**STM32G071RB低功耗实验**

**第一步**：了解Cortex\_M0的多种工作模式

STM32G0器件具有FlexPowerControl，它提高了功耗管理模式的灵活性，进一步降低了应用的总体功耗。“运行”模式最高支持64MHz的系统时钟，功耗仅100μA/MHz。系统时钟为16Mhz时功耗更低，仅93μA/MHz。

STM32G0支持7种主要的的低功耗模式，分别是：低功耗运行、睡眠、低功耗睡眠、停止0、停止1、待机和关机模式。每个模式又支持多种配置，提供7种额外的子模式。此外STM32G0还支持电池备份区域“VBAT”。

两种停止模式“停止0”和“停止1”是保留全部RAM数据内容的最低功耗模式，仅需2μS即可实现从STOP唤醒到16Mhz的运行模式。在停止模式下，SRAM和外设寄存器的内容得以保留，所有高端外设时钟停止，可以使能32KHz外部振荡器和32KHz内部振荡器，多种外设事件可以用来唤醒STOP停止模式下的MCU。

在“停止0”模式下，调压器配置为主调压器模式，所以在该模式下，VCORE区域中的所有时钟停止，PLL、HSL16和HSE晶振被禁用。通过内部或外部低速振荡器提供时钟的RTC可以保持激活状态，BOR（欠压复位）始终被使能。在“停止0”模式下，大多数外设时钟均关闭，只有几种外设仍在工作。

“停止1”模式是本次实验中我们要用到的模式，它与“停止0”模式十分相似。区别在于，功耗显著降低，原因是主调压器停止，并用低功耗调压器代替。无RTC的“停止1”模式的电流消耗为1.3μA。通常在电压为3V，Flash存储器不供电且RTC禁用的情况下，唤醒时间为5μS，唤醒调压器在“模式1或2”时的系统时钟为HIC16MHz。Flash和HSI16是可配置的，可以停止或保持使能状态。

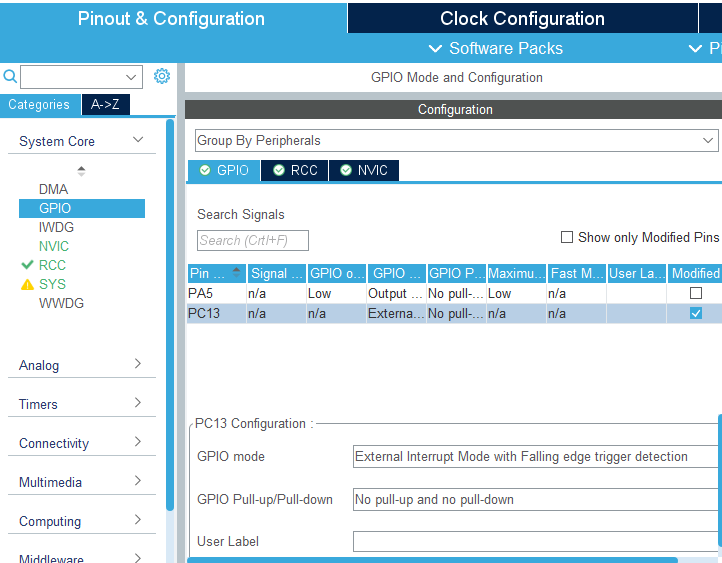
由于我们要使用RTC唤醒微控制器，以下简要介绍RTC（既实时时钟）。RTC外设作为超低功耗日历，其闹铃功能在所有低功耗模式下均保持运行状态。此外，在使用32KHz低功耗外部振荡器（LSE）时钟时，即使主电源断开，而VBAT区域通过备用电源供电，RTC仍然保持工作。RTC在电压为1.8V时的电流消耗仅300nA，包括LSE功耗。硬件日历以BCD格式显示，这可以降低软件负担，特别是在必须显示日期和时间时。备份寄存器和篡改检测属于TAMP外设。

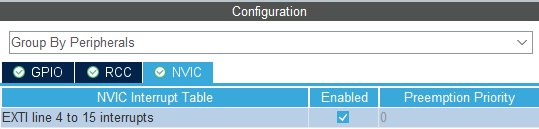
**第二步**：确定实验方向，及其主要操作

在本实验中，我们要将微控制器设置为“停止一”低功耗模式，并通过RTC（实时时钟）唤醒。RTC被设置为每5S执行一次STM32唤醒。当STM32被唤醒时，我们要使LED点亮1S，然后回到“停止”模式。我们同样可以使用“USER”按键，连接到“USER”按键的IO（PC13）被设置为“外部中断”。

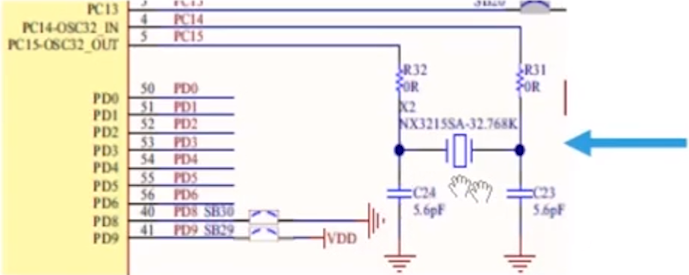
**第三步**：创建工程

①将PA5管脚设置为通用输出工作模式GPIO\_Output，将PC13管脚设置为中断工作模式GPIO\_EXTI13，如下图所示，将中断设置为下降沿触发，并使能该中断。

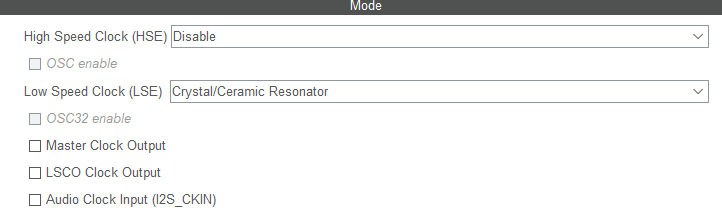
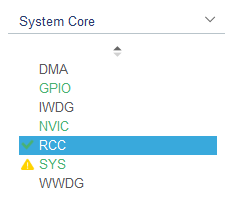




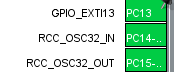
②然后，使能LSE即低速外部时钟。具体操作如下：



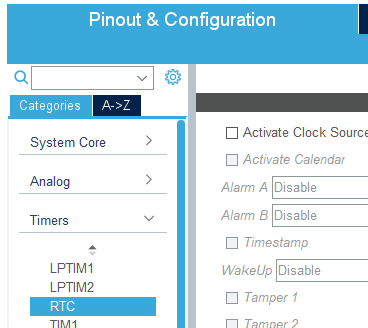
由上边原理图可知，有一个连接了PC14和PC15的32KHz晶振，我们要使用它为RTC提供时钟。选择Pinout&Configuration选项卡，展开System Core，点击RCC，将LSE设置为Crystal/Ceramic Resonator。如图：



设置完成后将选择PC14和PC15管脚（已经为其配置了OSC32输入和输出）。如图所示：



③下一步使能和配置RTC，在“引脚布局与配置”选项卡的“定时器下”展开RTC



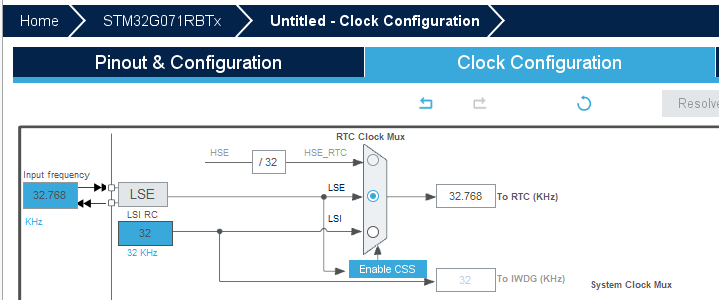
选中并激活时钟源。

93b2cff245da170e572d8764cc05e48

并选择内部唤醒作为唤醒模式：

c822a8d72ef132d8128b048704a27d7

接下来选择RTC时钟源，在“时钟配置”选项卡上，选择“LSE”作为RTC的“输入”。



现在，配置RTC。唤醒计数器的值计算如下，我们需要的唤醒时间为5S，因此要将唤醒计数器设定为“10246”。其计算过程如下：

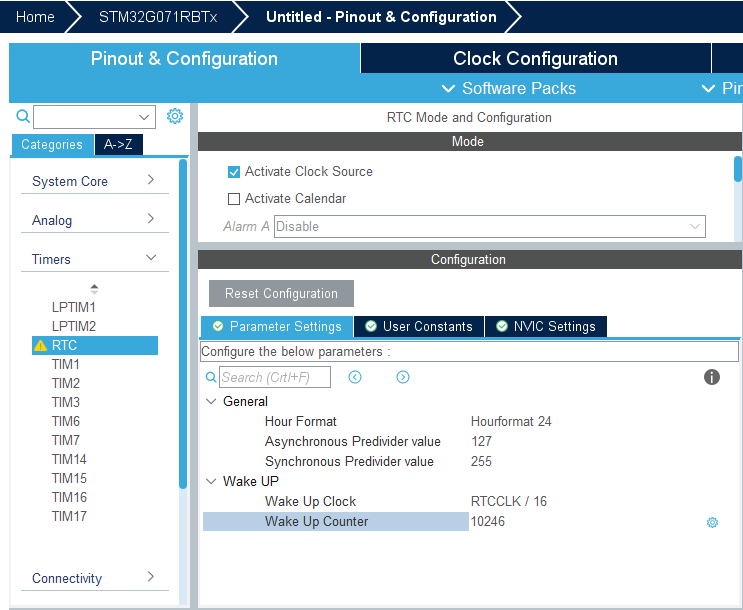
在将RTC设置为16分频的情况下。

Wakeup Time Base = RTC\_PRESCALER / LSE=16 / (32.768KHz) = 0.488ms

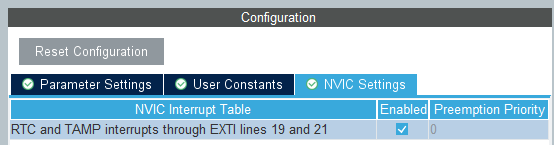
Wakeup Time = Wakeup Time Base \* Wakeup Counter = 0.488ms \* Wakeup Counter

所以Wakeup Counter = 5s / 0.488ms = 10246

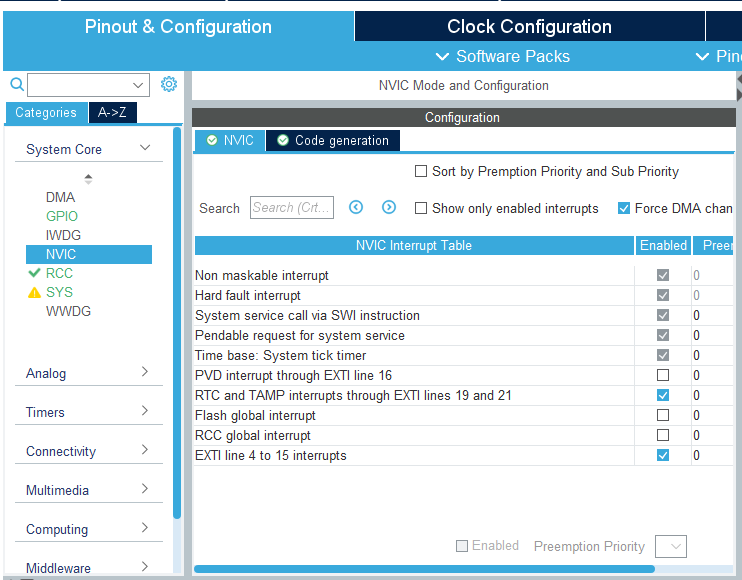
在“引脚布局与配置”选项卡上，点击“RTC”，在“配置”部分的RTC参数设置中，输入以下信息：“唤醒时钟”输入“RTCCLK/16”，“唤醒计数器”输入“10246”



首先需要启用中断：首先在“配置”部分的“NVIC设置”选项卡上，使能RTC中断



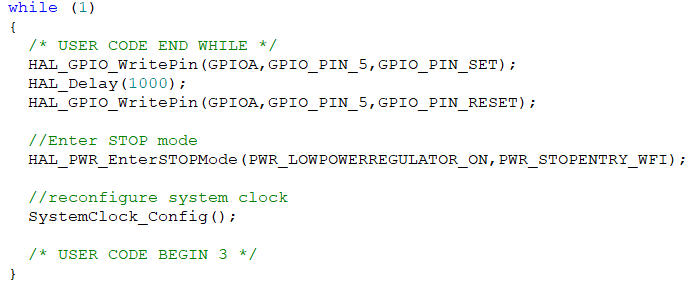
然后，在“系统视图”中点击“NVIC”确保RTC和EXTI[4...15]使能，如图所示：



所有配置完成之后，生成代码，打开工程。

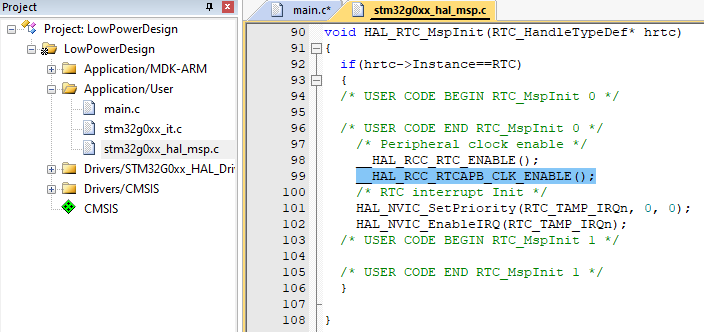
**第四步**：添加设计代码

在主函数while循环下添加



如上图所示，在运行模式LED灯点亮1秒钟，然后进入停止模式，停止模式被唤醒后，重新配时钟。

在之前的STM32CubeMX版本中，我们需要使能一个时钟才能让RTC代码正常运行，我们必须确认在文件“stm32g0xx\_hal\_msp.c”中的“HAL\_RTC\_MspInit”函数中使能了RTCAP时钟，如下图所示：



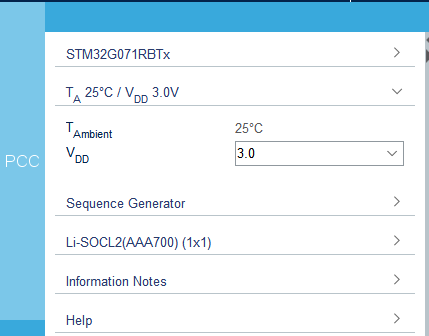
【如果没有，请添加这行代码】

**第五步**：编译并直接加载代码，不进入调试界面

为什么不进入调试界面？因为，当微控制器进入低功耗模式时，我们与调试工具将失去通信，因此我们只加载代码。在加载代码后，按下板子上的“复位按键”。并确保代码的运行符合指定要求，即以“运行”模式运行1秒钟，以“停止”模式运行5秒钟，并通过RTC唤醒。在微控制器处于“停止”模式时，我们可以按下“用户”按键，将微控制器从“停止”模式唤醒并同时点亮LED灯。

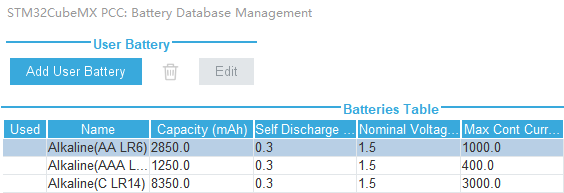
**第六步**：使用STM32CubeMX中的功耗工具，估算以上低功耗实验的平均功耗

在“工具”选项卡选择“3V”作为“VDD”。

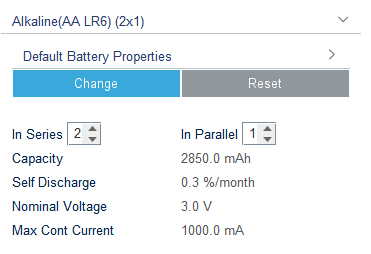


并选择所需要的电池。





让两节1.5V电池串联获取3V标准电压。



接下来开始添加步骤：

①添加“运行”模式。

“运行”模式特性：“电压范围”为“范围2-中等”，然后从Flash运行，VDD为3V，“电源”为电池。将以HSI提供的16MHz时钟频率运行。并使能引脚布局中涉及到的所有IP，我们将导入之前的项目中使用的所有不同IP或外设。然后持续时间选择为1s，实际上就是我们之前实现的“运行”模式闪烁或LED点亮时间为1s。

这一步添加完毕之后，电流消耗应为1.6mA，就是具有这些特性的运行模式下的功耗。

由下图1，“Results”栏可看出，“Step Consumption”为1.6mA符合预期。

②添加“停止1”模式

“停止1”模式特性：内存取址类型为“Flash掉电”，VDD为3V，“电源”为电池。将唤醒MCU并使用16MHz的HSI。只有RTC使能。该步骤的持续时间为5s。

这一步添加完毕之后，电流消耗应为3.4μA，就是具有这些特性的运行模式下的功耗。

由下图2，“Results”栏可看出，“Step Consumption”为3.4μA符合预期。

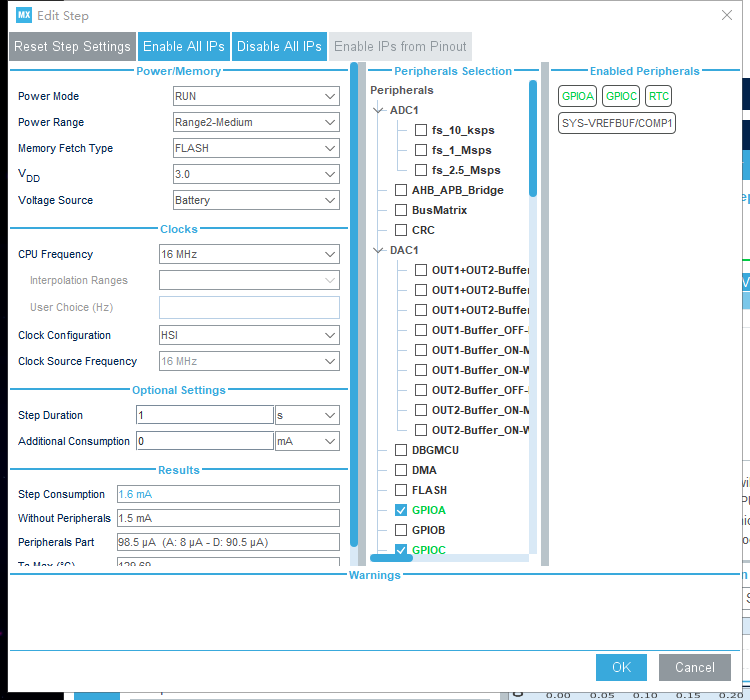


图1

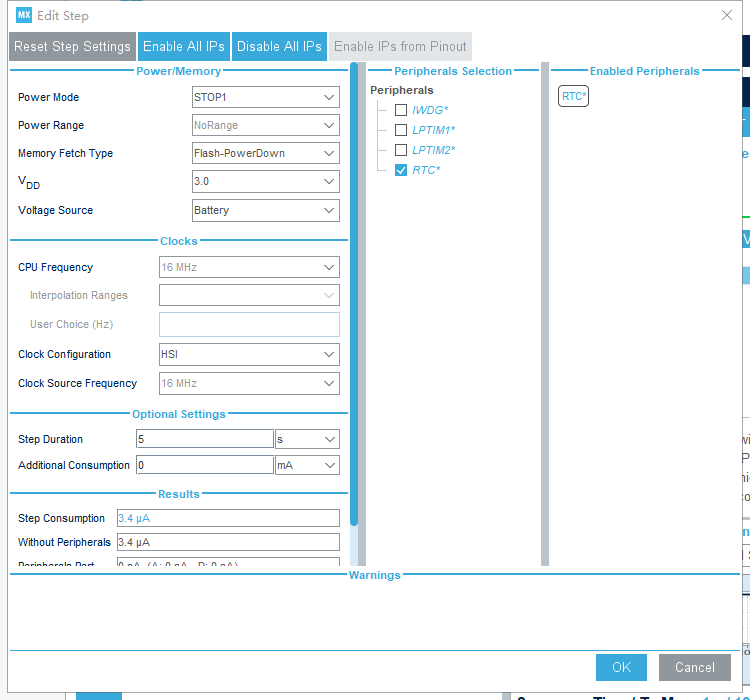


图2

③添加“从停止1唤醒”模式

“从停止1唤醒”模式特性：VDD为3V，“电源”为电池。9μs是唤醒最差的情况。

这一步添加完毕之后，电流消耗应为1.21mA，就是具有这些特性的运行模式下的功耗。

由下图3，“Results”栏可看出，“Step Consumption”为1.21mA符合预期。

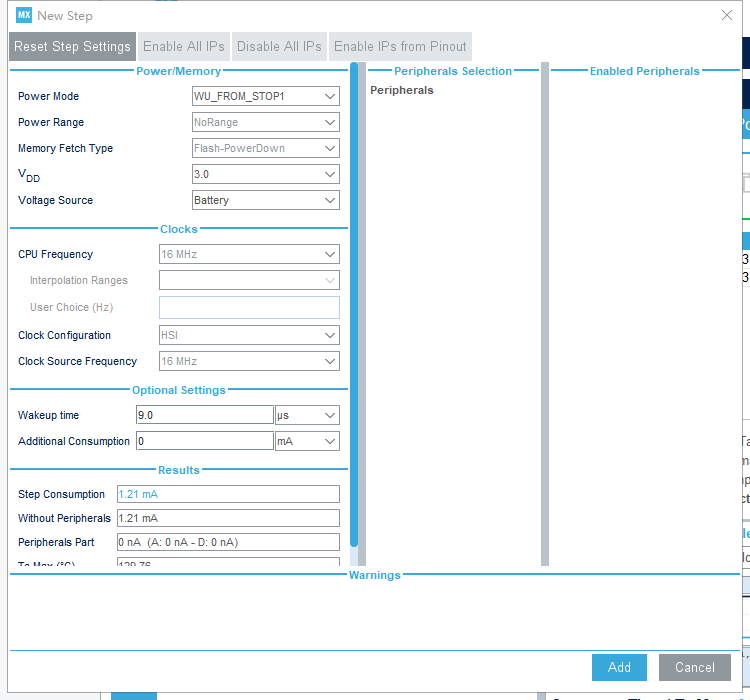
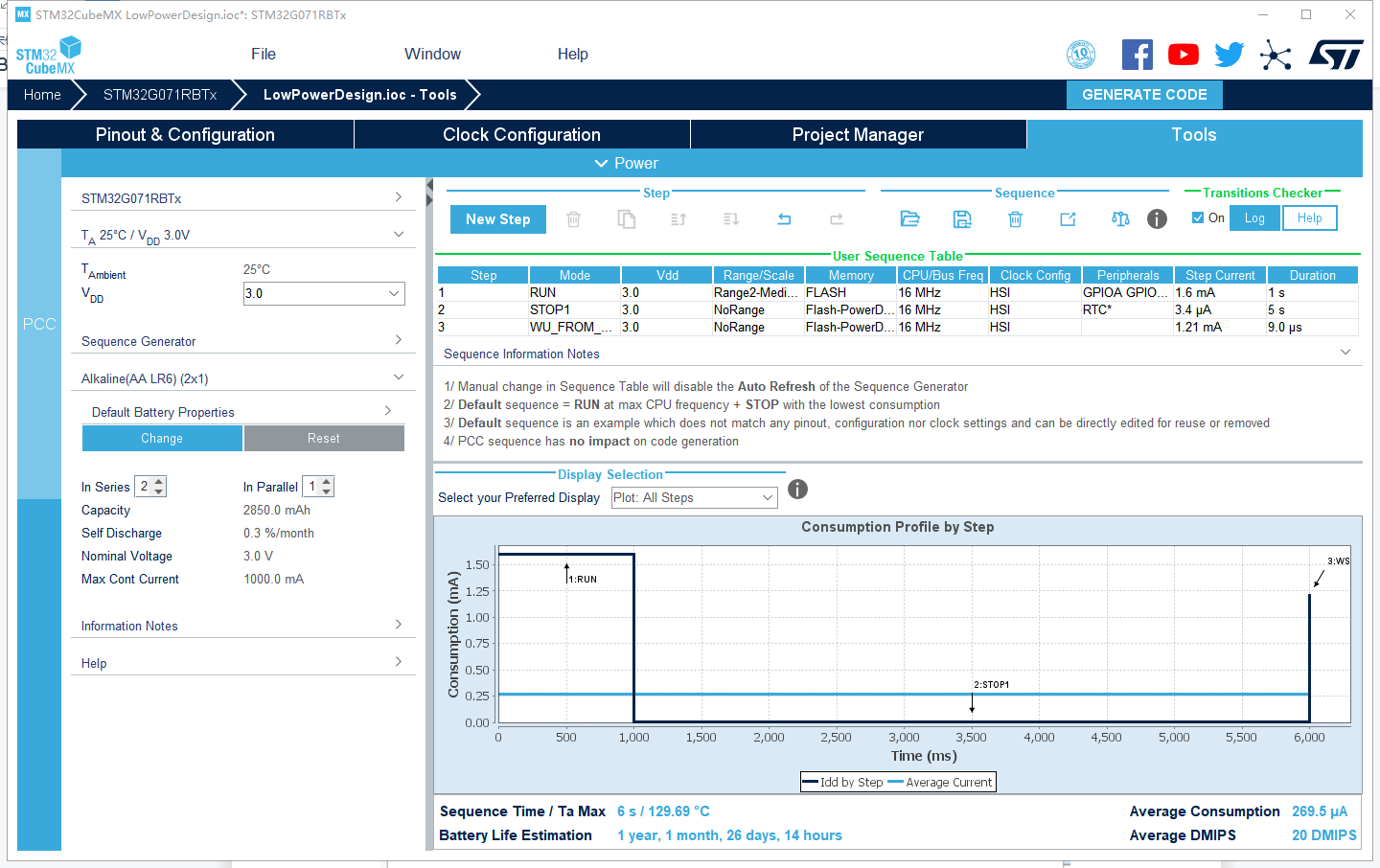


图3

“工具”选项卡的底部为我们提供了该应用的平均电流消耗，如下图所示。



从上图的电流消耗曲线，可以看出该应用的平均电流消耗为269.5μA，由于我们选择了AA电池，可以估算出电池寿命为1年1个月26天14小时。

**第七步**：拓展类研究之低功耗模式与LCD1602混合使用

①像之前一样配置低功耗模式的端口

②添加LCD1602的相关配置

③改写代码：低功耗 + LCD1602

在main.c文件开头添加LCD1602的子函数声明，如下：

void delay(void);

void lcdwritecmd(unsigned char cmd);

void lcdwritedata(unsigned char dat);

void lcdinit(void);

void lcdsetcursor(unsigned char x, unsigned char y);

void lcdshowstr(unsigned char x, unsigned char y, unsigned char \*str);

在主函数的while(1)循环之前，初始化LCD1602

lcdinit();

delay();

在循环中添加如下代码：

lcdshowstr(0,0,"Welcome To"); //“运行模式”

lcdshowstr(0,1,"Run Mode "); //输出多个空格是为了清除之前的输出

HAL\_Delay(2000); //延时两秒

lcdshowstr(0,0,"Welcome To"); //进入“低功耗”模式前一刻，改变输出

lcdshowstr(0,1,"Low Power Mode");

//Enter STOP mode 低功耗模式下不改变1602的输出

HAL\_PWR\_EnterSTOPMode(PWR\_LOWPOWERREGULATOR\_ON,PWR\_STOPENTRY\_WFI);

//reconfigure system clock

SystemClock\_Config();

最后添加LCD1602的子函数：

void delay ()

{

for(int i=0;i<99;i++)

for(int j=0;j<99;j++)

{}

}

void lcdwritecmd(unsigned char cmd)

{

// lcdwait();

delay();

GPIOB->ODR=0x00; //E=0\RW=0\RS=0

GPIOA->ODR=cmd;

GPIOB->ODR=0x04; //E=1\RW=0\RS=0

delay();

GPIOB->ODR=0x00; //E=0\RW=0\RS=0

}

void lcdwritedata(unsigned char dat)

{

// lcdwait();

delay();

GPIOB->ODR=0x01; //E=0\RW=0\RS=1

GPIOA->ODR=dat;

GPIOB->ODR=0x05; //E=1\RW=0\RS=1

delay();

GPIOB->ODR=0x01; //E=0\RW=0\RS=1

}

void lcdinit()

{

lcdwritecmd(0x38); //2hang moshi 5\*8dian zhen 8wei kuandu

lcdwritecmd(0x0c); //dakai xianshi guanbi guangbiao

lcdwritecmd(0x06); //wenzi budong dizhi zidong jiayi

lcdwritecmd(0x01); //qingping

}

void lcdsetcursor(unsigned char x, unsigned char y)

{

unsigned char address;

if(y==0) //diyihang cunchuqi dizhi 00kaishi

address=0x00+x;

else //dierhang cunchuqi dizhi 40kaishi

address=0x40+x;

lcdwritecmd(address|0x80); //xie cunchuqi dizhi

}

void lcdshowstr(unsigned char x, unsigned char y, unsigned char \*str)

{ //zai x,y chu xianshi zifu

lcdsetcursor(x,y);

while((\*str)!='\0') //bushi zifuchuan jiewei jixu

{

lcdwritedata(\*str); //xieshujv

str++; //zhizhen jiayi zhixiang xiayige zifu

}

}

**第八步**：结果展示与分析

运行模式如图1，低功耗模式如图2。

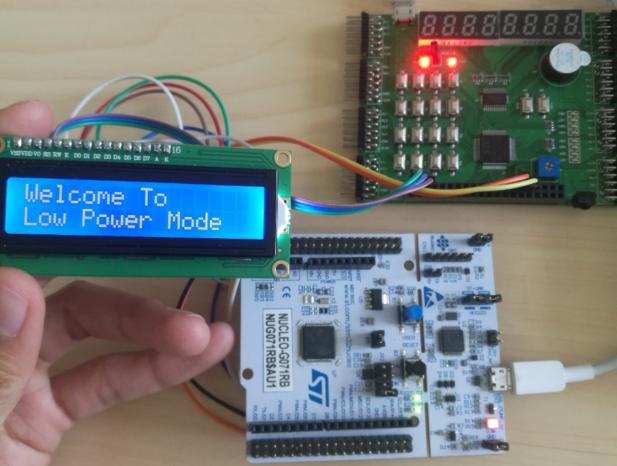


图 1 图 2

分析：按下复位按键后，开始执行主函数，先初始化HAL、Clock、GPIO、RTC以及LCD，之后进入循环while(1)。在运行模式下LCD1602输出“Run Mode”，持续两秒后，在进入低功耗模式的前一刻，改变LCD1602的输出为“Low Power Mode”。进入低功耗模式后，除RTC外的所有外设均无使能，所以不会给LCD1602发送新的信号，LCD1602保持原状“Low Power Mode”。在5s之后，由RTC唤醒MCU，重新进入运行模式。当然，如之前一样，也可以通过外部中断按键“USER”唤醒MCU。

猜想：如果我们改变LCD1602的输出为“Low Power Mode”是在进入低功耗模式之后，会出现什么情况呢？

将主函数while(1)循环下改为

lcdshowstr(0,0,"Welcome To");

lcdshowstr(0,1,"Run Mode ");

HAL\_Delay(2000);

//Enter STOP mode

HAL\_PWR\_EnterSTOPMode(PWR\_LOWPOWERREGULATOR\_ON,PWR\_STOPENTRY\_WFI);

lcdshowstr(0,0,"Welcome To");

lcdshowstr(0,1,"Low Power Mode");

//reconfigure system clock

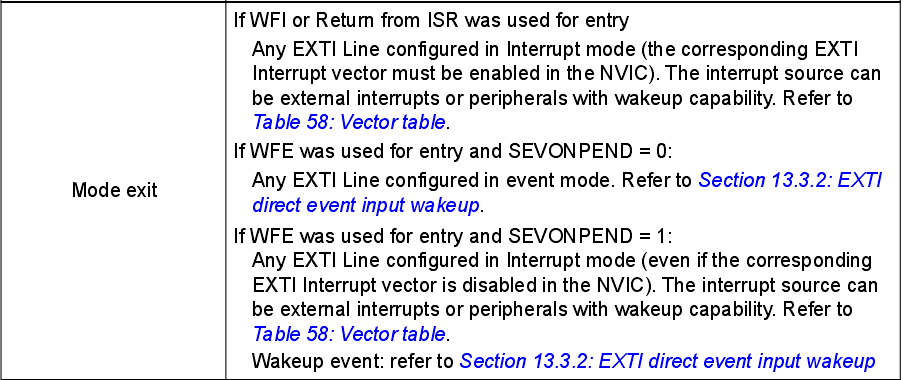
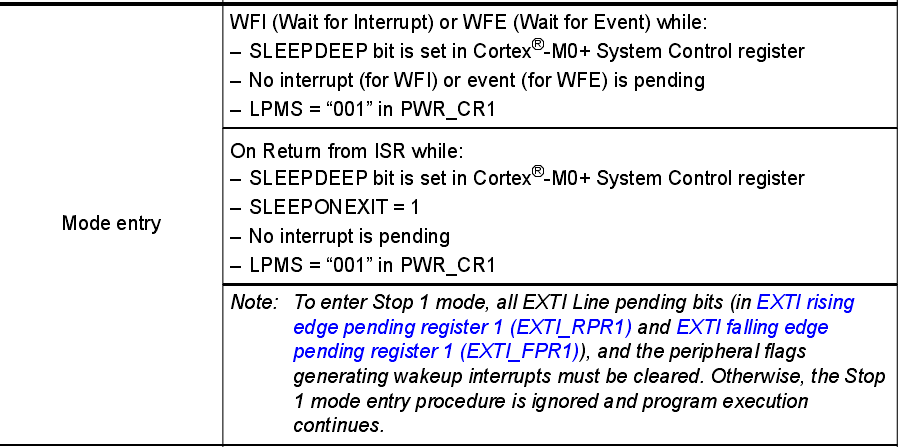
SystemClock\_Config();

之后观察结果，LCD1602将一直显示“Run Mode”，所以在低功耗的“停止1”模式下，MCD不会通过GPIO端口向LCD1602发送命令，GPIO端口使能被禁止。

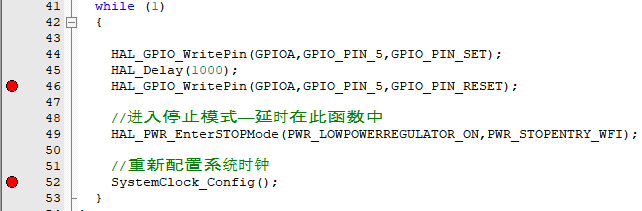
**第九步：**底层原理分析

目标：寻找外部中断触发系统唤醒的过程机理

①查看硬件手册关于STOP1模式的描述



②利用debug工具寻找中断到来前的等待



如上图所示，断点一到断点二之间的运行事件较长。断点二处按下F5开始运行后，LD4灯立即点亮，即说明HAL\_GPIO\_WritePin已经被执行。分析：断点一和断点二之间，等待中断到来。当实时时钟RTC产生中断时，立即退出低功耗模式。

③分析低功耗模式执行的内容：

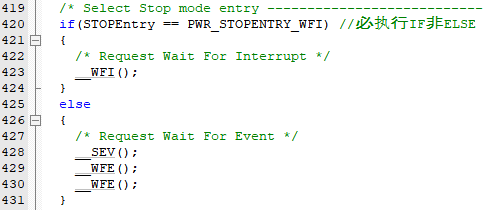
停止模式HAL\_PWR\_EnterSTOPMode函数（此刻产生延时等待中断）

实参1:PWR\_LOWPOWERREGULATOR\_ON也即PWR\_CR1\_LPR

实参2:PWR\_STOPENTRY\_WFI实为0x01U

函数内容：先检验输入的两个参数是否符合规范。再根据实参1判断进入停止0还是停止1模式，进入函数MODIFY\_REG，参数1为PWR->CR1指定寄存器，参数二PWR\_CR1\_LPMS辅助运算，参数三PWR\_LOWPOWERMODE\_STOP0/PWR\_LOWPOWERMODE\_STOP1指定如何修改寄存器的值即方式，通过修改此寄存器的值进入低功耗模式。设置Cortex系统控制寄存器的SLEEPDEEP位，函数SET\_BIT参数一为寄存器SCB->SCR，参数二为SCB\_SCR\_SLEEPDEEP

\_Msk实现按位或之后赋值给寄存器（置一）。之后用IF分hi结构来选择其等待中断唤醒还是等待事件唤醒，此实验直接配置为等待中断唤醒（两参数值始终相同）。当中断来临之后



退出等待函数。用CLEAR\_BIT参数一为寄存器SCB->SCR，参数二为SCB\_SCR\_SLEEPDEEP

\_Msk实现参数二取反后按位与寄存器的值之后再赋值给寄存器（清零）。结束。

④跳出来之后系统被唤醒，重新配置系统时钟SystemClock\_Config。进了STOP模式后，PLL停掉了，所以，如果开始的时钟配置，用的是PLL，那么唤醒后，需要重新配置RCC。

⑤总结：系统从停止模式唤醒时等待中断（中断均可唤醒，无需回调函数）。实时时钟RTC定时产生中断唤醒系统，本质也是中断唤醒系统。系统时钟：进入停止模式后被修改，退出后需要重新配置。实时时钟：不会受任何影响（停止模式，外部中断都不会影响、打断其正常运行），即编译结果为在两次自动唤醒之间按下中断，实时时钟不会复位而是继续运行，第二次自动唤醒正常到达。