目 录

第 1 章　中断控制(Interrupt) 1
1.1　中断基本编程方法.....1
1.2　中断库函数及例程.....3

第1章　中断控制(Interrupt)

函　数　原　型	页码
tBoolean IntMasterEnable(void)	4
tBoolean IntMasterDisable(void)	4
void IntEnable(unsigned long ulInterrupt)	4
void IntDisable(unsigned long ulInterrupt)	4
void IntPrioritySet(unsigned long ulInterrupt, unsigned char ucPriority)	8
long IntPriorityGet(unsigned long ulInterrupt)	8
void IntPriorityGroupingSet(unsigned long ulBits)	8
unsigned long IntPriorityGroupingGet(void)	8
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void))	11
void IntUnregister(unsigned long ulInterrupt)	11

1.1　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必须要首先使能，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，共 5 种，这要通过调用函数 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，通常要采取分 3 步走的方法：

- 调用片内外设具体中断的使能函数
- 调用函数 IntEnable( )，使能片内外设的总中断
- 调用函数 IntMasterEnable( )，使能处理器总中断

4．　编写中断服务函数

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　对于 GCC 编译器下的程序，中断服务函数的名称是事先约定好的。用户可以打开启动文件“ LM3S_Startup.s ”来查看每个中断服务函数的标准名称（参见表 1.1）。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) ”，参数和返回值都必须是 void 类型。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己指定，但还是推荐采用 GCC 下的标准名称，这样有

利于程序移植。

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 个管脚，每个管脚都可以产生中断，但是都共用同一个中断向量号 16，任一管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，则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 1.1 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、放心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 1.1　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
    }

    //  如果还有其它管脚的中断需要处理，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。

GCC　在 GCC 编译器下注册最简单，只要中断服务函数采用标准命名即可，而不需要去修改启动文件“ LM3S_Startup.s ”里的中断向量表。

Keil　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，根据注释内容把相应的“ IntDefaultHandler ”替换为“ I2C_ISR ”，并且在“ Vectors ”表格前面插入声明“ EXTERN　I2C_ISR ”，完成。

IAR　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，先插入函数声明“ void I2C_ISR(void); ”，然后在中断向量表里，根据注释内容把相应的“ 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。

表 1.1　GCC 编译器下的中断服务函数名称

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	45	FLASH_Control_ISR
2	Nmi_ISR	24	I2C_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	51	Timer3A_ISR
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		

1.2　中断库函数及例程

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

（Reset ISR）、不可屏蔽中断（NMI ISR）、硬件故障中断（Fault ISR），它们可能随时发生而无法通过软件禁止。参见表 1.2 和表 1.3 的描述。

库函数 IntEnable( )和 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、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、PWM 等，中断向量号在 16 以上。参见表 1.4、表 1.5 和表 1.6 的描述。

表 1.2　函数 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

表 1.3　函数 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 1.4　函数 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，具体取值请参考表 1.6 的描述
返回	无

表 1.5　函数 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，具体取值请参考表 1.6 的描述
返回	无

表 1.6　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　（硬件故障）
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）
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	SSI Rx and Tx （SSI 收发）
INT_SSI0	23	SSI0 Rx and Tx （SSI0 收发，与 INT_SSI 相同）
INT_I2C	24	I^2C Master and Slave （I^2C 主从）
INT_I2C0	24	I^2C0 Master and Slave （I^2C0 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 A （定时器 1 子定时器 A）
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	GPIO Port F （GPIO 端口 F）
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）
INT_I2C1	53	I^2C1 Master and Slave （I^2C1 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）

程序清单 1.2 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 端口并把相应的管脚设置为输入，然后配置中断触发类型并使能中断。

程序清单 1.2　GPIO 中断

```
#include   "systemInit.h"

//  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

//  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

    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

    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 管脚的中断类型
```

```
    GPIOPinIntEnable(KEY_PORT, KEY_PIN);                    //  使能 KEY 所在管脚的中断
    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
    }
}
```

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 IntPriorityGet( )用来管理一个片内外设的优先级，参见表 1.7 和表 1.8 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。

函数 IntPriorityGroupingSet( )和 IntPriorityGroupingGet( )用来管理抢占式优先级和子优先级的分组设置，参见表 1.9 和表 1.10 的描述。

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，则其它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它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 0～1 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。

表 1.7　函数 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，具体取值请参考表 1.6 的描述 ucPriority：要设定的优先级，应当取值(0～7) << 5，数值越小优先级越高
返回	无

表 1.8　函数 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，具体取值请参考表 1.6 的描述
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。

表 1.9　函数 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 1.10　函数 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 返回值是 3～7 效果等同

针对中断优先级，我们设计了一个简单的演示例程，参见程序清单 1.3。两路按键 KEY1、KEY2 输入采用不同的优先级中断，分别在各自的中断服务函数里控制指示灯 LED1、LED2。

在程序里，把 KEY1 中断设置为较高的优先级 1、KEY2 中断设置为较低的优先级 2。KEY1、KEY2 各自对应一个中断服务函数。在中断服务函数里只做两件事情：读取并清除中断状态、点亮对应的 LED 指示灯，最后进入一个死循环而不退出中断。

在程序运行后，如果先按 KEY1 点亮 LED1，再按 KEY2 时 LED2 不亮，原因是 KEY1 优先级比 KEY2 优先级高，KEY2 中断无法抢占 KEY1 中断。相反，如果先按 KEY2 点亮 LED2，再按 KEY1 时也能点亮 LED1，这说明较高优先级的 KEY1 中断能够抢占较低优先

级的 KEY2 中断。

我们可以在 main( )函数里插入一句对函数 IntPriorityGroupingSet( )的调用，如果参数是 0 或 1，则 KEY1 和 KEY2 中断都不能互相抢占对方；如果参数是 2，则 KEY1 中断的抢占式优先级为 0、KEY2 中断的抢占式优先级是 1，因此 KEY1 中断就可以抢占 KEY2 中断；如果参数是 3～7，则相当于默认的 8 级中断嵌套。

程序清单 1.3　中断优先级

```
//  包含必要的头文件
#include   "systemInit.h"

//  定义 LED1 和 LED2
#define   LED1_PERIPH              SYSCTL_PERIPH_GPIOD
#define   LED1_PORT                GPIO_PORTD_BASE
#define   LED1_PIN                 GPIO_PIN_0
#define   LED2_PERIPH              SYSCTL_PERIPH_GPIOG
#define   LED2_PORT                GPIO_PORTG_BASE
#define   LED2_PIN                 GPIO_PIN_2

//  定义 KEY1 和 KEY2
#define   KEY1_PERIPH              SYSCTL_PERIPH_GPIOD
#define   KEY1_PORT                GPIO_PORTD_BASE
#define   KEY1_PIN                 GPIO_PIN_1
#define   KEY2_PERIPH              SYSCTL_PERIPH_GPIOG
#define   KEY2_PORT                GPIO_PORTG_BASE
#define   KEY2_PIN                 GPIO_PIN_5

//  KEY1 中断初始化
void key1IntInit(void)
{
      SysCtlPeriEnable(KEY1_PERIPH);                          //  使能 KEY1 所在的 GPIO 端口
      GPIOPinTypeIn(KEY1_PORT, KEY1_PIN);                     //  设置 KEY1 所在管脚为输出
      GPIOIntTypeSet(KEY1_PORT, KEY1_PIN, GPIO_LOW_LEVEL);    //  设置 KEY1 的中断类型
      IntPrioritySet(INT_GPIOD, 1 << 5);                      //  设置 KEY1 中断优先级为 1
      GPIOPinIntEnable(KEY1_PORT, KEY1_PIN);                  //  使能 KEY1 所在管脚的中断
      IntEnable(INT_GPIOD);                                   //  使能 GPIOD 端口中断
}

//  KEY2 中断初始化
void key2IntInit(void)
{
      SysCtlPeriEnable(KEY2_PERIPH);                          //  使能 KEY2 所在的 GPIO 端口
      GPIOPinTypeIn(KEY2_PORT, KEY2_PIN);                     //  设置 KEY2 所在管脚为输出
      GPIOIntTypeSet(KEY2_PORT, KEY2_PIN, GPIO_LOW_LEVEL);    //  设置 KEY2 的中断类型
      IntPrioritySet(INT_GPIOG, 2 << 5);                      //  设置 KEY2 中断优先级为 2
```

```
    GPIOPinIntEnable(KEY2_PORT, KEY2_PIN);                  //  使能 KEY2 所在管脚的中断
    IntEnable(INT_GPIOG);                                   //  使能 GPIOG 端口中断
}

//  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

    SysCtlPeriEnable(LED1_PERIPH);                          //  使能 LED1 所在的 GPIO 端口
    GPIOPinTypeOut(LED1_PORT, LED1_PIN);                    //  设置 LED1 所在的管脚为输出
    GPIOPinWrite(LED1_PORT, LED1_PIN, 0x01);                //  熄灭 LED1

    SysCtlPeriEnable(LED2_PERIPH);                          //  使能 LED2 所在的 GPIO 端口
    GPIOPinTypeOut(LED2_PORT, LED2_PIN);                    //  设置 LED2 所在的管脚为输出
    GPIOPinWrite(LED2_PORT, LED2_PIN, 1 << 2);              //  熄灭 LED2

    key1IntInit( );                                         //  KEY1 中断初始化
    key2IntInit( );                                         //  KEY2 中断初始化
    IntMasterEnable( );                                     //  使能处理器中断

    for (;;)                                                //  等待按键中断
    {
    }
}

//  GPIOD 的中断服务函数
void GPIO_Port_D_ISR(void)
{
    unsigned long ulStatus;

    ulStatus = GPIOPinIntStatus(KEY1_PORT, true);           //  读取中断状态
    GPIOPinIntClear(KEY1_PORT, ulStatus);                   //  清除中断状态，重要

    if (ulStatus & KEY1_PIN)                                //  如果 KEY1 的中断状态有效
    {
        GPIOPinWrite(LED1_PORT, LED1_PIN, 0x00);            //  点亮 LED
        for (;;);                                           //  死循环，不退出中断服务函数
    }
}

//  GPIOG 的中断服务函数
void GPIO_Port_G_ISR(void)
```

```
{
    unsigned long ulStatus;

    ulStatus = GPIOPinIntStatus(KEY2_PORT, true);          //  读取中断状态
    GPIOPinIntClear(KEY2_PORT, ulStatus);                  //  清除中断状态，重要

    if (ulStatus & KEY2_PIN)                               //  如果 KEY2 的中断状态有效
    {
        GPIOPinWrite(LED2_PORT, LED2_PIN, 0x00);           //  点亮 LED2
        for (;;);                                          //  死循环，不退出中断服务函数
    }
}
```

3． 中断服务函数注册与注销

表 1.11　函数 IntRegister( )

功能	注册一个中断出现时被调用的函数
原型	void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void))
参数	ulInterrupt：指定的中断源，具体取值请参考表 1.6 的描述 pfnHandler：指向中断产生时被调用函数的指针
返回	无

表 1.12　函数 IntUnregister( )

功能	注销一个中断出现时被调用的函数
原型	void IntUnregister(unsigned long ulInterrupt)
参数	ulInterrupt：指定的中断源，具体取值请参考表 1.6 的描述
返回	无