

Complete Z-Machine Editor User Manual

Z-Machine Infogames Text Adventure Creator

by DJ Sures

Manual Version: 2026.02.08.00

This Tool Is Not Just an Editor

This software is a **full Z-Machine development toolchain**:

Editor → Compiler → Z-Machine Story File → Interpreter → Player

You are building real 1980s-compatible interactive fiction that runs on:

- Original CP/M systems
- NABU PC
- Apple II, Commodore 64 (via emulators)
- DOS (DOSBox)
- Modern Windows, Linux, macOS (Frotz, etc.)

This is not an emulator.

This is **authoring original Z-Machine software**, a recreation of the platform used by Infocom.

Table of Contents

1. Introduction

- 1.1 What Is This Software?
- 1.2 Who This Tool Is For
- 1.3 What You Can Create

2. The Z-Machine Explained

- 2.1 What Is the Z-Machine?
- 2.2 Why the Z-Machine Exists
- 2.3 How Interpreters Work
- 2.4 Z-Machine Versions (V1–V8)
- 2.5 Why This Editor Targets V3

3. History of Infocom & Interactive Fiction

- 3.1 The Birth of Zork
- 3.2 The Creation of Infocom
- 3.3 The Z-Machine Virtual Computer
- 3.4 Why Infocom's Architecture Still Matters
- 3.5 Famous Infocom Games

4. Story File Formats (.DAT, .Z3, .Z5, .Z8)

- 4.1 What a Story File Is
- 4.2 File Extensions Explained
- 4.3 Compatibility with Interpreters
- 4.4 Why .DAT and .Z3 Are the Same
- 4.5 Choosing the Right Format

5. How This Editor Works (Big Picture)

- 5.1 The Toolchain Pipeline
- 5.2 Project → Compiler → Story File
- 5.3 What Happens During Compile
- 5.4 What the Compiler Generates
- 5.5 How Interpreters Execute Your Game

6. Installing & Running the Editor

- 6.1 System Requirements
- 6.2 First Launch
- 6.3 Creating a New Project
- 6.4 Opening an Existing Project
- 6.5 Updating the Editor

7. Project Files (.zproj.json)

- 7.1 Project Structure
- 7.2 Manual Editing
- 7.3 Version Control (Git-Friendly)
- 7.4 Backups & Recovery
- 7.5 Project Portability

8. The Main Interface

- 8.1 Project Settings
- 8.2 Rooms Panel
- 8.3 Items Panel
- 8.4 Verbs Panel
- 8.5 Build & Compile Panel
- 8.6 Story Inspector

9. How the Game Engine Thinks

- 9.1 Input Parsing Pipeline
- 9.2 Dictionary & Tokenization
- 9.3 Verb Resolution
- 9.4 Noun Resolution
- 9.5 Action Dispatch
- 9.6 State Updates

10. Rooms (Locations)

- 10.1 Creating Rooms
- 10.2 Room IDs & Names
- 10.3 Room Descriptions
- 10.4 Exits & Navigation
- 10.5 Room-Based Conditions
- 10.6 Conditional Descriptions

11. Items (Objects)

- 11.1 Creating Items
- 11.2 Takeable vs Fixed Objects
- 11.3 Inventory
- 11.4 Hidden Items
- 11.5 Item States
- 11.6 Object-Based Conditions

12. Verbs (Actions & Commands)

- 12.1 Defining Verbs
- 12.2 Synonyms
- 12.3 Default Responses
- 12.4 Verb Overrides
- 12.5 Contextual Verbs
- 12.6 Verb Conflicts

13. Dictionary & Parsing

- 13.1 Tokenization
- 13.2 ZSCII Encoding
- 13.3 Dictionary Limits
- 13.4 Handling Unknown Words
- 13.5 Debugging Parser Errors

14. Scripting & Conditions

- 14.1 How Scripts Work
- 14.2 Conditions
- 14.3 Flags & State
- 14.4 Multi-Step Puzzles
- 14.5 Context-Sensitive Actions
- 14.6 Reusable Logic Patterns

15. Writing Responses & Game Logic

- 15.1 Player Feedback
- 15.2 Failure States
- 15.3 Hinting Without Spoiling
- 15.4 Branching Narratives
- 15.5 State-Dependent Text

16. Tutorial: Your First Adventure Game

- 16.1 Creating the Project
- 16.2 Designing Rooms
- 16.3 Adding Items
- 16.4 Creating Verbs
- 16.5 Writing Your First Puzzle
- 16.6 Compiling & Running

17. Editing Existing Story Files

- 17.1 What You Can Inspect

- 17.2 Why You Can't Edit Compiled Files
- 17.3 Extracting Dictionaries
- 17.4 Reverse Inspection
- 17.5 Migrating Old Games

18. Running Your Game

- 18.1 Using the Built-In Interpreter
- 18.2 Running in Frotz
- 18.3 Running on CP/M
- 18.4 Running on NABU PC
- 18.5 Distribution Testing

19. Exporting & Inspecting

- 19.1 Dictionary Export
- 19.2 Header Inspection
- 19.3 Object Table Inspection
- 19.4 Checksum Verification
- 19.5 Debug Views

20. Common Mistakes & Pitfalls

- 20.1 Broken Room Links
- 20.2 Missing Items
- 20.3 Parser Conflicts
- 20.4 Soft-Locking the Player
- 20.5 Debugging "Nothing Happens"

21. Advanced Design Techniques

- 21.1 Puzzle Design Patterns
- 21.2 Narrative Structure
- 21.3 Environmental Storytelling
- 21.4 Player Guidance
- 21.5 Replayability

22. Debugging & Testing Strategies

- 22.1 Playtesting Techniques
- 22.2 Troll Testing
- 22.3 Save/Load Testing
- 22.4 Edge Case Testing
- 22.5 Regression Testing

23. Versioning, Compatibility & Limits

- 23.1 Z-Machine V3 Limits
- 23.2 Memory Constraints
- 23.3 Object Limits
- 23.4 Performance Considerations
- 23.5 Forward Compatibility

24. Publishing Your Game

- 24.1 Packaging for Players
- 24.2 Including Interpreters
- 24.3 CP/M Distribution
- 24.4 Web Distribution
- 24.5 Archiving & Preservation

25. FAQ & Troubleshooting

- 25.1 Common Questions
- 25.2 Compatibility Issues
- 25.3 Performance Problems
- 25.4 Save File Issues
- 25.5 Reporting Bugs

2. What Is the Z-Machine? (Expanded)

The Z-Machine is a **portable virtual CPU** created by Infocom in 1979.

It defines:

- Memory layout
- Instruction set
- Object model
- Dictionary
- Text encoding (ZSCII)
- Input parsing rules
- Save/restore format

Infocom compiled all of their games into Z-Machine bytecode, then wrote **interpreters** for:

- Apple II

- C64
- IBM PC
- CP/M (MSX, Coleco Adam, NABU, IBM PC, etc...)
- Amiga
- Atari ST
- TRS-80

This is one of the earliest examples of:

Write once, run anywhere

...predating Java or .Net by almost 20 years.

Your editor generates a real Z-Machine V3 story file.

Nothing custom. Nothing proprietary.

This means your games will still run decades from now on any Z-Machine interpreter.

3. A Brief History of Infocom & the Z-Machine (Expanded)

Origins (1977–1979)

Zork began as a PDP-10 mainframe game written in MDL (a Lisp-like language) at MIT.

Infocom formed specifically to commercialize it.

Problem:

Every home computer was different.

Solution:

*They invented the **Z-Machine**.*

Instead of porting games, they ported the *interpreter*.

Why This Was Revolutionary

Before Infocom:

- Each game had to be rewritten per platform.

After Infocom:

- One compiled story file ran everywhere.

This enabled:

- Faster development
- Identical gameplay across platforms
- Easier bug fixes
- Smaller teams
- Faster releases

Modern engines (Unity, JVM, WASM) all follow this same idea.

Z-Machine Versions

Version Era		Notes
V1–V2	Very early	Rare, experimental
V3	1982–1987	Most Infocom titles
V4–V5	Later	Larger memory
V6	Graphics/sound	Rarely supported
V7–V8	Large stories	Modern IF

This editor targets **V3** because:

- Works on CP/M with a primary focus on NABU PC and Cloud CPM
- Small memory footprint
- Maximum compatibility
- Historically authentic

4. Z-Machine Files (.DAT, .Z3, etc.) (Expanded)

Infocom did not standardize file extensions.

Extension Meaning

.z3 Z-Machine Version 3

.dat Often renamed .z3

.z5 Version 5

.z8 Version 8

Under the hood:

- .dat and .z3 are identical formats
- Interpreters detect version from header
- The extension is cosmetic

Your editor produces:

story.z3

Both work identically.

9. How the Game Engine Thinks (Deeper Mental Model)

Internally the Z-Machine works like a tiny OS:

Input → Tokenizer → Dictionary → Parser → Action → State Change → Output

Step-by-step example:

Command:

use rusty key on door

Parser flow:

1. Normalize:

- use → verb
- rusty → adjective
- key → noun
- door → second noun

2. Dictionary lookup:

- key → item ID
- door → object ID

3. Rule matching:

- Does room contain door?
- Is key in inventory?
- Is door locked?

4. Script execution:

- If player_has(key) && door_locked
- Unlock door
- Print message

5. State update:

- door.locked = false

6. Output:

- "The rusty key turns and the door creaks open."

14. Scripting & Conditions (Deep Dive)

Your editor provides a **high-level scripting layer** that compiles into Z-Machine bytecode.

You do not write Z-assembly.

You write game logic.

Core Concepts

Concept	Meaning
player_has(item)	Item is in inventory
in_room(roomId)	Player location
flag_set(flag)	Boolean state

Concept	Meaning
item_in_room(item, room)	Spatial logic
item_used_on(a, b)	Contextual action

Example: Locked Door

```
IF player_has(key) AND in_room(cabin)
  set_flag(door_unlocked)
  print "You unlock the door."
  enable_exit(north)
ELSE
  print "The door is locked."
```

Example: Conditional Description

```
IF flag_set(power_on)
  print "The computer hums softly."
ELSE
  print "The computer is dark and lifeless."
```

Example: Multi-Step Puzzle

```
IF player_has(wire) AND player_has(battery) AND in_room(generator)
  set_flag(generator_fixed)
  print "The generator sputters to life."
ELSE
  print "You're missing something."
```

15. Writing Responses & Conditional Logic (Expanded)

Good IF games *teach players how to think.*

Bad:

Nothing happens.

Better:

You can't open the door with your hands. It looks like it needs a key.

Great:

You rattle the handle. The lock is old and rusted. A key might work.

Design rule:

- Every failure should hint at success
- Every puzzle should teach the mechanic
- Never leave players guessing what verbs exist

16. Step-By-Step: Making Your First Adventure (Expanded)

Example Game: “The Last Cabin”

Rooms:

- Cabin
- Forest
- Basement

Items:

- Key
- Lantern
- Battery

Verbs:

- unlock
- use
- climb
- search

Puzzles:

- Find lantern
- Add battery
- Light basement
- Find key
- Unlock door

This creates:

- Exploration
- Item dependency
- Environmental storytelling
- Multi-step progression

21. Advanced Design Tips (Expanded)

Don't Soft-Lock the Player

Bad:

- Drop key into pit
- No way to retrieve

Good:

- Allow retrieval
- Or reset puzzle

Layered Puzzle Design

Layer	Example
-------	---------

Discovery	Player sees locked door
-----------	-------------------------

Preparation	Finds key
-------------	-----------

Layer	Example
Execution	Uses key
Consequence	New area opens

Environmental Storytelling

Instead of:

There is a dead body.

Use:

The skeleton still clutches a rusted lantern. Scratch marks cover the stone floor.

22. Debugging & Testing (Expanded)

Test these cases:

- Random gibberish input
- Partial commands
- Verb-only commands
- Noun-only commands
- Commands in wrong room
- Repeated commands
- Save → quit → load → continue
- Try to break puzzles intentionally

Pro tip:

Play your game *like a troll*.

Try to ruin it.

23. Versioning, Compatibility & Limits (Expanded)

Z-Machine V3 constraints:

Feature Practical Limit

Rooms ~255

Objects ~255

Flags ~64

Memory ~128 KB

Text Compressed

This is perfect for:

- Mystery games
- Horror
- Puzzles
- Detective stories
- Small RPGs
- Educational adventures

24. Example Game Types You Can Create

Detective Noir

- Interrogate NPCs
- Gather clues
- Piece together evidence

Survival Horror

- Explore haunted house
- Limited light
- Locked doors
- Hidden notes

Sci-Fi Exploration

- Crashed ship
- Repair systems
- Restore power
- Escape planet

Fantasy Dungeon

- Keys
- Potions
- Traps
- Secret rooms

Educational Games

- History adventure
- Language learning
- Programming puzzles

25. FAQ (Expanded)

Is this reverse engineering Infocom?

No. The Z-Machine spec is public. This is a clean-room implementation.

Can I make my own interpreter?

Yes. The format is documented.

Will these games still run in 20 years?

Yes. Z-Machine interpreters are stable and widely implemented.

Can I export to other formats later?

Yes. Your .zproj.json is future-proof source code.

Final Thought

You're not just building a game editor.

You're reviving a **lost creative platform**.

This lets modern creators ship software for:

- 8-bit computers
- Vintage operating systems
- Emulators
- Future machines

That's insanely cool.