FreeRTOS 操作系统配置参数说明

金涛

（上海致远绿色能源股份有限公司　上海　201611 ）

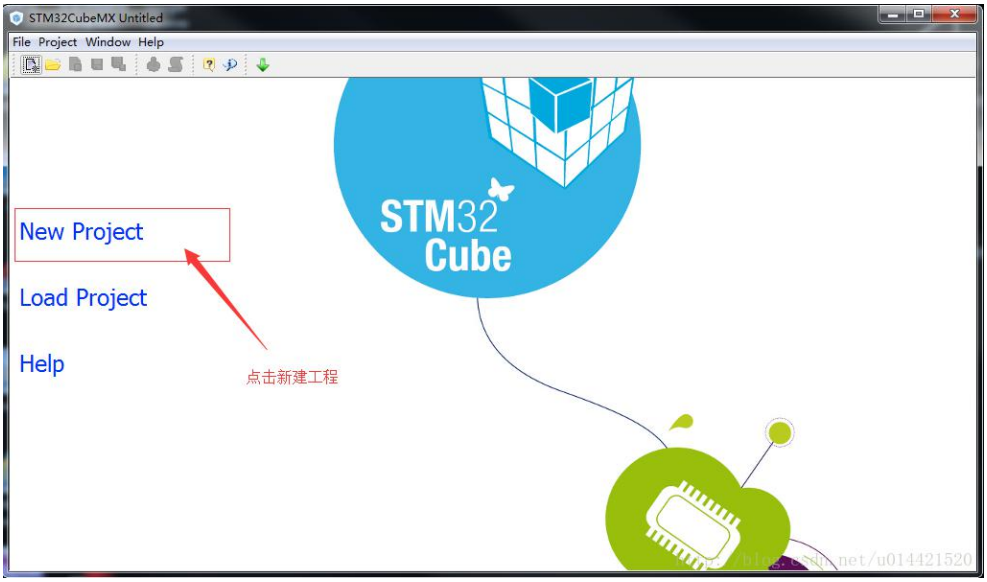
本文仅针对 FreeRTOS 操作系统的配置参数进行说明，其他内容不在累述。如有需要，请参考相关文档。

本文所涉及软件的版本如下图所示：

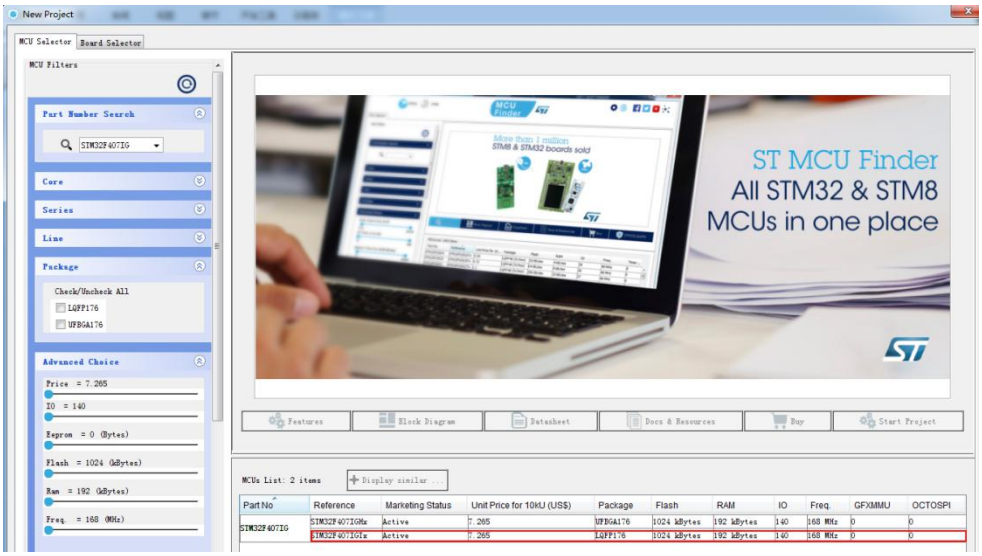
名称		版本
STM32Cube MX	Software to configure and manage STM32 MCUs	4.32.0
	Firmware Package for family STM32 F4	1.18.0
Keil		5.23.0.0

一、生成带 FreeRTOS 操作系统代码的步骤

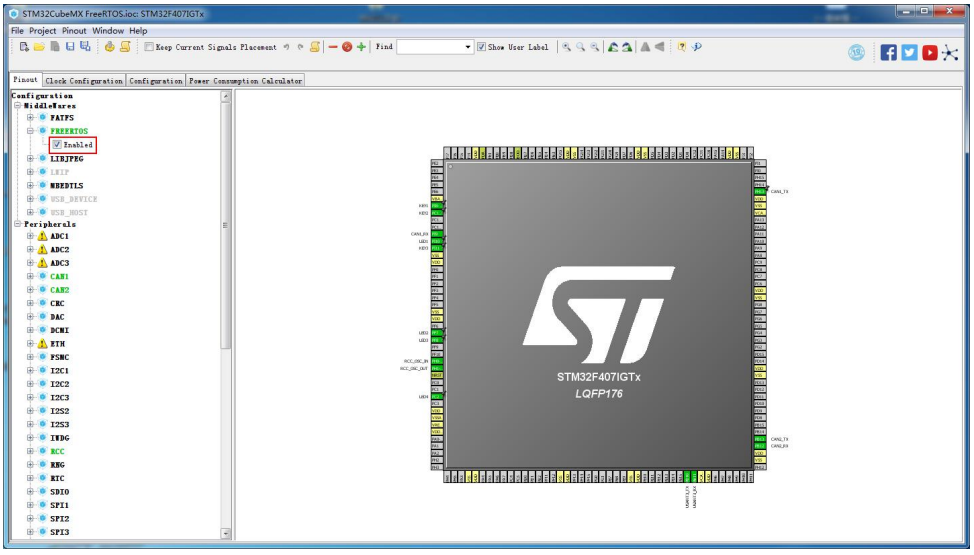
1. 下载安装的过程就不在赘述，直接开始使用。 双击打开 cube MX，点击新建工程；



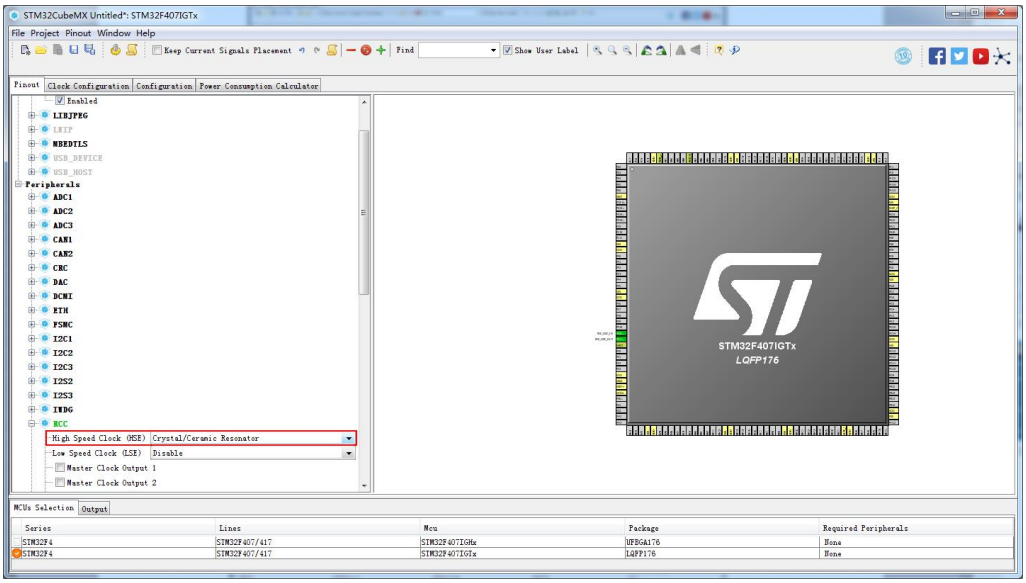
2. 本文以 STM32F407IGT6 作为所选 MCU，具体使用根据最终所选型号进行配置：选择 STM32F407IGTx；



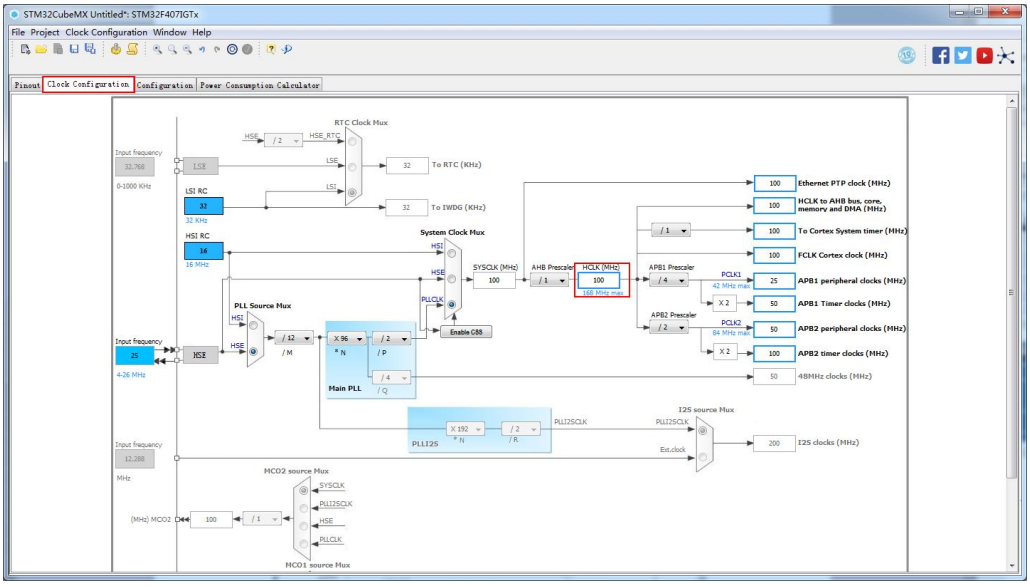
3. 使用 STM32CubeMX 来生成带 MiddleWare 的 FreeRTOS 的代码 ，可以在 STM32CubeMX 中使能 FreeRTOS：Configuration --> MiddleWare -- >FreeRTOS --> Enable；



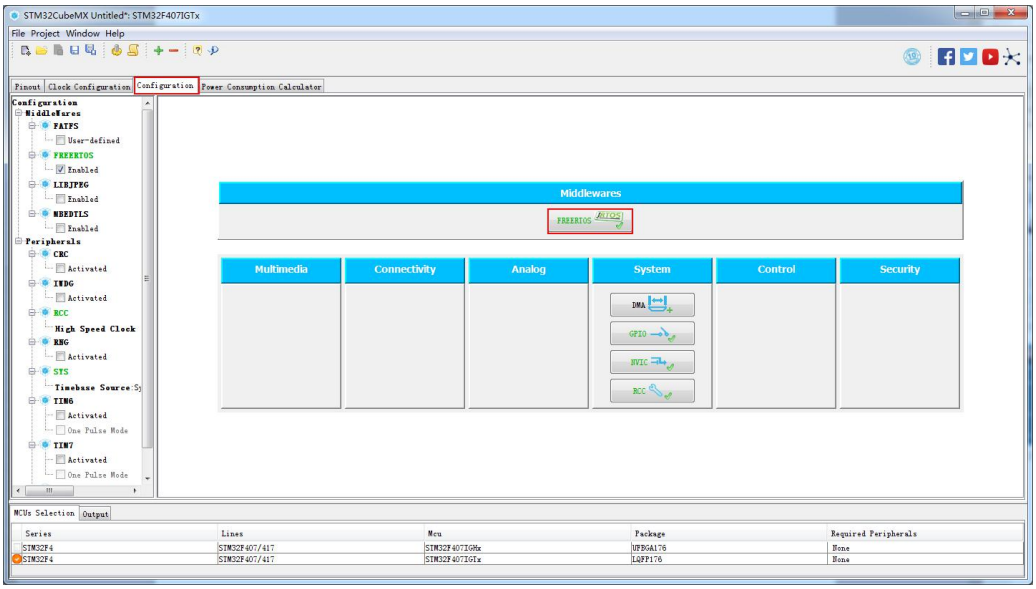
4. RCC 时钟源配置为外部时钟：Configuration --> Peripherals --> RCC --> HSE --> CryStal/Ceramic Resonator；

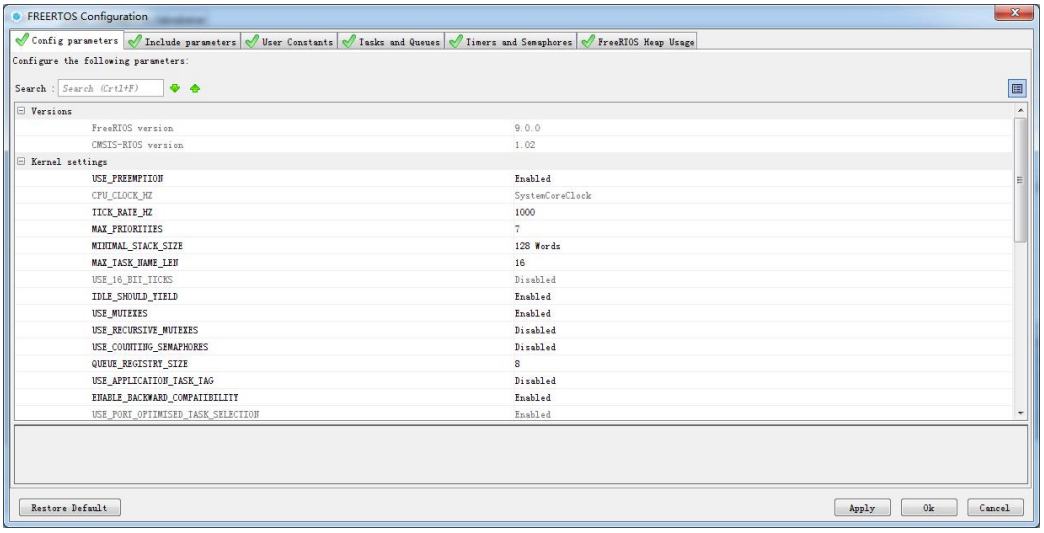


5. 配置时钟系统，直接将 HCLK 设置为 100MHz，其他的会根据此自动选择。



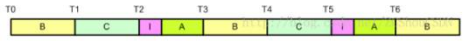
6. FreeRTOS 配置：Configuration --> FreeRTOS





FreeRTOS 宏定义参数配置		
Confi Parameters		
参数组	参数	释义
Versions	FreeRTOS Version	9.0.0
	CMSIS-RIOS Version	处理器系列的与供应商无关的硬件抽象层，1.02
Kernel Settings	USE_PREEMPTION	Enabled 使能抢占式调度器，Disabled 使用合作式调度器。勾选 Enabled
	CPU_CLOCK_HZ	参数用于定义 CPU 的主频，单位 Hz，不可修改
	TICK_RATE_HZ	此参数用于定义系统时钟节拍数，单位 Hz。取 1000Hz 即可。 系统节拍中断用来测量时间，因此，越高的测量频率意味着可测到越高的分辨率时间。但是，高的系统节拍中断频

		率也意味着 FreeRTOS 内核占用更多的 CPU 时间，因此会降低效率。缺省选项的系统节拍中断频率为 1000Hz，实际使用时不用这么高的系统节拍中断频率。 多个任务可以共享一个优先级，FreeRTOS 调度器为相同优先级的任务分享 CPU 时间，在每一个 RTOS 系统节拍中断到来时进行任务切换。高的系统节拍中断频率会降低分配给每一个任务的“时间片”持续时间。
	MAX_PRIORITIES	设置任务优先级的数量：配置应用程序有效的优先级数目。任何数量的任务都可以共享一个优先级，使用协程可以单独的给与它们优先权。见 MAX_CO _ROUTINE_PRIORITIES。 在 RTOS 内核中，每个有效优先级都会消耗一定量的 RAM，因此这个值不要超过你的应用实际需要的优先级数目。 每一个任务都会被分配一个优先级，优先级值从 0~ （MAX_PRIORITIES - 1）之间。低优先级数表示低优先级任务。空闲任务的优先级为 0（PriorityIdle），因此它是最低优先级任务。 FreeRTOS 调度器将确保处于就绪状态（Ready）或运行状态（Running）的高优先级任务比同样处于就绪状态的低优先级任务优先获取处理器时间。换句话说，处于运行状态的任务永远是高优先级任务。 处于就绪状态的相同优先级任务使用时间片调度机制共享处理器时间。
	MINIMAL_STACK_SIZE	此参数用于定义空闲任务的栈空间大小，单位字。默认 128 个字
	MAX_TASK_NAME_LEN	定义任务名最大的字符数
	USE_16_BIT_TICKS	系统时钟节拍计数使用 TickType_t 数据类型定义的。 如果使能了此宏定义，那么 TickType_t 定义的就是 16 位无符号数；如果没有使能，那么 TickType_t 定义的就是 32 位无符号数。对于 32 位架构的处理器，一定要禁止此宏定义
	IDLE_SHOULD_YIELD	只有满足以下两个条件时，此参数才有效果：

		1. 使能抢占式调度器。 2. 有创建与空闲任务同优先级的任务。 通过时间片共享同一个优先级的多个任务，如果共享的优先级大于空闲优先级，并假设没有更高优先级任务，这些任务应该获得相同的处理器时间。 但如果共享空闲优先级时，情况会稍微有些不同。当 IDLE_SHOULD_YIELD 为 Enabled 时，其它共享空闲优先级的用户任务就绪时，空闲任务立刻让出 CPU，用户任务运行，这样确保了能最快响应用户任务。处于这种模式下也会有不良效果（取决于你的程序需要），描述如下：  图中描述了四个处于空闲优先级的任务，任务 A、B 和 C 是用户任务，任务 I 是空闲任务。上下文切换周期性的发生在 T0、T1…T6 时刻。当用户任务运行时，空闲任务立刻让出 CPU，但是，空闲任务已经消耗了当前时间片中的一定时间。这样的结果就是空闲任务 I 和用户任务 A 共享一个时间片。用户任务 B 和用户任务 C 因此获得了比用户任务 A 更多的处理器时间。 ①、可以通过下面方法避免： 如果合适的话，将处于空闲优先级的各单独的任务放置到空闲钩子函数中； 创建的用户任务优先级大于空闲优先级； ②、设置 IDLE_SHOULD_YIELD 为 Disabled；将阻止空闲任务为用户任务让出 CPU，直到空闲任务的时间片结束。这确保所有处在空闲优先级的任务分配到相同多的处理器时间，但是，这是以分配给空闲任务更高比例的处理器时间为代价的。
	USE_MUTEXES	Enabled 使能互斥信号量；Disable 禁能互斥信号量。勾选 Enabled。 互斥型信号量必须是同一个任务申请，同一个任务释放，其他任务释放无效。二进制信号量，一个任务申请成功后，

		可以由另一个任务释放。 互斥型信号量是二进制信号量的子集
	USE_RECURSIVE_MUTEXES	Enabled 使能递归互斥信号量；Disable 禁能递归互斥信号量
	USE_COUNTING_SEMAPHORES	Enabled 使能计数信号量； Disable 禁能计数信号量
	QUEUE_REGISTRY_SIZE	通过此定义来设置可以注册的信号量和消息队列个数。 队列注册有两个目的，这两个目的都与内核调试有关： 注册队列的时候，可以给队列起一个名字，当使用调试组件的时候，通过名字可以很容易的区分不同队列。通过队列的相关信息，调试器可以很容易定位队列和信号量，能够定位信号量是因为 FreeRTOS 信号量也是基于队列实现的。 当然，如果没有使用内核方面的调试器，这个宏定义是没有意义的。
	USE_APPLICATION_TASK_TAG	使能任务标签功能
	ENABLE_BACKWARD_COMPATIBILITY	Enabled 使能新版本对老版本的兼容特性，即向后兼容或者说向下兼容； Disable 禁止此特性。 头文件 FreeRTOS.h 包含一系列#define 宏定义，用来映射版本 V8.0.0 和 V8.0.0 之前版本的数据类型名字。这些宏可以确保 RTOS 内核升级到 V8.0.0 或以上版本时，之前的应用代码不用做任何修改。在 FreeRTOSConfig.h 文件中设置宏 ENABLE_BACKWARD_COMPATIBILITY 为 Disable 会去掉这些宏定义，并且需要用户确认升级之前的应用没有用到这些名字。
	USE_PORT_OPTIMISED_TASK_SELECTION	FreeRTOS 的硬件有两种方法选择下一个要执行的任务：通用方法和特定于硬件的方法。 Disabled:可以用于所有 FreeRTOS 支持的硬件；完全用 C 实现，效率略低于特殊方法；不强制要求限制最大可用优先级数。 Enable：并非所有硬件都支持。依赖于一个或多个架构特定汇编指令（通常为计数前导零（CLZ）或等效指令），因

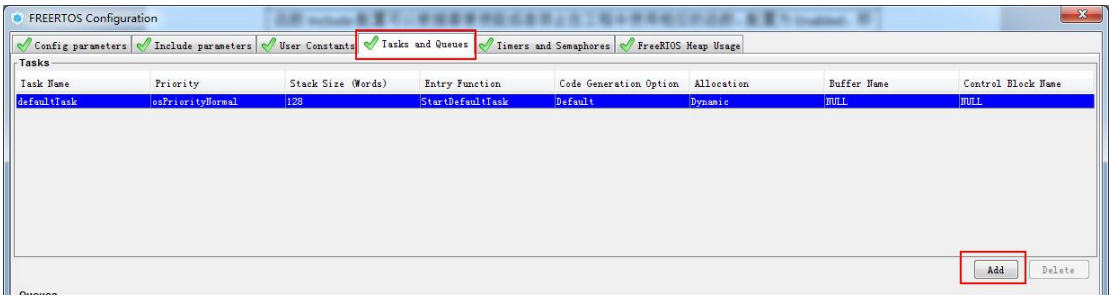
		此只能与其专门编写的体系结构一起使用；比通用方法更有效率
	USE_TICKLESS_IDLE	此配置用于使能 tickless 低功耗模式：Enabled 使能 tickless 低功耗模式；Disable 保持系统节拍（SysTick）中断一直运行。勾选 Disable。 常情况下，FreeRTOS 回调空闲任务钩子函数，在空闲任务钩子函数中设置微处理器进入低功耗模式来达到省电的目的。因为系统要响应系统节拍中断事件，因此使用这种方法会周期性的退出、再进入低功耗状态。如果系统节拍中断频率过快，则大部分电能和 CPU 时间会消耗在进入和退出低功耗状态上。FreeRTOS 的 tickless 空闲模式会在空闲周期时停止周期性系统节拍中断。停止周期性系统节拍中断可以使微控制器长时间处于低功耗模式。移植层需要配置外部唤醒中断，当唤醒事件到来时，将微控制器从低功耗模式唤醒。微控制器唤醒后，会重新使能系统节拍中断。由于微控制器在进入低功耗后，系统节拍计数器是停止的，但我们又需要知道这段时间能折算成多少次系统节拍中断周期，这就需要有一个不受低功耗影响的外部时钟源，即微处理器处于低功耗模式时它也在计时的，这样在重启系统节拍中断时就可以根据这个外部计时器计算出一个调整值并写入 FreeRTOS 系统节拍计数器变量中。
	USE_TASK_NOTIFICATIONS	设置 USE_TASK_NOTIFICATIONS 为 Enable 将会开启任务通知功能，有关的 API 函数也会被编译。设置宏 USE_TASK_NOTIFICATIONS 为 Disabled 则关闭任务通知功能，相关 API 函数也不会被编译。默认这个功能是开启的。开启后，每个任务多增加 8 字节 RAM。 每个 FreeRTOS 任务具有一个 32 位的通知值，FreeRTOS 任务通知相当于直接向任务发送一个事件，接收到通知的任务可以解除任务的阻塞状态（因等待任务通知而进入阻塞状态）。相对于以前必须分别创建队列、二进制信号量、计数信号量或事件组的情况，使用任务通知

		显然更灵活。更好的是，相比于使用信号量解除任务阻塞，使用任务通知可以快 45%（使用 GCC 编译器，-o2 优化级别）。
Memory manage settings	Memory allocation	动态内存/静态内存选择
	TOTAL_HEAP_SIZE	定义堆大小，FreeRTOS 内核，用户动态内存申请，任务栈等都需要用这个空间
	Memory Management scheme	FreeRTOS 内存管理方案，共有 5 种：heap_1，heap_2，heap_3，heap_4，heap_5
Hook function related definitions 钩子函	USE_IDLE_HOOK	Enable 使能空闲任务的钩子函数； Disable 禁能空闲任务钩子函数 当 FreeRTOS 调度器开始工作后，为了保证至少有一个任务在运行，空闲任务被自动创建，占用最低优先级（0 优先级）。对于已经删除的 FreeRTOS 任务，空闲任务可以释放分配给它们的堆栈内存。因此，在应用中应该注意，使用 vTaskDelete()函数时要确保空闲任务获得一定的处理器时间。除此之外，空闲任务没有其它特殊功能，因此可以任意的剥夺空闲任务的处理器时间。 空闲任务钩子是一个函数，这个函数由用户来实现，FreeRTOS 规定了函数的名字和参数，这个函数在每个空闲任务周期都会被调用。要创建一个空闲钩子： 1. 设置 USE_IDLE_HOOK 为 Enabled； 2. 定义一个函数，函数名和参数如下所示： void vApplicationIdleHook(void); 该钩子函数不可以调用会引起空闲任务阻塞的 API 函数（例如：vTaskDelay()、带有阻塞时间函数），在钩子函数内部可以使用协程。 使用空闲钩子函数设置 CPU 进入省电模式是很常见的。
	USE_TICK_HOOK	Enable 使能滴答定时器中断里面执行的钩子函数； Disable 禁能滴答定时器中断里面执行的钩子函数。 时间片中断可以周期性的调用一个被称为钩子函数（回调函数）的应用程序。时间片钩子函数可以很方便的实现一

		个定时器功能。 USE_TICK_HOOK 设置成 Enable 时才可以使用时间片钩子。一旦此值设置成 Enable，就要定义钩子函数
	USE_MALLOC_FAILED_HOOK	当创建任务，信号量或者消息队列时，FreeRTOS 通过函数 pvPortMalloc()申请动态内存。 Enable 使能动态内存申请失败时的钩子函数；Disable 禁能动态内存申请失败时的钩子函数
	USE_DAEMON_TASK_STARTUP_HOOK	如果 USE_TIMERS 和 USE_DAEMON_TASK_STARTUP_HOOK 都设置为 Enabled，那么应用程序必须定义一个钩子函数。当 FreeRTOS 守护程序任务（也称为定时器服务任务）第一次执行时，钩子函数将被精确调用一次。 需要 RTOS 运行的任何应用程序初始化代码都可以放在 hook 函数中
	CHECK_FOR_STACK_OVERFLOW	配置栈溢出的检测方法，Option1： 使用检测方法:1 ；Option1：使用检测方法；Option3：禁止使用
Run time and task stats gathering related definitions 任务运行信息获取配置	GENERATE_RUN_TIME_STATS	使能任务运行状态参数统计
	USE_TRACE_FACILITY	设置成 Enabled 表示启动可视化跟踪调试，会激活一些附加的结构体成员和函数。
	USE_STATS_FORMATTUNG_FUNCTIONS	设置宏 USE_TRACE_FACILITY 和 USE_STATS_FORMATTING_FUNCTIONS 为 Enabled 会编译 vTaskList()和 vTaskGetRunTimeStats()函数。如果将这两个宏任意一个设置为 Disabled，上述两个函数将被忽略
Co-routine related definitions 合作式任务配置	USE_CO_ROUTINES	使能合作式调度相关函数选项，勾选 Disabled
	MAX_CO_ROUTINE_PRIORITIES	定义可供用户使用的最大的合作式任务优先级。
Software time definitions 软件定时器配置	Use times	使能软件定时器
Interrupt nesting behaviour configuration 断言配置	LIBRARY_LOWEST_INTERRUPT_PRIORITY	此宏定义是用来配置 FreeRTOS 中用到的 SysTick 中断和 PendSV 中断的优先级 reeRTOSreeRTOS reeRTOSreeRTOS 用到的 SysTSysT ick 中断和 PendSV 中断的优先级。在 NVIC 分组设置为 4 的情

		况下，此宏定义的范围就是 0-15 ，即专门配置抢占优先级。这里 即专门配置抢占优先级。这里 配置 为了 15，即 SysTick 和 PendSV 都配配置为了最低优先级，实际项目中也建议配置为最低优先级。
	LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY	定义了受 FreeRTOS 管理的最高优先级中断。简单的说就是允许用户在这个中断服务程序里面调用 FreeRTOS 的 API 的最高优先级。设置 NVIC 的优先级分组为 4 的情况下（全部配置为抢占式优先级。又因为 STM32 的优先级设置仅使用 CM 内核 8bit 中的高 4bit，即只能区分 2^4 = 16 种优先级。因此当优先级分组设置为 4 的时候可供用户选择抢占式优先级为 0 到 15，共 16 个优先级，配置为 0 表示最高优先级，配置为 15 表示最低优先级，不存在子优先级。），配置 LIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 可以在抢占式优先级为 1 到 15 的中断里面调用 FreeRTOS 的 API 函数，抢占式优先级为 0 的中断里面是不允许调用的。
函数 Include 配置		
函数 Include 配置可以根据需要使能或者禁止在工程中使用相应的函数。配置为 Enabled，即可以在工程中使用相应函数。配置为 Disabled，即禁止在工程中使用相应函数。对于配置为 Enabled，但在工程中没有用到的函数，编译器一般都会把这些冗余函数删掉，不会添加到最终的 hex 文件中。		

7. 任务的建立



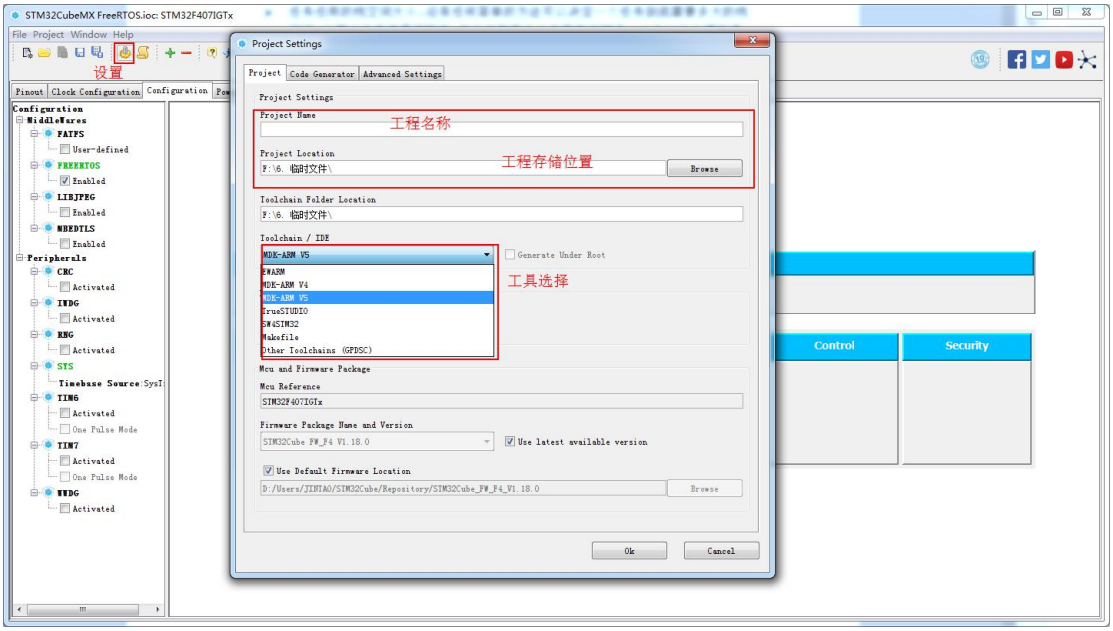
任务的建立可以在 CubeMX 中的 Task and Queues 中建立，也可以在 Keil 中建立。本文要求在 CubeMX 中添加任务，尽可能少的减少程序员的编程量。

- Task Name：任务名，格式要求 Task... ...;
- Priority：任务优先级。STM32Cube 对 FreeRTOS 进行了一些修改，优先级只有 7 个，优先级从低到高依次是：osPriorityIdle、osPriorityLow、osPriorityBelowNormal、osPriorityNormal、osPriorityAboveNormal、osPriorityHigh、osPriorityRealtime；

- Stack Size（Words）：任务栈大小（单位字）。定义常量 configMINIMAL_STACK_SIZE 来决定空闲
- 任务任用的栈空间大小。没有任何简单的方法可以决定一个任务到底需要多大的栈空间。计算出来虽然是可能的，但大多数用户会先简单地赋予一个自认为合理的值，然后利用 FreeRTOS 提供的特性来确证分配的空间既不欠也不浪费。本文会提供一些信息，可以知道如何去查询任务使用了多少栈空间。
- Entry Function：入口功能

8. 代码生成

工程设置：工程名称、工程的存储位置、开发环境及版本 --> 确定 -->代码生成



二、配置参数引申（可以略过）

1. FreeRTOS 支持的调度方式

（1） FreeRTOS 的调度方式

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

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。

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，才会执行同优先级任务之间的任务切换。

（2） 抢占式调度

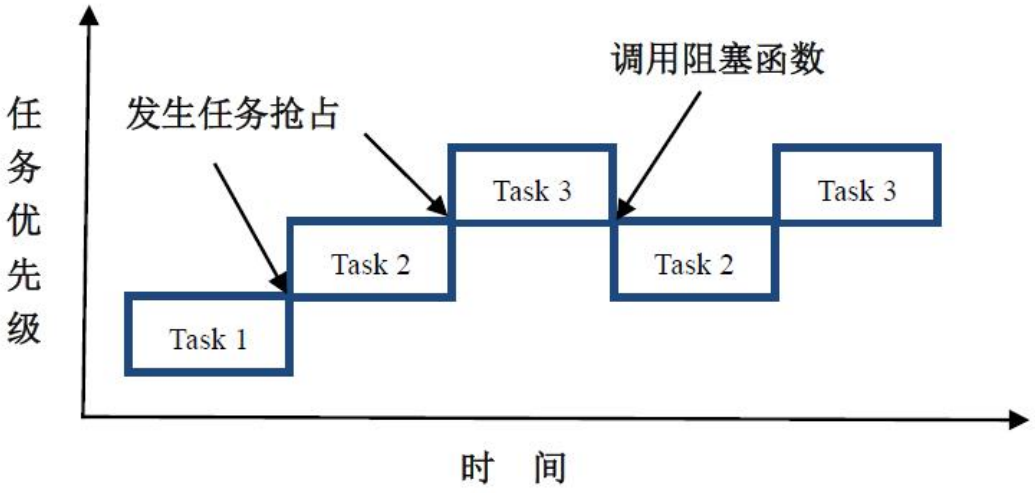
抢占式调度，最高优先级的任务一旦就绪，总能得到 CPU 的控制权。比如，当一个运行着的任务被其它高优先级的任务抢占，当前任务的 CPU 使用权就被剥夺了，或者说被挂起了，那个高优先级的任务立刻得到了 CPU 的控制权并运行。又比如，如果中断服务程序使一个高优先级的任务进入就绪态，中断完成时，被中断的低优先级任务被挂起，优先级高的那个任务开始运行。 使用抢占式调度器，使得最高优先级的任务什么时候可以得到 CPU 的控制权并运行是可知的，同时使得任务级响应时间得以最优化。

如果在 CubeMX 中的 USE_PREEMPTION 配置为 Enabled，即使用抢占式调度，那

么每个任务必须配置不同的优先级。当 FreeRTOS 多任务启动执行后，基本会按照如下的方式去执行：

首先执行的最高优先级的任务 Task1，Task1 会一直运行直到遇到系统阻塞式的 API 函数，比如延迟，事件标志等待，信号量等待，Task1 任务会被挂起，也就是释放 CPU 的执行权，让低优先级的任务得到执行。

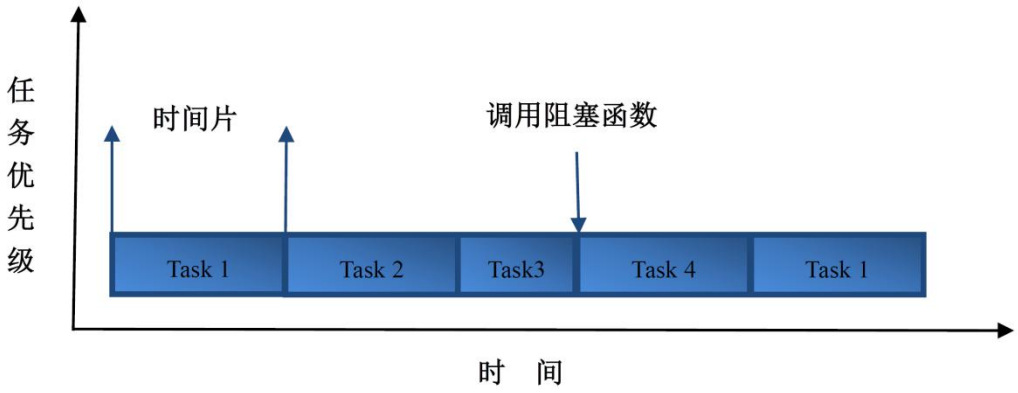
FreeRTOS 操作系统继续执行任务就绪列表中下一个最高优先级的任务 Task2，Task2 执行过程中有两种情况：Task1 由于延迟时间到，接收到信号量消息等方面的原因，使得 Task1 从挂起状态恢复到就绪态，在抢占式调度器的作用下，Task2 的执行会被 Task1 抢占。Task2 会一直运行直到遇到系统阻塞式的 API 函数，比如延迟，事件标志等待，信号量等待，Task2 任务会被挂起，继而执行就绪列表中下一个最高优先级的任务。如果用户创建了多个任务并且采用抢占式调度器的话，基本都是按照上面两条来执行。根据抢占式调度器，当前的任务要么被高优先级任务抢占，要么通过调用阻塞式 API 来释放 CPU 使用权让低优先级任务执行，没有用户任务执行时就执行空闲任务。



（3）时间片调度

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）。

在 FreeRTOS 操作系统中只有同优先级任务才会使用时间片调度，另外还需要用户在 FreeRTOSConfig.h 文件中使能宏定义： #define configUSE_TIME_SLICING 1 默认情况下，此宏定义已经在 FreeRTOS.h 文件里面使能了，用户可以不用在 FreeRTOSConfig.h 文件中再单独使能。



2. 任务栈大小的确定

（1） 粗略估算

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：

函数局部变量。

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。

函数返回地址，针对 M3 和 M4 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。

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 内核 MCU 来说，FPU 寄存器也是需要入栈的。

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

M4 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。实际应用中将这些都加起来是一件非常麻烦的工作，上面这些栈空间加起来的总和只是栈的最小需求，实际分配的栈大小可以在最小栈需求的基础上乘以一个安全系数，一般取 1.5-2。上面的计算是我们用户可以确定的栈大小，项目应用中还存在无法确定的栈大小，比如调用 printf 函数就很难确定实际的栈消耗。又比如通过函数指针实现函数的间接调用，因为函数指针不是固定的指向一个函数进行调用，而是根据不同的程序设计可以指向不同的函数，使得栈大小的计算变得比较麻烦。

（2） 利用调试方法打印任务栈使用情况

参见上文中“Run time and task stats gathering related definitions”设置，获取任务运行信息，以获得任务栈剩余大小。