MCU Memory, Startup, and Execution Flow (STM32F4 Example)

This document explains **MCU memory organization**, **startup process**, **vector table**, and **C-level initialization flow** in a professional and student-friendly way. The example is based on the **STM32F407 (ARM Cortex-M4)**.

Unlike desktop systems, MCUs have very limited memory (both Flash/ROM and RAM), no operating system (in bare-metal), and often no Memory Management Unit (MMU). That means memory must be handled **manually and carefully**.

Memory Layout in MCU

Typical memory segments in an MCU program (bare-metal C):

1. **Flash / ROM (Non-volatile)**
   - **.text** - Stores program code (instructions).
   - **.rodata** - Also stores constants (`const` variables, lookup tables, strings).
   - Persistent after reset.
2. **SRAM (Volatile RAM)**
   - Divided into:
     - **.data** → initialized global/static variables.
     - **.bss** → uninitialized global/static variables (zero-initialized at startup).
     - **Heap** → used for dynamic memory (`malloc`, `free`).
     - **Stack** → used for function calls, local variables, return addresses.
3. **Registers**
   - Small, ultra-fast storage inside CPU core (not general RAM).

MCU Memory Segments

Flash (Non-volatile)

- Holds the **vector table** at reset.
- Stores **program code (.text)**.
- Stores **read-only data (.rodata)**.
- Stores **initial values of global/static variables (.data load image)**.

SRAM (Volatile)

- Holds **initialized data (.data)**.
- Holds **uninitialized data (.bss)**.
- Provides **stack** (function calls, local variables).
- Provides **heap** (dynamic memory: malloc/free).

Challenges in MCU Memory Management

- **Small size**: RAM may be only a few KB.
- **No virtual memory**: direct addressing, no paging.
- **Heap risks**: malloc/free can fragment memory → dangerous for real-time embedded systems.
- **Stack overflows**: can corrupt heap or data if stack grows too large.

Question: How the source code is converted and divided into text segment, .data segment , .bss segment and etc- how these segments are copied from flash to sram, how these segments ,firmware saved in the flash.

Let's carefully walk from **C source code → compiled ELF → segments → Flash image → runtime in SRAM**.

Step 1: Compilation (C → Object Files)

- You write C:

```
const int table[3] = {1,2,3};     // const → .rodata
int count = 5;                    // global initialized → .data
int flag;                         // global uninitialized → .bss
```

  - Compiler (arm-none-eabi-gcc) generates **object files** (.o).
  - Each .o(object files) already has **sections**:
    - .text → machine instructions
    - .rodata → read-only consts
    - .data → initialized globals
    - .bss → uninitialized globals

At this stage, each .o still has **relocations** (addresses not yet final).

Step 2: Linking (Objects → ELF)

- Linker (ld) uses **linker script** to assign sections into real **memory regions**:

Example (STM32F407):

- .text + .rodata → Flash (0x0800xxxx)
- .data → SRAM (0x2000xxxx), **but its initial values are stored in Flash** (as *load image*)
- .bss → SRAM (0x2000xxxx), but no space in Flash (it's just zeroed at startup)

After linking → result = **ELF file** with symbol table + sections.

Step 3: Firmware Binary (ELF → HEX/BIN)

- We don't flash ELF (too big).
- objcopy creates .bin or .hex.
- This file contains only the **loadable image** (what goes to Flash).

Example layout in Flash (.bin):

0x08000000 → Vector table (SP, Reset_Handler, ISRs)
0x08000100 → .text (instructions)
0x08005000 → .rodata (const data)
0x08008000 → load image of .data (init values for RAM)

👉 .bss does **not exist** in Flash, since it's all zeros — startup code just clears RAM.

Step 4: Flash Programming

- The .bin is written into Flash by ST-LINK.
- Now MCU Flash contains:
    - Vector table
    - Code (.text)
    - Constants (.rodata)
    - Init image for .data

Step 5: Reset & Startup (Flash → SRAM)

1. CPU reads **MSP** + **Reset_Handler** from vector table.
2. Reset_Handler runs:
    - Copy .data init values (from Flash) → to .data region in SRAM.
    - Clear .bss → SRAM zeros.
    - Call main().

At runtime:

- **Code (.text)** executes directly from Flash.
- **Const (.rodata)** read directly from Flash.
- **.data** lives in SRAM with correct initial values.
- **.bss** lives in SRAM initialized to zero.
- **Stack/heap** grow dynamically in SRAM.

Visual Example (STM32F407, 1KB Flash + 512B SRAM sample)

Flash (0x08000000):

0x08000000: [Vector Table]
0x08000100: [Machine code: .text]
0x08000400: [Read-only const: .rodata]
0x08000600: [Init values for .data]

SRAM (0x20000000):

0x20000000: .data → [5, ...] (copied from Flash)
0x20000020: .bss → [0,0,0...] (cleared at startup)
0x20000100: heap ↑
0x20008000: stack ↓

So to answer in one sentence:

- **Compiler separates code/data into sections**.
- **Linker script maps sections to Flash/SRAM addresses**.
- **Objcopy creates binary with Flash content**.
- **Flasher writes binary to MCU Flash**.
- **Reset_Handler copies .data, clears .bss, then runs main().**

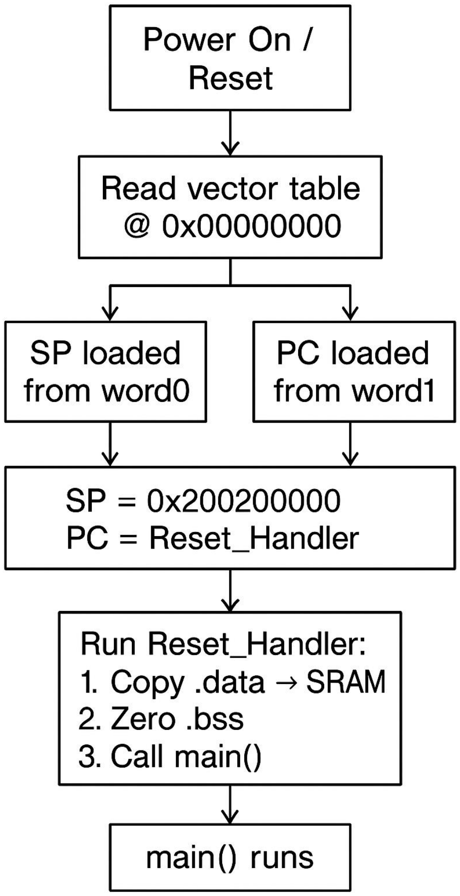
Vector Table

The **vector table** is the first thing in Flash (aliased at 0x00000000). It contains:

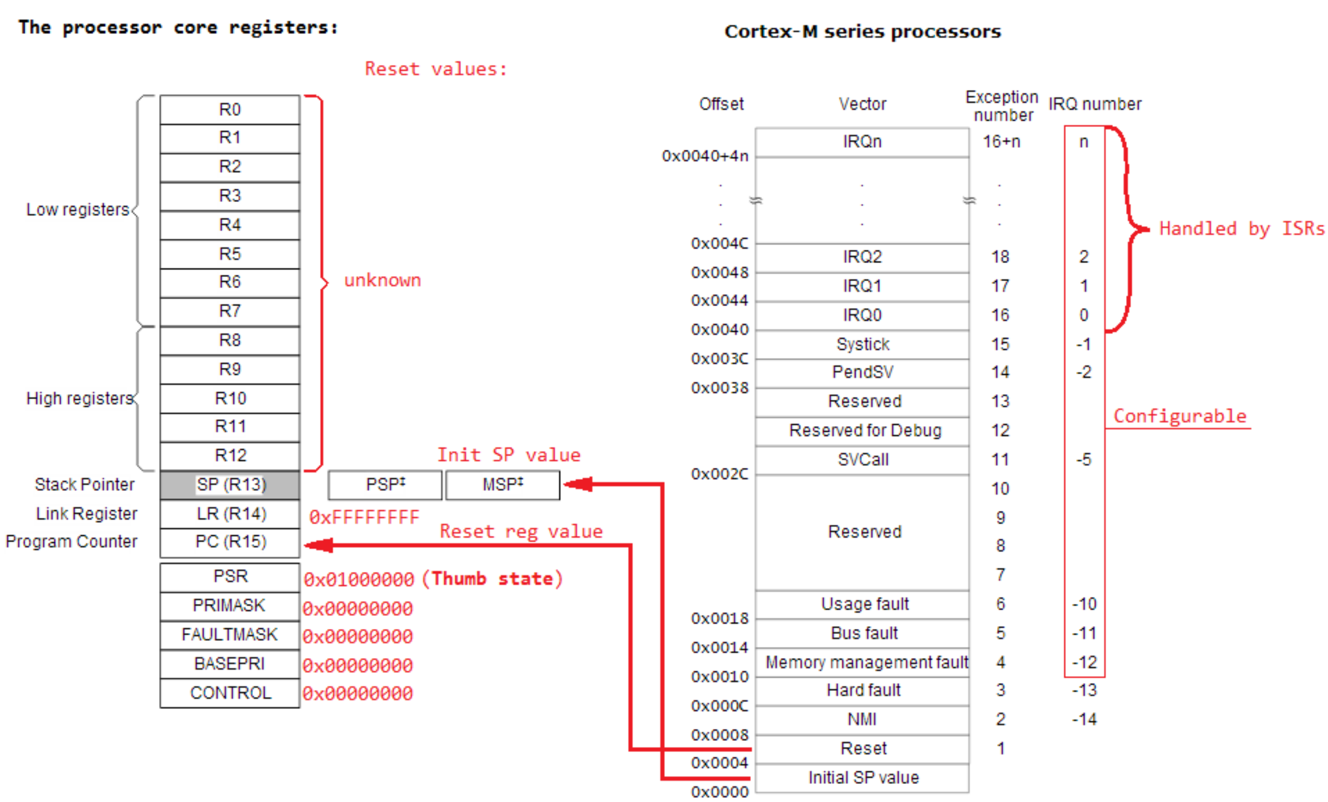
Address	Content	Purpose
0x00000000	0x20020000 (example)	Initial Stack Pointer (SP)
0x00000004	0x08000101 (example)	Reset_Handler (PC)
0x00000008	ISR for NMI	Interrupt handler
0x0000000C	ISR for HardFault	Interrupt handler
…	…	…

Reset Behavior

- **SP ← *(0x00000000)** → top of SRAM.
- **PC ← *(0x00000004)** → Reset_Handler in Flash.
- Execution jumps to Reset_Handler.



Reset Flow Visualization



Reset_Handler Responsibilities

At reset, the Reset_Handler performs:

1. **Copy .data from Flash → SRAM**

```
for (dst = &_sdata, src = &_sidata; dst < &_edata; )
    *(dst++) = *(src++);
```

2. **Zero out .bss**

```
for (dst = &_sbss; dst < &_ebss; )
    *(dst++) = 0;
```

3. **Set up stack & heap (via linker)**.

4. **Call main()**

```
main();
while (1);
```

Flash vs SRAM Segment Comparison (STM32F407 Example)

Segment	Location	Example Address Range	Purpose
Vector Table	Flash	0x08000000 – 0x080003FF	Interrupt vectors + Reset_Handler
.text	Flash	0x08000400 – 0x0807FFFF	Program code
.rodata	Flash	Mixed with .text	Constants, lookup tables
.data (init values)	Flash	Stored after .text	Used to initialize .data in SRAM
.data	SRAM	0x20000000 – 0x20000FFF	Initialized global/static vars
.bss	SRAM	0x20001000 – 0x20001FFF	Zeroed globals/statics
Heap	SRAM	0x20002000 – ...	Dynamic memory allocation
Stack	SRAM	Top: 0x20020000, grows down	Local vars, function frames

MCU Startup Sequence (Bare-Metal, no OS)

When you press reset (or power on), the MCU doesn't just "jump to main()". There are a few important steps first:

1. Reset Vector & Interrupt Vector Table

- At a fixed Flash address (like 0x00000000), the **vector table** is stored.
- It contains:
    - Initial **stack pointer value** (top of SRAM).
    - Reset handler address (function to call after reset).
    - Other interrupt handlers.

So, on reset: **MCU loads stack pointer** and jumps to **Reset_Handler()**.

---

2. Startup Code (Reset_Handler)

The startup code is provided by the compiler's runtime library (like crt0 or CMSIS in ARM Cortex-M).
This code does the **memory setup**:

1. **Copy .data section from Flash → SRAM**
    - Example:
    - int x = 10;   // .data
    - The value 10 is stored in Flash as part of the firmware image.
    - At reset, startup code copies it into SRAM so x lives in RAM.
2. **Zero initialize .bss section in SRAM**
    - Example:
    - int y;   // .bss
    - At startup, y is cleared to 0 in RAM.
3. **Setup heap and stack boundaries**
    - Defined in the linker script.
    - Heap usually starts after .bss and grows upward.
    - Stack starts at top of SRAM and grows downward.
4. **Call main()**
    - After all memory is ready, startup code calls your program's main().

Memory Segments in Detail

Segment	Location	Who Fills It	Example
.text (instructions)	Flash	Compiler/Linker at build time	void foo() {}
.rodata (read-only constants)	Flash	Compiler/Linker	const char msg[] = "Hello";
.data (initialized globals/statics)	Flash (initial values) → copied to SRAM	Startup code	int x = 5;
.bss (uninitialized globals/statics)	SRAM (zeroed)	Startup code	int counter;
Heap (dynamic alloc, malloc/free)	SRAM (grows upward)	Runtime library	malloc()
Stack (function locals, return addr)	SRAM (grows downward)	CPU hardware	function calls

---

Example Walkthrough

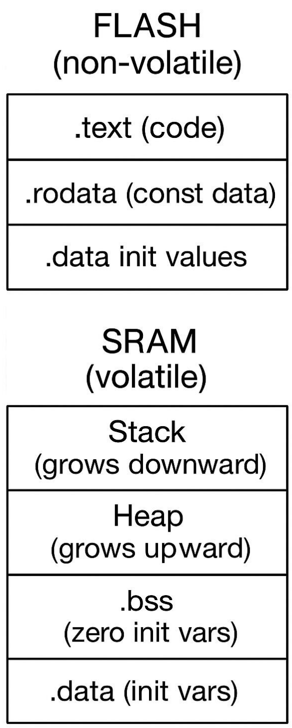
```
const char msg[] = "Flash string";          // .rodata → Flash

int counter;                                // .bss → SRAM, startup sets to 0
int val = 42;                               // .data → Flash init value, copied to SRAM

void foo(void)
 {                                          // .text → Flash
   int local = 5;                          // .stack → SRAM
   int *p = malloc(10 * sizeof(int));       // .heap → SRAM
   // use p...
   free(p);
}
```

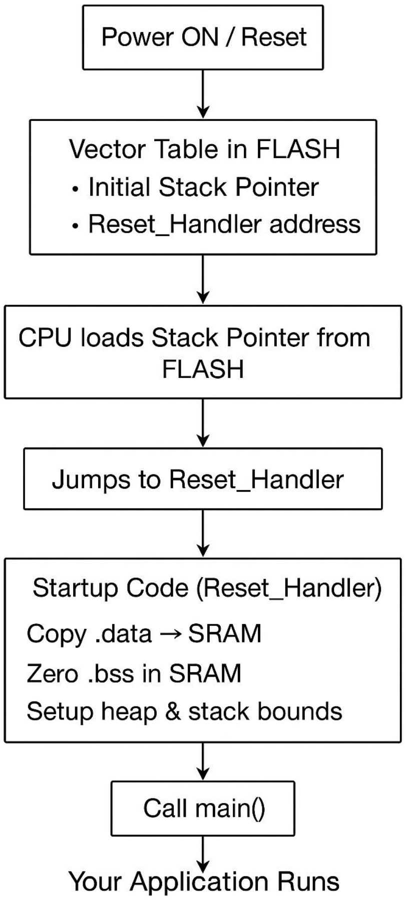
- When firmware is **flashed** → .text, .rodata, .data init values are programmed into Flash.
- On **reset** → startup code copies .data to RAM, zeros .bss.
- At **runtime** → stack and heap are used dynamically.

Visual Diagram (Typical MCU with Flash + SRAM)



- Flash holds **instructions and constants**.
- SRAM holds **mutable data** (variables, stack, heap).
- Startup code bridges the two: copying .data and zeroing .bss.

MCU Startup Flow (Text Visualization)



Example Walkthrough

1. **FLASH before reset:**
   - .text: contains instructions (main(), functions).
   - .rodata: contains constant strings.
   - .data init table: values for globals (e.g. int g = 5;).
2. **On Reset:**
   - CPU loads **stack pointer**.
   - CPU jumps to Reset_Handler.
3. **Reset_Handler executes:**
   - Copies 5 from Flash to SRAM → now g = 5 in RAM.
   - Clears .bss variables (int counter; → set to 0).
   - Prepares heap/stack regions.
4. **main() runs:**
   - Locals go on **stack**.
   - malloc() gets memory from **heap**.
   - Globals accessed from **SRAM**.
   - Constants (strings, tables) stay in **FLASH**.

Here's a mix of conceptual and practical questions (from beginner → intermediate → advanced):

◈ Level 1: Basics

1. Where is the **vector table** stored in an STM32 MCU at reset?
2. What two values does the CPU load from the first two entries of the vector table?
3. Which memory segment is used for **uninitialized global/static variables**?

◈ Level 2: Applied

4. If you declare `const int a = 5;`, in which memory segment does it go?
5. If you declare `int b = 5;`, where is it stored at runtime, and what happens at startup?
6. What does the **Reset_Handler** do before calling `main()`? (list 3 steps)

◈ Level 3: Advanced

7. In the linker script, why do we place `.data` in **SRAM** but also store its **load image in Flash**?
8. Suppose your stack grows too large — which memory segment could it collide with?
9. If the **PC is initialized with the value at address `0x00000004`**, explain what happens if that value is corrupted or invalid.

◈ Advanced "What-If" Scenarios

Q1. You declare:

```
int big_array[20000];
```

Where does this array go if declared **globally** vs **inside `main()`**? What happens if it doesn't fit?

Q2. You change the linker script so the **vector table is in SRAM** instead of Flash.

- Why might someone do this?
- What do you need to configure in the MCU for it to work?

Q3. If `.bss` is not zeroed during Reset_Handler, what kind of bugs could appear in your program?

Q4. Suppose you declare a **global const array** but accidentally forget const:

```
const int table[5] = {1,2,3,4,5};   // vs
int table2[5] = {1,2,3,4,5};
```

How does this change memory usage between Flash and SRAM?

Q5. You use **malloc()** repeatedly but never call **free()**.

- Where does this memory come from?
- What happens over time on a microcontroller with 128KB SRAM?

Q6. The stack pointer (MSP) is incorrectly initialized (e.g., not pointing to valid RAM).
What exactly will break first:

- local variable usage?
- function calls?
- interrupts?

Q7. Imagine you place **a variable in .rodata using const** but later try to modify it by force casting away const.
What happens at runtime (on Cortex-M with Flash memory)?

Best Practices

1. **Prefer static allocation**
   Allocate arrays, buffers, and structures at compile-time whenever possible. Example:
2. static uint8_t rx_buffer[128];
3. **Avoid malloc/free** in critical systems
   If dynamic memory is needed, use:
   - A **fixed-size memory pool**.
   - Custom allocators with bounded behavior.
4. **Monitor stack usage**
   - Place a known pattern at the stack start and check runtime usage.
   - Keep stack and heap separate if linker allows.
5. **Use linker scripts** wisely
   - Control placement of variables (.data, .bss, .heap, .stack).
   - Place frequently accessed data in fast SRAM or tightly coupled memory.
6. **Protect against overflow**
   - Watchdog timers to reset on corruption.
   - Stack guards / MPU (if MCU supports it).

Summary

- At reset, CPU reads SP and PC from vector table (0x00000000 and 0x00000004).
- Reset_Handler copies .data, clears .bss, then calls main().
- **Flash** holds program and constants. **SRAM** holds variables, stack, heap.
- Linker script + startup code work together to map all sections correctly.
- The **flow diagram + table** make it easy to imagine how memory is used step by step.