DND Sheet Design Document (Final)

##Project Goals
This project is meant to replace shitty DND character apps that cause problems for me. This is for my sole personal use FOR NOW, and will not be turning it in for a grade or anything. I want it to eventually look good because I will be using it, but that is not a priority. I am a UNLV sophomore in experience, but I will NOT be submitting this for anything.

Project Summary

This project is a manual-entry, zone-based character sheet / workspace system built around **two completely separate subsystems**: **Counters** and **Text**.

In both subsystems, **files on disk are the sole source of truth**. The UI is a direct projection of those files. The system intentionally avoids automation, rule enforcement, and hidden logic. Its goals are flexibility, predictability, and safety through explicit user control and reliance on external editors.

With one exception, **all files are read only on application launch**. That exception is the Counters runtime state file, which is written by the application and treated as authoritative for live counter values. All other data is snapshotted. There are no filesystem reads after launch except on the counters state file.

---

1) Counters Subsystem

Purpose

A persistent, independently scrolling area for frequently used numeric trackers (HP, resources, charges, etc.). This area is called the **Counter Zone (CZ)**.

Counter Zone (CZ) UI

* Displays a vertical list of counters.
* Each counter shows:

  * Name
  * Decrement control
  * Current value
  * Increment control
  * Reset control (↻) that sets the current value to the reset value

* Global controls:

  * **Reset All**: resets all counters to their reset values
  * **Edit Counters**: opens the counter definition file externally
* CZ scrolling is completely independent from the rest of the application.

Counter Definitions File (CSV)

* Stored at: `SheetRoot/Counters.csv`
* This file is **manually edited** and is **read only on application launch**.
* It is the authoritative source for counter definitions.

Format

Each row represents one counter and must contain **exactly three logical fields**:

1. `id` (string)
2. `name` (string)
3. `reset` (number, ^-?\d+$)

Header rows are optional; if present, they are treated as normal rows (no special handling).

Example:

```
hp,Health,15
mp,Mana,32
hd,Hit Dice,4
pots,Potions,6
```

Parsing and Error Handling (Intentional Defaults)

* Every row always produces a counter definition.
* If a row does **not** contain exactly three values:

  * Treat it as: `error,error,-1`
* If the row has three values but the final value (`reset`) is blank or non-numeric:

  * Set `reset = -1`
* `id` and `name` are taken as-is (after CSV unquoting) unless the row-count rule triggers the fallback.

Malformed rows are therefore **visible in the UI** rather than silently discarded.

ID Stability Rule

* The `id` field is the **stable identity** of a counter.
* Changing a counter’s `name` preserves its state.
* Changing a counter’s `id` creates a **new counter** with no prior state.
* If there are multiple lines in the csv file with the same id, they will appear multiple times with the same value (stored in the json file)

Counter Runtime State (JSON)

* Stored at: `SheetRoot/Counters.state.json`
* This file is **managed exclusively by the application** and must not be edited manually.
* It is the authoritative source for **current counter values** while the app is running.

Contents

* The file stores **only current values**, keyed by counter `id`.
* No definition data (`name`, `reset`) is duplicated here.

Example:

```json
{
  "values": {
    "hp": 12,
    "mp": -31
  }
}
```

Launch-Time Reconciliation Rules

On application launch:

1. Load and parse `Counters.csv` into counter definitions.
2. Load `Counters.state.json` if it exists; otherwise treat it as empty.
3. For each counter defined in the CSV:

  * If a current value exists in the JSON and is numeric, use it.
  * Otherwise, initialize the current value to the counter’s reset value.
4. State entries whose `id` does not exist in the CSV are ignored and removed on the next write.

This ensures CSV edits are safe and predictable:

* Adding a counter → appears with `current = reset`
* Removing a counter → disappears and its state is discarded
* Changing `name` → label updates, current value preserved
* Changing `reset` → reset behavior changes, current value preserved

---

Runtime Interaction Model (File-Authoritative)

All UI actions follow the same rule:

> **UI actions modify files; the UI updates in response to files.**

For any interaction (increment, decrement, reset, reset all, inline edit):

1. Read `Counters.state.json`
2. Compute the new value(s)
3. Atomically write the updated file
4. Refresh the UI from the file contents

There is no authoritative UI state and no persistent in-memory source of truth.

Direct Manipulation

* Double-clicking a counter’s displayed number turns it into an inline text field.
* This edits the **current value only**, not the reset value.
* Commit on Enter or blur of a numeric value.
* Cancel on Escape.
* Cancel on Enter of blur of a non-numeric value.

File Safety and Concurrency

* All writes to `Counters.state.json` are **atomic** (write temp file, then replace).

* The application enforces a **single-instance lock** at startup.

  * If another instance is already running, the second instance must not take control.
* External edits to the state file are unsupported and undefined behavior.

---

2) Text Subsystem

Purpose

A flexible navigation and content workspace called the **Text Zone (TZ)**, backed entirely by a snapshot of folders and plain text files.

File System as Structure

* Root directory: `SheetRoot/’
* Each section’s content is stored in a file named `text.txt` inside that folder. If this file doesn’t exist, it is assumed to be blank.
* The TZ ignores anything that is not a folder or a file called text.txt.

Example:

```
SheetRoot/
    text.txt
    counters.csv
    counters.state.json
    1Inventory/
      text.txt
    1Backpack/
      text.txt
    2Equipment/
      text.txt
    2Spells/
      text.txt
```

UI Navigation Model

Persistent Top Row (Root Sections)

* The first level of folders under `SheetRow` is displayed as a **row of tabs/buttons**.
* This row is always visible to allow quick switching between top-level sections.

Deeper Navigation (Single Stacked Column)

* Selecting a top-level section shows a single column listing its immediate subfolders.
* Selecting a subfolder navigates deeper.
* The column panel is replaced in-place (no multi-column expansion).
* A **Back** button navigates one level upward.

Content Area

* Fixed size.
* Displays the contents of the selected folder’s `text.txt`.
* If `text.txt` is missing, the content area is blank.

Ordering via Hidden Prefix Character

* Sorting is alphabetical based on folder name.
* The **first character** of each folder name is hidden in the UI.
* Users control order by naming folders with prefix characters.

Example:

* On disk: `1Inventory`, `2Spells`, `3Stats`
* In UI: `Inventory`, `Spells`, `Stats`

Editing Workflow

* All structure changes are performed via the filesystem:

  * Creating, renaming, deleting folders
  * Editing `text.txt` files externally
* The Text subsystem is **read only on application launch**.
* No live refresh occurs while the app is running.

---

3) System Architecture and Principles

Two Separate Systems

The application consists of two non-connected parts:

1. **Counters**

   * Definitions: `Counters.csv` (manual, launch-only)
   * State: `Counters.state.json` (app-managed, live)
2. **Text**

   * Structure and content: folder tree + `text.txt` files (manual, launch-only)

There are no dependencies or references between the two systems.

Core Design Principles

* Files are the source of truth.
* UI is a projection of disk state.
* Manual authoring first; no automation or rule enforcement.
* Safety through simplicity and explicit control.
* External editors are preferred for content creation.
* Minimal hidden logic; behavior is predictable and visible.

One-Line Description

A two-part, file-defined, manually authored character/workspace UI: **Counters** rendered from a CSV with live, file-authoritative runtime state, and **Text** rendered from a prefix-ordered folder tree of plain text sections—fully independent, predictable, and optimized for external editing.
