# EXPERIMENT 9 - DEMONSTRATION OF FREERTOS

LAB CODE AND TITLE - 19CCE263 EMBEDDED COMPUTING LAB
EXPERIMENT NUMBER - 9
DATE - 05/07/2022 (TUESDAY)

* **AIM:**
Implement task management in a multi-tasking system using FreeRTOS.

① **EXAMPLE 7. DEFINING AN IDLE TASK HOOK FUNCTION:**

① Declare a variable that will be incremented by the hook function.
② Idle hook functions MUST be called vApplicationIdleHook(), take no parameters, and return void.
③ This hook function does nothing but increment a counter.
④ The string to print out is passed in via the parameter. Cast this to a character pointer.
⑤ As per most tasks, this task is implemented in an infinite loop.
⑥ Print out the number of this task AND the number of times ulIdleCycleCount has been incremented.
⑦ Delay for a period of 250 milliseconds.

→ **C PROGRAM CODE:-**

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* Stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"
```

```
/* The task function. */
void vTaskFunction( void *pvParameters );

/* A variable that is incremented by the idle task hook function. */
static unsigned long ulIdleCycleCount = 0UL;

/* Define the strings that will be passed in as the task parameters.
These are defined const and off the stack to ensure they remain
valid when the tasks are executing. */
const char *pcTextForTask1 = "Task 1 is running, ulIdleCycleCount = ";
const char *pcTextForTask2 = "Task 2 is running, ulIdleCycleCount = ";

/*-----------------------------------------------------------*/

int main(void)
{
  /* Create the first task at priority 1... */
  xTaskCreate( vTaskFunction, "Task 1", 240, (void *) pcTextForTask1, 1,
NULL);

  /* ... and the second task at priority 2. The priority is the second to
  last parameter. */
  xTaskCreate( vTaskFunction, "Task 2", 240, (void *) pcTextForTask2, 2,
  NULL);

  /* start the scheduler so our tasks start executing. */
  vTaskStartScheduler();

  for(;;);
}

/*-----------------------------------------------------------*/
```

```
void vTaskFunction (void *pvParameters)
{
   char *pcTaskName;

   /* The string to print out is passed in via the parameter. Cast this
   to a character pointer. */
   pcTaskName = (char *) pvParameters;

   /* As per most tasks, this task is implemented in an infinite loop.*/
   for (;;)
   {
     /* Print out the name of this task AND the number of times
        ulIdleCycleCount has been incremented. */
     vPrintStringAndNumber (pcTaskName, ulIdleCycleCount);

     /* Delay for a period. This time we use a call to vTaskDelay()
     which puts the task into the Blocked state until the delay
     period has expired. The delay period is specified in 'ticks'. */
     vTaskDelay (250/ portTICK_RATE_MS);
   }
}
/*-----------------------------------------------------------*/

/* Idle hook functions MUST be called vApplicationIdleHook(), take no
parameters, and return void. */
void vApplicationIdleHook (void)
{
  /* This hook function does nothing but increment a counter. */
  ulIdleCycleCount ++;
}
/*-----------------------------------------------------------*/
```

```
void vApplicationMallocFailedHook (void)
{
  /* This function will only be called if an API call to create a
  task, queue or semaphore fails because there is too little heap
  RAM remaining - and config USE_MALLOC_FAILED_HOOK is set to 1
  in FreeRTOSConfig.h. */
  for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook (xTaskHandle *pxTask,
signed char *pcTaskName)
{
  /* This function will only be called if a task overflow its stack. Note
  that stack overflow checking does slow down the context switch
  implementation and will be only performed if config CHECK_FOR_STACK_
  OVERFLOW is set to either 1 or 2 in FreeRTOSConfig.h. */
  for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationTickHook (void)
{
  /* This example does not use the tick hook to perform any
  processing. The tick hook will only if config USE_TICK_HOOK is set
  to 1 in FreeRTOS Config.h. */
}
```

② **EXAMPLE 8 - CHANGING TASK PRIORITIES:**

① Task 1 is created with the highest priority, so is guaranteed to run first. Task 1 prints out a couple of strings before raising the priority of Task 2 to above its own priority.

(2) Task 2 starts to run (enters the Running state) as soon as it has the highest relative priority. Only one task can be in the Running state at any one time; so, when Task 2 is in the Running state, Task 1 is in the Ready state.

(3) Task 2 prints out a message before setting its own priority back to below that of Task 1.

(4) Task 2 setting its priority back down means Task 1 is once again the highest priority task, so Task 1 re-enters the Running state forcing Task 2 back into the Ready state.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* stellaris library includes. */
/#include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The two task functions. */
void vTask1 (void *pvParameters);
void vTask2 (void *pvParameters);

/* Used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

/*-----------*/
```

```
int main (void)
{
/* Create the first task at priority 2. This time the task parameter
is not used and is set to NULL. The task handle is also not used so
likewise is also set to NULL. */
xTaskCreate (vTask1, "Task1", 240, NULL, 2, NULL);
    /* The task is created at priority 2^. */

/* Create the second task at priority 1 - which is lower than the
priority given to Task 1. Again the task parameter is not used so is
set to NULL. But this time we want to obtain a handle to the task
so pass in the address of the xTask2Handle variable. */
xTaskCreate (vTask2, "Task2", 240, NULL, 1, &xTask2Handle);
    /* The task handle is the last parameter ^^^^^^^^ */

/* start the scheduler so our tasks start executing. */
vTaskStartScheduler();

for(;;);
}
/*-----------------------------------------------------*/
void vTask1 (void *pvParameters)
{
unsigned portBASE_TYPE uxPriority;

/* This task will always run before Task2 as it has the higher
priority. Neither Task1 nor Task2 ever block so both will always be
in either the Running or the Ready state.

Query the priority at which this task is running - passing in
NULL means "return our own priority". */
uxPriority = uxTaskPriorityGet (NULL);
```

```
for( ;; )
{
    /* Print out the name of this task. */
    vPrintString( "Task1 is running\r\n" );

    /* Setting the Task2 priority above the Task1 priority will
    cause Task2 to immediately start running (as then Task2 will
    have the higher priority of the two created tasks). */
    vPrintString( "About to raise the Task2 priority\r\n" );
    vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

    /* Task1 will only run when it has a priority higher than
    Task2. Therefore, for this task to reach this point Task2 must
    already have executed and set its priority back down to 0. */
}
}

/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

    /* Task1 will always run before this task as Task1 has the
    higher priority. Neither Task1 nor Task2 ever block so will
    always be in either the Running or the Ready state.

    Query the priority at which this task is running - passing in
    NULL means "return our own priority". */
    uxPriority = uxTaskPriorityGet( NULL );
```

P.T.O.

```
    for (;;)
    {
        /* For this task to reach this point Task1 must have
        already run and set the priority of this task higher than its
        own. */

        /* Print out the name of this task. */
        vPrintString ("Task2 is running\r\n");

        /* set our priority back down to its original value. Passing
        in NULL as the task handle means "change our own priority".
        Setting the priority below that of Task1 will cause Task1 to
        immediately start running again. */
        vPrintString ("About to lower the Task2 priority\r\n");
        vTaskPrioritySet (NULL, (uxPriority - 2));
    }
}
/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook (void)
{
    /* This function will only be called if an API call to create a
    task, queue or semaphore fails because there is too little heap
    RAM remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
    FreeRTOSConfig.h. */
    for (;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook (xTaskHandle *pxTask,
signed char *pcTaskName)
{
```

```
    /* This function will only be called if a task overflows its
    stack. Note that stack overflow checking does slow down the
    context switch implementation and will only be performed if
    configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
    FreeRTOSConfig.h. */
    for( ;; );
}
/*-----------------------------------------------------------*/

void
vApplicationIdleHook( void )
{
    /* This example does not use idle hook to perform any
    processing. The idle hook will only be called if configUSE_IDLE_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/


void vApplicationTickHook( void )
{
    /* This example does not use the tick hook to perform any processing.
    The tick hook will only be called if configUSE_TICK_HOOK is set to 1 in
    FreeRTOSConfig.h. */
}
```

(3) **EXAMPLE 9. DELETING TASKS :**

(1) Task 1 is created by main() with priority 1. When it runs, it creates Task 2 at priority 2. Task 2 is now the highest priority task, so it starts to execute immediately. ~~The source for main().~~

(2) Task 2 does nothing but delete itself. It could delete itself by passing NULL to vTaskDelete() but instead, for demonstration purposes, it uses its own task handle. ~~The source~~

③ When Task 2 has been deleted, Task1 is again the highest priority task, so continues executing - at which point it calls vTaskDelay() to block for a short period.

④ The Idle task executes while Task 1 is in the blocked state and frees the memory that was allocated to the now deleted Task 2.

⑤ When Task1 leaves the blocked state it again becomes the highest priority Ready state task and so pre-empts the Idle task. When it enters the Running state it creates Task 2 again, and so it goes on.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"

/* stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The two task functions. */
void vTask1 (void *pvParameters);
void vTask2 (void *pvParameters);

/* Used to hold the handle of Task 2. */
xTaskHandle xTask2Handle;

/*-----------------------------------------------------------*/
```

PTO →

```
int main (void)
{
    /* Note that this project uses the heap_2.c sample memory
    allocator, as this implements vPortFree() as well as pvPortMalloc().
    The free function is required to release heap memory when a task
    is deleted.

    Create the first task to priority 1. This time the task parameter
    is not used and is set to NULL. The task handle is also not used
    so likewise is also set to NULL. */
    xTaskCreate( vTask1, "Task1", 240, NULL, 1, NULL );
        /* The task is created at priority 1 ^. */

    /* Start the scheduler so our tasks start executing. */
    vTaskStartScheduler();

    for( ;; );
}
/*-----------------------------------------------------------*/

void vTask1( void *pvParameters )
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task1 is running\n" );

        /* Create task 2 at a higher priority. Again the task perimeter
        is not used so is set to NULL - BUT this time we want to
        obtain a handle to the task so pass in the address of the
        xTask2Handle variable. */
```

```
    xTaskCreate( vTask2, "Task 2", 240, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter ^^^^^^^^^^^ */

    /* Task2 has/had the higher priority, so for Task1 to reach here
    Task2 must have already executed and deleted itself. Delay for
    100 milliseconds. */
    vTaskDelay( xDelay100ms );
  }
}
/*-----------------------------------------------------------*/

void vTask2( void *pvParameters )
{
  /* Task 2 does nothing but delete itself. To do this it would call
  vTaskDelete() using a NULL parameter, but instead and purely for
  demonstration purposes it instead calls vTaskDelete() with its
  own task handle. */
  vPrintString( "Task2 is running and about to delete itself\n" );
  vTaskDelete( xTask2Handle );
}
/*-----------------------------------------------------------*/

void vApplicationMallocFailedHook( void )
{
  /* This function will only be called if an API call to create a task,
  queue or semaphore fails because there is too little heap RAM
  remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
  FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/
```

```
void vApplicationStackOverflowHook (xTaskHandle *pxTask, signed
char *pcTaskName)
{
   /* This function will only be called if a task overflows its stack.
   Note that stack overflow checking does slow down the context
   switch implementation and will only be performed if
   configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
   FreeRTOSConfig.h. */
   for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook (void)
{
  /* This example does not use the idle hook to perform any
   processing. The idle hook will only be called if configUSE_IDLE_HOOK
   is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/

void vApplicationTickHook (void)
{
  * This example does not use the tick hook to perform any
   processing. The tick hook will only be called if configUSE_TICK_HOOK
   is set to 1 in FreeRTOSConfig.h. */
}
```

① **EXAMPLE 10. BLOCKING WHEN RECEIVING FROM A QUEUE:**

I. Implementation of the task that writes to the queue -

① Two instances of this task are created, one that writes continuously the value 100 to the queue, and another that writes continuously the value 200 to the same queue.

② The task parameter is used to pass these values into each task instance.

III. Implementation of the task that receives data from the queue -

(1) The receiving task specifies a block time of 100 milliseconds, so will enter the Blocked state to wait for data to become available.

(2) It will leave the Blocked state when either data is available on the queue or 100 milliseconds passes without data becoming available.

(3) In this example, the 100 milliseconds timeout should never expire, as there are two tasks writing continuously to the queue.

IV. Definition of the main() function -

(1) This simply creates the queue and the three tasks before starting the scheduler.

(2) The queue is created to hold a maximum of five long values, even though the priorities of the tasks are set such that the queue will never contain more than one item at a time.

→ C PROGRAM CODE :-

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"

/* Stellaris library includes. */
// #include "hw_types.h"
// #include "hw_memmap.h"
// #include "sysctl.h"

/* Demo includes. */
#include "basic_io.h"

/* The tasks to be created. Two instances are created of the sender
task while only a single instance is created of the receiver task. */
static void vSenderTask (void *pvParameters);
static void vReceiverTask (void *pvParameters);
```

```
/* Declare a variable of type xQueueHandle. This is used to store
   the queue that is accessed by all three tasks. */
xQueueHandle xQueue;

int main (void)
{
  /* The queue is created to hold a maximum of 5 long values. */
  xQueue = xQueueCreate (5, sizeof (long));

  if (xQueue != NULL)
  {
    /* Create two instances of the task that will write to the
       queue. The parameter is used to pass the value that the
       task should write to the queue, so one task will
       continuously write 200 to the queue. Both tasks are
       created at priority 1. */
    xTaskCreate (vSenderTask, "Sender 1", 240, (void *) 100, 1, NULL);
    xTaskCreate (vSenderTask, "Sender 2", 240, (void *) 200, 1, NULL);

    /* Create the task that will read from the queue. The task
       is created with priority 2, so above the priority of the
       sender tasks. */
    xTaskCreate (vReceiverTask, "Receiver", 240, NULL, 2, NULL);

    /* start the scheduler so the created tasks start executing. */
    vTaskStartScheduler();
  }
  else
  {
    /* The queue could not be created. */
  }
```

```
    /* If all is well we will never reach here as the scheduler will
    now be running the tasks. If we do reach here then it is likely
    that there was insufficient heap memory available for a
    resource to be created. */
    for (;;);
}
/*-----------------------------------------------------------*/

static void vSenderTask (void *pvParameters)
{
  long lValueToSend;
  portBASE_TYPE xStatus;

  /* Two instances are created of this task so the value that is
  sent to the queue is passed in via the task parameter rather
  than be hard coded. This way each instance can use a different
  value. Cast the parameter to the required type. */
  lValueToSend = (long) pvParameters;

  /* As per most tasks, this task is implemented within an
  infinite loop. */
  for (;;)
  {
    /* The first parameter is the queue to which data is being
    sent. The queue was created before the scheduler was started,
    so before this task started to execute.

    The second parameter is the address of the data to be sent.

    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 should the queue already be full. In this case we
    don't specify a block time because there should always be space in the
    queue. */
```

```
xStatus = xQueueSendToBack (xQueue, &lValueToSend, 0);

    if (xStatus != pdPass)
    {
      /* We could not write to the queue because it was full - this
      must be an error as the queue should never contain more than
      one item! */
      vPrintString ("Could not send to the queue.\r\n");
    }

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

static void vReceiverTask (void *pvParameters)
{
/* Declare the variable that will hold the values received from
 the queue. */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;

    /* This task is also defined within an infinite loop */
    for (;;)
    {
      /* As this task unblocks immediately that data is written to
      the queue this call should always find the queue empty. */
      if (uxQueueMessagesWaiting (xQueue) != 0)
      {
        vPrintString ("Queue should have been empty!\r\n");
      }
```

```
/* The first parameter is the queue from which data is to be
received. The queue is created before the scheduler is started,
and therefore before this task runs for the first time.

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 data.

The last parameter is the block time - the maximum amount of time
that the task should remain in the Blocked state to wait for
data to be available should the queue already be empty. */
xStatus = xQueueReceive (xQueue, &lReceivedValue, xTicksToWait);

if (xStatus == pdPASS)
{
  /* Data was successfully received from the queue, print out
  the received value. */
  vPrintStringAndNumber ("Received = ", lReceivedValue);
}
else
{
  /* We did not receive anything from the queue even after
  waiting for 100 ms. This must be an error as the sending tasks
  are free running and will be continuously writing to the queue. */
  vPrintString ("Could not receive from the queue.\r\n");
}
}
}
/*-----------------------------------------------------------*/


void vApplicationMallocFailedHook (void)
{
  /* This function will only be called if an API call to create a task,
  queue or semaphore fails because there is too little heap RAM
```

```
    remaining - and configUSE_MALLOC_FAILED_HOOK is set to 1 in
    FreeRTOSConfig.h. */
    for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed
char *pcTaskName)
{
    /* This function will only be called if a task overflows its stack.
    Note that stack overflow checking does slow down the context
    switch implementation and will only be performed if
    configCHECK_FOR_STACK_OVERFLOW is set to either 1 or 2 in
    FreeRTOSConfig.h. */
    for(;;);
}
/*-----------------------------------------------------------*/

void vApplicationIdleHook(void)
{
    /* This example does not use the idle hook to perform any
    processing. The idle hook will only be called if configUSE_IDLE_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
/*-----------------------------------------------------------*/

void vApplicationTickHook(void)
{
    /* This example does not use the tick hook to perform any
    processing. The tick hook will only be called if configUSE_TICK_HOOK
    is set to 1 in FreeRTOSConfig.h. */
}
```

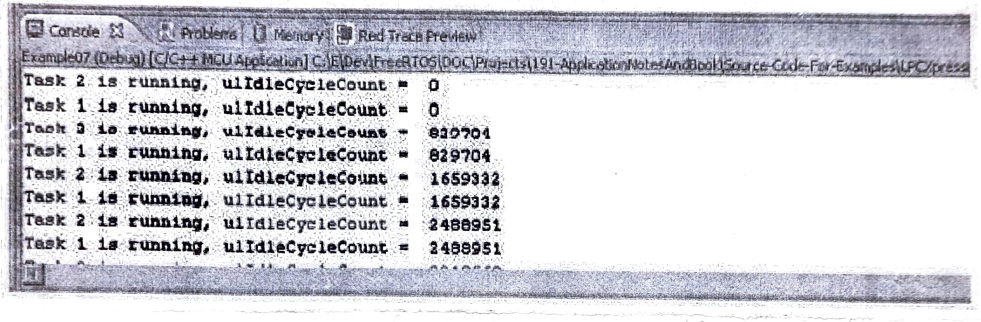


Figure 1 - The output produced when example 7 is executed
It shows that the idle task hook function is called approximately 83000-times between each iteration of the application tasks.

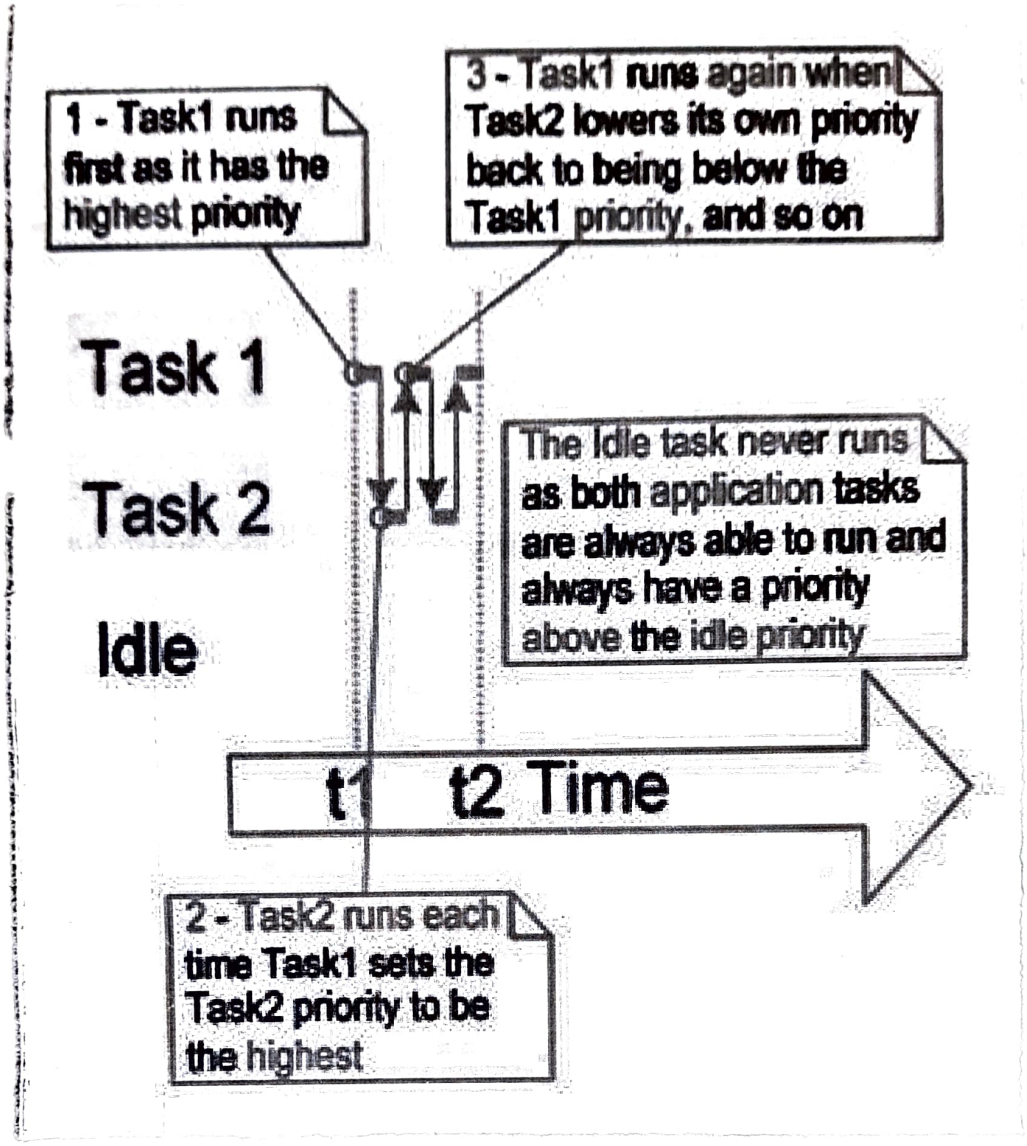


Figure 2 - The sequence of task execution when running Example 2.

It demonstrates the sequence in which the Example 8 tasks execute, with the resultant output shown in Figure 3

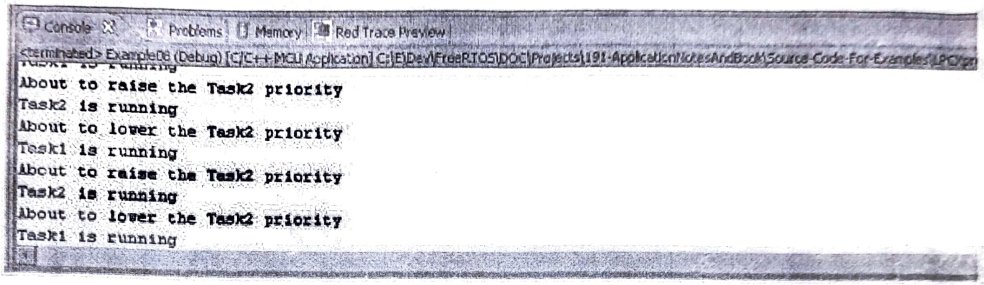


Figure 3- The output produced when Example 6 is executed

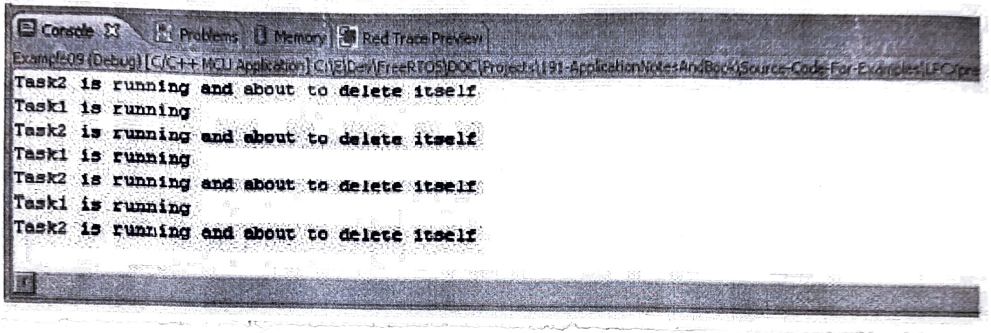


Figure 4- The output produced when Example 9 is executed

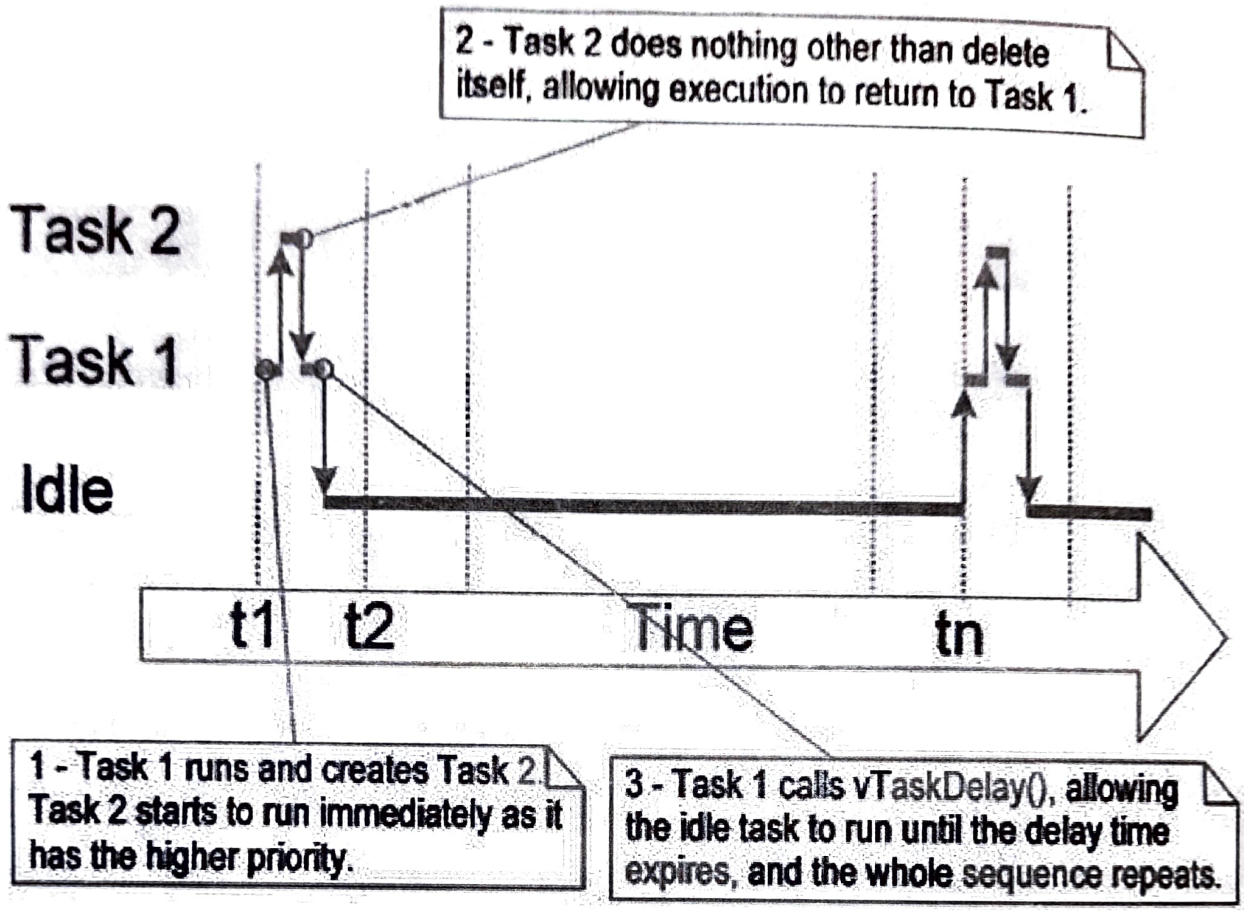


Figure 5- The execution sequence for Example 2

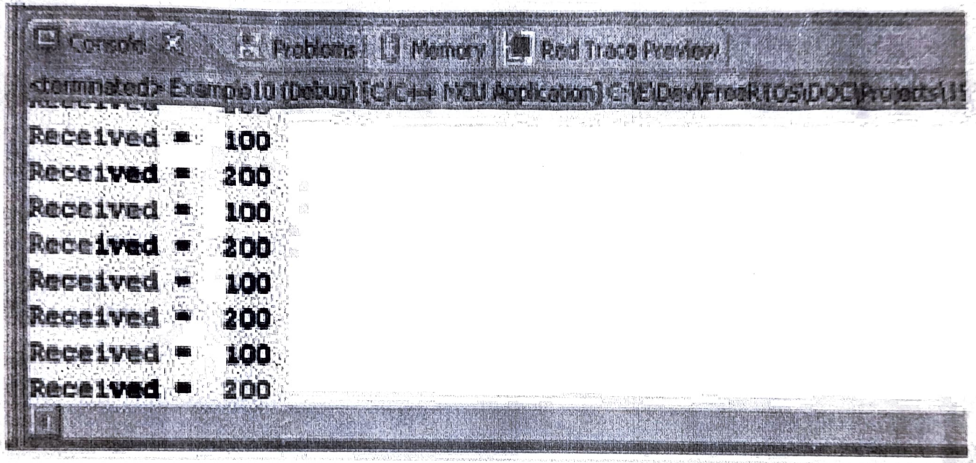


Figure 5- The output produced when Example 10 is executed.

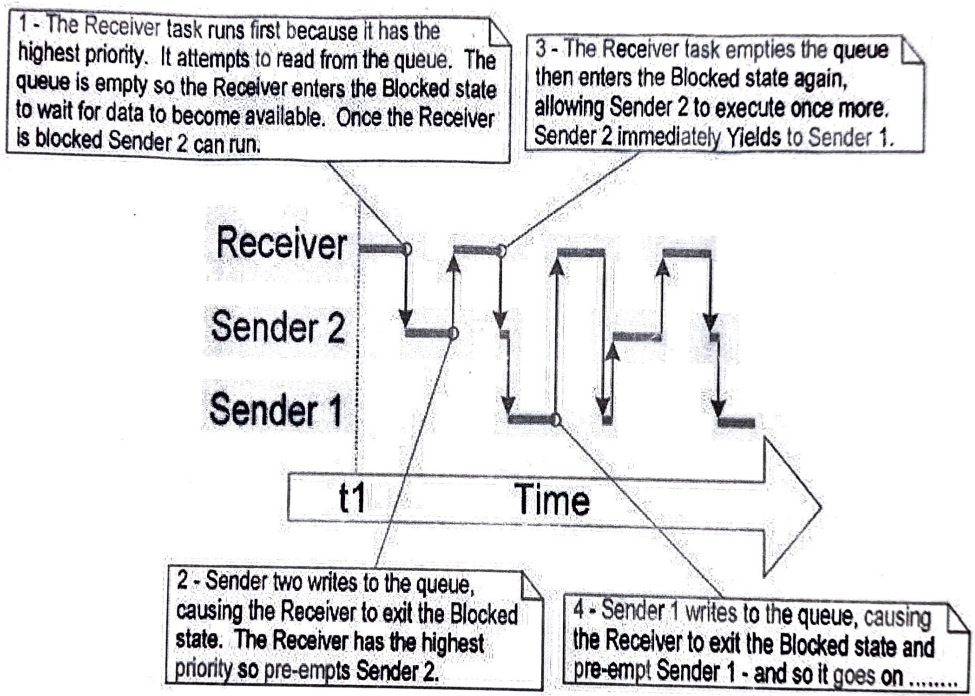


Figure 7 - The Execution sequence for Example 10

* **RESULT:**

Thus, implemented task management in a multi-tasking system using FreeRTOS. All the simulation results were verified successfully.