Embedded C Interview Question and Answer.

Set -2

Linkedin

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

Level - Hard

Uttam Basu

1) **Explain how you would write an Interrupt Service Routine (ISR) in Embedded C. What precautions must be taken when accessing shared variables between the main code and ISR? How do you make your code reentrant and race-condition safe?**

In most microcontroller platforms (e.g., ARM Cortex-M, AVR, MSP430), you define an ISR like this:

Example (for ARM Cortex-M using CMSIS):

```
void TIM2_IRQHandler(void)
{
    // Clear the interrupt flag first
    TIM2->SR &= ~TIM_SR_UIF;

    // Your interrupt logic here
    toggle_LED();
}
```

Key Characteristics:

- ISRs must be void return type, take no parameters, and usually have specific names depending on the microcontroller's vector table.
- Must be kept short and fast — no delays, avoid complex logic, avoid blocking calls.

Precautions for Shared Variables Between ISR and Main Code

When both ISR and main code access the same variable (e.g., a counter, flag, or buffer), you must protect against race conditions.

Strategies:

1. **Use volatile keyword:**

```
volatile uint8_t flag = 0;
```

Ensures the compiler doesn't optimize away repeated reads or writes.

2. **Disable interrupts temporarily in main code:**

```
__disable_irq();  // or cli() on AVR
shared_var++;
__enable_irq();   // or sei()
```

But use sparingly—blocking interrupts too long can cause missed events.

3. **Use atomic operations if supported:**
   - Some microcontrollers have atomic bit set/clear instructions.
   - Otherwise, disable interrupts just around the critical section.

4. **Double-buffering or circular buffers for data streams.**

Making Code Reentrant and Race-Condition Safe

Definitions:

- **Reentrant code:** Can be safely interrupted and called again (e.g., by an ISR) without corrupting state.
- **Race condition:** Occurs when two threads (or ISR and main) access shared data simultaneously and at least one writes to it.

Prevention Techniques:

- Avoid using static or global variables inside functions unless access is synchronized.
- Keep ISRs non-blocking — no printf(), dynamic memory allocation, or long loops.
- For multi-byte shared variables (like uint32_t on an 8-bit MCU), protect with critical sections:

```
uint32_t get_counter_safe(void) {
    uint32_t val;
    __disable_irq();
    val = counter;
    __enable_irq();
    return val;
}
```

Summary:

Concern	Solution
Shared variable inconsistency	volatile, critical sections
ISR affecting main state	Keep ISR short, use flags or buffers
Reentrancy	Avoid globals/static unless protected
Race conditions	Use atomic access or disable interrupts briefly

Uttam Basu

2) What is the memory layout of a C program in an embedded system (text, data, bss, heap, stack)? How can understanding this layout help in debugging stack overflows or memory corruption issues?

A typical memory layout looks like this (from low to high memory addresses):

Memory Segment	Description
Text (Code)	←Contains compiled program instructions (read-only)
Initialized Data	←.data section – Read/Write global variables that are initialized
Uninitialized Data	←.bss section – Read/Write global variables that are uninitialized (zeroed)
Heap	← Dynamic memory (allocated via malloc, free, etc.); grows upwards
(Free Space)	
Stack	←Local variables, function call frames; grows downwards

Section Breakdown:

Section	Description	Example
.text	Code section, constants, and read-only data	Function code, const data
.data	Initialized global/static variables	int x = 10;
.bss	Uninitialized global/static vars	static int count;
Heap	Used by malloc(), dynamic allocation	char *buf = malloc(100);
Stack	Used for function calls, local vars	int a = 5;, return addresses

Why It Matters — Debugging Memory Issues

1. Stack Overflow

- Stack grows downward, heap grows upward.
- If stack and heap collide, it causes corruption and weird crashes.
- Common in recursive functions, or deep call stacks with large local arrays.

Debug Tip:
Use a stack guard region, or fill the stack with a known pattern (0xAA) at startup and monitor it during runtime to detect overflow.

2. Heap Corruption

- Caused by buffer overruns, double free(), writing out-of-bounds, etc.
- Especially dangerous because heap and stack may be adjacent.
- Embedded systems may not have full MMU protection — so corruption silently kills.

Debug Tip:
Use custom memory allocators, or define a fixed-size heap to prevent overwrites. Static allocation is often preferred in embedded systems.

3. Incorrect Use of .bss and .data

- .bss must be zero-initialized at startup, typically done by startup code.
- If the startup code misses this, you'll see strange values in uninitialized globals.

Debug Tip:
Verify the linker script and startup code are correctly initializing memory sections.

4. Flash vs RAM Confusion

- .text and .rodata usually go in flash memory (non-volatile).
- .data, .bss, heap, stack go in RAM.
- If your microcontroller has limited RAM and you place big constants in .data instead of .rodata, you waste RAM.

Debug Tip:
Use const to keep data in flash:

```
const char msg[] = "Stored in flash";
```

Summary Table:

Memory Area	Read/Write	Volatile	Init at Start	Common Issues
`.text`	Read-only	No	Yes (from flash)	Can't be modified
`.data`	R/W	Yes	Yes	RAM usage issues
`.bss`	R/W	Yes	Zeroed	Corruption if not zeroed
`Heap`	R/W	Yes	No	Fragmentation, overflows
`Stack`	R/W	Yes	No	Overflow, collision with heap

3) Imagine you're writing a driver for a peripheral with strict timing requirements. How do you ensure deterministic behavior in your C code? Would you prefer a polling mechanism or an interrupt-driven one, and why?

In systems with **strict timing constraints**, your code must behave **predictably** and **within known time bounds** — always responding fast enough to prevent data loss or system failure.

How to Ensure Deterministic Behavior in C

1. **Minimize Jitter & Latency:**
    - Avoid variable-delay functions like `printf()`, floating-point operations, or loops with unpredictable exit times.
2. **Use Fixed-Cycle Code Paths:**
    - Write critical paths with **predictable execution time**.
    - Use tools like instruction cycle count analyzers or logic analyzers for measurement.
3. **Disable Unneeded Interrupts Temporarily:**
    - Prevent interrupt preemption during ultra-critical sections.
4. **Prioritize Code Efficiency:**
    - Prefer fixed-size buffers, static memory allocation, and optimized bit manipulation.
5. **Leverage Hardware Features:**
    - Use **DMA**, **timer peripherals**, and **hardware FIFOs** to offload the CPU and reduce latency.

6. **Use Compiler Attributes Carefully:**
    - Example: `__attribute__((always_inline))` or `__attribute__((section(".fastcode")))` to place time-critical routines in fast RAM.

Polling vs Interrupts – Which One?

Mechanism	Pros	Cons
Polling	Fully deterministic; simple to debug	Wastes CPU cycles; tight loops can block other tasks
Interrupts	Efficient CPU usage; fast response to events	Less predictable timing if interrupt nesting or latency occurs

Use Polling When:

- Timing is **ultra-critical and periodic**, e.g., bit-banging protocols like software UART or WS2812 LED driving.
- System is **simple** or **single-task**, with no RTOS or multitasking.

Use Interrupts When:

- Event-driven peripherals like UART, SPI, or timers.
- When CPU must handle multiple tasks concurrently.
- Timing is critical, **but not down to the microsecond level**.

Example:

Scenario: Reading data from a high-speed ADC every 1µs.

- **Best Approach:**
    - Configure a hardware **timer interrupt** to trigger every 1µs.
    - In ISR, read ADC data into a **pre-allocated buffer**.
    - Use **DMA** if available to remove CPU from the data path.
    - Offload processing to main loop or lower-priority thread.

Real-Time Scheduling Consideration (If RTOS is used):

- Use **high-priority real-time threads** for deterministic tasks.
- Avoid blocking calls in ISR or critical threads.
- Consider **priority inversion** risks and use mutexes with priority inheritance.

Summary:

Criteria	Recommendation
Ultra-high timing accuracy	Polling (short, tight loops)
CPU efficiency + responsiveness	Interrupt-driven
Complex systems / multitasking	Interrupts + buffer queues
Determinism needed in RTOS	High-priority thread or timer callback

4) You need to set, clear, and toggle individual bits in a memory-mapped hardware register. Write portable C macros for these operations. Bonus: How would you ensure they are optimized by the compiler?

Portable C Macros for Bit Operations:

These macros assume that `REG` is a pointer to a `volatile` memory-mapped hardware register (usually `uint32_t*` or `#define`d address).

```
// Set bit n in register
#define SET_BIT(REG, BIT)     ((REG) |= (1U << (BIT)))

// Clear bit n in register
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(1U << (BIT)))

// Toggle bit n in register
#define TOGGLE_BIT(REG, BIT)  ((REG) ^= (1U << (BIT)))

// Check if bit n is set
#define IS_BIT_SET(REG, BIT)  (((REG) >> (BIT)) & 1U)
```

Example Usage

```
#define GPIO_PORTA_OUT   (*(volatile uint32_t *)0x40020014)

void set_pin_high(void) {
    SET_BIT(GPIO_PORTA_OUT, 5);     // Set PA5
}

void clear_pin(void) {
    CLEAR_BIT(GPIO_PORTA_OUT, 5);   // Clear PA5
}
```

Note: Always use `volatile` when accessing hardware registers to prevent the compiler from optimizing away reads/writes.

Ensuring Compiler Optimization

To ensure these macros are efficient use `inline` Functions for Type-Safety (optional):

Some devs prefer inline functions over macros:

```
static inline void set_bit(volatile uint32_t *reg, uint8_t bit) {
    *reg |= (1U << bit);
}
```

Benefits:

- Type checking
- Debuggable in IDE
- Can be forced inline with `__attribute__((always_inline))`

Compiler Flags:

Use **optimization flags** like `-O2` or `-O3` during compilation to generate compact code (e.g., direct `ORR`, `AND` instructions on ARM Cortex-M).

Check Assembly (if you're hardcore):

Use `objdump`, or enable disassembly view in your IDE to ensure the compiler is not adding any unnecessary instructions.

Example (ARM):

```
LDR     r0, =0x40020014
LDR     r1, [r0]
ORR     r1, r1, #0x20    ; set bit 5
STR     r1, [r0]
```

Caution

- Never use these macros on read-only registers.
- Ensure atomicity if the register is accessed from **multiple contexts** (e.g., main + ISR).
- Use critical sections or bit-banding if available.

5) What are `inline` functions in Embedded C? Can you guarantee that a function declared as `inline` will be inlined by the compiler? Why or why not? What implications does this have in systems with limited flash or RAM?

An `inline` function is a hint to the compiler to **insert the function's code at each call site** instead of performing a traditional call (which involves pushing to stack, jumping, etc.).

Example:

```c
static inline uint8_t get_high_nibble(uint8_t val) {
    return (val >> 4) & 0x0F;
}
```

- Saves the overhead of a **function call** (stack push/pop, return address).
- Often used in **performance-critical** or **bit manipulation** code.

Can You Guarantee That a Function Will Be Inlined?

No, you **cannot** guarantee that a function marked `inline` will actually be inlined.

Why?

Because `inline` is just a **suggestion to the compiler**, not a command.

The compiler may **refuse to inline** if:

- The function is **too large**.
- It is called **too many times**.
- Inlining **increases code size too much** (important in embedded).
- Optimization flags do not favor inlining.

To **strongly suggest inlining**, you can use compiler-specific attributes:

GCC / Clang:

```c
static inline __attribute__((always_inline)) void fast_func(void) {
    // Critical code
}
```

Even then, the compiler **may still ignore it** if it violates constraints (like register pressure or size limits).

Implications in Embedded Systems (Flash vs RAM)

Advantages of Inlining

- **Speed boost**: Removes function call overhead, especially important in tight loops or ISRs.
- **Cleaner register usage**: Reduces push/pop and return overhead.
- **Better optimization**: The compiler can optimize surrounding code better when inlined.

Disadvantages

- **Code size increases**: Each inlined instance adds code, which can lead to **flash overflow** (very bad in small MCUs).
- **Instruction cache pressure** (on higher-end MCUs).
- May **increase RAM usage** indirectly if it causes the compiler to spill more registers to the stack.

Best Practices for Embedded Systems:

Do	Don't
Use inline for short, critical functions	Inline large functions – may bloat flash
Use static inline in headers for performance	Expect guaranteed inlining
Use compiler attributes carefully (always_inline)	Overuse inline assuming it always helps
Profile/code size check after inlining	Ignore memory constraints

6) What happens from the moment power is applied to an embedded system until `main()` is called? Describe the role of the startup file, vector table, and initialization of sections.

Step-by-Step: From Power-On to `main()`

1. Power-On Reset (POR) or Reset Event

- As soon as power is applied (or a reset button is pressed), the microcontroller's **hardware reset circuitry** activates.

- The **Program Counter (PC)** is set to a specific location, typically address **0x00000000**, or wherever the **reset vector** is defined.

2. Vector Table Lookup

- The **vector table** is a table of function pointers located at a fixed memory address (often `0x00000000` in most MCUs).
- The first two entries are:
  - **Initial Stack Pointer (SP)** – points to top of the stack.
  - **Reset Handler (Reset_ISR)** – the function to execute on reset.

Example (ARM Cortex-M vector table):

```
__attribute__((section(".isr_vector")))
const uint32_t VectorTable[] = {
    (uint32_t)&_estack,       // Initial SP value
    (uint32_t)&Reset_Handler, // Reset handler
    ...
};
```

3. Reset Handler (Defined in Startup File)

- The **startup file** (usually written in assembly or C) defines `Reset_Handler`.
- This is where **initialization** happens before calling `main()`.

4. Initialization of Memory Sections

The startup code performs **memory setup**:

Copy `.data` from Flash to RAM:

- Initialized global/static variables are stored in **Flash** but used from **RAM**.

```
for (p = _sdata; p < _edata; p++) {
    *p = *_sidata; // Copy from flash to RAM
```

Zero out `.bss` section:

- Uninitialized global/static variables should be zeroed.

```
for (p = _sbss; p < _ebss; p++) {
    *p = 0;
}
```

These symbols (`_sdata`, `_edata`, etc.) are defined in the **linker script**.

5. Optional System Initialization

- Sometimes, early **hardware-specific setup** occurs:
    - Clock initialization
    - Watchdog disable
    - Floating Point Unit (FPU) enable
    - Cache configuration
    - Relocating vector table (if needed)

6. Call `main()`

Finally, after everything is ready:

```
int main(void);
```

Control is transferred to your application. This is where your firmware truly begins.

Recap Diagram

```
[ Power On / Reset ]
        ↓
[ Vector Table @ 0x00000000 ]
        ↓
[ Reset_Handler in startup file ]
        ↓
[ Initialize .data and .bss sections ]
        ↓
[ Optional hardware init (e.g. clocks) ]
        ↓
[ Call main() ]
```

Why Is This Important?

Understanding this helps with:

- **Debugging boot-time issues**
- Ensuring **global variables are valid before use**
- **Writing a custom bootloader**
- Knowing where to place critical code (e.g. startup diagnostics)

Uttam Basu

7) How would you configure a microcontroller timer to generate an interrupt every 1 millisecond? What are the critical calculations you need, and how do clock frequency and prescaler factor in?

Generate a timer interrupt every 1 ms

Step 1: Understand the Timer's Input Clock

The timer is usually fed by a system clock (f<sub>clk</sub>), often from:

- Internal oscillator
- PLL output
- External crystal

Let's say:

```
f_clk = 72 MHz    // system clock driving the timer
```

Step 2: Choose a Prescaler

The prescaler divides the system clock to slow down the timer counter.

```
timer_clk = f_clk / prescaler
```

You want to choose the prescaler and timer period so that the timer interrupt happens every 1 ms.

Step 3: Calculate Timer Reload Value (ARR / TOP)

If we want a 1 ms timer interrupt:

```
period = 1 ms = 0.001 sec
Count_required = timer_clk * 0.001
```

Assume:

- prescaler = 72
- Then timer_clk = 72 MHz / 72 = 1 MHz
- So:

```
Count_required = 1 MHz * 0.001 = 1000 ticks
```

So you want the timer to count 1000 ticks, then overflow and trigger an interrupt.

Final Formula

```
ARR (Auto Reload Register) = (f_clk / prescaler) * desired_time_interval
- 1
```

Example:

```
f_clk = 72,000,000 Hz
prescaler = 72  → timer_clk = 1,000,000 Hz (1 tick = 1 µs)
desired interval = 1 ms = 1000 µs

ARR = (1,000,000 * 0.001) - 1 = 999
```

So set:

```
PRESCALER = 71  // (72 - 1) since prescaler is zero-based
ARR = 999
```

Timer will overflow every 1000 µs = 1 ms

Step 4: Configuration (e.g., STM32-style Pseudocode)

```
TIMx->PSC = 71;       // Prescaler
TIMx->ARR = 999;      // Auto-reload
TIMx->DIER |= TIM_DIER_UIE;  // Enable update interrupt
TIMx->CR1 |= TIM_CR1_CEN;    // Enable counter
NVIC_EnableIRQ(TIMx_IRQn);   // Enable IRQ in NVIC
```

Critical Considerations

Parameter	Why It Matters
Clock frequency	Drives all timer calculations
Prescaler	Affects resolution and max period
ARR value	Defines overflow period
Timer width	8-bit, 16-bit, 32-bit defines max range
ISR efficiency	Must finish before next interrupt to avoid overrun