

Usando el Kernel de Tiempo Real FreeRTOS:

(Traducción de la Guía práctica por Richard Barry)

Capítulo 2: Manejo de Colas

2.1 Introducción y alcance de este capítulo:

Las aplicaciones que utilizan FreeRTOS se estructuran como un conjunto de tareas independientes – cada tarea es efectivamente un mini programa en sí misma. Es probable que esta colección de tareas autónomas tenga que comunicarse entre sí a fin de que colectivamente puedan proporcionar la funcionalidad del sistema. Las colas son utilizadas por todas las comunicaciones FreeRTOS y sincronización de mecanismos.

Alcance

Este capítulo tiene como objetivo dar a los lectores una buena comprensión de:

- Como crear una cola
- Como una cola gestiona los datos que contiene
- Como enviar datos a una cola
- Como recibir datos de una cola
- Lo que significa Bloquear en una cola
- El efecto que tiene sobre las prioridades de las tareas el escribir y leer de una cola

Solo la comunicación de tarea a tarea será tratada en este capítulo. La comunicación Tarea a interrupción e interrupción a tarea será tratada en capítulo 3.

2.2 Características de una cola

Almacenamiento de datos

Una cola puede contener un número finito de elementos de datos de tamaño fijo. El número máximo de elementos que una cola puede contener se llama su "longitud". Tanto la longitud y el tamaño de cada elemento de datos se establecen cuando la cola es creada.

Normalmente las colas son usadas como buffers primero en entrar-primero en salir (FIFO) donde los datos se escriben en el final de la cola y se retira de la parte frontal (cabeza) de la cola. También es posible escribir en la parte delantera de una Cola.

La escritura de datos a una cola se lleva a cabo copiando byte por byte de los datos a ser almacenados en la cola en sí misma.

La lectura de datos de una cola provoca una copia de los datos que se eliminan de la cola. Figura 19 muestra datos siendo escritos hacia y leídos desde una cola, y el efecto sobre los datos almacenados en la cola en cada operación.

Acceso por múltiples tareas

Las colas son objetos en sí mismos, no pertenecen ni están asignados a ninguna tarea en particular. Múltiples tareas pueden escribir en la misma cola y leer de la misma cola. Que una cola tenga múltiples escritores es muy común mientras que una cola tenga múltiples lectores es poco común.

El bloqueo en las lecturas de las colas

Cuando una tarea intenta leer de una cola, opcionalmente puede especificar un tiempo de "bloqueo". Este es el tiempo en que la tarea debe mantenerse en el estado Bloqueado esperando a que los datos estén disponibles en la cola (la cola está vacía).

Una tarea que está en el estado Bloqueado esperando a que los datos estén disponibles en la cola pasa automáticamente a estado "listo" cuando otra tarea o interrupción coloca datos en la cola. La tarea también se moverá automáticamente desde el estado bloqueado al estado Listo si el tiempo de bloqueo especificado termina antes que el dato se ponga disponible.

Las colas pueden tener múltiples lectores así que es posible que una sola cola tenga más de una tarea bloqueada esperando por los datos. Cuando este es el caso solo una tarea será desbloqueada cuando el dato se ponga disponible. La tarea que se desbloquea será siempre la de mayor prioridad que estaba esperando por los datos. Si las tareas bloqueadas tienen igual prioridad, entonces la tarea que ha estado esperando más tiempo por el dato será la que se desbloquea.

El bloqueo en las escrituras de la cola

Así como cuando se lee de una cola, una tarea opcionalmente puede especificar un tiempo de bloqueo cuando se escribe en una cola. En este caso el tiempo de bloqueo es el tiempo máximo que la tarea debe ser retenida en el estado bloqueado mientras espera a que el espacio esté disponible en la cola (la cola está llena).

Las colas pueden tener múltiples escritores por lo que es posible que una cola llena tenga más de una tarea bloqueada a la espera de poder completar una operación de envío. Cuando este sea el caso sólo una tarea se desbloqueará cuando haya espacio disponible en la cola. La tarea que se desbloquea siempre será la tarea más alta prioridad que estaba esperando por el espacio. Si las tareas bloqueadas tienen la misma prioridad entonces se desbloqueará la tarea que más tiempo ha estado esperando a que se libere el espacio.

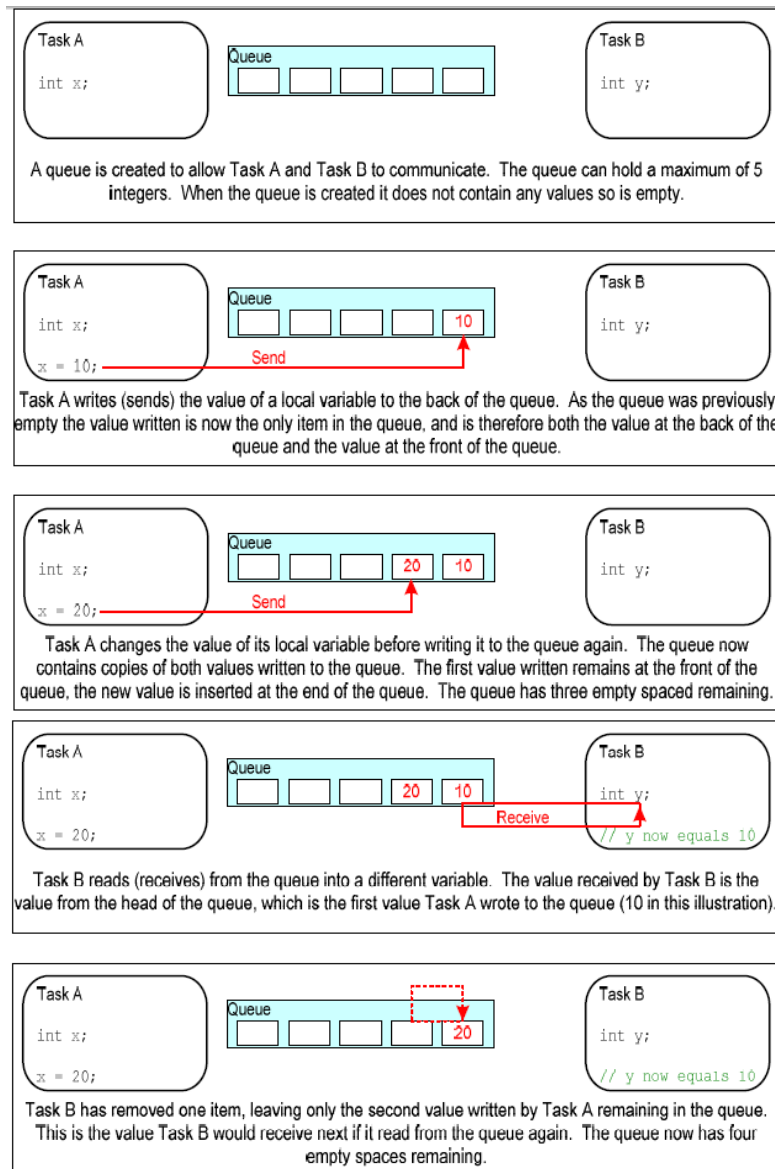


Figura 19 - Un ejemplo de secuencia de lectura y escrituras hacia y desde una cola

1. Una cola es creada para permitir la comunicación entre la tarea A y B. La cola puede almacenar un máximo de 5 enteros. Cuando la cola es creada está vacía.
2. La tarea A escribe (envía) el valor de una variable local al fondo de la cola. Como la cola estaba vacía el valor escrito es ahora el único ítem en la cola, y por lo tanto es el valor del principio de la cola y el valor del final de la cola.
3. La tarea A cambia el valor de su variable local antes de volver a escribir en la cola de nuevo. La cola ahora contiene las copias de los dos valores escritos en ella. El primer valor escrito se mantiene en el frente de la cola, el nuevo valor es insertado en el final de la cola. La cola tiene 3 lugares vacíos restantes.

Usando el Kernel de Tiempo Real FreeRTOS – Cap 2 / Manejo de Colas

4. La tarea B lee (recibe) desde la cola a una variable diferente. El valor recibido por la tarea B es el valor de la cabeza de la cola, el cual es el primer valor que la tarea A escribió en la cola (10).

5. La tarea B ha removido un ítem, dejando solo el segundo valor escrito por la tarea A en la cola. Este es el valor que la tarea B recibirá si realiza otra lectura. La cola ahora tiene cuatro espacios vacíos restantes.

2.3 Usando una cola

Función API xQueueCreate():

Una cola debe ser explícitamente creada antes de ser usada. Las colas son referenciadas usando variables de tipo xQueueHandle. xQueueCreate() es usada para crear una cola y devuelve un xQueueHandle para referenciar a la cola creada.

FreeRTOS asigna RAM de la pila FreeRTOS cuando se crea una cola. La memoria RAM se utiliza para mantener tanto las estructuras de datos de colas y los objetos que se encuentran en la cola. xQueueCreate() devolverá NULL si no hay memoria RAM suficiente disponible para la cola que se cree. El CAPÍTULO 5 ofrece más información sobre la gestión de memoria.

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

Listado 29 – Prototipo de la función API xQueueCreate ().

Tabla 7 xQueueCreate () parámetros y valores de retorno

Nombre del Parámetro / Valor de Retorno	Descripción
uxQueueLength	El máximo número de ítems que la cola puede contener en cualquier momento.
uxItemSize	El tamaño en bytes de cada elemento de datos que pueden ser almacenados en la cola.
Return Value	Si se devuelve NULL la cola no se ha podido crear porque no había suficiente memoria disponible de FreeRTOS para asignar los datos y estructura de la cola. Un valor no nulo(non-NULL) indica que la cola se creó exitosamente. El valor devuelto debe ser almacenado como el handle a la cola creada.

Función API xQueueSendToBack() and xQueueSendToFront():

Como era de esperar, xQueueSendToBack() se utiliza para enviar datos a la parte posterior (cola) de una cola, y xQueueSendToFront() se utiliza para enviar datos a la parte frontal (cabeza) de una cola.

xQueueSend() es equivalente a y exactamente el mismo que xQueueSendToBack().

Nunca llame xQueueSendToFront() o xQueueSendToBack() desde una rutina de servicio de interrupción. Las versiones seguras de interrupción xQueueSendToFrontFromISR() y xQueueSendToBackFromISR() deberían ser utilizadas en su lugar. Estos se describen en el capítulo 3.

```
portBASE_TYPE xQueueSendToFront( xQueueHandle xQueue,
                                 const void * pVItemToQueue,
                                 portTickType xTicksToWait
```

);

Listado 30 - Prototipo de la función API The xQueueSendToFront()

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

Listado 31 - Prototipo de la función API The xQueueSendToBack()

Tabla 8 xQueueSendToFront() y xQueueSendToBack() parámetros y valores de retorno

Nombre del Parámetro / Valor de Retorno	Descripción
xQueue	El Handle de la cola a la que se está enviando los datos. El Handle de la cola es el que fue retornado por xQueueCreate () cuando se creó la cola.
pvItemToQueue	Un puntero a los datos que se copiarán en la cola. El tamaño de cada elemento que la cola puede contener se establece cuando se crea la cola, por lo que esta cantidad de bytes se copiará desde pvItemToQueue a el área de almacenamiento de la cola.
xTicksToWait	La cantidad máxima de tiempo que la tarea debe permanecer en el estado bloqueado esperando a que el espacio esté disponible en la cola, si la cola ya está llena. Tanto xQueueSendToFront() y xQueueSendToBack() devolverán inmediatamente si xTicksToWait es 0 y la cola ya está llena. El tiempo de bloque se especifica en Ticks por lo que el tiempo absoluto depende de la frecuencia. La constante portTICK_RATE_ puede ser utilizada para convertir un tiempo especificado en milisegundos a un tiempo especificado en Ticks. Setear xTicksToWait a portMAX_DELAY hará que la tarea espere indefinidamente, siempre que INCLUDE_vTaskSuspend este seteado en 1 en FreeRTOSConfig.h.
Returned value	Hay dos posibles valores de retorno: 1. pdPASS -pdPASS sólo se devuelve si los datos fueron enviados exitosamente a la cola. -Si se especifica un tiempo de bloque (xTicksToWait distinto de cero), entonces es posible que la tarea se haya Bloqueado a esperar a que el espacio este disponible en la cola antes de la función devuelva - pero los datos se escribieron correctamente en la cola antes de que expirara el tiempo de bloque. 2. errQUEUE_FULL

	<p>errQUEUE_FULL será devuelto si los datos no pudieron ser escritos en la cola porque la cola ya estaba llena.</p> <p>Si se especificó un tiempo de bloqueo (xTicksToWait distinto de cero), entonces la tarea se habrá colocado en el estado Bloqueado a esperar que otra tarea o interrupción hagan espacio en la cola, pero el tiempo de bloqueo especificado ha caducado antes de que esto ocurriera.</p>
--	--

Funcion API xQueueReceive() y xQueuePeek() :

xQueueReceive () se utiliza para recibir (leer) un elemento de una cola. El elemento que se recibe se elimina de la cola.

xQueuePeek () se utiliza para recibir un elemento de una cola sin que el elemento se elimine de la cola. xQueuePeek () recibirá el elemento de la cabeza de la cola sin modificar los datos que se almacena en la cola, o el orden en que los datos se almacenan en la cola.

Nunca llame xQueueReceive () o xQueuePeek () desde una rutina de servicio de interrupción. La función api de interrupción xQueueReceiveFromISR () se describe en el Capítulo 3.

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

Figura 20 - Prototipo de la función API xQueueReceive()

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

Listado 32 - Prototipo de la función API xQueuePeek()

Tabla 9 xQueueReceive() y xQueuePeek() parámetros y valores de retorno

Nombre del Parámetro / Valor de Retorno	Descripción
xQueue	El Handle de la cola desde la cual se están recibiendo los datos (leyendo). El Handle de la cola es el que fue retornado por xQueueCreate () cuando se creó la cola.
pvBuffer	Un puntero a la memoria en la que se copiarán los datos recibidos. El tamaño de cada elemento de datos que la cola contiene se establece cuando se crea la cola. La memoria a la que apunta pvBuffer debe ser al menos lo suficientemente grande como para sostener esos bytes.

xTicksToWait	<p>La cantidad máxima de tiempo que la tarea debe permanecer en el estado bloqueado esperando a que el dato esté disponible en la cola, si la cola está vacía.</p> <p>Tanto xQueueSendToFront() y xQueueSendToBack() devolverán inmediatamente si xTicksToWait es 0 y la cola ya está llena.</p> <p>El tiempo de bloque se especifica en Ticks por lo que el tiempo absoluto depende de la frecuencia. La constante portTICK_RATE_ puede ser utilizada para convertir un tiempo especificado en milisegundos a un tiempo especificado en Ticks.</p> <p>Setear xTicksToWait a portMAX_DELAY hará que la tarea espere indefinidamente, siempre que INCLUDE_vTaskSuspend este seteado en 1 en FreeRTOSConfig.h.</p>
Returned value	<p>Hay dos posibles valores de retorno:</p> <p>1. pdPASS -pdPASS sólo se devuelve si los datos fueron leídos exitosamente de la cola.</p> <p>-Si se especifica un tiempo de bloque (xTicksToWait distinto de cero), entonces es posible que la tarea se haya Bloqueado a que el dato esté disponible en la cola - pero los datos se leyeron correctamente de cola antes de que expirara el tiempo de bloque.</p> <p>2. errQUEUE_FULL errQUEUE_FULL será devuelto si los datos no pudieron ser leídos en la cola porque la cola estaba vacía.</p> <p>Si se especifico un tiempo de bloqueo (xTicksToWait distinto de cero), entonces la tarea se habrá colocado en el estado Bloqueado a esperar que otra tarea o interrupción envíen data a la cola, pero el tiempo de bloqueo especificado ha caducado antes de que esto ocurriera.</p>

Funcion API uxQueueMessagesWaiting():

uxQueueMessagesWaiting () se utiliza para consultar el número de elementos que se encuentran actualmente en una cola.

Nunca llame uxQueueMessagesWaiting () desde una rutina de servicio de interrupción. uxQueueMessagesWaitingFromISR se debe utilizar en su lugar.

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

Listado 33 - Prototipo de la función API uxQueueMessagesWaiting()

Tabla 10 uxQueueMessagesWaiting() parámetros y valores de retorno

Nombre del Parámetro / Valor de Retorno	Descripción
---	-------------

xQueue	El Handle de la cola que se está consultando. El Handle de la cola que retorno xQueueCreate () cuando se creó la cola.
Returned value	El número de elementos que la cola consultada contiene. Si se devuelve 0 entonces la cola está vacía.

Ejemplo 10. El bloqueo al recibir de una cola

Este ejemplo demuestra la creación de una cola, los datos que se envían a la cola desde múltiples tareas, y los datos que se reciben de la cola. Se crea la cola para mantener los elementos de datos de tipo long. Las tareas que envían a la cola no especifican un tiempo de bloqueo, mientras que la tarea que recibe de la cola sí lo hace.

La prioridad de las tareas que envían a la cola es menor que la prioridad de la tarea que recibe de la cola. Esto significa que la cola no debe contener más de un artículo, porque tan pronto como los datos se envían a la cola la tarea que recibe se desbloqueará, adelantarse a la tarea de enviar, y quitar los datos – dejando la cola vacía de nuevo.

El Listado 34 muestra la implementación de la tarea que escribe en la cola. Se crean dos instancias de esta tarea, uno que escribe continuamente el valor 100 a la cola, y otra que escribe continuamente el valor 200 a la misma cola. El parámetro de tarea se utiliza para pasar estos valores en cada instancia de tarea.

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

    /* Se crean dos instancias de esta tarea por lo que el valor que se envía a la cola se hace pasar a través del parámetro en tarea - de esta manera cada instancia puede utilizar un valor diferente. La cola fue creada para contener valores de tipo largo. */
    lValueToSend = ( long ) pvParameters;

    /* Como la mayoría de tareas, esta tarea se lleva a cabo dentro de un bucle infinito.*/
    for( ;; )
    {
        /* Enviar el valor a la cola.
        El primer parámetro es la cola a la que se está enviando datos. La cola fue creada antes de que el scheduler inicie, así que antes de que esta tarea comencé a ejecutarse.
        El segundo parámetro es la dirección de los datos a enviar, en este caso la dirección de lValueToSend.
        El tercer parámetro es el tiempo de Bloqueo - el tiempo que la tarea debe mantenerse en el estado Bloqueado de esperar a que el espacio esté disponible en la cola (la cola debe ya estar llena). En este caso no se especifica un tiempo de bloqueo porque la cola no debe contener más de un elemento, por lo que nunca será completa. */
        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
        if( xStatus != pdPASS )
        {
            /* La operación de envío no pudo completar debido a que la cola estaba llena - esto debe ser un error ya que la cola nunca debe contener más de un elemento!*/
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```



```

        /* Permite que la otra tarea remitente a ejecutar. taskYIELD () informa al programador que
        un cambio a otra tarea debe ocurrir ahora en lugar de mantener esta tarea en estado de
        ejecución hasta el final de la porción de tiempo actual. */
        taskYIELD();
    }
}

```

Listado 34 - Implementación de la tarea de envió usada en el ejemplo 10.

El Listado 35 muestra la implementación de una tarea que recibe los datos de una cola. La tarea de recibir especifica un tiempo de bloqueo de 100 milisegundos por lo que entrará en el estado Bloqueado para esperar para que los datos estén disponibles. Se dejará el estado bloqueado cuando cualquier dato esté disponible en la cola, o 100 milisegundos pasen sin que haya datos disponibles. En este ejemplo, el tiempo de espera de 100 milisegundos nunca debe expirar, ya que hay dos tareas que están escribiendo continuamente a la cola.

```

static void vReceiverTask( void *pvParameters )
{
    /* Declaración de la variable que contendrá los valores recibidos de la cola. */
    long lReceivedValue;
    portBASE_TYPE xStatus;
    const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* Esta tarea también se define dentro de un bucle infinito. */
    for( ;; )
    {
        /* Esta llamada debe siempre encontrar la cola vacía porque esta tarea eliminará de inmediato los datos que se escriben en la cola.*/
        if( uxQueueMessagesWaiting( xQueue ) != 0 )
        {
            vPrintString( "Queue should have been empty!\r\n" );
        }
        /* Recibir datos de la cola.
        El primer parámetro es la cola desde la que los datos se van a recibir. Se crea la cola antes de que se inicie el scheduler, y por lo tanto antes de que esta tarea se ejecuta por primera vez.
        El segundo parámetro es el buffer en el que se colocarán los datos recibidos. En este caso, el búfer es simplemente la dirección de una variable que tiene el tamaño necesario para contener los datos recibidos.
        El último parámetro es el tiempo de bloqueo - la cantidad máxima de tiempo que la tarea debe permanecer en el estado bloqueado para esperar a los datos estén disponibles (si la cola ya está vacía). En este caso la constante portTICK_RATE_MS se utiliza para convertir 100 milisegundos a un tiempo especificado en ticks. */
        xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
        if( xStatus == pdPASS )
        {
            /* Los datos se recibieron con éxito de la cola, imprime el valor recibido.*/
            vPrintStringAndNumber( "Received = ", lReceivedValue );
        }
    }
}

```

```

        else
        {
            /* Los datos no se recibieron de la cola, incluso después de esperar 100ms.
            Este debe ser un error ya que las tareas de envío están corriendo libremente y van
            a escribir de forma continua a la cola. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}

```

Listado 35 - Implementación de la tarea de recepción usada en el ejemplo 10.

El Listado 36 contiene la definición de la función main(). Esto simplemente crea la cola y las tres tareas antes de iniciar el scheduler. Se crea la cola para contener un máximo de 5 valores long a pesar de que las prioridades de las tareas se establecen por lo que la cola nunca en realidad contienen más de un elemento a la vez.

/* Declarar una variable de tipo xQueueHandle. Esto se utiliza para almacenar la referencia a la cola que se accede por las tres tareas.*/

xQueueHandle xQueue;

int main(void)

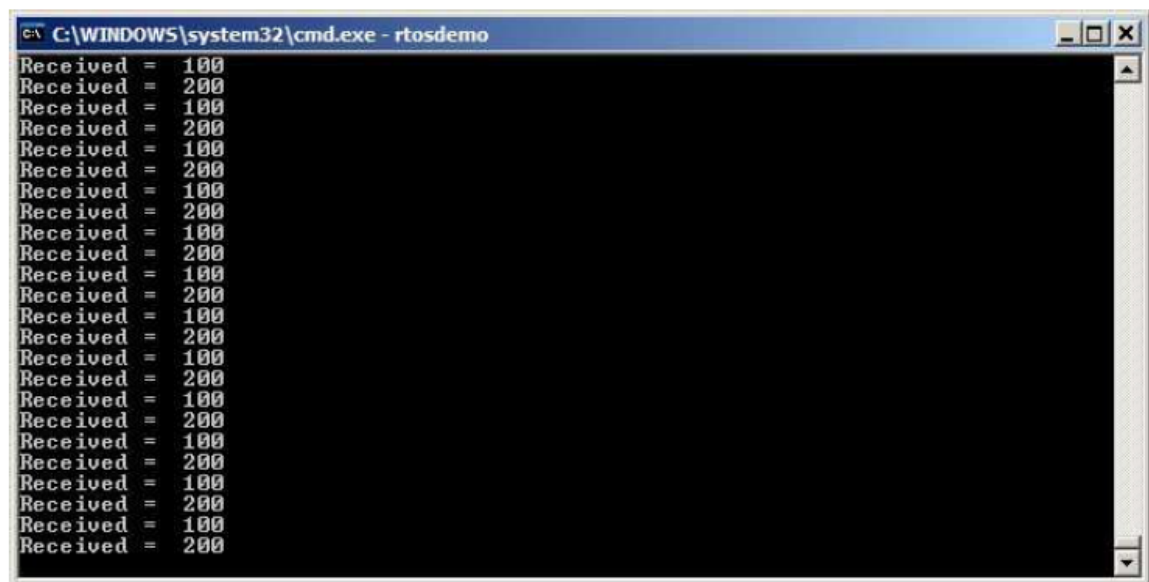
```

{
    /* Se crea la cola para contener un máximo de 5 valores, cada uno de los cuales es lo suficientemente grande como para contener una variable de tipo long.*/
    xQueue = xQueueCreate( 5, sizeof( long ) );
    if( xQueue != NULL )
    {
        /* Cree dos instancias de la tarea que va a enviar a la cola. El parámetro de tarea se utiliza para pasar el valor que la tarea va a escribir a la cola, por lo que una tarea continuamente va a escribir 100 a la cola mientras que la otra tarea va a escribir 200 a la cola. Ambas tareas se crean con prioridad 1.*/
        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );
        /* Crear la tarea que va a leer de la cola. La tarea se crea con prioridad 2, por lo que por encima de la prioridad de las tareas que envían. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );
        /* Inicie el scheduler para que las tareas creadas comiencen su ejecución. */
        vTaskStartScheduler();
    }
    else
    {
        /* La cola no pudo ser creada. */
    }
    /* Si todo está bien, entonces main () nunca va a llegar a aquí como el scheduler no estará ejecutando las tareas. Si main () llega aquí, entonces es probable que no había suficiente memoria disponible para la tarea ociosa que se creará.
    CAPÍTULO 5 ofrece más información sobre la gestión de memoria.*/
    for( ;; );
}

```

Listado 35 - Implementación del main del ejemplo 10.

Las tareas que envían a la cola llaman taskYIELD () en cada iteración de su bucle infinito. taskYIELD () informa al scheduler que un cambio a otra tarea debe ocurrir ahora en lugar de mantener la tarea en ejecución hasta el final de la porción de tiempo actual. Una tarea que llama a taskYIELD () es, en efecto, el voluntariado para ser eliminado del estado de ejecución. Como ambas tareas que envían a la cola tienen una prioridad idéntica, cada vez que uno llama taskYIELD () la otra comienza la ejecución - la tarea que llama taskYIELD () se mueve al estado Preparado como la otra tarea de enviar se trasladó al estado de ejecución. Esto hace que las dos tareas que envían lo haga cada una en un turno. La salida producida por el Ejemplo 10 se muestra en la Figura 21.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
Received = 100
Received = 200
```

Figura 21 - Muestra la salida producida por la ejecución del Ejemplo 10

La Figura 22 demuestra la secuencia de ejecución.

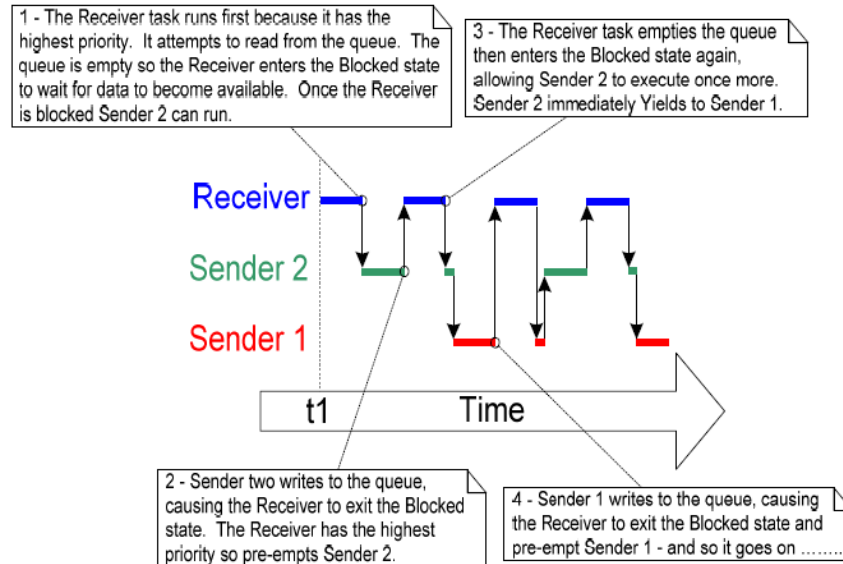


Figura 22 - Muestra la secuencia de ejecución producida por la ejecución del Ejemplo 10

1. La tarea que recibe se ejecuta primero porque tiene la mayor prioridad. Esta intenta leer la cola. La cola está vacía entonces la tarea se bloquea y espera por el dato. Una vez que el receptor es bloqueado la tarea que escribe 2 se ejecuta.
2. Escribe 2 escribe en la cola, provocando que el receptor salga del estado bloqueado. El receptor tiene mayor prioridad que la tarea que escribe 2.
3. El receptor vacía la cola y entra en estado bloqueado de nuevo, permitiendo a la tarea que envía 2 ejecutarse de nuevo. Envía 2 inmediatamente permite la ejecución de envío 1.
4. La tarea de envío 1 escribe en la cola, provocando que la tarea receptora salga del estado bloqueado y vuelva sender 1 y así...

Usando colas para transferir tipos compuestos

Es común para una tarea de recibir datos de múltiples fuentes en una sola cola. A menudo, el receptor de los datos tiene que saber de donde provienen los datos para poder determinar cómo deben ser procesados. Una manera sencilla de lograr esto es utilizar la cola para transferir estructuras en las que tanto el valor de los datos y la fuente de los datos están contenidos en los campos de la estructura. Este esquema se muestra en la Figura 23.

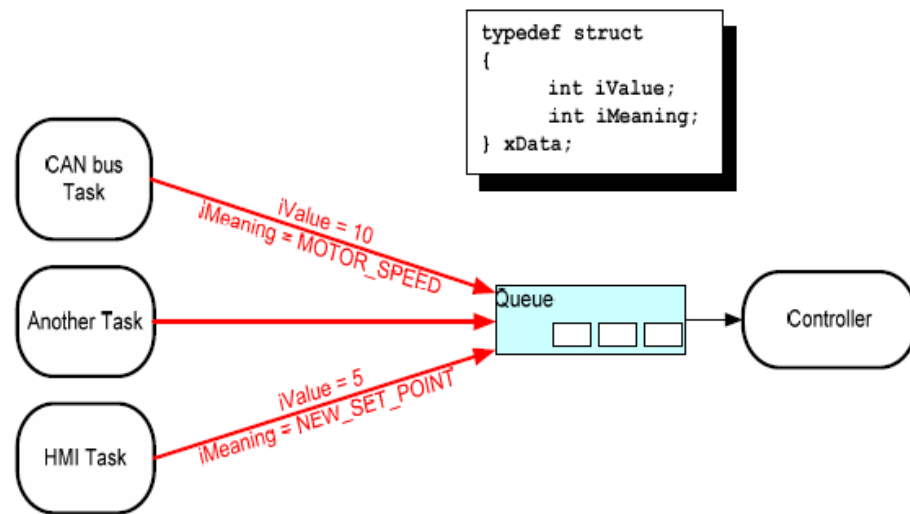


Figura 23 - Ejemplo de Escenario donde estructuras son enviadas a una cola

Refiriéndose a la Figura 23:

- Se crea una cola que tiene estructuras de tipo XDATA. Los miembros de la estructura permiten tanto un valor de datos y un código que indica el significado de los datos para ser enviados a la cola en un solo mensaje.
- Una tarea controladora central se utiliza para realizar la función del sistema primario. Esto tiene que reaccionar a las entradas y cambios en el estado del sistema que le sean comunicadas en la cola.
- Una tarea bus CAN se usa para encapsular la funcionalidad de interfaz de bus CAN. Cuando la tarea bus CAN ha recibido y decodificado un mensaje lo envía ya decodificado a la tarea controladora en una estructura de xData. El miembro iMeaning de la estructura transferida se utiliza para permitir que la tarea de control sepa lo que los datos son - en el caso representado se trata de una velocidad del motor. El miembro iValue de la estructura transferida se utiliza para que la tarea Control conozca el valor real de la velocidad del motor.
- Una tarea Human Machine Interface (HMI) se utiliza para encapsular toda la funcionalidad HMI. El operador de la máquina puede probablemente introducir comandos y consultar valores de diferentes maneras que tienen que ser detectadas e interpretadas entre la tarea HMI. Cuando un nuevo comando se introduce, la tarea HMI envía el comando a la tarea del controlador en una estructura de datos extendidos (xData). El miembro iMeaning de la estructura transferida se utiliza para permitir que la tarea de control sepa lo que los datos son - en el caso representado se trata de un nuevo valor del punto de ajuste. El miembro iValue de la estructura transferida se utiliza para que la tarea Control de conocer conozco el valor actual del punto de ajuste.

Ejemplo 11. El bloqueo al enviar a una cola / Envío de Estructuras en una cola

El Ejemplo 11 es similar a la del ejemplo 10, pero las prioridades de las tareas se invierten por lo que la tarea de recepción tiene una prioridad más baja que las tareas de envío. También la cola se utiliza para pasar estructuras entre las tareas en lugar de enteros largos simples.

Listado 37 muestra la definición de la estructura utilizada en el Ejemplo 11.

```
/* Definir el tipo de estructura que se pasará en la cola. */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;
/* Declare dos variables de tipo XDATA que se pasarán en la cola. */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Utilizado por Emisor1. */
    { 200, mainSENDER_2 } /* Utilizado por Emisor2. */
};
```

Listado 37 La definición de la estructura que se va a pasar en una cola, además de la declaración de dos variables para su uso para el ejemplo.

En el Ejemplo 10 la tarea de recibir tenía la más alta prioridad por lo que la cola no contenía más de un elemento. Esto se debía a que la tarea de recepción adelantaba a las tareas que enviaban tan pronto como los datos se colocan en la cola. En el Ejemplo 11 las tareas que envían tienen la mayor prioridad por lo que la cola normalmente estará llena. Esto se debe a que, tan pronto como la tarea de recepción elimina un elemento de la cola una de las tareas que envía inmediatamente a continuación, y se vuelve a llenar la cola. La tarea que envía volverá a entrar en el estado Bloqueado a la espera de que el espacio esté disponible en la cola de nuevo.

Listado 38 muestra la implementación de la tarea de envío. La tarea de enviar especifica un tiempo de bloqueo de 100 milisegundos (lo que estará en el estado Bloqueado esperando a que el espacio esté disponible cada vez que la cola se llena). Dejará el estado bloqueado cuando esté disponible el espacio en la cola o 100 milisegundos pasen sin espacio disponible. En este ejemplo, el tiempo de espera de 100 milisegundos no debe expirar por que la tarea de recepción está haciendo continuamente el espacio mediante la eliminación de los elementos de la cola.

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

```

/* como la mayoría de tareas, esta tarea se lleva a cabo dentro de un bucle infinito. */
for( ;; )
{
    /* Enviar a la cola.
    El segundo parámetro es la dirección de la estructura que se envía. La
    dirección se pasa como parámetro de tarea por lo que se utilizan pvParameters directamente.
    El tercer parámetro es el tiempo de Bloqueo - el tiempo de la tarea debe mantenerse en el estado
    Bloqueado que esperar a que el espacio esté disponible en la cola. Se especifica un tiempo de blo-
    queo porque las tareas que envían tienen una prioridad más alta que la tarea de recepción así que
    se espera que la cola se llene. La tarea de recepción será de ejecutar y eliminar elementos de la cola
    cuando ambas tareas envío están en el estado bloqueado. */
    xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );
    if( xStatus != pdPASS )
    {
        /* La operación de envío no se pudo completar, incluso después de esperar 100ms.
        Esto debe ser un error como la tarea de recepción debe hacer espacio en la cola tan pronto como
        ambas tareas que envían están en el estado Bloqueado.*/
        vPrintString( "Could not send to the queue.\r\n" );
    }
    /* Permite que la otra tarea enviar a ejecutar. */
    taskYIELD();
}
}

```

Listado 38 - La implementación de la tarea de enviar del Ejemplo 11.

La tarea de recibir tiene la prioridad más baja de modo que sólo se ejecutará cuando ambas tareas envío están en el estado bloqueado. Las tareas que envían solamente entrarán en el estado bloqueado cuando la cola está llena, por lo que la tarea de recepción sólo se ejecuta cuando la cola está llena. Por lo tanto, siempre espera para recibir datos, incluso sin tener que especificar un tiempo de bloqueo

La implementación de la tarea de recepción se muestra en el Listado 39.

```

static void vReceiverTask( void *pvParameters )
{

```

/ Declarar la estructura que contendrá los valores recibidos de la cola. */*

xData xReceivedStructure;

portBASE_TYPE xStatus;

/ Esta tarea también se define dentro de un lazo infinito. */*

for(;;)

{

/ Debido a que tiene la prioridad más baja esta tarea sólo se ejecutará cuando las tareas de envío están en el estado bloqueado. Las tareas que envían solamente entrará en el estado bloqueado cuando la cola está llena por lo que esta tarea siempre espera que el número de elementos en la cola para ser igual a la longitud de la cola - 3 en este caso. */**if(uxQueueMessagesWaiting(xQueue) != 3)*

{

vPrintString("Queue should have been full!\r\n");

}

/ Recibir de la cola.**El segundo parámetro es el buffer en el que se colocarán los datos recibidos. En este caso, el búfer es simplemente la dirección de una variable que tiene el tamaño requerido para sostener la estructura recibida.**El último parámetro es el tiempo bloqueo - la cantidad máxima de tiempo que la tarea se mantendrá en el estado bloqueado para esperar a los datos estén disponibles. En este caso, el tiempo de bloqueo no es necesario porque esta tarea sólo se ejecuta cuando la cola está llena. */**xStatus = xQueueReceive(xQueue, &xReceivedStructure, 0);**if(xStatus == pdPASS)*

{

/ Los datos se recibieron con éxito de la cola, imprima el valor recibido y la fuente del valor. */**if(xReceivedStructure.ucSource == mainSENDER_1)*

{

vPrintStringAndNumber("From Sender 1 = ", xReceivedStructure.ucValue);

}

else

{

vPrintStringAndNumber("From Sender 2 = ", xReceivedStructure.ucValue);

}

}

else

{

/ Nada se recibió de la cola. Este debe ser un error ya que esta tarea sólo se ejecuta cuando la cola está llena. */**vPrintString("Could not receive from the queue.\r\n");*

}

}

}

Listado 39 - La definición de la tarea de recepción para el Ejemplo 11

Usando el Kernel de Tiempo Real FreeRTOS – Cap 2 / Manejo de Colas

El main () cambia sólo ligeramente del ejemplo anterior. Se crea la cola para contener tres estructuras XDATA y las prioridades de las tareas de envío y recepción están invertidas. La ejecución de main () se muestra en el Listado 40.

```
int main( void )
{
    /* Se crea la cola para contener un máximo de 3 estructuras de tipo XDATA*/
    xQueue = xQueueCreate( 3, sizeof( xData ) );
    if( xQueue != NULL )
    {
        /* Cree dos instancias de la tarea que va a escribir a la cola. El parámetro se utiliza para
        transmitir la estructura que la tarea va a escribir a la cola, por lo que una tarea enviará con-
        tinuamente xStructsToSend [0] a la cola, mientras que la otra tarea enviará continuamente
        xStructsToSend [1]. Ambas tareas se crean en prioridad 2 que está por encima de la priori-
        dad del receptor. */
        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp;xStructsToSend[ 0 ]), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp;xStructsToSend[ 1 ]), 2, NULL );
        /* Crear la tarea que va a leer de la cola. La tarea se crea con prioridad 1, por lo que debajo
        de la prioridad de las tareas de escritura. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );
        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* La cola no pudo ser creada. */
    }
    /* Si todo está bien, entonces main () nunca va a llegar a aquí como el scheduler ahora va a estar
    ejecutando las tareas. Si main () sí llega aquí, entonces lo más probable es que no había memoria
    suficiente disponible para la tarea ociosa que se creará. CAPÍTULO 5 ofrece más información sobre
    la gestión de memoria. */
    for( ;; );
}
```

Listado 40 - La implementación de main() para el Ejemplo 11

Como en el ejemplo 10 las tareas que envían a la cola se alternan en cada iteración de su bucle infinito de modo se turnan para enviar datos a la cola. La salida producida por el Ejemplo 11 se muestra en la Figura 24.

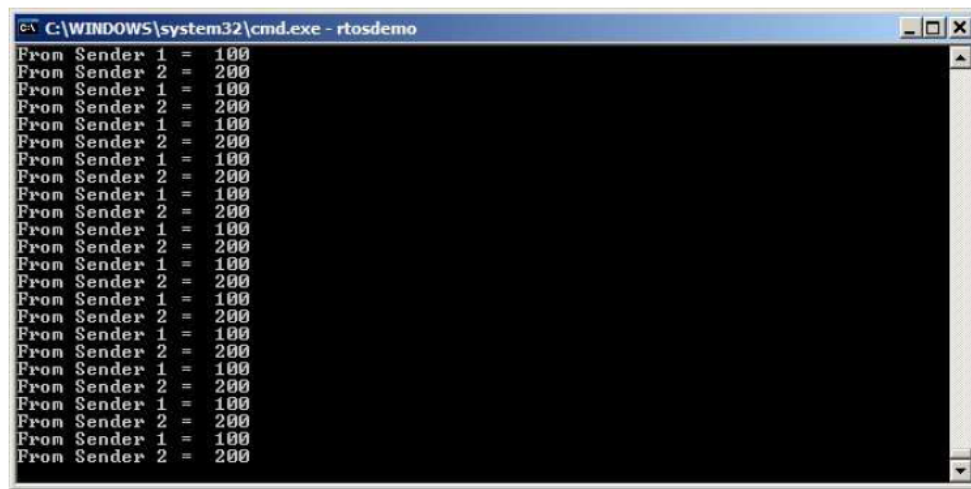


Figura 24 - La salida que produce el ejemplo 11

La Figura 25 demuestra la secuencia de ejecución que resulta de tener la prioridad de las tareas que envían por encima de la de la tarea de recepción. Una explicación más detallada de la figura 25 se proporciona dentro de la Tabla 12.

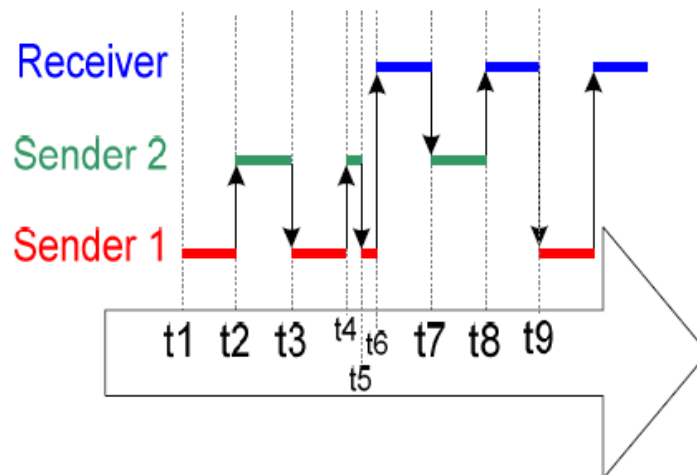


Figura 25 - La secuencia de ejecución producida por Ejemplo 11

Tabla 11 Descripción para la Figura 25

Tiempo	Descripción
T1	Tarea envío 1 ejecuta y envía datos a la cola.
T2	Envío 1 cede a envío 2. Envío 2 escribe datos a la cola.
T3	Envío 2 da vuelta al Envío 1. Envío 1 escribe datos en la cola que hacen la cola llena.
T4	Envío 1 cede a envío 2
T5	Envío 2 intenta de escribir datos a la cola. Debido a que la cola ya está llena de Envío 2 entra en el estado Bloqueado esperando a que el espacio esté disponible, lo que permite envío 1 para ejecutar una vez más.
T6	envío 1 intenta escribir datos en la cola. Debido a la cola ya está lleno envío 1 también entra en el estado Bloqueado que esperar a que el espacio esté disponible. Ahora, tanto del envío 1 y 2 están en estado bloqueado, por lo que la tarea de menor prioridad receptor puede ejecutarse.
T7	La tarea receptor elimina un elemento de la cola. Tan pronto como haya espacio en la cola el v1o 2 deja el estado bloqueado y, en su tarea de mayor prioridad, se adelanta la tarea del receptor. Envío 2 escribe a la cola, llenando el espacio recién creado por la tarea receptor. La cola está ahora llena de nuevo. Envío 2 llama taskYIELD () pero envío 1 se encuentra todavía en el estado bloqueado de manera envío 2 se vuelve a seleccionar como la tarea Estado listo y continúa ejecutando.
T8	envío 2 intenta escribir en la cola. La cola ya está llena, así envío 2 entra en el estado bloqueado. Una vez más, tanto del envío 1 y envío 2 están en el estado bloqueado por lo que la tarea del receptor puede ejecutarse.
T9	La tarea del receptor elimina un elemento de la cola. Tan pronto como haya espacio en la cola el Envío 1 deja el estado bloqueado y, en su tarea de mayor prioridad, se adelanta la tarea del receptor. envío 1 escribe a la cola, llenando el espacio recién creado por la tarea receptor. La cola está ahora llena de nuevo. Envío 1 llama taskYIELD () pero Envío 2 todavía está en el estado bloqueado de manera Envío 1 se vuelve a seleccionar como la tarea Estado lista y continúa ejecutando. Envío 1 intenta escribir en la cola, pero la cola está llena de modo Envío 1 entra en el estado bloqueado. Tanto Envío 1 y 2 del Envío están de nuevo en el estado bloqueado, lo que permite la tarea menor Receptor prioridad a ejecutar.

2.4 TRABAJANDO CON DATOS GRANDES

Si el tamaño de los datos que se almacena en la cola es grande, entonces es preferible usar la cola para transferir punteros a los datos en lugar de copiar los datos en sí dentro y fuera de la cola byte a byte. Transferencia de punteros es más eficiente, tanto en tiempo de procesamiento y la cantidad de RAM necesaria para crear la cola. Sin embargo, cuando las colas son de punteros se debe tener cuidado para asegurar que:

1. El propietario de la RAM siendo apuntado sea claramente definido.

Al compartir la memoria entre las tareas a través de un puntero es esencial para asegurar que ambas tareas no modifiquen los contenidos de la memoria al mismo tiempo, o tomar cualquier otra acción

Usando el Kernel de Tiempo Real FreeRTOS – Cap 2 / Manejo de Colas

que podría hacer que el contenido de la memoria pase ser inválida o inconsistente. Lo ideal sería que solamente la tarea de enviar tenga permitido acceder a la memoria hasta que un puntero a la memoria haya sido cargado en cola, y sólo la tarea de recepción debería permitir acceder a la memoria después de que el puntero se ha recibido de la cola.

2. La memoria RAM siendo apuntada sigue siendo válida

Si la memoria siendo apuntada fue asignada dinámicamente entonces exactamente una tarea debe ser responsable de liberar la memoria. Ninguna tareas debería tratar de acceder a la memoria después de que haya sido liberada.

Un puntero nunca debe ser usado para acceder a datos que han sido asignados en una pila de tareas. Los datos no serán válidos después de que la pila ha cambiado.