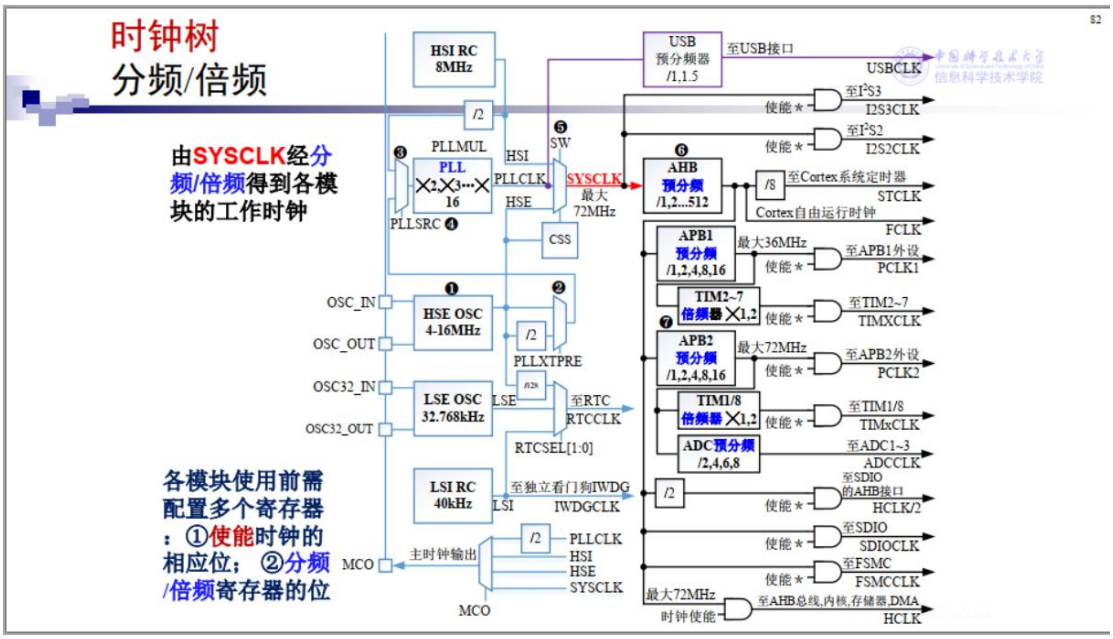
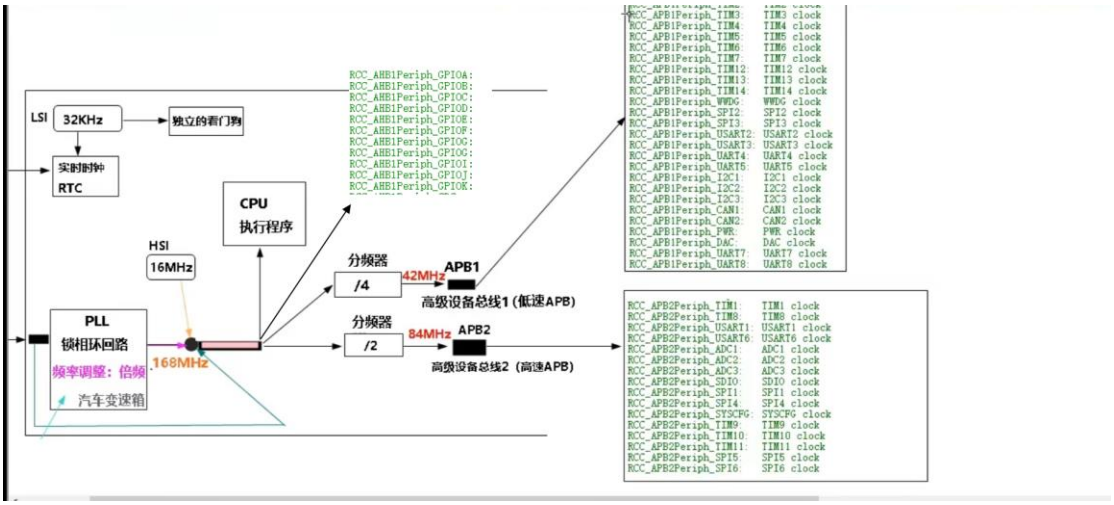
计嵌第八章部分知识点　张旭老师班　　助教 杨冬

一、时钟树

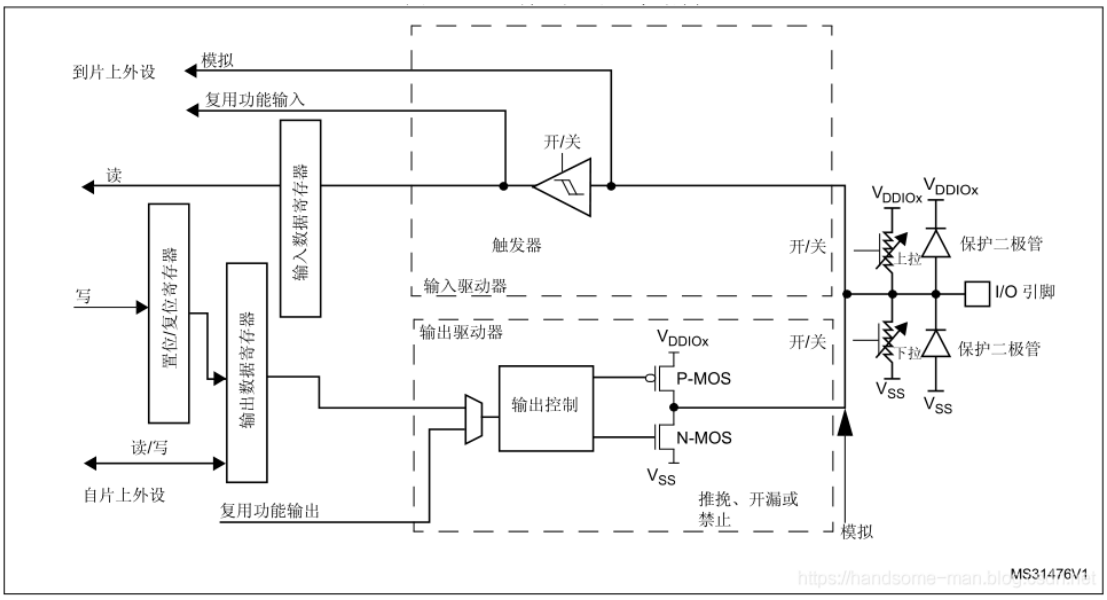


时钟树简化版本如下



二、GPIO 配置流程

GPIO 电路图



(一)、以 GPIO 端口作为普通输出或普通输入时的配置

按照以下三个步骤完成对 GPIO 端口的配置

```
//使能(打开)端口 G 的硬件时钟，就是对端口 F 供电
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOG, ENABLE);
//初始化 GPIO 引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11;          //第 11 根引脚
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;  // 输出模式，若 GPIO 要从外引入信
                                                    号需要配置为GPIO_Mode_IN
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;//推挽输出，增加输出电流能力。
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//高速响应
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;//没有使能上下拉电阻
//使用固件库函数对 GPIOG 相关寄存器组初始化
GPIO_Init(GPIOG, &GPIO_InitStructure);
```

(二)、以 GPIO 端口作为其他外设的输出时，需要设置为复用模式

如在 PWM 实验中引脚 PB10 作为定时器 2 的输出，配置如下

```
//使能(打开)端口 B 的硬件时钟，就是对端口 F 供电
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
//初始化 GPIO 引脚
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;          //第 10 根引脚
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF; //输出模式
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;  //推挽输出，增加输出电流能力。
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;//高速响应
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;    //没有使能上下拉电阻
//使用固件库函数对 GPIOG 相关寄存器组初始化
GPIO_Init(GPIOB, &GPIO_InitStructure);
//将 PB10 引脚连接到定时器 2
GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_TIM2);
```

GPIO 相关数据结构

```
typedef struct
{
  uint32_t GPIO_Pin;             /*!< Specifies the GPIO pins to be configured.
                                      This parameter can be any value of @ref GPIO_pins_define */

  GPIOMode_TypeDef GPIO_Mode;    /*!< Specifies the operating mode for the selected pins.
                                      This parameter can be a value of @ref GPIOMode_TypeDef */

  GPIOSpeed_TypeDef GPIO_Speed;  /*!< Specifies the speed for the selected pins.
                                      This parameter can be a value of @ref GPIOSpeed_TypeDef */

  GPIOOType_TypeDef GPIO_OType;  /*!< Specifies the operating output type for the selected pins.
                                      This parameter can be a value of @ref GPIOOType_TypeDef */

  GPIOPuPd_TypeDef GPIO_PuPd;    /*!< Specifies the operating Pull-up/Pull down for the selected pins.
                                      This parameter can be a value of @ref GPIOPuPd_TypeDef */
}GPIO_InitTypeDef;
```

GPIO 模式选择

```
typedef enum
{
  GPIO_Mode_IN   = 0x00, /*!< GPIO Input Mode */
  GPIO_Mode_OUT  = 0x01, /*!< GPIO Output Mode */
  GPIO_Mode_AF   = 0x02, /*!< GPIO Alternate function Mode */
  GPIO_Mode_AN   = 0x03  /*!< GPIO Analog Mode */
}GPIOMode_TypeDef;
```

输出方式：推挽/普通输出

```
typedef enum
{
  GPIO_OType_PP = 0x00,
  GPIO_OType_OD = 0x01
}GPIOOType_TypeDef;
```

输出频率选择

```
typedef enum
{
  GPIO_Low_Speed     = 0x00, /*!< Low speed     */
  GPIO_Medium_Speed  = 0x01, /*!< Medium speed */
  GPIO_Fast_Speed    = 0x02, /*!< Fast speed    */
  GPIO_High_Speed    = 0x03  /*!< High speed    */
}GPIOSpeed_TypeDef;
```

是否电阻上拉或者下拉，一般在外部负载无上下拉电阻时才配置

```
typedef enum
{
  GPIO_PuPd_NOPULL = 0x00,
  GPIO_PuPd_UP     = 0x01,
  GPIO_PuPd_DOWN   = 0x02
}GPIOPuPd_TypeDef;
```

三、定时器

定时器分为系统定时器和硬件定时器

在 STM32F407 系列微控制器中，存在两种类型的定时器：系统定时器（SysTick）和硬件定

时器（TIM）。

1. 系统定时器（SysTick）：

系统定时器是 STM32F407 微控制器中内置的定时器。它可以用来提供基本的系统定时功能，通常用于操作系统或实时任务的时间管理。系统定时器具有以下特点：

- 24 位的倒计数器，可以配置为不同的时钟源（通常使用 CPU 时钟）。
- 可以通过加载计数器值，设置重装载值和使能中断来实现定时功能。
- 它可以用作滴答定时器并产生特定的时间间隔中断。

2. 硬件定时器（TIM）：

STM32F407 微控制器提供了多个硬件定时器（TIM1-TIM17），它们是高级定时器单元，并且具有高度可配置的功能。硬件定时器主要用于生成精确的定时信号、测量脉冲宽度和频率、PWM 输出等。一些常见的特点如下：

- 多个独立的定时器通道和计数器，可以独立配置和使用。
- 支持多种工作模式，包括定时计数、输入捕获、输出比较、PWM 生成等。
- 可以配置不同的时钟源（如内部时钟、外部时钟或其他定时器作为时钟源）。
- 支持中断、DMA 传输和触发输出等功能，可以与其他外设模块进行有效的协同工作。

总结：

系统定时器（SysTick）适用于简单的系统定时需求和低精度的时间管理，而硬件定时器（TIM）则具有更多的高级功能和更高的精度，适用于各种定时和测量应用场景。开发者可以根据具体的需求选择使用适当的定时器。

系统定时器配置流程

使用系统定时器对 LED 灯进行定时闪烁

直接使用函数 SysTick_Config(SystemCoreClock / 10);设置相关的周期，在重写 void SysTick_Handler(void)函数即可

硬件定时器配置流程

原理说明

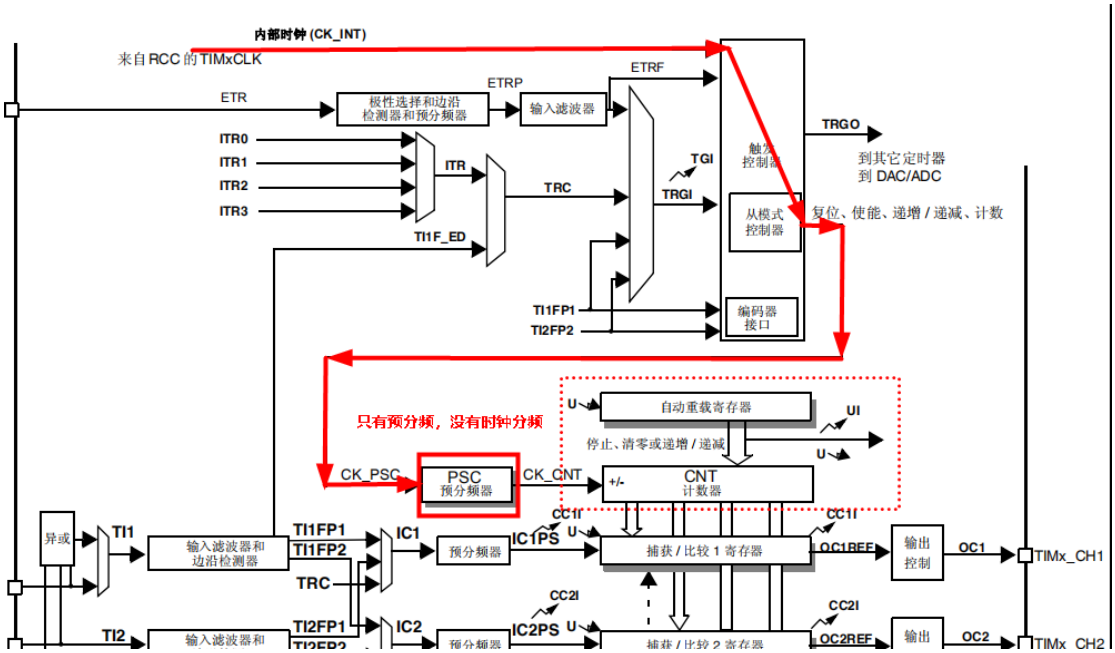
基本定时器中，16 位计数器只能工作在向上计数模式，自动重装载寄存器 ARR 保存定时器溢出值。

工作时，计数器从 0 开始，在定时器时钟 CK_CNT 触发下不断累加计数。当计数器 CNT 计数值等于 ARR 中保存的预设值时，产生溢出事件，可触发中断或 DMA 请求（已使能时）。然后， CNT 计数值被清零，重新开始向上计数。

基本定时器延时时间，可由以下公式计算：延时时间=(ARR+1)*(PSC+1)/TIMxCLK

基本定时器

电路图说明



其中内部时钟指从 APB1 或 APB2 的时钟，APB1 为 84MHz APB2 为 168MHz

对硬件定时器的配置主要体现在对预分频器和计数器的设置，以及计数方式的确定，基本流程如下

```
//使能定时器 2 硬件时钟
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
//配置定时器 2 分频值、计数值等
TIM_TimeBaseStructure.TIM_Period = (10000/100) - 1;//计数值，0~4999，决定定时时间为 1/2 秒
TIM_TimeBaseStructure.TIM_Prescaler = 8400 - 1;  //8400-1+1=8400,进行 8400 的预分频值，进行第一次分频
TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数的方法
TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure)
//配置定时器 2 中断的触发方式：时间更新
TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
```

因为使用定时器一般都是主程序被动等待定时，所以一般需要和中断相关联

```
//配置定时器 2 的中断优先级
NVIC_InitStructure.NVIC_IRQChannel = TIM2_IRQn;
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStructure);
//使能定时器 2 工作
TIM_Cmd(TIM2, ENABLE);
```

除此以外还需要自己重写相关中断处理函数如本次需要重写void TIM2_IRQHandler()函数

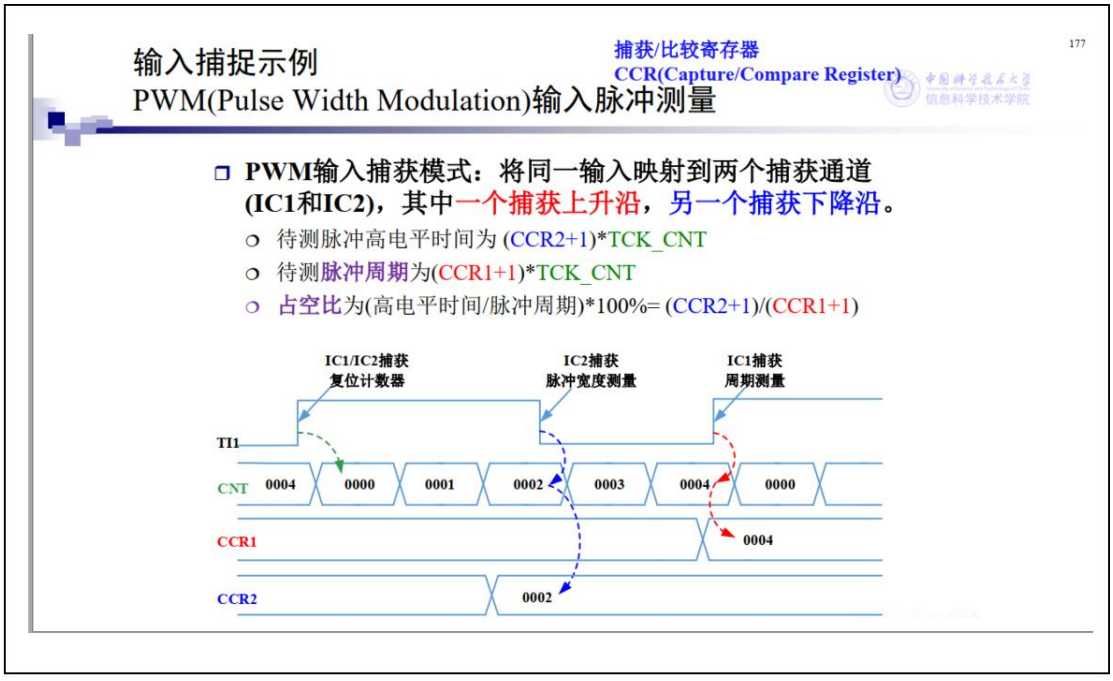
```
void TIM2_IRQHandler()
{
    //判断标志位
    if(SET==TIM_GetITStatus(TIM2, TIM_IT_Update))
    {
        PGout(11)^=1;

        //清空标志位
        TIM_ClearITPendingBit(TIM2,TIM_IT_Update);
    }
```

使用定时器完成呼吸灯功能(PWM 实验)

定时器中可以完成 PWM 输入和 PWM 输出两种功能

PWM 输入把输入映射到两个捕获通道，一个捕获上升沿，并记录当时计数器 CNT1 值，另一个捕获通道捕获下降沿，并记录捕获时的 CNT2 值，使用 T*(CNT2-CNT1)计算输入脉冲的周期，示意图如下

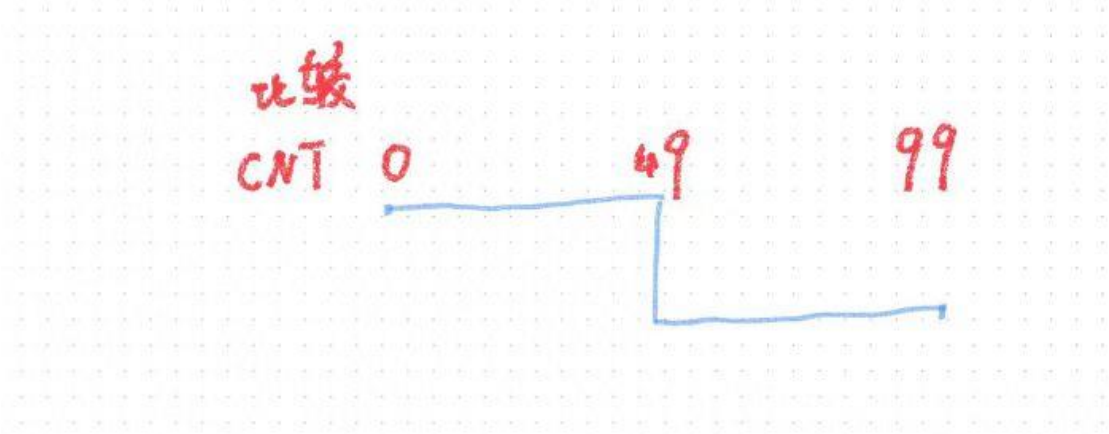


PWM 输出则是预设比较寄存器，在定时器计数值 CNT 在小于和大于比较寄存器值时产生高低两种不同的电平，从而产生不同占空比的脉冲信号，如 CNT 变化范围为 0~99，假设当 CNT 小于比较寄存器值时产生高电平，大于时产生低电平图示如下

假设比较寄存器值为 49，占空比为 50%，脉冲如下

CNT：0 → 50 → 99

产生高电平　　产生低电平



PWM 配置代码如下

```
//使能定时器 2 硬件时钟
RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);
//配置定时器 2 分频值、计数值等
TIM_TimeBaseStructure.TIM_Period = (10000/100) - 1;      //计数值，0~99，决定定时时间为 10ms 秒
TIM_TimeBaseStructure.TIM_Prescaler = 8400 - 1;      //8400-1+1=8400,进行 8400 的预分频值，进行第一次分频

 TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;//向上计数的方法

TIM_TimeBaseInit(TIM2, &TIM_TimeBaseStructure);
//使能定时器 2 工作
TIM_Cmd(TIM2, ENABLE);
//通道 1 工作在 PWM1 模式
TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;
TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;//打开/关闭脉冲输出
TIM_OCInitStructure.TIM_Pulse = 50;                    //比较值
TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;  //有效状态为高电平，则无效状态为低电平
TIM_OC3Init(TIM2, &TIM_OCInitStructure);
```

使用函数TIM_SetCompare1(TIM2, pwm); 对占空比进行动态设置

四、中断

在 STM32F407 微控制器中，中断是一种重要的事件处理机制，可以用于响应外部或内部事件，以及实时处理各类任务。以下是 STM32F407 中断的基本介绍：

1. 中断控制器（NVIC）：

STM32F407 具有嵌入式的中断控制器（Nested Vectored Interrupt Controller，NVIC），用于管理和控制中断。NVIC 用于优先级管理、中断向量表的索引和中断使能等操作。

2. 可中断的外部事件：

STM32F407 可以接收和处理各种外部事件的中断，如外部 GPIO 引脚的状态改变、定时器和串口的事件等。通过配置相关的中断线和中断优先级，可以在具体的外部事件发生时触发中断处理程序。

3. 内部事件和系统异常：

STM32F407 还具有一些内部事件和系统异常，例如硬件错误、系统时钟溢出、DMA 传输完成等。这些事件可以通过在系统配置中使能相应的中断来触发中断处理程序。

4. 中断处理程序（Interrupt Service Routine，ISR）：

中断处理程序是一个特殊的函数，用于处理中断事件。当中断发生时，控制器会自动跳转到相应的中断向量，并执行对应的中断处理程序。在中断处理程序中，可以执行特定的处理逻辑、更新状态或采取其他行动。

5. 中断优先级：

中断优先级在 NVIC 中起到重要的作用，决定了中断的响应顺序。具有较高优先级的中断将优先执行，而低优先级的中断则可能会被屏蔽。可以根据应用需求配置中断的优先级。

6. 中断嵌套：

在 STM32F407 中，支持嵌套中断。这意味着，当一个中断正在执行时，可以被一个更高优先级的中断中断，并暂停当前中断的执行。嵌套中断允许实时任务和紧急事件得到及时响应。

7. 中断清除：

在中断处理程序中，需要明确清除中断标志，以便允许更多的中断事件被处理。清除中断标志可以通过读取或写入相关寄存器的方式实现。

总结：

中断是 STM32F407 中一种重要的事件处理机制，通过中断处理程序和优先级管理，可以实现对外部和内部事件的及时响应和处理。开发者可以根据具体的需求，配置和管理中断，以实现特定的功能和任务。

其他详细内容可参考老师 ppt 或教材讲义

外部中断基本配置过程

```
EXTI_InitTypeDef   EXTI_InitStructure;
GPIO_InitTypeDef   GPIO_InitStructure;
NVIC_InitTypeDef   NVIC_InitStructure;

  /* Enable GPIOA clock */
   RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
   /* Enable SYSCFG clock */
   RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);

   /* Configure PA0 pin as input floating */
   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;
   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
   GPIO_Init(GPIOA, &GPIO_InitStructure);
   /* Connect EXTI Line0 to PA0 pin */
   SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA,EXTI_PinSource0);

   /* Configure EXTI Line0 */
   EXTI_InitStructure.EXTI_Line = EXTI_Line0;
   EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
   EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
```

```
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
    EXTI_Init(&EXTI_InitStructure);

    /* Enable and set EXTI Line0 Interrupt to the lowest
priority */
    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
```

五、USART

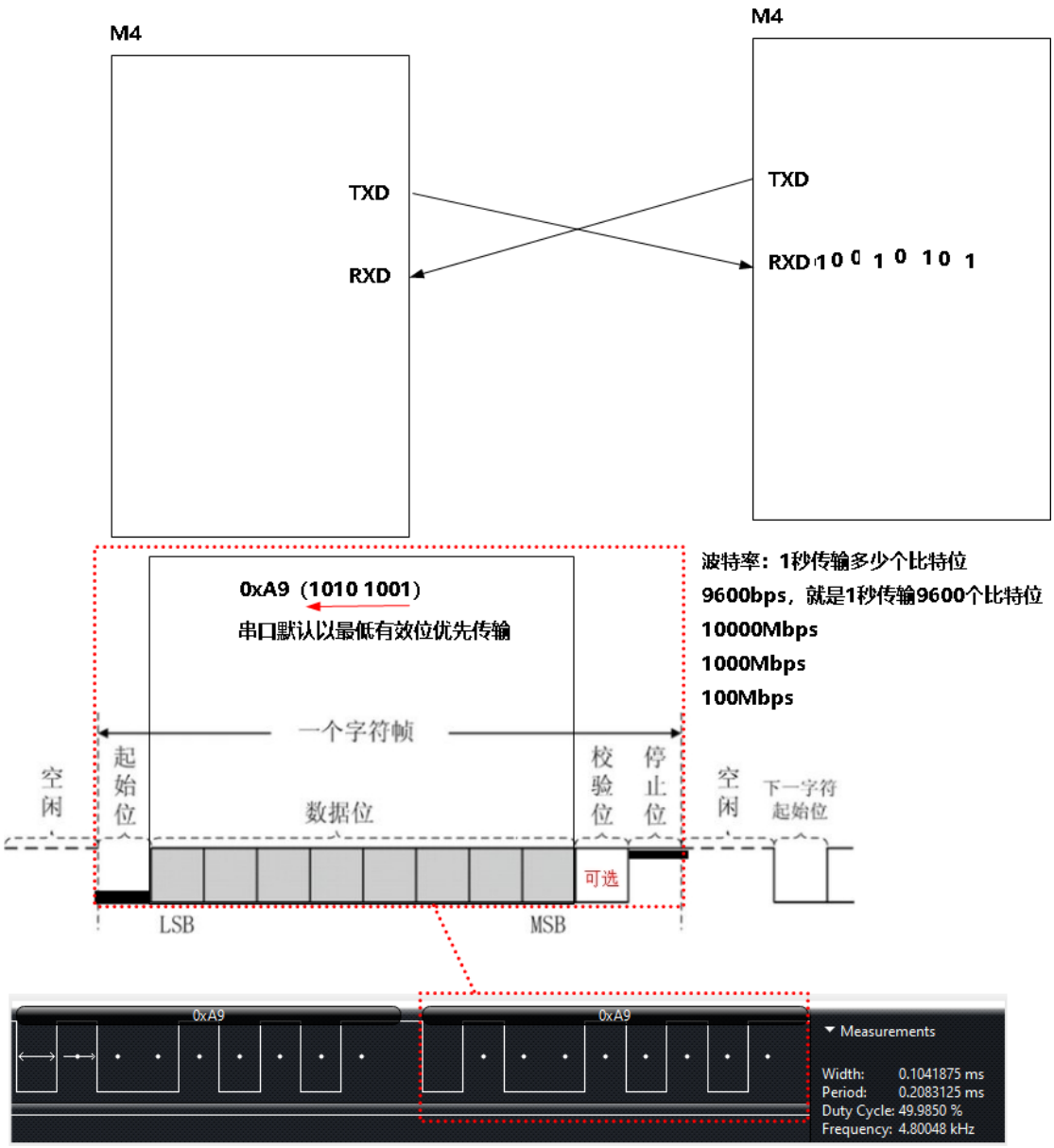
USART（Universal Synchronous/Asynchronous Receiver Transmitter）是一种常用的串行通信接口，用于实现与外部设备（如计算机、传感器、模块等）之间的数据传输。

1. 特点：
   - 支持全双工通信，同时具有发送和接收功能。
   - 可以实现多种数据格式，例如异步通信（UART）和同步通信（SPI、I2C）。
   - 提供多个 USART 外设，具有多个串口通信通道。
2. USART 的工作模式：
   - 异步模式（UART）：数据以异步方式传输，不需要外部时钟信号，在其中包括起始位、数据位、校验位和停止位。
   - 同步模式：数据以同步方式传输，使用外部时钟信号，如 SPI 和 I2C。
3. USART 接口的硬件结构：
   - USART 包含一个发送寄存器和一个接收寄存器，用于发送和接收数据。
   - 支持多种中断，如发送完成中断、接收完成中断、字节发送完成中断等。
   - 支持硬件流控制标准，如 RTS/CTS 和 DTR/DSR。
4. USART 配置：
   - 配置 USART 使用的引脚和引脚功能。
   - 配置 USART 的通信参数，如波特率、数据位数、停止位数、校验位等。
   - 配置 USART 的工作模式，如异步模式或同步模式。
   - 配置中断使能和中断优先级。
   - 配置硬件流控制（可选）。
5. 数据传输和接收：
   - 数据的发送：将要发送的数据写入发送寄存器，USART 将自动发送数据。
   - 数据的接收：通过读取接收寄存器获取接收到的数据。

总结：

在 STM32F407 微控制器中，USART 是一种可配置的串行通信接口，可以实现与外部设备之间的数据传输。可以根据具体需求配置 USART 的通信参数、工作模式和中断，实现可靠的数据收发。USART 在嵌入式系统中广泛应用于与其他设备的通信，如串口调试、传感器数据采集、通信模块等。

简单图示



USART 配置流程

```
void init_USART1(unsigned int bound)
{
    //打开USART1 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1, ENABLE);
    //打开GPIOA 时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE);
    //GPIOA 初始化
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9|GPIO_Pin_10;
    GPIO_InitStructure.GPIO_OType=GPIO_OType_PP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    //GPIOA 端口复用
```

```
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource9, GPIO_AF_USART1);
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource10, GPIO_AF_USART1);
    //USART 初始化
    USART_InitStructure.USART_BaudRate = bound;
    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_Rx |
USART_Mode_Tx;
    USART_Init(USART1,  &USART_InitStructure);
    //中断方式，接收数据时产生中断
    USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);
    //中断设置
    NVIC_InitStructure.NVIC_IRQChannel = USART1_IRQn   ;
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority =
0x0F;
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0F;
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStructure);
    //开启 USART1
    USART_Cmd(USART1, ENABLE);
}
//中断处理函数
void USART1_IRQHandler(void)
{
    u8 Res;
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)
    {
        Res =USART_ReceiveData(USART1);

        //补充代码
        //对应操作 USART_RX_STA++;
        USART_RX_BUF[USART_RX_STA++]=Res;
        USART_ClearITPendingBit(USART1, USART_IT_RXNE);
    }
}
int main(void)
{
    char USART_SendBuf[]="Hello!";
    int i;
```

```
    int j=0;
    int Len=strlen(USART_SendBuf);
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    delay_ms(168);
    init_LCD_GPIO();
    init_LCD();
    init_USART1(115200);
    delay_ms(168);
    while(1)
    {
        delay_ms(2000);
        if(USART_TX_EN)
        {
            for(i=0;i<Len;i++)
            {
                USART_SendData(USART1, USART_SendBuf[i]);
                while(USART_GetFlagStatus(USART1,USART_FLAG_TXE
)!=SET);
            }
            USART_TX_EN=0;
        }
        if(USART_RX_STA==Len)
        {
            USART_TX_EN=1;
            USART_RX_STA=0;
            LCD_Display_Words(j,0,USART_RX_BUF);
            j++;
            if(j==4)
            {
                j=0;
                delay_ms(200);
                LCD_Clear();
            }

        }

    }

}
```