第3章　系统控制

Cortex-M3 处理器系控制部分看起来是由诸多功能杂乱的函数组成的，但是经过仔细分析后，大体上可以划分为 8 个比较清晰的部分：LDO 控制、时钟控制、复位控制、外设控制、睡眠与深度睡眠和杂项功能。

3.1　电源结构与 LDO 控制

1. 电源结构

图 3-1 为 Fury 和 DustDevil 家族电源结构示意图，这是通常的电路接法。3.3V 和 GND 是总电源。VDDA 和 GNDA 是模拟电源，为了减小数字电源对模拟电源的干扰，一般的 PCB 设计方法是：所有 VDDA 就近连接在一起，最后用 1 根导线连接到总电源 3.3V 上；所有的 GNDA 就近连接在一起，最后用 1 根导线连接到总地线 GND 上（俗称“一点接地”）。VDD25 是内核电源，额定工作电压为 2.5V，一般直接由内置的 LDO 提供电源，如果有必要也可以由外部的 2.5V 电源供电。对外的 GPIO 接口采用 VDD 作为驱动电源。

图 3-2 为 Sandstorm 家族的电源结构，相对比较简单。模拟电源没有独立出来，LDO 输出在内部已经连接到内核。

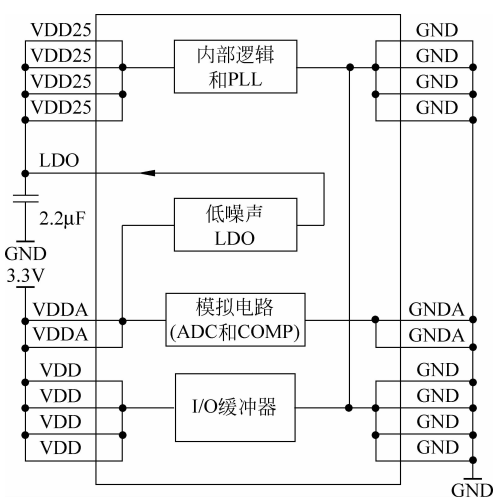

图 3-1　Fury 和 DustDevil 家族电源结构

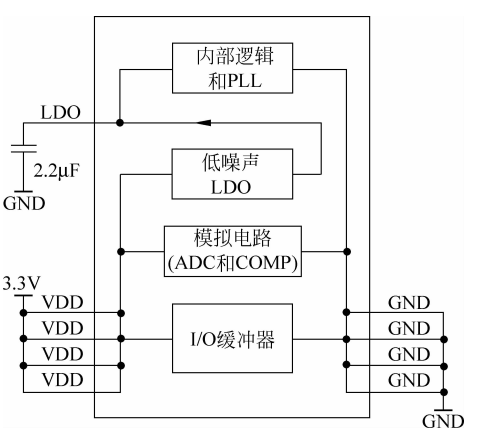

图 3-2　Sandstorm 家族电源结构

注意：这里一直提到的关于 Stellaris 的几个系列。

(1) Sandstorm 家族：LM3S100、LM3S300、LM3S600、LM3S800 系列。

(2) Fury 家族：LM3S1000、LM3S2000、LM3S6000、LM3S8000 系列。

(3) DustDevil 家族：改进集成 USB OTG，可选择主机或设备方式，增加 DMA 和

PWM 输出。

2. 内置的 LDO

LDO 是“Low Drop-Out”的缩写，是一种线性直流电源稳压器。LDO 的显著特点是输入与输出之间的低压差能达到数百毫伏，而传统线性稳压器(如 7805)一般在 1.5V 以上。例如，Exar(原 Sipex)公司的 LDO 芯片 SP6205，当额定的输出为 3.3V/500mA 时，典型压差仅为 0.3V，因此输入电压只要不低于 3.6V 就能满足要求，而效率可高达 90%。这种低压差特性可以带来降低功耗、缩小体积等好处。

Stellaris 系列 ARM 集成有一个内部的 LDO 稳压器，为处理器内核及片内外设提供稳定的电源。这样，只需要为整颗芯片提供单一的 3.3V 电源就能够使其正常工作，简化了系统电源设计并节省了成本。LDO 输出电压默认值是 2.50V，通过软件可以在 2.25～2.75V 之间调节，步进 50mV。降低 LDO 输出电压可以节省功耗。LDO 引脚除了给处理器内核供电以外，还可以为芯片以外的电路供电，但是要注意控制电流大小和电压波动，以免干扰处理器内核的正常运行。

片内 LDO 输入电压是芯片电源 VDDA(额定 3.3V)，LDO 输出到一个名为“LDO”的引脚。对于 Fury 和 DustDevil 家族(LM3S1000 以上型号)，LDO 引脚要连接到内核电源 VDD25 引脚上，对于 Sandstorm 家族(LM3S1000 以下型号)，VDD25 引脚是内置的，因此不必从外部连接。参考图 3-1 和图 3-2。

注意：在 LDO 引脚和 GND 之间必须接一个 1～3.3μF 的瓷片电容，推荐值是 2.2μF。在启用片内锁相环 PLL 之前，必须要将 LDO 电压设置在最高的 2.75V，否则可能造成系统工作异常。

3. LDO 控制库函数

控制 LDO 的库函数有 3 个。函数 SysCtlLDOSet()用来设置 LDO 的输出电压，在需要节省功耗时可以调得低一些，在耗电较大的应用场合要调得高一些，在启用 PLL 之前必须设置在最高的 2.75V，如表 3-1 所示。

表 3-1 函数 SysCtlLDOSet()

函数名称	SysCtlLDOSet()
功能	设置 LDO 的输出电压
原型	void SysCtlLDOSet(unsigned long ulVoltage)
参数	ulVoltage：要设置的 LDO 输出电压，应当取下列值之一： SYSCTL_LDO_2_25V //LDO 输出 2.25V SYSCTL_LDO_2_30V //LDO 输出 2.30V SYSCTL_LDO_2_35V //LDO 输出 2.35V SYSCTL_LDO_2_40V //LDO 输出 2.40V SYSCTL_LDO_2_45V //LDO 输出 2.45V SYSCTL_LDO_2_50V //LDO 输出 2.50V SYSCTL_LDO_2_55V //LDO 输出 2.55V SYSCTL_LDO_2_60V //LDO 输出 2.60V SYSCTL_LDO_2_65V //LDO 输出 2.65V SYSCTL_LDO_2_70V //LDO 输出 2.70V SYSCTL_LDO_2_75V //LDO 输出 2.75V
返回	无
备注	复位后，默认的 LDO 输出电压是 2.50V。在启用 PLL 之前，必须将 LDO 输出设置在最高的 2.75V，否则可能造成系统工作异常

函数 SysCtlLDOGet()用来获取 LDO 当前的输出电压值。获取的电压值为 2.25～2.75V,步长为 50mV,如表 3-2 所示。

表 3-2　函数 SysCtlLDOGet()

函数名称	SysCtlLDOGet()
功能	获取 LDO 的电压输出值
原型	unsigned long SysCtlLDOGet(void)
参数	无
返回	LDO 当前电压值,与表 3-1 当中参数 ulVoltage 的取值相同

函数 SysCtlLDOConfigSet()用来管理 LDO 故障,主要用于在 LDO 出现故障时,是否允许处理器产生复位信号,如表 3-3 所示。这个函数通常用不到。

表 3-3　函数 SysCtlLDOConfigSet()

函数名称	SysCtlLDOConfigSet()
功能	配置 LDO 失效控制
原型	void SysCtlLDOConfigSet(unsigned long ulConfig)
参数	ulConfig: 所需 LDO 故障控制的配置,应当取下列值之一: SYSCTL_LDOCFG_ARST　　//允许 LDO 故障时产生复位 SYSCTL_LDOCFG_NORST　//禁止 LDO 故障时产生复位
返回	无

4. LDO 控制例程

程序清单 3-1 是控制 LDO 输出电压的示例。在程序中,数组 ulTab[ ]保存所有 LDO 可能的设置电压,在主循环里,每隔 3.5 秒利用 SysCtlLDOSet()函数修改一次 LDO 输出电压值,同时发送到 UART 显示。在 3.5 秒间隔里,可以拿万用表来测量 LDO 引脚的实际电压值大小。

程序清单 3-1　SysCtl 例程: 控制 LDO 输出电压

```
文件: main.c
#include "systemInit.h"
#include "uartGetPut.h"
#include <stdio.h>
//主函数(程序入口)
int main(void)
{
  const unsigned long ulTab[11]=                    //定义 LDO 电压数值表
  {
      SYSCTL_LDO_2_25V,
      SYSCTL_LDO_2_30V,
      SYSCTL_LDO_2_35V,
      SYSCTL_LDO_2_40V,
      SYSCTL_LDO_2_45V,
      SYSCTL_LDO_2_50V,
      SYSCTL_LDO_2_55V,
      SYSCTL_LDO_2_60V,
      SYSCTL_LDO_2_65V,
```

```
        SYSCTL_LDO_2_70V,
        SYSCTL_LDO_2_75V
        };
int i;
  char s[40];
  clockInit();                                    //时钟初始化：晶振,6MHz
  uartInit();                                     //UART 初始化
for(;;)
  {
    for(i=0; i < 11; i++)
  {
    SysCtlLDOSet(ulTab[i]);                       //设置 LDO 输出电压
    sprintf(s,"LDO=2.%d(V)\r\n",25 + 5 * i);      //显示 LDO 电压值
    uartPuts(s);
    SysCtlDelay(3500 * (TheSysClock / 3000));     //延时约 3500ms,以便观察 LDO 输出
  }
  uartPuts("\r\n");
  }
}
```

例程分析：这个例程中主要使用了 SysCtlLDOSet()函数来实现 LDO 电压的设置，同时将 LDO 电压值通过串口输出，读者可以尝试使用 SysCtlLDOGet()函数来读取 LDO 电压值。

3.2 时钟控制

1. 时钟系统框图

图 3-3 为 Sandstorm 家族(以 LM3S615 为典型)的时钟系统结构图。时钟来源是主振荡器(MOSC)或 12MHz 内部振荡器(IOSC)，最终产生的系统时钟(System Clock)用于 Cortex-M3 处理器内核以及大多数片内外设，PWM(脉宽调制)时钟在系统时钟基础上进一步分频获得，ADC(模—数转换)时钟是恒定分频的 16.667MHz 输出(一般要求启用锁相环 PLL)。

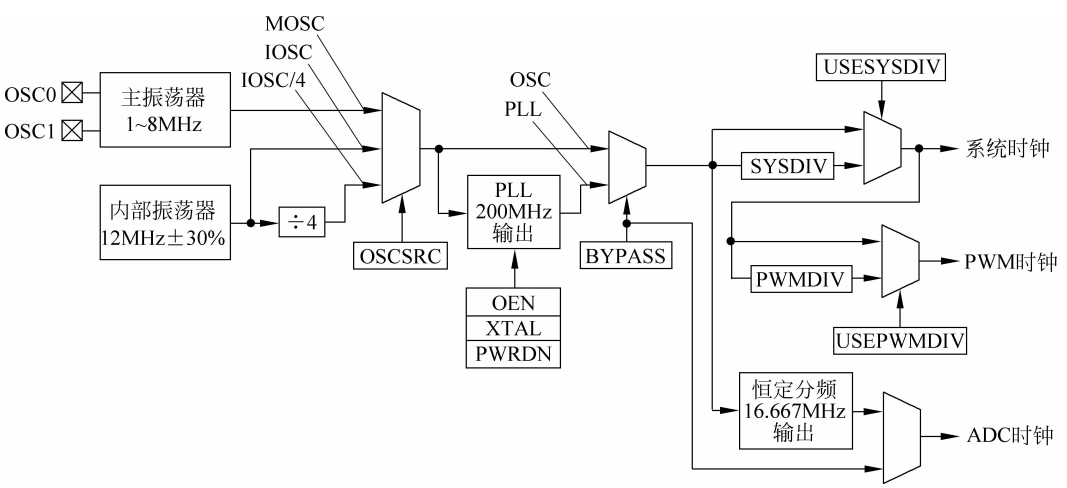

图 3-3 Sandstorm 家族时钟系统

图 3-4 为 Fury 家族(以 LM3S8962 为典型)的时钟系统结构图,要比 LM3S615 复杂些。时钟来源除了 MOSC 和 IOSC 以外,还可以是 30kHz 内部振荡器(INT30),以及来自冬眠模块的 32.768kHz 外部有源振荡器(EXT32)。ADC 时钟有两个来源可以选择。

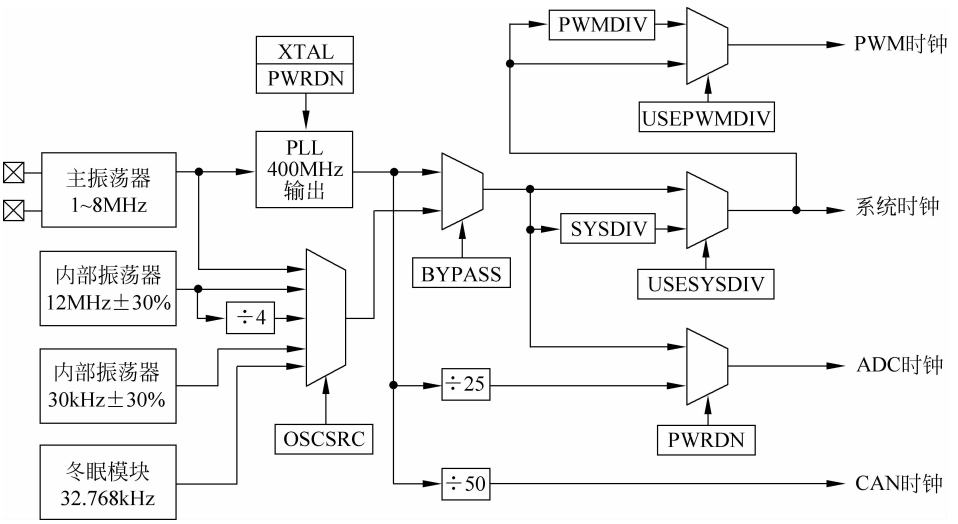

图 3-4　Fury 家族时钟系统

图 3-5 为 DustDevil 家族(以 LM3S5749 为典型)的时钟系统结构图,新增了一个 USB 模块专用的 PLL 单元,输出 240MHz,经 4 分频后作为 USB 时钟。

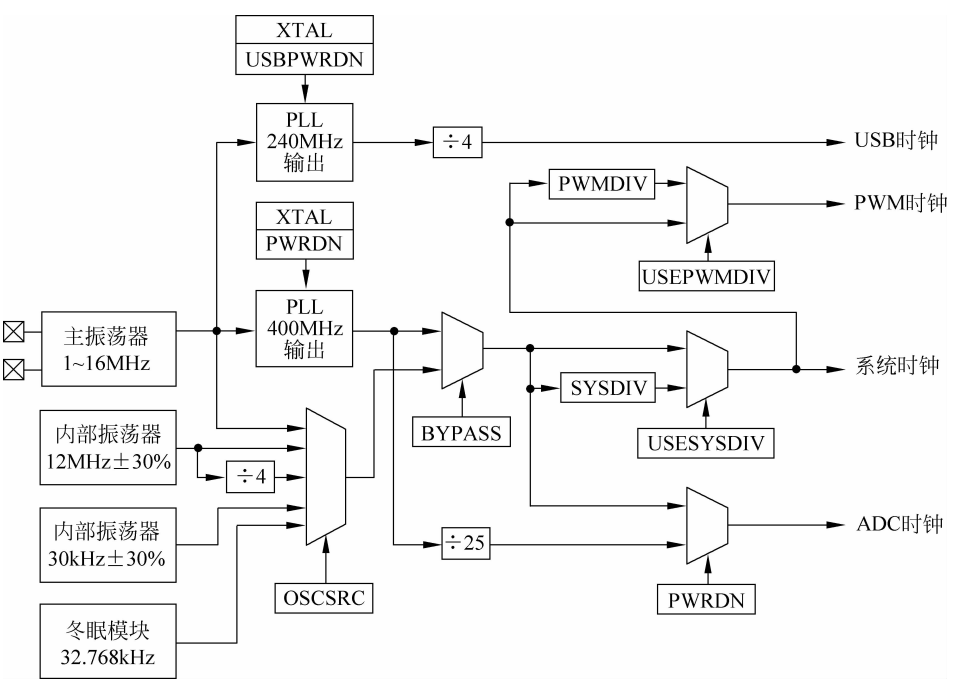

图 3-5　DustDevil 家族时钟系统

2. 振荡器

主振荡器 MOSC 可以连接一个 1～8.192MHz 的外部晶体(DustDevil 家族可以支持到 16.384MHz)。典型接法如图 3-6 所示,电阻 R_1 可以加速起振过程,C_1 和 C_2 取值要适当,不宜过大或过小。如果不使用晶体,则外部的有源振荡信号也可以从 OSC0 引脚输入,要求信号幅度介于 0～3.3V 之间,此时 OSC1 引脚应当悬空。

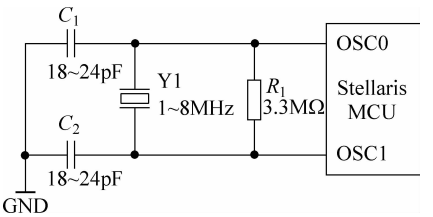

图 3-6 MOSC 外接晶振典型用法

Sandstorm 家族上电默认采用 MOSC,如果不接晶振也不提供外部振荡信号输入,则无法启动。Fury 和 DustDevil 家族上电默认采用 IOSC,如果软件上没有配置 MOSC,则外部晶振不会起振。

Stellaris 系列 ARM 集成有两个内部振荡器,一个是 12MHz 高速振荡器 IOSC,一个是 30kHz 低速振荡器 INT30(Sandstorm 家族没有 30kHz 振荡器)。内部振荡器误差较大,约±30%,这是由于 IC 制造工艺的特点形成的,因此对时钟精度有严格要求的场合不适宜采用内部振荡器。IOSC 经 4 分频后(IOSC/4)标称为 3MHz,也可以作为系统时钟的一个来源。芯片在较低的时钟速率下运行能够明显节省功耗。

对于集成有冬眠模块(Hibernation Module)的型号(Sandstorm 家族都没有冬眠模块),在冬眠备用电源引脚 VBAT 正常供电的情况下,可以从冬眠专用的晶振引脚 XOSC0 输入 32.768kHz 的外部有源振荡信号(不直接支持 32.768kHz 晶体)作为系统时钟源。

3. 锁相环 PLL

Stellaris 系列 ARM 内部集成有一个锁相环(Phase Locked Loop,PLL)。PLL 输出频率固定为 400MHz(Sandstorm 家族为 200MHz),误差为±1%。如果选用 PLL,则 MOSC 频率必须在 3.579545～8.192MHz 之间才能使 PLL 精确地输出 400MHz。

经 OSC 或 PLL 产生的时钟可以经过 1～64 分频(Sandstorm 家族只能支持到 16 分频)后得到系统时钟(System Clock),分频数越大越省电。

注意:由于 Cortex-M3 内核最高运行频率为 50MHz,因此如果要使用 PLL,则至少要进行 4 以上的分频(硬件会自动阻止错误的软件配置)。启用 PLL 后,系统功耗将明显增大。

在所有型号中,不论 PLL 输出是 200MHz 还是 400MHz,只要分频数相同,则对 PLL 的分频结果都是一样的,统一按照 200MHz 进行计算。例如 LM3S615 芯片的 PLL 是 200MHz 输出,LM3S1138 芯片的 PLL 是 400MHz 输出,但执行以下函数调用后,最终的系统时钟都是 20MHz:

```
SysCtlClockSet(SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ | SYSCTL_
SYSDIV_10);
```

4. PWM 和 ADC 时钟

PWM(脉宽调制)模块的时钟(PWM Clock)是在系统时钟基础上经进一步分频得到的,允许的分频数是:1、2、4、8、16、32、64,参见表 3-6 对函数 SysCtlPWMClockSet()的描述。

Stellaris 系列 ARM 内部集成有 10 位 ADC 模块,不同型号的采样速率也不同:125K、

250K、500K、1M。单位：sps(次采样/秒)。例如 LM3S8938 的 ADC 采样速率是 1Msps。ADC 模块要求工作在额定的 16MHz 时钟下才能保证±1LSB 的精度(IC 工艺原因)，而每采样一次需要 16 个时钟周期。对于实际采样速率达不到 1M 的型号，ADC 模块还可以对输入的 16MHz 时钟进行分频以获得恰当的工作时钟速率，参见表 3-8 对函数 SysCtlADCSpeedSet()的描述。

针对 ADC 时钟有 16MHz 额定输入的这一要求，可以采用两种方法提供：一是启用 PLL 单元，固定的分频数可以保证 ADC 时钟在 16MHz 左右(参见图 3-3 和图 3-4)，可能存在的问题是功耗比较大；二是采用 16MHz 或 16.384MHz 晶振，好处是功耗较低。当然，有很多型号直接支持的晶振只能达到 8.192MHz，对于这种情况可以考虑从 OSC0 引脚直接输入 16MHz 的有源振荡信号。

5. 时钟控制库函数

SysCtlClockSet()是个功能复杂的库函数，负责系统时钟功能的设置。SysCtlClockGet()函数用来获取已设置的系统时钟频率，参见表 3-4 和表 3-5 的描述。

表 3-4　函数 SysCtlClockSet()

函数名称	SysCtlClockSet()
功能	系统时钟设置
原型	void SysCtlClockSet(unsigned long ulConfig)
参数	ulConfig：时钟配置字，应当取下列各组数值之间的“或运算”组合形式。 • 系统时钟分频值 SYSCTL_SYSDIV_1　//振荡器不分频(不可用于 PLL) SYSCTL_SYSDIV_2　//振荡器 2 分频(不可用于 PLL) SYSCTL_SYSDIV_3　//振荡器 3 分频(不可用于 PLL) SYSCTL_SYSDIV_4　//振荡器 4 分频，或对 PLL 的分频结果为 50MHz SYSCTL_SYSDIV_5　//振荡器 5 分频，或对 PLL 的分频结果为 40MHz ⋮ SYSCTL_SYSDIV_64　//振荡器 64 分频，或对 PLL 的分频结果为 3.125MHz **注**：对 Sandstorm 家族最大分频数只能取到 16。不同型号 PLL 输出为 200MHz 或 400MHz，但分频时都按 200MHz 进行计算，这保持了软件上的兼容性。由于 Cortex-M3 内核最高工作频率为 50MHz，因此启用 PLL 时必须进行 4 以上的分频(硬件会自动阻止错误的软件配置)。 • 使用 OSC 还是 PLL SYSCTL_USE_PLL　//采用锁相环 PLL 作为系统时钟源 SYSCTL_USE_OSC　//采用 OSC(主振荡器或内部振荡器)作为系统时钟源 **注**：如果选用 PLL 作为系统时钟，则本函数将询问 PLL 锁定中断状态位来确定 PLL 是何时锁定的，PLL 锁定时间最多不会超过 0.5ms。由于启用 PLL 时会消耗较大的功率，因此在启用 PLL 之前，要求必须先将 LDO 电压设置在 2.75V，否则可能造成芯片工作不稳定。 • OSC 时钟源选择 SYSCTL_OSC_MAIN　//主振荡器作为 OSC SYSCTL_OSC_INT　//内部 12MHz 振荡器作为 OSC SYSCTL_OSC_INT4　//内部 12MHz 振荡器 4 分频后作为 OSC

续表

参数	SYSCTL_OSC_INT30 //内部 30kHz 振荡器作为 OSC SYSCTL_OSC_EXT32 //外接 32.768kHz 有源振荡器作为 OSC **注**：内部 12MHz、30kHz 振荡器有±30%的误差，对时钟精度有严格要求的场合不适宜采用。采用内部 30kHz 和外部 32.768kHz 振荡器，能够明显节省功耗，但是 Sandstorm 家族不支持这两种低频振荡器。采用外部 32.768kHz 振荡器时，不能直接用晶体而必须是从 XOSC0 引脚输入的有源振荡信号，并且要保证冬眠模块(Hibernation Module)VBAT 引脚的正常供电。 • 外接晶体频率 SYSCTL_XTAL_1MHZ //外接晶体 1MHz SYSCTL_XTAL_1_84MHZ //外接晶体 1.8432MHz SYSCTL_XTAL_2MHZ //外接晶体 2MHz SYSCTL_XTAL_2_45MHZ //外接晶体 2.4576MHz SYSCTL_XTAL_3_57MHZ //外接晶体 3.579545MHz SYSCTL_XTAL_3_68MHZ //外接晶体 3.6864MHz SYSCTL_XTAL_4MHZ //外接晶体 4MHz SYSCTL_XTAL_4_09MHZ //外接晶体 4.096MHz SYSCTL_XTAL_4_91MHZ //外接晶体 4.9152MHz SYSCTL_XTAL_5MHZ //外接晶体 5MHz SYSCTL_XTAL_5_12MHZ //外接晶体 5.12MHz SYSCTL_XTAL_6MHZ //外接晶体 6MHz SYSCTL_XTAL_6_14MHZ //外接晶体 6.144MHz SYSCTL_XTAL_7_37MHZ //外接晶体 7.3728MHz SYSCTL_XTAL_8MHZ //外接晶体 8MHz SYSCTL_XTAL_8_19MHZ //外接晶体 8.192MHz SYSCTL_XTAL_10MHZ //外接晶体 10MHz SYSCTL_XTAL_12MHZ //外接晶体 12MHz SYSCTL_XTAL_12_2MHZ //外接晶体 12.288MHz SYSCTL_XTAL_13_5MHZ //外接晶体 13.56MHz SYSCTL_XTAL_14_3MHZ //外接晶体 14.31818MHz SYSCTL_XTAL_16MHZ //外接晶体 16MHz SYSCTL_XTAL_16_3MHZ //外接晶体 16.384MHz **注**：对于 2008 年新推出的 DustDevil 家族，如 LM3S9B96，外接晶体频率可以达到 16.384MHz，以前的型号只能达到 8.192MHz，详细情况请以具体型号的数据手册为准。启用 PLL 时，所支持的晶振频率必须在(3.57～8.192)MHz 之间，否则可能造成失锁。 • 振荡源禁止 SYSCTL_INT_OSC_DIS //禁止内部振荡器 SYSCTL_MAIN_OSC_DIS //禁止主振荡器 **注**：禁止不用的振荡器可以节省功耗。为了能够使用外部时钟源，主振荡器必须被使能，试图禁止正在为芯片提供时钟的振荡器会被硬件阻止。
返回	无

例程分析。

(1) 采用 6MHz 晶振作为系统时钟：

```
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ |
               SYSCTL_SYSDIV_1);
```

(2) 采用 16MHz 晶振 4 分频作为系统时钟：

```
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_MAIN | SYSCTL_XTAL_16MHZ |
               SYSCTL_SYSDIV_4);
```

(3) 采用内部 12MHz 振荡器作为系统时钟：

```
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_INT | SYSCTL_SYSDIV_1);
```

(4) 采用内部 12MHz 振荡器 4 分频作为系统时钟：

```
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_INT4 | SYSCTL_SYSDIV_1);
```

(5) 采用内部 30kHz 振荡器作为系统时钟：

```
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_INT30 | SYSCTL_SYSDIV_1);
```

(6) 外接 6MHz 晶体，采用 PLL 作为系统时钟，分频结果为 20MHz：

```
SysCtlLDOSet(SYSCTL_LDO_2_75V);
SysCtlClockSet(SYSCTL_USE_PLL | SYSCTL_OSC_MAIN | SYSCTL_XTAL_6MHZ |
               SYSCTL_SYSDIV_10);
```

表 3-5　函数 SysCtlClockGet()

函数名称	SysCtlClockGet()
功能	获取系统时钟速率
原型	unsigned long SysCtlClockGet(void)
参数	无
返回	返回当前配置的系统时钟速率，单位：Hz
备注	如果在调用本函数之前，从没有通过调用函数 SysCtlClockSet()来配置时钟，或者时钟直接由一个晶体(或外部时钟源)来提供而该晶体(或外部时钟源)并不属于支持的标准晶体频率，则不会返回精确的结果

函数 SysCtlPWMClockSet()和 SysCtlPWMClockGet()用来管理 PWM 模块的时钟，参见表 3-6 和表 3-7 的描述。

表 3-6　函数 SysCtlPWMClockSet()

函数名称	SysCtlPWMClockSet()
功能	设置 PWM 时钟的预分频数
原型	void SysCtlPWMClockSet(unsigned long ulConfig)
参数	ulConfig：PWM 时钟配置，应当取下列值之一： SYSCTL_PWMDIV_1　　//PWM 时钟预先进行 1 分频(不分频) SYSCTL_PWMDIV_2　　//PWM 时钟预先进行 2 分频 SYSCTL_PWMDIV_4　　//PWM 时钟预先进行 4 分频

续表

参数	SYSCTL_PWMDIV_8 //PWM 时钟预先进行 8 分频 SYSCTL_PWMDIV_16 //PWM 时钟预先进行 16 分频 SYSCTL_PWMDIV_32 //PWM 时钟预先进行 32 分频 SYSCTL_PWMDIV_64 //PWM 时钟预先进行 64 分频
返回	无

表 3-7 函数 SysCtlPWMClockGet()

函数名称	SysCtlPWMClockGet()
功能	获取 PWM 时钟的预分频数
原型	unsigned long SysCtlPWMClockGet(void)
参数	无
返回	返回 PWM 时钟的预分频数，与表 3-6 当中参数 ulConfig 的取值相同

函数 SysCtlADCSpeedSet()和 SysCtlADCSpeedGet()用来管理 ADC 模块的时钟(速度)，参见表 3-8 和表 3-9 的描述。

表 3-8 函数 SysCtlADCSpeedSet()

函数名称	SysCtlADCSpeedSet()
功能	设置 ADC 的采样速度
原型	void SysCtlADCSpeedSet(unsigned long ulSpeed)
参数	ulSpeed：采样速度，取下列值之一： SYSCTL_ADCSPEED_1MSPS //采样速率：1M 次采样/秒 SYSCTL_ADCSPEED_500KSPS //采样速率：500K 次采样/秒 SYSCTL_ADCSPEED_250KSPS //采样速率：250K 次采样/秒 SYSCTL_ADCSPEED_125KSPS //采样速率：125K 次采样/秒
返回	无

表 3-9 函数 SysCtlADCSpeedGet()

函数名称	SysCtlADCSpeedGet()
功能	获取 ADC 的采样速度
原型	unsigned long SysCtlADCSpeedGet(void)
参数	无
返回	返回 ADC 的采样速度，与表 3-8 当中参数 ulSpeed 的取值相同

6. 时钟控制例程

程序清单 3-2 演示了系统时钟设置函数 SysCtlClockSet()函数的用法。在程序中，函数 ledFlash()可以设 LED 指示灯闪烁数次，采用固定周期数的延时函数 delay()。在主循环里，系统时钟采用不同的配置，结果 LED 闪烁速度随着变快或者变慢。要注意设置 PLL 的要点：必须首先将 LDO 的输出电压设置在 2.75V。这是因为启用 PLL 后系统功耗会立即增大许多，如果 LDO 电压不够高则容易造成芯片工作不稳定。

程序清单 3-2　SysCtl 例程：系统时钟设置

```
#include "systemInit.h"
//定义 LED
#define LED_PERIPH        SYSCTL_PERIPH_GPIOF
#define LED_PORT          GPIO_PORTF_BASE
#define LED_PIN           GPIO_PIN_2

//延时
void delay(unsigned long ulVal)
{
    while(--ulVal !=0);
}

//LED 闪烁 usN 次
void ledFlash(unsigned short usN)
{
    do
    {
        GPIOPinWrite(LED_PORT,LED_PIN,0x00);      //点亮 LED
        delay(200000UL);

        GPIOPinWrite(LED_PORT,LED_PIN,1 << 2);  //熄灭 LED
        delay(300000UL);
    } while(--usN !=0);
}

//主函数(程序入口)
int main(void)
{
    SysCtlPeriEnable(LED_PERIPH);                    //使能 LED 所在的 GPIO 端口
    GPIOPinTypeOut(LED_PORT,LED_PIN);                //设置 LED 所在引脚为输出
    for(;;)
    {
        SysCtlLDOSet(SYSCTL_LDO_2_50V);              //设置 LDO 输出电压
        SysCtlClockSet(SYSCTL_USE_OSC |              //系统时钟设置
                       SYSCTL_OSC_MAIN |             //采用主振荡器
                       SYSCTL_XTAL_6MHZ |            //外接 6MHz 晶体
                       SYSCTL_SYSDIV_3);             //3 分频
        ledFlash(5);                                 //2MHz 系统时钟,缓慢闪烁
        SysCtlClockSet(SYSCTL_USE_OSC |              //系统时钟设置
                       SYSCTL_OSC_INT |              //内部振荡器(12MHz±30%)
                       SYSCTL_SYSDIV_2);             //2 分频
        ledFlash(8);                                 //6MHz 系统时钟,较快闪烁
        SysCtlLDOSet(SYSCTL_LDO_2_75V);              //配置 PLL 前须将 LDO 设为 2.75V
        SysCtlClockSet(SYSCTL_USE_PLL |              //系统时钟设置,采用 PLL
                       SYSCTL_OSC_MAIN |             //主振荡器
```

```
                SYSCTL_XTAL_6MHZ |          //外接 6MHz 晶振
                SYSCTL_SYSDIV_10);         //分频结果为 20MHz
        ledFlash(12);                      //20MHz 系统时钟,快速闪烁
    }
}
```

代码分析：通过设置 SysCtlClockSet()的参数设置当前处理器主运行时钟，从而改变时钟周期，延时的时间也有改变，从而导致 LED 闪烁的间隔发生改变。SysCtlClockSet()这个函数非常重要，基本上处理器在初始化时都要用到这个函数。

3.3 复位控制

1. 复位源

在 Stellaris 系列 ARM 有多种复位源，所有复位标志都集中保存在一个复位原因寄存器(RSTC)里。

Stellaris 系列一共有 6 个复位源：

(1) 外部复位输入引脚(RST)有效(Assertion)；

(2) 上电复位(POR)上电时，芯片自动复位，称为“上电复位”(Power On Reset)，上电复位后，POR 标志置位；

(3) 内部掉电 BOR(Brown-Out Reset)；

(4) 由软件启动的复位 SW(利用软件复位寄存器)；

(5) 看门狗定时器复位 WDT；

(6) LDO 复位。

- 外部复位(EXT)芯片正在工作时，如果复位引脚/RST 被拉低、延迟、再拉高，则芯片产生复位。这种复位称为“外部复位”(External Reset)。外部复位后，EXT 标志置位，其他标志(POR 除外)都被清零。

图 3-7 为实用的 RC 复位电路。R_1 和 C_1 构成基本的 RC 电路，上电瞬间会输出一个 RC 充电信号(e 指数曲线)，$\overline{\text{RST}}$引脚在内部是一个施密特输入结构，能把缓慢变化的 RC 曲线整形成为一个负脉冲，作为内部逻辑的 Reset 信号。按下复位键 KEY 可以强制 C_1 放电，形成手动复位。R_2 起的作用是限制放电电流，如果没有 R_2 则按键时由于 C_1 正负极瞬间短路会形成很大的冲击电流。断电再上电时，二极管 D_1 可以加速 C_1 上的放电速度，改善复位响应速度。图 3-8 为 CAT811 集成复位电路，S 型复位门槛电压是 2.93V，建议在$\overline{\text{RST}}$引脚接一个下拉电阻。在上电时或按下 KEY 时，CAT811 会自动输出一个宽度为 240ms(典型值)的低电平复位脉冲。

外部复位引脚(nRST)可将微控制器复位。该复位信号使内核及所有外设复位，JTAG TAP 控制器除外，复位序列如下。

(1) 外部复位引脚 nRST 有效(输出低电平)，然后失效(输出高电平)。

(2) 在 nRST 失效之后，必须给晶体振荡器一定的时间允许它稳定下来，控制器内部有一个主振荡器对这段时间 t7 计数(15～30ms)。在此期间，控制器其余部分的内部复位保

持有效。

(3) 内部复位释放,微控制器加载初始堆栈指针和初始程序计数器,并取出由程序计数器指定的第一条指令,然后开始执行,复位时序见图 3-9。

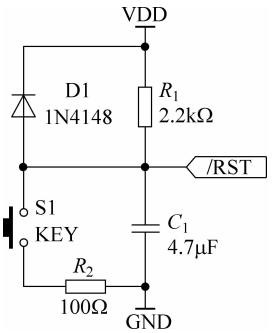

图 3-7　实用的 RC 复位电路

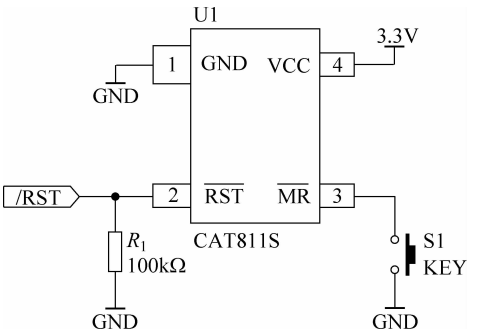

图 3-8　CAT811 集成复位电路

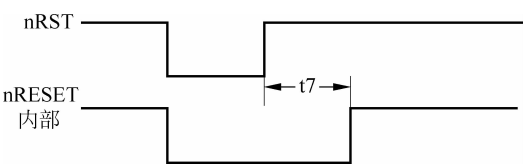

图 3-9　外部复位时序图

① 软件复位(SW)

芯片正在工作时,执行函数 SysCtlReset()会产生软件复位(Software Reset),效果跟外部复位相同。软件复位后,SW 标志置位,其他复位标志不变。

② 看门狗复位(WDT)

如果使能了看门狗模块的复位功能,若因为没有及时“喂狗”而产生的复位称为“看门狗复位”(WatchDog Reset)。看门狗复位后,WDT 标志置位,其他复位标志不变。

③ 掉电复位(BOR)

掉电检测的结果可以用来触发中断或产生复位,如果用于产生复位,则这种复位称为掉电复位(Brown Out Reset)。掉电复位后,BOR 标志被置位,其他复位标志不变。

注意:掉电不是“断电”。掉电一词的英文是 Brown Out,其本意是“把灯火弄暗”(不是“弄灭”)。“断电”指芯片的供电被彻底切断,没有了电源,芯片的一切功能都谈不上了;掉电指芯片原先供电正常,后来供电跌落到某个较低的电压值时的工作状态。掉电检测功能能够自动查知掉电过程(门槛电压标称值为 2.9V)。

④ LDO 复位(LDO)

当 LDO 供电不可调整时,例如 LDO 输出引脚在短时间内被强制接到 GND,芯片所产生的复位称为 LDO 供电不可调整复位(LDO Power Not OK Reset),简称 LDO 复位。LDO 复位后,LDO 标志置位,其他标志不变。

⑤ 上电复位

上电 POR 电路检测电源是否上升,并在检测到电压上升到阈值(V_{TH})时,产生片内复位脉冲。为使用片内电路,nRST 输入需要连接一个上拉电阻((1～10)kΩ)。在片内上电

复位脉冲结束时，器件必须在指定的工作范围内操作。指定的工作参数包括电源电压、频率和温度等。如果在 POR 结束时没有满足工作条件，则处理器不能正常工作。

2. 复位控制函数

SysCtlReset()是软件复位函数，调用后芯片产生一次热复位，参见表 3-10 的描述。

表 3-10 函数 SysCtlReset()

函数名称	SysCtlReset()
功能	软件复位
原型	void SysCtlReset(void)
参数	无
返回	无

函数 SysCtlResetCauseClear()和 SysCtlResetCauseGet()用来管理复位原因，参见表 3-11 和表 3-12 的描述。

表 3-11 函数 SysCtlResetCauseClear()

函数名称	SysCtlResetCauseClear()
功能	清除芯片的复位原因
原型	void SysCtlResetCauseClear(unsigned long ulCauses)
参数	ulCauses：要清除的复位源，应当取下列值之一或者它们之间的任意“或运算”组合形式。 SYSCTL_CAUSE_LDO //LDO 供电不可调整引起的复位 SYSCTL_CAUSE_SW //软件复位 SYSCTL_CAUSE_WDOG //看门狗复位 SYSCTL_CAUSE_BOR //掉电复位 SYSCTL_CAUSE_POR //上电复位 SYSCTL_CAUSE_EXT //外部复位
返回	无

表 3-12 函数 SysCtlResetCauseGet()

函数名称	SysCtlResetCauseGet()
功能	获取芯片复位的原因
原型	unsigned long SysCtlResetCauseGet(void)
参数	无
返回	复位的原因，与表 3-11 当中参数 ulCauses 的取值相同

函数 SysCtlBrownOutConfigSet()用来管理掉电时的动作，可以产生一次复位或中断，参见表 3-13 的描述。

表 3-13 函数 SysCtlBrownOutConfigSet()

函数名称	SysCtlBrownOutConfigSet()
功能	配置掉电控制

续表

原型	void SysCtlBrownOutConfigSet(unsigned long ulConfig,unsigned long ulDelay)
参数	ulConfig：希望的掉电控制的配置，应当取下列值之间的任意“或运算”组合形式。 SYSCTL_BOR_RESET　　//复位代替中断 SYSCTL_BOR_RESAMPLE　　//在生效之前重新采样 BOR ulDelay：重新采样一个有效的掉电信号之前要等待内部振荡器周期数，该值只在 SYSCTL_BOR_RESAMPLE 被设置后并且小于 8192 时才有意义
返回	无

3. 复位控制例程

程序清单 3-3 演示了几个系统复位控制函数的用法。首次上电时，通过 UART 显示“Power on reset”和“External reset”；如果按下“复位”按钮，则显示“External reset”；不去按键，稍等一会儿会自动执行软件复位，显示“Software reset”。如果还存在其他可能的复位方式，也会正确显示出来。

程序清单 3-3　SysCtl 例程：系统复位控制

```
#include "systemInit.h"
#include "uartGetPut.h"
#include <stdio.h>
//主函数(程序入口)
int main(void)
{
    unsigned long ulCauses;
        clockInit();                                    //时钟初始化：晶振,6MHz
        uartInit();                                     //UART 初始化
        ulCauses=SysCtlResetCauseGet();                 //读取复位原因
        //判断具体是哪个复位源
        if(ulCauses & SYSCTL_CAUSE_LDO)                 uartPuts("LDO power not OK reset\r\n");
        if(ulCauses & SYSCTL_CAUSE_SW)                  uartPuts("Software reset\r\n");
        if(ulCauses & SYSCTL_CAUSE_WDOG)                uartPuts("Watchdog reset\r\n");
        if(ulCauses & SYSCTL_CAUSE_BOR)                 uartPuts("Brown-out reset\r\n");
        if(ulCauses & SYSCTL_CAUSE_POR)                 uartPuts("Power on reset\r\n");
        if(ulCauses & SYSCTL_CAUSE_EXT)                 uartPuts("External reset\r\n");
                                                        uartPuts("\r\n");
        SysCtlResetCauseClear(SYSCTL_CAUSE_LDO |        //清除所有复位源
                              SYSCTL_CAUSE_SW |
                              SYSCTL_CAUSE_WDOG |
                              SYSCTL_CAUSE_BOR |
                              SYSCTL_CAUSE_POR |
                              SYSCTL_CAUSE_EXT);
        SysCtlDelay(4500 * (TheSysClock / 3000));       //延时约 4500ms
        SysCtlReset();                                  //软件复位
        for(;;)                                         //不会执行到这里
        {
        }
}
```

代码分析：SysCtlResetCauseGet()可以读出复位源，所以程序开发者利用这个函数读取复位源，进行判断，通过串口输出。

3.4 外设控制

Stellaris 系列 ARM 所有片内外设只有在使能后才可以工作，如果直接对一个尚未使能的外设进行操作，则会进入硬故障中断(Fault ISR)。

SysCtlPeripheralEnable()是使能片内外设的库函数，我们已经非常熟悉了。如果一个片内外设暂时不被使用，则可以用库函数 SysCtlPeripheralDisable()将其禁止，以节省功耗，参见表 3-14 和表 3-15 的描述。

表 3-14 函数 SysCtlPeripheralEnable()

函数名称	SysCtlPeripheralEnable()
功能	使能一个片内外设
原型	void SysCtlPeripheralEnable(unsigned long ulPeripheral)
参数	ulPeripheral：要使能的片内外设，应当取下列值之一： SYSCTL_PERIPH_PWM //PWM(脉宽调制) SYSCTL_PERIPH_ADC //ADC(模—数转换) SYSCTL_PERIPH_HIBERNATE //Hibernation module(冬眠模块) SYSCTL_PERIPH_WDOG //Watchdog(看门狗) SYSCTL_PERIPH_UART0 //UART 0(串行异步收发器 0) SYSCTL_PERIPH_UART1 //UART 1(串行异步收发器 1) SYSCTL_PERIPH_UART2 //UART 2(串行异步收发器 2) SYSCTL_PERIPH_SSI //SSI(同步串行接口) SYSCTL_PERIPH_SSI0 //SSI 0(同步串行接口 0，与 SSI 等同) SYSCTL_PERIPH_SSI1 //SSI 1(同步串行接口 1) SYSCTL_PERIPH_QEI //QEI(正交编码接口) SYSCTL_PERIPH_QEI0 //QEI 0(正交编码接口 0，与 QEI 等同) SYSCTL_PERIPH_QEI1 //QEI 1(正交编码接口 1) SYSCTL_PERIPH_I2C //I^2C(互联 I^2C 总线) SYSCTL_PERIPH_I2C0 //I^2C 0(互联 I^2C 总线 0，与 I^2C 等同) SYSCTL_PERIPH_I2C1 //I^2C 1(互联 I^2C 总线 1) SYSCTL_PERIPH_TIMER0 //Timer 0(定时器 0) SYSCTL_PERIPH_TIMER1 //Timer 1(定时器 1) SYSCTL_PERIPH_TIMER2 //Timer 2(定时器 2) SYSCTL_PERIPH_TIMER3 //Timer 3(定时器 3) SYSCTL_PERIPH_COMP0 //Analog comparator 0(模拟比较器 0) SYSCTL_PERIPH_COMP1 //Analog comparator 1(模拟比较器 1) SYSCTL_PERIPH_COMP2 //Analog comparator 2(模拟比较器 2) SYSCTL_PERIPH_GPIOA //GPIO A(通用输入输出端口 A) SYSCTL_PERIPH_GPIOB //GPIO B(通用输入输出端口 B) SYSCTL_PERIPH_GPIOC //GPIO C(通用输入输出端口 C)

续表

参数	SYSCTL_PERIPH_GPIOD　//GPIO D(通用输入输出端口 D) SYSCTL_PERIPH_GPIOE　//GPIO E(通用输入输出端口 E) SYSCTL_PERIPH_GPIOF　//GPIO F(通用输入输出端口 F) SYSCTL_PERIPH_GPIOG　//GPIO G(通用输入输出端口 G) SYSCTL_PERIPH_GPIOH　//GPIO H(通用输入输出端口 H) SYSCTL_PERIPH_CAN0　//CAN 0(控制局域网总线 0) SYSCTL_PERIPH_CAN1　//CAN 1(控制局域网总线 1) SYSCTL_PERIPH_CAN2　//CAN 2(控制局域网总线 2) SYSCTL_PERIPH_ETH　//ETH(以太网) SYSCTL_PERIPH_IEEE 1588　//IEEE 1588 SYSCTL_PERIPH_UDMA　//uDMA controller(μDMA 控制器) SYSCTL_PERIPH_USB0　//USB0 controller(USB0 控制器)
返回	无

表 3-15　函数 SysCtlPeripheralDisable()

函数名称	SysCtlPeripheralDisable()
功能	禁止一个片内外设
原型	void SysCtlPeripheralDisable(unsigned long ulPeripheral)
参数	ulPeripheral：要禁止的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

其他外设控制还包括外设复位、确认外设是否存在、睡眠与深度睡眠等，参见表 3-16～表 3-22 的描述。

表 3-16　函数 SysCtlPeripheralReset()

函数名称	SysCtlPeripheralReset()
功能	复位一个片内外设
原型	void SysCtlPeripheralReset(unsigned long ulPeripheral)
参数	ulPeripheral：要复位的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

表 3-17　函数 SysCtlPeripheralPresent()

函数名称	SysCtlPeripheralPresent()
功能	确认某个片内外设是否存在
原型	tBoolean SysCtlPeripheralPresent(unsigned long ulPeripheral)
参数	ulPeripheral：要确认的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同，并增加以下几个。 SYSCTL_PERIPH_PLL　//PLL(锁相环) SYSCTL_PERIPH_TEMP　//Temperature sensor(温度传感器) SYSCTL_PERIPH_MPU　//Cortex-M3 MPU(Cotex-M3 存储器保护单元)
返回	要确认的外设如果实际存在则返回 true，如果不存在则返回 false

表 3-18 函数 SysCtlPeripheralSleepEnable()

函数名称	SysCtlPeripheralSleepEnable()
功能	使能一个在睡眠模式下工作的片内外设
原型	void SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)
参数	ulPeripheral：要使能的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

表 3-19 函数 SysCtlPeripheralSleepDisable()

函数名称	SysCtlPeripheralSleepDisable()
功能	禁止一个在睡眠模式下工作的片内外设
原型	void SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)
参数	ulPeripheral：要禁止的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

表 3-20 函数 SysCtlPeripheralDeepSleepEnable()

函数名称	SysCtlPeripheralDeepSleepEnable()
功能	使能一个在深度睡眠模式下工作的片内外设
原型	void SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)
参数	ulPeripheral：要使能的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

表 3-21 函数 SysCtlPeripheralDeepSleepDisable()

函数名称	SysCtlPeripheralDeepSleepDisable()
功能	禁止一个在深度睡眠模式下工作的片内外设
原型	void SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)
参数	ulPeripheral：要禁止的片内外设，与表 3-14 当中参数 ulPeripheral 的取值相同
返回	无

表 3-22 函数 SysCtlPeripheralClockGating()

函数名称	SysCtlPeripheralClockGating()
功能	控制睡眠或深度睡眠模式中的外设时钟选择
原型	void SysCtlPeripheralClockGating(tBoolean bEnable)
参数	bEnable：如果在睡眠或深度睡眠下的外设被配置为该使用时取值 true 否则取值 false
返回	无

3.5 睡眠与深度睡眠

1. 睡眠与深度睡眠模式

Stellaris 系列 ARM 主要有 3 种工作模式：运行模式(Run Mode)、睡眠模式(Sleep Mode)、深度睡眠模式(Deep Sleep Mode)。有许多型号还单独具有极为省电的冬眠模块

(Hibernation Module)。而对各个模式下的外设时钟选通以及系统时钟源的控制主要由表 3-23 中的寄存器来完成。

表 3-23　运行模式控制寄存器

名　　称	描　　述
RCC、RCC2	运行模式时钟配置
RCGC0～RCGC2	运行模式时钟选通控制
SCGC0～SCGC2	睡眠模式时钟选通控制
DCGC0～DCGC2	深度睡眠模式时钟选通控制
DSLPCLKCFG	深度睡眠模式时钟配置

运行模式是正常的工作模式,处理器内核将积极地执行代码。在睡眠模式下,系统时钟不变,但处理器内核不再执行代码(内核因不需要时钟而省电)。在深度睡眠模式下,系统时钟可变,处理器内核同样也不再执行代码。深度睡眠模式比睡眠模式更为省电。有关这 3 种工作模式的具体区别请参见表 3-24 的描述。

表 3-24　运行、睡眠和深度睡眠对照表

处理器模式 / 比较项目	运行模式 (Run Mode)	睡眠模式 (Sleep Mode)	深度睡眠模式 (Deep Sleep Mode)
处理器、存储器	活动	停止(存储器内容保持不变)	停止(存储器内容保持不变)
功耗大小	大	大	很小
外设时钟源	所有时钟源都可用,包括晶振、内部 12MHz 振荡器、内部 30kHz 振荡器、PLL 以及外部 32.768kHz 有源时钟信号	由运行模式进入睡眠模式时,系统时钟的配置保持不变	在进入深度睡眠后可自动关闭功耗较高的主振荡器,改用功耗较低的内部振荡器。 若使用 PLL,则进入深度睡眠后 PLL 可以被自动断电,改用 OSC 的 16 或 64 分频作为系统时钟。处理器被唤醒后,首先恢复原先的时钟配置,再执行代码

调用函数 SysCtlSleep()可以使处理器进入睡眠模式,调用函数 SysCtlDeepSleep()可以使处理器进入深度睡眠模式,参见表 3-25 和表 3-26 的描述。

表 3-25　函数 SysCtlSleep()

函数名称	SysCtlSleep()
功能	使处理器进入睡眠模式
原型	void SysCtlSleep(void)
参数	无
返回	无(在处理器未被唤醒前不会返回)

表 3-26 函数 SysCtlDeepSleep()

函数名称	SysCtlDeepSleep()
功能	使处理器进入深度睡眠模式
原型	void SysCtlDeepSleep(void)
参数	无
返回	无(在处理器未被唤醒前不会返回)

处理器进入睡眠或深度睡眠后,就停止活动。当出现一个中断时,可以唤醒处理器,使其从睡眠或深度睡眠模式返回到正常的运行模式。因此在进入睡眠或深度睡眠之前,必须配置某个片内外设的中断并允许其在睡眠或深度睡眠模式下继续工作,如果不这样,则只有复位或重新上电才能结束睡眠/深度睡眠状态。处理器唤醒后首先执行中断服务程序,退出后接着执行主程序当中后续的代码。

函数 SysCtlSleep()是使用 WFI 汇编指令,使处理器立即进入睡眠模式,并等待中断异常发生唤醒处理器。

函数 SysCtlDeepSleep()先使能系统控制寄存器(NVICSC)的深度睡眠位,然后再使用 WFI 汇编指令,使处理器立即进入深度睡眠模式,并等待中断异常发生唤醒处理器。

2. 基础时钟源

LM3S 系列单片机的系统时钟可由下列的基础时钟源转换而来。

(1) 主振荡器(MOSC):由外部晶体振荡器或单端时钟源来驱动。

(2) 12MHz 内部振荡器(IOSC):内部振荡器是片内时钟源。它不需要使用任何外部元件便可工作。内部振荡器的频率是 12MHz±30%。

(3) 30kHz 内部振荡器:内部 30kHz 振荡器与内部 12MHz 振荡器类似,它提供 30kHz±30%的工作频率,主要用于在深度睡眠的节电模式中。

(4) 外部实时振荡器:外部实时振荡器提供一个低频率、精确的时钟基准。它的目的是给系统提供一个实时时钟源。实时振荡器是休眠模块的一部分,它也可为深度睡眠和休眠模式提供一个精确的时钟源。

3. 睡眠模式配置操作

睡眠模式下,处理器内核和存储器子系统都不使用时钟。外设仅在相应的时钟选通位使能时,才使用时钟。睡眠模式下,系统时钟源和频率均与运行模式下的相同。其配置流程如图 3-10 所示。

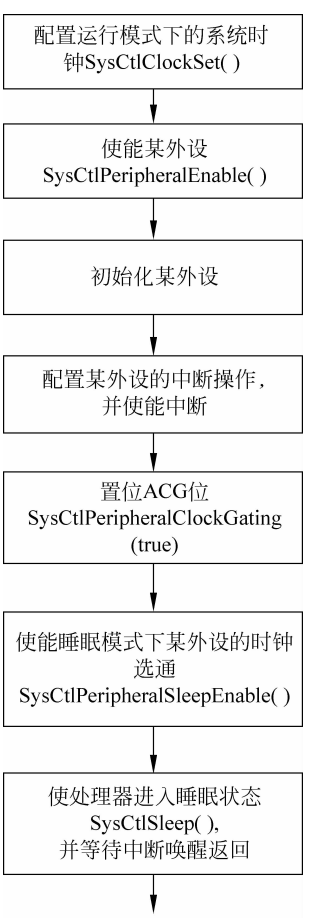

图 3-10 睡眠模式的配置流程

4. 睡眠与深度睡眠例程

程序清单 3-4 是睡眠模式的实例。程序在初始化时点亮 LED，表明处于运行模式；此后进入睡眠模式，处理器暂停运行，并以熄灭 LED 来指示；当出现 KEY 中断时，处理器被唤醒，先执行中断服务函数，退出中断后接着执行主程序当中的后续代码；按照程序的安排，唤醒后点亮 LED，延时一段时间后再次进入睡眠模式，等待 KEY 中断唤醒，如此反复。

程序清单 3-4　SysCtl 例程：睡眠省电模式

```
文件：main.c
#include "config.h"

/*SleepMode 为睡眠方式控制*/
/*0 ：深度睡眠方式*/
/*1 ：普通睡眠方式*/
#define SleepMode 0                                          //睡眠方式
#define SleepCLK 30000   //睡眠时的时钟频率——深度睡眠时本例为 30kHz,普通睡眠时本例为 6MHz
#define SleepTime (SleepCLK)                                 //睡眠时间 10s

#define LED2 GPIO_PIN_2
#define LED3 GPIO_PIN_3
#define LED4 GPIO_PIN_4

#define Start (1 << 0)
#define Write (1 << 1)
#define Read (0 << 1)

volatile unsigned long RegVal;

int main(void)
{
    /*设置系统时钟——6MHz*/
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
                   SYSCTL_XTAL_6MHZ| SYSCTL_OSC_MAIN);

    /*使能外设 GPIOA*/
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ETH);

    /*配置引脚驱动*/
    GPIOPadConfigSet(GPIO_PORTA_BASE,LED2|LED3|LED4,
                     GPIO_STRENGTH_2MA,GPIO_PIN_TYPE_STD_WPU);
    GPIODirModeSet(GPIO_PORTA_BASE,LED2|LED3|LED4,
                   GPIO_DIR_MODE_OUT);

    HWREG(0x40048020)=Start | Read | (1<<3);     //MR1
    RegVal=HWREG(0x40048030);
    HWREG(0x40048020)=Start | Read | (0<<3);     //MR0
    RegVal=HWREG(0x40048030);

    HWREG(0x40048020)=Start | Write | (0<<3);    //写 MR0
```

```
    HWREG(0x4004802C)=RegVal | (1<<11);

    /*测试引脚功能*/
    GPIOPinWrite(GPIO_PORTA_BASE,LED2|LED3|LED4,~(LED2|LED3|LED4));
                                                                    //亮 LED
    TimeDelay(500000);
    GPIOPinWrite(GPIO_PORTA_BASE,LED2|LED3|LED4,LED2|LED3|LED4);    //灭 LED

    SysCtlPeripheralDisable(SYSCTL_PERIPH_ETH);
    //SysCtlPeripheralDisable(SYSCTL_PERIPH_GPIOA);

    /*设置 SysTick 的加载值*/
    SysTickPeriodSet(SleepTime);

    /*使能 SysTick 的中断*/
    SysTickIntEnable();

    /*使能总中断*/
    IntMasterEnable();

    /*启动 SysTick*/


    /*使用 DCGCn 寄存器或 SCGCn 寄存器进行控制睡眠时的外设使能,需置位 RCC 寄存器的
ACG 位,以开启自动时钟门控*/
    SysCtlPeripheralClockGating(true);

    /*选择深度睡眠时的系统时钟源——本例选择 30kHz 内部时钟振荡器*/
    HWREG(SYSCTL_DSLPCLKCFG)=(3 << 4);


#if SleepMode==0

    /*在深度睡眠的情况下使能外设 TIMER0,使得 TIMER0 在系统时钟的情况下继续运行*/
    HWREG(SYSCTL_DCGC0)=0x00000000;
    HWREG(SYSCTL_DCGC1)=0x00000000;
    HWREG(SYSCTL_DCGC2)=0x00000000;
    //SysCtlPeripheralDeepSleepEnable(SYSCTL_PERIPH_GPIOA);
    //SysCtlPeripheralDeepSleepEnable(SYSCTL_PERIPH_TIMER0);
    /*使 CM3 进入深度睡眠模式——内核停止运行,程序停止运行,必须由中断唤醒内核*/
    SysTickEnable();
    SysCtlDeepSleep();
#endif

#if SleepMode==1

    HWREG(SYSCTL_SCGC0)=0x00000000;
    HWREG(SYSCTL_SCGC1)=0x00000000;
    HWREG(SYSCTL_SCGC2)=0x00000000;
    SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_TIMER0);
```

```
    /* 在普通睡眠的情况下使能外设 TIMER0,使得 TIMER0 在系统时钟的情况下继续运行 */

    /* 使 CM3 进入普通睡眠模式——内核停止运行,程序停止运行,必须由中断唤醒内核 */
    SysCtlSleep();
#endif
    /* 中断唤醒后,继续运行程序 */
    while(1)
    {
      TimeDelay(500000);
      GPIOPinWrite(GPIO_PORTA_BASE,LED3 | LED4,~(LED3 | LED4));
      TimeDelay(500000);
      GPIOPinWrite(GPIO_PORTA_BASE,LED3 | LED4,(LED3 | LED4));
    }
}

void SysTick_ISR(void)
{
  /* LED 取反输出 */
  GPIOPinWrite(GPIO_PORTA_BASE,LED2,~GPIOPinRead(GPIO_PORTA_BASE,LED2));
}
```

程序清单 3-5 是深度睡眠模式的实例。为了便于演示深度睡眠模式下系统时钟的变化,在例程中增加了蜂鸣器的驱动函数 sound()。在这里采用的是交流蜂鸣器,也称无源蜂鸣器,发声频率等于驱动它的方波频率。产生方波的方法是利用 Timer 的 16 位 PWM 功能。有关 Timer 模块的用法将在后续章节里详细讨论。

在程序清单 3-5 里,初始化时 Timer 模块的时钟(等同于系统时钟)设置为 PLL 输出 12.5MHz(请修改 systemInit.c 里的 clockInit()函数),蜂鸣器发声频率为 2500Hz,表现为尖叫。在进入深度睡眠模式后,PLL 被自动禁止,Timer 模块的时钟改由 IOSC 的 16 分频来提供,此时蜂鸣器的发声频率变成约 150Hz,表现为低沉的叫声。按下 KEY 以后,处理器会被唤醒,Timer 模块的时钟恢复为原来的配置,于是蜂鸣器重新尖叫。

程序清单 3-5　SysCtl 例程:深度睡眠省电模式

文件:main.c

```
#include "config.h"

/* SleepMode 为睡眠方式控制 */
/* 0 : 深度睡眠方式 */
/* 1 : 普通睡眠方式 */
#define SleepMode 0                                          //睡眠方式
#define SleepCLK 30000 //睡眠时的时钟频率——深度睡眠时本例为 30kHz,普通睡眠时本例为 6MHz
#define SleepTime (SleepCLK)                                 //睡眠时间 10s

#define LED2 GPIO_PIN_2
#define LED3 GPIO_PIN_3
#define LED4 GPIO_PIN_4

#define Start (1 << 0)
```

```
#define Write (1 << 1)
#define Read (0 << 1)

volatile unsigned long RegVal;

int main(void)
{
    /* 设置系统时钟——6MHz */
    SysCtlClockSet(SYSCTL_SYSDIV_1 | SYSCTL_USE_OSC |
                   SYSCTL_XTAL_6MHZ| SYSCTL_OSC_MAIN);

    /* 使能外设 GPIOA */
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ETH);

    /* 配置引脚驱动 */
    GPIOPadConfigSet(GPIO_PORTA_BASE,LED2|LED3|LED4,
                     GPIO_STRENGTH_2MA,GPIO_PIN_TYPE_STD_WPU);
    GPIODirModeSet(GPIO_PORTA_BASE,LED2|LED3|LED4,
                   GPIO_DIR_MODE_OUT);


    HWREG(0x40048020)=Start | Read | (1<<3);     //MR1
    RegVal=HWREG(0x40048030);

    HWREG(0x40048020)=Start | Read | (0<<3);     //MR0
    RegVal=HWREG(0x40048030);


    HWREG(0x40048020)=Start | Write | (0<<3);    //写 MR0
    HWREG(0x4004802C)=RegVal | (1<<11);

    /* 测试引脚功能 */
    GPIOPinWrite(GPIO_PORTA_BASE,LED2|LED3|LED4,~(LED2|LED3|LED4));//亮 LED
    TimeDelay(500000);
    GPIOPinWrite(GPIO_PORTA_BASE,LED2|LED3|LED4,LED2|LED3|LED4);      //灭 LED

    SysCtlPeripheralDisable(SYSCTL_PERIPH_ETH);
    //SysCtlPeripheralDisable(SYSCTL_PERIPH_GPIOA);

    /* 设置 SysTick 的加载值 */
    SysTickPeriodSet(SleepTime);

    /* 使能 SysTick 的中断 */
    SysTickIntEnable();

    /* 使能总中断 */
```

```
    IntMasterEnable();

    /* 启动 SysTick */


    /* 使用 DCGCn 寄存器或 SCGCn 寄存器进行控制睡眠时的外设使能,需置位 RCC 寄存器的
ACG 位,以开启自动时钟门控 */
    SysCtlPeripheralClockGating(true);

    /* 选择深度睡眠时的系统时钟源——本例选择 30kHz 内部时钟振荡器 */
    HWREG(SYSCTL_DSLPCLKCFG)=(3 << 4);


#if SleepMode==0

    /* 在深度睡眠的情况下使能外设 TIMER0,使得 TIMER0 在系统时钟的情况下继续运行 */
    HWREG(SYSCTL_DCGC0)=0x00000000;
    HWREG(SYSCTL_DCGC1)=0x00000000;
    HWREG(SYSCTL_DCGC2)=0x00000000;
    //SysCtlPeripheralDeepSleepEnable(SYSCTL_PERIPH_GPIOA);
    //SysCtlPeripheralDeepSleepEnable(SYSCTL_PERIPH_TIMER0);
    /* 使 CM3 进入深度睡眠模式——内核停止运行,程序停止运行,必须由中断唤醒内核 */
    SysTickEnable();
    SysCtlDeepSleep();
#endif

#if SleepMode==1

    HWREG(SYSCTL_SCGC0)=0x00000000;
    HWREG(SYSCTL_SCGC1)=0x00000000;
    HWREG(SYSCTL_SCGC2)=0x00000000;
    SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_GPIOA);
    SysCtlPeripheralSleepEnable(SYSCTL_PERIPH_TIMER0);
    /* 在普通睡眠的情况下使能外设 TIMER0,使得 TIMER0 在系统时钟的情况下继续运行 */

    /* 使 CM3 进入普通睡眠模式——内核停止运行,程序停止运行,必须由中断唤醒内核 */
    SysCtlSleep();
#endif
    /* 中断唤醒后,继续运行程序 */
    while(1)
    {
      TimeDelay(500000);
      GPIOPinWrite(GPIO_PORTA_BASE,LED3 | LED4,~(LED3 | LED4));
      TimeDelay(500000);
      GPIOPinWrite(GPIO_PORTA_BASE,LED3 | LED4,(LED3 | LED4));
    }
}

void SysTick_ISR(void)
{
```

```
    /* LED 取反输出 */
    GPIOPinWrite(GPIO_PORTA_BASE, LED2, ~GPIOPinRead(GPIO_PORTA_BASE, LED2));
}
```

3.6 杂项功能

这是一组杂项功能的函数，包括延时、存储器大小、特定引脚是否存在以及高速 GPIO 等。函数 SysCtlDelay()提供一个产生固定长度延时的方法，参见表 3-27 的描述。它是用内嵌汇编语言的方式来编写的，可以在使用不同软件开发工具情况下而让程序的延时保持一致，具有较好的可移植性。以下是实现 SysCtlDelay()函数的汇编源代码，每个循环花费 3 个系统时钟周期：

```
SysCtlDelay
    SUBS R0, #1          ;R0 减 1,R0 实际上就是参数 ulCount
    BNE SysCtlDelay      ;如果结果不为 0 则跳转到 SysCtlDelay
    BX LR                ;子程序返回
```

表 3-27 函数 SysCtlDelay()

函数名称	SysCtlDelay()
功能	延时
原型	void SysCtlDelay(unsigned long ulCount)
参数	ulCount：延时周期计数值，延时长度＝3×ulCount×系统时钟周期
返回	无
备注	SysCtlDelay(20)；　　//延时 60 个系统时钟周期 SysCtlDelay(150 * (SysCtlClockGet() / 3000))；　　//延时 150ms

函数 SysCtlFlashSizeGet()和函数 SysCtlSRAMSizeGet()用来获取当前芯片的 Flash 和 SRAM 存储器大小，返回的单位是 byte(字节)，参见表 3-28 和表 3-29 的描述。

表 3-28 函数 SysCtlFlashSizeGet()

函数名称	SysCtlFlashSizeGet()
功能	获取片内 Flash 的大小
原型	unsigned long SysCtlFlashSizeGet(void)
参数	无
返回	Flash 的大小(单位：字节)

表 3-29 函数 SysCtlSRAMSizeGet()

函数名称	SysCtlSRAMSizeGet()
功能	获取片内 SRAM 的大小
原型	unsigned long SysCtlSRAMSizeGet(void)
参数	无
返回	SRAM 的大小(单位：字节)

函数 SysCtlPinPresent()用来确认非 GPIO 片内外设的特定功能引脚是否存在，参见表 3-30 的描述。

表 3-30　函数 SysCtlPinPresent()

函数名称	SysCtlPinPresent()
功能	确认非 GPIO 片内外设的特定功能引脚是否存在
原型	tBoolean SysCtlPinPresent(unsigned long ulPin)
参数	ulPin：待断定的引脚，应当取下列值之一： SYSCTL_PIN_PWM0　　//PWM0　引脚 SYSCTL_PIN_PWM1　　//PWM1　引脚 SYSCTL_PIN_PWM2　　//PWM2　引脚 SYSCTL_PIN_PWM3　　//PWM3　引脚 SYSCTL_PIN_PWM4　　//PWM4　引脚 SYSCTL_PIN_PWM5　　//PWM5　引脚 SYSCTL_PIN_PWM6　　//PWM6　引脚 SYSCTL_PIN_PWM7　　//PWM7　引脚 SYSCTL_PIN_C0MINUS　　//C0－　引脚 SYSCTL_PIN_C0PLUS　　//C0＋　引脚 SYSCTL_PIN_C0O　　//C0O　引脚 SYSCTL_PIN_C1MINUS　　//C1－　引脚 SYSCTL_PIN_C1PLUS　　//C1＋　引脚 SYSCTL_PIN_C1O　　//C1O　引脚 SYSCTL_PIN_C2MINUS　　//C2－　引脚 SYSCTL_PIN_C2PLUS　　//C2＋　引脚 SYSCTL_PIN_C2O　　//C2O　引脚 SYSCTL_PIN_MC_FAULT0　　//MC0 Fault　引脚 SYSCTL_PIN_ADC0　　//ADC0　引脚 SYSCTL_PIN_ADC1　　//ADC1　引脚 SYSCTL_PIN_ADC2　　//ADC2　引脚 SYSCTL_PIN_ADC3　　//ADC3　引脚 SYSCTL_PIN_ADC4　　//ADC4　引脚 SYSCTL_PIN_ADC5　　//ADC5　引脚 SYSCTL_PIN_ADC6　　//ADC6　引脚 SYSCTL_PIN_ADC7　　//ADC7　引脚 SYSCTL_PIN_CCP0　　//CCP0　引脚 SYSCTL_PIN_CCP1　　//CCP1　引脚 SYSCTL_PIN_CCP2　　//CCP2　引脚 SYSCTL_PIN_CCP3　　//CCP3　引脚 SYSCTL_PIN_CCP4　　//CCP4　引脚 SYSCTL_PIN_CCP5　　//CCP5　引脚 SYSCTL_PIN_32KHZ　　//32kHz　引脚
返回	如果要确认的外设引脚存在则返回 true，否则返回 false

函数 SysCtlGPIOAHBEnable()和 SysCtlGPIOAHBDisable()用来管理 GPIO 高速访问总线的使用，参见表 3-31 和表 3-32 的描述。

在2008年新推出的DustDevil家族(LM3S3×××/5×××系列,以及部分LM3S1×××/2×××型号)里,新增了一项GPIO高速总线访问(GPIO peripheral for Access from the High speed Bus,AHB)的功能。在原来的型号里,访问一次外设需要花费2个系统时钟,在50MHz的主频下采用执行汇编指令的方法不断翻转GPIO,获得的方波频率最高为50MHz÷4=12.5MHz。而通过AHB,访问一次GPIO仅需花费1个系统时钟周期,此时通过汇编指令翻转GPIO获得的方波频率最高可达25MHz。

复位时GPIO高速总线访问功能是禁止的,可以通过调用函数SysCtlGPIOAHBEnable()来使能。原来操作GPIO时,在相关函数里采用的GPIO端口基址是GPIO_PORTA_BASE和GPIO_PORTB_BASE等,在使能AHB功能后要相应地换成GPIO_PORTA_AHB_BASE和GPIO_PORTB_AHB_BASE等,才能正确地使用AHB功能。

表3-31 函数SysCtlGPIOAHBEnable()

函数名称	SysCtlGPIOAHBEnable()
功能	使能GPIO模块通过高速总线来访问
原型	void SysCtlGPIOAHBEnable(unsigned long ulGPIOPeripheral)
参数	ulGPIOPeripheral:要使能的GPIO模块,应当取下列值之一: SYSCTL_PERIPH_GPIOA //GPIO A(通用输入输出端口A) SYSCTL_PERIPH_GPIOB //GPIO B(通用输入输出端口B) SYSCTL_PERIPH_GPIOC //GPIO C(通用输入输出端口C) SYSCTL_PERIPH_GPIOD //GPIO D(通用输入输出端口D) SYSCTL_PERIPH_GPIOE //GPIO E(通用输入输出端口E) SYSCTL_PERIPH_GPIOF //GPIO F(通用输入输出端口F) SYSCTL_PERIPH_GPIOG //GPIO G(通用输入输出端口G) SYSCTL_PERIPH_GPIOH //GPIO H(通用输入输出端口H)
返回	无

表3-32 函数SysCtlGPIOAHBDisable()

函数名称	SysCtlGPIOAHBDisable()
功能	禁止GPIO模块通过高速总线来访问
原型	void SysCtlGPIOAHBDisable(unsigned long ulGPIOPeripheral)
参数	ulGPIOPeripheral:要禁止的GPIO模块
返回	无

3.7 中断控制

3.7.1 中断基本概念

中断(Interrupt)是MCU实时地处理内部或外部事件的一种机制。当某种内部或外部事件发生时,MCU的中断系统将迫使CPU暂停正在执行的程序,转而去进行中断事件的处理,中断处理完毕后,又返回被中断的程序处,继续执行下去。

图 3-11 给出了中断过程的示意图。主程序正在执行，当遇到中断请求(Interrupt Request)时，暂停主程序的执行转而去执行中断服务例程(Interrupt Service Routine，ISR)，称为响应，中断服务例程执行完毕后返回到主程序断点处并继续执行主程序。

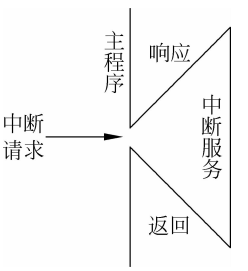

图 3-11　中断过程示意图

3.7.2　Stellaris 中断基本编程方法

利用 Stellaris 外设驱动库编写一个中断程序的基本方法流程如下。

1. 使能相关片内外设，并进行基本的配置

对于中断源所涉及的片内外设首先必须要使能，使能的方法是调用头文件<sysctl.h>中的函数 SysCtlPeripheralEnable()。使能该片内外设以后，还要进行必要的基本配置。

2. 设置具体中断的类型或触发方式

不同片内外设具体中断的类型或触发方式也各不相同。在使能中断之前，必须对其进行正确的设置。以 GPIO 为例，分为边沿触发、电平触发两大类，为此需要通过调用函数 GPIOIntTypeSet()来进行设置。

3. 使能中断

对于 Stellaris 系列 ARM，使能一个片内外设的具体中断，通常要采取 3 步走的方法：

(1) 调用片内外设具体中断的使能函数；

(2) 调用函数 IntEnable()，使能片内外设的总中断；

(3) 调用函数 IntMasterEnable()，使能处理器总中断。

4. 编写中断服务函数

C 语言是函数式语言，ISR 可以称为“中断服务函数”。中断服务函数从形式上跟普通函数类似，但在命名及具体的处理上有所不同。

中断服务函数命名　在 Keil 或 IAR 开发环境下，中断服务函数的名称可以由程序员自行指定，但是为了提高程序的可移植性，建议采用标准的中断服务函数名称，参见表 3-33。例如，GPIOB 端口的中断服务函数名称是 GPIO_Port_B_ISR，对应的函数头应当是 void GPIO_Port_B_ISR(void)。

注意：参数和返回值都必须是 void 类型。

表 3-33　中断服务函数标准名称

向量号	中断服务函数名	向量号	中断服务函数名	向量号	中断服务函数名
0	(堆栈初值)	22	UART1_ISR	44	System_Control_ISR
1	reset_handler	23	SSI_ISR 或 SSI0_ISR	45	FLASH_Control_ISR
2	Nmi_ISR	24	I2C_ISR 或 I2C0_ISR	46	GPIO_Port_F_ISR
3	Fault_ISR	25	PWM_Fault_ISR	47	GPIO_Port_G_ISR
4	(MPU)	26	PWM_Generator_0_ISR	48	GPIO_Port_H_ISR
5	(Bus fault)	27	PWM_Generator_1_ISR	49	UART2_ISR
6	(Usage fault)	28	PWM_Generator_2_ISR	50	SSI1_ISR
7	(Reserved)	29	QEI_ISR 或 QEI0_ISR	51	Timer3A_ISR

续表

向量号	中断服务函数名	向量号	中断服务函数名	向量号	中断服务函数名
8	(Reserved)	30	ADC_Sequence_0_ISR	52	Timer3B_ISR
9	(Reserved)	31	ADC_Sequence_1_ISR	53	I2C1_ISR
10	(Reserved)	32	ADC_Sequence_2_ISR	54	QEI1_ISR
11	SVCall_ISR	33	ADC_Sequence_3_ISR	55	CAN0_ISR
12	(Debug monitor)	34	Watchdog_Timer_ISR	56	CAN1_ISR
13	(Reserved)	35	Timer0A_ISR	57	CAN2_ISR
14	PendSV_ISR	36	Timer0B_ISR	58	ETHERNET_ISR
15	SysTick_ISR	37	Timer1A_ISR	59	HIBERNATE_ISR
16	GPIO_Port_A_ISR	38	Timer1B_ISR	60	USB0_ISR
17	GPIO_Port_B_ISR	39	Timer2A_ISR	61	PWM_Generator_3_ISR
18	GPIO_Port_C_ISR	40	Timer2B_ISR	62	uDMA_ISR
19	GPIO_Port_D_ISR	41	Analog_Comparator_0_ISR	63	uDMA_Error_ISR
20	GPIO_Port_E_ISR	42	Analog_Comparator_1_ISR		
21	UART0_ISR	43	Analog_Comparator_2_ISR		

中断状态查询 一个具体的片内外设可能存在多个子中断源，但是都共用同一个中断向量。例如 GPIOA 有 8 个引脚，每个引脚都可以产生中断，但是都共用同一个中断向量号 16，任一引脚发生中断时都会进入同一个中断服务函数。为了能够准确区分每一个子中断源，就需要利用中断状态查询函数，例如 GPIO 的中断状态查询函数是 GPIOPinIntStatus()。如果不使能中断，而采取纯粹的“轮询”编程方式，也是利用中断状态查询函数来确定是否发生了中断以及具体是哪个子中断源产生的中断。

中断清除 对于 Stellaris 系列 ARM 的所有片内外设，在进入其中断服务函数后，中断状态并不能自动清除，而必须采用软件清除(但是属于 Cortex-M3 内核的中断源例外，因为它们不属于“外设”)。如果中断未被及时清除，则在退出中断服务函数时会立即再次触发中断而造成混乱。清除中断的方法是调用相应片内外设的中断清除函数。例如 GPIO 端口的中断清除函数是 GPIOPinIntClear()。

程序清单 3-6 以 GPIOA 中断为例，给出了外设中断服务函数的经典编写方法。关键是先将外设的中断状态读到变量 ulStatus 里，然后及时地清除全部中断状态，剩下的工作就是排列多个 if 语句分别进行处理了。

程序清单 3-6 典型的中断服务函数编写方法

```
//GPIOA 的中断服务函数
void GPIO_Port_A_ISR(void)
{
  unsigned long ulStatus;

  ulStatus=GPIOPinIntStatus(GPIO_PORTA_BASE,true);      //读取中断状态
GPIOPinIntClear(GPIO_PORTA_BASE,ulStatus);              //清除中断状态,重要

if (ulStatus & GPIO_PIN_0)                              //如果 PA0 的中断状态有效
{
```

```
    //在这里添加 PA0 的中断处理代码
}
if (ulStatus & GPIO_PIN_1)                          //如果 PA1 的中断状态有效
{
    //在这里添加 PA1 的中断处理代码
}
//如果还有其他引脚的中断需要处理,请继续并列类似的 if 语句
}
```

5. 注册中断服务函数

现在,中断服务函数虽然已经编写完成,但是当中断事件产生时程序还无法找到它,因为还缺少最后一个步骤——注册中断服务函数。注册方法有两种:一是直接利用中断注册函数,好处是操作简单、可移植性好,缺点是由于把中断向量表重新映射到 SRAM 中而导致执行效率下降;另一种方法需要修改启动文件,好处是执行效率很高,缺点是可移植性不够好。经过权衡考虑后,我们还是推荐大家采用后一种方法,因为效率优先、操作也并不复杂。在不同的软件开发环境下,通过修改启动文件注册中断服务函数的方法也各不相同。

1) Keil 开发环境下的操作

在 Keil 开发环境下,启动文件 Startup.s 是用汇编写的,以中断服务函数 void I2C_ISR(void)为例,找到 Vectors 表格,在其前面插入声明:

```
EXTERN  I2C_ISR
```

再根据 Vectors 表格的注释内容找到外设 I2C0 的位置,把相应的 IntDefaultHandler 替换为 I2C_ISR,完成。

2) IAR 开发环境下的操作

在 IAR 开发环境下,启动文件 startup_ewarm.c 是用 C 语言写的,很好理解。仍以中断服务函数 void I2C_ISR(void)为例,先在向量表的前面插入函数声明:

```
void I2C_ISR(void);
```

然后在向量表里,根据注释内容找到外设 I2C0 的位置,把相应的 IntDefaultHandler 替换为 I2C_ISR,完成。

在上述几个步骤完成后,就可以等待中断事件的到来了。当中断事件产生时,程序就会自动跳转到对应的中断服务函数去处理。

3.7.3　中断库函数

1. 中断使能与禁止

调用库函数 IntMasterEnable()将使能 ARM Cortex-M3 处理器内核的总中断,调用库函数 IntMasterDisable()将禁止 ARM Cortex-M3 处理器内核响应所有中断。例外情况是复位中断(Reset ISR)、不可屏蔽中断(NMI ISR)和硬故障中断(Fault ISR),它们可能随时发生而无法通过软件禁止,参见表 3-34 和表 3-35 的描述。

库函数 IntEnable()和库函数 IntDisable()是对某个片内功能模块的中断进行总体上的使能控制。中断分为两大类:一类是属于 ARM Cortex-M3 内核的,如 NMI 和 SysTick 等,中断向量号在 15 以内;另一类是 Stellaris 系列 ARM 特有的,如 GPIO、UART 和 PWM

等,中断向量号在 16 以上,参见表 3-36、表 3-37 和表 3-38 的描述。

表 3-34 函数 IntMasterEnable()

函数名称	IntMasterEnable()
功能	使能处理器中断
原型	tBoolean IntMasterEnable(void)
参数	无
返回	如果在调用该函数之前处理器中断是使能的,则返回 false 如果在调用该函数之前处理器中断是禁止的,则返回 true
备注	对复位 Reset、不可屏蔽中断 NMI、硬故障 Fault 无效

表 3-35 函数 IntMasterDisable()

函数名称	IntMasterDisable()
功能	禁止处理器中断
原型	tBoolean IntMasterDisable(void)
参数	无
返回	如果在调用该函数之前处理器中断是使能的,则返回 false 如果在调用该函数之前处理器中断是禁止的,则返回 true

表 3-36 函数 IntEnable()

函数名称	IntEnable()
功能	使能一个片内外设的中断
原型	void IntEnable(unsigned long ulInterrupt)
参数	ulInterrupt:指定被使能的片内外设中断
返回	无

表 3-37 函数 IntDisable()

函数名称	IntDisable()
功能	禁止一个片内外设的中断
原型	void IntDisable(unsigned long ulInterrupt)
参数	ulInterrupt:指定被使能的片内外设中断
返回	无

表 3-38 Stellaris 系列 ARM 的中断源

中断名称	中断向量号	功能描述
FAULT_NMI	2	NMI fault(不可屏蔽中断故障)
FAULT_HARD	3	Hard fault(硬故障)
FAULT_MPU	4	MPU fault(存储器保护单元故障)
FAULT_BUS	5	Bus fault(总线故障)
FAULT_USAGE	6	Usage fault(使用故障)
FAULT_SVCALL	11	SVCall(软件中断)
FAULT_DEBUG	12	Debug monitor(调试监控)

续表

中 断 名 称	中断向量号	功 能 描 述
FAULT_PENDSV	14	PendSV(系统服务请求)
FAULT_SYSTICK	15	System Tick(系统节拍定时器)
INT_GPIOA	16	GPIO Port A(GPIO 端口 A)
INT_GPIOB	17	GPIO Port B(GPIO 端口 B)
INT_GPIOC	18	GPIO Port C(GPIO 端口 C)
INT_GPIOD	19	GPIO Port D(GPIO 端口 D)
INT_GPIOE	20	GPIO Port E(GPIO 端口 E)
INT_UART0	21	UART0 Rx and Tx(UART0 收发)
INT_UART1	22	UART1 Rx and Tx(UART1 收发)
INT_SSI	23	SSI0 Rx and Tx(SSI0 收发,与 INT_SSI 相同)
INT_SSI0	23	SSI0 Rx and Tx(SSI0 收发,与 INT_SSI 相同)
INT_I2C	24	I2C Master and Slave(I2C 主从)
INT_I2C0	24	I2C0 Master and Slave(I2C0 主从,与 INT_I2C 相同)
INT_PWM_FAULT	25	PWM Fault(PWM 故障)
INT_PWM0	26	PWM Generator 0(PWM 发生器 0)
INT_PWM1	27	PWM Generator 1(PWM 发生器 1)
INT_PWM2	28	PWM Generator 2(PWM 发生器 2)
INT_QEI	29	Quadrature Encoder(正交编码器)
INT_QEI0	29	Quadrature Encoder 0(正交编码器 0,与 INT_QEI 相同)
INT_ADC0	30	ADC Sequence 0(ADC 采样序列 0)
INT_ADC1	31	ADC Sequence 1(ADC 采样序列 1)
INT_ADC2	32	ADC Sequence 2(ADC 采样序列 2)
INT_ADC3	33	ADC Sequence 3(ADC 采样序列 3)
INT_WATCHDOG	34	Watchdog timer(看门狗定时器)
INT_TIMER0A	35	Timer 0 subtimer A(定时器 0 子定时器 A)
INT_TIMER0B	36	Timer 0 subtimer B(定时器 0 子定时器 B)
INT_TIMER1A	37	Timer 1 subtimer B(定时器 1 子定时器 B)
INT_TIMER1B	38	Timer 1 subtimer B(定时器 1 子定时器 B)
INT_TIMER2A	39	Timer 2 subtimer A(定时器 2 子定时器 A)
INT_TIMER2B	40	Timer 2 subtimer B(定时器 2 子定时器 B)
INT_COMP0	41	Analog Comparator 0(模拟比较器 0)
INT_COMP1	42	Analog Comparator 1(模拟比较器 1)
INT_COMP2	43	Analog Comparator 2(模拟比较器 2)
INT_SYSCTL	44	System Control(PLL,OSC,BO)(系统控制,PLL、OSC、BO)
INT_FLASH	45	Flash Control(闪存控制)
INT_GPIOF	46	Flash Control(闪存控制)
INT_GPIOG	47	GPIO Port G(GPIO 端口 G)
INT_GPIOH	48	GPIO Port H(GPIO 端口 H)
INT_UART2	49	UART2 Rx and Tx(UART2 收发)
INT_SSI1	50	SSI1 Rx and Tx(SSI1 收发)
INT_TIMER3A	51	Timer 3 subtimer A(定时器 3 子定时器 A)
INT_TIMER3B	52	Timer 3 subtimer B(定时器 3 子定时器 B)

续表

中 断 名 称	中断向量号	功 能 描 述
INT_I2C1	53	I2C1 Master and Slave(I2C1 主从)
INT_QEI1	54	Quadrature Encoder 1(正交编码器 1)
INT_CAN0	55	CAN0(CAN 总线 0)
INT_CAN1	56	CAN1(CAN 总线 1)
INT_CAN2	57	CAN2(CAN 总线 2)
INT_ETH	58	Ethernet(以太网)
INT_HIBERNATE	59	Hibernation module(冬眠模块)
INT_USB0	60	USB 0 Controller(USB0 控制器)
INT_PWM3	61	PWM Generator 3(PWM 发生器 3)
INT_UDMA	62	uDMA controller(μDMA 控制器)
INT_UDMAERR	63	uDMA Error(μDMA 错误)

2. 中断优先级

ARM Cortex-M3 处理器内核可以配置的中断优先级最多可以有 256 级。虽然 Stellaris 系列 ARM 只实现了 8 个中断优先级,但对于一个实际的应用来说已经足够了。在较为复杂的控制系统中,中断优先级的设置会显得非常重要。

函数 IntPrioritySet()和函数 IntPriorityGet()用来管理一个片内外设的优先级,参见表 3-39 和表 3-40 的描述。当多个中断源同时产生时,优先级最高的中断首先被处理器响应并得到处理。正在处理较低优先级中断时,如果有较高优先级的中断产生,则处理器立即转去处理较高优先级的中断。正在处理的中断不能被同级或较低优先级的中断所打断。

表 3-39 函数 IntPrioritySet()

函数名称	IntPrioritySet()
功能	设置一个中断的优先级
原型	void IntPrioritySet(unsigned long ulInterrupt,unsigned char ucPriority)
参数	ulInterrupt:指定的中断源 ucPriority:要设定的优先级,应当取值 0～7,数值越小优先级越高
返回	无

表 3-40 函数 IntPriorityGet()

函数名称	IntPriorityGet()
功能	获取一个中断的优先级
原型	long IntPriorityGet(unsigned long ulInterrupt)
参数	ulInterrupt:指定的中断源
返回	返回中断优先级数值,该返回值除以 32(即右移 5 位)后才能得到优先级数 0～7。 如果指定了一个无效的中断,则返回－1

函数 IntPriorityGroupingSet()和函数 IntPriorityGroupingGet()用来管理抢占式优先级和子优先级的分组设置,参见表 3-41 和表 3-42 的描述。

表 3-41　函数 IntPriorityGroupingSet()

函数名称	IntPriorityGroupingSet()
功能	设置中断控制器的优先级分组
原型	void IntPriorityGroupingSet(unsigned long ulBits)
参数	ulBits：指定抢占式优先级位的数目，取值 0～7，但对 Stellaris 系列 ARM 取值 3～7 效果等同
返回	无

表 3-42　函数 IntPriorityGroupingGet()

函数名称	IntPriorityGroupingGet()
功能	获取中断控制器的优先级分组
原型	unsigned long IntPriorityGroupingGet(void)
参数	无
返回	抢占式优先级位的数目，范围 0～7，但对 Stellaris 系列 ARM 返回值 3～7 效果等同

重要规则：多个中断源在它们的抢占式优先级相同的情况下，子优先级不论是否相同，如果某个中断已经在服务当中，则其他中断源都不能打断它(可以末尾连锁)；只有抢占式优先级高的中断才可以打断其他抢占式优先级低的中断。

由于 Stellaris 系列 ARM 只实现了 3 个优先级位，因此实际有效的抢占式优先级位数只能设为 0～3 位。如果抢占式优先级位数为 3，则子优先级都是 0，实际上可嵌套的中断层数是 8 层；如果抢占式优先级位数为 2，则子优先级为 0 级和 1 级，实际可嵌套的层数为 4 层；依次类推，当抢占式优先级位数为 0 时，实际可嵌套的层数为 1 层，即不允许中断嵌套。

3.7.4　GPIO 中断控制例程

程序清单 3-7 是 GPIO 中断的例子。在程序中，用按键 KEY 作为外部中断输入，先使能 KEY 所在的 GPIO 端口并把相应的引脚设置为输入，然后配置中断触发类型并使能中断。

程序清单 3-7　GPIO 中断

```
文件：main.c
#include "systemInit.h"
//定义 LED
#define LED_PERIPH          SYSCTL_PERIPH_GPIOF
#define LED_PORT            GPIO_PORTF_BASE
#define LED_PIN             GPIO_PIN_2

//定义 KEY
#define KEY_PERIPH          SYSCTL_PERIPH_GPIOD
#define KEY_PORT            GPIO_PORTD_BASE
#define KEY_PIN             GPIO_PIN_7

//主函数(程序入口)
int main(void)
```

```
{
    clockInit();                                                    //时钟初始化：晶振,6MHz
    SysCtlPeriEnable(LED_PERIPH);                                   //使能 LED 所在的 GPIO 端口
    GPIOPinTypeOut(LED_PORT,LED_PIN);                               //设置 LED 所在引脚为输出
    SysCtlPeriEnable(KEY_PERIPH);                                   //使能 KEY 所在的 GPIO 端口
    GPIOPinTypeIn(KEY_PORT,KEY_PIN);                                //设置 KEY 所在引脚为输入
    GPIOIntTypeSet(KEY_PORT,KEY_PIN,GPIO_LOW_LEVEL);//设置 KEY 引脚的中断类型
    GPIOPinIntEnable(KEY_PORT,KEY_PIN);                             //使能 KEY 所在引脚的中断
    IntEnable(INT_GPIOD);                                           //使能 GPIOD 端口中断
    IntMasterEnable();                                              //使能处理器中断

    for (;;)                                                        //等待 KEY 中断
    {
    }
}

//GPIOD 的中断服务函数
void GPIO_Port_D_ISR(void)
{
    unsigned char ucVal;
    unsigned long ulStatus;
    ulStatus=GPIOPinIntStatus(KEY_PORT,true);                       //读取中断状态
    GPIOPinIntClear(KEY_PORT,ulStatus);                             //清除中断状态,重要
    if (ulStatus & KEY_PIN)                                         //如果 KEY 的中断状态有效
    {
        ucVal=GPIOPinRead(LED_PORT,LED_PIN);                        //翻转 LED
        GPIOPinWrite(LED_PORT,LED_PIN,~ucVal);
        SysCtlDelay(10*(TheSysClock/3000));                         //延时约 10ms,消除按键抖动
        while (GPIOPinRead(KEY_PORT,KEY_PIN)==0x00);                //等待 KEY 抬起
        SysCtlDelay(10*(TheSysClock/3000));                         //延时约 10ms,消除松键抖动
    }
}
```

小　结

本章主要介绍了系统控制部分，包括电源结构、LDO 控制、时钟控制、复位控制、外设控制、睡眠与深度睡眠、杂项功能和中断操作。

思 考 题

一、填空题

1. Standstorm 家族的处理器的时钟来源_____、_____。
2. Fury、DustDeril 家族的处理器的时钟来源_____、_____、_____。
3. Stellaris 微控制器提供一个集成的 LDO 稳压电源，电压调节范围是：_____。

4. 睡眠模式指的是_____。

5. 深度睡眠模式指的是_____。

6. Stellaris 微控制器系统需要_____ V 的电源供电。

二、问答题

1. 简述 Stellaris 处理器 Fury、DustDevil 和 Sandstorm 家族电源结构的特点。

2. 简述 Stellaris 处理器 Fury、DustDevil 和 Sandstorm 家族时钟的特点。

3. 在 Stellaris 系列 ARM 中有多种复位源，所有复位标志都集中保存在一个复位原因寄存器(RSTC)里，其复位源的种类有哪些？

4. Stellaris 系列 ARM 中有多种复位源，分别简述复位源其复位条件。

5. 大功率设备往往也是具有一定危险性的设备，如电梯系统。如果系统意外产生某种故障，应当立即使电机停止运行(即令 PWM 输出无效)，以避免其长时间处于危险的运行状态。在 Stellaris 系列处理器上做了哪些有效的措施防止事故发生？

6. 简述 Standstorm 家族中 LM3S101 处理器电源结构，包括模拟电源、数字电源及 LDO。

7. 简述 Standstorm 家族中 LM3S8962 处理器电源结构，包括模拟电源、数字电源及 LDO。

8. 简述 LDO 控制库函数 SysCtlLDOSet()和 SysCtlLDOGet()的作用。

9. LDO 控制库函数 SysCtlLDOSet()可以用来设置处理器 LDO 电压，设置范围是多少？步长为多少？

10. Stellaris 微控制器内部和外部的晶振频率范围是多少？使用 PLL 功能时的工作频率是多少？

11. 请利用库函数设置外部晶振 8MHz，利用 PLL 后，让系统内核时钟为 20MHz。

12. 请利用库函数设置处理器进入睡眠模式，并说明如何退出睡眠模式？