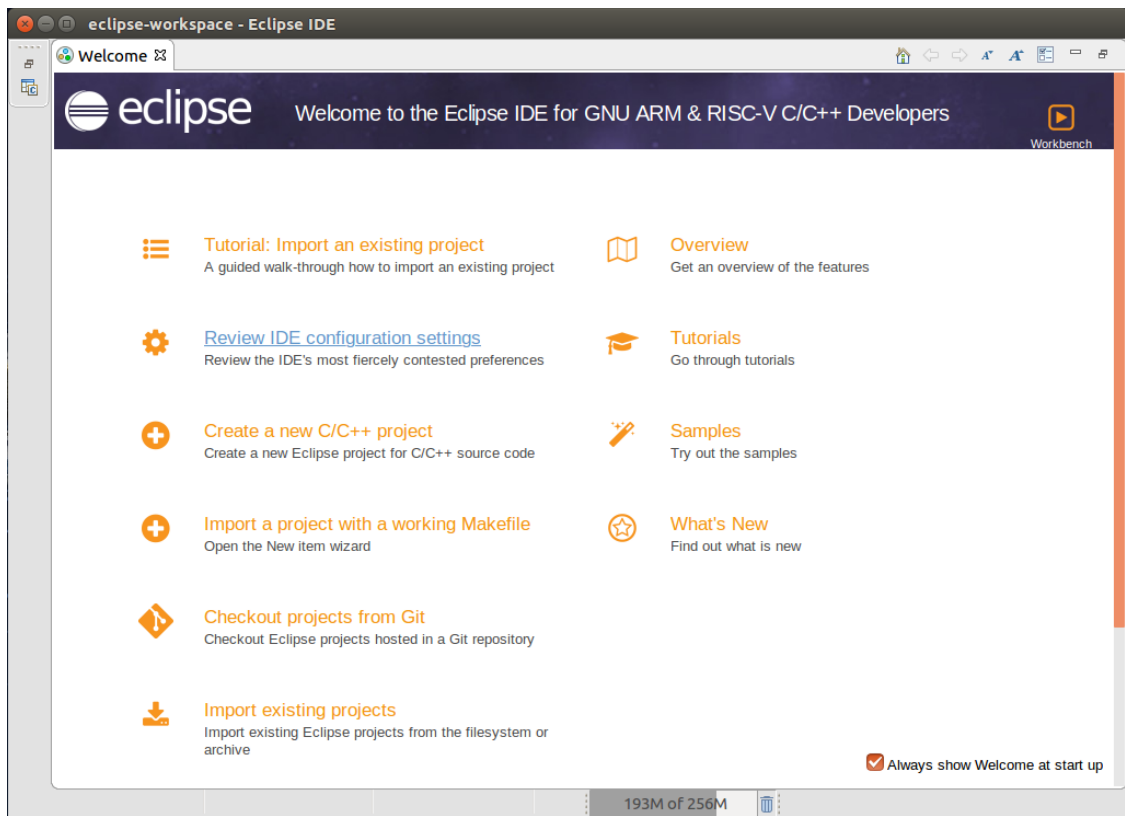


图4.2.4 进入 Eclipse 界面

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

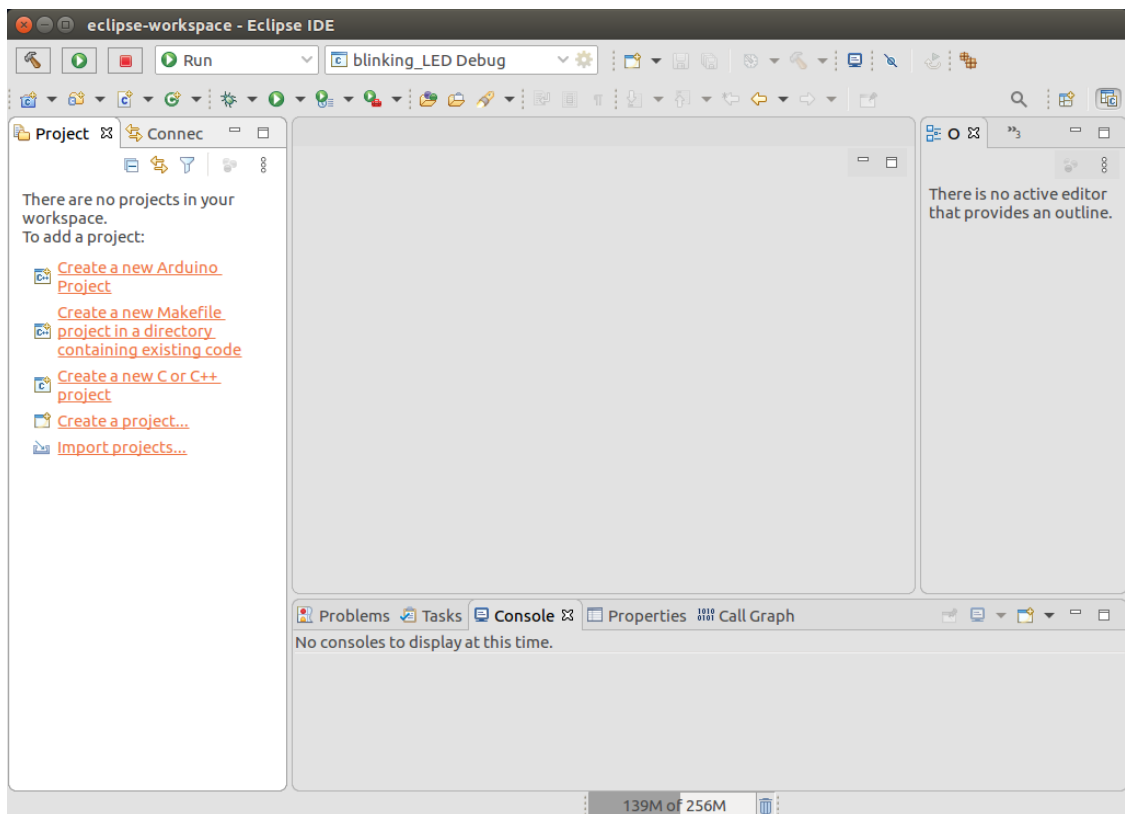


图4.2.5 进入 Eclipse 界面

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。（注：把鼠标移动到顶部菜单栏就会看到 File 选项）



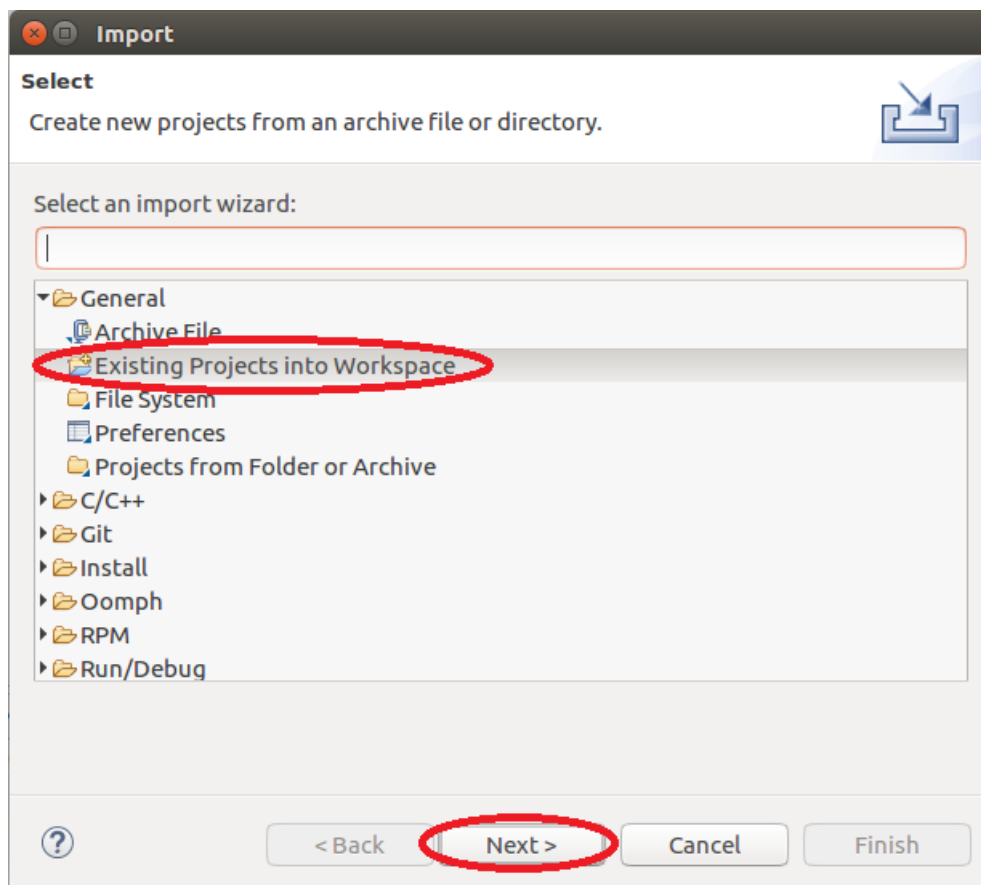


图4.2.6 选择导入工程类型

7. 点击 Browse 导入已有的工程。

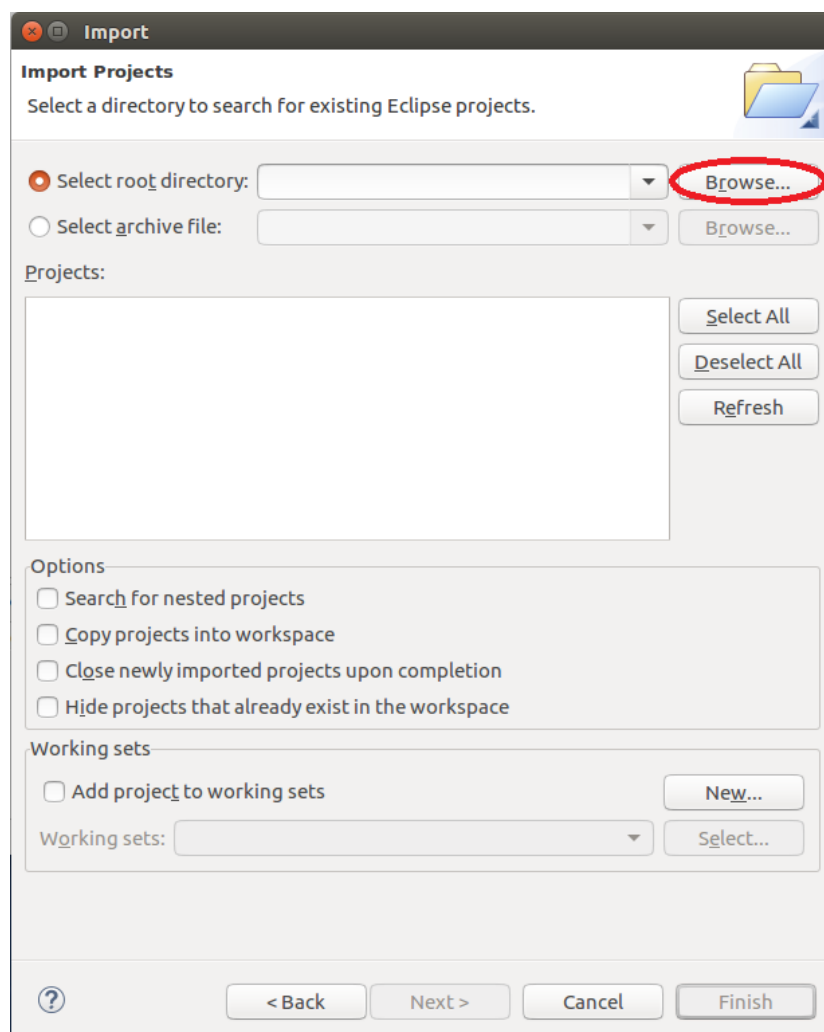


图4.2.7 点击 Browse

8. 选择要添加的 demo\_pwm 工程，点击 OK。

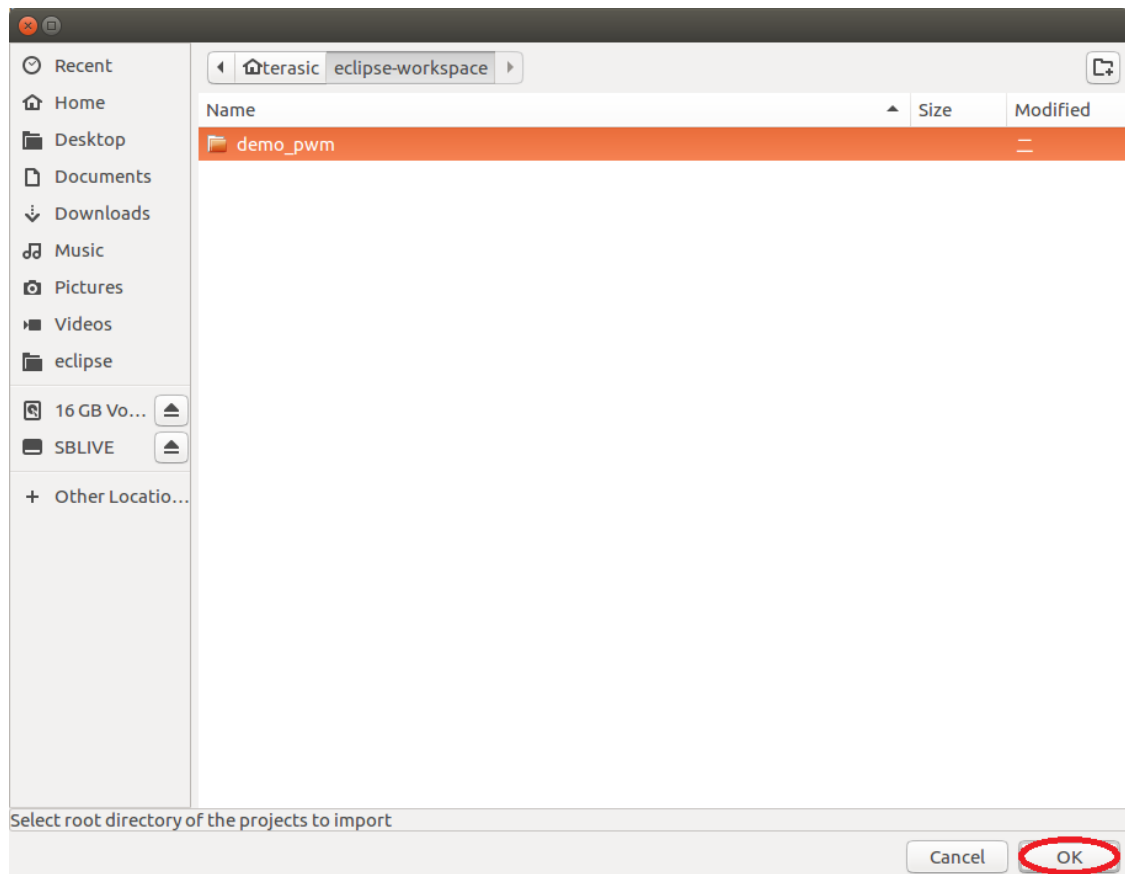


图4.2.8 添加 demo\_pwm 工程

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

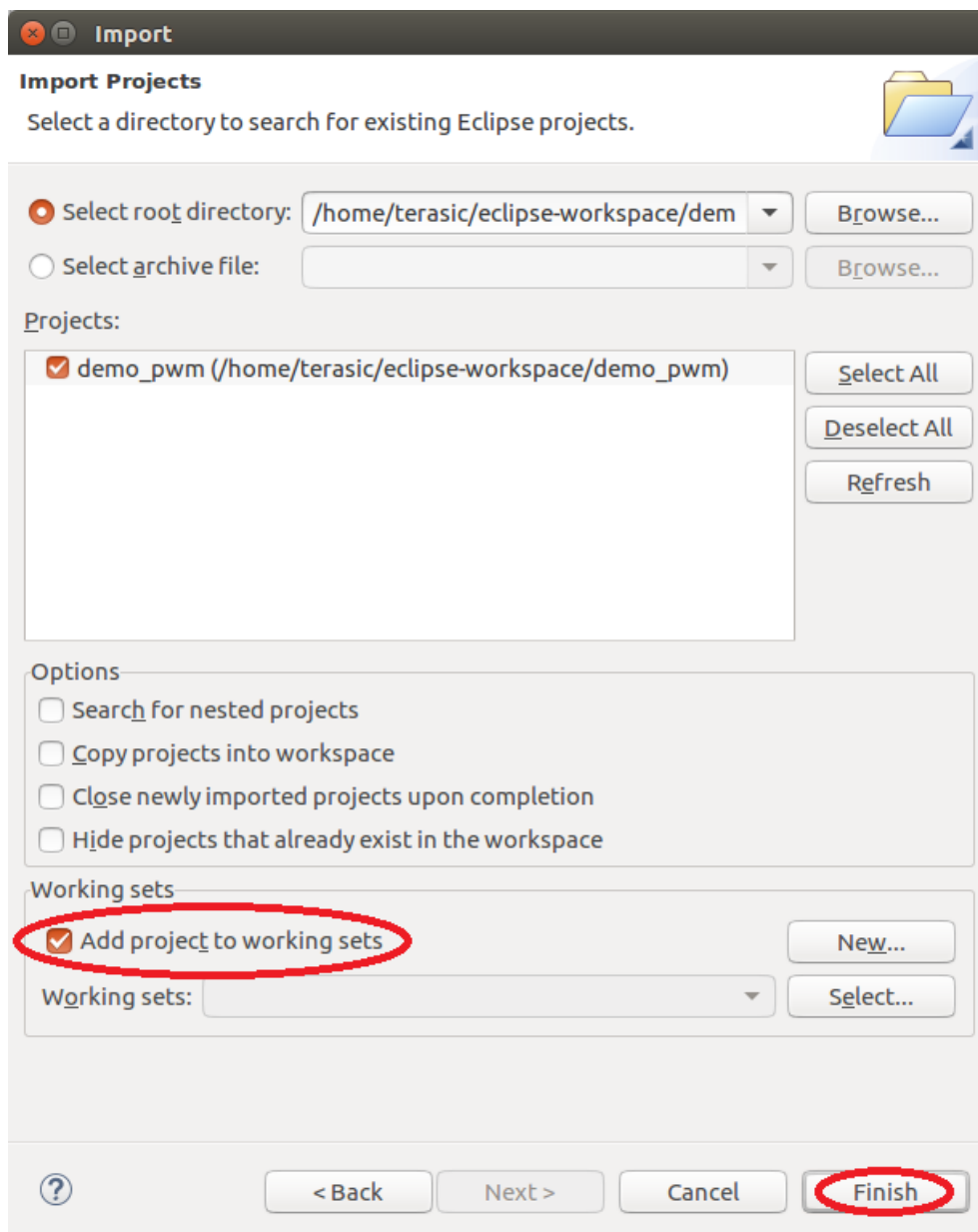


图4.2.9 添加 demo\_pwm 工程到工作空间

10. 导入后的工程界面如图 4.2.10 所示。

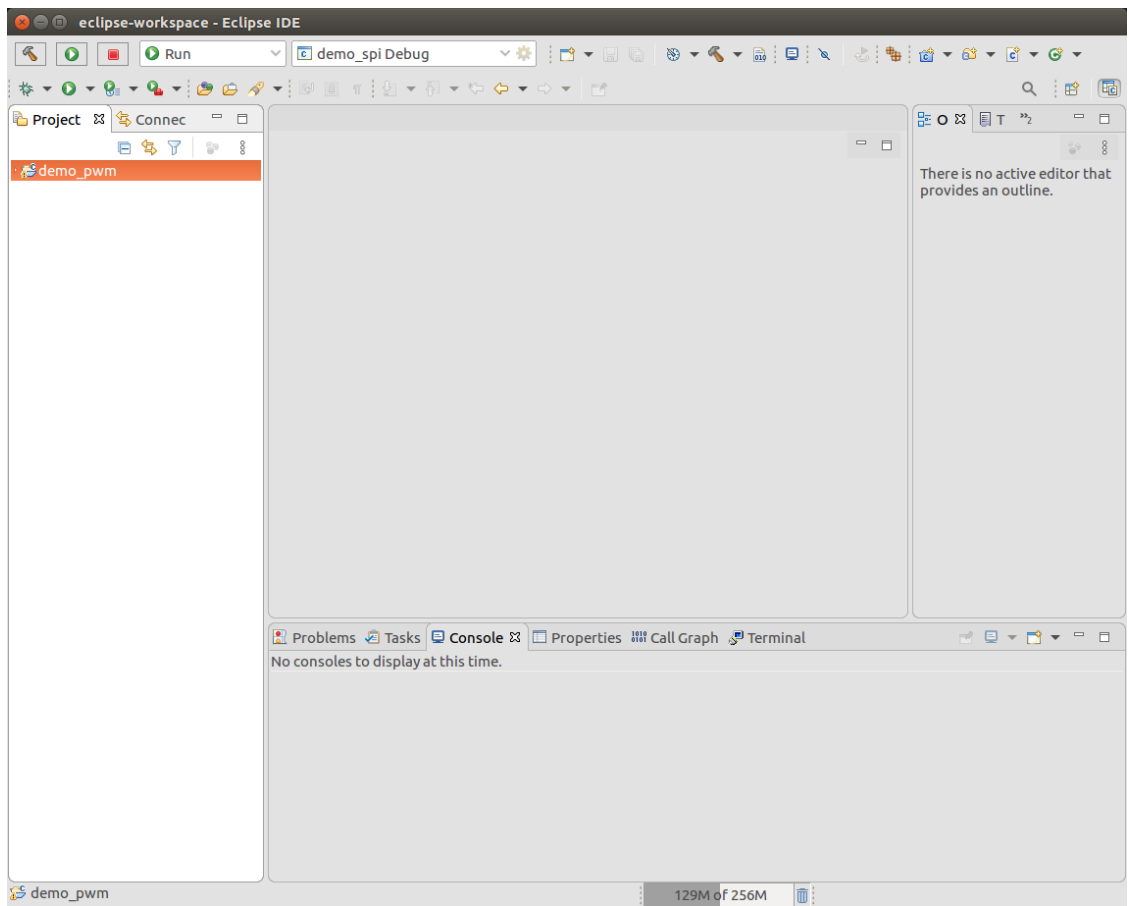


图4.2.10 导入后的工程界面

#### 4.2.2 修改 main.c 文件

点击 `demo_pwm` --> `src` 下拉框，双击打开 `main.c` 文件，复制 4.1.1 节中的 `demo_pwm.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。

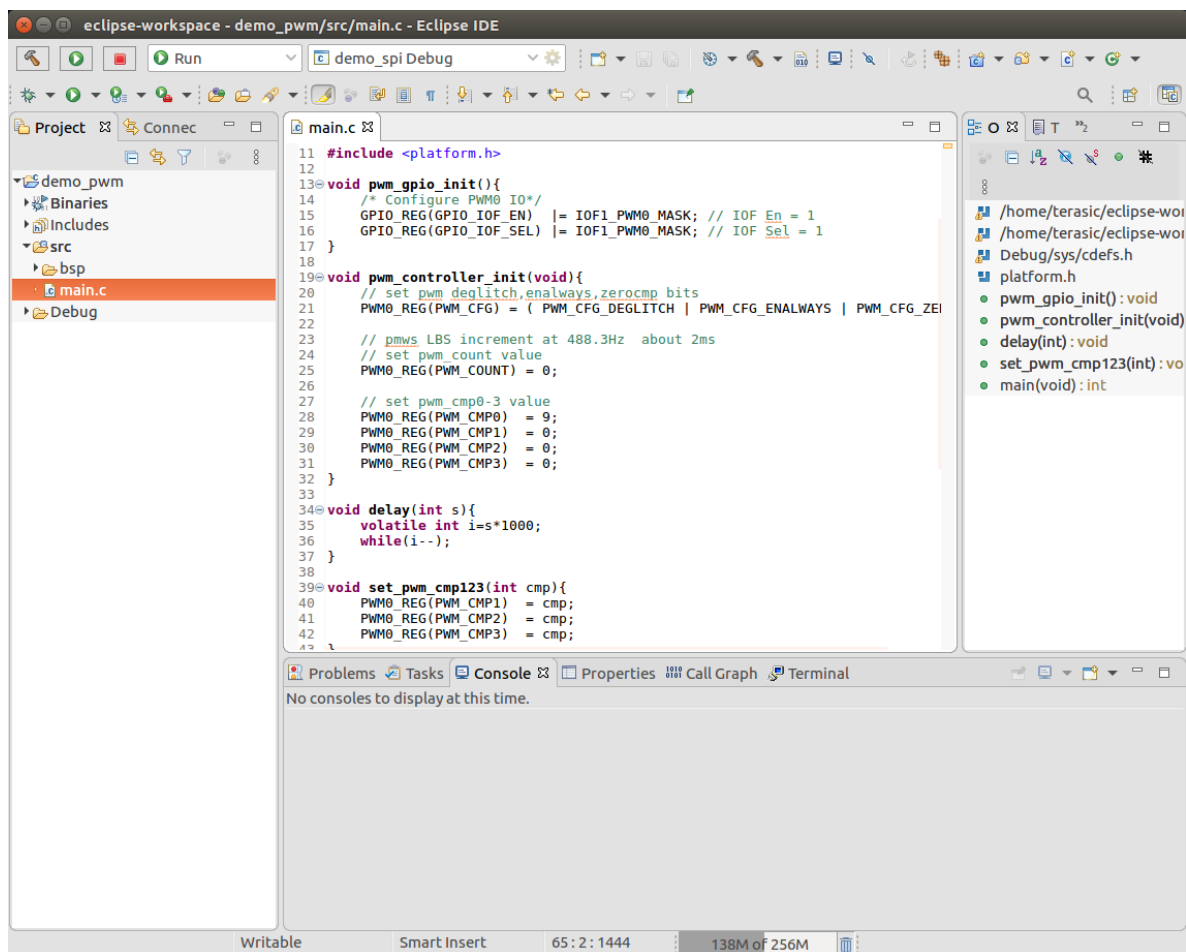


图4.2.11 修改 main.c 文件

### 4.2.3 编译 demo\_pwm 工程

1. 在 Eclipse 主界面中，选中 demo\_pwm 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 4.2.12 所示。

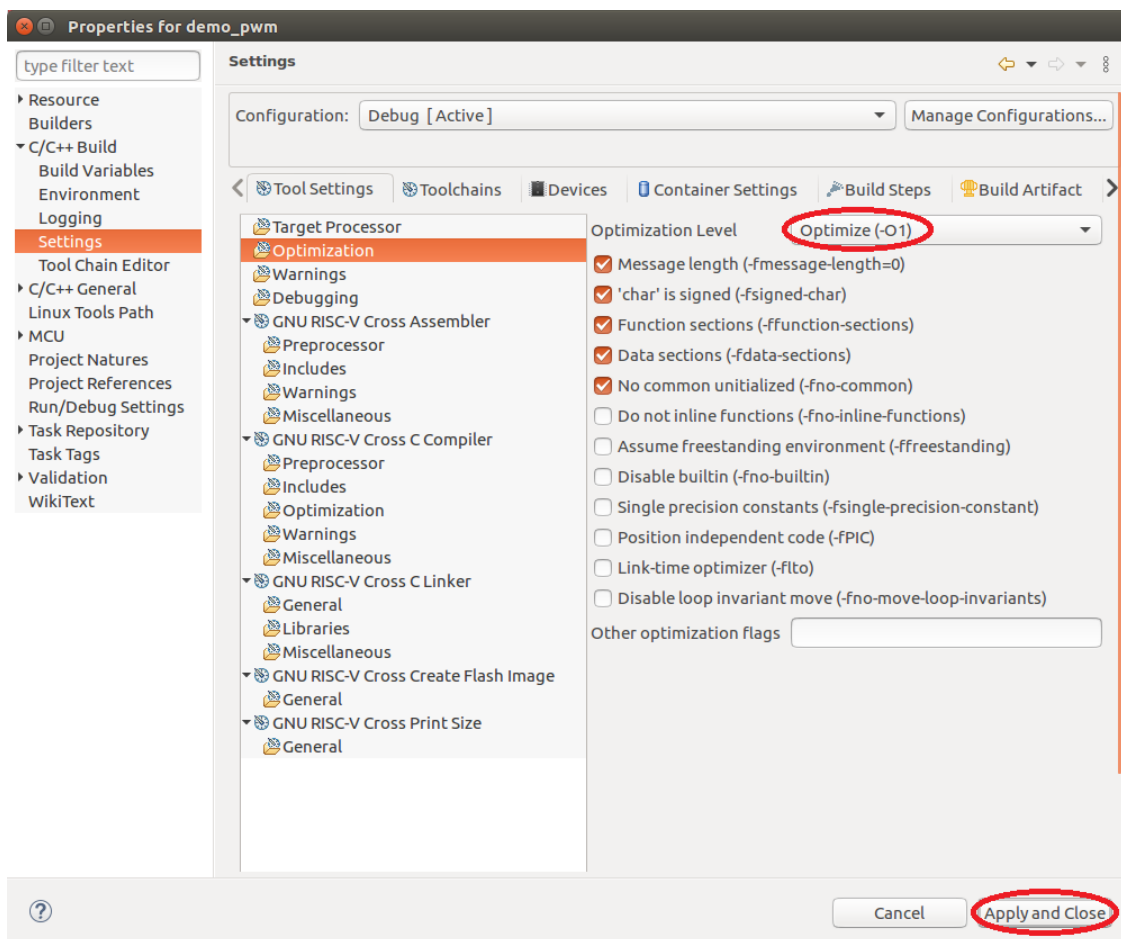


图4.2.12 修改 Optimization level

- 选中 demo\_pwm 工程，右键点击 Clean Project；再次选中 demo\_pwm 工程，右键点击 Build Project，若 demo\_pwm 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 4.2.13 所示。需要右键点击 Refresh 在 Debug 下拉项中看到生成的 .elf 文件。

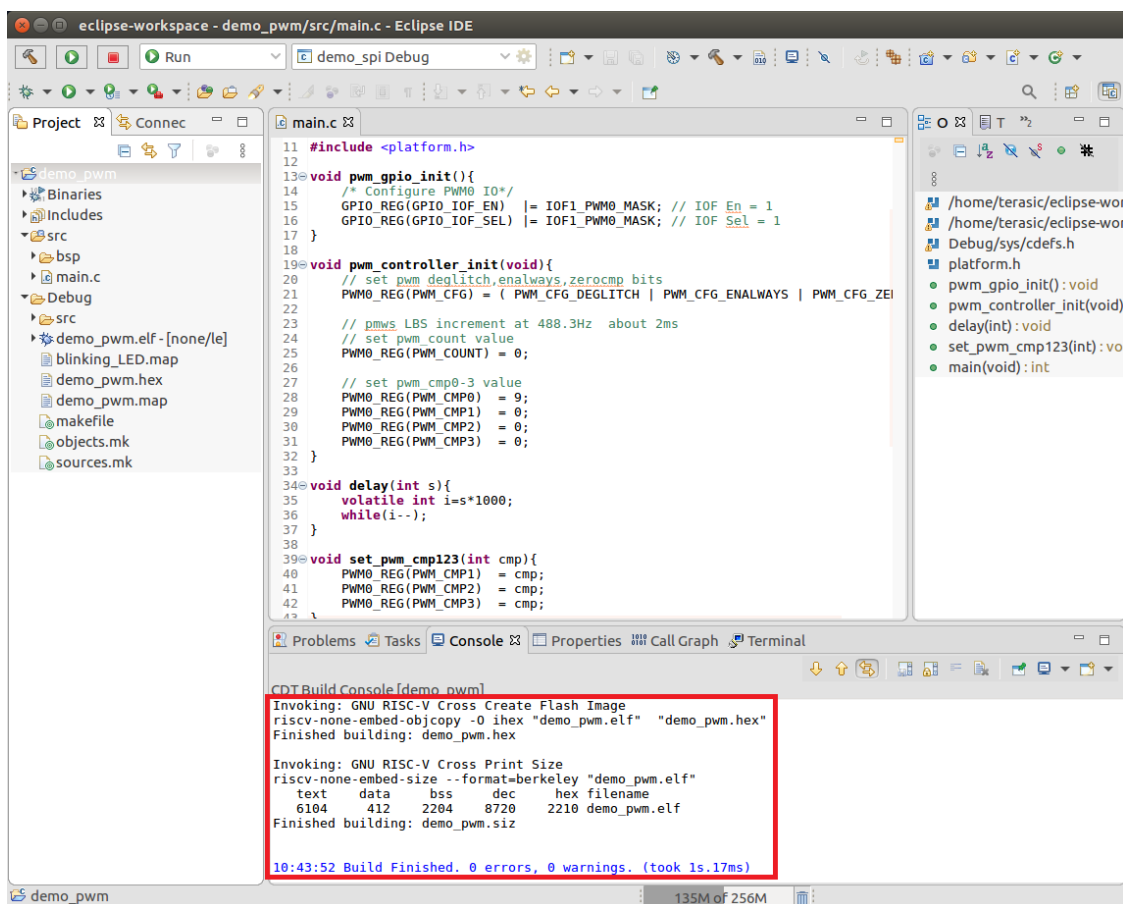


图4.2.13 编译成功

3. 右键 Debug 下拉选项中的 "blinking\_LED.map", 点击 Delete, 删除完成后如图 4.2.14 所示。

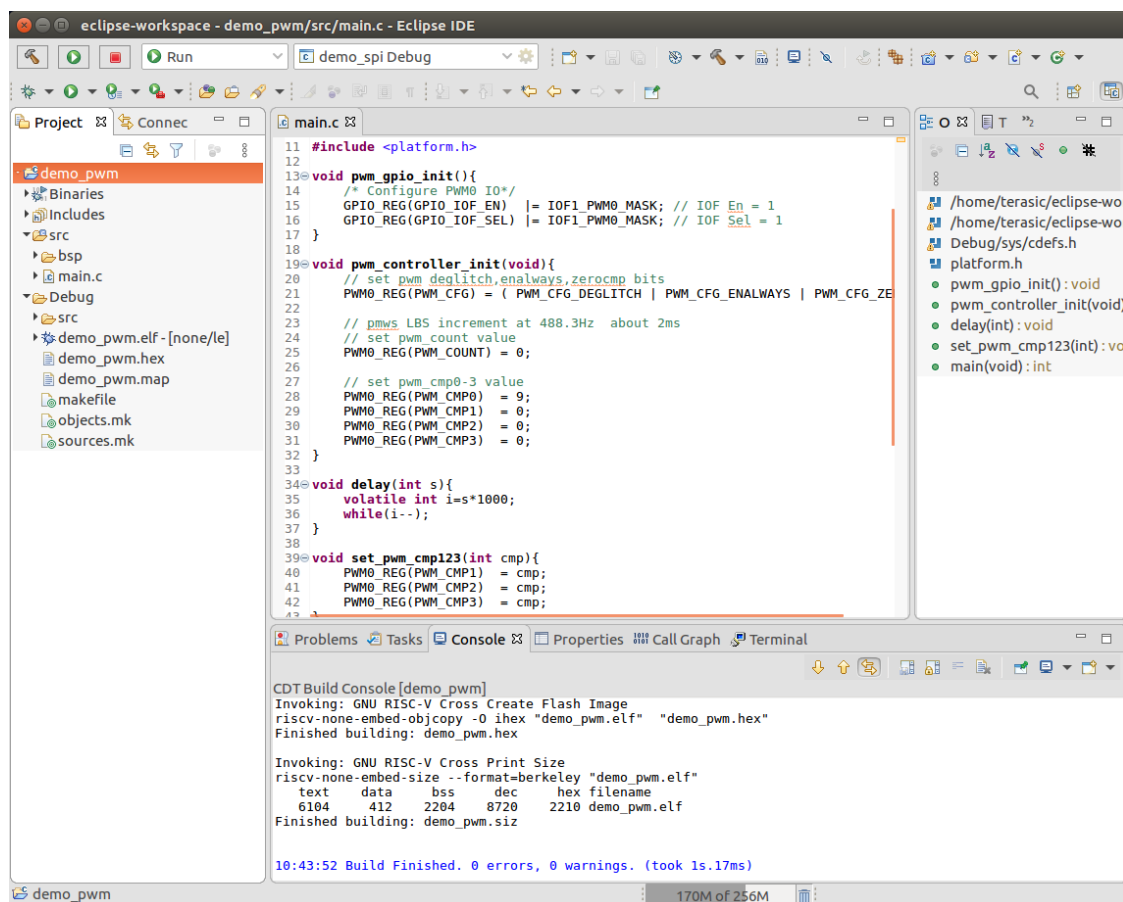


图4.2.14 删除 blinking\_LED.map 文件

## 4.2.4 运行 demo\_pwm 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。

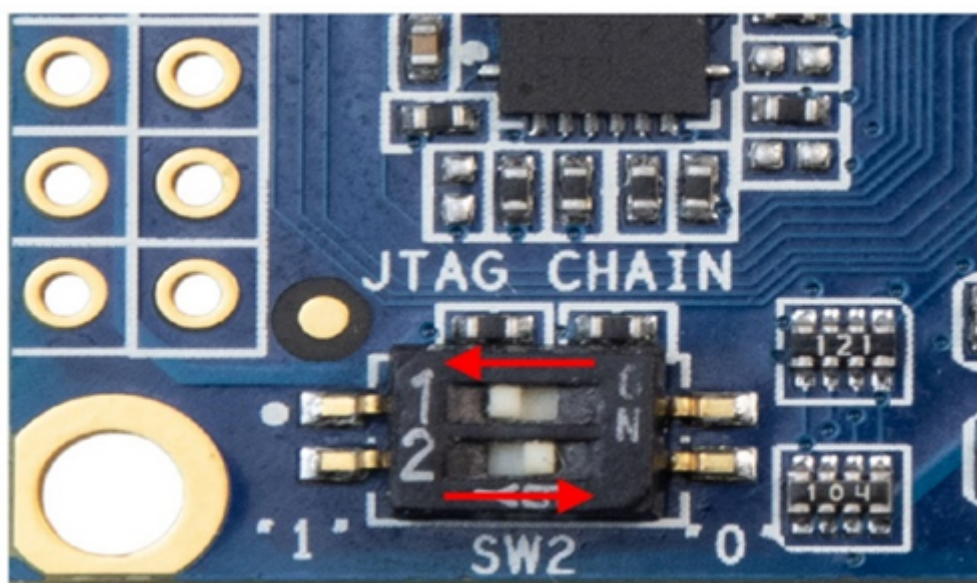


图4.2.15 设置 SW2 开关

- 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\_pwm 工程，右键点击 Run As -> Run Configurations..., 双击 GDB OpenOCD Debugging 会出现如图 4.2.16 所示的 demo\_pwm Debug 界面，在 Config options 中添加 "-f /home/terasic/Desktop/TCORE-RISC-V-E203/TRRV-E-SDK/bsp/tcore-e203/env/openocd\_tcore.cfg" 和 "-s /home/terasic/Desktop/TCORE-RISC-V-E203/TRRV-E-SDK/bsp/tcore-e203/env/"，在 Commands 中添加 "set arch riscv:rv32"，点击 Run 运行 demo\_pwm 工程。

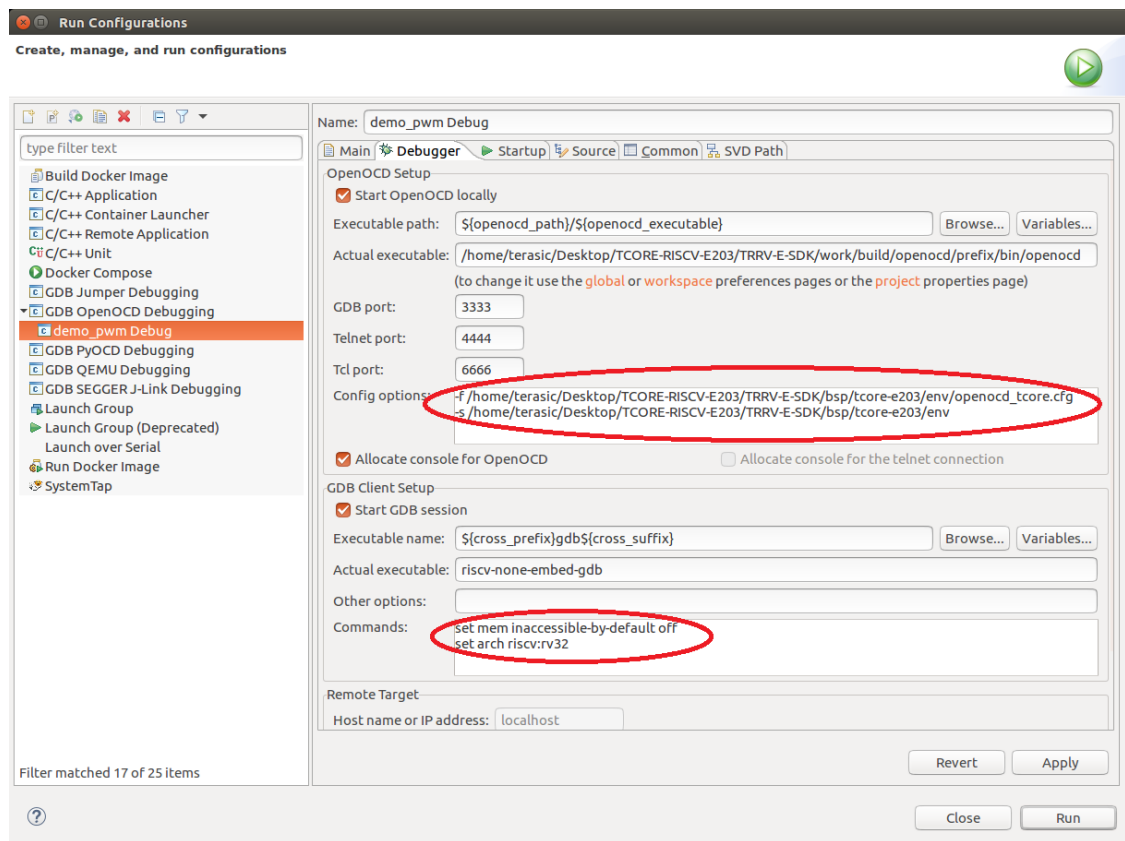


图4.2.16 运行配置

3. 程序下载成功后，如图 4.2.17 所示。

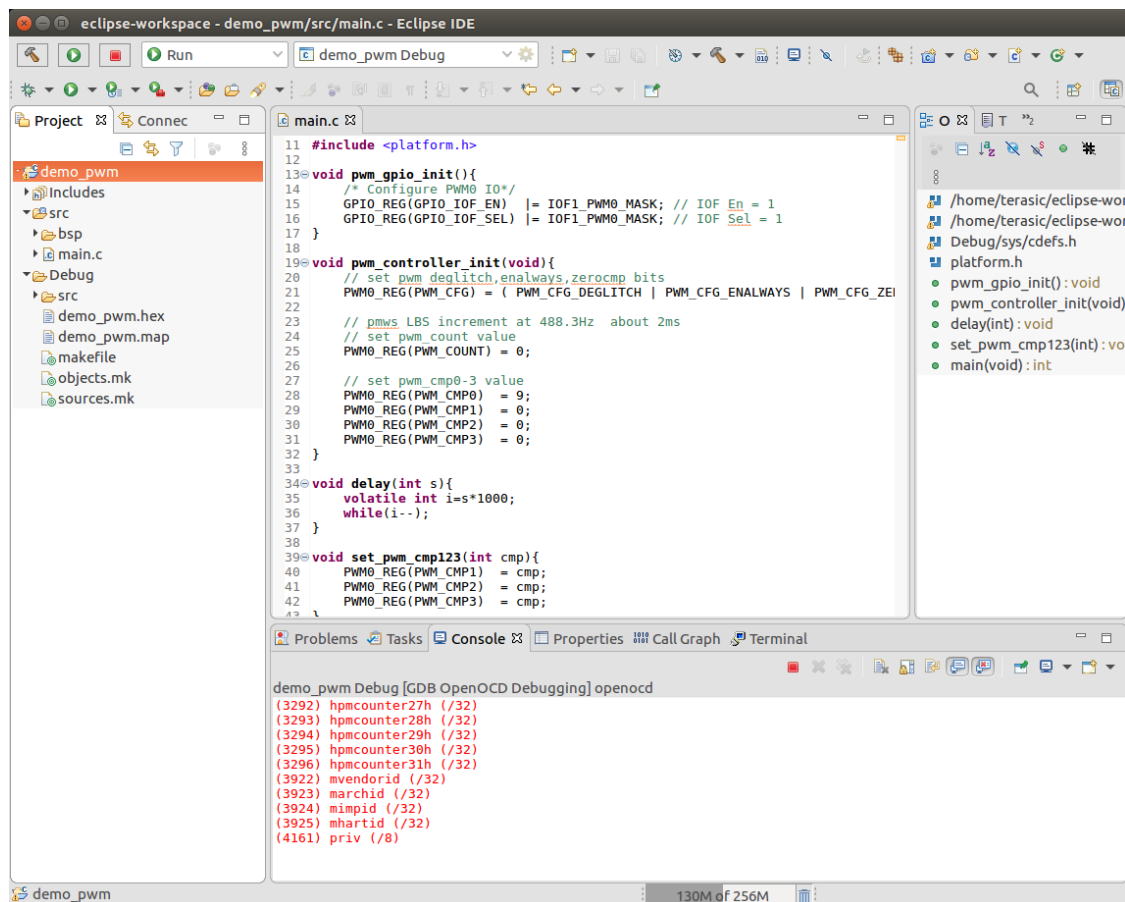




图4.2.17 运行 demo\_pwm 工程

4.2.5 运行结果

程序下载完成后，先按 KEY0 键复位。LED1、LED2、LED3 亮度一致且由亮变暗再变亮，循环交替，形成呼吸灯的效果。

附录

1. 修订历史

版本	时间	修改记录
V1.0	2020.08.10	初始版本

2. 版权声明

本文档为友晶科技自主编写的原创文档，未经许可，不得以任何方式复制或者抄袭本文档之部分或者全部内容。

版权所有，侵权必究。

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](http://www.terasic.com.cn)

邮箱：[support@terasic.com.cn](mailto:support@terasic.com.cn)

微信公众号：



订阅号



服务号