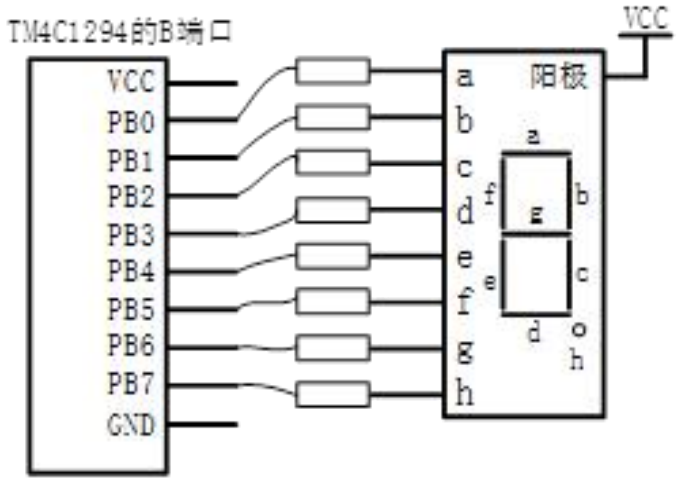
- 作业：
  - 使用PB口驱动一个7段数码管，画出端口与数码管的电路接线。写出端口的初始化程序。在数码管上显示数字7，写出相关的程序代码。




```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);

GPIOPinTypeGPIOOutput(GPIO_PORTB_BASE,
GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 |
GPIO_PIN_3 | GPIO_PIN_4 | GPIO_PIN_5 |
GPIO_PIN_6 | GPIO_PIN_7);
```

```
uint16_t LEDtb[]={0xc0, 0f9h, 0a4h, 0b0h, 99h, 92h, 82h,0f8h, 080h, 90h};
void LEDDisp(int value){
GPIOPinWrite(GPIO_PORTB_BASE,0xff, LEDtb [value]);}
LEDDisp(7);
```

微机系统与接口
——中断与异常

王江峰 副研究员
电气工程学院

1 中断与异常基本概念

- 如果下面有一段程序

```
While(1){
DoSomething();//耗时3s

CheckKey();//检测是否有按键按下

Display();//显示数据
}
```

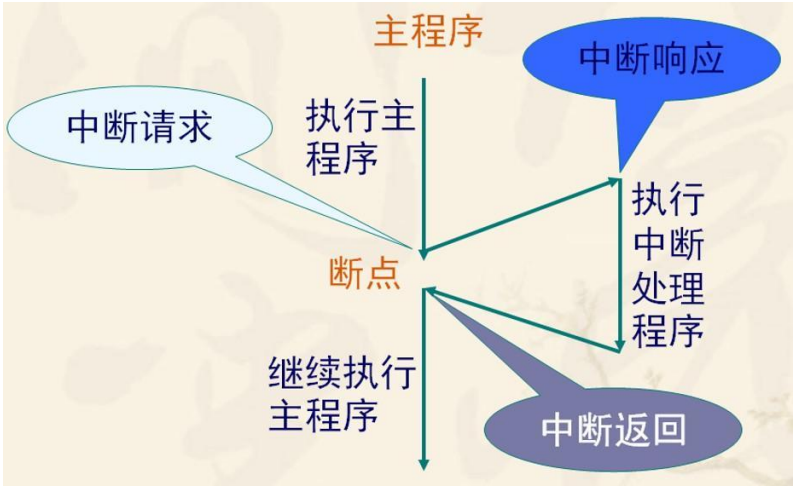
- 有可能检测不到按键
- 如何能够保证按键按下后能够及时检测到并响应？

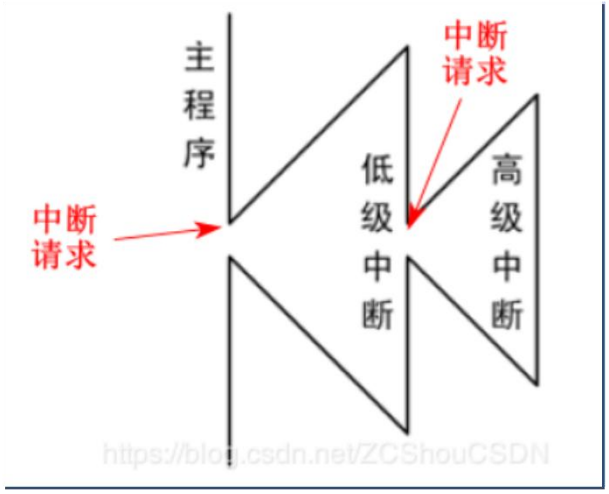
1 中断与异常基本概念

- CPU的中断和异常
  - 程序运行过程中，发生了不可预知的事件需要暂停当前的任务立刻处理

    ——查看按键是否按下，查看定时器是否超时

    ——除法除数为零、系统故障





1 中断与异常基本概念

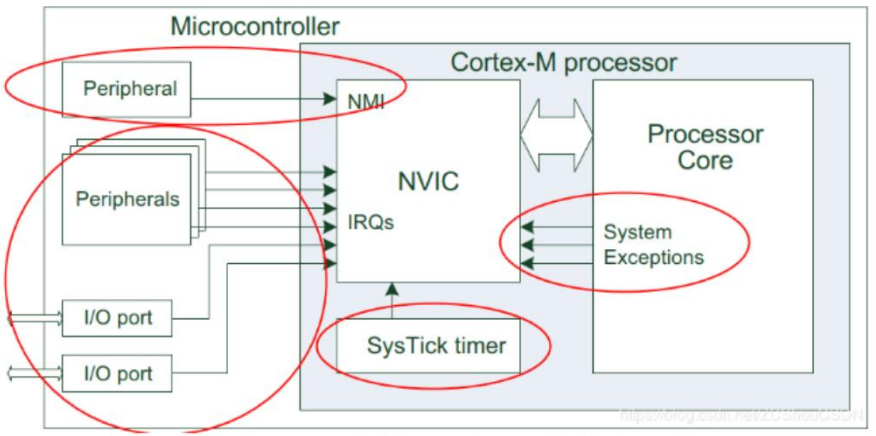
- CPU的中断和异常
  - 由嵌套向量中断控制器NVIC（Nested Vectored Interrupt Controller）管理

——灵活的中断和异常管理：禁止、使能、软件触发，清除、电平触发、脉冲触发

——支持嵌套：优先级管理与抢占

——向量化入口：自动定位中断

——中断也被看做是一种异常



Cortex-M3和 Cotex-M4 的 NVIC 最多支持 240 个 IRQ(中断请求)、1 个不可屏蔽中断(NMI)、1 个 Systick(滴答定时器)定时器中断和多个系统异常。而 Cortex-M0 最多支持32个IRQ、1 个不可屏蔽中断(NMI)、1 个 Systick(滴答定时器)定时器中断和多个系统异常。

IRQ： 多数由定时器、IO端口、通信接口等外设产生
NMI： 通常由看门狗定时器或者掉电检测器等外设产生
其他： 主要来自系统内核

[1]http://www.360doc.com/content/19/0121/17/16758694_810423077.shtml

1 中断与异常基本概念

- CPU的中断和异常
  - 在ARM中，一般由处理器系统内部引发的事件，称作异常，如重启Reset，硬件错误等。由处理器外部，也就是外设引发的事件，称作中断，如GPIO模块等
  - Cortex-M架构支持多种异常和外部中断，每个中断和异常都有编号，编号1~15为系统异常，16及以上的则为中断输入
  - 包括中断在内的大多数的异常的优先级都是可编程的，一些系统异常具有固定的优先级

1 中断与异常基本概念

- CPU的中断和异常
  - 系统异常列表

表 7.1 系统异常列表

异常编号	异常类型	优先级	描　述
1	复位	−3(最高)	复位
2	NMI	−2	不可屏蔽中断(外部 NMI 输入)
3	硬件错误	−1	所有的错误都可能会引发,前提是相应的错误处理未使能
4	MemManage 错误	可编程	存储器管理错误,存储器管理单元(MPU)冲突或访问非法位置
5	总线错误	可编程	总线错误。当高级高性能总线(AHB)接口收到从总线的错误响应时产生(若为取指也被称作预取终止,数据访问则为数据终止)
6	使用错误	可编程	程序错误或试图访问协处理器导致的错误(Cortex-M3 和 Cortex-M4 处理器不支持协处理器)
7～10	保留	NA	—
11	SVC	可编程	请求管理调用。一般用于 OS 环境且允许应用任务访问系统服务
12	调试监控	可编程	调试监控。在使用基于软件的调试方案时,断点和监视点等调试事件的异常
13	保留	NA	—
14	PendSV	可编程	可挂起的服务调用。OS 一般用该异常进行上下文切换
15	SYSTICK	可编程	系统节拍定时器。当其在处理器中存在时,由定时器外设产生。可用于 OS 或简单的定时器外设

1 中断与异常基本概念

- CPU的中断和异常
  - 外部中断列表

表 7.2 中断列表

异常编号	异常类型	优先级	描述
16	外部中断＃0	可编程	可由片上外设或外设中断源产生
17	外部中断＃1	可编程	
…	…	…	
255	外部中断＃239	可编程	

  - 异常编号与中断编号不同
  - 中断编号=异常编号-16

2 中断的响应流程

- 中断的响应流程

中断产生		中断响应	
外设		NVIC	CPU
产生中断	发出中断	接收中断	接收中断

- 接受异常请求
  - ——处理器正在运行
  - ——异常处于使能状态
  - ——异常的优先级高于当前等级
  - ——异常未被屏蔽

2 中断的响应流程

- 中断的响应流程
  - 异常进入

——CPU内核寄存器和返回地址被压入当前使用的栈

——取出中断向量

——取出待执行异常处理的指令

——更新NVIC寄存器和内核寄存器

可以同时进行

更新中断状态

更新程序逻辑的执行状态：程序状态寄存器（PSR）、链接寄存器（LR）、程序计数器(PC)、堆栈指针(SP)

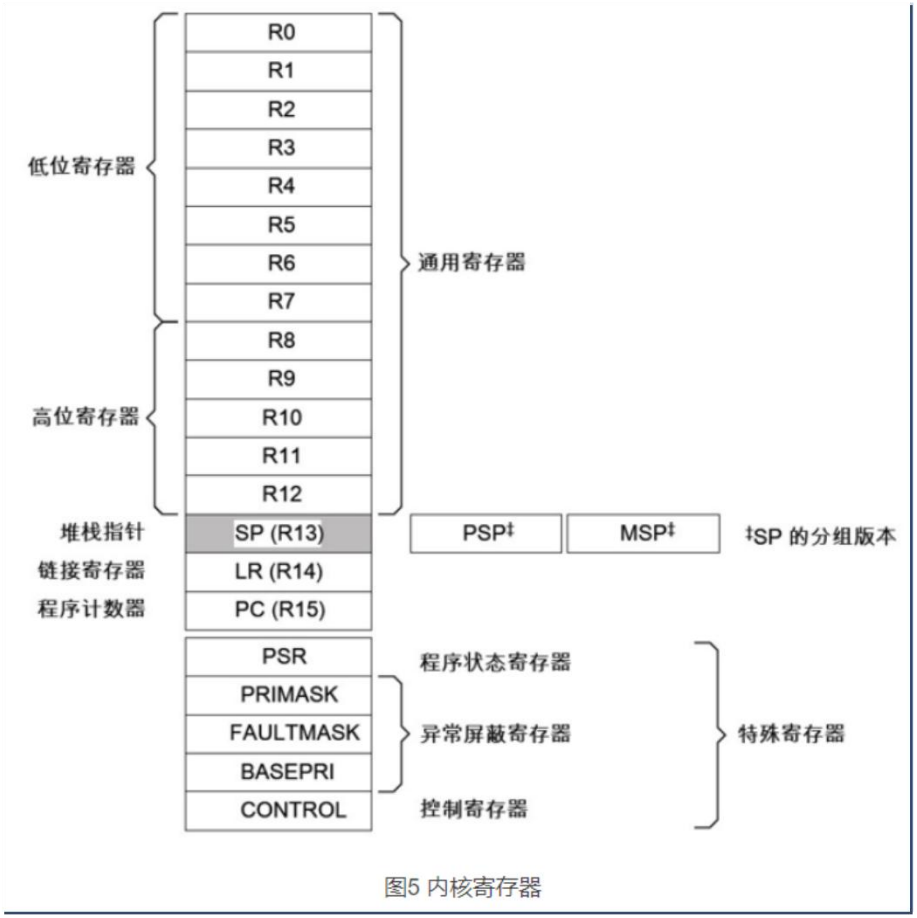


图5 内核寄存器

2 中断的响应流程

- 中断向量表

Figure 2-6. Vector Table

Exception number	IRQ number	Offset	Vector
(N+16)	(N)	0x040 + 0x(N*4)	IRQ N
.		.	.
.		.	.
.		.	.
		0x004C	
18	2	0x0048	IRQ2
17	1	0x0044	IRQ1
16	0	0x0040	IRQ0
15	-1	0x003C	Systick
14	-2	0x0038	PendSV
13			Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
0		0x0000	Initial SP value

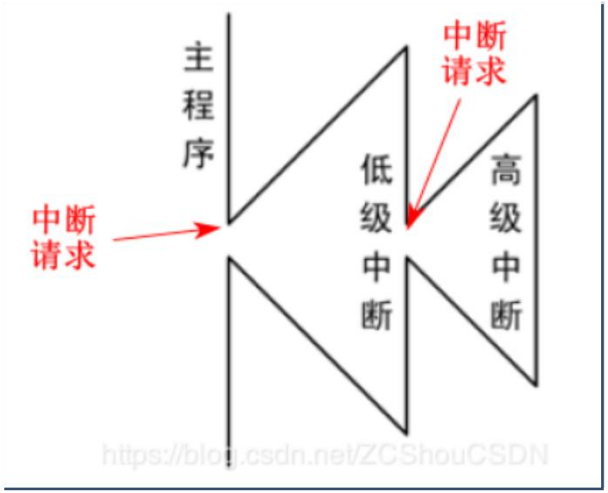
2 中断的响应流程

- 中断的响应流程
  - 执行异常处理

——执行相应的中断服务函数

——如果有更高优先级的异常产生，则接受新的异常，当前程序被抢占（异常嵌套）

——如果新的异常的优先级等于或者低于当前的异常，则等到当前异常处理完成后才会得到处理



2 中断的响应流程

- 中断的响应流程
  - 异常返回

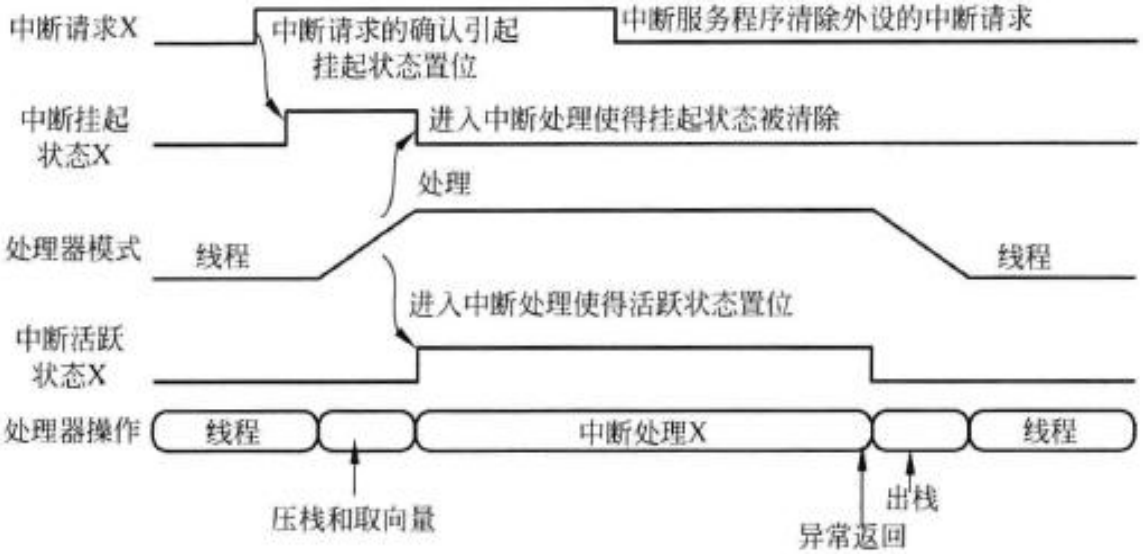
——CPU内核寄存器出栈

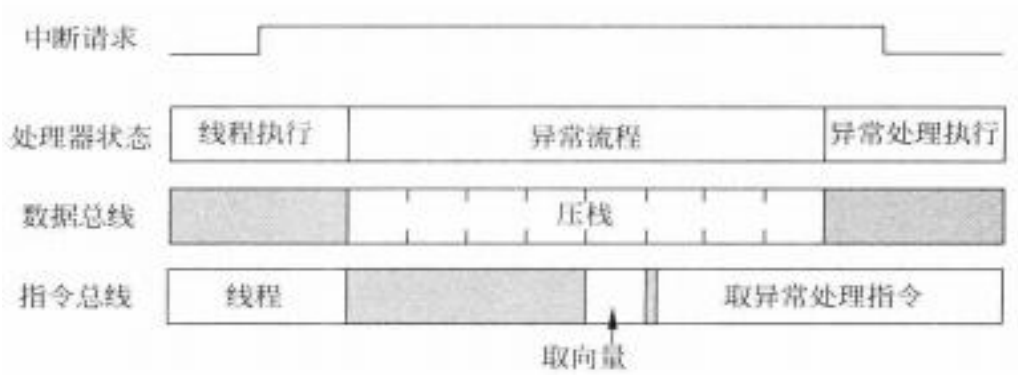
——NVIC寄存器更新

——CPU取出之前被中断的程序指令，程序恢复执行

2 中断的响应流程

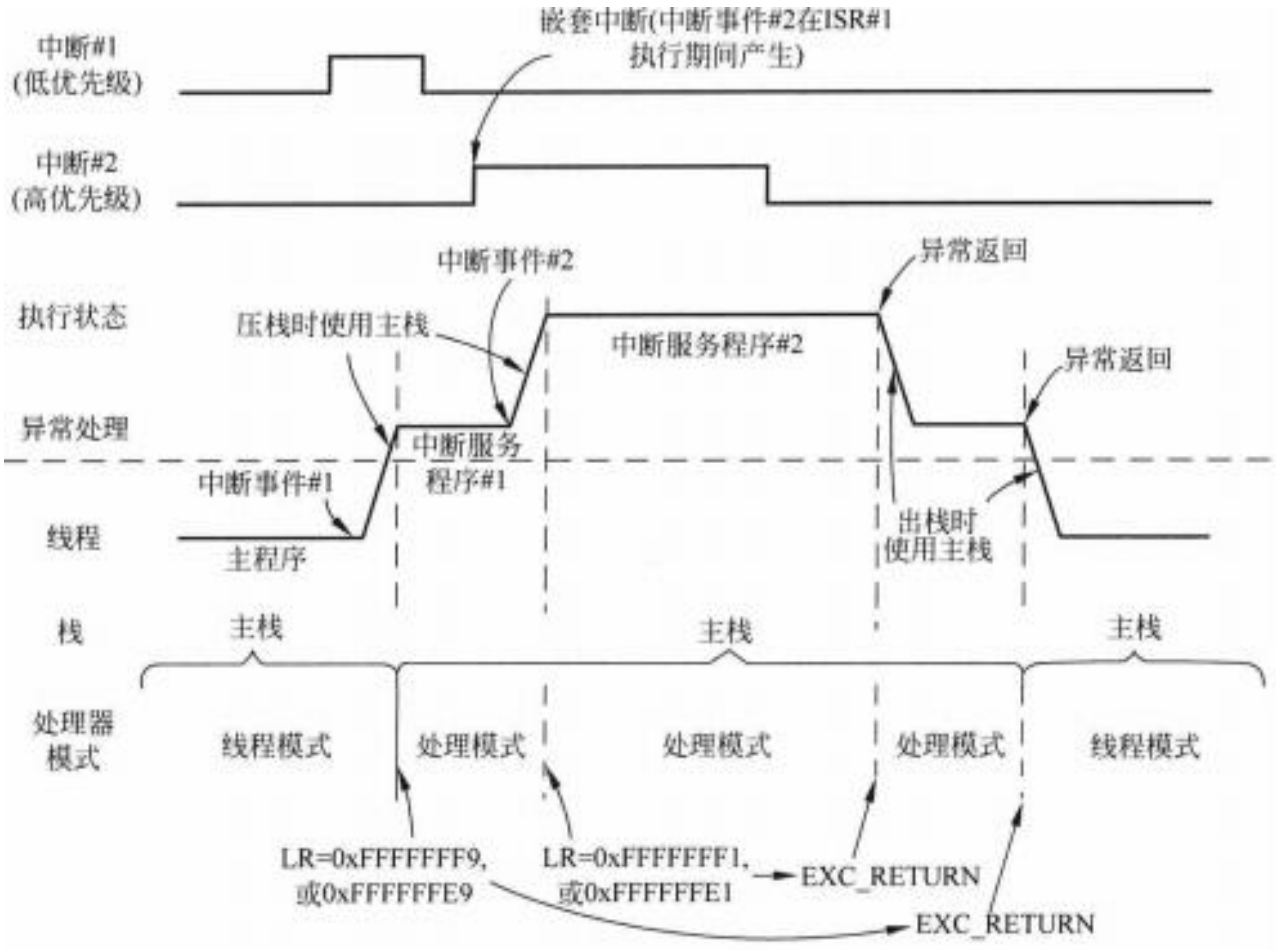
- 中断的响应流程





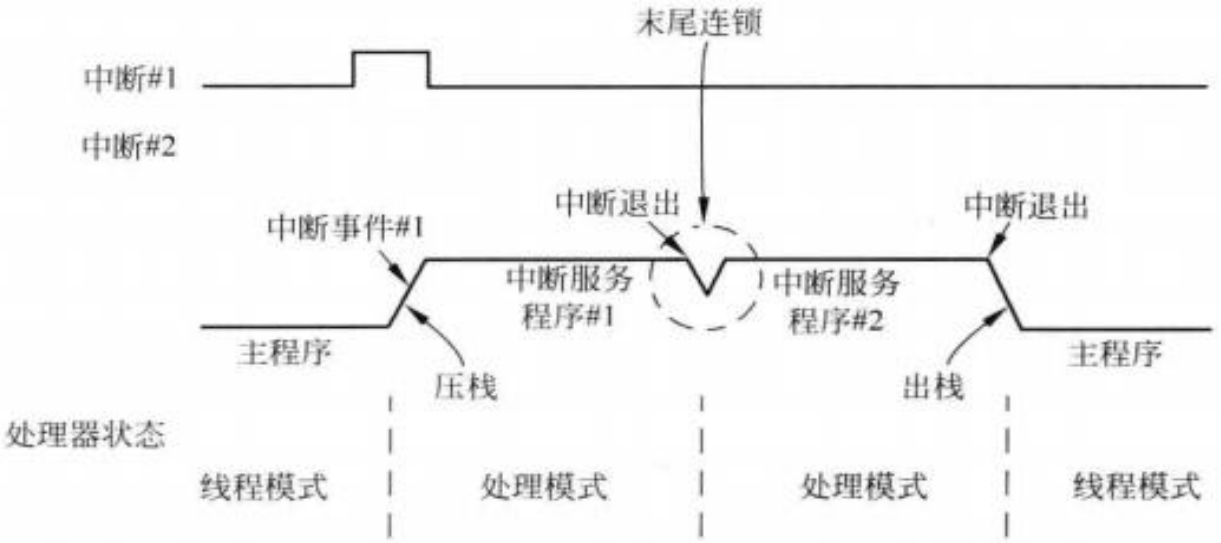
2 中断的响应流程

- 中断的响应流程
  - 嵌套中断：两个中断同时发生时，优先服务优先级高的中断



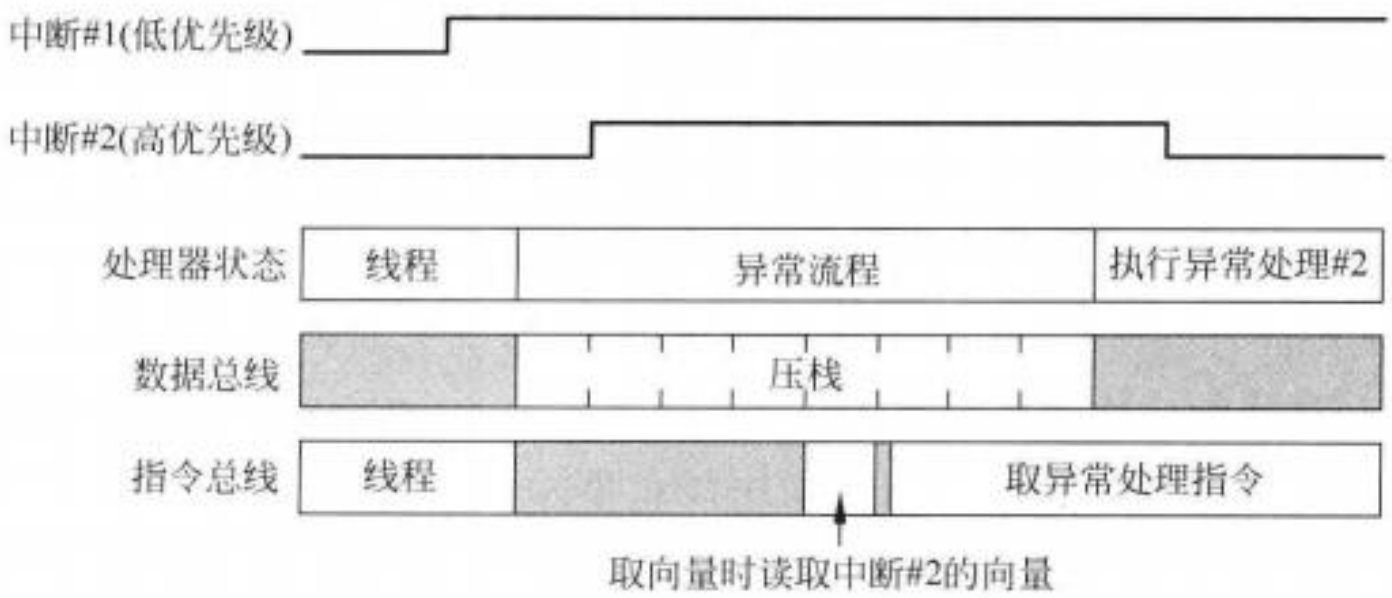
3 中断的响应流程

- 中断的响应流程
  - 尾链



2 中断的响应流程

- 中断的响应流程
  - 延迟到达



3 中断控制寄存器的访问

- CPU中断的控制
  - 总中断屏蔽寄存器（PRIMASK）（内核）

Priority Mask Register (PRIMASK)

Type RW, reset 0x0000.0000

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved															PRIMASK
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:1	reserved	RO	0x0000.000	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
0	PRIMASK	RW	0	Priority Mask Value Description 1 Prevents the activation of all exceptions with configurable priority. 0 No effect.

  - CPU寄存器组中的PRIMASK寄存器的最低位控制CPU是否接收中断请求。必须将PRIMASK设置为零，才能使CPU接收中断

3 中断控制寄存器的访问

- CPU中断的控制
  - 总中断屏蔽寄存器（PRIMASK）（内核）
  - PRIMASK是CPU寄存器组中的寄存器，在总线上没有地址，因此必须使用汇编整理对其操作：

在汇编代码中，CPSID　CPSIE 用于快速的开关中断。

CPSID	I	;PRIMASK=1,	;关中断
CPSIE	I	;PRIMASK=0,	;开中断
CPSID CPSIE	F F	;FAULTMASK=1, ;FAULTMASK=0	;关异常 ;开异常

3 中断控制寄存器的访问

- CPU中断的控制
  - 总中断屏蔽寄存器（PRIMASK）（内核）
  - 库函数提供了IntMasterEnable()函数开启CPU中断

IntMasterEnable

Enables the processor interrupt.

Prototype:

```
bool
IntMasterEnable(void)
```

Description:

This function allows the processor to respond to interrupts. This function does not affect the set of interrupts enabled in the interrupt controller; it just gates the single interrupt from the controller to the processor.

Example: Enable interrupts to the processor.

```
//
// Enable interrupts to the processor.
//
IntMasterEnable();
```

Returns:

Returns **true** if interrupts were disabled when the function was called or **false** if they were initially enabled.

3 中断控制寄存器的访问

- CPU中断的控制
  - 总中断屏蔽寄存器（PRIMASK）（内核）
  - 库函数提供了IntMasterEnable()函数开启CPU中断

TivaWare_C_Series-2.1.4.178\driverlib\interrupt.c

```
bool
IntMasterEnable(void)
{
    //
    // Enable processor interrupts.
    //
    return(CPUcpsie());
}
```

3 中断控制寄存器的访问

- CPU中断的控制
  - 总中断屏蔽寄存器（PRIMASK）（内核）
  - 库函数提供了IntMasterEnable()函数开启CPU中断

TivaWare_C_Series-2.1.4.178\driverlib\cpu.c

```
uint32_t
CPUcpsie(void)
{
    //
    // Read PRIMASK and enable interrupts.
    //
    __asm("    mrs     r0, PRIMASK\n"
          "    cpsie   i\n"
          "    bx      lr\n");//程序返回

    return(0);
}
```

3 中断控制寄存器的访问

- NVIC寄存器的控制
  - 中断使能寄存器（ENn）

Interrupt 0-31 Set Enable (EN0), offset 0x100

Interrupt 32-63 Set Enable (EN1), offset 0x104

Interrupt 64-95 Set Enable (EN2), offset 0x108

Interrupt 96-113 Set Enable (EN3), offset 0x10C

Interrupt 0-31 Set Enable (EN0)

Base 0xE000.E000
Offset 0x100
Type RW, reset 0x0000.0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Enable

Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, enables the interrupt.

A bit can only be cleared by setting the corresponding `INT[n]` bit in the **DISn** register.

3 中断控制寄存器的访问

- NVIC寄存器的控制
  - 中断禁止寄存器（DISn）

Register 8: Interrupt 0-31 Clear Enable (DIS0), offset 0x180
Register 9: Interrupt 32-63 Clear Enable (DIS1), offset 0x184
Register 10: Interrupt 64-95 Clear Enable (DIS2), offset 0x188
Register 11: Interrupt 96-113 Clear Enable (DIS3), offset 0x18C

Interrupt 0-31 Clear Enable (DIS0)

Base 0xE000.E000
Offset 0x180
Type RW, reset 0x0000.0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
INT															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:0	INT	RW	0x0000.0000	Interrupt Disable

Value	Description
0	On a read, indicates the interrupt is disabled. On a write, no effect.
1	On a read, indicates the interrupt is enabled. On a write, clears the corresponding `INT[n]` bit in the **EN0** register, disabling interrupt [n].

3 中断控制寄存器的访问

- NVIC寄存器的控制
  - 对ENn写1，开启对应的中断
  - 对DISn写1，关闭对应的中断
  - 库函数中，提供了IntEnable()和IntDisable()函数，来开启和关闭指定的中断和异常

```
IntEnable(INT_GPIOJ);//接收来自GIPOJ端口的中断请求
IntDisable(INT_UART0);//关闭来自UART0模块的中断请求
```

3 中断控制寄存器的访问

- IntEnable

IntEnable

Enables an interrupt.

Prototype:

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

Parameters:

ui32Interrupt specifies the interrupt to be enabled.

Description:

The specified interrupt is enabled in the interrupt controller. The *ui32Interrupt* parameter must be one of the valid **INT**_* values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. Other enables for the interrupt (such as at the peripheral level) are unaffected by this function.

Example: Enable the UART 0 interrupt.

```
//
// Enable the UART 0 interrupt in the interrupt controller.
//
IntEnable(INT_UART0);
```

Returns:

None.

3 中断控制寄存器的访问

- IntEnable TivaWare_C_Series-2.1.4.178\driverlib\interrupt.c

```
void IntEnable(uint32_t ui32Interrupt)
{
    // Check the arguments.
    ASSERT(ui32Interrupt < NUM_INTERRUPTS);
    // Determine the interrupt to enable.
    if(ui32Interrupt == FAULT_MPU)
    {
        // Enable the MemManage interrupt.
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_MEM;
    }
    else if(ui32Interrupt == FAULT_BUS)
    {
        // Enable the bus fault interrupt.
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_BUS;
    }
    else if(ui32Interrupt == FAULT_USAGE)
    {
        // Enable the usage fault interrupt.
        HWREG(NVIC_SYS_HND_CTRL) |= NVIC_SYS_HND_CTRL_USAGE;
    }
```

3 中断控制寄存器的访问

- IntEnable TivaWare_C_Series-2.1.4.178\driverlib\interrupt.c

```
else if(ui32Interrupt == FAULT_SYSTICK)
    {
        // Enable the System Tick interrupt.
        HWREG(NVIC_ST_CTRL) |= NVIC_ST_CTRL_INTEN;
    }
    else if(ui32Interrupt >= 16)
    {
        // Enable the general interrupt.
        HWREG(g_pui32EnRegs[(ui32Interrupt - 16) / 32]) =
            1 << ((ui32Interrupt - 16) & 31);
    }
}

static const uint32_t g_pui32EnRegs[] =
{
    NVIC_EN0, NVIC_EN1, NVIC_EN2, NVIC_EN3, NVIC_EN4
};
```

```
#define NVIC_EN0        0xE000E100  // Interrupt 0-31 Set Enable
#define NVIC_EN1        0xE000E104  // Interrupt 32-63 Set Enable
#define NVIC_EN2        0xE000E108  // Interrupt 64-95 Set Enable
#define NVIC_EN3        0xE000E10C  // Interrupt 96-127 Set Enable
```

3 中断控制寄存器的访问

- IntEnable　　TivaWare_C_Series-2.1.4.178\driverlib\interrupt.c
  - 输入参数ui32Interrupt为异常编号，事先在hw_ints.h文件中定义好

```
#define INT_GPIOA_TM4C129       16        // GPIO Port A
#define INT_GPIOB_TM4C129       17        // GPIO Port B
#define INT_GPIOC_TM4C129       18        // GPIO Port C
#define INT_GPIOD_TM4C129       19        // GPIO Port D
#define INT_GPIOE_TM4C129       20        // GPIO Port E
#define INT_GPIOJ_TM4C129       67        // GPIO Port J
#define INT_GPIOK_TM4C129       68        // GPIO Port K
#define INT_GPIOL_TM4C129       69        // GPIO Port L
```

  - 写 IntEnable(INT_GPIOJ)时，INT_GPIOJ会通过一定的宏定义，在编译时自动转换为INT_GPIOJ_TM4C129，即67。这种方式是为了适应多种类型的芯片。

3 中断控制寄存器的访问

- IntDisable
  - IntDisable函数与IntEnable的实现过程类似，只是将数组g_pui32EnRegs换成了g_pui32Dii16Regs

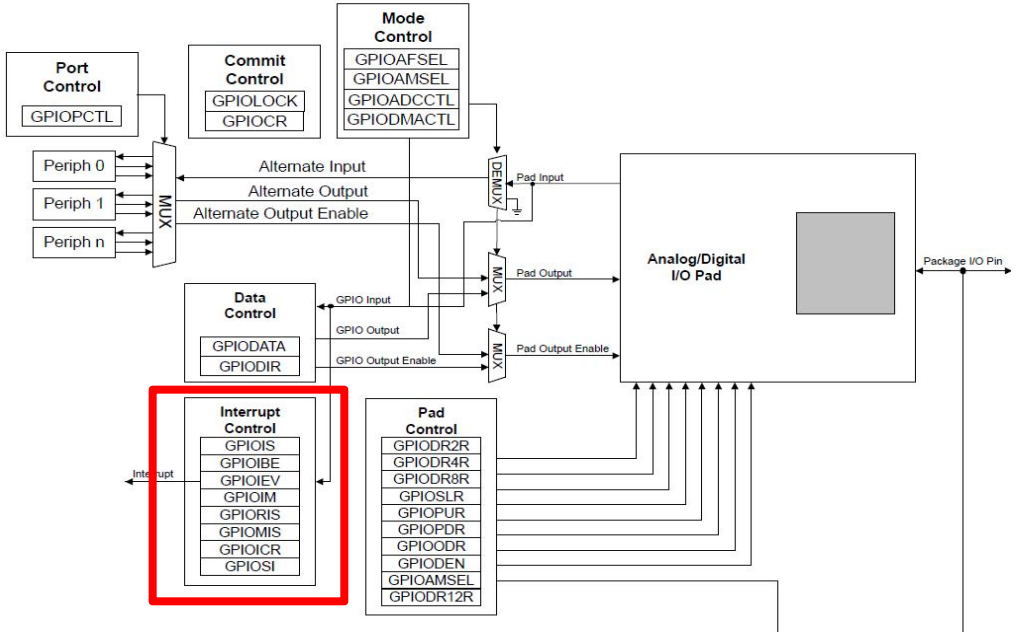
```
static const uint32_t g_pui32Dii16Regs[] =
{
    NVIC_DIS0, NVIC_DIS1, NVIC_DIS2, NVIC_DIS3, NVIC_DIS4
};
```

TivaWare_C_Series-2.1.4.178\inc\ hw_nvic.h

```
#define NVIC_DIS0               0xE000E180  // Interrupt 0-31 Clear Enable
#define NVIC_DIS1               0xE000E184  // Interrupt 32-63 Clear Enable
#define NVIC_DIS2               0xE000E188  // Interrupt 64-95 Clear Enable
#define NVIC_DIS3               0xE000E18C  // Interrupt 96-127 Clear Enable
```

4 GPIO的中断和操作

- GPIO的中断源管理
  - GPIO的中断，由一组寄存器管理
  - 设置中断的触发条件：电平触发（高电平、低电平）、边沿触发（上升沿、下降沿）
  - 中断的状态和设置：中断状态、中断使能和屏蔽，中断清除



4 GPIO的中断和操作

- GPIO的中断源管理
  - GPIO中断感知寄存器（GPIOIS）
    ——0边沿触发
    ——1电平触发

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IS	RW	0x00	GPIO Interrupt Sense

Value	Description
0	The edge on the corresponding pin is detected (edge-sensitive).
1	The level on the corresponding pin is detected (level-sensitive).

4 GPIO的中断和操作

- GPIO的中断源管理
  - GPIO中断边沿控制（GPIOIBE）
    - ——0：由GPIOIEV决定中断触发边沿
    - ——1：上升沿、下降沿都触发中断

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								IBE							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IBE	RW	0x00	GPIO Interrupt Both Edges Value Description 0 Interrupt generation is controlled by the **GPIO Interrupt Event (GPIOIEV)** register (see page 761). 1 Both edges on the corresponding pin trigger an interrupt.

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断事件寄存器（GPIOIEV）
    - ——0：下降沿或低电平触发
    - ——1：上升沿或高电平触发

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

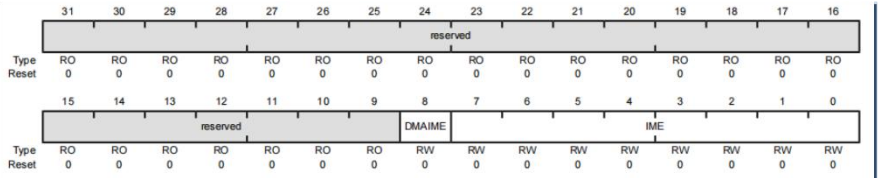
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved								IEV							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:8	reserved	RO	0x0000.00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
7:0	IEV	RW	0x00	GPIO Interrupt Event

Value	Description
0	A falling edge or a Low level on the corresponding pin triggers an interrupt.
1	A rising edge or a High level on the corresponding pin triggers an interrupt.

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断屏蔽寄存器（GPIOIM）
    - ——0：相应位的中断被屏蔽
    - ——1：相应位的中断使能
    - ——配置中断时，一定要先将中断屏蔽



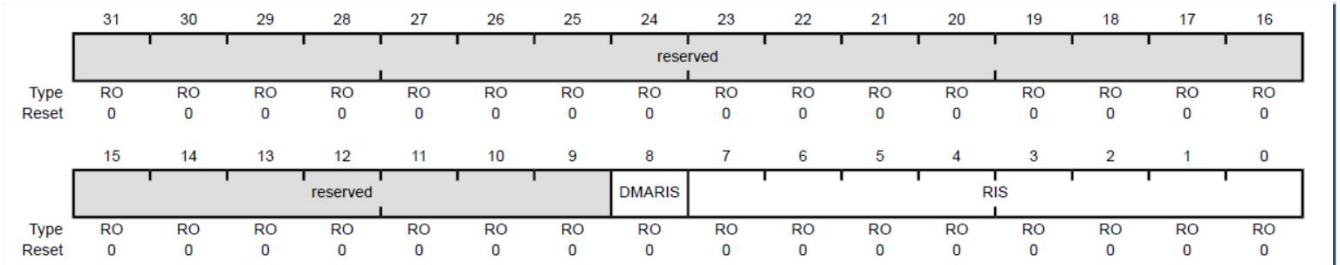
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved							DMAIME	IME							
Type	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:9	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
8	DMAIME	RW	0	GPIO uDMA Done Interrupt Mask Enable Value Description 0 The µDMA done interrupt is masked and does not cause an interrupt. 1 The µDMA done interrupt is not masked and can generate an interrupt to the interrupt controller.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable Value Description 0 The interrupt from the corresponding pin is masked. 1 The interrupt from the corresponding pin is sent to the interrupt controller.

4 GPIO的中断和操作

- GPIO的中断源管理
  - 原始中断状态寄存器（GPIORIS）
    - ——0：相应的位不满足触发条件
    - ——1：相应的位满足触发条件



Bit/Field	Name	Type	Reset	Description
7:0	RIS	RO	0x00	GPIO Interrupt Raw Status

Value	Description
0	An interrupt condition has not occurred on the corresponding pin.
1	An interrupt condition has occurred on the corresponding pin.

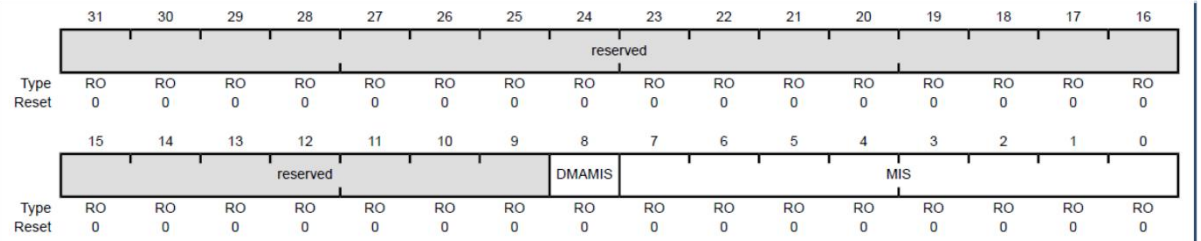
For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

- ➢ 如果是边沿触发的中断，必须通过GPIOICR寄存器清除中断
- ➢ 如果是电平触发的中断，中断条件消失后，相应的位自动清除

4 GPIO的中断和操作

- GPIO的中断源管理
  - 屏蔽中断状态寄存器（GPIOMIS）
    - ——0：相应的位不满足触发条件或满足条件但中断被屏蔽
    - ——1：相应的位满足触发条件且中断未被屏蔽



Bit/Field	Name	Type	Reset	Description
7:0	MIS	RO	0x00	GPIO Masked Interrupt Status

Value	Description
0	An interrupt condition on the corresponding pin is masked or has not occurred.
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.

For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断清除寄存器（ GPIOICR ）

    ——在相应位写1清除中断

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved							DMAIC	IC							
Type	RO	RO	RO	RO	RO	RO	RO	W1C	W1C	W1C	W1C	W1C	W1C	W1C	W1C	W1C
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断选择寄存器（GPIOSI）

    ——PORTP和PORTQ每个管脚都有相应的中断向量

```
IntDefaultHandler,                      // GPIO Port P (Summary or P0)
IntDefaultHandler,                      // GPIO Port P1
IntDefaultHandler,                      // GPIO Port P2
IntDefaultHandler,                      // GPIO Port P3
IntDefaultHandler,                      // GPIO Port P4
IntDefaultHandler,                      // GPIO Port P5
IntDefaultHandler,                      // GPIO Port P6
IntDefaultHandler,                      // GPIO Port P7
IntDefaultHandler,                      // GPIO Port Q (Summary or Q0)
IntDefaultHandler,                      // GPIO Port Q1
IntDefaultHandler,                      // GPIO Port Q2
IntDefaultHandler,                      // GPIO Port Q3
IntDefaultHandler,                      // GPIO Port Q4
IntDefaultHandler,                      // GPIO Port Q5
IntDefaultHandler,                      // GPIO Port Q6
IntDefaultHandler,                      // GPIO Port Q7
IntDefaultHandler,                      // GPIO Port R
IntDefaultHandler,                      // GPIO Port S
```

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断选择寄存器（GPIOSI）

——GPIOSI.SUM选择是为每个管脚发送中断信号，还是汇总后通过0号管脚发送

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved															SUM
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

0	SUM	RW	0	Summary Interrupt

Value	Description
0	All port pin interrupts are OR'ed together to produce a summary interrupt. **Note:** The OR'ed summary interrupt occurs on bit 0 of the **GPIORIS** register. For summary interrupt mode, software should set the **GPIOIM** register to 0xFF and mask the port pin interrupts 1 through 7 in the **Interrupt Clear Enable (DISn)** register (see "NVIC Register Descriptions" on page 153). When servicing this interrupt, write a 1 to the corresponding bit in the **UNPENDn** register to clear the pending interrupt in the NVIC and clear the **GPIORIS** register pin interrupt bits by setting the IC field of the **GPIOICR** register to 0xFF.
1	Each pin has its own interrupt vector.

4 GPIO的中断和操作

- 设置GPIO中断
  - 库函数提供了GPIOIntEnable和GPIOIntTypeSet使能中断并设置通断的触发条件

```
GPIOIntTypeSet(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0,GPIO_RISING_EDGE);

GPIOIntEnable(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0);
```

4 GPIO的中断和操作

- 设置GPIO中断

GPIOIntTypeSet

Sets the interrupt type for the specified pin(s).

Prototype:

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

为特定的引脚进行中断触发的设置，可以设置位电平触发或者边沿触发。

4 GPIO的中断和操作

- 设置GPIO中断

Parameters:

ui32Port is the base address of the GPIO port.
ui8Pins is the bit-packed representation of the pin(s).
ui32IntType specifies the type of interrupt trigger mechanism.

Description:

This function sets up the various interrupt trigger mechanisms for the specified pin(s) on the selected GPIO port.

One of the following flags can be used to define the *ui32IntType* parameter:

- **GPIO_FALLING_EDGE** sets detection to edge and trigger to falling
- **GPIO_RISING_EDGE** sets detection to edge and trigger to rising
- **GPIO_BOTH_EDGES** sets detection to both edges
- **GPIO_LOW_LEVEL** sets detection to low level
- **GPIO_HIGH_LEVEL** sets detection to high level

In addition to the above flags, the following flag can be OR'd in to the *ui32IntType* parameter:

- **GPIO_DISCRETE_INT** sets discrete interrupts for each pin on a GPIO port.

The **GPIO_DISCRETE_INT** is not available on all devices or all GPIO ports, consult the data sheet to ensure that the device and the GPIO port supports discrete interrupts.

The pin(s) are specified using a bit-packed byte, where each bit that is set identifies the pin to be accessed, and where bit 0 of the byte represents GPIO port pin 0, bit 1 represents GPIO port pin 1, and so on.

Note:

In order to avoid any spurious interrupts, the user must ensure that the GPIO inputs remain stable for the duration of this function.

Returns:

None.

4 GPIO的中断和操作

- GPIOIntTypeSet

```
void GPIOIntTypeSet(uint32_t ui32Port, uint8_t ui8Pins,
                    uint32_t ui32IntType)
{
    HWREG(ui32Port + GPIO_O_IBE) = ((ui32IntType & 1) ?
                                    (HWREG(ui32Port + GPIO_O_IBE) | ui8Pins) :
                                    (HWREG(ui32Port + GPIO_O_IBE) & ~(ui8Pins)));
    HWREG(ui32Port + GPIO_O_IS) = ((ui32IntType & 2) ?
                                   (HWREG(ui32Port + GPIO_O_IS) | ui8Pins) :
                                   (HWREG(ui32Port + GPIO_O_IS) & ~(ui8Pins)));
    HWREG(ui32Port + GPIO_O_IEV) = ((ui32IntType & 4) ?
                                    (HWREG(ui32Port + GPIO_O_IEV) | ui8Pins) :
                                    (HWREG(ui32Port + GPIO_O_IEV) & ~(ui8Pins)));

    HWREG(ui32Port + GPIO_O_SI) = ((ui32IntType & 0x10000) ?
                                   (HWREG(ui32Port + GPIO_O_SI) | 0x01) :
                                   (HWREG(ui32Port + GPIO_O_SI) & ~(0x01)));
}
```

GPIOIBE

0：由GPIOIEV决定中断触发边沿

1：上升沿、下降沿都触发中断

GPIOIS

0边沿触发

1电平触发

GPIOIEV

0：下降沿或低电平触发

1：上升沿或高电平触发

```
#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
#define GPIO_DISCRETE_INT 0x00010000  // Interrupt for individual pins
```

4 GPIO的中断和操作

- GPIOIntEnable

GPIOIntEnable

Enables the specified GPIO interrupts.

Prototype:

```
void
GPIOIntEnable(uint32_t ui32Port,
              uint32_t ui32IntFlags)
```

Parameters:

ui32Port is the base address of the GPIO port.
ui32IntFlags is the bit mask of the interrupt sources to enable.

Description:

This function enables the indicated GPIO interrupt sources. Only the sources that are enabled can be reflected to the processor interrupt; disabled sources have no effect on the processor.

The *ui32IntFlags* parameter is the logical OR of any of the following:

- **GPIO_INT_PIN_0** - interrupt due to activity on Pin 0.
- **GPIO_INT_PIN_1** - interrupt due to activity on Pin 1.
- **GPIO_INT_PIN_2** - interrupt due to activity on Pin 2.
- **GPIO_INT_PIN_3** - interrupt due to activity on Pin 3.
- **GPIO_INT_PIN_4** - interrupt due to activity on Pin 4.
- **GPIO_INT_PIN_5** - interrupt due to activity on Pin 5.
- **GPIO_INT_PIN_6** - interrupt due to activity on Pin 6.
- **GPIO_INT_PIN_7** - interrupt due to activity on Pin 7.
- **GPIO_INT_DMA** - interrupt due to DMA activity on this GPIO module.

4 GPIO的中断和操作

- GPIOIntEnable

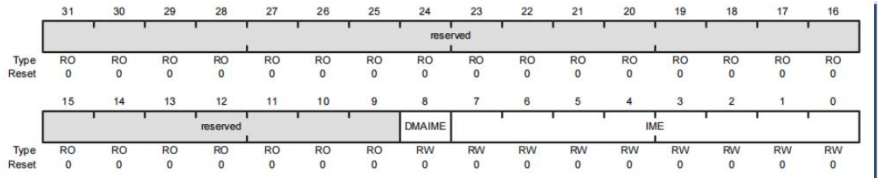
```
void
GPIOIntEnable(uint32_t ui32Port, uint32_t ui32IntFlags)
{
    // Check the arguments.
    ASSERT(_GPIOBaseValid(ui32Port));
    // Enable the interrupts.
    HWREG(ui32Port + GPIO_O_IM) |= ui32IntFlags;
}
```

输入参数在gpio.h中定义：

```
#define GPIO_INT_PIN_0          0x00000001
#define GPIO_INT_PIN_1          0x00000002
#define GPIO_INT_PIN_2          0x00000004
#define GPIO_INT_PIN_3          0x00000008
#define GPIO_INT_PIN_4          0x00000010
#define GPIO_INT_PIN_5          0x00000020
#define GPIO_INT_PIN_6          0x00000040
#define GPIO_INT_PIN_7          0x00000080
```

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断屏蔽寄存器（GPIOIM）
    - ——0：相应位的中断被屏蔽
    - ——1：相应位的中断使能
    - ——配置中断时，一定要先将中断屏蔽



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved							DMAIME	IME							
Type	RO	RO	RO	RO	RO	RO	RO	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:9	reserved	RO	0	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.
8	DMAIME	RW	0	GPIO uDMA Done Interrupt Mask Enable Value Description 0 The µDMA done interrupt is masked and does not cause an interrupt. 1 The µDMA done interrupt is not masked and can generate an interrupt to the interrupt controller.
7:0	IME	RW	0x00	GPIO Interrupt Mask Enable Value Description 0 The interrupt from the corresponding pin is masked. 1 The interrupt from the corresponding pin is sent to the interrupt controller.

4 GPIO的中断和操作

- GPIOIntClear
  - 库函数提供了GPIOIntClear函数，用来清除GPIO的中断

GPIOIntClear

Clears the specified interrupt sources.

Prototype:

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

Parameters:

ui32Port is the base address of the GPIO port.

ui32IntFlags is the bit mask of the interrupt sources to disable.

Description:

Clears the interrupt for the specified interrupt source(s).

The *ui32IntFlags* parameter is the logical OR of the **GPIO_INT_*** values.

Note:

Because there is a write buffer in the Cortex-M processor, it may take several clock cycles before the interrupt source is actually cleared. Therefore, it is recommended that the interrupt source be cleared early in the interrupt handler (as opposed to the very last action) to avoid returning from the interrupt handler before the interrupt source is actually cleared. Failure to do so may result in the interrupt handler being immediately reentered (because the interrupt controller still sees the interrupt source asserted).

Returns:

None.

4 GPIO的中断和操作

- GPIOIntClear

TivaWare_C_Series-2.1.4.178\driverlib\gpio.c

```
GPIOIntClear(uint32_t ui32Port, uint32_t ui32IntFlags)
{
    // Check the arguments.
    ASSERT(_GPIOBaseValid(ui32Port));
    // Clear the interrupts.
    HWREG(ui32Port + GPIO_O_ICR) = ui32IntFlags;
}
```

4 GPIO的中断和操作

- GPIO的中断源管理
  - 中断清除寄存器（ GPIOICR ）

    ——在相应位写1清除中断

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
	reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	reserved							DMAIC	IC							
Type	RO	RO	RO	RO	RO	RO	RO	W1C	W1C	W1C	W1C	W1C	W1C	W1C	W1C	W1C
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

7:0	IC	W1C	0x00	GPIO Interrupt Clear

Value	Description
0	The corresponding interrupt is unaffected.
1	The corresponding interrupt is cleared.

4 GPIO的中断和操作

- GPIOIntStatus
  - 库函数提供了GPIOIntStatus查看中断的具体来源

GPIOIntStatus

Gets interrupt status for the specified GPIO port.

Prototype:

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

Parameters:

ui32Port is the base address of the GPIO port.

bMasked specifies whether masked or raw interrupt status is returned.

Description:

If *bMasked* is set as **true**, then the masked interrupt status is returned; otherwise, the raw interrupt status is returned.

Returns:

Returns the current interrupt status for the specified GPIO module. The value returned is the logical OR of the **GPIO_INT_*** values that are currently active.

4 GPIO的中断和操作

- GPIOIntStatus
  - 库函数提供了GPIOIntStatus查看中断的具体来源

```
uint32_t
GPIOIntStatus(uint32_t ui32Port, bool bMasked)
{
    ASSERT(_GPIOBaseValid(ui32Port));
    // Return the interrupt status.
    if(bMasked)
    {
        return(HWREG(ui32Port + GPIO_O_MIS));
    }
    else
    {
        return(HWREG(ui32Port + GPIO_O_RIS));
    }
}
```

4 GPIO的中断和操作

- GPIO的中断源管理
  - 原始中断状态寄存器（GPIORIS）
    - ——0：相应的位不满足触发条件
    - ——1：相应的位满足触发条件

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved							DMARIS	RIS							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
7:0	RIS	RO	0x00	GPIO Interrupt Raw Status

Value	Description
0	An interrupt condition has not occurred on the corresponding pin.
1	An interrupt condition has occurred on the corresponding pin.

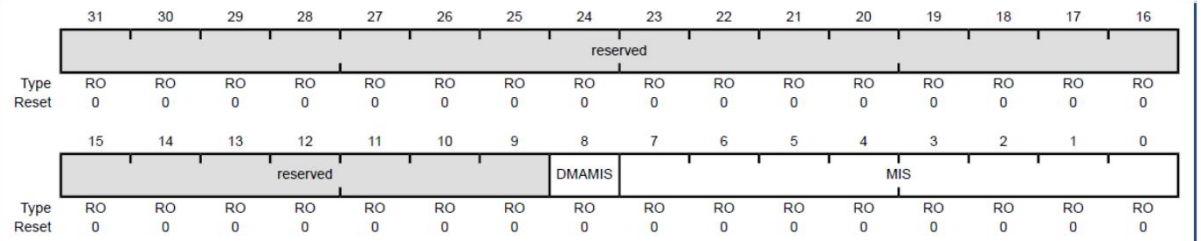
For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

- ➢ 如果是边沿触发的中断，必须通过GPIOICR寄存器清除中断
- ➢ 如果是电平触发的中断，中断条件消失后，相应的位自动清除

4 GPIO的中断和操作

- GPIO的中断源管理
  - 屏蔽中断状态寄存器（GPIOMIS）
    - ——0：相应的位不满足触发条件或满足条件但中断被屏蔽
    - ——1：相应的位满足触发条件且中断未被屏蔽



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
reserved															
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
reserved							DMAMIS	MIS							
Type	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
7:0	MIS	RO	0x00	GPIO Masked Interrupt Status

Value	Description
0	An interrupt condition on the corresponding pin is masked or has not occurred.
1	An interrupt condition on the corresponding pin has triggered an interrupt to the interrupt controller.

For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the **GPIOICR** register.

For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.

练习

以下为一段中断的初始化程序，每行代码分别影响了哪些寄存器？这行代码执行过程中，是怎样操作这些寄存器的？

- IntMasterEnable();
- IntEnable(INT_GPIOJ); // INT_GPIOJ=67

- GPIOIntTypeSet(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1,GPIO_RISING_EDGE);

- GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1);
- GPIOIntEnable(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1);

5 中断服务函数的编写

- 中断服务函数的编写
  - 手动更改中断向量表
  - 调用中断注册函数，动态添加中断服务函数

5 中断服务函数的编写

- 手动更改中断向量表
  - 中断向量表在startup_ccs.c文件中定义

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((uint32_t)&__STACK_TOP),
                                            // The initial stack pointer
    ResetISR,                               // The reset handler
    NmiSR,                                  // The NMI handler
    FaultISR,                               // The hard fault handler
    IntDefaultHandler,                      // The MPU fault handler
    IntDefaultHandler,                      // The bus fault handler
    IntDefaultHandler,                      // The usage fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    IntDefaultHandler,                      // SVCall handler
    IntDefaultHandler,                      // Debug monitor handler
    0,                                      // Reserved
    IntDefaultHandler,                      // The PendSV handler
    IntDefaultHandler,                      // The SysTick handler
    IntDefaultHandler,                              // GPIO Port A
    IntDefaultHandler,                              // GPIO Port B
    IntDefaultHandler,                              // GPIO Port C
    IntDefaultHandler,                      // GPIO Port D
    IntDefaultHandler,                      // GPIO Port E
    IntDefaultHandler,                      // UART0 Rx and Tx
    IntDefaultHandler,                      // UART1 Rx and Tx
    IntDefaultHandler,                      // SSI0 Rx and Tx
    IntDefaultHandler,                      // I2C0 Master and Slave
```

5 中断服务函数的编写

- 手动更改中断向量表
  - 中断向量表在startup_ccs.c文件中定义
  - 只要将表中的对应元素换为中断服务函数的地址，就可以使CPU在响应中断时，进入该中断服务函数
  - 默认的中断服务函数是IntDefaultHandler函数，即陷入死循环

```
static void
IntDefaultHandler(void)
{
    // Go into an infinite loop.
    while(1)
    {
    }
}
```

5 中断服务函数的编写

- 手动更改中断向量表
  - 中断向量表在startup_ccs.c文件中定义
  - 只要将表中的对应元素换为中断服务函数的地址，就可以使CPU在响应中断时，进入该中断服务函数

    修改GPIO Port J这一行，将IntDefaultHandler换为IntGPIOj函数。

    IntGPIOj, // GPIO Port J

    既然startup_ccs.c文件用到了IntGPIOj函数，需要在头部声明该函数

```
extern void IntGPIOj(void);
```

    IntGPIOj函数可以在其他文件中实现，比如在main.c中

  - 默认的中断服务函数是IntDefaultHandler函数，即陷入死循环

5 中断服务函数的编写

- 调用中断注册函数，动态添加中断服务函数
  - 中断向量表数组固化在flash中，在程序运行过程中无法修改
  - 如果需要在程序运行过程中动态添加或改变中断服务函数，需要中断向量表移动到RAM中
  - ARM处理器提供了移动中断向量表的特性，可以重新设置向量表在总线上的位置

5 中断服务函数的编写

- 调用中断注册函数，动态添加中断服务函数

Vector Table Offset (VTABLE)

Base 0xE000.E000
Offset 0xD08
Type RW, reset 0x0000.0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OFFSET															
Type	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW	RW
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OFFSET						reserved									
Type	RW	RW	RW	RW	RW	RW	RO	RO	RO	RO	RO	RO	RO	RO	RO	RO
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Bit/Field	Name	Type	Reset	Description
31:10	OFFSET	RW	0x000.00	Vector Table Offset When configuring the OFFSET field, the offset must be aligned to the number of exception entries in the vector table. Because there are 112 interrupts, the offset must be aligned on a 1024-byte boundary.
9:0	reserved	RO	0x00	Software should not rely on the value of a reserved bit. To provide compatibility with future products, the value of a reserved bit should be preserved across a read-modify-write operation.

5 中断服务函数的编写

- 调用中断注册函数，动态添加中断服务函数
  - 库函数中提供了GPIOIntRegister动态注册GPIO的中断服务函数

GPIOIntRegister

Registers an interrupt handler for a GPIO port.

Prototype:

```
void
GPIOIntRegister(uint32_t ui32Port,
                void (*pfnIntHandler)(void))
```

Parameters:

ui32Port is the base address of the GPIO port.

pfnIntHandler is a pointer to the GPIO port interrupt handling function.

Description:

This function ensures that the interrupt handler specified by *pfnIntHandler* is called when an interrupt is detected from the selected GPIO port. This function also enables the corresponding GPIO interrupt in the interrupt controller; individual pin interrupts and interrupt sources must be enabled with GPIOIntEnable().

See also:

IntRegister() for important information about registering interrupt handlers.

Returns:

None.

5 中断服务函数的编写

- 调用中断注册函数，动态添加中断服务函数
  - 库函数中提供了GPIOIntRegister动态注册GPIO的中断服务函数

TivaWare_C_Series-2.1.4.178\driverlib\gpio.c

```
void
GPIOIntRegister(uint32_t ui32Port, void (*pfnIntHandler)(void))
{
    uint32_t ui32Int;
    // Check the arguments.
    ASSERT(_GPIOBaseValid(ui32Port));
    // Get the interrupt number associated with the specified GPIO.
    ui32Int = _GPIOIntNumberGet(ui32Port);
    ASSERT(ui32Int != 0);
    // Register the interrupt handler.
    IntRegister(ui32Int, pfnIntHandler);
    // Enable the GPIO interrupt.
    IntEnable(ui32Int);
}
```

5 中断服务函数的编写

- IntRegister

IntRegister

Registers a function to be called when an interrupt occurs.

Prototype:

```
void
IntRegister(uint32_t ui32Interrupt,
            void (*pfnHandler)(void))
```

Parameters:

ui32Interrupt specifies the interrupt in question.

pfnHandler is a pointer to the function to be called.

Description:

This function is used to specify the handler function to be called when the given interrupt is asserted to the processor. The *ui32Interrupt* parameter must be one of the valid **INT_*** values listed in Peripheral Driver Library User's Guide and defined in the inc/hw_ints.h header file. When the interrupt occurs, if it is enabled (via IntEnable()), the handler function is called in interrupt context. Because the handler function can preempt other code, care must be taken to protect memory or peripherals that are accessed by the handler and other non-handler code.

Note:

The use of this function (directly or indirectly via a peripheral driver interrupt register function) moves the interrupt vector table from flash to SRAM. Therefore, care must be taken when linking the application to ensure that the SRAM vector table is located at the beginning of SRAM; otherwise the NVIC does not look in the correct portion of memory for the vector table (it requires the vector table be on a 1 kB memory alignment). Normally, the SRAM vector table is so placed via the use of linker scripts. See the discussion of compile-time versus run-time interrupt handler registration in the introduction to this chapter.

5 中断服务函数的编写

- IntRegister

```
IntRegister(uint32_t ui32Interrupt, void (*pfnHandler)(void))
{
    uint32_t ui32Idx, ui32Value;
    if(HWREG(NVIC_VTABLE) != (uint32_t)g_pfnRAMVectors)
    {
        // Copy the vector table from the beginning of FLASH to the RAM vector table.
        ui32Value = HWREG(NVIC_VTABLE);
        for(ui32Idx = 0; ui32Idx < NUM_INTERRUPTS; ui32Idx++)
        {
            g_pfnRAMVectors[ui32Idx] = (void (*)(void))HWREG((ui32Idx * 4) +
                                                    ui32Value);
        }
        // Point the NVIC at the RAM vector table.
        HWREG(NVIC_VTABLE) = (uint32_t)g_pfnRAMVectors;
    }
    // Save the interrupt handler.
    g_pfnRAMVectors[ui32Interrupt] = pfnHandler;
}
```

5 中断服务函数的编写

- IntRegister

#pragma DATA_ALIGN(g_pfnRAMVectors, 1024)
#pragma DATA_SECTION(g_pfnRAMVectors, ".vtable")
void (*g_pfnRAMVectors[NUM_INTERRUPTS])(void);
.vtable的存储空间在cmd文件中定义：

```
SECTIONS
{
    .intvecs:    > APP_BASE
    .text    :   > FLASH
    .const   :   > FLASH
    .cinit   :   > FLASH
    .pinit   :   > FLASH
    .init_array : > FLASH

    .vtable  :   > RAM_BASE
    .data    :   > SRAM
    .bss     :   > SRAM
    .sysmem  :   > SRAM
    .stack   :   > SRAM
}
```

可见.vtable指向RAM_BASE，即`#define RAM_BASE 0x20000000`

5 中断服务函数的编写

- IntRegister

```
GPIOIntRegister(GPIO_PORTJ_AHB_BASE,IntGPIOj);
```

将IntGPIOj注册成为GPIOJ的中断服务函数，并将向量表移动到SRAM中。

5 中断服务函数的编写

- 中断服务函数的编写
  - 进入中断服务函数后，首先要判断中断的具体来源
  - 不同的引脚的事件可能对应不同的逻辑
  - 调用GPIOIntStatus函数判断中断的具体来源
  - NVIC的中断标志会自动清除，但是GPIO中的中断标志要手动清除
  - GPIO模块中，由哪个引脚引发了中断，就清除哪个引脚的中断标志
  - 如果没有正确清除中断标识，程序将因为持续进入中断服务函数而死机

5 中断服务函数的编写

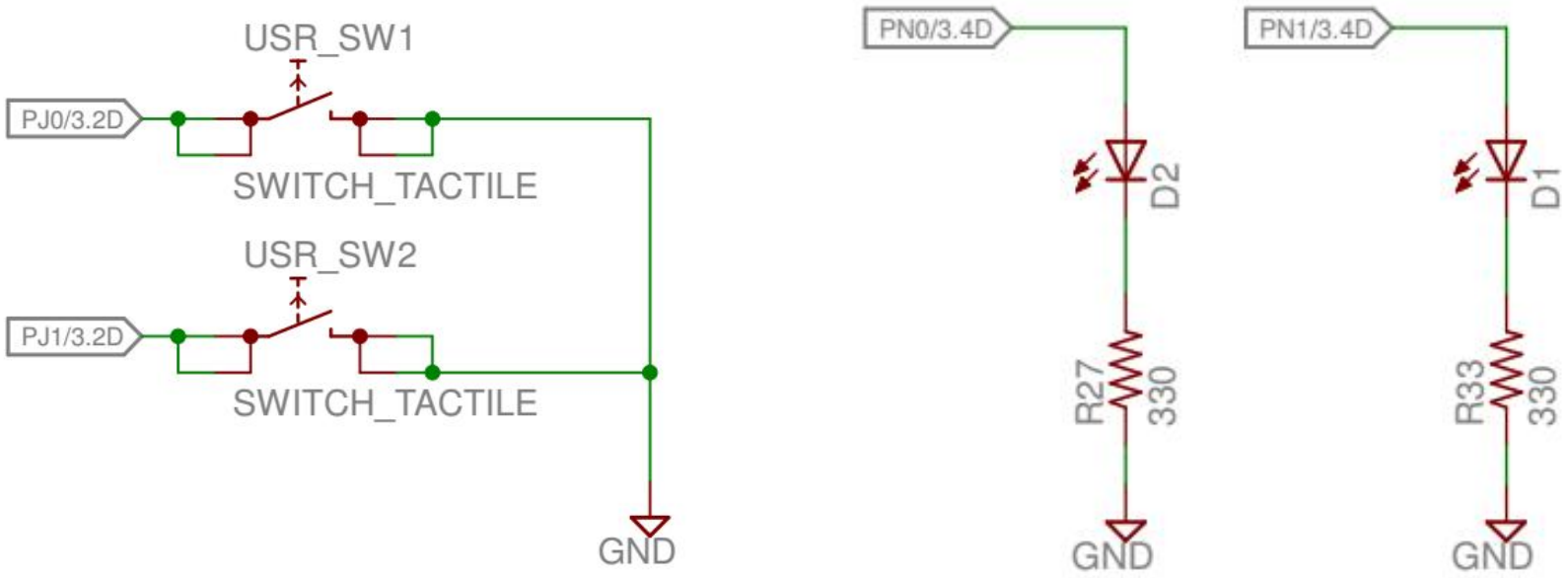
- 中断服务函数的编写

```
void
IntGPIOj(void)
{
        if(GPIOIntStatus(GPIO_PORTJ_AHB_BASE,true)==GPIO_INT_PIN_0){
                // do something
                GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0);
        }
        if(GPIOIntStatus(GPIO_PORTJ_AHB_BASE,true)==GPIO_INT_PIN_1){
                // do something
                GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1);
        }
}
```

- **练习：**
- 用PN0和PN1驱动开发板上的两个LED灯，PN0控制LED0，PN1控制LED1。PJ0和PJ1接两个简单按键。每按一次PJ0，LED0的状态就翻转一次（由灭变亮或者由亮变灭）。每按一次PJ1，LED1的状态就改变一次。
  - 画出这四个引脚的硬件连接图。
  - 使用中断的方式响应按键，编写以上用到的四个按键及其相关的初始化函数。
  - 编写完成所需功能的中断服务函数。

6 中断驱动的按键程序示例

- 中断驱动的简单按键程序
  - 硬件连接



6 中断驱动的按键程序示例

- 中断驱动的简单按键程序
  - 初始化

```
/引脚的输入输出配置
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOJ);
GPIOPinTypeGPIOInput(GPIO_PORTJ_AHB_BASE, GPIO_PIN_0|GPIO_PIN_1);


GPIOPadConfigSet(GPIO_PORTJ_AHB_BASE,GPIO_PIN_0,GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);

GPIOPadConfigSet(GPIO_PORTJ_AHB_BASE,GPIO_PIN_1,GPIO_STRENGTH_2MA,
GPIO_PIN_TYPE_STD_WPU);

GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
```

6 中断驱动的按键程序示例

- 中断驱动的简单按键程序
  - 初始化

```
//开启总中断和NVIC里的GPIOJ中断
IntMasterEnable();
IntEnable(INT_GPIOJ);

//配置引脚的中断、注册中断服务函数并开启GPIO模块的相关中断
GPIOIntTypeSet(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0,GPIO_FALLING_EDGE);

GPIOIntTypeSet(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1,GPIO_FALLING_EDGE);

GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0|GPIO_INT_PIN_1);

GPIOIntRegister(GPIO_PORTJ_AHB_BASE,IntGPIOj);

GPIOIntEnable(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0|GPIO_INT_PIN_1);
```

6 中断驱动的按键程序示例

- 中断驱动的简单按键程序
  - 中断服务函数

```
void IntGPIOj(void)
{     //首先判断是那个引脚产生的中断
      if(GPIOIntStatus(GPIO_PORTJ_AHB_BASE,true)==GPIO_INT_PIN_0){
          //延迟一段时间后，检查引脚是否还是低电平
          SysCtlDelay(g_ui32SysClock/30);
          //如果是低电平，则反转输出
          if(GPIOPinRead(GPIO_PORTJ_AHB_BASE,GPIO_PIN_0)==0){
                if(GPIOPinRead(GPIO_PORTN_BASE,GPIO_PIN_0)){
                GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_0,0);}
                else{
                GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_0,GPIO_PIN_0);}}
          //清除中断，来什么中断就清除什么中断，不能错
          GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_0);
       }
```

6 中断驱动的按键程序示例

- 中断驱动的简单按键程序
  - 中断服务函数

```
        if(GPIOIntStatus(GPIO_PORTJ_AHB_BASE,true)==GPIO_INT_PIN_1){
                // do something
            SysCtlDelay(g_ui32SysClock/30);
            if(GPIOPinRead(GPIO_PORTJ_AHB_BASE,GPIO_PIN_1)==0){
                if(GPIOPinRead(GPIO_PORTN_BASE,GPIO_PIN_1)){
                GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_1,0);}
                else{
                GPIOPinWrite(GPIO_PORTN_BASE,GPIO_PIN_1,GPIO_PIN_1);}
            }
            GPIOIntClear(GPIO_PORTJ_AHB_BASE,GPIO_INT_PIN_1);
        }
    }
```

- 练习
  - PN1，PN0，PF4、PF0分别控制LED1、LED2、LED3、LED4四个灯
  - 用矩阵键盘的1、2、3、4四个按键控制这四个灯。
  - 按一下按键1，LED1的状态翻转一次，按一下按键2，LED2的状态翻转一次，按一下按键3，LED3的状态翻转一次，按一下按键4，LED4的状态翻转一次

6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 硬件连接

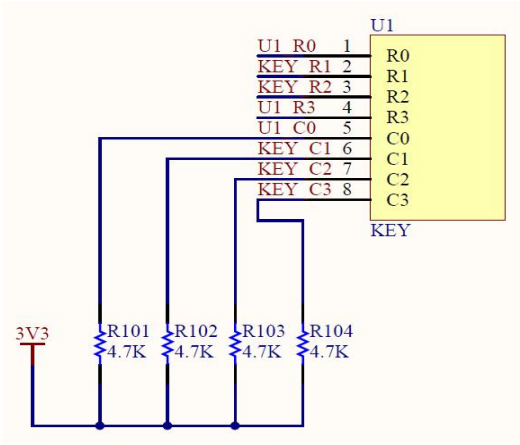
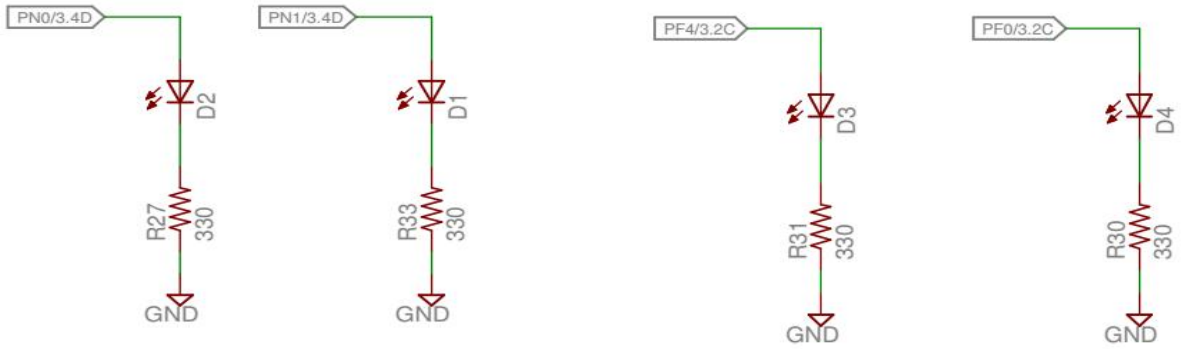


表 3 TM4C1294 与键盘连接引脚

Boosterpack 引脚序号	GPIO	键盘引脚
D1_6	PD1	ROW_0
A2_5	PD4	ROW_1
A2_6	PD5	ROW_2
B1_7	PD7	ROW_3
D1_10	PP2	column_0
D2_8	PP3	column_1
A2_8	PP4	column_2
D2_3	PP5	column_3



6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 初始化LED输出端口

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPION);

GPIOPinTypeGPIOOutput(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1);
GPIOPinWrite(GPIO_PORTN_BASE, GPIO_PIN_0|GPIO_PIN_1, 0x00);

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_0|GPIO_PIN_4);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0|GPIO_PIN_4, 0x00);
```

6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 初始化按键端口和GPIOP的中断

```
key_init();
KeyWrite(0xF0);
IntMasterEnable();
IntEnable(INT_GPIOP0);

GPIOIntTypeSet(GPIO_PORTP_BASE,GPIO_INT_PIN_2|GPIO_INT_PIN_3|GPIO_INT_PIN_4|GP
IO_INT_PIN_5,GPIO_FALLING_EDGE);

GPIOIntClear(GPIO_PORTP_BASE,GPIO_INT_PIN_2|GPIO_INT_PIN_3|GPIO_INT_PIN_4|GPIO
_INT_PIN_5);

GPIOIntRegister(GPIO_PORTP_BASE,IntGPIOP0);

GPIOIntEnable(GPIO_PORTP_BASE,GPIO_INT_PIN_2|GPIO_INT_PIN_3|GPIO_INT_PIN_4|GPI
O_INT_PIN_5);
```

6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 中断服务函数

```
void IntGPIOP0(void ){
uint32_t IntStatus=GPIOIntStatus(GPIO_PORTP_BASE,true);
   if(IntStatus&GPIO_INT_PIN_2|GPIO_INT_PIN_3|GPIO_INT_PIN_4|GPIO_INT_PIN_5){
        SysCtlDelay(g_ui32SysClock/30);
        if ((KeyRead() & 0xF0) < 0xF0)//是否有键按下
        {check_key(); // 调用check_key(),获取键值}
        ui8PinData=key_val;
        if(KeyQueueIndex<10){
            KeyQueueIndex++;
            KeyQueue[KeyQueueIndex]=ui8PinData;}
        KeyWrite(0xF0);
        GPIOIntClear(GPIO_PORTP_BASE,IntStatus);
        return;
   }
   GPIOIntClear(GPIO_PORTP_BASE,IntStatus);
}
```

6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 事务处理代码

```
while(1){
        int KeyTemp;
        if(KeyQueueIndex>=0){
            KeyTemp=KeyQueue[KeyQueueIndex];
            switch(KeyTemp){
            case 12: {// 1
                GPIOPinToggle(GPIO_PORTN_BASE,GPIO_PIN_1);break;}
            case 8: {// 2
                GPIOPinToggle(GPIO_PORTN_BASE,GPIO_PIN_0); break;}
            case 4: {// 3
                GPIOPinToggle(GPIO_PORTF_BASE,GPIO_PIN_4); break;}
            case 13: {// 4
                GPIOPinToggle(GPIO_PORTF_BASE,GPIO_PIN_0); break;}
            default: break;}
            KeyQueueIndex--;}
  }
```

6 中断驱动的按键程序示例

- 中断驱动的扫描按键程序
  - 事务处理代码

```
void GPIOPinToggle(uint32_t ui32Port, uint8_t ui8Pin){
    if(GPIOPinRead(ui32Port,ui8Pin)){
        GPIOPinWrite(ui32Port,ui8Pin,0);}
    else{
        GPIOPinWrite(ui32Port,ui8Pin,ui8Pin);
    }
}
```