



第3章 S800的嵌入式实验

- **3.1 实验一 时钟选择与 GPIO 实验 (扩展)**
(ref. B-chapter5, 10, C-chapter26,14)
- **3.2 实验二 I2C GPIO扩展及SYSTICK中断实验**
(ref. B-chapter3,18, C-chapter16,17,28)
- **3.3 实验三 UART串行通讯口实验**
(ref. B-chapter16, C-chapter30)

本章节参考资料:

- A. 自编讲义《嵌入式系统实验教程》
- B. Tiva™ TM4C1294NCPDT
Microcontroller Data Sheet
- C. TivaWare™ Peripheral Driver
Library User's Guide
- D. S800板介绍V0.65



实验一 时钟选择与 GPIO 实验（扩展）

■ 实验目的

- 理解NVIC的中断控制原理，掌握中断编程的一般步骤和程序组织形式，进一步掌握GPIO中断编程，实现输入输出控制
- 了解GPIO按键防抖处理方法，正确判断按键事件
- 熟悉SYSTICK 工作原理，能利用SysTick定时器进行定时控制



实验一 时钟选择与 GPIO 实验（扩展）

■ 预备知识

- 3.1.4 中断控制 (ref. B-chapter3, C-chapter17)
- 3.1.5 SysTick定时器 (ref. B-chapter3, C-chapter28)



3.1.4 TM4C1294NCPDT中断控制

- **NVIC向量中断控制器**与Cortex-M4处理器之间采用紧耦合接口
 - TM4C1294NCPDT NVIC 支持106个外部中断和11个系统异常
 - 3位优先级配置位（有效位 Bit[7:5]），除了3个固定的优先级（-1， -2， -3）之外，可设置8个优先级，分别为0x00~0x0E0
 - 支持抢占式优先级（组优先级）和非抢占式优先级（子优先级）
 - 抢占式响应：组优先级高的中断事件可以打断已被响应的组优先级低的中断服务程序。俗称**中断嵌套**；
 - 非抢断式响应：在组优先级相同的情况下，子优先级高的中断优先被响应。但如果存在低子优先级的中断正在执行，则高子优先级的中断要等低子优先级中断执行结束后才能得到响应，即不能嵌套。
 - 支持末尾连锁和快速中断处理
 - 末尾连锁的中断等待时间可降低为6个时钟周期（一般中断等待时间12时钟周期）



	优先级固定 -3、-2、-1							优先级可调整						
位置	0	1	2	3	4	5	6	7-10	11	12	13	14	15	≥16
异常类型	—	复位	NMI	硬故障	存储器管理	总线故障	使用故障	—	SVCall	调试监控器	—	PendSV	SysTick	外部中断

系统异常

- 所有的异常本身具有**硬件优先级**，其硬件优先级顺序决定于位置号（中断号），中断号越低，硬件优先级越高（不抢占）
- 可以通过软件设置异常的优先级，称为**软件优先级**。软件设置优先于硬件优先级
 - 注：软件优先级的设置对复位，NMI，和硬故障无效
- 用户可设置的最高优先级为 0，0 号优先级也是所有可调整优先级的默认值



■ 基于优先级的策略

➤ 规则1：组优先级决定抢占行为

- 多个中断源情况下，只有组优先级更高的中断源才可以打断某个已经在服务当中的组优先级较低的中断，而组优先级相同或更低的中断源不能打断它（可以末尾连锁）；
- TIVA C系列ARM只实现了3个优先级位，实际有效的组优先级位数只能设为0 ~ 3位。复位默认情况下，组优先级为0位，子优先级为3位，即不允许中断嵌套

➤ 规则2：子优先级决定响应次序

- 若多个组优先级相同的中断源同时产生中断，则子优先级高的中断被优先响应

➤ 规则3：中断号是默认的子优先级次序

- 多个中断源在组优先级和子优先级均相同的情况下，按中断矢量号的大小决定优先次序，即中断矢量号小的中断优先被响应。如SYSTICK优先于UART0中断



■ 基于优先级采取的动作：

动作	描述
抢占	产生条件：新的异常比当前的 ISR 或线程的优先级更高 发生时刻：ISR 或线程正在执行 中断结果：当前处于线程状态，则进入挂起中断；当前处于 ISR 状态，则产生中断嵌套 附加动作：处理器自动保存状态并压栈
末尾连锁	产生条件：新的异常优先级比当前正在返回的 ISR 的优先级更高 发生时刻：当前 ISR 结束时 中断结果：跳过出栈操作，将控制权转向新的 ISR
返回	产生条件：没有新的异常或没有比被压栈的 ISR 优先级更高的异常 发生时刻：当前 ISR 结束时 中断结果：执行出栈操作，并返回到被压栈的 ISR 或线程模式 附加动作：自动将处理器状态恢复为进入 ISR 之前的状态
迟来	产生条件：新的异常比正在保存状态的占先优先级更高 发生时刻：当前 ISR 开始时 中断结果：处理器转去处理优先级更高的中断



中断编程的操作步骤

■ TM4C1294NCPDT中断编程的基本步骤

1. 使能相关片上外设，并进行基本的配置
 2. 设置具体中断的类型或触发方式 等
 3. 设置中断分组和中断优先级（可选）
 4. 使能中断（三级控制）
 5. 编写和注册中断服务函数
- 由各外设的初始化完成
- 通过NVIC中断控制设置

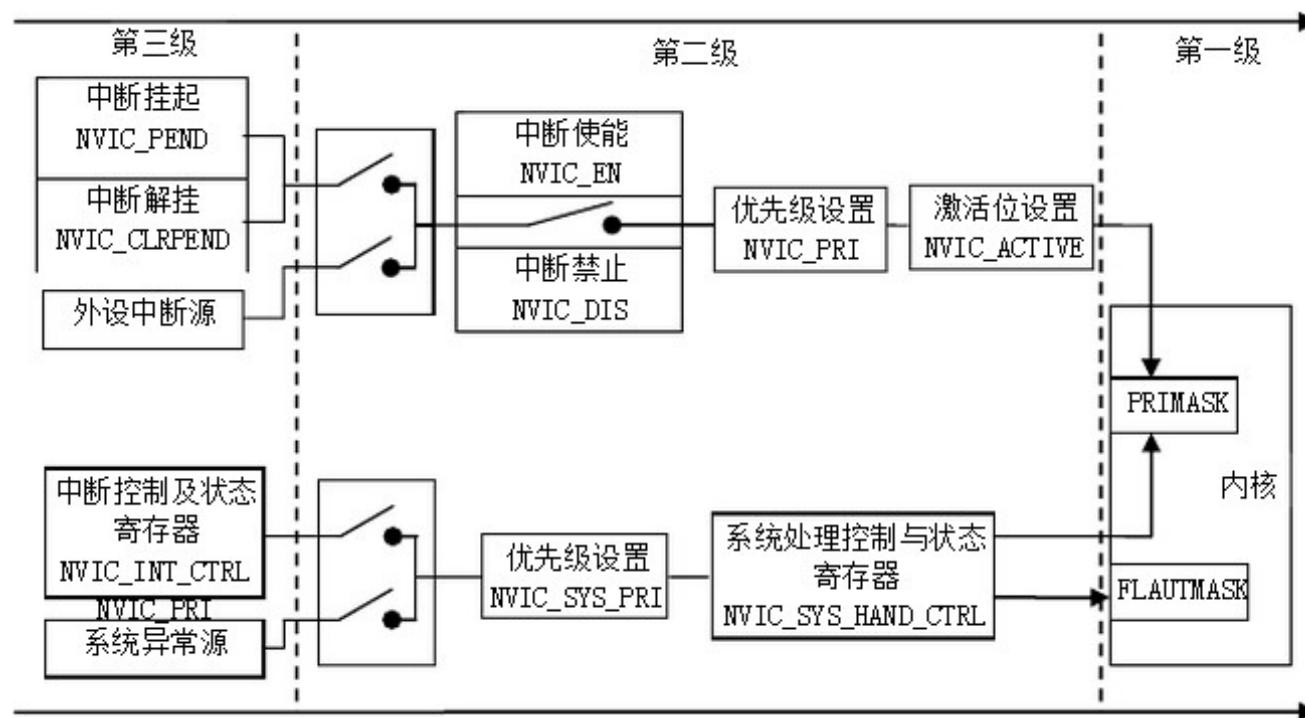


■ 中断使能的三级控制机制

1. 内核级

2. NVIC级

3. 异常/中断源级





■ TivaWare 中断控制库函数

(源程序在driverlib/interrupt.c中, API定义在driverlib/interrupt.h)

■ 中断使能和禁止:

- IntMasterEnable() 和 IntMasterDisable(): 使能和禁止处理器中断 (内核级, 寄存器 PRIMASK 被置 0或1, 禁止时只有NMI和硬件故障能被响应)
- IntEnable() 和 IntDisable(): 使能和禁止某个中断源 (NVIC级)
- xxxIntEnable(): 使能指定外设的中断 (源级, 参见各外设的库函数)

```
bool IntMasterEnable(void);
```

```
void IntEnable(uint32_t ui32Interrupt);
```

其中: ui32Interrupt为中断号



■ 中断类型号定义 (inc/hw_ints.h) (用作IntEnable等函数的参数)

```
#define FAULT_NMI          2      // NMI fault
#define FAULT_HARD        3      // Hard fault
#define FAULT_MPU         4      // MPU fault
#define FAULT_BUS         5      // Bus fault
#define FAULT_USAGE       6      // Usage fault
#define FAULT_SVCALL      11     // SVCcall
#define FAULT_DEBUG       12     // Debug monitor
#define FAULT_PENDSV      14     // PendSV
#define FAULT_SYSTICK     15     // System Tick

#define INT_GPIOA         16     // GPIO Port A
#define INT_GPIOB         17     // GPIO Port B
#define INT_GPIOC         18     // GPIO Port C
#define INT_GPIOD         19     // GPIO Port D
#define INT_GPIOE         20     // GPIO Port E
#define INT_UART0         21     // UART0
.....
```

Vector Number	Interrupt Number (Bit in Interrupt Registers)	Vector Address or Offset	Description
0-15	-	0x0000.0000 - 0x0000.003C	Processor exceptions
16	0	0x0000.0040	GPIO Port A
17	1	0x0000.0044	GPIO Port B
18	2	0x0000.0048	GPIO Port C
19	3	0x0000.004C	GPIO Port D
20	4	0x0000.0050	GPIO Port E
21	5	0x0000.0054	UART0
22	6	0x0000.0058	UART1
23	7	0x0000.005C	SSI0
24	8	0x0000.0060	I ² C0
25	9	0x0000.0064	PWM Fault
26	10	0x0000.0068	PWM Generator 0
27	11	0x0000.006C	PWM Generator 1
28	12	0x0000.0070	PWM Generator 2
29	13	0x0000.0074	QEI0
30	14	0x0000.0078	ADC0 Sequence 0
31	15	0x0000.007C	ADC0 Sequence 1
32	16	0x0000.0080	ADC0 Sequence 2
33	17	0x0000.0084	ADC0 Sequence 3
34	18	0x0000.0088	Watchdog Timers 0 and 1
35	19	0x0000.008C	16/32-Bit Timer 0A
36	20	0x0000.0090	16/32-Bit Timer 0B
37	21	0x0000.0094	16/32-Bit Timer 1A



■ 中断优先级设置

- `IntPrioritySet()` 和 `IntPriorityGet()` : 设置和检查中断的优先级
- `IntPriorityGroupingSet()`: 设置中断分组

void `IntPrioritySet` (uint32_t ui32Interrupt, uint8_t ui8Priority);

void `IntPriorityGroupingSet` (uint32_t ui32Bits);

其中: ui32Interrupt为中断号, ui8Priority为中断优先级,
ui32Bits指定抢占优先级的位数, 默认为7 (全抢占)

抢占级位数	Bit765二进制位	组优先级	子优先级	抢占式可选配置	子优先可选配置
0	b .yyy	无	3位: [7-5]	0	0~7
1	b x.yy	1位: [7]	2位: [6-5]	0~1	0~3
2	b xx.y	2位: [7-6]	1位: [5]	0~3	0~1
3-7	b xxx.	3位: [7-5]	无	0~7	0



■ NVIC中断优先级判断举例

设有两个中断源A和B，优先级分别设置为

■ `IntPrioritySet(INTA, 0x60);` 即INTA的优先级 = b01100000

■ `IntPrioritySet(INTB, 0x20);` 即INTB的优先级 = b00100000

1. 若设置抢占级位数=0x02, `IntPriorityGroupingSet(2);` 查表得优先级按照 bxx.y 来划分, 故

■ INTA的抢占优先级 = 01, 子优先级 = 1

■ INTB的抢占优先级 = 00, 子优先级 = 1

即, B的抢占优先级比A的高, B的中断可以抢占A的中断处理

2. 若设置抢占级位数=0x01, `IntPriorityGroupingSet(1);` 查表得优先级按照 bx.yy 来划分, 故

■ INTA的抢占优先级 = 0, 子优先级 = 11

■ INTB的抢占优先级 = 0, 子优先级 = 01

即, A和B处于同一个抢占优先级, 他们两个互相不能抢占。B中断的子优先级高, 当两个中断同时发生时, 会先处理B中断。



■ 编写和注册中断服务函数

- 中断服务函数命名和参数：使用默认注册的中断服务函数名称进行重写，无参数(void)，无返回类型(void)
 - 在生成项目时自动生成了所有的中断函数的入口函数
(参见startup_TM4C129.s中断向量表中定义的[WEAK]函数)

※ 关于[WEAK]符号

- 程序中不标记WEAK的函数是强符号
- 同名的强符号只能有一个，否则编译器报"重复定义"错误
- 允许有一个强符号和多个同名的弱符号，但定义会选择强符号
- 当没有强符号只有多个同名的弱符号时，链接器选择占用内存空间最大的那个



■ 默认的中断向量表 ([startup_TM4C129.s](#))

EXPORT __Vectors

```
__Vectors    DCD    __initial_sp           ; Top of Stack
              DCD    Reset_Handler         ; Reset Handler
              DCD    NMI_Handler           ; NMI Handler
              DCD    HardFault_Handler     ; Hard Fault Handler
              DCD    MemManage_Handler     ; MPU Fault Handler
              DCD    BusFault_Handler      ; Bus Fault Handler
              DCD    UsageFault_Handler    ; Usage Fault Handler

              DCD    SVC_Handler           ; SVCcall Handler

              DCD    PendSV_Handler        ; PendSV Handler
              DCD    SysTick_Handler       ; SysTick Handler

              ; External Interrupts

              DCD    GPIOA_Handler         ; 0: GPIO Port A
              DCD    GPIOB_Handler         ; 1: GPIO Port B

              DCD    GPIOJ_Handler         ; 51: GPIO PortJ

__Vectors_End
```



■ 默认的中断向量表 ([startup_TM4C129.s](#)) (续)

; Reset Handler

```
Reset_Handler PROC
    EXPORT Reset_Handler    [WEAK]
    IMPORT SystemInit
    IMPORT __main
    LDR    R0, =SystemInit
    BLX    R0
    LDR    R0, =__main
    BX     R0
ENDP
```


; Dummy Exception Handlers (infinite loops which can be modified)

.....

```
GPIOJ_Handler PROC
    EXPORT GPIOJ_Handler    [WEAK]
    B      .
ENDP
```



■ 重写对应的 [WEAK] 中断服务函数

- 中断状态查询：一个外设只有一个中断类型号/中断向量，但可能有多个中断源，需要通过中断状态查询语句（形如`xxxIntStatus()`）进一步确定
- 中断清除：对于**多源**中断，进入中断服务函数后，必须调用中断清除函数（形如`xxxIntClear()`）清除相应外设的中断请求 
- ※ 注意：执行清除中断请求需要几个处理器周期，因此**中断清除应当尽早进行**（一般在中断状态查询之后立即清除）

```
uint32_t GPIOIntStatus(uint32_t ui32Port, bool bMasked);
```

```
void GPIOIntClear(uint32_t ui32Port, uint32_t ui32IntFlags);
```

其中：ui32Port为端口基地址，bMasked指是否只含屏蔽后的中断源，
ui32IntFlags指定清除的位，通常直接使用GPIOIntStatus函数的返回值



3.1.5 SysTick定时器

- 系统节拍定时器SysTick
 - SysTick, 直译为“系统滴答”, 即系统的心跳
 - 集成在Cortex-M4内核中的一个功能单元
 - 具体硬件实现: 24位单调递减计数器
 - 对内核的时钟脉冲计数, 写入即清,
 - 实现: 从设定值开始减1计数, 过零自动重载, 循环
 - 系统节拍定时器, 用作任务切换
 - 也可用作简单计数器



实验一 时钟选择与 GPIO 实验

■ 实验内容

■ 扩展实验1-5：中断方式实现

- 当第一次短按USR_SW1键时，闪烁LED_M0
- 当第二次短按USR_SW1键时，熄灭LED_M0
- 当第三次短按USR_SW1键时，闪烁LED_M1
- 当第四次短按USR_SW1键时，熄灭LED_M1

- #### ■ 编程要点
1. 程序组织
 2. GPIO的中断编程
 3. 按键检测和计数

前台程序

```
void GPIOJ_Handler(void)
{
    unsigned long intStatus;

    intStatus = GPIOIntStatus(GPIO_PORTJ_BASE, true);
    GPIOIntClear(GPIO_PORTJ_BASE, intStatus );

    if (intStatus & GPIO_PIN_0) { //PJ0中断?
        btn_cnt = btn_cnt % 4 + 1;
    }
}
```

后台程序

```
int main(void)
{
    S800_Clock_Init();
    S800_GPIO_Init();

    IntMasterEnable();//开中断

    while (1){
        switch(btn_cnt){
            ...
        }
    }
}
```



实验一 时钟选择与 GPIO 实验

■ 实验内容

■ 扩展实验1-5：中断方式实现

- 当第一次短按USR_SW1键时，闪烁LED_M0
- 当第二次短按USR_SW1键时，熄灭LED_M0
- 当第三次短按USR_SW1键时，闪烁LED_M1
- 当第四次短按USR_SW1键时，熄灭LED_M1

- #### ■ 编程要点
1. 程序组织
 2. GPIO的中断编程
 3. 按键检测和计数



■ 中断解析：函数与中断服务函数的区别

■ 函数：由程序控制它的调用，出现时间**确定**

- 定义、声明，函数名和参数自己定
- 主程序或其它函数中直接调用（调用之前需要先函数声明）

■ 中断服务函数：事件驱动，出现时间**不定**

- 使用系统预设的中断服务函数名定义，无参数、无返回类型。
- 由特定事件驱动，不能直接调用
- **问题：如何调用中断服务函数**

```
_Vectors      DCD    __initial_sp          ; Top of Stack
               DCD    Reset_Handler       ; Reset Handler
               DCD    NMI_Handler         ; NMI Handler
               DCD    HardFault_Handler   ; Hard Fault Handler
               DCD    GPIOJ_Handler       ; 51: GPIO Port J
```



■ 中断解析：中断服务函数的调用

■ 中断响应流程

- (外部设备) 置位相应的中断请求信号通知CPU (类似INTR)
- (CPU) 执行完单条指令后检测是否有中断请求信号
 - 有 → 进入中断响应 (类似INTA) (条件：处理器中断被使能)
 - 无 → 继续下一条指令
- (外部设备) 将中断号告知CPU
- (CPU) 读取中断号
- (CPU) 根据中断号去中断向量表读取中断服务程序入口地址
- (CPU) 进入中断服务程序

■ 程序员如何做？

- 目的：配合CPU找到中断程序
- 使用.s文件中预设的中断服务函数名编写中断服务程序



■ GPIO中断编程的基本步骤

1. GPIO外设使能，并进行基本的配置
2. 设置GPIO引脚的中断类型或触发方式 等
3. 设置中断分组和GPIO中断优先级（单任务可省略）
4. 使能中断（GPIO引脚源级、NVIC级、内核级）
5. 重写GPIO中断服务函数



■ TivaWare GPIO中断控制库函数

- `GPIOIntEnable()` 和 `GPIOIntDisable()`: 使能和禁止GPIO引脚的中断
- `GPIOIntTypeSet()`: 设置GPIO引脚的中断触发方式
- `GPIOIntStatus()`: 读取GPIO各引脚的中断请求状况
- `GPIOIntClear()`: 清除GPIO指定引脚的中断请求

```
void GPIOIntEnable(uint32_t ui32Port, uint8_t ui8Pins);
```

```
void GPIOIntTypeSet (uint32_t ui32Port, uint8_t ui8Pins, uint32_t ui32IntType);
```

```
uint32_t GPIOIntStatus(uint32_t ui32Port, bool bMasked);
```

```
void GPIOIntClear(uint32_t ui32Port, uint32_t ui32IntFlags);
```

其中: `ui32Port`为端口基地址, `ui8Pins`为引脚位(含多个位的“或”),
`ui32IntType`指定中断触发方式, `bMasked`指是否只含屏蔽后的中断源,
`ui32IntFlags`指定清除的位, 通常直接使用`GPIOIntStatus`函数的返回值



- GPIO中断触发方式定义 ([driver/gpio.h](#))
(用作GPIOIntTypeSet函数的参数)

```
#define GPIO_FALLING_EDGE    0x00000000 // Interrupt on falling edge
#define GPIO_RISING_EDGE     0x00000004 // Interrupt on rising edge
#define GPIO_BOTH_EDGES      0x00000001 // Interrupt on both edges
#define GPIO_LOW_LEVEL        0x00000002 // Interrupt on low level
#define GPIO_HIGH_LEVEL       0x00000006 // Interrupt on high level
```



1. GPIO外设使能，并进行基本的配置

- 调用系统外设控制 `SysCtlPeripheralEnable()` 函数使能外设

`SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);` //使能J口

`GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);` //设PJ0为输入

`GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0,
GPIO_STRENGTH_2MA,GPIO_PIN_TYPE_STD_WPU);`

2. 设置GPIO引脚的中断类型或触发方式

- 注意：不同外设中断的类型或触发方式略有不同

`GPIOIntTypeSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_FALLING_EDGE);`

3. 设置中断分组和GPIO中断优先级（使用默认设置）



4. 使能中断（GPIO引脚源级、NVIC级、内核级）

- **源级**：调用中断源的中断使能函数，形如**xxxIntEnable()**

GPIOIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_0); //使能PJ0引脚中断

```
void GPIOIntEnable(uint32_t ui32Port, uint8_t ui8Pins);
```

其中：ui32Port为端口基地址，ui8Pins为引脚位(含多个位的“或”)

- **NVIC级**：调用函数**IntEnable()**，使能相应外设中断

IntEnable(INT_GPIOJ); //开启PortJ中断源

- **内核级**：调用函数**IntMasterEnable()**，使能处理器中断

IntMasterEnable(); //打开中断，PRIMASK置0



5. 重写 GPIO PortJ 的中断服务函数

void GPIOJ_Handler(void) //使用系统预设的 PortJ 中断服务函数名

```
{
    unsigned long intStatus;
    intStatus = GPIOIntStatus(GPIO_PORTJ_BASE, true); //读取中断状态
    GPIOIntClear(GPIO_PORTJ_BASE, intStatus ); //清中断源
    //确定子中断源并作相应处理
    if (intStatus & GPIO_PIN_0){ //PJ0?
        btn_cnt = btn_cnt % 4 + 1; //按键次数计数, btn_cnt为全局变量
    }
    if (intStatus & GPIO_PIN_1){ //PJ1? (若PJ1中断也使能)
        .....
    }
}
```

__Vectors	DCD	__initial_sp	; Top of Stack
	DCD	Reset_Handler	; Reset Handler
	DCD	NMI_Handler	; NMI Handler
	DCD	HardFault_Handler	; Hard Fault Handler
	DCD	GPIOJ_Handler	; 51: GPIO Port J



■ GPIO中断编程主程序示例

```
#include "interrupt.h"    //增加
#include "hw_ints.h"      //增加
uint32_t ui32SysClock;    //定义一个全局变量用来存放当前时钟频率
int btn_cnt = 0;
int main(void)
{
    S800_Clock_Init();    // 设置系统时钟
    S800_GPIO_Init();     // 初始化GPIO引脚
    IntMasterEnable();    //使能处理器总中断
    while(1) {    switch(btn_cnt) ... //控制LED灯    }
}

void S800_Clock_Init(void)
{
    ui32SysClock = SysCtlClockFreqSet((SYSCTL_OSC_INT | SYSCTL_USE_OSC), 16000000);
}
```



```
void S800_GPIO_Init(void)
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);    //Enable PortF
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);    //Enable PortJ

    //Set PF0,PF1 as Output pin
    GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0 | GPIO_PIN_1);

    //Set the PJ0 as input pin
    GPIOPinTypeGPIOInput(GPIO_PORTJ_BASE, GPIO_PIN_0);
    GPIOPadConfigSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_STRENGTH_2MA,
                                                              GPIO_PIN_TYPE_STD_WPU);

    //设置PJ0下降沿触发
    GPIOIntTypeSet(GPIO_PORTJ_BASE, GPIO_PIN_0, GPIO_FALLING_EDGE);

    GPIOIntEnable(GPIO_PORTJ_BASE, GPIO_PIN_0);    //使能PJ0中断

    IntEnable(INT_GPIOJ);    //使能PJ口中断
}
```



实验一 时钟选择与 GPIO 实验

■ 实验内容

■ 扩展实验1-6：使用SysTick定时器功能控制LED_M0和LED_M1灯的闪烁频率

- 当第一次短按USR_SW1键时，以1s周期闪烁LED_M0
- 当第二次短按USR_SW1键时，熄灭LED_M0
- 当第三次短按USR_SW1键时，以2s周期闪烁LED_M1
- 当第四次短按USR_SW1键时，熄灭LED_M1

■ 编程要点

1. 程序组织

2. GPIO的中断按键检测和计数

3. SysTick定时器的中断编程



前后台程序组织

■ 后台程序

```
int main(void)
{
    S800_Clock_Init();
    S800_GPIO_Init();
    S800_SysTick_Init();

    IntMasterEnable();

    while (1) {
        if (systick_500ms_status) {
            systick_500ms_status = 0;
            ...
        }
        if (systick_1s_status) {
            systick_1s_status = 0;
            ...
        }
        if (btn_cnt == 2)
            ...
    }
}
```

■ 前台程序

```
void SysTick_Handler(void)
{
    static uint8_t k = 0;

    systick_500ms_status = 1;
    if (k) systick_1s_status = 1;
    k = 1 - k;
}
```

```
void GPIOJ_Handler(void)
{
    unsigned long intStatus;

    intStatus = GPIOIntStatus(GPIO_PORTJ_BASE, true);
    GPIOIntClear(GPIO_PORTJ_BASE, intStatus);

    if (intStatus & GPIO_PIN_0) { //PJ0中断?
        btn_cnt = btn_cnt % 4 + 1;
    }
}
```




实验一 时钟选择与 GPIO 实验

■ 实验内容

■ 扩展实验1-6：使用SysTick定时器功能控制LED_M0和LED_M1灯的闪烁频率

- 当第一次短按USR_SW1键时，以1s周期闪烁LED_M0
- 当第二次短按USR_SW1键时，熄灭LED_M0
- 当第三次短按USR_SW1键时，以2s周期闪烁LED_M1
- 当第四次短按USR_SW1键时，熄灭LED_M1

■ 编程要点

1. 程序组织

2. GPIO的中断按键检测和计数

3. SysTick定时器的中断编程



■ TivaWare SysTick库函数

- 源程序driverlib/systick.c, API定义在driverlib/systick.h

■ SysTick配置

- `SysTickPeriodSet()`: 设置SysTick定时/计数初值
- `uint32_t SysTickValueGet(void)`: 读取SysTick当前计数值

`void SysTickPeriodSet(uint32_t ui32Period);`

ui32Period: 定时/计数初值, 取值范围1~16,777,216 (即 2^{24})

定时周期 = ui32Period × 系统时钟周期

系统时钟	最大定时周期
16MHz	1048ms
50MHz	335ms
80MHz	209ms



■ SysTick计时启动和停止

- void SysTickEnable(void): 使能SysTick, 开始计数
- void SysTickDisable(void): 关闭SysTick, 停止计数

■ SysTick中断控制

- void SysTickIntEnable(void): 使能SysTick中断
- void SysTickIntDisable(void): 除能SysTick中断

- ※ 注1: SysTick是系统异常, 它的中断使能只需要两个步骤, 即SysTickIntEnable(); 和 IntMasterEnable();
- ※ 注2: SysTick是单源中断, 它的中断状态由硬件自动删除, 不需要中断清除
- ※ 注3: 学会用一个定时器产生多个定时时间



■ SysTick编程示例（用作普通定时器）

```
#include "interrupt.h"    //增加
#include "hw_ints.h"      //增加
#include "systick.h"      //增加
uint32_t ui32SysClock;    //定义一个全局变量用来存放当前时钟频率
uint8_t systick_500ms = 0; //定义一个全局变量来标记定时状态

int main(void)
{
    S800_Clock_Init(); S800_GPIO_Init();        // 设置系统时钟，初始化GPIO引脚
    S800_SysTick_Init();                        // 初始化SysTick
    IntMasterEnable();                          //使能处理器总中断

    while(1) {
        if (systick_500ms) { //每500ms反转一次LED
            GPIOPinWrite( GPIO_PORTF_BASE, GPIO_PIN_0, ~GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_0));
            systick_500ms = 0;
        }
    }
}
```



■ SysTick编程示例（续）

```
void S800_SysTick_Init(void)
{
    SysTickPeriodSet(ui32SysClock/2); //定时500ms
    //注意ui32SysClock/2 < 16777216, 即 ui32SysClock <= 32M
    SysTickEnable();           //启动定时
    SysTickIntEnable();        //中断使能
}

void SysTick_Handler(void)
{
    // 不需要中断清除函数，硬件会自动清除SysTick中断状态
    systick_500ms = 1;
}
```



- 完成实验一，提交实验报告和源程序 -