LAB TASK 11

Task # 01:

Your team is developing a low-level text-processing module for an embedded system that receives raw sensor log strings containing both valid characters and noise symbols such as '#'. The system requires a procedure that scans the entire log buffer to remove all '#' characters and copy only clean characters into a new output buffer. You are asked to implement this procedure using string instructions, specifically LODSB, STOSB, CMPSB, and REP prefixes, instead of manual indexing. Write an assembly routine (MASM syntax) that takes pointers to the input and output buffers and processes each byte using MOVSB-style logic, filtering out unwanted characters while preserving the order of valid data.

Task # 02:

A graphics driver module in your software initializes multiple pixel rows with a repeated color pattern before rendering. To optimize performance, the lead developer wants the initialization procedure implemented using STOSB/STOSW and REP instructions instead of loop-based stores. You must write a MASM procedure that accepts a destination pointer and a repeat count, then fills the entire buffer with a specified 8-bit or 16-bit color value using REP STOSB or REP STOSW depending on a mode flag. The case study requires you to correctly set up DF, AL/AX, and EDI so the buffer is filled efficiently without manual loops.

Task # 03:

A small embedded logging system stores event messages in a fixed-size memory region, but each message must first be processed by a utility routine that determines the message length and then copies it into a separate log buffer. Your task is to write a MASM procedure that performs both operations: first use LODSB with REPNE SCASB or manual scanning to compute the string length until the null terminator is found, and then use MOVSB with the REP prefix to copy exactly that many bytes to the destination buffer. The procedure must avoid manual index arithmetic and rely entirely on string instructions while correctly handling the direction flag and pointer registers.

Task # 04:

A scientific simulation program stores temperature readings from multiple sensors in a dynamically sized 2D array (rows = sensors, columns = hourly readings), and you are asked to design an Irvine32 MASM procedure that performs three operations on this array: (1) Find the maximum temperature in the entire matrix, (2) Compute the average temperature for each sensor row, and (3) Normalize the matrix by subtracting the global maximum from every element. The routine must receive the base address of the array, the number of rows, and the number of columns, and then use nested traversal with row-major addressing ((row * cols + col) * 4) to access each element through indirect addressing. You must store the per-row averages in a separate output array, update the original matrix in place, and print both the global maximum and each row's average using Irvine32's WriteInt and WriteString. The solution requires careful register management, saving and restoring registers with PUSH/POP, using conditional jumps to create loop structures, and ensuring correct pointer movement for multi-level traversal without relying on high-level constructs.

Task # 05:

A warehouse-tracking subsystem stores product quantities in an unsorted integer list that is allocated at runtime, and your task is to write an Irvine32 MASM procedure that sorts this list in ascending order using the Bubble Sort algorithm. The procedure must accept three parameters: the base address of the array, the number of elements, and a flag that indicates whether the sorted output should also be displayed on the screen. You are required to implement the nested Bubble Sort loops manually using registers and conditional jumps (no high-level constructs), and use indirect addressing ([esi], [esi+4]) to compare adjacent elements. Your algorithm must swap values correctly using temporary registers, repeatedly pass through the array until no more swaps occur, and update ESI accordingly for each comparison. If the flag indicates printing, the sorted elements must be displayed using Irvine32's WriteInt, WriteString, and a newline after each value. The routine must preserve registers properly using PUSH/POP and must handle arrays of variable length.