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 16, 2014.

Copyright Notice

Copyright (c) 2014 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.
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 "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:

```
+--------+
|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:

```
+--------+
|00001000|00000010|
+--------+
```

+ 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 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).

2. Compressed representation overview

A compressed data set consists of a header and a series of meta-blocks corresponding to successive meta-blocks of input data. The meta-block sizes are limited to bytes and the maximum meta-block size is 268,435,456 bytes.

The header contains the size of a sliding window on the input data that is sufficient to keep on the intermediate storage at any given point during decoding the stream.

Each meta-block is compressed using a combination of the LZ77 algorithm (Lempel-Ziv 1977, [LZ77]) and Huffman coding. The Huffman trees for each block are independent of those for previous or subsequent blocks; the LZ77 algorithm may use a reference to a duplicated string occurring in a previous meta-block, up to sliding window size input bytes before.

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>. 
Each command in the compressed data is represented using three kinds of Huffman codes: one kind of code tree 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 kind of code tree for literals, and a third kind of code tree for distances. The code trees for each meta-block appear in a compact form just before the compressed data in the meta-block header.

The sequence of each type of value in the representation of a command (insert-and-copy lengths, literals and distances) within a meta-block is further divided into blocks. In the "brotli" format, blocks are not contiguous chunks of compressed data, but rather the pieces of compressed data belonging to a block are interleaved with pieces of data belonging to other blocks. Each meta-block can be logically decomposed into a series of insert-and-copy length blocks, a series of literal blocks and a series of distance blocks. These are also called the three block categories: a meta-block has a series of blocks for each block category. Note that the physical structure of the meta-block is a series of commands, while the three series of blocks is the logical structure. Consider the following example:

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

The meta-block here has 4 commands, and each three types of symbols within these commands can be rearranged for example into the following logical block structure:

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

The subsequent blocks within each block category must have different block types, but blocks further away in the block sequence can have the same types. The block types are numbered from 0 to the maximum block type number of 255 and the first block of each block category must have 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 length>. The block-switch commands are represented in the compressed data before the start of each new block using a Huffman code tree for block types and a separate Huffman code tree for block lengths for each block category. In the above example the physical layout of the meta-block is the following:

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

Note that the block switch commands for the first blocks are not part of the meta-block compressed data part, they are encoded in the meta-block header. The code trees for block types and lengths (total of six Huffman code trees) appear in a compact form in the meta-block header.

Each type of value (insert-and-copy lengths, literals and distances) can be encoded with any Huffman tree from a collection of Huffman trees of the same kind appearing in the meta-block header. The particular Huffman tree 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 input 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 Huffman tree depends only on the block type (in fact, the index of the Huffman tree is the block type number). 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 Huffman tree indices for each block type and context ID, called the context map, is encoded in a compact form in the meta-block header.

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

3. Compressed representation of Huffman codes

3.1. Introduction to 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:
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 [HUFFMAN] in Chapter 5, references for additional information on Huffman codes.)

Note that in the "brotli" 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. Use of Huffman coding in the "brotli" format

The Huffman codes used for each alphabet in the "brotli" format are canonical Huffman 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 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:

```c
    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.

```c
    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:
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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Length</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>010</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>011</td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>110</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>G</td>
<td>4</td>
<td>1110</td>
</tr>
<tr>
<td>H</td>
<td>4</td>
<td>1111</td>
</tr>
</tbody>
</table>

3.3. Alphabet sizes

Huffman 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 length codes, the alphabet size is 26. For distance codes, block type codes and the Huffman codes used in compressing the context map, the alphabet size is dynamic and is based on other parameters.

3.4. Simple Huffman codes

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

A simple Huffman code can have only up to four symbols with non-zero code length. The format of the simple Huffman code is as follows:
2 bits: value of 1 indicates a simple Huffman code
2 bits: NSYM - 1, where NSYM = # of symbols with non-zero code length

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 Huffman 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 machine integer representing the symbol modulo the alphabet size of the Huffman code.

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

* if NSYM = 1, the code length for the one symbol is one at this stage, but only to distinguish it from the other zero code length symbols, when encoding this symbol in the compressed data stream using this Huffman code later, no actual bits are emitted. Similarly, when decoding a symbol using this Huffman 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 Huffman 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 Huffman codes

A complex Huffman code is a canonical Huffman 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 Huffman 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:

\[ \text{repeat count} = (4 \times (\text{repeat count} - 2)) + (3 - 6 \text{ on the next 2 bits}) \]

Example: Codes 7, 16 (+2 bits 11), 16 (+2 bits 10) will expand to 22 code lengths of 7:

\[ (1 + 4 \times (6 - 2) + 5) \]

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:

\[ \text{repeat count} = (8 \times (\text{repeat count} - 2)) + (3 - 10 \text{ on the next 3 bits}) \]

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 Huffman code construction algorithm given earlier. A complex Huffman code must have at least two non-zero code lengths.

The bit lengths of the Huffman code over the code length alphabet are compressed with the following static Huffman code:

<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>1111</td>
</tr>
</tbody>
</table>

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

2 bits: HSKIP, values of 0, 2 or 3 represent the respective number of leading zeros. (Value of 1 indicates the Simple Huffman 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

The code lengths of code length symbols are between 0 and 5 and they are represented with 2 - 5 bits according to the static Huffman 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 \((32 >> \text{code length})\) over all the non-zero code lengths must equal to 32.

Sequence of code lengths symbols, encoded using the code length Huffman 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 \((32768 >> \text{code length})\) 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 <distance code, extra bits>. The distance code and the extra bits are encoded back-to-back, the distance code is encoded using a Huffman code over the distance code alphabet, while the extra bits value is encoded as a fixed-width machine integer. 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, and the number of direct distance codes, denoted by NDIRECT. Both of these parameters are encoded in the meta-block header. We will also use the following derived parameter:

\[
\text{POSTFIX}\_\text{MASK} = ((1 << \text{NPOSTFIX}) - 1)
\]

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

0: last distance  
1: second last distance  
2: third last distance  
3: fourth last distance  
4: last distance - 1  
5: last distance + 1
The ring-buffer of four last distances is initialized by the values 16, 15, 11 and 4 (i.e. the fourth last is set to 16, the third last to 15, the second 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 code 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) last distance" means the (second, third, fourth) last distance that was not represented by a 0 distance code. Similarly, distances that represent static dictionary words (see Section 8.) are not pushed to the ringbuffer of last distances.

The next NDIRECT distance codes, from 16 to 15 + NDIRECT, represent distances from 1 to NDIRECT. Neither the distance short codes, nor the NDIRECT direct distance codes have any extra bits.

Distance codes 16 + NDIRECT and greater all have extra bits, the number of extra bits for a distance code "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 code alphabet is \((16 + \text{NDIRECT} + (48 \ll \text{NPOSTFIX}))\).

Given a distance code "dcode" (>= 16 + 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 Huffman code. This section provides the details to this encoding.

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

<insert-and-copy length code, insert extra bits, 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 Huffman code over the insert-and-copy length code alphabet, while the extra bits values are encoded as fixed-width machine integers. 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 code 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>---- ---- --------</td>
<td>---- ---- --------</td>
<td>---- ---- --------</td>
</tr>
<tr>
<td>0   0   0</td>
<td>8   2   10-13</td>
<td>16  6   130-193</td>
</tr>
<tr>
<td>1   0   1</td>
<td>9   2   14-17</td>
<td>17  7   194-321</td>
</tr>
<tr>
<td>2   0   2</td>
<td>10  3   18-25</td>
<td>18  8   322-577</td>
</tr>
<tr>
<td>3   0   3</td>
<td>11  3   26-33</td>
<td>19  9   578-1089</td>
</tr>
<tr>
<td>4   0   4</td>
<td>12  4   34-49</td>
<td>20 10  1090-2113</td>
</tr>
<tr>
<td>5   0   5</td>
<td>13  4   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:
To convert an insert-and-copy length code to an insert length code and a copy length code, the following table can be used:

<table>
<thead>
<tr>
<th>Insert length code</th>
<th>Copy length code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>0-63</td>
</tr>
<tr>
<td>8-15</td>
<td>64-127</td>
</tr>
<tr>
<td>16-23</td>
<td>&lt;--- distance code 0</td>
</tr>
</tbody>
</table>

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 \(<\text{block type, block length}\>\). 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 Huffman code over the block type code alphabet. A block type code 0 means that the block type is the same as the type of the second last block from the same block category, while a block type code 1 means that the block type equals the last block type plus one. If the last block type is the maximal possible, then a block type code 1 means block type 0. Block type codes 2 - 257 represent block types 0 - 255. The second last and last block types are initialized with 0 and 1, respectively, at the beginning of each meta-block.

The first block type of each block category must be 0 and the block type of the first block switch command is therefore not encoded in the compressed data.

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 \([0..\text{NBLTYPESL}-1]\), \([0..\text{NBLTYPESI}-1]\), and \([0..\text{NBLTYPESD}-1]\), respectively. From this it follows that the alphabet size of literal, insert-and-copy length and distance block type codes is \(\text{NBLTYPES} + 2\), \(\text{NBLTYPESI} + 2\) and \(\text{NBLTYPESD} + 2\), respectively.

Each block length in the compressed data is represented with a pair \(<\text{block length code, extra bits}\>\). The block length code and the extra bits are encoded back-to-back, the block length code is encoded using a Huffman code over the block length code alphabet, while the extra bits value is encoded as a fixed-width machine integer. The number of extra bits can be 0 - 24, and it is dependent on the block length code. The symbols of the block length code alphabet, along with the number of extra bits and the range of block lengths are as follows:
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 Huffman 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 Huffman tree in the array of literal and distance Huffman trees.

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.

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, 4, 4, 0, 0, 4, 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, 44, 32, 32, 40, 28, 12,
12, 48, 52, 52, 52, 48, 52, 52, 52, 52, 52, 52, 52, 52, 48,
52, 52, 52, 52, 52, 48, 52, 52, 52, 52, 24, 12, 28, 12, 12,
12, 56, 60, 60, 60, 56, 60, 60, 60, 60, 60, 60, 60, 60, 56,
60, 60, 60, 60, 60, 56, 60, 60, 60, 60, 24, 12, 28, 12, 0,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,

Lut1 :=
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,
2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3, 2, 3,

Lut2 :=
0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1,
1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1,
1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
Given p1 is the last decoded byte and p2 is the second last decoded 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 kinds of context maps, for literals and 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 Huffman tree 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 Huffman tree for encoding a literal or distance code with context ID "cid" and block type "bltype" is

index of literal Huffman tree = CMAPL[bltype * 64 + cid]

index of distance Huffman tree = CMAPD[bltype * 4 + cid]

The values of the context map are encoded with the combination of run length encoding for zero values and Huffman 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 Huffman code has the following \(RLEMAX + NTREES\) symbols:

- 0: value zero
- 1: repeat a zero 2-3 times, read 1 bit for repeat length
- 2: repeat a zero 4-7 times, read 2 bits for repeat length
- \(\ldots\)
- \(RLEMAX\): repeat a zero \((2^{RLEMAX})-(2^{(RLEMAX+1)} - 1)\) times, read \(RLEMAX\) bits for repeat length
- \(RLEMAX + 1\): value 1
- \(\ldots\)
- \(RLEMAX + NTREES - 1\): 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 1xxxx
- Huffman code with alphabet size \(NTREES + RLEMAX\)
- Context map size values encoded with the above Huffman 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 Huffman code indexes

For the encoding of NTREES see Section 9.2.

8. Static dictionary

At any given point during decoding the compressed data, a reference to a duplicated string in the output produced so far has a maximum backward distance value, which is the minimum of the window size and the number of output bytes produced. However, decoding a distance from the input 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:

\[
\begin{align*}
\text{NWORDS}[\text{length}] &= 0 & \text{if length} &< 4 \\
\text{NWORDS}[\text{length}] &= (1 << \text{NDBITS}[\text{lengths}]) & \text{if length} &\geq 4
\end{align*}
\]

DOFFSET and DICTSIZE are defined by the following recursion:

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

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

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

Each static dictionary word has NTRANSFORMS 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:

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

The string copied to the output stream is computed by applying the transformation to the base dictionary word. If transform_id is greater than NTRANSFORMS - 1 or length is greater than 24, the compressed data set is invalid.

Each word transformation has the following form:

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

where the \_i subscript denotes the transform_id above. Each \text{T}_i is one of the following 20 elementary transforms:

\[
\text{Identity}, \text{OmitLast}1, \ldots, \text{OmitLast}9, \text{UppercaseFirst}, \text{UppercaseAll}, \text{OmitFirst}1, \ldots, \text{OmitFirst}9
\]

The form of these elementary transforms are as follows:

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

UppercaseFirst(word) = first UTF-8 character of word upper-cased

UppercaseAll(word) = all UTF-8 characters of word upper-cased

OmitFirstk(word) = the last (length(word) - k) bytes of word, or empty string if length(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:

\[
\begin{align*}
i &= 0 \\
&\text{while } i < \text{length(word)}: \\
&\quad \text{if } \text{word}[i] < 192: \\
&\quad \quad \text{if } \text{word}[i] \geq 97 \text{ and } \text{word}[i] \leq 122: \\
&\quad \quad \quad \text{word}[i] = \text{word}[i] \oplus 32 \\
&\quad \quad \quad i = i + 1 \\
&\quad \quad \text{else if } \text{word}[i] < 224: \\
&\quad \quad \quad \text{if } i + 1 < \text{length(word)}: \\
&\quad \quad \quad \quad \text{word}[i + 1] = \text{word}[i + 1] \oplus 32 \\
&\quad \quad \quad \quad i = i + 2 \\
&\quad \quad \text{else:} \\
&\quad \quad \quad \text{if } i + 2 < \text{length(word)}: \\
&\quad \quad \quad \quad \text{word}[i + 2] = \text{word}[i + 2] \oplus 5 \\
&\quad \quad \quad \quad i = i + 3 
\end{align*}
\]

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.

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:

1-4 bits: WBITS, a value in the range 16 - 24, value 16 is encoded with one 0 bit, and values 17 - 24 are encoded with bit pattern 1xxx
The size of the sliding window, which is the maximum value of any non-dictionary reference backward distance, is given by the following formula:

\[
\text{window size} = (1 << \text{WBITS}) - 16
\]

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: ISLAST, set to 1 if this is the last meta-block
1 bit: ISEMPTY, set to 1 if the meta-block is empty, this field is only present if ISLAST bit is set, since only the last meta-block can be empty
2 bits: MNIBBLES - 4, where MNIBBLES is # of nibbles to represent the length

MNIBBLES x 4 bits: MLEN - 1, where MLEN is the length of the meta-block in the input data in bytes

1 bit: ISUNCOMPRESSED, if set to 1, any bits of input 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 ISLAST bit is not set

1-11 bits: NBLTYPESL, # of literal block types, encoded with the following variable length code:

<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>1000</td>
</tr>
<tr>
<td>3-4</td>
<td>1001x</td>
</tr>
<tr>
<td>5-8</td>
<td>1010xx</td>
</tr>
<tr>
<td>9-16</td>
<td>1011xxx</td>
</tr>
<tr>
<td>17-32</td>
<td>1100xxxxxx</td>
</tr>
<tr>
<td>33-64</td>
<td>1101xxxxxx</td>
</tr>
<tr>
<td>65-128</td>
<td>1110xxxxxxx</td>
</tr>
<tr>
<td>129-256</td>
<td>1111xxxxxxx</td>
</tr>
</tbody>
</table>

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

Huffman code over the block length code alphabet for literal block lengths, appears only if NBLTYPESL >= 2
Block length code + Extra bits for first literal block length, appears only if NBLTYPESL >= 2

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

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

Huffman code over the block length code alphabet for insert-and-copy block lengths, only if NBLTYPESI >= 2

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

1-11 bits: NBLTYPESD, # of distance block types, encoded with the same variable length code as above

Huffman code over the block type code alphabet for distance block types, appears only if NBLTYPESD >= 2

Huffman code over the block length code alphabet for distance block lengths, only if NBLTYPESD >= 2

Block length code + Extra bits for first distance block length, 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 Huffman 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 Huffman 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 Huffman codes for literals

NBLTYPESI Huffman codes for insert-and-copy lengths

NTREESD Huffman 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 has ended

Block length code + Extra bits for next insert-and-copy block length, appears only if NBLTYPESI >= 2 and the previous insert and-copy block has ended

Insert-and-copy length, encoded as in section 5, using the insert-and-copy length Huffman 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 has ended

Block length code + Extra bits for next literal block length, appears only if NBLTYPESL >= 2 and the previous literal block has ended

Next byte of the input data, encoded with the literal Huffman code with the index determined by the previous two bytes of the input 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 has ended

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

Distance code, encoded as in section 4, using the distance
Huffman 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 output data is as follows:

```plaintext
    read window size
    do
        read ISLAST bit
        if ISLAST
            read ISEMPTY bit
            if ISEMPTY
                break from loop
        read MLEN
        if not ISLAST
            read ISUNCOMPRESSED bit
            if ISUNCOMPRESSED
                skip any bits up to the next byte boundary
                copy MLEN bytes of input to the output stream
                continue to the next meta-block
        loop for each three block categories (i = L, I, D)
            read NBLTYPESi
            if NBLTYPESi >= 2
                read Huffman code for block types, HTREE_BTYPE_i
                read Huffman code for block lengths, HTREE_BLEN_i
                read block length, BLEN_i
                set block type, BTYPE_i to 0
                initialize second last and last block types to 0 and 1
            else
                set block type, BTYPE_i to 0
                set block length, 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
```

Alakuijala & Szabadka   Expires November 16, 2014   [Page 27]
fill CMAPD[] with zeros
read array of Huffman codes for literals, HTREEL[]
read array of Huffman codes for insert-and-copy, HTREEI[]
read array of Huffman codes for distances, HTREED[]
do
  if BLEN_I is zero
    read block type using HTREE_BTYPE_I and set BTYPE_I
    read block length 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
    read block length 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 bytes of output
    read literal using HTREEL[CMAPL[64 * BTYPE_L + CIDL]]
    copy literal to output stream
  if number of output bytes produced in the loop is MLEN
    break from loop
  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
      read block length using HTREE_BLEN_D and set BLEN_D
      decrement BLEN_D
      compute context ID, CIDD from CLEN
      read distance code with HTREED[CMAPD[4 * BTYPE_D + CIDD]]
      compute distance by distance short code substitution
      move backwards distance bytes in the output stream, and
      copy CLEN bytes from this position to the output stream,
      or look up the static dictionary word and copy it to the
      output stream
    while number of output bytes produced in the loop < 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 cannot refer
past the beginning of the output stream and it can not be greater
than the window size; any such distance must be 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 output stream.
11. Security Considerations

As with any compressed file formats, decompressor implementations should handle all input byte sequences, not only those that conform to this specification and non-conformant input 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 input. Therefore decompressor implementations should perform bound-checking for each memory access that result from values decoded from the input stream.

12. IANA Considerations

This document has no actions for IANA.

13. Informative References


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/ directory within the font-compression-reference open-source project: https://code.google.com/p/font-compression-reference/source/browse/

Appendix A. Static dictionary data

The hexadecimal form of the DICT array is the following:

74696d65646f776e6c6966656c65676261636b636f646561746173686f776f6e6c7973697465636974796f70656e6a7573746c696b65676f726b
6f6e6c7973697465636974796f70656e6a7573746c696b65676f726b
6e7427574746f6e265651757479746856d6d737666f72676f74536561726368
616e63686f7261c6d6f73746c6f67757656756562662e6273703b636f75727365
61262e62646776696365696e3ecf2e133a205d65616520646964616c6f6766
657342547494e204d657869636f73746172747363656e7472656865667874
616466696749736c6e64617365747374567d06972655363686f656c6d66
66f72746469762563746e61726c796d3e56716e5c63566c565734200a4f
6e656a666e67657252235268696c67697671726467661646c656c56
69d706f72744666666963756265766724736b696e6c736e6174696665370
6f727473467677657665666c792028626e626568696e6446653743
6f726c6f767765746e6f62636f626567696e73706c616e647373
71637369746172674967373746563303070786c53616641616e61
656e63796368656d61696e427261a696c637661706c656c566f6776
223e6d2567666e644d73616c65616363656e73617374616e6162656e65
66f6f71656e7374617267496737374656330306c696e6465646c65
276526632220f3e0d0a2e67696622206f666c6166666f616465722f6866
2764736974675277377269766c697374656e656d61696e3c3d
73697a65632220d21670765616c67456774222065616e636865646965
67656765666f6263616c66696e6564696573696e6774657368617373
72656765666f6263616c66696e6564696573696e6774657368617373
72656765666f6263616c66696e6564696573696e6774657368617373
67656765666f6263616c66696e6564696573696e6774657368617373
67656765666f6263616c66696e6564696573696e6774657368617373
67656765666f6263616c66696e6564696573696e6774657368617373
7475726e6647tc207bd3bd61696e223e696e6c696e65736e5661746572
7160223e6661696c65616363656e7375736d696e75746562656163656e7875
74655733135370787e65737461746572656d6f7465636d61696e22266c6e6b6564
7269678443b7e6676e616c666f6f726d616c312eeb74646c6736967e65707072
696e6365666c6f7173ae706e672220666f72756d2e413636537337016705
72733736f756e6473657874656e644865696678743c696465525554462382
26616d703b204265666f72652e2057697468737456496f67776e65736d61
6e61676570726f6669746a5175657279616e6e65716c507162167d362f5767
687466616d6f7573766f6f666c656c6f6e6f6e65735220b297b69373276156c
736179696e676465636665666f6f6d652223e6865616452656e737572656272
616e636870696563657362ccf63e3b3774617465746f7022e3c72613639
6e67725763796a52622d2667736b7061636974973736578516c6275265175
2e6a07672203103c030303062f7461696e7469646c6573616d6f756e222c
496e636e6f6d6f666d6f68696f65722206c972696373746d616970666465
6564666f756e74956f6f676f2e46616d696c796c6f6665646d61726b6574
6c736320566965606c6f6757727562665923b671622066f72572f746769
76696e677572726f7235446561696e7d656c73657b696e7365727442cc6f67
3c2f666f6f74652667696e2e6661737465726167655e47433c626f647920
3130708207037071626761666f641679a756e696f24666c6172706c
6163656463f76657273706c7567696e352c303030207061675223e62f67347
6f66e2e746573742861766714657364654f6375617265617074734666
6736656561696e6457687365696e676a5723675974e6577732c2f7667256967
6578706572466966a75727977696474683d436f666965534415254206163
726f7335f696d6167675468726561646e6f61746976576f636b57462f7822
6f6e657363616c656e646172696f7066cc3ad7469636173616e746572696f72
6573646367556e646172696f706f6cc3ad7469636173616e746572696f72
64696665756e6369666477656c6573646f63756d656e746572696f70
6573646367556e646172696f706f6cc3ad7469636173616e746572696f72
6e7465737065726d616e646172696f706f6cc3ad7469636173616e746572696f72
676f657363616c656e646172696f706f6cc3ad7469636173616e746572696f72
6573646367556e646172696f706f6cc3ad7469636173616e746572696f72
64696665756e6369666477656c6573646f63756d656e746572696f70
6573646367556e646172696f706f6cc3ad7469636173616e746572696f72
6e7465737065726d616e646172696f706f6cc3ad7469636173616e746572696f72

Internet-Draft                   Brotli                         May 2014

Alakuijala & Szabadka   Expires November 16, 2014              [Page 76]
Internet-Draft                   Brotli                         May 2014

Alakuijala & Szabadka   Expires November 16, 2014             [Page 103]
The number of words for each length is given by the following bit-depth array:

\[
\text{NDBITS} := 0, 0, 0, 0, 10, 10, 11, 11, 10, 10,
\]
### Appendix B. List of word transformations

<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;</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>
<tr>
<td>21</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;&gt;&quot;</td>
</tr>
<tr>
<td>22</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;\n&quot;</td>
</tr>
<tr>
<td>23</td>
<td>&quot;&quot;</td>
<td>OmitLast3</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>24</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;]&quot;</td>
</tr>
<tr>
<td>25</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; for &quot;</td>
</tr>
<tr>
<td>26</td>
<td>&quot;&quot;</td>
<td>OmitFirst3</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>27</td>
<td>&quot;&quot;</td>
<td>OmitLast2</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>28</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; a &quot;</td>
</tr>
<tr>
<td>29</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; that &quot;</td>
</tr>
<tr>
<td>30</td>
<td>&quot;&quot;</td>
<td>UppercaseFirst</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>31</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;. &quot;</td>
</tr>
<tr>
<td>32</td>
<td>&quot;, &quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>33</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;, &quot;</td>
</tr>
<tr>
<td>34</td>
<td>&quot;&quot;</td>
<td>OmitFirst4</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>35</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; with &quot;</td>
</tr>
<tr>
<td>36</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>37</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; from &quot;</td>
</tr>
<tr>
<td>38</td>
<td>&quot;&quot;</td>
<td>Identity</td>
<td>&quot; by &quot;</td>
</tr>
<tr>
<td>39</td>
<td>&quot;&quot;</td>
<td>OmitFirst5</td>
<td>&quot;&quot;</td>
</tr>
<tr>
<td>40</td>
<td>&quot;&quot;</td>
<td>OmitFirst6</td>
<td>&quot;&quot;</td>
</tr>
</tbody>
</table>
"the " Identity ""
"OmitLast4 ""
"Identity ". The "
"UppercaseAll ""
"Identity " on "
"Identity " as "
"Identity " is "
"OmitLast7 ""
"OmitLast1 "ing "
"Identity "\n"
"Identity ":""
"Identity ", "
"Identity ", ed ""
"OmitFirst9 ""
"OmitFirst7 ""
"OmitLast6 ""
"Identity "{""
"UppercaseFirst ", "
"OmitLast8 ""
"Identity " at "
"Identity "ly "
"the " Identity " of "
"OmitLast5 ""
"OmitLast9 ""
"UppercaseFirst ", "
"UppercaseFirst \""
"Identity "{""
"UppercaseAll ""
"UppercaseFirst \">)"
"Identity "))""
"Identity "."
".com/" Identity ""
"the " Identity " of the "
"UppercaseFirst ", "
"Identity ". This "
"Identity ", "
"Identity ""
"UppercaseFirst "{""
"UppercaseFirst ", "
"Identity " not "
"Identity ")=\"
"Identity ", er "
"UppercaseAll ""
"Identity ")al "
"UppercaseAll ""
"Identity ")al "
"UppercaseAll ")\"
"UppercaseAll ")\"
Authors’ Addresses

Jyrki Alakuijala
Google, Inc

Email: jyrki@google.com

Zoltan Szabadka
Google, Inc

Email: szabadka@google.com