Brotli Compressed Data Format
draft-alakuijala-brotli-04

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.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on November 11, 2015.

Copyright Notice

Copyright (c) 2015 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Table of Contents

1. Introduction ................................................. 3
   1.1. Purpose .............................................. 3
   1.2. Intended audience ...................................... 3
   1.3. Scope ................................................ 3
   1.4. Compliance ........................................... 4
   1.5. Definitions of terms and conventions used ............. 4
       1.5.1. Packing into bytes ............................ 4
2. Compressed representation overview .......................... 5
3. Compressed representation of prefix codes .................. 9
   3.1. Introduction to prefix coding .......................... 9
   3.2. Use of prefix coding in the brotli format .............. 10
   3.3. Alphabet sizes ....................................... 12
   3.4. Simple prefix codes ................................... 12
   3.5. Complex prefix codes .................................. 13
4. Encoding of distances ....................................... 15
5. Encoding of literal insertion lengths and copy lengths ..... 17
6. Encoding of block switch commands .......................... 19
7. Context modeling ............................................ 21
   7.1. Context modes and context ID lookup for literals ....... 21
   7.2. Context ID for distances .............................. 23
   7.3. Encoding of the context map ........................... 23
8. Static dictionary ............................................ 25
9. Compressed data format ...................................... 27
   9.1. Format of the stream header ............................ 27
   9.2. Format of the meta-block header ....................... 28
   9.3. Format of the meta-block data ........................ 30
10. Decoding algorithm ......................................... 31
11. Security Considerations .................................... 33
12. IANA Considerations ........................................ 34
13. Informative References .................................... 34
14. Source code ................................................. 34
15. Acknowledgements .......................................... 34
Appendix A. Static dictionary data ............................ 34
Appendix B. List of word transformations ...................... 115
Authors’ Addresses .............................................. 118
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 a compression ratio comparable to the best currently available general-purpose compression methods, and in particular considerably better than the gzip program;
* Decompresses much faster than current LZMA implementations.

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.

1.2. Intended audience

This specification is intended for use by software implementers to compress data into and/or decompress data from the brotli 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.

This specification uses heavily the notations and terminology introduced in the DEFLATE format specification [RFC 1951]. For the sake of completeness, we always include the whole text of the relevant parts of RFC 1951, therefore familiarity with the DEFLATE format is helpful but not required.

The compressed data format defined in this specification is an integral part of the WOFF 2.0 web font file format [WOFF2], therefore this specification is also intended for implementers of WOFF 2.0 compressors and decompressors.

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.

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:

```
+--------+
|7 6 5 4 3 2 1 0|
+--------+
```

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
+-------+-------+
|0 0 0 0 1 0 0 0|0 0 0 0 0 0 1 0|
+-------+-------+
```

^        ^
|        |
+ more significant byte = 2 x 256
+ less significant byte = 8

1.5.1. 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 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 prefix codes are packed starting with the least-significant bit of the data element. These are referred to here as integer values and are considered unsigned.
- Prefix 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 prefix codes in bit-reversed order (i.e., with the first bit of the code in the relative LSB position).

2. Compressed representation overview

A compressed data set consists of a header and a series of meta-blocks. Each meta-block decompresses to a sequence of 1 to 16,777,216 (16 MiB) uncompressed bytes. The final uncompressed data is the concatenation of the uncompressed sequences from each meta-block.

The header contains the size of the sliding window that was used during compression. The decompressor must retain at least that amount of uncompressed data prior to the current position in the stream, in order to be able to decompress what follows. The sliding window size is a power of two, minus 16, where the power is in the range of 16 to 24. The possible sliding window sizes range from 64 KiB – 16 B to 16 MiB – 16 B.

Each meta-block is compressed using a combination of the LZ77 algorithm (Lempel-Ziv 1977, [LZ77]) and Huffman coding. The result of Huffman coding is referred to here as a prefix code. The prefix codes for each meta-block are independent of those for previous or subsequent meta-blocks; the LZ77 algorithm may use a reference to a duplicated string occurring in a previous meta-block, up to the sliding window size of uncompressed bytes before. In addition, in the brotli format, a string reference may instead refer to a static dictionary entry.
Each meta-block consists of two parts: a meta-block header that describes the representation of the compressed data part, and a compressed data part. The compressed data consists of a series of commands. Each command consists of two parts: a sequence of literal bytes (of strings that have not been detected as duplicated within the sliding window), and a pointer to a duplicated string, represented as a pair <length, backward distance>. There can be zero literal bytes in the command. The minimum length of the string to be duplicated is two, but the last command in the meta-block is permitted to have only literals and no pointer to a string to duplicate.

Each command in the compressed data is represented using three categories of prefix codes: one set of prefix codes are for the literal sequence lengths (also referred to as literal insertion lengths) and backward copy lengths (that is, a single code word represents two lengths, one of the literal sequence and one of the backward copy), a separate set of prefix codes are for literals, and a third set of prefix codes are for distances. The prefix code descriptions for each meta-block appear in a compact form just before the compressed data in the meta-block header. The insert and copy length and distance prefix codes may be followed by extra bits that are added to the base values determined by the codes. The number of extra bits is determined by the code.

One meta-block command then appears as a sequence of prefix codes:

   Insert and copy length, literal, literal, ..., literal, distance

where the insert and copy defines the number of literals that immediately follow and the copy length, and the distance defines how far back to go for the copy, used in combination with the copy length. The resulting uncompressed data is the sequence of bytes:

   literal, literal, ..., literal, copy, copy, ..., copy

where the number of literal bytes and copy bytes are determined by the insert and copy length code. (The number of bytes copied for a static dictionary entry can vary from the copy length.)

The last command in the meta-block may end with the last literal if the total uncompressed length of the meta-block has been satisfied. In that case there is no distance in the last command, and the copy length is ignored.

There can be more than one prefix code for each category, where the prefix code to use for the next element of that category is determined by the context of the compressed stream that precedes that
element. Part of that context is three current block types, one for each category. A block type is in the range of 0..255. For each category there is a count of how many elements of that category remain to be decoded using the current block type. Once that count is expended, a new block type and block count is read from the stream immediately preceding the next element of that category, which will use the new block type.

The insert and copy block type directly determines which prefix code to use for the next insert and copy element. For the literal and distance elements, the respective block type is used in combination with other context information to determine which prefix code to use for the next element.

Consider the following example:

(IaC0, L0, L1, L2, D0)(IaC1, D1)(IaC2, L3, L4, D2)(IaC3, L5, D3)

The meta-block here has four commands, contained in parentheses for clarity, where each of the three categories of symbols within these commands can be interpreted using different block types. Here we separate out each category as its own sequence to show an example of block types assigned to those elements. Each square-bracketed group is a block that uses the same block type:

[IaC0, IaC1][IaC2, IaC3]  <-- insert-and-copy: block types 0 and 1
[L0, L1][L2, L3, L4][L5]  <-- literals: block types 0, 1, and 0
[D0][D1, D2, D3]          <-- distances: block types 0 and 1

The subsequent blocks within each block category must have different block types, but we see that block types can be reused later in the meta-block. The block types are numbered from 0 to the maximum block type number of 255 and the first block of each block category is type 0. The block structure of a meta-block is represented by the sequence of block-switch commands for each block category, where a block-switch command is a pair <block type, block count>. The block-switch commands are represented in the compressed data before the start of each new block using a prefix code for block types and a separate prefix code for block counts for each block category. For the above example the physical layout of the meta-block is then:

IaC0 L0 L1 LBlockSwitch(1, 3) L2 D0 IaC1 DBlockSwitch(1, 3) D1
IaCBlockSwitch(1, 2) IaC2 L3 L4 D2 IaC3 LBlockSwitch(0, 1) L5 D3

where *BlockSwitch(t, n) switches to block type t for a count of n elements. Note that in this example DBlockSwitch(1, 3) immediately
precedes the next required distance D1. It does not follow the last
distance of the previous block, D0. Whenever an element of a category
is needed, and the block count for that category has reached zero,
then a new block type and count is read from the stream just before
reading that next element.

The block switch commands for the first blocks of each category are
not part of the meta-block compressed data. Instead the first block
type is defined to be 0, and the first block count for each category
is encoded in the meta-block header. The prefix codes for the block
types and counts, a total of six prefix codes over the three
categories, are defined in a compact form in the meta-block header.

Each category of value (insert-and-copy lengths, literals and
distances) can be encoded with any prefix code from a collection of
prefix codes belonging to the same category appearing in the meta-
block header. The particular prefix code used can depend on two
factors: the block type of the block the value appears in, and the
context of the value. In the case of the literals, the context is
the previous two bytes in the uncompressed data, and in the case of
distances, the context is the copy length from the same command. For
insert-and-copy lengths, no context is used and the prefix code
depends only on the block type. In the case of literals and
distances, the context is mapped to a context ID in the range 0..63
for literals and 0..3 for distances and the matrix of the prefix code
indices for each block type and context ID, called the context map,
is encoded in a compact form in the meta-block header.

For example, the prefix code to use to decode L2 depends on the block
type (1), and the literal context ID determined by the two
uncompressed bytes that were decoded from L0 and L1. Similarly, the
prefix code to use to decode D0 depends on the block type (0), and
the distance context ID determined by the copy length decoded from
IaC0. The prefix code to use to decode IaC3 depends only on the block
type (1).

In addition to the parts listed above (prefix code for insert- and-
copy lengths, literals, distances, block types and block counts and
the context map), the meta-block header contains the number of
uncompressed bytes coded in the meta-block and two additional
parameters used in the representation of match distances: the number
of postfix bits and the number of direct distance codes.

A compressed meta-block may be marked in the header as the last meta-
block, which terminates the compressed stream.

A meta-block may instead simply store the uncompressed data directly
as bytes on byte boundaries with no coding or matching strings. In
this case the meta-block header information only contains the number of uncompressed bytes and the indication that the meta-block is uncompressed. An uncompressed meta-block cannot be the last meta-block.

A meta-block may also be empty, which generates no uncompressed data at all. An empty meta-block may contain metadata information as bytes starting on byte boundaries, which are not part of either the sliding window or the uncompressed data. Thus, these metadata bytes can not be used to create matching strings in subsequent meta-blocks and are not used as context bytes for literals.

3. Compressed representation of prefix codes

3.1. Introduction to prefix 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:

```
/
\ 
0 1
/
\ 
A
/
\ 
B
0 1
/
\ 
C
D
```

A parser can decode the next symbol from the compressed stream by walking down the tree from the root, at each step choosing the edge corresponding to the next compressed data 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
prefix code is called a Huffman code. (See [HUFFMAN] in Chapter 5, references for additional information on Huffman codes.)

Note that in the brotli format, the prefix 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. Use of prefix coding in the brotli format

The prefix codes used for each alphabet in the brotli format are canonical prefix codes, which 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>110</td>
</tr>
<tr>
<td>D</td>
<td>111</td>
</tr>
</tbody>
</table>

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

Given this rule, we can define the canonical prefix 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:

<table>
<thead>
<tr>
<th>N</th>
<th>bl_count[N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 2 computes the following next_code values:

<table>
<thead>
<tr>
<th>N</th>
<th>next_code[N]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>

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

### 3.3. Alphabet sizes

Prefix codes are used for different purposes in the brotli format, and each purpose has a different alphabet size. For literal codes the alphabet size is 256. For insert-and-copy length codes the alphabet size is 704. For block count codes, the alphabet size is 26. For distance codes, block type codes and the prefix codes used in compressing the context map, the alphabet size is dynamic and is based on other parameters.

### 3.4. Simple prefix codes

The first two bits of the compressed representation of each prefix code distinguishes between simple and complex prefix codes. If this value is 1, then a simple prefix code follows. Otherwise the value indicates the number of leading zeros.

A simple prefix code can have only up to four symbols with non-zero code length. The format of the simple prefix code is as follows:

2 bits: value of 1 indicates a simple prefix code
2 bits: NSYM - 1, where NSYM = # of symbols coded

NSYM symbols, each encoded using ALPHABET_BITS bits

1 bit: tree-select, present only for NSYM = 4

The value of ALPHABET_BITS depends on the alphabet of the prefix code: it is the smallest number of bits that can represent all symbols in the alphabet. E.g. for the alphabet of literal bytes, ALPHABET_BITS is 8. The value of each of the NSYM symbols above is the value of the ALPHABETS_BITS width integer value. (If the integer value is greater than or equal to the alphabet size, then the stream should be rejected as invalid.)

Note that the NSYM symbols may not be presented in sorted order. Prefix codes of the same bit length must be assigned to the symbols
in sorted order.

The (non-zero) code lengths of the symbols can be reconstructed as follows:

* if NSYM = 1, the code length for the one symbol is zero -- when encoding this symbol in the compressed data stream using this prefix code, no actual bits are emitted. Similarly, when decoding a symbol using this prefix code, no bits are read and the one symbol is returned.

* if NSYM = 2, both symbols have code length 1.

* if NSYM = 3, the code lengths for the symbols are 1, 2, 2 in the order they appear in the representation of the simple prefix code.

* if NSYM = 4, the code lengths (in order of symbols decoded) depend on the tree-select bit: 2, 2, 2, 2, (tree-select bit 0) or 1, 2, 3, 3 (tree-select bit 1).

3.5. Complex prefix codes

A complex prefix code is a canonical prefix code, defined by the sequence of code lengths, as discussed in Paragraph 3.2, above. For even greater compactness, the code length sequences themselves are compressed using a prefix code. The alphabet for code lengths is as follows:

0 - 15: Represent code lengths of 0 - 15
16: Copy the previous non-zero code length 3 - 6 times
   The next 2 bits indicate repeat length
   (0 = 3, ... , 3 = 6)
   If this is the first code length, or all previous code lengths are zero, a code length of 8 is repeated 3 - 6 times
   A repeated code length code of 16 modifies the repeat count of the previous one as follows:
   repeat count = (4 * (repeat count - 2)) + (3 - 6 on the next 2 bits)
   Example: Codes 7, 16 (+2 bits 11), 16 (+2 bits 10)
            will expand to 22 code lengths of 7
            (1 + 4 * (6 - 2) + 5)
17: Repeat a code length of 0 for 3 - 10 times.
   (3 bits of length)
   A repeated code length code of 17 modifies the repeat count of the previous one as follows:
repeat count = (8 * (repeat count - 2)) +
(3 - 10 on the next 3 bits)

Note that a code of 16 that follows an immediately preceding 16 modifies the previous repeat count, which becomes the new repeat count. The same is true for a 17 following a 17. A sequence of three or more 16 codes in a row or three of more 17 codes in a row is possible, modifying the count each time. Only the final repeat count is used. The modification only applies if the same code follows. A 16 repeat does not modify an immediately preceding 17 count, nor vice versa.

A code length of 0 indicates that the corresponding symbol in the alphabet will not occur in the compressed data, and should not participate in the prefix code construction algorithm given earlier. A complex prefix code must have at least two non-zero code lengths.

The bit lengths of the prefix code over the code length alphabet are compressed with the following static prefix code (where the bits shown are reversed in the actual compressed stream):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>1110</td>
</tr>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>3</td>
<td>01</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>111</td>
</tr>
</tbody>
</table>

We can now define the format of the complex prefix code as follows:

2 bits: HSKIP, values of 0, 2 or 3 represent the respective number of skipped code lengths. The skipped lengths are taken to be zero. (An HSKIP of 1 indicates a Simple prefix code.)

Code lengths for symbols in the code length alphabet given just above, in the order: 1, 2, 3, 4, 0, 5, 17, 6, 16, 7, 8, 9, 10, 11, 12, 13, 14, 15. If HSKIP is 2, then the code lengths for symbols 1 and 2 are zero, and the first code length is for symbol 3. If HSKIP is 3, then the code length for symbol 3 is also zero, and the first code length is for symbol 4.

The code lengths of code length symbols are between 0 and 5 and they are represented with 2 - 4 bits according to the static prefix code above. A code length of 0 means
the corresponding code length symbol is not used.

If HSKIP is 2 or 3, a respective number of leading code lengths are implicit zeros and are not present in the code lengths sequence above.

If there are at least two non-zero code lengths, any trailing zero code lengths are omitted, i.e. the last code length in the sequence must be non-zero. In this case the sum of \((32 \gg \text{code length})\) over all the non-zero code lengths must equal to 32.

If the lengths have been read for the entire code length alphabet and there was only one non-zero code length, then the prefix code has one symbol whose code has zero length. In this case, that symbol results in no bits being emitted by the compressor, and no bits consumed by the decompressor. That single symbol is immediately returned when this code is decoded.
An example of where this occurs is if the entire code to be represented has symbols of length 8. E.g. a literal code that represents all literal values with equal probability. In this case the single symbol is 16, which repeats the previous length. The previous length is taken to be 8 before any code length code lengths are read.

Sequence of code lengths symbols, encoded using the code length prefix code. Any trailing 0 or 17 must be omitted, i.e. the last encoded code length symbol must be between 1 and 16. The sum of \((32768 \gg \text{code length})\) over all the non-zero code lengths in the alphabet, including those encoded using repeat code(s) of 16, must equal to 32768.

4. Encoding of distances

As described in Section 2, one component of a compressed meta-block is a sequence of backward distances. In this section we provide the details to the encoding of distances.

Each distance in the compressed data part of a meta-block is represented with a pair \(<\text{distance code, extra bits}>\). The distance code and the extra bits are encoded back-to-back, the distance code is encoded using a prefix code over the distance alphabet, while the extra bits value is encoded as a fixed-width integer value. The number of extra bits can be 0 – 24, and it is dependent on the distance code.
To convert a distance code and associated extra bits to a backward
distance, we need the sequence of past distances and two additional
parameters, the number of "postfix bits", denoted by NPOSTFIX (0..3),
and the number of direct distance codes, denoted by NDIRECT (0..120).
Both of these parameters are encoded in the meta-block header. We
will also use the following derived parameter:

\[
\text{POSTFIX\_MASK} = ((1 \ll NPOSTFIX) - 1)
\]

The first 16 distance symbols are special symbols that reference past
distances as follows:

0: last distance
1: second-to-last distance
2: third-to-last distance
3: fourth-to-last distance
4: last distance - 1
5: last distance + 1
6: last distance - 2
7: last distance + 2
8: last distance - 3
9: last distance + 3
10: second-to-last distance - 1
11: second-to-last distance + 1
12: second-to-last distance - 2
13: second-to-last distance + 2
14: second-to-last distance - 3
15: second-to-last distance + 3

The ring buffer of four last distances is initialized by the values
16, 15, 11 and 4 (i.e. the fourth-to-last is set to 16, the third-to-
last to 15, the second-to-last to 11 and the last distance to 4) at
the beginning of the *stream* (as opposed to the beginning of the
meta-block) and it is not reset at meta-block boundaries. When a
distance symbol 0 appears, the distance it represents (i.e. the last
distance in the sequence of distances) is not pushed to the ring
buffer of last distances, in other words, the expression "(second,
third, fourth)-to-last distance" means the (second, third,
fourth)-to-last distance that was not represented by a 0 distance
symbol. Similarly, distances that represent static dictionary words
(see Section 8.) are not pushed to the ring buffer of last distances.

The next NDIRECT distance symbols, from 16 to 15 + NDIRECT, represent
distances from 1 to NDIRECT. Neither the distance special symbols,
nor the NDIRECT direct distance symbols are followed by any extra
bits.

Distance symbols 16 + NDIRECT and greater all have extra bits, where
the number of extra bits for a distance symbol "dcode" is given by the following formula:

\[ \text{ndistbits} = 1 + ((\text{dcode} - \text{NDIRECT} - 16) \gg (\text{NPOSTFIX} + 1)) \]

The maximum number of extra bits is 24, therefore the size of the distance symbol alphabet is \((16 + \text{NDIRECT} + (48 \ll \text{NPOSTFIX}))\).

Given a distance symbol "dcode" (\(\geq 16 + \text{NDIRECT}\)), and extra bits "dextra", the backward distance is given by the following formula:

\[
\begin{align*}
\text{hcode} & = (\text{dcode} - \text{NDIRECT} - 16) \gg \text{NPOSTFIX} \\
\text{lcode} & = (\text{dcode} - \text{NDIRECT} - 16) \& \text{POSTFIX}\_\text{MASK} \\
\text{offset} & = ((2 + (\text{hcode} \& 1)) \ll \text{ndistbits}) - 4; \\
\text{distance} & = ((\text{offset} + \text{dextra}) \ll \text{NPOSTFIX}) + \text{lcode} + \text{NDIRECT} + 1
\end{align*}
\]

5. Encoding of literal insertion lengths and copy lengths

As described in Section 2, the literal insertion lengths and backward copy lengths are encoded using a single prefix code. This section provides the details to this encoding.

Each \(<\text{insertion length}, \text{copy length}>\) pair in the compressed data part of a meta-block is represented with the following triplet:

\(<\text{insert-and-copy length code}, \text{insert extra bits}, \text{copy extra bits}>\)

The insert-and-copy length code, the insert extra bits and the copy extra bits are encoded back-to-back, the insert-and-copy length code is encoded using a prefix code over the insert-and-copy length code alphabet, while the extra bits values are encoded as fixed-width integer values. The number of insert and copy extra bits can be 0 - 24, and they are dependent on the insert-and-copy length code.

Some of the insert-and-copy length codes also express the fact that the distance symbol of the distance in the same command is 0, i.e. the distance component of the command is the same as that of the previous command. In this case, the distance code and extra bits for the distance are omitted from the compressed data stream.

We describe the insert-and-copy length code alphabet in terms of the (not directly used) insert length code and copy length code alphabets. The symbols of the insert length code alphabet, along with the number of insert extra bits and the range of the insert lengths are as follows:
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0 2 10-13</td>
<td>16 6 130-193</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1 2 14-17</td>
<td>17 7 194-321</td>
</tr>
<tr>
<td>2 0 2</td>
<td>3 18-25</td>
<td>18 8 322-577</td>
</tr>
<tr>
<td>3 0 3</td>
<td>11 26-33</td>
<td>19 9 578-1089</td>
</tr>
<tr>
<td>4 0 4</td>
<td>12 34-49</td>
<td>20 10 1090-2113</td>
</tr>
<tr>
<td>5 0 5</td>
<td>13 50-65</td>
<td>21 12 2114-6209</td>
</tr>
<tr>
<td>6 1 6,7</td>
<td>14 5 66-97</td>
<td>22 14 6210-22593</td>
</tr>
<tr>
<td>7 1 8,9</td>
<td>15 5 98-129</td>
<td>23 24 22594-16799809</td>
</tr>
</tbody>
</table>

The symbols of the copy length code alphabet, along with the number of copy extra bits and the range of copy lengths are as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 2</td>
<td>1 10,11</td>
<td>16 5 70-101</td>
</tr>
<tr>
<td>1 0 3</td>
<td>9 12,13</td>
<td>17 5 102-133</td>
</tr>
<tr>
<td>2 0 4</td>
<td>10 2 14-17</td>
<td>18 6 134-197</td>
</tr>
<tr>
<td>3 0 5</td>
<td>11 18-21</td>
<td>19 7 198-325</td>
</tr>
<tr>
<td>4 0 6</td>
<td>12 22-29</td>
<td>20 8 326-581</td>
</tr>
<tr>
<td>5 0 7</td>
<td>13 30-37</td>
<td>21 9 582-1093</td>
</tr>
<tr>
<td>6 0 8</td>
<td>14 38-53</td>
<td>22 10 1094-2117</td>
</tr>
<tr>
<td>7 0 9</td>
<td>15 54-69</td>
<td>23 24 2118-16779333</td>
</tr>
</tbody>
</table>

To convert an insert-and-copy length code to an insert length code and a copy length code, the following table can be used:
First, look up the cell with the 64 value range containing the insert-and-copy length code, this gives the insert length code and the copy length code ranges, both 8 values long. The copy length code within its range is determined by the lowest 3 bits of the insert-and-copy length code, and the insert length code within its range is determined by bits 3-5 (counted from the LSB) of the insert- and-copy length code. Given the insert length and copy length codes, the actual insert and copy lengths can be obtained by reading the number of extra bits given by the tables above.

If the insert-and-copy length code is between 0 and 127, the distance code of the command is set to zero (the last distance reused).

6. Encoding of block switch commands

As described in Section 2, a block-switch command is a pair <block type, block count>. These are encoded in the compressed data part of the meta-block, right before the start of each new block of a particular block category.

Each block type in the compressed data is represented with a block type code, encoded using a prefix code over the block type code alphabet. A block type symbol 0 means that the new block type is the same as the type of the previous block from the same block category, i.e. the block type that preceded the current type, while a block type symbol 1 means that the new block type equals the current block type plus one. If the current block type is the maximal possible,
then a block type symbol of 1 results in wrapping to a new block type of 0. Block type symbols 2 - 257 represent block types 0 - 255 respectively. The previous and current block types are initialized to 1 and 0, respectively, at the end of the meta-block header.

Since the first block type of each block category is 0, the block type of the first block switch command is not encoded in the compressed data. Instead the block count for each category that has more than one type is encoded in the meta-block header.

Since the end of the meta-block is detected by the number of uncompressed bytes produced, the block counts for any of the three categories need not count down to exactly zero at the end of the meta-block.

The number of different block types in each block category, denoted by NBLTYPESL, NBLTYPESI, and NBLTYPESD for literals, insert-and-copy lengths and distances, respectively, is encoded in the meta-block header, and it must equal to the largest block type plus one in that block category. In other words, the set of literal, insert-and-copy length and distance block types must be [0..NBLTYPESL-1], [0..NBLTYPESI-1], and [0..NBLTYPESD-1], respectively. From this it follows that the alphabet size of literal, insert-and-copy length and distance block type codes is NBLTYPES + 2, NBLTYPESI + 2 and NBLTYPESD + 2, respectively.

Each block count in the compressed data is represented with a pair <block count code, extra bits>. The block count code and the extra bits are encoded back-to-back, the block count code is encoded using a prefix code over the block count code alphabet, while the extra bits value is encoded as a fixed-width integer value. The number of extra bits can be 0 - 24, and it is dependent on the block count code. The symbols of the block count code alphabet, along with the number of extra bits and the range of block counts are as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0 2 1-4</td>
<td>1 4 65-80</td>
<td>18 7 369-496</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 2 5-8</td>
<td>10 4 81-96</td>
<td>19 8 497-752</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2 9-12</td>
<td>11 4 97-112</td>
<td>20 9 753-1264</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 2 13-16</td>
<td>12 5 113-144</td>
<td>21 10 1265-2288</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 3 17-24</td>
<td>13 5 145-176</td>
<td>22 11 2289-4336</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 3 25-32</td>
<td>14 5 177-208</td>
<td>23 12 4337-8432</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 3 33-40</td>
<td>15 5 209-240</td>
<td>24 13 8433-16624</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 3 41-48</td>
<td>16 6 241-304</td>
<td>25 24 16625-16793840</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 4 49-64</td>
<td>17 6 305-368</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The first block switch command of each block category is special in the sense that it is encoded in the meta-block header, and as described earlier the block type code is omitted, since it is an implicit zero.

7. Context modeling

As described in Section 2, the prefix tree used to encode a literal byte or a distance code depends on the context ID and the block type. This section specifies how to compute the context ID for a particular literal and distance code, and how to encode the context map that maps a <context ID, block type> pair to the index of a prefix code in the array of literal and distance prefix codes.

7.1. Context modes and context ID lookup for literals

The context for encoding the next literal is defined by the last two bytes in the stream (p1, p2, where p1 is the most recent byte), regardless if these bytes are produced by backward references or by literal insertions. At the start of the stream p1 and p2 are initialized to zero.

There are four methods, called context modes, to compute the Context ID:

* MSB6, where the Context ID is the value of six most significant bits of p1,
* LSB6, where the Context ID is the value of six least significant bits of p1,
* UTF8, where the Context ID is a complex function of p1, p2, optimized for text compression, and
* Signed, where Context ID is a complex function of p1, p2, optimized for compressing sequences of signed integers.

The Context ID for the UTF8 and Signed context modes is computed using the following lookup tables Lut0, Lut1, and Lut2.

Lut0 :=

0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 0, 0, 4, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
8, 12, 16, 12, 20, 12, 16, 24, 28, 12, 12, 32, 12, 36, 12,
44, 44, 44, 44, 44, 44, 44, 44, 32, 32, 24, 40, 28, 12,
12, 48, 52, 52, 48, 52, 52, 52, 48, 52, 52, 52, 52, 52, 48,
52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52, 52,
12, 56, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44,
44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44,
44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44,
44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44,
44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44, 44,
The lengths and zlib CRC-32 (ITU-T Recommendation V.42) check values of each of these tables as a sequence of bytes are as follows:

<table>
<thead>
<tr>
<th>Table</th>
<th>Length</th>
<th>CRC-32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lut0</td>
<td>256</td>
<td>0x8e91efb7</td>
</tr>
</tbody>
</table>
Given p1 is the last uncompressed byte and p2 is the second-to-last uncompressed byte, the context IDs can be computed as follows:

- For LSB6: Context ID = p1 & 0x3f
- For MSB6: Context ID = p1 >> 2
- For UTF8: Context ID = Lut0[p1] | Lut1[p2]
- For Signed: Context ID = (Lut2[p1] << 3) | Lut2[p2]

The context modes LSB6, MSB6, UTF8, and Signed are denoted by integers 0, 1, 2, 3.

The context mode is defined for each literal block type and they are stored in a consecutive array of bits in the meta-block header, always two bits per block type.

### 7.2. Context ID for distances

The context for encoding a distance code is defined by the copy length corresponding to the distance. The context IDs are 0, 1, 2, and 3 for copy lengths 2, 3, 4, and more than 4, respectively.

### 7.3. Encoding of the context map

There are two context maps, one for literals and one for distances. The size of the context map is 64 * NBLTYPESL for literals, and 4 * NBLTYPESD for distances. Each value in the context map is an integer between 0 and 255, indicating the index of the prefix code to be used when encoding the next literal or distance.

The context map is encoded as a one-dimensional array, CMAPL[0..(64 * NBLTYPESL - 1)] and CMAPD[0..(4 * NBLTYPESD - 1)].

The index of the prefix code for encoding a literal or distance code with context ID "cid" and block type "bltype" is

- index of literal prefix code = CMAPL[bltype * 64 + cid]
- index of distance prefix code = CMAPD[bltype * 4 + cid]

The values of the context map are encoded with the combination of run length encoding for zero values and prefix coding. Let RLEMAX denote the number of run length codes and NTREES denote the maximum value in the context map plus one. NTREES must equal the number of different values in the context map, in other words, the different values in the context map must be the [0..NTREES-1] interval. The alphabet of...
the prefix code has the following RLEMAX + NTREES symbols:

0: value zero
1: repeat a zero 2 to 3 times, read 1 bit for repeat length
2: repeat a zero 4 to 7 times, read 2 bits for repeat length
...
RLEMAX: repeat a zero (1 << RLEMAX) to (1 << (RLEMAX+1))-1 times, read RLEMAX bits for repeat length
RLEMAX + 1: value 1
...
RLEMAX + NTREES - 1: value NTREES - 1

If RLEMAX = 0, the run length coding is not used, and the symbols of the alphabet are directly the values in the context map. We can now define the format of the context map (the same format is used for literal and distance context maps):

1-5 bits: RLEMAX, 0 is encoded with one 0 bit, and values 1 - 16 are encoded with bit pattern xxxx1 (so 01001 is 5)

Prefix code with alphabet size NTREES + RLEMAX
Context map size values encoded with the above prefix code and run length coding for zero values
1 bit: IMTF bit, if set, we do an inverse move-to-front transform on the values in the context map to get the prefix code indexes

For the encoding of NTREES see Section 9.2. We define the inverse move-to-front transform used in this specification by the following C language function:

```c
void InverseMoveToFrontTransform(uint8_t* v, int v_len) {
    uint8_t mtf[256];
    int i;
    for (i = 0; i < 256; ++i) {
        mtf[i] = (uint8_t)i;
    }
    for (i = 0; i < v_len; ++i) {
        uint8_t index = v[i];
        uint8_t value = mtf[index];
        v[i] = value;
        for (; index; --index) {
            mtf[index] = mtf[index - 1];
        }
        mtf[0] = value;
    }
}
```
8. Static dictionary

At any given point during decoding the compressed data, a reference to a duplicated string in the uncompressed data produced so far has a maximum backward distance value, which is the minimum of the window size and the number of uncompressed bytes produced. However, decoding a distance from the compressed stream, as described in section 4, can produce distances that are greater than this maximum allowed value. The difference between these distances and the first invalid distance value is treated as reference to a word in the static dictionary given in Appendix A. The maximum valid copy length for a static dictionary reference is 24. The static dictionary has three parts:

* DICT[0..DICTSIZE], an array of bytes
* DOFFSET[0..24], an array of byte offset values for each length
* NDBITS[0..24], an array of bit-depth values for each length

The number of static dictionary words for a given length is:

\[
\text{NWORDS[length]} = \begin{cases} 
0 & \text{(if length < 4)} \\
(1 << \text{NDBITS[length]}) & \text{(if length >= 4)}
\end{cases}
\]

DOFFSET and DICTSIZE are defined by the following recursion:

\[
\begin{align*}
\text{DOFFSET}[0] & = 0 \\
\text{DOFFSET}[\text{length} + 1] & = \text{DOFFSET}[\text{length}] + \text{length} \times \text{NWORDS[length]} \\
\text{DICTSIZE} & = \text{DOFFSET}[24] + 24 \times \text{NWORDS[24]}
\end{align*}
\]

The offset of a word within the DICT array for a given length and index is:

\[
\text{offset(length, index)} = \text{DOFFSET[length]} + \text{index} \times \text{length}
\]

Each static dictionary word has 121 different forms, given by applying a word transformation to a base word in the DICT array. The list of word transformations is given in Appendix B. The static dictionary word for a <length, distance> pair can be reconstructed as follows:

\[
\begin{align*}
\text{word_id} & = \text{distance} - (\text{max allowed distance} + 1) \\
\text{index} & = \text{word_id} \% \text{NWORDS[length]} \\
\text{base_word} & = \text{DICT[offset(length, index)..<offset(length, index+1)-1]} \\
\text{transform_id} & = \text{word_id} >> \text{NDBITS[length]}
\end{align*}
\]

The string copied to the uncompressed stream is computed by applying
the transformation to the base dictionary word. If transform_id is greater than 120 or length is greater than 24, the compressed data set is invalid.

Each word transformation has the following form:

\[
\text{transform}_i(\text{word}) = \text{prefix}_i + T_i(\text{word}) + \text{suffix}_i
\]

where the \(i\) subscript denotes the transform_id above. Each \(T_i\) is one of the following 21 elementary transforms:

- Identity, OmitLast1, ..., OmitLast9, UppercaseFirst, UppercaseAll, OmitFirst1, ..., OmitFirst9

The form of these elementary transforms are as follows:

\[
\text{Identity}(\text{word}) = \text{word}
\]

\[
\text{OmitLast}k(\text{word}) = \text{the first (length(\text{word}) - k) bytes of word, or empty string if length(\text{word}) < k}
\]

\[
\text{UppercaseFirst}(\text{word}) = \text{first UTF-8 character of word upper-cased}
\]

\[
\text{UppercaseAll}(\text{word}) = \text{all UTF-8 characters of word upper-cased}
\]

\[
\text{OmitFirst}k(\text{word}) = \text{the last (length(\text{word}) - k) bytes of word, or empty string if length(\text{word}) < k}
\]

For the purposes of UppercaseAll, word is parsed into UTF-8 characters and converted to upper-case by taking 1 - 3 bytes at a time, using the algorithm below:

\[
i = 0
\]

\[
\text{while } i < \text{length(\text{word})}:
\]

\[
\begin{align*}
\text{if } & \text{word}[i] < 192: \\
& \text{if word}[i] \geq 97 \text{ and word}[i] \leq 122: \\
& \quad \text{word}[i] = \text{word}[i] \oplus 32 \\
& \quad i = i + 1 \\
\text{else if } & \text{word}[i] < 224: \\
& \quad \text{if } i + 1 < \text{length(\text{word})}: \\
& \quad \quad \text{word}[i + 1] = \text{word}[i + 1] \oplus 32 \\
& \quad \quad i = i + 2 \\
\text{else:} \\
& \quad \text{i + 2 < length(\text{word})}: \\
& \quad \quad \text{word}[i + 2] = \text{word}[i + 2] \oplus 5 \\
& \quad \quad i = i + 3
\end{align*}
\]

For UppercaseFirst, the same algorithm is used, but the loop is
executed only once.

Appendix B. contains the list of transformations by specifying the prefix, elementary transform and suffix components of each of them. Note that the OmitFirst8 elementary transform is not used in the list of transformations. The strings in Appendix B. are in C string format with respect to escape (backslash) characters.

9. Compressed data format

In this section we describe the format of the compressed data set in terms of the format of the individual data items described in the previous sections.

9.1. Format of the stream header

The stream header has only the following one field:

1-7 bits: WBITS, a value in the range 10 - 24, encoded with the following variable length code (as it appears in the compressed data, where the bits are parsed from right to left):

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0100001</td>
</tr>
<tr>
<td>11</td>
<td>0110001</td>
</tr>
<tr>
<td>12</td>
<td>1000001</td>
</tr>
<tr>
<td>13</td>
<td>1010001</td>
</tr>
<tr>
<td>14</td>
<td>1100001</td>
</tr>
<tr>
<td>15</td>
<td>1110001</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>0000001</td>
</tr>
<tr>
<td>18</td>
<td>0011</td>
</tr>
<tr>
<td>19</td>
<td>0101</td>
</tr>
<tr>
<td>20</td>
<td>0111</td>
</tr>
<tr>
<td>21</td>
<td>1001</td>
</tr>
<tr>
<td>22</td>
<td>1011</td>
</tr>
<tr>
<td>23</td>
<td>1101</td>
</tr>
<tr>
<td>24</td>
<td>1111</td>
</tr>
</tbody>
</table>

Note that bit pattern 0010001 is invalid and must not be used.

The size of the sliding window, which is the maximum value of any non-dictionary reference backward distance, is given by the following formula:
9.2. Format of the meta-block header

A compliant compressed data set has at least one meta-block. Each meta-block contains a header with information about the uncompressed length of the meta-block, and a bit signaling if the meta-block is the last one. The format of the meta-block header is the following:

1 bit: IS LAST, set to 1 if this is the last meta-block
1 bit: IS LAST EMPTY, set to 1 if the last meta-block is empty, this field is only present if IS LAST bit is set -- if it is 1, then the meta-block and the brotli stream ends at that bit, with any remaining bits in the last byte of the compressed stream filled with zeros (if the fill bits are not zero, then the stream should be rejected as invalid)
2 bits: MNIBBLES, # of nibbles to represent the uncompressed length, encoded as follows: if set to 3, MNIBBLES is 0, otherwise MNIBBLES is the value of this field plus 4. If MNIBBLES is 0, the meta-block is empty, i.e. it does not generate any uncompressed data. In this case, the rest of the meta-block has the following format:

1 bit: reserved, must be zero
2 bits: MSKIPBYTES, # of bytes to represent metadata length

MSKIPBYTES x 8 bits: MSKIPLEN - 1, where MSKIPLEN is the number of metadata bytes; this field is only present if MSKIPBYTES is positive, otherwise MSKIPLEN is 0 (if MSKIPBYTES is greater than 1, and the last byte is all zeros, then the stream should be rejected as invalid)
0 - 7 bits: fill bits until the next byte boundary, must be all zeros
MSKIPLEN bytes of metadata, not part of the uncompressed data or the sliding window

MNIBBLES x 4 bits: MLEN - 1, where MLEN is the length of the meta-block uncompressed data in bytes (if the number of nibbles is greater than 4, and the last nibble is all zeros, then the stream should be rejected as invalid)
1 bit: ISUNCOMPRESSED, if set to 1, any bits of compressed data up to the next byte boundary are ignored, and the rest of the meta-block contains MLEN bytes of literal data; this field is only present if the ISLAST bit is not set (if the ignored bits are not all zeros, the stream should be rejected as invalid)

1-11 bits: NBLTYPESL, # of literal block types, encoded with the following variable length code (as it appears in the compressed data, where the bits are parsed from right to left, so 0110111 has the value 12):

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0001</td>
</tr>
<tr>
<td>3-4</td>
<td>x0011</td>
</tr>
<tr>
<td>5-8</td>
<td>xx0101</td>
</tr>
<tr>
<td>9-16</td>
<td>xxx0111</td>
</tr>
<tr>
<td>17-32</td>
<td>xxx1001</td>
</tr>
<tr>
<td>33-64</td>
<td>xxx1011</td>
</tr>
<tr>
<td>65-128</td>
<td>xxx1101</td>
</tr>
<tr>
<td>129-256</td>
<td>xxx1111</td>
</tr>
</tbody>
</table>

Prefix code over the block type code alphabet for literal block types, appears only if NBLTYPESL >= 2

Prefix code over the block count code alphabet for literal block counts, appears only if NBLTYPESL >= 2

Block count code + Extra bits for first literal block count, appears only if NBLTYPESL >= 2

1-11 bits: NBLTYPESI, # of insert-and-copy block types, encoded with the same variable length code as above

Prefix code over the block type code alphabet for insert-and-copy block types, only if NBLTYPESI >= 2

Prefix code over the block count code alphabet for insert-and-copy block counts, only if NBLTYPESI >= 2

Block count code + Extra bits for first insert-and-copy block count, only if NBLTYPESI >= 2

1-11 bits: NBLTYPESD, # of distance block types, encoded with the same variable length code as above
Prefix code over the block type code alphabet for distance block types, appears only if NBLTYPESD >= 2

Prefix code over the block count code alphabet for distance block counts, only if NBLTYPESD >= 2

Block count code + Extra bits for first distance block count, only if NBLTYPESD >= 2

2 bits: NPOSTFIX, parameter used in the distance coding

4 bits: four most significant bits of NDIRECT, to get the actual value of the parameter NDIRECT, left-shift this four bit number by NPOSTFIX bits

NBLTYPESL x 2 bits: context mode for each literal block type

1-11 bits: NTREESL, # of literal prefix trees, encoded with the same variable length code as NBLTYPESL

Literal context map, encoded as described in Paragraph 7.3, appears only if NTREESL >= 2, otherwise the context map has only zero values

1-11 bits: NTREESD, # of distance prefix trees, encoded with the same variable length code as NBLTYPESD

Distance context map, encoded as described in Paragraph 7.3, appears only if NTREESD >= 2, otherwise the context map has only zero values

NTREESL prefix codes for literals

NBLTYPESI prefix codes for insert-and-copy lengths

NTREESD prefix codes for distances

9.3. Format of the meta-block data

The compressed data part of a meta-block consists of a series of commands. Each command has the following format:

Block type code for next insert-and-copy block type, appears only if NBLTYPESI >= 2 and the previous insert-and-copy block count is zero

Block count code + Extra bits for next insert-and-copy block count, appears only if NBLTYPESI >= 2 and the
previous insert-and-copy block count is zero

Insert-and-copy length, encoded as in section 5, using the insert-and-copy length prefix code with the current insert-and-copy block type index

Insert length number of literals, with the following format:

Block type code for next literal block type, appears only if NBLTYPESL >= 2 and the previous literal block count is zero

Block count code + Extra bits for next literal block count, appears only if NBLTYPESL >= 2 and the previous literal block count is zero

Next byte of the uncompressed data, encoded with the literal prefix code with the index determined by the previous two bytes of the uncompressed data, the current literal block type, and the context map, as described in Paragraph 7.3.

Block type code for next distance block type, appears only if NBLTYPESD >= 2 and the previous distance block count is zero

Block count code + Extra bits for next distance block length, appears only if NBLTYPESD >= 2 and the previous distance block count is zero

Distance code, encoded as in section 4, using the distance prefix code with the current distance block type index, appears only if the distance code is not an implicit 0, as indicated by the insert-and-copy length code

The number of commands in the meta-block is such that the sum of insert lengths and copy lengths over all the commands gives the uncompressed length, MLEN encoded in the meta-block header.

10. Decoding algorithm

The decoding algorithm that produces the uncompressed data is as follows:

```plaintext
read window size
do
  read ISLAST bit
  if ISLAST
```

Alakuijala & Szabadka Expires November 11, 2015 [Page 31]
read ISLASTEMPTY bit
if ISLASTEMPTY
    break from loop
read MNIBBLES
if MNIBBLES is zero
    verify reserved bit is zero
read MSKIPLLEN
    skip any bits up to the next byte boundary
    skip MSKIPLLEN bytes
    continue to the next meta-block
else
    read MLEN
if not ISLAST
    read ISUNCOMPRESSED bit
if ISUNCOMPRESSED
    skip any bits up to the next byte boundary
    copy MLEN bytes of compressed data as literals
    continue to the next meta-block
loop for each three block categories (i = L, I, D)
read NBLTYPESi
if NBLTYPESi >= 2
    read prefix code for block types, HTREE_BTYPE_i
    read prefix code for block counts, HTREE_BLEN_i
    read block count, BLEN_i
    set block type, BTYPE_i to 0
else
    set block type, BTYPE_i to 0
    set block count, BLEN_i to 268435456
read NPOSTFIX and NDIRECT
read array of literal context modes, CMODE[]
read NTREESL
if NTREESL >= 2
    read literal context map, CMAPL[]
else
    fill CMAPL[] with zeros
read NTREESD
if NTREESD >= 2
    read distance context map, CMAPD[]
else
    fill CMAPD[] with zeros
read array of prefix codes for literals, HTREEL[]
read array of prefix codes for insert-and-copy, HTREEI[]
read array of prefix codes for distances, HTREED[]
do
    if BLEN_I is zero
        read block type using HTREE_BTYPE_I and set BTYPE_I
        save previous block type
read block count using HTREE_BLEN_I and set BLEN_I
decrement BLEN_I
read insert and copy length, ILEN, CLEN with HTREEI[BTYPE_I]
loop for ILEN
  if BLEN_L is zero
    read block type using HTREE_BTYPE_L and set BTYPE_L
    save previous block type
    read block count using HTREE_BLEN_L and set BLEN_L
    decrement BLEN_L
    look up context mode CMODE[BTYPE_L]
    compute context ID, CIDL from last two uncompressed bytes
    read literal using HTREEI[CMAPL[64 * BTYPE_L + CIDL]]
    write literal to uncompressed stream
  if number of uncompressed bytes produced in the loop for
  this meta-block is MLEN, then break from loop (in this
  case the copy length is ignored and can have any value)
  if distance code is implicit zero from insert-and-copy code
    set backward distance to the last distance
  else
    if BLEN_D is zero
      read block type using HTREE_BTYPE_D and set BTYPE_D
      save previous block type
      read block count using HTREE_BLEN_D and set BLEN_D
      decrement BLEN_D
      compute context ID, CIDD from CLEN
      read distance code with HTREEI[CMAPD[4 * BTYPE_D + CIDD]]
      compute distance by distance short code substitution
      move backwards distance bytes in the uncompressed data and
      copy CLEN bytes from this position to the uncompressed
      stream, or look up the static dictionary word, transform
      the word as directed, and copy the result to the
      uncompressed stream
      while number of uncompressed bytes for this meta-block < MLEN
        while not ISLAST
          Note that a duplicated string reference may refer to a string in a
          previous meta-block, i.e. the backward distance may cross one or more
          meta-block boundaries. However a backward copy distance will not
          refer past the beginning of the uncompressed stream or the window
          size; any such distance is interpreted as a reference to a static
          dictionary word. Also note 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 uncompressed stream.

11. Security Considerations

As with any compressed file formats, decompressor implementations
should handle all compressed data byte sequences, not only those that
conform to this specification, where non-conformant compressed data
sequences should be discarded. A possible attack against a system
containing a decompressor implementation (e.g. a web browser) is to
exploit a buffer overflow caused by an invalid compressed data.
Therefore decompressor implementations should perform bound-checking
for each memory access that result from values decoded from the
compressed stream.

12. IANA Considerations

This document has no actions for IANA.

13. Informative References

Redundancy Codes", Proceedings of the Institute of Radio
Engineers, September 1952, Volume 40, Number 9, pp.
1098-1101.

[LZ77] Ziv J., Lempel A., "A Universal Algorithm for Sequential
Data Compression", IEEE Transactions on Information

http://www.ietf.org/rfc/rfc1951.txt

2.0", W3C WebFonts Working Group,
http://www.w3.org/TR/WOFF2/

14. Source code

Source code for a C language implementation of a brotli compliant
decompressor and a C++ language implementation of a compressor is
available in the brotli open-source project:
https://github.com/google/brotli

15. Acknowledgements

The authors would like to thank Mark Adler for providing helpful
review comments, validating the specification by writing an
independent decompressor and suggesting improvements to the format
and the text of the specification.

Appendix A. Static dictionary data
The hexadecimal form of the DICT array is the following, where the length is 122,784 bytes and the zlib CRC-32 of the byte sequence is 0x5136cb04.
a49ae6b395e8a784e5aeb6e5b185e4b9a6e5ba7e8bf9ee68a5e7ab8be58db3
e4b88ee6a5e68a5e80e5b7a7e5a5a5e8bf90e799bbee585a5e4bba5e69da5e790
86e8aeae4b0a8b4e6bb6e887ae794b1e4b8ade58d8ee58a9ee585ace5a688e5
a688e79c9f6ada3e4b88de6999e9e585a8e6698e7e59088e5908ce4b87e580bce
5e88abeb4babae79b9f7e9da3e585b7e4bd93e4b896e7b9aa5b2a2e9998fe588
9be4b99ae689bfef688be5e2a99e9998be6c98e4baba4ebf9d6e8c1e5958e5eb
aebe67babe4fbae5e8bf0e6b9be5b7a6e58f3e882e4ebe4ede7a9d4e6a188
ae3e9e9998e794b5e4b8ae7bab8e79086e794fe591be5d5ae3e4bca0e4b
bbe58a1e6ada3e5c8cfe789b9e8888e4b886e9da5e58d8fe4ec9aae58faaee8
83bde5b9d7e84b6e5089a5a7e5aeb9e68c7e5a9be8c8f90e8a18c
b697a5e5b9f7e8b3a3e5aeb6e86b5e8bf87e59c9fe5cb0e6b59e61b9fe69
afe4b988e6a8e8e5b9bfef8ebeb0e59cbea68f8e8ff0e5e85f9h5e8c96
d4bca0e7b9f6e6ad8e6c8e6b4f9de999ae8afb7e7a88be5c8bce79697e7bb
8ef8bf87e8f8e7e5e6ee4b988e5b88de6984e5b98e8a87e5b986e69a65e5b94e5ba6e69d82e5
bf97e7be8e4b88ded69c80e9a9b8e799bbee999e69a065a5e58a0e5b7a5a
5e85f8de8b4a4e69996e7a888e8b98e59d7e8baa4e4b93e69d7e5ba6e68587
ba5e94aee6890e69cace5bda2e5b8e5f9c9e8b18e587ae5b39e489c6e
96b59e88eae7a9eb5d9e4baeace6b1e8818e5cace79819e5d8f6e97e8818e488d8
ed79bbbeb4fae19a1b5e959da2e5886e9927e9d1e9a1b5e7a1ee5e9ae59b
beee4b98e6bd9e59d80e7a7a7afe69e8e9499ae8afafe79bbae79a4e5ae9e8de
b4969c8a5e585b3e9a39e69e8e69d3e79785e6af92e5aae0e789a9
e999a4e4ba86e8a995e8ab96e796bee79785e5f8ae697b66b182e8b4ade7a7b
98e782b9e584b7e6949be58f9e7e7cace599b9e6bf39e9878fe694bee58a0b1e
4e889e8e4b9a9e46a4e5a5d97e4b83e5886e96969e68584e89991e6980a7e688b2e5b7a5a
94e5b9d3e5be8e5b888e696b9e4bebe6a0a1e5b9ade8821e5b82e688be5
b18e6a08f7e9babe59198e5b7a5e4fabe8737e8a784e9871993e585b7
6e97cace7bd9e7b93e59088e6a1a3e6a188e58ab3e58aa8e5f8a4e5f9e7be
8ee5853e5bce58b5be7e6949be58f9e7e7cace599b9e6bf39e9878fe694bee58a0b1e
4e889e8e4b9a9e46a4e5a5d97e4b83e5886e96969e68584e89991e6980a7e688b2e5b7a5a
94e5b9d3e5be8e5b888e696b9e4bebe6a0a1e5b9ade8821e5b82e688be5
b18e6a08f7e9babe59198e5b7a5e4fabe8737e8a784e9871993e585b7
6e97cace7bd9e7b93e59088e6a1a3e6a188e58ab3e58aa8e5f8a4e5f9e7be
8ee5853e5bce58b5be7e6949be58f9e7e7cace599b9e6bf39e9878fe694bee58a0b1e
4e889e8e4b9a9e46a4e5a5d97e4b83e5886e96969e68584e89991e6980a7e688b2e5b7a5a
94e5b9d3e5be8e5b888e696b9e4bebe6a0a1e5b9ade8821e5b82e688be5
b18e6a08f7e9babe59198e5b7a5e4fabe8737e8a784e9871993e585b7
6e97cace7bd9e7b93e59088e6a1a3e6a188e58ab3e58aa8e5f8a4e5f9e7be

73696e6720416c736f2c20b203725420374797065203d20a0a0a0c21222d2d4e7761726f7335
5636f726475702769617465466f726569676e5072656d69676572636869636f
657356672747516c72657475726e73436f6d656e74506f7765726564696e
6c696563b70f7665727479636861d6265724c6967669e672066f6c756d65
d64696e6720677261676974796c696669656e61696e696e6762696769
65636536737547469697677261676974796c69666562096e438617046572
2d73686164f6f774e6f74634e60da0207265745726e37346164
697567d7646754737367127958e6777427617656c7368656e4206279779
686f2016275777f726b2069e66613765c7479616e67756c617277686f2068
6166146172670f727476446e60e0a0a5366656250207636c6966366f3b7663
6172675736b657977f672646972077696c6563697479206f6287486697329
3b416e647265720756e697175620636865666f6f7220d6f7265533030
708382b0274574752e3b7273696f6e3d232706c7567696e7377697468669e2e
6865736356c6537461746966e64656665627616c76656e747572657075626c
9736875656e77420746756e73696f6163617462576373636f6d5207464e66
9696e76752734475965206670656f706c6562657867606c6f747768617420
976861762d66f7e91206d16a6f7222a226874674906e20686972306d65
6e755223e0a6df6e74686c796f6666696572636f756e63696c676561666e
676575656e2069e535766d61727946174502066f6c79616c6774966774
5e6537373616e420276173586d70657265706c6d5355636f66626420
68656172696752753736961e6c6f6661677341626527416c617465
72616c73654026620736d616c6c222e2e617070656e4464207769746866
65646572616c6261662b2066256e6561746844557307697454361706974
616c677267556e473292c2061e4e2070657263656e746974620762666d63c
6f73696e6736e6f7461696e49e373465616466696746556e617320767562
c6e2e7961686f6e726573706f6e646966786754626f723633755725726566
6c6553746f72676166933d20d61746862e5646974969e6762e6c696562
70615460696e675212077686f6c656f7466617269706976656e420
6f66206261727669677768656e3206974968656164657220686f6d520666f72
6573576d657265726616de6547374726f6e673e68656174696e67725746169
6e73636c6f7564677761729206f62064d20162630316be677f696e6796e2e
2070617274426574776566e6c6573736f6e736366c6f73657347697247516f
6c6965636773232e637263737465445e44202d2d3e66616d6f756320617761
72646564c6963656e73654865616c746820666169726c79207765616c745879
6d9e669d616c4167162696361e63f6e706574656c616265222e73396e67
96e6766172626572734276173696c2964697363753733723276506c6136534f7
2e65766f729766e6f7420636f77723565646170656162736d616b620
750706f7256e6456662f746820666c6f7365473671772074686566
66696365736366f6f75737662866c735776656e206865666e666f7263
57607567386967756177576774205554462d3922e3a461e67e46173796e2e0
6d6f7374696ea75762564557375616c67966172696766636c6f73757526
6f626a65374206465666e6e63575535206f6204d564969361c3c626f64
793e0a656766656e4e746255753564656794364657369784765566e49
736c61669632303030303030656e7469726520776964656c79206163746976
652087497970656666e65206361e63f6c6f7f203d737065616b65725678
74566e473506879736967357467572616e3c74626f64793e6756e657261

Alakuijala & Szabadska Expires November 11, 2015 [Page 53]
The number of words for each length is given by the following bit-depth array:

\[
\text{NDBITS} := 0, 0, 0, 0, 10, 10, 11, 11, 10, 10,
\]
\[
10, 10, 9, 9, 8, 7, 7, 8, 7,
\]
\[
7, 6, 6, 5, 5
\]

**Appendix B. List of word transformations**

The string literals are in C format, with respect to the use of backslash escape characters.

In order to generate a length and check value, the transforms can be converted to a series of bytes, where each transform is the prefix sequence of bytes plus a terminating zero byte, a single byte value identifying the transform, and the suffix sequence of bytes plus a terminating zero. The value for the transforms are 0 for Identity, 1 for UppercaseFirst, 2 for UppercaseAll, 3 to 11 for OmitFirst1 to OmitFirst9, and 12 to 20 for OmitLast1 to OmitLast9. The byte sequences that represent the 121 transforms are then concatenated to a single sequence of bytes. The length of that sequence is 657 bytes, and the zlib CRC is 0x00f1fd60.

<table>
<thead>
<tr>
<th>ID</th>
<th>Prefix</th>
<th>Transform</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>1</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>2</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>3</td>
<td>&quot;&quot;</td>
<td>OmitFirst1</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>4</td>
<td>&quot;&quot;</td>
<td>UppercaseFirst</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>5</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; the &quot;</td>
</tr>
<tr>
<td>6</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>7</td>
<td>&quot;s&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>8</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; of &quot;</td>
</tr>
<tr>
<td>9</td>
<td>&quot;&quot;</td>
<td>UppercaseFirst</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>10</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; and &quot;</td>
</tr>
<tr>
<td>11</td>
<td>&quot;&quot;</td>
<td>OmitFirst2</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>12</td>
<td>&quot;&quot;</td>
<td>OmitLast1</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>13</td>
<td>&quot;,&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>14</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;, &quot;</td>
</tr>
<tr>
<td>15</td>
<td>&quot;&quot;</td>
<td>UppercaseFirst</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>16</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; in &quot;</td>
</tr>
<tr>
<td>17</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; to &quot;</td>
</tr>
<tr>
<td>18</td>
<td>&quot;e&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>19</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>20</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;.&quot;</td>
</tr>
</tbody>
</table>
69 " " UppercaseFirst "\">" 
70 " " Identity "\=" 
71 " " Identity "" 
72 ".com/" Identity "" 
73 " the " Identity " of the " 
74 " " UppercaseFirst "" 
75 " " Identity ". This " 
76 " " Identity "" 
77 ". " Identity "" 
78 " " UppercaseFirst "(" 
79 " " UppercaseFirst "" 
80 " " Identity " not " 
81 " " Identity "\=" 
82 " " Identity "er " 
83 " " UppercaseAll " " 
84 " " Identity "al " 
85 " " UppercaseAll " " 
86 " " Identity "=" 
87 " " UppercaseAll "\" 
88 " " UppercaseFirst "" 
89 " " Identity "{" 
90 " " Identity "ful " 
91 " " UppercaseFirst " " 
92 " " Identity "ive " 
93 " " Identity "less " 
94 " " UppercaseAll " " 
95 " " Identity "est " 
96 " " UppercaseFirst "" 
97 " " UppercaseAll "\">" 
98 " " Identity "=" 
99 " " UppercaseFirst "" 
100 " " Identity "ize " 
101 " " UppercaseAll "" 
102 "\xc2\xa0" Identity " " 
103 " " Identity " " 
104 " " UppercaseFirst "\=" 
105 " " UppercaseAll "\=" 
106 " " Identity "ous " 
107 " " UppercaseAll " " 
108 " " UppercaseFirst "=" 
109 " " UppercaseFirst " " 
110 " " UppercaseAll "\=" 
111 " " UppercaseAll " " 
112 " " UppercaseAll " " 
113 " " UppercaseAll "(" 
114 " " UppercaseAll " " 
115 " " UppercaseAll " " 
116 " " UppercaseAll "="
Authors’ Addresses

Jyrki Alakuijala
Google, Inc

Email: jyrki@google.com

Zoltan Szabadka
Google, Inc

Email: szabadka@google.com