Embedded C Interview Question and Answer.

Set -3

Linkedin

Owner	UttamBasu
Author	Uttam Basu
Linkedin	www.linkedin.com/in/uttam-basu/

Level - Easy

Uttam Basu

1) What happens if you don’t use `volatile` for a memory-mapped hardware register?

If you **don’t** use the `volatile` keyword for a hardware register (or any variable that can change outside of program flow), the **compiler assumes it doesn't change unexpectedly**. As a result, the compiler may **optimize away reads/writes**, leading to **incorrect or unexpected behavior**.

Example:

Let’s say we’re polling a hardware status register to wait until a flag becomes 1:

```
#define STATUS_REG   (*(unsigned char*)0x40000000)

while (STATUS_REG == 0);  // Wait for the flag
```

If `STATUS_REG` is **not declared as `volatile`**, the compiler might **optimize** this to:

```
unsigned char temp = STATUS_REG;
while (temp == 0);  // Infinite loop! Even if the hardware changes
STATUS_REG
```

Because the compiler **doesn’t see any code that could change `STATUS_REG`**, it assumes it never changes — and skips re-reading it from memory. So you might end up stuck in that loop **forever**, even if the hardware updates the register.

Correct Usage:

```
#define STATUS_REG   (*(volatile unsigned char*)0x40000000)
```

Now the compiler knows it **must re-read** `STATUS_REG` on every loop iteration — preventing this dangerous optimization.

Summary:

- Without `volatile`: Compiler might **cache** the value →**stale data**, bugs, infinite loops.
- With `volatile`: Compiler **always fetches the actual value** →stays in sync with hardware.
- Applies to:
    - Hardware registers
    - Global variables modified in ISRs
    - Shared memory in multi-threaded systems

Uttam Basu

2) Explain the difference between a stack overflow and a heap overflow.

A. Stack Overflow:
A **stack overflow** occurs when you use **more stack memory than is available**, typically due to:

- Too many nested function calls (deep recursion)
- Large local variables (like big arrays on the stack)

In Embedded Systems:

- Stack size is **limited and fixed** (especially on microcontrollers)
- Can lead to **crashes, data corruption**, or overwriting other memory sections like global variables

Example:

```
void recursive_function() {
    int buffer[1024];
    recursive_function();  // infinite recursion
}
```

This will keep allocating `buffer` on the stack →eventually the stack overflows.

B. Heap Overflow: A **heap overflow** happens when a program writes more data to a dynamically allocated block (via `malloc`) than it was allocated for.

In Embedded Systems:

- Can corrupt the heap structure
- May overwrite adjacent memory or cause a crash
- Dangerous if memory is tight or `malloc` is poorly managed

Example:

```
char *ptr = malloc(10);
strcpy(ptr, "This is more than 10 bytes!");
```

Here, you're writing **beyond the bounds** of what was allocated — this is a **heap overflow**.

[Uttam Basu]()

Key Differences:

Feature	Stack Overflow	Heap Overflow
Memory Region	Stack (function calls, locals)	Heap (dynamic memory)
Cause	Deep recursion, large locals	Writing past allocated buffer
Detection	Often causes crash/reset	Harder to detect, may corrupt data
Common in	Recursive functions	Unsafe memory operations (e.g. strcpy)
Embedded concern	Stack is small →easier to overflow	Heap use is discouraged or limited

3) How does `static` behave differently for global, local, and function variables and ISR?

A. `Static` Global Variable:

- Limits the **scope to the current file (translation unit)**.
- Prevents external linkage — it **can't be accessed using** `extern` in other files.

```
static int sensor_threshold = 50;  // Only visible in this file
```

B. `Static` Local Variable (Inside Function):

- Variable retains its **value between function calls**.
- Initialized **only once** during program startup, not every time the function is called.

```
void count_calls() {
    static int counter = 0;
    counter++;
    printf("%d\n", counter);
}
```

Output: `1, 2, 3...` each time it's called.

C. `Static` Function:

- Restricts **visibility of the function to the file** (like private functions in OOP).

```
static void helper_function() { ... }
```

D. `Static` in ISR (Interrupt Service Routine):

- Used for **persistent state** (e.g., counters, debounce logic).
- Critical for **small, efficient ISRs** since you can't use dynamic memory or large stack usage.

```
void ISR_handler(void) {
    static uint8_t toggle = 0;
    toggle = !toggle;
}
```

4) How would you detect a memory leak in an embedded system with no OS and limited RAM?

What Is a Memory Leak?

A memory leak occurs when:

- You `malloc()` (or similar) memory
- Then **lose all references to it** without `free()`-ing it
- The memory stays allocated forever — even though it's no longer used

Over time, this can **exhaust RAM** and crash the system.

Detect Memory Leaks in Bare-Metal Embedded Systems:

A. Custom malloc/free wrappers (Heap tracking)

Wrap `malloc()` and `free()` to **log allocations**, keep count, and track active blocks.

```c
typedef struct {
    void* ptr;
    size_t size;
    const char* tag;
} MemTrack;
#define MAX_TRACKED_BLOCKS 50
MemTrack mem_map[MAX_TRACKED_BLOCKS];

void* my_malloc(size_t size, const char* tag) {
    void* ptr = malloc(size);
    for (int i = 0; i < MAX_TRACKED_BLOCKS; ++i) {
        if (mem_map[i].ptr == NULL) {
            mem_map[i].ptr = ptr;
            mem_map[i].size = size;
            mem_map[i].tag = tag;
            break;
        }
    }
    return ptr;
}

void my_free(void* ptr) {
    for (int i = 0; i < MAX_TRACKED_BLOCKS; ++i) {
        if (mem_map[i].ptr == ptr) {
            mem_map[i].ptr = NULL;
            mem_map[i].size = 0;
            mem_map[i].tag = NULL;
            break;
        }
    }
    free(ptr);
}
```

Then, add a function to **print the memory map** or check for unfreed blocks at runtime or on system exit/reboot.

B. Track Heap Usage with a Watermark

If you can't afford full tracking, at least monitor **peak heap usage**.

- Fill the heap region with a known pattern (e.g., `0xAA`)
- After runtime, check how much of that pattern remains → shows how much heap was used

Some embedded libraries or toolchains (e.g., `newlib`, `malloc-stats`, FreeRTOS heap_x.c) support this.

Uttam Basu

C. Static Analysis Tools

Use tools like:

- **PC-lint / Flexelint**
- **Coverity**
- **Clang Static Analyzer**
- Some IDEs (e.g., IAR, Keil) have built-in leak detection for statically analyzed memory paths

These help find leaks at compile-time — especially if allocation happens conditionally and free is skipped.

D. Hardware Debugger + Map File

- Use the **linker map file** to find heap location
- Use a hardware debugger (e.g., J-Link, ST-Link, etc.) to inspect heap memory at runtime
- If memory usage keeps growing between known states →potential leak

E. Avoid Dynamic Allocation Altogether

The best memory leak is one that can't happen. ▯

In small embedded systems, consider:

- Using **fixed-size memory pools** (e.g., from a memory manager or RTOS)
- Using **static allocations** only
- Allocating once at startup and never freeing

Summary:

Method	Pros	Cons
Custom malloc/free tracking	Detects exact leaks	Code/data overhead
Heap watermark	Low overhead	Doesn't show where leak is
Static analysis tools	Finds bugs before they happen	Not always accurate on runtime
Debugger + map file	Great for manual inspection	Tedious, not automated
Avoid dynamic memory	Most robust approach	Less flexible design

Uttam Basu

5) What is memory-mapped I/O, and how does it differ from port-mapped I/O?

Memory-Mapped I/O (MMIO):

In **Memory-Mapped I/O**, **peripheral devices** (like GPIO, UART, ADC, etc.) are assigned specific **addresses in the memory address space**.

So you access I/O devices the **same way you access regular variables** or memory — using pointers.

Example (common in ARM Cortex-M microcontrollers):

```
#define GPIO_PORTA (*(volatile uint32_t*)0x40004000)
GPIO_PORTA = 0x01;  // Set pin
```

- `0x40004000` is not RAM — it's the address of a **hardware register**.
- The CPU talks to hardware via normal **load/store instructions**.
- Works with **standard C pointers**.

Let's go on more details, **MCU** has a memory map like this:

Region	Address	Notes
Code (Flash)	0x00000000	
SRAM (Data)	0x20000000	
Peripherals (MMIO)	0x40000000	GPIO, UART, ADC, etc.
System Control Space	0xE0000000	

Everything — RAM, Flash, and I/O — is mapped into a single **32-bit address space**.

Example:

Controlling an LED with GPIO on STM32 (Memory-Mapped I/O):

Let's say we want to turn on an LED connected to **GPIO Port A, Pin 5**. We'd write to a register like `GPIOA->ODR` (Output Data Register) — which lives at a **fixed address** in the memory space.

STM32 GPIO Register Layout:

```
#define GPIOA_BASE    0x40020000UL
#define GPIOA_ODR     (*(volatile uint32_t *)(GPIOA_BASE + 0x14))
```

Code to Turn On the LED:

```
#define LED_PIN  (1 << 5)  // Pin 5

int main(void) {
    // Set PA5 high
    GPIOA_ODR |= LED_PIN;

    while (1);
}
```

That GPIOA_ODR is a memory-mapped **hardware register**. You're talking to hardware using **pointer dereferencing** — just like normal memory.

Visual Diagram: Memory-Mapped I/O Access:

```
   Your Code                CPU/MCU           Memory Space
   ---------               -----------      ------------------------
   GPIOA_ODR |= 0x20;        --->             Address 0x40020014
                                             +----------------------+
                                             | GPIOA Output Reg     |
                                             +----------------------+
                                             | Value: 0x00000020    |
```

But You Never Actually Write That Manually...

Most modern embedded toolchains (STM32 HAL, CMSIS, AVR, etc.) give you **predefined macros and structs** like this:

```
GPIOA->ODR |= (1 << 5);
```

Where GPIOA is defined as:

```
#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)
```

And `GPIO_TypeDef` is a `struct` mapping all GPIO registers:

```
typedef struct {
    volatile uint32_t MODER;
    volatile uint32_t OTYPER;
    volatile uint32_t OSPEEDR;
    volatile uint32_t PUPDR;
    volatile uint32_t IDR;
    volatile uint32_t ODR;  // Output Data Register
    // ...
} GPIO_TypeDef;
```

Advantages of Memory-Mapped I/O:

Feature	Benefit
Simple access	Use standard C syntax and instructions
Efficient	No special instructions required
Unified space	Memory and I/O in the same address space

Port-Mapped I/O (PMIO):

In **Port-Mapped I/O**, I/O devices have a **separate address space**, and you need **special instructions** (like `IN` and `OUT` in x86 assembly) to access them.

Example (in x86 assembly):

```
MOV DX, 0x03F8    ; COM1 port
MOV AL, 'A'
OUT DX, AL        ; Send character to COM1
```

- Uses separate I/O space (not memory).
- Can’t access with regular C pointers.

Port-Mapped I/O: Downsides (especially for embedded):

Feature	Drawback
Not portable	Only exists on some architectures (like x86)
Requires ASM	Can't use standard C to access
Slower access	Often more restricted and limited

[Uttam Basu]()

Summary:

Feature	Memory-Mapped I/O	Port-Mapped I/O
Address space	Shared with memory	Separate I/O space
Access method	Regular pointers in C	Special CPU instructions
Portability	Very portable (used in ARM, RISC-V, etc.)	Limited (x86-like architectures)
Simplicity	Easy to use in C	Needs low-level code
Common in embedded	Yes	Rare (almost never)

In Embedded Systems (e.g., ARM Cortex-M, AVR, STM32, etc.)

Memory-Mapped I/O is the standard — peripherals are accessed via register addresses defined in header files.

6) What are race conditions? How can you prevent them in embedded C?

Race Condition:

A **race condition** happens when:

- Two or more contexts (e.g., main loop + ISR, or two tasks in RTOS)
- **Access shared data** (read or write)
- And **at least one of them writes**
- Without **proper synchronization**

Result:

Behavior depends on the **timing of execution**, which is **unpredictable** →leads to **data corruption**, crashes, or wrong behavior.

Example of a Race Condition (Main vs. ISR):

```
volatile uint8_t button_press_count = 0;

void ISR_Button_Handler(void) {
    button_press_count++;   // Could be interrupted mid-increment!
}

int main() {
    if (button_press_count > 0) {
        button_press_count--;   // Could corrupt the value
        // process button
    }
}
```

This is **not safe** — `++` and `--` are **not atomic** for multi-byte or even some single-byte values depending on CPU architecture.

How to Prevent Race Conditions in Embedded C?

A. Disable Interrupts During Critical Sections

Temporarily disable interrupts while accessing shared data.

```
__disable_irq();            // or cli() for AVR
button_press_count--;
__enable_irq();             // or sei() for AVR
```

Use this carefully to **minimize the disabled time** so other ISRs aren't blocked too long.

B. Use Atomic Operations

On some platforms (e.g., ARM Cortex-M), you can use **atomic bit manipulation instructions** or functions.

If available:

```
#include <stdatomic.h>
atomic_uint button_press_count;
```

On many MCUs, you can also **emulate atomic access** with `LDREX/STREX` instructions or use CMSIS atomic functions.

C. Use Volatile Correctly

Mark shared variables as `volatile` to prevent compiler optimizations that might cache the value.

```
volatile uint8_t flag;  // Always read from memory
```

But note: `volatile` **does not** make access safe — it just prevents the compiler from optimizing it out.

D. Use Mutexes / Semaphores (with RTOS)

If you're using an RTOS like FreeRTOS:

```
xSemaphoreTake(mutex, portMAX_DELAY);
shared_var++;
xSemaphoreGive(mutex);
```

Ensures **only one task** accesses the critical region at a time.

E. Double Buffering or Message Queues

Instead of sharing raw variables, **send data through a buffer or queue**.

- ISR pushes data to a buffer or queue
- Main loop pops and processes

This separates data access and avoids contention.

Summary:

Technique	Use Case	Notes
Disable interrupts	Main loop vs ISR	Simple, use sparingly
Atomic operations	Multi-context systems	Preferred if hardware supports
Volatile	ISR-shared variables	Prevents compiler caching
Mutexes/semaphores (RTOS)	Task-to-task data	Clean and safe
Message queues / buffers	ISR →main data handoff	Highly recommended

Uttam Basu

Tips:

Safe counter increment between main and ISR

```
// Increment in ISR
void ISR_Handler(void) {
    button_press_count++;
}

// Decrement in main safely
uint8_t local_count;

__disable_irq();
if (button_press_count > 0) {
    button_press_count--;
    __enable_irq();
    // process button
} else {
    __enable_irq();
}
```

This ensures atomicity, avoids race conditions, and keeps ISRs fast (which is key).

7) What is a linker script?

A **linker script** is like the **map and instruction sheet** that tells the linker **how to lay out your embedded program in memory**.

When you're working on **embedded systems**, memory layout is **not optional** — it's crucial. You need to tell your program exactly where to put things like:

- Code (`.text`)
- Initialized data (`.data`)
- Uninitialized data (`.bss`)
- Stack, heap, interrupt vector table, etc.

This is what the linker script handles.

Definition:

A linker script is a configuration file used by the linker (usually `ld`) to control how your program's sections are mapped to physical addresses in memory.

Why It's Important in Embedded C

Unlike PCs, microcontrollers:

- Have **fixed memory sizes**
- Have **specific addresses** for peripherals, bootloaders, vectors, etc.
- Often use **bare-metal** memory (no OS/virtual memory)

So, we must **manually control** the memory layout — and that's what linker scripts do.

Basic Anatomy of a Linker Script

Example (simplified STM32):

```
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
  RAM   (rwx): ORIGIN = 0x20000000, LENGTH = 128K
}

SECTIONS
{
  .text : {
    KEEP(*(.isr_vector))      /* Interrupt vector table */
    *(.text*)                 /* Application code */
    *(.rodata*)               /* Read-only data */
  } > FLASH

  .data : {
    _sdata = .;
    *(.data*)
    _edata = .;
  } > RAM AT > FLASH

  .bss : {
    _sbss = .;
    *(.bss*)
    *(COMMON)
    _ebss = .;
  } > RAM
}
```

What It Controls:

Section	Meaning
MEMORY	Defines physical memory regions (Flash, RAM)
SECTIONS	Maps program sections (.text, .data, .bss, etc.) to memory
ORIGIN	Start address of region
LENGTH	Size of region
> RAM	Store in RAM
AT > FLASH	Load from FLASH

Real-World Uses:

- Place interrupt vector table at `0x08000000`
- Put a bootloader in lower Flash, app in higher Flash
- Reserve memory for peripherals or DMA
- Define memory for special buffers (e.g. USB, Ethernet)

Summary:

Feature	Purpose
Define memory layout	FLASH, RAM, peripherals, etc.
Control placement	Vector tables, code, data, stack, heap
Custom sections	DMA buffers, bootloaders, memory-mapped areas
Absolutely essential	For bare-metal and low-level embedded work

Uttam Basu