QUEUE MANAGEMENT

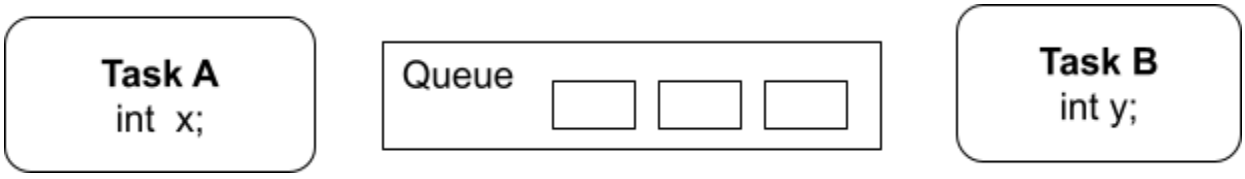
INDEX

1. Characteristics of a Queue

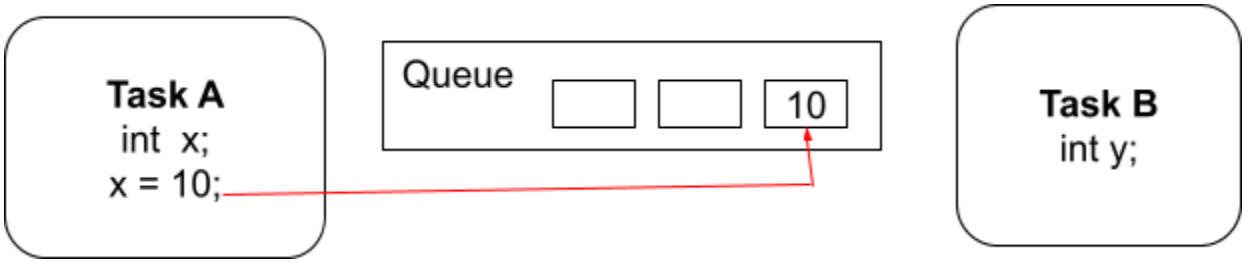
- Queue is an abstract data Structure and Queue is opened at both the ends.
- Queue provides a **task to task , task to interrupt, and interrupt to task communication** mechanism at runtime.
- Except **global variables** no data can be transferred from one task to another task in **Task Management.**
- **Memory allocation** is taken care of by the **kernel** while creating a queue.
- 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.
- Queues are normally used as **First In First Out(FIFO)** , where data is written to the end of the queue and removed from the front of the queue.
- It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue.

1.1. Example Sequence of writes to, and reads from a queue

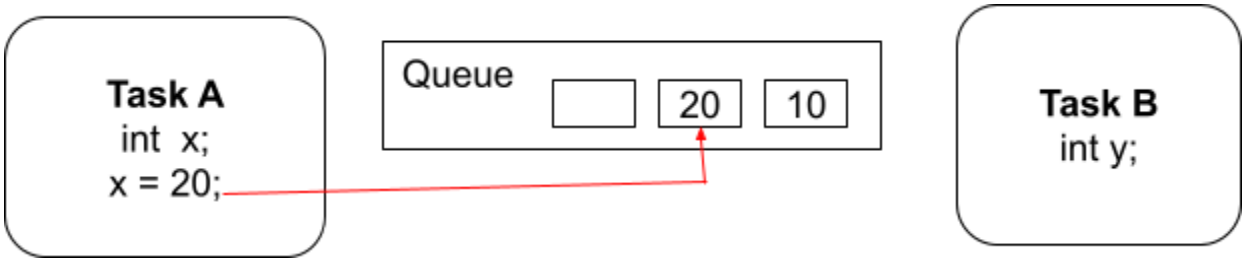
- A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 3 integers. When the queue is created it does not contain any values so is empty.



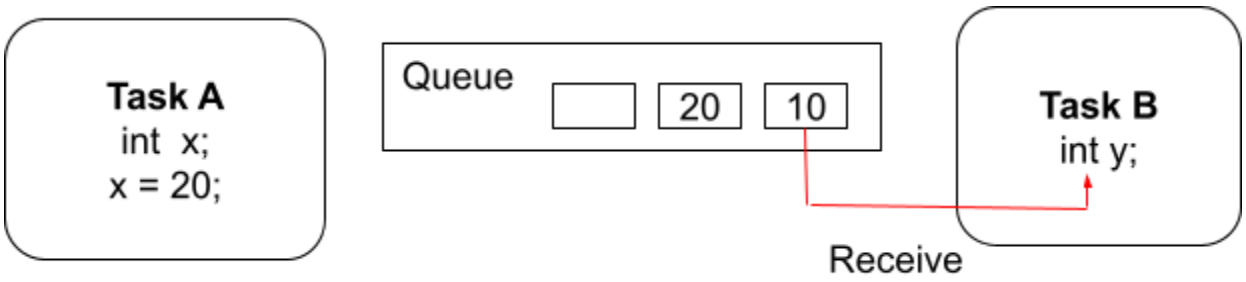
- Task A sends the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



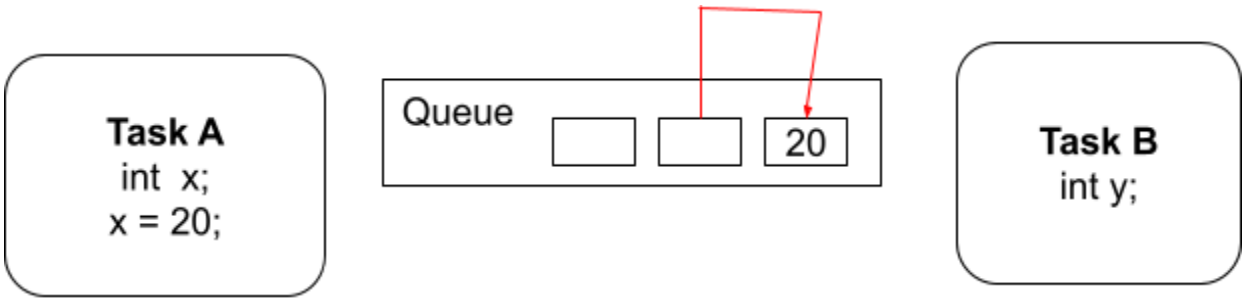
- Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue.The queue has 1 empty space remaining.



- Task B receives from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue.



- Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has 2 empty spaces remaining.



- There are two ways in which queue behaviour can be implemented:
    - **Queue by Copy** : It means the data sent to the queue is copied byte for byte into the queue.
        - By default the FreeRTOS method implements Queue by copy.
        - The RTOS takes complete responsibility for allocating the memory used to store data.
        - The sending task and the receiving task are completely decoupled
            - **Sending Task**: This task can place items into the queue without needing to wait for the receiving task to be ready to process them. If the queue is full, the sending task can choose to wait (block)

until there is space, or return immediately based on the configuration.
- **Receiving Task**: This task can retrieve items from the queue whenever it is ready to process them. If the queue is empty, the receiving task can choose to wait (block) until an item becomes available, or return immediately based on the configuration.

- **Queue by reference** : It means the queue only holds pointers to the data sent to the queue, not the data itself.

1.2. Access by Multiple Tasks

- Queues are objects in their own right and can be accessed by any task or ISR that knows of their existence. Any number of tasks can be written to the same queue, and any number of tasks can be read from the same queue.
- The FreeRTOS queue API ensures that the operations on the queue are thread-safe. Multiple tasks can send to and receive from the same queue without additional synchronisation.

2. Using a Queue

- A Queue must be explicitly created before it can be used.
- Queues are referenced by handles, which are variables of type QueueHandle_t.
- FreeRTOS allocates RAM from the FreeRTOS heap when a queue is created. The RAM is used to hold both the queue data structures and the items that are contained in the queue.

2.1. Create a Queue

- **QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);**
  - **Parameters:**
    - **uxQueueLength** → The maximum number of items that the queue being created can hold at one time.
    - **uxItemSize** → The size in bytes of each data item that can be stored in the queue.

  - **Return :**
    - If NULL is returned , then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

- A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

- After a queue has been created the xQueueReset() API function can be used to return the queue to its original empty state.

2.2. xQueueSend()

- #include “FreeRTOS.h”
  #include “queue.h”
- **BaseType_t xQueueSend(QueueHandle_t xQueue,**
  const void * pvItemToQueue,
  TickType_t xTicksToWait);

2.2.1. xQueueSendToBack()

- **BaseType_t xQueueSendToBack(QueueHandle_t xQueue,**
  const void * pvItemToQueue,
  TickType_t xTicksToWait);

2.2.2. xQueueSendToFront()

- **BaseType_t xQueueSendToFront(QueueHandle_t xQueue,**
  const void * pvItemToQueue,
  TickType_t xTicksToWait);

- xQueueSend() and xQueueSendToBack() perform the same operation. Both send data to the back of a queue.

- **Parameters:**
  - **xQueue** → The handle of the queue to which the data is being sent.
  - **pvItemToQueue** → A pointer to the data to be copied into the queue.
    - The size of each item the queue can hold is set when queue is created, and that many bytes will be copied from pvItemToQueue storage area.
  - **xTicksToWait** → The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.
    - Return immediately **If xTicksToWait is 0** and the queue is already full
    - Set **xTicksToWait to portMAX_DELAY** will cause the task to wait indefinitely .The task waits until the queue is dequeued (the last element of the queue will be empty). An external de-queue block will be necessary to run the task if the queue is already full due to max_DELAY.

        - Set **xTicksToWait to pdMS_TO_TICKS()** wait for finite ticks time and if the queue is full wait some time and after it returns.

- **Return :**
    - **pdPASS** → Returned if data was successfully sent to the queue.
        - If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for space to become available in the queue before the function returned, but data was successfully written to the queue before the block time expired.
    - **errQUEUE_FULL** → Returned if data could not be written to the queue because the queue was already full.
        - If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to make room in the queue, but the specified block time expired before that happened.

2.3. xQueueReceive()

- **BaseType_t xQueueReceive(QueueHandle_t xQueue,**
  void *pvBuffer,
  TickType_t xTicksToWait);

- Read an item from a queue.
- **Parameters :**
    - xQueue → The handle of the queue from which the data is being received.
    - pvBuffer → A pointer to the memory into which the received data will be copied .
        - The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
    - xTicksToWait → The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.
        - If **xTicksToWait is zero**, then xQueueReceive() will return immediately if the queue is already empty.
        - Set **xTicksToWait to portMAX_DELAY** will cause the task to wait indefinitely (without timing out) provided **INCLUDE_vTaskSuspend** is set to 1 in **FreeRTOSConfig.h**
        - Set **xTicksToWait to pdMS_TO_TICKS()** wait for finite ticks time and if the queue is already empty wait some time and after it returns.
- **Return :**
    - **pdPASS** → Returned if data was successfully Read from the queue.
        - If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for

data to become available in the queue , but data was successfully read from the queue before the block time expired.

- **errQUEUE_EMPTY** → Returned if data could not be read from the queue because the queue was already empty.
  - If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the specified block time expired before that happened.

2.4. vQueueDelete()

- **void vQueueDelete(TaskHandle_t pxQueueToDelete);**
- **Parameters** :
  - **pxQueueToDelete** →The handle of the queue being deleted. Semaphore handles can also be used
- **Return** : None
- **Note :** Queues are used to pass data between tasks and between tasks and interrupts. Tasks can opt to block on a queue/semaphore (with an optional timeout) if they attempt to send data to the queue/semaphore and the queue/semaphore is already full, or they attempt to receive data from a queue/semaphore and the queue/semaphore is already empty. A queue/semaphore must not be deleted if there are any tasks currently blocked on it

2.5. xQueuePeek()

- **BaseType_t xQueuePeek(QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait);**
- Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.
- **Parameters :**
  - xQueue → The handle of the queue from which data is to be read.
  - pvBuffer → A pointer to the memory into which the data read from the queue will be copied. The length of the buffer must be at least equal to the queue item size.
  - xTicksToWait :
    - If xTicksToWait is zero, the xQueuePeek() will return immediately if the queue is already empty.
    - pdMS_TO_TICKS()
    - portMAX_DELAY
- **Return** :
  - pdPASS → Returned if data was successfully read from the queue.
  - errQUEUE_EMPTY → Returned if data cannot be read from the queue because the queue is already empty.

2.6. xQueueOverwrite()

- **BaseType_t xQueueOverwrite(QueueHandle_t xQueue, const void *pvItemToQueue);**
- A version of xQueueSendToBack() that will write to the queue even if the queue is full, overwriting data that is already held in the queue. xQueueOverwrite() is intended for use with queues that have a length of one, meaning the queue is either empty or full.
- **Parameters :**
    - **xQueue** → The handle of the queue to which the data is to be sent.
    - **pvItemToQueue** → A pointer to the data to be copied into the queue.

- **Return** : xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

2.7. xQueueReset()

- **BaseType_t xQueueReset(QueueHandle_t xQueue);**
- Resets a queue to its original empty state. Any data contained in the queue at the time it is reset is discarded.
- **Parameters** :
    - xQueue → The handle of the queue that is being reset.
- **Return** : pdPASS

2.8. xQueueCreateSet()

- **QueueSetHandle_t xQueueCreateSet(const UBaseType_t uxEventQueueLength);**
- Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously.
- A queue set must be explicitly created using a call to **xQueueCreateSet()** before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to **xQueueAddToSet(). xQueueSelectFromSet()** is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.
- **Parameters** :
    - **uxEventQueueLength** → Queue sets store events that occur on the queues and semaphores contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once. For example :

    - If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18.
    - If a queue set is to hold three binary semaphores then uxEventQueueLength should be set to (1 + 1 + 1), or 3.
    - If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3 then uxEventQueueLength should be set to (5+3) = 8.
- **Return** :
    - **NULL** → The queue set could not be created.
    - **Any other value** →The queue set was created successfully. The returned value is a handle by which the created queue set can be referenced.
- **Note** :
    - Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.
    - An additional 4 bytes of RAM are required for each space in every queue added to a queue set. Therefore a counting semaphore that has a high maximum count value should not be added to a queue set.

2.9. xQueueAddToSet()

- **BaseType_t xQueueAddToSet(QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet);**
- Adds a queue or semaphore to a queue set that was previously created by a call to xQueueCreateSet(). A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.
- **Parameters** :
    - **xQueueOrSemaphore** → The handle of the queue or semaphore being added to the queue set (cast to an QueueSetMemberHandle_t type).
    - **xQueueSet** → The handle of the queue set to which the queue or semaphore is being added.
- **Return** :
    - **pdPASS** →The queue or semaphore was successfully added to the queue set.
    - **pdFAIL** → The queue or semaphore could not be added to the queue set because it is already a member of a different set.

2.10.xQueueRemoveFromSet()

- **BaseType_t xQueueRemoveFromSet(QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t xQueueSet);**

- Remove a queue or semaphore from a queue set. A queue or semaphore can only be removed from a queue set if the queue or semaphore is empty
- configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueRemoveFromSet() API function to be available
- **Parameters :**
    - xQueueOrSemaphore → The handle of the queue or semaphore being removed from the queue set (cast to an QueueSetMemberHandle_t type).
    - xQueueSet → The handle of the queue set in which the queue or semaphore is included
- **Return :**
    - **pdPASS** → The queue or semaphore was successfully removed from the queue set
    - **pdFAIL** → The queue or semaphore was not removed from the queue set because either the queue or semaphore was not in the queue set, or the queue or semaphore was not empty.

3. Examples

3.1. Creating a Queue

- Queues are used for communication between tasks and can store a fixed number of items of a specified size.
- To create a queue, send data to the queue from one task, and receive data from the queue in another task.

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdarg.h>

void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}

QueueHandle_t xQueue;
```

```
void vSenderTask(void *pvvar) {
   int ivalueToSend;

   for(;;) {
      ivalueToSend = 1;
      if (xQueueSend(xQueue, &ivalueToSend, pdMS_TO_TICKS(100)) == pdPASS)
      {
         vPrintString("Sent : %d\n", ivalueToSend);
      }
      else
      {
         vPrintString("Queue is full. Failed to send : %d\n", ivalueToSend);
      }
      vTaskDelay(pdMS_TO_TICKS(500));
   }
}

void vReceiverTask(void *pvvar) {
   int iReceivedValue;

   for(;;) {

      if (xQueueReceive(xQueue, &iReceivedValue, pdMS_TO_TICKS(100)) == pdPASS)
      {
         vPrintString("Received : %d\n", iReceivedValue);
      }
      else
      {
         vPrintString("Queue is empty. No data received \n");
      }
      vTaskDelay(pdMS_TO_TICKS(500));
   }
}

int main(void) {
   xQueue = xQueueCreate(5, sizeof(int));
   if (xQueue != NULL) {
      xTaskCreate(vSenderTask, "sender",130,NULL,1,NULL);
      xTaskCreate(vReceiverTask, "Receiver",130,NULL,1,NULL);
      vTaskStartScheduler();
   }
   else {
```

```
        printf("Queue creation failed\n");
    }
    for(;;) {
    }
}
```

OUTPUT :

```
Sent : 1
Received : 1
Sent : 1
Received : 1
Sent : 1
Received : 1
```

3.2. Deleting a Queue

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdarg.h>

void vPrintString(const char *format, ...) {
   va_list args;
   va_start(args, format);
   vprintf(format, args);
   va_end(args);
   fflush(stdout);
}

QueueHandle_t xQueue;

void vSenderTask(void *pvvar) {
   int ivalueToSend;

   for(;;) {
      ivalueToSend = 1;
      if (xQueueSend(xQueue, &ivalueToSend, pdMS_TO_TICKS(100)) == pdPASS)
      {
         vPrintString("Sent : %d\n", ivalueToSend);
      }
```

```
        else
        {
            vPrintString("Queue is full. Failed to send : %d\n", ivalueToSend);
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vReceiverTask(void *pvvar) {
    int iReceivedValue;

    for(;;) {

        if (xQueueReceive(xQueue, &iReceivedValue, pdMS_TO_TICKS(100)) == pdPASS)
        {
            vPrintString("Received : %d\n", iReceivedValue);
        }
        else
        {
            vPrintString("Queue is empty. No data received \n");
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vDeleteTask(void *pvvar) {
    for(;;) {
        vPrintString("After Deleting the queue \n");
        vQueueDelete(xQueue);
    }
}

int main(void) {
    xQueue = xQueueCreate(5, sizeof(int));
    if (xQueue != NULL) {
        vPrintString("Queue can be created\n");
        xTaskCreate(vSenderTask, "sender",130,NULL,1,NULL);
        xTaskCreate(vReceiverTask, "Receiver",130,NULL,1,NULL);
        xTaskCreate(vDeleteTask, "Deleting",130,NULL,1,NULL);
        vTaskStartScheduler();

    }
    else {
        vPrintString("The queue can not be created\n");
```

```
    }
    for(;;) {
    }
}

/*
Queue can be created
After Deleting the queue
*/
```

OUTPUT:

```
/*
Queue can be created
Sent : 1ReceivedAfter D
 etineleting the queue
After Deleting the queue
*/
```

3.3. Number of items in a queue by using uxQueueMessagesWaiting

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdarg.h>

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

QueueHandle_t xQueue;

void vSenderTask(void *pvvar)  {
```

```
    UBaseType_t uxNumberOfItems;
    int ivalueToSend;

    for(;;) {
        for (int i = 0 ; i < 5; i++) {
            ivalueToSend = i;
            if (xQueueSend(xQueue, &ivalueToSend, pdMS_TO_TICKS(100)) == pdPASS)
            {

               vPrintString("Sent : %d\n", ivalueToSend);

            }
            else
            {


                vPrintString("Queue is full. Failed to send : %d\n", ivalueToSend);

            }

            uxNumberOfItems = uxQueueMessagesWaiting(xQueue);

            vPrintString("number of items in a queue %d\n",uxNumberOfItems);

            vTaskDelay(pdMS_TO_TICKS(500));
        }

    }
}



int main(void) {
    xQueue = xQueueCreate(5, sizeof(int));
    if (xQueue != NULL) {
        xTaskCreate(vSenderTask, "sender",130,NULL,1,NULL);
        vTaskStartScheduler();
    }
    else {
        printf("Queue creation failed\n");
    }
    for(;;) {
    }
}
```

OUTPUT :

```
Sent : 0
number of items in a queue 1
Sent : 1
number of items in a queue 2
Sent : 2
number of items in a queue 3
Sent : 3
number of items in a queue 4
Sent : 4
number of items in a queue 5
Queue is full. Failed to send : 0
number of items in a queue 5
Queue is full. Failed to send : 1
number of items in a queue 5
Queue is full. Failed to send : 2
number of items in a queue 5
Queue is full. Failed to send : 3
number of items in a queue 5
Queue is full. Failed to send : 4
number of items in a queue 5
Queue is full. Failed to send : 0
number of items in a queue 5
```

3.4. QueuePeek

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdarg.h>

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}
```

```
QueueHandle_t xQueue;

void vSenderTask(void *pvvar)  {
   UBaseType_t uxNumberOfItems;
   int ivalueToSend;

   for(;;) {
      for (int i = 0 ; i < 5; i++) {
         ivalueToSend = i;
         if (xQueueSendToBack(xQueue, &ivalueToSend, pdMS_TO_TICKS(100)) == pdPASS)
         {

            vPrintString("Sent : %d\n", ivalueToSend);

         }
         else
         {


            vPrintString("Queue is full. Failed to send : %d\n", ivalueToSend);

         }

         uxNumberOfItems = uxQueueMessagesWaiting(xQueue);

         vPrintString("number of items in a queue %d\n",uxNumberOfItems);

         vTaskDelay(pdMS_TO_TICKS(500));
      }

   }
}

void vPeekTask (void *pvar) {
   int i;
   if (xQueue != NULL) {
      if ( xQueuePeek(xQueue, &i, 100) == pdPASS) {
         taskENTER_CRITICAL();
         vPrintString("Peek value : %d\n",i);
         taskEXIT_CRITICAL();
      }
   }
   else {
      vPrintString("the queue could not be created\n");
```

```
    }
    for(;;) {

    }
}

int main(void) {
    xQueue = xQueueCreate(5, sizeof(int));
    if (xQueue != NULL) {
        xTaskCreate(vSenderTask, "sender",130,NULL,1,NULL);
        xTaskCreate(vPeekTask, "peek",130,NULL,1,NULL);
        vTaskStartScheduler();
    }
    else {
        printf("Queue creation failed\n");
    }
    for(;;) {
    }
}
```

OUTPUT :

```
Sent : Peek value : 0
0unumber of items in a queue 1
Sent : 1
number of items in a queue 2
Sent : 2
number of items in a queue 3
Sent : 3
number of items in a queue 4
Sent : 4
number of items in a queue 5
Queue is full. Failed to send : 0
number of items in a queue 5
Queue is full. Failed to send : 1
number of items in a queue 5
Queue is full. Failed to send : 2
number of items in a queue 5
Queue is full. Failed to send : 3
number of items in a queue 5
Queue is full. Failed to send : 4
number of items in a queue 5
Queue is full. Failed to send : 0
number of items in a queue 5
```

```
Queue is full. Failed to send : 1
number of items in a queue 5
Queue is full. Failed to send : 2
number of items in a queue 5
Queue is full. Failed to send : 3
number of items in a queue 5
Queue is full. Failed to send : 4
number of items in a queue 5
```

3.5. xQueueReset

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include <stdio.h>
#include <stdarg.h>

// Mutex for synchronized printing
SemaphoreHandle_t xPrintMutex;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

QueueHandle_t xQueue;

void vApplicationIdleHook(void)
{
}

void vSenderTask(void *pvParameters) {
    int ivalueToSend = 0;
    UBaseType_t uxNumberOfItems;
```

```
    for (;;) {
        // Check the number of items in the queue
        uxNumberOfItems = uxQueueMessagesWaiting(xQueue);

        if (uxNumberOfItems == 3) {
            // Reset the queue when it is full with 3 items
            xQueueReset(xQueue);
            xSemaphoreTake(xPrintMutex, portMAX_DELAY);
            vPrintString("Queue reset\n");
            xSemaphoreGive(xPrintMutex);
        } else {
            // Attempt to send items to the queue
            if (xQueueSend(xQueue, &ivalueToSend, portMAX_DELAY) == pdPASS) {
                xSemaphoreTake(xPrintMutex, portMAX_DELAY);
                vPrintString("Sent: %d\n", ivalueToSend);
                xSemaphoreGive(xPrintMutex);

                ivalueToSend++;
            } else {
                vPrintString("Failed to send: %d\n", ivalueToSend);

            }

            uxNumberOfItems = uxQueueMessagesWaiting(xQueue);

            xSemaphoreTake(xPrintMutex, portMAX_DELAY);
            vPrintString("Number of items in the queue: %d\n", uxNumberOfItems);
            xSemaphoreGive(xPrintMutex);
        }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

void vReceiverTask(void *pvParameters) {
    int receivedValue;

    for (;;) {
        // Attempt to receive items from the queue
        if (xQueueReceive(xQueue, &receivedValue, portMAX_DELAY) == pdPASS) {
            xSemaphoreTake(xPrintMutex, portMAX_DELAY);
            vPrintString("Received: %d\n", receivedValue);
            xSemaphoreGive(xPrintMutex);
        } else {
```

```
            vPrintString("Failed to receive\n");

        }

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main(void) {
    xQueue = xQueueCreate(5, sizeof(int));
    xPrintMutex = xSemaphoreCreateMutex();

    if (xQueue != NULL && xPrintMutex != NULL) {
        xTaskCreate(vSenderTask, "Sender", 130, NULL, 1, NULL);
        xTaskCreate(vReceiverTask, "Receiver", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        vTaskStartScheduler();
    } else {
        vPrintString("Queue or Mutex creation failed\n");
    }

    for (;;) {
    }

}
```

OUTPUT :

```
Sent: 0
Number of items in the queue: 1
Received: 0
Sent: 1
Number of items in the queue: 1
Received: 1
Sent: 2
Number of items in the queue: 1
Sent: 3
Number of items in the queue: 2
Received: 2
Sent: 4
Number of items in the queue: 2
Sent: 5
Number of items in the queue: 3
```

```
Received: 3
Sent: 6
Number of items in the queue: 3
Queue reset
Sent: 7
Number of items in the queue: 1
Received: 7
Sent: 8
Number of items in the queue: 1
Received: 8
Sent: 9
Number of items in the queue: 1
Sent: 10
Number of items in the queue: 2
Received: 9
Sent: 11
Number of items in the queue: 2
Sent: 12
Number of items in the queue: 3
Received: 10
Sent: 13
Number of items in the queue: 3
Queue reset
Sent: 14
Number of items in the queue: 1
Received: 14
Sent: 15
Number of items in the queue: 1
Received: 15
Sent: 16
Number of items in the queue: 1
Sent: 17
Number of items in the queue: 2
Received: 16
Sent: 18
Number of items in the queue: 2
Sent: 19
Number of items in the queue: 3
Received: 17
Sent: 20
Number of items in the queue: 3
```

3.6 QueuePeek using both sender and receiver task

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include <stdio.h>
#include <stdarg.h>

SemaphoreHandle_t xPrintMutex;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void) { }

QueueHandle_t xQueue;

void vSenderTask(void *pvvar) {
    int ivalueToSend = 0;

    for (;;) {

        if (xQueueSendToBack(xQueue, &ivalueToSend, pdMS_TO_TICKS(100)) == pdPASS) {
            xSemaphoreTake(xPrintMutex, portMAX_DELAY);
            vPrintString("Sent: %d\n", ivalueToSend);
            xSemaphoreGive(xPrintMutex);
        } else {

            vPrintString("Queue is full. Failed to send: %d\n", ivalueToSend);

        }
         ivalueToSend ++ ;
        UBaseType_t uxNumberOfItems = uxQueueMessagesWaiting(xQueue);
        xSemaphoreTake(xPrintMutex, portMAX_DELAY);
        vPrintString("Number of items in the queue: %d\n", uxNumberOfItems);
        xSemaphoreGive(xPrintMutex);
```

```
            vTaskDelay(pdMS_TO_TICKS(200));
        }
    }


void vReceiverTask(void *pvvar) {
    int iReceivedValue;

    for (;;) {
        if (xQueueReceive(xQueue, &iReceivedValue, pdMS_TO_TICKS(100)) == pdPASS) {
            xSemaphoreTake(xPrintMutex, portMAX_DELAY);
            vPrintString("Received: %d\n", iReceivedValue);
            xSemaphoreGive(xPrintMutex);
        }

        vTaskDelay(pdMS_TO_TICKS(700));
    }
}

void vPeekTask(void *pvar) {
    int i;

    for (;;) {
        if (xQueue != NULL) {
            if (xQueuePeek(xQueue, &i, pdMS_TO_TICKS(100)) == pdPASS) {
                xSemaphoreTake(xPrintMutex, portMAX_DELAY);
                vPrintString("Peek value: %d\n", i);
                xSemaphoreGive(xPrintMutex);
            }
        } else {

            vPrintString("The queue could not be created\n");

        }

        vTaskDelay(pdMS_TO_TICKS(400)); // Add a delay to allow peeking periodically
    }
}

int main(void) {
    xQueue = xQueueCreate(5, sizeof(int));
    xPrintMutex = xSemaphoreCreateMutex();
```

```
    if (xQueue != NULL && xPrintMutex != NULL) {
        xTaskCreate(vSenderTask, "Sender", 130, NULL, 1, NULL);
        xTaskCreate(vReceiverTask, "Receiver", 130, NULL, 1, NULL);
        xTaskCreate(vPeekTask, "Peek", 130, NULL, 1, NULL);
        vTaskStartScheduler();
    } else {
        printf("Queue or Mutex creation failed\n");
    }

    for (;;) { }

}
```

Output:

```
Sent: 0
Number of items in the queue: 1
Received: 0
Sent: 1
Number of items in the queue: 1
Sent: 2
Number of items in the queue: 2
Peek value: 1
Sent: 3
Number of items in the queue: 3
Received: 1
Sent: 4
Number of items in the queue: 3
Peek value: 2
Sent: 5
Number of items in the queue: 4
Sent: 6
Number of items in the queue: 5
Peek value: 2
Received: 2
Sent: 7
Number of items in the queue: 5
Peek value: 3
Queue is full. Failed to send: 8
Number of items in the queue: 5
Queue is full. Failed to send: 9
Number of items in the queue: 5
Received: 3
```

```
Peek value: 4
Sent: 10
Number of items in the queue: 5
Peek value: 4
Queue is full. Failed to send: 11
Number of items in the queue: 5
Received: 4
Sent: 12
Number of items in the queue: 5
Peek value: 5
Queue is full. Failed to send: 13
Number of items in the queue: 5
Peek value: 5
Queue is full. Failed to send: 14
Number of items in the queue: 5
Received: 5
Sent: 15
Number of items in the queue: 5
Peek value: 6
Queue is full. Failed to send: 16
Number of items in the queue: 5
Peek value: 6
```