UNIVERSIDAD NEBRIJA

MEMORIA PRACTICA 1
SISTEMAS TOLERANTES A FALLO EN TIEMPO REAL

José María Fernández Gómez.

OBJETIVOS

En esta practica se va a tratar de crear un proyecto que emplee el sistema operativo en tiempo real proporcionado, importando todos sus módulos y funciones para que puedan ser empleadas por un programa main que nosotros modificaremos.

Este programa main se encargará de crear 6 tareas o tasks gestionadas por el rtos

Y tratarán de alternar el estado de 6 LEDs diferentes.

FUNCIONES

void TASK_LED1(void);

void TASK_LED2(void);

void TASK_LED3(void);

void TASK_LED4(void);

void TASK_LED5(void);

void TASK_LED6(void);

Estos de aquí son los prototipos de las funciones declaradas gestionadas por el sistema operativo en tiempo real, cada una de ellas tratará de gestionar un LED individualmente, so reciben ningún parámetro y tampoco devuelven nada.

OS_TCB task_led1_TCB;

OS_TCB task_led2_TCB;

OS_TCB task_led3_TCB;

OS_TCB task_led4_TCB;

OS_TCB task_led5_TCB;

OS_TCB task_led6_TCB;

A continuación, se tienen que crear los TCB o bloques del control de tarea correspondientes a cada hilo o task, como se indica en las líneas superiores.

Por último, han de declararse y reservar memoria para el stack de cada hilo, al ser un sistema de 32 bits se reserva un array de 1024 posiciones del tipo cpu_stk.

```
CPU_STK task_led1_STK[1024];
CPU_STK task_led2_STK[1024];
CPU_STK task_led3_STK[1024];
CPU_STK task_led4_STK[1024];
CPU_STK task_led5_STK[1024];
CPU_STK task_led6_STK[1024];
```

FUNCION MAIN

```
OSTaskCreate(
        (OS_TCB *) & task_led1_TCB,
        (CPU_CHAR *) "Tarea 1. Control led y mensaje uart",
        (OS_TASK_PTR) TASK_LED1,
        (void *) 0,
        (OS_PRIO) 3,
        (CPU_STK *) & task_led1_STK[0],
        (CPU_STK_SIZE) 0u,
        (CPU_STK_SIZE) 1024u,
        (OS_MSG_QTY) 0u,
        (OS_TICK) 10u, //10*System Tick period
        (void *) 0,
        (OS_OPT) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
        (OS_ERR *) & os_err);
```

Dentro del flujo principal del RTOS se tiene que crear el hilo a nivel del sistema operativo e indicarle que función tiene que ejecutar, esto lo hacemos con el prototipo mostrado en la parte superior, primero indicándole el tcb donde inicializarse, después el output en formato string (o * char) que irá indicando la CPU, la función como tal a realizar pasada por referencia, la prioridad a la hora de ser interrumpida (se le ha asignado a todas la misma prioridad al no haber interrupciones en esta práctica) y por último la referencia a la zona de memoria reservada para su pila o stack. También se le pasan otra serie de parámetros menos relevantes como el tamaño de la pila o la velocidad en ticks a ejecutar la tarea. Este mismo prototipo se ha de repetir 6 veces, correspondiendo cada declaración a cada hilo.

DECLARACIÓN DE FUNCIONES

En el siguiente paso, se codifica el comportamiento de cada uno de los hilos, esta es la parte más funcional del RTOS, ya que es la parte en la que realiza las funciones para lo que lo hemos diseñado. En este caso solo se encarga de transmitir por pantalla un mensaje y alternar el estado de un LED. Esto se repite en varias tareas como se requiere en el guion de la práctica. Al final de cada hilo se incluye una instrucción de delay. Este delay es el delay eficiente ya que permite al RTOS replanificar y no estar gastando NOPS, como se trabajaba previamente.

```
void TASK_LED1(void) {
    OS_ERR os_err;

    while (1) {
        sprintf(txdata, "TASK1\r\n");
        EnviarString(txdata, Uart3);
        LED_GREEN = !LED_GREEN_Read;
        OSTimeDly(100, OS_OPT_TIME_DLY, &os_err); //100 ticks del RTOS ->
100ms

    }
}
```