FreeRTOS SMP Change Description

2020/01/13

Overview

Significant changes have been made to the kernel, almost entirely within tasks.c. New config options have been added. There are also new requirements for ports – there are some new port macros that must be implemented by each port, some port macros are no longer needed, and some existing port macros require slightly different behavior.

Changes made to the portable layer

Each port must now provide a way to start the scheduler on multiple cores. It must also provide to the kernel two spinlocks. It does not matter how these locks are implemented. On xcore they are hardware locks provided by the xcore architecture. It also must provide a way for cores to interrupt each other.

New Behavior

xPortStartScheduler()

This must now start the scheduler on the number of cores specified by the new config option configNUM_CORES. Each core must end this function by restoring the context of the task assigned to it. pxCurrentTCB has been renamed to pxCurrentTCBs and turned into an array indexed by core number. Each core must index into this array to get the stack pointer for the task it must restore.

Task context save and restore

As stated above, pxCurrentTCB has been renamed to pxCurrentTCBs and made into an array indexed by core. Whenever task state needs to be saved or restored, this array must now be indexed first to get the stack pointer. Beyond this, context is saved and restored as usual.

portDISABLE_INTERRUPTS()

This must now also return the interrupt mask prior to disabling. It may still be called without checking the return value.

portENTER_CRITICAL()

This must be defined to vTaskEnterCritical(). This requires that portCRITICAL_NESTING_IN_TCB be defined to 1.

portEXIT_CRITICAL()

This must be defined to vTaskExitCritical(). This requires that portCRITICAL_NESTING_IN_TCB be defined to 1.

portSET_INTERRUPT_MASK_FROM_ISR()

This must call vTaskEnterCritical() in addition to returning the interrupt mask prior to calling it. If the port does not support nested interrupts then it may return a dummy value.

portCLEAR_INTERRUPT_MASK_FROM_ISR(x)

This must first call vTaskExitCritical() before setting the interrupt mask to the value in x. If the port does not support nested interrupts it may ignore x and not set the interrupt mask.

New Macros

portRESTORE_INTERRUPTS(x)

This must set the interrupt mask to the value in x. It is used after calling portDISABLE_INTERRUPTS() to restore the interrupt mask to what it was prior to calling portDISABLE_INTERRUPTS(). The value x should be what was returned by portDISABLE_INTERRUPTS().

portGET_CORE_ID()

This must return the core ID (between 0 and configNUM_CORES-1) of the calling core.

portYIELD_CORE(x)

This must interrupt the core with ID x. The core that is interrupted must behave as if it called portYIELD() (i.e. save the current task context, call vTaskSwitchContext(), and then restore the context of the new current task).

portCHECK_IF_IN_ISR()

This must return pdTRUE if it is called from within an ISR, otherwise pdFALSE if it is not.

portGET_TASK_LOCK()

This must acquire a spinlock. The lock implementation must be recursive. If it is acquired N times by a core, it must then be released N times before another core is able to acquire it.

portRELEASE_TASK_LOCK()

This must release the spinlock acquired by portGET_TASK_LOCK(). The lock must be recursive. For example, if portGET_TASK_LOCK() has been called twice, then a single call to this will only decrement a counter and not release the lock.

portGET_ISR_LOCK()

Same as portGET_TASK_LOCK() but this must acquire a different spinlock. See the description for portGET_TASK_LOCK().

portRELEASE_ISR_LOCK()

This must release the spinlock acquired by portGET_ISR_LOCK(). See the description for portRELEASE_TASK_LOCK().

Removed Macros

portYIELD_WITHIN_API()

This macro is no longer used by SMP FreeRTOS. The port agnostic function vTaskYieldWithinAPI() in tasks.c replaces it.

Changes made to the kernel

The kernel has been modified to support multiple cores. This means that when a task becomes ready the scheduler now must decide which core to run it on. Almost all the changes made to support this new behavior have been made to tasks.c and will be described below.

New Configuration Options

To support multiple cores, the new config option configNUM_CORES has been added and may be set to anything greater than 0, provided the hardware and port supports it.

That there are multiple cores makes it possible for multiple tasks to run simultaneously. This also means that it can be possible for multiple tasks with different priority levels to run simultaneously. Unfortunately, this breaks a common assumption among existing FreeRTOS applications that a task will never be preempted by another task with a lower priority[1]. This means that existing FreeRTOS applications that rely on this assumption will break when run in an SMP environment that allows tasks with different priority levels to run simultaneously.

To fix this, a new config option configRUN_MULTIPLE_PRIORITIES has been introduced. If it is set to 0 then multiple tasks may run simultaneously, but only if they have the same priority. If it is set to 1 then tasks with different priority levels will be allowed to run simultaneously. For example, assume there are two cores, and that each is running a task of priority level five. Then a task with priority level six is woken up. If configRUN_MULTIPLE_PRIORITIES is set to 0, then both tasks will be preempted. One core will run the new task with priority level six. The other core will be forced to run an idle task. If, however, configRUN_MULTIPLE_PRIORITIES is set to 1, then only one of the two tasks will be preempted to allow the new task to run, resulting in both a task with priority level five and a task with priority level six running simultaneously.

It is encouraged that configRUN_MULTIPLE_PRIORITIES be set to 1 for new applications designed to fully take advantage of SMP.

configNUM_CORES

The number of cores available to run tasks. May be set to anything greater than 0, provided the hardware and port supports it.

configRUN_MULTIPLE_PRIORITIES

Set to 1 to allow tasks with different priority levels to run simultaneously on different cores.

Set to 0 to only allow tasks with the same priority level to run simultaneously. See discussion above.

New Behavior

Mutual Exclusion

One complication introduced with SMP is mutual exclusion. In a single core system, it is typically possible to enter a critical section by disabling all interrupts. This prevents all ISRs from running which in turn prevents context switches, guaranteeing that no other task or ISR will be able to execute until the

[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.

critical section is exited by re-enabling interrupts. Unfortunately, this is not the case in an SMP environment. When interrupts are disabled in a multi-core system, they are not disabled for all cores but rather only for the core that disables them. Even if interrupts were disabled for all cores, the other cores would continue to run their tasks and would not be prevented from simultaneously entering critical sections. Thus, simply disabling interrupts is not a viable method for entering a critical section in an SMP environment.

The solution is to use spinlocks. To enter a critical section, a thread first disables interrupts and then acquires a spinlock. To exit a critical section, a thread first releases the spinlock and then re-enables interrupts. For example, when Task A is inside a critical section, tasks on other cores will continue to run and may be still be interrupted. But if any attempt to enter a critical section they will be unable to acquire the spinlock until Task A releases it. This guarantees that only one task can be inside a critical section at a time.

This solution, however, is not entirely compatible with application code written for a single core system. On a single core system, ISRs can always assume that all tasks are outside of critical sections, and thus may safely operate on shared data that tasks must protect with critical sections. In an SMP environment this is not true, as ISRs may still run on one core while a task on another core is inside a critical section. It is now necessary for ISRs to also enter critical sections by acquiring the spinlock.

The FreeRTOS kernel uses two mechanisms to protect shared data. The first is critical sections, which disables interrupts and prevents context switches. The second mechanism is by suspending the scheduler, which does not disable interrupts, but does prevent context switches. It does this by incrementing a “suspended” counter which is checked by vTaskSwitchContext(). If it is non-zero it does not perform a context switch (but does note that one should occur upon resuming the scheduler).

To continue to support these two mechanisms in SMP FreeRTOS, two different spinlocks are used. They have been named the “task” and “ISR” locks. When a task enters a critical section, it disables interrupts and acquires both locks. When an ISR enters a critical section, it acquires only the ISR lock. When a task suspends the scheduler, it acquires only the task lock[2] and increments the suspended counter.

This means that when a task is inside a critical section, no other task or ISR may enter a critical section, and no other task may enter a section protected by suspending the scheduler. When a task is in a section protected by suspending the scheduler, no other task may do the same or enter a critical section. However, ISRs may continue to run and enter critical sections on any core. When an ISR is in a critical section, no other task or ISR may enter a critical section. As a side effect of the implementation, tasks are also unable to enter sections protected by suspending the scheduler while an ISR on another core is in a critical section. This is because incrementing the suspended counter is protected by both locks, as it must be able to be safely read while holding either lock.

Fortunately, FreeRTOS has macros for entering and exiting critical sections from within ISRs, and all ISR code in the kernel already uses them to protect shared data. In single core FreeRTOS these only need to be implemented by ports that supported nested interrupts. In SMP FreeRTOS these now must acquire and release the ISR lock.

[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.

Idle Tasks

In single core FreeRTOS there is a single idle task. In SMP FreeRTOS, one idle task is created per core. Thus, if there are no ready tasks, each core gets assigned an idle task. A new member to identify idle tasks has been added to the structure that represents each task. This is used to by the kernel to distinguish between the system idle tasks and other application tasks that also run at the idle priority level. This distinction must be made when configRUN_MULTIPLE_PRIORITIES is set to 0. If a task with a priority level greater than the idle priority level is ready and running on one core, then no application tasks with a priority level less than it may run on other cores. However, if there are more cores than the number of tasks able to run, these unused cores still must do something. Therefore, these cores are scheduled a system idle task. The distinction is necessary here in order to ensure that these cores are only scheduled system idle tasks, rather than tasks created by the application that also have the idle priority.

Context Switching

There is only one function - vTaskSwitchContext() - that ultimately decides which task to run. This function is almost exclusively run at the end of ISRs and when portYIELD() (which essentially interrupts the calling core and lands inside an ISR) is called. There are many functions, however, that cause tasks to move into or out of the ready state. Whenever this happens they try to determine if vTaskSwitchContext() would switch out the current task, for example if the calling task is moved out of the ready state, or (when preemption is enabled) if a task with a priority greater than the priority of the calling task is moved into the ready state. When these functions determine that the current task must be switched out they will call portYIELD().

This is easy enough when there is only a single core but is complicated when there is more than one. For example, assume a two-core system with Tasks A, B, and C. If Task A with priority 2 and running on Core 0 moves Task B with priority 3 into the ready state, Task A does not necessarily want to yield to allow Task B to run on Core 0. If Core 1 is running Task C with priority 1 or 0 then it is better for Task C to yield to allow Task B to run on Core 1. This will result in Core 0 running Task A with priority 2, Core 1 running Task B with priority 3, and Task C with priority 1 (or 0) still ready but unable to run.

To solve this, a new function prvYieldForTask() has been added. Most functions that previously checked the priority of a newly unblocked task, compared this with the priority of the calling task, and then decided whether or not to call portYIELD() now instead call this new function with the newly unblocked task as one of its arguments. This function looks at the task running on each core and determines which ones, if any, need to yield. Tasks on other cores that need to yield are interrupted. If the calling task must yield, then it does so upon exiting the critical section, as it is required that prvYieldForTask() be called from within a critical section.

The function that actually performs the context switches, vTaskSwitchContext(), also needed to be modified. In single core FreeRTOS all it must do is switch in the next highest priority ready task. When there are multiple tasks of this same highest priority it just selects the next one each time it is called: Task A -> Task B -> Task C -> Task A, etc.

For SMP, vTaskSwitchContext() must select the next highest priority task that is not already running on another core. For it to know this, a new run state member has been added to the structure that

represents each task. This member either holds the ID of the core that it is currently running on, or either that it is not running or is waiting to yield.

If all the ready tasks at the highest ready priority level are already running on other cores and configRUN_MULTIPLE_PRIORITIES is set to 1, then it can search for ready tasks at lower priority levels and schedule the next one it finds. However, if configRUN_MULTIPLE_PRIORITIES is 0, then it will schedule a system idle task.

Additional Complications

One issue that was identified during initial testing was when one task is waiting to enter a critical section while another task interrupts it because it must yield for another task. When the task is finally able to acquire the locks and enter the critical section, it does not yield because interrupts on its core are disabled. It instead continues to run the code in the critical section and only yields once the critical section is exited and interrupts are re-enabled. A similar problem exists when the task is instead waiting to suspend the scheduler. Interrupts are not disabled after the scheduler is suspended, but the yield will be unable to result in a context switch until after the scheduler is resumed.

There are cases where it is critical that the context switch resulting from the yield happen first before the code within the critical section is executed, specifically when configRUN_MULTIPLE_PRIORITIES is 0. To address this issue, the new function prvCheckForRunStateChange() has been added. It is called at the bottom of vTaskEnterCritical() and vTaskSuspendAll() before interrupts are re-enabled. It is not called, however, in nested calls where the task was already either in a critical section or in a scheduler suspension. Once a task is already in a critical section, then it must finish and exit it before yielding.

The new prvCheckForRunStateChange() function checks to see if the task was interrupted by another task. If it was, it temporarily resets the critical section and scheduler suspension counters, releases both locks, and re-enables interrupts. Upon enabling interrupts, the yield immediately takes place. When the task is eventually rescheduled and continues running it immediately disables interrupts, reacquires the locks, and restores the counters. Again, it must check to see if it was interrupted while waiting to reacquire the locks. If it was it must repeat. Otherwise it may return and continue to run the code within the critical section.

The solution described above prevents all the errors encountered in the FreeRTOS tests that were caused when critical sections were able to run before yielding for tasks running on other cores.

Code Change Descriptions

All new functions and variables are listed and described below. Most functions and variables that have been modified are also listed and their changes described. Functions with only very minor changes, however, are not listed, for example where the only change is a call to vTaskYieldWithinAPI() instead of portYIELD_WITHIN_API().

task.h

taskRESTORE_INTERRUPTS(x)

Maps to the new portRESTORE_INTERRUPTS(x)

taskCHECK_IF_IN_ISR()

Maps to the new portCHECK_IF_IN_ISR()

taskVALID_CORE_ID(xCoreID)
Returns pdTRUE if xCoreID is a valid core ID, otherwise returns pdFALSE.

XTaskGetIdleTaskHandle()
This now returns a pointer to a list of length configNUM_CORES of TaskHandle_t objects. This is because there is one idle task per core.

A future release may change this behavior to just point at the first idle task to keep the API the same. A new function may be introduced instead to return the list.

vTaskSwitchContext(xCoreID)
This now takes the core ID as an argument.

xTaskGetCurrentTaskHandleCPU(xCoreID)
This new function is similar to xTaskGetCurrentTaskHandle() but returns the handle for the task running on the core specified by xCoreID.

vTaskYieldWithinAPI()
This new function replaces the port macro portYIELD_WITHIN_API().

tasks.c
The functions and data structures in tasks.c are primarily responsible for creating, scheduling, and deleting tasks. It also contains the routine called by the tick interrupt.

taskYIELD_IF_USING_PREEMPTION()
When configUSE_PREEMPTION is not 0, this is now mapped to the new function vTaskYieldWithinAPI() rather than to portYIELD_WITHIN_API().

taskSELECT_HIGHEST_PRIORITY_TASK()
This macro has been removed and replaced with the function prvSelectHighestPriorityTask().

TCB_t structure
Two new members have been added to this structure. The first is xTaskRunState which has the new type TaskRunning_t. This indicates which core the task is actively running on, or if it is either yielding or not running.

The second new member xIsIdle is set to pdTRUE if the task is a system idle task, otherwise pdFALSE.

pxCurrentTCBs[]
This was renamed from pxCurrentTCB and turned into an array of length configNUM_CORES. Indexed by core ID, each element points to the TCB for the task running on the same core as the index.

pxCurrentTCB
This is now a macro that calls the new function xTaskGetCurrentTaskHandle(). This returns the TCB pointer for the task running on the calling core.

xYieldPendings[]
This was renamed from xYieldPending and turned into an array of length configNUM_CORES. Indexed by core ID, each element indicates if there if a yield is pending for the task running on the same core as the index.

xYieldPending

This is now a macro that calls the new function prvGetCurrentYieldPending(). This returns pdTRUE if there is a yield pending for the calling task. Otherwise it returns pdFALSE.

xIdleTaskHandle[]

This is now an array of length configNUM_CORES. Each element points to a system idle task TCB. The idle tasks are not locked to cores, and thus they are in no particular order.

prvGetCurrentYieldPending(void)

This is a new function. It returns the element in xYieldPendings indexed by the core ID of the calling core.

prvGetLowestPriorityCore(uxMaxPriority, xCoreMask[])

This is a new function. It must be called from within a critical section. It returns the ID of the core running the lowest priority task that is running. This function is largely OBE as it was replaced by the new function prvYieldForTask(), although it is still called in one place.

prvCheckForRunStateChange(void)

This is a new function. It is called immediately following entering a critical section or disabling the scheduler to give the task a chance to yield first in case another task has requested that it do so while it was waiting to acquire locks. It detects this by checking to see if its current run state is taskTASK_YIELDING. See the discussion above titled Additional Complications.

prvYieldCore(xCoreID)

This is a new function. It must be called from within a critical section. It requests that the core with ID xCoreID yields. It does this by interrupting it using the new port macro portYIELD_CORE(). It also changes its current run state to taskTASK_YIELDING so that prvCheckForRunStateChange() can detect if this happens while the task is waiting to enter a critical section.

If xCoreID is the ID of the calling core, then its yield pending flag is instead set so that it will yield upon exiting the critical section.

prvYieldForTask(pxTCB, xPreemptEqualPriority)

This is a new function. It must be called from within a critical section. If configRUN_MULTIPLE_PRIORITIES is 1 then it looks at the actively running task on each core and sends a yield request to the one with the lowest priority level that is less than (or equal to if xPreemptEqualPriority is pdTRUE) the priority level of the newly unblocked task represented by pxTCB. If no task meets this criterion then none are requested to yield.

If configRUN_MULTIPLE_PRIORITIES is 0, then in addition to the above, all non-idle tasks that have a priority level less than the priority level of the newly unblocked task are also requested to yield. This ensures that if the newly unblocked task has a higher priority level than all currently running tasks, that they all yield so that only one priority level runs at a time. One of the tasks that is requested to yield should end up scheduling the newly unblocked task. The rest will schedule system idle tasks.

prvSelectHighestPriorityTask(xCoreID)

This is a new function but replaces the macro taskSELECT_HIGHEST_PRIORITY_TASK() and provides similar behavior. It selects the next task with the highest priority level that is not currently running on

another core or yielding. If all the ready tasks at the highest ready priority level are already running on other cores and configRUN_MULTIPLE_PRIORITIES is set to 1, then it can search for ready tasks at lower priority levels and select the next one it finds. However, if configRUN_MULTIPLE_PRIORITIES is 0, then it will select a system idle task.

prvInitialiseNewTask()

The only change made to this function is to initialize the new TCB struct members xTaskRunState and xIsIdle. xTaskRunState gets taskTASK_NOT_RUNNING, and xIsIdle gets pdTRUE if pxTaskCode is the idle task, otherwise it gets pdFALSE.

prvAddNewTaskToReadyList()

This function has been modified to select the most appropriate core to assign the new task to when the scheduler is not yet running. If the scheduler is running then it now calls prvYieldForTask().

vTaskDelete()

This function has been modified to correctly detect if the task being deleted is currently running on any core. If the task is running, then it yields the core it is on.

eTaskGetState()

This function has been modified to return eRunning when the task is running on any core, rather than just the calling core.

vTaskPrioritySet()

This function has been modified to call prvYieldForTask() when the priority of the task is raised, and to correctly check if the task is running on any core when lowering its priority. If it is, a yield request is sent to the core it is running on.

vTaskSuspend()

This function has been modified to correctly detect if the task being suspended is currently running on any core. If the task is running, then it yields the core it is on. In the case where the scheduler is not yet running the code is slightly modified due to the fact that pxCurrentTCBs is now an array and to handle the new xTaskRunState TCB struct member. PrvSelectHighestPriorityTask() is also called in place of vTaskSwitchContext(). The location of the critical section exit also had to be moved in order to safely index pxCurrentTCBs[].

vTaskResume()

This function has been modified to call prvYieldForTask().

xTaskResumeFromISR()

This function has been modified to call prvYieldForTask().

vTaskStartScheduler()

This function has been modified to create as many idle tasks as there are cores. The creation of the timer task has also been moved to the beginning before creating the idle tasks. Before creating the idle tasks, if configRUN_MULTIPLE_PRIORITIES is 1, then the highest priority level of all the tasks assigned to cores is found. All cores assigned tasks with lower priority levels are reassigned no task. Then when the idle tasks are created any cores assigned no task get assigned an idle task.

vTaskSuspendAll()

This function has been modified to acquire the task lock and call prvCheckForRunStateChange() as discussed previously. See the discussions above titled Mutual Exclusion and Additional Complications. The body of the function is also skipped if the scheduler is not yet running.

xTaskResumeAll()

This function has minor modifications to deal with xYieldPendings as an array and the fact that when xYieldPendings is set for the calling core, exiting the critical section will result in a yield. The body of the function is also skipped if the scheduler is not yet running.

xTaskGetIdleTaskHandle()

This function has been modified to return the idle task handle array.

xTaskAbortDelay()

This function has been modified to call prvYieldForTask().

xTaskIncrementTick()

This function has been modified to enter a critical section. In single core FreeRTOS it was only ever called in an interrupt or from xTaskResumeAll() within a critical section, so it did not need to be within a critical section. However, since it modifies task lists that may also be modified by tasks within critical sections on other cores, it is necessary that this function be inside one.

It has also been modified to call prvYieldForTask() when unblocking tasks in the delayed list, as well as to send yield requests to all cores that require time slicing and have pending yields.

It is required that only core 0 sets up a timer interrupt that calls this function.

vTaskSwitchContext()

This function has been modified to acquire and later release both locks and to call prvSelectHighestPriorityTask() rather than taskSELECT_HIGHEST_PRIORITY_TASK(). It also now takes a single parameter that is the ID of the core on which to switch the context.

xTaskRemoveFromEventList()

This function has been modified to call prvYieldForTask().

vTaskRemoveFromUnorderedEventList()

This function has been modified to call prvYieldForTask().

prvIdleTask()

The idle task has been modified to yield only if the number of ready tasks at the idle priority is greater than the number cores. This way system idle tasks will only yield if there are application tasks that also have the idle priority level.

prvCheckTasksWaitingTermination()

This function has been modified to double check uxDeletedTasksWaitingCleanUp since it is possible for it to get modified by another idle task. It has also been modified to verify that the task to be deleted has actually been switched out by vTaskSwitchContext(), since it is possible in SMP that it is still running on another core.

xTaskGetCurrentTaskHandle()

This function has been modified to return the task handle of the task running on the calling core. It must index into pxCurrentTCBs[] with portGET_CORE_ID() with interrupts disabled. If interrupts are not disabled, then a context switch could happen between getting the core ID and indexing into the array. If the calling task lands on a different core, then it will return the wrong task handle.

xTaskGetSchedulerState()

This function has been modified to enter a critical section. If the scheduler has been suspended by the calling task then it will be able to enter the critical section, see that the scheduler is suspended, and return taskSCHEDULER_SUSPENDED. If the scheduler has been suspended by another task, then it will not be able to enter the critical section until the other task has resumed the scheduler. Then it will see that the scheduler is not suspended and call taskSCHEDULER_RUNNING. This simply ensures that it only returns taskSCHEDULER_SUSPENDED if the calling task is the one that suspended the scheduler.

vTaskYieldWithinAPI()

This is a new function. If the calling task is inside a critical section, then it sets the xYieldPendings flag for the calling core so that the yield will be performed upon exiting the critical section. Otherwise it yields immediately by calling portYIELD().

vTaskEnterCritical()

This function must be called by portENTER_CRITICAL(). It has been modified to acquire the ISR lock, as well as the task lock if called from a task and not an ISR. It also calls prvCheckForRunStateChange(). See the discussions above titled Mutual Exclusion and Additional Complications.

vTaskExitCritical()

This function must be called by portEXIT_CRITICAL(). It has been modified to release the ISR lock as well as the task lock if called from a task and not an ISR, and the scheduler is not suspended. See the discussions above titled Mutual Exclusion and Additional Complications. It also performs a yield if xYieldPending is not pdFALSE, as vTaskYieldWithinAPI() sets xYieldPending to pdTRUE if called from within a critical section.

xTaskGenericNotify()

This function has been modified to call prvYieldForTask().

xTaskGenericNotifyFromISR()

This function has been modified to call prvYieldForTask().

vTaskNotifyGiveFromISR()

This function has been modified to call prvYieldForTask().

xTaskGetIdleRunTimeCounter()

This function has been modified to return the total amount of time spent in all the system idle tasks combined.