Game Passwords

Knox Game Design

May 2026

History

- Commonly used for console games in the 8-bit NES era and 16-bit SNES era (1985-1994)
- 250 NES games had a password system
- Only a few games like Zelda, Dragon Warrior, and StarTropics had batteries for save games
- Passwords essentially became obsolete for console games after Playstation introduced removable memory cards (1995)

KNOX GAME DESIGN

- Pros
  - Last forever
  - Never changing
  - Can be used or shared
    - Printed in magazines or books
  - Example – rent a game multiple times, continue where you left off using password
  - Won’t get erased like save games
  - Use for debugging game (many / infinite lives, upgrades, etc)
- Cons
  - Having to write down long series of characters
    - Pre-cell phone cameras
  - Strange characters
  - Similar looking characters (O/o, 1/l, etc)
  - Players can skip playing the actual game
  - Losing the paper where passwords are written

KNOX GAME DESIGN

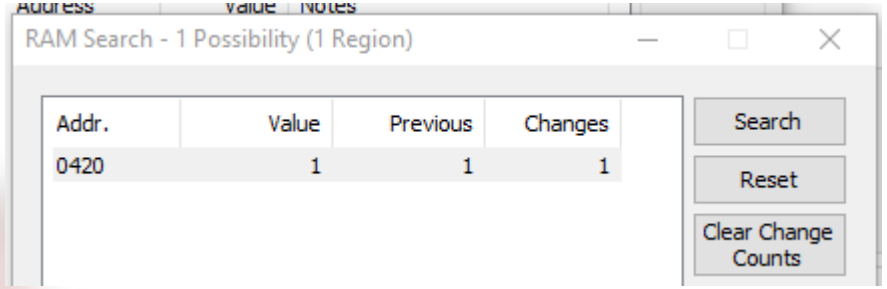
Designing a retro game password system

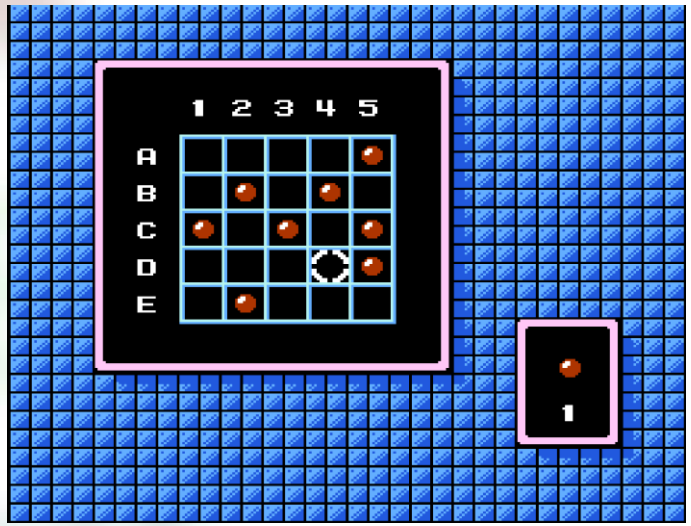
- Password must be able to capture all of the player's progress, completed levels, collected items, etc
- Assume pre-Internet
  - Passwords must not be easily guessable
  - No ability look at game source (OpCodes)
- Static strings
- Validate passwords
  - checksum
- Obfuscation
  - bit shifts
  - hash with user input

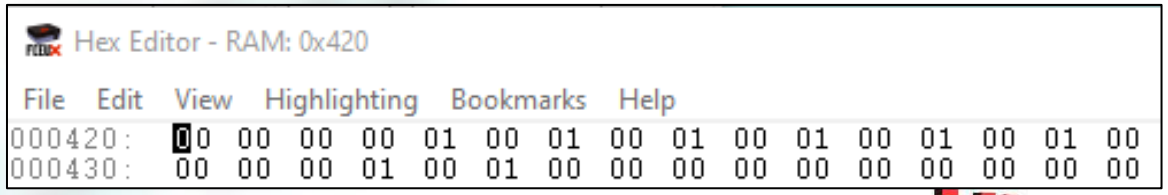
KNOX GAME DESIGN

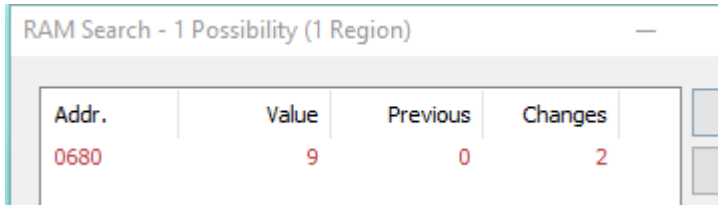
Mega Man 2

- Grid based password
- Row “A” is number of energy tanks
- Remaining is boss alive / dead (stage completed)
- Password input location starts at $0420 and stops at $0438
  - Found with FCEUX RAM Search
  - Looked for change from 0 to 1 (guessed). Assumed they used a whole byte for marker (guessed again).
  - A1 = $0420, A2 = $0421, ... , B1 = $0425, ..., E5 = $0438
  - Memory locations only apply to the password entry screen
- Markers remaining is at $0680
  - FCEUX RAM search for 1 to 0, then 9 on reset









KNOX GAME DESIGN

Mega Man 2

- Cracked password system
- First select number of energy tanks from row “A”
- Then select alive/dead for each boss from the table according to number of energy tanks

	ENERGY TANKS				
	0	1	2	3	4
ENERGY TANKS	A1	A2	A3	A4	A5
BUBBLE ALIVE	C3	C4	C5	D1	D2
BUBBLE DEAD	D1	D2	D3	D4	D5
AIR ALIVE	D2	D3	D4	D5	E1
AIR DEAD	E3	E4	E5	B1	B2
QUICK ALIVE	C4	C5	D1	D2	D3
QUICK DEAD	B4	B5	C1	C2	C3
HEAT ALIVE	D5	E1	E2	E3	E4
HEAT DEAD	B2	B3	B4	B5	C1
WOOD ALIVE	B5	C1	C2	C3	C4
WOOD DEAD	D3	D4	D5	E1	E2
METAL ALIVE	E1	E2	E3	E4	E5
METAL DEAD	E5	B1	B2	B3	B4
FLASH ALIVE	E4	E5	B1	B2	B3
FLASH DEAD	C1	C2	C3	C4	C5
CRASH ALIVE	E2	E3	E4	E5	B1
CRASH DEAD	C5	D1	D2	D3	D4

Source: RetroMaggedon Mega Man 2

KNOX GAME DESIGN

Metroid

- password length: 24 “Metroid” characters
  - 6 bits each
- 18 bytes (144 bits)
- First 16 bytes (128 bits) are game data
- Passwords are “encrypted” with bit rotation (“key” is byte index 16)
- Passwords are validated with a checksum (byte index 17)
- bytes are converted to Metroid password in 6 bit chunks
  - Metroid “0” = 0 dec = 000000 bin
  - Metroid “A” = 10 dec = 001010 bin
  - Metroid “J” = 13 dec = 010011 bin
  - Metroid “a” = 36 dec = 100100 bin

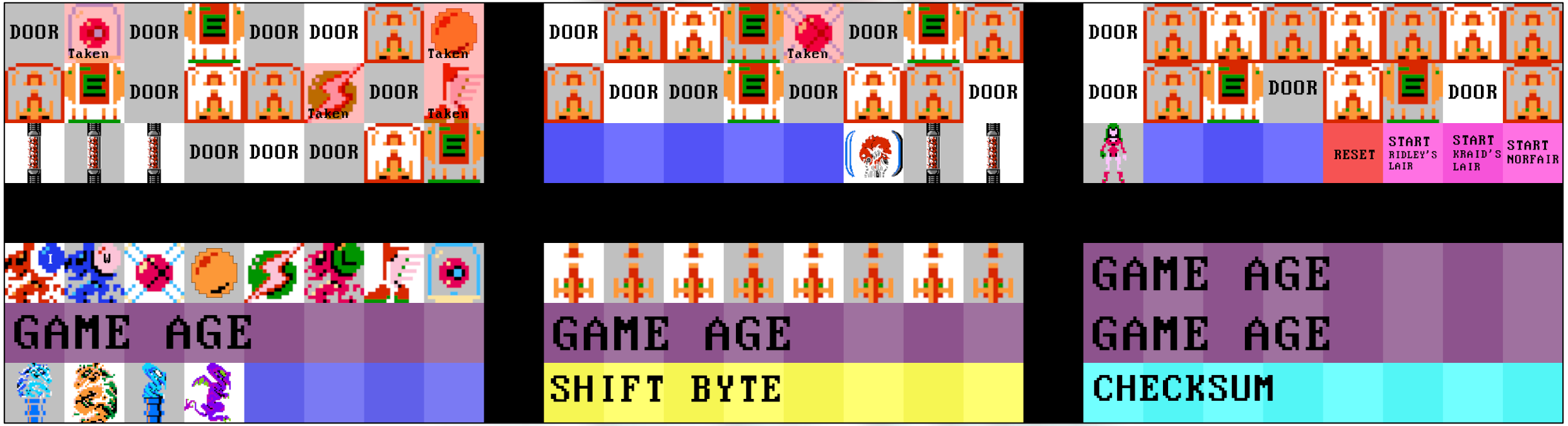
MT	dec	hex	bin	MT	dec	hex	bin
0	0	00	000000	a	36	24	100100
1	1	01	000001	b	37	25	100101
2	2	02	000010	c	38	26	100110
3	3	03	000011	d	39	27	100111
4	4	04	000100	e	40	28	101000
5	5	05	000101	f	41	29	101001
6	6	06	000110	g	42	2A	101010
7	7	07	000111	h	43	2B	101011
8	8	08	001000	i	44	2C	101100
9	9	09	001001	j	45	2D	101101
A	10	0A	001010	k	46	2E	101110
B	11	0B	001011	l	47	2F	101111
C	12	0C	001100	m	48	30	110000
D	13	0D	001101	n	49	31	110001
E	14	0E	001110	o	50	32	110010
F	15	0F	001111	p	51	33	110011
G	16	10	010000	q	52	34	110100
H	17	11	010001	r	53	35	110101
I	18	12	010010	s	54	36	110110
J	19	13	010011	t	55	37	110111
K	20	14	010100	u	56	38	111000
L	21	15	010101	v	57	39	111001
M	22	16	010110	w	58	3A	111010
N	23	17	010111	x	59	3B	111011
O	24	18	011000	y	60	3C	111100
P	25	19	011001	z	61	3D	111101
Q	26	1A	011010	?	62	3E	111110
R	27	1B	011011	-	63	3F	111111
S	28	1C	011100				
T	29	1D	011101				
U	30	1E	011110				
V	31	1F	011111				
W	32	20	100000				
X	33	21	100001				
Y	34	22	100010				
Z	35	23	100011				

MT	J	U	S	T	I	N	B	A	I	L	E	Y
DEC	19	30	28	29	18	23	11	10	18	21	14	34
HEX	13	1E	1C	1D	12	17	0B	0A	12	15	0E	22

```
>00:8FF1: B9 9A 69   LDA $699A,Y @ $699A = #$13
```

GAME DESIGN

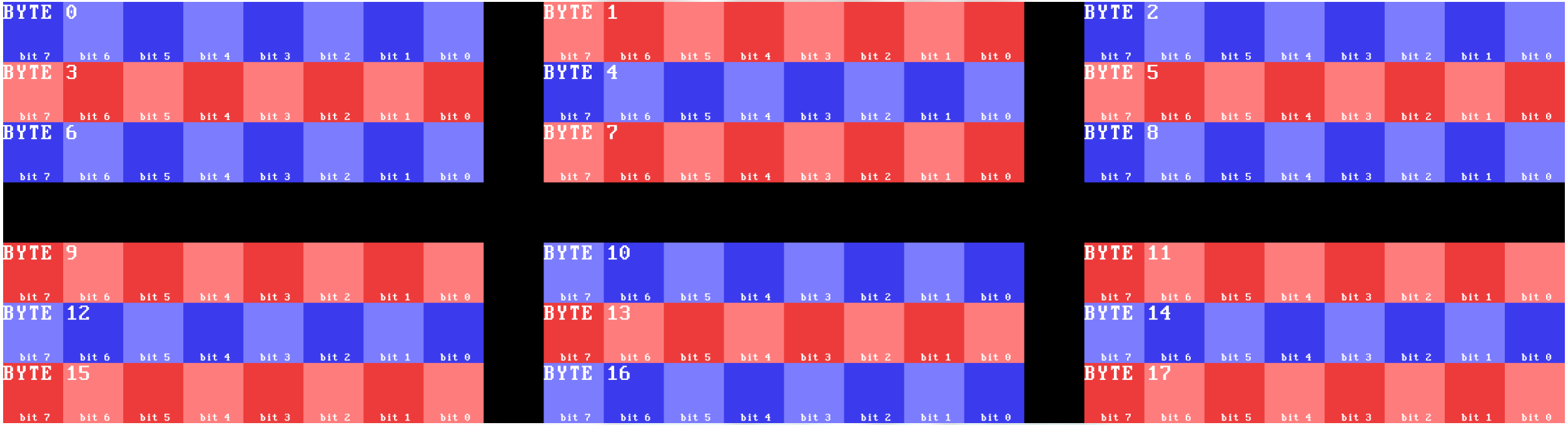
Metroid



- References
  - Metroid Password Format Guide by John David Ratliff
  - Metroid password tool by Alex Rasmussen / truepeacein.space

KNOX GAME DESIGN

Metroid



- Byte order (Endianness)
  - Little endian (least significant byte first)
  - Big endian (most significant byte first)
- Binary is usually written
  - most significant bit (MSB) on the left
  - least significant bit (LSB) on the right

- Encode in sets of 3 bytes
  - Each row above becomes 4 “Metroid” characters
- Similar to base64 encoding
  - Different character set values, rules for filling unused bits

KNOX GAME DESIGN

Metroid – password encoding

- Missile count
  - starts at byte index 10
  - Takes a value 0x00 to 0xFF (0 to 255)
  - new missile drops won't appear until a missile container is collected
  - collecting a missile drop with reset missile count to maximum possible missile number
- Powerups
  - stored at byte index 9
  - Powerups can be represented as binary values (see byte table)
  - Powerups can be OR'ed together to enable multiple powerups

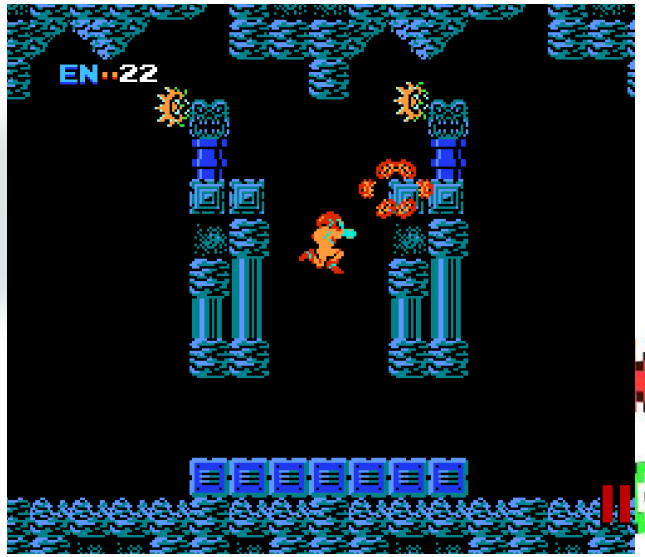
```
32 bytes = [0x00] * PASSWORD_SIZE_BYTES
33
34 MISSILE_BYTE = 10
35 bytes[MISSILE_BYTE] = 0x19  # 25 missiles
36
37 POWERUP_BYTE = 9
38 POWERUP_BOMBS_BIT = 0b00000001
39 POWERUP_HIGHJUMP_BIT = 0b00000010
40 POWERUP_LONGBEAM_BIT = 0b00000100
41 POWERUP_SCREWATTACK_BIT = 0b00001000
42 POWERUP_MARUMARI_BIT = 0b00010000
43 POWERUP_VARIA_BIT = 0b00100000
44 POWERUP_WAVBEAM_BIT = 0b01000000
45 POWERUP_ICEBEAM_BIT = 0b10000000
46 bytes[POWERUP_BYTE] = POWERUP_ICEBEAM_BIT | POWERUP_SCREWATTACK_BIT | PO
WERUP_HIGHJUMP_BIT
47
48 setChecksum(bytes)
49
50 for i in range(6):
51   bytesToMetroidAlphabet([bytes[(3 * i)], bytes[(3 * i) + 1], bytes[(3 *
 i) + 2]])
```

Example: start with 25 missiles, ice beam, screw attack, and high jump

```
python metroid.py
000000 000000
YXa000 00002Z
```

Run the script

Enter password in game



Verify it works

KNOX GAME DESIGN

Metroid

```
def setChecksum(bytes):
  CHECKSUM_BYTE = 17
  checksum_i = 0
  for i in range(CHECKSUM_BYTE):
    checksum_i += bytes[i]
  print(f"checksum: {checksum_i}")
  bytes[CHECKSUM_BYTE] = checksum_i & 0xFF
```

Calculate checksum
Limit to last 8 bits using bitwise AND 255

```
def bytesToMetroidAlphabet(bytes):
  chars = ['', '', '', '']

  part1 = (bytes[0] & 0b11111100)
  chars[0] = (part1 >> 2)

  part1 = (bytes[0] & 0b00000011)
  part2 = (bytes[1] & 0b11110000)
  chars[1] = (part1 << 4) | (part2 >> 4)

  part1 = (bytes[1] & 0b00001111)
  part2 = (bytes[2] & 0b11000000)
  chars[2] = (part1 << 2) | (part2 >> 6)

  chars[3] = bytes[2] & 0b00111111

  print("".join(METROID_ALPHABET[x] for x in chars))
```

Convert to Metroid characters

```
ENERGY_TANKS = [
                { "byte": 0, "bit": 0b00010000 },
                { "byte": 1, "bit": 0b00010000 },
                { "byte": 1, "bit": 0b00000010 },
                { "byte": 3, "bit": 0b01000000 },
                { "byte": 4, "bit": 0b00010000 },
                { "byte": 5, "bit": 0b00100000 },
                { "byte": 5, "bit": 0b00000100 },
                { "byte": 6, "bit": 0b00000001 },
              ]

for tank in ENERGY_TANKS:
  bytes[tank["byte"]] |= tank["bit"]
```

List of dicts to hold energy tank byte / bit locations

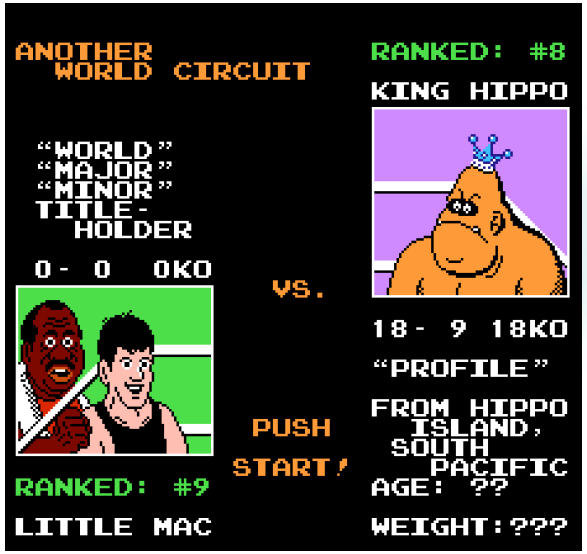
```
METROID_ALPHABET = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwx
yz?-'
```

Metroid character array (lookup)

KNOX GAME DESIGN

Punch Out

- Password for each of the completed circuits
- Also password for accessing circuit not accessible through regular game



KNOX GAME DESIGN

Other NES games

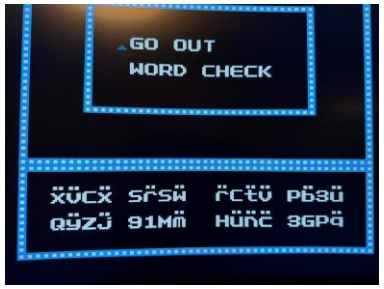
Metal Gear
32 characters, 25 spaces

Kid Icarus
64 characters, 24 spaces

Valkyrie no Bouken
36 characters, 18 spaces

Clash at Demonhead
32 characters, 30 spaces

Faxanadu
64 characters, 32 spaces
(not all spaces required)



Guardian Legend
64 characters, 32 spaces

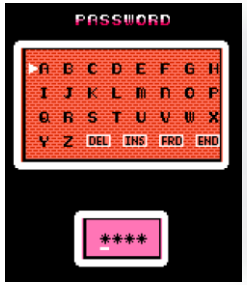
Willow
64 characters, 16 spaces

Ironsword: Wizards and Warriors II
26 characters, 5 spaces

Castlevania III
4 characters, 16 spaces
(hashed with player name)

Bubble Bobble
10 characters, 5 spaces

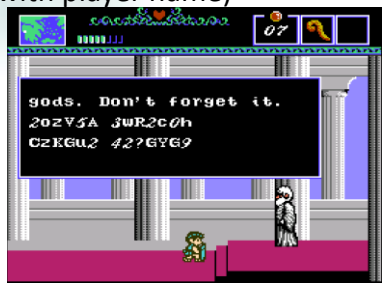
Swords and Serpents
password for each character
and game password



Adventure of Lolo
26 characters, 4 spaces

Monster Party
70 characters, 9 spaces

Iron Tank
10 characters, 7 spaces



Battle of Olympus
64 characters, 26 spaces

KNOX GAME DESIGN

Combinatorics

- characters = n
- spaces = r
- Arrangements (sometimes also called “permutations” or “combinations”)
  - n^r
  - character repetition allowed
  - order is important
  - empty spaces not allowed
  - common in game passwords
- Permutation
  - nPr = n!/(n-r)!
  - character repetition is not allowed
  - order is important
- Combinations
  - nCr = n! / (n-r)!r!
  - character repetition is not allowed
  - order is not important

game	characters (n)	spaces (r)	arrangements	permutations	combinations
Bubble Bobble	10	5	1.00E+05	3.02E+04	2.52E+02
Adventure of Lolo	26	4	4.57E+05	3.59E+05	1.50E+04
Iron Tank	10	7	1.00E+07	6.05E+05	1.20E+02
Ironsword	26	5	1.19E+07	7.89E+06	6.58E+04
Mega Man 2	2	25	3.36E+07		
Castlevania III	4	16	4.29E+09		
Monster Party	70	9	4.04E+16	2.36E+16	6.50E+10
Mega Man 3	3	36	1.50E+17		
Valkyrie no Bouken	36	18	1.03E+28	5.81E+25	9.08E+09
Willow	64	16	7.92E+28	1.02E+28	4.89E+14
Metal Gear	32	25	4.25E+37	5.22E+31	3.37E+06
Metroid	64	24	2.23E+43	1.56E+41	2.51E+17
Kid Icarus	64	24	2.23E+43	1.56E+41	2.51E+17
Clash at Demonhead	32	30	1.43E+45	1.32E+35	4.96E+02
Battle of Olympus	64	26	9.13E+46	2.43E+44	6.02E+17
Faxanadu	64	32	6.28E+57	4.82E+53	1.83E+18
Guardian Legend	64	32	6.28E+57	4.82E+53	1.83E+18

```
1 million = 1,000,000 = 1.00E+06
1 billion = 1,000,000,000 = 1.00E+09
1 trillion = 1,000,000,000,000 = 1.00E+12
```

KNOX GAME DESIGN

NES password REST API tool

- Developed by me (ldsmith 2026)
- Attempt at creating a unified API for generating NES passwords
- Developed with FastAPI (Python)
- POST /game/<id>/password
  - First look up game ID with /game?name=<query>
  - <id> is the game cart ID
  - send parameters in POST data
  - parameters are different for each game
  - See /docs for POST body schema (parameters)
- Modular – more games can be added
  - New games added in /mods subdirectory by game name
  - main.py has to be updated to point at new game
- Can use Postman to test API