HEAP MEMORY MANAGEMENT

INDEX:

1. Static vs Dynamic Allocation

Static allocation	Dynamic allocation
Memory allocates at compile time	Memory allocates at run time
The compiler determines the size and location of memory statically	Program requests memory using functions like malloc and receives a pointer
Memory remains reserved for entire duration of program execution	The allocated memory remains reserved until the program explicitly deallocates using functions like free,delete
Allocated before execution	Allocated during the execution of program
No memory allocation or free during execution	Memory allocation and free can be done during execution
Implements using stack/heap	Implements using Data segments
Faster	Slow
Uses more memory space	Uses less memory space

1.1 Persistent and Non-Persistent memory

Persistent memory : stores data even after the power is off . It is also called non volatile memory.

Example : pendrive,ROM

Non-persistent memory : if power is lost the memory stored is lost.It is also called volatile memory

Example : RAM

Advantages and disadvantages of static memory allocation

ADV:

1. RAM allocated while loading program, use whenever we want it

DIS:

1. cannot modify size

Advantages and disadvantages of dynamic memory allocation

ADV:

1. increase or decrease memory at size at run time

DIS:

1. slow execution due to allocation at run time
2. BSS not only used by dynamic memory but also other section of program

1.2 Memory Layout and sections

In computer programming, the memory layout of a program is often divided into three main segments: code, data, and BSS (Block Started by Symbol). These segments play crucial roles in how your program interacts with memory:

1. Code Segment:

- **Function:** Stores the actual machine code instructions that your program executes. This includes compiled code from your source code (e.g., C, Python) containing the logic and functionality of your program.
- **Properties:**
    - **Read-only:** The code segment is typically read-only as the instructions themselves shouldn't be modified during program execution.
    - **Shared:** In some cases, the code segment might be shared between multiple programs or libraries, improving memory efficiency.

2. Data Segment:

- **Function:** Stores initialized global and static variables declared in your program. These variables have a fixed value assigned during compilation or program initialization.
- **Properties:**
    - **Read-write:** The data segment is read-write as the values of these variables can be changed during program execution.
    - **Allocated memory:** The size of the data segment is determined by the number and sizes of the initialized variables your program defines.

3. BSS Segment (Block Started by Symbol):

- **Function:** Stores uninitialized global and static variables. These variables don't have an initial value explicitly assigned during compilation.
- **Properties:**
    - **Read-write:** Similar to the data segment, the BSS segment is read-write as the values of these variables can be modified during program execution.
    - **Allocated memory:** The size of the BSS segment is determined by the number and sizes of the uninitialized variables your program declares. However, unlike the data segment, the BSS segment is typically initialized with zeros by the operating system when the program loads into memory.

Segment	**Function**	**Properties**
Code	Stores machine code instructions	Read-only, Shared (potentially)
Data	Stores initialized global/static variables	Read-write, Allocated memory
BSS	Stores uninitialized global/static variables	Read-write, Allocated memory (initialized with zeros)

1.3 Difference between stack and heap memory

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(...)**

2. Memory allocation

2.1 Memory allocation in OS

- malloc(),calloc(),realloc(),free() these 4 functions for dynamic allocation in OS but does not support in FreeRTOS
- when we require 4 bytes for malloc() the linux kernel always allocates page wise memory . Therefore it allocates 4kB memory.
- since 1 page size = 4kB. Even though we need 1 byte it allocates 4kb memory

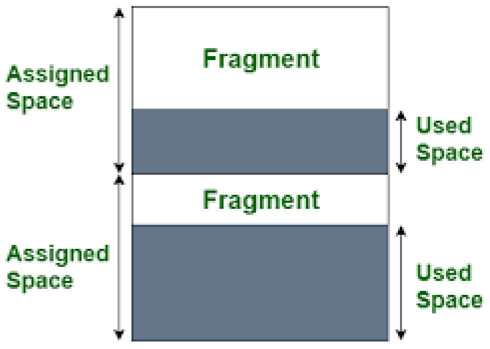
2.2 Memory allocation in FreeRTOS

- FreeRTOS kernel objects such as task,queue,semaphore, eventgroups can be allocated dynamically and freed immediately when not required.
- Task control block is a data structure contains all details of task
- In FreeRTOS dynamically allocation features:
  - Reduces design and planning effort

  - Minimizes the RAM footprint
  - Simplifies the API
- Memory assigned for each task:
  - TCB (fixed size)
  - Memory for stack (not fixed)

2.3 Why standard C library not used?

1. Not always available on small embedded system
2. Implementation is large, take up valuable code space
3. Rarely thread safe
4. Not deterministic (the amount of time taken to execute the functions will differ from call to call)
5. They can suffer from fragmentation (the freed blocks in the middle of RAM may not be used by the tasks requiring more space)



Internal Fragmentation

6. There is no MMU (memory management unit)
7. 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.
8. memory allocation is part of portable layer
9. FreeRTOS provides 5 memory allocation algorithms ,user can choose according to application which are located in Source/portable/MemMang folder
   - Heap_1.c
   - Heap_2.c
   - Heap_3.c
   - Heap_4.c
   - Heap_5.c
10. We can also have our own memory management algorithm

2.4 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** is set to 1 or left undefined, create RTOS objects using dynamically allocated RAM:

- xTaskCreate()
- xQueueCreate()
- xTimerCreate()
- xEventGroupCreate()
- xSemaphoreCreateBinary()
- xSemaphoreCreateCounting()
- xSemaphoreCreateMutex()
- xSemaphoreCreateRecursiveMutex()

2.5 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

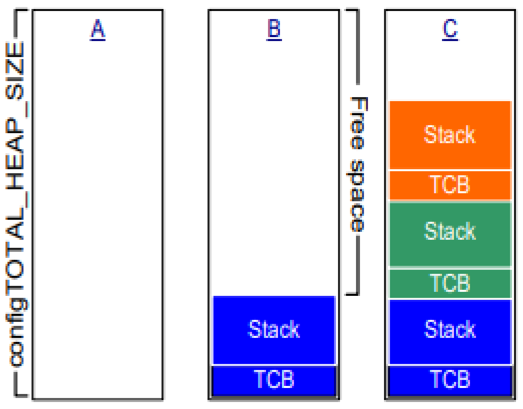
The following API functions, which are available if **configSUPPORT_STATIC_ALLOCATION** is set to 1, allow RTOS objects to be created using memory provided by the application writer. To provide memory the application writer simply needs to declare a variable of the appropriate object type, then pass the address of the variable into the RTOS API function. The StaticAllocation.c standard demo/test task is provided to demonstrate how the functions are used:

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

3. Memory Management Algorithms

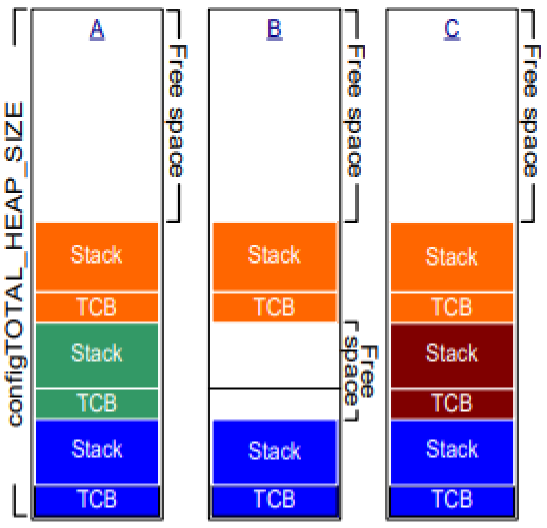
3.1 Heap_1

- It is common for small embedded systems to only create tasks and other kernel objects before the scheduler has been started.
- Memory only gets dynamically allocated by the kernel before the application starts to perform any real-time functionality, and the memory remains allocated for the lifetime of the application
- IT does not consider any of the more complex memory allocation issues, such as determinism and fragmentation, and can instead just consider attributes such as code size and simplicity
- 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
- Referring to below figure :
    - 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.



3.2 Heap_2

- 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, it allows memory to be freed
- Again, 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. For example, consider the scenario where:
    - The heap contains three blocks of free memory that are 5 bytes, 25 bytes, and 100 bytes, respectively.
    - pvPortMalloc() is called to request 20 bytes of RAM.
    - The smallest free block of RAM into which the requested number of bytes will fit is the 25-byte block, so pvPortMalloc() splits the 25-byte block into one block of 20 bytes and one block of 5 bytes, before returning a pointer to the 20-byte block. The new 5-byte block remains available to future calls to pvPortMalloc().
- 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, provided the size of the stack allocated to the created tasks does not change



- The above figure demonstrates how the best fit algorithm works when a task is created, deleted, and then created again.
    - 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. Tasks are created using the xTaskCreate() API function, which is. The calls to pvPortMalloc() occur internally within xTaskCreate().
    - 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 size of the stack allocated to the newly created task is identical to that allocated to the previously deleted task, so the best fit algorithm ensures that the block of RAM previously allocated to the stack of the deleted task is reused to allocate the stack 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().

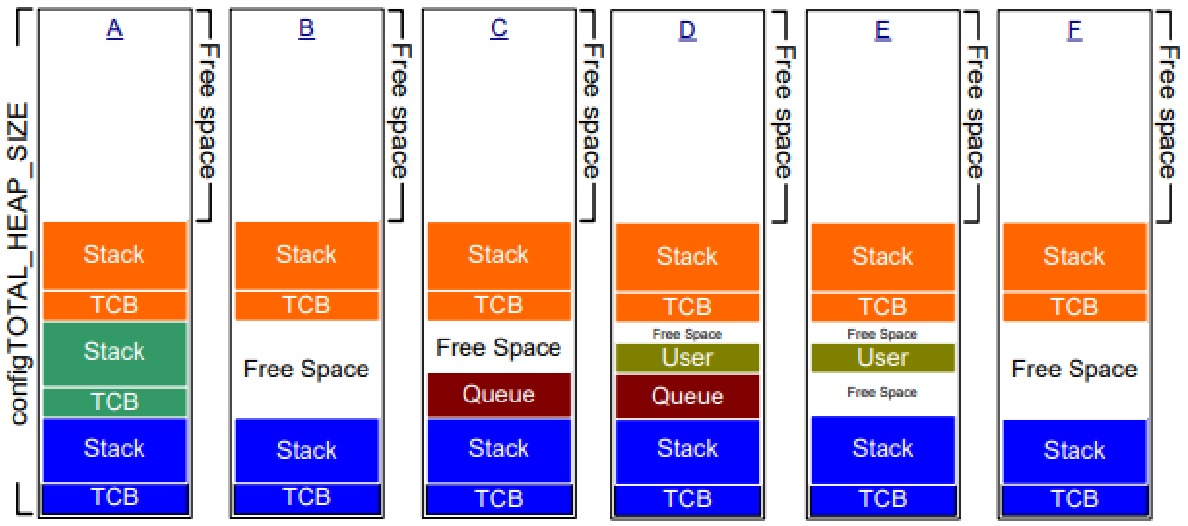
3.3 Heap_3

- Heap_3.c uses the standard library malloc() and free() functions, so the size of the heap is defined by the linker configuration(.ld), and the **configTOTAL_HEAP_SIZE** setting has no affect
- Linker file contains definitions of different memory locations where you can store the starting address. Configuring the linker script to allocate enough memory for the heap.
- Source code application and linker file together generate .bin file
- Heap_3 makes malloc() and free() thread-safe by temporarily suspending the FreeRTOS scheduler

3.4 Heap_4

- Like heap_1 and heap_2, heap_4 works by subdividing an array into smaller blocks
- As before, the array is statically declared, and dimensioned by configTOTAL_HEAP_SIZE, so will make the application appear to consume a lot of RAM, even before any memory has actually been allocated from the array
- As before, the array is statically declared, and dimensioned by configTOTAL_HEAP_SIZE, so will make the application appear to consume a lot of RAM, even before any memory has actually been allocated from the array
- The first fit algorithm ensures pvPortMalloc() uses the first free block of memory that is large enough to hold the number of bytes requested. For example, consider the scenario where:
    - The heap contains three blocks of free memory that, in the order in which they appear in the array, are 5 bytes, 200 bytes, and 100 bytes, respectively
    - pvPortMalloc() is called to request 20 bytes of RAM.
    - The first free block of RAM into which the requested number of bytes will fit is the 200-byte block, so pvPortMalloc() splits the 200-byte block into one block of 20 bytes, and one block of 180 bytes , before returning a pointer to the 20-byte block. The new 180-byte block remains available to future calls to pvPortMalloc().
- 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

- **EXAMPLE:**



i. A shows the array after three tasks have been created. A large free block remains at the top of the array.

ii. B shows the array after one of the tasks has been deleted. The large free block at the top of the array remains. There is also a free block where the TCB and stack of the 1 bytes. task that has been deleted were previously allocated. Note that, unlike when heap_2 was demonstrated, the memory freed when the TCB was deleted, and the memory freed when the stack was deleted, does not remain as two separate free blocks, but is instead combined to create a larger single free block.

iii. C shows the situation after a FreeRTOS queue has been created. Queues are created using the xQueueCreate() API function, which is described in section 4.3. xQueueCreate() calls pvPortMalloc() to allocate the RAM used by the queue. As heap_4 uses a first fit algorithm, pvPortMalloc() will allocate RAM from the first free RAM block that is large enough to hold the queue, which in Figure 7, was the RAM freed when the task was deleted. The queue does not consume all the RAM in the free block however, so the block is split into two, and the unused portion remains available to future calls to pvPortMalloc()

iv. D shows the situation after pvPortMalloc() has been called directly from application code, rather than indirectly by calling a FreeRTOS API function. The user allocated block was small enough to fit in the first free block, which was the block between the memory allocated to the queue, and the memory allocated to the following TCB.

    The memory freed when the task was deleted has now been split into three separate blocks; the first block holds the queue, the second block holds the user allocated memory, and the third block remains free.

v. E show the situation after the queue has been deleted, which automatically frees the memory that had been allocated to the deleted queue. There is now free memory on either side of the user allocated block.

vi. F shows the situation after the user allocated memory has also been freed. The memory that had been used by the user allocated block has been combined with the free memory on either side to create a larger single free block

- Heap_4 is not deterministic, but is faster than most standard library implementations of malloc() and free()

3.5 Heap_5

- The algorithm used by heap_5 to allocate and free memory is identical to that used by 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
- Heap_5 is useful when the RAM provided by the system on which FreeRTOS is running does not appear as a single contiguous (without space) block in the system's memory map
- At the time of writing, heap_5 is the only provided memory allocation scheme that must be explicitly initialized before pvPortMalloc() can be called.
- Heap_5 is initialized using the **vPortDefineHeapRegions()** API function.
- When heap_5 is used, vPortDefineHeapRegions() must be called before any kernel objects (tasks, queues, semaphores, etc.) can be created.

4. Heap related utility functions

4.1 vPortDefineHeapRegions()

vPortDefineHeapRegions() is used to specify the start address and size of each separate memory area that together makes up the total memory used by heap_5.

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

Each separate memory areas is described by a structure of type HeapRegion_t. A description of all the available memory areas is passed into vPortDefineHeapRegions() as an array of HeapRegion_t structures.

4.1.1 HeapRegion_t structure

```
typedef struct HeapRegion

{

        /* The start address of a block of memory that will be part of the heap.*/

        uint8_t *pucStartAddress;

        /* The size of the block of memory in bytes. */

         size_t xSizeInBytes;

} HeapRegion_t;
```

4.2 xPortGetFreeHeapSize()

- The xPortGetFreeHeapSize() API function returns the number of free bytes in the heap at the time the function is called. It can be used to optimize the heap size.
- For example, if xPortGetFreeHeapSize() returns 2000 after all the kernel objects have been created, then the value of configTOTAL_HEAP_SIZE can be reduced by 2000
- xPortGetFreeHeapSize() is not available when heap_3 is used.
- **Syntax**: size_t xPortGetFreeHeapSize(void);
- **Return :** The number of bytes that remain unallocated in the heap at the time xPortGetFreeHeapSize() is called

4.3 xPortGetMinimumEverFreeHeapSize()

- The xPortGetMinimumEverFreeHeapSize() API function returns the minimum number of unallocated bytes that have ever existed in the heap since the FreeRTOS application started executing
- The value returned by xPortGetMinimumEverFreeHeapSize() is an indication of how close the application has ever come to running out of heap space

- For example, if xPortGetMinimumEverFreeHeapSize() returns 200, then, at some time since the application started executing, it came within 200 bytes of running out of heap space.
- xPortGetMinimumEverFreeHeapSize() is only available when heap_4 or heap_5 is used.
- Syntax : size_t xPortGetMinimumEverFreeHeapSize(void);
- Return : The minimum number of unallocated bytes that have existed in the heap since the FreeRTOS application started executing.

4.4 vApplicationMallocFailedHook()

- pvPortMalloc() can be called directly from application code. It is also called within FreeRTOS source files each time an kernel object is created
- If pvPortMalloc() cannot return a block of RAM because a block of the requested size does not exist, then it will return NULL, then the kernel object will not be created.
- All the example heap allocation schemes can be configured to call a hook (or callback) function if a call to pvPortMalloc() returns NULL
- If configUSE_MALLOC_FAILED_HOOK is set to 1 in FreeRTOSConfig.h, then the application must provide a malloc failed hook function that has the name and prototype shown below

```
#define configUSE_MALLOC_FAILED_HOOK 1
#define configTOTAL_HEAP_SIZE ( ( size_t ) ( 10 * 1024 ) )
// Example: 10 KB heap

void vApplicationMallocFailedHook( void ) {
   /* Handle memory allocation failure */
   printf("Memory allocation failed!\n");
   taskDISABLE_INTERRUPTS();
   for (;;) {
      // Optionally log the error, reset the system, or take other actions
   }
}
```

4.5 vApplicationStackOverflowHook()

- The vApplicationStackOverflowHook function in FreeRTOS is a user-defined hook function that is called when a stack overflow is detected in a task. This function is very useful for debugging and ensuring the reliability of your embedded application.
- To enable stack overflow checking, you need to set the **configCHECK_FOR_STACK_OVERFLOW** macro in your FreeRTOS configuration file (FreeRTOSConfig.h).

```
void vApplicationStackOverflowHook(TaskHandle_t xTask, char *pcTaskName) {
    // Handle stack overflow here
    printf("Stack overflow detected in task %s\n", pcTaskName);
    taskDISABLE_INTERRUPTS();
    for(;;); // Halt the system
}
```

- The xTask and pcTaskName parameters pass to the hook function the handle and name of the offending task respectively. Note however, depending on the severity of the overflow, these parameters could themselves be corrupted, in which case the pxCurrentTCB variable can be inspected directly.

- **`pxCurrentTCB`** is a pointer used internally by FreeRTOS to keep track of the currently executing task. TCB stands for Task Control Block, which is a data structure used by FreeRTOS to store information about a task. Each task in FreeRTOS has its own TCB, and **`pxCurrentTCB`** points to the TCB of the task that is currently running.

4.5.1 Three levels of stack overflow

- o **Stack Overflow Detection - Method 1**
  - It is likely that the stack will reach its greatest (deepest) value after the RTOS kernel has swapped the task out of the Running state because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.

  - This method is quick but not guaranteed to catch all stack overflows. Set configCHECK_FOR_STACK_OVERFLOW to 1 to use this method.

- o **Stack Overflow Detection - Method 2**
  - When a task is first created its stack is filled with a known value. When swapping a task out of the Running state the RTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity. The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.

  - This method is less efficient than method one, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.

  - Set configCHECK_FOR_STACK_OVERFLOW to 2 to use this method.

- o **Stack Overflow Detection - Method 3**
  - Set configCHECK_FOR_STACK_OVERFLOW to 3 to use this method.
  - This method is available only for selected ports. When available, this method enables ISR stack checking.

5. Example codes

5.1 Use of vPortDefineHeapRegions() , vApplicationMallocFailedHook() , vApplicationStackOverflowHook()

```
#include "FreeRTOS.h"

#include "task.h"

#include "portable.h"

#include <stdio.h>


// Define the memory regions that will be used for the heap. These regions should be contiguous
blocks of memory.

static uint8_t ucHeap1[1024];

static uint8_t ucHeap2[2048];

static uint8_t ucHeap3[4096];


// Define an array of HeapRegion_t structures to specify the start addresses and sizes of the memory
regions.

HeapRegion_t xHeapRegions[] = {

    { ucHeap1, sizeof(ucHeap1) },  // Region 1

    { ucHeap2, sizeof(ucHeap2) },  // Region 2

    { ucHeap3, sizeof(ucHeap3) },  // Region 3

    { NULL, 0 }                    // Terminator

};


// Example task function

void vExampleTask(void *pvParameters) {

    for (;;) {

        // Task code goes here.

        vTaskDelay(pdMS_TO_TICKS(1000));  // Delay for 1000 ms

    }
}


int main(void) {
    // Configure the heap regions
    vPortDefineHeapRegions(xHeapRegions);


    // Create example tasks
    xTaskCreate(vExampleTask, "Task1", configMINIMAL_STACK_SIZE, NULL,
tskIDLE_PRIORITY + 1, NULL);


    // Start the scheduler
    vTaskStartScheduler();


    // The code should never reach here unless there's insufficient heap available to start the scheduler
    for (;;);
}


void vApplicationMallocFailedHook(void) {
    // Malloc failed hook implementation
    printf("Malloc failed!\n");
    for (;;);
}


void vApplicationStackOverflowHook(TaskHandle_t xTask, char *pcTaskName) {
    // Stack overflow hook implementation
    printf("Stack overflow in task: %s\n", pcTaskName);
    for (;;);
}
```

Output :

- If there is memory allocation failure it prints : **Malloc failed!**
- If there is stack overflow it returns : **Stack overflow in task: Task1**

5.2 Use of xPortGetFreeHeapSize() , xPortGetMinimumEverFreeHeapSize()

```
#include <stdio.h>
#include "stm32f4xx.h"
#include "FreeRTOS.h"
#include "task.h"
#include "heap_utils.h"

void vApplicationIdleHook(void)
{
}



// Task that prints the free heap size
void vFreeHeapSizeTask(void *pvParameters) {
   size_t freeHeapSize;
   for (;;) {
       freeHeapSize = xPortGetFreeHeapSize();
       printf("Free Heap Size: %u bytes\n", (unsigned int)freeHeapSize);
       vTaskDelay(pdMS_TO_TICKS(1000));  // Delay for 1000 ms
   }
}


void vMinEverFreeHeapSizeTask(void *pvParameters) {
   size_t minEverFreeHeapSize;
   for (;;) {
       minEverFreeHeapSize = xPortGetMinimumEverFreeHeapSize();
       printf("Min Ever Free Heap Size: %u bytes\n", (unsigned int)minEverFreeHeapSize);
       vTaskDelay(pdMS_TO_TICKS(1000));  // Delay for 1000 ms
```

```
    }
}



int main(void) {
    // Create tasks
    xTaskCreate(vFreeHeapSizeTask, "FreeHeapTask", configMINIMAL_STACK_SIZE, NULL,1,
NULL);
    xTaskCreate(vMinEverFreeHeapSizeTask, "MinEverFreeHeapTask",
configMINIMAL_STACK_SIZE, NULL,1, NULL);
    // Start the scheduler
    vTaskStartScheduler();

    // The code should never reach here unless there's insufficient heap available to start the scheduler
    for (;;);
}
```

Output:

```
Free Heap Size: 912 bytes
Min Ever Free Heap Size: 912 bytes
Free Heap Size: 912 bytes
Min Ever Free Heap Size: 912 bytes
```