FreeRTOS Memory Management

---

1. Task is in the Heap

When you use xTaskCreate() (the most common API to make a FreeRTOS task):

- FreeRTOS allocates memory from the **heap** for:
  - The **Task Control Block (TCB)** – a small structure that stores info about the task.
  - The **stack** – where the task's local variables, return addresses, and context are stored.
- If your heap is too small, the task creation will fail.

---

2. TCB – Task Control Block

The **TCB** is a data structure inside FreeRTOS that holds:

- Task name
- Task priority
- Pointer to task stack
- Current state (Ready, Running, Blocked, Suspended)
- Linked list pointers (so the scheduler can manage tasks)
- Task runtime stats (if enabled)

Think of the TCB as the **"identity card + notebook"** of the task.

---

3. xTaskCreateStatic()

- In newer versions, you can use **static allocation**:
- StaticTask_t myTCB;
- StackType_t myStack[STACK_SIZE];
-
- TaskHandle_t handle = xTaskCreateStatic(
-     myTaskFunction, "TaskName",
-     STACK_SIZE, NULL,
-     tskIDLE_PRIORITY,
-     myStack, &myTCB);
- Here:
  - You **provide the TCB and stack yourself** (on global/static memory).

- FreeRTOS **does not use the heap**.
- Useful for:
  - Safety-critical systems
  - Avoiding heap fragmentation
  - Systems without a heap at all

---

4. Fragmenting

- **Heap fragmentation** happens when many allocations/frees leave "holes" in memory.
- Example: You have 100 bytes total. You allocate 30, 20, 40, then free the 20 → now you have 70 free but split into two chunks (30+40). You can't fit a new allocation of 50 even though you have 70 total.
- This can cause **xTaskCreate()** or pvPortMalloc() to fail.
- That's why xTaskCreateStatic() is sometimes better → **no fragmentation risk**.

---

5. Heap Management Schemes (1–5)

FreeRTOS provides **configurable heap implementations**:

- **Heap_1**:
  - Very simple, no free().
  - Tasks/objects stay allocated forever.
  - No fragmentation.
- **Heap_2**:
  - Supports free().
  - Simple best-fit allocator.
  - Can fragment.
- **Heap_3**:
  - Just uses the C library malloc() and free().
  - Thread-safe (wrapped in RTOS critical sections).
  - Fragmentation risk depends on compiler library.
- **Heap_4**:
  - More advanced, coalesces adjacent free blocks to reduce fragmentation.
  - Most commonly used.
- **Heap_5**:
  - Like Heap_4 but allows multiple separate memory regions.
  - Useful in systems where memory is split across banks.

You choose which one by setting configFRTOS_MEMORY_SCHEME (or including the correct heap_x.c file).

---

6. Thread Safe?

- FreeRTOS pvPortMalloc() and vPortFree() are **thread-safe** → they use critical sections (disabling interrupts or scheduler) when modifying heap structures.
- This means multiple tasks can call malloc()/free() without corrupting memory.
- But still, **logical issues** (like fragmentation, out of memory) are your responsibility.

✅ **Summary Cheat Sheet:**

- **xTaskCreate()** → allocates TCB + stack on heap.
- **TCB** → stores task state, priority, stack pointer, etc.
- **xTaskCreateStatic()** → avoids heap, you provide memory.
- **Fragmentation** → scattered holes in heap, avoided with static allocation.
- **Heap_1–5** → different memory allocators (from super simple to advanced).
- **Thread Safe** → FreeRTOS heap functions are safe across tasks.

---

1. Arduino and FreeRTOS

- On ESP32, the **Arduino core runs on top of FreeRTOS**.
- At startup, Arduino creates one FreeRTOS task called **loopTask**.
- setup() runs once, then loopTask repeatedly calls loop().

👉 Hidden implementation looks like:

```
void loopTask(void *pvParameters) {
  setup();        // runs once
  for(;;) {
    loop();      // runs forever
  }
}
```

So Arduino's setup/loop = just **one FreeRTOS task**.

---

2. Task Deletion

- vTaskDelete(TaskHandle_t task) deletes a task.
- If you pass **NULL**, it deletes the **currently running task**.
- Inside Arduino:
    - If used in loop(), deletes loopTask during execution.
    - If used at the end of setup(), kills loopTask right after setup finishes → loop() will never run.

Example:

```
void setup() {
  xTaskCreatePinnedToCore(myTask, "MyTask", 2048, NULL, 1, NULL, 1);
  vTaskDelete(NULL);   // kills loopTask
}

void loop() {} // never runs
```

---

3. Multiple Tasks

- After setup() runs:
    - loopTask (Arduino's default task) continues.
    - Any new tasks you created also start running.
- Unless you delete loopTask, you will have **2 tasks running** (or more, if you create more).

---

4. Listing Tasks (Debugging) — Quick

Use vTaskList() to see all running tasks:

```
char buf[512];
vTaskList(buf);
Serial.println("Task\tState\tPrio\tStack\tNum\tCore");
Serial.println(buf);
```

Prints something like:

```
Task       State  Prio  Stack  Num  Core
IDLE0        R     0     116    3    0
IDLE1        R     0     116    4    1
loopTask     R     1     356    5    1
MyTask       R     1     372    6    1
```

🔷 **What is a stack canary?**

- A **stack canary** is a known, fixed “magic value” (like 0xA5A5A5A5) placed at the **end of a task’s stack** when the task is created.
- FreeRTOS (or the compiler’s stack protection) periodically checks whether that value is still intact.

🔷 **Why it works**

- If your task **overflows its stack**, it will overwrite memory beyond the stack’s end.
- The first thing it hits is the **canary value**.
- If the canary no longer matches the expected value → **stack overflow detected** → error handler runs (in ESP32, this usually panics and reboots).

🔷 **FreeRTOS and ESP32**

- FreeRTOS has two levels of stack overflow checking, controlled by configCHECK_FOR_STACK_OVERFLOW:
  - **Method 1**: Checks stack pointer boundaries.
  - **Method 2**: Uses canary values at the stack’s end (stronger).
- On ESP32 Arduino, Method 2 (canary check) is usually enabled by default.

✅ **Summary in one line**:

Yes — stack canary works by writing a sentinel value at the end of each task’s stack. If it changes, FreeRTOS knows the task overflowed its stack and triggers an error.
