FREE RTOS - updated 1482 - 3000

From line 1482 ,

```
#if ( INCLUDE_vTaskSuspend == 1 )
```

This macro checks if a task is suspended or not.

prvTaskIsSuspended() - 1484

- This function checks whether a given process is suspended or not.
- For the return value to be pdTrue, three conditions are required to be true :
    - If the task is actually in the suspended list ,
    - If the task is not resumed by the Interrupt Service Routine.
    - Task must be in suspended state or without timeout.

  If all 3 are true , the return value gets to pdTrue.

vTaskResume() - 1530

- This function is used to resume a suspended task.
- Again, for this to succeed in its action, it needs to satisfy certain conditions:
    - The Process Control Block must not be NULL or it should not be the current executing process as it is impossible to resume a current executing process.
    - prvTaskIsSuspended (taskToBeResumed) must be pdTrue
- Now, we enter the critical section by the function taskENTER_CRITICAL();
- There is also a possibility that the resumed process might have had higher priority than the current process. In this case, we preempt the current process.

vTaskResumeFromISR() - 1584

- If both xTaskResumeFromISR() and vTaskSuspend is 1 , only then this function will be invoked.

- This function is similar to the previous function, that is PCB must not be NULL or it should not be the current executing process.
- If the resumed task has higher priority than the current executing task, then xYieldRequired becomes true, symbolizing a need for context switch.

vTaskStartScheduler() - 1654

- This function is responsible for starting the task scheduler and initializing the idle and timer task.
- If INCLUDE_xTaskGetIdleTaskHandle is enabled, an idle process is created using xTaskCreate() and the task's handle is stored in xIdleTaskHandle for later retrieval using the xTaskGetIdleTaskHandle.
- The code in the else block does the same job but doesn't store any of the task's handles.
- If **configUSE_TIMERS** are enabled , the xTimerCreateTimerTask creates the timer tasks. If the idle task is succesfully created, pdPASS is returned
- portDISABLE_INTERRUPTS macro prevents a tick from occurring before or during the call to xPortStartScheduler().
- The portCONFIGURE_TIMER_FOR_RUN_TIME_STATS is called to configure the timer/counter used to generate the run-time counter time base.
- Now, if the function returns pdFalse, it indicates that the scheduler could not start properly.
- If pdTrue, we are cruising !!

vTaskEndScheduler() - 1732

- The vTaskEndScheduler first disables all the possible interrupts and then makes xSchedulerRunning as pdFalse.
- Then, it ultimately calls the vPortEndScheduler(), which ends the Scheduler program.

vTaskSuspendAll() - 1743

- This function suspends all the tasks in the scheduler.
- It increments the uxSchedulerSuspended, which is a type of UBaseType_t. The vTaskResume all does the decrement job, which resumes all the operations, which is contradictory to this function.

prvGetExpectedIdleTime() - 1755

- This function is only enabled if the configUSE_TICKLESS_IDLE is not 0. This means that tickless idle is enabled, which allows the system to enter a low-power state during idle periods, conserving energy by reducing frequency of timer interrupts.
- This function calculates the expected idle time of the system.
- The pxCurrentTCB->uxPriority > tskIDLE_PRIORITY is true if the current task is not an idle task. In this case, the expected idle task time is 0.
- listCURRENT_LIST_LENGTH(&(pxReadyTasksLists[tskIDLE_PRIORITY])) > 1 checks if there are idle priority tasks in ready queue. If there are and if time slice is enabled, then the next tick interrupt is enabled and the expected idle time is 0.
- Else, it returns the difference of xNextTaskUnblockTime - xTickCount as the expected time.

xTaskResumeAll() - 1781

- This function does the reverse of vTaskSuspendAll(). The uxSchedulerSuspended is updated back to 0 here.
- Here, if xPendingReadyList is not empty, it removes all the tasks and adds it to the ready queue using the prvAddTaskToReadyList()
- It also checks if there are any pending ticks. If yes, then it sets xYieldPending as pdTrue, symbolizing a need for context switch.

xTaskGetTickCount() and xTaskGetTickCountFromISR() - 1874 & 1889

- These 2 functions the current tick counts are returned.

- If you wonder what are tick counts, in RTOS implementation has got time units divided into discrete units called ticks. These are variously used in timing and scheduling purposes.

uxTaskGetNumberOfTasks() - 1920 : Gets number of tasks

pcTaskGetTaskName() - 1930

- It gets an object of TaskHandle_t and using prvGetTCBFromHandle, it gets the task control block
- From the given task control block, we only return the name of the task now.

uxTaskGetSystemState() - 1945

- This function is used to retrieve the current system state and information.
- First, it suspends all tasks using the xTaskSuspendAll().
- Now, if the size of the pxTaskStatusArray is greater than uxCurrentNumberOftasks,
    - A loop iterates the ready list starting from the highest priority to idle priority.
    - The information from the ready list is filled in the list of TaskStatus_t called **pxTaskStatusArray**
- Now after the ready list, it also iterates the list of blocked process list and then appends it into **pxTaskStatusArray**
- If task deletion and suspension is enabled, even that is extracted from macros and is stored in the **pxTaskStatusArray**
- If run-time stats are enabled (**configGENERATE_RUN_TIME_STATS** = 1) and pulTotalRunTime is not NULL , it obtains the actual run time.
- Finally, all tasks are resumed again using **xTaskResumeAll()**

vTaskStepTick () - 2036

- It takes a parameter xTicksToJump, which represents the number of ticks to jump.

- traceINCREASE_TICK_COUNT macro is then called to make the required change.

updateTableSameCriticalityOverrun() - 2048

- The function takes **taskOverrunId** as a parameter. The function iterates over all the tasks in the **globalRuntimeBudgetTable** (It is a multi-dimensional array that stores the runtime budgets of tasks in a global context.)
- Inside the loop, it retrieves execution behavior of current task and the overrun task.
- If the execution behavior of overrun task is less than current execution behavior, there is a need to update the behavior.
- So, the purpose of the function is to handle cases where a task overruns its allotted time budget.

xTaskIncrementTick() - 2163

- The function serves the following purposes:
  - Increments the tick count, updates run-time statistics, handles budget overruns, unblocks tasks, manages task switching and executes the tick hook function.
  - The configuration variables : **configGENERATE_RUN_TIME_STATS** , **configUSE_PREEMPTION , configUSE_TIME_SLICING , configUSE_TICK_HOOK** are responsible for the above functions discussed in point 1.

vTaskSetApplicationTaskTag() - 2469

- The function sets a hook function , task tag for a task. You may wonder …..what is a hook function after reading for too long ? It is a callback or a function that extends the functionality of a task beyond its normal execution. They are responsible for the house keeping functions like monitoring and logging, resource management , synchronization , error handling and recovery.
- If task is NULL, we set our own task hook by setting to current TCB.
- The function then returns xTCB→ pxTaskTag

xTaskCallApplicationTaskHook() - 2500

- The purpose of this function is to execute the taskHook() of the specific task

- If xtask is NULL, then it executes the taskHook() of the currentTCB() and then returns the value.

vTaskPlaceOnUnorderedEventList() - 2651

- This function is responsible for placing a task on an unordered event list (It is a list data structure with list of tasks that are waiting for an event to occur.)
- xTimeToWake is the time at which it should wake up the task if event doesn't occur.
- The function gets a parameter called xItemValue, stores list item to xEventListItem, field of the current task's TCB.
- Before adding to blocked list, it is first removed from ready list using uxListRemove.
- Depending on the config, the xTimeToWake is set. It is calculated by xTickCount + xTicksToWait.

vTaskPlaceOnEventListRestricted() - 2719

- This function is also responsible for placing tasks onto an event restricted list.
- `vTaskPlaceOnEventListRestricted` is optimized for scenarios where a single task is waiting on an event list, while `vTaskPlaceOnUnorderedEventList` provides additional flexibility and considerations for event group usage, including indefinite waits and a different handling of the blocking mechanism.

xTaskRemoveFromEventList() - 2762

- This function is to remove a task from the pxEventList
- After removing the task from the event List, it is appended to the ready list.
- These operations are possible only if the resumed task is not suspended.

xTaskRemoveFromUnorderedEventList() - 2830

- This function is to remove a task from the unordered event List.

- After removing the task from the unordered event List, it is appended to the ready list.
- These operations are possible only if the resumed task is not suspended.

Also, there are some other utility functions like **uxTaskGetNumber()** (Gets task id) , **vTaskMissedYield()** (sets xYieldPending to pdTrue) , **vTaskSetTimeoutState()** (sets the timeout for the processes) and **vTaskSetNumber()** (Sets task id for a task) which also contribute to the functioning of the freeRTOS code.