TASK MANAGEMENT

Index

1. Task States and Transition **3**
- Task 3
- Running State 3
- Ready State 3
- Blocked State 3
- Suspended State 4

2. Task Priorities **5**

3. Task Management APIs **6**

3.1. Task APIs fall into three categories 6

3.1.1. Task Creation 6
- xTaskCreate 6
- vTaskDelete 7

3.1.2. Task Control 7
- vTaskDelay 7
- vTaskDelayUntil() 8
- vTaskPrioritySet() 8
- uxTaskPriorityGet 9
- vTaskSuspend 9
- vTaskResume 10
- xTaskAbortDelay 10

3.1.3. Task Utilities 11
- xTaskGetCurrentTaskHandle 11
- xTaskGetIdleTaskHandle 11
- eTaskGetState 11
- pcTaskGetName 12
- xTaskGetHandle 12
- xTaskGetTickCount 12
- uxTaskGetSystemState 13

3.2. Time Measurement and the Tick Interrupt 14

3.3. Context Switching 15

3.3.1 Task Context 15

3.3.2 Task Stack 15

3.3.3 Detailed Example of TaskA Context Switch 15

3.4. Scheduling Algorithms 19

3.4.1 Configuring the Scheduling Algorithm 19

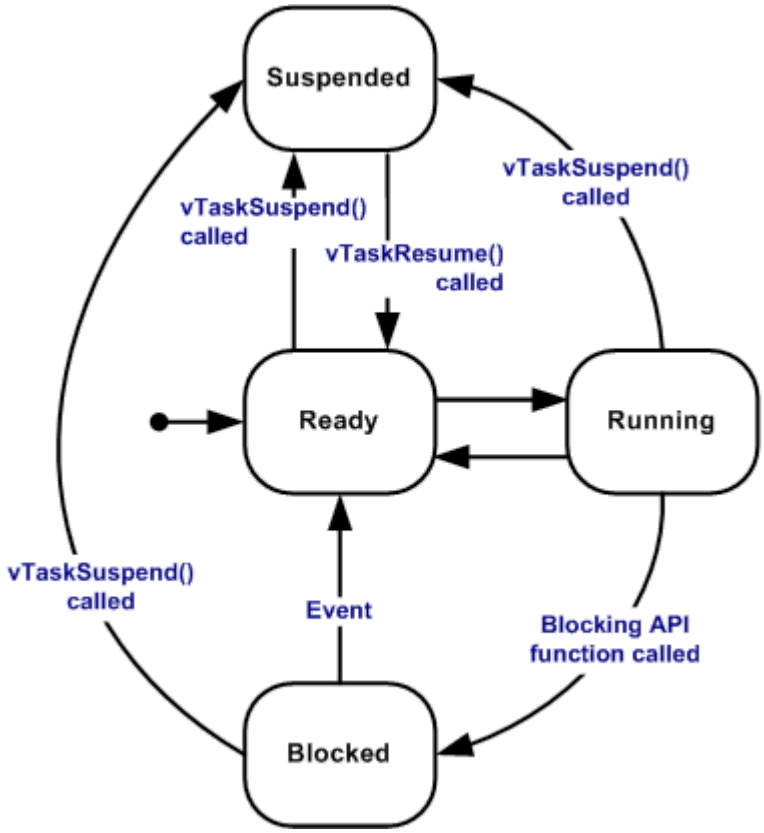
Fixed Priority 20

Pre-emptive 20

Time Slicing 20

3.4.2 Prioritized Pre-emptive Scheduling with Time Slicing 20
3.4.3 Prioritized Pre-emptive Scheduling (Without Time Slicing) 23
3.4.4 Co-operative Scheduling 24
4. Macros **25**
4.1 pdMS_TO_TICKS() 25
4.2 taskYIELD() 26
4.3 taskDISABLE_INTERRUPTS() 26
4.4 taskENABLE_INTERRUPTS() 26
4.5 taskENTER_CRITICAL() 27
4.6 taskEXIT_CRITICAL() 27
4.7 Comparison of interrupt macros 27
5. Examples **29**
5.1 Creating a Task and print it is created successfully or not 29
5.2 Get number of task by using uxTaskGetNumberOfTasks() 31
5.3 Get Task Details by using uxTaskGetSystemState() 33
5.4 The two tasks appear to execute simultaneously 41
5.5 Task 2 within Task1 and both the tasks are the same priorities. 43
Case 1 : When configUSE_TIME_SLICING is 0 and two tasks are having some delay. 43
Case 2 : When configUSE_TIME_SLICING is 0 and two tasks are having no delay. 45
Case 3 : When configUSE_TIME_SLICING is 1 and two tasks are having no delay. 45
Case 4 : When configUSE_TIME_SLICING is 1 and two tasks are having some delay. 46
5.6 Use of the vTaskDelete() 46
5.7 Difference between the vTaskDelay() and vTaskDelayUntil() 48
5.8 Led blinking 50
Case 1 : Toggling led to a single task. 50
Case 2 : led on and led off within the different tasks 52
5.9 Use of the vTaskPrioritySet() and vTaskPriorityGet() 54
5.10 Use of the vTaskSuspend() and xTaskResume() 56
5.11 Use of the xTaskAbortDelay() 58
5.12 Use of the xTaskGetTickCount() 60
5.13 Use of the Idle Task and Idle Task Hook 62
5.14 Use of the taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS() 64
5.15 Use of the taskENTER_CRITICAL() and taskEXIT_CRITICAL() 66

1. Task States and Transition



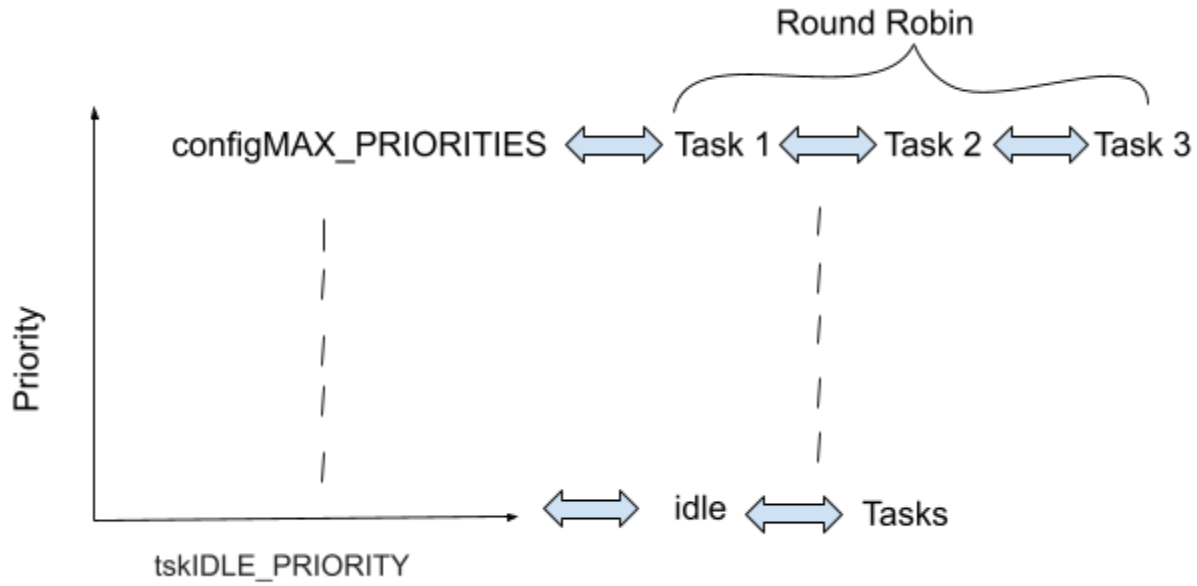
- **Task**
  - It is simple logic and implementation during the execution of the program to achieve a common functionality used multiple times.
- **Running State**
  - The task which is currently executing on the CPU.
- **Ready State**
  - The task that is ready to run and is waiting for CPU time. It has all the resources it needs except the CPU.
  - Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.
- **Blocked State**
  - If a task is currently waiting for either an external event or a temporary event, then it is in a blocked state.
  - For Example ,

        - If a task calls **vTaskDelay()** it will be in block until the delay period has expired.
        - Tasks can also be in block state waiting for queue, semaphore etc,..
        - Tasks in the Blocked state normally have a 'timeout' period, after which the tasks will be unblocked, even if the event for which the task was waiting for has not occurred.
        - Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

- **Suspended State**
    - Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a timeout.
    - Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the **vTaskSuspend()** and **xTaskResume()** API calls respectively.
    - Example :
        - **Sensor Reading Task:** Reads data from sensors.
        - **Data Processing Task:** Processes the collected sensor data.
        - **Communication Task:** Sends the processed data to the server.
        - **Maintenance Task:** Performs system maintenance tasks such as firmware updates or diagnostic checks.
        - In this scenario, the Maintenance Task might only need to run occasionally, such as once a day or when triggered by a specific event (e.g., a command from the server). When the Maintenance Task is not needed, it can be put into a suspended state to conserve system resources.

2. Task Priorities

- The FreeRTOS scheduler always ensures the highest priority task that can enter the running state. Tasks of equal priority are transitioned into and out of the Running state in turn.
- The **uxPriority** parameter of the API function used to create the task gives the task its initial priority. The **vTaskPrioritySet()** API function changes a task's priority after its creation.
- Any number of tasks can share the same priority. If **configUSE_TIME_SLICING** is not defined, or if **configUSE_TIME_SLICING** is set to 1, then Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme.
- If the port in use implements a port optimised task selection mechanism that uses a 'count leading zeros' type instruction (for task selection in a single instruction) and **configUSE_PORT_OPTIMISED_TASK_SELECTION** is set to 1 in **FreeRTOSConfig.h,** then **configMAX_PRIORITIES** cannot be higher than 32. In all other cases **configMAX_PRIORITIES** can take any value within a region - but for regions of RAM usage efficiency should be kept to the minimum value actually necessary.



- Low priority numbers denote low priority tasks. The idle task has priority zero (tskIDLE_PRIORITY).

3. Task Management APIs

3.1. Task APIs fall into three categories

Include task.h when we use task management APIs.

3.1.1. Task Creation

- **xTaskCreate**

  - BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    const configSTACK_DEPTH_TYPE uxStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask
    );

  - **Parameters :**
    - **pvTaskCode** : Pointer to the task entry function.
      - Tasks are normally implemented as an infinite loop and the function which implements the task must never attempt to return or exit.
    - **pcName** : Name for the task , Used to facilitate debugging , but can also be used to obtain a task handle.
      - The maximum length of a task's name is defined by **configMAX_TASK_NAME_LEN** in FreeRTOSConfig.h.
    - **uxStackDepth** : The number of words to allocate for use as the task's stack.
      - The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32-bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes).

        - The stack depth multiplied by the stack width must not exceed the maximum value that can be contained in a variable of type uint16_t.
        - The size of the stack used by the Idle task is defined by the application defined constant **configMINIMAL_STACK_SIZE → 1 .**
    - **pvParameters** : A value that is passed as the parameter to the created task.
    - **uxPriority** : The priority at which the created task will execute.
    - **pxCreatedTask** : Used to pass a handle to the created task out of the xTaskCreate() function. pxCreatedTask is optional and can be set to NULL.

  - **Returns** : If the task was created successfully then pdPASS is returned. Otherwise **errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY** is returned.

- **vTaskDelete**
    - **void vTaskDelete(TaskHandle_t pxTask);**
    - Deleted tasks no longer exist so cannot enter the Running state.
    - When a task is deleted, it is the responsibility of the idle task to free the memory that had been used to hold the deleted task's stack and data structures (task control block-TCB).
    - Only memory that is allocated to a task by the kernel itself is automatically freed when a task is deleted.
    - **Parameters** :
      - **pxTask** →The handle of the task being deleted.
      - A task can delete itself by passing NULL in place of a valid task handle.

      Returns : None

3.1.2. Task Control

- **vTaskDelay**
    - **void vTaskDelay(const TickType_t xTicksToDelay);**
    - **INCLUDE_vTaskDelay** must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.

- This function puts the calling task into the Blocked state for a specified number of tick periods. After this period has elapsed, the task will transition back to the Ready state.
- **vTaskDelay(pdMS_TO_TICKS(xTimeInMs));**
- Here, **xTimeInMs** is the delay duration in milliseconds, which is converted to tick periods using the **pdMS_TO_TICKS** macro.
- **Parameters** :
  - **xTicksToDelay** → The amount of time, in tick periods, that the calling task should block.
  - For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state and remain in the Blocked state until the tick count reached 10,100.
  - **Return** : None

- **vTaskDelayUntil()**
  - **void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement);**
  - **INCLUDE_vTaskDelayUntil** must be set to 1 in FreeRTOSConfig.h for the vTaskDelay() API function to be available.
  - The task that calls vTaskDelayUntil() into the Blocked state until an absolute time is reached.The task that called vTaskDelayUntil() exits the Blocked state exactly at the specified time, not at a time that is relative to when vTaskDelayUntil() was called.
  - **Parameters**:
    - **xLastWakeTime** is a variable that holds the last wake time of the task. This variable must be initialised before the first call to vTaskDelayUntil().
    - **xTimeIncrement** is the time interval in tick periods between consecutive executions of the task.
  - **Return** : None

- **vTaskPrioritySet()**
  - **void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority);**
  - The vTaskPrioritySet() API function is used to change the priority of any task the scheduler has started.
  - **Parameters** :
    - **pxTask :** Handle of the task
    - **uxNewPriority :** The priority to which the subject task is to be set. This is capped automatically to the maximum

available priority of (configMAX_PRIORITIES – 1), where configMAX_PRIORITIES is a compile time constant set in the FreeRTOSConfig.h header file.

- **uxTaskPriorityGet**
  - **UBaseType_t uxTaskPriorityGet(const TaskHandle_t pxTask);**
  - Used to query the priority of a task.
  - uxTaskPriorityGet() API function is available only when **INCLUDE_uxTaskPriorityGet** is set to 1 in FreeRTOSConfig.h.
  - **Parameters:**
    - **pxTask** → Handle of the task to be queried. Passing a NULL handle results in the priority of the calling task being returned.
    - A task can query its own priority by passing NULL in place of a valid task handle.
  - **Return** : The priority of xTask.

- **vTaskSuspend**
  - **void vTaskSuspend(TaskHandle_t pxTaskToSuspend);**
  - **INCLUDE_vTaskSuspend** must be set to 1 for the vTaskSuspend()
  - Place a task into the Suspended state. A task that is in the Suspended state will never be selected to enter the Running state. The only way of removing a task from the Suspended state is to make it the subject of a call to vTaskResume().
  - Tasks in the Suspended state are not available to the scheduler.

  - **Then vTaskSuspend() can be called to place a task into the suspended state before the scheduler has been started (before vTaskStartScheduler() has been called). This will result in the task (effectively) starting in the suspended state.**

  - **Parameters** :
    - **pxTaskToSuspend** → The handle of the task being suspended. A task may suspend itself by passing NULL in place of a valid task handle.

  - **Return** :
    - None

- **vTaskResume**
  - **void vTaskResume(TaskHandle_t pxTaskToResume);**
  - **INCLUDE_vTaskSuspend** must be set to 1 for the vTaskResume().
  - Transition a task from the Suspended state to the Ready state. The task must have previously been placed into the Suspended state using a call to vTaskSuspend().

  - **vTaskResume() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).**

  - **Parameters** :
    - **pxTaskToResume** → The handle of the task being resumed.
  - **Return** :
    - None

- **xTaskAbortDelay**
  - **BaseType_t xTaskAbortDelay(TaskHandle_t xTask);**
  - **INCLUDE_xTaskAbortDelay** must be set to 1 for the xTaskAbortDelay() API function to be available.

  - **xTaskAbortDelay() will move a task from the Blocked state to the Ready state even if the event for which the task is waiting for has not occurred, and the timeout specified when the task entered the Blocked state has not elapsed.**

  - While a task is in the Blocked state it is not available to the scheduler , and will not consume any processing time.

  - A task that is in the Blocked state is either waiting for a timeout period to elapse, or waiting with a timeout for an event to occur, after which the task will automatically leave the Blocked state and enter the Ready state. There are many examples :

    - If a task calls **vTaskDelay()** it will enter the Blocked state until the timeout specified by the function's parameter has elapsed, at which time the task will automatically leave the Blocked state and enter the Ready state.
    - If a task calls **ulTaskNotifyTake()** when its notification value is zero it will enter the Blocked state until either it receives a notification or the timeout specified by one of

the function's parameters has elapsed, at which time the task will automatically leave the Blocked state and enter the Ready state.

- **Parameters**:
  - **xTask** → handle the task.
- **Return** :
  - If the task referenced by xTask was removed from the Blocked state then **pdPASS** is returned.
  - If the task referenced by xTask was not removed from the Blocked state because it was not in the Blocked state then **pdFAIL** is returned.

3.1.3. Task Utilities

- **xTaskGetCurrentTaskHandle**
  - **TaskHandle_t xTaskGetCurrentTaskHandle(void);**
  - **INCLUDE_xTaskGetCurrentTaskHandle** must be set to 1 in FreeRTOSConfig.h for xTaskGetCurrentTaskHandle() to be available.
  - Returns the handle of the task is in Running state.
  - **Parameters** : None

- **xTaskGetIdleTaskHandle**
  - **TaskHandle_t xTaskGetIdleTaskHandle(void);**
  - **INCLUDE_xTaskGetIdleTaskHandle** must be set to 1 in FreeRTOSConfig.h for xTaskGetIdleTaskHandle() to be available.
  - Returns the task handle associated with the Idle task. The Idle task is created automatically when the scheduler is started.
  - **Parameters** : None.
  - **Return** : The handle of the Idle task.

- **eTaskGetState**
  - **eTaskState eTaskGetState(TaskHandle_t pxTask);**
  - **INCLUDE_eTaskGetState** must be set to 1 in FreeRTOSConfig.h for the eTaskGetState() API function to be available.
  - Returns as an enumerated type the state in which a task existed at the time eTaskGetState() was executed.
  - **Parameters** :
    - **pxTask** → The handle of the task.
  - **Return** :

State	Return Value
Running	eRunning (the task is querying its own state)
Ready	eReady
Blocked	eBlocked
Suspended	eSuspended
Deleted	eDeleted (the task's structures are waiting to be cleaned up)

- **pcTaskGetName**
  - **char * pcTaskGetName(TaskHandle_t xTaskToQuery);**
  - Queries the human readable text name of a task.
  - **Parameters** :
    - **xTaskToQuery** → The handle of the task.
  - **Return** :
    - Task names are standard NULL terminated C strings. The value returned is a pointer to the subject task's nam

- **xTaskGetHandle**
  - **TaskHandle_t xTaskGetHandle(const char *pcNameToQuery);**
  - **Parameters** :
    - **pcNameToQuery** → The name of the task being queried. The name is specified as a standard NULL terminated C string.
  - **Return** :
    - If a task has the exact same name as specified by the pcNameToQuery parameter then the handle of the task will be returned. If no tasks have the name specified by the pcNameToQuery parameter then NULL is returned.

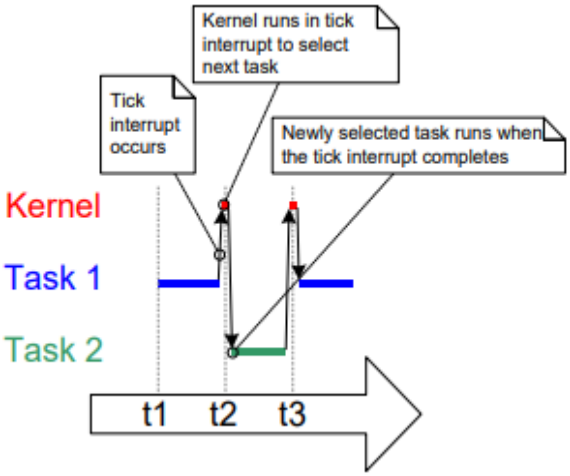
- **xTaskGetTickCount**
  - **TickType_t xTaskGetTickCount(void);**
  - The tick count is the total number of tick interrupts that have occurred since the scheduler was started.
  - **Parameters** : None.
  - **Return** : the tick count value.
  - The frequency at which the tick count overflows depends on both the tick frequency and the data type used to hold the count value. If **configUSE_16_BIT_TICKS** is set to **1,** then the tick count will be held in a 16-bit variable. If **configUSE_16_BIT_TICKS** is set to **0**, then the tick count will be held in a 32-bit variable.

- **uxTaskGetSystemState**
    - UBaseType_t uxTaskGetSystemState(
      TaskStatus_t * const pxTaskStatusArray,
      const UBaseType_t uxArraySize,
      uint32_t * const pulTotalRunTime);
    - The **uxTaskGetSystemState()** function in FreeRTOS is used to obtain information about the state of all tasks in the system. This function populates an array of TaskStatus_t structures with information about each task, which can be useful for debugging and monitoring purposes.

      pxTaskStatusArray: Pointer to an array of TaskStatus_t structures that will be filled with information about each task in the system.
      uxArraySize: The number of elements in the pxTaskStatusArray array. This should be at least equal to the number of tasks in the system.
      pulTotalRunTime: Pointer to a variable that will receive the total run time of the system, as a time base count. If this information is not needed, pass NULL.

3.2. Time Measurement and the Tick Interrupt

- If both tasks were created at the same priority and both tasks were always able to run. Therefore each task executed for a ‘time slice’, entering the Running state at the start of a time slice, and exiting the Running state at the end of a time slice.
- To be able to select the next task to run , the scheduler itself must execute at the end of each time slice. A periodic interrupt, called the **‘tick interrupt’**.
- If **configTICK_RATE_HZ** is set 100Hz , then the tick time is 10 milliseconds. The time between two tick interrupts is called the **tick period**.One time slice equals to one tick period.



From **example 2**, continue this figure. The execution sequence expanded to show the tick interrupt executing.

3.3. Context Switching

3.3.1 Task Context

- Each task in FreeRTOS has its own context, which includes the values of all these registers. The context is saved when a task is switched out and restored when the task is switched back in. This allows the task to continue execution exactly where it left off.

3.3.2 Task Stack

- Each task in FreeRTOS is allocated its own stack. The stack is a region of memory used for storing:
    - Function call return addresses
    - Local variables
    - Task context during context switches

3.3.3 Detailed Example of TaskA Context Switch

Let's consider an example where TaskA is preempted by TaskB.

TaskA Execution

1. TaskA Running:
    - TaskA is executing its code, using general-purpose registers for computations.
    - The program counter (PC) points to the current instruction of TaskA.
    - The stack pointer (SP) points to the current top of TaskA's stack.
2. Preemption by TaskB:
    - An interrupt or a scheduler decision occurs, indicating that TaskB should run.
    - The current context of TaskA (registers, PC, status register) is pushed onto TaskA's stack.

TaskB Execution

3. Switching to TaskB:
    - The stack pointer (SP) is switched to point to TaskB's stack.
    - The context (registers, PC, status register) for TaskB is popped from TaskB's stack.
    - TaskB starts executing from the instruction pointed to by its program counter.

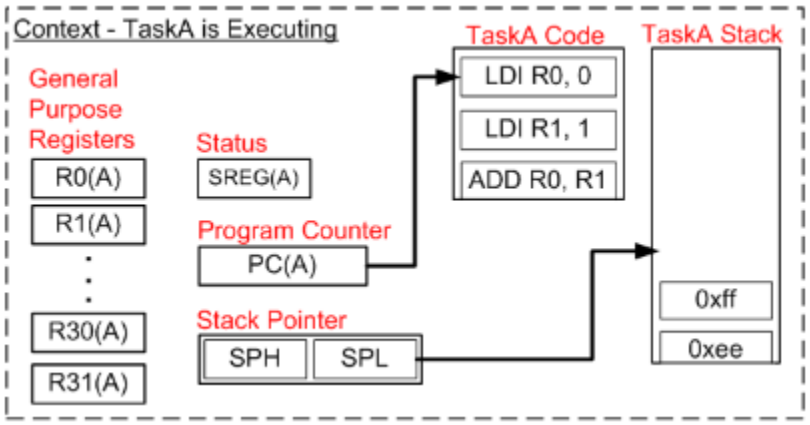
Returning to TaskA

4. Context Switching Back to TaskA:
    - When TaskB is done or another context switch occurs, TaskA might be scheduled to run again.
    - The context of the current task (e.g., TaskB) is saved onto its stack.
    - The stack pointer (SP) is switched back to TaskA's stack pointer.
    - TaskA's context is restored from its stack.
    - TaskA continues execution from where it left off.

Assume, the process of switching from a lower priority task, called TaskA, to a higher priority task, called TaskB.

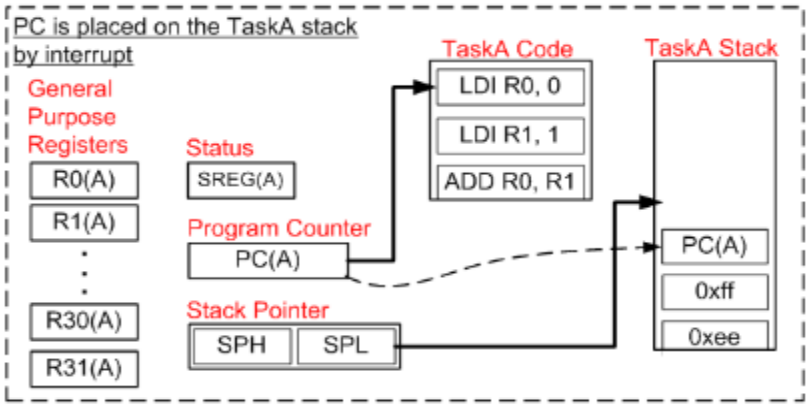
Step 1 :

- This example starts with TaskA executing. TaskB has previously been suspended so its context has already been stored on the TaskB stack.
- TaskA has the context demonstrated by the diagram below.



1. **General-Purpose Registers:** These are used by the CPU to perform various operations and hold temporary data during task execution.
2. **Status Register:** This register holds flags that indicate the state of the CPU (e.g., carry, zero, overflow flags).
3. **Program Counter (PC):** This register holds the address of the next instruction to be executed.
4. **Stack Pointer (SP):** This register points to the current top of the stack, which is used for storing temporary data, return addresses, and local variables.

- The (A) label within each register shows that the register contains the correct value for the context of task A.



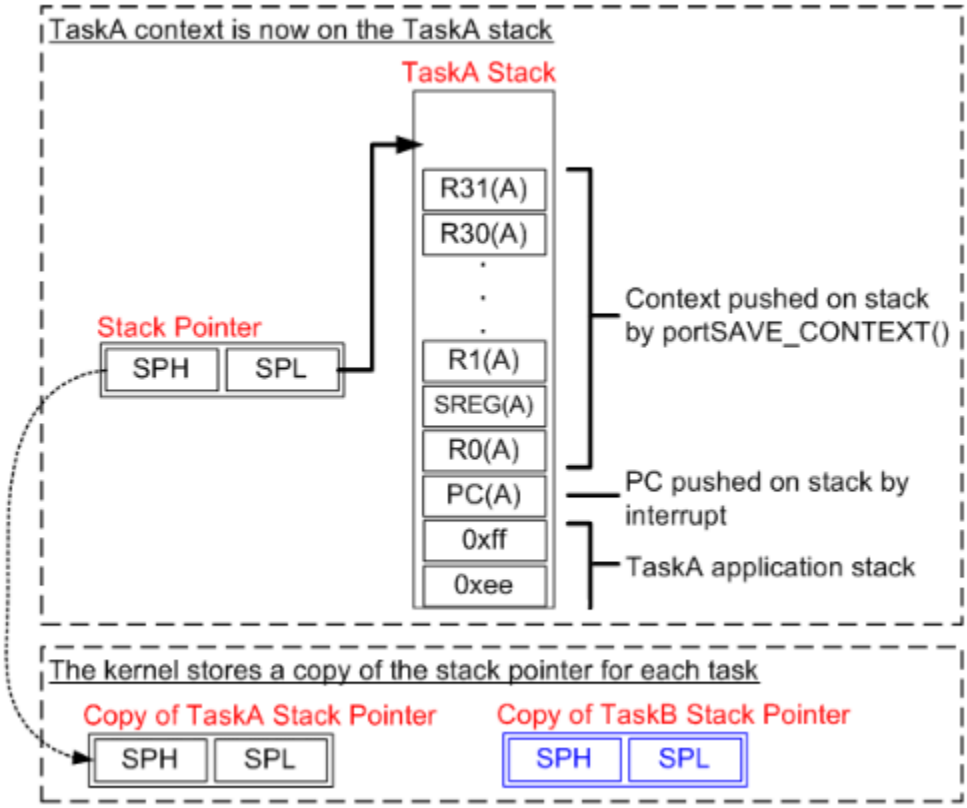
Step 2:

The RTOS tick interrupt occurs

The RTOS tick occurs just as TaskA is about to execute an LDI instruction. When the interrupt occurs the AVR microcontroller automatically places the current program counter (PC) onto the stack before jumping to the start of the RTOS tick ISR.

Step 3:

The RTOS tick interrupt executes



Step 4:

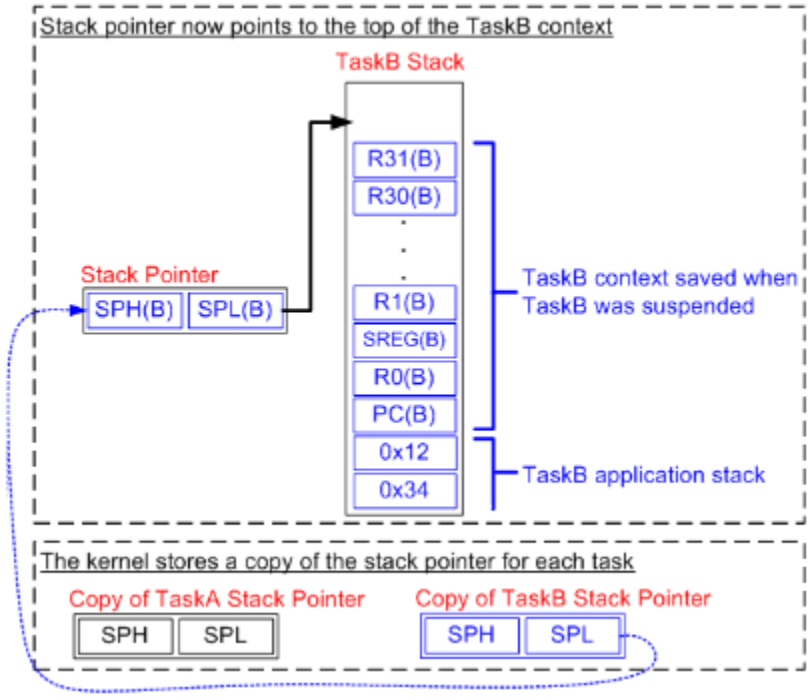
Incrementing the Tick Count

- The RTOS function **vTaskIncrementTick()** executes after the TaskA context has been saved. For the purposes of this example assume that incrementing the tick count has caused TaskB to become ready to run. TaskB has a higher priority than TaskA so **vTaskSwitchContext()** selects TaskB as the task to be given processing time when the ISR completes.

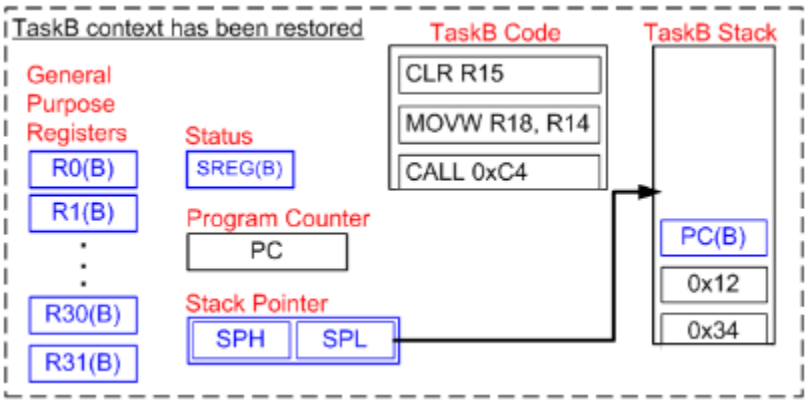
Step 5:

The TaskB stack pointer is retrieved

- The TaskB context must be restored. The first thing RTOS macro **portRESTORE_CONTEXT** does is retrieve the TaskB stack pointer from the copy taken when TaskB was suspended. The TaskB stack pointer is loaded into the processor stack pointer, so now the AVR stack points to the top of the TaskB context.



Step 6:



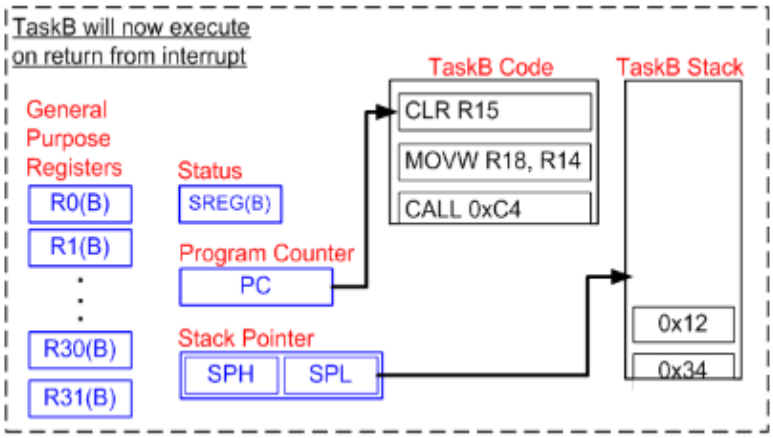
Restore the TaskB context

- portRESTORE_CONTEXT() completes by restoring the TaskB context from its stack into the appropriate processor registers.
- Only the program counter remains on the stack.

Step 7:

The RTOS tick exits

- **vPortYieldFromTick()** returns to SIG_OUTPUT_COMPARE1A() where the final instruction is a return from interrupt (RETI). A RETI instruction assumes the next value on the stack is a return address placed onto the stack when the interrupt occurred.
- The RTOS tick interrupted TaskA, but is returning to TaskB - the context switch is complete.



3.4. Scheduling Algorithms

- The task that is actually running (using processing time) is in the Running state. On a single core processor there can only be **one task in the Running state** at any given time.
- Synchronisation events occur when a task or interrupt service routine sends information using a task notification, queue, event group, or one of the many types of semaphore. They are generally used to signal asynchronous activity, such as data arriving at a peripheral.

3.4.1 Configuring the Scheduling Algorithm

1. The Scheduling algorithm is the software routine that decides which Ready state task to Running state.
2. The algorithm can be changed using the **configUSE_PREEMPTION** and **configUSE_TIME_SLICING** configuration constants. Both constants are defined in **FreeRTOSConfig.h.**
3. **configUSE_TICKLESS_IDLE** also affects the scheduling algorithm (this is the advanced option to minimise the power consumption.

4. In all possible configurations the FreeRTOS scheduler will ensure tasks that share a priority are selected to enter the Running state in turn. This **‘take it in turn’** policy is often referred to as **Round Robin Scheduling**.
5. A **Round Robin** scheduling algorithm does not guarantee time is shared equally between tasks of equal priority, only that Ready state tasks of equal priority will enter the Running state in turn.

Fixed Priority

- Do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority, or that of other tasks

Pre-emptive

- Pre-emptive scheduling algorithms will immediately ‘pre-empt’ the Running state task if a task that has a priority higher than the Running state task enters the Ready state.
- Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state.

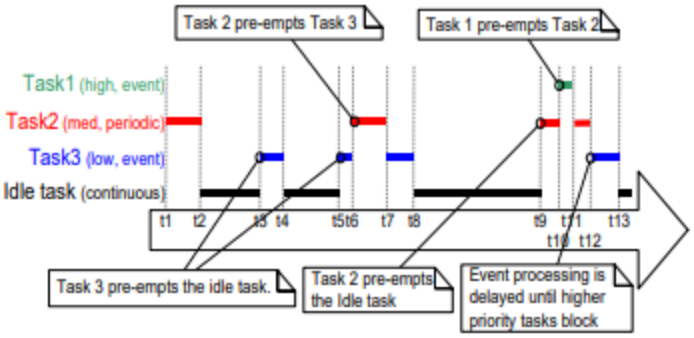
Time Slicing

- Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state.
- A time slice is equal to the time between two RTOS tick interrupts.

3.4.2 Prioritized Pre-emptive Scheduling with Time Slicing

configUSE_PREEMPTION → 1
configUSE_TIME_SLICING → 1

- **The Sequence in which tasks are selected to enter the Running state when all the tasks in an application have a unique priority.**

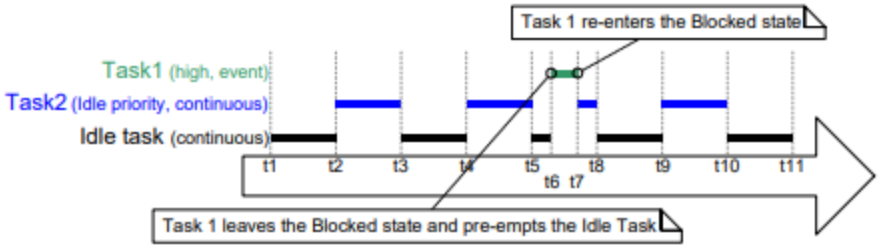


- **Idle Task** : The idle task is running at the lowest priority, so gets pre-empted every time a higher priority task enters the Ready state—for example, at times t3, t5 and t9.

- **Task 3**
  - Task 3 is a low-priority event-driven task that waits for its event of interest, transitioning from the Blocked state to the Ready state. It executes at times t3 and t5, and between t9 and t12. Task 3 is the highest priority task, while higher priority tasks Task 1 and Task 2 continue to execute.
- **Task 2**
  - Task 2 is a periodic task with a higher priority than Task 3, but below Task 1. It executes at times t1, t6, and t9. Task 2 pre-empts Task 3, completes processing, and blocks at time t8, allowing Task 3 to continue processing.
- **Task 1**
  - Task 1 is also an event-driven task. It executes with the highest priority of all, so can pre-empt any other task in the system. The only Task 1 event shown occurs at time t10, at which time Task 1 pre-empts Task 2. Task 2 can complete its processing only after Task 1 has re-entered the Blocked state at time t11.

- **The sequence in which tasks are selected to enter the Running state when two tasks run at the same priority.**

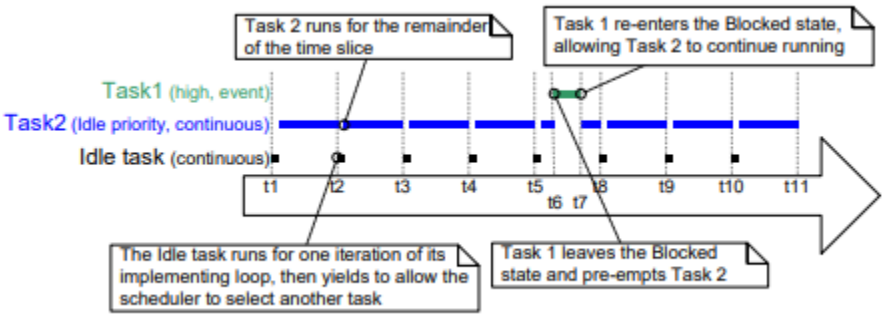
  When **configIDLE_SHOULD_YIELD** is set to 0 then,



- **The Idle Task and Task 2**
  - The Idle task and Task 2 are continuous processing tasks with a priority of 0 and are allocated processing time by time slicing. They enter the Running state in turn, potentially causing both tasks to be in the Running state for part of the same time slice.

- **Task 1**
  - Task 1 is an event-driven task with higher priority than Idle. It spends most time in the Blocked state waiting for its event, transitioning to the Ready state. At time t6, it becomes the highest priority task, preempting the Idle task.

When **configIDLE_SHOULD_YIELD** is set to 1

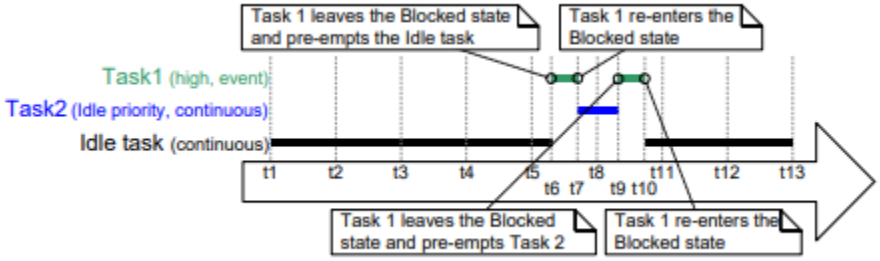


- The task selected to enter the Running state after the Idle task does not execute for an entire time slice, but instead executes for whatever remains of the time slice during which the Idle task yielded.

3.4.3 Prioritized Pre-emptive Scheduling (Without Time Slicing)

configUSE_PREEMPTION → 1
configUSE_TIME_SLICING → 0

6. If the time slicing is used, and there is more than one ready state task at the highest priority that is able to run, then the scheduler will select a new task to enter the Running state During each RTOS tick interrupt.
7. If the time slicing is not used, then the scheduler will only select a new task to enter the Running state when either :
    - A higher priority task enters the Ready state.
    - The task in the Running state enters the Blocked or Suspended state.
8. However, turning time slicing off can also result in tasks of equal priority receiving greatly different amounts of processing time , For this reason, running the scheduler without time slicing is considered an advanced technique .

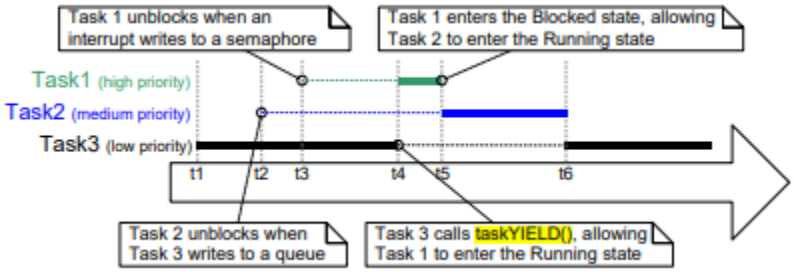


9. Above fig, Assume **configIDLE_SHOULD_YIELD** is set to 0
    - **Tick Interrupts**
        - Tick interrupts occur at times t1, t2, t3, t4, t5, t8, t11, t12 and t13.
    - **Task 1**
        - Task 1 is a high priority event driven task that spends most of its time in the Blocked state waiting for its event of interest. Task 1 transitions from the Blocked state to the Ready state each time the event occurs. Task 1 processing an event between times t6 and t7, then again between times t9 and t10.
    - **The Idle Task and Task 2**
        - The Idle task and Task 2 are continuous processing tasks with a priority of 0 and do not enter the Blocked state. Task 1 pre-empts the Idle task at time t6, after it has been in the Running state for over four tick periods. Task 2 also pre-empts the Idle task at time t9, after it has received more processing time than Task 2.

3.4.4 Co-operative Scheduling

configUSE_PREEMPTION → 0
configUSE_TIME_SLICING → Any value

10. When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule) by calling **taskYIELD()**. Tasks are never pre-empted, so time slicing cannot be used.



- Task 1
    - Task 1 is the highest priority, starting in the Blocked state and waiting for a semaphore. It exits the Blocked state at time t3 and enters the Ready state. Using the co-operative scheduler, Task 1 remains in the Ready state until time t4.

- Task 2
    - Task 2 is a priority between Task 1 and Task 3. It starts in the Blocked state, waiting for a message from Task 3. If a pre-emptive scheduler was used, Task 2 would become the Running state task. However, using the co-operative scheduler, it remains in the Ready state until Task 1 enters the Blocked state. Task 2 then re-enters the Blocked state.

4. Macros

4.1 pdMS_TO_TICKS()

- Convert a time specified in milliseconds to a time specified in ticks, which is used by the FreeRTOS kernel to manage time delays.

- #ifndef pdMS_TO_TICKS
  #define pdMS_TO_TICKS(xTimeInMs) ((TickType_t) (((TickType_t) (xTimeInMs) * (TickType_t) configTICK_RATE_HZ) / (TickType_t) 1000))
  #endif
- pdMS_TO_TICKS() in Target 1→ SourceGroup → main.c → projdefs.h.
- This macro multiplies the time in milliseconds by the tick rate (ticks per second) and then divides by 1000 to convert milliseconds to ticks. However, this conversion can lead to issues if the tick frequency (**configTICK_RATE_HZ**) is greater than 1000 Hz.

    - **Overflow and Precision Issues**
        1. **Overflow:**
            - When the tick rate is greater than 1000 Hz, multiplying xTimeInMs by configTICK_RATE_HZ can result in values that exceed the range of the TickType_t type, causing overflow.
            - **For example**, if xTimeInMs is a large value and configTICK_RATE_HZ is 2000, the result of xTimeInMs * 2000 might be too large to fit in a 32-bit integer, leading to incorrect results.
        2. **Precision:**
            - The division by 1000 is meant to convert from milliseconds to ticks. If configTICK_RATE_HZ is not a multiple of 1000, the division might lead to rounding errors.
            - When the tick rate is 1000 Hz or less, each millisecond conversion is straightforward and precise. However, for tick rates greater than 1000 Hz, the precision of the conversion might degrade.
    - **Practical Example :**
      Consider ‘configTICK_RATE_HZ’ is set to 2000 Hz(1tick = 0.5 milliseconds)
        1. pdMS_TO_TICKS(1000) = (1000 * 2000) / 1000 = 2000 ticks
            - This is fine.
        2. pdMS_TO_TICKS(1) = (1 * 2000) / 1000 = 2 ticks
            - Here, 1 millisecond corresponds to 2 ticks

3. For a very large time span like 1,000,000 milliseconds:
    - pdMS_TO_TICKS(1000000) = (1000000 * 2000) / 1000
    - This calculation results in 2000000 ticks, which might be too large for a 32-bit integer, causing overflow.

4.2 taskYIELD()

- **void taskYIELD(void);**
- Yield to another task of equal priority. Yielding is where a task volunteers to leave the Running state, without being pre-empted, and before its time slice has been fully utilized.
- **Parameters**: None
- **Return** : None

- **TaskYIELD() is a function that is only called from an executing task and should not be called during the scheduler's Initialization state. If a task calls it, another Ready state task of equal priority is selected to enter the Running state, ensuring the task is executed.**

4.3 taskDISABLE_INTERRUPTS()

- **void taskDISABLE_INTERRUPTS(void);**
- Disables all maskable interrupts(the processor can ignore these interrupts) globally.
- If the FreeRTOS port doesn't use the **configMAX_SYSCALL_INTERRUPT_PRIORITY** kernel configuration constant, calling **taskDISABLE_INTERRUPTS()** will disable interrupts globally. If it does, it will disable interrupts at and below the priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY, with higher priority interrupts enabled. If nesting is required, use **taskENTER_CRITICAL() and taskEXIT_CRITICAL().**
- **Parameters**: None
- **Return** : None

4.4 taskENABLE_INTERRUPTS()

- **void taskENABLE_INTERRUPTS(void);**
- Re-enables interrupts, effectively setting the interrupt priority level to its original state.
- **taskENABLE_INTERRUPTS()** will enable all interrupt priorities, but it's not designed for nesting. If nesting is needed, use **taskENTER_CRITICAL() and**

taskEXIT_CRITICAL(). Some FreeRTOS API functions use critical sections that re-enable interrupts if nesting is zero.
- **Parameters**: None
- **Return** : None

4.5 taskENTER_CRITICAL()

- **void taskENTER_CRITICAL(void);**
- Critical sections are entered by calling taskENTER_CRITICAL(), and subsequently exited by calling taskEXIT_CRITICAL().
- **Parameters**: None
- **Return** : None

4.6 taskEXIT_CRITICAL()

- **void taskEXIT_CRITICAL(void);**
- restores the interrupt enable state to what it was before the critical section was entered.
- They support nested critical sections. This means you can call taskENTER_CRITICAL() multiple times, and you must call taskEXIT_CRITICAL() the same number of times to actually re-enable interrupts.
- **Parameters**: None
- **Return** : None

4.7 Comparison of interrupt macros

Interrupt Handling:

- **taskENTER_CRITICAL() / taskEXIT_CRITICAL()**: Disable interrupts only up to a certain priority level, allowing high-priority interrupts to occur.
- **taskDISABLE_INTERRUPTS() / taskENABLE_INTERRUPTS()**: Disable all maskable interrupts globally, regardless of priority.

Nesting:

- **taskENTER_CRITICAL() / taskEXIT_CRITICAL():** Support nested critical sections. Multiple calls to taskENTER_CRITICAL() require an equal number of taskEXIT_CRITICAL() calls to re-enable interrupts.
- **taskDISABLE_INTERRUPTS() / taskENABLE_INTERRUPTS():** Do not support nested critical sections. A single call to taskENABLE_INTERRUPTS() re-enables interrupts regardless of how many times taskDISABLE_INTERRUPTS() was called.

Use Cases:

- **taskENTER_CRITICAL() / taskEXIT_CRITICAL():** Preferable for general-purpose critical sections within FreeRTOS tasks, especially when dealing with nested critical sections and when you need certain high-priority interrupts to still occur.
- **taskDISABLE_INTERRUPTS() / taskENABLE_INTERRUPTS()**: Suitable for very short, non-nested critical sections where it is essential to disable all interrupts globally.

5. Examples

5.1 Creating a Task and print it is created successfully or not

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vTask3 (void *pvvar);


void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}

void vApplicationIdleHook(void)
{
   vPrintString("Idle...\n");
}

int main(void)
{
   BaseType_t xReturned;

   vPrintString("Starting project\n");
   xReturned = xTaskCreate ( vTask3 ,"Task 1", 130, NULL,1,NULL);
   if (xReturned == pdPASS) {
      vPrintString("pdPASS\n");
   }
   vTaskStartScheduler();

   while (1)
```

```
        {

        }
}

void vTask3 (void *pvvar) {
   for (;;) {
      vPrintString("Task 1 is running...\n");
   }
}
```

Output :

```
Starting project
pdPASS
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
```

5.2 Get number of task by using uxTaskGetNumberOfTasks()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>


void vApplicationIdleHook(void)
{
}

TaskHandle_t t1 = NULL;
TaskHandle_t t2 = NULL;

void task1 (void *p){

   while(1){
      printf("%s",p);
      vTaskDelay(pdMS_TO_TICKS(1000));
   }
}

void task2 (void *p){

   while(1){
      BaseType_t TaskCount = uxTaskGetNumberOfTasks();
      printf("%s",p);
      printf("Number of tasks : %d\n",(int)TaskCount);
      vTaskDelay(pdMS_TO_TICKS(1000));

   }
}


int main(void)
{
   char* s1 = "lochu\n";
   char* s2 = "vara\n";
   xTaskCreate(task1,"task1",100,s1,1,&t1);
   xTaskCreate(task2,"task2",100,s2,2,&t2);
   /* Write your code here*/
   vTaskStartScheduler();
```

```
    printf("Starting project\n");
        while (1)
        {

        }
}
```

OUTPUT :

```
/*
vara
Number of tasks : 4
lochu
vara
Number of tasks : 4
lochu
vara
Number of tasks : 4
lochu
vara
Number of tasks : 4
lochu
*/
```

- The 4 tasks here running are
    - Task 1
    - Task 2
    - Idle task
    - Timer service task

5.3 Get Task Details by using uxTaskGetSystemState()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>


void vApplicationIdleHook(void)
{
}

TaskHandle_t t1 = NULL;
TaskHandle_t t2 = NULL;

void task1 (void *p){

   while(1){
      printf("%s",p);
      vTaskDelay(pdMS_TO_TICKS(1000));
   }
}

void task2 (void *p){

   while(1){
      printf("%s",p);
      vTaskDelay(pdMS_TO_TICKS(1000));

   }
}


void task3(void* pvParameters) {
   while(1){
      // Get the number of tasks
      UBaseType_t taskCount = uxTaskGetNumberOfTasks();
      printf("Number of tasks: %lu\n", (unsigned long)taskCount);

      // Iterate through tasks and print their details
      for (UBaseType_t i = 0; i < taskCount; i++) {
         TaskStatus_t taskStatusArray[taskCount];
         UBaseType_t taskCountActual = uxTaskGetSystemState(taskStatusArray,
taskCount, NULL);
```

```
            for (UBaseType_t j = 0; j < taskCountActual; j++) {
                TaskStatus_t *taskStatus = &taskStatusArray[j];
                printf("Task Name: %s\n", taskStatus->pcTaskName);
                printf("Task Handle: %p\n", (void*)taskStatus->xHandle);
                printf("Task State: %lu\n", (unsigned long)taskStatus->eCurrentState);
                printf("Task Priority: %lu\n", taskStatus->uxCurrentPriority);
                printf("Task Stack High Water Mark: %lu\n", (unsigned
long)taskStatus->usStackHighWaterMark);
            }
        }

        vTaskDelay(pdMS_TO_TICKS(5000));  // Print every 5 seconds
    }
}
int main(void)
{
    char* s1 = "lochu\n";
    char* s2 = "vara\n";
    xTaskCreate(task1,"task1",100,s1,0,&t1);
    xTaskCreate(task2,"task2",100,s2,1,&t2);
    xTaskCreate(task3, "task3",100,NULL, 2, NULL);
    /* Write your code here*/
    vTaskStartScheduler();
    printf("Starting project\n");
          while (1)
          {

          }
}
```

OUTPUT :

```
/*
Number of tasks: 5
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
```

Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 89
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 119
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 89
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 89
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 89
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 119
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233

Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 89
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 119
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 89
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 89
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 89
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0

Task Stack High Water Mark: 119
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 89
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 119
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 89
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
vara
lochu
vara
lochu
vara
lochu
vara
lochu
vara
lochu
Number of tasks: 5
Task Name: task3
Task Handle: 20000620

Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 37
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 117
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 28
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 37
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 28
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 117
Task Name: Tmr Svc

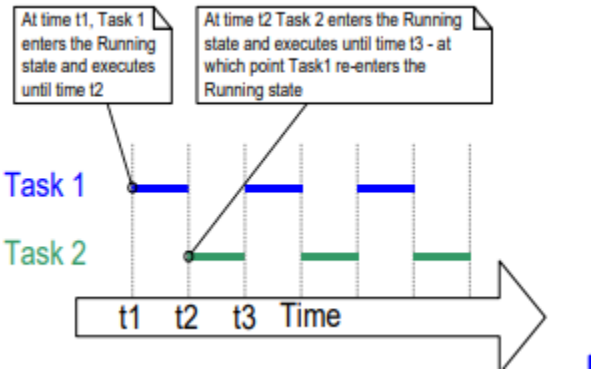
```
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 37
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 117
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 28
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 37
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 28
```

Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 117
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
Task Name: task3
Task Handle: 20000620
Task State: 0
Task Priority: 2
Task Stack High Water Mark: 34
Task Name: task2
Task Handle: 20000428
Task State: 1
Task Priority: 1
Task Stack High Water Mark: 37
Task Name: IDLE
Task Handle: 20000890
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 117
Task Name: task1
Task Handle: 20000230
Task State: 1
Task Priority: 0
Task Stack High Water Mark: 28
Task Name: Tmr Svc
Task Handle: 20000da0
Task State: 2
Task Priority: 2
Task Stack High Water Mark: 233
vara
lochu
vara
lochu
vara
lochu
vara
lochu
vara

lochu
*/

5.4 The two tasks appear to execute simultaneously

However, as both tasks are executing on the same processor core, this cannot be the case. In reality, both tasks are rapidly entering and exiting the Running state. Both tasks are running at the same priority, and so share time on the same processor core.




```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vTask1 (void *pvvar);
void vTask2 (void *pvvar);


void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

/*
void vApplicationIdleHook(void)
{
    vPrintString("Idle...\n");
```

```
}
*/

int main(void)
{
   xTaskCreate(vTask1,"Task 1", 130,NULL,1,NULL);
   xTaskCreate(vTask2,"Task 2", 130,NULL,1,NULL);
   vTaskStartScheduler();

   while (1)
         {

         }
}

// Implementation of task 1
void vTask1 (void *pvvar)
{
   for (;;) {
      vPrintString("Task 1 is running...\n");
      vTaskDelay(1000);
   }
}

// Implementation of task 2
void vTask2 (void *pvvar)
{
   for (;;) {
      vPrintString("Task 2 is running...\n");
      vTaskDelay(1000);
   }
}
```

OUTPUT :

```
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
```

```
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
```

5.5 Task 2 within Task1 and both the tasks are the same priorities.

Case 1 : When configUSE_TIME_SLICING is 0 and two tasks are having some delay.

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vTask2 (void *pvvar);
void vTask1 (void *pvvar);


void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}


int main(void)
{
   xTaskCreate(vTask1,"Task 1", 130,NULL,1,NULL);
   vTaskStartScheduler();
```

```
    while (1)
          {

          }
}

// Implementation of task 1
void vTask1 (void *pvvar)
{
     /* If this task code is executing then the scheduler must already have been started.
Create the other task before entering the infinite loop */
    xTaskCreate(vTask2,"Task 2", 130,NULL,1,NULL);
    for (;;) {
        vPrintString("Task 1 is running...\n");
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

// Implementation of task 2
void vTask2 (void *pvvar)
{
    for (;;) {
        vPrintString("Task 2 is running...\n");
        vTaskDelay(pdMS_TO_TICKS(250));
    }
}
```

Output :

```
Task 1Task 2 i is runns runninging...
...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running…
```

Case 2 : When configUSE_TIME_SLICING is 0 and two tasks are having no delay.

OUTPUT :

Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...
Task 1 is running...

Case 3 : When configUSE_TIME_SLICING is 1 and two tasks are having no delay.

OUTPUT :

Task 1Task 2 i is runns runninging...
...
Task Task 1 is1 is run runningning..Tas...
Taskk 1 is r 1 is rununning..ning...
.
Task 1 Task 1 iis Task s running2 is run...
Taskning...
2 is runTask 2 ining...
s runningTask 1 i...
Task s runnin2 is rung...
Taskning...
2 is ruTask 2 isnning... running
Task 1 i...
Tasks runnin 2 is rung...
Tasning...
k 2 is ruTask 2 inning...s running
Task 1 ...

Case 4 : When configUSE_TIME_SLICING is 1 and two tasks are having some delay.

OUTPUT :

```
Task 1Task 2 i is runns runninging...
...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running…
```

5.6 Use of the vTaskDelete()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vTask1 (void *pvvar);
void vfun1(void);


void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}


int main(void)
{
    vfun1();
```

```
    vTaskStartScheduler();

    while (1)
          {

          }
}

// Implementation of task 1
void vTask1 (void *pvvar)
{
     /* If this task code is executing then the scheduler must already have been started.
Create the other task before entering the infinite loop */
    for (;;) {
       vPrintString("Task 1 is running...\n");
       vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vfun1(void) {
    TaskHandle_t xH1;
    if ( (xTaskCreate(vTask1,"Task 1", 130,NULL,1,&xH1)) != pdPASS) {
             vPrintString("The task could not be created because there was not enough
FreeRTOS heap memory\n");
    }
    else {
      vPrintString("Delete the task just Created\n");
       vTaskDelete(xH1);
    }
    vTaskDelete(NULL);
}
```

Output : Delete the task just Created

5.7 Difference between the vTaskDelay() and vTaskDelayUntil()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vTask1 (void *pvvar);
void vTask2 (void *pvvar);


void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}


int main(void)
{
   xTaskCreate(vTask1,"Task1",130, NULL,1,NULL);
   xTaskCreate(vTask2,"Task2",130,NULL,1,NULL);

   vTaskStartScheduler();

   while (1)
         {

         }
}

void vTask1 (void *pvvar)
{
   for (;;) {
      vPrintString("Task 1 is running...\n");
      vTaskDelay(pdMS_TO_TICKS(500));
   }
}
```

```
void vTask2(void *pvvar) {
   TickType_t xLastWakeTime;
   const TickType_t xPeriod = pdMS_TO_TICKS(50);
   xLastWakeTime = xTaskGetTickCount();
   for (;;) {
      vPrintString("Task 2 is running...\n");
      vTaskDelayUntil(&xLastWakeTime,xPeriod);
   }
}
```

Output :

```
Task 1 Task 2 is runniis running...
ng...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 1 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
Task 2 is running...
```

5.8 Led blinking

Case 1 : Toggling led to a single task.

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vApplicationIdleHook(void)
{
}


void vTask1 (void *pvvar);
void vTask2 (void *pvvar);


void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

int main(void) {
    xTaskCreate(vTask1, "Task 1",130,NULL,1,NULL);
    xTaskCreate(vTask2, "Task 2",130,NULL,1,NULL);
    vTaskStartScheduler();
    for(;;) {
    }

}

void vTask1(void *pvvar) {
     // Enable clock for GPIOA peripheral
    RCC->AHB1ENR |= 1;

    // Configure GPIOA pin 5 as output
    GPIOA->MODER |= GPIO_MODER_MODER5_0;
    //GPIOA->MODER |= 1;
```

```
    for(;;) {
        vPrintString("Led blinking\n");

        // Toggle GPIOA pin 5
        //GPIOA->ODR |= GPIO_ODR_OD5_Msk;
        GPIOA->ODR ^= GPIO_ODR_OD5;

        // Delay for 1 second (100 milliseconds)
       vTaskDelay(100);
    }
}

void vTask2(void *pvvar) {
    for(;;) {
        vPrintString("Task 2 is running...\n");
        vTaskDelay(50);
    }
}
```

OUTPUT :

```
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
Led blinking
Task 2 is running...
Task 2 is running...
```

```
Led blinking
Task 2 is running...
Task 2 is running...
```

Case 2 : led on and led off within the different tasks

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vApplicationIdleHook(void)
{
}

void vTask1 (void *pvvar);
void vTask2 (void *pvvar);


void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}

int main(void) {
   xTaskCreate(vTask1, "Task 1",130,NULL,1,NULL);
   xTaskCreate(vTask2, "Task 2",130,NULL,1,NULL);
   vTaskStartScheduler();
   for(;;) {
   }

}

void vTask1(void *pvvar) {
   RCC->AHB1ENR |= 1;
   GPIOA->MODER |= GPIO_MODER_MODER5_0;

   for(;;) {
      vPrintString("Led on -> Task 1 is running... \n");
```

```
        GPIOA->ODR |= GPIO_ODR_OD5;
        vTaskDelay(100);
    }
}

void vTask2(void *pvvar) {
    RCC->AHB1ENR |= 1;
    GPIOA->MODER |= GPIO_MODER_MODER5_0;
    for(;;) {
        vPrintString("Led off -> Task 2 is running... \n");
        GPIOA->ODR &= GPIO_ODR_OD5;
        vTaskDelay(50);
    }
}
```

OUTPUT:

```
Led off -> Task 2 is running...
Led on -> Task 1 is running...
Led off -> Task 2 is running...
Led off -> Task 2 is running...
Led on -> Task 1 is running...
Led off -> Task 2 is running...
Led off -> Task 2 is running...
Led on -> Task 1 is running...
Led off -> Task 2 is running...
Led off -> Task 2 is running...
Led on -> Task 1 is running...
Led off -> Task 2 is running...
Led off -> Task 2 is running...
Led on -> Task 1 is running...
```

5.9 Use of the vTaskPrioritySet() and vTaskPriorityGet()

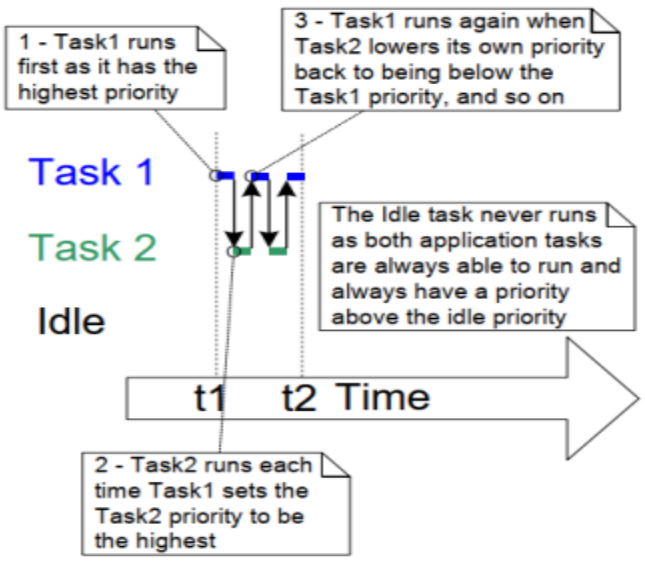
```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vApplicationIdleHook(void)
{
}

void vTask1 (void *pvvar);
void vTask2 (void *pvvar);


void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}
TaskHandle_t xTask2Handle = NULL;

int main(void) {
   xTaskCreate(vTask1, "Task 1",130,NULL,2,NULL);
   xTaskCreate(vTask2, "Task 2",130,NULL,1,&xTask2Handle);
   vTaskStartScheduler();
   for(;;) {
   }

}

void vTask1(void *pvvar) {
   UBaseType_t uxPriority;
   uxPriority = uxTaskPriorityGet(NULL);
   for(;;) {
      vPrintString("Task 1 is running... \n");
      vPrintString("About to raise the Task2 priority\n");
      vPrintString("Task 1 : %d\n",uxPriority);
      vTaskPrioritySet(xTask2Handle, (uxPriority + 1));
```

```
    }
}

void vTask2(void *pvvar) {
    UBaseType_t uxPriority;
    uxPriority = uxTaskPriorityGet(NULL);
    for(;;) {
        vPrintString("Task 2 is running... \n");
        vPrintString("About to lower the Task2 priority\n");
        vPrintString("Task 2 : %d\n",uxPriority);
// Task 1 to immediately start running again - preempting this task.
        vTaskPrioritySet(NULL, (uxPriority - 2));

    }
}
```

OUTPUT :

```
Task 1 is running...
About to raise the Task2 priority
Task 1 : 2
Task 2 is running...
About to lower the Task2 priority
Task 2 : 3
Task 1 is running...
About to raise the Task2 priority
Task 1 : 2
Task 2 is running...
About to lower the Task2 priority
Task 2 : 3
```

Explanation:



Task 1 is created with the highest priority, so it is guaranteed to run first. Task 1 prints the strings before raising the priority of Task 2 above its own priority.

Task 2 starts to run (enters the Running state) as soon as it has the highest relative priority. Only

one task can be in the Running state at any one time, so when Task 2 is in the Running state, Task 1 is in the Ready state.

Task 2 prints a message before setting its own priority back down to below that of Task 1.

When Task 2 sets its priority back down, then Task 1 is once again the highest priority task, so Task 1 reenters the Running state, forcing Task 2 back into the Ready state

5.10 Use of the vTaskSuspend() and xTaskResume()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void)
{
   // vPrintString("Idle..\n");
}

void vTask1 (void *pvvar);
void vfun1(void);
void vTask2 (void *pvvar);


TaskHandle_t xH1 = NULL;
TaskHandle_t xH2 = NULL;

int main(void) {

    xTaskCreate(vTask1, "Task 1",130,NULL,1,&xH1);
    vTaskStartScheduler();
```

```
    for(;;) {
    }

}


void vfun1(void) {

    if (xTaskCreate(vTask2, "Task 2",130,NULL,1,&xH2) != pdPASS) {
        vPrintString("The task was not created successfully\n");
    }
    else {
        /* Use the handle of the created task to place the task in the suspended state.
        This can be done before Scheduler has been started */
        taskENTER_CRITICAL();
        vPrintString("Suspended the task2 \n");
        vTaskSuspend(xH2);
        vPrintString("resumed the task2 \n");
        taskEXIT_CRITICAL();

        vTaskResume(xH2);
    }
}

void vTask1(void *pvvar) {
    vPrintString("Entered task 1\n");
    vfun1();
    for(;;) {
        vPrintString("Task 1 is running... \n");
        vTaskDelay(500);
    }
}


void vTask2(void *pvvar) {

    for(;;) {
        vPrintString("Task 2 is running... \n");
        vTaskDelay(1000);
    }
}
```

OUTPUT :

```
Entered task 1
Suspended the task2
resumed the task2
TaskTas 2 is ruknning..nning... .
ing..
.
Task 1 is running...
Task 2 is runninTask 1 g...
is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 1 is running...
Task 2 is running...
Task 1 is running...
Task 1 is running...
```

5.11 Use of the xTaskAbortDelay()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}

void vApplicationIdleHook(void)
{
```

```
   vPrintString("Idle..\n");
}

TaskHandle_t xH1 = NULL;

void vTask1 (void *pvvar);
void vfun1(TaskHandle_t xH1);
void vTask2 (void *pvvar);


int main(void) {

    xTaskCreate(vTask1, "Task 1",130,NULL,1,&xH1);
    vfun1(xH1);
    xTaskCreate(vTask2, "Task 2",130,NULL,1,NULL);
    vTaskStartScheduler();
    for(;;) {
    }

}

void vTask1(void *pvvar) {

    for(;;) {
        vPrintString("Task 1 is running... \n");
        vTaskDelay(100);
    }
}

void vfun1(TaskHandle_t xH1) {
    if(xTaskAbortDelay(xH1) == pdFAIL) {
        vPrintString("Task 1 was not in the Blocked state\n");
    }
    else {
        vPrintString("Task 1 was in the Blocked state, but is not now\n");
    }
}

void vTask2(void *pvvar) {
    for(;;) {
        vPrintString("Task 2 is running... \n");
        vTaskDelay(50);
    }
}
```

5.12 Use of the xTaskGetTickCount()

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h> // Include this for va_start

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void)
{
   // vPrintString("Idle..\n");
}

void vTask1 (void *pvvar);
void vfun1(void *pv1);
void vTask3(void *pvvar);

TaskHandle_t xH1;


int main(void) {

    xTaskCreate(vfun1, "Tfun",130,NULL,2,NULL);
    vTaskStartScheduler();
    for(;;) {
    }

}

void vTask1(void *pvvar) {

    for(;;) {
        vPrintString("Task 1 is running... \n");
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
```

```
}

void vfun1(void *pv1) {
   TickType_t xTime1, xTime2, xExecutionTime;

   // Get the time the function started.
   xTime1 = xTaskGetTickCount();
   vPrintString("fun start : %d\n",xTime1);

   xTaskCreate(vTask1, "Task 1",130,NULL,2,NULL);
   xTaskCreate(vTask3, "Task 3",130,NULL,2,NULL);


   xTime2 = xTaskGetTickCount();
   vPrintString("fun end : %d\n",xTime2);


   xExecutionTime = xTime2 - xTime1;
   vPrintString("Execution : %d\n",xExecutionTime);

   for(;;) {
      vTaskDelay(pdMS_TO_TICKS(1000));
   }

}

 void vTask3(void *pvvar) {

   for(;;) {
      vPrintString("Task 3 is running... \n");
      vTaskDelay(pdMS_TO_TICKS(1000));
   }
}
```

OUTPUT :

```
fun start : 0
fun end Task 3 i 1
Execut: : 1
ins : 1
inion : 1
gg...
...
ning...
```

```
Task 3 is runninTask 1 ig...
s running...
Task 3 is runninTask 1 ig...
s running...
Task 3 is runniTask 1 ing...
s running...
Task 3 is runninTask 1 g...
is running...
Task 3 is runniTask 1 ing...
s running...
Task 3 is runniTask 1 ng...
is running...
Task 3 is runniTask 1 ng...
is running...
Task 3 is runninTask 1 ig...
s running...
```

5.13 Use of the Idle Task and Idle Task Hook

- In FreeRTOS, the **idle task** is a special task that is created automatically when the scheduler is started. This task has the lowest priority and runs whenever no other task is able to run (i.e., all other tasks are either blocked or suspended). The idle task ensures the CPU is not left idle and can be used for low-priority background processing.
- The **idle task hook** is a mechanism that allows the application to execute a function each time the idle task runs. such as freeing memory or handling background operations.

```
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>
#include <stdarg.h>

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
```

```
}

// Idle Task Hook Function
void vApplicationIdleHook(void) {
    // This function will be called by the idle task.
    vPrintString("Idle task is running...\n");
}

// Task functions
void vTask1(void *pvParameters);
void vTask2(void *pvParameters);

int main(void) {
    // Create tasks
    xTaskCreate(vTask1, "Task 1", 130, NULL, 0, NULL);
    xTaskCreate(vTask2, "Task 2", 130, NULL, 1, NULL);

    // Start the scheduler
    vTaskStartScheduler();

    // Should never reach here
    for(;;);
}

// Task 1 function
void vTask1(void *pvParameters) {
    for(;;) {
        vPrintString("Task 1 is running...\n");
        vTaskDelay(100);
    }
}

// Task 2 function
void vTask2(void *pvParameters) {
    for(;;) {
        vPrintString("Task 2 is running...\n");
        vTaskDelay(50);
    }
}
```

OUTPUT :

```
Task 2 is running...
Task 1 is rIdle tau runninsk is rug.nning...
```

```
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
IdleTask 2 is running...
 2 is running...
..
Idle task is running...
Idle task is running...
Idle task is running...
Idle task is running...
```

5.14 Use of the taskDISABLE_INTERRUPTS() and taskENABLE_INTERRUPTS()

```
#include "FreeRTOS.h"
#include "task.h"

// Shared variable
volatile int sharedVar = 0;

void vTask1(void *pvParameters) {
    for (;;) {
        // Disable interrupts to enter a critical section
        taskDISABLE_INTERRUPTS();

        // Critical section: updating shared variable
        sharedVar++;

        // Enable interrupts to leave the critical section
        taskENABLE_INTERRUPTS();
```

```
        // Simulate some processing delay
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vTask2(void *pvParameters) {
    for (;;) {
        // Disable interrupts to enter a critical section
        taskDISABLE_INTERRUPTS();

        // Critical section: reading and modifying shared variable
        int localCopy = sharedVar;
        localCopy++;
        sharedVar = localCopy;

        // Enable interrupts to leave the critical section
        taskENABLE_INTERRUPTS();

        // Simulate some processing delay
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main(void) {
    // Create tasks
    xTaskCreate(vTask1, "Task 1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
    xTaskCreate(vTask2, "Task 2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

    // Start the scheduler
    vTaskStartScheduler();

    // The program should never reach here
    for (;;) {
    }
}
```

OUTPUT :

```
Task 1 incremented sharedVar to 1
Task 2 incremented sharedVar to 2
Task 1 incremented sharedVar to 3
Task 2 incremented sharedVar to 4
Task 1 incremented sharedVar to 5
```

```
Task 2 incremented sharedVar to 6
Task 1 incremented sharedVar to 7
Task 2 incremented sharedVar to 8
```

5.15 Use of the taskENTER_CRITICAL() and taskEXIT_CRITICAL()

```
#include "FreeRTOS.h"
#include "task.h"
#include <stdio.h>

volatile int sharedRegister = 0;

void vTaskWriteRegister(void *pvParameters) {
    for (;;) {
        taskENTER_CRITICAL();
        sharedRegister = 0xABCD;
        printf("Task Write: Register = 0x%X\n", sharedRegister);
        taskEXIT_CRITICAL();
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

void vTaskReadRegister(void *pvParameters) {
    for (;;) {
        taskENTER_CRITICAL();
        int regValue = sharedRegister;
        printf("Task Read: Register = 0x%X\n", regValue);
        taskEXIT_CRITICAL();
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main(void) {
    xTaskCreate(vTaskWriteRegister, "WriteTask", configMINIMAL_STACK_SIZE, NULL,
1, NULL);
    xTaskCreate(vTaskReadRegister, "ReadTask", configMINIMAL_STACK_SIZE, NULL,
1, NULL);

    vTaskStartScheduler();
    for (;;) {
    }
}
```

OUTPUT :

```
Task Write: Register = 0xABCD
Task Read: Register = 0xABCD
Task Write: Register = 0xABCD
Task Read: Register = 0xABCD
Task Write: Register = 0xABCD
Task Read: Register = 0xABCD
```