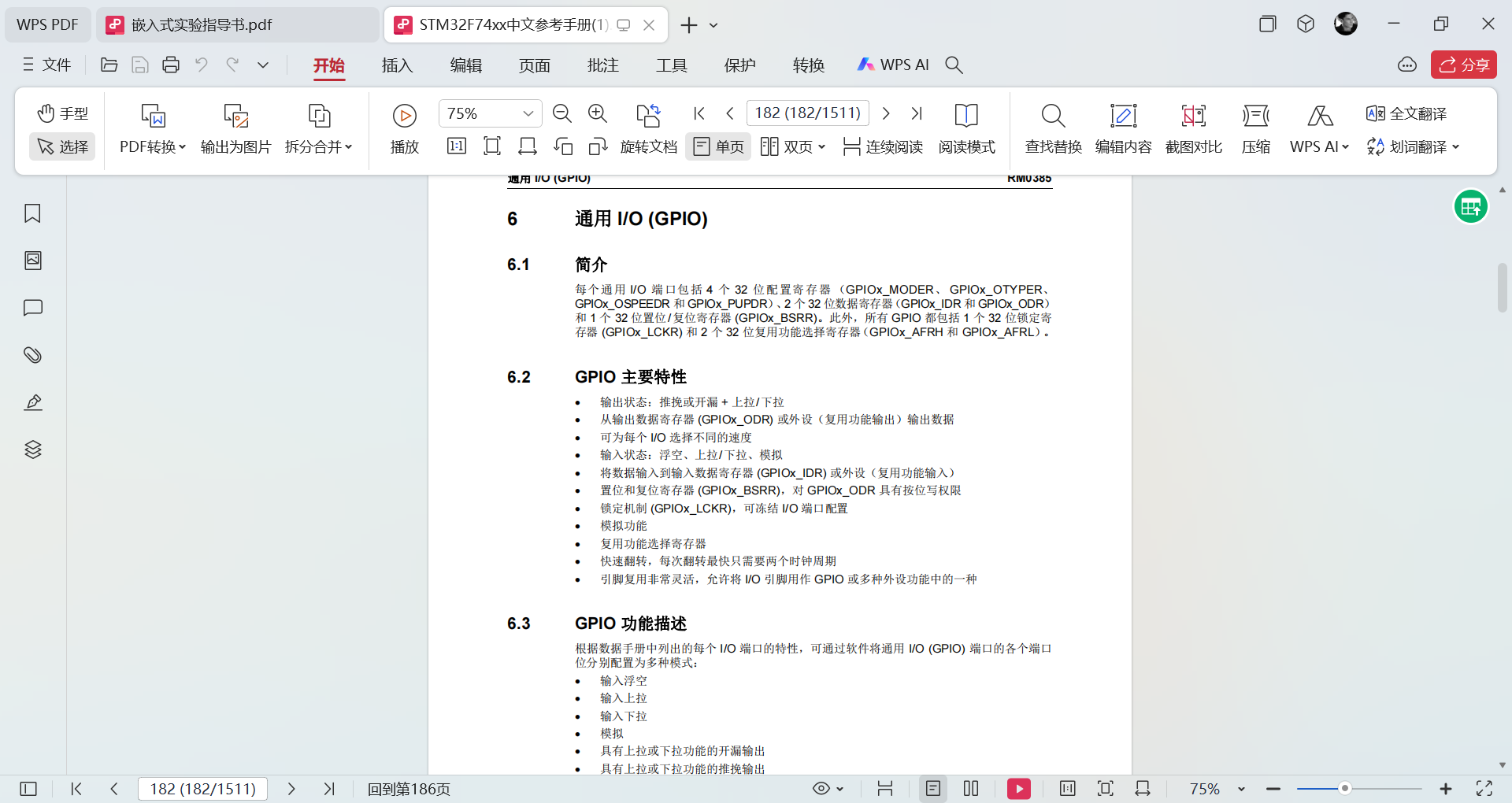
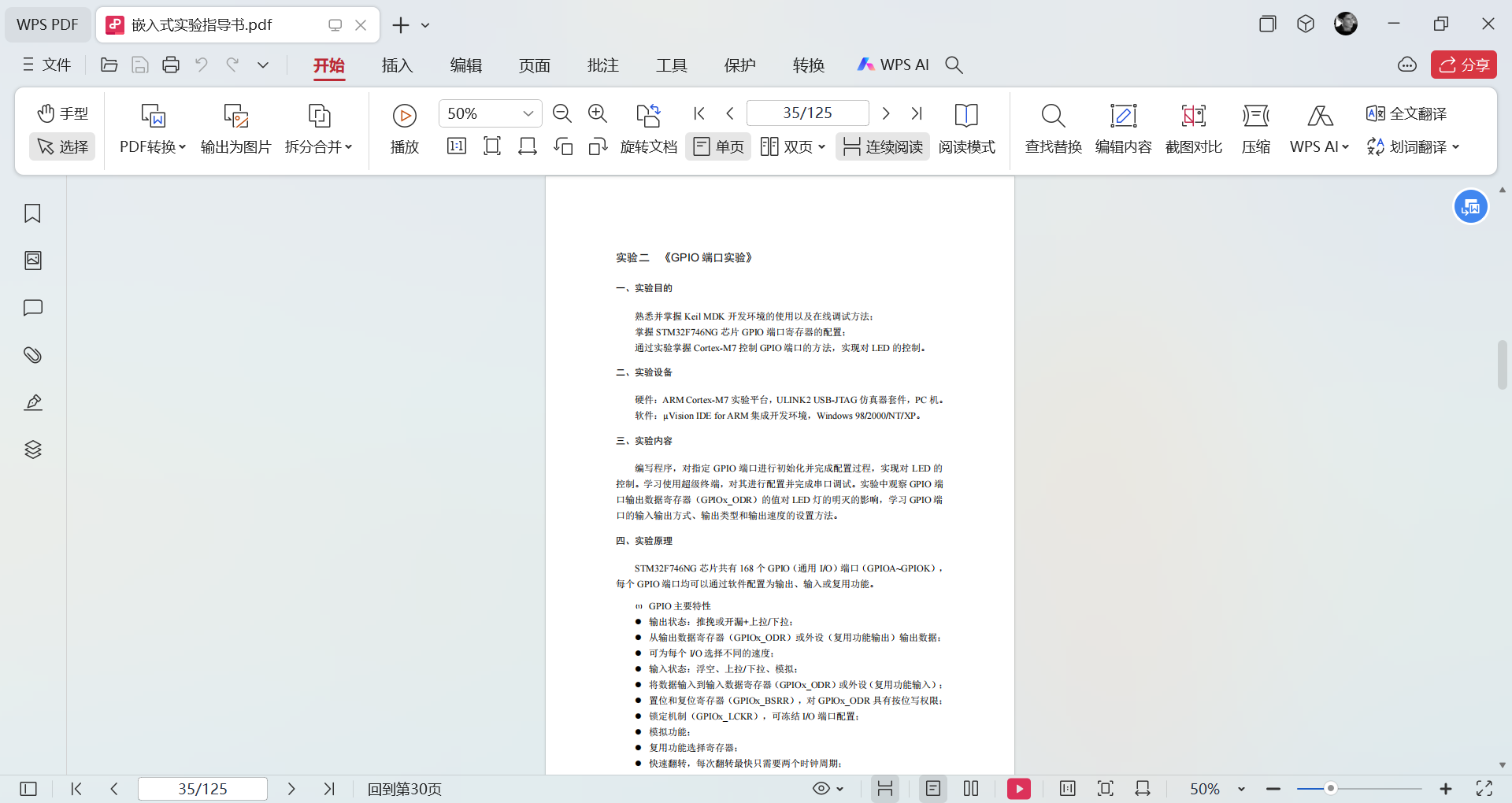
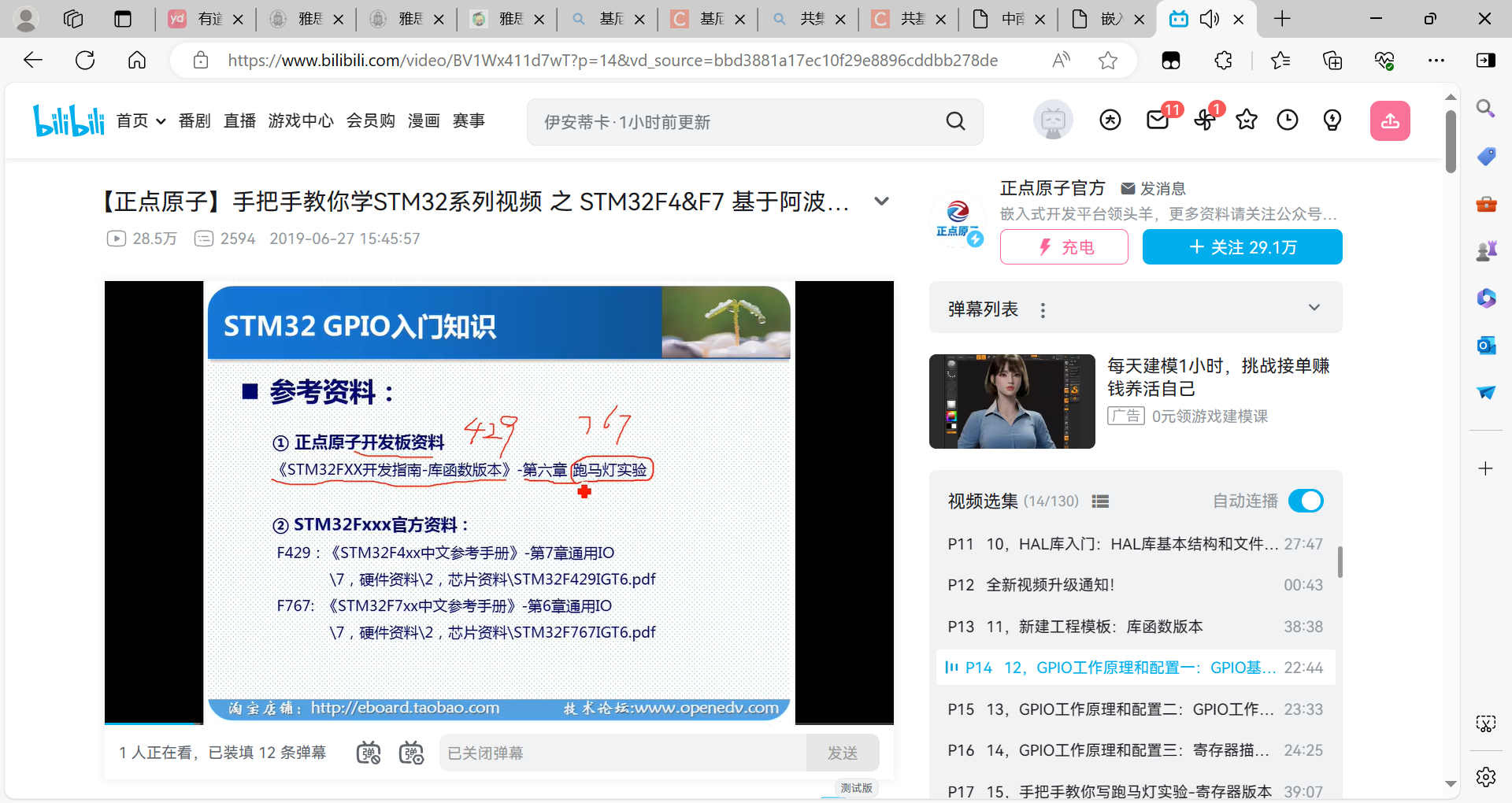
# 一、自学内容



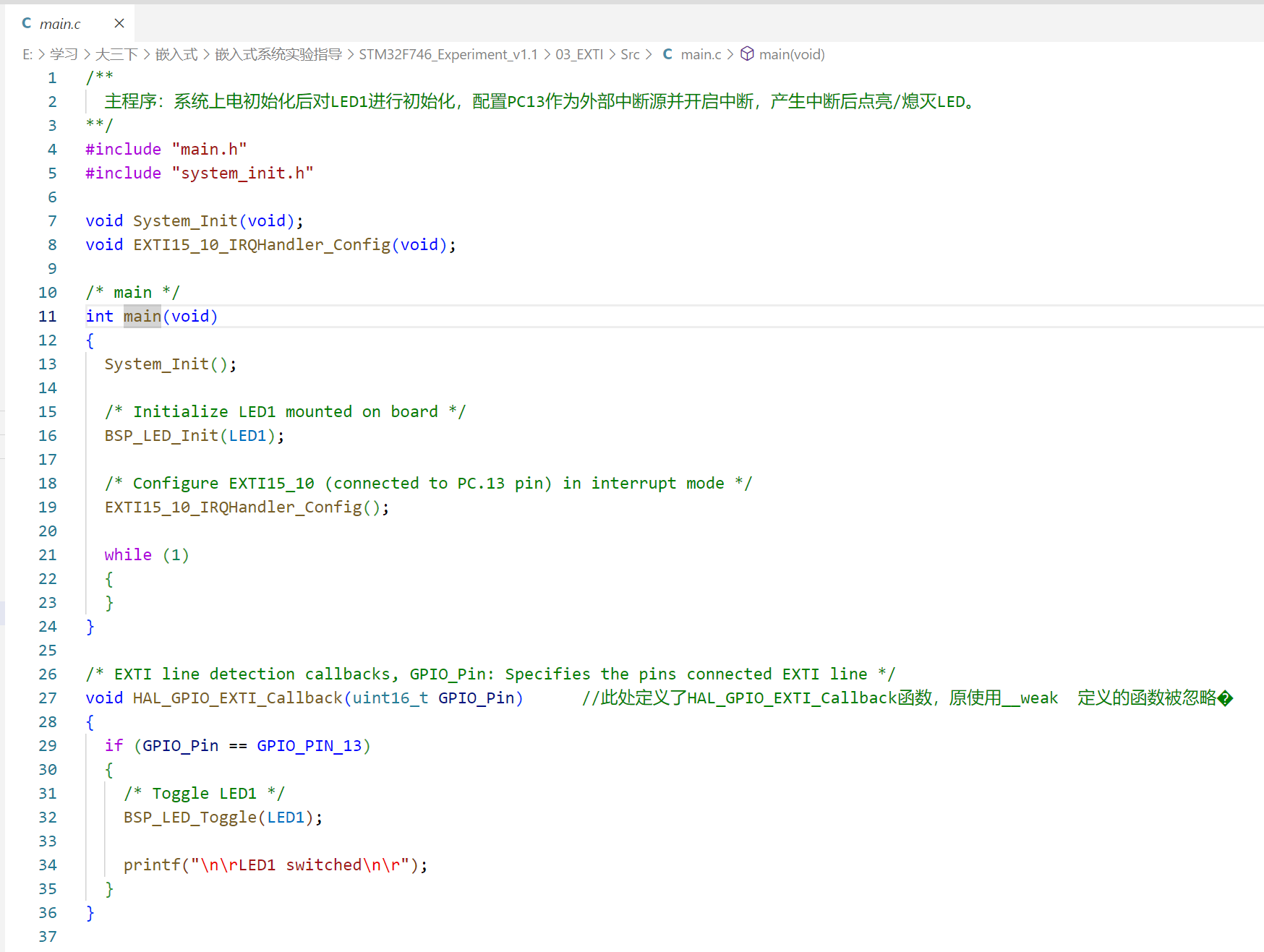




# 二、作业内容

## 1. 以STM32F746\_Experiment\_v1.1\03\_EXTI\Src\main.c函数为代码起点，通过向内层代码跳转（CTRL + 鼠标左键），弄清楚整个代码的执行流程（截图分析执行流程）。

main.c中的main函数

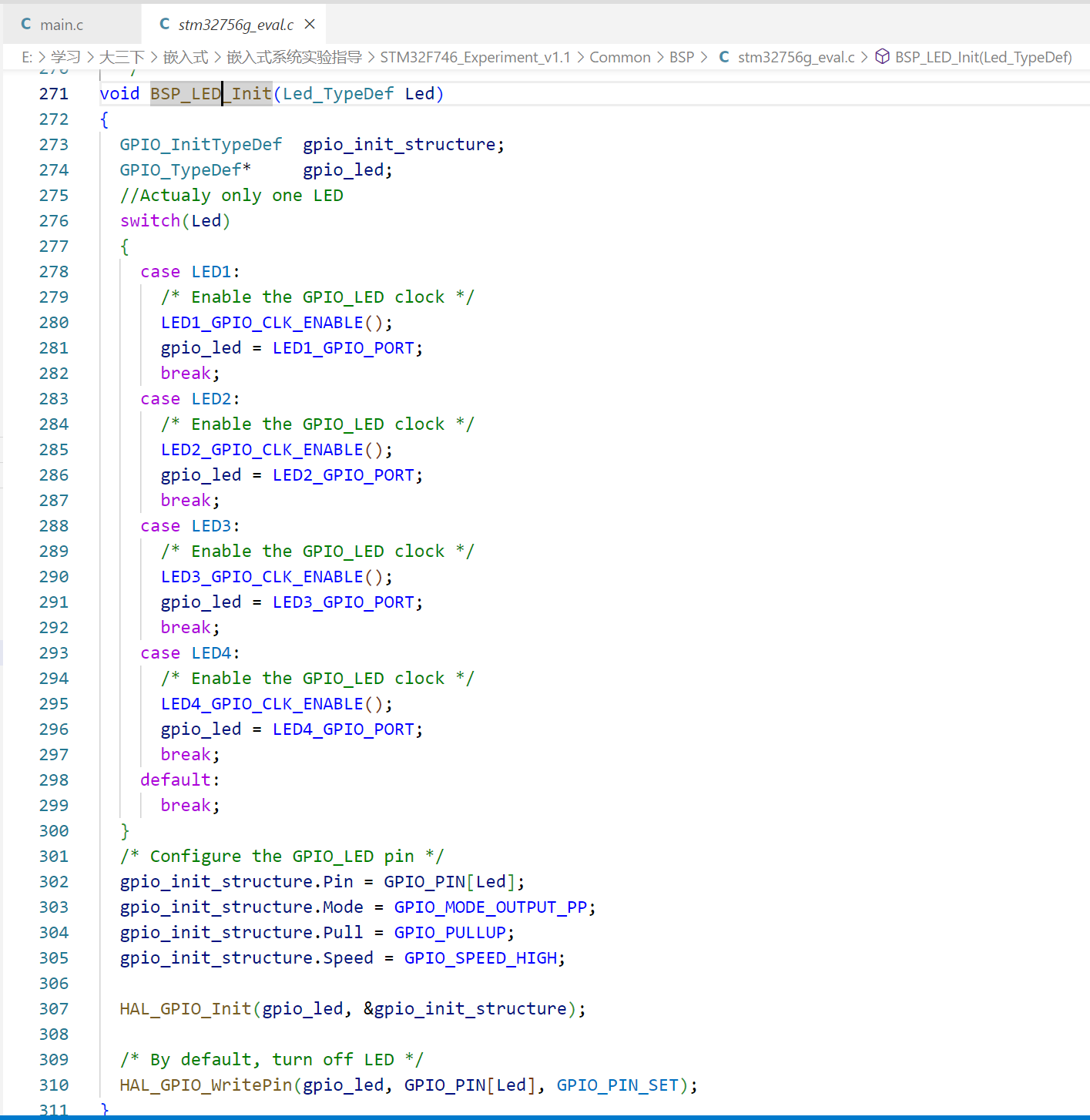


1. 首先调用System\_Init()函数进行系统初始化，跳转到对应界面得到的是



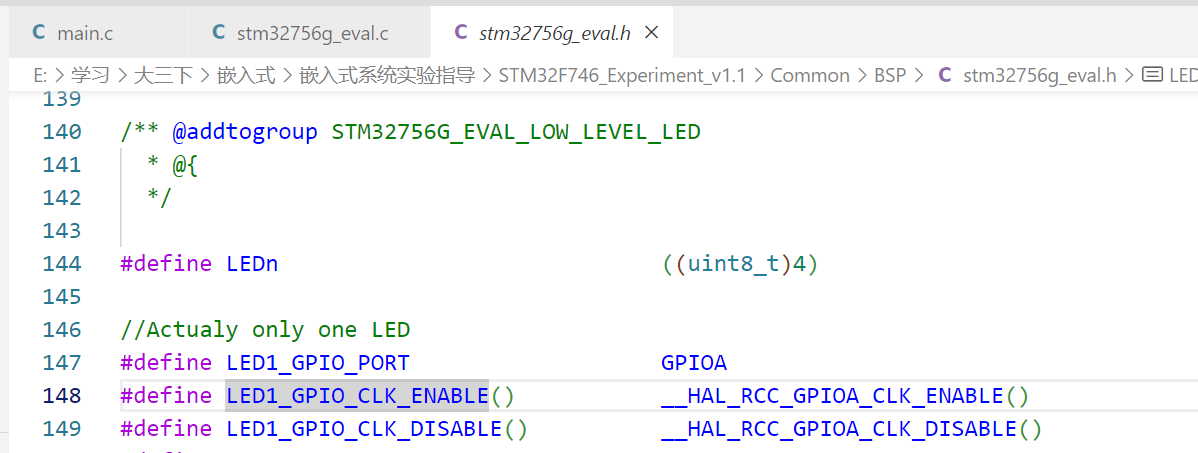
调用了四个函数初始化嵌入式系统

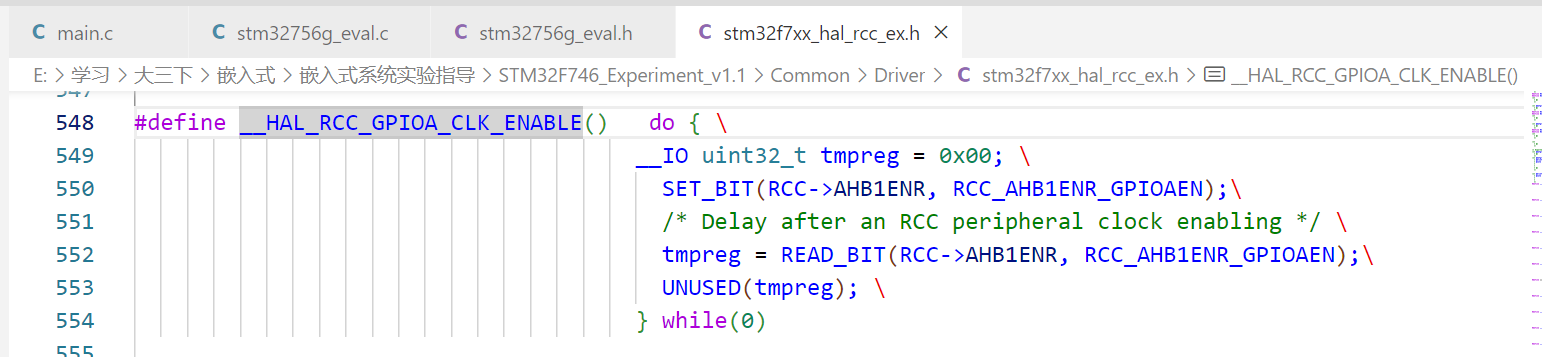
1. 接下来调用BSP\_LED\_Init() 初始化指定的LED，在这里我们初始化的是LED1



我们这里初始化了LED1，所以在函数中只执行case LED1，引脚编号也是设置的LED1

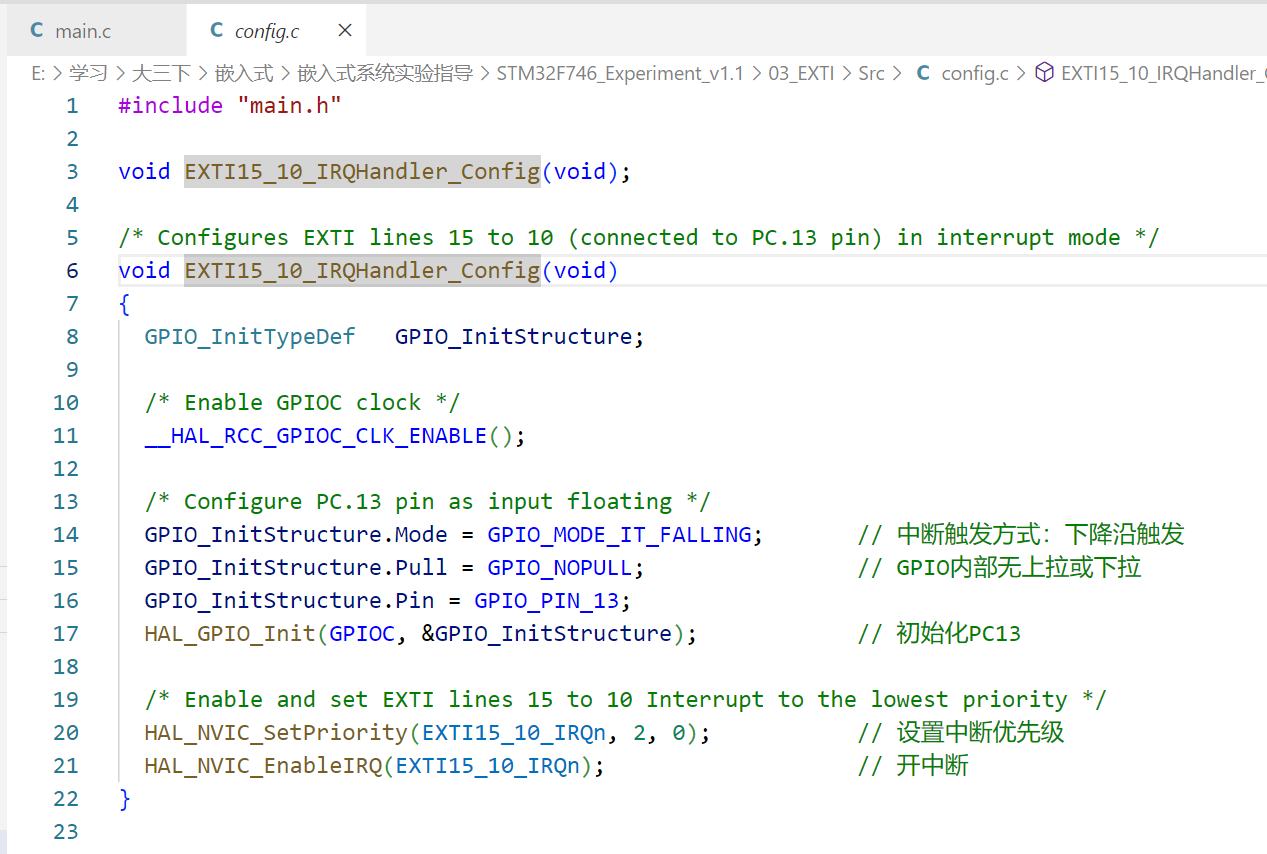
继续跳转





用于在嵌入式系统中启用 GPIOA 的时钟。该宏定义了一段代码块，用于设置与验证 RCC（复位与时钟控制）外设的 AHB1 外设时钟使能寄存器 (AHB1ENR) 中的 GPIOA 使能位 (GPIOAEN)。

1. 配置连接到PC13引脚的外部中断



* 使能GPIOC外设时钟，以便使用PC.13引脚。
* 配置PC.13引脚为下降沿触发的中断模式，无内部上拉或下拉。
* 设置并使能EXTI15\_10中断，优先级设置为2（相对较低的优先级）

## 2. 以源码和注释的形式，讲解代码的执行流程，重点是描述下面函数内代码的执行与配置原理（讲解关键语句即可）

**System\_Init()\UART\_Config()**

|  |
| --- |
| void UART\_Config(void)  {    /\* 配置UART外设 \*/    /\* 将USART外设设置为异步模式（UART模式） \*/    /\* UART配置如下：        - 字长 = 8位        - 停止位 = 1个停止位        - 奇偶校验 = 无校验        - 波特率 = 115200波特        - 硬件流控制禁用（RTS和CTS信号） \*/      // 设置USART实例    UartHandle.Instance        = USARTx;    // 设置波特率为115200    UartHandle.Init.BaudRate   = 115200;    // 设置字长为8位    UartHandle.Init.WordLength = UART\_WORDLENGTH\_8B;    // 设置停止位为1个停止位    UartHandle.Init.StopBits   = UART\_STOPBITS\_1;    // 设置无奇偶校验    UartHandle.Init.Parity     = UART\_PARITY\_NONE;    // 禁用硬件流控制    UartHandle.Init.HwFlowCtl  = UART\_HWCONTROL\_NONE;    // 设置UART模式为发送和接收    UartHandle.Init.Mode       = UART\_MODE\_TX\_RX;    // 初始化UART    HAL\_UART\_Init(&UartHandle);  } |

具体配置包括波特率、字长、停止位、奇偶校验、硬件流控制和模式等参数，然后将参数实例传入初始化函数进行初始化

**System\_Init()\UART\_Config()\HAL\_UART\_Init()**

|  |
| --- |
| HAL\_StatusTypeDef HAL\_UART\_Init(UART\_HandleTypeDef \*huart)  {    /\* 检查 UART 句柄是否分配 \*/    if(huart == NULL)    {      return HAL\_ERROR;    }      if(huart->Init.HwFlowCtl != UART\_HWCONTROL\_NONE)    {      /\* 检查硬件流控制参数 \*/      assert\_param(IS\_UART\_HWFLOW\_INSTANCE(huart->Instance));    }    else    {      /\* 检查 UART 实例参数 \*/      assert\_param(IS\_UART\_INSTANCE(huart->Instance));    }      if(huart->State == HAL\_UART\_STATE\_RESET)    {      /\* 分配锁资源并初始化 \*/      huart->Lock = HAL\_UNLOCKED;      /\* 初始化底层硬件：GPIO，时钟 \*/      HAL\_UART\_MspInit(huart);    }    /\* 设置 UART 状态为忙 \*/    huart->State = HAL\_UART\_STATE\_BUSY;    /\* 禁用外设 \*/    \_\_HAL\_UART\_DISABLE(huart);      /\* 设置 UART 通信参数 \*/    if (UART\_SetConfig(huart) == HAL\_ERROR)    {      return HAL\_ERROR;    }    /\* 如果需要高级特性初始化，则配置 \*/    if (huart->AdvancedInit.AdvFeatureInit != UART\_ADVFEATURE\_NO\_INIT)    {      UART\_AdvFeatureConfig(huart);    }    /\* 在异步模式下，必须清除以下位：       - USART\_CR2 寄存器中的 LINEN 和 CLKEN 位，       - USART\_CR3 寄存器中的 SCEN、HDSEL 和 IREN 位 \*/    huart->Instance->CR2 &= ~(USART\_CR2\_LINEN | USART\_CR2\_CLKEN);    huart->Instance->CR3 &= ~(USART\_CR3\_SCEN | USART\_CR3\_HDSEL | USART\_CR3\_IREN);    /\* 启用外设 \*/    \_\_HAL\_UART\_ENABLE(huart);    /\* 在将 huart->State 设置为就绪之前检查 TEACK 和/或 REACK \*/    return (UART\_CheckIdleState(huart));  } |

**BSP\_LED\_Init()**

|  |
| --- |
| void BSP\_LED\_Init(Led\_TypeDef Led)  {    GPIO\_InitTypeDef  gpio\_init\_structure;    GPIO\_TypeDef\*     gpio\_led;      // 根据传入的LED类型初始化相应的GPIO端口      switch(Led)      {          case LED1:              /\* 使能LED1的GPIO时钟 \*/        LED1\_GPIO\_CLK\_ENABLE();              gpio\_led = LED1\_GPIO\_PORT;              break;          case LED2:              /\* 使能LED2的GPIO时钟 \*/        LED2\_GPIO\_CLK\_ENABLE();              gpio\_led = LED2\_GPIO\_PORT;              break;          case LED3:              /\* 使能LED3的GPIO时钟 \*/        LED3\_GPIO\_CLK\_ENABLE();              gpio\_led = LED3\_GPIO\_PORT;              break;          case LED4:              /\* 使能LED4的GPIO时钟 \*/        LED4\_GPIO\_CLK\_ENABLE();              gpio\_led = LED4\_GPIO\_PORT;              break;          default:              break;      }    // 配置GPIO端口      gpio\_init\_structure.Pin = GPIO\_PIN[Led];           // 设置引脚号      gpio\_init\_structure.Mode = GPIO\_MODE\_OUTPUT\_PP;   // 设置引脚为推挽输出模式      gpio\_init\_structure.Pull = GPIO\_PULLUP;           // 设置引脚上拉      gpio\_init\_structure.Speed = GPIO\_SPEED\_HIGH;      // 设置引脚速度为高    // 初始化GPIO      HAL\_GPIO\_Init(gpio\_led, &gpio\_init\_structure);        // 默认关闭LED      HAL\_GPIO\_WritePin(gpio\_led, GPIO\_PIN[Led], GPIO\_PIN\_SET); |

**BSP\_LED\_Init()\HAL\_GPIO\_Init()**

|  |
| --- |
| /\*   \* 函数名：HAL\_GPIO\_Init   \* 参数：   \*   - GPIOx：GPIO端口指针，表示要初始化的GPIO端口   \*   - GPIO\_Init：GPIO初始化结构体指针，包含了GPIO初始化的各项参数   \* 返回值：无   \* 功能：初始化指定GPIO端口的各项配置   \*/  void HAL\_GPIO\_Init(GPIO\_TypeDef  \*GPIOx, GPIO\_InitTypeDef \*GPIO\_Init)  {    uint32\_t position = 0x00;    uint32\_t ioposition = 0x00;    uint32\_t iocurrent = 0x00;    uint32\_t temp = 0x00;    /\* 检查参数的有效性 \*/    assert\_param(IS\_GPIO\_ALL\_INSTANCE(GPIOx));    assert\_param(IS\_GPIO\_PIN(GPIO\_Init->Pin));    assert\_param(IS\_GPIO\_MODE(GPIO\_Init->Mode));    assert\_param(IS\_GPIO\_PULL(GPIO\_Init->Pull));    /\* 配置端口引脚 \*/    for(position = 0; position < GPIO\_NUMBER; position++)    {      /\* 获取引脚位置 \*/      ioposition = ((uint32\_t)0x01) << position;      /\* 获取当前引脚位置的值 \*/      iocurrent = (uint32\_t)(GPIO\_Init->Pin) & ioposition;      if(iocurrent == ioposition)      {        /\*--------------------- 配置GPIO模式 ------------------------\*/        /\* 如果选择了复用功能模式 \*/        if((GPIO\_Init->Mode == GPIO\_MODE\_AF\_PP) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_OD))        {          /\* 检查复用功能参数的有效性 \*/          assert\_param(IS\_GPIO\_AF(GPIO\_Init->Alternate));            /\* 配置当前IO的复用功能 \*/          temp = GPIOx->AFR[position >> 3];          temp &= ~((uint32\_t)0xF << ((uint32\_t)(position & (uint32\_t)0x07) \* 4)) ;          temp |= ((uint32\_t)(GPIO\_Init->Alternate) << (((uint32\_t)position & (uint32\_t)0x07) \* 4));          GPIOx->AFR[position >> 3] = temp;        }        /\* 配置IO方向模式（输入、输出、复用或模拟） \*/        temp = GPIOx->MODER;        temp &= ~(GPIO\_MODER\_MODER0 << (position \* 2));        temp |= ((GPIO\_Init->Mode & GPIO\_MODE) << (position \* 2));        GPIOx->MODER = temp;        /\* 如果选择了输出或复用功能模式 \*/        if((GPIO\_Init->Mode == GPIO\_MODE\_OUTPUT\_PP) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_PP) ||           (GPIO\_Init->Mode == GPIO\_MODE\_OUTPUT\_OD) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_OD))        {          /\* 检查速度参数的有效性 \*/          assert\_param(IS\_GPIO\_SPEED(GPIO\_Init->Speed));          /\* 配置IO速度 \*/          temp = GPIOx->OSPEEDR;          temp &= ~(GPIO\_OSPEEDER\_OSPEEDR0 << (position \* 2));          temp |= (GPIO\_Init->Speed << (position \* 2));          GPIOx->OSPEEDR = temp;          /\* 配置IO输出类型 \*/          temp = GPIOx->OTYPER;          temp &= ~(GPIO\_OTYPER\_OT\_0 << position) ;          temp |= (((GPIO\_Init->Mode & GPIO\_OUTPUT\_TYPE) >> 4) << position);          GPIOx->OTYPER = temp;        }        /\* 激活当前IO的上拉或下拉电阻 \*/        temp = GPIOx->PUPDR;        temp &= ~(GPIO\_PUPDR\_PUPDR0 << (position \* 2));        temp |= ((GPIO\_Init->Pull) << (position \* 2));        GPIOx->PUPDR = temp;        /\*--------------------- 配置外部中断模式 ------------------------\*/        /\* 配置当前IO的外部中断或事件 \*/        if((GPIO\_Init->Mode & EXTI\_MODE) == EXTI\_MODE)        {          /\* 使能SYSCFG时钟 \*/          \_\_HAL\_RCC\_SYSCFG\_CLK\_ENABLE();          temp = SYSCFG->EXTICR[position >> 2];          temp &= ~(((uint32\_t)0x0F) << (4 \* (position & 0x03)));          temp |= ((uint32\_t)(GPIO\_GET\_INDEX(GPIOx)) << (4 \* (position & 0x03)));          SYSCFG->EXTICR[position >> 2] = temp;          /\* 清除外部中断线配置 \*/          temp = EXTI->IMR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & GPIO\_MODE\_IT) == GPIO\_MODE\_IT)          {            temp |= iocurrent;          }          EXTI->IMR = temp;          temp = EXTI->EMR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & GPIO\_MODE\_EVT) == GPIO\_MODE\_EVT)          {            temp |= iocurrent;          }          EXTI->EMR = temp;          /\* 清除上升沿下降沿触发配置 \*/          temp = EXTI->RTSR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & RISING\_EDGE) == RISING\_EDGE)          {            temp |= iocurrent;          }          EXTI->RTSR = temp;          temp = EXTI->FTSR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & FALLING\_EDGE) == FALLING\_EDGE)          {            temp |= iocurrent;          }          EXTI->FTSR = temp;        }      }    }  } |

**BSP\_LED\_Init()\HAL\_GPIO\_WritePin()**

|  |
| --- |
| void HAL\_GPIO\_WritePin(GPIO\_TypeDef\* GPIOx, uint16\_t GPIO\_Pin, GPIO\_PinState PinState)  {    /\* 检查参数是否合法 \*/    assert\_param(IS\_GPIO\_PIN(GPIO\_Pin));       // 确认GPIO\_Pin是有效的GPIO引脚    assert\_param(IS\_GPIO\_PIN\_ACTION(PinState)); // 确认PinState是有效的引脚状态（高电平或低电平）    if(PinState != GPIO\_PIN\_RESET) // 如果PinState不是低电平（即设置为高电平）    {      GPIOx->BSRR = GPIO\_Pin; // 设置对应的GPIO引脚为高电平    }    else // 如果PinState是低电平    {      GPIOx->BSRR = (uint32\_t)GPIO\_Pin << 16; // 设置对应的GPIO引脚为低电平    }  } |

**EXTI15\_10\_IRQHandler\_Config()**

|  |
| --- |
| void EXTI15\_10\_IRQHandler\_Config(void)  {    GPIO\_InitTypeDef   GPIO\_InitStructure;    /\* 启用GPIOC时钟 \*/    \_\_HAL\_RCC\_GPIOC\_CLK\_ENABLE();      /\* 配置PC13引脚为输入浮空模式 \*/    GPIO\_InitStructure.Mode = GPIO\_MODE\_IT\_FALLING;        // 中断触发方式：下降沿触发    GPIO\_InitStructure.Pull = GPIO\_NOPULL;                 // GPIO内部无上拉或下拉    GPIO\_InitStructure.Pin = GPIO\_PIN\_13;    HAL\_GPIO\_Init(GPIOC, &GPIO\_InitStructure);             // 初始化PC13引脚    /\* 启用并设置EXTI15到10线中断，优先级最低 \*/    HAL\_NVIC\_SetPriority(EXTI15\_10\_IRQn, 2, 0);            // 设置中断优先级    HAL\_NVIC\_EnableIRQ(EXTI15\_10\_IRQn);                    // 开启中断  } |

**EXTI15\_10\_IRQHandler\_Config()\HAL\_GPIO\_Init()**

|  |
| --- |
| /\*   \* 函数名：HAL\_GPIO\_Init   \* 参数：   \*   - GPIOx：GPIO端口指针，表示要初始化的GPIO端口   \*   - GPIO\_Init：GPIO初始化结构体指针，包含了GPIO初始化的各项参数   \* 返回值：无   \* 功能：初始化指定GPIO端口的各项配置   \*/  void HAL\_GPIO\_Init(GPIO\_TypeDef  \*GPIOx, GPIO\_InitTypeDef \*GPIO\_Init)  {    uint32\_t position = 0x00;    uint32\_t ioposition = 0x00;    uint32\_t iocurrent = 0x00;    uint32\_t temp = 0x00;    /\* 检查参数的有效性 \*/    assert\_param(IS\_GPIO\_ALL\_INSTANCE(GPIOx));    assert\_param(IS\_GPIO\_PIN(GPIO\_Init->Pin));    assert\_param(IS\_GPIO\_MODE(GPIO\_Init->Mode));    assert\_param(IS\_GPIO\_PULL(GPIO\_Init->Pull));    /\* 配置端口引脚 \*/    for(position = 0; position < GPIO\_NUMBER; position++)    {      /\* 获取引脚位置 \*/      ioposition = ((uint32\_t)0x01) << position;      /\* 获取当前引脚位置的值 \*/      iocurrent = (uint32\_t)(GPIO\_Init->Pin) & ioposition;      if(iocurrent == ioposition)      {        /\*--------------------- 配置GPIO模式 ------------------------\*/        /\* 如果选择了复用功能模式 \*/        if((GPIO\_Init->Mode == GPIO\_MODE\_AF\_PP) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_OD))        {          /\* 检查复用功能参数的有效性 \*/          assert\_param(IS\_GPIO\_AF(GPIO\_Init->Alternate));            /\* 配置当前IO的复用功能 \*/          temp = GPIOx->AFR[position >> 3];          temp &= ~((uint32\_t)0xF << ((uint32\_t)(position & (uint32\_t)0x07) \* 4)) ;          temp |= ((uint32\_t)(GPIO\_Init->Alternate) << (((uint32\_t)position & (uint32\_t)0x07) \* 4));          GPIOx->AFR[position >> 3] = temp;        }        /\* 配置IO方向模式（输入、输出、复用或模拟） \*/        temp = GPIOx->MODER;        temp &= ~(GPIO\_MODER\_MODER0 << (position \* 2));        temp |= ((GPIO\_Init->Mode & GPIO\_MODE) << (position \* 2));        GPIOx->MODER = temp;        /\* 如果选择了输出或复用功能模式 \*/        if((GPIO\_Init->Mode == GPIO\_MODE\_OUTPUT\_PP) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_PP) ||           (GPIO\_Init->Mode == GPIO\_MODE\_OUTPUT\_OD) || (GPIO\_Init->Mode == GPIO\_MODE\_AF\_OD))        {          /\* 检查速度参数的有效性 \*/          assert\_param(IS\_GPIO\_SPEED(GPIO\_Init->Speed));          /\* 配置IO速度 \*/          temp = GPIOx->OSPEEDR;          temp &= ~(GPIO\_OSPEEDER\_OSPEEDR0 << (position \* 2));          temp |= (GPIO\_Init->Speed << (position \* 2));          GPIOx->OSPEEDR = temp;          /\* 配置IO输出类型 \*/          temp = GPIOx->OTYPER;          temp &= ~(GPIO\_OTYPER\_OT\_0 << position) ;          temp |= (((GPIO\_Init->Mode & GPIO\_OUTPUT\_TYPE) >> 4) << position);          GPIOx->OTYPER = temp;        }        /\* 激活当前IO的上拉或下拉电阻 \*/        temp = GPIOx->PUPDR;        temp &= ~(GPIO\_PUPDR\_PUPDR0 << (position \* 2));        temp |= ((GPIO\_Init->Pull) << (position \* 2));        GPIOx->PUPDR = temp;        /\*--------------------- 配置外部中断模式 ------------------------\*/        /\* 配置当前IO的外部中断或事件 \*/        if((GPIO\_Init->Mode & EXTI\_MODE) == EXTI\_MODE)        {          /\* 使能SYSCFG时钟 \*/          \_\_HAL\_RCC\_SYSCFG\_CLK\_ENABLE();          temp = SYSCFG->EXTICR[position >> 2];          temp &= ~(((uint32\_t)0x0F) << (4 \* (position & 0x03)));          temp |= ((uint32\_t)(GPIO\_GET\_INDEX(GPIOx)) << (4 \* (position & 0x03)));          SYSCFG->EXTICR[position >> 2] = temp;          /\* 清除外部中断线配置 \*/          temp = EXTI->IMR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & GPIO\_MODE\_IT) == GPIO\_MODE\_IT)          {            temp |= iocurrent;          }          EXTI->IMR = temp;          temp = EXTI->EMR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & GPIO\_MODE\_EVT) == GPIO\_MODE\_EVT)          {            temp |= iocurrent;          }          EXTI->EMR = temp;          /\* 清除上升沿下降沿触发配置 \*/          temp = EXTI->RTSR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & RISING\_EDGE) == RISING\_EDGE)          {            temp |= iocurrent;          }          EXTI->RTSR = temp;          temp = EXTI->FTSR;          temp &= ~((uint32\_t)iocurrent);          if((GPIO\_Init->Mode & FALLING\_EDGE) == FALLING\_EDGE)          {            temp |= iocurrent;          }          EXTI->FTSR = temp;        }      }    }  } |

**HAL\_GPIO\_EXTI\_Callback()**

|  |
| --- |
| void HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin)            // 定义HAL\_GPIO\_EXTI\_Callback回调函数，当GPIO引脚发生外部中断时被调用  {      if (GPIO\_Pin == GPIO\_PIN\_13)                         // 检查触发中断的GPIO引脚是否为13号引脚      {          /\* Toggle LED1 \*/          BSP\_LED\_Toggle(LED1);                            // 切换LED1的状态（如果是亮的则熄灭，反之亦然）          printf("\n\rLED1 switched\n\r");                 // 向串口打印消息，表示LED1的状态已切换      }  } |

**HAL\_GPIO\_EXTI\_Callback()\BSP\_LED\_Toggle()**

|  |
| --- |
| void BSP\_LED\_Toggle(Led\_TypeDef Led)  {      GPIO\_TypeDef\* gpio\_led;  // 定义一个指向GPIO端口的指针变量      // 实际上这里只有一个LED被处理      switch(Led)      {          case LED1:  // 如果传入的参数是LED1              gpio\_led = LED1\_GPIO\_PORT;  // 将指针指向LED1的GPIO端口              break;          case LED2:  // 如果传入的参数是LED2              gpio\_led = LED2\_GPIO\_PORT;  // 将指针指向LED2的GPIO端口              break;          case LED3:  // 如果传入的参数是LED3              gpio\_led = LED3\_GPIO\_PORT;  // 将指针指向LED3的GPIO端口              break;          case LED4:  // 如果传入的参数是LED4              gpio\_led = LED4\_GPIO\_PORT;  // 将指针指向LED4的GPIO端口              break;          default:  // 如果传入的参数不是以上几种情况              break;  // 什么也不做      }      HAL\_GPIO\_TogglePin(gpio\_led, GPIO\_PIN[Led]);  // 切换对应GPIO端口的引脚状态（高电平变低电平，低电平变高电平）  } |

切换指定LED的状态

## 3. 解释原理，上述代码中，

**1）在哪个地方设置的GPIO口的中断和触发模式**

EXTI15\_10\_IRQHandler\_Config()函数：

|  |
| --- |
| /\* Configure PC.13 pin as input floating \*/    GPIO\_InitStructure.Mode = GPIO\_MODE\_IT\_FALLING;           // 设置PC13引脚为下降沿触发中断 |

**2）如何绑定到中断函数HAL\_GPIO\_EXTI\_Callback**

通过中断处理程序（IRQ Handler）和相应的中断处理函数（HAL\_GPIO\_EXTI\_IRQHandler）间接绑定的

1. 配置GPIO引脚为中断模式：这包括设置GPIO引脚的模式为中断模式（上升沿、下降沿或双边沿触发），并初始化GPIO引脚。
2. 配置中断优先级和使能中断：通过设置NVIC的中断优先级并使能中断。
3. 实现中断处理程序（IRQ Handler）：在IRQ Handler中调用 HAL\_GPIO\_EXTI\_IRQHandler 函数。
4. 实现中断回调函数 HAL\_GPIO\_EXTI\_Callback：在回调函数中处理具体的中断事件。