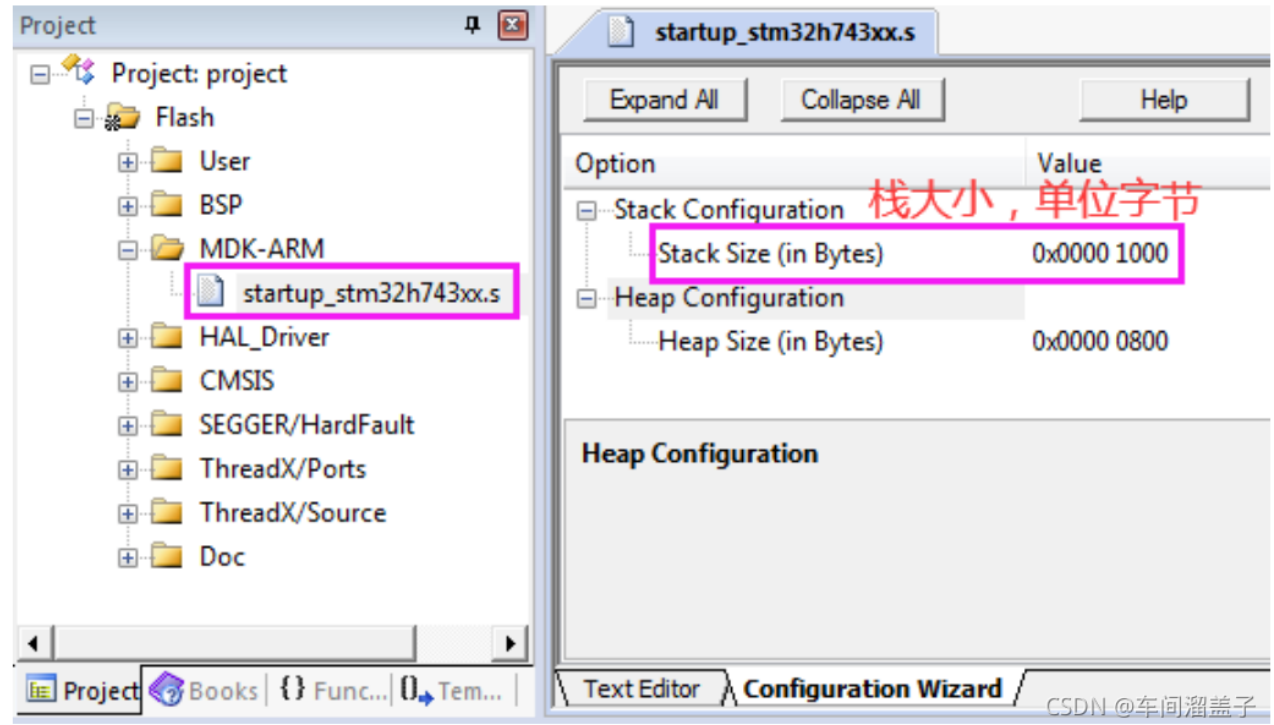
(163条消息) ThreadX_笔记_车间溜盖子的博客-CSDN博客_threadx

1.ThreadX–任务管理

1.1 裸机-栈

管是裸机编程还是 RTOS 编程，栈的分配大小都非常重要。 局部变量，函数调用时的现场保护和返回地址，函数的形参，进入中断函数前和中断嵌套等都需要栈空间，栈空间定义小了会造成系统崩溃。

裸机的情况下，用户可以在这里配置栈大小：



1.2 RTOS–栈

不同于裸机编程，在 RTOS 下，每个任务都有自己的栈空间。对于 ThreadX 来说，支持动态内存分配方式和静态分配方式，本教程全部是静态分配方式。

在 RTOS 下，上面1.1 截图中设置的栈大小有了一个新的名字叫系统栈空间，而任务栈是不使用这里的空间的。任务栈不使用这里的栈空间，哪里使用这里的栈空间呢？答案就在中断函数和中断嵌套。

1.3 MSP、PSP

MSP 主堆栈指针和 PSP 进程堆栈指针,或者叫**PSP** 任务堆栈指针也是可以的。在 **ThreadX** 操作系统中，主堆栈指针 **MSP** 是给系统栈空间使用的，进程堆栈指针 **PSP** 是给任务栈使用的。也就是说,在**ThreadX** 任务中，所有栈空间的使用都是通过 **PSP** 指针进行指向的。一旦进入了中断函数以及可能发生的中断嵌套都是用的 **MSP** 指针 。这个知识点要记住它，当前可以不知道这是为什么，但是一定要记住。

1.4 实际应用中系统栈空间分配多大？

⚫ 64 字节

对于 Cortex-M3 内核和未使用 FPU（浮点运算单元）功能的 Cortex-M4/M7 内核在发生中断时需要将 16 个通用寄存器全部入栈，每个寄存器占用 4 个字节，也就是 16*4 = 64 字节的空间。可能发生几次中断嵌套就是要 64 乘以几即可。当然，这种是最坏执行情况，也就是所有的寄存器都入栈。

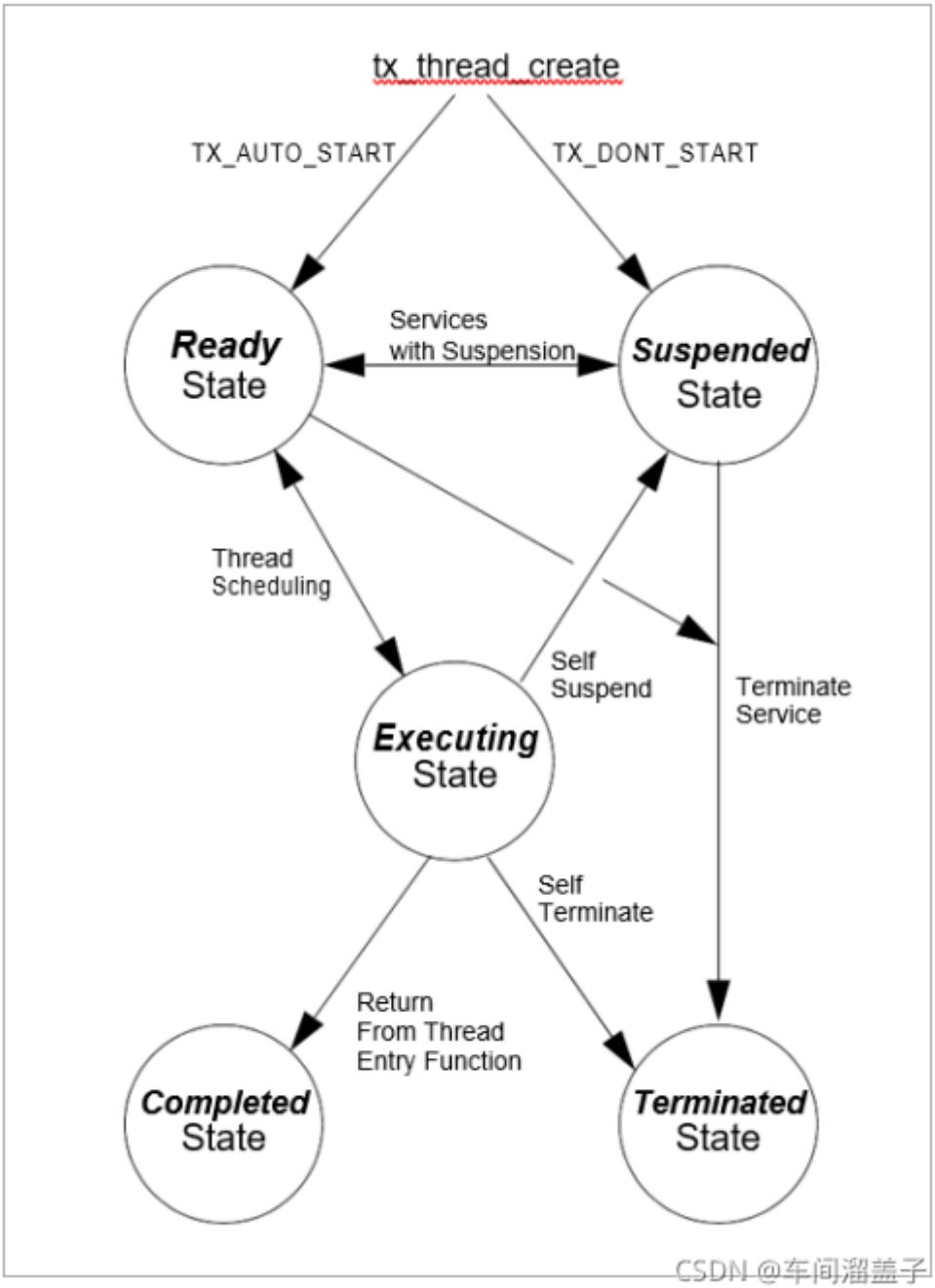
（注：任务执行的过程中发生中断的话，有 8 个寄存器是自动入栈的，这个栈是任务栈，进入中断以后其余寄存器入栈以及发生中断嵌套都是用的系统栈）

⚫ 200 字节

对于具有 FPU（浮点运算单元）功能的 Cortex-M4/M7 内核，如果在任务中进行了浮点运算，那么在发生中断的时候除了 16 个通用寄存器需要入栈，还有 34 个浮点寄存器也是要入栈的，也就是(16+34)*4 = 200 字节的空间。当然，这种是最坏执行情况，也就是所有的寄存器都入栈。

（注：任务执行的过程中发送中断的话，有 8 个通用寄存器和 18 个浮点寄存器是自动入栈的，这个栈是任务栈，进入中断以后其余通用寄存器和浮点寄存器入栈以及发生中断嵌套都是用的系统栈）

1.5 ThrreadX 的任务状态



◆ Executing State 执行态 当任务处于实际运行状态被称之为执行态，即 CPU 的使用权被这个任务占用。

◆ Ready State 就绪态

处于就绪态的任务是指那些能够运行（没有被挂起），但是当前没有运行的任务，因为同优先级或更

高优先级的任务正在运行。

◆ Suspended State 挂起态

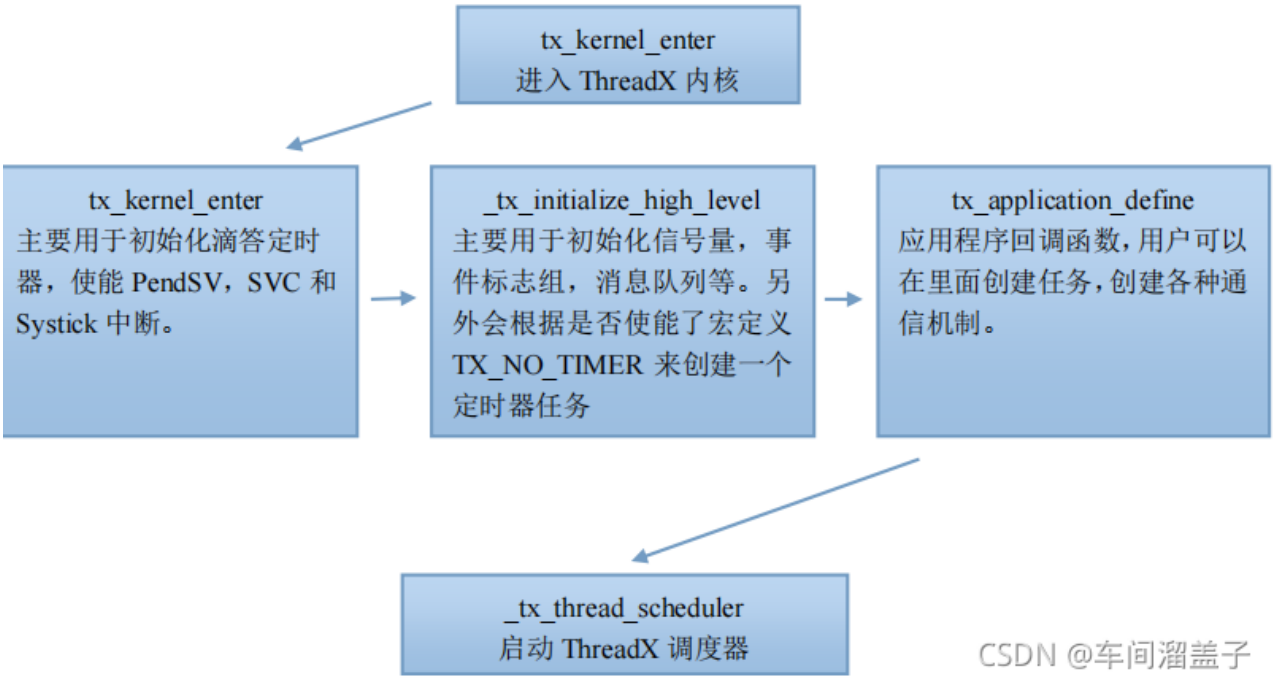
ThreadX 的挂起包含了阻塞，即由于等待信号量，消息队列，事件标志组等而处于的状态也是挂起态，任务调用延迟函数或者对任务进行挂起操作（有专门的挂起函数）也会处于挂起状态。 ◆ Completed State 完成态

任务返回的状态称之为完成态，正常情况下每个任务是死循环，独立执行，不会返回。 ◆ Terminated State 终止态

终止任务执行的状态称之为终止态。

1.6 ThreadX 启动流程

如下：



```
VOID  _tx_initialize_kernel_enter(VOID)
{

    /* Determine if the compiler has pre-initialized ThreadX.  */
    if (_tx_thread_system_state != TX_INITIALIZE_ALMOST_DONE)
    {

        /* No, the initialization still needs to take place.  */

        /* Ensure that the system state variable is set to indicate
           initialization is in progress.  Note that this variable is
           later used to represent interrupt nesting.  */
        _tx_thread_system_state =  TX_INITIALIZE_IN_PROGRESS;

        /* Call any port specific preprocessing.  */
        TX_PORT_SPECIFIC_PRE_INITIALIZATION

        /* Invoke the low-level initialization to handle all processor specific
           initialization issues.  */
        _tx_initialize_low_level();

        /* Invoke the high-level initialization to exercise all of the
           ThreadX components and the application's initialization
           function.  */
        _tx_initialize_high_level();

        /* Call any port specific post-processing.  */
        TX_PORT_SPECIFIC_POST_INITIALIZATION
    }

    /* Optional processing extension.  */
    TX_INITIALIZE_KERNEL_ENTER_EXTENSION

    /* Ensure that the system state variable is set to indicate
       initialization is in progress.  Note that this variable is
       later used to represent interrupt nesting.  */
    _tx_thread_system_state =  TX_INITIALIZE_IN_PROGRESS;

    /* Call the application provided initialization function.  Pass the
       first available memory address to it.  */
    tx_application_define(_tx_initialize_unused_memory);

    /* Set the system state in preparation for entering the thread
       scheduler.  */
    _tx_thread_system_state =  TX_INITIALIZE_IS_FINISHED;

    /* Call any port specific pre-scheduler processing.  */
    TX_PORT_SPECIFIC_PRE_SCHEDULER_INITIALIZATION

    /* Enter the scheduling loop to start executing threads!  */
    _tx_thread_schedule();

#ifdef TX_SAFETY_CRITICAL

    /* If we ever get here, raise safety critical exception.  */
    TX_SAFETY_CRITICAL_EXCEPTION(__FILE__, __LINE__, 0);
#endif
}
```

此函数依次调用了下面四个主要函数：
_tx_initialize_low_level ：主要用于初始化滴答定时器，使能 PendSV，SVC 和 Systick 中断。

_tx_initialize_high_level：主要用于初始化信号量，事件标志组，消息队列等。另外会根据是否使能了宏定义 TX_NO_TIMER 来创建一个定时器任务。

tx_application_define：应用程序回调函数，用户可以在里面创建任务，创建各种通信机制。

_tx_thread_scheduler：启动 ThreadX 调度器。

2.栈大小及溢出

2.1 任务栈大小的确定

在基于 RTOS 的应用设计中，每个任务都需要自己的栈空间，应用不同，每个任务需要的栈大小也是不同的。将如下的几个选项简单的累加就可以得到一个粗略的栈大小：

◆ 函数的嵌套调用，针对每一级函数用到栈空间的有如下四项：

1. 函数局部变量
2. 函数形参，一般情况下函数的形参是直接使用的 CPU 寄存器，不需要使用栈空间，但是这个函数中如果还嵌套了一个函数的话，这个存储了函数形参的 CPU 寄存器内容是要入栈的。所以建议大家也把这部分算在栈大小中。
3. 函数返回地址，针对 M3、 M4 和 M7 内核的 MCU，一般函数的返回地址是专门保存到 LR（Link Register）寄存器里面的，如果这个函数里面还调用了一个函数的话，这个存储了函数返回地址的 LR 寄存器内容是要入栈的。所以建议大家也把这部分算在栈大小中。
4. 函数内部的状态保存操作也需要额外的栈空间。

◆ 任务切换，任务切换时所有的寄存器都需要入栈，对于带 FPU 浮点处理单元的 M4/M7 内核 MCU 来 说，FPU 寄存器也是需要入栈的。

◆ 针对 M3 内核和 M4/M7 内核的 MCU 来说，在任务执行过程中，如果发生中断：

1. M3 内核的 MCU 有 8 个寄存器是自动入栈的，这个栈是任务栈，进入中断以后其余寄存器入栈以及发生中断嵌套都是用的系统栈。
2. M4/M7 内核的 MCU 有 8 个通用寄存器和 18 个浮点寄存器是自动入栈的，这个栈是任务栈，进入中断以后其余通用寄存器和浮点寄存器入栈以及发生中断嵌套都是用的系统栈。

◆ 进入中断以后使用的局部变量以及可能发生的中断嵌套都是用的系统栈，这点要注意。

一般来说，用户可以事先给任务分配一个大的栈空间，然后通过第 8 章介绍的调试方法打印任务栈的使用情况，运行一段时间就会有个大概的范围了。这种方法比较简单且实用些。

用户分配的栈空间不够用了，会导致栈溢出。

2.2 ThrreadX 的栈溢出检测机制

2.2.1 实现原理

（注：有些应用场景，这种栈检测是检测不出来的）。

ThreadX 提供了在运行时检查每个任务的栈是否损坏的功能。默认情况下，ThreadX 在创建过程中使用 oxEF 数据模式填充任务的每个字节。如果应用程序使能了宏定义 TX_ENABLE_STACK_CHECKING编译工程，则ThreadX 将检查每个任务的栈在挂起或恢复时是否损坏。如果检测到栈损坏，则 ThreadX将调用用户使用函数 tx_thread_stack_error_notify 设置的回调函数。否则，如果未指定堆栈错误处理程序，则 ThreadX 将调用内部_tx_thread_stack_error_handler 例程。

⚫ 栈溢出检测方法
除了 TreadX 提供的栈溢出检测机制，还有其它的栈溢出检测机制，大家可以在 Mircrium 官方发布的
如下这个博文中学习：
https://www.micrium.com/detecting-stack-overflows-part-2-of-2/

2.2.2 实现方法

◆ 使能栈检测
推荐直接在 tx_port.h 里面使能：
#define TX_ENABLE_STACK_CHECKING
◆ 注册回调：
大家可以随意设置注册的函数名：
tx_thread_stack_error_notify(my_stack_error_handler);
◆ 回调函数的实现
代码如下，：

```
void my_stack_error_handler(TX_THREAD *thread_ptr)
{
App_Printf("=================================================================\r\n");
 App_Printf("如下任务被检测出栈溢出\r\n");
 App_Printf("=================================================================\r\n");
 App_Printf(" 任务优先级 任务栈大小 当前使用栈 最大栈使用 任务名\r\n");
 App_Printf(" Prio StackSize CurStack MaxStack Taskname\r\n");

 TX_THREAD *p_tcb; /* 定义一个任务控制块指针 */
 p_tcb = &AppTaskStartTCB;

 /* 遍历任务控制列表 TCB list)，打印所有的任务的优先级和名称 */
 do
 {
 if(p_tcb != (TX_THREAD *)thread_ptr)
 {
 p_tcb = p_tcb->tx_thread_created_next;
 }
 else
 {

 App_Printf(" %2d %5d %5d %5d %s\r\n",
 p_tcb->tx_thread_priority,
 p_tcb->tx_thread_stack_size,
 (int)p_tcb->tx_thread_stack_end - (int)p_tcb->tx_thread_stack_ptr,
 (int)p_tcb->tx_thread_stack_end - (int)p_tcb->tx_thread_stack_highest_ptr,
 p_tcb->tx_thread_name);

 while(1);
 }
 }while(1);
}
```

3.ThrreadX 中断优先级配置，含 BasePrrii 配置方案

于这个 NVIC，有个重要的知识点就是优先级分组，抢占优先级和子优先级，下面就以 STM32 为 例进行介绍，STM32F1xx，F4xx，H7xx 都是只使用了这个 8 位寄存器的高四位[7:4]。

优先级分组	抢占优先级	响应优先级	描述
NVIC_PriorityGroup_0	0	0到15的取值	高4位全部是响应优先级，无抢占优先级
NVIC_PriorityGroup_1	0到1的取值	0到7的取值	抢占优先级为1位，响应优先级为3位
NVIC_PriorityGroup_2	0到3的取值	0到3的取值	抢占优先级为2位， 响应优先级为2位
NVIC_PriorityGroup_3	0到7的取值	0到1的取值	抢占优先级为3位，响应优先级为1位
NVIC_PriorityGroup_4	0到15的取值	0	高4位全部是抢占优先级，无响应优先级

Reset、NMI、Hard Fault 优先级为负数，高于普通中断优先级，且优先级不可配置。

在这里继续强调下这一点，在 NVIC 分组为 4 的情况下，抢占优先级可配置范围是 0-15，那么数值越小，抢占优先级的级别越高，即 0 代表最高优先级，15 代表最低优先级。

4.ThrreadX 任务优先级修改及其分配方案

4.1 任务优先级说明

下面对 ThreadX 优先级相关的几个重要知识点进行下说明，这些知识点在以后的使用中务必要掌握牢固。

◆ ThreadX 中任务的最大优先级数值是通过 tx_port.h 文件中的 TX_MAX_PRIORITIES 进行配置的，用 户实际可以使用的优先级范围是 0 到 configMAX_PRIORITIES – 1。比如我们配置此宏定义为 32，那么用户可以使用的优先级号是 0 到 31，不包含 32，对于这一点，初学者要特别的注意。并且TX_MAX_PRIORITIES 的宏定义设置的数值必须是 32 的整数倍：

#define TX_MAX_PRIORITIES 32

◆ 用户配置任务的优先级数值越小，那么此任务的优先级越高（0 是最高优先级任务），ThreadX 没有空闲任务，如果大家创建空闲任务，需要将其设置为最低优先级。

◆ 建议用户配置宏定义 TX_MAX_PRIORITIES 的最大值不要超过 32，即用户任务可以使用的优先级范围是 0 到 31。

⚫ 因为对于 CM 内核的移植文件，有专用的汇编指令 CLZ（Count Leading Zeros），通过这些指令可以加速算法执行速度。

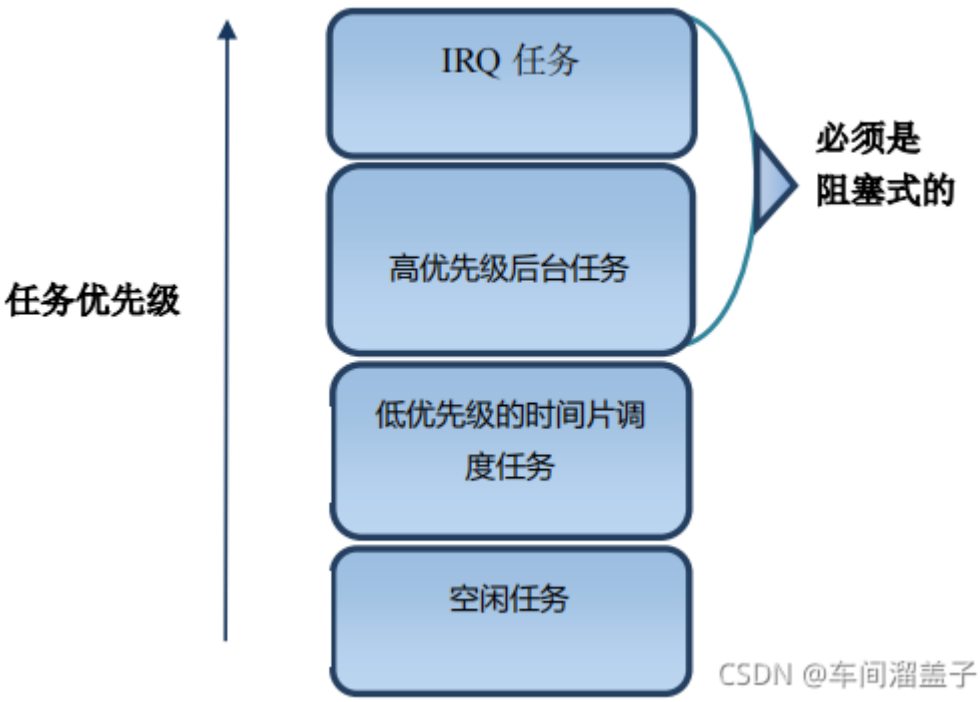
⚫ 比通用方式高效。

⚫ ThreadX 查找最高优先级任务是通过定义的 32bit 数组，ULONG _tx_thread_preempted_maps[TX_MAX_PRIORITIES/32];

如果 TX_MAX_PRIORITIES 设置为 32，那么仅需一个 32bit 变量就可以方便记录 ;32 不同优先级任务，程序运行时查找最高优先级任务也方便。

◆ ThreadX 中处于运行状态的任务永远是当前能够运行的最高优先级任务。下一章节讲解调度器，大家会对这个知识点有一个全面的认识。

4.2 任务优先级分配方案



◆ IRQ 任务：IRQ 任务是指通过中断服务程序进行触发的任务，此类任务应该设置为所有任务里面优先级最高的。

◆ 高优先级后台任务：比如按键检测，触摸检测，USB 消息处理，串口消息处理等，都可以归为这一类任务。

◆ 低优先级的时间片调度任务：比如 GUI 的界面显示，LED 数码管的显示等不需要实时执行的都可以归为这一类任务。实际应用中用户不必拘泥于将这些任务都设置为优先级 1 的同优先级任务，可以设置多个优先级，只需注意这类任务不需要高实时性。

◆ 空闲任务：空闲任务是系统任务。

◆ 特别注意：IRQ 任务和高优先级任务必须设置为阻塞式（调用消息等待或者延迟等函数即可），只有这样，高优先级任务才会释放 CPU 的使用权，,从而低优先级任务才有机会得到执行。
这里的优先级分配方案是我们推荐的一种方式，实际项目也可以不采用这种方法。调试出适合项目需求的才是最好的。

4.3 中断优先级和任务优先级区别

部分初学者也容易在这两个概念上面出现问题。简单的说，这两个之间没有任何关系，不管中断的优先级是多少，中断的优先级永远高于任何任务的优先级，即任务在执行的过程中，中断来了就开始执行中断服务程序。

另外对于 STM32 来说，中断优先级的数值越小，优先级越高。同样，ThreadX 的任务优先级也是任务优先级数值越小，任务优先级越高。

5.ThreadX任务调度–抢占式、时间片、合作式

5.1 ThrreadX 支持的调度方式

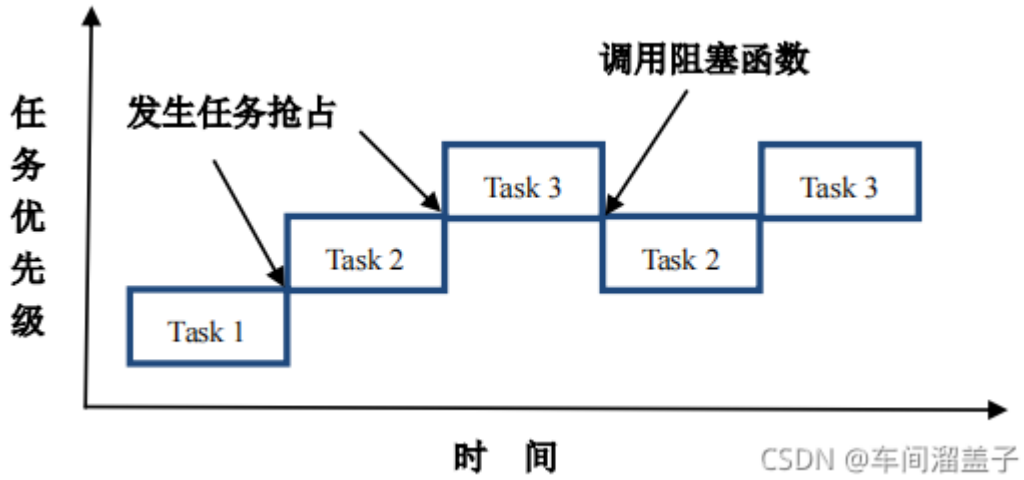
ThreadX 操作系统支持三种调度方式：抢占式调度，时间片调度和合作(轮询)式调度。实际应用主要是抢占式调度和时间片调度，合作(轮询)式调度用到的很少。

◆ 抢占式调度
每个任务都有不同的优先级，任务会一直运行直到被高优先级任务抢占或者遇到阻塞式的 API 函数，比如 tx_thread_sleep。

◆ 时间片调度
每个任务都有相同的优先级，任务会运行固定的时间片个数或者遇到阻塞式的 API 函数，比如tx_thread_sleep，才会执行同优先级任务之间的任务切换。

5.2 ThreadX 抢占式调度器的实现

下面我们通过如下的框图来说明一下抢占式调度在 ThreadX 中的运行过程，让大家有一个形象的认识。



运行条件：
◆ 这里仅对抢占式调度进行说明。
◆ 创建 3 个任务 Task1，Task2 和 Task3。
◆ Task1 的优先级为 1，Task2 的优先级为 2，Task3 的优先级为 3。FreeRTOS 操作系统是设置的数值越小任务优先级越低，故 Task3 的优先级最高，Task1 的优先级最低。
◆ 此框图是 ThreadX 操作系统运行过程中的一部分。

运行过程描述如下：
◆ 此时任务 Task1 在运行中，运行过程中由于 Task2 就绪，在抢占式调度器的作用下任务 Task2 抢占Task1 的执行。Task2 进入到运行态，Task1 由运行态进入到就绪态。

◆ 任务 Task2 在运行中，运行过程中由于 Task3 就绪，在抢占式调度器的作用下任务 Task3 抢占 Task2的执行。Task3 进入到运行态，Task2 由运行态进入到就绪态。

◆ 任务 Task3 运行过程中调用了阻塞式 API 函数，比如 tx_thread_sleep，任务 Task3 被挂起，在抢占式调度器的作用下查找到下一个要执行的最高优先级任务是 Task2，任务 Task2 由就绪态进入到运行态。

◆ 任务 Task2 在运行中，运行过程中由于 Task3 再次就绪，在抢占式调度器的作用下任务 Task3 抢占Task2 的执行。Task3 进入到运行态，Task2 由运行态进入到就绪态。

5.3 ThreadX 时间片调度器

5.3.1 时间片调度器基本概念

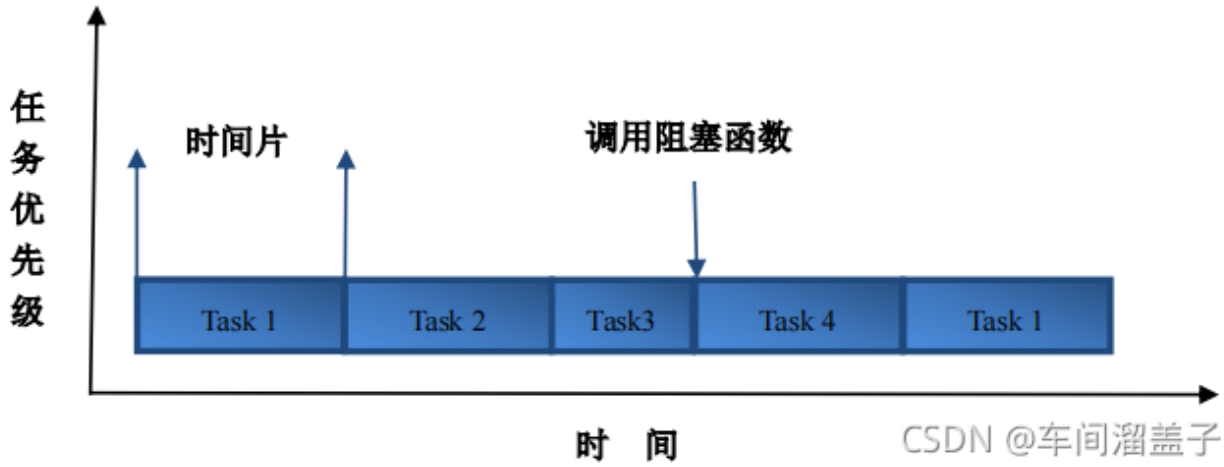
在小型的嵌入式 RTOS 中，最常用的的时间片调度算法就是 Round-robin 调度算法。这种调度算法可以用于抢占式或者合作式的多任务中。另外，时间片调度适合用于不要求任务实时响应的情况。

实现 Round-robin 调度算法需要给同优先级的任务分配一个专门的列表，用于记录当前就绪的任务，并为每个任务分配一个时间片（也就是需要运行的时间长度，时间片用完了就进行任务切换）。

5.3.2 时间片调度器的实现

在 ThreadX 操作系统中只有同优先级任务才会使用时间片调度，通过函数 tx_thread_create 的第 9个形参可以设置时间片大小，参数范围 0,到 0xFFFFFFFF，设置为 TX_NO_TIME_SLICE（对应的是数值 0）表示禁止时间片。

下面我们通过如下的框图来说明一下时间片调度在 ThreadX 中的运行过程，让大家有一个形象的认识。



运行条件：

◆ 这里仅对时间片调度进行说明。

◆ 创建 4 个同优先级任务 Task1，Task2，Task3 和 Task4。

◆ 每个任务分配的时间片大小是 5 个系统时钟节拍。

运行过程描述如下：

◆ 先运行任务 Task1，运行够 5 个系统时钟节拍后，通过时间片调度切换到任务 Task2。

◆ 任务 Task2 运行够 5 个系统时钟节拍后，通过时间片调度切换到任务 Task3。

◆ 任务 Task3 在运行期间调用了阻塞式 API 函数，调用函数时，虽然 5 个系统时钟节拍的时间片大小还没有用完，此时依然会通过时间片调度切换到下一个任务 Task4。（注意，没有用完的时间片不会再使用，下次任务 **Task3** 得到执行还是按照 **5** 个系统时钟节拍运行）

◆ 任务 Task4 运行够 5 个系统时钟节拍后，通过时间片调度切换到任务 Task1。

上面就是一个简单的同优先级任务通过时间片调度进行任务调度和任务切换的过程。