Software Requirements Specification

File Name	Exchange Store
Creation Date	20.09.2025
Authored by	Kuzey Arda Bulut
Reviewed by	

REVISION HISTORY

VERSION	DATE	DESCRIPTION	AUTHOR
V1	15.09.2025	Choosed to do Exchange Store Application and did some reserach about the economic terms	**Kuzey Arda Bulut**
V2	16.09.2025	Created the program's general structure	**Kuzey Arda Bulut**
V3	20.09.2025	Implemented the heap objects	**Kuzey Arda Bulut**
V4	21.09.2025	Finalized the Project	**Kuzey Arda Bulut**

CONTENTS

1 SW System Overview

Specify the purpose and the overview of the SRS.

1.1.1 Main Actors

- ***Client**: requests a currency exchange and expects the converted result.*
- ***Cashier**: primary user in Release 1, inputs transactions and reads results.*
- ***System (Backend)**: validates inputs, performs conversion, and stores transaction details.*
- ***Administrator** (future role): manages exchange rates, reserves, and generates automated reports (not part of Release 1).*

1.1.2 Business Process

1. *Cashier selects **FROM** and **TO** currencies from a fixed list (LOC, USD, EUR, GBP, JPY).*
2. *Cashier enters the amount to convert.*
3. *System validates input (amount > 0, valid indices).*
4. *If valid → system computes result using predefined demo rates and prints it.*
5. *Each transaction is recorded in memory with details (date, time, currencies, rate, profit).*
6. *At the end of the day, cashier can view a **manual profit summary**. (Later releases: automated daily reports, receipt printing, CSV storage, reserve checks.)*

1.1.3 Problem Areas

- *Invalid inputs: negative or zero amounts, out-of-range currency indices.*
- *No persistence in Release 1 → all transactions lost after program exits.*
- *Manual profit entry/reporting introduces risk of errors.*
- *Hardcoded exchange rates → not realistic for actual business use.*
- *(Later problems: insufficient reserves, inconsistent rate updates, missing error logs.)*

1.1.4 Conclusion (Three Axes)

- ***Right Product**: focus on a simple **console-based exchange tool**, not a financial calculator. Release 1 supports fixed conversions only.*
- ***Done Right**: input validation (amount > 0, valid currency codes) is enforced; clear error messages are shown. Transaction details are tracked in memory.*
- ***Managed Right**: Release 1 uses a simple structure (main.c + utils.c/h). Future releases will modularize (e.g., exchange.c, reports.c), add CSV logging, denomination breakdown, and automated reporting.*

1.2 Purpose

Describe the purpose of the system. What problem does it solve? Who are the intended users? Why is it being developed?

The system is designed to simulate the daily operations of a currency exchange office. Currently, there is no comprehensive exchange store simulation available that enables cashiers to check exchange rates, perform transactions, issue receipts, and generate end-of-day reports. The intended users of the system include exchange store staff and stakeholders, such as cashiers, customers, and management.

1.3 Scope

Define the scope of the system. What functionality is included? What is explicitly excluded? Mention benefits and key features.

Included (Scope):

- *Perform exchanges between supported currencies.*
- *Input of amount and currency pair (FROM, TO).*
- *Validation of input (amount > 0, valid indices).*
- *Calculation using fixed, predefined exchange rates.*
- *Display converted result on screen.*

- *Denomination breakdowns: suggest note distribution for payouts.*
- *View and update demo exchange rates (set manually by administrator in later releases).*
- *Generate receipts and simple daily reports (manual profit entry in Release 1).*

Excluded (Not in Scope):

- *Real-time integration with market/online data.*
- *Internet access or networking functionality.*
- *Graphical User Interface (console only).*
- *Persistent storage (Release 1 has no files, no database).*
- *Automated reserve management (only manual adjustments in later releases).*

1.4 Stakeholders

Identify all user roles and stakeholders who interact with or benefit from the system.

- **Client** — requests an exchange, provides amount and currency pair, and views the result.
- **Cashier** — main user in Release 1; operates the program, inputs data, and reads results.
- **Administrator** — defines and updates fixed exchange rates (planned for later releases).

1.4 User Stories

Document user requirements as user stories to capture functional needs from the user perspective.

- *As a customer, I want to see the current exchange rates so that I can decide quickly whether to proceed with a transaction.*
- *As a customer, I want the system to prevent an exchange when reserves are insufficient so that I don't waste time.*
- *As a cashier, I want balances and reserves to update instantly after each transaction so that I always have accurate availability.*
- *As a manager, I want the system to log all transactions so that I can generate accurate reports.*

1.5 General Constraints

List technical and business constraints such as programming language, operating system, performance limitations, and standards.

The system is implemented in the C programming language by using the ISO C11 standard. It can be easily compiled and run on Linux systems through the Makefile; however, Windows users can also execute the program by compiling it. Performance constraints are minimal since the system is console-based. However, the design also introduces limitations. The system only supports five currencies (LOC, USD, EUR, GBP, and JPY), and payout suggestions are restricted to integer denominations, meaning fractional notes are not considered. Moreover, networking has not yet been implemented, so all currency prices hard-coded on the code. These constraints shape the technical and business requirements, balancing simplicity with functionality. In addition, LOC is the local currency in our exchange store which is made up and created by indexing Turkish Lira in order to have realistic exchange rates.

2 SW Functional Requirements

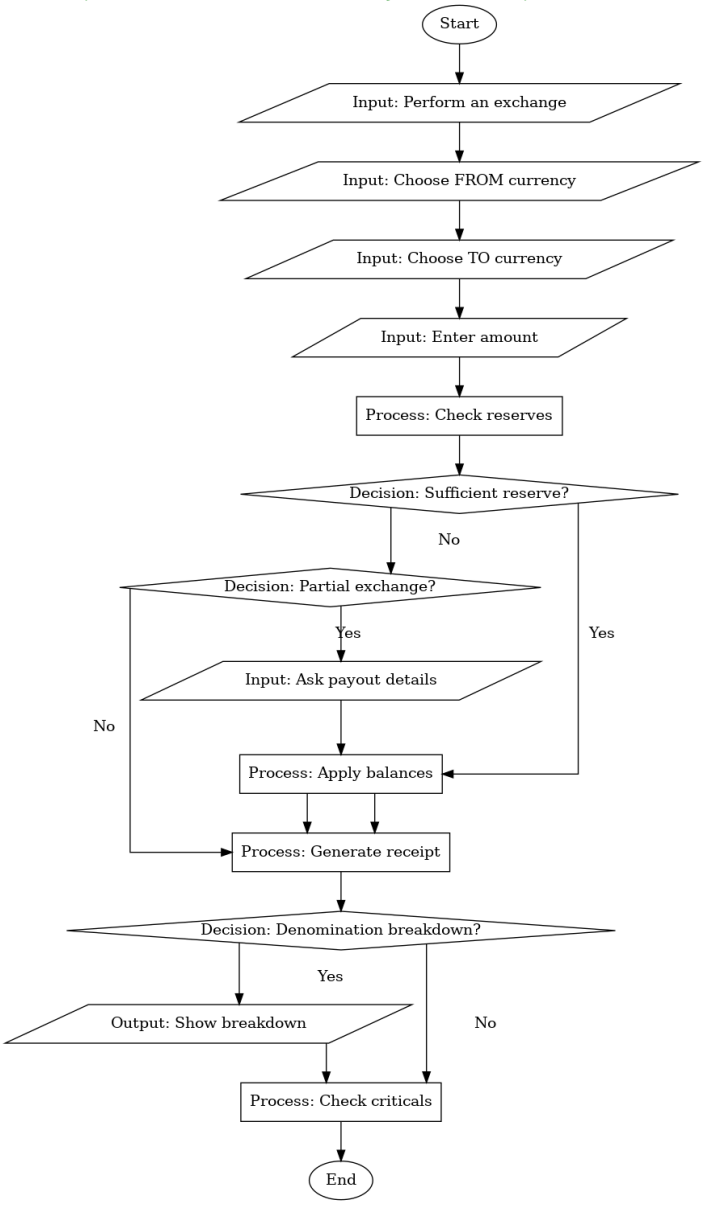
2.1 Features / Functions to be Implemented

List functional requirements derived from User Stories. Each function must be directly traceable to a User Story.

The system provides several essential features that form its functional core. Users can perform full or partial currency exchanges, which automatically update the reserves of the respective currencies and calculate the profit made. Management functions allow for exchange rates to be displayed and updated, ensuring flexibility in adapting to new financial conditions. Cashiers can modify reserves by depositing or withdrawing funds and set critical minimum levels in order to ensure about operational continuity. Receipts are generated after each exchange, while end-of-day reports summarize all activity, showing profits and balances in an organized manner.

2.2 Flowcharts

Provide flowcharts to illustrate system workflows and user interactions. Recommended: 2-3 key scenarios.



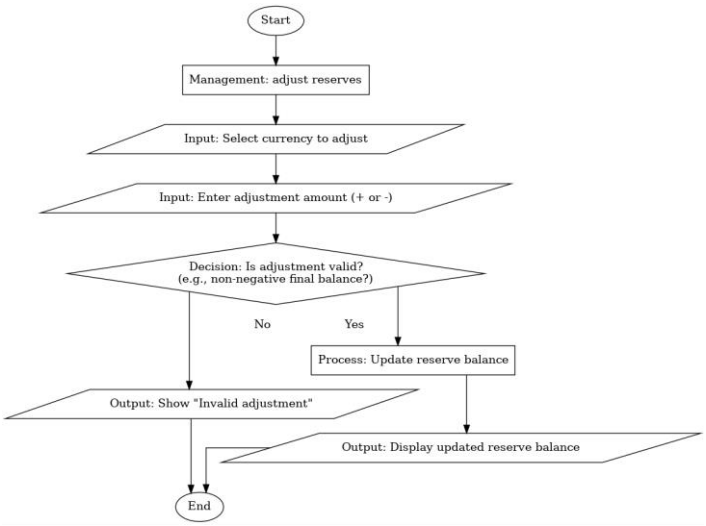
Algorithm 1: Perform an Exchange

1. User selects **Perform an exchange**.
2. User chooses **FROM currency**.
3. User chooses **TO currency**.
4. User enters the **amount**.
5. Program checks if the entered amount is valid (> 0 and numeric). If not → show error and re-prompt.
6. Program checks the **reserves** for the selected currency.
7. If reserves are **sufficient**:

- Apply balances.
- Generate receipt.
- Ask if denomination breakdown is needed.
- If yes → show note breakdown.
- If no → skip breakdown.
- Check critical balances.
- End.

8. If reserves are **not sufficient**:

- Program asks if **partial exchange** is acceptable.
- If yes → user provides payout details → program applies balances → generate receipt → denomination breakdown decision → check criticals → end.
- If no → program generates receipt (with failure/notice) → end.



Algorithm 2: Adjust Reserves

1. Management selects **Adjust reserves**.
2. System asks to choose the **currency** to adjust.
3. System asks to enter the **adjustment amount** (positive or negative).
4. Program checks if the **adjustment is valid** (e.g., balance cannot go below zero).
    a. If invalid → show error **"Invalid adjustment"** → end.
    b. If valid → update reserve balance.
5. Display the updated reserve balance.
6. End the process.

2.3 Decision Tables

Provide decision tables to capture conditional business rules and system logic.

System logic is also governed by decision rules, particularly for handling exceptional cases. For example, if the reserve of the target currency is insufficient, the system decides whether to proceed with a partial payout, offering the customer the remainder in local currency, or to cancel the transaction. Similarly, the system chooses to accept a fresh input from the user rather than failing if the user enters improper data, such as a negative quantity or an unsupported currency index. These decision tables provide consistent and predictable system behavior by capturing the conditional logic.

2.4 Data Dictionary

Define data entities, attributes, data types, and constraints used in the system.

The system uses structured data representations to track state. Currency names are stored as character arrays, while reserves, critical minimums, buy rates, and sell rates are stored as double-precision floating-point arrays indexed by currency type. Profit is tracked as a single floating-point variable in local currency. Denominations for each currency are defined as constant arrays of integers, supporting the breakdown of payouts into notes. This structured approach ensures that data is easy to manage and accessible to all system functions.

2.5 I/O Contracts

Specify inputs, outputs, and expected behavior of key system functions.

The system's input and output behavior is clearly defined. Inputs consist of menu choices, currency indices, and numeric amounts entered by the user. Outputs include printed receipts, updated balances, exchange rate tables, and end-of-day reports. The system ensures safe behavior: invalid inputs trigger re-prompts, and insufficient reserves trigger error messages. This makes the I/O contracts reliable and predictable, protecting both users and the system from unexpected states.

2.1 Acceptance Criteria

Define measurable criteria used to verify that each requirement is correctly implemented and tested.

The following criteria define successful implementation for the current release:

- *All exchange transactions are correctly processed according to hard-coded demo rates.*
- *Partial payouts are allowed if the requested amount exceeds available reserves, with profits*

calculated on the executed portion only.

- *Warnings are issued whenever currency balances fall below the defined critical minimums.*
- *End-of-day reports display per-currency starting balances, ending balances, transaction counts, and total profit.*
- *Data is consistently logged in CSV files for daily summaries.*

3 SW Non-Functional Requirements

3.1 Coding Standard

Specify coding conventions and standards to be followed throughout development.

*The product follows recognized coding guidelines to guarantee readability and maintainability. All functions are named in **snake_case**, constants are written in capital letters, and I**SO C11** standards are followed. This ensures consistency, portability, and adherence to academic best practices.*

3.2 Modular Design

Specify structural requirements for modularity, maintainability, and extensibility of the software.

The program is implemented across multiple source files with functionality organized into logical modules. Menu handling and utility functions scenarios are separated into distinct function groups in order to ensure clear structure and maintainability. This modularity facilitates future extensions, such as adding new currencies, and implementing advanced persistence mechanisms, or expanding reporting features.

3.3 Constraints (Technical + Business)

Define non-functional technical and business limitations that restrict or influence design and implementation.

Non-functional constraints include the reliance on double-precision arithmetic, which may introduce rounding issues, though acceptable for educational use. The system supports only one active user at a time and does not provide concurrency or networking features. Business-wise, the implementation is intended for academic demonstration rather than deployment in a real financial institution.

4 SW Design Artifacts

Modules & Responsibilities

Define each software module, its responsibilities, and key functions it implements.

The design consists of distinct modules with defined responsibilities. The main loop handles the menu interface and user navigation. Exchange scenarios process transactions, update balances, and generate receipts. Management scenarios allow configuration of rates, reserves, and critical thresholds. The end-of-day scenario compiles and displays reports, while utility functions handle tasks such as safe input collection, denomination breakdown, and receipt formatting. This structure keeps the design clean and easy to maintain.

Data Structures

List and define data structures used in the implementation, including fields, data types, and constraints.

Data structures are primarily arrays and constants, but balances, rates, and thresholds are managed through a heap-allocated structure referenced by a global pointer. This allows flexible initialization at runtime and proper cleanup upon program termination. Profit is maintained as a single floating-point variable. Denominations for each currency are defined as integer arrays, supporting note breakdown. This approach combines efficiency with modularity while keeping data access straightforward.

File Formats & Persistence

Define file formats, data storage structures, and persistence requirements.

*The system currently stores transactional data in CSV files to enable end-of-day reporting and profit tracking. While core exchange rates and reserves remain in memory during execution, daily transactions are persisted in files such as **sales_<date>.csv**. In future development stages, additional persistence will be introduced to cover exchange rates, currency reserves, and customer receipts across program executions.*

Error Handling Plan

Specify error conditions, error codes, and handling strategies.

Validation checks and safe input procedures are used to handle errors. The system discards any invalid data entered by the user and asks again. The system cancels the transaction and produces a clear error notice if reserves are not sufficient. Every time reserves drop below safe levels, alarms are shown and critical minimums are continuously monitored. This method guarantees dependability without being overly complicated.

5 Testing & Validation

Acceptance Tests

Provide test cases that demonstrate fulfillment of functional and non-functional requirements.

Testing focuses on validating both normal and edge cases. For example, exchanging 100 USD to EUR with sufficient reserves should update balances correctly and generate a receipt. Attempting an exchange when reserves are insufficient should trigger an error message and prevent any changes. When balances fall below a critical minimum, the system should immediately alert the cashier. The end-of-day report must always reflect accurate profit and balance data.

Metrics

Provide quantitative measures of system size, complexity, and quality (e.g., lines of code, number of functions, defects fixed).

The implementation consists of approximately 500 lines of code and around 25 functions that represents a moderately sized educational project. Key defects resolved during development include ensuring input validation loops and implementing correct checks for partial payouts. These metrics show that the system is both functional and robust for its scope.

Retrospective

Summarize key lessons learned: what worked well, what challenges were encountered, and what should be improved.

The development process revealed several strengths and challenges. The menu-driven structure, modular scenarios, and receipt system worked particularly well, providing a smooth and intuitive flow. Challenges included managing reserve validation and designing consistent rules for partial payouts. Future improvements could include adding persistent storage, implementing structured error codes, and supporting multiple users or concurrent sessions. These lessons will guide the next stages of development.

6 Release Coverage Table

Section	CP1 (W4)	CP2 (W8)	CP3 (W12)	Final (W16)
1.1 Purpose	☑			
1.2 Scope	☑			
1.3 Stakeholders	☑			
1.4 User Stories (+ Criteria)	☑	☑	☑	☑
1.5 Constraints (Tech+Business)	☑	☑	☑	☑
2.2 Flowcharts	☑		☑	☑
2.3 Decision Tables (Rules)		☑	☑	☑
2.4 Data Dictionary		☑	☑	☑
2.5 I/O Contracts		☑	☑	☑
2.6 Acceptance Criteria		☑	☑	☑
3. Non-Functional (Coding, Modularity, Constraints)			☑	☑
4. Design Artifacts			☑	☑
5. Testing & Validation – Acceptance Tests			☑	☑
5. Testing & Validation – Metrics			☑	☑
5. Testing & Validation – Retrospective				☑