IMPERIAL COLLEGE LONDON

E4.52
ISE3.35

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2008

EEE/ISE PART III/IV: MEng, BEng and ACGI

Corrected Copy

REAL-TIME OPERATING SYSTEMS

Tuesday, 13 May 10:00 am

Time allowed: 3:00 hours

There are SIX questions on this paper.

Answer FOUR questions.

All questions carry equal marks.

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible First Marker(s) : T.J.W. Clarke
Second Marker(s) : Y.K. Demiris

Special instructions for invigilators

The booklet RTOS Exam Notes 2008 should be distributed with the paper.

Special instructions for students

You may use the booklet RTOS Exam Notes 2008 which is a reproduction of that published on the course web-site before the exam.

The Questions

1.

(a) Explain the role of extended rate monotonic analysis in real-time system design, detailing all the assumptions necessary for the RMA theorem to apply.

[4]

(b) *Figure 1.1* details three tasks, T1, T2, T3 in a real-time system using two semaphores, A & B, to ensure exclusive access to resources R_A and R_B. An access time of 0 indicates that the resource is not used. Explain (with an appropriate execution trace) why this system suffers from priority inversion. Use extended RMA to determine one or more inequalities on *t1*, *t2* which guarantee the system will meet all its deadlines.

[8]

(c) A given real-time application has been extensively tested on an embedded CPU and appears to work. When a faster processor is used to run the same application, two tasks generate none of the expected output and appear to be halted. List all possible reasons for this problem, and state how you would distinguish between them. Comment on the correctness of the application running on the original processor.

[8]

	Job Time	**Job Period**	**Time accessing R_A**	**Time accessing R_B**
T1	20 us	200 us	*t1*	*0*
T2	10 us	100 us	0	*t2*
T3	100 us	400 us	*t1*	*t2*

Figure 1.1

2.

(a) A priority-scheduled real-time system consists of four jobs with the characteristics shown in *Figure 2.1*. Answer the following questions for all values $t1 > 50$ us, using inequalities on $t1$ as necessary to determine the answer.

(i) How would you prioritise these jobs under an RTOS with prioritised tasks?

(ii) Can you state with certainty whether or not all tasks will meet their deadlines, and if so will they, run? Give reasons for your answer. You may assume RTOS task-switching overheads are negligible.

[6]

(b) Inter-task communication is introduced to the system of *Figure 2.1* which results in blocking as specified in *Figure 2.2* every blocked task job period. State, giving reasons, which of the non-zero times in Figure 2.2 affect the system's operation? What is the minimum value of $t1$ for which the system meets all of its deadlines?

[6]

(c) Suppose that tasks, with blocking as in *Figure 2.2*, are scheduled using Earliest Deadline First (EDF) scheduling. What is the minimum value of $t1$ for which the system meets all of its deadlines?

[4]

(d) Assume that the blocking in *Figure 2.2* may happen at any time and that the scheduling algorithm has no information about whether the blocking of a task has already occurred. By considering the worst-case execution trace of each task, or otherwise, determine how EDF scheduling must be modified when applied to this system? What is the maximum CPU utilisation that can be achieved under EDF meeting all deadlines?

[4]

Job	**Job Time**	**Job Period**
X	50 us	$t1$ us
Y	1 us	10 us
Z	100 us	400 us
W	50 us	250 us

Figure 2.1

		Blocking task			
		X	**Y**	**Z**	**W**
Blocked task	**X**	0	0	0	0
	Y	0	0	0	0
	Z	0	20 us	0	0
	W	0	13 us	0	0

Figure 2.2

3. This question relates to the v4.0.5 FreeRTOS implementation of queues: source code for FreeRTOS v4.0.5 is contained in the Exam Notes 2008.

(a) Tasks A & B execute the code shown in *Figure 3.1*, in which task A sends message *m1* to queue *q1*. Assuming that task A has lower priority than task B, and both tasks are initially ready to run at the start of the two code fragments, trace through the sequence of kernel code executed until either task A or task B reaches point X.

[6]

(b) Explain what would change in your part a answer if task B had lower priority than task A.

[4]

(c) State under precisely what circumstances might ISRs execute concurrently with lines 505-525 of QueueSend(), explaining your reasoning.

[4]

(d) Suppose that Tasks A,B,C are as in *Figure 3.1* with priority C>B>A. While task A is executing QueueSend() an ISR sends a message *m2* to *q1*. In which sections of QueueSend code can the ISR happen? Determine in each case which task receives each of the messages *m1*, *m2*.

[6]

Task A	**Task B, C**

`QueueSend(q1, &m1, 1000);`
`QueueReceive(q1, &buffa,1000);`
`TaskDelay(1);`
`/* X */`
	`while (1) {`
	`      QueueReceive(q1, &buffb,1000);`
	`      /* X */`
	`}`

Figure 3.1

4. The FreeRTOS generic task list implementation can be found in Exam Notes 2008.

(a) What are the operations needed to implement a RTOS task READY list, and how are they used in the RTOS kernel?

[4]

(b) What is the *worst-case* time performance of these operations as implemented in FreeRTOS using its doubly-linked generic task list package? For each operation with data-dependent time quantify the dependence.

[8]

(c) Write pseudocode for an implementation of the necessary task READY list operations using singly-linked lists. For each operation contrast the performance of the FreeRTOS implementation with that of your new implementation.

[8]

5.

(a) What conditions on the resource dependency graph are necessary and sufficient for a system to be deadlocked? Illustrate how, writing code at the application level using the FreeRTOS semaphore API, deadlock can be detected and normal operation restored. State any properties of the application that are required for this method to work. How instead can deadlock be prevented when using FreeRTOS semaphores?

[8]

(b) Two tasks T1, T2 (where T1 has higher priority than T2) share exclusive resources R_A,R_B. Write pseudo-code using the FreeRTOS API to allow exclusive access of the two resources using:

(i) Semaphores. In this case your code must detect as many errors as possible.

(ii) Scheduler locking.

(iii) Interrupt locking.

Contrast the merits of your three solutions, and state under what circumstances interrupt latency is affected.

[6]

(c) Prove that priority ceiling protocol (PCP), as defined in Exam Notes 2008, eliminates deadlock.

[6]

6.

(a) Two RTOS kernel implementations A & B have performance as specified in *Figure 6.1*. Determine, stating clearly your reasons, which you would consider the better choice for a real-time embedded system. What characteristics of an application, if any, would change this choice?

[4]

(b) Specify an Event Register API which allows inter-task communication (communication with ISRs need not be considered). Illustrate, with appropriate pseudocode, how a *single* three-task synchronisation barrier could be implemented using this API. Indicate the state changes (if necessary) to allow correct subsequent barrier.

[4]

(c) Write pseudocode using the event register API that will create a three-task synchronisation barrier which resets all internal state so that it can be executed repeatedly.

[4]

(d) The Event Register API used in part b is implemented with code which has a maximum critical section length of N*10us, where N is the number of waiting tasks. The critical section is implemented by disabling interrupts. It is proposed to add the code to kernels A and B. Discuss all the performance costs of this addition in each case both for an arbitrary application and for the code described in your answer to part b.

[4]

(d) Describe one way in which the API implementation in part c could be rewritten to reduce interrupt latency.

[4]

Kernel	Average interrupt latency	Maximum interrupt latency	Average task switch time	Maximum task switch time
A	3 us	30 us	50 us	70 us
B	10 us	25 us	65 us	70 us

Figure 6.1

[END]

RTOS EXAM NOTES 2008

Priority Ceiling Protocol definition

B. Definition

Having illustrated the basic idea of the priority ceiling protocol and its properties, we now present its definition.

1) Job J, which has the highest priority among the jobs ready to run, is assigned the processor, and let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by jobs other than job J. Before job J enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. Job J will be blocked and the lock on S will be denied, if the priority of job J is not higher than the priority ceiling of semaphore S^*.[4] In this case, job J is said to be blocked on semaphore S^* and to be blocked by the job which holds the lock on S^*. Otherwise, job J will obtain the lock on semaphore S and enter its critical section. When a job J exits its critical section, the binary semaphore associated with the critical section will be unlocked and the highest priority job, if any, blocked by job J will be awakened.

2) A job J uses its assigned priority, unless it is in its critical section and blocks higher priority jobs. If job J blocks higher priority jobs, J *inherits* P_H, the highest priority of the jobs blocked by J. When J exits a critical section, it resumes the priority it had at the point of entry into the critical section.[5] Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of previous priority must be indivisible.

3) A job J, when it does not attempt to enter a critical section, can preempt another job J_L if its priority is higher than the priority, inherited or assigned, at which job J_L is executing.

Task.h

```c
typedef void * xTaskHandle;
#define taskYIELD()                     portYIELD()
#define taskENTER_CRITICAL()            portENTER_CRITICAL()
#define taskEXIT_CRITICAL()             portEXIT_CRITICAL()
#define taskDISABLE_INTERRUPTS()        portDISABLE_INTERRUPTS()
#define taskENABLE_INTERRUPTS()         portENABLE_INTERRUPTS()

/*-----------------------------------------------------------
 * TASK CREATION API
 *----------------------------------------------------------*/

signed portBASE_TYPE xTaskCreate(   pdTASK_CODE pvTaskCode, const signed portCHAR * const pcName,
                                    unsigned portSHORT usStackDepth, void *pvParameters,
                                    unsigned portBASE_TYPE uxPriority, xTaskHandle *pvCreatedTask );

void vTaskDelete( xTaskHandle pxTask );

/*-----------------------------------------------------------
 * TASK CONTROL API
 *----------------------------------------------------------*/

void vTaskDelay( portTickType xTicksToDelay );
void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement );
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
void vTaskSuspend( xTaskHandle pxTaskToSuspend );
void vTaskResume( xTaskHandle pxTaskToResume );
portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume );

/*-----------------------------------------------------------
 * SCHEDULER CONTROL
 *----------------------------------------------------------*/

void vTaskStartScheduler( void );
void vTaskEndScheduler( void );
void vTaskSuspendAll( void );
signed portBASE_TYPE xTaskResumeAll( void );

/*-----------------------------------------------------------
 * TASK UTILITIES
 *----------------------------------------------------------*/

portTickType xTaskGetTickCount( void );
unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
void vTaskPlaceOnEventList( xList *pxEventList, portTickType xTicksToWait );
signed portBASE_TYPE xTaskRemoveFromEventList( const xList *pxEventList );
void vTaskCleanUpResources( void );
inline void vTaskSwitchContext( void );
xTaskHandle xTaskGetCurrentTaskHandle( void );
```

Semaphr.c

```c
#define vSemaphoreCreateBinary( xSemaphore )        {                                           \
        xSemaphore = xQueueCreate( ( unsigned portCHAR ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH );      \
        if( xSemaphore != NULL )                                                                   \
        {                                                                                          \
                xSemaphoreGive( xSemaphore );                                                      \
        }                                                                                          \
}

#define xSemaphoreTake( xSemaphore, xBlockTime )                                                   \
                        xQueueReceive( ( xQueueHandle ) xSemaphore, NULL, xBlockTime )

#define xSemaphoreGive( xSemaphore ) xQueueSend( ( xQueueHandle ) xSemaphore, NULL, semGIVE_BLOCK_TIME )

#define xSemaphoreGiveFromISR( xSemaphore, xTaskPreviouslyWoken )                                  \
                        xQueueSendFromISR( ( xQueueHandle ) xSemaphore, NULL, xTaskPreviouslyWoken )
```

From Task.h - related to lists package

```
signed portBASE_TYPE xTaskRemoveFromEventList( const xList *pxEventList )
{
tskTCB *pxUnblockedTCB;
portBASE_TYPE xReturn;

	/* THIS FUNCTION MUST BE CALLED WITH INTERRUPTS DISABLED OR THE
	SCHEDULER SUSPENDED.  It can also be called from within an ISR. */

	/* The event list is sorted in priority order, so we can remove the
	first in the list, remove the TCB from the delayed list, and add
	it to the ready list.

	If an event is for a queue that is locked then this function will never
	get called - the lock count on the queue will get modified instead.  This
	means we can always expect exclusive access to the event list here. */
	pxUnblockedTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
	vListRemove( &( pxUnblockedTCB->xEventListItem ) );

	if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
	{
		vListRemove( &( pxUnblockedTCB->xGenericListItem ) );
		prvAddTaskToReadyQueue( pxUnblockedTCB );
	}
	else
	{
		/* We cannot access the delayed or ready lists, so will hold this
		task pending until the scheduler is resumed. */
		vListInsertEnd( ( xList * ) &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
	}

	if( pxUnblockedTCB->uxPriority >= pxCurrentTCB->uxPriority )
	{
		/* Return true if the task removed from the event list has
		a higher priority than the calling task.  This allows
		the calling task to know if it should force a context
		switch now. */
		xReturn = pdTRUE;
	}
	else
	{
		xReturn = pdFALSE;
	}

	return xReturn;
}
```

List.h

```c
/*
 * Definition of the only type of object that a list can contain.
 */
struct xLIST_ITEM
{
	portTickType xItemValue;			/*< The value being listed.  In most cases this is
							used to sort the list in descending order. */
	volatile struct xLIST_ITEM * pxNext;	/*< Pointer to the next xListItem in the list. */
	volatile struct xLIST_ITEM * pxPrevious;/*< Pointer to the previous xListItem in the list. */
	void * pvOwner;		/*< Pointer to the object (normally a TCB) that contains the list item. */
	void * pvContainer;	/*< Pointer to the list in which this list item is placed (if any). */
};
typedef struct xLIST_ITEM xListItem;	/* For some reason lint wants this as two separate definitions. */

struct xMINI_LIST_ITEM
{
	portTickType xItemValue;
	volatile struct xLIST_ITEM *pxNext;
	volatile struct xLIST_ITEM *pxPrevious;
};
typedef struct xMINI_LIST_ITEM xMiniListItem;

/*
 * Definition of the type of queue used by the scheduler.
 */
typedef struct xLIST
{
	volatile unsigned portBASE_TYPE uxNumberOfItems;
	volatile xListItem * pxIndex;		/*  Used to walk through the list */
	volatile xMiniListItem xListEnd;	/*  List item that contains the maximum possible item value */
} xList;

#define listSET_LIST_ITEM_OWNER( pxListItem, pxOwner ) ( pxListItem )->pvOwner = ( void * ) pxOwner

#define listSET_LIST_ITEM_VALUE( pxListItem, xValue )		( pxListItem )->xItemValue = xValue

#define listGET_LIST_ITEM_VALUE( pxListItem )			( ( pxListItem )->xItemValue )

#define listLIST_IS_EMPTY( pxList )	( ( pxList )->uxNumberOfItems == ( unsigned portBASE_TYPE ) 0 )

#define listCURRENT_LIST_LENGTH( pxList )		( ( pxList )->uxNumberOfItems )

#define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList )						\
	/* Increment the index to the next item and return the item, ensuring */		\
	/* we don't return the marker used at the end of the list.  */			\
	( pxList )->pxIndex = ( pxList )->pxIndex->pxNext;					\
	if( ( pxList )->pxIndex == ( xListItem * ) &( ( pxList )->xListEnd ) )		\
	{											\
		( pxList )->pxIndex = ( pxList )->pxIndex->pxNext;				\
	}											\
	pxTCB = ( pxList )->pxIndex->pvOwner


#define listGET_OWNER_OF_HEAD_ENTRY( pxList )	( ( pxList->uxNumberOfItems != ( unsigned portBASE_TYPE ) 0
) ? ( (&( pxList->xListEnd ))->pxNext->pvOwner ) : ( NULL ) )

#define listIS_CONTAINED_WITHIN( pxList, pxListItem ) ( ( pxListItem )->pvContainer == ( void * ) pxList )

void vListInitialise( xList *pxList );

void vListInitialiseItem( xListItem *pxItem );

void vListInsert( xList *pxList, xListItem *pxNewListItem );

void vListInsertEnd( xList *pxList, xListItem *pxNewListItem );

void vListRemove( xListItem *pxItemToRemove );
```

List.c

```
#include <stdlib.h>
#include "FreeRTOS.h"
#include "list.h"

/*-----------------------------------------------------------
 * PUBLIC LIST API documented in list.h
 *----------------------------------------------------------*/

void vListInitialise( xList *pxList )
{
	/* The list structure contains a list item which is used to mark the end of the list.  To initialise
	   the list the list end is inserted as the only list entry. */
	pxList->pxIndex = ( xListItem * ) &( pxList->xListEnd );

	/* The list end value is the highest possible value in the list to ensure it
		remains at the end of the list. */
	pxList->xListEnd.xItemValue = portMAX_DELAY;

	/* The list end next and previous pointers point to itself so we know when the list is empty. */
	pxList->xListEnd.pxNext = ( xListItem * ) &( pxList->xListEnd );
	pxList->xListEnd.pxPrevious = ( xListItem * ) &( pxList->xListEnd );

	pxList->uxNumberOfItems = 0;
}

void vListInitialiseItem( xListItem *pxItem )
{
	/* Make sure the list item is not recorded as being on a list. */
	pxItem->pvContainer = NULL;
}

void vListInsertEnd( xList *pxList, xListItem *pxNewListItem )
{
volatile xListItem * pxIndex;

	/* Insert a new list item into pxList, but rather than sort the list, makes the new list item the
last
	   item to be removed by a call to pvListGetOwnerOfNextEntry.  This means it has to be the item
	   pointed to by the pxIndex member. */
	pxIndex = pxList->pxIndex;

	pxNewListItem->pxNext = pxIndex->pxNext;
	pxNewListItem->pxPrevious = pxList->pxIndex;
	pxIndex->pxNext->pxPrevious = ( volatile xListItem * ) pxNewListItem;
	pxIndex->pxNext = ( volatile xListItem * ) pxNewListItem;
	pxList->pxIndex = ( volatile xListItem * ) pxNewListItem;

	/* Remember which list the item is in. */
	pxNewListItem->pvContainer = ( void * ) pxList;

	( pxList->uxNumberOfItems )++;
}
```

```
void vListInsert( xList *pxList, xListItem *pxNewListItem )
{
volatile xListItem *pxIterator;
portTickType xValueOfInsertion;

        /* Insert the new list item into the list, sorted in ulListItem order. */
        xValueOfInsertion = pxNewListItem->xItemValue;

     /* If the list already contains a list item with the same item value then the new list item should be
      placed after it.  This ensures that TCB's which are stored in ready lists (all of which have the same
        ulListItem value) get an equal share of the CPU.  However, if the xItemValue is the same as the back
        marker the iteration loop below will not end.  This means we need to guard against this by checking
        the value first and modifying the algorithm slightly if necessary. */
        if( xValueOfInsertion == portMAX_DELAY )
        {
                pxIterator = pxList->xListEnd.pxPrevious;
        }
        else
        {
                for( pxIterator = ( xListItem * ) &( pxList->xListEnd );
                      pxIterator->pxNext->xItemValue <= xValueOfInsertion;
                          pxIterator = pxIterator->pxNext )
                {
                /* There is nothing to do here, we are just iterating to the wanted insertion position. */
                }
        }

        pxNewListItem->pxNext = pxIterator->pxNext;
        pxNewListItem->pxNext->pxPrevious = ( volatile xListItem * ) pxNewListItem;
        pxNewListItem->pxPrevious = pxIterator;
        pxIterator->pxNext = ( volatile xListItem * ) pxNewListItem;

        /* Remember which list the item is in.  This allows fast removal of the item later. */
        pxNewListItem->pvContainer = ( void * ) pxList;

        ( pxList->uxNumberOfItems )++;
}

void vListRemove( xListItem *pxItemToRemove )
{
        xList * pxList;
        pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
        pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;

        /* The list item knows which list it is in.  Obtain the list from the list item. */
        pxList = ( xList * ) pxItemToRemove->pvContainer;

        /* Make sure the index is left pointing to a valid item. */
        if( pxList->pxIndex == pxItemToRemove )
        {
                pxList->pxIndex = pxItemToRemove->pxPrevious;
        }

        pxItemToRemove->pvContainer = NULL;
        ( pxList->uxNumberOfItems )--;
}
/*-----------------------------------------------------------*/
```

Queue.h

```
typedef void * xQueueHandle;

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

signed portBASE_TYPE xQueueSend( xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait );

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

unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );

void vQueueDelete( xQueueHandle xQueue );

signed portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue, const void *pvItemToQueue, signed portBASE_TYPE
xTaskPreviouslyWoken );

signed portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue, void *pvBuffer, signed portBASE_TYPE
*pxTaskWoken );
```

Queue.c

```
/*-----------------------------------------------------------
 * PUBLIC LIST API documented in list.h
 *----------------------------------------------------------*/

/* Constants used with the cRxLock and cTxLock structure members. */
#define queueUNLOCKED  ( ( signed portBASE_TYPE ) -1 )

/*
 * Definition of the queue used by the scheduler.
 * Items are queued by copy, not reference.
 */
typedef struct QueueDefinition
{
        signed portCHAR *pcHead;  /*< Points to the beginning of the queue storage area. */
        signed portCHAR *pcTail;  /*< Points to the byte at the end of the queue storage area.
                                      Once more byte is allocated than necessary to store the queue items,
                                     this is used as a marker. */

        signed portCHAR *pcWriteTo;    /*< Points to the free next place in the storage area. */
        signed portCHAR *pcReadFrom;   /*< Points to the last place that a queued item was read from. */

        xList xTasksWaitingToSend;     /*< List of tasks that are blocked waiting to post onto this queue.
                                                Stored in priority order. */
        xList xTasksWaitingToReceive;   /*< List of tasks that are blocked waiting to
                                              read from this queue.  Stored in priority order. */

        unsigned portBASE_TYPE uxMessagesWaiting;/*< The number of items currently in the queue. */
        unsigned portBASE_TYPE uxLength;       /*< The length of the queue defined as the number
                                               of items it will hold, not the number of bytes. */
        unsigned portBASE_TYPE uxItemSize;     /*< The size of each items that the queue will hold. */

        signed portBASE_TYPE xRxLock;          /*< Stores the number of items received from the queue
                                               (removed from the queue) while the queue was locked.
                                               Set to queueUNLOCKED when the queue is not locked. */
        signed portBASE_TYPE xTxLock;          /*< Stores the number of items transmitted to the queue
                                               (added to the queue) while the queue was locked.
                                               Set to queueUNLOCKED when the queue is not locked. */
} xQUEUE;
/*-----------------------------------------------------------*/

/*
 * Inside this file xQueueHandle is a pointer to a xQUEUE structure.
 * To keep the definition private the API header file defines it as a
 * pointer to void.
 */
typedef xQUEUE * xQueueHandle;
```

```
/*
 * Unlocks a queue locked by a call to prvLockQueue.  Locking a queue does not
 * prevent an ISR from adding or removing items to the queue, but does prevent
 * an ISR from removing tasks from the queue event lists.  If an ISR finds a
 * queue is locked it will instead increment the appropriate queue lock count
 * to indicate that a task may require unblocking.  When the queue in unlocked
 * these lock counts are inspected, and the appropriate action taken.
 */
static signed portBASE_TYPE prvUnlockQueue( xQueueHandle pxQueue );

/*
 * Uses a critical section to determine if there is any data in a queue.
 *
 * @return pdTRUE if the queue contains no items, otherwise pdFALSE.
 */
static signed portBASE_TYPE prvIsQueueEmpty( const xQueueHandle pxQueue );

/*
 * Uses a critical section to determine if there is any space in a queue.
 *
 * @return pdTRUE if there is no space, otherwise pdFALSE;
 */
static signed portBASE_TYPE prvIsQueueFull( const xQueueHandle pxQueue );

/*
 * Macro that copies an item into the queue.  This is done by copying the item
 * byte for byte, not by reference.  Updates the queue state to ensure it's
 * integrity after the copy.
 */
#define prvCopyQueueData( pxQueue, pvItemToQueue )                                                      \
{                                                                                                       \
	memcpy( ( void * ) pxQueue->pcWriteTo, pvItemToQueue, ( unsigned ) pxQueue->uxItemSize );   \
	++( pxQueue->uxMessagesWaiting );                                                               \
	pxQueue->pcWriteTo += pxQueue->uxItemSize;                                                      \
	if( pxQueue->pcWriteTo >= pxQueue->pcTail )                                                     \
	{                                                                                               \
		pxQueue->pcWriteTo = pxQueue->pcHead;                                                   \
	}                                                                                               \
}

/*
 * Macro to mark a queue as locked.  Locking a queue prevents an ISR from accessing the queue event lists.
 */
#define prvLockQueue( pxQueue )                 \
{                                               \
	taskENTER_CRITICAL();                   \
		++( pxQueue->xRxLock );         \
		++( pxQueue->xTxLock );         \
	taskEXIT_CRITICAL();                    \
}
/*-----------------------------------------------------------
 * PUBLIC QUEUE MANAGEMENT API documented in queue.h
 *----------------------------------------------------------*/

xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize )
{
xQUEUE *pxNewQueue;
size_t xQueueSizeInBytes;

	/* Allocate the new queue structure. */
	if( uxQueueLength > ( unsigned portBASE_TYPE ) 0 )
	{
		pxNewQueue = ( xQUEUE * ) pvPortMalloc( sizeof( xQUEUE ) );
		if( pxNewQueue != NULL )
		{
			/* Create the list of pointers to queue items.  The queue is one byte
			longer than asked for to make wrap checking easier/faster. */
			xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize ) + ( size_t ) 1;

			pxNewQueue->pcHead = ( signed portCHAR * ) pvPortMalloc( xQueueSizeInBytes );
			if( pxNewQueue->pcHead != NULL )
			{
				/* Initialise the queue members as described above where the
				queue type is defined. */
				pxNewQueue->pcTail = pxNewQueue->pcHead + ( uxQueueLength * uxItemSize );
				pxNewQueue->uxMessagesWaiting = 0;
				pxNewQueue->pcWriteTo = pxNewQueue->pcHead;
				pxNewQueue->pcReadFrom = pxNewQueue->pcHead + ( ( uxQueueLength - 1 ) *
										uxItemSize );
				pxNewQueue->uxLength = uxQueueLength;
				pxNewQueue->uxItemSize = uxItemSize;
				pxNewQueue->xRxLock = queueUNLOCKED;
				pxNewQueue->xTxLock = queueUNLOCKED;
```

```
                        /* Likewise ensure the event queues start with the correct state. */
                        vListInitialise( &( pxNewQueue->xTasksWaitingToSend ) );
                        vListInitialise( &( pxNewQueue->xTasksWaitingToReceive ) );

                        return  pxNewQueue;
                    }
                    else
                    {
                        vPortFree( pxNewQueue );
                    }
            }
    }

    /* Will only reach here if we could not allocate enough memory or no memory
    was required. */
    return NULL;
}

signed portBASE_TYPE xQueueSend( xQueueHandle pxQueue, const void *pvItemToQueue, portTickType xTicksToWait )
{
signed portBASE_TYPE xReturn;

    /* Make sure other tasks do not access the queue. */
    vTaskSuspendAll();

    /* Make sure interrupts do not access the queue event list. */
    prvLockQueue( pxQueue );

    /* If the queue is already full we may have to block. */
    if( prvIsQueueFull( pxQueue ) )
    {
        /* The queue is full - do we want to block or just leave without
        posting? */
        if( xTicksToWait > ( portTickType ) 0 )
        {
    /* We are going to place ourselves on the xTasksWaitingToSend event list, and will get woken should
    the delay expire, or space become available on the queue. As detailed above we do not require mutual
    exclusion on the event list as nothing else can modify it or the ready lists while we have the
    scheduler suspended and queue locked.

    It is possible that an ISR has removed data from the queue since we checked if any was available.  If
    this is the case then the data will have been copied from the queue, and the queue variables updated,
    but the event list will not yet have been checked to see if anything is waiting as the queue is
    locked. */
                vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToSend ), xTicksToWait );

    /* Force a context switch now as we are blocked.  We can do this from within a critical section as the
    task we are switching to has its own context.  When we return here (i.e. we unblock) we will leave the
    critical section as normal.

    It is possible that an ISR has caused an event on an unrelated and unlocked queue.  If this was the
    case then the event list for that queue will have been updated but the ready lists left unchanged -
    instead the readied task will have been added to the pending ready list. */
                taskENTER_CRITICAL();
                {
    /* We can safely unlock the queue and scheduler here as interrupts are disabled.  We must not yield
    with anything locked, but we can yield from within a critical section.

    Tasks that have been placed on the pending ready list cannot be tasks that are waiting for events on
    this queue.  See in comment xTaskRemoveFromEventList(). */
                        prvUnlockQueue( pxQueue );

                        /* Resuming the scheduler may cause a yield.  If so then there
                        is no point yielding again here. */
                        if( !xTaskResumeAll() )
                        {
                            taskYIELD();
                        }

    /* Before leaving the critical section we have to ensure exclusive access again. */
                        vTaskSuspendAll();
                        prvLockQueue( pxQueue );
                }
                taskEXIT_CRITICAL();
        }
    }

    /* When we are here it is possible that we unblocked as space became available on the queue.
    It is also possible that an ISR posted to the queue since we left the critical section, so it may be
    that again there is no space.  This would only happen if a task and ISR post onto the same queue. */
    taskENTER_CRITICAL();
    {
```

```
                if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
                {
                        /* There is room in the queue, copy the data into the queue. */
                        prvCopyQueueData( pxQueue, pvItemToQueue );
                        xReturn = pdPASS;

                        /* Update the TxLock count so prvUnlockQueue knows to check for
                        tasks waiting for data to become available in the queue. */
                        ++( pxQueue->xTxLock );
                }
                else
                {
                        xReturn = errQUEUE_FULL;
                }
        }
        taskEXIT_CRITICAL();

        /* We no longer require exclusive access to the queue.  prvUnlockQueue will remove any tasks suspended
        on a receive if either this function or an ISR has posted onto the queue. */
        if( prvUnlockQueue( pxQueue ) )
        {
        /* Resume the scheduler - making ready any tasks that were woken by an event while the scheduler was
        locked.  Resuming the scheduler may cause a yield, in which case there is no point yielding again
        here. */
                if( !xTaskResumeAll() )
                {
                        taskYIELD();
                }
        }
        else
        {
        /* Resume the scheduler - making ready any tasks that were woken
        by an event while the scheduler was locked. */
                xTaskResumeAll();
        }

        return xReturn;
}

signed portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue, const void *pvItemToQueue, signed portBASE_TYPE
xTaskPreviouslyWoken )
{
        /* Similar to xQueueSend, except we don't block if there is no room in the queue.  Also we don't
        directly wake a task that was blocked on a queue read, instead we return a flag to say whether a
        context switch is required or not (i.e. has a task with a higher priority than us been woken by this
        post). */
        if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
        {
                prvCopyQueueData( pxQueue, pvItemToQueue );

                /* If the queue is locked we do not alter the event list.  This will
                be done when the queue is unlocked later. */
                if( pxQueue->xTxLock == queueUNLOCKED )
                {
                        /* We only want to wake one task per ISR, so check that a task has
                        not already been woken. */
                        if( !xTaskPreviouslyWoken )
                        {
                                if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) )
                                {
                                        if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) )
                                                        != pdFALSE )
                                        {
                                                /* The task waiting has a higher priority so record that a
                                                context switch is required. */
                                                return pdTRUE;
                                        }
                                }
                        }
                }
                else
                {
                /* Increment the lock count so the task that unlocks the queue
                knows that data was posted while it was locked. */
                        ++( pxQueue->xTxLock );
                }
        }

        return xTaskPreviouslyWoken;
}
```

```
signed portBASE_TYPE xQueueReceive( xQueueHandle pxQueue, void *pvBuffer, portTickType xTicksToWait )
{
signed portBASE_TYPE xReturn;

	/* This function is very similar to xQueueSend().  See comments within
	xQueueSend() for a more detailed explanation.*/

	/* Make sure other tasks do not access the queue. */
	vTaskSuspendAll();

	/* Make sure interrupts do not access the queue. */
	prvLockQueue( pxQueue );

	/* If there are no messages in the queue we may have to block. */
	if( prvIsQueueEmpty( pxQueue ) )
	{
		/* There are no messages in the queue, do we want to block or just leave with nothing? */
		if( xTicksToWait > ( portTickType ) 0 )
		{
			vTaskPlaceOnEventList( &( pxQueue->xTasksWaitingToReceive ), xTicksToWait );
			taskENTER_CRITICAL();
			{
				prvUnlockQueue( pxQueue );
				if( !xTaskResumeAll() )
				{
					taskYIELD();
				}

				vTaskSuspendAll();
				prvLockQueue( pxQueue );
			}
			taskEXIT_CRITICAL();
		}
	}

	taskENTER_CRITICAL();
	{
		if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
		{
			pxQueue->pcReadFrom += pxQueue->uxItemSize;
			if( pxQueue->pcReadFrom >= pxQueue->pcTail )
			{
				pxQueue->pcReadFrom = pxQueue->pcHead;
			}
			--( pxQueue->uxMessagesWaiting );
			memcpy( ( void * ) pvBuffer, ( void * ) pxQueue->pcReadFrom,
								( unsigned ) pxQueue->uxItemSize );

			/* Increment the lock count so prvUnlockQueue knows to check for
			tasks waiting for space to become available on the queue. */
			++( pxQueue->xRxLock );
			xReturn = pdPASS;
		}
		else
		{
			xReturn = pdFAIL;
		}
	}
	taskEXIT_CRITICAL();

	/* We no longer require exclusive access to the queue. */
	if( prvUnlockQueue( pxQueue ) )
	{
		if( !xTaskResumeAll() )
		{
			taskYIELD();
		}
	}
	else
	{
		xTaskResumeAll();
	}

	return xReturn;
}
```

```c
signed portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue, void *pvBuffer, signed portBASE_TYPE
*pxTaskWoken )
{
signed portBASE_TYPE xReturn;

	/* We cannot block from an ISR, so check there is data available. */
	if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
	{
		/* Copy the data from the queue. */
		pxQueue->pcReadFrom += pxQueue->uxItemSize;
		if( pxQueue->pcReadFrom >= pxQueue->pcTail )
		{
			pxQueue->pcReadFrom = pxQueue->pcHead;
		}
		--( pxQueue->uxMessagesWaiting );
		memcpy( ( void * ) pvBuffer, ( void * ) pxQueue->pcReadFrom,
						( unsigned ) pxQueue->uxItemSize );

		/* If the queue is locked we will not modify the event list.  Instead we update the lock count
		so the task that unlocks the queue will know that an ISR has removed data while the queue was
		locked. */
		if( pxQueue->xRxLock == queueUNLOCKED )
		{
		/* We only want to wake one task per ISR, so check that a task has not already been woken. */
			if( !( *pxTaskWoken ) )
			{
				if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) )
				{
					if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) )
							!= pdFALSE )
					{
					/* The task waiting has a higher priority than us so
					force a context switch. */
						*pxTaskWoken = pdTRUE;
					}
				}
			}
		}
		else
		{
		/* Increment the lock count so the task that unlocks the queue
			knows that data was removed while it was locked. */
			++( pxQueue->xRxLock );
		}

		xReturn = pdPASS;
	}
	else
	{
		xReturn = pdFAIL;
	}

	return xReturn;
}

unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle pxQueue )
{
unsigned portBASE_TYPE uxReturn;

	taskENTER_CRITICAL();
		uxReturn = pxQueue->uxMessagesWaiting;
	taskEXIT_CRITICAL();

	return uxReturn;
}

void vQueueDelete( xQueueHandle pxQueue )
{
	vPortFree( pxQueue->pcHead );
	vPortFree( pxQueue );
}
```

```
static signed portBASE_TYPE prvUnlockQueue( xQueueHandle pxQueue )
{
signed portBASE_TYPE xYieldRequired = pdFALSE;

	/* THIS FUNCTION MUST BE CALLED WITH THE SCHEDULER SUSPENDED. */

	/* The lock counts contains the number of extra data items placed or
	removed from the queue while the queue was locked.  When a queue is
	locked items can be added or removed, but the event lists cannot be
	updated. */
	taskENTER_CRITICAL();
	{
		--( pxQueue->xTxLock );

		/* See if data was added to the queue while it was locked. */
		if( pxQueue->xTxLock > queueUNLOCKED )
		{
			pxQueue->xTxLock = queueUNLOCKED;

			/* Data was posted while the queue was locked.  Are any tasks
			blocked waiting for data to become available? */
			if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToReceive ) ) )
			{
				/* Tasks that are removed from the event list will get added to
				the pending ready list as the scheduler is still suspended. */
				if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToReceive ) ) != pdFALSE
)
				{
					/* The task waiting has a higher priority so record that a
					context switch is required. */
					xYieldRequired = pdTRUE;
				}
			}
		}
	}
	taskEXIT_CRITICAL();

	/* Do the same for the Rx lock. */
	taskENTER_CRITICAL();
	{
		--( pxQueue->xRxLock );

		if( pxQueue->xRxLock > queueUNLOCKED )
		{
			pxQueue->xRxLock = queueUNLOCKED;

			if( !listLIST_IS_EMPTY( &( pxQueue->xTasksWaitingToSend ) ) )
			{
				if( xTaskRemoveFromEventList( &( pxQueue->xTasksWaitingToSend ) ) != pdFALSE )
				{
					xYieldRequired = pdTRUE;
				}
			}
		}
	}
	taskEXIT_CRITICAL();

	return xYieldRequired;
}

static signed portBASE_TYPE prvIsQueueEmpty( const xQueueHandle pxQueue )
{
signed portBASE_TYPE xReturn;

	taskENTER_CRITICAL();
		xReturn = ( pxQueue->uxMessagesWaiting == ( unsigned portBASE_TYPE ) 0 );
	taskEXIT_CRITICAL();

	return xReturn;
}

static signed portBASE_TYPE prvIsQueueFull( const xQueueHandle pxQueue )
{
signed portBASE_TYPE xReturn;

	taskENTER_CRITICAL();
		xReturn = ( pxQueue->uxMessagesWaiting == pxQueue->uxLength );
	taskEXIT_CRITICAL();

	return xReturn;
}
```