Aplicaciones de Sistemas Embebidos con Doble Núcleo

Desarrollo de Aplicaciones con FreeRTOS

UTN FRA
Departamento de Ingeniería Electrónica
Laboratorio de Sistemas Embebidos

20 de agosto de 2024

- Existen recursos para comunicar y sincronizar tareas (Queues, Semaphores, Mutex).
- Proporcionan medios para bloquear tareas.
- Pueden compartir datos de un contexto a otro.
- Tienen APIs diferenciadas para usar en tareas e interrupciones.

Las Queues o Colas permiten almacenar y compartir una cantidad de elementos del mismo tipo entre tareas.

- Se debe declarar primero la Queue.
- Para crearla, se usa xQueueCreate.
- Se indica cuantos elementos y tamaño.
- Los elementos deben ser del mismo tipo.
- Los elementos en la Queue se copian por valor.

```
// Declaracion de la queue
QueueHandle_t queue;

// Queue de 10 chars
queue = xQueueCreate(10, sizeof(char));
// Queue de 5 uint32_t
queue = xQueueCreate(5, sizeof(uint32_t));
// Queue de 1 uint16_t*
queue = xQueueCreate(1, sizeof(uint16_t*));

// Estructura especial
typedef struct {
    uint32_t foo;
    char baz[10];
    bool bar;
} custom_t;

// Queue de 1 custom_t
queue = xQueueCreate(1, sizeof(custom_t));
```

Las APIs xQueueSendToBack y xQueueSendToFront envían datos al final o comienzo de una cola respectivamente con la posibilidad de bloquear la tarea que la usa si la cola se encuentra llena y no es posible escribir. Tomando el caso de xQueueSendToBack tenemos:

```
xQueueSendToBack(
    queue,   // Queue para escribir
    &data,   // Puntero a variable con datos
    10       // Cantidad de ticks a bloquearse
);
```

Se puede usar la macro portMAX_DELAY si la tarea debe bloquearse indefinidamente hasta poder escribir.

La API xQueueOverwrite tiene un efecto similar a xQueueSendToBack pero directamente pisa el último valor de la Queue, sin bloquearse.

```
xQueueOverwrite(
    queue,      // Queue para escribir
    &data,      // Puntero a variable con datos
);
```

En este caso, no se usa el tercer parámetro para indicar ticks de bloqueo ya que esta API siempre sobreescribe el valor de la Queue. Se usa en Queues de un elemento.

La API de xQueueReceive sirve como contraparte de las APIs de xQueueSend. Permite leer la Queue indicada y, de no ser posible, bloquear la tarea. Una vez que se lee el dato, de quita de la Queue.

```
xQueueReceive(
    queue,      // Queue para leer
    &data,      // Puntero donde guardar dato
    100         // Cantidad de ticks a bloquearse
);
```

Similar al caso de xQueueSend, podemos usar el tercer parámetro con portMAX_DELAY para bloquear la tarea hasta que haya un dato disponible para leer.

Como la API de xQueueReceive, el caso de xQueuePeek lee la Queue pero no retira el dato de ella, solo lo copia en una nueva variable.

```
xQueuePeek(
    queue,      // Queue para leer
    &data,      // Puntero donde guardar dato
    100         // Cantidad de ticks a bloquearse
);
```

Si la Queue no tiene nada para leer, la tarea se va a bloquear la cantidad de ticks que se indiquen en el tercer parámetro. Puede usarse portMAX_DELAY para bloquear hasta que haya un dato para leer.

Podemos eliminar una Queue y liberar la memoria utilizada con la API vQueueDelete. Lo único necesario es el handle de la Queue.

```
// Se elimina la QUeue de la que se pasa el handle (QueueHandle_t)
vQueueDelete(queue);
```

Se puede, por otro lado, volver al valor inicial de la Queue reiniciándola con la API xQueueReset.

```
// Se resetea la Queue de la que se pasa el handle (QueueHandle_t)
xQueueReset(queue);
```

Existen versiones de las APIs anteriores que están pensadas para trabajar desde interrupciones (ISR), entre ellas: xQueueReceiveFromISR, xQueueSendToBackFromISR, xQueueSendToFrontFromISR, xQueueOverwriteFromISR y xQueuePeekFromISR.

```
// Variable para verificar cambio de contexto
BaseType_t task_woken = pdFALSE;

// Escribe en la Queue y verifica si es necesario el cambio de contexto
xQueueSendToBackFromISR(
    queue,       // Queue a la que escribir
    &data,       // Puntero de donde sacar el dato
    &task_woken // Cambio de contexto
);

// Si task_woken cambio a pdTRUE, se va a la tarea apropiada
portYIELD_FROM_ISR(task_woken);
```

Un Semaphore es una implementación de una Queue con dos estados posibles: Dado (Given) o Tomado (Taken).
Ocupan mucho menos RAM que una Queue y son útiles para sincronizar tareas. Se crea uno con xSemaphoreCreateBinary.

```
// Se crea la variable para el Semaphore
SemaphoreHandle_t semphr;

// Inicializacion del Semaphore
semphr = xSemaphoreCreateBinary();
```

Para eliminar un Semaphore, se usa la API vSemaphoreDelete.

```
// Elimino el Semaphore con el handle
vSemaphoreDelete(semphr);
```

Una tarea se bloquea cuando intenta hacer un xSemaphoreTake y no está disponible.
Se desbloquea cuando otra tarea hace un xSemaphoreGive.

```
// Tarea que da Semaphore
void task_give(void *params) {

    while(1) {
        // Da el Semaphore para desbloquear la tarea
        xSemaphoreGive(semphr);
        // Otras cosas...
    }
}
```

```
// Tarea que toma Semaphore
void task_take(void *params) {

    while(1) {
        // Toma el Semaphore cuando este disponible
        // se bloquea hasta que lo alguien lo de
        xSemaphoreTake(semphr);
        // Otras cosas...
    }
}
```

Una tarea que toma un Semaphore no es necesario que lo devuelva para que otra lo de.

Conmo con las Queues, existen APIs dedicadas para llamar desde interrupciones para los Semaphores.
Las alternativas para estas APIs son xSemaphoreGiveFromISR y xSemaphoreTakeFromISR.

```
// Variable para verificar cambio de contexto
BaseType_t task_woken = pdFALSE;

// Da el Semaphore y verifica si es necesario el cambio de contexto
xSemaphoreGiveFromISR(
    semphr,      // Semaphore que dar
    &task_woken // Cambio de contexto
);

// Si task_woken cambio a pdTRUE, se va a la tarea apropiada
portYIELD_FROM_ISR(task_woken);
```

Los Mutex son un tipo de Semaphore especial que se usan para exclusión mutua.
La API para crear un Mutex es xSemaphoreCreateMutex.

```
// Variable para el Mutex
SemaphoreHandle_t mutex;

// Inicializo el Mutex
mutex = xSemaphoreCreateMutex();
```

Las APIs son las mismas que para el caso de un Semaphore normal, pero una tarea que toma un Mutex, debe devolverlo para que otra tarea lo tome.

Este es un Semaphore especial que cuando una tarea lo da, incrementa una cuenta y cuando se toma, la cuenta baja.
Son especialmente útiles para contar eventos.

```
// Variable para el Semaphore Counting
SemaphoreHandle_t semphr;

// Creo el Semaphore Counting
semphr = xSemaphoreCreateCounting(
    100,            // Techo para contar
    0               // Valor inicial
);

// Obtengo el valor del Semaphore Counting
uint32_t count = uxSemaphoreGetCount(semphr);

// Reinicio el Semaphore Counting
xQueueReset(semphr);
```

Las APIs son las mismas que para manejar un Semaphore.

Algunas propuestas para practicar

1. En un proyecto llamado **05_display_controller**, hacer un programa que muestre del 00 al 99 en el display 7 segmentos. Se incrementa el número con S1, se decrementa con S2 y se resetea con USR.
2. En un proyecto llamado **05_proportional_control**, armar un programa que lea RV21 y RV22. En función de la diferencia, se enciende con mayor intensidad el D1.
3. En un proyecto llamado **05_frequency_counter** hacer un programa que cuente la frecuencia de una señal pulsante y la muestre por consola.

Cada ejercicio que se resuelva, subirlo al repositorio personal del curso.

Algunos recursos útiles

- Manual del LPC845
- Manual del LPC845 Breakout Board
- Documentación del SDK del LPC845
- Esquemático del kit del laboratorio
- RTOS Fundamentals - FreeRTOS
- The FreeRTOS Reference Manual
- Mastering the FreeRTOS Real Time Kernel
- Working with Queues in FreeRTOS
- FreeRTOS Queue Example
- FreeRTOS Semaphore Example
- FreeRTOS Mutex Example
- FreeRTOS Semaphore Counting Example