4th. April, 2019	Course Code: CSE345/347/MCT-Elective	Time : 1:00 Hr
Embedded System Design; Mid Term		
The Exam Consists of 2 Questions in 4 Pages	**Total Marks: 25% Marks**	**1/4**

Student Name:	**Course Code:**

Q1:

1- CANCELLED

2- Write the suitable initialization function ADC_init(void) – using Tivaware APIs, to configure the ADC as follows: void ADC_init(void){

a. //Enable the clock for the Port to which the sensors are connected. _______

SYSCTL_RCGCGPIO_R |= 0X10;

____

b. //Enable the clock for the ADC module.

SYSCTL_RCGCADC_R |= 0X1;

____

c. //Disable the sequencer used.

ADC0_ACTSS_R &= ~0x8;

____

d. //Configure the sequencer for Processor trigger.

ADC0_EMUX_R &=~0XF000;

____

e. //Configure the steps of the sequencer.

ADC0 _SSCTL 3 _R |= 0 x 6

____

ADC0 _SSCTL 3 _R |= 1 3);

____

ADC0 _SSCTL 3 _R &= 1 3);

____

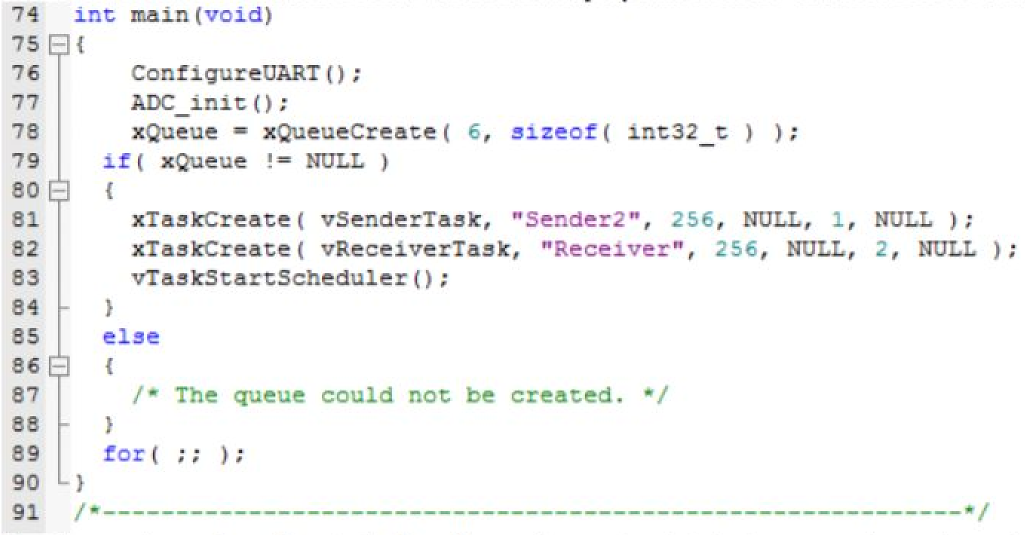
f. //Enable the sequencer.

ADC0 _ACTSS_R |= 0 x 8 ;

____

Question (2):

For the following Cortex M4 FreeRTOS code snippet, assume slice time is 1ms. For the same hardware connected in Question (1). The sender task will Trigger the conversion and send the data to the receiver task. Don't use Global variables array update in order to avoid loss of data.



```c
int main(void)
{
    ConfigureUART();
    ADC_init();
    xQueue = xQueueCreate( 6, sizeof( int32_t ) );
  if( xQueue != NULL )
  {
    xTaskCreate( vSenderTask, "Sender2", 256, NULL, 1, NULL );
    xTaskCreate( vReceiverTask, "Receiver", 256, NULL, 2, NULL );
    vTaskStartScheduler();
  }
  else
  {
    /* The queue could not be created. */
  }
  for( ;; );
}
/*-----------------------------------------------------------*/
```

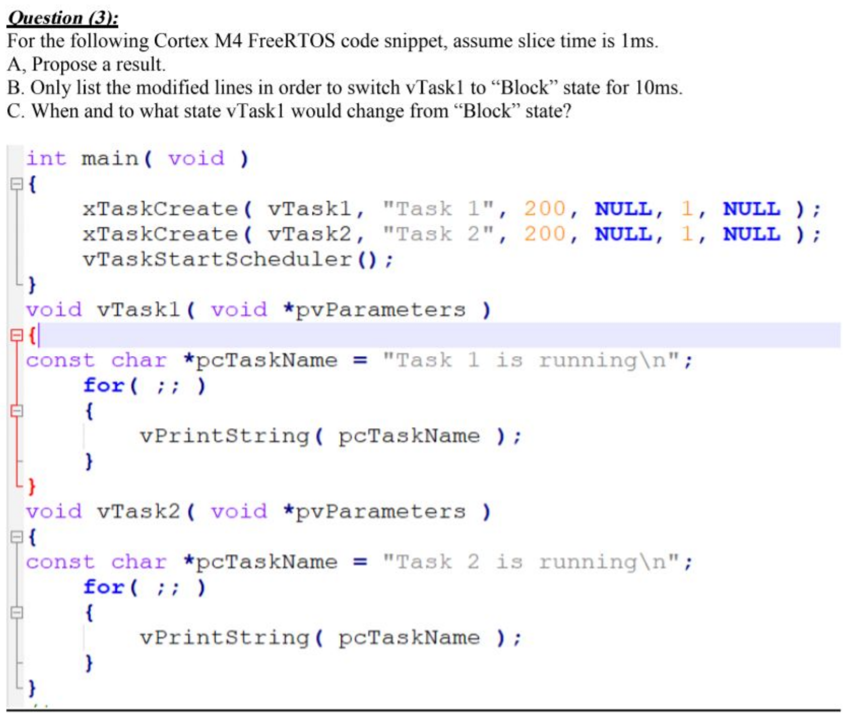
For the sender and receiver tasks handle accelerometer data between sender and receiver tasks with periodicity of 10ms between each 2 updates. *The receiver task prints the readings by UARTprintf, indicating* ***which axis is being printed.***

```
static void vSenderTask( void *pvParameters )
{
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send to the queue.

        The second parameter is the address of the structure being sent.  The
        address is passed in as the task parameter so pvParameters is used
        directly.

        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        if the queue is already full.  A block time is specified because the
        sending tasks have a higher priority than the receiving task so the queue
        is expected to become full.  The receiving task will remove items from
        the queue when both sending tasks are in the Blocked state. */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete, even after waiting for 100ms.
            This must be an error as the receiving task should make space in the
            queue as soon as both sending tasks are in the Blocked state. */
            vPrintString( "Could not send to the queue.\n" );
        }

        /* Allow the other sender task to execute. */
        taskYIELD();
    }
}
```

Listing 39. The implementation of the sending task for Example 11.

```c
static void vReceiverTask( void *pvParameters )
{
/* Declare the structure that will hold the values received from the queue. */
xData xReceivedStructure;
portBASE_TYPE xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* Because it has the lowest priority this task will only run when the
        sending tasks are in the Blocked state.  The sending tasks will only enter
        the Blocked state when the queue is full so this task always expects the
        number of items in the queue to be equal to the queue length - 3 in this
        case. */
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\n" );
        }

        /* Receive from the queue.

        The second parameter is the buffer into which the received data will be
        placed.  In this case the buffer is simply the address of a variable that
        has the required size to hold the received structure.

        The last parameter is the block time - the maximum amount of time that the
        task will remain in the Blocked state to wait for data to be available
        if the queue is already empty.  In this case a block time is not necessary
        because this task will only run when the queue is full. */
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value and the source of the value. */
            if( xReceivedStructure.ucSource == mainSENDER_1 )
            {
             vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
             vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            /* Nothing was received from the queue.  This must be an error
            as this task should only run when the queue is full. */
            vPrintString( "Could not receive from the queue.\n" );
        }
    }
}
```

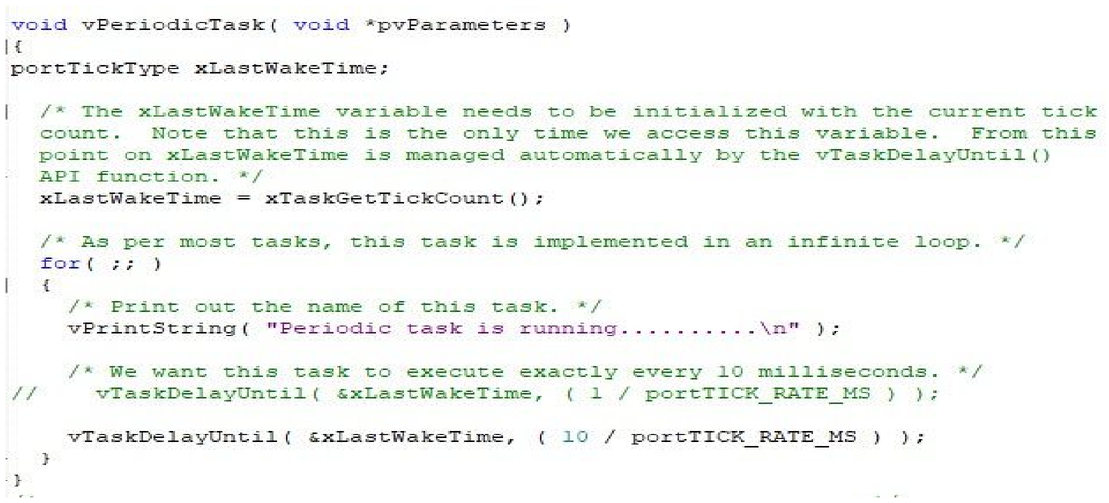
Mid term 2018



Question (3):
For the following Cortex M4 FreeRTOS code snippet, assume slice time is 1ms.
A, Propose a result.
B. Only list the modified lines in order to switch vTask1 to "Block" state for 10ms.
C. When and to what state vTask1 would change from "Block" state?

```
int main( void )
{
    xTaskCreate( vTask1, "Task 1", 200, NULL, 1, NULL );
    xTaskCreate( vTask2, "Task 2", 200, NULL, 1, NULL );
    vTaskStartScheduler();
}
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\n";
    for( ;; )
    {
        vPrintString( pcTaskName );
    }
}
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\n";
    for( ;; )
    {
        vPrintString( pcTaskName );
    }
}
```

a) Task 1 is running
   Task 2 is running
   Task 1 is running
   Task 2 is running
   Task 1 is running
   Task 2 is running
   b)



```
void vPeriodicTask( void *pvParameters )
{
portTickType xLastWakeTime;

  /* The xLastWakeTime variable needs to be initialized with the current tick
  count.  Note that this is the only time we access this variable.  From this
  point on xLastWakeTime is managed automatically by the vTaskDelayUntil()
  API function. */
  xLastWakeTime = xTaskGetTickCount();

  /* As per most tasks, this task is implemented in an infinite loop. */
  for( ;; )
  {
    /* Print out the name of this task. */
    vPrintString( "Periodic task is running..........\n" );

    /* We want this task to execute exactly every 10 milliseconds. */
//    vTaskDelayUntil( &xLastWakeTime, ( 1 / portTICK_RATE_MS ) );

    vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
  }
}
```

c) time slice =1ms
So after 10 time slice vTask1 is moved to ready state to be executed from the os