昵称：sky1991
园龄：4年8个月
粉丝：28
关注：0
+加关注

< 2017年3月 >

日	一	二	三	四	五	六
26	27	28	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签

最新随笔
1. 安卓使用WIFI调试程序
2. 一步步写STM32 OS【四】OS基本框架
3. 一步步写STM32 OS【三】PendSV与堆栈操作
4. 一步步写STM32 OS【二】环境搭建
5. 一步步写STM32 OS【一】 序言
6. [转]关于WM_NCHITTEST消息
7. [转]ARM QT实现多点触摸
8. W3C SVG标准Web颜色值列表
9. [转]SYSZUXpinyin中文输入法的移植（到QT）

一步步写STM32 OS【三】PendSV与堆栈操作

一、什么是PendSV

PendSV是可悬起异常，如果我们把它配置最低优先级，那么如果同时有多个异常被触发，它会在其他异常执行完毕后再执行，而且任何异常都可以中断它。更详细的内容在《Cortex-M3 权威指南》里有介绍，下面我摘抄了一段。

OS 可以利用它“缓期执行”一个异常——直到其它重要的任务完成后才执行动 作。悬起 PendSV 的方法是：手工往 NVIC的 PendSV悬起寄存器中写 1。悬起后，如果优先级不够 高，则将缓期等待执行。

PendSV的典型使用场合是在上下文切换时（在不同任务之间切换）。例如，一个系统中有两个就绪的任务，上下文切换被触发的场合可以是：
1、执行一个系统调用
2、系统滴答定时器（SYSTICK）中断，（轮转调度中需要）

让我们举个简单的例子来辅助理解。假设有这么一个系统，里面有两个就绪的任务，并且通过SysTick异常启动上下文切换。但若在产生 SysTick 异常时正在响应一个中断，则 SysTick异常会抢占其 ISR。在这种情况下，OS是不能执行上下文切换的，否则将使中断请求被延迟，而且在真实系统中延迟时间还往往不可预知——任何有一丁点实时要求的系统都决不能容忍这 种事。因此，在 CM3 中也是严禁没商量——如果 OS 在某中断活跃时尝试切入线程模式，将触犯用法fault异常。

为解决此问题，早期的 OS 大多会检测当前是否有中断在活跃中，只有在无任何中断需要响应 时，才执行上下文切换（切换期间无法响应中断）。然而，这种方法的弊端在于，它可以把任务切 换动作拖延很久（因为如果抢占了 IRQ，则本次 SysTick在执行后不得作上下文切换，只能等待下 一次SysTick异常），尤其是当某中断源的频率和SysTick异常的频率比较接近时，会发生“共振”， 使上下文切换迟迟不能进行。现在好了，PendSV来完美解决这个问题了。PendSV异常会自动延迟上下文切换的请求，直到 其它的 ISR都完成了处理后才放行。为实现这个机制，需要把 PendSV编程为最低优先级的异常。如果 OS检测到某 IRQ正在活动并且被 SysTick抢占，它将悬起一个 PendSV异常，以便缓期执行 上下文切换。

使用 PendSV 控制上下文切换个中事件的流水账记录如下：

1. 任务 A呼叫 SVC来请求任务切换（例如，等待某些工作完成）

2. OS接收到请求，做好上下文切换的准备，并且悬起一个 PendSV异常。

3. 当 CPU退出 SVC后，它立即进入 PendSV，从而执行上下文切换。

4. 当 PendSV执行完毕后，将返回到任务 B，同时进入线程模式。

5. 发生了一个中断，并且中断服务程序开始执行

6. 在 ISR执行过程中，发生 SysTick异常，并且抢占了该 ISR。

7. OS执行必要的操作，然后悬起 PendSV异常以作好上下文切换的准备。

8. 当 SysTick退出后，回到先前被抢占的 ISR中，ISR继续执行

9. ISR执行完毕并退出后，PendSV服务例程开始执行，并且在里面执行上下文切换

10. 当 PendSV执行完毕后，回到任务 A，同时系统再次进入线程模式。

我们在uCOS的PendSV的处理代码中可以看到：

```
OS_CPU_PendSVHandler
    CPSID I ; 关中断
    ;保存上文
    ;.....................
    ;切换下文
    CPSIE I ;开中断
    BX LR ;异常返回
```

10. [转]交叉编译tslib1.4

我的标签

stm32(5)

OS(4)

MSP430(3)

调试(2)

串口(1)

按键消抖(1)

NPlot(1)

3d(1)

51(1)

ati(1)

更多

随笔分类(81)

Arm(11)

C#、QT(10)

FPGA(4)

Java(1)

Linux(3)

单片机(48)

一步步写STM32 OS(4)

随笔档案(84)

2014年3月 (1)

2013年11月 (3)

2013年10月 (1)

2013年8月 (1)

2013年6月 (1)

2013年5月 (9)

2013年4月 (5)

2013年2月 (3)

2012年11月 (5)

它在异常一开始就关闭了中端，结束时开启中断，中间的代码为临界区代码，即不可被中断的操作。PendSV异常是任务切换的堆栈部分的核心，由他来完成上下文切换。PendSV的操作也很简单，主要有设置优先级和触发异常两部分：

```
NVIC_INT_CTRL EQU 0xE000ED04 ; 中断控制寄存器
NVIC_SYSPRI14 EQU 0xE000ED22 ; 系统优先级寄存器(优先级14).
NVIC_PENDSV_PRI EQU 0xFF ; PendSV优先级(最低).
NVIC_PENDSVSET EQU 0x10000000 ; PendSV触发值

; 设置PendSV的异常中断优先级

LDR R0, =NVIC_SYSPRI14
LDR R1, =NVIC_PENDSV_PRI
STRB R1, [R0] ; 触发PendSV异常
LDR R0, =NVIC_INT_CTRL
LDR R1, =NVIC_PENDSVSET
STR R1, [R0]
```

二、堆栈操作

Cortex M4有两个堆栈寄存器，主堆栈指针（MSP）与进程堆栈指针（PSP），而且任一时刻只能使用其中的一个。MSP为复位后缺省使用的堆栈指针，异常永远使用MSP，如果手动开启PSP，那么线程使用PSP，否则也使用MSP。怎么开启PSP？

```
MSR     PSP, R0                                 ; Load PSP with new process
SP
    ORR     LR, LR, #0x04                       ; Ensure exception return
uses process stack
```

很容易就看出来了，置LR的位2为1，那么异常返回后，线程使用PSP。

写OS首先要将内存分配搞明白，单片机内存本来就很小，所以我们当然要斤斤计较一下。在OS运行之前，我们首先要初始化MSP和PSP，OS_CPU_ExceptStkBase是外部变量，假如我们给主堆栈分配1KB（256*4）的内存即OS_CPU_ExceptStk[256],则OS_CPU_ExceptStkBase=&OS_CPU_ExceptStk[256-1]。

```
EXTERN  OS_CPU_ExceptStkBase
   ;PSP清零，作为首次上下文切换的标志
   MOVS     R0, #0
   MSR      PSP, R0
   ;将MSP设为我们为其分配的内存地址
   LDR      R0, =OS_CPU_ExceptStkBase
   LDR      R1, [R0]
   MSR      MSP, R1
```

然后就是PendSV上下文切换中的堆栈操作了，如果不使用FPU，则进入异常自动压栈xPSR，PC，LR，R12，R0-R3，我们还要把R4-R11入栈。如果开启了FPU，自动压栈的寄存器还有S0-S15，还需吧S16-S31压栈。

```
MRS     R0, PSP
    SUBS    R0, R0, #0x20        ;压入R4-R11
    STM     R0, {R4-R11}

    LDR     R1, =Cur_TCB_Point    ;当前任务的指针
    LDR     R1, [R1]
    STR     R0, [R1]             ; 更新任务堆栈指针
```

出栈类似，但要注意顺序

2012年10月 (5)
2012年9月 (4)
2012年8月 (27)
2012年7月 (13)
2012年6月 (6)


```
LDR     R1, =TCB_Point     ;要切换的任务指针
    LDR     R2, [R1]
    LDR     R0, [R2]           ; R0为要切换的任务堆栈地址

    LDM     R0, {R4-R11}      ; 弹出R4-R11
    ADDS    R0, R0, #0x20

    MSR     PSP, R0         ;更新PSP
```

三、OS实战

新建os_port.asm文件，内容如下：

```
NVIC_INT_CTRL   EQU     0xE000ED04                              ; Interrupt control
state register.
NVIC_SYSPRI14   EQU     0xE000ED22                              ; System priority
register (priority 14).
NVIC_PENDSV_PRI EQU           0xFF                              ; PendSV priority value
(lowest).
NVIC_PENDSVSET  EQU     0x10000000                              ; Value to trigger
PendSV exception.

  RSEG CODE:CODE:NOROOT(2)
  THUMB


  EXTERN  g_OS_CPU_ExceptStkBase

  EXTERN  g_OS_Tcb_CurP
  EXTERN  g_OS_Tcb_HighRdyP

  PUBLIC OSStart_Asm
  PUBLIC PendSV_Handler
  PUBLIC OSCtxSw

OSCtxSw
    LDR     R0, =NVIC_INT_CTRL
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]
    BX      LR                                              ; Enable interrupts at
processor level

OSStart_Asm
    LDR     R0, =NVIC_SYSPRI14                              ; Set the PendSV
exception priority
    LDR     R1, =NVIC_PENDSV_PRI
    STRB    R1, [R0]

    MOVS    R0, #0                                          ; Set the PSP to 0 for
initial context switch call
    MSR     PSP, R0

    LDR     R0, =g_OS_CPU_ExceptStkBase                       ; Initialize the MSP
to the OS_CPU_ExceptStkBase
    LDR     R1, [R0]
    MSR     MSP, R1

    LDR     R0, =NVIC_INT_CTRL                              ; Trigger the PendSV
exception (causes context switch)
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]

    CPSIE   I                                               ; Enable interrupts at
processor level

OSStartHang
    B       OSStartHang                                     ; Should never get here



PendSV_Handler
    CPSID   I                                               ; Prevent interruption
during context switch
    MRS     R0, PSP                                         ; PSP is process stack
```

```
pointer
    CBZ     R0, OS_CPU_PendSVHandler_nosave                     ; Skip register save the
first time

    SUBS    R0, R0, #0x20                                       ; Save remaining regs
r4-11 on process stack
    STM     R0, {R4-R11}

    LDR     R1, =g_OS_Tcb_CurP                                      ; OSTCBCur-
>OSTCBStkPtr = SP;
    LDR     R1, [R1]
    STR     R0, [R1]                                            ; R0 is SP of process
being switched out

                                                                ; At this point, entire
context of process has been saved
OS_CPU_PendSVHandler_nosave
    LDR     R0, =g_OS_Tcb_CurP                                      ; OSTCBCur  =
OSTCBHighRdy;
    LDR     R1, =g_OS_Tcb_HighRdyP
    LDR     R2, [R1]
    STR     R2, [R0]

    LDR     R0, [R2]                                        ; R0 is new process SP; SP =
OSTCBHighRdy->OSTCBStkPtr;

    LDM     R0, {R4-R11}                                        ; Restore r4-11 from new
process stack
    ADDS    R0, R0, #0x20

    MSR     PSP, R0                                             ; Load PSP with new
process SP
    ORR     LR, LR, #0x04                                       ; Ensure exception
return uses process stack

    CPSIE   I
    BX      LR                                                  ; Exception return will
restore remaining context

    END
```

main.c内容如下：

```
#include "stdio.h"
#define OS_EXCEPT_STK_SIZE 1024
#define TASK_1_STK_SIZE 1024
#define TASK_2_STK_SIZE 1024

typedef unsigned int OS_STK;
typedef void (*OS_TASK)(void);

typedef struct OS_TCB
{
  OS_STK *StkAddr;
}OS_TCB,*OS_TCBP;


OS_TCBP g_OS_Tcb_CurP;
OS_TCBP g_OS_Tcb_HighRdyP;

static OS_STK OS_CPU_ExceptStk[OS_EXCEPT_STK_SIZE];
OS_STK *g_OS_CPU_ExceptStkBase;

static OS_TCB TCB_1;
static OS_TCB TCB_2;
static OS_STK TASK_1_STK[TASK_1_STK_SIZE];
static OS_STK TASK_2_STK[TASK_2_STK_SIZE];

extern void OSStart_Asm(void);
extern void OSCtxSw(void);

void Task_Switch()
{
  if(g_OS_Tcb_CurP == &TCB_1)
```

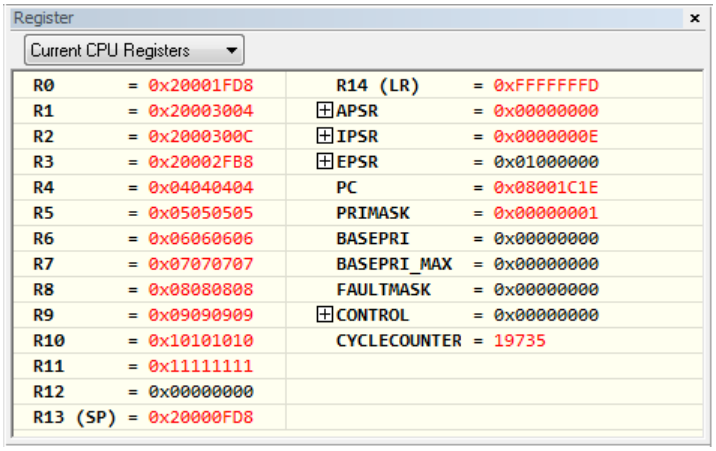
```
    g_OS_Tcb_HighRdyP=&TCB_2;
  else
    g_OS_Tcb_HighRdyP=&TCB_1;

  OSCtxSw();
}


void task_1()
{
  printf("Task 1 Running!!!\n");
  Task_Switch();
  printf("Task 1 Running!!!\n");
  Task_Switch();
}

void task_2()
{

  printf("Task 2 Running!!!\n");
  Task_Switch();
  printf("Task 2 Running!!!\n");
  Task_Switch();
}

void Task_End(void)
{
  printf("Task End\n");
  while(1)
  {}
}

void Task_Create(OS_TCB *tcb,OS_TASK task,OS_STK *stk)
{
    OS_STK  *p_stk;
    p_stk      = stk;
    p_stk      = (OS_STK *)((OS_STK)(p_stk) & 0xFFFFFFF8u);

    *(--p_stk) = (OS_STK)0x01000000uL;                          //xPSR
    *(--p_stk) = (OS_STK)task;                                  // Entry Point
    *(--p_stk) = (OS_STK)Task_End;                                      // R14 (LR)
    *(--p_stk) = (OS_STK)0x12121212uL;                          // R12
    *(--p_stk) = (OS_STK)0x03030303uL;                          // R3
    *(--p_stk) = (OS_STK)0x02020202uL;                          // R2
    *(--p_stk) = (OS_STK)0x01010101uL;                          // R1
    *(--p_stk) = (OS_STK)0x00000000u;                           // R0

    *(--p_stk) = (OS_STK)0x11111111uL;                          // R11
    *(--p_stk) = (OS_STK)0x10101010uL;                          // R10
    *(--p_stk) = (OS_STK)0x09090909uL;                          // R9
    *(--p_stk) = (OS_STK)0x08080808uL;                          // R8
    *(--p_stk) = (OS_STK)0x07070707uL;                          // R7
    *(--p_stk) = (OS_STK)0x06060606uL;                          // R6
    *(--p_stk) = (OS_STK)0x05050505uL;                          // R5
    *(--p_stk) = (OS_STK)0x04040404uL;                          // R4

    tcb->StkAddr=p_stk;
}


int main()
{

  g_OS_CPU_ExceptStkBase = OS_CPU_ExceptStk + OS_EXCEPT_STK_SIZE - 1;

  Task_Create(&TCB_1,task_1,&TASK_1_STK[TASK_1_STK_SIZE-1]);
  Task_Create(&TCB_2,task_2,&TASK_2_STK[TASK_1_STK_SIZE-1]);

  g_OS_Tcb_HighRdyP=&TCB_1;

  OSStart_Asm();

  return 0;
}
```

编译下载并调试：

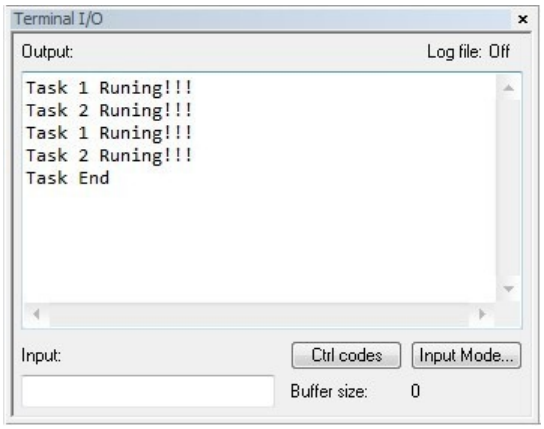
在此处设置断点

```
MSR     PSP, R0
ORR     LR, LR, #0x04

CPSIE   I
BX      LR

END
```

此时寄存器的值，可以看到R4-R11正是我们给的值，单步运行几次，可以看到进入了我们的任务task_1或task_2，任务里打印信息，然后调用Task_Switch进行切换，OSCtxSw触发PendSV异常。



IO输出如下：



至此我们成功实现了使用PenSV进行两个任务的互相切换。之后，我们使用使用SysTick实现比较完整的多任务切换。

[源码下载] stepbystep_stm32_os_PendSV.rar

分类: 一步步写STM32 OS

标签: stm32, OS

sky1991
关注 - 0
粉丝 - 28
+加关注

« 上一篇：一步步写STM32 OS【二】环境搭建
» 下一篇：一步步写STM32 OS【四】OS基本框架

posted @ 2013-11-02 13:36 sky1991 阅读(14397) 评论(3) 编辑 收藏

评论列表

#1楼 2016-04-15 09:34 李可jesscia

博主，我是初学嵌入式，看了你的内容很有用，求解一个问题，我想"验证中断级任务切换"，但是不会编写"OSTickISR"汇编，求助

支持(0) 反对(0)

#2楼[楼主] 2016-04-15 09:44 sky1991

@ 李可jesscia
哈哈，博客好久不更新了，没想到还有评论。这里说声抱歉，这个博客是我本科时写的，现在已经不做那些东西了。年久失修，多数都已忘却。所以这个还要靠你自己了，加油。。。

支持(0) 反对(0)

#3楼 2016-04-15 09:50 李可jesscia

我大四呢，这不是选了一个坑爹的毕业设计嘛，看到你的帖子了，很赞！

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。