Real-time Embedded Systems Lab Manual

Thread Management in FreeRTOS – Part 4 (Lab 7 - 9)

Thread Management – Idle Thread and Idle Thread Hook, changing the Thread priority

Lab 7 Defining an idle thread hook function

The idle thread is created automatically when the kernel is started. It executes whenever there are no application threads wishing to execute, to ensure there is always at least one thread 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 thread by using an idle hook function, which is called automatically by the idle thread once per iteration of the idle thread loop. In FreeRTOS, idle thread hook functions must have the name and prototype as the following

void vApplicationIdleHook( );

Such functions must adhere to the following rules:

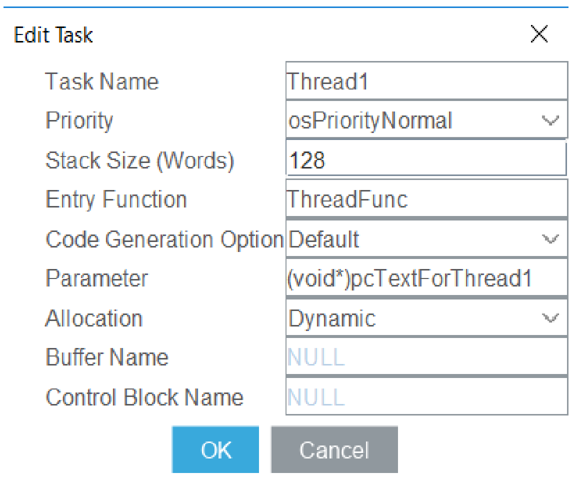
- An idle thread hook function must **never attempt to block or suspend**. The idle thread will execute only when no other threads are able to do so.
- If the application uses the *osThreadTerminate()* library function, the idle thread hook must always **return to its caller within a reasonable time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

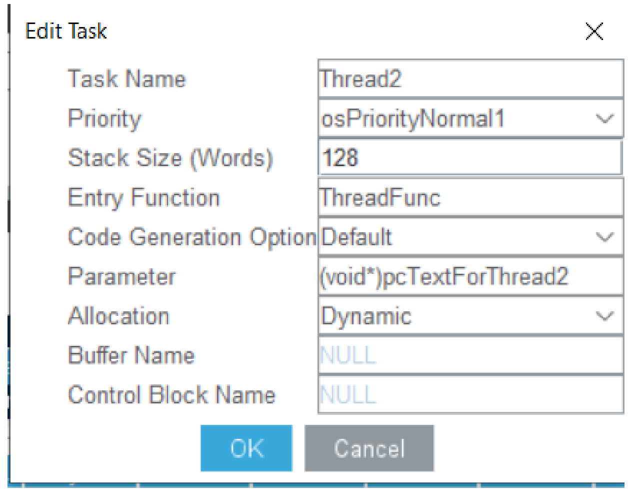
Major programming steps:

1. Assume that we use the MCU configuration file of lab4 as a start point and copy it to a new directory for lab7. You could also use the configuration file of any previous lab. Just make sure the following modes are set:
   a) The debug **Mode** of the **SYS** module under **System Core** category is set **Serial Wire**;
   b) The **Mode** of the **USART1** module under the **Connectivity** category is set **Asynchronous;**
   c) The **interface** of the **FreeRTOS** under **Middleware** category is set **CMSIS_V2**.

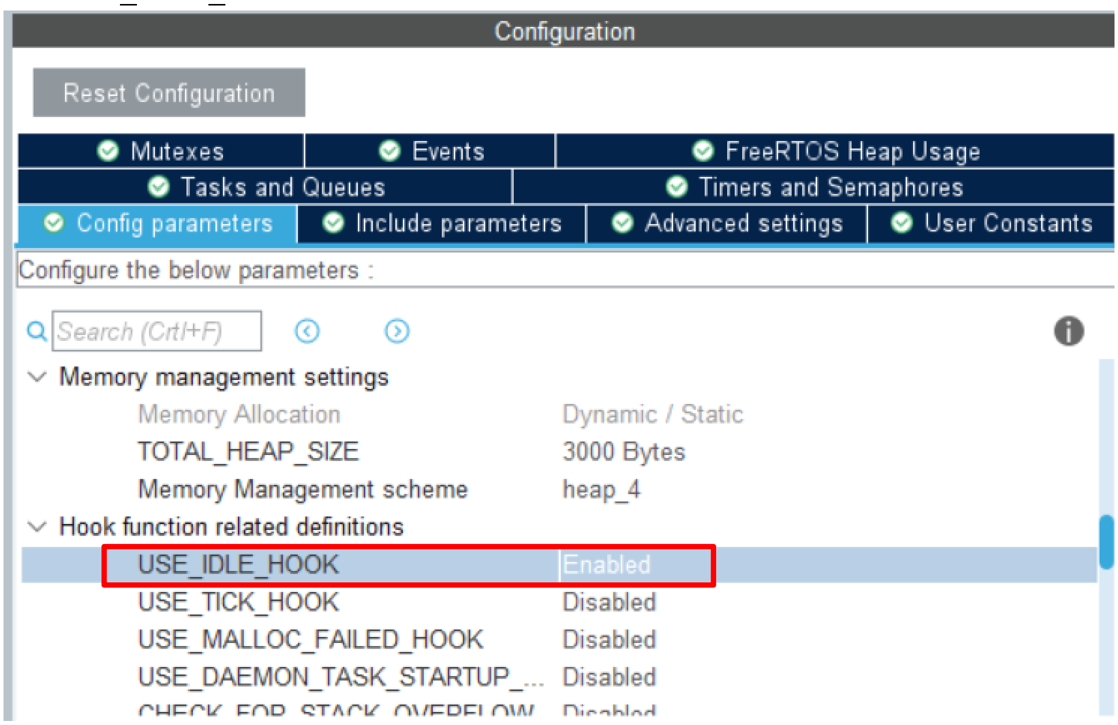
   If you did not save the MCU configuration file, please follow the step 1 – 4 in the first lab manual *ThreadLab_p1_CMSISv2.docx*.

2. Open the FREERTOS **Mode and Configuration** panels; select **Tasks and Queues** tab; Edit/Create two tasks as following.





3. Select the **Config parameters** tab under the FreeRTOS Configuration of STM32CubeMX. Set "USE_IDLE_HOOK" Enabled.



4. Select the **User Constants** tab under the FreeRTOS Configuration of STM32CubeMX. Revise the values of two constants pcTextForThread1 and pcTextForThread2.
Name: pcTextForThread1 Value: "\n\rThread 1 count = %ld\n\r"
Name: pcTextForThread2 Value: "\n\rThread 2 count = %ld\n\r"
Note: the constant mainDELAY_LOOP_COUNT is not used in this lab.

Config parameters | Include parameters | Advanced settings | User Constants

Search Constants

Search (Crtl+F)

add remove

Constant Name	Constant Value
pcTextForThread2	"\n\rThread 2 count = %d\n\r"
pcTextForThread1	"\n\rThread 1 count = %d\n\r"
mainDELAY_LOOP_COUNT	0xffffff

5. Save the configuration file to **Generate Code**.

6. Edit the main.c file.

   a) Include the header file stdio.h since this lab will call a library function declared in this header called *sprintf()*.

```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */
```

   b) Define putchUSART1() and putsUSART1() to use the USART1 channel.

```
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

void putsUSART1 (char* ptr)
{
    while(*ptr)
    {
        putchUSART1(*ptr++);
    }
}
/* USER CODE END 0 */
```

   c) Edit the main() function by calling a putsUSART1() statement as below.

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS lab 7\n\r");
  /* USER CODE END 2 */
```

d) Within /* USER CODE BEGIN PV */ and /* USER CODE END PV */, declare a **global** variable *ulIdleCycleCount* with the *unsigned long* type and initialize it as the value 0UL; And declare another global variable *buffer* with the char type array, the array size could be set as 200.

```
/* USER CODE BEGIN PV */
unsigned long ulIdleCycleCount = 0UL;
char buffer[200];
/* USER CODE END PV */
```

e) Within /* USER CODE BEGIN 0 */ and /* USER CODE END 0 */, besides the redefinition of PUTCHAR_PROTOTYPE, we define another new function *vPrintStringAndNumber()* to apply the mutual exclusion to printing a string variable and a number variable by **blocking** the scheduler with *osKernelLock*() and *osKernelUnlock*():

```
void vPrintStringAndNumber(char const *pcString, unsigned long ulValue)
{
  osKernelLock();
  {
        sprintf(buffer, pcString, ulValue);
        putsUSART1(buffer);
  }
  osKernelUnlock();
}
```

f) This *ThreadFunc()* function definition is similar to the one in lab4. The main difference is that you call the *vPrintStringAndNumber()* function within the infinite *for(;;)* loop. The two parameters that we apply are a string variable *(char *)argument* and a number variable *ulIdleCycleCount*.

```
void ThreadFunc(void const * argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    vPrintStringAndNumber((char *)argument, ulIdleCycleCount);
    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

7. Edit the freertos.c file.

   a) Declare the external global variable defined in another file (i.e., main.c).

```
/* Private variables -----------------------
/* USER CODE BEGIN Variables */
extern unsigned long ulIdleCycleCount;
/* USER CODE END Variables */
```

b) *Edit the vApplicationIdleHook*() function with simple functionality: increments the global counter *ulIdleCycleCount* by 1.

```
/* Hook prototypes */
void vApplicationIdleHook(void);

/* USER CODE BEGIN 2 */
void vApplicationIdleHook( void )
{
   /* vApplicationIdleHook() will only be called if configUSE_IDLE_HOOK is set
   to 1 in FreeRTOSConfig.h. It will be called on each iteration of the idle
   task. It is essential that code added to this hook function never attempts
   to block in any way (for example, call xQueueReceive() with a block time
   specified, or call vTaskDelay()). If the application makes use of the
   vTaskDelete() API function (as this demo application does) then it is also
   important that vApplicationIdleHook() is permitted to return to its calling
   function, because it is the responsibility of the idle task to clean up
   memory allocated by the kernel to any task that has since been deleted. */
  ulIdleCycleCount++;
}
/* USER CODE END 2 */
```

c)

8. Build, run the project, and check the output at the Tera Term Window.

Questions:
1. How many CMSIS-RTOSv2 API functions are called in the main.c file? Please list them and briefly summarize their usage.
2. After you enable USE_TICK_HOOK in the MCU configuration file, regenerate the code and move the variable *ulIdleCycleCount* increment statement to *vApplicationTickHook()* function. What are the display messages in the Tera Term?
Please give your answers in the lab report.

Lab 8 Changing threads priorities

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

a) *osThreadSetPriority*()

Its prototype is as

osStatus osThreadSetPriority (osThreadId thread_id, osPriority priority)

- *thread_id* is obtained by osThreadCreate() or osThreadGetId(). A thread can change its own priority by passing **NULL** in place of a valid thread ID.
- *priority* is the new priority for the thread. 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.
- Return type is the status code *osStatus* that indicates the execution status of the function.

b) *osThreadGetPriority*()

Its prototype is as

osPriority osThreadGetPriority (osThreadId thread_id)

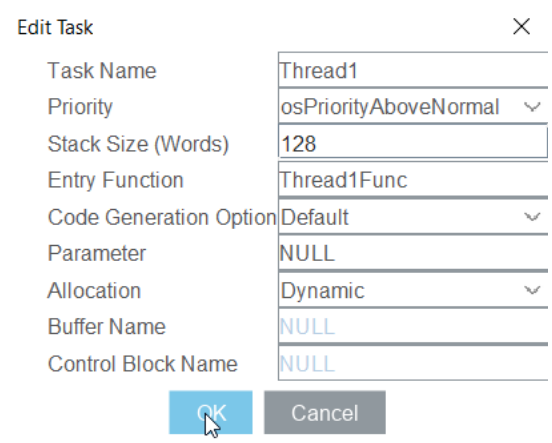
It returns the priority currently assigned to the thread with ID *thread_id* specified in the input parameter.

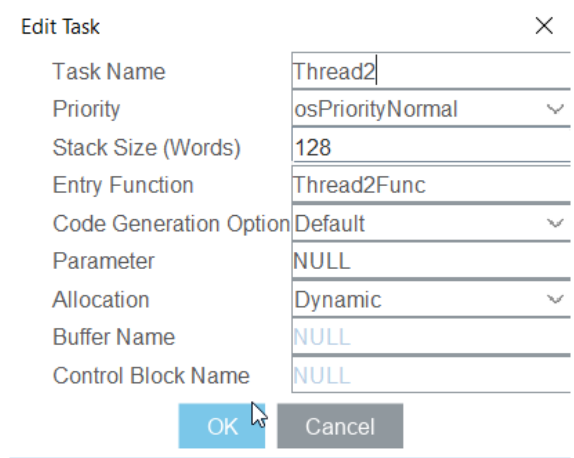
The scheduler will always select the highest Ready state thread as the thread to enter the Running state. This part demonstrates this by using the *osThreadSetPriority()* API function to change the priority of two threads relative to each other.

Two threads are created at two different priorities. Neither thread 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 thread with the highest relative priority will always be the thread selected by the scheduler to be in the Running state.

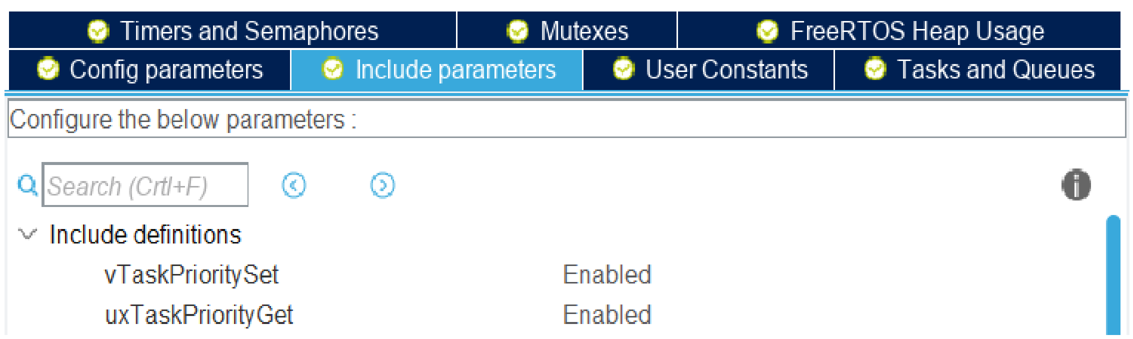
1. Assume that you reuse the MCU configuration file of lab7 as a start point.

2. Open the FREERTOS **Mode and Configuration** panels; select **Tasks and Queues** tab; Edit/Create two tasks as following.
   Note:





3. Select the *Include parameters* tab. Make sure that the two Include definitions*: vTaskPrioritySet* and *uxTaskPriorityGet* are enabled.



4. Select the **User Constants** tab. Make sure there exists a constant **mainDELAY_LOOP_COUNT** with the value **0xffffff.** If not, you could create one.
5. Save the configuration file to generate the code.

6. Edit the main.c file,
   a) Define *putchUSART1()* and *putsUSART1()* to display debug message via the USART1 channel in the main.c.

```c
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

void putsUSART1 (char* ptr)
{
    while(*ptr)
    {
        putchUSART1(*ptr++);
    }
}
/* USER CODE END 0 */
```

b) Edit the main() function by calling putsUSART1() statement as below.

```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS lab 8\n\r");
  /* USER CODE END 2 */
```

c) Edit the function *Thread1Func()*,

i. Firstly, declare a local variable *uxPriority* with the type unsigned *osPriority_t*.

ii. Then, before entering the infinite *for(;;)* loop,call *osThreadGetPriority*() to access its own priority (by passing the input parameter Thread1Handle) and assign the returned value to the variable *uxPriority*.

iii. Within the *for(;;)* loop, add a null *for* loop delay as in lab1.

iv. Then, in the *for(;;)* loop, call *osThreadSetPriority* () with two parameters to raise the priority of *Thread2;*

- The 1st parameter is set as *Thread2Handle*;
- The 2nd one is set as a higher priority which is equal to its own priority plus 1 - **uxPriority + 1**.
-

```c
/* USER CODE END Header_Thread1Func */
void Thread1Func(void *argument)
{
  /* USER CODE BEGIN 5 */
    volatile unsigned long ul;

    osPriority_t uxPriority;
    uxPriority = osThreadGetPriority(Thread1Handle);
  /* Infinite loop */
  for(;;)
  {
      putsUSART1("\n\rThread1 is running: Raise the priority of Thread2\n\r");
      for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {
          /*
           * This loop is just a crude delay implementation.
           * There is nothing to do in here.*/
      }
      /*
       * Setting the Thread2 priority above the Thread1 priority will cause
       * Thread2 to immediately start running (as then Thread2 will have the
       * higher priority among two created thread instances.)
       * */
      osThreadSetPriority(Thread2Handle, uxPriority + 1);
      //osDelay(1);
  }
  /* USER CODE END 5 */
}
```

d) Edit the function *Thread2Func(),*

  i. Declare a local variable *uxPriority* as in *Thread1Func*().
  ii. Before entering the infinite *for(;;)* loop, call *osThreadGetPriority*() to access the priority of *Thread1* (by passing the input parameter as *Thread1Handle*).
  iii. Within the *for(;;)* loop, add a null for loop delay as in lab1.
  iv. Then, in the *for(;;)* loop, call *osThreadSetPriority*()with two parameters to lower the priority of *Thread2Func()* itself.
    - The 1st parameter is set as *Thread2Handle*;
    - The 2nd one is set as a lower priority which is equal to the priority of *Thread2* minus 1**: uxPriority - 1**.

```
void Thread2Func(void *argument)
{
  /* USER CODE BEGIN Thread2Func */
    volatile unsigned long ul;

    osPriority_t uxPriority;
    uxPriority = osThreadGetPriority(Thread1Handle);
  /* Infinite loop */
  for(;;)
  {
      putsUSART1("\n\rThread2 is running: Lower the priority of itself\n\r");
      for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {
          /*
           * This loop is just a crude delay implementation.
           * There is nothing to do in here.*/
      }
      /*
       * Setting the Thread2 (itself)'s priority below the Thread1 priority will cause
       * Thread2 to immediately be preempted (as then Thread1 will have the
       * higher priority among two created thread instances.)
       * */
      osThreadSetPriority(Thread2Handle, uxPriority - 1);

    //osDelay(1);
  }
  /* USER CODE END Thread2Func */
}
```

Note: both *Thread1Func()* and *Thread2Func()* are **continuous** processing threads as the null for loop instead of calling the *osDelay()* API function is used to generate delay.

7. Build, run the project, and check the print output in the *Tera Terminal Window*.

Questions:

1. How many CMSIS-RTOSv2 API functions are called in the main.c file? Please list them and briefly summarize their usage.
2. After you enable USE_TICK_HOOK in the MCU configuration file, regenerate the code and move the variable *ulIdleCycleCount* increment statement to *vApplicationTickHook()* function. What are the display messages in the Tera Term?

Please give your answers in the lab report.

Questions:

1. How many CMSIS-RTOSv2 API functions are called in the main.c file? Please list them and briefly summarize their usage.
2. Set a breakpoint at the osThreadGetPriority() statement. Step into this function and check which FreeRTOS API function(s) are called?
3. Set a breakpoint at the osThreadSetPriority() statement. Step into this function and check which FreeRTOS API function(s) are called?

Please give your answers in the lab report.

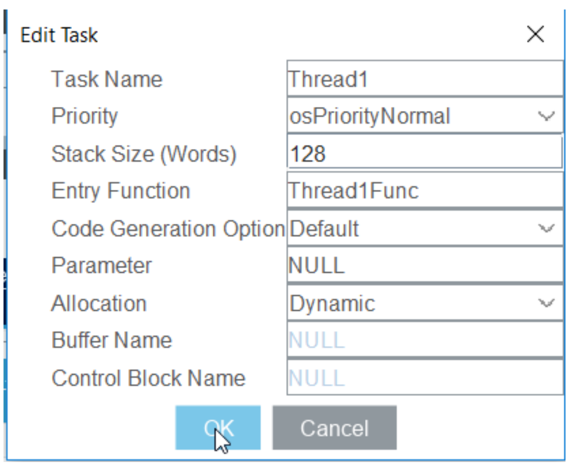
Lab 9 Deleting threads

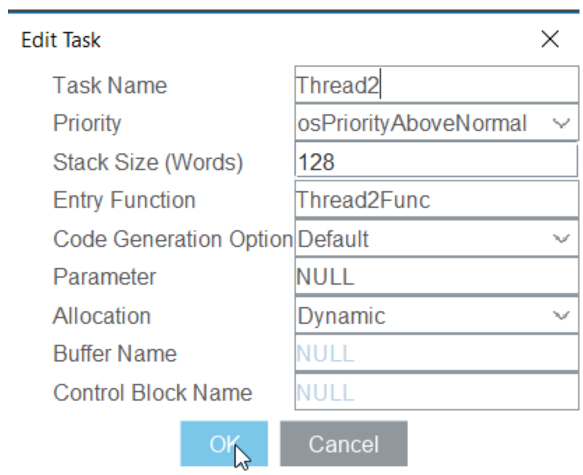
In this part, we will make use of the *osThreadTerminate(osThreadId thread_id)* API function which is used by a thread to delete itself or any other thread. As it is the responsibility of the idle thread to free memory allocated to threads that have since been deleted, it is important that applications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

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

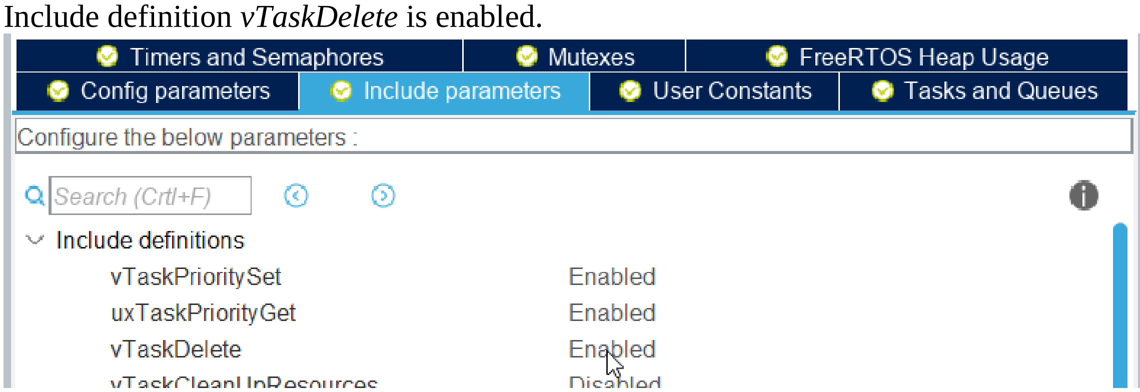
Please modify the configuration and the ode to implement the expected execution behavior as the following.

1. Assume that you reuse the MCU configuration file of lab8 as a start point and copy it to a new directory for lab9.
2. Open the FREERTOS **Mode and Configuration** panels; select **Tasks and Queues** tab; Edit/Create two tasks as following.
   Note: They have the similar settings as two tasks in lab8, but the priority levels are switched.





3. Select the *Include parameters* tab under FreeRTOS Configuration panel. Make sure that the Include definition *vTaskDelete* is enabled.



4. Generate the C project in the chosen IDE.

5. Edit the main.c file,
   a) Define *putchUSART1() and putsUSART1()* to display debugging message via the USART1 channel in the main.c similar to that in lab8.
   b) Edit the main() function by calling *putsUSART1()* as below.

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS lab 9\n\r");
  /* USER CODE END 2 */
```

   c) Copy the statement of creating Thread2 in the main() function to the Thread1Func().

```
  /* Create the thread(s) */
  /* creation of Thread1 */
  Thread1Handle = osThreadNew(Thread1Func, NULL, &Thread1_attributes);

  /* creation of Thread2 */
  Thread2Handle = osThreadNew(Thread2Func, NULL, &Thread2_attributes);
```

```
/* USER CODE END Header_Thread1Func */
void Thread1Func(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {



    osDelay(1);
  }
  /* USER CODE END 5 */
}
```

   d) Thread1Func() looks like the following

```
/* USER CODE END Header_Thread1Func */
void Thread1Func(void *argument)
{
  /* USER CODE BEGIN 5 */
    int32_t state;
  /* Infinite loop */
  for(;;)
  {
    putsUSART1("\n\rThread1 is running and creating Thread2\n\r");

    /* creation of Thread2 */
    state = osKernelLock();
    Thread2Handle = osThreadNew(Thread2Func, NULL, &Thread2_attributes);
    osKernelRestoreLock(state);

    osDelay(1000);
  }
  /* USER CODE END 5 */
}
```

- In this way, within the *Thread1Func()* function, we create another thread Thread2 at the higher priority **osPriorityAboveNormal**.
- Thus, when Thread1 runs, it creates Thread2 which is the highest priority thread. Then Thread2 can start to execute immediately.

Note, we declare a local variable called *state* before the infinite loop. Within the infinite loop, this variable holds the return value of *osKernelLock()* function, and is later used as the parameter of the *osKernelRestoreLock()*. The pair of o*sKernelLock()* and *osKernelRestoreLock()* is commonly applied to ensure the critical code in between can complete atomically without preemption. Without such protection, the variable *Thread2Handle* might be NULL even the creation of Thread2 is successful.

e) Within the *Thread2Func()* function, it does nothing but delete itself. It calls *osThreadTerminate()* and passes NULL or *Thread2Handle* -- the Thread handle of Thread2 itself -- as input parameter.

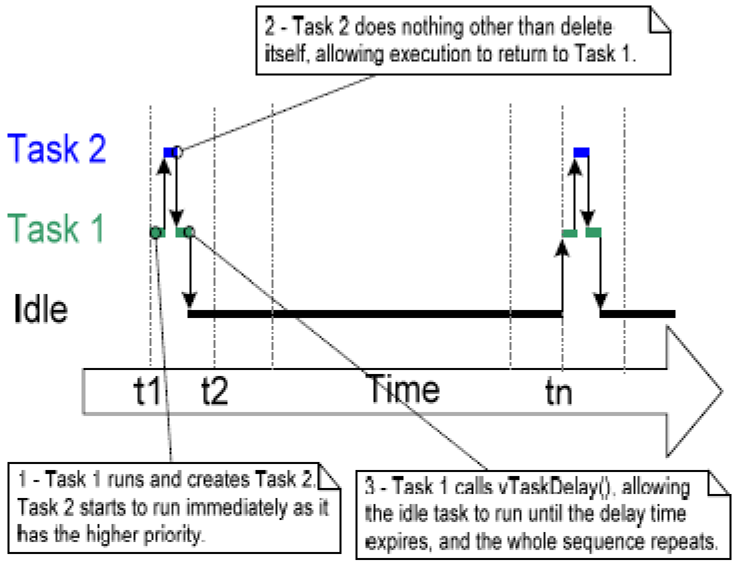
```
/* USER CODE END Header_Thread2Func */
void Thread2Func(void *argument)
{
  /* USER CODE BEGIN Thread2Func */
  /* Infinite loop */
//  for(;;)
//  {
//    osDelay(1);
//  }
    putsUSART1("\n\rThread2 is running and about to delete itself.\n\r");

    osThreadTerminate(Thread2Handle);
  /* USER CODE END Thread2Func */
}
```

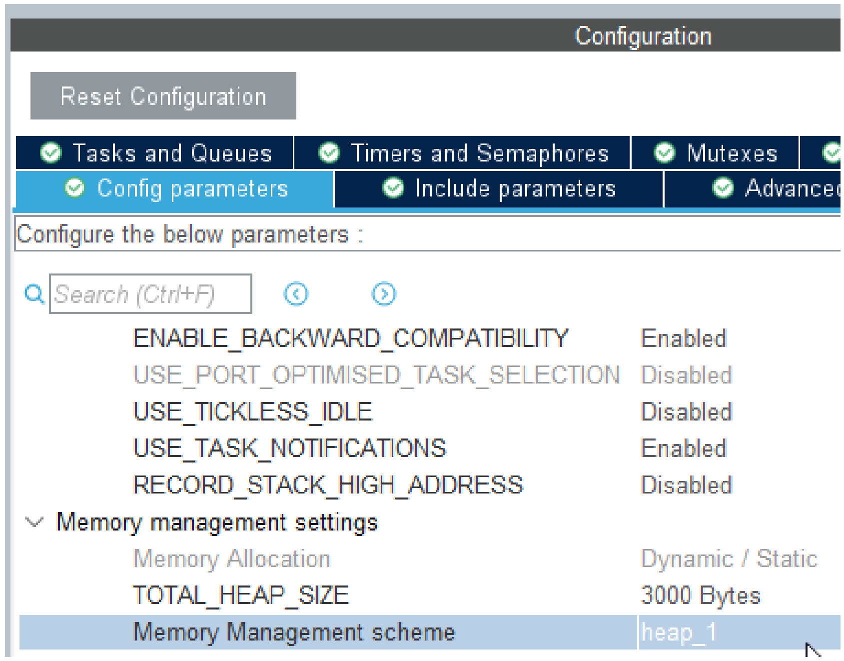
6. When Thread2 has been deleted, Thread1 is again the highest priority thread, so continues executing – at which point it calls *osDelay*() to block for a short period.

7. The idle thread executes while Thread1 is in the blocked state and frees the memory that was allocated to the now deleted Thread2. When Thread1 leaves the blocked state, it again becomes the highest priority Ready state thread and so preempts the idle thread. When it enters the Running state, it creates Thread2 again, and so it goes on.

   The expected execution pattern is illustrated as below.



Questions:
1. How many CMSIS-RTOSv2 API functions are called in the main.c file? Please list them and briefly summarize their usage.
2. Set a breakpoint at the osThreadTerminate() statement. Step into this function and check which FreeRTOS API function(s) are called?
3. After you change the heap file to heap_1.c in the configuration file as below, regenerate the code. Then observe and record the debug display message in the Tera Term.
Please give your answers in the lab report.


Configuration
Reset Configuration
Tasks and Queues
Timers and Semaphores
Mutexes
Config parameters
Include parameters
Advanced
Configure the below parameters :
Search (Ctrl+F)
ENABLE_BACKWARD_COMPATIBILITY Enabled
USE_PORT_OPTIMISED_TASK_SELECTION Disabled
USE_TICKLESS_IDLE Disabled
USE_TASK_NOTIFICATIONS Enabled
RECORD_STACK_HIGH_ADDRESS Disabled
Memory management settings
Memory Allocation Dynamic / Static
TOTAL_HEAP_SIZE 3000 Bytes
Memory Management scheme heap_1