



Gestión de memoria en FreeRTOS.

Agenda.

- Uso de la memoria en programas en C (repaso)
- Uso de memoria en FreeRTOS
- Modelo de Memoria dinámica y heap en FreeRTOS.
- Creación de tareas y recursos estáticos.
- Ejemplos.

Uso de memoria

cada tarea tiene su propia pila

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int x[4];


int main(void)
{
    int a,b;
    int *ptr;
    ptr = (int*)malloc(64*sizeof(int));
    return 0;
}
```

- ¿Cuántos bytes de la pila usa el programa de la izquierda?
- ¿Cuántos del heap?
¿Quién lo gestiona?
- ¿Dónde y cuándo se almacenaran las variables globales?

Uso de memoria en FreeRTOS

```
#define configCPU_CLOCK_HZ          ( SystemCoreClock )
#define configTICK_RATE_HZ          ((TickType_t)1000)
#define configMAX_PRIORITIES        ( 3 )
#define configMINIMAL_STACK_SIZE    ((uint16_t)128)
#define configTOTAL_HEAP_SIZE        ((size_t)3072)
#define configMAX_TASK_NAME_LEN     ( 16 )
#define configUSE_TRACE_FACILITY     1
```

```
xTaskCreate(tarea_led,
            "led",
            configMINIMAL_STACK_SIZE,
            NULL,
            tskIDLE_PRIORITY+1,
            NULL);
```



toda la memoria que va a necesitar para la tarea la pide al heap.

- Cuando se crea una tarea, se define el tamaño de la pila. Esto limita la cantidad de anidamientos de llamadas a función y la cantidad de **variables locales**.
- Se puede utilizar memoria del heap con **pvPortMalloc()** para tomar y **pvPortFree()** para liberar memoria.
- Las variables globales se van a reservar en **tiempo de compilación**.
- El código de la derecha se conoce como **asignación dinámica** en FreeRTOS ya que es el sistema operativo el que gestiona la asignación de memoria de la tarea a crear.

Uso de memoria en FreeRTOS (2)

- `xTaskCreate()` todas estas funciones le piden memoria al heap a través de `pvPortMalloc()`.
 - `xQueueCreate()`
 - `xTimerCreate()`
 - `xEventGroupCreate()`
 - `xSemaphoreCreateBinary()`
 - `xSemaphoreCreateCounting()`
 - `xSemaphoreCreateMutex()`
 - `xSemaphoreCreateRecursiveMutex()`
- Las API de la derecha son las que “consumen” memoria en FreeRTOS.
 - Cada vez que se las llama recurren a la función **`pvPortMalloc()`** para guardar datos en memoria (por ejemplo: TCB y pila en `xTaskCreate`).
 - El comportamiento de como se utiliza la memoria dinámica (heap) depende ***modelo de memoria*** utilizado en la configuración de FreeRTOS

```
/* Allocate the memory for the heap. */
#if( configAPPLICATION_ALLOCATED_HEAP == 1 )
    /* The application writer has already defined the array used for the RTOS
    heap - probably so it can be placed in a special segment or address. */
    extern uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
#else
    static uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
#endif /* configAPPLICATION_ALLOCATED_HEAP */
```

heap_4.c

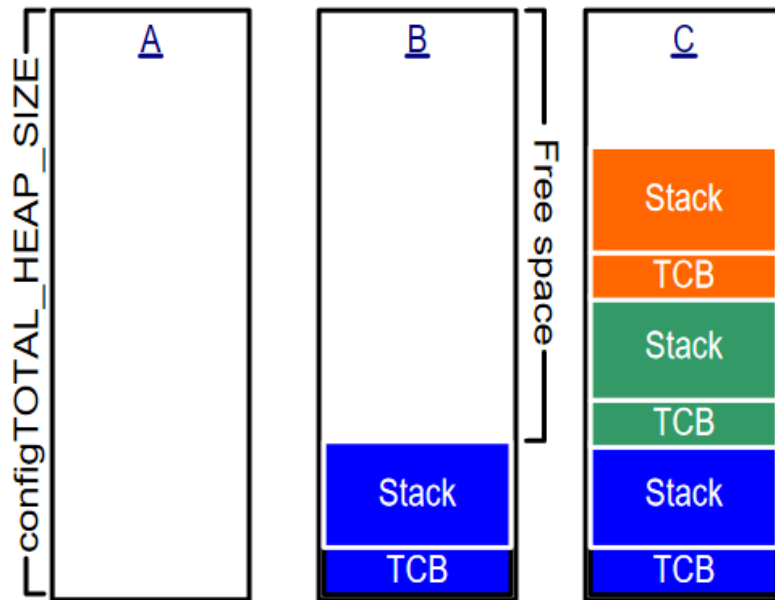
Modelos de memoria dinámica

- FreeRTOS (o cualquier SO) necesita memoria para mantener sus recursos (tarefas, semáforos, colas, etc.)
 - FreeRTOS puede asignar memoria dinámicamente, pero no usa malloc() y free() por defecto por los siguientes motivos:
 - No todos los sistemas las implementan.
 - Suelen ser pesadas (mucha memoria de código)
 - No están pensadas para el uso concurrente
 - No son determinísticas (el tiempo entre llamada y llamada para mismos parámetros es diferente)
 - Por esos motivos FreeRTOS implementa sus propios esquemas de memoria dinámica (sólo puede usarse un esquema por proyecto).
- Esquemas de gestión de memoria:
 - **heap_1.** Permite tomar memoria pero no liberarla
 - **heap_2.** Permite liberar memoria pero no une bloques libres contiguos.
 - **heap_3.** Utiliza malloc() y free() y agrega código para poder usarlas de manera concurrente.
 - **heap_4.** Utiliza el algoritmo del primer ajuste y puede unir bloques de memoria adyacente que se libera. Es que se utiliza por defecto en el STM32CubeIDE. **el primer lugar donde entre , lo uso.**
 - **heap_5.** Es similar al heap_4 pero puede utilizar memoria no contigua para ubicar el heap.

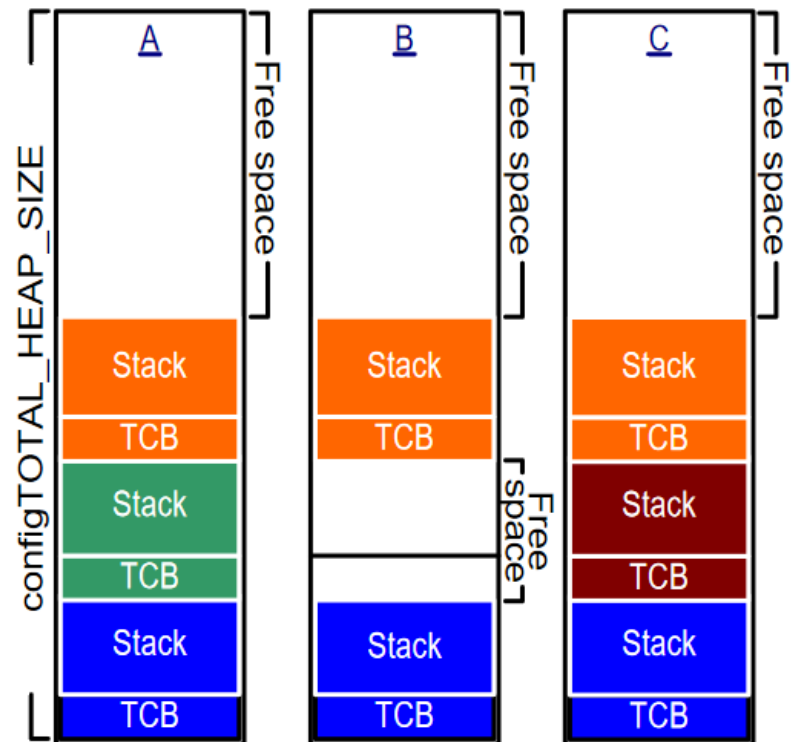
es mas rapido y es bastante bueno (es el que se usa)

Ejemplos heap_1 y heap_2

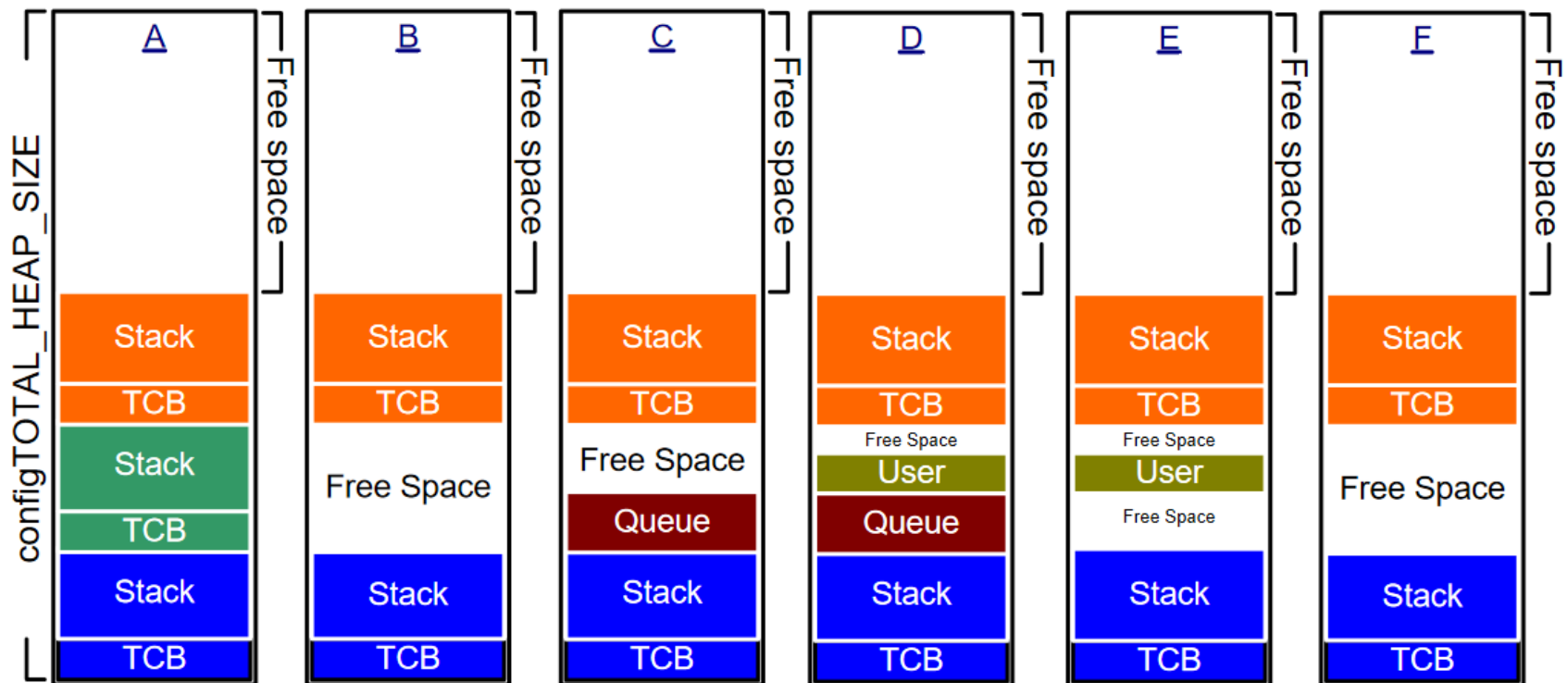
heap_1



heap_2

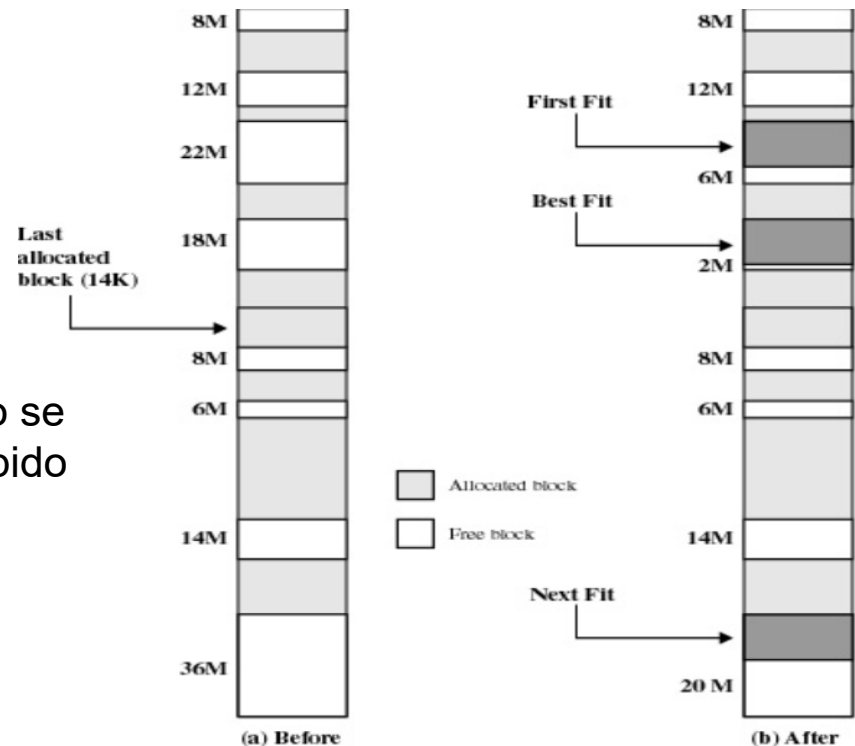


Ejemplo heap_4



Algoritmo del primer ajuste (first fit algorithm)

- Los tres algoritmos más utilizados para tomar memoria para una tarea/procesos.
 - **Algoritmo del primer ajuste.** El primer espacio lo suficientemente grande se usa.
 - **Algoritmo del mejor ajuste.** Se usa el espacio en memoria más parecido al necesitado. es el mas lento
 - **Algoritmo del siguiente ajuste.** Se busca el espacio necesario luego del último utilizado. es el mas rapido pero se acaba la memoria rapido
- El algoritmo normalmente más utilizado es el del primer ajuste porque suele ser más rápido y más eficiente (el mejor ajuste suele dejar bloques de memoria muy pequeños sin utilizar).



Liberando memoria

- En caso de utilizar el esquema de memoria heap_4 o heap_5 es necesario liberar la memoria.
- La responsabilidad de liberar la memoria es de la tarea idle().
- La tarea ociosa llama a la función **vPortFree** para liberar memoria.
- El esquema de memoria heap_4 o heap_5 es útil cuando se crean y destruyen tareas, si las tareas y los recursos de sincronización se crean cuando inicia el sistema y no cambian son más útiles esquemas como el heap_1, heap_2 o asignación estática.

en un sistema embebido generalmente no se eliminan tareas. Por ende nosotros no lo usaremos.

```
void vPortFree( void *pv )
{
    uint8_t *puc = ( uint8_t * ) pv;
    BlockLink_t *pxLink;
    if( pv != NULL )
    {
        /* The memory being freed will have an BlockLink_t structure immediately
        before it. */
        puc -= xHeapStructSize;

        /* This casting is to keep the compiler from issuing warnings. */
        pxLink = ( void * ) puc;

        /* Check the block is actually allocated. */
        configASSERT( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 );
        configASSERT( pxLink->pxNextFreeBlock == NULL );

        if( ( pxLink->xBlockSize & xBlockAllocatedBit ) != 0 )
        {
            if( pxLink->pxNextFreeBlock == NULL )
            {
                /* The block is being returned to the heap - it is no longer
                allocated. */
                pxLink->xBlockSize &= ~xBlockAllocatedBit;

                vTaskSuspendAll();
                {
                    /* Add this block to the list of free blocks. */
                    xFreeBytesRemaining += pxLink->xBlockSize;
                    traceFREE( pv, pxLink->xBlockSize );
                    prvInsertBlockIntoFreeList( ( ( BlockLink_t * ) pxLink ) );
                }
                ( void ) xTaskResumeAll();
            }
            else
            {
                mtCOVERAGE_TEST_MARKER();
            }
        }
        else
        {
            mtCOVERAGE_TEST_MARKER();
        }
    }
}
```

Juan Alarcón.
jalarcon@electron.frba.utn.edu.ar

Probando esquemas de memorias

- Tomando el ejemplo del led parpadeante para FreeRTOS y cambiando los esquemas de asignación de memoria se recompiló el mismo programa.

```
make -j4 all
arm-none-eabi-size  ledfreertos.elf
   text    data    bss     dec     hex filename
  15864     24    8104   23992   5db8 ledfreertos.elf
Finished building: default.size.stdout
```

11:09:21 Build Finished. 0 errors, 0 warnings. (took 501ms)

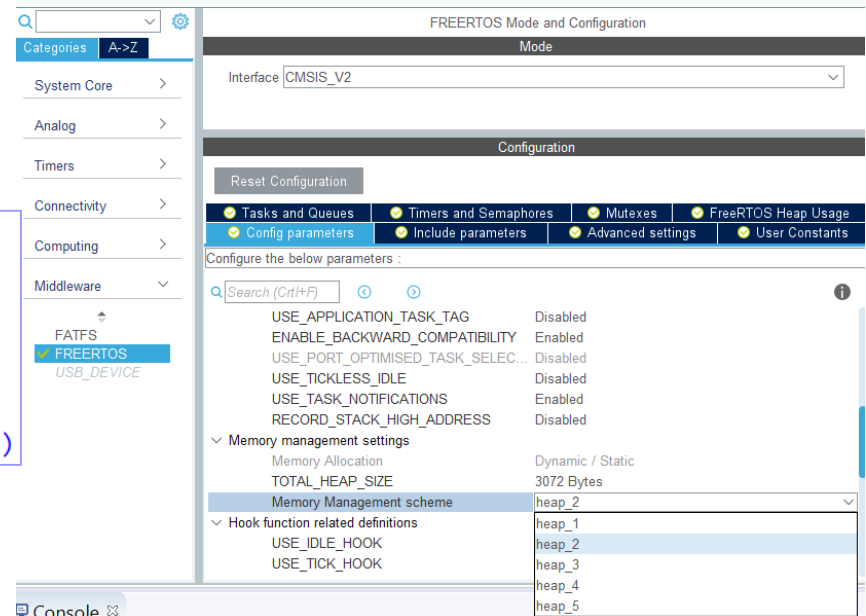
heap_1



```
make -j4 all
arm-none-eabi-size  ledfreertos.elf
   text    data    bss     dec     hex filename
  15728     28    8108   23864   5d38 ledfreertos.elf
Finished building: default.size.stdout
```

11:07:27 Build Finished. 0 errors, 0 warnings. (took 470ms)

heap_2



```
make -j4 all
arm-none-eabi-size  ledfreertos.elf
   text    data    bss     dec     hex filename
  16616     24    8120   24760   60b8 ledfreertos.elf
Finished building: default.size.stdout
```

11:10:35 Build Finished. 0 errors, 0 warnings. (took 439ms)

heap_4

Asignación de memoria estática en FreeRTOS

- `xTaskCreateStatic()`
- `xQueueCreateStatic()`
- `xTimerCreateStatic()`
- `xEventGroupCreateStatic()`
- `xSemaphoreCreateBinaryStatic()`
- `xSemaphoreCreateCountingStatic()`
- `xSemaphoreCreateMutexStatic()`
- `xSemaphoreCreateRecursiveMutexStatic()`

■ Estas API necesitan que se les pasen los mismos parámetros que sus versiones dinámicas más los buffers necesarios para que se pueda gestionar el recurso correspondiente.

```
NOTE: This is the number of words the stack will hold, not the number of
bytes. For example, if each stack item is 32-bits, and this is set to 100,
then 400 bytes (100 * 32-bits) will be allocated. */
#define STACK_SIZE 200

/* Structure that will hold the TCB of the task being created. */
StaticTask_t xTaskBuffer;

/* Buffer that the task being created will use as its stack. Note this is
an array of StackType_t variables. The size of StackType_t is dependent on
the RTOS port. */
StackType_t xStack[ STACK_SIZE ];

/* Function that implements the task being created. */
void vTaskCode( void * pvParameters )
{
    /* The parameter value is expected to be 1 as 1 is passed in the
    pvParameters value in the call to xTaskCreateStatic(). */
    configASSERT( ( uint32_t ) pvParameters == 1UL );

    for( ;; )
    {
        /* Task code goes here. */
    }
}

/* Function that creates a task. */
void vOtherFunction( void )
{
    TaskHandle_t xHandle = NULL;

    /* Create the task without using any dynamic memory allocation. */
    xHandle = xTaskCreateStatic(
        vTaskCode,          /* Function that implements the task. */
        "NAME",             /* Text name for the task. */
        STACK_SIZE,        /* Number of indexes in the xStack array. */
        ( void * ) 1,       /* Parameter passed into the task. */
        tskIDLE_PRIORITY,   /* Priority at which the task is created. */
        xStack,             /* Array to use as the task's stack. */
        &xTaskBuffer );     /* Variable to hold the task's data structure. */

    /* puxStackBuffer and pxTaskBuffer were not NULL, so the task will have
    been created, and xHandle will be the task's handle. Use the handle
    to suspend the task. */
    vTaskSuspend( xHandle );
}
```

Juan Alarcón.

jalarcon@electron.frba.utn.edu.ar

Ejemplo. Tarea y semáforo estáticos

```
/* USER CODE BEGIN PV */
    StackType_t xStack[STACK_LEN];
    StaticTask_t xTaskBuffer;
    StaticSemaphore_t xSemToggleLedBuffer;
    xSemaphoreHandle semToggleLed;
/* USER CODE END PV */

xTaskCreateStatic(TareaLed,
    "led_estatico",
    STACK_LEN,
    NULL,
    tskIDLE_PRIORITY+1,
    xStack,
    &xTaskBuffer);

semToggleLed = xSemaphoreCreateBinaryStatic(&xSemToggleLedBuffer);
```

```
make -j4 all
arm-none-eabi-size  semIntFreeRTOS_estatico.elf
  text    data    bss     dec     hex filename
 17848     24   5776   23648    5c60 semIntFreeRTOS_estatico.elf
Finished building: default.size.stdout
```

```
15:41:59 Build Finished. 0 errors, 0 warnings. (took 469ms)
```

Bibliografía.

- Mastering the FreeRTOS™ Real Time Kernel. Richard Barry.
https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf
- FreeRTOS <http://www.freertos.org/>