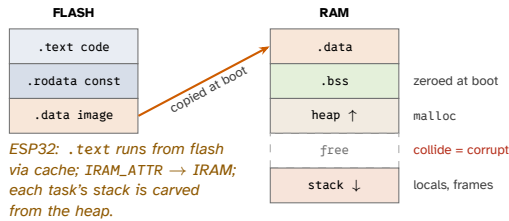


In one line: Code and constants sit in **flash** (.text, .rodata); initialised globals in .data (RAM, with an image in flash), zero ones in .bss (RAM, zeroed at boot, *no flash*); malloc in the **heap**; locals on the **stack**. Struct members sit at multiples of their alignment → padding. Byte order in memory is the **endianness**. Arithmetic on anything narrower than int happens in **int**.

Picture — segments on an MCU



declaration	lives in	starts as
int g = 5; (file scope)	.data	5 (flash copy)
int z; int z = 0;	.bss	0 guaranteed
static int n; in a function	.bss	0, keeps value
const int k = 5; (file)	.rodata	5, flash
"literal"	.rodata	read-only
int x; in a function	stack/register	indeterminate
const int c; in a fn	stack	const ≠ flash
p = malloc(n)	heap (p: stack)	garbage

Before main the startup code (crt0) copies the .data image flash → RAM, zeroes .bss, sets the stack pointer. size app.elf / idf.py size print text/data/bss. **Stack overflow** (your Q21): recursion or a big local (uint8_t buf[8192]); no MMU → *no fault*, it silently overwrites the neighbour. Detect: configCHECK_FOR_STACK_OVERFLOW 2 (canary), uxTaskGetStackHighWaterMark().

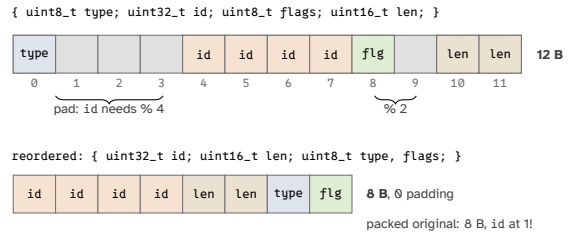
Struct padding — the rules

- Each member starts at a multiple of its **alignment** (usually its size); the compiler inserts padding before it.
- sizeof rounds up to the largest alignment, so arrays of the struct stay aligned: {uint32_t a; uint8_t b;} = 8.
- Fix: order members **largest** → **smallest**. Pin wire formats: _Static_assert(sizeof(struct Msg) == 8, "layout"); offsetof(struct Msg, len) (<stddef.h).
- Padding bytes are unspecified: don't memcmp structs; memset before sending one (info leak).
- __attribute__((packed)): no padding, but members may be unaligned → slower byte-wise code, and &m.id used as a uint32_t * faults: **ESP32 LoadStoreAlignment panic**, Cortex-M0 HardFault (M3/M4 tolerate plain LDR).

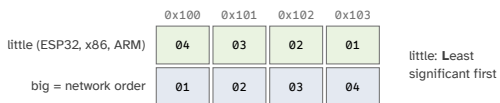
Likely questions

- Why does .bss cost no flash? — only its size is stored.
- Read a u16 off the wire? — p[0] << 8 | p[1], or memcpy + ntohs.
- i = -1; i < sizeof buf? — false: i becomes size_t.

Picture — struct Msg byte by byte (32-bit)



Picture — 0x01020304 at address 0x100



Example — big-endian u32 from a buffer

```

uint32_t rd_be32(const uint8_t *p) { // any alignment
    return (uint32_t)p[0] << 24 | (uint32_t)p[1] << 16
        | (uint32_t)p[2] << 8 | p[3]; // cast! p[0]<<24 is int
}

uint32_t v; memcpy(&v, buf + 1, 4); v = ntohl(v); // same
uint32_t bad = *(uint32_t *) (buf + 1); // unaligned+aliasing: UB

```

Shifts act on *values*: endian-independent. hton*/ntoh* swap on little-endian only.

Integer promotion + usual conversions

- Narrower than int (uint8_t, uint16_t, char, bit-fields) → int first. uint8_t a=200, b=100; a+b = 300; only uint8_t r = a+b; truncates to 44.
- ~x on uint8_t 0xFF = int 0xFFFFFFF0 ≠ 0 — write (uint8_t)~x.
- uint16_t 65535*65535 is int math → signed overflow, **UB**. Cast: (uint32_t)a*b.
- Same rank, mixed sign → **unsigned wins**: -1 < 1u is false; int i=-1; i < sizeof buf is false (size_t).

Interview traps

- Q11:** globals are never on the stack; .bss is 0, not random. Only locals start indeterminate.
- Q5/Q19:** draw the layout — offsets, pads, trailing pad.
- Q9:** you reversed it: bytes 01 02 03 04 read as uint32_t on ESP32 = 0x04030201.
- Q14:** right output, wrong reason — no wrap, it is int 300.

Remember

Flash keeps code, constants and the .data image; .bss costs no flash. **Big first**, no padding. **Little-endian** = little end first. Small ints become int.