[MS-PATCH]: LZX DELTA Compression and Decompression

Revision Summary			
Author	Date	Version	Comments
Microsoft Corporation	April 4, 2008	0.1	Initial Availability.
Microsoft Corporation	June 27, 2008	1.0	Initial Release.
Microsoft Corporation	August 6, 2008	1.01	Revised and edited technical content.
Microsoft Corporation	September 3, 2008	1.02	Revised and edited technical content.
Microsoft Corporation	December 3, 2008	1.03	Updated IP notice.
Microsoft	March 4,	1.04	Revised and edited technical content.

Corporation	2009		

Table of Contents

1 Introduction

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.

1.1 Glossary

The following term is defined in [MS-OXGLOS]:

little-endian

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

1.2 References

1.2.1 Normative References

[LZ77] Lempel, A., and Ziv, J., "A Universal Algorithm for Sequential Data Compression", IEEE Transactions On Information Theory, Vol. IT-23, No. 3, May 1977, http://www.cs.duke.edu/courses/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf.

[MS-CAB] Microsoft Corporation, "Cabinet File Format", June 2008.

[MS-MCI] Microsoft Corporation, "MCI Compression and Decompression", June 2008.

[MS-OXGLOS] Microsoft Corporation, "Exchange Server Protocols Master Glossary", June 2008.

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

1.2.2 Informative References

None.

1.3 Structure Overview

LZX is an LZ77-based Microsoft compression engine described in the Microsoft Cabinet SDK. LZXD (D for Delta) is a derivative of the Microsoft Cabinet LZX format with some modifications to facilitate efficient delta compression.

1.4 Relationship to Protocols and Other Structures

For more information about data compression formats, see [MS-CAB] and [MS-MCI].

1.5 Applicability Statement

None.

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	…

A block can be one of following types:

- Uncompressed
- Aligned offset
- Verbatim

The structure of these blocks is specified in sections 2.15.1, 2.15.2, and 2.15.3.

2.1 LZ77

LZ77 refers to the well-known Lempel-Ziv 1977 sliding window data compression algorithm, as specified in [LZ77].

2.2 LZX

LZX is an LZ77-based compressor that uses static Huffman encoding and a sliding window of selectable size. LZX is most commonly known as part of the Microsoft Cabinet compression format, as specified in [MS-CAB]. 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 offset is constrained to be less than the current position in the output stream, up to the size of the sliding window.

2.3 LZXD

LZXD is an LZX variant modified to facilitate efficient delta-compression. LZXD provides a mechanism for both compressor and decompressor to refer to a common reference set of data, and relaxes the constraint that match offset be constrained to less than the current position in the output stream, allowing match offset to refer to the logically prepended reference data. This effectively enables the compressed data stream to encode "matches" both from the reference data and from the uncompressed data stream.

2.4 Bitstream

An LZXD Bitstream is encoded as a sequence of aligned 16-bit integers stored in the order least-significant-byte most-significant-byte, 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 (with byte boundaries highlighted) would be as follows:

i	j	k	l	m	n	o	p	a	b	c	d	e	f	g	h	y	z	A	B	C	D	E	F	q	r	s	t	u	v	w	x

2.5 Window Size

The sliding window size MUST be a power of 2, from 2^{17} (128 KB) up to 2^{25} (32 MB). The window size is not stored in the compressed data stream, and MUST be specified to the decoder before decoding begins. The preferred window size is 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 multiple of 32,768 and the size of the subject data.

2.6 Reference Data

For delta compression, the reference data is a sequence of bytes given to the compressor prior to compressing the subject data. The exact same reference data sequence MUST be given to the decompressor prior to 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 simply 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**									

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.

2.7 Huffman Trees

LZXD uses canonical Huffman tree structures to represent elements. Huffman trees 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 further 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, then all tree nodes to the left 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 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. NUM_POSITIONS_SLOTS refers to the position slots required, as specified in section 2.8. The value of **NUM_POSITION_SLOTS** depends on the specified window size as described in section 2.8. The Length Tree comprises 249 elements. Other trees, such as the Aligned Offset Tree (comprising 8 elements), and the Pre-Trees (comprising 20 elements each), have a smaller role.

2.8 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

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
32 MB	290

2.9 Repeated Offsets

LZXD extends the conventional LZ77 format 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 non-repeated match offset," an offset of 1 means "use the second most recent non-repeated 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 an LRU (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 would an LRU queue.

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

Match offset X where...	Operation
$X \neq R0$ and $X \neq R1$ and $X \neq R2$	$R2 \leftarrow R1$ $R1 \leftarrow R0$ $R0 \leftarrow X$
$X = R0$	None
$X = R1$	swap $R0 \Leftrightarrow R1$
$X = R2$	swap $R0 \Leftrightarrow R2$

2.10 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 2.15.5.

2.11 E8 Call Translation

E8 Call Translation is an optional feature that is sometimes used when the data to compress contains x86 instruction sequences. E8 Translation operates as a pre-processing stage prior to compressing each chunk, and the compressed stream header contains a bit that indicates whether the decoder shall reverse the translation as a post-processing 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 pre-processing step is performed on the uncompressed input prior to 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.
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, but the translation reversal is:

```
            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 below) indicates the presence or absence of two 16-bit fields immediately following the single bit. If the bit is set, E8 translation is enabled 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 always disabled after the 32,768th chunk (after 1 GB of uncompressed data).

Field	Comments	Size
E8 translation	0-disabled, 1-enabled	1 bit

Field	Comments	Size
Translation size high word	Only present if enabled	0 or 16 bits
Translation size low word	Only present if enabled	0 or 16 bits

2.12 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. In order 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 re-align 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 significantly smaller than 32 KB or possibly larger than 32 KB if the data is incompressible.

The LZXD engine encodes a byte-aligned **little-endian** 16-bit compressed chunk size prefix field preceding each compressed chunk in the compressed byte stream. 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 absolute byte offset location of this chunk prefix plus 2 (for the size of the chunk size prefix field) plus the current chunk size.

2.13 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 indicates which type of block follows, and the **Block Size** field indicates the number of uncompressed bytes represented by the block. Following the generic Block Header, there is a type-specific header that describes the remainder of the block.

Field	Comments	Size
Block Type	See valid values in section 2.14	3 bits
Block Size MSB	Block size high 8 bits of 24	8 bits
Block Size byte 2	Block size middle 8 bits of 24	8 bits
Block Size LSB	Block size low 8 bits of 24	8 bits

2.14 Block Type

Each block of compressed data begins with a 3-bit field indicating the block type, followed by the **Block Size** and then type-specific **Block Data**. Of the eight possible values, only three are valid types.

Bits	Value	Meaning
001	1	Verbatim block
010	2	Aligned offset block
011	3	Uncompressed block
other	0, 4-7	Invalid

2.15 Block Size

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

2.15.1 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 re-align the following bitstream.

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

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

R0	LSB to MSB (little endian dword)	4 bytes
R1	LSB to MSB (little endian dword)	4 bytes
R2	LSB to MSB (little endian dword)	4 bytes
Uncompressed raw data bytes	Can use direct memcpy	1...2^{24}-1 bytes
Padding to re-align 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.

2.15.2 Verbatim Block

A verbatim block consists of the following fields following the generic Block Header:

Entry	Comments	Size
Pre-tree 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 pre-tree	Variable
Pre-tree for remainder of main tree	20 elements, 4 bits each	80 bits
Path lengths of remaining elements of main tree	Encoded using pre-tree	Variable
Pre-tree for length tree	20 elements, 4 bits each	80 bits
Path lengths of elements in length tree	Encoded using pre-tree	Variable
Token sequence (matches and literals)	Described later	Variable

2.15.3 Aligned Offset Block

An aligned offset block consists of the following, the only difference from Verbatim header being the existence of the Aligned Offset Tree preceding the other trees.

Entry	Comments	Size
Aligned offset tree	8 elements, 3 bits each	24 bits
Pre-tree 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 pre-tree	Variable
Pre-tree for remainder of main tree	20 elements, 4 bits each	80 bits
Path lengths of remaining elements of main tree	Encoded using pre-tree	Variable
Pre-tree for length tree	20 elements, 4 bits each	80 bits
Path lengths of elements in length tree	Encoded using pre-tree	Variable
Token sequence (matches and literals)	Described later	Variable

2.15.4 Encoding the Trees and Pre-Trees

Because all trees used in LZXD 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), LZX 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 from 0 to 16 (inclusive) 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, then 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
0-16	Len[x] = (prev_len[x] + code) mod 17

17	Zeroes = getbits(4)
	Len[x] = 0 for next (4 + Zeroes) elements

18	Zeroes = getbits(5)
	Len[x] = 0 for next (20 + Zeroes) elements

19	Same = getbits(1)
	Decode new Code
	Value = (prev_len[x] + Code) 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. Zeroes, 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 will have a value 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 *pre-tree*. The pre-tree is generated dynamically according to the frequencies of the 20 allowable tree codes. The structure of the pre-tree is encoded in a total of 80 bits by using 4 bits to output the path length of each of the 20 pre-tree elements. Once again, a zero path length indicates a zero frequency element.

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 pre-tree Huffman codes.

2.15.5 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-4). The match offset and match length are split into sub-components and encoded separately. For matches of length 257..32768, the token indicates match length 257 and then there is an additional Extra Length value encoded in the bitstream following the other **Match** subcomponent fields.

Figure 1 shows the match subcomponents.

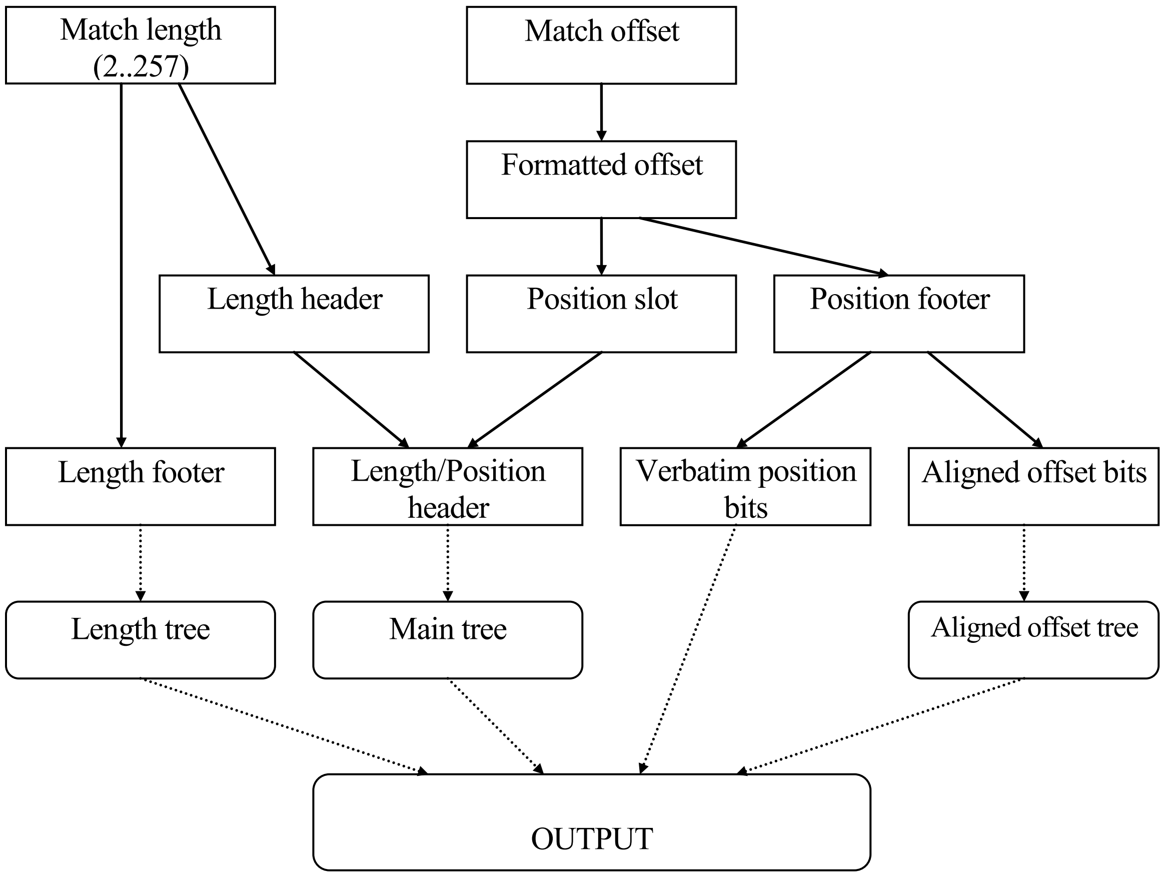

Figure 1: Diagram of match encoding subcomponents

2.15.6 Converting Match Offset into Formatted Offset Values

The match offset, range 1… (WINDOW_SIZE-4), 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 to not 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
```

2.15.7 Converting Formatted Offset into Position Slot and Position Footer Values

The formatted offset is subdivided into a position slot and 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 table on the following page. 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	Base plus position footer range
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
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

Position slot number	Base position	Footer bits	Base plus position footer range
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

2.15.8 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 block type is "aligned offset". 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 there are less than 3 bits 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 are a relatively larger number of Position Footers with identical lower 3 bits, Aligned Offset Block could be used to reduce the number of bits required to represent the Position Footer component in the match encoding.

Verbatim block could be used when the lower 3 bits of the Position Footer are relatively evenly distributed.

The following is pseudocode for 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
```

2.15.9 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, from 0...7 (inclusive), 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 pseudocode for 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
```

The following table shows some examples of conversions of some match lengths to header and footer values.

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
9	7	0
10	7	1
50	7	41
256	7	247
257 or larger	7	248

2.15.10 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.

2.16 Extra Length

If the match length is 257 or larger, the encoded match length token (or match length, as specified in section 2.15.5) value is 257, and an encoded Extra Length field follows the other match encoding components, as specified in section 2.16.1, 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 length of the match is >= 257. If this is the case, look for the Extra Length field after the other match encoding components in the bitstream. Then look at the prefix of the Extra Length field. If the prefix is 0, decode the next 8 bits and add 257 to get the match length. If the prefix is 10, decode the next 10 bits and add 257 +256 to the decoded value to get the match length. If the prefix is 110, decode the next 12 bits and add 257 +256 + 1024 to the decoded value to get the match length. If the prefix is 111, decode the next 15 bits and add 257 to the decoded value to get the match length.

2.16.1 Encoding a Match

The match is finally output in up to five components, in the following order:

1. Main Tree element at index (len_pos_header + 256).
2. If length_footer != null, then Length Tree element length_footer.
3. If verbatim_bits != null, then output verbatim_bits.
4. If aligned_offset_bits != null, then output element aligned_offset from the aligned offset tree.
5. If match length 257 or larger, output appropriate Extra Length prefix and value.

2.17 *Encoding a Literal*

A literal byte that is not part of a match is encoded simply as a Main Tree element index 0..256 corresponding to the value of the literal byte.

2.17.1 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 pseudocode for decoding a match or an uncompressed character.

```
main_element = main_tree.decode_element()

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

   window[ curpos ] ← (byte) main_element
   curpos ← curpos + 1

else /* is a match */

   length_header ← (main_element - 256) & 7

   if (length_header == 7)
      match_length ← length_tree.decode_element() + 7 + 2
   else
      match_length ← length_header + 2 /* no length footer */
   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)
   else /* not a repeated offset */
      offset_bits ← footer_bits[ position_slot ]

      if (block_type == aligned_offset_block)
         if (offset_bits >= 3) /* this means there are some aligned
                bits */
            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

         else /* block_type == verbatim_block */
            verbatim_bits ← readbits(offset_bits)
            formatted_offset ← base_position[ position_slot ] +
                    verbatim_bits
         endif

         match_offset ← formatted_offset - 2

         /* update repeated offset LRU 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 )
         endif

         match_length ← 257 + extra_len

      endif

      /* copy match data */
      for (i = 0; i < match_length; i++)
         window[curpos + i] ← window[curpos + i - match_offset]

      curpos ← curpos + match_length

   endif
```

3 Structure Examples

The following is an example of a sample encoding sequence of a simple 3-byte text input "abc" encoded as uncompressed block type.

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)	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 Considerations

None.

5 Appendix A: Office/Exchange Behavior

The information in this specification is applicable to the following versions of Office/Exchange:

- Microsoft Office 2003
- Microsoft Exchange Server 2003
- Microsoft Office 2007
- Microsoft Exchange Server 2007

Exceptions, if any, are noted below. Unless otherwise specified, any statement of optional behavior in this specification prescribed using the terms SHOULD or SHOULD NOT implies Office/Exchange behavior in accordance with the SHOULD or SHOULD NOT prescription. Unless otherwise specified, the term MAY implies Office/Exchange does not follow the prescription.

Index