E4.52
ISE3.35

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2007

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

Corrected Copy

REAL-TIME OPERATING SYSTEMS

Tuesday, 8 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 should be distributed with the paper.

Special instructions for students

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

The Questions

1.

a) Write pseudo-code using the FreeRTOS API to illustrate mutually exclusive access to a shared resource using (i) critical sections and (ii) semaphores. When writing an application, how would you choose between these two methods?

[4]

b) In the FreeRTOS API critical sections can be enforced in two distinct ways. What are they, and what are their relative advantages and disadvantages?

[4]

c) Explain why priority inversion is a problem in real-time systems, illustrating your answer with an appropriate execution trace. How could an application under FreeRTOS overcome this problem?

[4]

d) "My RTOS application appears to have no liveness problems, therefore rate monotonic analysis is unnecessary". Discuss.

[4]

e) Under FreeRTOS, state giving reasons what is the expected sleep time from a single call of **`TaskDelay(n)`**? Under precisely what circumstances, and why, will use of **`TaskDelayUntil()`** lead to more accurate timing than **`TaskDelay()`**?

[4]

2.

a) Contrast the merits of writing real-time application code under a set of prioritised interrupts, or a set of prioritised tasks.

[4]

b) Using the FreeRTOS API, describe, with pseudo-code, two ways in which application code in a task could be synchronised with an interrupt, stating which you would prefer to use and why.

[4]

c) A priority-scheduled real-time system consists of four jobs with the characteristics shown in Figure 2.1 when run under CPU A.

(i)` How would you run 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 under CPU A? Give reasons for your answer. You may assume RTOS task-switching overheads are negligible.

(iii) You are asked to choose a speed-rating for a CPU to run these jobs. What is the minimum speed, relative to A, that would guarantee all deadlines met?

[6]

d) Inter-task communication is introduced to the system of Figure 2.1 which results in the blocking specified in Figure 2.2 every job period. Answer the three parts of d) for the new system. How would your answers change if earliest deadline first scheduling were used?

[6]

Job	Job Time	Job Period
X	50us	220us
Y	1us	7us
Z	100us	300us
W	125us	250us

Figure 2.1

Job	Blocking Time
X	0
Y	0
Z	60us
W	5us

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 booklet RTOS Exam Notes.

a) Discuss in detail how FreeRTOS implements copying of message data and the implications of this for the implementation, and the application programmer.

[4]

b) Explain the operation of **`xQueueReceive()`** when a task suspends waiting for a message from a queue, and then returns with a message posted from an ISR while the queue is locked . You may use the line numbers in the Exam Notes booklet to identify source code in your answer.

[4]

c) Describe the operation of **`QueueSend()`** and **`QueueReceive()`** during the sequence of events in Figure 3.1. What is problematic about FreeRTOS v4.0.5 behaviour under this sequence, and how could application code cope with this behaviour?

[6]

d) Describe one way in which the FreeRTOS v4.0.5 queue implementation might be improved to provide better behaviour under the case described in c).

[6]

Time	Event
Initially	Task priorities: D > C > B > A Two tasks A,B are waiting on messages from empty queue Q1, task C is running.
1	Task C calls **`QueueSend(Q1)`**
2	Task D preempts C
3	Task D calls **`QueueReceive(Q1)`**
4	Task D sleeps
5	Task B runs

Figure 3.1

4. This question relates to the FreeRTOS task list package implementation, source code for which can be found in the booklet RTOS Exam Notes.

a) Describe, with the aid of a suitable diagram, the data structures used in the FreeRTOS task list package. [2]

b) What are the operations needed to implement a RTOS ready list, and how are these implemented by FreeRTOS using its task list package? [4]

c) For each pointer field in the task list package, discuss what would be the consequences, good or bad, for the FreeRTOS ready task list implementation if the pointer field were omitted? Do not consider any other uses of task lists. [6]

d) In FreeRTOS, detail for what purposes task lists are used, and discuss the merits of using a general purpose task list package. [4]

e) MicroC/OS-II implements task lists using a bit array, packed 8 bits per byte, together with a 256 byte constant array to perform efficient selection of the most significant bit set within a byte. Describe briefly how a set of tasks is represented in this implementation. Discuss the advantages and disadvantages of this when compared with FreeRTOS. [4]

5.

a) Describe and contrast the merits of priority inheritance protocol (PIP) and ceiling priority protocol (CPP) as solutions to priority inversion.

[4]

b) What conditions on the resource dependency graph are necessary and sufficient for a system to be deadlocked? How, writing code at the application level, can this situation be avoided?

[4]

c) Figure 5.1 describes three scenarios S1, S2, S3 in a real-time system. In each case state, giving reasons, what you can deduce about whether deadlock, starvation, or livelock might be responsible for the lack of progress.

[6]

d) Show how priority ceiling protocol (PCP) as defined in the Exam Notes eliminates priority inversion.

[4]

e) Does PCP implement deadlock prevention, avoidance, or recovery?

[2]

Task	**Priority**	**S1**	**S2**	**S3**
A	4	P	P	P
B	3	P	S	B
C	2	B	S	P
D	1	P	B	B

P= making progress, S = running, making no progress, B=permanently blocked

Figure 5.1

6. Answer **ONE only** of the following questions. Credit will be given for answers which are concise, clear, and complete.

(a) Real-time Operating Systems normally have a single frequency "tick" interrupt. What would be the consequences for RTOS implementation and usage if this were replaced by a variable-time interrupt?

(b) RTOS porting depends on both compiler and CPU architecture. Examine how each of these can influence the code necessary for an RTOS port giving (possibly hypothetical) examples, and a checklist of the issues that affect RTOS implementation, together with how easy it is for an RTOS implementation to incorporate their variability into configuration switches that do not require new code to be written.

(c) Discuss how the FreeRTOS API implements semaphores, and what are the merits of more complex implementations that incorporate solutions to priority inversion.

(d) Discuss ways in which an RTOS implementation could achieve faster task-level latency, and to what extent this is dependent on specific hardware or compiler features.

(e) Event registers are often provided in an RTOS API. List a set of features that could be implemented in an event register API. For each feature, summarise, with reasons, the costs of implementation, and state with examples how it might enable better application programming.

[20]

[END]

RTOS EXAM NOTES 2007

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

```
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

```
#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

```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;
}
```

Isc 3.35
E4. 52

Solutions 2007

FOUR Questions in 180 minutes => 45 min per question

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

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

a)

```
taskENTER_CRITICAL()
/*access resource*/
taskEXIT_CRITICAL()
```

```
xQueueHandle sema;
vSemaphoreCreateBinary( sema);
xSemaphoreTake(sema);
/*access resource*/
xSemaphoregive(sema);
```

[3B]

b)

(1) interrupt disable [ENTER/EXIT]_CRITICAL_SECTION() as above

Benefit: fast, no priority inversion

disadvantage: may increase global interrupt latency

(2) scheduler locking:

```
vTaskSuspendAll()
/*critical section*/
vTaskResumeAll()
```

Benefit: allows interrupts while locked, no priority inversion

Disadvantage: can't be used to share resource with interrupt

[4B]

c) PI can increase the delay of a high priority task when it is sharing a resource with a low priority task. Specifically:

LPT runs and takes S

HPT runs & waits on S

LPT continues

MPT preempts LPT

MPT blocks

LPT runs and gives S

HPT takes S

In this trace any number of tasks with priority greater than LPT and less than HPT can delay progress of HPT

Mend the problem by implementing priority inheritance protocol (PIP) using FreeRTOS dynamic task priorities:

vTaskPrioritySet()

uxTaskPriorityGet()

increase the priority of LPT to that of HPT for the length of time that LPT holds S and HPT is waiting on S

[4B]

d) Strictly speaking, CPU starvation (which may happen if RMA analysis does not guarantee deadlines) is a form of starvation and therefore liveness problem. However just because it does not happen, it does not mean that it is guaranteed not to happen given future minor changes in conditions or code. therefore RMA is still useful because it guarantees no CPU starvation (though does not say anything about other liveness problems).

[4A]

e) Single call results in wait for n clock ticks => time of min(0, (n-1))*tick-time to n*tick-time

TaskDelayUntil is useful for repeated delays where the time from one call to the next is critical. This will be precise providing the total task (+ higher priority task) execution time is less than the specified delay. This is better than TaskDelay() – which will go wrong if the above total time is greater than one time-tick.

[4A]

2.

a)

Interrupts: no RTOS required => more compact code, faster switching.

Disadvantages: Prioritised interrupts are required. Code must be written as separate ISRs with no state preserved in ISR between calls. RTOS communication & synchronisation primitives can't be used.

RTOS: reverse of above.

[4B]

b) EITHER use a binary semaphore, posting from the ISR and taking the semaphore in the task to implement wait for next interrupt.:

SemaphoreGiveFromISR(sema)--> SemaphoreTake(sema)

OR use task suspend resume:

vTaskSuspend(0) – will suspend current task

xtaskResumeFromISR(taskH) – will restart task from ISR

Note task handle must be stored after task creation.

The suspend/resume solution does not require a semaphore (queue) and is therefore both more compact and faster.

[4B/A]

c)

(i)` Y,X,W,Z (decreasing priority order)

(ii) Utilisation = 50/220+1/7+30/300+40/250=0.63

RMA limit n(2^(1/n)-1) for 4 tasks = 0.757

RMA theorem => all tasks meet deadlines with certainty

(iii) 0.63/0.757=0.83 that have A is minimum speed

[6C]

d) Add blocking time to CPU time so: (i) no change, (ii)+0.22=> RMA conditions not met. CANT TELL whether deadlines will be met. (iii) 0.85/0.757= 1.12X speed of A. With EDF scheduling, assuming CPU+blocking is less than 100%, can guarantee all deadlines. speed 0.85/1 would be minimum.

[6C]

3. *This question tests whether students understand the implementation of RTOSes by examining in detail the behaviour of some real (but not ideal, and hence now replaced) code.*

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

a) Discuss in detail how FreeRTOS implements copying of message data and the implications of this for the implementation, and the application programmer.

[4B]

b) line 629 IF is true since Q is empty. Line 634 If is true since timeout must be non-zero for task to wait. Line 640 – task suspends.

ISR posts message and rewakes task, which continues from line 640.

643: scheduler is locked

Line 652 test is true, since message is waiting to be picked up

Message is extracted from queue (with possible read pointer wrap-around (line 657)

Line 665: Lock count is incremented in case Q was previously full

666: set pdPass return value

668-671 skipped

678 – scheduler unlocked, causes no change of task

680 – taskYield() does not result in change of task.

return pdPass (set from 666)

[4A]

c) See below for details. Problem: task B returns early with a failure when it has not timed out. Application code could check whether timeout was real and if not try again to receive a message by calling QueueReceive() again with an adjusted timeout. This must be implemented with a loop, since spurious wakeup can happen at any time.

[6A]

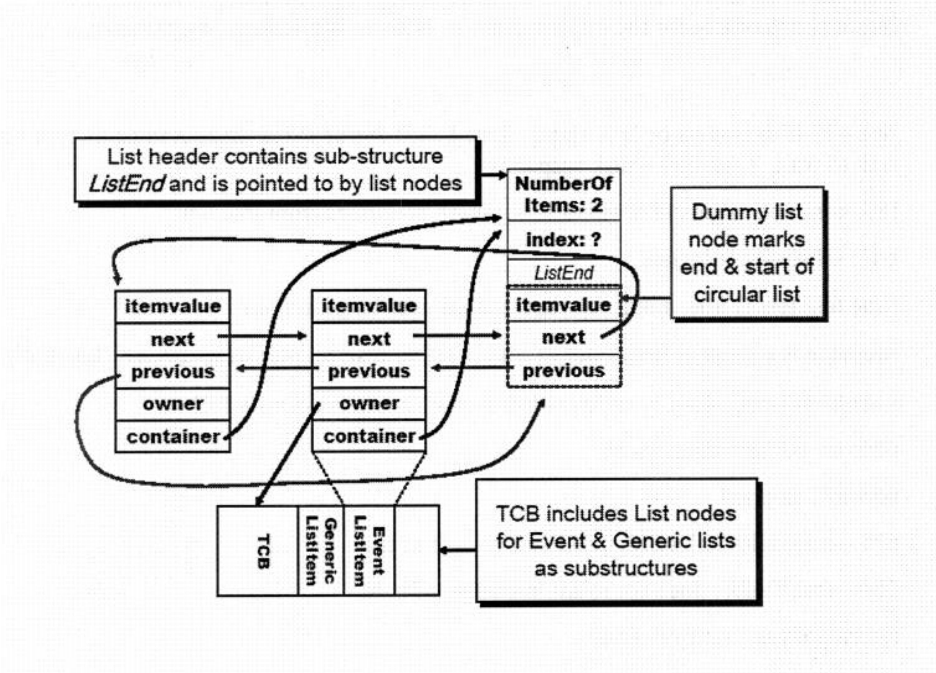
Time	Event	Details
1	task C calls QueueSend(Q1)	In QueueSend(), from prvUnlockQueue, B, the highest priority waiter, is awoken but does not yet run
2	task D preempts C	
3	task D calls QueueReceive(Q1)	D finds the message that was posted in Q1, and returns immediately with this.
4	task D sleeps	
5	task B runs	B runs inside QueueReceive and finds that no message is available, it returns with a failure (as from timeout)

d) Solution: Distinguish between a timeout wakeup and a (perhaps spurious) event wakeup by checking timeout value (which is a local variable) and go back to sleep if the latter has not elapsed. Can distinguish because the wakeup time can be calculated on QueueReceive() entry and stored in local variable for reference.

[6D]

4. This question relates to the FreeRTOS task list package implementation, source code for which can be found in the booklet RTOS Exam Notes.

a)



[2]

[2B]

b)

add task to ready list

delete task from ready list

find & remove highest priority task from ready list

move to next task (round-robin) of set of tasks of identical priority in ready list.

ready list is implemented as array of (circular) task lists, one for each priority. add & delete task from task list functions are used as necessary. All tasks within one list are same priority so ordering is not relevant, tasks are added to end of list (different from start in the case that there are multiple tasks of same priority). List of traversed continuously in case time-slicing is needed.

[4B]

c) next & previous. Could delete one of these to make singly-linked. Delete task would take longer, other operations (except move to next task) would be quicker.

container – not needed for ready list, since know what it is

owner – not needed for ready list, since know offset between listitem node and task TCB so can reconstruct this.

[6A]

d) Used for ready list, suspended task list, delayed task list, overflow delayed task list, event lists (two per message queue). General package considerably reduces RTOS code size, however not all features are used in all cases, so separate code would be slightly more efficient.

[4B]

e) Each separate priority is represented by one bit. Fixed number of priorities allowed (e.g. 64) => 8 bytes needed for table. Must have unique task per priority. Tasks are represented as in list by setting appropriate bit. This means that dynamic priority would require changes to task lists (all of them containing the task).

[4A]

5.

a) Both stop priority inversion from time that a HPT is waiting by ensuring task with lock on resource is same priority as HPT. CPP requires static analysis of code (which may be additional burden for programmer). It has advantage that low priority tasks will have high priority whenever they lock the resource. this makes no difference to worst case HPT delay. However it reduces average case HPT delay by making it less likely that another lower priority task will have resource locked when HPT requests lock.

[4B]

b) A cycle in the graph <=> deadlock

This can be avoided by ordering resources and making sure all tasks claim shared resources in the same (global) order. (resources can be released in any order).

[4B]

c) S1. only one task is blocked, so can't be deadlock. Task is blocked, so can't be livelock. Must be starvation. Two higehr priority tasks must be hogging some resource.

S2. B,C are running & making no progress – they must be livelocked.

S3. B & D are blocked. B cannot be starved since only one higher prio task, hence it must be deadlocked. D cannot be blocked because B must be deadlocked with D. Hence B,D are deadlocked.

[6A]

d) Condition 2 of PCP means that any task locking a resource will inherit dynamically the priority of any task waiting on the lock. Condition 3 means that this will allow preemption if necessary of a lower priority task.

[4A]

e) Prevention, since it constrains resource lockers to wait until such time as deadlock is no longer possible.

[2A]

6.

This question tests whether the student have deeper understanding of some aspect of the topic. All students have RTOS implementation coursework relating to one of these five options. The given question tests whether they understand the wider issues relating to their implementation. The answers will depend on the precise coursework topic.

[20]