中断与异常3

程晨闻
东南大学电气工程学院

➢ GPIO的中断源管理

- 电平触发（高电平、低电平）、边沿触发（上升沿、下降沿）
- 中断状态、中断使能和屏蔽，中断清除

➢ 设置GPIO中断和操作

- GPIOIntEnable
- GPIOIntTypeSet
- GPIOIntClear
- GPIOIntStatus

- 中断服务函数的编写
- 中断驱动的简单按键程序示例
- 中断驱动的扫描按键程序示例

➢ 中断服务函数的编写

– 手动更改中断向量表（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
```

➢ 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)
    {
    }
}
```

➢ 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

➢ 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

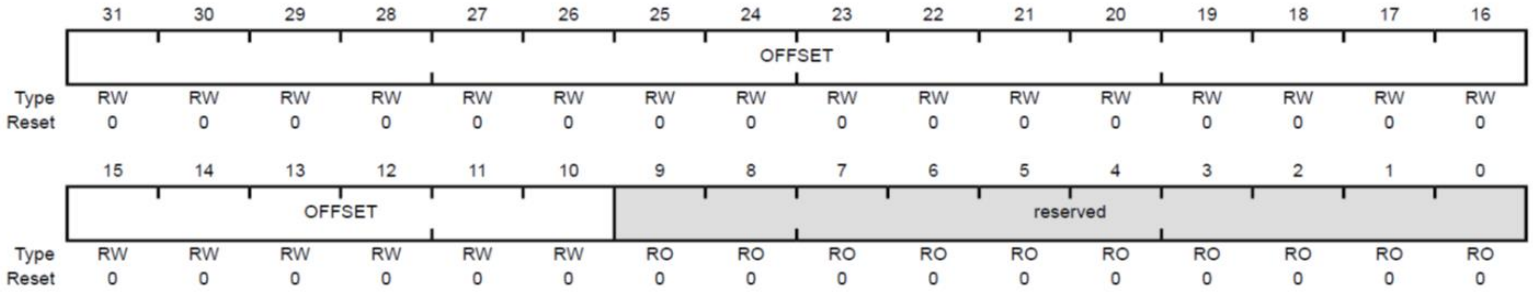
- 中断向量表数组固化在flash中，在程序运行过程中无法修改
- 如果需要在程序运行过程中动态添加或改变中断服务函数，需要中断向量表移动到RAM中
- ARM处理器提供了移动中断向量表的特性,可以重新设置向量表在总线上的位置

➢ 中断向量表偏移

- 原本中断向量表位于0x0000 0000（FLASH）
- 如果VTABLE设为0x2000 0000,则向量表就位于从0x2000 0000起始的空间中（RAM）
- 可在程序运行时任意修改向量表中的中断服务函数的地址

Vector Table Offset (VTABLE)

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



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.

➢ 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.

➢ GPIOIntRegister 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);
}
```

➢ IntRegister

- 根据中断向量号注册中断服务函数
- 将中断向量表移动到SRAM中

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.

➢ IntRegister

```
void
IntRegister(uint32_t ui32Interrupt, void (*pfnHandler)(void))
{
    uint32_t ui32Idx, ui32Value;
    // Check the arguments.
    ASSERT(ui32Interrupt < NUM_INTERRUPTS);
    // Make sure that the RAM vector table is correctly aligned.
    ASSERT(((uint32_t)g_pfnRAMVectors & 0x000003ff) == 0);
    // See if the RAM vector table has been initialized.
    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;
}
```

➢ 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`

➢ 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。

➢ 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

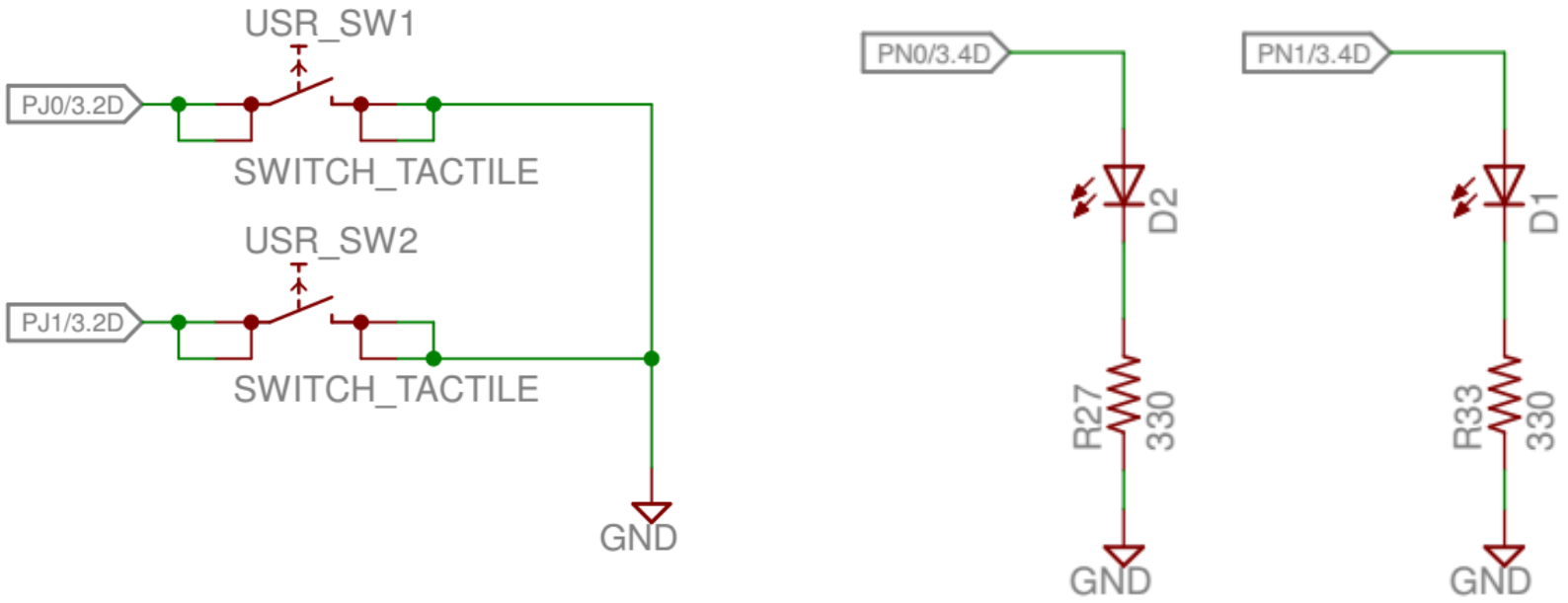
➢ 中断服务函数的编写

```
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);
        }
}
```

- 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。

➢ 中断驱动的简单按键程序

– 硬件连接



➢ 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);
```

➢ 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);
```

➢ 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);
        }
```

➢ 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);
}
}
```

➢ 作业：中断驱动的扫描按键程序

- 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

➢ 中断驱动的扫描按键程序

– 硬件连接

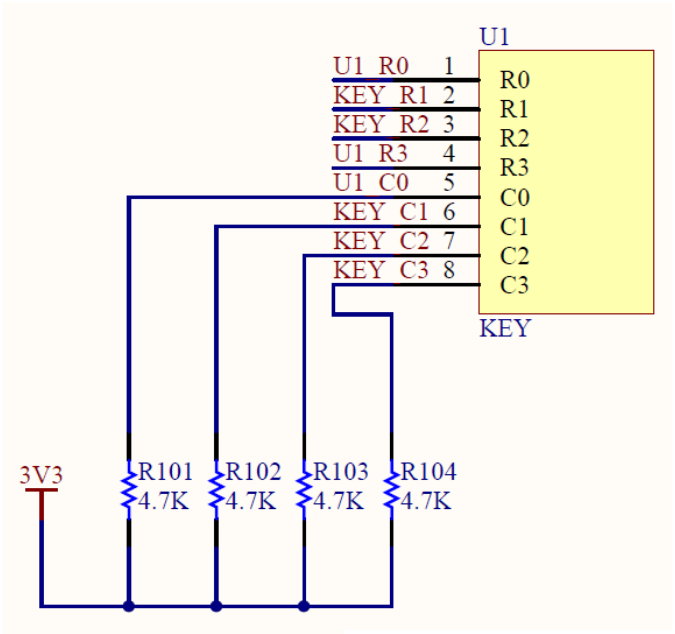
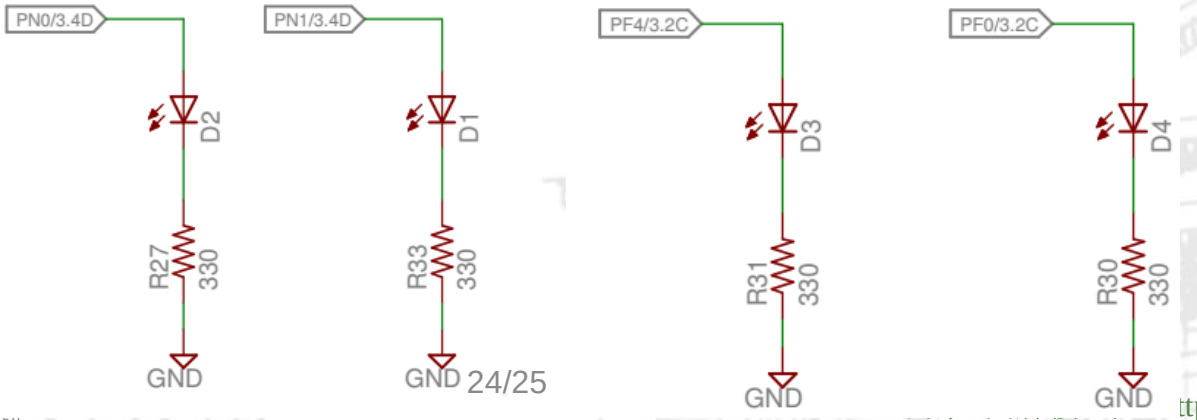


表 3 TM4C1294 与键盘连接引脚

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



谢谢！