

STM32L0 使用指南

作者: 向仔州

CuBumx 5.3.0 运行环境搭建.....	4
CuBuMX 5.3.0 使用.....	5
GPIO 输出和我的 F072 文档一样.....	9
USART2 串口 2 CuBuMX 配置.....	10
串口接收中断配置.....	11
串口发送数据例程.....	12
串口中断接收数据例程(该方式在 PC 端发送数据很快的情况下会出现接收卡死).....	12
串口重定向实现 printf (和 F072 一样)	15
采用空闲中断处理串口接收的不定长度数据.....	15
空闲中断实现串口接收不定长度数据(接收数据每次长度都不一样).....	17
__HAL_UART_ENABLE_IT(...) //开启空闲中断.....	17
串口+DMA 实现空闲中断.....	18
__HAL_UART_ENABLE_IT(...) //开启串口接收中断.....	18
__HAL_UART_ENABLE_IT(...) //开启串口 2 空闲接收中断.....	18
__HAL_UART_CLEAR_IDLEFLAG(...); //清除串口 2 空闲中断标志位, 防止开机不停的触发空闲中断.....	18
HAL_UART_Receive_DMA(...)//使能 DMA 中断接收.....	19
串口+DMA 空闲中断 HAL_UART_DMAStop(...) //关闭 DMA 库函数 BUG, 一定要注意.....	21
如果不使用系统默认引脚做串口, 而是使用复用引脚做串口, 忘记在 CubeMX 设置, 那么就要修改库函数.....	22
STM32L 串口 1 开机启动, 马上用串口 1 接收发送数据会出现死机现象(注意).....	22
HardFault_Handler () 问题.....	23
ADC 初始化造成串口中断无法接收, 重大问题.....	24
为了防止串口数据丢包, 在 DMA 基础上, 采用环形队列做缓存.....	25
C 语言实现串口队列缓存.....	28
GPIO 输入输出(和 F072 操作一样).....	32
HAL_GPIO_WritePin (...) //GPIO 输出电平和 F072 操作一样	
IO 口输入电平检测.....	34
STM32L051 引脚外部输入中断使用.....	35
STM32L051 定时器 2 使用.....	36
HAL_TIM_Base_Start_IT(...) //启动定时器.....	37
STM32L0 支持读写内部 FLASH(EEROM)功能, 将数据永久保存.....	38
HAL_FLASH_Unlock(); //开锁, 写 flash 要开锁之后才能写.....	38
HAL_FLASH_Program(...) //数据写入内部 FLASH	
HAL_FLASH_Lock(); //锁住, 写完 flash 之后要锁住, 数据才能保存在 flash 中	
如何向 EEPROM 连续写入数据呢? 比如数组存储 EEPROM.....	45
STM32L0 芯片获取全球唯一 ID.....	46
用 SYSTICK 做系统, 微妙, 毫秒延时.....	48
STM32L0 RTC 时间操作.....	49
HAL_RTC_GetTime(...) //获取时分秒函数.....	50

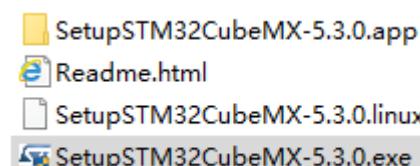
HAL_RTC_GetDate(...) //获取年月日函数.....	51
设置 RTC 时间，然后读出来.....	51
HAL_RTC_SetTime(...) //向 RTC 内部设置时间，时分秒.....	51
HAL_RTC_SetDate(...) //向 RTC 内部设置时间，年月日.....	52
RTC 外部时钟使用.....	53
RTC 闹钟 A 中断.....	54
HAL_RTC_AlarmAEventCallback(...)	55
闹钟 A 实现自定义时间重复中断.....	56
STM32L0 低功耗实现.....	57
STM32 停止模式.....	57
HAL_RTCEx_SetWakeUpTimer_IT(...) //设置唤醒计数器与中断.....	57
HAL_PWR_EnterSTOPMode(...) //进入停止模式，谨慎使用这种模式，操作不好可能导致单片机无法下载程序.....	57
STM32 睡眠模式实现.....	60
HAL_PWR_EnterSLEEPMode(...) //进入睡眠模式.....	60
睡眠模式 RTC 闹钟 A 唤醒.....	61
STM32 超低功耗睡眠模式.....	63
有一种办法能解决低功耗无法下载程序的情况.....	64
超低功耗睡眠模式 RTC 闹钟 A 唤醒.....	64
待机模式.....	66
HAL_PWR_EnableWakeUpPin(...) //设置 WKUP1 或者 WKUP2 为唤醒引脚.....	66
_HAL_RCC_PWR_CLK_ENABLE() // 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式.....	66
HAL_PWR_EnterSTANDBYMode() //进入待机模式.....	66
待机模式唤醒操作.....	66
ADC 使用.....	69
ADC 单通道采集实现.....	69
Sampling Time 采样时间详解，主要得出 ADC 转换一次数据的时间.....	70
ADC DMA 实现多通道采集.....	72
HAL_ADC_Start_DMA(...) //启动 DMA ADC, 获取 ADC 数据.....	73
用 ADC DMA 获取温度传感器通道和内部参考电压通道的数据.....	74
ADC 过采样实现.....	75
C 语言过采样代码实现.....	77
STM32 硬件过采样实现.....	78
STM32 ADC 用内部电压，采集芯片供电电压，监控芯片电源.....	79
在使用 STM32 内部 VREFINT 做基准电压反算 VDDA 电压时，或者启动 ADC 多个通道采集时一定要注意采样时间大小问题(CYCLES), ADC_SAMPLETIME_1CYCLE_5 1.5 个 CYCLES 整了我一天。.....	81
在 ADC DMA 中，多个 ADC 通道采集，但是通道不是连续的，是跳跃的，比如只采集 1, 3, 5 通道，那么 DMA 缓存是按照 5 个数组元素依次摆放的吗?，如 1, 2, 3, 4, 5，我们用数组下标取 1, 3, 5 就能获取 ADC 1,3,5 通道的数据 ? 其实这是不行的.....	82
LPUART1 低功耗串口使用.....	82
LPUART1 常规功耗接收数据测试.....	83
LPUART1 + DMA + 空闲中断接收数据.....	84

LPUART1 低功耗唤醒单片机.....	86
当移植一个用 HAL 生成好的驱动程序到另外一个 L051 工程的时候，注意配置文件.....	88
报错：出现 identifieris undefined 未定义某个驱动函数，(重点).....	88
IO 管脚外部中断实现.....	88
在使用 cubemx 生成新代码的时候，如何保证以前工程的代码不被覆盖?(重点).....	91
独立看门狗 IWDG.....	92
HAL_IWDG_Refresh(...) //独立看门狗喂狗.....	93
IWDG 唤醒休眠模式的 MCU.....	94
窗口看门狗(WWDG).....	95
HAL_WWDG_Refresh(...) //窗口看门狗喂狗.....	96

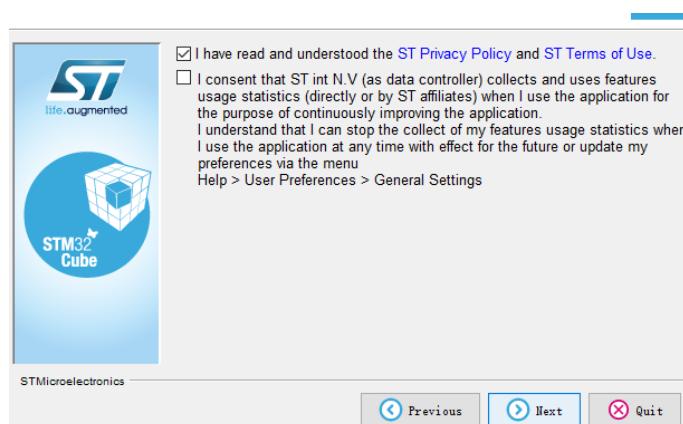
CuBumx 5.3.0 运行环境搭建

下载适用于 Windows 的 Java

推荐 Version 8 Update 271 (文件大小: 1.98 MB)
发行日期: 2020 年 10 月 20 日



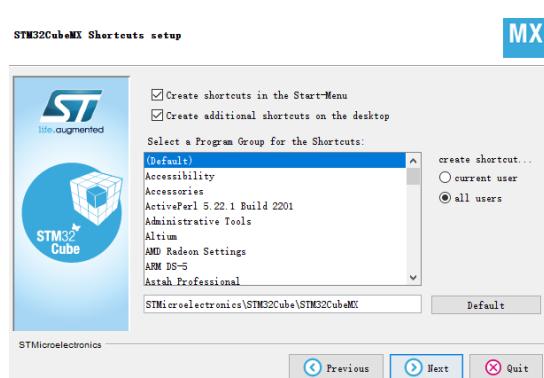
运行 CubeMX 安装软件



Previous Next Quit

打钩，下一步

安装路径默认 C:\Program Files (x86)\STMicroelectronics\STM32Cube\STM32CubeMX



将开始栏的软件图标拷贝到桌面

打开 MX 软件

CuBuMX 5.3.0 使用

有些 CuBuMX 软件操作可以参考我《STM32F072-M0_User_guide.pdf》，有些不能参考。因为软件版本有更新，现在是 5.3.0 了。

搭建基础工程，LED 为例



STM32CubeMX Untitled

File Wind

Home >

New Project ... Ctrl-N

Load Project ... Ctrl-L

搜索想要的芯片 STM32L051

STM32L051C8

Core >

Series >

Line >

Package >

Other >

Price = 1.314

IO = 37

Eeprom = 2048 (Bytes)

Flash = 64 (kBytes)

Ram = 8 (kBytes)

Freq. = 32 (MHz)

Ultra-low-power Arm Cortex-M0+ MCU with 64 Kbytes of Flash memory, 32 MHz

ACTIVE Active Product is in mass production

Unit Price for 10KU (US\$) : 1.314

The access line ultra-low-power STM32L051x6/8 microcontrollers incorporate the high-performance Arm Cortex-M0+ 32-bit R1 32 MHz frequency, a memory protection unit (MPU), high-speed embedded memories (64 Kbytes of Flash program memory, 2 EEPROM and 8 Kbytes of RAM) plus an extensive range of enhanced I/Os and peripherals.

The STM32L051x6/8 devices provide high power efficiency for a wide range of performance. It is achieved with a large choice external clock sources, an internal voltage adaptation and several low-power modes.

The STM32L051x6/8 devices offer several analog features, one 12-bit ADC with hardware oversampling, two ultra-low-power timers, one low-power timer (LPTIM), three general-purpose 16-bit timers and one basic timer, one RTC and one SysTick with timebases. They also feature two watchdogs, one watchdog with independent clock and window capability and one window wa clock.

Moreover, the STM32L051x6/8 devices embed standard and advanced communication interfaces: up to two I2C, two SPIs, on

MCUs/MPUs List: 2 items

Part No	Reference	Marketing St.	Unit Price for 10KU...	Board	Package	Flash	RAM
STM32L051C8Tx	Active	1.314		LQFP48	64 kBytes	8 kBytes	3
STM32L051C8...	Active	1.314		UFQFP...	64 kBytes	8 kBytes	3

Pinout & Configuration

Clock Configuration

Additional Software

Mode

High Speed Clock (HSE) Disable

Low Speed Clock (LSE) Disable

Master Clock Output 1

Categories A-Z

System Core

DMA

GPIO

IWDG

NVIC

RCC

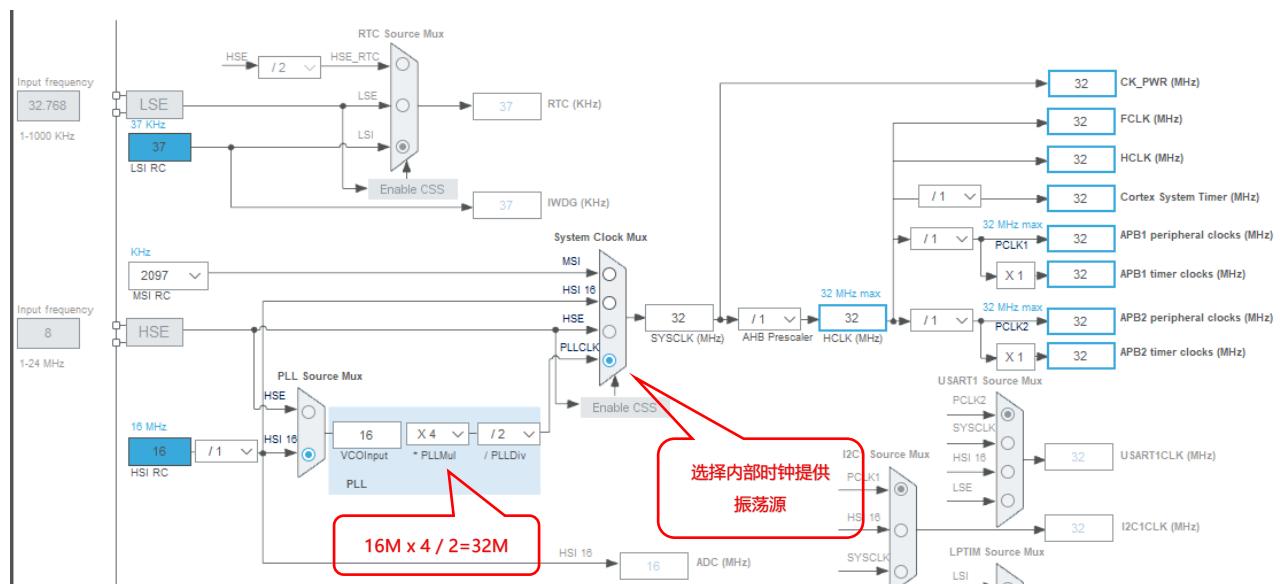
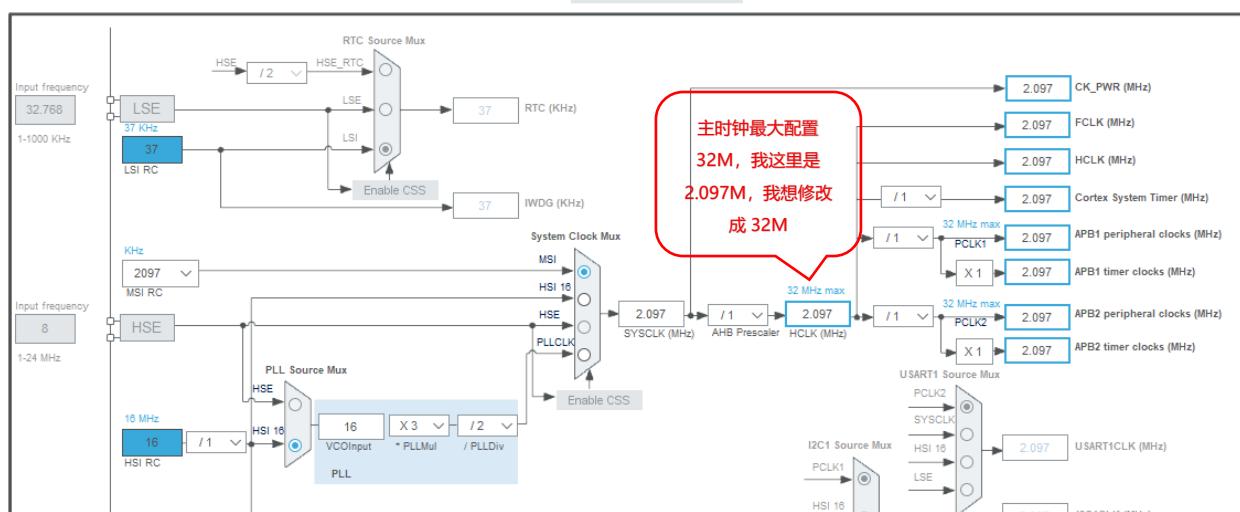
SYS

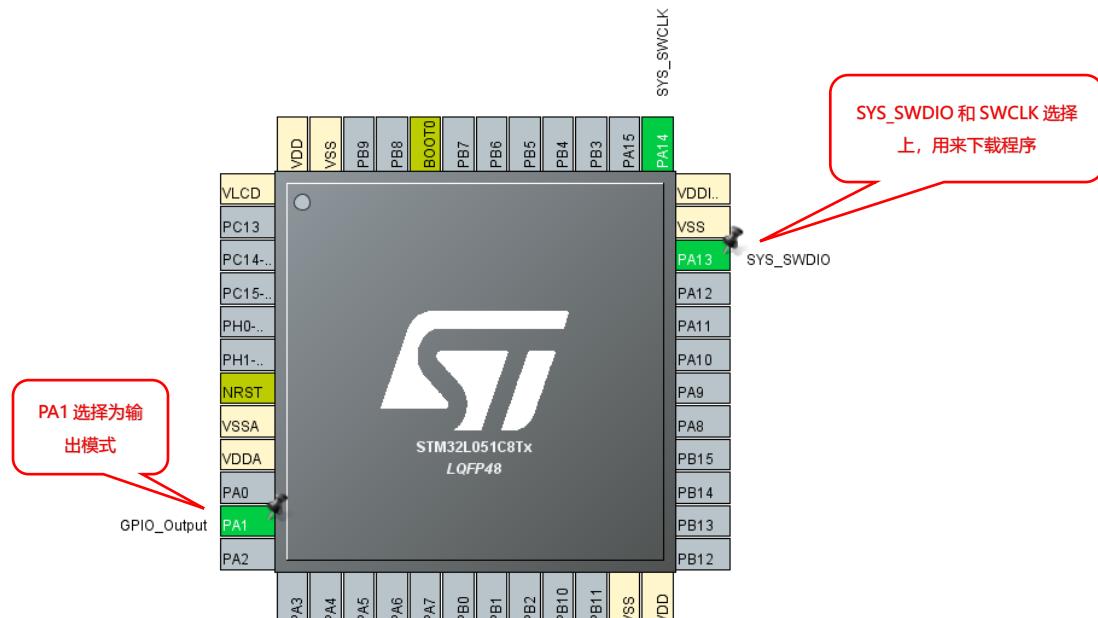
WWDG

Analog

Configuration

Reset Configuration





Pinout & Configuration

Clock Configuration

Additional Software

Pins

Categories: A-Z

System Core: DMA, GPIO, IWDG, NVIC, RCC, SYS, WWDG

GPIO Mode and Configuration

Configuration

Group By Peripherals:

GPIO: SYS:

Search Signals: Search (Ctrl+F)

Show only Modified Pins:

Pin Name	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pull-down	Maximum output speed	Fast Mode	User Label	Modified
PA1	n/a	Low	Output Push...	No pull-up ...	Low	n/a		<input type="checkbox"/>

然后点击这行

点击 GPIO 就可以选择你配置的 GPIO

GPIO: SYS:

Search Signals: Search (Ctrl+F)

Show only Modified Pins:

Pin Name	Signal on Pin	GPIO output level	GPIO mode	GPIO Pull-up/Pull-down	Maximum output speed	Fast Mode	User Label	Modified
PA1	n/a	Low	Output Push...	No pull-up ...	Low	n/a		<input type="checkbox"/>

PA1 Configuration:

GPIO output level	Low	配置 GPIO 工作模式
GPIO mode	Output Push Pull	
GPIO Pull-up/Pull-down	No pull-up and no pull-down	
Maximum output speed	Low	给 GPIO 取名字
User Label		

配置工程输出

The screenshot shows the Project Manager tab of the CubuMX interface. It includes sections for Project Settings (Project Name: XSTM32L0, Project Location: D:/02 系统夹\Desktop\STM32L0), Application Structure (Basic), Toolchain Folder Location (D:\02 系统夹\Desktop\STM32L0\xSTM32L0\), Toolchain / IDE (MDK-ARM V5), and Linker Settings (Minimum Heap Size: 0x200, Minimum Stack Size: 0x400). Red annotations highlight: "取工程名" (Get Project Name) pointing to the Project Name field; "选择工程管理" (Select Project Management) pointing to the Project Manager tab; "选择工程保存路径, 但是这里有中文名" (Select project save path, but there is Chinese name) pointing to the Project Location field; and "选择 Keil5 编译" (Select Keil5 Compilation) pointing to the Toolchain / IDE dropdown.

This screenshot shows the Project Manager tab with detailed settings. Under Project, it lists STM32Cube MCU packages and embedded software packs, with the option "Copy all used libraries into the project folder" selected. Under Code Generator, it shows Generated files settings: "Generate peripheral initialization as a pair of 'c/.h' files per peripheral" (checked), "Backup previously generated files when re-generating" (unchecked), "Keep User Code when re-generating" (checked), and "Delete previously generated files when not re-generated" (checked). A red annotation box with the text "记住, 在 Code Generator 下选择上输出 C/H 文件" (Remember, under Code Generator, select output C/H files) is placed over the checked "Generate peripheral initialization" option.

The screenshot shows the Project Manager tab again, with the GENERATE CODE button highlighted in blue at the top right. A red annotation box with the text "点击右上角生成工程" (Click the top right corner to generate the project) points to this button.

注意: 如果是中文路径工程将生成失败, 如果生成过程中遇到更新, 你点击 yes 就是。

This screenshot shows a search results window titled "Search needed files for the project...". It displays a list of files found in the project. Below the search bar is a "Code Generation" window with "Open Folder" and "Close" buttons.

A warning message box states: "The Code is successfully generated under D:/02 系统夹/Desktop/STM32L0/xSTM32L0 but MDK-ARM V5project generation have a problem." An "OK" button is at the bottom.

这就是中文路径的问题。

如果是英文路径, 就生成下面这样,

A success message box states: "The Code is successfully generated under E:/STM32L0Project/xSTM32L0". It includes "Open Folder", "Open Project", and "Close" buttons.

成功了, 可以打开编译使用。

注意: CubuMX 在以前生成的工程中, 再使用 **XSTM32L0.ioc** 选择结果新外设生成工程, 这样将会把以前的工程逻辑全部覆盖完。所以 CubuMX 移植性差的原因就在这里, 该问题还是没有解决。所以只有把需要的, 不需要的通通生成好。

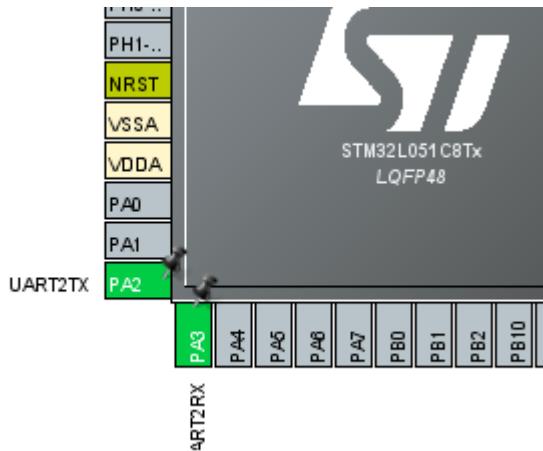
GPIO 输出和我的 F072 文档一样

```
int main(void)
{
    HAL_Init();
    SystemClock_Config(); //时钟初始化
    MX_GPIO_Init(); //GPIO初始化

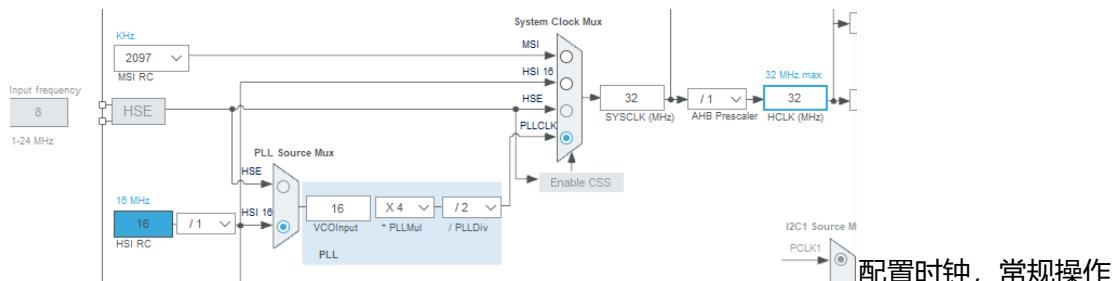
    while (1)
    {
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_Delay(500);

        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_RESET);
        HAL_Delay(500);
    }
}
```

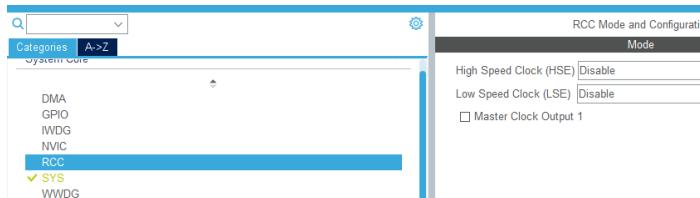
USART2 串口 2 CuBuMX 配置



按照 f072 文档选择 IO 端口为串口



配置时钟，常规操作



选择内部时钟



按照 F072 方式配置串口就是

串口接收中断配置

NVIC Settings DMA Settings GPIO Settings
 Parameter Settings User Constants
 NVIC Interrupt Table Enabled Preempt
 USART2 global interrupt / USART2 wake-up interrupt through EXTI line... 3

- I2C1
- I2C2
- LPUART1
- SPI1
- SPI2
- USART1
- USART2**

然后去 NVIC 配置项选择串口中断优先级

Pin ...	Signal o...	GPIO ou...	GPIO m...	GPIO P...	Maximu...	Fast Mode	User Label	Modified
PA2	USART2...	n/a	Alternat...	No pull...	Very High	n/a	UART2TX	<input checked="" type="checkbox"/>
PA3	USART2...	n/a	Alternat...	Pull-up	Very High	n/a	UART2RX	<input checked="" type="checkbox"/>

Categories A-Z

System Core

- DMA
- GPIO
- IWDG
- NVIC**
- RCC
- SYS**
- WWDG

Analog

Timers

GPIO

- IWDG
- NVIC
- RCC
- SYS**
- WWDG

PA3 Configuration :

GPIO mode: Alternate Function Push Pull

GPIO Pull-up/Pull-down: Pull-up

Maximum output speed: Very High

User Label: UART2RX

Search Signals: 串口接收, 中断要将接收引脚设置成下拉模式

Show only Modified Pins:

生成代码

串口发送数据例程

```
int main(void)
{
    uint8_t data = 0xfe;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init(); //里面只有配置GPIOA的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在cubeMX默认初始化的

    while (1)
    {
        HAL_UART_Transmit(&huart2, &data, 1, 1); //发送字节
        HAL_Delay(500);
    }
    /* USER CODE END 3 */
}
```

串口中断接收数据例程(该方式在 PC 端发送数据很快的情况下会出现接收卡死)

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
//开启串口接收中断，每次中断之后，都要再执行一次才能再开启接收中断
huart: 写入串口号
pData: 传入接收中断存放数据的地址
Size: 接收数据的数量
HAL_StatusTypeDef: 返回 HAL_OK 成功，HAL_ERROR 错误，HAL_BUSY 占用，HAL_TIMEOUT 超时
```

```
static unsigned char pdata;

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init(); //里面只有配置 GPIOA 的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在 cubeMX 默认初始化的
    HAL_UART_Receive_IT(&huart2, &pdata, 1); //一定要先执行一次接收，才能实现中断接收

    while (1)
    {
        HAL_Delay(1000);
    }
}

void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
    HAL_UART_Receive_IT(&huart2, &pdata, 1); //接受电脑发过来的数据
    HAL_UART_Transmit(&huart2, &pdata, 1, 1); //发数据到 PC 将数据转发回电脑
}
```

优化串口接受中断

```
void MX_USART2_UART_Init(void)
{
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 9600;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart2) != HAL_OK)
    {
        Error_Handler();
    }
    HAL_UART_MspInit(&huart2);
}
```

其实 HAL 生成的串口没有初始化中断优先级，在串口初始化之后可以加入

HAL_UART_MspInit 可能不需要在 Init 初始化，貌似有系统回调函数解决。

这里要说明一下

调用 HAL_UART_Receive_IT() 相当于开启接收中断 ---> 如果有数据，会触发中断 USART2_IRQHandler() ---> HAL_UART_IRQHandler() ---> UART_Receive_IT()

所以进入中断之后，如果不调用 HAL_UART_Receive_IT()

```
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
    printf("USART2 IRQ\r\n");

    // HAL_UART_Transmit(&huart2, &pdata, 1, 1); // 发数据到PC, 将数据转发回电脑
    // HAL_UART_Receive_IT(&huart2, &pdata, 1); // 接受电脑发过来的数据
}
```

如果把 HAL_UART_Receive_IT 屏蔽了的话，那么系统下次得到的串口中断不会被触发。好恶心

```
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);
    printf("USART2 IRQ\r\n");

    // HAL_UART_Transmit(&huart2, &pdata, 1, 1); // 发数据到PC, 将数据转发回电脑
    HAL_UART_Receive_IT(&huart2, &pdata, 1); // 接受电脑发过来的数据
}
```

打开 HAL_UART_Receive_IT



PC 1m 秒发送一次数据，相当于 1K 的速度了，也没有问题。

所以还得调整下 HAL_UART_Receive_IT 调用方式

```
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);

    HAL_UART_Receive(&huart2, &pdata, 1, 1); //因为是1个字节，所以接收长度为1
    HAL_UART_Transmit(&huart2, &pdata, 1, 1); //发数据到PC,将数据转发回电脑

    HAL_UART_Receive_IT(&huart2, &pdata, 1); //接受电脑发过来的数据
}
```

用 Receive 接受一个字节，发送一个字节，可以在 1ms 时间间隔正常工作

但是这样操作发现会出现数据丢失的情况

因为，不管是发送数据还是接收数据都会触发中断，进而最终会调用到 `UART_Receive_IT()` 这个函数，在这个函数里面分别对接收和发送做了处理，可以看看源码。

```
static unsigned char pdata;
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init(); //里面只有配置 GPIOA 的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在 cubeMX 默认初始化的
    HAL_UART_Receive_IT(&huart2,&pdata,1); //一定要先执行一次接收，才能实现中断接收

    while (1)
    {
    }
}

void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2);

    HAL_UART_Receive_IT(&huart2,&pdata,1); //开启中断
    HAL_UART_Transmit(&huart2,&pdata,1); //发送之后触发中断，导致中断消耗一帧
    HAL_UART_Receive_IT(&huart2,&pdata,1); //重新开启中断，补充消耗的一次中断
}
```

这样修改后，PC 发到板子，板子转发到 PC 的数据正常。

串口重定向实现 printf (和 F072 一样)

```
#include "main.h"
#include "usart.h"
#include "gpio.h"
#include "stdio.h" //一定要加入 stdio.h FILE 变量才有效

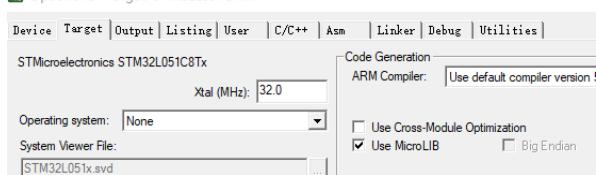
void SystemClock_Config(void);

int fputc(int ch,FILE *f)
{
    HAL_UART_Transmit(&huart2,(uint8_t *) &ch,1,1); //重定向发送字符
    return ch;
}

int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init(); //里面只有配置 GPIOA 的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在 cubeMX 默认初始化的

    while (1)
    {
        printf("xxxxxxxx\r\n");
        HAL_Delay(1000);
    }
}
```



记住一定要勾选上 Use MicroLIB

采用空闲中断处理串口接收的不定长度数据

采用空闲中断做接收之前，要先实现串口回调函数

```
void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2); //清除中断标志位
}
```

在中断函数中取消掉逻辑代码，清除中断就行了

```
/*不管是串口1接收中断，还是串口2接收中断都要进入HAL_UART_RxCpltCallback回调函数*/
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)
{
    if(UartHandle->Instance == USART2) //所以我必须判断是不是串口2接收中断
    {
        printf("Rx Callback\r\n");
        HAL_UART_Receive_IT(&huart2,&pdata,1); //重新开启中断，补充消耗的一次中断
    }
}
```

一旦串口接收中断发生，系统会自动去执行 `void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)` 回调函数，所以我们只需要定义实现回调函数就是了。

```
static unsigned char pdata;

int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init(); //这里面只有配置 GPIOA 的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在 cubeMX 默认初始化的
    HAL_UART_Receive_IT(&huart2,&pdata,1); //一定要先执行一次接收，才能实现中断

    while (1)
    {
    }
}

void USART2_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart2); //清除中断标志位
}
```

```
/*不管是串口 1 接收中断，还是串口 2 接收中断都要进入 HAL_UART_RxCpltCallback 回调函数*/
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle)
{
    if(UartHandle->Instance == USART2) //所以我必须判断是不是串口 2 接收中断
    {
        //printf("Rx Callback\r\n");
        /*中断发生后 pdata 变量值已经被系统串口接收的数据修改，这里只需要发回 PC 就行*/
        HAL_UART_Transmit(&huart2,&pdata,1,1);
        while(HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX); //检测 UART 发送结束
        HAL_UART_Receive_IT(&huart2,&pdata,1); //重新开启串口接收中断
    }
}
++ -- . . . . .
11 22 33 44 55 66 77 88 99
11 22 33 44 55 66 77 88 99 11 22 33 44 55 66 77 88 99
11 22 33 44 55 66 77 88 99
11 22 33 44 55 66 77 88 99
11 22 33 44 55 66 77 88 99
```

这种单字节接收发送还是会出现一帧数据字节超出范围，多接受一帧数据的情况。

最后发现不是代码问题，是串口接收软件的问题

使用大虾丁丁版的串口助手就没有这个问题。而且我也是一帧数据很多字节发送，时间间隔 100ms 所以串口回调函数实现的代码没有问题。而且串口软件的接收超时时间是 20ms 太小了，可以改 100ms

空闲中断实现串口接收不定长度数据(接收数据每次长度都不一样)

空闲中断也叫作帧中断，判断串口一个数据包接收完之后产生的中断，怎么理解呢？

常规的串口中断接收流程



1.这种做法的问题在于，如果你主程序执行的任务很多，就很有可能因为串口频繁中断，造成主程序运行不流畅，

2.就是在串口数据到来的时候发生了中断，我对中断进行标记status = 1。这时候主程序就要循环到status = 1的位置去处理串口程序

```
void USART2_IRQHandler(void)
{
    status = 1 // 中断发生了，需要主程序处理
}
main()
{
    while(1)
    {
        .....经过很多程序.....
        if(status == 1)
            处理串口程序
    }
}
```

如果主程序在这里处理了很多程序，这时候串口又接收到一个字节数据，但是主程序还没有执行到status == 1去处理上一次串口中断的数据，这样就会造成数据字节丢失

空闲中断的优势，除了接收不定长度数据外，还可以开辟一个很大的缓存给空闲中断

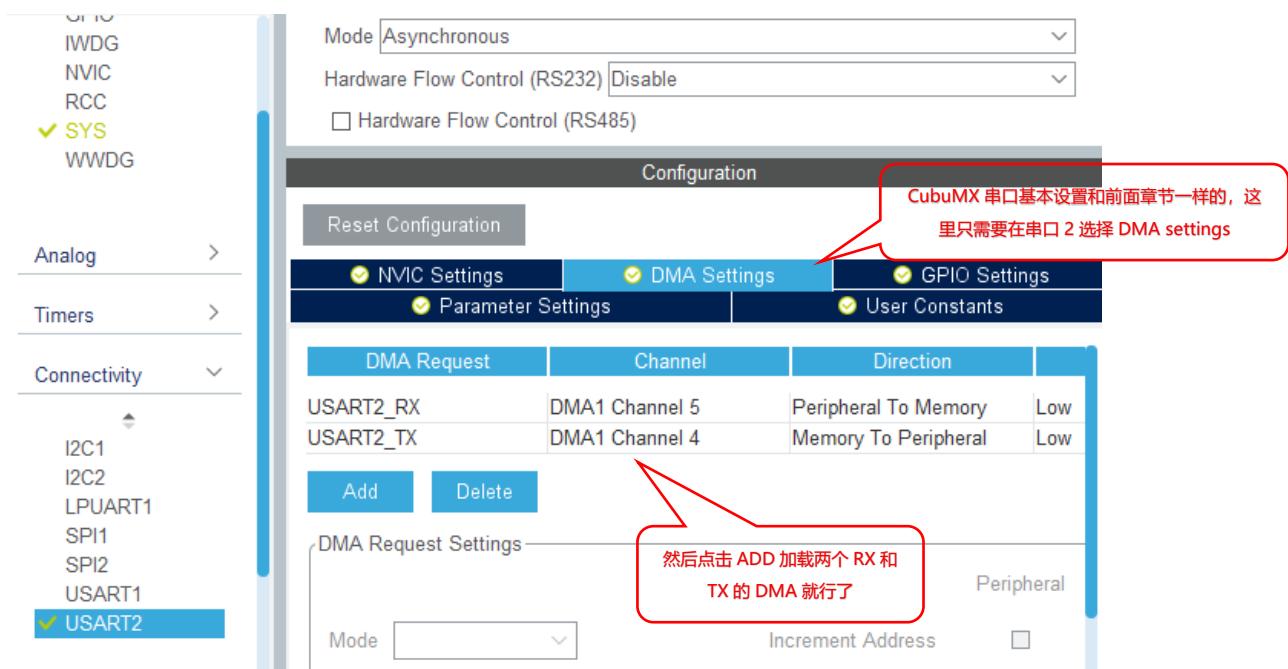
比如我串口调试助手 写入01 02 03 04 05 06 07 08 09 0A 0B 点击发送



这个帧间隔就触发串口
空闲中断

用空闲中断就可以判断 STM32 接收了多少个数据包。这样主程序不用每个字节都去中断处理，等一个数据包满了我要得缓存，或者数据包出现结束，触发空闲中断，我主程序采取处理。_HAL_UART_ENABLE_IT(...) //开启空闲中断

串口+DMA 实现空闲中断



```
_HAL_UART_ENABLE_IT(&huart2,UART_IT_RXNE); //开启串口接收中断
_HAL_UART_ENABLE_IT(&huart2,UART_IT_IDLE); //开启串口 2 空闲接收中断
_HAL_UART_CLEAR_IDLEFLAG(&huart2); //清除串口 2 空闲中断标志位, 防止开机不停的触发空闲中断
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init(); //在串口初始化中添加内容
    HAL_UART_Receive_DMA(&huart2,data,512); //使能 DMA 中断接收, 接收数据最大 512 字节

    while (1)
    {
    }
}

void MX_USART2_UART_Init(void)
{
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 9600;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart2) != HAL_OK)
    {
        Error_Handler();
    }

    _HAL_UART_ENABLE_IT(&huart2,UART_IT_RXNE); //开启串口接收中断
    _HAL_UART_ENABLE_IT(&huart2,UART_IT_IDLE); //开启串口 2 空闲接收中断
    _HAL_UART_CLEAR_IDLEFLAG(&huart2); //清除串口 2 空闲中断标志位
}
```

```

HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t size)
    //使能 DMA 中断接收
huart: 传入串口号
pData: 传入自定义的缓存数组地址, DMA 接收的串口数据放入该数组。
size: DMA 自动搬运串口数据到缓存数组需要多少个, 这个大小根据 pData 自定义数组大小一致
uint8_t data[512]; //DMA接收串口数据的缓存
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART2_UART_Init();
    HAL_UART_Receive_DMA(&huart2, data, 512); //使能DMA中断接收, 接收数据最大512字节

    while (1)
    {
    }

void HAL_UART_RxCpCallBack(UART_HandleTypeDef *huart, uint16_t Len); //函数声明
void USART2_IRQHandler(void)
{
    uint16_t temp = 0;
    uint16_t RxLen = 0;
//    printf("USART2_IRQHandler \r\n"); //除了一般调试, 中断里面少打印 printf, 不然会导致数据包错误
    if(__HAL_UART_GET_FLAG(&huart2, UART_FLAG_IDLE) == SET) //触发空闲中断
    {
        __HAL_UART_CLEAR_IDLEFLAG(&huart2); //清除串口 2 空闲中断标志位
        HAL_UART_DMAStop(&huart2); //关闭 DMA
//        temp = huart2.Instance->ISR; //有些人说__HAL_UART_CLEAR_IDLEFLAG 无法清除空闲中断
//        temp = huart2.Instance->RDR; //必须读一下 SR 和 DR 才能清除, 我这里没发现任何问题
        temp = hdma_usart2_rx.Instance->CNDTR; //获取 DMA 中未传输的数据个数
        RxLen = 512 - temp; //用自定义 buffer 的总大小 减去 未传输个数的大小, 就是接收到实际数据的大小
        HAL_UART_RxCpCallBack(&huart2, RxLen); //执行自定义回调函数
    }
}

HAL_UART_IRQHandler(&huart2); //清除中断标志位
HAL_UART_Receive_DMA(&huart2, data, 512); //重新打开 DMA 接收, 类似前面普通串口的 Receive_IT
}
/*自定义回调函数*/
void HAL_UART_RxCpCallBack(UART_HandleTypeDef *huart, uint16_t Len)
{
    if(huart->Instance == USART2) //证明是串口 2 接收
    {
        if(Len > 0)
        {
            printf("Len = %d\r\n", Len);
//            HAL_UART_Transmit(&huart2, data, Len, 100);
//            //返回数据长了 1ms 绝对只能发 3 个以下数据, 所以用 100ms 超时可以整包发送,
//            //超时时间根据你一次性要发多少数据量来决定
        }
        else
        {
        }
        memset(data, 0x00, sizeof(data)); //数据清 0, 准备接收下一次新数据
    }
    //在这里可以制定一个标志位告诉主程序, 可以处理串口的一帧数据了
}

```

记住 `hdma_usart2_rx` 需要全局声明一下才能在其它 C 文件调用

在 DMA.h 文件里面声明

```
UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;
DMA_HandleTypeDef hdma_usart2_tx;
```

uart.c 文件

```
/* Includes -----
#include "main.h"

extern DMA_HandleTypeDef hdma_usart2_rx;
extern DMA_HandleTypeDef hdma_usart2_tx;
```

DMA.h 文件

通讯端口 串口设置 显示 发送 多字符串 小工具 帮助 联系作者 PCB打样

```
[20:37:06.316]发->◇□"3DUfw帧口
[20:37:06.360]收-<◆Len = 9
[20:37:06.607]发->◇□"3DUfw帧口
[20:37:06.650]收-<◆Len = 9
[20:37:07.101]发->◇□"3DUfw帧口
[20:37:07.144]收-<◆Len = 9
[20:37:07.477]发->◇□"3DUfw帧口
[20:37:07.520]收-<◆Len = 9
[20:37:07.873]发->◇□"3DUfw帧口
[20:37:07.916]收-<◆Len = 9
```



测试数据发送的数量正确

```
/*自定义回调函数*/
void HAL_UART_RxCpCallBack(UART_HandleTypeDef *huart, uint16_t Len)
{
    if(huart->Instance == USART2) //证明是串口2接收
    {
        if(Len > 0)
        {
            //printf("Len = %d\r\n", Len);
            HAL_UART_Transmit(&huart2, data, Len, 100);
            //返回数据长了1ms绝对只能发3个以下数据，所以用100ms超时可以整包发送,
            //超时时间根据你一次性要发多少数据量来决定
        }
        else
        {
            }
        memset(data, 0x00, sizeof(data)); //数据清0，准备接收下一次新数据
    }
}
```

关闭数量打印，看看 PC 发给我
的数据，我接收方式是否正确



50ms 间隔发送，PC 发送到 STM32，STM32 接收数据正确。如果低于 50ms 就会出现 PC 发送了几次 STM32 才返回一次，可能是波特率太低造成的。但是我测试发现可能是电脑接收反应太慢造成的。

重点提醒，在和 PC 串口助手测试串口收发的时候谨记要少使用 printf，不然串口接收会出现重包现象，比如串口接收的是 13 字节的数据包，很有可能变成 24 字节，串口缓存溢出。根据测试不一定完全是 printf 的 BUG，看下面介绍。

串口+DMA 空闲中断 HAL_UART_DMAStop(&huart2); //关闭 DMA 库函数 BUG，一定要注意

```
void USART2_IRQHandler(void)//bug 程序
{
    if(__HAL_UART_GET_FLAG(&huart2,UART_FLAG_IDLE) == SET ) //触发空闲中断
    {
        __HAL_UART_CLEAR_IDLEFLAG(&huart2);//清除串口 2 空闲中断标志位
        HAL_UART_DMAStop(&huart2); //关闭 DMA ，这个函数在串口频繁发送就会出问题
        temp = hdma_usart2_rx.Instance->CNDTR; //获取 DMA 中未传输的数据个数
        RxLen = 512 - temp; //用自定义 buffer 的总大小 减去 未传输个数的大小，就是接收到实际数据的大小
    }
    HAL_UART_IRQHandler(&huart2); //清除中断标志位
    HAL_UART_Receive_DMA(&huart2,data,512); //重新打开 DMA 接收，类似前面普通串口的 Receive_IT
}

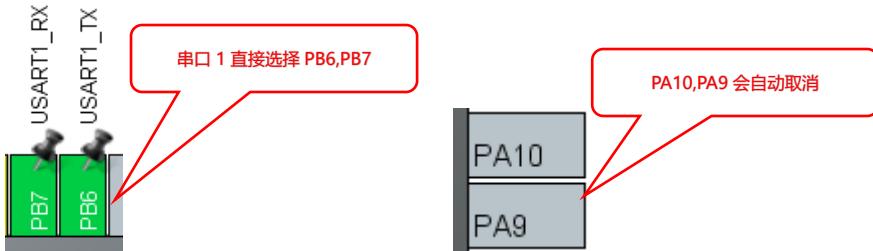
void USART2_IRQHandler(void) //修改如下
{
    if(__HAL_UART_GET_FLAG(&huart2,UART_FLAG_IDLE) == SET ) //触发空闲中断
    {
        __HAL_UART_CLEAR_IDLEFLAG(&huart2);//清除串口 2 空闲中断标志位
        /*串口空闲接收中断不能直接使用HAL_UART_DMAStop(huart);关闭 DMA*****/
        CLEAR_BIT(huart2.Instance->CR3, USART_CR3_DMAR);
        HAL_DMA_Abort(huart.hdmarx);
        CLEAR_BIT(huart.Instance->CR1, (USART_CR1_RXNEIE | USART_CR1_PEIE));
        CLEAR_BIT(huart.Instance->CR3, USART_CR3_EIE);
        /* At end of Rx process, restore huart->RxState to Ready */
        huart.RxState = HAL_UART_STATE_READY;
        /******一定是改成上面这种寄存器方式才能让串口接收数据不重帧*****/
        temp = hdma_usart2_rx.Instance->CNDTR; //获取 DMA 中未传输的数据个数
        RxLen = 512 - temp; //用自定义 buffer 的总大小 减去 未传输个数的大小，就是接收到实际数据的大小
    }
    HAL_UART_IRQHandler(&huart2); //清除中断标志位
    HAL_UART_Receive_DMA(&huart2,data,512); //重新打开 DMA 接收，类似前面普通串口的 Receive_IT
}
```

如果不使用系统默认引脚做串口，而是使用复用引脚做串口，忘记在 CubeMX 设置，那么就要修改库函数

USART1

如果在 CubeMX 选择了串口 1，默认是 PA9,PA10

如果你想 PB7 和 PB6 做串口 1，你就在 CubeMX 直接选择 PB7,PB6 做串口就是了。



如果已经设置了 PA9,PA10 做串口 1，工程项目已经建立完成，我想这时候改成 PB6,PB7 怎么办呢？

MX_USART1_UART_Init(); //串口1初始化 先进入串口初始化函数

主要不是修改串口初始化函数，而是修改 usart.c 里面的 MspInit 函数

```
void HAL_UART_MspInit(UART_HandleTypeDef* uartHandle)
{
    GPIO_InitStruct.Pin = GPIO_PIN_6; //PA9改成PB6 TX
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF0_USART1; //GPIO_AF4_USART1改为GPIO_AF0_USART1
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = GPIO_PIN_7; //PA10改为PB7 RX
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    GPIO_InitStruct.Alternate = GPIO_AF0_USART1; //GPIO_AF4_USART1改为GPIO_AF0_USART1
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

把管教号 GPIO_InitStruct.Pin 进行修改

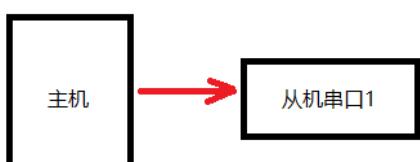
然后将 GPIO_InitStruct.Alternate = GPIO_AF4_USART1 引脚复用口改成 PB6,PB7 的 GPIO_AF0_USART1 引脚复用口。

STM32L 串口 1 开机启动，马上用串口 1 接收发送数据会出现死机现象(注意)

有可能串口 2 也会出现这种，但是串口 2 是手动电脑发送接收，没有暴露出来

STM32F072 系列也要注意

上电瞬间，也许从机串口1还没有初始化好，主机突然向从机串口1发数据，从机串口1回数据造成死机



```

void USART1_IRQHandler(void) //串口1接受数据
{
    HAL_UART_IRQHandler(&huart1);
    HAL_UART_Receive_DMA(&huart1, UART1_parmer.USART_RXdata, USART_BUFF_LEN); //重新打开DMA接收, 类似前面普通串口的Receive_IT
    HAL_UART_Transmit(&huart1, (uint8_t *)sendBuffer, sizeof(sendBuffer), 100); //发送回主机
}

```

这种操作开机一瞬间是不行的, 所以开机最好延时几百毫秒在这样操作



开机瞬间 串口1中断收到莫名其妙的1个字节,
导致主函数串口1接受处理函数执行, 马上回发数据给主机

这种也会导致系统死机, 最好的方法就是让判断len>2 不执行开机瞬间的串口回发函数HAL_UART_Transmit

串口 1 或串口 2 如果接了另外一个板子的串口, 开机瞬间串口 1/2 中断确实会接收到错误码, 但是开机死机绝对不是串口 1 或者串口 2 接收错误码造成的

HardFault_Handler () 问题

系统开机死机首先要确认 stm32l0xx_it.c 里面的 HardFault_Handler 是否被中断死循环执行

```

void HardFault_Handler (void)
{
    /* USER CODE BEGIN HardFault_IRQn_0 */
    /* USER CODE END HardFault_IRQn_0 */
    while (1)
    {
        /* USER CODE BEGIN W1_HardFault_IRQn_0 */
        /* USER CODE END W1_HardFault_IRQn_0 */
    }
}

```

如果开机系统一直在 HardFault_Handler 里面死循环，证明有如下情况：

数组操作越界

内存溢出访问越界

堆栈溢出程序跑飞

中断处理错误

```
Stack_Size      EQU      0x1000
Stack_Mem       AREA     STACK, NOINIT, READWRITE, ALIGN=3
__initial_sp    SPACE    Stack_Size
```

我这里是数组或全局变量超出栈区范围，将栈区从 0x800(2048)修改成 0x1000(4096)搞定。

ADC 初始化造成串口中断无法接收，重大问题

```
status = ADC_Init_driver(3,4,10,11); //ADC通道初始化 //如果采集内部电压，只能单独初始化一个通道
USART1_USART2_Init(); //初始化串口1 2
GPIO_Init(); //初始化ADG709 IO口
set_EXuart_chn(3); //选择U2TX4 U2RX4
HAL_UART_Receive_IT(&huart2, &U2RXdata, 1); //一定要先执行一次接收，才能实现中断接收

if(status == 1) //ADC初始化必须放在串口初始化之前，不然会导致串口无法中断接收，尤其是我这里是移植的ADC
    printf("parmer error\r\n");
else if(status == 2)
    printf("parmer no match\r\n");
else if(status == 3)
    printf("invalid channel\r\n");
else
    printf("ADC Init success\r\n");
```

必须 ADC 先初始化，串口再初始化。

因为我是 HAL 库生成串口后，移植的串口到现有的 ADC 工程，所以出现了开机串口接收一下，然后串口就无法再次接受了。所以初始化顺序需要调整。

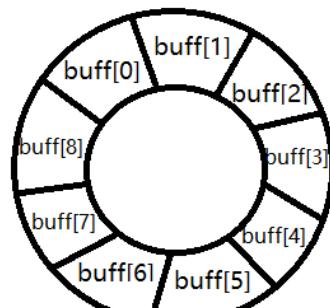
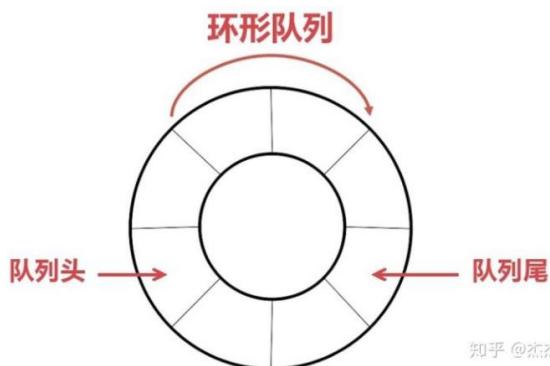
ADC_Init_driver(3,4,10,11); //ADC 通道初始化 ADC 初始化在前

USART1_USART2_Init(); //初始化串口 1 2 串口初始化在后

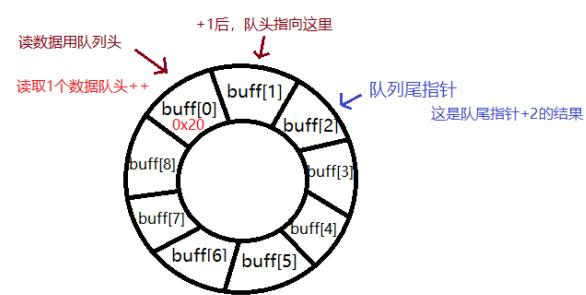
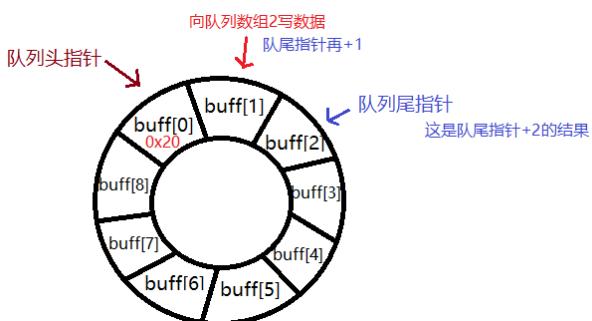
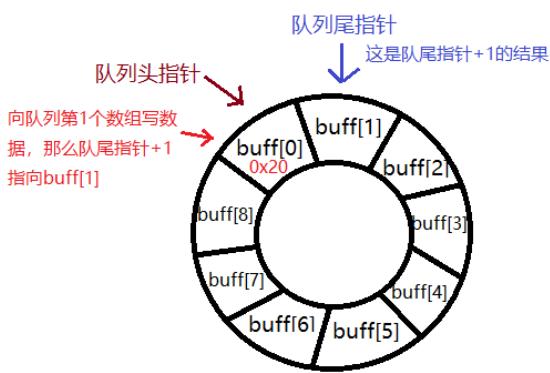
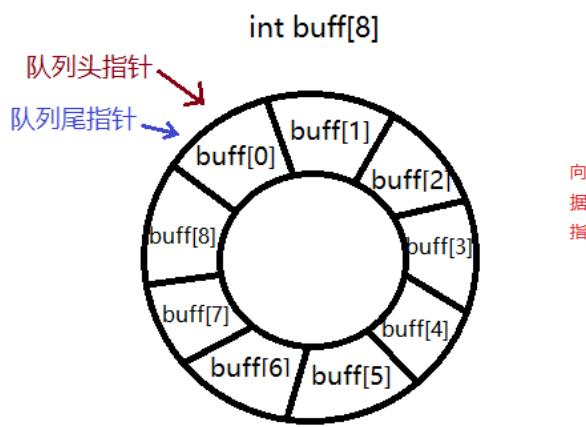
按照这样的顺序初始化自己移植的 ADC 和串口就可以了

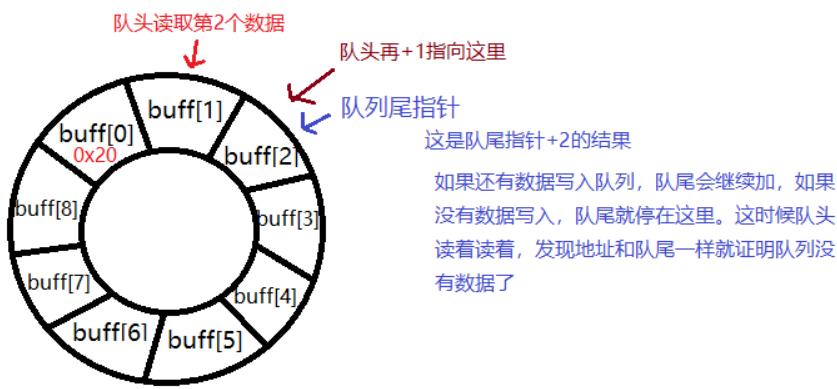
为了防止串口数据丢包，在 DMA 基础上，采用环形队列做缓存

int buff[8]

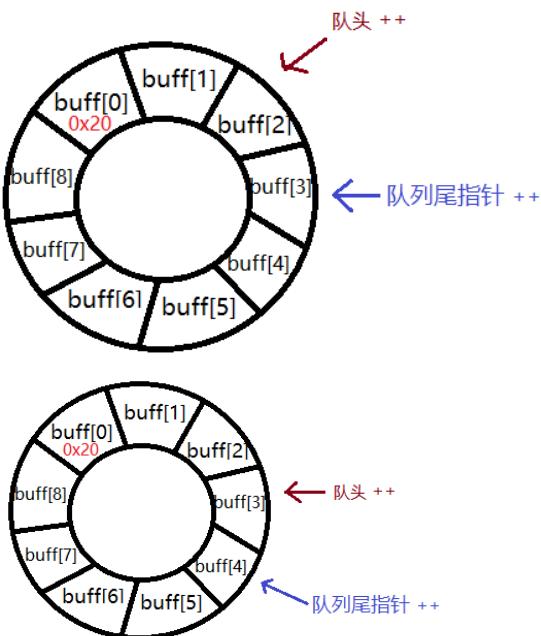


其实环形队列就是把数组做成一个环，用指针不停循环的读8个数组的数据或者写数据

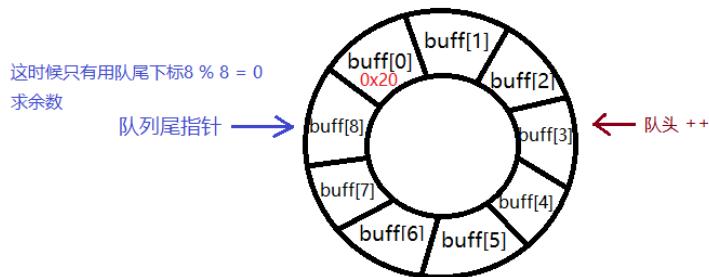


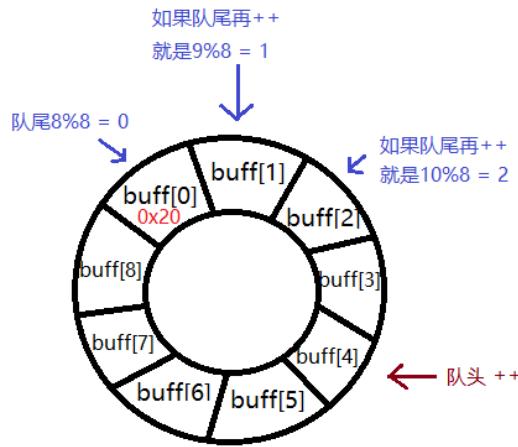


如果串口一直发数据，接收队列的队尾会一直加



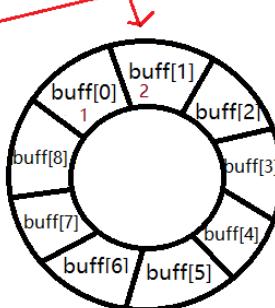
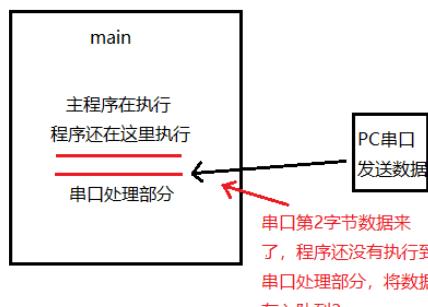
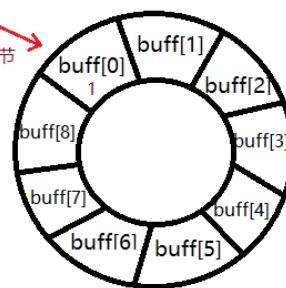
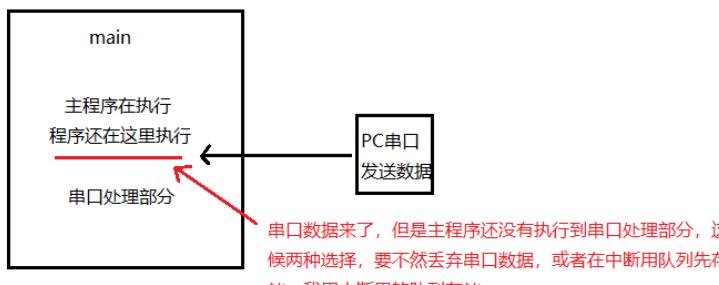
如果队列尾旋转一圈指向 buff[8]地址了，如何又回到 buff[0]呢？

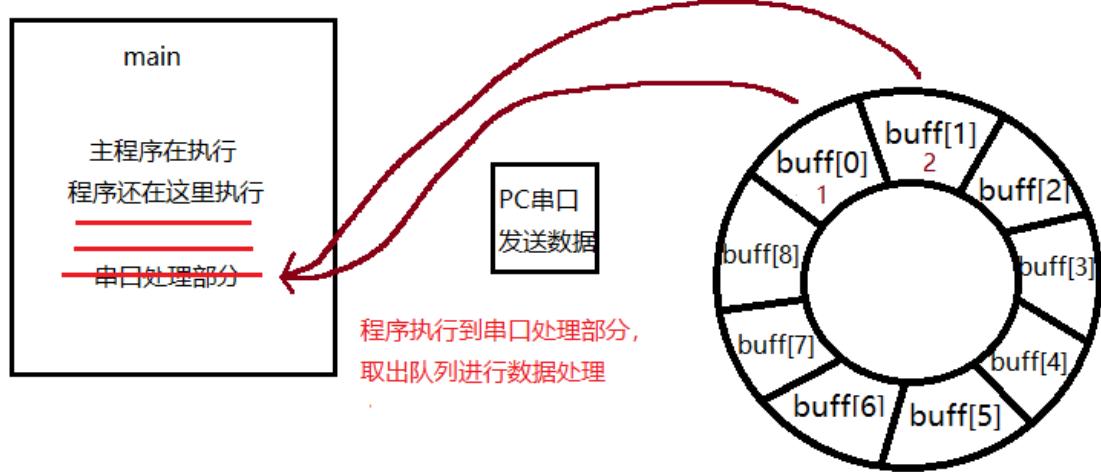




队头一样的旋转原理

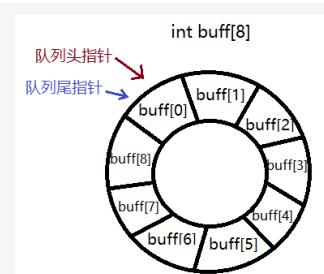
串口使用队列的好处





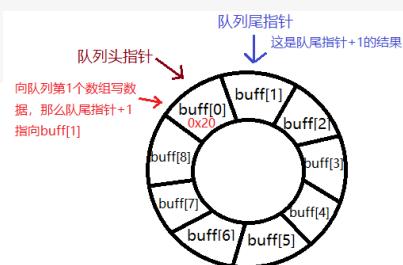
C 语言实现串口队列缓存

```
void RingBuff_Init(void)
{
    //初始化相关信息
    ringBuff.Head = 0;
    ringBuff.Tail = 0;
    ringBuff.Length = 0;
}
```



// 往环形缓冲区写入 u8 类型的数据

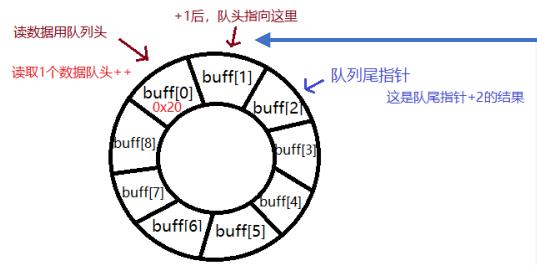
```
u8 Write_RingBuff(u8 data)
{
    if(ringBuff.Length >= RINGBUFF_LEN) //判断缓冲区是否已满
    {
        return FALSE;
    }
    ringBuff.Ring_Buff[ringBuff.Tail]=data;
    ringBuff.Tail = (ringBuff.Tail+1)%RINGBUFF_LEN; //防止越界非法访问
    ringBuff.Length++;
    return TRUE;
}
```



```

u8 Read_RingBuff(u8 *rData)
{
    if(ringBuff.Length == 0) //判断非空
    {
        return FALSE;
    }
    *rData = ringBuff.Ring_Buff[ringBuff.Head]; //先进先出 FIFO，从缓冲区头出
    ringBuff.Head = (ringBuff.Head+1)%RINGBUFF_LEN; //防止越界非法访问
    ringBuff.Length--;
    return TRUE;
}

```

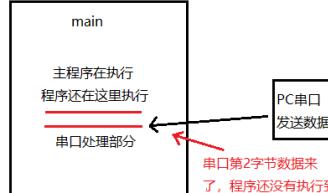


```

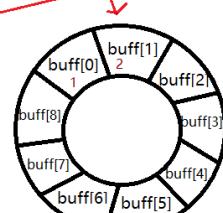
void USART1_IRQHandler(void)
{
    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET) //接收中断
    {
        USART_ClearITPendingBit(USART1, USART_IT_RXNE); //清楚标志位
        Write_RingBuff(USART_ReceiveData(USART1)); //读取接收到的数据
    }
}

```

`ringBuff.Length++;`
 该变量是为了让用户指定队列是否为空，因为向队列写数据
`ringBuff.Length+1`
 向队列读数据 `ringBuff.Length-1`
 所以只有 `ringBuff.Length == 0`
 表示队列数据被读完，没有新的数据
 向队列写入，队列为空。

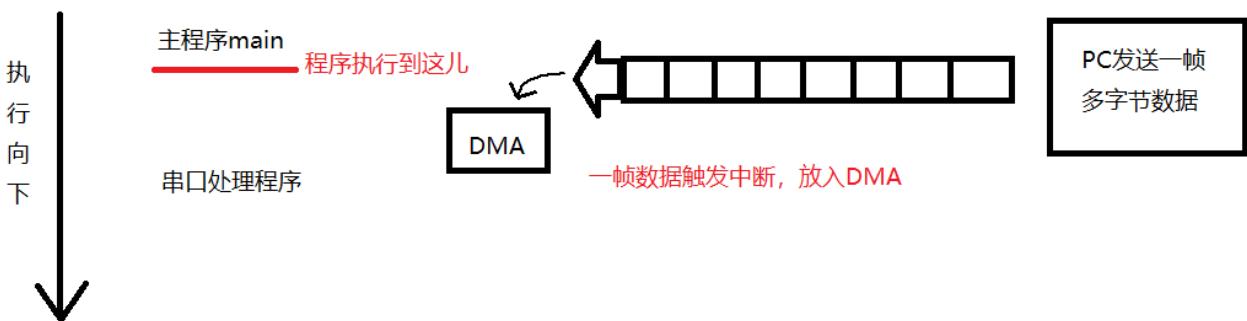


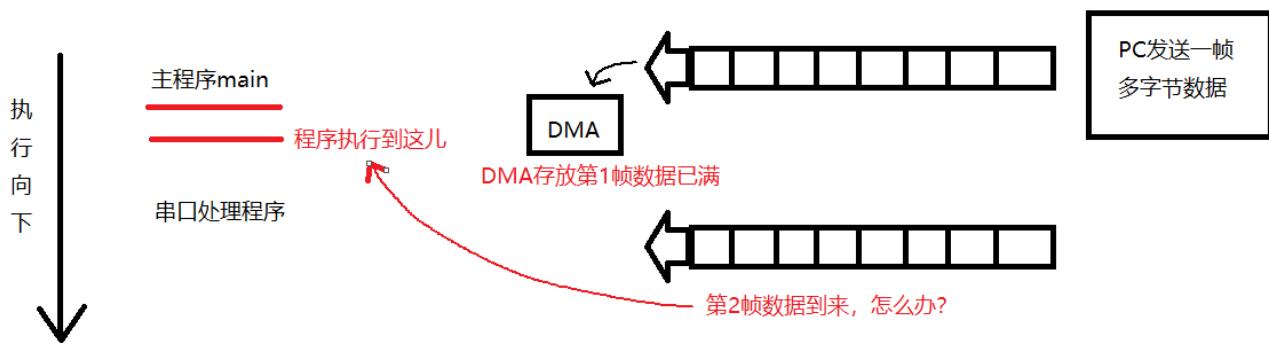
串口第2字节数据来了，程序还没有执行到串口处理部分，将数据存入队列2



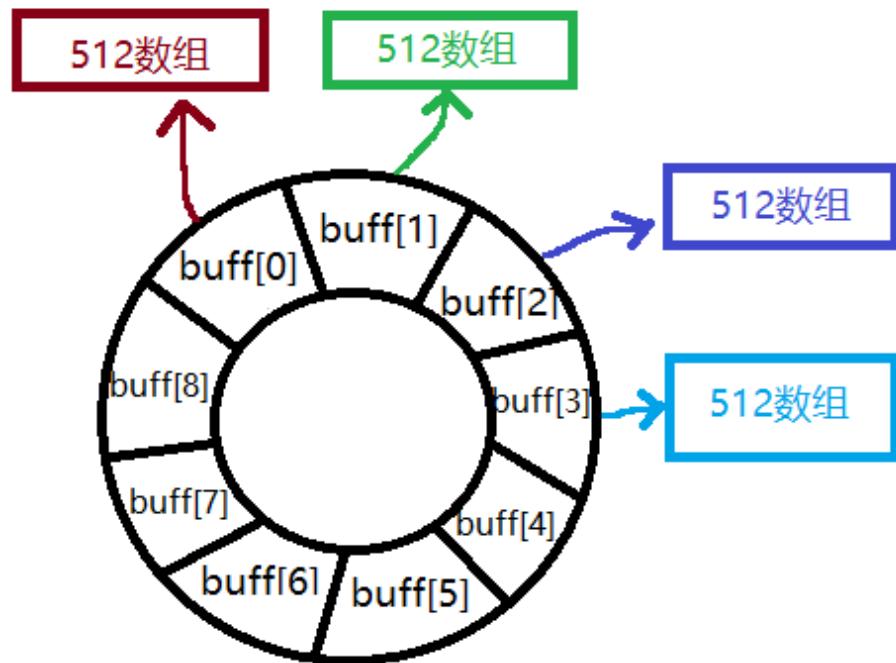
其实很多单片机里面有 DMA 已经解决了串口单个字节缓存问题

但是如果串口是一帧数据一帧数据发送过来，就算我有空闲中断，也无法全部接受得完，尤其是每一帧数据时间间隔很短，毫秒(ms)以下。





不可能把第 2 帧数据丢弃了撒，所以可以建立个环形缓冲区，缓冲区每一个数组都是二维数组



环形缓冲区里面每个数组指向一个很大的数组，这就是二维数组

```
#define USART2_Buff_LEN 512 //串口缓存大小
#define Ring_Buff_Node 8    //环形缓冲区节点数

typedef struct
{
    uint8_t Buffer[USART2_Buff_LEN]; //每个节点 buffer 的大小
    uint8_t RxLen;                //每个节点接受本次串口帧长度
}Rx_Buff;

/*环形缓冲区结构体*/
typedef struct
{
    uint16_t head;               //环形队列头
    uint16_t Tail;               //环形队列尾
    uint16_t Length;              //环形缓冲区节点，表示自己操作的节点号
    Rx_Buff Ring_Buff[Ring_Buff_Node]; //环形缓冲区节点数

}RingBuff_t;

RingBuff_t ringBuff; //创建一个环形缓冲区
```

```

void USART2_IRQHandler(void)
{
    uint16_t temp = 0;
    if(__HAL_UART_GET_FLAG(&huart2,UART_FLAG_IDLE) == SET ) //触发空闲中断
    {
        __HAL_UART_CLEAR_IDLEFLAG(&huart2); //清除串口 2 空闲中断标志位
        HAL_UART_DMAStop(&huart2); //关闭 DMA
        temp = hdma_usart2_rx.Instance->CNDTR; //获取 DMA 中未传输的数据个数
        RxLen = USART2_Buff_LEN - temp; //用自定义 buffer 的总大小 减去 未传输个数的大小，就是接收到实际数据的大小

        HAL_UART_RxCpCallBack(&huart2,RxLen); //执行自定义回调函数
    }

    HAL_UART_IRQHandler(&huart2); //清除中断标志位
    HAL_UART_Receive_DMA(&huart2,data,USART2_Buff_LEN); //重新打开 DMA 接收，类似前面普通串口的 Receive_IT
}

void HAL_UART_RxCpCallBack(UART_HandleTypeDef *huart,uint16_t len)
{
    //printf("RxCpCallBack\r\n");
    if(huart->Instance == USART2)
    {
        //printf("len = %d\r\n",len);
        if(len > 0)
        {
            Write_RingBuff(data,len); //向环形缓冲区写入串口接收的一帧数据
        }
        else
        {
            //memset(data,0x00,sizeof(data));
        }
    }

    //如果在中断直接使用 DMA 发送，这里 memset 不能执行，因为 DMA 发送要晚于中断，
    //所以发送数据会被清 0
    memset(data,0x00,sizeof(data));
}

*****
*环形缓冲区初始化
*将队头,队尾,长度初始化为 0
*****
void RingBuff_Init(void)
{
    ringBuff.head = 0;
    ringBuff.Tail = 0;
    ringBuff.Length = 0;
}

*****
* 向环形缓冲区写入串口一帧的数据
* *data(IN): 传入串口 buffer, 将串口 buffer 接受的数据拷贝进缓冲区
* len(IN): 传入串口要写入的数据长度
*****
int Write_RingBuff(uint8_t *data,uint8_t len)
{
    if(ringBuff.Length >= Ring_Buff_Node) //判断缓冲区是否已满
    {
        return -1; //缓冲区已满，返回错误
    }
    ringBuff.Ring_Buff[ringBuff.Tail].RxLen = len; //本次写入一帧数据的长度

    for(int i = 0; i < len + 1;i++)
    {
        ringBuff.Ring_Buff[ringBuff.Tail].Buffer[i] = data[i];
    }
}

```

```

ringBuff.Tail = (ringBuff.Tail+1)%Ring_Buff_Node; //向缓冲区写入一帧数据后,队尾+1, 防止越界非法访问
ringBuff.Length ++; //缓冲区节点已经写入 1 帧数据, 节点+1
return 0; //写入成功返回 0
}

/*****************/
* 从环形缓冲区读取一个帧的数据
* *rData(Out): 从缓冲区节点读出一帧数据
* *len(Out): 返回读取这一帧数据的长度
/*****************/
int Read_RingBuff(uint8_t *rData,uint8_t *len)
{
    if(ringBuff.Length == 0)//判断非空, 如果节点为空, 证明没有数据
    {
        return -1;
    }

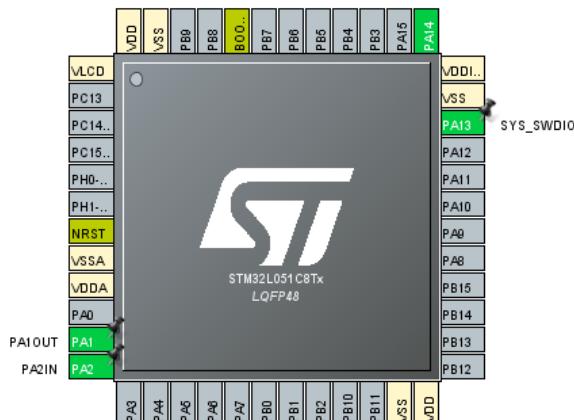
    *len = ringBuff.Ring_Buff[ringBuff.head].RxLen; //读取本帧数据长度返回到用户

    for(int i = 0; i < *len + 1; i++) //读取缓冲区数据到自定义 buffer
    {
        rData[i] = ringBuff.Ring_Buff[ringBuff.head].Buffer[i];//先进先出 FIFO, 从缓冲区头出
    }

    ringBuff.head = (ringBuff.head+1)%Ring_Buff_Node;//防止越界非法访问
    ringBuff.Length--; //读取 1 个节点, 就消除该节点
    return 0; //读取成功返回 0
}

```

GPIO 输入输出(和 F072 操作一样)



选择 PA1 为输出, PA2 为输入

Pinout & Configuration Clock Configuration Pinout

Additional Software

Categories A-Z

System Core

- DMA
- GPIO**
- IWDG
- NVIC
- RCC
- SYS
- WWDG

GPIO Mode and Configuration

Configuration

Group By Peripherals

GPIO SYS

Search Signals Search (Ctrl+F) Show only Modified Pins

Pin ...	Signal o.	GPIO o.	GPIO m.	GPIO P.	Maximu...	Fast M...	User La...	Modified
PA1	n/a	Low	Output ...	Pull-up	Low	n/a	PA1OUT	<input checked="" type="checkbox"/>
PA2	n/a	n/a	Input m...	Pull-up	n/a	n/a	PA2IN	<input checked="" type="checkbox"/>

勾选 PA1 和 PA2

GPIO output level	Low
GPIO mode	Output Push Pull
GPIO Pull-up/Pull-down	Pull-up
Maximum output speed	Low
User Label	PA1OUT

GPIOA1 自己取名为 PA1OUT，设置为推挽输出，上拉输出模式，开机默认输出低点平。

Pin ...	Signal o...	GPIO o...	GPIO m...	GPIO P...	Maximu...	Fast M...	User La...	Modified
PA1	n/a	Low	Output ...	Pull-up	Low	n/a	PA1OUT	<input checked="" type="checkbox"/>
PA2	n/a	n/a	Input m...	Pull-up	n/a	n/a	PA2IN	<input checked="" type="checkbox"/>

PA2 Configuration :

GPIO mode	Input mode
GPIO Pull-up/Pull-down	Pull-up
User Label	PA2IN

GPIOA2 为输入模式，上拉输入，自己取名为 PA2IN

```

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    while (1)
    {
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_SET);
        HAL_Delay(100);
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_1, GPIO_PIN_RESET);
        HAL_Delay(100);
    }
}

void MX_GPIO_Init(void) //GPIO 初始化
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(PA1OUT_GPIO_Port, PA1OUT_Pin, GPIO_PIN_RESET);
    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = PA1OUT_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(PA1OUT_GPIO_Port, &GPIO_InitStruct);
}

```

```

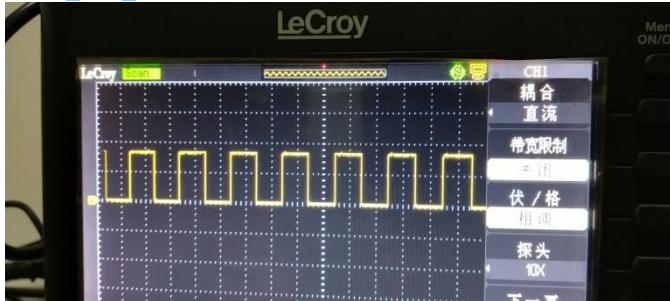
/*Configure GPIO pin : PtPin */
GPIO_InitStruct.Pin = PA2IN_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(PA2IN_GPIO_Port, &GPIO_InitStruct);
}

```

HAL_GPIO_WritePin (GPIO 组, GPIO 单个引脚, 电平状态); //GPIO 输出电平和 F072 操作一样

GPIO_PIN_RESET = 0

GPIO_PIN_SET = 1



IO 口输入高低电平正常

IO 口输入电平检测

GPIO 输入检测，只有用之前的串口来实验

```

#define PA1OUT_Pin GPIO_PIN_1
#define PA1OUT_GPIO_Port GPIOA
#define PA0IN_Pin GPIO_PIN_0 //为了方便我将接受电平输入的 IO 口改成了 PA0
#define PA0IN_GPIO_Port GPIOA

void MX_GPIO_Init(void)
{
    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE();

    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(PA1OUT_GPIO_Port, PA1OUT_Pin, GPIO_PIN_RESET);
    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = PA1OUT_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(PA1OUT_GPIO_Port, &GPIO_InitStruct);

    /*Configure GPIO pin : PtPin */
    GPIO_InitStruct.Pin = PA0IN_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    HAL_GPIO_Init(PA0IN_GPIO_Port, &GPIO_InitStruct);

}

int main(void)
{
    GPIO_PinState state;

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init(); //里面只有配置GPIOA的时钟，串口引脚没有在里面初始化
    MX_USART2_UART_Init(); //串口引脚应该是系统在cubeMX默认初始化的
    HAL_UART_Receive_IT(&huart2, &pdata, 1); //一定要先执行一次接收，才能实现中断接收

    while (1)
    {
        state = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0); //读取IO口电平状态
        if(state == GPIO_PIN_SET)
            printf("PA0 == 1\r\n");
        else
            printf("PA0 == 0\r\n");

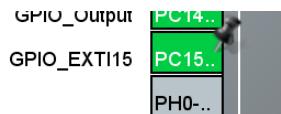
        HAL_Delay(500);
    }
}

```

PA0 == 1
PA0 == 0
PA0 == 1
PA0 == 1
PA0 == 1
PA0 == 0
PA0 == 0
PA0 == 0
PA0 == 1

默认情况 PA0 接收的电平为 1，因为我 PA0 选择的上拉输入，但是我用外部低电平输入，PA0 输出为 0 了，证明 PA0 输入模式成功

STM32L051 引脚外部输入中断使用



如果我要是使用 PC15 引脚中断，必须提前配置管脚为 EXTI

GPIO mode: External Interrupt Mode with Rising edge trigger detection

GPIO Pull-up/Pull-down: Pull-down

Callout text: PC15 引脚，设置上升沿触发中断

Callout text: 引脚内部默认下拉电阻

	Enabled	Pre
Non maskable interrupt	<input checked="" type="checkbox"/>	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0
Pendable request for system service	<input checked="" type="checkbox"/>	0
Time base: System tick timer	<input checked="" type="checkbox"/>	0
Window watchdog interrupt	<input checked="" type="checkbox"/>	3
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0
RTC global interrupt through EXTI lines 17, 19 and 20 and LSE CSS interrupt ...	<input checked="" type="checkbox"/>	2
Flash and EEPROM global interrupt	<input type="checkbox"/>	0
RCC global interrupt	<input type="checkbox"/>	0
EXTI line 4 to 15 interrupts	<input checked="" type="checkbox"/>	0
USART1 global interrupt / USART1 wake-up interrupt through EXTI line 25	<input checked="" type="checkbox"/>	3

Callout text: PC15 中断源在中断线 4~15

Callout text: 生成代码

```

SystemClock_Config();

/* USER CODE BEGIN SysInit */
/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();

/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    HAL_Delay(1000);
}

```

```

void EXTI4_15_IRQHandler(void)
{
    /* USER CODE BEGIN EXTI4_15_IRQn_0 */
    /* USER CODE END EXTI4_15_IRQn_0 */
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_15);
    /* USER CODE BEGIN EXTI4_15_IRQn_1 */
    printf("PC15 IRQ.... \r\n");
    /* USER CODE END EXTI4_15_IRQn_1 */
}

```

Callout text: PC15 中断在 GPIO 中已经初始化

Callout text: PC15 外部中断程序生成在 it.c 中

引脚上升沿测试，没有问题，打印出了 PC15 IRQ...

STM32L051 定时器 2 使用

L051 通用定时器只有 4 个

Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/compare channels	Complementary outputs
TIM2	16-bit	Up, down, up/down	Any integer between 1 and 65536	Yes	4	No
TIM21, TIM22	16-bit	Up, down, up/down	Any integer between 1 and 65536	No	2	No
TIM6	16-bit	Up	Any integer between 1 and 65536	Yes	0	No

16 位可以计数从 0~65535 和 f072 一样

Up 向上计数，从 0 加到 65535
down 向下计数，从 65535 减到 0

计数时间可以用主时钟分频，分频系数为 1~65536

TIM2 支持 4 路 PWM

选择内部时钟，设置的 32M 内部时钟

设置 TIM2 定时参数

设置 Parameter 参数

设置前我有必要讲一下 TIM2 算法

$$\text{Delay} = ((\text{ARR} + 1) * (\text{PSC} + 1)) / \text{Tclk}$$

$\text{Tclk} = 32\text{M}$ (系统时钟)

$\text{PSC} = 3199$ (就是将系统时钟32M/3199 得到分频后的时钟)

$\text{ARR} = 4999$ (就是分频后的时钟不停的++加到4999产生一次中断)

因为 PSC 内部会自动加1，所以这里 3199 分频其实是 3200 分频

$$32000000/3200 = 10000 \text{ (10Khz 脉冲)} 0.0001 \text{ 秒 (100us)} \text{ 产生一次脉冲}$$

$$100\text{us} \text{ 加 } 4999 \text{ 次 } 0.0001 \times 4999 = 0.4999 \text{ 秒}$$

$$\text{Delay} = 500\text{ms} \text{ 也就是 TIM2 500ms 中断一次}$$



Configure the below parameters :

Search (Ctrl+F) (i)

Counter Settings

- Prescaler (PSC - 16 bits val... 3199 分频 3199, 内部加 1 实际是 3200)
- Counter Mode Up 向上计数
- Counter Period (AutoReload... 4999 向上计数到 4999 产生溢出, TIM2 中断)
- Internal Clock Division (CKD) No Division
- auto-reload preload Enable 溢出后自动重装载

Configuration

Reset Configuration 打开 TIM2 中断

User Constants	NVIC Settings	DMA Settings
Parameter Settings		
NVIC Interrupt Table		Enabled
TIM2 global interrupt	<input checked="" type="checkbox"/>	1

IVDDG
NVIC
RCC
SYS
WWDG

Analog >

Timers >

- LPTIM1
- RTC
- TIM2**

NVIC Interrupt Table

	Enabled	Pre
Non maskable interrupt	<input checked="" type="checkbox"/>	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0
Pendable request for system service	<input checked="" type="checkbox"/>	0
Time base: System tick timer	<input checked="" type="checkbox"/>	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0
Flash and EEPROM global interrupt	<input type="checkbox"/>	0
RCC global interrupt	<input type="checkbox"/>	0
TIM2 global interrupt	<input checked="" type="checkbox"/>	1
USART2 global interrupt / USART2 wake-up interrupt through EXTI line 26	<input checked="" type="checkbox"/>	3

Enabled Preemption Priority 1

设置 TIM2 中断优先级为 1

串口打印自行设置，现在生成代码

HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim) //启动定时器

*htim: 传入要启动的定时器号

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_TIM2_Init();
    MX_USART2_UART_Init();

    HAL_TIM_Base_Start_IT(&htim2); //一定要执行启动定时器2

    while (1)
    {
    }
}
```

```

void TIM2_IRQHandler(void)
{
    printf("TIM2....\r\n");
    HAL_TIM_IRQHandler(&htim2);
}

```

500 毫秒定时器中断 打印正常

TIM2....
TIM2....
TIM2....
TIM2....
TIM2....
TIM2....
TIM2....

STM32L0 支持读写内部 FLASH(EEROM)功能，将数据永久保存



STM32L051x6 STM32L051x8

Access line ultra-low-power 32-bit MCU Arm®-based Cortex®-M0+, up to 64 KB Flash, 8 KB SRAM, **2 KB EEPROM**, ADC

数据 EEPROM	0x0808 0000 - 0x0808 07FF	2 KB	-	数据 EEPROM
-----------	---------------------------	------	---	-----------

STM32L051 有 2K 的 EEPROM,但是这个 EEPROM 和 flash 是在同一块存储区地址上的, 所以使用该 EEPROM 必须用读写内部 flash 的方法来实现存储。该 EEPROM 擦写次数**最多 100000** 次所以小心使用。

#define EEPROM_BASE_ADDR 0x08080000 //这是 EEPROM 在 flash 区的起始地址

这个 0x0808 0000 地址以上的 2K EEPROM 不会因为下载程序而被 flash 擦除, 这是块特殊区域。

STM32的EEPROM因为使用的是flash区域的存储方式, 所以是按照4字节偏移地址存储的, 逻辑如下:

32位数据	0x0808 0000	向0x0808 0000 写入1个int, 或者16位数据之后
无法写入, 造成死机	0x0808 0001	我地址+1 然后再写入新的32位数据存储

这是因为EEPROM是
flash存储方式, 地址
不能1个1个的偏移,
必须4字节偏移才行

32位数据	0x0808 0000	向0x0808 0000 写入1个int, 或者16位数据之后
32位数据	0x0808 0004	向0x0808 0004 写入1个int, 写入成功

所以STM32L0 EEPROM存储是4字节地址偏移

HAL_FLASH_Unlock(); //开锁, 写 flash 要开锁之后才能写

HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address, uint32_t Data)
//数据写入内部 FLASH

HAL_FLASH_Lock(); //锁住, 写完 flash 之后要锁住, 数据才能保存在 flash 中

STM32L0 FLASH (eeprom) 基地址是 0x0808 0000 开始

```

int main(void)
{
    uint32_t data;
    HAL_Init();

    SystemClock_Config();

```

```

MX_GPIO_Init();
MX_USART2_UART_Init();
HAL_FLASH_Unlock(); //解锁写
HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, 0x08080000, 0x1A); //将 0x1A 数据写入 0x0808 0000 位置
HAL_FLASH_Lock(); //锁定保存

while (1)
{
}

int main(void)
{
    uint32_t data;
    HAL_Init();

    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART2_UART_Init();

    // HAL_FLASH_Unlock();
    // HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, 0x08080000, 0x1A);
    // HAL_FLASH_Lock();

    while (1)
    {
        data = *(__IO uint32_t *)0x08080000; // 读取刚才存入 0x0808 0000 地址上的数据
        printf("flash data = %x\r\n", data);
        HAL_Delay(500);
    }
}

```

读出正确

我们操作 EEPROM 最好不用上面这种方式，我用专用的写 EEPROM，封装好的 flash 函数

下面进行代码实验

```

/*
Func: EEPROM数据按字(32位)写入
Note: 字当半字用
*/
void EEPROM_Write()
{
    uint32_t data = 0xEEFF9988;

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁
    HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_WORD, 0x08080000, data); //向指定EEPROM地址写32位/16位数据
    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM
}

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_USART1_UART_Init();

    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {

        data_8bit = *(__IO uint8_t *) 0x08080000;
        data_16bit = *(__IO uint16_t *) 0x08080000;
        data_32bit = *(__IO uint32_t *) 0x08080000;

        printf("data_8bit = %x\r\n", data_8bit);
        printf("data_16bit = %x\r\n", data_16bit);
        printf("data_32bit = %x\r\n", data_32bit);
        HAL_Delay(1000);
    }
}

```

根据串口打印，数据貌似正常写入了，但是我们要给单片机断电，然后上电再来确定数据是否被正常写入

data_8bit = 88
data_16bit = 9988
data_32bit = eeff9988

```

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_USART1_UART_Init();

//    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {

        data_8bit = *(__IO uint8_t *) 0x08080000;
        data_16bit = *(__IO uint16_t *) 0x08080000;
        data_32bit = *(__IO uint32_t *) 0x08080000;

        printf("data_8bit = %x\r\n", data_8bit);
        printf("data_16bit = %x\r\n", data_16bit);
        printf("data_32bit = %x\r\n", data_32bit);
        HAL_Delay(1000);
    }
}

```

取消再次写入数据函数，然后将程序下载进单片机，单片机重新断电，再上电

数据还是正常打印，证明 eeff9988 数据确实成功的写入 EEPROM 0x0808 0000 地址保存成功了
 data_8bit = 88 是按照字节取，就是取 0x0808 0000 地址上 32 位数据的最低 8 位
 data_16bit 是按照 16 位取，就是取 0x0808 0000 地址上 32 位数据的低 16 位
 data_32bit 就是取 0x0808 0000 地址上 完整的 32 位数据

下面尝试地址增加 1 的 BUG

```

/*
  Func: EEPROM数据按字(32位)写入
  Note: 字当半字用
*/
void EEPROM_Write()
{
//    uint32_t data = 0xEEFF9988; // 0x0808 0000地址上的数据保留
//    uint32_t data = 0xAABBCCDD; //0x0808 0000地址+1 0x0808 0001

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁
    HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_WORD, 0x08080001, data);
//我们尝试在增加1之后的地址上写新的数据

    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM

}

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_USART1_UART_Init();

    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {

        /*获取之前存入0x0808 0000的数据，看看向新地址0x0808 0001写数据，会不会擦除掉以前0x0808 0000上的数据*/
        data_8bit = *(__IO uint8_t *) 0x08080000;
        data_16bit = *(__IO uint16_t *) 0x08080000;
        data_32bit = *(__IO uint32_t *) 0x08080000;
    }
}

```

地址+1 写入新数据

写入数据，多半在这里死机

```

printf("data_8bit = %x\r\n",data_8bit);
printf("data_16bit = %x\r\n",data_16bit);
printf("data_32bit = %x\r\n",data_32bit);

/*获取存入0x0808 0001上的数据*/
data_8bit = *(__IO uint8_t *) (0x08080000 + 1);
data_16bit = *(__IO uint16_t *) (0x08080000 + 1);
data_32bit = *(__IO uint32_t *) (0x08080000 + 1); //读取新数据

printf("data_8bit +1 = %x\r\n",data_8bit);
printf("data_16bit +1 = %x\r\n",data_16bit);
printf("data_32bit +1 = %x\r\n",data_32bit);

HAL_Delay(1000);
}

```

开机就直接死机，证明 EEPROM 地址不能+1 来偏移

符合本章节最先说明的内容

32位数据

0x0808 0000

向0x0808 0000 写入1个int，或者16位数据之后

无法写入，造成死机

0x0808 0001

我地址+1 然后再写入新的32位数据存储

这是因为EEPROM是
flash存储方式，地址
不能1个1个的偏移，
必须4字节偏移才行

32位数据

0x0808 0000

向0x0808 0000 写入1个int，或者16位数据之后

32位数据

0x0808 0004

向0x0808 0004 写入1个int，写入成功

所以STM32L0 EEPROM存储是4字节地址偏移

不知道EEPROM必须4字节偏移是不是和写EEPROM数据函数的
FLASH_TYPEPROGRAMDATA_WORD参数有关，后面再测试验证

```

/*
Func: EEPROM数据按字(32位)写入
Note: 字当半字用
*/
void EEPROM_Write()
{
    uint32_t data = 0xEEFF9988; // 0x0808 0000地址上的数据保留
    uint32_t data = 0xAABBCCDD; //0x0808 0000地址+4 0x0808 0004

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁

    HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_WORD, 0x08080004, data);
    //我们再次尝试在增加4之后的地址上写新的数据

    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM
}

```

EEPROM 修改为4
字节偏移

main()函数

```
EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取之前存入0x0808 0000的数据，看看向新地址0x0808 0004写数据，会不会擦除掉以前0x0808 0000上的数据*/
    data_8bit = *(__IO uint8_t *) 0x08080000;
    data_16bit = *(__IO uint16_t *) 0x08080000;
    data_32bit = *(__IO uint32_t *) 0x08080000;

    printf("data_8bit = %x\r\n", data_8bit);
    printf("data_16bit = %x\r\n", data_16bit);
    printf("data_32bit = %x\r\n", data_32bit);

    /*获取存入0x0808 0004上的数据*/
    data_8bit = *(__IO uint8_t *) (0x08080000 + 4);
    data_16bit = *(__IO uint16_t *) (0x08080000 + 4);
    data_32bit = *(__IO uint32_t *) (0x08080000 + 4);

    printf("data_8bit +4 = %x\r\n", data_8bit);
    printf("data_16bit +4 = %x\r\n", data_16bit);
    printf("data_32bit +4 = %x\r\n", data_32bit);

    data_8bit = 88
    data_16bit = 9988
    data_32bit = eeff9988
    data_8bit +4 = dd
    data_16bit +4 = cddd
    data_32bit +4 = aabbccdd
}
```

新存储的数据都按
照4字节读取

以前存放在 0x0808 0000 的数据也没有被覆盖掉

新写入的数据读取正常

证明 EEPROM 虽然使用的是 flash 区域，但是在保留其余地址写入数据的时候，做了处理，不会再你写新数据的时候，直接擦除整个 EEPROM 2K 的地址。

下面 STM32 断电再上电，看看新数据是否保留

```
// EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取之前存入0x0808 0000的数据，看看向新地址0x0808 0004写数据，会不会擦除掉以前0x0808 0000上的数据*/
    data_8bit = *(__IO uint8_t *) 0x08080000;
    data_16bit = *(__IO uint16_t *) 0x08080000;
    data_32bit = *(__IO uint32_t *) 0x08080000;

    printf("data_8bit = %x\r\n", data_8bit);
    printf("data_16bit = %x\r\n", data_16bit);
    printf("data_32bit = %x\r\n", data_32bit);

    /*获取存入0x0808 0004上的数据*/
    data_8bit = *(__IO uint8_t *) (0x08080000 + 4);
    data_16bit = *(__IO uint16_t *) (0x08080000 + 4);
    data_32bit = *(__IO uint32_t *) (0x08080000 + 4);

    printf("data_8bit +4 = %x\r\n", data_8bit);
    printf("data_16bit +4 = %x\r\n", data_16bit);
    printf("data_32bit +4 = %x\r\n", data_32bit);
}
```

取消 EEPROM 写入，重新下载进 STM32，这样 STM32 就只能读取 EEPROM 数据了

测试结果和上面一样，读取数据正常。

FLASH_TYPEPROGRAMDATA_BYTE 单字节写入 EEPROM 参数测试

```
/*
* Func: EEPROM数据按字节(8位)写入
* Note: 字当半字用
*/
void EEPROM_Write()
{
    uint8_t data = 0x11; // 0x0808 0000地址上写数据

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁
    HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_BYTE, 0x08080000, data); //起始地址写入1字节数据
    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM
}
```

用 FLASH_TYPEPROGRAMDATA_BYTE 写入 1 字节数据

```

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_USART1_UART_Init();

    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {
        /*获取EEPROM起始地址存入的1字节数据*/
        data_8bit = *(__IO uint8_t *) 0x08080000;
        printf("data_8bit = %x\r\n", data_8bit);

        HAL_Delay(1000);
    }
}

```

data_8bit = 11
data_8bit = 11
data_8bit = 11

单字节数据写入成功

取消写入 EEPROM 数据, STM32 断电重读

```

// EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取EEPROM起始地址存入的1字节数据*/
    data_8bit = *(__IO uint8_t *) 0x08080000;

    printf("data_8bit = %x\r\n", data_8bit);

    HAL_Delay(1000);
}

```

data_8bit = 11
data_8bit = 11
data_8bit = 11

单字节数据存储成功

下面地址+1 来存储数据, 看看会不会出现 FLASH_TYPEPROGRAMDATA_WORD 参数方式写入时造成的死机

```

void EEPROM_Write()
{
    // uint8_t data = 0x11; // 0x0808 0000 地址上数据是11
    uint8_t data = 0x22; //0x0808 0001地址写数据 0x0808 0001

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁
    HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_BYTE, 0x08080000+1, data); //起始地址+1写入1字节数据
    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁, 数据才能存入EEPROM
}

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_USART1_UART_Init();

    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {
        /*获取EEPROM起始地址存入的1字节数据*/
        data_8bit = *(__IO uint8_t *) 0x08080000;
        printf("data_8bit = %x\r\n", data_8bit);

        data_8bit = *(__IO uint8_t *) 0x08080001; //读取地址+1的数据
        printf("data_8bit + 1 = %x\r\n", data_8bit);

        HAL_Delay(1000);
    }
}

```

单字节数据写入偏移+1 的地址成功

data_8bit = 11
data_8bit + 1 = 22
data_8bit = 11
data_8bit + 1 = 22

证明前面地址偏移必须+4，如果地址偏移必须+1会造成死机的理论是错误的，只是因为没有选择正确的写入 EEPROM 参数。

FLASH_TYPEPROGRAMDATA_WORD 是 32 位写入 EEPROM，所以地址偏移必须 +4

FLASH_TYPEPROGRAMDATA_BYTE 是 8 位写入 EEPROM，所以地址偏移是+1 的方式

下面测试单字节，STM32 掉电重读，数据是否还保存在 EEPROM

```
// EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取EEPROM起始地址存入的1字节数据*/
    data_8bit = *(__IO uint8_t *) 0x08080000;

    printf("data_8bit = %x\r\n", data_8bit);
    data_8bit = *(__IO uint8_t *) 0x08080001; //读取地址+1的数据
    printf("data_8bit + 1 = %x\r\n", data_8bit);

    HAL_Delay(1000);
}
```

data_8bit = 11
data_8bit + 1 = 22
data_8bit = 11
data_8bit + 1 = 22

STM32 掉电重读，数据还是保存在 EEPROM。运行正确

FLASH_TYPEPROGRAMDATA_HALFWORD EEPROM 16 位数据写入，MODBUS 用得比较多

```
/*
Func: EEPROM数据按2字节(16位)写入
Note:
-----*/
void EEPROM_Write()
{
    uint16_t data = 0xAABB; // 0x0808 0000 地址上数据是AABB

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁

    HAL_FLASHEx_DATAEEPROM_Program( FLASH_TYPEPROGRAMDATA_HALFWORD, 0x08080000, data );
    //起始地址写入2字节数据

    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM
}

EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取EEPROM起始地址存入的1字节数据*/
    data_16bit = *(__IO uint16_t *) 0x08080000;

    printf("data_16bit = %x\r\n", data_16bit);

    HAL_Delay(1000);
}
```

data_16bit = aabb
data_16bit = aabb
data_16bit = aabb 16 位写入数据正常

地址偏移+2，写入 16 位数据，测试是否死机

```
/*
Func: EEPROM数据按2字节(16位)写入
Note:
-----*/
void EEPROM_Write()
{
//    uint16_t data = 0xAABB; // 0x0808 0000 地址上数据是AABB
    uint16_t data = 0xCCDD; // 0x0808 0002 地址上数据是CCDD

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁

    HAL_FLASHEx_DATAEEPROM_Program( FLASH_TYPEPROGRAMDATA_HALFWORD, 0x08080002, data );
    //起始地址写入2字节数据

    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁，数据才能存入EEPROM
}
```

```

EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取EEPROM起始地址存入的2字节数据*/
    data_16bit = *(__IO uint16_t *) 0x08080000;
    printf("data_16bit = %x\r\n",data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080002;
    printf("data_16bit = %x\r\n",data_16bit);

    HAL_Delay(1000);
}

```

data_16bit = aabb
data_16bit = cddd
data_16bit = aabb
data_16bit = cddd 证明地址+2 可以成功写入

STM32 掉电重读测试

```

// EEPROM_Write(); //向EEPROM写数据
while (1)
{
    /*获取EEPROM起始地址存入的2字节数据*/
    data_16bit = *(__IO uint16_t *) 0x08080000;
    printf("data_16bit = %x\r\n",data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080002;
    printf("data_16bit = %x\r\n",data_16bit);

    HAL_Delay(1000);
}

```

data_16bit = aabb
data_16bit = cddd
data_16bit = aabb
data_16bit = cddd

数据读取正常，证明测试成功

所以在使用 **HAL_FLASHEx_DATAEEPROM_Program**(写数据格式, 地址, 数据), 向 EEPROM 写数据的时候, 到底地址每次能偏移多少, 是根据写数据格式参数来的。

如何向 EEPROM 连续写入数据呢? 比如数组存储 EEPROM

```

Func: EEPROM数据按2字节(16位)写入
Note:
void EEPROM_Write()
{
    uint16_t MODBUSarray[12] = {0x0101,
                                0x0202,
                                0x0303,
                                0x0404,
                                0x0505,
                                0x0606,
                                0x0707,
                                0x0808,
                                0x0909,
                                0xa0a0,
                                0xb0b0,
                                0xc0c0};

    HAL_FLASHEx_DATAEEPROM_Unlock(); //写EEPROM之前必须先解锁
    uint16_t i2 = 0;
    for(uint16_t i = 0; i<12; i++)
    {
        HAL_FLASHEx_DATAEEPROM_Program(FLASH_TYPEPROGRAMDATA_HALFWORD, 0x08080000+i2, MODBUSarray[i]);
        //地址每次偏移+2, 循环写入
        i2 = i2 + 2;
    }
    HAL_FLASHEx_DATAEEPROM_Lock(); //写完EEPROM之后必须上锁, 数据才能存入EEPROM
}

```

将数组的元素依次循环写入地址, 每次循环地址+2, 因为是存放16位数组数据

```

int main(void)
{
    uint8_t data_8bit;
    uint16_t data_16bit;
    uint32_t data_32bit;

    HAL_Init();

    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();

    EEPROM_Write(); //向EEPROM写数据
    while (1)
    {
        //EEPROM_Read();
        /*获取EEPROM起始地址存入的2字节数据*/
        data_16bit = *(__IO uint16_t *) 0x08080000;
        printf("data_16bit = %x\r\n", data_16bit);
        data_16bit = *(__IO uint16_t *) 0x08080002;
        printf("data_16bit = %x\r\n", data_16bit);
        data_16bit = *(__IO uint16_t *) 0x08080004;
        printf("data_16bit = %x\r\n", data_16bit);
        data_16bit = *(__IO uint16_t *) 0x08080006;
        printf("data_16bit = %x\r\n", data_16bit);
        data_16bit = *(__IO uint16_t *) 0x08080008;
        printf("data_16bit = %x\r\n", data_16bit);
        data_16bit = *(__IO uint16_t *) 0x0808000a;
        printf("data_16bit = %x\r\n", data_16bit);
        HAL_Delay(1000);
    }
}

```

读取地址中 16 位数据

```

data_16bit = 101
data_16bit = 202
data_16bit = 303
data_16bit = 404
data_16bit = 505
data_16bit = 606

```

测试结果证明多个 16 位数据连续成功写入 EEPROM

```

//EEPROM_Write(); //向EEPROM写数据
while (1)
{
    //EEPROM_Read();
    /*获取EEPROM地址存入的2字节数据*/
    data_16bit = *(__IO uint16_t *) 0x08080000;
    printf("data_16bit = %x\r\n", data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080002;
    printf("data_16bit = %x\r\n", data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080004;
    printf("data_16bit = %x\r\n", data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080006;
    printf("data_16bit = %x\r\n", data_16bit);
    data_16bit = *(__IO uint16_t *) 0x08080008;
    printf("data_16bit = %x\r\n", data_16bit);
    data_16bit = *(__IO uint16_t *) 0x0808000a;
    printf("data_16bit = %x\r\n", data_16bit);
    HAL_Delay(1000);
}

```

```

data_16bit = 101
data_16bit = 202
data_16bit = 303
data_16bit = 404
data_16bit = 505
data_16bit = 606

```

掉电重新读取数据成功

STM32L0 芯片获取全球唯一 ID

#define UID_BASE 0x1FF80050 全球唯一 ID 基地址在 0x1FF80050

```

uint32_t UID[3];
UID[0] = (uint32_t)(*((uint32_t *)UID_BASE));
UID[1] = (uint32_t)(*((uint32_t *) (UID_BASE + 4)));
UID[2] = (uint32_t)(*((uint32_t *) (UID_BASE + 14))); //第 3 分 UID 地址偏移是 14 不是 8

```

为了方便我们直接用 HAL 库获取 ID

```
uint32_t HAL_GetHalVersion(void); //获取 UID 号
uint32_t HAL_GetREVID(void); //获取版本 ID
uint32_t HAL_GetDEVID(void); //获取设备 ID
uint32_t HAL_GetUIDw0(void); //获取设备全球唯一 ID，我们主要使用这个
uint32_t HAL_GetUIDw1(void); //获取设备全球唯一 ID，我们主要使用这个
uint32_t HAL_GetUIDw2(void); //获取设备全球唯一 ID，我们主要使用这个
```

```
int main(void)
{
    uint32_t UID; //存放UID
    uint32_t REVID; //存放版本ID
    uint32_t DEVID; //存放设备ID
    uint32_t UID0; //存放设备全球唯一ID低位
    uint32_t UID1; //存放设备全球唯一ID中位
    uint32_t UID2; //存放设备全球唯一ID高位

    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART1_UART_Init();
    while (1)
    {
        UID = HAL_GetHalVersion(); //获取UID号
        printf("UID = %x\r\n",UID);
        REVID = HAL_GetREVID(); //获取版本号
        printf("REVID = %x\r\n",REVID);
        DEVID = HAL_GetDEVID(); //获取设备ID
        printf("DEVID = %x\r\n",DEVID);
        UID0 = HAL_GetUIDw0(); //获取设备全球唯一ID
        UID1 = HAL_GetUIDw1();
        UID2 = HAL_GetUIDw2();

        printf("UID0  UID1  UID2  =  %x  %x  %x\r\n",UID0,UID1,UID2);
        HAL_Delay(1000);
    }
}
```

获取 UID0 , UID1, UID2 成功

REVID = 1038
DEVID = 417
UID0 UID1 UID2 = a473830 35323735 2e0051

用 SYSTICK 做系统，微妙，毫秒延时

```
#include "stm32l0xx_hal.h"
#include "delay.h"

uint32_t ui_opt_us;
uint32_t ui_opt_ms;

/*****************/
/*系统滴答定时器延时函数初始化，在 main 函数使用延时前调用该初始化
* 32M 时钟频率
*****************/
void delay_init()
{
    HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK_DIV8);
    ui_opt_us = HAL_RCC_GetHCLKFreq() / 8000000; //这是控制微妙的参数
    ui_opt_ms = ui_opt_us * 1000; //这是控制毫秒的参数
}

/*****************/
/*xus: 微秒延时，1us 误差在 2~3 微妙
*****************/
void delay_us(uint32_t xus)
{
    uint32_t ui_tmp = 0x00;
    SysTick->LOAD = xus * ui_opt_us;
    SysTick->VAL = 0x00;
    SysTick->CTRL = 0x01;

    do
    {
        ui_tmp = SysTick->CTRL;
    }while((ui_tmp & 0x01) && !(ui_tmp & (1 << 16)));

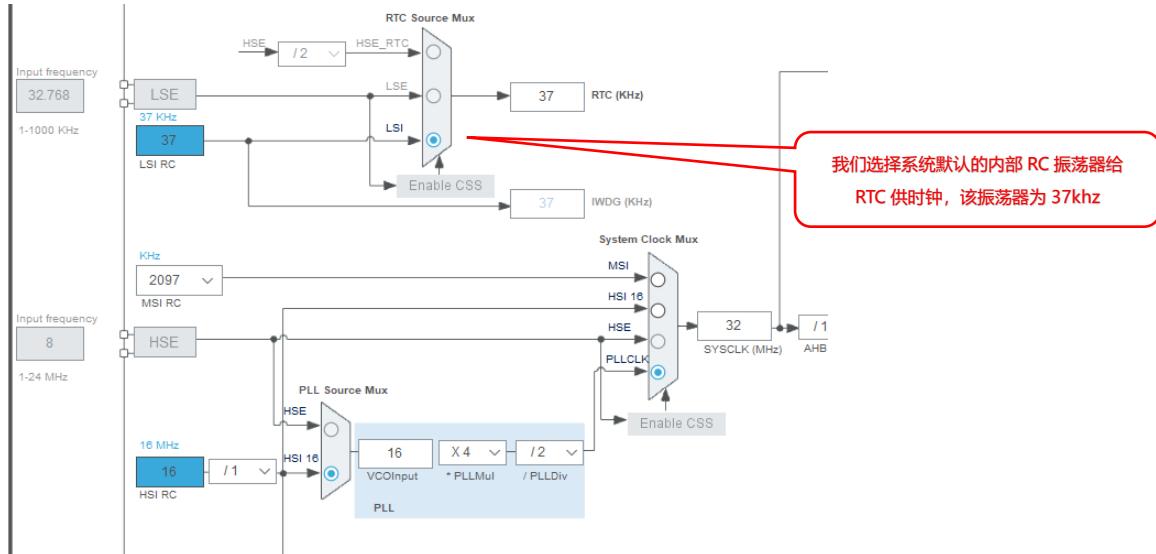
    SysTick->CTRL = 0x00;
    SysTick->VAL = 0x00;
}

/*****************/
/*xms: 毫秒延时
*****************/
void delay_ms(uint32_t xms)
{
    uint32_t ui_tmp = 0x00;
    SysTick->LOAD = xms * ui_opt_ms;
    SysTick->VAL = 0x00;
    SysTick->CTRL = 0x01;

    do
    {
        ui_tmp = SysTick->CTRL;
    }while((ui_tmp & 0x01) && !(ui_tmp & (1 << 16)));

    SysTick->CTRL = 0x00;
    SysTick->VAL = 0x00;
}
```

STM32L0 RTC 时间操作



如果是内部 RC 振荡 37Khz 作为 RTC 时钟源，那么计算 RTC 1秒输出一次脉冲的公式如下：

STM32L0 的预分频器被分成两个预分频器

一个 7 位的异步预分频器 (AsynchPrediv)

一个 13 位同步预分频器 (SynchPrediv)

两个都要使用

$$\text{RTC clk} = \frac{\text{LSI}}{(\text{AsynchPrediv} + 1) \times (\text{SynchPrediv} + 1)}$$

$$\text{RTC clk} = \frac{\text{LSE}}{(\text{AsynchPrediv} + 1) \times (\text{SynchPrediv} + 1)}$$

我们 RTC 是 1 秒计数一次，那么就是 1Hz

$$\text{RTC clk} = \frac{\text{LSI}}{(\text{AsynchPrediv} + 1) \times (\text{SynchPrediv} + 1)} = \frac{37000}{(124+1) \times (295+1)} = 1\text{Hz}$$

$$\text{如果 LSE 外部时钟 } 32.768\text{K: RTC clk} = \frac{\text{LSE}}{(\text{AsynchPrediv} + 1) \times (\text{SynchPrediv} + 1)} = \frac{32768}{(127+1) \times (255+1)} = 1\text{Hz}$$

我们现在用的内部RC，所以选择 AsynchPrediv = 127, SynchPrediv = 255

写错了这是外部

1. 选择 RTC

2. 激活 RTC 校准

3. 按照上面公式填入, 异步
预分频和同步预分频值

4. 选择二进制数
据格式输出

RTC 读取时间实现，生成工程，

RTC_TimeTypeDef 里面存放的是小时，分钟，秒

RTC_DateTypeDef 里面存放的是，年，月，日，星期

HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format) //获取时分秒函数

hrtc: 传入系统默认生成的 RTC 句柄, hrtc。

*sTime: 建立结构体传入，获取时分秒

Format: sTime 时间输出格式 RTC_FORMAT_BIN 二进制输出
RTC_FORMAT_BCD 输出

我 HAL 库选择的是二进制输出，所以这里要填 RTC_FORMAT_BIN

HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate, uint32_t Format) //获取年月日函数

hrtc: 传入系统默认生成的 RTC 句柄, hrtc。

* sDate: 建立结构体传入，获取年月日星期

Format: sTime 时间输出格式 RTC_FORMAT_BIN 二进制输出
RTC_FORMAT_BCD 输出

我 HAL 库选择的是二进制输出，所以这里要填 RTC_FORMAT_BIN

注意: HAL_RTC_GetTime(...) 和 HAL_RTC_GetDate(...) 这两个函数要匹配使用才有效果，不然会出现时间获取失败

```
int main(void)
{
    RTC_TimeTypeDef stimestructure; //时分秒获取结构
    RTC_DateTypeDef sdatestructure; //年月日星期获取结构

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_RTC_Init(); //初始化 RTC 寄存器
    MX_USART1_UART_Init();

    while (1)
    {
        HAL_RTC_GetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN);

        printf("%02d:%02d:%02d\r\n", stimestructure.Hours, stimestructure.Minutes, stimestructure.Seconds);
        HAL_Delay(1000);
    }
}
```

```
00:00:00
00:00:00
00:00:00
00:00:00
00:00:00
00:00:00
```

在没有执行 HAL_RTC_GetDate()的情况下直接使用 HAL_RTC_GetTime()获取时分秒，时间获取失败

```

while (1)
{
    HAL_RTC_GetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(&hrtc, &sdatestructure, RTC_FORMAT_BIN);

    printf("%02d:%02d:%02d\r\n", stimestructure.Hours, stimestructure.Minutes, stimestructure.Seconds);
    HAL_Delay(1000);
}

```

```

00:00:00
00:00:01
00:00:02
00:00:03
00:00:04
00:00:05
00:00:06
00:00:07

```

在 GetTime 后加入 GetDate 函数。时间获取就正常了。

注意: 在调用 HAL_Get_Date() 前必需先调用 HAL_Get_Time, 否则取出来的时间, 会存在延时, 即取值错误!

```

HAL_RTC_GetDate(&hrtc, &sdatestructure, RTC_FORMAT_BIN); //这个顺序是错误的
HAL_RTC_GetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN); //这个顺序是错误的

```

以下顺序才是正确的

```

HAL_RTC_GetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN);
HAL_RTC_GetDate(&hrtc, &sdatestructure, RTC_FORMAT_BIN);

```

设置 RTC 时间, 然后读出来

HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTime, uint32_t Format) //向 RTC 内部设置时间, 时分秒

hrtc: 传入系统默认生成的 RTC 句柄, hrtc。

***sTime:** 建立结构体传入, 获取时分秒, 也可以向结构体写入时分秒, 用 HAL_RTC_SetTime 进行时间设置

Format: 写入时间的格式 RTC_FORMAT_BIN 二进制写入

RTC_FORMAT_BCD BCD 写入

```

stimestructure.Hours=10; //写小时
stimestructure.Minutes=5; //写分钟
stimestructure.Seconds=1; //写秒

if (HAL_RTC_SetTime(&hrtc, &stimestructure, RTC_FORMAT_BCD) != HAL_OK)
{
    Error_Handler();
}
while (1)
{
    HAL_RTC_GetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(&hrtc, &sdatestructure, RTC_FORMAT_BIN);

    printf("%02d:%02d:%02d\r\n", stimestructure.Hours, stimestructure.Minutes, stimestructure.Seconds);
    HAL_Delay(1000);
}

```

```

10:05:01
10:05:02
10:05:03
10:05:04
10:05:05
10:05:06

```

测试结果没有问题

这里写错了, 写入的时候不能以 BCD 写入,
要用 RTC_FORMAT_BIN 二进制写入, 不然
打印出来是 BCD 的值, 相差太远

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc, RTC_DateTypeDef *sDate,
uint32_t Format) //向 RTC 内部设置时间, 年月日
参数同上 注意*sDate 参数是写入 RTC_DateTypeDef 类型的数据
```

下面把完整的，时分秒年月日，写出来

```
int main(void)
{
    RTC_TimeTypeDef stimestructure;
    RTC_DateTypeDef sdatestructure;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_RTC_Init(); //初始化RTC寄存器
    MX_USART1_UART_Init();

    stimestructure.Hours=10; //写小时
    stimestructure.Minutes=5; //写分钟
    stimestructure.Seconds=1; //写秒

    if (HAL_RTC_SetTime(&hrtc, &stimestructure, RTC_FORMAT_BIN) != HAL_OK)
    {
        Error_Handler();
    }

    sdatestructure.Date = 27; //写天数
    sdatestructure.Month = 3; //写月份
    sdatestructure.Year = 20; //写年

    if (HAL_RTC_SetDate(&hrtc, &sdatestructure, RTC_FORMAT_BIN) != HAL_OK)
    {
        Error_Handler();
    }
}
```

```
10:05:01
2020/03/27
10:05:02
2020/03/27
10:05:03
2020/03/27
10:05:04
2020/03/27
10:05:05
2020/03/27
```

时间写入和获取都正常

注意 设置完成后，重启依旧会恢复到原初始化状态，修改 MX_RTC_Init 函数，将设置日期及时间注释掉，可解决掉电恢复初始化问题。但掉电日期依旧恢复默认，不知为何原因，后期实际用途中再深入使用。

进入 MX_RTC_Init() 初始化函数，取消掉每次初始化 RTC 寄存器的时候顺带初始化时间参数的功能。

```
/* RTC init function */
void MX_RTC_Init(void)
{
    RTC_TimeTypeDef sTime = {0};
    RTC_DateTypeDef sDate = {0};

    /** Initialize RTC Only
    */
    hrtc.Instance = RTC;
    hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
    hrtc.Init.AsynchPrediv = 124;
    hrtc.Init.SynchPrediv = 295;
    hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
    hrtc.Init.OutPutRemap = RTC_OUTPUT_REMAP_NONE;
    hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
    hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
    if (HAL_RTC_Init(&hrtc) != HAL_OK)
    {
        Error_Handler();
    }

    /* USER CODE BEGIN Check_RTC_BKUP */

    /* USER CODE END Check_RTC_BKUP */

    /** Initialize RTC and set the Time and Date
    */
    // sTime.Hours = 0;
    // sTime.Minutes = 0;
    // sTime.Seconds = 0;
    // sTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
    // sTime.StoreOperation = RTC_STOREOPERATION_RESET;
    // if (HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BIN) != HAL_OK)
    // {
    //     Error_Handler();
    // }
    // sDate.WeekDay = RTC_WEEKDAY_MONDAY;
    // sDate.Month = RTC_MONTH_JANUARY;
    // sDate.Date = 1;
    // sDate.Year = 0;

    // if (HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BIN) != HAL_OK)
    // {
    //     Error_Handler();
    // }
}
```

主要清除这部分，RTC 初始化时，同时设置 RTC 时间的函数，一看就明白，这部分重新设置时分秒年月日的函数全部注释掉

RTC 外部时钟使用

Home > STM32L051C8Tx > WSNSoftwareFrame2V0.ioc - Pinout & Configuration >

Pinout & Configuration | Clock Configuration

Categories A-Z

System Core

- DMA
- GPIO
- IWDG
- NVIC
- RCC (Selected)
- SYS
- WWDG

Clock Configuration

RCC Mode and Configuration

Mode

High Speed Clock (HSE) Crystal/Ceramic Resonator

Low Speed Clock (LSE) Crystal/Ceramic Resonator

Master Clock Output 1

1.如果 RTC 需要使用外部 32.768Khz 晶振做时钟源，需要先 打开外部低速时钟 LSE 配置

2.这样才能在时钟树 中选择 LSE 选项

```
graph LR
    Input[Input frequency 32.768 kHz] --> LSE[LSE]
    LSE --> HSE[HSE / 2]
    HSE --> RTC[RTC Source Mux]
    RTC --> RTC32[32.768 RTC (kHz)]
    RTC32 --> IWDG[IWDG (kHz)]
    LSE --> LSE_LSI[LSI]
    LSE_LSI --> RTC32
    LSE_LSI --> EnableCSS[Enable CSS]
    EnableCSS --> IWDG
```

Pinout & Configuration

Clock Configuration

RTC Mode and Configuration

Mode

- Activate Clock Source 激活时钟源
- Activate Calendar 激活日历功能
- Timestamp
- WakeUp Internal WakeUp 内部唤醒，这个要注意，我下面会讲
- Tamper 1
- Tamper 2
- Calibration Disable

Configuration

Parameter Settings

Configure the below parameters :

Synchronous Predivider value: 255 选择二级制输出

Calendar Time

Data Format	Binary data format
Hours	0
Minutes	0
Seconds	0

RTC 闹钟 A 中断

Timers

RTC

Configuration

Parameter Settings

Configure the below parameters :

Alarm A Internal Alarm A 打开闹钟 A

Calendar Date

Week Day	Monday
Month	January
Date	1
Year	0

Alarm A

Hours	0
Minutes	0
Seconds	0
Sub Seconds	0
Alarm Mask Date Week day	Disable
Alarm Mask Hours	Disable
Alarm Mask Minutes	Disable
Alarm Mask Seconds	Disable
Alarm Sub Second Mask	All Alarm SS fields are masked.
Alarm Date Week Day Sel	Date
Alarm Date	1

这时候闹钟 A 就会和 RTC 的时分秒进行对比，如果闹钟 A 时间 = RTC 当前时间，触发闹钟 A 中断。

Parameter Settings	User Constants	NVIC Settings		
NVIC Interrupt Table			Enabled	Preempt
RTC global interrupt through EXTI lines 17, 19 and 20 and LSE CSS interrupt through EXTI line 19	<input checked="" type="checkbox"/>	0		

因为闹钟 A 要触发中断，所以必须打开 RTC 中断。

NVIC	NVIC Interrupt Table	Enabled	Preemption Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0	
Hard fault interrupt	<input checked="" type="checkbox"/>	0	
System service call via SWI instruction	<input checked="" type="checkbox"/>	0	
Pendable request for system service	<input checked="" type="checkbox"/>	0	
Time base: System tick timer	<input checked="" type="checkbox"/>	0	
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0	
RTC global interrupt through EXTI lines 17, 19 and 20 and LSE CSS interrupt through EXTI line 19	<input checked="" type="checkbox"/>	0	

```
/** Enable the Alarm A */
sAlarm.AlarmTime.Hours = 0;
sAlarm.AlarmTime.Minutes = 0;
sAlarm.AlarmTime.Seconds = 5; 在 RTC 初始化代码中设置闹钟 A 触发时间，我现在设置 RTC 5 秒时间到触发闹钟 A 中断
sAlarm.AlarmTime.SubSeconds = 0;
sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
sAlarm.AlarmDateWeekDay = 1;
sAlarm.Alarm = RTC_ALARM_A;
if (HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK)
{
    Error_Handler();
}
```

//这个函数是自己单独加入的，一定要是这个名字，这是闹钟 A 钩子函数，所以闹钟 A 中断触发函数是自己加的，千万不要以为 RTC_IRQHandler 中断函数是可以用于闹钟 A 中断，这是不对的。

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc1)
```

```
{
    printf("AlarmA... \r\n");
}
```

```
00:00:00
00:00:01
00:00:02
00:00:03
00:00:04
AlarmA...
00:00:05
```

运行 5 秒闹钟 A 触发

闹钟 A 实现自定义时间重复中断

重复闹钟中断是为了测试闹钟中断之后，再次设置闹钟的功能

在闹钟中断中设置新时间

```
extern RTC_AlarmTypeDef sAlarm;

static uint16_t Count = 10; //第2次闹钟中断从10秒之后开始

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc) //一定要形参的hrtc, 不允许外部extern 引入hrtc
{
    printf("AlarmA... \r\n");

    Count = Count + 5; //每5秒中断一次

    sAlarm.AlarmTime.Hours = 0;
    sAlarm.AlarmTime.Minutes = 0;
    sAlarm.AlarmTime.Seconds = Count; //15,20,25,30.....5秒间隔中断一次
    sAlarm.AlarmTime.SubSeconds = 0;
    sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
    sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
    sAlarm.AlarmMask = RTC_ALARMASK_NONE;
    sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
    sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
    sAlarm.AlarmDateWeekDay = 1;
    sAlarm.Alarm = RTC_ALARM_A;
    if (HAL_RTC_SetAlarm_IT(hrtc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK) //将新设置的闹钟设置进去
    {
        Error_Handler();
    }
}
```

这个 hrtc 直接用中断函数
的形参*hrtc, 不要用外部
extern 引入的 hrtc

```
00:00:05
00:00:06
00:00:07
00:00:08
00:00:09
AlarmA...
00:00:10
00:00:11
00:00:12
00:00:13
00:00:14
AlarmA...
00:00:15
00:00:16
00:00:17
00:00:18
00:00:19
AlarmA...
00:00:20
00:00:21
00:00:22
00:00:23
00:00:24
```

```
/** Enable the WakeUp
 */
if (HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0, RTC_WAKEUPCLOCK_RTCCLK_DIV16) != HAL_OK)
{
    Error_Handler();
}
```

内部唤醒会在 RTC 初始化的时候设置。

如果内部唤醒这儿是 0, 就会造成系统不停的进入

RTC_IRQHandler

这是因为这个函数是设置唤醒时间的, 这儿不该写 0, 而是写等待多少时间唤醒

HAL_RTCEx_SetWakeUpTimer_IT 函数会在低功耗模式章节, 停止模式功能介绍。

所以如果有闹钟 A 中断, 那么后面的低功耗章节, 都可以使用闹钟 A 唤醒。那么内部唤醒就必须取消掉。

```
/** Enable the WakeUp
 */
// if (HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0, RTC_WAKEUPCLOCK_RTCCLK_DIV16) != HAL_OK)
// {
//     Error_Handler();
// }
```

如果不使用闹钟 A 中断唤醒休眠, 也可以使用 HAL_RTCEx_SetWakeUpTimer_IT 来设置唤醒时间。

闹钟 B 闹钟中断

暂时未找到解决方法.....

STM32L0 低功耗实现

STM32 停止模式

表 31. 低功耗模式汇总

模式名称	进入	唤醒	对 V_{CORE} 域时钟的影响	对 V_{DD} 域时钟的影响	调压器
停止	PDDS、LPSDSR 位 + SLEEPDEEP 位 + WFI, 从 ISR 返回或 WFE	任意 EXTI 线 (在 EXTI 寄存器中配置, 内部线和外部线)			在低功耗模式下打开 (取决于 PWR_CR)

停止 有 RTC	0.82 μ A (1.8 V)	无	关闭	启动	冻结	LSE, LSI	关闭	启动
	1.0 μ A (3 V)							

```
HAL_StatusTypeDef HAL_RTCEx_SetWakeUpTimer_IT(RTC_HandleTypeDef *hrtc, uint32_t WakeUpCounter, uint32_t WakeUpClock) //设置唤醒计数器与中断
```

*hrtc : 传入 RTC 句柄, 我们使用 RTC 来唤醒

WakeUpCounter: 设置定时唤醒时间, 就是 RTC 定时多久才能唤醒单片机, 计算方法如下

WakeUpClock: 唤醒时钟

数值 = 定时时间 (秒) * (RTC 时钟) / 唤醒时钟

比如我定时 10 秒唤醒

数值 = 定时时间 (秒) * 37000(内部 LSI) / 16_(RTC_WAKEUPCLOCK_RTCCLK_DIV16) = 定时时间 (秒) * 2312

数值 = 10 * 37000 / 16 = 10 * 2312

例如: HAL_RTCEx_SetWakeUpTimer_IT(&RTCHandle, 1*2312, RTC_WAKEUPCLOCK_RTCCLK_DIV16); //1 秒唤醒一次

```
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

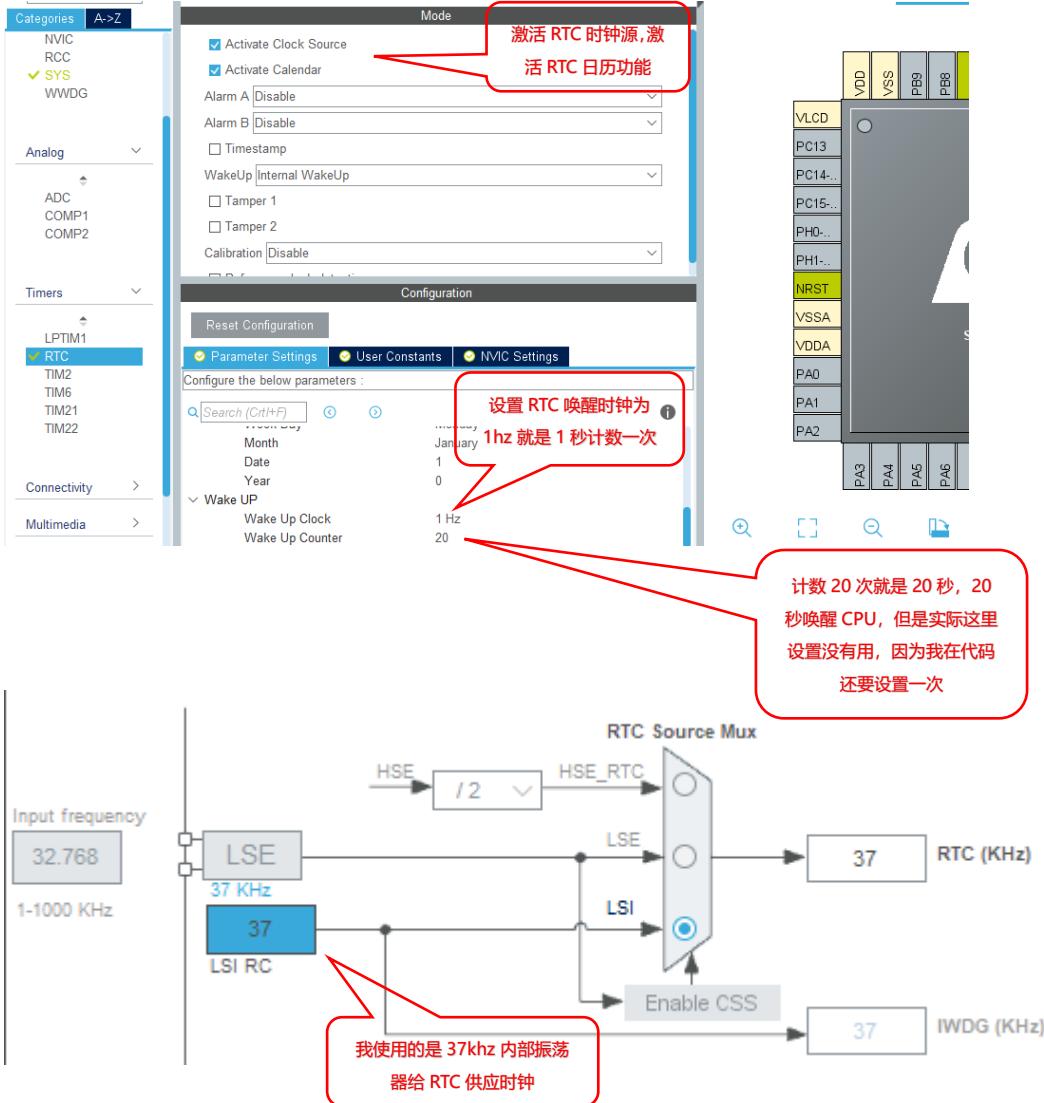
//进入停止模式, 谨慎使用这种模式, 操作不好可能导致单片机无法下载程序

解决办法只有先将 boot0 接高电平, 然后让 NRST 引脚低电平复位, 这时候就可以用 keil 重新下载

其实理论上只需要按住 NRST 复位按键点击下载, 放开复位键就可以成功下载, 但是有一种情况, 就是在初始化停止模式的时候把所有 GPIO 时钟关闭了, 那么每次芯片上电就会进入 GPIO 时钟关闭的模式。导致 SWD 引脚无法使用。所有需要 boot0 拉高电平, 配合 NRST 复位按钮下载。

但是之后一定要将 boot0 一直拉到低电平, 因为做产品的时候, 上电运行是 boot0 为低电平才有效。不然芯片上电不工作。

下面设置 RTC



代码示例

```
#define RTC_CLOCK_SOURCE_LSI
/*#define RTC_CLOCK_SOURCE_LSE*/
#ifndef RTC_CLOCK_SOURCE_LSI //LSI 37Khz 作为 RTC 的唤醒时钟 , 上图介绍过
#define RTC_SYNCH_PREDIV    0x7C
#define RTC_ASYNCH_PREDIV   0x0127
#endif
#ifndef RTC_CLOCK_SOURCE_LSE
#define RTC_ASYNCH_PREDIV   0x7F
#define RTC_SYNCH_PREDIV    0x00FF
#endif
int fputc(int ch,FILE *f)
{
    HAL_UART_Transmit(&huart1,(uint8_t *) &ch,1,1); //重定向发送字符
    return ch;
}

void SystemClock_Config(void);
RTC_HandleTypeDef RTCHandle; //创建一个唤醒 CPU 的 RTC 时钟结构

static void SystemPower_Config(void) //如果要使用停止模式, 在主函数初始化的时候就需要初始化停止模式相关的寄存器
{
    GPIO_InitTypeDef GPIO_InitStructure;
    /* Enable Ultra low power mode */
}
```

```

HAL_PWREx_EnableUltraLowPower(); //使能低功耗模式
/* Enable the fast wake up from Ultra low power mode */
HAL_PWREx_EnableFastWakeUp(); //使能快速唤醒
/* Select HSI as system clock source after Wake Up from Stop mode */
__HAL_RCC_WAKEUPSTOP_CLK_CONFIG(RCC_STOP_WAKEUPCLOCK_HSI); //从停止模式唤醒后选择内部 HSI 作为系统时钟
__HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU); // 清除唤醒标志
__HAL_RCC_PWR_CLK_ENABLE(); //使能停止模式时钟

// /* Enable GPIOs clock */
// __HAL_RCC_GPIOA_CLK_ENABLE();
// __HAL_RCC_GPIOB_CLK_ENABLE();
// __HAL_RCC_GPIOC_CLK_ENABLE();
// __HAL_RCC_GPIOD_CLK_ENABLE();
// __HAL_RCC_GPIOH_CLK_ENABLE();

// /* Configure all GPIO port pins in Analog Input mode (floating input trigger OFF) */
// GPIO_InitStructure.Pin = GPIO_PIN_All;
// GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
// GPIO_InitStructure.Pull = GPIO_NOPULL;
// HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
// HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);
// HAL_GPIO_Init(GPIOC, &GPIO_InitStructure);
// HAL_GPIO_Init(GPIOD, &GPIO_InitStructure);
// HAL_GPIO_Init(GPIOH, &GPIO_InitStructure);

// /* Disable GPIOs clock */
// __HAL_RCC_GPIOA_CLK_DISABLE();
// __HAL_RCC_GPIOB_CLK_DISABLE();
// __HAL_RCC_GPIOC_CLK_DISABLE();
// __HAL_RCC_GPIOD_CLK_DISABLE();
// __HAL_RCC_GPIOH_CLK_DISABLE();

/* Configure RTC */
RTCHandle.Instance = RTC;
/* Configure RTC prescaler and RTC data registers as follow:
- Hour Format = Format 24
- Asynch Prediv = Value according to source clock
- Synch Prediv = Value according to source clock
- OutPut = Output Disable
- OutPutPolarity = High Polarity
- OutPutType = Open Drain */
RTCHandle.Init.HourFormat = RTC_HOURFORMAT_24;
RTCHandle.Init.AsynchPrediv = RTC_ASYNCH_PREDIV;
RTCHandle.Init.SynchPrediv = RTC_SYNCH_PREDIV;
RTCHandle.Init.OutPut = RTC_OUTPUT_DISABLE;
RTCHandle.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
RTCHandle.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
if(HAL_RTC_Init(&RTCHandle) != HAL_OK)
{
    /* Initialization Error */
    Error_Handler();
}
}

static void SystemClockConfig_STOP(void) //停止模式唤醒后，需要启动的电源和时钟
{
    RCC_ClkInitTypeDef RCC_ClkInitStruct;
    RCC_OscInitTypeDef RCC_OscInitStruct;

    /* Enable Power Control clock */
    __HAL_RCC_PWR_CLK_ENABLE();

    /* The voltage scaling allows optimizing the power consumption when the device is
       clocked below the maximum system frequency, to update the voltage scaling value
       regarding system frequency refer to product datasheet. */
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);
}

```

```

/* Get the Oscillators configuration according to the internal RCC registers */
HAL_RCC_GetOscConfig(&RCC_OsclInitStruct);

/* After wake-up from STOP reconfigure the system clock: Enable HSI and PLL */
RCC_OsclInitStruct.OscillatorType = RCC OSCILLATORTYPE_HSI;
RCC_OsclInitStruct.HSEState = RCC_HSE_OFF;
RCC_OsclInitStruct.HSISState = RCC_HSI_ON;
RCC_OsclInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OsclInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
RCC_OsclInitStruct.PLL.PLLMUL = RCC_PLL_MUL4;
RCC_OsclInitStruct.PLL.PLLDIV = RCC_PLL_DIV2;
RCC_OsclInitStruct.HSICalibrationValue = 0x10;
if(HAL_RCC_OscConfig(&RCC_OsclInitStruct) != HAL_OK)
{
    Error_Handler();
}

/* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2
   clocks dividers */
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
if(HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
{
    Error_Handler();
}

int main(void)
{
    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_RTC_Init();
    MX_USART1_UART_Init();
    SystemPower_Config(); // 使用停止模式前，配置停止模式初始化
    while (1)
    {
        printf("STOP 11111111\r\n"); // 程序停止之前运行
        HAL_Delay(1000);
        HAL_RTCE_DeactivateWakeUpTimer(&hrtc); // 关闭计数器
        _HAL_RTC_WAKEUPTIMER_EXTI_CLEAR_FLAG(); // 清除 RTC 唤醒标志，否则第二次以后无法进入停止模式
        HAL_RTCE_SetWakeUpTimer_IT(&hrtc, 5*2312, RTC_WAKEUPCLOCK_RTCCLK_DIV16); // 设置停止模式持续时间 5 秒
        HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI); // 进入停止模式
        SystemClockConfig_STOP(); // 5 秒之后唤醒，重新启动电源和时钟
        printf("RUN 22222222\r\n"); // 你会发现程序时接着执行的，而不是复位重新执行
    }
}

```

这就是停止模式，程序不会复位重新执行，而是接着执行。

STM32 睡眠模式实现

低功耗模式	电流消耗	CPU	Flash / EEPROM	RAM	DMA & 外设	时钟	LCD	RTC
睡眠	41 μA/MHz (范围 1)	无	启动	启动	ACTIVE	任意	可用	
	36 μA/MHz (范围 2)							
	35 μA/MHz (范围 3)							

HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry) // 进入睡眠模式

Regulator : PWR_MAINREGULATOR_ON 睡眠模式保持电压调节器开启

PWR_LOWPOWERREGULATOR_ON 睡眠模式低功率调节器开启

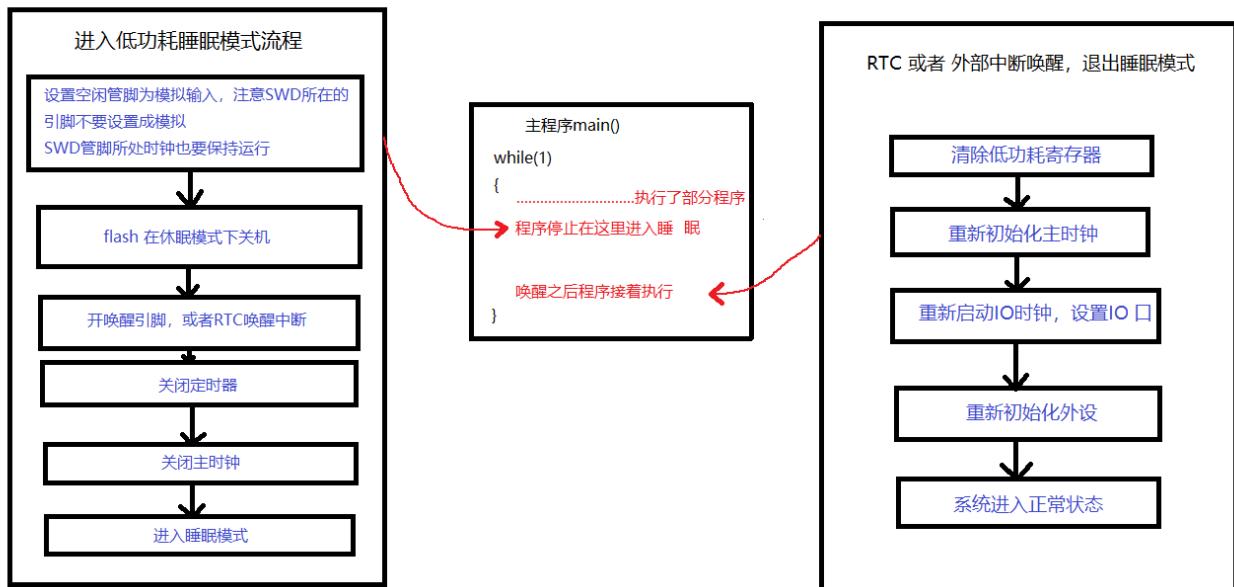
SLEEPEntry: PWR_SLEEPENTRY_WFI 在 WFI 指令下进入睡眠模式

PWR_SLEEPENTRY_WFE 在 WFE 指令下进入睡眠模式

在 WFI 指令进入的睡眠模式,可以由 NVIC 中启用的任意中断源唤醒,然后去唤醒的中断线,执行中断程序。

在 WFE 指令进入的睡眠模式,可以由事件方式唤醒设备,接着主函数执行程序,而不是去唤醒的中断线执行中断程序。

低功耗睡眠模式执行流程,停止模式可以参考



不影响下载程序的睡眠方式

```
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if(time == 5)
    {
        HAL_SuspendTick(); // 进入低功耗睡眠模式时一定要关闭sysTick中断唤醒,不然每毫秒都会被sysTick中断唤醒
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); // 进入睡眠模式
        HAL_ResumeTick(); // 唤醒后恢复sysTick中断
    }
}

xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
```

系统执行 5 秒之后,直接睡眠。这种只是普通睡眠模式,不是超低功耗睡眠模式。

睡眠模式 RTC 闹钟 A 唤醒

```
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if(time == 5) // 5秒后休眠
    {
        time = 0;
        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); // 进入睡眠模式
        HAL_ResumeTick();
    }
}
```

```

/** Enable the Alarm A
*/
sAlarm.AlarmTime.Hours = 0;
sAlarm.AlarmTime.Minutes = 0;
sAlarm.AlarmTime.Seconds = 10;
sAlarm.AlarmTime.SubSeconds = 0;
sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
sAlarm.AlarmDateWeekDay = 1;
sAlarm.Alarm = RTC_ALARM_A;
if (HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK)
{
    Error_Handler();
}

/** Enable the WakeUp
*/
// if (HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0, RTC_WAKEUPCLOCK_RTCCLK_DIV16) != HAL_OK) //取消掉内部唤醒，内部唤醒是分频RTC的LSI时钟实现的
// {
//     Error_Handler();
// }

```

```

extern RTC_AlarmTypeDef sAlarm;

static uint16_t Count = 10; //第2次唤醒系统重10秒之后唤醒

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc) //一定要形参的hrtc，不允许外部extern 引入hrtc
{
    printf("AlarmA... \r\n");

    Count = Count + 10; //每10秒唤醒一次

    sAlarm.AlarmTime.Hours = 0;
    sAlarm.AlarmTime.Minutes = 0;
    sAlarm.AlarmTime.Seconds = Count; //10,20,30,40唤醒
    sAlarm.AlarmTime.SubSeconds = 0;
    sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
    sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
    sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
    sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
    sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
    sAlarm.AlarmDateWeekDay = 1;
    sAlarm.Alarm = RTC_ALARM_A;
    if (HAL_RTC_SetAlarm_IT(hrtc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK) //将新设置的闹钟设置进去
    {
        Error_Handler();
    }
}

xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
AlarmA...
RTC_IRQHandler.....
time = 0
time = 1
time = 2
time = 3
time = 4
AlarmA...
RTC_IRQHandler.....
time = 0
time = 1
time = 2
time = 3
time = 4
AlarmA...
RTC_IRQHandler.....

```

STM32 超低功耗睡眠模式

低功耗模式	电流消耗	CPU	Flash / EEPROM	RAM	DMA & 外设	时钟	LCD	RTC
低功耗睡眠	4.65 μA (外设关)	无	关闭	启动	ACTIVE	MSI	可用	

配置方式: 进入方式: 进入睡眠模式前, 配置为低功耗运行模式, 然后就要上节的睡眠模式休眠。

唤醒方式: 任意中断唤醒, 或者任意事件唤醒

HAL_PWREx_EnableUltraLowPower() //开启低功耗模式

HAL_PWREx_EnableFastWakeUp() //开启超低功耗快速唤醒

```

GPIO_InitTypeDef GPIO_InitStructure = {0};

int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if (time == 5)
    {
        HAL_RCC_PWR_CLK_ENABLE(); //启动电源控制时钟
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗模式
        HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

        /* 先启动所有GPIO时钟 */
        __HAL_RCC_GPIOA_CLK_ENABLE();
        __HAL_RCC_GPIOB_CLK_ENABLE();
        __HAL_RCC_GPIOC_CLK_ENABLE();
        __HAL_RCC_GPIOD_CLK_ENABLE();
        __HAL_RCC_GPIOH_CLK_ENABLE();

        /* 所有IO口成模拟之后, ST-LINK的下载口和SWD下载引脚都无法使用 */
        GPIO_InitStructure.Pin = GPIO_PIN_All; //将所有IO口设置成模拟输入才能进一步降低功耗
        GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
        GPIO_InitStructure.Pull = GPIO_NOPULL;

        HAL_GPIO_Init(GPIOA, &GPIO_InitStructure); //设置PA组IO为模拟输入
        HAL_GPIO_Init(GPIOB, &GPIO_InitStructure); //设置PB组IO为模拟输入
        HAL_GPIO_Init(GPIOC, &GPIO_InitStructure); //设置PC组IO为模拟输入
        HAL_GPIO_Init(GPIOH, &GPIO_InitStructure); //设置PH组IO为模拟输入

        //再次关闭所有GPIO时钟
        __HAL_RCC_GPIOA_CLK_DISABLE();
        __HAL_RCC_GPIOB_CLK_DISABLE();
        __HAL_RCC_GPIOC_CLK_DISABLE();
        __HAL_RCC_GPIOH_CLK_DISABLE();

        HAL_SuspendTick(); //进入低功耗睡眠模式时一定要关闭sysTick中断唤醒, 不然每毫秒都会被sysTick中断唤醒
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); //进入睡眠模式
        HAL_ResumeTick(); //唤醒后恢复sysTick中断
    }
}

```

和前面章节 STOP 停止一样, 把 IO 口设置成模拟输入, 一样的无法直接下载程序, 但是所有 IO 口设为输入可以进一步降低功耗

```

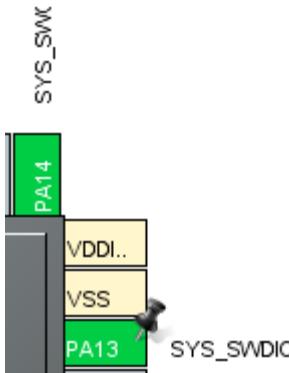
xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4

```

程序执行到 5 秒后, 进入低功耗睡眠, 无法下载程序。

在这种情况下只有长按复位按钮, 瞬间松开同时点击下载, 才能把程序下载进去。

有一种办法能解决低功耗无法下载程序的情况



我的 SWD 下载口是 PA 引脚, 那我是不修改 PA14 和 PA13 为模拟输入就

可以解决这个问题呢?

```
if(time == 5)
{
    __HAL_RCC_PWR_CLK_ENABLE(); //启动电源控制时钟
    HAL_PWREx_EnableUltraLowPower(); //开启低功耗模式
    HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

    /* 先启动所有GPIO时钟 */
    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOD_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();

    /*所有IO口成模拟之后, ST-LINK的下载口和SWD下载引脚都无法使用*/
    GPIO_InitStructure.Pin = GPIO_PIN_All; //将所有IO口设置成模拟输入才能进一步降低功耗
    GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStructure.Pull = GPIO_NOPULL;

    // HAL_GPIO_Init(GPIOA, &GPIO_InitStructure); //设置PA组引脚为模拟输入, 让程序能下载
    HAL_GPIO_Init(GPIOB, &GPIO_InitStructure); //设置PB组IO为模拟输入
    HAL_GPIO_Init(GPIOC, &GPIO_InitStructure); //设置PC组IO为模拟输入
    HAL_GPIO_Init(GPIOH, &GPIO_InitStructure); //设置PH组IO为模拟输入

    /*再次关闭所有GPIO时钟*/
    __HAL_RCC_GPIOA_CLK_DISABLE();
    __HAL_RCC_GPIOB_CLK_DISABLE();
    __HAL_RCC_GPIOC_CLK_DISABLE();
    __HAL_RCC_GPIOH_CLK_DISABLE();

    HAL_SuspendTick(); //进入低功耗睡眠模式时一定要关闭sysTick中断唤醒, 不然每毫秒都会被sysTick中断唤醒
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); //进入睡眠模式
    HAL_ResumeTick(); //唤醒后恢复sysTick中断
}
```

经过测试, 确实将 PA 口不设置为模拟输入。那怕是把 PA 口的时钟关了, 在超低功耗模式下, 都能正常下载程序。可以考虑只将 SWD 的 PA13,PA14 设置成普通 IO, 其余 PA 口设置成模拟输入。应该也是可以的。

超低功耗睡眠模式 RTC 闹钟 A 唤醒

```
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if(time == 5) //5秒后休眠
    {
        time = 0;
        __HAL_RCC_PWR_CLK_ENABLE();
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗模式
        HAL_PWREx_EnableFastWakeUp(); //开启低功耗快速唤醒

        /*先启动GPIO时钟*/
        __HAL_RCC_GPIOA_CLK_ENABLE();
        __HAL_RCC_GPIOB_CLK_ENABLE();
        __HAL_RCC_GPIOC_CLK_ENABLE();
        __HAL_RCC_GPIOD_CLK_ENABLE();
        __HAL_RCC_GPIOH_CLK_ENABLE();

        GPIO_InitStructure.Pin = GPIO_PIN_All;
        GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
        GPIO_InitStructure.Pull = GPIO_NOPULL;
```

```

//      HAL_GPIO_Init(GPIOA,&GPIO_InitStructure); //不要关闭串口和下载口时钟，这样休眠可以下载，唤醒看得到打印
HAL_GPIO_Init(GPIOB,&GPIO_InitStructure);
HAL_GPIO_Init(GPIOC,&GPIO_InitStructure);
HAL_GPIO_Init(GPIOH,&GPIO_InitStructure);

    HAL_RCC_GPIOA_CLK_DISABLE();
    HAL_RCC_GPIOB_CLK_DISABLE();
    HAL_RCC_GPIOC_CLK_DISABLE();
    HAL_RCC_GPIOD_CLK_DISABLE();
    HAL_RCC_GPIOH_CLK_DISABLE();

    HAL_SuspendTick();
HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI); //进入睡眠模式
HAL_ResumeTick();

    HAL_RCC_GPIOA_CLK_ENABLE(); //休眠唤醒后必须接着开时钟
    HAL_RCC_GPIOB_CLK_ENABLE(); //休眠唤醒后必须接着开时钟
    HAL_RCC_GPIOC_CLK_ENABLE(); //休眠唤醒后必须接着开时钟
    HAL_RCC_GPIOD_CLK_ENABLE(); //休眠唤醒后必须接着开时钟
    HAL_RCC_GPIOH_CLK_ENABLE(); //休眠唤醒后必须接着开时钟
}

printf("continue to run\r\n"); //休眠唤醒后接着执行
}

```

主程序

```

/** Enable the Alarm A
*/
sAlarm.AlarmTime.Hours = 0;
sAlarm.AlarmTime.Minutes = 0;
sAlarm.AlarmTime.Seconds = 10;
sAlarm.AlarmTime.SubSeconds = 0;
sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
sAlarm.AlarmDateWeekDay = 1;
sAlarm.Alarm = RTC_ALARM_A;
if (HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK)
{
    Error_Handler();
}

/** Enable the WakeUp
*/
// if (HAL_RTCEx_SetWakeUpTimer_IT(&hrtc, 0, RTC_WAKEUPCLOCK_RTCCLK_DIV16) != HAL_OK) //取消掉内部唤醒，内部唤醒是分频RTC的LSI时钟实现的。免得和闹钟A冲突
// {
//     Error_Handler();
// }

}

```

闹钟设置

```

void RTC_IRQHandler(void)
{
    /* USER CODE BEGIN RTC_IRQn 0 */

    /* USER CODE END RTC_IRQn 0 */
    HAL_RTC_AlarmIRQHandler(&hrtc);
    HAL_RTCEx_WakeUpTimerIRQHandler(&hrtc);
    /* USER CODE BEGIN RTC_IRQn 1 */

    printf("RTC_IRQHandler....\r\n");

    /* USER CODE END RTC_IRQn 1 */

extern RTC_AlarmTypeDef sAlarm;

static uint16_t Count = 10; //第2次唤醒系统重10秒之后唤醒

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc) //一定要形参的hrtc，不允许外部extern 引入hrtc
{
    printf("AlarmA... \r\n");

    Count = Count + 10; //每10秒唤醒一次

    sAlarm.AlarmTime.Hours = 0;
    sAlarm.AlarmTime.Minutes = 0;
    sAlarm.AlarmTime.Seconds = Count; //10,20,30,40唤醒
    sAlarm.AlarmTime.SubSeconds = 0;
    sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
    sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;
    sAlarm.AlarmMask = RTC_ALARMMASK_NONE;
    sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
    sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
    sAlarm.AlarmDateWeekDay = 1;
    sAlarm.Alarm = RTC_ALARM_A;
    if (HAL_RTC_SetAlarm_IT(hrctc, &sAlarm, RTC_FORMAT_BIN) != HAL_OK) //将新设置的闹钟设置进去
    {
        Error_Handler();
    }
}

```



这说明了低功耗睡眠模式，唤醒后是接着程序执行，而不是整个程序复位。

待机模式

低功耗模式	电流消耗	CPU	Flash / EEPROM	RAM	DMA & 外设	时钟	LCD	RTC
待机 有 RTC	655 nA (3 V)	关闭	关闭	关闭	关闭	LSE	关闭	启动
	845 nA (1.8 V)							
待机	290 nA	关闭	关闭	关闭	关闭	-	关闭	关闭

唤醒方式: WKUP 引脚上升沿, RTC 闹钟(闹钟 A/闹钟 B), RTC 唤醒事件, NRST 外部按键复位, IWDG 复位。

WKUP 唤醒引脚有, PC13(WKUP2), PA0(WKUP1)。

HAL_PWR_EnableWakeUpPin(uint32_t WakeUpPinx) //设置 WKUP1 或者 WKUP2 为唤醒引脚

WakeUpPinx : PWR_WAKEUP_PIN1 也就是 PA0 引脚唤醒

PWR_WAKEUP_PIN2 也就是 PC13 引脚唤醒

_HAL_RCC_PWR_CLK_ENABLE() // 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式

HAL_PWR_EnterSTANDBYMode() //进入待机模式

```
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if (time == 5)
    {
        _HAL_RCC_PWR_CLK_ENABLE(); // 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式
        HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); //启用连接到PA0的WakeUp Pin
        HAL_PWR_EnterSTANDBYMode(); //进入待机模式
    }
}

xxxxzzzz
time = 0
time = 1
time = 2
time = 3
time = 4
```

运行 5 秒之后, 进入待机模式, 在待机情况下是无法下载程序的, 只有长按复位键才行。

待机模式唤醒操作

WKUP1, WKUP2 引脚上升沿唤醒

1、第一个问题，经过分析是硬件设计导致，wakeup上升沿唤醒待机模式，而我的按键设计成wakeup引脚直接上拉，导致不断唤醒待机模式；

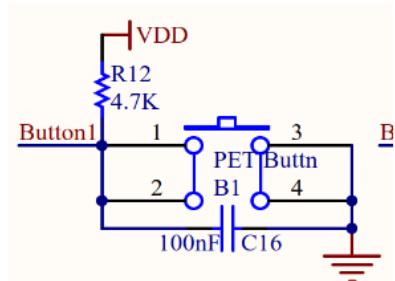
位 8 EWUP1：使能 WKUP 引脚 1 (Enable WKUP pin 1)

此位由软件置 1 和清零。

0: WKUP 引脚 1 用于通用 I/O。WKUP 引脚 1 上的事件不会将器件从待机模式唤醒。

1: WKUP 引脚 1 用于从待机模式唤醒器件并被强制配置成输入下拉（WKUP 引脚 1 出现上升沿时从待机模式唤醒系统）。

注：此位通过系统复位进行复位。

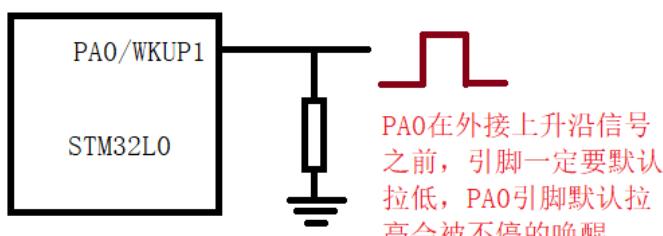


2、发现第二个问题，检查系统是否从待机模式恢复的判断标志改为 PWR_FLAG_WU 最好，只有发生外部唤醒事件，才会将此位置为 1，而 PWR_FLAG_SB 只要进入待机模式，此位均为 1，如果通过复位方式唤醒 MCU，这里的 SB 标志依然成立，判断存在考虑不周全的情况。

```
*****待机模式唤醒测试代码*****
LED_Red_On;
LED_Green_On;
__HAL_RCC_PWR_CLK_ENABLE(); // 使能电源管理单元的时钟, 必须要使能时钟才能进入待机模式
HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1); // 禁用所有使用的唤醒源: PWR_WAKEUP_PIN1 connected to PA.00
if (__HAL_PWR_GET_FLAG(PWR_FLAG_WU) != RESET) // 检查系统是否从待机模式恢复
{
    // __HAL_PWR_CLEAR_FLAG(d); // 清除待机标志
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU); // WUF 为 1 表示收到唤醒事件, 这里清除唤醒标志
    printf("待机唤醒复位 \r\n");
}

else printf("非待机唤醒复位 \r\n");
delay_ms(5000);
LED_Red_Off;
LED_Green_Off;
printf("进入待机模式 \r\n");
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); // 启用连接到 PA.00 的 WakeUp Pin
HAL_PWR_EnterSTANDBYMode(); // 进入待机模式
```

下面以一个案例为例，来测试超低功耗下待机模式用 WKUP(PA0)引脚唤醒方式



```

int main(void)
{
    HAL_Init();
    /* Configure the system clock */
    SystemClock_Config();
    /* USER CODE BEGIN SysInit */
    module_Init();
    /* USER CODE END SysInit */
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART1_UART_Init();
    MX_ADC_Init();
    // MX_RTC_Init(); //注意，测试待机模式的时候要关闭 RTC，除非你屏蔽掉了 RTC 触发中断
    printf("stm32L051 111111111111\r\n");
    /* USER CODE BEGIN WHILE */
    GPIO_InitTypeDef GPIO_InitStructure = {0};
    uint8_t Count = 0;

    while (1)
    {
        /* USER CODE BEGIN 3 */
        if (_HAL_PWR_GET_FLAG(PWR_FLAG_WU) != RESET)//检查系统是否从待机模式恢复
        {
            _HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
            //WUF 为 1 表示收到唤醒事件，这里清除唤醒标志，每次唤醒必须清楚该标志位，不然唤醒一次后就再也无法进入待机模式了
            printf("待机唤醒复位 \r\n");
        }

        printf("Count = %d\r\n", Count); //计数 10 次进入待机模式
        Count++;
        if(Count == 10)
        {
            _HAL_RCC_PWR_CLK_ENABLE(); // 使能电源管理单元的时钟,必须要使能时钟才能进入待机模式
            HAL_PWREx_EnableUltraLowPower(); //开启低功耗模式
            HAL_PWREx_EnableFastWakeUp(); //开启低功耗快速唤醒
            HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); //设置 PA0 为 WAKE UP 唤醒
            /*先启动 GPIO 时钟*/
            _HAL_RCC_GPIOA_CLK_ENABLE();
            _HAL_RCC_GPIOB_CLK_ENABLE();
            _HAL_RCC_GPIOC_CLK_ENABLE();
            _HAL_RCC_GPIOD_CLK_ENABLE();
            _HAL_RCC_GPIOH_CLK_ENABLE();

            GPIO_InitStructure.Pin = GPIO_PIN_All;
            GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
            GPIO_InitStructure.Pull = GPIO_NOPULL;

            // HAL_GPIO_Init(GPIOA,&GPIO_InitStructure); //不要关闭串口和下载口时钟，这样休眠可以下载。
            //注意：在待机模式下 STANDBYMode 就算有 PA 时钟也无法下载程序
            HAL_GPIO_Init(GPIOB,&GPIO_InitStructure);
            HAL_GPIO_Init(GPIOC,&GPIO_InitStructure);
            HAL_GPIO_Init(GPIOH,&GPIO_InitStructure);
        }
    }
}

```

```
    _HAL_RCC_GPIOA_CLK_DISABLE();
    _HAL_RCC_GPIOB_CLK_DISABLE();
    _HAL_RCC_GPIOC_CLK_DISABLE();
    _HAL_RCC_GPIOD_CLK_DISABLE();
    _HAL_RCC_GPIOH_CLK_DISABLE();

    HAL_PWR_EnterSTANDBYMode(); //进入待机模式
}

HAL_Delay(1000);

}
```

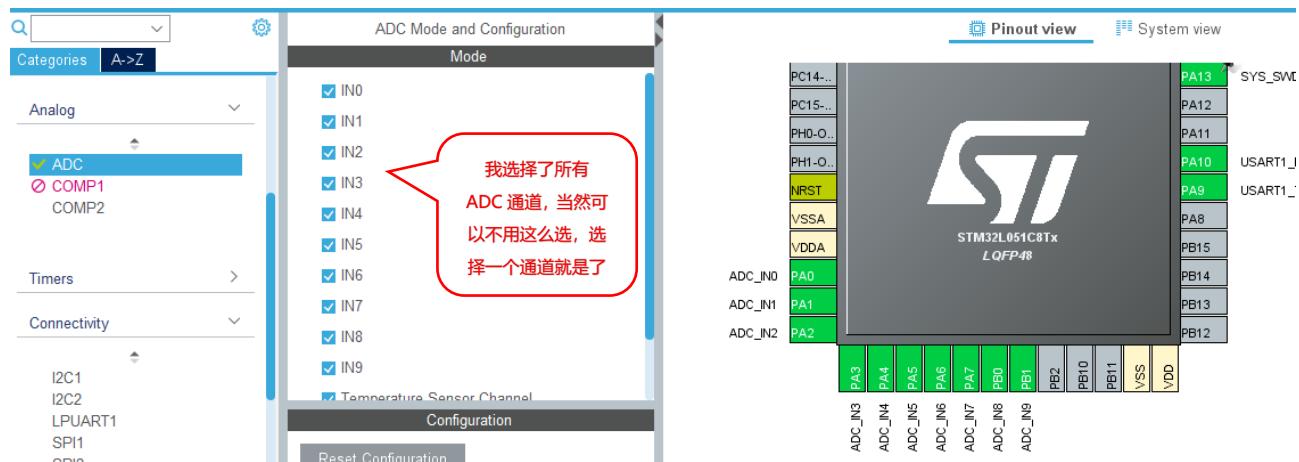
```
stm32L051 111111111111  
Count = 0  
Count = 1  
Count = 2  
Count = 3  
Count = 4  
Count = 5  
Count = 6  
Count = 7  
Count = 8  
Count = 9  
  
Count = 7  
Count = 8  
Count = 9  
stm32L051 111111111111  
待机唤醒复位  
Count = 0  
Count = 1  
Count = 2  
Count = 3  
Count = 4  
Count = 5  
Count = 6  
Count = 7  
Count = 8  
Count = 9
```

系统进入了待机模式 Count = 9

PA0 上升沿喚醒

ADC 使用

ADC 单通道采集实现

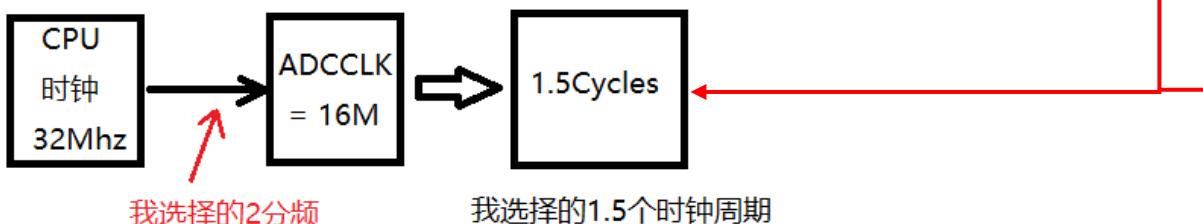




Configure the below parameters :

<input type="button" value="Search (Ctrl+F)"/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="i"/>
ADC_Settings	Clock Prescaler	Synchronous clock mode divided by 2	选择 2 分频，系统时钟 32M，那么 $32M/2 = 16M$ ADC 时钟为 16M，这是 ADC 外设要求最大不超过 16M
	Resolution	ADC 12-bit resolution	使用 12 位 ADC 分辨率
	Data Alignment	Right alignment	
	Scan Direction	Forward	
	Continuous Conversion Mode	Enabled	ADC 采集数据右对齐
	Discontinuous Conversion Mode	Disabled	
	DMA Continuous Requests	Disabled	
	End Of Conversion Selection	End of single conversion	ADC 从 1 通道开始向下扫描采集
	Overrun behaviour	Overrun data preserved	
	Low Power Auto Wait	Disabled	
	Low Frequency Mode	Disabled	ADC 连续采集数据
	Auto Off	Disabled	
	Oversampling Mode	Disabled	采样时间为 1.5 个时钟周期，怎么理解呢？
ADC-Regular_ConversionMode	Sampling Time	1.5 Cycles	
	External Trigger Conversion So...	Regular Conversion launched by software	
	External Trigger Conversion Ed...	None	
WatchDog	Enable Analog WatchDog Mode	<input type="checkbox"/>	软件触发，也就是代码触发

Sampling Time 采样时间详解，主要得出 ADC 转换一次数据的时间



`hadc.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;`

ADC转换时间公式如下：

ADC转换时间= ADC采样时间+12.5个ADC时钟周期(Cycles) 12.5是12位转换精度时的值

如果ADCCLK = 16M

Cycles = 1.5 也是就是ADC采样时间为1.5个时钟周期

$1/1142857\text{hz}$

ADC转换时间 = $16M / (1.5 + 12.5) = 1142857\text{hz} = 1.142857\text{M} = 0.8\mu\text{s}$

如果ADCCLK = 16M

Cycles = 3.5

ADC转换时间 = $16M / (3.5 + 12.5) = 1\text{M}$ 所以转换时间为 $1\mu\text{s}$ 明显转换时间慢于1.5Cycles

代码实验

```
int main(void)
{
    __IO uint32_t ADC_DATA = 0;

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_ADC_Init(); //ADC 初始化参数配置
    MX_USART1_UART_Init();

    if (HAL_ADC_Init(&hadc) != HAL_OK) //初始化ADC
    {
        Error_Handler();
    }

    /* 启动ADC校准 如果是休眠后唤醒也一定要执行一次ADC校准 */
    if (HAL_ADCEx_Calibration_Start(&hadc, ADC_SINGLE_ENDED) != HAL_OK)
    {
        Error_Handler();
    }

    if (HAL_ADC_Start(&hadc) != HAL_OK) //启动ADC转换
    {
        /* Start Conversation Error */
        Error_Handler();
    }

    while (1)
    {
        HAL_ADC_PollForConversion(&hadc, 10); //等待ADC转换完成,超时时间10毫秒

        /* 检测ADC转换是否完成,如果转换完成,获取ADC转换的数据 */
        if ((HAL_ADC_GetState(&hadc) & HAL_ADC_STATE_REG_EOC) == HAL_ADC_STATE_REG_EOC)
        {
            /* 获取ADC转换数据 */
            ADC_DATA = HAL_ADC_GetValue(&hadc);
        }

        printf("ADC data = %d \r\n", ADC_DATA);
        HAL_Delay(1000);
    }
}
```

ADC 初始化参数配置

ADC 初始化,为什么不把初始化代码放到
MX_ADC_Init()里面

这是因为如果是做低功耗产品,在唤醒的时
候要灵活执行一次 ADC 初始化

MX_ADC_Init()初始化解析

```
/* ADC init function */
void MX_ADC_Init(void)
{
    ADC_ChannelConfTypeDef sConfig = {0};

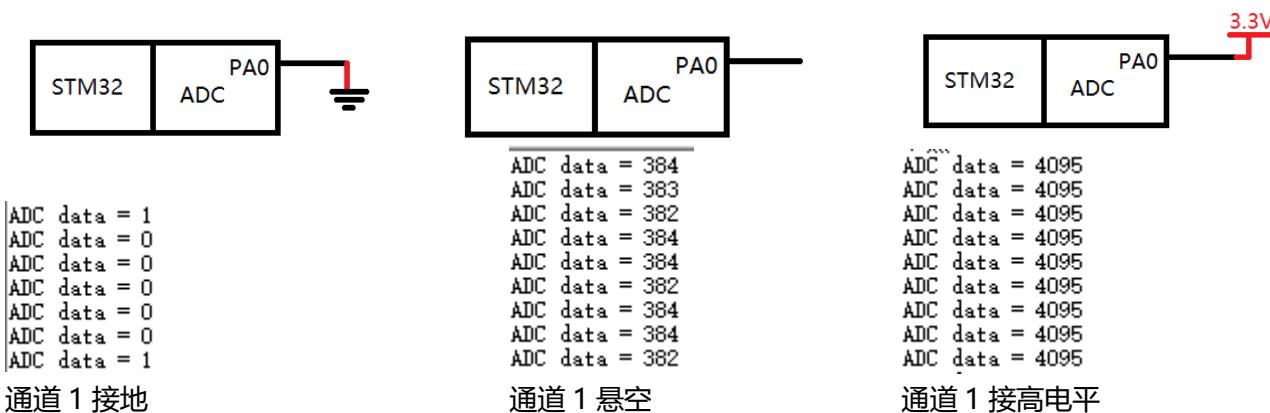
    /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion) */
    hadc.Instance = ADC1;
    hadc.Init.OversamplingMode = DISABLE; //取消ADC过采样模式,过采样用来提高ADC精度,比如12位ADC提高到14位
    hadc.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2; //主时钟分频给ADCCLK
    hadc.Init.Resolution = ADC_RESOLUTION_12B; //12位ADC分辨率
    hadc.Init.SamplingTime = ADC_SAMPLETIME_1CYCLE_5; //1.5个ADC时钟周期
    hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD; //从通道0向下扫描采集
    hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT; //采集的数据又对齐
    hadc.Init.ContinuousConvMode = ENABLE; //连续采集
    hadc.Init.DiscontinuousConvMode = DISABLE;
    hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START; //代码触发ADC转换
    hadc.Init.DMAContinuousRequests = DISABLE;
    hadc.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
    hadc.Init.LowPowerAutoWait = DISABLE;
    hadc.Init.LowPowerFrequencyMode = DISABLE;
    hadc.Init.LowPowerAutoPowerOff = DISABLE;
    if (HAL_ADC_Init(&hadc) != HAL_OK)
    {
        Error_Handler();
    }

    /** Configure for the selected ADC regular channel to be converted. */
    sConfig.Channel = ADC_CHANNEL_0; //选择ADC采集的通道,也就是采集哪个引脚
    sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
    if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
    {
        Error_Handler();
    }
}
```

```

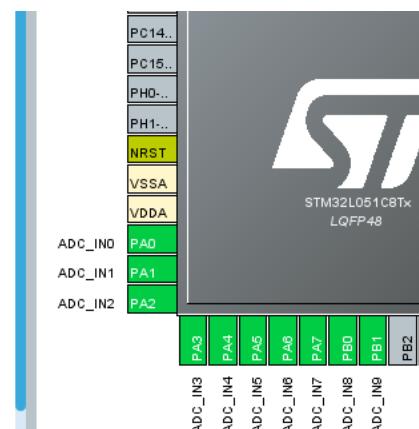
/** Configure for the selected ADC regular channel to be converted.
 */
// sConfig.Channel = ADC_CHANNEL_1;
// if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
// {
//   Error_Handler();
// }
// /** Configure for the selected ADC regular channel to be converted.
// */
// sConfig.Channel = ADC_CHANNEL_2;
// if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
// {
//   Error_Handler();
// }
// /** Configure for the selected ADC regular channel to be converted.
// */
// sConfig.Channel = ADC_CHANNEL_3;
// if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
// {
//   Error_Handler();
// }
// /** Configure for the selected ADC regular channel to be converted.

```



ADC DMA 实现多通道采集

- IN0
 - IN1
 - IN2
 - IN3
 - IN4
 - IN5
 - IN6
 - IN7
 - IN8
 - IN9
 - Temperature Sensor Channel
 - Vrefint Channel
- 如果要采集内部温度传感器，勾选这个
- 如果需要采集参考电压，选择这个



Clock Prescaler	Synchronous clock mode divided by 2
Resolution	ADC 12-bit resolution
Data Alignment	Right alignment
Scan Direction	Forward
Continuous Conversion Mode	Enabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	End of single conversion

Overrun behaviour	Overrun data preserved
Low Power Auto Wait	Disabled
Low Frequency Mode	Disabled
Auto Off	Disabled
Oversampling Mode	Disabled

ADC-Regular_ConversionMode

Sampling Time	1.5 Cycles
External Trigger Conversion Source	Regular Conversion launched by software
External Trigger Conversion Edge	None

WatchDog

Enable Analog WatchDog Mode

设置方式和单通道采集一样

DMA Request	Channel	Direction	Priority
ADC	DMA1 Channel 1	Peripheral To Memory	Low

Add Delete

DMA Request	Channel	Direction	Priority
ADC	DMA1 Channel 1	Peripheral To Memory	Low

Add Delete

DMA Request Settings

Mode: Normal	Increment Address: <input type="checkbox"/>	Peripheral: <input checked="" type="checkbox"/>	Memory: <input checked="" type="checkbox"/>
Data Width: Half Word	Length: Half Word		

HAL_StatusTypeDef HAL_ADC_Start_DMA(ADC_HandleTypeDef* hadc, uint32_t* pData, uint32_t Length) //启动 DMA ADC, 获取 ADC 数据

hadc: 传入 ADC 地址

* pData: 传入数组指针, 数组每一个下标表示一个 ADC 通道,

比如 pData[0] 表示 channel0

比如 pData[1] 表示 channel1

Length: 执行一次 HAL_ADC_Start_DMA 函数, 采集多少个通道的数据, 写 1 表示只采集 channel0

通道的数据给数组 0, 写 1 表示采集 channel0, channel1 数据给数组。注意这个 Length 长度是根据上一页 DMA 配置来的, 我内存配置的是 half word。

```
int main(void)
{
    uint16_t adc_value[10];

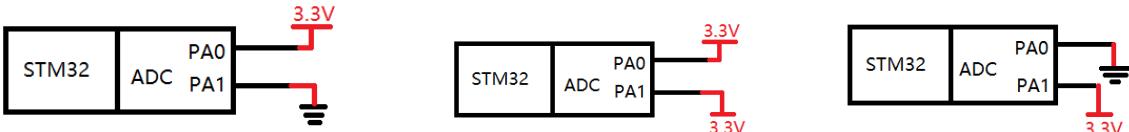
    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_ADC_Init();
    MX_USART1_UART_Init();

    while (1)
    {
        HAL_ADC_Start_DMA(&hadc, (uint32_t *)adc_value, 2);
        printf("channel 0 = %d\r\n", adc_value[0]);
        printf("channel 1 = %d\r\n", adc_value[1]);
        HAL_Delay(1000);
    }
}
```

这是采集通道 0, 通道 1 的数据



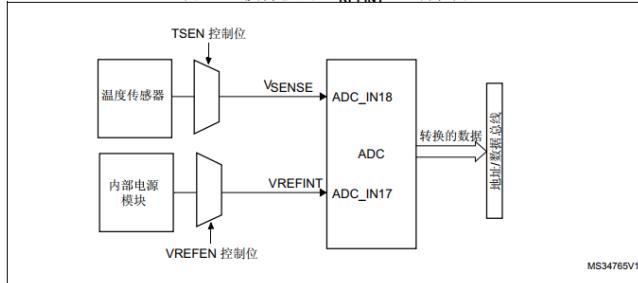
```
channel 0 = 4095
channel 1 = 57
channel 0 = 4095
channel 1 = 57
channel 0 = 4095
channel 1 = 57
channel 0 = 4095
channel 1 = 56
channel 0 = 4095
```

```
channel 1 = 4095
channel 0 = 4095
channel 1 = 4095
channel 0 = 4095
channel 1 = 4095
channel 0 = 4095
```

```
channel 0 = 56
channel 1 = 4095
channel 0 = 56
channel 1 = 4095
```

用 ADC DMA 获取温度传感器通道和内部参考电压通道的数据

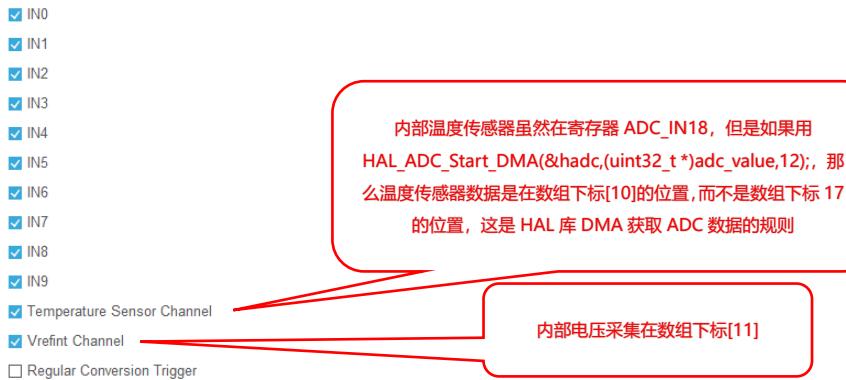
图 53. 温度传感器和 V_{REFINT} 通道方框图



这里虽然标准温度是通道 18，电压是通道 17. 实际

DMA 采集转换电压在数组下标[16]，温度在数组下标[17]

虽然温度传感器是在 ADC_IN18 和 ADC_IN17 通道，但是用 HAL 生成的 ADC DMA，我们是用数组获取。所以有些芯片的 ADC 通道只有 0 ~ 9，那么 HAL 库就是如下配置的：



```
int main(void)
{
    uint16_t adc_value[13];

    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_ADC_Init();
    MX_USART1_UART_Init();

    HAL_ADCEx_Calibration_Start(&hadc, ADC_SINGLE_ENDED);

    while (1)
    {
        HAL_ADC_Start_DMA(&hadc, (uint32_t *)adc_value, 12);
        for(uint16_t i = 0; i < 20 ; i++)
        {
            printf("channel %d = %d\r\n", i, adc_value[i]);
        }
        HAL_Delay(3000);
    }
}
```

读取 12 个 ADC 通道数据，看看有几个通道是正常运行

```
channel 0 = 45
channel 1 = 436
channel 2 = 4095
channel 3 = 2557
channel 4 = 1370
channel 5 = 799
channel 6 = 754
channel 7 = 725
channel 8 = 711
channel 9 = 710
channel 10 = 1463
channel 11 = 738
channel 12 = 0
channel 13 = 0
channel 14 = 0
channel 15 = 0
channel 16 = 0
channel 17 = 0
channel 18 = 0
channel 19 = 0
```

查看结果，确实温度传感器在数组下标[10]，证明要按照 HAL 库设置来确定 ADC 通道位置

根据通道配置 sConfig.Channel 有可能内部 VDDA 采集电压在通道 10，内部温度在通道 11.自行测试

ADC 过采样实现

过采样可以将低分辨率的ADC提高成高分辨率ADC：

传感器真实输出电压	白噪声电压	白噪声电压叠加到传感器输出	ADC采集到电压, 数字输出
0.5mV	1.2mV	1.7mV	1
0.5mV	0.6mV	1.1mV	1
0.5mV	-0.6mV	-0.1mV	0
0.5mV	-1.2mV	-0.7mV	0

传感器输出的电压很有可能混入环境噪声，所以我们要将环境白噪声考虑进去，ADC采集到的电压是环境白噪声叠加到传感器真实电压一起输出

我们将白噪声电压叠加到传感器输出的电压，用ADC采集出来，ADC采集4次 得到如下：

白噪声电压叠加到传感器输出
1.7mV
1.1mV
-0.1mV
-0.7mV

将4次电压加起来 = $1.7\text{mV} + 1.1\text{mV} + (-0.1)\text{mV} + (-0.7)\text{mV} = 2\text{mV}$

如果我们做平均滤波 $2\text{mV} / 4$ (采用次数) = 0.5mV

0.5mV 就是我们想要采样得到的值，但是注意

这是我们平时做滤波的方法，但是没有提高

ADC分辨率精度

这是因为我们平时做滤波的时候把小数部分去掉了，那么 0.5mV 取整数后就是 0，所以 ADC 数字量经过滤波之后就是输出的 0，其实 ADC 数字量是有电压的，只是被我们取整干掉了。

过采样就是需要保留小数部分

白噪声电压叠加到传感器输出
1.7mV
1.1mV
-0.1mV
-0.7mV

将4次电压加起来 = $1.7\text{mV} + 1.1\text{mV} + (-0.1)\text{mV} + (-0.7)\text{mV} = 2\text{mV}$

不是 $2\text{mV} / 4 = 0.5\text{mV}$

而是 $2\text{mV} / 2 = 1\text{mV}$ (保留小数)

换句话说就是把ADC的 0.5mV 放大了2倍，用更大的变量去承载

这时候 ADC 1 个 LSB , 一个位如果输出 0 表示 0.5mV, 如果一个位输出 1 表示 1mv

有了过采样 ADC采集数据过程是什么样的?

10位ADC										
ADC数据	9	8	7	6	5	4	3	2	1	0
寄存器	0	0	0	0	0	0	0	0	0	0

0.5mV电压 ---> ADC采集数字量 如果10位ADC
1个LSB分辨率是1mV, 那么 < 1mv ADC无法分辨

你看, 每个ADC寄存器位都是0

10位ADC										
ADC数据	9	8	7	6	5	4	3	2	1	0
寄存器	0	0	0	0	0	0	0	0	1	0

0.5mV电压 叠加白噪声, 采样4次 累加和 为2mv

如果使用常用的平均值滤波 就是 $2\text{mV} / 4 = 0.5\text{mv}$

10位ADC										
ADC数据	9	8	7	6	5	4	3	2	1	0
寄存器	0	0	0	0	0	0	0	0	0	0

0.5mV电压 叠加白噪声, $2\text{mV} / 4 = 0.5\text{mv}$
虽然电压滤波后是0.5mV, 但是我的ADC分辨率是1mV—一个LSB, 所以ADC输出数字量为0

10位ADC										
ADC数据	9	8	7	6	5	4	3	2	1	0
寄存器	0	0	0	0	0	0	0	0	1	0

0.5mV电压 叠加白噪声, 采样4次 累加和 为2mv

如果使用常用的平均值滤波 就是 $2\text{mV} / 4 = 0.5\text{mv}$

10位ADC										
ADC数据	9	8	7	6	5	4	3	2	1	0
寄存器	0	0	0	0	0	0	0	0	0	1

0.5mV电压 叠加白噪声,
如果用过采样 $2\text{mV} / 2 = 1\text{mV}$
也就是>>1右移1位

这个右移 1 位得到的最低位就作为 11 位 ADC 的值, 所以这个 1 就表示 0.5mV。其实就把 0.5mV 放大到 1mV, 方便 1mV 分辨率的 10 位 ADC 采样。

过采样最大能提升多少精度?

ADC 位数	采样次数	每秒采样次数
12	1	1M

13	4	250K
14	16	62.5k
15	64	15.6K
16	256	3.9K
17	1024	976
18	4096	244
19	16384	61
20	65536	15

比如我是 12 位 ADC 每增加 1 位分辨率，就会增加 4 的倍数采样次数，采样速度从软件角度就会降低。
如果我是 10 位 ADC，也是一样的每增加 1 位分辨率，就会增加 4 的倍数采样次数，采样速度从软件角度就会降低。

C 语言过采样代码实现

```
/****************************************************************************
**
** Function name:      adc_deal
** Descriptions:       AD 过采样计算 12 位提升 16 位过采样
** input parameters:  无
** output parameters: 无
** Returned value:    无
**
*/
u8 adc_deal(AD_PARA* adpara)
{
    u16 i,j;

    ADC_ok=0;
    for(j=0;j<3;j++)
    {
        for(i=0;i<256;i++)
        {
            samp[j] +=adc_buff[i][j]; //先累加 N 次 12 位的采样值
        }
        samp[j] >>= 4; //右移 4 位就是 16 位
        cysamp[j]+=samp[j];
        samp[j]=0;
    }
    cyten++;
}

if (cyten==2)
{
    for(j=0;j<3;j++)
}
```

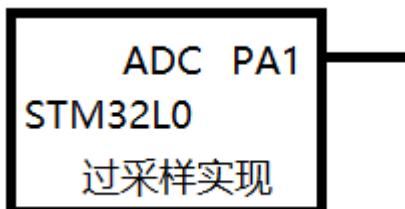
```

    {
        cysample[j]=cysamp[j]/2;
        cysamp[j]=0;
    }
    cyten=0;

    for(j=0;j<3;j++)
    {
        adpara->TRUTHVAL = adpara->SAMPVAL = filter(ad_samp[j],cysample[j]);
        adpara++;
    }
    return(1);
}
return(0);
}

```

STM32 硬件过采样实现



我们只测试 PA1 通道

ADC 其余设置和 ADC_DMA 章节一样

使能过采样功能

过采样率设置 128

>>3 右移 3 位, 精度从 12 位提升到 16 位, 在 STM32 从 0 算就是 16 位精度, 其余芯片不一定, 应该>>4 位才是 16 位精度

生成代码

```

int main(void)
{
    uint16_t adc_value[13];
    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_ADC_Init();
    MX_USART1_UART_Init();

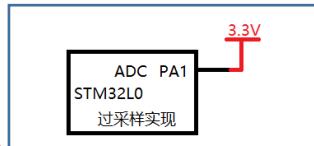
    HAL_ADCEx_Calibration_Start(&hadc, ADC_SINGLE_ENDED);

    while (1)
    {
        HAL_ADC_Start_DMA(&hadc, (uint32_t *)adc_value, 12);

        for(uint16_t i = 0; i < 20 ;i++)
        {
            printf("channel %d = %d\r\n", i, adc_value[i]);
        }
    }
}

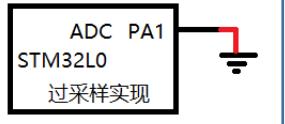
```

channel 0 = 3994
 channel 1 = 65519
 channel 2 = 10367
 channel 3 = 65519
 channel 4 = 10019
 channel 5 = 65519
 channel 6 = 9959
 channel 7 = 65520
 channel 8 = 9933
 channel 9 = 65520
 channel 10 = 10069
 channel 11 = 65519
 channel 12 = 0



1 通道 PA1 引脚接的高电平, 采集到最大电压 65519

channel 0 = 3978
 channel 1 = 3
 channel 2 = 3987
 channel 3 = 4
 channel 4 = 3929
 channel 5 = 5
 channel 6 = 3950
 channel 7 = 6
 channel 8 = 4019
 channel 9 = 6
 channel 10 = 3952
 channel 11 = 4
 channel 12 = 0



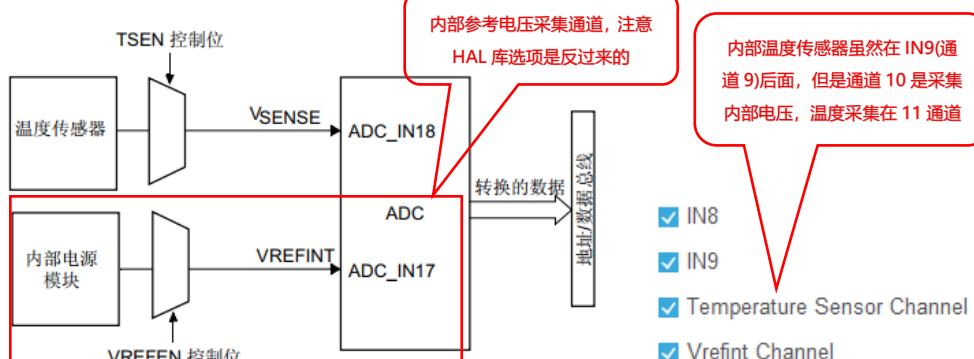
1 通道 PA1 引脚接的低电平, 采集到最大电压 3, 相当于地

这就是 ADC 过采样, 我们只使能了通道 1, 其余通道的值是误码, 不需要理会。

将得到的过采样值累加 10 次, 进行除以 10 平均, 就得到 16 位稳定电压, 记住是累加过采样之后的值, 而不是我们以前那样过采样之前的值累加。

STM32 ADC 用内部电压, 采集芯片供电电压, 监控芯片电源

图 53. 温度传感器和 V_{REFINT} 通道方框图



所以 通道 10 采集的是内部电压值, 使用这个内部电压去估算 VDDA 引脚的电压, 如果电源电压 3.3V 同时给 VDDA 供电, 那么内部电压就可以反算芯片供电电压.

```

IN Channel Valotage 0 = 0.219141
IN Channel Valotage 1 = 0.421362
IN Channel Valotage 2 = 0.546240
IN Channel Valotage 3 = 0.600220
IN Channel Valotage 4 = 0.357715
IN Channel Valotage 5 = 0.284399
IN Channel Valotage 6 = 0.475342
IN Channel Valotage 7 = 3.299194
IN Channel Valotage 8 = 2.383154
IN Channel Valotage 9 = 1.786157
IN Channel Valotage 10 = 1.290674
IN Channel Valotage 11 = 0.604248

```

获取通道 10 的电压(内部电压)

以下公式可求得为器件供电的实际的 V_{DDA} 电压:

$$V_{DDA} = 3 \text{ V} \times VREFINT_CAL / VREFINT_DATA$$

3V : 固定值

$VREFINT_CAL$ 寄存器,芯片出厂内部电压校准值,直接获取即可.

$VREFINT_DATA$: 通道 10 获取的参考电压数字量 , ($VDDA$ 电压下降,那么通道 10 参考电压也会下降, 用这种方式反算电源电压)

Table 8. Internal voltage reference measured values

Calibration value name	Description	Memory address
$VREFINT_CAL$	Raw data acquired at temperature of 25 °C $V_{DDA} = 3 \text{ V}$	0x1FF8 0078 - 0x1FF8 0079

0x1ff8 0078 地址上的数据是 8 位的 , 要用 8 位变量获取.

0x1ff8 0079 地址上的数据是 8 位的 , 要用 8 位变量获取.

```

uint8_t data1 = 0, data2 = 0;
data1 = *(__IO uint8_t *)0x1FF80078); //获取低位
data2 = *(__IO uint8_t *)0x1FF80079); //获取高位
uint16_t VREFINT_CAL = 0;

```

```

VREFINT_CAL = data2; //高位写入
VREFINT_CAL = (uint16_t)((VREFINT_CAL << 8) | data1); //低位写入

```

$$V_{DDA} = 3 \text{ V} \times VREFINT_CAL / VREFINT_DATA$$

uint16_t VREINT = 0; //获取基准电压

VREINT = get_ADC_Channel_Value(10); //获取通道 10 电压, get_ADC_Channel_Value 自己封装的函数

```

float vref = 0, ret = 0;
float LsbVolatge = 0; //每个 LSB 电压值
LsbVolatge = (3.3/4096);

```

```

ret = (float)(3 * VREFINT_CAL); //得到校准电压,强转 float, 得到小数
vref = (float)(ret/VREINT); //得到 VDDA 电压

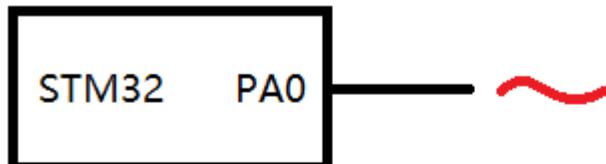
```

```

vref = 3.467641
vref = 3.465230
vref = 3.462821
vref = 3.460417 反算 VDDA 电压 , 结果正确.

```

在使用 STM32 内部 VREFINT 做基准电压反算 VDDA 电压时, 或者启动 ADC 多个通道采集时一定要注意采样时间大小问题 (CYCLES) , ADC_SAMPLETIME_1CYCLE_5 1.5 个 CYCLES 整了我一天。



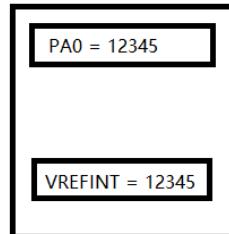
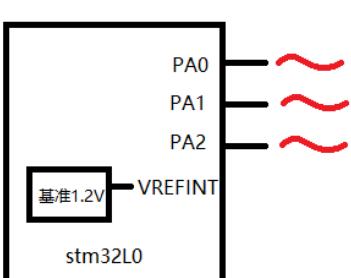
整个单片机只使用1个通道采集数据

那么在ADC初始化函数中设置

```
hadcDrv.Init.SamplingTime = ADC_SAMPLETIME_1CYCLE_5
```

设置1.5个采样时间没有问题

如果 ADC 有多个通道采集, 比如我这次用到了 VREFINT 通道 ADC_CHANNEL_17, 还有其它通道



就会出现PA0通道电压变化, VREFINT采集的基准电压也跟着变化, 其实不是基准电压变了, 而是VREFINT采集速度太快, 没来得及建立采样周期导致的

如果还是在ADC初始化中使用

```
hadcDrv.Init.SamplingTime = ADC_SAMPLETIME_1CYCLE_5
```

所以 STM32 要高速采集, 就只能用一个 ADC 通道。

如果要多通道采集, 那么就必须降低 CYCLE 采样时间, 将系统变成低速采集。

```

/** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
 */
hadcDrv.Instance = ADC1;
hadcDrv.Init.OversamplingMode = DISABLE; //取消过采样
hadcDrv.Init.Oversample.Ratio = ADC_OVERSAMPLING_RATIO_128;
hadcDrv.Init.Oversample.RightBitShift = ADC_RIGHTBITSHIFT_3;
hadcDrv.Init.Oversample.TriggeredMode = ADC_TRIGGEREDMODE_SINGLE_TRIGGER;
hadcDrv.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2; //在CPU MSI 2M时钟驱动下不分频ADC_CLOCK_SYNC_PCLK_DIV1
hadcDrv.Init.Resolution = ADC_RESOLUTION_12B; //分辨率12bit 满刻度4096
hadcDrv.Init.SamplingTime = ADC_SAMPLETIME_79CYCLES_5; //ADC_SAMPLETIME_1CYCLE_5 //多通道采集不能用1.5个CYCLE
hadcDrv.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD; //向下扫描

```

hadcDrv.Init.SamplingTime = ADC_SAMPLETIME_79CYCLES_5; 将采样时间降低到 79.5 个周期, 就可以多通道采集, 如果 ADC 通道更多, 那么就要尝试设置更慢的采样时间, 如 ADC_SAMPLETIME_160CYCLES_5

经过以上设置问题得到解决

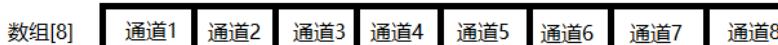
在 ADC DMA 中，多个 ADC 通道采集，但是通道不是连续的，是跳跃的，比如只采集 1, 3, 5 通道，那么 DMA 缓存是按照 5 个数组元素依次摆放的吗？，如 1, 2, 3, 4, 5，我们用数组下标取 1, 3, 5 就能获取 ADC 1,3,5 通道的数据？其实这是不行的

```
HAL_ADC_Start_DMA(&hadcDrv, (uint32_t *)xadc_value, 12);
HAL_Delay(1);
HAL_ADC_Stop_DMA(&hadcDrv);

printf("array[0] = %d\r\n", xadc_value[0]);
printf("array[1] = %d\r\n", xadc_value[1]);
printf("array[2] = %d\r\n", xadc_value[2]);
```

HAL_ADC_Start_DMA(&hadcDrv, (uint32_t *)xadc_value, 12); 如果我采集 12 个 ADC 通道
但是我初始化的时候是 1, 3, 5 通道，那么 1, 3, 5 通道数据如何缓存的？

HAL_ADC_Start_DMA(..., xadc_value, 12) //DMA采集的1, 3, 5通道
那么在缓存中是这样存放的吗



其实不是

实际存放方式如下：



```
HAL_ADC_Start_DMA(&hadcDrv, (uint32_t *)xadc_value, 12);
HAL_Delay(1);
HAL_ADC_Stop_DMA(&hadcDrv);

printf("array[0] = %d\r\n", xadc_value[0]);
printf("array[1] = %d\r\n", xadc_value[1]);
printf("array[2] = %d\r\n", xadc_value[2]);
```

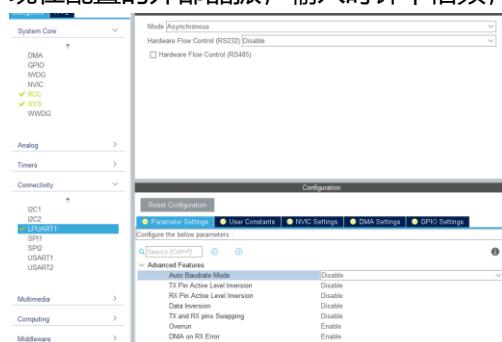
第一次启动 DMA 和第 2 次启动 DMA 之间必须延时，不然 PA1 采用不准。停止 DMA 也可以加上

所以 ADC DMA 的数组数据获取方式不是跟着通道来，而是根据通道数量多少，从第 0 个元素开始排列

LPUART1 低功耗串口使用

LPUART1 常规功耗模式

现在配置的外部晶振，输入时钟不倍频，系统时钟 8M



配置 LPUART1 串口常规参数

NVIC Interrupt Table		Enabled	Preemption Priority
LPUART1 global interrupt / LPUART1 wake-up interrupt through EXTI line 28		<input checked="" type="checkbox"/>	0

使能 LPUART1 中断

NVIC Interrupt Table		Enabled	Preemption Priority
LPUART1 global interrupt / LPUART1 wake-up interrupt through EXTI line 28		<input checked="" type="checkbox"/>	1

设置串口中断优先级 1

Pin Name	Signal on Pin	GPIO output	GPIO mode	GPIO Pull-up/Pull-down	Maximum output speed	Fast Mode	User Label	Modified
PB10	LPUART1_TX	n/a	Alternate Function	Pull-down	Low	n/a		<input checked="" type="checkbox"/>
PB11	LPUART1_RX	n/a	Alternate Function	Pull-down	Low	n/a		<input checked="" type="checkbox"/>

因为是低功耗串口，我们设置 IO 口下拉，因为上拉会消耗电流，串口 IO 口的速度为低速模式

```
int main(void)
{
    /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
    HAL_Init();

    /* Configure the system clock */
    SystemClock_Config();

    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_LPUART1_UART_Init();

    uint8_t data = 0x1;

    while (1)
    {
        HAL_UART_Transmit(&hlpuart1, &data, 1, 1); //发送数据
        HAL_Delay(500);
    }
    /* USER CODE END 3 */
}
```

发送测试成功

LPUART1 常规功耗接收数据测试



代码例程

```
int main(void)
{
    HAL_Init();
    /* Configure the system clock */
    SystemClock_Config();
    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_USART1_UART_Init(); //初始化打印串口 115200
    MX_LPUART1_UART_Init(); //初始化 LPUART1 串口 115200
    HAL_UART_Receive_IT(&huart1,&pdata,1); //因为初始化没有第一次使能中断接收，所以这儿必须使能
    HAL_UART_Receive_IT(&hlpuart1,&pdata,1); //因为初始化没有第一次使能中断接收，所以这儿必须使能

    printf("xxxxzzzz\r\n");

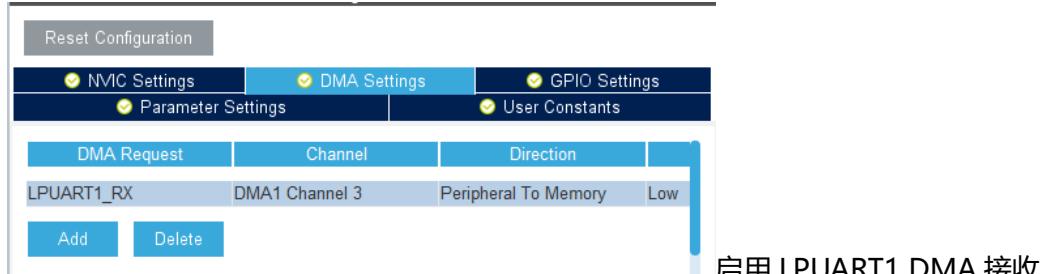
    int time = 0;
    uint8_t data = 0xa1;
    while (1)
    {
        printf("time = %d\r\n",time++);
        HAL_Delay(1000);
    }
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle) //LPUART1 中断发生后会调用这个回调函数
{
    if(UartHandle->Instance == USART1) //所以我必须判断是不是串口 1 接收中断
    {
        printf("Rx Callback uart1 IRQ.\r\n");
        HAL_UART_Receive_IT(&huart1,&pdata,1); //重新开启串口接收中断
    }
    else if(UartHandle->Instance == LPUART1)
    {
        HAL_UART_Transmit(&hlpuart1,&pdata,1,1);
        while(HAL_UART_GetState(&hlpuart1) == HAL_UART_STATE_BUSY_TX); //检测 UART 发送结束
        HAL_UART_Receive_IT(&hlpuart1,&pdata,1); //重新开启串口接收中断
    }
    else
    {
    }
}
}
```

我现在发现 LPUART1 接收一个字节会触发两次中断，虽然收到了数据，但是丢字节，不正常。

下面使用空闲中断来解决这个问题

LPUART1 + DMA + 空闲中断接收数据



代码例程如下：

```
int main(void)
{
    HAL_Init();

    /* Configure the system clock */
    SystemClock_Config();

    /* Initialize all configured peripherals */
    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART1_UART_Init();
    MX_LPUART1_UART_Init();

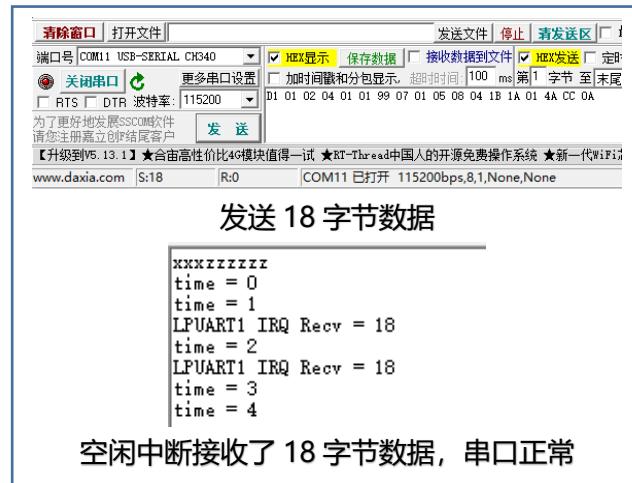
    /* USER CODE BEGIN 2 */
    HAL_UART_Receive_IT(&huart1,&pdata,1); //打印串口中断
    _HAL_UART_ENABLE_IT(&hluart1, UART_IT_IDLE); //开启 LPUART1 空闲中断
    HAL_UART_Receive_DMA(&hluart1,RecvBuffer,512); //LPUART1 空闲中断的 DMA 开启
    printf("xxxxxxxx\r\n");

    int time = 0;
    while (1)
    {
        printf("time = %d\r\n",time++);
        HAL_Delay(1000);
    }
}

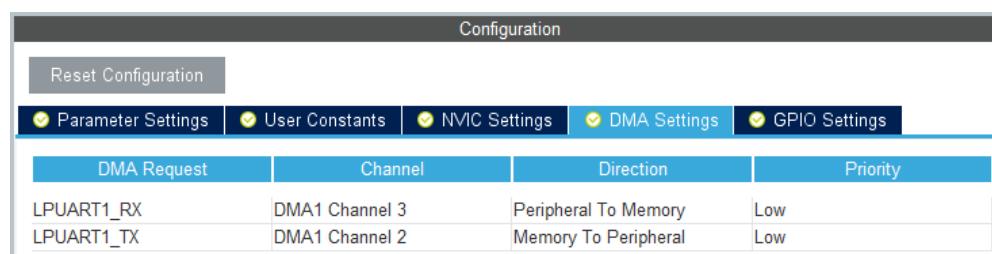
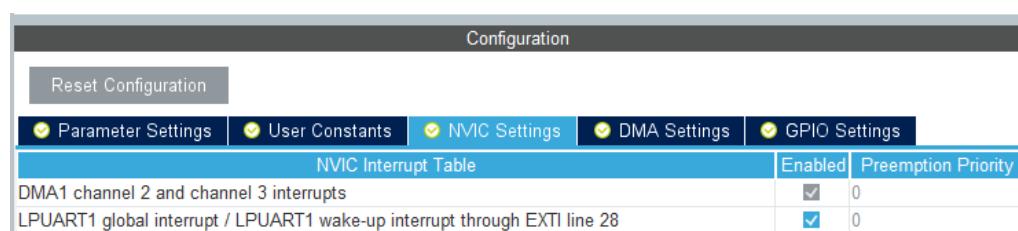
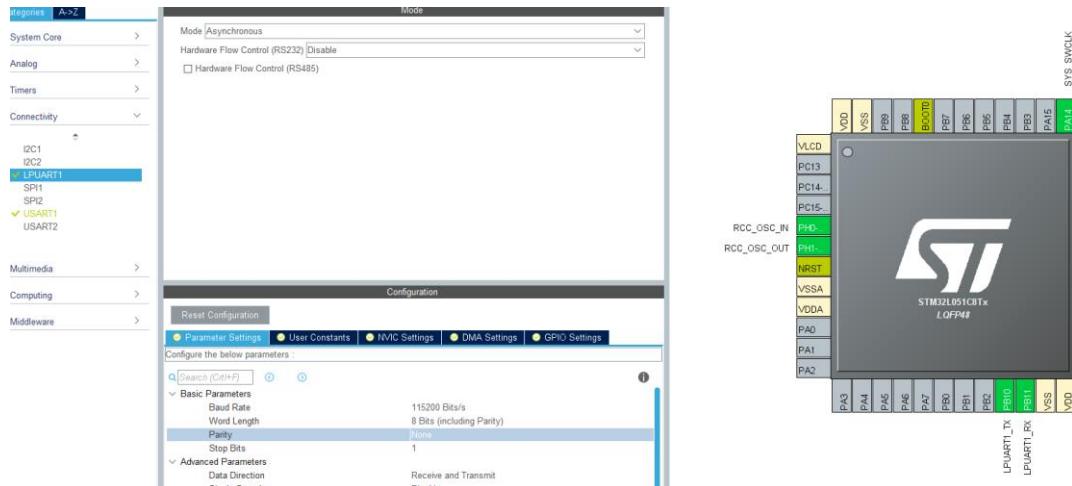
extern unsigned char RecvBuffer[512];

void LPUART1_IRQHandler(void) //DMA 空闲中断, 必须再中断函数中进行接收数据和计算
{
    uint16_t temp = 0;
    uint16_t RxLen = 0;
    /* USER CODE END LPUART1_IRQHandler 0 */
    HAL_UART_IRQHandler(&hluart1);
    /* USER CODE BEGIN LPUART1_IRQHandler 1 */
    if(_HAL_UART_GET_FLAG(&hluart1,UART_FLAG_IDLE) == SET) //触发空闲中断
    {
        _HAL_UART_CLEAR_IDLEFLAG(&hluart1); //清除串口 2 空闲中断标志位
        HAL_UART_DMAStop(&hluart1); //关闭 DMA
        // temp = huart2.Instance->ISR; //有些人说 _HAL_UART_CLEAR_IDLEFLAG 无法清除空闲中断
        // temp = huart2.Instance->RDR; //必须读一下 SR 和 DR 才能清除, 我这里没发现任何问题
        temp = hdma_lpuart1_rx.Instance->CNDTR; //获取 DMA 中未传输的数据个数
        RxLen = 512 - temp; //用自定义 buffer 的总大小 减去 未传输个数的大小, 就是接收到实际数据的大小
        HAL_UART_Receive_DMA(&hluart1,RecvBuffer,512); //重新打开 DMA 接收, 类似前面普通串口的 Receive_IT
        printf("LPUART1 IRQ Recv = %d\r\n",RxLen);
    }
    /* USER CODE END LPUART1_IRQHandler 1 */
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle) //保留串口中断的钩子函数
{
    if(UartHandle->Instance == USART1) //所以我必须判断是不是串口 1 接收中断
    {
        printf("Rx Callback uart1 IRQ.\r\n");
        HAL_UART_Receive_IT(&huart1,&pdata,1); //重新开启串口接收中断
    }
    else if(UartHandle->Instance == LPUART1)
    {
    }
    else
    {
    }
}
```



LPUART1 低功耗唤醒单片机



USART1 global interrupt / USART1 wake-up interrupt through EXTI line 25	<input checked="" type="checkbox"/>	3
LPUART1 global interrupt / LPUART1 wake-up interrupt through EXTI line 28	<input checked="" type="checkbox"/>	3

借鉴上一节 LPUART1 + DMA + 空闲中断的代码不变。加入低功耗休眠功能，看看 LPUART1 是否能唤醒设备。

低功耗睡眠模式 LPUART1 唤醒测试

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();

    MX_GPIO_Init();
    MX_DMA_Init();
    MX_USART1_UART_Init();
    MX_LPUART1_UART_Init();

    HAL_UART_Receive_IT(&huart1,&pdata,1); //打印串口中断
    _HAL_UART_ENABLE_IT(&hluart1,UART_IT_IDLE); //开启 LPUART1 空闲中断
    HAL_UART_Receive_DMA(&hluart1,RecvBuffer,512); //LPUART1 空闲中断的 DMA 开启
    printf("xxxxxxxx\r\n");
}
```

```

int time = 0;
while (1)
{
    printf("time = %d\r\n",time++);

    if(time > 5)
    {
        HAL_UART_DMAPause(&hlpuart1);
        __HAL_RCC_PWR_CLK_ENABLE();
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗运行模式
        HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI); //进入睡眠模式
        HAL_ResumeTick();
    }
    HAL_Delay(1000);
}
}

```

```

xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
LPUART1 IRQ Recv = 18
time = 5

```

在设备进入睡眠模式之前，LPUART1 串口能接受数据。

```

LPUART1 IRQ Recv = 0
LPUART1 IRQ Recv = 0
time = 6
LPUART1 IRQ Recv = 0
LPUART1 IRQ Recv = 0
time = 7
LPUART1 IRQ Recv = 0
time = 8
LPUART1 IRQ Recv = 0
time = 9

```

设备进入低功耗之后，LPUART1 发送数据能够唤醒设备。但是 LPUART1 收不到真实数据了。我怀疑这是没有重新启动时钟造成的。

其实取消掉暂停 DMA 函数 HAL_UART_DMAPause(&hlpuart1)就可以了。

```

int time = 0;
while (1)
{
    printf("time = %d\r\n",time++);

    if(time > 5)
    {
//        HAL_UART_DMAPause(&hlpuart1); //取消暂停 DMA
//        __HAL_RCC_PWR_CLK_ENABLE();
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗运行模式
        HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI); //进入睡眠模式
        HAL_ResumeTick();

        printf("Wake Up\r\n"); //低功耗唤醒接着执行
    }
    HAL_Delay(1000);
}

```

```

Wake Up
LPUART1 IRQ Recv = 18
LPUART1 IRQ Recv = 18
LPUART1 IRQ Recv = 18
time = 17
LPUART1 IRQ Recv = 19
Wake Up
LPUART1 IRQ Recv = 19
LPUART1 IRQ Recv = 19
time = 18

```

设备休眠后，接收一帧 LPUART1 的数据，就真实获取

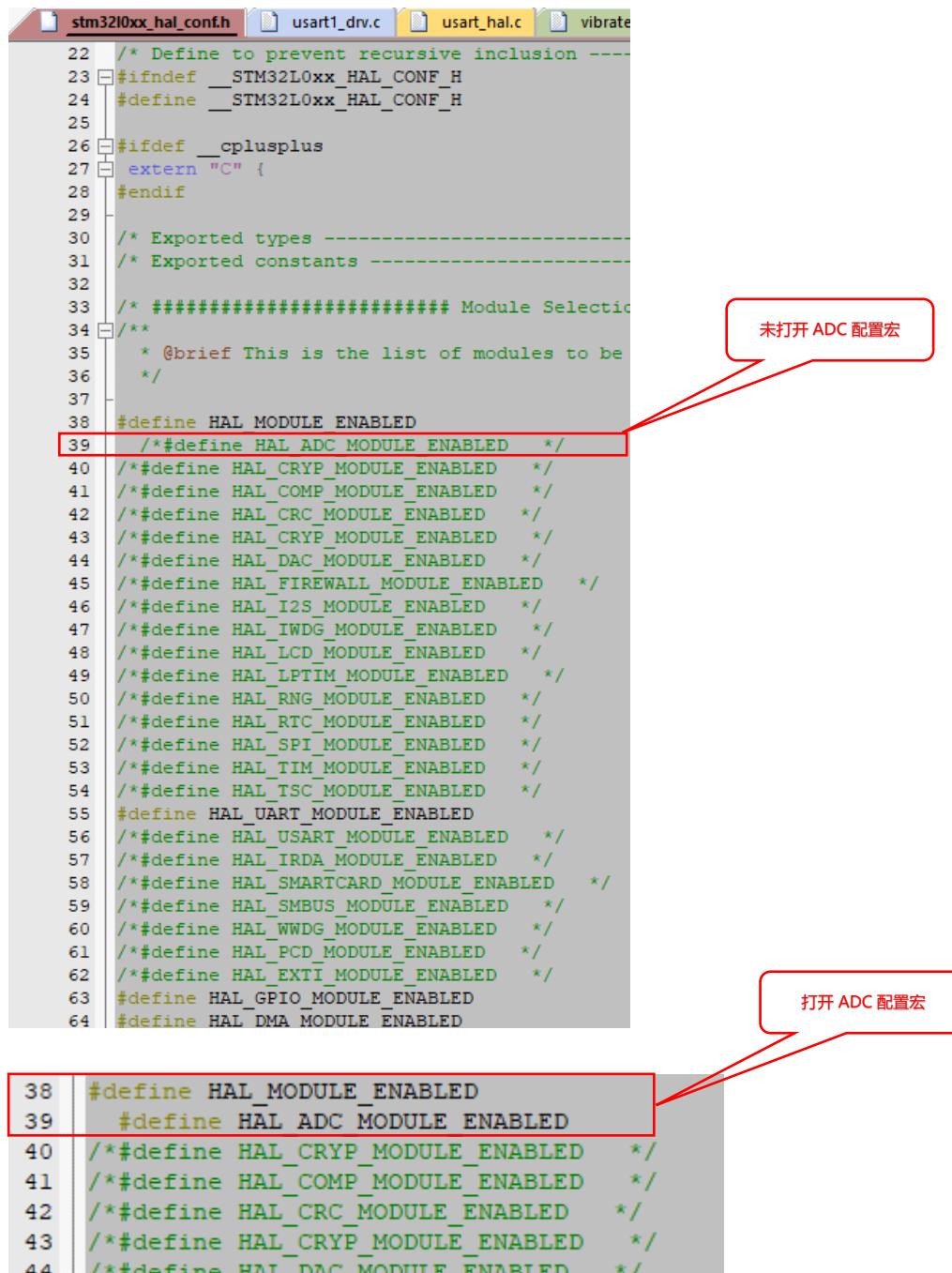
当移植一个用 HAL 生成好的驱动程序到另外一个 L051 工程的时候，注意配置文件

报错: 出现 identifieris undefined 未定义某个驱动函数, (重点)

```
\driver\adc_driver.c(320): error: #20: identifier "ADC_CHANNEL_VREFINT" is undefined
```

我现在就以移植 ADC 为例来说明这个问题

当把其余工程 cubemx 配置好的 ADC 移植到当前工程的时候注意事项如下



```
22 /* Define to prevent recursive inclusion -----  
23 ifndef __STM32L0xx_HAL_CONF_H  
24 define __STM32L0xx_HAL_CONF_H  
25  
26 ifdef __cplusplus  
27 extern "C" {  
28 endif  
29  
30 /* Exported types -----  
31 /* Exported constants -----  
32  
33 ##### Module Selection -----  
34 /*  
 * @brief This is the list of modules to be  
 */  
35  
36 #define HAL_MODULE_ENABLED  
37 /*#define HAL_ADC_MODULE_ENABLED */  
38 /*#define HAL_CRYP_MODULE_ENABLED */  
39 /*#define HAL_COMP_MODULE_ENABLED */  
40 /*#define HAL_CRC_MODULE_ENABLED */  
41 /*#define HAL_CRYP_MODULE_ENABLED */  
42 /*#define HAL_DAC_MODULE_ENABLED */  
43 /*#define HAL_FIREWALL_MODULE_ENABLED */  
44 /*#define HAL_I2S_MODULE_ENABLED */  
45 /*#define HAL_IWDG_MODULE_ENABLED */  
46 /*#define HAL_LCD_MODULE_ENABLED */  
47 /*#define HAL_LPTIM_MODULE_ENABLED */  
48 /*#define HAL_rng_MODULE_ENABLED */  
49 /*#define HAL_RTC_MODULE_ENABLED */  
50 /*#define HAL_SPI_MODULE_ENABLED */  
51 /*#define HAL_TIM_MODULE_ENABLED */  
52 /*#define HAL_TSC_MODULE_ENABLED */  
53  
54 #define HAL_UART_MODULE_ENABLED  
55  
56 /*#define HAL_USART_MODULE_ENABLED */  
57 /*#define HAL_IRDA_MODULE_ENABLED */  
58 /*#define HAL_SMARTCARD_MODULE_ENABLED */  
59 /*#define HAL_SMBUS_MODULE_ENABLED */  
60 /*#define HAL_WWDG_MODULE_ENABLED */  
61 /*#define HAL_PCD_MODULE_ENABLED */  
62 /*#define HAL_EXTI_MODULE_ENABLED */  
63  
64 #define HAL_GPIO_MODULE_ENABLED  
#define HAL_DMA_MODULE_ENABLED
```

未打开 ADC 配置宏

打开 ADC 配置宏

```
38 #define HAL_MODULE_ENABLED  
39 #define HAL_ADC_MODULE_ENABLED  
40 /*#define HAL_CRYP_MODULE_ENABLED */  
41 /*#define HAL_COMP_MODULE_ENABLED */  
42 /*#define HAL_CRC_MODULE_ENABLED */  
43 /*#define HAL_CRYP_MODULE_ENABLED */  
44 /*#define HAL_DAC_MODULE_ENABLED */
```

然后进行编译

```
linking...  
..\..\out\WSNV1V0.axf: Error: L6218E: Undefined symbol HAL_ADC_ConfigChannel (referred from adc_driver.o).  
..\..\out\WSNV1V0.axf: Error: L6218E: Undefined symbol HAL_ADC_Init (referred from adc_driver.o).  
..\..\out\WSNV1V0.axf: Error: L6218E: Undefined symbol HAL_ADC_Start_DMA (referred from adc_driver.o).  
Not enough information to list image symbols.
```

编译过程中发现找不到 HAL 库里面的 ADC 函数，这是因为遗漏了其它工程用 HAL 生成好的 ADC 文件或者自己 Drivers 目录下的 ADC 驱动文件没有加入进工程

```
+--- stm32l0xx_hal_adc_ex.c  
+--- stm32l0xx_hal_adc.c
```

就是这两个 h 文件和 c 文件

将这两个文件加入到工程中。进行编译。

```
linking...  
..\..\out\WSNV1V0.axf: Error: L6200E: Symbol Error_Handler multiply defined (by adc_driver.o and main.o).
```

编译过程中发现自己写的 ADC 驱动文件有重名现象，其实不是 ADC 驱动的问题。

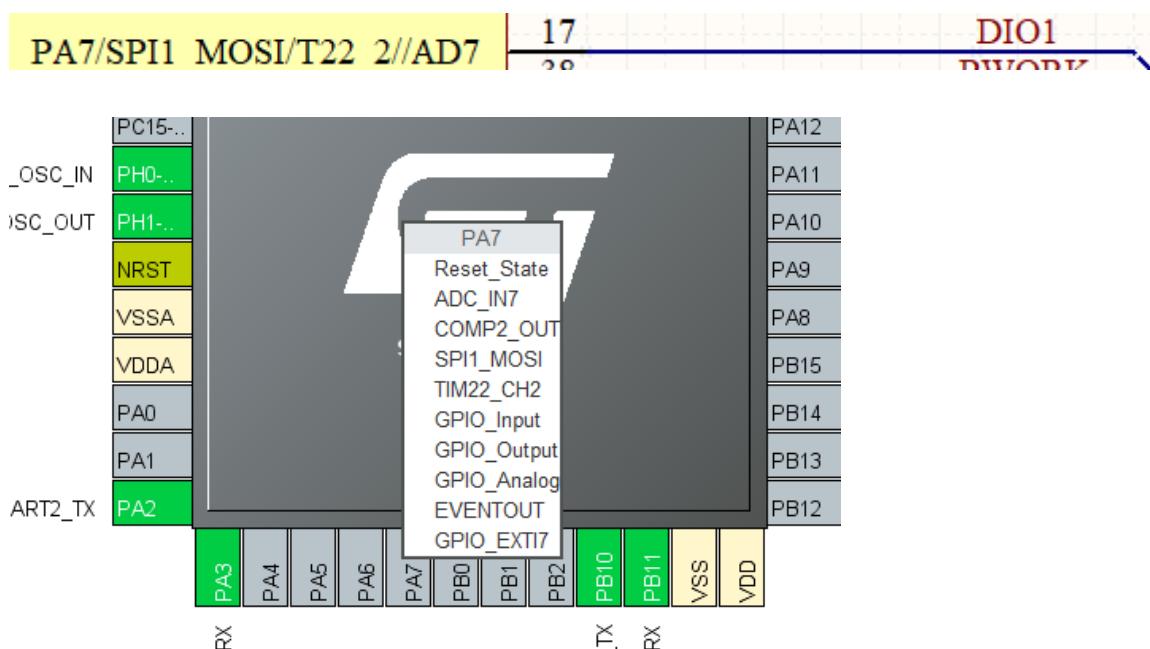
```
void Error_Handler(void)  
{  
    /* USER CODE BEGIN Error_Handler_Debug */  
    /* User can add his own implementation to report the HAL error return state */  
  
    /* USER CODE END Error_Handler_Debug */  
}
```

就是这个 Error_Handler 重名了。因为驱动初始化在错误情况下会调用该函数。所以我加入了该函数。但是本工程中 main.h 已经实现了 Error_Handler。所以我这里写多余了。

```
//void Error_Handler(void)  
//{  
//    /* USER CODE BEGIN Error_Handler_Debug */  
//    /* User can add his own implementation to report the HAL error return state */  
  
//    /* USER CODE END Error_Handler_Debug */  
//}  
  
void Error_Handler(void);
```

只声明不实现就 OK 了，因为这个 Error_Handler 是全局的。

IO 管脚外部中断实现



System Core

- DMA
- GPIO**
- IWDG
- NVIC
- RCC**
- SYS**
- WWDG

Analog

- ADC**
- COMP1
- COMP2

Timers

- LPTIM1**
- RTC
- TIM2**
- TIM6
- TIM21
- TIM22

Connectivity

- I2C1

GPIO

Search Signals

Search (Ctrl+F)

Group By Peripherals

GPIO LPUART1 RCC SYS USART1 USART2 NVIC

Pin Name	Signal on Pin	GPIO outp...	GPIO mode	GPIO Pull...	Maximum o...	Fast Mode	User Label	Modified
PA7	n/a	n/a	External Int...	Pull-up	n/a	n/a		<input checked="" type="checkbox"/>

1.选择 GPIO

PA7 Configuration :

GPIO mode : External Interrupt Mode with Rising edge trigger detection

GPIO Pull-up/Pull-down : Pull-up

User Label :

2.选择外部中断上升沿触发

3.PA7 引脚默认上拉电阻

NVIC

- RCC**
- SYS**
- WWDG

Analog

- ADC**
- COMP1
- COMP2

Timers

- LPTIM1**
- RTC
- TIM2**
- TIM6
- TIM21
- TIM22

Connectivity

- I2C1

NVIC Interrupt Table

	Enabled	Preemption Priority
Non maskable interrupt	<input checked="" type="checkbox"/>	0
Hard fault interrupt	<input checked="" type="checkbox"/>	0
System service call via SWI instruction	<input checked="" type="checkbox"/>	0
Pendable request for system service	<input checked="" type="checkbox"/>	0
Time base: System tick timer	<input checked="" type="checkbox"/>	0
PVD interrupt through EXTI line 16	<input type="checkbox"/>	0
Flash and EEPROM global interrupt	<input type="checkbox"/>	0
RCC global interrupt	<input type="checkbox"/>	0
EXTI line 4 to 15 interrupts	<input checked="" type="checkbox"/>	3
DMA1 channel 2 and channel 3 interrupts	<input checked="" type="checkbox"/>	0
DMA1 channel 4, channel 5, channel 6 and channel 7 interrupts	<input checked="" type="checkbox"/>	0
USART1 global interrupt / USART1 wake-up interrupt through EXTI line 25	<input checked="" type="checkbox"/>	1
USART2 global interrupt / USART2 wake-up interrupt through EXTI line 26	<input checked="" type="checkbox"/>	2
LPUART1 global interrupt / LPUART1 wake-up interrupt through EXTI line 28	<input checked="" type="checkbox"/>	3

4.PA7 外部中断优先级设置

5.PA7 在外部中断线 4~15, 中断优先级选 3

Enabled Preemption Priority 3

在使用 cubemx 生成新代码的时候, 如何保证以前工程的代码不被覆盖?(重点)

Home > STM32L051C8Tx > WSNSoftwareFrame2V0.ioc - Project Manager

Pinout & Configuration | Clock Configuration | Project

Project

STM32Cube MCU packages and embedded software packs

Copy all used libraries into the project folder
 Copy only the necessary library files
 Add necessary library files as reference in the toolchain project configuration file

Generated files

Generate peripheral initialization as a pair of '.c/.h' files per peripheral
 Backup previously generated files when re-generating
 Keep User Code when re-generating
 Delete previously generated files when not re-generated

在生成新代码的时候这个必须选上

Code Generator

HAL Settings

Set all free pins as analog (to optimize the power consumption)

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_LPUART1_UART_Init();
MX_USART1_UART_Init();
MX_USART2_UART_Init();
MX_RTC_Init();
/* USER CODE BEGIN 2 */

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    /*1111111111111111 这一段就是我原来工程的代码*/
}
/* USER CODE END 3 */
}
```

以前工程的代码必须卸载 USER CODE BEGIN 和 USER CODE END 之间, 必须这样, 写在其它地方都不行

这样重新生成代码之后, 以前工程的代码不会被覆盖。

```
/* USER CODE END 2 */

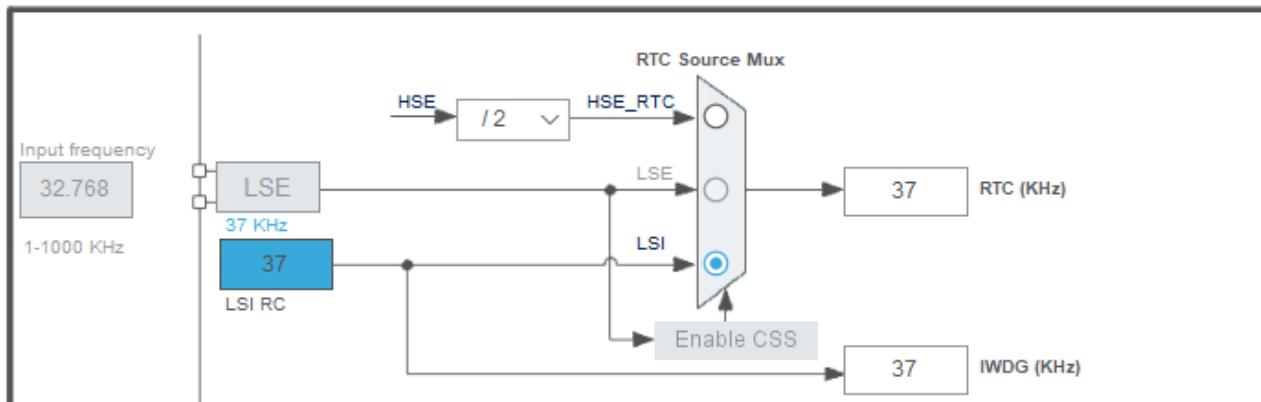
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    /* USER CODE END WHILE */
    /*1111111111111111 这一段就是我原来工程的代码*/
    /* USER CODE BEGIN 3 */

}
/* USER CODE END 3 */
}
```

写在这个位置都不行, 必须是 BEGIN 和 END 之间, 所以这些官方生成的注释不要取消掉

独立看门狗 IWDG

独立看门狗是直接进行芯片复位。和窗口看门狗(WWDG)不一样。



STM32L051 的 IWDG 直接使用的内部 LSI 37khz 时钟

LSI 时钟频率: 使用内部低速时钟, 即 37Khz。但是 37Khz 经过 IWDG, IWDG 使用的计时时钟是不稳定的。所以直接人为取 40Khz 进行计算。

Categories A-Z

System Core

- DMA
- GPIO
- IWDG**
- NVIC
- RCC
- SYS
- WWDG

Analog

Timers

Connectivity

Multimedia

Computing

Middleware

PR[2:0]: Prescaler divider
These bits are write access I written by software to select **Status register (IWDG_SR)**

- 000: divider /4
- 001: divider /8
- 010: divider /16
- 011: divider /32
- 100: divider /64
- 101: divider /128
- 110: divider /256
- 111: divider /256

Activated 激活独立看门狗

PR: 即 Prescaler register (IWDG_PR), 预分频因子。4 分频, 那么此处 PR 也就是 0

Configuration

Parameter Settings User Constants

Configure the below parameters :

Search (Ctrl+F)

Watchdog Clocking

- IWDG counter clock prescaler: 4
- IWDG window value: 4095
- IWDG down-counter reload value: 4095

RLR: 即 Reload register (IWDG_RLR), 重载寄存器的值, 也就是 4095

cubemx 初始化 IWDG 复位时间为:

复位时间 (单位秒) $T_{out} = \frac{PR(\text{分频因子}) * \text{reload}(\text{重装载值})}{\text{LSI 内部时钟频率}}$

$T_{out} = \frac{4 * 4095}{40000(40K)}$ 人为设定

$T_{out} = 0.4096S$ 也就是系统运行 0.4096 秒, 如果不喂狗, 就会复位

生成代码如下

```
/* Configure the system clock */
SystemClock_Config();
/* USER CODE BEGIN SysInit */
/* USER CODE END SysInit */
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();
MX_IWDG_Init();
/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    printf("time = %d\r\n", time++);
    HAL_Delay(1000);
}

/* USER CODE END 3 */
```

```
xxxxxxxx
time = 0
```

因为是 0.4ms 看门狗就要喂狗，但是我主程序延时的 1 秒，所以系统不停的复位。

```
IWDG_HandleTypeDef hiwdg;

/* IWDG init function */
void MX_IWDG_Init(void)
{

    hiwdg.Instance = IWDG;
    hiwdg.Init.Prescaler = IWDG_PRESCALER_4;
    hiwdg.Init.Window = 4095;
    hiwdg.Init.Reload = 4095;
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK)
    {
        Error_Handler();
    }
}
```

这是以前的分频系数 0.5 秒复位

```
IWDG_HandleTypeDef hiwdg;

/* IWDG init function */
void MX_IWDG_Init(void)
{

    hiwdg.Instance = IWDG;
    hiwdg.Init.Prescaler = IWDG_PRESCALER_256; //我对LSI进行256分频
    hiwdg.Init.Window = 4095;
    hiwdg.Init.Reload = 2500; //重装时间改成2500 最后算下来Tout = 256*2500/40000 = 16S (16秒复位1次)
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK)
    {
        Error_Handler();
    }
}
```

```
xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
time = 5
time = 6
time = 7
time = 8
time = 9
time = 10
time = 11
time = 12
time = 13
time = 14
time = 15
time = 16
time = 17
xxxxxxxx
```

这是修改后的分频系数和重载系数，16 秒复位一次。测试之后证明，确实是 16 秒复位一次。

下面加入喂狗模式

```
HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg) //独立看门狗喂狗
```

```

int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    printf("time = %d\r\n", time++);
    HAL_Delay(1000);
    HAL_IWDG_Refresh(&hiwdg); //喂狗
}

```

```

xxxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
time = 5
time = 6
time = 7
time = 8
time = 9
time = 10
time = 11
time = 12
time = 13
time = 14
time = 15
time = 16
time = 17
time = 18
time = 19
time = 20

```

喂狗之后，系统 16 秒之后没有复位，正常运行。

IWDG 唤醒休眠模式的 MCU

注意: IWDG 是可以用芯片复位方式唤醒 MCU 低功耗睡眠的

```

MX_USART1_UART_Init();
MX_IWDG_Init();
/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

    if(time > 5)
    {
        HAL_RCC_PWR_CLK_ENABLE();
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗运行模式
        HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); //进入睡眠模式
        HAL_ResumeTick();
    }
}

```

```

xxxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
time = 5
xxxxxxxxx
time = 0
time = 1
time = 2

```

系统 5 秒之后睡眠，过了 16 秒之后又复位重启了。

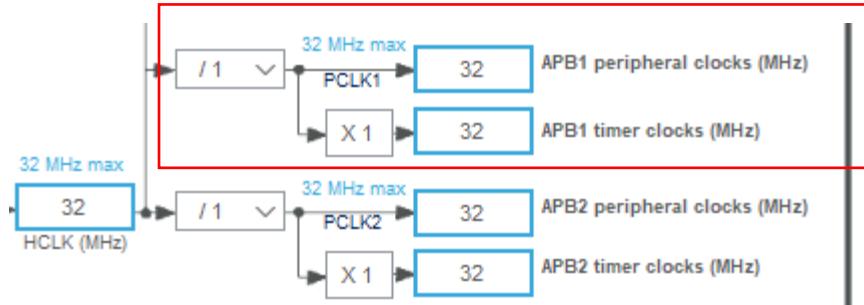
所以 IWDG 开启之后是无法关闭的，这就导致我低功耗模式无法使用 IWDG。只有常规模式 IWDG 才有意义。也可以采用将看门狗复位时间 > 休眠时间，这样第二次唤醒时就可以马上喂狗。

下面看看窗口看门狗 WWDG 能否解决低功耗问题

窗口看门狗(WWDG)

窗口看门狗 (WWDG) 也是程序中用的比较多的，通常被用来监测由外部干扰或不可预见的逻辑条件造成的应用程序背离正常的运行序列而产生的软件故障

独立开门狗是直接进行芯片复位，



窗口看门狗采用的是 APB1 输出的时钟。

Parameter	Value
WWDG counter clock prescaler	8
WWDG window value	64
WWDG free-running downcount...	127

记住：计数值必须大于窗口值，不然死机。

cubemx 初始化 WWDG 复位时间为：

$$\text{刷新时间 } t \text{ (毫秒单位)} = \frac{1}{(\text{APB1时钟} / 4096 / \text{分频系数}) * (\text{计数器值} - \text{窗口值})}$$

刷新的意思是，不得早于刷新时间喂狗，不然会复位，也不得低于刷新时间喂狗，不然也会复位

根据我现在cubemx配置

$$t(\text{ms}) = 1 / (32000 / 4096 / 8) * (127 - 64) = 64.512\text{ms} \text{ 最小刷新时间}$$

$$t(\text{ms}) = 1 / (32000 / 4096 / 8) * (127 - 63) = 65.536\text{ms} \text{ 最晚刷新时间}$$

NVIC Interrupt Table	Enabled	Preemption Priority
Window watchdog interrupt	<input checked="" type="checkbox"/>	0

开启窗口看门狗中断

Window watchdog interrupt 3 WWDG 优先级最低

生成代码如下

```
SystemClock_Config();

/* USER CODE BEGIN SysInit */
/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
// MX_RTC_Init();
MX_USART1_UART_Init();
MX_WWDG_Init();
/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

}
```

```
xxxxxxxx
time = 0
```

我设置的 WWDG 是 64ms，时间太短，系统不停的复位。这是因为你没在规定的回调函数中喂狗。

void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwdg) //WWDG 喂狗回调钩子
函数，必须实现，不然系统会不停的复位

HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwdg) //窗口看门狗喂狗

```
MX_GPIO_Init();
// MX_RTC_Init();
MX_USART1_UART_Init();
MX_WWDG_Init();

/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);

}
```

```
xxxxxxxx
time = 0
time = 1
time = 2
time = 3
time = 4
time = 5
time = 6
time = 7
time = 8
time = 9
time = 10
time = 11
time = 12
time = 13
time = 14
```

系统不在不停复位了，运行正常。

WWDG 在规定喂狗的时间内会调用回调函数，这个回调函数就解决了，早喂狗或者晚喂狗导致系统复位的问题，这个回调函数就是规定时间内喂狗。

```
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwdg)
{
    HAL_WWDG_Refresh(hwdg);
    //printf("\r\n 喂狗成功 !!!\r\n");
}
```

```
/* WWDG init function */
void MX_WWDG_Init(void)
{
    hwdg.Instance = WWDG;
    hwdg.Init.Prescaler = WWDG_PRESCALER_8;
    hwdg.Init.Window = 64;
    hwdg.Init.Counter = 127;
    hwdg.Init.EWIMode = WWDG_EWI_ENABLE;
    if (HAL_WWDG_Init(&hwdg) != HAL_OK)
    {
        Error_Handler();
    }
}
```

这是以前的分频系数，现在我把时间改长点，看看是不是按照我的时间要求调用的喂狗回调函数。

```
void MX_WWDG_Init(void)
{
    hwwdg.Instance = WWDG;
    hwwdg.Init.Prescaler = WWDG_PRESCALER_8; //分频系数改成8
    hwwdg.Init.Window = 64;
    hwwdg.Init.Counter = 1000; //尝试修改了计数值，意义不大
    hwwdg.Init.EWIMode = WWDG_EWI_ENABLE;
    if (HAL_WWDG_Init(&hwwdg) != HAL_OK)
    {
        Error_Handler();
    }
}
```

我尝试修改了计数值，意义不大

```
MX_WWDG_Init();

/* USER CODE BEGIN 2 */
printf("xxxxxxxx\r\n");
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
int time = 0;
while (1)
{
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */

    printf("time = %d\r\n", time++);
    HAL_Delay(1000);
    if(time > 5)
    {

        HAL_RCC_PWR_CLK_ENABLE();
        HAL_PWREx_EnableUltraLowPower(); //开启低功耗运行模式
        HAL_PWREx_EnableFastWakeUp(); //开启超低功耗快速唤醒

        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI); //进入睡眠模式
        HAL_ResumeTick();
    }
}

```

```
time = 51
time = 52
time = 53
time = 54
time = 55
time = 56
time = 57
time = 58
time = 59
time = 60
time = 61
time = 62
time = 63
time = 64
time = 65
```

看来窗口看门狗也无法让系统进入低功耗模式。