Queue Management

(slides are based on Prof. Dr. Jian-Jia Chen and http://www.freertos.org)

Anas Toma

LS 12, TU Dortmund

November 23, 2017

Introduction

What is A Queue?

- A queue is a particular kind of collection in which the entities in the collection are kept in order.
- **Principal operations**: the addition of entities to the rear terminal position and removal of entities from the front terminal position.
- This makes the queue a **First-In-First-Out (FIFO)** data structure.

Introduction (cont.)

Why do we need queues?

- FreeRTOS applications are structured as a set of independent tasks (mini programs)
  - Communication?
  - Use queues for communication and synchronization

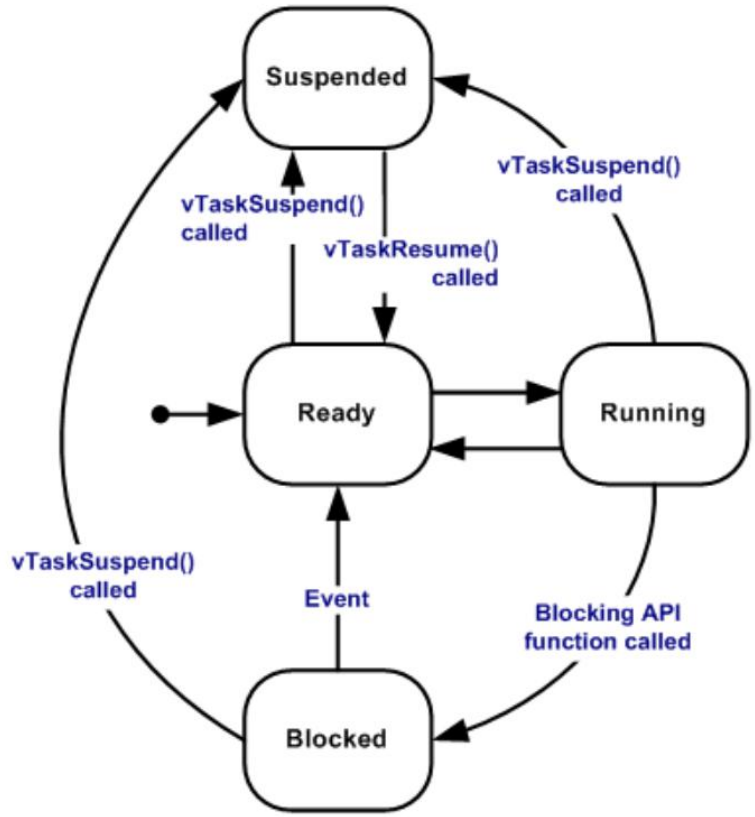
- Queues provide communication mechanism
  - **task-to-task**
  - task-to-interrupt
  - interrupt-to-task

Characteristics of A Queue

- Data Storage
  - A queue can hold a finite number of (fixed-sized) data items.
  - The maximum number of items a queue can hold is called its "length" or "capacity".
  - Usually, the size of a data item and the length (capacity) of a queue is set when being initialized
  - Normal operations are to "add" to the rear and to "remove" from the head
  - Exceptional operations can also allow to "add" to the head
  - Writing data to a queue causes a byte-for-byte copy of the data
  - Reading data from a queue causes a byte-for-byte copy of the data
- Access by Multiple Tasks
  - Items in a queue do not have any specific owners
  - These items are shared and accessed by multiple tasks
  - It is usual to have multiple writers, but unusual to have multiple readers.

Blocking Behavior from A Queue Operation - Read

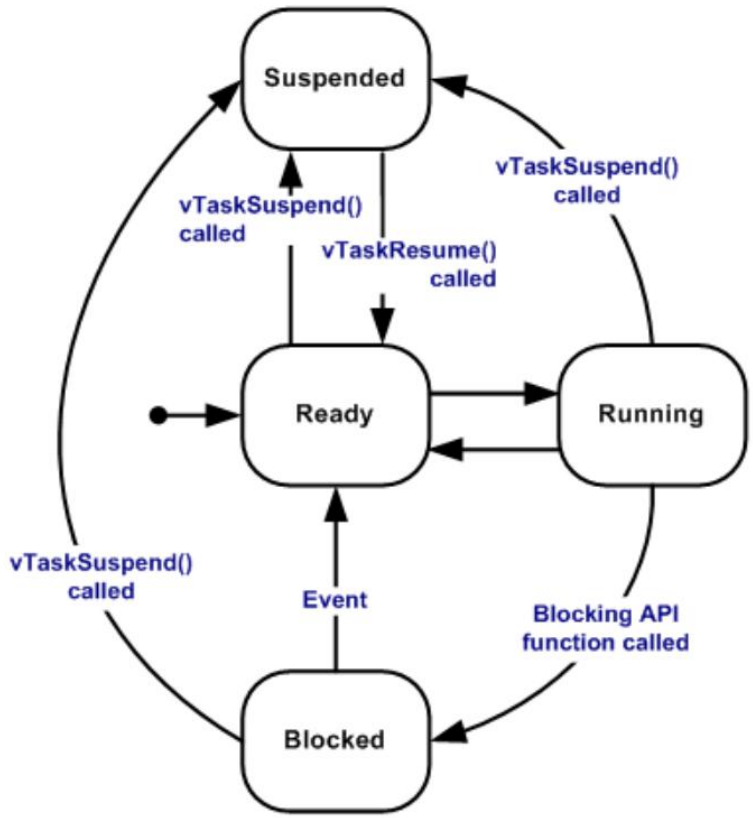
- When a task attempts to read from a queue, it can specify a **“block” time**
- If the queue is empty:
  - The task should be kept in the Blocked state to wait for data to be available from the queue **until the specified “block” time expires**
  - A task that is in the “Blocked” state, waiting for data to be come available from a queue is automatically moved to the “Ready” state **when another task or interrupt places data** into the queue
- When multiple readers are present, the task with the **highest priority** is usually chosen to be moved to the “ready” state.



[http://www.freertos.org]

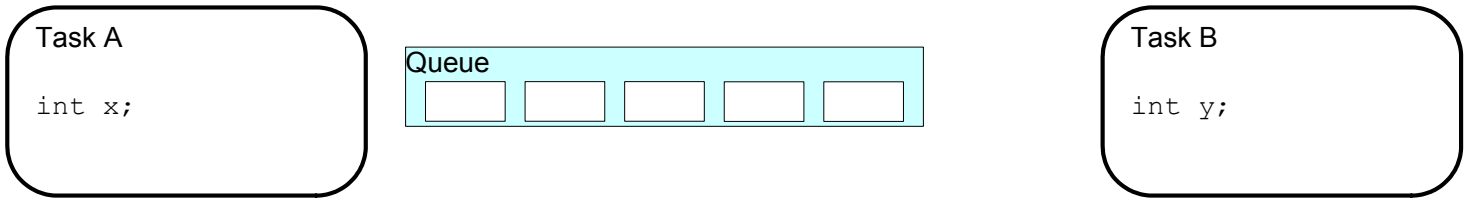
Blocking Behavior from A Queue Operation - Write

- When a task attempts to write to a queue, it can specify a **“block” time**
- If the queue is full:
  - The task should be kept in the Blocked state to wait for the queue to be available for writing **until the specified “block” time expires**
  - A task that is in the “Blocked” state, waiting for the queue to be come available for writing is automatically moved to the “Ready” state **when another task or interrupt removes data from the queue**
- When multiple writers are present, the task with the **highest priority** is usually chosen to be moved to the “ready” state.

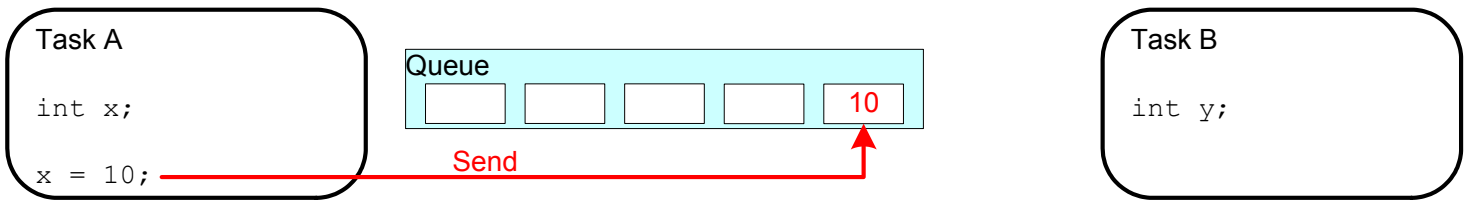


[http://www.freertos.org]

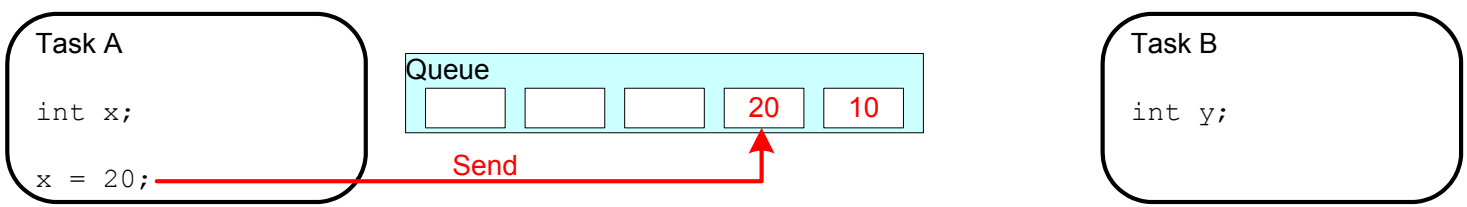
How A Queue Works



A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

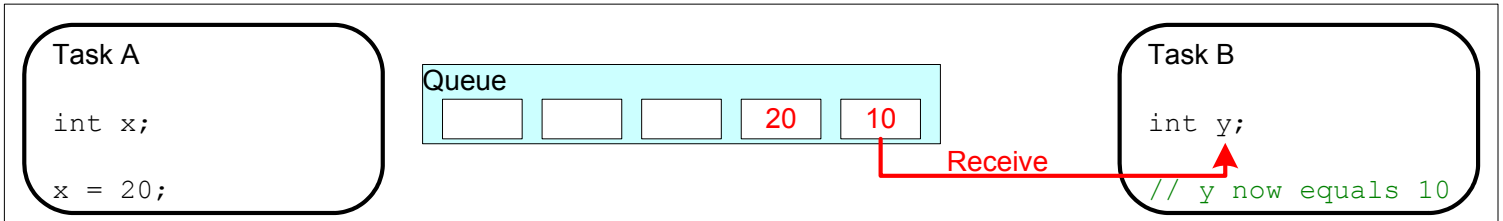


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

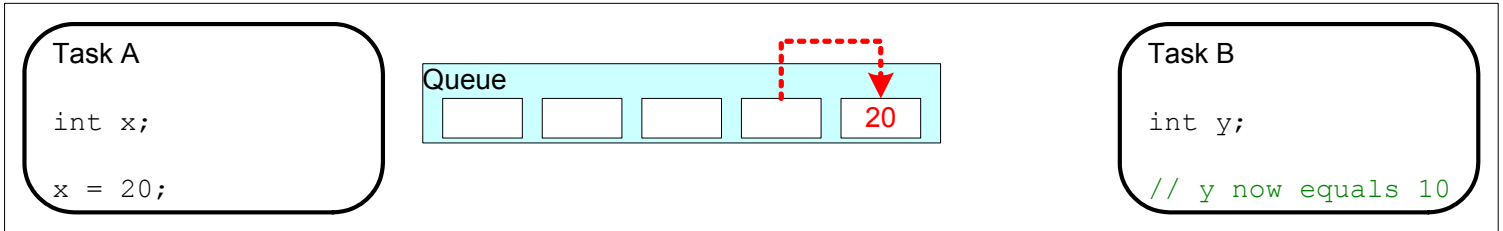


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.

How A Queue Works (cont.)



Task B reads (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 (10 in this illustration).



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 four empty spaces remaining.

FreeRTOS: Create A Queue

xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize)

- The created queue is stored in the heap memory of FreeRTOS
  - The memory holds both the queue data structures and the queued items
- The return value should be stored as the handle (pointer) to the queue.
  - If the returned value is NULL, the queue allocation fails due to insufficient heap memory
- Look at queue.c for
  - Definition of queue data structure xQUEUE
  - Function xQueueCreate
    - Why do we need to call pvPortMalloc twice?

Definition of Queue

```
typedef struct QueueDefinition
{
  signed char *pcHead;/*< Points to the beginning of the queue storage area.*/
  signed char *pcTail;/*< Points to the byte at the end of the queue storage
area.*/

  signed char *pcWriteTo; /*< Points to the free next place in the storage area*/
  signed char *pcReadFrom;/*< Points to the last place that a queued item was read
from.*/

  xList xTasksWaitingToSend; /*< List of tasks that are blocked waiting to post
onto this queue. Stored in priority order. */
  xList xTasksWaitingToReceive; /*< List of tasks that are blocked waiting to read
from this queue. Stored in priority order. */

  volatile unsigned portBASE_TYPE uxMessagesWaiting;/*< The number of items
currently in the queue. */
  unsigned portBASE_TYPE uxLength; /*< The length of the queue defined as the
number of items it will hold, not the number of bytes. */
  unsigned portBASE_TYPE uxItemSize; /*< The size of each items that the queue
will hold. */

  ...

} xQUEUE;
```

xQueueCreate

```
xQueueHandle xQueueGenericCreate( unsigned portBASE_TYPE uxQueueLength, unsigned
portBASE_TYPE uxItemSize, unsigned char ucQueueType ){
xQUEUE *pxNewQueue; size_t xQueueSizeInBytes; xQueueHandle xReturn = NULL;

  /* Allocate the new queue structure. */
  if( uxQueueLength > ( unsigned portBASE_TYPE ) 0 ){
    pxNewQueue = ( xQUEUE * ) pvPortMalloc( sizeof( xQUEUE ) );
    if( pxNewQueue != NULL ){
      /* Create the list of pointers to queue items.  The queue is one byte
      longer than asked for to make wrap checking easier/faster. */
      xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize ) + ( size_t ) 1;
      pxNewQueue->pcHead = ( signed char * ) pvPortMalloc( xQueueSizeInBytes );
      if( pxNewQueue->pcHead != NULL ){
        /* Initialise the queue members as described above where the queue type is
defined. */
        pxNewQueue->uxLength = uxQueueLength;
        pxNewQueue->uxItemSize = uxItemSize;
        xQueueGenericReset( pxNewQueue, pdTRUE );
        ...
        xReturn = pxNewQueue;}
      else{
        traceQUEUE_CREATE_FAILED( ucQueueType );
        vPortFree( pxNewQueue );
      }
    }
  }
  configASSERT( xReturn );return xReturn;}
```

APIs for Queue Operations

- portBASE_TYPE xQueueSendToFront(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);
  - send an item to the front of the queue
  - queue.h: #define xQueueSendToFront(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_FRONT)
- portBASE_TYPE xQueueSendToBack(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);
  - send an item to the end of the queue
  - queue.h: #define xQueueSendToBack(xQueue, pvItemToQueue, xTicksToWait) xQueueGenericSend((xQueue), (pvItemToQueue), (xTicksToWait), queueSEND_TO_BACK)
- portBASE_TYPE xQueueReceive(xQueueHandle xQueue, void * pvBuffer, portTickType xTicksToWait);
  - receive an item from the front of the queue
  - queue.h: #define xQueueReceive(xQueue, pvBuffer, xTicksToWait) xQueueGenericReceive((xQueue), (pvBuffer), (xTicksToWait), pdFALSE)
  - use xQueuePeek to receive an item without removing it from the queue.

APIs for Queue Operations (cont.)

- The API's return immediately if:
  - xTicksToWait is 0
  - the queue is full
- The blocking time is specified in ticks period
  - Use the constant **portTICK_RATE_MS** to convert between the time in milliseconds and in ticks.
  - Time in ticks = time in milliseconds / portTICK_RATE_MS

- portBASE_TYPE uxQueueMessagesWaiting(const xQueueHandle xQueue);
  - Query the number of items in a queue

Critical Sections

- A critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.
- Why do we have to worry about the critical section problem here when "maintaining" a queue?
- Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use.

FreeRTOS: Critical Sections (task.h)

- taskENTER_CRITICAL
  - Macro to mark the start of a critical code region.
  - Preemptive context switches cannot occur when in a critical region.
  - This may alter the stack. Used with care!
- taskEXIT_CRITICAL
  - Macro to mark the end of a critical code region.
  - Preemptive context switches cannot occur when in a critical region.
  - This may alter the stack. Used with care!

```
void vPortEnterCritical( void )
{
    portDISABLE_INTERRUPTS();
    uxCriticalNesting++;
}
/*-----------------------------------------------------------*/

void vPortExitCritical( void )
{
    uxCriticalNesting--;
    if( uxCriticalNesting == 0 )
    {
        portENABLE_INTERRUPTS();
    }
}
```

xQueueGenericSend

```
signed portBASE_TYPE xQueueGenericSend( xQueueHandle pxQueue, const void * const pvItemToQueue, portTickType xTicksToWait, portBASE_TYPE xCopyPosition )
{
signed portBASE_TYPE xEntryTimeSet = pdFALSE;
xTimeOutType xTimeOut;

    configASSERT( pxQueue );
    configASSERT( !( ( pvItemToQueue == NULL ) && ( pxQueue->uxItemSize != ( unsigned portBASE_TYPE ) 0U ) ) );

    for( ;; )
    {
        taskENTER_CRITICAL();
        {
            /* Is there room on the queue now?  To be running we must be
            the highest priority task wanting to access the queue. */
            if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
            {
                traceQUEUE_SEND( pxQueue );
                prvCopyDataToQueue( pxQueue, pvItemToQueue, xCopyPosition );

                if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) == pdFALSE )
                {
                    if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) == pdTRUE )
                    {
                        /* The unblocked task has a priority higher than
                        our own so yield immediately.  Yes it is ok to do
                        this from within the critical section - the kernel
                        takes care of that. */
                        portYIELD_WITHIN_API();
                    }
                }

                taskEXIT_CRITICAL();

                return pdPASS;
```

xQueueGenericSend (cont.)

```
        }
        else
        {
            if( xTicksToWait == ( portTickType ) 0 )
            {
                /* The queue was full and no block time is specified (or
                the block time has expired) so leave now. */
                taskEXIT_CRITICAL();

                /* Return to the original privilege level before exiting
                the function. */
                traceQUEUE_SEND_FAILED( pxQueue );
                return errQUEUE_FULL;
            }
            else if( xEntryTimeSet == pdFALSE )
            {
                /* The queue was full and a block time was specified so
                configure the timeout structure. */
                vTaskSetTimeOutState( &xTimeOut );
                xEntryTimeSet = pdTRUE;
            }
        }
    }
    taskEXIT_CRITICAL();

    /* Interrupts and other tasks can send to and receive from the queue
    now the critical section has been exited. */

    vTaskSuspendAll();
    prvLockQueue( pxQueue );

    /* Update the timeout state to see if it has expired yet. */
    if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) == pdFALSE )
    {
        if( prvIsQueueFull( pxQueue ) != pdFALSE )
        {
            traceBLOCKING ON QUEUE SEND( pxQueue );
```

xQueueGenericSend (cont.)

```
            vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToSend ), xTicksToWait );

            /* Unlocking the queue means queue events can effect the
            event list.  It is possible that interrupts occurring now
            remove this task from the event list again - but as the
            scheduler is suspended the task will go onto the pending
            ready last instead of the actual ready list. */
            prvUnlockQueue( pxQueue );

            /* Resuming the scheduler will move tasks from the pending
            ready list into the ready list - so it is feasible that this
            task is already in a ready list before it yields - in which
            case the yield will not cause a context switch unless there
            is also a higher priority task in the pending ready list. */
            if( xTaskResumeAll() == pdFALSE )
            {
                portYIELD_WITHIN_API();
            }
        }
        else
        {
            /* Try again. */
            prvUnlockQueue( pxQueue );
            ( void ) xTaskResumeAll();
        }
    }
    else
    {
        /* The timeout has expired. */
        prvUnlockQueue( pxQueue );
        ( void ) xTaskResumeAll();

        /* Return to the original privilege level before exiting the
        function. */
        traceQUEUE_SEND_FAILED( pxQueue );
        return errQUEUE_FULL;
    }
```

xQueueGenericReceive

```
signed portBASE_TYPE xQueueGenericReceive( xQueueHandle pxQueue, void * const pvBuffer, portTickType
 xTicksToWait, portBASE_TYPE xJustPeeking )
{
signed portBASE_TYPE xEntryTimeSet = pdFALSE;
xTimeOutType xTimeOut;
signed char *pcOriginalReadPosition;

    configASSERT( pxQueue );
    configASSERT( !( ( pvBuffer == NULL ) && ( pxQueue->uxItemSize != ( unsigned portBASE_TYPE ) 0U
) ) );

    /* This function relaxes the coding standard somewhat to allow return
    statements within the function itself.  This is done in the interest
    of execution time efficiency. */

    for( ;; )
    {
        taskENTER_CRITICAL();
        {
            /* Is there data in the queue now?  To be running we must be
            the highest priority task wanting to access the queue. */
            if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
            {
                /* Remember our read position in case we are just peeking. */
                pcOriginalReadPosition = pxQueue->pcReadFrom;

                prvCopyDataFromQueue( pxQueue, pvBuffer );

                if( xJustPeeking == pdFALSE )
                {
                    traceQUEUE_RECEIVE( pxQueue );

                    /* We are actually removing data. */
                    --( pxQueue->uxMessagesWaiting );

                    #if ( configUSE_MUTEXES == 1 )
                    {
```

ISR-Safe APIs

Never call the following API's from an interrupt service routine:

- xQueueSendToFront()
- xQueueSendToBack()
- xQueueReceive
- etc.

Use the following:

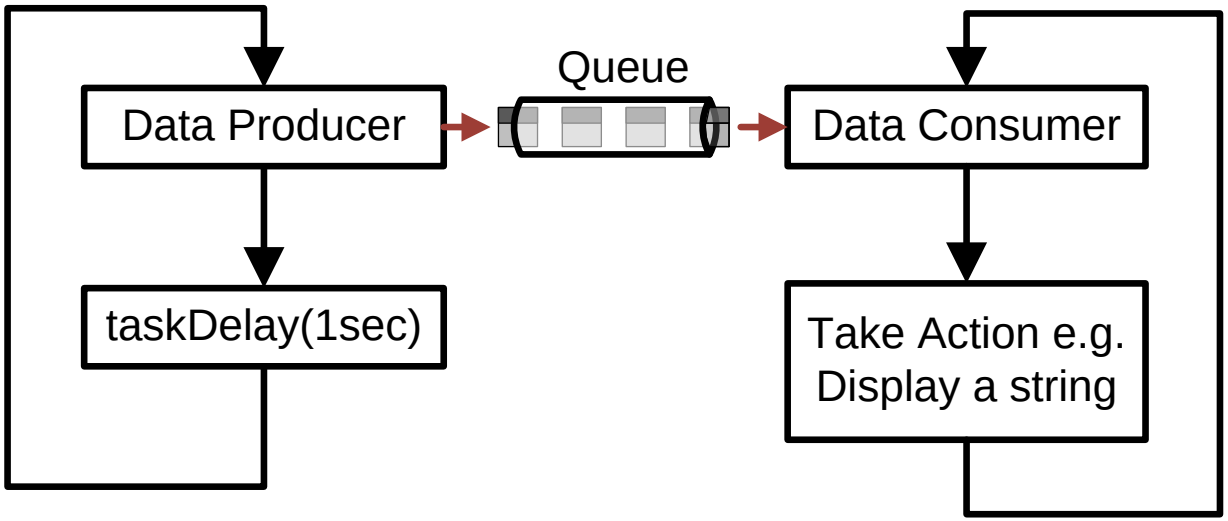
- xQueueSendToFrontFromISR
- xQueueSendToBackFromISR
- xQueueReceiveFromISR
- etc.
- Be careful when using semaphores and queues in ISR

How Is Semaphore Implemented in FreeRTOS?

```
1 #define vSemaphoreCreateBinary( xSemaphore ) { \
2 ( xSemaphore ) = xQueueCreate( ( unsigned ↩
      portBASE_TYPE ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH↩
       ); \
3 if( ( xSemaphore ) != NULL ) \
4 { \
5 xSemaphoreGive( ( xSemaphore ) ); \
6 } \
7                                                        }↩

8 #define xSemaphoreTake( xSemaphore, xBlockTime ) ↩
      xQueueGenericReceive( ( xQueueHandle ) ( ↩
      xSemaphore ), NULL, ( xBlockTime ), pdFALSE )
9 #define xSemaphoreGive( xSemaphore ) ↩
      xQueueGenericSend( ( xQueueHandle ) ( xSemaphore ↩
      ), NULL, semGIVE_BLOCK_TIME, queueSEND_TO_BACK )
```

Queue: Functional Aspects



- Single Queue can have multiple 'producers' and 'consumers'
- Consumer blocks when queue empty
- Producer blocks when queue full

Queue Example: Reciever

```
1 /*Globally accessbile queue handle*/
2 xQueueHandle DispTaskQ;
3
4 /*Consumer Task*/
5 void vDispTask( void *pvParameters )
6 {
7   char * xMessage;
8
9   /*Create a queue with a length of one char* */
10  DispTaskQ = xQueueCreate(1, sizeof( char* ));
11
12  for( ;; ){
13
14    /* Waiting for the message from sender*/
15    xQueueReceive( DispTaskQ, &xMessage, ↩
          portMAX_DELAY );
16    print( xMessage, 15, 10, 11 );
17    }
18 }
```

Queue Example: Sender

```
1  /*Producer Task*/
2  void vSendTask( void *pvParameters )
3  {
4    char * str;
5    char counter=0;
6    str = pvPortMalloc(25*sizeof(char));
7
8    for( ;; ){
9      vTaskDelay( 500 );
10     sprintf(str, "%u. Send On Q",counter++);
11     xQueueSend(DispTaskQ, &str, 0);
12     }
13 }
```

Large Items in the Queue

- Receiving and sending require a copy operation of “data”
- It is inefficient to exchange large data structure in the queue
- A more efficient way is to send the “pointers”
  - However, the designers have to be very careful for using such a feature.
  - The owner of the RAM being pointed to is clearly defined.
  - A data item should never be changed until the recipients finish the operations on it.
  - The RAM being pointed to remains valid.
  - A pointer should never be used to access data that has been allocated on a task stack. The data will not be valid after the stack frame has changed.