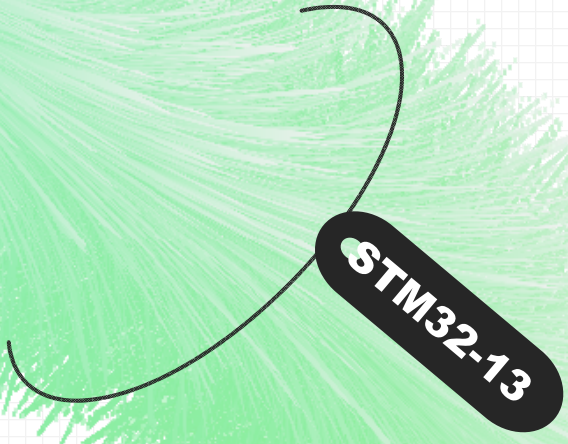
13. STM32的低功耗管理

北京科技大学
计算机与通信工程学院

目录

CONTENTS.



STM32的电源管理简介

电源对电子设备的重要性不言而喻，它是保证系统稳定运行的基础，而保证系统能稳定运行后，又有低功耗的要求。

在很多应用场合中都对电子设备的功耗要求非常苛刻，如某些传感器信息采集设备，仅靠小型的电池提供电源，要求工作长达数年之久，且期间不需要任何维护；由于智慧穿戴设备的小型化要求，电池体积不能太大导致容量也比较小，所以也很有必要从控制功耗入手，提高设备的续行时间。

STM32有专门的电源管理外设监控电源并管理设备的运行模式，确保系统正常运行，并尽量降低器件的功耗。

电源管理—电源监控器

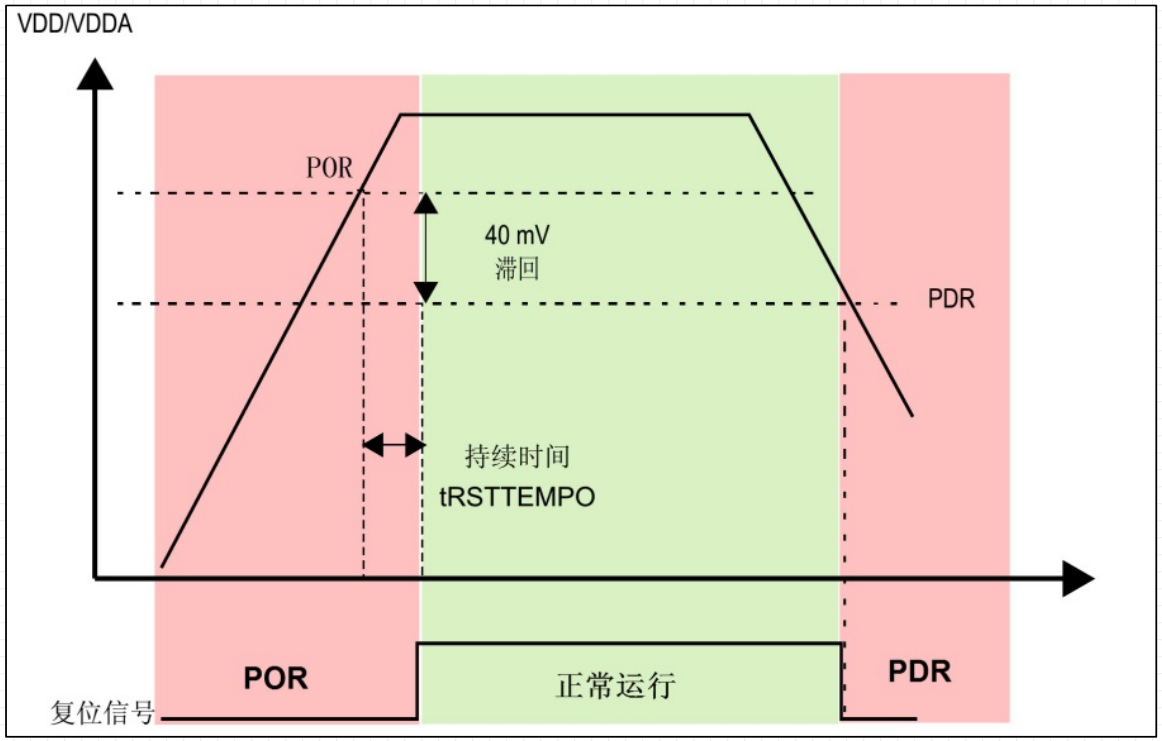
STM32芯片主要通过引脚VDD从外部获取电源，在它的内部具有电源监控器用于检测VDD的电压，以实现复位功能及掉电紧急处理功能，保证系统可靠地运行。

1. 上电复位与掉电复位(POR与PDR)

当检测到VDD的电压低于阈值VPOR及VPDR时，无需外部电路辅助，STM32芯片会自动保持在复位状态，防止因电压不足强行工作而带来严重的后果。在刚开始电压低于VPOR时(约1.92V)，STM32保持在上电复位状态(POR，Power On Reset)，当VDD电压持续上升至大于VPOR时，芯片开始正常运行，而在芯片正常运行的时候，当检测到VDD电压下降至低于VPDR阈值(约1.88V)，会进入掉电复位状态(PDR，Power Down Reset)。

上电复位与掉电复位(POR与PDR)

1.



可编程电压检测器PVD

上述POR、PDR功能是使用其电压阈值与外部供电电压VDD比较，当低于工作阈值时，会直接进入复位状态，这可防止电压不足导致的误操作。

除此之外，STM32还提供了可编程电压检测器PVD，它也是实时检测VDD的电压，当检测到电压低于编程的VPVD阈值时，会向内核产生一个PVD中断(EXTI16线中断)以使内核在复位前进行紧急处理。该电压阈值可通过电源控制寄存器PWR_CSR设置。

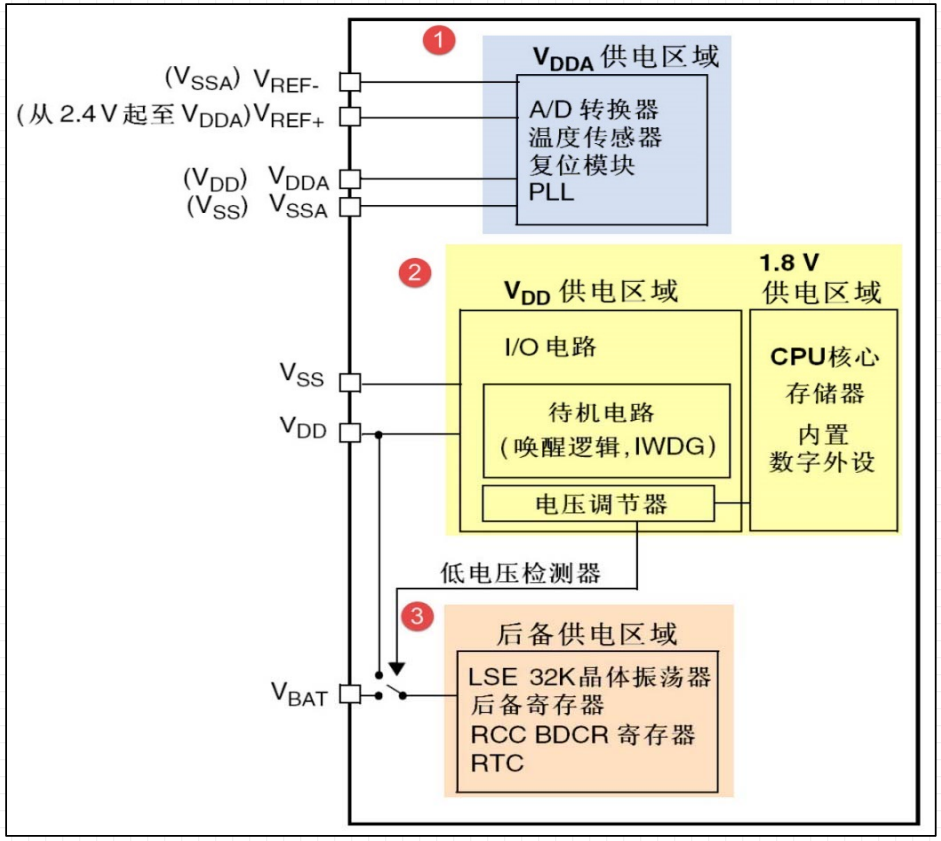
3. 可编程电压检测器PVD

使用PVD可配置8个等级，如下表。其中的上升沿和下降沿分别表示类似前面图中的VDD电压上升过程及下降过程的阈值。

阈值等级	条件	最小值	典型值	最大值	单位
级别0	上升沿	2.1	2.18	2.26	V
	下降沿	2	2.08	2.16	V
级别1	上升沿	2.19	2.28	2.37	V
	下降沿	2.09	2.18	2.27	V
级别2	上升沿	2.28	2.38	2.48	V
	下降沿	2.18	2.28	2.38	V
级别3	上升沿	2.38	2.48	2.58	V
	下降沿	2.28	2.38	2.48	V
级别4	上升沿	2.47	2.58	2.69	V
	下降沿	2.37	2.48	2.59	V
级别5	上升沿	2.57	2.68	2.79	V
	下降沿	2.47	2.58	2.69	V
级别6	上升沿	2.66	2.78	2.9	V
	下降沿	2.56	2.68	2.8	V
级别7	上升沿	2.76	2.88	3	V
	下降沿	2.66	2.78	2.9	V

STM32的电源系统区域划分

为了方便进行电源管理，STM32把它的外设、内核等模块跟据功能划分了供电区域，其内部电源区域划分如图。



STM32的电源系统

STM32的电源系统主要分为备份域电路、内核电路以及ADC电路三部分，介绍如下：

- ADC电源及参考电压（V_{DDA}供电区域）

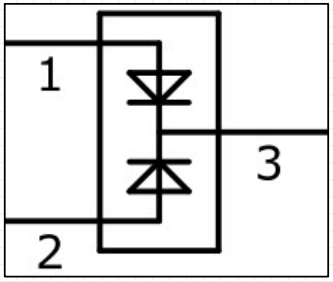
  为了提高转换精度，STM32的ADC配有独立的电源接口，方便进行单独的滤波。ADC的工作电源使用V_{DDA}引脚输入，使用V_{SSA}作为独立的地连接，V_{REF}引脚则为ADC提供测量使用的参考电压。

STM32的电源系统

- 备份域电路（后备供电区域）

STM32的LSE振荡器、RTC及备份寄存器这些器件被包含进备份域电路中，这部分的电路可以通过STM32的V_{BAT}引脚获取供电电源，在实际应用中一般会使用3V的钮扣电池对该引脚供电。

在图中备份域电路的左侧有一个电源开关结构，它的功能类似下图的双二极管，在它的“1”处连接了V_{BAT}电源，“2”处连接了V_{DD}主电源(一般为3.3V)，右侧“3”处引出到备份域电路中。当V_{DD}主电源存在时，由于V_{DD}电压较高，备份域电路通过V_{DD}供电，节省钮扣电池的电源，仅当V_{DD}掉电时，备份域电路由钮扣电池通过V_{BAT}供电，保证电路能持续运行，从而可利用它保留关键数据。



STM32的功耗模式

按功耗由高到低排列，STM32具有运行、睡眠、停止和待机四种工作模式。

上电复位后STM32处于运行状态时，当内核不需要继续运行，就可以选择进入后面的三种低功耗模式降低功耗，这三种模式中，电源消耗不同、唤醒时间不同、唤醒源不同，用户需要根据应用需求，选择最佳的低功耗模式。

这三种低功耗模式层层递进，运行的时钟或芯片功能越来越少，因而功耗越来越低。

表18 运行模式下的典型电流消耗，数据处理代码从内部Flash中运行

符号	参数	条件	f_{HCLK}	典型值(1)		单位
				使能所有外设(2)	关闭所有外设	
I_{DD}	运行模式下的供应电流	外部时钟(3)	72MHz	51	30.5	mA
			48MHz	34.6	20.7	
			36MHz	26.6	16.2	
			24MHz	18.5	11.4	
			16MHz	12.8	8.2	
			8MHz	7.2	5	
			4MHz	4.2	3.1	
			2MHz	2.7	2.1	
			1MHz	2	1.7	
			500kHz	1.6	1.4	
			125kHz	1.3	1.2	
		运行于高速内部RC振荡器(HSI)，使用AHB预分频以减低频率	64MHz	45	27	mA
			48MHz	34	20.1	
			36MHz	26	15.6	
			24MHz	17.9	10.8	
			16MHz	12.2	7.6	

电流消耗在30mA至50mA之间

符号	参数	条件	f_{HCLK}	典型值(1)		单位
				使能所有外设(2)	关闭所有外设	
I_{DD}	运行模式下的供应电流	外部时钟(3)	72MHz	29.5	6.4	mA
			48MHz	20	4.6	
			36MHz	15.1	3.6	
			24MHz	10.4	2.6	
			16MHz	7.2	2	
			8MHz	3.9	1.3	
			4MHz	2.6	1.2	
			2MHz	1.85	1.15	
			1MHz	1.5	1.1	
			500kHz	1.3	1.05	
			125kHz	1.2	1.05	
		运行于高速内部RC振荡器(HSI)，使用AHB预分频以减低频率	64MHz	25.6	5.1	mA
			48MHz	19.4	4	
			36MHz	14.5	3	
			24MHz	9.8	2	
			16MHz	6.6	1.4	
			8MHz	3.3	0.7	
			4MHz	2	0.6	
			2MHz	1.25	0.55	
			1MHz	0.9	0.5	
			500kHz	0.7	0.45	
			125kHz	0.6	0.45	

睡眠模式电流消耗
在6.4mA至29.5mA之间

表17 停机和待机模式下的典型和最大电流消耗

符号	参数	条件	典型值(1)		最大值		单位
			V_{DD}/V_{BAT} = 2.4V	V_{DD}/V_{BAT} = 3.3V	T_A = 85℃	T_A = 105℃	
I_{DD}	停机模式下的供应电流	调压器处于运行模式，低速和高速内部RC振荡器和高速振荡器处于关闭状态(没有独立看门狗)	34.5	35	379	1130	μA
		调压器处于低功耗模式，低速和高速内部RC振荡器和高速振荡器处于关闭状态(没有独立看门狗)	24.5	25	365	1110	
	待机模式下的供应电流	低速内部RC振荡器和独立看门狗处于开启状态	3	3.8	-	-	
		低速内部RC振荡器处于开启状态，独立看门狗处于关闭状态	2.8	3.6	-	-	
		低速内部RC振荡器和独立看门狗处于关闭状态，低速振荡器和RTC处于关闭状态	1.9	2.1	5(2)	6.5(2)	
I_{DD_VBAT}	备份区域的供应电流	低速振荡器和RTC处于开启状态	1.1	1.4	2(2)	2.3(2)	

停机模式：电流消耗大约在25uA左右

待机模式：电流消耗最低，通常在2uA左右

STM32的低功耗模式

模式	说明	进入方式	唤醒方式	对1.8V区域时钟的影响	对VDD区域 时钟的影响	调压器
睡眠	内核停止，所有外设包括M3核心的外设，如 NVIC 、 系 统 时 钟(SysTick)等仍在运行	调用WFI命令	任一中断	内核时钟关，对其他时钟和ADC时钟无影响	无	开
		调用WFE命令	唤醒事件			
停止	所有的时钟都已停止	配置PWR_CR寄存器的PDDS +LPDS位+SLEEPDEEP位+WFI或WFE命令	任一外部中断(在外部中断寄存器中设置)	关闭所有1.8V区域的时钟	HSI和HSE的振荡器关闭	开启或处于低功耗模式(依据电源控制寄存器的设定)
待机	1.8V 电源关闭	配置PWR_CR寄存器 的 PDDS +SLEEPDEEP位+WFI或WFE命令	WKUP 引脚的上升沿、RTC 闹钟事件、NRST 引脚上的外部复位、IWDG 复位			关

低功耗模式1： 睡眠模式

在睡眠模式中，仅关闭了内核时钟，内核停止运行，但其片上外设，CM3核心的外设全都还照常运行。

有两种方式进入睡眠模式，它的进入方式决定了从睡眠唤醒的方式，分别是WFI(wait for interrupt)和WFE(wait for event)，即由等待“中断”唤醒和由“事件”唤醒。睡眠模式的各种特性见下表：

特性	说明
立即睡眠	在执行 WFI 或 WFE 指令时立即进入睡眠模式。
退出时睡眠	在退出优先级最低的中断服务程序后才进入睡眠模式。
进入方式	内核寄存器的SLEEPDEEP = 0，然后调用WFI或WFE指令即可进入睡眠模式； 另外若内核寄存器的SLEEPONEXIT=0时，进入“立即睡眠”模式，SLEEPONEXIT=1时，进入“退出时睡眠”模式。
唤醒方式	如果是使用WFI指令睡眠的，则可使用任意中断唤醒； 如果是使用WFE指令睡眠的，则由事件唤醒。
睡眠时	关闭内核时钟，内核停止，而外设正常运行，在软件上表现为不再执行新的代码。这个状态会保留睡眠前的内核寄存器、内存的数据。
唤醒延迟	无延迟。
唤醒后	若由中断唤醒，先进入中断，退出中断服务程序后，接着执行WFI指令后的程序；若由事件唤醒，直接接着执行WFE后的程序。

与电源管理相关的内核寄存器SCR

4.4.5 System control register (SCB_SCR)

Address offset: 0x10

Reset value: 0x0000 0000

Required privilege: Privileged

The SCR controls features of entry to and exit from low power state.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											SEVON PEND	Res.	SLEEP DEEP	SLEEP ON EXIT	Res.
											rw		rw	rw	

低功耗模式2：停止模式

在停止模式中，进一步关闭了其它所有的时钟，于是所有的外设都停止了工作。

停止模式中，其1.8V区域的部分电源没有关闭，还保留了内核的寄存器、内存的信息，所以从停止模式唤醒，并重新开启时钟后，还可以从上次停止处继续执行代码。

停止模式可以由任意一个外部中断(EXTI)唤醒，在停止模式中可以选择电压调节器为开模式或低功耗模式。停止模式的各种特性见下表：

低功耗模式2： 停止模式

特性	说明
调压器低功耗模式	在停止模式下调压器可工作在正常模式或低功耗模式，可进一步降低功耗
进入方式	内核寄存器的SLEEPDEEP =1，PWR_CR寄存器中的PDDS=0，然后调用WFI或WFE指令即可进入停止模式； PWR_CR 寄存器的LPDS=0时，调压器工作在正常模式，LPDS=1时工作在低功耗模式；
唤醒方式	如果是使用WFI指令睡眠的，可使用任意EXTI线的中断唤醒； 如果是使用WFE指令睡眠的，可使用任意配置为事件模式的EXTI线事件唤醒。
停止时	内核停止，片上外设也停止。这个状态会保留停止前的内核寄存器、内存的数据。
唤醒延迟	基础延迟为HSI振荡器的启动时间，若调压器工作在低功耗模式，还需要加上调压器从低功耗切换至正常模式下的时间。
唤醒后	若由中断唤醒，先进入中断，退出中断服务程序后，接着执行WFI指令后的程序；若由事件唤醒，直接接着执行WFE后的程序。唤醒后，STM32会使用HIS作为系统时钟。

低功耗模式3.待机模式

待机模式，它除了关闭所有的时钟，还把1.8V区域的电源也完全关闭了，也就是说，从待机模式唤醒后，由于没有之前代码的运行记录，只能对芯片复位，重新检测boot条件，从头开始执行程序。

待机模式有四种唤醒方式，分别是WKUP(PA0)引脚的上升沿，RTC闹钟事件，NRST引脚的复位和IWDG(独立看门狗)复位。

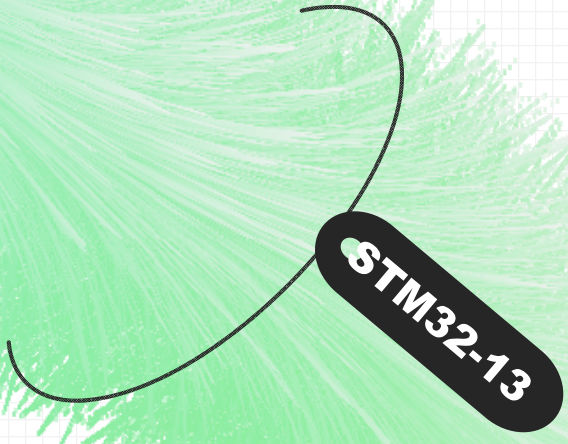
特性	说明
进入方式	内核寄存器的SLEEPDEEP =1，PWR_CR寄存器中的PDDS=1，PWR_CR寄存器中的唤醒状态位WUF=0，然后调用WFI或WFE指令即可进入待机模式；
唤醒方式	通过WKUP引脚的上升沿，RTC闹钟、唤醒、入侵、时间戳事件或NRST引脚外部复位及IWDG复位唤醒。
待机时	内核停止，片上外设也停止；内核寄存器、内存的数据会丢失；除复位引脚、RTC_AF1引脚及WKUP引脚，其它I/O口均工作在高阻态。
唤醒延迟	芯片复位的时间
唤醒后	相当于芯片复位，在程序表现为从头开始执行代码。

低功耗模式下的备份域电源

在以上睡眠模式、停止模式及待机模式中，若备份域电源正常供电，备份域内的RTC都可以正常运行，备份域内的寄存器的数据会被保存，不受功耗模式影响。

目录

CONTENTS.



电源管理相关的库函数及命令

STM32标准库对电源管理提供了完善的函数及命令，使用它们可以方便地进行控制。

配置PVD监控功能

PVD可监控VDD的电压，当它低于阈值时可产生PVD中断以让系统进行紧急处理，这个阈值可以直接使用库函数PWR_PVDLevelConfig配置成前面阈值表中说明的阈值等级。

WFI与WFE命令

在前面可了解到进入各种低功耗模式时都需要调用WFI或WFE命令，它们实质上都是内核指令，在库文件core_cm3.h中把这些指令封装成了函数：

代码清单 41-1 WFI 与 WFE 的指令定义(core_cm3.h 文件)

```
1
2  /** brief   等待中断
3
4      等待中断 是一个暂停执行指令
5      暂停至任意中断产生后被唤醒
6   */
7  #define __WFI                                   __wfi
8
9
10 /** brief   等待事件
11
12       等待事件 是一个暂停执行指令
13       暂停至任意事件产生后被唤醒
14  */
15 #define __WFE                                   __wfe
```

WFI与WFE命令

对于这两个指令，应用时只需要知道，调用它们都能进入低功耗模式，需要使用函数的格式“__WFI();”和“__WFE();”来调用(因为__wfi及__wfe是编译器内置的函数，函数内部使用调用了相应的汇编指令)。

其中WFI指令决定了它需要用中断唤醒，而WFE则决定了它可用事件来唤醒，关于它们更详细的区别可查阅《cortex-CM3权威指南》了解。

进入停止模式

直接调用WFI和WFE指令可以进入睡眠模式，而进入停止模式则还需要在调用指令前设置一些寄存器位，STM32标准库把这部分的操作封装到PWR_EnterSTOPMode函数中了，它的定义如下：

代码清单 41-2 进入停止模式

```
1
2  /**
3  * @brief 进入停止模式
4  *
5  * @note    在停止模式下所有 I/O 的会保持在停止前的状态
6  * @note    从停止模式唤醒后，会使用 HSI 作为时钟源
7  * @note    调压器若工作在低功耗模式，可减少功耗，但唤醒时会增加延迟
8  * @param   PWR_Regulator: 设置停止模式时调压器的工作模式
9  *             @arg PWR_MainRegulator_ON: 调压器正常运行
10 *             @arg PWR_Regulator_LowPower: 调压器低功耗运行
11 * @param   PWR_STOPEntry: 设置使用 WFI 还是 WFE 进入停止模式
12 *             @arg PWR_STOPEntry_WFI: WFI 进入停止模式
13 *             @arg PWR_STOPEntry_WFE: WFE 进入停止模式
14 * @retval None
15 */
16 void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry)
17 {
18     uint32_t tmpreg = 0;
19     /* 检查参数 */
20     assert_param(IS_PWR_REGULATOR(PWR_Regulator));
21     assert_param(IS_PWR_STOP_ENTRY(PWR_STOPEntry));
22
23     /* 设置调压器的模式 ------------*/
24     tmpreg = PWR->CR;
25     /* 清除 PDDS 及 LPDS 位 */
26     tmpreg &= CR_DS_MASK;
27     /* 根据 PWR_Regulator 的值(调压器工作模式)配置 LPDS,MRLVDS 及 LPLVDS 位*/
28     tmpreg |= PWR_Regulator;
29     /* 写入参数值到寄存器 */
30     PWR->CR = tmpreg;
31     /* 设置内核寄存器的 SLEEPDEEP 位 */
32     SCB->SCR |= SCB_SCR_SLEEPDEEP;
33
34     /* 设置进入停止模式的方式-----------------*/
35     if (PWR_STOPEntry == PWR_STOPEntry_WFI) {
36         /* 需要中断唤醒 */
37         __WFI();
38     } else {
39         /* 需要事件唤醒 */
40         __WFE();
41     }
42
43     /* 以下的程序是当重新唤醒时才执行的，清除 SLEEPDEEP 位的状态 */
44     SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP);
45 }
46
```

进入停止模式

这个函数有两个输入参数，分别用于控制调压器的模式及选择使用WFI或WFE停止，代码中先是根据调压器的模式配置PWR_CR寄存器，再把内核寄存器的SLEEPDEEP位置1，这样再调用WFI或WFE命令时，STM32就不是睡眠，而是进入停止模式了。函数结尾处的语句用于复位SLEEPDEEP位的状态，由于它是在WFI及WFE指令之后的，所以这部分代码是在STM32被唤醒的时候才会执行。

要注意的是进入停止模式后，STM32的所有I/O都保持在停止前的状态，而当它被唤醒时，STM32使用HSI作为系统时钟(8MHz)运行，由于系统时钟会影响很多外设的工作状态，所以一般我们在唤醒后会重新开启HSE，把系统时钟设置回原来的状态。

进入待机模式

类似地，STM32标准库也提供了控制进入待机模式的函数，其定义如下：

代码清单 41-3 进入待机模式

```
/**
* @brief 进入待机模式
* @note   待机模式时，除以下引脚，其余引脚都在高阻态：
*           -复位引脚
*           - RTC AF1 引脚 (PC13) (需要使能侵入检测、时间戳事件或 RTC 闹钟事件)
*           - RTC_AF2 引脚 (PI8) (需要使能侵入检测或时间戳事件)
*           - WKUP 引脚 (PA0) (需要使能 WKUP 唤醒功能)
* @note  在调用本函数前还需要清除 WUF 寄存器位
* @param  None
* @retval None
*/
void PWR_EnterSTANDBYMode(void)
{
    /* 清除 Wake-up 标志 */
    PWR->CR |= PWR_CR_CWUF;
    /* 选择待机模式 */
    PWR->CR |= PWR_CR_PDDS;
    /* 设置内核寄存器的 SLEEPDEEP 位  */
    SCB->SCR |= SCB_SCR_SLEEPDEEP;
    /* 存储操作完毕时才能进入待机模式，使用以下语句确保存储操作执行完毕 */
#if defined ( __CC_ARM   )
    __force_stores();
#endif
    /* 等待中断唤醒 */
    __WFI();
}
```

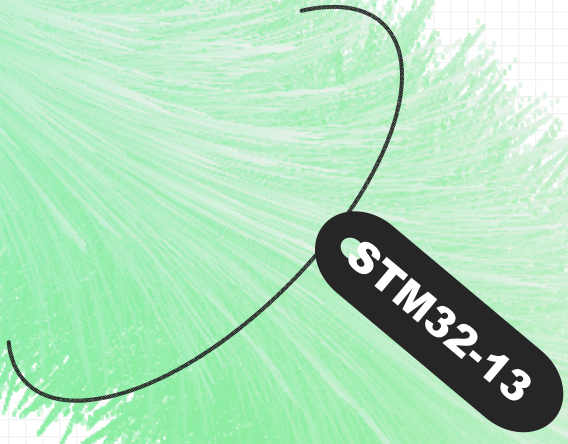
进入待机模式

该函数中先配置了PDDS寄存器位及SLEEPDEEP寄存器位，接着调用__force_stores函数确保存储操作完毕后再调用WFI指令，从而进入待机模式。这里值得注意的是，待机模式也可以使用WFE指令进入的，如果您有需要可以自行修改。

在进入待机模式后，除了被使能了的用于唤醒的I/O，其余I/O都进入高阻态，而从待机模式唤醒后，相当于复位STM32芯片，程序重新从头开始执行。

目录

CONTENTS.



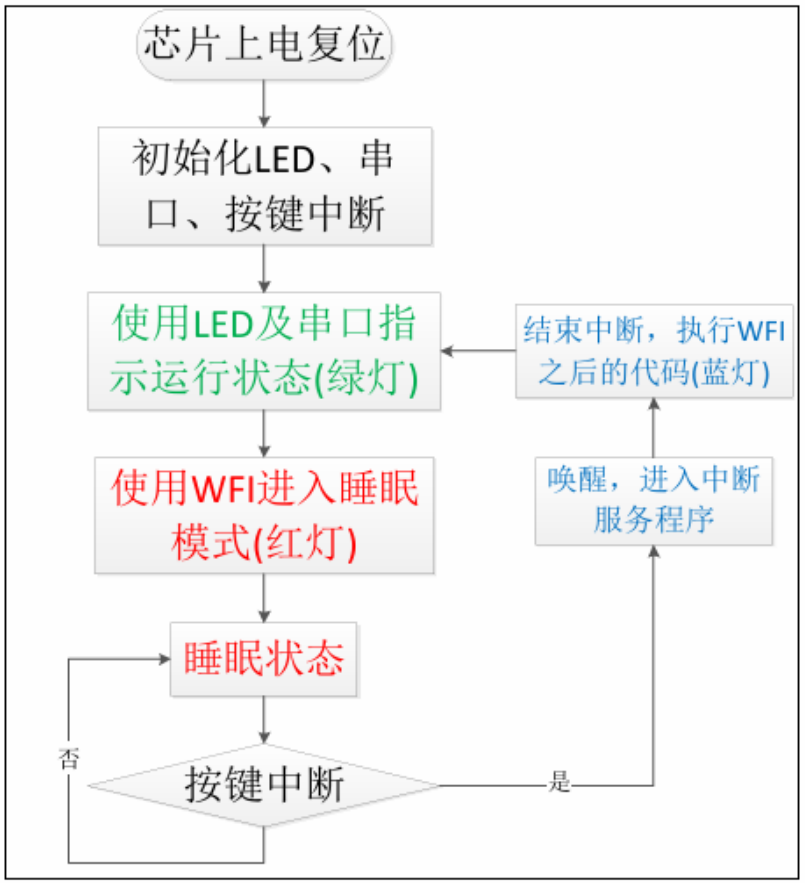
Ex1.睡眠模式实验

程序设计要点：

(1) 初始化用于唤醒的中断按键；

(2) 进入睡眠状态；

(3) 使用按键中断唤醒芯片；



程序流程

Main代码

```c
int main(void)
{

  LED_GPIO_Config();

  /*初始化USART1*/
  USART_Config();

  /* 初始化按键为中断模式，按下中断后会进入中断服务函数 */
  EXTI_Key_Config();

  while(1)
  {
    /**********执行任务******************************/
    printf("\r\n STM32正常运行，亮绿灯\r\n");

    LED_GREEN;
    Delay(0x3FFFFF);

    /******任务执行完毕，进入睡眠降低功耗***********/
    printf("\r\n 进入睡眠模式，按KEY1或KEY2按键可唤醒\r\n");

    //使用红灯指示，进入睡眠状态
    LED_RED;
    //进入睡眠模式
    __WFI();  //WFI指令进入睡眠

    //等待中断唤醒  K1或K2按键中断

    /***被唤醒，亮蓝灯指示***/
    LED_BLUE;
    Delay(0x1FFFFF);

    printf("\r\n 已退出睡眠模式\r\n");
    //继续执行while循环

  }

}
```

SLEEPDEEP缺省为0

进入方式	内核寄存器的SLEEPDEEP = 0，然后调用WFI或WFE指令即可进入睡眠模式； 另外若内核寄存器的SLEEPONEXIT=0时，进入“立即睡眠”模式，SLEEPONEXIT=1时，进入“退出时睡眠”模式。

KEY配置 – 外部中断

```
52  void EXTI_Key_Config(void)
53  {
54    GPIO_InitTypeDef GPIO_InitStructure;
55    EXTI_InitTypeDef EXTI_InitStructure;
56
57    /*开启按键GPIO口的时钟*/
58    RCC_APB2PeriphClockCmd(KEY1_INT_GPIO_CLK,ENABLE);
59
60    /* 配置 NVIC 中断*/
61    NVIC_Configuration();
62
63  /*--------------------------KEY1配置-----------------------------*/
64    /* 选择按键用到的GPIO */
65    GPIO_InitStructure.GPIO_Pin = KEY1_INT_GPIO_PIN;
66    /* 配置为浮空输入 */
67    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
68    GPIO_Init(KEY1_INT_GPIO_PORT, &GPIO_InitStructure);
69
70    /* 选择EXTI的信号源 */
71    GPIO_EXTILineConfig(KEY1_INT_EXTI_PORTSOURCE, KEY1_INT_EXTI_PINSOURCE);
72    EXTI_InitStructure.EXTI_Line = KEY1_INT_EXTI_LINE;
73
74    /* EXTI为中断模式 */
75    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;
76    /* 上升沿中断 */
77    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;
78    /* 使能中断 */
79    EXTI_InitStructure.EXTI_LineCmd = ENABLE;
80    EXTI_Init(&EXTI_InitStructure);
81
82    /*--------------------------KEY2配置-----------------------------*/
83    /* 选择按键用到的GPIO */
```

外部中断服务函数

stm32f10x_it.c

```
148
149  void KEY1_IRQHandler(void)
150  {
151    //确保是否产生了EXTI Line中断
152    if(EXTI_GetITStatus(KEY1_INT_EXTI_LINE) != RESET)
153    {
154      LED_BLUE;
155      printf("\r\n KEY1 按键中断唤醒 \r\n");
156      EXTI_ClearITPendingBit(KEY1_INT_EXTI_LINE);
157    }
158  }
159
160  void KEY2_IRQHandler(void)
161  {
162    //确保是否产生了EXTI Line中断
163    if(EXTI_GetITStatus(KEY2_INT_EXTI_LINE) != RESET)
164    {
165      LED_BLUE;
166      printf("\r\n KEY2 按键中断唤醒 \r\n");
167      //清除中断标志位
168      EXTI_ClearITPendingBit(KEY2_INT_EXTI_LINE);
169    }
170  }
171
```

注意：
当系统处于睡眠模式低功耗状态时(包括后面讲解的停止模式及待机模式)，使用 DAP下载器是无法给芯片下载程序的，所以下载程序时要先把系统唤醒。
或者使用如下方法：
按着板子的复位按键，使系统处于复位状态，然后点击电脑端的下载按钮下载程序，这时再释放复位按键，就能正常给板子下载程序了。

Ex2.停止模式实验

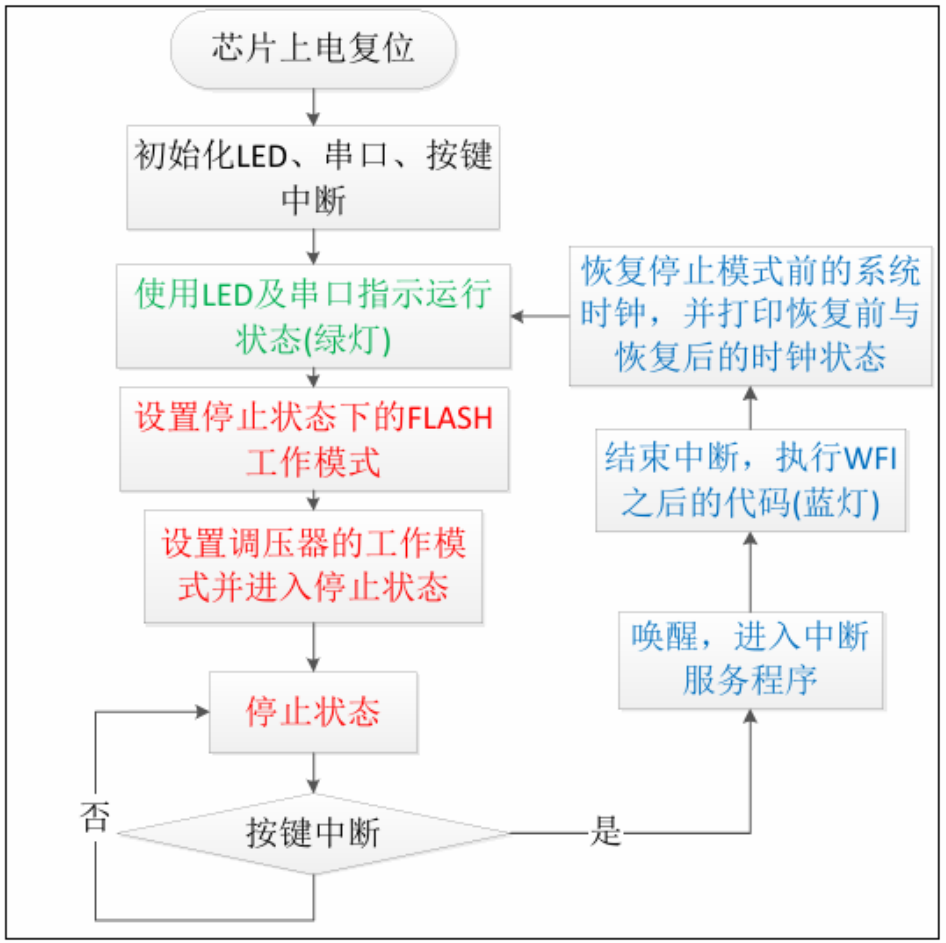
程序设计要点：

(1) 初始化用于唤醒的中断按键；

(2) 选择电压调节器的工作模式并进入停止状态；

(3) 使用按键中断唤醒芯片；

(4) 重启HSE时钟，使系统完全恢复停止前的状态。



程序流程

Main代码

```c
int main(void)
{

  RCC_ClocksTypeDef clock_status_wakeup,clock_status_config;
  uint8_t clock_source_wakeup,clock_source_config;


  LED_GPIO_Config();

  /*初始化USART1*/
  USART_Config();

  /* 初始化按键中断，按下按键后会进入中断服务程序 */
  EXTI_Key_Config();

  while(1)
  {
    /**********执行任务******************************/
    printf("\r\n STM32正常运行，亮绿灯\r\n");

    LED_GREEN;
    Delay(0x3FFFFF);

    /*****任务执行完毕，进入停止降低功耗***********/

    printf("\r\n 进入停止模式，按KEY1或KEY2按键可唤醒\r\n");

    //使用红灯指示，进入停止状态
    LED_RED;

    /* 进入停止模式，设置电压调节器为低功耗模式，等待中断唤醒 */
    PWR_EnterSTOPMode(PWR_Regulator_LowPower,PWR_STOPEntry_WFI);

```

```c
 * @note    在停止模式下所有 I/O 的会保持在停止前的状态
 * @note    从停止模式唤醒后，会使用 HSI 作为时钟源
 * @note    调压器若工作在低功耗模式，可减少功耗，但唤醒时会增加延迟
 * @param   PWR_Regulator: 设置停止模式时调压器的工作模式
 *             @arg PWR_MainRegulator_ON: 调压器正常运行
 *             @arg PWR_Regulator_LowPower: 调压器低功耗运行
 * @param   PWR_STOPEntry: 设置使用 WFI 还是 WFE 进入停止模式
 *             @arg PWR_STOPEntry_WFI: WFI 进入停止模式
 *             @arg PWR_STOPEntry_WFE: WFE 进入停止模式
 * @retval None
 */
void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STO
{
```

Main代码

```
61      /**********************被唤醒*************************/
62
63      //获取刚被唤醒时的时钟状态
64      //时钟源
65      clock_source_wakeup = RCC_GetSYSCLKSource ();
66      //时钟频率
67      RCC_GetClocksFreq(&clock_status_wakeup);
68
69      //从停止模式下被唤醒后使用的是HSI时钟，此处重启HSE时钟,使用PLLCLK
70      SYSCLKConfig_STOP();
71
72      //获取重新配置后的时钟状态
73      //时钟源
74      clock_source_config = RCC_GetSYSCLKSource ();
75      //时钟频率
76      RCC_GetClocksFreq(&clock_status_config);
77
78      //因为刚唤醒的时候使用的是HSI时钟，会影响串口波特率，输出不对，所以在重新配置时钟源后才使用串口输出。
79      printf("\r\n重新配置后的时钟状态：\r\n");
80      printf(" SYSCLK频率:%d,\r\n HCLK频率:%d,\r\n PCLK1频率:%d,\r\n PCLK2频率:%d,\r\n 时钟源:%d (0表示HSI，8表示PLLCLK)\n",
81        clock_status_config.SYSCLK_Frequency,
82        clock_status_config.HCLK_Frequency,
83        clock_status_config.PCLK1_Frequency,
84        clock_status_config.PCLK2_Frequency,
85        clock_source_config);
86
```

Main代码

```
87      printf("\r\n刚唤醒的时钟状态：\r\n");
88      printf(" SYSCLK频率:%d,\r\n HCLK频率:%d,\r\n PCLK1频率:%d,\r\n PCLK2频率:%d,\r\n 时钟源:%d (0表示HSI，8表示PLLCLK)\n",
89        clock_status_wakeup.SYSCLK_Frequency,
90        clock_status_wakeup.HCLK_Frequency,
91        clock_status_wakeup.PCLK1_Frequency,
92        clock_status_wakeup.PCLK2_Frequency,
93        clock_source_wakeup);
94
95      /*指示灯*/
96      LED_BLUE;
97      Delay(0x1FFFFF);
98
99      printf("\r\n 已退出停止模式\r\n");
100     //继续执行while循环
101
102   }
103 }
104
```

参考

```
42 /** @defgroup RCC_Exported_Types
43   * @{
44   */
45
46 typedef struct
47 {
48   uint32_t SYSCLK_Frequency;  /*!< returns SYSCLK clock frequency expressed in Hz */
49   uint32_t HCLK_Frequency;    /*!< returns HCLK clock frequency expressed in Hz */
50   uint32_t PCLK1_Frequency;   /*!< returns PCLK1 clock frequency expressed in Hz */
51   uint32_t PCLK2_Frequency;   /*!< returns PCLK2 clock frequency expressed in Hz */
52   uint32_t ADCCLK_Frequency;  /*!< returns ADCCLK clock frequency expressed in Hz */
53 }RCC_ClocksTypeDef;
54
```

```
900 /**
901   * @brief  Returns the frequencies of different on chip clocks.
902   * @param  RCC_Clocks: pointer to a RCC_ClocksTypeDef structure which wi
903   *         the clocks frequencies.
904   * @note   The result of this function could be not correct when using
905   *         fractional value for HSE crystal.
906   * @retval None
907   */
908 void RCC_GetClocksFreq(RCC_ClocksTypeDef* RCC_Clocks)
909 {
910   uint32_t tmp = 0, pllmull = 0, pllsource = 0, presc = 0;
```

main代码 - 停机唤醒后配置系统时钟

```
113 /**
114   * @brief  停机唤醒后配置系统时钟: 使能 HSE, PLL
115   *         并且选择PLL作为系统时钟.
116   */
117 static void SYSCLKConfig_STOP(void)
118 {
119   /* After wake-up from STOP reconfigure the system clock */
120   /* 使能 HSE */
121   RCC_HSEConfig(RCC_HSE_ON);
122
123   /* 等待 HSE 准备就绪 */
124   while (RCC_GetFlagStatus(RCC_FLAG_HSERDY) == RESET)   {  }
125
126   /* 使能 PLL */
127   RCC_PLLCmd(ENABLE);
128
129   /* 等待 PLL 准备就绪 */
130   while (RCC_GetFlagStatus(RCC_FLAG_PLLRDY) == RESET)   {  }
131
132   /* 选择PLL作为系统时钟源 */
133   RCC_SYSCLKConfig(RCC_SYSCLKSource_PLLCLK);
134
135   /* 等待PLL被选择为系统时钟源 */
136   while (RCC_GetSYSCLKSource() != 0x08)   {  }
137 }
```

外部中断服务函数

stm32f10x_it.c

```
145
146  void KEY1_IRQHandler(void)
147  {
148    //确保是否产生了EXTI Line中断
149    if(EXTI_GetITStatus(KEY1_INT_EXTI_LINE) != RESET)
150    {
151      LED_BLUE;
152      //由于停止唤醒后使用的是HSI时钟，与原来使用的HSE时钟时的频率不一致，会影响波特率，若此处直接printf会乱码
153      //printf("\r\n KEY1 按键中断唤醒 \r\n");
154      EXTI_ClearITPendingBit(KEY1_INT_EXTI_LINE);
155    }
156  }
157
158  void KEY2_IRQHandler(void)
159  {
160    //确保是否产生了EXTI Line中断
161    if(EXTI_GetITStatus(KEY2_INT_EXTI_LINE) != RESET)
162    {
163      LED_BLUE;
164      //由于停止唤醒后使用的是HSI时钟，与原来使用的HSE时钟时的频率不一致，会影响波特率，若此处直接printf会乱码
165      //printf("\r\n KEY2 按键中断唤醒 \r\n");
166      //清除中断标志位
167      EXTI_ClearITPendingBit(KEY2_INT_EXTI_LINE);
168    }
169  }
```