1 呼吸灯

1.1 呼吸灯简介

呼吸灯，就是指灯光设备的亮度随着时间由暗到亮逐渐增强，再由亮到暗逐渐衰减，很有节奏感地一起一伏，就像是在呼吸一样，因而被广泛应用于手机、电脑等电子设备的指示灯中。冰冷的电子设备应用呼吸灯后，顿时增添了几分温暖。

本章介绍如何使用 STM32 定时器的 PWM 输出功能，实现呼吸灯。

1.2 呼吸灯控制原理

要使用数字器件控制灯光的强弱，我们很自然就想到 PWM(脉冲宽度调制)技术。假如以 LED 作为灯光设备，且由控制器输出的 PWM 信号可以直接驱动 LED，PWM 信号中的低电平可点亮 LED 灯。当 LED 以较高的频率进行开关(亮灭)切换时，由于视觉暂留效应，人眼是看不到 LED 灯的闪烁现象的，反映到人眼中能感觉到的是亮度的差别。即以一定的时间长度为周期，LED 灯亮的平均时间越长，亮度就越高，反之越暗。因此，我们可以使用高频率的 PWM 信号，通过调制信号的占空比，控制 LED 灯的亮度。

那么具体我们应该控制 LED 灯以怎样的亮度曲线变化能够达到最好的效果呢？

亮度随着时间逐渐变强再衰减，可以用两种常见的数学函数表示，分别是半个周期的正弦函数与指数上升曲线及其对称得到的下降曲线。见图 1-1。

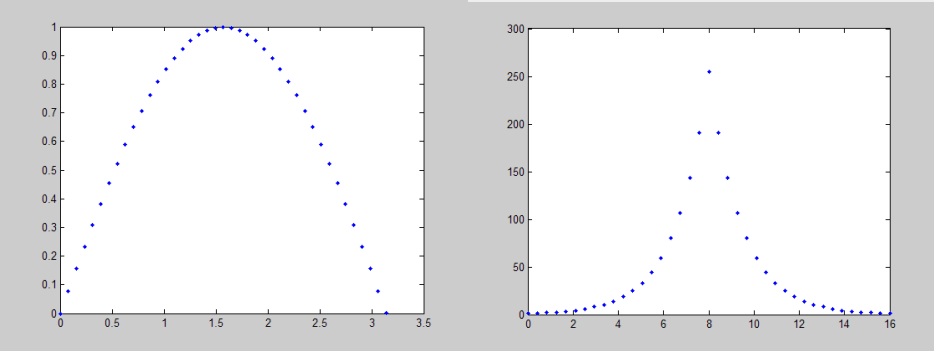


图 1-1 正弦曲线与指数曲线

相对来说，使用下凹函数曲线灯光处于暗的状态更长，所以指数函数的曲线更符合我们呼吸灯的亮度变化要求。

接下来就要确定呼吸灯的呼吸频率(即一个亮度起伏过程)。据统计，成人的一个呼吸周期为 3 秒钟，即吸气时间(亮度上升时间) 1.5 秒，呼气时间(亮度衰减时间)1.5 秒。我们使用定时器即可精确控制它的呼吸频率，当然，读者想把呼吸灯的频率调快或慢一点都是可以的，呼吸周期为 3 秒钟只是一个参考值。

1.3 呼吸灯实验分析

接下来我们通过实例来分析呼吸灯，通过学习这个程序，读者会进一步熟悉 STM32 的定时器和 PWM 技术。

1.3.1 实验描述及工程文件清单

实验描述	使用 STM32 的定时器 3 的第 3 通道控制 ISO 板上的 LED1 实验呼吸灯的效果。
硬件连接	PB0 – LED1　通过跳帽 J37 相连，实验时需要接上跳帽
用到的库文件	*startup/start_stm32f10x_hd.c* *CMSIS/core_cm3.c* *CMSIS/system_stm32f10x.c* *FWlib/stm32f10x_gpio.c* *FWlib/stm32f10x_rcc.c* *FWlib/stm32f10x_tim.c* *FWlib/misc.c*
用户编写的文件	*USER/main.c* *USER/stm32f10x_it.c* *USER/bsp_breathing.c* *USER/key.c*

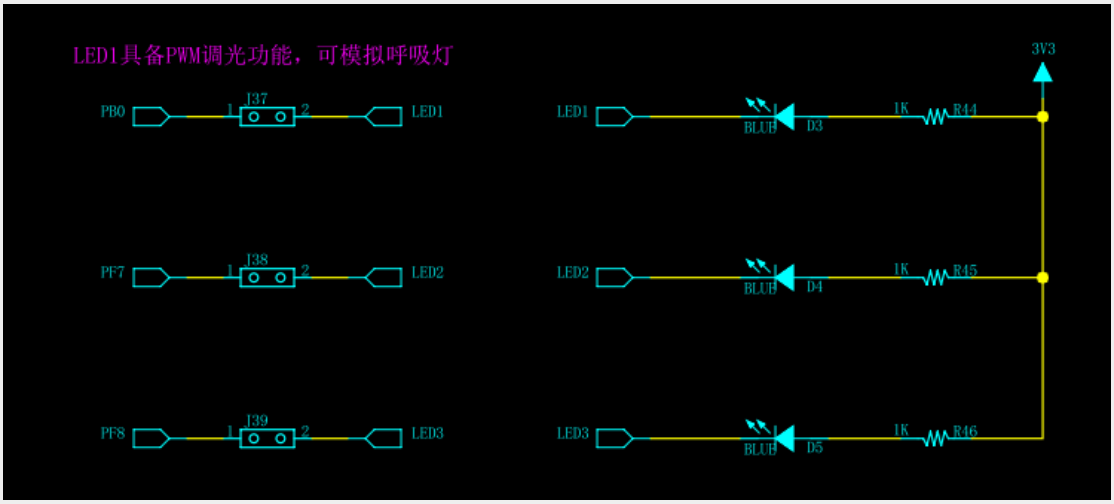


图 1-2 呼吸灯 LED1 硬件原理图

1.3.1 配置工程环境

本定时器呼吸灯实验中我们用到了 GPIO、RCC、TIM 外设，还使用了中断，所以我们先要把以下库文件添加到工程:stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_tim.c、misc.c。

新建 pwm_output.c 及 pwm_output.h 文件，并在 stm32f10x_conf.h 中把使用到的 ST 库的头文件注释去掉。见代码清单 1-1。

代码清单 1-1 呼吸灯的 stm32f10x_conf.h 文件配置

```
1. /**
2.   ******************************************************************
3.   * @file    Project/STM32F10x_StdPeriph_Template/stm32f10x_conf.h
4.   * @author  MCD Application Team
5.   * @version V3.5.0
6.   * @date    08-April-2011
7.   * @brief   Library configuration file.
8.   ******************************************************************/

9. #include "stm32f10x_gpio.h"
10.#include "stm32f10x_rcc.h"
11.#include "stm32f10x_tim.h"
12.#include "misc.h"
13.
```

1.3.2 main 文件

本工程的 main 函数十分简单，仅调用了一个初始化呼吸灯的函数 TIM3_Breathing_Init()，见代码清单 1-2。

代码清单 1-2 呼吸灯的 main 函数

```
#include "stm32f10x.h"
#include "bsp_breathing.h"


/**
  * @brief  主函数
  * @param  无
  * @retval 无
  */
int main(void)
{
    /* 初始化呼吸灯 */
    TIM3_Breathing_Init();

    while(1);

}
```

1.3.3 配置定时器输出 PWM

注：本实验与本系列教程中定时器章节中的 PWM 实验十分类似，关于定时器的部分本章会有所简略，可参考定时器章节进行学习。

初始化呼吸灯的函数 TIM3_Breathing_Init 按步骤调用了为 GPIO 初始化函数 TIM3_GPIO_Config 和定时器模式初始化函数 TIM3_Mode_Config，见代码清单 1-3。

代码清单 1-3 TIM3_Breathing_Init 函数定义

```
/**
  * @brief  TIM3 呼吸灯初始化
  *         配置 PWM 模式和 GPIO
  * @param  无
  * @retval 无
  */
void TIM3_Breathing_Init(void)
{
    TIM3_GPIO_Config();
    TIM3_Mode_Config();
}
```

❑ 生成指数曲线 PWM 数据

要实现 LED 亮度随着指数曲线变化，我们需要使用占空比呈指数曲线变化的 PWM 信号，而这样的信号由定时器经过查表产生。这个表的数据存储在程序中的数组 indexWave 中，见代码清单 1-4。

代码清单 1-4 LED 亮度等级 PWM 表

```
1.  /* LED 亮度等级 PWM 表 */
2.  uint8_t indexWave[] = {1,1,2,2,3,4,6,8,10,14,19,25,33,44,59,80,107,143
    ,191,255,255,191,143,107,80,59,44,33,25,19,14,10,8,6,4,3,2,2,1,1};
```

这个表中的数字意义是什么？我们将要如何使用呢？

把这个表的数据画成图，图 1-3。

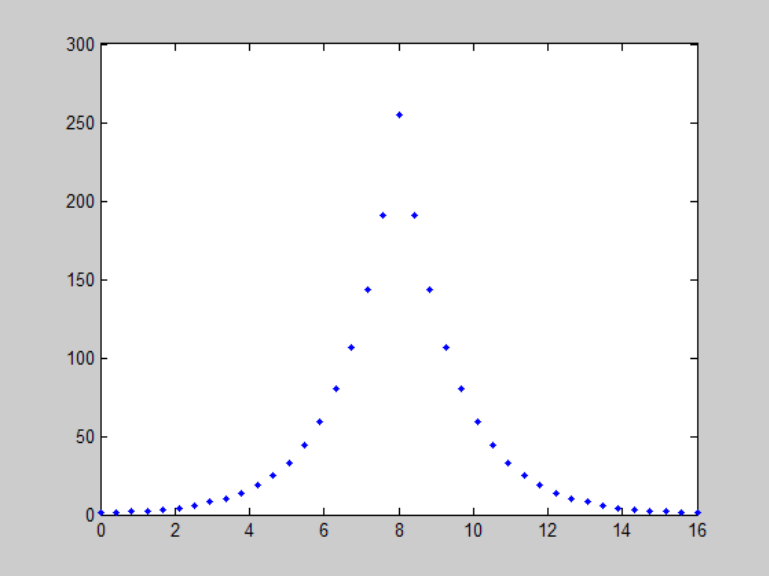


图 1-3 LED 表变化曲线

这个表有 40 个数字，从图中可以看到这些数字呈指数上升再衰减，正好是呼吸灯的一个控制周期。数字的大小范围是 0~255，即把 LED 的亮度分为了 0~255 个等级。

假如我们把定时器的脉冲计数器 TIMx_CNT 上限设置为 255，把这个表的数据一个一个地赋值到定时器的比较寄存器 TIMx_CCR 中，那么在每个 PWM 周期中，当 TIMx_CNT 的计数值小于比较寄存器 TIMx_CCR 的值时， 就会在通

道中输出低电平，点亮 LED，而随着 TIMx_CCR 的值由 LED 亮度表得来，所以 LED 点亮的时间就会呈图中的曲线变化，实现呼吸灯的功能。

这个表的数据是野火使用 matlab 软件生成的，该 matlab 代码见代码清单 1-5，可供读者参考。该代码运行后会生成一个“index_wave.c”的文件，用户把该文件中的数据复制到工程中的数组中即可。

代码清单 1-5 用于生成 LED 亮度曲线的 matlab 代码

```
1.
2.  %本代码用于产生呼吸灯使用的指数函数数据
3.  clear;
4.
5.  x = [0 : 8/19 : 8];          %设置序列 ，指数上升
6.  up = 2.^x ;                  %求上升指数序列
7.  up = uint8(up);              %化为 8 位数据
8.
9.  y = [8: -8/19 :0];           %设置序列 ，指数下降
10. down = 2.^y ;                %求下降指数序列
11. down = uint8(down);          %化为 8 位数据
12.
13. line = [[0:8/19:8],[8:8/19:16]]          %拼接序列
14. val = [up , down]                        %拼接输出序列
15.
16. dlmwrite('index_wave.c',val);          %输出到文件 index_wave.c
17. plot(line,val,'.');                    %显示波形图
18.
```

❑ 初始化 GPIO

下面分析具体的定时器配置代码。本实验使用 PB0 作为定时器 PWM 输出通道，先对它进行初始化。作 PWM 输出通道的引脚需要被配置为复用推挽输出模式。见 代码清单 1-6。

代码清单 1-6 初始化作为 PWM 通道 的 I/O

```
1.   /**
2.    * @brief  配置 TIM3 复用输出 PWM 时用到的 I/O
3.    * @param  无
4.    * @retval 无
5.    */
6.  static void TIM3_GPIO_Config(void)
7.  {
8.    GPIO_InitTypeDef GPIO_InitStructure;
9.
10.   /* GPIOB clock enable */
11.   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);
```

```
12.
13.   /* 配置呼吸灯用到的 PB0 引脚 */
14.   GPIO_InitStructure.GPIO_Pin =  GPIO_Pin_0 ;
15.   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP; // 复用推挽输出
16.   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
17.
18.   GPIO_Init(GPIOB, &GPIO_InitStructure);
19. }
```

❑ **配置定时器模式**

在 TIM3_Mode_Config 函数中，完成了呼吸灯所需要的定时器 PWM 输出模式配置，见代码清单 1-7。

代码清单 1-7 TIM3_Mode_Config 函数定义

```
1. /**
2.   * @brief  配置 TIM3 输出的 PWM 信号的模式，如周期、极性
3.   * @param  无
4.   * @retval 无
5.   */
6. static void TIM3_Mode_Config(void)
7. {
8.     TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
9.     TIM_OCInitTypeDef  TIM_OCInitStructure;

10.   //PCLK1 经过 2 倍频后作为 TIM3 的时钟源等于 72MHz
11.   //使能 TIM3 时钟
12.   RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

13.   /* 基本定时器配置 */
14.   //当定时器从 0 计数到 255，即为 266 次，为一个定时周期
15.   TIM_TimeBaseStructure.TIM_Period = 255;

16.   //设置预分频 2000 分频
17.   TIM_TimeBaseStructure.TIM_Prescaler = 1999;

18.   //设置时钟分频系数：不分频(这里用不到)
19.   TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1 ;

20.   //向上计数模式
21.   TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
22.
23.   TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);
24.
25.   /* PWM 模式配置 */
26.   //配置为 PWM 模式 1
27.   TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;

28.   //使能输出
29.   TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
30.
31.   //设置初始 PWM 脉冲宽度为 0
32.   TIM_OCInitStructure.TIM_Pulse = 0;
```

```
33.   //当定时器计数值小于 CCR1_Val 时为低电平
34.   TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_Low;
35.
36.   //使能通道 3
37.   TIM_OC3Init(TIM3, &TIM_OCInitStructure);

38.   //使能预装载
39.   TIM_OC3PreloadConfig(TIM3, TIM_OCPreload_Enable);

40.   //使能 TIM3 重载寄存器 ARR
41.   TIM_ARRPreloadConfig(TIM3, ENABLE);

42.   //使能定时器 3
43.   TIM_Cmd(TIM3, ENABLE);

44.   //使能 update 中断
45.   TIM_ITConfig(TIM3,TIM_IT_Update, ENABLE);

46. //配置中断优先级
47.     NVIC_Config_PWM();

48.
49. }
```

这个定时器的模式配置主要分为三个部分，分别为时基初始化，输出模式初始化和中断配置。

- 时基初始化

代码中的第 10~23 行是定时器的时基初始化，这部分主要负责配置定时器的定时周期、时钟频率、计数方式等。它使用到库函数 TIM_TimeBaseInit，利用结构体 TIM_TimeBaseInitTypeDef 进行配置，该结构体有以下成员：

1) TIM_Period

定时周期，实质是存储到重载寄存器 TIMx_ARR 的数值，脉冲计数器从 0 累加到这个值上溢或从这个值自减至 0 下溢。这个数值加 1 然后乘以时钟源周期就是实际定时周期。

本实验中向该成员赋值为 255，即定时周期为(255+1)* T ，T 为定时器的时钟周期。

2) TIM_Prescaler

对定时器时钟 TIMxCLK 的预分频值，分频后作为脉冲计数器 TIMx_CNT 的驱动时钟，得到脉冲计数器的时钟频率为：$f_{CK_CNT}=f_{TIMx_CLK}/(N+1)$，其中 N 为即为赋给本成员的时钟分频值。

本实验给 TIM_Prescaler 成员赋值为 1999，即对时钟 2000 分频，所以定时器的时钟周期 T 为 2000/72000000。

注：本实验中的 TIM_Prescaler 的值在下一小节给出具体说明。

3) TIM_ClockDivision

时钟分频因子。怎么又出现一个配置时钟分频的呢？要注意这个 TIM_ClockDivision 和上面的 TIM_Prescaler 是不一样的。TIM_Prescaler 预分频配置是对 TIMxCLK 进行分频，分频后的时钟被输出到脉冲计数器 TIMx_CNT，而 TIM_ClockDivision 虽然也是对 TIMxCLK 进行分频，但它分频后的时钟频率为 f_{DTS}，是被输出到定时器的 ETRP 数字滤波器部分，会影响滤波器的采样频率。TIM_ClockDivision 可以被配置为 1 分频（$f_{DTS}=f_{TIMxCLK}$）、2 分频及 4 分频。ETRP 数字滤波器的作用是对外部时钟 TIMxETR 进行滤波。

本实验中是使用内部时钟 TIMxCLK 作为定时器时钟源的，没有进行滤波所以配置 TIM_ClockDivision 为任何数值都没有影响。

4) TIM_CounterMode

本成员配置的为脉冲计数器 TIMx_CNT 的计数模式，分别为向上计数，向下计数，及中央对齐模式。向上计数即 TIMxCNT 从 0 向上累加到 TIM_Period 中的值，（重载寄存器 TIMx_ARR 的值），产生上溢事件；向下计数则 TIMxCNT 从 TIM_Period 的值累减至 0，产生下溢事件。而中央对齐模式则为向上、向下计数的合体，TIMxCNT 从 0 累加到 TIM_Period 的值减 1 时，产生一个上溢事件，然后向下计数到 1 时，产生一个计数器下溢事件，再从 0 开始重新计数。

本实验中 TIM_CounterMode 成员被赋值为 TIM_CounterMode_Up(向上计数模式)。

填充完配置参数后，调用库函数 TIM_TimeBaseInit()把这些控制参数写到寄存器中，定时器的时基配置就完成了。

● 输出模式配置

在本函数代码的第 25~37 行是关于定时器的输出模式配置的。通用定时器的输出模式由 TIM_OCInitTypeDef 类型结构体的以下几个成员来设置：

1) TIM_OCMode

输出模式配置，主要使用的为 PWM1 和 PWM2 模式。

PWM1 模式是：在向上计数时，当 TIMx_CNT<TIMx_CCRn (比较寄存器，其数值等于 TIM_Pulse 成员的内容) 时，通道 n 输出为有效电平，否则为无效电平；在向下计数时，当 TIMx_CNT>TIMx_CCRn 时通道 n 为无效电平，否则为有效电平。PWM2 模式跟 PWM1 模式相反。

其中的有效电平和无效电平并不是固定地对应高电平和低电平，也是需要配置的，由下面介绍的 TIM_OCPolarity 成员配置。

本实验中使用 PWM1 输出模式。

2) TIM_OutputState

配置输出模式的状态使能或关闭输出。

本实验中向该成员赋值为 TIM_OutputState_Enable(使能输出)。

3) TIM_OCPolarity

有效电平的极性，把 PWM 模式中的有效电平设置为高电平或低电平。

本实验中向该成员赋值为 TIM_OCPolarity_Low(有效电平为低电平)，因为在上面把输出模式配置为 PWM1 模式，向上计数，所以在 TIMx_CNT<TIMx_CCRn 时，通道 n 输出为低电平，否则为高电平。

4) TIM_Pulse

直译为跳动，本成员的参数值即为比较寄存器 TIMx_CCR 的数值，当脉冲计数器 TIMx_CNT 与 TIMx_CCR 的比较结果发生变化时，输出脉冲将发生跳变。

本实验中就是通过不断改变比较寄存器 TIMx_CCR 的值，赋予它指数曲线数据，达到控制 PWM 信号的占空比呈指数曲线变化的目的。在本函数代码中，我们对该成员赋予初始为 0，而改变比较寄存器 TIMx_CCR 值的操作是在中断服务函数中修改的。

填充完输出模式初始化结构体后，调用输出模式初始化函数 TIM_OCxInit()对通道进行初始化。

- 定时器中断及其它配置

本函数剩下的代码用 TIM_OCxPreloadConfig() 配置了各通道的比较寄存器 TIM_CCR 预装载使能；使用 TIM_ARRPreloadConfig()把重载寄存器 TIMx_ARR 使能，调用了 TIM_ITConfig()配置定时器更新中断，每个定时周期结束后触发一次。该中断的优先级由函数 NVIC_Config_PWM()配置，见代码清单 1-8。

代码清单 1-8 定时器中断优先级配置

```
1.  /**
2.    * @brief  配置嵌套向量中断控制器 NVIC
3.    * @param  无
4.    * @retval 无
5.    */
6.  static void NVIC_Config_PWM(void)
7.  {
8.    NVIC_InitTypeDef NVIC_InitStructure;
9.
10.   /* Configure one bit for preemption priority */
11.   NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);
12.
13.   /* 配置 TIM3_IRQ 中断为中断源 */
14.   NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;
15.   NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
16.   NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
17.   NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
18.   NVIC_Init(&NVIC_InitStructure);
19. }
```

配置好中断，下面就要编写中断服务函数，见代码清单 1-9。

代码清单 1-9 呼吸灯的定时器中断服务函数（本函数存放在 stm32f10x_it.c 文件中）

```
1.  /* 呼吸灯中断服务函数 */
2.  void TIM3_IRQHandler(void)
3.  {
4.      static uint8_t pwm_index = 0;        //用于 PWM 查表
5.      static uint8_t period_cnt = 0;      //用于计算周期数
6.
7.      if (TIM_GetITStatus(TIM3, TIM_IT_Update) != RESET) //TIM_IT_Update
8.          {
9.              period_cnt++;
10.              //若输出的周期数大于 10，输出下一种脉冲宽的 PWM 波
11.             if(period_cnt >= 10) /
12.             {
13.                 //根据 PWM 表修改定时器的比较寄存器值
14.                 TIM3->CCR3 = indexWave[pwm_index];
15.
16.                 pwm_index++;       //标志 PWM 表的下一个元素
```

```
17.
18.                 //若 PWM 脉冲表已经输出完成一遍，重置 PWM 查表标志
19.                 if( pwm_index >=  40)
                      {
20.                     pwm_index=0;
21.                 }
22.
23.                 period_cnt=0;    //重置周期计数标志
24.
25.         }
26.       TIM_ClearITPendingBit (TIM3, TIM_IT_Update);//必须要清除中断标志位
27.
28.     }
29.
30.}
```

本中断服务函数在每次定时器更新事件发生时执行一次(即 256 个定时器时钟周期)。函数中使用了静态变量 pwm_index 和 period_cnt，它们分别用来查找 PWM 表元素和记录同样占空比的脉冲输出了多少次。

本代码的目的是每 10 次定时器中断更新一次 PWM 表中的数据到比较寄存器 TIMx_CCR 中，当遍历完 PWM 表的 40 个元素时，再重头开始遍历 PWM 表，周而复始，重复 LED 的呼吸过程。

整个呼吸过程的时间计算方法如下：

因为定时器的 TIM_Prescaler 设置为 1999；

所以定时器的时钟频率：f_{TIM} = 72000000/(TIM_Prescaler+1) = 36000 Hz

即定时器的时钟周期为：$t_{TIM} = 1/f_{TIM} = 1/36000$ s

因为定时器的 TIM_Period 设置为 255；

所以定时器的中断周期为：$t_{int} = t_{TIM} * (TIM_Period+1) = 0.0071$ s

因为 PWM 表有 pwm_index = 40 个亮度占空比数据，同种占空比信号输出 period_cnt = 10 次

所以一个呼吸周期 T = t_{int} *40 *10 = 2.844 s

这个呼吸周期长度和前面提到的 3s 近似，下载测试，实验效果很好。

● **疑问**

有读者问，为什么要在中断服务函数中设置一个变量 period_cnt 让同种占空比数据输出 10 次呢？如果把定时器的 TIM_Prescaler 设置成 19999(原来的 10 倍)，然后每种占空比信号由原来的 10 次改为 1 次，那么总的呼吸周期还是一样的 2.844s ，为什么不采用这样看起来更简单的方案？ 野火曾测试过，后一种方案的实验现象是 LED 会出现闪烁，原因就让读者亲自去思考吧，野火不想码字了。^_^

1.3.4 实验现象

下载程序，连接好开发板上的 LED1 跳帽 J37，见图 1-4。即可以看到 LED1 呼吸灯的效果。

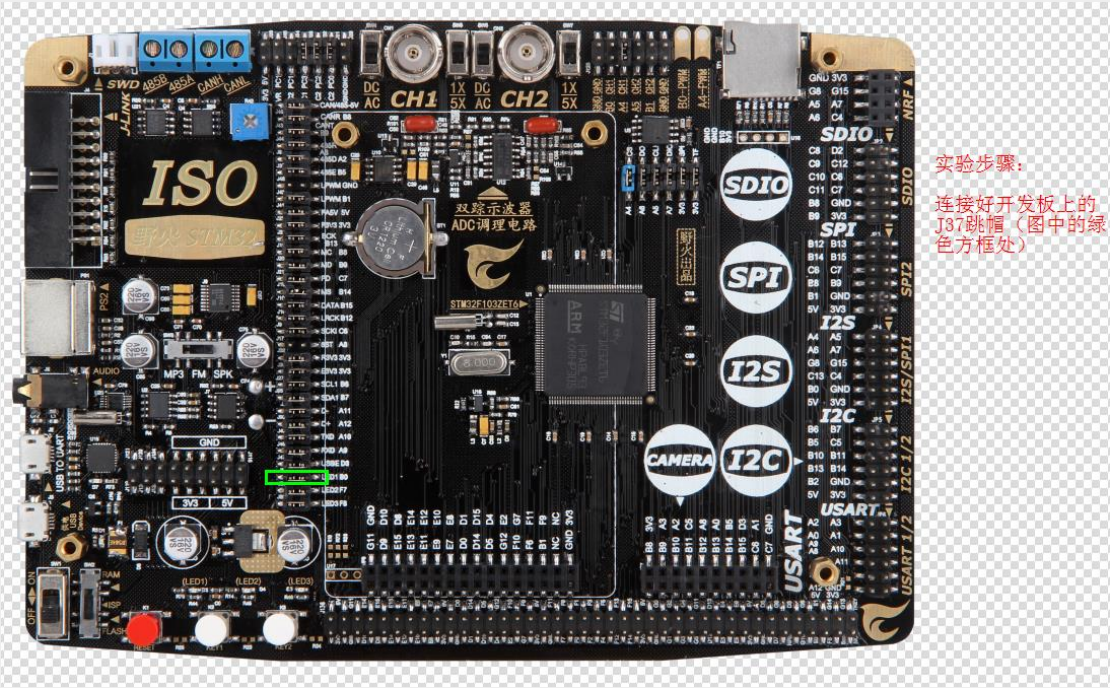


图 1-4 呼吸灯的跳帽连接