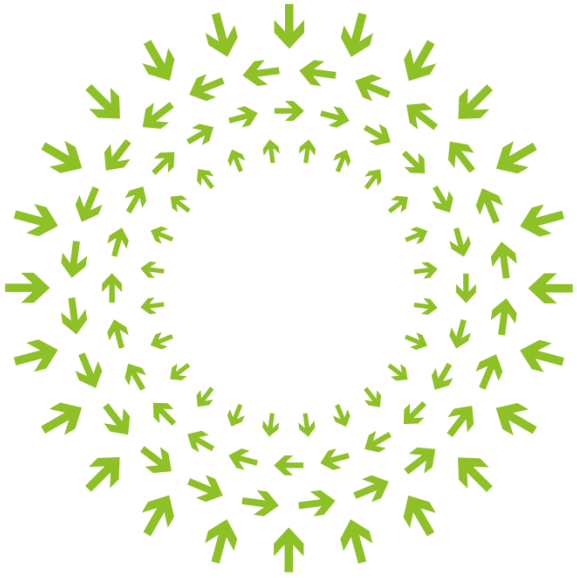


OPEN

Compute Project

Project Zipline
Compression Specification

Author:
Microsoft Corporation

Revision History

Date	Description
03/11/2019	Version 1.0
	-

License

Contents

Table of Figures

Table of Tables

1.1 Purpose

This specification defines a lossless compressed data format that is independent of CPU type, operating system, file system, and character set, and is suitable for compression using the Project Zipline algorithm. The Verilog RTL being contributed was written given the name XP10. To remain consistent with the Verilog RTL, XP10 will be referenced heavily in this document. The

format uses the XP10 compression method and CRC32/CRC64 checksum method for detection of data corruption. This document is the authoritative specification of the XP10 compressed data format.

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

1.2 Intended Audience

This specification is intended for use by software or hardware implementers to compress data into the XP10 format and to decompress data from the XP10 format. This specification assumes a basic background in programming at the level of bits and other primitive data representations.

1.3 Scope

This document specifies a method for representing a sequence of bytes as a (usually shorter) sequence of bits and a method for packing the latter bit sequence into bytes.

1.4 Compliance

Unless otherwise indicated in this document, a compliant decompressor must be able to accept and decompress any data set that conforms to all the specifications presented here. A compliant compressor must produce data sets that conform to all the specifications presented here.

1.5 Terms and Conventions

Bitstream: A stream of data in binary form.

BLT: Bit length table.

Canonical Huffman code: A type of Huffman code with unique properties that allow it to be described in a compact manner. Canonical Huffman code generation is described on the Wikipedia page (https://en.wikipedia.org/wiki/Canonical_Huffman_code).

Codeword: Current byte position in the input bitstream.

CRC: Cyclic redundancy check.

Huffman alphabet: A set of symbols used in Huffman encoding.

Huffman codes: A set of variable-length bit sequences for an alphabet of symbols. To provide compression, more frequent symbols are assigned shorter bit sequences. The bottom-up Huffman construction process is optimal in the sense that the total length of the data is minimized, given the number of times each symbol occurs.

Huffman symbol: See “prefix code.”

LIT: Literal.

Long symbol: Huffman alphabet used to represent match lengths.

LSB: Least significant bit.

LZ77: A general-purpose compression technique introduced by Lempel and Ziv in 1977. Byte sequences that are the same as previous sequences are replaced by a (length, distance) pair that unambiguously references the earlier sequence.

MSB: Most significant bit.

MTF: Move-to-front.

Offset: Distance in the past where a match is found.

Prefix Code: A type of code system, typically variable-length, having the prefix property, where no valid codeword in the system is a prefix of any other valid codeword in the set.

PTR: Pointer to past, consisting of an (offset, length) pair.

Retrospective Huffman Codes: Huffman codes for each Huffman symbol are computed based on its frequency in the coding block. A new Huffman tree is computed for each coding block based on the frequencies of the Huffman symbols.

Short symbol: Huffman alphabet used to represent Literals, MTF and PTR matches.

Static Huffman Codes: Each Huffman symbol has a fixed Huffman code. Huffman code is independent of the frequency of the Huffman symbol and stay the same across coding blocks.

Symbol alphabet: The long and short symbol used in the LZ77 stage before it is encoded.

2 Overview

2.1 What is Project Zipline and XP10?

Project Zipline is a lossless compression data format developed by Microsoft. The Verilog RTL being contributed was written given the name XP10. To remain consistent with the Verilog RTL, XP10 will be referenced heavily in this document. As with other compression formats, XP10 defines the compressed buffer in a way that enables an XP10 decompressor to recover the original uncompressed data. XP10 can be used as a standalone file format and software or as an associated data format used with other compression software.

2.2 Compression Algorithm Overview

The XP10 compression algorithm performs compression in two stages, LZ77 matching and Huffman encoding. It continues work on these two stages until it reaches the end of the input stream.

1. **LZ77 Match**: In this stage, the compressor searches for prior duplicate strings in the input stream using the LZ77 algorithm (Lempel-Ziv 1977, see [1] in the References section). If it finds a match, the compressor emits “{offset, length} pair”, where “offset” specifies how far back from the current position the match is seen, and “length” specifies the length of the match. The minimum length of the match is parameterizable to either 3 or 4 bytes. If no prior match is found, then the current symbol is emitted as a “literal.” The compressor uses the following two techniques to improve the overall compression ratio:

    - **Lazy Matching**: After a match of length “n” has been found, the compressor searches for a longer match starting at the next input symbol. If it finds a longer match, the previous match

is ignored, and the previous symbol is encoded as a literal. The longer match is then encoded as an offset-length pair. Otherwise the original match is emitted. A delayed match window of "n" means that matches for the next "n" symbols are considered for optimal encoding.

- **Move-to-Front (MTF) Buffer:** Offset values from the recently generated {offset, length} pairs are cached in a buffer. If the offset value of a new match matches any of the entries from the cached buffer, the match is encoded using the {index, length} pair where "index" is the entry position in the buffer. The MTF buffer is sized at four entries. All the entries in the buffer are initialized to 0 at the start of the coding frame. During an update all the entries are moved one position (i.e entry 0 is moved to entry 1, entry 1 is moved to entry 2, entry2 is moved to entry 3) and the new Offset value is inserted into entry 0.

The generated literals, {index, length} or {offset, length} pairs, are referred to as LZ77 symbols and are stored in an intermediate buffer. The {index, length} pairs are also referred to as MTF matches and {offset, length} pairs are also referred to as pointer-to-past (PTR) matches.

2. **Huffman Encode:** In this stage, the compressor generates canonical Huffman codes (see [2] in the References section) for the LZ77 symbols produced in the LZ77 match stage. The LZ77 symbols are then encoded using the generated canonical Huffman codes. Canonical Huffman codes are stored in a compressed format in the block header.

2.3 Options and Compliance

Many options affect compression ratio and performance. Options that affect the decompressor will be reflected in the frame and block headers, including window size and minimum match length. XP10–compliant decompressors must support all options that affect decompressors, but each compressor may choose to implement only certain options.

The following is a list of options:

- **Compression levels:** The compression level determines the search window size. Larger windows may find matches of longer lengths, but they are more resource intensive. XP10 supports window sizes ranging from 4 KB to 16 MB. A compressor may choose to implement a smaller window size, but the decompressor must be able to support all compression levels. If a hardware implementation lacks support for a window size then a software library must be provided for decompressing all larger window sizes.

Table 1: XP10 Compression Level Mapping

Compression Level	Window Size (Bytes)
1–2	Not specified
3	4,096
4	8,192
5	16,384
6	65,536

7	262,144
8	1,048,576
9	4,194,304
10	16,777,216

- **Minimum match length:** XP10 uses a minimum match length of 3 or 4 bytes.
- **Delayed match windows:** This option affects compressor performance only, as explained in section 2.2.
- **Use of the prefix engine:** XP10 supports 63 predefined prefixes, each up to 1 KB in size. An XP10–compliant decompressor must have storage for all 63 predefined prefixes. Note that prefix encoding zero is a special encoding, used to indicate that no predefined prefix is used.

3 Compressed Representation

3.1 Data Ordering

3.1.1 Overall Convention

Unless otherwise specified, this document assumes bit 0 to be the least significant bit and is written on the right, and also assumes little endian representation in byte ordering—for example, the least significant byte is stored first at the lower memory address.

3.1.2 Packing Bits to Bytes

XP10 uses the same rule for packing bits to bytes as DEFLATE (see [3] in the References section). For the sake of completeness, we include the entire text of the relevant parts of DEFLATE RFC 1951 in this section.

Compressed data consists of data elements of various bit lengths. The following rules apply when packing data elements into bytes to form the final compressed byte sequence:

- Data elements are packed into bytes in order of increasing bit number within the byte—for example, starting with the least-significant bit of the byte.
- Data elements other than Huffman codes are packed starting with the least-significant bit of the data element.
- Huffman codes are packed starting with the most-significant bit of the code.

In other words, if one were to print out the compressed data as a sequence of bytes, starting with the first byte at the *right* margin and proceeding to the *left*, with the most significant bit of each byte on the left as usual, one would be able to parse the result from right to left, with fixed-width elements in the correct MSB-to-LSB order and Huffman codes in bit-reversed order (that is, with the first bit of the code in the relative LSB position).

3.2 Overview

The XP10 compressed bitstream is called a frame. A frame consists of a frame header and a series of blocks. Each block consists of a block header and data block. Data blocks can be uncompressed raw data or compressed data, which is identified in the block header, as defined in Table 4, presented in section 3.4, "XP10 Block."

The settings in the frame header are used consistently among the blocks within the frame. XP10 has no maximum frame size. The last block indicates the end of the frame by setting the LAST_BLK field in its block header, as defined in Table 4. Implementations of XP10 may choose to limit the uncompressed file size to avoid resource hogging. In this case a software decompressor must be supplied to handle larger frame size. The final uncompressed data is the concatenation of the uncompressed symbols from each block.

XP10 calculates the CRC over the original uncompressed data. Whether to use CRC32 or CRC64 depends on the option chosen in the frame header. If the XP10 file format is used, the CRC field follows the end of the last block as a footer at the next byte boundary. If the XP10 data format is used, it is up to the compression utility to determine where it outputs the footer.

Note that XP10 has no header protection. XP10 chooses to deal with a single CRC error. Header errors are detected because the CRC for the overall data will not match.

Table 2 shows an XP10 structure, preceded by a color key that identifies structure elements.

Color Key for Table 2:

Frame header encapsulation
Compressed data block
Uncompressed data block

Table 2: XP10 Frame Components Example

XP10 Frame Header
XP10 Block (block 0)
XP10 Block (block 1)
XP10 Block (block 2)
…
XP10 Block (block n)
CRC 64 or CRC 32

As mentioned in section 2.1, XP10 can be used as a standalone compression file format. In this case, the frame encapsulation information is output to the same buffer as the data blocks. The decompressor will have all the information needed to decompress the frame given a single buffer location.

A compression utility (software or hardware) can also choose to output the frame encapsulation information to a separate buffer from the data blocks. The corresponding decompression utility will need to receive the information captured in the frame encapsulation through alternative methods in order to decompress the data blocks.

3.3 XP10 Frame Definition

Table 3 list the fields of the XP10 frame. Each field follows right after another with no additional information before, between, or after.

Table 3: XP10 Frame Fields

Field		Bits	Description
XP10_ID		[31:0]	Hardcoded to 0xC039E510. Indicates that the frame is in the XP10 file format.
Flags	WINDOW_SIZE_SEL	[2:0]	Reflects the compression level listed in Table 1. 3'h0: 4,096 3'h1: 8,192 3'h2: 16,384 3'h3: 65,536 3'h4: 262,144 3'h5: 1,048,576 3'h6: 4,194,304 3'h7: 16,777,216
	MIN_MATCH_LEN_SEL	[3]	Selects minimum match length between 3 and 4. In XP10, PTR and MTF use the same minimum match lengths. • **0:** minimum length = 3 • **1:** minimum length = 4
	Mode	[5:4]	Indicates the operating mode for the predefined prefix and predefined Huffman options: • **00:** normal operation—no predefined prefix and no predefined Huffman • **01:** user-defined prefix and no predefined Huffman • **10:** predefined prefix and no predefined Huffman • **11:** predefined prefix and predefined Huffman
	PREDEF_SEL	[11:6]	If the mode is 2'b10 or 2'b11, this field chooses which of the 63 predefined prefixes is used. A value of 0 means that no suitable predefined prefix and no predefined Huffman is found, so normal

			compression or decompression should be performed. For all other modes, this field is ignored.
	RSVD	[13:12]	Reserved. Must be always 0.
	CRC_OPTION	14	• **0:** CRC64 • **1:** CRC32 (optional for small frame size)
	FLG_EXTRA	[15]	• **0:** no extra flag presented • **1:** bit flag extension is present No FLG_EXTENTION fields are currently defined, therefore FLG_EXTRA is always 0.
	FLG_EXTENTION	[31:16]	Reserved. Must be always 0.
…XP10_blocks…			The XP10 blocks that contain the compressed or uncompressed blocks. No padding separates the blocks.
CRC		64/32	CRC64/CRC32 calculation over the original input file based on the CRC_OPTION field in the frame header. This field starts on a byte boundary. If needed, padding bits of zero value will be added to achieve byte alignment.

3.4 XP10 Block

XP10 frames may contain one or more blocks. Each block contains the block header, optional MTF fields, the optional symbol table, and either compressed data or raw uncompressed data as identified in the block header. No paddings exist before or in the middle of the block or between the blocks. Only the end of the last block may be padded with zeros so that the CRC field in the frame footer starts on a byte boundary when the frame footer is stored in the same data buffer as the blocks (file format mode). XP10 block headers contains fixed-length and variable-length headers.

3.4.1 Fixed-Length Headers

The fixed header contains the fields that are always present in the XP10 block header, as specified in Table 4. Each field follows another with nothing in between.

Table 4: XP10 Fixed Block Field Definition

Field	Bits	Description
OUTPUT_SIZE	[27:0]	The output's compressed or uncompressed data size in bits, including the block headers (fixed and variable-length headers). The size does not include the padding bits.
RSVD	28	Reserved. Must be always 0.
BLK_TYPE	29	1'b0 = uncompressed 1'b1 = compressed

MTF_HDR_PRESENT	30	The MTF header is present. If BLK_TYPE is uncompressed, MTF_HDR is not present, regardless of this bit's value.
LAST_BLK	31	1'b1 = this block is the last block in the frame 1'b0 = this block is not the last block in the frame

3.4.2 Compressed Data Header

The compressed data header block is present only if the BLK_TYPE is 1'b1.

Some XP10 block headers exist only when certain fields in the fixed-length block header are set for certain settings.

3.4.2.1 MTF Offset Header

MTF offset headers are present only if the MTF_HDR_PRESENT bit is set. It comes after the fixed-length header. The MTF offset header can be used to help track the MTF state from the previous block if the information is lost from a previous uncompressed data block. It is expected that a new frame will not reference any "stale" entries from a previous frame. The header size is a minimum of 20 bits and defaults to 0.

Table 5: Example of MTF Offset Field

Bits	**MSB**							**LSB 0**
Description	MTF_OFFSET3_LSB	MTF_OFFSET3_EXP	MTF_OFFSET2_LSB	MTF_OFFSET2_EXP	MTF_OFFSET1_LSB	MTF_OFFSET1_EXP	MTF_OFFSET0_LSB	MTF_OFFSET0_EXP

The MTF offset is represented using the (MTF_OFFSET_EXP, MTF_OFFSET_LSB. MTF_OFFSET_EXP = Floor(Log_2(MTF_OFFSET))). If MTF_OFFSET_EXP != 0, the MTF_OFFSET_LSB field is written with MTF_OFFSET – 2 ^ (MTF_OFFSET_EXP) next; otherwise, MTF_OFFSET_LSB is not written.

3.4.2.2 Symbol Table Header

Symbol table headers are present only if BLK_TYPE is compressed. Table 6 shows the content of the symbol header and the conditions in which it is present. Note that there are separate short and long symbols encode type field allows the long symbols to be encode differently than the short symbols. Although the frame header may have the MODE_FIELD = 2'b11, which enable the use of predefine-Huffman, a block has the option of using the Retrospective Huffman if it gives better compression ratio. Predefined Huffman is always used in conjunction with prefix in XP10, however XP10 does not prevent

the case of subsequent block uses of predefined Huffman Table. The predefined Huffman table requirement are described in Section 3.5.1.2.

Table 6: Symbol Table Header

Field	Bits	Description
SHORT_SYMBOL_ENCODE_TYPE	[1:0]	Present if BLK_TYPE=1 2'b00: simple encode 2'b01: predefined Huffman 2'b10: retrospective Huffman 2'b11: reserved
Short Huffman Symbol Table		Present if SHORT_SYMBOL_ENCODE_TYPE == 2'b10
LONG_SYMBOL_ENCODE_TYPE	[3:2]	Present if BLK_TYPE=1. 2'b00: simple encode 2'b01: predefined Huffman 2'b10: retrospective Huffman 2'b11: reserved
Long Huffman Symbols		Present if LONG_SYMBOL_ENCODE_TYPE == 2'b10

3.5 Symbol Encode

3.5.1 Symbol Encoding Methods

Symbol tables contain encoded symbols from the LZ77 stage. Short symbols are encoded followed by the long symbol. XP10 can encode a block in three ways: simple encode, predefined Huffman, and retrospective Huffman. This selection is stored in the symbol table header defined in Table 6. The long and short symbols can use different encoding modes.

3.5.1.1 Simple Encode

One way of encoding the symbols is to use M=Floor (Log_2(symbol alphabet size A)) bits to encode the first N symbols where N=2^(M+1) – A where the symbol is represented with the symbol value itself. The remaining A-N symbols may be encoded with (M+1) bits, where the symbol is encoded to (N << 1). In this simple encode mode, there is no symbol table in the output bitstream.

For example, for 704 short symbols, simple encode uses 9 bits to represent the first 1024-704 = 320 symbols; the remaining 384 symbols will be encoded with 10 bits. For the long symbol table, simple encode uses 8 bits to represent the 256 symbols. For 576 short symbols, simple encode uses 9 bits to represent the first 1024-576 = 448 symbols and 10 bits to encode the remaining 128 symbols. Table 7 gives a complete list of the symbol's lengths at each compression level when using simple encode.

Table 7: Symbol Table Bit Length for Different Compression Levels

Level	Window Size	Long/Short	Symbol	Bit Length	Bit Length Start Code
10	16777216	Short	[0-319]	9	0
			[320-703]	10	640

		Long	[0-255]	8	0
9	4194304	Short	[0-351]	9	0
			[352-671]	10	704
		Long	[0-1]	7	0
			[2-253]	8	4
8	1,048,576	Short	[0-383]	9	0
			[384-639]	10	769
		Long	[0-3]	7	0
			[4-251]	8	8
7	2621,44	Short	[0-415]	9	0
			[416-607]	10	832
		Long	[0-5]	7	0
			[6-249]	8	12
6	65,536	Short	[0-447]	9	0
			[448-575]	10	896
		Long	[0-7]	7	0
			[8-247]	8	16
5	16,384	Short	[0-479]	9	0
			[480-543]	10	960
		Long	[0-9]	7	0
			[10-245]	8	20
4	8,192	Short	[0-495]	9	0
			[496-527]	10	992
		Long	[0-10]	7	0
			[11-244]	8	22
3	4096	Short	[0-511]	9	0
		Long	[0-11]	7	0
			[12-243]	8	24

3.5.1.2 Predefined Huffman Encode

XP10 provides a set of predefined Huffman tables that are not stored in the compressed data. XP10 allows 63 sets of predefined Huffman tables to be used with the prefix. The set is selected by the PREDEF_SEL field in the frame header described previously in Table 3.

The predefined Huffman code provides information for short symbol BLT (bit length table) and long symbol BLT. The BLT table records the bit length for each symbol. The data structure of the BLT is

implementationspecific. However, the data structure needs to capture the bit length for all valid long and short symbols regardless of whether they are used in the bitstream. The provided bit lengths for symbols need to generate valid Huffman codes.

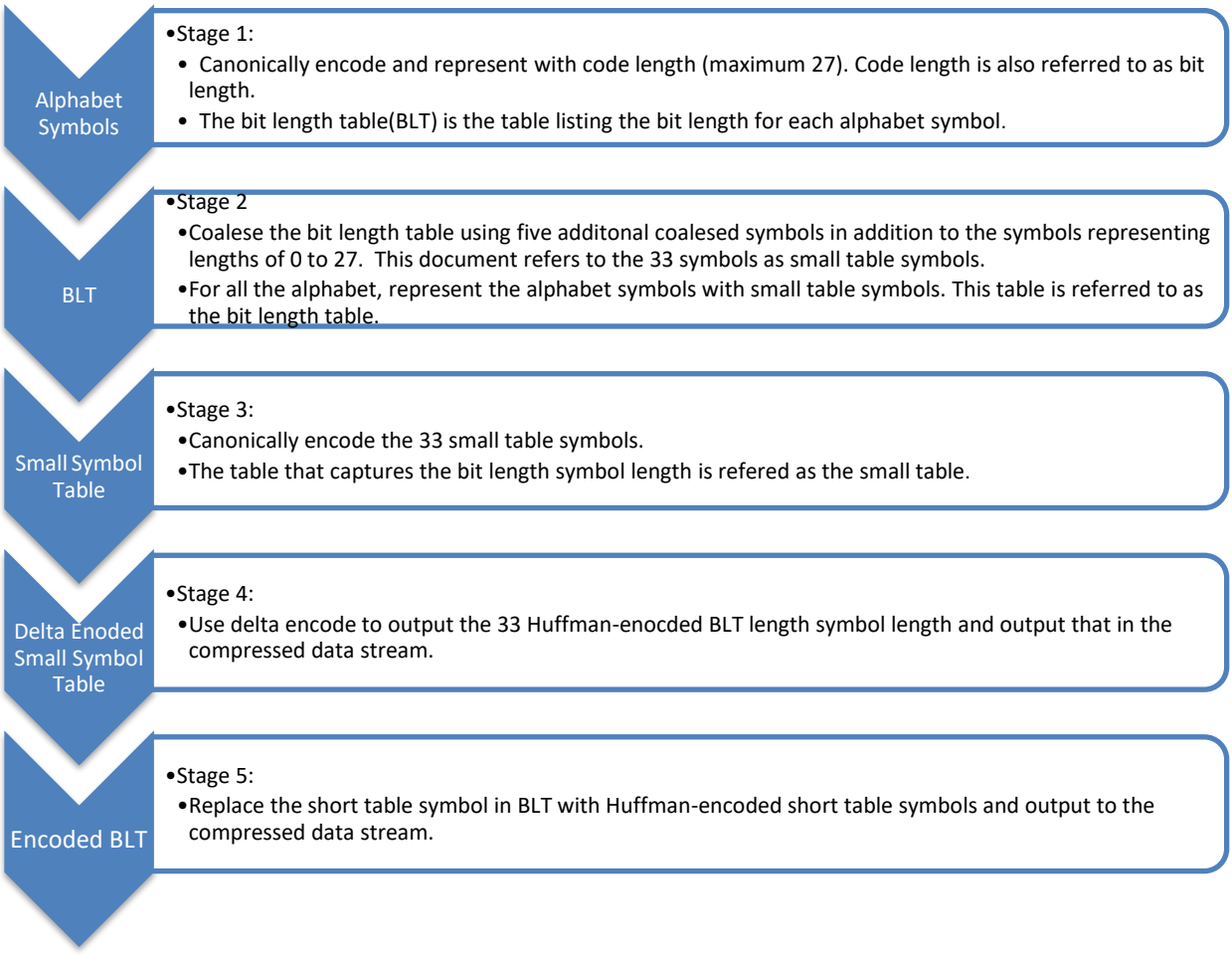
3.5.1.3 Huffman Encoding

3.5.1.3.1 Overview

XP10 performs multiple stages of Huffman encoding to reduce the symbol table size. Figure 1 shows the various stages in the Huffman symbol encoding process. There are separate Huffman encode process for long and short symbol and they are independently encoded. Short symbols are processed first, followed by long symbols. Each symbol table contains two parts:

1. Delta encoded small table (output at stage 4).
2. Encoded symbol bit length table (output at stage 5).

Figure 1: Huffman Symbol Encoding Process



3.5.1.3.2 Stage 1

The first stage compressor uses canonical Huffman code to encode the LZ77 result that contains the histograms on short symbols and long symbols that is referred as the symbol alphabet. The encoded

variable-length codewords are referred as the Huffman symbol alphabet. The maximum length of the Huffman symbol alphabet is 27. This would require a maximum of 5 bits to record the Huffman symbol alphabet length. Symbols that do not occur within the input are assigned zero length. Table 8 shows an example of the intermediate result at the end of Stage 1.

Table 8: Example of Huffman Alphabet Length Encoding

Symbol	Bit Length
0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	8
11	0
12	0
13	0
14	0
15	0
…	…
58	11
…	
66	11
…	…
504	11
505	11
…	
521	12
…	…
703	20

3.5.1.3.3 Stage 2

The code length of the Huffman symbol can be represented using 28 symbols. XP10 uses five additional symbols, ranging from symbols 28 to 32. These five symbols are used to further back-reference or coalesce the code length. These 33 symbols are called small table symbols, and their definition is provided in Table 9. The notation Length[i] means the codeword length of the i-th Huffman symbol alphabet. FILL_COUNT is hardcoded to 16 for XP10.

Table 9: Small Table Symbol Definition

Huffman Bit Length Symbol	Defines	Counting Conditions
0..27	Maps to each code length	When the next five conditions do not occur
28	HUFFMAN_ENCODED_TABLE_FILL	length[i] == 0 till the next FILL_COUNT boundary (where k <= i < ((k+16)>>4) << 4)
29	HUFFMAN_ENCODED_TABLE_ZERO_REPT	length[i] == 0 for the next k consecutive symbols (k >= 5)
30	HUFFMAN_ENCODED_TABLE_PREV	length[i] == length[i-1]
31	HUFFMAN_ENCODED_TABLE_ROW_0	length[i] == length[i- FILL_COUNT]
32	HUFFMAN_ENCODED_TABLE_ROW_1	length[i] == length[i-FILL_COUNT] + 1

3.5.1.3.3.1 HUFFMAN_ENCODED_TABLE_FILL

If a symbol has zero length and the subsequent symbols until the next FILL_COUNT boundary all have zero length, the symbol can be replaced by a HUFFMAN_ENCODED_TABLE_FILL. FILL_COUNT is hardcoded to 16 in XP10. This can be expressed as follows:

> *If Len[i] = 0 for all k <= i < k + FILL_COUNT - (k%FILL_COUNT),*
>
> *replace all Symbol[i] in the range of k <= I < K+ FILL_COUNT – (k%FILL_COUNT) with one Symbol[HUFFMAN_ENCODED_TABLE_FILL]*

Example 1:

If Len[i] == 0 for all 13 <= i <=19, a HUFFMAN_ENCODED_TABLE_FILL symbol can replace and encode the information in symbol[13], symbol[14], and symbol[15]. The remaining zero-length symbols in this range are symbol[16], symbol[17], symbol[18], and symbol[19]. These symbols will still use symbol length 0 to encode.

Example 2:

If Len[i]==0 for all 13< = i <= 38, two HUFFMAN_ENCODED_TABLE_FILL can replace symbol[13] to symbol[31]. Symbol[32] to symbol[35] can be replaced by HUFFMAN_ENCODED_TABLE_ZERO_REPT and a length field described in section 3.5.1.3.3.2.

3.5.1.3.3.2 HUFFMAN_ENCODED_TABLE_ZERO_REPT

HUFFMAN_ENCODED_TABLE_ZERO_REPT is another zero-length coalescing symbol that can be used in replacing consecutive symbols. It requires a minimum of five consecutive zero-length symbols. The consecutive K 0-length symbols can be replaced with a symbol for HUFFMAN_ENCODED_TABLE_ZERO_REPT followed by the encoded length field to capture k, the repeat length, which is encoded in the following ways:

If (5<=k<8) write 2'b(k-5); else {

write 2'b3; k-=8;

while (k>=7) {

write 3'h7

k-=7

}

write 3'hk

}

3.5.1.3.3.3 HUFFMAN_ENCODED_TABLE_PREV

HUFFMAN_ENCODED_TABLE_PREV is used to reference the same bit length seen in the previous nonzero Huffman alphabet. For example, if Huffman alphabet 13 and Huffman alphabet 14 both have a Huffman code length of 26, Huffman alphabet 13 will be represented using small table symbol 26 but Huffman alphabet 14 can be replaced with HUFFMAN_ENCODED_TABLE_PREV. The default previous nonzero value is 8 at the beginning of the comparison.

3.5.1.3.3.4 HUFFMAN_ENCODED_TABLE_ROW_0

HUFFMAN_ENCODED_TABLE_ROW_0 is used to reference the same bit length seen previously in Huffman alphabet at a distance of 16. If Huffman alphabet 13 and Huffman alphabet 29 both have a Huffman code length of 26, Huffman alphabet 13 will be represented using small table symbol 26 but Huffman alphabet 29 can be replaced with HUFFMAN_ENCODED_TABLE_ROW_0.

3.5.1.3.3.5 HUFFMAN_ENCODED_TABLE_ROW_1

HUFFMAN_ENCODED_TABLE_ROW_1 is used to reference the bit length that is one more than what is seen previously in Huffman alphabet at a distance of 16. If Huffman alphabet 13 has a length of 26 and Huffman alphabet 30 has a Huffman code length of 27, Huffman alphabet 13 will be represented using small table symbol 26 but Huffman alphabet 30 can be replaced with HUFFMAN_ENCODED_TABLE_ROW_1 instead of using small table symbol 27.

With the small table symbols, the example in Table 8 will be coalesced to the example in Table 10 (entries in gray represent symbols that are not necessary to explain the scheme).

Table 10: Example of Small Table Symbol Representation of the Short Huffman Symbols Table

Symbol	Bit Length	Small Table Symbol Representation

		Small Table Symbol	Notes
0	0	29	
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	0		
10	8	30	Uses HUFFMAN_ENCODED_TABLE_PREV because the default is 8.
11	0	28	
12	0		
13	0		
14	0		
15	0		
…	…	…	
50	11	11	
…		…	
66	11	31	Has the same length as symbol[50].
…	…	…	
504	11	11	
505	11	30	Has the same length as symbol[504].
…	…	…	
521	12	32	Length as symbol[505] + 1.
…	…	…	
703	0		

3.5.1.3.4 Stage 3

With the small table symbol representation of the BLT, a histogram can be generated from counting the frequency of the small table symbols presented in the bit length table. The 33 small table symbols are encoded again with the histogram using canonical Huffman code. The encoded small table symbol length has a maximum length of 8.

Table 11: Example of Huffman-Encoded Small Table

Small Table Symbol	Bit-Length	Encoded Small Table Symbol Value	In the Bitstream
0	2	2'b0	2'b00
1	0		
2	0		
3	0		
4	7	7'h7e	7'h3f
5	6	6'h3c	6'hf
6	6	6'h3d	6'h2f
7	6	6'h3e	6'h1f
8	5	5'h1a	5'hb
9	5	5'1b	5'h1b
10	5	5'h1c	5'h7
11	4	4'ha	4'h5
12	4	4'hb	4'hd
13	0		
14	0		
15	0		
16	0		
17	0		
18	0		
19	0		
20	0		
21	0		
22	0		
23	0		
24	0		
25	0		
26	0		
27	0		
28	3	3'h4	3'h1
29	7	7'h7f	7'h7f
30	2	2'h1	2'h2
31	4	4'hc	4'h3
32	5	5'h1D	5'h17

3.5.1.3.5 Stage 4

The small table is output to the bitstream using delta encode using the following algorithm.

Previous symbol length prev = 4

For each symbol

K= length of the symbol

If (k == prev) Write 1'b0

Else

Write 1'b1.

If (k > prev) write 3'b(K-1) else write 3'bk

Prev = k;

End foreach

In the example shown in Table 11, the small table will be encoded, as shown in Table 12.

Table 12: Example of Delta Encoded Small Table

Small Table Symbol	Bit-Length	Delta Encode
0	2	1'b1, 3'h2
1	0	1'b1, 3h0
2	0	1'b0
3	0	1'b0
4	7	1'b1, 3'h6
5	6	1'b1, 3'h6
6	6	1'b0
7	6	1'b0
8	5	1'b0
9	5	1'b1, 3'h5
10	5	1'b0
11	4	1'b1, 3'h4
12	4	1'b0
13	0	1'b1, 3'h0
14	0	1'b0
15	0	1'b0
16	0	1'b0
17	0	1'b0

18	0	1'b0
19	0	1'b0
20	0	1'b0
21	0	1'b0
22	0	1'b0
23	0	1'b0
24	0	1'b0
25	0	1'b0
26	0	1'b0
27	0	1'b0
28	3	1'b1,3'h2
29	7	1'b1, 3'h6
30	2	1'b1, 3'h2
31	4	1'b1, 3'h3
32	5	1'b1, 3'h4

3.5.1.3.6 Stage 5

The bit length table is written out to the data stream using the encoded small table symbols. The length field is written out to the data stream using the algorithm described earlier. The final representation of the encoded short symbol BLT table that is present in the output data stream (entries in gray represent symbols that are not necessary to explain the scheme) is shown below. A similar encoded long symbol BLT table will also occur in the bitstream if the Huffman encode scheme is used.

Table 13: Encoded Short Symbol BLT Table

Symbol	Bit Length	Small Table Symbol Representation	Final Encoded Version
		Small Table Symbol	Encoded Symbol
0	0	29	7'h7f followed by 2'h3, 2'h2 to represent 10 consecutive symbols with bit length 0.
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		

9	0		
10	8	30	2'h2
11	0	28	3'h1
12	0		
13	0		
14	0		
15	0		
…	…		
50	11	11	4'h5
…			
66	11	31	4'h3
…	…		
504	11	11	4'h5
505	11	30	2'h2
…	…		
521	12	32	5'h17
…	…		
703	0		

3.6 Symbol Definition

To build canonical Huffman codes, XP10 converts LZ77 match results into symbols. The LZ77 compression stage produces three types of match results, as follows.

- **Literal (LIT):** LIT emits the current byte as it is because it could not find any good matches from the past.
- **Pointer to Past (PTR):** PTR is a match to a previous point, containing the offset and length.
- **Move-to-Front (MTF):** The algorithm maintains offsets of four most recent matches and tries to find matches out of these offsets. Because the offset is one of the last N (0..3) most recent offsets, it can encode the offset via a 2-bit value.

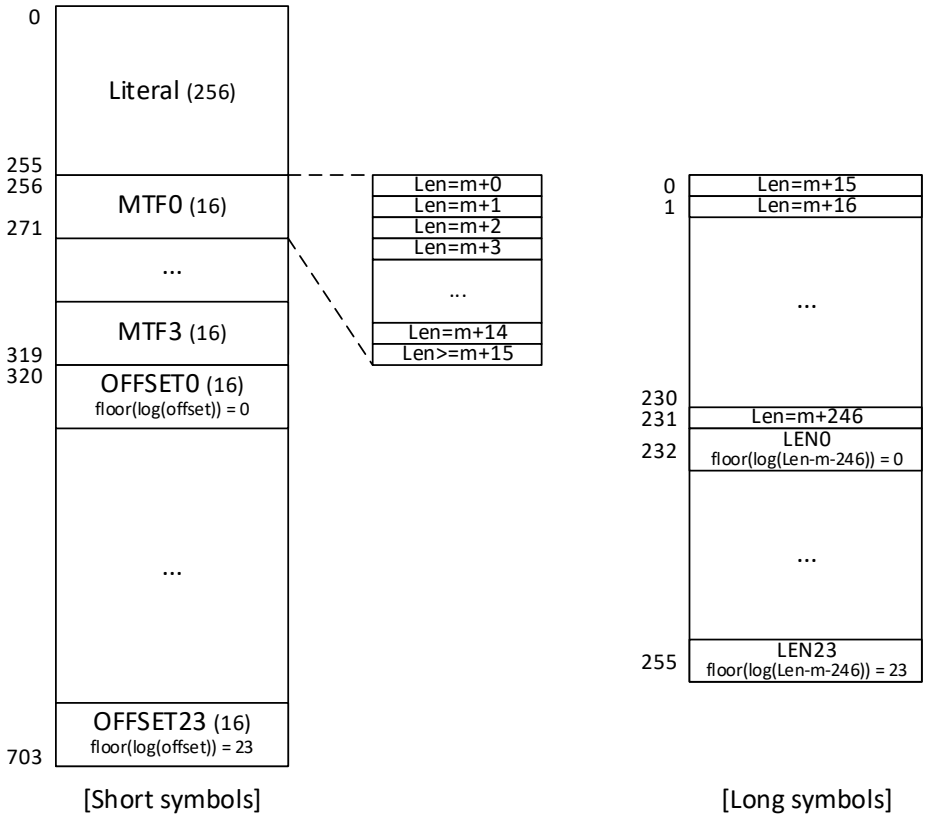
These results are converted into intermediate symbols from which canonical Huffman codes are produced. There are two types of intermediate symbols, short symbols (described in section 3.6.13) and long symbols (described in section 3.6.24). XP10 can have maximum of 704 short symbols and 256 long symbols at level 10. For lower compression levels, the number of the symbol is trimmed down with the search window size. The number of short symbols = 320 + $\log_2$ (Window size) *16. The number of long symbols = 232 + Ceiling (log(Window Size – 246 – MIN_MATCH_LEN)) .Table 14 lists the number of short and long symbols corresponding to each compression level.

Table 14: Number of Short and Long Symbols for Each Compression Level

Level	Window Size	Number of Short Symbols	Number of Long Symbols
3	4,096	512	244
4	8,192	528	245
5	16,384	544	246
6	65,536	576	248
7	262,144	608	250
8	1,048,576	640	252
9	4,194,304	672	254
10	16,777,216	704	256

Figure 2 shows an overview of how the short and long symbols are assigned. Symbol m in the diagram refers to MIN_ MATCH_LEN.

Figure 2: XP10 Level 10 Symbol Assignment



3.6.1 Short Symbols

Short symbols are used for representing literals and MTF and PTR matches. The first 256 short symbols represent literal values according to their ASCII values. A configurable MIN_MATCH_LEN is chosen to determine which matches may be expressed using only a short symbol. This configuration selection is

reflected in the frame header field MIN_MATCH_LEN_SEL. Any match that is less than the minimum length is expressed using literals.

Symbols 256 to 319 are reserved for MTF matches. PTR match offset uses symbol 320 to symbol $\log_2$(window size) + 320 because, for XP10, the match offset and match length is capped at the window size.

Table 15 (preceded by a color key) shows an example of short symbol table with 704 symbols. Each MTF or PTR group has 16 symbols in it where each symbol specifies its length to MIN+0, MIN+1, MIN+2, …, MIN+14, and longer than or equal to MIN+15, respectively. The last symbols in groups, whose match lengths are longer than or equal to MIN+15, further form long symbols described section 3.6.24.

There are four groups of MTF matches based on which recent offset each one matched against in the order from most recent to least recent. PTR matches are divided into 24 groups based on the offset size. The region for PTR matches is divided logarithmically. Group N includes offset from 2^N to 2^(N+1)–1, where N ranges from 0 to 23. Based on this division, XP10 can support offsets up to 2^24, which is equivalent to a 16 MB window size.

Table 15: Short Symbol Table for 704 Short Symbols

Color Key for Table 15:

Literals
MTF
PTR

Symbol Value	Number of Entries	Type	Description	Example
0 … 255	256	Literals	0-255 map to the 255 ASIC characters	Symbol “97” for literal “a”
256 … 271	16	MTF0	Match length = MIN _MATCH_LEN + 0	Symbol “283” for MTF match with length 14 and index 1 if MIN_MTF_MATCH _LEN =3
			Match length = MIN_MATCH_LEN + 1	
			Match length = MIN_MATCH_LEN + 2	
			…	
			Match length = MIN_MATCH_LEN + 14	
			Match length >= MIN_MATCH_LEN + 15	
304 … 319	16	MTF3	Match length = MIN_MATCH_LEN + 0	
			Match length = MIN_MATCH_LEN + 1	
			Match length = MIN_MATCH_LEN + 2	
			…	
			Match length = MIN_MATCH_LEN + 14	

			Match length >= MIN_MATCH_LEN + 15	
320 … 703	16	PTR offset0 Floor(log $_2$(offset)) = 0	Match length = MIN_MATCH_LEN + 0	PTR match with length 14 and offset 0x8245 is encoded as short symbol 570
			Match length = MIN_MATCH_LEN + 1	
			…	
			Match length = MIN_MATCH_LEN + 14	
			Match length >= MIN_MATCH_LEN + 15	
	16	PTR offset1 Floor(log $_2$(offset)) = 1	Match length = MIN_MATCH_LEN + 0	
			Match length = MINMATCH_LEN + 1	
			…	
			Match length = MIN_MATCH_LEN + 14	
			Match length >= MIN_MATCH_LEN + 15	
		…		
	16	PTR offset 23 Floor(log $_2$ (offset)) = 23	Match length = MIN_MATCH_LEN + 0	
			Match length = MIN_MATCH_LEN + 1	
			…	
			Match length = MIN_MATCH_LEN + 14	
			Match length >= MIN_MATCH_LEN + 15	

3.6.2 Long Symbols

Long symbols are used for representing match lengths greater than or equal to MIN_MATCH_LEN + 15. Each long symbol is preceded by a short symbol that encodes a match length of MIN_MATCH_LEN + 15.

Out of the 256 long symbols, the first 232 symbols specify the match length from MIN_MATCH_LEN+15 to MIN_MATCH_LEN+246. For example, a PTR match with a length of 45 and offset 0x8245 is encoded as short symbol 575 long symbol 26 offset “000 0010 0100 0101”.

The rest of the 24 symbols use logarithmic groupings on the length (MIN_MATCH_LEN+246) to encode match lengths longer than MIN_MATCH_LEN+246. For example, a PTR match with a length of 345 and offset 0x8245 is encoded as short symbol 575 long symbol 238, length “011111”, offset “000 0010 0100 0101”.

3.7 Intermediate Representation

With the symbol conversion described here, LZ77 compression stores each symbol’s match information such as offset and length, if needed, and also histograms its frequency for further Huffman code construction.

Table 16 summarize which symbol, offset, and length fields are required to completely express different matches encoded to the data stream. Shaded table cells indicate fields that are packed into the byte stream with the encoded symbol table. The short symbol is packed first, followed by long symbol, length field, and then the offset, using the packing rule described in section 3.1.2.

Table 16: LZ77 Matches to Intermediate Symbol Mapping

Match Type	Match Offset	Length Range	Short Symbol	First Long Symbol	Length	Offset
LIT	N/A	N/A	Y	—	—	—
MTF	N/A	MIN_ MATCH_LEN + [0,14]	Y	—	—	—
MTF	N/A	MIN_MATCH_LEN + [15,247]	Y	Y	—	—
MTF	N/A	MIN_MATCH_LEN + [248,FRAME_SIZE]	Y	Y	Y	—
PTR	=1	MIN_MATCH_LEN + [0,14]	Y	—	—	—
PTR	=1	MIN_MATCH_LEN + [15,247]	Y	Y	—	—
PTR	=1	MIN_MATCH_LEN + [248,FRAME_SIZE]	Y	Y	Y	—
PTR	>1	MIN_MATCH_LEN + [0,14]	Y	—	—	Y
PTR	>1	MIN_MATCH_LEN + [15,247]	Y	Y	—	Y
PTR	>1	MIN_MATCH_LEN + [248,FRAME_SIZE]	Y	Y	Y	Y

The compressor uses an intermediate representation to save the preceding symbol information effectively in the buffer. Offsets and lengths are both limited to the size of the search window; therefore, for a smaller search window, a smaller number of bits is required to represent offset and length. The widths of these fields are implementation-specific as long as the field has enough bits to represent the maximum valid value for that field. For example, software may choose 2 bytes for each field and hardware design may choose 10 bits to implement the 704 short symbols.

The rules on how the compressor needs to represent are nonetheless the same. The representation is specified as follows.

1. For a literal, store it as it is.
2. For a short match whose length is shorter than MIN+15, save its short symbol index at the current location. Its least 4 bits should be less than 15.
    a. If the match is MTF, you don't need any further information because MTF describes a match completely—for example, match length and what recent offset to refer to.
    b. If the match is PTR, save the offset next.
3. For a long match whose length is longer than or equal to MIN+15 and less than or equal to MIN+246, save its short symbol index at the current location first. Its least 4 bits should be

exactly 15. Then, you need to store its long symbol index that ranges from 0 to 231 to the next location.

   a. If the match is MTF, you don't need any further information because MTF describes a match completely—for example, match length and what recent offset to refer to.
   b. If the match is PTR, save the offset at the next location based on the bit length of the offset.

4. For a very long match whose length is longer than MIN+246, save its short symbol index at the current location first. Its least 4 bits should be exactly 15. Then, you need to store its long symbol index that ranges from 232 to 255 to the next location. As this cannot describe its exact match length, you will need the next one or two locations based on the bit length of the match length.
   a. If the match is MTF, you don't need any further information because MTF describes a match completely—for example, match length and what recent offset to use.
   b. If the match is PTR, save the offset at the next location based on the bit length of the offset.

3.8 Compressed Data

After the compressor creates two sets of Huffman code (one each for the short and long symbols), it reads the LZ77 results in the intermediate representation buffer and replaces them with the final compressed data. The short symbol in the intermediate representation is replaced with Huffman alphabet for short symbols. If applicable long symbols exist, they are replaced with the Huffman alphabet for the long symbols, followed by the applicable length field that is to specify a very long match whose length is longer than MIN_MATCH_LEN +246. The length field for the long match is represented in (log (len – MIN_LEN – 246) number of bits. The maximum number of bits for representing the length field is $\log_2$(max_window_size). If the match is a PTR with an offset greater than 1, the applicable offset field is stored in Ceiling(log(offset)) number of bits. The maximum number of bits for representing the offset field is $\log_2$(max_window size).

4 Prefixes

4.1 Predefined Prefixes

A predefined prefix is a predefined uncompressed data stream that is used to seed the hash table initially in the match engine. This is equivalent to feeding the predefined prefix into the search window without identifying any matches. A compressor runs a small sample of data and determines which predefined prefix to use. The prefix data is uncompressed and uses the same search window as regular data. It is rolled out of the window as more of the to-be-compressed data frame is processed. For example, a window size of 4 KB with a 1 KB prefix will have the prefix data start to roll out of the window when 3 KB of the input frame is processed in the search window. Software should use the window size of 8 KB for a 4 KB data frame to keep the prefix data available for matching.

XP10 provides 63 predefined prefixes. The size of each predefined prefix is implementation dependent. Prefix zero is reserved to indicate no suitable prefix for the given input could be found.

The decompressor needs to have the information for the 63 predefined prefixes. It is assumed that higher level software will ensure that the compressor and decompressor are both using the same prefixes. At some future point, an API may be defined to register and/or request a specific set of prefixes known to work well with a given type of data, such as: HTML, executables from a given compiler, English language text files, etc.

4.2 User-Defined Prefixes

The only support for a user-defined prefix in the XP10 format is if the mode field in the frame header is set to value of 2'b01. Any additional information required to process a user-defined prefix (such as location or length) must be supplied through the interface of that implementation (APIs, registers, and so on). The exact means is implementation dependent. This is equivalent to feeding the user prefix into the search window without identifying any matches. If the user-defined prefix exceeds the search window size, only the latter part of the prefix will be referenced during compression.

There is a 4 Byte CRC32 at the end of both the User and the Predetermined prefixes. CRC32 is computed using the polynomial listed in Appendix B.1. These 4 Bytes of CRC are not part of the prefix data and are not included in the prefix length provided to Compressor and Decompressor. For example, a 1KB prefix will have 1024 Bytes of prefix data and an additional 4 Bytes of CRC. The provider of User-defined or Predefined prefixes must always provide the associated CRC32. Applications using either the User-Defined or the Predetermined prefix may use the CRC bytes for integrity check of the associated prefix data. The use of CRC by the compressor or decompressor is optional and is implementation dependent.

5 References

[1] Ziv, J., Lempel, A., “A Universal Algorithm for Sequential Data Compression,” IEEE Transactions on Information Theory, Vol. 23, No. 3, pp. 337–343.

[2] Huffman, D. A., “A Method for the Construction of Minimum Redundancy Codes,” Proceedings of the Institute of Radio Engineers, September 1952, Volume 40, Number 9, pp. 1098–1101.

[3] RFC 1951: Deflate Compressed Data Format Specification version 1.3 (http://www.faqs.org/rfcs/rfc1951.html).

6 Appendix

6.1 Appendix A: XP10 Compact Frame Header

As the XP10 frame and block headers/footers can occupy up to 20 bytes, a smaller header format is also implemented. This compact format is intended for handling short (<= 8 KB) input frames, with only one coding block produced in the compressed output. Note the following:

- Only one header is present—no separate frame and block headers, as in XP10. This single header combines the frame and coding block information.
- No CRCs are included in the format—integrity checking must be done at a higher level.
- No format identifier (such as “Magic Number”) is available.
- Window size is either 4k or 8k only.
- MIN_MATCH_LEN_SEL is fixed at 4.
- MTF offset header is eliminated, but the symbol table header is still present in the coding block portion of the header.

Table 17 defines this format.

Table 17: XP10 Compact Frame Fields

Field		Bits	Description
Flags	OUTPUT_SIZE	[15:0]	If compressed, this field contains the size of the compressed frame only and does not include the 24 bits used for the CFH header. If the data is uncompressible, bits [15:3] of this field contains the byte count of the uncompressed data only, and bits [2:0] are reserved for CFH version. (Currently 0 for Xpress 10 spec 1.0).
	PREDEF_SEL	[21:16]	When MODE == 0, this field chooses which of the 64 predefined prefixes is used. A value of 0 means that no suitable predefined prefix and no predefined Huffman is found, so normal decompression should be performed. If MODE == 1, PREDEF_SEL == 0 indicates uncompressed data output. All other PREDEF_SEL encodings are reserved.
	MODE	[22]	If this bit is set, and PREDEF_SEL == 0, then the Xpress10_block will contain the original, uncompressed, data. The OUTPUT_SIZE field [15:3] will contain the byte count of this data. (Does not include the three bytes of header information, in order to accommodate a full 8KB uncompressed frame). If this bit is clear, PREDEF_SEL is used to select a prefix as described above.
	WINDOW_SIZE_SEL	[23]	Selects 8k match window size when set, or 4k window size when clear. Reserved encoding when data is not compressed.

XP10 Compact Frame Coding Block or Uncompressed data		If PREDEF_SEL == 0 and MODE == 1, then this is the original uncompressed data from the input frame. Otherwise, it is XP10 Compact Frame Coding block that contains the compressed data.

Table 18: XP10 Compact Frame Coding Block Fields

Field	Bits	Description
SHORT_SYMBOL_ENCODE_TYPE	[1:0]	Present if PREDEF_SEL != 0 and MODE == 0 2'b00: simple encode 2'b01: predefined Huffman 2'b10: retrospective Huffman 2'b11: reserved
Short Huffman Symbol Table		Present if SHORT_SYMBOL_ENCODE_TYPE == 2'b10
LONG_SYMBOL_ENCODE_TYPE	[3:2]	Present if PREDEF_SEL != 0 and MODE == 0 2'b00: simple encode 2'b01: predefined Huffman 2'b10: retrospective Huffman 2'b11: reserved
Long Huffman Symbols		Present if LONG_SYMBOL_ENCODE_TYPE == 2'b10

6.2 Appendix B: CRC Calculation

The following code is used for calculating the CRC field in the footer of XP10.

6.2.1 CRC32 Calculation Code

```
const uint32_t POLY32 = 0x82f63b78ULL;
static uint32_t XP10Crc32(const unsigned char* data, const size_t length)
{
    uint32_t crc = 0xFFFFFFFFULL;
    for (size_t i = 0; i < length; i++)
    {
        crc ^= data[i];
        for (int j = 0; j < 8; j++)
        {
            if (crc & 1)
            {
                crc = (crc >> 1) ^ POLY32;
            }
            else
```

```
            {
                crc = (crc >> 1);
            }
        }
    }
    return (crc ^ 0xFFFFFFFFULL);
}
```

6.2.2 CRC64 Calculation Code

```
onst uint64_t POLY64 = 0x9a6c9329ac4bc9b5ULL;
static uint64_t XP10Crc64(const unsigned char* data, const size_t length)
{
    uint64_t crc = 0xFFFFFFFFFFFFFFFFULL;
    for (size_t i = 0; i < length; i++)
    {
        crc ^= data[i];
        for (int j = 0; j < 8; j++)
        {
            if (crc & 1)
            {
                crc = (crc >> 1) ^ POLY64;
            }
            else
            {
                crc = (crc >> 1);
            }
        }
    }
    return (crc ^ 0xFFFFFFFFFFFFFFFFULL);
}
```

6.3 Appendix C: Frame Header Example

The following is a sample frame header for the following settings:

- Window size: 64 KB
- Minimum match length: 4
- Mode: normal (no prefix or predetermined Huffman)
- CRC: CRC32

Input Byte #							
7	6	5	4	3	2	1	0
00	00	00	0B	C0	39	E5	10
Flags				XP10_ID			

6.4 Appendix D: Block Header Example

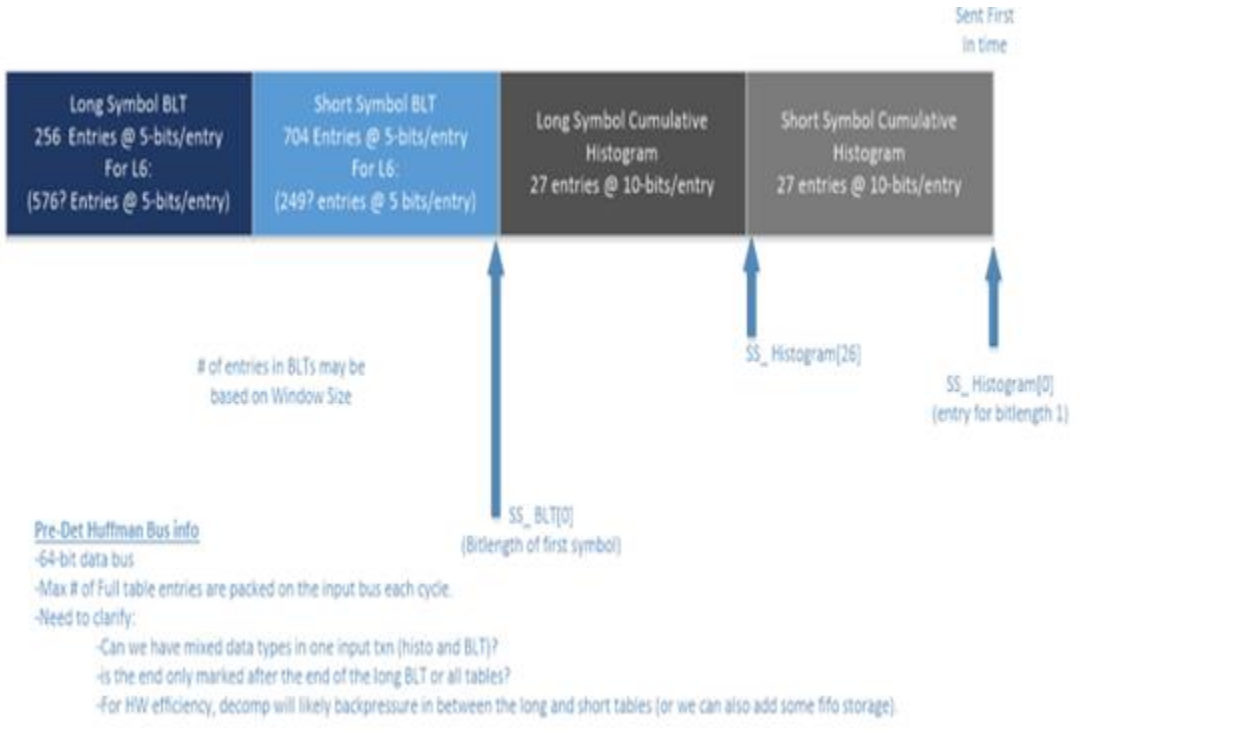
The following is a sample block header for the following values:

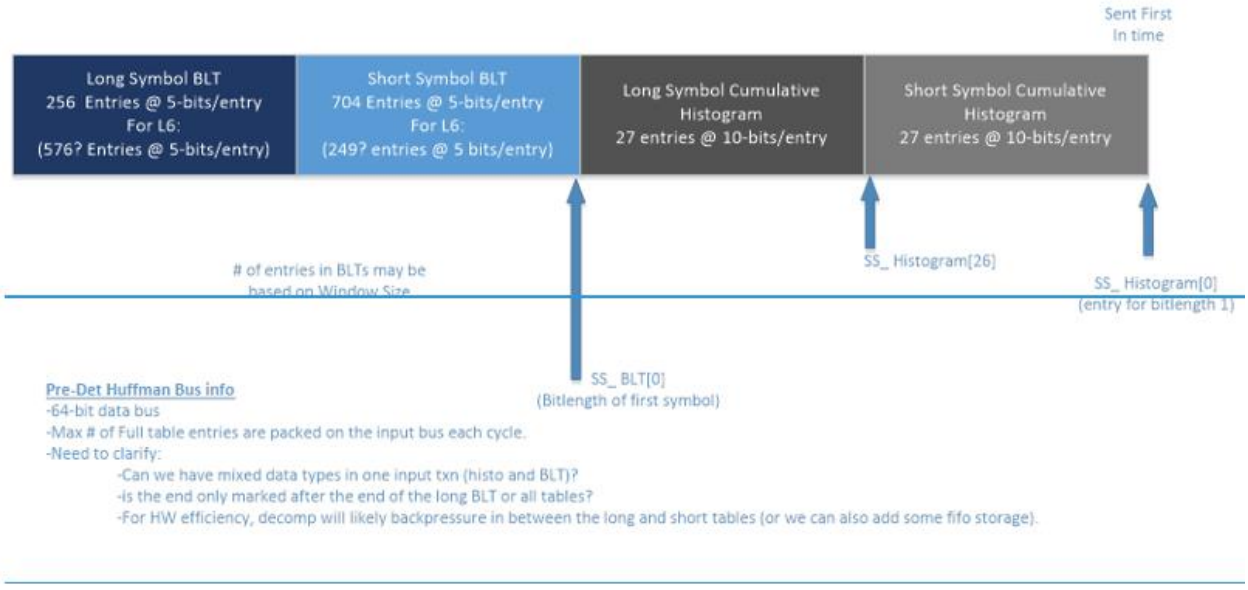
- Output size (bits): 391
- BLK_TYPE: compressed
- MTF header present: yes
- Last block: no
- MTF offsets:
  Offset0: 0, Offset1: 65, Offset2: 53791, Offset3: 2468

Note: The yellow box in this header example is a symbol table header. The orange box has bits 31:28 of the fixed block header.

Input Byte #										
10	9	8	7	6	5	4	3	2	1	0
0	1a	45	d2	1f	78	26	60	00	01	87
	MTF Header							OUTPUT_SIZE (bits)		

6.5 Appendix E: Example of a Predefined Huffman Table Data Structure





6.6 Appendix F: Known Predefined Prefixes

TBD: There are currently no well know predefined prefixes specified.