[MS-PATCH]:

LZX DELTA Compression and Decompression

Intellectual Property Rights Notice for Open Specifications Documentation

- **Technical Documentation.** Microsoft publishes Open Specifications documentation ("this documentation") for protocols, file formats, data portability, computer languages, and standards support. Additionally, overview documents cover inter-protocol relationships and interactions.
- **Copyrights**. This documentation is covered by Microsoft copyrights. Regardless of any other terms that are contained in the terms of use for the Microsoft website that hosts this documentation, you can make copies of it in order to develop implementations of the technologies that are described in this documentation and can distribute portions of it in your implementations that use these technologies or in your documentation as necessary to properly document the implementation. You can also distribute in your implementation, with or without modification, any schemas, IDLs, or code samples that are included in the documentation. This permission also applies to any documents that are referenced in the Open Specifications documentation.
- **No Trade Secrets**. Microsoft does not claim any trade secret rights in this documentation.
- **Patents**. Microsoft has patents that might cover your implementations of the technologies described in the Open Specifications documentation. Neither this notice nor Microsoft's delivery of this documentation grants any licenses under those patents or any other Microsoft patents. However, a given Open Specifications document might be covered by the Microsoft Open Specifications Promise or the Microsoft Community Promise. If you would prefer a written license, or if the technologies described in this documentation are not covered by the Open Specifications Promise or Community Promise, as applicable, patent licenses are available by contacting iplg@microsoft.com.
- **License Programs**. To see all of the protocols in scope under a specific license program and the associated patents, visit the Patent Map.
- **Trademarks**. The names of companies and products contained in this documentation might be covered by trademarks or similar intellectual property rights. This notice does not grant any licenses under those rights. For a list of Microsoft trademarks, visit www.microsoft.com/trademarks.
- **Fictitious Names**. The example companies, organizations, products, domain names, email addresses, logos, people, places, and events that are depicted in this documentation are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

Reservation of Rights. All other rights are reserved, and this notice does not grant any rights other than as specifically described above, whether by implication, estoppel, or otherwise.

Tools. The Open Specifications documentation does not require the use of Microsoft programming tools or programming environments in order for you to develop an implementation. If you have access to Microsoft programming tools and environments, you are free to take advantage of them. Certain Open Specifications documents are intended for use in conjunction with publicly available standards specifications and network programming art and, as such, assume that the reader either is familiar with the aforementioned material or has immediate access to it.

Support. For questions and support, please contact dochelp@microsoft.com.

Revision Summary

Date	Revision History	Revision Class	Comments
4/4/2008	0.1	New	Initial Availability.
6/27/2008	1.0	Major	Initial Release.
8/6/2008	1.01	Minor	Revised and edited technical content.
9/3/2008	1.02	Minor	Revised and edited technical content.
12/3/2008	1.03	Minor	Updated IP notice.
3/4/2009	1.04	Minor	Revised and edited technical content.
4/10/2009	2.0	Major	Updated technical content and applicable product releases.
7/15/2009	3.0	Major	Revised and edited for technical content.
11/4/2009	3.0.1	Editorial	Revised and edited the technical content.
2/10/2010	3.1.0	Minor	Updated the technical content.
5/5/2010	3.1.1	Editorial	Revised and edited the technical content.
8/4/2010	4.0	Major	Significantly changed the technical content.
11/3/2010	4.0	None	No changes to the meaning, language, or formatting of the technical content.
3/18/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
8/5/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
10/7/2011	4.0	None	No changes to the meaning, language, or formatting of the technical content.
1/20/2012	5.0	Major	Significantly changed the technical content.
4/27/2012	5.0	None	No changes to the meaning, language, or formatting of the technical content.
7/16/2012	5.0	None	No changes to the meaning, language, or formatting of the technical content.
10/8/2012	5.1	Minor	Clarified the meaning of the technical content.
2/11/2013	5.1	None	No changes to the meaning, language, or formatting of the technical content.
7/26/2013	6.0	Major	Significantly changed the technical content.
11/18/2013	6.0	None	No changes to the meaning, language, or formatting of the technical content.
2/10/2014	6.0	None	No changes to the meaning, language, or formatting of the technical content.
4/30/2014	6.1	Minor	Clarified the meaning of the technical content.
7/31/2014	6.1	None	No changes to the meaning, language, or formatting of the

Date	Revision History	Revision Class	Comments
			technical content.
10/30/2014	6.1	None	No changes to the meaning, language, or formatting of the technical content.
3/16/2015	7.0	Major	Significantly changed the technical content.
5/26/2015	7.0	None	No changes to the meaning, language, or formatting of the technical content.
9/14/2015	7.0	None	No changes to the meaning, language, or formatting of the technical content.
6/13/2016	7.0	None	No changes to the meaning, language, or formatting of the technical content.
9/14/2016	7.0	None	No changes to the meaning, language, or formatting of the technical content.
9/19/2017	7.0	None	No changes to the meaning, language, or formatting of the technical content.
7/24/2018	8.0	Major	Significantly changed the technical content.
10/1/2018	9.0	Major	Significantly changed the technical content.
8/18/2020	10.0	Major	Significantly changed the technical content.

Table of Contents

1 Introduction

LZX DELTA Compression and Decompression enables one set of data to be compressed within the context of a reference set of data that is supplied to both the compressor and the decompressor.

Sections 1.7 and 2 of this specification are normative. All other sections and examples in this specification are informative.

1.1 Glossary

This document uses the following terms:

encoding: A process that specifies a Content-Transfer-Encoding for transforming character data from one form to another.

Lempel-Ziv Extended (LZX): An LZ77-based compression engine, as described in [UASDC], that is a universal lossless data compression algorithm. It performs no analysis on the data.

Lempel-Ziv Extended Delta (LZXD): A derivative of the Lempel-Ziv Extended (LZX) format with some modifications to facilitate efficient delta compression. Delta compression is a technique in which one set of data can be compressed within the context of a reference set of data that is supplied both to the compressor and decompressor. Delta compression is commonly used to encode updates to similar existing data sets so that the size of compressed data can be significantly reduced relative to ordinary non-delta compression techniques. Expanding a delta-compressed set of data requires that the exact same reference data be provided during decompression.

little-endian: Multiple-byte values that are byte-ordered with the least significant byte stored in the memory location with the lowest address.

offline address book (OAB): A collection of address lists that are stored in a format that a client can save and use locally.

padding: Bytes that are inserted in a data stream to maintain alignment of the protocol requests on natural boundaries.

path length: The number of edges in the canonical Huffman tree between the top of the tree and the element.

stream: A flow of data from one host to another host, or the data that flows between two hosts.

MAY, SHOULD, MUST, SHOULD NOT, MUST NOT: These terms (in all caps) are used as defined in [RFC2119]. All statements of optional behavior use either MAY, SHOULD, or SHOULD NOT.

1.2 References

Links to a document in the Microsoft Open Specifications library point to the correct section in the most recently published version of the referenced document. However, because individual documents in the library are not updated at the same time, the section numbers in the documents may not match. You can confirm the correct section numbering by checking the Errata.

1.2.1 Normative References

We conduct frequent surveys of the normative references to assure their continued availability. If you have any issue with finding a normative reference, please contact dochelp@microsoft.com. We will assist you in finding the relevant information.

[Cormen] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., "Introduction to Algorithms", 3rd edition, Massachusetts Institute of Technology, 2009, ISBN: 978-0-262-03384-8.

[IEEE1003.1] The Open Group, "IEEE Std 1003.1, 2004 Edition", 2004, http://www.unix.org/version3/ieee_std.html

[MS-DTYP] Microsoft Corporation, "Windows Data Types".

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997, http://www.rfc-editor.org/rfc/rfc2119.txt

[UASDC] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression", May 1977, http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf

1.2.2 Informative References

[MS-OXOAB] Microsoft Corporation, "Offline Address Book (OAB) File Format and Schema".

[MS-OXPROTO] Microsoft Corporation, "Exchange Server Protocols System Overview".

1.3 Overview

Lempel-Ziv Extended Delta (LZXD) compression provides a mechanism for both the compressor and the decompressor to refer to a common reference set of data. It relaxes the constraint that the match offset be constrained to less than the current position in the output **stream**, allowing the match offset to refer to the logically prepended reference data. This relaxed constraint effectively enables the compressed data stream to encode "matches" both from the reference data and from the uncompressed data stream.

1.4 Relationship to Protocols and Other Structures

LZXD (D for Delta) is an **LZX** variant that is modified to facilitate efficient delta compression.

LZX is a compressor that is based on the Lempel-Ziv 1977 (LZ77) sliding window data compression algorithm, as described in [UASDC], that uses static Huffman **encoding** and a sliding window of selectable size. Data symbols are encoded either as an uncompressed symbol or as a logical (offset, length) pair indicating that length symbols shall be copied from a displacement of offset symbols from the current position in the output **stream**. The value of the offset is constrained to be less than the current position in the output stream, up to the size of the sliding window.

The LZXD compression format is used by [MS-OXOAB] to compress data in the **offline address book (OAB)**.

For conceptual background information and overviews of the relationships and interactions between this and other protocols, see [MS-OXPROTO].

1.5 Applicability Statement

LZXD compression is commonly used to encode updates to similar existing data sets so that the size of compressed data can be significantly reduced relative to ordinary compression techniques that do not use the delta between a common reference set of data. One use for this compression format is the compression data in **OAB** version 4 Differential Patch or Compressed OAB Template files.

1.6 Versioning and Localization

None.

1.7 Vendor-Extensible Fields

None.

2 Structures

LZXD compressed data consists of a header that indicates the file translation size, followed by a sequence of compressed blocks. A **stream** of uncompressed input can be output as multiple compressed LZXD blocks to improve compression, because each compressed block contains its own statistical tree structures.

Header	Block	Block	Block	...

Figure 1: The structure of LZXD compressed data

A block can be one of the following types:

- Uncompressed block, as specified in section 2.3.2.1.
- Verbatim block, as specified in section 2.3.2.2.
- Aligned offset, as specified in section 2.3.2.3.

In this document, ranges are specified using interval notation. A range in parenthesis "()" does not include the upper and lower endpoints. A range in brackets "[]" does include the upper and lower endpoints.

2.1 Concepts

2.1.1 Bitstream

An **LZXD** bitstream is encoded as a sequence of aligned 16-bit integers stored in the least-significant-byte to most-significant-byte order, also known as byte-swapped, or **little-endian**, words. Given an input **stream** of bits named a, b, c,..., x, y, z, A, B, C, D, E, F, the output byte stream MUST be as follows:

										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1

Figure 2: An example output byte stream

2.1.2 Window Size

The sliding window size MUST be a power of 2, from 2^17 (128 kilobytes (KB)) up to 2^25 (32 megabytes (MB)). The window size is not stored in the compressed data **stream** and MUST be specified to the decoder before decoding begins. The window size SHOULD be the smallest power of two between 2^17 and 2^25 that is greater than or equal to the sum of the size of the reference data rounded up to a multiple of 32,768 and the size of the subject data.

2.1.3 Reference Data

For delta compression, the reference data is a sequence of bytes given to the compressor before compressing the subject data. The exact same reference data sequence MUST be given to the decompressor before decompression. The reference data sequence is treated as logically prepended to the subject data sequence being compressed or decompressed. During decompression, match offsets are negative displacements from the "current position" in the output **stream**, up to the specified window size. When match offset values exceed the number of bytes already emitted in the uncompressed output stream, they are pointing into the reference data that is logically prepended to the subject data.

Offset	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Value	A	B	C	D	E	F	G	H	I	J	a	b	c	D	E	F	a	b	c	e
	Reference data sequence										Subject data sequence									

Figure 3: Example reference data and subject data

In this example, the reference data is 10 bytes long and consists of the sequence "ABCDEFGHIJ". The data to be compressed, or the subject data, is also 10 bytes long (although the data does not have to be the same length as the reference data) and consists of "abcDEFabce". A valid encoded sequence would consist of the following tokens:

'a', 'b', 'c', (match offset -10, length 3), (match offset -6, length 3), 'e'

The first match offset exceeds the amount of subject data already in the window, pointing instead into the reference data portion. The second match offset does not exceed the amount of subject data in the window and instead refers to a portion of the subject data previously compressed or decompressed.

2.1.4 Repeated Offsets

LZXD compression extends the conventional Lempel-Ziv 1977 sliding window data compression algorithm format, as specified in [UASDC], in several ways, one of which is in the use of repeated offset codes. Three match offset codes, named the repeated offset codes, are reserved to indicate that the current match offset is the same as that of one of the three previous matches, which is not itself a repeated offset.

The three special offset codes are encoded as offset values 0, 1, and 2 (for example, encoding an offset of 0 means "use the most recent nonrepeated match offset"; an offset of 1 means "use the second most recent nonrepeated match offset"; and so on). All remaining encoded offset values are displaced by real offset +2, as is shown in the following table, which prevents matches at offsets WINDOW_SIZE, WINDOW_SIZE-1, and WINDOW_SIZE-2.

Encoded offset	Real offset
0	Most recent real match offset
1	Second most recent match offset
2	Third most recent match offset
3	1 (closest allowable)
4	2
5	3
6	4
7	5
8	6
500	498
X+2	X
WINDOW_SIZE-1 (maximum possible)	WINDOW_SIZE-3

The three most recent real match offsets are kept in a list, the behavior of which is explained as follows:

- Let R0 be defined as the most recent real offset.
- Let R1 be defined as the second most recent offset.
- Let R2 be defined as the third most recent offset.

The list is managed similarly to a least recently used queue, with the exception of the cases when R1 or R2 is output. In these cases, R1 or R2 is simply swapped with R0, which requires fewer operations than a least recently used queue would.

The initial state of R0, R1, R2 is (1, 1, 1).

Match offset X where...	Operation
X≠R0 and X≠R1 and X≠R2	R2←R1 R1←R0 R0←X
X = R0	None
X = R1	swap R0↔R1
X = R2	swap R0↔R2

2.1.5 Match Lengths

The minimum match length (number of bytes) encoded by **LZXD** is 2 bytes, and the maximum match length is 32,768 bytes. However, no match of any length can span a modulo 32-KB boundary in the uncompressed **stream**. Match-length encoding is combined with match-position encoding as described in section 2.6. Match length can be larger than the repeated offset, which means the matched substrings can overlap.

2.1.6 Position Slot

The window size determines the number of window subdivisions, or position slots, as shown in the following table.

Window size	Position slots required
128 KB	34
256 KB	36
512 KB	38
1 MB	42
2 MB	50
4 MB	66
8 MB	98
16 MB	162

Window size	Position slots required
32 MB	290

2.2 Header

2.2.1 Chunk Size

The **LZXD** compressor emits chunks of compressed data. A chunk represents exactly 32 KB of uncompressed data until the last chunk in the **stream**, which can represent less than 32 KB. To ensure that an exact number of input bytes represent an exact number of output bytes for each chunk, after each 32 KB of uncompressed data is represented in the output compressed bitstream, the output bitstream is padded with up to 15 bits of zeros to realign the bitstream on a 16-bit boundary (even byte boundary) for the next 32 KB of data. This results in a compressed chunk of a byte-aligned size. The compressed chunk could be smaller than 32 KB or larger than 32 KB if the data is incompressible when the chunk is not the last one.

The LZXD engine encodes a compressed, chunk-size prefix field preceding each compressed chunk in the compressed byte stream. The compressed, chunk-size prefix field is a byte aligned, **little-endian**, 16-bit field. The chunk prefix chain could be followed in the compressed stream without decompressing any data. The next chunk prefix is at a location computed by the absolute byte offset location of this chunk prefix plus 2 (for the size of the chunk-size prefix field) plus the current chunk size.

2.2.2 E8 Call Translation

E8 call translation is an optional feature that can be used when the data to compress contains x86 instruction sequences. E8 translation operates as a preprocessing stage before compressing each chunk, and the compressed **stream** header contains a bit that indicates whether the decoder shall reverse the translation as a postprocessing step after decompressing each chunk.

The x86 instruction beginning with a byte value of 0xE8 is followed by a 32-bit, **little-endian** relative displacement to the call target. When E8 call translation is enabled, the following preprocessing steps are performed on the uncompressed input before compression (assuming little-endian byte ordering):

Let chunk_offset refer to the total number of uncompressed bytes preceding this chunk.

Let E8_file_size refer to the caller-specified value given to the compressor or decoded from the header of the compressed stream during decompression.

The following example shows how E8 translation is performed for each 32-KB chunk of uncompressed data (or less than 32 KB if last chunk to compress).

```
if (( chunk_offset < 0x40000000 ) && ( chunk_size > 10 ))
   for ( i = 0; i < (chunk_size - 10); i++ )
if ( chunk_byte[ i ] == 0xE8 )
   long current_pointer = chunk_offset + i;
long displacement =    chunk_byte[ i+1 ] |
chunk_byte[ i+2 ] << 8  |
chunk_byte[ i+3 ] << 16 |
chunk_byte[ i+4 ] << 24;
long target = current_pointer + displacement;
if (( target >= 0 ) && ( target < E8_file_size+current_pointer))
if ( target >= E8_file_size )
target = displacement - E8_file_size;
endif
```

```
chunk_byte[ i+1 ] = (byte)( target );
chunk_byte[ i+2 ] = (byte)( target >> 8 );
chunk_byte[ i+3 ] = (byte)( target >> 16 );
chunk_byte[ i+4 ] = (byte)( target >> 24 );
endif
   i += 4;
endif
endfor
endif
```

After decompression, the E8 scanning algorithm is the same. The following example shows how E8 translation reversal is performed.

```
long value =    chunk_byte[ i+1 ]       |
chunk_byte[ i+2 ] << 8  |
chunk_byte[ i+3 ] << 16 |
chunk_byte[ i+4 ] << 24;

if (( value >= -current_pointer ) && ( value < E8_file_size ))
if ( value >= 0 )
displacement = value - current_pointer;
else
displacement = value + E8_file_size;
endif
chunk_byte[ i+1 ] = (byte)( displacement );
chunk_byte[ i+2 ] = (byte)( displacement >> 8 );
chunk_byte[ i+3 ] = (byte)( displacement >> 16 );
chunk_byte[ i+4 ] = (byte)( displacement >> 24 );
endif
```

The first bit in the first chunk in the **LZXD** bitstream (following the 2-byte, chunk-size prefix described in section 2.2.1) indicates the presence or absence of two 16-bit fields immediately following the single bit. If the bit is set, E8 translation is enabled for all the following chunks in the stream using the 32-bit value derived from the two 16-bit fields as the E8_file_size provided to the compressor when E8 translation was enabled. Note that E8_file_size is completely independent of the length of the uncompressed data. E8 call translation is disabled after the 32,768th chunk (after 1 gigabyte (GB) of uncompressed data).

Field	Comments	Size
E8 translation	0-disabled, 1-enabled	1 bit
Translation size high word	Only present if enabled	0 or 16 bits
Translation size low word	Only present if enabled	0 or 16 bits

2.3 Block

2.3.1 Block Header

An **LZXD** block represents a sequence of compressed data that is encoded with the same set of Huffman trees, or a sequence of uncompressed data. There can be one or more LZXD blocks in a compressed **stream**, each with its own set of Huffman trees. Blocks do not have to start or end on a chunk boundary; blocks can span multiple chunks, or a single chunk can contain multiple blocks. The number of chunks is related to the size of the data being compressed, while the number of blocks is related to how well the data is compressed. The **Block Type** field, as specified in section 2.3.1.1, indicates which type of block follows, and the **Block Size** field, as specified in section 2.3.1.2, indicates the number of uncompressed bytes represented by the block. Following the generic block header is a type-specific header that describes the remainder of the block.

Field	Comments	Size
Block Type	See valid values in section 2.3.1.1	3 bits
Block Size most significant bit	Block size is the high 8 bits of 24	8 bits
Block Size byte 2	Block size is the middle 8 bits of 24	8 bits
Block Size least significant bit	Block size is the low 8 bits of 24	8 bits

2.3.1.1 Block Type Field

Each block of compressed data begins with a 3-bit **Block Type** field, followed by the **Block Size** field, as specified in section 2.3.1.2, and then type-specific block data, as specified in section 2.3.2. Of the eight possible values, only three are valid values for the **Block Type** field.

Bits	Value	Meaning
001	1	Verbatim block
010	2	Aligned offset block
011	3	Uncompressed block
other	0, 4-7	Not valid

2.3.1.2 Block Size Field

The **Block Size** field indicates the number of uncompressed bytes that are represented by the block. The maximum value for the **Block Size** field is 2^24-1 (16 MB-1, or 0x00FFFFFF). The **Block Size** field is encoded in the bitstream as three 8-bit fields comprising a 24-bit value, most significant to least significant, immediately following the value of the **Block Type** field.

2.3.2 Block Data

2.3.2.1 Uncompressed Block

Following the generic block header, an uncompressed block begins with 1 to 16 bits of zero **padding** to align the bit buffer on a 16-bit boundary. At this point, the bitstream ends and a byte **stream** begins. Following the zero padding, new 32-bit values for R0, R1, and R2 are output in **little-endian** form, followed by the uncompressed data bytes themselves. Finally, if the uncompressed data length is odd, one extra byte of zero padding is encoded to realign the following bitstream.

Field	Comments	Size
Padding to align following field on 16-bit boundary	Bits have a value of zero	Variable, [1..16] bits

Then, the following fields are encoded directly in the byte stream, not in the bitstream of byte-swapped 16-bit words:

Field	Comments	Size
R0	Least significant to most significant byte (little-endian **DWORD** ([MS-DTYP]))	4 bytes
R1	Least significant to most significant byte (little-endian **DWORD**)	4 bytes
R2	Least significant to most significant byte (little-endian **DWORD**)	4 bytes
Uncompressed raw data bytes	Can use the direct **memcpy** function, as specified in [IEEE1003.1]	[2^24 - 1] bytes
Padding to realign bitstream	Only if uncompressed size is odd	0 or 1 byte

Then the bitstream of byte-swapped 16-bit integers resumes for the next **Block Type** field (if there are subsequent blocks).

The decoded R0, R1, and R2 values are used as initial repeated offset values to decode the subsequent compressed block if present.

2.3.2.2 Verbatim Block

The fields of a verbatim block that follow the generic block header are listed in the following table.

Entry	Comments	Size
Pretree for first 256 elements of main tree	20 elements, 4 bits each	80 bits
Path lengths of first 256 elements of main tree	Encoded using pretree	Variable
Pretree for remainder of main tree	20 elements, 4 bits each	80 bits
Path lengths of remaining elements of main tree	Encoded using pretree	Variable
Pretree for length tree	20 elements, 4 bits each	80 bits
Path lengths of elements in length tree	Encoded using pretree	Variable
Token sequence (matches and literals)	Specified in section 2.6	Variable

2.3.2.3 Aligned Offset Block

An aligned offset block is identical to the verbatim block except for the presence of the aligned offset tree preceding the other trees.

Entry	Comments	Size
Aligned offset tree	8 elements, 3 bits each	24 bits
Pretree for first 256 elements of main tree	20 elements, 4 bits each	80 bits
Path lengths of first 256 elements of main tree	Encoded using pretree	Variable
Pretree for remainder of main tree	20 elements, 4 bits each	80 bits
Path lengths of remaining elements of main tree	Encoded using pretree	Variable

Entry	Comments	Size
Pretree for length tree	20 elements, 4 bits each	80 bits
Path lengths of elements in length tree	Encoded using pretree	Variable
Token sequence (matches and literals)	Specified in section 2.6	Variable

2.4 Huffman Trees

LZXD compression uses canonical Huffman tree structures to represent elements. Huffman trees, as specified in [Cormen], are well known in data compression and are not described here. Because an LZXD decoder uses only the **path lengths** of the Huffman tree to reconstruct the identical tree, the following constraints are made on the tree structure.

For any two elements with the same path length, the lower-numbered element MUST be farther left on the tree than the higher-numbered element. An alternative way of stating this constraint is that lower-numbered elements MUST have lower path traversal values; for example, 0010 (left-left-right-left) is lower than 0011 (left-left-right-right).

For each level, starting at the deepest level of the tree and then moving upward, leaf nodes MUST start as far left as possible. An alternative way of stating this constraint is that if any tree node has children, all tree nodes to the right of it with the same path length MUST also have children.

A non-empty Huffman tree MUST contain at least two elements. In the case where all but one tree element has zero frequency, the resulting tree MUST minimally consist of two Huffman codes, "0" and "1".

LZXD compression uses several Huffman tree structures. The main tree comprises 256 elements that correspond to all possible 8-bit characters, plus 8 * **NUM_POSITION_SLOTS** elements that correspond to matches. The **NUM_POSITION_SLOTS** elements refer to the position slots required, as specified in section 2.1.6. The value of the **NUM_POSITION_SLOTS** elements depends on the specified window size as described in section 2.1.6. The length tree comprises 249 elements. Other trees, such as the aligned offset tree (comprising 8 elements), and the pretrees (comprising 20 elements each), have a smaller role.

2.5 Encoding the Trees and Pretrees

Because all trees used in **LZXD** compression are created in the form of a canonical Huffman tree, the **path length** of each element in the tree is sufficient to reconstruct the original tree. The main tree and the length tree are each encoded using the method described here. However, the main tree is encoded in two components as if it were two separate trees, the first tree corresponding to the first 256 tree elements (uncompressed symbols), and the second tree corresponding to the remaining elements (matches).

Because trees are output several times during compression of large amounts of data (multiple blocks), LZXD optimizes compression by encoding only the delta path lengths between the current and previous trees. In the case of the very first such tree, the delta is calculated against a tree in which all elements have a zero path length.

Each tree element can have a path length of [0, 16], where a zero path length indicates that the element has a zero frequency and is not present in the tree. Tree elements are output in sequential order starting with the first element. Elements can be encoded in one of two ways: if several consecutive elements have the same path length, run-length encoding is employed; otherwise, the element is output by encoding the difference between the current path length and the previous path length of the tree, mod 17. To represent a canonical Huffman tree, specify the path lengths of each of the elements in the tree. The following table specifies how to interpret a code.

Code	Operation
0 to 16	Len[*x*] = (prev_len[*x*] - code + 17) mod 17
17	*Zeros* = getbits(4) Len[*x*] = 0 for next (4 + *Zeros*) elements
18	*Zeros* = getbits(5) Len[*x*] = 0 for next (20 + *Zeros*) elements
19	*Same* = getbits(1) Decode new code *Value* = (prev_len[*x*] - code + 17) mod 17 Len[*x*] = Value for next (4 + Same) elements

Codes 17, 18, and 19 are used to represent consecutive elements that have the same path length. *Zeros*, *Same*, and *Value* are variables created for the purpose of this sample code, and **getbits(*n*)** is a function that fetches the next *n* bits from the bitstream. "Decode new code" is used to parse the next code from the bitstream, which has a value range of [0, 16].

Each of the 17 possible values of (len[*x*] - prev_len[*x*]) mod 17, plus three additional codes used for run-length encoding, are not output directly as 5-bit numbers but are instead encoded via a Huffman tree called the pretree. The pretree is generated dynamically according to the frequencies of the 20 allowable tree codes. The structure of the pretree is encoded in a total of 80 bits by using 4 bits to output the path length of each of the 20 pretree elements. Once again, a zero path length indicates a zero-frequency element.

Code	Operation
Length of tree code 0	4 bits
Length of tree code 1	4 bits
Length of tree code 2	4 bits
...	...
Length of tree code 18	4 bits
Length of tree code 19	4 bits

The "real" tree is then encoded using the pretree Huffman codes.

2.6 Compressed Token Sequence

The compressed token sequence (bitstream) contains the Huffman-encoded matches and literals using the Huffman trees specified in the block header. Decompression continues until the number of decompressed bytes corresponds exactly to the number of uncompressed bytes indicated in the block header.

The representation of an unmatched literal character in the output is simply the appropriate element index [0..255] from the main Huffman tree.

The representation of a match in the output involves several transformations, as shown in the following diagram. At the top of the diagram are the match length [2..257] and the match offset [0..WINDOW_SIZE-3]. The match offset and match length are split into subcomponents and encoded separately. For matches of length [258..32768], the token indicates match length 257, and then the additional value of the **Extra Length** field is encoded in the bitstream following the other match subcomponent fields.

The match subcomponents are shown in the following figure.

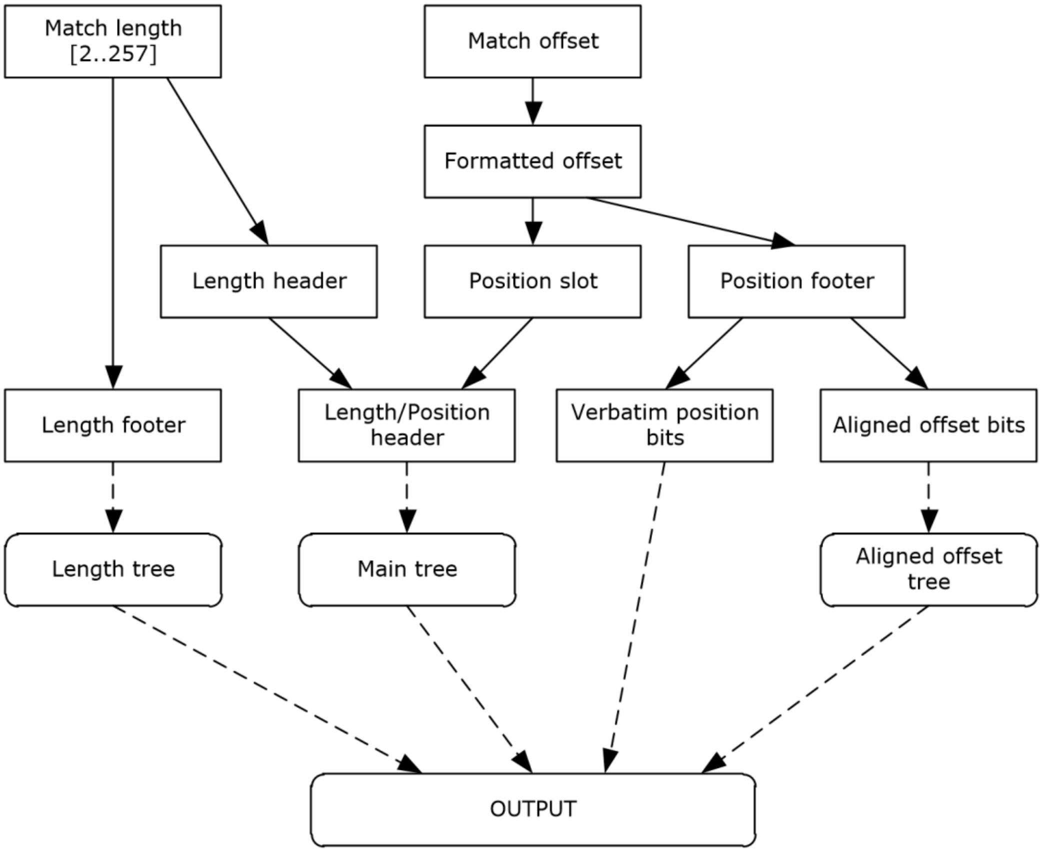

Figure 4: Match encoding subcomponents

2.6.1 Converting Match Offset into Formatted Offset Values

The match offset, range [1..WINDOW_SIZE-3], is converted into a formatted offset by determining whether the offset can be encoded as a repeated offset, as shown in the following pseudocode. It is acceptable not to encode a match as a repeated offset even if it is possible to do so.

```
if offset == R0 then
   formatted offset ←  0
else if offset == R1 then
   formatted offset ←  1
else if offset == R2 then
   formatted offset ←  2
else
   formatted offset ←  offset + 2
endif
```

2.6.2 Converting Formatted Offset into Position Slot and Position Footer Values

The formatted offset is subdivided into a position slot and a position footer. The position slot defines the most significant bits of the formatted offset in the form of a base position as shown in the following table. The position footer defines the remaining least significant bits of the formatted offset. As the following table shows, the number of bits dedicated to the position footer grows as the formatted offset becomes larger, meaning that each position slot addresses a larger and larger range.

The number of position slots available depends on the window size. The number of bits of position footer for each position slot is fixed and is shown in the following table.

Position slot number	Base position	Footer bits	Range of base position and position footer (formatted offset)
0 (R0)	0	0	0
1 (R1)	1	0	1
2 (R2)	2	0	2
3 (offset 1)	3	0	3
4 (offset 2..3)	4	1	4-5
5 (offset 4..5)	6	1	6-7
6 (offset 6..9)	8	2	8-11
7 (..etc..)	12	2	12-15
8	16	3	16-23
9	24	3	24-31
10	32	4	32-47
11	48	4	48-63
12	64	5	64-95
13	96	5	96-127
14	128	6	128-191
15	192	6	192-255
16	256	7	256-383
17	384	7	384-511
18	512	8	512-767
19	768	8	768-1023
20	1024	9	1024-1535
21	1536	9	1536-2047
22	2048	10	2048-3071
23	3072	10	3072-4095
24	4096	11	4096-6143

Position slot number	Base position	Footer bits	Range of base position and position footer (formatted offset)
25	6144	11	6144-8191
26	8192	12	8192-12287
27	12288	12	12288-16383
28	16384	13	16384-24575
29	24576	13	24576-32767
30	32768	14	32768-49151
31	49152	14	49152-65535
32	65536	15	65536-98303
33	98304	15	98304-131071
34	131072	16	131072-196607
35	196608	16	196608-262143
36	262144	17	262144-393215
37	393216	17	393216-524287
38	524288	17	524288-655359
39	655360	17	655360-786431
40	786432	17	786432-917503
41	917504	17	917504-1048575
42	1048576	17	1048576-1179647
..etc..	..etc..	17 (all)	..etc..
288	33292288	17	33292288-33423359
289	33423360	17	33423360-33554431

The following pseudocode demonstrates how to determine the position slot and the position footer.

```
position_slot ← calculate_the position_slot from the formatted_offset
position_footer_bits ← determine the number of footer bits from the position slot value
if position_footer_bits > 0
    position_footer ← formatted_offset & ((2^position_footer_bits)-1)
else
    position_footer ← null
```

2.6.3 Converting Position Footer into Verbatim Bits or Aligned Offset Bits

The position footer can be further subdivided into verbatim bits and aligned offset bits if the current value of the **Block Type** field is 010 (aligned offset), as specified in section 2.3.1.1. If the current block is not an aligned offset block, there are no aligned offset bits, and the verbatim bits are the position footer.

If aligned offsets are used, the lower 3 bits of the position footer are the aligned offset bits, while the remaining portion of the position footer is the verbatim bits. In the case where fewer than 3 bits are in the position footer (for example, formatted offset is <= 15), it is not possible to take the "lower 3 bits of the position footer", and therefore, there are no aligned offset bits and the verbatim bits and the position footer are the same.

In situations where it is determined that there is a relatively larger number of position footers with identical lower 3 bits, the aligned offset block could be used to reduce the number of bits required to represent the position footer component in the match encoding.

The verbatim block could be used when the lower 3 bits of the position footer are relatively evenly distributed.

The following is a pseudocode example of splitting the position footer into verbatim bits and aligned offset.

```
if block_type is aligned_offset_block then
   if formatted_offset <= 15 then
      verbatim_bits ← position_footer
      aligned_offset ← null
   else
      aligned_offset ← position_footer
      verbatim_bits ← position_footer >> 3
   endif
else
   verbatim_bits ← position_footer
   aligned_offset ← null
endif
```

2.6.4 Converting Match Length into Length Header and Length Footer Values

The match length is converted into a length header and a length footer. The length header can have one of eight possible values, with a range of [0, 7], indicating a match of length 2, 3, 4, 5, 6, 7, 8, or a length greater than 8. If the match length is 8 or less, there is no length footer. Otherwise, the value of the length footer is equal to the match length minus 9. The following is a pseudocode example of obtaining the length header and footer.

```
if match_length <= 8
   length_header ← match_length-2
   length_footer ← null
else
   length_header ← 7
   length_footer ← match_length-9
endif
```

Match length	Length header	Length footer value
2	0	None
3	1	None
4	2	None
5	3	None
6	4	None
7	5	None
8	6	None

Match length	Length header	Length footer value
9	7	0
10	7	1
…	…	…
256	7	247
257 or larger	7	248

2.6.5 Converting Length Header and Position Slot into Length/Position Header Values

The length/position header is the stage that correlates the match position with the match length (using only the most significant bits) and is created by combining the length header and the position slot, as follows:

```
len_pos_header ←(position_slot << 3) + length_header
```

This operation creates a unique value for every combination of match length 2, 3, 4, 5, 6, 7, 8 with every possible position slot. The remaining match lengths greater than 8 are all lumped together and, as a group, are correlated with every possible position slot.

2.6.6 Extra Length Field

If the match length is 257 or larger, the encoded match length token (or match length, as specified in section 2.6) value is 257, and an encoded **Extra Length** field follows the other match encoding components, as specified in section 2.6.7, in the bitstream.

Prefix (in binary)	Number of bits to decode	Base value to add to decoded value
0	8	257
10	10	257 + 256
110	12	257 + 256 + 1024
111	15	257

If the encoded match length token is equal to 257, it indicates the length of the match is >= 257. If this is the case, the **Extra Length** field is after the other match encoding components in the bitstream. If the prefix of the **Extra Length** field is 0, the match length is the decoded value of the next 8 bits plus 257. If the prefix is 10, the match length is the decoded value of the next 10 bits plus 257 plus 256. If the prefix is 110, the match length is the decoded value of the next 12 bits plus 257 plus 256 plus 1024. If the prefix is 111, the match length is the decoded value of the next 15 bits plus 257.

2.6.7 Encoding a Match

The match is finally output as part of the compressed bitstream in up to five components, in the following order:

1. Main tree element at index (len_pos_header + 256).

2. If length_footer != null, the output length tree element is length_footer.
3. If verbatim_bits != null, the output is verbatim_bits.
4. If aligned_offset_bits != null, the output element is aligned_offset from the aligned offset tree.
5. If the match length is 257 or larger, the output consists of the prefix and value of the **Extra Length** field (section 2.6.6).

2.6.8 Encoding a Literal

A literal byte that is not part of a match is encoded simply as a main tree element index with a range of [0, 255] corresponding to the value of the literal byte.

2.7 Decoding Matches and Literals (Aligned and Verbatim Blocks)

Decoding is performed by first decoding an element from the main tree and then, if the item is a match, determining which additional components are required to decode to reconstruct the match. The following is a pseudocode example of decoding a match or an uncompressed character.

```
main_element = main_tree.decode_element()


/* Check if it is a literal character. */
if (main_element < 256 )


/* It is a literal, so copy the literal to output. */
window[ curpos ] ← (byte) main_element
curpos ← curpos + 1

/* Decode the match. For a match, there are two components, offset and length. */
else
length_header ← (main_element - 256) & 7

if (length_header == 7)

/* Length of the footer. */
match_length ← length_tree.decode_element() + 7 + 2
else
match_length ← length_header + 2 /* no length footer */

/* Decoding a match length (if a match length < 257). */
endif

position_slot ← (main_element - 256) >> 3

/* Check for repeated offsets (positions 0,1,2). */
if (position_slot == 0)
match_offset ← R0
else if (position_slot == 1)
match_offset ← R1
swap(R0 ↔ R1)
else if (position_slot == 2)
match_offset ← R2
swap(R0 ↔ R2)

/* Not a repeated offset. */
else
offset_bits ← footer_bits[ position_slot ]

if (block_type == aligned_offset_block)

/* This means there are some aligned bits. */
if (offset_bits > 3)
```

```
verbatim_bits ← (readbits(offset_bits-3)) << 3
aligned_bits  ← aligned_offset_tree.decode_element();
else /* 0, 1, or 2 verbatim bits */
verbatim_bits ← readbits(offset_bits)
aligned_bits  ← 0
endif

formatted_offset ← base_position[ position_slot ]
+ verbatim_bits + aligned_bits

/* Block_type is a verbatim_block. */
else
verbatim_bits ← readbits(offset_bits)
formatted_offset ← base_position[ position_slot ] + verbatim_bits
endif

/* Decoding a match offset. */
match_offset ← formatted_offset – 2

/* Update repeated offset least recently used queue. */
R2 ← R1
R1 ← R0
R0 ← match_offset

endif

/* Check for extra length. */

if (match_length == 257)
if (readbits( 1 ) != 0)
if (readbits( 1 ) != 0)
if (readbits( 1 ) != 0)
extra_len = readbits( 15 )
else
extra_len = readbits( 12 ) + 1024 + 256
endif
else
extra_len = readbits( 10 ) + 256
endif
else
extra_len = readbits( 8 )

/* Decode the extra length. */
endif

/* Get the match length (if match length >= 257). */
match_length ← 257 + extra_len

endif

/* Get match length and offset. Perform copy and paste work. */
for (i = 0; i < match_length; i++)
window[curpos + i] ← window[curpos + i – match_offset]

curpos ← curpos + match_length

endif
```

3 Structure Examples

The **LZXD** bitstream is to be interpreted as a sequence of aligned 16-bit integers stored in the order least significant byte to most significant byte (**little-endian** words).

The only exception is the uncompressed data bytes stored in the uncompressed block interpreted as a sequence of bytes. The following example is a sample encoding sequence of a simple 3-byte text input "abc" encoded with a **Block Type** field value of 3 (uncompressed block).

Bits to decode	Value of decoded bits	Interpretation
16	0x0014	Chunk size: 20 bytes
1	0	E8 translation:disabled
3	3 (binary 011)	**Block Type**: uncompressed
24	0x000003	**Block Size**: 3 bytes
4	binary 0000	Padding to word-align following
4 bytes	0x00000001 (little-endian **DWORD** ([MS-DTYP]))	R0: 1
4 bytes	0x00000001 (little-endian **DWORD**)	R1: 1
4 bytes	0x00000001 (little-endian **DWORD**)	R2: 1
3 bytes	0x61, 0x62, 0x63	Uncompressed bytes: "abc"
1 byte	0x00	Padding to restore word alignment

This is the raw hexadecimal compressed byte sequence of the encoded fields:

```
14 00 00 30 30 00 01 00 00 00 01 00 00 00 01 00 00 00 61 62 63 00
```

4 Security

4.1 Security Considerations for Implementers

None.

4.2 Index of Security Parameters

None.

5 Appendix A: Product Behavior

The information in this specification is applicable to the following Microsoft products or supplemental software. References to product versions include updates to those products.

- Microsoft Exchange Server 2003
- Microsoft Exchange Server 2007
- Microsoft Exchange Server 2010
- Microsoft Exchange Server 2013
- Microsoft Exchange Server 2016
- Microsoft Office Outlook 2003
- Microsoft Office Outlook 2007
- Microsoft Outlook 2010
- Microsoft Outlook 2013
- Microsoft Outlook 2016
- Microsoft Exchange Server 2019
- Microsoft Outlook 2019

Exceptions, if any, are noted in this section. If an update version, service pack or Knowledge Base (KB) number appears with a product name, the behavior changed in that update. The new behavior also applies to subsequent updates unless otherwise specified. If a product edition appears with the product version, behavior is different in that product edition.

Unless otherwise specified, any statement of optional behavior in this specification that is prescribed using the terms "SHOULD" or "SHOULD NOT" implies product behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term "MAY" implies that the product does not follow the prescription.

6 Change Tracking

This section identifies changes that were made to this document since the last release. Changes are classified as Major, Minor, or None.

The revision class **Major** means that the technical content in the document was significantly revised. Major changes affect protocol interoperability or implementation. Examples of major changes are:

- A document revision that incorporates changes to interoperability requirements.
- A document revision that captures changes to protocol functionality.

The revision class **Minor** means that the meaning of the technical content was clarified. Minor changes do not affect protocol interoperability or implementation. Examples of minor changes are updates to clarify ambiguity at the sentence, paragraph, or table level.

The revision class **None** means that no new technical changes were introduced. Minor editorial and formatting changes may have been made, but the relevant technical content is identical to the last released version.

The changes made to this document are listed in the following table. For more information, please contact dochelp@microsoft.com.

Section	**Description**	**Revision class**
2.3.2.1 Uncompressed Block	Updated size data for clarity.	Minor
2.7 Decoding Matches and Literals (Aligned and Verbatim Blocks)	Corrected psuedocode if statement to (offset bits > 3).	Major

7 Index