Microsoft LZX Data Compression Format

Abstract

This document describes the Microsoft LZX data compression format, as used in Microsoft cabinet files. This information may be used to create or extract from Microsoft cabinet files which utilize LZX data compression. The format of the surrounding cabinet file is described in other documents.

Table of Contents

Introduction

This document is a design specification for the format of LZX compressed data used in the LZX compression mode of Microsoft's CAB file format. The purpose of this document is to allow anyone to encode or decode LZX compressed data. This document describes only the format of the output –it does not provide any specific algorithms for match location, tree generation, etc.

Before proceeding with the design specification itself, a few important concepts are described in the following pages.

Concepts

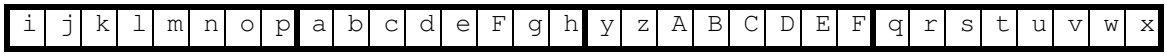
LZ77

LZX is an LZ77 based compressor that uses static Huffman encoding and a sliding window of selectable size. Data symbols are encoded either as an uncompressed symbol, or as an (offset, length) pair indicating that *length* symbols should 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 size of the sliding window.

Bitstream

An LZX bitstream is a sequence of 16 bit integers stored in the order least-significant-byte most-significant-byte. 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 shown below.

Output byte stream



Window size

The window size must be a power of 2, from 2^{15} to 2^{21}. The window size is not stored in the compressed data stream, and must instead be passed to the decoder before decoding begins.

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

Window size / Position slot table

Window size	Position slots required
32K	30
64K	32
128K	34
256K	36
512K	38
1 MB	40
2 MB	42

Trees

LZX uses canonical Huffman tree structures to represent elements. Huffman trees are well known in data compression and are not described here. Since an LZX decoder uses only the path lengths of the Huffman tree to reconstruct the identical tree, the following constraints are made on the tree structure:

1. 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).

2. For each level, starting at the deepest level of the tree and then moving upwards, 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.

3. Zero length Huffman codes are not permitted, therefore a tree must contain at least 2 elements. In the case where all tree elements are zero frequency, or all but one tree element is zero frequency, the resulting tree must consist of the two Huffman codes "0" and "1". In the latter case, constraint #1 still applies.

LZX uses several Huffman tree structures. The most important tree is the *main* tree, which comprises 256 elements corresponding to all possible ASCII characters, plus 8 * NUM_POSITION_SLOTS (see above) elements corresponding to matches. The second most important tree is the *length* tree, which 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.

Repeated offsets

LZX 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 (i.e. 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", etc.). All remaining offset values are displaced by +3, as is shown in the table below, which prevents matches at offsets WINDOW_SIZE, WINDOW_SIZE-1, and WINDOW_SIZE-2.

Correlation between encoded offset and real offset

Encoded offset	Real offset
0	Most recent non-repeated match offset
1	Second most recent non-repeated match offset
2	Third most recent non-repeated 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)*	WIDOW_SIZE-3

The three most recent non-repeated match offsets are kept in a list, the behavior of which explained below:

Let R0 be defined as the most recent non-repeated offset
Let R1 be defined as the second most recent non-repeated offset
Let R2 be defined as the third most recent non-repeated 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, which are fairly uncommon, R1 or R2 is simply swapped with R0, which requires fewer operations than would an LRU queue. The compression penalty from doing so is essentially zero and it removes a small computational overhead from the decoder.

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

Management of the repeated offsets list

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 $R0 \Leftrightarrow R2$

Constants

The following named constants are used frequently in this document:

MIN_MATCH	Smallest allowable match length	2
MAX_MATCH	Largest allowable match length	257
NUM_CHARS	Number of uncompressed character types	256
WINDOW_SIZE	Window size	Varies
NUM_POSITION_SLOTS	Number of window subdivisions	Dependent upon WINDOW_SIZE
MAIN_TREE_ELEMENTS	Number of elements in main tree	NUM_CHARS + NUM_POSITION_SLOTS*8
NUM_SECONDARY_LENGTHS	Number of elements in length tree	249

LZX compressed data format

LZX compressed data consists of a header indicating the file translation size (which is described later), followed by a sequence of compressed blocks. A stream of uncompressed input may be output as multiple compressed LZX blocks to improve compression, since each compressed block contains its own statistical tree structures.

Header	Block	Block	Block	…

Cabinet block size

The cabinet file format requires that for any particular CFDATA block, the indicated number of compressed input bytes must represent exactly the indicated number of uncompressed output bytes. Furthermore, each CFDATA block must represent 32768 uncompressed bytes, with the exception of the last CFDATA block in a folder, which may represent less than 32768 uncompressed bytes.

The LZX block size is independent of the CFDATA block size; an LZX block can represent 200,000 uncompressed bytes, for example. In order to ensure that an exact number of input bytes represent an exact number of output bytes, after each 32768^{th} uncompressed byte is represented, the output bit buffer is byte aligned on a 16-bit boundary by outputting 0-15 zero bits. The bit buffer is flushed in an identical manner after the final CFDATA block in a folder. Furthermore, the compressor may not emit any matches that span a 32768-byte boundary in the input (for example, at position 65528 in the input, the compressor cannot emit a match with a length of 50; the maximum allowable match length at this point would be 6).

One additional constraint is that, for any given CFDATA block, the compressed size of a CFDATA block may not occupy more than 32768+6144 bytes (i.e. 32K of uncompressed input may not grow by more than 6K when compressed).

Header structure

The header consists of either a zero bit indicating no encoder preprocessing, or a one bit followed by a *file translation size*, a value which is used in encoder preprocessing.

0		
1	Most significant 16 bits of file translation size	Least significant 16 bits of file translation size

Encoder preprocessing

The encoder may optionally perform a preprocessing stage on all CFDATA input blocks (size <= 32K) which improves compression on 32-bit Intel 80x86 code. The translation is performed before the data is passed to the compressor, and therefore an appropriate reverse translation must be performed on the output of the decompressor. A bit indicating whether preprocessing was used is stored in the compression header (see above).

The preprocessing stage translates 80x86 CALL instructions, which begin with the E8 (hex) opcode, to use absolute offsets instead of relative offsets.

Preprocessing is disabled after the 32768^{th} CAB input frame in a folder (where a CAB input frame is 32768 bytes) in order to avoid signed/unsigned arithmetic complexity. This change can obviously occur only when a folder represents at least 1 gigabyte of uncompressed data.

CALL byte sequence (E8 followed by 32 bit offset)

E8 r_0 r_1 r_2 r_3

Performing the relative-to-absolute conversion

relative_offset $\leftarrow r_0 + r_1*2^8 + r_2*2^{16} + r_3*2^{24}$
new_value $\leftarrow$ conversion_function(current_location, relative_offset)
$a_0 \leftarrow$ bits 0-7 of new_value
$a_1 \leftarrow$ bits 8-15 of new_value
$a_2 \leftarrow$ bits 16-23 of new_value
$a_3 \leftarrow$ bits 24-31 of new_value

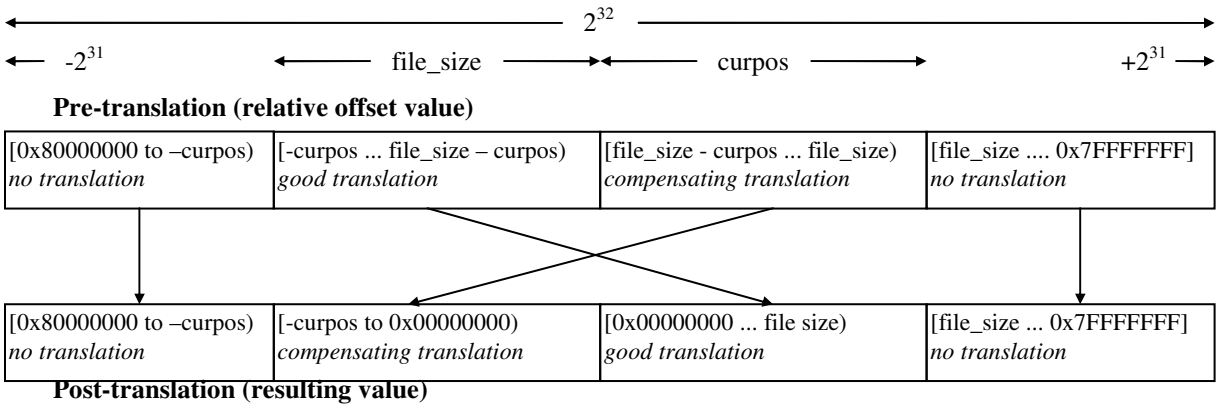
Translated CALL byte sequence

E8 a_0 a_1 a_2 a_3

The diagram below illustrates the relative-to-absolute conversion function, where *curpos* is the current offset within all uncompressed data seen in the current cabinet folder, and *file_size* is the file translation size from the compression header (*file_size* is unrelated to the size of the actual file being decompressed).

The translation is performed “in place” on the input data without using extra codes to indicate whether a translation occurred (i.e. there is a direct mapping from a 32-bit value to a 32-bit value), therefore there is a one-to-one correlation between pre- and post- translated values.

Offset translation diagram



From the diagram one can see that values in the range of *0x80000000 (-2^{31})* to *-curpos*, and *file_size* to *0x7FFFFFFF (+2^{31})* are left unchanged. The translation algorithm operates as follows on an input block of size *input_size*, where 0 <= *input_size* <= 32768. No translation may be performed on the last 6 bytes of the input block

```
if (input_size < 6)
       return /* don't perform translation if < 6 input bytes */

for (i = 0; i < input_size; i++)

       if (input_data[i] == 0xE8)
              if (i >= input_size-6)
                     break;
              endif

              … perform translation illustrated above …
       endif
```

Block structure

Each block of compressed data begins with a 3 bit header describing the block type, followed by the block itself. The allowable block types are:

0	Undefined
1	Verbatim block
2	Aligned offset block
3	Uncompressed block
4-7	Undefined

Uncompressed block format

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 *bytestream* begins. The data that follows is encoded as bytes for performance. Following the zero padding, new values for R0, R1, and R2 are output in little-endian form, followed by the uncompressed data bytes themselves.

1-16 bits	4 bytes	4 bytes	4 bytes	n bytes
zero padding	R0 (LSB first)	R1 (LSB first)	R2 (LSB first)	Uncompressed data

Verbatim block

A verbatim block consists of the following:

Entry	Comments	Size
Number of uncompressed bytes accounted for in this block	Range of $1...2^{24}$	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
Compressed literals	Described later	Variable

Aligned offset block

An aligned offset block consists of the following:

Entry	Comments	Size
Number of uncompressed bytes accounted for in this block	Range of $1...2^{24}$	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
Aligned offset tree	8 elements, 3 bits each	24 bits
Compressed literals	Described later	Variable

The aligned offset tree comprises only 8 elements, each of which is encoded as a 3 bit path length. Since the size of this tree is so small, no additional compression is performed on it.

Encoding the trees and pre-trees

Since all trees used in LZX 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 below. 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).

Since trees are output several times during compression of large amounts of data, LZX optimises 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 may 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 may 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. These output methods are described below:

Tree codes

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

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.

Pre-tree

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.

Compressed literals

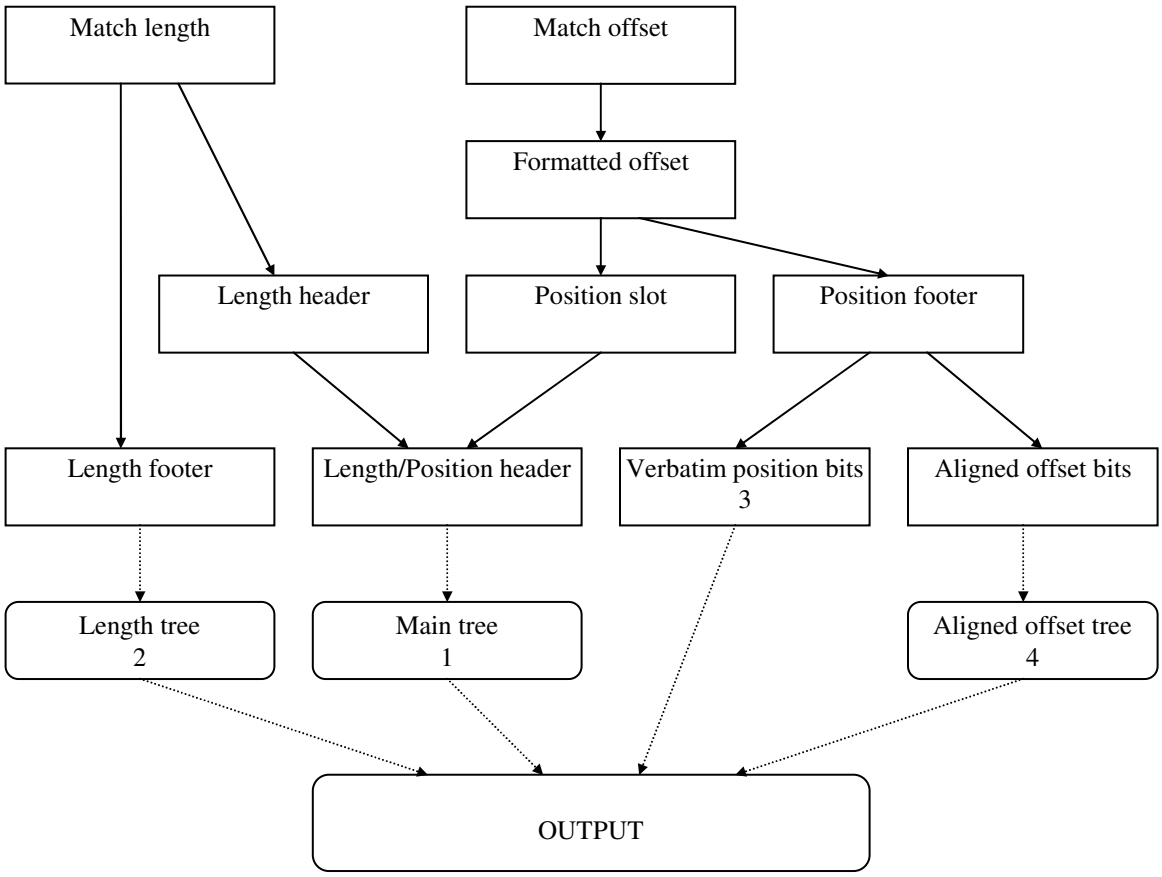
The compressed literals that make up the bulk of either a verbatim block or an aligned offset block immediately follow the tree data (as shown in the diagram for each block type). These literals, which comprise matches and unmatched characters, will, when decompressed, correspond to exactly the number of uncompressed bytes indicated in the block header.

The representation of an unmatched character in the output is simply the appropriate element 0…(NUM_CHARS-1) Huffman-encoded using the main 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 (MIN_MATCH…MAX_MATCH) and the match offset (0…WINDOW_SIZE-4). The match offset and match length are split into sub-components and encoded separately.

As mentioned previously, in order to remain compatible with the cabinet file format, the compressor may not emit any matches that span a 32768-byte boundary in the input.

Diagram of match sub-components



Match offset ⇒ Formatted offset

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 below. It is acceptable to not encode a match as a repeated offset even if it is possible to do so.

Converting a match offset to a formatted offset

```
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
```

Formatted offset ⇒ Position slot, Position footer

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 table on the following page. The position footer defines the remaining least significant bits of the formatted offset. As the 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 position slot table for the maximum window size of 2 megabytes, is shown in the table below.

The position slot table

Position slot number	Base position	Number of position footer bits	Range of base position and position footer
0	0	0	0
1	1	0	1
2	2	0	2
3	3	0	3
4	4	1	4-5
5	6	1	6-7
6	8	2	8-11
7	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
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
43	1179648	17	1179648-1310719
44	1310720	17	1310720-1441791
45	1441792	17	1441792-1572863
46	1572864	17	1572864-1703935
47	1703936	17	1703936-1835007
48	1835008	17	1835008-1966079
49	1966080	17	1966080-2097151

In order to determine the position footer, it is first necessary to determine the position slot. Then, a simple lookup can be performed on the position slot to determine the number of bits, B, in the position footer. The B least significant bits of the formatted offset are the position footer. Pseudocode for obtaining the position slot and position footer are shown below, as is the lookup array (named *extra_bits*).

The extra_bits table

n (position slot)	extra_bits[n] (number of position footer bits)
0	0
1	0
2	0
3	0
4	1
5	1
6	2
7	2
8	3
9	3
10	4
11	4
12	5
13	5
14	6
15	6
16	7
17	7
18	8
19	8
20	9
21	9
22	10
23	10
24	11
25	11
26	12
27	12
28	13
29	13
30	14
31	14
32	15
33	15
34	16
35	16
36-49	17

Calculating the position slot and position footer

```
position_slot ← calculate_position_slot(formatted_offset)
position_footer_bits ← extra_bits[ position_slot ]
if position_footer_bits > 0
       position_footer ← formatted_offset & ((2^position_footer_bits)-1)
else
       position_footer ← null
```

Position footer ⇒ Verbatim bits, Aligned offset bits

The position footer may be further subdivided into verbatim bits and aligned offset bits if the current block uses aligned offsets. If the current block is not an aligned offset block then there are no aligned offset bits, and the verbatim bits are the position footer.

If aligned offsets are used, then the lower 3 bits of the position footer are the aligned offset bits, while the remaining portion of the position footer are the verbatim bits. In the case where there are less than 3 bits in the position footer (i.e. 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.

Pseudocode for splitting position footer into verbatim bits and aligned offset

```
if block_type = 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
```

Match length ⇒ Length header, Length footer

The match length is converted into a length header and a length footer. The length header may 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, then there is no length footer. Otherwise the value of the length footer is equal to the match length minus 9.

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
```

Example conversions of some match lengths to header and footer values

Match length	Length header	Length footer value
2 (MIN_MATCH)	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
257 (MAX_MATCH)	7	248

Length header, Position slot ⇒ Length/Position header

The Length/Position header is the stage which 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 shown below:

```
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.

Encoding a match

The match is finally output in up to four components, as follows:

1. Output element (len_pos_header + NUM_CHARS) from the main tree
2. If length_footer != null, then output element length_footer from the length tree
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

Decoding a match or an uncompressed character

Decoding is performed by first decoding an element using the main tree and then, if the item is a match, determining which additional components are necessary to reconstruct the match. Pseudocode for decoding a match or an uncompressed character is shown below:

```
main_element = main_tree.decode_element()

if (main_element < NUM_CHARS) /* is an uncompressed character */

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

else /* is a match */

        length_header ← (main_element - NUM_CHARS) & NUM_PRIMARY_LENGTHS

        if (length_header == NUM_PRIMARY_LENGTHS)
                match_length ← length_tree.decode_element() +
                        NUM_PRIMARY_LENGTHS + MIN_MATCH
        else
                match_length ← length_header + MIN_MATCH /* no length footer */
        endif

        position_slot ← (main_element - NUM_CHARS) >> 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 */
                extra ← extra_bits[ position_slot ]

                if (block_type == aligned_offset_block)
                        if (extra > 3) /* this means there are some aligned bits */
                                verbatim_bits ← (readbits(extra-3)) << 3
                                aligned_bits  ← aligned_offset_tree.decode_element();
                        else if (extra > 0) /* just some verbatim bits */
                                verbatim_bits ← readbits(extra)
                                aligned_bits  ← 0
                        else /* no verbatim bits */
                                verbatim_bits ← 0
                                aligned_bits  ← 0
                        endif

                        formatted_offset ← base_position[ position_slot ] +
                                                verbatim_bits + aligned_bits
                else /* block_type == verbatim_block */
                        if (extra > 0) /* if there are any extra bits */
                                verbatim_bits ← readbits(extra)
                        else
                                verbatim_bits ← 0
                        endif

                        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

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

        curpos ← curpos + match_length
endif
```