FreeRTOS – Part 2

Da-Wei Chang

Sources:

- *The freeRTOS website, https://www.freertos.org/*
- *Cortex-M3 Devices Generic User Guide, ARM 2010.*

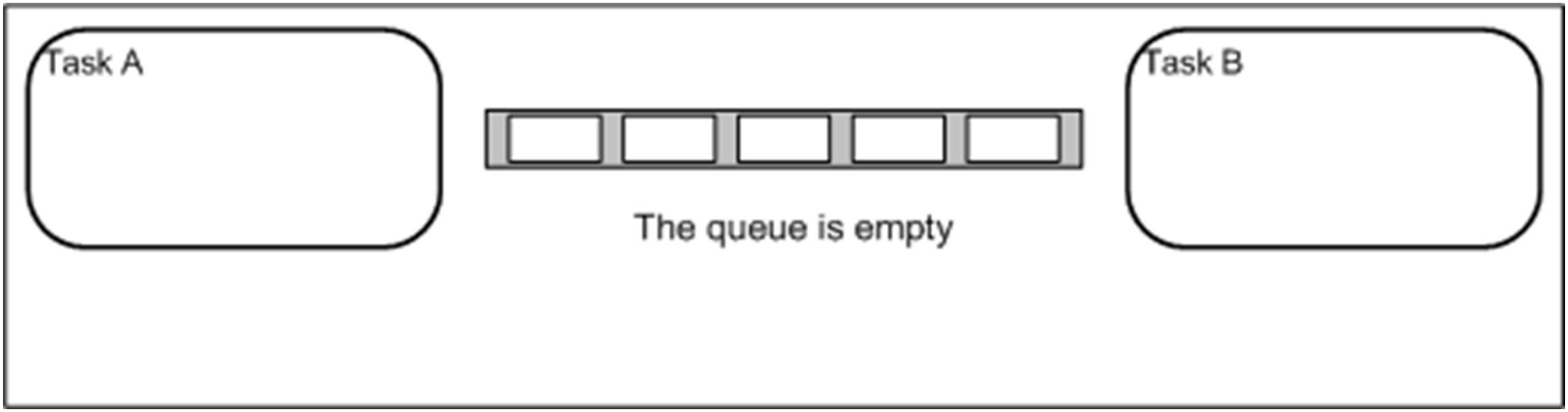
Inter-Task Communication

FreeRTOS Inter-task Communication Primitives

- Queues
- Binary/Counting Semaphores
- Mutexes and Recursive Mutexes
- Task Notifications
- Event Groups
- Stream and Message Buffers

FreeRTOS Queues

- The primary form of inter-task communications
- Can be used to **send/receive messages** between
  - Tasks
  - ISRs and Tasks



Queues

- **For users**, a message is just a data item
  - FreeRTOS does not define the message structure
  - For users, a queue is just a stream of data items
    - Users can define the structure of their messages (i.e., data items)
- Messages are sent through queues by **copying**
  - Send: copy to the Q
  - Receive: copy from the Q
- Messages can be pointers or the actual data items/structures
  - Therefore, for large data, a message can be just a pointer to the actual data item to avoid large copying overhead

Blocking on Queues

- Queue APIs permit a blocking time to be specified when
  - Sending to a full Q, or
  - Receiving from an empty Q

- If more than one tasks block on the same queue, the task *with the highest priority* will be unblocked first

- ISRs can use queues, ***But***
  - ISRs must use the APIs that end with "*FromISR*"
    - e.g., **xQueueSendFromISR(), xQueueReceiveFromISR(),…**
    - **Never blocking in these APIs!!!**

Queue Example - Defining Messages

```
/* The message structure */
struct AMessage
{
   char ucMessageID;
   char ucData[ 20 ];
} xMessage;


/* Queue used to send and receive complete struct AMessage structures. */
QueueHandle_t   xStructQueue = NULL;

/* Queue used to send and receive pointers to struct AMessage structures. */
QueueHandle_t   xPointerQueue = NULL;
```

Queue Example - Creating Message Qs

```
void vCreateQueues( void )
{
    xMessage.ucMessageID = 0xab;
    memset( &( xMessage.ucData ), 0x12, 20 );

    /* Create the queues */
    xStructQueue = xQueueCreate( 10,  sizeof( xMessage ) );          /* a Q for 10 msgs */
    xPointerQueue = xQueueCreate ( 10,  sizeof( &xMessage ) );       /* a Q for 10 msg ptrs */

    if( ( xStructQueue == NULL ) || ( xPointerQueue == NULL ) )
    {
        /* memory full, handle the error here.....
    }
}
```

Queue Example - Sending Messages

```
/* Task that writes to the queues. */
void vATask( void *pvParameters )
{
    struct AMessage *pxPointerToxMessage;

    /* Send the entire structure to the queue created to hold 10 structures. */
    xQueueSend( xStructQueue, ( void * ) &xMessage,  ( TickType_t ) 0 );

    /* Store the address of the xMessage variable in a pointer variable. */
    pxPointerToxMessage = &xMessage;

    /* Send the address of xMessage to the queue created to hold 10 pointers. */
    xQueueSend (xPointerQueue, ( void * ) &pxPointerToxMessage, ( TickType_t ) 0 );

    /* ... Rest of task code goes here. */
}
```

xTicksToWait = 0: don't block if the Q is already full.

Queue Example - Receiving Messages

```
/* Task that reads from the queues. */
void vADifferentTask( void *pvParameters )
{
    struct AMessage xRxedStructure, *pxRxedPointer;

    if( xStructQueue != NULL )
    {
        if( xQueueReceive( xStructQueue, &( xRxedStructure ), ( TickType_t ) 10 ) == pdPASS )
        {    /* xRxedStructure now contains a copy of xMessage. */ }
    }
    if( xPointerQueue != NULL )
    {
        if( xQueueReceive( xPointerQueue, &( pxRxedPointer ), ( TickType_t ) 10 ) == pdPASS )
        {  /* *pxRxedPointer now points to xMessage. */  }
    }
    /* ... Rest of task code goes here. */
}
```

xTicksToWait = **10: wait 10 ticks for incoming msgs if the Q is empty!**

Queue Control Block - Simplified Version

```
typedef struct QueueDefinition {
   int8_t *pcHead;                  /*< the beginning of the queue storage area */
   int8_t *pcWriteTo;               /*< the free next place in the storage area */
   int8_t *pcTail;                  /*< the end of the queue storage area */
   int8_t *pcReadFrom;              /*< the last place that a queued item was read from */

   List_t xTasksWaitingToSend;      /*< Tasks waiting to Tx, stored in priority order */
   List_t xTasksWaitingToReceive;   /*< Tasks waiting to Rx, stored in priority order. */
   UBaseType_t uxLength;            /*< Q size  (in # of items) */
   UBaseType_t uxItemSize;          /*< The size of each item */

....

#if( ( configSUPPORT_STATIC_ALLOCATION == 1 ) && ( configSUPPORT_DYNAMIC_ALLOCATION == 1 ) )
   uint8_t ucStaticallyAllocated;  /* = pdTRUE if memory of the Q WAS statically allocated */
#endif
} xQUEUE;

typedef xQUEUE Queue_t;
```

Queue Creation

Specify that we are creating a basic Q, not a semaphore/mutex. (*Note, the implementation of semaphore/mutex is also based on Q*)

```
/* in queue.h */

#if( configSUPPORT_DYNAMIC_ALLOCATION == 1 )
   #define xQueueCreate( uxQueueLength, uxItemSize ) \
      xQueueGenericCreate( ( uxQueueLength ), ( uxItemSize ), ( queueQUEUE_TYPE_BASE ) )
#endif
```

```
#define queueQUEUE_TYPE_BASE                ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_SET                 ( ( uint8_t ) 0U )
#define queueQUEUE_TYPE_MUTEX               ( ( uint8_t ) 1U )
#define queueQUEUE_TYPE_COUNTING_SEMAPHORE  ( ( uint8_t ) 2U )
#define queueQUEUE_TYPE_BINARY_SEMAPHORE    ( ( uint8_t ) 3U )
#define queueQUEUE_TYPE_RECURSIVE_MUTEX     ( ( uint8_t ) 4U )
```

xQueueGenericCreate (under Dynamic Allocation)

```
QueueHandle_t xQueueGenericCreate(const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize,
const uint8_t ucQueueType )
{
    Queue_t *pxNewQueue;
    size_t xQueueSizeInBytes;
    uint8_t *pucQueueStorage;

    configASSERT( uxQueueLength > ( UBaseType_t ) 0 );

    if( uxItemSize == ( UBaseType_t ) 0 ) {
       /* There is not going to be a queue storage area. */
       xQueueSizeInBytes = ( size_t ) 0;
    } else {
       /* Allocate enough space to hold the maximum number of items  */
       xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize );
    }
    pxNewQueue = ( Queue_t * ) pvPortMalloc( sizeof( Queue_t ) + xQueueSizeInBytes );
```

xQueueGenericCreate (under Dynamic Allocation)

```
    if( pxNewQueue != NULL ) {

        /* Jump past the queue structure to find the location of the queue storage area. */
        pucQueueStorage = ( uint8_t * ) pxNewQueue;
        pucQueueStorage += sizeof( Queue_t );

        #if( configSUPPORT_STATIC_ALLOCATION == 1 ) {
            pxNewQueue->ucStaticallyAllocated = pdFALSE;  /* tell the caller this Q is created dynamically */
        }
       #endif /* configSUPPORT_STATIC_ALLOCATION */

        prvInitialiseNewQueue(uxQueueLength, uxItemSize, pucQueueStorage, ucQueueType, pxNewQueue);

    }
    else {
        traceQUEUE_CREATE_FAILED( ucQueueType );
        mtCOVERAGE_TEST_MARKER();
    }
    return pxNewQueue;
}
```

prvInitialiseNewQueue()

```
static void prvInitialiseNewQueue( const UBaseType_t uxQueueLength, const UBaseType_t uxItemSize, uint8_t *
pucQueueStorage, const uint8_t ucQueueType, Queue_t *pxNewQueue )
{
   if( uxItemSize == ( UBaseType_t ) 0 ) {
      /* No RAM was allocated for the queue storage area, but pcHead cannot
      be set to NULL because NULL is used as a key to say the queue is used as
      a mutex!  Therefore, just set pcHead to point to the queue  */
      pxNewQueue->pcHead = ( int8_t * ) pxNewQueue;
   }
   else  {
      pxNewQueue->pcHead = ( int8_t * ) pucQueueStorage; /* Set the head to the start of the Q storage area. */
   }

   /* Initialise the queue members as described where the queue type is  defined. */
   pxNewQueue->uxLength = uxQueueLength;        pxNewQueue->uxItemSize = uxItemSize;
   ( void ) xQueueGenericReset( pxNewQueue, pdTRUE );  /* for init. of the other Q fields */
   ....
   traceQUEUE_CREATE( pxNewQueue );
}
```

xQueueGenericReset()

```
BaseType_t xQueueGenericReset( QueueHandle_t xQueue, BaseType_t xNewQueue ) {
   Queue_t * const pxQueue = xQueue;

   configASSERT( pxQueue );

   taskENTER_CRITICAL();
   {
      pxQueue->u.xQueue.pcTail = pxQueue->pcHead + ( pxQueue->uxLength * pxQueue->uxItemSize );

      pxQueue->uxMessagesWaiting = ( UBaseType_t ) 0U;

      pxQueue->pcWriteTo = pxQueue->pcHead;                /* the next place to be written */

      pxQueue->u.xQueue.pcReadFrom =                       /* the last place read from the Q */
         pxQueue->pcHead + ( ( pxQueue->uxLength - 1U ) * pxQueue->uxItemSize );
```

The definition of *pcTail* in Slide 11 is simplified.
Real definition of *pcTail* ➔ *u.xQueue.pcTail*

Same as *pcTail*, the definition of *pcReadFrom* in Slide 11 is simplified.
Real definition of *pcReadFrom* ➔ *u.xQueue.pcReadFrom*

xQueueGenericReset()

```
    if( xNewQueue == pdFALSE ) {  /* Not a New Q !!*/
        /* If there are tasks blocked waiting to write to the queue, then we should be unblocked one
            to allow it write to the Q. */
        if( listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) == pdFALSE )  {
            if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE ) {
                queueYIELD_IF_USING_PREEMPTION();   /* reschedule if preemption is used */
            } else { mtCOVERAGE_TEST_MARKER(); }
        } else { mtCOVERAGE_TEST_MARKER(); }
     }
     else { /* a New Q !!  Ensure the event queues start in the correct state. */
        vListInitialise( &( pxQueue->xTasksWaitingToSend ) );
        vListInitialise( &( pxQueue->xTasksWaitingToReceive ) );
     }
  }
  taskEXIT_CRITICAL();

  /* A value is returned for calling semantic consistency with previous versions. */
  return pdPASS;
}
```

FreeRTOS Semaphores

- Three types of semaphores
  - Binary
  - Counting
  - Mutex
- Creating semaphores
  - **xSemaphoreCreate*XXX*(), *XXX* = Binary, Counting, or Mutex**

- xSemaphoreTake() : wait/take a (binary/counting) semaphore or mutex
- xSemaphoreGive() : release/give a semaphore or mutex

FreeRTOS Semaphores

- Semaphore APIs permit a block time to be specified
  - the max # of ticks a task should blocked when attempting to ***take*** a semaphore
- If more than one tasks block on the same semaphore, the task with *the highest priority* will be unblocked the next time the semaphore becomes available

Binary Semaphores

- Can be used for both ~~mutual exclusion~~ and **synchronization**
- However, binary semaphores do not support priority inheritance protocol!
  - *priority inversion may occur!!!*
  - It is better to use ***mutex*** (see later) for mutual exclusion

- Created in the '*empty*' (i.e., *not available*) state
  - must first be given using the xSemaphoreGive() before it can subsequently be taken (obtained) using the xSemaphoreTake()

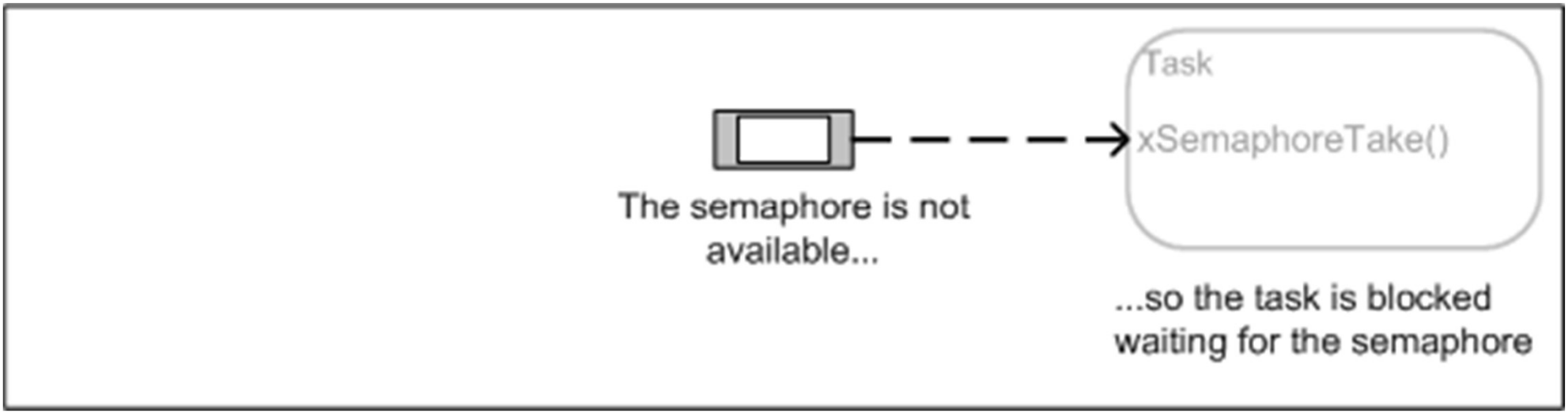
Binary Semaphores

- Once obtained, a binary semaphore **need NOT** to be **given back**
  - For example, Task A can continuously 'gives' the semaphore while Task B continuously 'taking' the semaphore!!!
    - E.g., Task A: dispatcher task, Task B: worker task.

    B can wait (take the semaphore) when no jobs assigned to him

    A can signal (give the semaphore) B when A assigns a job to B
- Binary semaphores can be used in ISRs
  - For example, you can replace the **Task A** above with an ISR, and Task B becomes a deferred interrupt handling task (see the next slide)

Binary Semaphores

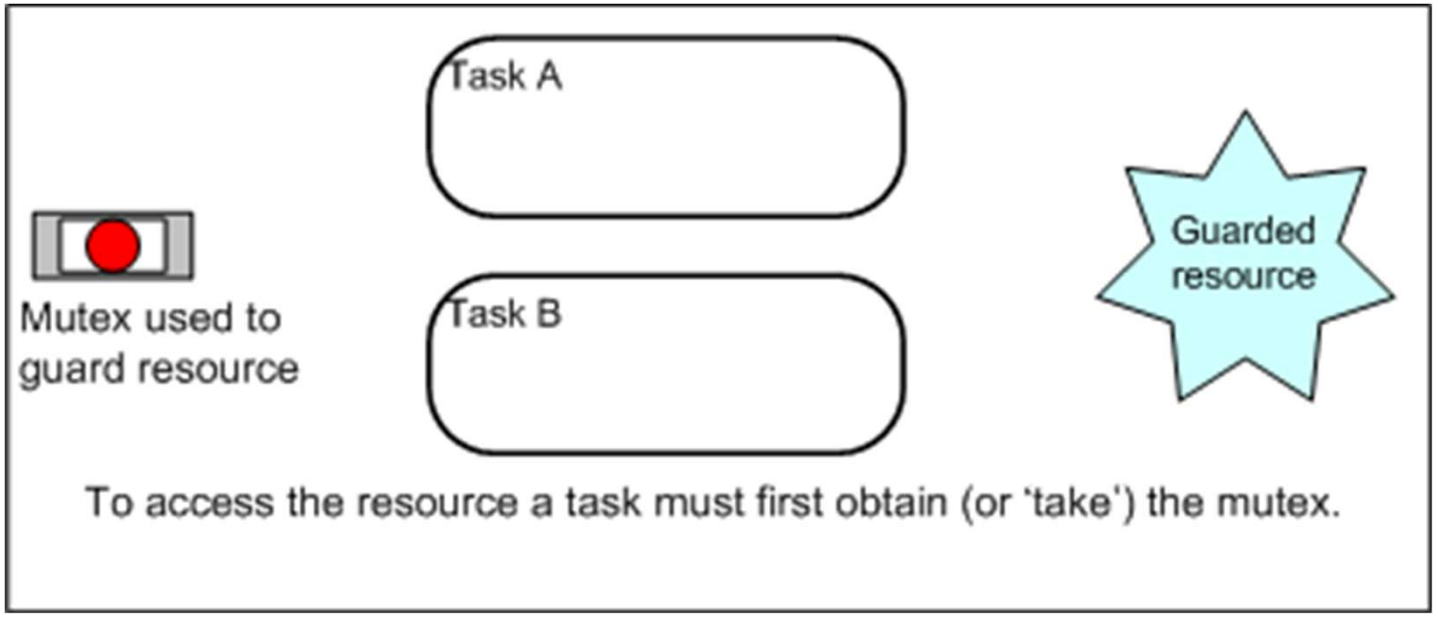


Counting Semaphores

- Two typical user cases
  - Counting pending events
    - a task gives a semaphore S each time an event occurs (S.value++)
    - another task takes S each time it processes an event (S.value--)
    - Initially, S.value = 0
  - Resource management
    - Management of N resources ➔ S.value = N initially
    - Get a resource (S.value--)
    - Release a resource (S.value++)

Mutexes

- Mutexes = binary semaphores + **priority inheritance** mechanism
  - When a high priority task A blocks on a mutex, the priority of the mutex holder B is temporarily raised to the priority of A.
- A better choice for implementing simple **mutual exclusion**



Mutexes

- Mutexes **should NOT** be used from an interrupt
  - priority inheritance only makes sense for tasks, not ISRs
  - An ISR should not block to wait for a resource

- **Recursive Mutex**
  - A recursive mutex that can be taken **repeatedly** by the mutex holder
    - Non-recursive mutex will cause deadlock if the mutex holder tries to take the mutex again !
  - API: xSemaphoreGiveRecursive(), xSemaphoreTakeRecursive()
  - if a task successfully *takes* the same mutex ***k*** times, then the mutex will not be available to any other task until it has also *given* the mutex back exactly ***k*** times !

RTOS Task Notifications

- Available from FreeRTOS v8.2.0
- Each RTOS task has a 32-bit **notification value**
  - initialized to **zero** during task creation
- task notification = send an event to a task that can
  - **unblock** the receiving task
  - optionally, update the receiving task's notification value, in the following ways
    - Set a new value *without overwriting* the old value
    - *Overwrite* the old value
    - Set *one or more bits* in the receiving task's notification value
    - *Increment* the receiving task's notification value
- Can be used as lightweight *queues*, *semaphores* or *event groups*
  - Unblocking an RTOS task with a direct notification is *45% faster* and *uses less RAM* than unblocking a task with a binary semaphore

Task Notifications

- Send a notification via the **xTaskNotify()/xTaskNotifyGive()** (and their interrupt safe equivalents)
  - Remain **pending** until the receiving task calls xTaskNotifyWait() or ulTaskNotifyTake()
  - If the receiving task was already blocked waiting for a notification, it will be **unblocked** and the notification will be cleared
- Configuration
  - **configUSE_TASK_NOTIFICATIONS** [FreeRTOSConfig.h]
    - Default set as 1

ulTaskNotifyTake()

- Intended for use when a task notification is used as a lightweight binary or counting semaphore
- Can be used to replace the xSemaphoreTake() API
- ulTaskNotifyTake()
  - Can block to wait for a the task's notification value to be non-zero
  - On exit, it can
    - clear the task's notification value to zero, or ➔ **like a binary semaphore**
    - decrement the task's notification value ➔ **like a counting semaphore**

ulTaskNotifyTake()

- uint32_t **ulTaskNotifyTake**(BaseType_t xClearCountOnExit, TickType_t xTicksToWait);
- xClearCountOnExit
  - pdTRUE: reset the notification value to 0 before return
  - pdFALSE: decrement the notification value before return
- xTicksToWait
  - maximum blocking time if a notification is **not already pending** when ulTaskNotifyTake() is called
  - The time is specified in *ticks*.
- **Returns**
  - The notification value **before** it is decremented or cleared

xTaskNotifyGive()

- Used in pair with **ulTaskNotifyTake()**
  - For lightweight binary or counting semaphores
- Similar to **xSemaphoreGive**()
- Must **NOT** be called from an interrupt service routine
  - In an ISR, use **vTaskNotifyGiveFromISR**() instead

xTaskNotifyGive()

- BaseType_t **xTaskNotifyGive**(TaskHandle_t xTaskToNotify);
  - Increment the notification value of the task xTaskToNotify
  - The task handle can be obtained on task creation
  - Return value: pdPASS
- xTaskNotifyGive() is a macro that calls **xTaskNotify()** with the *eAction* parameter set to *eIncrement*

The *eAction* Parameter of xTaskNotify()

BaseType_t xTaskNotify (TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction *eAction*);

eNoAction	• Notification value of the target task is not updated
eSetBits	• Notification value of the target task will be **bitwise ORed** with **ulValue** • A lightweight alternative to an event group
eIncrement	• The notification value of the target task will be **incremented by 1** • Equivalent to a call to xTaskNotifyGive()
eSetValueWith-Overwrite	• The notification value of the target task is **set to ulValue** • A lightweight alternative to xQueueOverwrite()
eSetValueWithout-Overwrite	• If the target task **does not already** have a notification pending then its notification value will be **set to ulValue** • If the target task already **has a notification pending**, then its notification value is **not updated**, and the call to xTaskNotify() **fails** (return pdFALSE) • A lightweight xQueueSend() with queue length = 1

Example

```
…
/* Handles for the tasks create by main(). */
static TaskHandle_t xTask1 = NULL, xTask2 = NULL;

/* Create two tasks that send notifications back and forth to each other, then start the RTOS scheduler. */
void main( void )
{
   xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );
   xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );
   vTaskStartScheduler();
}

static void prvTask1( void *pvParameters )
{
   for( ;; )
   {
      /* Send a notification to prvTask2(), bringing it out of the Blocked  state. */
      xTaskNotifyGive( xTask2 );

      /* Block to wait for prvTask2() to notify this task. */
      ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
   }
}
```

Example (cont.)

```
static void prvTask2( void *pvParameters )
{
   for( ;; )
   {
      /* Block to wait for prvTask1() to notify this task. */
      ulTaskNotifyTake( pdTRUE, portMAX_DELAY );

      /* Send a notification to prvTask1(), bringing it out of the Blocked state. */
      xTaskNotifyGive( xTask1 );
   }
}
```

vTaskNotifyGiveFromISR()

- Similar to **xTaskNotifyGive()**, but can be used in an **ISR**
- Used in pair with **ulTaskNotifyTake()**
  - For lightweight binary or counting semaphores
  - Similar to **xSemaphoreGiveFromISR**()
- **void vTaskNotifyGiveFromISR(**

  TaskHandle_t xTaskToNotify,

  BaseType_t *pxHigherPriorityTaskWoken **);**
- pxHigherPriorityTaskWoken will be set to **pdTRUE (by the callee)** if
  - sending the notification caused a task to unblock
  - priority of the unblocked task > priority of the current task

So, the caller ISR can determine **whether or not to do a context switch** when it is finished!

Example - Using Tasks for Interrupt Processing

```
/* An interrupt handler.
It does NOT perform any processing, instead it unblocks a high priority task to process the interrupt.
If the priority of the task is high enough then the interrupt will return directly to the task */

void vANInterruptHandler( void ) {
   BaseType_t xHigherPriorityTaskWoken;

   prvClearInterruptSource(); /* Clear the interrupt. */

   /* Initialise xHigherPriorityTaskWoken to pdFALSE.  If priority of the unblocked task > priority
    of the currently task, xHigherPriorityTaskWoken will be set to pdTRUE. */
   xHigherPriorityTaskWoken = pdFALSE;

   /* Unblock the handling task */
   vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

   /* Force a context switch if xHigherPriorityTaskWoken = pdTRUE.*/
   portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Trigger **PendSV** in ARM CM3 if xHigherPriorityTaskWoken == pdTRUE

Example - Using Tasks for Interrupt Processing (cont.)

```
/* The handler task */
void vHandlingTask( void *pvParameters ) {
   BaseType_t xEvent;

   for( ;; ) {

      /* Block indefinitely to wait for a notification.  */
      ulTaskNotifyTake( pdTRUE,  portMAX_DELAY ); /* Block indefinitely. */

      /* The RTOS task notification is used as a binary semaphore,
          so only go back to wait for further notifications
          when ALL events pending in the peripheral have been processed. */
      do {
         xEvent = xQueryPeripheral();
         if( xEvent != NO_MORE_EVENTS ) {    vProcessPeripheralEvent( xEvent );  }
      } while( xEvent != NO_MORE_EVENTS );
   }
}
```

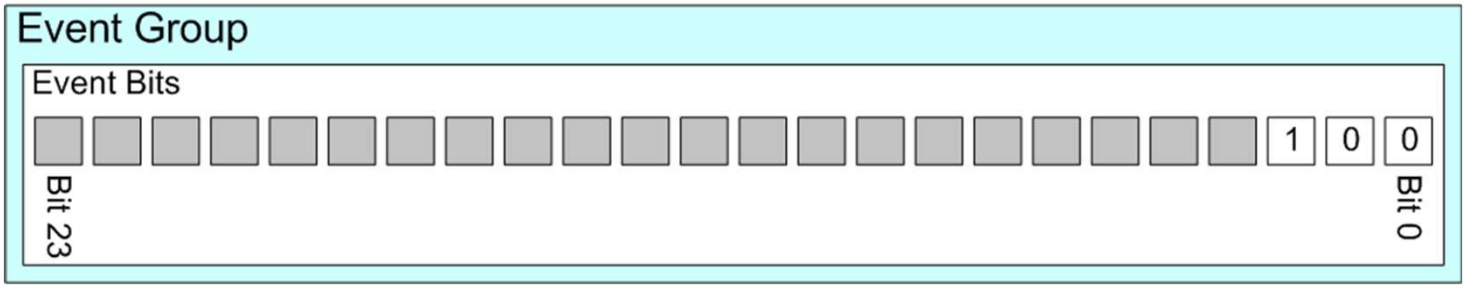
Using Task Notifications as Light Weight Mailboxes

- Task notifications can be used to send data to a task
- More lightweight, but **more restricted** than an RTOS queue
  - Only 32-bit values can be sent
  - Only one notification value at any one time
    - The value is saved as the receiving task's notification value
- Send messages via xTaskNotify() or xTaskNotifyFromISR()
  - The *eAction* parameter set to either *eSetValueWithOverwrite* or *eSetValueWithoutOverwrite*
- Receive messages via xTaskNotifyWait()

FreeRTOS Event Groups

- Event group = a set of event bits/flags
  - An event bit = 1 : the event occurred
  - An event bit = 0 : the event not occurred
- # of bits in an event group (EventGroupHandle_t)
  - 8 if configUSE_16_BIT_TICKS = 1
  - 24 if configUSE_16_BIT_TICKS = 0
- All the event bits in an event group are stored in a single unsigned variable of type EventBits_t

```
/* The type that holds event bits always
matches TickType_t */
typedef TickType_t EventBits_t;
```



FreeRTOS Event Groups

- Created by xEventGroupCreate() or xEventGroupCreateStatic()
  - EventGroupHandle_t xEventGroupCreate(void);
- With event group APIs, users can
  - Set **one or more event bits** within an event group
    - xEventGroupSetBitsFromISR() or xEventGroupSetBits()
  - Clear one or more event bits within an event group
    - xEventGroupClearBitsFromISR() or xEventGroupClearBits()
  - Wait for a set of one or more event bits to become set
    - xEventGroupWaitBits()

Example API- API for Waiting Event(s)

- EventBits_t **xEventGroupWaitBits**(const EventGroupHandle_t xEventGroup,
  const EventBits_t **uxBitsToWaitFor**, const BaseType_t xClearOnExit,
  const BaseType_t **xWaitForAllBits**, TickType_t xTicksToWait);

- uxBitsToWaitFor
  - A bitwise value that indicates the bit(s) to wait for
  - E.g. 5 (101) if bits 0 and 2 are wait
- xWaitForAllBits
  - pdTRUE ➔ return when **ALL** the bits specified in uxBitsToWaitFor must be set!
  - pdFALSE ➔ return when **ANY** bit specified in uxBitsToWaitFor is set!
- Return event bits at the time
  - The event bits being waited for became set (**before clear**), or
  - The timer expires

The user can check the return value to see which event(s) were set, or the timer expires.

Example

```
#define BIT_0    ( 1 << 0 )
#define BIT_4    ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup ) {
  EventBits_t uxBits;
  const TickType_t xTicksToWait = 100 / portTICK_PERIOD_MS;

   /* Wait a maximum of 100ms for either bit 0 or bit 4 to be set.  Clear the bits before exiting. */
   uxBits = xEventGroupWaitBits(
         xEventGroup,   /* The event group being tested. */
         BIT_0 | BIT_4, /* The bits to wait for */
         pdTRUE,        /* BIT_0 & BIT_4 should be cleared before returning. */
         pdFALSE,       /* Don't wait for both bits, either bit will do. */
         xTicksToWait );/* Wait a maximum of 100ms for either bit to be set. */

  if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) ) { /* both bits were set. */ }
  else if( ( uxBits & BIT_0 ) != 0 ) { /* xEventGroupWaitBits() returned because just BIT_0 was set. */  }
  else if( ( uxBits & BIT_4 ) != 0 ) { /* xEventGroupWaitBits() returned because just BIT_4 was set. */ }
  else { /* xEventGroupWaitBits() returned because xTicksToWait ticks passed  */  }
}
```

Task Notifications as Event Groups

- Task notifications can be used as a more lightweight event group implementation
- Bits in the receiver's notification value become event bits
  - Wait bits by xTaskNotifyWait()
  - Set bits by xTaskNotify()/xTaskNotifyFromISR()
    - Parameter *eAction* set to ***eSetBits***
- xTaskNotifyFromISR() is faster than xEventGroupSetBitsFromISR()
  - xTaskNotifyFromISR() runs entirely in ISR
  - xEventGroupSetBitsFromISR() defers some jobs to the daemon task

Task Notifications as Event Groups

- However, the receiving task **CANNOT** wait for multiple bits to be active at the same time (i.e., the case when xWaitForAllBits is set as pdTRUE)
  - Instead, the receiving task is unblocked when any bit becomes set
    - The receiving task must test for bit combinations itself

FreeRTOS Stream Buffers

- A task-task, or ISR-task communication primitive
- A stream buffer passes a continuous stream of bytes
- Data is passed by copy
  - data copied into the buffer by the sender and
  - data copied out of the buffer by the receiver
- Optimized for ***single reader single writer*** scenario
  - e.g., passing data from an ISR to a task
  - e.g., passing data from one core to another core

  Note: NOT safe to have multiple different writers or multiple different readers!
- Based on task notifications!
  - Task notifications must be enabled to use stream buffers.

The Stream Buffer API

- **xStreamBufferCreate**(), xStreamBufferCreateStatic()
- **xStreamBufferSend**(), xStreamBufferSendFromISR()
- **xStreamBufferReceive**(), xStreamBufferReceiveFromISR()
- vStreamBufferDelete()
- xStreamBufferBytesAvailable(), xStreamBufferSpacesAvailable()
- **xStreamBufferSetTriggerLevel**()
- xStreamBufferReset()
- xStreamBufferIsEmpty(), xStreamBufferIsFull()

API Example - xStreamBufferReceive()

- size_t **xStreamBufferReceive(**

  StreamBufferHandle_t xStreamBuffer, void *pvRxData,

  size_t xBufferLengthBytes, TickType_t xTicksToWait);

- pvRxData: pointer to the buffer for the received data
- xTicksToWait: the max time to wait
- xBufferLengthBytes: size of the received buffer (i.e., the pvRxData)
  - The receiver waits (i.e., enters the blocked state) if the stream buffer is **empty**
  - Later, the blocked receiver will be unblocked when
    - The wait time (i.e., the xTicksToWait) expires, or
    - The data size in the stream buffer >= **trigger level** of the stream buffer
- **Return**
  - The number of bytes actually received

Stream Buffer Structure

```
typedef struct StreamBufferDef_t {
        volatile size_t xTail;                  /* point to the next item to read */
        volatile size_t xHead;                  /* point to the next item to write */
        size_t xLength;                         /* The length of the buffer  */
        size_t xTriggerLevelBytes;              /*  # of bytes that must be in the stream buffer before a
                                                    task that is waiting for data is unblocked. */
        volatile TaskHandle_t xTaskWaitingToReceive;  /* The task waiting for receive data */
        volatile TaskHandle_t xTaskWaitingToSend;     /* The task waiting to send data */
        uint8_t *pucBuffer;                     /* Points to the buffer itself  */
        uint8_t ucFlags;

} StreamBuffer_t;
```

- **Both Tx and Rx can block**
  - Sender can block on full stream buffer; Receiver can block on empty stream buffer
    - Stream buffer size is specified on stream buffer creation
- **Single sender, single receiver**
  - Only a task in xTaskWaitingToReceive, only a task in xTaskWaitingToSend

Selected Code in xStreamBufferReceive()

```
if( xTicksToWait != ( TickType_t ) 0 ) {   /* blocking/waiting is allowed.... */
        taskENTER_CRITICAL();
        {
            xBytesAvailable = prvBytesInBuffer( pxStreamBuffer ); /* get # of bytes in stream buf */

            if( xBytesAvailable <= 0 ) {
                xTaskNotifyStateClear( NULL );  /* Clear notification state before waiting for data. */

                /* Should only be one reader. */
                configASSERT( pxStreamBuffer->xTaskWaitingToReceive == NULL );
                pxStreamBuffer->xTaskWaitingToReceive = xTaskGetCurrentTaskHandle();
            }
        }
        taskEXIT_CRITICAL();
        if( xBytesAvailable <= 0 ) {
            xTaskNotifyWait( ( uint32_t ) 0, ( uint32_t ) 0, NULL, xTicksToWait ); /* sleep, wait for data */
            pxStreamBuffer->xTaskWaitingToReceive = NULL;  /* wakeup, no longer waiting */
            xBytesAvailable = prvBytesInBuffer( pxStreamBuffer );  /* Recheck data available once wakeup */
        }
}
```

- **Based on Task notifications!**
- **The receiver will be woken up when the number of bytes in the stream buffer >= trigger level**

Selected Code in xStreamBufferReceive() (cont.)

```
if( xBytesAvailable > 0 ) {  /* Rx success...*/
        xReceivedLength = prvReadMessageFromBuffer( pxStreamBuffer, pvRxData, xBufferLengthBytes,
                                                    xBytesAvailable, xBytesToStoreMessageLength );

        /* Was a task waiting for space in the buffer? */
        if( xReceivedLength != ( size_t ) 0 ) {
            traceSTREAM_BUFFER_RECEIVE( xStreamBuffer, xReceivedLength );
            sbRECEIVE_COMPLETED( pxStreamBuffer );  /* call the receive complete macro */
        }
}
else {     /* Rx failed...*/  traceSTREAM_BUFFER_RECEIVE_FAILED( xStreamBuffer );  }


return xReceivedLength;
```

Default Implementation of the Receive Complete Macro

Wake up the blocked sender

```
#ifndef sbRECEIVE_COMPLETED
    #define sbRECEIVE_COMPLETED( pxStreamBuffer )                                \
        vTaskSuspendAll();                                                       \
        {                                                                        \
            if( ( pxStreamBuffer )->xTaskWaitingToSend != NULL )                 \
            {                                                                    \
                ( void ) xTaskNotify( ( pxStreamBuffer )->xTaskWaitingToSend,    \
                                      ( uint32_t ) 0,                            \
                                      eNoAction );                               \
                ( pxStreamBuffer )->xTaskWaitingToSend = NULL;                   \
            }                                                                    \
        }                                                                        \
        xTaskResumeAll();
#endif /* sbRECEIVE_COMPLETED */
```

Send Complete and Receive Complete Macros

- sbRECEIVE_COMPLETED()/ sbRECEIVE_COMPLETED_FROM_ISR()
  - Called when data is read from a stream buffer
  - Default action
    - Wake up the blocked sender
- sbSEND_COMPLETED()/sbSEND_COMPLETED_FROM_ISR()
  - Called when data is written to a stream buffer and **available data size >= trigger level**
  - Default action
    - Wake up the blocked receiver

These macros can be overwritten by the applications!

Changing the Behavior of the Send/Receive Completed Macros

The send/receive completed macros can be overwritten by the applications!

- Define your own functions in FreeRTOSConfig.h
  - #define **sbSEND_COMPLETED**(pxStreamBuffer) *myfunc*(pxStreamBuffer)
- **Useful for core-to-core communication**
  - An example
    - In *myfunc*(), you send an interrupt to the destination core, and
    - ISR of the destination core calls xStreamBuffer**SendCompleted**FromISR() to unblock the waiting receiving task on the destination core

**** We will describe an example later*** by using message buffer

- Reference: FreeRTOS/Demo/Common/Minimal/MessageBufferAMP.c

FreeRTOS Message Buffers

- A task-task, or ISR-task communication primitive
- Allow *variable-length* discrete messages to be passed
  - A single message buffer can have *messages with different sizes*
- The implementation is built on top of *stream buffers*, so
  - For single reader single writer
  - Based on task notifications

The Message Buffer API

- xMessageBufferCreate(), xMessageBufferCreateStatic()
- xMessageBufferSend(), xMessageBufferSendFromISR()
- xMessageBufferReceive(), xMessageBufferReceiveFromISR()
- vMessageBufferDelete()
- xMessageBufferSpacesAvailable()
- xMessageBufferReset()
- xMessageBufferIsEmpty()
- xMessageBufferIsFull()

API Example - xMessageBufferReceive()

- size_t **xMessageBufferReceive(** ← Receive a SINGLE message from the message buffer

  MessageBufferHandle_t xMessageBuffer, void *pvRxData,

  size_t xBufferLengthBytes, TickType_t xTicksToWait);

- pvRxData: pointer to the buffer for the received data
- xTicksToWait: the max time to wait
- xBufferLengthBytes: size of the received buffer (i.e., the pvRxData)
  - The receiver waits (i.e., enters the blocked state) if the message buffer is **empty**
  - If the length of the message is greater than xBufferLengthBytes then the message will be left in the message buffer and zero is returned.
- **Return**
  - The number of bytes of the received message

The Message Buffer Structure

```
typedef struct StreamBufferDef_t {
        volatile size_t xTail;                          /* point to the next item to read */
        volatile size_t xHead;                          /* point to the next item to write */
        size_t xLength;                                 /* The length of the buffer  */
        size_t xTriggerLevelBytes;                      /*  # of bytes that must be in the stream buffer before a
                                                            task that is waiting for data is unblocked. */
        volatile TaskHandle_t xTaskWaitingToReceive;    /* The task waiting for receive data */
        volatile TaskHandle_t xTaskWaitingToSend;       /* The task waiting to send data */
        uint8_t *pucBuffer;                             /* Points to the buffer itself  */
        uint8_t ucFlags;

} StreamBuffer_t;
```

A message buffer structure is a *special* stream buffer with
.sbFLAGS_IS_MESSAGE_BUFFER is set in *ucFlags*
.message size is written to the stream buffer before each message write

Message Buffer

- Reuse the implementation of stream buffer

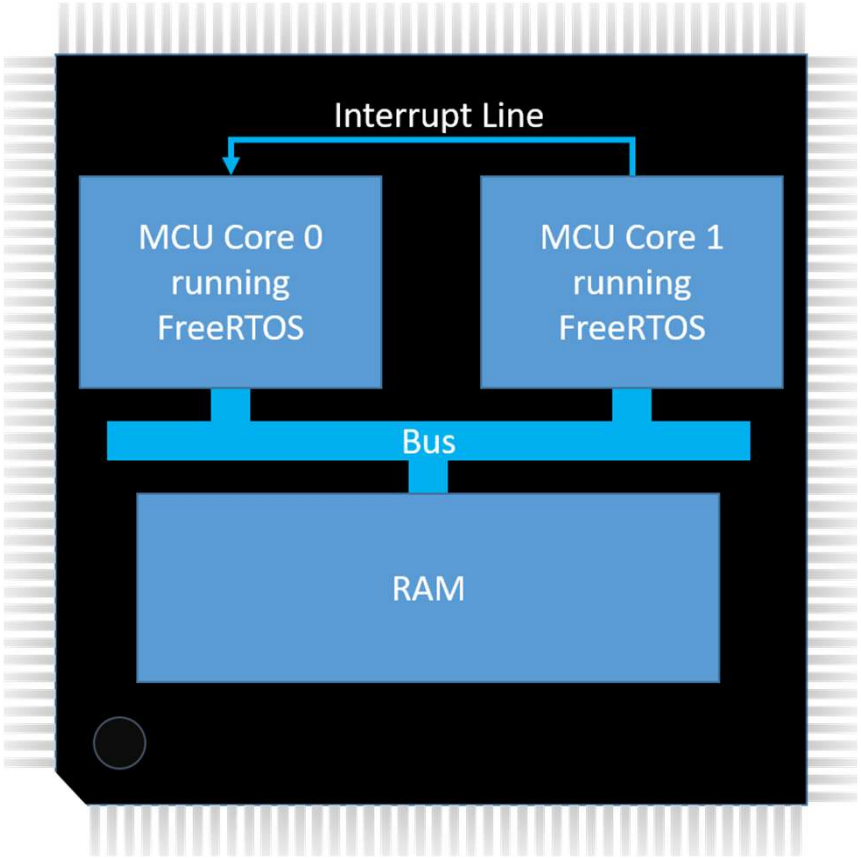
in **include/message_buffer.h**

```
#define xMessageBufferSend( xMessageBuffer, pvTxData, xDataLengthBytes, xTicksToWait )
    xStreamBufferSend( ( StreamBufferHandle_t ) xMessageBuffer, pvTxData, xDataLengthBytes, xTicksToWait )

#define xMessageBufferReceive( xMessageBuffer, pvRxData, xBufferLengthBytes, xTicksToWait )
    xStreamBufferReceive( ( StreamBufferHandle_t ) xMessageBuffer, pvRxData, xBufferLengthBytes, xTicksToWait )

…
```

Using Message Buffers for Multicore Communication - Hardware Configuration



Hardware Configuration

- Each core runs its own FreeRTOS
  - Like Asymmetric Multi-Processor (AMP)

- Shared memory between cores
- Inter-core interrupt

https://www.freertos.org/2020/02/simple-multicore-core-to-core-communication-using-freertos-message-buffers.html

Simplified Pseudocode for Message Buffer Send/Receive

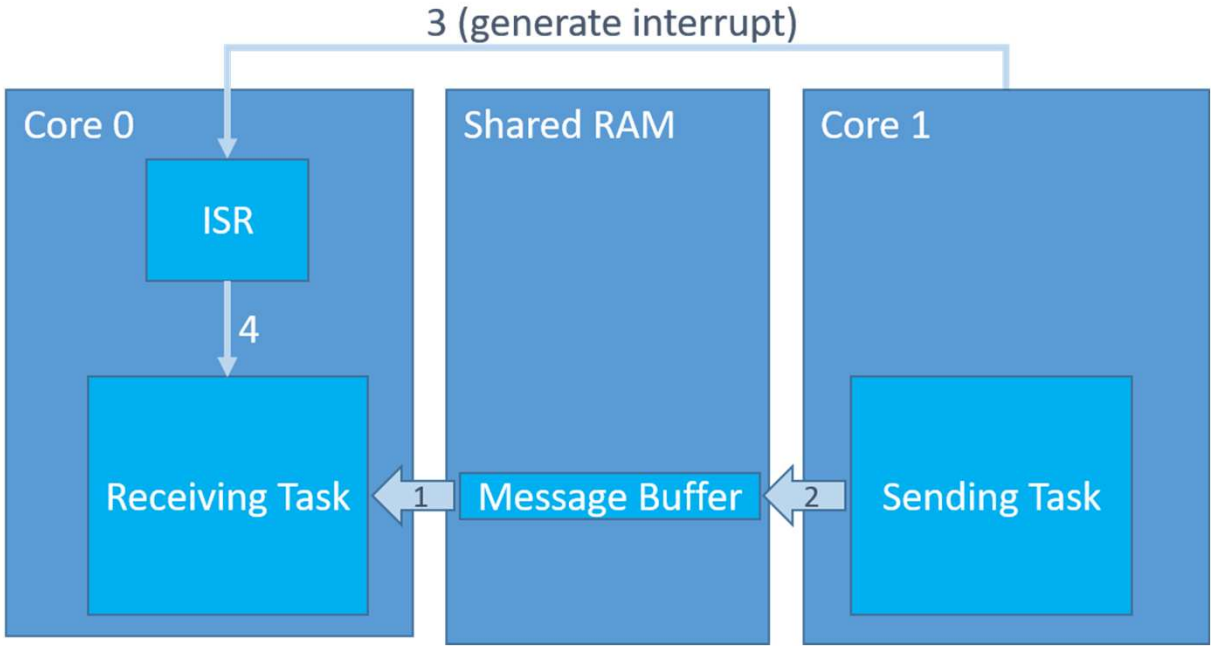
```
xMessageBufferSend()
{
   /* If a time out is specified and there isn't enough
   space in the message buffer to send the data, then
   enter the blocked state to wait for more space. */
   if( time out != 0 )
   {
      while( there is insufficient space in the buffer &&
            not timed out waiting )
      {
         Enter the blocked state to wait for space in the buffer
      }
   }

   if( there is enough space in the buffer )
   {
      write data to buffer
      sbSEND_COMPLETED()
   }
}
```

```
xMessageBufferReceive()
{
   /* If a time out is specified and the buffer doesn't
   contain any data that can be read, then enter the
   blocked state to wait for the buffer to contain data. */
   if( time out != 0 )
   {
      while( there is no data in the buffer &&
            not timed out waiting )
      {
         Enter the blocked state to wait for data
      }
   }

   if( there is data in the buffer )
   {
      read data from buffer
      sbRECEIVE_COMPLETED()
   }
}
```

Using Message Buffers for Multicore Communication - The Overall Flow



1. The receiving task on core 0 attempts to read from an empty message buffer and enters the blocked state
2. The sending task on core 1 writes data to the message buffer
3. The modified ***sbSEND_COMPLETED()*** triggers an interrupt to core 0
4. The **ISR** on core 0 calls ***xMessageBufferSendCompletedFromISR()*** to unblock the receiving task

Using Message Buffers for Multicore Communication - Multiple Msg. Buffers

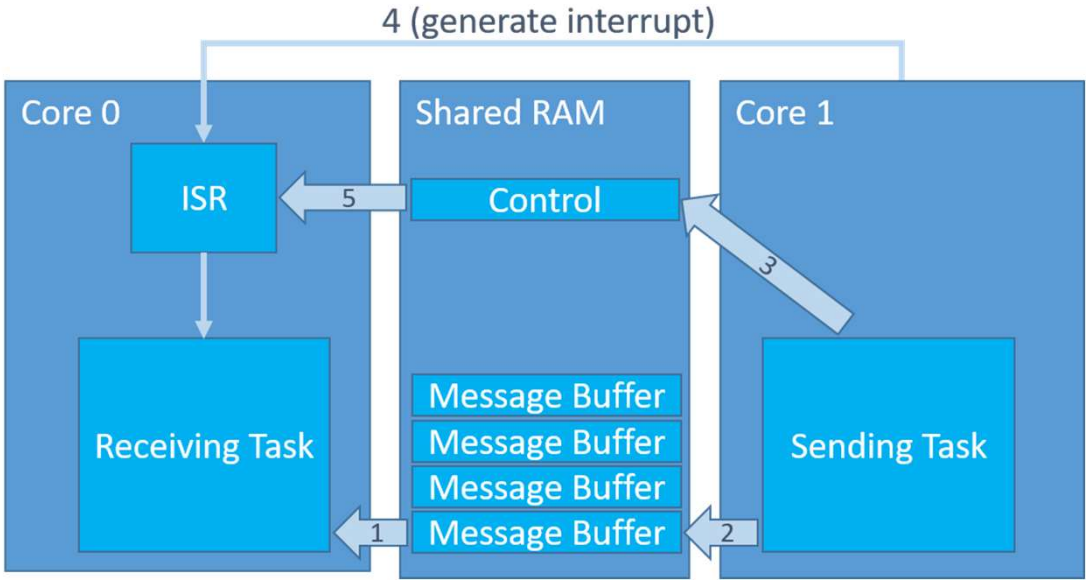
- When calling xMessageBufferSendCompletedFromISR(), the ISR needs to pass the handle of the message buffer as input, but the ISR does not know which message buffer is associated with this interrupt!
  - No problem if a single message buffer is used in the system
    - Always pass the same message buffer
  - "***Solutions***" for **multiple** message buffers
    - Use a dedicated interrupt for each message buffer ➔ but only few interrupt lines
    - Let the ISR query each message buffer to see if it contains data ➔ long latency
    - Combine multiple message buffers into one ➔ complicated

These "solutions" are inefficient if there are a large or unknown number of message buffers!

Using Message Buffers for Multicore Communication - Multiple Msg. Buffers

An Efficient Solution

Using a ***control*** message buffer
-- used to pass message buffer handles



1. The receiving task attempts to read from an empty message buffer and then blocked
2. The sending task writes data to the message buffer, say message buffer 1
3. *sbSEND_COMPLETED*() sends the handle of the message buffer 1 to the control message buffer
4. *sbSEND_COMPLETED*() triggers an interrupt to core 0
5. The ISR gets the handle of the message buffer 1 from the control message buffer, then passes the handle to xMessageBufferSendCompletedFromISR() to unblock the receiving task

The Overwritten sbSEND_COMPLETED()

```
void vMysbSEND_COMPLETED( MessageBufferHandle_t xUpdatedBuffer )
{
    size_t BytesWritten;

    /* write the xUpdatedBuffer to the control message buffer.
       If this function was called because data was written to the control message buffer then do nothing. */
    if( xUpdatedBuffer != xControlMessageBuffer )
    {
        BytesWritten = xMessageBufferSend(  xControlMessageBuffer,
                                            &xUpdatedBuffer,
                                            sizeof( xUpdatedBuffer ),
                                            0 );

        /* If the bytes could not be written then the control message buffer is too small! */
        configASSERT( BytesWritten == sizeof( xUpdatedBuffer );

        /* Generate interrupt in the other core (pseudocode). */
        GenerateInterrupt();
    }
}
```

The message buffer that has data now

The ISR

```
void InterruptServiceRoutine( void )
{
    MessageBufferHandle_t xUpdatedMessageBuffer;
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Receive the handle(s) of the message buffer(s) that contain data from the control message buffer.
       Ensure to drain the buffer before returning. */
    while( xMessageBufferReceiveFromISR( xControlMessageBuffer,  &xUpdatedMessageBuffer,
                                         sizeof(xUpdatedMessageBuffer), &xHigherPriorityTaskWoken )
                                == sizeof( xUpdatedMessageBuffer ) ) {
       /* sends a notification to any task that is blocked on xUpdatedMessageBuffer message buffer waiting for data to arrive. */
       xMessageBufferSendCompletedFromISR( xUpdatedMessageBuffer, &xHigherPriorityTaskWoken );
    }

    /* Normal FreeRTOS "yield from interrupt" semantics */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Will be set to pdTRUE if the call unblocks a task that has a priority above that of the current task!