Brotli Compressed Data Format

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 document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7932.

Copyright Notice

Copyright (c) 2016 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 ......................................................4  
   1.4. Compliance .................................................4  
   1.5. Definitions of Terms and Conventions Used .................4  
       1.5.1. Packing into Bytes ..................................5  
2. Compressed Representation Overview ..............................6  
3. Compressed Representation of Prefix Codes ......................10  
   3.1. Introduction to Prefix Coding ................................10  
   3.2. Use of Prefix Coding in the Brotli Format ..................11  
   3.3. Alphabet Sizes ............................................13  
   3.4. Simple Prefix Codes .......................................14  
   3.5. Complex Prefix Codes ......................................15  
4. Encoding of Distances ..........................................17  
5. Encoding of Literal Insertion Lengths and Copy Lengths ........19  
6. Encoding of Block-Switch Commands ..............................22  
7. Context Modeling ...............................................23  
   7.2. Context ID for Distances ..................................26  
   7.3. Encoding of the Context Map ................................26  
8. Static Dictionary ..............................................28  
9. Compressed Data Format .........................................31  
   9.1. Format of the Stream Header ................................31  
   9.2. Format of the Meta-Block Header ...........................32  
   9.3. Format of the Meta-Block Data .............................35  
10. Decoding Algorithm ............................................36  
11. Considerations for Compressor Implementations .................38  
   11.1. Trivial Compressor .......................................39  
   11.2. Aligning Compressed Meta-Blocks to Byte Boundaries .......39  
   11.3. Creating Self-Contained Parts within the Compressed Data 40  
12. Security Considerations .......................................41  
13. IANA Considerations ...........................................42  
14. Informative References ........................................43  
Appendix A. Static Dictionary Data ................................44  
Appendix B. List of Word Transformations ..........................124  
Appendix C. Computing CRC-32 Check Values .........................127  
Appendix D. Source Code ..........................................127  
Acknowledgments ..................................................127  
Authors’ Addresses ...............................................128
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; hence, it 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; hence, it 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, 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 densely as the best currently available specialized algorithms.

This document is the authoritative specification of the brotli compressed data format. It defines the set of valid brotli compressed data streams and a decoder algorithm that produces the uncompressed data stream from a valid brotli compressed data stream.

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 [RFC1951]. 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 File Format 2.0 [WOFF2]; therefore, this specification is also intended for implementers of WOFF 2.0 compressors and decompressors.

1.3. Scope

This document 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 that 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 (lsb), and since we write numbers with the most significant digit on the left, we also write bytes with the most significant bit (msb) on the left. In the diagrams below, we number the bits of a byte so that bit 0 is the least significant bit, i.e., the bits are numbered:

```
+--------+
|76543210|
+--------+
```
Within a computer, a number may occupy multiple bytes. All multi-byte numbers in the format described here are stored with the least significant byte first (at the lower memory address). For example, the decimal number 520 is stored as:

```
0 1
+--------+--------+
|00001000|00000010|
+--------+--------+
```

```
^    ^
|    |
| + more significant byte = 2 * 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. Therefore, we must 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).

As an example, consider packing the following data elements into a sequence of 3 bytes: 3-bit integer value 6, 4-bit integer value 2, prefix code 110, prefix code 10, 12-bit integer value 3628.
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 0 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 10 to 24. The possible sliding window sizes range from 1 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, which is represented as a pair <length, backward distance>. There can be zero literal bytes in the command. The minimum length of the string to be
Each command in the compressed data is represented using three categories of prefix codes:

1) 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.

2) One set of prefix codes are for literals.

3) One 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 length defines an insertion length and a copy length. The insertion length determines the number of literals that immediately follow. The distance defines how far back to go for the copy and the copy length determines the number of bytes to copy. 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 length. 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 insert-and-copy block type directly determines which prefix code to use for the next insert-and-copy length. 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 xBlockSwitch(t, n) switches to block type t for a count of n elements. In this example, note that 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 are 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. The matrix of the prefix code indexes 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, 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 cannot 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. The code for a symbol is the sequence of 0’s and 1’s on the edges leading from the root to the leaf labeled with that symbol. For example:

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

A parser can decode the next symbol from 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 that 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] for additional information on Huffman codes.)

In the brotli format, note that 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 [HUFFMAN] 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>

That is, 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:

```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:

<table>
<thead>
<tr>
<th>N</th>
<th>bl_count[N]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>------------</td>
</tr>
<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></td>
<td>------------</td>
</tr>
<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

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 parameters defined in later sections. The following table summarizes the alphabet sizes for the various prefix codes and the sections of this document in which they are defined.

<table>
<thead>
<tr>
<th>Prefix Code</th>
<th>Alphabet Size</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>literal</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>distance</td>
<td>16 + NDIRECT + (48 &lt;&lt; NPOSTFIX)</td>
<td>Section 4</td>
</tr>
<tr>
<td>insert-and-copy length</td>
<td>704</td>
<td>Section 5</td>
</tr>
<tr>
<td>block count</td>
<td>26</td>
<td>Section 6</td>
</tr>
<tr>
<td>block type</td>
<td>NBLTYPESx + 2, (where x is I, L, or D)</td>
<td>Section 6</td>
</tr>
<tr>
<td>context map</td>
<td>NTREESx + RLEMAXx, (where x is L or D)</td>
<td>Section 7</td>
</tr>
</tbody>
</table>
3.4. Simple Prefix Codes

The first two bits of the compressed representation of each prefix code distinguish between simple and complex prefix codes. If this value is 1, then a simple prefix code follows as described in this section. Otherwise, a complex prefix code follows as described in Section 3.5.

A simple prefix code can have 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 = number 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. For example, for the alphabet of literal bytes, ALPHABET_BITS is 8. The value of each of the NSYM symbols above is the value of the ALPHABET_BITS width integer value. If the integer value is greater than or equal to the alphabet size, or the value is identical to a previous value, 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 Section 3.2. 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
   \( (0 = 3, \ldots, 3 = 6) \)
   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.
The next 3 bits indicate repeat length
   \( (0 = 3, \ldots, 7 = 10) \)
   A repeated code length code of 17 modifies the repeat count of the previous one as follows:
   \[ \text{repeat count} = (8 \times (\text{repeat count} - 2)) + (3..10 \text{ on the next 3 bits}) \]

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 or 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 it 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 variable-length code (as it appears in the compressed data, where the bits are parsed from right to left):

<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>0111</td>
</tr>
<tr>
<td>2</td>
<td>011</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>01</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, the number of skipped code lengths, can have values of 0, 2, or 3. 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 variable-length 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 length 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. For example, 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 length symbols, which is at most the size of the alphabet, 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 be equal to 32768. If the number of times to repeat the previous length or repeat a zero length would result in more lengths in total than the number of symbols in the alphabet, then the stream should be rejected as invalid.

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 \(<\text{distance code}, \text{extra bits}>\). 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 \(\text{NPOSTFIX} \ (0..3)\), and the number of direct distance codes, denoted by \(\text{NDIRECT} \ (0..120)\). Both of these parameters are encoded in the meta-block header. We will also use the following derived parameter:

\[
\text{POSTFIX\_MASK} = (1 \ll \text{NPOSTFIX}) - 1
\]
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 the 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-to-last distance" means the second-to-last distance that was not represented by a 0 distance symbol (and similar for "third-to-last distance" and "fourth-to-last distance"). Similarly, distances that represent static dictionary words (see Section 8) are not pushed to the ring buffer of last distances.

If a special distance symbol resolves to a zero or negative value, the stream should be rejected as invalid.

If NDIRECT is greater than zero, then the next NDIRECT distance symbols, from 16 to 15 + NDIRECT, represent distances from 1 to NDIRECT. Neither the special distance 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:

\[ \text{ndistbits} = 1 + ((\text{dcode} - \text{NDIRECT} - 16) >> (\text{NPOSTFIX} + 1)) \]
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" \((>= 16 + \text{NDIRECT})\), and extra bits "dextra", the backward distance is given by the following formula:

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

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>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>2</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>2</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>18</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>11</td>
<td>3</td>
<td>19</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>12</td>
<td>4</td>
<td>20</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>13</td>
<td>4</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>14</td>
<td>5</td>
<td>22</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>15</td>
<td>5</td>
<td>23</td>
<td>24</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>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>8</td>
<td>1</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>9</td>
<td>1</td>
<td>17</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>10</td>
<td>2</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>11</td>
<td>2</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>12</td>
<td>3</td>
<td>20</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>13</td>
<td>3</td>
<td>21</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>14</td>
<td>4</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>15</td>
<td>4</td>
<td>23</td>
<td>24</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:

<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>0..7</td>
<td>128..191</td>
</tr>
<tr>
<td>8..15</td>
<td>256..319</td>
</tr>
<tr>
<td>16..23</td>
<td>448..511</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 bits 0..2 (counted from the lsb) of the insert-and-copy length code. 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. If a block category has only one block type, the block count of the first block-switch command is also omitted from the compressed data; otherwise, it 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 NBLTYPESL + 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>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1..4</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5..8</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>9..12</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>13..16</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>17..24</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>25..32</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>41..48</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>49..64</td>
<td>9</td>
<td>4</td>
<td>4</td>
<td>65..80</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>4</td>
<td>81..96</td>
<td>11</td>
<td>4</td>
<td>4</td>
<td>97..112</td>
<td>12</td>
<td>5</td>
<td>5</td>
<td>113..144</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>5</td>
<td>145..176</td>
<td>14</td>
<td>5</td>
<td>5</td>
<td>177..208</td>
<td>15</td>
<td>5</td>
<td>5</td>
<td>209..240</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>6</td>
<td>241..304</td>
<td>17</td>
<td>6</td>
<td>6</td>
<td>305..368</td>
<td>18</td>
<td>7</td>
<td>7</td>
<td>369..496</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>8</td>
<td>497..752</td>
<td>20</td>
<td>9</td>
<td>9</td>
<td>753..1264</td>
<td>21</td>
<td>10</td>
<td>10</td>
<td>1265..2288</td>
</tr>
<tr>
<td>22</td>
<td>11</td>
<td>11</td>
<td>2289..4336</td>
<td>23</td>
<td>12</td>
<td>12</td>
<td>4337..8432</td>
<td>24</td>
<td>13</td>
<td>13</td>
<td>8433..16624</td>
</tr>
<tr>
<td>25</td>
<td>14</td>
<td>14</td>
<td>16625..32768</td>
<td>26</td>
<td>15</td>
<td>15</td>
<td>32768..65535</td>
<td>27</td>
<td>16</td>
<td>16</td>
<td>65535..131072</td>
</tr>
<tr>
<td>28</td>
<td>17</td>
<td>17</td>
<td>131072..262144</td>
<td>29</td>
<td>18</td>
<td>18</td>
<td>262144..524288</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>19</td>
<td>19</td>
<td>524288..1048576</td>
<td>31</td>
<td>20</td>
<td>20</td>
<td>1048576..2097152</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>21</td>
<td>21</td>
<td>2097152..4194304</td>
<td></td>
<td></td>
<td></td>
<td></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 block type and the context ID. 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 <block type, context ID> 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 of whether these bytes are produced by uncompressed meta-blocks, backward references, static dictionary 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:

* **LSB6**, where the Context ID is the value of six least significant bits of p1,

* **MSB6**, where the Context ID is the value of six most 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, 0, 4, 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, 44, 44, 44, 44, 44, 44,
12, 48, 52, 52, 52, 48, 52, 52, 48, 52, 52, 48, 52, 52, 48,
52, 52, 52, 52, 52, 48, 52, 52, 52, 52, 52, 24, 12, 28, 12,
12, 56, 60, 60, 60, 56, 60, 60, 60, 60, 60, 60, 60, 60, 56,
60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60, 60,
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, 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,
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, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
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,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5,
6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6

The lengths and the CRC-32 check values (see Appendix C) 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>
<tr>
<td>Lut1</td>
<td>256</td>
<td>0xd01a32f4</td>
</tr>
<tr>
<td>Lut2</td>
<td>256</td>
<td>0x0dd7a0d6</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]

From the lookup tables defined above and the operations to compute the context IDs, we can see that context IDs for literals are in the range of 0..63.

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

A 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 maps are two-dimensional matrices, encoded as one-dimensional arrays:

\[
\text{CMAPL}[0..(64 \times \text{NBLTYPESL} - 1)] \\
\text{CMAPD}[0..(4 \times \text{NBLTYPESD} - 1)]
\]

The index of the prefix code for encoding a literal or distance code with block type, BTYPE_x, and context ID, CIDx, is:

\[
\text{index of literal prefix code} = \text{CMAPL}[64 \times \text{BTYPE}_L + \text{CIDL}] \\
\text{index of distance prefix code} = \text{CMAPD}[4 \times \text{BTYPE}_D + \text{CIDD}]
\]

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. If a run length would result in
more lengths in total than the size of the context map, then
the stream should be rejected as invalid.

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.

Note that \( RLEMAX \) may be larger than the value necessary to represent
the longest sequence of zero values. Also, the \( NTREES \) value is
encoded right before the context map as described in 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;
    }
}
```

Note that the inverse move-to-front transform will not produce values
outside the \([0..NTREES-1]\) interval.
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. In this case, the distance is treated as a reference to a word in the static dictionary given in Appendix A. The copy length for a static dictionary reference must be between 4 and 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:

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

DOFFSET and DICTSIZE are defined by the following recursion:

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

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 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 \(<\text{length}, \text{distance}>\) pair can be reconstructed as follows:

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

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 the length is smaller than 4 or greater than 24, then the compressed stream should be rejected as 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, FermentFirst, FermentAll, OmitFirst1, ..., OmitFirst9, OmitLast1, ..., OmitLast9

The form of these elementary transforms is as follows:

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

FermentFirst(\text{word}) = \text{see below}

FermentAll(\text{word}) = \text{see below}

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

OmitLastk(\text{word}) = \text{the first (length(\text{word}) - k) bytes of word, or empty string if length(\text{word}) < k}
We define the FermentFirst and FermentAll transforms used in this specification by the following C language functions:

```c
int Ferment(uint8_t* word, int word_len, int pos) {
    if (word[pos] < 192) {
        if (word[pos] >= 97 and word[pos] <= 122) {
            word[pos] = word[pos] ^ 32;
        }
        return 1;
    } else if (word[pos] < 224) {
        if (pos + 1 < word_len) {
            word[pos + 1] = word[pos + 1] ^ 32;
        }
        return 2;
    } else {
        if (pos + 2 < word_len) {
            word[pos + 2] = word[pos + 2] ^ 5;
        }
        return 3;
    }
}

void FermentFirst(uint8_t* word, int word_len) {
    if (word_len > 0) {
        Ferment(word, word_len, 0);
    }
}

void FermentAll(uint8_t* word, int word_len) {
    int i = 0;
    while (i < word_len) {
        i += Ferment(word, word_len, i);
    }
}
```

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.

The maximum number of additional bytes that a transform may add to a base word is 13. Since the largest base word is 24 bytes long, a buffer of 38 bytes is sufficient to store any transformed words (counting a terminating zero byte).
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:

\[
\text{window size} = (1 << \text{WBITS}) - 16
\]
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: ISLASTEMPTY, if set to 1, the meta-block is empty; this
field is only present if ISLAST 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, number of nibbles to represent the uncompressed
length, encoded with the following fixed-length code:

<table>
<thead>
<tr>
<th>Value</th>
<th>Bit Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>00</td>
</tr>
<tr>
<td>5</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

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, number of bytes to represent
metadata length

MSKIPBYTES * 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 * 4 bits: MLEN - 1, where MLEN is the length of the meta-block uncompressed data in bytes (if MNIBBLES 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, number 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>xxxxx1001</td>
</tr>
<tr>
<td>33..64</td>
<td>xxxxx1011</td>
</tr>
<tr>
<td>65..128</td>
<td>xxxxxxx1101</td>
</tr>
<tr>
<td>129..256</td>
<td>xxxxxxx1111</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, number 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, appears only if NBLTYPESI >= 2

Prefix code over the block count code alphabet for insert-and-copy block counts, appears only if NBLTYPESI >= 2
Block count code + extra bits for first insert-and-copy block count, appears only if NBLTYPESI >= 2

1..11 bits: NBLTYPESD, number 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, appears only if NBLTYPESD >= 2

Block count code + extra bits for first distance block count, appears 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 * 2 bits: context mode for each literal block type

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

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

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

Distance context map, encoded as described in Section 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 Section 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 count, 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 index determined by the copy length, the current distance block type, and the distance context map, as described in Section 7.3, 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 the uncompressed bytes produced (i.e., the number of literals inserted plus the number of bytes copied from past data or generated from the static dictionary) over all the commands gives the uncompressed length, MLEN encoded in the meta-block header.

If the total number of uncompressed bytes produced after the insert part of the last command equals MLEN, then the copy length of the last command is ignored and will not produce any uncompressed output. In this case, the copy length of the last command can have any value. In any other case, if the number of literals to insert, the copy length, or the resulting dictionary word length would cause MLEN to be exceeded, then the stream should be rejected as invalid.

If the last command of the last non-empty meta-block does not end on a byte boundary, the unused bits in the last byte must be zeros.

10. Decoding Algorithm

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

read window size
    do
        read ISLAST bit
        if ISLAST
            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
  initialize second-to-last and last block types to 0 and 1
else
  set block type, BTYPE_i to 0
  set block count, BLEN_i to 16777216
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 literal prefix codes, HTREEL[]
read array of insert-and-copy length prefix codes, HTREEI[]
read array of distance prefix codes, 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 symbol using HTREEI[BTYPE_I]
    compute insert length, ILEN, and copy length, CLEN
    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 HTREEL[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 using HTREED[CMAPD[4*BTYPE_D + CIDD]]
    compute distance by distance short code substitution
    if distance code is not zero,
        and distance is not a static dictionary reference,
        push distance to the ring buffer of last distances
    if distance is less than the max allowed distance plus one
        move backwards distance bytes in the uncompressed data,
        and copy CLEN bytes from this position to
        the uncompressed stream
    else
        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

If the stream ends before the completion of the last meta-block, then
the stream should be rejected as invalid.

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. Considerations for Compressor Implementations

Since the intent of this document is to define the brotli compressed
data format without reference to any particular compression
algorithm, the material in this section is not part of the definition
of the format, and a compressor need not follow it in order to be
compliant.
11.1. Trivial Compressor

In this section, we present a very simple algorithm that produces a valid brotli stream representing an arbitrary sequence of uncompressed bytes in the form of the following C++ language function.

```cpp
string BrotliCompressTrivial(const string& u) {
    if (u.empty()) {
        return string(1, 6);
    }
    int i;
    string c;
    c.append(1, 12);
    for (i = 0; i + 65535 < u.size(); i += 65536) {
        c.append(1, 248);
        c.append(1, 255);
        c.append(1, 15);
        c.append(&u[i], 65536);
    }
    if (i < u.size()) {
        int r = u.size() - i - 1;
        c.append(1, (r & 31) << 3);
        c.append(1, r >> 5);
        c.append(1, 8 + (r >> 13));
        c.append(&u[i], r + 1);
    }
    c.append(1, 3);
    return c;
}
```

Note that this simple algorithm does not actually compress data, that is, the brotli representation will always be bigger than the original, but it shows that every sequence of N uncompressed bytes can be represented with a valid brotli stream that is not longer than N + (3 * (N >> 16) + 5) bytes.

11.2. Aligning Compressed Meta-Blocks to Byte Boundaries

As described in Section 9, only those meta-blocks that immediately follow an uncompressed meta-block or a metadata meta-block are guaranteed to start on a byte boundary. In some applications, it might be required that every non-metadata meta-block starts on a byte boundary. This can be achieved by appending an empty metadata meta-block after every non-metadata meta-block that does not end on a byte boundary.
11.3. Creating Self-Contained Parts within the Compressed Data

In some encoder implementations, it might be required to make a sequence of bytes within a brotli stream self-contained, that is, such that they can be decompressed independently from previous parts of the compressed data. This is a useful feature for three reasons. First, if a large compressed file is damaged, it is possible to recover some of the file after the damage. Second, it is useful when doing differential transfer of compressed data. If a sequence of uncompressed bytes is unchanged and compressed independently from previous data, then the compressed representation may also be unchanged and can therefore be transferred very cheaply. Third, if sequences of uncompressed bytes are compressed independently, it allows for parallel compression of these byte sequences within the same file, in addition to parallel compression of multiple files.

Given two sequences of uncompressed bytes, U0 and U1, we will now describe how to create two sequences of compressed bytes, C0 and C1, such that the concatenation of C0 and C1 is a valid brotli stream, and that C0 and C1 (together with the first byte of C0 that contains the window size) can be decompressed independently from each other to U0 and U1.

When compressing the byte sequence U0 to produce C0, we can use any compressor that works on the complete set of uncompressed bytes U0, with the following two changes. First, the ISLAST bit of the last meta-block of C0 must not be set. Second, C0 must end at a byte-boundary, which can be ensured by appending an empty metadata meta-block to it, as in Section 11.2.

When compressing the byte sequence U1 to produce C1, we can use any compressor that starts a new meta-block at the beginning of U1 within the U0+U1 input stream, with the following two changes. First, backward distances in C1 must not refer to static dictionary words or uncompressed bytes in U0. Even if a sequence of bytes in U1 would match a static dictionary word, or a sequence of bytes that overlaps U0, the compressor must represent this sequence of bytes with a combination of literal insertions and backward references to bytes in U1 instead. Second, the ring buffer of last four distances must be replenished first with distances in C1 before using it to encode other distances in C1. Note that both compressors producing C0 and C1 have to use the same window size, but the stream header is emitted only by the compressor that produces C0.

Note that this method can be easily generalized to more than two sequences of uncompressed bytes.
12. 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 rejected as invalid.

A possible attack against a system containing a decompressor implementation (e.g., a web browser) is to exploit a buffer overflow triggered by invalid compressed data. Therefore, decompressor implementations should perform bounds-checking for each memory access that result from values decoded from the compressed stream and derivatives thereof.

Another possible attack against a system containing a decompressor implementation is to provide it (either valid or invalid) compressed data that can make the decompressor system’s resource consumption (CPU, memory, or storage) to be disproportionately large compared to the size of the compressed data. In addition to the size of the compressed data, the amount of CPU, memory, and storage required to decompress a single compressed meta-block within a brotli stream is controlled by the following two parameters: the size of the uncompressed meta-block, which is encoded at the start of the compressed meta-block, and the size of the sliding window, which is encoded at the start of the brotli stream. Decompressor implementations in systems where memory or storage is constrained should perform a sanity-check on these two parameters. The uncompressed meta-block size that was decoded from the compressed stream should be compared against either a hard limit, given by the system’s constraints or some expectation about the uncompressed data, or against a certain multiple of the size of the compressed data. If the uncompressed meta-block size is determined to be too high, the compressed data should be rejected. Likewise, when the complete uncompressed stream is kept in the system containing the decompressor implementation, the total uncompressed size of the stream should be checked before decompressing each additional meta-block. If the size of the sliding window that was decoded from the start of the compressed stream is greater than a certain soft limit, then the decompressor implementation should, at first, allocate a smaller sliding window that fits the first uncompressed meta-block, and afterwards, before decompressing each additional meta-block, it should increase the size of the sliding window until the sliding window size specified in the compressed data is reached.
Correspondingly, possible attacks against a system containing a compressor implementation (e.g., a web server) are to exploit a buffer overflow or cause disproportionately large resource consumption by providing, e.g., uncompressible data. As described in Section 11.1, an output buffer of

\[ S(N) = N + (3 \times (N >> 16) + 5) \]

bytes is sufficient to hold a valid compressed brotli stream representing an arbitrary sequence of \( N \) uncompressed bytes. Therefore, compressor implementations should allocate at least \( S(N) \) bytes of output buffer before compressing \( N \) bytes of data with unknown compressibility and should perform bounds-checking for each write into this output buffer. If their output buffer is full, compressor implementations should revert to the trivial compression algorithm described in Section 11.1. The resource consumption of a compressor implementation for a particular input data depends mostly on the algorithm used to find backward matches and on the algorithm used to construct context maps and prefix codes and only to a lesser extent on the input data itself. If the system containing a compressor implementation is overloaded, a possible way to reduce resource usage is to switch to more simple algorithms for backward reference search and prefix code construction, or to fall back to the trivial compression algorithm described in Section 11.1.

A possible attack against a system that sends compressed data over an encrypted channel is the following. An attacker who can repeatedly mix arbitrary (attacker-supplied) data with secret data (passwords, cookies) and observe the length of the ciphertext can potentially reconstruct the secret data. To protect against this kind of attack, applications should not mix sensitive data with non-sensitive, potentially attacker-supplied data in the same compressed stream.

13. IANA Considerations

The "HTTP Content Coding Registry" has been updated with the registration below:

```
+--------+-------------------------------------+------------+
| Name   | Description                         | Reference  |
+--------+-------------------------------------+------------+
| br     | Brotli Compressed Data Format       | RFC 7932   |
+--------+-------------------------------------+------------+
```
14. Informative References


Appendix A. Static Dictionary Data

The hexadecimal form of the DICT array is the following, where the length is 122,784 bytes and the CRC-32 of the byte sequence is 0x5136cb04.
Alakuijala & Szabadka         Informational                    [Page 45]
Alakuijala & Szabadka  Informational  [Page 58]
Alakuijala & Szabadka Informational [Page 61]
Alakuijala & Szabadka  Informational  [Page 65]
RFC 7932                         Brotli                        July 2016

Alakuijala & Szabadka         Informational                   [Page 115]
Brotli                        Informational                   [Page 122]

Alakuijala & Szabadka
The number of words for each length is given by the following bit-depth array:

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

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 FermentFirst, 2 for FermentAll, 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 648 bytes, and the CRC-32 is 0x3d965f81.

<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>FermentFirst</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>FermentFirst</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>FermentFirst</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>
</tbody>
</table>
"" Identity "\"
" Identity ".
" Identity "\">
" Identity "\n"
" Identity "}
" Identity " for 
" OmitFirst3 
" OmitLast2 
" Identity " a 
" Identity " that 
" FermentFirst 
" Identity ".
" Identity "
" Identity "
" Identity ",
" OmitFirst4 
" Identity " with 
" Identity "/
" Identity " from 
" Identity " by 
" OmitFirst5 
" OmitFirst6 
" the " Identity 
" OmitLast4 
" Identity ". The 
" FermentAll 
" Identity " on 
" Identity " as 
" Identity " is 
" OmitLast7 
" OmitLast1 "ing "
" Identity "\n\t"
" Identity ".
" Identity "ed 
" OmitFirst9 
" OmitFirst7 
" OmitLast6 
" Identity "(" 
" FermentFirst ",
" OmitLast8 
" Identity " at 
" Identity "ly 
" the " Identity " of 
" OmitLast5 
" OmitLast9 
" FermentFirst ",
Appendix C. Computing CRC-32 Check Values

For the purpose of this specification, we define the CRC-32 check value of a byte sequence with the following C language function:

```c
uint32_t CRC32(const uint8_t* v, const int len) {
    const uint32_t poly = 0xedb88320UL;
    uint32_t crc, c;
    int i, k;
    crc = 0xffffffffUL;
    for (i = 0; i < len; ++i) {
        c = (crc ^ v[i]) & 0xff;
        for (k = 0; k < 8; k++)
            c = c & 1 ? poly ^ (c >> 1) : c >> 1;
        crc = c ^ (crc >> 8);
    }
    return crc ^ 0xffffffffUL;
}
```

Appendix D. Source Code

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

Acknowledgments

The authors would like to thank Mark Adler, Eugene Kliuchnikov, Robert Obryk, Thomas Pickert, Joe Tsai, and Lode Vandevenne 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.
Authors’ Addresses

Jyrki Alakuijala
Google, Inc.

Email: jyrki@google.com

Zoltan Szabadka
Google, Inc.

Email: szabadka@google.com