Stream: Internet Engineering Task Force (IETF)
RFC: 8878
Obsoletes: 8478
Category: Informational
Published: February 2021
ISSN: 2070-1721
Authors: Y. Collet, *Facebook*; M. Kucherawy, Ed., *Facebook*

RFC 8878
Zstandard Compression and the 'application/zstd' Media Type

Abstract

Zstandard, or "zstd" (pronounced "zee standard"), is a lossless data compression mechanism. This document describes the mechanism and registers a media type, content encoding, and a structured syntax suffix to be used when transporting zstd-compressed content via MIME.

Despite use of the word "standard" as part of Zstandard, readers are advised that this document is not an Internet Standards Track specification; it is being published for informational purposes only.

This document replaces and obsoletes RFC 8478.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Zstandard, or "zstd" (pronounced "zee standard"), is a data compression mechanism, akin to gzip [RFC1952].

Despite use of the word "standard" as part of its name, readers are advised that this document is not an Internet Standards Track specification; it is being published for informational purposes only.

This document describes the Zstandard format. Also, to enable the transport of a data object compressed with Zstandard, this document registers a media type, content encoding, and structured syntax suffix that can be used to identify such content when it is used in a payload.

2. Definitions

Some terms used elsewhere in this document are defined here for clarity.

uncompressed: Describes an arbitrary set of bytes in their original form, prior to being subjected to compression.

compressed: Describes the result of passing a set of bytes through this mechanism. The original input has thus been compressed.

decompressed: Describes the result of passing a set of bytes through the reverse of this mechanism. When this is successful, the decompressed payload and the uncompressed payload are indistinguishable.

encode: The process of translating data from one form to another; this may include compression, or it may refer to other translations done as part of this specification.

decode: The reverse of "encode"; describes a process of reversing a prior encoding to recover the original content.

frame: Content compressed by Zstandard is transformed into a Zstandard frame. Multiple frames can be appended into a single file or stream. A frame is completely independent, has a defined beginning and end, and has a set of parameters that tells the decoder how to decompress it.

block: A frame encapsulates one or multiple blocks. Each block contains arbitrary content, which is described by its header, and has a guaranteed maximum content size that depends upon frame parameters. Unlike frames, each block depends on previous blocks for proper decoding. However, each block can be decompressed without waiting for its successor, allowing streaming operations.

natural order: A sequence or ordering of objects or values that is typical of that type of object or value. A set of unique integers, for example, is in "natural order" if, when progressing from one element in the set or sequence to the next, there is never a decrease in value.

The naming convention for identifiers within the specification is Mixed_Case_With_Underscores. Identifiers inside square brackets indicate that the identifier is optional in the presented context.

3. Compression Algorithm

This section describes the Zstandard algorithm.

The purpose of this document is to define a lossless compressed data format that is a) independent of the CPU type, operating system, file system, and character set and b) suitable for file compression and pipe and streaming compression, using the Zstandard algorithm. The text of the specification assumes a basic background in programming at the level of bits and other primitive data representations.

The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage; hence, it can be used in data communications. The format uses the Zstandard compression method, and an optional xxHash-64 checksum method [XXHASH], for detection of data corruption.

The data format defined by this specification does not attempt to allow random access to compressed data.

Unless otherwise indicated below, a compliant compressor must produce data sets that conform to the specifications presented here. However, it does not need to support all options.

A compliant decompressor must be able to decompress at least one working set of parameters that conforms to the specifications presented here. It may also ignore informative fields, such as the checksum. Whenever it does not support a parameter defined in the compressed stream, it must produce an unambiguous error code and associated error message explaining which parameter is unsupported.

This specification is intended for use by implementers of software to compress data into Zstandard format and/or decompress data from Zstandard format. The Zstandard format is supported by an open-source reference implementation, written in portable C, and available at [ZSTD].

3.1. Frames

Zstandard compressed data is made up of one or more frames. Each frame is independent and can be decompressed independently of other frames. The decompressed content of multiple concatenated frames is the concatenation of each frame's decompressed content.

There are two frame formats defined for Zstandard: Zstandard frames and skippable frames. Zstandard frames contain compressed data, while skippable frames contain custom user metadata.

3.1.1. Zstandard Frames

The structure of a single Zstandard frame is as follows:

Magic_Number	4 bytes
Frame_Header	2-14 bytes
Data_Block	n bytes
[More Data_Blocks]	
[Content_Checksum]	4 bytes

Table 1: The Structure of a Single Zstandard Frame

Magic_Number: 4 bytes, little-endian format. Value: 0xFD2FB528.

Frame_Header: 2 to 14 bytes, detailed in Section 3.1.1.1.

Data_Block: Detailed in Section 3.1.1.2. This is where data appears.

Content_Checksum: An optional 32-bit checksum, only present if Content_Checksum_Flag is set. The content checksum is the result of the XXH64() hash function [XXHASH] digesting the original (decoded) data as input, and a seed of zero. The low 4 bytes of the checksum are stored in little-endian format.

The magic number was selected to be less probable to find at the beginning of an arbitrary file. It avoids trivial patterns (0x00, 0xFF, repeated bytes, increasing bytes, etc.), contains byte values outside of the ASCII range, and doesn't map into UTF-8 space, all of which reduce the likelihood of its appearance at the top of a text file.

3.1.1.1. Frame Header

The frame header has a variable size, with a minimum of 2 bytes up to a maximum of 14 bytes depending on optional parameters. The structure of Frame_Header is as follows:

Frame_Header_Descriptor	1 byte
[Window_Descriptor]	0-1 byte
[Dictionary_ID]	0-4 bytes
[Frame_Content_Size]	0-8 bytes

Table 2: The Structure of Frame_Header

3.1.1.1.1. Frame_Header_Descriptor

The first header's byte is called the Frame_Header_Descriptor. It describes which other fields are present. Decoding this byte is enough to tell the size of Frame_Header.

Bit Number	Field Name
7-6	Frame_Content_Size_Flag
5	Single_Segment_Flag
4	(unused)
3	(reserved)
2	Content_Checksum_Flag
1-0	Dictionary_ID_Flag

Table 3: The Frame_Header_Descriptor

In Table 3, bit 7 is the highest bit, while bit 0 is the lowest one.

3.1.1.1.1.1. Frame_Content_Size_Flag

This is a 2-bit flag (equivalent to Frame_Header_Descriptor right-shifted 6 bits) specifying whether Frame_Content_Size (the decompressed data size) is provided within the header. Frame_Content_Size_Flag provides FCS_Field_Size, which is the number of bytes used by Frame_Content_Size according to Table 4:

Frame_Content_Size_Flag	0	1	2	3
FCS_Field_Size	0 or 1	2	4	8

Table 4: Frame_Content_Size_Flag Provides FCS_Field_Size

When Frame_Content_Size_Flag is 0, FCS_Field_Size depends on Single_Segment_Flag: if Single_Segment_Flag is set, FCS_Field_Size is 1. Otherwise, FCS_Field_Size is 0; Frame_Content_Size is not provided.

3.1.1.1.1.2. Single_Segment_Flag

If this flag is set, data must be regenerated within a single continuous memory segment.

In this case, Window_Descriptor byte is skipped, but Frame_Content_Size is necessarily present. As a consequence, the decoder must allocate a memory segment of a size equal to or larger than Frame_Content_Size.

In order to protect the decoder from unreasonable memory requirements, a decoder is allowed to reject a compressed frame that requests a memory size beyond the decoder's authorized range.

For broader compatibility, decoders are recommended to support memory sizes of at least 8 MB. This is only a recommendation; each decoder is free to support higher or lower limits, depending on local limitations.

3.1.1.1.1.3. Unused Bit

A decoder compliant with this specification version shall not interpret this bit. It might be used in a future version to signal a property that is not mandatory to properly decode the frame. An encoder compliant with this specification must set this bit to zero.

3.1.1.1.1.4. Reserved Bit

This bit is reserved for some future feature. Its value must be zero. A decoder compliant with this specification version must ensure it is not set. This bit may be used in a future revision to signal a feature that must be interpreted to decode the frame correctly.

3.1.1.1.1.5. Content_Checksum_Flag

If this flag is set, a 32-bit Content_Checksum will be present at the frame's end. See the description of Content_Checksum above.

3.1.1.1.1.6. Dictionary_ID_Flag

This is a 2-bit flag (= Frame_Header_Descriptor & 0x3) indicating whether a dictionary ID is provided within the header. It also specifies the size of this field as DID_Field_Size:

Dictionary_ID_Flag	0	1	2	3
DID_Field_Size	0	1	2	4

Table 5: Dictionary_ID_Flag

3.1.1.1.2. Window Descriptor

This provides guarantees about the minimum memory buffer required to decompress a frame. This information is important for decoders to allocate enough memory.

The Window_Descriptor byte is optional. When Single_Segment_Flag is set, Window_Descriptor is not present. In this case, Window_Size is Frame_Content_Size, which can be any value from 0 to 2^{64} - 1 bytes (16 ExaBytes).

Bit Number	7-3	2-0
Field Name	Exponent	Mantissa

Table 6: Window_Descriptor

The minimum memory buffer size is called Window_Size. It is described by the following formulas:

```
   windowLog = 10 + Exponent;
   windowBase = 1 << windowLog;
   windowAdd = (windowBase / 8) * Mantissa;
   Window_Size = windowBase + windowAdd;
```

The minimum Window_Size is 1 KB. The maximum Window_Size is (1<<41) + 7*(1<<38) bytes, which is 3.75 TB.

In general, larger Window_Size values tend to improve the compression ratio, but at the cost of increased memory usage.

To properly decode compressed data, a decoder will need to allocate a buffer of at least Window_Size bytes.

In order to protect decoders from unreasonable memory requirements, a decoder is allowed to reject a compressed frame that requests a memory size beyond the decoder's authorized range.

For improved interoperability, it's recommended for decoders to support values of Window_Size up to 8 MB and for encoders not to generate frames requiring a Window_Size larger than 8 MB. It's merely a recommendation though, and decoders are free to support higher or lower limits, depending on local limitations.

3.1.1.1.3. Dictionary_ID

This is a field of variable size, which contains the ID of the dictionary required to properly decode the frame. This field is optional. When it's not present, it's up to the decoder to know which dictionary to use.

Dictionary_ID field size is provided by DID_Field_Size. DID_Field_Size is directly derived from the value of Dictionary_ID_Flag. One byte can represent an ID 0-255; 2 bytes can represent an ID 0-65535; 4 bytes can represent an ID 0-4294967295. Format is little-endian.

It is permitted to represent a small ID (for example, 13) with a large 4-byte dictionary ID, even if it is less efficient.

Within private environments, any dictionary ID can be used. However, for frames and dictionaries distributed in public space, Dictionary_ID must be attributed carefully. The following ranges are reserved for use only with dictionaries that have been registered with IANA (see Section 7.4):

low range:	<= 32767
high range:	>= (1 << 31)

Any other value for Dictionary_ID can be used by private arrangement between participants.

Any payload presented for decompression that references an unregistered reserved dictionary ID results in an error.

3.1.1.1.4. Frame_Content_Size

This is the original (uncompressed) size. This information is optional. Frame_Content_Size uses a variable number of bytes, provided by FCS_Field_Size. FCS_Field_Size is provided by the value of Frame_Content_Size_Flag. FCS_Field_Size can be equal to 0 (not present), 1, 2, 4, or 8 bytes.

FCS Field Size	Range
0	unknown
1	0 - 255
2	256 - 65791
4	$0 - 2^{32} - 1$
8	$0 - 2^{64} - 1$

Table 7: Frame_Content_Size

Frame_Content_Size format is little-endian. When FCS_Field_Size is 1, 4, or 8 bytes, the value is read directly. When FCS_Field_Size is 2, the offset of 256 is added. It's allowed to represent a small size (for example, 18) using any compatible variant.

3.1.1.2. Blocks

After Magic_Number and Frame_Header, there are some number of blocks. Each frame must have at least 1 block, but there is no upper limit on the number of blocks per frame.

The structure of a block is as follows:

Block_Header	Block_Content
3 bytes	n bytes

Table 8: The Structure of a Block

Block_Header uses 3 bytes, written using little-endian convention. It contains three fields:

Last_Block	Block_Type	Block_Size
bit 0	bits 1-2	bits 3-23

Table 9: Block_Header

3.1.1.2.1. Last_Block

The lowest bit (Last_Block) signals whether this block is the last one. The frame will end after this last block. It may be followed by an optional Content_Checksum (see Section 3.1.1).

3.1.1.2.2. Block_Type

The next 2 bits represent the Block_Type. There are four block types:

Value	Block_Type
0	Raw_Block
1	RLE_Block
2	Compressed_Block
3	Reserved

Table 10: The Four Block Types

Raw_Block: This is an uncompressed block. Block_Content contains Block_Size bytes.

RLE_Block: This is a single byte, repeated Block_Size times. Block_Content consists of a single byte. On the decompression side, this byte must be repeated Block_Size times.

Compressed_Block: This is a compressed block as described in Section 3.1.1.3. Block_Size is the length of Block_Content, namely the compressed data. The decompressed size is not known, but its maximum possible value is guaranteed (see below).

Reserved: This is not a block. This value cannot be used with the current specification. If such a value is present, it is considered to be corrupt data, and a compliant decoder must reject it.

3.1.1.2.3. Block_Size

The upper 21 bits of Block_Header represent the Block_Size.

When Block_Type is Compressed_Block or Raw_Block, Block_Size is the size of Block_Content (hence excluding Block_Header).

When Block_Type is RLE_Block, since Block_Content's size is always 1, Block_Size represents the number of times this byte must be repeated.

Block_Size is limited by Block_Maximum_Size (see below).

3.1.1.2.4. Block_Content and Block_Maximum_Size

The size of Block_Content is limited by Block_Maximum_Size, which is the smallest of:

- Window_Size
- 128 KB

Block_Maximum_Size is constant for a given frame. This maximum is applicable to both the decompressed size and the compressed size of any block in the frame.

The reasoning for this limit is that a decoder can read this information at the beginning of a frame and use it to allocate buffers. The guarantees on the size of blocks ensure that the buffers will be large enough for any following block of the valid frame.

If the compressed block is larger than the uncompressed one, sending the uncompressed block (i.e., a Raw_Block) is recommended instead.

3.1.1.3. Compressed Blocks

To decompress a compressed block, the compressed size must be provided from the Block_Size field within Block_Header.

A compressed block consists of two sections: a Literals_Section (Section 3.1.1.3.1) and a Sequences_Section (Section 3.1.1.3.2). The results of the two sections are then combined to produce the decompressed data in Sequence Execution (Section 3.1.1.4).

To decode a compressed block, the following elements are necessary:

- Previous decoded data, up to a distance of Window_Size, or the beginning of the Frame, whichever is smaller. Single_Segment_Flag will be set in the latter case.
- List of "recent offsets" from the previous Compressed_Block.

- The previous Huffman tree, required by Treeless_Literals_Block type.
- Previous Finite State Entropy (FSE) decoding tables, required by Repeat_Mode, for each symbol type (literals length codes, match length codes, offset codes).

Note that decoding tables are not always from the previous Compressed_Block:

- Every decoding table can come from a dictionary.
- The Huffman tree comes from the previous Compressed_Literals_Block.

3.1.1.3.1. Literals_Section_Header

All literals are regrouped in the first part of the block. They can be decoded first and then copied during Sequence Execution (see Section 3.1.1.4), or they can be decoded on the flow during Sequence Execution.

Literals can be stored uncompressed or compressed using Huffman prefix codes. When compressed, an optional tree description can be present, followed by 1 or 4 streams.

Literals_Section_Header
[Huffman_Tree_Description]
[Jump_Table]
Stream_1
[Stream_2]
[Stream_3]
[Stream_4]

Table 11: Compressed Literals

3.1.1.3.1.1. Literals_Section_Header

This field describes how literals are packed. It's a byte-aligned variable-size bit field, ranging from 1 to 5 bytes, using little-endian convention.

Literals_Block_Type	2 bits
Size_Format	1-2 bits
Regenerated_Size	5-20 bits
[Compressed_Size]	0-18 bits

Table 12: Literals_Section_Header

In this representation, bits at the top are the lowest bits.

The Literals_Block_Type field uses the two lowest bits of the first byte, describing four different block types:

Literals_Block_Type	Value
Raw_Literals_Block	0
RLE_Literals_Block	1
Compressed_Literals_Block	2
Treeless_Literals_Block	3

Table 13: Literals_Block_Type

Raw_Literals_Block: Literals are stored uncompressed. Literals_Section_Content is Regenerated_Size.

RLE_Literals_Block: Literals consist of a single-byte value repeated Regenerated_Size times. Literals_Section_Content is 1.

Compressed_Literals_Block: This is a standard Huffman-compressed block, starting with a Huffman tree description. See details below. Literals_Section_Content is Compressed_Size.

Treeless_Literals_Block: This is a Huffman-compressed block, using the Huffman tree from the previous Compressed_Literals_Block or a dictionary if there is no previous Huffman-compressed literals block. Huffman_Tree_Description will be skipped. Note that if this mode is triggered without any previous Huffman table in the frame (or dictionary, per Section 5), it should be treated as data corruption. Literals_Section_Content is Compressed_Size.

The Size_Format is divided into two families:

- For Raw_Literals_Block and RLE_Literals_Block, it's only necessary to decode Regenerated_Size. There is no Compressed_Size field.
- For Compressed_Block and Treeless_Literals_Block, it's required to decode both Compressed_Size and Regenerated_Size (the decompressed size). It's also necessary to decode the number of streams (1 or 4).

For values spanning several bytes, the convention is little endian.

Size_Format for Raw_Literals_Block and RLE_Literals_Block uses 1 or 2 bits. Its value is (Literals_Section_Header[0]>>2) & 0x3.

Size_Format == 00 or 10: Size_Format uses 1 bit. Regenerated_Size uses 5 bits (value 0-31). Literals_Section_Header uses 1 byte. Regenerated_Size = Literal_Section_Header[0]>>3.

Size_Format == 01: Size_Format uses 2 bits. Regenerated_Size uses 12 bits (values 0-4095). Literals_Section_Header uses 2 bytes. Regenerated_Size = (Literals_Section_Header[0]>>4) + (Literals_Section_Header[1]<<4).

Size_Format == 11: Size_Format uses 2 bits. Regenerated_Size uses 20 bits (values 0-1048575). Literals_Section_Header uses 3 bytes. Regenerated_Size = (Literals_Section_Header[0]>>4) + (Literals_Section_Header[1]<<4) + (Literals_Section_Header[2]<<12).

Only Stream_1 is present for these cases. Note that it is permitted to represent a short value (for example, 13) using a long format, even if it's less efficient.

Size_Format for Compressed_Literals_Block and Treeless_Literals_Block always uses 2 bits.

Size_Format == 00: A single stream. Both Regenerated_Size and Compressed_Size use 10 bits (values 0-1023). Literals_Section_Header uses 3 bytes.

Size_Format == 01: 4 streams. Both Regenerated_Size and Compressed_Size use 10 bits (values 0-1023). Literals_Section_Header uses 3 bytes.

Size_Format == 10: 4 streams. Both Regenerated_Size and Compressed_Size use 14 bits (values 0-16383). Literals_Section_Header uses 4 bytes.

Size_Format == 11: 4 streams. Both Regenerated_Size and Compressed_Size use 18 bits (values 0-262143). Literals_Section_Header uses 5 bytes.

Both the Compressed_Size and Regenerated_Size fields follow little-endian convention. Note that Compressed_Size includes the size of the Huffman_Tree_Description when it is present.

3.1.1.3.1.2. Raw_Literals_Block

The data in Stream_1 is Regenerated_Size bytes long. It contains the raw literals data to be used during Sequence Execution (Section 3.1.1.3.2).

3.1.1.3.1.3. RLE_Literals_Block

Stream_1 consists of a single byte that should be repeated Regenerated_Size times to generate the decoded literals.

3.1.1.3.1.4. Compressed_Literals_Block and Treeless_Literals_Block

Both of these modes contain Huffman-coded data. For Treeless_Literals_Block, the Huffman table comes from the previously compressed literals block, or from a dictionary; see Section 5.

3.1.1.3.1.5. Huffman_Tree_Description

This section is only present when the Literals_Block_Type type is Compressed_Literals_Block (2). The format of Huffman_Tree_Description can be found in Section 4.2.1. The size of Huffman_Tree_Description is determined during the decoding process. It must be used to determine where streams begin.

```
   Total_Streams_Size = Compressed_Size
                        - Huffman_Tree_Description_Size
```

3.1.1.3.1.6. Jump_Table

The Jump_Table is only present when there are 4 Huffman-coded streams.

(Reminder: Huffman-compressed data consists of either 1 or 4 Huffman-coded streams.)

If only 1 stream is present, it is a single bitstream occupying the entire remaining portion of the literals block, encoded as described within Section 4.2.2.

If there are 4 streams, Literals_Section_Header only provides enough information to know the decompressed and compressed sizes of all 4 streams combined. The decompressed size of each stream is equal to (Regenerated_Size+3)/4, except for the last stream, which may be up to 3 bytes smaller, to reach a total decompressed size as specified in Regenerated_Size.

The compressed size of each stream is provided explicitly in the Jump_Table. The Jump_Table is 6 bytes long and consists of three 2-byte little-endian fields, describing the compressed sizes of the first 3 streams. Stream4_Size is computed from Total_Streams_Size minus the sizes of other streams.

```
   Stream4_Size = Total_Streams_Size - 6
                  - Stream1_Size - Stream2_Size
                  - Stream3_Size
```

Note that if Stream1_Size + Stream2_Size + Stream3_Size exceeds Total_Streams_Size, the data are considered corrupted.

Each of these 4 bitstreams is then decoded independently as a Huffman-coded stream, as described in Section 4.2.2.

3.1.1.3.2. Sequences_Section

A compressed block is a succession of sequences. A sequence is a literal copy command, followed by a match copy command. A literal copy command specifies a length. It is the number of bytes to be copied (or extracted) from the Literals_Section. A match copy command specifies an offset and a length.

When all sequences are decoded, if there are literals left in the Literals_Section, these bytes are added at the end of the block.

This is described in more detail in Section 3.1.1.4.

The Sequences_Section regroups all symbols required to decode commands. There are three symbol types: literals length codes, offset codes, and match length codes. They are encoded together, interleaved, in a single "bitstream".

The Sequences_Section starts by a header, followed by optional probability tables for each symbol type, followed by the bitstream.

```
   Sequences_Section_Header
     [Literals_Length_Table]
     [Offset_Table]
     [Match_Length_Table]
     bitStream
```

To decode the Sequences_Section, it's necessary to know its size. This size is deduced from the size of the Literals_Section: Sequences_Section_Size = Block_Size - Literals_Section_Header - Literals_Section_Content.

3.1.1.3.2.1. Sequences_Section_Header

This header consists of two items:

- Number_of_Sequences
- Symbol_Compression_Modes

Number_of_Sequences is a variable size field using between 1 and 3 bytes. If the first byte is "byte0":

- if (byte0 == 0): there are no sequences. The sequence section stops here. Decompressed content is defined entirely as Literals_Section content. The FSE tables used in Repeat_Mode are not updated.
- if (byte0 < 128): Number_of_Sequences = byte0. Uses 1 byte.
- if (byte0 < 255): Number_of_Sequences = ((byte0 - 128) << 8) + byte1. Uses 2 bytes.
- if (byte0 == 255): Number_of_Sequences = byte1 + (byte2 << 8) + 0x7F00. Uses 3 bytes.

Symbol_Compression_Modes is a single byte, defining the compression mode of each symbol type.

Bit Number	Field Name
7-6	Literal_Lengths_Mode
5-4	Offsets_Mode
3-2	Match_Lengths_Mode
1-0	Reserved

Table 14: Symbol_Compression_Modes

The last field, Reserved, must be all zeroes.

Literals_Lengths_Mode, Offsets_Mode, and Match_Lengths_Mode define the Compression_Mode of literals length codes, offset codes, and match length codes, respectively. They follow the same enumeration:

Value	Compression_Mode
0	Predefined_Mode
1	RLE_Mode
2	FSE_Compressed_Mode
3	Repeat_Mode

Table 15: Literals_Lengths_Mode, Offsets_Mode, and Match_Lengths_Mode

Predefined_Mode: A predefined FSE (see Section 4.1) distribution table is used, as defined in Section 3.1.1.3.2.2. No distribution table will be present.

RLE_Mode: The table description consists of a single byte, which contains the symbol's value. This symbol will be used for all sequences.

FSE_Compressed_Mode: Standard FSE compression. A distribution table will be present. The format of this distribution table is described in Section 4.1.1. Note that the maximum allowed accuracy log for literals length code and match length code tables is 9, and the maximum accuracy log for the offset code table is 8. This mode must not be used when only one symbol is present; RLE_Mode should be used instead (although any other mode will work).

Repeat_Mode: The table used in the previous Compressed_Block with Number_Of_Sequences > 0 will be used again, or if this is the first block, the table in the dictionary will be used. Note that this includes RLE_Mode, so if Repeat_Mode follows RLE_Mode, the same symbol will be repeated. It also includes Predefined_Mode, in which case Repeat_Mode will have the same outcome as Predefined_Mode. No distribution table will be present. If this mode is used without any previous sequence table in the frame (or dictionary; see Section 5) to repeat, this should be treated as corruption.

3.1.1.3.2.1.1. Sequence Codes for Lengths and Offsets

Each symbol is a code in its own context, which specifies Baseline and Number_of_Bits to add. Codes are FSE compressed and interleaved with raw additional bits in the same bitstream.

Literals length codes are values ranging from 0 to 35, inclusive. They define lengths from 0 to 131071 bytes. The literals length is equal to the decoded Baseline plus the result of reading Number_of_Bits bits from the bitstream, as a little-endian value.

Literals_Length_Code	Baseline	Number_of_Bits
0-15	length	0
16	16	1
17	18	1
18	20	1
19	22	1
20	24	2
21	28	2
22	32	3
23	40	3
24	48	4
25	64	6
26	128	7
27	256	8
28	512	9
29	1024	10
30	2048	11
31	4096	12
32	8192	13
33	16384	14
34	32768	15
35	65536	16

Table 16: Literals Length Codes

Match length codes are values ranging from 0 to 52, inclusive. They define lengths from 3 to 131074 bytes. The match length is equal to the decoded Baseline plus the result of reading Number_of_Bits bits from the bitstream, as a little-endian value.

Match_Length_Code	Baseline	Number_of_Bits
0-31	Match_Length_Code + 3	0
32	35	1
33	37	1
34	39	1
35	41	1
36	43	2
37	47	2
38	51	3
39	59	3
40	67	4
41	83	4
42	99	5
43	131	7
44	259	8
45	515	9
46	1027	10
47	2051	11
48	4099	12
49	8195	13
50	16387	14
51	32771	15
52	65539	16

Table 17: Match Length Codes

Offset codes are values ranging from 0 to N.

A decoder is free to limit its maximum supported value for N. Support for values of at least 22 is recommended. At the time of this writing, the reference decoder supports a maximum N value of 31.

An offset code is also the number of additional bits to read in little-endian fashion and can be translated into an Offset_Value using the following formulas:

```
   Offset_Value = (1 << offsetCode) + readNBits(offsetCode);
   if (Offset_Value > 3) Offset = Offset_Value - 3;
```

This means that maximum Offset_Value is $(2^{N+1}) - 1$, supporting back-reference distance up to $(2^{N+1}) - 4$, but it is limited by the maximum back-reference distance (see Section 3.1.1.1.2).

Offset_Value from 1 to 3 are special: they define "repeat codes". This is described in more detail in Section 3.1.1.5.

3.1.1.3.2.1.2. Decoding Sequences

FSE bitstreams are read in reverse of the direction they are written. In zstd, the compressor writes bits forward into a block, and the decompressor must read the bitstream backwards.

To find the start of the bitstream, it is therefore necessary to know the offset of the last byte of the block, which can be found by counting Block_Size bytes after the block header.

After writing the last bit containing information, the compressor writes a single 1 bit and then fills the rest of the byte with zero bits. The last byte of the compressed bitstream cannot be zero for that reason.

When decompressing, the last byte containing the padding is the first byte to read. The decompressor needs to skip the up to 7 bits of 0-padding as well as the first 1 bit that occurs. Afterwards, the useful part of the bitstream begins.

FSE decoding requires a 'state' to be carried from symbol to symbol. For more explanation on FSE decoding, see Section 4.1.

For sequence decoding, a separate state keeps track of each literals length, offset, and match length code. Some FSE primitives are also used. For more details on the operation of these primitives, see Section 4.1.

The bitstream starts with initial FSE state values, each using the required number of bits in their respective accuracy, decoded previously from their normalized distribution. It starts with Literals_Length_State, followed by Offset_State, and finally Match_Length_State.

Note that all values are read backward, so the 'start' of the bitstream is at the highest position in memory, immediately before the last 1 bit for padding.

After decoding the starting states, a single sequence is decoded Number_Of_Sequences times. These sequences are decoded in order from first to last. Since the compressor writes the bitstream in the forward direction, this means the compressor must encode the sequences starting with the last one and ending with the first.

For each of the symbol types, the FSE state can be used to determine the appropriate code. The code then defines the Baseline and Number_of_Bits to read for each type. The description of the codes for how to determine these values can be found in Section 3.1.1.3.2.1.

Decoding starts by reading the Number_of_Bits required to decode offset. It does the same for Match_Length and then for Literals_Length. This sequence is then used for Sequence Execution (see Section 3.1.1.4).

If it is not the last sequence in the block, the next operation is to update states. Using the rules precalculated in the decoding tables, Literals_Length_State is updated, followed by Match_Length_State, and then Offset_State. See Section 4.1 for details on how to update states from the bitstream.

This operation will be repeated Number_of_Sequences times. At the end, the bitstream shall be entirely consumed; otherwise, the bitstream is considered corrupted.

3.1.1.3.2.2. Default Distributions

If Predefined_Mode is selected for a symbol type, its FSE decoding table is generated from a predefined distribution table defined here. For details on how to convert this distribution into a decoding table, see Section 4.1.

3.1.1.3.2.2.1. Literals Length Codes

The decoding table uses an accuracy log of 6 bits (64 states).

```
   short literalsLength_defaultDistribution[36] =
     { 4, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 2, 1, 1, 1, 1, 1,
       -1,-1,-1,-1
     };
```

3.1.1.3.2.2.2. Match Length Codes

The decoding table uses an accuracy log of 6 bits (64 states).

```
   short matchLengths_defaultDistribution[53] =
     { 1, 4, 3, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,-1,-1,
       -1,-1,-1,-1,-1
     };
```

3.1.1.3.2.2.3. Offset Codes

The decoding table uses an accuracy log of 5 bits (32 states) and supports a maximum N value of 28, allowing offset values up to 536,870,908.

If any sequence in the compressed block requires a larger offset than this, it's not possible to use the default distribution to represent it.

```
   short offsetCodes_defaultDistribution[29] =
     { 1, 1, 1, 1, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1,-1,-1,-1,-1,-1
     };
```

3.1.1.4. Sequence Execution

Once literals and sequences have been decoded, they are combined to produce the decoded content of a block.

Each sequence consists of a tuple of (literals_length, offset_value, match_length), decoded as described in the Sequences_Section (Section 3.1.1.3.2). To execute a sequence, first copy literals_length bytes from the decoded literals to the output.

Then, match_length bytes are copied from previous decoded data. The offset to copy from is determined by offset_value:

- if Offset_Value > 3, then the offset is Offset_Value - 3;
- if Offset_Value is from 1-3, the offset is a special repeat offset value. See Section 3.1.1.5 for how the offset is determined in this case.

The offset is defined as from the current position (after copying the literals), so an offset of 6 and a match length of 3 means that 3 bytes should be copied from 6 bytes back. Note that all offsets leading to previously decoded data must be smaller than Window_Size defined in Frame_Header_Descriptor (Section 3.1.1.1.1).

3.1.1.5. Repeat Offsets

As seen above, the first three values define a repeated offset; we will call them Repeated_Offset1, Repeated_Offset2, and Repeated_Offset3. They are sorted in recency order, with Repeated_Offset1 meaning "most recent one".

If offset_value is 1, then the offset used is Repeated_Offset1, etc.

There is one exception: when the current sequence's literals_length is 0, repeated offsets are shifted by 1, so an offset_value of 1 means Repeated_Offset2, an offset_value of 2 means Repeated_Offset3, and an offset_value of 3 means Repeated_Offset1 - 1_byte.

For the first block, the starting offset history is populated with the following values: Repeated_Offset1 (1), Repeated_Offset2 (4), and Repeated_Offset3 (8), unless a dictionary is used, in which case they come from the dictionary.

Then each block gets its starting offset history from the ending values of the most recent Compressed_Block. Note that blocks that are not Compressed_Block are skipped; they do not contribute to offset history.

During the execution of the sequences of a Compressed_Block, the Repeated_Offsets' values are kept up to date, so that they always represent the three most recently used offsets. In order to achieve that, they are updated after executing each sequence in the following way:

When the sequence's offset_value does not refer to one of the Repeated_Offsets -- when it has value greater than 3, or when it has value 3 and the sequence's literals_length is zero -- the Repeated_Offsets' values are shifted back one, and Repeated_Offset1 takes on the value of the offset that was just used.

Otherwise, when the sequence's offset_value refers to one of the Repeated_Offsets -- when it has value 1 or 2, or when it has value 3 and the sequence's literals_length is non-zero -- the Repeated_Offsets are reordered, so that Repeated_Offset1 takes on the value of the used Repeated_Offset, and the existing values are pushed back from the first Repeated_Offset through to the Repeated_Offset selected by the offset_value. This effectively performs a single-stepped wrapping rotation of the values of these offsets, so that their order again reflects the recency of their use.

The following table shows the values of the Repeated_Offsets as a series of sequences are applied to them:

offset_value	literals_length	Repeated_Offset1	Repeated_Offset2	Repeated_Offset3	Comment
		1	4	8	starting values
1114	11	1111	1	4	non-repeat
1	22	1111	1	4	repeat 1; no change
2225	22	2222	1111	1	non-repeat
1114	111	1111	2222	1111	non-repeat
3336	33	3333	1111	2222	non-repeat
2	22	1111	3333	2222	repeat 2; swap 1 & 2
3	33	2222	1111	3333	repeat 3; rotate 3 to 1
1	0	2221	2222	1111	insert resolved offset

offset_ value	literals_ length	Repeated_ Offset1	Repeated_ Offset2	Repeated_ Offset3	Comment
1	0	2222	2221	3333	repeat 2

Table 18: Repeated_Offsets

3.1.2. Skippable Frames

Magic_Number	Frame_Size	User_Data
4 bytes	4 bytes	n bytes

Table 19: Skippable Frames

Skippable frames allow the insertion of user-defined metadata into a flow of concatenated frames.

Skippable frames defined in this specification are compatible with skippable frames in [LZ4].

From a compliant decoder perspective, skippable frames simply need to be skipped, and their content ignored, resuming decoding after the skippable frame.

It should be noted that a skippable frame can be used to watermark a stream of concatenated frames embedding any kind of tracking information (even just a Universally Unique Identifier (UUID)). Users wary of such possibility should scan the stream of concatenated frames in an attempt to detect such frames for analysis or removal.

The fields are:

Magic_Number: 4 bytes, little-endian format. Value: 0x184D2A5?, which means any value from 0x184D2A50 to 0x184D2A5F. All 16 values are valid to identify a skippable frame. This specification does not detail any specific tagging methods for skippable frames.

Frame_Size: This is the size, in bytes, of the following User_Data (without including the magic number nor the size field itself). This field is represented using 4 bytes, little-endian format, unsigned 32 bits. This means User_Data can't be bigger than $(2^{32}-1)$ bytes.

User_Data: This field can be anything. Data will just be skipped by the decoder.

4. Entropy Encoding

Two types of entropy encoding are used by the Zstandard format: FSE and Huffman coding. Huffman is used to compress literals, while FSE is used for all other symbols (Literals_Length_Code, Match_Length_Code, and offset codes) and to compress Huffman headers.

4.1. FSE

FSE, short for Finite State Entropy, is an entropy codec based on [ANS]. FSE encoding/decoding involves a state that is carried over between symbols, so decoding must be done in the opposite direction as encoding. Therefore, all FSE bitstreams are read from end to beginning. Note that the order of the bits in the stream is not reversed; they are simply read in the reverse order from which they were written.

For additional details on FSE, see "FiniteStateEntropy" [FSE].

FSE decoding involves a decoding table that has a power-of-2 size and contains three elements: Symbol, Num_Bits, and Baseline. The base 2 logarithm of the table size is its Accuracy_Log. An FSE state value represents an index in this table.

To obtain the initial state value, consume Accuracy_Log bits from the stream as a little-endian value. The next symbol in the stream is the Symbol indicated in the table for that state. To obtain the next state value, the decoder should consume Num_Bits bits from the stream as a little-endian value and add it to Baseline.

4.1.1. FSE Table Description

To decode FSE streams, it is necessary to construct the decoding table. The Zstandard format encodes FSE table descriptions as described here.

An FSE distribution table describes the probabilities of all symbols from 0 to the last present one (included) on a normalized scale of (1 << Accuracy_Log). Note that there must be two or more symbols with nonzero probability.

A bitstream is read forward, in little-endian fashion. It is not necessary to know its exact size, since the size will be discovered and reported by the decoding process. The bitstream starts by reporting on which scale it operates. If low4bits designates the lowest 4 bits of the first byte, then Accuracy_Log = low4bits + 5.

This is followed by each symbol value, from 0 to the last present one. The number of bits used by each field is variable and depends on:

Remaining probabilities + 1: For example, presuming an Accuracy_Log of 8, and presuming 100 probabilities points have already been distributed, the decoder may read any value from 0 to (256 - 100 + 1) == 157, inclusive. Therefore, it must read log$_2$sup(157) == 8 bits.

Value decoded: Small values use 1 fewer bit. For example, presuming values from 0 to 157, inclusive, are possible, 255 - 157 = 98 values are remaining in an 8-bit field. The first 98 values (hence, from 0 to 97) use only 7 bits, and values from 98 to 157 use 8 bits. This is achieved through the scheme in Table 20:

Value Read	Value Decoded	Bits Used
0 - 97	0 - 97	7
98 - 127	98 - 127	8
128 - 225	0 - 97	7
226 - 255	128 - 157	8

Table 20: Values Decoded

Symbol probabilities are read one by one, in order. The probability is obtained from Value Decoded using the formula P = Value - 1. This means the value 0 becomes the negative probability -1. This is a special probability that means "less than 1". Its effect on the distribution table is described below. For the purpose of calculating total allocated probability points, it counts as 1.

When a symbol has a probability of zero, it is followed by a 2-bit repeat flag. This repeat flag tells how many probabilities of zeroes follow the current one. It provides a number ranging from 0 to 3. If it is a 3, another 2-bit repeat flag follows, and so on.

When the last symbol reaches a cumulated total of (1 << Accuracy_Log), decoding is complete. If the last symbol makes the cumulated total go above (1 << Accuracy_Log), distribution is considered corrupted.

Finally, the decoder can tell how many bytes were used in this process and how many symbols are present. The bitstream consumes a round number of bytes. Any remaining bit within the last byte is simply unused.

The context in which the table is to be used specifies an expected number of symbols. That expected number of symbols never exceeds 256. If the number of symbols decoded is not equal to the expected, the header should be considered corrupt.

The distribution of normalized probabilities is enough to create a unique decoding table. The table has a size of (1 << Accuracy_Log). Each cell describes the symbol decoded and instructions to get the next state.

Symbols are scanned in their natural order for "less than 1" probabilities as described above. Symbols with this probability are being attributed a single cell, starting from the end of the table and retreating. These symbols define a full state reset, reading Accuracy_Log bits.

All remaining symbols are allocated in their natural order. Starting from symbol 0 and table position 0, each symbol gets allocated as many cells as its probability. Cell allocation is spread, not linear; each successor position follows this rule:

```
   position += (tableSize >> 1) + (tableSize >> 3) + 3;
   position &= tableSize - 1;
```

A position is skipped if it is already occupied by a "less than 1" probability symbol. Position does not reset between symbols; it simply iterates through each position in the table, switching to the next symbol when enough states have been allocated to the current one.

The result is a list of state values. Each state will decode the current symbol.

To get the Number_of_Bits and Baseline required for the next state, it is first necessary to sort all states in their natural order. The lower states will need 1 more bit than higher ones. The process is repeated for each symbol.

For example, presuming a symbol has a probability of 5, it receives five state values. States are sorted in natural order. The next power of 2 is 8. The space of probabilities is divided into 8 equal parts. Presuming the Accuracy_Log is 7, this defines 128 states, and each share (divided by 8) is 16 in size. In order to reach 8, 8 - 5 = 3 lowest states will count "double", doubling the number of shares (32 in width), requiring 1 more bit in the process.

Baseline is assigned starting from the higher states using fewer bits, and proceeding naturally, then resuming at the first state, each taking its allocated width from Baseline.

state order	0	1	2	3	4
width	32	32	32	16	16
Number_of_Bits	5	5	5	4	4
range number	2	4	6	0	1
Baseline	32	64	96	0	16
range	32-63	64-95	96-127	0-15	16-31

Table 21: Baseline Assignments

The next state is determined from the current state by reading the required Number_of_Bits and adding the specified Baseline.

See Appendix A for the results of this process that are applied to the default distributions.

4.2. Huffman Coding

Zstandard Huffman-coded streams are read backwards, similar to the FSE bitstreams. Therefore, to find the start of the bitstream, it is necessary to know the offset of the last byte of the Huffman-coded stream.

After writing the last bit containing information, the compressor writes a single 1 bit and then fills the rest of the byte with 0 bits. The last byte of the compressed bitstream cannot be 0 for that reason.

When decompressing, the last byte containing the padding is the first byte to read. The decompressor needs to skip the up to 7 bits of 0-padding as well as the first 1 bit that occurs. Afterwards, the useful part of the bitstream begins.

The bitstream contains Huffman-coded symbols in little-endian order, with the codes defined by the method below.

4.2.1. Huffman Tree Description

Prefix coding represents symbols from an a priori known alphabet by bit sequences (codewords), one codeword 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.

Given an alphabet with known symbol frequencies, the Huffman algorithm allows the construction of an optimal prefix code using the fewest bits of any possible prefix codes for that alphabet.

The prefix code must not exceed a maximum code length. More bits improve accuracy but yield a larger header size and require more memory or more complex decoding operations. This specification limits the maximum code length to 11 bits.

All literal values from zero (included) to the last present one (excluded) are represented by Weight with values from 0 to Max_Number_of_Bits. Transformation from Weight to Number_of_Bits follows this pseudocode:

```
if Weight == 0
  Number_of_Bits = 0
else
  Number_of_Bits = Max_Number_of_Bits + 1 - Weight
```

The last symbol's Weight is deduced from previously decoded ones, by completing to the nearest power of 2. This power of 2 gives Max_Number_of_Bits the depth of the current tree.

For example, presume the following Huffman tree must be described:

Literal Value	Number_of_Bits
0	1
1	2
2	3
3	0
4	4

Literal Value	Number_of_Bits
5	4

Table 22: Huffman Tree

The tree depth is 4, since its longest element uses 4 bits. (The longest elements are those with the smallest frequencies.) Value 5 will not be listed as it can be determined from the values for 0-4, nor will values above 5 as they are all 0. Values from 0 to 4 will be listed using Weight instead of Number_of_Bits. The pseudocode to determine Weight is:

```
if Number_of_Bits == 0
  Weight = 0
else
  Weight = Max_Number_of_Bits + 1 - Number_of_Bits
```

It gives the following series of weights:

Literal Value	Weight
0	4
1	3
2	2
3	0
4	1

Table 23: Weights

The decoder will do the inverse operation: having collected weights of literals from 0 to 4, it knows the last literal, 5, is present with a nonzero Weight. The Weight of 5 can be determined by advancing to the next power of 2. The sum of $2^{(\text{Weight-1})}$ (excluding 0's) is 15. The nearest power of 2 is 16. Therefore, Max_Number_of_Bits = 4 and Weight[5] = 16 - 15 = 1.

4.2.1.1. Huffman Tree Header

This is a single byte value (0-255), which describes how the series of weights is encoded.

headerByte < 128: The series of weights is compressed using FSE (see below). The length of the FSE-compressed series is equal to headerByte (0-127).

headerByte >= 128: This is a direct representation, where each Weight is written directly as a 4-bit field (0-15). They are encoded forward, 2 weights to a byte with the first weight taking the top 4 bits and the second taking the bottom 4; for example, the following operations could be used to read the weights:

```
Weight[0] = (Byte[0] >> 4)
Weight[1] = (Byte[0] & 0xf),
etc.
```

The full representation occupies ceiling(Number_of_Symbols/2) bytes, meaning it uses only full bytes even if Number_of_Symbols is odd. Number_of_Symbols = headerByte - 127. Note that maximum Number_of_Symbols is 255 - 127 = 128. If any literal has a value over 128, raw header mode is not possible, and it is necessary to use FSE compression.

4.2.1.2. FSE Compression of Huffman Weights

In this case, the series of Huffman weights is compressed using FSE compression. It is a single bitstream with two interleaved states, sharing a single distribution table.

To decode an FSE bitstream, it is necessary to know its compressed size. Compressed size is provided by headerByte. It's also necessary to know its maximum possible decompressed size, which is 255, since literal values span from 0 to 255, and the last symbol's Weight is not represented.

An FSE bitstream starts by a header, describing probabilities distribution. It will create a decoding table. For a list of Huffman weights, the maximum accuracy log is 6 bits. For more details, see Section 4.1.1.

The Huffman header compression uses two states, which share the same FSE distribution table. The first state (State1) encodes the even-numbered index symbols, and the second (State2) encodes the odd-numbered index symbols. State1 is initialized first, and then State2, and they take turns decoding a single symbol and updating their state. For more details on these FSE operations, see Section 4.1.

The number of symbols to be decoded is determined by tracking the bitStream overflow condition: if updating state after decoding a symbol would require more bits than remain in the stream, it is assumed that extra bits are zero. Then, symbols for each of the final states are decoded and the process is complete.

4.2.1.3. Conversion from Weights to Huffman Prefix Codes

All present symbols will now have a Weight value. It is possible to transform weights into Number_of_Bits, using this formula:

```
if Weight > 0
    Number_of_Bits = Max_Number_of_Bits + 1 - Weight
else
    Number_of_Bits = 0
```

Symbols are sorted by Weight. Within the same Weight, symbols keep natural sequential order. Symbols with a Weight of zero are removed. Then, starting from the lowest Weight, prefix codes are distributed in sequential order.

For example, assume the following list of weights has been decoded:

Literal	Weight
0	4
1	3
2	2
3	0
4	1
5	1

Table 24: Decoded Weights

Sorting by weight and then the natural sequential order yields the following distribution:

Literal	Weight	Number_Of_Bits	Prefix Codes
3	0	0	N/A
4	1	4	0000
5	1	4	0001
2	2	3	001
1	3	2	01
0	4	1	1

Table 25: Sorting by Weight

4.2.2. Huffman-Coded Streams

Given a Huffman decoding table, it is possible to decode a Huffman-coded stream.

Each bitstream must be read backward, starting from the end and going up to the beginning. Therefore, it is necessary to know the size of each bitstream.

It is also necessary to know exactly which bit is the last. This is detected by a final bit flag: the highest bit of the last byte is a final-bit-flag. Consequently, a last byte of 0 is not possible. And the final-bit-flag itself is not part of the useful bitstream. Hence, the last byte contains between 0 and 7 useful bits.

Starting from the end, it is possible to read the bitstream in a little-endian fashion, keeping track of already used bits. Since the bitstream is encoded in reverse order, starting from the end, read symbols in forward order.

For example, if the literal sequence "0145" was encoded using the above prefix code, it would be encoded (in reverse order) as:

Symbol	Encoding
5	0000
4	0001
1	01
0	1
Padding	00001

Table 26: Literal Sequence "0145"

This results in the following 2-byte bitstream:

```
   00010000 00001101
```

Here is an alternative representation with the symbol codes separated by underscores:

```
   0001_0000 00001_1_01
```

Reading the highest Max_Number_of_Bits bits, it's possible to compare the extracted value to the decoding table, determining the symbol to decode and number of bits to discard.

The process continues reading up to the required number of symbols per stream. If a bitstream is not entirely and exactly consumed, hence reaching exactly its beginning position with all bits consumed, the decoding process is considered faulty.

5. Dictionary Format

Zstandard is compatible with "raw content" dictionaries, free of any format restriction, except that they must be at least 8 bytes. These dictionaries function as if they were just the content part of a formatted dictionary.

However, dictionaries created by "zstd --train" in the reference implementation follow a specific format, described here.

Dictionaries are not included in the compressed content but rather are provided out of band. That is, the Dictionary_ID identifies which should be used, but this specification does not describe the mechanism by which the dictionary is obtained prior to use during compression or decompression.

A dictionary has a size, defined either by a buffer limit or a file size. The general format is:

Magic_Number	Dictionary_ID	Entropy_Tables	Content

Table 27: Dictionary General Format

Magic_Number: 4 bytes ID, value 0xEC30A437, little-endian format.

Dictionary_ID: 4 bytes, stored in little-endian format. Dictionary_ID can be any value, except 0 (which means no Dictionary_ID). It is used by decoders to check if they use the correct dictionary. If the frame is going to be distributed in a private environment, any Dictionary_ID can be used. However, for public distribution of compressed frames, the following ranges are reserved and shall not be used:

low range: <= 32767

high range: >= (2^{31})

Entropy_Tables: Follow the same format as the tables in compressed blocks. See the relevant FSE and Huffman sections for how to decode these tables. They are stored in the following order: Huffman table for literals, FSE table for offsets, FSE table for match lengths, and FSE table for literals lengths. These tables populate the Repeat Stats literals mode and Repeat distribution mode for sequence decoding. It is finally followed by 3 offset values, populating repeat offsets (instead of using {1,4,8}), stored in order, 4 bytes little-endian each, for a total of 12 bytes. Each repeat offset must have a value less than the dictionary size.

Content: The rest of the dictionary is its content. The content acts as a "past" in front of data to be compressed or decompressed, so it can be referenced in sequence commands. As long as the amount of data decoded from this frame is less than or equal to Window_Size, sequence commands may specify offsets longer than the total length of decoded output so far to reference back to the dictionary, even parts of the dictionary with offsets larger than Window_Size. After the total output has surpassed Window_Size, however, this is no longer allowed, and the dictionary is no longer accessible.

6. Use of Dictionaries

Provisioning for use of dictionaries with zstd is being explored. See, for example, [DICT-SEC]. The likely outcome will be a registry of well-tested dictionaries optimized for different use cases and identifiers for each, possibly with a private negotiation mechanism for use of unregistered dictionaries.

To ensure compatibility with the future specification of use of dictionaries with zstd payloads, especially with MIME, content encoded with the media type registered here should not use a dictionary. The exception to this requirement might be a private dictionary negotiation, suggested above, which is not part of this specification.

7. IANA Considerations

IANA has updated two previously existing registrations and made one new registration as described below.

7.1. The 'application/zstd' Media Type

The 'application/zstd' media type identifies a block of data that is compressed using zstd compression. The data is a stream of bytes as described in this document. IANA has added the following to the "Media Types" registry:

Type name: application

Subtype name: zstd

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See Section 8 of RFC 8878.

Interoperability considerations: N/A

Published specification: RFC 8878

Applications which use this media type: anywhere data size is an issue

Fragment identifier considerations: No fragment identifiers are defined for this type.

Additional information:

Deprecated alias names for this type: N/A
Magic number(s): 4 bytes, little-endian format. Value: 0xFD2FB528
File extension(s): zst
Macintosh file type code(s): N/A

Person & email address to contact for further information: Yann Collet <cyan@fb.com>

Intended usage: common

Restrictions on usage: N/A

Author: Murray S. Kucherawy

Change Controller: IETF

Provisional registration: no

For further information: See [ZSTD]

7.2. Content Encoding

IANA has added the following entry to the "HTTP Content Coding Registry" within the "Hypertext Transfer Protocol (HTTP) Parameters" registry:

Name: zstd

Description: A stream of bytes compressed using the Zstandard protocol

Reference: RFC 8878

7.3. Structured Syntax Suffix

IANA has registered the following into the "Structured Syntax Suffix" registry:

Name: Zstandard

+suffix: +zstd

Encoding Considerations: binary

Interoperability Considerations: N/A

Fragment Identifier Considerations: The syntax and semantics of fragment identifiers specified for +zstd should be as specified for 'application/zstd'.

Security Considerations: See Section 8 of RFC 8878.

Contact: Refer to the author for the 'application/zstd' media type.

Author/Change Controller: IETF

7.4. Dictionaries

Work in progress includes development of dictionaries that will optimize compression and decompression of particular types of data. Specification of such dictionaries for public use will necessitate registration of a code point from the reserved range described in Section 3.1.1.1.3 and its association with a specific dictionary.

At present, there are no such dictionaries published for public use, so this document has made no immediate request of IANA to create such a registry.

8. Security Considerations

Any data-compression method involves the reduction of redundancy in the data. Zstandard is no exception, and the usual precautions apply.

One should never compress a message whose content must remain secret with a message generated by a third party. Such a compression can be used to guess the content of the secret message through analysis of entropy reduction. This was demonstrated in the Compression Ratio Info-leak Made Easy (CRIME) attack [CRIME], for example.

A decoder has to demonstrate capabilities to detect and prevent any kind of data tampering in the compressed frame from triggering system faults, such as reading or writing beyond allowed memory ranges. This can be guaranteed by either the implementation language or careful bound checkings. Of particular note is the encoding of Number_of_Sequences values that cause the decoder to read into the block header (and beyond), as well as the indication of a Frame_Content_Size that is smaller than the actual decompressed data, in an attempt to trigger a buffer overflow. It is highly recommended to fuzz-test (i.e., provide invalid, unexpected, or random input and verify safe operation of) decoder implementations to test and harden their capability to detect bad frames and deal with them without any adverse system side effect.

An attacker may provide correctly formed compressed frames with unreasonable memory requirements. A decoder must always control memory requirements and enforce some (system-specific) limits in order to protect memory usage from such scenarios.

Compression can be optimized by training a dictionary on a variety of related content payloads. This dictionary must then be available at the decoder for decompression of the payload to be possible. While this document does not specify how to acquire a dictionary for a given compressed payload, it is worth noting that third-party dictionaries may interact unexpectedly with a decoder, leading to possible memory or other resource-exhaustion attacks. We expect such topics to be discussed in further detail in the Security Considerations section of a forthcoming RFC for dictionary acquisition and transmission, but highlight this issue now out of an abundance of caution.

As discussed in Section 3.1.2, it is possible to store arbitrary user metadata in skippable frames. While such frames are ignored during decompression of the data, they can be used as a watermark to track the path of the compressed payload.

9. References

9.1. Normative References

[ZSTD] "Zstandard", <http://www.zstd.net>.

9.2. Informative References

[ANS] Duda, J., "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding", January 2014, <https://arxiv.org/pdf/1311.2540>.

[CRIME] "CRIME", June 2018, <https://en.wikipedia.org/w/index.php?title=CRIME&oldid=844538656>.

[DICT-SEC] Handte, F., "Security Considerations Regarding Compression Dictionaries", Work in Progress, Internet-Draft, draft-handte-httpbis-dict-sec-00, 29 October 2019, <https://tools.ietf.org/html/draft-handte-httpbis-dict-sec-00>.

[Err5786] RFC Errata, "Erratum ID 5786", RFC 8478, <https://www.rfc-editor.org/errata/eid5786>.

[Err6303] RFC Errata, "Erratum ID 6303", RFC 8478, <https://www.rfc-editor.org/errata/eid6303>.

[FSE] "FiniteStateEntropy", commit 12a533a, July 2020, <https://github.com/Cyan4973/FiniteStateEntropy/>.

[LZ4] "LZ4 Frame Format Description", commit ec735ac, January 2019, <https://github.com/lz4/lz4/blob/master/doc/lz4_Frame_format.md>.

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc1952>.

[XXHASH] "xxHash", <http://www.xxhash.org>.

Appendix A. Decoding Tables for Predefined Codes

This appendix contains FSE decoding tables for the predefined literals length, match length, and offset codes. The tables have been constructed using the algorithm as given above in Section 4.1.1. The tables here can be used as examples to crosscheck that an implementation has built its decoding tables correctly.

A.1. Literals Length Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0
0	0	4	0
1	0	4	16
2	1	5	32
3	3	5	0
4	4	5	0
5	6	5	0
6	7	5	0
7	9	5	0

State	Symbol	Number_Of_Bits	Base
8	10	5	0
9	12	5	0
10	14	6	0
11	16	5	0
12	18	5	0
13	19	5	0
14	21	5	0
15	22	5	0
16	24	5	0
17	25	5	32
18	26	5	0
19	27	6	0
20	29	6	0
21	31	6	0
22	0	4	32
23	1	4	0
24	2	5	0
25	4	5	32
26	5	5	0
27	7	5	32
28	8	5	0
29	10	5	32
30	11	5	0
31	13	6	0

State	Symbol	Number_Of_Bits	Base
32	16	5	32
33	17	5	0
34	19	5	32
35	20	5	0
36	22	5	32
37	23	5	0
38	25	4	0
39	25	4	16
40	26	5	32
41	28	6	0
42	30	6	0
43	0	4	48
44	1	4	16
45	2	5	32
46	3	5	32
47	5	5	32
48	6	5	32
49	8	5	32
50	9	5	32
51	11	5	32
52	12	5	32
53	15	6	0
54	17	5	32
55	18	5	32

State	Symbol	Number_Of_Bits	Base
56	20	5	32
57	21	5	32
58	23	5	32
59	24	5	32
60	35	6	0
61	34	6	0
62	33	6	0
63	32	6	0

Table 28: Literals Length Code

A.2. Match Length Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0
0	0	6	0
1	1	4	0
2	2	5	32
3	3	5	0
4	5	5	0
5	6	5	0
6	8	5	0
7	10	6	0
8	13	6	0
9	16	6	0
10	19	6	0
11	22	6	0

State	Symbol	Number_Of_Bits	Base
12	25	6	0
13	28	6	0
14	31	6	0
15	33	6	0
16	35	6	0
17	37	6	0
18	39	6	0
19	41	6	0
20	43	6	0
21	45	6	0
22	1	4	16
23	2	4	0
24	3	5	32
25	4	5	0
26	6	5	32
27	7	5	0
28	9	6	0
29	12	6	0
30	15	6	0
31	18	6	0
32	21	6	0
33	24	6	0
34	27	6	0
35	30	6	0

State	Symbol	Number_Of_Bits	Base
36	32	6	0
37	34	6	0
38	36	6	0
39	38	6	0
40	40	6	0
41	42	6	0
42	44	6	0
43	1	4	32
44	1	4	48
45	2	4	16
46	4	5	32
47	5	5	32
48	7	5	32
49	8	5	32
50	11	6	0
51	14	6	0
52	17	6	0
53	20	6	0
54	23	6	0
55	26	6	0
56	29	6	0
57	52	6	0
58	51	6	0
59	50	6	0

State	Symbol	Number_Of_Bits	Base
60	49	6	0
61	48	6	0
62	47	6	0
63	46	6	0

Table 29: Match Length Code Table

A.3. Offset Code Table

State	Symbol	Number_Of_Bits	Base
0	0	0	0
0	0	5	0
1	6	4	0
2	9	5	0
3	15	5	0
4	21	5	0
5	3	5	0
6	7	4	0
7	12	5	0
8	18	5	0
9	23	5	0
10	5	5	0
11	8	4	0
12	14	5	0
13	20	5	0
14	2	5	0
15	7	4	16

State	Symbol	Number_Of_Bits	Base
16	11	5	0
17	17	5	0
18	22	5	0
19	4	5	0
20	8	4	16
21	13	5	0
22	19	5	0
23	1	5	0
24	6	4	16
25	10	5	0
26	16	5	0
27	28	5	0
28	27	5	0
29	26	5	0
30	25	5	0
31	24	5	0

Table 30: Offset Code

Appendix B. Changes since RFC 8478

The following are the changes in this document relative to RFC 8478:

- Applied errata [Err5786] and [Err6303].
- Clarified forward compatibility regarding dictionaries.
- Clarified application of Block_Maximum_Size.
- Added structured media type suffix registration.
- Clarified that the content checksum is always 4 bytes.
- Clarified handling of reserved and corrupt inputs.
- Added fragment identifier considerations to the media type registration.

Acknowledgments

zstd was developed by Yann Collet.

Felix Handte and Nick Terrell provided feedback that went into this revision and RFC 8478. RFC 8478 also received contributions from Bobo Bose-Kolanu, Kyle Nekritz, and David Schleimer.

Authors' Addresses

Yann Collet
Facebook
1 Hacker Way
Menlo Park, CA 94025
United States of America
Email: cyan@fb.com

Murray S. Kucherawy (EDITOR)
Facebook
1 Hacker Way
Menlo Park, CA 94025
United States of America
Email: msk@fb.com