HAL库底层实现

第一、STM32Cube 固件包分析

1.Drivers 文件夹

a.BSP 文件夹：称为板级支持包，用于适配 ST 官方对应的开发板的硬件驱动程 序，每一种开发板对应一个文件夹。例如触摸屏，LCD，SRAM 以及 EEPROM 等板载硬件资源等驱动。

b.CMSIS 文件夹：CMSIS 标准的软件抽象层组件相关文件，主要包括 DSP 库(DSP_LIB 文件夹)，Cortex-M 内核 及其设备文件（Include 文件夹）

c.STM32F1xx_H AL_Driver文件夹：它包含了所有的 STM32F1xx 系列HAL 库头文件和源件，也就是所有底层硬件抽象层 API 声明和定义（**它的作用是屏蔽了复杂的硬件寄存器操作**，统一了外设的接口函 数。**该文件夹包含 Src 和 Inc 两个子文件夹，其中 Src 子文件夹存放的是.c 源文件，Inc 子文件夹存放的是.h 头文件**。）

2.CMSIS 文件夹中的 Device 和 Include

a.stm32f1xx.h：选择性包含某一特定的 STM32F1 系列芯片的头文件。

b.stm32f103xb.h：这个文件的主要作用是定义声明寄存器以及封装内存操 作，以结构体和宏定义标识符的形式（包含所有**外设寄存器的定义，内存映射**）

c.startup_STM32F103xx.s：系列芯片的启动文件，主要是进行堆栈的初始化，中 断向量表以及中断函数定义等。启动文件有一个很重要的作 用就是系统复位后引导进入 main 函数。

d.system_stm32f1xx.c /system_stm32f1xx.h：主要是声明和定义了系统初始化函数 SystemInit（它并没有设置具体的时钟值。**在使用标准库的时候，SystemInit 函数会帮我们配置 好系统时钟配置相关的各个寄存器**） 以及系统时钟更新函数 SystemCoreClockUpdate

3.HAL 库文件介绍

a.sm32f1xx_hal.c/stm32f1xx_hal.h：主要实现HAL库的初始化、系统滴答，HAL库延时函数

b.stm32f1xx_hal_conf.h：stm32f1xx_hal.h 引用了这 个文件，用来对 HAL 库进行裁剪。由于 Hal 库的 很多配置都是通过预编译的条件宏来决定是否使 用这一 HAL 库的功能

c.stm32hxx_hal_def.h：HAL常用定义、枚举、宏和结构定义。返回值类 HAL_StatusTypeDef 就是在这个文件中定义的

d.stm32f1xx_hal_cortex.h/stm32f1xx_hal_cortex.c：它是一些 Cortex 内核通用函数声明和定义，例如中断优先级 NVIC 配置，MPU，系统软复位以及 Systick 配置等

e.stm32f1xx_hal_ppp.c/stm32f1xx_hal_ppp.h:外设驱动函数。对于所有 STM32 驱动名称都相 同，ppp 代表一类外设，包含该外设的操作 API 函 数

💡 HAL 库驱动部分与外设句柄相关的宏（这些函数功能只需要操控，一位寄存器），可以考虑用宏定义函数，来替代位操作，下面是相关函数类型：

宏定义结构	用途
__HAL_PPP_ENABLE_IT(__HANDLE__, __INTERRUPT__)	使能外设中断
__HAL_PPP_DISABLE_IT(__HANDLE__, __INTERRUPT__)	禁用外设中断
__HAL_PPP_GET_IT (__HANDLE__, __ INTERRUPT __)	获取外设某一中断源
__HAL_PPP_CLEAR_IT (__HANDLE__, __ INTERRUPT __)	清除外设中断
__HAL_PPP_GET_FLAG (__HANDLE__, __FLAG__)	获取外设的状态标记
__HAL_PPP_CLEAR_FLAG (__HANDLE__, __FLAG__)	清除外设的状态标记
__HAL_PPP_ENABLE(__HANDLE__)	使能某一外设
__HAL_PPP_DISABLE(__HANDLE__)	禁用某一外设
__HAL_PPP_XXXX (__HANDLE__, __PARAM__)	针对外设的特殊操作
__HAL_PPP_GET_ IT_SOURCE (__HANDLE__, __INTERRUPT __)	检查外设的中断源

第二、HAL 库基本配置

1.stm32f1xx_hal_conf.h用户配置文件

内容：用于裁剪 HAL 库和定义一些变量，释放一些无用定义。

调用关系：main→stm32f1xx_hal.h→stm32f1xx_hal_conf.h（清除无用的头文件定义）

注1：所有驱动外设文件都进行了条件编译，只有定义了对应的宏，才进行编译

注2：必须定义#define HAL_MODULE_ENABLED，在stm32f1xx_hal.c中所用内容进行条件编译中，定了该宏才会有内容（HAL_Init()、HAL_Delay()、HAL_GetTick()）进行编译。

注3：#define TICK_INT_PRIORITY　　((uint32_t)0x0F) TICK_INT_PRIORITY 是滴答定时器的优先级，那么假如该中断的优先级高于滴答定时器的优先级，就会导致滴答定时 器中断服务函数一直得不到运行，程序便会卡死在这里。所以滴答定时器的中断优先级一定要 比这些中断高。

2.stm32f1xx_hal.c 文件

内容：HAL 库的初始化、系统滴答、基准电压配置、IO 补偿、低功耗、EXTI 配置等都集合在这个文件里面

HAL_Init()

使能 Flash 的预取缓冲器（内核内容）

```
__HAL_FLASH_PREFETCH_BUFFER_ENABLE();
```

中断优先级分组（区别设置优先级）

```
 HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4);
```

配置systick定时器作位基础时钟源（1ms 产生一个中断，默认为高速时钟源 HSI）

```
//抢占优先级的级别高于响应优先级。而数值越小所代表的优先级就越高
//TICK_INT_PRIORITY抢占优先级,宏定义的是15，抢占优先最低

//如果其他中断服务函数调用了 HAL_Delay()，必须小心，滴答定时 器中断必须具有比调用了HAL_Delay()函数的其他中断服务函数的优先级高(数值较低)，否则 会导致滴答定时器中断服务函数一直得不到执行，从而卡死在这里


HAL_InitTick(TICK_INT_PRIORITY);

/*
__weak HAL_StatusTypeDef HAL_InitTick(uint32_t TickPriority)
{
  if (HAL_SYSTICK_Config(SystemCoreClock / (1000U / uwTickFreq)) > 0U)
//将systick设置成1ms基准
  {
  return HAL_ERROR;
  }
  // __NVIC_PRIO_BITS  4(1向左移动4位，括号里为16，即抢占优先级最大值不能超过16)
  if (TickPriority < (1UL << __NVIC_PRIO_BITS ))
  {
    HAL_NVIC_SetPriority(SysTick_IRQn, TickPriority, 0U);
    uwTickPrio = TickPriority;
  }
  else
  {
    return HAL_ERROR;
  return HAL_OK;
}
```

```
27 */
```

调用初始化回调函数（使能AFIO和PWR）

```
1 void HAL_MspInit(void)
2 {
3   __HAL_RCC_AFIO_CLK_ENABLE();
4   __HAL_RCC_PWR_CLK_ENABLE();
5 }
```

HAL_IncTick()

该函数在滴答定时器时钟中断服务函数中被调用，一般滴答定时器 1ms 中断一次

```
1 //定义了一个32位全局变量，1ms中断进行计数+1，通过获取计数值进行定时
2
3 __weak void HAL_IncTick(void)
4 {
5   uwTick += uwTickFreq;
6 }
7
8 //中断函数
9 void SysTick_Handler(void)
10 {
11   HAL_IncTick();
12 }
```

HAL_GetTick()/HAL_GetTickFreq()

获取全局变量 uwTick当前计数值

```
1 __weak uint32_t HAL_GetTick(void)
2 {
3   return uwTick;
4 }
5
6 extern HAL_TickFreqTypeDef uwTickFreq;
7 HAL_TickFreqTypeDef uwTickFreq = HAL_TICK_FREQ_DEFAULT;
8
9 typedef enum
10 {
11   HAL_TICK_FREQ_10HZ         = 100U,
12   HAL_TICK_FREQ_100HZ        = 10U,
13   HAL_TICK_FREQ_1KHZ         = 1U,
14   HAL_TICK_FREQ_DEFAULT      = HAL_TICK_FREQ_1KHZ
```

```
} HAL_TickFreqTypeDef;
```

获取滴答定时器中断频率

```
 //定义枚举变量，包含所需中断频率，标识是被定义了数值，并非默认
typedef enum
{
  HAL_TICK_FREQ_10HZ         = 100U,
  HAL_TICK_FREQ_100HZ        = 10U,
  HAL_TICK_FREQ_1KHZ         = 1U,
  HAL_TICK_FREQ_DEFAULT      = HAL_TICK_FREQ_1KHZ
} HAL_TickFreqTypeDef;

//定义中断频率
extern HAL_TickFreqTypeDef uwTickFreq;
HAL_TickFreqTypeDef uwTickFreq = HAL_TICK_FREQ_DEFAULT;

//HAL_SYSTICK_Config(SystemCoreClock / (1000U / uwTickFreq))   //调用uwTickFreq参数设置内核寄存器

//
HAL_TickFreqTypeDef HAL_GetTickFreq(void)
{
  return uwTickFreq;
}
```

HAL_SetTickFreq()

设置滴答定时器中断频率

```
 //uwTickFreq是全局变量，先备份原先的中断频率，以防新的中断频率配置不成功，可以启用原先的中断频率

 //uwTickPrio为15

HAL_StatusTypeDef HAL_SetTickFreq(HAL_TickFreqTypeDef Freq)
{
  HAL_StatusTypeDef status  = HAL_OK;
  HAL_TickFreqTypeDef prevTickFreq;

  assert_param(IS_TICKFREQ(Freq));

  if (uwTickFreq != Freq)
  {
    /*备份滴答定时器中断频率 */
    prevTickFreq = uwTickFreq;

    /* 更新被 HAL_InitTick()调用的全局变量 uwTickFreq */
```

```
    uwTickFreq = Freq;

    /*应用新的滴答定时器中断频率  */
    status = HAL_InitTick(uwTickPrio);

    if (status != HAL_OK)
    {
      /* 恢复之前的滴答定时器中断频率 */
      uwTickFreq = prevTickFreq;
    }
  }
  return status;
}
```

HAL_Delay()

HAL 库的延时函数

```
//通过获取滴答定时器中断计数器值，作比较来进行延时

_weak void HAL_Delay(uint32_t Delay)
{
  uint32_t tickstart = HAL_GetTick();
  uint32_t wait = Delay;

  /* 添加频率以保证最小的等待时间 */
  if (wait < HAL_MAX_DELAY)
  {
    wait += (uint32_t)(uwTickFreq);
  }

  while ((HAL_GetTick() - tickstart) < wait)
  {
  }
}
```

3.功能函数

SET_BIT/CLEAR_BITREAD_BIT等

直接操作寄存器，宏定义函数

```
//REG、BIT都是32位寄存器，其中BIT只有一位是1

//设置按位“或”操作，清除按位“与”操作
```

```

#define SET_BIT(REG, BIT)     ((REG) |= (BIT))

#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))

#define READ_BIT(REG, BIT)    ((REG) & (BIT))

#define CLEAR_REG(REG)        ((REG) = (0x0))

#define WRITE_REG(REG, VAL)   ((REG) = (VAL))

#define READ_REG(REG)         ((REG))



重点说明一下：配置对应寄存器位
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))

stm32中定义 “1”为有效，“0”为无效（这种位相当于开关---使能或失能独立的位，“1”为有效），所以通常用“|” 和 “&”直接解决

对于配置多位，也就是“1”和“0”都为有效值，上面的方法就不适用了
采用直接赋值的方式的要求：
---首先其他位的值不变的情况下，对对应位配置
--“|”----------寄存器 | 0时，对应的位不变，寄存器0的位 | 对应的值 = 对应的值
--将|上的"值"保证不操作的位，其“值”为0，操作的的位，其“位”为0，其“值”要输入的值

CLEARMASK------需操作的位,写“1”。取反，保证不需操作的位为“0”

(READ_REG(REG)) & (~(CLEARMASK)))----将寄存器操作位，写0



```

第三、STM32时钟配置

1.晶振源的选择

STM32F103c8t6的晶振源（在用户配置文件中）

```

//外部高速时钟源，选择的是8Mhz

#if !defined  (HSE_VALUE)
  #define HSE_VALUE    8000000U
```

```
#endif /* HSE_VALUE */

//内部高速时钟源，选择的是8Mhz
#if !defined  (HSI_VALUE)
  #define HSI_VALUE    8000000U
#endif /* HSI_VALUE */

//内部低速时钟源，选择的是40khz
#if !defined  (LSI_VALUE)
 #define LSI_VALUE      40000U
#endif /* LSI_VALUE */


//外部高速时钟源，选择的是32.768khz
#if !defined  (LSE_VALUE)
  #define LSE_VALUE    32768U
#endif /* LSE_VALUE */
```

2.RCC时钟线路配置

stm32cubemx配置时钟树函数SystemClock_Config()

数据结构定义

晶振源定义

```

//激活(使能晶振)，设置分频器、倍频器、选择器
typedef struct
{
  uint32_t OscillatorType;       / * 需要选择配置的振荡器类型，HSE、HSI、LSE、LSI*/
  uint32_t HSEState;             /* HSE 状态 ,激活、停用等*/
  uint32_t HSEPredivValue;       /* HSE 预分频值(HSE第一个分频器，通常不分频)*/

  uint32_t LSEState;             /* LSE 状态 */
  uint32_t HSIState;             /* HIS 状态 */
  uint32_t HSICalibrationValue; /* HIS 校准值 */
  uint32_t LSIState;             /* LSI 状态 */
  RCC_PLLInitTypeDef PLL;        /* PLL 配置 */
}RCC_OscInitTypeDef;
```

时钟倍频、时钟选择

```
typedef struct
{
```

```
  uint32_t PLLState;      /* PLL 状态 */---------使能
  uint32_t PLLSource;     /* PLL 时钟源 */-------选择器

  uint32_t PLLMUL;        /* PLL 倍频系数 M */---倍频器
} RCC_PLLInitTypeDef;
```

初始化CPU、AHB和APB总线时钟

```
typedef struct
{
  uint32_t ClockType; /* 要配置的时钟 */
  uint32_t SYSCLKSource; /* 系统时钟源 */
  uint32_t AHBCLKDivider; /* AHB 分频系数 */
  uint32_t APB1CLKDivider; /* APB1 分频系数 */
  uint32_t APB2CLKDivider; /* APB2 分频系数 */
}RCC_ClkInitTypeDef;
```

RCC配置函数

SystemClock_Config()系统RCC配置

```
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};


  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON;
  RCC_OscInitStruct.HSEPredivValue = RCC_HSE_PREDIV_DIV1;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL9;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /****************** 具体配置*************************/
  /* 选中 PLL 作为系统时钟源并且配置 HCLK,PCLK1 和 PCLK2 */
  /* 设置系统时钟时钟源为 PLL */
  /* AHB 分频系数为 1 */
  /* APB1 分频系数为 2 */
  * APB2 分频系数为 1 */
  /* 同时设置 FLASH 延时周期为 2WS，也就是 3 个 CPU 周期。*/

```

```
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) !=
HAL_OK)
  {
    Error_Handler();
  }
}
```

HAL_RCC_OscConfig()函数的主要内容就是配置和激活微控制器的振荡器

```

//HAL_RCC_OscConfig()选择配置的振荡器，进行激活·····以配置HSE为例

1.检查配置结构体是否为空
2.如果系统时钟来源于HSE，这意味着HSE必须保持开启,确保在系统依赖HSE或者PLL使用HSE作为输入源的情况下，不能关闭HSE，以防止系统时钟丢失，否则会返回错误(代码中，先if特殊段(会影响主体的情况)，再进行主体)
3.激活HSE振荡器
4.判断是否激活成功，并选择超时时间

学习激活激活HSE振荡器，操作底层寄存器
激活HSE振荡器寄存器---RCC_CR(第16位--HSEON: 外部高速时钟使能)
RCC_CR|RCC_CR_HSEON操作  RCC_CR_HSEON = 0x00010000


#define __HAL_RCC_HSE_CONFIG(__STATE__)
    \
                    do{
    \
                      if ((__STATE__) == RCC_HSE_ON)   //激活HES振荡器
    \
                      {
    \
                        SET_BIT(RCC->CR,|RCC_CR_HSEON);
    \
                      }
    \
                      else if ((__STATE__) == RCC_HSE_OFF)
     \
                      {
    \
```

```
22                        CLEAR_BIT(RCC->CR, RCC_CR_HSEON);
      \
23                        CLEAR_BIT(RCC->CR, RCC_CR_HSEBYP);
       \
24                      }
      \
25                      else if ((__STATE__) == RCC_HSE_BYPASS)
      \
26                      {
      \
27                        SET_BIT(RCC->CR, RCC_CR_HSEBYP);
       \
28                        SET_BIT(RCC->CR, RCC_CR_HSEON);
      \
29                      }
      \
30                      else
       \
31                      {
      \
32                        CLEAR_BIT(RCC->CR, RCC_CR_HSEON);
      \
33                        CLEAR_BIT(RCC->CR, RCC_CR_HSEBYP);
       \
34                      }
      \
35                    }while(0U)
36
37
38 HAL_StatusTypeDef HAL_RCC_OscConfig(RCC_OscInitTypeDef
   *RCC_OscInitStruct)
39 {
40   uint32_t tickstart;
41   uint32_t pll_config;
42
43   /* Check Null pointer */
44   if (RCC_OscInitStruct == NULL)
45   {
46     return HAL_ERROR;
47   }
48   /*------------------------------ HSE Configuration -----------------
   -------*/
49   if (((RCC_OscInitStruct->OscillatorType) & RCC_OSCILLATORTYPE_HSE) ==
   RCC_OSCILLATORTYPE_HSE)
50   {
51     /* Check the parameters */
52     assert_param(IS_RCC_HSE(RCC_OscInitStruct->HSEState));
53
```

```
54    /* When the HSE is used as system clock or clock source for PLL in
   these cases it is not allowed to be disabled */
55    if ((__HAL_RCC_GET_SYSCLK_SOURCE() == RCC_SYSCLKSOURCE_STATUS_HSE)
56        || ((__HAL_RCC_GET_SYSCLK_SOURCE() ==
   RCC_SYSCLKSOURCE_STATUS_PLLCLK) && (__HAL_RCC_GET_PLL_OSCSOURCE() ==
   RCC_PLLSOURCE_HSE)))
57    {
58      if ((__HAL_RCC_GET_FLAG(RCC_FLAG_HSERDY) != RESET) &&
   (RCC_OscInitStruct->HSEState == RCC_HSE_OFF))
59      {
60        return HAL_ERROR;
61      }
62    }
63    else
64    {
65      /* Set the new HSE configuration ----------------------------------
   -------*/
66      __HAL_RCC_HSE_CONFIG(RCC_OscInitStruct->HSEState);        //激活
   HSE振荡器
67
68
69      /* Check the HSE State */
70      if (RCC_OscInitStruct->HSEState != RCC_HSE_OFF)
71      {
72        /* Get Start Tick */
73        tickstart = HAL_GetTick();
74
75        /* Wait till HSE is ready */
76        while (__HAL_RCC_GET_FLAG(RCC_FLAG_HSERDY) == RESET)    //检查
   HSE是否准备就绪，寄存器有该位
77        {
78            if ((HAL_GetTick() - tickstart) > HSE_TIMEOUT_VALUE) //检查
   是否超时
79          {
80            return HAL_TIMEOUT;
81          }
82        }
83      }
84      else
85      {
86        /* Get Start Tick */
87        tickstart = HAL_GetTick();
88
89        /* Wait till HSE is disabled */
90        while (__HAL_RCC_GET_FLAG(RCC_FLAG_HSERDY) != RESET)
91        {
92          if ((HAL_GetTick() - tickstart) > HSE_TIMEOUT_VALUE)
93          {
```

```
            return HAL_TIMEOUT;
          }
        }
      }
    }
  }
```

HAL_RCC_ClockConfig()时钟设置

```
//HAL_RCC_ClockConfig()包含CPU、AHB、APB1、APB2的设置，主要的就是设置分频器

//主要内容：
1.用于更新Flash的延时设置，以适应新的系统时钟配置
2.分别配置HCLK、SYSCLK、PCLK1、PCLK2对应的分频器

该参数主要作用FLatency：

FLASH_LATENCY_2是STM32微控制器中Flash存储器访问的预充电延迟周期数。在STM32的微处理器中，当系统时钟频率增加时，为了保证Flash读取操作的正确性，需要增加Flash的预充电时间，也就是访问Flash前的等待周期。这是因为高速时钟下，Flash需要更多的时间准备数据。

在HAL_RCC_ClockConfig()函数中，FLASH_LATENCY_2作为第二个参数传递，用于更新Flash的延时设置，以适应新的系统时钟配置。这样可以确保系统在高速运行时仍能正确地从Flash读取指令，避免因时钟速度过快导致的读取错误。

--以HCLK为主讲解
//关于FLatency不做论述
1.判断是否需要配置AHB时钟（HCLK）
2.将APB1（PCLK1）和 APB2（PCLK2）分频器分频系数至最大值，其目的是后续要改变HCLK的分频，这样做可以确保在改变过程中APB1的时钟速度不会超出其最大规定范围
3.设置HCLK的分频器

---理解一下分频器的操作寄存器
RCC_CFGR_HPRE = 0x000000F0   AHBCLKDivider = 0x00000000U
MODIFY_REG(RCC->CFGR, RCC_CFGR_HPRE, RCC_ClkInitStruct->AHBCLKDivider);
#define MODIFY_REG(REG, CLEARMASK, SETMASK)  WRITE_REG((REG), (((READ_REG(REG)) & (~(CLEARMASK))) | (SETMASK)))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))

HAL_StatusTypeDef HAL_RCC_ClockConfig(RCC_ClkInitTypeDef *RCC_ClkInitStruct, uint32_t FLatency)
{
  uint32_t tickstart;

```

```
  if (RCC_ClkInitStruct == NULL)
  {
    return HAL_ERROR;
  }

  assert_param(IS_RCC_CLOCKTYPE(RCC_ClkInitStruct->ClockType));
  assert_param(IS_FLASH_LATENCY(FLatency));

#if defined(FLASH_ACR_LATENCY)

  if (FLatency > __HAL_FLASH_GET_LATENCY())
  {
    __HAL_FLASH_SET_LATENCY(FLatency);
    if (__HAL_FLASH_GET_LATENCY() != FLatency)
  {
    return HAL_ERROR;
  }
}

#endif /* FLASH_ACR_LATENCY */
/*-------------------------- HCLK Configuration -----------------------
--*/
if (((RCC_ClkInitStruct->ClockType) & RCC_CLOCKTYPE_HCLK) ==
RCC_CLOCKTYPE_HCLK)
  {
    if (((RCC_ClkInitStruct->ClockType) & RCC_CLOCKTYPE_PCLK1) ==
RCC_CLOCKTYPE_PCLK1)
    {
      MODIFY_REG(RCC->CFGR, RCC_CFGR_PPRE1, RCC_HCLK_DIV16);
    }

    if (((RCC_ClkInitStruct->ClockType) & RCC_CLOCKTYPE_PCLK2) ==
RCC_CLOCKTYPE_PCLK2)
    {
      MODIFY_REG(RCC->CFGR, RCC_CFGR_PPRE2, (RCC_HCLK_DIV16 << 3));
    }
    assert_param(IS_RCC_HCLK(RCC_ClkInitStruct->AHBCLKDivider));
    MODIFY_REG(RCC->CFGR, RCC_CFGR_HPRE, RCC_ClkInitStruct-
>AHBCLKDivider);
  }

   ....................................................
   ....................................................
   ....................................................

}
```

3.其他外设时钟使能

外设时钟使能

```
1
2 如：操作寄存器APB2ENR
3 __HAL_RCC_GPIOA_CLK_ENABLE()
4 __HAL_RCC_GPIOB_CLK_ENABLE()
5 __HAL_RCC_TIM1_CLK_ENABLE()
6 __HAL_RCC_SPI1_CLK_ENABLE()
7 __HAL_RCC_USART1_CLK_ENABLE()
8
9 详细--
10 RCC_APB2ENR_IOPAEN ------为0x00000004
11
12 #define __HAL_RCC_GPIOA_CLK_ENABLE()   do { \
13                                          __IO uint32_t tmpreg; \
14                                          SET_BIT(RCC->APB2ENR,
   RCC_APB2ENR_IOPAEN);\ --------------将APB2ENR第3位
15                                          /* Delay after an RCC peripheral
   clock enabling */\
16                                          tmpreg = READ_BIT(RCC->APB2ENR,
   RCC_APB2ENR_IOPAEN);\
17                                          UNUSED(tmpreg); \
18                                        } while(0U)
19
```

第四、STM32-GPIO配置

1.数据结构定义

数据类型与寄存器映射

```
1
2 //寄存器映射----GPIOx配置寄存器地址映射
3
4 //硬件的地址（包含：代码存储和寄存器），将外设基地址赋给定义的结构体
5 //这里结构体变量是没有进行定义变量的，定义变量也是把其首地址给变量
6 //不初始化结构体，是因为结构体变量其实已经开辟的空间，只需地址进行操作
7
8 #define GPIOA                 ((GPIO_TypeDef *)GPIOA_BASE)
9 #define GPIOA_BASE            (APB2PERIPH_BASE + 0x00000800UL)
10
11
```

```
//GPIO的操作寄存器
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;

typedef struct
{
  uint32_t Pin;        /* 引脚号 */           -----输出输入数据寄存器，是按0-15顺序排列，所以写/读1，对应位进行“或”|“与”操作
  uint32_t Mode;       /* 模式设置 */         -----设置GPIOx_CRL
  uint32_t Pull;       /* 上拉下拉设置 */     -----设置GPIOx_CRL
  uint32_t Speed;      /* 速度设置 */         -----设置GPIOx_CR
} GPIO_InitTypeDef;

#define GPIO_MODE_INPUT (0x00000000U)                    /* 输入模式 */
#define GPIO_MODE_OUTPUT_PP (0x00000001U)                /* 推挽输出 */
#define GPIO_MODE_OUTPUT_OD (0x00000011U)                /* 开漏输出 */
#define GPIO_MODE_AF_PP (0x00000002U)                    /* 推挽式复用 */
#define GPIO_MODE_AF_OD (0x00000012U)                    /* 开漏式复用 */
define GPIO_MODE_AF_INPUT  GPIO_MODE_INPUT
#define GPIO_MODE_ANALOG (0x00000003U)                    /* 模拟模式 */
#define GPIO_MODE_IT_RISING (0x10110000u)                 /* 外部中断，上升沿触发检测 */
#define GPIO_MODE_IT_FALLING (0x10210000u)                /* 外部中断，下降沿触发检测 */
 #define GPIO_MODE_IT_RISING_FALLING (0x10310000u)        /* 外部中断，上升和下降双沿触发检测 */
#define GPIO_MODE_EVT_RISING (0x10120000U)                /*外部事件，上升沿触发检测 */
#define GPIO_MODE_EVT_FALLING (0x10220000U)               /*外部事件，下降沿触发检测 */
#define GPIO_MODE_EVT_RISING_FALLING (0x10320000U)        /* 外部事件，上升和下降双沿触发检测 */

```

2.初始化GPIO配置

MX_GPIO_Init()配置结构体

```
```

```
//GPIO初始化函数以下操作:
1.使能对应GPIOX的时钟
2.根据所需，想要GPIO端口初始化为高/低
3.配置GPIO端口的参数结构体
4.调用初始化函数进行设置

使能对应外设时钟的寄存器操作:

RCC_APB2ENR_IOPAEN ------为0x00000004

#define __HAL_RCC_GPIOA_CLK_ENABLE()   do { \
                                          __IO uint32_t tmpreg; \
                                          SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);\ --------------将APB2ENR第3位
                                          /* Delay after an RCC peripheral clock enabling */\
                                          tmpreg = READ_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);\
                                          UNUSED(tmpreg); \
                                        } while(0U)

void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOD_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();

  HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_SET);
  HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_RESET);

  GPIO_InitStruct.Pin = LED1_Pin|LED2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_MEDIUM;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  }
```

▾ HAL_GPIO_Init()配置寄存器

```


GPIO配置函数以下操作:
1.用于检查传入参数的有效性，确保GPIO实例、引脚和模式都是合法的
2.遍历所有引脚，直到所有引脚都已处理,当引脚配置功能一致，可以一起配置
```

3.设置配置内容，需注意的是：设置位内容 = GPIO模式 + 速度

```
void HAL_GPIO_Init(GPIO_TypeDef  *GPIOx, GPIO_InitTypeDef *GPIO_Init)
{
  uint32_t position = 0x00u;        ------------用于跟踪当前处理的引脚位置。
  uint32_t ioposition;              ------------用于计算当前引脚的位置掩码
  uint32_t iocurrent;               ------------用于存储当前引脚的实际位置
  uint32_t temp;                    ------------临时变量，可能用于后续的位操作
  uint32_t config = 0x00u;          ------------用于存储配置的值
  __IO uint32_t *configregister;    ------------指向配置寄存器（CRL或CRH）的指针
  uint32_t registeroffset;          ------------用于计算CNF和MODE位在CRL或CRH寄存器中的位置


  //用于检查传入参数的有效性，确保GPIO实例、引脚和模式都是合法的。
  assert_param(IS_GPIO_ALL_INSTANCE(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Init->Pin));
  assert_param(IS_GPIO_MODE(GPIO_Init->Mode));

 //在while循环中，position从0开始，每次循环右移GPIO_Init->Pin的值，直到所有引脚都已处理完毕。对于每一个引脚：

   1.计算当前引脚的位置掩码ioposition。
   2.使用按位与操作&检查当前引脚是否需要配置。
   3.如果需要配置，则进入if语句块中进行具体的配置操作

假设GPIO_Init->Pin的值是0b1010（即引脚1和3需要配置），右移操作和与操作的过程如下：

    position = 0:
    ioposition = 0b0001
    iocurrent = 0b1010 & 0b0001 = 0b0000（不配置）

    position = 1:
    ioposition = 0b0010
    iocurrent = 0b1010 & 0b0010 = 0b0010（配置）

    position = 2:
    ioposition = 0b0100
    iocurrent = 0b1010 & 0b0100 = 0b0000（不配置）

    position = 3:
    ioposition = 0b1000
    iocurrent = 0b1010 & 0b1000 = 0b1000（配置）


  while (((GPIO_Init->Pin) >> position) != 0x00u)
```

```
  {

    ioposition = (0x01uL << position);


    iocurrent = (uint32_t)(GPIO_Init->Pin) & ioposition;

    if (iocurrent == ioposition)
    {
      assert_param(IS_GPIO_AF_INSTANCE(GPIOx));

      switch (GPIO_Init->Mode)
      {

        case GPIO_MODE_OUTPUT_PP:
          /* Check the GPIO speed parameter */
          assert_param(IS_GPIO_SPEED(GPIO_Init->Speed));
          config = GPIO_Init->Speed + GPIO_CR_CNF_GP_OUTPUT_PP;
          break;

          ..........................................................
          ..........................................................
          ..........................................................
      }
      //配置的GPIO端口为高位寄存器 还是 低位寄存器
      configregister = (iocurrent < GPIO_PIN_8) ? &GPIOx->CRL     : &GPIOx->CRH;
      //用于计算CNF和MODE位在CRL或CRH寄存器中的位置
      registeroffset = (iocurrent < GPIO_PIN_8) ? (position << 2u) : ((position - 8u) << 2u);

      //进行寄存器操作
      MODIFY_REG((*configregister), ((GPIO_CRL_MODE0 | GPIO_CRL_CNF0) << registeroffset), (config << registeroffset));
    }

    position++;
  }
}


```

3.GPIO功能函数

HAL_GPIO_ReadPin()

```
//读GPIO数据寄存器内容
```

```
  操作寄存器:
  GPIOx->IDR ----------IDR（16位为GPIO端口的状态）
  GPIO_Pin   ---------定义的是4位16进制数，对应的是16端口号
  &操作 --------- 0/1 取决于GPIOx->IDR的内容

GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
  GPIO_PinState bitstatus;

  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  if ((GPIOx->IDR & GPIO_Pin) != (uint32_t)GPIO_PIN_RESET)
  {
    bitstatus = GPIO_PIN_SET;
  }
  else
  {
    bitstatus = GPIO_PIN_RESET;
  }
  return bitstatus;
}


```

HAL_GPIO_WritePin()

```
//写GPIO数据寄存器思路:
1.获取GPIO数据寄存器地址
2.在对应位置写即可（1: 有效  0: 无效 ）

//BSRR寄存器 -------1: 有效  0: 无效  ------高16位置零，第16位置1
  想置1，寄存器赋值端口号   想置0: 左移16位赋值端口号


void HAL_GPIO_WritePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin,
GPIO_PinState PinState)
{
  /* Check the parameters */
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  assert_param(IS_GPIO_PIN_ACTION(PinState));

  if (PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
  else
```

```
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin << 16u;
  }
}
```

获取中断标志位和清除标志位

```
1、HAL_GPIO_TogglePin(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) ----------
------电平翻转

中断被置标志位，必须用软件清零。以下原因：
1>.中断无法再次触发：如果挂起标志位没有被清除，当同一个外部事件再次发生时，EXTI线路的中断
不会再次触发。这是因为中断控制器认为中断事件还没有处理完毕。

2>.中断处理程序可能陷入死循环：在某些情况下，如果中断处理程序没有清除挂起标志位，且中断优
先级允许嵌套中断，那么中断处理程序可能会被不断触发，导致处理程序陷入死循环。

3>.系统行为不确定：未清除的挂起标志位可能导致系统行为不确定，尤其是在复杂系统中，未清除的
中断标志位可能干扰其他中断处理逻辑，影响系统的稳定性和可靠性

2、__HAL_GPIO_EXTI_GET_FLAG(__EXTI_LINE__)----------------------------
------检查是否设置了指定的EXTI行标志
3、__HAL_GPIO_EXTI_CLEAR_FLAG(__EXTI_LINE__)--------------------------
------清除指定的EXTI行标志
4.__HAL_GPIO_EXTI_GET_IT(__EXTI_LINE__)
5.__HAL_GPIO_EXTI_CLEAR_IT(__EXTI_LINE__)

操作寄存器EXTI_PR------------当在外部中断线上发生了选择的边沿事件，该位被置’1’。在该位
中写入’1’可以清除它，也可以 通过改变边沿检测的极性清除

__HAL_GPIO_EXTI_GET_FLAG：更倾向于在一般情况下使用，用于检查是否有事件挂起。
__HAL_GPIO_EXTI_GET_IT：更倾向于在中断处理程序中使用，用于检查是否有中断挂起。

```

4.GPIO复用重映射

操作AFIO_MAPR寄存器，进行重映射

```

//重映射全部函数，在GPIO头文件有定义

AFIO_MAPR---该寄存器包含所有映射的，需注意的是下载代码的引脚是固定不变的，所以AFIO_MAPR

AFIO_MAPR-重映射，即将对应默认功能，映射到其他引脚上


AFIO_MAPR_USART1_REMAP   0x00000004         AFIO_MAPR_USART1_REMAP  0x070000
```

```
如：GPIO引脚用于USART-----#define __HAL_AFIO_REMAP_USART1_ENABLE()  AFIO_REMAP

#define AFIO_REMAP_ENABLE(REMAP_PIN)       do{ uint32_t tmpreg = AFIO->MAPR;
                                               tmpreg |= AFIO_MAPR_SWJ_CFG;
                                               tmpreg |= REMAP_PIN;
                                               AFIO->MAPR = tmpreg;
                                               }while(0u)
同理可知：
#define __HAL_AFIO_REMAP_USART1_DISABLE() AFIO_REMAP_DISABLE(AFIO_MAPR_USART
#define __HAL_AFIO_REMAP_I2C1_ENABLE()  AFIO_REMAP_ENABLE(AFIO_MAPR_I2C1_REM
#define __HAL_AFIO_REMAP_I2C1_DISABLE() AFIO_REMAP_DISABLE(AFIO_MAPR_I2C1_RE
#define __HAL_AFIO_REMAP_TIM1_ENABLE()  AFIO_REMAP_PARTIAL(AFIO_MAPR_TIM1_RE
#define __HAL_AFIO_REMAP_TIM1_DISABLE()  AFIO_REMAP_PARTIAL(AFIO_MAPR_TIM1_R
```

第五、NVIC 和 EXTI配置

1.数据结构定义

数据类型与寄存器映射

```

NVIC中断管理操作寄存器（它是内核寄存器）
typedef struct
{
  __IOM uint32_t ISER[8U];               /*  中断使能寄存器     */
        uint32_t RESERVED0[24U];
  __IOM uint32_t ICER[8U];               /* 中断清除使能寄存器 */
        uint32_t RSERVED1[24U];
  __IOM uint32_t ISPR[8U];               /* 中断使能挂起寄存器 */
        uint32_t RESERVED2[24U];
  __IOM uint32_t ICPR[8U];               /* 中断解挂寄存器     */
        uint32_t RESERVED3[24U];
  __IOM uint32_t IABR[8U];               /* 中断有效位寄存器   */
        uint32_t RESERVED4[56U];
  __IOM uint8_t  IP[240U];               /*中断优先级寄存器（8Bit 位宽）  */
        uint32_t RESERVED5[644U];
  __OM  uint32_t STIR;                   /*软件触发中断寄存器 */
}  NVIC_Type;
```

2.NVIC功能函数

HAL_NVIC_SetPriorityGrouping()--优先级分组

```

```

```
//设置抢占优先级与响应优先级的位数

void HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup以下选择)

参数 PriorityGroup以下选择

  *         @arg NVIC_PRIORITYGROUP_0: 0 bits for preemption priority
  *                                    4 bits for subpriority
  *         @arg NVIC_PRIORITYGROUP_1: 1 bits for preemption priority
  *                                    3 bits for subpriority
  *         @arg NVIC_PRIORITYGROUP_2: 2 bits for preemption priority
  *                                    2 bits for subpriority
  *         @arg NVIC_PRIORITYGROUP_3: 3 bits for preemption priority
  *                                    1 bits for subpriority
  *         @arg NVIC_PRIORITYGROUP_4: 4 bits for preemption priority
  *                                    0 bits for subpriority

```

HAL_NVIC_SetPriority()-----设置中断源的优先级

```

void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority,
uint32_t SubPriority);

参数IRQn为中端源，有40多种中源
参数PreemptPriority，SubPriority，优先级等级---0~15选择，其越低优先越高
```

HAL_NVIC_EnableIRQ-----中断使能

```

void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);

参数IRQn为中端源，有40多种中源
```

其他函数

```
void HAL_NVIC_disableIRQ(IRQn_Type IRQn);-------------中断除能函数
void HAL_NVIC_SystemReset(void); ---------------------系统复位函数
```

中断服务、处理、回调函数

```

解释一下：中断服务、处理、回调函数、中断向量表的调用关系

1.什么是中断向量表：一个存储在固定内存地址的表格，其中每个条目包含一个指向中断服务程序（ISR）的指针，当中断发生时，处理器会根据中断向量表跳转到相应的ISR，以处理特定的中断事件。
```

```
5
6  2.具体调用过程：首、先是产生中断事件、中断管理器（NVIC是内核结构）会生成一个中断请求，
7                  然、中断控制器根据中断源的优先级和当前处理器状态决定是否响应中断请求
8                  再、如果决定响应中断请求，处理器会暂停当前的执行流，并从中断向量表中查找对应中断源的ISR地址。
9                  最后、处理器会跳转到这个ISR地址，执行相应的中断处理程序。(这个程序就是中断服务函数)
10
11 3.HAL库的定义：hal库中是定义中断向量表和中断处理函数、中断回调函数。没有定义中断服务。函数名是代表地址的、通常是用向量表的函数、写一个中断服务函数
12
13 4.中断服务、处理、回调函数的调用关系
14
15 //在启动代码或向量表中定义ISR（例如，EXTI0中断）:
16 void EXTI0_IRQHandler(void);
17
18 //中断服务程序（ISR）通常由HAL库或用户代码实现:
19 void EXTI0_IRQHandler(void)
20 {
21    // 调用具体的中断处理函数
22    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
23 }
24
25 //HAL库提供的中断处理函数负责检查中断挂起标志位、清除标志位并调用回调函数:
26 void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
27 {
28   // 检查中断挂起标志位
29   if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET) {
30       // 清除中断挂起标志位
31       __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
32
33       // 调用用户定义的回调函数
34       HAL_GPIO_EXTI_Callback(GPIO_Pin);
35   }
36 }
37
38 //用户可以在应用程序中定义回调函数，以处理特定的应用逻辑:
39 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
40 {
41   if (GPIO_Pin == GPIO_PIN_0) {
42       // 处理GPIO引脚0的中断事件
43   }
44 }
45
```

第六、USART串口配置

1.数据结构定义

数据类型与寄存器映射

```

//寄存器映射----USARTX配置寄存器地址映射
//硬件的地址（包含：代码存储和寄存器），将外设基地址赋给定义的结构体

//USART1-----指向外设串口USART1第一个寄存器的地址

#define USART1_BASE         (APB2PERIPH_BASE + 0x00003800UL)
#define USART1              ((USART_TypeDef *)USART1_BASE)

//寄存器地址结构体
typedef struct
{
  __IO uint32_t SR;         /*!< USART Status register,
Address offset: 0x00 */
  __IO uint32_t DR;         /*!< USART Data register,
Address offset: 0x04 */
  __IO uint32_t BRR;        /*!< USART Baud rate register,
Address offset: 0x08 */
  __IO uint32_t CR1;        /*!< USART Control register 1,
Address offset: 0x0C */
  __IO uint32_t CR2;        /*!< USART Control register 2,
Address offset: 0x10 */
  __IO uint32_t CR3;        /*!< USART Control register 3,
Address offset: 0x14 */
  __IO uint32_t GTPR;       /*!< USART Guard time and prescaler register,
Address offset: 0x18 */
} USART_TypeDef;

//配置参数结构体
typedef struct
{
    USART_TypeDef *Instance;                        /* UART 寄存器基地址 */
    UART_InitTypeDef Init;                          /* UART 通信参数 */
    uint8_t *pTxBuffPtr;                            /* 指向 UART 发送缓冲区 */
    uint16_t TxXferSize;                            /* UART 发送数据的大小 */
    __IO uint16_t TxXferCount;                      /* UART 发送数据的个数 */
    uint8_t *pRxBuffPtr;                            /* 指向 UART 接收缓冲区 */
    uint16_t RxXferSize;                            /* UART 接收数据大小 */
    __IO uint16_t RxXferCount;                      /* UART 接收数据的个数 */
    DMA_HandleTypeDef *hdmatx;                      /* UART 发送参数设置（DMA）
*/
    DMA_HandleTypeDef *hdmarx;                      /* UART 接收参数设置（DMA）
*/
```

```
    HAL_LockTypeDef Lock;                                /* 锁定对象 */
    __IO HAL_UART_StateTypeDef gState;                   /* UART 发送状态结构体 */
    __IO HAL_UART_StateTypeDef RxState;                  /* UART 接收状态结构体 */
    __IO uint32_t ErrorCode;                             /* UART 操作错误信息 */
}UART_HandleTypeDef;


//UART通信参数结构体
typedef struct
{
    uint32_t BaudRate;                                    /* 波特率 */
    uint32_t WordLength;                                  /* 字长 */
    uint32_t StopBits;                                    /* 停止位 */
    uint32_t Parity;                                      /* 校验位 */
    uint32_t Mode;                                        /* UART 模式 */
    uint32_t HwFlowCtl;                                   /* 硬件流设置 */
    uint32_t OverSampling;                                /* 过采样设置 */
}UART_InitTypeDef;

//通常用户在需要配置通信参数结构体，即可，其他参数通常都传参，通过函数进行配置：
1）BaudRate：波特率设置。一般设置为 2400、9600、19200、115200。
2）WordLength：数据帧字长，可选 8 位或 9 位。这里我们设置为 8 位字长数据格式。
3）StopBits：停止位设置，可选 0.5 个、1 个、1.5 个和 2 个停止位，一般我们选择 1 个停止位。
4）Parity：奇偶校验控制选择，我们设定为无奇偶校验位。
5）Mode：UART 模式选择，可以设置为只收模式，只发模式，或者收发模式。这里我们设置 为全双工收发模式。
6）HwFlowCtl：硬件流控制选择，我们设置为无硬件流控制。
7）OverSampling：过采样选择，选择 8 倍过采样或者 16 过采样，一般选择 16 过采样。
```

2.初始化USART配置

MX_USART1_UART_Init()配置结构体

```
//USART初始化函数以下操作：
1.配置USART外设参数结构体
2.调用初始化函数进行设置
3.初始化函数是调用了，回调函数HAL_UART_MspInit()
4.在回调函数中配置GPIO-NVIC

void MX_USART1_UART_Init(void)
{
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 9600;
    huart1.Init.WordLength = UART_WORDLENGTH_8B;
    huart1.Init.StopBits = UART_STOPBITS_1;
    huart1.Init.Parity = UART_PARITY_NONE;
```

```
    huart1.Init.Mode = UART_MODE_TX_RX;
    huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart1.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart1) != HAL_OK)
    {
        Error_Handler();
    }
}
```

HAL_UART_Init()配置寄存器

```


句柄是什么?
句柄是一种用于管理和操作硬件外设或软件资源的结构体。它包含了外设的配置信息、状态信息以及操作该外设所需的函数指针等。通过使用句柄，可以使外设的初始化、配置和操作更加模块化和抽象化，便于管理和维护代码。

//初始化函数以下操作
1.判断传参的句柄是否有效
2.用于检查传入参数的有效性
3.调用回调函数HAL_UART_MspInit()
4.禁用外设USART
5.设置UART通信参数
6.清除异步通信，要清除的位（在异步模式下，以下位必须保持清空）
7.启用外设USART
8.设置tx、rx的状态（就绪、繁忙、超时）


//HAL_UART_Init()----------删除一些判断是否有效

HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart)
{
  if (huart == NULL)
  {
    return HAL_ERROR;
  }
.........................................
.........................................
  //调用回调函数HAL_UART_MspInit()
  HAL_UART_MspInit(huart);

  huart->gState = HAL_UART_STATE_BUSY;

  //禁用外设USART
  __HAL_UART_DISABLE(huart);

  //设置UART通信参数
```

```
  UART_SetConfig(huart);

  //清除异步通信，要清除的位（在异步模式下，以下位必须保持清空）
  CLEAR_BIT(huart->Instance->CR2, (USART_CR2_LINEN | USART_CR2_CLKEN));
  CLEAR_BIT(huart->Instance->CR3, (USART_CR3_SCEN | USART_CR3_HDSEL |
USART_CR3_IREN));

  //启用外设USART
  __HAL_UART_ENABLE(huart);

  //设置tx、rx的状态（就绪、繁忙、超时）
  huart->ErrorCode = HAL_UART_ERROR_NONE;
  huart->gState = HAL_UART_STATE_READY;
  huart->RxState = HAL_UART_STATE_READY;
  huart->RxEventType = HAL_UART_RXEVENT_TC;

  return HAL_OK;
}

------------操作寄存器---------------


USART_CR1_UE   0x00002000
//启用外设USART--------操作USART_CR1寄存器的第13位----------UE：USART使能位
#define __HAL_UART_ENABLE(__HANDLE__)               ((__HANDLE__)-
>Instance->CR1 |=  USART_CR1_UE)


USART_CR2_LINEN    0x00004000       USART_CR2_CLKEN   0x00000800
//在异步模式下，有些位必须保持清空-----操作USART_CR2(USART_CR3)----------CLKEN：时
钟使能、LINEN：LIN模式使能
CLEAR_BIT(huart->Instance->CR2, (USART_CR2_LINEN | USART_CR2_CLKEN));
//同理可知
CLEAR_BIT(huart->Instance->CR3, (USART_CR3_SCEN | USART_CR3_HDSEL |
USART_CR3_IREN));

```

▾ UART_SetConfig()通信参数配置

```

//分寄存器进行配置

static void UART_SetConfig(UART_HandleTypeDef *huart)
{
  /*------------------------- USART CR2 Configuration --------------------
---*/
  /* Configure the UART Stop Bits: Set STOP[13:12] bits
     according to huart->Init.StopBits value */
```

```
  MODIFY_REG(huart->Instance->CR2, USART_CR2_STOP, huart->Init.StopBits);

  /*-------------------------- USART CR1 Configuration -----------------------*/
  tmpreg = (uint32_t)huart->Init.WordLength | huart->Init.Parity | huart->Init.Mode;
  MODIFY_REG(huart->Instance->CR1,
             (uint32_t)(USART_CR1_M | USART_CR1_PCE | USART_CR1_PS | USART_CR1_TE | USART_CR1_RE),
             tmpreg);

 /*-------------------------- USART CR3 Configuration -----------------------*/
  /* Configure the UART HFC: Set CTSE and RTSE bits according to huart->Init.HwFlowCtl value */
  MODIFY_REG(huart->Instance->CR3, (USART_CR3_RTSE | USART_CR3_CTSE), huart->Init.HwFlowCtl);

   /*-------------------------- USART BRR Configuration -----------------------*/
#if defined(USART_CR1_OVER8)
  if (huart->Init.OverSampling == UART_OVERSAMPLING_8)
  {
    huart->Instance->BRR = UART_BRR_SAMPLING8(pclk, huart->Init.BaudRate);
  }
  else
  {
    huart->Instance->BRR = UART_BRR_SAMPLING16(pclk, huart->Init.BaudRate);
  }
#else
  huart->Instance->BRR = UART_BRR_SAMPLING16(pclk, huart->Init.BaudRate);

}
```

HAL_UART_MspInit()

```
//USART回调函数以下操作:
1.判断是否是对应外设句柄-----初始化函数是所有串口外设共用的，所以回调函也是共用的
2.配置tx、rx的引脚
3.配置中断优先级和使能


void HAL_UART_MspInit(UART_HandleTypeDef* uartHandle)
{

    GPIO_InitTypeDef GPIO_InitStruct = {0};
    if(uartHandle->Instance==USART1)
    {
```

```
        /* USART1
        __HAL_RCC_USART1_CLK_ENABLE();

        __HAL_RCC_GPIOA_CLK_ENABLE();

        /**USART1 GPIO Configuration
        PA9     ------> USART1_TX
        PA10     ------> USART1_RX
        */

        GPIO_InitStruct.Pin = GPIO_PIN_9;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

        GPIO_InitStruct.Pin = GPIO_PIN_10;
        GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

        /* USART1 interrupt Init */
        HAL_NVIC_SetPriority(USART1_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(USART1_IRQn);
    }
}
```

3.USART功能函数

HAL_UART_Transmit()

在HAL库中，使用状态机来管理UART的发送和接收操作有以下几个主要原因：

1.资源共享：由于发送和接收共用同一个数据寄存器，状态机可以确保在任何时刻只进行一个操作，避免数据冲突。
2.中断处理：状态机可以帮助管理中断的优先级和顺序，确保数据的正确传输和接收。
3.错误处理：状态机可以有效管理和处理在发送和接收过程中可能出现的各种错误（如超时、帧错误、奇偶校验错误等）。
4.数据流控制：状态机可以确保数据流在发送和接收之间有序进行，避免数据丢失或重复。

该函数内容：
1.检查发送串口状态是否就绪-----由于发送和接收共用同一个数据寄存器，状态机可以确保在任何时刻只进行一个操作，避免数据冲突------考虑状态机
2.检查函数传参是否有效
3.置USART结构体的状态---------为无错误，发送busy
4.获取滴答定时器值为判断发送超时函数的参数---------这里的超时是指将所有数据全部发送完

5.设置结构体------待发送内容地址，内容字节大小
6.由于发送数据桢---8位和9位桢结构（含奇偶校验位）-------因此9位就不可能用 uint8*就_t *存储，而是用uint16_t *-----我们用8位桢
7.轮询发送，发送次数为发送字节数-----因为串口一次发送8个字节
8.检测超时,将内容赋值给发送寄存器，减字节数，轮询完置USART为就绪态

为什么指针有类型----指针不就是存地址的吗?

因为指针是有运算的，指针+1“相当于”跳过对应类型字节大小，比如uint8*就跳过8字节，指向对应位置

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, const uint8_t *pData, uint16_t Size, uint32_t Timeout)
{
  const uint8_t  *pdata8bits;
  const uint16_t *pdata16bits;
  uint32_t tickstart = 0U;

  /* Check that a Tx process is not already ongoing */
  if (huart->gState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return  HAL_ERROR;
    }

    huart->ErrorCode = HAL_UART_ERROR_NONE;
    huart->gState = HAL_UART_STATE_BUSY_TX;

    /* Init tickstart for timeout management */
    tickstart = HAL_GetTick();

    huart->TxXferSize = Size;
    huart->TxXferCount = Size;

    /* In case of 9bits/No Parity transfer, pData needs to be handled as a uint16_t pointer */
    if ((huart->Init.WordLength == UART_WORDLENGTH_9B) && (huart->Init.Parity == UART_PARITY_NONE))
    {
      pdata8bits  = NULL;
      pdata16bits = (const uint16_t *) pData;
    }
    else
```

```
58      {
59        pdata8bits  = pData;
60        pdata16bits = NULL;
61      }
62
63      while (huart->TxXferCount > 0U)
64      {
65        if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TXE, RESET,
    tickstart, Timeout) != HAL_OK)
66        {
67          return HAL_TIMEOUT;
68        }
69        if (pdata8bits == NULL)
70        {
71          huart->Instance->DR = (uint16_t)(*pdata16bits & 0x01FFU);
72          pdata16bits++;
73        }
74        else
75        {
76          huart->Instance->DR = (uint8_t)(*pdata8bits & 0xFFU);
77          pdata8bits++;
78        }
79        huart->TxXferCount--;
80      }
81
82      if (UART_WaitOnFlagUntilTimeout(huart, UART_FLAG_TC, RESET, tickstart,
    Timeout) != HAL_OK)
83      {
84        return HAL_TIMEOUT;
85      }
86
87      /* At end of Tx process, restore huart->gState to Ready */
88      huart->gState = HAL_UART_STATE_READY;
89
90      return HAL_OK;
91    }
92    else
93    {
94      return HAL_BUSY;
95    }
96  }
```

超时判断机制

```
1
2  该函数的主体的-----通过滴答定时器来计算时间，并且与超时时间比较，判断是否超时，通过发送完成标志位来描述过程
3
```

该函数参数：
串口句柄-------来置状态机----若超时，即重启---串口就绪
标志位------发送完成标志
滴答定时器第一获取次值
超时时间

```
static HAL_StatusTypeDef UART_WaitOnFlagUntilTimeout(UART_HandleTypeDef *huart, uint32_t Flag, FlagStatus Status,
                                                     uint32_t Tickstart, uint32_t Timeout)
{
  /* Wait until flag is set */
  while ((__HAL_UART_GET_FLAG(huart, Flag) ? SET : RESET) == Status)
  {
    /* Check for the Timeout */
    if (Timeout != HAL_MAX_DELAY)
    {
      if ((Timeout == 0U) || ((HAL_GetTick() - Tickstart) > Timeout))
      {
        /* Disable TXE, RXNE, PE and ERR (Frame error, noise error, overrun error) interrupts for the interrupt process */
        ATOMIC_CLEAR_BIT(huart->Instance->CR1, (USART_CR1_RXNEIE | USART_CR1_PEIE | USART_CR1_TXEIE));
        ATOMIC_CLEAR_BIT(huart->Instance->CR3, USART_CR3_EIE);

        huart->gState  = HAL_UART_STATE_READY;
        huart->RxState = HAL_UART_STATE_READY;

        /* Process Unlocked */
        __HAL_UNLOCK(huart);

        return HAL_TIMEOUT;
      }
    }
  }
  return HAL_OK;
}
```

HAL_UART_Receive_IT()

```

//USART串口接收中断-------该函数就是--检查状态机，调用UART_Start_Receive_IT(huart, pData, Size)(为函数主体)

HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
  /* Check that a Rx process is not already ongoing */
```

```
  if (huart->RxState == HAL_UART_STATE_READY)
  {
    if ((pData == NULL) || (Size == 0U))
    {
      return HAL_ERROR;
    }

    /* Set Reception type to Standard reception */
    huart->ReceptionType = HAL_UART_RECEPTION_STANDARD;

    return (UART_Start_Receive_IT(huart, pData, Size));
  }
  else
  {
    return HAL_BUSY;
  }
}


//该函数内容----配置结构体的接收内容----并使能错误中断和接收寄存器非空中断

----接收内容处理在回调函数里
----

HAL_StatusTypeDef UART_Start_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
{
  huart->pRxBuffPtr = pData;
  huart->RxXferSize = Size;
  huart->RxXferCount = Size;

  huart->ErrorCode = HAL_UART_ERROR_NONE;
  huart->RxState = HAL_UART_STATE_BUSY_RX;

  if (huart->Init.Parity != UART_PARITY_NONE)
  {
    /* Enable the UART Parity Error Interrupt */
    __HAL_UART_ENABLE_IT(huart, UART_IT_PE);
  }

  /* Enable the UART Error Interrupt: (Frame error, noise error, overrun error) */
  __HAL_UART_ENABLE_IT(huart, UART_IT_ERR);

  /* Enable the UART Data Register not empty Interrupt */
  __HAL_UART_ENABLE_IT(huart, UART_IT_RXNE);

  return HAL_OK;
```

```
}


重点：通常pData设置为8 bit， Size 1字节 ------中断一次，可以在回调函数里处理接收数据内容，并重新调用中断使能
      若是pData是数组，size不是1的话------由于串口是接收1字节中断一次，不处理接收的数据，就会被覆盖----所以处理数据
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
{
    // 处理接收中断
    if (__HAL_UART_GET_IT(huart, UART_IT_RXNE))
    {
        // 从接收寄存器读取数据
        *huart->pRxBuffPtr++ = (uint8_t)(huart->Instance->DR & 0xFF);

        // 更新接收计数
        if (--huart->RxXferCount == 0)
        {
            // 接收完成
            __HAL_UART_DISABLE_IT(huart, UART_IT_RXNE);
            huart->RxState = HAL_UART_STATE_READY;

            // 调用接收完成回调函数
            HAL_UART_RxCpltCallback(huart);
        }
    }
}
```

HAL_UART_IRQHandler()

以下是 HAL_UART_IRQHandler 的主要功能和流程：

1.检查并处理 RXNE 中断：

  如果接收数据寄存器非空中断（RXNE）被触发，读取接收寄存器中的数据，并存储到接收缓冲区中。
  更新接收计数，检查是否接收到预期的字节数。如果接收完成，禁用 RXNE 中断并调用接收完成回调函数。

2.检查并处理 TXE 中断：

  如果发送数据寄存器空中断（TXE）被触发，将下一个字节写入发送寄存器。
  更新发送计数，检查是否所有数据已发送。如果发送完成，禁用 TXE 中断并调用发送完成回调函数。

3.检查并处理 TC 中断：

  如果传输完成中断（TC）被触发，更新传输状态并调用传输完成回调函数。

4.检查并处理错误中断：

  如果发生错误中断，如帧错误、噪声错误、溢出错误等，读取错误标志，清除错误标志，并调用错误回调函数。

5.清除中断标志位

在处理完中断之后，通常需要清除相应的中断标志位，以避免重复触发。对于某些错误中断，需要手动清除标志位，例如：

  帧错误（FE）：通过读取状态寄存器（SR）并读取数据寄存器（DR）来清除。
  溢出错误（ORE）：通过读取状态寄存器（SR）并读取数据寄存器（DR）来清除。
  噪声错误（NE）：通过读取状态寄存器（SR）并读取数据寄存器（DR）来清除。

对于其他中断，如 RXNE 和 TXE，硬件会在适当的时候自动清除标志位

HAL_UART_Receive_DMA()

__HAL_UART_ENABLE_IT(HANDLE, INTERRUPT)

参数情况：

__HANDLE__: Enable the specified UART interrupt.

__INTERRUPT__: specifies the UART interrupt source to enable

USART中断源有：这里中断源，只是触发方式

```
@arg UART_IT_CTS:  CTS change interrupt

*            @arg UART_IT_LBD:  LIN Break detection interrupt
*            @arg UART_IT_TXE:  Transmit Data Register empty interrupt
*            @arg UART_IT_TC:   Transmission complete interrupt
*            @arg UART_IT_RXNE: Receive Data register not empty interrupt
*            @arg UART_IT_IDLE: Idle line detection interrupt
*            @arg UART_IT_PE:   Parity Error interrupt
*            @arg UART_IT_ERR:  Error interrupt(Frame error, noise error, overrun error)

#define __HAL_UART_DISABLE_IT(__HANDLE__, __INTERRUPT__)
```

```
 ((((__INTERRUPT__) >> 28U) == UART_CR1_REG_INDEX)? ((__HANDLE__)->Instance->CR1 &= ~((__INTERRUPT__) & UART_IT_MASK)): \
 (((__INTERRUPT__) >> 28U) == UART_CR2_REG_INDEX)? ((__HANDLE__)->Instance->CR2 &= ~((__INTERRUPT__) & UART_IT_MASK)): \
 ((__HANDLE__)->Instance->CR3 &= ~ ((__INTERRUPT__) & UART_IT_MASK)))

```

其他函数

```
//Checks whether the specified UART flag is set or not
//@retval : The new state of __FLAG__ (TRUE or FALSE)
__HAL_UART_GET_FLAG(**HANDLE**, **FLAG**)

//Clears the specified UART pending flag
//retval None
__HAL_UART_CLEAR_FLAG(**HANDLE**, **FLAG**)

//Checks whether the specified UART interrupt source is enabled or not
//retval The new state of __IT__ (TRUE or FALSE).
__HAL_UART_GET_IT_SOURCE(**HANDLE**, **IT**)


```

第七、TIM定时器配置

第八、RTC实时时钟配置

1.数据结构定义

数据类型与寄存器映射

```

//寄存器映射----RTC配置寄存器地址映射
//硬件的地址（包含：代码存储和寄存器），将外设基地址赋给定义的结构体


#define RTC           ((RTC_TypeDef *)RTC_BASE)
#define RTC_BASE      (APB1PERIPH_BASE + 0x00002800UL)

//寄存器地址结构体
typedef struct
{
  __IO uint32_t CRH;
```

```
  __IO uint32_t CRL;
  __IO uint32_t PRLH;
  __IO uint32_t PRLL;
  __IO uint32_t DIVH;
  __IO uint32_t DIVL;
  __IO uint32_t CNTH;
  __IO uint32_t CNTL;
  __IO uint32_t ALRH;
  __IO uint32_t ALRL;
} RTC_TypeDef;

配置参数结构
typedef struct
{
    RTC_TypeDef *Instance;                                  /* 寄存器基地址 */
    RTC_InitTypeDef Init;                                   /* RTC 配置结构体 */
    RTC_DateTypeDef DataToUpdata;                           /* RTC 日期结构体 */
    HAL_LockTypeDef Lock;                                   /* RTC 锁定对象 */
    __IO HAL_RTCStateTypeDef State;                         /* RTC 设备访问状态 */
}RTC_HandleTypeDef;

//配置RTC参数结构体
typedef struct
{
    uint32_t AsynchPrediv;                                  /* 异步预分频系数 */
    uint32_t OutPut;                                        /* 被发送到 RTC Tamper
引脚的信号 */
}RTC_InitTypeDef;

//AsynchPrediv 用来设置 RTC 的异步预分频系数
--------我们使用外部 32.768K 的晶振作为时钟的输入频率， 那么我们要设置这两个寄存器的值
为 32767，得到一秒钟的计数频率

//OutPut 用 来 选 择 RTC 输 出 到 Tamper 引脚的信号

//日期结构体
typedef struct
{
  uint8_t WeekDay;
  uint8_t Month;
  uint8_t Date;
  uint8_t Year;
} RTC_DateTypeDef;

//闹钟结构体
typedef struct
{
  RTC_TimeTypeDef AlarmTime;     -----设置什么时候闹钟触发
```

```
  uint32_t Alarm;                 ------哪个闹钟触发--STM32F1只有一个
} RTC_AlarmTypeDef;
```

2.初始化RTC实时配置

配置RTC流程

```

1.配置给RTC提供时钟的晶振，并配置选择器和分频器
2.配置初始化结构体---配置RTC的结构体--句柄和分频器，进行寄存器初始化
3.回调初始化函数-使能电源时钟，和RCC时钟、备份寄存器时钟、解除写保护，若要中断，则配置中断优先级和使能中断
4.设置时间和日期，可通过函数调用（操作RTC计数器实现）
5.若要配置闹钟中断，即先配置闹钟结构体，调要闹钟中断即可（有闹钟中断使能的）
6.获取时间
```

MX_RTC_Init()配置结构体

```

//RTC配置初始化函数以下操作:
1.配置RTC参数结构体，并调用配置寄存器函数进行配置
2.配置时间与日期结构体
3.调用的HAL的设置时间和日期的函数

//这里配置的数据和日期存在代码里（不是寄存器里），但计数值是保存在寄存器里

void MX_RTC_Init(void)
{
  RTC_TimeTypeDef sTime = {0};
  RTC_DateTypeDef DateToUpdate = {0};


  hrtc.Instance = RTC;
  hrtc.Init.AsynchPrediv = RTC_AUTO_1_SECOND;
  hrtc.Init.OutPut = RTC_OUTPUTSOURCE_ALARM;
  if (HAL_RTC_Init(&hrtc) != HAL_OK)
  {
    Error_Handler();
  }


  sTime.Hours = 0x11;
  sTime.Minutes = 0x11;
  sTime.Seconds = 0x11;

  if (HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BCD) != HAL_OK)
  {
```

```
    Error_Handler();
  }
  DateToUpdate.WeekDay = RTC_WEEKDAY_SUNDAY;
  DateToUpdate.Month = RTC_MONTH_MARCH;
  DateToUpdate.Date = 0x3;
  DateToUpdate.Year = 0x24;

  if (HAL_RTC_SetDate(&hrtc, &DateToUpdate, RTC_FORMAT_BCD) != HAL_OK)
  {
    Error_Handler();
  }
/

}
```

HAL_RTC_Init()配置寄存器

```

注这里的LSE振荡器，分频系数，是系统初始化由stm32cubeMx初始化好了

//初始化函数以下操作
1.是否进行了RTC输出---没有、闹钟脉冲信号输出、秒脉冲信号输出，分别进行不通的操作
2.配置分频系数--AsynchPrediv是否配置RTC_AUTO_1_SECOND，若是则系统根据RCC提供的时钟进行结算，得到1s计时。若不是，则自己保证1s
3.初始化默认时间

HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc)
{
.....................................
.....................................
.....................................

    /* Clear Flags Bits */
    CLEAR_BIT(hrtc->Instance->CRL, (RTC_FLAG_OW | RTC_FLAG_ALRAF | RTC_FLAG_SEC));

    if (hrtc->Init.OutPut != RTC_OUTPUTSOURCE_NONE)
    {
      /* Disable the selected Tamper pin */
      CLEAR_BIT(BKP->CR, BKP_CR_TPE);
    }

    /* Set the signal which will be routed to RTC Tamper pin*/
    MODIFY_REG(BKP->RTCCR, (BKP_RTCCR_CCO | BKP_RTCCR_ASOE | BKP_RTCCR_ASOS), hrtc->Init.OutPut);

    if (hrtc->Init.AsynchPrediv != RTC_AUTO_1_SECOND)
    {
```

```
29        /* RTC Prescaler provided directly by end-user*/
30        prescaler = hrtc->Init.AsynchPrediv;
31      }
32      else
33      {
34        /* RTC Prescaler will be automatically calculated to get 1 second
  timebase */
35        /* Get the RTCCLK frequency */
36        prescaler = HAL_RCCEx_GetPeriphCLKFreq(RCC_PERIPHCLK_RTC);
37
38        /* Check that RTC clock is enabled*/
39        if (prescaler == 0U)
40        {
41          /* Should not happen. Frequency is not available*/
42          hrtc->State = HAL_RTC_STATE_ERROR;
43          return HAL_ERROR;
44        }
45        else
46        {
47          /* RTC period = RTCCLK/(RTC_PR + 1) */
48          prescaler = prescaler - 1U;
49        }
50      }
51
52      /* Configure the RTC_PRLH / RTC_PRLL */
53      WRITE_REG(hrtc->Instance->PRLH, ((prescaler >> 16U) & RTC_PRLH_PRL));
54      WRITE_REG(hrtc->Instance->PRLL, (prescaler & RTC_PRLL_PRL));
55
56      /* Wait for synchro */
57      if (RTC_ExitInitMode(hrtc) != HAL_OK)
58      {
59        hrtc->State = HAL_RTC_STATE_ERROR;
60
61        return HAL_ERROR;
62      }
63
64      /* Initialize date to 1st of January 2000 */
65      hrtc->DateToUpdate.Year = 0x00U;
66      hrtc->DateToUpdate.Month = RTC_MONTH_JANUARY;
67      hrtc->DateToUpdate.Date = 0x01U;
68
69      /* Set RTC state */
70      hrtc->State = HAL_RTC_STATE_READY;
71
72      return HAL_OK;
73  }
74
```

HAL_RTC_MspInit()

```

//RTC回调函数：
1.使能电源时钟 PWR----------------系统已经设置好
2.使能备份时钟----操作的APB1ENR
3.取消备份区域写保护HAL_PWR_EnableBkUpAccess()（直接操作的是寄存器位）-----备份寄存器被侵入，寄存器会清零


void HAL_RTC_MspInit(RTC_HandleTypeDef* rtcHandle)
{

  if(rtcHandle->Instance==RTC)
  {
  /* USER CODE BEGIN RTC_MspInit 0 */

  /* USER CODE END RTC_MspInit 0 */
    HAL_PWR_EnableBkUpAccess();
    /* Enable BKP CLK enable for backup registers */
    __HAL_RCC_BKP_CLK_ENABLE();
    /* RTC clock enable */
    __HAL_RCC_RTC_ENABLE();

    /* RTC interrupt Init */
    HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
  /* USER CODE BEGIN RTC_MspInit 1 */

  /* USER CODE END RTC_MspInit 1 */
  }
}
```

使能中断函数

```

```

3.RTC功能函数

HAL_RTC_SetTime()/HAL_RTC_GetTime()

```

//RTC 核心：由一组可编程计数器组成，主要分成两个模块。第一个模块是 RTC 的预分 频模块，它可编程产生 1 秒的 RTC 时间基准 TR_CLK。RTC 的预分频模块包括了一个 20 位的 可编程分频器(RTC 预分频器)。如果在 RTC_CR 寄存器中设置相对应的允许位，则在每个 TR_CLK 周期中 RTC 产生一个中断(秒中断)。第二个模块是一个 32 位的可编程计数器，可被 初始化为当前的系统时间，
```

```
一个 32 位的时钟计数器，按秒钟计算，可以记录 4294967296 秒， 约合 136 年左右，作为一般应用足够了

//
设置时间：RTC通过获取时间和日期，来更新RTC计数器，而RTC计数时钟1s使RTC计数器+1
获取时间：通过RTC计数器，来算当前时间和日期

//
我们要访问 RTC 和 RTC 备份区域就必须先使能电源时钟，然后使能 RTC 即后备区域访问。
电源时钟使能，通过 RCC_APB1ENR 寄存器来设置；RTC 及 RTC 备份寄存器的写访问，通过 PWR_CR 寄存器的 DBP 位设置

//
这里的时间设置和时间获取------------------重写（好好看）


uint8_t rtc_set_time(uint16_t syear, uint8_t smon, uint8_t sday, uint8_t hour, uint8_t min, uint8_t sec)
{
     uint32_t seccount = 0;
     seccount = rtc_date2sec(syear, smon, sday, hour, min, sec); /* 将年月日时分秒转换成总秒钟数 */
     __HAL_RCC_PWR_CLK_ENABLE(); /* 使能电源时钟 */
    __HAL_RCC_BKP_CLK_ENABLE(); /* 使能备份域时钟 */
     HAL_PWR_EnableBkUpAccess(); /* 取消备份域写保护 */

     RTC->CRL |= 1 << 4; /* 进入配置模式 */

     //配置RTC计数器
     RTC->CNTL = seccount & 0xffff;
     RTC->CNTH = seccount >> 16;

     RTC->CRL &= ~(1 << 4); /* 退出配置模式 */
     /* 等待 RTC 寄存器操作完成, 即等待 RTOFF == 1 */
     while (!__HAL_RTC_ALARM_GET_FLAG(&g_rtc_handle, RTC_FLAG_RTOFF));

     return 0;
}



//HAL提供
HAL_RTC_GetTime(&hrtc,(RTC_TimeTypeDef*)&time->Hours,RTC_FORMAT_BIN);

HAL_RTC_SetTime(&hrtc,(RTC_TimeTypeDef*)&time->Hours,RTC_FORMAT_BIN);
```

▾ HAL_RTC_SetDate()/HAL_RTC_GetDate

```

HAL_RTC_GetDate(&hrtc,(RTC_DateTypeDef*)&time->WeekDay,RTC_FORMAT_BIN);


HAL_RTC_SetDate(&hrtc,(RTC_DateTypeDef*)&time->WeekDay,RTC_FORMAT_BIN);
```

HAL_RTC_SetAlarm_IT()

```

配置闹钟寄存器，并使能闹钟中断


```

第九、DMA转运配置

1.数据结构定义

数据类型与寄存器映射

```

//寄存器映射----DMA配置寄存器地址映射
//硬件的地址（包含：代码存储和寄存器），将外设基地址赋给定义的结构体

#define DMA1_Channel6       ((DMA_Channel_TypeDef *)DMA1_Channel6_BASE)
#define DMA1_Channel6_BASE    (AHBPERIPH_BASE + 0x0000006CUL)

//DMA寄存器通道寄存器（DMA1---1到7）
typedef struct
{
  __IO uint32_t CCR;
  __IO uint32_t CNDTR;
  __IO uint32_t CPAR;
  __IO uint32_t CMAR;
} DMA_Channel_TypeDef;

//对通道进行配置
typedef struct __DMA_HandleTypeDef
{
    DMA_Channel_TypeDef          *Instance;       /* 寄存器基地址 */

  DMA_InitTypeDef               Init;             /* DAM 通信参数 */


```

```
  HAL_LockTypeDef                Lock;              /* DMA 锁对象 */

  __IO HAL_DMA_StateTypeDef      State;             /* DMA 传输状态 */

  void                          *Parent;            /* DMA 传输状态 */


} DMA_HandleTypeDef;

typedef struct
{
    uint32_t Direction;                             /* 传输方向，例如存储器到外设 DMA_MEMORY_TO_PERIPH */
    uint32_t PeriphInc;                             /* 外设（非）增量模式，非增量模式 DMA_PINC_DISABLE */
    uint32_t MemInc;                                /* 存储器（非）增量模式，增量模式 DMA_MINC_ENABLE */
    uint32_t PeriphDataAlignment;                   /* 外设数据大小：8/16/32 位 */
    uint32_t MemDataAlignment;                      /* 存储器数据大小：8/16/32 位 */
    uint32_t Mode;                                  /* 模式：循环模式/普通模式 */
    uint32_t Priority;                              /* DMA 优先级：低/中/高/非常高 */

}DMA_InitTypeDef;

//很明显配置的结构体中是没有发送地址和目的地址的，通过函数进行连接的

//
HAL 库为了处理各类外设的 DMA 请求,在调用相关函数之前，需要调用一个宏定义标识符，来连接DMA和外设句柄。例如要使用串口DMA发送，所以方式为：

  __HAL_LINKDMA(uartHandle,hdmarx,hdma_usart2_rx);

 比如：g_uart1_handler 是串口初始化句柄，我们在 usart.c 中定义过了。g_dma_handle 是 DMA 初始化句柄。hdmatx 是外设句柄结构体的成员变量

//在能够通过DMA转运的外设，它的结构体中都用接收数据缓存的地址，和发送数据的缓存地址
```

2.初始化DMA转运配置

DMA转运流程

```

//Stm32cubeMX的配置流程

```

1.（自编写的DMA初始化函数）：只需使能DMA时钟，使能中断和配置优先级，配置结构体参数在对应的转运外设初始函数里
2.这里中断（通道转运完成中断），中断的内容---处理接收的数据

```
//中断函数
void DMA1_Channel5_IRQHandler(void)
{
    // 检查是否为传输完成中断
    if (DMA_GetITStatus(DMA1_IT_TC5))
    {
        // 清除中断标志
        DMA_ClearITPendingBit(DMA1_IT_TC5);
        // 处理接收到的数据
        ProcessReceivedData();
    }
}
```

3.配置DMA结构体-----调初始化DMA结构体函数，进行初始化
4.连接 DMA 和外设句柄：   __HAL_LINKDMA(uartHandle,hdmarx,hdma_usart2_rx);

----------内容就将hdmarx（DMA的结构体内容） 赋值给 uartHandle（UART的结构体）的成员DMA结构体

uartHandle：初始化UART的结构体类型
hdmarx：UART的结构体内容DMA结构体类型
hdma_usart2_rx：初始化DMA结构体类型

```
#define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD__, __DMA_HANDLE__)
        \
                          do{
     \
                                (__HANDLE__)->__PPP_DMA_FIELD__ = &
(__DMA_HANDLE__); \
                                (__DMA_HANDLE__).Parent = (__HANDLE__);
      \
                            } while(0U)
```

5.调用DMA串口接收函数：
HAL_UART_Receive_DMA(&huart2,Usart2type.Usart2DMARecBuff,USART2_DMA_REC_SIZE);

huart2 ： UART外设函数

3.DMA功能函数

函数

```

  __HAL_RCC_DMA1_CLK_ENABLE(); /* DMA1 时钟使能 */
  __HAL_RCC_DMA2_CLK_ENABLE(); /* DMA2 时钟使能 */

 // HAL 库还提供了对串口的 DMA 发送的停止，暂停，继续等操作函数：
   HAL_StatusTypeDef HAL_UART_DMAStop(UART_HandleTypeDef *huart); /* 停止 */
   HAL_StatusTypeDef HAL_UART_DMAPause(UART_HandleTypeDef *huart); /* 暂停 */
   HAL_StatusTypeDef HAL_UART_DMAResume(UART_HandleTypeDef *huart); /* 恢复 */

   //查询 DMA 传输通道的状态:是否转运完成
   __HAL_DMA_GET_FLAG(&g_dma_handle, DMA_FLAG_TC4);

   __HAL_DMA_CLEAR_FLAG(&g_dma_handle, DMA_FLAG_TC4);

```

```

```

第十、ADC模数转化配置

1.数据结构定义

数据类型与寄存器映射

```

//寄存器映射----ADC配置寄存器地址映射
//硬件的地址（包含：代码存储和寄存器），将外设基地址赋给定义的结构体

#define ADC1                    ((ADC_TypeDef *) ADC1_BASE)
#define ADC1_BASE               (APB2PERIPH_BASE + 0x00002400UL)

//ADC寄存器通道寄存器
typedef struct
{
  __IO uint32_t SR;
  __IO uint32_t CR1;
  __IO uint32_t CR2;
  __IO uint32_t SMPR1;
  __IO uint32_t SMPR2;
  __IO uint32_t JOFR1;
  __IO uint32_t JOFR2;
  __IO uint32_t JOFR3;
```

```
  __IO uint32_t JOFR4;
  __IO uint32_t HTR;
  __IO uint32_t LTR;
  __IO uint32_t SQR1;
  __IO uint32_t SQR2;
  __IO uint32_t SQR3;
  __IO uint32_t JSQR;
  __IO uint32_t JDR1;
  __IO uint32_t JDR2;
  __IO uint32_t JDR3;
  __IO uint32_t JDR4;
  __IO uint32_t DR;
} ADC_TypeDef;

//ADC结构体
typedef struct
{
    ADC_TypeDef *Instance;                      /* ADC 寄存器基地址 */
    ADC_InitTypeDef Init;                       /* ADC 参数初始化结构体变量
*/
    DMA_HandleTypeDef *DMA_Handle;              /* DMA 配置结构体 */
    HAL_LockTypeDef Lock;                       /* ADC 锁定对象 */
    __IO uint32_t State;                        /* ADC 工作状态 */
    __IO uint32_t ErrorCode;                    /* ADC 错误代码 */
}ADC_HandleTypeDef;

//ADC参数初始化结构体
typedef struct
{
    uint32_t DataAlign;                         /* 设置数据的对齐方式 */
    uint32_t ScanConvMode;                      /* 扫描模式 */
    FunctionalState ContinuousConvMode;         /* 开启连续转换模式否则就是单次
转换模式 */
    uint32_t NbrOfConversion;                   /* 设置转换通道数目 */
    FunctionalState DiscontinuousConvMode;      /* 是否使用规则通道组间断模式
*/
    uint32_t NbrOfDiscConversion;               /* 配置间断模式的规则通道个数
*/
    uint32_t ExternalTrigConv;                  /* ADC 外部触发源选择 */
} ADC_InitTypeDef;


1) DataAlign：用于设置数据的对齐方式，这里可以选择右对齐或者是左对齐，该参数可选为：
ADC_DATAALIGN_RIGHT 和 ADC_DATAALIGN_LEFT。
2) ScanConvMode：配置是否使用扫描。如果是单通道转换使用 ADC_SCAN_DISABLE，如果 是多
通道转换使用 ADC_SCAN_ENABLE。
3) ContinuousConvMode：可选参数为 ENABLE 和 DISABLE，配置自动连续转换还是单次转
换。使用 ENABLE 配置为使能自动连续转换；使用 DISABLE 配置为单次转换，转换一次 后停止需
```

要手动控制才重新启动转换。
4) NbrOfConversion：指定规则组转换通道数目，范围是：1~16。
5) DiscontinuousConvMode：配置是否使用规则通道组间断模式，比如要转换的通道有 1、2、5、7、8、9，那么第一次触发会进行通道 1 和 2，下次触发就是转换通道 5 和 7，这样不连 续的转换，依次类推。此参数只有将 ScanConvMode 使能，还有 ContinuousConvMode 失能 的情况下才有效，不可同时使能。
6) NbrOfDiscConversion：配置间断模式的通道个数，禁止规则通道组间断模式后，此参数忽略。
7) ExternalTrigConv：外部触发方式的选择，如果使用软件触发，那么外部触发会关闭。

```
//ADC 通道配置函数
typedef struct
{
  uint32_t Channel;                          /* ADC 转换通道*/
  uint32_t Rank;                             /* ADC 转换顺序 */
  uint32_t SamplingTime;                     /* ADC 采样周期 */
} ADC_ChannelConfTypeDef;

```

2.初始化ADC模数转化配置

ADC配置流程

```

1.配置RCC给ADC采集时钟，若是用stm32cubeMX，设置在SystemClock_Config()函数中
2.配置ADC结构体（外设基地址、状态变量、DMA转运），ADC参数初始化结构体（采集方式：单次、连续、触发方式等等），调用初始化寄存器函数
3.在回调函数中，使能时钟（ADC,GPIO）、配置GPIO引脚(模拟输入)、配置ADC 通道配置函数（建立与ADC的联系）
4.即外设配完，调用校准，开始采样-----采样是否完成，获取ADC的值，停止采样

```

MX_ADC1_Init()函数初始化

```

//
1.配置的是规则组-----扫描-连续模式-软件触发-转运通道的个数-数据右对齐
2.调用寄存器初始化
3.配置通道结构体（哪个通道，转化顺序，采样周期）

void MX_ADC1_Init(void)
{
  ADC_ChannelConfTypeDef sConfig = {0};

  hadc1.Instance = ADC1;
  hadc1.Init.ScanConvMode = ADC_SCAN_ENABLE;
```

```
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = ENABLE;
  hadc1.Init.NbrOfDiscConversion = 1;
  hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 4;
  if (HAL_ADC_Init(&hadc1) != HAL_OK)
  {
    Error_Handler();
  }

  sConfig.Channel = ADC_CHANNEL_0;
  sConfig.Rank = ADC_REGULAR_RANK_1;
  sConfig.SamplingTime = ADC_SAMPLETIME_1CYCLE_5;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }


  sConfig.Channel = ADC_CHANNEL_1;
  sConfig.Rank = ADC_REGULAR_RANK_2;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }


  sConfig.Channel = ADC_CHANNEL_4;
  sConfig.Rank = ADC_REGULAR_RANK_3;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }


  sConfig.Channel = ADC_CHANNEL_5;
  sConfig.Rank = ADC_REGULAR_RANK_4;
  if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }


}
```

▾ HAL_ADC_MspInit()

```

//使能时钟（GPIO和ADC）+ 配置GPIO


void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
{

  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(adcHandle->Instance==ADC1)
  {

    __HAL_RCC_ADC1_CLK_ENABLE();

    __HAL_RCC_GPIOA_CLK_ENABLE();
    /**ADC1 GPIO Configuration
    PA0-WKUP     ------> ADC1_IN0
    PA1     ------> ADC1_IN1
    PA4     ------> ADC1_IN4
    PA5     ------> ADC1_IN5
    */
    GPIO_InitStruct.Pin = GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_4|GPIO_PIN_5;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);


  }
}
```

3.ADC模数转化配置功能函数

功能函数的运用

```

//ADC 的自校准函数
HAL_StatusTypeDef HAL_ADCEx_Calibration_Start(ADC_HandleTypeDef *hadc);

//ADC 转换启动函数
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef *hadc);

//等待ADC规则组转换完成函数
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef *hadc,
uint32_t Timeout);

//停止ADC 转换启动函数
HAL_ADC_Stop(&hadc1);

//获取常规组 ADC 转换值函数
```

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef *hadc);


运用:
  HAL_ADCEx_Calibration_Start(&hadc1);

    for(uint8_t i=0; i<50; i++)
    {

            for(uint8_t i=0; i<4; i++)
            {
                    HAL_ADC_Start(&hadc1);
                    HAL_ADC_PollForConversion(&hadc1, 50);
                    ADC_Value[i] = ADC_Value[i] + HAL_ADC_GetValue(&hadc1);
            }
            HAL_ADC_Stop(&hadc1);
    }
    for(uint8_t i=0; i<4; i++)
    {
            ADC_Value[i] = ADC_Value[i]/50;
    }
```