目 录

第 1 章 系统控制(SysCtl)....1
1.1 LDO 控制....2
1.2 时钟控制....4
1.3 复位控制....11
1.4 外设控制....14
1.5 睡眠与深度睡眠....17
1.6 杂项功能....22
1.7 中断操作....24
1.8 时钟验证....25

第1章　系统控制(SysCtl)

函　数　原　型	页码
void SysCtlLDOSet(unsigned long ulVoltage)	2
unsigned long SysCtlLDOGet(void)	3
void SysCtlLDOConfigSet(unsigned long ulConfig)	3
void SysCtlClockSet(unsigned long ulConfig)	6
unsigned long SysCtlClockGet(void)	8
void SysCtlPWMClockSet(unsigned long ulConfig)	9
unsigned long SysCtlPWMClockGet(void)	9
void SysCtlADCSpeedSet(unsigned long ulSpeed)	9
unsigned long SysCtlADCSpeedGet(void)	9
void SysCtlUSBPLLEnable(void)	10
void SysCtlUSBPLLDisable(void)	10
void SysCtlReset(void)	12
void SysCtlResetCauseClear(unsigned long ulCauses)	12
unsigned long SysCtlResetCauseGet(void)	13
void SysCtlBrownOutConfigSet(unsigned long ulConfig, unsigned long ulDelay)	13
void SysCtlPeripheralEnable(unsigned long ulPeripheral)	14
void SysCtlPeripheralDisable(unsigned long ulPeripheral)	15
void SysCtlPeripheralReset(unsigned long ulPeripheral)	15
tBoolean SysCtlPeripheralPresent(unsigned long ulPeripheral)	16
void SysCtlPeripheralSleepEnable(unsigned long ulPeripheral)	16
void SysCtlPeripheralSleepDisable(unsigned long ulPeripheral)	16
void SysCtlPeripheralDeepSleepEnable(unsigned long ulPeripheral)	16
void SysCtlPeripheralDeepSleepDisable(unsigned long ulPeripheral)	16
void SysCtlPeripheralClockGating(tBoolean bEnable)	16
void SysCtlSleep(void)	17
void SysCtlDeepSleep(void)	17
void SysCtlDelay(unsigned long ulCount)	22
unsigned long SysCtlFlashSizeGet(void)	22
unsigned long SysCtlSRAMSizeGet(void)	23
tBoolean SysCtlPinPresent(unsigned long ulPin)	23
void SysCtlGPIOAHBEnable(unsigned long ulGPIOPeripheral)	24
void SysCtlGPIOAHBDisable(unsigned long ulGPIOPeripheral)	24
void SysCtlIntRegister(void (*pfnHandler)(void))	24
void SysCtlIntUnregister(void)	24
void SysCtlIntEnable(unsigned long ulInts)	25
void SysCtlIntDisable(unsigned long ulInts)	25
unsigned long SysCtlIntStatus(tBoolean bMasked)	25
void SysCtlIntClear(unsigned long ulInts)	25
void SysCtlIOSCVerificationSet(tBoolean bEnable)	26

void SysCtlMOSCVerificationSet(tBoolean bEnable)	26
void SysCtlPLLVerificationSet(tBoolean bEnable)	26
void SysCtlClkVerificationClear(void)	26

系统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、杂项功能、中断操作、时钟验证。这其中比较常用和重要的函数并不多，掌握它们的具体用法也不复杂，尤其是最后两个部分通常情况下都用不到。

1.1 LDO 控制

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 管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 VDD（范围 3.0～3.6V），LDO 输出到一个名为“ LDO ”的管脚。对于 Fury 和 DustDevil 家族（LM3S1000 以上型号），LDO 管脚要连接到内核电源 VDD25 管脚上，对于 Sandstorm 家族（LM3S1000 以下型号）VDD25 管脚是内置的，因此不必从外部连接。

注意：在 LDO 管脚和 GND 之间必须接一个 1～3.3μF 的瓷片电容。

注意：在启用片内锁相环 PLL 之前，必须要将 LDO 电压设置在 2.75V。

表 1.1　函数 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

表 1.2 函数 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 1.1 当中参数 ulVoltage 的取值相同

表 1.3 函数 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

程序清单 1.1 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，我们可以拿万用表来测量 LDO 管脚的实际电压值大小。

程序清单 1.1 SysCtl 例程：控制 LDO 输出电压

```
#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
    };

    short i;
    char s[40];

    jtagWait( );                                    //  防止 JTAG 失效，重要！
    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
        }

        uartPuts("\r\n");
    }
}
```

1.2 时钟控制

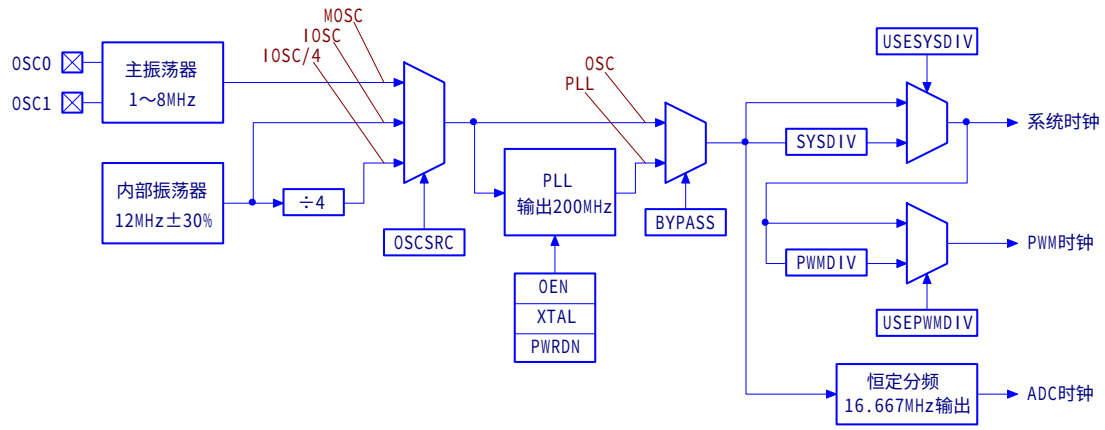

图 1.1　LM3S615 时钟系统结构图

如图 1.1 所示，为 LM3S615 芯片的时钟系统结构图。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（要求必须启用锁相环 PLL）。

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。

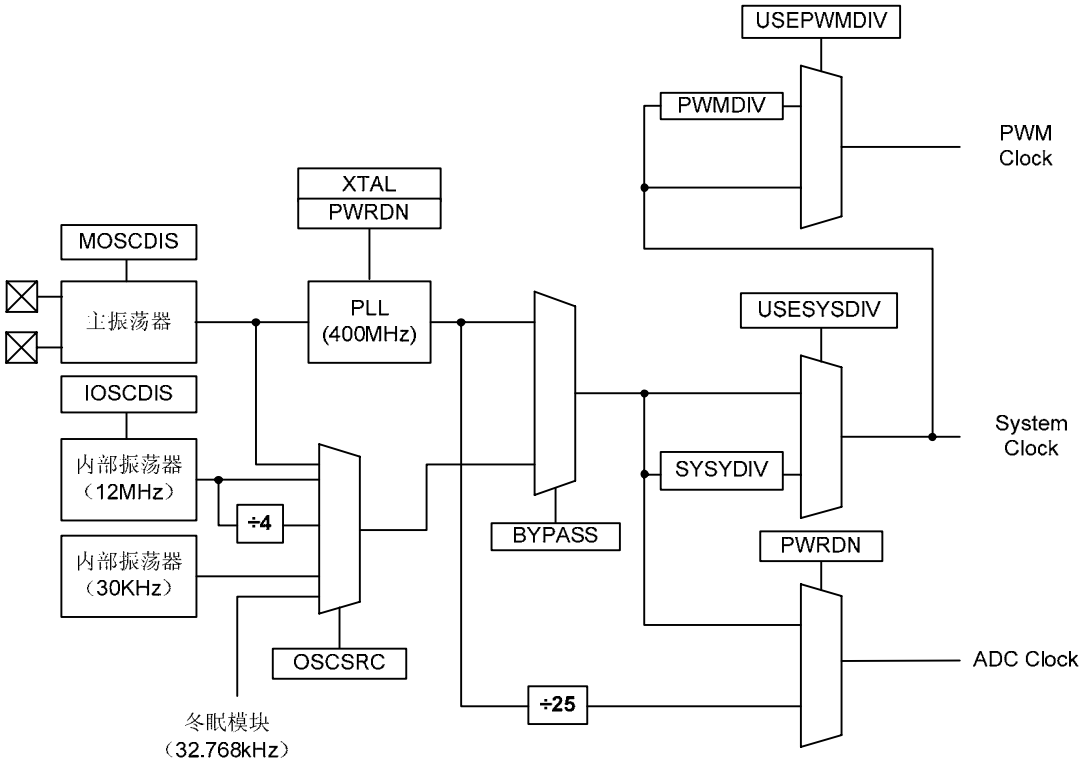

图 1.2　LM3S1138 时钟系统结构图

如图 1.2 所示，为 LM3S1138 芯片的时钟系统结构图，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。

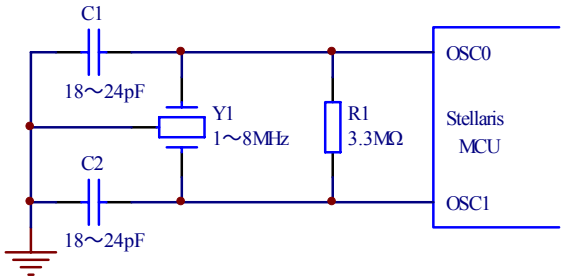

图 1.3　MOSC 外接晶振典型用法

主振荡器 MOSC 可以连接一个 1～8.192MHz 的外部晶体（2008 年新出型号可以支持到 16.384MHz）。典型接法如图 1.3 所示。如果不使用晶体，则外部的有源振荡信号也可以从 OSC0 管脚输入，要求信号幅度介于 0～3.3V 之间，此时 OSC1 管脚应当悬空。

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，因此对时钟精度有要求严格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），在冬眠备用电源管脚 VBAT 正常供电的情况下，可以从冬眠专用的晶振管脚 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。

Stellaris 系列 ARM 内部集成有一个 PLL（Phase Locked Loop，锁相环）。PLL 输出频率固定为 400MHz（Sandstorm 家族为 200MHz），误差±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。

注意：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);
```

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。参见函数 SysCtlPWMClockSet( )。

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 LM3S1138 的 ADC 采样速率是 1M。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，参见函数 SysCtlADCSpeedSet( )。

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 1.1 和图 1.2），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 管脚直接输入 16MHz 的有源振荡信号。

表 1.4　函数 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，则本函数将轮询 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
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，对时钟精度有要求严格的场合不适宜采用。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 管脚输入的有源振荡信号，并且要保证冬眠模块（Hibernation Module）VBAT 管脚的正常供电。

● 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 年新推出的型号，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 3.57～8.192MHz 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

示例	// 采用 6MHz 晶振作为系统时钟
	SysCtlClockSet(SYSCTL_USE_OSC |
	SYSCTL_OSC_MAIN |
	SYSCTL_XTAL_6MHZ |
	SYSCTL_SYSDIV_1);
	
	// 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);
	
	// 采用内部 12MHz 振荡器作为系统时钟
	SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_INT | SYSCTL_SYSDIV_1);
	
	// 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);
	
	// 采用内部 30KHz 振荡器作为系统时钟
	SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_INT30 | SYSCTL_SYSDIV_1);
	
	// 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 1.5　函数 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 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（参

	考表 1.4），则不会返回精确的结果。

表 1.6　函数 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 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 1.7　函数 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 1.6 当中参数 ulConfig 的取值相同

表 1.8　函数 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 1.9　函数 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 1.8 当中参数 ulSpeed 的取值相同

表 1.10　函数 SysCtlUSBPLLEnable( )

功能	使能 USB 模块专用的 PLL 单元
原型	void SysCtlUSBPLLEnable(void)
参数	无
返回	无
说明	2008 年新推出的 LM3S3xxx 和 5xxx 系列芯片集成有 USB 2.0 全速/OTG/Host/Device 功能，其 PLL 单元是专用的，输出 240MHz，经固定的 4 分频后作为 USB 模块的工作时钟。

表 1.11　函数 SysCtlUSBPLLDisable( )

功能	禁止 USB 模块专用的 PLL 单元
原型	void SysCtlUSBPLLDisable(void)
参数	无
返回	无

程序清单 1.2 演示了系统时钟设置函数 SysCtlClockSet( )函数的用法。在程序中，函数 ledFlash( )可以使 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 1.2　SysCtl 例程：系统时钟设置

```
#include   "systemInit.h"

//  定义 LED
#define   LED_PERIPH                SYSCTL_PERIPH_GPIOG
#define   LED_PORT                  GPIO_PORTG_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)
{
        jtagWait( );                                            //  防止 JTAG 失效，重要！

        SysCtlPeriEnable(LED_PERIPH);                           //  使能 LED 所在的 GPIO 端口
        GPIOPinTypeOut(LED_PORT, LED_PIN);                      //  设置 LED 所在管脚为输出

        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
        }
}
```

1.3 复位控制

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。

- 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。

- 外部复位（EXT）

芯片正在工作时，如果复位管脚/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，其它标志（POR 除外）都被清零。

- 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）。软件复位后，SW 标志置位，其它复位标志不变。

- 看门狗复位（WDT）

如果使能了看门狗模块的复位功能，则因为没有及时“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，其它复位标志不变。

- 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，其它复位标志不变。

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）。

- LDO 复位（LDO）

当 LDO 供电不可调整时，例如 LDO 输出管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，其它标志不变。

表 1.12　函数 SysCtlReset( )

功能	软件复位
原型	void SysCtlReset(void)
参数	无
返回	该函数不会返回，一旦调用就会使整个芯片产生复位

表 1.13　函数 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 1.14　函数 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 1.14 当中参数 ulCauses 的取值相同

表 1.15　函数 SysCtlBrownOutConfigSet( )

功能	配置掉电控制
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的内部振荡器周期数，该值只在 SYSCTL_BOR_RESAMPLE 被设置后并且小于 8192 时才有意义
返回	无

程序清单 1.3 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”。如果还存在其它可能的复位方式，也会正确显示出来。

程序清单 1.3　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;

      jtagWait( );                                          //   防止 JTAG 失效，重要！
      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 3500ms
    SysCtlReset( );                                             //  软件复位

    for (;;)                                                    //  不会执行到这里
    {
    }
}
```

1.4 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。使能片内外设的函数是 SysCtlPeripheralEnable( )，对该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，则可以用函数 SysCtlPeripheralDisable( )将其禁止，以节省功耗。其它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。

表 1.16　函数 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 IC 总线） SYSCTL_PERIPH_I2C0 // I^2C 0 （互联 IC 总线 0，与 I^2C 等同） SYSCTL_PERIPH_I2C1 // I^2C 1 （互联 IC 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） 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_IEEE1588 // IEEE1588 SYSCTL_PERIPH_UDMA // uDMA controller （μDMA 控制器） SYSCTL_PERIPH_USB0 // USB0 controller （USB0 控制器）
返回	

表 1.17 函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.18 函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.19　函数 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 1.16 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 1.20　函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.21　函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.22　函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.23　函数 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 1.16 当中参数 ulPeripheral 的取值相同
返回	无

表 1.24　函数 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应该使用时取值 true，否则取值 false
返回	无

1.5 睡眠与深度睡眠

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)。

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 1.25 的描述。

表 1.25 运行、睡眠、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	小	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。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。

表 1.26 函数 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 1.27 函数 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）

程序清单 1.4 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 1.4　SysCtl 例程：睡眠省电模式

```
#include   "systemInit.h"

#define   SysCtlPeriClkGating          SysCtlPeripheralClockGating
#define   SysCtlPeriSlpEnable          SysCtlPeripheralSleepEnable

//   定义 LED
#define   LED_PERIPH                   SYSCTL_PERIPH_GPIOG
#define   LED_PORT                     GPIO_PORTG_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_1

//   按键初始化
void keyInit(void)
{
      SysCtlPeriEnable(KEY_PERIPH);                              //   使能 KEY 所在的 GPIO 端口
      GPIOPinTypeIn(KEY_PORT, KEY_PIN);                          //   设置 KEY 所在管脚为输入
      GPIOIntTypeSet(KEY_PORT, KEY_PIN, GPIO_LOW_LEVEL);         //   设置 KEY 的中断类型
      GPIOPinIntEnable(KEY_PORT, KEY_PIN);                       //   使能 KEY 中断
      IntEnable(INT_GPIOD);                                      //   使能 GPIOD 中断
      IntMasterEnable( );                                        //   使能处理器中断
}

//   主函数（程序入口）
int main(void)
{
      jtagWait( );                                               //   防止 JTAG 失效，重要！
      clockInit( );                                              //   时钟初始化：晶振，6MHz
      keyInit( );                                                //   按键初始化

      SysCtlPeriEnable(LED_PERIPH);                              //   使能 LED 所在的 GPIO 端口
      GPIOPinTypeOut(LED_PORT, LED_PIN);                         //   设置 LED 所在管脚为输出
```

```
    GPIOPinWrite(LED_PORT, LED_PIN, 0x00);                  //  点亮 LED，表示工作状态
    SysCtlDelay(2500 * (TheSysClock / 3000));

    //  允许在睡眠模式下外设采用寄存器 SCGCn 配置时钟
    SysCtlPeriClkGating(true);

    //  允许 KEY 所在 GPIO 端口在睡眠模式下继续工作
    SysCtlPeriSlpEnable(KEY_PERIPH);

    for (;;)
    {
        GPIOPinWrite(LED_PORT, LED_PIN, 1 << 2);            //  熄灭 LED，表示进入睡眠
        SysCtlSleep( );                                     //  使处理器进入睡眠模式
        GPIOPinWrite(LED_PORT, LED_PIN, 0x00);              //  点亮 LED，表示已被唤醒
        SysCtlDelay(2500 * (TheSysClock / 3000));           //  工作一段时间后，再次睡眠
    }
}

//  GPIOD 的中断服务函数
void GPIO_Port_D_ISR(void)
{
    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 中断状态有效
    {
        SysCtlDelay(10 * (TheSysClock / 3000));             //  延时，以消除按键抖动

        while (GPIOPinRead(KEY_PORT, KEY_PIN) == 0);        //  等待按键抬起

        SysCtlDelay(10 * (TheSysClock / 3000));             //  延时，以消除松键抖动
    }
}
```

程序清单 1.5 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 模块的用法我们将在后续章节里详细讨论。

在程序清单 1.5 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 1.5　SysCtl 例程：深度睡眠省电模式

```
#include   "systemInit.h"
#include   <hw_sysctl.h>
#include   <timer.h>

#define   SysCtlPeriClkGating          SysCtlPeripheralClockGating
#define   SysCtlPeriDSlpEnable         SysCtlPeripheralDeepSleepEnable

//   定义 KEY
#define   KEY_PERIPH                   SYSCTL_PERIPH_GPIOD
#define   KEY_PORT                     GPIO_PORTD_BASE
#define   KEY_PIN                      GPIO_PIN_1

//   按键初始化
void keyInit(void)
{
      SysCtlPeriEnable(KEY_PERIPH);                              //   使能 KEY 所在的 GPIO 端口
      GPIOPinTypeIn(KEY_PORT, KEY_PIN);                          //   设置 KEY 所在管脚为输入
      GPIOIntTypeSet(KEY_PORT, KEY_PIN, GPIO_LOW_LEVEL);         //   设置 KEY 的中断类型
      GPIOPinIntEnable(KEY_PORT, KEY_PIN);                       //   使能 KEY 中断
      IntEnable(INT_GPIOD);                                      //   使能 GPIOD 中断
      IntMasterEnable( );                                        //   使能处理器中断
}

//   在 PG4/CCP3 管脚产生 2500KHz 方波，使蜂鸣器发声
void sound(void)
{
      SysCtlPeriEnable(SYSCTL_PERIPH_TIMER1);                    //   使能 TIMER1 模块
      SysCtlPeriEnable(SYSCTL_PERIPH_GPIOG);                     //   使能 CCP3 所在的 GPIO 端口
      GPIOPinTypeTimer(GPIO_PORTG_BASE, GPIO_PIN_4);             //   配置相关管脚为 Timer 功能

      TimerConfigure(TIMER1_BASE, TIMER_CFG_16_BIT_PAIR |        //   配置 TimerB 为 16 位 PWM
                                  TIMER_CFG_B_PWM);

      TimerLoadSet(TIMER1_BASE, TIMER_B, 5000);                  //   设置 TimerB 初值
      TimerMatchSet(TIMER1_BASE, TIMER_B, 2500);                 //   设置 TimerB 匹配值
      TimerEnable(TIMER1_BASE, TIMER_B);
}

//   主函数（程序入口）
int main(void)
{
```

```
    jtagWait( );                                        //  防止 JTAG 失效，重要！
    clockInit( );                                       //  时钟初始化：PLL，12.5MHz
    keyInit( );                                         //  按键初始化

    //  允许 Timer1 模块在深度睡眠模式下继续工作
    SysCtlPeriDSlpEnable(SYSCTL_PERIPH_TIMER1);

    //  允许 Buzzer 所在的 GPIO 端口在深度睡眠模式下继续工作
    SysCtlPeriDSlpEnable(SYSCTL_PERIPH_GPIOG);

    //  允许 KEY 所在 GPIO 端口在深度睡眠模式下继续工作
    SysCtlPeriDSlpEnable(KEY_PERIPH);

    //  允许在深度睡眠下外设采用寄存器 DCGCn 配置时钟
    SysCtlPeriClkGating(true);

    //  置位 DSLPCLKCFG 寄存器中的 IOSC 位，将来进入深度睡眠模式后，系统时钟改由 IOSC 提供
    HWREGBITW(SYSCTL_DSLPCLKCFG, 0) = 1;

    sound( );                                           //  蜂鸣器发声 2500Hz，尖叫

    for (;;)
    {
        //  延时一段时间，此时 Timer 模块的时钟由 PLL 提供
        SysCtlDelay(2500 * (TheSysClock / 3000));

        //  进入深度睡眠，等待按键唤醒，PLL 被禁止，Timer 模块的时钟改由 IOSC/16 提供
        SysCtlDeepSleep( );                             //  蜂鸣器发声约 150Hz，低沉
    }
}

//  GPIOD 的中断服务函数
void GPIO_Port_D_ISR(void)
{
    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 中断状态有效
    {
        SysCtlDelay(10 * (TheSysClock / 3000));         //  延时，以消除按键抖动

        while (GPIOPinRead(KEY_PORT, KEY_PIN) == 0);    //  等待按键抬起
```

```
            SysCtlDelay(10 * (TheSysClock / 3000));                    //  延时，以消除松键抖动
        }
}
```

1.6 杂项功能

这是一组杂项功能的函数，包括延时、存储器大小、特定管脚是否存在、高速 GPIO 等。

函数 SysCtlDelay( )提供一个产生一个固定长度延时的方法。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
```

函数 SysCtlFlashSizeGet( )和 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）。

函数 SysCtlPinPresent( )用来确认非 GPIO 片内外设的特定功能管脚是否存在。

函数 SysCtlGPIOAHBEnable( )和 SysCtlGPIOAHBDisable( )用来管理 GPIO 高速访问总线的使用。

在 2008 年新推出的型号（LM3S3xxx/5xxx 系列，以及部分 LM3S1xxx/2xxx 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、GPIO_PORTB_BASE 等，在使能 AHB 功能后要相应地换成 GPIO_PORTA_AHB_BASE、GPIO_PORTB_AHB_BASE 等，才能正确地使用 AHB 功能。

表 1.28　函数 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
示例	SysCtlDelay(20);　　// 延时 60 个系统时钟周期 SysCtlDelay(150 * (SysCtlClockGet( ) / 3000));　　// 延时 150ms

表 1.29　函数 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 1.30　函数 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

表 1.31　函数 SysCtlPinPresent( )

功能	确认非 GPIO 片内外设的特定功能管脚是否存在
原型	tBoolean SysCtlPinPresent(unsigned long ulPin)
参数	ulPin：待断定的管脚，应当取下列值之一： SYSCTL_PIN_PWM0　// PWM0 管脚 SYSCTL_PIN_PWM1　// PWM1 管脚 SYSCTL_PIN_PWM2　// PWM2 管脚 SYSCTL_PIN_PWM3　// PWM3 管脚 SYSCTL_PIN_PWM4　// PWM4 管脚 SYSCTL_PIN_PWM5　// PWM5 管脚 SYSCTL_PIN_PWM6　// PWM6 管脚 SYSCTL_PIN_PWM7　// PWM7 管脚 SYSCTL_PIN_C0MINUS　// C0- 管脚 SYSCTL_PIN_C0PLUS　// C0+ 管脚 SYSCTL_PIN_C0O　// C0o 管脚 SYSCTL_PIN_C1MINUS　// C1- 管脚 SYSCTL_PIN_C1PLUS　// C1+ 管脚 SYSCTL_PIN_C1O　// C1o 管脚 SYSCTL_PIN_C2MINUS　// C2- 管脚 SYSCTL_PIN_C2PLUS　// C2+ 管脚 SYSCTL_PIN_C2O　// C2o 管脚 SYSCTL_PIN_MC_FAULT0　// MC0 Fault 管脚 SYSCTL_PIN_ADC0　// ADC0 管脚 SYSCTL_PIN_ADC1　// ADC1 管脚 SYSCTL_PIN_ADC2　// ADC2 管脚 SYSCTL_PIN_ADC3　// ADC3 管脚 SYSCTL_PIN_ADC4　// ADC4 管脚 SYSCTL_PIN_ADC5　// ADC5 管脚 SYSCTL_PIN_ADC6　// ADC6 管脚 SYSCTL_PIN_ADC7　// ADC7 管脚 SYSCTL_PIN_CCP0　// CCP0 管脚 SYSCTL_PIN_CCP1　// CCP1 管脚

	SYSCTL_PIN_CCP2 // CCP2 管脚 SYSCTL_PIN_CCP3 // CCP3 管脚 SYSCTL_PIN_CCP4 // CCP4 管脚 SYSCTL_PIN_CCP5 // CCP5 管脚 SYSCTL_PIN_32KHZ // 32kHz 管脚
返回	如果要确认的外设管脚存在则返回 true，否则返回 false

表 1.32 函数 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 1.33 函数 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

1.7 中断操作

系统控制中断的操作是比较简单的，在早期的型号里，实际编程可能用到的中断源只有 PLL 锁定中断和掉电复位中断，并且也不常用。因此这些中断控制函数一般不要使用。

表 1.34 函数 SysCtlIntRegister( )

功能	注册一个系统控制中断的服务函数
原型	void SysCtlIntRegister(void (*pfnHandler)(void))
参数	pfnHandler：函数指针，指向中断产生时被调用函数
返回	无

表 1.35 函数 SysCtlIntUnregister( )

功能	注销系统控制中断的服务函数
原型	void SysCtlIntUnregister(void)

参数	无
返回	无

表 1.36　函数 SysCtlIntEnable( )

功能	使能指定的系统控制中断源
原型	void SysCtlIntEnable(unsigned long ulInts)
参数	ulInts：要使能的中断源，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_INT_PLL_LOCK　　//　PLL 锁定中断 SYSCTL_INT_CUR_LIMIT　　//　电流限制中断（不要用） SYSCTL_INT_IOSC_FAIL　　//　内部振荡器失效中断（不要用） SYSCTL_INT_MOSC_FAIL　　//　主振荡器失效中断（不要用） SYSCTL_INT_POR　　//　上电复位中断（不要用） SYSCTL_INT_BOR　　//　掉电复位中断 SYSCTL_INT_PLL_FAIL　　//　PLL 失效中断（不要用） 注：标明“不要用”的中断源在早期的型号里存在但不常用，而在后续的型号里已取消
返回	无

表 1.37　函数 SysCtlIntDisable( )

功能	禁止指定的系统控制中断源
原型	void SysCtlIntDisable(unsigned long ulInts)
参数	ulInts：要禁止的中断源
返回	无

表 1.38　函数 SysCtlIntStatus( )

功能	获取当前系统控制中断的状态
原型	unsigned long SysCtlIntStatus(tBoolean bMasked)
参数	bMasked：屏蔽标志，如果是 true 则返回屏蔽的中断状态，如果是 false 则返回原始的中断状态
返回	原始的中断状态或允许反映到处理器中的中断状态，与表 1.36 当中参数 ulInts 的取值相同

表 1.39　函数 SysCtlIntClear( )

功能	清除指定的系统控制中断源
原型	void SysCtlIntClear(unsigned long ulInts)
参数	ulInts：要清除的中断源，与表 1.36 当中参数 ulInts 的取值相同
返回	无

1.8　时钟验证

下面几个函数可以提供对内部振荡器 IOSC、主振荡器 MOSC、锁相环 PLL 的时钟验证功能，但都不常用。

表 1.40　函数 SysCtlIOSCVerificationSet( )

功能	配置内部振荡器验证定时器
原型	void SysCtlIOSCVerificationSet(tBoolean bEnable)
参数	bEnable：当内部振荡器验证定时器应当被使能时取值 true
返回	无

表 1.41　函数 SysCtlMOSCVerificationSet( )

功能	配置主振荡器验证定时器
原型	void SysCtlMOSCVerificationSet(tBoolean bEnable)
参数	bEnable：当主振荡器验证定时器应当被使能时取值 true
返回	无

表 1.42　函数 SysCtlPLLVerificationSet( )

功能	配置 PLL 验证定时器
原型	void SysCtlPLLVerificationSet(tBoolean bEnable)
参数	bEnable：当 PLL 验证定时器应该被使能时取值 true
返回	无

表 1.43　函数 SysCtlClkVerificationClear( )

功能	清除时钟验证状态
原型	void SysCtlClkVerificationClear(void)
参数	无
返回	无