Google

Code

Archive

puz - FileFormat.wiki

Export to GitHub

---

summary Detailed file format documentation

Table of contents:

Introduction

PUZ is a file format commonly used by commercial software for crossword puzzles. There is, to our knowledge, no documentation of the format available online. This page (and the implementations) is the result of a bit of reverse engineering work.

The documentation is mostly complete. Implementations based on this documentation seem to support, for example, all (or the vast majority of) New York Times puzzles. The few remaining unknown pieces are noted.

We have no real financial interest in this; it was just a fun hack.

File Contents

The file is laid out like this: 1. a fixed-size header with information like the width and height of the puzzle 1. the puzzle solution and the current state of the cells, with size determined by the puzzle dimensions described in the previous section 1. a series of NUL-terminated variable-length strings with information like the author, copyright, the puzzle clues and a note about the puzzle. 1.

optionally, a series of sections with additional information about the puzzle, like rebuses, circled squares, and timer data.

Header Format

Define a *short* to be a little-endian two byte integer. The file header is then described in the following table.

| **Component** | **Offset** | **End** | **Length** | **Type** | **Description** | |:--------------|:-----------|:--------|:-----------|:---------|:----------------| | Checksum | 0x00 | 0x01 | 0x2 | short | overall file checksum | | File Magic | 0x02 | 0x0D | 0xC | string | NUL-terminated constant string: 4143 524f 5353 2644 4f57 4e00 ("ACROSS&DOWN") |

The following checksums are described in more detail in a separate section below. | **Component** | **Offset** | **End** | **Length** | **Type** | **Description** | |:--------------|:-----------|:--------|:-----------|:---------|:----------------| | CIB Checksum | 0x0E | 0x0F | 0x2 | short | (defined later) | | Masked Low Checksums | 0x10 | 0x13 | 0x4 | | A set of checksums, XOR-masked against a magic string. | | Masked High Checksums | 0x14 | 0x17 | 0x4 | | A set of checksums, XOR-masked against a magic string. |

| **Component** | **Offset** | **End** | **Length** | **Type** | **Description** | |:--------------|:-----------|:--------|:-----------|:---------|:----------------| | Version String(?) | 0x18 | 0x1B | 0x4 | string | e.g. "1.2\0" | | Reserved1C(?) | 0x1C | 0x1D | 0x2 | ? | In many files, this is uninitialized memory | | Scrambled Checksum | 0x1E | 0x1F | 0x2 | short | In scrambled puzzles, a checksum of the real solution (details below). Otherwise, 0x0000. | | Reserved20(?) | 0x20 | 0x2B | 0xC | ? | In files where Reserved1C is garbage, this is garbage too. | | Width | 0x2C | 0x2C | 0x1 | byte | The width of the board | | Height | 0x2D | 0x2D | 0x1 | byte | The height of the board | | # of Clues | 0x2E | 0x2F | 0x2 | short | The number of clues for this board | | Unknown Bitmask | 0x30 | 0x31 | 0x2 | short | A bitmask. Operations unknown. | | Scrambled Tag | 0x32 | 0x33 | 0x2 | short | 0 for unscrambled puzzles. Nonzero (often 4) for scrambled puzzles. |

Puzzle Layout and State

Next come the board solution and player state. (If a player works on a puzzle and then saves their game, the cells they've filled are stored in the state. Otherwise the state is all blank cells and contains a subset of the information in the solution.)

Boards are stored as a single string of ASCII, with one character per cell of the board beginning at the top-left and scanning in reading order, left to right then top to bottom. We'll use this board as a running example (where # represents a black cell, and the letters are the filled-in solution). ``` C A T

A

R

```

At the end of the header (offset 0x34) comes the solution to the puzzle. Non-playable (ie: black) cells are denoted by '.'. So for this example, the board is stored as nine bytes: `CAT..A..R`

Next comes the player state, stored similarly. Empty cells are stored as '-', so the example board before any cells had been filled in is stored as: `---..-..-`

## Strings Section

Immediately following the boards comes the strings. All strings are encoded in ISO-8859-1 and end with a NUL. Even if a string is empty, its trailing NUL still appears in the file. In order, the strings are:

| Description | Example | |:------------|:--------| | Title | Theme: .PUZ format | | Author | J. Puz / W. Shortz | | Copyright | (c) 2007 J. Puz | | Clue#1 | Cued, in pool | | ... | ...more clues... | | Clue#n | Quiet | | Notes | http://mywebsite |

These first three example strings would appear in the file as the following, where \0 represents a NUL: `Theme: .PUZ format\0J. Puz / W. Shortz\0(c) 2007 J. Puz\0`

In some NYT puzzles, a "Note" has been included in the title instead of using the designated notes field. In all the examples we've seen, the note has been separated from the title by a space (ASCII 0x20) and begins with the string "NOTE:" or "Note:". It's not known if this is flagged anywhere else in the file. It doesn't seem that Across Lite handles these notes - they are just included with the title (which looks ugly).
```

The clues are arranged numerically. When two clues have the same number, the Across clue comes before the Down clue.

Clue Assignment

Nowhere in the file does it specify which cells get numbers or which clues correspond to which numbers. These are instead derived from the shape of the puzzle.

Here's a sketch of one way to assign numbers and clues to cells. First, some helper functions: ```

Returns true if the cell at (x, y) gets an "across" clue number.

def cell_needs_across_number(x, y): # Check that there is no blank to the left of us if x == 0 or is_black_cell(x-1, y): # Check that there is space (at least two cells) for a word here if x+1 < width and is_black_cell(x+1): return True return False

def cell_needs_down_number(x, y): # ...as above, but on the y axis ```

And then the actual assignment code: ```

An array mapping across clues to the "clue number".

So across_numbers[2] = 7 means that the 3rd across clue number

points at cell number 7.

across_numbers = []

cur_cell_number = 1

Iterate through th

for y in 0..height: for x in 0..width: if is_black_cell(x, y): continue

```
assigned_number = False
if cell_needs_across_number(x, y):
  across_numbers.append(cur_cell_number)
  cell_numbers[x][y] = cell_number
  assigned_number = True
if cell_needs_down_number(x, y):
  # ...as above, with "down" instead
if assigned_number:
  cell_number += 1
```

```

# Checksums

The file format uses a variety of checksums.

The checksumming routine used in PUZ is a variant of CRC-16. To checksum a region of memory, the following is used:

``` unsigned short cksum_region(unsigned char *base, int len, unsigned short cksum) { int i;

for (i = 0; i < len; i++) { if (cksum & 0x0001) cksum = (cksum >> 1) + 0x8000; else cksum = cksum >> 1; cksum += *(base+i); }

return cksum; } ```

The CIB checksum (which appears as its own field in the header as well as elsewhere) is a checksum over eight bytes of the header starting at the board width: `c_cib = cksum_region(data + 0x2C, 8, 0);`

The primary board checksum uses the CIB checksum and other data: ``` cksum = c_cib; cksum = cksum_region(solution, w*h, cksum); cksum = cksum_region(grid, w*h, cksum);

if (strlen(title) > 0) cksum = cksum_region(title, strlen(title)+1, cksum);

if (strlen(author) > 0) cksum = cksum_region(author, strlen(author)+1, cksum);

if (strlen(copyright) > 0) cksum = cksum_region(copyright, strlen(copyright)+1, cksum);

for(i = 0; i < num_of_clues; i++) cksum = cksum_region(clue[i], strlen(clue[i]), cksum);

if (strlen(notes) > 0) cksum = cksum_region(notes, strlen(notes)+1, cksum); ```

# Masked Checksums

The values from 0x10-0x17 are a real pain to generate. They are the result of masking off and XORing four checksums; 0x10-0x13 are the low bytes, while 0x14-0x17 are the high bytes.

To calculate these bytes, we must first calculate four checksums:

1. CIB Checksum:

```
c_cib = cksum_region(CIB, 0x08, 0x0000);
```

2. Solution Checksum:

```
c_sol = cksum_region(solution, w*h, 0x0000);
```

3. Grid Checksum:

```
c_grid = cksum_region(grid, w*h, 0x0000);
```

4. A partial board checksum: ``` c_part = 0x0000;

if (strlen(title) > 0) c_part = cksum_region(title, strlen(title)+1, c_part);

if (strlen(author) > 0) c_part = cksum_region(author, strlen(author)+1, c_part);

if (strlen(copyright) > 0) c_part = cksum_region(copyright, strlen(copyright)+1, c_part);

for (int i = 0; i < n_clues; i++) c_part = cksum_region(clue[i], strlen(clue[i]), c_part);

if (strlen(notes) > 0) c_part = cksum_region(notes, strlen(notes)+1, c_part); ```

Once these four checksums are obtained, they're stuffed into the file thusly:

``` file[0x10] = 0x49 ^ (c_cib & 0xFF); file[0x11] = 0x43 ^ (c_sol & 0xFF); file[0x12] = 0x48 ^ (c_grid & 0xFF); file[0x13] = 0x45 ^ (c_part & 0xFF);

file[0x14] = 0x41 ^ ((c_cib & 0xFF00) >> 8); file[0x15] = 0x54 ^ ((c_sol & 0xFF00) >> 8); file[0x16] = 0x45 ^ ((c_grid & 0xFF00) >> 8); file[0x17] = 0x44 ^ ((c_part & 0xFF00) >> 8); ```

Note that these hex values in ASCII are the string "ICHEATED".

# Locked/Scrambled Puzzles

The header contains two pieces related to scrambled puzzles. The short at 0x32 records whether the puzzle is scrambled. If it is scrambled, the short at 0x1E is a checksum suitable for verifying an attempt at unscrambling. If the correct solution is laid out as a string in column-major order, omitting black squares, then 0x1E contains cksum_region(string,0x0000).

## Scrambling Algorithm

The algorithm used to scramble the puzzles, discovered by Mike Richards, is documented in his comments below. Eventually, they will be migrated to the main body of the document.

# Extra Sections

The known extra sections are:

| **Section Name** | **Description** | |:----------------|:----------------| | GRBS | where rebuses are located in the solution | | RTBL | contents of rebus squares, referred to by GRBS | | LTIM | timer data | | GEXT | circled squares, incorrect and given flags | | RUSR | user-entered rebus squares |

In official puzzles, the sections always seem to come in this order, when they appear. It is not known if the ordering is guaranteed. The GRBS and RTBL sections appear together in puzzles with rebuses. However, sometimes a GRBS section with no rebus squares appears without an RTBL,
```

especially in puzzles that have additional extra sections.

The extra sections all follow the same general format, with variation in the data they contain. That format is:

| **Component** | **Length** (bytes) | **Description** | |:--------------|:-------------------|:---------------| | Title | 0x04 | The name of the section, these are given in the previous table | | Length | 0x02 | The length of the data section, in bytes, not counting the null terminator | | Checksum | 0x02 | A checksum of the data section, using the same algorithm described above | | Data | variable | The data, which varies in format but is always terminated by null and has the specified length |

The format of the data for each section is described below.

GRBS

The GRBS data is a "board" of one byte per square, similar to the strings for the solution and user state tables except that black squares, letters, etc. are not indicated. The byte for each square of this board indicates whether or not that square is a rebus. Possible values are:

- **0** indicates a non-rebus square.
- **1+n** indicates a rebus square, the solution for which is given by the entry with key **n** in the RTBL section.

If a square is a rebus, only the first letter will be given by the solution board and only the first letter of any fill will be given in the user state board.

RTBL

The RTBL data is a string containing the solutions for any rebus squares.

These solutions are given as an ascii string. For each rebus there is a number, a colon, a string and a semicolon. The number (represented by an ascii string) is always two characters long - if it is only one digit, the first character is a space. It is the key that the GRBS section uses to refer to this entry (it is one less than the number that appears in the corresponding rebus grid squares). The string is the rebus solution.

For example, in a puzzle which had four rebus squares containing "HEART", "DIAMOND", "CLUB",

and "SPADE", the string might be:

```
" 0:HEART; 1:DIAMOND;17:CLUB;23:SPADE;"
```

Note that the keys need not be consecutive numbers, but in official puzzles they always seem to be in ascending order. An individual key may appear multiple times in the GRBS board if there are multiple rebus squares with the same solution.

LTIM

The LTIM data section stores two pieces of information: how much time the solver has used and whether the timer is running or stopped. The two pieces are both stored as ascii strings of numbers, separated by a comma. First comes the number of seconds elapsed, then "0" if the timer is running and "1" if it is stopped. For example, if the timer were stopped at 42 seconds when the puzzle was saved, the LTIM data section would contain the ascii string:

```
"42,1"
```

In C, for example, if `ltim` were a pointer to the LTIM data section, it could be parsed with:

``` int elapsed, stopped;

sscanf((char*)ltim,"%d,%d",&elapsed,&stopped); ```

# GEXT

The GEXT data section is another "board" of one byte per square. Each byte is a bitmask indicating that some style attributes are set. The meanings of four bits are known:

- 0x10 means that the square was previously marked incorrect
- 0x20 means that the square is currently marked incorrect
- 0x40 means that the contents were given
- 0x80 means that the square is circled.

None, some, or all of these bits may be set for each square. It is possible that they have reserved other values.
```

RUSR

The RUSR section is currently undocumented.

What remains

This section contains a list of pieces of the format that we haven't yet figured or documented. If you have, please let us know!

- The various unknown parts of the header, mentioned at the beginning
- The algorithm used for scrambling puzzles is documented in the comments, it still needs to be integrated into the main text.
- The RUSR data section format is also described in the comments but not the main text.

Credit

Most of this document is by Josh Myer, with some work also done by Evan Martin. Chris Casinghino added documentation of the optional extra sections, with help from Michael Greenberg. The commenters below, including mrichards42@gmx.com and boisvert42, also contributed.