一步步写STM32 OS【一】序言

sky1991 2013-10-28 原文

一直想写个类似uCOS的OS，近段时间考研复习之余忙里偷闲，总算有点成果了。言归正传，我觉得OS最难的部分首先便是上下文切换的问题，他和MCU的架构有关，所以对于不同的MCU，这部分需要移植。一旦这个问题解决了，整个OS算是成功了一半了，当然，是对于简单的OS。

好了，要写一个OS，首先需要一个开发板和仿真器。我的开发板是STM32F4DISCOVERY,自带ST-LINK V2仿真器，板载MCU为STM32F407VGT6，支持FPU，32位ARM Cortex-M4F核，1024KB FLASH，192 KB RAM，总之很强大。对STM32其他系列，本OS几乎不需修改修改就可使用。开发环境为IAR for ARM 6.5，如果是MDK的话，也是大同小异，汇编部分需要修改。

研究了一下UCOS-II的Cortex-M4的Port部分，觉得很好，就直接拿来用了，修改的很少。首先我们来看一下这一部分几个比较重要的函数，打开os_cpu_a.asm文件，定位到下面的地方，注释我改成中文了。当OS初始化完毕后，执行OSStart，OSStart最后调用OSStartHighRdy函数，注意在此之前的线程模式和异常模式的堆栈都是MSP，在此之后线程模式的堆栈是PSP，异常模式的堆栈仍是MSP。

```
1.  OSStartHighRdy
2.      LDR       R0, =NVIC_SYSPRI14          ; 设置PendSV的异常中
    断优先级
3.      LDR       R1, =NVIC_PENDSV_PRI
4.      STRB      R1, [R0]
5.
6.      MOVS      R0, #                       ; 初始化PSP=0
7.      MSR       PSP, R0
8.
9.      LDR       R0, =OS_CPU_ExceptStkBase   ; 初始化异常堆栈MSP地
    址
10.     LDR       R1, [R0]
11.     MSR       MSP, R1
12.
```

```
    LDR     R0, =OSRunning              ; 置OSRunning = TRUE
    MOVS    R1, #
    STRB    R1, [R0]

    LDR     R0, =NVIC_INT_CTRL          ; 触发PendSV异常 (引起上下文切换)
    LDR     R1, =NVIC_PENDSVSET
    STR     R1, [R0]

    CPSIE   I                           ; 开启中断，于是进入PendSV异常

OSStartHang
    B       OSStartHang                 ; 正常情况下，不应运行到这
```

其中最核心的函数应该是OS_CPU_PendSVHandler了，它处理PendSV中断，完成上下文切换。

```
OS_CPU_PendSVHandler
    CPSID   I                                   ; 关中断
    MRS     R0, PSP                             ; 获得PSP
    CBZ     R0, OS_CPU_PendSVHandler_nosave; PSP为0跳到OS_CPU_PendSVHandler_nosave，即不保存上文，直接进入下文。
                                                ; 问什么呢，因为首次调用，是没有上文的。
                                                ; 保存上文
    SUBS    R0, R0, #0x20                       ; 因为寄存器是32位的，4字节对齐，自动压栈的寄存器有8个，所以偏移为8*0x04=0x20
    STM     R0, {R4-R11}                        ; 除去自动压栈的寄存器外，需手动将R4-R11压栈

    LDR     R1, =OSTCBCur                       ; 保存上文的SP指针 OSTCBCur->OSTCBStkPtr = SP;
    LDR     R1, [R1]
    STR     R0, [R1]

OS_CPU_PendSVHandler_nosave                     ; 切换下文
    PUSH    {R14}                               ; LR压栈，下面要调用C函数
    LDR     R0, =OSTaskSwHook                   ; 调用OSTaskSwHook();
    BLX     R0
    POP     {R14}

    LDR     R0, =OSPrioCur                      ; 置OSPrioCur = OSPrioHighRdy;
    LDR     R1, =OSPrioHighRdy
    LDRB    R2, [R1]
```

```
23.     STRB    R2, [R0]
24.
25.     LDR     R0, =OSTCBCur                   ; 置OSTCBCur  =
    OSTCBHighRdy;
26.     LDR     R1, =OSTCBHighRdy
27.     LDR     R2, [R1]
28.     STR     R2, [R0]
29.
30.     LDR     R0, [R2]                        ; R0中的值为新任
    务的SP; SP = OSTCBHighRdy->OSTCBStkPtr;
31.     LDM     R0, {R4-R11}                    ; 手动弹出 R4-
    R11
32.     ADDS    R0, R0, #0x20
33.
34.     MSR     PSP, R0                         ; PSP = 新任务SP
35.     ORR     LR, LR, #0x04                   ; 确保异常返回后
    使用PSP
36.     CPSIE   I
37.     BX      LR                              ; 退出异常，从PSP
    弹出xPSR，PC，LR，R0-R3，进入新任务运行
```

之后我们在此基础上构建自己的OS，首先完成两个任务互相调用，然后是加入SysTick的任务调度，最后加入信号量，邮箱等功能。

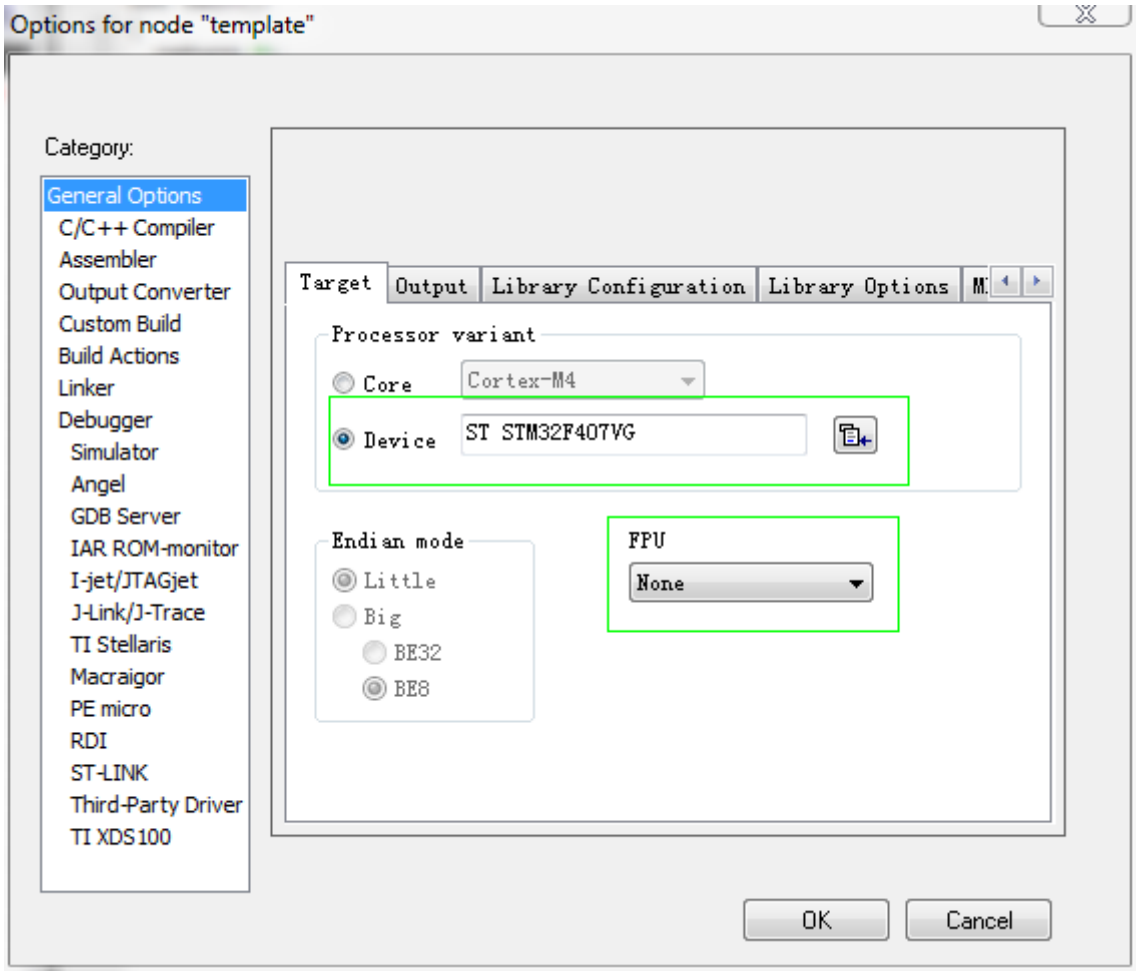
一步步写STM32 OS【二】环境搭建

sky1991 2013-11-02 原文

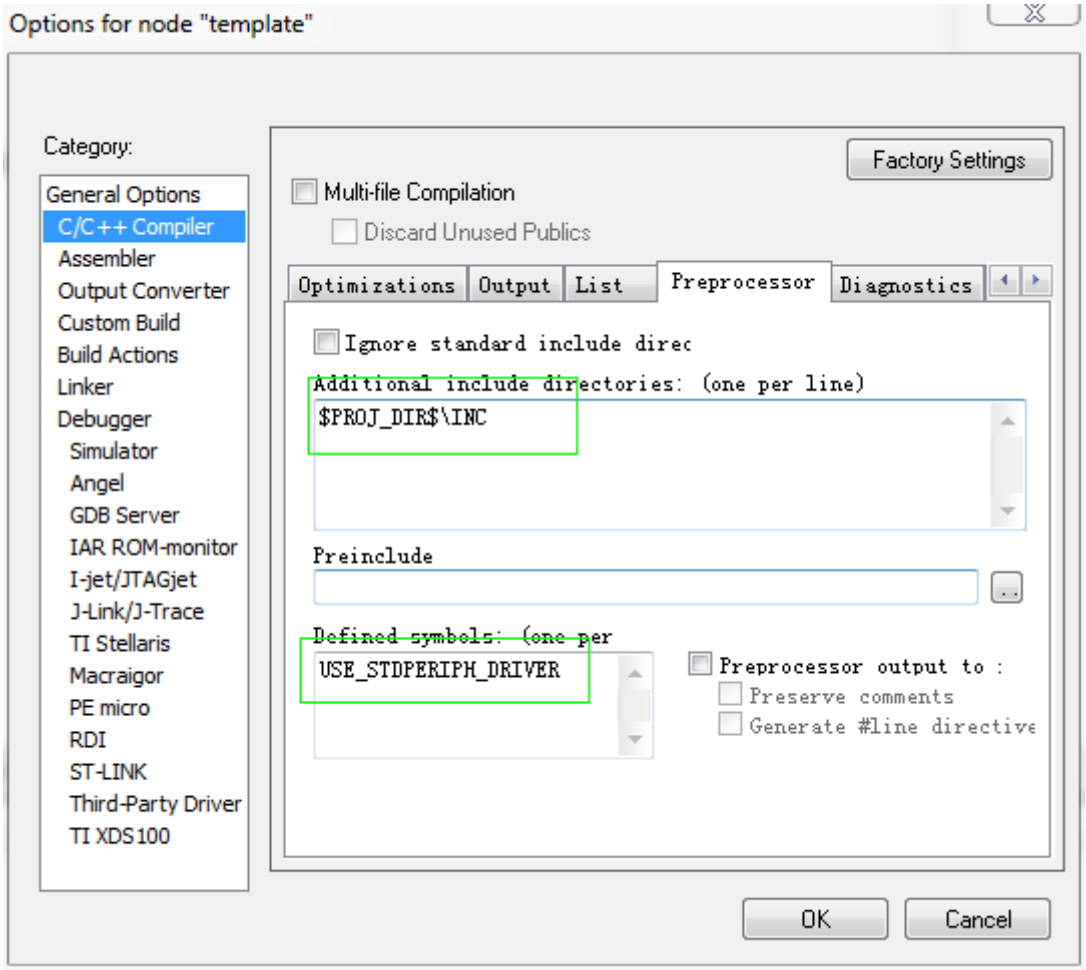
一、安装IAR for ARM6.5

二、新建工程

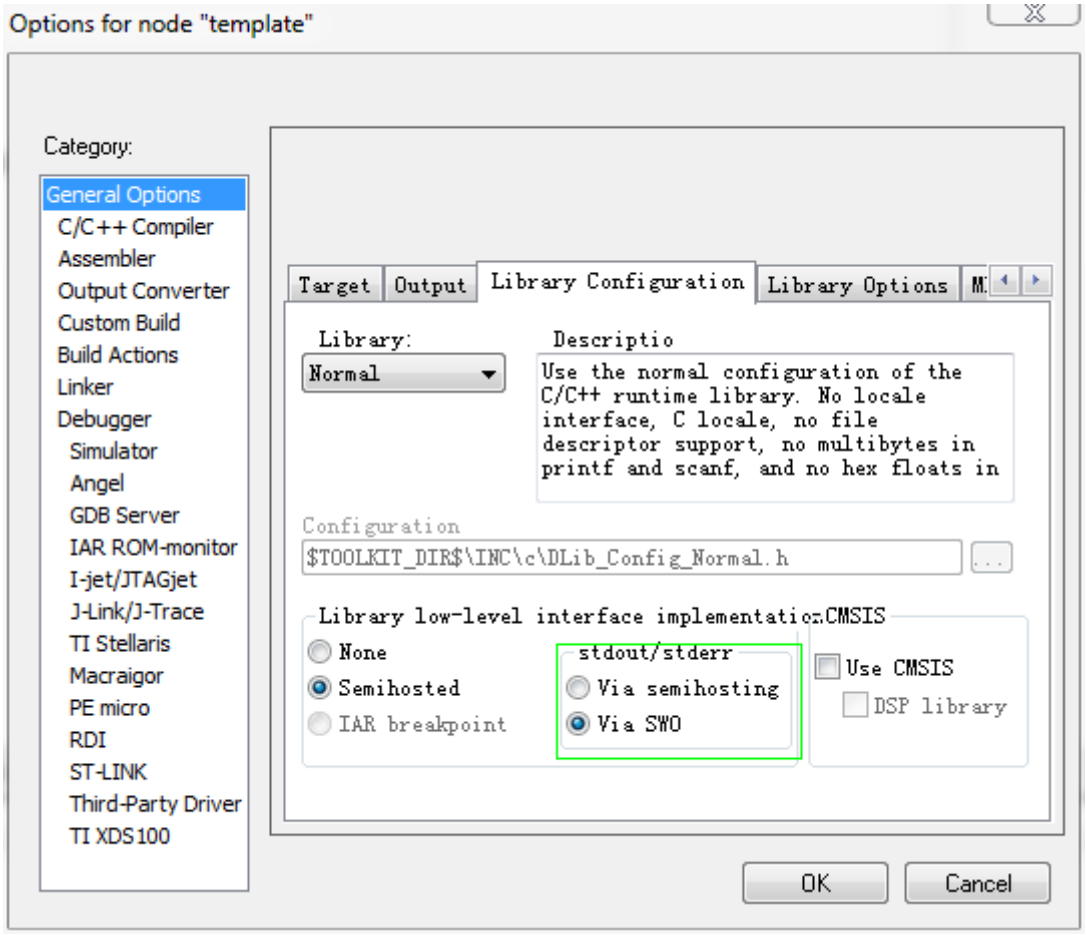
1、选择处理器：STM32F407VG，暂不使用FPU



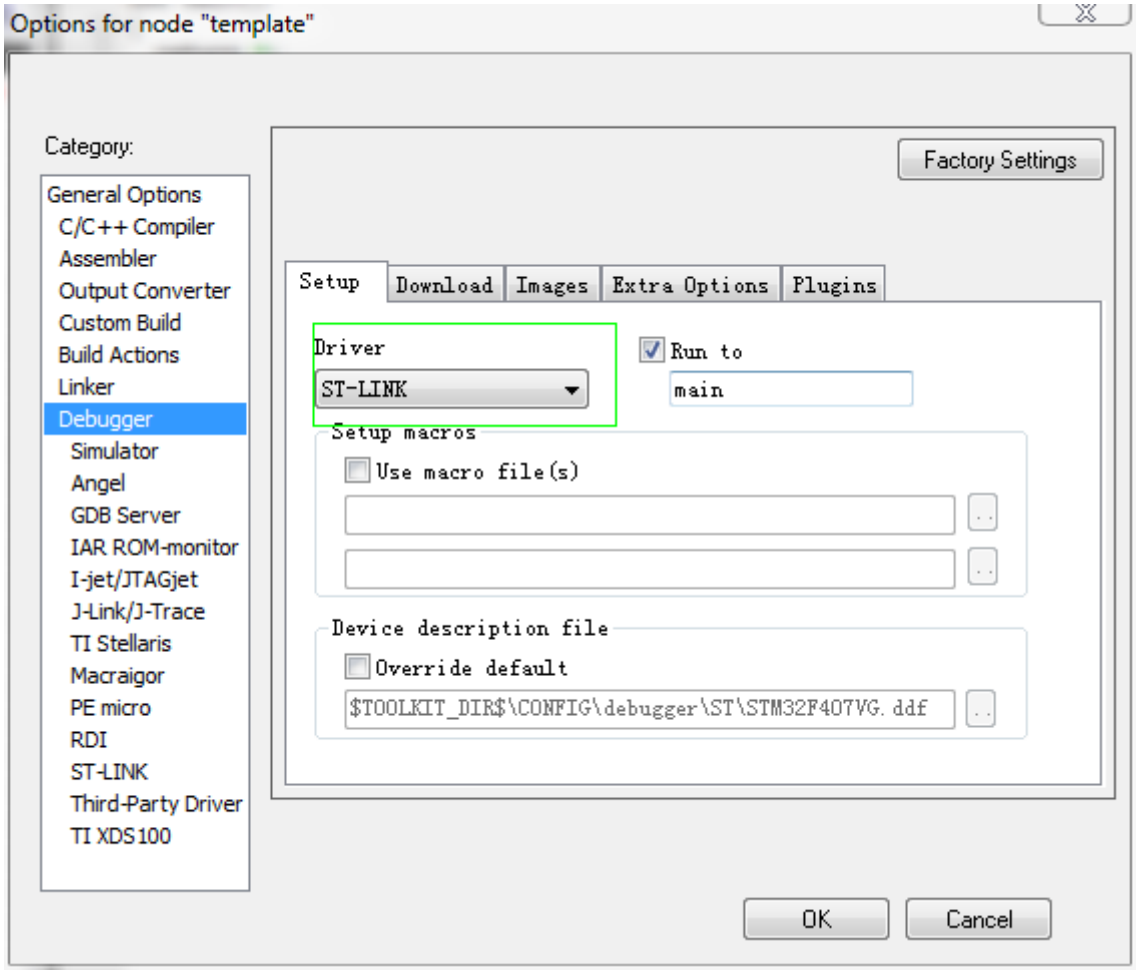
2、必要的路径配置和宏定义



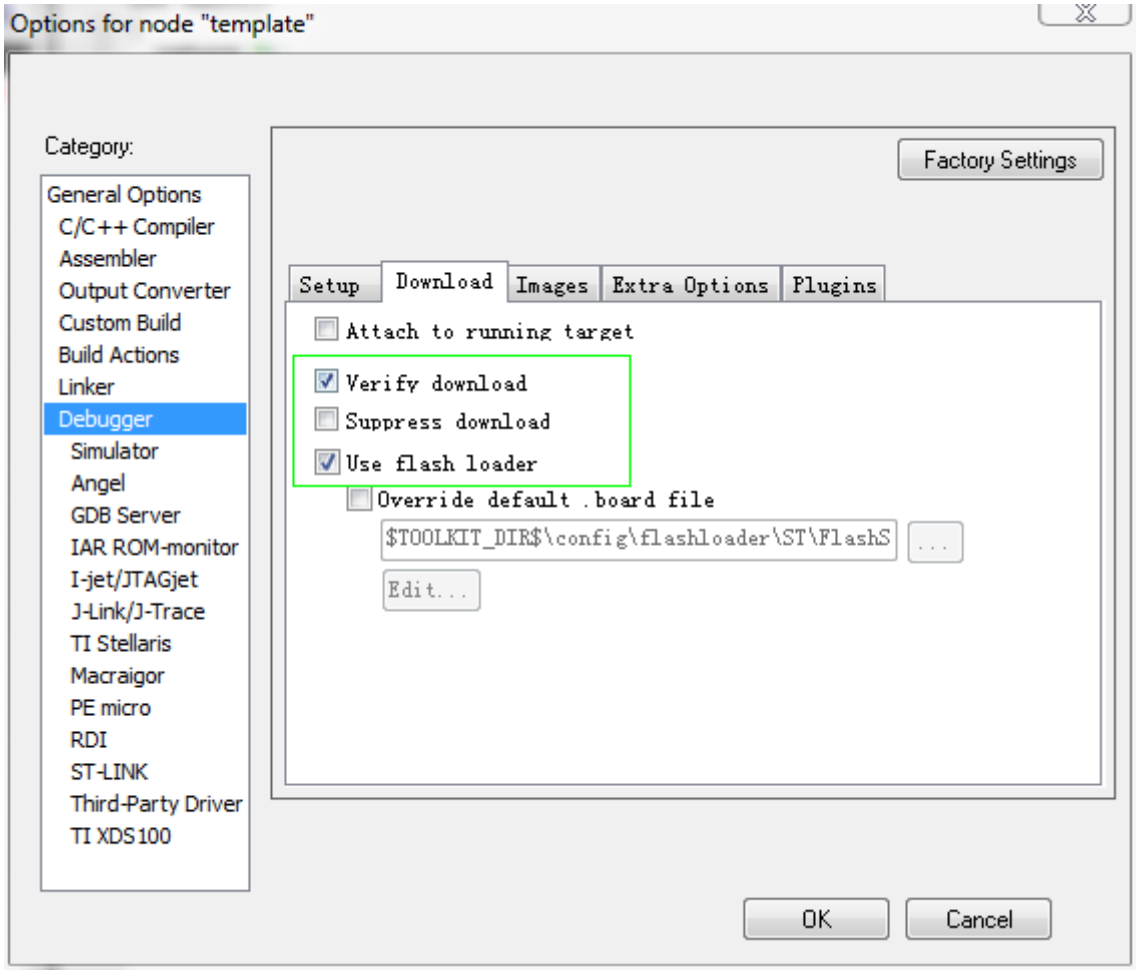
3、使用SWO重定向IO输出



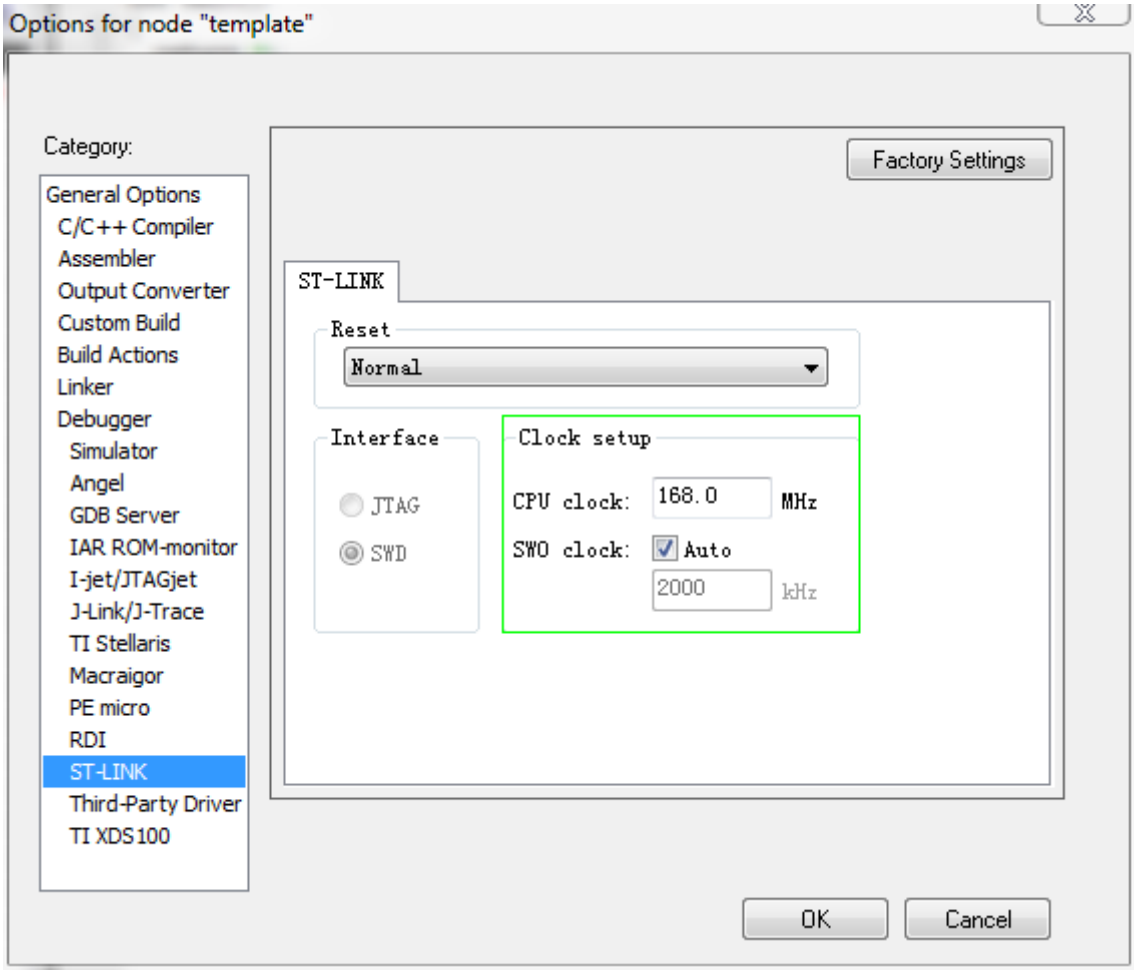
4、使用ST-LINK仿真器


Options for node "template"
Category:
General Options
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
GDB Server
IAR ROM-monitor
I-jet/JTAGjet
J-Link/J-Trace
TI Stellaris
Macraigor
PE micro
RDI
ST-LINK
Third-Party Driver
TI XDS100
Factory Settings
Setup
Download
Images
Extra Options
Plugins
Driver
ST-LINK
Run to
main
Setup macros
Use macro file(s)
Device description file
Override default
$TOOLKIT_DIR$\CONFIG\debugger\ST\STM32F407VG.ddf
OK
Cancel

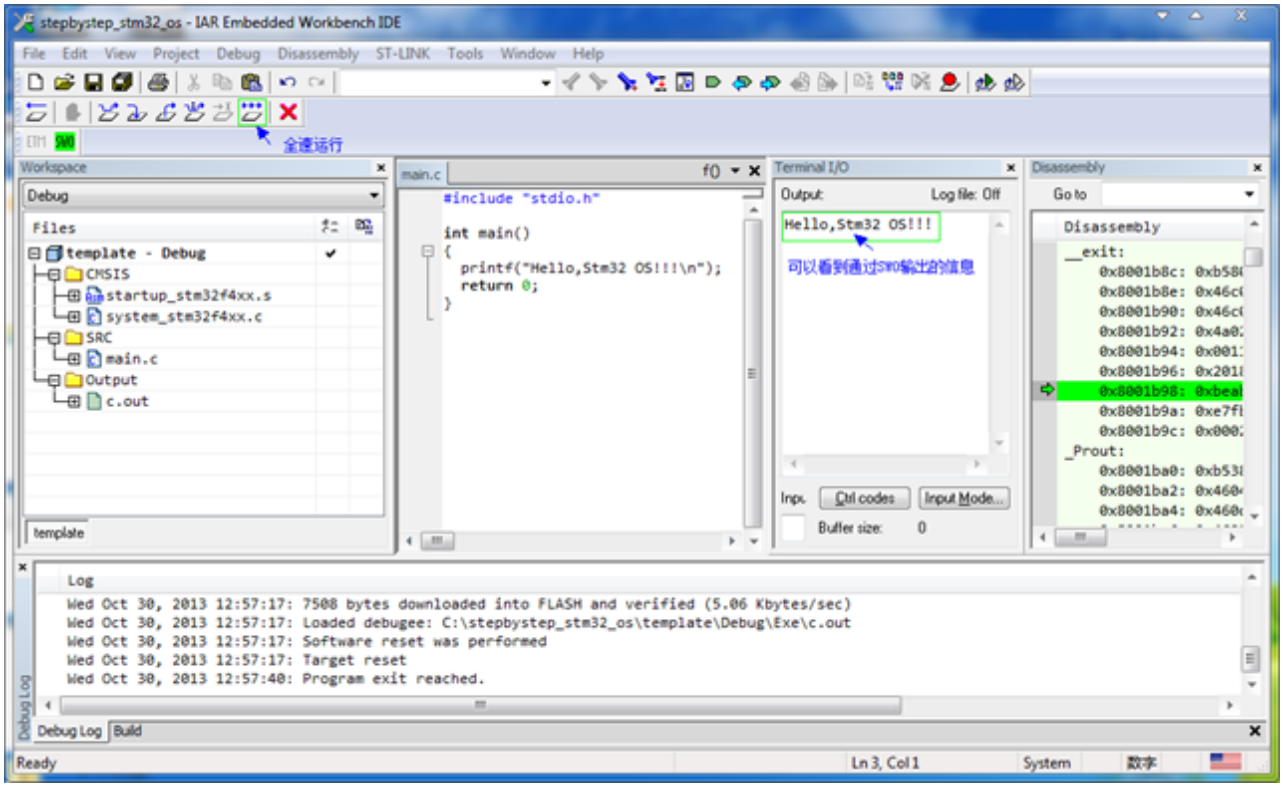
5、下载配置



6、设置CPU频率，防止SWO输出乱码


Options for node "template"
Category:
General Options
C/C++ Compiler
Assembler
Output Converter
Custom Build
Build Actions
Linker
Debugger
Simulator
Angel
GDB Server
IAR ROM-monitor
I-jet/JTAGjet
J-Link/J-Trace
TI Stellaris
Macraigor
PE micro
RDI
ST-LINK
Third-Party Driver
TI XDS100
Factory Settings
ST-LINK
Reset
Normal
Interface
JTAG
SWD
Clock setup
CPU clock:
168.0
MHz
SWO clock:
Auto
2000
kHz
OK
Cancel

三、代码调试


stepbystep_stm32_os - IAR Embedded Workbench IDE
全速运行
Workspace
Debug
Files
template - Debug
CMSIS
startup_stm32f4xx.s
system_stm32f4xx.c
SRC
main.c
Output
c.out
#include "stdio.h"
int main()
{
printf("Hello,Stm32 OS!!!\n");
return 0;
}
Terminal I/O
Output:
Log file: Off
Hello,Stm32 OS!!!
可以看到通过SWO输出的信息
Disassembly
Log
Wed Oct 30, 2013 12:57:17: 7508 bytes downloaded into FLASH and verified (5.06 Kbytes/sec)
Wed Oct 30, 2013 12:57:17: Loaded debugee: C:\stepbystep_stm32_os\template\Debug\Exe\c.out
Wed Oct 30, 2013 12:57:17: Software reset was performed
Wed Oct 30, 2013 12:57:17: Target reset
Wed Oct 30, 2013 12:57:40: Program exit reached.
Debug Log
Build
Ready
Ln 3, Col 1
System

四、工程下载

stepbystep_stm32_os_template.rar

一步步写STM32 OS【三】PendSV与堆栈操作

sky1991 2013-11-02 原文

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
```

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, #
   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

```
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 )
  THUMB

  EXTERN  g_OS_CPU_ExceptStkBase

  EXTERN  g_OS_Tcb_CurP
  EXTERN  g_OS_Tcb_HighRdyP

  PUBLIC OSStart_Asm
  PUBLIC PendSV_Handler
```

```
16.   PUBLIC OSCtxSw
17.
18.  OSCtxSw
19.      LDR      R0, =NVIC_INT_CTRL
20.      LDR      R1, =NVIC_PENDSVSET
21.      STR      R1, [R0]
22.      BX       LR
     ; Enable interrupts at processor level
23.
24.  OSStart_Asm
25.      LDR      R0, =NVIC_SYSPRI14
     ; Set the PendSV exception priority
26.      LDR      R1, =NVIC_PENDSV_PRI
27.      STRB     R1, [R0]
28.
29.      MOVS     R0, #
     ; Set the PSP to 0 for initial context switch call
30.      MSR      PSP, R0
31.
32.      LDR      R0, =g_OS_CPU_ExceptStkBase
     ; Initialize the MSP to the OS_CPU_ExceptStkBase
33.      LDR      R1, [R0]
34.      MSR      MSP, R1
35.
36.      LDR      R0, =NVIC_INT_CTRL
     ; Trigger the PendSV exception (causes context switch)
37.      LDR      R1, =NVIC_PENDSVSET
38.      STR      R1, [R0]
39.
40.      CPSIE    I
     ; Enable interrupts at processor level
41.
42.  OSStartHang
43.      B        OSStartHang
     ; Should never get here
44.
45.  PendSV_Handler
46.      CPSID    I
     ; Prevent interruption during context switch
47.      MRS      R0, PSP
     ; PSP is process stack pointer
48.      CBZ      R0, OS_CPU_PendSVHandler_nosave
     ; Skip register save the first time
49.
50.      SUBS     R0, R0, #0x20
     ; Save remaining regs r4-11 on process stack
51.      STM      R0, {R4-R11}
52.
53.      LDR      R1, =g_OS_Tcb_CurP
     ; OSTCBCur->OSTCBStkPtr = SP;
54.      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;
```

```
19.
20.  static OS_TCB TCB_1;
21.  static OS_TCB TCB_2;
22.  static OS_STK TASK_1_STK[TASK_1_STK_SIZE];
23.  static OS_STK TASK_2_STK[TASK_2_STK_SIZE];
24.
25.  extern void OSStart_Asm(void);
26.  extern void OSCtxSw(void);
27.
28.  void Task_Switch()
29.  {
30.    if(g_OS_Tcb_CurP == &TCB_1)
31.      g_OS_Tcb_HighRdyP=&TCB_2;
32.    else
33.      g_OS_Tcb_HighRdyP=&TCB_1;
34.
35.    OSCtxSw();
36.  }
37.
38.  void task_1()
39.  {
40.    printf("Task 1 Running!!!\n");
41.    Task_Switch();
42.    printf("Task 1 Running!!!\n");
43.    Task_Switch();
44.  }
45.
46.  void task_2()
47.  {
48.
49.    printf("Task 2 Running!!!\n");
50.    Task_Switch();
51.    printf("Task 2 Running!!!\n");
52.    Task_Switch();
53.  }
54.
55.  void Task_End(void)
56.  {
57.    printf("Task End\n");
58.    )
59.    {}
60.  }
61.
62.  void Task_Create(OS_TCB *tcb,OS_TASK task,OS_STK *stk)
63.  {
64.      OS_STK  *p_stk;
65.      p_stk       = stk;
66.      p_stk       = (OS_STK *)((OS_STK)(p_stk)  &
     0xFFFFFFF8u);
67.
68.      *(--p_stk)  = (OS_STK)0x01000000uL;
     //xPSR
```

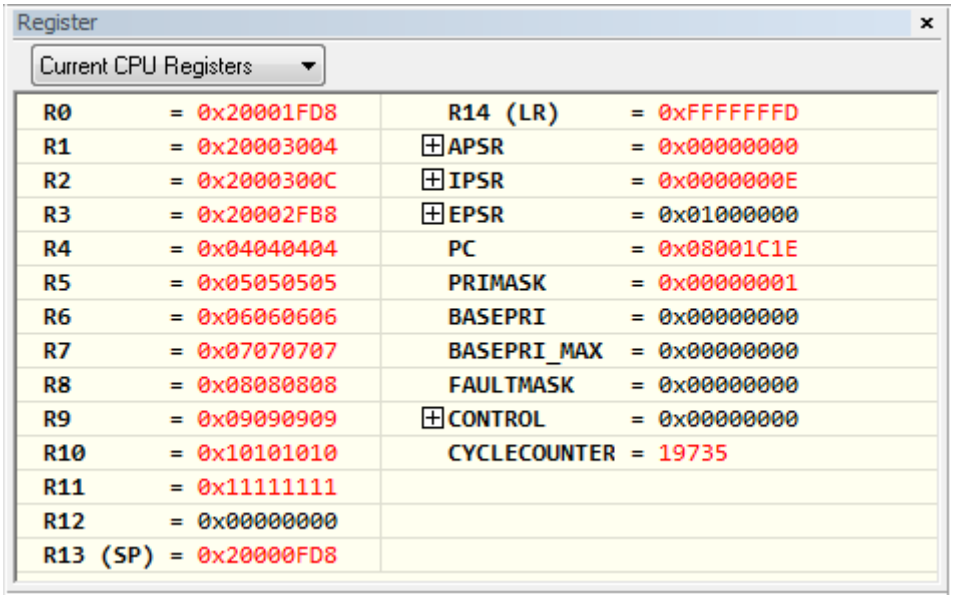
```
69.      *(--p_stk) = (OS_STK)task;
    // Entry Point
70.      *(--p_stk) = (OS_STK)Task_End;
    // R14 (LR)
71.      *(--p_stk) = (OS_STK)0x12121212uL;
    // R12
72.      *(--p_stk) = (OS_STK)0x03030303uL;
    // R3
73.      *(--p_stk) = (OS_STK)0x02020202uL;
    // R2
74.      *(--p_stk) = (OS_STK)0x01010101uL;
    // R1
75.      *(--p_stk) = (OS_STK)0x00000000u;
    // R0
76.
77.      *(--p_stk) = (OS_STK)0x11111111uL;
    // R11
78.      *(--p_stk) = (OS_STK)0x10101010uL;
    // R10
79.      *(--p_stk) = (OS_STK)0x09090909uL;
    // R9
80.      *(--p_stk) = (OS_STK)0x08080808uL;
    // R8
81.      *(--p_stk) = (OS_STK)0x07070707uL;
    // R7
82.      *(--p_stk) = (OS_STK)0x06060606uL;
    // R6
83.      *(--p_stk) = (OS_STK)0x05050505uL;
    // R5
84.      *(--p_stk) = (OS_STK)0x04040404uL;
    // R4
85.
86.      tcb->StkAddr=p_stk;
87.  }
88.
89.  int main()
90.  {
91.
92.    g_OS_CPU_ExceptStkBase = OS_CPU_ExceptStk +
    OS_EXCEPT_STK_SIZE - ;
93.
94.
    Task_Create(&TCB_1,task_1,&TASK_1_STK[TASK_1_STK_SIZE-]);
95.
    Task_Create(&TCB_2,task_2,&TASK_2_STK[TASK_1_STK_SIZE-]);
96.
97.    g_OS_Tcb_HighRdyP=&TCB_1;
98.
99.    OSStart_Asm();
100.
101.   ;
102. }
```

编译下载并调试：

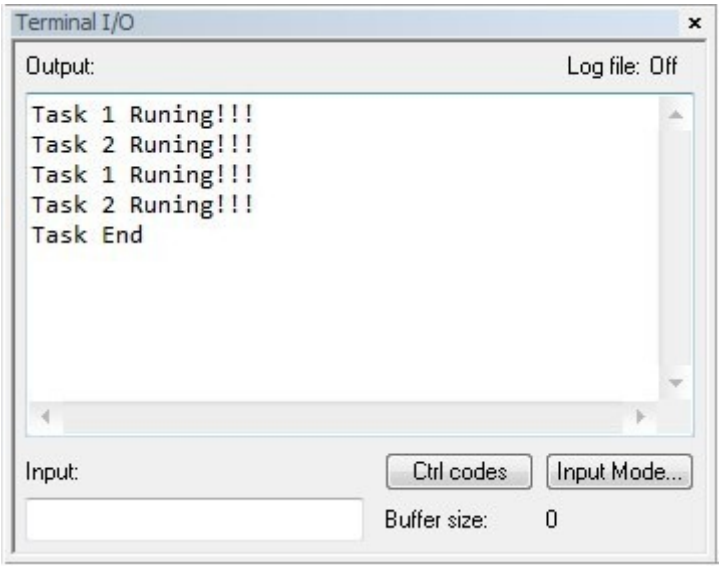
在此处设置断点

```
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

一步步写STM32 OS【四】OS基本框架

sky1991 2013-11-03 原文

一、上篇回顾

上一篇文章中，我们完成了两个任务使用PendSV实现了互相切换的功能，下面我们接着其思路往下做。这次我们完成OS基本框架，即实现一个非抢占式（已经*调度*的进程执行完成，然后根据优先级*调度*等待的进程）的任务调度系统，至于抢占式的，就留给大家思考了。上次代码中Task_Switch实现了两个任务的切换，代码如下：

```
1.  void Task_Switch()
2.  {
3.    if(g_OS_Tcb_CurP == &TCB_1)
4.      g_OS_Tcb_HighRdyP=&TCB_2;
5.    else
6.      g_OS_Tcb_HighRdyP=&TCB_1;
7.    OSCtxSw();
8.  }
```

我们把要切换任务指针付给跟_OS_Tcb_HighRdyP，然后调用OSCtxSw触发PendSV异常，就实现了任务的切换。如果是多个任务，我们只需找出就绪任务中优先级最大的切换之即可。

二、添加任务调度功能

为了实现这一目标我们至少需要知道任务的状态和时间等数据。我们定义了一个任务状态枚举类型OS_TASK_STA，方便添加修改状态。在OS_TCB结构体中添加了两个成员TimeDly和State，TimeDly是为了实现OS_TimeDly，至于State与优先级一起是作为任务切换的依据。

```
1.  typedef enum OS_TASK_STA
2.  {
3.    TASK_READY,
4.    TASK_DELAY,
```

```
} OS_TASK_STA;

typedef struct OS_TCB
{
  OS_STK *StkAddr;
  OS_U32 TimeDly;
  OS_TASK_STA State;
}OS_TCB,*OS_TCBP;
```

说到任务切换，我们必须面对临界区的问题，在一些临界的代码两端不加临界区进去和退出代码，会出现许多意想不到的问题。以下地方需要特别注意，对关键的全局变量的写操作、对任务控制块的操作等。进入临界区和退出临界区需要关闭和开启中断，我们采用uCOS中的一部分代码：

```
PUBLIC OS_CPU_SR_Save
  PUBLIC OS_CPU_SR_Restore

OS_CPU_SR_Save
    MRS     R0, PRIMASK
    CPSID   I
    BX      LR

OS_CPU_SR_Restore
    MSR     PRIMASK, R0
    BX      LR
```

```
#define  OS_USE_CRITICAL    OS_U32 cpu_sr;
#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
#define  OS_PendSV_Trigger() OSCtxSw()
```

一个OS至少要有任务表，我们可以用数组，当然也可以用链表。为了简单，我们使用数组，使用数组下表作为优先级。当然，必要的地方一定要做数组越界检查。

```
#define OS_TASK_MAX_NUM 32
OS_TCBP OS_TCB_TABLE[OS_TASK_MAX_NUM];
```

为了使OS更完整，我们定义几个全局变量，OS_TimeTick记录系统时间，g_Prio_Cur记录当前运行的任务优先级，g_Prio_HighRdy记录任务调度后就绪任务中的最高优先级。

```
OS_U32 OS_TimeTick;
OS_U8 g_Prio_Cur;
OS_U8 g_Prio_HighRdy;
```

下面三个函数与PendSV一起实现了任务的调度功能。

OS_Task_Switch函数功能：找出已就绪最高优先级的任务，并将其TCB指针赋值给g_OS_Tcb_HighRdyP，将其优先级赋值g_Prio_HighRdy。注意其中使用了临界区。

```
void OS_Task_Switch(void)
{
  OS_S32 i;
  OS_TCBP tcb_p;
  OS_USE_CRITICAL
  ;i<OS_TASK_MAX_NUM;i++)
  {
    tcb_p=OS_TCB_TABLE[i];
    if(tcb_p == NULL) continue;
    if(tcb_p->State==TASK_READY) break;
  }
  OS_ENTER_CRITICAL();
  g_OS_Tcb_HighRdyP=tcb_p;
  g_Prio_HighRdy=i;
  OS_EXIT_CRITICAL();
}
```

OS_TimeDly至当前任务为延时状态，并将延时时间赋值给当前TCB的TimeDly成员，并调用OS_Task_Switch函数，然后触发PendSV进行上下文切换。OS_Task_Switch找到就绪状态中优先级最高的，并将其赋值相关全局变量，作为上下文切换的依据。

```
void OS_TimeDly(OS_U32 ticks)
{
    OS_USE_CRITICAL

    OS_ENTER_CRITICAL();
    g_OS_Tcb_CurP->State=TASK_DELAY;
    g_OS_Tcb_CurP->TimeDly=ticks;
    OS_EXIT_CRITICAL();
    OS_Task_Switch();
    OS_PendSV_Trigger();
}
```

SysTick_Handler实现系统计时，并遍历任务表，任务若是延时状态，就令其延时值减一，若减完后为零，就将其置为就绪状态。

```
void SysTick_Handler(void)
{
  OS_TCBP tcb_p;
  OS_S32 i;
```

```
  OS_USE_CRITICAL

  OS_ENTER_CRITICAL();
  ++OS_TimeTick;
    ;i<OS_TASK_MAX_NUM;i++)
    {
      tcb_p=OS_TCB_TABLE[i];
      if(tcb_p == NULL) continue;
      if(tcb_p->State==TASK_DELAY)
      {
        --tcb_p->TimeDly;
        )
          tcb_p->State=TASK_READY;
      }
    }
  OS_EXIT_CRITICAL();
}
```

当所有任务都没就绪怎么办？这时就需要空闲任务了，我们把它设为优先级最低的任务。WFE指令为休眠指令，当来中断时，退出休眠，然后看看有没有已就绪的任务，有则调度之，否则继续休眠，这样可以减小功耗哦。

```
void OS_Task_Idle(void)
{
  )
  {
    asm("WFE");
    OS_Task_Switch();
    OS_PendSV_Trigger();
  }
}
```

当一个任务只运行一次时（例如下面main.c的task1），结束时就会调用OS_Task_End函数，此函数会调用OS_Task_Delete函数从任务表中删除当前的任务，然后调度任务。

```
void OS_Task_Delete(OS_U8 prio)
{
  if(prio >= OS_TASK_MAX_NUM) return;
  OS_TCB_TABLE[prio]=;
}

void OS_Task_End(void)
{
  printf("Task of Prio %d End\n",g_Prio_Cur);
  OS_Task_Delete(g_Prio_Cur);
  OS_Task_Switch();
  OS_PendSV_Trigger();
```

```
}
```

三、OS实战

下面是完整的main.c代码：

```
#include "stdio.h"
#include "stm32f4xx.h"

#define OS_EXCEPT_STK_SIZE 1024
#define TASK_1_STK_SIZE 128
#define TASK_2_STK_SIZE 128
#define TASK_3_STK_SIZE 128

#define TASK_IDLE_STK_SIZE 1024
#define OS_TASK_MAX_NUM 32
#define OS_TICKS_PER_SECOND 1000

#define  OS_USE_CRITICAL    OS_U32 cpu_sr;
#define  OS_ENTER_CRITICAL()  {cpu_sr = OS_CPU_SR_Save();}
#define  OS_EXIT_CRITICAL()   {OS_CPU_SR_Restore(cpu_sr);}
#define  OS_PendSV_Trigger() OSCtxSw()

typedef signed char OS_S8;
typedef signed short OS_S16;
typedef signed int OS_S32;
typedef unsigned char OS_U8;
typedef unsigned short OS_U16;
typedef unsigned int OS_U32;
typedef unsigned int OS_STK;

typedef void (*OS_TASK)(void);

typedef enum OS_TASK_STA
{
  TASK_READY,
  TASK_DELAY,
} OS_TASK_STA;

typedef struct OS_TCB
{
  OS_STK *StkAddr;
  OS_U32 TimeDly;
  OS_U8 State;
}OS_TCB,*OS_TCBP;

OS_TCBP OS_TCB_TABLE[OS_TASK_MAX_NUM];
OS_TCBP g_OS_Tcb_CurP;
OS_TCBP g_OS_Tcb_HighRdyP;
OS_U32 OS_TimeTick;
```

```
OS_U8 g_Prio_Cur;
OS_U8 g_Prio_HighRdy;

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_TCB TCB_3;
static OS_TCB TCB_IDLE;
static OS_STK TASK_1_STK[TASK_1_STK_SIZE];
static OS_STK TASK_2_STK[TASK_2_STK_SIZE];
static OS_STK TASK_3_STK[TASK_3_STK_SIZE];
static OS_STK TASK_IDLE_STK[TASK_IDLE_STK_SIZE];

extern OS_U32 SystemCoreClock;

extern void OSStart_Asm(void);
extern void OSCtxSw(void);
extern OS_U32 OS_CPU_SR_Save(void);
extern void OS_CPU_SR_Restore(OS_U32);

void task_1(void);
void task_2(void);
void task_3(void);

void OS_Task_Idle(void);
void OS_TimeDly(OS_U32);
void OS_Task_Switch(void);
void OS_Task_Create(OS_TCB *,OS_TASK,OS_STK *,OS_U8);
void OS_Task_Delete(OS_U8);
void OS_Task_End(void);
void OS_Init(void);
void OS_Start(void);

void task_1(void)
{
    printf("[%d]Task 1 Runing!!!\n",OS_TimeTick);


OS_Task_Create(&TCB_2,task_2,&TASK_2_STK[TASK_2_STK_SIZE-]
,);

OS_Task_Create(&TCB_3,task_3,&TASK_3_STK[TASK_3_STK_SIZE-]
,);
}

void task_2(void)
{
  )
  {
    printf("[%d]Task 2 Runing!!!\n",OS_TimeTick);
```

```
93.       OS_TimeDly();
94.     }
95.   }
96.
97.   void task_3(void)
98.   {
99.     )
100.    {
101.      printf("[%d]Task 3 Runing!!!\n",OS_TimeTick);
102.      OS_TimeDly();
103.    }
104.  }
105.
106.  void OS_Task_Idle(void)
107.  {
108.    )
109.    {
110.      asm("WFE");
111.      OS_Task_Switch();
112.      OS_PendSV_Trigger();
113.    }
114.  }
115.
116.  void OS_TimeDly(OS_U32 ticks)
117.  {
118.      OS_USE_CRITICAL
119.
120.      OS_ENTER_CRITICAL();
121.      g_OS_Tcb_CurP->State=TASK_DELAY;
122.      g_OS_Tcb_CurP->TimeDly=ticks;
123.      OS_EXIT_CRITICAL();
124.      OS_Task_Switch();
125.      OS_PendSV_Trigger();
126.  }
127.
128.  void OS_Task_Switch(void)
129.  {
130.    OS_S32 i;
131.    OS_TCBP tcb_p;
132.    OS_USE_CRITICAL
133.    ;i<OS_TASK_MAX_NUM;i++)
134.    {
135.      tcb_p=OS_TCB_TABLE[i];
136.      if(tcb_p == NULL) continue;
137.      if(tcb_p->State==TASK_READY) break;
138.    }
139.    OS_ENTER_CRITICAL();
140.    g_OS_Tcb_HighRdyP=tcb_p;
141.    g_Prio_HighRdy=i;
142.    OS_EXIT_CRITICAL();
143.  }
144.
```

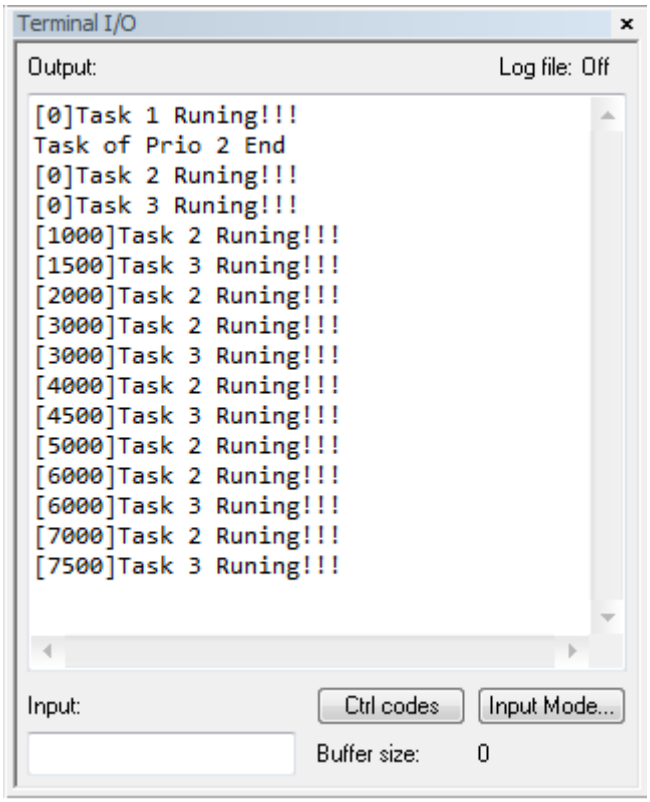
```
145.  void OS_Task_Delete(OS_U8 prio)
146.  {
147.    if(prio >= OS_TASK_MAX_NUM) return;
148.    OS_TCB_TABLE[prio]=;
149.  }
150.
151.  void OS_Task_End(void)
152.  {
153.    printf("Task of Prio %d End\n",g_Prio_Cur);
154.    OS_Task_Delete(g_Prio_Cur);
155.    OS_Task_Switch();
156.    OS_PendSV_Trigger();
157.  }
158.
159.  void OS_Task_Create(OS_TCB *tcb,OS_TASK task,OS_STK
      *stk,OS_U8 prio)
160.  {
161.      OS_USE_CRITICAL
162.      OS_STK  *p_stk;
163.      if(prio >= OS_TASK_MAX_NUM) return;
164.
165.      OS_ENTER_CRITICAL();
166.
167.      p_stk       = stk;
168.      p_stk       = (OS_STK *)((OS_STK)(p_stk)  &
      0xFFFFFFF8u);
169.
170.      *(--p_stk) = (OS_STK)0x01000000uL;
      //xPSR
171.      *(--p_stk) = (OS_STK)task;
      // Entry Point
172.      *(--p_stk) = (OS_STK)OS_Task_End;                    //
      R14 (LR)
173.      *(--p_stk) = (OS_STK)0x12121212uL;
      // R12
174.      *(--p_stk) = (OS_STK)0x03030303uL;
      // R3
175.      *(--p_stk) = (OS_STK)0x02020202uL;
      // R2
176.      *(--p_stk) = (OS_STK)0x01010101uL;
      // R1
177.      *(--p_stk) = (OS_STK)0x00000000u;
      // R0
178.
179.      *(--p_stk) = (OS_STK)0x11111111uL;
      // R11
180.      *(--p_stk) = (OS_STK)0x10101010uL;
      // R10
181.      *(--p_stk) = (OS_STK)0x09090909uL;
      // R9
182.      *(--p_stk) = (OS_STK)0x08080808uL;
      // R8
```

```
183.      *(--p_stk) = (OS_STK)0x07070707uL;
      // R7
184.      *(--p_stk) = (OS_STK)0x06060606uL;
      // R6
185.      *(--p_stk) = (OS_STK)0x05050505uL;
      // R5
186.      *(--p_stk) = (OS_STK)0x04040404uL;
      // R4
187.
188.      tcb->StkAddr=p_stk;
189.      tcb->TimeDly=;
190.      tcb->State=TASK_READY;
191.      OS_TCB_TABLE[prio]=tcb;
192.
193.      OS_EXIT_CRITICAL();
194.  }
195.
196.  void SysTick_Handler(void)
197.  {
198.
199.    OS_TCBP tcb_p;
200.    OS_S32 i;
201.    OS_USE_CRITICAL
202.
203.    OS_ENTER_CRITICAL();
204.    ++OS_TimeTick;
205.      ;i<OS_TASK_MAX_NUM;i++)
206.      {
207.        tcb_p=OS_TCB_TABLE[i];
208.        if(tcb_p == NULL) continue;
209.        if(tcb_p->State==TASK_DELAY)
210.        {
211.          --tcb_p->TimeDly;
212.          )
213.            tcb_p->State=TASK_READY;
214.        }
215.      }
216.    OS_EXIT_CRITICAL();
217.  }
218.
219.  void OS_Init(void)
220.  {
221.    int i;
222.    g_OS_CPU_ExceptStkBase = OS_CPU_ExceptStk +
      OS_EXCEPT_STK_SIZE - ;
223.    asm("CPSID   I");
224.    ;i<OS_TASK_MAX_NUM;i++)
225.      OS_TCB_TABLE[i]=;
226.    OS_TimeTick=;
227.
      OS_Task_Create(&TCB_IDLE,OS_Task_Idle,&TASK_IDLE_STK[TASK_
      IDLE_STK_SIZE-],OS_TASK_MAX_NUM-);
```

```
228.  }
229.
230.  void OS_Start(void)
231.  {
232.    OS_Task_Switch();
233.    SystemCoreClockUpdate();
234.    SysTick_Config(SystemCoreClock/OS_TICKS_PER_SECOND);
235.    OSStart_Asm();
236.  }
237.
238.  int main()
239.  {
240.
241.    OS_Init();
242.
      OS_Task_Create(&TCB_1,task_1,&TASK_1_STK[TASK_1_STK_SIZE-]
      ,);
243.    OS_Start();
244.
245.    ;
246.  }
```

os_port.asm变化不大，具体内容可以下载文章末尾提供的工程参考。

老规矩，下载调试，全速运行，观察Terminal IO窗口：



从输出来看，我们已经完成了目标。但不保证稳定性，可能有不少Bugs。至此，可以说其实写一个OS并不难，难的是写一个稳定安全高效的OS。所以，现在只是走了一小步，想

要完成一个成熟的OS，还需要不断测试，不断优化。例如，我们采用数组存储任务表，也可以采用链表，各有优缺点。我们只有一个任务表，也可以分成多个表，例如就续表，等待表等等。我们的任务调度部分运行时间不确定，对于实时OS，这是不可以的，怎么修改呢，例如像uCOS的查找表法那样。现在我们的系统只能创建并调度任务，还未加入其他功能，例如信号量、邮箱、队列、内存管理等。其实到了这里，大家完全可以发挥自己的创造力，参照本文开发自己的OS。如果以后有时间的话，还会再写几篇文章继续完善我们的OS。

四、工程下载

stepbystep_stm32_os_basic.rar