Implementación de tareas en FreeRTOS sobre Arduino Uno

Microprocesadores y Sistemas Embebidos
Prof. Carlos Xavier Rosero

1. Introducción

Un sistema operativo existente dentro de los dispositivos empotrados (embebidos, integrados) tales como microcontroladores, se denomina sistema operativo en tiempo real (RTOS). La temporización juega un papel importante en las tareas en tiempo real dentro de un RTOS ya que éstas son deterministas. El determinismo se asocia a que el tiempo de respuesta a cualquier evento es siempre constante, de modo que se puede garantizar que cualquier evento en particular ocurra en un tiempo fijo. Un RTOS está diseñado para ejecutar aplicaciones con sincronización muy precisa y un alto grado de confiabilidad; también ayuda en la multitarea con un solo núcleo. Un tutorial interesante sobre esta temática se puede encontrar en
`https://circuitdigest.com/article/understanding-rtos-and-how-to-use-it-for-embedded-systems`

En este laboratorio, se utiliza FreeRTOS para implementar tareas en Arduino Uno. FreeRTOS es una clase de RTOS que es lo suficientemente pequeño como para ejecutarse en microcontroladores de 8/16 bits, aunque su uso no se limita a estos microcontroladores. Es completamente de código abierto y su código está disponible en Github. Si se conocen algunos conceptos básicos de RTOS, entonces es fácil usar FreeRTOS porque tiene una interfaz de programación de aplicaciones (API) bien documentada que se puede usar directamente en el código sin conocer la parte de backend. La documentación completa de FreeRTOS se puede encontrar en
`https://www.freertos.org/Documentation/RTOS_book.html`

2. Objetivos

- Familiarizarse con el sistema operativo de tiempo real FreeRTOS.
- Conocer sobre creación y administración de tareas en FreeRTOS.
- Implementar tareas que realicen operaciones concurrentes en Arduino Uno.

3. ¿Cómo funciona un RTOS?

Tomado de `https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial1-creating-freertos-task-to-blink-led-in-arduino-uno`

Antes de empezar a trabajar con RTOS, se debe definir qué es una *tarea*. Es una pieza de código que se puede programar en la CPU para ser ejecutada. Por lo tanto, si desea realizar alguna tarea, debe programarse utilizando el retraso del kernel o el

uso de interrupciones; este trabajo lo realiza el *scheduler* (planificador) presente en el kernel. En un procesador de un solo núcleo, el planificador ayuda a que las tareas se ejecuten en un período de tiempo particular, pero parece que se están ejecutando diferentes tareas simultáneamente. Cada tarea se ejecuta de acuerdo con la prioridad que se le da.

3.1. Ejemplo con una tarea sencilla

Se desea crear una tarea para que un LED parpadee con un intervalo de un segundo y poner esta tarea en la prioridad más alta. Además de la tarea LED, existe una tarea más creada por el kernel, se conoce como *idle task* (tarea inactiva). Ésta se crea cuando no hay alguna tarea disponible para su ejecución. Esta tarea siempre se ejecuta con la prioridad más baja, es decir, prioridad 0.

Al analizar la Fig. 1 se observa que la ejecución comienza con la tarea LED y se ejecuta durante un tiempo específico, luego, durante el tiempo restante, la tarea inactiva se ejecuta hasta que se produce una *tick interrupt*. Luego, el kernel decide qué tarea debe ejecutarse de acuerdo con la prioridad de la tarea y el tiempo total transcurrido de la tarea LED. Cuando se completa 1 segundo, el kernel vuelve a elegir la tarea LED para ejecutar porque tiene una prioridad más alta que la tarea inactiva. También se puede decir que la tarea LED se adelanta a la tarea inactiva. Si hay más de dos tareas con la misma prioridad, se ejecutan *round robin*(por turnos) durante un tiempo específico.

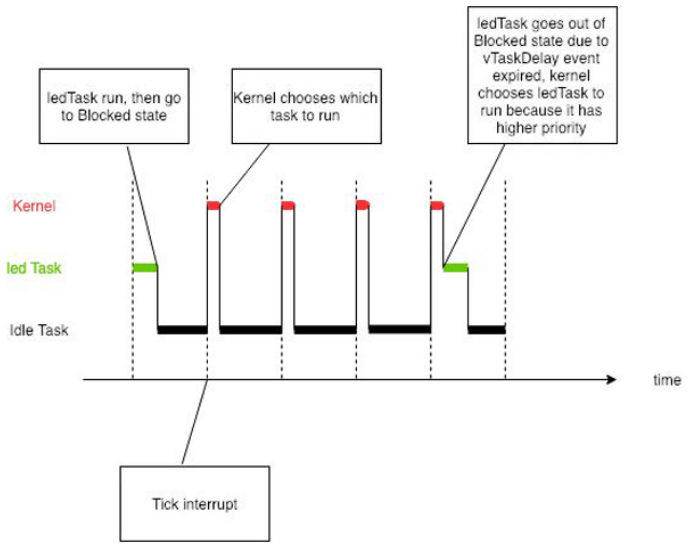

Figura 1: Utilización del procesador por el kernel y por una tarea

3.2. Estado de ejecución de una tarea

El diagrama de estados de la Fig. 2 muestra el cambio de una tarea desde el estado *not running* al estado *running* (en ejecución). Cada tarea recién creada pasa al estado *ready* (listo). Si una tarea creada (Task 1) tiene prioridad más alta que otras tareas, pasará al estado de ejecución. Si al ponerse en estado de ejecución

reemplaza a otra tarea que estaba ejecutándose (*preemption*), esta última volverá al estado listo.

Por otro lado, si Task 1 es bloqueada mediante el uso de la API de bloqueo (*blocking API*), la CPU no se involucrará con esta tarea hasta el tiempo de espera definido por el usuario.

Si Task 1 se suspende desde el estado de ejecución utilizando la API de suspensión (*suspend API*), entonces pasará al estado *suspended* (suspendido) y no estará disponible para el planificador nuevamente. Si Task 1 se reanuda (*resume*) desde el estado suspendido, volverá al estado listo. Esta es la idea básica de cómo se ejecutan las tareas y cómo cambian sus estados.

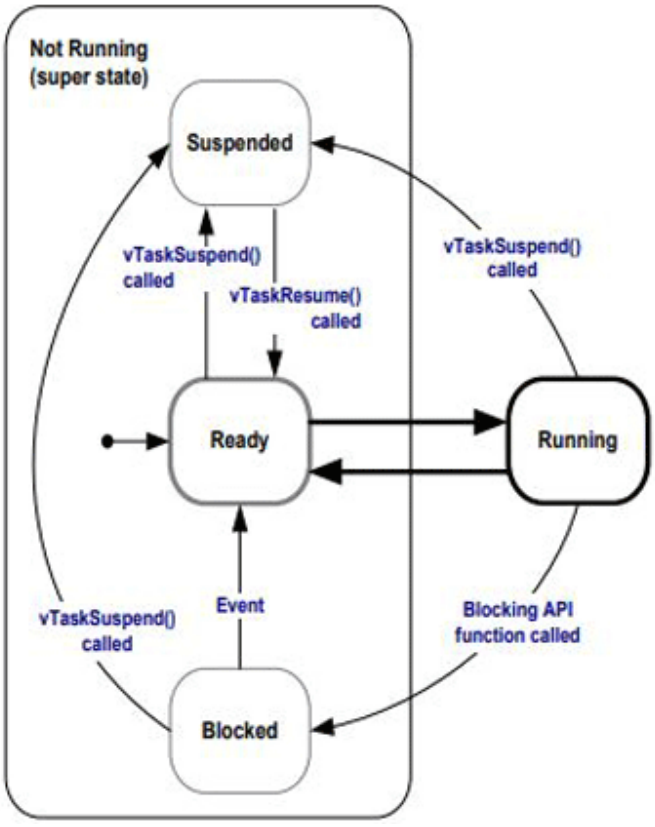

Figura 2: Diagrama de ejecución de una tarea en FreeRTOS

4. Términos de uso frecuente en RTOS

Tomado de `https://circuitdigest.com/microcontroller-projects/arduino-freertos-tutorial1-creating-freertos-task-to-blink-led-in-arduino-uno`

1. *Task:* es un fragmento de código que se puede planificar en la CPU para ser ejecutada.

2. *Scheduler:* es responsable de seleccionar una tarea de la lista de estado *ready* y ponerla en el estado (*running*). Los planificadores a menudo se implementan

para mantener ocupados todos los recursos de la computadora (como en el equilibrio de carga).

3. *Preemption:* es el acto de interrumpir temporalmente una tarea que ya se está ejecutando con la intención de sacarla del estado de ejecución sin su cooperación.

4. *Context Switching:* en *preemption* basado en prioridad, el planificador compara la prioridad de las tareas en *running* con la prioridad de la lista de tareas en *ready* en cada interrupción de *sysstick*. Si hay alguna tarea en la lista cuya prioridad es mayor que la tarea en ejecución, se produce un *context switch* (cambio de contexto). Básicamente, en este proceso los contenidos de diferentes tareas se guardan en su respectiva memoria de pila.

5. *Types of Scheduling policies:*

   - *Preemptive Scheduling:* las tareas se ejecutan con el mismo intervalo de tiempo sin tener en cuenta las prioridades.
   - *Priority-based Preemptive:* la tarea de alta prioridad se ejecuta primero.
   - *Co-operative Scheduling:* el cambio de contexto ocurre sólo con la cooperación de las tareas en ejecución. La tarea se ejecuta continuamente hasta que se llame al *task yield*.

6. *Kernel Objects:* para señalizar a una tarea que debe realizar algún trabajo se utiliza el proceso de sincronización. Para realizar este proceso se utilizan objetos del Kernel. Algunos objetos del Kernel son eventos (*events), semáforos (*semaphores), colas (*queues), mutex, buzones (*mailboxes), etc.

De la discusión anterior, se rescatan algunas ideas básicas sobre el concepto de RTOS y ahora se implementar una aplicación sencilla con FreeRTOS en Arduino.

5. Ejemplo

- Conecte el Arduino Uno al computador utilizando el cable USB, abra el entorno de desarrollo de Arduino e instale la biblioteca *FreeRTOS* en el entorno de Arduino.
- Cree un nuevo sketch en el entorno de Arduino, incluya las librerías necesarias y defina los pines a utilizar.
- En el método *setup()*, cree las tareas utilizando la función *xTaskCreate()*.
- Inicie el planificador de tareas llamando a *vTaskStartScheduler()*.
- Implemente el código de las tareas en las funciones correspondientes.
- Cargue el sketch al Arduino Uno y observe el comportamiento de las tareas.

```
// Incluye las librerías necesarias
#include <Arduino_FreeRTOS.h>
#include <task.h>

// Definir los pines a utilizar
#define LED_PIN 13

// Declarar las funciones de las tareas
void TaskBlink(void *pvParameters);
void TaskSerialPrint(void *pvParameters);

// Crear los objetos de las tareas
TaskHandle_t TaskBlinkHandle;
TaskHandle_t TaskSerialPrintHandle;

void setup() {
        // Inicializar el puerto serie
        Serial.begin(9600);

        // Crea las tareas
        xTaskCreate(
        TaskBlink,                  // Función de la tarea
        "Blink",                    // Nombre de la tarea
        128,                        // Tamaño de la pila (en palabras)
        NULL,                       // Parámetros de la tarea
        2,                          // Prioridad de la tarea
        &TaskBlinkHandle            // Manejador de la tarea
        );

        xTaskCreate(
        TaskSerialPrint,            // Función de la tarea
        "SerialPrint",              // Nombre de la tarea
        128,                        // Tamaño de la pila (en palabras)
        NULL,                       // Parámetros de la tarea
        1,                          // Prioridad de la tarea
        &TaskSerialPrintHandle      // Manejador de la tarea
        );

        // Inicia el planificador de tareas
        vTaskStartScheduler();
}

void loop() {
        // No se ejecutará nada aquí, ya que las tareas se ejecutan en
}
```

```
// Tarea que hace parpadear al LED cada segundo
void TaskBlink(void *pvParameters) {
        (void) pvParameters;

        pinMode(LED_PIN, OUTPUT);

        while (1) {
                digitalWrite(LED_PIN, HIGH);
                vTaskDelay(pdMS_TO_TICKS(500));

                digitalWrite(LED_PIN, LOW);
                vTaskDelay(pdMS_TO_TICKS(500));
        }
}

// Tarea que imprime un mensaje en el puerto serie cada dos segundos
void TaskSerialPrint(void *pvParameters) {
        (void) pvParameters;

        while (1) {
                Serial.println("¡Hola desde FreeRTOS en Arduino Uno!");
                vTaskDelay(pdMS_TO_TICKS(2000));
        }
}
```

6. Conclusiones

Se han implementado tareas en el sistema operativo de tiempo real FreeRTOS sobre Arduino Uno. El estudiante ha adquirido conocimientos sobre la creación y administración de tareas utilizando la biblioteca de FreeRTOS, así como sobre la coordinación de operaciones concurrentes en un entorno de tiempo real.

7. Trabajo a realizar

Se han implementado tareas independientes en FreeRTOS. Ahora, se necesita implementar un mecanismo de comunicación entre tareas utilizando colas.

1. Cree dos tareas adicionales, por ejemplo, *TaskSender* y *TaskReceiver*.

2. Defina una cola utilizando la función *xQueueCreate()* en el método *setup()*.

3. En la tarea *TaskSender*, envíe mensajes a la cola utilizando la función *xQueueSend()* con los datos que desee enviar.

4. En la tarea *TaskReceiver*, recibe los mensajes de la cola utilizando la función *xQueueReceive()* y muestre los datos recibidos en el puerto serie.

5. Asegúrese de asignar las prioridades adecuadas a las tareas y de utilizar adecuadamente la cola compartida para la comunicación entre tareas.

6. Escriba un breve informe dando las respuestas a los ítems anteriores. El trabajo se puede realizar en pareja (¡sólo 2 personas, no más!)