Aplicaciones de Sistemas Embebidos con Doble Núcleo

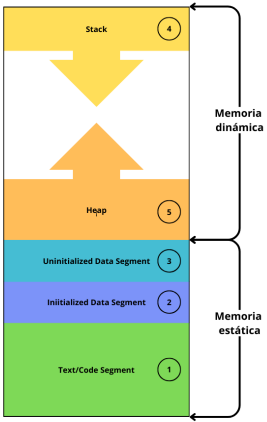
Introducción a FreeRTOS

UTN FRA
Departamento de Ingeniería Electrónica
Laboratorio de Sistemas Embebidos

16 de agosto de 2024

Índice

- Tareas y pseudo paralelismo de ejecución.
- Gestión fina de tiempo.
- Prioridades de eventos y tareas.
- Lógica más modular.
- Manejo dinámico de recursos.
- Porteado a múltiples platformas.

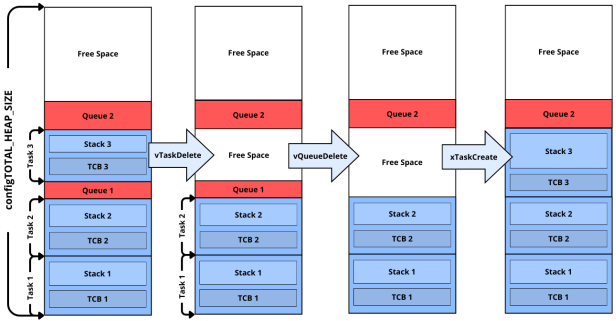


1. Text/Code Segment: intrucciones en Flash o EEPROM.
2. Initialized Data Segment: variables globales o estáticas inicializadas.
3. Uninitialized Data Segment: variables globales o estáticas no inicializadas o inicializadas con 0.
4. Stack: variables locales y registros del procesador con cada llamado de función.
5. Heap: RAM, variables inicializadas con *malloc()*.

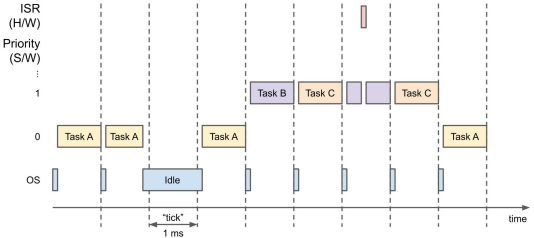
Opciones de Heap para FreeRTOS

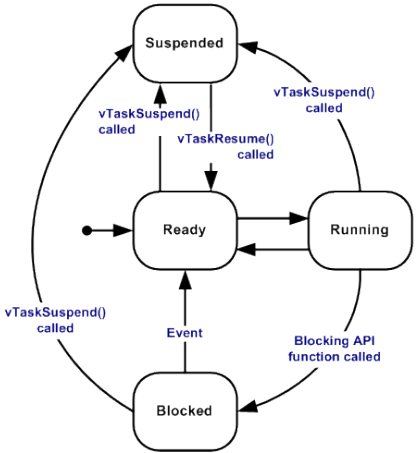
1. Heap 1: Crea pero no libera recursos.
2. Heap 2: Libera recursos, no es eficiente en la reasignación de recursos.
3. Heap 3: Versión más segura y más estándar de *malloc()* y *free*.
4. Heap 4: Implementación más eficiente de Heap 2.
5. Heap 5: Soporta Heap fragmentada en bancos.

- Implementa *pvPortMalloc()* y *vPortFree()*.
- Bloques contiguos de memoria liberada hacen se pueden combinar.
- Util en aplicaciones con reasignación de recursos permanente.
- Menos propenso a la fragmentación de memoria.



- El scheduler corre la tarea de mas alta prioridad en estado ready.
- Si nada la bloquea, una tarea corre por un tick del RTOS.
- Las interrupciones por hardware superan en prioridad a las tareas.
- Tareas de misma prioridad en estado ready se alternan.
- Si no hay tareas que correr, siempre está la Idle.





- Son funciones que nunca terminan ni retornan.
- Tienen posibles estados de *ready*, *running*, *suspended* y *blocked*.
- Tienen stack propio.
- Tienen prioridades asociadas.
- Se pueden crear y eliminar dinámicamente.

La API xTaskCreate se encarga de reservar y registrar los recursos necesarios para correr una tarea.

1. *pvTaskCode* es la función que se va a usar de tarea.
2. *pcName* es el nombre de la tarea en el debugger.
3. *usStackDepth* es la cantidad de words del stack de la tarea.
4. *pvParameters* es un puntero a datos que se puedan pasar a la tarea a través de **void** *params.
5. *uxPriority* es la prioridad de la tarea.
6. *pxCreatedTask* es el handle de la tarea.

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    configSTACK_DEPTH_TYPE usStackDepth,
    void * pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * pxCreatedTask
);
```

```c
#include "task.h"

// API para crear una tarea
xTaskCreate(
    task_fn,      // Funcion a usar
    "Task",       // Nombre para debug
    64,           // Stack size
    NULL,         // Parametros
    1,            // Prioridad
    NULL          // Sin handle
);

// En algun lado del programa

/**
 * @brief Tarea de ejemplo
 */
void task_fn(void *params) {

    while(1) {
        // Nunca termina
    }
}
```

- *task_fn* para a ser la función que trabaja de tarea.
- En el debugger, vamos a encontrarla con el nombre "Task".
- Se asignaron 64 words (256 bytes) de stack.
- No se pasan parámetros a la tarea (NULL).
- La prioridad es 1 (por encima de IDLE).
- No hay handle para la tarea.

- Se pueden pasar parámetros a las tareas casteando el dato como puntero a *void*.
- Esto es útil para poder crear varias tareas con la misma función.
- El parámetro debe ser casteado al tipo correcto dentro de la tarea.

```c
/**
 * @brief Estructura para un LED
 */
typedef struct {
    uint32_t port;
    uint32_t pin;
} led_t;
```

```c
/**
 * @brief Tarea template
 */
void task_template(void *params) {
    // Casteo al tipo de dato
    led_t led = *(led_t*) params;

    while(1) {
        GPIO_PortToggle(GPIO, led.port, 1 << led.pin);
    }
}
```

Esta posibilidad hace ofrece una gran ventaja de que se puedan crear tareas más eficientes a partir de una misma función.

```
// Estructura para una tarea
led_t led_1 = { 1, 0 };

// Creo tarea del mismo template
xTaskCreate(
    task_template,      // Misma tarea de template
    "LED-1",
    configMINIMAL_STACK_SIZE,
    (void*) &led_1,     // Cambia el dato enviado
    tskIDLE_PRIORITY + 1,
    NULL
);
```

```
// Estructura para una tarea
led_t led_2 = { 1, 2 };

// Creo tarea del mismo template
xTaskCreate(
    task_template,      // Misma tarea de template
    "LED-2",
    configMINIMAL_STACK_SIZE,
    (void*) &led_2,     // Cambia el dato enviado
    tskIDLE_PRIORITY + 1,
    NULL
);
```

- Las variables TaskHandle_t guardan una referencia a una tarea.
- Algunas APIs de FreeRTOS usan estas referencias para cambiar propiedades de la tarea. Entre ellas:
  - vTaskDelete() que elimina una tarea.
  - uxTaskPriorityGet() que obtiene la prioridad de una tarea.
  - vTaskPrioritySet() que setea un nuevo valor de prioridad.
  - vTaskResume() que reanuda una tarea suspendida.
  - vTaskSuspend() que suspende una tarea.

```
// Handle para tarea
TaskHandle_t handle;

// API para crear una tarea
xTaskCreate(
    task_fn,     // Funcion a usar
    "Task",      // Nombre para debug
    128,         // Stack size
    NULL,        // Parametros
    1,           // Prioridad
    &handle      // Referencia a handle
);

// Cambio de prioridad a 2
vTaskPrioritySet(handle, 2);
// Elimino tarea
vTaskDelete(handle);
```

Si configTICK_RATE_HZ vale 1000, el tick es de 1ms y coinciden los ms con los ticks.

```
// Bloqueo por 500 ticks (500 ms)
vTaskDelay(500);
```

Al usar el vTaskDelayUntil() hay que obtener la cantidad de ticks del RTOS.

```
// Obtengo la cantidad de ticks actuales
TickType_t xLastWakeTime = xTaskGetTickCount();
vTaskDelayUntill(&xLastWakeTime, 250)
```

Si configTICK_RATE_HZ no es 1000, podemos obtener los ticks para una cantidad de ms con una macro:

```
// Ticks para 500 ms
uint32_t ticks = pdMS_TO_TICKS(500);
```

- vTaskDelay() bloquea la tarea por una determinada cantidad de ticks.
- vTaskDelayUntil() bloquea la tarea por una cantidad específica de ticks.
- vTaskDelay() bloquea relativo a cuando se llama la API, vTaskDelayUntil() bloquea la tarea por esa cantidad de ticks desde que la tarea se corre.

Algunas propuestas para practicar

1. En un proyecto llamado **04_reloj**, hacer un programa que cuente de 00 a 59 en el display 7 segmentos cada 1 segundo. Al llegar a 60 debe volver a empezar en 00.
2. En un proyecto llamado **04_alarma**, armar un sistema de dos tareas donde, al presionar un botón, active una tarea que haga sonar el buzzer. Con otro botón, se activa una tarea que lo apaga.

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
- RTOS Task Scheduling and Prioritization
- FreeRTOS Memory Management