Memory Management

Program Execution Notes

- **Instructions location**
    - In Ben Eater's CPU https://www.youtube.com/watch?v=yl8vPW5hydQ → stored in EEPROM (ROM).
    - In modern PCs → stored in the **.text section** of RAM (after being loaded from disk).
- **.text section**
    - Holds machine code instructions.
    - Read-only region of program memory (prevents accidental modification).
- **Program Counter (PC)**
    - CPU register pointing to the next instruction address.
    - Starts at a fixed reset vector address (depends on CPU).
    - Moves sequentially unless instruction changes it (jump, call, branch).
- **Fetch–Decode–Execute cycle**

1. **Fetch** instruction from memory (.text in RAM).
2. **Decode** what it means.
3. **Execute** operation.
4. Update PC → repeat.

- **ROM vs RAM difference**
    - ROM (EEPROM, flash) → non-volatile, permanent storage of code.
    - RAM → fast, temporary workspace for executing programs.
    - Modern systems load program instructions from disk → RAM → CPU executes from RAM.

Perfect — here's a clean set of notes you can keep for reference:

---

1. General Concepts

- **Sections in a program:**
  - **.text** → program instructions (machine code).
  - **.rodata** → read-only constants (strings, lookup tables).
  - **.data** → initialized globals/statics.
  - **.bss** → uninitialized globals/statics (zeroed at startup).
  - **Heap** → dynamic allocations (malloc, new).
  - **Stack** → function calls, local variables, return addresses.
- **Program Counter (PC)** → always points to the next instruction to fetch, decode, execute.

---

2. Embedded Systems (e.g. ARM Cortex-M, AVR, ESP32)

- **Program storage:**
  - .text and .rodata → stored in **flash/ROM** (non-volatile).
  - CPU fetches instructions directly from flash.
- **RAM usage:**
  - .data → copied from flash to RAM at startup.
  - .bss → zero-initialized in RAM.
  - Heap (optional, grows upward).
  - Stack (grows downward from top of RAM).

Layout (simplified):

```
Flash (ROM):
0x0000   Reset vector, ISRs
      Program instructions (.text)
      Constants (.rodata)

RAM:
0x2000_0000  .data (initialized globals)
           .bss  (uninitialized globals)
           Heap (grows up)
           ...
           Stack (grows down)
0x2000_FFFF  End of RAM
```

---

3. PCs (with OS, e.g. Linux, Windows)

- **Program storage:**
  - Program is on disk/SSD.
  - OS **loads .text + .rodata + .data + .bss** into **RAM** when you start it.
  - CPU executes instructions from RAM (not from disk).
- **Memory in RAM contains:**
  - .text → program instructions.
  - .rodata → constants.
  - .data, .bss.
  - Heap (dynamic).
  - Stack (local variables, calls).

Layout (simplified):

```
RAM:
0x0000_0000   Program instructions (.text)
              Read-only constants (.rodata)
              Initialized data (.data)
              Uninitialized data (.bss)
              Heap (grows up)
              ...
              Stack (grows down)
0xFFFF_FFFF   End of virtual address space
```

---

4. Key Differences

Aspect	Embedded System	PC (with OS)
Program storage	Flash/ROM (non-volatile)	Disk/SSD → loaded to RAM
Execution location	Directly from flash	From RAM
RAM role	Variables + stack/heap	Everything (code + data)
Loader/OS	Often none (bare-metal)	OS handles loading

---

5. Why the Difference?

- **Embedded:** Flash is directly addressable and reasonably fast. RAM is small, so only variables live there.
- **PC:** Disks are too slow → programs must be copied to RAM before execution. CPUs rely on caches + virtual memory.

✅ **Summary:**

- In **embedded systems**, .text runs from **flash**, RAM is for data.
- In **PCs**, .text and all sections are loaded into **RAM**, and the CPU runs from there.

Why split .data vs .bss?

1. **Storage efficiency in flash**
   - **.data (initialized statics/globals)** → needs to store the *initial values* in flash so they can be copied to RAM at startup.
     - Example:
     - int x = 42; // must keep 42 somewhere → stored in flash → copied into RAM
   - **.bss (uninitialized statics/globals)** → by definition, you don't care about the initial value. C standard says they *must start at 0*.
     - Instead of wasting flash space by storing a bunch of zeros, the startup code just **clears that region of RAM**.
     - Example:
     - int y; // no explicit value → placed in .bss → startup code sets y=0 in RAM

   👉 This saves ROM/flash space (only the *size* of .bss is stored, not the actual zero bytes).

---

2. **Startup performance**
   - Copying 1000 variables with initial values → requires 1000 reads from flash.
   - Zeroing .bss → just a fast memory clear loop (set region to 0).
   - Much faster than copying stored zeros.

---

3. **Standard compliance**
   - The **C standard** guarantees that:
     - Uninitialized global/static variables → default to 0.
     - Local variables (on the stack) → *not automatically initialized*.
   - The .bss section is the mechanism compilers + linkers use to implement this guarantee efficiently.

Example

```
int a = 5;    // goes into .data (flash stores "5", copied to RAM)
int b;        // goes into .bss (flash stores nothing, RAM cleared to 0)
```

Flash contents:

[data] 0x05
[bss] (nothing, just metadata saying "reserve N bytes")

Startup sequence:

1. Copy values from .data in flash → RAM.
2. Clear .bss region in RAM → all zeros.

✅ **Summary:**
We split .data and .bss because:

- .data requires stored initial values.
- .bss doesn't need storage → save flash space and speed up startup.

Yes — you've got the core idea right 👍 Let me give you clean notes on **stack vs heap** so it's 100% clear:

---

1. Stack

- **Purpose:**
  - Stores **function call frames**: local variables, return addresses, saved registers.
- **Behavior:**
  - Grows **downward** from the top of RAM.
  - Each function call **pushes** a new frame.
  - When the function returns, the frame is **popped** (memory automatically freed).
- **Management:**
  - Automatic, handled by CPU + compiler.
  - Size is **fixed at runtime** (e.g. set by linker or OS).
- **Lifetime:**
  - Variables exist only during function execution.
- **Problems:**
  - If too many nested calls or large locals → **stack overflow**.

Example:

```
void foo() {
   int x = 42;   // stored on stack
}  // x destroyed when foo() returns
```

---

2. Heap

- **Purpose:**
  - Stores **dynamically allocated memory** (objects created with malloc, new, free).
- **Behavior:**
  - Grows **upward** from the end of .bss/.data.
  - Managed explicitly by programmer.
- **Management:**
  - You request memory → allocator finds space on the heap.
  - You must free it later.
- **Lifetime:**
  - Variables exist until you free them (or program ends).
- **Problems:**
  - Memory leaks (if you forget to free).
  - Fragmentation (lots of small allocations).

Example:

```
int *p = malloc(10 * sizeof(int));  // heap allocation
// use p...
free(p);                        // must release
```

---

3. Stack vs Heap Side by Side

Feature	Stack	Heap
Who manages	Compiler/CPU (automatic)	Programmer (malloc/free)
Growth	Downward from top of RAM	Upward from data section
Lifetime	Until function returns	Until freed (or program ends)
Speed	Very fast (simple pointer move)	Slower (needs allocator logic)
Size	Fixed at startup (linker/OS)	Flexible (but limited by RAM)
Typical use	Function locals, return addr	Large/variable-sized buffers

---

4. In Embedded Systems

- **Stack:**
  - Critical resource, often small (e.g. 1–4 KB).

    - Must be carefully sized — stack overflows can crash MCU.
- **Heap:**
    - Sometimes avoided entirely → can be dangerous on small MCUs.
    - Embedded devs often use **static allocation** instead of heap.

✅ **Summary:**

- **Stack** = automatic, function-based, fixed-size, very fast.
- **Heap** = manual, dynamic, flexible, but slower and error-prone.
- On **embedded systems**, stack is used heavily, heap only if absolutely necessary.