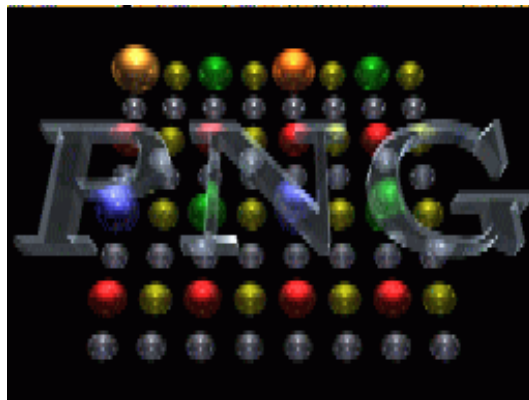


# Assignment 6 / Multimedia – The PNG-Format

Kenan Esau

February 2001



Organization: TAE/Esslingen  
Course: M.Sc Distributed Systems Engineering  
Lecturer: Prof. Dr. K. Khakzar

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Why another new File Format ?</b>	<b>3</b>
<b>3</b>	<b>The Format</b>	<b>3</b>
3.1	Some important Chunks . . . . .	4
3.1.1	IHDR Image Header . . . . .	4
3.1.2	IDAT Image Data . . . . .	5
3.1.3	IEND Image Trailer . . . . .	6
3.2	Filtering . . . . .	6
<b>4</b>	<b>Compression Algorithm used in PNG</b>	<b>6</b>
4.1	The LZ77 Algorithm . . . . .	7
4.2	Huffman Encoding . . . . .	7
4.3	The Deflate Algorithm . . . . .	8
<b>5</b>	<b>Conclusions</b>	<b>9</b>

# 1 Introduction

This assignment is about the image format PNG and the deflate compression algorithm which is currently used in this format. I think this format is very interesting since it is completely free. It was designed with portability and platform independence in mind. At the same time there was spend quite an effort to ensure that there are no patents which could restrict the use of this format. The first part of this assignment is about the PNG file format.

The compression algorithm used by this format is also very interesting since it is a combination of the LZ77 algorithm and Huffman encoding. The second part of the assignment treats the data compression issues.

## 2 Why another new File Format ?

Since GIF is no longer freely usable an adequate replacement which uses an unpatented compression scheme is needed. Besides this annoying patent topic the GIF-format had other quite annoying constraints and a new format would be a good chance to overcome those constraints. Although just replacing GIF's compression method would avoid the patent-problem, this does not solve shortcomings like the fact that GIF does not support true color images, alpha channels, gamma correction or chromacity information.

## 3 The Format

PNG (pronounce ping) was created with simplicity in mind. Thus the format is very simple. Each PNG file is preceded by a signature which consists of eight bytes. The file signature is:

0x89 0x50 0x4e 0x47 0x0d 0x0a 0x1a 0x0a

A PNG file is composed out of "chunks". Those chunks follow directly behind the file signature. A valid PNG file consists at least of an IHDR chunk, one or more IDAT chunks, and an IEND chunk [1] . A chunk consist of four fields:

1. **Length**

This is a 4-byte unsigned integer value indicating the length of the chunk data field described below.

2. **Chunk Type**

This is a 4-byte unsigned integer value which is a type code. Type codes are restricted to consist of uppercase and lowercase ASCII-letters. An encoder/decoder has to treat this value a 4-byte unsigned integer value and not as a character string.

### 3.1 Some important Chunks

---

#### 3. Chunk Data

The data bytes appropriate to the chunk type. This field can also be empty (Length field of 0).

#### 4. CRC

Each chunk is secured by a 4-byte CRC calculated on the preceding bytes of the chunk including the chunk type code and data fields but not the length field.

A minimal PNG file might look like this: The file signature followed by the IHDR-Chunk and one or more IDAT-Chunks. The end of the file is marked by the IEND-Chunk.



Figure 1: *Minimal PNG File*

### 3.1 Some important Chunks

In the following subsections the chunks for the file header, the image data and the end of file marker are presented. Besides the chunks presented there are many others. For example there is a chunk which describes a palette (PLTE). This chunk is mandatory for some certain types of color (see table 1 in section 3.1.1). Other chunks include information about a gamma value or chromacity to adjust the image to the type of hardware the user is using to view the image. This is necessary since a picture does not look the same on different hardware platforms (eg. MAC, PC, SGI).

The concept of chunks makes the PNG format very easy to extend. New functionality can easily be implemented by introducing some new kind of chunk. At the same time old PNG-encoders would still be compatible since they would still be able to read a file with the new chunks. It is impossible to make use of unknown chunks but the known ones can still be interpreted. So the image is at least displayed perhaps not in all its details and not with all the new fancy features introduced by an unknown chunk but at least it works.

#### 3.1.1 IHDR Image Header

The IHDR chunk has to be the first chunk of the file. It has to be placed directly after the file signature. The data-field of this chunk contains:

- **Width** 4 bytes  
Contains the width of the picture in pixels.

### 3.1 Some important Chunks

---

- **Height 4 bytes**  
Contains the height of the picture in pixels.
- **Bit depth 1 byte**  
This is an integer value giving the number of bits per sample or per palette (not per pixel).
- **Color type 1 byte**  
This value describes the interpretation of the image data. Valid Values for the color type are listed in table 1.

Color Type	Allowed Bit Depths	Interpretation
0	1, 2, 4, 8, 16	Each pixel is a gray scale sample.
2	8, 16	Each pixel is a RGB triple.
3	1, 2, 4, 8	Each pixel is a palette index; a PLTE chunk must appear.
4	8, 16	Each pixel is a gray scale sample, followed by an alpha sample
6	8, 16	Each pixel is a gray scale sample, followed by an alpha sample.

Table 1: List of color types available with PNG

- **Compression method 1 byte**  
This byte contains an number indicating the method used to compress the data. At the moment only method 0 (deflate/inflate) is defined. This field is provided for possible future extensions or proprietary variants. With this approach the PNG format is not coupled to a certain compress method like it is the case with GIF.
- **Filter method 1 byte**  
At the present only method 0 is defined. Method 0 is adaptive filtering with five basic filter types.
- **Interlace method 1 byte**  
This field indicates transmission order of the image data. At the moment only two values are defined: 0 (no interlace) or 1 (Adam7 interlace).

#### 3.1.2 IDAT Image Data

This chunk contains the data of the image. Thus there is no additional structure defined for the data field in this chunk. The data field just contains the compressed, filtered image data. The compression and the filter method are defined by the IHDR chunk (see 3.1.1). For more details on filtering and compression see sections 3.2 and 4.

## 3.2 Filtering

---

### 3.1.3 IEND Image Trailer

The IEND chunk is the last chunk of the image. It marks the end of the PNG data stream all data appearing behind this chunk is irrelevant for each decoder which wants to read a PNG file.

## 3.2 Filtering

Every scan line of the picture is preceded by a byte indicating one of five filter types. These filters are completely loss less and are applied to the raw image data before it is compressed and written to a file. The objective of applying a filter to the data is to increase redundancy and there for enhancing its compression ratio.

A suggestion in the PNG specification of how to determine which filter produces the data which can be compressed best is to apply every filter to a scan line, compress it and check the size. This is not very fast but it is easy to implement and this approach results in the best possible compression ratio.

The filters are very simple and easy to implement. For example the sub-filter calculates the difference between the value of each byte of a pixel and the value of the corresponding byte of the prior pixel. The following formula is applied to each scan line:

$$\text{Sub}(x) = \text{Raw}(x) - \text{Raw}(x-\text{bpp})$$

Note that this computation is done regardless of the bit depth of a picture. If a pixel consists of more than one byte the MSB of the nth pixel is predicted by the MSB of the (n-1)th pixel (this is the way bpp is defined see [1]).

## 4 Compression Algorithm used in PNG

The compression algorithm used in PNG is the deflate-algorithm. Deflate is a loss less compression method which uses a combination of the LZ77 algorithm (see section 4.1) and Huffman coding (see section 4.2) [4]. Deflate is independent of CPU type, operating system, file system and character set, and hence can be used for interchange [4].

Since no loss less compression method can compress every possible data set one can show easily that in the worst case an expansion of 5 bytes per 32K-byte block is possible with this algorithm. That is 0.015% for large data sets. Graphical data can be compressed very efficient so compression rates much greater than three are possible.

## 4.1 The LZ77 Algorithm

---

The next two subsections describe the two basic principles used for the deflate algorithm which is currently used with the PNG format. Section 4.3 describes the implementation of the deflate algorithm in more detail.

### 4.1 The LZ77 Algorithm

The LZ77 algorithm was presented by Abraham Lempel and Jacob Ziv in 1977 in their dictionary based scheme for text compression [5]. Text compression refers to loss less compression for all types of data.

In 1982 James Storer and Thomas Szymanski presented an improved algorithm which was based on the original LZ77. They called their approach LZss. But if we talk of LZ77 algorithms today we usually mean this improved approach.

The principle of LZ77 based algorithms is very simple. If you find a pattern in a stream of data which has already appeared somewhere else earlier in this stream you put a kind of reference to this earlier appearance instead of writing the whole pattern a second time to the stream.

The main difference between the original LZ77 and the LZss algorithm was that the LZ77 algorithm wrote a offset/length pair every time it found a match even if this match was only one byte – in this case the algorithm increased the size of the “compressed” output stream instead of making it smaller. LZss is a little bit smarter in this case. It uses bit flags, which indicate if the next byte in the input stream is, a literal (a byte), or a pair of offset/length (a reference to a pattern) [5].

### 4.2 Huffman Encoding

The principle of Huffman encoding is to use fewer bits for literals which appear more often in a data stream and use more bits for the literals which appear less frequently. But how can it work if different literals can have different bit-length? Huffman encoded literals have to have a property called the prefix property – that is no bit-sequence encoding of a literal is the prefix of any other bit-sequence [6].

The second approach – instead of assigning the length of the encoded literals corresponding to their frequency – is just to define the length of the literals. This is the approach used with the deflate algorithm.

Huffman coding – also called prefix coding – represents symbols from an a priory known alphabet by bit sequences of different lengths. But a parser can always parse an encoded string unambiguously symbol-by-symbol [4] due to the prefix property. Such a code is defined in terms of a binary tree in which the two edges descending from each non-leaf are labeled 0 and 1. The final leaves can be labeled with the symbols of the

### 4.3 The Deflate Algorithm

---

alphabet. The codes for a symbol are defined by the sequence of 0's and 1's on the edges leading from the root to the leaf labeled with that symbol:

A Huffman tree can also build dynamically. After analysis of the frequency of appearance of the literals in the data stream which has to be compressed it can be determined very efficiently which symbols should have a shorter code and which not. This scheme is also called dynamic Huffman encoding and can be used with the deflate algorithm too (see section 4.3).

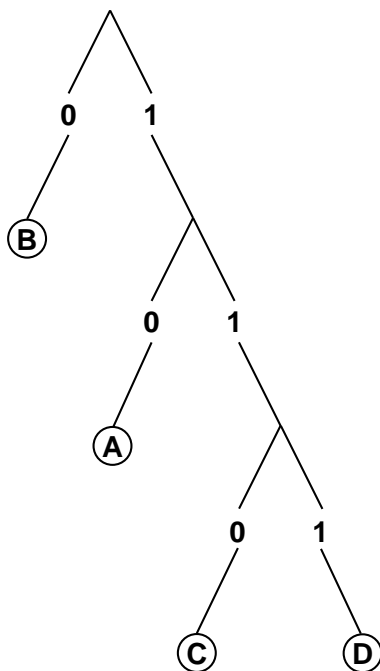


Figure 2: *Huffman Tree with 4 Symbols*

According to figure 2 the symbol “B” is encoded with one bit: “0” and the symbol “C” is encoded with the three bit sequence “110” ...

### 4.3 The Deflate Algorithm

As already mentioned in section 4 the deflate algorithm is a combination of the principles presented in sections 4.1 and 4.2.

There are three distinct alphabets: The “alphabet of bytes” ranging from 0 to 255 and references to patterns. Such a reference looks like `<length, backward distance>`. The parameter `length` describes how many bits should be copied to the actual position of the output stream beginning from `backward distance` bits before the actual position in the output stream.

## 5 Conclusions

---

The two alphabets of length-codes and the alphabet of bytes are merged into one alphabet [4], where values 0..255 represent literal bytes, the value 256 indicates end-of-block, and values 257..285 represent length codes. The length codes can be accompanied by some extra bits indicating a value which has to be added to the length.

If the Huffman codes are fixed as described above the scheme is also called fixed Huffman encoding. But by analysing the frequency of appearance of the literals in a stream a much more efficient Huffman code can be built. This approach is also called dynamic Huffman encoding. If dynamic Huffman encoding is used with the deflate algorithm two distinct tables of Huffman codes are stored immediately after the header bits and before the actual compressed data. One table is for the “alphabets of bytes” (the literals) and the length values and a second table is used for the distance codes.

### **Example – using fixed Huffman codes:**

Assuming the usage of fixed Huffman codes as defined in [4]. If an encoder finds a duplicated pattern which already occurred 127 bits earlier and has a length of 36 bits it has to write something like <36, 127> to the output stream.

A length of 36 bits is encoded – according to RFC-1951 [4] – with the code 273. This code has 3 extra bits which indicate the value which has to be added to 35 (000 means 35, 001 means 36, ...). Since our length of the pattern is 36 we have to set the extra bits to 001.

The literal value 273 which means “take a length between 35 and 42 bits” is encoded 01000100 (compare RFC-1951 section 3.2.6).

The last thing we have to determine is how many bits do we have to go back to find the pattern? There are the distance codes 0-29. They are represented by (fixed-length) 5-bit codes, with possible additional bits. In our case we need to go back 127 bits. Thus we have to use the code 13 which as 5 additional bits and there for ranges from 97 to 128 bits backward. 13 is encoded 01101 which means go back 97 bits. Now we have to add 30 to 29 to go back 127 bits. This is encoded by the 5 additional bits which are set to 11110.

The whole process is shown in figure 3

## 5 Conclusions

The PNG format seems to be a very good substitute for the GIF format. Its increasing use on the web and the increasing support by different applications like web browsers and image manipulation programs shows the capabilities of this format. The only thing GIF could do which PNG

## 5 Conclusions

---

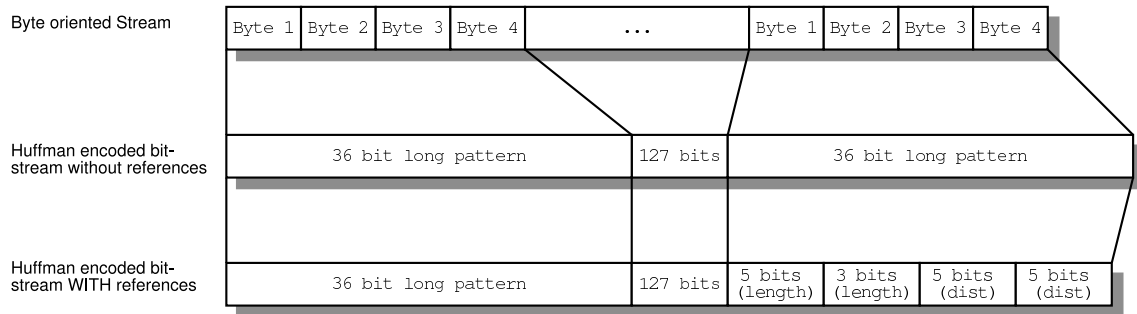


Figure 3: *Process of the deflate algorithm*

can not do is little animations. But even here are new formats like MNG which are based on the PNG standard which overcome this shortcoming.

The appearance of new formats like MNG – for animations – and JNG – a lossy MNG companion with alpha channel – which are based on PNG shows the success of the approach the PNG developers chose. While the use of the PNG format and the support of the applications is increasing time has to show if those new formats can repeat the success of PNG.

The story of GIF and PNG shows the need of protection from silly software patents. It is the software patents which very often stifle innovation not the open source community or anything developed by it.

### References

- [1] PNG Development Group, “PNG Specification Version 1.2”
- [2] RFC-2083, “PNG-Format”
- [3] RFC-1950, “Zlib Specification”
- [4] RFC-1951, “Deflate Specification”
- [5] Arturo San Emeterio Campos, “LZ77 the basics of compression (2nd ed)”
- [6] Owen L. Astrachan, “Huffman Coding: A CS2 Assignment”