

转

进入OS前的两步之PendSV(任务切换)

2016年10月01日 22:39:48 WuAiJiDian 阅读数 1584

先了解下如何使用PendSV异常。（为何要使用PendSV而不是其他的异常，请参考《cortex-M3权威指南》）

1，如何设定PendSV优先级？

表D.16 系统异常优先级寄存器 0xE000_ED18 - 0xE000_ED23

地址	名称	类型	复位值	描述
0xE000_ED18	PRI_4			存储器管理 fault 的优先级
0xE000_ED19	PRI_5			总线 fault 的优先级
0xE000_ED1A	PRI_6			用法 fault 的优先级
0xE000_ED1B	-	-	-	-
0xE000_ED1C	-	-	-	-
0xE000_ED1D	-	-	-	-
0xE000_ED1E	-	-	-	-
0xE000_ED1F	PRI_11			SVC 优先级
0xE000_ED20	PRI_12			调试监视器的优先级
0xE000_ED21	-	-	-	-
0xE000_ED22	PRI_14			PendSV 的优先级
0xE000_ED23	PRI_15			SysTick 的优先级

NVIC_SYSPRI14 EQU 0xE000ED22

NVIC_PENDSV_PRI EQU 0xFF

LDR R0, =NVIC_SYSPRI14 LDR R1, =NVIC_PENDSV_PRI

STRB R1, [R0]

2，如何触发PendSV异常？

表8.5 中断控制及状态寄存器ICSR (地址：0xE000_ED04)

位段	名称	类型	复位值	描述
31	NMIPENDSET	R/W	0	写 1 以悬起 NMI。因为 NMI 的优先级最高且从不掩蔽，在置位此位后将立即进入 NMI 服务例程。
28	PENDSVSET	R/W	0	写 1 以悬起 PendSV。读取它则返回 PendSV 的状态
27	PENDSVCLR	W	0	写 1 以清除 PendSV 悬起状态
26	PENDTSET	R/W	0	写 1 以悬起 SysTick。读取它则返回 PendSV 的状态

往ICSR第28位写1，即可将PendSV异常挂起。若是当前没有高优先级中断产生，那么程序将会进入PendSV handler

NVIC_INT_CTRL EQU 0xE000ED04

NVIC_PENDSVSET EQU 0x10000000

LDR R0, =NVIC_INT_CTRL

LDR R1, =NVIC_PENDSVSET

STR R1, [R0]

3，编写PendSV异常handler

这里用PendSV_Handler来触发LED点亮，以此证明PendSV异常触发的设置是正确的。



```
#include "stm32f10x_conf.h"

#define LED0 *((volatile unsigned long *) (0x422101a0)) //PA8

unsigned char flag=0;

void LEDInit(void)
```

```
{

RCC->APB2ENR|=1<<2;

GPIOA->CRH&=0xFFFFF0;

GPIOA->CRH|=0X0000003;

    GPIOA->ODR|=1<<8;

}

__asm void SetPendSVPro(void)

{

NVIC_SYSPRI14 EQU 0xE00ED22

NVIC_PENDSV_PRI EQU 0xFF


    LDR R1, =NVIC_PENDSV_PRI

    LDR R0, =NVIC_SYSPRI14

    STRB R1, [R0]

    BX LR

}

__asm void TriggerPendSV(void)

{

NVIC_INT_CTRL EQU 0xE00ED04

NVIC_PENDSVSET EQU 0x1000000


    LDR R0, =NVIC_INT_CTRL

    LDR R1, =NVIC_PENDSVSET

    STR R1, [R0]

    BX LR

}

int main(void)

{

    SetPendSVPro();

    LEDInit();

    TriggerPendSV();

    while(1);

}

void PendSV_Handler(void)

{
```



2



```
LED0=0;

}
```

复制代码

👍
2

💬

🔖

📱

<

>

上述代码可以正常点亮LED，说明PendSV异常是正常触发了。

OK，是时候挑战任务切换了。

如何实现任务切换？三个步骤：

步骤一：在进入中断前先设置PSP。

```
curr_task = 0;
```

设置任务0为当前任务

```
__set_PSP((PSP_array[curr_task] + 16*4));
```

设置PSP指向task0堆栈的栈顶位置

```
__set_CONTROL(0x3);
```

设置为用户级，并使用PSP堆栈。

```
__ISB();
```

指令同步隔离，暂不知道干啥用

步骤二：将当前寄存器的内容保存到当前任务堆栈中。进入ISR时，cortex-m3会自动保存八个寄存器到PSP中，剩下的几个需要我们手动保存。

步骤三：在Handler中将下一个任务的堆栈中的内容加载到寄存器中，并将PSP指向下一个任务的堆栈。这样就完成了任务切换。

要在PendSV 的ISR中完成这两个步骤，我们先需了解下在进入PendSV ISR时，cortex-M3做了什么？

1，入栈。会有8个寄存器自动入栈。入栈内容及顺序如下：

地址	寄存器	被保存的顺序
旧SP (N-0)	原先已压入的内容	-
(N-4)	xPSR	2
(N-8)	PC	1
(N-12)	LR	8
(N-16)	R12	7
(N-20)	R3	6
(N-24)	R2	5
(N-28)	R1	4
新SP (N-32)	R0	3

在步骤一中，我们已经设置了PSP，那这8个寄存器就会自动入栈到PSP所指地址处。

2，取向量。找到PendSV ISR的入口地址，这样就能跳到ISR了。，

3，更新寄存器内容。

做完这三步后，程序就进入ISR了。

进入ISR前，我们已经完成了步骤一，cortex-M3已经帮我们完成了步骤二的一部分，剩下的需要我们手动完成。

在ISR中添加代码如下：

```
MRS R0, PSP
```

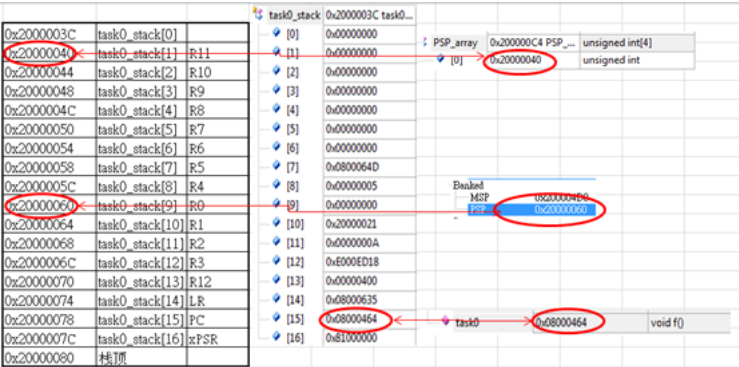
保存PSP到R0。为什么是PSP而不是MSP。因为在OS启动的时候，我们已经把SP设置为PSP了。这样使得用户程序使用任务堆栈，OS使用主堆栈，不会互相干扰。用户程序导致OS崩溃。

```
STMDB R0!, {R4-R11}
```

保存R4~R11到PSP中。C语言表达是*(--R0)={R4~R11}，R0中值先自减1，然后将R4~R11的值保存到该值所指向的地址中，即PSP中。

STMDB Rd!, {寄存器列表} 连续存储多个字到Rd中的地址值所指地址处。每次存储前，Rd先自减一次。

若是ISR是从task0进来，那么此时task0的堆栈中已经保存了该任务的寄存器参数。保存完成后，当前任务堆栈中的内容如下(假设是task0)



左边表格是预期值，右边是keil调试的实际值。可以看出，是一致的。在任务初始化时(步骤一)，我们将PSP指向任务0的栈顶0x20000080。在进入PendSV之前，入栈八个值，此时PSP指向了0x20000060。然后我们再保存R4~R11到0x20000040~0x2000005C。

这样很容易看明白，如果需要下次再切换到task0，只需恢复R4~R11，再将PSP指向0x20000060即可。

所以切换到另一个任务的代码：

```
LDR R1, =__cpp (&curr_task)

LDR R3, =__cpp (&PSP_array)

LDR R4, =__cpp (&next_task)

LDR R4, [R4]
```

获取下一个任务的编号

```
STR R4, [R1]
```

```
Curr_task=next_task
```

```
LDR R0, [R3, R4, LSL #2]
```

获得任务堆栈地址，若是task0，那么R0=0x20000040 (R0=R3+R4*4)

```
LDMIA R0!, {R4-R11}
```

恢复堆栈中的值到R4~R11。R4=*(R0++)。执行完后，R0中值变为0x20000060

LDMIA Rd! {寄存器列表} 先将Rd中值所指地址处的值送出寄存器中，Rd再自增1。

```
MSR PSP, R0
```

```
PSP=R0。
```

```
BX LR
```

中断返回

完整代码如下：

复制代码

```
#include "stm32f10x.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_rcc.h"
#include "stdio.h"
#include "misc.h"

#define HW32_REG(ADDRESS) (*(volatile unsigned long *) (ADDRESS))
```

```
#define LED0 *((volatile unsigned long *) (0x422101a0)) //PA8
void USART1_Init(void);
void task0(void);
unsigned char flag=1;

uint32_t curr_task=0;    // 当前执行任务
uint32_t next_task=1;    // 下一个任务
uint32_t task0_stack[17];
uint32_t task1_stack[17];
uint32_t PSP_array[4];

u8 task0_handle=1;
u8 task1_handle=1;

void task0(void)
{
    while(1)
    {
        if(task0_handle==1)
        {
            printf("task0\n");
            task0_handle=0;
            task1_handle=1;
        }
    }
}

void task1(void)
{
    while(1)
    {
        if(task1_handle==1)
        {
            printf("task1\n");
            task1_handle=0;
            task0_handle=1;
        }
    }
}

void LEDInit(void)
{
    RCC->APB2ENR|=1<<2;
    GPIOA->CRH&=0xFFFFFFF0;
    GPIOA->CRH|=0X00000003;
    GPIOA->ODR|=1<<8;
}

__asm void SetPendSVPro(void)
{
    NVIC_SYSPRI14 EQU 0xE00ED22
    NVIC_PENDSV_PRI EQU 0xFF

    LDR R1, =NVIC_PENDSV_PRI
    LDR R0, =NVIC_SYSPRI14
    STRB R1, [R0]
    BX LR
}

__asm void TriggerPendSV(void)
{
    NVIC_INT_CTRL EQU 0xE00ED04
    NVIC_PENDSVSET EQU 0x1000000

    LDR R0, =NVIC_INT_CTRL
```



2



```

LDR    R1, =NVIC_PENDSVSET
STR    R1, [R0]
BX     LR
}

int main(void)
{
    USART1_Init();

    SetPendSVPro();
    LEDInit();

    printf("OS test\n");

    PSP_array[0] = ((unsigned int) task0_stack) + (sizeof task0_stack) - 16*4;
    //PSP_array中存储的为task0_stack数组的尾地址-16*4, 即task0_stack[1023-16]地址
    HW32_REG((PSP_array[0] + (14<<2))) = (unsigned long) task0; /* PC */
    //task0的PC存储在task0_stack[1023-16]地址 +14<<2中, 即task0_stack[1022]中
    HW32_REG((PSP_array[0] + (15<<2))) = 0x01000000; /* xPSR */

    PSP_array[1] = ((unsigned int) task1_stack) + (sizeof task1_stack) - 16*4;
    HW32_REG((PSP_array[1] + (14<<2))) = (unsigned long) task1; /* PC */
    HW32_REG((PSP_array[1] + (15<<2))) = 0x01000000; /* xPSR */

    /* 任务0先执行 */
    curr_task = 0;

    /* 设置PSP指向任务0堆栈的栈顶 */
    __set_PSP((PSP_array[curr_task] + 16*4));

    SysTick_Config(9000000);
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //72/8=9MHZ
    /* 使用堆栈指针, 非特权级状态 */
    __set_CONTROL(0x3);

    /* 改变CONTROL后执行ISB (architectural recommendation) */
    __ISB();

    /* 启动任务0 */
    task0();
    //LED0=0;
    while(1);
}

__asm void PendSV_Handler(void)
{
    // 保存当前任务的寄存器内容
    MRS    R0, PSP          // 得到PSP R0 = PSP
    // xPSR, PC, LR, R12, R0-R3已自动保存
    STMDB  R0!, {R4-R11} // 保存R4-R11共8个寄存器得到当前任务堆栈

    // 加载下一个任务的内容
    LDR    R1, =__cpp (&curr_task)
    LDR    R3, =__cpp (&PSP_array)
    LDR    R4, =__cpp (&next_task)

```



2



```

LDR R4,[R4] //得到下一个任务的ID
STR R4,[R1] //设置 curr_task = next_task
LDR R0,[R3, R4, LSL #2] //从PSP_array中获取PSP的值
LDMIA R0!,{R4-R11} //将任务堆栈中的数值加载到R4-R11中
//ADDS R0, R0, #0x20
MSR PSP, R0 //设置PSP指向此任务
//ORR LR, LR, #0x04
BX LR //返回
//xPSR, PC, LR, R12, R0-R3会自动的恢复
ALIGN 4
}

void SysTick_Handler(void)
{
    flag=~flag;
    LED0=flag;
    if(curr_task==0)
        next_task=1;
    else
        next_task=0;
    TriggerPendSV();
}

void USART1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    USART_InitTypeDef USART_InitStructure;

    /* config USART1 clock */
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);

    /* USART1 GPIO config */
    /* Configure USART1 Tx (PA.09) as alternate function push-pull */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);
    /* Configure USART1 Rx (PA.10) as input floating */
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /* USART1 mode config */
    USART_InitStructure.USART_BaudRate = 9600;
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;
    USART_InitStructure.USART_StopBits = USART_StopBits_1;
    USART_InitStructure.USART_Parity = USART_Parity_No ;
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
    USART_Init(USART1, &USART_InitStructure);
    USART_Cmd(USART1, ENABLE);
}

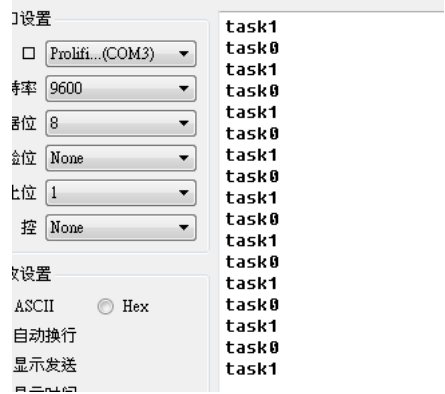
int fputc(int ch, FILE *f)
{
    USART_SendData(USART1, (unsigned char) ch);
    while (!(USART1->SR & USART_FLAG_TXE));

    return (ch);
}

```



测试后结果如图：



可以看出，两个任务可以切换了。

上述代码参考《cortex-M3权威指南》和《安富莱_STM32-V5开发板_μCOS-III教程》得来。

2

WuAiJiDian

关注

原创1

粉丝194

喜欢105

评论27

等级: 博客

访问: 10万+

积分: 804

排名: 8万+

最新文章

什么是Data Consistency?

使用STM32测量频率和占空比的几种方法

面向初学者的XCP——XCP协议的通信的构造和功能

基于XCP协议分析和理解

利用反电动势的过零点来测转子位置在讨论无转子位置

个人分类

生活感悟1篇

C语言11篇

IDE工具使用1篇

单片机13篇

芯片外设1篇

展开

归档

2019年5月1篇

2018年7月3篇

2018年3月4篇

2018年1月1篇

2017年12月2篇

展开

热门文章

永磁同步电机(PMSM)的FOC闭环控制详解

阅读数 51138

浅谈上、下拉电阻的作用

阅读数 9074

基于UDS的汽车通信故障诊断机制与处理策略

阅读数 8307

SVPWM算法原理及详解

阅读数 5298

利用反电动势的过零点来测转子位置在讨论无转子位置

阅读数 4750

最新评论

永磁同步电机(PMSM)的FOC闭...

sy243772901：转载别人文章的时候，最起码的源地址都不贴出来，这根本不尊重原创作者

SVPWM算法原理及详解

qq_39840900：学习学习

基于UDS的汽车通信故障诊断机制与...

mbtmxk：深度好文，学习了 除了一些错别字，可以称得上完美

永磁同步电机(PMSM)的FOC闭...

weixin_42530385：貌似图十一有错，相电压和线电压弄反了

永磁同步电机(PMSM)的FOC闭...

gongyuan073：剽窃狗 盗别人文章连原文链接都不放 司马货 原文链接：https://blog.csdn.ne' ...



程序人生



CSDN资讯

 QQ客服

 kefu@csdn.net

 客服论坛

 400-660-0108

工作时间 8:30-22:00

关于我们

招聘

广告服务

网站地图

 百度提供站内搜索 京ICP备19004658号

京公网安备11010502030143

©1999-2019 北京创新乐知网络技术有限公司

网络110报警服务


经营性网站备案信息

北京互联网违法和不良信息举报中心

中国互联网举报中心

家长监护

版权申诉

2





