Real Time Operating System "FreeRTOS" QUEUE; Task to Task Communication

Sherif Hammad

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books)
by Richard Barry

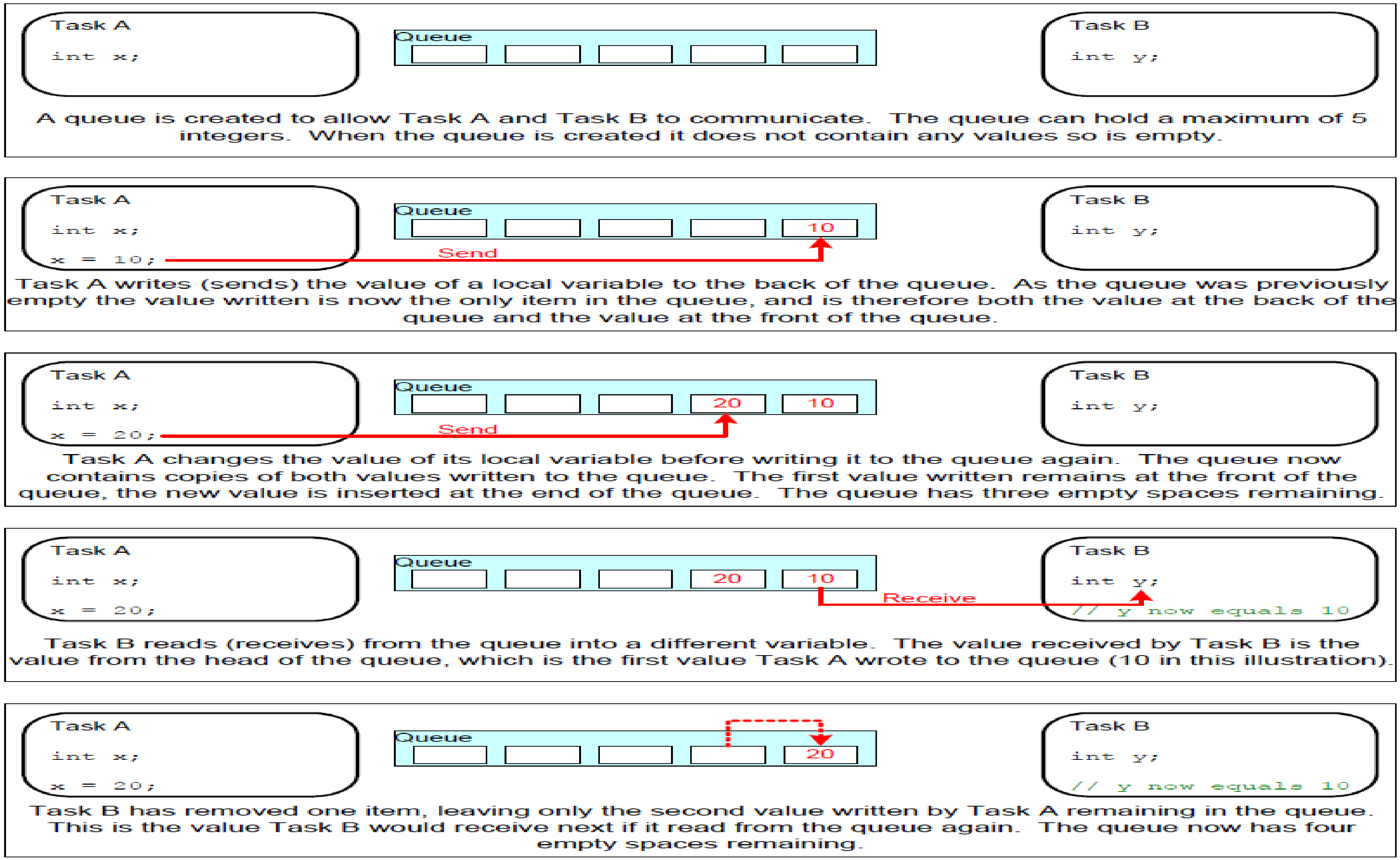
Agenda

- What is a queue?
- 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.

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 to a queue is a “physical append” to end of the queue.
- Reading from a queue is a “logical remove” from the top of the queue.

What is Queue?



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.

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.

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.
- 'block' time 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. */
```

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();
    }
}
```

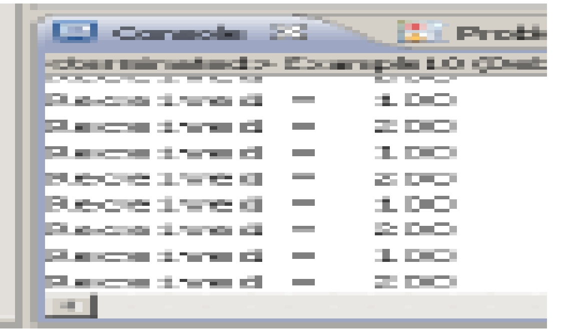
Listing 35. Implementation of the sending task used in Example 10

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

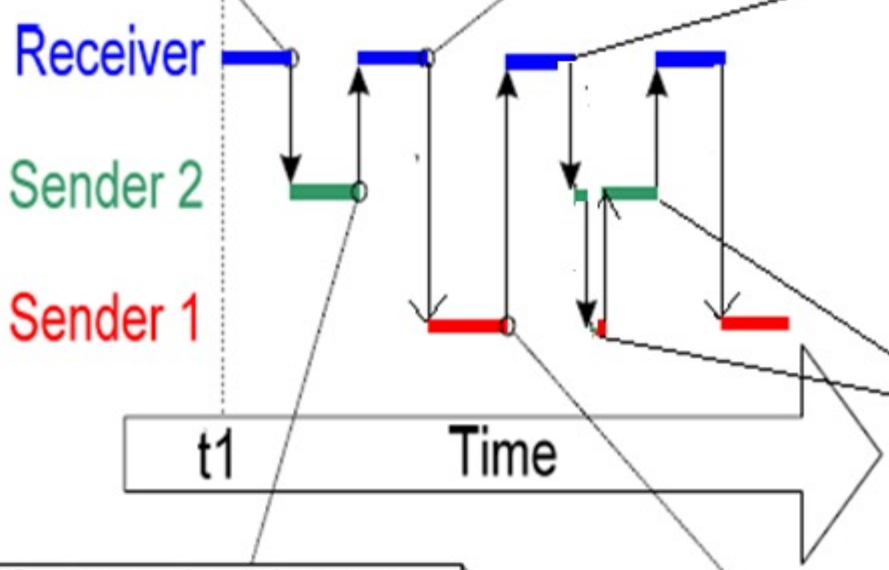


```
Console
<terminated> Example10 (Debug
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
```

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

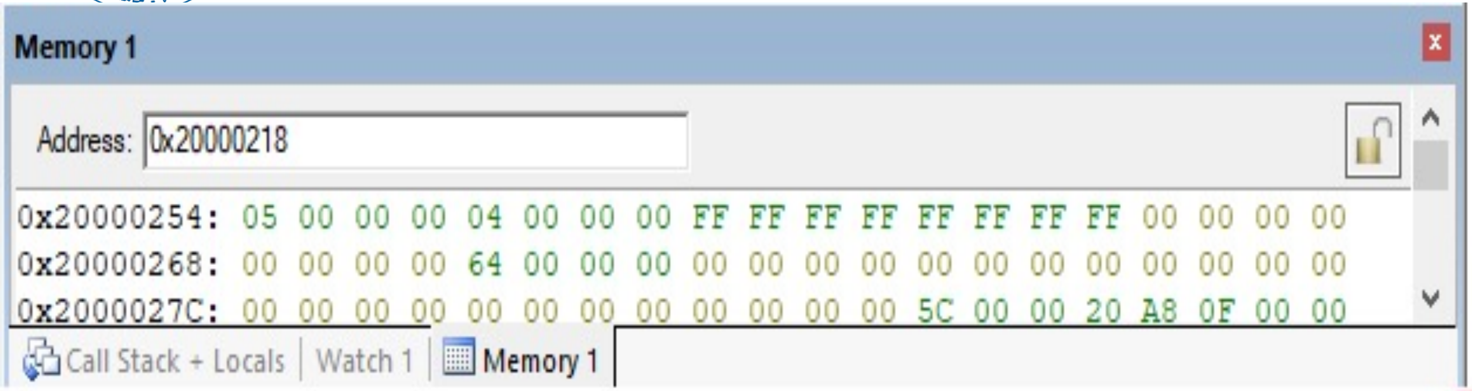


6&7- Prempeted Sender 2 completes and Yields and so Sender 1. Receiever is Blocked. So Sender 2 writes to the empty queue before preempted again

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



Memory 1

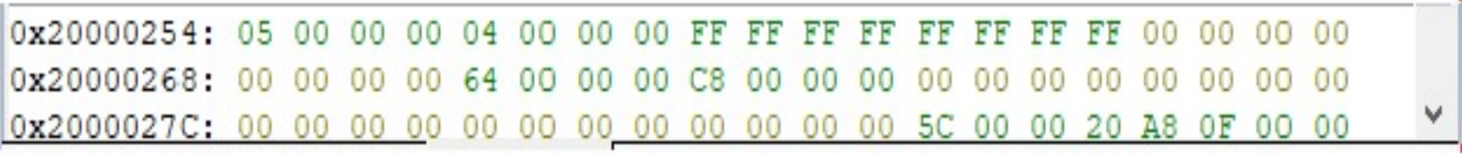
Address: 0x20000218

```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 64 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x2000027C: 00 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Call Stack + Locals | Watch 1 | Memory 1

Sender 1: Write No. 1

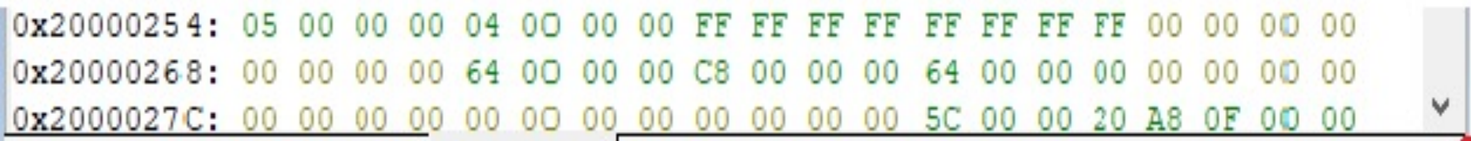
Receiver reads 64 just after



```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 64 00 00 00 C8 00 00 00 00 00 00 00 00 00 00 00
0x2000027C: 00 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Sender 2: Write No. 2

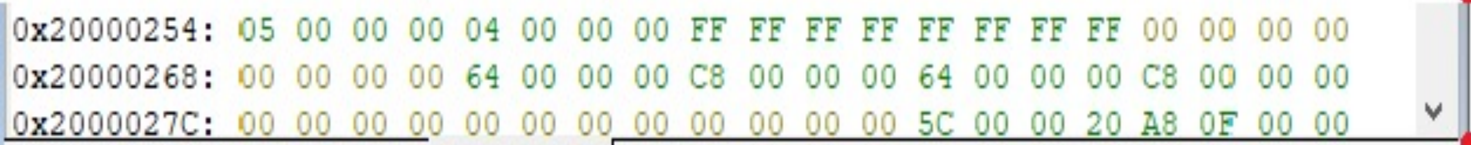
Receiver reads C8 just after



```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 64 00 00 00 C8 00 00 00 64 00 00 00 00 00 00 00
0x2000027C: 00 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Sender 1: Write No. 3

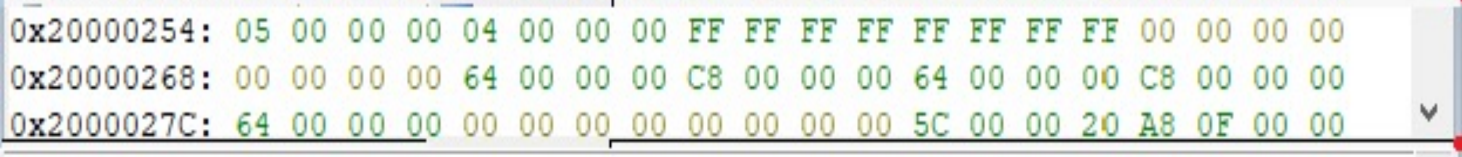
Receiver reads 64 just after



```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 64 00 00 00 C8 00 00 00 64 00 00 00 C8 00 00 00
0x2000027C: 00 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Sender 2: Write No. 4

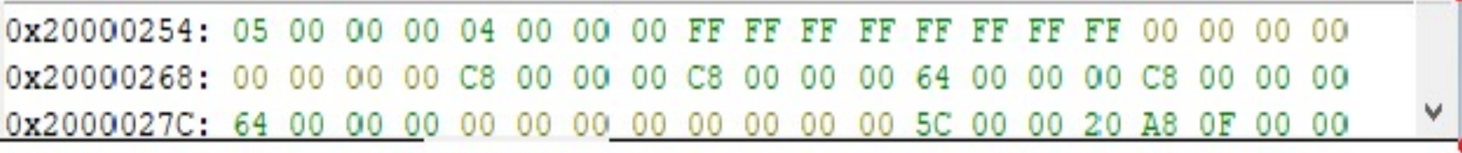
Receiver reads C8 just after



```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 64 00 00 00 C8 00 00 00 64 00 00 00 C8 00 00 00
0x2000027C: 64 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Sender 1: Write No. 5

Receiver reads 64 just after



```
0x20000254: 05 00 00 00 04 00 00 00 FF FF FF FF FF FF FF FF 00 00 00 00
0x20000268: 00 00 00 00 C8 00 00 00 C8 00 00 00 64 00 00 00 C8 00 00 00
0x2000027C: 64 00 00 00 00 00 00 00 00 00 00 00 5C 00 00 20 A8 0F 00 00
```

Sender 2: Writes No. 6

Circularly

Receiver reads C8 just after