阿熊的FreeRTOS教程系列！

哈喽大家好！我是你们的老朋友阿熊！STM32教程系列更新完结已经有一段时间了，视频反馈还是不错的，从今天开始我们将会更新我们的FreeRTOS的教程

由于东西真的太多了，也纠结了很久要不要讲这个系列，毕竟难度真的很大，怕在难以做到那么通俗易懂，经过一段时间的考虑，还是决定好了给大家做一个入门级的讲解使用，由于FreeRTOS的内容真的很多，作为还是学生的我使用的也相对较少，操作系统层面的东西，我会用最大的能力去让大家理解，主要讲述主要功能，学完以后保证大伙可以理解80%以上的FreeRTOS的使用场景，好了废话不多说，开始我们的课程吧！

第四章：任务的状态

上节课我们提了一嘴FreeRTOS当中的“阻塞模式”，然后本节课将会将这个全部展开讲述一下我们任务的其他的状态，好了话不多说，我们直接开始吧

壹：状态讲解

总的来说我们的状态可以分为两种，一种是运行状态，一种是非运行状态

运行状态（**Runing**）

这个应该不需要过多的解释，它就是任务正在运行时候的这个状态被叫做运行状态

非运行状态（**Not Running**）

这个并不只是一种运行状态而是好几种状态的统称

详细的来说它可以分为三种状态分别是：

阻塞状态(Blocked)、暂停状态(Suspended)、就绪状态(Ready)

阻塞状态**(Blocked)**：

堵住了，需要等待

我们可以把执行任务当做一个那个行车的过程，我们的阻塞状态就是我们是进入了堵车的情况，然后把道路空出来，就给其它任务去执行了，这样的话我们的其他任务或者说低优先级的任务才有机会执行

暂停状态**(Suspended)**：

像这种状态一般就是将其手动设置为暂停状态，当他进入暂停状态之后就是真的暂停了，我们不解除暂停状态，他就永远不会执行

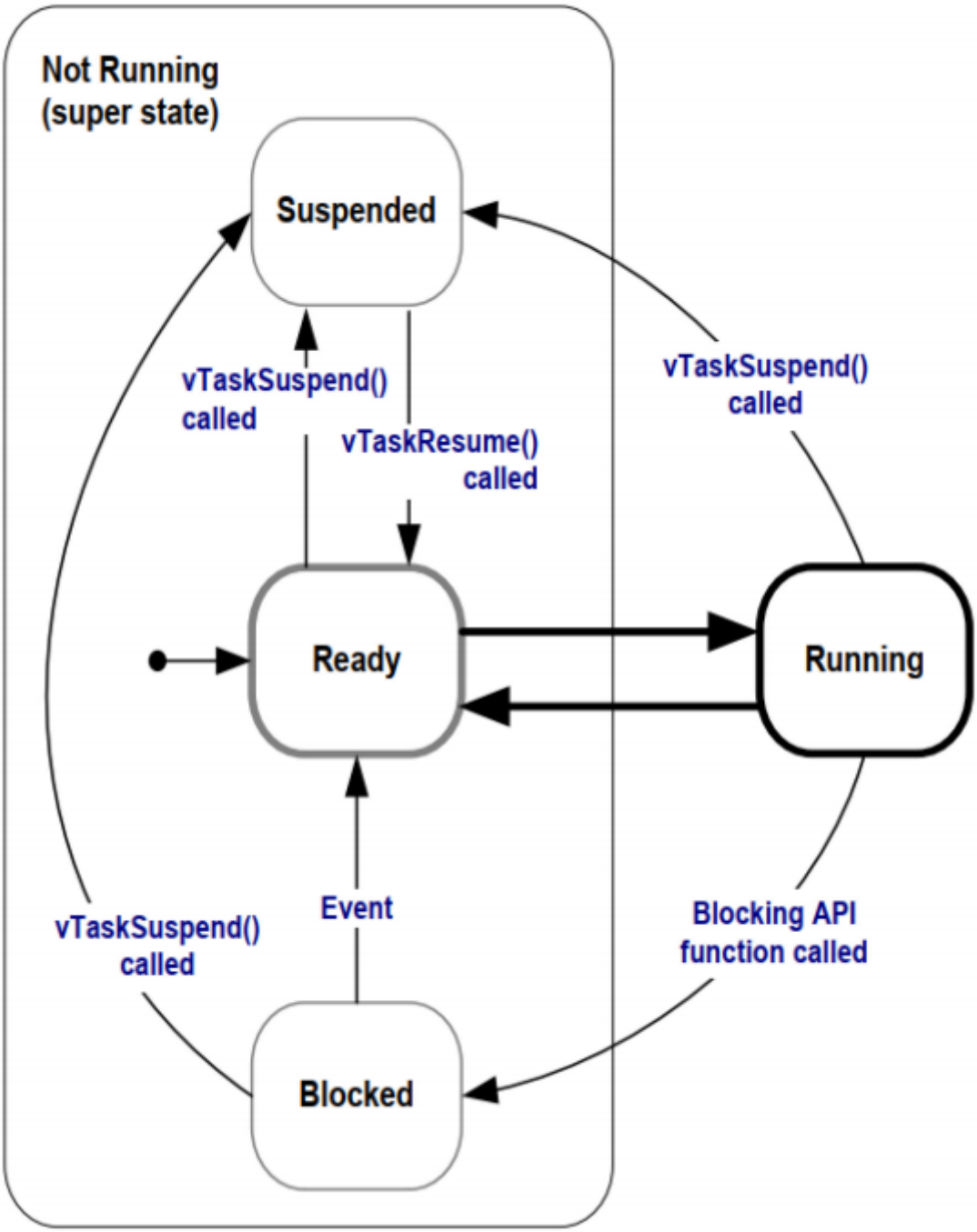
像前面一样，如果把它当做行车的过程的话，就是我们“任务小车”直接离开了车道

就绪状态**(Ready):**

这个任务完全准备好了，随时可以运行：只是还轮不到它。这时，它就处于就绪态

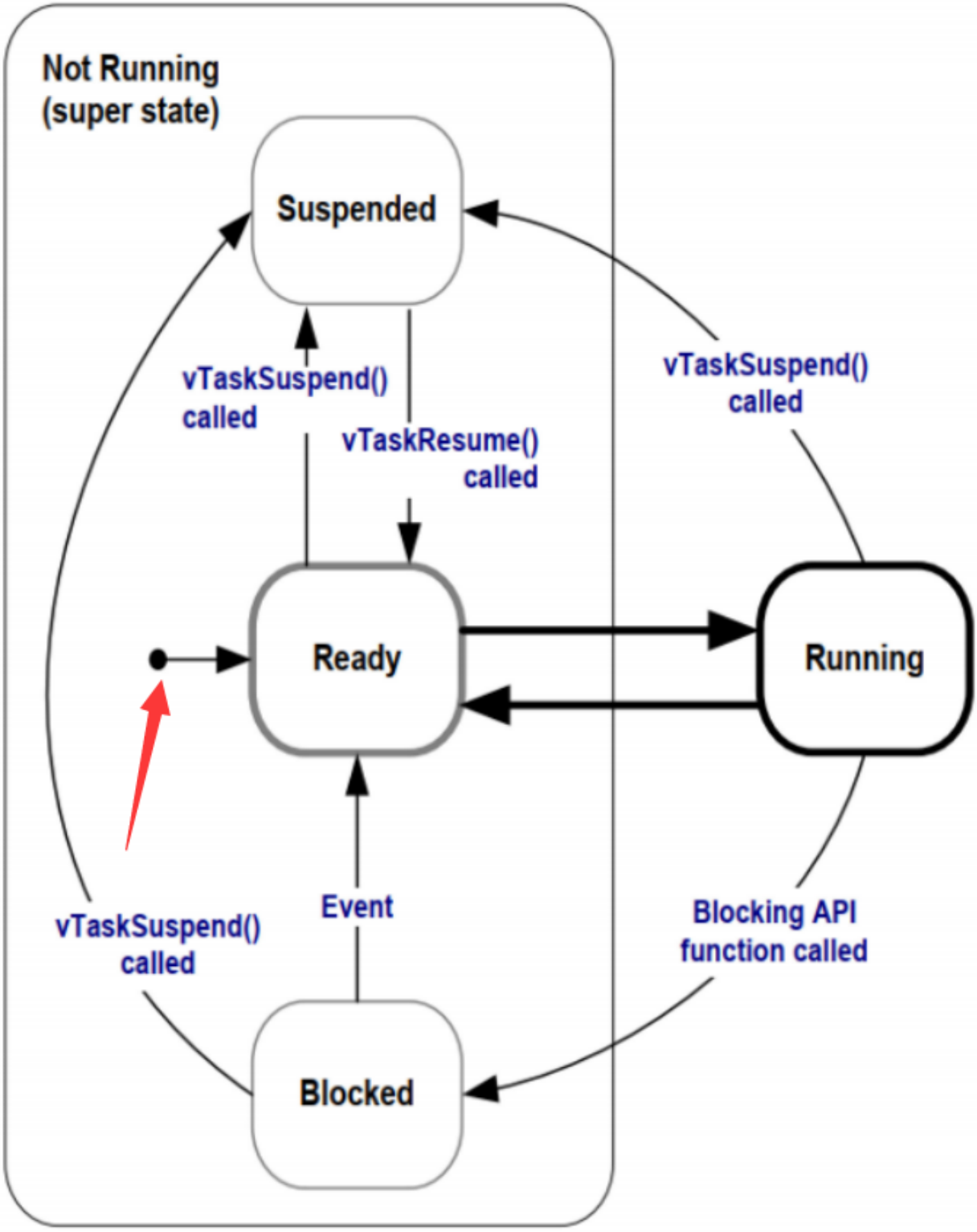
像前面一样，如果我们把它当做行车的过程的话，我们的任务小车就是已经具备了行车的条件，但是由于我们的那个灯是红灯，所以还是跑不起来

贰：状态之间的转换



这里是一张我们的那个状态转换的图表

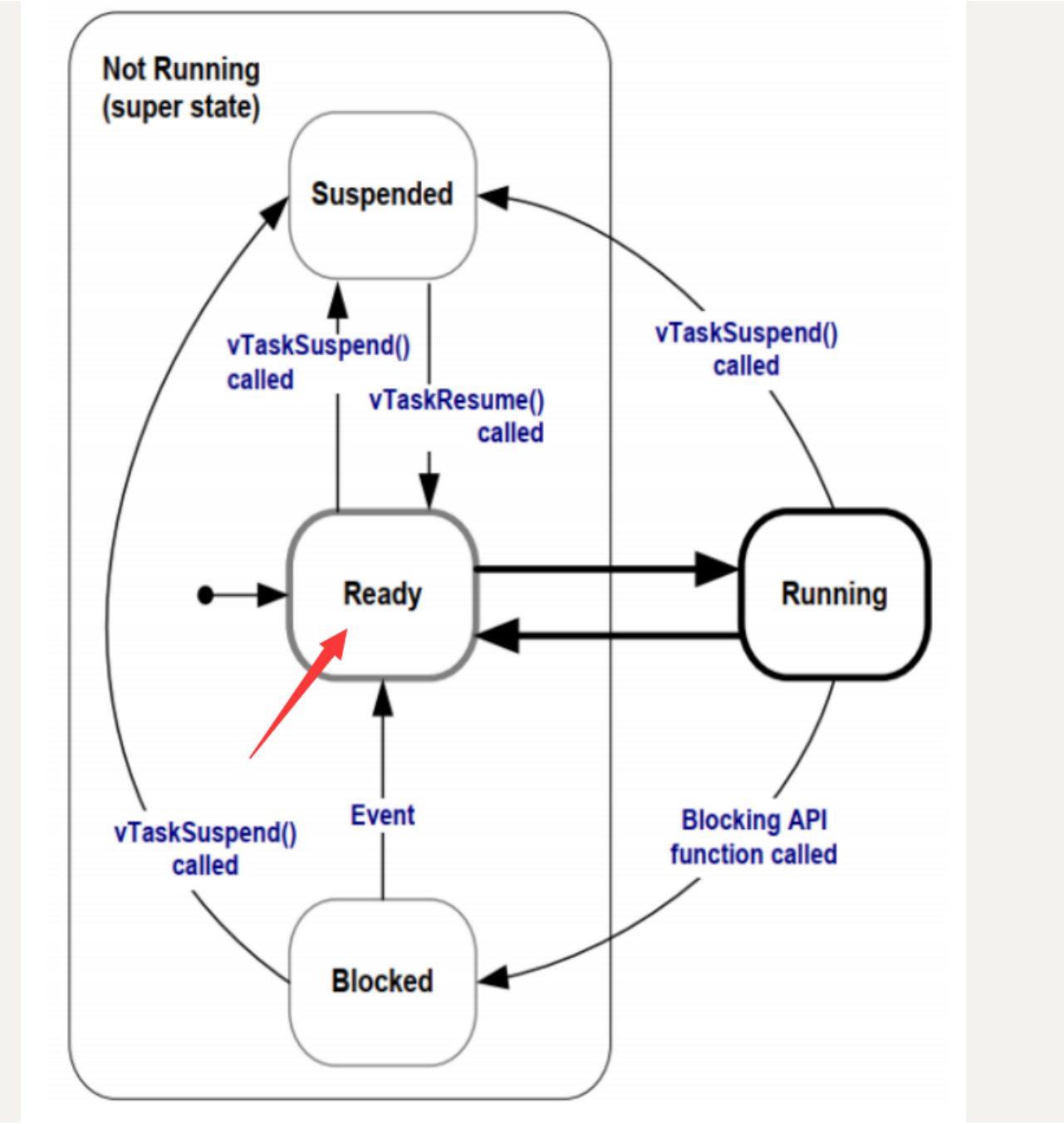
任务刚创建：



我们任务刚创建的时候，就是这个小黑点，它默认是会直接进入我们的就绪状态，但是它并不会直接运行，一般是在那个调度器启动之后。才会有任务进入运行状态，因为各个任务之间要进行相互的比较和判断，谁可以先执行

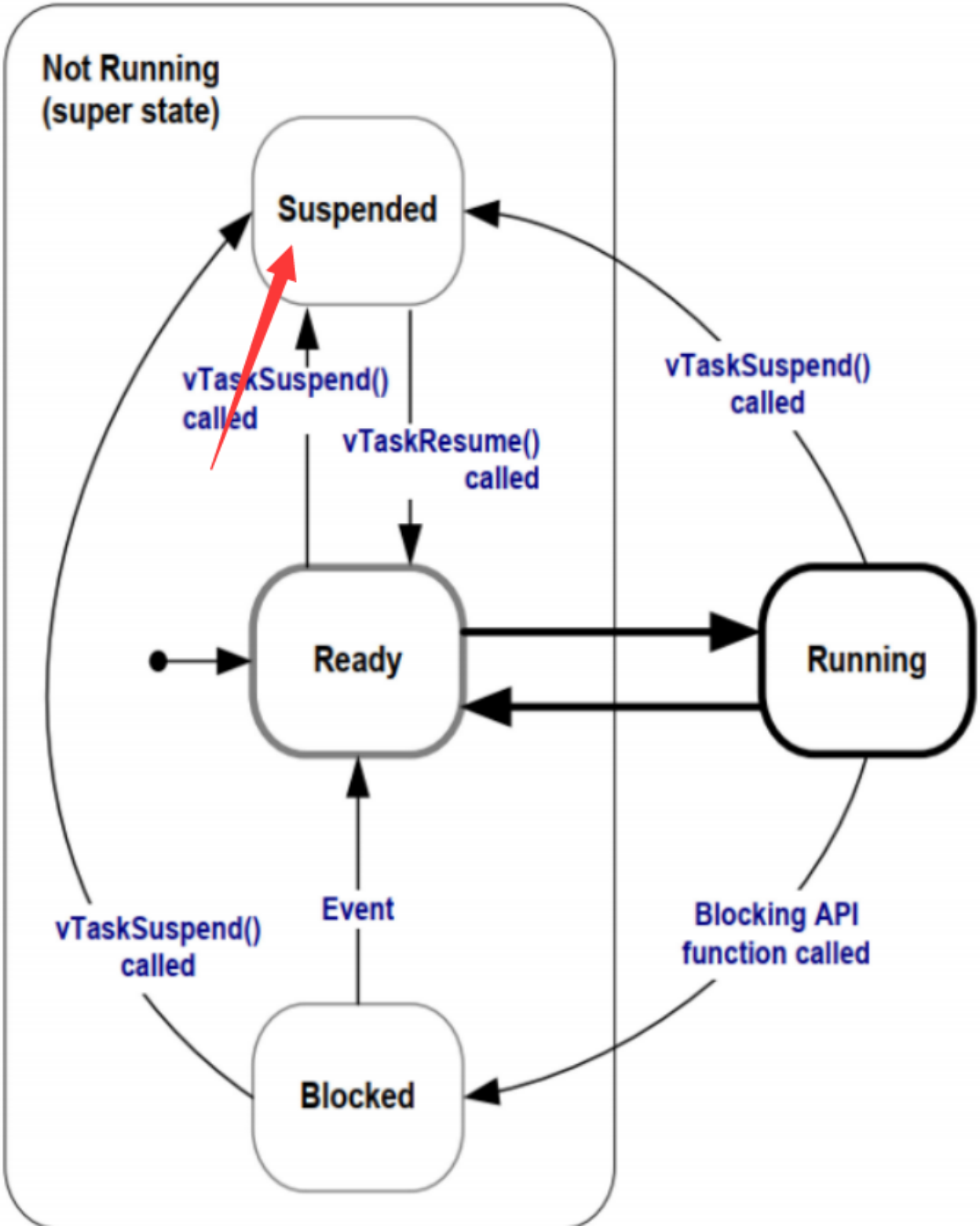
这里提一下，我们创建完任务之后启动任务调度器默认是最高优先级的先执行，如果同优先级的话，我们一般是后创建的那个任务会先执行

就绪状态：



我们的就绪任务状态就是我们图片中最中央的这个位置，因为通常我们的项目中任务会比较多，但是我们每次只能执行一个任务，这样的话就会有些任务，它具备了执行的条件，但是优先级抢不过其他的任务，他就是就绪状态

暂停状态：



暂停状态也有人把它叫做我们的悬挂状态，这个状态是一个比较特殊的状态，就是我们直接手动给它叫停了，我们在任何状态下都可以使用对应的函数将其设置为这个暂停状态，这样的话他就永远不会被执行，直到我们将其恢复

void vTaskSuspend(TaskHandle_t xTaskToSuspend);

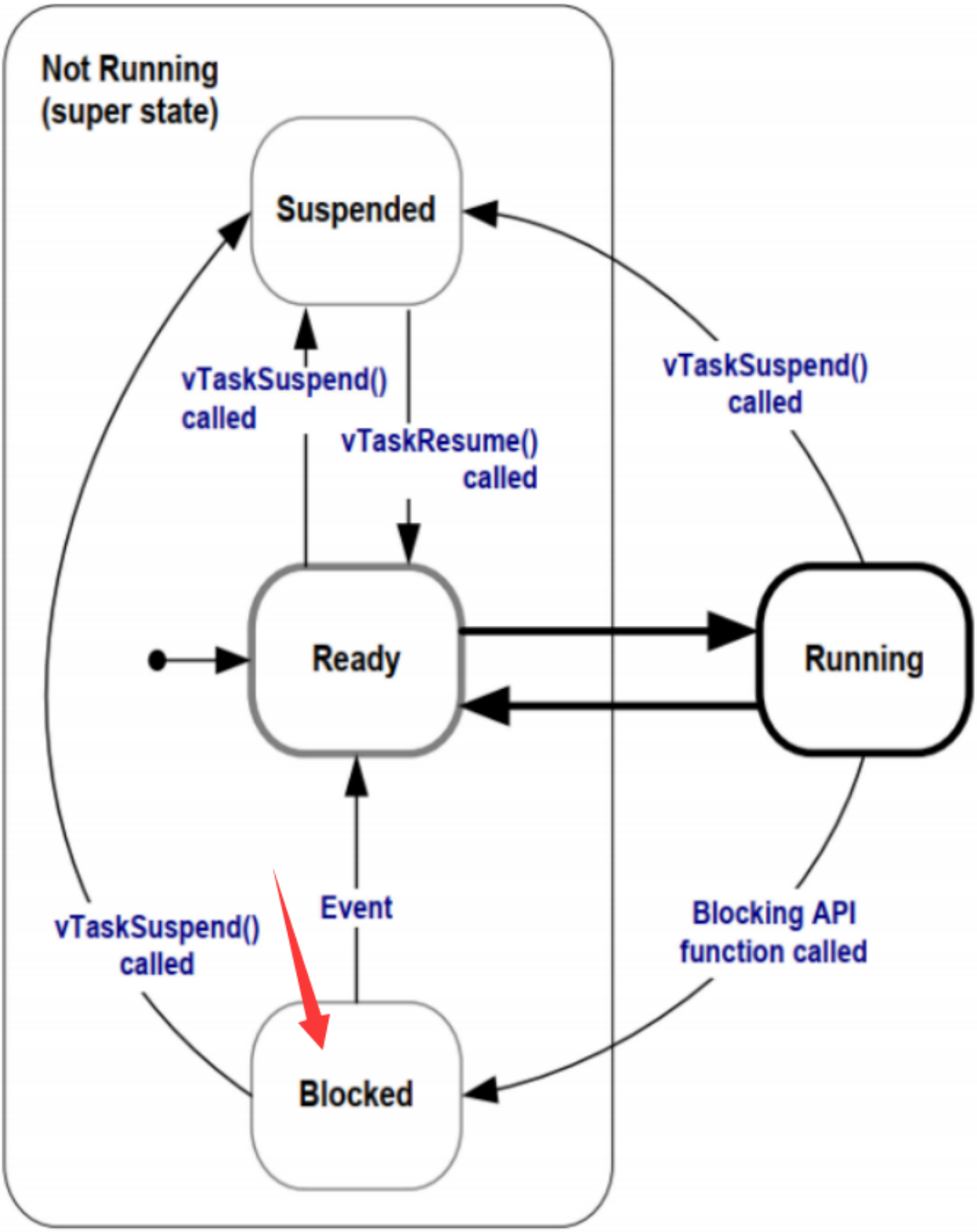
其唯一的进入方法就是使用该函数进入到暂停状态

void vTaskResume(TaskHandle_t xTaskToResume);

这个是推出的函数，退出回到就绪状态

这两个函数传入的参数都是任务的句柄（后续会讲解到！）

阻塞状态：



这个就是我们的老朋友了，我们使用对应的有阻塞功能的函数，就可以让我们现在的任务进行短时间的阻塞，然后把我们那个任务的执行权交给已经在就绪状态的任务，让他有时间去执行

一般情况进入阻塞状态的两个函数：

void　vTaskDelay(const TickType_t xTicksToDelay);

第1个函数它就比较简单，就非常像我们的HAL_Delay()函数。然后它里面传入的是我们的那个需要堵塞tick数,然后这个tick，就是指我们的那个节拍单位默认就是毫秒嘛，所以默认情况下它就是和我们的这个HAL_Delay是没有很大区别的，只不过还有HAL_Delay就是在空跑程序，而我们的这个是让它进入了阻塞的状态

BaseType_t　xTaskDelayUntil(TickType_t * const pxPreviousWakeTime, const TickType_t xTimeIncrement);

参数	说明
pxPreviousWakeTime	指针，指向一个变量(指针说明这个变量即可以当做输入类型的，也可以当做输出类型的)。该变量保存任务最后一次解除阻塞的时间。第一次使用前，该变量必须初始化为当前时间。之后这个变量会在vTaskDelayUntil()函数内自动更新
xTimeIncrement	周期循环时间。当时间等于(*pxPreviousWakeTime + xTimeIncrement)时，任务解除阻塞。如果不改变参数xTimeIncrement的值，调用该函数的任务会按照固定频率执行

如果指定的唤醒时间已经达到，vTaskDelayUntil()立刻返回（不会有阻塞）。因此，使用vTaskDelayUntil()周期性执行的任务，无论任何原因（比如，任务临时进入挂起状态）停止了周期性执行，使得任务少运行了一个或多个执行周期，那么需要重新计算所需要的唤醒时间。这可以通过传递给函数的指针参数pxPreviousWake指向的值与当前系统时钟计数值比较来检测，在大多数情况下，这并不是必须的

一般情况下，我们将第**1**个延时函数叫做相对时间延时，然后第**2**个叫做绝对时间延时

叁：两个延时函数的验证

然后这一小节我们将会在我们的项目中给大家验证一下，两个延时函数它的区别，带大家认识一下相对延时和绝对延时

相对延时：

是指两次任务执行的间隔时间是相对的（延时时间=任务执行时间+需要延时的时间）

绝对延时：

是指两次任务执行的间隔时间是绝对的（延时时间=需要延时的时间）

验证方法：

实验：

Task1使用相对延时500MS，Task2使用绝对延时500MS，使用HAL_Dlay(200)模拟任务占用时间，观察现象结果

现象：

```
[2022-08-10 10:15:58.236 R]Task2

[2022-08-10 10:15:58.437 R]Task1

[2022-08-10 10:15:58.731 R]Task2

[2022-08-10 10:15:59.137 R]Task1

[2022-08-10 10:15:59.229 R]Task2

[2022-08-10 10:15:59.724 R]Task2

[2022-08-10 10:15:59.835 R]Task1

[2022-08-10 10:16:00.225 R]Task2

[2022-08-10 10:16:00.535 R]Task1

[2022-08-10 10:16:00.737 R]Task2
```

任务一延时了700ms左右，

任务二演示了较为准确的500MS