HEAP MANAGEMENT

INDEX:

1. Difference between stack and heap memory

**Difference between stack and heap memory**

Stack and heap are two types of memory used for different purposes in programming. Understanding the differences between them is crucial for managing memory efficiently in your applications. Here is a detailed comparison:

**Stack Memory**

1. **Memory Allocation**:
   * **Automatic**: Stack memory is allocated and deallocated automatically when functions are called and return, respectively.
   * **LIFO (Last In, First Out)**: Stack memory allocation follows a strict LIFO order.
2. **Size**:
   * **Limited**: The stack size is typically much smaller than the heap and is determined by the system or compiler settings. Exceeding the stack size limit causes a stack overflow.
3. **Access Speed**:
   * **Fast**: Access to stack memory is very fast because it operates on the LIFO principle, with simple push and pop operations.
4. **Storage**:
   * **Local Variables**: Stack memory stores local variables, function parameters, return addresses, and control flow data.
5. **Lifetime**:
   * **Short-Lived**: The lifetime of variables stored in the stack is limited to the duration of the function in which they are declared.
6. **Management**:
   * **Managed by Compiler**: The compiler manages stack memory, making it easier to use but less flexible.
7. **Thread Safety**:
   * **Thread-Local**: Each thread has its own stack, which means stack memory is inherently thread-safe.
8. **Example Usage**:

void function() {

int localVar = 10; // localVar is allocated on the stack

}

**Heap Memory**

1. **Memory Allocation**:
   * **Manual**: Heap memory is allocated and deallocated manually by the programmer using functions like **malloc**, **calloc**, **realloc**, and **free** in C, or **new** and **delete** in C++.
2. **Size**:
   * **Larger and Dynamic**: The heap is generally much larger than the stack and can grow dynamically as needed, limited by the system's memory.
3. **Access Speed**:
   * **Slower**: Access to heap memory is slower than stack memory because it involves more complex memory management and potential fragmentation.
4. **Storage**:
   * **Dynamically Allocated Variables**: Heap memory stores dynamically allocated variables and objects whose size and lifetime are not known at compile time.
5. **Lifetime**:
   * **Long-Lived**: The lifetime of variables in the heap is controlled by the programmer and can span beyond the function that created them.
6. **Management**:
   * **Programmer Managed**: The programmer is responsible for allocating and deallocating heap memory, which provides flexibility but also requires careful management to avoid memory leaks and fragmentation.
7. **Thread Safety**:
   * **Not Inherently Thread-Safe**: Heap memory is shared among threads, so concurrent access requires synchronization mechanisms to avoid race conditions.
8. **Example Usage**:

void function() {

int\* heapVar = (int\*)malloc(sizeof(int)); // heapVar is allocated on the heap

\*heapVar = 10;

free(heapVar); // Manually deallocating heap memory

}

**Summary**

| **Feature** | **Stack** | **Heap** |
| --- | --- | --- |
| **Allocation** | Automatic | Manual |
| **Size** | Limited | Larger, dynamic |
| **Access Speed** | Fast | Slower |
| **Storage** | Local variables | Dynamically allocated objects |
| **Lifetime** | Short-lived | Long-lived |
| **Management** | Managed by compiler | Managed by programmer |
| **Thread Safety** | Thread-local | Requires synchronization |
| **Example Usage** | **int localVar** | **int\* heapVar = malloc(...)** |

**Memory management**

The RTOS kernel needs RAM each time a task, queue, mutex, software timer, semaphore or event group is created. The RAM can be automatically dynamically allocated from the RTOS heap within the RTOS API object creation functions, or it can be [provided by the application writer](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html).

If RTOS objects are created dynamically then the standard C library malloc() and free() functions can sometimes be used for the purpose, but **...**

1. they are not always available on embedded systems,
2. they take up valuable code space,
3. they are not thread safe, and
4. they are not deterministic (the amount of time taken to execute the function will differ from call to call)

**...** so more often than not an alternative memory allocation implementation is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, allowing an application specific implementation appropriate for the real time system being developed to be provided. When the RTOS kernel requires RAM, instead of calling malloc(), it instead calls pvPortMalloc(). When RAM is being freed, instead of calling free(), the RTOS kernel calls vPortFree().

FreeRTOS offers several heap management schemes that range in complexity and features. It is also possible to provide your own heap implementation, and even to use two heap implementations simultaneously. Using two heap implementations simultaneously permits task stacks and other RTOS objects to be placed in fast internal RAM, and application data to be placed in slower external RAM.

**Static vs Dynamic memory allocation**

FreeRTOS V9.0.0 and onwards gives the application writer the ability to instead provide the memory themselves, allowing the following objects to optionally be created without any memory being allocated dynamically:

* Tasks
* Software Timers
* Queues
* Event Groups
* Binary Semaphores
* Counting Semaphores
* Recursive Semaphores
* Mutexes

Whether it is preferable to use static or dynamic memory allocation is dependent on the application, and the preference of the application writer. Both methods have pros and cons, and both methods can be used within the same RTOS application.

**Creating an RTOS Object Using Dynamically Allocated RAM (run time)**

Creating RTOS objects dynamically has the benefit of greater simplicity, and the potential to minimise the application's maximum RAM usage:

* Fewer function parameters are required when an object is created.
* The memory allocation occurs automatically, within the RTOS API functions.
* The application writer does not need to concern themselves with allocating memory themselves.
* The RAM used by an RTOS object can be re-used if the object is deleted, potentially reducing the application's maximum RAM footprint.
* The memory allocation scheme used can be chosen to best suite the application, be that heap\_1.c for simplicity and determinism often necessary for safety critical applications, heap\_4.c for fragmentation protection, heap\_5.c to split the heap across multiple RAM regions, or an allocation scheme provided by the application writer themselves.

The following API functions, which are available if **[configSUPPORT\_DYNAMIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_DYNAMIC_ALLOCATION)** is set to 1 or left undefined, create RTOS objects using dynamically allocated RAM:

* [xTaskCreate()](https://www.freertos.org/a00125.html)
* [xQueueCreate()](https://www.freertos.org/a00116.html)
* [xTimerCreate()](https://www.freertos.org/FreeRTOS-timers-xTimerCreate.html)
* [xEventGroupCreate()](https://www.freertos.org/xEventGroupCreate.html)
* [xSemaphoreCreateBinary()](https://www.freertos.org/xSemaphoreCreateBinary.html)
* [xSemaphoreCreateCounting()](https://www.freertos.org/CreateCounting.html)
* [xSemaphoreCreateMutex()](https://www.freertos.org/CreateMutex.html)
* [xSemaphoreCreateRecursiveMutex()](https://www.freertos.org/xSemaphoreCreateRecursiveMutex.html)

**Creating an RTOS Object Using Statically Allocated RAM (compile time)**

Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control:

* RTOS objects can be placed at specific memory locations.
* The maximum RAM footprint can be determined at link time, rather than run time.
* The application writer does not need to concern themselves with graceful handling of memory allocation failures.
* It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation

The following API functions, which are available if **[configSUPPORT\_STATIC\_ALLOCATION](https://www.freertos.org/a00110.html" \l "configSUPPORT_STATIC_ALLOCATION)** is set to 1, allow RTOS objects to be created using memory provided by the application writer. To provide memory the application writer simply needs to declare a variable of the appropriate object type, then pass the address of the variable into the RTOS API function. The [StaticAllocation.c](https://sourceforge.net/p/freertos/code/HEAD/tree/trunk/FreeRTOS/Demo/Common/Minimal/StaticAllocation.c" \t "_blank) standard demo/test task is provided to demonstrate how the functions are used:

* [xTaskCreateStatic()](https://www.freertos.org/xTaskCreateStatic.html)
* [xQueueCreateStatic()](https://www.freertos.org/xQueueCreateStatic.html)
* [xTimerCreateStatic()](https://www.freertos.org/xTimerCreateStatic.html)
* [xEventGroupCreateStatic()](https://www.freertos.org/xEventGroupCreateStatic.html)
* [xSemaphoreCreateBinaryStatic()](https://www.freertos.org/xSemaphoreCreateBinaryStatic.html)
* [xSemaphoreCreateCountingStatic()](https://www.freertos.org/xSemaphoreCreateCountingStatic.html)
* [xSemaphoreCreateMutexStatic()](https://www.freertos.org/xSemaphoreCreateMutexStatic.html)
* [xSemaphoreCreateRecursiveMutexStatic()](https://www.freertos.org/xSemaphoreCreateRecursiveMutexStatic.html)

**Heap memory management**

The RTOS kernel needs RAM each time a task, queue, mutex, software timer, semaphore or event group is created. The RAM can be automatically dynamically allocated from the RTOS heap within the RTOS API object creation functions, or it can be [provided by the application writer](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html).

If RTOS objects are created dynamically then the standard C library malloc() and free() functions can sometimes be used for the purpose, but **...**

1. they are not always available on embedded systems,
2. they take up valuable code space,
3. they are not thread safe, and
4. they are not deterministic (the amount of time taken to execute the function will differ from call to call)

**...** so more often than not an alternative memory allocation implementation is required.

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, allowing an application specific implementation appropriate for the real time system being developed to be provided. When the RTOS kernel requires RAM, instead of calling malloc(), it instead calls **pvPortMalloc().** When RAM is being freed, instead of calling free(), the RTOS kernel calls **vPortFree().**

FreeRTOS offers several heap management schemes that range in complexity and features. It is also possible to provide your own heap implementation, and even to use two heap implementations simultaneously. Using two heap implementations simultaneously permits task stacks and other RTOS objects to be placed in fast internal RAM, and application data to be placed in slower external RAM.

## **Memory allocation implementations included in the RTOS source code**

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.

Each provided implementation is contained in a separate source file (heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c and heap\_5.c respectively) which are located in the Source/Portable/MemMang directory of the main RTOS source code download. Other implementations can be added as needed. Exactly one of these source files should be included in a project at a time [the heap defined by these portable layer functions will be used by the RTOS kernel even if the application that is using the RTOS opts to use its own heap implementation].

Following below:

* [heap\_1](https://www.freertos.org/a00111.html#heap_1) - the very simplest, does not permit memory to be freed.
* [heap\_2](https://www.freertos.org/a00111.html#heap_2) - permits memory to be freed, but does not coalescence adjacent free blocks.
* [heap\_3](https://www.freertos.org/a00111.html#heap_3) - simply wraps the standard malloc() and free() for thread safety.
* [heap\_4](https://www.freertos.org/a00111.html#heap_4) - coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
* [heap\_5](https://www.freertos.org/a00111.html#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](https://www.freertos.org/Static_Vs_Dynamic_Memory_Allocation.html).
* heap\_2 is now considered legacy as the newer heap\_4 implementation is preferred.