



FreeRTOS - Practica 3

≡ Autor/Fuente	Schattmann, Tomas Emanuel; Tedesco, Aldana; Mengoni, Emiliano
≡ Materia	Sistemas de tiempo real

Ejercicio 1 - Tareas

FreeRTOS nos brinda la posibilidad de elegir el modo de interacción que tendrán las tareas, estos pueden ser cooperativo, apropiativo o mixto. Dependiendo de la funcionalidad que estemos intentando implementar podemos seleccionar uno u otro teniendo en cuenta sus ventajas y desventajas.

En el modo apropiativo el scheduler del sistema operativo puede quitar o brindar el control del CPU a las tareas en cualquier momento. En cambio en el modo cooperativo, cuando se le da el control del CPU a una tarea, ésta se ejecutará hasta que explícitamente brinde el control a otra tarea. El modo mixto permite implementar aspectos de los dos modos antes descritos.

Es por esto, que al crear y manejar tareas debemos tener en cuenta en qué modo estamos trabajando.

Cuando definimos una tarea debemos tener en cuenta los siguientes puntos:

- Funcionamiento de las tareas: Una tarea, es considerada como un programa pequeño. Tiene un entrypoint y normalmente deben ejecutarse en un loop infinito, sin tener un “return” y sin poder ejecutar más allá del final de la tarea.
- Parámetros de creación de la tarea: Cada tarea implementada tiene un nombre, una pila propia, una prioridad determinada, una serie de parámetros que se deben pasar por referencia y un handler.

Al crear una tarea debemos contemplar que tamaño de la pila que deseamos que tenga, que prioridad y que handler.

Para crear una tarea se utiliza la API xTaskCreate().

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE pvTaskCode,
    const signed portCHAR * const pcName,
    unsigned portSHORT usStackDepth,
    void *pvParameters,
    unsigned portBASE_TYPE uxPriority,
    xTaskHandle *pxCreatedTask
);
```

pvTaskCode	Es un puntero a donde esta declarada la funcion.
pcName	Es un nombre descriptivo para la tarea. No es usado por FreeRTOS, simplemente es incluido para un debug mas amigable.
usStackDepth	Indica al kernel la cantidad maxima de palabras (32 bits) que la pila que se va a implementar puede soportar. Su tamaño maximo no debe superar
pvParametres	Se pasa un puntero de tipo void (void *), el valor asignado a pvParameters sera el valor pasado a la funcion.
uxPriority	Define la prioridad de la tarea, su rango es de 0 a configMAX_PRIORITIES - 1.
pxCreatedTask	Se utiliza para pasar un handler a la funcion siendo creada. El handler puede por ejemplo cambiar la prioridad de la tarea en un momento dado o mismo eliminarla.

La función tiene 2 valores de retorno:

- pdTrue: Indica que la tarea fue creada de manera exitosa.
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY: Indica que la tarea no pudo ser creada debido a que no hay memoria RAM suficiente para alojar las estructuras de datos de la tarea y su pila.

Luego para la gestión de la tarea se utiliza también la API, entre las distintas opciones tenemos:

- vTaskDelete: Finaliza una tarea.
- vTaskDelay: Detiene una tarea un tiempo determinado.
- vTaskPrioritySet: Asigna prioridad de una tarea existente.
- uxTaskPriorityGet: Consulta la prioridad de una tarea existente.
- vTaskSuspended: Suspende una tarea.
- vTaskResume: Resume la ejecución de una tarea.

Ejercicio 2 - Implementación de tareas

En este ejercicio utilizamos un Arduino UNO, para el cual hemos desarrollado el siguiente código:

```
// Incluimos la librería Arduino FreeRTOS
#include <Arduino_FreeRTOS.h>

void Task1(void *pvParameters);
void Task2(void *pvParameters);
void Task3(void *pvParameters);

void setup() {
    Serial.begin(230400);

    /**
     * Creamos las tareas
     */
    xTaskCreate(Task1, // Task function
                "Tarea1", // Task name
                128, // Stack size
                NULL,
                0, // Priority
                NULL); // Task handler

    xTaskCreate(Task2, // Task function
```

```

    "Tarea1", // Task name
    128, // Stack size
    NULL,
    0, // Priority
    NULL); // Task handler

xTaskCreate(Task3, // Task function
    "Tarea1", // Task name
    128, // Stack size
    NULL,
    0, // Priority
    NULL); // Task handler

vTaskStartScheduler();
}

void loop() {}

void Task1(void *pvParameters) {

    const char *pcTaskName = "Tarea 1 se esta ejecutando \n\r";
    for (;;) {
        Serial.println(pcTaskName);

    }
}

void Task2(void *pvParameters) {

    const char *pcTaskName = "Tarea 2 se esta ejecutando \n\r";
    for (;;) {
        Serial.println(pcTaskName);
    }
}

void Task3(void *pvParameters) {

    const char *pcTaskName = "Tarea 3 se esta ejecutando \n\r";
    for (;;) {
        Serial.println(pcTaskName);
    }
}

```

En base a este código hicimos 4 pruebas de ejecución variando en cada una de ellas la prioridad de cada una de las tareas.

Aquí se presenta una tabla de los valores de prioridad en conjunto con las observaciones realizadas en base al output serie:

Prioridad Task 1	Prioridad Task 2	Prioridad Task 3	Observaciones
0	0	0	Aproximadamente cada tarea imprime su linea correspondiente

			13 veces antes de que se entregue el control del CPU a otra. Se observa equidad en el tiempo de ejecución asignado a cada tarea.
0	0	1	Se observa casi un monopolio de ejecución de la tarea 3.
0	1	1	La tarea 2 y 3 predominan y reparten equitativamente el uso de la CPU, la tarea 1 tiene ejecuciones esporádicas.
0	1	2	Se observa nuevamente un monopolio de ejecución de la tarea 3.

Ejercicio 3 - Herramientas de sincronización

Considerando que cada una de las tareas es un pequeño programa en si mismo, para el funcionamiento óptimo del sistema que estemos desarrollando necesitaremos brindar comunicación entre estas.

En base a esto surgen los 2 tipos de herramientas de sincronización:

- Herramientas que comunican tareas con tareas: Colas.
- Herramientas que comunican tarea con rutina de interrupción y viceversa: Semáforos.

Colas

Una cola puede albergar un número finito de datos de longitud fija. El máximo número de objetos que puede albergar es lo que llamamos el “largo” de la cola.

Normalmente se utilizan colas como buffers FIFO (First In First Out) donde se agregan los datos por el final y se recuperan por el inicio. Con posibilidad de poder escribir también datos al inicio de la cola.

Las colas son accedidas por múltiples tareas, sin embargo no pertenecen ni son asignadas por ninguna de ellas. Esto demanda un sistema de sincronización para la lectura y escritura de las mismas.

Cuando una tarea desea leer un dato de una cola, existe la posibilidad de especificar un tiempo de bloqueo. Esto declara el tiempo que la tarea debe mantenerse en el estado bloqueado para esperar a que se complete la lectura del dato.

A su vez existen tiempos de bloqueos para cuando se quiere escribir un dato en la cola. Este tiempo indica cuanto debe esperar la tarea en el estado bloqueado a que se libere memoria para poder escribir el dato.

Para usar esta herramienta poseemos la API de gestión de colas que nos permite crear una cola, añadir o quitar datos, leer un dato y no sacarlo o consultar la cantidad de lugares disponibles.

Semáforos

Los semáforos se utilizan para bloquear y desbloquear tareas, principalmente cuando ocurre una interrupción. Esto permite que la mayor parte del procesamiento de las mismas se haga en una tarea sincronizada y no en la rutina de servicio de la interrupción.

En FreeRTOS existen dos tipos de semáforos, binarios y contadores que se gestionan mediante la API de semáforos.

Semáforos binarios

Estos tipos de semáforos pueden ser usados para bloquear o desbloquear tareas en momentos determinados, principalmente cuando ocurre una interrupción. Esto se hace ya que la ISR en lo debe durar lo menos posible, entonces al ejecutarse da valor un semáforo que luego activa una tarea sincronizada que resuelve la interrupción.

Cuando la tarea sincronizada termina, quita el valor al semáforo para desactivarse.

Las tareas sincronizadas suelen tener mas prioridad que el resto de las tareas ya que deben apropiarse del CPU en momentos dados para resolver los eventos de interrupción.

Semáforos contadores

Se utilizan cuando tenemos múltiples recursos idénticos disponibles y varias tareas las cuales pueden solicitar acceso a estos recursos. Si una tarea adquiere la posesión del recurso, se decrementa el contador, y cuando una tarea libera un recurso, se incrementa.

Esto permite una gestión y acceso a recursos de manera optima.

Ejercicio 4

A)

```
#include <Arduino_FreeRTOS.h>
#include "task.h"
#include "semphr.h"
#include "stdio.h"

// Definir semáforos
SemaphoreHandle_t xSemaphore;

// Prototipos de funciones de tarea
void tarea1(void *pvParameters);
void tarea2(void *pvParameters);
void tarea3(void *pvParameters);

void setup(void) {
    // Crear semáforos
    xSemaphore = xSemaphoreCreateCounting(3,0);

    // inicializamos en uno
    xSemaphoreGive(xSemaphore);

    Serial.begin(9600);

    if (xSemaphore != NULL) {
        // Crear las tareas
        xTaskCreate(tarea1, "Tarea 1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea3, "Tarea 3", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea2, "Tarea 2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

        // Iniciar el planificador de FreeRTOS
        vTaskStartScheduler();
    }
}

void loop() {}

void tarea1(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 1) {
            Serial.println("Tarea 1 - ");
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
        }
    }
}
```

```

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

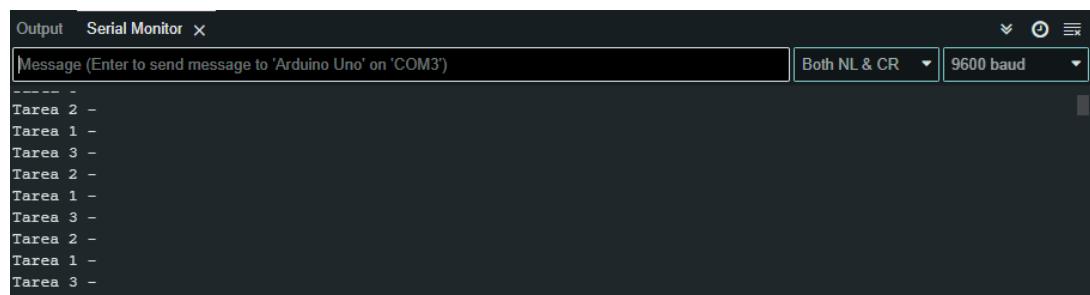
void tarea3(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 3) {
            Serial.println("Tarea 3 - ");
            xSemaphoreTake(xSemaphore, portMAX_DELAY);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

void tarea2(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 2) {
            Serial.println("Tarea 2 - ");
            xSemaphoreTake(xSemaphore, portMAX_DELAY);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

```



B)

```

#include <Arduino_FreeRTOS.h>
#include "task.h"
#include "semphr.h"
#include "stdio.h"

// Definir semáforos
SemaphoreHandle_t xSemaphore;

```

```

// Prototipos de funciones de tarea
void tarea1(void *pvParameters);
void tarea2(void *pvParameters);
void tarea3(void *pvParameters);

void setup(void) {
    // Crear semáforos
    xSemaphore = xSemaphoreCreateCounting(3,3);

    Serial.begin(9600);

    if (xSemaphore != NULL) {
        // Crear las tareas
        xTaskCreate(tarea1, "Tarea 1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea2, "Tarea 2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea3, "Tarea 3", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

        // Iniciar el planificador de FreeRTOS
        vTaskStartScheduler();
    }
}

void loop() {}

void tarea1(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 0) {
            Serial.println("Tarea 1 - ");
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

void tarea3(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 1) {
            Serial.println("Tarea 3 - ");
            xSemaphoreTake(xSemaphore, portMAX_DELAY);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

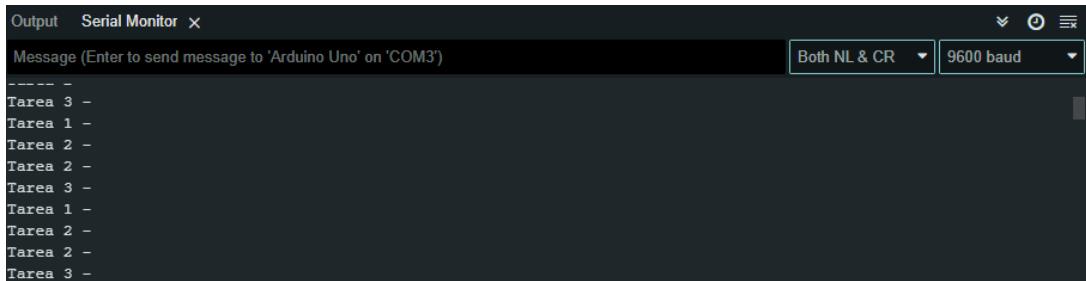
void tarea2(void *pvParameters) {

```

```

int semcount = 0;
for (;;) {
    // Esperar a que se libere el semáforo de Tarea 1
    semcount = uxSemaphoreGetCount(xSemaphore);
    if (semcount == 2 || semcount == 3) {
        Serial.println("Tarea 2 - ");
        xSemaphoreTake(xSemaphore, portMAX_DELAY);
    }
    vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
}
}

```



C)

```

#include <Arduino_FreeRTOS.h>
#include "task.h"
#include "semphr.h"
#include "stdio.h"

// Definir semáforos
SemaphoreHandle_t xSemaphore;

// Prototipos de funciones de tarea
void tarea1(void *pvParameters);
void tarea2(void *pvParameters);
void tarea3(void *pvParameters);

void setup(void) {
    // Crear semáforos
    xSemaphore = xSemaphoreCreateCounting(4,4);

    Serial.begin(9600);

    if (xSemaphore != NULL) {
        // Crear las tareas
        xTaskCreate(tarea1, "Tarea 1", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea2, "Tarea 2", configMINIMAL_STACK_SIZE, NULL, 1, NULL);
        xTaskCreate(tarea3, "Tarea 3", configMINIMAL_STACK_SIZE, NULL, 1, NULL);

        // Iniciar el planificador de FreeRTOS
        vTaskStartScheduler();
    }
}

```

```

}

void loop() {}

void tarea1(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 1) {
            Serial.println("Tarea 1 - ");
            xSemaphoreTake(xSemaphore, portMAX_DELAY);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

void tarea3(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 4 || semcount == 3 || semcount == 2) {
            Serial.println("Tarea 3 - ");
            xSemaphoreTake(xSemaphore, portMAX_DELAY);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

void tarea2(void *pvParameters) {
    int semcount = 0;
    for (;;) {
        // Esperar a que se libere el semáforo de Tarea 1
        semcount = uxSemaphoreGetCount(xSemaphore);
        if (semcount == 0) {
            Serial.println("Tarea 2 - ");
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
            xSemaphoreGive(xSemaphore);
        }

        vTaskDelay(pdMS_TO_TICKS(500)); // Retardo de 500 milisegundos
    }
}

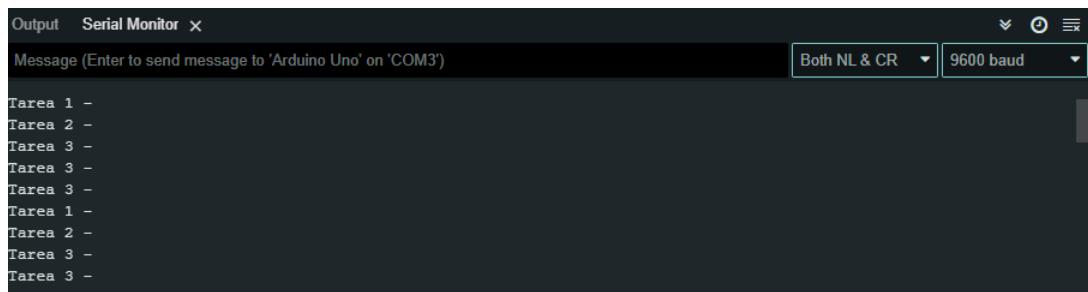
```

Output Serial Monitor X

Message (Enter to send message to 'Arduino Uno' on 'COM3')

Tarea 1 -
Tarea 2 -
Tarea 3 -
Tarea 3 -
Tarea 1 -
Tarea 2 -
Tarea 3 -
Tarea 3 -

Both NL & CR 9600 baud

A screenshot of the Arduino Serial Monitor window. The title bar says "Output Serial Monitor X". The main area shows the text "Message (Enter to send message to 'Arduino Uno' on 'COM3')". Below that, there is a list of task names followed by a hyphen and a space character. The tasks listed are Tarea 1, Tarea 2, Tarea 3, Tarea 3, Tarea 1, Tarea 2, Tarea 3, and Tarea 3. At the bottom right of the monitor window, there are two dropdown menus: "Both NL & CR" and "9600 baud".