FreeRTOS 使用指南

繁星电子开发团队制作

作为一个轻量级的操作系统，FreeRTOS 提供的功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能等，可基本满足较小系统的需要。FreeRTOS 内核支持优先级调度算法，每个任务可根据重要程度的不同被赋予一定的优先级，CPU 总是让处于就绪态的、优先级最高的任务先运行。FreeRT0S 内核同时支持轮换调度算法，系统允许不同的任务使用相同的优先级，在没有更高优先级任务就绪的情况下，同一优先级的任务共享 CPU 的使用时间。

FreeRTOS 的内核可根据用户需要设置为可剥夺型内核或不可剥夺型内核。当 FreeRTOS 被设置为可剥夺型内核时，处于就绪态的高优先级任务能剥夺低优先级任务的 CPU 使用权，这样可保证系统满足实时性的要求；当 FreeRTOS 被设置为不可剥夺型内核时，处于就绪态的高优先级任务只有等当前运行任务主动释放 CPU 的使用权后才能获得运行，这样可提高 CPU 的运行效率

FreeRTOS 对系统任务的数量没有限制。

一 变量类型定义

```
#define portCHAR          char
#define portFLOAT         float
#define portDOUBLE        double
#define portLONG          long
#define portSHORT         short
#define portSTACK_TYPE    unsigned portLONG
#define portBASE_TYPE     long
```

二 任务函数

- **任务创建**

头文件：task.h

portBASE_TYPE **xTaskCreate (**

pdTASK_CODE **pvTaskCode**, 指向任务的实现函数的指针。效果上仅仅是函数名

const portCHAR * **const pcNane**, 具有描述性的任务名。FreeRTOS 不会使用它。

unsigned portSHORT **usStackDepth**, 指定任务堆栈的大小

void ***pvParameters**, 指针用于作为一个参数传向创建的任务

unsigned **portBASE_TYPE uxPriority,** 任务运行时的优先级

xTaskHandle ***pvCreatedTask** 用于传递任务的句柄，可以引用从而对任务进行其他操作。

)

说明：

1. 这里的任务是指一个永远不会退出的 C 函数，通常是一个死循环。
2. **pcNane** 其只是单纯地用于辅助调试。应用程序可以通过定义常量 config_MAX_TASK_NAME_LEN 来定义任务名的最大长度——包括’\0’结束符。如果传入的字符串长度超过了这个最大值，字符串将会自动被截断
3. **usStackDepth** 这个值指定的是栈空间可以保存多少个字(word)，而不是多少个字节(byte)。栈空间大小为 usStackDepth*4(bytes)。
4. **uxPriority** 优先级的取值范围可以从最低优先级 0 到最高优先级(configMAX_PRIORITIES－1)。

返回:

1. pdPASS 表明任务创建成功，准备运行。
2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY 由于内存堆空间不足，FreeRTOS 无法分配足够的空间来保存任务结构数据和任务栈，因此无法创建任务。

- **任务删除**

头文件：task.h

void **vTaskDelete (**

xTaskHandle **pxTask** 处理要删除的任务。传递 NULL 将删除自己

)

说明：

1. **FreeRTOSConfig.h 中的** INCLUDE_vTaskDelete=1，这个函数才能用。从 RTOS 实时内核管理中移除任务。要删除的任务将从就绪，封锁，挂起，事件列表中移除。
2. 任务被删除后就不复存在，也不会再进入运行态
3. 空闲任务负责释放内核分配给已删除任务的内存。

使用提示：只有内核为任务分配的内存空间才会在任务被删除后由空闲任务自动回收。任务自己占用的内存或资源需要由应用程序自己显式地释放

- **任务延时**

头文件：task.h

void vTaskDelay **(**

portTickType **xTicksToDelay** 时间数量，调用任务应该锁住的时间片周期

)

说明：

1. **FreeRTOSConfig.h 中的** INCLUDE_vTaskDelay=1，这个函数才能用。
2. 延时任务为已知**时间片**，任务被锁住剩余的实际时间由时间片速率决定。portTICK_RATE_MS 常量以时间片速率来计算实际时间
3. vTaskDelay()指定一个任务希望的时间段，这个时间之后任务解锁。
4. vTaskDelay()不提供一个控制周期性任务频率的好方法，和其他任务和中断一样，在调用 vTaskDelay()后将影响频率

提示：vTaskDelayUntil()，这个交替的 API 函数设计了执行固定的频率。它是指定的一个绝对时间（而不是一个相对时间）后，调用任务解锁。--------可以实现周期性任务执行。

- **任务延迟到指定时间**

头文件：task.h

void **vTaskDelayUntil (**

portTickType ***pxPreviousWakeTime**，指定一个变量来掌握任务最后开启的时间，第一次使用时必须使用当前时间来初始化，在 vTaskDelayUntil 中，这个变量是自动修改的

portTickType **xTimeIncrement** 循环周期时间

)

说明：

1. **FreeRTOSConfig.h 中的** INCLUDE_vTaskDelayUntil=1,这个函数才能用。
2. 延时一个任务到指定时间，这个和 vTaskDelay()不同，vTaskDelay 是延时一个相对时间，而 vTaskDelayUntil 是延时一个绝对时间
3. 常量 portTICK_RATE_MS 用来计算时间片频率的实时时间- 按照一个时间片周期
4. 任务将在一定时间开启（*pxPreviousWakeTime + xTimeIncrement）。使用相同的 xTimeIncrement 参数值来调用 vTaskDelayUntil()将使任务按固定的周期执行。

注意：vTaskDelayUntil() 如果指定的苏醒时间使用完，将立即返回。因此，一个使用 vTaskDelayUntil() 来周期性的执行的任务，如果执行周期因为任何原因（例如任务是临时为悬挂状态）暂停而导致任务错过一个或多个执行周期，那么需要重新计算苏醒时间。通过检查像 pxPreviousWakeTime 可变的参数来组织当前时间片计数。然而在大多数使用中并不是必须的。

使用提示：如果一个任务想按固定的频率运行，如让一个 LED 灯，按 1KHz 频率运行，如果只有一个任务那么调用 vTaskDelay 或 vTaskDelayUntil 都能完成，但是如果有多个任务，vTaskDelay 就不行了，因为优先级或其它问题，它不知道什么时候再能运行，因此其不是周期执行，也就谈不上固定频率了，这时就要用 vTaskDelayUntil 这个函数了

- **获得任务优先级**

头文件：task.h

unsigned portBASE_TYPE **uxTaskPriorityGet (**

xTaskHandle **pxTask** 需要处理的任务. 当传递 NULL 时，将返回调用该任务的优先级

)

说明：FreeRTOSConfig.h 中的 INCLUDE_vTaskPriorityGet=1,这个函数才能用

返回：pxTask 的优先级

- **设置任务优先级**

头文件：task.h

void **vTaskPrioritySet (**

xTaskHandle **pxTask ,** 需要设置优先级的任务。当传递 NULL，将设置调用任务的优先级

unsigned portBASE_TYPE **uxNewPriority** 任务需要设置的优先级

)

说明：FreeRTOSConfig.h 中的 INCLUDE_vTaskPrioritySet 为 1，才能使用此函数.如果设置的优先级高于当前执行任务的优先级，则上下文切换将在此函数返回前发生

- **挂起任务**

头文件：task.h

void vTaskSuspend **(**

xTaskHandle **pxTaskToSuspend** 处理需要挂起的任务。传递 NULL 将挂起调用此函数的任务

)

说明：FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 为 1，才能使用此函数。当挂起一个任务时，不管优先级是多少，不需要占用任何微控制器处理器时间。调用 vTaskSuspend 不会累积——即：在同一任务中调用 vTaskSuspend 两次，但只需要调用一次 vTaskResume()就能使挂起的任务就绪

- **唤醒挂起的任务**

头文件：task.h

void vTaskResume **(**

xTaskHandle **pxTaskToResume** 就绪任务的句柄

)

说明：FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspend 为 1，才能使用此函数。必须是调用 vTaskSuspend () 后挂起的任务，才有可能通过调用 vTaskResume ()从新运行

- **从中断唤醒挂起的任务**

头文件：task.h

portBase_TYPE **vTaskResumeFromISR (**

xTaskHandle **pxTaskToResum** 就绪任务的句柄

)

说明：FreeRTOSConfig.h 中的 INCLUDE_vTaskSuspendhe 和 INCLUDE_xTaskResumeFromISR 都为 1，才能使用此函数。

vTaskResumeFromISR()不应该用于任务和中断同步，因为可能会在中断发生期间，任务已经挂起——这样导致错过中断。使用信号量最为同步机制将避免这种偶然性。

返回：pdTRUE： 如果唤醒了任务将引起上下文切换。pdFALSE：用于 ISR 确定是否上下文切换

- **为任务分配标签值**

头文件：task.h

void **vTaskSetApplicationTaskTag (**

xTaskHandle **xTask ,** 将分配给标签值的任务。传递 NULL 将分配标签给调用的任务。

pdTASK_HOOK_CODE **pxTagValue** 分配给任务的标签值 类型为 pdTASK_HOOK_CODE 允许一个函数指针赋值给标签，因此实际上任何值都可以分配

)

说明：FreeRTOSConfig.h 中的 onfigUSE_APPLICATION_TASK_TAG 为 1，这个函数才能可用。这个值仅在应用程序中使用，内核本身不使用它。

- xTaskCallApplicationTaskHook

三：内核函数

- **启动实时内核处理**

头文件：task.h

void **vTaskStartScheduler (void);**

说明： 当 vTaskStartScheduler() 被调用时，空闲任务自动创建。如果 vTaskStartScheduler() 成功调用，这个函数不返回，直到执行任务调用 vTaskEndScheduler()。如果可供给空闲任务的 RAM 不足，那么函数调用失败，并立即返回。

- **停止实时内核运行**

头文件：task.h

void **vTaskEndScheduler (void);**

说明： 所有创建的任务将自动删除，并且多任务（优先级或合作式）将停止。当 vTaskStartScheduler()调用时，执行将再次开始，像 vTaskStartScheduler()仅仅返回。

注意： vPortEndScheduler ()导致所有由内核分配的资源释放——但是不会释放由应用程序的任务分配的资源。

- **挂起所有活动的实时内核，同时允许中断（包括内核滴答）**

头文件：task.h

void vTaskSuspendAll **(void);**

说明： 任务在调用 vTaskSuspendAll ()后，这个任务将继续执行，不会有任何被切换的危险，直到调用 xTaskResumeAll ()函数重启内核。

注意： API 中有可能影响影响上下文切换的函数（例如，vTaskDelayUntil(), xQueueSend()等等），一定不能在调度器挂起时被调用。

四：队列管理

队列是内部通信的主要形式。它可以用于在任务和任务之间以及任务和中断之间发送消息。在大多数情况下使用线程安全 FIFO（先进先出）缓存，新数据放在队列的最后，虽然数据也可以放在前面。
队列可以包含固定大小的 '项目' - 每个项目的大小和队列可以保存项目的最大数量在创建队列时就已经定义了。

项目以复制而不是引用的方式放入队列，因此最好使放入队列项目的大小成为最小。以复制的方式放入队列可以使你的系统设计极大的简化，因为两个任务不会同时访问数据。队列帮助你管理所有的互斥问题。

如果你希望在队列中使用大的项目，可能最好用插入队列指针 - 但是这样做必须注意要确保你的系统明确定义任务和/或中断是数据的"所以者"。

队列 API 函数可以指定阻塞的时间。阻塞时间代表任务进入阻塞状态或者等待队列中数据时（当任务读取队列但是队列是空的时候）的最大'节拍'数，或者等待队列空间变为可以使用（当任务需要写数据到队列，但是队列已满时）。当一个以上任务在同一个队列中被阻塞时，高优先级的任务先解除阻塞。

- **创建一个新的队列**

头文件：queue. H

```
xQueueHandle xQueueCreate (
                unsigned portBASE_TYPE uxQueueLength, 队列中包含最大项目数量
                unsigned portBASE_TYPE uxItemSize 队列中每个项目所需的字节数
              );
```

说明：创建一个新的队列。为新的队列分配所需的存储内存，并返回一个队列处理。

注意：项目通过复制而不是引用排队，因此，所需的字节数，将复制给每个项目。队列中每个项目必须分配同样大小。

返回：如果队列成功创建，则返回一个新建队列的处理。如果不能创建队列，将返回 0。

- **传递一个项目到队列**

头文件：queue. H

```
portBASE_TYPE xQueueSend (
                xQueueHandle xQueue, 将项目传进的队列
                const void * pvItemToQueue, 项目的指针【源数据】
                portTickType xTicksToWait 等待的最大时间量【时间使用滴答周期】
              );
```

说明：传递一个项目到队列。这个项目通过复制而不是通过引用排队。这个函数不能从中断服务程序调用。

注意：当队列满时，肯定传递不成功，则等待 xTicksToWait 个滴答周期后再传递，但如果 xTicksToWait 设置为 0，调用将立即返回。

返回：pdTRUE：项目成功传递。否则为：errQUEUE_FULL.

- **传递项目到一个队列中的后面**

头文件：queue. H

```
portBASE_TYPE xQueueSendToBack (
                xQueueHandle xQueue, 将项目传进的队列
                const void * pvItemToQueue, 项目的指针【源数据】
                portTickType xTicksToWait 等待的最大时间量
              );
```

说明：这个与 xQueueSend 是一样的，参照 xQueueSend 的用法

- **从队列接收一个项目**

头文件：queue. H

portBASE_TYPE **xQueueReceive (**
xQueueHandle **xQueue,** 发送项目的队列句柄
void ***pvBuffer**, 指向缓冲区的指针，将接收的项目被复制进去
portTickType **xTicksToWait** 任务中断并等待队列中可用空间的最大时间
);

说明：这个项目通过复制接收，因此缓冲器必须提供足够大的空间。这个函数一定不能在中断服务程序中使用当队列空时，肯定复制传递不成功，则等待 xTicksToWait 个滴答周期后再复制，但如果 xTicksToWait 设置为 0，调用将立即返回。

返回：如果项目成功被队列接收为 pdTRUE ，否则为 pdFALSE。

- **从中断传递一个项目到队列的后面**

头文件：queue. H

portBASE_TYPE **xQueueSendFromISR (**
xQueueHandle **pxQueue**, 将项目传进的队列
const void ***pvItemToQueue**, 项目的指针【源数据】
portBASE_TYPE ***pxHigherPriorityTaskWoken** 因空间数据问题被挂起的任务是否解锁
);

说明：如果传进队列而导致因空间数据问题被挂起的任务解锁，并且解锁的任务的优先级高于当前运行任务，xQueueSendFromISR 将设置 *pxHigherPriorityTaskWoken 到 pdTRUE。当 pxHigherPriorityTaskWoken 被设置为 pdTRUE 时，则在中断退出之前将请求任务切换。

返回：pdTRUE：数据成功传递进队列。否则为：errQUEUE_FULL。

- **从中断传递项目到一个队列中的后面**

头文件：queue. H

portBASE_TYPE **xQueueSendToBackFromISR (**
xQueueHandle **pxQueue**,
const void ***pvItemToQueue,**
portBASE_TYPE ***pxHigherPriorityTaskWoken**
);

说明：参照 xQueueSendFromISR

- **从中断传递一个项到队列的前面**

头文件：queue. H

portBASE_TYPE **xQueueSendToFrontFromISR (**
xQueueHandle **pxQueue**,
const void ***pvItemToQueue**,
portBASE_TYPE ***pxHigherPriorityTaskWoken**
);

说明：参照 xQueueSendFromISR

- **中断时从队列接收一个项目**

头文件：queue. H

portBASE_TYPE **xQueueReceiveFromISR (**

xQueueHandle **pxQueue**, 发送项目的队列句柄

void ***pvBuffer**, 指向缓冲区的指针，将接收的项目被复制进去

portBASE_TYPE ***pxTaskWoken** 任务将锁住，等待队列中的可用空间

);

说明： 如果 xQueueReceiveFromISR 引起一个任务解锁，*pxTaskWoken 将设置为 pdTRUE，否则*pxTaskWoken 保留不变

返回： pdTRUE ：如果项目成功从队列接收。否则为： pdFALSE

- **为队列命名，并加入队列到登记管理中**

头文件：queue.h

void vQueueAddToRegistry (

xQueueHandle **xQueue**, 将要添加登记的队列句柄

signed portCHAR ***pcQueueName**, 为指定的队列命名。 仅仅是文本串，方便调试。

);

说明： 队列登记有两个特点， 均与内核的相关调试有关：

允许文本式名字，使队列在调试 GUI 中方便定义。

包含调试器需要的信息，如：定位每个已经登记的队列和信号量。

队列的登记没有目的，除非使用内核相关的调试。

configQUEUE_REGISTRY_SIZE 定义了队列和信号量的最大数目.仅当使用内核相关的调试时需要显示已经登记的信号量和队列。

- **从登记管理中移除队列**

头文件：queue.h

void **vQueueUnregisterQueue** (

xQueueHandle **xQueue**, 从登记管理处中移出的队列句柄

);

说明： 队列登记有两个特点， 均与内核的相关调试有关：

允许文本式名字，使队列在调试 GUI 中方便定义。

包含调试器需要的信息，如：定位每个已经登记的队列和信号量。

队列的登记没有目的，除非使用内核相关的调试。

configQUEUE_REGISTRY_SIZE 定义了队列和信号量的最大数目.仅当使用内核相关的调试时需要显示已经登记的信号量和队列。

五：信号量

- **使用已存在的队列结构来创建计数型信号量**

头文件：semphr. H

xSemaphoreHandle **xSemaphoreCreateCounting** (

unsigned portBASE_TYPE **uxMaxCount**, 可以达到的最大计数值。

unsigned portBASE_TYPE **uxInitialCount** 信号量创建时分配的初始值

)

说明：两种典型应用

事件计数

在这种应用的情形下，事件处理程序会在每次事件发生时发送信号量（增加信号量计数值），而任务处理程序会在每次处理事件时请求信号量（减少信号量计数值）。因此计数值为事件发生与事件处理两者间的差值，在这种情况下计数值初始化为 0 是合适的。

资源管理

在这种应用情形下，计数值指示出可用的资源数量。任务必须首先“请求”信号量来获得资源的控制权--减少信号量计数值。当计数值降为 0 时表示没有空闲资源。任务使用完资源后“返还”信号量--增加信号量计数值。在这种情况下计数值初始化为与最大的计数值相一致是合适的，这指示出所有的空闲资源。

返回：已创建的信号量句柄，为 xSemaphoreHandle 类型，如果信号量无法创建则为 NULL。

- **使用已存在的队列结构来创建互斥锁信号量的宏**

头文件：semphr. H

xSemaphoreHandle **xSemaphoreCreateMutex** (void)

说明：通过此宏创建的互斥锁可以使用 xSemaphoreTake()与 xSemaphoreGive()宏来访问。不能使用 xSemaphoreTakeRecursive()与 xSemaphoreGiveRecursive()宏

二元信号量与互斥锁十分相像，不过两者间有细微的差别：互斥锁包含一个优先级继承机制，而信号量没有。这种差别使得二元信号量更适合于实现同步（任务之间或任务与中断之间），互斥锁更适合于实现简单的互斥。

当有另外一个具有更高优先级的任务试图获取同一个互斥锁时，已经获得互斥锁的任务的优先级会被提升。已经获得互斥锁的任务将继承试图获取同一互斥锁的任务的优先级。这意味着互斥锁必须总是要返还的，否则高优先级的任务将永远也不能获取互斥锁，而低优先级的任务将不会放弃优先级的继承。

二元信号量并不需要在得到后立即释放，因此任务同步可以通过一个任务/中断持续释放信号量而另外一个持续获得信号量来实现。

互斥锁与二元信号量均赋值为 xSemaphoreHandle 类型，并且可以在任何此类型参数的 API 函数中使用。

返回：已创建的信号量句柄，需要为 xSemaphoreHandle 类型。

- **使用已存在的队列结构来创建递归互斥锁的宏**

头文件：semphr. H

xSemaphoreHandle **xSemaphoreCreateRecursiveMutex** (void)

说明：通过此宏创建的互斥锁可以使用 xSemaphoreTakeRecursive()与 xSemaphoreGiveRecursive()宏 来访问。不能使用 xSemaphoreTake()与 xSemaphoreGive()宏

一个递归的互斥锁可以重复地被其所有者“获取”。在其所有者为每次的成功“获取”请求调用 xSemaphoreGiveRecursive()前，此互斥锁不会再次可用。例如，如果一个任务成功“获取”同一个互斥锁 5 次，则在其“释放”互斥锁恰好为 5 次后，其他任务才可以使用此互斥锁。

这种类型的信号量使用一个优先级继承机制，因此已取得信号量的任务“必须总是”在不再需要信号量时立刻“释放”。

互斥类型的信号量不能在中断服务程序中使用。

可以参考 vSemaphoreCreateBinary()来获得一个二选一运行的实现方式，可以在中断服务程序中实现纯粹的同步（一个任务或中断总是“释放”信号量，而另一个总是“获取”信号量）。

返回： 已创建的信号量句柄，需要为 xSemaphoreHandle 类型。

- **获取信号量的宏**

头文件：semphr. H

xSemaphoreTake (

xSemaphoreHandle **xSemaphore**, 将被获得的信号量句柄，此信号量必须已经被创建

portTickType **xBlockTime** 等待信号量可用的时钟滴答次数

)

说明： 当信号量不可用时，则等待 xBlockTime 个时钟滴答，再看是否可用，当是为 0 时如果不可用，则立即退出，因此为 0 时可以达到对信号量轮询的作用。

返回： 如果成功获取信号量则返回 pdTRUE，如果 xBlockTime 超时而信号量还未可用则返回 pdFALSE。

- **递归获得互斥锁信号量的宏**

头文件：semphr. H

xSemaphoreTakeRecursive (

xSemaphoreHandle **xMutex**, 将被获得的互斥锁句柄

portTickType **xBlockTime** 等待信号量可用的时钟滴答次数

)

说明： FreeRTOSConfig.h 中的 configUSE_RECURSIVE_MUTEXES 必须设置为 1 才可以使用此宏。

一个递归型的互斥锁可以被其所有者重复地“获取”， 在其所有者为每次成功的“获取”请求调用 xSemaphoreGiveRecursive()前，此互斥锁不会再次可用。

返回： 如果成功获取信号量则返回 pdTRUE，如果 xBlockTime 超时而信号量还未可用则返回 pdFALSE。

- **释放信号量的宏**

头文件：semphr. H

xSemaphoreGive (

xSemaphoreHandle **xSemaphore** 即将释放的信号量的句柄，在信号量创建是返回

)

返回： 如果信号量成功释放返回 pdTRUE，如果发生错误则返回 pdFALSE。信号量使用的是队列，因此如果队列没有位置用于发送消息就会发生一个错误——说明开始时没有正确获取信号量。

- **用于递归释放，或‘返还’，互斥锁信号量的宏**

头文件：semphr. H

xSemaphoreGiveRecursive (

xSemaphoreHandle **xMutex** 将被释放或‘返还’的互斥锁的句柄

)

返回： 如果信号量成功释放则为 pdTRUE

- **从中断释放一个信号量的宏**

头文件：semphr. H

xSemaphoreGiveFromISR (

xSemaphoreHandle **xSemaphore**，将被释放的信号量的句柄

portBASE_TYPE ***pxHigherPriorityTaskWoken** 因空间数据问题被挂起的任务是否解锁

)

返回：如果信号量成功释放则返回 pdTRUE，否则返回 errQUEUE_FULL

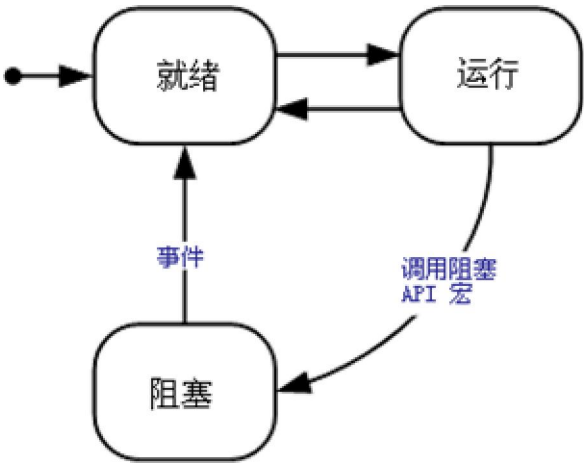
六：联合函数

一个联合程序可以以下面的状态中的一种存在:

运行：当一个联合程序正在执行时它就是处于运行状态，它就占用了处理器。

就绪：就绪状态的联合程序是那些可以执行（没有被阻塞）但是还没有被执行的程序。一个联合程序可能处于就绪状态是因为另外一个相同或高优先级的联合程序正处于运行状态，或一个任务处于运行状态——这只会在系统同时使用了任务和联合程序时发生。

阻塞：如果一个联合程序正处于暂时等待或等待外部事件，它就是处于阻塞状态。例如，联合程序调用 crDELAY() 它就会被阻塞（放入到阻塞状态）直到达到延时时间 - 一个临时事件。被阻塞的联合程序不会被调度。



有效的联合程序状态转换

联合程序属性：

每个联合程序被分配一个从 0 到 (configMAX_CO_ROUTINE_PRIORITIES - 1) 的优先级。configMAX_CO_ROUTINE_PRIORITIES 在 FreeRTOSConfig.h 中定义并在基本系统中设置。configMAX_CO_ROUTINE_PRIORITIES 的数值越大 FreeRTOS 消耗的 RAM 越多。

低优先级联合程序使用小数值。

联合程序优先级只针对其他联合程序，任务的优先级总是高于联合程序

- 创建一个新的联合程序并且将其增加到联合程序的就绪列表中

头文件：croutine.h

portBASE_TYPE **xCoRoutineCreate** (
crCOROUTINE_CODE **pxCoRoutineCode**, 联合程序函数的指针
unsigned portBASE_TYPE **uxPriority**, 优先级
unsigned portBASE_TYPE **uxIndex** 当不同的联合程序使用同一个函数来运行时用于相互识别
);

返回：如果联合程序成功创建并增加到就绪列表中则返回 pdPASS，否则返回 ProjDefs.h 中定义的错误代码

- 把一个联合程序延时一个特定的时间

头文件：croutine.h

void **crDELAY** (
xCoRoutineHandle **xHandle**，要延时的联合程序的句柄
portTickType **xTicksToDelay** 联合程序要延时的时间片数
)

- 联合程序向其他联合程序发送数据

头文件：croutine.h

crQUEUE_SEND (
xCoRoutineHandle **xHandle**，调用的联合程序的句柄
xQueueHandle **pxQueue**，数据将被发送到的队列的句柄
void ***pvItemToQueue**，将被发送到队列的数据的指针
portTickType **xTicksToWait**，如果此刻队列没有可用空间，此值为联合程序用于阻塞等待队列空间可用的时间片数。
portBASE_TYPE ***pxResult** 一个指向 pxResult 变量的指针，如果数据成功发送到队列就会被设置为 pdPASS，否则设置为 ProjDefs.h 中定义的错误码。
)

- 联合程序接受其他联合程序发送的数据

头文件：croutine.h

void **crQUEUE_RECEIVE**(
xCoRoutineHandle **xHandle**,
xQueueHandle **pxQueue**,
void ***pvBuffer**,
portTickType **xTicksToWait**,
portBASE_TYPE ***pxResult**
)

说明：同上

- 中断处理程序向联合程序发送数据

头文件：croutine.h

portBASE_TYPE **crQUEUE_SEND_FROM_ISR**(
xQueueHandle **pxQueue**，将发送到的队列的句柄
void ***pvItemToQueue**，将被发送到队列的条目的指针
portBASE_TYPE **xCoRoutinePreviouslyWoken**
)

- 联合程序向中断处理程序发送数据

头文件：croutine.h

portBASE_TYPE crQUEUE_SEND_FROM_ISR(
xQueueHandle pxQueue,
void *pvBuffer,
portBASE_TYPE * pxCoRoutineWoken
)

- 运行一个联合程序

头文件：croutine.h

void **vCoRoutineSchedule** **(** void **);**

说明：vCoRoutineSchedule() 与性具有最高优先级的就绪联合程序。此联合程序将运行，直到其阻塞，让出系统或被任务抢占。联合程序是合作式运行的，所以一个联合程序不会被另一个联合程序抢占，但是可以被任务抢占。

如果一个应用程序同时包含任务与联合程序，则 vCoRoutineSchedule 应该在空闲任务中调用（在一个空闲任务钩子中）

Version 0.0.1

2011-04-11