西华师范大学计算机学院

STM32 单片机原理及应用
实验总结与期末开卷考试宝典

课程名称：单片机原理及应用

适用教材：基于标准库/HAL库的STM32嵌入式系统开发

实验项目：实验一 至 实验九 完整大纲

特点：保姆级逐行注释代码 + 考点总结 + 经典预测题库

整理时间：2026年7月

【开卷备考必读说明】

本宝典是针对西华师范大学计算机学院《单片机原理及应用》课程期末考试（开卷考）精心整理的复习资料。根据老师关于“单选、判断、简答、程序分析、程序设计，吃透做过的实验代码每一行”的考前指导，本资料对全部 *9* 个实验进行了系统归纳。

每个实验由 硬件原理、库函数、核心源码逐行深入解释、以及量身定制的期末模拟试题（含答案和详细解析）组成。代码注释不仅说明“这行做什么”，更说明了“为什么要这样写，参数含义是什么”，保证平日未完全消化课堂知识的同学也能通过阅读本资料考出及格以上的优异成绩。

复习目录 / Study Guide Table of Contents

〔说明：本资料目录页码根据 *PDF* 最终排版真实映射，支持点击直接跳转对应实验，非常方便快速检索。〕

目录是空的，因为你没有使用被设为在目录中显示的段落样式。

★ Experiment 1: GPIO Control & Water Lamp

实验一：GPIO口控制与流水灯实验

■ 一、 实验目的与核心任务

掌握GPIO引脚的推挽输出模式（GPIO_Mode_Out_PP）配置。控制GPIOE端口的低8位（PE0~PE7）连接的8个LED灯实现单灯流动点亮，并深入理解SysTick内核定时器延时函数的工作原理。

■ 二、 硬件原理与引脚连接

8个LED灯分别连接在芯片的 GPIOE0, GPIOE1, GPIOE2, GPIOE3, GPIOE4, GPIOE5, GPIOE6, GPIOE7 引脚上。极性为：引脚输出低电平（0）时点亮，高电平（1）时熄灭。

■ 三、 软件设计底层原理

1. GPIO工作模式：推挽输出（Out_PP）适合驱动电流（可输出高/低电平）；开漏输出（Out_OD）需要外接上拉电阻。

2. 固件库配置流程：使能外设时钟（APB2总线）-> 配置引脚、工作模式、最大输出速度 -> 调用 GPIO_Init 完成初始化。

3. SysTick定时器：Cortex-M3内核级别的24位递减计数器。通过向系统计数器加载重装值（LOAD），设置控制寄存器（CTRL），当计数值递减到0时，标志位溢出并可选触发中断。

■ 四、 关键寄存器与常用库函数

- - RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState)：使能或失能APB2外设时钟。
- - GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)：初始化指定GPIO外设的引脚配置。
- - GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal)：向指定GPIO端口写入16位数值以控制全部引脚。

■ 五、 核心程序（含逐行保姆级注释）

```
// === led.c ===
#include "led.h"
#include "stm32f10x.h" // 引入STM32标准库头文件

void LEDInit(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure; // 定义GPIO初始化结构体

    // 1. 开启GPIOE端口的时钟（GPIOE挂载在APB2高速总线上）
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
```

```
    // 2. 配置结构体参数
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_All;          // 配置该端口的所有引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;      // 设置为推挽输出模式（适
合驱动LED）
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;     // 设置输出速度为10MHz

    // 3. 调用初始化函数，将配置写入GPIOE的控制寄存器
    GPIO_Init(GPIOE, &GPIO_InitStructure);
}

// === Delay.c ===
#include "stm32f10x.h"

// 硬件晶振为72MHz，HCLK为72MHz。SysTick使用HCLK作为时钟源时，1微秒需要72个时钟周期。
void Delay_us(uint32_t xus)
{
    SysTick->LOAD = 72 * xus;                // 1. 设置系统定时器自动重装载值（1us计数
72次）
    SysTick->VAL = 0x00;                     // 2. 清空当前计数值和溢出标志
    SysTick->CTRL = 0x00000005;              // 3. 设置时钟源为HCLK(bit2=1)，开启定
时器(bit0=1)
    while(!(SysTick->CTRL & 0x00010000));    // 4. 等待计数器递减到0（CTRL寄存器
bit16会由硬件置1）
    SysTick->CTRL = 0x00000004;              // 5. 关闭系统定时器
}

void Delay_ms(uint32_t xms)
{
    while(xms--)
    {
        Delay_us(1000); // 循环调用1000微秒延时实现毫秒延时
    }
}

// === main.c ===
#include "stm32f10x.h"
#include "led.h"
#include "delay.h"

int main(void)
{
    uint16_t led_data;
    LEDInit(); // 初始化GPIOE

    while(1)
    {
        led_data = 0x0001; // 初始为 0000 0000 0000 0001（PE0引脚高电平，其余为低）
        for(int i = 0; i < 8; i++)
        {
            // 注意：开发板LED采用低电平点亮。
            // 故要点亮第 i 个灯，应向GPIOE写入按位取反的值（第i位置0点亮，其余位置1熄
灭）。
            GPIO_Write(GPIOE, ~led_data);
            Delay_ms(150);     // 延时150毫秒
```

```
            led_data = led_data << 1; // 数据左移一位，移动到下一个LED
        }
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在STM32F103中，GPIO引脚配置为（ ）模式时，引脚既可以输出高低电平，又具有较强的双向驱动能力?

A. 模拟输入 (AIN)

B. 漏极开路输出 (Out_OD)

C. 推挽输出 (Out_PP)

D. 浮空输入 (IN_FLOATING)

【参考答案】 C

【考点解析】 推挽输出（*Out_PP*）使用两个互补的*MOS*管，能输出高电平或低电平，输出电流和灌电流能力强，适合直接驱动*LED*等外设。

【判断题 2】SysTick是内核外设，它的递减计数器（VAL）递减到0时，控制与状态寄存器（CTRL）的第16位 COUNTFLAG 会被硬件置1，且读取该寄存器后该标志位会自动清零。（ ）

【参考答案】 正确

【考点解析】 *SysTick* 的第*16*位 *COUNTFLAG* 确实是溢出标志。当计数到*0*时硬件置*1*，读取 *CTRL* 寄存器会清空该位，这是典型的*“*读清零*”*机制。

【简答题 3】在使用STM32外设（如GPIOE）前，为什么必须调用 `RCC_APB2PeriphClockCmd` 来开启其时钟?

【参考答案】 STM32为了实现低功耗设计，默认将所有外设的时钟全部关闭。如果外设时钟未开启，外设的寄存器将无法工作，读写该外设寄存器的操作也将无效。因此使用任何外设前必须先开启对应的总线时钟。

【考点解析】 这是嵌入式常考基础题，考察时钟树和低功耗机制。

【程序分析题 4】阅读 main.c，解释 `GPIO_Write(GPIOE, ~led_data);` 中为什么要对 `led_data` 进行按位取反（`~`）操作?

【参考答案】 因为硬件电路上LED的阴极连接在GPIOE引脚，阳极连接3.3V。这意味着当GPIOE引脚输出低电平（0）时，LED导通发光；输出高电平（1）时，两端无电压差，LED熄灭。`led_data` 中为1的位代表要点亮的灯，按位取反后该位变成0，从而在物理引脚输出低电平将其点亮，其余位变为1保持熄灭状态。

【考点解析】 考察软件逻辑与硬件电路极性（高电平点亮 *vs* 低电平点亮）的配合。

【程序设计题 5】编写初始化 GPIOB 端口第 5 号引脚（PB5）为推挽输出模式的函数 `void GPIOB_Init(void)`。

【参考答案】 ```c\nvoid GPIOB_Init(void)\n{\n   GPIO_InitTypeDef GPIO_InitStructure;\n   RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); // 使使能GPIOB时钟\n   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;   // 选择第5引脚\n   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;   // 推挽输出\n   GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;   // 50MHz速度\n   GPIO_Init(GPIOB, &GPIO_InitStructure);   // 初始化\n}\n```

【考点解析】 重点是写对开启时钟宏和结构体字段赋值。

★ Experiment 2: Key Scan & Software Debouncing
实验二：按键输入扫描与软件消抖实验

■ 一、 实验目的与核心任务

掌握GPIO的输入模式配置，实现按键状态的轮询扫描检测。学会利用软件延时方法消除机械按键的触点抖动，并处理独立按键控制多路LED及蜂鸣器状态变化的程序逻辑设计。

■ 二、 硬件原理与引脚连接

- 4个按键连接在 GPIOD0, GPIOD1, GPIOD2, GPIOD3。按键按下时引脚接地（低电平），松开时悬空，需配置为上拉输入（GPIO_Mode_IPU）。
- 8个LED灯接在 GPIOE 端口。
- 蜂鸣器接在 GPIOC 端口，高电平触发鸣叫。

■ 三、 软件设计底层原理

1. 输入模式选择：上拉输入（IPU）引脚默认通过内部电阻拉高到VDD。当机械按键按下时引脚强制接地变成低电平，通过检测低电平即可判定按键按下。

2. 按键抖动：由于机械弹片的弹性接触，在按下和释放的瞬间会产生 10-20ms 的电压震荡抖动。如果不进行消抖处理，程序会误判定发生了多次按键动作。

3. 软件消抖：在首次检测到低电平后，延时 10-20ms 避开抖动期，再次检测该引脚，若仍为低电平才确认按键按下。松手检测通过 `while(GPIO_ReadInputDataBit(...) == 0)` 循环等待引脚恢复为高电平。

■ 四、 关键寄存器与常用库函数

- - GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)：读取指定引脚的输入电平状态（返回0或1）。
- - GPIO_ReadInputData(GPIO_TypeDef* GPIOx)：读取整个端口输入寄存器的16位数值。

■ 五、 核心程序（含逐行保姆级注释）

```
// === key.c ===
#include "key.h"
#include "stm32f10x.h"
#include "delay.h"

// 初始化PD0~PD3为上拉输入
void KEYInit(void)
{
    GPIO_InitTypeDef  GPIO_InitStructure;
    // 1. 开启GPIOD时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD, ENABLE);

    // 2. 配置引脚参数
```

```
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2 |
GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;       // 配置为上拉输入模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;

    // 3. 执行初始化
    GPIO_Init(GPIOD, &GPIO_InitStructure);
}

// 轮询按键检测函数
uint8_t KEY_GETNUM(void)
{
    // 读取整个GPIOD输入状态，取反。若有按键（变为0），取反后对应低4位对应的位会变成1
    uint16_t KEYval = ~GPIO_ReadInputData(GPIOD);

    // 1. 初步判断是否有按键按下（若低4位不为0，说明有按键按下）
    if(KEYval & 0x000F)
    {
        Delay_ms(10); // 2. 软件消抖延时10ms，避开触点抖动期

        // 3. 再次判断具体是哪个引脚仍处于低电平（按键状态确认）
        if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_0) == 0)
        {
            while(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_0) == 0); // 松手检测：死
循环直到按键释放
            return 1; // 返回按键编号1
        }
        else if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_1) == 0)
        {
            while(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_1) == 0);
            return 2;
        }
        else if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_2) == 0)
        {
            while(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_2) == 0);
            return 3;
        }
        else if(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3) == 0)
        {
            while(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_3) == 0);
            return 4;
        }
    }
    return 0; // 无按键按下返回0
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】机械按键在按下和松开的瞬间会产生机械抖动，抖动时间通常在（ ）左右?

A. 1 ~ 2 ms
B. 10 ~ 20 ms
C. 100 ~ 200 ms
D. 1 ~ 2 秒

【参考答案】 B

【考点解析】 考察机械按键的物理特性。抖动时间在*10-20ms*，因此软件消抖通常采用*10ms*或*20ms*的延时。

【判断题 2】按键松手检测语句 `while(GPIO_ReadInputDataBit(GPIOD, GPIO_Pin_0) == 0);` 会导致程序在按键未释放时一直阻塞在这一行代码中，从而降低CPU执行效率。（ ）

【参考答案】 正确

【考点解析】 死循环等待松手虽然简单，但在等待期间*CPU*什么都不能做，是典型的阻塞式设计。在复杂系统中通常用状态机或中断代替。

【简答题 3】如果不使用软件消抖和硬件消抖，按键控制LED状态翻转的程序在实际运行中会出现什么问题？为什么？

【参考答案】 会出现LED状态翻转错乱、按键按下后LED灯状态随机变化（有时亮有时灭）的问题。因为按键按下时由于机械抖动产生了一连串的高低电平脉冲，单片机主循环运行极快，会在几毫秒内检测到多次“按下”事件，从而使LED状态翻转多次，导致最终状态不可控。

【考点解析】 考查消抖的必要性。

【程序分析题 4】阅读 `KEY_GETNUM` 函数，分析 `KEYval = ~GPIO_ReadInputData(GPIOD);` 这一行代码的作用。

【参考答案】 这行代码用于读取整个GPIOD端口所有16个引脚的输入电平。由于按键接在PD0~PD3，按键按下时对应的引脚会输入低电平（0），其余未按下的引脚及未使用的引脚由于内部上拉会保持高电平（1）。通过按位取反（`~`）操作后，原本按下的低电平（0）位变成了1，未按下的高电平位变成了0。这使得检测按键状态变为了简单的位与操作（如 `& 0x000F`），方便后续判断。

【考点解析】 考查按位取反在按键检测中的巧妙应用。

【程序设计题 5】编写一段在 `main` 循环中检测按键1（PD0），当检测到按下并释放后，将PE0引脚上的LED灯状态翻转的代码。

【参考答案】 ```c\nif(KEY_GETNUM() == 1)\n{\n   if(GPIO_ReadOutputDataBit(GPIOE, GPIO_Pin_0) == 0)\n GPIO_SetBits(GPIOE, GPIO_Pin_0);\n   else\n   GPIO_ResetBits(GPIOE, GPIO_Pin_0);\n}\n```

【考点解析】 考察如何用库函数对引脚状态实施翻转。

★ Experiment 3: External Interrupts (EXTI)

实验三：外部中断与中断嵌套实验

■ 一、 实验目的与核心任务

理解STM32的中断系统及NVIC中断控制器的工作机制。掌握外部中断线路（EXTI）的映射与触发方式配置，学会编写中断服务程序，实现中断嵌套（即高优先级中断打断低优先级中断）的功能。

■ 二、 硬件原理与引脚连接

- 3个按键PD0, PD1, PD2分别配置为外部中断输入源，对应中断通道 EXTI0, EXTI1, EXTI2。
- 8个LED灯接在 GPIOE 上作为状态显示。

■ 三、 软件设计底层原理

1. EXTI线路映射：STM32的GPIO端口引脚16个，分为16组外部中断线（EXTI0~EXTI15）。通过AFIO外设配置，PA0, PB0, PC0, PD0等任何端口的0号引脚都可以映射到EXTI0。但**同一时刻每条中断线只能映射一个引脚**。

2. NVIC中断控制器：管理芯片的中断响应和优先级。优先级分为抢占优先级和响应（子）优先级。高抢占优先级的外部中断可以打断低抢占优先级的外部中断，从而发生**中断嵌套**。若抢占优先级相同，则无法嵌套，通过子优先级或硬件中断向量表顺序排队。

3. 中断服务程序规则：中断服务程序由硬件自动调用，必须在特定名称的ISR函数中编写（如`EXTI0_IRQHandler`）。在函数内，必须先检测中断挂起位，处理完毕后**必须手动调用清除标志位函数**，否则CPU会反复进入中断死机。

■ 四、 关键寄存器与常用库函数

- - GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource)：配置引脚源与EXTI中断线映射。
- - EXTI_Init(EXTI_InitTypeDef* EXTI_InitStruct)：初始化EXTI外设参数。
- - NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct)：将中断通道使能及优先级分组写入NVIC寄存器。
- - EXTI_ClearITPendingBit(uint32_t EXTI_Line)：清除中断挂起位标志。

■ 五、 核心程序（含逐行保姆级注释）

```
// === exti_key.c ===
#include "exti_key.h"
#include "stm32f10x.h"

void EXTI_KEY_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    EXTI_InitTypeDef EXTI_InitStructure;
```

```
    NVIC_InitTypeDef NVIC_InitStructure;

    // 1. 使能GPIOD端口和AFIO复用功能时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOD | RCC_APB2Periph_AFIO, ENABLE);

    // 2. 初始化PD0/PD1/PD2为上拉输入模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
    GPIO_Init(GPIOD, &GPIO_InitStructure);

    // 3. 将GPIOD的0,1,2号引脚映射到EXTI0, EXTI1, EXTI2中断线上
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource0);
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource1);
    GPIO_EXTILineConfig(GPIO_PortSourceGPIOD, GPIO_PinSource2);

    // 4. 配置EXTI线路参数
    EXTI_InitStructure.EXTI_Line = EXTI_Line0 | EXTI_Line1 | EXTI_Line2;
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;       // 模式为中断模式
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;   // 下降沿触发（按键
按下接地产生下降沿）
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;                 // 线路使能
    EXTI_Init(&EXTI_InitStructure);

    // 5. 配置NVIC优先级分组：选择组2（2位抢占优先级，2位响应优先级）
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);

    // 配置PD2(EXTI2)：最高优先级。抢占优先级0，子优先级0
    NVIC_InitStructure.NVIC_IRQChannel = EXTI2_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0; // 抢占最高
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    // 配置PD1(EXTI1)：中等优先级。抢占优先级1，子优先级0
    NVIC_InitStructure.NVIC_IRQChannel = EXTI1_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_Init(&NVIC_InitStructure);

    // 配置PD0(EXTI0)：最低优先级。抢占优先级2，子优先级0
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
    NVIC_Init(&NVIC_InitStructure);
}

// === stm32f10x_it.c (中断服务函数) ===
#include "stm32f10x_it.h"
#include "led.h"
#include "delay.h"

// EXTI1 中断服务函数（PD1按下触发，较低优先级）
void EXTI1_IRQHandler(void)
{
    // 检测是否是EXTI1触发的中断
    if (EXTI_GetITStatus(EXTI_Line1) != RESET)
```

```
    {
        // 循环点亮流水灯，模拟一个耗时较长的处理过程，以便观察是否会被抢占打断
        for(int i = 0; i < 5; i++)
        {
            GPIO_Write(GPIOE, 0x00F0); // PE4~PE7亮
            Delay_ms(400);
            GPIO_Write(GPIOE, 0x000F); // PE0~PE3亮
            Delay_ms(400);
        }
        // 清除中断挂起位
        EXTI_ClearITPendingBit(EXTI_Line1);
    }
}

// EXTI2 中断服务函数（PD2按下触发，高优先级抢占）
void EXTI2_IRQHandler(void)
{
    if (EXTI_GetITStatus(EXTI_Line2) != RESET)
    {
        // 发生抢占嵌套时，8个LED灯快速闪烁3次
        for(int i = 0; i < 3; i++)
        {
            GPIO_Write(GPIOE, 0x00FF); // 全亮
            Delay_ms(150);
            GPIO_Write(GPIOE, 0x0000); // 全灭
            Delay_ms(150);
        }
        EXTI_ClearITPendingBit(EXTI_Line2);
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】STM32F103 的端口引脚 PA1 和 PB1 分别连接两个独立按键，（ ）配置为外部中断输入以分别触发不同的中断函数?

A. 可以同时

B. 不能同时，因为共用EXTI1中断线

C. 必须启用AFIO重映射才可以同时

D. 取决于引脚输入模式的配置

【参考答案】 B

【考点解析】 *EXTI1* 物理线只有一路，同一时刻在 *PA1*、*PB1*、*PC1*、*PD1* 等引脚中只能选择其中一个引脚映射到 *EXTI1* 线。如果强行都配置，后配置的会覆盖先配置的。

【判断题 2】在NVIC中断控制器中，如果两个中断源的抢占优先级不同，抢占优先级高的中断可以打断抢占优先级低的中断服务程序，发生中断嵌套。（ ）

【参考答案】 正确

【考点解析】 中断嵌套必须在“抢占优先级”不同的前提下发生。响应（子）优先级的高低只能决定当多个中断同时申请时谁先响应，不能打断正在执行的中断。

【简答题 3】为什么外部中断服务程序的末尾必须调用 `EXTI_ClearITPendingBit` 函数？如果不调用，会发生什么？

【参考答案】 因为当外部中断发生时，挂起寄存器（PR）中对应的标志位会被硬件置1，以向CPU申请中断。执行完中断服务程序后，如果不用软件代码将该标志位清零，当程序退出中断服务函数的瞬间，NVIC会检测到挂起位仍为1，从而误认为又产生了一次新的中断。这会导致CPU反复无限次进入该中断，主程序完全卡死。

【考点解析】 高频考点，考查*PR*寄存器标志位清除机制。

【程序分析题 4】NVIC配置为组2（`NVIC_PriorityGroup_2`）。如果中断A的抢占优先级为1，子优先级为2；中断B的抢占优先级为2，子优先级为0。若B正在执行，此时A触发，会发生什么？若A正在执行，B触发，又会发生什么？

【参考答案】 组2中抢占优先级分为0~3（数字越小优先级越高）。A的抢占优先级（1）高于B的抢占优先级（2）。

1. 当B正在执行时，A触发，由于A的抢占优先级高，A可以抢占打断B，系统发生中断嵌套，CPU转去执行A，执行完A后返回继续执行B。
2. 当A正在执行时，B触发，由于B的抢占优先级低，无法抢占A，B必须排队等待，直到A执行完毕退出后，B才能得到响应执行。

【考点解析】 考查*NVIC*优先级计算与抢占规则。

【程序设计题 5】请补充 EXTI 中断服务函数 `void EXTI0_IRQHandler(void)` 的框架，要求包含检查中断标志位和清除中断挂起位的标准代码。

【参考答案】 ```c\nvoid EXTI0_IRQHandler(void)\n{\n   if (EXTI_GetITStatus(EXTI_Line0) != RESET)\n   {\n   EXTI_ClearITPendingBit(EXTI_Line0);\n   }\n}\n```

【考点解析】 重点考察中断服务程序的标准逻辑模板。

★ Experiment 4: Timer Interrupts (TIM)

实验四：定时器中断配置实验

■ 一、 实验目的与核心任务

掌握通用定时器（TIM2）的初始化与计数周期配置。利用定时器产生精准的定时中断，在中断服务程序中控制LED状态翻转，实现无软件延时阻塞的并行程序运行。

■ 二、 硬件原理与引脚连接

- LED灯1和2接在 GPIOE0 和 GPIOE1 上。
- 按键1和2接在 GPIOD 端口。

■ 三、 软件设计底层原理

1. 定时器时钟源：通用定时器挂载在 APB1 低速总线上。由于系统设计，其输入时钟自动倍频为 APB1 时钟的2倍，即 72MHz。

2. 定时器计数器工作模式：向上计数模式下，计数器（CNT）从0开始计数到自动重装载值（ARR），产生溢出事件（UEV）并重置为0并触发更新中断。

3. 定时周期计算公式：

\[T_{out} = \frac{(ARR + 1) \times (PSC + 1)}{f_{clk}} \]

其中，\(ARR\) 为自动重装载值，\(PSC\) 为预分频系数，\(f_{clk}\) 为定时器输入时钟频率（一般为72MHz）。

■ 四、 关键寄存器与常用库函数

- - TIM_TimeBaseInit(TIM_TypeDef* TIMx, TIM_TimeBaseInitTypeDef* TIM_TimeBaseInitStruct)：配置预分频器、自动重装载值、计数模式和时钟分割。
- - TIM_ITConfig(TIM_TypeDef* TIMx, uint16_t TIM_IT, FunctionalState NewState)：使能或失能指定的定时器中断（如更新中断 TIM_IT_Update）。
- - TIM_Cmd(TIM_TypeDef* TIMx, FunctionalState NewState)：启动或关闭定时器计数器。

■ 五、 核心程序（含逐行保姆级注释）

```
// === timer.c ===
#include "timer.h"
#include "stm32f10x.h"

// 初始化TIM2定时器：配置为每隔 2 秒产生一次更新中断
void TIMERInit(void)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
```

```
    // 1. 开启TIM2时钟（TIM2挂载在APB1总线上）
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);

    // 2. 配置定时器基本参数
    // 计算计数频率：72MHz / 7200 = 10,000Hz（即每0.1ms计数一次）
    TIM_TimeBaseInitStructure.TIM_Prescaler = 7199;                // 预分频器值（实际PSC = 7199 + 1 = 7200）
    TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up; // 向上计数模式
    // 计算定时周期：10,000Hz 下计数 20,000 次产生一次溢出（即 2 秒）
    TIM_TimeBaseInitStructure.TIM_Period = 19999;                  // 自动重装载值（实际ARR = 19999 + 1 = 20000）
    TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;    // 时钟分割，不分频
    TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStructure);

    // 3. 开启定时器的“更新中断”（Update Interrupt）
    TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);

    // 4. 定时器默认处于关闭状态，调用 TIM_Cmd 可以启动/关闭它
    TIM_Cmd(TIM2, ENABLE);

    // 5. 配置 NVIC 中断通道控制器
    NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;                // 中断通道为 TIM2 中断
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;      // 抢占优先级 1
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;             // 子优先级 3
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                // 中断通道使能
    NVIC_Init(&NVIC_InitStructure);
}

// === stm32f10x_it.c（中断服务程序） ===
#include "stm32f10x_it.h"
#include "stm32f10x.h"

// TIM2 中断服务程序
void TIM2_IRQHandler(void)
{
    // 检查是否发生定时器更新溢出中断
    if (TIM_GetITStatus(TIM2, TIM_IT_Update) != RESET)
    {
        // 翻转PE0端口上的LED1灯状态
        if(GPIO_ReadOutputDataBit(GPIOE, GPIO_Pin_0) == 0)
            GPIO_SetBits(GPIOE, GPIO_Pin_0);
        else
            GPIO_ResetBits(GPIOE, GPIO_Pin_0);

        // 必须清除更新中断挂起标志位，否则会反复触发中断
        TIM_ClearITPendingBit(TIM2, TIM_IT_Update);
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】当STM32系统时钟为72MHz时，通用定时器预分频系数配置为 PSC = 35999，自动重装载值配置为 ARR = 1999，则该定时器产生中断的频率是多少？（）

A. 1 Hz
B. 10 Hz
C. 0.5 Hz
D. 2 Hz

【参考答案】 A

【考点解析】 时钟频率 *72MHz*。计数频率为 *72,000,000 / (35999+1) = 2,000Hz*。溢出计数次数为 *ARR + 1 = 2000*次。溢出周期为 *2000 / 2000 = 1*秒，因此产生中断的频率是 *1Hz*。

【判断题 2】STM32的定时器中断服务程序中也可以调用 `Delay_ms` 进行长延时操作，这不会影响其他外设和主程序的运行。（）

【参考答案】 错误

【考点解析】 中断服务程序（*ISR*）应当“快速进、快速出”。如果在中断里使用死等待的长延时，会导致*CPU*被该中断完全独占，其他同级或低级中断无法响应，主程序也会被卡死，这在嵌入式设计中是大忌。

【简答题 3】简述通用定时器（TIMx）在向下计数模式（TIM_CounterMode_Down）下的工作原理是什么？

【参考答案】 向下计数模式下，计数器（CNT）从自动重装载值（ARR）开始递减计数，每次时钟脉冲到来，CNT减1，一直减到0。当CNT减到0时，计数器会产生溢出事件（发生向下溢出），同时将CNT重新加载为ARR的值，并重新开始递减计数。

【考点解析】 考查定时器的计数工作模式原理。

【程序分析题 4】在定时器中断程序中，如果不加 `TIM_ClearITPendingBit(TIM2, TIM_IT_Update);` 这一句，硬件上LED灯会呈现什么状态？为什么？

【参考答案】 LED灯会卡在初始状态或者保持常亮/常灭，无法实现闪烁。因为不清除更新中断标志位，CPU在退出中断处理程序的瞬间就会被NVIC再次认为发生了更新中断，重新强制进入中断服务程序。这会导致CPU完全陷入中断死循环中，无法返回主程序，LED状态翻转无法随时间交替进行。

【考点解析】 考查定时器标志位清除机制。

【程序设计题 5】编写配置定时器 TIM2 溢出周期为 10ms 的初始化结构体赋值代码（预分频系数 PSC 和 自动重装载值 ARR，系统时钟为 72MHz）。

【参考答案】 ```c\nTIM_TimeBaseStructure.TIM_Prescaler = 719;\nTIM_TimeBaseStructure.TIM_Period = 999;\n```

【考点解析】 考察时间公式参数换算能力。

★ Experiment 5: PWM Output & Breathing LED
实验五：PWM波形输出与呼吸灯实验

■ 一、 实验目的与核心任务

掌握通用定时器（TIM3）的比较输出通道（OC）配置，了解脉冲宽度调制（PWM）技术的工作原理。实现通过改变PWM输出的占空比控制LED灯的平均亮度，呈现出由暗变亮再由亮变暗的“呼吸”效果。

■ 二、 硬件原理与引脚连接

呼吸灯接在 GPIOA7 引脚，该引脚为 TIM3 的通道2（TIM3_CH2）复用引脚。工作模式必须配置为复用推挽输出（GPIO_Mode_AF_PP）。

■ 三、 软件设计底层原理

1. PWM 占空比亮度控制：通过快速开启和关闭引脚电平，只要切换频率足够高（通常 >50Hz），由于人眼的视觉暂留效应，人眼会感知到高电平所占时间比例（占空比）带来的亮度变化。

2. PWM 模式 1 与模式 2：

- **PWM 模式 1**：向上计数时，只要计数器值 CNT < 比较值 CCR，输出即为有效电平（通常配置高电平有效），否则输出为无效电平。

3. PWM 周期与频率：由 ARR 和 PSC 决定。占空比由 CCR 的值决定。占空比公式：

\[\text{Duty} = \frac{CCR}{ARR + 1} \times 100\% \]

■ 四、 关键寄存器与常用库函数

- - TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct)：初始化定时器的比较输出通道2（CH2）参数。
- - TIM_SetCompare2(TIM_TypeDef* TIMx, uint16_t Compare2)：修改比较寄存器 CCR2 的数值，用于动态调整占空比。

■ 五、 核心程序（含逐行保姆级注释）

```
// === pwm.c ===
#include "pwm.h"
#include "stm32f10x.h"

void TIM3_PWM_Init(void)
{
    GPIO_InitTypeDef        GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
    TIM_OCInitTypeDef        TIM_OCInitStructure;

    // 1. 开启 TIM3 定时器和 GPIOA 端口的时钟
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);
```

```
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // 2. 配置 PA7 引脚（TIM3_CH2）为“复用推挽输出”模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;        // 复用推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 3. 配置定时器计数基础配置（频率 = 72MHz / 72 = 1MHz，即1us计数一次）
    TIM_TimeBaseStructure.TIM_Prescaler = 71;                  // 预分频器（PSC = 71+1 = 72）
    TIM_TimeBaseStructure.TIM_Period = 999;                    // 自动重装载值（ARR = 999+1 = 1000，即周期为 1ms）
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

    // 4. 配置比较输出通道2 (OC2) 参数
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;          // 使用 PWM 模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; // 开启比较输出使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 极性为高电平有效
    TIM_OC2Init(TIM3, &TIM_OCInitStructure);

    // 5. 开启通道缓存和重装载使能，启动定时器 TIM3
    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
    TIM_ARRPreloadConfig(TIM3, ENABLE);
    TIM_Cmd(TIM3, ENABLE);
}

// === main.c ===
#include "stm32f10x.h"
#include "pwm.h"
#include "delay.h"

int main(void)
{
    u16 pwm_val = 0; // 比较值 CCR2 变量
    u8 dir = 1;      // 亮度递增/递减方向标志：1为增加，0为减少

    TIM3_PWM_Init(); // 初始化 TIM3 通道2 输出 PWM

    while(1)
    {
        Delay_ms(1); // 延时1毫秒，使呼吸变化速度适中

        if(dir) pwm_val++; // 递增占空比，灯渐亮
        else pwm_val--;    // 递减占空比，灯渐暗

        // 边界检测：当 CCR2 达到 ARR (999) 时开始递减，到0时开始递增
        if(pwm_val >= 999) dir = 0;
        if(pwm_val == 0)   dir = 1;

        // 动态修改 TIM3_CCR2 的值，改变占空比
        TIM_SetCompare2(TIM3, pwm_val);
```

```
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在PWM输出实验中，若定时器自动重装载值配置为 ARR = 999，当修改比较寄存器 CCR 值为 300 时，该通道输出的占空比是（ ）？

A. 30%
B. 33.3%
C. 20%
D. 70%

【参考答案】 A

【考点解析】 占空比公式为 *CCR / (ARR + 1) = 300 / 1000 = 30%*。

【判断题 2】定时器引脚如果不配置为“复用功能”，而是直接配置为普通推挽输出（Out_PP），引脚也能输出定时器产生的 PWM 波形。（ ）

【参考答案】 错误

【考点解析】 普通推挽输出模式下，引脚的电平状态只能由输出数据寄存器（*ODR*）控制。若想将引脚控制权交给定时器（片上外设），必须将其配置为复用推挽输出（*AF_PP*）。

【简答题 3】请简述“PWM 模式 1”在向上计数模式（CounterMode_Up）下的电平输出规则是什么？

【参考答案】 在向上计数时，计数器 CNT 从0开始递增。只要当前计数器的值小于比较捕获寄存器的值（CNT < CCR），通道输出有效电平（若配置极性为High，则输出高电平）；当计数器的值大于或等于比较捕获寄存器的值（CNT >= CCR）时，通道输出无效电平（低电平）。

【考点解析】 经典考点，考查定时器核心控制逻辑。

【程序分析题 4】在 `main` 循环中，为什么调用 `TIM_SetCompare2` 就可以实时改变 LED 灯的明暗状态？其背后的硬件原理是什么？

【参考答案】 调用 `TIM_SetCompare2(TIM3, pwm_val);` 在底层实际上是向定时器通道2的比较捕获寄存器（TIM3_CCR2）写入数值 `pwm_val`。随着 `pwm_val` 的递增和递减，引脚输出的 PWM 波形高电平所占的时间段（占空比）随之连续增加和减少，流过LED灯的平均电流发生连续变化，从而在人眼视觉暂留作用下呈现出平滑过渡的呼吸效果。

【考点解析】 考查库函数、硬件寄存器和物理现象的映射关系。

【程序设计题 5】编写将 TIM3 比较寄存器 CCR2 的值直接设定为 500 的标准库代码语句。

【参考答案】 `TIM_SetCompare2(TIM3, 500);` 或直接操作寄存器 `TIM3->CCR2 = 500;`

【考点解析】 考查最核心的*PWM*占空比写入函数。

★ Experiment 6: Serial Communication (USART)

实验六：串口通信与 printf 重定向实验

■ 一、 实验目的与核心任务

掌握通用同步异步收发器（USART）的工作原理及串口通信协议格式。实现单片机与电脑串口调试助手的双向数据交换，学会配置串口接收中断以及重定向标准库 `printf` 函数到单片机串口输出。

■ 二、 硬件原理与引脚连接

- 串口发送 TX 接在 GPIOA9（USART1_TX）。
- 串口接收 RX 接在 GPIOA10（USART1_RX）。
- 开发板与电脑的USB转串口模块（如CP2102）的 GND 必须短接共地。

■ 三、 软件设计底层原理

1. 串行异步通信协议：不需要同步时钟线，通信双方需约定相同的**波特率**。数据帧格式包括：1位起始位（低电平）、数据位（通常8位）、校验位（奇、偶或无校验）和停止位（高电平，通常1位）。本次配置为：9600波特率，8位数据，无校验，1位停止位（9600, 8N1）。

2. 接收中断（RXNE）：当引脚 RX 接收满 1 字节数据时，硬件将接收非空标志位（RXNE）置1，并产生中断请求。CPU进入 `USART1_IRQHandler` 中断函数，读取数据寄存器（DR），读取动作会自动将 RXNE 标志位清零。

3. printf 重定向：标准库中的 `printf` 最终通过调用 `fputc` 输出字符。我们在串口驱动中重新编写 `fputc` 函数，将要发送的字符通过 `USART_SendData` 发出。同时，必须在编译器中禁用半主机模式（Semihosting）。

■ 四、 关键寄存器与常用库函数

- - USART_Init(USART_TypeDef* USARTx, USART_InitTypeDef* USART_InitStruct)：配置波特率、字长、停止位、奇偶校验、硬件流控制及收发模式。
- - USART_ITConfig(USART_TypeDef* USARTx, uint16_t USART_IT, FunctionalState NewState)：使能中断（如接收中断 USART_IT_RXNE）。
- - USART_SendData(USART_TypeDef* USARTx, uint16_t Data)：将1字节数据写入发送数据寄存器（DR）。
- - USART_ReceiveData(USART_TypeDef* USARTx)：从接收数据寄存器（DR）读取1字节数据。

■ 五、 核心程序（含逐行保姆级注释）

```
// === usart.c ===
#include "usart.h"
```

```
// 定义接收缓冲区，用于存储串口收到的数据
uint8_t usart1_rx_buf[USART1_RX_BUF_SIZE] = {0};
uint8_t usart1_rx_cnt = 0; // 缓冲区当前计数指针

void USART1_Init(void)
{
    GPIO_InitTypeDef        GPIO_InitStruct;
    USART_InitTypeDef       USART_InitStruct;
    NVIC_InitTypeDef        NVIC_InitStruct;

    // 1. 开启 USART1 外设和 GPIOA 端口的时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA,
ENABLE);

    // 2. 配置 PA9 (TX) 引脚为“复用推挽输出（AF_PP）”模式
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 3. 配置 PA10 (RX) 引脚为“浮空输入”模式
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN_FLOATING; // 接收数据不需要输出能力，浮
空输入即可
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 4. 配置 USART1 通信参数 (9600, 8N1)
    USART_InitStruct.USART_BaudRate =
9600;                                   // 波特率 9600
    USART_InitStruct.USART_WordLength =
USART_WordLength_8b;                   // 8位数据位
    USART_InitStruct.USART_StopBits =
USART_StopBits_1;                       // 1位停止位
    USART_InitStruct.USART_Parity =
USART_Parity_No;                        // 无奇偶校验
    USART_InitStruct.USART_HardwareFlowControl =
USART_HardwareFlowControl_None; // 无硬件流控制
    USART_InitStruct.USART_Mode = USART_Mode_Tx |
USART_Mode_Rx;              // 同时开启发送和接收模式
    USART_Init(USART1, &USART_InitStruct);

    // 5. 开启串口接收中断（RXNE - Receive Data Register Not Empty）
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);

    // 6. 配置串口中断在 NVIC 中的通道及优先级
    NVIC_InitStruct.NVIC_IRQChannel = USART1_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPreemptionPriority =
1;                     // 抢占优先级 1
    NVIC_InitStruct.NVIC_IRQChannelSubPriority =
1;                             // 子优先级 1
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);

    // 7. 使能（开启）USART1 串口外设
    USART_Cmd(USART1, ENABLE);
```

```
}

// 发送单字节函数
void USART1_SendByte(uint8_t byte)
{
    // 等待发送缓冲区变空（TXE - Transmit Data Register Empty 标志位为 SET）
    while (USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET);
    USART_SendData(USART1, byte); // 写入数据寄存器发送
}

void USART1_SendString(uint8_t *str)
{
    while (*str != '\0')
    {
        USART1_SendByte(*str++); // 循环发送直到字符串结束符
    }
}

// === 重定向 printf 函数 ===
#pragma import(__use_no_semihosting)
struct __FILE { int handle; };
FILE __stdout;
void _sys_exit(int x) { x = x; }

// 覆写标准库 fputc，将输出重定向到串口USART1
int fputc(int ch, FILE *f)
{
    USART1_SendByte((uint8_t)ch);
    return ch;
}

// === USART1 中断服务程序 ===
void USART1_IRQHandler(void)
{
    // 检查是否是串口接收寄存器非空（RXNE）触发的中断
    if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        uint8_t data = USART_ReceiveData(USART1);

        if (usart1_rx_cnt < USART1_RX_BUF_SIZE)
        {
            usart1_rx_buf[usart1_rx_cnt++] = data;
        }

        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在USART通信中，当单片机要向电脑发送数据时，需要查询发送状态标志位（ ）为高，表示发送数据寄存器变空，可以写入下一个字节?

A. RXNE
B. TC
C. TXE
D. ORE

【参考答案】 C

【考点解析】 *TXE (Transmit Data Register Empty)* 表示发送数据寄存器为空。当其为*SET*（*1*）时，我们可以安全写入新数据。

【判断题 2】在串口通信中，为了使 `printf` 函数能向串口输出字符，只要重写 `fputc` 函数即可，不需要在编译配置中处理半主机模式（Semihosting）。（ ）

【参考答案】 错误

【考点解析】 如果不关闭半主机模式，编译时系统会引入调试代理代码，导致程序在脱离*J-Link*下载器单独运行时卡死在 *printf* 内部。因此必须使用 *`#pragma import(__use_no_semihosting)`* 明确禁用半主机。

【简答题 3】什么是串行异步通信中的“波特率”？如果单片机配置波特率为9600，而电脑串口调试助手配置为115200，会出现什么现象？为什么？

【参考答案】 波特率（Baud Rate）是指单位时间内传送的二进制有效位数，代表了传输数据的速度。如果通信双方配置的波特率不匹配，会导致通信完全失败，电脑助手上会接收到一堆杂乱无章的“乱码”。因为异步通信没有时钟线，双方完全依靠约定的波特率频率进行定时脉冲的采样采样电平，频率不一致会导致采样位置严重错乱。

【考点解析】 考查波特率的本质和异步通信原理。

【程序分析题 4】在 `USART1_IRQHandler` 中，读取数据的语句是 `uint8_t data = USART_ReceiveData(USART1);`。请分析这行读取操作如何影响了串口的中断标志位?

【参考答案】 当串口接收满一个字节时，接收非空标志位（RXNE）由硬件置1。当执行读取数据寄存器（DR）的函数 `USART_ReceiveData` 后，硬件检测到数据已被读走，会自动将标志位 RXNE 清零（即复位为0）。这个硬件自动清除机制防止了反复触发接收中断，因此软件上不需要强制调用清除挂起位函数。

【考点解析】 考察串口数据寄存器*DR*的读清零机制。

【程序设计题 5】编写串口1中断服务函数，要求：当接收到一个字符后，立即将该字符加 1 并通过串口发送回去（即实现简单的字符移位回显）。

【参考答案】 ```c\nvoid USART1_IRQHandler(void)\n{\n   if (USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)\n   {\n   uint8_t data = USART_ReceiveData(USART1);\n   USART1_SendByte(data + 1);\n   USART_ClearITPendingBit(USART1, USART_IT_RXNE);\n   }\n}\n```

【考点解析】 考察如何在中断中实现基本的交互逻辑。

★ Experiment 7: Direct Memory Access (DMA)

实验七：DMA 直接内存访问传输实验

■ 一、 实验目的与核心任务

掌握直接内存访问（DMA）技术的工作原理。实现通过 DMA 将内存中的大容量缓冲区数据直接搬运至串口（USART1）发送数据寄存器，释放 CPU 运算资源，使其无需参与搬运即可完成高速、批量的串行数据发送。

■ 二、 硬件原理与引脚连接

- 串口发送引脚 GPIOA9（USART1_TX）。
- 触发按键接在 GPIOD0。

■ 三、 软件设计底层原理

1. DMA 数据搬运原理：DMA 是一种硬件控制的数据交换机制。它可以直接控制系统总线，在没有 CPU 干预的情况下，实现内存与外设、内存与内存之间的极速数据复制。传输时，CPU 可以去执行其他代码，大大提升了系统效率。

2. DMA传输三要素：

- **源地址**（Source Address）：数据搬运的起点。

- **目的地址**（Destination Address）：数据搬运的终点。

- **传输数量**（BufferSize）：本次要传输的数据个数。

3. 增量模式配置：为了搬运数组，内存地址必须配置为**地址递增使能（MemoryInc_Enable）**；由于数据最终全部写入单个串口数据寄存器（USART1_DR），外设地址必须配置为**地址不递增（PeripheralInc_Disable）**。

4. DMA1 通道映射：USART1_TX 发送请求绑定在 DMA1 的通道4（DMA1_Channel4）上，配置时不可混淆。

■ 四、 关键寄存器与常用库函数

- - DMA_Init(DMA_Channel_TypeDef* DMAy_Channelx, DMA_InitTypeDef* DMA_InitStruct): 初始化指定的DMA通道寄存器配置。
- - DMA_Cmd(DMA_Channel_TypeDef* DMAy_Channelx, FunctionalState NewState): 开启或关闭指定的DMA通道计数器。
- - DMA_SetCurrDataCounter(DMA_Channel_TypeDef* DMAy_Channelx, uint16_t DataNumber): 动态写入本次传输的字数/数据数量（必须在通道关闭状态下写入）。

• - USART_DMACmd(USART_TypeDef* USARTx, uint16_t USART_DMAReq, FunctionalState NewState): 配置串口向DMA发送请求使能状态。

■ 五、 核心程序（含逐行保姆级注释）

```
// === usart_dma.c ===
#include "usart_dma.h"
#include "stm32f10x.h"

// 全局发送缓冲区
uint8_t dma_send_buf[2048] = {0};

void USART1_DMA_Init(void)
{
    GPIO_InitTypeDef        GPIO_InitStructure;
    USART_InitTypeDef       USART_InitStructure;
    DMA_InitTypeDef         DMA_InitStructure;

    // 1. 开启时钟：GPIOA、USART1 (APB2总线) 以及 DMA1 (AHB高性能总线)
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_USART1,
ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);

    // 2. 配置 PA9 (TX) 为复用推挽输出模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 3. 配置 USART1 串口通信参数 (9600, 8N1)
    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No;
    USART_InitStructure.USART_HardwareFlowControl =
USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Tx; // 仅需发送模式
    USART_Init(USART1, &USART_InitStructure);

    // 4. 配置 DMA1 通道4（对应 USART1_TX 外设）
    DMA_DeInit(DMA1_Channel4); // 恢复初始配置
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&USART1->DR; // 外设地
址：串口数据寄存器DR地址
    DMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)dma_send_buf;     // 内存基
地址：发送缓冲区数组的首地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;                 // 传输方
向：内存为源，外设为目的地(DST)
    DMA_InitStructure.DMA_BufferSize = 0;                              // 初始传
输长度设为0，发送时再动态分配
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;   // 外设地
址不递增（保持DR寄存器地址不变）
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;            // 内存地
址递增（依次搬运数组中的各个字节）
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte; //
```

```
数据宽度：外设为1字节
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;          //
内存为1字节
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;                          // 工作模
式：正常单次模式（发完停止，不循环）
    DMA_InitStructure.DMA_Priority = DMA_Priority_Medium;                  // 通道优
先级：中等
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;                          // 禁用内存
到内存模式（当前是外设模式）
    DMA_Init(DMA1_Channel4, &DMA_InitStructure);

    // 5. 使使能串口向DMA发送请求的硬件连接线，并开启串口
    USART_DMACmd(USART1, USART_DMAReq_Tx, ENABLE);
    USART_Cmd(USART1, ENABLE);
}

// 启动一次 DMA 传输发送
void USART1_DMA_Send(uint16_t len)
{
    // 1. 动态设定传输长度前，必须先关闭DMA通道
    DMA_Cmd(DMA1_Channel4, DISABLE);

    // 2. 写入本次传输的数据长度
    DMA_SetCurrDataCounter(DMA1_Channel4, len);

    // 3. 重新开启通道，DMA控制器将立即开始自动搬运发送数据
    DMA_Cmd(DMA1_Channel4, ENABLE);

    // 4. 查询等待传输完成标志位（TC4 - Transfer Complete Channel 4）为高，代表全部发送
完毕
    while(DMA_GetFlagStatus(DMA1_FLAG_TC4) == RESET);

    // 5. 手动清除传输完成标志位，以备下一次发送
    DMA_ClearFlag(DMA1_FLAG_TC4);
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在配置 DMA 传输时，如果需要将内存中的数据批量发送到外设寄存器，传输方向（DMA_DIR）应该配置为（ ）？

A. DMA_DIR_PeripheralSRC
B. DMA_DIR_PeripheralDST
C. DMA_DIR_MemoryToMemory
D. DMA_DIR_MemoryToPeripheral

【参考答案】 B

【考点解析】 在固件库中，*DMA_DIR_PeripheralDST* 表示外设是数据的目的地，也就是从内存搬运到外设的发送模式。

【判断题 2】DMA 数据搬运是由 DMA 控制器完全接管系统总线完成的。在此期间，CPU 可以完全不被挂起，去处理其他中断或执行复杂的后台运算。（ ）

【参考答案】 正确

【考点解析】 *DMA* 极大地减轻了 *CPU* 的工作负担，只有在发生总线访问冲突时，*DMA* 会分时复用总线，*CPU* 极少被挂起。

【简答题 3】为什么在 DMA 配置中，外设地址增量模式必须配置为 `DMA_PeripheralInc_Disable`？

【参考答案】 因为所有的源数据必须批量写入同一个外设目的寄存器。如果配置为递增，当搬运完第1个字节后，目的地址会自动加1，变成未绑定的无效寄存器地址，导致数据写入错误，无法被串口外设接收并发送。

【考点解析】 考察 *DMA* 外设地址控制原理。

【程序分析题 4】在 `USART1_DMA_Send(uint16_t len)` 中，为什么在调用 `DMA_SetCurrDataCounter` 重设发送长度前，必须先调用 `DMA_Cmd(DMA1_Channel4, DISABLE)` 关闭通道？

【参考答案】 因为 STM32 硬件设计规定，DMA 传输计数寄存器（CNDTR）是在通道开启期间由硬件只读的。如果直接修改计数器的值，会导致数据传输计数错乱甚至引发总线硬件错误。因此修改前必须先关闭 DMA 通道以解锁该寄存器，修改完毕后再重新使能通道启动传输。

【考点解析】 高频考点，考查 *DMA* 动态修改计数器寄存器的顺序逻辑。

【程序设计题 5】编写配置 DMA1 通道4 的源地址为内存数组 `dma_send_buf`，目的地址为串口发送数据寄存器 `USART1->DR` 的代码语句。

【参考答案】 ```c\nDMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&USART1->DR;\nDMA_InitStructure.DMA_MemoryBaseAddr = (uint32_t)dma_send_buf;\n```

【考点解析】 考查 *DMA* 寻址指针类型转换与寄存器物理寻址。

★ Experiment 8: ADC & DMA Conversion

实验八：ADC模数转换与DMA多通道采集实验

■ 一、 实验目的与核心任务

理解STM32模数转换器（ADC）的工作原理与参数换算。掌握多通道ADC扫描模式的配置，利用 DMA（直接内存访问）将多通道转换的数据自动存入缓冲区，解决多通道共用单一数据寄存器导致的冲突覆盖问题。

■ 二、 硬件原理与引脚连接

- 3个ADC通道输入引脚分别为：PA0（通道0）、PA1（通道1）、PA2（通道2）。
- 采集模式为：PA0接3.3V参考电压，PA1接GND（0V），PA2接电位器的滑动端，用于输出可调模拟电压。

■ 三、 软件设计底层原理

1. 12位逐次逼近型ADC原理：将 0~3.3V 的模拟电压离散化为 12 位二进制数字量（数值范围 0~4095）。模数转换核心计算公式：

$$ V_{in} = \frac{ADC_{\text{val}} \times V_{ref}}{4096} $$

2. 多通道扫描模式与DMA机制：STM32 的多个 ADC 通道共用同一个 ADC 数据寄存器（ADC_DR）。当开启**扫描模式（ScanConvMode = ENABLE）**时，ADC 会按照配置顺序依次转换多个通道。如果不用 DMA，前一个通道的转换结果就会被后一个通道的结果直接覆盖。因此，配置 **DMA1 通道 1** 随 ADC 转换完成（EOC）自动将数据搬运到指定的内存数组中。

■ 四、 关键寄存器与常用库函数

- - RCC_ADCCLKConfig(uint32_t RCC_PCLK2): 配置ADC专用的时钟分频器，时钟频率不能超过 14MHz。
- - ADC_RegularChannelConfig(ADC_TypeDef* ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime): 在规则组队列中配置通道的转换顺序和采样时间。
- - ADC_DMACmd(ADC_TypeDef* ADCx, FunctionalState NewState): 使使能ADC向DMA发送请求的功能。
- - ADC_SoftwareStartConvCmd(ADC_TypeDef* ADCx, FunctionalState NewState): 启动软件触发转换器开始工作。

■ 五、 核心程序（含逐行保姆级注释）

```
// === adc.c ===
#include "adc.h"

// 存储3个通道转换结果的全局缓冲区
uint16_t adc_multi_buf[3] = {0};
```

```
void ADC_Multi_DMA_Init(void)
{
    GPIO_InitTypeDef        GPIO_InitStruct;
    ADC_InitTypeDef         ADC_InitStruct;
    DMA_InitTypeDef         DMA_InitStruct;

    // 1. 开启外设时钟：GPIOA、ADC1 (APB2高速总线) 以及 DMA1 (AHB总线)
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_ADC1, ENABLE);
    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);

    // 2. 配置ADC时钟分频：PCLK2的6分频，即 72MHz / 6 = 12MHz（不超过限制的 14MHz）
    RCC_ADCCLKConfig(RCC_PCLK2_Div6);

    // 3. 配置 PA0/PA1/PA2 引脚为“模拟输入（AIN）”模式，保证引脚的输入阻抗最大
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_2;
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AIN; // 模拟输入
    GPIO_Init(GPIOA, &GPIO_InitStruct);

    // 4. 配置 DMA1 通道1（对应 ADC1 传输请求）
    DMA_DeInit(DMA1_Channel1);
    DMA_InitStruct.DMA_PeripheralBaseAddr = (uint32_t)&ADC1->DR;      // 外设地
址：ADC数据寄存器地址
    DMA_InitStruct.DMA_MemoryBaseAddr = (uint32_t)adc_multi_buf;       // 内存地
址：接收数组首地址
    DMA_InitStruct.DMA_DIR = DMA_DIR_PeripheralSRC;                  // 方向：外设
为源(SRC)，内存为目的地
    DMA_InitStruct.DMA_BufferSize = 3;                               // 一次扫描传
输3个通道的数据
    DMA_InitStruct.DMA_PeripheralInc = DMA_PeripheralInc_Disable;   // 外设地址不
递增（保持读DR不变）
    DMA_InitStruct.DMA_MemoryInc = DMA_MemoryInc_Enable;             // 内存地址递
增（依次写入 buf[0], buf[1], buf[2]）
    DMA_InitStruct.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //
数据宽度：半字(16位)
    DMA_InitStruct.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;         //
内存为半字
    DMA_InitStruct.DMA_Mode = DMA_Mode_Circular;                     // 模式：循环
模式（采集完一轮后重新循环）
    DMA_InitStruct.DMA_Priority = DMA_Priority_Medium;
    DMA_InitStruct.DMA_M2M = DMA_M2M_Disable;
    DMA_Init(DMA1_Channel1, &DMA_InitStruct);
    DMA_Cmd(DMA1_Channel1, ENABLE); // 使使能 DMA 通道

    // 5. 配置 ADC1 工作参数
    ADC_InitStruct.ADC_Mode = ADC_Mode_Independent;                  // 独立工作模
式
    ADC_InitStruct.ADC_ScanConvMode = ENABLE;                        // 开启“多通
道扫描模式”
    ADC_InitStruct.ADC_ContinuousConvMode = ENABLE;                  // 开启“连续
转换模式”（不断工作）
    ADC_InitStruct.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;// 不使用外部
硬件触发（软件代码触发）
```

```
    ADC_InitStruct.ADC_DataAlign = ADC_DataAlign_Right;              // 数据右对齐
    ADC_InitStruct.ADC_NbrOfChannel = 3;                             // 参与扫描的
通道总数为 3 个
    ADC_Init(ADC1, &ADC_InitStruct);

    // 6. 配置规则组转换序列和通道采样周期时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_0, 1,
ADC_SampleTime_239Cycles5); // 序列1：通道0(PA0)
    ADC_RegularChannelConfig(ADC1, ADC_Channel_1, 2,
ADC_SampleTime_239Cycles5); // 序列2：通道1(PA1)
    ADC_RegularChannelConfig(ADC1, ADC_Channel_2, 3,
ADC_SampleTime_239Cycles5); // 序列3：通道2(PA2)

    // 7. 使使能 ADC 与 DMA 的连线，并使使能启动 ADC
    ADC_DMACmd(ADC1, ENABLE);
    ADC_Cmd(ADC1, ENABLE);

    // 8. 开启 ADC 自校准
    ADC_ResetCalibration(ADC1);
    while(ADC_GetResetCalibrationStatus(ADC1));
    ADC_StartCalibration(ADC1);
    while(ADC_GetCalibrationStatus(ADC1));

    // 9. 软件触发开启第一次转换（由于是连续模式，触发一次后将无限循环自动采集）
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在多通道 ADC 扫描采集模式下，为了防止多个通道的模数转换结果发生冲突和覆盖，应配合使用（）？

A. 中断嵌套机制

B. DMA 传输

C. 软件延时等待

D. 定时器捕获

【参考答案】 B

【考点解析】 多通道扫描转换由 *ADC* 自动顺序完成。由于只有一个 *ADC_DR* 规则数据寄存器，如果不配合使用 *DMA* 搬运，前一个通道的数据就会在几微秒内被后一个通道的数据直接覆盖，导致数据丢失。

【判断题 2】STM32F103 的 ADC 精度为 12 位，当参考电压为 3.3V 时，如果读取到的 ADC 值为 0，代表测量电压为 0V，若读取到的值为 4095，则代表测量电压为 3.3V。（）

【参考答案】 正确

【考点解析】 *12* 位 *ADC* 对应数字量范围是 *0~4095*（*2*的*12*次方减*1*）。

【简答题 3】简述 ADC 模数转换中“自校准（Reset & Start Calibration）”的目的和步骤。

【参考答案】 自校准的作用是减少芯片制造过程中各个 ADC 内部电容等模拟电路偏差带来的增益和零点偏差误差，极大提升转换的精度。步骤为：在开启 ADC 模块后，1. 调用 `ADC_ResetCalibration` 复位校准寄存器；2. 等待复位完成；3. 调用 `ADC_StartCalibration` 开启校准；4. 硬件自动校准，直至标志位清零代表校准结束。

【考点解析】 考查 *ADC* 的校准过程。

【程序分析题 4】分析 `ADC_Multi_DMA_Init` 函数中，为什么 GPIO 的引脚模式配置为 `GPIO_Mode_AIN` 模拟输入，而不是浮空输入 `GPIO_Mode_IN_FLOATING`？

【参考答案】 `GPIO_Mode_AIN` 是专门的模拟输入模式。在该模式下，GPIO 端口内部的数字输入施密特触发器会被强制关闭，且内部的上下拉电阻也会被断开，使引脚内部模拟链路直接连通到 ADC 上。这可最大程度减少引脚内部数字门电路的漏电流和高频数字噪声的电平干扰，确保模拟信号测量的纯净度。

【考点解析】 考查 *AIN* 模式与 *GPIO* 内部数字电路特性的配合原理。

【程序设计题 5】计算并编写转换函数：已知测得的 12 位 ADC 数字量为 `adc_val`，以公式计算出对应的模拟电压值 `vol`（浮点型，单位 V）并返回。

【参考答案】 ```c\nfloat Get_Voltage(uint16_t adc_val)\n{\n   return (float)adc_val * 3.3f / 4095.0f;\n}\n```

【考点解析】 考查 *ADC* 分辨率数字量转换公式。

★ Experiment 9: Timer Input Capture & PWM Measure
实验九：定时器输入捕获与脉宽测量实验

■ 一、 实验目的与核心任务

掌握通用定时器输入捕获（Input Capture）模式的配置方法，理解捕获电平跳变锁存计数器的工作原理。结合实验五的 PWM 输出，实现通过定时器5通道1捕获测量外界输入高电平信号的精确宽度（微秒级）。

■ 二、 硬件原理与引脚连接

- TIM3_CH2 (PA7) 配置为 PWM 输出脚，生成频率 2kHz（周期 500us）、占空比可调的波形。
- TIM5_CH1 (PA0) 配置为输入捕获脚，利用杜邦线将 PA7 与 PA0 短接，形成自发自收闭环测试回路。

■ 三、 软件设计底层原理

1. 输入捕获原理：当配置的引脚上检测到指定的电平跳变（如上升沿或下降沿）时，定时器的硬件控制器会自动将当前计数器值 CNT 锁存到捕获/比较寄存器 CCR 中，同时产生捕获中断。

2. 测量高电平宽度的状态机逻辑：

- **步骤一**：配置通道为**上升沿捕获**，开启捕获中断。当输入信号从低电平跳变为高电平（上升沿）时触发中断，记录此时的锁存值，或将计数器 CNT 直接清零，开始重新计时。然后将捕获极性修改为**下降沿捕获**。

- **步骤二**：当高电平结束，信号从高电平跳变为低电平（下降沿）时触发中断。读取此时的 CCR 寄存器值，该值即为高电平持续的计数值。记录后，将捕获极性重新修改为**上升沿捕获**，准备测量下一个周期的高电平。

3. 时间换算公式：如果配置预分频系数使得计数器 1us 累加一次，则读取的计数值直接等于微秒数时间值。

■ 四、 关键寄存器与常用库函数

- - TIM_ICInit(TIM_TypeDef* TIMx, TIM_ICInitTypeDef* TIM_ICInitStruct): 配置输入捕获通道的输入选择、预分频比、滤波器及捕获极性。
- - TIM_OC2Init(TIM_TypeDef* TIMx, TIM_OCInitTypeDef* TIM_OCInitStruct): 配置比较输出模式参数。
- - TIM_OC2PreloadConfig(TIM_TypeDef* TIMx, uint16_t TIM_OCPreload): 使使能输出比较通道2的预装载配置。

■ 五、 核心程序（含逐行保姆级注释）

```
// === tim_pwm_capture.c ===
#include "tim_pwm_capture.h"
#include "stm32f10x.h"
```

```
// TIM3_CH2 PA7 输出 500us 周期 PWM 的初始化函数
void TIM3_PWM_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_OCInitTypeDef  TIM_OCInitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);

    // PA7 复用推挽输出
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    // 周期 500us = (ARR+1) * (PSC+1) / 72MHz = 500 * 72 / 72,000,000
    TIM_TimeBaseStructure.TIM_Period = 500-1;                 // ARR = 499
    TIM_TimeBaseStructure.TIM_Prescaler = 72-1;               // PSC = 71
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseStructure);

    // PWM 模式 1
    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High; // 高电平有效

    TIM_OC2Init(TIM3, &TIM_OCInitStructure);
    TIM_OC2PreloadConfig(TIM3, TIM_OCPreload_Enable);
    TIM_ARRPreloadConfig(TIM3, ENABLE);
    TIM_Cmd(TIM3, ENABLE);
}

// 动态修改占空比的比较值
void TIM3_SetDuty(uint16_t duty)
{
    TIM_SetCompare2(TIM3, duty);
}

// TIM5_CH1 PA0 输入捕获配置函数（测量脉宽，1us计数一次）
void TIM5_Capture_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
    TIM_ICInitTypeDef  TIM_ICInitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;

    // 1. 开启时钟：TIM5 (APB1) 和 GPIOA
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM5, ENABLE);
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

    // 2. 配置 PA0 为“浮空输入”模式
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
```

```
    // 3. 配置 TIM5 计数器：1us计数一次，最大计数 65535us
    TIM_TimeBaseStructure.TIM_Period = 65535;                    // 自动重装载值最大
    TIM_TimeBaseStructure.TIM_Prescaler = 72-1;                  // 72分频，频率1MHz
    TIM_TimeBaseStructure.TIM_ClockDivision = 0;
    TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;
    TIM_TimeBaseInit(TIM5, &TIM_TimeBaseStructure);

    // 4. 初始化 TIM5 输入捕获通道1 (IC1)
    TIM_ICInitStructure.TIM_Channel = TIM_Channel_1;             // 配置通道 1
    TIM_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;// 初始配置为“上升沿捕获”
    TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; // 直连输入通道
    TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;      // 不分频，每次跳变都捕获
    TIM_ICInitStructure.TIM_ICFilter = 0x00;                     // 滤波器不使使能
    TIM_ICInit(TIM5, &TIM_ICInitStructure);

    // 5. 开启捕获中断，并配置 NVIC 中断优先级
    TIM_ITConfig(TIM5, TIM_IT_CC1, ENABLE); // 开启比较捕获1中断

    NVIC_InitStructure.NVIC_IRQChannel = TIM5_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);

    TIM_Cmd(TIM5, ENABLE); // 开启 TIM5
}

// === stm32f10x_it.c (捕获状态机逻辑) ===
#include "stm32f10x_it.h"
#include "stm32f10x.h"

// 记录高电平宽度的变量 (us)
uint32_t high_level_width = 0;
// 状态标志：0表示尚未捕获到上升沿，1表示已捕获到上升沿，正在等待下降沿
uint8_t capture_state = 0;

void TIM5_IRQHandler(void)
{
    // 检查是否是输入捕获1通道的中断（CC1 - Capture/Compare 1 Flag）
    if (TIM_GetITStatus(TIM5, TIM_IT_CC1) != RESET)
    {
        if (capture_state == 0) // 状态0：捕获到上升沿
        {
            TIM_SetCounter(TIM5, 0); // 1. 定时器计数值 CNT 清零，重新开始计时
            TIM_OC1PolarityConfig(TIM5, TIM_ICPolarity_Falling); // 2. 将捕获极性修改为“下降沿捕获”
            capture_state = 1;       // 3. 标记进入等待下降沿状态
        }
        else // 状态1：捕获到下降沿
        {
            // 1. 读取捕获比较寄存器 CCR1 的值，该值即为高电平持续的微秒数
```

```
            high_level_width = TIM_GetCapture1(TIM5);
            TIM_OC1PolarityConfig(TIM5, TIM_ICPolarity_Rising);  // 2. 将极性重新
恢复为“上升沿捕获”
            capture_state = 0;        // 3. 标记回到初始状态，等待下一个脉冲
        }

        // 清除捕获中断挂起位
        TIM_ClearITPendingBit(TIM5, TIM_IT_CC1);
    }
}
```

■ 六、 高频常考预测题库（单选/判断/简答/分析/设计）

【单选题 1】在定时器输入捕获实验中，在检测到上升沿后将计数器清零，若再次捕获到下降沿时计数器未溢出，高电平脉宽时间如何计算？（）

A. 直接读取 CCR 寄存器的值

B. 读取 CCR 寄存器的值并加 1

C. 读取 CNT 寄存器的值

D. 将两次 CCR 寄存器的差值相减

【参考答案】 A

【考点解析】 因为在检测到上升沿时，计数器 *CNT* 已经清零重计。因此，在下降沿发生时，捕获寄存器 *CCR* 锁存的值即是从上升沿到下降沿之间定时器计数脉冲的累计个数，直接等于该高电平的时间。

【判断题 2】输入捕获功能不能用于测量方波信号的频率，只能用于测量脉冲的宽度。（）

【参考答案】 错误

【考点解析】 输入捕获不仅可以测脉宽，也能方便地测出周期和频率：配置为连续两次上升沿捕获，两次记录的计数值差值即为信号周期，取倒数即是频率。

【简答题 3】在输入捕获中断服务程序中，为什么需要在捕获到上升沿后动态调用 `TIM_OC1PolarityConfig` 切换捕获极性?

【参考答案】 因为测量高电平的宽度需要捕获高电平的两个关键边界：起点（上升沿）和终点（下降沿）。如果保持上升沿捕获，当下降沿到来时定时器无法感知。因此必须在捕获到上升沿起点后，立即将捕获极性更改为下降沿，以触发下降沿中断来捕捉终点，测量结束后再恢复为上升沿以便循环工作。

【考点解析】 考察输入捕获状态机的基本逻辑。

【程序分析题 4】在 `TIM5_IRQHandler` 中，如果测得的脉宽超长（如 100 毫秒），而 TIM5 溢出重载 ARR 仅配置为 65535us。如果不加额外逻辑处理，测量的结果会正确吗？为什么？应该怎么解决?

【参考答案】 不正确。因为脉宽时间（100ms）超出了定时器的最大单次计数范围（65.5ms）。在等待下降沿期间，计数器 CNT 会发生计数溢出并重装，导致最终读取的 CCR 值变得比真实值小很多。解决办法是：开启定时器的更新中断（Update Interrupt），并在全局定义一个计数溢出次数变量 `overflow_cnt`。在更新中断里累加该变量，最终脉宽时间计算为 `overflow_cnt * 65536 + CCR`。

【考点解析】 考查定时器溢出与脉宽补偿机制。

【程序设计题 5】编写将 TIM5 通道 1 极性更改为下降沿捕获的代码语句。

【参考答案】 `TIM_OC1PolarityConfig(TIM5, TIM_ICPolarity_Falling);` 或 `TIM5->CCER |= TIM_CCER_CC1P;`

【考点解析】 考查捕获边沿切换的配置函数。