Real-time Embedded Systems Lab Manual

Queue Management in FreeRTOS (lab 10 - 11)

1. Characteristics of a Queue in FreeRTOS

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

Writing data to a queue causes a byte-for-byte copy of the data to be stored in the queue itself. Reading data from a queue causes the copy of the data to be removed from the queue.

Access by multiple threads

Queues are objects in their own right that are not owned by or assigned to any particular thread. Any number of threads can write to the same queue and any number of threads can read from the same queue. A queue having multiple writers is very common, whereas a queue having multiple readers is quite rare.

Blocking on Queue Reads

When a thread attempts to read from a queue it can optionally specify a 'block' time. This is the time the thread should be kept in the Blocked state to wait for data to be available from the queue should the queue already be empty. A thread that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another thread or interrupt places data into the queue. The thread will also be moved automatically from the Blocked state to the Ready state if the specified block time expires before data become available.

Queue can have multiple readers so it is possible for a single queue to have more than one thread blocked on it waiting for data. When this is the case, only one thread will be unblocked when data becomes available. The thread that is unblocked will always be the highest priority thread that is waiting for data. If the blocked threads have equal priority, the thread that has been waiting for data the longest will be unblocked.

Blocking on Queue Writes

As reading from a queue, a thread can optionally specify a block time when writing to a queue. In this case, the block time is the maximum time the thread should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.

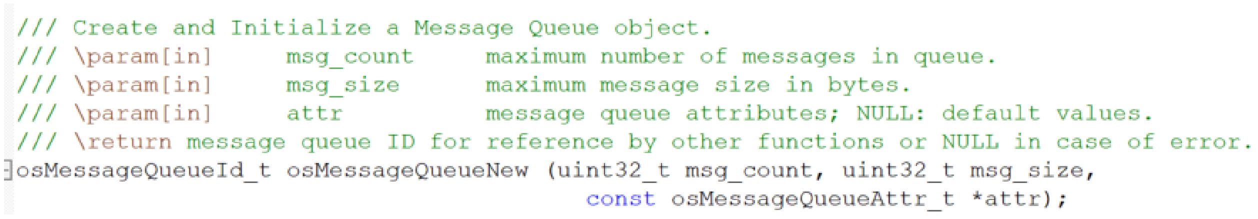
Queue can have multiple writers, so it is possible for a full queue to have more than one thread blocked on it waiting to complete a send operation. When this is the case, only one therad will be unblocked when free space on the queue becomes available. The thread that is unblocked will always be the highest priority thread that is waiting for space. If the blocked threads have equal priority, the thread that has been waiting for space the longest will be unblocked.

2. Using a message Queue

osMessageQueueNew () API function

A queue must be explicitly created before it can be used. osMessageQueueNew () is used to create and initialize a message queue object and returns a message queue object identifier if there is no error.

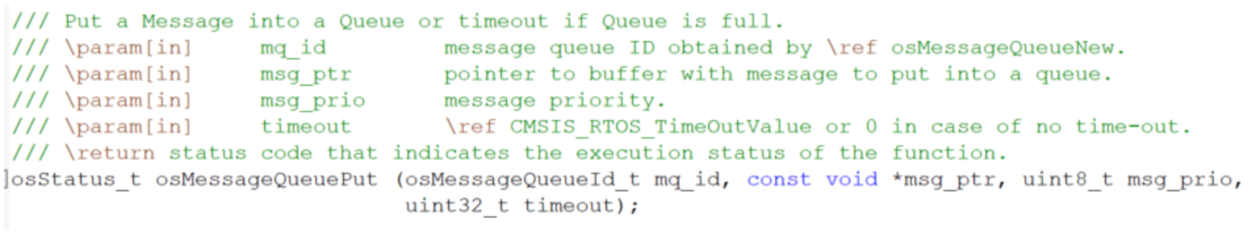
FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. osMessageCreate () will return NULL if there is insufficient heap RAM available for the queue to be created. The osMessageQueueNew () API function prototype (declared in *cmsis_os2.h* file) is as the following.



```
/// Create and Initialize a Message Queue object.
/// \param[in]     msg_count     maximum number of messages in queue.
/// \param[in]     msg_size      maximum message size in bytes.
/// \param[in]     attr          message queue attributes; NULL: default values.
/// \return message queue ID for reference by other functions or NULL in case of error.
osMessageQueueId_t osMessageQueueNew (uint32_t msg_count, uint32_t msg_size,
                                      const osMessageQueueAttr_t *attr);
```

osMessageQueuePut () API function

osMessageQueuePut () is used to send data to the back (tail) of a queue. Their prototypes are as the following.



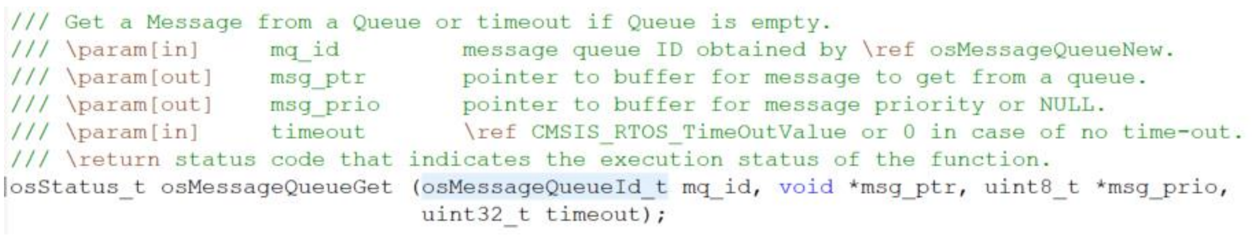
```
/// Put a Message into a Queue or timeout if Queue is full.
/// \param[in]     mq_id         message queue ID obtained by \ref osMessageQueueNew.
/// \param[in]     msg_ptr       pointer to buffer with message to put into a queue.
/// \param[in]     msg_prio      message priority.
/// \param[in]     timeout       \ref CMSIS_RTOS_TimeOutValue or 0 in case of no time-out.
/// \return status code that indicates the execution status of the function.
osStatus_t osMessageQueuePut (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio,
                              uint32_t timeout);
```

/* *timeout* is the maximum amount of time the thread should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

/* Common return values: if data was successfully sent to the queue, *osOK* will be returned; if data could not be written to the queue in the given timeout, *osErrorTimeout* will be returned. */

osMessageQueueGet() API function

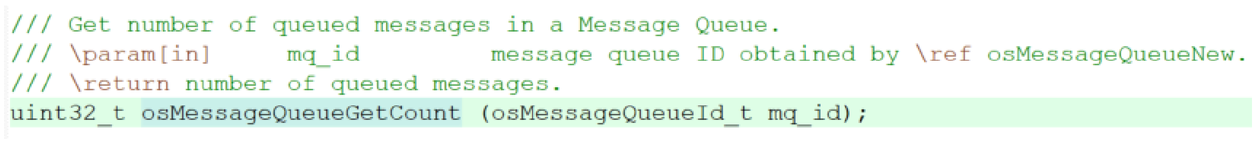
It is used to receive (read) an item from a queue. It prototype is as the following (cmsis_os2.h).



```
/// Get a Message from a Queue or timeout if Queue is empty.
/// \param[in]     mq_id         message queue ID obtained by \ref osMessageQueueNew.
/// \param[out]    msg_ptr       pointer to buffer for message to get from a queue.
/// \param[out]    msg_prio      pointer to buffer for message priority or NULL.
/// \param[in]     timeout       \ref CMSIS_RTOS_TimeOutValue or 0 in case of no time-out.
/// \return status code that indicates the execution status of the function.
osStatus_t osMessageQueueGet (osMessageQueueId_t mq_id, void *msg_ptr, uint8_t *msg_prio,
                              uint32_t timeout);
```

osMessageQueueGetCount() API function

This is used to query the number of items that are currently in the queue. It prototype is as the following.



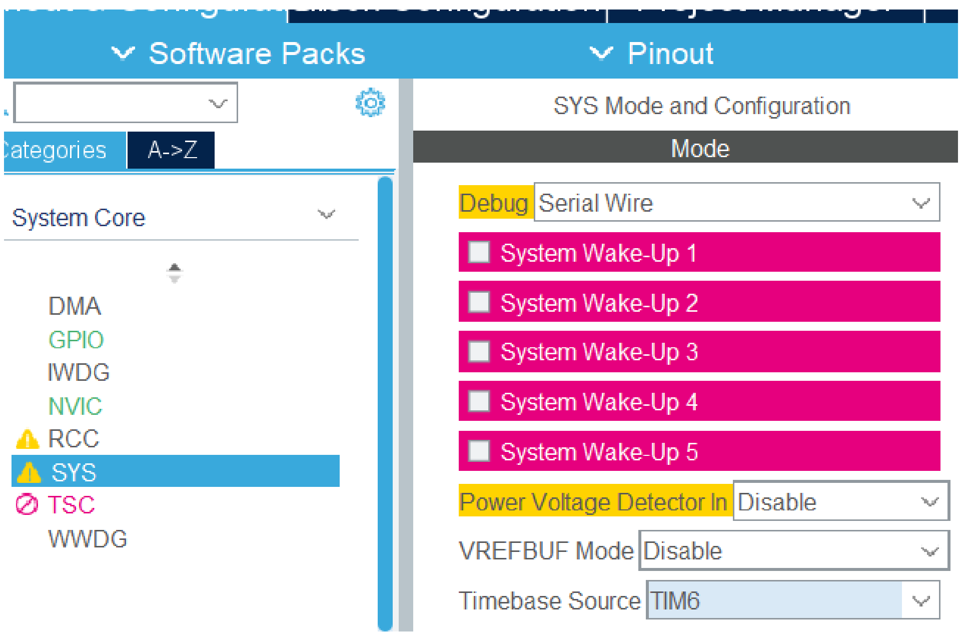
```
/// Get number of queued messages in a Message Queue.
/// \param[in]     mq_id         message queue ID obtained by \ref osMessageQueueNew.
/// \return number of queued messages.
uint32_t osMessageQueueGetCount (osMessageQueueId_t mq_id);
```

Lab 10 Blocking when receiving a message from a queue

This part demonstrates a queue being created, data being sent to the queue from multiple threads, and data being received from the queue. The queue is created to hold data items of type long. The threads that send to the queue do not specify a block time, whereas the thread that receives from the queue does.

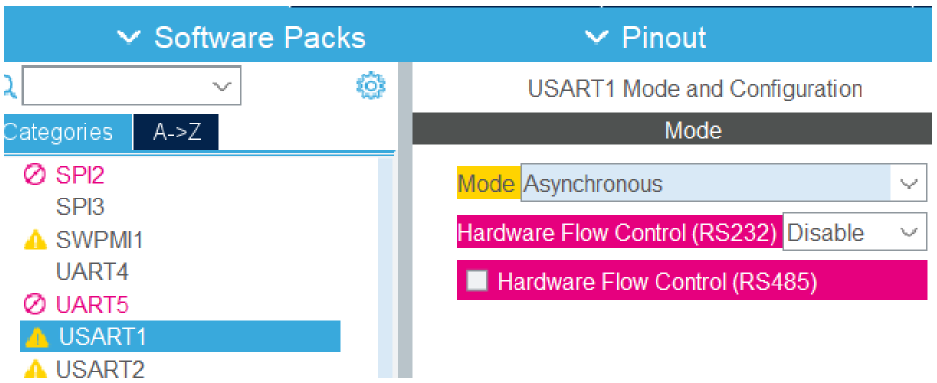
The priority of the threads that send to the queue is **lower** than the priority of the thread that receives from the queue. This means that the queue should never contain more than one item because, as soon as data is sent to the queue the receiving thread will unlock, pre-empt the sending thread, and remove the data- leaving the queue empty once again.

1. Assume that we use the MCU configuration file of lab7 as a start point and copy it to a new directory for lab10. 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.



Debug Serial Wire
System Wake-Up 1
System Wake-Up 2
System Wake-Up 3
System Wake-Up 4
System Wake-Up 5
Power Voltage Detector In Disable
VREFBUF Mode Disable
Timebase Source TIM6

   b) The **Mode** of the **USART1** module under the **Connectivity** category is set **Asynchronous;**



USART1 Mode and Configuration

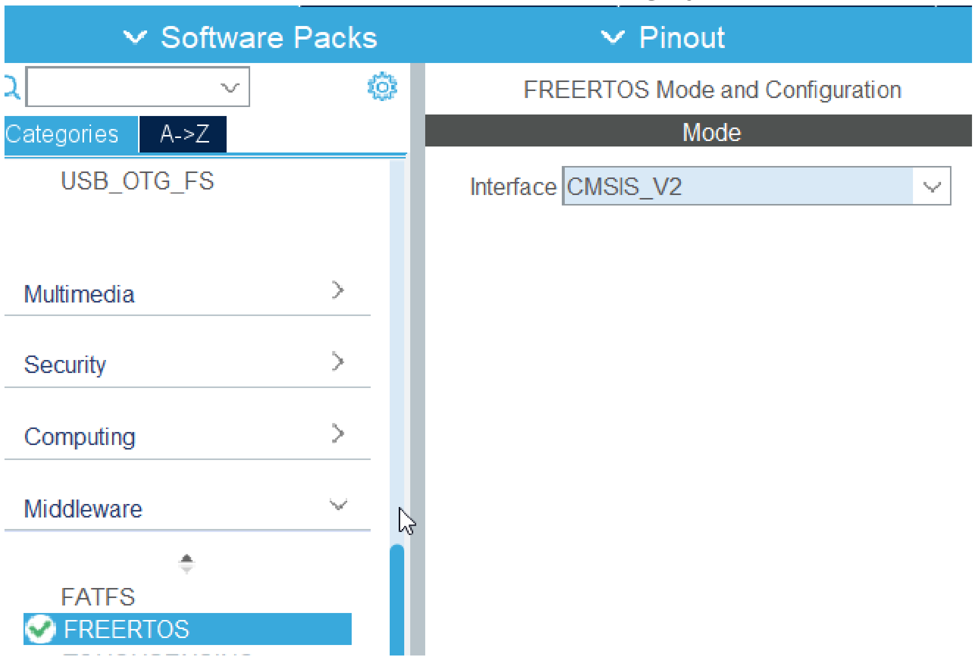
Mode

Mode: Asynchronous

Hardware Flow Control (RS232): Disable

Hardware Flow Control (RS485)

c) The **interface** of the **FreeRTOS** under **Middleware** category is set **CMSIS_V2**.



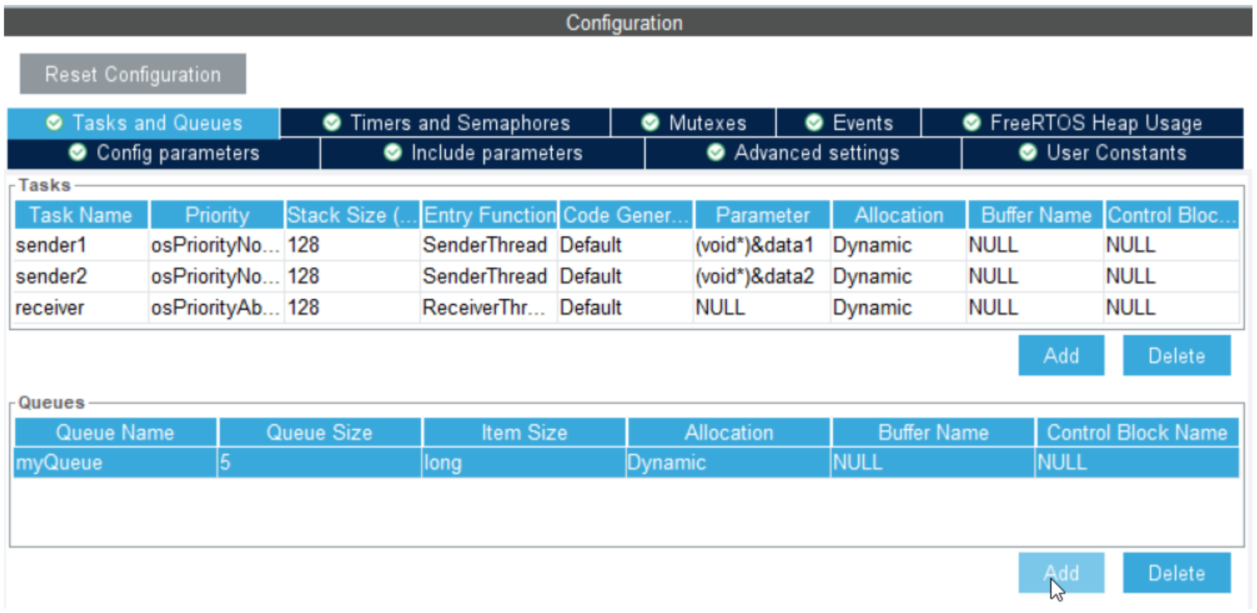
FREERTOS Mode and Configuration

Mode

Interface: 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. Create two Sender threads, one Receiver thread and a queue.


Configuration

Reset Configuration

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 Function	Code Gener...	Parameter	Allocation	Buffer Name	Control Bloc...
sender1	osPriorityNo...	128	SenderThread	Default	(void*)&data1	Dynamic	NULL	NULL
sender2	osPriorityNo...	128	SenderThread	Default	(void*)&data2	Dynamic	NULL	NULL
receiver	osPriorityAb...	128	ReceiverThr...	Default	NULL	Dynamic	NULL	NULL

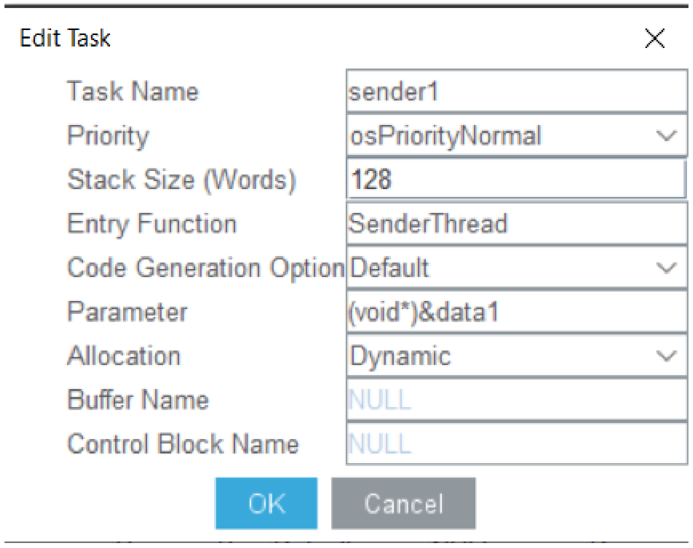
Add Delete

Queues

Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block Name
myQueue	5	long	Dynamic	NULL	NULL

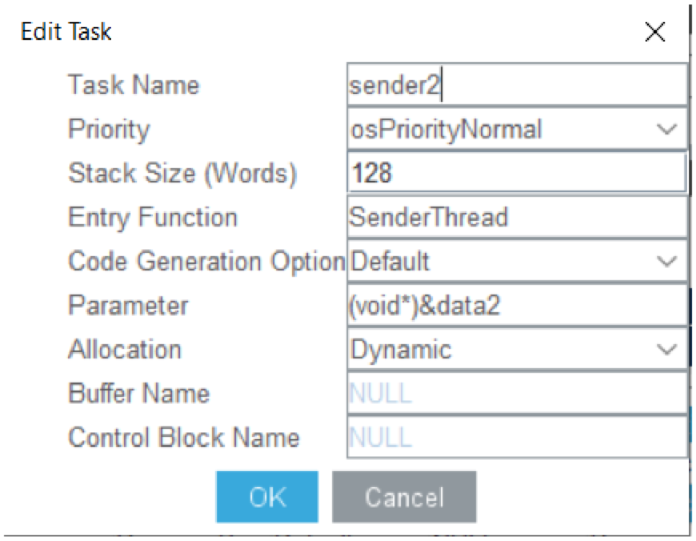
Add Delete

a) Edit sender1 and sender2, set their priority as *osPriorityNormal*.


Edit Task

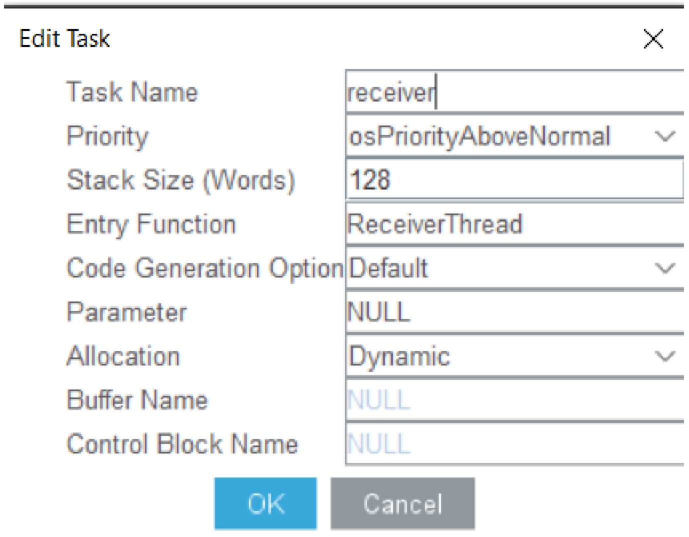
Task Name	sender1
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	SenderThread
Code Generation Option	Default
Parameter	(void*)&data1
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

OK Cancel


Edit Task

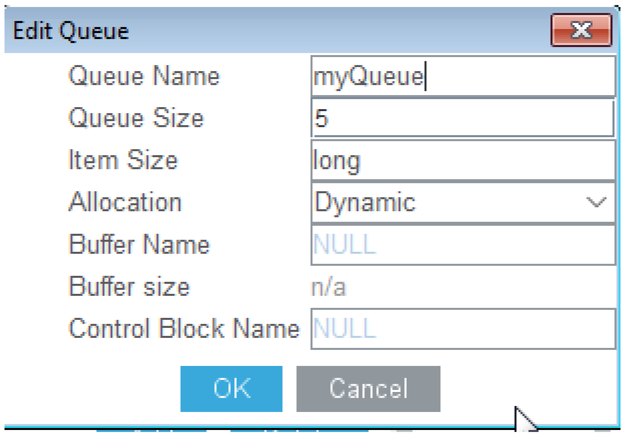
Task Name	sender2
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	SenderThread
Code Generation Option	Default
Parameter	(void*)&data2
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

b) Add receiver thread, set its priority as *osPriorityAboveNormal*.


Edit Task

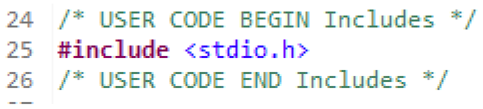
Task Name	receiver
Priority	osPriorityAboveNormal
Stack Size (Words)	128
Entry Function	ReceiverThread
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL

c) Add myQueue, set its size as 5 and item size as long.


Edit Queue

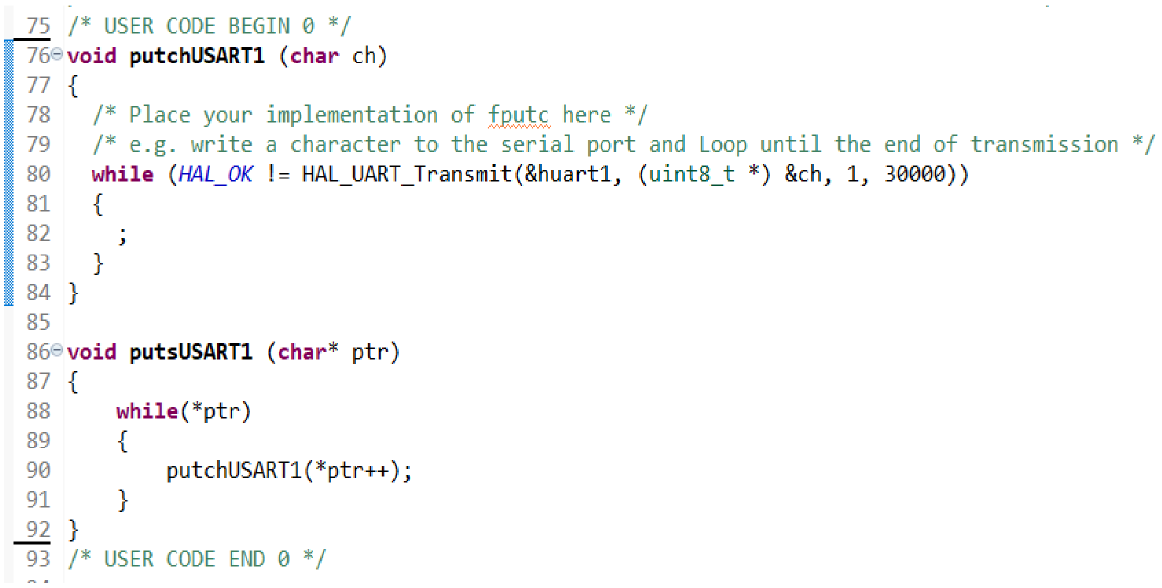
Queue Name	myQueue
Queue Size	5
Item Size	long
Allocation	Dynamic
Buffer Name	NULL
Buffer size	n/a
Control Block Name	NULL

3. Save the configuration file to generate the code.
4. 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()*.



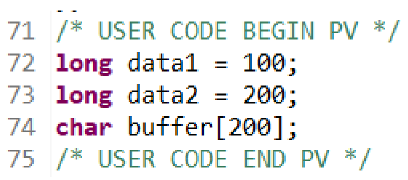
```c
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */
```

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



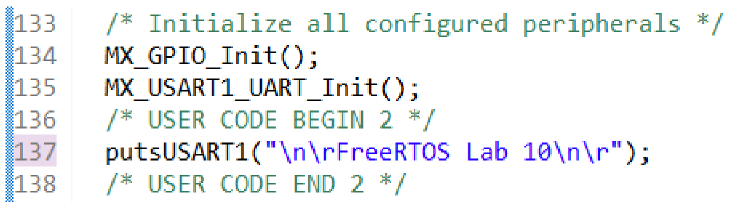
```c
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

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

   a) Within /* USER CODE BEGIN PV */ and /* USER CODE END PV */, declare two global constant integers data1 and data2 with the *long* type, and initialize them by 100 and 200 respectively. They are the data sent by two sender tasks to the queue. Also, define a char type array called buffer.



```c
/* USER CODE BEGIN PV */
long data1 = 100;
long data2 = 200;
char buffer[200];
/* USER CODE END PV */
```

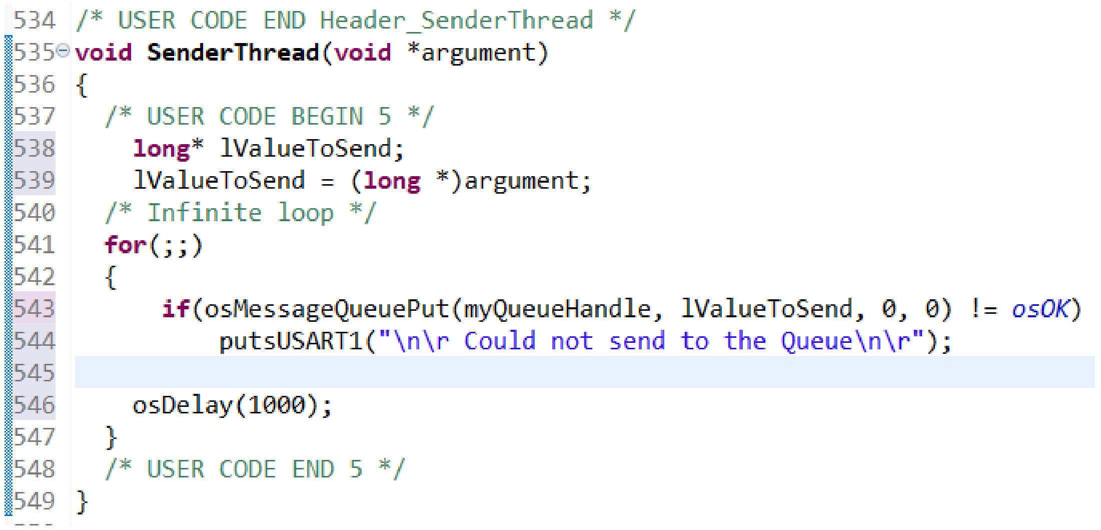
   b) Edit the main() function by calling the putsUSART1() as below.



```c
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS Lab 10\n\r");
  /* USER CODE END 2 */
```

   d) Edit the SenderThread() function.

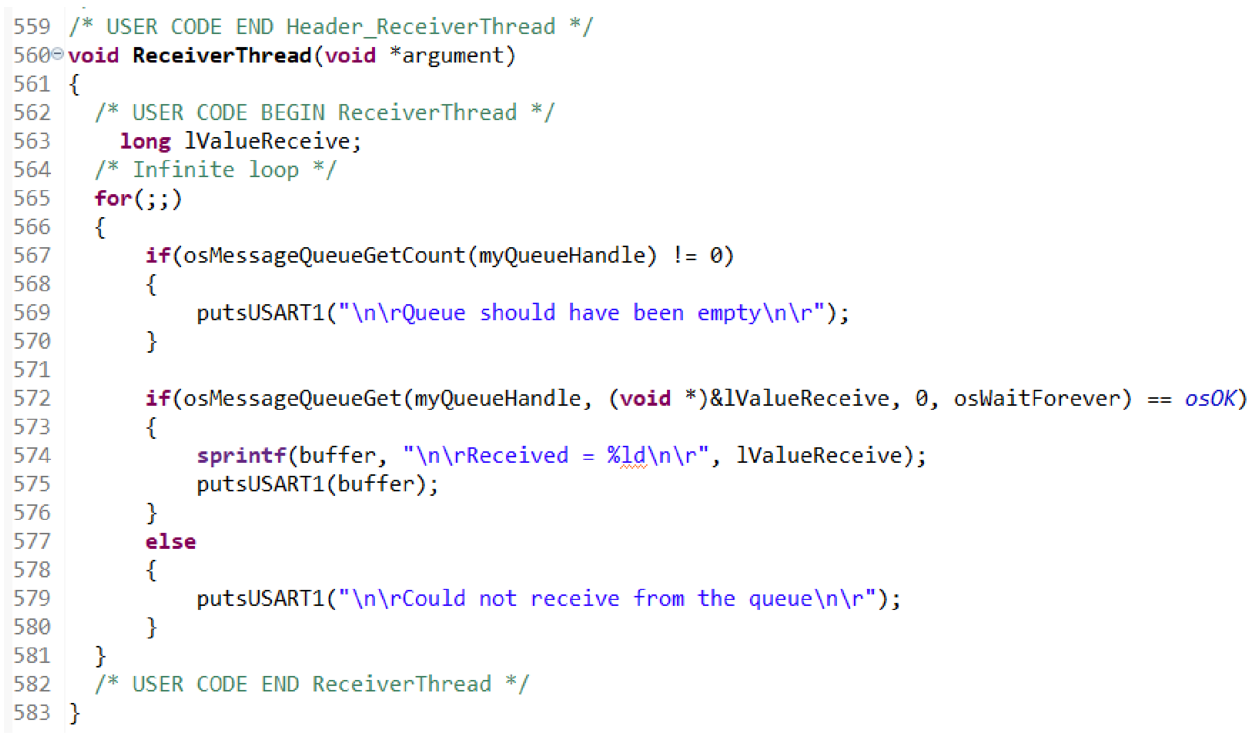
- Firstly, declare a local variable: *lValueToSend* with *long* type *pointer* to hold the address of the variable to be written to the queue;
- Then cast input parameter *argument* from (*void* *) to (*long* *) type and assign it to *lValueToSend*;
- Inside the infinite *for(;;)* loop, call *osMessageQueuePut*() function to write the data *lValueToSend* to the *myQueueHandle* with *the Wait* time set to 0, and check if the Write operation returns osOK. If not, send a string "\n\r Could not send to the queue.\n\r" to display.



```
/* USER CODE END Header_SenderThread */
void SenderThread(void *argument)
{
  /* USER CODE BEGIN 5 */
    long* lValueToSend;
    lValueToSend = (long *)argument;
  /* Infinite loop */
  for(;;)
  {
      if(osMessageQueuePut(myQueueHandle, lValueToSend, 0, 0) != osOK)
          putsUSART1("\n\r Could not send to the Queue\n\r");

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

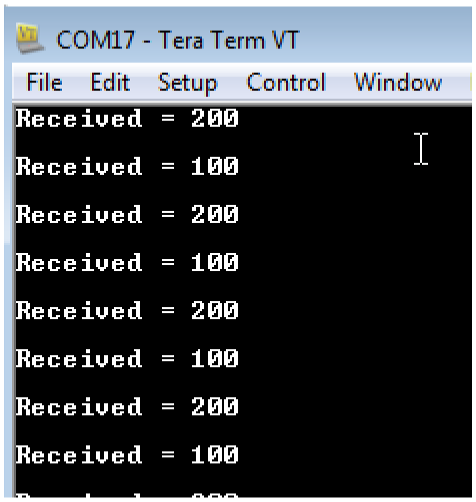
c) Edit ReceiverThread() function.

- Firstly, declare a local variable: *lValueReceive*with *long* type to store the variable received from the queue;
- Inside the *for(;;){}* loop body, check whether the queue is empty by calling *osMessageQueueGetCount* () function, if not empty (i.e. 0), print a string "\n\rQueue should have been empty!\n\r". Note, this message should not be displayed in the normal situation. Why? (The Receiver thread has higher priority than two Senders thread.)
- Then, call *osMessageQueueGet*() function to receive *lValueReceive* from the *Queue* with *the* waiting time as *osWaitForever*, and check if the return value is osOK, if yes, print a string and a number with *"Received = "* and *lValueReceive* respectively; if not, send a string "\n\rCould not receive from the queue.\n\r" to display.



```c
/* USER CODE END Header_ReceiverThread */
void ReceiverThread(void *argument)
{
  /* USER CODE BEGIN ReceiverThread */
    long lValueReceive;
  /* Infinite loop */
  for(;;)
  {
      if(osMessageQueueGetCount(myQueueHandle) != 0)
      {
          putsUSART1("\n\rQueue should have been empty\n\r");
      }

      if(osMessageQueueGet(myQueueHandle, (void *)&lValueReceive, 0, osWaitForever) == osOK)
      {
          sprintf(buffer, "\n\rReceived = %ld\n\r", lValueReceive);
          putsUSART1(buffer);
      }
      else
      {
          putsUSART1("\n\rCould not receive from the queue\n\r");
      }
  }
  /* USER CODE END ReceiverThread */
}
```

Expected outputs are shown here.



```
Received = 200

Received = 100

Received = 200

Received = 100

Received = 200

Received = 100

Received = 200

Received = 100
```

Using Message Queue to transfer compound types

It is common for a thread to receive data from multiple sources on a single Queue. Often, the receiver of the data needs to know where the data came from, to allow it to determine how the data should be processed. A simple way to achieve this is to use the Queue to transfer structures where both the value of the data and the sources of the data are contained in the structure fields.

Lab 11 Blocking when sending to a Queue or sending structures on a Queue

This part is similar to lab 10, but the task priorities are reversed so the receiving task has a lower priority than the sending tasks. Also the Queue is used to pass structures, rather than simple long integers, between the tasks.

The structure used in this part is as the following.

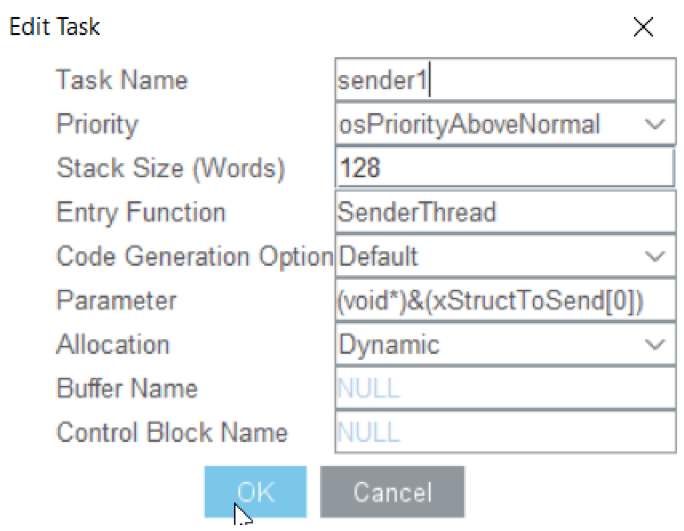
```
typedef struct  // define the structure type that will be passed on the Queue
{
        unsigned char ucValue;
        unsigned char ucSource;
}xData;

const xData xStructsToSend[2] =
{
        {100, mainSENDER_1},
        {200, mainSENDER_2}
};
```

You could copy *lab10.ioc* as *lab11.ioc* as the starting point.

1. Select Middleware>>FREERTOS, open the "Tasks and Queues" tab on the FREERTOS Configuration panel.
   Modify the *sender1* and *sender2* created in lab10.
   - Change **Priority** to osPriorityAboveNormal
   - Change **Parameter** to (void*)&(xStructToSend[0]), (void*)&(xStructToSend[1]) respectively

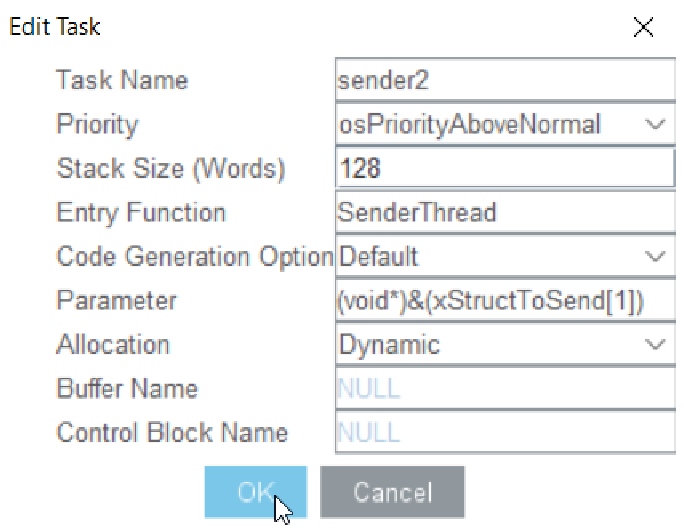
   Modify *receiver* by lowering its Priority to **osPriorityNormal**.

   And modify the queue by changing its **Item Size** to **xData;** reducing **Queue Size** to 3.



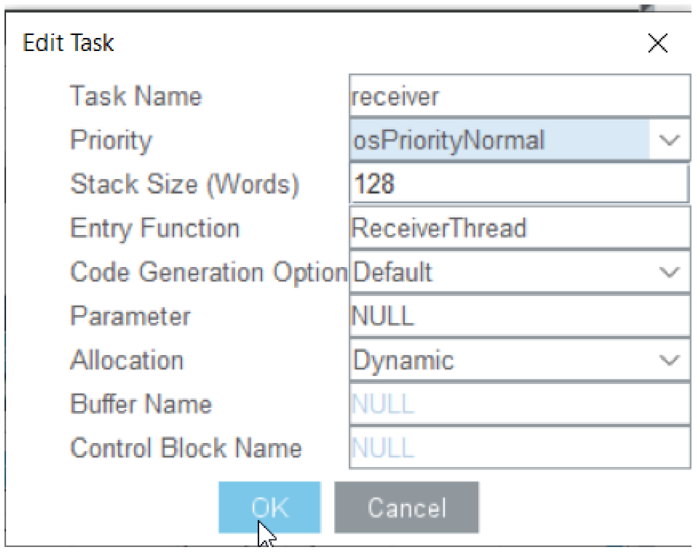
Edit Task

Task Name	sender1
Priority	osPriorityAboveNormal
Stack Size (Words)	128
Entry Function	SenderThread
Code Generation Option	Default
Parameter	(void*)&(xStructToSend[0])
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL



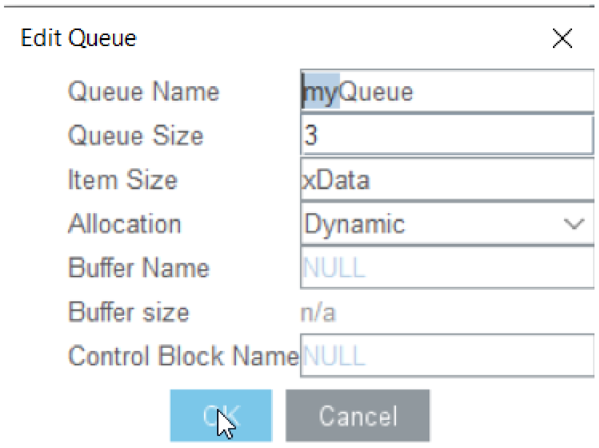
Edit Task

Task Name	sender2
Priority	osPriorityAboveNormal
Stack Size (Words)	128
Entry Function	SenderThread
Code Generation Option	Default
Parameter	(void*)&(xStructToSend[1])
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL



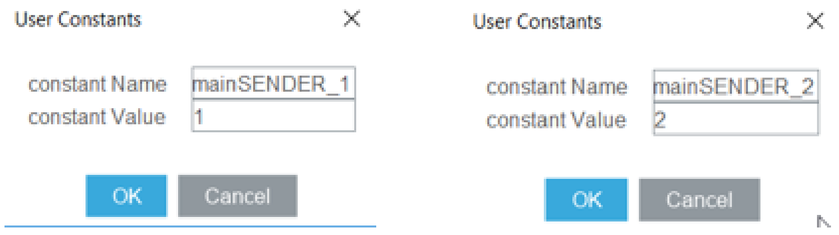
Edit Task

Task Name	receiver
Priority	osPriorityNormal
Stack Size (Words)	128
Entry Function	ReceiverThread
Code Generation Option	Default
Parameter	NULL
Allocation	Dynamic
Buffer Name	NULL
Control Block Name	NULL


Edit Queue

Queue Name	myQueue
Queue Size	3
Item Size	xData
Allocation	Dynamic
Buffer Name	NULL
Buffer size	n/a
Control Block Name	NULL

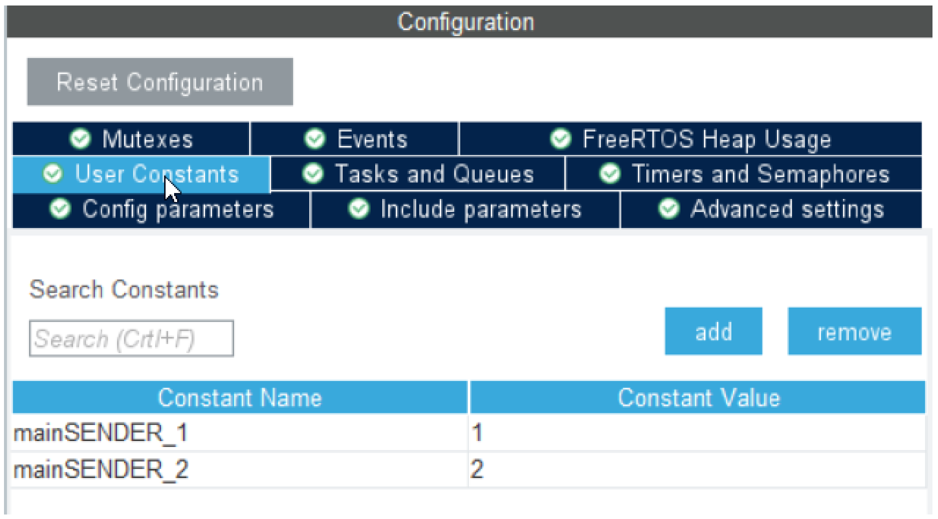
2. Select **User Constants** tab, and define two User Constants as below:


User Constants

constant Name	mainSENDER_1
constant Value	1

User Constants

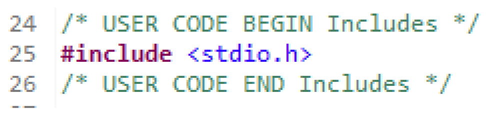
constant Name	mainSENDER_2
constant Value	2



Constant Name	Constant Value
mainSENDER_1	1
mainSENDER_2	2

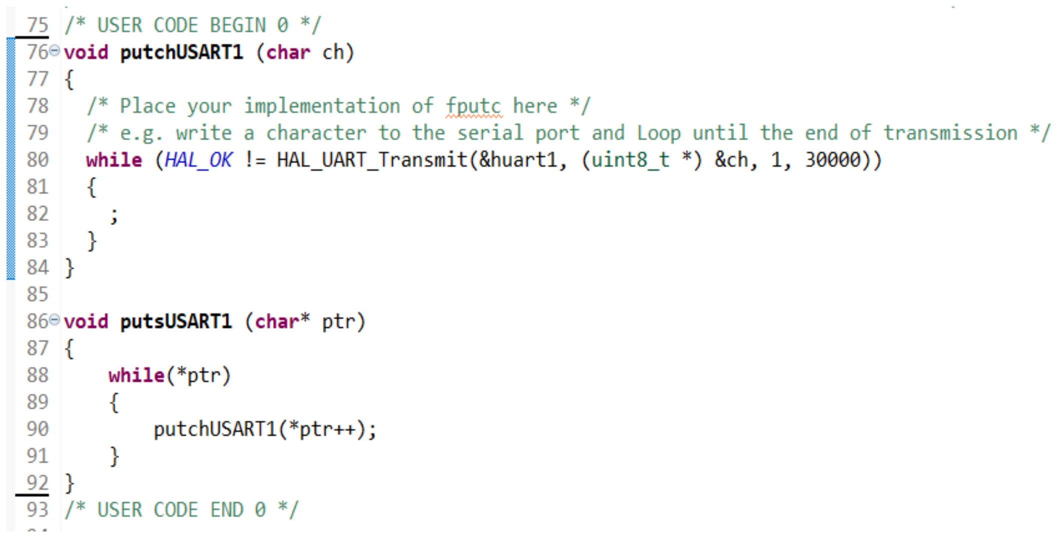
3. Generate code and open the project.

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()*.



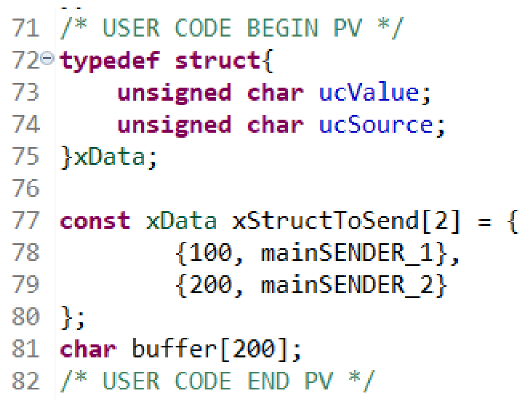
```
/* USER CODE BEGIN Includes */
#include <stdio.h>
/* USER CODE END Includes */
```

b) Define putchUSART1() and putsUSART1() to use the USART1 channel as in lab10.



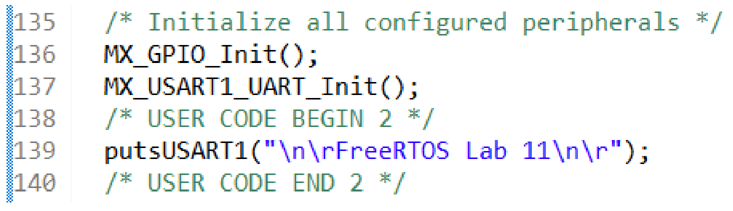
```
/* USER CODE BEGIN 0 */
void putchUSART1 (char ch)
{
  /* Place your implementation of fputc here */
  /* e.g. write a character to the serial port and Loop until the end of transmission */
  while (HAL_OK != HAL_UART_Transmit(&huart1, (uint8_t *) &ch, 1, 30000))
  {
    ;
  }
}

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

c) Define a *struct xData* data type and declare a global array variable *xStructToSend* with this type and initialize it, besides the char type array variable *buffer*.



```
/* USER CODE BEGIN PV */
typedef struct{
    unsigned char ucValue;
    unsigned char ucSource;
}xData;

const xData xStructToSend[2] = {
        {100, mainSENDER_1},
        {200, mainSENDER_2}
};
char buffer[200];
/* USER CODE END PV */
```

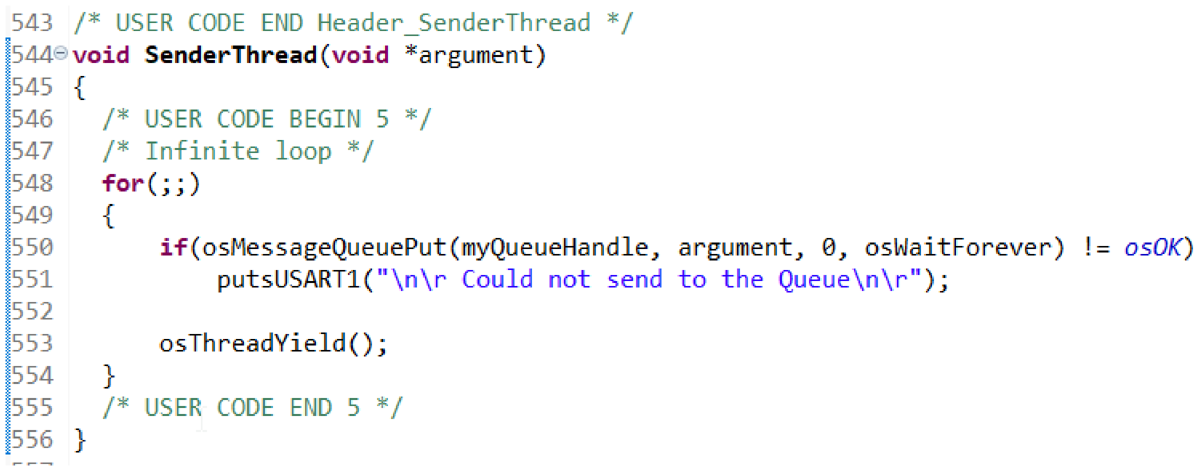
d) In the main() function, add the following statements, and keep the generated code unchanged.



```
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_USART1_UART_Init();
  /* USER CODE BEGIN 2 */
  putsUSART1("\n\rFreeRTOS Lab 11\n\r");
  /* USER CODE END 2 */
```

e) Edit SenderThread().

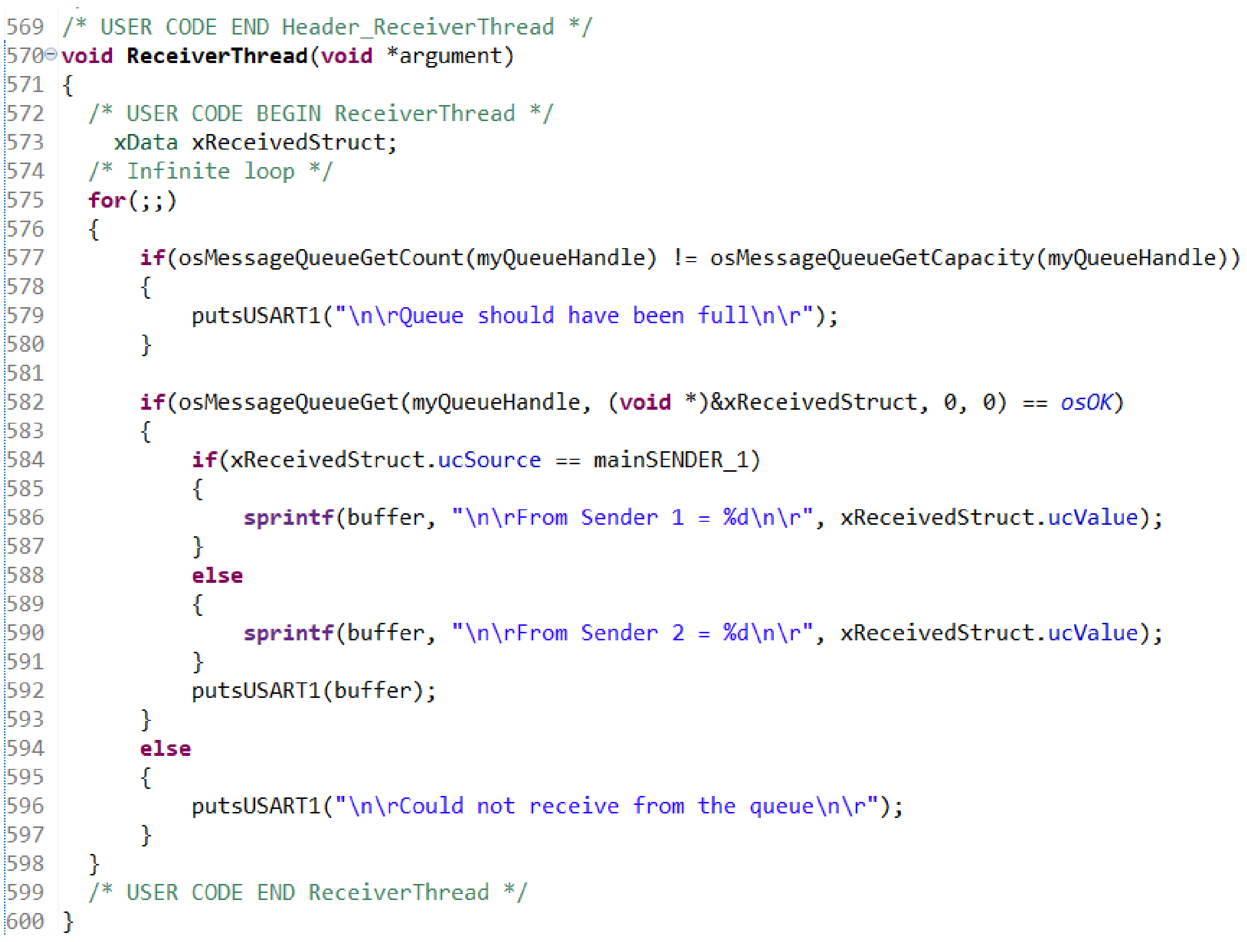
- Inside the infinite *for(;;)* loop, call *osMessageQueuePut*() function to write the data passed by function parameter *argument* to the *myQueueHandle* with *the Wait* time set to *osWaitForever*, and check if the Write operation returns osOK. If not, print a string “\n\r Could not send to the queue.\n\r”.



```
/* USER CODE END Header_SenderThread */
void SenderThread(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
      if(osMessageQueuePut(myQueueHandle, argument, 0, osWaitForever) != osOK)
          putsUSART1("\n\r Could not send to the Queue\n\r");

      osThreadYield();
  }
  /* USER CODE END 5 */
}
```

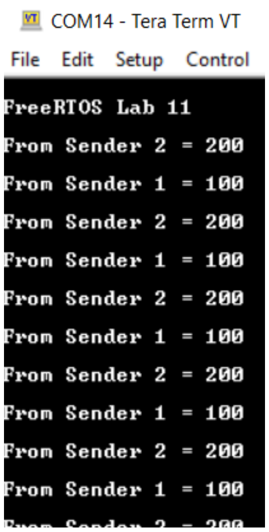
f) Edit ReceiverThread() function,

- Firstly, declare a variable "xReceivedStruct" with the type of "xData" struct. It will hold the values received from the queue.
- Inside the infinite *for(;;)* loop, check whether xQueue is full (use *osMessageQueueGetCapacity(myQueueHandle)*) by calling the osMessageQueueGetCount () with the input parameter *myQueueHandle*. If the return value of this function is not equal to 3, print a message "Queue should have been full!\r\n".
- Then, call *osMessageQueueGet*() function to receive *data* from the *Queue* with the waiting time as *0*, and check if the return value is osOK, if yes, further check the message source and print a string with sender ID and a value respectively; if not, print a string “\n\rCould not receive from the queue.\n\r”.



```c
/* USER CODE END Header_ReceiverThread */
void ReceiverThread(void *argument)
{
  /* USER CODE BEGIN ReceiverThread */
    xData xReceivedStruct;
  /* Infinite loop */
  for(;;)
  {
      if(osMessageQueueGetCount(myQueueHandle) != osMessageQueueGetCapacity(myQueueHandle))
      {
          putsUSART1("\n\rQueue should have been full\n\r");
      }

      if(osMessageQueueGet(myQueueHandle, (void *)&xReceivedStruct, 0, 0) == osOK)
      {
          if(xReceivedStruct.ucSource == mainSENDER_1)
          {
              sprintf(buffer, "\n\rFrom Sender 1 = %d\n\r", xReceivedStruct.ucValue);
          }
          else
          {
              sprintf(buffer, "\n\rFrom Sender 2 = %d\n\r", xReceivedStruct.ucValue);
          }
          putsUSART1(buffer);
      }
      else
      {
          putsUSART1("\n\rCould not receive from the queue\n\r");
      }
  }
  /* USER CODE END ReceiverThread */
}
```

Please note: The sending thread specifies a block time of *osWaitForever* instead of 0. It will call osThreadYield() which Pass control to next thread that is in state **READY** only when the queue is full.
Thus, this receiver task always expects the number of items in the queue to be equal to the queue capacity = 3, then this task can start to execute.



```
FreeRTOS Lab 11
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
```