[MS-XCA]: Xpress Compression Algorithm

This topic lists the Errata found in [MS-XCA] since it was last published.

Since this topic is updated frequently, we recommend that you subscribe to these RSS or Atom feeds to receive update notifications.

Errata are subject to the same terms as the Open Specifications documentation referenced.

RSS

Atom

Errata below are for Protocol Document Version V5.0 – 2018/09/12.

<table>
<tr><th>Errata Published*</th><th>Description</th></tr>
<tr><td>2020/02/17</td><td>

In Section 2.3.4, Processing, we updated the pseudocode for the encoding method for match lengths greater than 65535.

Changed from:

```
    If MatchLength >= 7
        MatchLength -= 7
        If LastLengthHalfByte == 0
            LastLengthHalfByte = OutputPosition
            Write the byte value min(MatchLength, 15) to OutputPosition
            OutputPosition += 1
        Else
            OutputBuffer[LastLengthHalfByte] |= min(15, MatchLength) << 4
            LastLengthHalfByte = 0
        If MatchLength >= 15
            MatchLength -= 15
            Write the byte value min(MatchLength, 255) to OutputPosition
            OutputPosition += 1
            If MatchLength >= 255
                MatchLength += 15 + 7
                Write the 2-byte value MatchLength to OutputPosition
                OutputPosition += 2
```

Changed to:

```
    If MatchLength < 7
        // This is the simple case. The length fits in 3 bits.
```

</td></tr>
</table>

Errata Published*	Description

```
        MatchOffset += MatchLength
        Write MatchOffset the 2-byte value to OutputPosition
        OutputPosition += 2
    Else
        // The length does not fit 3 bits. Record a special value to
        // indicate a longer length.
        MatchOffset |= 7
        Write MatchOffset the 2-byte value to OutputPosition
        OutputPosition += 2
        MatchLength -= 7
        // Try to encode the length in the next 4 bits. If we previously
        // encoded a 4-bit length, we'll use the high 4 bits from that byte.
        If LastLengthHalfByte == 0
            LastLengthHalfByte = OutputPosition
            If MatchLength < 15
                Write single byte value of MatchLength to OutputPosition
                OutputPosition += 1
            Else
                Write single byte value of 15 to OutputPosition
                OutputPosition++
                goto EncodeExtraLen
        Else
            If MatchLength < 15
                OutputBuffer[LastLengthHalfByte] |= MatchLength << 4
                LastLengthHalfByte = 0
            Else
                OutputBuffer[LastLengthHalfByte] |= 15 << 4
                LastLengthHalfByte = 0
    EncodeExtraLen:
                // We've already used 3 bits + 4 bits to encode the length
                // Next use the next byte.
                MatchLength -= 15
                If MatchLength < 255
                    Write single byte value of MatchLength to OutputPosition
```

Errata Published*	Description

	`        OutputPosition += 1`
	`    Else`
	`        // Use two more bytes for the length`
	`        Write single byte value of 255 to OutputPosition`
	`        OutputPosition += 1`
	`        MatchLength += 7 + 15`
	`        If MatchLength < (1 << 16)`
	`            Write two-byte value MatchLength to OutputPosition`
	`            OutputPosition += 2`
	`        Else`
	`            Write two-byte value of 0 to OutputPosition`
	`            OutputPosition += 2`
	`            Write four-byte value of MatchLength to OutputPosition`
	`            OutputPosition += 4`

2020/02/17	In Section 2.3.4 Processing, we added clarifying information about the maximum MatchLength. Changed from: The fastest variant of the Xpress Compression Algorithm avoids the cost of the Huffman[IEEE-MRC] pass by encoding the LZ77 [UASDC] literals and matches in a simple way. The encoding process is similar to the method described in section 2.1.4.1, with the key difference that the largest match offset it can encode is 8192 instead of the 65535 limit of the Huffman format. The literal or match flags are encoded in 32-bit chunks. Literals are encoded with a simple byte value. Matches are encoded with a 16-bit value, where the high 13 bits represent the offset and the low 3 bits represent the length. Long lengths are encoded with an additional 4 bits, then 8 bits, and then 16 bits. The following pseudocode provides an outline of the encoding method. Changed to: The fastest variant of the Xpress Compression Algorithm avoids the cost of the Huffman[IEEE-MRC] pass by encoding the LZ77 [UASDC] literals and matches in a simple way. The encoding process is similar to the method described in section 2.1.4.1, with the key difference that the largest match offset it can encode is 8192 instead of the 65535 limit of the Huffman format. The literal or match flags are encoded in 32-bit chunks. Literals are encoded with a simple byte value. Matches are encoded with a 16-bit value, where the high 13 bits represent the offset and the low 3 bits represent the length. Long lengths are encoded with an additional 4 bits, then 8 bits, and then 16 bits. The MatchLength is represented by a ULONG, a 32-bit unsigned integer (see [MS-DTYP] section 2.2.51); therefore, the maximum value is 4,294,967,295. The following pseudocode provides an outline of the encoding method.
2020/02/17	In Section 2.2.4 Processing, we corrected the pseudocode by replacing DecodedValue with HuffmanSymbol and added a clarifying comment to the pseudocode to explain why the HuffmanSymbol needs to be right-shifted by 4 bits.

Errata Published*	Description
	Changed from:

```
…
Loop until a decompression terminating condition
   Build the decoding table
   CurrentPosition = 256    // start at the end of the Huffman table
   NextBits = Read16Bits(InputBuffer + CurrentPosition)
   CurrentPosition += 2
   NextBits <<= 16
   NextBits |= Read16Bits(InputBuffer + CurrentPosition)
   CurrentPosition += 2
   ExtraBits = 16
   BlockEnd = OutputPosition + 65536
   Loop until a block terminating condition
       If OutputPosition >= BlockEnd then terminate block processing
       Loop until a literal processing terminating condition
           Next15Bits = NextBits >> (32 – 15)
           HuffmanSymbol = DecodingTable[Next15Bits]
           HuffmanSymbolBitLength = the bit length of HuffmanSymbol, from
the table in
                                    the input buffer
           If HuffmanSymbol <= 0
               NextBits <<= HuffmanSymbolBitLength
               ExtraBits -= HuffmanSymbolBitLength
                Do
                    HuffmanSymbol = - HuffmanSymbol
                    HuffmanSymbol += (NextBits >> 31)
                    NextBits *= 2
                    ExtraBits = ExtraBits - 1
                    HuffmanSymbol = DecodingTable[HuffmanSymbol]
                While DecodedValue <= 0
            Else
                DecodedBitCount = DecodedValue & 15
                NextBits <<= DecodedBitCount
                ExtraBits -= DedcodedBitCount
```

Errata Published*	Description

```
            HuffmanSymbol >>= 4
            HuffmanSymbol -= 256
            If ExtraBits < 0
                NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-ExtraBits)
                ExtraBits += 16
                CurrentPosition += 2
            If HuffmanSymbol >= 0
                If HuffmanSymbol == 0
                    If the entire input buffer has been read and
                    the expected decompressed size has been written to the output buffer
                        Decompression is complete.  Return with success.
                Terminate literal processing
            Else
                Output the byte value of HuffmanSymbol to the output stream
        End of literal processing Loop
        MatchLength = HuffmanSymbol mod 16
        MatchOffsetBitLength = HuffmanSymbol / 16
        If MatchLength == 15
            MatchLength = ReadByte(InputBuffer + CurrentPosition)
            CurrentPosition += 1
            If MatchLength == 255
                MatchLength = Read16Bits(InputBuffer + CurrentPosition)
                CurrentPosition += 2
                If MatchLength < 15
                    The compressed data is invalid. Return error.
                MatchLength = MatchLength - 15
            MatchLength = MatchLength + 15
        MatchLength = MatchLength + 3
        MatchOffset = NextBits >> (32 – MatchOffsetBitLength)
        MatchOffset += (1 << MatchOffsetBitLength)
        NextBits <<= MatchOffsetBitLength
        ExtraBits -= MatchOffsetBitLength
        If ExtraBits < 0
```

Errata Published*	Description

```
           NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-
ExtraBits)

           ExtraBits += 16

           CurrentPosition += 2

       For i = 0 to MatchLength - 1

           Output OutputBuffer[CurrentOutputPosition – MatchOffset + i]

    End of block loop

End of decoding loop
```

Changed to:

```
…
Loop until a decompression terminating condition

   Build the decoding table

   CurrentPosition = 256    // start at the end of the Huffman table

   NextBits = Read16Bits(InputBuffer + CurrentPosition)

   CurrentPosition += 2

   NextBits <<= 16

   NextBits |= Read16Bits(InputBuffer + CurrentPosition)

   CurrentPosition += 2

   ExtraBits = 16

   BlockEnd = OutputPosition + 65536

   Loop until a block terminating condition

       If OutputPosition >= BlockEnd then terminate block processing

       Loop until a literal processing terminating condition

           Next15Bits = NextBits >> (32 – 15)

           HuffmanSymbol = DecodingTable[Next15Bits]

           HuffmanSymbolBitLength = the bit length of HuffmanSymbol, from
the table in
                                    the input buffer

           If HuffmanSymbol <= 0

               NextBits <<= HuffmanSymbolBitLength

               ExtraBits -= HuffmanSymbolBitLength

                Do

                    HuffmanSymbol = - HuffmanSymbol
```

Errata Published*	Description

```
                    HuffmanSymbol += (NextBits >> 31)

                    NextBits *= 2

                    ExtraBits = ExtraBits - 1

                    HuffmanSymbol = DecodingTable[HuffmanSymbol]

                While HuffmanSymbol <= 0

            Else

                DecodedBitCount = HuffmanSymbol & 15

                NextBits <<= DecodedBitCount

                ExtraBits -= DedcodedBitCount

           HuffmanSymbol >>= 4 // Shift by 4 bits to get the symbol value

                               // (the lower 4 bits are the bit length of
the symbol)

           HuffmanSymbol -= 256

           If ExtraBits < 0

               NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-
ExtraBits)

               ExtraBits += 16

               CurrentPosition += 2

           If HuffmanSymbol >= 0

               If HuffmanSymbol == 0

                   If the entire input buffer has been read and

                   the expected decompressed size has been written to the
output buffer

                       Decompression is complete.  Return with success.

               Terminate literal processing

           Else

               Output the byte value of HuffmanSymbol to the output stream

       End of literal processing Loop

       MatchLength = HuffmanSymbol mod 16

       MatchOffsetBitLength = HuffmanSymbol / 16

       If MatchLength == 15

           MatchLength = ReadByte(InputBuffer + CurrentPosition)

           CurrentPosition += 1

           If MatchLength == 255

               MatchLength = Read16Bits(InputBuffer + CurrentPosition)
```

<table>
<tr><th>Errata Published*</th><th>Description</th></tr>
<tr><td></td><td><pre>
                CurrentPosition += 2
                If MatchLength < 15
                    The compressed data is invalid. Return error.
                MatchLength = MatchLength - 15
            MatchLength = MatchLength + 15
        MatchLength = MatchLength + 3
        MatchOffset = NextBits >> (32 – MatchOffsetBitLength)
        MatchOffset += (1 << MatchOffsetBitLength)
        NextBits <<= MatchOffsetBitLength
        ExtraBits -= MatchOffsetBitLength
        If ExtraBits < 0
            NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-ExtraBits)
            ExtraBits += 16
            CurrentPosition += 2
        For i = 0 to MatchLength - 1
            Output OutputBuffer[CurrentOutputPosition – MatchOffset + i]
    End of block loop
End of decoding loop
</pre></td></tr>
<tr><td>2019/12/09</td><td>In Section 2.1, LZ77+Huffman Compression Algorithm Details, described how data is processed for the Huffman variant.

Changed from:
The overall compression algorithm for the Huffman [IEEE-MRC] variant can be divided into three stages, which are performed in this order:
...

Changed to:
The overall compression algorithm for the Huffman [IEEE-MRC] variant can handle an arbitrary amount of data. Data is processed in 64k blocks, and the encoded results are stored in-order. After the final block, the end-of-file (EOF) symbol is encoded. Each 64k block is run through three stages, which are performed in this order:
...

In Section 2.2.4, Processing, described the decompression process and clarified how the compression stream handles the bytes for long match lengths in the pseudocode.

Changed from:</td></tr>
</table>

Errata Published*	Description
	The decompression algorithm uses the 256-byte Huffman table to reconstruct the canonical Huffman [IEEE-MRC] representations of each symbol. Next, the Huffman stream of LZ77 ([UASDC]) literals and matches is decoded to reproduce the original data. The following method can be used to construct a decoding table. The decoding table will have 2^15 entries because 15 is the maximum bit length permitted by the Xpress Compression Algorithm for a Huffman code. If a symbol has a bit length of X, it has 2^(15 – X) entries in the table that point to its value. The order of symbols in the table is sorted by bit length (from low to high), and then by symbol value (from low to high). These requirements represent the agreement of canonicalness with the compression end of the algorithm. The following pseudocode shows the table construction method: ... The compression stream is designed to be read in (mostly) 16-bit chunks, with a 32-bit register maintaining at least the next 16 bits of input. This strategy allows the code to seamlessly handle the bytes for long match lengths, which would otherwise be awkward. The following pseudocode demonstrates this method.

```
Build the decoding table
CurrentPosition = 256    // start at the end of the Huffman table
NextBits = Read16Bits(InputBuffer + CurrentPosition)
CurrentPosition += 2
NextBits <<= 16
NextBits |= Read16Bits(InputBuffer + CurrentPosition)
CurrentPosition += 2
ExtraBits = 16
Loop until a terminating condition
  Next15Bits = NextBits >> (32 – 15)
  HuffmanSymbol = DecodingTable[Next15Bits]
  HuffmanSymbolBitLength = the bit length of HuffmanSymbol, from the table in
                  the input buffer
  NextBits <<= HuffmanSymbolBitLength
  ExtraBits -= HuffmanSymbolBitLength
  If ExtraBits < 0
     NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-ExtraBits)
     ExtraBits += 16
     CurrentPosition += 2
  If HuffmanSymbol < 256
     Output the byte value HuffmanSymbol to the output stream.
  Else If HuffmanSymbol == 256 and
        the entire input buffer has been read and
        the expected decompressed size has been written to the output buffer
     Decompression is complete.  Return with success.
  Else
     HuffmanSymbol = HuffmanSymbol - 256
     MatchLength = HuffmanSymbol mod 16
     MatchOffsetBitLength = HuffmanSymbol / 16
     If MatchLength == 15
        MatchLength = ReadByte(InputBuffer + CurrentPosition)
        CurrentPosition += 1
        If MatchLength == 255
```

Errata Published*	Description

```
            MatchLength = Read16Bits(InputBuffer + CurrentPosition)
            CurrentPosition += 2
            If MatchLength < 15
                The compressed data is invalid. Return error.
            MatchLength = MatchLength - 15
        MatchLength = MatchLength + 15
    MatchLength = MatchLength + 3
    MatchOffset = NextBits >> (32 – MatchOffsetBitLength)
    MatchOffset += (1 << MatchOffsetBitLength)
    NextBits <<= MatchOffsetBitLength
    ExtraBits -= MatchOffsetBitLength
    If ExtraBits < 0
        Read the next 2 bytes the same as the preceding (ExtraBits < 0) case
    For i = 0 to MatchLength - 1
        Output OutputBuffer[CurrentOutputPosition – MatchOffset + i]
...
```

Changed to:

The decompression processes a series of blocks to form the decompressed output. Each block is processed in-order, and its decoded content written to the output stream is in-order. When processing a block, we check for terminating conditions for both block and overall decoding.

The decompression algorithm uses the 256-byte Huffman table to reconstruct the canonical Huffman [IEEE-MRC] representations of each symbol. Next, the Huffman stream of LZ77 ([UASDC]) literals and matches is decoded to reproduce the original data.

The following method can be used to construct a decoding table. The decoding table will have 2^15 entries because 15 is the maximum bit length permitted by the Xpress Compression Algorithm for a Huffman code. If a symbol has a bit length of X, it has 2^(15 – X) entries in the table that point to its value. The order of symbols in the table is sorted by bit length (from low to high), and then by symbol value (from low to high). These requirements represent the agreement of canonicalness with the compression end of the algorithm. The following pseudocode shows the table construction method:

...

The compression stream is designed to be read in (mostly) 16-bit chunks, with a 32-bit register maintaining at least the next 16 bits of input. This strategy allows the code to seamlessly handle the bytes for long match lengths, which would otherwise be awkward. The following pseudocode demonstrates this method.

```
Loop until a decompression terminating condition
   Build the decoding table
   CurrentPosition = 256    // start at the end of the Huffman table
   NextBits = Read16Bits(InputBuffer + CurrentPosition)
   CurrentPosition += 2
   NextBits <<= 16
   NextBits |= Read16Bits(InputBuffer + CurrentPosition)
   CurrentPosition += 2
   ExtraBits = 16
   BlockEnd = OutputPosition + 65536

   Loop until a block terminating condition
```

Errata Published*	Description

```
If OutputPosition >= BlockEnd then terminate block processing
Loop until a literal processing terminating condition
    Next15Bits = NextBits >> (32 – 15)
    HuffmanSymbol = DecodingTable[Next15Bits]
    HuffmanSymbolBitLength = the bit length of HuffmanSymbol, from the table in
                             the input buffer
    If HuffmanSymbol <= 0
        NextBits <<= HuffmanSymbolBitLength
        ExtraBits -= HuffmanSymbolBitLength

         Do
             HuffmanSymbol = - HuffmanSymbol
             HuffmanSymbol += (NextBits >> 31)
             NextBits *= 2
             ExtraBits = ExtraBits - 1
             HuffmanSymbol = DecodingTable[HuffmanSymbol]
         While DecodedValue <= 0
     Else
         DecodedBitCount = DecodedValue & 15
         NextBits <<= DecodedBitCount
         ExtraBits -= DedcodedBitCount
    HuffmanSymbol >>= 4
    HuffmanSymbol -= 256
    If ExtraBits < 0
        NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-ExtraBits)
        ExtraBits += 16
        CurrentPosition += 2
    If HuffmanSymbol >= 0
        If HuffmanSymbol == 0
            If the entire input buffer has been read and
            the expected decompressed size has been written to the output buffer
                Decompression is complete.  Return with success.
        Terminate literal processing
    Else
        Output the byte value of HuffmanSymbol to the output stream
End of literal processing Loop

MatchLength = HuffmanSymbol mod 16
MatchOffsetBitLength = HuffmanSymbol / 16
If MatchLength == 15
    MatchLength = ReadByte(InputBuffer + CurrentPosition)
    CurrentPosition += 1
    If MatchLength == 255
        MatchLength = Read16Bits(InputBuffer + CurrentPosition)
        CurrentPosition += 2
        If MatchLength < 15
            The compressed data is invalid. Return error.
        MatchLength = MatchLength - 15
```

Errata Published*	Description

	MatchLength = MatchLength + 15
	      MatchLength = MatchLength + 3
	      MatchOffset = NextBits >> (32 – MatchOffsetBitLength)
	      MatchOffset += (1 << MatchOffsetBitLength)
	      NextBits <<= MatchOffsetBitLength
	      ExtraBits -= MatchOffsetBitLength
	      If ExtraBits < 0
	         NextBits |= Read16Bits(InputBuffer + CurrentPosition) << (-ExtraBits)
	         ExtraBits += 16
	         CurrentPosition += 2
	      For i = 0 to MatchLength - 1
	         Output OutputBuffer[CurrentOutputPosition – MatchOffset + i]
	   End of block loop
	End of decoding loop
	...

2019/09/02	In Section 2.4.4, Processing, pseudocode supporting longer matches has been updated
	
	Changed from:
	...
	The match length can be greater than the match offset, and this necessitates the 1-byte-at-a-time copying strategy shown in the following pseudocode.
	
	    BufferedFlags = 0
	    BufferedFlagCount = 0
	    InputPosition = 0
	    OutputPosition = 0
	    LastLengthHalfByte = 0
	    Loop until break instruction or error
	        If BufferedFlagCount == 0
	            BufferedFlags = read 4 bytes at InputPosition
	            InputPosition += 4
	            BufferedFlagCount = 32
	        BufferedFlagCount = BufferedFlagCount – 1
	        If (BufferedFlags & (1 << BufferedFlagCount)) == 0
	            Copy 1 byte from InputPosition to OutputPosition. Advance both.
	        Else
	            If InputPosition == InputBufferSize
	                Decompression is complete. Return with success.
	            MatchBytes = read 2 bytes from InputPosition
	            InputPosition += 2
	            MatchLength = MatchBytes mod 8
	            MatchOffset = (MatchBytes / 8) + 1
	            If MatchLength == 7
	                If LastLengthHalfByte == 0
	                    MatchLength = read 1 byte from InputPosition
	                    MatchLength = MatchLength mod 16
	                    LastLengthHalfByte = InputPosition
	                    InputPosition += 1
	                Else
	                    MatchLength = read 1 byte from LastLengthHalfByte position
	                    MatchLength = MatchLength / 16
	                    LastLengthHalfByte = 0
	                If MatchLength == 15
	                    MatchLength = read 1 byte from InputPosition
	                    InputPosition += 1
	                    If MatchLength == 255

Errata Published*	Description

```
                    MatchLength = read 2 bytes from InputPosition
                    InputPosition += 2
                    If MatchLength < 15 + 7
                       Return error.
                    MatchLength -= (15 + 7)
                MatchLength += 15
            MatchLength += 7
        MatchLength += 3
        For i = 0 to MatchLength – 1
            Copy 1 byte from OutputBuffer[OutputPosition – MatchOffset]
            OutputPosition += 1
```

Changed to:

...

The match length can be greater than the match offset, and this necessitates the 1-byte-at-a-time copying strategy shown in the following pseudocode.

```
BufferedFlags = 0
BufferedFlagCount = 0
InputPosition = 0
OutputPosition = 0
LastLengthHalfByte = 0
Loop until break instruction or error
    If BufferedFlagCount == 0
        BufferedFlags = read 4 bytes at InputPosition
        InputPosition += 4
        BufferedFlagCount = 32
    BufferedFlagCount = BufferedFlagCount – 1
    If (BufferedFlags & (1 << BufferedFlagCount)) == 0
        Copy 1 byte from InputPosition to OutputPosition.  Advance both.
    Else
        If InputPosition == InputBufferSize
            Decompression is complete.  Return with success.
        MatchBytes = read 2 bytes from InputPosition
        InputPosition += 2
        MatchLength = MatchBytes mod 8
        MatchOffset = (MatchBytes / 8) + 1
        If MatchLength == 7
            If LastLengthHalfByte == 0
                MatchLength = read 1 byte from InputPosition
                MatchLength = MatchLength mod 16
                LastLengthHalfByte = InputPosition
                InputPosition += 1
            Else
                MatchLength = read 1 byte from LastLengthHalfByte position
                MatchLength = MatchLength / 16
                LastLengthHalfByte = 0
            If MatchLength == 15
                MatchLength = read 1 byte from InputPosition
                InputPosition += 1
                If MatchLength == 255
                    MatchLength = read 2 bytes from InputPosition
                    InputPosition += 2
                    If MatchLength == 0
                        MatchLength = read 4 bytes from InputPosition
                        InputPosition += 4 bytes
                    If MatchLength < 15 + 7
                       Return error.
                    MatchLength -= (15 + 7)
                MatchLength += 15
            MatchLength += 7
        MatchLength += 3
```

Errata Published*	Description

	For i = 0 to MatchLength - 1
	    Copy 1 byte from OutputBuffer[OutputPosition - MatchOffset]
	    OutputPosition += 1

2019/07/08	In Section 2.1.4.2, Huffman Code Construction Phase, clarified that the sorting algorithm used in the Huffman Code construction phase is stable. Changed from: ... The following flowchart illustrates the length-limited canonical Huffman code construction method. ... Changed to: ... The following flowchart illustrates the length-limited canonical Huffman code construction method. Note that the sorting algorithm used in the Huffman Code construction phase is stable. ...
2019/07/08	In Section 2.1.4.3 Final Encoding Phase, clarified that some implementations of the decompression algorithm expect a terminating Huffman symbol and that it is recommended the encoding algorithm append this symbol. Changed from: Some implementations of the decompression algorithm expect an extra symbol to mark the end of the data. For example, certain implementations fail during decompression if the Huffman symbol 256 is not found after the actual data. For this reason, the encoding algorithm appends this symbol and increments the count of symbol 256 before the Huffman codes are constructed. Changed to: Implementations of the decompression algorithm may expect an extra symbol to mark the end of the data. For example, certain implementations fail during decompression if the Huffman symbol 256 is not found after the actual data. For this reason, the encoding algorithm SHOULD append this symbol and increment the count of symbol 256 before the Huffman codes are constructed.

*Date format: YYYY/MM/DD