Real-time Embedded Systems Lab Manual

Interrupt Management in FreeRTOS (Lab 12 - 14)

Lab 12 Using a binary semaphore to synchronize a thread with an interrupt

This example uses a binary semaphore to unlock a thread from within an interrupt service routine – effectively synchronizing the thread with the interrupt.

A simple periodic thread *PeriodThread(void *pvParameters)* is used to periodically request user to press the User button (blue one) on the board.

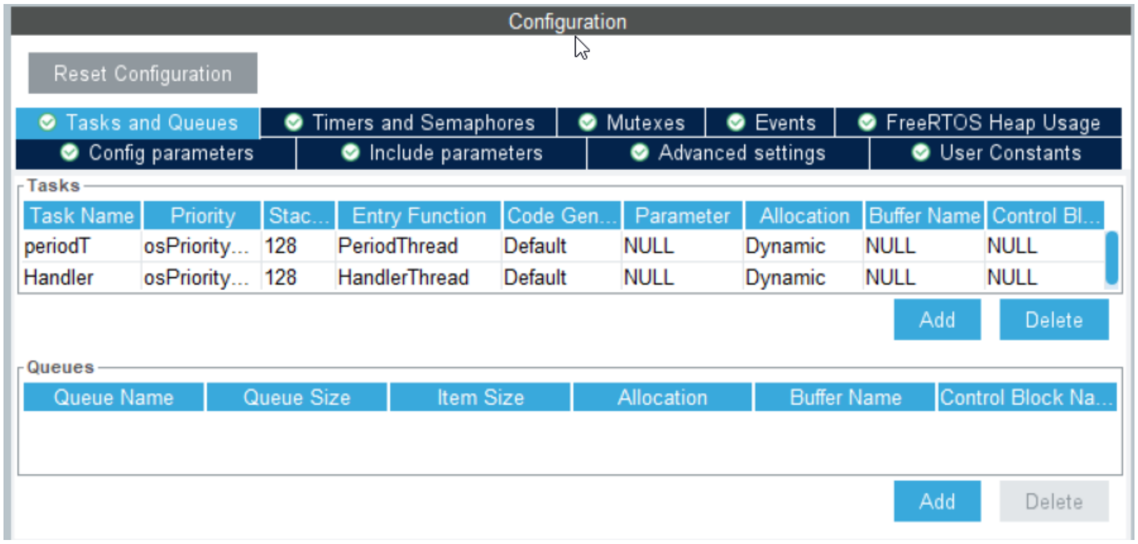
The implementation of the handler thread *HandlerThread(void *pvParameters)* – the thread that is synchronized with the software interrupt through the use of a binary semaphore, is also given. A message is printed out from each iteration of the thread, so the sequence in which the thread and the interrupt execute is evident from the output produced when the example is executed.

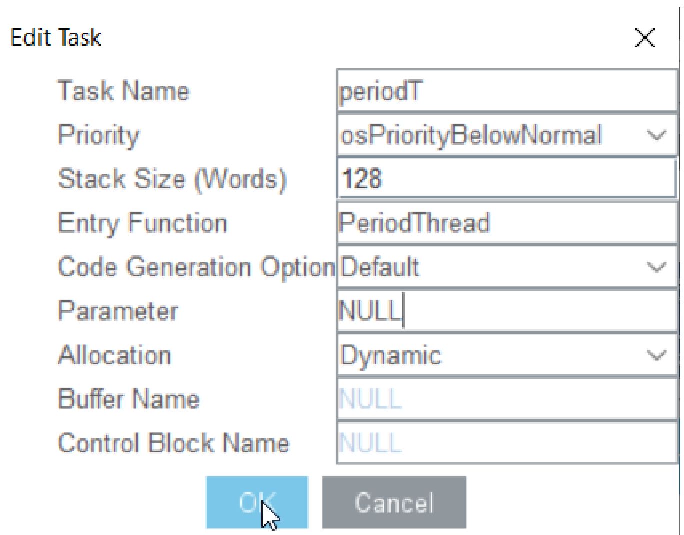
The interrupt service routine *HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)* is simply a standard C function. It does very little other than 'give' the semaphore to unblock the handler task.

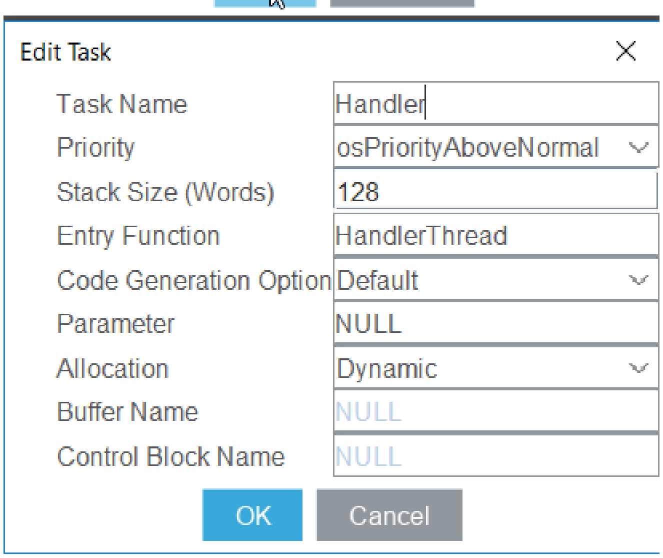
1. Assume that we use the MCU configuration file of lab10 as a start point and copy it to a new directory for lab12. You could also use the configuration file of any previous lab. Just make sure the following modes are set:
   a) The debug **Mode** of the **SYS** module under **System Core** category is set **Serial Wire**; the Timebase Source is set as TIM6 or TIM7.
   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*

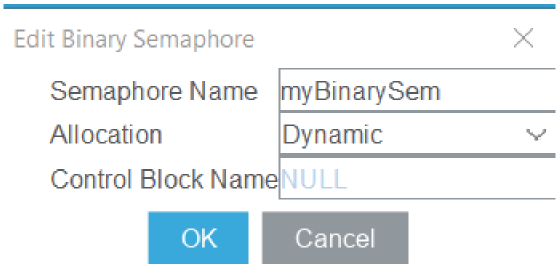
2. Select Middleware>>FREERTOS, open the "Tasks and Queues" tab on the FREERTOS Configuration panel. Edit one periodic thread and one Handler thread. (Delete the **sender2** and **receiver** tasks of lab10 and edit the **sender1** task to be **periodT**. Also delete the Queue added in lab 10.)

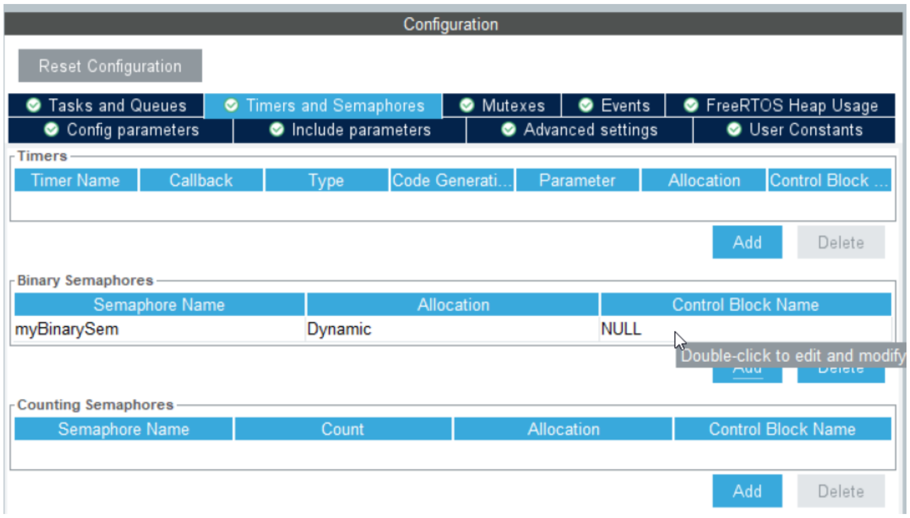




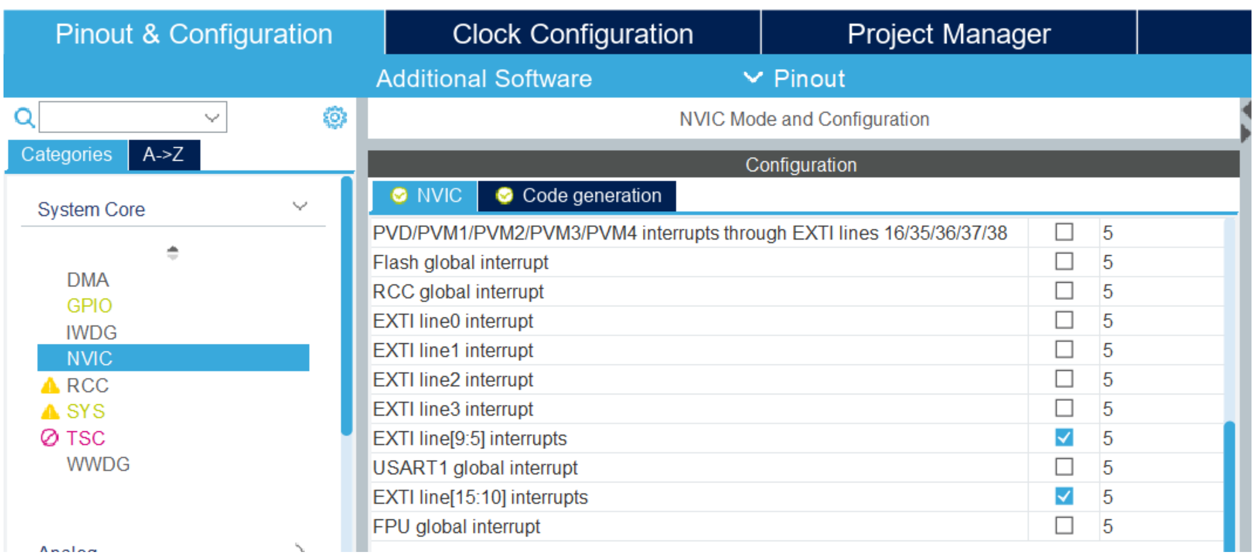


3. Open the “Timers and Semaphores” tab. Define a binary semaphore.





4. Select System Core >> NVIC, make sure that *EXTI line[9:5] interrupts* and *EXTI line[15:10] interrupts* are enabled. (They are enabled by default.)



5. Save the configure file and generate code.

6. Edit the main.c file.

   a) 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 */
```

   a) 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 12\n\r");
  /* USER CODE END 2 */
```

   b) Within `/* USER CODE BEGIN 4 */ and /* USER CODE END 4 */,` near the end of main.c, Implement the HAL_GPIO_EXTI_Callback function to release the binary semaphore.

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_13)
    {
        putsUSART1("\n\rUser Button pressed. Interrupt: - to give a Semaphore.\n\r ");
        osSemaphoreRelease(myBinarySemHandle);
    }
}
/* USER CODE END 4 */
```

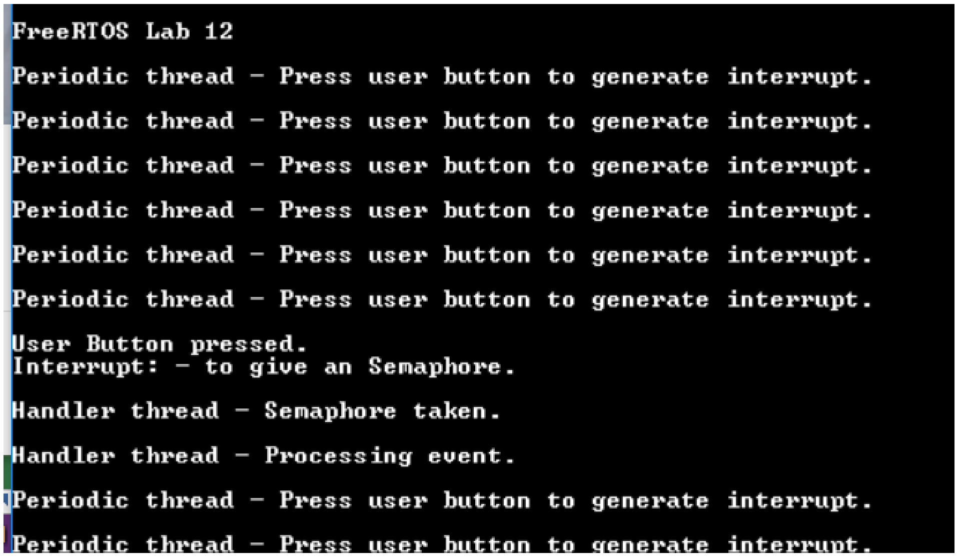
   c) Implement the Handler Thread that waits for the semaphore released by the Interrupt Callback function and processes the remaining of interrupt service function if any.

```c
/* USER CODE END Header_HandlerThread */
void HandlerThread(void *argument)
{
  /* USER CODE BEGIN HandlerThread */
    osSemaphoreAcquire(myBinarySemHandle, 0);
  /* Infinite loop */
  for(;;)
  {
      osSemaphoreAcquire(myBinarySemHandle, osWaitForever);

      putsUSART1("\n\rHandler thread - Semaphore taken. \r\n");

      putsUSART1("\n\rHandler thread - Processing event. \r\n");
    //osDelay(1);
  }
  /* USER CODE END HandlerThread */
}
```

d) Define the PeriodThread function that reminds user to press user button to generate the interrupt.

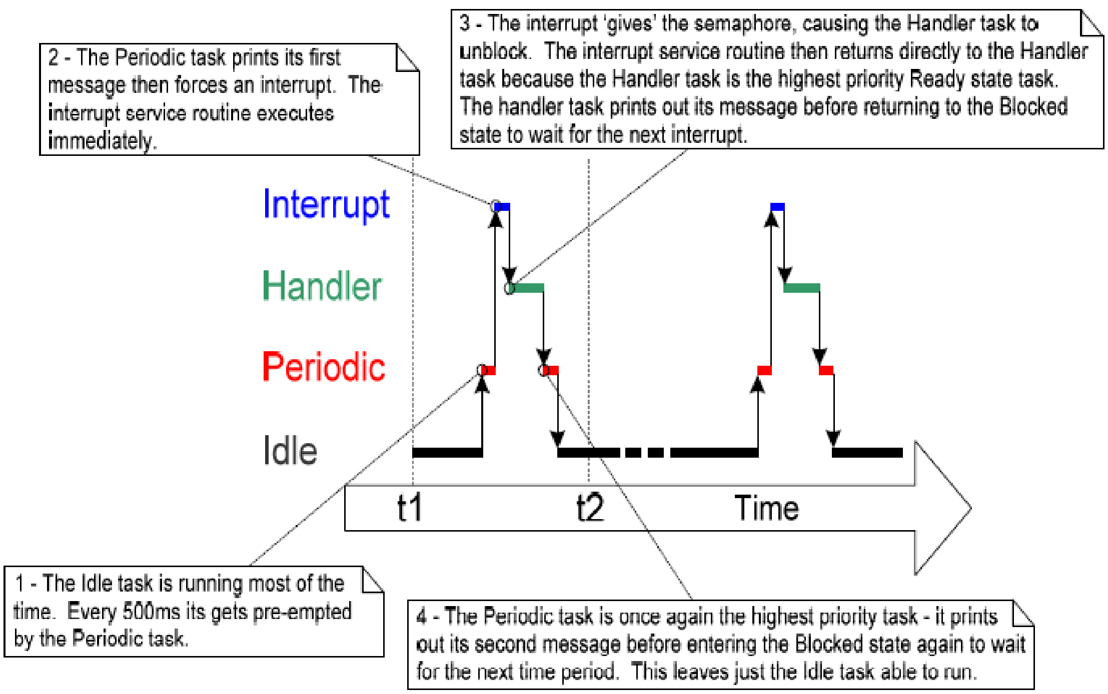
```c
/* USER CODE END Header_PeriodThread */
void PeriodThread(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1000);
    putsUSART1("\n\rPeriod thread - Press user button to generate interrupt.\n\r");
  }
  /* USER CODE END 5 */
}
```

Lab 12 produces the output shown in the following figure. As expected, the handler task executes as soon as the interrupt is generated, so the output from the handler task splits the output produced by the periodic task.



```
FreeRTOS Lab 12
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
User Button pressed.
Interrupt: - to give an Semaphore.
Handler thread - Semaphore taken.
Handler thread - Processing event.
Periodic thread - Press user button to generate interrupt.
Periodic thread - Press user button to generate interrupt.
```

The execution sequence follows the pattern below.



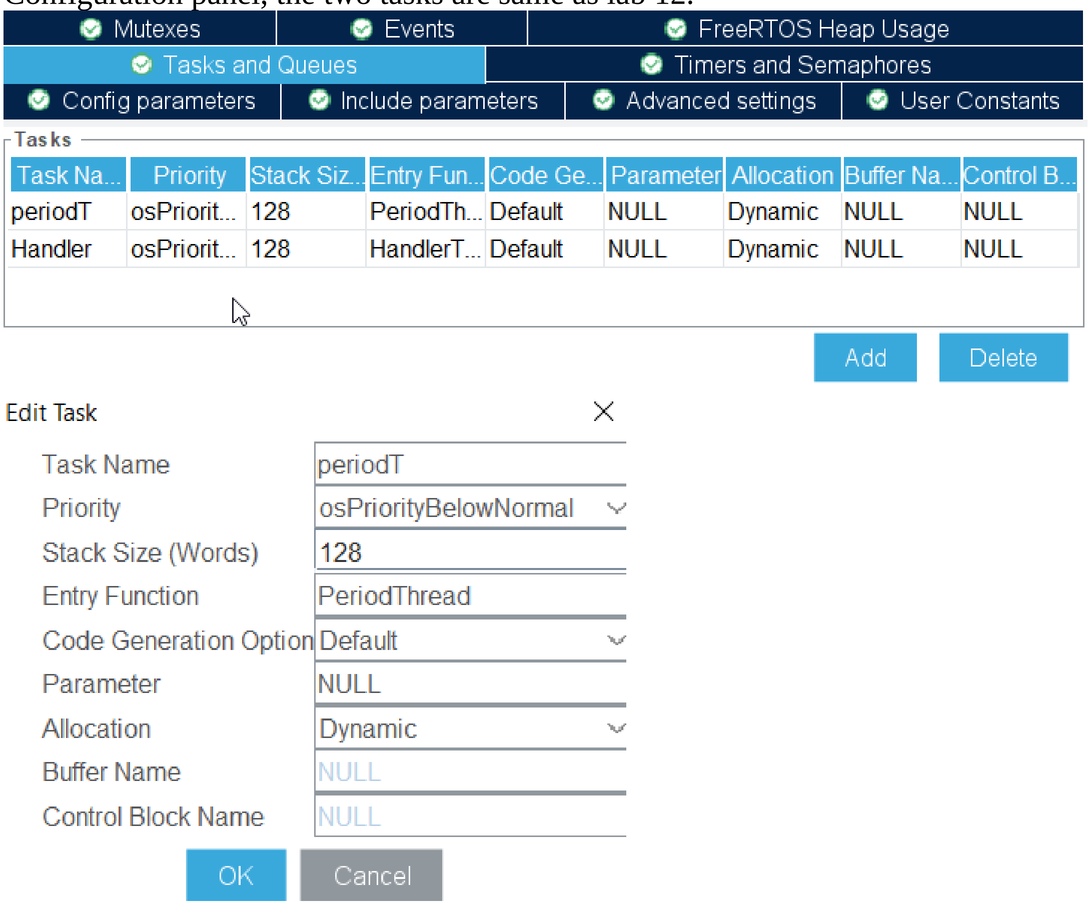
Counting Semaphore

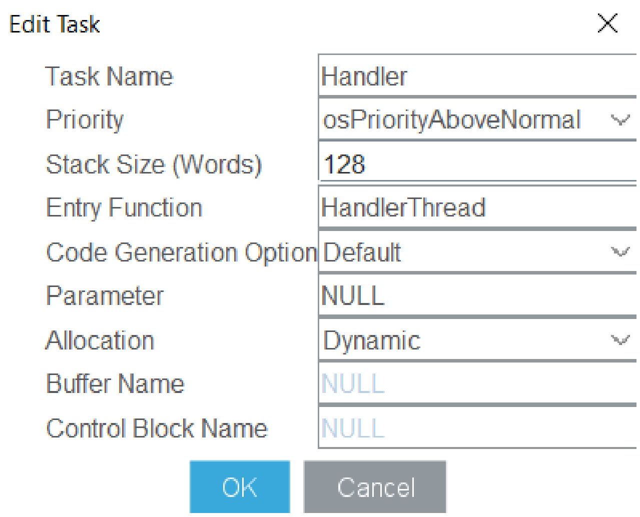
Lab 13. Using a counting semaphore to synchronize a task with an interrupt

This example improves the example 12 implementation by using a counting semaphore in place of the binary semaphore.

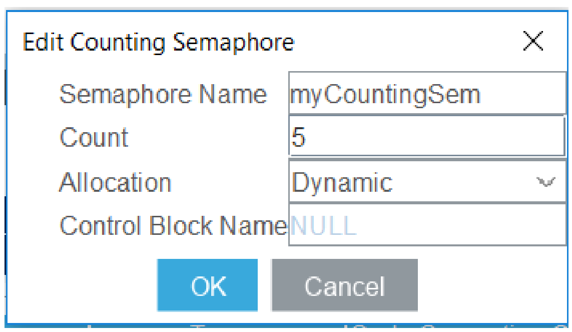
Create a new folder for lab 13, and configure peripherals as lab12, (You could copy *lab12.ioc* and rename it as *lab13.ioc* as the starting point.)

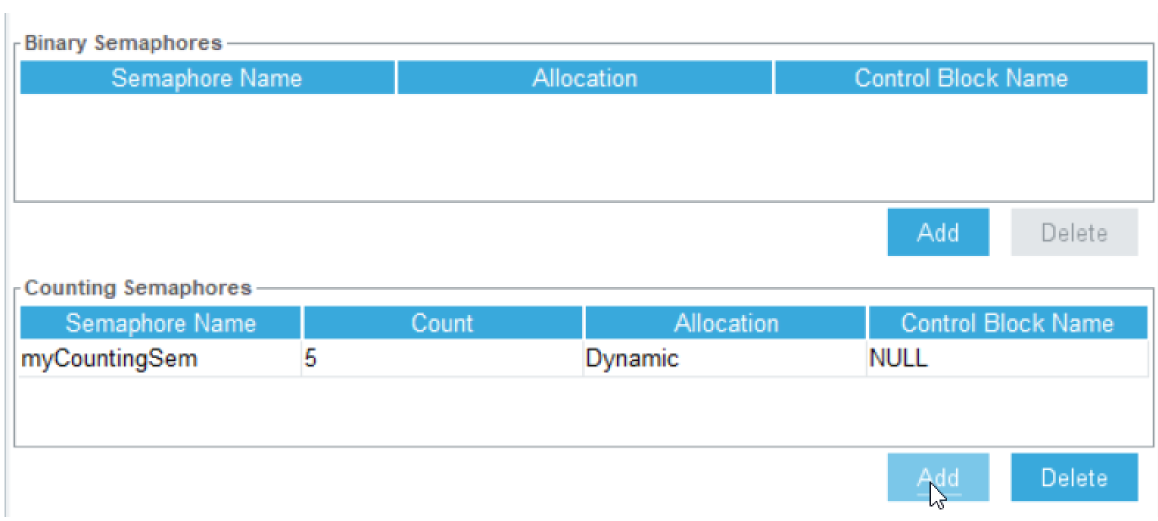
1. Select Middleware>>FREERTOS, open the "Tasks and Queues" tab on the Freertos Configuration panel, the two tasks are same as lab 12.



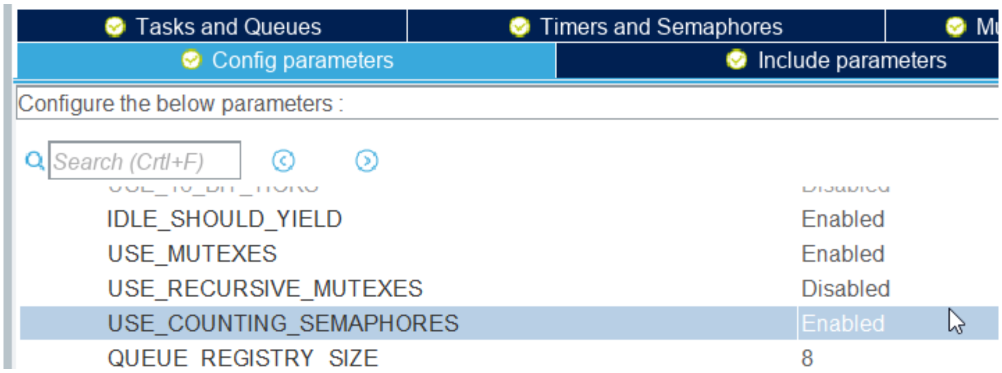


2. Open the “Timers and Semaphores” tab. Delete the binary semaphore created in lab12, and define a counting semaphore instead.





3. Open the “Config parameters” tab. Enable the USE_COUNTING_SEMAPHORES if it has not been enabled.



4. Save the configuration file to generate code.
5. Edit the main.c file.

   a) Define putchUSART1() and putsUSART1() to use the USART1 channel as in lab12.

   b) In the main() function, call putsUSART1() to display message, and keep the generated code unchanged.

```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS Lab 13\n\r");
  /* USER CODE END 2 */
```

   c) Implement the HAL_GPIO_EXTI_Callback function to release the semaphore three times.

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == GPIO_PIN_13)
    {
        putsUSART1("\n\rUser Button pressed. Interrupt: to give three counting semaphores.\n\r ");
        osSemaphoreRelease(myCountingSemHandle);
        osSemaphoreRelease(myCountingSemHandle);
        osSemaphoreRelease(myCountingSemHandle);
    }
}
/* USER CODE END 4 */
```

Implement the Handler Thread that waits for the semaphore released by the Interrupt Callback function and processes the remaining of interrupt service function if any.

```
/* USER CODE END Header_HandlerThread */
void HandlerThread(void *argument)
{
  /* USER CODE BEGIN HandlerThread */
    /*Take the semaphore 5 times to start with so the semaphores is empty before the
     * infinite loop is entered. The semaphore was created before the scheduler was started,
     * also before this task ran for the first time. */
    for(uint8_t i = 0; i < 5; i++)
    {
        osSemaphoreAcquire(myCountingSemHandle, 0);
    }
  /* Infinite loop */
  for(;;)
  {
      osSemaphoreAcquire(myCountingSemHandle, osWaitForever);

      putsUSART1("\n\rHandler thread - one counting semaphore taken. \r\n");

      putsUSART1("\n\rHandler thread - Processing event. \r\n");
   // osDelay(1);
  }
  /* USER CODE END HandlerThread */
}
```

Define the PeriodThread function that reminds user to press user button to generate the interrupt in the same way as in lab12.

```
/* USER CODE END Header_PeriodThread */
void PeriodThread(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
        osDelay(1000);
        putsUSART1("\n\rPeriod thread - Press user button to generate interrupt.\n\r");
  }
  /* USER CODE END 5 */
}
```

To simulate multiple events occurring at high frequency, the interrupt service routine is changed to ‘release’ the semaphore more than once per interrupt (3 times in this lab). Each event is latched in the semaphore’s count value.

The output produced is shown below. As can be seen, the Handler task processes all three events each time an interrupt is generated. The events are latched into the count value of the semaphore, allowing the Handler task to process them in turn.

```
FreeRTOS lab 13
Period thread - Press user button to generate interrupt.
Period thread - Press user button to generate interrupt.
User Button pressed. Interrupt: to give three counting semaphores.
Handler thread - one counting semaphore taken.
Handler thread - Processing event.
Handler thread - one counting semaphore taken.
Handler thread - Processing event.
Handler thread - one counting semaphore taken.
Handler thread - Processing event.
Period thread - Press user button to generate interrupt.
Period thread - Press user button to generate interrupt.
Period thread - Press user button to generate interrupt.
```

Using Queues within an Interrupt Service Routine.

Lab 14. Sending and receiving on a queue from within an interrupt

This example demonstrate *osMessageGet()* and osMessagePut() being used within the same interrupt.

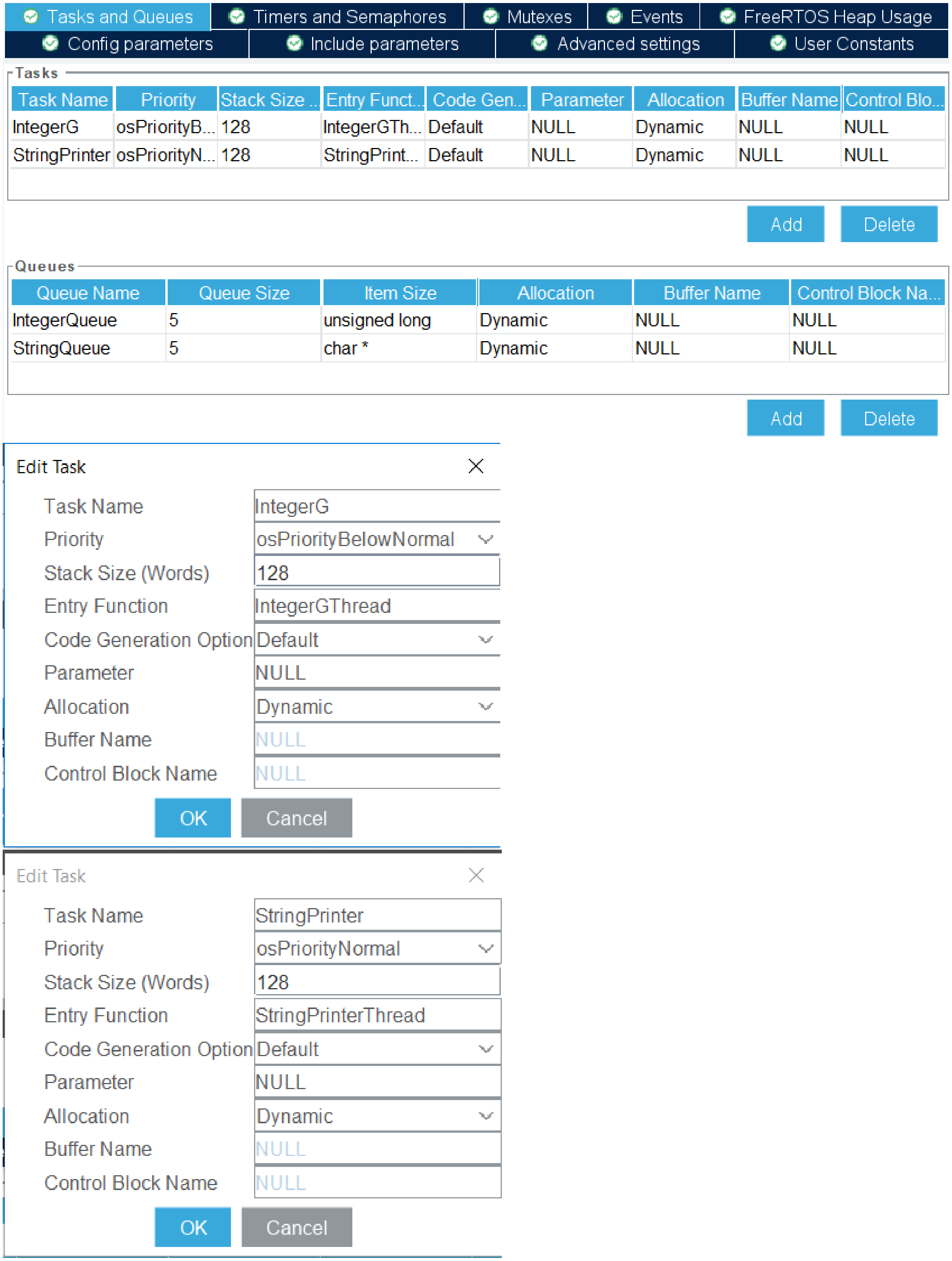
A periodic task *IntegerGThread(void *pvParamters)* is created that sends five numbers to a queue every 1000 milliseconds. It requests user to generate interrupt by pressing the user button only after sending all five values.

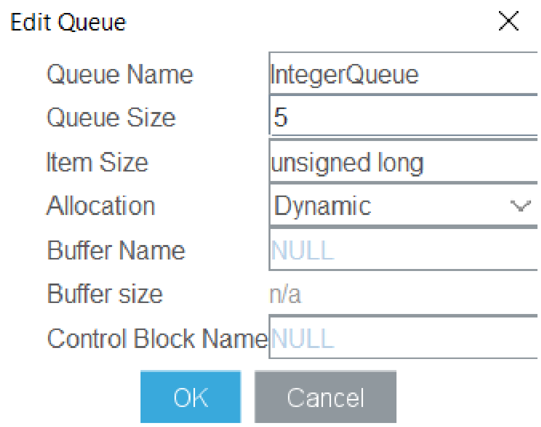
The interrupt service routine *HAL_GPIO_EXTI_Callback(void)* calls *osMessageGet*() repeatedly, until all values written to the queue by the periodic thread have been removed, and the queue is left empty. The last two bits of each received valued are used as an index into an array of strings, with a pointer to the string at the corresponding index position being sent to a different queue using a call to osMessagePut*()*.

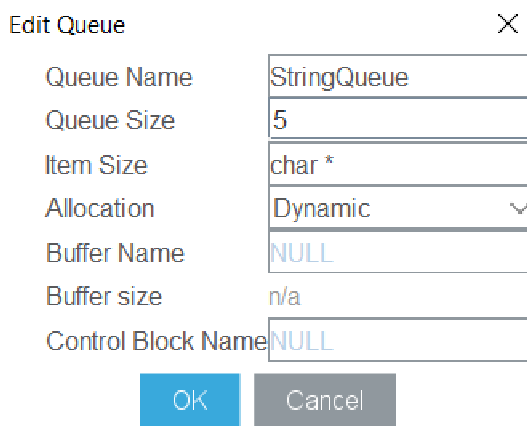
The thread *StringPrinter(void *pvParameters)* that receives the character pointers from the interrupt service routine blocks on the queue until a message arrives, printing out each string as it is received.

Create a new folder for lab 14, and copy *lab12.ioc* and rename it as *lab14.ioc* in the newly created folder as the starting point.

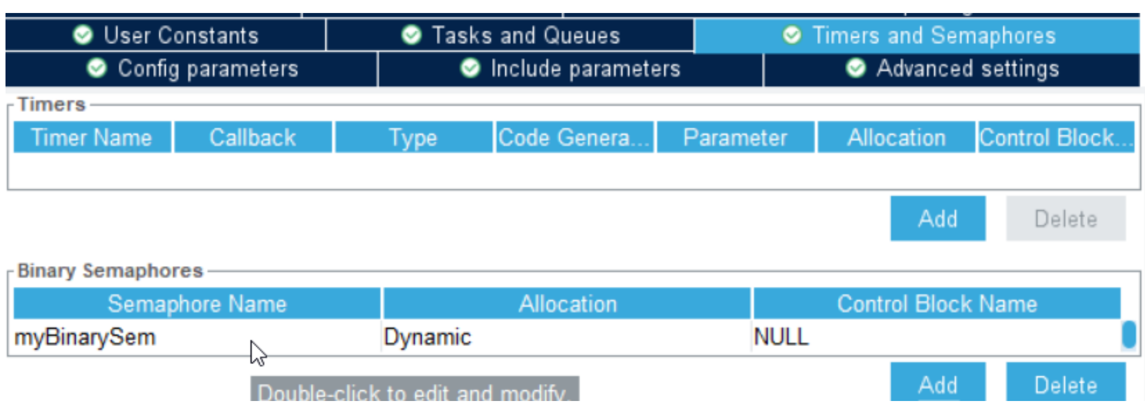
1. Select Middleware>>FREERTOS, open the “Tasks and Queues” tab on the FREERTOS Configuration panel. Create two threads: IntegerG and StringPrinter; and two Queues.
   a) ***IntegerG*** is synchronized with the interrupt call back function via a binary semaphore.
   b) **StringPrinter** has higher priority than IntegerG and displays the string data received from the **String Queue**.
   c) IntegerQueue is a Queue with the size 5 to store the *unsigned long* type data.
   d) StringQueue is a Queue with the size 5 to store the char* type which points to the character string.


Tasks and Queues
Timers and Semaphores
Mutexes
Events
FreeRTOS Heap Usage
Config parameters
Include parameters
Advanced settings
User Constants
Tasks
Task Name | Priority | Stack Size ... | Entry Funct... | Code Gen... | Parameter | Allocation | Buffer Name | Control Blo...
IntegerG | osPriorityB... | 128 | IntegerGTh... | Default | NULL | Dynamic | NULL | NULL
StringPrinter | osPriorityN... | 128 | StringPrint... | Default | NULL | Dynamic | NULL | NULL
Add
Delete
Queues
Queue Name | Queue Size | Item Size | Allocation | Buffer Name | Control Block Na...
IntegerQueue | 5 | unsigned long | Dynamic | NULL | NULL
StringQueue | 5 | char * | Dynamic | NULL | NULL
Add
Delete
Edit Task
Task Name IntegerG
Priority osPriorityBelowNormal
Stack Size (Words) 128
Entry Function IntegerGThread
Code Generation Option Default
Parameter NULL
Allocation Dynamic
Buffer Name NULL
Control Block Name NULL
OK
Cancel
Edit Task
Task Name StringPrinter
Priority osPriorityNormal
Stack Size (Words) 128
Entry Function StringPrinterThread
Code Generation Option Default
Parameter NULL
Allocation Dynamic
Buffer Name NULL
Control Block Name NULL
OK
Cancel





2. Open the “Timers and Semaphores” tab. Keep the binary semaphore created in lab12.



3. Save the configuration file to generate code.
4. Edit 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 as in lab12.

```
/* 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) Within /* USER CODE BEGIN PV */ and /* USER CODE END PV */, declare a global variable named as *pcStrings* as an array of 4 strings and initialize it as below. Also create a char type array variable *buffer*.

```
/* USER CODE BEGIN PV */
static const char *pcStrings[] =
{
        "String 0",
        "String 1",
        "String 2",
        "String 3"
};
char buffer[200];
/* USER CODE END PV */
```

d) 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 14\n\r");
  /* USER CODE END 2 */
```

e) Implement the HAL_GPIO_EXTI_Callback function to release the binary semaphore
- After checking the interrupt does come from user button (Pin 13 line), release the binary semaphore.
- Then, repeatedly receive all integers from the *Integer Queue*, apply the bitwise operation to reduce the received integer to a value between 0 and 3, then use it as

index to access the global array of constant strings, and send the corresponding string to the String Queue.

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    static unsigned long ulReceivedNumber;

    if(GPIO_Pin == GPIO_PIN_13)
    {
        osSemaphoreRelease(myBinarySemHandle);
        putsUSART1("\n\rUser Button pressed. Interrupt callback: send selected strings to StringQueue.\n\r ");
        while(osMessageQueueGetCount(IntegerQueueHandle) != 0){
            osMessageQueueGet(IntegerQueueHandle, &ulReceivedNumber, 0, 0);
            ulReceivedNumber &= 0x03;

            osMessageQueuePut(StringQueueHandle, &(pcStrings[ulReceivedNumber]), 0, 0);
        }
    }
}
```

f) Define the Integer Generation thread, which puts an integer into the IntegerQueue and waits forever until the binary semaphore is to be released by the interrupt function.
   - Before the infinite for loop, acquire the binary semaphore if it exists; otherwise, continue.
   - Within the infinite for loop, use a *for* loop structure to send five integers to the Integer Queue, then wait for the interrupt to release the binary semaphore forever.

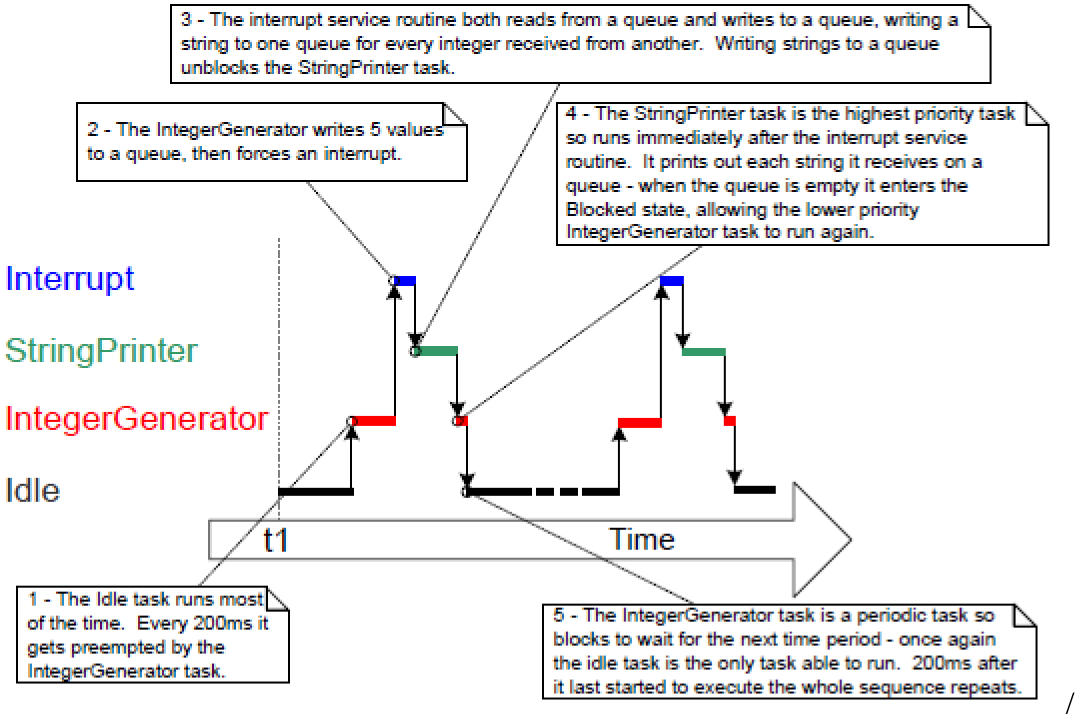
```
/* USER CODE END Header_IntegerGThread */
void IntegerGThread(void *argument)
{
  /* USER CODE BEGIN 5 */
    unsigned long ulValueToSend = 0;
    osSemaphoreAcquire(myBinarySemHandle, 0);
  /* Infinite loop */
  for(;;)
  {
    osDelay(1000);
    putsUSART1("\n\rInteger Generation Thread - send five integers to IntegerQueue\r");
    for(int i = 0; i < 5; i++)
    {
        osMessageQueuePut(IntegerQueueHandle, &ulValueToSend, 0, 0);
        ulValueToSend++;
    }

    putsUSART1("\n\rWait for an interrupt, please press user button. \n\r");
    osSemaphoreAcquire(myBinarySemHandle, 0);
  }
  /* USER CODE END 5 */
}
```

g) Define the StringPrinter thread, which simply receives the string from the StringQueue and print it in the tera term.

```c
/* USER CODE END Header_StringPrinterThread */
void StringPrinterThread(void *argument)
{
  /* USER CODE BEGIN StringPrinterThread */
    char *pcString;
  /* Infinite loop */
  for(;;)
  {
      osMessageQueueGet(StringQueueHandle, &pcString, 0, osWaitForever);

      sprintf(buffer, "\n\rStringPrinter Thread - Received: %s\n\r", pcString);
      putsUSART1(buffer);

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

5. The output produced is shown below. As can be seen, the interrupt receives all five integers and produces five strings in response.

```
FreeRTOS lab14.
Integer Generation Thread - Send five integers to IntegerQueue
Wait for an interrupt, please press user button.
Interrupt Callback - Send Selected Strings to StringQueue.
StringPrinter Thread - Received: String 0
StringPrinter Thread - Received: String 1
StringPrinter Thread - Received: String 2
StringPrinter Thread - Received: String 3
StringPrinter Thread - Received: String 0
```


3 - The interrupt service routine both reads from a queue and writes to a queue, writing a string to one queue for every integer received from another. Writing strings to a queue unblocks the StringPrinter task.
2 - The IntegerGenerator writes 5 values to a queue, then forces an interrupt.
4 - The StringPrinter task is the highest priority task so runs immediately after the interrupt service routine. It prints out each string it receives on a queue - when the queue is empty it enters the Blocked state, allowing the lower priority IntegerGenerator task to run again.
Interrupt
StringPrinter
IntegerGenerator
Idle
t1
Time
1 - The Idle task runs most of the time. Every 200ms it gets preempted by the IntegerGenerator task.
5 - The IntegerGenerator task is a periodic task so blocks to wait for the next time period - once again the idle task is the only task able to run. 200ms after it last started to execute the whole sequence repeats.