Freertos中的List是双向循环链表

但是， 在多任务系统中，每个任务都是独立的，互不干扰的，所以要为每个任务都分  
配独立的栈空间，这个栈空间通常是一个预先定义好的全局数组， 也可以是动态分配的一  
段内存空间，但它们都存在于 RAM 中。

系统为了顺利的调度任务，为每个任务都额外定义了一个任务控制块，  
这个任务控制块就相当于任务的身份证，里面存有任务的所有信息，比如任务的栈指针，  
任务名称， 任务的形参等。有了这个任务控制块之后，以后系统对任务的全部操作都可以  
通过这个任务控制块来实现。TCB

1 typedef struct tskTaskControlBlock  
2 {  
3 volatile StackType\_t \*pxTopOfStack; /\* 栈顶 \*/ (1)  
4  
5 ListItem\_t xStateListItem; /\* 任务节点 \*/ (2)  
6  
7 StackType\_t \*pxStack; /\* 任务栈起始地址 \*/ (3)  
8 /\* 任务名称，字符串形式 \*/(4)  
9 char pcTaskName[ configMAX\_TASK\_NAME\_LEN ];  
10 } tskTCB;

栈顶指针，作为 TCB 的第一个成员

任务节点，这是一个内置在 TCB 控制块中的链表节点，通过这个  
节点，可以将任务控制块挂接到各种链表中。

任务栈起始地址。

任务堆栈大小，单位为字！！！！！！

FreeRTOS 中，任务的创建有两种方法，一种是使用动态创建，一  
种是使用静态创建。 动态创建时，任务控制块和栈的内存是创建任务时动态分配的， 任务  
删除时，内存可以释放。 静态创建时，任务控制块和栈的内存需要事先定义好，是静态的  
内 存 ， 任 务 删 除 时 ， 内 存 不 能 释 放 。

配置 PendSV 和 SysTick 的中断优先级为最低。 SysTick 和  
PendSV 都会涉及到系统调度，系统调度的优先级要低于系统的其它硬件中断优先级， 即优  
先相应系统中的外部硬件中断， 所以 SysTick 和 PendSV 的中断优先级配置为最低。

prvStartFirstTask()函数用于开始第一个任务，主要做了两个动作，一个是更新 MSP 的  
值，二是产生 SVC 系统调用，然后去到 SVC 的中断服务函数里面真正切换到第一个任务。

\_\_asm void prvStartFirstTask( void )  
9 {  
10 PRESERVE8 **(2)**  
11  
12 /\* 在 Cortex-M 中， 0xE000ED08 是 SCB\_VTOR 这个寄存器的地址， **(3)**  
13 里面存放的是向量表的起始地址，即 MSP 的地址 \*/  
14 ldr r0, =0xE000ED08 **(4)**  
15 ldr r0, [r0] **(5)**  
16 ldr r0, [r0] **(6)**  
17  
18 /\* 设置主堆栈指针 msp 的值 \*/  
19 msr msp, r0 **(7)**  
20  
21 /\* 使能全局中断 \*/ **(8)**  
22 cpsie i  
23 cpsie f  
24 dsb  
25 isb  
26  
27 /\* 调用 SVC 去启动第一个任务 \*/  
28 svc 0 **(9)**  
29 nop  
30 nop  
31 }

1 #define xPortPendSVHandler PendSV\_Handler  
2 #define xPortSysTickHandler SysTick\_Handler  
**3 #define vPortSVCHandler SVC\_Handler**

vPortSVCHandler()函数开始真正启动第一个任务，不再返回，

xPortPendSVHandler中断服务函数是真正实现任务切换的地方

（在整个系统中，中断使用的是主堆栈，  
栈指针使用的是 MSP），因为接下来要调用函数 vTaskSwitchContext，调用函数时，返回  
地址自动保存到 R14 中，所以一旦调用发生， R14 的值会被覆盖（ PendSV 中断服务函数执  
行完毕后，返回的时候需要根据 R14 的值来决定返回处理器模式还是任务模式，出栈时使  
用的是 PSP 还是 MSP）

/\* 带返回值的关中断函数，可以嵌套，可以在中断里面使用 \*/ **(2)**  
16 #define portSET\_INTERRUPT\_MASK\_FROM\_ISR() ulPortRaiseBASEPRI()  
17 ulPortRaiseBASEPRI( void )  
18 {  
19 uint32\_t ulReturn, ulNewBASEPRI = configMAX\_SYSCALL\_INTERRUPT\_PRIORITY; **(2)-**①  
20 \_\_asm  
21 {  
22 mrs ulReturn, basepri **(2)-**②  
23 msr basepri, ulNewBASEPRI **(2)-**③  
24 dsb  
25 isb  
26 }  
27 return ulReturn; **(2)-**④  
28 }

中断里的关中断函数和任务里的关中断函数的用途？？？？

操作系统里面的最小的时间单位就是  
SysTick 的中断周期，我们称之为一个 tick， SysTick 中断服务函数在 port.c.c 中实现，

void xPortSysTickHandler( void )  
2 {  
3 /\* 关中断 \*/  
4 vPortRaiseBASEPRI(); **(1)**  
5  
6 /\* 更新系统时基 \*/  
7 xTaskIncrementTick(); **(2)**  
8  
9 /\* 开中断 \*/

查 找 最 高 优 先 级 的 就 绪 任 务 有 两 种 方 法 ， 具 体 由  
configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION 这个宏控制， 定义为 0 选择通用方法，  
定义为 1 选择根据处理器优化的方法，该宏默认在 portmacro.h 中定义为 1，即使用优化过  
的方法

任务延时列表表维护着一条双向链表，每个节点代表了正在延时的任务，节点按照延  
时时间大小做升序排列。 当每次时基中断（ SysTick 中断） 来临时， 就拿系统时基计数器的  
值 xTickCount 与下一个任务的解锁时刻变量 xNextTaskUnblockTime 的值相比较， 如果相  
等， 则表示有任务延时到期， 需要将该任务就绪， 否则只是单纯地更新系统时基计数器  
xTickCount 的值， 然后进行任务切换。

FreeRTOS 定义了两个任务延时列表，当系统时基计数器  
xTickCount 没有溢出时，用一条列表，当 xTickCount 溢出后， 用另外一条列表。

延时的到期时间比xTickCount 最大值还大时，溢出。则将到期时间插入溢出列表

在 RTOS 中，最小的时间单位为一个 tick，即 SysTick 的中断周期，  
RT-Thread 和 μC/OS 可以指定时间片的大小为多个 tick，但是 FreeRTOS 不一样，时间片只  
能是一个 tick。 与其说 FreeRTOS 支持时间片，倒不如说它的时间片就是正常的任务调度。

listGET\_OWNER\_OF\_NEXT\_ENTRY()函数的妙处在于它并不是获取链表下的第一个  
节点的 OWNER，而且用于获取下一个节点的 OWNER。有下一个那么就会有上一个的说  
法，怎么理解？假设当前链表有 N 个节点，当第 N 次调用该函数时， pxIndex 则指向第 N  
个节点， 即每调用一次， 节点遍历指针 pxIndex 则会向后移动一次，用于指向下一个节点。

（实现了相同优先级的任务，时间片轮转调度）

taskRESET\_READY\_PRIORITY() 函 数 的 妙 处 在 于 清 除 优 先 级 位 图 表  
uxTopReadyPriority 中相应的位时候，会先判断当前优先级链表下是否还有其它任务，如果  
有则不清零。 假设当前实验中，任务 1 会调用 vTaskDelay()，会将自己挂起，只能是将任  
务 1 从就绪列表删除，不能将任务 1 在优先级位图表 uxTopReadyPriority 中对应的位清 0，  
因为该优先级下还有任务 2，否则任务 2 将得不到执行

任务实体通常是一个不带返回值的无限循环的 C 函数，函数  
体必须有阻塞的情况出现，不然任务（如果优先权恰好是最高）会一直在 while 循环里面  
执行，导致其它任务没有执行的机会。

所以freertos中是存在进程饿死的情况的，一定要避免

Freertos中，任务的删除，可以在任务函数的结尾，调用任务删除函数RTOS\_TaskDelete(函数指针);

在未初始化内存的时候一旦调用了xTaskCreate()函数， FreeRTOS 就会帮我们自动进行内存的初始化，

调用函数 xPortStartScheduler()来启动系统节拍定时器（一般都是  
使用 SysTick） 并启动第一个任务。

SysTick 用于产生系统节拍时钟，提供一个时间片，如果多个任务共享同一个优先级，  
则每次 SysTick 中断，下一个任务将获得一个时间片。

有人可能会问，那 SysTick 的优先级配置为最低，那延迟的话系统时间会不会有偏差？答案是不会的，因为 SysTick 只是当次响应中断被延迟了，而 SysTick 是硬件定时器，它一直在计时，  
这一次的溢出产生中断与下一次的溢出产生中断的时间间隔是一样的，至于系统是否响应  
还是延迟响应， 这个与 SysTick 无关，它照样在计时。

但是处理还是要延迟一到多个中断周期的。所有还是有轻微的影响的。

FreeRTOS 中的任务是抢占式调度机制，高优先级的任务可打断低优先级任务，低优先  
级任务必须在高优先级任务阻塞或结束后才能得到调度。 同时 FreeRTOS 也支持时间片轮  
转调度方式，只不过时间片的调度是不允许抢占任务的 CPU 使用权。

在系统中，当有比当前任务优先  
级更高的任务就绪时，当前任务将立刻被换出，高优先级任务抢占处理器运行。

任务调度的原则是一旦任务状态发生了改变，并且当前运行的任务优  
先级小于优先级队列组中任务最高优先级时，立刻进行任务切换（除非当前系统处于中断  
处理程序中或禁止任务切换的状态）。

调度器恢复可  
以调用 xTaskResumeAll()函数，调用了多少次的 vTaskSuspendAll()就要调用多少次  
xTaskResumeAll()进行恢复，

xTaskResumeFromISR() 专 门 用 在 中 断 服 务 程 序 中 。 无 论 通 过 调 用 一 次 或 多 次  
vTaskSuspend()函数而被挂起的任务，也只需调用一次 xTaskResumeFromISR()函数即可解  
挂 。

绝对延时函数 vTaskDelayUntil()，这个绝  
对延时常用于较精确的周期运行任务，比如我有一个任务， 希望它以固定频率定期执行，  
而不受外部的影响，任务从上一次运行开始到下一次运行开始的时间间隔是绝对的，而不  
是相对的，

vTaskDelayUntil() 与 vTaskDelay () 一 样 都 是 用 来 实 现 任 务 的 周 期 性 延 时 。 但  
vTaskDelay ()的延时是相对的，是不确定的，它的延时是等 vTaskDelay ()调用完毕后开始  
计算的。 并且 vTaskDelay ()延时的时间到了之后，如果有高优先级的任务或者中断正在执  
行，被延时阻塞的任务并不会马上解除阻塞，所有每次执行任务的周期并不完全确定。而  
vTaskDelayUntil()延时是绝对的，适用于周期性执行的任务。 当(\*pxPreviousWakeTime +  
xTimeIncrement)时间到达后， vTaskDelayUntil()函数立刻返回，如果任务是最高优先级的，  
那么任务会立马解除阻塞， 所以说 vTaskDelayUntil()

断服务函数是一种需要特别注意的上下文环境，它运行在非任务的执行环境下（一  
般为芯片的一种特殊运行模式（也被称作特权模式）），在这个上下文环境中不能使用挂  
起当前任务的操作，不允许调用任何会阻塞运行的 API 函数接口

空闲任务是  
一个非常短小的循环。 用户可以通过空闲任务钩子方式，在空闲任务上钩入自己的功能函  
数。通常这个空闲任务钩子能够完成一些额外的特殊功能，例如系统运行状态的指示，系  
统省电模式等。

对于空闲任务钩子上挂接的空闲钩子函数，它应该满足以下的条件：  
 永远不会挂起空闲任务；  
 不应该陷入死循环，需要留出部分时间用于系统处理系统资源回收。  
4. 任务的执行时间：