FileFormat
Detailed file format documentation

Updated Feb 20, 2010 by Chris.Casinghino@gmail.com

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
2. the puzzle solution and the current state of the cells, with size determined by the puzzle dimensions described in the previous section
3. a series of NUL-terminated variable-length strings with information like the author, copyright, the puzzle clues and a note about the puzzle.
4. 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.

Comment by bradley....@gmail.com, Apr 24, 2008

Note that "cryptic" puzzles, like those found here: http://world.std.com/~wij/puzzles/cru/ have a version number of 1.3

Comment by mrichard...@gmx.com, Jul 21, 2009

All puzzles saved by Across Lite v2 have version 1.3, not just cryptics.

I've seen a version 1.2c as well as 1.2\0 . . . Perhaps the 'c' means that they were saved by Crossword Compiler. Just a guess.

For version 1.3 puzzles:

- The "primary" and "partial board" checksums include the notes section.

i.e. this goes after the calculation of each:

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

For all puzzles:

- If the title, author, copyright, or notes are blank, they don't factor into the checksums. This should work for all puzzles:

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

- The unknown value at 0x1e is 0x0000 in all puzzles that are not scrambled. In scrambled puzzles, this value is a checksum for the unscrambled grid.
- The values at 0x30 and 0x32 are flags that indicate what kind of puzzle we're dealing with.
  - The only values I've seen in the 0x30 number are:
    - 0x0001 = Normal puzzle
    - 0x0401 = Diagramless puzzle
  - And for 0x32
    - 0x0000 = Normal puzzle
    - 0x0002 = Scrambled solution
    - 0x0004 = No solution (e.g if the user created a Text format puzzle and entered X for the entire solution).

There are a pile of extra sections that can follow the grid, each with their own checksum. The general format is:

- title (4-byte string)
- length (short)
- checksum (short)
- data
- nul

Known sections (Across Lite 1):

- GEXT -- information about incorrect flags, circles, etc.

Known sections (Across Lite 2):

- LTIM -- time
- RUSR -- user rebus entries
- GRBS -- solution rebus entries
- RTBL -- table used to look up GRBS values

GRBS and RTBL must be found together

Comment by mrichard...@gmx.com, Jul 31, 2009

Oops . . . proofreading.

That should be

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

Comment by project member Chris.Casinghino@gmail.com, Oct 30, 2009

Note also that GRBS sometimes occurs without RTBL in puzzles that have no rebus squares. I've only observed this in puzzles that also have other sections, where it is common.

Comment by boisver...@gmail.com, Nov 6, 2009

A few more details:

1. If there's a notepad, this will affect the checksum. The primary checksum and c_part checksums then need to be updated via

```
cksum = cksum_region(notepad, strlen(notepad)+1, cksum);
c_part = cksum_region(notepad, strlen(notepad)+1, c_part);
```

2. "Unknown32" is a short that tells Across Lite whether the solution is scrambled. In an unscrambled puzzle it is 0, otherwise nonzero (usually 4)
3. If the puzzle is scrambled, then "Unknown" is a checksum of the puzzle solution. Specifically, if the (unscrambled) solution is laid out in a string in column-major order with all black squares removed, then "Unknown" equals cksum_region(solution,0x0000).

Love the site! Let's hope we can get this fully documented.

Comment by mrichard...@gmx.com, Nov 17, 2009

This is approximately the scrambling code:

```
 def scramble(solution, key):
     """
     Solution is the puzzle's solution as a string running down instead of across:
     i.e. if the puzzle is:
         C A T
         # # A
         # # R
     solution is C..A..TAR

     Key is a list of 4 digits

     """
     scrambled = solution

     for key_num in key:
         last_scramble = scrambled
         scrambled = ''

         for i, letter in enumerate(last_scramble):
             letter_val = ord(letter) + key[i % 4]

             # Make sure this letter is a capital letter
             if letter_val > 90:
                 letter_val -= 26

             scrambled += chr(letter_val)

         scrambled = shift_string(scrambled, key_num)
         scrambled = scramble_string(scrambled)

 def shift_string(scrambled, num):
     return scrambled[num:] + scrambled[:num]

 def scramble_string(scrambled):
     # Split the string in half
     mid = len(scrambled) / 2
     front = scrambled[:mid]
     back  = scrambled[mid:]

     # Assemble the parts:
     # back[0], front[0], back[1], front[1] . . .
     return_str = ''
     for f, b in zip(front, back):
```

```
            return_str += b + f

        # If len(scrambled) is odd, the last character got left off
        if len(scrambled) % 2 != 0:
            return_str += back[-1]

        return return_str
```

It's a rough translation of the C++ code from http://wx-xword.svn.sourceforge.net/viewvc/wx-xword/trunk/src/puz/Scrambler.cpp (That is a crossword puzzle program I've been working on for a while isn't a plug at all?. Anything I know about the format is somewhere is the source code tree for that project, and it's decently documented).

More later on the RTBL format.

Mike

Comment by mrichard...@gmx.com, Nov 18, 2009

(I meant "More later on the RUSR format . . .)

RUSR: A "grid" of nul-terminated strings representing the user-entered rebus squares. If the user entry for a square is a symbol, the entry is a number enclosed in square braces.

Scrambling: It's been a while since I wrote the scrambling code, so the above translation has the following error: the input string does not include black squares. Thus, if the crossword puzzle grid were:

```
ABC
##D
EFG
```

The input to the scrambling function would be `'AEBFCDG'` The resulting scrambled solution is then put in place of the real solution (again, running down instead of across) in the puzzle file after re-inserting the black squares.

e.g. Scramble the puzzle above with key 1234

```
scramble('AEBFCDG', [1,2,3,4]) -> 'MLOOPKJ'
```

Thus the solution grid would turn into

```
MOP
##K
LOJ
```

The unscrambling algorithm is, naturally, exactly the opposite of the scrambling algorithm.

There are only so many possible keys (keys must be 4-digit numbers and cannot start with 0), so a brute-force approach to unscrambling puzzles shouldn't take too long.

But that takes the fun out of the puzzle! More interesting to me is the fact that (as has been noted) the unknown32 value is a checksum of the unscrambled puzzle. Across Lite won't tell you if a locked/scrambled puzzle is solved correctly, but it's a simple matter to checksum the user-entered grid to see if the puzzle is solved correctly.

Mike

Comment by adejarn...@gmail.com, Sep 29, 2011

Thanks for providing this, it's very useful. I have posted a python implementation of the format here: http://github.com/alexdej/puzpy. It is fairly robust -- I have tested it on over 10,000 .puz files from various sources.

Comment by crud...@gmail.com, Sep 27, 2012

Superb document. I am in the process of a ruby library for this, currently have everything but scrambled implemented.

One comment: for GEXT it mentions "None, some, or all of these bits may be set for each square." However (at least according to AcrossLite?) if a square is revealed, only 0x40 is set, regardless of whether it was previously marked as previously or currently incorrect. Similarly with previously marked as incorrect seems to unset currently marked incorrect.

Comment by matteo.p...@gmail.com, Nov 3, 2013

Hi, thanks for alla of your work. I've created a class that reads .puz file using .Net. It's written in c#. This first version doesn't work with scrumbled puzzle. Here is source code: https://github.com/mapo80/puz

Your wiki has an error for "Clue Assignment". This is right version:

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

1. Check that there is no blank to the left of us

if x == 0 or is_black_cell(x-1, y):

1. 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):

1. ...as above, but on the y axis

Comment by fixbox2...@gmail.com, Nov 10, 2013

I compiled readpuz and can successfully convert .txt to .puz. The puzzle loads into across lite, but the clues don't match the puzzle. After reviewing the source code, I don't see where the clues get rearranged when doing the conversion from txt to puz. I'm thinking it's missing the clue assignment code.

Comment by martinde...@gmail.com, Jan 22, 2014

i am writing a ruby implementation using alex's python code referred to above as a second reference, since it seems a little more complete than the document above.

two additions i can suggest to the docs:

1. There is potentially some junk before the header field; the safest way to detect start-of-file is to search forward for the file magic string ACROSS&DOWN and then start two bytes before that.

2. The "unknown bitmask" in the penultimate header field is the puzzle type,

Normal = 0x0001 Diagramless = 0x0401

Comment by danvdk, Dec 9, 2014

The New York Times has recently started highlighting related clues, e.g. "See 14-Down". It doesn't seem to be based solely on the text of the clues. On the 12/07/2014 puzzle (http://www.nytimes.com/crosswords/game/2014/12/07/daily/), for example, you can highlight 102D which reads "Mythological figure hinted at by the answers to the eight starred clues as well as this puzzle's design" and see the eight other clues starred. Is this information encoded in the .puz file, or is the NY Times site getting it through a side channel?

Comment by binaryf...@gmail.com, May 31, 2015

GEXT is missing the pencil information:

0x08 means pencil