Real-time Embedded systems Lab3

**Task Management – Idle Task and Idle Task Hook, changing the task priority**

Example 7 Defining an idle task hook function

The idle task is created automatically when the kernel is started. It executes whenever there are no application tasks wishing to execute, to ensure there is always at least one task that is able to run. It can be used to measure spare processing capacity, to perform background checks, or simply to place the processor into a low-power mode.

It is possible to add application specific functionality directly into the idle task by using an idle hook function, which is called automatically by the idle task once per iteration of the idle task loop. In FreeRTOS, idle task hook functions must have the name and prototype as the following

*void vApplicationIdleHook(void);*

Such functions must adhere to the following rules:

* An idle task hook function must **never attempt to block or suspend**. The idle task will execute only when no other tasks are able to do so.
* If the application uses the *vTaskDelete()* library function, the idle task hook must always **return to its** **caller within a reasonable time period**. This is because the idle task is responsible for cleaning up kernel resources after a task has been deleted. If the idle task remains permanently in the idle hook function, the required clean-up cannot occur.

Perform the following steps:

1. Open the *FreeRTOSConfig.h* header file, and define the macro *configUSE\_IDLE\_HOOK* in this file as 1 to enable using the idle hook function.
2. Open the main.c file.
3. Declare a global variable *ulIdleCycleCount* with the *unsigned long* type and initialize it as the value 0UL.
4. Within the *vApplicationIdleHook*() function, implement the simple functionality: increments the global counter *ulIdleCycleCount* by 1.
5. The *vTaskFunction()* function is similar to the one in Example 4. The main difference is that you call the *printf()* function within the infinite *for( ;; )* loop. This function requires three parameters and is used to print both a string variable (*pcTaskName*) and a number variable (*ulIdleCycleCount)* at the same time.
6. Build, run the project, and check the output at the Tera Term Window.

Example 8 Changing tasks priorities

We first introduce two API functions that will be used in this example.

* *vTaskPrioritySet*()

Its prototype is as

*void vTaskPrioritySet (xTaskHandle pxTask,*

*unsigned portBASE\_TYPE uxNewPriority);*

* *pxTask* is the handle of the task whose priority is being modified. A task can change its own priority by passing NULL in place of a valid task handle.
* *uxNewPriority* is 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 option set in the *FreeRTOSConfig.h* header file.
* *uxTaskPriorityGet*()

Its prototype is as

*unsigned portBASE\_TYPE uxTaskPriorityGet (xTaskHandle pxTask);*

It returns the priority currently assigned to the task *pxTask* specified in the input parameter.

The scheduler will always select the highest Ready state task as the task to enter the Running state. This part demonstrates this by using the *vTaskPrioritySet()* API function to change the priority of two tasks relative to each other.

Two tasks are crated at two different priorities. Neither task makes any API function calls that could cause it to enter the Block state, so both are always in either the Ready state or the Running state. As a result, the task with the highest relative priority will always be the task selected by the scheduler to be in the Running state.

Please perform the following steps.

1. Open the *FreeRTOSConfig.h* header file, Set two macros *INCLUDE\_vTaskPrioritySet* and *INCLUDE\_uxTaskPriorityGet* to 1 if they are not, so as to enable the usage of the *vTaskPrioritySet()* and *uxTaskPriorityGet()* APIs*.*

In the main.c file,

1. First, declare two global variables *xTask1Handle* and *xTask2Handle* with the type *xTaskHandle* as the handle of *vTask1()* and *vTask2()* respectively.

Within the function *vTask1()*,

1. First declare a local variable *uxPriority* with the type unsigned portBASE\_TYPE.
2. Before entering the infinite *for(;;)* loop of the function *vTask1(),*call *uxTaskPriorityGet*() to access its own priority (by passing the input parameter NULL) and assign the returned value to the variable *uxPriority*.
3. Within the infinite *for(;;)* loop, call *vTaskPrioritySet*() with two parameters to raise the priority of *vTask2()*

* The 1st parameter is set as *xTask2Handle*;
* The 2nd one is set as a higher priority which is equal to its own priority obtained in step 6 **plus 1**.

Within the function *vTask2(),*

1. First declare a local variable *uxPriority* as in vTask1 ().
2. Before entering the infinite *for(;;)* loop, call *uxTaskPriorityGet*() to access the priority of *vTask1* (by pass the input parameter as *xTask1Handle*).
3. Within the *for(;;)* loop*,* call *vTaskPrioritySet*()with two parameters to lower the priority of *vTask2()* itself.

* The 1st parameter is set as *xTask2Handle*;
* The 2nd one is set as a lower priority which is equal to the priority of *vTask1()* obtained in step 9 **minus 1**.

Note: both *vTask1()* and *vTask2()* are continuous processing tasks without using the null for loop or calling the *vTaskDelay()* API function to generate delay.

Within the function main(),

1. Create a task instance of *vTask1* with the high priority at 2, so this task is guaranteed to run first. Here, the first parameter passed to *xTaskCreate*() is *vTask1*, and the last one is &*xTask1Handle*.
2. Similarly, create a task instance of *vTask2* with the low priority at 1. The first parameter passed to *xTaskCreate*() is *vTask2* and the last one is &*xTask2Handle*.
3. Build, run the project, and check the output strings in the *Tera* *Terminal Window*.

Example 9 Deleting tasks

In this part, we will make use of the *vTaskDelete(xTaskHandle pxTaskToDelete)* API function which is used by a task to delete itself or any other task. As it is the responsibility of the idle task to free memory allocated to tasks that have since been deleted, it is important that applications using the *vTaskDelete*() do not completely starve the idle task of all processing time.

Note that only memory allocated to a task by the kernel itself will be freed automatically when the task is deleted. Any memory or other resource that the implementation of the task allocates itself must be freed explicitly.

Please modify the code to implement the expected execution behavior as the following.

In the FreeRTOSConfig.h header file

1. Ensure that the macro *INCLUDE\_vTaskDelete* is setto 1 to enable using vTaskDelete() API function.

In the main.c file,

1. Within the *main()* function, create a task instance Task 1 of vTask1 at the priority 1.
2. Within the *vTask1()* function, create another task instance Task 2 of vTask2 at the priority 2.

Thus, when Task 1 runs, it creates Task 2 which is the highest priority task. Then Task 2 can start to execute immediately.

1. Within the *vTask2()* function, it does nothing but delete itself. It calls *vTaskDelete()* and passes NULL -- the task handle of vTask2 itself -- as input parameter.
2. When Task 2 has been deleted, Task 1 is again the highest priority task, so continues executing – at which point it calls vTaskDelay() to block for a short period.
3. The idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.
4. When Task 1 leaves the blocked state, it again becomes the highest priority Ready state task and so preempts the idle task. When it enters the Running state, it creates Task 2 again, and so it goes on.

The expected execution pattern is illustrated as below.

