Academic Report on the Design and Implementation of a Console-Based Bill Management System (BMS)

Prepared by: ASHWIN KUMAR
Prepared for: PROGRAMING IN C (Dr Dolly Das)

A Comprehensive Analysis of Software Structure and Workflow

Project Description: This report documents the design, structure, and functional logic of a simple C-language application developed for managing retail or small business billing operations, including item entry, subtotal calculation, application of discounts, tax assessment, and data persistence via file storage.

Abstract

This report details the design and implementation of a console-based Bill Management System (BMS) developed in the C programming language. The primary objective of the BMS is to provide a streamlined, command-line utility for managing transaction records, specifically focusing on data input, financial calculation, and structured output. The system utilizes fundamental C structures ('Item' and 'Bill') to encapsulate transactional data, promoting modularity and code clarity.

The core functionalities include adding individual items (with name, price, and quantity), displaying the current running total, calculating the final bill (incorporating user-defined discounts and a fixed 18% GST/Tax rate), and persisting the generated bill details to a plain text file. The system's architecture is event-driven, operating via a continuous menu loop that directs control flow to specialized functions. A critical focus of this analysis is the detailed workflow of each function, which is comprehensively explained through descriptive text and accompanying flowcharts. These flowcharts visually map the logic, decision points, and I/O operations, ensuring a clear understanding of the system's operational sequence and robustness. The BMS serves as an effective demonstration of structured programming principles, data structure utilization, and basic file handling in C.

1 System Overview and Main Logic

1.1 Introduction

The Bill Management System (BMS) is a foundational application designed to automate the process of generating a sales invoice. Developed using ANSI C, the program emphasizes efficiency and reliability in handling transaction data. This report will dissect the system's architecture, analyzing its structural components and providing an in-depth, function-by-function breakdown of its logical flow, essential for understanding the software's operation.

1.2 System Architecture and Data Structures

The system relies on two primary data structures defined in the source code:

1. **'Item' Structure:** Stores attributes of a single product: 'name' (string), 'price' (float), 'quantity' (integer), and 'total' (calculated float).
2. **'Bill' Structure:** Acts as the central data container, holding an array of up to 50 'Item' structures ('items'), the 'itemCount', and all financial summaries ('subtotal', 'tax', 'discount', 'grandTotal'), along with the transaction 'date'.

1.3 Main Program Flowchart Analysis

The 'main' function initializes the 'Bill' structure and manages the user interface (UI) through a continuous loop. This section outlines the master control flow of the entire application.

Flowchart Description: Main Program Logic

The 'main' function starts by initializing all bill fields (item count, totals) to zero. It then enters a `while(1)` loop, presenting the user with the main menu options (1-5). It takes the user's input (`choice`) and uses a `switch` statement to direct execution. Valid choices execute their respective functions (`addItem`, `displayBill`, `calculateBill`, `saveBillToFile`). An invalid choice displays an error. If the user selects option 5, the loop is exited, a thank-you message is printed, and the program terminates using `exit(0)`.

2 Detailed Functional Flowcharts: Data Input and Display

2.1 Item Addition (`addItem` Function Flowchart)

The `addItem` function is responsible for gathering and storing transaction data for a single product. It incorporates an essential data integrity check to ensure the system does not overflow its defined capacity (MAX_ITEMS $= 50$).

Flowchart Description: `addItem` **Logic**

1. **Start.**
2. **Decision Node:** Is `itemCount` $\geq$ MAX_ITEMS?
   - **Yes:** Print "Bill is full!" message and **Return**. (Prevents Buffer Overflow).
   - **No:** Proceed.
3. **Input:** Prompt user and read `item name`.
4. **Input:** Prompt user and read `price` (float).
5. **Input:** Prompt user and read `quantity` (integer).
6. **Process:** Calculate item total:

$$\text{Total} = \text{Price} \times \text{Quantity}.$$

7. **Storage:** Store the newly created `Item` object into the `bill->items` array at the `itemCount` index.
8. **Process:** Increment `bill->itemCount`.
9. **Output:** Print "Item added successfully!" message.
10. **End.**

2.2 Bill Display (`displayBill` Function Flowchart)

The `displayBill` function is a read-only process used to provide the user with a real-time summary of the items added so far, without calculating the final financial totals (discount/tax).

Flowchart Description: `displayBill` **Logic**

1. **Start.**
2. **Decision Node:** Is `itemCount` == 0?
   - **Yes:** Print "No items in the bill yet!" message and **Return**.
   - **No:** Proceed.
3. **Output:** Print the bill header, column titles ("Item Name", "Price", "Quantity", "Total"), and separation lines.
4. **Loop Initialization:** Initialize counter $i = 0$.
5. **Loop Condition:** Is $i <$ itemCount?
   - **No:** Exit loop.
   - **Yes:** Proceed.

6. **Output:** Print the details (Name, Price, Quantity, Total) for bill->items$[i]$.
7. **Process:** Increment i. (Go back to Loop Condition).
8. **Output:** Print the footer separation line.
9. **End.**

3 Detailed Functional Flowchart: Financial Calculation

3.1 Final Calculation (`calculateBill` Function Flowchart)

The `calculateBill` function is the most complex module, handling all financial logic: subtotal aggregation, dynamic discount application, fixed tax calculation, and generation of the final payable amount (`grandTotal`). It also captures the transaction date.

Flowchart Description: `calculateBill` Logic

1. **Start.**
2. **Decision Node:** Is `itemCount` == 0?
   - **Yes:** Print "No items to calculate!" message and **Return**.
   - **No:** Proceed.
3. **Process: Subtotal Calculation:** Initialize `subtotal` to 0. Loop through all items and sum their individual `total` fields into `subtotal`.
4. **Input: Discount:** Prompt user and read `discountPercent` (float).
5. **Process: Discount Calculation:**

$$\text{Discount Amount} = (\text{Subtotal} \times \text{DiscountPercent})/100.$$

6. **Process: Tax Calculation (18% GST):**

$$\text{Tax Amount} = ((\text{Subtotal} - \text{Discount}) \times 18.0)/100.$$

7. **Process: Grand Total Calculation:**

$$\text{Grand Total} = \text{Subtotal} - \text{Discount} + \text{Tax}.$$

8. **Process: Date Capture:** Call `getCurrentDate()` to populate `bill->date`.
9. **Output: Final Bill Display:**
   - Print "FINAL BILL" header and Date.
   - Loop through all items and print their details (Itemized List).
   - Print financial summary (Subtotal, Discount Amount, Tax Amount, Grand Total).
10. **End.**

4 Data Persistence and Conclusion

4.1 Data Persistence (`saveBillToFile` Function Flowchart)

The `saveBillToFile` function handles the crucial task of ensuring the transaction record is stored permanently. It prompts for a filename and exports the detailed final bill into a text file.

Flowchart Description: `saveBillToFile` Logic

1. **Start.**
2. **Decision Node:** Is `itemCount` == 0?
   - **Yes:** Print "No bill to save!" message and **Return**.
   - **No:** Proceed.
3. **Input:** Prompt user for the `filename` (e.g., `bill.txt`).
4. **Process: File Opening:** Attempt to open the file using `fopen(filename, "w")` (write mode).
5. **Decision Node:** Is the file pointer NULL (Open Failed)?
   - **Yes:** Print "Error opening file!" message and **Return**.
   - **No:** Proceed.
6. **Process: Writing to File:**
   - Use `fprintf` to write the "FINAL BILL" header, Date, Itemized List (looping through all items), and the financial summary (Subtotal, Discount, Tax, Grand Total) to the opened file stream.
7. **Process: File Closing:** Close the file using `fclose(file)`.
8. **Output:** Print "Bill saved successfully to `filename`!" message.
9. **End.**

4.2 Conclusion and Future Scope

The Bill Management System successfully demonstrates a functional, console-based solution for sales record management. The structured approach using C structs ensures clean separation of data and logic, while the menu-driven interface provides an intuitive user experience.

The program's core strength lies in its modularity (distinct functions for input, display, calculation, and persistence) and its comprehensive financial logic, which accounts for both discounts and mandatory taxation.

For future enhancements, the system could be expanded to include:

1. **Data Retrieval:** Adding a function to load existing bills from a file.
2. **Error Handling:** Implementing more robust input validation (e.g., ensuring price and quantity are positive, and discount percentage is between 0 and 100).
3. **Dynamic Tax Rates:** Allowing the user to input the tax percentage instead of hardcoding the 18% rate.

This project serves as a solid foundation for developing more complex inventory and management systems.