Heap Memory Management

Dynamic Memory Allocation and its Relevance to FreeRTOS

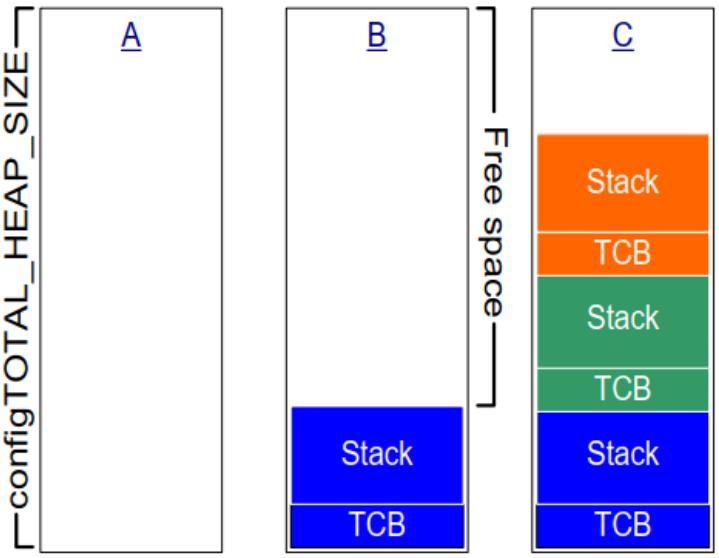
- From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time To make FreeRTOS as easy to use as possible.
- Objects are not statically allocated at compile-time, but dynamically allocated at run-timen time
- FreeRTOS allocates RAM each time a kernel object is created, and frees RAM each time a kernel object is deleted
- Memory can be allocated using the standard C library malloc() and free() functions, but they may not be suitable, or appropriate, for one or more of the following reasons
    - They are not always available on small embedded systems.
    - Their implementation can be relatively large, taking up valuable code space.
    - They are rarely thread-safe.
    - They are not deterministic
    - They can complicate the linker configuration.
    - They can be the source of difficult to debug errors

Options for Dynamic Memory Allocation

- From FreeRTOS V9.0.0 kernel objects can be allocated statically at compile time, or dynamically at run time
- FreeRTOS now treats memory allocation as part of the portable layer
- When FreeRTOS requires RAM, instead of calling malloc(), it calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the kernel calls vPortFree()
- pvPortMalloc() has the same prototype as the standard C library malloc() function, and vPortFree() has the same prototype as the standard C library free() function
- pvPortMalloc() and vPortFree() are public functions, so can also be called from application code
- FreeRTOS comes with five example implementations of both pvPortMalloc() and vPortFree(), Heap_1.c to Heap_5.c all of which are located in the FreeRTOS/Source/portable/MemMang directory.
- FreeRTOS applications can use one of the example implementations, or provide their own

Heap_1.c

- Heap_1.c implements a very basic version of pvPortMalloc(), and does not implement vPortFree(). *Applications that never delete a task*, or other kernel object, have the potential to use heap_1.
- The heap_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made. The array is called the FreeRTOS heap.
  The total size (in bytes) of the array is set by the definition configTOTAL_HEAP_SIZE within FreeRTOSConfig.h.
- Each created task requires a task control block (TCB) and a stack to be allocated from the heap



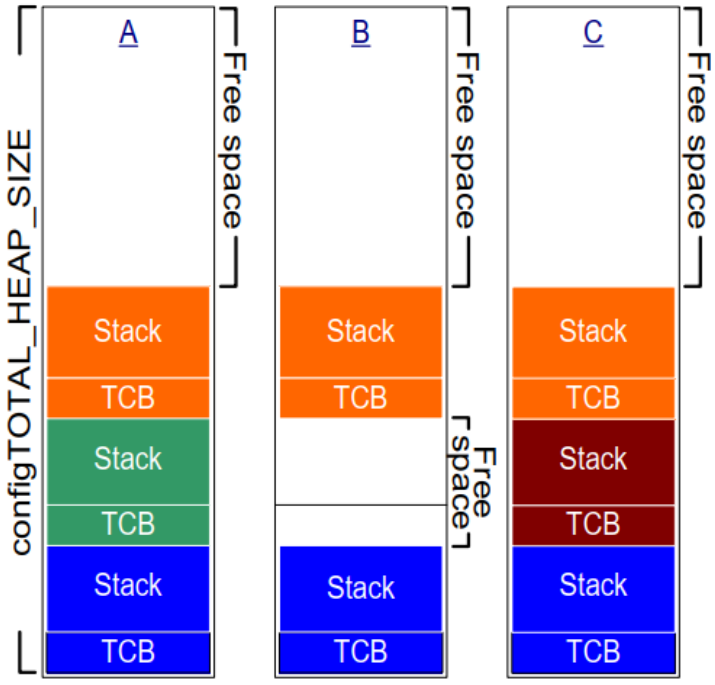
RAM being allocated from the heap_1 array each time a task is created

- Each created task requires a task control block (TCB) and a stack to be allocated from the heap.
- A shows the array before any tasks have been created the entire array is free.
- B shows the array after one task has been created.
- C shows the array after three tasks have been created

Heap_2.c

- Heap_2 is retained in the FreeRTOS distribution *for backward compatibility*, but its use is not recommended for new designs.
- Consider using heap_4 instead of heap_2, *as heap_4 provides enhanced functionality*

- Heap_2.c also works by subdividing an array that is dimensioned by configTOTAL_HEAP_SIZE
- It uses a ***best fit algorithm*** to allocate memory and, unlike heap_1, it does allow memory to be freed
- *The array is statically declared, so will make the application appear to consume a lot of RAM, even before any memory from the array has been assigned.*
- The best fit algorithm ensures that pvPortMalloc() uses the free block of memory that is *closest in size to the number of bytes requested*.
- Unlike heap_4, Heap_2 does not combine adjacent free blocks into a single larger block, so it is more susceptible to fragmentation. However, fragmentation is not an issue if the blocks being allocated and subsequently freed are always the same size
- Heap_2 is suitable for an application that creates and deletes tasks repeatedly,



RAM being allocated and freed from the heap_2 array as tasks are created and deleted

- A shows the array after three tasks have been created. A large free block remains at the top of the array
- B shows the array after one of the tasks has been deleted. The large free block at the top of the array remains. There are now also two smaller free blocks that were previously allocated to the TCB and stack of the deleted task

- C shows the situation after another task has been created. Creating the task has resulted in two calls to pvPortMalloc(), one to allocate a new TCB, and one to allocate the task stack.
- Every TCB is exactly the same size, so the best fit algorithm ensures that the block of RAM previously allocated to the TCB of the deleted task is reused to allocate the TCB of the new task.
- The larger unallocated block at the top of the array remains untouched.
- Heap_2 is not deterministic, but is faster than most standard library implementations of malloc() and free().

Heap_3.c

- Heap_3.c uses the standard library malloc() and free() functions, so the size of the heap is defined by the linker configuration, and the configTOTAL_HEAP_SIZE setting has no affect
- Heap_3 makes malloc() and free() thread-safe by temporarily suspending the FreeRTOS scheduler.

Heap_4.c

- Heap_4 uses a ***first fit*** algorithm to allocate memory.
- Like heap_1 and heap_2, heap_4 works by subdividing an array into smaller blocks.
- The first fit algorithm ensures pvPortMalloc() uses the first free block of memory that is large enough to hold the number of bytes requested
- Heap_4 combines (coalescences) adjacent free blocks into a single larger block, minimizing the risk of fragmentation, and making it suitable for applications that repeatedly allocate and free different sized blocks of RAM.
- Heap_4 is not deterministic, but is faster than most standard library implementations of malloc() and free().

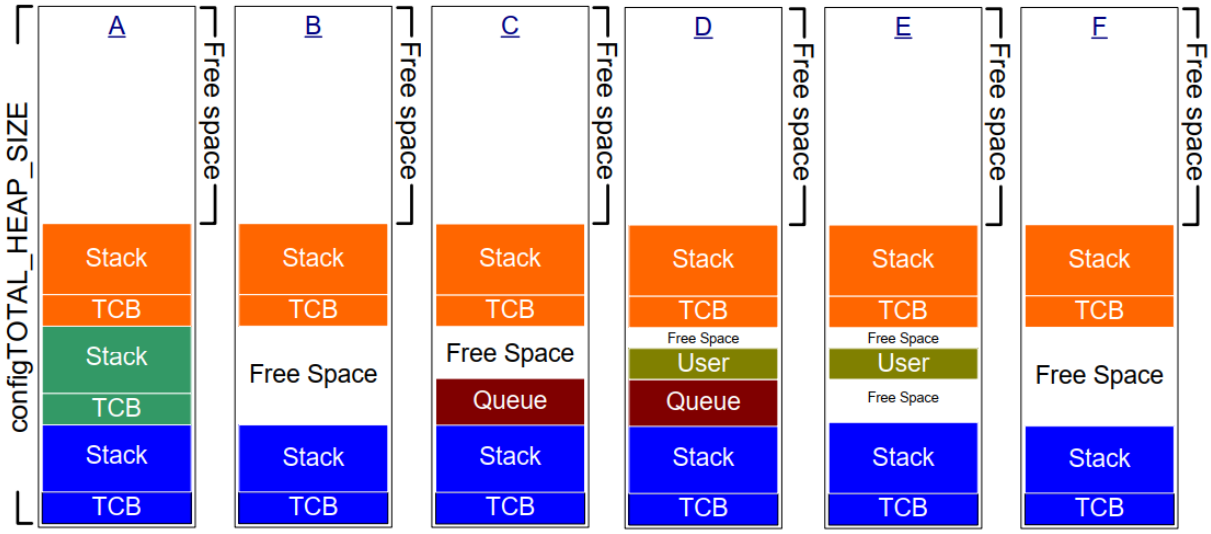


Figure 7. RAM being allocated and freed from the heap_4 array

Setting a Start Address for the Array Used By Heap_4

- Sometimes it is necessary for an application writer to place the array used by heap_4 at a specific memory address.
- For example, the stack used by a FreeRTOS task is allocated from the heap, so it might be necessary to ensure the heap is located in fast internal memory, rather than slow external memory.
- By default, the array used by heap_4 is declared inside the heap_4.c source file, and its start *address is set automatically by the linker.*
- if the configAPPLICATION_ALLOCATED_HEAP compile time configuration constant is set to 1 in FreeRTOSConfig.h, *then the array must instead be declared by the application that is using FreeRTOS*. If the array is declared as part of the application, then the application's writer can set its start address
- If configAPPLICATION_ALLOCATED_HEAP is set to 1 in FreeRTOSConfig.h, then a uint8_t array called ucHeap, and dimensioned by the configTOTAL_HEAP_SIZE setting, must be declared in one of the application's source files.
- the syntax required by the GCC compiler to declare the array, and place the array in a memory section called .my_heap

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( ( section( ".my_heap" ) ) );
```

Heap_5.c

- The algorithm used by heap_5 to allocate and free memory is identical to that used by heap_4
- Unlike heap_4, heap_5 is not limited to allocating memory from a single statically declared array; heap_5 can allocate memory from multiple and separated memory spaces.
- vPortDefineHeapRegions() is used to specify the start address and size of each separate memory ar ea that together makes up the total memory used by heap_5.

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions );
```

Reference : Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide by Richard Barry