免费实时操作系统（FreeRTOS）对称多处理（SMP）变更描述

2020/01/13

概述

内核已发生重大变化，几乎全部都在 tasks.c 中。新增了配置选项。对端口也有新的要求——每个端口都必须实现一些新的端口宏，一些端口宏不再需要，还有一些现有的端口宏需要略有不同的行为。

对便携层所做的更改

现在，每个端口都必须提供一种在多个内核上启动调度器的方法。它还必须向内核提供两个自旋锁。这些锁的实现方式并不重要。在 xcore 上，它们是由 xcore 架构提供的硬件锁。它还必须提供一种内核相互中断的方法。

新行为

xPortStartScheduler()

现在，这必须根据新配置选项 configNUM_CORES 指定的内核数量启动调度程序。每个内核都必须通过恢复分配给它的任务的上下文来结束此函数。pxCurrentTCB 已被重命名为 pxCurrentTCBs，并转变为一个由内核编号索引的数组。每个内核都必须索引到这个数组中，以获取它必须恢复的任务的堆栈指针。

任务上下文的保存与恢复

如上所述，pxCurrentTCB 已被重命名为 pxCurrentTCBs 并做成了一个由核心索引的数组。每当需要保存或恢复任务状态时，现在必须首先索引这个数组以获取堆栈指针。除此之外，上下文的保存和恢复方式与往常一样。

portDISABLE_INTERRUPTS()

现在，在禁用中断之前，它还必须返回中断掩码。它仍可能在未检查返回值的情况下被调用。

portENTER_CRITICAL()

这必须被定义为 vTaskEnterCritical（） 。这要求将 portCRITICAL_NESTING_IN_TCB 定义为 1 。

portEXIT_CRITICAL()

这必须被定义为 vTaskExitCritical（） 。这要求将 portCRITICAL_NESTING_IN_TCB 定义为 1 。

portSET_INTERRUPT_MASK_FROM_ISR()

除了在调用 vTaskEnterCritical（） 之前返回中断掩码外，还必须调用 vTaskEnterCritical（） 。如果端口不支持嵌套中断，则可能会返回一个虚拟值。

portCLEAR_INTERRUPT_MASK_FROM_ISR(x)

在将中断掩码设置为 x 中的值之前，必须首先调用 vTaskExitCritical（） 。如果端口不支持嵌套中断，可能会忽略 x 并且不设置中断掩码。

新宏

portRESTORE_INTERRUPTS(x)

这必须将中断掩码设置为 x 中的值。它在调用 portDISABLE_INTERRUPTS（） 之后被使用，以将中断掩码恢复为调用 portDISABLE_INTERRUPTS（） 之前的值。x 的值应该是 portDISABLE_INTERRUPTS（） 返回的值。

portGET_CORE_ID()

这必须返回调用核心的核心 ID（在 0 到 configNUM_CORES - 1 之间）。

portYIELD_CORE(x)

这必须中断具有 ID x 的核心。被中断的核心必须表现得好像它调用了 portYIELD（） （即保存当前任务上下文，调用 vTaskSwitchContext（） ，然后恢复新当前任务的上下文）。

端口 检查是否在中断服务程序中

如果是在中断服务程序（ISR）内部调用，则必须返回 pdTRUE；否则返回 pdFALSE 。

portGET_TASK_LOCK()

这必须获取一个自旋锁。锁的实现必须是递归的。如果一个内核获取了 N 次，那么在另一个内核能够获取它之前，它必须被释放 N 次。

port 释放任务锁

这必须释放由 portGET_TASK_LOCK（） 获取的自旋锁。该锁必须是递归的。例如，如果 portGET_TASK_LOCK（） 已被调用两次，那么对此的单次调用只会递减一个计数器，而不会释放锁。

portGET_ISR_LOCK()

与 portGET_TASK_LOCK（） 相同，但此函数必须获取一个不同的自旋锁。有关 portGET_TASK_LOCK（） 的描述，请参阅。

portRELEASE_ISR_LOCK()

这必须释放由 portGET_ISR_LOCK（） 获取的自旋锁。有关 portRELEASE_TASK_LOCK（） 的描述，请参阅。

已移除宏

portYIELD_WITHIN_API()

此宏不再被 SMP FreeRTOS 使用。tasks.c 中的与端口无关的函数 vTaskYieldWithinAPI（） 取代了它。

对内核所做的更改

内核已被修改以支持多核。这意味着当一个任务准备就绪时，调度程序现在必须决定在哪个内核上运行它。为支持这一新行为所做的几乎所有更改都已应用于 tasks.c 中，并将在下面进行描述。

新的配置选项

为了支持多核，新增了配置选项 configNUM_CORES，只要硬件和端口支持，可将其设置为大于 0 的任何值。

存在多个内核使得多个任务能够同时运行成为可能。这也意味着具有不同优先级的多个任务有可能同时运行。不幸的是，这打破了现有 FreeRTOS 应用程序中的一个常见假设，即一个任务永远不会被优先级更低的任务抢占。这意味着依赖于这一假设的现有 FreeRTOS 应用程序在允许具有不同优先级的任务同时运行的 SMP 环境中运行时将会出错。

为了解决这个问题，引入了一个新的配置选项 configRUN_MULTIPLE_PRIORITIES。如果将其设置为 0，则多个任务可以同时运行，但前提是它们具有相同的优先级。如果将其设置为 1，则具有不同优先级级别的任务将被允许同时运行。例如，假设有两个内核，并且每个内核都在运行优先级为五级的任务。然后唤醒了一个优先级为六级的新任务。如果 configRUN_MULTIPLE_PRIORITIES 设置为 0，那么两个任务都会被抢占。一个内核将运行优先级为六级的新任务。另一个内核将被迫运行一个空闲任务。然而，如果 configRUN_MULTIPLE_PRIORITIES 设置为 1，则两个任务中只有一个会被抢占，以允许新任务运行，从而实现优先级为五级和优先级为六级的一个任务同时运行。

对于旨在充分利用对称多处理（SMP）的新应用程序，建议将 configRUN_MULTIPLE_PRIORITIES 设置为 1 。

配置 核心数

可用于运行任务的核心的数量。只要硬件和端口支持，可设置为大于 0 的任何数字。

配置 运行多种优先级

设置为 1 时，允许具有不同优先级级别的任务在不同内核上同时运行。

设置为 0 时，仅允许具有相同优先级的任务同时运行。请参阅上述讨论。

新行为

互斥性

SMP 引入的一个复杂性是互斥性。在单核系统中，通常可以通过禁用所有中断来进入临界部分。这会阻止所有中断服务例程运行，进而阻止上下文切换，确保在.....之前没有其他任务或中断服务例程能够执行。

[1] This is still technically true in SMP. However, in a single core environment this also means that a high priority task can be sure that a lower priority task will not be able to execute until it willingly gives up control. The same cannot be said in an SMP environment.

临界部分是通过重新启用中断来退出的。不幸的是，在SMP环境中并非如此。在多核系统中，当中断被禁用时，它们并不是为所有核心禁用，而是为禁用中断的核心禁用。即使中断为所有核心禁用，其他核心仍会继续运行它们的进程，并不会被阻止同时进入临界部分。因此，简单地禁用中断并不是在SMP环境中进入临界部分的可行方法。

解决方案是使用自旋锁。要进入临界区域，一个线程首先禁用中断，然后获取自旋锁。要退出临界区域，一个线程首先释放自旋锁，然后重新启用中断。例如，当任务A处于临界区域内时，其他核心的任务会继续运行，并且可能仍然会被中断。但如果任何尝试进入临界区域的线程无法获取自旋锁，直到任务A释放它为止。这保证了每次只有一个任务处于临界区域内。

然而，这种解决方案并不完全与为单核系统编写的应用程序代码兼容。在单核系统中，中断服务例程总是可以假设所有任务都在临界区之外，因此可以安全地对任务必须用临界区保护共享数据进行操作。在对称多处理（SMP）环境中，情况并非如此，因为中断服务例程可能仍在其中一个核心上运行，而另一个核心上的任务可能处于临界区内部。现在，中断服务例程也必须通过获取自旋锁进入临界区。

FreeRTOS 内核使用两种机制来保护共享数据。第一种是关键部分，它禁用中断并防止上下文切换。第二种机制是通过暂停调度程序，它不禁用中断，但防止上下文切换。它通过递增一个“暂停”计数器来实现，vTaskSwitchContext（） 会检查该计数器。如果计数器非零，则不执行上下文切换（但请注意，在恢复调度程序时应该会发生一次上下文切换）。

为了继续在 SMP FreeRTOS 中支持这两种机制，使用了两种不同的自旋锁。它们被命名为“任务”锁和“中断服务例程（ISR）”锁。当任务进入临界区段时，它禁用中断并获取这两个锁。当中断服务例程进入临界区段时，它仅获取 ISR 锁。当任务挂起调度程序时，它仅获取任务锁 2 并递增挂起计数器。

这意味着当一个任务处于临界区域内时，其他任务或中断服务例程（ISR）不能进入临界区域，并且其他任务也不能通过暂停调度程序来进入受保护的区域。当一个任务处于通过暂停调度程序来保护的区域内时，其他任务不能进行同样的操作或进入临界区域。然而，中断服务例程可以在任何核心上继续运行并进入临界区域。当中断服务例程处于临界区域内时，其他任务或中断服务例程不能进入临界区域。作为实现的一个副作用，当另一个核心上的中断服务例程处于临界区域内时，任务也无法进入通过暂停调度程序来保护的区域。这是因为增加暂停计数器受到两个锁的保护，因为它必须在持有任一锁的情况下能够被安全读取。

幸运的是，FreeRTOS 提供了在 ISR 中进入和退出临界区的宏，内核中的所有 ISR 代码都已经使用它们来保护共享数据。在单核 FreeRTOS 中，这些只需要由支持嵌套中断的端口来实现。在 SMP FreeRTOS 中，现在必须获取和释放 ISR 锁。

[2] When suspending the scheduler, the ISR lock must also be briefly acquired in order to safely increment the suspended counter. It is then released.

闲置任务

在单核 FreeRTOS 中，存在一个空闲任务。在 SMP FreeRTOS 中，每个核心都会创建一个空闲任务。因此，如果没有就绪任务，每个核心都会分配一个空闲任务。在表示每个任务的结构中添加了一个新的成员来标识空闲任务。内核使用它来区分系统空闲任务和其他也以空闲优先级运行的应用任务。当 configRUN_MULTIPLE_PRIORITIES 设置为 0 时，必须进行这种区分。如果一个具有高于空闲优先级优先级的任务就绪并在一个核心上运行，那么优先级低于它的任何应用任务都不能在其他核心上运行。然而，如果核心的数量多于能够运行的任务数量，这些未使用的核心仍然必须做点什么。因此，这些核心会被分配一个系统空闲任务。在这里进行这种区分是必要的，以确保这些核心只被分配系统空闲任务，而不是应用程序创建的任务，这些任务也具有空闲优先级。

上下文切换

只有 vTaskSwitchContext（） 这一个函数最终决定运行哪个任务。这个函数几乎只在中断服务程序（ISR）结束时以及调用 portYIELD（） 时运行（portYIELD（） 本质上会中断调用核心并进入 ISR）。然而，有许多函数会导致任务进入或离开就绪状态。每当发生这种情况时，它们都会尝试确定 vTaskSwitchContext（） 是否会切换出当前任务，例如，如果调用任务离开了就绪状态，或者（当启用抢占时）如果优先级高于调用任务的任务进入就绪状态。当这些函数确定必须切换出当前任务时，它们会调用 portYIELD（） 。

当只有一个核心时，这很容易处理，但当有多个核心时，就变得复杂了。例如，假设有一个双核系统，有任务 A、B 和 C。如果优先级为 2 且在核心 0 上运行的任务 A 将优先级为 3 的任务 B 移入就绪状态，任务 A 不一定愿意让位以允许任务 B 在核心 0 上运行。如果核心 1 正在运行优先级为 1 或 0 的任务 C，那么任务 C 最好让位以允许任务 B 在核心 1 上运行。这将导致核心 0 运行优先级为 2 的任务 A，核心 1 运行优先级为 3 的任务 B，而优先级为 1（或 0）的任务 C 仍然就绪但无法运行。

为了解决这个问题，添加了一个新的函数 prvYieldForTask（）。以前大多数检查新解锁任务的优先级、将其与调用任务的优先级进行比较，然后决定是否调用 portYIELD（） 的函数，现在改为调用这个新函数，并将新解锁任务作为其参数之一。这个函数会查看每个核心上运行的任务，并确定哪些任务（如果有的话）需要让步。其他核心上需要让步的任务会被中断。如果调用任务必须让步，那么它会在退出临界区时进行让步，因为要求 prvYieldForTask（） 必须在临界区内调用。

实际执行上下文切换的函数 vTaskSwitchContext（） 也需要进行修改。在单核 FreeRTOS 中，它必须做的就是在下一个最高优先级的就绪任务之间进行切换。当存在多个具有相同最高优先级的任务时，每次调用时它都会选择下一个任务：任务 A -> 任务 B -> 任务 C -> 任务 A 等等。

对于对称多处理（SMP），vTaskSwitchContext（） 必须选择下一个优先级最高的任务，该任务不能已经在其他内核上运行。为了使其知晓这一点，已向结构体中添加了新的运行状态成员。

表示每个任务。此成员要么保存其当前正在运行的核心 ID，要么表示它未运行或者正在等待让出。

如果最高就绪优先级级别上所有就绪任务已经在其他内核上运行，并且 configRUN_MULTIPLE_PRIORITIES 设置为 1，那么它可以搜索较低优先级级别上的就绪任务，并调度它找到的下一个任务。然而，如果 configRUN_MULTIPLE_PRIORITIES 为 0，那么它将调度一个系统空闲任务。

额外的复杂性

在初始测试期间发现的一个问题是，当一个任务等待进入关键部分时，另一个任务会中断它，因为它必须为另一个任务让步。当任务最终能够获取锁并进入关键部分时，它不会让步，因为其核心的中断被禁用。相反，它会在关键部分继续运行代码，只有在退出关键部分并重新启用中断后才会让步。当任务等待暂停调度程序时也存在类似的问题。在调度程序暂停后，中断不会被禁用，但让步将无法导致上下文切换，直到调度程序恢复之后。

在某些情况下，由 yield 导致的上下文切换必须在关键代码段内的代码执行之前先发生，特别是在 configRUN_MULTIPLE_PRIORITIES 为 0 时。为了解决这个问题，新增了函数 prvCheckForRunStateChange（）。它在 vTaskEnterCritical（） 和 vTaskSuspendAll（） 的底部被调用，在中断重新启用之前。然而，在嵌套调用中，如果任务已经在关键代码段或调度暂停中，则不会调用该函数。一旦任务已经在关键代码段中，那么在 yield 之前它必须完成并退出。

新的 prvCheckForRunStateChange（） 函数用于检查任务是否被其他任务中断。如果是，它会暂时重置临界区域和调度程序暂停计数器，释放两个锁，并重新启用中断。启用中断后，会立即发生让步。当任务最终重新安排并继续运行时，它会立即禁用中断，重新获取锁，并恢复计数器。同样，它必须检查在等待重新获取锁的过程中是否被中断。如果是，就必须重复。否则，它可能会返回并继续运行临界区域内的代码。

上述解决方案避免了在 FreeRTOS 测试中出现的所有错误，这些错误是由于关键部分能够在为其他内核上的任务让步之前运行而导致的。

代码变更描述

以下列出了所有新的函数和变量，并对其进行了描述。大多数已修改的函数和变量也被列出，并描述了其更改内容。然而，仅发生非常小的更改的函数，例如，唯一的更改是从 portYIELD_WITHIN_API（） 改为调用 vTaskYieldWithinAPI（） 的函数，则不列出。

“任务.h”

任务 恢复中断(x)

新端口的映射 RESTORE_INTERRUPTS(x

任务:检查是否在中断服务程序中

新端口的映射 检查是否在中断服务程序中

任务有效的核心 ID（xCoreID）
如果 xCoreID 是有效的内核 ID，则返回 pdTRUE，否则返回 pdFALSE 。

XTaskGetIdleTaskHandle()
这现在会返回一个指向一个列表的指针，该列表的长度为 configNUM_CORES 个 TaskHandle_t 对象。这是因为每个核心都有一个空闲任务。

未来的版本可能会将这种行为改为仅指向第一个空闲任务，以保持 API 不变。可能会引入一个新的函数来返回列表。

vTaskSwitchContext(x 内核 ID
这现在将核心 ID 作为参数。

xTaskGetCurrentTaskHandleCPU（x 内核 ID）
这个新函数类似于 xTaskGetCurrentTaskHandle（） ，但返回的是由 xCoreID 指定的核心上运行的任务的句柄。

vTaskYieldWithinAPI()
这个新函数取代了端口宏 portYIELD_WITHIN_API（） 。

“任务.c”
tasks.c 中的函数和数据结构主要负责任务的创建、调度和删除。它还包含由时钟中断调用的例程。

任务 如果采用抢占式调度则 yield
当 configUSE_PREEMPTION 不为 0 时，这现在被映射到新函数 vTaskYieldWithinAPI（） 而不是 portYIELD_WITHIN_API（） 。

任务 选择最高优先级任务
此宏已被删除，并被函数 prvSelectHighestPriorityTask（） 所取代。

TCB_t 结构
这个结构新增了两个成员。首先是 xTaskRunState，其类型为新的 TaskRunning_t 类型。这表明任务正在哪个核心上积极运行，或者它是在让步还是未运行。

第二个新成员 xIsIdle 如果任务是系统空闲任务则设置为 pdTRUE，否则设置为 pdFALSE 。

pxCurrentTCBs[]
这被重新命名为 pxCurrentTCB，并转变为一个长度为 configNUM_CORES 的数组。通过核心 ID 进行索引，每个元素都指向与索引相同的核心上运行的任务的 TCB 。

像素当前时间控制按钮
这现在是一个宏，它调用新函数 xTaskGetCurrentTaskHandle（） 。该函数返回在调用核心上运行的任务的 TCB 指针。

x 收益率 待定 数组
这被重新命名为 xYieldPending，并转变为一个长度为 configNUM_CORES 的数组。通过核心 ID 进行索引，每个元素表明在该索引所对应的核心上运行的任务是否有待进行的暂停。

x 收益率待定
这现在是一个宏，它调用新函数 prvGetCurrentYieldPending（） 。如果调用任务有待处理的挂起中断请求，则返回 pdTRUE 。否则返回 pdFALSE 。

“x 空闲任务句柄数组”
这现在是一个长度为 configNUM_CORES 的数组。每个元素都指向一个系统空闲任务 TCB。空闲任务并非锁定到特定的内核，因此它们没有特定的顺序。

prvGetCurrentYieldPending（void）
这是一个新函数。它返回由调用核心的核心 ID 索引的 xYieldPendings 中的元素。

prvGetLowestPriorityCore(uxMaxPriority, xCoreMask[])
这是一个新函数。它必须在关键部分内调用。它返回运行当前正在运行的低优先级任务的核心 ID。由于被新函数 prvYieldForTask（） 所取代，此函数在很大程度上已不再需要，不过仍有一处调用它。

prvCheckForRunStateChange（void）
这是一个新功能。在进入关键部分或禁用调度程序之后，会立即调用它，以便给任务一个机会，在等待获取锁的过程中，如果另一个任务请求它放弃，它就放弃。它通过检查当前运行状态是否为 taskTASK_YIELDING 来检测到这一点。请参阅上述标题为“额外的复杂性”的讨论。

prvYieldCore(xCoreID)
这是一个新函数。它必须在关键部分内调用。它请求具有 ID xCoreID 的核心让步。它通过使用新的端口宏 portYIELD_CORE（） 中断该核心来实现这一点。它还将当前运行状态更改为 taskTASK_YIELDING，以便 prvCheckForRunStateChange（） 能够检测到在任务等待进入关键部分时是否发生这种情况。

如果 xCoreID 是调用核心的 ID，那么其挂起等待标志会被设置，以便在退出临界区时进行挂起。

prvYieldForTask(pxTCB,xPreemptEqualPriority)
这是一个新功能。它必须在关键部分内调用。如果 configRUN_MULTIPLE_PRIORITIES 为 1，则它会查看每个核心上正在运行的任务，并向优先级级别低于（如果 xPreemptEqualPriority 为 pdTRUE 则等于）由 pxTCB 表示的新解锁任务的优先级级别且优先级级别最低的任务发送让步请求。如果没有任务符合此标准，则不会请求任何任务让步。

如果 configRUN_MULTIPLE_PRIORITIES 为 0，那么除了上述情况外，所有优先级低于新解除阻塞任务优先级的非空闲任务也被要求让步。这确保了如果新解除阻塞任务的优先级高于当前所有运行的任务，那么它们都会让步，以便一次只有一个优先级运行。被要求让步的任务中，有一项最终应该安排新解除阻塞任务。其余任务将安排系统空闲任务。

prvSelectHighestPriorityTask(xCoreID)
这是一个新功能，但它取代了宏任务 SELECT_HIGHEST_PRIORITY_TASK（） 并提供类似的行为。它选择当前未在运行的最高优先级任务的下一个任务。

另一个核心或让步。如果处于最高就绪优先级级别的所有就绪任务已经在其他核心上运行，并且 configRUN_MULTIPLE_PRIORITIES 设置为 1，那么它可以搜索较低优先级级别的就绪任务，并选择它找到的下一个任务。然而，如果 configRUN_MULTIPLE_PRIORITIES 为 0，那么它将选择系统空闲任务。

prvInitialiseNewTask()

对 this 函数所做的唯一更改是初始化新的 TCB 结构体成员 xTaskRunState 和 xIsIdle。如果 pxTaskCode 是空闲任务，则 xTaskRunState 设置为 taskTASK_NOT_RUNNING，xIsIdle 设置为 pdTRUE；否则，xIsIdle 设置为 pdFALSE。

prvAddNewTaskToReadyList()

此函数已修改，以便在调度程序尚未运行时选择最合适的内核来为新任务分配。如果调度程序正在运行，那么它现在会调用 prvYieldForTask（） 。

vTaskDelete()

此函数已修改，能够正确检测到被删除的任务当前是否在任何核心上运行。如果任务正在运行，则它会返回该任务所在的核心。

“eTaskGetState()”

此函数已修改，当任务在任何核心上运行时（而不仅仅是调用核心的核心），都会返回 eRunning 。

vTaskPrioritySet()

此函数已修改，当任务的优先级提高时调用 prvYieldForTask（） ，并且在降低任务的优先级时正确检查任务是否在任何核心上运行。如果是，则向其运行的核心发送一个让步请求。

vTaskSuspend()

此函数已修改，以正确检测正在暂停的任务当前是否在任何核心上运行。如果任务正在运行，则它将返回任务所在的核心。在调度程序尚未运行的情况下，由于 pxCurrentTCBs 现在是一个数组，并且要处理新的 xTaskRunState TCB 结构体成员，代码略有修改。PrvSelectHighestPriorityTask（） 也被调用，以代替 vTaskSwitchContext（）。为了安全地索引 pxCurrentTCBs[]，临界节段的退出位置也必须移动。

vTaskResume()

此函数已被修改以调用 prvYieldForTask（） 。

xTaskResumeFromISR()

此函数已被修改以调用 prvYieldForTask（） 。

vTaskStartScheduler()

此函数已被修改，以创建与内核数量相同的空闲任务。计时器任务的创建也已移至创建空闲任务之前。在创建空闲任务之前，如果 configRUN_MULTIPLE_PRIORITIES 为 1，则找出分配给内核的所有任务的最高优先级级别。所有被分配较低优先级级别的内核都被重新分配为无任务。然后，当创建空闲任务时，任何被分配无任务的内核都会被分配一个空闲任务。

vTaskSuspendAll()
如前所述，此函数已被修改以获取任务锁并调用 prvCheckForRunStateChange（） 。请参阅上述标题为“互斥和额外复杂性”的讨论。如果调度程序尚未运行，则也会跳过该函数的主体。

xTaskResumeAll()
此函数进行了一些小的修改，以处理将 xYieldPendings 作为数组的情况，以及当为调用核心设置 xYieldPendings 时，退出临界区会导致一个 yield 这一事实。如果调度程序尚未运行，则也会跳过函数体。

xTaskGetIdleTaskHandle()
此函数已被修改以返回空闲任务句柄数组。

xTaskAbortDelay()
此函数已被修改以调用 prvYieldForTask（） 。

xTaskIncrementTick()
此函数已被修改以进入临界区。在单核 FreeRTOS 中，它仅在中断中或在临界区内的 xTaskResumeAll（） 中被调用，因此它不需要处于临界区。然而，由于它修改的任务列表也可能被其他内核临界区内的任务所修改，所以此函数必须处于临界区内部。

它还经过了修改，在解除延迟列表中的任务阻塞时调用 prvYieldForTask（） ，并向所有需要时间切片且存在未决的 yield 请求的核心发送 yield 请求。

要求只有 0 号内核设置定时器中断来调用此函数。

vTaskSwitchContext()
此函数已被修改，以获取并随后释放两个锁，并调用 prvSelectHighestPriorityTask（） 而不是 taskSELECT_HIGHEST_PRIORITY_TASK（） 。它现在还接受一个单独的参数，即要在其上切换上下文的核心的 ID 。

xTaskRemoveFromEventList()
此函数已被修改以调用 prvYieldForTask（） 。

vTaskRemoveFromUnorderedEventList()
此函数已被修改以调用 prvYieldForTask（） 。

prvIdleTask()
空闲任务已被修改，只有在空闲优先级的就绪任务数量大于核心数量时才会让出。这样，只有当存在具有空闲优先级级别的应用程序任务时，系统空闲任务才会让出。

prvCheckTasksWaitingTermination()

由于 uxDeletedTasksWaitingCleanUp 有可能被另一个空闲任务修改，因此已对该函数进行了修改以进行双重检查。同时，还对其进行了修改以验证待删除的任务实际上已通过 vTaskSwitchContext（） 被切换出，因为在多核系统中，该任务仍有可能在另一个内核上运行。

xTaskGetCurrentTaskHandle()
此函数已被修改，以返回在调用核心上运行的任务的任务句柄。它必须在禁用中断的情况下，使用 portGET_CORE_ID（） 对 pxCurrentTCBs[] 进行索引。如果未禁用中断，则在获取核心 ID 和索引数组之间可能会发生上下文切换。如果调用任务在另一个核心上运行，则它将返回错误的任务句柄。

xTaskGetSchedulerState()
此函数已被修改以进入临界区域。如果调度程序已被调用任务挂起，那么它将能够进入临界区域，检查调度程序是否被挂起，并返回 taskSCHEDULER_SUSPENDED。如果调度程序已被其他任务挂起，那么在其他任务恢复调度程序之前，它将无法进入临界区域。然后，它会检查调度程序是否未被挂起，并调用 taskSCHEDULER_RUNNING。这只是为了确保只有在调用任务挂起调度程序的情况下，它才会返回 taskSCHEDULER_SUSPENDED。

vTaskYieldWithinAPI()
这是一个新功能。如果调用任务处于临界区域内，那么它会为调用核心设置 xYieldPendings 标志，以便在退出临界区域时执行 yield 操作。否则，它会立即通过调用 portYIELD（） 来执行 yield 操作。

vTaskEnterCritical()
此函数必须通过 portENTER_CRITICAL（） 调用。它已被修改，以获取中断服务例程锁，如果从中断服务例程调用则获取中断服务例程锁，如果从任务调用则获取任务锁。它还调用 prvCheckForRunStateChange（）。请参阅上述标题为“互斥和额外复杂性”的讨论。

vTaskExitCritical()
此函数必须由 portEXIT_CRITICAL（） 调用。如果从任务而非中断服务例程调用，并且调度程序未暂停，则已修改该函数以释放中断服务例程锁以及任务锁。如果 xYieldPending 不为 pdFALSE，则还会执行一次任务切换，因为 vTaskYieldWithinAPI（） 如果在临界段内调用，会将 xYieldPending 设置为 pdTRUE。请参阅上述标题为“互斥和额外复杂性”的讨论。

xTaskGenericNotify()
此函数已被修改以调用 prvYieldForTask（） 。

xTaskGenericNotifyFromISR()
此函数已被修改以调用 prvYieldForTask（） 。

vTaskNotifyGiveFromISR()
此函数已被修改以调用 prvYieldForTask（） 。

xTaskGetIdleRunTimeCounter()
此函数已被修改，以返回所有系统空闲任务的总时间。