AGH	*Akademia Górniczo-Hutnicza w Krakowie*		
Faculty: ESA	Academic Year 2019/2020	Year of study	Fields of science:
Student name:	**Ioannis Manousaridis**		
Course:	**Real Time Operating Systems in Control Applications**		
Exercise nr:	**08**		
Exercise subject:	**FreeRTOS – communication with queues**		
Lecturer: dr inż. Grzegorz Wróbel			
Date: 00.00.2020			Grade:

1 Exercise purpose

The purpose of the exercise is to get familiar with the mechanisms of communication between tasks using queues and the inter-task communication. The aim is to create tasks that the will receive and send data through queues.

2 Assumptions / Theory

The FreeRTOS provides the possibility for communication between tasks either with queues or semaphores. In this lab, the queues are used.

A queue is a first-in first-out (FIFO) abstract data type that is heavily used in computing. Uses for queues involve any case in which the user wants things to happen in the order that they were called, but where the computer can't keep up to speed.

Tasks can send data to a queue and receive from it.

Figure 1: Task A and B communicate through a queue. Task A is sending data and task B is receiving. The next element that Task B will receive is the letter ‘A’. The queue has three empty slots at this point.

3 Description of implementation

The queue.h library is used for dealing with queus in FreeRTOS. The base of the code can be seen in the figure 2. There is a task for sending data to three different queues and there is another task for receiving the data from the different queues and printing the received data.

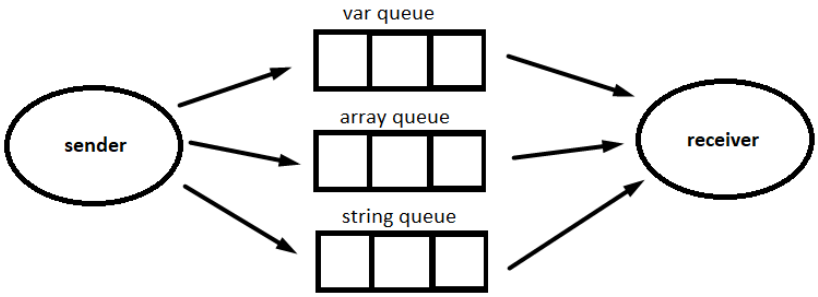

Figure 2: The sener task send data to the three queues and receiver takes the data from each queue.

The main function creates the three queues with different size in each cell. The var queue in each cell has space for an integer, the array queue in each cell has a space for an integer array of length 5 and the string queue has a space of 10 characters. Each queue has 3 cells. Afterwards, it creates the two tasks and the scheduler start the communication between the sender and the receiver. In case of insufficient memory of the heap the scheduler will stop, as mentioned in the lab 7 and an appropriate message will be printed in the screen of the user.

The sender task sends an intiger which increases by one every time, an integer arrray which gets random numbers from 0 to 9 and a string with random capital characters. First send the integer to var queue, then the array in the array queue and then the string to the string queue. If one of the queues are full, there is a specific amount of time that it will wait to send the data before proceeding to next queue for sending data. If it fails to send data to one queue, a message will be printed mentioned in which queue a failure occured. Finally, when the sender sends the data to all the queues it will wait for 2000 clock ticks before sending new data to the queues.

In the same concept, the receiver reads data from the queues in the same order, first from var queue, then from the array queue and finally from the string queue. If it receives data from the queue it will the received data, otherwise a message will be printed that will mention from which queue it did not receive data. Also, the receiver waits for receiving data from each queue for a different amount of time.

4 Conclusions

In general the following approaches were tested:

a) The base code which is included in the report.
b) Creating two tasks that both receive and send data. The task 1 was sending integers and was receiving arrays and vica versa for the task 2. Two queues were used in this approach.
c) The queue operation if the receive is performed by a non-blocking function (xTicksToWait = 0). The results here varied depending in the vTaskDelay(). Both the

receiver have a vTaskDelay() function. If the vTaskDelay in receiver is deleted and the vTaskDelay in the sender is set on 500, the receiver will receive data only every 500 clock ticks as it can be seen in the figure 3. This is normal, because the receiver in every loop will check the queues and will receive the data from them if there is any. Otherwise, as the xTicksToWait is set to 0, it will not wait and it will print that it failed to receive data. In this example, a lot of fails in the receiving of the data were printed as the sender is much slower than the receiver. If we delete both of the vTaskDelay() function the receiver will again not receive a lot of data, but when it will receive it will receive a lot of data in the row. This takes place because the first time that the receiver is given data, it will waste some time for reading the data and printing on the screen. This delay will give the opportunity to the sender to be faster than the receiver for a small period of time as it can be seen in figure 4, in which the receiver got 4 times in a row data from the queue. The number 4 of course is not random, but it has to do with the number of cells in the queues which is 3 and the fact that when the queue is filled the sender is going to wait for a specific amount of time in this queue and will try to send the data again. By the time the sender sends data once again, the receiver will have already empty the queue and read a lot of times the queue again. That's why after four times, the user will get multiple errors. Finally, if we add a vTaskDelay() function in the receiver only, there will be no problem and it will receive all the data from the sender.

d) The waiting time for free space in the queue is longer than the set blocking time (xTicksToWait) of the xQueueSend function. In that case, if the number are not an extreme occasion, the receiver always will receive the data and there will be no failure when sending data. This is due to the fact that if a queue if full, the sender will wait more time than the receiver and eventually the sender will send the data. This is very useful because in this case there is no loss of data.

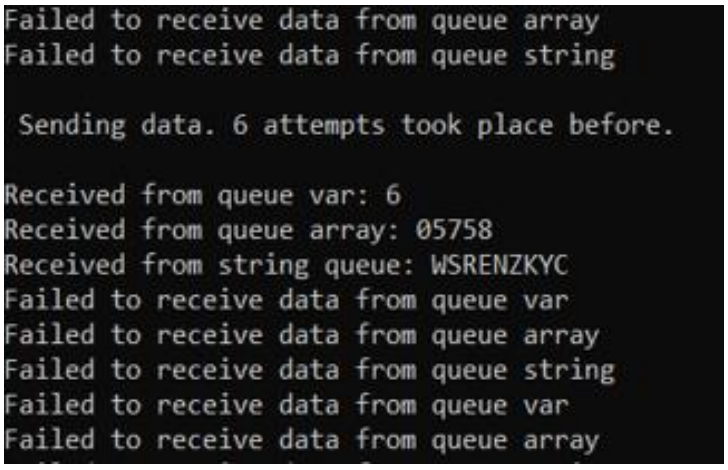

Figure 3: The results when only the sender has a delay function of of 500 clock ticks.

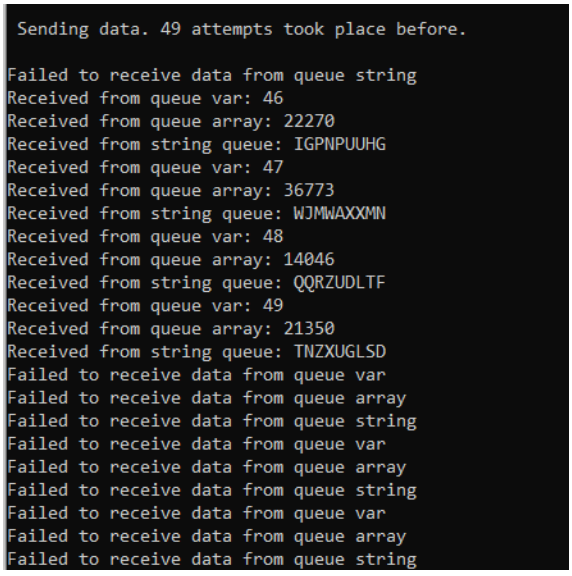

Figure 4: The results when both the receiver and the sender did not have any delay function.