150 Firmware & Embedded Systems Coding Interview Questions

A curated list of 150 coding questions grouped by topic and difficulty level for firmware and embedded systems interviews at NVIDIA, Qualcomm, TI, Google, and Microsoft.

Table of Contents

Bit Manipulation

Easy

1. **Check Even or Odd** – Determine if an integer is even or odd using bitwise operations (e.g., `n & 1` check).
2. **Get i-th Bit** – Return the value of the bit at position i (0-indexed from LSB) of an integer (use shift and mask).
3. **Set i-th Bit** – Set the bit at position i of an integer to 1 (use `OR` with appropriate mask).
4. **Clear i-th Bit** – Clear the bit at position i of an integer to 0 (use `AND` with inverted mask).
5. **Toggle i-th Bit** – Flip the bit at position i of an integer (use `XOR` with mask).
6. **Power of Two Check** – Check if an integer is a power of two (and >0) by verifying only one bit is set (e.g., `n & (n-1) == 0`).
7. **Count Set Bits** – Count the number of 1-bits in an integer's binary representation (Brian Kernighan's algorithm) – *LeetCode 191: Number of 1 Bits*.
8. **Parity of Number** – Compute the parity (even or odd count of 1-bits) of a 32-bit integer (use XOR fold or bit-mask tricks).
9. **Swap Two Numbers (XOR)** – Swap two integer variables without using a temporary variable, utilizing XOR operations.

Medium

1. **Reverse Bits** – Reverse the 32-bit binary representation of a number (e.g., `43261596` becomes `964176192`) – *LeetCode 190: Reverse Bits*.
2. **Hamming Distance** – Calculate the number of differing bits between two integers (count bits in `x ^ y`) – *LeetCode 461: Hamming Distance*.
3. **Add Without Plus** – Add two integers without using the `+` operator, using bitwise XOR (sum) and AND (carry) in a loop – *LeetCode 371: Sum of Two Integers*.
4. **Bitwise Multiply** – Multiply two integers using bit shifting and addition (simulate long multiplication in binary).
5. **Bit Flips to Convert** – Given two integers, find how many bits are different (bits to flip to convert one into the other).
6. **Swap Odd and Even Bits** – Swap all odd-position bits with even-position bits in an integer (e.g., bit0 with bit1, bit2 with bit3, etc.).
7. **Single Number** – In an array where every other number appears twice, find the unique number using XOR – *LeetCode 136: Single Number*.
8. **Next Power of 2** – Given a positive integer, find the next highest power of two (e.g., for 5 output 8).

Hard

1. **Divide without Division** – Implement integer division of two numbers without using the `/` operator (use repeated subtraction or bit shifts for efficiency).
2. **Insert Bits** – Given two 32-bit numbers `N` and `M`, and bit positions `i` to `j`, insert `M` into `N` such that `M` occupies bits `i...j` (clear `i..j` in `N` and OR with `M` shifted).
3. **Next Number with Same 1s** – Given an integer, find the next larger number with the same number of 1-bits in its binary representation (bit manipulation puzzle involving rightmost non-trailing one).
4. **Single Number II** – In an array where every number appears three times except one appears once, find that single one (use bit counting mod 3) – *LeetCode 137: Single Number II*.
5. **Single Number III** – In an array where every number appears twice except two numbers appear once each, find those two non-repeating numbers (use bit partitioning by set/unset bit) – *LeetCode 260*.

Pointers & Pointer Arithmetic

Easy

1. **Implement `strlen`** – Write `size_t strlen(const char *s)` to find the length of a C-string (iterate pointer until `'\0'`).
2. **Implement `strcpy`** – Write `char* strcpy(char *dest, const char *src)` to copy a C-string (character by character copy until null terminator).
3. **Implement `strcmp`** – Write `int strcmp(const char *a, const char`

`*b)` to compare two strings lexicographically (iterate and compare characters).
4. **Swap via Pointers** – Write a function `void swap(int *a, int *b)` that swaps the values of two integers using pointer references.
5. **Function Pointer Declaration** – Declare and use a function pointer for a function that, say, returns int and takes two ints (test knowledge of syntax: `int (*fp)(int,int) = ...;`).

Medium

1. **Implement `strchr`** – Write `char* strchr(const char *s, char c)` to find the first occurrence of a character in a string (return pointer into the string or `NULL`).
2. **Implement `strcat`** – Write `char* strcat(char *dest, const char *src)` to concatenate two strings (append source to end of dest).
3. **2D Array via 1D Pointer** – Given a flat array and dimensions `rows` and `cols`, implement `T get(array, r,c)` and `set(array, r,c,value)` using pointer arithmetic (compute index as `r*cols + c`).
4. **XOR Linked List** – Implement a memory-efficient doubly-linked list where each node stores `ptrXOR = prev ^ next` instead of separate pointers. Write functions to insert and traverse this XOR linked list.

Hard

1. **Sort Strings (Function Pointers)** – Given an array of C-string pointers, sort them lexicographically using a custom comparison function (implement or use `qsort` with a comparator function pointer).
2. **Function Pointer Callback Mechanism** – Design a callback registry: e.g., implement `register_callback(void (*cb)(void))` and later invoke all registered callbacks. (Tests understanding of storing and calling function pointers.)

Memory Management

Easy

1. **Implement `memset`** – Write `void *memset(void *dest, int value, size_t n)` to set a block of memory to a byte value (iterate over bytes setting each to the given value).
2. **Implement `calloc`** – Write a `void* calloc(size_t count, size_t size)` that allocates zero-initialized memory (allocate `count*size` bytes and use memset to 0).
3. **Implement `strdup`** – Write a function that duplicates a string by allocating new memory and copying the content (uses `malloc` and `strcpy`).

Medium

1. **Implement `memcpy`** – Write `void* memcpy(void *dest, const void *src, size_t n)` to copy memory from source to destination (copy byte-by-byte). *(Assume regions do not overlap.)*
2. **Implement `memmove`** – Write a safer `memcpy` that handles overlapping source and destination regions correctly (copy forward or backward depending on overlap).
3. **Resize (Realloc)** – Implement a function `void* resize(void *ptr, size_t oldSize, size_t newSize)` that reallocates a memory block to a new size (allocate new block, `memcpy` old data, free old block).

Hard

1. **Aligned Malloc/Free** – Implement `void* aligned_malloc(size_t size, size_t alignment)` and corresponding `aligned_free(void* ptr)` so that the returned memory address is a multiple of the given power-of-two alignment. (Hint: you may need to store the original pointer for freeing.)
2. **Custom Memory Allocator** – Using a fixed-size buffer (e.g., a char array), implement `my_malloc` and `my_free`. Manage the free list within the buffer (e.g., using a simple first-fit allocation strategy).

Linked Lists

Easy

1. **Reverse Singly Linked List** – Reverse a singly linked list in-place (iterative pointer manipulation) – *LeetCode 206: Reverse Linked List.*
2. **Reverse Doubly Linked List** – Given a doubly-linked list, reverse it by swapping the next and prev pointers of all nodes (iterate and swap).
3. **Merge Two Sorted Lists** – Merge two sorted singly linked lists into one sorted list (splicing nodes into a new list) – *LeetCode 21: Merge Two Sorted Lists.*
4. **Find Middle of List** – Use two-pointer technique (slow/fast) to find the middle node of a singly linked list in one pass.
5. **Remove Duplicates (Sorted List)** – Given a sorted linked list, remove all duplicate values (skip over nodes with the same value) – *LeetCode 83: Remove Duplicates from Sorted List.*

Medium

1. **Detect Cycle in List** – Determine if a linked list has a loop/cycle using Floyd's cycle-finding algorithm (tortoise and hare) – *LeetCode 141: Linked List Cycle.*
2. **Find Cycle Start** – If a loop is present, find the node where the cycle begins (detect cycle then use two-pointer meeting point method) – *LeetCode*

142: Linked List Cycle II.

3. **Delete Node without Head** – Given only a pointer to a node (not the head) in a singly linked list, delete that node in-place (copy data from next node and unlink next).
4. **Remove Nth from End** – Remove the N-th node from the end of the list in one pass (use two pointers offset by N) – *LeetCode 19: Remove Nth Node from End.*
5. **Swap Two Nodes** – Swap two nodes in a linked list without swapping data (rearrange pointers). For example, given keys x and y, swap their node positions in the list.
6. **Odd-Even List** – Reorder a linked list such that all nodes at odd indices come first followed by nodes at even indices (preserve relative order) – *LeetCode 328: Odd Even Linked List.*
7. **Swap Nodes in Pairs** – Swap every two adjacent nodes in the list – *LeetCode 24: Swap Nodes in Pairs.*
8. **Intersection of Two Lists** – Given two singly linked lists that converge (Y-shaped), find the intersection starting node (use two-pointer technique cycling at end) – *LeetCode 160: Intersection of Linked Lists.*
9. **Check Palindrome List** – Check if a singly linked list is a palindrome (by finding middle, reversing second half, and comparing halves).

Hard

1. **Sort Linked List** – Sort a linked list in O(n log n) time (often via Merge Sort on list, which requires finding mid and merging sorted halves) – *LeetCode 148: Sort List.*
2. **Reverse Nodes in k-Group** – Given a linked list, reverse every k nodes (group), leaving the remainder as is – *LeetCode 25: Reverse Nodes in k-Group.*
3. **Flatten Multilevel List** – Given a linked list where nodes may have a child pointer to a subordinate list, flatten it so that all nodes appear in a single-level singly linked list (depth-first traversal).
4. **Add Two Numbers** – Treat two linked lists as numbers (digits stored in nodes, least significant first) and return the sum as a linked list – *LeetCode 2: Add Two Numbers.*

Arrays & Strings

Easy

1. **Reverse Array** – Reverse the elements of an array in place (two-pointer swap from ends moving inward).
2. **Find Min and Max** – Traverse an array once to find the minimum and maximum values.
3. **Binary Search** – Given a sorted array, implement binary search to find the index of a target (or -1 if not found).

4. **Remove Duplicates (Sorted Array)** – Remove duplicates in-place from a sorted array and return new length (shift unique elements forward) – *LeetCode 26: Remove Duplicates from Sorted Array.*
5. **Move Zeros to End** – Given an array, move all zero values to the end while maintaining the order of non-zeros – *LeetCode 283: Move Zeroes.*
6. **Check Palindrome (String)** – Check if a string reads the same forwards and backwards (two-pointer technique, case or non-alphanumeric handling as needed).
7. **First Unique Character** – Find the first non-repeating character in a string (use frequency count) – *LeetCode 387: First Unique Character in a String.*
8. **Check Anagrams** – Determine if two strings are anagrams of each other (sort and compare, or use frequency counting) – *LeetCode 242: Valid Anagram.*
9. **Plus One** – Given an array of digits representing a non-negative integer, add one to the number (handle carry propagation) – *LeetCode 66: Plus One.*
10. **Missing Number** – Given an array of size N with numbers 0..N, one number is missing; find it (use XOR of all indices and values or Gauss sum) – *LeetCode 268: Missing Number.*
11. **Merge Sorted Arrays** – Given two sorted arrays, where the first has enough extra space at the end, merge the second into the first in sorted order – *LeetCode 88: Merge Sorted Array.*

Medium

1. **Two Sum** – Find two numbers in an array that add up to a target sum (use hashing or two-pointer if array is sorted) – *LeetCode 1: Two Sum.*
2. **Rotate Array** – Rotate an array to the right by k steps in-place (cyclically bring end elements to front) – *LeetCode 189: Rotate Array.*
3. **Maximum Subarray** – Find the contiguous subarray with the largest sum (Kadane's algorithm for maximum subarray sum) – *LeetCode 53: Maximum Subarray.*
4. **Implement `atoi`** – Convert a string to an integer, handling optional whitespace, sign, overflow, etc. – *LeetCode 8: String to Integer (atoi).*
5. **Reverse Words in Sentence** – Given a string sentence, reverse the order of words in-place (or using O(n) extra space): e.g., `"I am here"` → `"here am I"`.
6. **Implement `strstr`** – Implement substring search to find the first occurrence of a pattern in a text (naive O(n*m) or KMP for bonus)* – LeetCode 28: Implement strStr()*.
7. **Longest Common Prefix** – Given an array of strings, find the longest common prefix among them – *LeetCode 14: Longest Common Prefix.*
8. **Find Duplicate Number** – In an array of n+1 integers where each is in [1, n] range, find the one duplicate (Floyd's cycle detection in array) – *LeetCode 287: Find the Duplicate Number.*

9. **Check String Rotation** – Check if one string is a rotation of another (e.g., “abcdef” and “cdefab”) by checking if `s2` is substring of `s1+s1`.
10. **Spiral Matrix Print** – Given a 2D matrix, print the elements in spiral order (simulate layer-by-layer traversal) – *LeetCode 54: Spiral Matrix.*
11. **Sort Colors (Dutch National Flag)** – Sort an array of 0s, 1s, and 2s in-place (three-way partitioning) – *LeetCode 75: Sort Colors.*
12. **Majority Element** – Given an array, find the element that appears more than n/2 times (Boyer–Moore voting algorithm) – *LeetCode 169: Majority Element.*

Hard

1. **Next Permutation** – Given an array (or string) of numbers, rearrange them to the next lexicographic permutation (or lowest if none) – *LeetCode 31: Next Permutation.*
2. **First Missing Positive** – Given an unsorted array, find the smallest positive integer not present in the array (solve in O(n) using index marking) – *LeetCode 41: First Missing Positive.*
3. **Search in Rotated Array** – Given a sorted array that’s been rotated, search for a target value in O(log n) time (modified binary search) – *LeetCode 33: Search in Rotated Sorted Array.*
4. **Rotate Matrix 90°** – Given an N×N matrix of integers, rotate it 90 degrees clockwise in-place (transpose and then reverse each row) – *LeetCode 48: Rotate Image.*

Stacks, Queues & Circular Buffers

Easy

1. **Implement Stack (Array)** – Create a stack using an array with `push`, `pop`, `peek` operations (LIFO behavior, track top index).
2. **Implement Stack (Linked List)** – Create a stack using a singly linked list (push/pop at the head of the list).
3. **Implement Queue (Array)** – Create a circular queue using an array with `enqueue` and `dequeue` (use head/tail indices mod capacity).
4. **Implement Queue (Linked List)** – Create a FIFO queue using a linked list (enqueue at tail, dequeue from head).

Medium

1. **Implement Circular Buffer** – Design a fixed-size circular buffer with read and write indices (support wrap-around for `enqueue/dequeue` operations).
2. **Two Stacks in One Array** – Implement two stack data structures using a single array (one grows from left end, other from right end, to optimize space).
3. **Queue using Two Stacks** – Implement a queue using two stacks (one stack for enqueue, one for dequeue; amortized O(1) operations).

4. **Stack using Two Queues** – Implement a stack using two queues (push or pop operations become costly transfers between queues).
5. **Min Stack** – Design a stack that, in addition to push/pop, can return the minimum element in O(1) time (maintain auxiliary stack of mins) – *LeetCode 155: Min Stack*.
6. **Valid Parentheses** – Given a string containing brackets `()[]{}`, determine if the sequence is balanced using a stack – *LeetCode 20: Valid Parentheses*.
7. **Next Greater Element** – For each element in an array, find the next greater element to its right (use a stack to efficiently process this) – *LeetCode 496: Next Greater Element I*.

Hard

1. **Sort Stack** – Given a stack, sort its elements (e.g., using one additional stack as auxiliary storage) – no recursion, only stack operations allowed.
2. **Sliding Window Maximum** – Given an array and window size k, find the maximum in each sliding window of size k (use deque to achieve O(n) solution) – *LeetCode 239: Sliding Window Maximum*.
3. **Evaluate Reverse Polish Notation** – Evaluate the value of an arithmetic expression in Reverse Polish (Postfix) notation using a stack – *LeetCode 150: Evaluate Reverse Polish Notation*.

Function Pointers & Callbacks (C/C++)

Easy

1. **Function Pointer Basic** – Define a function pointer and use it to call a function (e.g., a pointer to a function that takes int and returns int).
2. **Callback Invocation** – Write a function `repeat(void (*fn)(), int n)` that takes a function pointer and calls that function n times (demonstrate basic callback usage).
3. **Array of Function Pointers** – Create an array of function pointers (for example, an array of operations like add, sub, mul) and use it to invoke different behaviors based on an index.

Medium

1. **Comparator Function (qsort)** – Given an array of structs (e.g., `{id, name}`), implement a comparison function to sort by `id` and use C's `qsort` to sort the array (demonstrates using function pointer as callback for sorting).
2. **Interrupt Vector Table** – Simulate an interrupt vector table using an array of function pointers. Implement `register_interrupt(int idx, void (*handler)())` to store handlers and `handle_interrupt(idx)` to invoke the handler for a given index (if set).

3. **State Machine via Function Pointers** – Implement a simple state machine in C. For example, have an enum of states and an array of function pointers `state_handlers[]` where each function implements the state's behavior. Show transitioning by calling a different function pointer.

Hard

1. **Polymorphism in C (Function Pointers in Struct)** – Use function pointers within a struct to simulate polymorphic behavior. For example, define a `struct Shape` with a function pointer `area()`; implement specific shapes (Circle, Rectangle) with their own `area` functions and demonstrate calling through the base struct pointer.

Structs, Bit-fields & Endianness

Easy

1. **Endianness Check** – Write a function to check if the system is little-endian or big-endian (e.g., store 1 in an `int` and inspect the first byte via a `char*` cast).
2. **Byte Swap (32-bit)** – Given a 32-bit integer, swap its byte order (convert little-endian to big-endian or vice versa) by shifting and masking bytes (e.g., swap endianness of 0xAABBCCDD to 0xDDCCBBAA).

Medium

1. **Bit-field Struct Definition** – Define a `struct` with bit-fields to pack multiple values into a single byte or word. *Example:* create a struct for an 8-bit register where bit0 is a flag, bits1-3 are a code, and bits4-7 are a count.
2. **Extract Packed Fields** – Given a packed 32-bit value (e.g., an RGB pixel with 8 bits for R, G, B, and A), extract each component by masking and shifting.
3. **Unaligned 32-bit Read** – Implement a function to read a 32-bit integer from a buffer of bytes that may not be aligned (assemble `buf[0] | buf[1]<<8 | buf[2]<<16 | buf[3]<<24` taking into account endianness as needed).

Hard

1. **Float Bit Analysis** – Using a union or pointer casting, interpret a 32-bit float's binary representation. Extract the sign, exponent, and mantissa fields (IEEE-754) and print them. (Demonstrates understanding of bit-level layout of floating point numbers.)
2. **CRC Calculation** – Compute a CRC-32 checksum for a byte array (implement the polynomial division algorithm bit-by-bit or byte-wise).

(This tests understanding of bitwise operations and is often encountered in firmware for data integrity.)

Binary Trees

Easy

1. **Tree Traversal (Inorder)** – Implement inorder traversal of a binary tree (left-root-right) using recursion.
2. **Tree Height** – Compute the height (max depth) of a binary tree (using recursion to find max of left/right subtree heights + 1).
3. **Invert Binary Tree** – Mirror a binary tree by swapping left and right children recursively – *LeetCode 226: Invert Binary Tree*.

Medium

1. **Validate BST** – Given a binary tree, determine if it is a binary search tree (BST) (check inorder is sorted or use min/max constraints in recursion) – *LeetCode 98: Validate Binary Search Tree*.
2. **Level Order Traversal** – Perform level-order (breadth-first) traversal of a binary tree (use a queue to print nodes level by level) – *LeetCode 102: Binary Tree Level Order Traversal*.
3. **Left View of Tree** – Print the left view of a binary tree (the first node at each level when viewed from the left side).
4. **Lowest Common Ancestor** – Find the lowest common ancestor of two nodes in a binary tree (if tree is a BST, can use BST properties; otherwise, use recursion to find split point) – *LeetCode 236: Lowest Common Ancestor of a Binary Tree*.
5. **Symmetric Tree Check** – Check if a binary tree is symmetric (a mirror of itself around center) by comparing left and right subtrees – *LeetCode 101: Symmetric Tree*.

Hard

1. **Serialize/Deserialize Tree** – Convert a binary tree to a serialized format (string or array) and vice versa (e.g., level-order with null markers) – *LeetCode 297: Serialize and Deserialize Binary Tree*.
2. **Delete Node in BST** – Delete a given value from a BST and maintain BST properties (handle cases: leaf, one child, two children – replace with inorder successor) – *LeetCode 450: Delete Node in a BST*.
3. **Diameter of Tree** – Compute the diameter of a binary tree (length of the longest path between any two nodes, which may or may not pass through root) – *LeetCode 543: Diameter of Binary Tree*.
4. **Kth Smallest in BST** – Find the k-th smallest element in a BST (inorder traversal until k-th element) – *LeetCode 230: Kth Smallest Element in a BST*.

5. **Distance Between Nodes** – Given a binary tree and two node values, find the distance (number of edges) between them. (Find LCA, then distance = dist(root,a)+dist(root,b)-2*dist(root,LCA)).

OS Concepts & Concurrency

Easy

1. **Thread-safe Counter** – Implement a simple thread-safe counter class with `increment()` and `get()` methods (use a mutex lock or atomic variable to prevent race conditions).
2. **Producer-Consumer (Bounded Buffer)** – Illustrate a producer-consumer problem solution: use a fixed-size circular buffer with two threads, one producing and one consuming, synchronized with a mutex and two semaphores (for "empty" and "full" count). *(Pseudocode for `wait()`/`signal()` calls is acceptable.)*

Medium

1. **Spinlock** – Implement a basic spinlock in C: use an atomic flag that one thread sets to 1 to acquire the lock and others loop (spin) until it becomes 0 to acquire. Provide `lock()` and `unlock()` functions.
2. **Reader-Writer Lock (Conceptual)** – Sketch an implementation of a reader-writer lock using mutexes/condition variables or semaphores (multiple readers can hold if no writers, writer has exclusive access).

Hard

1. **Dining Philosophers (Semaphore Solution)** – Describe the semaphore-based solution for the Dining Philosophers problem (allocate 5 philosophers, 5 chopstick semaphores, prevent deadlock by ordering resource acquisition or using an additional waiter semaphore).
2. **Lock-Free Stack (Atomic)** – Implement a lock-free stack push and pop using atomic compare-and-swap (CAS). (Use a loop to atomically update the head pointer.)

This document contains coding problems that reflect common patterns in embedded firmware interviews at companies like NVIDIA, Qualcomm, TI, Google, and Microsoft. They cover low-level coding skills in C/C++ – from bit manipulation and pointer arithmetic to memory management, data structures, and concurrency – ensuring a well-rounded preparation for firmware and embedded systems interviews.