Like other modern RTOS Kernel, FreeRTOS Kernel also provides a process for **inter-task data communication**. These are known as **message Queues**. They are the underlying primitive used by all FreeRTOS communication and synchronization mechanisms. They are used to **send and receive messages** between tasks.

Queues Introduction

A **message queue** is a kind of FIFO buffer that holds fixed-size data items. Also, the number of items a queue can hold is also fixed, after its initialization. Usually, **tasks write** data to the end of the buffer and read from the front end of the buffer. But it is also possible to write at the front end. Multiple writers and readers can write and read from the buffer.

But only one writer/reader can access buffer at a time and Other, tasks remain block. Therefore, blocking is possible both on reads and writes to buffer.

Blocking on Queue Reads

Blocking on reads possible in following scenarios:

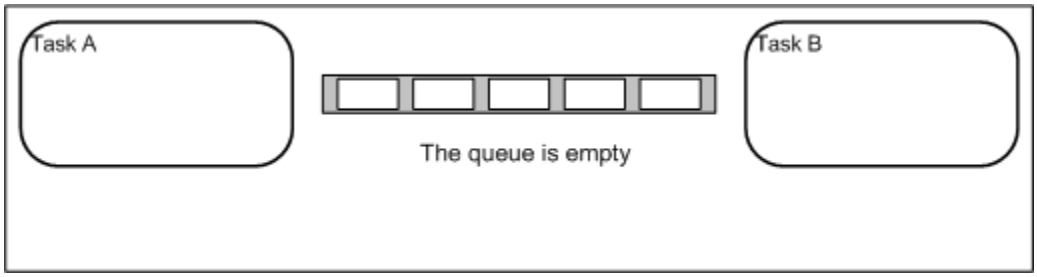
1. If **multiple tasks** are ready to receive data from the message queue, then the highest priority task gets to read data first and lowest priority one read data at the end. Meanwhile, other tasks remain block. We can also specify the maximum blocking time of a task while sending a read request. But different tasks can also have different blocking time.
2. The other possible case is when a queue is empty. In such a case, all read requests go to a blocking state. Once data become available from the **message queue** (when another task places data into the ready queue), all readers moved to the ready state according to their priority.

Blocking on Queue Writes

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

Queues can have multiple writers so it is possible that a full queue will have more than one task blocked on it waiting to complete a send operation. When this is the case only one task will be unblocked when space on the queue becomes available.

The task that is unblocked will always be the **highest priority task** that was waiting for space. If the blocked tasks have equal priority, then it will be the task that has been waiting for space the longest that is unblocked.
This simulation shows the sequence of reads/writes:



Souce: FreeRTOS

Use FreeRTOS Queue API

Till now we have covered the basics of Queues write and read process. Now let's begin with the main part of this tutorial. This section covers the following contents of **FreeRTOS queue API**:

- How to create queue
- How to write and read data/message

Creating Queue with FreeRTOS

xQueueCreate() API Function is use to create a queue with required number of buffer elements and size of each element.

```
QueueHandle_t xQueueCreate( UBaseType_t  Queue_Length, UBaseType_t Item_Size );
```

xQueueCreate() allocates memory according to queue length and data size. Assigned memory also holds the required data structure for a queue. Therefore, be careful while using it with Arduino Because Arduino comes with limited memory.

This function use reference by handles with a return type variable QueueHandle_t. This handler decides if enough memory is available on Arduino to create queue not.

Argument	Description
Return Value	Handler returns NULL value, if enough memory is not available on heap to create queue otherwise it returns Non-Null value
Queue_Length	The length of queue that is the total number of elements it can hold
Item_Size	Size of each item in bytes

For example, to **create a Queue**, create a variable of type xQueueHandle. This is used to store the handle to the queue return variable.

```
xQueueHandle  long_queue;
```

The queue is created to hold a maximum of 5 values, each of which is large enough to hold a variable of type long.

```
Long_queue = xQueueCreate( 5, sizeof( long ) );
```

Before creating tasks that read and write data to queue, you should check long_queue handler return value this:

```
if( long_queue != NULL )
{
//create read and write message routines
}
```

This is how we create queue with FreeRTOS in Arduino IDE. On top of that, xQueueReset() API function can also be used to reset it to its original state.

FreeRTOS Sending data to Queue

FreeRTOS API provides these two functions to read and write message to/from queue.

1. xQueueSendToBack() or xQueueSendToFront() (both have same functionality)
2. xQueueSendToFront()

As their name suggests, these functions write data to the front or back end of a buffer. These function takes three arguments as an input and returns one value that is success or failure of message write.

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

Arguments	Details
xQueue	This provides the reference of the queue to which we want to write data. It is the same handle variable that we used while creating queue with xQueueCreate()
pvItemToQueue	A pointer variable to the message that we want to send to queue
TickType_t xTicksToWait	In case, if queue is full, it specifies the maximum blocking time of a task until space to become available. Passing value zero will force the task to return without writing, If queue is full. Similarly, passing portMAX_DELAY to function will cause the task to remain in blocking state indefinitely
Return value	It returns two values such as pdPASS (if data is successfully written) and errQUEUE_FUL(if data could not be written)

FreeRTOS Receiving data from Queue

Similar to write API functions, xQueueReceive() is used to read message/data from a queue.

Note: Once you read the data using xQueueReceive(), it will be removed from the queue.

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

All input arguments and return value of xQueueReceive() has same working as write API except second argument “const pvBuffer”. It is a pointer to a variable to which we want to store received data.

Example

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


TaskHandle_t TaskHandle_1;
TaskHandle_t TaskHandle_2;
 TaskHandle_t TaskHandle_4;
static QueueHandle_t msg_queue1;
static QueueHandle_t msg_queue2;

/* Task4 with priority 4 */
static void MyTask4(void* p)
{
    printf("\nTask4 Running, priority: ");
    printf("%d", uxTaskPriorityGet(TaskHandle_4));
    long lValueToSend;

lValueToSend = ( long ) p;
for( ;; )
{
xQueueSend( msg_queue2, &lValueToSend, portMAX_DELAY );
printf( "Sent = ");
printf("%lf",lValueToSend);
lValueToSend++;
vTaskDelay(50);

   }


/* Task2 receiver with priority 2 */
static void MyTask2(void* p)
{
   long lReceivedValue;
for( ;; )
{

if (xQueueReceive( msg_queue1, &lReceivedValue, portMAX_DELAY ) == pdPASS)
{
printf( "Received = ");
printf("%l",lReceivedValue);
}
if (xQueueReceive( msg_queue2, &lReceivedValue, portMAX_DELAY ) == pdPASS)
{
printf( "Received = ");
printf("%l",lReceivedValue);
}

vTaskDelay(500);

}
```

```
/* Task1 sender  with priority 1 */
static void MyTask1(void* p)
{
    long lValueToSend;

lValueToSend = ( long ) p;
for( ;; )
{
xQueueSend( msg_queue1, &lValueToSend, portMAX_DELAY );
printf( "Sent = ");
printf("%lf",lValueToSend);
lValueToSend++;
vTaskDelay(50);
}


void main_blinky(void)
{

    printf("\nIn main function");

   msg_queue1 = xQueueCreate( 4, sizeof( long ) );
msg_queue2 = xQueueCreate( 6, sizeof( long ) );
if( msg_queue1 != NULL )
{
xTaskCreate( MyTask1, "Sender1", 240, ( void * ) 100, 1, NULL );
xTaskCreate( MyTask2, "Receiver", 240, NULL, 1, NULL );
}

if(msg_queue2 != NULL)
{
xTaskCreate( MyTask4, "Sender2", 240, ( void * ) 200, 1, NULL );
}
    vTaskStartScheduler();
    while (1)
    {
        printf("Loop Function");
        for (int i = 0; i < 100; i++);

    }
}
```