Master.

E4.52
ISE3.35

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2009

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

REAL-TIME OPERATING SYSTEMS

Friday, 15 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 2009 should be distributed with the paper.

Special instructions for students

You may use the booklet RTOS Exam Notes 2009 which is a reproduction of that published on the course web-site before the exam.

The Questions

1.

(a) Explain carefully the different characteristics of deadlock, starvation, and livelock in a real-time system. [8]

(b) Write, using pseudo-code, an example application which might suffer from real livelock. Suggest, with modified pseudo-code, how the livelock in your example can be detected and prevented. [8]

(c) An application uses busy-wait loops to ensure exclusive resource use. It is observed that all tasks continue to run, but no progress is made. Comment on possible reasons for this. [4]

2. 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 2009.

(a) Explain with an appropriate timeline what is the sequence of task states of task X sending two messages, and task Y receiving 1 message, communicating through a message queue of length 1 which is initially empty, in such a way that both tasks X and Y block temporarily. State on your diagram what causes each state transition. [5]

(b) Identify the source code lines in Exam Notes 2009 in the execution *QueueSend()* and *QueueReceive()* at each point in your timeline. [5]

(c) State under precisely what circumstances the FreeRTOS v4.0.5 code could result in a call to *QueueReceive()* returning with an error code when there has been no timeout. [5]

(d) Considering the case of one sender and multiple receivers on an empty queue discuss what semantics for *QueueReceive()* would be ideal in a pre-emptive priority-based system, where higher priority tasks should if possible have priority in message queues. Discuss why implementation of this would be difficult. [5]

3.

(a) A priority-scheduled real-time system consists of four jobs with the characteristics shown in *Figure 3.1*. Answer the following questions, using inequalities on *t1, t2* as necessary to determine the answer. You may assume RTOS task-switching overheads are negligible.

(i) How would you prioritise these jobs under an RTOS with prioritised tasks? (Your answer may depend on *t1* and *t2*).

(ii) Assuming the prioritisation in (i), state for which values of *t1, t2* the following cases apply:

A. The system will certainly not meet all deadlines

B. It is not known whether the system will meet all deadlines

C. The system is guaranteed to meet all deadlines.

[8]

(b) Suppose that the jobs W, Z share mutually exclusive access to a resource R. The resource is claimed for a 20 us critical section once in each job period by both W & Z. Determine what are the appropriate blocking times to use in extended RMA for this system, and hence establish an inequality on *t1,t2* which guarantees all deadlines are met in the new system.

[6]

(c) Suppose that in the system described by *Figure 3.1* and part (a) job Z has higher priority than W, all other relative priorities are as in your answer to part (a) (i). In the revised system calculate an inequality on *t1,t2* which guarantees all deadlines by finding an upper limit on the time that W is unable to execute because Z is executing and treating this as blocking time.

[6]

Job	Job Time	Job Period
X	50 us	*t1* us
Y	100 us	*t2* us
Z	50 us	500 us
W	20 us	200 us

Figure 3.1

4.

Figure 4.1 shows part of the code for tasks A,B,C & D in a FreeRTOS application. There are no calls to *printer_open*(), *printer_close*() and *print*() except as indicated in *Figure 4.1*. It is specified that:

(1) A,B,C have all executed *printer_open*() more recently than *printer_close*() whenever function *print*() is called.

(2) A,B.C have all executed *printer_close*() after *print*() returns before any one of them executes *printer_open*() or *print*() is called again.

(3) *printer_open*() and *printer_close*() are never active at the same time as *print*().

Write FreeRTOS code for segments S_{A1}, S_{B1}, S_{C1}, S_{A2}, S_{B2}, S_{C2} and S_{D1}, S_{D2}, and a function *initialise()* called once before any of the code in *Figure 4.1* is executed which ensures that conditions (1), (2) and (3) above always ahold. Explain clearly why your code ensures each of these three conditions is satified.

[20]

Task A	**Task B**	**Task C**	**Task D**
....	....	....	....
printer_open()	*printer_open*()	*printer_open*()	S_{D1}
S_{A1}	S_{B1}	S_{C1}	*print*()
printer_close()	*printer_close*()	*printer_close*()	S_{D2}
S_{A2}	S_{B2}	S_{C2}	.....
......	......	......	

Figure 4.1

5. A game is played with N rods labelled 1, 2, ... N. Rod x can contain any non-negative number of beads $B(x)$. A single move consists of examining the number of beads on two rods, x and y, and changing the number of beads on x, y simultaneously to new values according to a function f:

$$B(x) := f(\, B(x), B(y))$$

$$B(y) := f(\, B(y), B(x))$$

The two calls to function $f()$ are both made using the old values of $B(x)$, $B(y)$, and the function $f()$ has a return value which depends only on its parameters.

It is proposed to implement a timed simulation of this strategy with a number of tasks each implementing a sequence of moves using randomly chosen rods, and locations in shared memory implementing the values $B(x)$. Multiple moves can be made in parallel by different tasks as long as each move uses a disjoint set of rods.

(a) Using scheduler locking write pseudo-code for a function:

`Move(int x, int y)`

which implements a single move on rods x,y and ensures both that the memory values $B(i)$ are always consistent and that moves which interfere with each other will therefore never be made.

[4]

(b) Using semaphores write an implementation of **`Move(int x, int y)`**. State in what ways this is better, and in what ways worse, than your answer to (a).

[10]

(c) Prove that your answer to part (b) cannot deadlock.

[6]

6.

(a) Explain the difference between scheduler locking and interrupt locking in FreeRTOS.

[4]

(b) State three ways in which a critical section can be implemented in an RTOS. In each case determine under what circumstances the implementation will increase overall worst-case task or interrupt latency.

[6]

(c) Explain precisely how a clock tick is implemented in FreeRTOS 4.0.5, illustrating the sequence of operations with an appropriate diagram, if a clock tick interrupt occurs:

(i) while interrupts are locked.

(ii) while the scheduler is locked.

Detail changes to any data structures affected by the transaction.

[10]

[END]

RTOS EXAM NOTES 2009

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

```
/*
 * 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;
}
```

```
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;
}
```

E4.52
ISE3.35

Solutions 2009

FOUR Questions in 180 minutes => 45 min per question

Answer codes: A=analysis, B=bookwork, D=design, C= new application of learnt theory

Question 1.

This question tests whether the students understand some of the issues when writing application code for RTOS

a)
Deadlock:
- cyclic resource dependency graph.
- Once started cannot end regardless of schedule.
- Must involve 2 or more tasks indefinately suspended

Starvation:
- 100% utilisation of some resource (e.g. CPU)
- can be stopped at any time by changing schedule
- 1 or more tasks indefinitely suspended

Livelock:
- All tasks execute but no progress is made by one or more tasks
- dependent on relative timing between tasks

[8B]

b)
```
TaskA()
{
 done = 0
 do something
 while (not done) {
   lockA:= 1;
   delay(1)
   if (lockB == 0)   {
      do something
      done = 1;
      lockA = 0;
   }
 }
}
```
TaskB identical but swap A,B

[8C]

c)

Pseudo-livelock (AKA pseudo-deadlock) where resources are used with a cyclic dependency graph but busy-wait means that no process suspends. The situation is effectively deadlock.

[4A]

Question 2.

a)

1. Y: *QueueReceive*() (API call in Y causes Y to block)
2. X: *QueueSend*() (API call in X causes Y to go from blocked to ready)
3. X: *QueueSend*() (API call in X causes X to block)
4. Y: starts executing (caused by call to scheduler when no higher priority task is ready)
5. X: Y executing causes the message to be picked up and X to unblock and post its second message

[5A]

b)

1. 619-642
2. 473-482, 535-544,550-559, 561?, 571
3. 472-519
4. 643-669,675-678,680 or 680-682 or 687, 690
5. 520-544, 550-554,571

[5B/A]

c) *QueueReceive*() returns an error code if called with no message available and then a task posts a message which unblocks the task, however before it executes another tasks calls *QueueReceive*() and picks up the posted message.

[5B/A]

d) Ideally, at the time that no higher priority task is executing, the highest priority waiting task on a queue should receive a posted message. The problem is that when a message is posted a higher priority task may be executing thereby delaying the time at which this decision should be made. But if it is not known which task to make ready until the scheduler is called at some subsequent time this greatly complicates the RTOS – this condition would need to be checked for every time a task suspended.

[5B]

Question 3

This question tests student's understanding of scheduling algorithms and analysis of real-time performance. The last two parts require innovative thinking.

a)

(i) Use RMA priority order, longer periods have lower priorities

(ii)

$U = 20/200+50/500+100/t2+50/t1= 0.2+100/t2+50/t1 < 4(2^{1/4}-1) = 0.757 \Rightarrow 100/t2+50/t1 < 0.557$

For U meeting this inequality C applies.

For U > 100% A applies

Otherwise B applies.

[8A]

b)

W is higher priority than Z therefore Z can block W but W cannot block Z. Blocking time is therefore 20us max per W period. Using extended RMA we add 20/200=0.1 onto the utilisation to get new inequality:

$100/t2+50/t1 < 0.457$

Note that X,Y have no extra blocking time. The affect of W being blocked on all other tasks is accounted for in the increased utilisation from W.

[6A]

c)

The maximum time W cannot execute because Z is executing (per W period) is the job time of Z (50us). So the utilisation becomes:

$(50+20)/200+50/500+100/t2+50/t1= 0.45+100/t2+50/t1 \Rightarrow 100/t2+50/t1 < 0.307$

[6A]

Question 4.

This question requires synchronisation similar to barrier – but with the two sides of the barrier separated in time. Many solutions are possible. One solution is:

```
xSemaphoreHandle  SemA1,SemB1,SemC1, SemX2;

initialise()
{
    vSemaphoreBinaryCreate(SemA1) ;
    xSemaphoreTake(SemA1); /* start with no token */
 /* repeat for all other semaphores */
}
SD1()
{
   /* this code will wait till A,B,C have executed printer_open()
   xSemaphoreTake( SemaA1);
   xSemaphoreTake(SemaB1);
   xSemaphoreTake(SemaC1);
}
SD2()
{
   int i;
   for (i=0; i < 3; i++) SemaGive(SemaX2);
         /* this code will release A,B,C to execute printer_close()
   xSemaphoreTake( SemaA1);
   xSemaphoreTake(SemaB1);
   xSemaphoreTake(SemaC1);
   for (i=0; i < 3; i++) SemaGive(SemaX2);
      /* this code will release A,B,C to execute printer_close()
}
SX1()
{
   SemaGive(SemaX1); /* must precede print() */
   SemaTake(SemaX2); /* succeeds after print() */
}
SX2()
{
   SemaGive(SemaX1);
   SemaTake(SemaX2);
}
```

(3) print() can only be executed when D has all three tokens and therefore A,B,C are inside SX1 or SX2

(1,2) From initialisation all printer_open(0 calls must happen before D finishes SD1, whereas all printer_close() calls must happen after print() but before any task finished SX2.

[20D]

Question 5.

a)

```
move(int x , int y) {
lock scheduler
a := B(x);
b := B(y);
B(x) := f(a,b);
B(y) := f(b,a);
unlock scheduler
}
```

[6D]

b) Use an array of binary semaphores Sem[i] , one semaphore for each rod, each initialised to a token count of 1.

```
move( int x, int y) {
if (x > y) {
        SemaTake(Sem[x]);
        SemaTake(Sem[y]);
} else {
        SemaTake(Sem[y]);
        SemaTake(Sem[x]);
}
a := B(x);
b := B(y);
B(x) := f(a,b);
B(y) := f(b,a);
SemaGive(Sem[x]);
SemaGive(Sem[y]);
```

Semaphore better - independent MOVE operations do not slow each other

Worise - slower + more RAM used.

[10D]

c)

The order in which semaphore Sem[x] is claimed is always larger x first hence cyclic dependence graph can never be formed and deadlock not happen.

[4A]

Question 6.

(a)

Interrupt locking temporarily stops interrupts from happening, Any outstanding interrupts are executed at end of section.

Scheduler locking prevents pre-emption from happening but allows interrupts. The highest priority ready task will be scheduled at the end of the section.

[4B]

(b)

Interrupt lock . If CS length is greater than current max CS length it will increase interrupt latency and (therefore) also task latency.

Scheduler lock. If lock length is greater than current max scheduler lock length it will increase task latency, but have no effect on interrupt latency.

Semaphore protection. No effect on task or interrupt latency.

[6A/B]

(c)
(i) If interrupts are locked the tick ISR is delayed until end of lock, and then happens as normal.
(ii) If scheduler is locked the tick ISR happens but does nothing except:
Call user hook function (if defined)
add one to uxTicksMissed (this global is 0 when scheduler is unlocked)

Normally the tick ISR executed the tick function, see below.

When the scheduler is unlocked, if uxTicksMissed is >0 the tick function is executed a number of times equal to the value of uxTicksMissed, and uxTicksMissed set back to 0.

The tick function itself executes:

- increment xTickCount
- if xTickCount is 0 (therefore has overflowed) swap DelatedTaskList and OverflowDelayedTaskList
- Scan through DelayedTaskList and remove tasks that have timeout == xTickCount, adding them to ready list.
- Move tasks from PendingTaskList to DelayedTaskList

[10A]