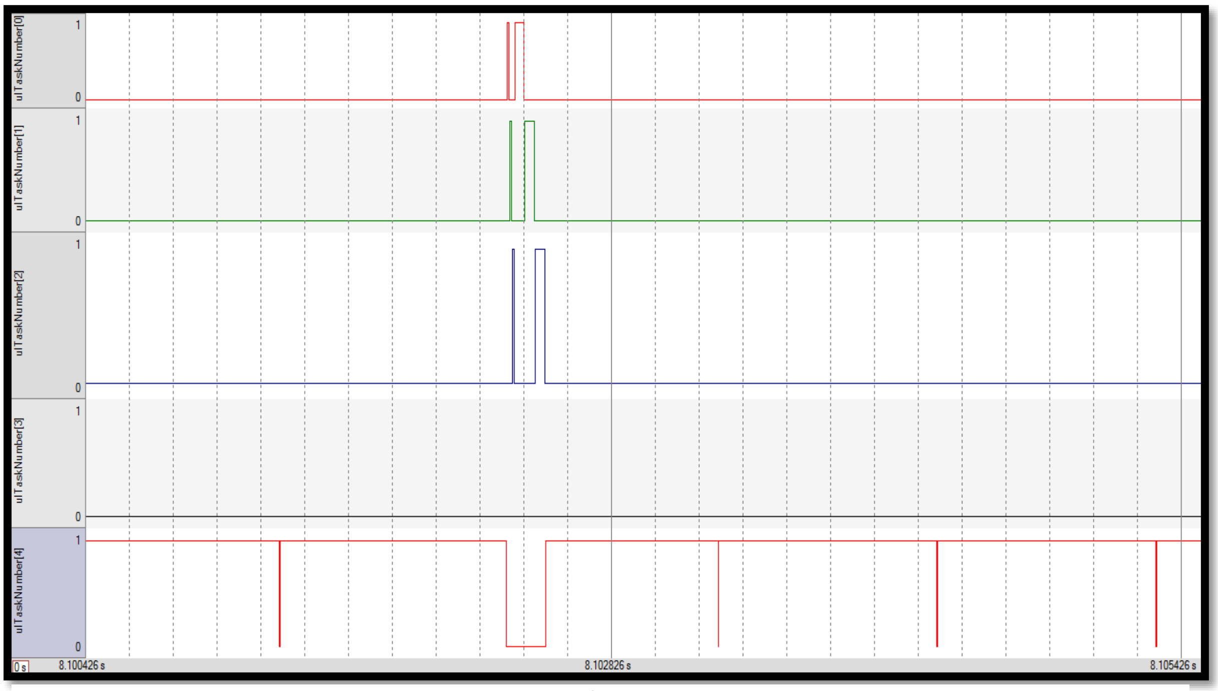
Assignment 2

Q1.

Code Implementation:

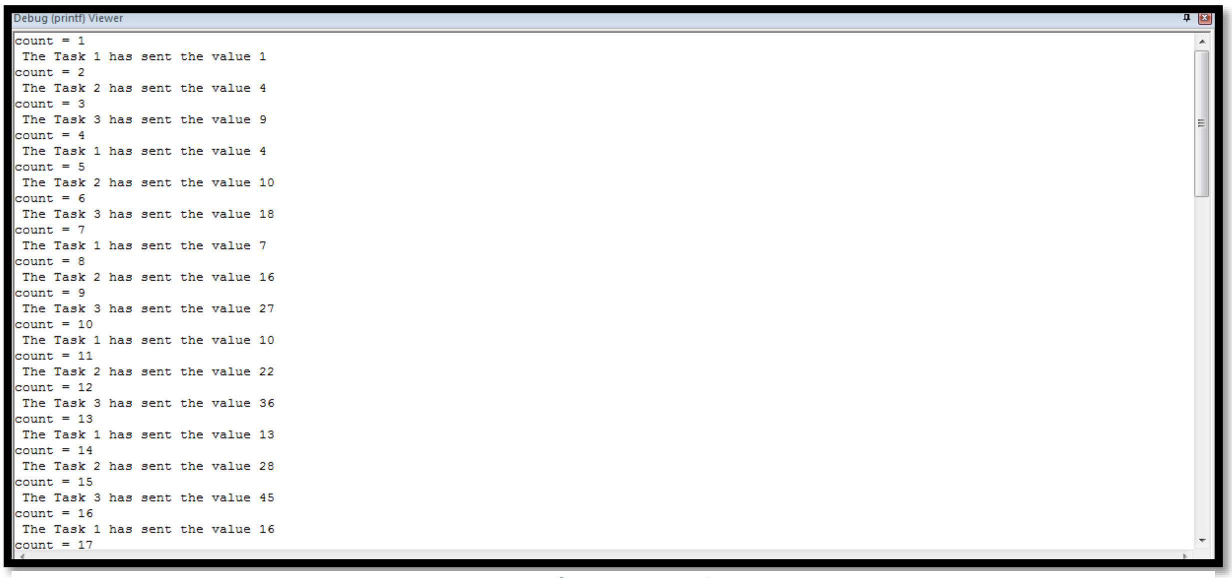
- **Created 4 tasks t1, t2, t3, tp. (t1, t2, t3 –same priority and tp –less priority than t1)**
- **Set the Task Numbers for t1, t2, t3, tp as 1,2,3,4 respectively.**
- **Created 4 Queues q1, q2, q3, qp.**
- **Tasks t1 will wait for the data to appear on the Queue q1, and t2 for q2, t3 for q3.**
- **The Tasktp will periodical wakeup as it is periodic task, whenever it wakes up it increments the value of temp and count.**
- **The count Value is used to loop between 1 ,2,3.,Whenever count value is 1 ,the count value is sent to the q1,if count is 2 ,count is sent to q1 and soon.**
- **Whenever any task (t1, t2, t3) gets the count value, that will multiply its id with the count value and send it to the queue qp.**



Logic Analyser Output

Explanation:

- **At the starting, temp variable becomes 1 and it enters into q1, then task t1 executes and multiplies its id value and send that to qp. Whenever it sends the value to qp, no message will be there in q1 so t1 will go to wait state.**
- **Meanwhile temp becomes 2, and so t2 executes and multiplies its id value and send that to qp. Whenever it sends the value to qp, no message will be there in q2 so t2 will go to wait state.**
- **Similarly t3 also executes when temp becomes 3 and this cycle repeates.So we can observe the execution of t1, t2, t3 in that specific order.**
- **And during when t1, or t2 or t3 executes, idle task won't execute, else if no other task is executing then idle task will run.**



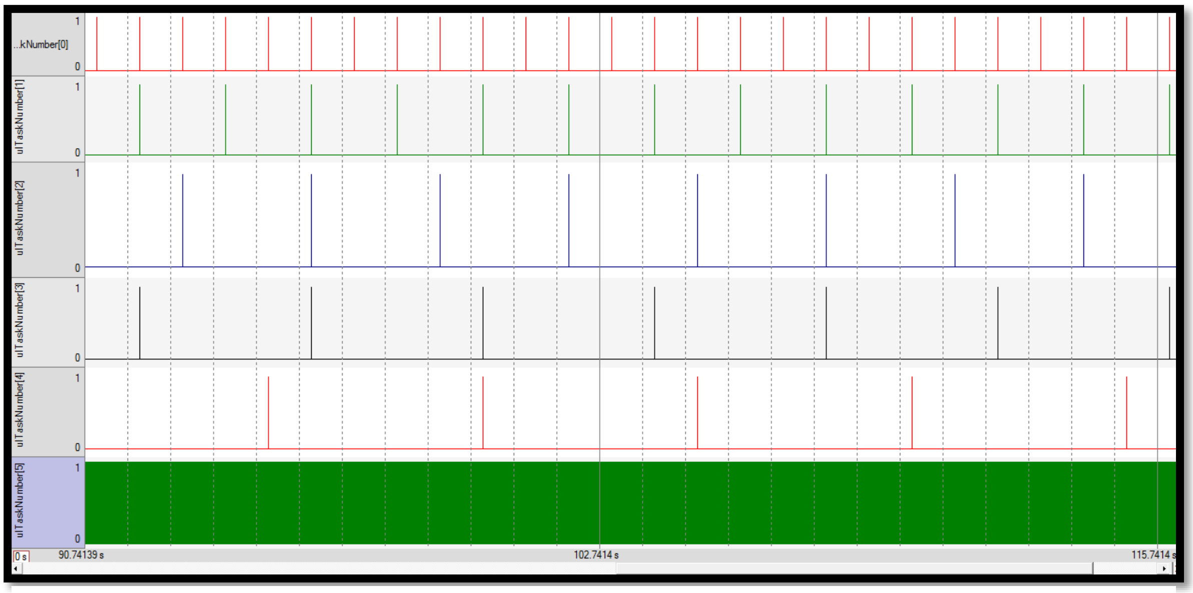
```
count = 1
 The Task 1 has sent the value 1
count = 2
 The Task 2 has sent the value 4
count = 3
 The Task 3 has sent the value 9
count = 4
 The Task 1 has sent the value 4
count = 5
 The Task 2 has sent the value 10
count = 6
 The Task 3 has sent the value 18
count = 7
 The Task 1 has sent the value 7
count = 8
 The Task 2 has sent the value 16
count = 9
 The Task 3 has sent the value 27
count = 10
 The Task 1 has sent the value 10
count = 11
 The Task 2 has sent the value 22
count = 12
 The Task 3 has sent the value 36
count = 13
 The Task 1 has sent the value 13
count = 14
 The Task 2 has sent the value 28
count = 15
 The Task 3 has sent the value 45
count = 16
 The Task 1 has sent the value 16
count = 17
```

Printf Output Console

Q2.

Code Implementation:

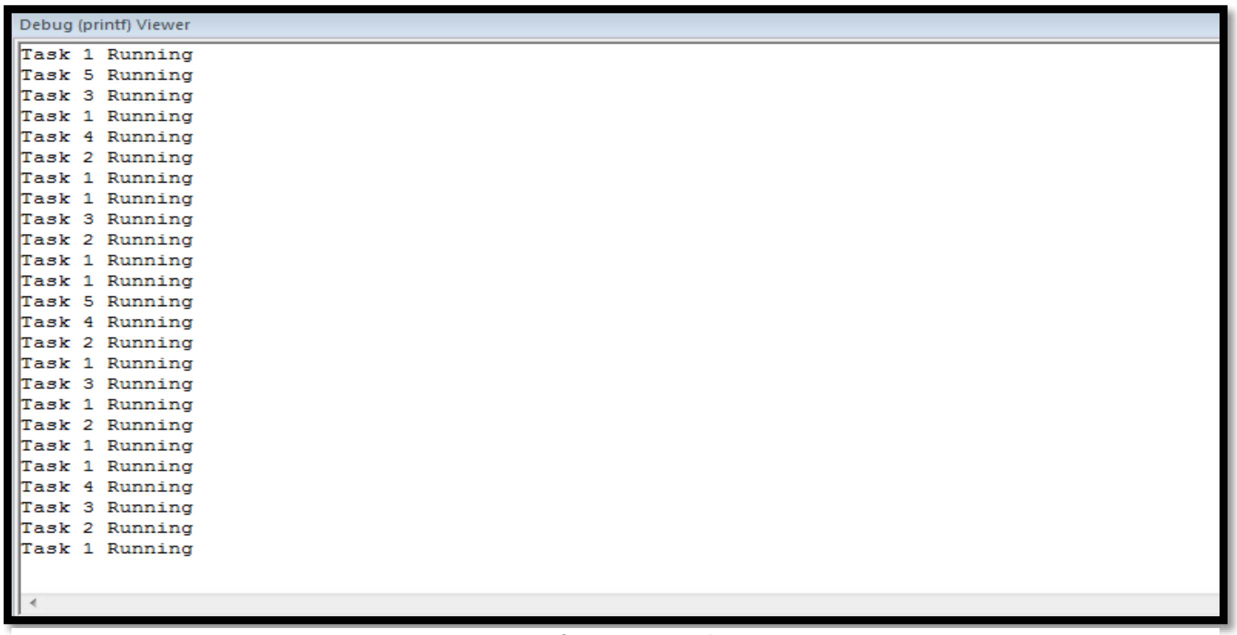
- **Created 4 tasks t1,t2,t3,t4,t5 with priorities 1,2,3,4,5 respectively.**
- **Task Id's are given by the settasknumber() function, The task ids of t1,t2,t3,t4,t5 are 1,2,3,4,5 respectively.**
- **Whenever the task gets scheduled, it will get its task id by using gettasknumber () function and it will print its ID and go to waiting state.**
- **The required delay of 1 second is obtained by dividing the number of ticks with the portTICK_RATE_MS which is a prebuild macro.**



Logic Analyser Output

Explanation:

- As we can observe from the logical analyser output, that task 1 is occurring for every frequently than task 2 which is very frequent when compared to other tasks and soon.
- Whenever the task waiting delay is over it wakes up and see whether any high priority task is executing or not, if executing then it will go to sleep again.
- After the execution of every task, that respective task will enter the waiting state until its tick delay.
- After the multiples of lcm(1,2,3,4,5) all the tasks will get wakeup ,but as the priority of t5 is more ,so t5 will executed forcing the other tasks to go to waiting state until their delay ticks.


Debug (printf) Viewer

```
Task 1 Running
Task 5 Running
Task 3 Running
Task 1 Running
Task 4 Running
Task 2 Running
Task 1 Running
Task 1 Running
Task 3 Running
Task 2 Running
Task 1 Running
Task 1 Running
Task 5 Running
Task 4 Running
Task 2 Running
Task 1 Running
Task 3 Running
Task 1 Running
Task 2 Running
Task 1 Running
Task 1 Running
Task 4 Running
Task 3 Running
Task 2 Running
Task 1 Running
```

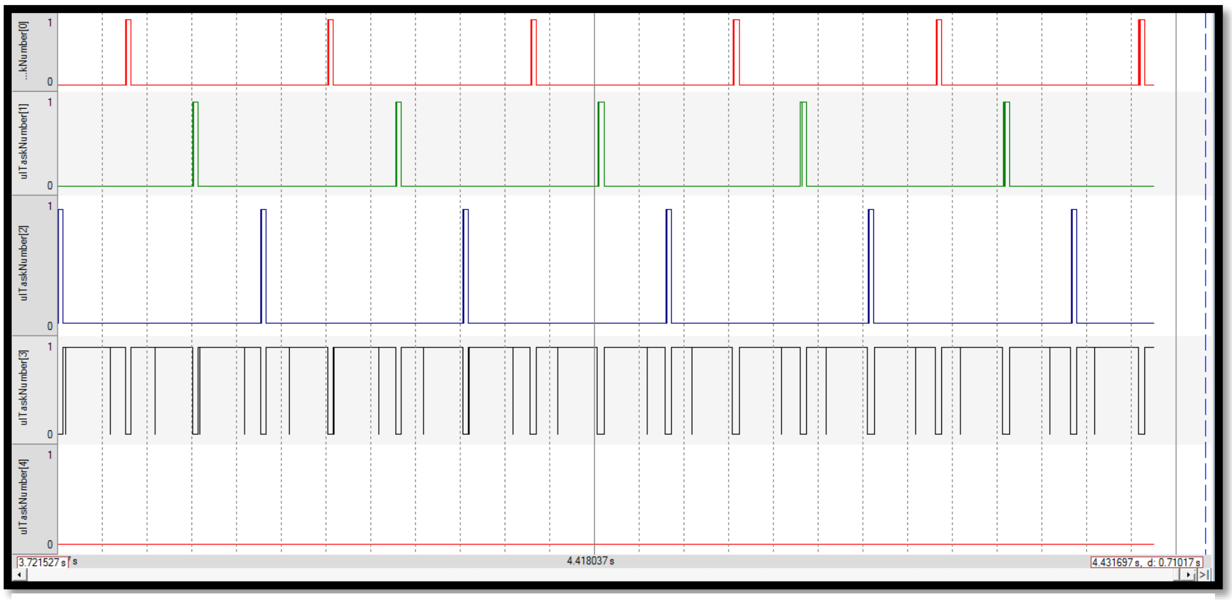
Printf Output Console

Q3.

Code Implementation:

- A function is made which has to be given the Queue Handler as an input, and the function will return all the information about the Queue, namely length, size of item in Queue etc.
- The xQueueHandle is of pointer to the structure xQueue defined in Queue.c. It has several data members which carry relevant information regarding queue like list of waiting tasks and also number of tasks waiting. To avoid direct user access of kernel data structures the xQueueHandle pointer is type casted to void pointer in queue.h. So in order to get that features we need to give our own function.

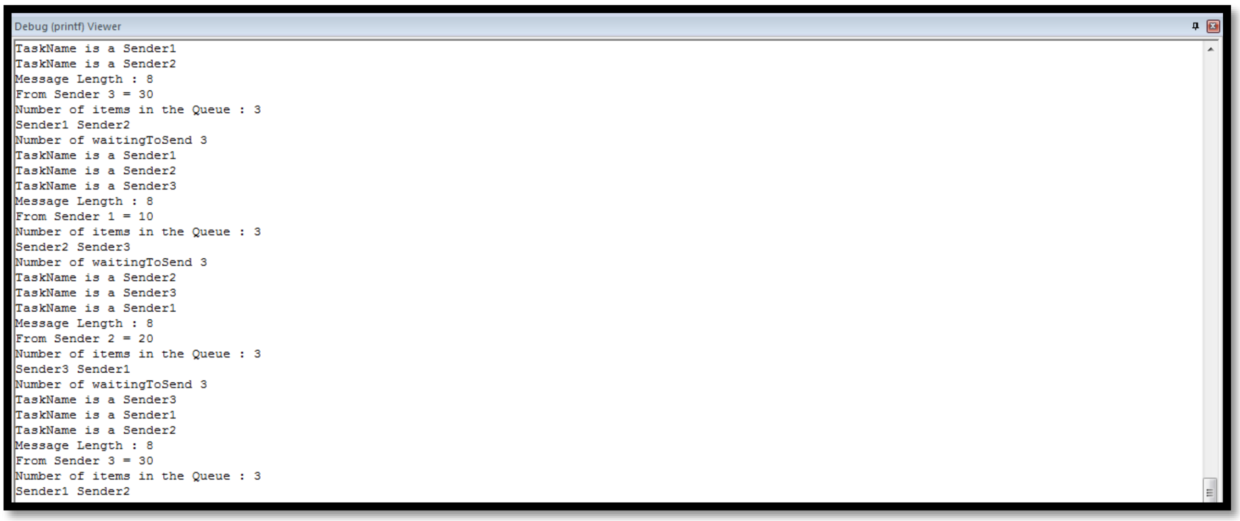
- As for security purposes it is defined as void type in queue.h and hence to call the structure of QueueDefinition, we need to pass the queue pointer to queue.c, access the structure of QueueDefinition and get the pointer to structure type“xList”.
- “xList” has the information about how many tasks are waiting to send data in Queue which is given by “uxNumberOfItems”, pointer pxIndex which is used to move through the list and “xListEnd”, which again is a structure type of “xMiniListItems”.
- xMiniListItems has three parameters namely xItemValue, pointer to the next item and pointer to the previous item. Also the pointers are of type struct “xList_Item” which has a parameter pointer “pvOwner”. This pvOwner basically points to the Task Control Block which contains all the details of the tasks which are waiting to send data I the queue. Basically for this pvOwner pointer, we need to go through these many pointing structure values.
- To find out the task name I have to get to xListItem which is the double circular linked list and contains the TCB (task control block) which cannot be accessed from the application so I added a function in the task.c (kernel) file to read all the name of TCB which are blocked under the sender.
- Now since we have the pointer pointing to Task Control Block, we can now access the Name of the task which is waiting to send data in the Queue.
- The other bits like No. of messages in the queue, queue length, item size are all stored in the QueueDefinition Structure which can be directly handled by xQueueHandle and I have written separate functions for that, and simply by calling that functions we get the output.



Logic Analyser Output

Explanation:

- **There are three tasks, 2 sender tasks and 1 receiver task. Sender task has higher priority and hence it will run first.**
- **Once sender task is in blocked state, the receiver task would run, which will call my defined function and then print the corresponding output data.**



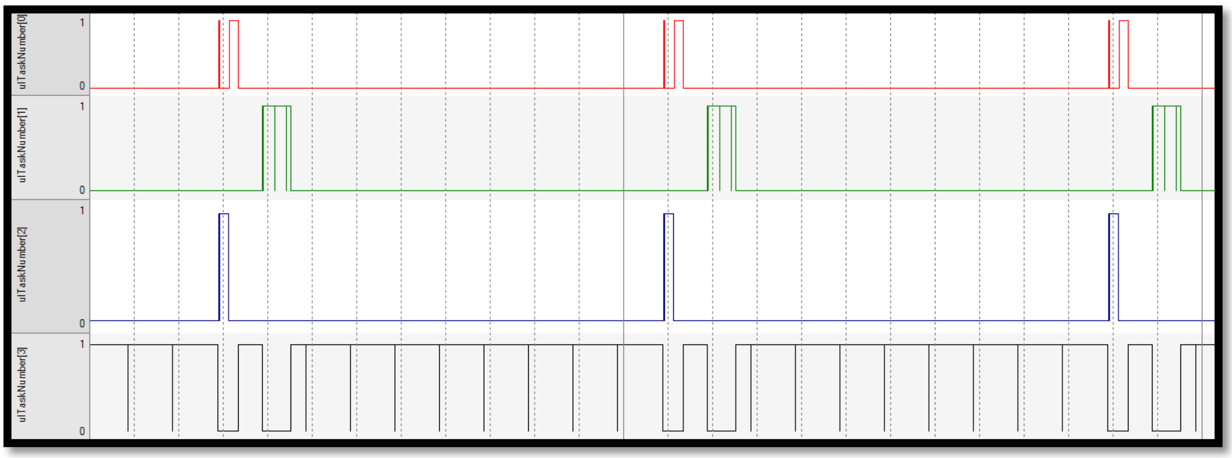
```
TaskName is a Sender1
TaskName is a Sender2
Message Length : 8
From Sender 3 = 30
Number of items in the Queue : 3
Sender1 Sender2
Number of waitingToSend 3
TaskName is a Sender1
TaskName is a Sender2
TaskName is a Sender3
Message Length : 8
From Sender 1 = 10
Number of items in the Queue : 3
Sender2 Sender3
Number of waitingToSend 3
TaskName is a Sender2
TaskName is a Sender3
TaskName is a Sender1
Message Length : 8
From Sender 2 = 20
Number of items in the Queue : 3
Sender3 Sender1
Number of waitingToSend 3
TaskName is a Sender3
TaskName is a Sender1
TaskName is a Sender2
Message Length : 8
From Sender 3 = 30
Number of items in the Queue : 3
Sender1 Sender2
```

Printf () Output Console

Q4.

Code Implementation:

- **Created tasks t1, t2, t3 where t1 and t3 are sending data to queue and t2 is receiving from queue.**
- **Set the Task ID for all the 3 tasks t1, t2, and t3 as 1, 2, and 3.**
- **Create a single Queue q1.**
- **Wrote a Wrapper Function xMyQueueSend () which will work on the top of the inbuilt xQueueSend function which can send the bundle of data that consists of both the task id and the task message.**
- **Similarly wrote a Wrapper Function xMyRecieve () which will work on the top of the inbuilt xQueueReceive () function and can receive a bundle of data that consists of both the task id and the message sent by the task.**
- **The xMyQueueSend () function will take the structure which has the task id and the values in a bundle and send that bundle to that queue q1.**
- **Similarly xQueueReceive() function will take the structure size data from the queue q1 and display the task id and task value separtely.So this function has the capability to know which sender has sent which data.**

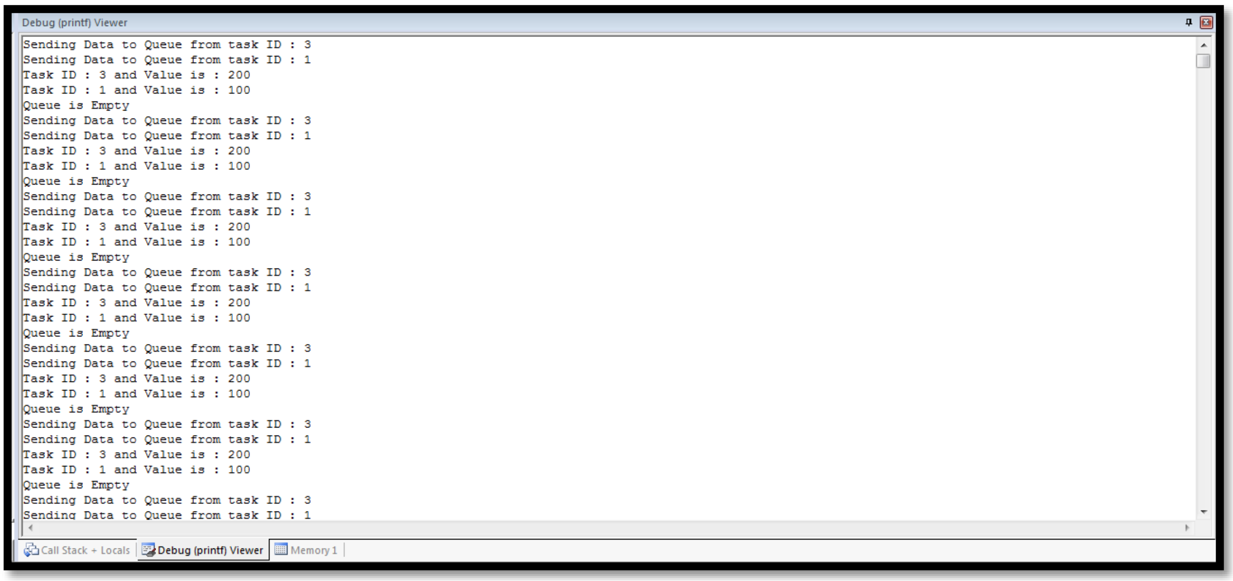


Logic Analyser Output

Explanation:

In the Logical Analyser output

- **Task1 is executed first, and will send the data i.e. task1 id as well as the task message to the queue q1 and enter into the wait sate for 10 ticks. Even task 1 or task3 can be executed at this moment because all were having the same priority.**
- **Then Task3 is executed and it will send its data as well as its id to the queue q1.**
- **When the task2 is executes it will receive as many number of structure sizes from the queue and print their respective task id as well as task values.**
- **Whenever there is no task to execute then idle task will be executes.**



```
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
Task ID : 3 and Value is : 200
Task ID : 1 and Value is : 100
Queue is Empty
Sending Data to Queue from task ID : 3
Sending Data to Queue from task ID : 1
```

Printf () Output Console

Q5.

Implementation of semaphore and Mutex in FreeRtos:

- **Binary semaphores and mutexes are very similar but have some subtle differences: Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.**
- **Each binary semaphore require a small amount of RAM that is used to hold the semaphore's state. If a binary semaphore is created using xSemaphoreCreateBinary () then this RAM is automatically allocated from the FreeRTOS heap. If a binary semaphore is created usingxSemaphoreCreateBinaryStatic () then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time. See the Static Vs Dynamic allocation page for more information.**
- **The semaphore is created in the 'empty' state, meaning the semaphore must first be given using thexSemaphoreGive () API function before it can subsequently be taken (obtained) using thexSemaphoreTake () function.**

In FreeRTOS Binary Semaphore & Mutex is implemented using same data structure which has been used to implement different type of Queues.

XQueueHandle xQueueCreateMutex (unsigned char ucQueueType)

/*Mutex can be created using this API*/

The implementation of the above function is done using a tail pointer (*pcTail) to point xMutexHolder and a head pointer (*pcHead) which will point to currently Mutex is acquired or not.If it is null then that is free. It also has the arguments like uxItemSize which is assigned Zero and uxLenghth which is assigned to 1 because we want to use Queue as a Mutex and it also uses two lists are also created which will keep track of other tasks which already tried to acquire the Mutex in the Waiting list.

- **This can be used by Calling XsemaphoreCreateMutex (), we cannot use it directly as defined in the semphr.h header file.**

Including Priority Inversion:

Including a priority inversion mechanism is straight forward. Basically the semaphore requires an extra member that contains the priority of the task that currently holds the semaphore, as follows:

- **TaskA is priority 1, TaskB priority 2.**
- **TaskA taskes the semaphore, its priority is stored in the semaphore.**
- **TaskB attempts to get the semaphore. It has a higher priority, so the priority of TaskA is raised to that of TaskB.**
- **TaskA releases the semaphore, and as it does so has its priority reset back to that stored in the semaphore.**

Including Priority Ceiling in FreeRtos:

The task control block already has two priority members uxPriority and uxBasePriority that we can reuse for the priority ceiling implementation. We would have to update the queue structure used by mutexes (defined in queue.c) to include a value for the ceiling priority. The ceiling protocol should be simpler to implement than the existing inheritance protocol.