

*400 行程序写一个 RTOS*

*----skaiuijing*

# 声明

本文档由本人的一系列博客整理而来，主要讲解 RTOS 原理及如何写一个小型的 RTOS。

在一开始，笔者只是在一些平台发表相关博客，后面有读者建议整理成相关文档，想来想去，文章数量过多确实有系统整理的必要，于是便有了本文档。

本文档将介绍 RTOS 的大致框架和驱动程序的编写及相关理论，带领读者使用四百行 c 语言及少量汇编代码完成一个小型的 RTOS 内核调度器及内存管理器，并详细讲解 RTOS 相关原理及一些知名开源 RTOS 源码。

与 MIT 协议类似，本书可被随意转发、修改、使用等等，只要标注作者及来源即可。

仓库地址：[https://github.com/skaiui2/SKRTOS\\_sparrow](https://github.com/skaiui2/SKRTOS_sparrow)

作者：skaiuijing

# 目录

## 目录

声明 .....	2
实时操作系统简介 .....	7
移植 RTOS .....	8
框架 .....	15
框图 .....	15
实现 .....	15
RTOS 的工作原理 .....	16
如何切换? .....	16
观察 .....	17
总结 .....	21
基本思想 .....	21
引言 .....	22
封装思想 .....	22
面向对象思想 .....	22
程序 = 数据结构 + 算法 .....	23
结语 .....	23
内存管理 .....	23
前言 .....	23
原因 .....	24
指针 .....	24
小内存管理算法 .....	25
malloc 分配 .....	25
分配时的策略 .....	26
free 释放 .....	26
总结 .....	27
代码实现内存管理 .....	28
初始化 .....	28
分配 .....	29
释放 .....	31
插入与合并 .....	31
问题 .....	34
GDB 调试 .....	34
前言 .....	34
调试错误的内存管理算法 .....	36
寻找案发现场 .....	36
栈回溯 .....	36

问题简化原则 .....	37
观察程序执行发现具体问题 .....	41
问题追踪原则 .....	41
keil 调试 .....	43
总结 .....	44
内核框架 .....	45
前言 .....	45
程序 = 数据结构 + 算法 .....	45
抢占式内核 .....	45
我们需要实现什么? .....	45
如何实现实时性? .....	46
调度器对象 .....	47
切换任务 .....	47
任务对象 .....	47
任务控制块 .....	48
栈对象 .....	49
总结 .....	52
arm cm3 架构 .....	53
前言 .....	53
封装与接口思想 .....	53
arm 架构 .....	53
中断与上下文切换 .....	55
任务的启动 .....	56
任务的保存与切换 .....	57
总结 .....	57
思考题 .....	57
时间触发系统设计 .....	58
前言 .....	58
sparrow 的运行 .....	59
任务对象 .....	59
总结 .....	62
调度器创建与启动 .....	63
前言 .....	63
调度器初始化 .....	64
调度器启动 .....	64
开启第一个任务 .....	67
实验 .....	69
总结 .....	69
实现多线程 .....	71
前言 .....	71
PendSV 中断 .....	71
PendSV: .....	72
实验验证 .....	78
总结 .....	79

问题.....	79
并发与竞态.....	80
前言.....	80
操作系统中的并发与竞态.....	80
临界区与原子操作.....	81
如何防止临界区被打断? .....	81
进入临界区.....	82
退出临界区.....	83
实验.....	83
总结.....	86
问题.....	86
调度策略.....	87
前言.....	87
进程的分类.....	87
Linux0.11 版本调度算法.....	88
RTOS 内核.....	90
调度算法.....	90
FreeRTOS 的调度算法.....	91
rt-thread 内核的调度算法.....	92
总结.....	95
实现调度算法.....	96
前言.....	96
修改程序.....	96
实验.....	98
实验现象: .....	100
总结.....	100
问题.....	100
时钟.....	101
前言.....	101
systick 时钟.....	101
如何使用时钟.....	103
定时上下文切换.....	103
延时阻塞态.....	104
如何设计算法? .....	104
如何解决溢出的问题? .....	104
总结.....	106
阻塞延时的实现.....	106
配置 SysTick 时钟.....	106
添加延时表.....	108
实验.....	111
实验现象.....	113
总结.....	113
问题.....	113
本文结语.....	113



# 实时操作系统简介

实时操作系统 (RTOS)，是一种在嵌入式开发中常见的操作系统，比较知名的有 FreeRTOS、Vxworks、qnx、zephyr 这些。操作系统的主要功能是封装丑陋的硬件并提供良好的接口，实时操作系统也不例外，它不仅能实现基本的任务管理功能，很多 RTOS 还提供了丰富的服务便于程序的开发，例如 zephyr，它提供了设备树这些类 Linux 开发的方式，还有 FreeRTOS，支持多种架构和不同编译器，还有 TCP/IP、http 库可供使用。

在嵌入式开发中，不仅有实时操作系统，也有非实时操作系统，例如 Linux、Android 和 Windows，毕竟每个操作系统都要编写大量的驱动程序，例如 Linux 内核，就有一千五百多万行的代码都是驱动程序。

大型非实时操作系统中，现在 Windows 驱动程序的开发已经接近绝迹，Linux 内核驱动程序的开发倒是如火如荼，Android 系统是在 Linux 内核上建立了一层 AOSP 框架，与 Linux 驱动开发是十分相似的。

Windows、Android 和 Linux 系统（例如 ubuntu 这些发行版），之所以是非实时操作系统，是因为它们对程序的执行时间并不具有严格的确定性。这些大型操作系统更注重用户的体验。开创了个人电脑与操作系统先河的微软，作为世界科技巨头，与每个人的生活都息息相关，正如笔者现在编写本书，就是基于 Windows10 的 office。

Windows 是人类迄今为止最复杂的软件工程，可能以后也很难超越。

至于实时操作系统，则更多是面向工业产品，像飞行控制等领域，对时间的确定性是有严格要求的。人对时间的敏感与 CPU 不同，在 Windows 上工作时卡顿数分钟对使用者来说可能无所谓，但在 1 毫秒内，CPU 已经执行了无数次程序，对工业界来说是无法忍受的。据说 C 语言之父丹尼斯在开发 UNIX 时，为了控制一个高速旋转的磁盘，特意为它编写了一个 pid 算法（一种控制算法），如果这个磁盘的 pid 算法失效了一瞬间，高速旋转的动能也会导致灾难。

所以非实时操作系统和实时操作系统，一个很大的区别就是面向的场景不同。实时性的本质，是一种确定性与可靠性。

好了，现在相信你已经知道了什么是实时操作系统，那么让我们开始写一个实时操作系统吧！

试一试！

开玩笑的，还是让我们先进行 RTOS 的移植吧。

# 移植 RTOS

好的，现在先让我们进行一个小实验：对 sparrow 进行移植并使用。

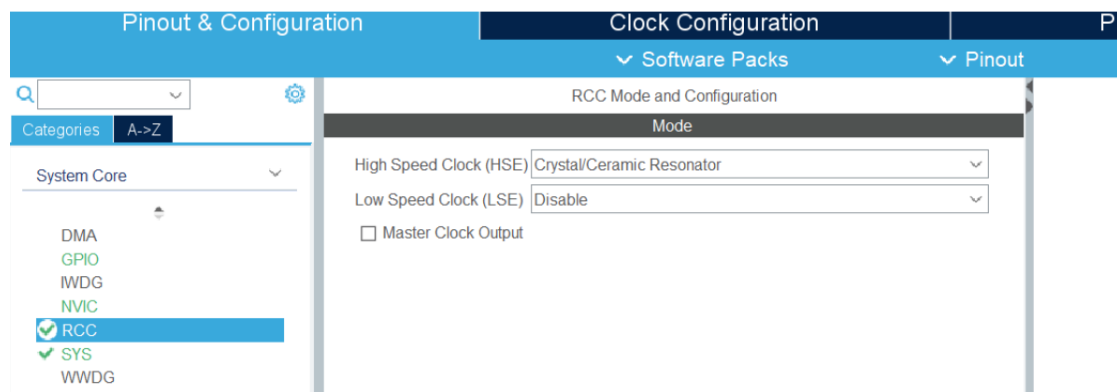
移植步骤与 FreeRTOS 的移植是差不多的。

移植并使用

读者需要一块单片机和一块下载器，比如 stm32f103c8t6 最小系统板。例如这种：

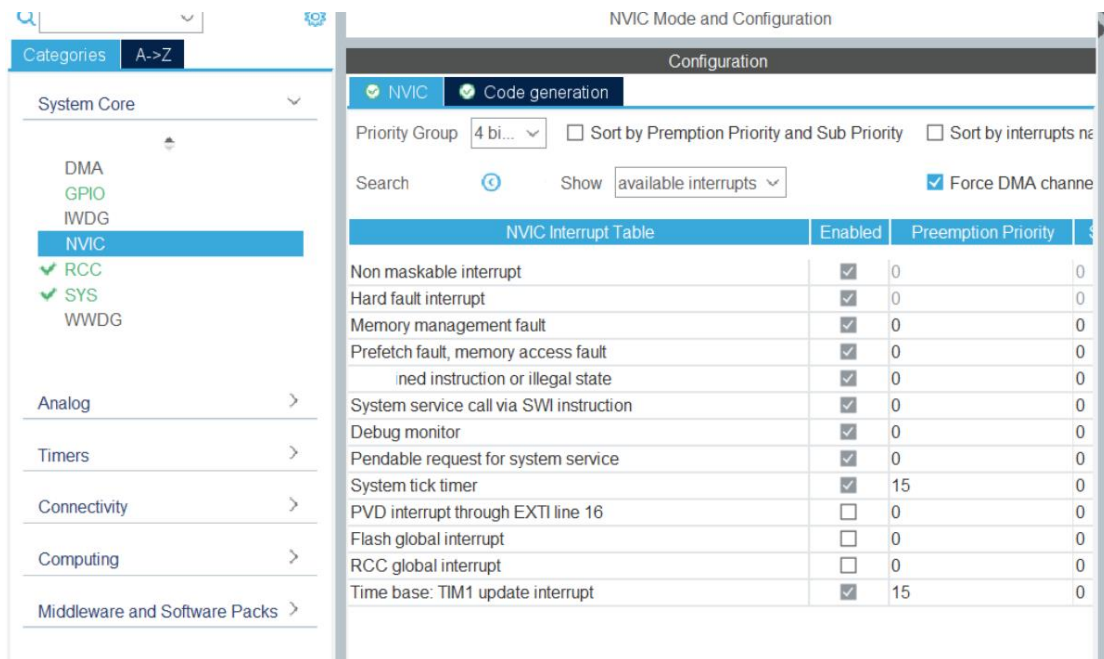


Rcc 设置如下：



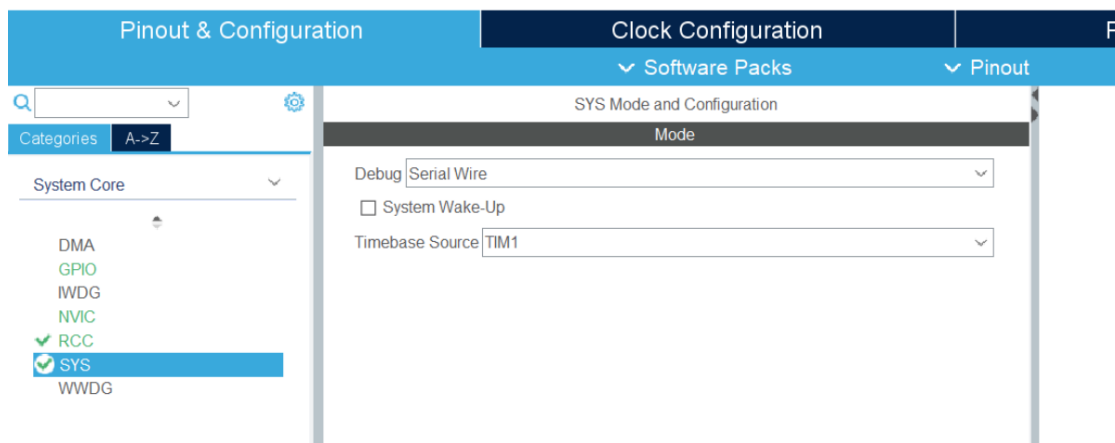
中断设置如下。当然，中断的设置并不是固定的，根据实际情况更改即可：





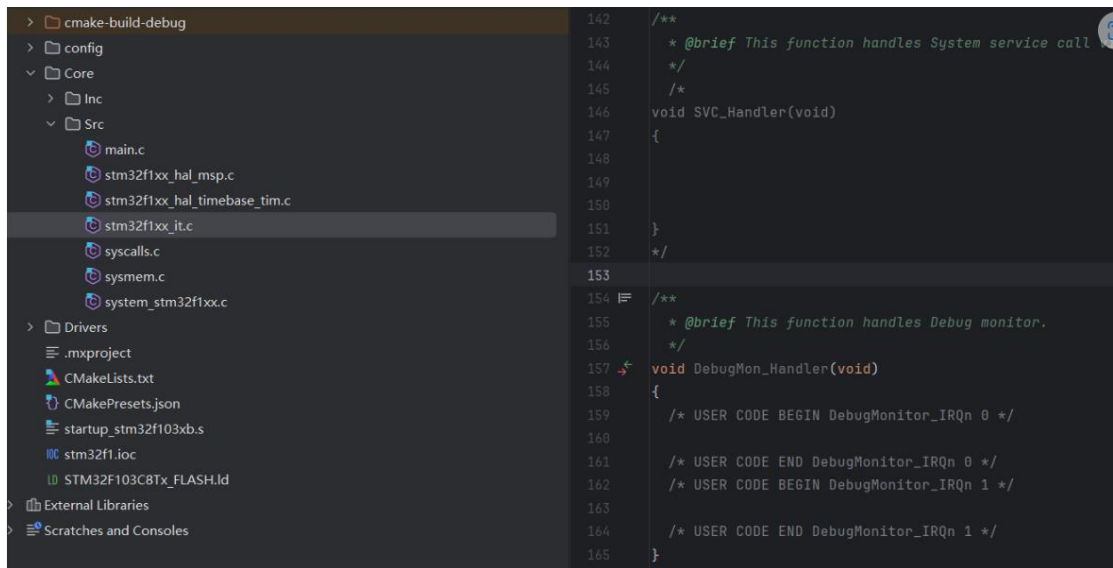
这里是重点!!!: 不要设置为 systicks 时钟, 这与 RTOS 是冲突的, 因为 RTOS 会使用 systick 时钟。

debug 要设置为 serial wire, 不然 mcu 下载后会卡死!

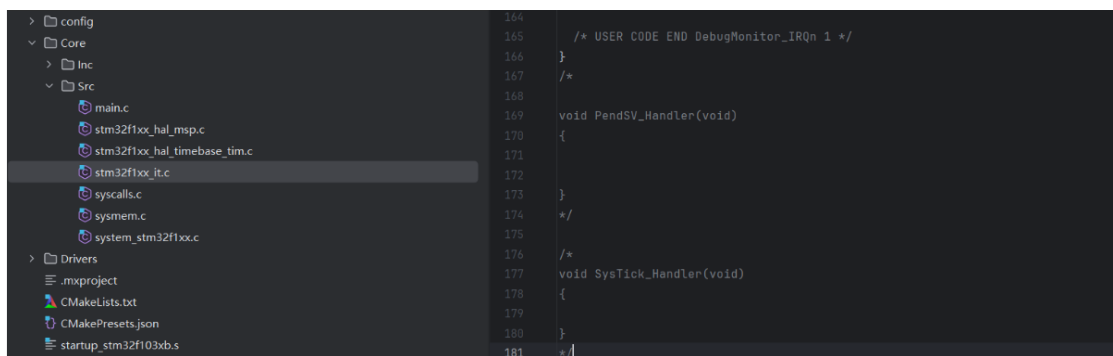


再把 PC13 设置为 output:





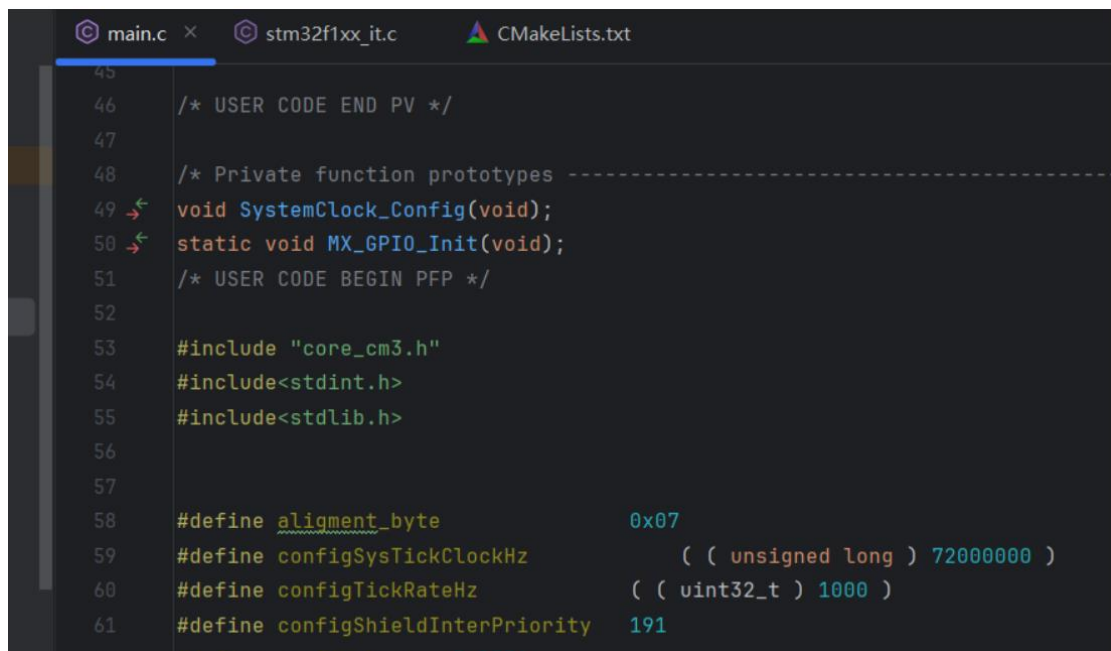
注释中断：



现在读者打开 sparrow 源码，找到 sparrow.c 文件，直接 ctrl a 然后 ctrl c 复制全部代码，然后粘贴到 main.c 中，例如粘贴在

```
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
```

这两行代码后面：



```
45
46  /* USER CODE END PV */
47
48  /* Private function prototypes -----
49  void SystemClock_Config(void);
50  static void MX_GPIO_Init(void);
51  /* USER CODE BEGIN PFP */
52
53  #include "core_cm3.h"
54  #include<stdint.h>
55  #include<stdlib.h>
56
57
58  #define alignment_byte          0x07
59  #define configSysTickClockHz    ( ( unsigned long ) 72000000 )
60  #define configTickRateHz        ( ( uint32_t ) 1000 )
61  #define configShieldInterPriority 191
```

然后在 task area 开头的注释下面添加这些代码:

//Task Area!The user must create task handle manually because of debugging and specification

```
TaskHandle_t tcbTask1 = NULL;
```

```
TaskHandle_t tcbTask2 = NULL;
```

```
void led_bright( )
```

```
{
    while (1) {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
        TaskDelay(1000);
    }
}
```

```
void led_extinguish( )
```

```
{
    while (1) {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
        TaskDelay(500);
    }
}
```

```
void APP( )
```

```
{

    xTaskCreate(    led_bright,
                  128,
```

```

        NULL,
        2,
        &tcbTask1
    );

    xTaskCreate(    led_extinguish,
                   128,
                   NULL,
                   3,
                   &tcbTask2
    );
}

```

main 函数这样写即可：



```

571 int main(void)
572 {
573     HAL_Init();
574     SystemClock_Config();
575     MX_GPIO_Init();
576
577     SchedulerInit();
578     APP();
579     SchedulerStart();
580
581     while (1)
582     {
583
584     }
585     /* USER CODE END 3 */
586 }

```

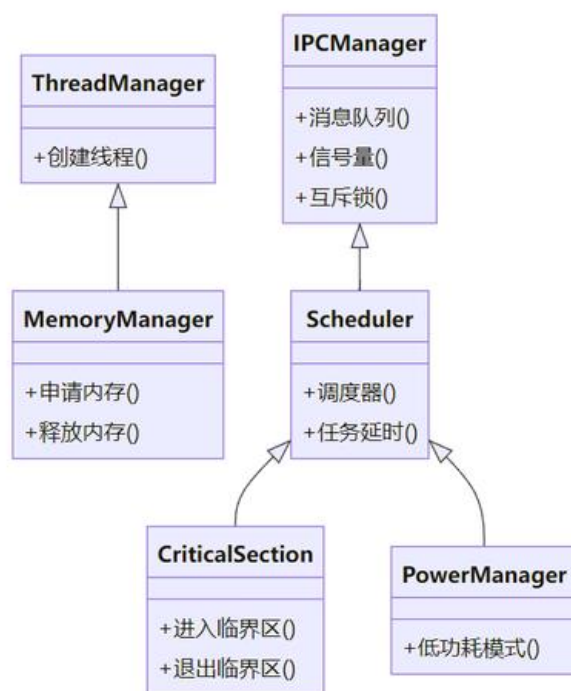
读者对代码进行编译，然后下载到单片机中，可以发现 stm32f103c8t6 上的 PC13 引脚开始闪烁。

以上就是 skRTOS\_Sparrow 的移植及基本使用，笔者在接下来将会讲解它的原理，并且指导读者如何一步步写出一个 RTOS！

# 框架

## 框图

笔者将会教读者手把手完成的这个 RTOS，它的总功能如下：



## 实现

对于调度层，它实现了动态内存管理、阻塞延时、多线程、临界区、优先级、低功耗空闲任务等功能。

sparrow 基本功能：多线程、多优先级、临界区、自定义空闲任务（默认是进入低功耗模式）、阻塞延时、动态内存管理。

# RTOS 的工作原理

RTOS，也就是实时操作系统，它与裸机的一个重大区别就是拥有了多线程。多线程，简单理解就是多个任务同时运行。

先介绍并行与并发的概念：

并行：简单理解就是多核 CPU 同时执行多个任务。

并发：单核 CPU 在一段时间内执行多个任务，达到同时执行多个任务的效果。

如何做到同时？

我们知道 mcu 在同一时间只能做一件事情，执行一个上下文的代码，在裸机中我们使用 while (1) 的形式来进行轮询，while (1) 中是任务，通常是执行完一个任务代表的函数再去执行下一个，一个任务代表执行的最小单元，那么我们可不可以把任务进行分割呢？

试想，如果 mcu 不断切换任务，那不就能做到看起来是在同时做两件事情吗？这当然是可以的。

并发的本质就是不断切换任务达到模拟同时的效果，它把任务分割为更小的单元，从更微观的层面分配任务执行的时间。

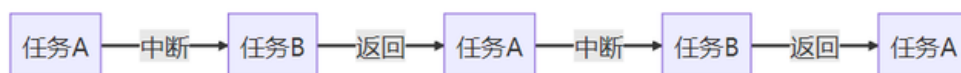
我们把切换任务的过程称为上下文切换。

## 如何切换？

读者想一想，在 arm 架构中，有什么东西能够持续执行，完成上下文切换的任务？

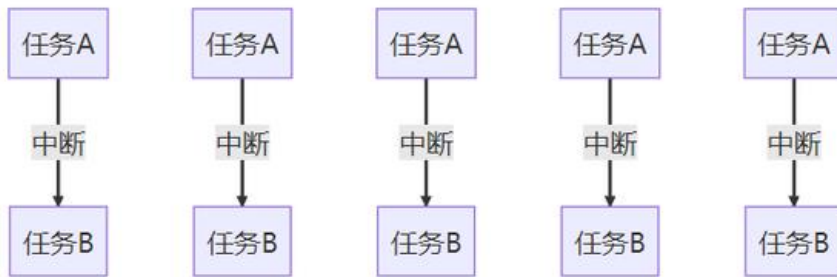
答案是：中断！

中断是一种异常，在发生中断时，单片机会保存现场，在中断完成后，会及时恢复现场。那么，假设只有两个任务，一个是正常执行的任务，一个是中断中的任务，假设中断不断发生，从微观层面上看，单片机的任务执行过程是这样的：



但是从宏观层面上，可不可以理解成这样呢？





把两个的任务的完成想象成两个增长的进度条，如果一个进度条结束后另一个进度条才开始，我们会察觉到先后顺序，这就不具有并发性。但是，如果我们把任务的尺度分割得更小，两个进度条交替运行，那么两个任务的完成度就像在同时增长一样。

读者发现没有，如果我们能利用中断不断分割任务执行的尺度，那么它不就看起来是并发的吗？

## 观察

在第一篇博客中我们已经完成了 Sparrow 的移植，现在让我们开始上手操作。

让程序先跑起来

先在 Sparrow 启动程序这里打上断点（笔者使用的调试软件是 gdb，读者也可以使用 keil，都是一样的）：

```
568      * @brief The application entry point.
569      * @retval int
570      */
571  int main(void)
572  {
573      HAL_Init();
574      SystemClock_Config();
575      MX_GPIO_Init();
576      SchedulerInit();
577      APP();
578      SchedulerStart();
579
580
581      while (1)
582      {
583
584      }
585
586  }
```

现在我们进入了启动程序内部：

```

496
497
498 void __attribute__(( always_inline )) SchedulerStart( void )
499 {
500     /* Start the timer that generates the tick ISR.  Interrupts are disabled
501     * here already. */
502     ( *( ( volatile uint32_t * ) 0xe000ed20 ) ) |= ( ( ( uint32_t ) 255UL ) << 16UL );
503     ( *( ( volatile uint32_t * ) 0xe000ed20 ) ) |= ( ( ( uint32_t ) 255UL ) << 24UL );
504
505     /* Stop and clear the SysTick. */
506     SysTick->CTRL = 0UL;
507     SysTick->VAL = 0UL;
508     /* Configure SysTick to interrupt at the requested rate. */
509     SysTick->LOAD = ( configSysTickClockHz / configTickRateHz ) - 1UL;
510     SysTick->CTRL = ( ( 1UL << 2UL ) | ( 1UL << 1UL ) | ( 1UL << 0UL ) );
511
512     /* Start the first task. */
513     __asm volatile (

```

通过注释，我们可以判断，这是在配置中断，联系上文笔者所说，上下文切换与中断息息相关，所以我们要去查找程序中相关的中断。  
继续执行程序，我们发现我们进入了一个由汇编程序组成的函数中，这是 SVC 中断，就是它开启了第一个任务：

```

235
236
237 #define vPortSVCHandler SVC_Handler
238 #define xPortPendSVHandler PendSV_Handler
239
240 void __attribute__(( naked )) vPortSVCHandler( void )
241 {
242     __asm volatile (
243         "    ldr r3, pxCurrentTCBConst2    \n"
244         "    ldr r1, [r3]                  \n"
245         "    ldr r0, [r1]                  \n"
246         "    ldmia r0!, {r4-r11}           \n"
247         "    msp psp, r0                   \n"
248         "    isb                           \n"
249         "    mov r0, #0                     \n"
250         "    msp basepri, r0                \n"
251         "    orr r14, #0xd                 \n"
252         "    bx r14                        \n"
253         "

```

继续执行，果然，它进入了第一个任务内部：

```

537     TaskDelay( ticks: 1000);
538 }
539 }
540
541 void led_extinguish( )
542 → {
543     while (1) {
544         HAL_GPIO_WritePin( GPIOx: GPIOC, GPIO_Pin: GPIO_PIN_13, PinState: GPIO_PIN_SET);
545         TaskDelay( ticks: 500);
546     }
547 }
548
549 void APP( )

```

为了观察上下文切换，程序中的中断是首要观察目标，我们在 xPortPendSVHandler 和 SysTick\_Handler 里面打上断点，继续执行，在执行完 TaskDelay 后，程序进入了 xPortPendSVHandler 中断函数内部：

```

253     "                                \n"
254     " .align 4                        \n"
255     "pxCurrentTCBConst2: .word pxCurrentTCB      \n"
256     );
257 }
258
259 void __attribute__(( naked )) xPortPendSVHandler( void )
260 {
261 → __asm volatile
262 (
263     " mrs r0, psp                    \n"
264     " isb                            \n"
265     "                                \n"
266     " ldr r3, pxCurrentTCBConst      \n"
267     " ldr r2, [r3]                   \n"
268     "                                \n"

```

现在让笔者告诉大家，它就是执行上下文切换的中断，在一定时间内它都会触发，完成上下文切换。

继续执行，程序来到了 vTaskSwitchContext，看函数名称读者便知道，上下文切换要完成了：

```

461     return TopZeroNumber;
462 }
463
464 void vTaskSwitchContext( void )
465 {
466 → pxCurrentTCB = TcbTaskTable[ FindHighestPriority()];
467 }
468
469 void EnterSleepMode(void)
470 {
471     SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
472     __WFI();
473 }

```

对于单片机来说，一个周期内能做无数次事情，我们继续一行行执行，不知道要执行多久，所以我们直接快速运行程序（gdb 使用 c 命令运行到下一个断点处），我们发现我们来到了 systick 中断这里：

```
402 }
403
404
405 void SysTick_Handler(void)
406 {
407     uint32_t xre = xEnterCritical();
408     CheckTicks();
409     xExitCritical( xReturn: xre);
410 }
411
412 uint32_t * pxPortInitialiseStack( uint32_t * pxTopOfStack,
413                                     TaskFunction_t pxCode,
```

对 systick 有了解的朋友知道，它是一个时钟，也就是说，它会在一定时间周期触发。

继续快速运行，我们发现我们又来到了 pendsv 中断内部：

```
255     "pxCurrentTCBConst2: .word pxCurrentTCB\n"
256     );
257 }
258
259 void __attribute__(( naked )) xPortPendSVHandler( void )
260 {
261     __asm volatile
262     (
263         " mrs r0, psp\n"
264         " isb\n"
265         " ldr r3, pxCurrentTCBConst\n"
266         " ldr r2, [r3]\n"
267         " stmb r0!, {r4-r11}\n"
268         " str r0, [r2]\n"
269         " stmb sp!, {r3, r14}\n"
270     )
```

难道说，systick 中断会在一定周期触发 PendSV 中断，从而不断进行上下文切换吗？

是的，这是正确的！

## 总结

通过对 RTOS 原理的一些简单介绍以及上手调试，我们大概知道了 RTOS 的运行原理：通过中断来进行上下文切换，同时 systick 中断会不断触发 PendSV 中断，使各个任务不断进行切换，从而模拟线程同时运行。

思考题

为什么要使用中断触发中断的方式？把两个中断的内容放在同一个中断里面不好吗？

## 基本思想

# 引言

接下来介绍本系列教程的基本思想。

如果读者阅读过 Sparrow 的源码，就会发现里面除了 c 语言，还有一大堆汇编和二进制，这些奇奇怪怪的符号，光是看着就让人感觉头疼了，难道学习写 RTOS 之前自己必须去学习汇编吗？还是说教程中会引入一大堆古老的汇编语言，但是学习这种老旧的汇编语言有什么用呢？

笔者认为：不用担心这个问题！本系列教程只要有 c 语言基础即可，不过剩下深入的内容需要读者自己去发掘。

## 封装思想

为了避免杂乱的汇编影响读者写 RTOS 的体验，笔者将会简单介绍封装思想。

封装，简单理解就是将一些程序看作是一个个的黑匣子，不去关心它内部的实现细节，而将重点放在它能实现的功能上。

应用软件下是操作系统，操作系统下是指令集，而指令集又不过是硬件提供的接口。如果我们在面对问题时不使用封装思想，这样一层又一层下去，你怎么可能学得会呢？

所以，如果读者没有学过汇编，也不想为了写 RTOS 而花费精力学习汇编语言，那么面对汇编与各种奇怪的地址，请使用封装思想！不需要去关注其内部实现的具体细节，而应当把重点放在它实现的功能上。

当然，考虑大家不同的基础与需求，笔者也是会给出大部分汇编语言的具体解释的，精确到每一行指令到底干了什么事情。但笔者不会放在本系列教程中，因为笔者之前有一些分析 RTOS 内部汇编的博客，例如对 SVC、PendSV 上下文切换内部的每一条指令的解释。当讲到有关汇编的程序时，笔者会介绍其封装实现的功能，同时给出具体分析汇编指令的链接，以供有兴趣的读者阅读。

## 面向对象思想

一个 RTOS，内部必然是复杂的，所以我们需要有模块化的思想，面向对象就是一种很好的模块化思想。

简单介绍一下面向对象。

面向对象，就是将软件设计为由多个“对象”组成来简化代码的开发和维护。

打个比方，A 让 B 去帮她拿一本书。在这里，A、B 就是两个不同的对象，”拿一本书“是 B 的一个方法，A 让 B 去做事情，这是 A 的一个方法。

这样一看，原本混乱的逻辑顿时有了条理，我们可以发现，任何程序都可以看成是不同对象相互作用的结果，这有助于我们开发模块化的程序，例如，如果这句话变成了：A 让 B 去帮她拿书包。我们不必重新描述这个过程，只需要在 B 的方法里面添加“拿书包”这个方法，而不用改动 A。

# 程序 = 数据结构 + 算法

程序 = 数据结构 + 算法，这句话相信很多读者都听过。其实读者发现没有，如果你把数据结构看成对象，把算法看成方法，这是不是与面向对象思想有一些接近？

笔者建议，读者在编写程序时，可以尽可能的对结构体进行封装，可以将各种数据结构看出对象，基本的算法（例如增删查改）都可以看作这个对象的方法。其实链表是一个很好的理解 程序 = 数据结构 + 算法 的素材，不过笔者的 Sparrow 以小巧和简洁为目标，所以没有使用链表，因为使用后肯定会远远超过 400 行代码了，实在是负担不起，而且也没这个必要。

## 结语

笔者简单介绍了本系列教程的基本思想，这些都需要读者细细体会。下一篇博客，笔者将会带领大家开始写第一行程序了。

第一个开始的部分，是动态内存管理部分，笔者将会教大家写一个更适合单片机体质的内存管理算法，类似于 c 语言中的 malloc 和 free 函数

## 内存管理

### 前言

本节我们将会开始具体的程序编写。我们首先要开发的模块是内存管理算法部分。为什么要从它开始呢？

## 原因

我们在 c 语言中，经常要用到 malloc 申请内存，然后再 free 释放内存，但是 c 语言标准库的 malloc 放在 RTOS 中是相当危险的，因为它并没有针对实时性进行优化，所以，为 RTOS 设计一个内存管理算法是有必要的。

## 指针

先简单复习一下指针：

指针的本质是要求值必须为地址的变量。

int a, 代表在内存中开辟一片空间，空间存放 a 的值。int \*a, 代表在内存中开辟一片空间，空间存放 a 的值，只不过此时 a 的值是一个地址，可以通过这个值找到另一块内存空间。

当我们进行 int \*a = malloc(sizeof(xxx)) 的操作时，实际上就是改变这片空间存储的值。此时 a 就是一个有意义的值，它代表一个存在且可以被存放的内存空间的地址。

那么，当我们随机定义一个指针并且要对这片内存操作时，会发生什么呢？例如：

```
int *a;
```

```
*a = 1;
```

当我们定义指针 a 时，a 的值是随机的，那么我们解引用 a，计算机找不到这片空间，此时错误就会发生了。

也就是说，如果我们要手动管理内存，那么必须要分配空闲的内存地址，必须对内存进行严格的管理。

## 写下第一行程序

sparrow.c

```
#include "core_cm3.h"
```

```
#include<stdint.h>
```

```
#include<stdlib.h>
```

```
#define config_heap 8*1024
```

```
#define Class(class)    \
typedef struct class class;\
struct class
```

```
static uint8_t AllHeap[config_heap];
```

如上所述，我们需要定义一个大数组。它由编译器分配，位于内存的全局区。这就是我们手动管理的内存空间。

简单介绍一下 Class 这个宏，我们知道 c 语言宏定义的本质是替换，可以看看替换前后：

替换前：



```
Class(Person) {  
    int a;  
};
```

替换后：

```
typedef struct Person Person;  
struct Person {  
    int a;  
};
```

这是笔者习惯性的一种用法，也是对自己的一种编程规范，目的就是告诉自己或者阅读代码的人：“这些代码是采用面向对象思想编写的，请尽量不要使用面向过程的思维来思考程序。”

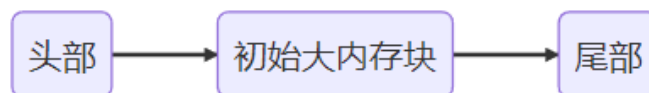
## 小内存管理算法

小内存管理算法，通常也称为内存分配器，用于在内存资源有限的嵌入式系统中高效地管理内存。这些算法的目标是减少内存碎片，提高内存利用率，并确保分配和释放内存的操作尽可能快速、稳定。

它采用不断分割的方法对内存进行动态分配，分为初始化，分割，释放三个步骤，为了简洁起见，笔者直接放图：

初始化

我们需要把定义的大数组先初始化：



## malloc 分配

分配内存时，根据需要分配空间的大小，返回对应的地址，最终内存块会被不断切割：



最终读者会看到，空闲内存块会越来越小，而且也越来越不连续。

虽然通过指针，通常情况下我们可以把不连续的空间当成连续的空间使用，这些不连续的空间被称为碎片。但是，当不连续的空间越来越小时，我们很可能找不到一个足够大的空间碎片满足存储请求，尽管总的空闲空间可能仍然满足。这样下去，系统绝对是会崩溃的。

为了避免碎片问题，也有一些方法被广泛采用。

## 分配时的策略

为了减少碎片，通常有两种策略：best-fit 和 first-fit。

best-fit 算法：这个策略趋向于将大的碎片保留下来满足后续的请求。我们根据空闲空间块的大小，将它们分在若干个容器中，在容器中，存储块按照他们的大小进行排列，这使得寻找 best-fit 块变得更加容易。

first-fit 算法：在这个策略中，对象被放置在地址最低的、能够容纳对象的碎片中。这种策略的优点是速度快，但是内存分配性能比较差。

## free 释放

释放内存块，就是将内存块加入空闲内存块表。同时会有一些方法减少碎片。

分配时的策略

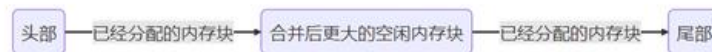
合并内存块

同时，如果我们检测到 free 释放的内存块的地址与其他空闲内存块是连续的，我们会合并它们：

合并前：



合并后：



这是内存管理算法的整体思想。

定义算法中的结构体

现在，我们已经知道内存管理包含哪些对象了，让我们接着写下这些程序：

sparrow.c

```
#define alignment_byte 0x07
```

```

#define MIN_size      ((size_t) (HeapStructSize << 1))

Class(heap_node) {
    heap_node *next;
    size_t BlockSize;
};

Class(xheap) {
    heap_node head;
    heap_node *tail;
    size_t AllSize;
};

xheap TheHeap = {
    .tail = NULL,
    .AllSize = config_heap,
};

static const size_t HeapStructSize = (sizeof(heap_node) +
(size_t)(alignment_byte)) &~(alignment_byte);

```

每个内存块都会有一个头部，也就是 heap\_node，同时为了管理整个内存块，我们还要定义一个 xheap 并且对其进行初始化。MIN\_size 的值头部字节对齐后的值的两倍，是为了判断分配的内存块的空间被切割后，剩下的内存能够不能够放得下头部 heap\_node 并有一部分存储空间，如果能，那么就分割这块内存，如果不能，那么就不分割。

至于 &~(alignment\_byte)，代表八字节对齐。因为 alignment\_byte 是 0x07，所以进行&~操作后的数会八字节对齐。我们通常使用 a%8 来判断这个地址是否八字节对齐，但是某些情况下，可以使用 a&(alignment\_byte)代替 a%8，详细内容在第七章里。

## 总结

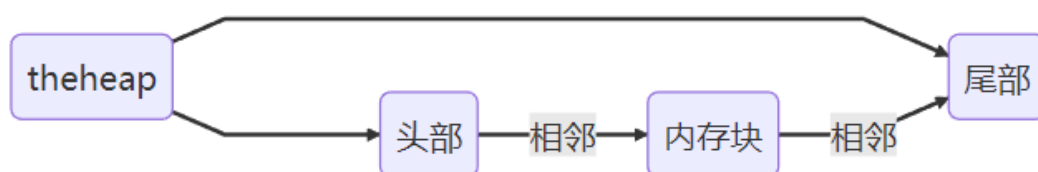
本章笔者简单介绍了内存管理算法的基本概念，并且定义了一些基本的结构体和宏，希望读者能够领会内存管理算法的精髓。

# 代码实现内存管理

上一篇笔者讲了内存分配的算法，是适合让我们一步步实现这些算法了，一共有四个函数，分别是初始化、分配、释放、插入。

## 初始化

初始化后的内存分布如图所示：



简单来说，就是我们要在大内存数组起始地址和终结地址分别创建两个 `heap_node` 节点，用于管理这个内存块，然后将它们使用指针连起来。还记得上一篇博客中我们定义的 `theheap` 吗？它负责记录头部和尾部。

像 `&= ~` 这些操作是字节对齐，字节对齐可以提高 CPU 运行速度，也可以防止 error 的发生。

CPU 会根据数据类型的不同，访问时取不同的连续的字节数，未对齐的数据类型可能会导致灾难的发生（不过编译器通常会帮助我们优化这些工作，例如数据类型转换、字节对齐等等）

Sparrow 中最大的数据类型是 `uint_32`，也就是 4 字节，但是是考虑浮点数，还是采用八字节对齐。

另外还要给读者介绍一点小知识：对于正整数  $X$ ， $Y = 2^n$ ， $X \% Y$  可以被  $X \& (Y - 1)$  代替。

举个例子， $X \% 4 = X \& 3$ 。

证明过程不难理解：

我们都知道二进制左移一位代表乘以 2 倍，那么对于  $2^n$ ， $X$  中比  $n$  位高的位的都是  $2^n$  的倍数，所以  $X \% (2^n)$  的结果必然是  $X$  低  $n$  位的值。既然我们需要低  $n$  位的值，那么直接  $X \& (2^n - 1)$  就可以得到  $X$  低  $n$  位的值了。

程序如下：此时，`firstnode` 就是头部。

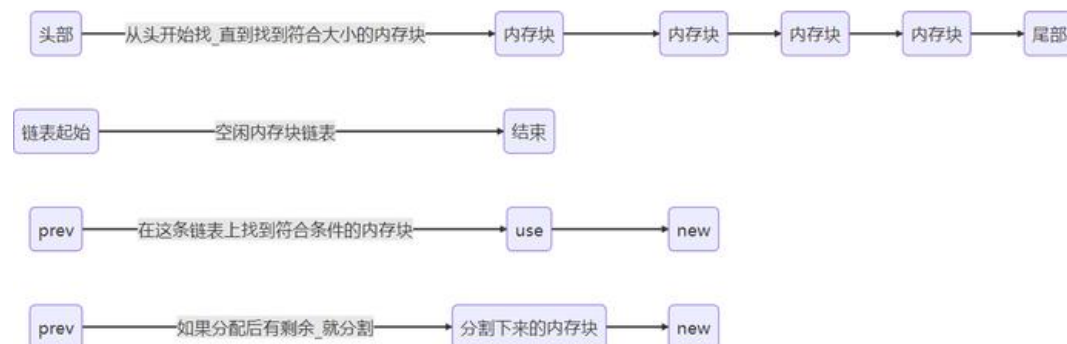
```

void heap_init( void )
{
    heap_node *first_node;
    uint32_t start_heap ,end_heap;
    //get start address
    start_heap =(uint32_t) AllHeap;
    if( (start_heap & alignment_byte) != 0){
        start_heap += alignment_byte ;
        start_heap &= ~alignment_byte;
        TheHeap.AllSize -= (size_t)(start_heap -
(uint32_t)AllHeap); //byte alignment means move to high address,so sub
it!
    }
    TheHeap.head.next = (heap_node *)start_heap;
    TheHeap.head.BlockSize = (size_t)0;
    end_heap = start_heap + (uint32_t)TheHeap.AllSize -
(uint32_t)HeapStructSize;
    if( (end_heap & alignment_byte) != 0){
        end_heap &= ~alignment_byte;
        TheHeap.AllSize = (size_t)(end_heap - start_heap );
    }
    TheHeap.tail = (heap_node *)end_heap;
    TheHeap.tail->BlockSize = 0;
    TheHeap.tail->next =NULL;
    first_node = (heap_node *)start_heap;
    first_node->next = TheHeap.tail;
    first_node->BlockSize = TheHeap.AllSize;
}

```

## 分配

分配的算法如下，程序使用的是 first-fit 算法：



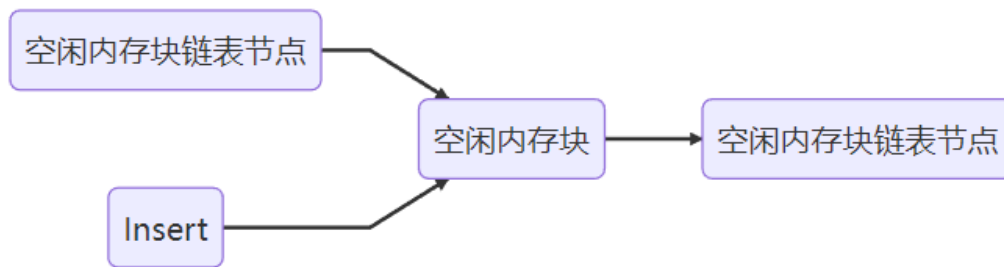
在初始化后,空闲内存块已经形成了一条链表,这条链表被称为空闲内存块链表,内存块被使用就被踢出这条链表,被释放就加入这条链表,如果内存块大于要申请的大小,就把多余的部分切割,然后加入这条内存链表。  
malloc 会找到合适的内存块,并且返回它的地址,源码如下:

```
void *heap_malloc(size_t WantSize)
{
    heap_node *prev_node;
    heap_node *use_node;
    heap_node *new_node;
    size_t alignment_require_size;
    void *xReturn = NULL;
    WantSize += HeapStructSize;
    if((WantSize & alignment_byte) != 0x00) {
        alignment_require_size = (alignment_byte + 1) - (WantSize &
alignment_byte); //must 8-byte alignment
        WantSize += alignment_require_size;
    } //You can add the TaskSuspend function ,that make here be an
atomic operation
    if(TheHeap.tail == NULL ) {
        heap_init();
    } //Resume
    prev_node = &TheHeap.head;
    use_node = TheHeap.head.next;
    while((use_node->BlockSize) < WantSize) { //check the size is fit
        prev_node = use_node;
        use_node = use_node->next;
        if(use_node == NULL) {
            return xReturn;
        }
    }
    xReturn = (void*)( ( (uint8_t*)use_node ) + HeapStructSize );
    prev_node->next = use_node->next ;
    if( (use_node->BlockSize - WantSize) > MIN_size ) {
        new_node = (void *) (((uint8_t *) use_node) + WantSize);
        new_node->BlockSize = use_node->BlockSize - WantSize;
        use_node->BlockSize = WantSize;
        new_node->next = prev_node->next;
        prev_node->next = new_node;
    } //Finish cutting
    TheHeap.AllSize -= use_node->BlockSize;
    use_node->next = NULL;
    return xReturn;
}
```

## 释放

当有内存块需要释放时，直接插入空闲内存块链表即可即可。

插入前：



插入后：



程序如下：

```
void heap_free(void *xReturn)
{
    heap_node *xlink;
    uint8_t *xFree = (uint8_t*)xReturn;

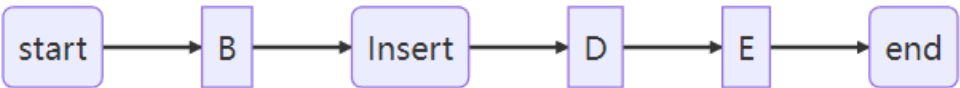
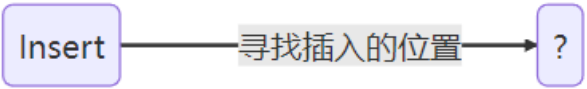
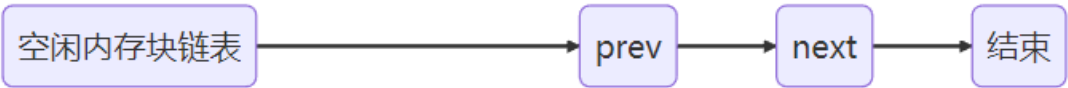
    xFree -= HeapStructSize;//get the start address of the heap struct
    xlink = (void*)xFree;
    TheHeap.AllSize += xlink-&gtBlockSize
    InsertFreeBlock((heap_node*)xlink);
}
```

InsertFreeBlock 不仅实现插入，同时实现合并内存块

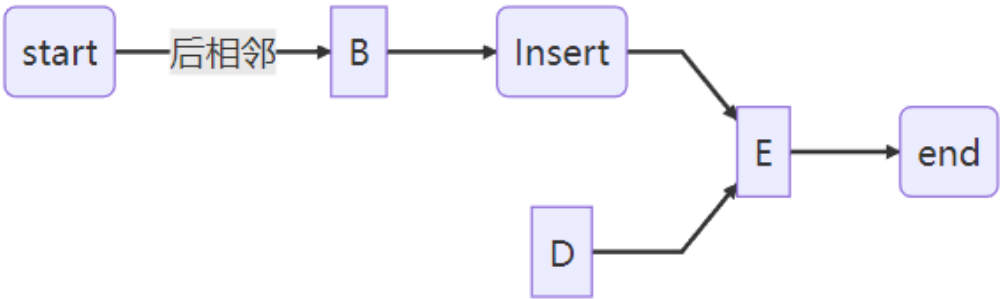
## 插入与合并

如果我们检测到 free 释放的内存块的地址与其他空闲内存块是连续的，我们会

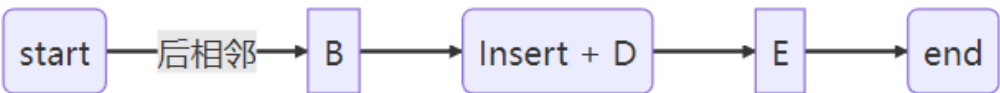
合并它们，这里是内存管理算法的精髓所在。



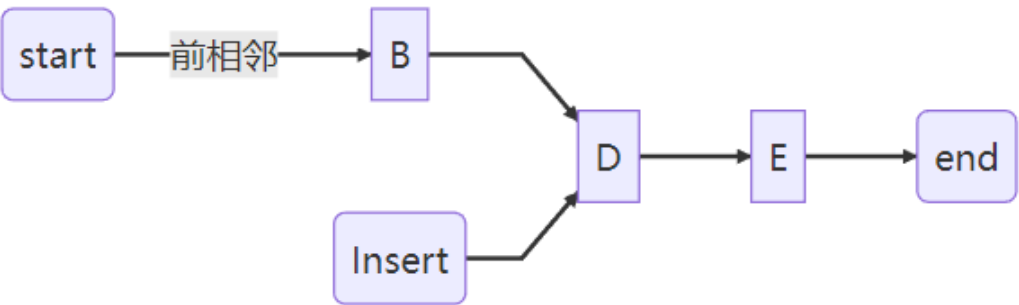
后相邻插入合并：



等价于：



前相邻插入合并：



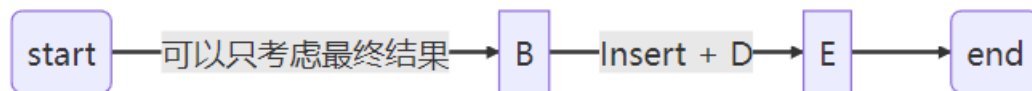


等价于：

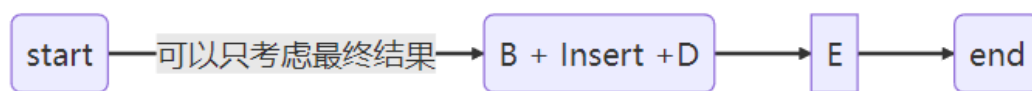


前后都相邻插入合并：

注：B 直接指向 E



等价于：



共有三种情况：前相邻、后相邻、前后都相邻。

如果前后相邻是相互独立的，那么互不影响，如果不独立，通过后，前的顺序可以得到前后都相邻的结果，因此该顺序正确。

所以程序中，我们先进行后相邻合并判断，再进行前相邻合并判断。

可能读者看不懂这些图是什么意思，请你对照着代码画一下图片，描绘各个节点的关系，然后你再来看一下我的图片，相信你一定会豁然开朗。

程序如下：

```
static void InsertFreeBlock(heap_node* xInsertBlock)
{
    heap_node *first_fit_node;
    uint8_t* getaddr;

    for(first_fit_node = &TheHeap.head; first_fit_node->next <
xInsertBlock; first_fit_node = first_fit_node->next)
    { /*finding the fit node*/ }

    xInsertBlock->next = first_fit_node->next;
    first_fit_node->next = xInsertBlock;

    getaddr = (uint8_t*)xInsertBlock;
    if((getaddr + xInsertBlock->BlockSize) ==
(uint8_t*)(xInsertBlock->next)) {
        if (xInsertBlock->next != TheHeap.tail) {
            xInsertBlock->BlockSize += xInsertBlock->next->BlockSize;
            xInsertBlock->next = xInsertBlock->next->next;
        }
    }
}
```

```

        } else {
            xInsertBlock->next = TheHeap.tail;
        }
    }
    getaddr = (uint8_t*)first_fit_node;
    if((getaddr + first_fit_node->BlockSize) == (uint8_t*)
xInsertBlock) {
        first_fit_node->BlockSize += xInsertBlock->BlockSize;
        first_fit_node->next = xInsertBlock->next;
    }
}

```

## 问题

小内存管理算法产生碎片是必然的，但是我们看到现代 OS 使用页表的形式进行映射，我们能不能参考页表，使用位映射内存块的形式设计内存管理算法从而解决碎片问题？

# GDB 调试

## 前言

上一篇笔者讲完了内存管理算法的完整实现，为了让读者面对 bug 时不至于茫然不知所措，笔者觉得有必要介绍一些调试方法。（不仅仅是 gdb，也包括 keil，由于 keil 偏图形化，不需要过多介绍，所以笔者会重点讲 gdb 命令）

## gdb 调试

本小节笔者会介绍各种 gdb 的小命令，例如 b, s, p, n, list, x/10x, disassemble, bt 等等命令。也介绍栈回溯等技术。

gdb 是一个非常强大的工具，但是笔者发现网上没有什么教程讲使用 gdb 调试嵌入式程序，而且发现很多人对调试技术并不重视，所以笔者觉得有必要讲一讲 gdb 的使用。

让我们在实战中学习 gdb 的使用！

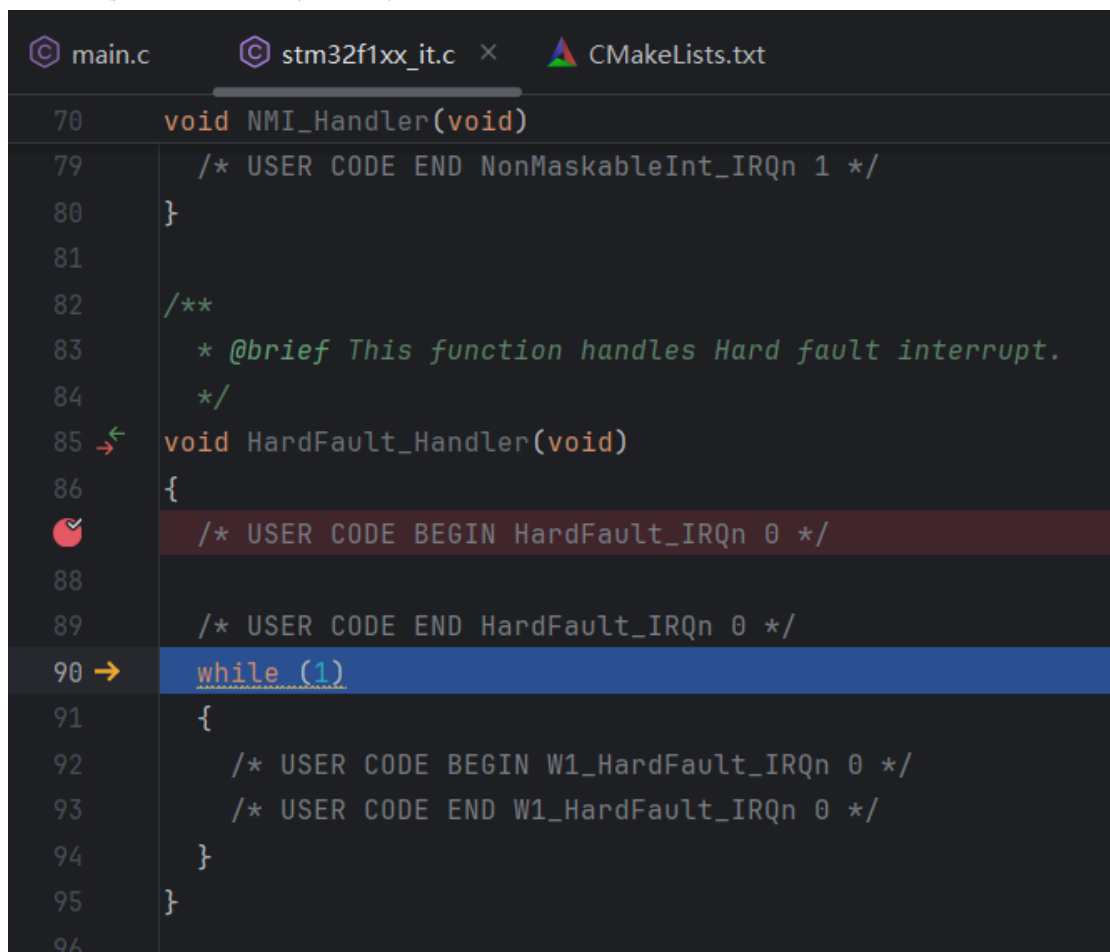
笔者修改了内存管理算法，使它成为了一个错误的工程，现在该如何调试它呢？

在 HardFault\_Handler 函数内部打上断点

HardFault\_Handler 函数是一个非常重要的函数，当我们的单片机出现各种奇怪的问题时，此时单片机就会触发这个中断，然后我们会发现程序一直在这里转圈圈，这其实是官方为了帮助我们 debug 而设置的一个功能：（断点最好打在 while (1) 上面）

### b 命令

b 命令的作用就是打上断点，在 cli 下，读者可以手动点击 87 行打上断点，也可以使用 b 87 命令打上断点



```
70 void NMI_Handler(void)
71
72 /* USER CODE END NonMaskableInt_IRQn 1 */
73
74
75 /**
76  * @brief This function handles Hard fault interrupt.
77  */
78 void HardFault_Handler(void)
79 {
80 /* USER CODE BEGIN HardFault_IRQn 0 */
81
82
83 /* USER CODE END HardFault_IRQn 0 */
84 while (1)
85 {
86 /* USER CODE BEGIN W1_HardFault_IRQn 0 */
87 /* USER CODE END W1_HardFault_IRQn 0 */
88 }
89 }
```

### c 命令

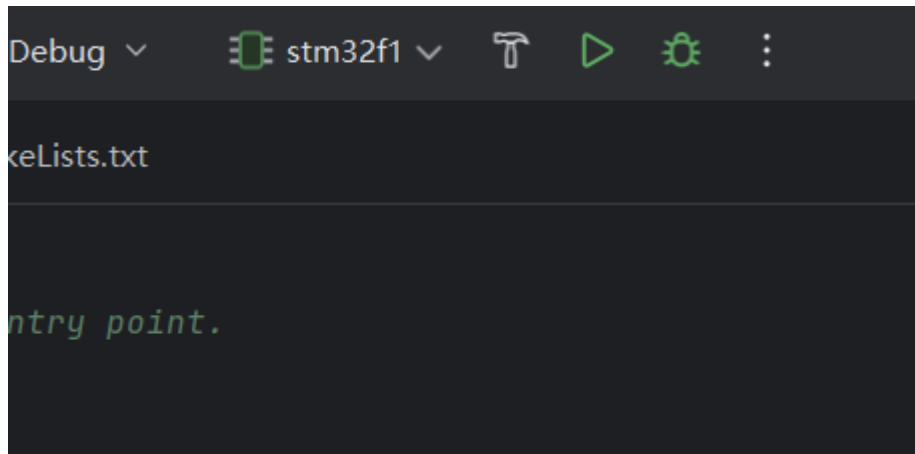
使用 c 命令，程序会直接运行到断点处然后停下来。



```
(gdb) c
Continuing.
```

### r 命令

r 命令是运行命令，不过在 clion 的 gdb 里面，点击右边那个小虫子，程序就自动运行到断点处了，如果是在 linux 环境下使用 gdb 是需要使用 r 命令运行程序的，但是在 clion 下是不用的。



```
Debug ▼ stm32f1 ▼ [Run] [Stop] [Toggle Breakpoint] [Settings] [More]
keLists.txt
ntry point.
```

## 调试错误的内存管理算法

当我们运行错误的内存管理算法的程序时，会发现程序跑进了 HardFault\_Handler，那么，是哪里导致了问题呢？

## 寻找案发现场

### 栈回溯

“目前的主流 CPU 架构都是用栈来进行函数调用的，栈上记录了函数的返回地址，因此通过递归式寻找放在栈上的函数返回地址，便可以追溯出当前线程的函数调用序列，这便是栈回溯（stack backtrace）的基本原理。通过栈回溯产生的函数调用信息称为 call stack（函数调用栈）。栈回溯是记录和探索程序执行踪迹的极佳方法，使用这种方法，可以快速了解程序的运行轨迹，看其“从哪里来，向哪里去”。”

以上解释来自张银奎老师的《软件调试》。

笔者简单解释，就是看进入 HardFault\_Handler 之前是哪些函数在不断嵌套调用。

## bt 命令

bt 命令可以帮助我们查看函数的嵌套调用：

```
(gdb) bt
#0  HardFault_Handler () at C:\Users\lei\ClionProjects\stm32f1\Core\Src\stm32f1xx_it.c:90
#1  <signal handler called>
#2  0x0000021c in heap_init () at C:\Users\lei\ClionProjects\stm32f1\Core\Src\main.c:112
#3  0x00000284 in heap_malloc (wantsize=15) at C:\Users\lei\ClionProjects\stm32f1\Core\Src\main.c:133
#4  0x000005aa in xTaskCreate (pxTaskCode=0x0000469 <leisureTask>, usStackSize=128, pvParameters=0x0, uxPriority=0, self=0x20002080 <leisureTcb>) at C:\Users\lei\ClionProjects\stm32f1\Core\Src\main.c:404
#5  0x000004b4 in SchedulerInit () at C:\Users\lei\ClionProjects\stm32f1\Core\Src\main.c:572
#6  0x00000764 in main () at C:\Users\lei\ClionProjects\stm32f1\Core\Src\main.c:572
#7  0x00001d0e in Reset_Handler () at C:\Users\lei\ClionProjects\stm32f1\startup_stm32f103xb.s:100
Backtrace stopped: previous frame identical to this frame (corrupt stack?)
```

我们现在发现了，在进入 HardFault\_Handler 之前，程序一路嵌套调用，最后在 heap\_init 处发生了错误。

现在该引出调试的两大原则了：问题简化原则和问题追踪原则。

## 问题简化原则

我们现在已经知道了，是 heap\_init 导致了错误，也就是说，很有可能是我们的内存管理算法出现了问题。为了验证我们的猜想，我们需要单独 debug 内存管理算法部分：



```
main.c x stm32f1xx_it.c CMakeLists.txt
202
203 /**
204  * @brief The application entry point.
205  * @retval int
206  */
207 int main(void)
208 {
209     HAL_Init();
210     SystemClock_Config();
211     MX_GPIO_Init();
212     int *a = heap_malloc( wantsize: sizeof (int *));
213     *a = 1;
214
215     while (1)
216     {
```

在 main 函数中，笔者删除了 sparrow 的其他程序，单独 debug 内存管理算法，让我们看看是不是它引起了错误，直接输入 c 命令运行到断点处，看看会不会再次进入 hardfault：

```
main.c  stm32f1xx_it.c  CMakeLists.txt
84      */
85  ↩ void HardFault_Handler(void)
86  {
87      /* USER CODE BEGIN HardFault_IRQn 0 */
88
89      /* USER CODE END HardFault_IRQn 0 */
90  →  while (1)
91  {
92      /* USER CODE BEGIN W1_HardFault_IRQn 0 */
93      /* USER CODE END W1_HardFault_IRQn 0 */
94  }
95  }
96
97  /**
98   * @brief This function handles Memory management fault.
99   */
```

问题依旧存在！这说明确实是内存管理算法有问题。

## 案发现场的确认

为了搞清楚是那一行命令导致的问题，我们需要查看跳转到 HardFault\_Handler 前程序在执行那一行程序，为此，我们可以借助堆栈指针。

arm cm3 架构的单片机采用的是双堆栈，也就是有两个 stack 指针(psp 和 msp)，分别在不同场合使用。

我们需要查看寄存器 R14(LR) 的值。如果 R14(LR) = 0xFFFFFFFF9，继续查看 MSP（主堆栈指针）的值，如果 R14(LR) = 0xFFFFFFF9D，继续查看 PSP（进程栈指针）的值。

### info registers 命令

使用该命令，可以帮助我们快速查看 arm 所有寄存器的值：

```
(gdb) info registers
r0          0x4          4
r1          0x2          2
r2          0x0          0
r3          0x20000000    536870912
r4          0x20002084    536879236
r5          0x2000097c    536873340
r6          0x20         32
r7          0x20004f8c    536891276
r8          0x0          0
r9          0x0          0
r10         0x0          0
r11         0x0          0
r12         0x0          0
sp          0x20004f8c    0x20004f8c
lr          0xffffffff9   -7
pc          0x8000488     0x8000488 <HardFault_Handler+4>
xpsr       0x61000003    1627389955
msp         0x20004f8c    0x20004f8c
psp         0x2000023c    0x2000023c <allheap+516>
```

### x/10x \$msp 命令

笔者的 lr 是 0xFFFFFFFF9，所以需要查看 msp 后面的内存。

x/10x \$msp 命令会从 \$msp 寄存器的值开始，检查并显示接下来 10 个内存单元的内容，每个单元的值以十六进制格式显示：

```
(gdb) x/10x $msp
0x20004f8c: 0x20004fb0  0x00000004  0x00000002  0x00000000
0x20004f9c: 0x20000000  0x00000000  0x08000281  0x08000218
0x20004fac: 0x61000000  0x00002000
```

看着这些十六进制的数字不要慌，想一想我们的程序地址一般是从哪里开头的？一般不是在 0x08 后面吗？所以我们只需要查看 0x080 开头的地址的内容即可。

### list 命令

0x08000281 处的程序：

我们已经知道了错误是在 heap\_init 内存发生的，因此我们还需要进一步查看。

```
(gdb) list *0x08000281
0x08000281 is in heap_malloc (C:\Users\el\CLionProjects\stm32f1\Core\Src\main.c:135).
130     }//You can add the TaskSuspend function ,that make here be an atomic operation
131     if(theheap.tail== NULL )
132     {
133         heap_init();
134     }//Resume
135     prevnode = &theheap.head;
136     usenode = theheap.head.next;
137     while((usenode->blocksize) < wantsize )
138     {
139         prevnode = usenode;
```

0x08000218 处:

```
(gdb) list *0x08000218
0x08000218 is in heap_init (C:\Users\el\CLionProjects\stm32f1\Core\Src\main.c:112).
107     end_heap &= ~alignment_byte;
108     theheap.allsize = (size_t)(end_heap - start_heap );
109 }
110 theheap.tail = (heap_node*)end_heap;
111 theheap.tail->blocksize = 0;
112 theheap.tail->next = NULL;
113 firstnode = (heap_node*)start_heap;
114 firstnode->next = theheap.tail;
115 firstnode->blocksize = theheap.allsize;
116 }
```

## disassemble 命令

如果读者懂汇编语言，也可以查看汇编程序：



```
(gdb) disassemble 0x08000218
Dump of assembler code for function heap_init:
0x080001a4 <+0>: push    {r7}
0x080001a6 <+2>: sub    sp, #20
0x080001a8 <+4>: add    r7, sp, #0
0x080001aa <+6>: ldr    r3, [r7, #12]
0x080001ac <+8>: and.w  r3, r3, #7
0x080001b0 <+12>: cmp    r3, #0
0x080001b2 <+14>: beq.n  0x80001d2 <heap_init+46>
0x080001b4 <+16>: ldr    r3, [r7, #12]
0x080001b6 <+18>: adds   r3, #7
0x080001b8 <+20>: str    r3, [r7, #12]
0x080001ba <+22>: ldr    r3, [r7, #12]
0x080001bc <+24>: bic.w  r3, r3, #7
0x080001c0 <+28>: str    r3, [r7, #12]
0x080001c2 <+30>: ldr    r3, [pc, #120] ; (0x800023c <heap_init+152>)
0x080001c4 <+32>: ldr    r2, [r3, #12]
0x080001c6 <+34>: ldr    r3, [r7, #12]
0x080001c8 <+36>: subs   r3, r2, r3
0x080001ca <+38>: ldr    r2, [pc, #116] ; (0x8000240 <heap_init+156>)
0x080001cc <+40>: add    r3, r2
0x080001ce <+42>: ldr    r2, [pc, #108] ; (0x800023c <heap_init+152>)
0x080001d0 <+44>: str    r3, [r2, #12]
```

## 观察程序执行发现具体问题

### 问题追踪原则

#### s 和 n 命令

到现在，我们找到了案发现场，但是如果案发现场也仅仅是受害者呢？比如一个函数的错误执行导致修改了某个指针，案发现场的程序对这个指针进行了解引用，这个时候该怎么办呢？

此时我们需要重新梳理程序的执行，遵守问题追踪原则，使用 s 和 n 命令一步步执行程序，找出问题。

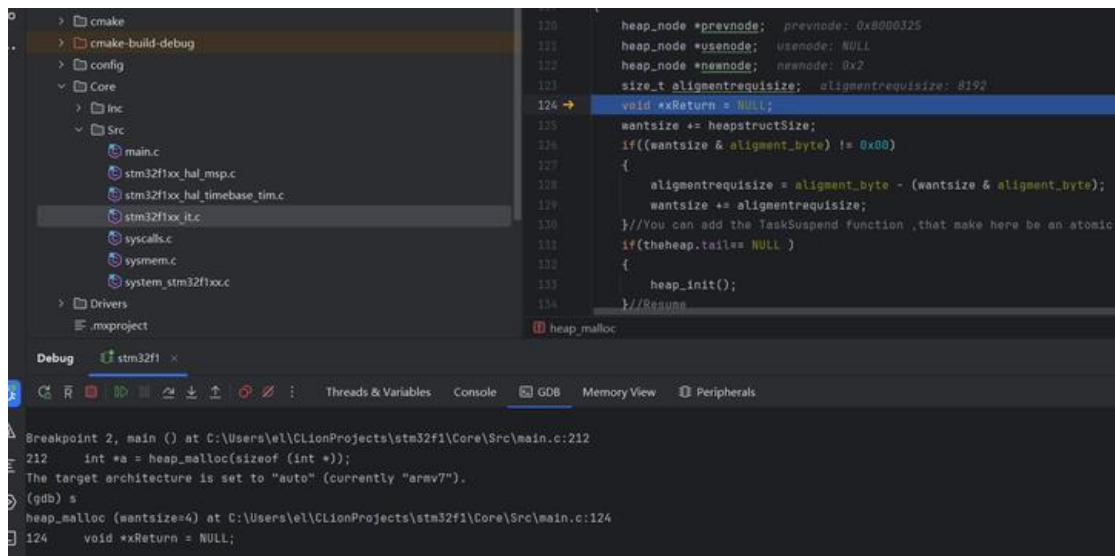
#### s 命令：

一行行执行程序，遇到函数会进入函数内部继续一行行执行命令。

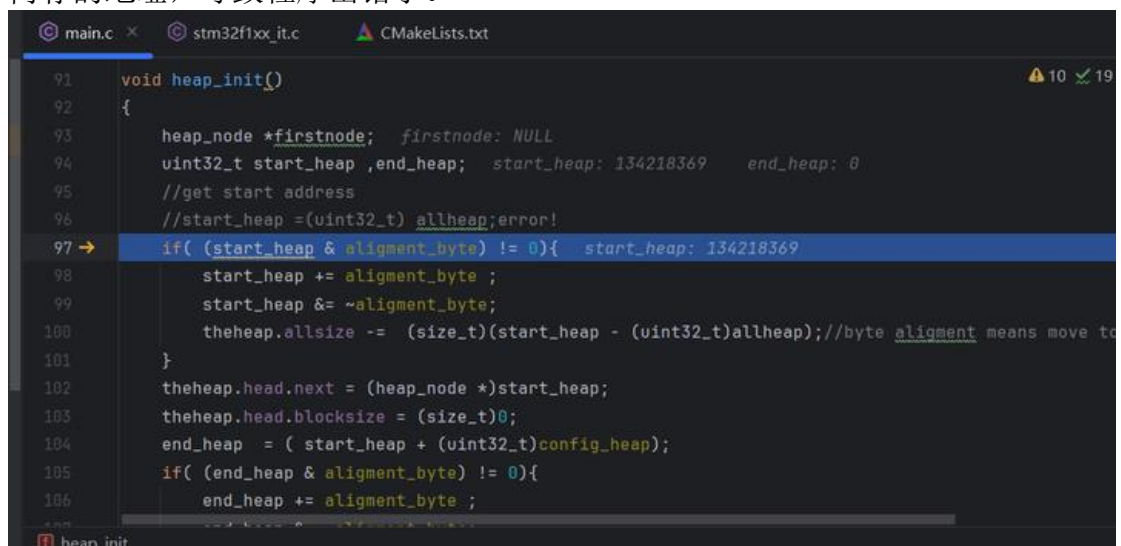
#### n 命令：

一行行执行程序，但是遇到函数不会进入函数内部，而是执行完这一行后转到下一行。

使用 s 命令，我们来到了 malloc 内部：

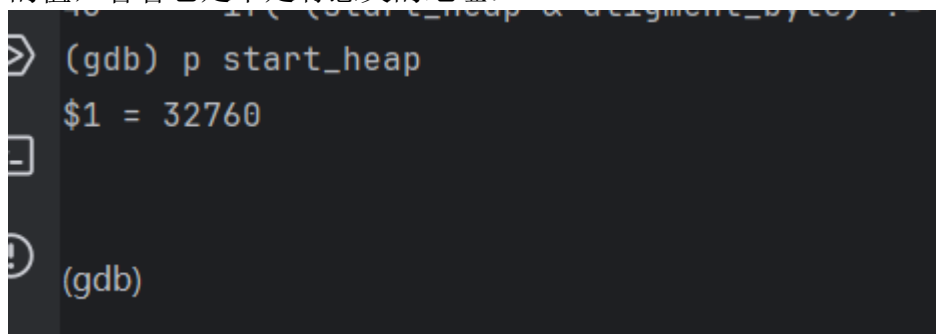


继续使用 s 命令，我们进入到了 heap\_init 函数内部：  
其实到这里读者应该都看得出了，笔者并没有给 start\_heap 传递 allheap 这片内存的地址，导致程序出错了。



p 命令

p，也就是 print，该命令可以帮助我们查看变量的值，我们先查看 start\_heap 的值，看看它是不是有意义的地址：



显然，它并不是，让我们再看看内存块的起始地址：

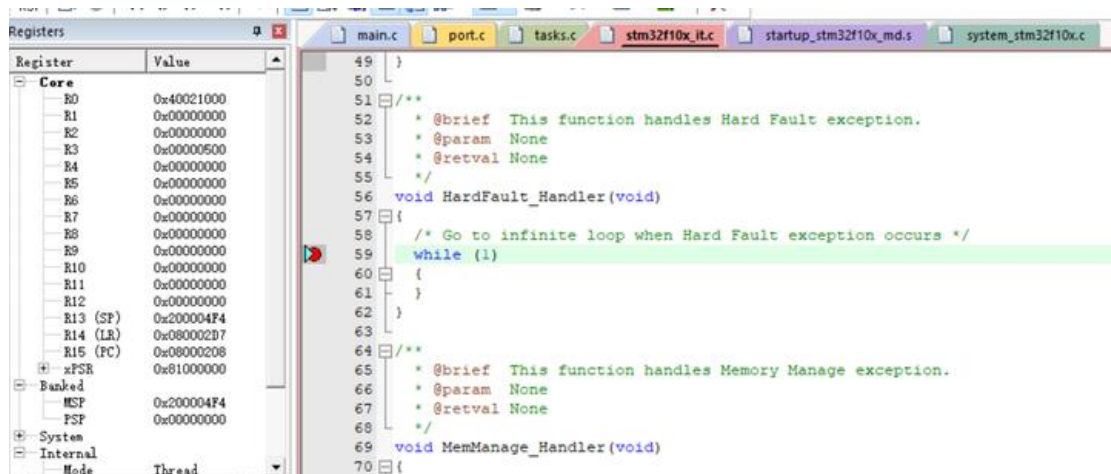
```
(gdb) p (uint32_t)allheap
$2 = 4227136
```

可以看出，确实是 start\_heap 没有赋给它内存块的地址导致出错的。  
修改了之后程序就能继续运行了：

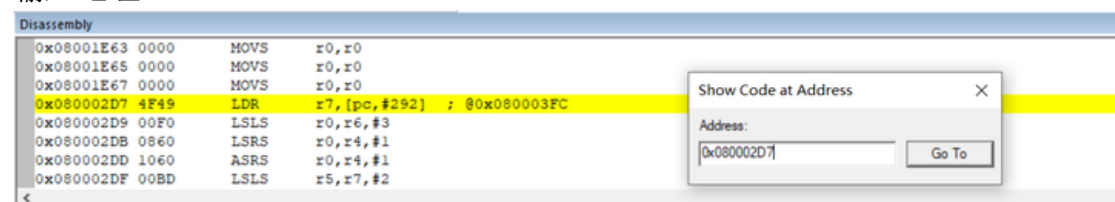
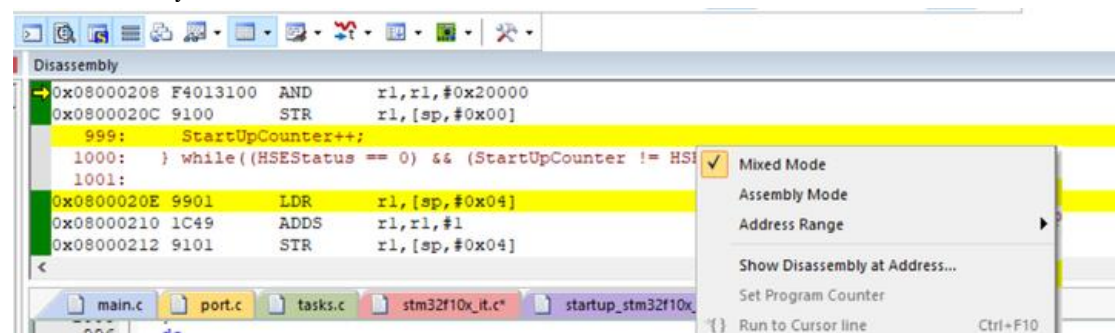
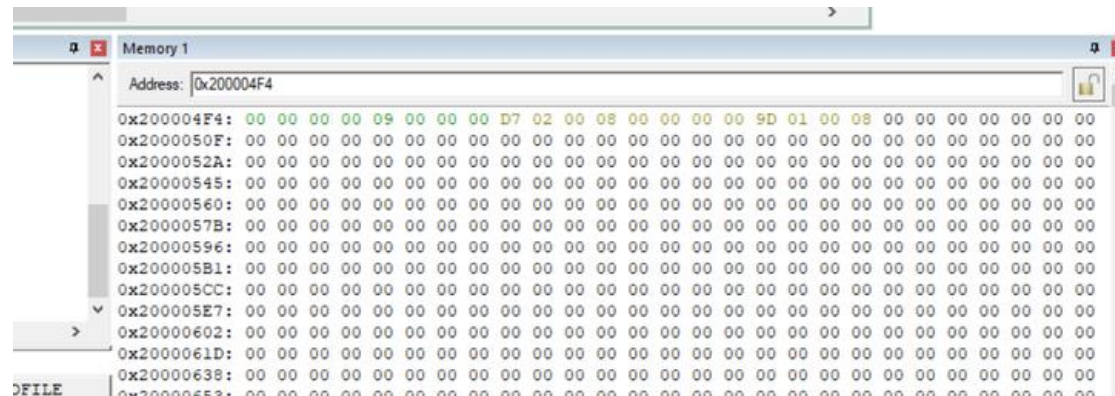
```
90
91 void heap_init()
92 {
93     heap_node *firstnode;
94     uint32_t start_heap ,end_heap;
95     //get start address
96     start_heap =(uint32_t) allheap;
97     if( (start_heap & alignment_byte) != 0){
98         start_heap += alignment_byte ;
99         start_heap &= ~alignment_byte;
100     }
101     theheap.allsize -= (size_t)(start_heap - (uint32_t)allheap);//t
102 }
```

## keil 调试

使用 keil 调试是一样的，keil 只要点击这个放大镜即可进入调试。  
keil 的右边会自动显示各个寄存器的值。



调试方法是一样的：我们需要查看寄存器 R14(LR) 的值。如果 R14(LR) = 0xFFFFFFE9，继续查看 MSP（主堆栈指针）的值，如果 R14(LR) = 0xFFFFFFFD，继续查看 PSP（进程栈指针）的值。



## 总结

heapmem.c 和 heapmem.h 可以作为一个库被广泛使用在嵌入式单片机程序中，并且比通用的 c 语言 malloc 效率更高，执行时间也相对固定，能够提高程序的性能。

# 内核框架

## 前言

通过前几章的学习，我们学会了如何为 RTOS 设计一个合理的内存管理算法。现在，是时候学习设计 RTOS 内核了。  
从我们的需求与应用出发，使用面向对象的思想，逐步构建一个 RTOS 的内核。

## 程序 = 数据结构 + 算法

面向对象思想本身和程序 = 数据结构 + 算法思想就是相通的，我们可以通过类和对象来表现数据结构，通过方法实现算法，从对象与对象的交互关系来构建，从而实现更加健壮的程序。

## 抢占式内核

抢占式内核是一种流行的调度策略，操作系统会选择就绪列表中优先级最高的任务，而 CPU 会执行这个任务。

## 我们需要实现什么？

如果读者有过使用 RTOS 的经历，那么请你思考：RTOS 实现了什么？

笔者先提出一点：多线程与优先级带来的实时性

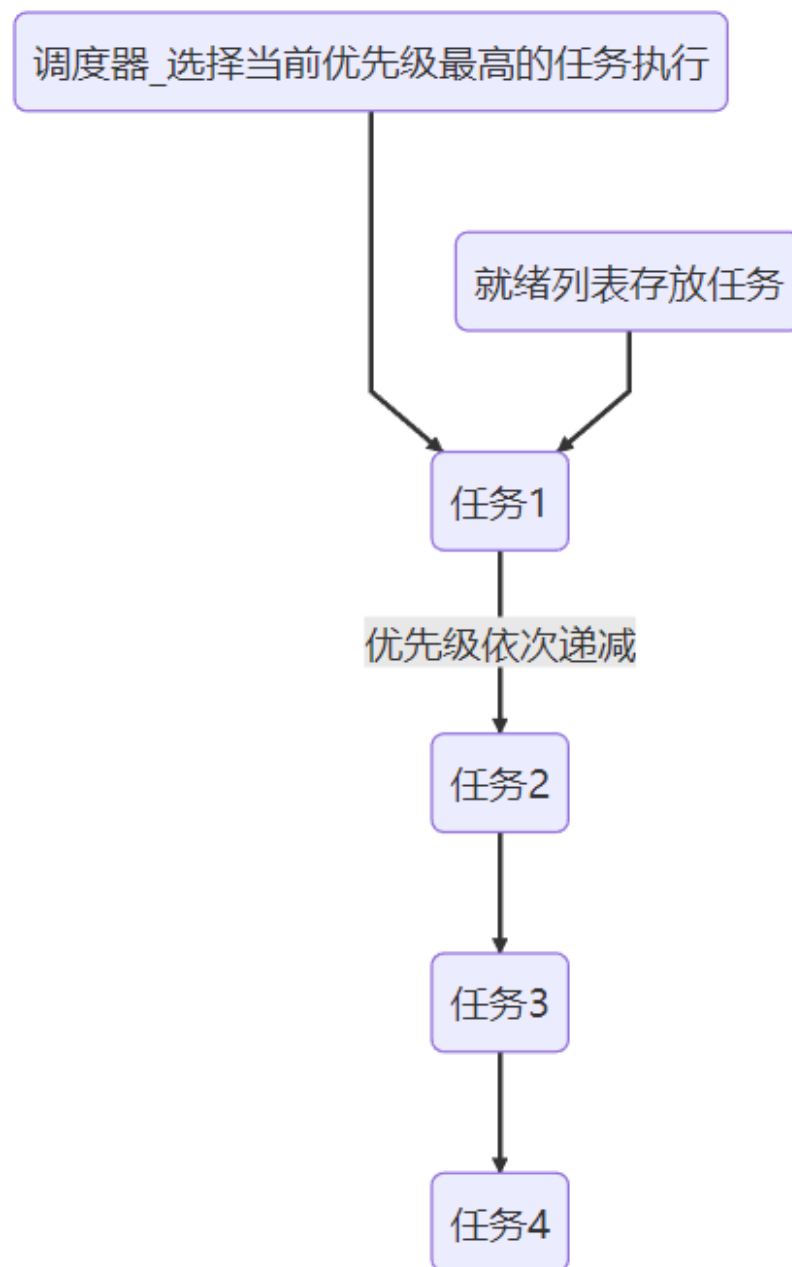
RTOS 将应用程序划分为多个独立的任务，也就是多线程。多线程允许同时执行多个任务，提高系统的处理能力和效率。例如，在嵌入式系统中，一个线程可以处理传感器数据，另一个线程可以更新用户界面。

实时性的需求，要求 RTOS 必须在指定的时间内完成关键任务。

我们创建一个又一个的任务(线程)，它们可以并发执行。设置优先级时，我们知道优先级高的任务会优先执行，从而满足实时性。当我们想更方便地管理任务时，我们会想到使用 RTOS。强大的实时性与并发执行的任务，这就是我们想要实现的结果，但是，我们该如何去实现它呢？

## 如何实现实时性？

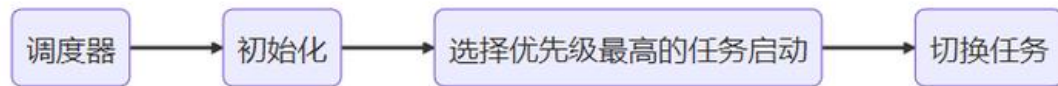
请读者想一想，我们创建任务设置优先级时，往往希望某些任务被优先执行，这是实时性实现的关键，也就是说，会有一个调度器来选择高优先级的任务，因此，我们得到了两个对象：任务和调度器。





## 调度器对象

通过上图我们可以推断，调度器会选择就绪列表中优先级高的任务。同时，就绪列表经常会发生变化，当优先级最高的任务发生变化，那么调度器还要切换任务，因此，我们得到了下图：

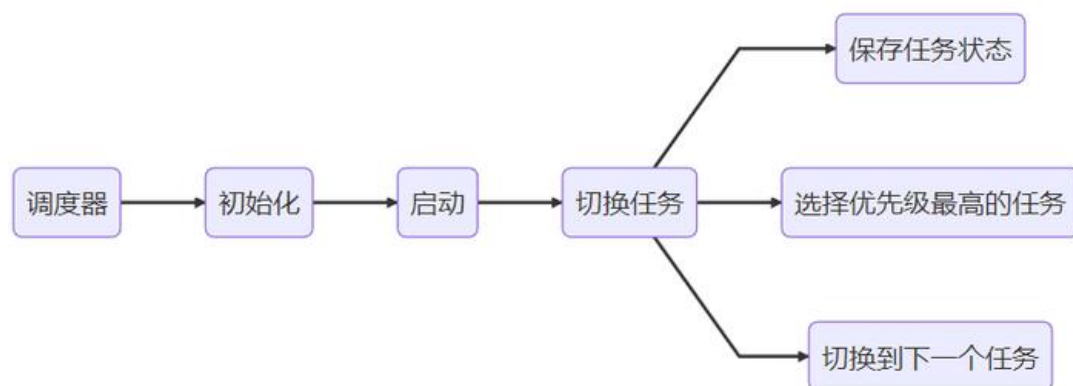


## 切换任务

切换任务时，我们肯定不希望先前任务的状态丢失，因此需要保存任务状态。线程（任务）切换如下：

1. 保存之前运行的线程的上下文
2. 选择优先级高的任务
3. 调用准备运行的线程的上下文

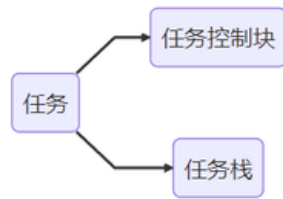
因此有了下图：



保存任务状态，这部分就涉及到和任务对象的交互了。

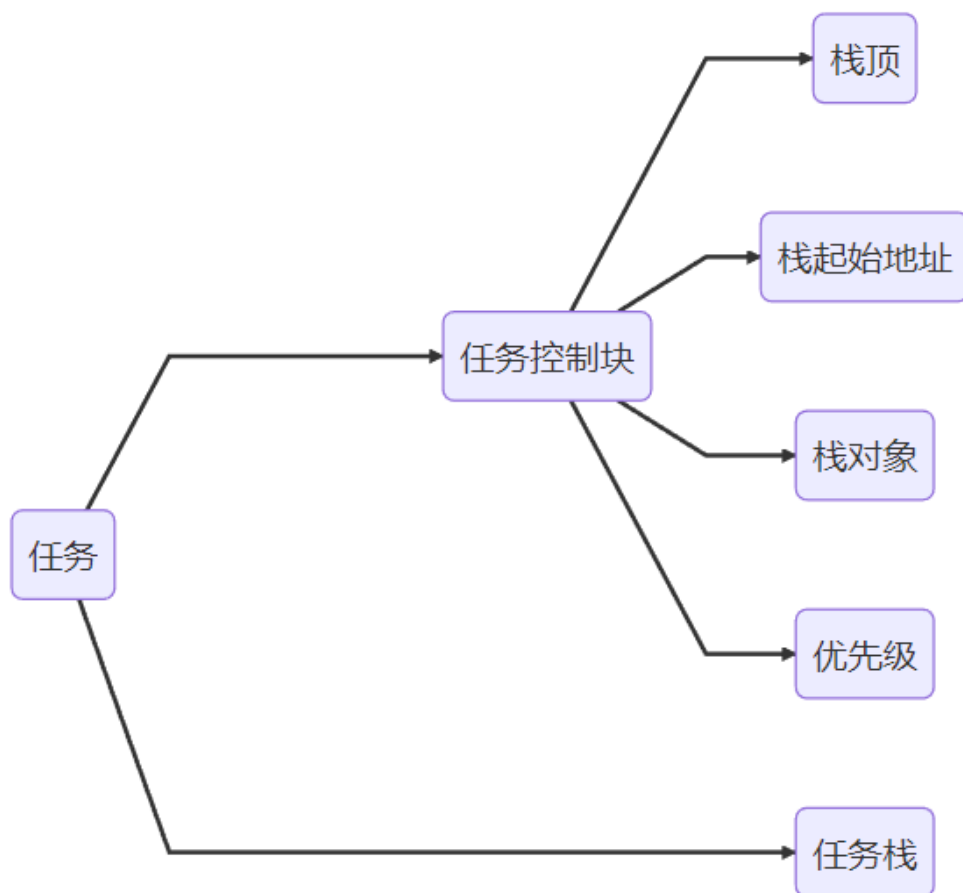
## 任务对象

为了方便管理任务，比如设置优先级啥的，我们肯定需要一个任务控制块，也方便我们把任务挂载到就绪列表中。同时，我们要保存当前状态，也就是说我们需要内存，那么这段内存我们给它命名为栈。



## 任务控制块

一个任务需要记录任务栈的信息，也就是 `pxTopOfStack`（栈顶）、`pxStack`（栈起始地址）、`self_stack`（栈对象）这三个成员。为了实时性，我们还需要优先级。把图进一步展开：





现在，我们关键的数据结构已经出来了，请读者写下这些代码：

（pxCurrentTCB 是任务对象指针，它会指向优先级最高的任务，这个任务会被执行）

sparrow.c

```
Class(TCB_t)
{
    volatile uint32_t * pxTopOfStack;
    unsigned long uxPriority;
    uint32_t * pxStack;
    Stack_register *self_stack;
};

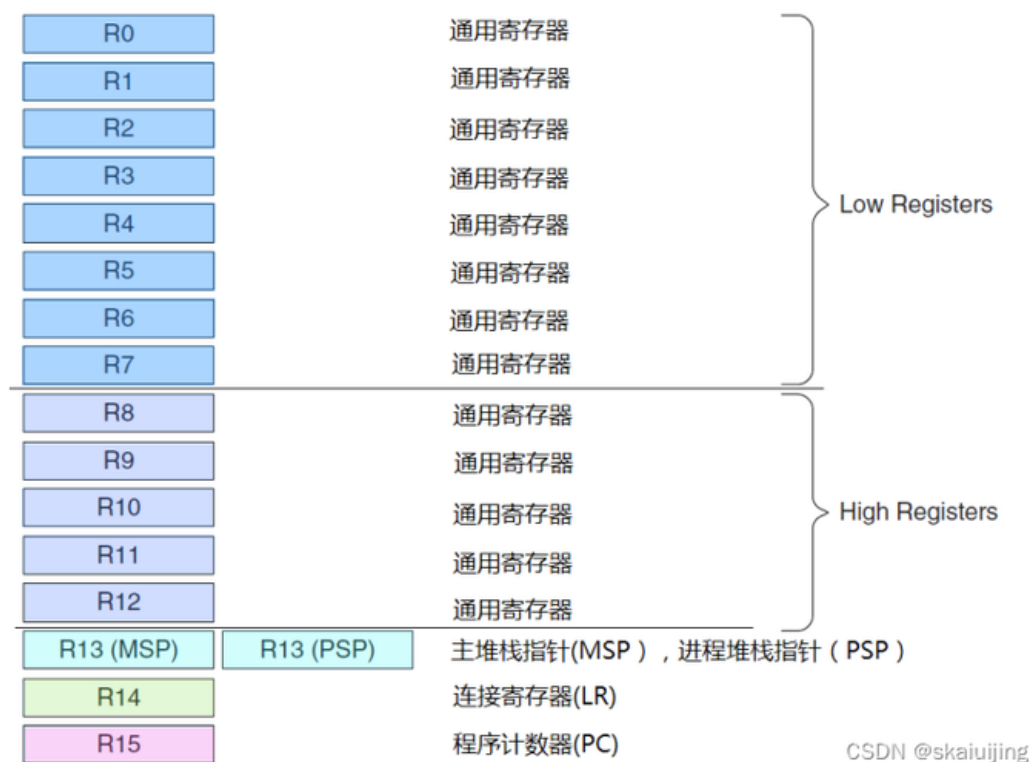
typedef TCB_t          *TaskHandle_t;

__attribute__((used)) TCB_t * volatile pxCurrentTCB = NULL;
typedef void (* TaskFunction_t)( void * );
```

## 栈对象

对于栈对象，我们要保存先前的任务状态，方便下一次任务执行时取出当前任务状态到 CPU 中，那么任务状态是 CPU 中的哪些信息呢？答案是寄存器（其实还有变量啥的，不过硬件会自动帮我们保存这些，所以我们只需要手动保存部分寄存器即可）

硬件自动保存 CPU 状态其实我们经常遇到，请读者想一想，中断发生时需要我们手动保存 CPU 状态吗？



以及 XPSR，它是非常重要的特殊寄存器：

特殊功能寄存器：

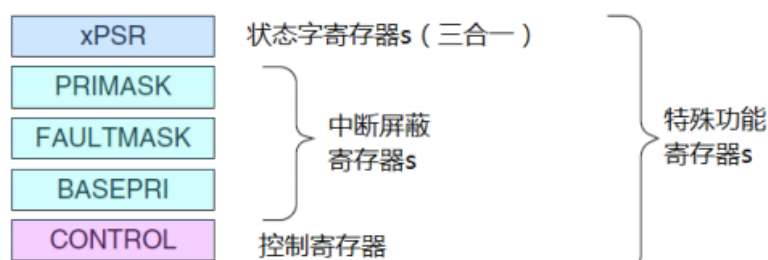
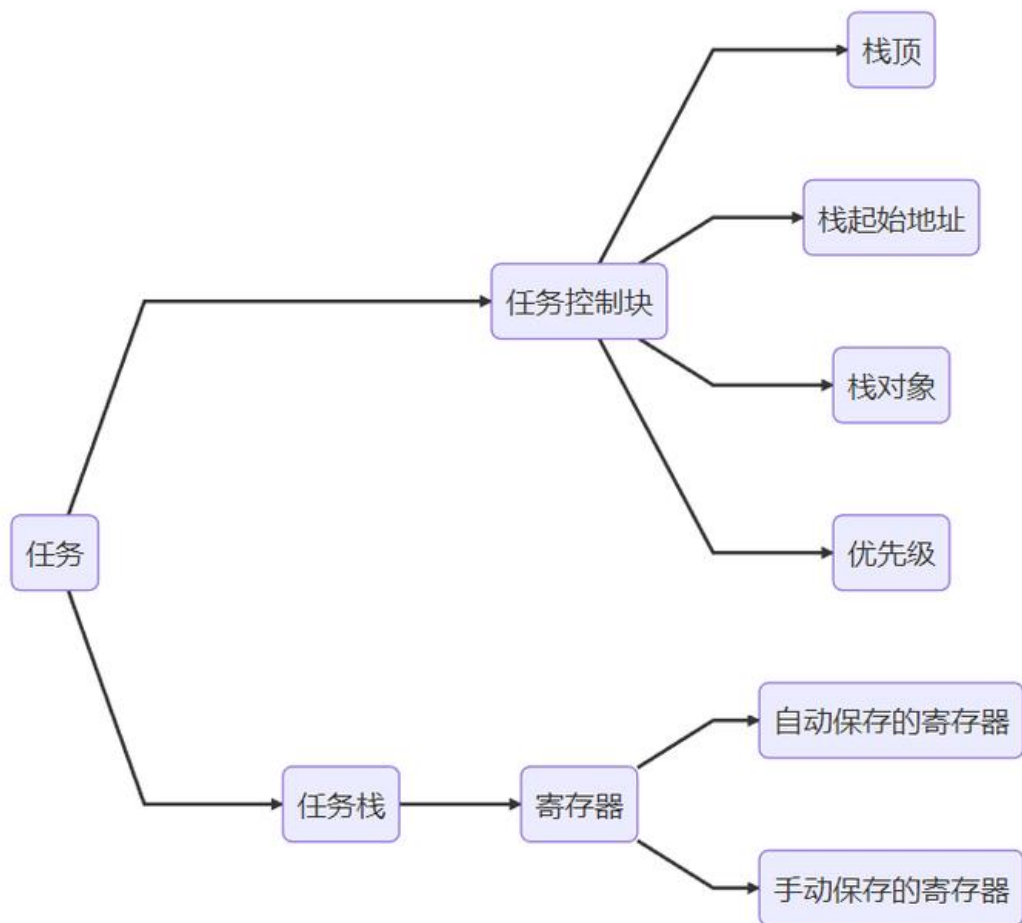


图 3.1 Cortex-M3 的寄存器组 CSDN @skaiuijing

寄存器包括两部分寄存器，一部分是发生中断时硬件自动帮我们保存的寄存器，另一部分是需要我们手动保存的寄存器。

因此继续展开我们的图：



因此让我们写下代码：

```
Class(Stack_register)
{
    //manual stacking
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
    uint32_t r10;
    uint32_t r11;
    //automatic stacking
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r12;
```

```
uint32_t LR;  
uint32_t PC;  
uint32_t xPSR;  
};
```

## 总结

将程序划分为多个任务，根据任务的重要性设置不同的优先级，使用调度器选择具体的任务，选择任务意味着会发生切换，切换意味着要保存当前任务的状态并加载下一个任务的状态（上下文切换），状态意味着任务具体的执行信息，执行信息意味着 CPU 的各种寄存器和变量的值，值的保存意味着要使用内存（栈），最后，使用内存保存这些值就要根据具体的 CPU 架构进行设计了。

我们得到的对象有：任务、调度器、栈。

任务对象需要执行具体的任务，调度器对象需要选择优先级更高的任务，栈对象需要保存任务对象的状态。

# arm cm3 架构

## 前言

通过上一篇文章，我们已经知道了一个 RTOS 的基本框架，寄存器这个概念已然跃出水面，摆在我们面前的难题是对 arm 架构的理解，以最常用的 stm32f10 系列为例，它的架构是 arm cortex m3。

## 封装与接口思想

由于 c 语言的操作尺度不足，我们只能使用汇编语言完成任务切换，同时很多中断的配置需要对芯片内部的一个又一个的角落进行编程。请读者必须明白，你并不是芯片厂商，你也不需要从门电路开始搭建 CPU。我们需要的是查找手册中的信息并使用它，在 arm 手册中寻找实现 RTOS 的指导，而不是关注 arm 架构的本质。我们的重点是关注实现的功能，而不是芯片手册的本质理解。如果读者没有 arm 架构和汇编基础，你大可直接跳过，只要知道我们使用的汇编和二进制程序实现了什么功能即可。换而言之，请使用封装与接口思想！

## arm 架构

简单介绍一下我们要在汇编尺度下完成的事情：产生 SVC 中断调用->开启第一个任务->任务执行 ->systick 中断触发 Pendsv 中断 ->pendsv 中断->保存原先任务状态 ->切换下一个任务状态->任务执行->systick 中断触发 Pendsv 中断 ->pendsv 中断->保存原先任务状态 ->切换下一个任务状态-----循环-----  
----

## 汇编语言：基本语法

汇编指令的最典型书写模式如下所示：

**标号**

**操作码            操作数 1,    操作数 2, ...    ;**

在 arm 架构中，有七种工作模式，37 个寄存器。

了解以下模式即可：

1. 正常情况下是 usr 模式
2. 用于高速数据传输和通道处理时切换为 FIQ 模式
3. 执行外部中断时是 irq 模式
4. 最高权限是管理模式 svc
5. 系统模式 sys

除了系统模式和用户模式，其他模式都被称之为异常。

在 arm 汇编中。我们使用寄存器进行编程，对数据的操控都要通过寄存器进行。寄存器分为 31 个通用寄存器和 6 个状态寄存器，在不同的状态下，我们能使用的寄存器可能会不同。

### 查找手册

为了让我们的框架落到实处，我们必须寻求 arm 架构手册的指导。

（手册中的堆栈，英文是 stack，建议直接理解为栈，因为堆和栈并不是同一种内存结构，用途大相径庭，例如堆溢出和栈溢出其实是不同的内存分配方式导致的，所以区分堆和栈是有必要的）

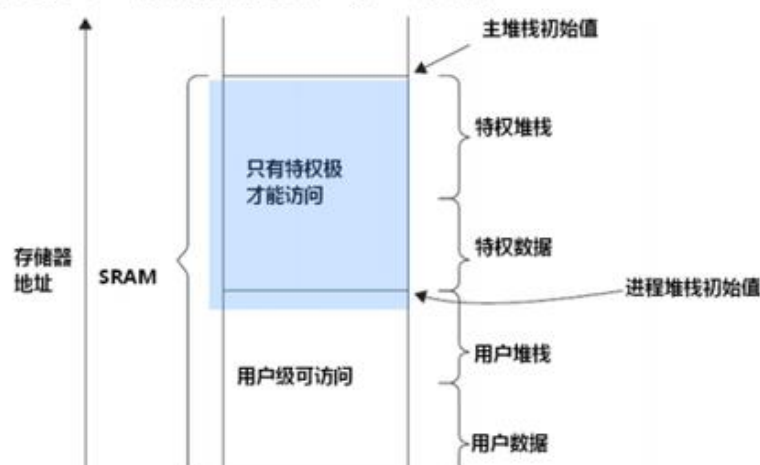
### 栈

让我们看看 arm cortex m3 官方手册是如何指导写出一个 os 的，这是关于栈的部分：

要在 CM3 中创建可靠扛打的系统，必须两手抓，两手都要硬。典型地，一个真正健壮的 CM3 软件系统都要使用实时操作系统内核的，通常会符合如下的要求：

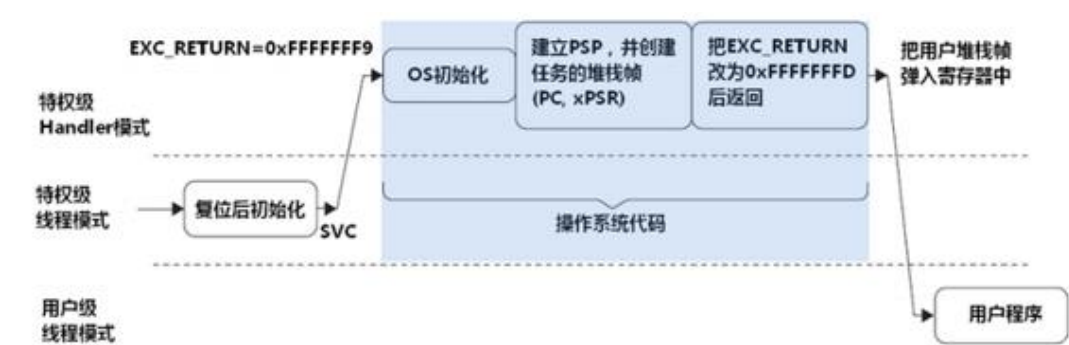
- 服务例程使用 MSP（在“非基级线程模式”中会讲到例外情况）
- 尽管异常服务例程使用 MSP，但是它们在形式上返回后，内容上却可以依然继续——而且此时还能使用 PSP，从而实现“可抢占的系统调用”，大幅提高实时性能
- 通过 SysTick，实时内核的代码每隔固定时间都被调用一次，运行在特权级水平上，负责任务的调度、任务时间管理以及其它系统例行维护
- 用户应用程序以线程的形式运行，使用 PSP，并且在用户级下运行
- 内核在执行关键部位的代码时，使用 MSP，并且在辅以 MPU 时，MSP 对应的堆栈只允许特权级访问

如图 12.1 所示，假设系统内存是一块 SRAM，则我们可以通过 MPU，把它分为两个 regions，其中一个用于用户级，另一个用于特权级。另外别忘了 CM3 的堆栈是“向下生长的满栈”，因此需要把这两个 SP 初始为指向这两个 regions 的顶端。



简单来说,就是使用双栈机制。关键性的代码使用 MSP 指针,此时处于特权模式。应用程序代码使用 PSP 指针,此时处于用户态。

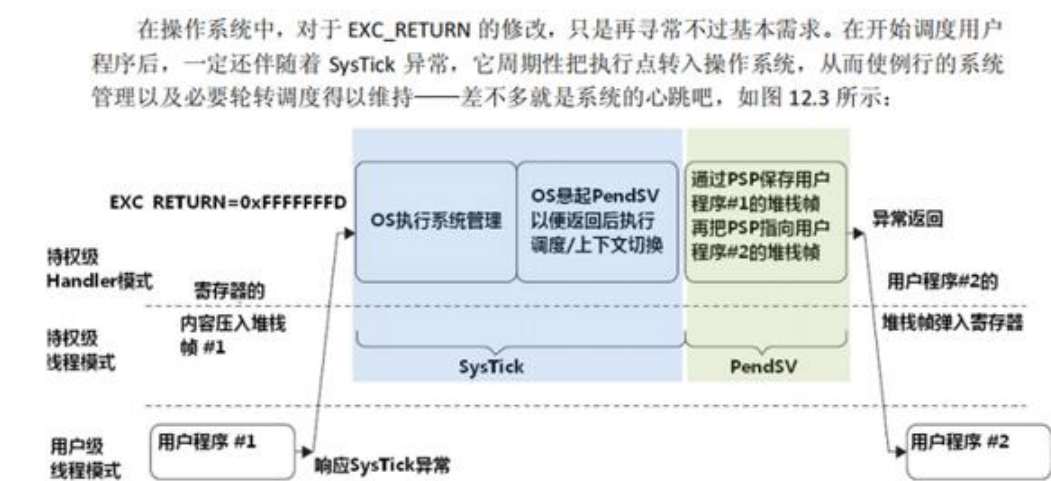
我们知道栈要初始化,初始化的理论依据可以参考 arm cortex m3 官方手册对 os 栈初始化的介绍:



## 中断与上下文切换

笔者在前面的博客讲过一点中断在 arm 架构中是完成任务切换实现多线程的关键因素,然后引出了 systick 中断和 Pendsv 中断的概念,它们都属于 arm cm3 的硬件资源。

让我们看看 arm cortex m3 官方手册:



笔者在之前调试 sparrow 时已经说过, systick 中断会触发 pendsv 中断完成上下文切换,这与图中是一样的。



# 任务的启动

显然，我们开启 RTOS 第一个任务，肯定要保证第一个任务马上执行，不会受到干扰，所以，为了能够启动 RTOS，sparrow 利用了 svc 中断，也就是系统中断，它的特点是触发后立即执行。

下图是中断向量表。  
中断向量表是一种数据结构，它用于存储中断服务程序（ISR）的入口地址。当硬件设备触发中断时，CPU 会根据中断号查找中断向量表，以获取对应 ISR 的地址，并跳转到该地址执行相应的中断处理程序：

表 2.2 Cortex-M3 异常类型

编号	类型	优先级	简介
0	N/A	N/A	没有异常在运行
1	复位	-3( 最高 )	复位
2	NMI	-2	不可屏蔽中断（来自外部 NMI 输入脚）
3	硬(hard) fault	-1	所有被除能的 fault，都将“上访”成硬 fault
4	MemManage fault	可编程	存储器管理 fault，MPU 访问犯规以及访问非法位置
5	总线 fault	可编程	总线错误（预取流产（Abort）或数据流产）
6	用法(usage) Fault	可编程	由于程序错误导致的异常
7-10	保留	N/A	N/A
11	SVCall	可编程	系统服务调用
12	调试监视器	可编程	调试监视器（断点，数据观察点，或者是外部调试请求
13	保留	N/A	N/A
14	PendSV	可编程	为系统设备而设的“可悬挂请求”（pendable request）
15	SysTick	可编程	系统滴答定时器（也就是周期性溢出的时基定时器——译注

于是，在 SVC 中断中，我们将把第一个任务的内容加载到了寄存器中，sparrow 开始执行任务。随后，来到 PendSV 函数，在这里，我们将会保存前一个任务的状态，并且切换到下一个任务。

PendSV（可悬起的系统调用），它和 SVC 协同使用。一方面，SVC 异常是必须立即得到响应的，应用程序执行 SVC 时都是希望所需的请求立即得到响应。另一方面，PendSV 则不同，它是可以像普通的中断一样被悬起的（不像 SVC 那样会上访。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

系统调用是提供给用户的接口，可以触发优先级抢占，滴答计时器中断时可以发生时间片轮转。

顺便一提，在 sparrow 中，如果创建任务时没有延时或者其他进入阻塞的方式，任务的优先级又非常高，那么任务就会一直执行。

这是因为产生系统调用或产生滴答计时器触发上下文切换时，高优先级的任务不会进入延时或阻塞，而是一直在就绪态，又因为优先级过高导致其它优先级的任务不能进行抢占。

## 任务的保存与切换

关于任务是如何被保存的，读者可以参考下文，知道如何保存，也就知道了如何切换：

### 中断 / 异常的响应序列

当CM3开始响应一个中断时，会在它看不见的体内奔涌起三股暗流：

- 入栈：把8个寄存器的值压入栈
- 取向量：从向量表中找出对应的服务程序入口地址
- 选择堆栈指针MSP/PSP，更新堆栈指针SP，更新连接寄存器LR，更新程序计数器PCing

从 arm 架构的角度看，RTOS 通过 SVC 中断开启了第一个任务，随后 SysTick 中断定时触发 PendSV 中断进行任务的保存与切换，这就是 sparrow 中的任务切换调度部分。

## 总结

以上就是本文要探讨的重点，即 arm cm3 中的栈、中断与上下文切换，笔者将会结合 sparrow 源码来理解官方手册。

## 思考题

1. 堆和栈到底有什么区别？
2. Sparrow RTOS 的内存管理算法分配的是哪一部分的内存？
3. arm cm3 的双 stack 指针分别使用的是哪一部分内存？
4. 我们能配置堆和栈的大小吗？如果可以，那么单片机的堆和栈的大小是在程序执行的哪一步被初始化的？
5. 什么情况下会导致堆和栈溢出？什么是“溢出”？

# 时间触发系统设计

## 前言

RTOS 有两种大类型，分别是事件触发系统和时间触发系统。

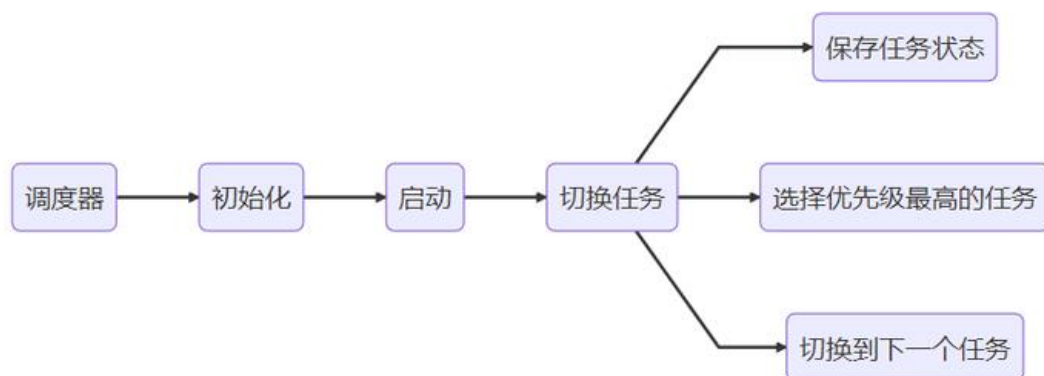
事件触发系统由中断驱动，通常采用中断线程化等技术，适用于响应速度要求高的场景，例如传感器数据采集，但是当大量事情一次性发生时，计算机将面临大问题。

（一般的中断是 CPU 放下一切全力去做一件事情，中断线程化就是把事情放线程里，等中断触发这个线程，然后让这个线程跟其他线程并行执行）

时间触发系统的任务调度基于定时器中断，适用于周期性任务和确定性要求高的场景，如控制系统。它的内部有一个时钟，每隔一定时间间隔就会触发一次时间中断，每次时钟中断时，调度器决定是否需要切换任务。

不过现在的 RTOS 往往会结合这两种类型，比如 FreeRTOS 就属于时间触发系统和事件触发系统的结合。

这就是要完成的框架：



## sparrow 的运行

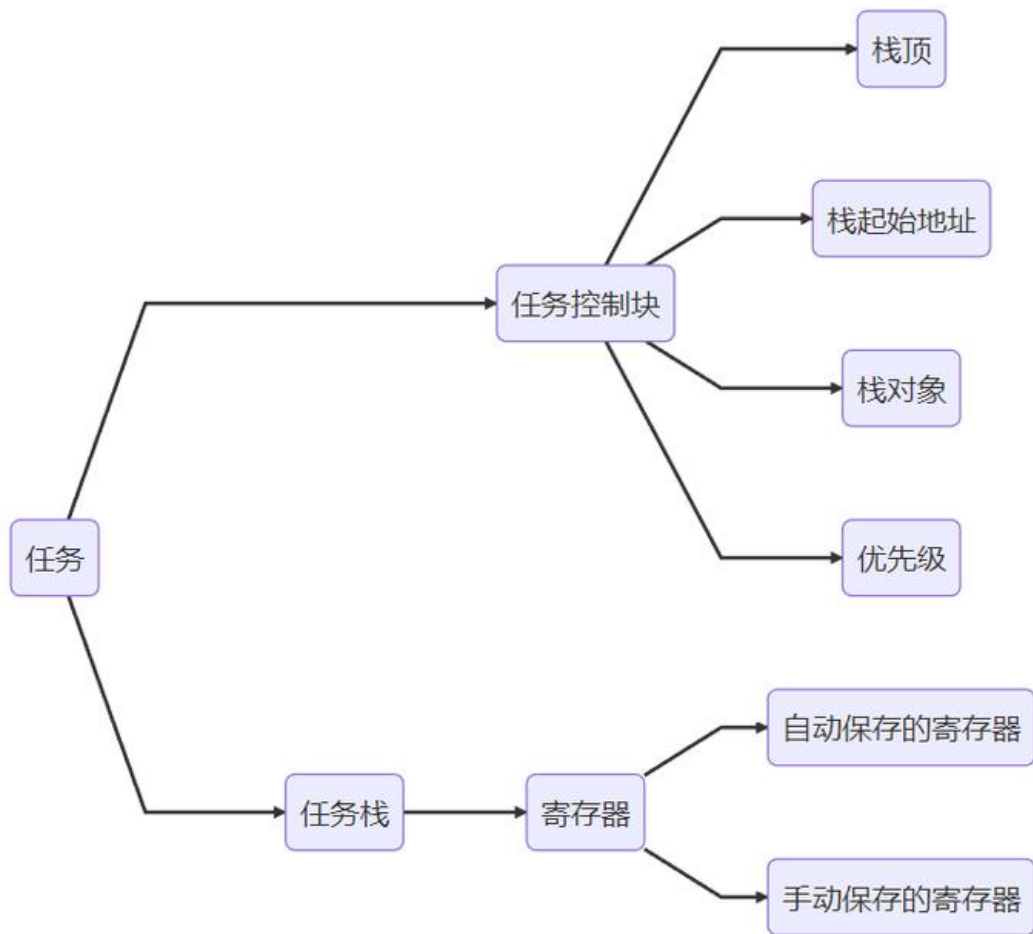
现在让我们来看看 Sparrow 的设计，因为它并不是一个完整的 RTOS 内核，现在还是时间触发系统的部分占主要成分：

当 Sparrow 运行起来后，调度器负责选择并且切换任务，结合笔者前面的 arm 架构的文章可知：

1. 调度器的组成有 SVC、Systick、PendSV 三个中断，它们都属于 arm cortex m3 架构的硬件资源。
2. SVC：负责当 MCU 复位后启动第一个任务，此时属于特权级线程模式，确保 os 的启动成功。
3. Systick：定时触发，触发 Systick 中断，Systick 中断会根据当前时钟检查是否需要把某些任务加入就绪列表，在那之后，会触发 PendSV 中断。
4. PendSV 中断：保存当前任务状态，然后选择最高优先级的任务进行切换。它是整个调度器的精髓所在。

## 任务对象

我们得到的任务对象的蓝图如下，对于 arm 架构和 rtos 架构大家可能有点陌生，所以笔者啰嗦一点，任务对象让我们直接开始写代码，在代码中结合框图理解：



Sparrow 是动态创建任务，对于任务栈和任务控制块 (TCB\_t)，我们都要分配内存。xTaskCreat 是用户与 RTOS 的接口，我们可以得到 TCB\_t 的各项参数并赋值，现在还没有就绪列表，所以笔者注释了几行代码。

(usStackDepth - (uint32\_t)1) 是为了获取栈顶的地址，我们分配内存会得到内存的第一个地址，所以加上栈的大小后要减 1 才是栈顶。举例：申请 500，起始地址为 1，那就是 1--500 这部分空间，1+500 -1 得到的值才是 500 这个栈顶。  
& (^... 是字节对齐。

xTaskCreat 完成了：

1. 获得用户设置的参数，例如优先级，栈的大小等等，然后初始化任务控制块。
2. 为任务栈和任务控制块开辟内存
3. 初始化任务栈，获得任务栈的栈顶参数

```

void xTaskCreate( TaskFunction_t pxTaskCode,
                 const uint16_t usStackDepth,
                 void * const pvParameters, //it can be used to debug
                 uint32_t uxPriority,
                 TaskHandle_t * const self )
{

```

```

uint32_t *topStack =NULL;
TCB_t *NewTcb = (TCB_t *)heap_malloc(sizeof(TCB_t));
*self = ( TCB_t *) NewTcb;

NewTcb->uxPriority = uxPriority;
NewTcb->pxStack = ( uint32_t *) heap_malloc( ( ( ( size_t )
usStackDepth ) * sizeof( uint32_t * ) ) );
topStack = NewTcb->pxStack + (usStackDepth - (uint32_t)1) ;
topStack = ( uint32_t *) (((uint32_t)topStack) & (~((uint32_t)
alignment_byte)));
NewTcb->pxTopOfStack
pxPortInitialiseStack(topStack, pxTaskCode, pvParameters, self);

pxCurrentTCB = NewTcb;

}

```

pxPortInitialiseStack 函数负责初始化任务栈，任务栈是保存寄存器的地方，让我们先看看 arm cm3 手册：

## 入栈

响应异常的第一个行动，就是自动保存现场的必要部分：依次把xPSR, PC, LR, R12以及 R3-R0由硬件自动压入适当的堆栈中：如果当响应异常时，当前的代码正在使用PSP，则压入PSP，即使用线程堆栈；否则压入MSP，使用主堆栈。一旦进入了服务例程，就将一直使用主堆栈。

假设入栈开始时，SP的值为N，则在入栈后，堆栈内部的变化如表9.1表示。又因为AHB接口上的流水线操作本质，地址和数据都在经过一个流水线周期之后才进入。另外，这种入栈在机器的内部，并不是严格按堆栈操作的顺序的——但是机器会保证：正确的寄存器将被保存到正确的位置，如图9.1和表9.1的第3列所示。

表9.1 入栈顺序以及入栈后堆栈中的内容

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

笔者在 stack\_register 结构体中已经添加了这些寄存器，让我们边写代码，边听笔者慢慢讲解：：

在 32 位架构下，一个指针大小为一个字，即 32 位，一个寄存器大小也为一个字。

pxTopOfStack-=16, 因为一开始它是指向栈顶的，这是为了容纳 stack\_register

结构体，把这个地址赋值给 stack\_register 结构体，现在结构体的起始地址就是结构体中的第一个寄存器。

然后设置 xPSR 寄存器：xPSR 的 24 位被置 1，表示这是 Thumb 指令状态，实际上 cm3 架构只支持 Thumb 指令状态。

设置 PC 寄存器：pxCode 是任务函数的地址，在 arm 架构中，分为 arm 指令集状态和 Thumb 指令集状态，你可以使用这两种指令集，区别是，Thumb 指令集寄存器最低位是 0，cm3 下都是 Thumb 指令集，所以，为了预防未定义问题，必须把任务的地址最低位进行清 0。

设置 LR 寄存器：LR 寄存器是用于存储任务调用返回地址的专用寄存器，当一个函数被调用时，返回地址会被保存到 LR 寄存器中，以便在函数执行完毕后能够正确返回到调用该函数的下一条指令。但是，rtos 中通常任务是不会返回的，所以这里的 pvParameters 其实是一个笔者用来调试的参数，观察栈有没有被破坏。读者可以把它改成自己的调试函数的地址。

设置 r0：self 也是一个笔者用来调试的参数，当任务执行后，r0 的值会被马上更改，主要是用来判断任务栈是否创建成功。

(\*self)->self\_stack = Stack：设置任务对象的栈对象，这一操作是为了方便调试时找到一个线程的栈地址。

```
uint32_t * pxPortInitialiseStack( uint32_t * pxTopOfStack,
                                   TaskFunction_t pxCode,
                                   void * pvParameters ,
                                   TaskHandle_t * const self)
{
    pxTopOfStack -= 16;
    Stack_register *Stack = (Stack_register *)pxTopOfStack;

    Stack->xPSR = 0x01000000UL;
    Stack->PC = ( ( uint32_t ) pxCode ) & ( ( uint32_t ) 0xffffffffUL );
    Stack->LR = ( uint32_t ) pvParameters;
    Stack->r0 = ( uint32_t ) self;
    (*self)->self_stack = Stack;

    return pxTopOfStack;
}
```

## 总结

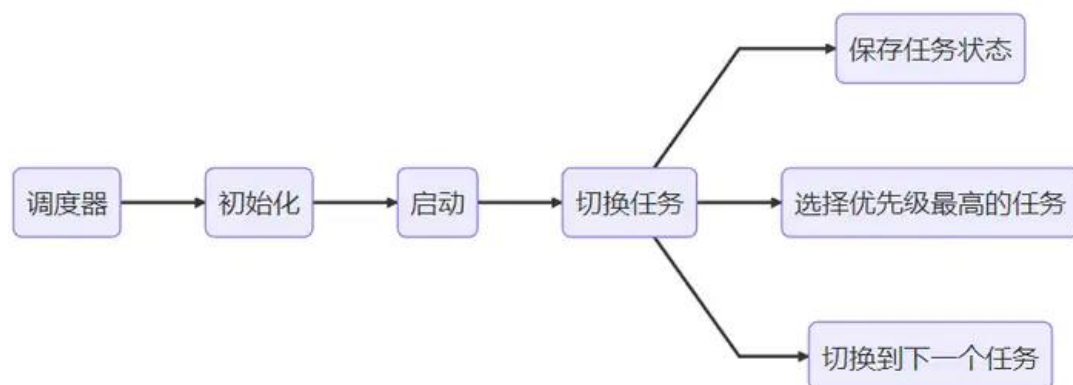
笔者先给出 Sparrow 的框图，根据框图一步步完成了任务的创建以及初始化部分。任务对象的创建已经完成了，下一节笔者将会带领大家完成调度器对象的创建，同时使用 SVC 中断启动第一个任务！

# 调度器创建与启动

## 前言

调度器是整个 RTOS 的核心，在前面我们得到了调度器对象的框架图，并且简单介绍了调度器的原理。

在本节中，我们将会初始化调度器并且启动第一个任务。



## 调度器初始化

调度器的初始化比较简单，我们创建了第一个任务 `leisureTask`，并且把它的各项参数传入任务创建函数中，创建对应的任务控制块与任务栈。由于现在还没有引入多优先级，因此注释掉 `TicksTableInit()` 函数。

第一个任务的内容是让单片机上的灯不断闪烁。

```
void EnterSleepMode(void)
{
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
    HAL_Delay(500);
    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
    HAL_Delay(500);
}

//Task handle can be hide, but in order to debug, it must be created
manually by the user
TaskHandle_t leisureTcb = NULL;

void leisureTask( )
{
    //leisureTask content can be manually modified as needed
    while (1) {
        EnterSleepMode();
    }
}

void SchedulerInit( void )
{
    xTaskCreate(    leisureTask,
                  128,
                  NULL,
                  0,
                  &leisureTcb
    );
}
```

## 调度器启动

调度器启动时，会找到向量表，然后触发 SVC 中断开启第一个任务。



ldr 这三行代码的作用是一找到主栈的栈顶指针，svc 下是特权模式运行，所以使用 msp 作为堆栈指针。然后将栈顶指针的值存储到 msp，现在 msp 就指向栈顶指针了，之后的四行代码就是开启全局中断和异常，然后开启 svc 中断响应。

```
void __attribute__((always_inline)) SchedulerStart( void )
{
    /* Start the first task. */
    __asm volatile (
        " ldr r0, =0xE000ED08 \n" /* Use the NVIC offset register
to locate the stack. */
        " ldr r0, [r0] \n"
        " ldr r0, [r0] \n"
        " msr msp, r0 \n" /* Set the msp back to the
start of the stack. */
        " cpsie i \n" /* Globally enable interrupts.
*/
        " cpsie f \n"
        " dsb \n"
        " isb \n"
        " svc 0 \n" /* System call to start first
task. */
        " nop \n"
        " .ltorg \n"
    );
}
```

关于它是如何找到栈顶指针的，可以参考下图，arm 手册上讲得非常好：

表 7.6 上电后的向量表

地址	异常编号	值 (32 位整数)
0x0000_0000	-	MSP 的初始值
0x0000_0004	1	复位向量 (PC 初始值)
0x0000_0008	2	NMI 服务例程的入口地址
0x0000_000C	3	硬 fault 服务例程的入口地址
...	...	其它异常服务例程的入口地址

因为地址 0 处应该存储引导代码，所以它通常是 Flash 或者是 ROM 器件，并且它们的值不得在运行时改变。然而，为了动态重分发中断，CM3 允许向量表重定位——从其它地址处开始定位各异常向量。这些地址对应的区域可以是代码区，但也可以是 RAM 区。在 RAM 区就可以修改向量的入口地址了。为了实现这个功能，NVIC 中有一个寄存器，称为“向量表偏移量寄存器”（在地址 0xE000\_ED08 处），通过修改它的值就能定位向量表。但必须注意的是：向量表的起始地址是有要求的：必须先求出系统中共有多少个向量，再把这个数字向上增大到是 2 的整次幂，而起始地址必须对齐到后者的边界上。例如，如果一共有 32 个中断，则共有 32+16（系统异常）=48 个向量，向上增大到 2 的整次幂后值为 64，因此地址地址必须能被 64\*4=256 整除，从而合法的起始地址可以是：0x0, 0x100, 0x200 等。向量表偏移量寄存器的定义如表 7.7 所示。

表 7.7 向量表偏移量寄存器(VTOR) (地址：0xE000\_ED08)

位段	名称	类型	复位值	描述
29	TBLBASE	RW	0	向量表是在 Code 区 (0)，还是在 RAM 区 (1)

简单来说就是，先找到向量表偏移量寄存器，再找到向量表，向量表记录了 msp 的初始值：

向量表 s

当一个发生的异常被 CM3 内核接受，对应的异常 handler 就会执行。为了决定 handler 的入口地址，CM3 使用了“向量表查表机制”。这里使用一张向量表。向量表其实是一个 WORD（32 位整数）数组，每个下标对应一种异常，该下标元素的值则是该异常 handler 的入口地址。向量表的存储位置是可以设置的，通过 NVIC 中的一个重定位寄存器来指出向量表的地址。在复位后，该寄存器的值为 0。因此，在地址 0 处必须包含一张向量表，用于初始时的异常分配。

表 3.5 向量表结构

异常类型	表项地址 偏移量	异常向量
0	0x00	MSP 的初始值
1	0x04	复位
2	0x08	NMI
3	0x0C	硬 fault
4	0x10	MemManage fault
5	0x14	总线 fault

## 开启第一个任务

采用封装的思想看待 SVC 调用，其实就是它把任务栈中的内容加载到了 CPU 的寄存器里面。

首先看代码：

```
#define vPortSVCHandler SVC_Handler

void __attribute__(( naked )) vPortSVCHandler( void )
{
    __asm volatile (
        "    ldr r3, pxCurrentTCBConst2      \n"
        "    ldr r1, [r3]                    \n"
        "    ldr r0, [r1]                    \n"
        "    ldmia r0!, {r4-r11}              \n"
        "    msr psp, r0                      \n"
        "    isb                             \n"
        "    mov r0, #0                       \n"
        "    msr basepri, r0                  \n"
        "    orr r14, #0xd                    \n"
        "    bx r14                          \n"
        "                                     \n"
        "    .align 4                         \n"
        "pxCurrentTCBConst2: .word pxCurrentTCB \n"
    );
}
```

这里就是把任务栈中的内容加载到 CPU，还记得任务栈中的内容吗？先看看任务栈的结构体笔者再解释代码：

```
Class(Stack_register)
{
    //automatic stacking
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
```

```

uint32_t r10;
uint32_t r11;
//manual stacking
uint32_t r0;
uint32_t r1;
uint32_t r2;
uint32_t r3;
uint32_t r12;
uint32_t LR;
uint32_t PC;
uint32_t xPSR;
};

```

前面的代码的含义就是：先令 r3 寄存器的值为 pxCurentTCB 的地址，再把这个地址指向的内容给 r1，现在 r1 存储的就是指针，我们知道指针就是地址，再把这个指针的值给 r0，那么 r0 就会找到任务控制块。

pxCurentTCB 的作用是指向当前运行的任务或即将运行的任务的控制块（TCB）。前面三行代码的作用是获取 pxCurentTCB 指向的任务栈，因为 TCB 的第一个成员就是栈顶指针。

从 r0 这个内存地址开始，把栈中往上九个地址的内容依次加载到 CPU 的寄存器 r4-r11 和 r14，然后 r0 自增（加载后 r0 寄存器刚好表示这个栈里面的 r0 位置），执行任务时将会使用 psp，于是我们更新 psp 这个栈指针。

然后清零 r0 寄存器，然后设置 basepri 寄存器为 0，就是打开所有中断，basepri 寄存器是一个用来控制屏蔽中断的寄存器，我们将会在临界区学习它。

最后两行代码的作用就是返回。

关于 basepri 寄存器，读者可以看看这些：

## 特殊功能寄存器

Cortex-M3 还在内核水平上搭载了若干特殊功能寄存器，包括  
程序状态字寄存器组（PSRs）  
中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）  
控制寄存器（CONTROL）

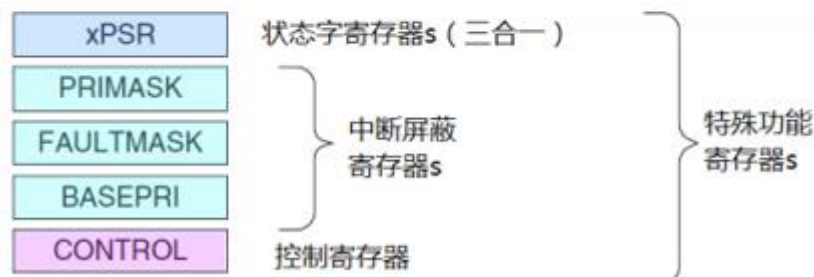


图 2.3：Cortex-M3 中的特殊功能寄存器集合

表 2.1 寄存器及其功能

寄存器	功能
xPSR	记录 ALU 标志（0 标志，进位标志，负数标志，溢出标志），执行状态，以及当前正服务的中断号
PRIMASK	除能所有的中断——当然了，不可屏蔽中断（NMI）才不甩它呢。
FAULTMASK	除能所有的 fault——NMI 依然不受影响，而且被除能的 faults 会“上访”，见后续章节的叙述。
BASEPRI	除能所有优先级不高于某个具体数值的中断。
CONTROL	定义特权状态（见后续章节对特权的叙述），并且决定使用哪一个堆栈指针

第 3 章对此有展开的叙述。

## 实验

在写下以上代码后，我们可以尝试编译代码，下载到开发板中，我们会发现 stm32f103c8t6 最小系统板上的指示灯闪烁。

## 总结

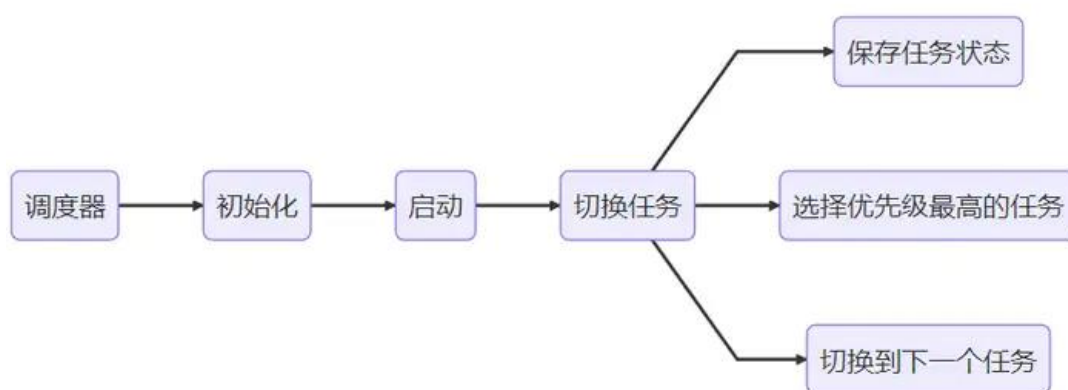
创建了调度器对象并且使用 SVC 中断从任务栈中加载内容到 CPU 寄存器中，从而

开启第一个任务。

# 实现多线程

## 前言

在上一篇博客中，我们使用 SVC 中断开启了第一个任务，这就是调度器的启动，接下来我们将在 SVC 工程的基础上，添加 PendSV 中断，进行切换任务从而实现多线程。



## PendSV 中断

根据前面的博客，我们已经知道了 PendSV 中断是上下文切换的真正场所，现在让我们利用 PendSV 中断实现上下文切换。

代码如下：

先在合适的地方加上这两行宏：

```
#define configMaxPriori 32  
#define configShieldInterPriority 191
```

我们先在调度器启动的函数里配置中断：

```
void __attribute__((always_inline)) SchedulerStart( void )  
{
```

```

//添加这一行
( *( ( volatile uint32_t * ) 0xE000ED20 ) ) |= ( ( ( uint32_t )
255UL ) << 16UL );
/* Start the first task. */
__asm volatile (
    " ldr r0, =0xE000ED08  \n"/* Use the NVIC offset register
to locate the stack. */
    " ldr r0, [r0]          \n"
    " ldr r0, [r0]          \n"
    " msr msp, r0           \n"/* Set the msp back to the
start of the stack. */
    " cpsie i               \n"/* Globally enable interrupts.
*/
    " cpsie f               \n"
    " dsb                   \n"
    " isb                   \n"
    " svc 0                 \n"/* System call to start first
task. */
    " nop                   \n"
    " .ltorg                \n"
    );
}

```

## PendSV:

我们要先保存上文，选择任务，再开启下文。

简单解释一下这些代码，其实如果读者看得懂 SVC 中断的过程，PendSV 中断也大差不差：

当 PendSV 中断响应时，xpsr, pc, (r14) lr, r12, r3, r2, r1, r0 的值会被自动存储到任务栈中，有读者可能会好奇，这是怎么找到任务栈的呢？不要忘记了 PSP！它会指向这个任务栈的栈顶，也就是说 psp 现在指向 r0。

现在我们要手动保存 stack 中 r0 数据下面的值。

pxCurrentTCB 指向的是 TCB，也就是任务控制块，前一行代码，是将 pxCurrentTCB 这个指针的地址加载到 r3；后一句，是将这个地址指向的内容加载到 r2，也就是说，r2 现在的内容就是 pxCurrentTCB 这个指针；如果再进行一次 ldr 操作，就能找到 TCB 了，读者可以去 SVC 调用的篇章看看。

接下来的这几行代码笔者之前讲过，就是进入临界区，关闭异常和中断，防止被打断。

执行 vTaskSwitchContext 函数后再退出临界区。



栈指针 sp 是保存作用，这里是将 r3 压入栈，r3 保存的是 pxCurrentTCB 指针的地址，也就是一个二级指针。这是因为在调用 vTaskSwitchContext 函数后，pxCurrentTCB（一级指针）的值会被更改为指向下一个任务，所以我们需要保存它的地址（二级指针），这样就能够找到切换后的任务的地址（一级指针）。之后是加载对应的任务栈的内容到寄存器，上下文切换是对称的。切换下文的内容可以参考 SVC 调用及之后的博客，读者看几遍就能理解了。顺便一提，attribute( ( naked ) )不要漏掉，这个关键字笔者还是不多赘述了，深究下去涉及到编译器对程序的处理，有兴趣的读者可以尝试看一下添或不添加这个关键字后的反汇编源码。

先看看任务对象结构体：

```
Class(TCB_t)
{
    volatile uint32_t * pxTopOfStack;
    unsigned long uxPriority;
    uint32_t * pxStack;
    Stack_register *self_stack;
};
```

让我们再看看栈中寄存器的值的布局，从 r4 到 xPSR，地址不断增加，xPSR 处于栈中最高的地址

```
Class(Stack_register)
{
    //manual stacking
    uint32_t r4;
    uint32_t r5;
    uint32_t r6;
    uint32_t r7;
    uint32_t r8;
    uint32_t r9;
    uint32_t r10;
    uint32_t r11;
    //automatic stacking
    uint32_t r0;
    uint32_t r1;
    uint32_t r2;
    uint32_t r3;
    uint32_t r12;
    uint32_t LR;
    uint32_t PC;
    uint32_t xPSR;
};
```

```

void __attribute__(( naked )) xPortPendSVHandler( void )
{
    __asm volatile
    (
        "      mrs r0, psp                                \n"
/*
把 psp 存储到 r0, psp 之前指向的是栈顶, arm 架构进入异常模式时, 硬件会自动保存寄存器同时更新 psp 的值, 现在 psp 的地址就是 r0 处。也就是我们需要手动保存的寄存器的部分
*/
        "      isb                                        \n" //屏障指令, 不影响理解
        "                                              \n"
        "      ldr    r3, pxCurrentTCBConst              \n"
// 加载当前任务对象结构体的地址的地址到 r3, r3 相当于二级指针
        "      ldr    r2, [r3]                            \n"
//再加载一次, 现在 r2 的内容就是任务对象结构体的地址
        "                                              \n"
        "      stmdb r0!, {r4-r11}                        \n"
/*此时从栈中的保存 r0 的地址开始, 存储 CPU 中 r4 到 r11 的值到栈中, 同时 r0 的值不断递减,
存储一个地址 r0 保存的地址的值就减 1, 也就是指向下一个更低的地址*/
        "      str r0, [r2]                                \n"
//把 r0 的值存储到 r2 指向的地址处, 也就是改变任务对象结构体的第一个元素 pxTopOfStack 的值
        "                                              \n"
        "      stmdb sp!, {r3, r14}                       \n"
/*这里是存储 r3 和 r14 的值, r3 相当于当前任务对象结构体的二级指针,
因为我们将要改变它的一级指针的指向, 让它指向另一个任务对象结构体*/
        "      mov r0, #0                                  \n" //进入临界区
        "      msr basepri, r0                            \n" //
        "      dsb                                          \n" //
        "      isb                                          \n"
        "      bl vTaskSwitchContext                      \n"
//改变 pxCurrentTCB 的值, 它是一级指针, 现在它指向了另一个任务对象结构体
        "      mov r0, #0                                  \n" //
        "      msr basepri, r0                            \n" // 退出临界区
        "      ldmbia sp!, {r3, r14}                      \n" // 加载保存的值
        "                                              \n"

```

```

        "    ldr r1, [r3]                                \n"
//r3 是二级指针，加载后得到当前任务对象结构体的一级指针
        "    ldr r0, [r1]                                \n"
/*再加载一次，现在 r0 的值就是切换后的任务对象结构体的第一个元素
pxTopOfStack 的值*/
        "    ldmia r0!, {r4-r11}                        \n"
/*同理，退出异常模式时，硬件会自动帮助我们完成 r0 上面的寄存器的加载，
我们只需要
加载下面的部分到 CPU 的寄存器即可，每加载一次，r0 的值就加 1，指向下一个
更高的地址*/
        "    msr psp, r0                                \n"
/*加载完后，现在 r0 的值就是栈顶地址，把这个值给 psp 保存，
方便硬件自动加载或者保存寄存器的值时能找到地方*/
        "    isb                                        \n"
        "    bx r14                                    \n"
/*返回，进入线程模式，因为我们上下文切换是在中断中进行的，也就是说我们在
arm 架构的异常模式，所以我们要返回进入线程模式，使用的栈指针也从 msp 变
成了 psp*/
        "    nop                                        \n"
        "    .align 4                                  \n"
//这里并没有浮点寄存器，4 字节对齐即可
        "pxCurrentTCBConst: .word pxCurrentTCB \n"
        "::"i" ( configShieldInterPriority )
    );
}

```

由于现在还不支持多优先级，因此采用轮询机制选择任务：

```

uint32_t x = 0;
void vTaskSwitchContext( void )
{
    x++;
    pxCurrentTCB = TcbTaskTable[ x %3 ];
}

```

现在我们需要加入就绪表和上下文切换的触发函数了。

在定义 SVC\_Handler 的地方加上这几行：

switchTask 是用来触发 PendSV 中断，往对应的地址位写入 1 即可，读者有兴趣可以查阅 arm cm3 手册，笔者不过多赘述：

```

#define switchTask()\
*( ( volatile uint32_t * ) 0xe000ed04 ) = 1UL << 28UL;

```

```
TaskHandle_t TcbTaskTable[configMaxPriori];
```

```
#define vPortSVCHandler SVC_Handler
```

```
#define xPortPendSVHandler PendSV_Handler
```

更改一下创建任务的函数：

```
void xTaskCreate( TaskFunction_t pxTaskCode,
                 const uint16_t usStackDepth,
                 void * const pvParameters, //You can use it for
debugging
                 uint32_t uxPriority,
                 TaskHandle_t * const self )
{
    uint32_t *topStack =NULL;
    TCB_t *NewTcb = (TCB_t *)heap_malloc(sizeof(TCB_t *));
    *self = ( TCB_t *) NewTcb;

    TcbTaskTable[uxPriority] = NewTcb;//添加这一行

    NewTcb->uxPriority = uxPriority;
    NewTcb->pxStack = ( uint32_t *) heap_malloc( ( ( ( size_t )
usStackDepth ) * sizeof( uint32_t * ) ) );
    topStack = NewTcb->pxStack + (usStackDepth - (uint32_t)1) ;
    topStack = ( uint32_t *) (((uint32_t)topStack) & (~(uint32_t)
alignment_byte)));
    NewTcb->pxTopOfStack = topStack;
    pxPortInitialiseStack(topStack, pxTaskCode, pvParameters, self);
    pxCurrentTCB = NewTcb;
}
```

在此之前，记得把空闲任务的内容更改，因为我们这次的实验还是点灯：

```
void EnterSleepMode(void)
{
    //什么都不做
}
```

```
//Task handle can be hide, but in order to debug, it must be created
```

manually by the user

```
TaskHandle_t leisureTcb = NULL;
```

```
void leisureTask( )
```

```
{//leisureTask content can be manually modified as needed
```

```
    while (1) {
```

```
        switchTask();//自动轮询到下一个
```

```
    }
```

```
}
```

现在我们可以添加多个任务了,把 main 函数这一块改成这样:

```
//Task Area!The user must create task handle manually because of  
debugging and specification
```

```
TaskHandle_t tcbTask1 = NULL;
```

```
TaskHandle_t tcbTask2 = NULL;
```

```
void led_bright( )
```

```
{
```

```
    while (1) {
```

```
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
```

```
        HAL_Delay(1000);
```

```
        switchTask();
```

```
    }
```

```
}
```

```
void led_extinguish( )
```

```
{
```

```
    while (1) {
```

```
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
```

```
        HAL_Delay(500);
```

```
        switchTask();
```

```
    }
```

```
}
```

```
void APP( )
```

```
{
```

```
    xTaskCreate(    led_bright,
```

```
                  128,
```

```
                  NULL,
```

```
                  1,
```

```

                                &tcbTask1
);

xTaskCreate(    led_extinguish,
               128,
               NULL,
               2,
               &tcbTask2
);
}

```

```

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    SchedulerInit();
    APP();

    SchedulerStart();

    while (1)
    {

    }
}

```

## 实验验证

读者在完成以上这些后，可以编译下载到 stm32f103c8t6 最小系统板中。实验现象为开发板上的灯不断闪烁。

## 总结

完成了 PendSV 中断的代码，并且引入任务就绪表。通过 PendSV 中断不断进行上下文切换，从而实现多线程。

## 问题

1. 我们使用 SVC 中断开启第一个任务，使用 systick 定时触发 pendsv 中断，pendsv 中断完成上下文切换，可不可以使用 systick 中断完成上下文切换？如果可以，怎么设计上下文在 systick 中断的执行位置？
2. 能不能合并 svc、systick、pendsv 三个中断？如果可以，哪个中断会被保留？在这个中断里怎么设计上下文切换算法？

# 并发与竞态

## 前言

在前面一篇文章中，笔者带领大家使用 PendSV 中断实现了多线程，也终于度过了关于 arm 架构最难的部分。现在，笔者可以带领大家稍微迈入一点大家所熟悉的操作系统的世界，也就是多线程的并发问题。

## 操作系统中的并发与竞态

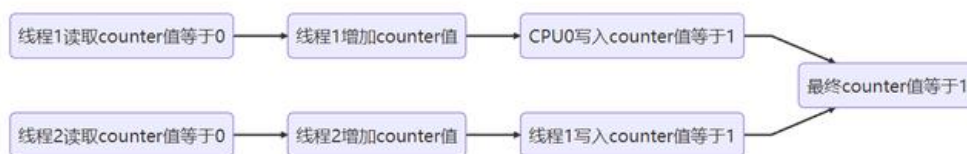
试想，RTOS 中如果两个线程同时对一个全局资源进行更改，那么会发生什么情况？例如，A 线程要向一个地址写入数据，恰好 B 线程也要向一个地址写入数据，这种情况下会发生什么呢？

显然，最终的结果是第二个完成写入的线程胜利。

举个详细一点的例子，两个线程同时递增一个变量，会发生什么呢？

考虑下面的情况：假设 counter 等于 0，两个线程同时读取 counter 变量的值并递增。

我们理所应当认为 counter 应该等于 2，实际上，counter 会等于 1。



我们把这种因为线程竞争而出现的问题称之为竞态。

竞态通常作为对资源的共享访问而产生，当两个线程访问相同的资源时，混合的可能性就永远存在。所以，笔者给出两点建议：

### 1. 只要可能，就应该避免资源的共享

这种思想最直接的方式就是减少全局变量的使用。如果我们使用全局变量，必须要考虑是否会造成多个线程并发访问的场景。然而，在实际开发中，这其实是一种奢望，因为硬件资源本身就是共享的，而且有时候为了内存不得不使用全局变量。

### 2. 必须显式地管理对共享资源的访问



我们必须确保一次只有一个线程访问共享资源，那么，我们该如何做到呢？

## 临界区与原子操作

共享资源往往被称之为临界资源，为了使得操作共享资源时只有一个线程，我们引入了临界区和原子操作的概念。它们的本质是一样的，都是在执行过程中不能被打断的一段代码。

### 临界区

在 RTOS 中通常是通过开关中断来实现。当然，早期的 unix 和 linux 等操作系统不支持对称多处理时，并发执行的唯一原因也是硬件中断服务，开关中断也是一种常用的实现对资源的唯一访问的手段。

### 原子操作

不一定通过开关中断实现，也可以通过汇编（ldrex 和 strex 等指令）或者锁等方式来实现对资源的唯一访问。

## 如何防止临界区被打断？

在 Sparrow RTOS 中，我们使用 PendSV 中断来完成上下文切换，从而实现多线程，这可能会造成竞态的出现。还有一种可能，就是外部中断的发生，导致当前任务对共享资源的访问被打断。我们发现，Sparrow 中的并发都来自中断，那么，只要在对临界资源的访问的程序中加上关闭中断的操作，这段程序就只有当前线程执行了，当然，执行完后我们肯定要开启中断。

先分析 RTOS 的运行架构，以 cm3 为例，它的中断屏蔽寄存器如下：

## 特殊功能寄存器

Cortex-M3 还在内核水平上搭载了若干特殊功能寄存器，包括  
程序状态字寄存器组（PSRs）  
中断屏蔽寄存器组（PRIMASK, FAULTMASK, BASEPRI）  
控制寄存器（CONTROL）

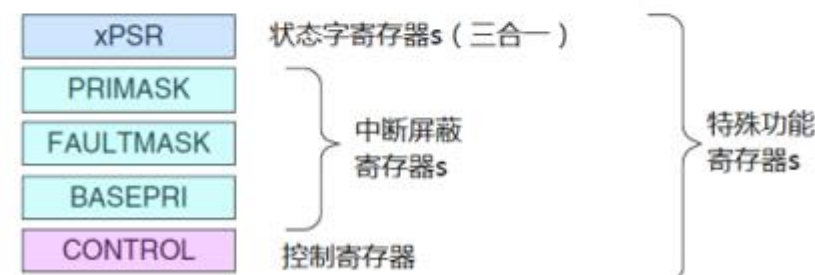


图 2.3：Cortex-M3 中的特殊功能寄存器集合

详细信息：

表 3.2 Cortex-M3 的屏蔽寄存器组

名字	功能描述
PRIMASK	这是个只有单一比特的寄存器。在它被置 1 后，就关掉所有可屏蔽的异常，只剩下 NMI 和硬 fault 可以响应。它的缺省值是 0，表示没有关中断。
FAULTMASK	这是个只有 1 个位的寄存器。当它置 1 时，只有 NMI 才能响应，所有其它的异常，甚至是硬 fault，也通通闭嘴。它的缺省值也是 0，表示没有关异常。
BASEPRI	这个寄存器最多有 9 位（由表达优先级的位数决定）。它定义了被屏蔽优先级的阈值。当它被设成某个值后，所有优先级号大于等于此值的中断都被关（优先级号越大，优先级越低）。但若被设成 0，则不关闭任何中断，0 也是缺省值。

现在我们将要使用 BASEPRI 寄存器，其实用其他两个都是可以的。

## 进入临界区

configShieldInterPriority 是宏，其值为 191，由于 BASEPRI 寄存器是一个八位的寄存器，它是高四位有效，所以它的值是 11，默认屏蔽 11 及以上的中断。下面的代码就是往 basepri 寄存器中写入 191，return xReturn 是考虑代码进入临界区时，可能又发生中断，中断里面也有临界区，当退出中断时，此时 basepri 的值就会变成默认状态的 0，这段代码之后的临界区就是无效的：

```
__attribute__((always_inline)) inline uint32_t xEnterCritical( void )
```

```

{
    uint32_t xReturn;
    uint32_t temp;

    __asm volatile(
        " cpsid i                \n"
        " mrs %0, basepri        \n"
        " mov %1, %2              \n"
        " msr basepri, %1         \n"
        " dsb                     \n"
        " isb                     \n"
        " cpsie i                \n"
        : "=r" (xReturn), "=r"(temp)
        : "r" (configShieldInterPriority)
        : "memory"
    );

    return xReturn;
}

```

## 退出临界区

将之前保存的值写入 basepri 寄存器即可：

```

__attribute__((always_inline)) void xEixtCritical( uint32_t xReturn )
{
    __asm volatile(
        " cpsid i                \n"
        " msr basepri, %0         \n"
        " dsb                     \n"
        " isb                     \n"
        " cpsie i                \n"
        :: "r" (xReturn)
        : "memory"
    );
}

```

## 实验

现在我们进行一个小实验来验证我们的临界区是否成功实现，我们需要验证临界区是否能够屏蔽中断，那我们使用什么中断进行验证呢？

不要忘了上下文切换函数 PendSV 本身就是一个中断。

先加上我们自己写的延时函数（hal 库的 hal\_delay 容易出 bug 导致卡死, 与它的计数默认使用中断进行有关），修改 led\_bright 函数如下：

```
void fone(uint32_t time)
{
    uint8_t i = 0;
    while (time-->0)
    {
        i = 10;
        while (i-->0)
            ;
    }
}
```

```
void ftwo(uint32_t time)
{
    uint8_t i = 0;
    while (time-->0)
    {
        i = 10;
        while (i-->0)
            ;
    }
}
```

```
void theone(uint32_t one)
{
    while (one-->0)
    {
        fone(1000);
    }
}
```

```
void thetwo(uint32_t two)
{
    while (two-->0)
    {
        ftwo(1000);
    }
}
```

```

uint32_t count = 0;
void led_bright( )
{
    while (1) {
        uint32_t xre = xEnterCritical();

        switchTask();
        theone(1000);
        count++;
        xExitCritical(xre);
    }
}

```

编译然后下载到开发板上进行调试，先介绍一下实验思路；我们在 count++ 上面添加了一个延时 1s 的函数。如果临界区没有屏蔽中断，那么延时肯定会触发 PendSV 中断，那我们运行到中断里面时，count 的值仍然是 0。如果临界区是正确的，那么 count 的值会是 1，这证明是在延时 1s 后才触发的中断。

先打上断点；



The screenshot shows a code editor with a dark background. The code is the same as in the first block. A red circular breakpoint is set on line 486, which is the line containing `uint32_t xre = xEnterCritical();`. The line numbers on the left range from 481 to 494. The code is as follows:

```

481
482     uint32_t count = 0;
483     void led_bright( )
484     {
485         while (1) {
486             uint32_t xre = xEnterCritical();
487
488             switchTask();
489             theone( one: 1000);
490             count++;
491             xExitCritical( xReturn: xre);
492         }
493     }
494

```

gdb 下，按 c 运行到断点处，现在 count 为 0。

在 PendSV 中断这里也打上断点：

```
262
263 ↩ void __attribute__(( naked )) xPortPendSVHandler( void )
264 {
265     ↩ __asm volatile
266     (
267         "    mrs r0, psp                \n"
268         "    isb                        \n"
269         "                                \n"
270         "    ldr r3, pxCurrentTCBConst  \n"
271         "    ldr r2, [r3]               \n"
272         "                                \n"
273         "    stmdb r0!, {r4-r11}         \n"
274         "    str r0, [r2]               \n"
275         "                                \n"
276         "    stmdb sp!, {r3, r14}        \n"
```

按 r 运行到 PendSV 中断里面。

打印：

```
(gdb) p count
$3 = 1
```

结果是 1，说明我们的临界区起作用了。

## 总结

讲解了操作系统中的一些并发问题，然后通过配置中断寄存器的方式实现了临界区，最后通过实验来验证临界区是否成功执行。

## 问题

能不能使用其他的中断控制寄存器完成临界区的设计？怎么设计？

# 调度策略

## 前言

在前面我们完成了 Sparrow 的临界区的代码，使用临界区，能够解决常见的并发问题，现在该完善我们的调度算法了。

调度算法在操作系统领域常常是热门的话题。不同的用途将会使用不同的调度策略。在本节，笔者将为大家介绍一些操作系统中的调度算法的知识，加深读者对操作系统的理解。

本文将介绍 Linux0.11 版本、FreeRTOS、RT-Thread 三种 OS 的调度策略。

### 非实时操作系统

对于非实时操作系统 Linux，调度策略的一个重要参数就是公平性，如果有  $n$  个任务，那么非实时操作系统的调度策略倾向于每个任务获得  $1/n$  的 CPU 时间。

### 实时操作系统

实时操作系统的调度算法强调可预测性和确定性，实时性并不是指系统运行有多快，而是整个系统可预测。从这一点来看，其实 FreeRTOS 和 RT-Thread 这些小型 RTOS 的实时性一般，这是其本身调度算法的限制。因为小型 RTOS 通常在资源受限的场景使用，对实时性的需求并不严苛，其调度算法强调在确定的时间内找出最高优先级的任务，但并不侧重整个系统的确定性预测。

### linux 内核中的调度策略

调度策略负责进行任务选择和任务切换，linux 内核中的调度基于分时技术：多个进程以“时间多路复用”的方式运行，将时间进行分割，不同段时间运行不同的任务，这段时间被称为时间片。

## 进程的分类

linux 内核中通常有三类进程：

交互式进程：用于和用户发生交互，对时间不是很敏感，时不时卡一卡也是在用户容许范围内。

批处理进程：在后台运行的进程，不与用户直接接触，例如编译程序和科学计算程序等等。

实时进程：对时间要求非常严格的程序，例如自动控制程序。

进程的抢占

linux 内核是抢占式的，也就是说，不同的进程有不同的优先级，如果响应的进程的优先级比当前运行的进程的优先级要高，那么当前的进程就会被响应的进程所抢占。

调度算法

早期 linux 版本的内核的调度算法比较简单：扫描并且计算所有就绪进程的优先级，然后选择最佳的进程来运行，参考的标准是时间片，时间片大的进程往往优先运行。这一算法的缺点非常明显，就是时间不固定。

当然，到了 2.6 版本之后，linux 内核的调度算法已经复杂到不可想象的程度了，这跟多处理器的出现也有很大的关系，此时，调度选择算法会在固定的时间选出任务执行，以确保 CPU 的运行速度。

## Linux 0.11 版本调度算法

让我们先看看 linux 内核 0.11 版本的调度算法。

```
while (1) {
    c = -1;
    next = 0;
    i = NR_TASKS;
    p = &task[NR_TASKS];

    while (--i) {
        if (!*--p)
            continue;
        if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
            c = (*p)->counter, next = i;
    }

    if (c) break;

    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
        if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
}

switch_to(next);
```

首先是前几行：

c 其实代表的是保存进程时间片的值

next 表示下一个进程的序列

i, p 都与挂载进程的任务队列有关，i 代表数组下标，p 是指向队列本身的指针。

```
while (--i) {
    if (!*--p)
        continue;
    if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
        c = (*p)->counter, next = i;
}
```

其实这就是一个很常见的比较大小的算法，首先检查这个进程的状态是不是运行状态，然后进行比较，如果在运行队列里遇到比当前进程时间片大的进程，那么就替换成它，c 更改为这个大的进程的时间片，next 更改为这个进程的下标，如图：





通过这个比较算法，我们筛选出了队列里时间片最大的任务。接下来就是切换了，如果要切换的进程的时间片没有用完，也就是  $c > 0$  时，这个 while 循环会直接结束，程序会跳转到 switch 这里，这就是进程切换的函数。

```

    if (c) break;

    for (p = &LAST_TASK; p > &FIRST_TASK; --p)
        if (*p) (*p)->counter = ((*p)->counter >> 1) + (*p)->priority;
}

```

如果这个优先级高的进程的时间片已经用完了，也就是  $c == 0$  时，会进入 for 循环，这是对时间片的重新分配。因为我们从前面的排序已经知道，c 就是被筛选出来的最大的时间片，如果  $c == 0$ ，意味着所有的进程都没有了时间片。

上面的 for 循环，其实就是根据优先级的大小来调整时间片的大小，最后每一个进程 counter 的值 = 原先值 / 2 + 优先级。

可能读者还会疑惑，为什么要加上  $*p \rightarrow \text{counter} \gg 1$  呢？位于运行状态的进程的时间片不是都为 0 了吗？

进程不止一种状态，显然，还在被阻塞的任务时间片肯定不为 0 啊，虽然运行状态的进程最终时间片都等于自身优先级的大小，但是阻塞任务就要考虑原先的时间片了，对原先时间片除以 2，这是对平均时间片的综合考虑的结果，时间片过短，切换的开销会变大，时间片过长，进程看起来就不会是并行的。

现在让我们来到 switch\_to 函数：

```

"cmpl %kexx, _current\n\t" \, 比较当前任务指针 (_current) 与任务 n 的指针 (task[n]) 是否相同
"je if\n\t" \, 如果相同，则跳转到标号 1，意味着当前任务与任务 n 相同，不需要切换
"movl %dx, %i\n\t" \, 将任务 n 的 TSS (任务状态段) 地址 (TSS(n)) 存储到 __tmp.b 中
"xchgl %kexx, _current\n\t" \, 交换当前任务指针 (_current) 与任务 n 的指针 (task[n])
"ljmp %0\n\t" \, 执行远跳转到任务 n 的代码段 (使用任务 n 的段选择子)
"cmpl %kexx, _last_task_used_math\n\t" \, 比较上一个使用浮点运算的任务指针 (_last_task_used_math) 与任务 n 的指针是否相同
"jne if\n\t" \, 如果不相同，则跳转到标号 1，意味着需要清除任务切换标志
"clts\n\t" \, 清除任务切换标志 (CR0 寄存器的 TS 位)
"1:" \, 标号 1，用于跳转

::"m" (*__tmp.a), "m" (*__tmp.b), \设置输入，进行内联汇编
"d" (TSS(n)), "c" ((long)task[n]), \传递TSS

```

在前文笔者简单讨论过进程的上下文切换，因为进程之间隔绝了资源，这往往意味着切换的步骤会变得简单一些，从代码中可以看出，主要是 TSS（一个保存进程状态信息的结构体）的切换，

通过 schedule 和 switch\_to 两个函数，我们知道了 linux0.11 版本是如何进行调度的，schedule 负责时间片的分配和进程的选择，switch\_to 负责进程的切换。操作系统的调度器本质就是在进行选择 and 切换。

# RTOS 内核

## 调度算法

现在该来到常见的 RTOS 内核了

RTOS 使用链表的调度算法一般如下：

使用一个数组 `array[Index]` 来存储多个链表头节点，`Index` 一般是优先级。会有一个 `Table` 用作位开关，例如定义一个 `uint32 Table = 00001111`，低四位为 1 表示 0, 1, 2, 3 这四个优先级都有任务，高四位为 0 表示上面的优先级都没任务。

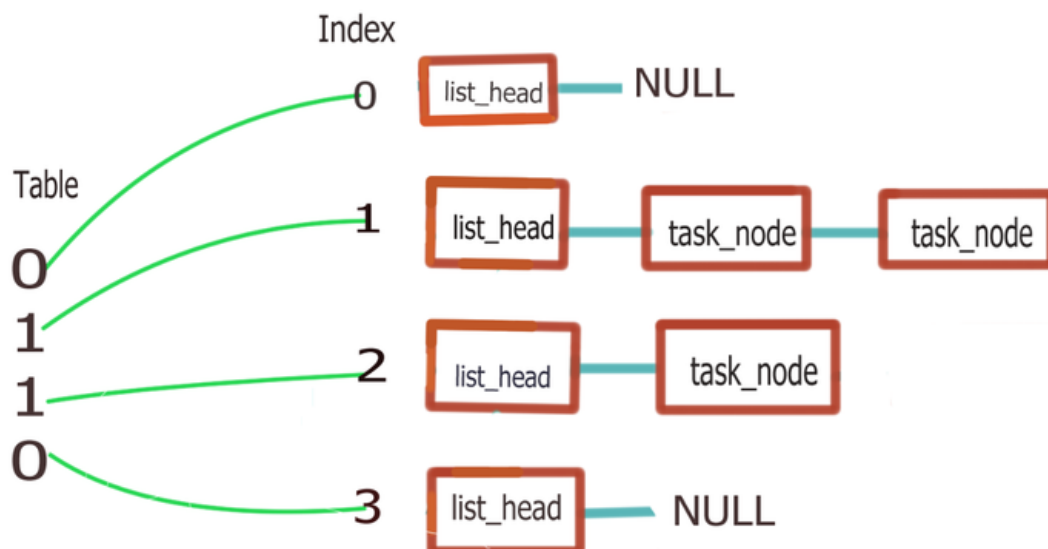
### 调度算法一

我们可以顺着数组的优先级下标，从优先级高的任务开始，通过 `list_head` 一个个找，直到任务链表里面有任务节点，那么这个任务一定是就绪链表中最高优先级的任务，接下来 CPU 就会执行这个任务。如果这个优先级里面还有其他任务，那么它们轮流共享 CPU 的时间片。

### 调度算法二

我们可以看 `Table` 的各个位的值，既然用 1 或 0 表示是否有任务，那么从高位往低位数，第一个出现的 1，代表这个优先级的任务是就绪链表中最高优先级的任务，同理，接下来 CPU 就会执行这个优先级的任务。

如图所示，显然，我们要寻找的最高优先级的任务节点就在下标为 1 的链表上：



接下来笔者会讲解 FreeRTOS 和 RT-Thread 的调度算法，分别对应算法一和算法二。

## FreeRTOS 的调度算法

让我们看看 tasks.c 文件中的这几行代码：

```
#if ( configUSE_PORT_OPTIMISED_TASK_SELECTION == 0 )

/* If configUSE_PORT_OPTIMISED_TASK_SELECTION is 0 then task selection is
performed in a generic way that is not optimised to any particular
microcontroller architecture. */

/* uxTopReadyPriority holds the priority of the highest priority ready
state task. */
#define taskRECORD_READY_PRIORITY( uxPriority )
{
    if( ( uxPriority ) > uxTopReadyPriority )
    {
        uxTopReadyPriority = ( uxPriority );
    }
} /* taskRECORD_READY_PRIORITY */
```

prvAddTaskToReadyList 函数在每个任务在添加到就绪列表时都会执行，它会与 uxTopReadyPriority 进行比较，然后 uxTopReadyPriority 记录两者中的较大者，不断的对每一个新添加进来的任务优先级进行比较，这其实就是在找出最大的优先级。

这样做，遍历时就能直接找到优先级最高的线程。

```
/*
#define prvAddTaskToReadyList( pxTCB )
{
    traceMOVED_TASK_TO_READY_STATE( pxTCB );
    taskRECORD_READY_PRIORITY( ( pxTCB )->uxPriority );
    vListInsertEnd( &(amp; pxReadyTasksLists[ ( pxTCB )->uxPriority ] ), &(amp; ( pxTCB )->xStateListItem ) );
    tracePOST_MOVED_TASK_TO_READY_STATE( pxTCB );
}
```

顺便一提，FreeRTOS 中的任务优先级跟我们在 mcu 中学习到的中断优先级判断是不一样的，也跟 RTT 和 uc/os 不一样，mcu 中的中断优先级是数字越小，优先级越大，RTT 和 uc/os 的任务优先级也是这样，但 FreeRTOS 中是任务优先级的数字越大，优先级越大。

```
#define taskSELECT_HIGHEST_PRIORITY_TASK()
{
    UBaseType_t uxTopPriority = uxTopReadyPriority;

    /* Find the highest priority queue that contains ready tasks. */
    while( listLIST_IS_EMPTY( &(amp; pxReadyTasksLists[ uxTopPriority ] ) ) )
    {
        configASSERT( uxTopPriority );
        --uxTopPriority;
    }

    /* listGET_OWNER_OF_NEXT_ENTRY indexes through the list, so the tasks of
the same priority get an equal share of the processor time. */
    listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &(amp; pxReadyTasksLists[ uxTopPriority ] ) );
    uxTopReadyPriority = uxTopPriority;
} /* taskSELECT_HIGHEST_PRIORITY_TASK */
```

taskSELECT\_HIGHEST\_PRIORITY\_TASK 就是通用的选择算法的具体函数了，它会先对 uxTopReadyPriority 的值进行保存。while 循环里的 configASSERT 是断言，

用来 debug，如果 uxTopPriority 小于 0，会触发它。

每一个优先级都有一个特定的链表，while 判断就绪链表的这个优先级是否有任务挂载在上面，如果没有，就继续找下一个优先级的链表，也就是一 uxTopPriority。

uxTopPriority 被设定为是代表最大优先级的数字，自减操作代表从就绪列表最后一个链表（优先级最高的链表）开始查找，直到找到任务链表不为空的优先级任务，那么这个任务肯定也是所有任务中优先级最大的任务，

然后获取这个任务的 TCB 并更新 pxCurrentTCB（切换的具体函数），最后更新 uxTopReadyPriority 的值。

## rt-thread 内核的调度算法

rt-thread 是一个偏 linux 的 RTOS，而且用面向对象的思想开发的，非常有特色。

笔者决定再讲一讲 rtt 的选择算法，因为这种算法很常见，用途也非常广泛。

```
/**
 * 该函数用于从一个32位的数中寻找第一个被置1的位（从低位开始），
 * 然后返回该位的索引（即位号）
 *
 * @return 返回第一个置1位的索引号。如果全为0，则返回0。
 */
int __rt_ffs(int value)
{
    /* 如果值为0，则直接返回0 */
    if (value == 0) return 0;
    /* 检查 bits [07:00]
     * 这里加1的原因是避免当第一个置1的位是位0时
     * 返回的索引号与值都为0时返回的索引号重复 */
    if (value & 0xff)
        return __lowest_bit_bitmap[value & 0xff] + 1;
    /* 检查 bits [15:08] */
    if (value & 0xff00)
        return __lowest_bit_bitmap[(value & 0xff00) >> 8] + 9;
    /* 检查 bits [23:16] */
    if (value & 0xff0000)
        return __lowest_bit_bitmap[(value & 0xff0000) >> 16] + 17;
    /* 检查 bits [31:24] */
    return __lowest_bit_bitmap[(value & 0xff000000) >> 24] + 25;
}
```

该算法的精髓在于这个表：

```

/*
 * __lowest_bit_bitmap[] 数组的解析
 * 将一个8位整形数的取值范围0~255作为数组的索引，索引值第一个出现1(从最低位开始)的位号作为该数组索引下的成员值。
 * 举例：十进制数10的二进制为：0000 1010,从最低位开始，第一个出现1的位号为bit1, 则有__lowest_bit_bitmap[10]=1
 * 注意：只需要找到第一个出现1的位号即可
 */
const rt_uint8_t __lowest_bit_bitmap[] =
{
    /* 00 */ 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 10 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 20 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 30 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 40 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 50 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 60 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 70 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 80 */ 7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* 90 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* A0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* B0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* C0 */ 6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* D0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* E0 */ 5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    /* F0 */ 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
}

```

RTT 使用的是查表法，让笔者简单介绍一下这种策略的思想。

我们要查找一个链表中优先级最高的任务，最简便的做法就是 for 循环，一个个遍历比较，找到优先级最高的任务，这种做法很简便，缺点是复杂度过高，达到了  $O(n)$ ，查询的时间很长。那么有没有使复杂度为 1 的方法呢？

于是 RTT 使用了空间换时间的策略。

假设我们有八个优先级，用八个位来表示它们。任务优先级为多少，那么就在哪个位置 1。

例如，task1 优先级为 5，那么优先级的 Table 数字就是 00010000。

在实时操作系统中，优先级数字越小，优先级越高，那么这个任务代表的 1 更靠近右侧。

既然知道了这些，那么，把所有 8 个位的组合都列出来，判断每一个不同的数字最低位的 1 在哪个位，然后直接查询，不就可以知道任务的优先级最高是多少吗？然后顺着优先级，去执行对应的任务。



```

void main()
{
    int prioritynum;
    int i;
    int a = 0;
    printf("0  ,");
    for (prioritynum = 0; prioritynum < 256; prioritynum++) {
        a = a + 1;
        for (i = 0; i < 8; i++)
        {
            if ((prioritynum >>i) & (0x01))
            {
                printf("%d  ", i);
                break;
            }
        }
        if (a == 16)
        {
            a = 0;
            printf("\n");
        }
    }
}

```

通过这个函数，可以将 prioritynum 最低位的 1 的位置打印出来。这个函数比较简单，就是循环遍历 0 到 255，也就是 00000000 到 11111111，让每一个数依次右移最多八位，找到最低位的 1。prioritynum 右移后，会与 1 进行与运算，也就是说，除非右移后的数最低位是 1，否则都会进行下一轮循环，直到找到最低位的 1 并且打印；随后程序又会跳出内层循环进行下一个 prioritynum 的运算。运行结果：

```

0 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
5 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
6 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
5 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
7 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
5 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
6 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
5 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,
4 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,3 ,0 ,1 ,0 ,2 ,0 ,1 ,0 ,

```

通过这个数组，我们可以将 prioritynum 作为数组下标查询最高优先级。这其实就是一种哈希算法的思想，用特定的数字映射特定的优先级。这种算法缺点也十分明显，就是数组占用空间太大了，一个 int 就要 4 个字节，上面的例子是优先级为 8 位，如果优先级有 32 位呢？那么占用空间将会达到：

4\*2 的 32 次方个字节。

```
if (value == 0) return 0;
/* 检查 bits [07:00]
这里加1的原因是避免当第一个置1的位是位0时
返回的索引号与值都为0时返回的索引号重复 */
if (value & 0xff)
    return __lowest_bit_bitmap[value & 0xff] + 1;
/* 检查 bits [15:08] */
if (value & 0xff00)
    return __lowest_bit_bitmap[(value & 0xff00) >> 8] + 9;
/* 检查 bits [23:16] */
if (value & 0xff0000)
    return __lowest_bit_bitmap[(value & 0xff0000) >> 16] + 17;
/* 检查 bits [31:24] */
return __lowest_bit_bitmap[(value & 0xff000000) >> 24] + 25;
```

为了解决这个问题，RTT 内核进行了四次判断，每次判断 8 个位，算法也很简洁，先读最低八位的值，判断是否为 0，不为 0 说明最大优先级肯定在最低八位中，然后直接查表即可。可能有读者好奇为什么要加上 1，其实观察就会发现，这个表的第一个 0，是在循环之前手动打印的，否则没有 256 个数，为了避免与 32 位全为 0 冲突，所以加上了 1。

简单来说，笔者手动让数组里的所有数向后移动了一位，那么相应的，把 prioritynum 作为下标读取表里的数，也要往后面移动一位。

如果最低八位没有，那就到次八位找，同样先判断是否在次八位中。如果在，那么右移八位然后查表，因为进行了右移操作，所以返回值要在最低八位的返回值的基础加上 8。

剩下的代码同理，留给读者自己分析了。

总的来说，这种基于哈希思想的查表法应用十分广泛，采用空间换时间的策略，具有一定的学习价值。如果某些程序对查询的时间要求比较高，可以尝试用查表策略代替之前的遍历策略。

## 总结

笔者介绍了 linux 内核、FreeRTOS、RT-Thread 的调度策略和算法，旨在加深读者对操作系统的理解。

## 实现调度算法

# 前言

在上一篇博客笔者介绍了操作系统中的调度算法，调度算法的本质就是选择下一个任务。

如果读者认真看了上一篇文章的内容,那么接下来讲解的 Sparrow 的算法应该是非常清晰易懂的。

在实时操作系统 Sparrow RTOS 中，我们引入了优先级的概念，就是为了使任务的运行更加具有实时性，优先级由我们手动进行调整，往往我们希望优先级高的任务优先执行，那么，该怎么做呢？

## 调度算法的设计

我们使用 ReadyBitTable 这个 uint32\_T 类型的变量来标识就绪的任务，只要任务是就绪态的，我们就执行 ReadyBitTable |= (1 << uxPriority) 操作，这样 ReadyBitTable 中为 1 的位就表示有就绪的任务。

那么如何得到对应的优先级数字呢？这里我们可以计算从高位到低位，离最近的1有多少个0，然后用31减去这个数目，就可以得到优先级的数字了。

在 arm 架构中，恰好有这么一条汇编指令 `clz`，它可以计算从高位到低位，离最近的 1 之间的 0 的数目，正好符合我们的需求。

举例如下:

[illegible]

不知道读者发现没有，其实这种方法跟上一节中所讲的 RTThread 的算法思想是一样的，只是我们使用 `clz` 指令代替了查表法。在 FreeRTOS 和 RT-Thread 等 RTOS 中，其实都有使用 `clz` 指令的方法，它们跟 Sparrow 的算法都大差不差。

## 修改程序

既然我们已经知道了思路，那么让我们开始写代码：



uint32\_t ReadyBitTable = 0;//添加到合适的地方即可

```
__attribute__( ( always_inline ) ) static inline uint8_t
FindHighestPriority( void )
{
    uint8_t TopZeroNumber;
    uint8_t temp;
    __asm volatile
    (
        "clz %0, %2\n"
        "mov %1, #31\n"
        "sub %0, %1, %0\n"
        : "=r" (TopZeroNumber), "=r" (temp)
        : "r" (ReadyBitTable)
        );
    return TopZeroNumber;
}

void vTaskSwitchContext( void )
{
    pxCurrentTCB = TcbTaskTable[ FindHighestPriority()];
}
```

修改 xTaskCreat 函数，添加一行代码：

```
void xTaskCreate( TaskFunction_t pxTaskCode,
                 const uint16_t usStackDepth,
                 void * const pvParameters,//You can use it for
debugging
                 uint32_t uxPriority,
                 TaskHandle_t * const self )
{
    uint32_t *topStack =NULL;
    TCB_t *NewTcb = (TCB_t *)heap_malloc(sizeof(TCB_t *));
    *self = ( TCB_t *) NewTcb;

    TcbTaskTable[uxPriority] = NewTcb;//

    NewTcb->uxPriority = uxPriority;
```

```

    NewTcb->pxStack = ( uint32_t *) heap_malloc( ( ( ( size_t )
usStackDepth ) * sizeof( uint32_t * ) ) );
    topStack = NewTcb->pxStack + (usStackDepth - (uint32_t)1) ;
    topStack = ( uint32_t *) (((uint32_t)topStack) & (~((uint32_t)
alignment_byte)));
    NewTcb->pxTopOfStack =
pxPortInitialiseStack(topStack, pxTaskCode, pvParameters, self);
    pxCurrentTCB = NewTcb;

    ReadyBitTable |= (1 << uxPriority); //添加这一行
}

```

修改空闲任务，任务内容为进入低功耗模式：

```

void EnterSleepMode(void)
{
    SCB->SCR &= ~SCB_SCR_SLEEPDEEP_Msk;
    __WFI();
}

```

//Task handle can be hide, but in order to debug, it must be created manually by the user  
TaskHandle\_t leisureTcb = NULL;

```

void leisureTask( )
{
    //leisureTask content can be manually modified as needed
    while (1) {
        EnterSleepMode();
    }
}

```

## 实验

验证思路

设置一个任务的优先级为最大，观察这个任务是不是一直在执行。

程序

修改任务：

//Task Area!The user must create task handle manually because of debugging and specification

```
TaskHandle_t tcbTask1 = NULL;
TaskHandle_t tcbTask2 = NULL;
```

```
void led_bright( )
{
    while (1) {

    }
}
```

```
void led_extinguish( )
{
    while (1) {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
        HAL_Delay(500);
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
        HAL_Delay(500);
        switchTask();
    }
}
```

当然，请确保 led\_extinguish 优先级足够大：

```
void APP( )
{

    xTaskCreate(    led_bright,
                   128,
                   NULL,
                   1,
                   &tcbTask1
    );

    xTaskCreate(    led_extinguish,
                   128,
                   NULL,
                   8,
                   &tcbTask2
    );
}
```

## 实验现象：

stm32f103c8t6 上的 led 灯一直在闪烁,说明任务 led\_extinguish 一直在执行。

## 总结

介绍了 Sparrow 的调度算法，然后编写程序实现了优先级抢占算法的设计，最后修改原先的工程对调度算法进行验证。

## 问题

能不能使用链表设计调度算法？从而取代现在的数表式算法，那么同理，能不能使用二叉树或其他数据结构设计调度算法？

# 时钟

## 前言

前面已经说过了，Sparrow 采用的是时间触发系统的设计，本章将会讲解时钟触发相关的算法。

重新回顾一下时间触发系统的定义：

时间触发系统的任务调度基于定时器中断，适用于周期性任务和确定性要求高的场景，如控制系统。它的内部有一个时钟，每隔一定时间间隔就会触发一次时间中断，每次时钟中断时，调度器决定是否需要切换任务。

## systick 时钟

systick 时钟是为 RTOS 而设计的一个中断时钟，它使得 RTOS 在不同架构之间的移植性难度大大减小。

### 13.1 SysTick 定时器

回顾第 8 章讲述 NVIC 时，曾走马观花地带过了 SysTick 定时器。复习一下：SysTick 是一个 24 位的倒数计数定时器，当计到 0 时，将从 RELOAD 寄存器中自动重载定时初值。只要不把它在 SysTick 控制及状态寄存器中的使能位消除，就永不停息。图 13.1 中小结了 SysTick 的相关寄存器。

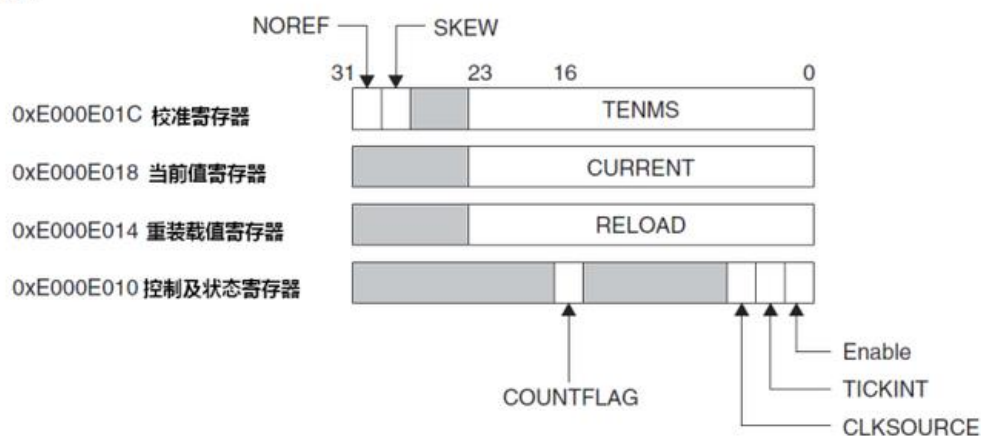


图 13.1 SysTick 相关寄存器的定义

在前面的文章中，笔者曾经调试过 Sparrow 来了解 RTOS 内部的中断使用，当时得出的结论是：SysTick 中断会在一定时间间隔响应，然后触发 PendSV 中断产生上下文切换：

就绪的任务，上下文切换被触发的场合可以是：

- 执行一个系统调用
- 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。如图 7.15 所示。

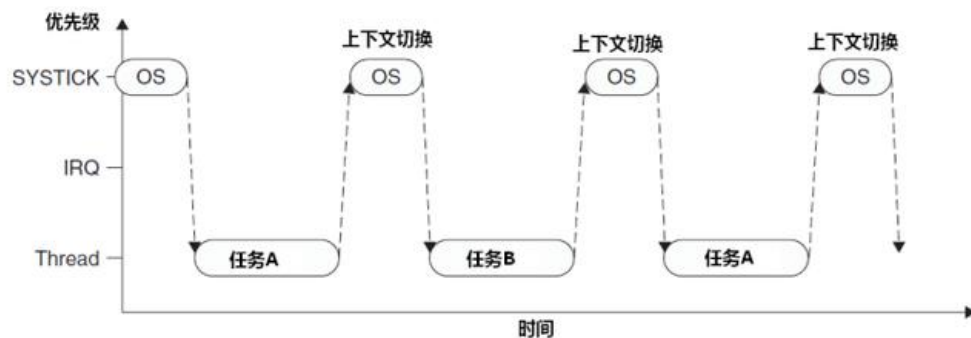


图 7.15 两个任务间通过 **SysTick** 进行轮转调度的简单模式

systick 中断用法如下：

```

// SYSTICK 初始化
void SysTickInit(void)
{
#define NVIC_STCSR ((volatile unsigned long *) (0xE000E010))
#define NVIC_RELOAD ((volatile unsigned long *) (0xE000E014))
#define NVIC_CURRVAL ((volatile unsigned long *) (0xE000E018))
#define NVIC_CALVAL ((volatile unsigned long *) (0xE000E01C))

    *NVIC_STCSR = 0; // 除能 SYSTICK
    *NVIC_RELOAD = 499999; // 基于50MHz主频的100Hz装载值
    *NVIC_CURRVAL = 0; // 清除当前值
    *NVIC_STCSR = 0x7; // 使能SYSTICK, 使能中断, 使用内核时钟
    return;
}

// SYSTICK 异常服务例程
void SysTickHandler(void)
{
    if (CurrState==RUN_STATE) {
        TickCounter++;
    }
    return;
}

```

## 如何使用时钟

### 定时上下文切换

参考上面的程序，我们可以先设置 SysTick 时钟，让它在一定时间间隔响应，然后在 SysTick 中断中加入 PendSV 中断的触发函数，这样就能达到定时上下文切换的目的。

除此之外，SysTick 本身就是一个定时器，RTOS 中需要定时的地方太多了，我们必须高效利用它的定时功能。

## 延时阻塞态

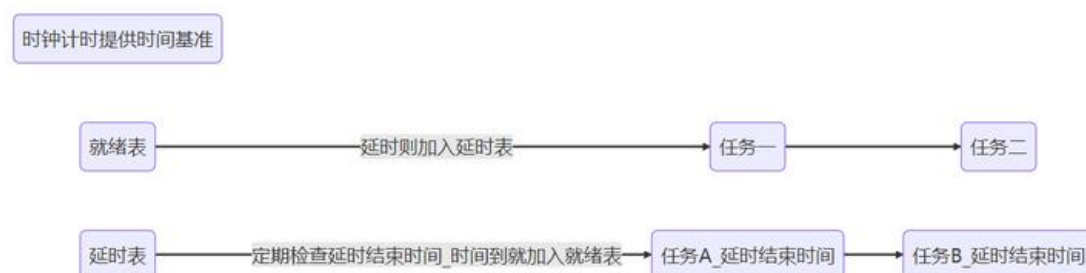
### 利用延时的时间

在 Sparrow 中，任务有就绪态、阻塞态、挂起态三种状态，任何任务只能有一种状态。延时就是一种阻塞态，表示它在等待某个事情的发生。

让我们想一想，在非 RTOS 的环境中，我们的延时一般都是空等，但是在 RTOS 环境中，任务被分解为线程的形式，每个线程尽量与其他线程并行执行，那么，我们是不是可以在其他线程延时等待时，把它移除就绪表，然后把时间让给其他线程执行呢？等时间一到，再把它加入就绪表。

这当然是可行的。

## 如何设计算法？



如图所示，我们需要一张延时表，当每个任务延时时，这个任务会被踢出就绪表，延时表会记录它延时到期的时间，每一次 SysTick 中断时，时间基数就会加 1，然后我们可以先比较延时表中的任务是否到时间，如果到时间，那么就从延时表删除，加入就绪表，如果没到，就继续等。

## 如何解决溢出的问题？

我们知道计算机中一个数的大小是有上限的，比如 `uint_32`，最大只能到  $2^{32} - 1$ 。如果发生了溢出那怎么办呢？

笔者提供两种思路：

1. 设置更大的计数单位



2. 使用两个表进行维护。

当然，Sparrow 使用的是后一种。

设置更大的计数单位

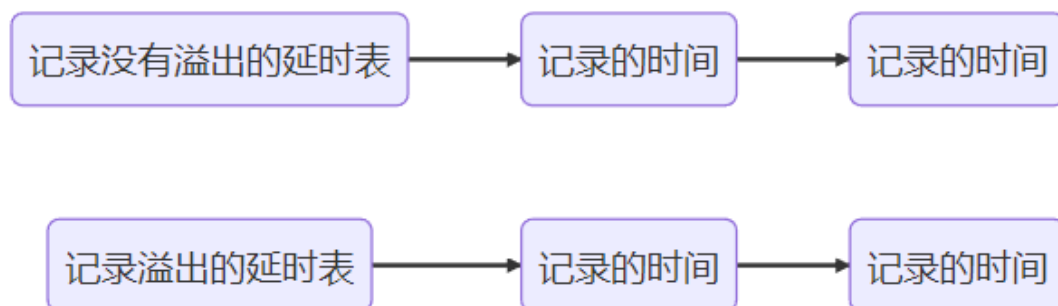
请读者想一想，我们每天都是 24 小时轮转，但我们是怎么表示时间的呢？是通过年、月、日来进行记录，也就是说，SysTick 的计数也可以这样干，每溢出一一次，前面的计数单位就加一，这样就能记录事情发生的先后了。

但是假设时间非常漫长，那么单片机会不断的设置更大的计数单位，从秒、分、时、日、月，一直到年，每一个数字都需要内存来存储，这样的方法，缺点是内存并不固定。

当然，这是考虑时间永久的情况下，实际生活中，1s 计数 1000 次，约等于  $2^{10}$ ，一分钟约等于计数  $2^{16}$ ，一小时约等于  $2^{22}$ ，三天约等于  $2^{28}$ ，48 天约等于  $2^{32}$ 。既然溢出一一次需要 48 天，那么只要我们使用两个计数单位，那么上限不就是  $48 \times 2^{32}$  天了？理论上就可以计数到产品损坏了。

使用两个表进行维护

正常情况下，任务的延时加上当前时间最多溢出一一次，例如  $1 + 2^{32} - 1$  会溢出一一次，虽然  $1 + 2^{32} - 1 + 2^{32}$  会溢出一两次，但是这个时间段太长了，太离谱了，根本不可能被使用。既然最多溢出一一次，那么，只要多一个表就可以解决溢出的问题。



当时间溢出发生时，由于此时记录没有溢出的延时表的任务肯定已经完成了延时，也就是说，这张表现在是空的。

此时只有溢出的延时表上的延时需要完成，当溢出再次发生时，我们又需要一张表，难道我们要重新申请一块内存来存储表吗？

显然，完全不需要，因为没有溢出的延时表是空的，我们可以重复利用它。现在，之前溢出的延时表变成了未溢出的延时表，之前未溢出的延时表被重新利用，拿来记录现在溢出的延时任务！

使用两个表进行维护，这就是 Sparrow 的延时策略。

## 总结

先讲解了时间触发型 RTOS 的设计，然后讲解 SysTick 时钟，它通常作为 RTOS 的定时器。为了充分利用定时器，引入延时阻塞的概念，解决了裸机中常见的延时空等待的问题。

## 阻塞延时的实现

在上一篇文章中，笔者介绍了实现时钟触发型 RTOS 的实现，介绍了 Sparrow 的时钟及延时阻塞的实现，现在让我们编写具体的代码。

### 配置 SysTick 时钟

先把宏添加：

```
#define configSysTickClockHz          ( ( unsigned long ) 72000000 )
```

```
#define configTickRateHz          ( ( uint32_t ) 1000 )
```

修改 SchedulerStart 函数，配置 SysTick 中断，装载对应的值，确保 1ms 发生一次中断。

参考 cm3 手册装载对应的值：

表8.9 SysTick控制及状态寄存器 ( 地址：0xE000\_E010 )

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经计到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数计数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

表8.10 SysTick重装载数值寄存器 ( 地址：0xE000\_E014 )

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数计数至零时，将被重装载的值

表8.11 SysTick当前数值寄存器 ( 地址：0xE000\_E018 )

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

表8.10 SysTick校准数值寄存器 ( 地址：0xE000\_E01C )

位段	名称	类型	复位值	描述
31	NOREF	R	-	1=没有外部参考时钟 (STCLK 不可用) 0=外部参考时钟可用
30	SKEW	R	-	1=校准值不是准确的 10ms 0=校准值是准确的 10ms
23:0	TENMS	R/W	0	在 10ms 的间隔中倒计数的格数。芯片设计者应

代码配置如下：

```
__attribute__(( always_inline )) inline void SchedulerStart( void )
{
    /* Start the timer that generates the tick ISR.  Interrupts are
    disabled
    * here already. */
    ( *( ( volatile uint32_t * ) 0xe000ed20 ) ) |= ( ( ( uint32_t )
255UL ) << 16UL );
    /*加入以下代码*/
    ( *( ( volatile uint32_t * ) 0xe000ed20 ) ) |= ( ( ( uint32_t )
```

```

255UL ) << 24UL );

/* Stop and clear the SysTick. */
SysTick->CTRL = 0UL;
SysTick->VAL = 0UL;
/* Configure SysTick to interrupt at the requested rate. */
SysTick->LOAD = ( configSysTickClockHz / configTickRateHz ) - 1UL;
SysTick->CTRL = ( ( 1UL << 2UL ) | ( 1UL << 1UL ) | ( 1UL <<
0UL ) );
/*加入以上代码*/

/* Start the first task. */
__asm volatile (
    " ldr r0, =0xE000ED08    \n"/* Use the NVIC offset register
to locate the stack. */
    " ldr r0, [r0]          \n"
    " ldr r0, [r0]          \n"
    " msr msp, r0           \n"/* Set the msp back to the
start of the stack. */
    " cpsie i               \n"/* Globally enable interrupts.
*/
    " cpsie f               \n"
    " dsb                   \n"
    " isb                   \n"
    " svc 0                 \n"/* System call to start first
task. */
    " nop                   \n"
    " .ltorg                \n"
    );
}

```

## 添加延时表

当然，不止延时表，程序中也有 SuspendBitTable 这些，这可以被使用者用来记录挂起的任务。

这样，Sparrow 的线程就有四种状态了：运行态、就绪态、阻塞态、挂起态。具体的状态转换函数就由感兴趣的读者自己去完成了。

### 简单回顾一下延时的算法：

使用两个数组，一个记录溢出的时钟值，一个记录未溢出的延时值。然后使用两

个指针，分别指向这两个表，当溢出发生时，两个指针交换地址，这样就完成了溢出表和未溢出表的交换。

### 介绍一下各个函数：

TicksTableInit:初始化两个延时表数组

TicksTableSwitch: 交换两个指针

TaskDelay: 先判断计数是否发生溢出，然后把值存储到不同的表中，延时表对应的位置 1，表示这个优先级对应的任务在延时。就绪表对应的位置 0，表示这个优先级对应的位没有任务需要执行，最后触发上下文切换。

CheckTicks: 每发生一次 SysTick 中断，先判断是否发生了溢出代表当前未溢出的延时数组的任务都已经延时完成，就进行切换。然后遍历未溢出的延时数组的值，判断是否有任务延时结束，如果是，那么就移除延时表，加入就绪表，最后触发上下文切换。

```
uint32_t  NextTicks = ~(uint32_t)0;
uint32_t  TicksBase = 0;
```

```
uint32_t DelayBitTable = 0;
uint32_t TicksTable[configMaxPriori];
uint32_t TicksTableAssist[configMaxPriori];
uint32_t* WakeTicksTable;
uint32_t* OverWakeTicksTable;
#define TicksTableInit( ) { \
    WakeTicksTable = TicksTable; \
    OverWakeTicksTable = TicksTableAssist; \
}
#define TicksTableSwitch( ){ \
    uint32_t *temp; \
    temp = WakeTicksTable; \
    WakeTicksTable = OverWakeTicksTable; \
    OverWakeTicksTable = temp; \
}
```

```
uint32_t SuspendBitTable = 0;
```

```
//the table is defined for signal mechanism
uint32_t BlockedBitTable = 0;
```

```
/*The RTOS delay will switch the task.It is used to liberate low-
priority task*/
void TaskDelay( uint16_t ticks )
{
```

```

uint32_t WakeTime = TicksBase + ticks;
TCB_t *self = pxCurrentTCB;
if( WakeTime < TicksBase)
{
    OverWakeTicksTable[self->uxPriority] = WakeTime;
}
else
{
    WakeTicksTable[self->uxPriority] = WakeTime;
}
/* This is a useless operation(for DelayBitTable), it can be
discarded.
    * But it is retained for the sake of normativity.For example,
view the status of all current tasks.*/
DelayBitTable |= (1 << (self->uxPriority) );
ReadyBitTable &= ~(1 << (self->uxPriority) );
switchTask();
}

```

```

void CheckTicks( void )
{
    TicksBase += 1;
    if( TicksBase == 0){
        TicksTableSwitch( );
    }
    for(int i=0 ; i < configMaxPriori;i++)
    {
        if( WakeTicksTable[i] > 0)
        {
            if ( TicksBase >= WakeTicksTable[i] )
            {
                WakeTicksTable[i] = 0;
                DelayBitTable &= ~(1 << i );//it is retained for the
sake of normativity.
                ReadyBitTable |= (1 << i);
            }
        }
    }
    switchTask();
}

```

**调度器初始化如下：**

```

void SchedulerInit( void )
{

```

```

    TicksTableInit();
    xTaskCreate(    leisureTask,
                  128,
                  NULL,
                  0,
                  &leisureTcb
    );
}

```

添加 systick 中断，为了防止被打断，加入临界区：

```

void SysTick_Handler(void)
{
    uint32_t xre = xEnterCritical();
    CheckTicks();
    xExitCritical(xre);
}

```

## 实验

验证思路：使用 Sparrow 的阻塞延时代替原先的延时，观察开发板的运行情况是否正常。

### 修改任务：

对于优先级和延时的设置是有要求的，因为灯亮和灯灭是互斥的。优先级设置不当，可能导致前脚执行灯亮，后脚就执行灯灭了，这样下去，虽然线程是并行执行，但是灯始终是灭的状态。

//Task Area!The user must create task handle manually because of debugging and specification

```

TaskHandle_t tcbTask1 = NULL;
TaskHandle_t tcbTask2 = NULL;

```

```

void led_bright( )
{
    while (1) {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);
    }
}

```

```

        TaskDelay(1000);
    }
}

void led_extinguish( )
{
    while (1) {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
        TaskDelay(500);
    }
}

void APP( )
{
    xTaskCreate(    led_bright,
                  128,
                  NULL,
                  1,
                  &tcbTask1
    );

    xTaskCreate(    led_extinguish,
                  128,
                  NULL,
                  2,
                  &tcbTask2
    );
}

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    SchedulerInit();
    APP();
}

```



```
SchedulerStart();  
  
while (1)  
{  
  
}  
}
```

## 实验现象

编译程序下载到开发板中，实验现象为 led 灯闪烁。

## 总结

编写了时钟触发和阻塞延时的具体代码，彻底完成了 Sparrow 的代码。经历二十天的编写，Sparrow 的教程也将迎来尾声。

下一篇博客，也就是第二十篇，笔者不得不为 Sparrow RTOS 的教程画上一个圆满的句号了，是时候让我们的这趟旅程落下帷幕了。

## 问题

能不能使用设置更大的计数单位设计时钟阻塞延时？如何设计程序？

## 本文结语

本教程到这里就完结了，祝大家学得开心！

