

# PoP for PC-98 file formats analysis by "zarazala\_9"

21 December 2021

This document contains the Japanese-to-English machine translation of file format documentation of *Prince of Persia* for PC-98, as published by "zarazala\_9". The original Japanese pages are still available at:

<http://zarala.g2.xrea.com/koneta/persia/persia.html>

They were formerly at:

[http://www.geocities.jp/zarazala\\_9/koneta/persia/persia.html](http://www.geocities.jp/zarazala_9/koneta/persia/persia.html)

To discuss this document:

<https://forum.princed.org/viewtopic.php?f=63&t=3901>

[info@popot.org](mailto:info@popot.org)

Analysis of <i>Prince of Persia</i> for PC-98 data.....	2
File position.....	4
Compressed data.....	6
Map data.....	9
Palette data.....	14
Font data.....	15
CHR file (map parts image).....	17
Character image.....	19
Single picture data.....	21
BGM data.....	24
User disk.....	29

<http://zarala.g2.xrea.com/koneta/persia/persia.html>

Top > Small material > Prince of Persia

## **Analysis of *Prince of Persia* for PC-98 data**

- File position (file retrieval)
- Compressed data
- Map data (MAP file)

• Graphic related

Palette data (PALET.DAT)

Font data (FONT.DAT/EFONT.DAT)

Map parts image (CHR file)

Character image

Single picture data

- BGM data (08/08/21 added)
- User disk (08/06/11 added)

[Files other than the above]

Sound effect data (EFC file) → Not investigated (format similar to BGM data?)

VCLIP.DAT → Uninvestigated

There are no plans to examine program files.

---

[Download Toolset \(for Windows\)](#) (Last updated: 28 March 2011)

The following tools are included.

datacut.exe	Extract data from a disk image on a file-by-file basis
makeimg.exe	Create a disk image by grouping each file
compress.exe	Compress and decompress data
chrconv.exe	Convert map part image file (* .CHR) to BMP
fontconv.exe	Convert font files (FONT.DAT, EFONT.DAT) to BMP
mapedit.exe	Map editor
patview.exe	Viewing character images
picconv.exe	Convert single picture data to BMP file

\* First, use datacut.exe to extract the file from the disk image.

\* Please use the data extracted / converted from the original version data (or modified / diverted data) only within your personal scope.

---

**Attention etc.**

As usual, it is the result of half-hearted investigation, so there may be mistakes.

Please note that the description of the expansion method (processing method) of image data etc. is "method when reading with a self-made tool etc." and may differ from the processing actually performed by the game.

Since the actual machine (PC-9801DA) is already unusable, analysis and confirmation are performed only with the emulator.

(In fact, the emulator is less dangerous even if the data is rewritten strangely, and it is convenient because it can dump memory etc.)

In principle, the value with \$ at the beginning is in hexadecimal, and the one without it is in decimal notation.

(However, it may be in hexadecimal notation even if \$ is not attached.)

When handling a value of 2 bytes or more, it is basically stored in little endian.

(It is stored in order from the lowest byte. In the case of "\$ 12345678", it will be "78 56 34 12" in order from the smallest address)

-----

Return

## File position

\* The contents of this page are for disk dumps and disk images with the same contents (so-called "solid format" such as TFD and XDF).  
(The address will be misaligned for images with headers.)  
Not applicable to other disk image formats.

I found the file list (probably) information, so I decided to retrieve the data in file units based on this information.

To retrieve the data I have a need to identify the storage location of the file, speaking the conclusion information of the file name and starting block position in, \$ 1000 to connection information of the block in the & \$ 400 to the two this If you refer to it, you can identify the storage location of the file and retrieve the data.

\* Note that the "block" here refers to the "area divided by (disk) \$ 400 (1,024) bytes".  
Block number 0 is \$ 0 to \$ 3FF,  
block number 1 is \$ 400 to \$ 7FF,  
block number 2 is \$ 800 to \$ CFF,  
block number 3 is \$ C00 to \$ FFF,  
...and so on.

(It may be called a logical sector, but I'm not familiar with the file system ...)

First, there is data showing the information of each file after \$ 1000 on the disk.  
(Hereafter, it will be called a file list)

Each file is composed of 16 bytes.

- The first 11 bytes (\$ 0 to \$ A) are the file name.
- The subsequent 3 bytes (\$ B to \$ D). Is unknown
- The remaining 2 bytes (\$ E to \$ F) represent the start block number (1 block = \$ 400 (1,024) bytes)

The file name is in the so-called 8.3 format, and the extension part is from the 9th byte onward (\$ 8 to \$ A).

If the first part (non-extension part) of the file name is less than 8 characters, it is filled with spaces (\$ 20).

The data does not include the period (.) In the extension part.

(In this site, the file name is written with a period such as "MUSIC.SYS" and "PALET.DAT".) In the

above file list, only the start position of the data is known, but the connection of data (block) and If you want to know the end of the file, refer to \$ 400 or later.

The data here is stored in units of 2 bytes

, and the block number that follows is written at the address of "\$ 400 + (block number \* 2)".

If it does not continue to the next block (at the end of the file

), it has a value greater than or equal to \$ FC00 (assumed to be unsigned).

At this time, the value obtained by subtracting \$ FC00 is the data size of the block.

However, in the case of \$ FC00, the entire block is the data part, not 0 bytes.

Example: In the case of END2.MUS The contents of the disk from \$ 11D0 to DA are "45 4E 44 32 20 20 20 20 4D 55 53", but if this is converted into a character string, it will be "END2 MUS". (4 spaces between "END2" and "MUS") Since the value of \$ 11DE to DF is " 4 1 00" (= \$ 0041), the data of END2.MUS starts from the block number \$ 41. You can see that there is. Next, if you refer to  $\$ 400 + (\$ 41 * 2) = \$ 482$ , it will be "\$ 8E", so you can see that the block is \$ 0041 → \$ 008E. Looking at it in the same way,  $\$ 400 + (\$ 8E * 2) = \$ 51C$  value → \$ 8F  $\$ 400 + (\$ 8F * 2) = \$ 51E$  value → From \$ 90, block \$ 8E → \$ 8F → \$ 90 You can see that the value of \$ 520 (corresponding to block \$ 90) is "B7 FE" (= \$ FEB7), which is larger than \$ FC00, so here (block \$ 90) is the final block of END2.MUS. Will be. The size of the final block will be \$ FEB7- \$ FC00 = \$ 2B7 (695) bytes.

Therefore,

- Block \$ 41 (\$ 10400 ~ \$ 107FF)
- Block \$ 8E (\$ 23800 ~ \$ 23BFF)
- Block \$ 8F (\$ 23C00 ~ \$ 23FFF)
- \$ 2B7 (695) bytes from the beginning of block \$ 90 (\$ 24000 ~ \$ 242B6)

The data of END2.MUS is obtained by connecting the above.

As you can see from the above example, please note that the data arrangement is not always continuous.

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_compress.html](http://zarala.g2.xrea.com/koneta/persia/persia_compress.html)

Top > Small material > Prince of Persia > Compressed data

## Compressed data

Some files are compressed and saved.

Here, I will explain the compressed data applied to the map file.

\* The compression format explained here is used for MAP files and some image files, but some files may be compressed in other formats.

The data is basically a repetition of

[Compression pattern number] and [Parameter]

, and the expansion method differs depending on the pattern number, and

the size and expansion size of the parameter part also change depending on the pattern number.

(Parameter part: 0 to 3 bytes Expansion size: Multiple of 4) When

expanding a compressed file,

1. Read the pattern number

2. Read the parameter part that follows (may not exist)

3. The data is expanded from the compression pattern and parameters and written to the expansion destination buffer.

Return to 4.1 (read the next compression pattern)

, and so on.

When you reach the end of the file, you are done.

(There seems to be no such thing as an identifier to indicate the end)

### Deployment method

I will explain the decompression method for each compression pattern.

-Values are basically expressed in hexadecimal.

• Numbers with [] indicate pattern numbers, and aa bb indicates parameter parts.

-Numbers with \* such as "\* 1" indicate that any value can be entered in the \* part.

-For "previous ○ byte ~" and "○ byte before ~", the ○ byte before the expansion destination address is referred to (for the expanded data).

#### • 00 (uncompressed)

Before deployment	After deployment
[00] aa bb cc dd	aa bb cc dd

#### • \* 1, \* 2 (repeated (copy) system)

Before deployment	After deployment
[01]	(Copy the previous 4 bytes)
[02] aa	aa aa aa aa
[11] aa	(Write the previous 4 bytes repeatedly (aa + 1) times)
[21] aa bb	(Write the previous 4 bytes repeatedly (bbaa + 1) times)
[81]	(Copy 4 bytes from 8 bytes before)

[91] aa	(Repeat the previous 8 bytes in 4 byte units (aa + 1) times)
[a1] aa bb	(Repeat the previous 8 bytes in 4 byte units (bbaa + 1) times)

Example:

When the front 8 bytes (after expansion) is 01 02 03 04 05 06 07 08,

[01] → 05 06 07 08

[11] 01 → 05 06 07 08 05 06 07 08

[81] → 01 02 03 04

[91] 01 → 01 02 03 04 05 06 07 08

[91] 02 → 01 02 03 04 05 06 07 08 01 02 03 04

\* Regarding [81] [91] [a1],

1. "8 bytes before the compression pattern comes"

2. It is unclear which method is used, "8 bytes before the expansion destination address", but I think that there is no problem because the expansion results should be the same for both.

• \* 3, \* 4

Before deployment	After deployment
[03] aa bb	aa bb bb bb
[04] aa bb	aa aa bb bb
[13] aa bb	aa bb aa aa
[14] aa bb	aa bb aa bb
[23] aa bb	aa aa bb aa
[24] aa bb	aa bb bb aa
[33] aa bb	aa aa aa bb
[44] aa bb cc	aa aa bb cc
[54] aa bb cc	aa bb aa cc
[64] aa bb cc	aa bb cc aa
[74] aa bb cc	aa bb bb cc
[84] aa bb cc	aa bb cc bb
[94] aa bb cc	aa bb cc cc

• \* 5, \* 6

Before deployment	After deployment
[* 5] ab cd	*b *a *d *c
[*6] ab cd	b* a* d* c*

Example:

[05] 12 34 → 02 01 04 03

[15] 12 34 → 12 11 14 13

.....

[F5] 12 34 → F2 F1 F4 F3

[06] 12 34 → 20 10 40 30

[16] 12 34 → 21 11 41 31

.....

[F6] 12 34 → 2F 1F 4F 3F

• \*7~\*a

\* The value entered in the "-" part differs depending on the lower digit (lower 8 bits) of the pattern number.

\* 7 = 00

\* 8 = ff

\* 9 = Value before 4 bytes

\* a = value before 8 bytes

Before deployment	After deployment
[0*] aa	-- -- -- aa
[1*] aa	-- -- aa --
[2*] aa	-- aa -- --
[3*] aa	aa -- -- --
[4*] aa bb	-- -- aa bb
[5*] aa bb	-- aa -- bb
[6*] aa bb	-- aa bb --
[7*] aa bb	aa -- -- bb
[8*] aa bb	aa -- bb --
[9*] aa bb	aa bb -- --
[a*] aa bb cc	-- aa bb cc
[b*] aa bb cc	aa -- bb cc
[c*] aa bb cc	aa bb -- cc
[d*] aa bb cc	aa bb cc --

Example:

[57] 44 55 → 00 44 00 55

[58] 44 55 → ff 44 ff 55

When the front 8 bytes (after expansion) is 01 02 03 04 05 06 07 08,

[59] 44 55 → 05 44 07 55

[5a] 44 55 → 01 44 03 55

• F7, f8 (fixed value)

Before deployment	After deployment
[f7]	00 00 00 00
[f8]	ff ff ff ff

Return



[http://zarala.g2.xrea.com/koneta/persia/persia\\_map.html](http://zarala.g2.xrea.com/koneta/persia/persia_map.html)

Top > Small material > Prince of Persia > Map data

## Map data

The MAP file (LEV00.MAP to LEV15.MAP) contains map data.

There are still some mysterious parts, but here I will list only the known parts.

-The MAP file is compressed. The addresses listed below are after expansion. See here for how to decompress compressed data .

### Basics

- The size of the map data (after expansion) is \$ 1400 (5,120) bytes
- One side consists of 24 rooms (24 screens), and each room consists of 10 horizontal squares \* 3 vertical squares of map parts  
(and each map) The parts are made up of a combination of small parts)
- Room numbers are assigned from 1 (not from 0) in order from the beginning of the data
- Are there any specific surface-specific devices in the map data?  
(Is it impossible for the program to decide on its own?)

- Correspondence between faces and files (12 faces are divided into 4)

file name	Level (face)	CHR file (small parts image)	PLATE file (device floor image)
LEV00.MAP	Demo screen	LEV01.CHR	PLATE1.DAT
LEV01.MAP ~ LEV03.MAP	1st to 3rd		
LEV04.MAP ~ LEV06.MAP	4 to 6	LEV04.CHR	PLATE4.DAT
LEV07.MAP ~ LEV09.MAP	7-9	LEV07.CHR	PLATE7.DAT
LEV10.MAP ~ LEV11.MAP	10 to 11	LEV10.CHR	PLATEA.DAT
LEV12.MAP	12th page (first half)	LEV12.CHR	PLATEC.DAT
LEV13.MAP	12 faces (after uniting with the alter ego)		
LEV14.MAP	12 sides (passage to the last boss)	LEV14.CHR	PLATEE.DAT
LEV15.MAP	12th page (last boss battle)	LEV15.CHR	

### data structure

\$ 0- \$ 7FF	Combination of parts (small → medium)
\$ 800- \$ 9FF	Combination of parts (medium → large)
\$ A00- \$ AFF	Map part attributes
\$ B00- \$ DCF	Map structure
\$ DD0- \$ 109F	Parameters of each cell?

\$ 10A0- \$ 129F	Switch settings
\$ 12A0- \$ 12FF	Map (room) connection
\$ 1300- \$ 133F	?? ?? ??
\$ 1340- \$ 1342	Start position setting
\$ 1343- \$ 1346	?? ?? ??
\$ 1347- \$ 135E	Initial position of enemies in each room
\$ 135F- \$ 13A6	?? ?? ??
\$ 13A7- \$ 13BE	Enemy thinking pattern
From \$ 13BF	?? ?? ??

#### (Regarding the combination of parts)

The CHR file contains small parts of 16 \* 16 dots (hereinafter referred to as "small parts").

First, combine these 4 \* 4 pieces (16 pieces in total) vertically and horizontally to make a medium-sized part (medium part) with 64 \* 64 dots.

Two of these middle parts are lined up vertically, and 64 \* 128 dots (\* 2 layers) parts (large parts) that are layered on two layers (front and back) are actually placed on the map as "map parts".  
increase.

#### • Combination from small parts to medium parts (\$ 0- \$ 7FF: \$ 800byte)

Create (register) one medium part by combining the specified 16 (4 \* 4) small parts.

The data is 16 bytes per middle part, and the small part numbers are stored horizontally from the upper left in 1 byte units.

(Specify the small part number from 256 pieces of \$ 00 to \$ FF in the order of storing CHR files.)

Up to 128 pieces (\$ 00 to \$ 7F) can be created.

#### • Combination from medium parts to large parts (\$ 800- \$ 9FF: \$ 200byte)

Create a large part (map part) by combining the medium parts created in the above item into two layers vertically.

Data from \$ 800 is for the lower layer (back), and data for \$ 900 is for the upper layer (front).

The upper middle part number is stored in the 1st byte and the lower middle part number is stored in the 2nd byte in units of 2 bytes for each large part.

You can also create up to 128 large parts (map parts) from \$ 00 to \$ 7F.

	Large parts \$ 00	Large parts \$ 01	...	Large parts \$ 7F
Lower layer (back)	\$ 800	\$ 802	...	\$ 8FE
	\$ 801	\$ 803		\$ 8FF
Upper layer (front)	\$ 900	\$ 902	...	\$ 9FE
	\$ 901	\$ 903		\$ 9FF

#### • Map part attributes (\$ A00 to \$ AFF: \$ 100byte)

Each part is stored in units of 2 bytes, but the 2nd byte (upper 16 bits) is unknown.

##### ○ Basic

+ \$ 01	floor
\$ 02	Wall
+ \$ 04	Torch (flame animation display)

\* Floor (+1) and torch attribute (+4) can be specified at the same time.

It is also possible to specify it at the same time as the subsequent devices.

##### ○ Devices / Objects

\* Normally, the floor attribute is also set for those marked with \*. (Add +1 to the value)

\$ 08	Crumbling floor
\$ 10 *	Switch floor (open)
\$ 18 *	Switch floor (closed)
\$ 20 *	Iron bars
\$ 28 *	spike
\$ 30 *	Recovery agents
\$ 38 *	Damage medicine
\$ 40 *	Physical strength UP medicine
\$ 48 *	Screen reversing drug
\$ 50 *	Floating drug
\$ 58 *	guillotine
\$ 60 *	sword
\$ 68 *	mirror
\$ 70 *	Doorway
\$ 78 *	skeleton
\$ 80	??
\$ 88	Background star

\* The iron grill and the doorway will be installed on the right side of the corresponding square.

\* \$ 80 is set on the square above the bar, but details are unknown.

#### • Map structure (\$ B00 ~ \$ DCF: \$ 2D0byte)

The map part with the number specified here will be placed.

The data is arranged in the horizontal direction from the upper left in order from the beginning of the data, and this continues for 10 squares in the horizontal direction \* 3 steps in the vertical direction \* 24 rooms.

#### • Parameters of each cell (\$ DD0 ~ \$ 109F: \$ 2D0byte)

Parameter information such as gimmicks is stored.

(The data storage order is the same as the map structure)

Switch floor	Switch number to operate
Iron bars	If it is 01, it will start in the raised (open) state.
Other gimmicks (guillotine, etc.)	Doesn't it work if it's not 00?

#### • Switch setting (\$ 10A0- \$ 129F: \$ 200byte)

The position of the target bar when the switch is pressed is stored.

The data is divided into two parts, \$ 10A0 to \$ 119F and \$ 11A0 to \$ 129F, and each corresponds to the switch number \$ 00 to \$ FF in 1-byte units.

(For example, in the case of switch number \$ 05, the target position is determined from the values of \$ 10A5 and \$ 11A5.)

The value contains multiple information in bit units as shown in the table.

The value contains multiple information in 8H units as shown in the table.								
	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
From \$ 10A0	Simultaneous start flag	Lower 2 bits of room number		Target position in the room (mass unit)				
From \$	Upper 3 bits of room number			unused?				

11A0	(bit4 ~ 2)	
------	------------	--

If this bit is 0 (note that it is not 1), the "simultaneous start flag" will also start the switch with the next number at the same time.

(That is, you can move multiple bars with one switch.)

If the simultaneous start flag of that switch is 0, the next switch will also be started, but if you try to move 6 or more at the same time, the game will behave. It will be strange.

#### • Map (room) connection (\$ 12A0- \$ 12FF: \$ 60byte)

The destination (room number) when you come to the edge of the screen is stored.

The destinations for moving to the left, right, up, and down are stored in order from the 1st byte in units of 4 bytes for each room.

(If you do not specify the destination, it will be 0.)

You cannot specify the number of the room itself. (The game will stop)

In addition, since the destination can be specified separately for each room, it is also possible to make a twisted map configuration.

(Go to the same room no matter which direction you go, go to another room when you return to the direction you came, etc.)

\* This is not the case in the actual game (data originally included), and images of all rooms

You can make one big map by connecting .

#### • Start position setting (\$ 1340- \$ 1342: \$ 3byte)

\$ 1340	room number
\$ 1341	Position (mass unit)
\$ 1342	Orientation (\$ 00 = rightward \$ FF = leftward) However, the opposite is true on the 1st and 13th sides. (Because there is no turning motion at the start)

#### • Initial position of enemy in each room (\$ 1347- \$ 135E: \$ 18byte)

The initial position (mass unit) of the enemy character is stored in the order of the room number.

(OFF for \$ 1F)

(As you can see from this data structure, only one enemy character can be placed in one room)

#### • Enemy thinking pattern (\$ 13A7 ~ \$ 13BE: \$ 18byte)

Specifies the enemy's thinking pattern.

(The data storage order is the same as the initial position of the enemy, in the order of the room number.)

This value changes the movement (strength) of the enemy.

#### Attention etc.

- Due to the graphics, the judgment of the terrain such as floors and walls is shifted to the right by half a square in the game.  
The judgment (attribute) of the rightmost cell of the connected room is applied to the half cell on the left edge of the screen.  
(If the left side is not connected to any room (00), it will be treated as a wall.)
- If the collapsing floor falls or you get an item, the part number of that square will be incremented by +1.  
Therefore, set the next number of these parts to the same graphic with the corresponding attribute turned off.  
(Even if the graphic is different, the image of that square will not be updated until the screen is switched)

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_pal.html](http://zarala.g2.xrea.com/koneta/persia/persia_pal.html)

Top > Small material > Prince of Persia > Palette data

## Palette data

"PALET.DAT" contains the definition data of the analog palette.

The data of palettes 1 to 7 is stored in this file in units of 2 bytes.

1st byte: lower 4 bits → blue brightness (strength)

1st byte: upper 4 bits → red brightness

2nd byte: lower 4 bits → green brightness Will be.








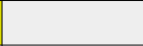
The brightness is specified in 16 steps from 0 to 15 (\$ F).

(Palette 0 is fixed to black (0,0,0))

The palettes actually used in the game are as follows.

The order itself is the same as the so-called digital 8 colors.

If you know the three primary colors of light (although the brightness is different), you should remember "Palette number +1 → blue, +2 → red, +4 → green".

Palette number	0	1	2	3	4	5	6	7
color								
Brightness (red, blue, green)	(0,0,0)	(0,0,9)	(A, 0,0)	(B, 0, B)	(0,8,0)	(0, B, D)	(D, D, 0)	(E, E, E)

\* Brightness is expressed in hexadecimal.

Return

## Font data

There are two font files, "FONT.DAT" and "EFONT.DAT".

file name	Use	Size (1 character)
FONT.DAT	Menus and messages (displaying remaining time, etc.)	16 * 16
EFONT.DAT	Ending staff roll	24 * 24

It has become.

The data represents 1 dot with 1 bit, and 1 byte contains information for 8 horizontal dots.  
The high-order bit corresponds to the dot on the left side and the low-order bit corresponds to the dot on the right side, and the part where the bit is 1 is drawn.

Example: For \$ 37 (00110111 in binary)

\$ 37 ↓ Convert to binary number ↓							
0	0	1	1	0	1	1	1

The data is stored horizontally from the upper left of the character.

The data size per character is ([horizontal size / 8] \* vertical size),  
which is \$ 20 (32) bytes for FONT.DAT and \$ 48 (72) bytes for EFONT.DAT.

In the font file, such data continues for the number of characters.

-Correspondence between coordinates and address

(The address is based on the beginning of the character (\$ 00))

- FONT.DAT (character size: 16 \* 16)

Coordinate	0-7	8-15
0	\$ 00	\$ 01
1	\$ 02	\$ 03
2	\$ 04	\$ 05
...		
15	\$ 1E	\$ 1F

- EFONT.DAT (Character size: 24 \* 24)

Coordinate	0-7	8-15	16-23
0	\$ 00	\$ 01	\$ 02
1	\$ 03	\$ 04	\$ 05
2	\$ 06	\$ 07	\$ 08
...			
23	\$ 45	\$ 46	\$ 47

--

Regarding the order of the stored characters,  
FONT.DAT is 0-9, AZ (uppercase only), ".", ",", "/", "←␣ (return key mark)". I am.  
EFONT.DAT is a bit special,

A B C D E F G H I J K L M O P S T Y a b c d e f g h i k l m n o p r s t u v w y z ( ) - .
---

And, instead of having both large and small alphabets, some letters are not included.  
(Probably only the characters used in the staff roll are included)

Return



Top > Small material > Prince of Persia > CHR file

## CHR file overview

The CHR file contains 256 16 \* 16 dot characters,  
which are combined into a single map part.  
(To be exact, make a medium-sized part (medium part) of 64 \* 64 dots combined vertically and  
horizontally 4 \* 4, and make a map part by combining this "medium part")

\* CHR files are compressed. The addresses written after that are the ones after expansion. Please refer to the compressed data page for how to decompress the compressed data.

Data in units of \$ 20 bytes becomes data for one character in 4 sets (\$ 80 bytes). Similar to font data, the data in each set is stored in the horizontal direction from the upper left of the character in a format that represents 8 horizontal dots in 1 byte (each dot corresponds to bit unit).

From \$ 00	Palette number +1
From \$ 20	Palette number +2
From \$ 40	Palette number +4
From \$ 60	Drawing flag (described later)

Example: \* Binary notation in parentheses

address	value
\$ 00	\$ 84 (10000100)
\$ 20	\$ C4 (11000100)
\$ 40	\$ 7F (01111111)
\$ 60	\$ C4 (11000100)

[illegible]

Drawing flag	\$ C4 (11000100)	1	1	0	0	0	1	0	0
Display color		3 (101)	6 (110)	4 (100)	4 (100)	4 (100)	7 (111)	4 (100)	4 (100)

### About drawing flag

\* I think that the following explanation is probably not accurate.

In the CHR file, in addition to the palette number information (3 bits), another bit of information (flag) is included for each dot.

The map part is divided into two layers (upper layer and lower layer), but the meaning of this flag also changes depending on the layer.

Upper layer	Flag 0	Performs OR calculation with the palette number of the dots in the lower layer (depending on the flag in the lower layer for the back and front)
	Flag 1	Draws in the foreground (excluding flames)
Underlayer	Flag 0	Drawn in the back
	Flag 1	In the wall attribute square, all characters other than the upper layer and flame are drawn , and in the floor attribute, it is drawn in front of a specific character.

The "specific character" of flag 1 in the lower layer is:

- A player or enemy who is falling
- The switch floor, the "part that sticks out to the right square" of the falling floor, etc.

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_chr.html](http://zarala.g2.xrea.com/koneta/persia/persia_chr.html)

Top > Small material > Prince of Persia > Character image

## Character image

Here, we will explain the following files.

(These files are all in the same format except for the difference between compressed and uncompressed)

FIRE.DAT	Uncompressed	Map background flame animation
CHTAB*.DAT	compression	Character image
PLATE*.DAT	compression	Image of collapsing floor, switch floor
TRAP.DAT	compression	Image of gimmicks

\* The following text is for uncompressed (decompressed) data.

### data form

- Header part (address \$ 00 ~)

The first 2 bytes contain the number of characters (patterns) stored in the file.

After that, the value indicating the start position of each character data continues in units of 4 bytes for the number of characters.

This is not a 4-byte (32-bit) value, but is divided into 2-byte values x 2, where the first 2 bytes represent the lower 4 bits of the address and the last 2 bytes represent the higher bits.

Therefore, the start address can be calculated from

(the value of the first 2 bytes & \$ F) + (the value of the last 2 bytes \* 16)

. ("&" Is an AND operation, in this case only the lower 4 bits are acquired)

-Character data (from the address written in the header part)

The first 2 bytes store size information.

1st byte --Horizontal size (\* 8 for this value is the number of dots in the horizontal direction)

2nd byte --Vertical size (= number of dots in the vertical direction)

After that, the data itself continues, but the

discriminant value (1 byte) → data part (none in some cases) → discriminant value → ... is repeated.

The discriminant value is a value that indicates whether it is "output according to the data that follows" or "output blank data" and the amount of data. (See below)

Since there is no data that indicates the end of character data, the end is judged based on whether or not the data for the image size is output after outputting the data according to the discrimination value .

Please note that the storage direction of image data is different from CHR files, etc., and the data in horizontal 8 dot units continues in the **"vertical direction"** .

The discriminant value depends on whether the most significant bit is set (greater than or equal to \$ 80 or less).

[If the discriminant value is less than \$ 80]

This is followed by 4 bytes of data (n + 1) times. (N = discrimination value)

The data represents 8 dots horizontally at one time (4 bytes), and each 1 byte has the following information.

(Similar to CHR files, each bit corresponds to 1 dot)

- 1st byte - Palette number + 1
- 2nd byte - Palette number +2
- 3rd byte - Palette number +4
- 4th byte - Drawing flag (If 0, isn't it drawn?)

[When the discrimination value is \$ 80 or more]

Blank data (equivalent to 00 00 00 00?) Is repeated (n- \$ 80 + 1) times.

(In this case, there is no data part)

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_pic.html](http://zarala.g2.xrea.com/koneta/persia/persia_pic.html)

Top > Small material > Prince of Persia > Single picture data

## Single picture data

This is an image file that is mainly used when not playing, such as at the opening.  
The following files are applicable.

TITLE*.DAT
title
JAFFER*.DAT
opening
OPEN*.DAT
Crystal ball part of the opening
JAF_*.DAT
Opening Jafar's face anime
PROOM.DAT
Princess room
MOYOU.DAT
Menu screen
ENDING*.DAT
Ending (ENDING6.DAT is a little special, details will be described later)

### data form

The first 4 bytes are size information. (2 bytes each for horizontal and vertical)

\$ 00 - Horizontal size (\* 8 for this value is the number of dots in the horizontal direction)

\$ 02 - Vertical size (= number of dots in the vertical direction)

It has become.

After that, "pattern number" and "parameter" are repeated like compressed data.

Basically, one pattern represents 8 dots horizontally, and this continues in the horizontal direction.

In other image files:

-Each bit in 1 byte corresponds to 1 dot

-There are multiple data such as ↑, and each bit corresponds to each bit of palette number (3 bits) and drawing flag (1 bit). However, even with this single image data, data in the same format can be obtained by expanding the data according to the pattern number.

(Since there is no data that corresponds to the drawing flag in a single picture, the data that can be obtained is 3 bytes (8dot \* 3bit).)

\* This means that "data can be obtained in the format of ↑", and it is actually in this format. It is unknown whether it is expanded (converted) and processed.

I haven't tried all of the patterns from \$ 00 to \$ FF, but the ones I have confirmed for the time being are as follows.

-Values are basically expressed in hexadecimal.

• Numbers with [] indicate pattern numbers, and aa bb indicates parameter parts.

-Numbers with "\*" such as "\* 7" indicate that any value can be entered in the "\*" part.  
 -If there is a "/" like [04/05], it means that it is the expansion result of both patterns.  
 ([\* 4] and [\* 5] have the same value except for the "-" part)

### Normal writing

Pattern number	Palette number		
	+1	+2	+4
[01] aa	aa	aa	aa
[02] aa bb	aa	aa	bb
[12] aa bb	aa	bb	aa
[22] aa bb	aa	bb	bb
[03] aa bb cc	aa	bb	cc
[04/05] aa	-	-	aa
[14/15] aa	-	aa	-
[24/25] aa	aa	-	-
[34/35] aa bb	-	aa	bb
[44/45] aa bb	aa	-	bb
[54/55] aa bb	aa	bb	-
[* 6]	All 8 dots become palette * numbers		

\* In the case of \* 4, the "-" part is always \$ 00

\* \* In the case of 5, the "-" part is always \$ FF

### Repeat / copy system

[00] aa ...	Repeat the following pattern (aa + 1) times
[* 0] ... (* is other than 0)	Repeat the following pattern (* + 1) times
[* 7] aa	(* + 1) Copy (aa × 8) dot from the line * Correction: Previously it was written as "(aa + 1) * 8 dot", but it is correct as above (2011/03/21)

\* In these patterns, it is not possible to specify that the right edge is exceeded. (An error will occur.)

\* Duplicate (multiple) cannot be specified for the repeat system. (The value specified last is valid.)

\* "Copy from ○ line" is ignored even if it is specified repeatedly. (Always executed only once)

Example: [12] For 7F AA (→ \$ 7F / \$ AA / \$ 7F)

Palette number bit0 (+1)	\$ 7F (01111111)		+1 (001)	+1 (001)	+1 (001)	+1 (001)	+1 (001)	+1 (001)	+1 (001)
Palette number bit1 (+2)	\$ AA (10101010)	+2 (010)		+2 (010)		+2 (010)		+2 (010)	
Palette number bit2 (+4)	\$ 7F (01111111)		+4 (100)	+4 (100)	+4 (100)	+4 (100)	+4 (100)	+4 (100)	+4 (100)
Display color		2 (010)	5 (101)	7 (111)	5 (101)	7 (111)	5 (101)	7 (111)	5 (101)

### About ENDING6.DAT

Only ENDING6.DAT contains multiple (12) data in one file.

At the beginning of the file, 12 start addresses (2 bytes) of each data follow, followed by the data body.

\$ 00	Address 1
-------	-----------

\$ 02	Address 2
...	...
\$ 16	Address 12
From \$ 18	Data 1
	Data 2
	...
	Data 12

\* Files other than ENDING6.DAT are only the part corresponding to "Data 1".

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_mus.html](http://zarala.g2.xrea.com/koneta/persia/persia_mus.html)

Top > Small material > Prince of Persia > BGM data

※ There are some parts that are not known yet, but for the time being, only the ones that have been found are listed.

I have little knowledge about FM synthesis, so I may have written something wrong.

## BGM data

This file contains music data. The extension is "MUS".

The following files are applicable.

MUSIC1.MUS	LEVEL4 ~ 6,10 ~ 11 and demo side
MUSIC2.MUS	LEVEL1 ~ 3,7 ~ 9
MUSIC3.MUS	LEVEL12-3
MUSIC4.MUS	LEVEL12-1 ~ 2
FIGHT.MUS	fight
FIGHTF.MUS	Battle (Last Boss)
FIGHTK.MUS	Battle (skeleton)
FIGHTS.MUS	Battle (alternate)
VICT1.MUS	Battle victory
EXIT.MUS	Song when the exit opens
CLEAR.MUS	clear
DEAD.MUS	Mistakes (falls, traps, etc.)
DEAD2.MUS	Miss (when killed by the enemy)
OPEN.MUS	Opening 1
ROOM1.MUS	Opening 2
ROOM2.MUS	Intermediate demo
END1.MUS	Ending 1
END2.MUS	Ending 2
END3.MUS	Ending 3
TIMEOUT.MUS	Time over
MENU.MUS	Menu screen
APPEAR.MUS	Last Boss Battle Intro
APPEAR2.MUS	Boss / Alternate Appearance
PICKUP.MUS	Get the sword
KUSURI.MUS	Medicine (recovery of physical strength)
POWUP.MUS	Medicine (upper limit of physical strength)

### data structure

It can be roughly divided as follows.

- performance information,
- phrases,
- and tone definition

Information corresponding to musical intervals, such as pitches, is stored in the phrase section (in phrase units).



The performance information section plays these phrases in the form of calling them (combining the phrases).

(For example, it consists of information such as "Play phrase A"-> "Repeat phrase B four times"-> "Play phrase A"-> "Return to first" .)

Performance information is for each part (channel). Each is independent.

#### • Performance information

<Header part>

Information on the start position of data and the channel to be used is stored first.

address	information
\$ 0	Tone information start address
\$ 4	Channel to use
From \$ 5	Performance information start address for each part (in 2byte units)

"Channel to use" is specified in bit units,

bit0 (+ \$ 1) ... SSG1

bit1 (+ \$ 2) ... SSG2

bit2 (+ \$ 4) ... SSG3

bit3 (+ \$ 8) ... FM1

bit4 (+ \$ 10) ... FM2

bit5 (+ \$ 20) ... FM3

It will be. In "Start address of each part", the information of the start address continues for the number of parts (= the number of channels used), but the channels that are not used at this time are skipped.

In other words, when using all channels

\$ 5 SSG1 start address

\$ 7 SSG2

\$ 9 SSG3

\$ B FM1

\$ D FM2

\$ F FM3

However, for example, if the value of \$ 4 is \$ 2A (only SSG2 and FM1,3 are used).

\$ 5 SSG2

\$ 7 FM1

\$ 9 FM3

It will be.

<Performance information>

The performance information of each part is stored from the addresses listed after \$ 5.

Basically, 3 bytes of data are continuous,

1st byte	Number of views (0 is an infinite loop?)
2nd and 3rd bytes	Address of the phrase to play (2 bytes)

It has become.

(For example, "02 40 01" will play the phrase stored in \$ 140 twice.)

As an exception,

If the first byte is \$ FE, it will move to the specified address (the playback position of the performance information). (Used when looping a song)

If the first byte is \$ FF, that part ends playback. (Only in this case, it is 1 byte data)

#### • Phrase

The data corresponding to the so-called musical score is stored in small pieces for each phrase.

One song is composed by combining these phrases.

The data itself is in text format (or MML format), and each data is separated by \$ 00.

(However, note that there are some differences from general MML data. )

#### -Main differences from general MML data

command	General MML	MUS data
T (tempo specification)	Number of quarter notes per minute	FM sound source Timer-B value (= affects the time per sound length)
Specifying the note length	1 bar n / 1 length (= note with ○)	Directly proportional to the specified value (length of note length 1 * specified value) * By the way, the L command does not exist, and the previous value is used as the note length omission value.
sharp	Attach to "behind" the specified pitch (example: C #)	Attach to "before" the pitch specification (example: #C)

#### -Explanation of each command

Any value can be entered in the {n} part.

command	meaning	Commentary
A to G {n}	Play the specified sound	n = note length (larger, longer, time per note length will be described later) corresponding to do-shi in the order of C, D, E, F, G, A, B Prefix with # (eg #C) to raise a semitone The note length default is the same length as the previous pronunciation
R {n}	rest	n = length (same as A to G)
&	Slur?	For the same pitch, such as "C & D & E", the part connected by & used between each pronunciation changes only the pitch without re-pronouncing (the volume envelope is not initialized and continues as it is). If the pitch is the same, it will be the same as Thailand. * If you want to continue the vibrato envelope, use [,] together.
[]	Continue vibrato at &	The part surrounded by [] does not initialize the vibrato envelope even if it becomes the next sound at the time of slur with &, and it continues as it is.
T {n}	Specify tempo	Details will be described later. * Note that this is not the general method of specifying tempo, "the number of quarter notes per minute", so note that the larger the value, the slower the tempo.
V {n}	Volume specification	Specify between 0 and 15
O {n}	Octave designation	
+	Raise one octave	
-	1 octave down	
^	Increase the volume by one level	
_	Decrease the volume by	

	one level	
) {n}	Pitch shift	Adds the values related to the pitch of the sound source register (from the original value) by the specified value.
!		(Used in APPEAR, ROOM1) Maybe to synchronize the production with BGM

With the T command, the value of (255-specified value) is written to \$ 27 (TIMER-B) of the sound source register.

Also, the time per note length = (T value + 1) \* 288 [μs].

(Ms (microseconds): one millionth of a second, 1/1000 of a millisecond (1/1000 seconds))

T command specified value is T, quarter note length is L, 4 minutes Assuming that the tempo of the note is T',

T and T' can be converted by the following formula.

$$T = (60000000 \div T' \div 288 \div L) - 1$$

$$T' = (60000000 \div (T + 1) \div 288 \div L)$$

### • Tone definition

Data is stored from the tone definition start address written in the header part (see above).

At the beginning, the definition data start position for each tone number is stored in 2 byte units.

(Since the tone number starts from 0, it continues from 0 → 1 → 2 ...)

The actual tone definition data starts from the address shown here, but the data content is different between SSG and FM.

\* The table below is only currently known. (There are still some unclear points.)

Also, I have not investigated the effective range of values.

\* The address is a relative address from the beginning of the data.

\* Please refer to other sites as each parameter of the sound source cannot be explained here.

### • SSG

address	data
\$ 00	Length of attack (rising of sound) (the larger the value, the longer)
\$ 01	Decay (attenuation) length
\$ 02	Volume change in attack / decay
\$ 03	Details are unknown Noise for \$ 80 or more (the way the sound sounds depends on the value)
\$ 04	Vibrato pitch change
\$ 05	Vibrato Pitch change cycle (speed of change)
\$ 06	Vibrato pitch change pattern It seems to be specified in bit units, but details are unknown
\$ 07	Time to start vibrato

### • FM

For each parameter other than vibrato, the value written to the sound source register is basically stored as it is.

address	data
\$ 00	Feedback (FB) / Algorithm (ALG)
\$ 01-04	Detune (DT) / Multiple (ML) (Operator (OP) 1,2,3,4 for each byte)
\$ 05-08	Total level (TL) (OP1-4)

\$ 09-0C	Key scale (KS) / Attack rate (AR) (OP1-4)
\$ 0D ~ 10	Decay rate (DR) (OP1-4)
\$ 11-14	Sustain rate (SR) (OP1-4)
\$ 15-18	Sustain Level (SL) / Release Rate (RR) (OP1-4)
\$ 19 ~ 1a	?? ?? ?? (not clear)
\$ 1b	Vibrato pitch change
\$ 1c	Vibrato Pitch change cycle (speed of change)
\$ 1d	Time to start vibrato
\$ 1e	?? ?? ?? (not clear)

The address where the two parameters are described is specified in bit units as shown below.

address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
\$ 00 (FB / ALG)			FB			ALG		
From \$ 01 (DT / ML)		DT			ML			
From \$ 09 (KS / AR)	KS			AR				
From \$ 15 (SL / RR)	SL				SR			

Return

[http://zarala.g2.xrea.com/koneta/persia/persia\\_user.html](http://zarala.g2.xrea.com/koneta/persia/persia_user.html)

Top > Small material > Prince of Persia > User disk

## User disk

Information on intermediate saves, the fastest recording of each side, and its replay data are recorded on the user disk.

### data structure

address	Data size	Explanation
\$ 0	2byte	Value of "01 00"
\$ 2	9byte	The string 'USER DISK' (55 53 45 52 20 44 49 53 4B)
\$ B	1byte	00
\$ C	4byte	User disk creation date and time
\$ 10	8byte * 10	Saved Data-Information 1
\$ 60	3byte * 10	Saved Data-Information 2
\$ 80	16byte * 12	Fastest record information
\$ 210	8byte * 10	Saved Data-Information 3
\$ 2000	8192 (\$ 2000) byte * 14	Replay data (fastest record)
\$ 20000	8192 (\$ 2000) byte * 2 * 10	Replay data? (Every save data?)

- For each item after \$ 10, the data will continue for the number of save data (the number of faces for the fastest recording).
- LEVEL 12 is actually divided into 3 parts, LEVEL 12-14, so there are 14 faces internally. (Actually, the map during the last boss battle is an independent map (LEVEL15), so to be exact, "LEVEL12 is internally 4 sides (15 sides in total)", but this map is a dedicated map called from LEVEL14, so it is actually There are 14 pages in total.

### User disk creation date and time

The date and time when the disk was initialized by "Create User Disk" are recorded.

This value is used to distinguish between discs, and if the value is different (= replaced with a different disc) when writing data for saving or fastest record update, a warning message will be displayed.

Date and time information is recorded bitwise as follows:

address	\$ C								\$ D								\$ E								\$ F							
bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
information	Year								Month				Day				Time				Minutes				Seconds							

\* Only the lower 6-bit part of the year (both are 0 in 2000 and 1964)

### Save data

Information on saves on the way is recorded.

It is roughly divided into 3 places, and data continues in each place as many as the number of save data (10 pieces).

\* In the following explanation, the address is \$ 0 at the beginning of each save data information.

- Information 1 (\$ 10 ~)

\$ 0	1byte	Level (face number)
\$ 1	1byte	life
\$ 2	1byte	Remaining time (minutes)
\$ 4	2byte	Remaining time (seconds)
\$ 6	1byte	Movement speed setting
\$ 7	1byte	Battle speed designation

If the level part is 0, the data will be treated as unregistered.

In addition, LEVEL12 is internally divided into three, LEVEL12-14, but in any case of 12-14, "12" is displayed in the game.

The second part of the remaining time is in 1/8 second units.

(1 = 1/8 seconds, 2 = 2/8 seconds, 8 = 1 second, 9 = 1 + 1/8 seconds (9/8 seconds) ...)

The speed setting is usually set from 1 to 9. You can, but if you rewrite this data directly to 0, the game speed will be faster than speed 1.

On the contrary, if it is set to 10 or more, the speed will be slower than 9.

- Information 2 (\$ 60 ~)

The time remaining when you start the face is recorded.

This is the standard value of the play time on each side, and the play time on that side is obtained by subtracting the remaining time at the time of clearing from this value, but this value is used (restart when saving). Only LEVEL 12 (with multiple points).

On the other side, it is based on the time remaining when it was last loaded.

(Other than LEVEL 12, if you load it and then save it again, it will change to the remaining time at the time of loading)

- Information 3 (\$ 210 ~)

The player name of each data is recorded.

#### Fastest record information (\$ 80 ~)

\$ 0	1byte	Hours (minutes)
\$ 2	2byte	Time (seconds)
\$ 4	8byte	name
\$ B	2byte	Presence / absence of replay data (Yes: FF FF, No: 00 00)
\$ E	1byte	life
\$ F	1byte	02 for LEVEL12 only, 00 for others (meaning to read replay data for n + 1 screens ?)

If the time is 00:00, there will be no recording (handled as blank data).

#### Fastest record replay data (\$ 2000 ~)

The fastest record replay of each side is recorded.

The data size per page is \$ 2000 bytes, which continues for the number of pages (14).

During play, if the replay data size exceeds \$ 2000 and the record is updated, the play time will be recorded but the replay data will not be saved.

For replay data, [key status] → [count] is repeated every 1 byte.

The "key state" is the sum of the values corresponding to the pressed key. (Refer to the table)

"Count" keeps the same key state for the specified count.

(Since it is 1 byte, the maximum is \$ FF, but if the key state is \$ 04 and it continues for a long time, it will be like "04 FF 04 FF 04 FF ...")

By the way, DEMOPLAY.KEY which is the key operation data of the demo screen. Has the same format.

• Key status

+ \$ 1	↑
+ \$ 2	↓
+ \$ 4	←
+ \$ 8	→
+ \$ 10	Space
+ \$ 20	Shift

Example:

↓ + → \$ 2 + \$ 8 = \$ A

When replay data is written, \$ 2000 bytes are written from the area where the replay data is recorded in the memory regardless of the play content.

Since this area is not cleared at the start of the surface, data that is not related to the replay has also been written to the disc.

(For this reason, it is difficult to tell how much replay data is.)

**Replay data for each save data? (\$ 20000 ~)**

I have not investigated in detail, but two replay data (what seems to be) are recorded (for each save data) for each save data.

Perhaps LEVEL 12 is divided into three sides, and it is an area for temporarily storing the replay data for the first two sides.

Return