Final

E4.52
ISE3.35

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2010

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

REAL-TIME OPERATING SYSTEMS

Thursday, 13 May 10:00 am

Time allowed: 1:30 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 Information for Invigilators: none.

Information for Candidates

Pre-published course notes can be found in the booklet RTOS Exam Notes.

The Questions

1.

a) State the assumptions behind the job model of computation, and the Rate Monotonic Analysis (RMA) utilisation limit.

[4]

b) Table 1.1 shows the CPU time (in cycles) and period (in microseconds) of four jobs. Using rate monotonic analysis or otherwise determine the optimum priorities and calculate a limit on the CPU cycle time which will ensure all deadlines are met if jobs have optimal priorities

[5]

c) Assume all four jobs from part b) use a single mutually exclusive shared resource once per job time. If the time for which each job locks the resource is R CPU cycles, and priority inversion is prevented, by usages of priority inheritance protocol, determine as a function of R the blocking time of each job for use in extended RMA, and hence calculate a new limit on CPU cycle time for all deadlines to be met.

[5]

d) The jobs from part b) are implemented so that the two lowest priority jobs have equal priority, but priorities are otherwise as in part b). Assume R is zero (no time waiting on shared resource). If no assumptions are made about which job is executed first when both jobs have equal priority, compare the execution of this system with the same system under optimal priorities. Hence, or otherwise, determine the correct blocking time for each job in order to apply extended RMA to the three priority system, and therefore calculate an inequality on the CPU cycle time for all deadlines to be met.

[6]

Job	CPU time / cycles	Period / μs
A	100	10
B	200	40
C	300	150
D	400	100

Table 1.1

2.

a) Describe the main performance characteristics of the FreeRTOS double-linked list package, and list the places in which it is used in FreeRTOS.

[8]

b) Explain the use of sentinel nodes in the FreeRTOS doubly-linked list package.

[4]

c) For each usage in part a), compare and contrast the use of doubly-linked lists with a similar singly-linked list package, using circular lists and a sentinel node.

[8]

3.

a) Explain how queues can be used to implement semaphores.

[4]

b) The pseudocode in Figure 3.1 describes a real-time system with tasks A, B, C, D and corresponding pseudocode **TaskA**, **TaskB**, **TaskC**, **TaskD**. The two queues **q[1]** and **q[2]** are initialised by the code **Init()**. The pseudocode comments *execute* and *wait* represent periods when the task is respectively executing, or suspended, for fixed but unspecified lengths of time.

Identify all potential liveness problems, stating the conditions required for each problem.

[16]

```
Init()
{
  for (i = 1; i  < 3; i++) {
    q[i] = QueueCreate();
    QueueSend(q[i]);
  }
}

TaskA()
{
  while (TRUE) {
    QueueReceive(q[2]);
    Queuereceive(q[1]);
    /* execute */
    QueueSend(q[2]);
    QueueSend(q[1]);
    /* wait */
  }
}
```

```
TaskB()
{
  while (TRUE) {
    QueueReceive(q[1]);
    QueueReceive(q[2]);
    /* execute */
    QueueSend(q[2]);
    QueueSend(q[1]);
    /* wait */
  }
}
```

```
TaskC()
{
  while(TRUE) {
    QueueReceive(q[1]);
    /* execute */
    QueueSend(q[1]);
    /* wait */
  }
}

TaskD()
{
  while(TRUE) {
    /* execute */
    /* wait */
  }
}
```

Figure 3.1

4.

a) Describe the three types of locking used in the FreeRTOS queue package.

[5]

b) Determine, with reference to the line numbers shown in the Exam Notes, at which points in the execution of QueueReceive() locking is changed. In each case, where a lock is started or stopped, state why this is either necessary or desirable.

[15]

5. Table 5.1 Lists the I/O devices serviced by a microcontroller, together with the required maximum latency time, frequency at which the device needs service, and length of service routine. Assume that interrupt entry time is E µs and return time is R µs, and that two interrupt levels are available *P1* and *P2* with *P1* higher priority than *P2* so that each device may raise a *P1* or *P2* interrupt, or no interrupt. Each interrupt has a separate vector, however when more than one device can raise an interrupt both use the same vector.

a) Explain how a foreground/background system can be used to implement the system, and why the latency time of any device can be delayed by at most one service of any other device.

[5]

b) Suppose two devices A & B are serviced from the same interrupt. Determine from the polling code in Table 5.2 what is the maximum latency for each device as a function of E, R, and device service times, assuming no other interrupts or critical sections.

[5]

c) Assume $E = R = 0$, and that polling overhead is negligible. List all possible strategies for servicing the devices which meet all latency times.

[5]

d) For each strategy in part c), determine the inequalities on E, R which must be satisfied for the strategy to meet each latency time, again assuming that polling time is negligible.

[5]

Device	Service Time S / µs	Service Frequency / Hz	Maximum Latency Time L /µs
keyboard	10	100	90
printer	10	2000	15
network	1	10000	3
disk	1	4000	1.5

Table 5.1

```
if (device A ready) service device A;
if (device B ready) service device B;
```

Table 5.2

6. A digital oscilloscope (DSO) front-end receives real-time data samples from four independent channels at possibly different data rates, and processes these samples independently with a fast fourier transform (FFT) to generate phase and magnitude spectra. The spectra are further processed to generate time-averaged spectra, which are sent in real time to four separate windows on an LCD screen.

The DSO can simultaneously display and output averaged spectra via a serial link which sends 8 bit bytes at a rate of 1000 bytes per second.

Operation modes are controlled by a keyboard which must be polled once every 10ms in order to detect key presses. Keys may affect all other operations.

A single hardware FFT unit is available which calculates one FFT in the time shown in Table 6.1 and is used to implement all the FFT operations. The CPU time used is the sum of that needed to transfer input and output to or from the unit, and is in addition to the FFT calculation time during which the CPU is not required.

Each channel may use a 256, 1024, or 4096 sample FFT as selected by the user, however at higher sample rates and large FFT sizes there will be a gap between succesive samples due to the time taken by the FFT.

a) Using the outside-in design strategy or otherwise, determine an appropriate set of tasks to implement this system, stating the function of each task.

[6]

b) Considering CPU and FFT unit utilisation, and stating any assumptions you make, calculate the fastest sample rate which can be processed with 4096 sample FFTs on all channels without missing any samples.

[8]

c) Discuss how the FFT unit should interact with a real-time operating system, how it is protected from incorrect use, and whether the system is likely to suffer liveness problems.

[6]

Number of samples	FFT time	CPU time
256	10 μs	5 μs
1024	60 μs	20 μs
4096	260 μs	80 μs

Table 6.1

RTOS EXAM NOTES 2010

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

```
#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 )++;
}
```

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

Queue.h

```
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;
```

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

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

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

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

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

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

Master
E 4.52
I & C 3.35

Answer to Question 1

1.

a)

The job model assumes that each task starts executing at fixed periodic times (known period), executes for fixed CPU time, then suspends until next period.

The RMA utilisation limit applies when:

Tasks are priority scheduled with priorities such that smaller periods have higher priority

Each task has deadline equal to its period

There is no blocking

[4]

b) Priorities: A>B>D>C (2 marks)

Utilisation: 10+5+2+4 = 21 cycles/us. Limit is $4(2^{1/4}-1) = 0.757$

so $21 / f < 0.757$ (f in cycles/us = MHz) => $f > 27.7$MHz, $T < 35.1$ns (3 marks)

[5]

c) Blocking only matters from lower priority task, so:

A: R

B: R

D: R

C: 0

New utilisation adds R/10+R/40+R/100 = 0.135R so

$f > (21 + 0.135R)/0.757$, $T < 0.757/(21+0.135R)$

[5]

d) In the new system D,C are equal priority. This will help C, but slow D. Hence the maximum blocking time (for D) is the execution time of C, and no other task is blocked. Extended RMA gives us $f > (21+ 3)/0.757 = 31.7$ MHz, $T <$ 31.5ns. [-3 marks if assume incorrectly C & D each block the other, $T <$ 28.4ns]

[6]

Answer to Question 2

a) Four pointers per list node

Insertion at head or tail/removal all O(1)

Insertion into sorted list O(n)

No special cases for start/end of list

used: ready list, event lists, delayed task lists

[8]

b) Sentinel node is used so that zero length lists are not a special cases

[4]

c)

Ready list: This is arranged as array of lists indexed by priority so list is unnecessary except when equal priority tasks exist. However in singly-linked list case deletion from list would be difficult O(n). Insertion slightly faster since two less pointers to update. Other operations similar.

Delayed task lists: sorted list insertion is slightly more slower since must either keep two pointers or use pointer with extra level of indirection. Removal more difficult O(n) not O(1).

Event lists: as for delayed task lists, same operations needed.

[8]

Answer to Question 3

3.

a) A queue with *n* messages is equivalent to a semaphore of token count *n*. QueueSend, QueueReceive take the place of SemaGive, SemaTake. Initialisation by posting a number of initial messages can therefore implement a binary or counting semaphore.

[4]

b) There are three potential liveness problems: deadlock between tasks A & B which claim q[1] & q[2] in different orders, and priority inversion on q[1] where any of tasks A,B,C could be delayed by task D, and starvation on q1 for A,B,C.

Conditions for deadlock are specific execution order - it can happen regardless of task priorities. Conditions for priority inversion are that some task (e.g. D) is intermediate in priority between a given pair of tasks of A,B,C, and that this task preempts the lower priority of the pair while it has claimed p[1]. Conditions for starvation are utilisation of q1 by highest priority two of A,B,C is > 100%

[16]

Answer to Question 4

4.

a)

Interrupt locking (critical sections) disables interrupts, and also therefore other tasks, from preemption.

Scheduler locking, allows interrupts, but other tasks cannot pre-empt

Queue locking, allows interrupts to occur, and add or remove data from queues, however state of tasks waiting on queues cannot be changed until queue is unlocked.

[5]

b) See exam notes.

line	interrupt	scheduler	queue	notes
623-626		X	X	allow queue event lists to be changed
635	X	X	X	lock interrupts to allow queue to be unlocked
637-638	X			unlock queue, then scheduler. Can't be preempted because interrupts are locked
640	(X)			Yield.. If task changes interrupts will be enabled for duration of change
643-644	X	X	X	lock queues & scheduler first to allow interrupt unlock
646		X	X	unlock interrupts to minimise latency
650	X	X	X	lock interrupts again, to post message
673		X	X	unlock interrupts as soon as possible
676-678				unlock queues & scheduler since all finished

[15]

Answer to Question 5

5. Table 5.1 Lists the I/O devices serviced by a microcontroller, together with the required response time, frequency at which the device needs service, and length of service routine. Assume that interrupt entry and return time is *E* us & *R* us respectively, and that two interrupt levels are available *P1* and *P2* with *P1* higher priority than *P2* so that each device may raise a *P1* or *P2* interrupt, or no interrupt. Each interrupt has a separate vector, however when more than one device can raise an interrupt both use the same vector.

a) Every device can be serviced in foreground by one of the two interrupts or in a background polling loop.

The shortest service frequency (100us) is longer than the longest service time (90us). thus if deadlines are met at most one service request from each device during response time.

[5]

b) The time before start of service A, B is B_S+R+E, R+E+A_S

[5]

c) Sum of service times of other devices at own or higher priority must be less than latency. Disk & network must be higher priority than anything else, but can be in either order. Printer and keyboard can be any order if lower than disk & network. So (assuming we use all interrupts & background):

	Background Loop	P2	P1
1	K	P	D & N
2	K&P	D	N
3	K&P	N	D
4	P	K	D&N

[5]

d)

Strategy 1: $D_L > E+R+N_S$,

(do not matter) $N_L > E+R+D_S$, $P_L > 3E+2R+D_S+N_S$, $K_L>4E+3R+P_S+D_S+N_S$

Strategy 2: $N_L > E$, $D_L>2E+R+N_S$,

(do not matter) $P_L>3E+2R+D_S+N_S+K_S$, (also swap P,K)

Strategy 3: as 2 but swap N,D

Strategy 4: as 1 but swap K,P

[5]

Table 5.2

Answer to Question 6

6.

a)

For each channel: sample_input, FFT, average (total 12 tasks). Average & FFT tasks could be combined, in which case there are 8 tasks.

Keyboard processing, LCD control, serial_out (total 3 tasks).

[6]

b) Let time to process 4096 samples on one channel be T.

$T > 260+80us \Rightarrow T > 340$

CPU utilisation for each task is 80/T. Assuming that FFT processing is the only significant CPU task, and that each channel is independent therefore RMA must be used with 11 tasks as per part a, and that tasks do not block, we have:

$(80/T)*4 < 0.715$ (RMA for 11 tasks) $\Rightarrow T > 448us$

The FFT unit itself has utilisation $4*340/T < 1$. This is clearly the limiting factor. Hence max sample rate is for $T > 1360us \Rightarrow 3.01$ samples/us.

Hence max sample rate is 9.14 samples/us. (Answers which estimate blocking are also allowed).

[8]

c)

The FFT is a single resource with mutually exclusive hardware access. It should be protected by a binary semaphore. In this system FFT unit utilisation is much more significant than CPU utilisation, so although priority inversion is possible it will ot affect performance.

[6]

