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

---

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.