# FreeRTOS - Dynamic Memory Management

Creating RTOS objects dynamically has the benefit of greater simplicity, and the potential to minimize the application's maximum RAM usage. FreeRTOS offers several heap management schemes to manage memory allocation in different application types.

#arm #stm32 #rtos #memory

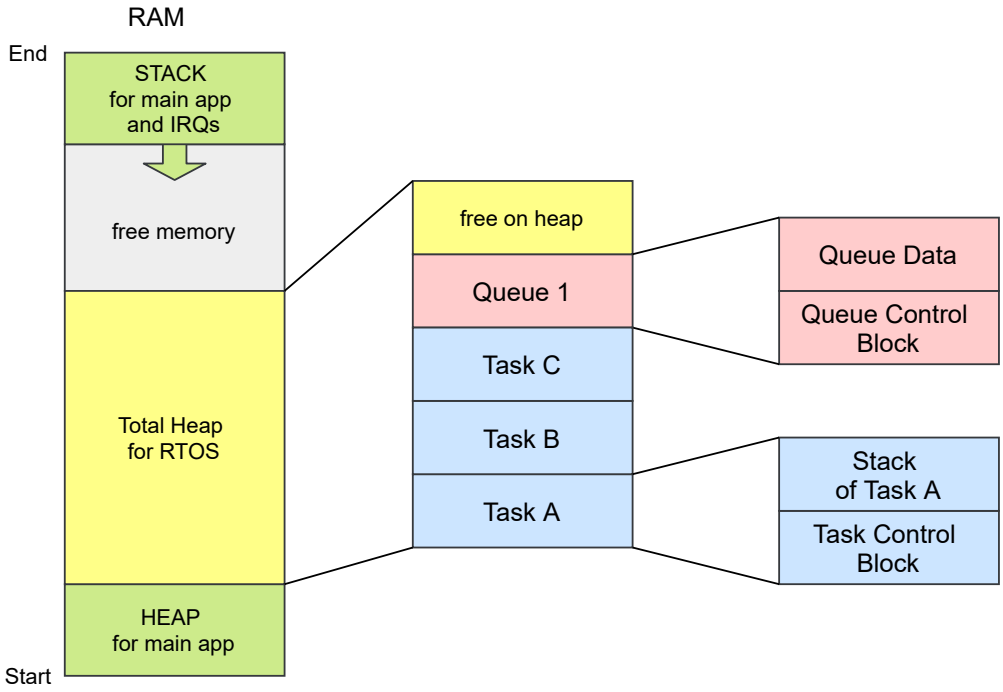
---

Last update: 2021-08-10 16:50:49

# Table of Content

# 1. Dynamic Memory Management

FreeRTOS uses a region of memory called Heap (into the RAM) to allocate memory for tasks, queues, timers, semaphores, mutexes and when dynamically creating variables. FreeRTOS heap is different from the system heap defined at the compiler level.




*Memory Heap in RTOS*

When FreeRTOS requires RAM, instead of calling the standard `malloc()`, it calls `PvPortMalloc()`. When it needs to free memory it calls `PvPortFree()` instead of the standard `free()`.

FreeRTOS offers several heap management schemes that range in complexity and features. The FreeRTOS download includes five sample memory allocation implementations, each of which are described in the following subsections. The subsections also include information on when each of the provided implementations might be the most appropriate to select.

Heap management schemes:

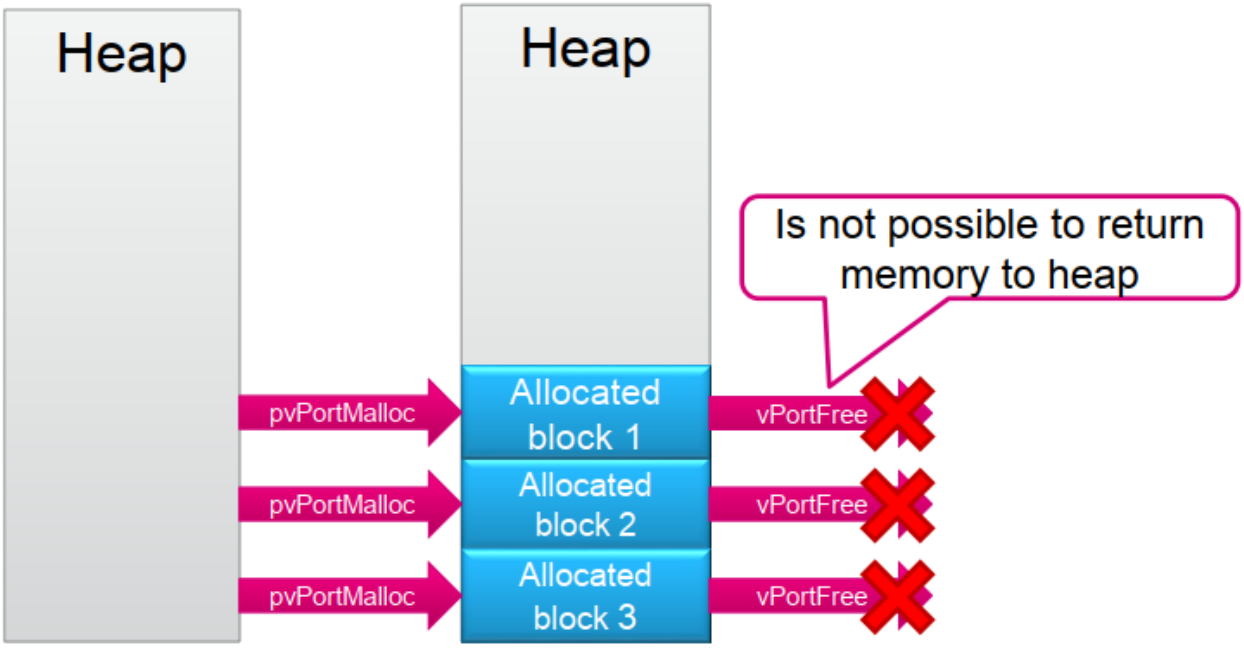
- heap_1 — the very simplest, does not permit memory to be freed.
- heap_2 — permits memory to be freed, but does not coalescence adjacent free blocks.
- heap_3 — simply wraps the standard `malloc()` and free() for thread safety.
- heap_4 — coalescence adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
- heap_5 — as per **heap_4**, with the ability to span the heap across multiple non-adjacent memory areas.

Notes:

- **heap_1** is less useful since FreeRTOS added support for static allocation.
- **heap_2** is now considered legacy as the newer **heap_4** implementation is preferred.

## 1.1. Heap_1

- This implementation uses *first fit algorithm* to allocate memory.
- It is the simplest allocation method (deterministic), but does not allow freeing of allocated memory.
- This could be interesting when no memory freeing is necessary.

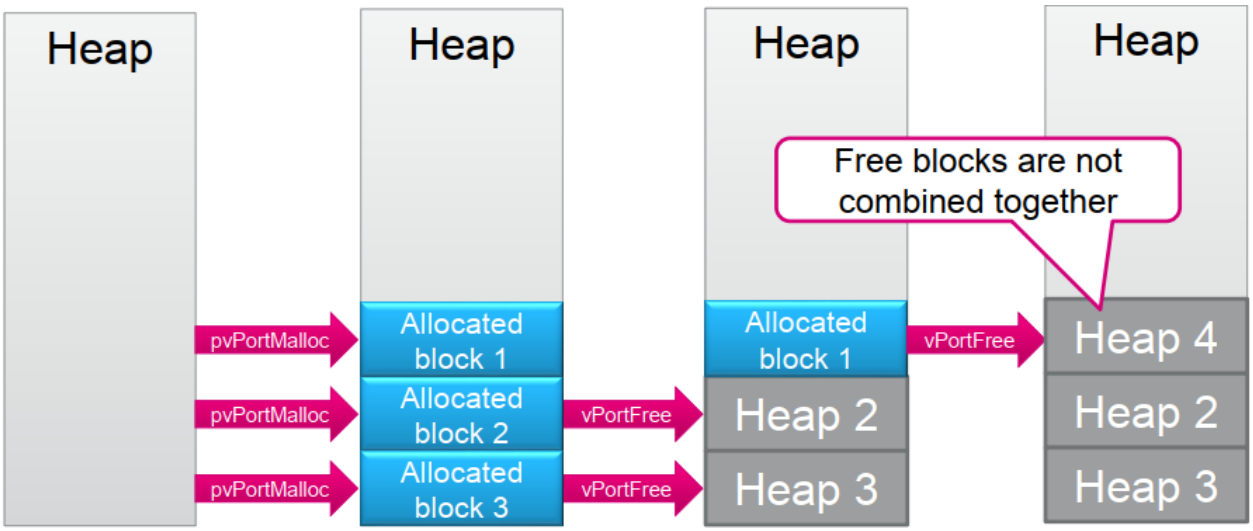



*Heap_1 method*

## 1.2. Heap_2

⚠ This implementation is not recommended to new projects. It's kept due to backward compatibility.

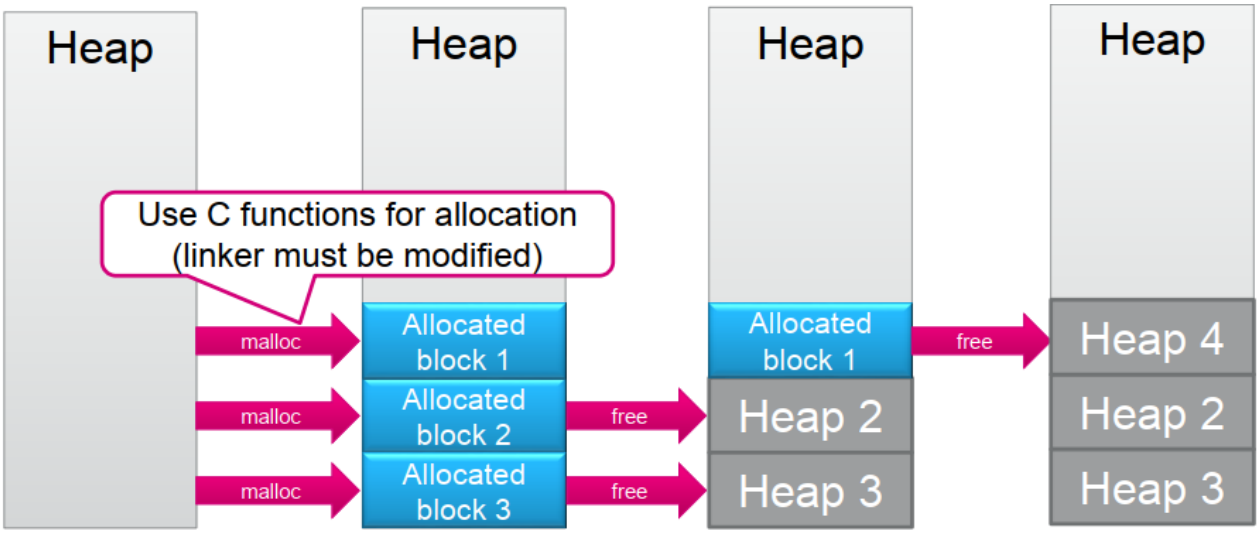
- This method implements the *best fit algorithm* for allocation.
- It allows memory `free()` operation but doesn't combine adjacent free blocks.
- This method has risk of fragmentation.




*Heap_2 method*

## 1.3. Heap_3

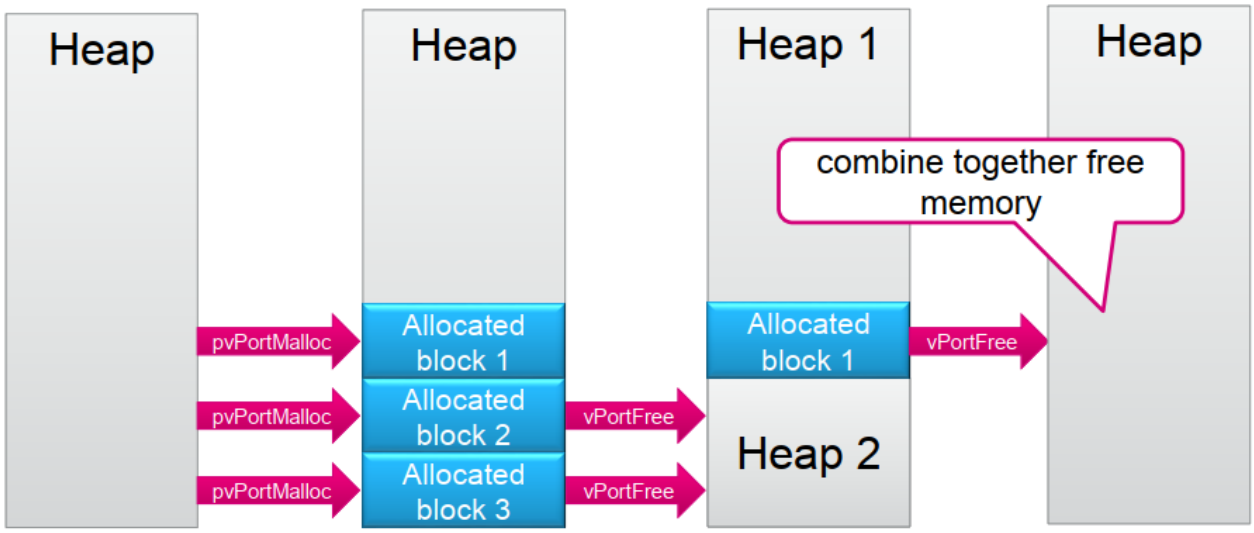
- This method implements simple wrapper for standard C library `malloc()` and `free()`; wrapper makes these functions thread safe, but makes code increase and not deterministic
- It uses linker heap region.
- The `configTOTAL_HEAP_SIZE` setting has no effect when this model is used




*Heap_3 method*

## 1.4. Heap_4

- This method uses *first fit algorithm* to allocate memory. It is able to combine adjacent free memory blocks into a single block.




*Heap_4 method*

- The heap is organized as a linked list: for better efficiency when dynamically allocating/Freeing memory. As consequence when allocating N bytes in the heap memory using `pvPortMalloc()` API it consumes:
  - Size of `BlockLink_t` (structure of the heap linked list) : 8 bytes.
  - Data to be allocated itself : N bytes.
  - Add padding to total allocated size (N + 8) to be 8 bytes aligned
- The memory array used by `heap_4` is declared within `heap_4.c` file and its start address is configured by the linker automatically.
  - To manually set the memory array address:
    - Set `configAPPLICATION_ALLOCATED_HEAP` to **1**
    - Declared a memory array: `uint8_t ucHeap[configTOTAL_HEAP_SIZE]`

## 1.5. Heap_5

- The Fit algorithm in this method is able to combine adjacent free memory blocks into a single block using the same algorithms as in `heap_4` , but supporting different memory regions (i.e. SRAM1, SRAM2) being not in linear memory space
- It is the only memory allocation scheme that must be explicitly initialized before any OS object can be created (before first call of `pvPortMalloc()` ).
  - Application specifies start address and size of each separate memory area.
  - Lower address appears in the array first
  - To initialize this scheme, `vPortDefineHeapRegions()` function should be called.

> An example for STM32L476 device with SRAM1 and SRAM2 areas:

```
#define SRAM1_OS_START (uint8_t *)0x2000 1000
#define SRAM1_OS_SIZE 0x0800 //2kB
#define SRAM2_OS_START (uint8_t *)0x1000 0000
#define SRAM2_OS_SIZE 0x1000 //4kB

/* Define */
Const HeapRegion_t xHeapRegions[] =
{
    {SRAM2_OS_START, SRAM2_OS_SIZE},
    {SRAM1_OS_START, SRAM1_OS_SIZE},
    {NULL, 0} /*terminates the array*/
}

/* Initialize */
vPortDefineHeapRegions(HeapRegions);
```

## 2. Manual allocation

There is an option to use alternative functions for memory management, however it is not recommended (inefficient) way of operation:

```
void StartTask1(void const * argument) {
    /* USER CODE BEGIN 5 */
    osPoolDef(Memory,0x100,uint8_t);
    PoolHandle = osPoolCreate(osPool(Memory));
    uint8_t* buffer=osPoolAlloc(PoolHandle);
    /* Infinite loop */
    for(;;) {
        osDelay(5000);
    }
    /* USER CODE END 5 */
}
```