FreeRTOS 参数配置说明(http://www.openmcu.net/post/kernel-config.html)

configUSE_PREEMPTION

为 1 时 RTOS 使用抢占式调度器，为 0 时 RTOS 使用协作式调度器。

注：在多任务管理机制上，操作系统可以分为抢占式和协作式两种。协作式操作系统是任务主动释放 CPU 后，切换到下一个任务。任务切换的时机完全取决于正在运行的任务。

configUSE_IDLE_HOOK

设置为 1 使用空闲钩子（Idle Hook），0 忽略空闲钩子。

注 1：空闲任务（Idle Task）

当 RTOS 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。对于已经删除的 RTOS 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，因此可以可以的剥夺空闲任务的处理器时间。

应用程序也可能和空闲任务共享同个优先级。

注 2：空闲任务钩子（Idle Task Hook）

空闲任务钩子是一个函数，每个空闲任务周期都被调用。如果你想将应用程序某功能运行在空闲任务优先级下，需要执行以下两个步骤：

1.在空闲任务钩子中实现某功能

功能必须位于一个死循环中（操作系统的任务一般都在一个死循环中实现），这个钩子函数不可以调用可以引起空闲任务阻塞的 API 函数（例如：vTaskDelay()、带有阻塞时间的队列和信号量函数），在钩子函数内部使用协程是被允许的。

2.为实现的功能创建一个空闲优先级任务

有很多灵活的解决方案，但会消耗比较多的 RAM，见

以下网址有更多使用空闲任务的例子：http://www.freertos.org/tutorial/

要创建一个空闲钩子：

1. 设置 FreeRTOSConfig.h 文件中的 configUSE_IDLE_HOOK 为 1；

2.定义一个函数，函数名和参数如下所示：

```
void vApplicationIdleHook( void );
```

使用空闲钩子函数设置 CPU 进入省电模式是很常见的。

configUSE_TICK_HOOK

设置为 1 使用时间片钩子（Tick Hook），0 忽略时间片钩子。

注：时间片钩子函数（Tick Hook Function）

时间片中断可以可选择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。

只有在 FreeRTOSConfig.h 中的 configUSE_TICK_HOOK 设置成 1 时才可以使用时间片钩子。一旦此值设置成 1，就要定义钩子函数，函数名和参数如下所示：

```
void vApplicationTickHook( void );
```

vApplicationTickHook()函数在中断服务程序中执行，因此这个函数必须非常短小，不能大量使用堆栈，不能调用以”FromISR" 或 "FROM_ISR”结尾的 API 函数。

在 FreeRTOSV7.2.0\FreeRTOS\Demo\Common\Minimal 文件夹下的 crhook.c 文件中有使用时间片钩子函数的例程。

configCPU_CLOCK_HZ

写入实际的 CPU 内核时钟频率，配置此值是为了正确的配置时钟外设。

configTICK_RATE_HZ

RTOS Tick 中断的频率。即一秒中断的次数，每次中断 RTOS 都会进行任务调度。

Tick 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 Tick 中断频率也意味着 RTOS 内核占用更多的 CPU 时间，因此会降低效率。RTOS 演示例程都是使用 Tick 中断频率为 1000HZ，这是为了测试 RTOS 内核，比实际使用的要高。（实际使用不是这么高的 Tick 中断频率）

多个任务可以共享一个优先级，RTOS 调度器为相同优先级的任务分享 CPU 时间，在每一个 RTOS Tick 中断到来时进行任务切换。高的 Tick 中断频率会降低每一个任务分配到的“时间片”。

configMAX_PRIORITIES

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 configMAX_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。

注：任务优先级

每一个任务都会被分配一个优先级，优先级值从 0~（configMAX_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（tskIDLE_PRIORITY）。

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。

configMINIMAL_STACK_SIZE

定义空闲任务使用的堆栈大小。通常此值不应小于对应处理器演示例程文件 FreeRTOSConfig.h 中定义的数值。

configTOTAL_HEAP_SIZE

RTOS 内核总计可用的有效的 RAM 大小。仅在你使用 FreeRTOS 下载包中附带的内存分配方案时，才有可能用到此值。详细见 FreeRTOS 内存管理：http://www.openmcu.net/post/FreeRTOS_MemoryManagement.html。

configMAX_TASK_NAME_LEN

调用任务函数时，需要设置描述任务信息的字符串，这个宏用来定义该字符串的最大长度。这里定义的长度包括字符串结束符'\0'。

configUSE_TRACE_FACILITY

设置成 1 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。

configUSE_16_BIT_TICKS

定义 Tick 计数器的变量类型，即定义 portTickType 是表示 16 位变量还是 32 位变量。

定义 configUSE_16_BIT_TICKS 为 1 意味着 portTickType 代表 16 位无符号整形，定义 configUSE_16_BIT_TICKS 为 0 意味着 portTickType 代表 32 位无符号整形。

使用 16 位类型可以大大提高 8 位和 16 位架构微处理器的性能，但这也限制了最大时钟计数为 65535 个'Tick'。因此，如果 Tick 频率为 250HZ（4MS 中断一次），对于任务最大延时或阻塞时间，16 位计数器是 262 秒，而 32 位是 17179869 秒。

configIDLE_SHOULD_YIELD

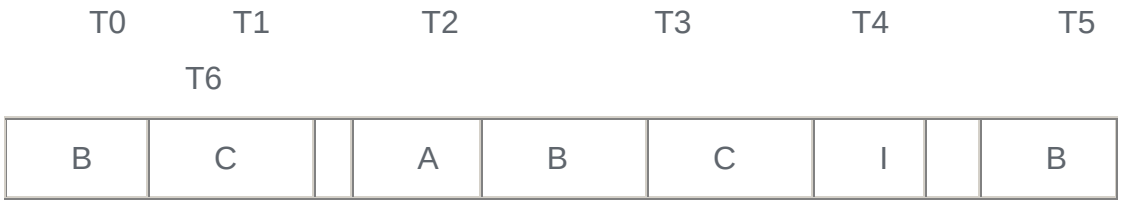
这个参数控制任务在空闲优先级中的特性。仅在满足下列条件后，才会起作用。

1.使用抢占式内核调度

2.空闲优先级中运行着用户任务。

通过时间片共享同一个优先级的多个任务，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。如果共享的优先级大于空闲优先级，那么更是这样。

但如果共享空闲优先级时，情况会稍微有些不同。当 configIDLE_SHOULD_YIELD 为 1 时，当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。当用户任务有效时，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。

可以通过下面方法避免：

l 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；

l 创建的用户任务优先级大于空闲优先级；

l 设置 IDLE_SHOULD_YIELD 为 0；

设置 configIDLE_SHOULD_YIELD 为 0 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。

configUSE_MUTEXES

设置为 1 表示使用互斥量，设置成 0 表示忽略互斥量。读者应该了解在 FreeRTOS 中互斥量和二进制信号量的区别。

关于互斥量和二进制信号量简单说：

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

configUSE_RECURSIVE_MUTEXES

设置成 1 表示使用递归互斥量，设置成 0 表示不使用。

configUSE_COUNTING_SEMAPHORES

设置成 1 表示使用计数信号量，设置成 0 表示不使用。

configUSE_ALTERNATIVE_API

设置成 1 表示使用“替代”队列函数（'alternative' queue functions），设置成 0 不使用。替代 API 在 queue.h 中有详细描述。

configCHECK_FOR_STACK_OVERFLOW

详见堆栈溢出检测部分。网址：http://www.openmcu.net/post/stack-overflow-checking.html

configQUEUE_REGISTRY_SIZE

队列记录有两个目的，都涉及到 RTOS 内核的调试：

1.它允许在调试 GUI 中使用一个队列的文本名称来简单识别队列；

2.包含调试器需要的每一个记录队列和信号量定位信息；

除了进行内核调试外，队列记录没有其它任何目的。

configQUEUE_REGISTRY_SIZE 定义可以记录的队列和信号量的最大数目。如果你想使用 RTOS 内核调试器查看队列和信号量信息，则必须先将这些队列和信号量进行注册，只有注册后的队列和信号量才可以使用 RTOS 内核调试器查看。查看 API 参考手册中的 vQueueAddToRegistry() 和 vQueueUnregisterQueue()函数获取更多信息。

configGENERATE_RUN_TIME_STATS

参见任务运行时间统计，网址： http://www.openmcu.net/post/run-time-stats.html

configUSE_CO_ROUTINES

设置成 1 表示使用协程，0 表示不使用协程。如果使用协程，必须在工程中包含 croutine.c 文件。

configMAX_CO_ROUTINE_PRIORITIES

应用程序协程（Co-routines）的有效优先级数目，任何数目的协程都可以共享一个优先级。使用协程可以单独的分配给任务优先级。见 configMAX_PRIORITIES。

注：关于协程

协程（Co-routines）主要用于资源发非常受限的嵌入式系统（RAM 非常少），通常不会用于 32 位微处理器。更多信息见：http://www.freertos.org/croutine.html

configUSE_TIMERS

设置成 1 使用软件定时器，为 0 不使用软件定时器功能。详细描述见 FreeRTOS software timers 。

configTIMER_TASK_PRIORITY

设置软件定时器服务/守护进程的优先级。详细描述见 FreeRTOS software timers 。

configTIMER_QUEUE_LENGTH

设置软件定时器命令队列的长度。详细描述见 FreeRTOS software timers。

configTIMER_TASK_STACK_DEPTH

设置软件定时器服务/守护进程任务的堆栈深度，详细描述见 FreeRTOS software timers 。

configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY

Cortex-M3、PIC24、dsPIC、PIC32、SuperH 和 RX600 硬件设备需要设置宏 configKERNEL_INTERRUPT_PRIORITY；PIC32、RX600 和 Cortex-M 硬件设备需要设置宏 configMAX_SYSCALL_INTERRUPT_PRIORITY。

Cortex-M3 和 Cortex-M4 用户要特别注意本节的结尾部分。

configKERNEL_INTERRUPT_PRIORITY 设置最低优先级。

注意下面的描述中，在中断服务例程中仅可以调用以“FromISR”结尾的 API 函数。

仅需要设置 configKERNEL_INTERRUPT_PRIORITY 的硬件设备：

configKERNEL_INTERRUPT_PRIORITY 用来设置 RTOS 内核自己的中断优先级。调用 API 函数的中断必须运行在这个优先级；不调用 API 函数的中断，可以运行在更高的优先级，所以这些中断不会被因 RTOS 内核活动而延时。

configKERNEL_INTERRUPT_PRIORITY 和 configMAX_SYSCALL_INTERRUPT_PRIORITY 都需要设置的硬件设备：

configKERNEL_INTERRUPT_PRIORITY 用来设置 RTOS 内核自己的中断优先级。configMAX_SYSCALL_INTERRUPT_PRIORITY 用来设置可以在中断服务程序中安全调用 FreeRTOS API 函数的最高中断优先级。

通过设置 configMAX_SYSCALL_INTERRUPT_PRIORITY 的优先级值大于（优先级值越大，优先级越高）configKERNEL_INTERRUPT_PRIORITY 可以实现完整的中断嵌套模式。这意味着 FreeRTOS 内核不能完全禁止中断，即使在临界区。此外，这对于分段内核架构的微处理器是有利的。请注意，当一个新中断发生后，某些微处理器架构会（在硬件上）禁止中断，这意味着从硬件响应中断到 FreeRTOS 重新使能中断之间的这段短时间内，中断是不可避免的被禁止的。

不调用 API 的中断可以运行在比 configMAX_SYSCALL_INTERRUPT_PRIORITY 高的优先级，因此不会因为执行 RTOS 内核而被延时。

例如：假如一个微控制器有 8 个中断优先级别：0 表示最低优先级，7 表示最高优先级（Cortex-M3 和 Cortex-M4 内核优先数和优先级别正好与之相反，见本章最后部分）。当两个配置选项分别为 4 和 0 时，下图描述了每一个优先级别可以和不可做的事件：

configMAX_SYSCALL_INTERRUPT_PRIORITY=4

configKERNEL_INTERRUPT_PRIORITY=0

这些配置参数允许非常灵活的中断处理：

l 在系统中可以像其它任务一样为中断处理任务分配优先级。这些任务通过一个相应中断唤醒。中断服务例程（ISR）内容应尽可能的精简---仅用于更新数据然后唤醒高优先级任务。ISR 退出后，直接运行被唤醒的任务，因此中断处理（根据中断获取的数据来进行的相应处理）在时间上是连续的，就像 ISR 在完成这些工作。这样做的好处是当中断处理任务执行时，所有中断都可以处在使能状态。

中断、中断服务例程（ISR）和中断处理任务是三码事：当中断来临时会进入中断服务例程，中断服务例程做必要的数据收集（更新），之后唤醒高优先级任务。这个高优先级任务在中断服务例程结束后立即执行，它可能是其它任务也可能是中断处理任务，如果是中断处理任务，那么就可以根据中断服务例程中收集的数据做相应处理。

l configMAX_SYSCALL_INTERRUPT_PRIORITY 接口有着更深一层的意义：在优先级介于 RTOS 内核中断优先级（等于 configKERNEL_INTERRUPT_PRIORITY）和 configMAX_SYSCALL_INTERRUPT_PRIORITY 之间的中断允许全嵌套中断模式并允许调用 API 函数。大于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的中断优先级绝不会因为执行 RTOS 内核而延时。

l 运行在大于 configMAX_SYSCALL_INTERRUPT_PRIORITY 的优先级中断是不会被 RTOS 内核所屏蔽的，因此也不受 RTOS 内核功能影响。这主要用于非常高的实时需求中。比如执行电机转向。但是，这类中断的中断服务例程中绝不可以调用 FreeRTOS 的 API 函数。

为了使用这个方案，应用程序要必须符合以下规则：调用 FreeRTOS API 函数的任何中断，都必须和 RTOS 内核处于同一优先级（由宏 configKERNEL_INTERRUPT_PRIORITY 设置），或者小于等于宏 configMAX_SYSCALL_INTERRUPT_PRIORITY 定义的优先级。

Cotex-M3 和 Cortex-M4 用户需要特别注意：请阅读 Cortex-M 设备中断优先级设置页面，英文网址：http://www.freertos.org/RTOS-Cortex-M3-M4.html。至少要记住：Cortex-M3 内核使用低的优先级数字表示高级别的优先级中断（比如某款 M3 内核有 8 级优先级，则优先级数字为 0 代表最高级别优先级，数字为 7 的代表最低级别优先级。与此不同的是，FreeRTOS 中优先级数字越大，表示优先级别越高，数字越低代表优先级别越低。这和 M3 内核表达优先级别是正好相反的，所以 FreeRTOS 的作者在这里强调一下），这看上去是违反直觉的，并且容易被忽视！在 M3、M4 内核中，如果你想分配给某个中断一个低优先级，则绝不可以分配给它们优先级 0（或者其它更低优先级数字），不然这样的中断实际上是系统中的最高级别优先级，如果这个优先级别大于 configMAX_SYSCALL_INTERRUPT_PRIORITY，则可能引起系统崩溃。

Cortex-M3 内核中最低优先级实际上是 255，但是不同厂商提供的 Cortex-M3 内核微处理器可能使用不同的优先级位数（关于优先级配置寄存器结构参见《The Definitive Guide to the ARM Cortex-M3》，也有中文译本《ARM Cortex-M3 权威指南》），比如，STM32 最低优先级实际上是 15，最高优先级是 0.

configASSERT

断言，调试时可以检查传入的参数是否合法。FreeRTOS 内核代码的关键点都会调用 configASSERT(x)函数，如果参数 x 为 0，则会抛出一个错误。这个错误很可能是传递给 FreeRTOS API 函数的无效参数引起的。定义 configASSERT()有助于调试时发现错误，但是，定义 configASSERT()也会增大应用程序代码量，增大运行时间。

configASSERT()等同于标准 C 库中的 assert()宏。不使用标准 C 库中的 assert()宏是因为不是所有的编译器都能够编译 FreeRTOS 提供的 assert.h 头文件。

INCLUDE Parameters

以“INCLUDE”起始的宏允许用户不编译那些应用程序不需要的实时内核组件（函数），这可以确保在你的嵌入式系统中 RTOS 占用最少的 ROM 和 RAM。

每个宏以这样的形式出现：

INCLUDE_FunctionName

在这里 FunctionName 表示一个你可以控制是否编译的 API 函数。如果你想使用该函数，就将这个宏设置成 1，如果不想使用，就将这个宏设置成 0。比如，对与

API 函数 vTaskDelete()：

#define INCLUDE_vTaskDelete 1

表示希望使用 vTaskDelete()，允许编译器编译该函数

#define INCLUDE_vTaskDelete 0

表示禁止编译器编译该函数。