# PNG

CS6025 Data Encoding

Yizong Cheng

3-31-15

Portable Network Gr

www.libpng.org/pub/png/spec/iso/index-object.html

W3C Recommendation

# Portable Network Graphics (PNG) Specification (Second Edition) Information technology — Computer graphics and image processing — Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E)

## W3C Recommendation 10 November 2003

**This version:**
http://www.w3.org/TR/2003/REC-PNG-20031110

**Latest version:**
http://www.w3.org/TR/PNG

**Previous version:**
http://www.w3.org/TR/2003/PR-PNG-20030520

**Editor:**
David Duce, Oxford Brookes University (Second Edition)

**Authors:**
See author list

Please refer to the errata for this document, which may include some normative corrections.

See also the translations of this document.

# PNG Signature + IHDR + … + IEND

- The first eight bytes of a PNG datastream always contain the following (decimal) values:
  - 137 80 78 71 13 10 26 10
- This signature indicates that the remainder of the datastream contains a single PNG image, consisting of a series of chunks beginning with an IHDR chunk and ending with an IEND chunk.

| | | | | |
|---|---|---|---|---|
| signature | IHDR | chunk… | chunk | IEND |

# Chunk

- Length (4 bytes)
- Chunk type (4 bytes)
- Chunk data (Length bytes)
- CRC32 (4 bytes)
  - for chunk type and chunk data only

| | | | |
|---|---|---|---|
| length | chunk type | chunk data | CRC |

# IHDR Chunk

- Length: 13 (0, 0, 0, 13)
- Chunk type IHDR ( 73, 72, 68, 82 )
- width (4 bytes)
- height (4 bytes)
- bit depth, color type, compression method, filter method, interlace method (1 byte each)
- CRC (4 bytes)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 13 | IHDR | width | height | depth | color | compr | filter | interlc | crc32 |

# Color Types and Allowed Bit Depths

- Greyscale (0) → 1, 2, 4, 8, or 16 bits per greyscale sample
- Truecolour (2) → 8 or 16 bits per R, G, B
- Indexed-colour (3) → 1, 2, 4, or 8 bits per palette index
- Greyscale with alpha (4) → 8 or 16 bits per greyscale sample and per alpha sample (A)
- Truecolour with alpha (6) → 8 or 16 for RGBA

# Alpha Channel

- An alpha channel, representing transparency information on a per-pixel basis, can be included in grayscale and truecolor PNG images.
- An alpha value of zero represents full transparency, and a value of $2^{bitdepth} - 1$ represents a fully opaque pixel.
- Intermediate values indicate partially transparent pixels that can be combined with a background image to yield a composite image.
- Thus, alpha is really the degree of opacity of the pixel.

# Beginning of a .png file

chunk data length

width

height

5 parameters

chunk type

CRC32

```
0       8950 4e47 0d0a 1a0a 0000 000d 4948 4452    .PNG........IHDR
10      0000 0780 0000 0438 0806 0000 00e8 d3c1    ........8.......
20      4300 0000 0473 4249 5408 0808 087c 0864    C....sBIT....|.d
30      8800 0000 0970 4859 7300 000b 1300 000b    .....pHYs.......
40      1301 009a 9c18 0000 0016 7445 5874 4372    ..........tEXtCr
50      6561 7469 6f6e 2054 696d 6500 3032 2f31    eation Time.02/1
60      302f 3130 ad39 eeb8 0000 001c 7445 5874    0/10.9......tEXt
70      536f 6674 7761 7265 0041 646f 6265 2046    Software.Adobe F
80      6972 6577 6f72 6b73 2043 5334 06b2 d3a0    ireworks CS4....
90      0008 0000 4944 4154 78da c4fd cb92 6549    ....IDATx.....eI
a0      b225 862d 33db e71c 778f 888c 7cd5 a3eb    .%.-3...w...|...
```

```
// read header of .png and get all parameters
 void readHeader(){
   int len = 0;
   try {
     len = System.in.read(headerBuffer);
   } catch (IOException e){
     System.err.println(e.getMessage());
     System.exit(1);
   }
   if (len != headerSize){
     System.err.println(" no header ");
     System.exit(1);
   }
   for (int i = 0; i < 16; i++){
     int a = headerBuffer[i];
     if (a < 0) a += 256;
     if (a != signature[i]){
       System.err.println(" not PNG ");
       System.exit(1);
     }
   }
```

```
for (int i = 0; i < 4; i++){
  int a = headerBuffer[i + 16];
  if (a < 0) a += 256;
  width <<= 8;
  width += a;
}
for (int i = 0; i < 4; i++){
  int a = headerBuffer[i + 20];
  if (a < 0) a += 256;
  height <<= 8;
  height += a;
}
bitDepth = headerBuffer[24];
colorType = headerBuffer[25];
compressionMethod = headerBuffer[26];
filterMethod = headerBuffer[27];
interlace = headerBuffer[28];
if (bitDepth != 8 || colorType != 6 ||
  compressionMethod != 0 || filterMethod != 0 ||
  interlace != 0){
    System.err.println("decoder not implemented");
    System.exit(1);
}
lineWidth = width * 4 + 1;
```

# IDAT = Image Data

- There may be multiple IDAT chunks; if so, they shall appear consecutively with no other intervening chunks.
- The compressed datastream is then the concatenation of the contents of the data fields of all the IDAT chunks.

| | | | |
|---|---|---|---|
| length | IDAT | data | crc32 |

```
boolean readChunk(){ // false after IEND
   int len = 0;
   // read chunk header
   try {
     len = System.in.read(chunkHeaderBuffer);
   } catch (IOException e){
     System.err.println(e.getMessage());
     System.exit(1);
   }
   if (len != chunkHeaderSize){
     System.err.println(" no chunk header ");
     System.exit(1);
   }
  // get chunk data length
  int chunkDataLength = 0;
  for (int i = 0; i < 4; i++){
     int a = chunkHeaderBuffer[i];
     if (a < 0) a += 256;
     chunkDataLength <<= 8;
     chunkDataLength += a;
  }
```

```
 // get chunk type
  String chunkType = new String(chunkHeaderBuffer, 4, 4);
  if (chunkType.equals("IEND")) return false;
  if (chunkType.equals("IDAT")){
    // place data in dataBuffer
    len = System.in.read(dataBuffer, compressedDataLength,
       chunkDataLength);
    compressedDataLength += chunkDataLength;
  }else if (chunkDataLength > 0){
    byte[] tmpBuffer = new byte[chunkDataLength];
    len = System.in.read(tmpBuffer);
  }
  if (len != chunkDataLength){
    System.err.println(" no chunk data ");
    System.exit(1);
  }
  // get CRC for the chunk
  len = System.in.read(crcBuffer);
  if (len != crcSize){
    System.err.println(" no CRC ");
    System.exit(1);
  }
  return true;  // there are more chunks
}
```

# Compression Method 0

- concatenation of IDAT in dataBuffer with compressedDataLength is compressed with GNU LZ77 DEFLATE algorithm.
- java.util.zip contains the algorithm.

```
Deflater compresser = new Deflater();
compresser.setInput(input);
compresser.finish();
int compressedDataLength = compresser.deflate(output);
Inflater decompresser = new Inflater();
decompresser.setInput(output, 0, compressedDataLength);
byte[] result = new byte[resultSize];
int resultLength = decompresser.inflate(result);
decompresser.end();
```

# Decompress IDAT Data in dataBuffer

```
// use Inflater to decompress data in
// dataBuffer
// decompressed data in resultBuffer
void decompress(){
  resultBuffer =
   new byte[width * height * 4 + height];
  Inflater decompresser = new Inflater();
  // your code
}
```

# gzip

- gzip is a software application used for file compression and decompression.
- The program was created by Jean-Loup Gailly and Mark Adler as a free software replacement for the compress program used in early Unix systems, and intended for use by the GNU Project (the "g" is from "GNU").
- Version 0.1 was first publicly released on 31 October 1992, and version 1.0 followed in February 1993.

# DEFLATE

- gzip is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE was intended as a replacement for LZW and other patent-encumbered data compression algorithms which, at the time, limited the usability of compress and other popular archivers.
- zlib is an abstraction of the DEFLATE algorithm in library form which includes support both for the gzip file format and a lightweight stream format in its API.
- The zlib stream format, DEFLATE, and the gzip file format were standardized respectively as RFC 1950, RFC 1951, and RFC 1952.

Network Working Group
Request for Comments: 1950
Category: Informational

P. Deutsch
Aladdin Enterprises
J-L. Gailly
Info-ZIP

May 1996

# ZLIB Compressed Data Format Specification version 3.3

A zlib stream has the following structure:

```
  0   1
+---+---+
|CMF|FLG|   (more-->)
+---+---+
```

(if FLG.FDICT set)

```
  0   1   2   3
+---+---+---+---+
|     DICTID    |   (more-->)
+---+---+---+---+
```

```
+=====================+---+---+---+---+
|...compressed data...|    ADLER32    |
+=====================+---+---+---+---+
```

Any data which may appear after ADLER32 are not part of the zlib stream.

**FLG (FLaGs)**

This flag byte is divided as follows:

```
bits 0 to 4  FCHECK  (check bits for CMF and FLG)
bit  5       FDICT   (preset dictionary)
bits 6 to 7  FLEVEL  (compression level)
```

The FCHECK value must be such that CMF and FLG, when viewed as a 16-bit unsigned integer stored in MSB order (CMF*256 + FLG), is a multiple of 31.

**FDICT (Preset dictionary)**

If FDICT is set, a DICT dictionary identifier is present immediately after the FLG byte. The dictionary is a sequence of bytes which are initially fed to the compressor without producing any compressed output. DICT is the Adler-32 checksum of this sequence of bytes (see the definition of ADLER32 below). The decompressor can use this identifier to determine which dictionary has been used by the compressor.

**FLEVEL (Compression level)**

These flags are available for use by specific compression methods. The "deflate" method (CM = 8) sets these flags as follows:

```
0 - compressor used fastest algorithm
1 - compressor used fast algorithm
2 - compressor used default algorithm
3 - compressor used maximum compression, slowest algorithm
```

The information in FLEVEL is not needed for decompression; it is there to indicate if recompression might be worthwhile.

# First Byte is often 78 (‘x’)

**CMF (Compression Method and flags)**

This byte is divided into a 4-bit compression method and a 4-bit information field depending on the compression method.

```
bits 0 to 3   CM      Compression method
bits 4 to 7   CINFO   Compression info
```

**CM (Compression method)**

This identifies the compression method used in the file. CM = 8 denotes the “deflate” compression method with a window size up to 32K. This is the method used by gzip and PNG (see references [1] and [2] in Chapter 3, below, for the reference documents). CM = 15 is reserved. It might be used in a future version of this specification to indicate the presence of an extra field before the compressed data.

**CINFO (Compression info)**

For CM = 8, CINFO is the base-2 logarithm of the LZ77 window size, minus eight (CINFO=7 indicates a 32K window size). Values of CINFO above 7 are not allowed in this version of the specification. CINFO is not defined in this specification for CM not equal to 8.

# Filter Method 0

- In order to improve lossless compression of the image, a prediction like that in JPEG-LS is used and differential is in the filtered data.
- Each scanline (row) is prepended by a filter type byte to indicate the filtering for the row.
- For each color channel, a function f(a,b,c) is used to predict x and x – f(a,b,c) replaces x.

| c | b |
|---|---|
| a | x |

# Five Filter Types

- 0: x is not replaced
- 1: replace x by x - a
- 2: replace x by x - b
- 3: replace x by x – (a + b) / 2
- 4: replace x by x - paeth(a,b,c) where paeth(a,b,c) is one of a, b, c that is closest to a + b - c.

# 1080 Filter Type Bytes

| filter type | R | G | B | A | R | G | B | A | … |
|---|---|---|---|---|---|---|---|---|---|

1444444444444444444444444444433444443344344443444434444334444424
4444424344444444444444444444444434444344434334444443344433444444
3333443343444444443334444444444344344444334433343444444444444444
4444444444444444444444444444444444444444443444444433344444444444
4444444444444444444444444444444444444433444444444444444444443444
4334344444433444444344444444444444444444444444444444444443444444
4444444433334333444344444444444444444444444444444444444443444444
4444444444434444444443333443343444443344443344333444443344444433
4444334433444444433344443444434444444444444444444444444444444444
4444443444444444444444444444444444444444444444444444444444444444
4444444444444433444334433444443344344333333333344334434444443344
3334444433334443444443444334433333343343334334333333333433333343
3433333333343333334333334433334333343343344334334333344344443334
4334433344443444443444344443334334343334334344344433444443334333
4434333433344333343334443344333334434433344343344334443333343333
3333334333334433344343334333334334433333433444334333333433334343
33433433333433333333333333333333331333333333333333333333

# Paeth Prediction for Filter Type 4

```
// Paeth prediction for filter type 4
int paeth(int a, int b, int c, int x){
  int p = a + b - c;
  int pa = a <= p ? p - a : a - p;
  int pb = b <= p ? p - b : b - p;
  int pc = c <= p ? p - c : c - p;
  return (pa <= pb && pa <= pc) ? a :
  (pb <= pc ? b : c);
}
```

```
// reverse filter method 0
void reverseFilter(){
 int offset = 0;
 // beginning position of the current scanline
 int a, b, c, x, r;
 //   c b
 //   a x
 // x is resultBuffer[offset + j]
 // r is its value after filter reversed
 for (int i = 0; i < height; i++){  // one scanline a time
   int filterType = resultBuffer[offset]; // filter type byte
   for (int j = 1; j < lineWidth; j++){
    // get a, b, c, x as nonnegative integers
    if (j < 4) a = 0; else a = resultBuffer[offset + j - 4];
    if (i == 0) b = c = 0;
    else{ b = resultBuffer[offset + j - lineWidth];
          if (j < 4) c = 0;
          else c = resultBuffer[offset + j - lineWidth - 4];
    }
    x = resultBuffer[offset + j];
    if (a < 0) a += 256; if (b < 0) b += 256;
    if (c < 0) c += 256; if (x < 0) x += 256;
```

```
// reverse filter for the 5 filter types
 switch (filterType){
  case 0: break;
  case 1: if (j >= 4){
     r = x + a; if (r >= 256) r -= 256;
     resultBuffer[offset + j] = (byte)r;
    }
    break;
  case 2:
     r = x + b; if (r >= 256) r -= 256;
     resultBuffer[offset + j] = (byte)r;
    break;
  case 3:
     r = x + (a + b) / 2; if (r >= 256) r -= 256;
     resultBuffer[offset + j] = (byte)r;
     break;
   case 4:
     r = x + paeth(a, b, c, x); if (r >= 256) r -= 256;
     resultBuffer[offset + j] = (byte)r;
     break;
   default: ;
 }
}
```

# BMP File Header

| 0 | 2 | 10 | 14 | 18 | 22 | 26 | 28 | 34 | |
|---|---|---|---|---|---|---|---|---|---|
| BM | file size | data offset | 40 | width | height | 1 | depth | data size | |

```
// fill 4 bytes in BMPHeader at offset with a number
void fillNumber(int offset, int number){
  int k = 0; for (; k < 4; k++){
    BMPHeader[offset + k] = (byte)(number % 256);
    number /= 256;
    if (number == 0) break;
  }
}

// fill non-zero parameters in BMPHeader
void fillBMPHeader(){
  for (int i = 0; i < BMPHeaderSize; i++) BMPHeader[i] = 0;
  BMPHeader[0] = 'B'; BMPHeader[1] = 'M';
  int rawDataSize = width * height * 3;
  fillNumber(2, rawDataSize + BMPHeaderSize);
  BMPHeader[10] = 54; BMPHeader[14] = 40;
  fillNumber(18, width);
  fillNumber(22, height);
  BMPHeader[26] = 1; BMPHeader[28] = 24;
  fillNumber(34, rawDataSize);
}
```

# Making BMPData BGRBGR…

```
// lossless BMP as output
void toBMP(){
 try {
  fillBMPHeader();
  System.out.write(BMPHeader);
  byte[] BMPData = new byte[width * height * 3];
  int n = 0;
  for (int i = height - 1; i >= 0; i--)
   for (int j = 0; j < width; j++)
     for (int k = 0; k < 3; k++)
    BMPData[n++] = resultBuffer[?];  // your pick
  System.out.write(BMPData);
 } catch (IOException e){
   System.err.println(e.getMessage());
   System.exit(1);
 }
}
```

# Homework 19: due 4-6-15

- Complete decompress() and toBMP() functions in H19.java
- java H19 < hg127.png > hg127.bmp
- Submit your source code and test result.

# H19 main

```
public static void main(String[] args){
  H19 h19 = new H19();
  h19.readHeader();
  h19.readData();
  h19.decompress();  // need work
  h19.reverseFilter();
  h19.toBMP();  // need work
}
```

```
public class H19{
  static final int headerSize = 33;
  static final int BMPHeaderSize = 54;
  static final int chunkHeaderSize = 8;
  static final int dataSize = 2000000;
  static final int crcSize = 4;
  static final int[] signature = new int[]{
    137, 80, 78, 71, 13, 10, 26, 10,
    0, 0, 0, 13, 73, 72, 68, 82 };
  byte[] headerBuffer = new byte[headerSize];
  byte[] chunkHeaderBuffer = new byte[chunkHeaderSize];
  byte[] dataBuffer = new byte[dataSize];
  byte[] crcBuffer = new byte[crcSize];
  byte[] BMPHeader = new byte[BMPHeaderSize];
  byte[] resultBuffer = null;
  int compressedDataLength = 0;
  int decompressedDataLength = 0;
  int width = 0; int height = 0; int lineWidth = 0;
  int bitDepth = -1; int colorType = -1;
  int compressionMethod = -1; int filterMethod = -1;
  int interlace = -1;
```