带你为 STM32 写一个不到 100 行代码的多任务 OS

-- Pony279

写在前面

本文面向的读者为了解 RTOS 但对于怎样写一个 RTOS 又无从入手的朋友，也可以是对 RTOS 没有任何概念的新手，但是至少要求你做过定时器实验和 GPIO 实验.

写这篇文档的主要原因是，虽然网上介绍编写 RTOS 的文章不少，但是在我看来都不够易懂,而且在有现成的 uC/OS 的情况下，网络上也很少与 STM32 有关的这类文章.

我将会在下面的内容中介绍 RTOS 的一点概念，并带你设计，调试和实现一个简易的多任务 OS.

事实上，在这篇文档里将要实现的多任务 OS 的代码极其简单，甚至不能称之为 OS! 我的主要目的是给大家展现一个从裸机通向多任务的 RTOS 的桥梁，引导大家到一个开始，而不是编写一个完全意义上的 RTOS. 其实在 RTOS 方面我也只是个菜鸟，所以如果你是高手，请不要批评程序的功能过于简单. 当然，如果我哪里讲得不对，或者在讲解的方式上还有改进的地方，欢迎能给我提出建议. 因为我还有打算在将来空闲的时间修正这份文档并不定期的发布后续文档噢 :-)

1. RTOS 是什么？为什么要有 RTOS?

RTOS(real time operating system)，也就是实时操作系统. 我并没有找到关于 RTOS 的严格定义，但在不少资料中有介绍 RTOS 的特点，如执行时间的可确定性，可裁剪和可固化性多任务抢占式等特性.

RTOS 具体的概念涉及的内容很多，下面我只从多任务入手，通过程序代码来引出 RTOS 必要性. **对 RTOS 已经有过了解的朋友可以直接跳过这一节.**

想必大家都做过 STM32 闪烁灯的实验吧，那么我们先用 ALIENTEK 的 Mini STM32 板来实现一个闪烁灯的实验代码**.(见附件 1_点灯实验)**，代码如下所示:

```
#include "sys.h"
using namespace periph;

// 软件延时函数
void delay(volatile int i){
    while(i--);
}

int main(){
```

```
    rcc.ClockCmd(&gpioa, ENABLE);   // 使能 GPIOA
    pa8.Config(OUTPUT); // 配置 pa8 为输出

    while(1){
        delay(500000);  // 延时
        pa8 != pa8;     // 取反
    }
}
```

主程序做的事情很简单，使能 GPIOA，配置 pa8 (对应红色 LED)为输出，然后在死循环 while(1) 里面不断的延时取反. 这就是一个最基本的程序结构.

注:1. 这个程序使用了我自己建立的工程模板，如 rcc.ClockCmd 等"奇怪"的写法都是自己编写的简陋的库函数，大家不必须深究里面的内容，这些内容和本文的主题搭不上边的.

2. 这个工程模板中的 STM32 系统时钟的初始化过程在我的工程模板中已经被隐藏起来了，在代码启动的过程中是会自动进行这些基本的初始化的. 如果你需要把这里的代码移植到自己的工程上，就需要自己灵活的修改了，后面我就不再说明这两点注意事项了.

相信上面的代码所做的事情是很容易理解的，那么现在我要对程序提出新的要求：我需要板子上的另外一个灯也闪烁.

这个当然也很简单啦，只要稍稍修改一下程序就行了，另一个灯(绿色 LED)对应的引脚为 pd2，所以只要初始化一下，然后在 while 循环中再添加一句 pd2 取反的语句不就行了？

```
int main(){

    rcc.ClockCmd(&gpioa, ENABLE);   // 使能 GPIOA
    pa8.Config(OUTPUT); // 配置 pa8 为输出

    rcc.ClockCmd(&gpiod,ENABLE);
    pd2.Config(OUTPUT);

    while(1){
        delay(500000);  // 延时
        pa8 != pa8;     // 取反
        pd2 != pd2;
    }
}
```

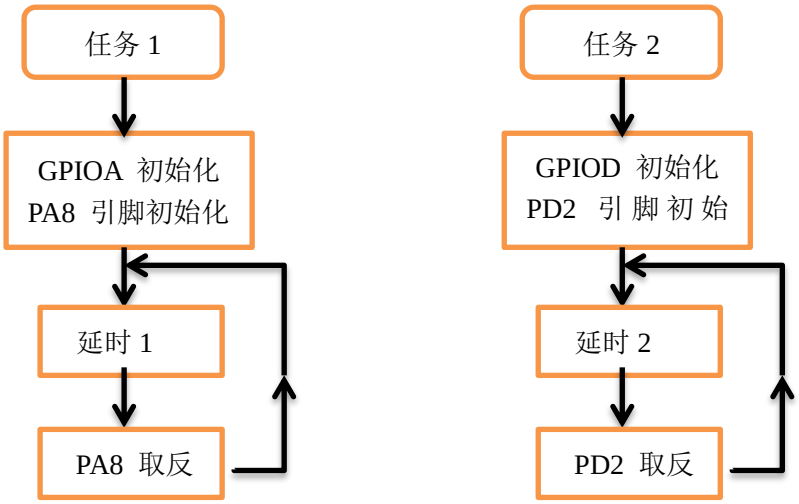
看到这里，都在对自己的 C 语言功底沾沾自喜吧. 可是，问题来了，**在实际中这两个灯所代表的含意是没有必要的关联的，但是这个的程序却把这两个没有任何逻辑关系的事情耦合到了一起.**

举个例子说，现在对程序又有了新的要求，红色 LED(pa8) 的频率保持原来的不变，而绿色 LED (pd2) 的频率需要变成原来的两倍，你打算把程序怎么改？当然如果你数学功底好的话，很快就会知道在怎么在 while(1) 里面修改参数，但是，如果每次修改 LED 的频率都需要改代码的多个部分，你累不累？更重要的是，是人就一定会犯错，修改以前的代码

往往就会带来新的 BUG，修改的代码越多，新 BUG 出现的可能性就越高！这不是什么程序员牛 B 不牛 B 的问题，这是数学上的必然的统计学规律！

而且，再进一步想，如果我有的 LED 不止两个，实际中可能是 3 个，4 个甚至更多．当然有经验的人会立刻想到使用一个定时器和一个全局的时钟基准来解决这个问题．我不否认这是个好办法．但是它能足够通用吗？最终的闪烁灯代码还是会放在一个 while(1) 里面，我们始终没有分离完两个全没有逻辑关系的任务！

其实，如果从一个更高的角度看待我们的程序要实现的功能，我们就可以把这人程序可以分成两个任务：



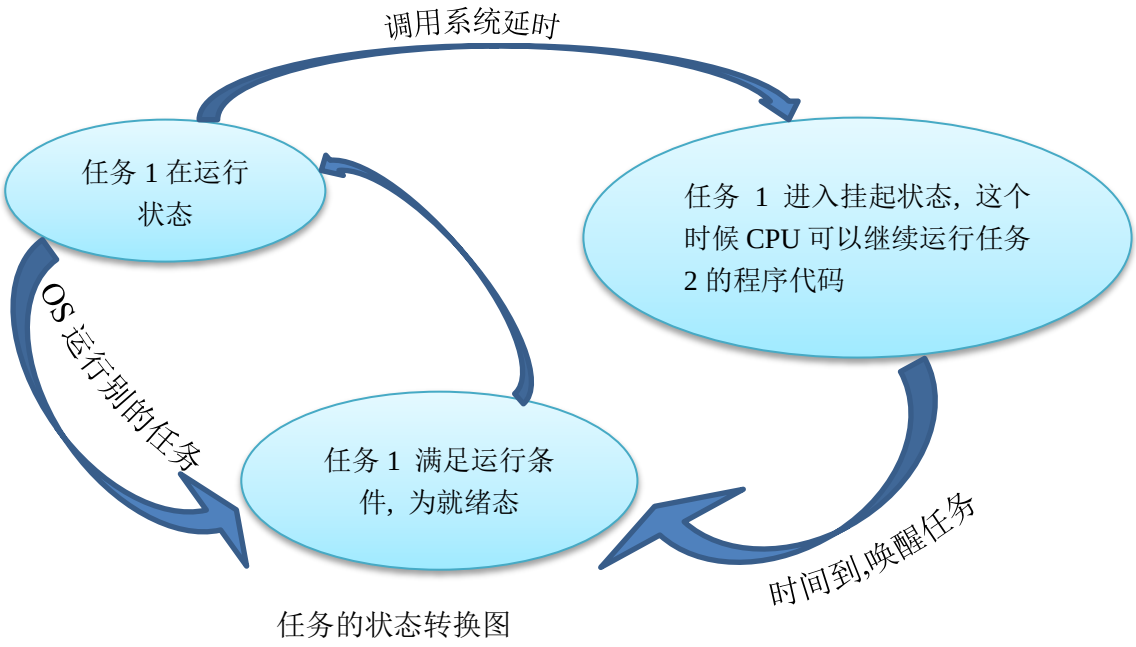
在两个任务没有必然的逻辑关系的情况下，这样看显然更加自然．所以我们更希望在程序中能所它们分离出来．

其实，这就是多任务操作系统的所要做的事情啦！也就是说，我们需要整出一个程序框架，让 CPU 一段时间执行任务 1，也就是让任务 1 获得执行权，隔一段时间执行任务 2，让任务 2 获得执行权，然后又执行任务 1 … 只要不停的在很的时间间隔做切换，看起来不就好像是任务 1 和任务 2 在同时执行了吗?

如果我们能整出这么一个框架，显然可以方便我们将来的程序开发，而且操作系统的另一个优势就是可以很好的利用系统资源．例如任务 1 因为需要延时，可以主动放弃执行权，让操作系统在 1s 后唤醒它，于是乎操作系统可以在任务 1 不运行的时候可以继续运行任务 2，而不是用 while(i--); 这样的软件延时来让 CPU 空转．

任务的状态如下图所示．这个状态图还是属于比较简易的形式，在复杂的操作系统中，通常可以分为 5 到 6 种状态．

这一节主要讲到的是操作系统的多任务，或者说并发性．关于操作系统的概念就讲到这里，其实还有很多概念我没有提到，例如并发性带来的同步问题，资源的共用带来的问题等等，有兴趣的可以去阅读操作系统相关的书籍，这些内容已经超出了本文的范围和我的能力啦 ：)



任务的状态转换图

2. 任务切换器的设计和实现

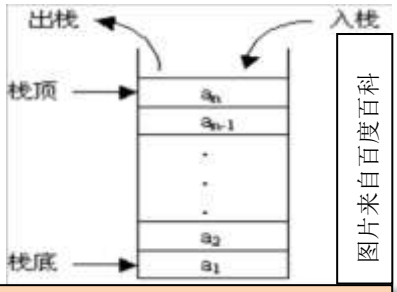
刚才讲到了我们需要整一个能让多个任务”同时”运行的框架. 那么这框架咋整呢? 仔细分析, 其实我们需要的最底层的东西就是 **CPU 使用权的控制和转移**. 至于怎么转移给谁, 在什么时候转移, 就属于调度算法的问题了. 也就是说, 在我看来, 任务调度器(scheduler)可以分成任务切换器和任务调度算法两个部分.

我们可以先不需要把算法搞得那么复杂, 要以用这样一个简单的方法: 现在只考虑两个任务的情况, 每隔一断时间就转换 CPU 的使用权. 这个时间间隔可以使用定时器中断产生. 那么剩下的问题就是 CPU 使用权的转移了.

那么转移 CPU 使用权我们需要做哪些工作呢?

首先, 对于从一个任务转移到另外一个任务, 可以通过在中断程序中, 把中断函数的返回地址设置到另外一个任务的代码上去. 这样, 在中断函数返回的时候, CPU 就跳转到另外一个任务去运行而不是在原来的任务上运行了.

对于任务来说, 它们都有自己的运行环境, 任务执行过程中通用寄存器里面的内容, 以及保存在栈里面的数据, 都属于任务的运行环境, 也是一个任务所私有的数据. 这就意味着, 在夺取它对 CPU 的使



什么是栈?

栈是一种先进后出的数据结构, 常常用于保存临时变量和返回地址. 例如函数 A 调用函数 B 的时候, 函数 A 会先把局部变量压入栈中, 保存函数 A 当前的状态, 然后才进入函数 B. 函数 B 干完它的事情以后又回到函数 A, 然后函数 A 再把之前保存的临时数据从栈中取出来, 继续使用. 栈在函数的嵌套调用中起到了至关重要的作用.

总结起来, 栈就是一种可以供嵌套使用的数据容器.

用权的同时，不可以破坏原任务的所有的数据，否则就无法切换回原来的任务了．那么我们就需要在进行任务切换的时候进行必要现场数据保护工作．而且，每个任务都必须使用不同的栈．

任务切换是在中断程序里面发生的，为了安全的转移 CPU 的使用权，我们就必须知道 CPU 在进入中断程序的时栈和各个寄存器是什么状态．有这个问题的详细内容，大家可以参考 ARM 公司的的资料 <Cortex-M3 R2P0 Technical Reference Manual> 中的第 5 章 Exceptions，也可以参考 <The definitive Guide to Cortex-M3> (这个有中文译本 <STM32 权威指南>) 中的 Chapter 9. Interrupt Behavior. 下面是我对这部分内容的简单介绍．

当要进入中断的时候，CPU 会先把寄存器 xPSR, PC, R0-R3, R12, LR 压入栈中进行保护．需要注意到的是，在 Cortex-M3 的通用寄存器中，还有 R4-R11 没有入栈，其实这样的做法是根据 ARM 的 C/C++过程调用标准决定的(C/C++ standard Procedure Call Standard for the ARM Architecture)，被压入栈的 8 个寄存器就是调用者自己保护的数据．不过我们现在不需要太关心这些寄存器是用来做什么的．最重要的是要知道，如果我们要切换任务，就还需要把 R4-R11 这 8 个寄存器压入栈中保护起来．右图就是刚进入中断时的栈的内容(注意 STM32 的栈是高地址往低地址生长的).

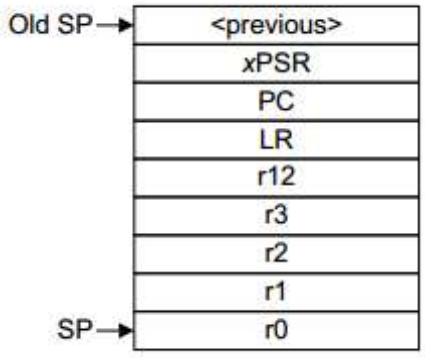
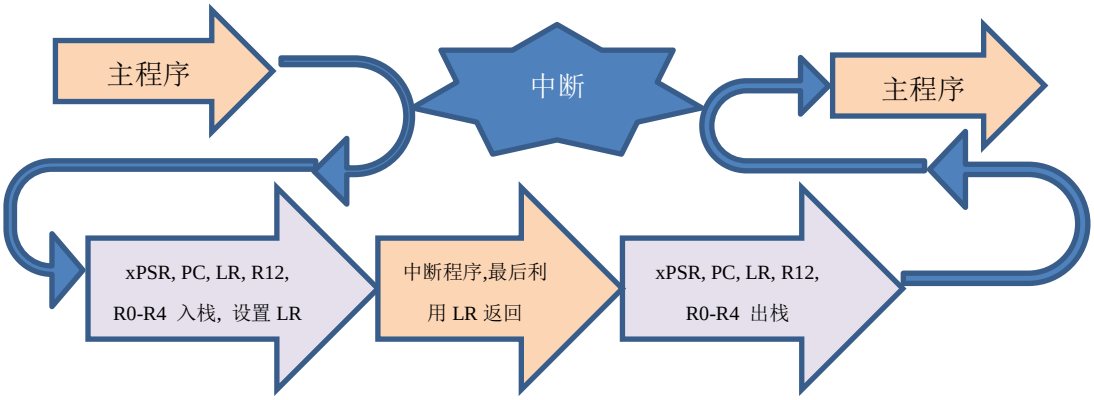


Figure 1 STM32 刚进入中断时的栈的内容

在 ARM 程序中，通常一个函数调用另外一个函数的时候会使用 BL 指令，这个指令会把当前的 PC(CPU 的程序计数器，或者叫程序指针，通常总指向下一条要被执行的指令)放到 LR 寄存器．当调用的函数返回时，就可以使用 “BX LR” 指令来把 LR 寄存器的内容复制到 PC 寄存器里面，从而实现函数的返回 (ARM 的 C/C++调用标准中规定在函数入口时，LR 寄存器中存放着函数的返回地址)．在中断出现的时候，LR 寄存器会设置成一个特殊的值，而不是中断之前的 PC 寄存器的内容．这样，当中断函数使用 BX LR 指令返回的时候，CPU 就能够知道是中断函数返回了，然后就取消 CPU 的中断状态，再把进入中断服务程序之前压入栈中的内容全部出栈，恢复原来寄存器中的内容，最终实现中断函数的返回．

下面是一个**简化后的中断流程图**．注意到主程序和中断程序我使用了一样的颜色，因为这些函数都是符合标准调用约定的函数．我们并没有使用特殊的指定来让编译器区别中断函数和非中断函数．每种 CPU 的中断函数的调用机制是不同的，例如在 51 单片机里面中断函数是需要用特殊的指令告诉编译器指定的函数是中断函数的.



有了这些信息，差不多就知道怎样进行任务切换了，但是千万不要急着写代码！最重要的是先列出在中断函数中应该做的事情，以及定义任务调度算法和任务切换器之间的接口，不先理清思路，很容易就会陷入苦逼调试的死循环的！**定时器中断函数的思路如下**：

1. **调用任务调度算法函数，由任务调度算法决定是否要切换任务，如果要切换任务，就返回新的任务栈的栈指针．如果不需要切换任务，就返回 0．任务调度算法接收当前任务栈顶作为参数，如果需要切换任务，那么就保存这个参数以供下次恢复任务使用．**
2. **如果任务调度算法函数说要切换任务，那么就直接任务切换，任务切换器的工作内容是：**
   1) **把 R4-R11 压入栈中，保护寄存器原来的内容．**
   2) **更新当前的栈指针为任务调度算法函数返回的栈指针．**
   3) **利用新的栈指针，从栈中恢复 R4-R11 的内容（pop {r4-r11}）．**
   4) **利用 bx lr 指令进行中断返回，实现任务切换．(返回地址保存在栈中，而我们把栈指针指向了另一个任务的栈，所以中断函数的返回地址就被我们偷偷地修改了)**
3. **如果不需要切换任务，直接利用 bx lr 指针进行中断返回．**

下面就是任务调度算法函数的代码了，为了实现上面的操作，C/C++ 本身是无能为力的，必须要使用汇编代码才能实现．这也是这篇文档中的唯一一块汇编代码，所以大家还是耐心的看吧：(因为我使用的是 C++ 环境，由于启动代码是用汇编写的，中断函数的名字需要以 C 语言的风格进行符号导出，所以前面加有 extern “C” 声明)

```
extern "C" __asm void TIM1_UP_IRQHandler(){

    import tick_and_switch  // 任务调度算法函数
    import clear_int_flag   // 定时器 1 中断清零函数
    REQUIRE8    // 加这两条对齐伪指令防止链接器报错
    PRESERVE8   // 8 字对齐

    mov r0, sp          // step 1
    sub r0, #(8*4)      // 调整为 r4-r11 压入栈后的状态
    push {lr}
    bl.w tick_and_switch    // 调用算法
    pop {lr}

    cmp r0,#0    // step 2
    beq end

    push {r4-r11}   // r4-r11 入栈
    mov sp,r0       // 更新 SP
    pop  {r4-r11}   // 用新的 SP 恢复 r4-r11

end             // step 3
    b.w clear_int_flag
```

```
}
```

为了简单，我就使用了大家相对比较熟悉的定时器1中断. 虽然 SysTick 中断的配置也很简单. 函数 clear_int_flag 是事先已经写好的定时器中断标志清零函数，并不是这里的重点大家不用过于关注. 注意到这里没有刚才提到的 “bx lr” 指令，其实这条指令是在定时器中断清零函数里. 而 “b.w clear_int_flag” 这条指令的意思是指跳转到定时器中断标志清零函数去执行但是不改变 LR 寄存器的内容. 所以定时器中断标志清零函数返回就相当于中断函数返回了.

至于调度算法，由于我们现在只两个任务的情况，所以每次都只返回另外一个任务栈的栈顶就行了. 需要注意问题的就是，我们的新任务的栈是需要经过初始化以后才能给任务切换器使用的. 因为任务切换器默认了，算法所返回的新的栈指针，指向的栈里面的内容，这些对应着刚才讲到的各个寄存器的内容，而且某些关键的值(比如对应 PC 的值)必须是合法的!

栈的初始化可以用一个函数来做，为了简单起见，这里使用定义一个全局的结构体变量来进行初始化:

```
typedef void(*ins_ptr)(void);
struct task_init_stack_frame {
    u32 r4_to_r11[8];       // lower address
    u32 r0;
    u32 r1;
    u32 r2;
    u32 r3;
    u32 r12;
    u32 lr;
    ins_ptr pc;
    u32 xpsr;               // higher address
  // 构造函数
    task_init_stack_frame(u32 r0_init, u32 lr_init, ins_ptr pc_init):
        r0(r0_init),lr(lr_init),pc(pc_init){}
};
void task(u32 a){ // 任务函数
  while(!a){
      delay(100000);
      pd2 = !pd2;
  }
}

task_init_stack_frame dummy(0,0,(ins_ptr)&task ); /* 为防止栈溢出而定义的数据空间 */
__align(8)  // arm 标准规定栈要 8 字节对齐
task_init_stack_frame base(0,0,(ins_ptr)&task );    // higher address
```

现在就可以编写我们的任务调度”算法”函数了，对于一个完整的操作系统来说，任务调

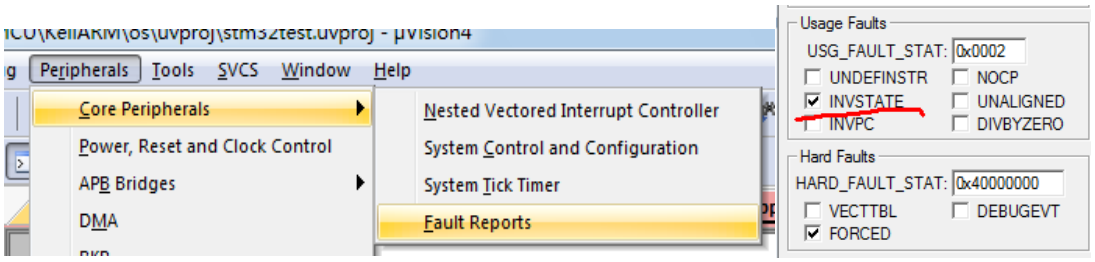
度算法是至关重要的一部分. 而这个文档主要是演示任务切换的部分, 只考虑两个任务的情况, 所以就简单的解决了.

```
/* 任务调度算法函数 */
extern "C" void* tick_and_switch(void* cur_stack){
    static void* next_stack = &base;
    void* temp = next_stack;      /* 只有两个任务交替 */
    next_stack = cur_stack;
    return temp;
}
```

这个时候主函数的 while 循环也就变成了只有一个任务了, 可以在主函数中进入 while(1)之前进行定时器的初始化, (具体的代码可以看**附件 2_os 工程**中的内容).

```
while(1){
    delay(500000);
    pa8 = !pa8;
}
```

如果现在进行软件仿真, 经过跟踪就会发现程序确实成功的切换到了 task 函数. 但是一到 task 函数后就立即跳到 HardFault_Handler 中断函数了. 那么出了什么原因导致的 HardFault 呢? 在 <The Definitive Guide to Cortex-M3> 7.5.5 Dealing with Faults 一节中有提到 , "During software development, we can **use the FSRs to determine the causes of errors** in the program and correct them." 然后通过 keil 的软件仿真时可能找到



打开再重新跟踪调试, 会发现进入 HardFault 瞬间有两个位被置位了, 来关注一下 INVSTATE, 还是去 The Definitive Guide to Cortex-M3 这本书里面搜索一下, 很快就可以找到那一位的意义了:

428 APPENDIX E

Table E.5 Usage Fault Status Register *Continued*

Bit	Possible Causes
INVSTATE (bit 1)	1. Loading branch target address to PC with LSB equals 0. Stacked PC should show the branch target. 2. LSB of vector address in vector table is 0. Stacked PC should show the starting of exception handler. 3. Stacked PSR corrupted during exception handling, so after the exception the core tries to return to the interrupted code in ARM state.

第 1 点, 看不懂. 第 2 点, 还是看不懂 … 不过话说回来, PC 指针是在 C++ 语言环境下

进行调整的的, C++ 是相对比较安全的, 先不管它. 关注一下第 3 点, PSR 寄存器, 我们一直没有搭理过它, 它不是通用寄存器, PSR 叫做 programming status register, 程序状态寄存器, 里面的每一位都有不同的含义, 具体细节我也不太想去了解, 最简单的解决方案就是抄袭一个合法的 PSR 的值来保存, 这个只要用软件仿真刚进入主函数时看看 PSR 寄存器的值, 然后复制一下就行了.

于是可以修改一下刚才的结构体, 加上红色部分:

```
typedef void(*ins_ptr)(void);

struct task_init_stack_frame {
    u32 r4_to_r11[8];       // lower address
    u32 r0;
    u32 r1;
    u32 r2;
    u32 r3;
    u32 r12;
    u32 lr;
    ins_ptr pc;
    u32 xpsr;               // higher address
  // 构造函数
    task_init_stack_frame(u32 r0_init, u32 lr_init, ins_ptr pc_init):
        r0(r0_init),lr(lr_init),pc(pc_init),xpsr(0x61000000){}
};
```

然后再编译程序仿真正常了!!! 下载也到板子上运行也正常了!!! 现象是两个灯同时闪烁, 绿色的 LED 的闪烁频率比红色 LED 闪烁频率快很多. 这就是一个多任务操作系统的雏形了!

嗯哼，为了呼应开头我所说的这里的代码 <甚至不能称之为OS>，所以我的任务到这里也就结束了. 也许你看了会想说: 尼玛 !!! 你这程序也太坑爹了吧!!! 

在开头也说了，这篇文档的主要目的是引导大家到一个开始，现在的代码自然存在很多的问题，而且很不完善，如果你有兴趣，就自己动手拓展你所需要的功能吧. 

3. 附录: 主要代码

```
#include "sys.h"
#include "stm32f10x__tim.h"
#include "IOstream.h"
using namespace periph;

// 软件延时
void delay(volatile int i){ while(i--); }

typedef void(*ins_ptr)(void);

struct task_init_stack_frame {
    u32 r4_to_r11[8];       // lower address
    u32 r0;
    u32 r1;
    u32 r2;
    u32 r3;
    u32 r12;
    u32 lr;
    ins_ptr pc;
    u32 xpsr;              // higher address
  // 构造函数
    task_init_stack_frame(u32   r0_init,   u32   lr_init,   ins_ptr
pc_init):
        r0(r0_init),lr(lr_init),pc(pc_init),xpsr(0x61000000){}
};
STATIC_ASSERT(sizeof(task_init_stack_frame)==(8*4+8*4));

void task(u32 a){  // 任务函数
    while(!a){
        delay(100000);
        pd2 = !pd2;
    }
```

```
}

task_init_stack_frame dummy(0,0,(ins_ptr)&task ); /* 为防止栈溢出而
定义的数据空间 */
__align(8)  // arm 标准规定栈要 8 字节对齐
task_init_stack_frame  base(0,0,(ins_ptr)&task  );      //  higher
address

/* 任务调度算法函数  */
extern "C" void* tick_and_switch(void* cur_stack){
    static void* next_stack = &base;
    void* temp = next_stack;     /* 只有两个任务交替 */
    next_stack = cur_stack;
    return temp;
}

int main(){

    rcc.ClockCmd(&gpioa,ENABLE);
    pa8.Config(OUTPUT);
    rcc.ClockCmd(&gpiod,ENABLE);
    pd2.Config(OUTPUT);

    rcc.ClockCmd(&tim1,ENABLE);
    TIM_BaseConfig cfg;
    cfg.prescaler = 720;
    cfg.autoReloadValue = 100;
    // 72 000 000 / (720*100) -> 1000 Hz 左右
    tim1.BaseConfig(cfg);
    tim1.UpdateEventIntCmd(ENABLE);
    tim1.CounterCmd(ENABLE);
    nvic.Config(TIM1_UP_IRQn, ENABLE);
    while(1){
        delay(500000);
        pa8 = !pa8;
    }
}

extern "C" __asm void TIM1_UP_IRQHandler(){

    import tick_and_switch  // 任务调度算法函数
    import clear_int_flag   // 定时器 1 中断清零函数
    REQUIRE8    // 加这两条对齐伪指令防止链接器报错
    PRESERVE8   // 8 字对齐
```

```
    mov r0, sp          // step 1
    sub r0, #(8*4)      // 调整为 r4-r11 压入栈后的状态
    push {lr}
    bl.w tick_and_switch    // 调用算法
    pop {lr}

    cmp r0,#0    // step 2
    beq end

    push {r4-r11}   // r4-r11 入栈
    mov sp,r0       // 更新 SP
    pop  {r4-r11}   // 用新的 SP 恢复 r4-r11

end             // step 3
    b.w clear_int_flag
}

extern "C" void clear_int_flag(){
    tim1.ClearUpdateEventIntFlag();
    return ;
}
```