Real Time Operating System
“FreeRTOS”
Queue

Agenda

- 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.
- Only task-to-task communication is covered

What is 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.**
- Normally, queues are used as **First In First Out (FIFO)** buffers where data is **written to the end (tail)** of the queue and **removed from the front** (head) of the queue.
- It is 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.**

What is Queue?

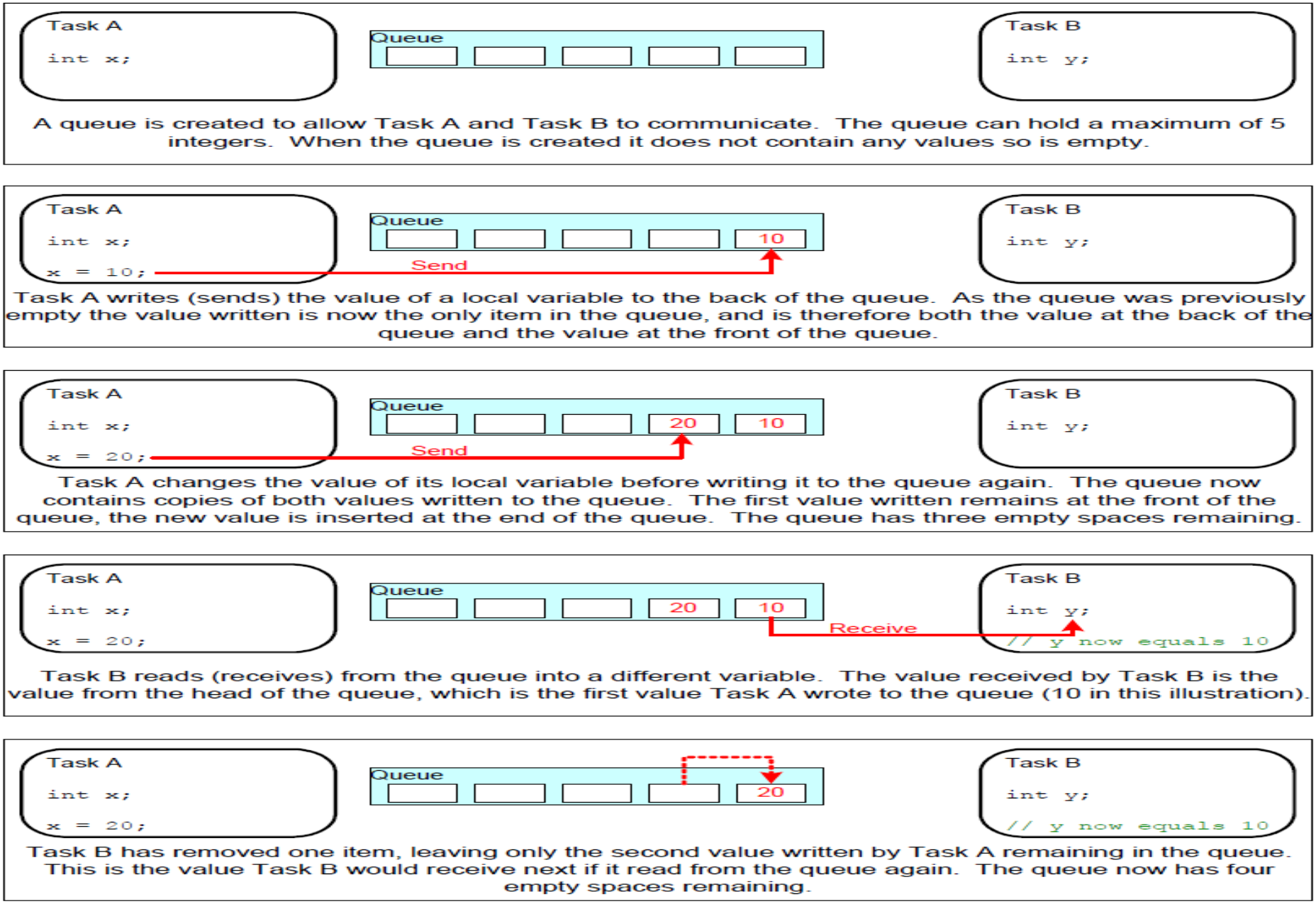


Figure 19. An example sequence of writes and reads to and from a queue

Blocking on Queue Reads

- When a task attempts to read from a queue it can optionally specify a ‘block’ time.
- This is the time 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.
- A task that is in the Blocked state, waiting for data to become available from a queue, is automatically moved to the Ready state when another task or interrupt places data into the queue.
- The task 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.
- Only one task will be unblocked when data becomes available. The task that is unblocked will always be the highest priority task that is waiting for data.
- If the blocked tasks have equal priority, then 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 block time is the maximum time the task should be held in the Blocked state to wait for space to become available on the queue, should the queue already be full.
- Queues can have multiple writers, so 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, then the task that has been waiting for space the longest will be unblocked.

Example 10. Blocking when receiving from a queue

```
/* Declare a variable of type xQueueHandle.  This is used to store the handle
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", 240, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, ( 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", 240, 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 37. The implementation of main() for Example 10

Example 10. Blocking when receiving from a queue

```
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.\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();
    }
}
```

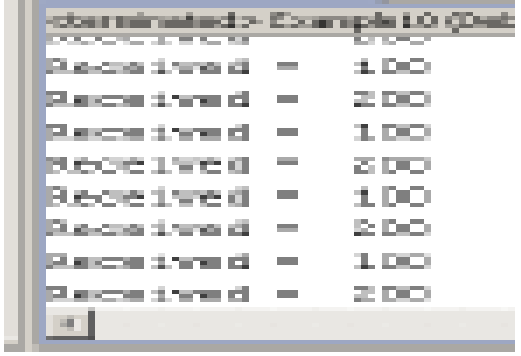
Listing 35. Implementation of the sending task used in Example 10

Example 10. Blocking when receiving from a queue

```
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!\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.\n" );
        }
    }
}
```

Listing 36. Implementation of the receiver task for Example 10

Example 10. Blocking when receiving from a queue



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.

5 - The Receiver task empties the queue then enters the Blocked State. CPU gets back to preempted tasks

Receiver

Sender 2

Sender 1

t1

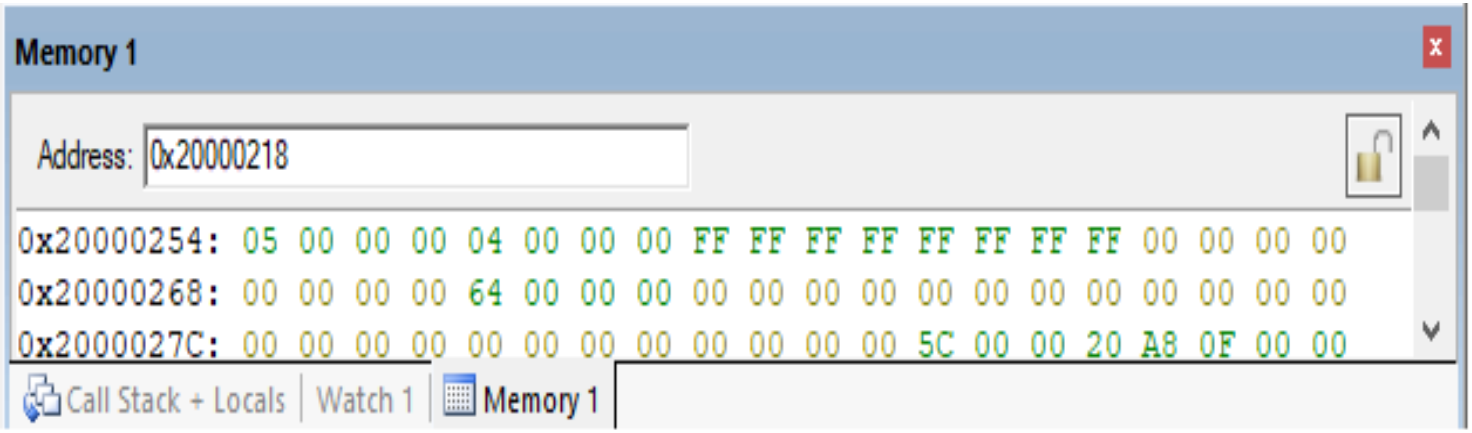
Time

6&7- Premepted Sender 2 completes and Yields and so Sender 1. Receiever is Blocked. So Sender 2 writes to the empty queue before preempted again

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.

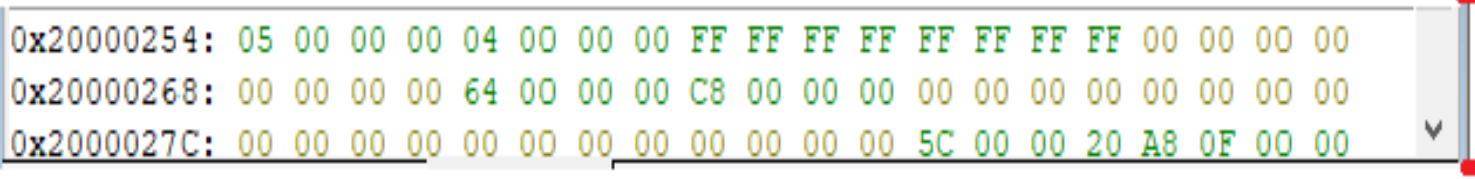
4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on .....

Example 10. Blocking when receiving from a queue



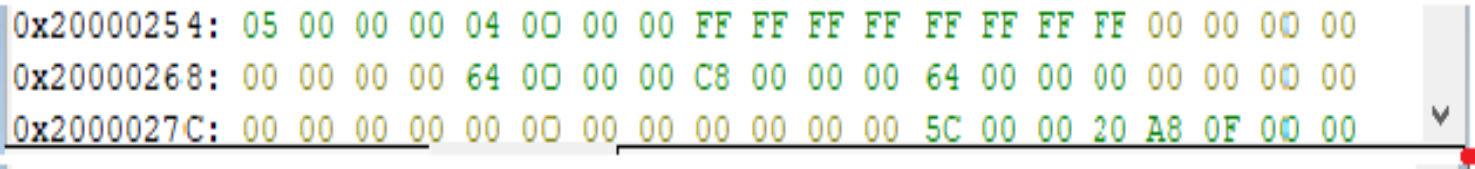
Sender 1: Write No. 1

Receiver reads 64 just after



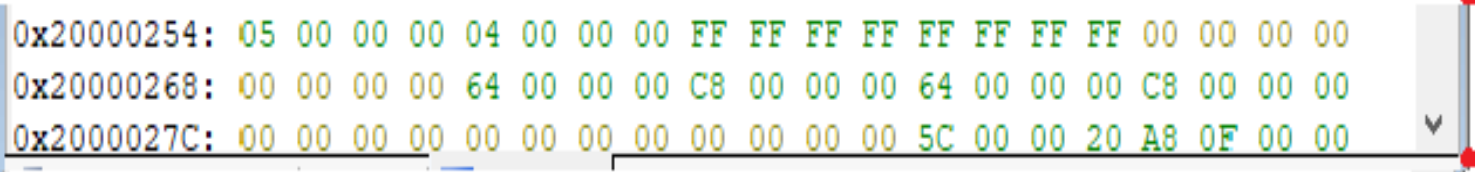
Sender 2: Write No. 2

Receiver reads C8 just after



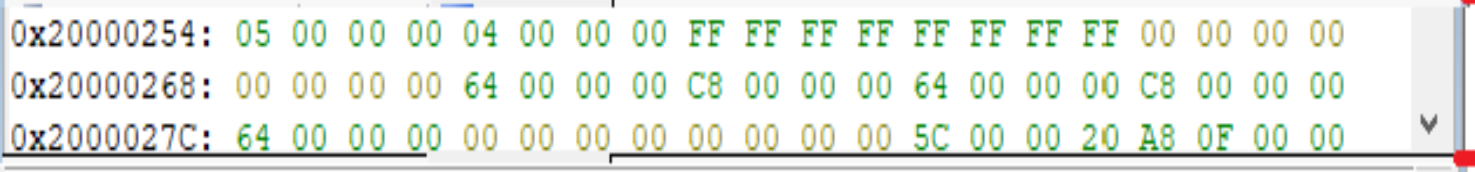
Sender 1: Write No. 3

Receiver reads 64 just after



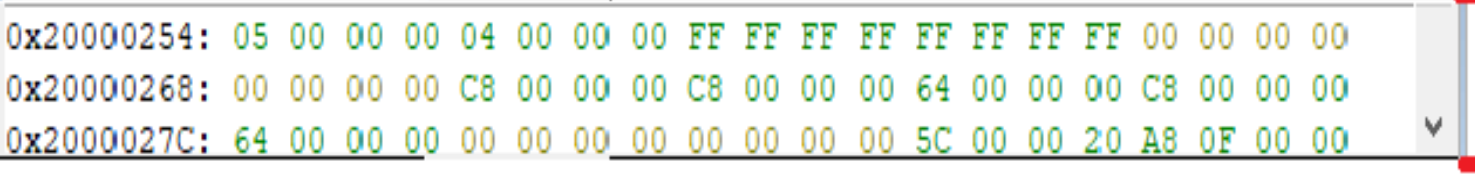
Sender 2: Write No. 4

Receiver reads C8 just after



Sender 1: Write No. 5

Receiver reads 64 just after



Sender 2: Writes No. 6 Circularly

Receiver reads C8 just after

Using Queues to Transfer Compound Types

- A task is to receive data from multiple sources on a single queue.
- The receiver of the data needs to know where the data came from
- Use the queue to transfer structures where both the value of the data and the source of the data are contained

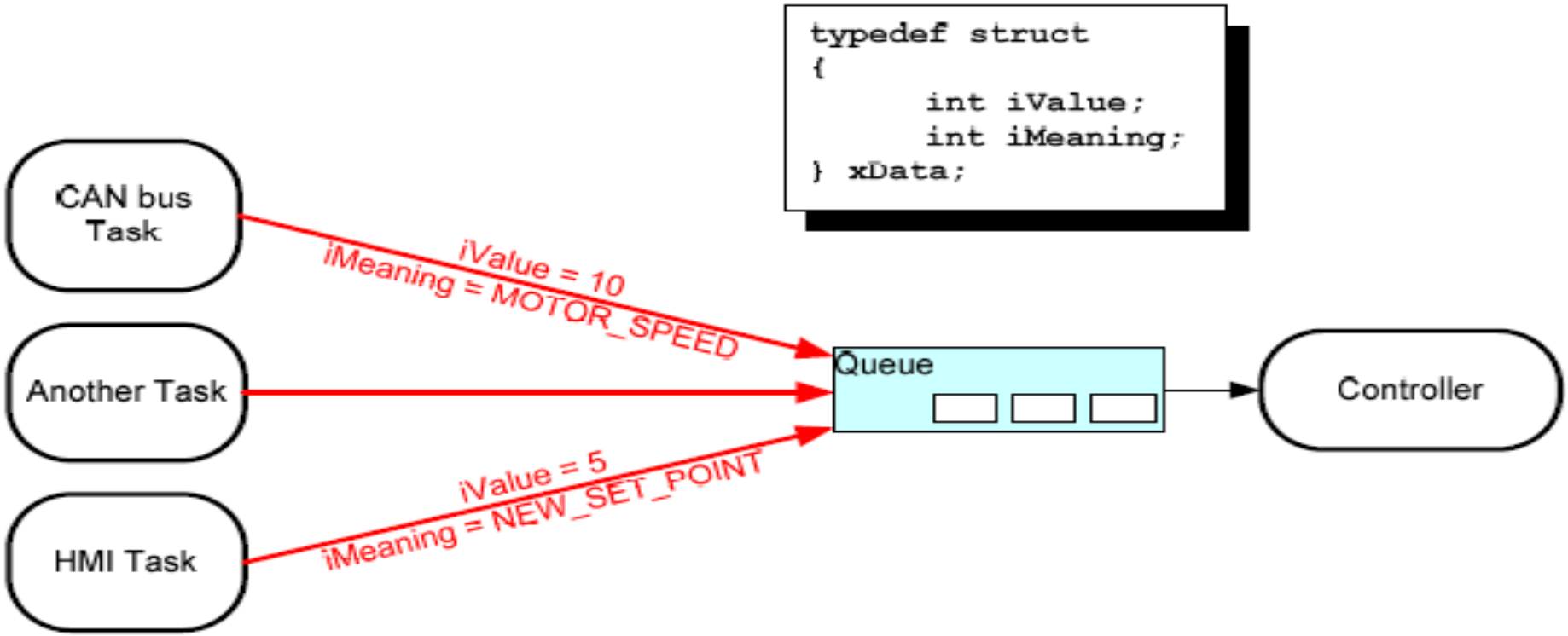


Figure 22. An example scenario where structures are sent on a queue

Example 11. Blocking when sending to a queue or sending structures on a queue

```
/* 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 38. The definition of the structure that is to be passed on a queue, plus the declaration of two variables for use by the example

Example 11. Blocking when sending to a queue or sending structures on a queue

```
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", 240, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 240, &( 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", 240, 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 41. The implementation of main() for Example 11

Example 11. Blocking when sending to a queue or sending structures on a queue

```
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 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.\n" );
        }

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

Listing 39. The implementation of the sending task for Example 11.

Example 11. Blocking when sending to a queue or sending structures on a queue

```
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!\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.\n" );
        }
    }
}
```

Listing 40. The definition of the receiving task for Example 11

Example 11. Blocking when sending to a queue or sending structures on a queue

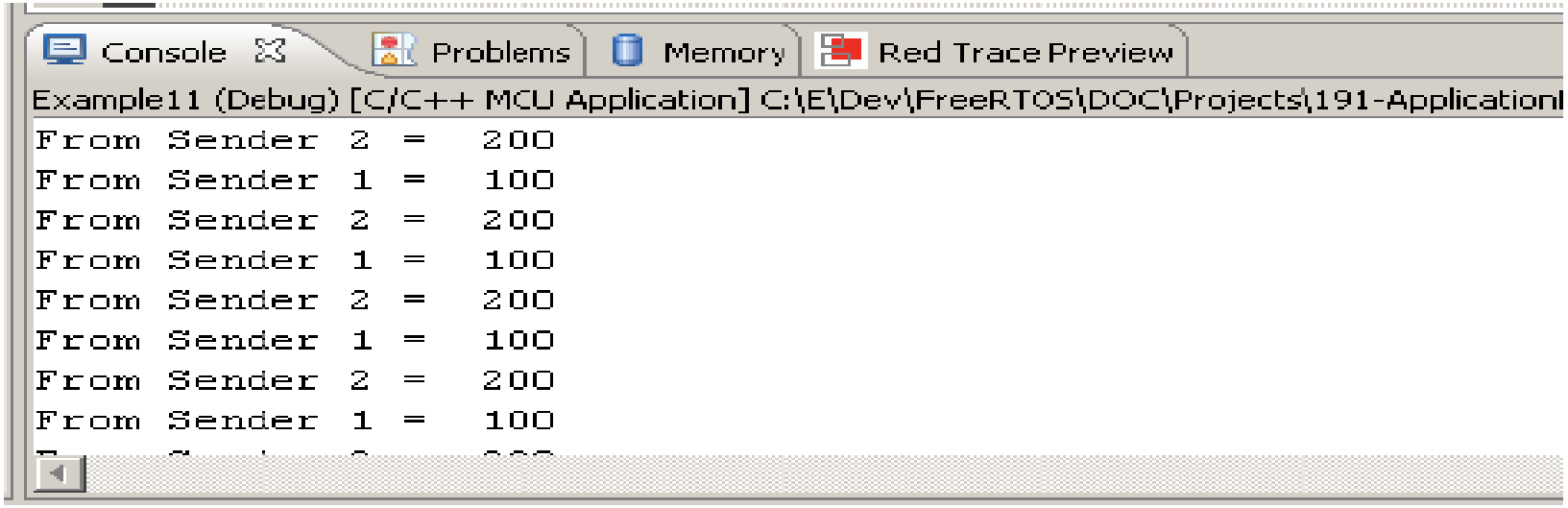


Figure 23. The output produced by Example 11

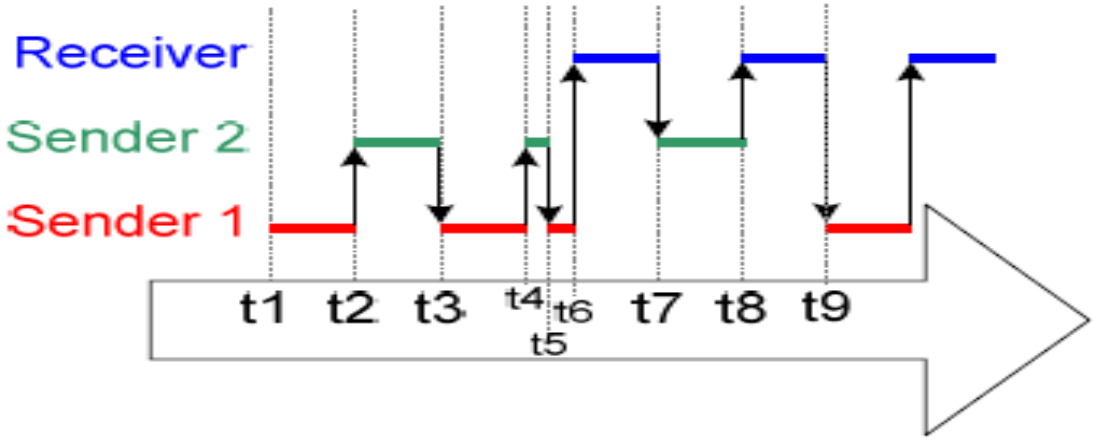


Figure 24. The sequence of execution produced by Example 11

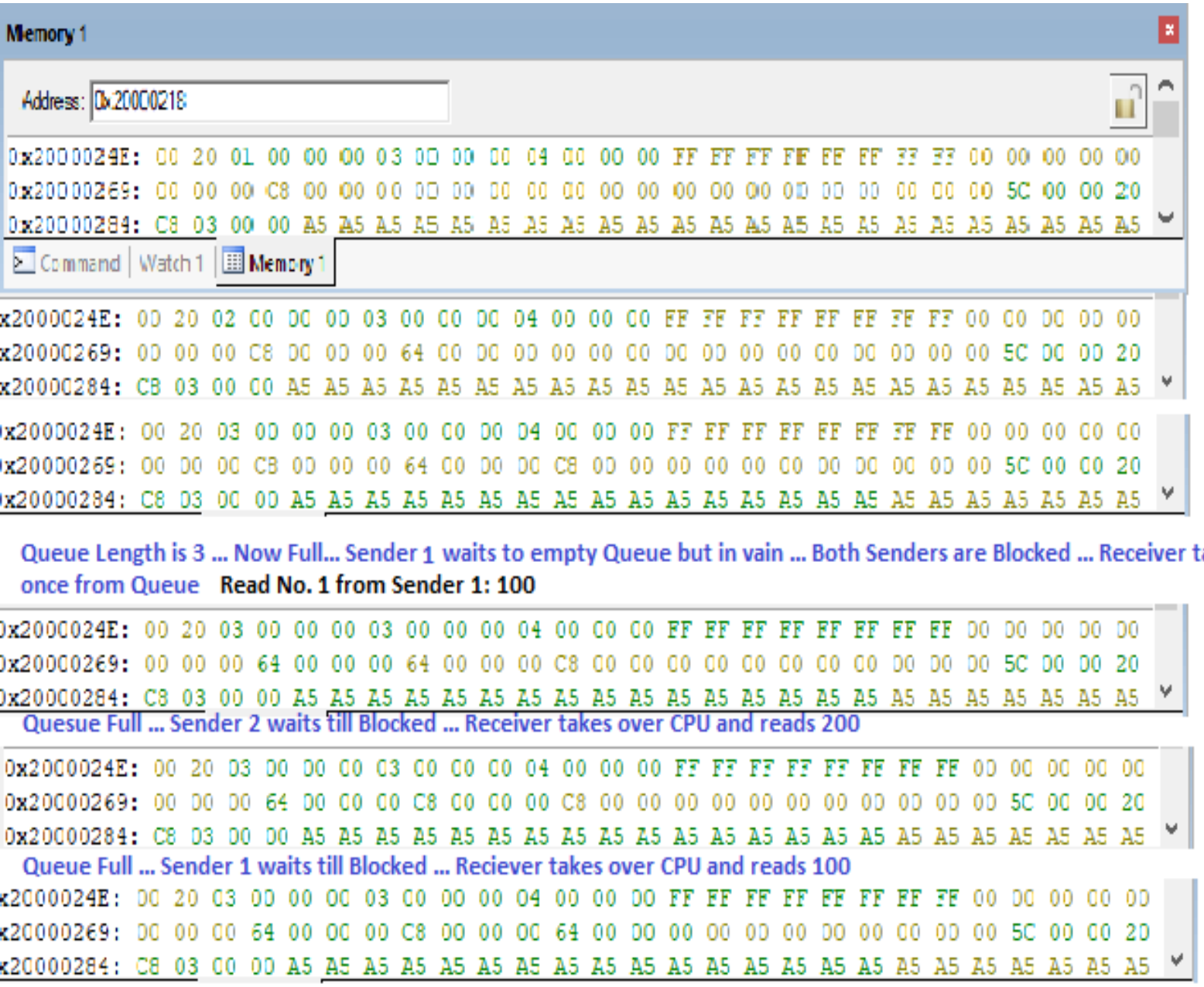
Example 11. Blocking when sending to a queue or sending structures on a queue

Time	Description
t1	Task Sender 1 executes and sends data to the queue.
t2	Sender 1 yields to Sender 2. Sender 2 writes data to the queue.
t3	Sender 2 yields back to Sender 1. Sender 1 writes data to the queue, making the queue full.
t4	Sender 1 yields to Sender 2.
t5	Sender 2 attempts to write data to the queue. Because the queue is already full, Sender 2 enters the Blocked state to wait for space to become available, allowing Sender 1 to execute once more.
t6	Sender 1 attempts to write data to the queue. Because the queue is already full, Sender 1 also enters the Blocked state to wait for space to become available. Now both Sender 1 and Sender 2 are in the Blocked state, so the lower priority Receiver task can execute.

Example 11. Blocking when sending to a queue or sending structures on a queue

Time	Description
t7	The receiver task removes an item from the queue. As soon as there is space on the queue, Sender 2 leaves the Blocked state and, as the higher priority task, pre-empts the Receiver task. Sender 2 writes to the queue, filling the space just created by the Receiver task. The queue is now full again. Sender 2 calls taskYIELD() but Sender 1 is still in the Blocked state, so Sender 2 is reselected as the Running state task and continues to execute.
t8	Sender 2 attempts to write to the queue. The queue is already full, so Sender 2 enters the Blocked state. Once again, both Sender 1 and Sender 2 are in the Blocked state, so the Receiver task can execute.
t9	The Receiver task removes an item from the queue. As soon as there is space on the queue, Sender 1 leaves the Blocked state and, as the higher priority task, pre-empts the Receiver task. Sender 1 writes to the queue, filling the space just created by the Receiver task. The queue is now full again. Sender 1 calls taskYIELD() but Sender 2 is still in the Blocked state, so Sender 1 is reselected as the Running state task and continues to execute. Sender 1 attempts to write to the queue but the queue is full, so Sender 1 enters the Blocked state. Both Sender 1 and Sender 2 are again in the Blocked state, allowing the lower priority Receiver task to execute.

Example 11. Blocking when sending to a queue or sending structures on a queue



Sender 2: Write No. 1 and Yields to Sender 1

Sender 1: Write No. 2 and Yields to Sender 2

Sender 2: Write No. 3 and Yields to Sender 1

Queue Length is 3 ... Now Full... Sender 1 waits to empty Queue but in vain ... Both Senders are Blocked ... Receiver takes over CPU and reads once from Queue **Read No. 1 from Sender 1: 100**

Sender 1: Write No. 4 and Yields to Sender 2

Quesue Full ... Sender 2 waits till Blocked ... Receiver takes over CPU and reads 200

Sender 2: Write No. 5 and Yields to Sender 1

Queue Full ... Sender 1 waits till Blocked ... Reciever takes over CPU and reads 100

Sender 1: Write No. 6 and Yields to Sender 2