L. Peter Deutsch
Aladdin Enterprises
08 Apr 1996

DEFLATE Compressed Data Format Specification version 1.3

File

draft-deutsch-deflate-spec-04.txt

Status of this Memo

This document is an Internet-Draft. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

To learn the current status of any Internet-Draft, please check the "1id-abstracts.txt" listing contained in the Internet- Drafts Shadow Directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (US East Coast), or ftp.isi.edu (US West Coast).

Distribution of this memo is unlimited.

A pointer to the latest version of this and related documentation in HTML format can be found at the URL <ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html>.

Notices

Abstract

This specification defines a lossless compressed data format that compresses data using a combination of the LZ77 algorithm and Huffman coding, with efficiency comparable to the best currently available general-purpose compression methods. The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage. The format can be implemented readily in a manner not covered by patents.

Contents

1 Introduction

1.1 Purpose

The purpose of this specification is to define a lossless compressed data format that:

- Is independent of CPU type, operating system, file system, and character set, and hence can be used for interchange;
- Can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an *a priori* bounded amount of intermediate storage, and hence can be used in data communications or similar structures such as Unix filters;
- Compresses data with efficiency comparable to the best currently available general-purpose compression methods, and in particular considerably better than the "compress" program;
- Can be implemented readily in a manner not covered by patents, and hence can be practiced freely;
- Is compatible with the file format produced by the current widely used gzip utility, in that conforming decompressors will be able to read data produced by the existing gzip compressor.

The data format defined by this specification does not attempt to:

- Allow random access to compressed data;
- Compress specialized data (e.g., raster graphics) as well as the best currently available specialized algorithms.

A simple counting argument shows that no lossless compression algorithm can compress every possible input data set. For the format defined here, the worst case expansion is 5 bytes per 32K-byte block, i.e., a size increase of 0.015% for large data sets. English text usually compresses by a factor of 2.5 to 3; executable files usually compress somewhat less; graphical data such as raster images may compress much more.

1.2 Intended audience

This specification is intended for use by implementors of software to compress data into "deflate" format and/or decompress data from "deflate" format.

The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations. Familiarity with the technique of Huffman coding is helpful but not required.

1.3 Scope

The specification specifies a method for representing a sequence of bytes as a (usually shorter) sequence of bits, and a method for packing the latter bit sequence into bytes.

1.4 Compliance

Unless otherwise indicated below, a compliant decompressor must be able to accept and decompress any data set that conforms to all the specifications presented here; a compliant compressor must produce data sets that conform to all the specifications presented here.

1.5 Definitions of terms and conventions used

Byte: 8 bits stored or transmitted as a unit (same as an octet). For this specification, a byte is exactly 8 bits, even on machines which store a character on a number of bits different from eight. See below, for the numbering of bits within a byte.

String: a sequence of arbitrary bytes.

1.6 Changes from previous versions

There have been no technical changes to the deflate format since version 1.1 of this specification. In version 1.2, some terminology was changed. Version 1.3 is a conversion of the specification to Internet Draft style.

2 Compressed representation overview

A compressed data set consists of a series of blocks, corresponding to successive blocks of input data. The block sizes are arbitrary, except that non-compressible blocks are limited to 65,535 bytes.

Each block is compressed using a combination of the LZ77 algorithm and Huffman coding. The Huffman trees for each block are independant of those for previous or subsequent blocks; the LZ77 algorithm may use a reference to a duplicated string occurring in a previous block, up to 32K input bytes before.

Each block consists of two parts: a pair of Huffman code trees that describe the representation of the compressed data part, and a compressed data part. (The Huffman trees themselves are compressed using Huffman encoding.) The compressed data consists of a series of elements of two types: literal bytes (of strings that have not been detected as duplicated within the previous 32K input bytes), and pointers to duplicated strings, where a pointer is represented as a pair <length, backward distance>. The representation used in the "deflate" format limits distances to 32K bytes and lengths to 258 bytes, but does not limit the size of a block, except for uncompressible blocks, which are limited as noted above.

Each type of value (literals, distances, and lengths) in the compressed data is represented using a Huffman code, using one code tree for literals and lengths and a separate code tree for distances. The code trees for each block appear in a compact form just before the compressed data for that block.

3 Detailed specification

3.1 Overall conventions

In the diagrams below, a box like this:

```
+---+
|   | <-- the vertical bars might be missing
+---+
```

represents one byte; a box like this:

```
+==============+
|              |
+==============+
```

represents a variable number of bytes.

Bytes stored within a computer do not have a “bit order”, since they are always treated as a unit. However, a byte considered as an integer between 0 and 255 does have a most- and least-significant bit, and since we write numbers with the most-significant digit on the left, we also write bytes with the most-significant bit on the left. In the diagrams below, we number the bits of a byte so that bit 0 is the least-significant bit, i.e., the bits are numbered:

```
+--------+
|76543210|
+--------+
```

Within a computer, a number may occupy multiple bytes. All multi-byte numbers in the format described here are stored with the least-significant byte first (at the lower memory address). For example, the decimal number 520 is stored as:

```
    0        1
+--------+--------+
|00001000|00000010|
+--------+--------+
 ^        ^
 |        |
 |        + more significant byte = 2 x 256
 + less significant byte = 8
```

3.1.1 Packing into bytes

This document does not address the issue of the order in which bits of a byte are transmitted on a bit-sequential medium, since the final data format described here is byte- rather than bit-oriented. However, we describe the compressed block format in below, as a sequence of data elements of various bit lengths, not a sequence of bytes. We must therefore specify how to pack these data elements into bytes to form the final compressed byte sequence:

- Data elements are packed into bytes in order of increasing bit number within the byte, i.e., starting with the least- significant bit of the byte.
- Data elements other than Huffman codes are packed starting with the least-significant bit of the data element.
- Huffman codes are packed starting with the most-significant bit of the code.

In other words, if one were to print out the compressed data as a sequence of bytes, starting with the first byte at the *right* margin and proceeding to the *left*, with the most-significant bit of each byte on the left as usual, one would be able to parse the result from right to left, with fixed-width elements in the correct MSB-to-LSB order and Huffman codes in bit-reversed order (i.e., with the first bit of the code in the relative LSB position).

3.2 Compressed block format

3.2.1 Synopsis of prefix and Huffman coding

Prefix coding represents symbols from an *a priori* known alphabet by bit sequences (codes), one code for each symbol, in a manner such that different symbols may be represented by bit sequences of different lengths, but a parser can always parse an encoded string unambiguously symbol-by-symbol.

We define a prefix code in terms of a binary tree in which the two edges descending from each non-leaf node are labeled 0 and 1 and in which the leaf nodes correspond one-for-one with (are labeled with) the symbols of the alphabet; then the code for a symbol is the sequence of 0's and 1's on the edges leading from the root to the leaf labeled with that symbol. For example:

```
      /\               Symbol    Code
     0  1              ------    ----
    /    \                 A      00
   /\     B                B       1
  0  1                     C     011
 /    \                    D     010
A     /\
     0  1
    /    \
   D      C
```

A parser can decode the next symbol from an encoded input stream by walking down the tree from the root, at each step choosing the edge corresponding to the next input bit.

Given an alphabet with known symbol frequencies, the Huffman algorithm allows the construction of an optimal prefix code (one which represents strings with those symbol frequencies using the fewest bits of any possible prefix codes for that alphabet). Such a code is called a Huffman code. (See reference [1] in Chapter 5, references for additional information on Huffman codes.)

Note that in the "deflate" format, the Huffman codes for the various alphabets must not exceed certain maximum code lengths. This constraint complicates the algorithm for computing code lengths from symbol frequencies. Again, see Chapter 5, references for details.

3.2.2 Use of Huffman coding in the "deflate" format

The Huffman codes used for each alphabet in the "deflate" format have two additional rules:

- All codes of a given bit length have lexicographically consecutive values, in the same order as the symbols they represent;
- Shorter codes lexicographically precede longer codes.

We could recode the example above to follow this rule as follows, assuming that the order of the alphabet is ABCD:

```
Symbol  Code
------  ----
A       10
B       0
C       110
D       111
```

I.e., 0 precedes 10 which precedes 11x, and 110 and 111 are lexicographically consecutive.

Given this rule, we can define the Huffman code for an alphabet just by giving the bit lengths of the codes for each symbol of the alphabet in order; this is sufficient to determine the actual codes. In our example, the code is completely defined by the sequence of bit lengths (2, 1, 3, 3). The following algorithm generates the codes as integers, intended to be read from most- to least-significant bit. The code lengths are initially in tree[I].Len; the codes are produced in tree[I].Code.

1) Count the number of codes for each code length. Let bl_count[N] be the number of codes of length N, N >= 1.

2) Find the numerical value of the smallest code for each code length:

```
code = 0;
bl_count[0] = 0;
for (bits = 1; bits <= MAX_BITS; bits++) {
    code = (code + bl_count[bits-1]) << 1;
    next_code[bits] = code;
}
```

3) Assign numerical values to all codes, using consecutive values for all codes of the same length with the base values determined at step 2. Codes that are never used (which have a bit length of zero) must not be assigned a value.

```
for (n = 0;  n <= max_code; n++) {
    len = tree[n].Len;
    if (len != 0) {
        tree[n].Code = next_code[len];
        next_code[len]++;
    }
}
```

Example:

Consider the alphabet ABCDEFGH, with bit lengths (3, 3, 3, 3, 3, 2, 4, 4). After step 1, we have:

```
N      bl_count[N]
-      -----------
2      1
3      5
4      2
```

Step 2 computes the following next_code values:

```
N      next_code[N]
-      ------------
1      0
2      0
3      2
4      14
```

Step 3 produces the following code values:

```
Symbol Length   Code
------ ------   ----
A       3        010
B       3        011
C       3        100
D       3        101
E       3        110
F       2         00
G       4       1110
H       4       1111
```

3.2.3 Details of block format

Each block of compressed data begins with 3 header bits containing the following data:

```
first bit       BFINAL
next 2 bits     BTYPE
```

Note that the header bits do not necessarily begin on a byte boundary, since a block does not necessarily occupy an integral number of bytes.

BFINAL is set iff this is the last block of the data set.

BTYPE specifies how the data are compressed, as follows:

```
00 - no compression
01 - compressed with fixed Huffman codes
10 - compressed with dynamic Huffman codes
11 - reserved (error)
```

The only difference between the two compressed cases is how the Huffman codes for the literal/ length and distance alphabets are defined.

In all cases, the decoding algorithm for the actual data is as follows:

```
do
   read block header from input stream.
   if stored with no compression
      skip any remaining bits in current partially
         processed byte
      read LEN and NLEN (see next section)
      copy LEN bytes of data to output
   otherwise
      if compressed with dynamic Huffman codes
         read representation of code trees (see
            subsection below)
      loop (until end of block code recognized)
         decode literal/length value from input stream
         if value < 256
            copy value (literal byte) to output stream
         otherwise
            if value = end of block (256)
               break from loop
            otherwise (value = 257..285)
               decode distance from input stream

               move backwards distance bytes in the output
               stream, and copy length bytes from this
               position to the output stream.
      end loop
while not last block
```

Note that a duplicated string reference may refer to a string in a previous block; i.e., the backward distance may cross one or more block boundaries. However a distance cannot refer past the beginning of the output stream. (An application using a preset dictionary might discard part of the output stream; a distance can refer to that part of the output stream anyway) Note also that the referenced string may overlap the current position; for example, if the last 2 bytes decoded have values X and Y, a string reference with <length = 5, distance = 2> adds X,Y,X,Y,X to the output stream.

We now specify each compression method in turn.

3.2.4 Non-compressed blocks (BTYPE=00)

Any bits of input up to the next byte boundary are ignored. The rest of the block consists of the following information:

```
  0   1   2   3   4...
+---+---+---+---+================================+
|  LEN  | NLEN  |... LEN bytes of literal data...|
+---+---+---+---+================================+
```

LEN is the number of data bytes in the block. NLEN is the one's complement of LEN.

3.2.5 Compressed blocks (length and distance codes)

As noted above, encoded data blocks in the "deflate" format consist of sequences of symbols drawn from three conceptually distinct alphabets: either literal bytes, from the alphabet of byte values (0..255), or <length, backward distance> pairs, where the length is drawn from (3..258) and the distance is drawn from (1..32,768). In fact, the literal and length alphabets are merged into a single alphabet (0..285), where values 0..255 represent literal bytes, the value 256 indicates end-of-block, and values 257..285 represent length codes (possibly in conjunction with extra bits following the symbol code) as follows:

```
     Extra               Extra               Extra
Code Bits Length(s) Code Bits Lengths   Code Bits Length(s)
---- ---- ------     ---- ---- -------   ---- ---- -------
 257   0     3       267   1   15,16     277   4   67-82
 258   0     4       268   1   17,18     278   4   83-98
 259   0     5       269   2   19-22     279   4   99-114
 260   0     6       270   2   23-26     280   4  115-130
 261   0     7       271   2   27-30     281   5  131-162
 262   0     8       272   2   31-34     282   5  163-194
 263   0     9       273   3   35-42     283   5  195-226
 264   0    10       274   3   43-50     284   5  227-257
 265   1  11,12      275   3   51-58     285   0    258
 266   1  13,14      276   3   59-66
```

The extra bits should be interpreted as a machine integer stored with the most-significant bit first, e.g., bits 1110 represent the value 14.

```
      Extra           Extra               Extra
 Code Bits Dist  Code Bits   Dist     Code Bits Distance
 ---- ---- ----  ---- ----  ------    ---- ---- --------
   0   0    1     10   4     33-48    20    9   1025-1536
   1   0    2     11   4     49-64    21    9   1537-2048
   2   0    3     12   5     65-96    22   10   2049-3072
   3   0    4     13   5     97-128   23   10   3073-4096
   4   1   5,6    14   6    129-192   24   11   4097-6144
   5   1   7,8    15   6    193-256   25   11   6145-8192
   6   2   9-12   16   7    257-384   26   12  8193-12288
   7   2  13-16   17   7    385-512   27   12 12289-16384
   8   3  17-24   18   8    513-768   28   13 16385-24576
   9   3  25-32   19   8   769-1024   29   13 24577-32768
```

3.2.6 Compression with fixed Huffman codes (BTYPE=01)

The Huffman codes for the two alphabets are fixed, and are not represented explicitly in the data. The Huffman code lengths for the literal/length alphabet are:

```
Lit Value    Bits        Codes
---------    ----        -----
  0 - 143     8          00110000 through
                         10111111
144 - 255     9          110010000 through
                         111111111
256 - 279     7          0000000 through
                         0010111
280 - 287     8          11000000 through
                         11000111
```

The code lengths are sufficient to generate the actual codes, as described above; we show the codes in the table for added clarity. Literal/length values 286-287 will never actually occur in the compressed data, but participate in the code construction.

Distance codes 0-31 are represented by (fixed-length) 5-bit codes, with possible additional bits as shown in the table shown in Paragraph 3.2.5, above. Note that distance codes 30-31 will never actually occur in the compressed data.

3.2.7 Compression with dynamic Huffman codes (BTYPE=10)

The Huffman codes for the two alphabets appear in the block immediately after the header bits and before the actual compressed data, first the literal/length code and then the distance code. Each code is defined by a sequence of code lengths, as discussed in Paragraph 3.2.2, above. For even greater compactness, the code length sequences themselves are compressed using a Huffman code. The alphabet for code lengths is as follows:

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```

A code length of 0 indicates that the corresponding symbol in the literal/length or distance alphabet will not occur in the block, and should not participate in the Huffman code construction algorithm given earlier. If only one distance code is used, it is encoded using one bit, not zero bits; in this case there is a single code length of one, with one unused code. One distance code of zero bits means that there are no distance codes used at all (the data is all literals).

We can now define the format of the block:

```
5 Bits: HLIT, # of Literal/Length codes - 257 (257 - 286)
5 Bits: HDIST, # of Distance codes - 1        (1 - 32)
4 Bits: HCLEN, # of Code Length codes - 4     (4 - 19)

(HCLEN + 4) x 3 bits: code lengths for the code length
   alphabet given just above, in the order: 16, 17, 18,
   0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15

   These code lengths are interpreted as 3-bit integers
   (0-7); as above, a code length of 0 means the
   corresponding symbol (literal/length or distance code
   length) is not used.

HLIT + 257 code lengths for the literal/length alphabet,
   encoded using the code length Huffman code

HDIST + 1 code lengths for the distance alphabet,
   encoded using the code length Huffman code

The actual compressed data of the block,
   encoded using the literal/length and distance Huffman
   codes

The literal/length symbol 256 (end of data),
   encoded using the literal/length Huffman code
```

The code length repeat codes can cross from HLIT + 257 to the HDIST + 1 code lengths. In other words, all code lengths form a single sequence of HLIT + HDIST + 258 values.

3.3 Compliance

A compressor may limit further the ranges of values specified in the previous section and still be compliant; for example, it may limit the range of backward pointers to some value smaller than 32K. Similarly, a compressor may limit the size of blocks so that a compressible block fits in memory.

A compliant decompressor must accept the full range of possible values defined in the previous section, and must accept blocks of arbitrary size.

4 Compression algorithm details

While it is the intent of this document to define the "deflate" compressed data format without reference to any particular compression algorithm, the format is related to the compressed formats produced by LZ77 (Lempel-Ziv 1977, see reference [2] below); since many variations of LZ77 are patented, it is strongly recommended that the implementor of a compressor follow the general algorithm presented here, which is known not to be patented per se. The material in this section is not part of the definition of the specification per se, and a compressor need not follow it in order to be compliant.

The compressor terminates a block when it determines that starting a new block with fresh trees would be useful, or when the block size fills up the compressor's block buffer.

The compressor uses a chained hash table to find duplicated strings, using a hash function that operates on 3-byte sequences. At any given point during compression, let XYZ be the next 3 input bytes to be examined (not necessarily all different, of course). First, the compressor examines the hash chain for XYZ. If the chain is empty, the compressor simply writes out X as a literal byte and advances one byte in the input. If the hash chain is not empty, indicating that the sequence XYZ (or, if we are unlucky, some other 3 bytes with the same hash function value) has occurred recently, the compressor compares all strings on the XYZ hash chain with the actual input data sequence starting at the current point, and selects the longest match.

The compressor searches the hash chains starting with the most recent strings, to favor small distances and thus take advantage of the Huffman encoding. The hash chains are singly linked. There are no deletions from the hash chains; the algorithm simply discards matches that are too old. To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a run-time parameter.

To improve overall compression, the compressor optionally defers the selection of matches ("lazy matching"): after a match of length N has been found, the compressor searches for a longer match starting at the next input byte. If it finds a longer match, it truncates the previous match to a length of one (thus producing a single literal byte) and then emits the longer match. Otherwise, it emits the original match, and, as described above, advances N bytes before continuing.

Run-time parameters also control this "lazy match" procedure. If compression ratio is most important, the compressor attempts a complete second search regardless of the length of the first match. In the normal case, if the current match is "long enough", the compressor reduces the search for a longer match, thus speeding up the process. If speed is most important, the compressor inserts new strings in the hash table only when no match was found, or when the match is not "too long". This degrades the compression ratio but saves time since there are both fewer insertions and fewer searches.

5 References

[1] Huffman, D. A., "A Method for the Construction of Minimum Redundancy Codes", Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp. 1098-1101.

[2] Ziv J., Lempel A., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337-343.

[3] Gailly, J.-L., and Adler, M., ZLIB documentation and sources, available in ftp://ftp.uu.net/pub/archiving/zip/doc/

[4] Gailly, J.-L., and Adler, M., GZIP documentation and sources, available as gzip-*.tar in ftp://prep.ai.mit.edu/pub/gnu/

[5] Schwartz, E. S., and Kallick, B. "Generating a canonical prefix encoding." Comm. ACM, 7,3 (Mar. 1964), pp. 166-169.

[6] "Efficient decoding of prefix codes", Hirschberg and Lelewer, Comm. ACM, 33,4, April 1990, pp. 449-459.

6 Security considerations

Any data compression method involves the reduction of redundancy in the data. Consequently, any corruption of the data is likely to have severe effects and be difficult to correct. Uncompressed text, on the other hand, will probably still be readable despite the presence of some corrupted bytes.

It is recommended that systems using this data format provide some means of validating the integrity of the compressed data. See reference [3], for example.

7 Source code

Source code for a C language implementation of a "deflate" compliant compressor and decompressor is available within the zlib package at ftp://ftp.uu.net/pub/archiving/zip/zlib/.

8 Acknowledgements

Trademarks cited in this document are the property of their respective owners.

Phil Katz designed the deflate format. Jean-Loup Gailly and Mark Adler wrote the related software described in this specification. Glenn Randers-Pehrson converted this document to Internet Draft and HTML format.

9 Author's address

L. Peter Deutsch

```
Aladdin Enterprises
203 Santa Margarita Ave.
Menlo Park, CA 94025

Phone: (415) 322-0103 (AM only)
FAX:   (415) 322-1734
EMail: <ghost@aladdin.com>
```


Questions about the technical content of this specification can be sent by email to

```
Jean-loup Gailly <gzip@prep.ai.mit.edu> and
Mark Adler <madler@alumni.caltech.edu>
```

Editorial comments on this specification can be sent by email to

```
L. Peter Deutsch <ghost@aladdin.com> and
Glenn Randers-Pehrson <randeg@alumni.rpi.edu>
```