Real-time Embedded Systems Lab Manual

Thread Management in FreeRTOS – Part 3 (Lab 4 - 6)

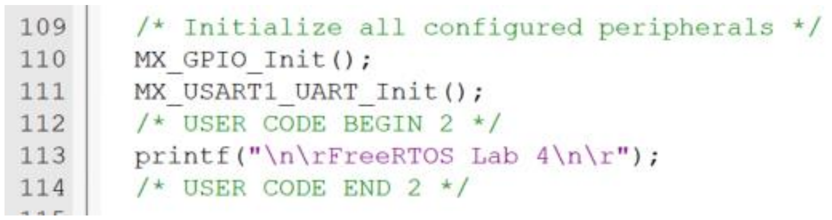
Lab 4 Using the Blocked state to create a delay

All the threads created so far have been ‘periodic’ – they have delayed for a period and print out their string, before delaying once more, and so on. So the thread effectively polled an incrementing loop counter using a null loop until it reached a fixed value. While executing the null loop, the thread remained in the Ready state, starving the other threads with lower priority.

This lab, we will replace the polling null loop with a call to the *osDelay(uint32_t millisec)* library function. It places the calling thread into the Blocked state for a fixed number of tick interrupts. While in the Blocked state, the thread does not use any processing time. Thus, the processing time is consumed only when there is work to be done.

The input parameter *osDelay* is the number of tick interrupts that the calling thread should remain in the blocked state before being transitioned back into the Ready state. For example, if a thread executes *osDelay (100)* while the tick count was **10,000**, then it would immediately enter the blocked state and remain there until the tick count reached **10, 100**.

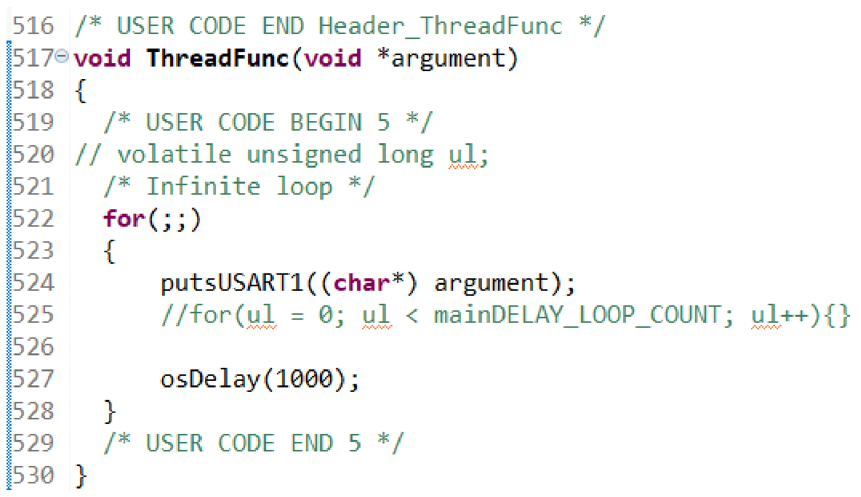
1. Assume you use the lab3 generated lab C project as the starting point. Edit the main.c file as the following:
   a) Slight modify the display message (i.e., from “…Lab 3…” to “…Lab 4…”) in the main() function.



```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART1_UART_Init();
/* USER CODE BEGIN 2 */
printf("\n\rFreeRTOS Lab 4\n\r");
/* USER CODE END 2 */
```

   b) Comment out the null *for* loop for generating delay and the related variable declaration (i.e., *ul*) in the *ThreadFunc()*.

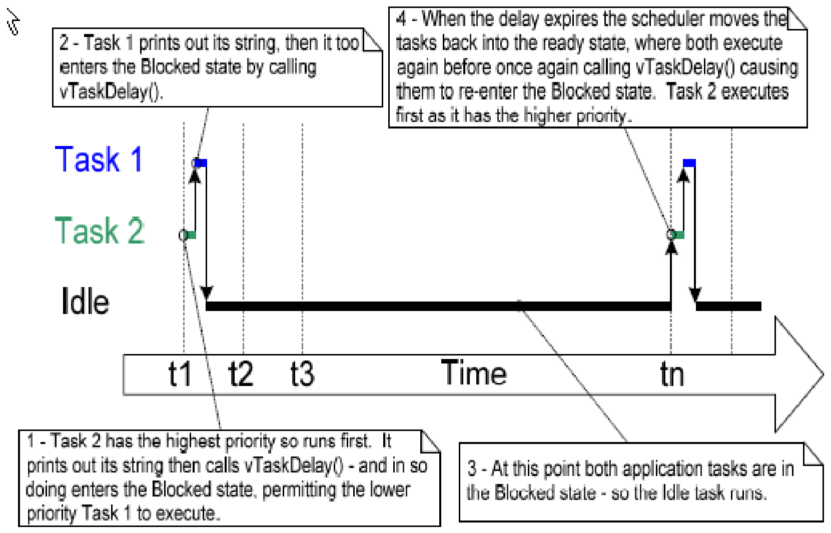
   c) Next, add a statement to call the ***osDelay( )*** library function. The input parameter is given as “*1000*”. (Delay for 1000 ms.)



```
/* USER CODE END Header_ThreadFunc */
void ThreadFunc(void *argument)
{
  /* USER CODE BEGIN 5 */
// volatile unsigned long ul;
  /* Infinite loop */
  for(;;)
  {
      putsUSART1((char*) argument);
      //for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++){}

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

2. Now, build and run the project.
3. Observe the output at Tera-term. You could find that **both threads** run even though they are created at different priorities.

The expected execution pattern illustrates below.



2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().
4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.
Task 1
Task 2
Idle
t1
t2
t3
Time
tn
1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.
3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

Lab 5. Converting the example threads to use *osDelayUntil*()

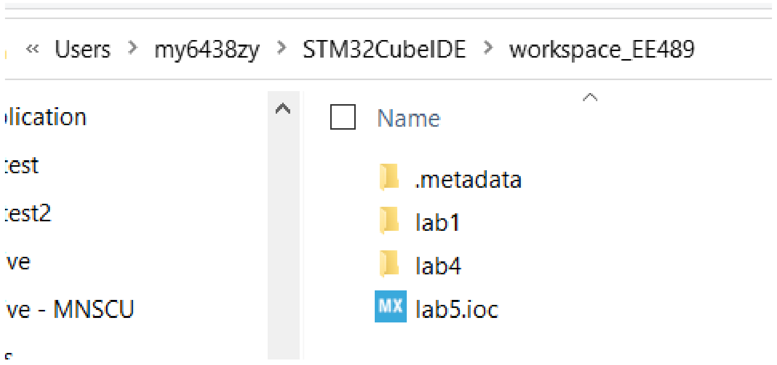
The two threads created in Lab 4 are real periodic threads, but using *osDelay*() does not guarantee that the frequency at which they run is fixed, as the time at which the threads leave the Blocked state is relative to when they call *osDelay*(). We solve this potential problem by using the *osDelayUntil()* library function.

Its prototype is as

osStatus_t osDelayUntil (uint32_t ticks)

- *ticks* is the ***absolute*** time in ticks. (wait until specified time)
- Return type is the status code *osStatus_t* that indicates the execution status of the function.

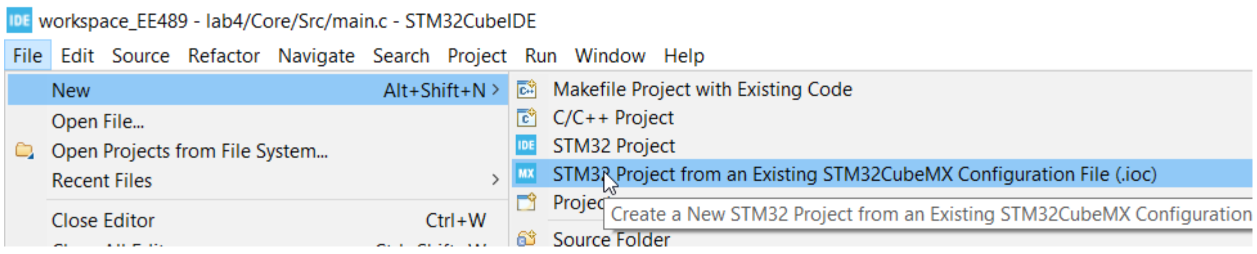
1. Assume you use the MCU configuration file of lab3 as the starting point. First, copy this file to the workspace directory and rename it as *lab5.ioc* as a start point.
   Make sure that the priority of Thread2 is set higher than that of Thread1, such as *osPriorityNormal1* for Thread2, *osPriorityNormal* for Thread1.



« Users › my6438zy › STM32CubeIDE › workspace_EE489

ılication
:est
:est2
ve
ve - MNSCU

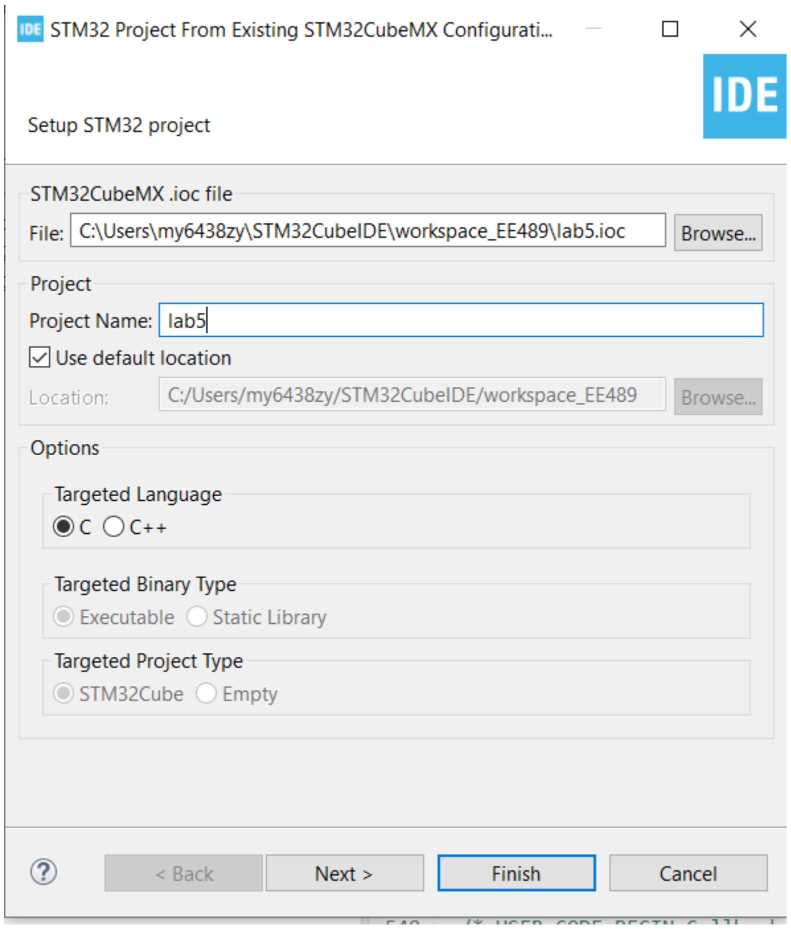
Name
.metadata
lab1
lab4
lab5.ioc



workspace_EE489 - lab4/Core/Src/main.c - STM32CubeIDE

File Edit Source Refactor Navigate Search Project Run Window Help
New Alt+Shift+N
Open File...
Open Projects from File System...
Recent Files
Close Editor Ctrl+W
Makefile Project with Existing Code
C/C++ Project
STM32 Project
STM32 Project from an Existing STM32CubeMX Configuration File (.ioc)
Projec
Source Folder

Create a New STM32 Project from an Existing STM32CubeMX Configuration


STM32 Project From Existing STM32CubeMX Configurati...

Setup STM32 project

STM32CubeMX .ioc file

File: C:\Users\my6438zy\STM32CubeIDE\workspace_EE489\lab5.ioc

Project

Project Name: lab5

[✓] Use default location

Location: C:/Users/my6438zy/STM32CubeIDE/workspace_EE489

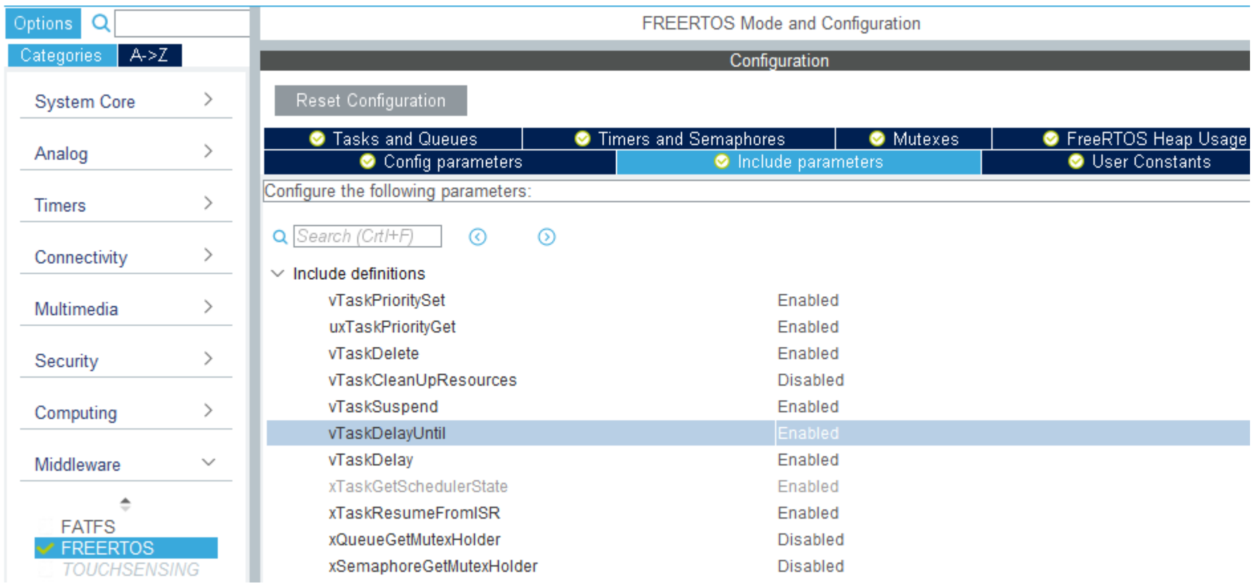
Options

Targeted Language: [● C] ○ C++

Targeted Binary Type: [● Executable] ○ Static Library

Targeted Project Type: [● STM32Cube] ○ Empty

2. Select Middleware>>FREERTOS, on the tab “Include parameters” of the Configuration panel, make sure that the *vTaskDelayUntil* is **enabled**.



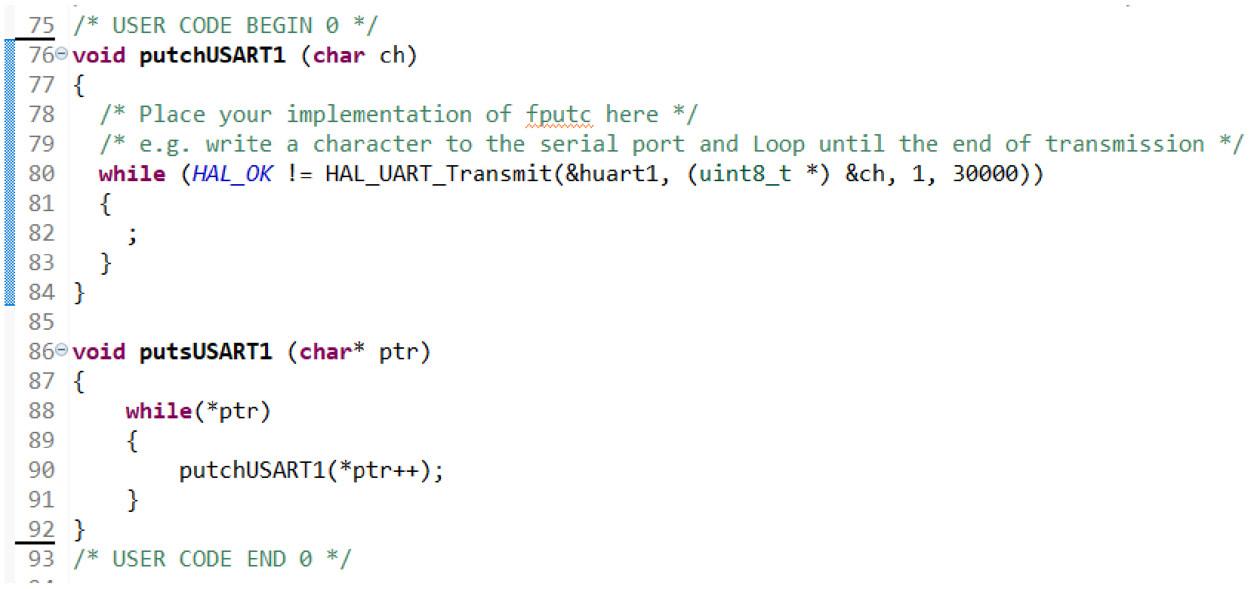
Configure the following parameters:

Include definitions

vTaskPrioritySet	Enabled
uxTaskPriorityGet	Enabled
vTaskDelete	Enabled
vTaskCleanUpResources	Disabled
vTaskSuspend	Enabled
vTaskDelayUntil	Enabled
vTaskDelay	Enabled
xTaskGetSchedulerState	Enabled
xTaskResumeFromISR	Enabled
xQueueGetMutexHolder	Disabled
xSemaphoreGetMutexHolder	Disabled

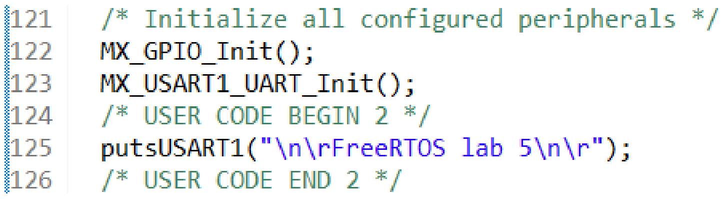
3. Click the *Generate Button* to create the C project.

4. Edit the main.c similar to that in lab1 to define *putchUSART1() and putsUSART1()*.



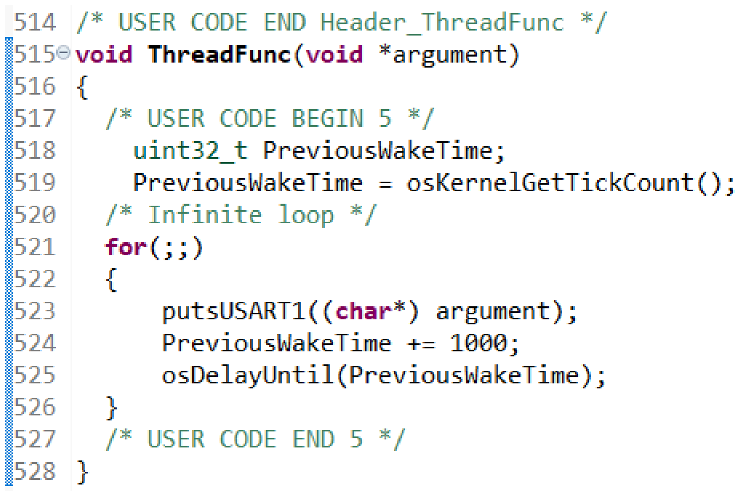
```c
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

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

In the main() function, call the putsUSART1() function.



```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS lab 5\n\r");
  /* USER CODE END 2 */
```

5. Edit the thread function *ThreadFunc()*.
   - Declare a new local variable *PreviousWakeTime* with the type *uint32_t*;
   - Initialize the variable *PreviousWakeTime* with the current tick count. Note that this is the only time we access this variable. This tick count could be obtained by calling the library function *osKernelGetTickCount* ().
   - Inside the infinite *for(;;)* loop, call *osDelayUntil()* library function instead of *osDelay()* to generate delay.
     Compute the absolute waiting time by adding 1000 ticks based on the current tick time.



```
/* USER CODE END Header_ThreadFunc */
void ThreadFunc(void *argument)
{
  /* USER CODE BEGIN 5 */
    uint32_t PreviousWakeTime;
    PreviousWakeTime = osKernelGetTickCount();
  /* Infinite loop */
  for(;;)
  {
      putsUSART1((char*) argument);
      PreviousWakeTime += 1000;
      osDelayUntil(PreviousWakeTime);
  }
  /* USER CODE END 5 */
}
```

6. Build and debug Lab5.

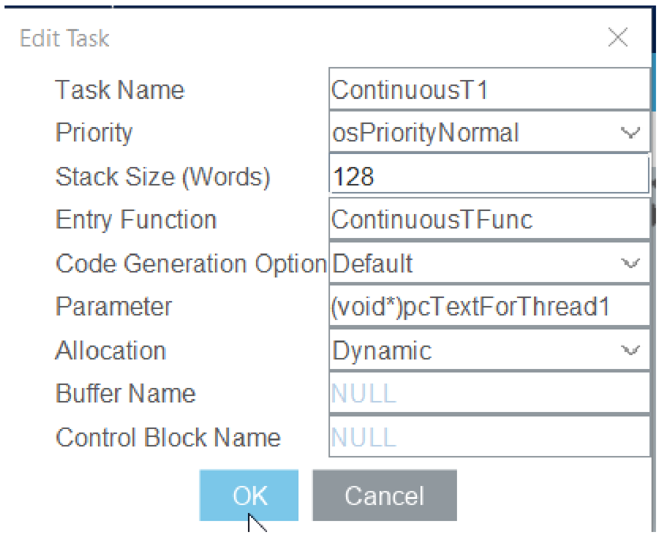
The output produced should be same as the one from Lab4.

Lab 6. Combining blocking and non-blocking (continuous) threads

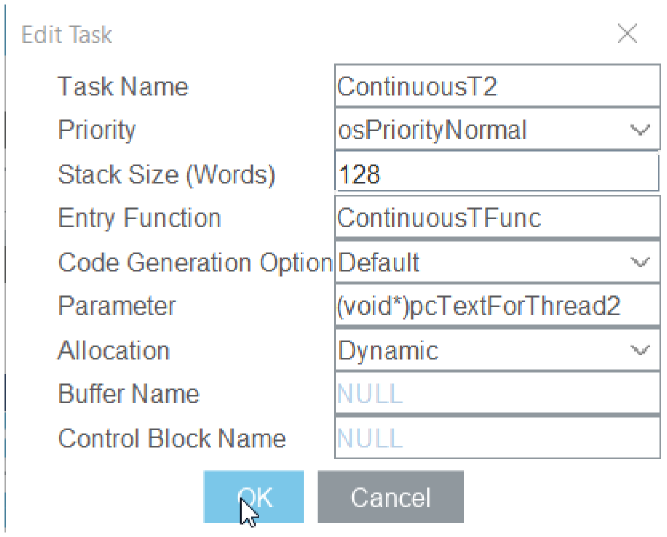
Previous labs have examined the behavior of both polling and blocking threads in isolation. This lab aims at demonstrating an execution sequence when two schemes are combined as follows.

1. Two *Continuous* threads are created in priority *osPriorityNormal*. These do nothing other than continuously print out a string.
   These threads never make any library function calls that could cause them to enter the Blocked state, so are always in either the Ready or the Running state. Threads of this nature are called *continuous processing* threads as they always have work to do.
2. A third thread is created at priority *osPriorityAboveNormal*. It just prints out a string periodically, so uses the *osDelayUntil()* library function to place itself into the Blocked state between each print iteration.

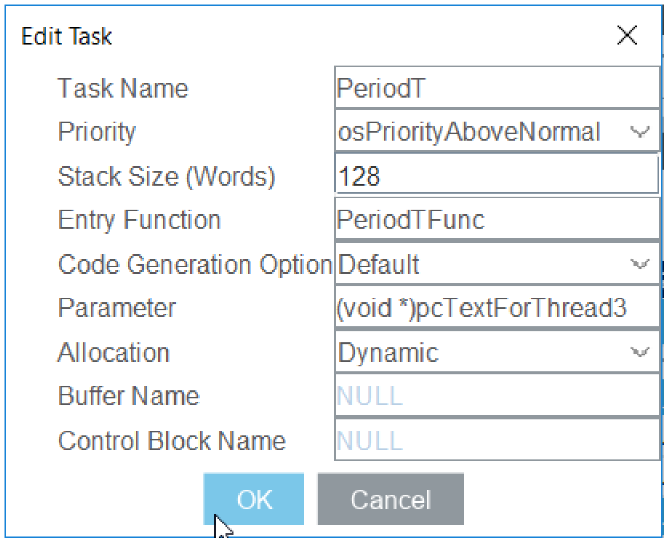
1. Assume you reuse and copy the MCU configuration file of lab5 in the same way as in lab6 as a start point.
2. Select the tab “Tasks and Queues” of the Configuration panel, and remove/edit the two tasks of lab 5 and design three new threads as shown below.



Task Name	ContinuousT1
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	ContinuousTFunc
Code Generation Option	Default
Parameter	(void*)pcTextForThread1
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

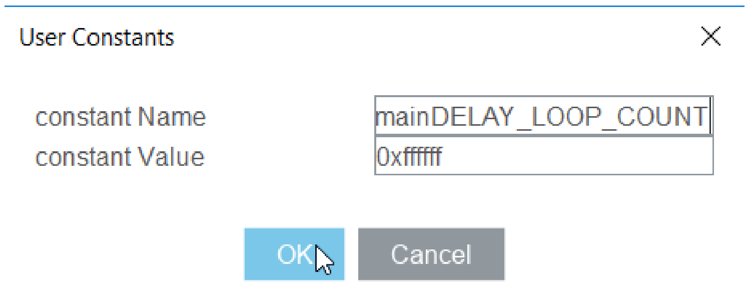

Edit Task

Task Name	ContinuousT2
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	ContinuousTFunc
Code Generation Option	Default
Parameter	(void*)pcTextForThread2
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

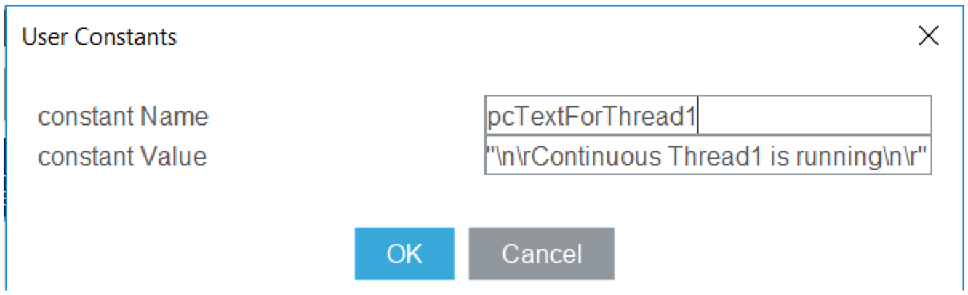

Edit Task

Task Name	PeriodT
Priority	osPriorityAboveNormal
Stack Size (Words)	128
Entry Function	PeriodTFunc
Code Generation Option	Default
Parameter	(void *)pcTextForThread3
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

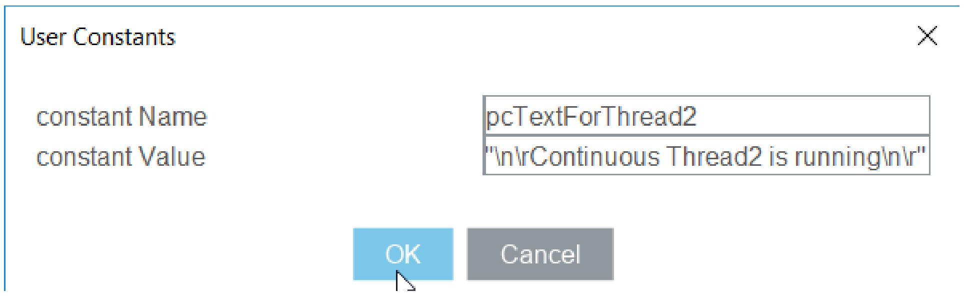
3. Edit the "User Constants". Enter four constants as the following.
   Constant name: *pcTextForThread1*; Constant value: "\n\rContinuous Thread 1 is running\n\r"
   Constant name: *pcTextForThread2*; Constant value: "\n\rContinuous Thread 2 is running\n\r"
   Constant name: *pcTextForThread3*; Constant value: "\n\rPeriodic Thread is running\n\r"
   Constant name: mainDELAY_LOOP_COUNT; Constant value: 0xffffff


User Constants

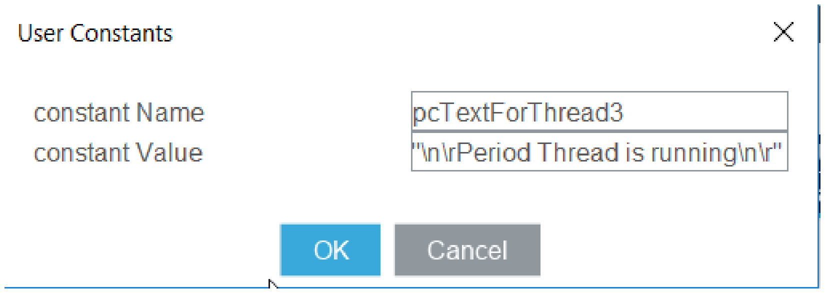
constant Name	mainDELAY_LOOP_COUNT
constant Value	0xffffff


User Constants

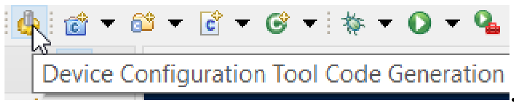
constant Name	pcTextForThread1
constant Value	"\n\rContinuous Thread1 is running\n\r"


User Constants

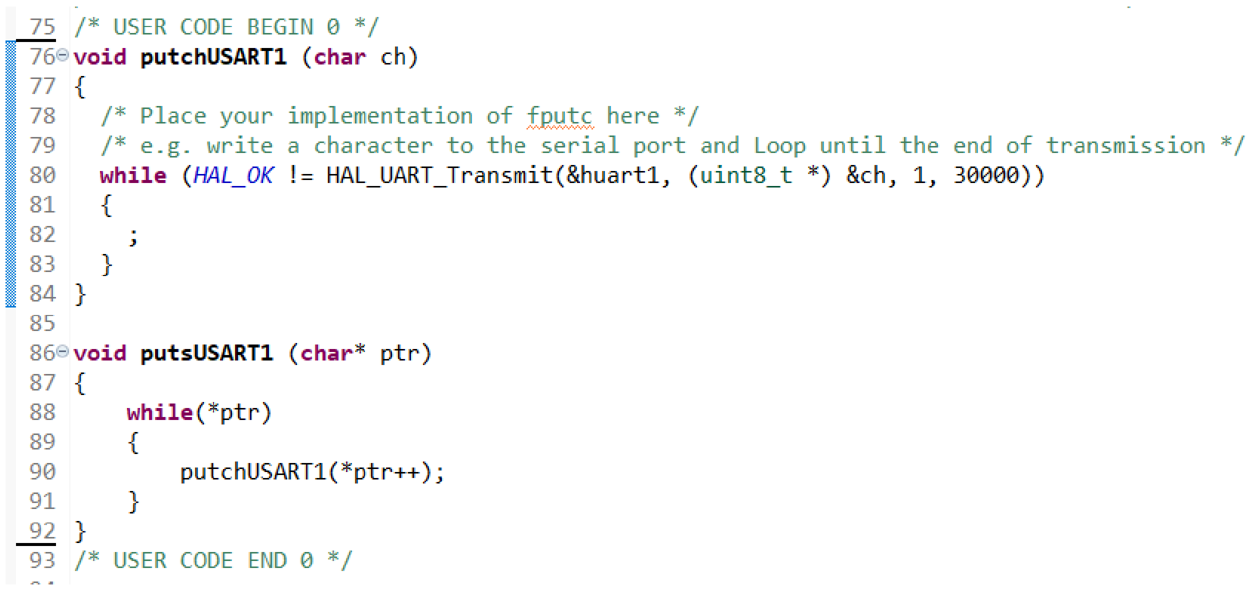
constant Name	pcTextForThread2
constant Value	"\n\rContinuous Thread2 is running\n\r"


User Constants

constant Name	pcTextForThread3
constant Value	"\n\rPeriod Thread is running\n\r"

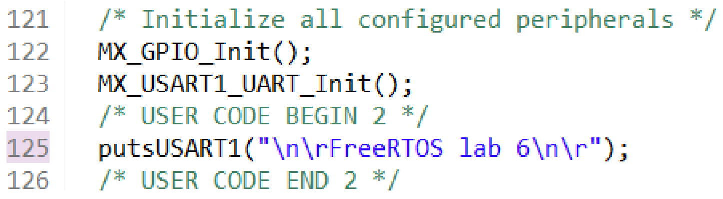
4. Click the *Generate Code* button .

5. Edit the main.c similar to that in lab1 to define *putchUSART1() and putsUSART()*.



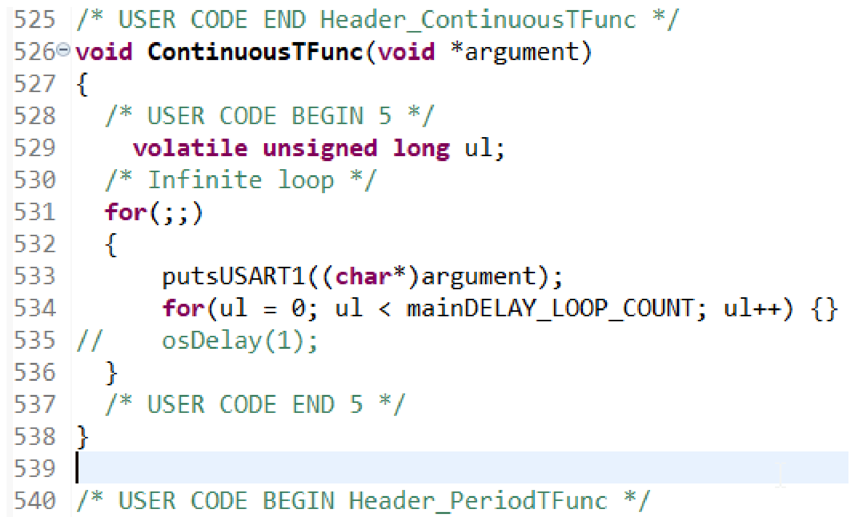
```
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

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

In the main() function, call putsUSART1() function



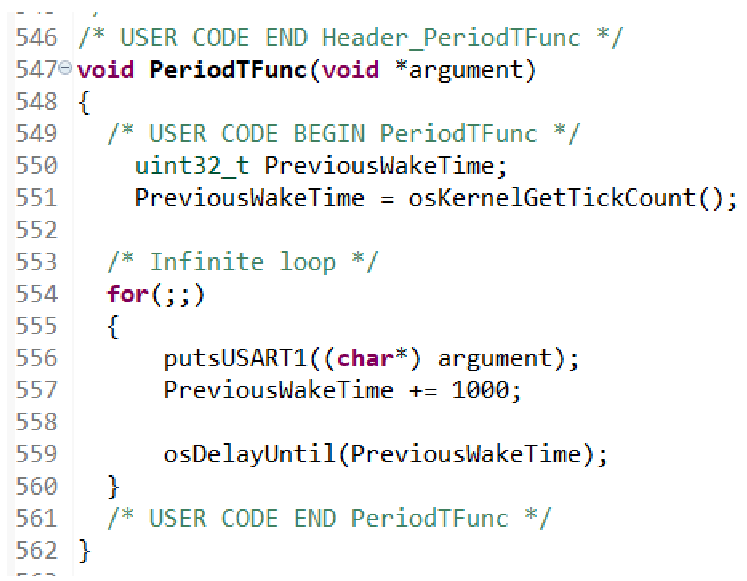
```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS lab 6\n\r");
  /* USER CODE END 2 */
```

6. Define the continuous processing thread function *ContinuousTFunc()*. It includes a null loop delay inside the infinite *for(; ;)* loop as the Thread1 (or Thread2) in Lab 1.



```c
/* USER CODE END Header_ContinuousTFunc */
void ContinuousTFunc(void *argument)
{
  /* USER CODE BEGIN 5 */
    volatile unsigned long ul;
  /* Infinite loop */
  for(;;)
  {
      putsUSART1((char*)argument);
      for(ul = 0; ul < mainDELAY_LOOP_COUNT; ul++) {}
//    osDelay(1);
  }
  /* USER CODE END 5 */
}

/* USER CODE BEGIN Header_PeriodTFunc */
```

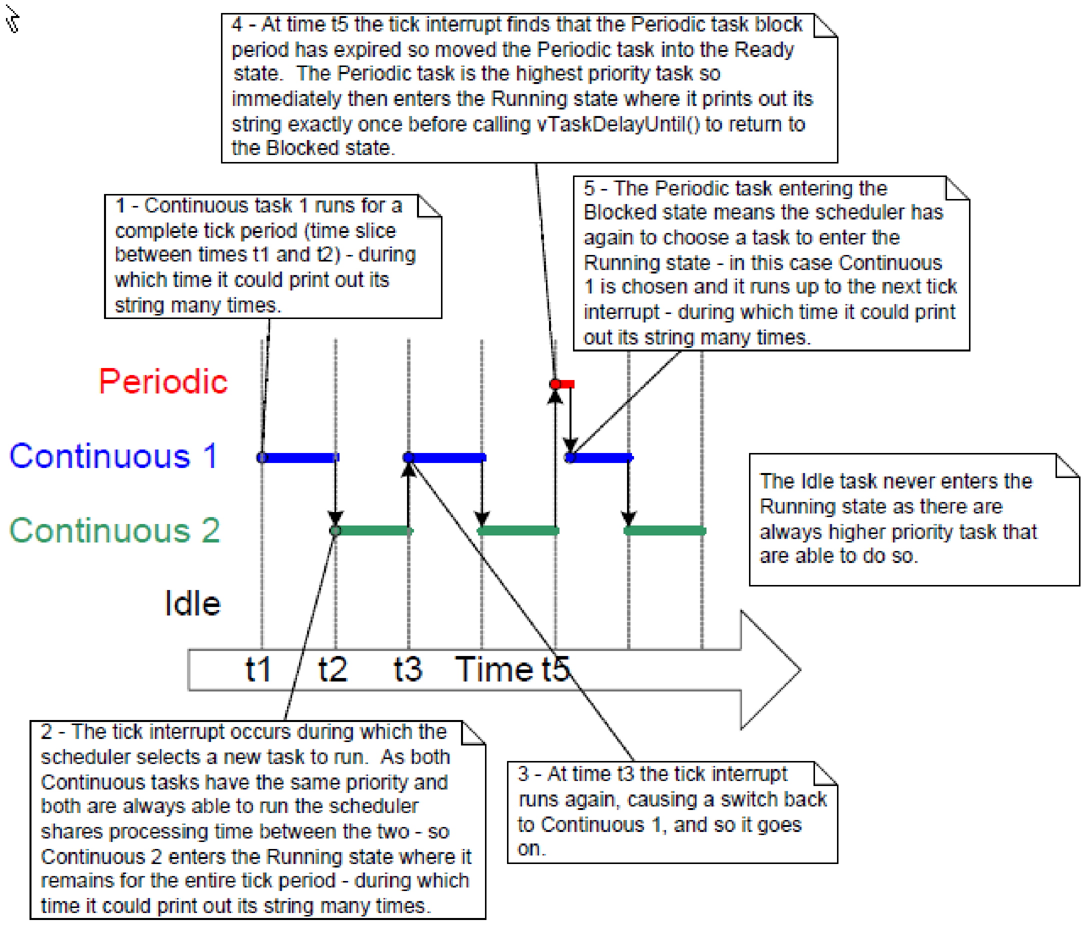
7. Edit another thread function for the third periodic thread *PeriodTFunc ()*. Within its infinite *for(; ;)* loop, call the *osDelayUntil()* library function to create the delay until the specified time as in the *ThreadFunc()* function in lab5.



```
/* USER CODE END Header_PeriodTFunc */
void PeriodTFunc(void *argument)
{
  /* USER CODE BEGIN PeriodTFunc */
    uint32_t PreviousWakeTime;
    PreviousWakeTime = osKernelGetTickCount();

  /* Infinite loop */
  for(;;)
  {
      putsUSART1((char*) argument);
      PreviousWakeTime += 1000;

      osDelayUntil(PreviousWakeTime);
  }
  /* USER CODE END PeriodTFunc */
}
```

8. Build and debug the project, observe the debugging result.

The execution pattern is expected to be as shown before.


4 - At time t5 the tick interrupt finds that the Periodic task block period has expired so moved the Periodic task into the Ready state. The Periodic task is the highest priority task so immediately then enters the Running state where it prints out its string exactly once before calling vTaskDelayUntil() to return to the Blocked state.
1 - Continuous task 1 runs for a complete tick period (time slice between times t1 and t2) - during which time it could print out its string many times.
5 - The Periodic task entering the Blocked state means the scheduler has again to choose a task to enter the Running state - in this case Continuous 1 is chosen and it runs up to the next tick interrupt - during which time it could print out its string many times.
Periodic
Continuous 1
Continuous 2
Idle
The Idle task never enters the Running state as there are always higher priority task that are able to do so.
t1
t2
t3
Time t5
2 - The tick interrupt occurs during which the scheduler selects a new task to run. As both Continuous tasks have the same priority and both are always able to run the scheduler shares processing time between the two - so Continuous 2 enters the Running state where it remains for the entire tick period - during which time it could print out its string many times.
3 - At time t3 the tick interrupt runs again, causing a switch back to Continuous 1, and so it goes on.

Questions:

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

Please give your answers in the lab report.