

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 $tI > 50$ us, using inequalities on tI 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 tI 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 tI 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	tI 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
<pre>QueueSend(q1, &m1, 1000); QueueReceive(q1, &buffa, 1000); TaskDelay(1); /* X */</pre>	<pre>while (1) { QueueReceive(q1, &buffb, 1000); /* X */ }</pre>

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 $N \times 10\mu s$, 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 μs	30 μs	50 μs	70 μs
B	10 μs	25 μs	65 μs	70 μs

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

```

1  typedef void * xTaskHandle;
2
3  #define taskYIELD()          portYIELD()
4  #define taskENTER_CRITICAL() portENTER_CRITICAL()
5  #define taskEXIT_CRITICAL()  portEXIT_CRITICAL()
6  #define taskDISABLE_INTERRUPTS() portDISABLE_INTERRUPTS()
7  #define taskENABLE_INTERRUPTS() portENABLE_INTERRUPTS()
8
9
10 /*-----*/
11 * TASK CREATION API
12 *-----*/
13
14 signed portBASE_TYPE xTaskCreate(    pdTASK_CODE pvTaskCode, const signed portCHAR * const pcName,
15                                     unsigned portSHORT usStackDepth, void *pvParameters,
16                                     unsigned portBASE_TYPE uxPriority, xTaskHandle *pvCreatedTask );
17
18 void vTaskDelete( xTaskHandle pxTask );
19
20 /*-----*/
21 * TASK CONTROL API
22 *-----*/
23
24 void vTaskDelay( portTickType xTicksToDelay );
25 void vTaskDelayUntil( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement );
26 unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
27 void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority );
28 void vTaskSuspend( xTaskHandle pxTaskToSuspend );
29 void vTaskResume( xTaskHandle pxTaskToResume );
30 portBASE_TYPE xTaskResumeFromISR( xTaskHandle pxTaskToResume );
31
32 /*-----*/
33 * SCHEDULER CONTROL
34 *-----*/
35
36 void vTaskStartScheduler( void );
37 void vTaskEndScheduler( void );
38 void vTaskSuspendAll( void );
39 signed portBASE_TYPE xTaskResumeAll( void );
40
41 /*-----*/
42 * TASK UTILITIES
43 *-----*/
44
45 portTickType xTaskGetTickCount( void );
46 unsigned portBASE_TYPE uxTaskGetNumberOfTasks( void );
47 void vTaskPlaceOnEventList( xList *pxEventList, portTickType xTicksToWait );
48 signed portBASE_TYPE xTaskRemoveFromEventList( const xList *pxEventList );
49 void vTaskCleanUpResources( void );
50 inline void vTaskSwitchContext( void );
51 xTaskHandle xTaskGetCurrentTaskHandle( void );
52
53
54

```

Semaphr.c

```

56
57 #define vSemaphoreCreateBinary( xSemaphore ) { \
58     xSemaphore = xQueueCreate( ( unsigned portCHAR ) 1, semSEMAPHORE_QUEUE_ITEM_LENGTH ); \
59     if( xSemaphore != NULL ) \
60     { \
61         xSemaphoreGive( xSemaphore ); \
62     } \
63 }
64
65 #define xSemaphoreTake( xSemaphore, xBlockTime ) \
66     xQueueReceive( ( xQueueHandle ) xSemaphore, NULL, xBlockTime )
67
68 #define xSemaphoreGive( xSemaphore ) xQueueSend( ( xQueueHandle ) xSemaphore, NULL, semGIVE_BLOCK_TIME )
69
70 #define xSemaphoreGiveFromISR( xSemaphore, xTaskPreviouslyWoken ) \
71     xQueueSendFromISR( ( xQueueHandle ) xSemaphore, NULL, xTaskPreviouslyWoken )

```

TASK & SEMAPHORE API

From Task.h - related to lists package

```

100 signed portBASE_TYPE xTaskRemoveFromEventList( const xList *pxEventList )
101 {
102     tskTCB *pxUnblockedTCB;
103     portBASE_TYPE xReturn;
104
105     /* THIS FUNCTION MUST BE CALLED WITH INTERRUPTS DISABLED OR THE
106        SCHEDULER SUSPENDED. It can also be called from within an ISR. */
107
108     /* The event list is sorted in priority order, so we can remove the
109        first in the list, remove the TCB from the delayed list, and add
110        it to the ready list.
111
112        If an event is for a queue that is locked then this function will never
113        get called - the lock count on the queue will get modified instead. This
114        means we can always expect exclusive access to the event list here. */
115     pxUnblockedTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
116     vListRemove( &( pxUnblockedTCB->xEventListItem ) );
117
118     if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
119     {
120         vListRemove( &( pxUnblockedTCB->xGenericListItem ) );
121         prvAddTaskToReadyQueue( pxUnblockedTCB );
122     }
123     else
124     {
125         /* We cannot access the delayed or ready lists, so will hold this
126            task pending until the scheduler is resumed. */
127         vListInsertEnd( ( xList * ) &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
128     }
129
130     if( pxUnblockedTCB->uxPriority >= pxCurrentTCB->uxPriority )
131     {
132         /* Return true if the task removed from the event list has
133            a higher priority than the calling task. This allows
134            the calling task to know if it should force a context
135            switch now. */
136         xReturn = pdTRUE;
137     }
138     else
139     {
140         xReturn = pdFALSE;
141     }
142
143     return xReturn;
144 }
145

```

List.h

```

146
147 /*
148  * Definition of the only type of object that a list can contain.
149  */
150 struct xLIST_ITEM
151 {
152     portTickType xItemValue;          /*< The value being listed. In most cases this is
153                                         used to sort the list in descending order. */
154     volatile struct xLIST_ITEM * pxNext; /*< Pointer to the next xListItem in the list. */
155     volatile struct xLIST_ITEM * pxPrevious; /*< Pointer to the previous xListItem in the list. */
156     void * pvOwner;                    /*< Pointer to the object (normally a TCB) that contains the list item. */
157     void * pvContainer;                /*< Pointer to the list in which this list item is placed (if any). */
158 };
159 typedef struct xLIST_ITEM xListItem; /* For some reason lint wants this as two separate definitions. */
160
161 struct xMINI_LIST_ITEM
162 {
163     portTickType xItemValue;
164     volatile struct xLIST_ITEM * pxNext;
165     volatile struct xLIST_ITEM * pxPrevious;
166 };
167 typedef struct xMINI_LIST_ITEM xMiniListItem;
168
169 /*
170  * Definition of the type of queue used by the scheduler.
171  */
172 typedef struct xLIST
173 {
174     volatile unsigned portBASE_TYPE uxNumberOfItems;
175     volatile xListItem * pxIndex;      /* Used to walk through the list */
176     volatile xMiniListItem xListEnd;   /* List item that contains the maximum possible item value */
177 } xList;
178
179 #define listSET_LIST_ITEM_OWNER( pxListItem, pxOwner ) ( pxListItem )->pvOwner = ( void * ) pxOwner
180
181 #define listSET_LIST_ITEM_VALUE( pxListItem, xValue ) ( pxListItem )->xItemValue = xValue
182
183 #define listGET_LIST_ITEM_VALUE( pxListItem ) ( ( pxListItem )->xItemValue )
184
185 #define listLIST_IS_EMPTY( pxList ) ( ( pxList )->uxNumberOfItems == ( unsigned portBASE_TYPE ) 0 )
186
187 #define listCURRENT_LIST_LENGTH( pxList ) ( ( pxList )->uxNumberOfItems )
188
189 #define listGET_OWNER_OF_NEXT_ENTRY( pxTCB, pxList ) \
190     /* Increment the index to the next item and return the item, ensuring */ \
191     /* we don't return the marker used at the end of the list. */ \
192     ( pxList )->pxIndex = ( pxList )->pxIndex->pxNext; \
193     if( ( pxList )->pxIndex == ( xListItem * ) &( ( pxList )->xListEnd ) ) \
194     { \
195         ( pxList )->pxIndex = ( pxList )->pxIndex->pxNext; \
196     } \
197     pxTCB = ( pxList )->pxIndex->pvOwner
198
199
200 #define listGET_OWNER_OF_HEAD_ENTRY( pxList ) ( ( pxList->uxNumberOfItems != ( unsigned portBASE_TYPE ) 0 \
201 ) ? ( &( ( pxList->xListEnd )->pxNext->pvOwner ) : ( NULL ) )
202
203 #define listIS_CONTAINED_WITHIN( pxList, pxListItem ) ( ( pxListItem )->pvContainer == ( void * ) pxList )
204
205 void vListInitialise( xList *pxList );
206
207 void vListInitialiseItem( xListItem *pxItem );
208
209 void vListInsert( xList *pxList, xListItem *pxNewListItem );
210
211 void vListInsertEnd( xList *pxList, xListItem *pxNewListItem );
212
213 void vListRemove( xListItem *pxItemToRemove );

```

```

214 List.c
215 #include <stdlib.h>
216 #include "FreeRTOS.h"
217 #include "list.h"
218
219 /*-----
220  * PUBLIC LIST API documented in list.h
221  *-----*/
222
223 void vListInitialise( xList *pxList )
224 {
225     /* The list structure contains a list item which is used to mark the end of the list. To initialise
226        the list the list end is inserted as the only list entry. */
227     pxList->pxIndex = ( xListItem * ) &(amp; pxList->xListEnd );
228
229     /* The list end value is the highest possible value in the list to ensure it
230        remains at the end of the list. */
231     pxList->xListEnd.xItemValue = portMAX_DELAY;
232
233     /* The list end next and previous pointers point to itself so we know when the list is empty. */
234     pxList->xListEnd.pxNext = ( xListItem * ) &(amp; pxList->xListEnd );
235     pxList->xListEnd.pxPrevious = ( xListItem * ) &(amp; pxList->xListEnd );
236
237     pxList->uxNumberOfItems = 0;
238 }
239
240 void vListInitialiseItem( xListItem *pxItem )
241 {
242     /* Make sure the list item is not recorded as being on a list. */
243     pxItem->pvContainer = NULL;
244 }
245
246 void vListInsertEnd( xList *pxList, xListItem *pxNewListItem )
247 {
248     volatile xListItem * pxIndex;
249
250     /* Insert a new list item into pxList, but rather than sort the list, makes the new list item the
251        last
252        item to be removed by a call to pvListGetOwnerOfNextEntry. This means it has to be the item
253        pointed to by the pxIndex member. */
254     pxIndex = pxList->pxIndex;
255
256     pxNewListItem->pxNext = pxIndex->pxNext;
257     pxNewListItem->pxPrevious = pxList->pxIndex;
258     pxIndex->pxNext->pxPrevious = ( volatile xListItem * ) pxNewListItem;
259     pxIndex->pxNext = ( volatile xListItem * ) pxNewListItem;
260     pxList->pxIndex = ( volatile xListItem * ) pxNewListItem;
261
262     /* Remember which list the item is in. */
263     pxNewListItem->pvContainer = ( void * ) pxList;
264
265     ( pxList->uxNumberOfItems )++;
266 }
267

```

```

268 void vListInsert( xList *pxList, xListItem *pxNewListItem )
269 {
270     volatile xListItem *pxIterator;
271     portTickType xValueOfInsertion;
272
273     /* Insert the new list item into the list, sorted in ulListItem order. */
274     xValueOfInsertion = pxNewListItem->xItemValue;
275
276     /* If the list already contains a list item with the same item value then the new list item should be
277     placed after it. This ensures that TCB's which are stored in ready lists (all of which have the same
278     ulListItem value) get an equal share of the CPU. However, if the xItemValue is the same as the back
279     marker the iteration loop below will not end. This means we need to guard against this by checking
280     the value first and modifying the algorithm slightly if necessary. */
281     if( xValueOfInsertion == portMAX_DELAY )
282     {
283         pxIterator = pxList->xListEnd.pxPrevious;
284     }
285     else
286     {
287         for( pxIterator = ( xListItem * ) &(amp; pxList->xListEnd );
288             pxIterator->pxNext->xItemValue <= xValueOfInsertion;
289             pxIterator = pxIterator->pxNext )
290         {
291             /* There is nothing to do here, we are just iterating to the wanted insertion position. */
292         }
293     }
294
295     pxNewListItem->pxNext = pxIterator->pxNext;
296     pxNewListItem->pxNext->pxPrevious = ( volatile xListItem * ) pxNewListItem;
297     pxNewListItem->pxPrevious = pxIterator;
298     pxIterator->pxNext = ( volatile xListItem * ) pxNewListItem;
299
300     /* Remember which list the item is in. This allows fast removal of the item later. */
301     pxNewListItem->pvContainer = ( void * ) pxList;
302
303     ( pxList->uxNumberOfItems )++;
304 }
305
306 void vListRemove( xListItem *pxItemToRemove )
307 {
308     xList * pxList;
309     pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->pxPrevious;
310     pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->pxNext;
311
312     /* The list item knows which list it is in. Obtain the list from the list item. */
313     pxList = ( xList * ) pxItemToRemove->pvContainer;
314
315     /* Make sure the index is left pointing to a valid item. */
316     if( pxList->pxIndex == pxItemToRemove )
317     {
318         pxList->pxIndex = pxItemToRemove->pxPrevious;
319     }
320
321     pxItemToRemove->pvContainer = NULL;
322     ( pxList->uxNumberOfItems )--;
323 }
324 /*-----*/

```

Queue.h

```

301 typedef void * xQueueHandle;
302
303 xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize );
304
305 signed portBASE_TYPE xQueueSend( xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait );
306
307 signed portBASE_TYPE xQueueReceive( xQueueHandle xQueue, void *pvBuffer, portTickType xTicksToWait );
308
309 unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
310
311 void vQueueDelete( xQueueHandle xQueue );
312
313 signed portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue, const void *pvItemToQueue, signed portBASE_TYPE
314 xTaskPreviouslyWoken );
315
316 signed portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue, void *pvBuffer, signed portBASE_TYPE
317 *pxTaskWoken );
318

```

Queue.c

```

320
321 /*-----
322  * PUBLIC LIST API documented in list.h
323  *-----*/
324
325 /* Constants used with the cRxLock and cTxLock structure members. */
326 #define queueUNLOCKED ( ( signed portBASE_TYPE ) -1 )
327
328 /*
329  * Definition of the queue used by the scheduler.
330  * Items are queued by copy, not reference.
331  */
332 typedef struct QueueDefinition
333 {
334     signed portCHAR *pcHead; /*< Points to the beginning of the queue storage area. */
335     signed portCHAR *pcTail; /*< Points to the byte at the end of the queue storage area.
336                               Once more byte is allocated than necessary to store the queue items,
337                               this is used as a marker. */
338
339     signed portCHAR *pcWriteTo; /*< Points to the free next place in the storage area. */
340     signed portCHAR *pcReadFrom; /*< Points to the last place that a queued item was read from. */
341
342     xList xTasksWaitingToSend; /*< List of tasks that are blocked waiting to post onto this queue.
343                               Stored in priority order. */
344     xList xTasksWaitingToReceive; /*< List of tasks that are blocked waiting to
345                               read from this queue. Stored in priority order. */
346
347     unsigned portBASE_TYPE uxMessagesWaiting; /*< The number of items currently in the queue. */
348     unsigned portBASE_TYPE uxLength; /*< The length of the queue defined as the number
349                               of items it will hold, not the number of bytes. */
350     unsigned portBASE_TYPE uxItemSize; /*< The size of each items that the queue will hold. */
351
352     signed portBASE_TYPE xRxLock; /*< Stores the number of items received from the queue
353                               (removed from the queue) while the queue was locked.
354                               Set to queueUNLOCKED when the queue is not locked. */
355     signed portBASE_TYPE xTxLock; /*< Stores the number of items transmitted to the queue
356                               (added to the queue) while the queue was locked.
357                               Set to queueUNLOCKED when the queue is not locked. */
358 } xQUEUE;
359 /*-----*/
360
361 /*
362  * Inside this file xQueueHandle is a pointer to a xQUEUE structure.
363  * To keep the definition private the API header file defines it as a
364  * pointer to void.
365  */
366 typedef xQUEUE * xQueueHandle;
367

```



```

368 /*
369  * Unlocks a queue locked by a call to prvLockQueue. Locking a queue does not
370  * prevent an ISR from adding or removing items to the queue, but does prevent
371  * an ISR from removing tasks from the queue event lists. If an ISR finds a
372  * queue is locked it will instead increment the appropriate queue lock count
373  * to indicate that a task may require unblocking. When the queue is unlocked
374  * these lock counts are inspected, and the appropriate action taken.
375  */
376 static signed portBASE_TYPE prvUnlockQueue( xQueueHandle pxQueue );
377
378 /*
379  * Uses a critical section to determine if there is any data in a queue.
380  *
381  * @return pdTRUE if the queue contains no items, otherwise pdFALSE.
382  */
383 static signed portBASE_TYPE prvIsQueueEmpty( const xQueueHandle pxQueue );
384
385 /*
386  * Uses a critical section to determine if there is any space in a queue.
387  *
388  * @return pdTRUE if there is no space, otherwise pdFALSE;
389  */
390 static signed portBASE_TYPE prvIsQueueFull( const xQueueHandle pxQueue );
391
392 /*
393  * Macro that copies an item into the queue. This is done by copying the item
394  * byte for byte, not by reference. Updates the queue state to ensure it's
395  * integrity after the copy.
396  */
397 #define prvCopyQueueData( pxQueue, pvItemToQueue )
398 {
399     memcpy( ( void * ) pxQueue->pcWriteTo, pvItemToQueue, ( unsigned ) pxQueue->uxItemSize );
400     ++( pxQueue->uxMessagesWaiting );
401     pxQueue->pcWriteTo += pxQueue->uxItemSize;
402     if( pxQueue->pcWriteTo >= pxQueue->pcTail )
403     {
404         pxQueue->pcWriteTo = pxQueue->pcHead;
405     }
406 }
407
408 /*
409  * Macro to mark a queue as locked. Locking a queue prevents an ISR from accessing the queue event lists.
410  */
411 #define prvLockQueue( pxQueue )
412 {
413     taskENTER_CRITICAL();
414     ++( pxQueue->uxRxLock );
415     ++( pxQueue->uxTxLock );
416     taskEXIT_CRITICAL();
417 }
418
419 /*-----
420  * PUBLIC QUEUE MANAGEMENT API documented in queue.h
421  *-----*/
422
423 xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength, unsigned portBASE_TYPE uxItemSize )
424 {
425     xQUEUE *pxNewQueue;
426     size_t xQueueSizeInBytes;
427
428     /* Allocate the new queue structure. */
429     if( uxQueueLength > ( unsigned portBASE_TYPE ) 0 )
430     {
431         pxNewQueue = ( xQUEUE * ) pvPortMalloc( sizeof( xQUEUE ) );
432         if( pxNewQueue != NULL )
433         {
434             /* Create the list of pointers to queue items. The queue is one byte
435              longer than asked for to make wrap checking easier/faster. */
436             xQueueSizeInBytes = ( size_t ) ( uxQueueLength * uxItemSize ) + ( size_t ) 1;
437
438             pxNewQueue->pcHead = ( signed portCHAR * ) pvPortMalloc( xQueueSizeInBytes );
439             if( pxNewQueue->pcHead != NULL )
440             {
441                 /* Initialise the queue members as described above where the
442                  queue type is defined. */
443                 pxNewQueue->pcTail = pxNewQueue->pcHead + ( uxQueueLength * uxItemSize );
444                 pxNewQueue->uxMessagesWaiting = 0;
445                 pxNewQueue->pcWriteTo = pxNewQueue->pcHead;
446                 pxNewQueue->pcReadFrom = pxNewQueue->pcHead + ( ( uxQueueLength - 1 ) *
447                     uxItemSize );
448                 pxNewQueue->uxLength = uxQueueLength;
449                 pxNewQueue->uxItemSize = uxItemSize;
450                 pxNewQueue->uxRxLock = queueUNLOCKED;
451                 pxNewQueue->uxTxLock = queueUNLOCKED;

```

QUEUE PACKAGE

```

451                                     /* Likewise ensure the event queues start with the correct state. */
452                                     vListInitialise( &(amp; pxNewQueue->xTasksWaitingToSend ) );
453                                     vListInitialise( &(amp; pxNewQueue->xTasksWaitingToReceive ) );
454
455                                     return pxNewQueue;
456
457                                     }
458                                     else
459                                     {
460                                         vPortFree( pxNewQueue );
461                                     }
462                                     }
463
464                                     }
465                                     /* Will only reach here if we could not allocate enough memory or no memory
466                                     was required. */
467                                     return NULL;
468
469                                     }
470
471                                     signed portBASE_TYPE xQueueSend( xQueueHandle pxQueue, const void *pvItemToQueue, portTickType xTicksToWait )
472                                     {
473                                         signed portBASE_TYPE xReturn;
474
475                                         /* Make sure other tasks do not access the queue. */
476                                         vTaskSuspendAll();
477
478                                         /* Make sure interrupts do not access the queue event list. */
479                                         prvLockQueue( pxQueue );
480
481                                         /* If the queue is already full we may have to block. */
482                                         if( prvIsQueueFull( pxQueue ) )
483                                         {
484                                             /* The queue is full - do we want to block or just leave without
485                                             posting? */
486                                             if( xTicksToWait > ( portTickType ) 0 )
487                                             {
488                                                 /* We are going to place ourselves on the xTasksWaitingToSend event list, and will get woken should
489                                                 the delay expire, or space become available on the queue. As detailed above we do not require mutual
490                                                 exclusion on the event list as nothing else can modify it or the ready lists while we have the
491                                                 scheduler suspended and queue locked.
492
493                                                 It is possible that an ISR has removed data from the queue since we checked if any was available. If
494                                                 this is the case then the data will have been copied from the queue, and the queue variables updated,
495                                                 but the event list will not yet have been checked to see if anything is waiting as the queue is
496                                                 locked. */
497                                                 vTaskPlaceOnEventList( &(amp; pxQueue->xTasksWaitingToSend ), xTicksToWait );
498
499                                                 /* Force a context switch now as we are blocked. We can do this from within a critical section as the
500                                                 task we are switching to has its own context. When we return here (i.e. we unblock) we will leave the
501                                                 critical section as normal.
502
503                                                 It is possible that an ISR has caused an event on an unrelated and unlocked queue. If this was the
504                                                 case then the event list for that queue will have been updated but the ready lists left unchanged -
505                                                 instead the readied task will have been added to the pending ready list. */
506                                                 taskENTER_CRITICAL();
507                                                 {
508                                                     /* We can safely unlock the queue and scheduler here as interrupts are disabled. We must not yield
509                                                     with anything locked, but we can yield from within a critical section.
510
511                                                     Tasks that have been placed on the pending ready list cannot be tasks that are waiting for events on
512                                                     this queue. See in comment xTaskRemoveFromEventList(). */
513                                                     prvUnlockQueue( pxQueue );
514
515                                                     /* Resuming the scheduler may cause a yield. If so then there
516                                                     is no point yielding again here. */
517                                                     if( !xTaskResumeAll() )
518                                                     {
519                                                         taskYIELD();
520                                                     }
521                                                 }
522
523                                                 /* Before leaving the critical section we have to ensure exclusive access again. */
524                                                 vTaskSuspendAll();
525                                                 prvLockQueue( pxQueue );
526                                             }
527                                             taskEXIT_CRITICAL();
528                                         }
529
530                                         /* When we are here it is possible that we unblocked as space became available on the queue.
531                                         It is also possible that an ISR posted to the queue since we left the critical section, so it may be
532                                         that again there is no space. This would only happen if a task and ISR post onto the same queue. */
533                                         taskENTER_CRITICAL();

```

```

534     if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
535     {
536         /* There is room in the queue, copy the data into the queue. */
537         prvCopyQueueData( pxQueue, pvItemToQueue );
538         xReturn = pdPASS;
539
540         /* Update the TxLock count so prvUnlockQueue knows to check for
541         tasks waiting for data to become available in the queue. */
542         ++( pxQueue->xTxLock );
543     }
544     else
545     {
546         xReturn = errQUEUE_FULL;
547     }
548 }
549 taskEXIT_CRITICAL();
550
551 /* We no longer require exclusive access to the queue. prvUnlockQueue will remove any tasks suspended
552 on a receive if either this function or an ISR has posted onto the queue. */
553 if( prvUnlockQueue( pxQueue ) )
554 {
555     /* Resume the scheduler - making ready any tasks that were woken by an event while the scheduler was
556     locked. Resuming the scheduler may cause a yield, in which case there is no point yielding again
557     here. */
558     if( !xTaskResumeAll() )
559     {
560         taskYIELD();
561     }
562 }
563 else
564 {
565     /* Resume the scheduler - making ready any tasks that were woken
566     by an event while the scheduler was locked. */
567     xTaskResumeAll();
568 }
569
570 return xReturn;
571 }
572
573 signed portBASE_TYPE xQueueSendFromISR( xQueueHandle pxQueue, const void *pvItemToQueue, signed portBASE_TYPE
574 xTaskPreviouslyWoken )
575 {
576     /* Similar to xQueueSend, except we don't block if there is no room in the queue. Also we don't
577     directly wake a task that was blocked on a queue read, instead we return a flag to say whether a
578     context switch is required or not (i.e. has a task with a higher priority than us been woken by this
579     post). */
580     if( pxQueue->uxMessagesWaiting < pxQueue->uxLength )
581     {
582         prvCopyQueueData( pxQueue, pvItemToQueue );
583
584         /* If the queue is locked we do not alter the event list. This will
585         be done when the queue is unlocked later. */
586         if( pxQueue->xTxLock == queueUNLOCKED )
587         {
588             /* We only want to wake one task per ISR, so check that a task has
589             not already been woken. */
590             if( !xTaskPreviouslyWoken )
591             {
592                 if( !listLIST_IS_EMPTY( &(amp; pxQueue->xTasksWaitingToReceive) ) )
593                 {
594                     if( xTaskRemoveFromEventList( &(amp; pxQueue->xTasksWaitingToReceive) )
595                     != pdFALSE )
596                     {
597                         /* The task waiting has a higher priority so record that a
598                         context switch is required. */
599                         return pdTRUE;
600                     }
601                 }
602             }
603         }
604         else
605         {
606             /* Increment the lock count so the task that unlocks the queue
607             knows that data was posted while it was locked. */
608             ++( pxQueue->xTxLock );
609         }
610     }
611
612     return xTaskPreviouslyWoken;
613 }
614

```

```

615 signed portBASE_TYPE xQueueReceive( xQueueHandle pxQueue, void *pvBuffer, portTickType xTicksToWait )
616 {
617     signed portBASE_TYPE xReturn;
618
619     /* This function is very similar to xQueueSend(). See comments within
620     xQueueSend() for a more detailed explanation.*/
621
622     /* Make sure other tasks do not access the queue. */
623     vTaskSuspendAll();
624
625     /* Make sure interrupts do not access the queue. */
626     prvLockQueue( pxQueue );
627
628     /* If there are no messages in the queue we may have to block. */
629     if( prvIsQueueEmpty( pxQueue ) )
630     {
631         /* There are no messages in the queue, do we want to block or just leave with nothing? */
632         if( xTicksToWait > ( portTickType ) 0 )
633         {
634             vTaskPlaceOnEventList( &(amp; pxQueue->xTasksWaitingToReceive ), xTicksToWait );
635             taskENTER_CRITICAL();
636             {
637                 prvUnlockQueue( pxQueue );
638                 if( !xTaskResumeAll() )
639                 {
640                     taskYIELD();
641                 }
642
643                 vTaskSuspendAll();
644                 prvLockQueue( pxQueue );
645             }
646             taskEXIT_CRITICAL();
647         }
648     }
649
650     taskENTER_CRITICAL();
651     {
652         if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
653         {
654             pxQueue->pcReadFrom += pxQueue->uxItemSize;
655             if( pxQueue->pcReadFrom >= pxQueue->pcTail )
656             {
657                 pxQueue->pcReadFrom = pxQueue->pcHead;
658             }
659             --( pxQueue->uxMessagesWaiting );
660             memcpy( ( void * ) pvBuffer, ( void * ) pxQueue->pcReadFrom,
661                 ( unsigned ) pxQueue->uxItemSize );
662
663             /* Increment the lock count so prvUnlockQueue knows to check for
664             tasks waiting for space to become available on the queue. */
665             ++( pxQueue->xRxLock );
666             xReturn = pdPASS;
667         }
668         else
669         {
670             xReturn = pdFAIL;
671         }
672     }
673     taskEXIT_CRITICAL();
674
675     /* We no longer require exclusive access to the queue. */
676     if( prvUnlockQueue( pxQueue ) )
677     {
678         if( !xTaskResumeAll() )
679         {
680             taskYIELD();
681         }
682     }
683     else
684     {
685         xTaskResumeAll();
686     }
687
688     return xReturn;
689 }
690

```

```

691 signed portBASE_TYPE xQueueReceiveFromISR( xQueueHandle pxQueue, void *pvBuffer, signed portBASE_TYPE
692 *pxTaskWoken )
693 {
694     signed portBASE_TYPE xReturn;
695
696     /* We cannot block from an ISR, so check there is data available. */
697     if( pxQueue->uxMessagesWaiting > ( unsigned portBASE_TYPE ) 0 )
698     {
699         /* Copy the data from the queue. */
700         pxQueue->pcReadFrom += pxQueue->uxItemSize;
701         if( pxQueue->pcReadFrom >= pxQueue->pcTail )
702         {
703             pxQueue->pcReadFrom = pxQueue->pcHead;
704         }
705         --( pxQueue->uxMessagesWaiting );
706         memcpy( ( void * ) pvBuffer, ( void * ) pxQueue->pcReadFrom,
707             ( unsigned ) pxQueue->uxItemSize );
708
709         /* If the queue is locked we will not modify the event list. Instead we update the lock count
710         so the task that unlocks the queue will know that an ISR has removed data while the queue was
711         locked. */
712         if( pxQueue->xRxLock == queueUNLOCKED )
713         {
714             /* We only want to wake one task per ISR, so check that a task has not already been woken. */
715             if( !( *pxTaskWoken ) )
716             {
717                 if( !listLIST_IS_EMPTY( &(amp; pxQueue->xTasksWaitingToSend) ) )
718                 {
719                     if( xTaskRemoveFromEventList( &(amp; pxQueue->xTasksWaitingToSend) )
720                         != pdFALSE )
721                     {
722                         /* The task waiting has a higher priority than us so
723                         force a context switch. */
724                         *pxTaskWoken = pdTRUE;
725                     }
726                 }
727             }
728         }
729         else
730         {
731             /* Increment the lock count so the task that unlocks the queue
732             knows that data was removed while it was locked. */
733             ++( pxQueue->xRxLock );
734         }
735
736         xReturn = pdPASS;
737     }
738     else
739     {
740         xReturn = pdFAIL;
741     }
742
743     return xReturn;
744 }
745
746 unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle pxQueue )
747 {
748     unsigned portBASE_TYPE uxReturn;
749
750     taskENTER_CRITICAL();
751     uxReturn = pxQueue->uxMessagesWaiting;
752     taskEXIT_CRITICAL();
753
754     return uxReturn;
755 }
756
757 void vQueueDelete( xQueueHandle pxQueue )
758 {
759     vPortFree( pxQueue->pcHead );
760     vPortFree( pxQueue );
761 }
762

```

```

763 static signed portBASE_TYPE prvUnlockQueue( xQueueHandle pxQueue )
764 {
765     signed portBASE_TYPE xYieldRequired = pdFALSE;
766
767     /* THIS FUNCTION MUST BE CALLED WITH THE SCHEDULER SUSPENDED. */
768
769     /* The lock counts contains the number of extra data items placed or
770     removed from the queue while the queue was locked. When a queue is
771     locked items can be added or removed, but the event lists cannot be
772     updated. */
773     taskENTER_CRITICAL();
774     {
775         --( pxQueue->xTxLock );
776
777         /* See if data was added to the queue while it was locked. */
778         if( pxQueue->xTxLock > queueUNLOCKED )
779         {
780             pxQueue->xTxLock = queueUNLOCKED;
781
782             /* Data was posted while the queue was locked. Are any tasks
783             blocked waiting for data to become available? */
784             if( !listLIST_IS_EMPTY( &(amp; pxQueue->xTasksWaitingToReceive) ) )
785             {
786                 /* Tasks that are removed from the event list will get added to
787                 the pending ready list as the scheduler is still suspended. */
788                 if( xTaskRemoveFromEventList( &(amp; pxQueue->xTasksWaitingToReceive) ) != pdFALSE )
789                 {
790                     /* The task waiting has a higher priority so record that a
791                     context switch is required. */
792                     xYieldRequired = pdTRUE;
793                 }
794             }
795         }
796     }
797     taskEXIT_CRITICAL();
798
799     /* Do the same for the Rx lock. */
800     taskENTER_CRITICAL();
801     {
802         --( pxQueue->xRxLock );
803
804         if( pxQueue->xRxLock > queueUNLOCKED )
805         {
806             pxQueue->xRxLock = queueUNLOCKED;
807
808             if( !listLIST_IS_EMPTY( &(amp; pxQueue->xTasksWaitingToSend) ) )
809             {
810                 if( xTaskRemoveFromEventList( &(amp; pxQueue->xTasksWaitingToSend) ) != pdFALSE )
811                 {
812                     xYieldRequired = pdTRUE;
813                 }
814             }
815         }
816     }
817     taskEXIT_CRITICAL();
818
819     return xYieldRequired;
820 }
821
822 static signed portBASE_TYPE prvIsQueueEmpty( const xQueueHandle pxQueue )
823 {
824     signed portBASE_TYPE xReturn;
825
826     taskENTER_CRITICAL();
827     xReturn = ( pxQueue->uxMessagesWaiting == ( unsigned portBASE_TYPE ) 0 );
828     taskEXIT_CRITICAL();
829
830     return xReturn;
831 }
832
833 static signed portBASE_TYPE prvIsQueueFull( const xQueueHandle pxQueue )
834 {
835     signed portBASE_TYPE xReturn;
836
837     taskENTER_CRITICAL();
838     xReturn = ( pxQueue->uxMessagesWaiting == pxQueue->uxLength );
839     taskEXIT_CRITICAL();
840
841     return xReturn;
842 }
843

```