---

title: FreeRTOS优化与错误排查方法 author: 杰杰 top: false cover: false toc: true mathjax: false date: 2019-10-05 09:30:58 img: coverImg: password: summary: tags:

- FreeRTOS
- RTOS
- 操作系统 categories: 操作系统

写在前面

主要是为刚接触 FreeRTOS 的用户指出那些新手通常容易遇到的问题。这里把最主要的篇幅放在栈溢出以及栈溢出j检测上，因为栈相关的问题是初学者遇到最多的问题。

printf-stdarg.c

当调用 **C 标准库** 的函数时，栈空间使用量可能会急剧上升，特别是 IO 与字符串处理函数，比如 sprintf()、printf()等。在 FreeRTOS 源码包中有一个名为 **printf-stdarg.c** 的文件。这个文件实现了一个栈效率优化版的小型 sprintf()、printf()，可以用来代替标准 C 库函数版本。在大多数情况下，这样做可以使得调用 sprintf()及相关函数的任务对栈空间的需求量小很多。 可能很多人都不知道freertos中有这样子的一个文件，它放在第三方资料中，路径为“**FreeRTOSv9.0.0\FreeRTOS-Plus\Demo\FreeRTOS_Plus_UDP_and_CLI_LPC1830_GCC**”，我们发布工程的时候就无需依赖 **C 标准库**，这样子就能减少栈的使用，能优化不少空间。 该文件源码（部分）：

```
static int print( char **out, const char *format, va_list args )
{
        register int width, pad;
        register int pc = 0;
        char scr[2];

        for (; *format != 0; ++format) {
                if (*format == '%') {
                        ++format;
                        width = pad = 0;
                        if (*format == '\0') break;
                        if (*format == '%') goto out;
                        if (*format == '-') {
                                ++format;
                                pad = PAD_RIGHT;
                        }
                        while (*format == '0') {
                                ++format;
                                pad |= PAD_ZERO;
                        }
                        for ( ; *format >= '0' && *format <= '9'; ++format) {
                                width *= 10;
                                width += *format - '0';
                        }
                        if( *format == 's' ) {
                                register char *s = (char *)va_arg( args, int );
                                pc += prints (out, s?s:"(null)", width, pad);
                                continue;
```

```
                        }
                        if( *format == 'd' || *format == 'i' ) {
                                pc += printi (out, va_arg( args, int ), 10, 1,
width, pad, 'a');
                                continue;
                        }
                        if( *format == 'x' ) {
                                pc += printi (out, va_arg( args, int ), 16, 0,
width, pad, 'a');
                                continue;
                        }
                        if( *format == 'X' ) {
                                pc += printi (out, va_arg( args, int ), 16, 0,
width, pad, 'A');
                                continue;
                        }
                        if( *format == 'u' ) {
                                pc += printi (out, va_arg( args, int ), 10, 0,
width, pad, 'a');
                                continue;
                        }
                        if( *format == 'c' ) {
                                /* char are converted to int then pushed on the
stack */
                                scr[0] = (char)va_arg( args, int );
                                scr[1] = '\0';
                                pc += prints (out, scr, width, pad);
                                continue;
                        }
                }
                else {
                out:
                        printchar (out, *format);
                        ++pc;
                }
        }
        if (out) **out = '\0';
        va_end( args );
        return pc;
}

int printf(const char *format, ...)
{
        va_list args;

        va_start( args, format );
        return print( 0, format, args );
}

int sprintf(char *out, const char *format, ...)
{
        va_list args;

        va_start( args, format );
```

```
        return print( &out, format, args );
}


int snprintf( char *buf, unsigned int count, const char *format, ... )
{
        va_list args;

        ( void ) count;

        va_start( args, format );
        return print( &buf, format, args );
}
```

使用的例子与 C 标准库基本一样：

```
int main(void)
{
        char *ptr = "Hello world!";
        char *np = 0;
        int i = 5;
        unsigned int bs = sizeof(int)*8;
        int mi;
        char buf[80];

        mi = (1 << (bs-1)) + 1;
        printf("%s\n", ptr);
        printf("printf test\n");
        printf("%s is null pointer\n", np);
        printf("%d = 5\n", i);
        printf("%d = - max int\n", mi);
        printf("char %c = 'a'\n", 'a');
        printf("hex %x = ff\n", 0xff);
        printf("hex %02x = 00\n", 0);
        printf("signed %d = unsigned %u = hex %x\n", -3, -3, -3);
        printf("%d %s(s)%", 0, "message");
        printf("\n");
        printf("%d %s(s) with %%\n", 0, "message");
        sprintf(buf, "justif: \"%-10s\"\n", "left"); printf("%s", buf);
        sprintf(buf, "justif: \"%10s\"\n", "right"); printf("%s", buf);
        sprintf(buf, " 3: %04d zero padded\n", 3); printf("%s", buf);
        sprintf(buf, " 3: %-4d left justif.\n", 3); printf("%s", buf);
        sprintf(buf, " 3: %4d right justif.\n", 3); printf("%s", buf);
        sprintf(buf, "-3: %04d zero padded\n", -3); printf("%s", buf);
        sprintf(buf, "-3: %-4d left justif.\n", -3); printf("%s", buf);
        sprintf(buf, "-3: %4d right justif.\n", -3); printf("%s", buf);

        return 0;
}
```

栈计算

每个任务都独立维护自己的栈空间， 任务栈空间总量在任务创建时进行设定。
`uxTaskGetStackHighWaterMark()`主要用来查询指定任务的运行历史中， 其栈空间还差多少就要溢出。这个值被称为栈空间的**High Water Mark**。 函数原型：

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask )
```

想要使用它，需要将对应的宏定义打开：`INCLUDE_uxTaskGetStackHighWaterMark`

函数描述：

参数	说明
xTask	被查询任务的句柄如果传入 NULL 句柄，则任务查询的是自身栈空间的高水线
返回值	任务栈空间的实际使用量会随着任务执行和中断处理过程上下浮动。uxTaskGetStackHighWaterMark()返回从任务启动执行开始的运行历史中，栈空间具有的最小剩余量。这个值即是栈空间使用达到最深时的剩下的未使用的栈空间。这个值越是接近 0，则这个任务就越是离栈溢出不远。

如果不知道怎么计算任务栈大小，就使用这个函数进行统计一下，然后将任务运行时最大的栈空间作为任务栈空间的80%大小即可。即假设统计得到的任务栈大小为常量 **A** ，那么在创建线程的时候需要 **X** 大小的空间，那么 **X * 80% = A**，算到的 **X** 作为任务栈大小就差不多了。

运行时栈检测

FreeRTOS 包含两种运行时栈j检测机制，由 FreeRTOSConfig.h 中的配置常量 configCHECK_FOR_STACK_OVERFLOW 进行控制。这两种方式都会增加上下切换开销。

栈溢出钩子函数(或称回调函数)由内核在j检测到栈溢出时调用。要使用栈溢出钩子函数，需要进行以下配置：

- 在 FreeRTOSConfig.h 中把 configCHECK_FOR_STACK_OVERFLOW 设为 **1** 或者 **2** 。
- 提供钩子函数的具体实现，采用下面所示的函数名和函数原型。

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR
*pcTaskName );
```

补充说明：

- 栈溢出钩子函数只是为了使跟踪调试栈空间错误更容易，而无法在栈溢出时对其进行恢复。函数的入口参数传入了任务句柄和任务名，但任务名很可能在溢出时已经遭到破坏。
- 栈溢出钩子函数还可以在中断的上下文中进行调用
- 某些微控制器在检测到内存访问错误时会产生错误异常，很可能在内核调用栈溢出钩子函数之前就触发了错误异常中断。

方法1

当 configCHECK_FOR_STACK_OVERFLOW 设置为 **1** 时选用方法 **1**。 任务被交换出去的时候，该任务的整个上下文被保存到它自己的栈空间中。这时任务栈的使用应当达到了一个峰值。当 configCHECK_FOR_STACK_OVERFLOW 设为**1** 时，内核会在任务上下文保存后检查栈指针是否还指向有效栈空间。一旦检测到栈指针的指向已经超出任务栈的有效范围，栈溢出钩子函数就会被调用。 方法 1 具有较快的执行速度，但栈溢出有可能发生在两次上下文保存之间，这种情况不会被检测到，因为这种检测方式仅在任务切换中检测。

方法2

将 configCHECK_FOR_STACK_OVERFLOW 设为 **2** 就可以选用方法 **2**。方法 **2**在方法 1 的基础上进行了一些补充。 当创建任务时，任务栈空间中就预置了一个标记。方法 **2** 会检查任务栈的最后 **20个字节**的数据，查看预置在这里的标记数据是否被覆盖。如果最后 **20 个字节的标记数据**与预设值不同，则栈溢出钩子函数就会被调用。 方法 **2** 没有方法 **1** 的执行速度快，但测试仅仅 20 个字节相对来说也是很快的。这种方法应该可以j检测到任何时候发生的栈溢出，虽然理论上还是有可能漏掉一些情况，但这些情况几乎是不可能发生的。

其它常见错误

> 在一个 Demo 应用程序中增加了一个简单的任务，导致应用程序崩溃

可能的情况：

1. 任务创建时需要在内存堆中分配空间。许多 Demo 应用程序定义的**堆空间**大小只够用于创建 Demo 任务——所以当任务创建完成后，就没有足够的剩余空间来增加其它的**任务，队列或信号量**。
2. 空闲任务是在 vTaskStartScheduler()调用中自动创建的。如果由于内存不足而无法创建空闲任务，vTaskStartScheduler()会直接返回。所以一般在调用 vTaskStartScheduler()后加上一条空循环 for(;;) / while(1)可以使这种错误更加容易调试。 **如果要添加更多的任务，可以增加内存堆空间大小（修改配置文件），或是删掉一些已存在的 Demo任务。**

> 在中断中调用一个 API 函数，导致应用程序崩溃

需要做的第一件事是检查中断是否导致了栈溢出。

然后检查API接口是否正确，除了具有后缀为FromISR函数名的 API 函数，千万不要在中断服务程序中调用其它 API 函数。

除此之外，还需要注意中断的优先级： FreeRTOSConfig.h文件中可以配置系统可管理的最高中断优先级数值，宏定义configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY是用于配置basepri寄存器的，当basepri设置为某个值的时候，会让系统不响应比该优先级低的中断，而优先级比之更高的中断则不受影响。就是说当这个宏定义配置为**5**的时候，中断优先级数值在**0**、**1**、**2**、**3**、**4**的这些中断是不受FreeRTOS管理的，不可被屏蔽，**同时也不能调用FreeRTOS中的API函数接口**，而中断优先级在5到15的这些中断是受到系统管理，可以被屏蔽的，也可以调用FreeRTOS中的API函数接口。

> 临界区无法正确嵌套

除了 taskENTER_CRITICA()和 taskEXIT_CRITICAL()，千万不要在其它地方修改控制器的中断使能位或优先级标志。这两个宏维护了一个嵌套深度计数，所以只有当所有的嵌套调用都退出后计数值才会为 0，也才会使能中断。

> 在调度器启动前应用程序就崩溃了

这个问题我也会遇到，如果一个中断会产生上下文切换，则这个中断不能在调度器启动之前使能。这同样适用于那些需要读写队列或信号量的中断。在调度器启动之前，不能进行上下文切换。 还有一些 API 函数不能在调度器启动之前调用。在调用 vTaskStartScheduler()之前，最好是限定只使用创建任务，队列和信号量的 API 函数。 比如有一些初始化需要中断的，或者在初始化完成的时候回产生一个中断，这些驱动的初始化最好放在一个任务中进行，我是这样子处理的，在`main`函数中创建一个任务，在任务中进行bsp初始化，然后再创建消息队列、信号量、互斥量、事件以及任务等操作。

> 在调度器挂起时调用 API 函数，导致应用程序崩溃

调用 vTaskSuspendAll()使得调度器挂起，而唤醒调度器调用 xTaskResumeAll()。千万不要在调度器挂起时调用其它 API 函数。

喜欢就关注我吧！

相关代码可以在公众号后台获取。