

# FreeRTOS

A real time operating system for embedded systems

# QUEUE MANAGEMENT

## 2.1 Queue Management

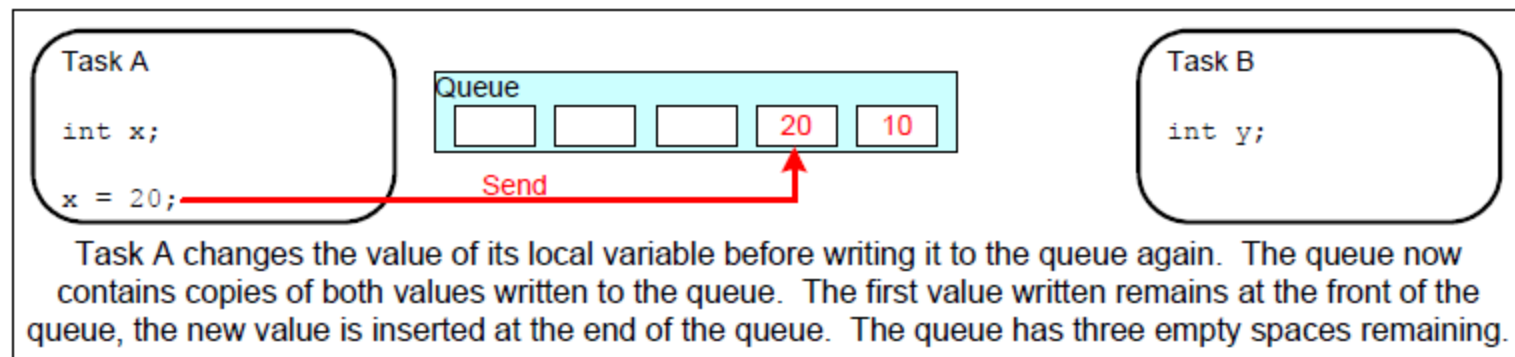
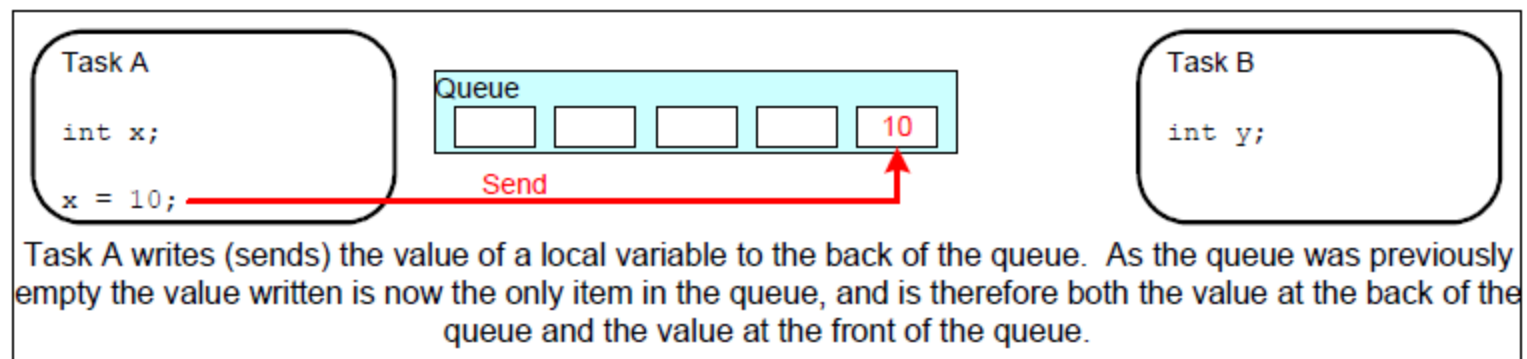
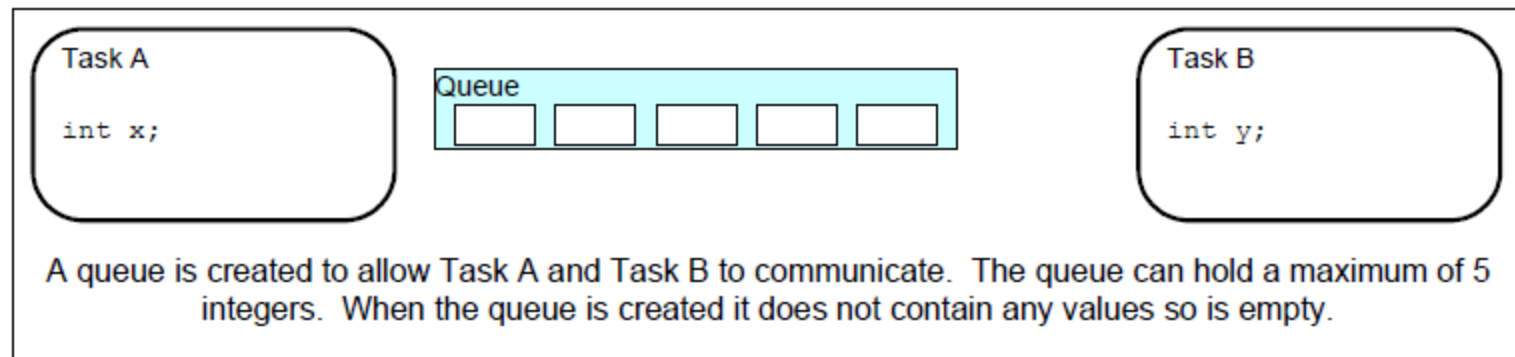
- FreeRTOS applications are structured as a set of independent tasks
  - Each task is effectively a mini program in its own right.
  - It will have to communicate with each other to collectively provide useful system functionality.
- Queue is the underlying primitive
  - Be used for communication and synchronization mechanisms in FreeRTOS.

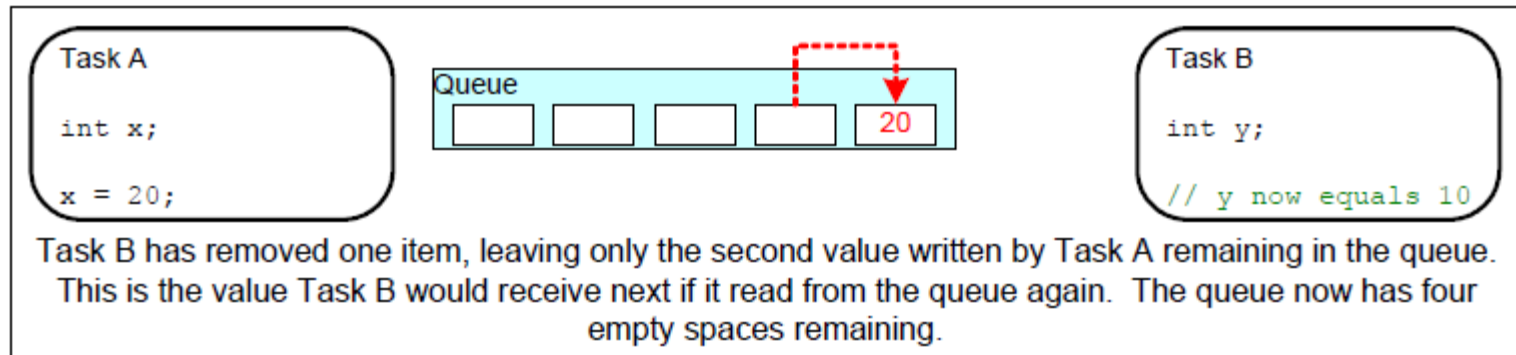
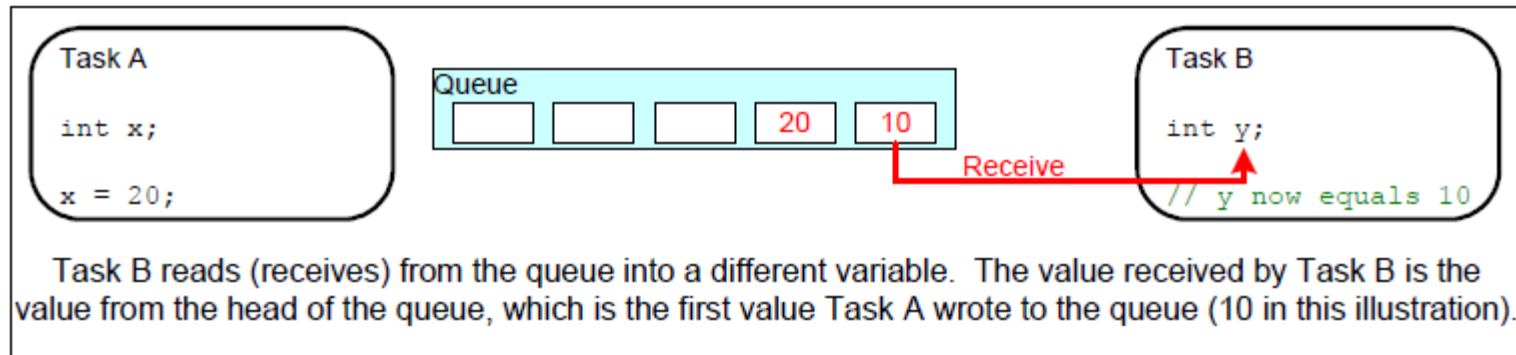
# Queue: Task-to-task communication

- Scope
  - How to create a queue
  - How a queue manages the data it contains
  - How to send data to a queue
  - How to receive data from a queue
  - What it means to block on a queue
  - The effect of task priorities when writing to and reading from a queue

## 2.2 Queue Characteristics – Data storage

- A queue can hold a finite number of fixed size data items.
  - Normally, used as FIFO buffers where data is written to the end of the queue and removed from the front of the queue.
  - Also possible to write to the front of a queue.
  - Writing data to a queue causes a byte-for-byte copy of the data to be stored in the queue itself.
  - Reading data from a queue causes the copy of the data to be removed from the queue.





- Queues are objects in their own right
  - Not owned by or assigned to any particular task
  - Any number of tasks can write to the same queue and any number of tasks can read from the same queue.
  - Very common to have multiple writers, but very rare to have multiple readers.

# Blocking on Queue Reads

- When a task attempts to read from a queue it can optionally specify a 'block' time
  - The maximum time that the task should be kept in the Blocked state to wait for data to be available from the queue should the queue already be empty.
  - It is automatically moved to the Ready state when another task or interrupt places data into the queue.
  - It will also be moved automatically from the Blocked state to the Ready state if the **specified block time** expires before data becomes available.
  - Queues can have multiple readers, so it is possible for a single queue to have more than one task blocked on it waiting for data. When this is the case:



# Blocking on Queue Reads

- Only one task will be unblocked when data becomes available.
  - Queue can have multiple readers.
    - So, it is possible for a single queue to have more than one task blocked on it waiting for data.
  - The task that is unblocked will always be the **highest priority** task that is waiting for data.
  - If the blocked tasks have equal priority, the task that has been waiting for data the **longest** will be unblocked.

# Blocking on Queue Writes

- A task can optionally specify a 'block' time when writing to a queue.
  - The maximum time that task should be held in the Blocked state to wait for space to be available on the queue, should the queue already be full.

# Blocking on Queue Writes

- Queue can have multiple writers.
  - It is possible for a full queue to have more than one task blocked on it waiting to complete a send operation.
- 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 is waiting for **space**.
  - If the blocked tasks have equal priority, the task that has been waiting for **space** the **longest** will be unblocked.

## 2.3 Using a Queue

- A queue must be **explicitly** created before it can be used.
  - FreeRTOS allocates RAM from the heap when a queue is created.
  - RAM holds both the queue data structures and the items that are contained in the queue.
- **xQueueCreate()** API Function
  - Be used to create a queue and returns an xQueueHandle to reference the queue it creates.

- Function Prototype

```
xQueueHandle xQueueCreate(  
    unsigned portBASE_TYPE uxQueueLength,  
    unsigned portBASE_TYPE uxItemSize);
```

- xQueueLength: The maximum number of items that the queue being created can hold at any one time.
- uxItemSize: the size in bytes of each data item that can be stored in the queue.
- Return Value:
  - if NULL is returned, the queue cannot be created as there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage.
  - A non-NULL value returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.

# xQueueSendToBack() and xQueueSendToFront() API Functions

- xQueueSendToBack()
  - Be equivalent to **xQueueSend()**
  - Be used to send data to the back (tail) of a queue
- xQueueSendToFront()
  - Be used to send data to the front (head) of a queue
- Please note, **never call these two API functions from an interrupt service routine (ISR).**
  - Interrupt-safe versions will be used in their place and described in next chapter.

- Function prototypes

```
portBASE_TYPE xQueueSendToBack (  
    xQueueHandle xQueue,  
    const void * pvlItemToQueue,  
    portTickType xTicksToWait);
```

```
portBASE_TYPE xQueueSendToFront(  
    xQueueHandle xQueue,  
    const void * pvlItemToQueue,  
    portTickType xTicksToWait);
```

- xQueue: The handle of the queue to which the data is being send (written). It will have been returned from the call to xQueueCreate() used to create the queue.

- `pvlItemToQueue`: a pointer to the data to be copied into the queue.
  - The size of each item that the queue can hold is set when the queue is created, so the data will be copied from *pvlItemToQueue* into the queue storage area.
- `xTicksToWait`: the maximum amount of time the task should remain in the Blocked state to wait for the space to become available on the queue, should the queue already be full.
  - if `xTicksToWait` is zero, both APIs will return immediately in case the queue is already full.
  - The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The constant `portTICK_RATE_MS` can be used to convert a time specified in MS into ticks.



- Returned value: **two possible** return values.
  - **pdPASS** will be returned if data was successfully sent to the queue.
    - If a block time was specified, it is possible that the calling task was placed in the Blocked state to wait for another task or interrupt to make room in the queue, before the function returned,
    - Data was successfully written to the queue before the block time expired.
  - **errQUEUE\_FULL** will be returned if data could not be written to the queue as the queue was already full.
    - In a similar scenario that a block time was specified, but it expired before space becomes available in the queue.

# xQueueReceive() and xQueuePeek() API Function

- xQueueReceive()
  - Be used to receive (consume) an item from a queue.  
The item received is removed from the queue.
- xQueuePeek()
  - Be used receive an item from the queue without the item being removed from the queue.
  - Receives the item from the head of the queue.
- Please note, never call these two API functions from an interrupt service routine (ISR).

- Function prototypes

```
portBASE_TYPE xQueueReceive (  
    xQueueHandle xQueue,  
    const void *pvBuffer,  
    portTickType xTicksToWait);
```

```
portBASE_TYPE xQueuePeek(  
    xQueueHandle xQueue,  
    const void * pvBuffer,  
    portTickType xTicksToWait);
```

- xQueue: The handle of the queue from which the data is being received (read). It will have been returned from the call to xQueueCreate().

- pvBuffer: a pointer to the memory into which the received data will be copied.
  - The memory pointed to by pvBuffer must be at least large enough to hold the data item held by the queue.
- xTicksToWait: the maximum amount of time the task should remain in the Blocked state to wait for the data to become available on the queue, should the queue already be empty.
  - if xTicksToWait is zero, both APIs will return immediately in case the queue is already empty.
  - The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The constant portTICK\_RATE\_MS can be used to convert a time specified in MS into ticks.

- Returned value: two possible return values.
  - **pdPASS** will be returned if data was successfully read from the queue.
    - If a block time was not zero, it is possible that the calling task was placed in the Blocked state to wait for another task or interrupt to send the data to the queue before the function is returned,
    - data was successfully read from the queue before the block time expired.
  - **errQUEUE\_EMPTY** will be returned if data could not be read from the queue as the queue was already empty.
    - In a similar scenario that a block time was not zero, but it expired before data was sent.

# uxQueueMessageWaiting() API Function

- Be used to query the number of items that are currently in a queue.

- Prototype

```
unsigned portBASE_TYPE uxQueueMessagesWaiting (  
    xQueueHandle xQueue);
```

- Returned value: the number of items that the queue being queried is currently holding. If zero is returned, the queue is empty.

# Example 10. Blocking when receiving from a queue

- To demonstrate
  - a queue being created,
    - Hold data items of type *long*
  - data being sent to the queue from multiple tasks,
    - Sending tasks do not specify a block time, lower priority than receiving task.
  - And data being received from the queue
    - Receiving task specifies a block time 100ms
- So, queue never contains more than one item
  - Once data is sent to the queue, the receiving task will unblock, pre-empt the sending tasks, and remove the data – leaving the queue empty once again.

```

static void vSenderTask( void *pvParameters )
{
    long lValueToSend;
    portBASE_TYPE xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value. The queue was created to hold values of type long,
    so cast the parameter to the required type. */
    lValueToSend = ( long ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send the value to the queue.

        The first parameter is the queue to which data is being sent. The
        queue was created before the scheduler was started, so before this task
        started to execute.

        The second parameter is the address of the data to be sent, in this case
        the address of lValueToSend.

        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        should the queue already be full. In this case a block time is not
        specified because the queue should never contain more than one item and
        therefore never be full. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full -
            this must be an error as the queue should never contain more than
            one item! */
            vPrintString( "Could not send to the queue.\r\n" );
        }

        /* Allow the other sender task to execute. taskYIELD() informs the
        scheduler that a switch to another task should occur now rather than
        keeping this task in the Running state until the end of the current time
        slice. */
        taskYIELD();
    }
}

```

**Listing 34 Implementation of the sending task used in Example 10.**



```

static void vReceiverTask( void *pvParameters )
{
    /* Declare the variable that will hold the values received from the queue. */
    long lReceivedValue;
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* This call should always find the queue empty because this task will
        immediately remove any data that is written to the queue. */
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }

        /* Receive data from the queue.

        The first parameter is the queue from which data is to be received. The
        queue is created before the scheduler is started, and therefore before this
        task runs for the first time.

        The second parameter is the buffer into which the received data will be
        placed. In this case the buffer is simply the address of a variable that
        has the required size to hold the received data.

        The last parameter is the block time – the maximum amount of time that the
        task should remain in the Blocked state to wait for data to be available
        should the queue already be empty. In this case the constant
        portTICK_RATE_MS is used to convert 100 milliseconds to a time specified in
        ticks. */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value. */
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
        else
        {
            /* Data was not received from the queue even after waiting for 100ms.
            This must be an error as the sending tasks are free running and will be
            continuously writing to the queue. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}

```

Listing 35 Implementation of the receiver task for Example 10.

- vSenderTask() does not specify a block time.
  - continuously writing to the queue

```
xStatus = xQueueSendToBack(xQueue, pvParameters, 0);
```

```
If (xStatus != pdPASS) {
```

```
    vPrintString("Could not send to the queue.\n"); }
```

```
taskYIELD();
```

- vReceiverTask() specifies a block time 100ms.
  - Enter the Blocked state to wait for data to be available, leaves it when either data is available on the queue, or 100ms expires, which should never occur.

```
xStatus = xQueueReceive(xQueue,
```

```
    &xReceivedValue, 100/portTICK_RATE_MS);
```

```
if (xStatus == pdPASS) {
```

```
    vPrintStringAndNumber("Received = ", IReceivedValue );
```

```
}
```

```

/* Declare a variable of type xQueueHandle. This is used to store the reference
to the queue that is accessed by all three tasks. */
xQueueHandle xQueue;

int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type long. */
    xQueue = xQueueCreate( 5, sizeof( long ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue. The task
        parameter is used to pass the value that the task will write to the queue,
        so one task will continuously write 100 to the queue while the other task
        will continuously write 200 to the queue. Both tasks are created at
        priority 1. */
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue. The task is created with
        priority 2, so above the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}

```

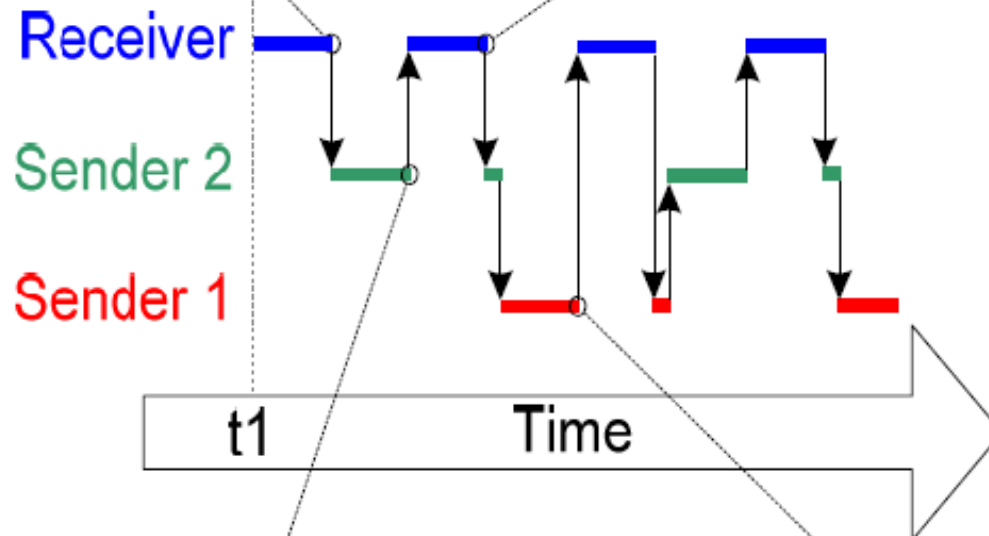
**Listing 36 The implementation of main()Example 10**



# Execution sequence

1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.

3 - The Receiver task empties the queue then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.



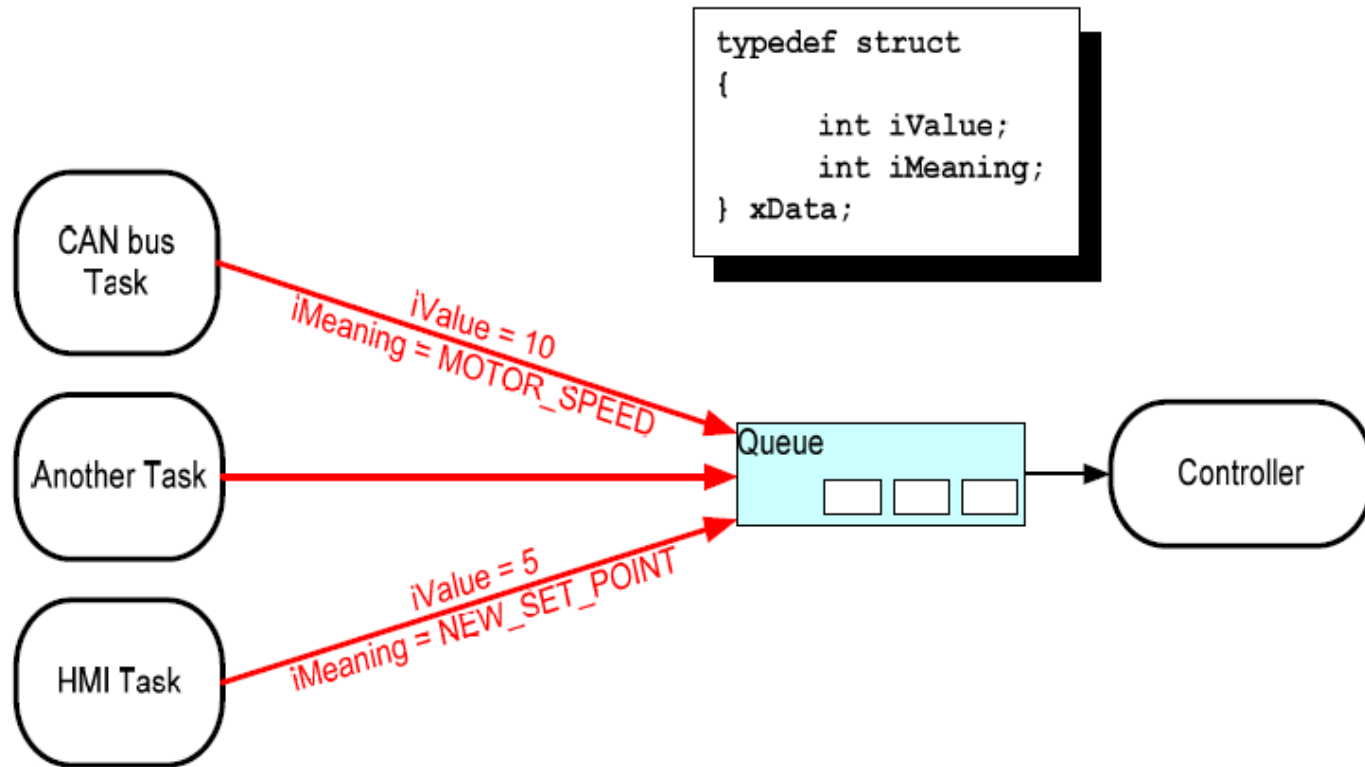
2 - Sender two writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.

4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on .....

# Using Queues to transfer compound types

- It is common for a task to receive data from multiple sources on a single queue.
  - Receiver needs to know the data source to allow it to determine how to process the data.
  - Use the queue to transfer structures which contain both data value and data source, like

```
typedef struct {  
    int iValue; // a data value  
    int iMeaning; // a code indicating data source  
} xData;
```



- Controller task performs the primary system function.
  - React to inputs and changes to the system state communicated to it on the queue.
  - A CAN bus task encapsulates the CAN bus interfacing functionality, like the actual motor speed value.
  - A HMI task encapsulates all the HMI functionality, like the actual new set point value.



# Example 11

- Two differences from Example 10
  - Receiving task has a **lower** priority than the sending tasks.
  - The queue is used to pass structures, rather than simple long integers between the tasks.
- The Queue will normally be full because
  - Once the receiving task removes an item from the queue, it is pre-empted by one of the sending tasks which then immediately refills the queue.
  - Then sending tasks re-enters the Blocked state to wait for space to become available on the queue again.



# Structure Type

```
/* Define the structure type that will be passed on the queue. */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;

/* Declare two variables of type xData that will be passed on the queue. */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Used by Sender1. */
    { 200, mainSENDER_2 } /* Used by Sender2. */

};
```

**Listing 37** The definition of the structure that is to be passed on a queue, plus the declaration of two variables for use by the example.

```

static void vSenderTask( void *pvParameters )
{
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

/* As per most tasks, this task is implemented within an infinite loop. */
for( ;; )
{
    /* Send to the queue.

The second parameter is the address of the structure being sent. The
address is passed in as the task parameter so pvParameters is used
directly.

The third parameter is the Block time - the time the task should be kept
in the Blocked state to wait for space to become available on the queue
if the queue is already full. A block time is specified because the
sending tasks have a higher priority than the receiving task so the queue
is expected to become full. The receiving task will on execute and remove
items from the queue when both sending tasks are in the Blocked state. */
    xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

    if( xStatus != pdPASS )
    {
        /* The send operation could not complete, even after waiting for 100ms.
        This must be an error as the receiving task should make space in the
        queue as soon as both sending tasks are in the Blocked state. */
        vPrintString( "Could not send to the queue.\r\n" );
    }

    /* Allow the other sender task to execute. */
    taskYIELD();
}
}

```

**Listing 38 The implementation of the sending task for Example 11.**

```

static void vReceiverTask( void *pvParameters )
{
    /* Declare the structure that will hold the values received from the queue. */
    xData xReceivedStructure;
    portBASE_TYPE xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* Because it has the lowest priority this task will only run when the
        sending tasks are in the Blocked state. The sending tasks will only enter
        the Blocked state when the queue is full so this task always expects the
        number of items in the queue to be equal to the queue length - 3 in this
        case. */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /* Receive from the queue.

        The second parameter is the buffer into which the received data will be
        placed. In this case the buffer is simply the address of a variable that
        has the required size to hold the received structure.

        The last parameter is the block time - the maximum amount of time that the
        task will remain in the Blocked state to wait for data to be available
        if the queue is already empty. In this case a block time is not necessary
        because this task will only run when the queue is full. */
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value and the source of the value. */
            if( xReceivedStructure.ucSource == mainSENDER_1 )
            {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            /* Nothing was received from the queue. This must be an error
            as this task should only run when the queue is full. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}

```

Listing 39 The definition of the receiving task for Example 11

```

int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type xData. */
    xqueue = xqueueCreate( 3, sizeof( xData ) );

    if( xqueue != NULL )
    {
        /* Create two instances of the task that will write to the queue. The
        parameter is used to pass the structure that the task will write to the
        queue, so one task will continuously send xStructsToSend[ 0 ] to the queue
        while the other task will continuously send xStructsToSend[ 1 ]. Both
        tasks are created at priority 2 which is above the priority of the receiver. */
        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp; xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp; xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue. The task is created with
        priority 1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    CHAPTER 5 provides more information on memory management. */
    for( ;; );
}

```

**Listing 40 The implementation of main() for Example 11**



- In `vSenderTask()`, the sending task specifies a block time of 100ms.
  - So, it enters the Blocked state to wait for space to become available each time the queue becomes full.
  - It leaves the Blocked state when either the space is available on the queue or 100ms expires without space be available (should never expire as receiving task is continuously removing items from the queue).

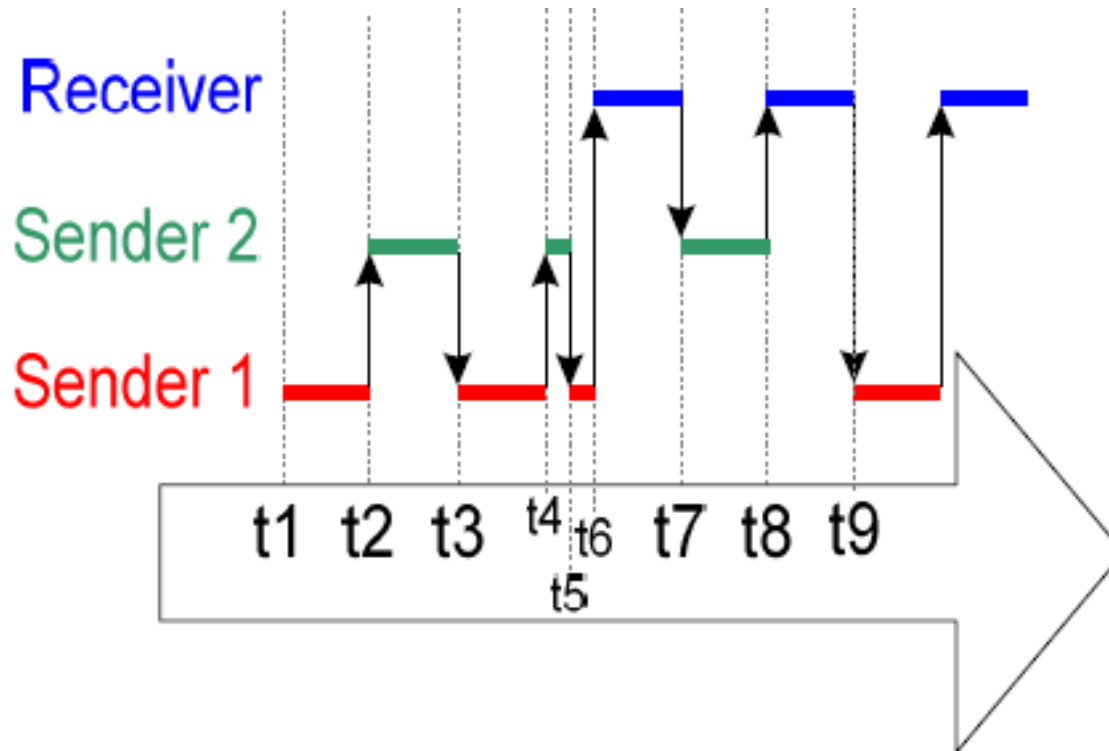
```
xStatus = xQueueSendToBack(xQueue,  
                           pvParameters, 100/portTICK_RATE_MS);  
If (xStatus != pdPASS) {  
    vPrintString("Could not send to the queue.\n");  
}  
taskYIELD();
```



- vReceiverTask() will run only when both sending tasks are in the Blocked state.
  - Sending tasks will enter the Blocked state only when the queue is full as they have higher priorities.
  - The receiving task will execute only when the queue is already full. -> it always expects to receive data even without a 'block' time.

```
xStatus = xQueueReceive(xQueue,  
                        &xReceivedStructure, 0);  
if (xStatus == pdPASS) {  
    // print the data received  
}
```

# Execution sequence – Sender 1 and 2 have higher priorities than Receiver





## 2.4 Working with large data

- It is not efficient to copy the data itself into and out of the queue byte by byte, when the size of the data being stored in the queue is large.
- It is preferable to use the queue to transfer pointers to the data.
  - More efficient in both processing time and the amount of RAM required to create the queue.
- But, when queuing pointers, extreme care must be taken to ensure that:

1. The owner of the RAM being pointed to is clearly defined.
  - When multiple tasks share memory via a pointer, they do not modify its contents simultaneously, or take any other action that cause the memory contents invalid or inconsistent.
    - Ideally, only the sending task is permitted to access the memory until a pointer to the memory has been queued, and only the receiving task is permitted to access the memory after the pointer has been received from the queue.

- ## 2. The RAM being pointed to remains valid.
- If the memory being pointed to was allocated dynamically, exactly one task be responsible for freeing the memory.
  - No task should attempt to access the memory after it has been freed.
  - 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.