XFLATE: A Random Access Extension to DEFLATE

Version: 1.0.0 (2017-05-22)
Source: https://github.com/dsnet/compress

Author: Joe Tsai ⟨joetsai@digital-static.net⟩
Website: http://digital-static.net

1 Introduction

We present XFLATE, an extension to DEFLATE that provides the ability to read chunks of a compressed data stream in a random access manner by encoding an index of the chunk locations into the data stream itself. The extension remains backwards compatible with DEFLATE such that all RFC 1951 compliant decoders will also be able to read XFLATE.

1.1 Background information

The DEFLATE format defined by RFC 1951 is arguably the world's most common compression format, combining decent compression ratios with decent compression and decompression rates. Unfortunately, it was never designed for random access decompression, which is useful in large compressed files such as Zip archives, disk images, DNA sequences, and others. This document proposes an extension to DEFLATE that provides random access properties, while also ensuring complete backwards compatibility with DEFLATE.

In order for a compression format to be randomly accessible, the compressed output needs to be comprised of individually compressed chunks and also needs to provide a way for those chunks to be easily located. In the terminology used by other compression formats, a table that records the location of every chunk is called an index. Formats like XZ, which are designed with random access in mind, make the index part of the format. Unfortunately, the design of DEFLATE provides no easy way to encode this meta-information into the stream in such a way that it does not alter the uncompressed output.

Our approach solves this issue by using the dynamic Huffman compressed blocks of DEFLATE. As an oversimplification, these blocks are comprised of two parts: a Huffman tree definition and a data section, which is interpreted by the preceding tree. By specifying that the data section contains no data, we can use the Huffman tree definition to encode arbitrary metadata. However, generating *valid* Huffman trees that still encode arbitrary metadata is no trivial matter, but is possible. As such, we describe the process in detail later.

With the ability to encode arbitrary metadata into the stream in such a way that does not affect the uncompressed output, one can see how we can extend DEFLATE to include an index that allows for random access decompression. This document describes in detail a format for encoding an index and also the format for encoding in-band metadata into a DEFLATE stream.

Contrary to most other approaches, which choose to extend Gzip in some way, our approach addresses the issue at the DEFLATE layer since it is the underlying compression algorithm of many other formats including Gzip, Zip, PNG, PDF, etc. If we can provide random access compression in DEFLATE, then other formats that rely on DEFLATE can potentially inherit those benefits.

1.2 Design goals

The following are some design goals of XFLATE:

- *Be backwards compatible with DEFLATE.* That is, any stream encoded as XFLATE be decodable by any compliant DEFLATE decoder without issue. The output data when decoded as XFLATE and as DEFLATE must be identical.
- *Maintain a streamed input and output.* When encoding to XFLATE, the size of the working memory is independent of the chunk size or the total amount of input data. Thus, the input may come from and the output may go to a process pipe.
- *Parallelizable compression and decompression.* Since there are no dependencies between each chunk, they can be individually compressed or decompressed in parallel. This allows for better utilization of modern processors with many cores.
- *Require no external index file.* Unlike several other solutions, the index table that contains information about where chunks are located is embedded in the compressed stream itself in a way that does not affect the uncompressed output.
- *Selectable chunk size.* The uncompressed size chosen for each chunk can be individually configured. The user may select larger chunks to improved storage efficiency at the cost of reduced random access performance and vice-versa.
- *Encode arbitrarily large data sets.* The format uses variable-length integer values to store sizes and allows for an infinite number of indexes to be chained together. This ensures that an unlimited amount of data may be represented by the format.
- *Be a simple extension.* This is a subjective metric, but we strive to make the format simple to understand, reason about, and implement. As such, we leverage existing encoding formats and checksums to allow for code reuse.

1.3 Related work

Some works related to this document are:

- DEFLATE: The DEFLATE algorithm specified in RFC 1951 and originally designed by Phil Katz is the source of this document's discussion.
- Zlib: The zlib library written by Mark Adler and Jean-Loup Gailly is the standard reference implementation for DEFLATE. In creating XFLATE, it was absolutely required that XFLATE be properly decodable by zlib.
- Gzip: The Gzip format specified by RFC 1952 uses DEFLATE internally, but provides no specified way of implementing random access decompression.
- BGZF: The BGZF format is an extension to Gzip that uses the "extra field" of Gzip to encode size information. Unless the index is stored externally, "random access" is achieved through following a linked-list of size offsets.
- DictZip: A Gzip extension using the "extra field" and uses 64 KiB chunks. It embeds an index, but has a 1.8 GiB max limit. Since the index is stored at the beginning, a streamable output is not practical since the contents of the index are determined *after* compressing the data.
- GZinga: A Gzip extension using the "comment field" to encode an index as the last Gzip file in the stream. This format achieves similar goals, but is still limited to the Gzip format. Also, we have concerns about the abuse of the "comment field" to store machine interpreted data.
- JZRan: Library that provides random access to any Gzip stream by literally storing the state of the decompressor at specific points. This approach is space inefficient.
- XZ: The XZ format, developed by Lasse Collin, is a modern format that uses a different compression algorithm than DEFLATE. It was designed to provide random access capabilities. The structure of XZ had a significant influence on the design of XFLATE.

2 Specification

This section assumes some knowledge of the DEFLATE format as specified in RFC 1951. Some review information is presented here as reference. Please refer to RFC 1951 as the absolute authority for DEFLATE specifics.

In the format specifications below, we use regular expression-like semantics to describe the structure. As with the POSIX standard for regular expressions, we use the following operators:

- Grouping: `()`
- Alternatives: `|`
- Quantification: `*` `+` `?` `{n}` `{n,m}`

2.1 Stream format

The XFLATE stream format is as follows:

Symbol		Expression
XflateStream	:=	StreamBlock* StreamFooter
├─**StreamBlock**	:=	MacroBlock* Index
│ ├─**MacroBlock**	:=	DeflateBlock* SyncBlock
│ └─**Index**	<=	IndexHeader IndexRecord* IndexCRC
│ ├─**IndexHeader**	:=	BackSize NumRecords TotalCompSize TotalRawSize
│ └─**IndexRecord**	:=	CompSize RawSize
└─**StreamFooter**	<-	Magic Flags BackSize

In the grammar above, the color-coding of the variables has the following meaning:

- Black: Represents some other symbol
- Purple: Represents data compressed using DEFLATE
- Green: Represents data encoded using meta blocks
- Orange: Represents values with a fixed byte length
- Blue: Represents values encoded using variable-length integers (VLI)

Furthermore, the grammar uses several different equivalence operators:

- `:=`: The left side is literally equivalent to the right side
- `<-`: The left side is equivalent to the right side encoded into a *single* meta block
- `<=`: The left side is equivalent to the right side encoded into *one or more* meta blocks
- `#=`: The left side is equivalent to the CRC-32 hash of the right side.

Assuming that meta blocks and sync blocks produce absolutely no data when decompressed, one can see that an XFLATE stream is effectively equivalent to a series of regular DEFLATE blocks:

```
XflateStream := StreamBlock* StreamFooter
             := (MacroBlock* Index)* StreamFooter
             := ((DeflateBlock* SyncBlock)* Index)* StreamFooter
             :≈ DeflateBlock*
```

In the sections to follow, we will describe each element and field in detail. For the time being, assume that there exists a function that encodes a sequence of arbitrary bytes into byte-aligned meta blocks, which when decompressed by a DEFLATE decoder, produces no uncompressed output. The specification for the meta block encoding will be discussed in detail in a later section.

2.1.1 MacroBlock

The macro block has the following format:

```
MacroBlock := DeflateBlock* SyncBlock
```

The MacroBlock is the primary means to encode compressed data. Each macro block must be compressed independently of each other. Since DEFLATE is functionally a combination of LZ77 and Huffman encoding, this means that each macro block may only use LZ77 distances that refer to data within the given macro block; it may not reference data emitted by a preceding macro block. This ensures that each macro block has no data dependencies on previous blocks. The presence of the SyncBlock ensures that macro blocks always start and end on byte-aligned offsets.

2.1.1.1 DeflateBlock

The DeflateBlock* section is comprised of zero or more DEFLATE blocks as emitted by a standard DEFLATE compressor. Other than the distance requirement outlined above, there are no restrictions on what the block may actually be. Note that meta blocks must be composed of regular DEFLATE blocks themselves. Thus, there is *no* requirement that the DEFLATE blocks used in the DeflateBlock to *not* be composing meta blocks themselves (even if accidentally). Of course, the block still must be RFC 1951 compliant, which means that the final bit (RFC 1951, section 3.2.3) must not be set since DeflateBlock is never the last block in the XFLATE stream.

2.1.1.2 SyncBlock

The SyncBlock symbol represents an empty raw DEFLATE block, which has the property of ending on a byte boundary. This block is required even if the preceding sequence of DeflateBlocks already ends on a byte-aligned edge. This block always ends with the 4-byte string: [0x00, 0x00, 0xff, 0xff].

This byte sequence aids in parallel decompression when reading the compressed input as a stream. A decompressor may choose to buffer a large quantity of compressed input and search for the occurrence of this sequence and speculatively decompress from the position following that sequence. Care must be taken since the presence of this sequence does not guarantee the termination of a MacroBlock as this sequence may occur naturally in a DEFLATE stream. The decompressor may only release the speculatively decompressed data if the real offset has caught up with the speculated sync offset.

2.1.2 Index

The index has the following format:

```
Index <= IndexHeader IndexRecord* IndexCRC
```

The index stores size information regarding all of the MacroBlocks that precede the index within the same StreamBlock. It is recommended that there only be one index per XFLATE stream, but this may not be possible due to the memory requirements of holding a potentially gigantic index.

2.1.2.1 IndexHeader

Grammar format:

```
IndexHeader := BackSize NumRecords TotalCompSize TotalRawSize
```

The BackSize is the literal size in bytes of the preceding meta-encoded Index. If this is the first index, then the size is zero. This causes all indexes to form a reverse linked list such that a reader can locate all other Indexes in the XFLATE stream starting from the StreamFooter. Supposing that a preceding index exists, it will be located at $OffsetOf(Index) - TotalCompSize - BackSize$.

The purpose of allowing several Index blocks to be chained together is to ensure that there is a bounded amount of memory needed to remember the index information. If the index table is becoming too large for an encoder to maintain in memory, it can flush out the current index and only needs to keep track of the literal size of the recently flushed Index.

The NumRecords is the number of IndexRecord objects in the Index and must also be equal to the number of MacroBlocks that precede the current index in the same StreamBlock. Each IndexRecord uniquely corresponds to a MacroBlock. The first IndexRecord corresponds with the first MacroBlock, the second IndexRecord corresponds with the second MacroBlock, and so on.

The TotalCompSize is the literal size in bytes of the compressed data representing the MacroBlock* section of the same StreamBlock. Since the previous sections are guaranteed to be byte-aligned, we do not need to worry about lengths in bits. The sum of the CompSize across all IndexRecords in the index must be equal to the TotalCompSize field.

The TotalRawSize is the size in bytes of the MacroBlock* section when it has been decompressed. The sum of the RawSize across all IndexRecords in the index must be equal to the TotalRawSize field.

By summing up the TotalCompSize and TotalRawSize across all indexes, a decoder can quickly determine the amount of compressed and uncompressed data that the entire XFLATE stream represents. It also allows for computing the compression ratio rather efficiently.

2.1.2.2 IndexRecord

Grammar format:

```
	IndexRecord := CompSize RawSize
```

The CompSize is the literal size in bytes of the corresponding MacroBlock when compressed.
The RawSize is the size in bytes of the uncompressed data in the corresponding MacroBlock.

2.1.2.3 IndexCRC

The IndexCRC is a 4-byte CRC-32 hash completed over the other fields in the Index. Specifically, it covers the following fields:

```
	IndexCRC #= IndexHeader IndexRecord*
```

The CRC-32 implementation used is ITU-T V.42 and is the same as what is used in Gzip. The CRC-32 polynomial used in that standard is 0xedb88320. For reference, see RFC 1952, section 8. The value is stored in little endian byte order.

2.1.3 StreamFooter

The stream footer has the following format:

```
	StreamFooter <- Magic Flags BackSize
```

The Magic is the 2-byte string: ['X', 'F'].

The `Flags` symbol represents a single byte value where each bit represents the presence of certain features of the format. For the current version of XFLATE, there are no features defined, so the value of the flags should be `0x00`. A reader must error if these bits are not zero.

The `BackSize` in the footer represents the literal number of compressed bytes occupied by the last `Index` preceding and adjacent to the `StreamFooter`. If no index exists, then the size is set to 0. The `BackSize` exists so that a decoder can read the footer and seek to the index since it will lie at $OffsetOf(StreamFooter) - BackSize$. The `BackSize` is encoded as a variable-length integer (VLI).

The requirement that the `StreamFooter` be encoded as a single meta block is to aid the reader in identifying the start of the footer when reading the stream from the end. Meta blocks have the property that it can encode at least 22 bytes in a single block, that the encoded block occupies at most 64 bytes, and that all blocks are identifiable by a magic sequence that cannot occur in the encoded output itself. Being able to encode at least 22 bytes means that there is sufficient space to store the entire footer. In order to locate the start of `StreamFooter`, a reader needs to read the last 64 bytes of the stream and do a reverse search for first occurrence of the meta block magic sequence.

In order to be compliant with DEFLATE, the `StreamFooter` block must have the final bit (RFC 1951, section 3.2.3) set to indicate that it is the last block in the DEFLATE stream. Logically, we can conclude that this is the only block with the final bit set since the stream footer is always present and is the last element. Thus, it is invalid for the final DEFLATE bit to be set on any other elements in XFLATE.

2.2 Meta block encoding

The backbone of what enables XFLATE to be possible is the meta block encoding, which we now discuss in detail. The key to encoding metadata into the stream is to somehow generate DEFLATE blocks that decompress to nothing, yet allow for the encoding of arbitrary metadata into that block. Unfortunately, DEFLATE does not provide for any form of “meta” type block. Rather, in DEFLATE there are 3 types of blocks: non-compressed blocks, compressed blocks with fixed Huffman codes, and compressed blocks with dynamic Huffman codes. In generating meta blocks, we utilize the compressed blocks with dynamic Huffman codes, which we call “dynamic blocks“ for short. The meta block is actually a subset of valid dynamic blocks. In short, they are dynamic blocks with metadata encoded into the Huffman code definition, with no actual compressed data. This section will describe the grammar for meta blocks and will not explain why they are valid dynamic blocks. An explanation about their validity as dynamic blocks can be found in Appendix C.1.

In the specifications below, when dealing with bit-strings, they are to be interpreted in the same way as DEFLATE (RFC 1951, section 3.1.1). Unless otherwise specified, any bit-strings that appear in this document have the MSB (most-significant bit) on the left and the LSB (least-significant bit) on the right. When several bit-strings are being joined, they are packed together starting with the LSB first. Here is an example of packing bit-strings into a byte-array:

1.	`{0111 100111 110 001 0 1 101011}`	Start with LSB on right
2.	`{1110 111001 011 100 0 1 110101}`	Reverse bits so that LSB is on the left
3.	`{11101110 01011100 01110101}`	Group bits into bytes, with LSB on left
4.	`{01110111 00111010 10101110}`	Reverse bits so that LSB is on the right
5.	`{0x77 0x3a 0xae}`	Convert bits to bytes

The MetaBlock format is as follows (where a · matches either a 0 or a 1):

Symbol		Expression
MetaBlock	:=	MetaHeader MetaBody MetaFooter
MetaHeader	:=	(· 10) (00··· 00000 ···0) (011 000 011 001 000 (000 000){7-HuffBits} 010) 0
MetaBody	:=	(0|01|011 ··|111 ·······)*
MetaFooter	:=	0{Padding} 0 1{HuffBits}

2.2.1 MetaHeader

Grammar format:

```
    MetaHeader := (· 10) (00··· 00000 ···0) (011 000 011 001 000 (000 000){7-HuffBits} 010) 0
```

In the grammar for the MetaHeader, there are three groups of unspecified bits. From left to right, these correspond to the encodings of the FinalBlock, Padding, and HuffBits fields.

The FinalBlock field is a single bit indicating whether this is the last block in the XflateStream. Since the StreamFooter is always the last block (as mentioned in section 2.1.3), this is a 1-bit only for that element.

The Padding field is a 3-bit unsigned integer with a value within the range of $[0..7]$. The padding field determines how many 0-bits to encode in the MetaFooter such that the entirety of the MetaBlock falls on a byte boundary.

The HuffBits field is a 3-bit unsigned integer representing the value $8 - HuffBits$ where $HuffBits$ is in the range of $[1..7]$. $HuffBits$ cannot be 8, so 000 is not a valid value for HuffBits. The semantic meaning of this field will be explored in the following section.

2.2.2 MetaBody

Grammar format:

```
    MetaBody := (0|01|011 ··|111 ·······)*
```

The MetaBody is a variable-length bit-string, which is transformed into a 256-bit intermediate bit-string, which is finally transformed into a metadata byte string (between 0 to 31 bytes long).

The MetaBody is a sequence of prefix codes of four possible codes: 0, 01, 011, and 111. The different codes can be differentiated from each other by parsing the MetaBody bit-by-bit (starting with the LSB) and checking if the current token matches one of the prefix codes.

Processing each code in the MetaBody works as follows:

- A 0 code appends a 0-bit to the intermediate buffer.
- A 01 code appends a 1-bit to the intermediate buffer.
- A 011 code copies the last bit in the intermediate buffer N times. If the intermediate buffer is empty, then the initial "last" bit is a 0-bit. The repeat length N is decoded as 3 plus the value of the next 2-bit unsigned integer in the MetaBody.
- A 111 code appends a 0-bit N times to the intermediate buffer. The repeat length N is decoded as 11 plus the value of the next 7-bit unsigned integer in the MetaBody.

Parsing of the MetaBody continues until the intermediate buffer is exactly 256 bits long. It is an error if a repeater code causes more than 256 bits to be decoded.

This table summarizes the semantics of each code:

Code		Extra Bits	Count	Value
0		0	1	0
01		0	1	1
011		2	3..6	*Last*
111		7	11..138	0

When processing the MetaBody into the intermediate bit-string, the following constraints must hold:

- A sequence of 8x 0-bits (i.e., 00000000) must never appear in the MetaBody bit-string.
- The number of 1-bits in the intermediate bit-string must exactly equal $2^{HuffBits}$.
- The last bit in the intermediate bit-string must be a 1-bit.

A decoder must check that these properties are upheld, while an encoder must not violate these.

2.2.2.1 Intermediate bit-string

The intermediate bit-string (exactly 256-bits long) follows the following grammar:

```
    InterBits := FinalMeta Invert Size (········){31} 1
```

The FinalMeta field is a single bit indicating whether the current MetaBlock is the last in a sequence of MetaBlocks. Since each MetaBlock can encode at most 31 bytes of metadata, it is necessary to use more than one block to encode longer strings of metadata. The Index may be composed of more than one block, but the StreamFooter must be composed of exactly one MetaBlock. Thus, the StreamFooter is the only MetaBlock with both the FinalBlock and FinalMeta bits set.

The Invert field is a single bit indicating whether the bits of every byte in the decoded metadata of this MetaBlock should be inverted. An encoder may set this bit when encoding a metadata string with many 1-bits to help satisfy the constraint regarding the total number of 1-bits.

The Size field is a 5-bit unsigned integer, N, representing the number of bytes of metadata; the bytes of which are composed of the next N octets (possibly inverted) following the Size field. Even if there is more metadata to encode, the size may be less than 31 so that the unused bits may be utilized to satisfy the constraint regarding the total number of 1-bits.

2.2.3 MetaFooter

Grammar format:

```
    MetaFooter := 0{Padding} 0 1{HuffBits}
```

The value of $Padding$ must be one such that the MetaFooter ends on a byte boundary and should only be composed of 0-bits. A decoder must verify that this is true, while an encoder must choose a value for $Padding$ such that this holds true. As such, an encoder will typically choose the value of $Padding$ once all other fields are known.

2.2.4 Properties

The meta block format was designed to guarantee certain properties to aid a reader in locating all the independently compressed chunks with minimal effort.

Every MetaBlock starts on a byte-boundary and can be identified by a unique magic sequence:

```
	MagicVals = [0x04, 0x00, 0x86, 0x05]
	MagicMask = [0xc6, 0x3f, 0xfe, 0xff]
```

The start of a MetaBlock at an arbitrary file offset can be identified by performing a logical AND of the next 4 bytes with the MagicMask and checking if it equals MagicVals. The uniqueness of this magic sequence only applies when searching for the start of a MetaBlock from a binary section representing only MetaBlocks. It is possible (however unlikely) that the magic sequence appear in DeflateBlocks.

Futhermore, every MetaBlock has these byte size properties:

Property	Value	Description
MinRawBytes	0	Minimum and maximum number of metadata bytes a block can encode
MaxRawBytes	31	
EnsureRawBytes	22	Number of bytes that a single block is ensured to encode
MinEncBytes	12	Minimum and maximum number of bytes an encoded block will occupy
MaxEncBytes	64	

The existence of a magic value and guaranteed max encoded size allows a decoder to parse the StreamFooter by reading the last 64 bytes in an XflateStream, searching in reverse for the magic value and decoding the discovered MetaBlock. The guarantee of being able to encode at least 22 bytes in a single MetaBlock gives considerable margin for encoding the StreamFooter as a single block.

2.3 Variable-length integers

Some integer fields are encoded using variable-length integers (VLI). The format used is identical to the VLI format specified by the XZ format (section 1.2). In summary, it provides the following properties:

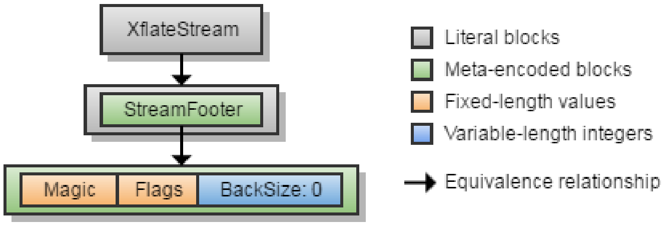
- Encodes any integer in the range of $[0..MaxUint64/2\,]$
- Occupies between $[1..9]$ bytes
- Smaller values encode into fewer bytes

Appendix A: Example XFLATE streams

For illustrative purposes, here are some examples of valid XFLATE streams.

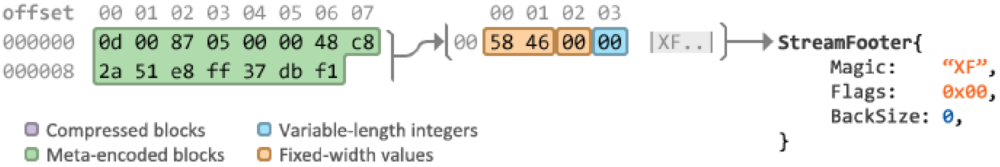
A.1 Empty stream

The simplest `XflateStream` is one that decompresses to absolutely no data.



Note that this stream still requires a `StreamFooter`, but the `BackSize` is always zero to indicate that there are no indexes present and, thus, has no uncompressed data.

There are actually several ways to represent the empty stream since there are multiple ways of performing the meta encoding for the footer. Below is the hex-dump of one such representation:



```
offset  00 01 02 03 04 05 06 07             00 01 02 03
000000  0d 00 87 05 00 00 48 c8         00  58 46 00 00   |XF..|  StreamFooter{
000008  2a 51 e8 ff 37 db f1                                           Magic:     "XF",
                                                                       Flags:     0x00,
                                                                       BackSize:  0,
                                                                  }
```

The column on the left is a hex-dump of a valid empty XFLATE stream. It consists of only a single meta block of which a hex-dump of the decoded metadata is shown in the middle column. As expected, the magic marker is clearly visible in the ASCII printout. The third column is a structured representation of the metadata after it has been parsed.

A.1.1 Decoding a MetaBlock

The `MetaBlock` from the above example is deconstructed as follows (LSB on the right):

```
MetaHeader := (1 10) (00001 00000 1000) (011 000 011 001 000 (000 000){3} 010) 0
MetaBody   := 01 0 0 0 01 0 (011 01) 01 01 0 01 0 0 01 01 0 0 0 01
              (111 1111111) (111 1001101) 01 (011 11) 01
MetaFooter := 0{1} 0 1{4}
```

With these decoded values:

```
FinalBlock: true
Padding:    1
HuffBits:   4
```

The `MetaBody` can be further deconstructed into the following intermediate bit-string:

```
InterBits  := 1 0 0 0 1 0 0{4} 1 1 0 1 0 0 1 1 0 0 0 1 0{138} 0{88} 1 1{6} 1
```

Which can be grouped by the relevant bit fields (LSB on the right):

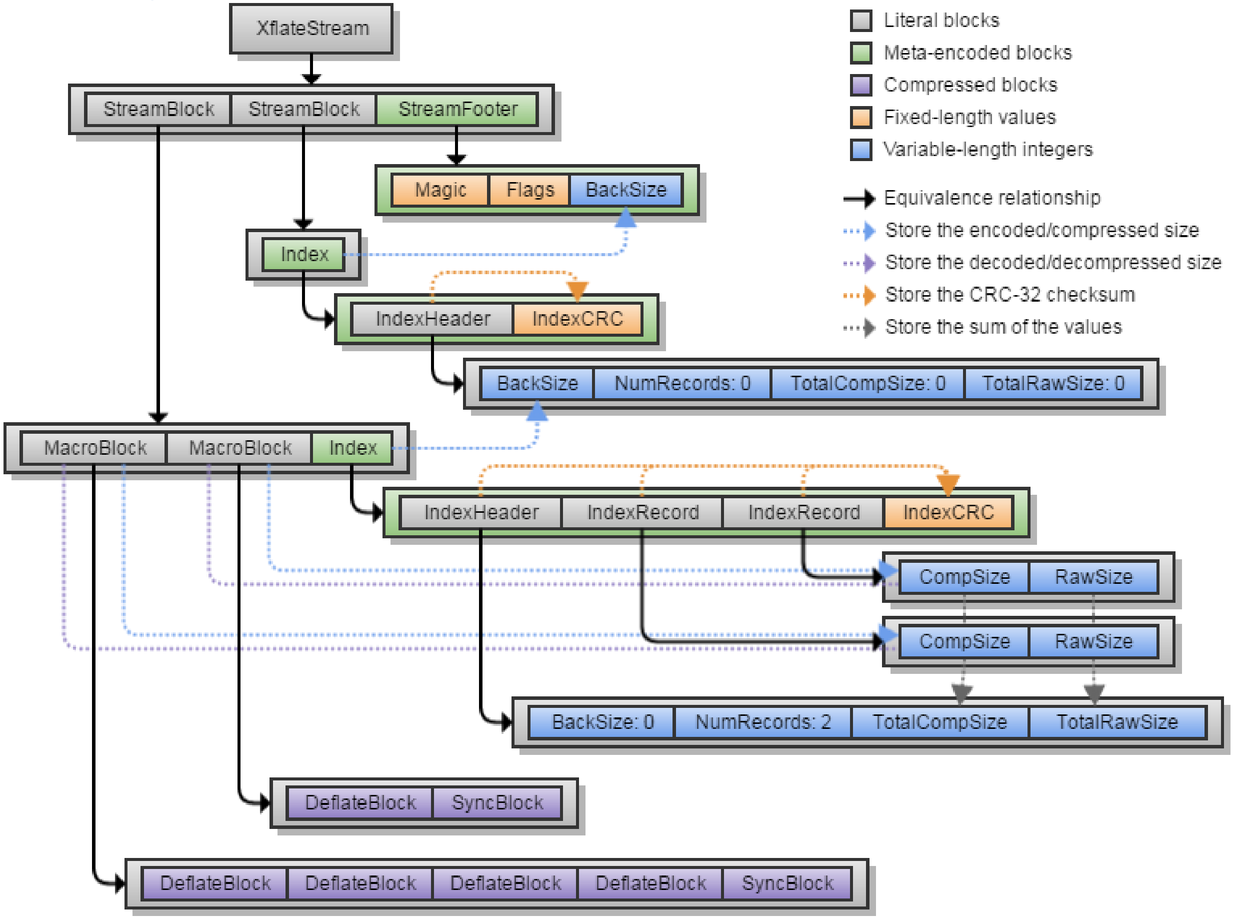
```
InterBits  := 1 0 00100 01011000 01000110 (00000000){28} 11111110 1
```

With these decoded values:

```
FinalMeta: true
Invert:    false
Size:      4
MetaData:  [0x58, 0x46, 0x00, 0x00]
```

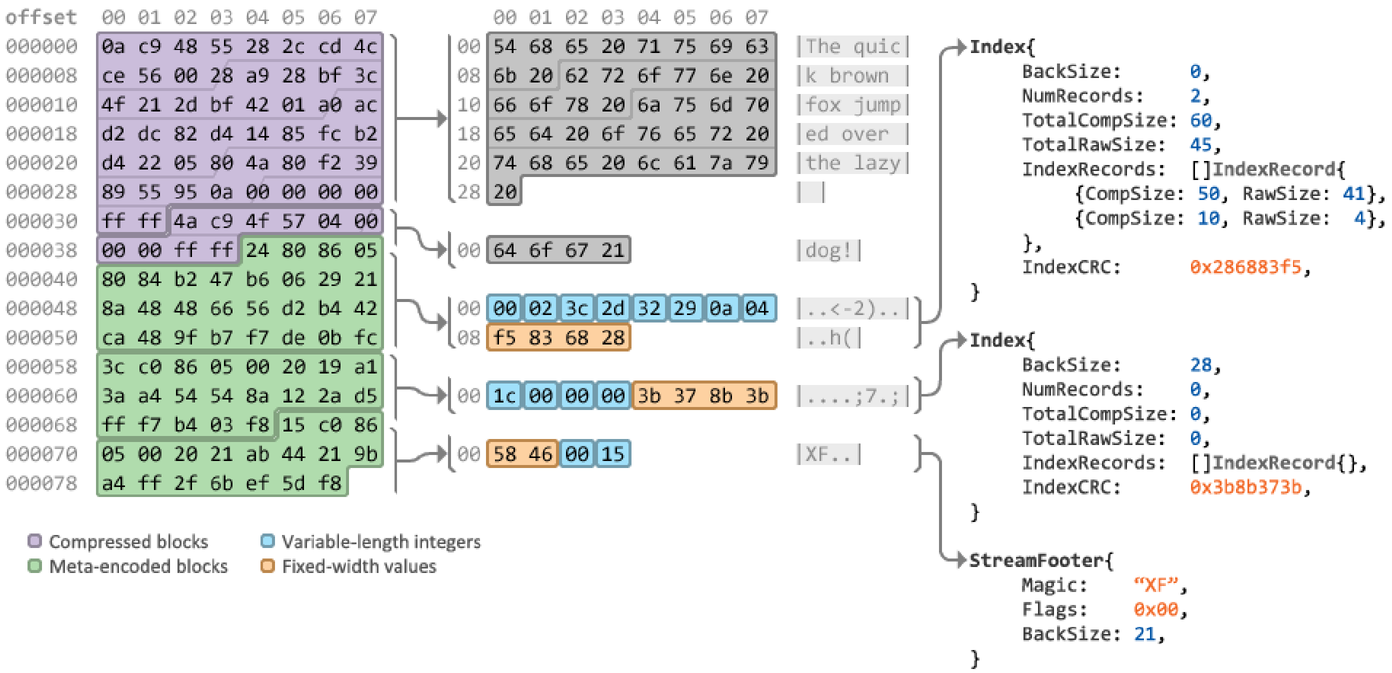
We note that there are exactly $2^{HuffBits} = 2^4 = 16$ `1`-bits in `InterBits` and that the last bit is a `1`-bit.

A.2 Multiple indexes in stream



The diagram above shows an exploded view of another `XflateStream`. This stream is comprised of two `StreamBlocks`. The first stream block contains actual data, while the second stream block contains no data. An efficient encoder would avoid outputting a `StreamBlock` with no uncompressed data, but this was done for the purpose of illustration. Notice how a decoder can completely back-trace all of the blocks starting from the `StreamFooter` by following all of the `Size` fields.

Below, we show the hex-dump of a valid XFLATE stream that actually follows the structure shown above. Similar to the previous hex-dump, the left column displays the bytes of the actual XFLATE stream. The two purple regions represent DEFLATE compressed `MacroBlocks`. In each macro block, the distinctive `0x0000ffff` bytes of the sync marker can be clearly seen. The three green regions following the macro blocks represent the two `Indexes` shown above and the `StreamFooter`. Since the macro blocks precede the first index, they belong to that index. There are no macro blocks preceding the second index, so that index contains no records.



```
offset  00 01 02 03 04 05 06 07
000000  0a c9 48 55 28 2c cd 4c
000008  ce 56 00 28 a9 28 bf 3c
000010  4f 21 2d bf 42 01 a0 ac
000018  d2 dc 82 d4 14 85 fc b2
000020  d4 22 05 80 4a 80 f2 39
000028  89 55 95 0a 00 00 00 00
000030  ff ff 4a c9 4f 57 04 00
000038  00 00 ff ff 24 80 86 05
000040  80 84 b2 47 b6 06 29 21
000048  8a 48 48 66 56 d2 b4 42
000050  ca 48 9f b7 f7 de 0b fc
000058  3c c0 86 05 00 20 19 a1
000060  3a a4 54 54 8a 12 2a d5
000068  ff f7 b4 03 f8 15 c0 86
000070  05 00 20 21 ab 44 21 9b
000078  a4 ff 2f 6b ef 5d f8
```

```
    00 01 02 03 04 05 06 07
00  54 68 65 20 71 75 69 63  |The quic|
08  6b 20 62 72 6f 77 6e 20  |k brown |
10  66 6f 78 20 6a 75 6d 70  |fox jump|
18  65 64 20 6f 76 65 72 20  |ed over |
20  74 68 65 20 6c 61 7a 79  |the lazy|
28  20                       | |

00  64 6f 67 21              |dog!|

00  00 02 3c 2d 32 29 0a 04  |..<-2)..|
08  f5 83 68 28              |..h(|

00  1c 00 00 00 3b 37 8b 3b  |....;7.;|

00  58 46 00 15              |XF..|
```

```
Index{
    BackSize:      0,
    NumRecords:    2,
    TotalCompSize: 60,
    TotalRawSize:  45,
    IndexRecords:  []IndexRecord{
        {CompSize: 50, RawSize: 41},
        {CompSize: 10, RawSize:  4},
    },
    IndexCRC:      0x286883f5,
}
Index{
    BackSize:      28,
    NumRecords:    0,
    TotalCompSize: 0,
    TotalRawSize:  0,
    IndexRecords:  []IndexRecord{},
    IndexCRC:      0x3b8b373b,
}
StreamFooter{
    Magic:    “XF”,
    Flags:    0x00,
    BackSize: 21,
}
```

The column in the middle shows the decompressed output for the macro blocks (in gray) and the decoded output of the meta blocks (in blue and orange). The column on the right shows a structured representation of the middle column after the decoder has parsed it.

In this example, the string `"The quick brown fox jumped over the lazy dog!"` is being compressed (although not very efficiently). Looking at the compressed representation on the left column, one can verify that the total compressed size is 60 bytes, split into a section of 50 bytes (offsets `0x0000` to `0x0032`) in the first macro block and 10 bytes (offsets `0x0032` to `0x003c`) in the second macro block. Similarly, by looking at the decompressed representation in the middle column, one can verify that there are indeed 45 bytes in that string, split into a section of 41 bytes in the first macro block and 4 bytes in the second macro block.

Lastly, one can verify the correctness of the back sizes. Starting with the footer, we can clearly see that the index preceding it does indeed occupy 21 bytes (offsets `0x0058` to `0x006d`). Looking at the last index now, we can see that the index preceding that one does indeed occupy 28 bytes (offsets `0x003c` to `0x0058`). Since we are now at the first index, the back size that it contains is 0 bytes, indicating that there are no more preceding indexes. Even encoding an empty index occupies at least 8 metadata bytes (4 bytes of VLIs and a 4 byte CRC), and thus occupies at least some positive number of meta-encoded bytes. This guarantees that use of a 0 byte size is a legal sentinel value for termination.

In order to keep this example small, none of the VLIs have large enough values to span multiple bytes.

Appendix B: Analysis of XFLATE format

Since the XFLATE format was designed to provide random access decompression, there is some overhead that causes an XFLATE stream to be larger than if the source had been compressed as a DEFLATE stream instead. The sources of overhead come from compression inefficiency due to chunking and overhead due to storing the index tables.

B.1 Effect of chunk size

The factor that has the most effect on the overhead of XFLATE over standard DEFLATE is the choice of chunk size. Since each chunk is independently compressed, they cannot benefit from potential LZ77 matches on sub-strings found in prior chunks. Using the DEFLATE compressor from Go1.4.2, we measure the overhead of chunking on various test files:

- **`go1.4.2.linux-amd64.tar`**: TAR archive containing both binary and text data
- **`enwik8.txt`**: Text file containing primarily English articles
- **`zeros.bin`**: 1GiB file of all zeros that is highly compressible
- **`sawtooth.bin`**: 1GiB file of repeating sequences of bytes iterating from 0 to 255

File		go1.4.2.linux-amd64.tar				enwik8.txt		
RawSize		233581568				100000000		
ChunkSize		65536	262144	1048576		65536	262144	1048576
#Chunks		3565	892	223		1526	382	96
CompSize$_{\text{STREAM}}$		62442686				36523872		
CompSize$_{\text{CHUNKED}}$		64256397	62911473	62553140		38134384	36940045	36628142
%Overhead		2.90%	0.75%	0.18%		4.41%	1.14%	0.29%

File		zeros.bin				sawtooth.bin		
RawSize		1073741824				1073741824		
ChunkSize		65536	262144	1048576		65536	262144	1048576
#Chunks		16384	4096	1024		16384	4096	1024
CompSize$_{\text{STREAM}}$		1043610				4165513		
CompSize$_{\text{CHUNKED}}$		1359877	1122309	1061893		9502720	5496832	4495360
%Overhead		30.31%	7.54%	1.75%		128.13%	31.96%	7.92%

CompSize$_{\text{STREAM}}$ represents the compressed size when encoded as a single DEFLATE stream, while **CompSize$_{\text{CHUNKED}}$** represents the compressed size when encoded with individually compressed chunks of the specified size. The **%Overhead** computes the percentage increase of output size of the chunked version relative to the single stream version. This overhead does not account for the space required to store any indexing metadata.

From the above table, we make some observations:

- Inputs that benefit most from LZ77 matches have the greatest overhead. Thus, applying chunking on **`sawtooth.bin`** is an effective measure of the worst-case overhead since it has almost no benefit from Huffman encoding and relies almost entirely on LZ77 for compression. The overhead for **`zeros.bin`** is not as extreme since it still benefits significantly from Huffman coding.

- As the chunk size increases, the overhead decreases since there is more data for LZ77 to take advantage of for compression. The choice of chunk size allows a user to balance the trade-offs between random access efficiency and compression overhead.
- A scheme that uses dynamic chunk sizes may be able to better balance the trade-off between overhead and random-accessibility. A larger chunk size may be chosen for highly compressible data with the assumption that this data is faster to compress and decompress.

B.2 Trade-offs in index format design

Another source of overhead is the storage of the index itself into the XFLATE stream. Several techniques were considered to make the encoded size of the index smaller:

- *Compression:* Compress the index itself using DEFLATE. Since XFLATE is already an extension on the DEFLATE format, it should not be difficult to call a DEFLATE encoder on the index.
- *Column-oriented layout*: The `RawSize` in every record is likely to be the same (assuming fixed chunk sizes). If we group the `CompSizes` of every record together, followed by the `RawSizes` of every record, we can gain a compression benefit due to adjacent repetition.
- *Variable-length integers (VLIs)*: Use variable-length encoding for integers such that smaller values require fewer bytes to encode as opposed to using a `uint64` for all integers.
- *Delta-encoding*: The `CompSize` and `RawSize` usually have values in the same range, we can compute the difference between the current value and the previous value and only store the delta, which is likely to be smaller. This technique works best with the use of VLIs.

To test the effectiveness of these techniques, indexes based on **`go1.4.2.linux-amd64.tar`** with 3565 chunks were generated using various combinations of the techniques:

Mode		Compres	Columns	VLIs	Delta		RawSize	MetaSize
Fixed							57040	41032
VLIs				✔			19510	30824
Delta				✔	✔		10829	17867
CompFixed		✔					10145	18927
CompVLIs		✔		✔			9636	17340
CompDelta		✔		✔	✔		8315	15002
ColumnFixed		✔	✔				9670	18298
ColumnVLIs		✔	✔	✔			7809	14388
ColumnDelta		✔	✔	✔	✔		7109	12827

The middle columns indicate which techniques were used, while the **RawSize** is the byte size of the formatted index, and the **MetaSize** is the byte size after using meta encoding. As expected, using more techniques leads to a smaller raw size. In general, meta encoding the index results in an increase of about 1.6x to 1.8x, with the exception of the **Fixed** mode which was interestingly smaller due to the meta encoding's efficiency at handling long runs of `0`-bits.

The final design only used VLIs since it provided decent benefit, without adding the complexity of the other techniques. Furthermore, even with the inefficiencies of the meta encoding, the index overhead was much smaller than the chunking overhead; so minimal encoding was not a critical priority.

Appendix C: Analysis of meta encoding format

In the design of the meta encoding format, multiple approaches were explored that balanced aspects of simplicity and encoding efficiency. The specific format presented in section 2.2 was deemed an acceptable balance of simplicity and efficiency.

C.1 Validity of meta block as DEFLATE block

As mentioned, the `MetaBlock` format is a special case of dynamic blocks from DEFLATE. In order to demonstrate this, we first review the format of `DynamicBlocks` themselves. Please refer to RFC 1951 as the authoritative source.

C.1.1 Review of dynamic blocks

The `DynamicBlock` has the following grammar:

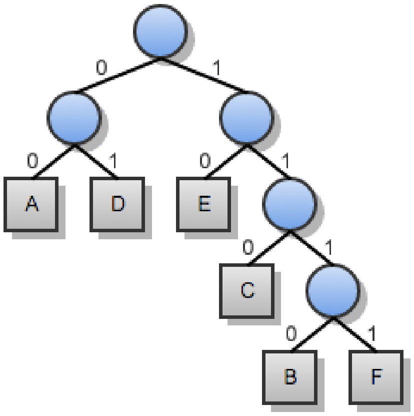
Symbol		Expression
DynamicBlock	:=	FinalBlock BlockType HuffHeader HuffTrees HuffData
├─HuffHeader	:=	NumLit NumDist NumHCLen
├─HuffTrees	:=	HCLens (LitLens DistLens)
│ ├─HCLens	:=	HCLen{NumHCLen+4}
│ ├─LitLens	:=	(HCCode Extra?){1,NumLit+257}
│ └─DistLens	:=	(HCCode Extra?){1,NumDist+1}
└─HuffData	:=	(LCode Extra? (Dcode Extra?)?)* LCodeEOB

Symbol		Expression		Significance
FinalBlock	:=	0|1		Is this the last block in DEFLATE stream?
BlockType	:=	10		Dynamic Huffman tree block type
NumLit	:=	[01]{5}		Number of literal symbols - 257 in *LitTree*
NumDist	:=	[01]{5}		Number of distance symbols - 1 in *DistTree*
NumHCLen	:=	[01]{4}		Number of length symbols - 4 in *HCTree*
HCLen	:=	[01]{3}		Bit-length for each *HCTree* code type
Extra	:=	[01]+		Unsigned value that augments prior symbol

The uncompressed data in a `DynamicBlock` is entirely stored by the `HuffData` field. The `HuffData` is essentially a sequence of `LCode` or `DCode` values that represent either literal byte values or distance/length pairs, terminated by a special “end-of-block” code, `LCodeEOB`. Each code may be followed by some number of extra bits depending on the semantics of the preceding code (although most codes will have no extra bits). Symbols colored in beige are Huffman encoded symbols using their respective Huffman tree. That is, `HCCode` uses $HCTree$, `LCode` uses $LitTree$, and `DCode` uses $DistTree$.

Huffman trees are a method of constructing a variable-length encoding such that symbols with more frequent occurrences can be assigned a shorter bit-string code. Huffman trees are essentially a data structure that maps some domain of alphabet symbols to a range of bit-string codes. The input range and output domain always have the same cardinality, and this conversion process is bijective (the mapping is reversible). When parsing any bit-stream bit-by-bit, it is possible to unambiguously determine the sequence of symbols that were used to generate that bit-stream.

An example Huffman tree is as follows:



Symbol		Code
A		00
B		0111
C		011
D		10
E		01
F		1111

The tree on the left and the table on the right represent the same mapping. There are 6 alphabet symbols $[A..F]$ being mapped to 6 distinct bit-string codes. The tree encodes the symbols as the leaves, while the codes are encoded as the path from the root node (the LSB) to each leaf (the MSB). The Huffman encoding used in DEFLATE is canonical, meaning that the tree is constructed in such a way that only the bit-lengths of each symbol is needed to reconstruct the tree. Thus, the tree in the example can be reconstructed only using the sequence of bit-lengths: $[2, 4, 3, 2, 2, 4]$.

Similarly, the $HCTree$, $LitTree$, and $DistTree$ are all constructed from a list of bit-lengths specified in the `HCLens`, `LitLens`, and `DistLens` fields. The number of symbols represented by these lists are stored in the `NumHCLen`, `NumLit`, and `NumDist` fields; they can take on a value in the ranges of $[257..286]$, $[1..30]$, and $[4..19]$, respectively. The $LitTree$ contains symbols needed to encode literal byte values (256 symbols), an end-of-block symbol, and also various copy lengths (up to 29 symbols). The $DistTree$ contains symbols needed to encode the copy distance (up to 30 symbols). The definitions of the $LitTree$ and $DistTree$ themselves are actually encoded using another mapping, $HCTree$ (up to 19 symbols).

The symbols of $HCTree$ are $[RL, R0a, R0b, 0, 8, 7, 9, 6, 10, 5, 11, 4, 12, 3, 13, 2, 14, 1, 15]$. The bit-lengths for each code are stored in `HCLens`. If the length of `HCLens` is shorter than 19, then the alphabet is right truncated to match in length. Each `HCLen` value is 3-bits long, meaning each `HCCode` can be up to 7-bits long. If the `HCLen` value is 0, then the associated symbol is not in $HCTree$. The symbols $[0..15]$ represent bit-lengths for codes in $LitTree$ and $DistTree$, while symbols RL, $R0a$, and $R0b$ are special symbols used to repeat other symbols. Code RL is used to repeat the last non-special symbol $[3..6]$ times. Code $R0a$ is used to repeat a zero symbol $[3..10]$ times. Code $R0b$ is used to repeat a zero symbol $[11..138]$ times. The repeat count is stored immediately after each repeater code, occupying 2, 3, and 7 bits respectively and added to the lower bound of the repeat range.

$HCTree$ is used to encode the bit-lengths of $LitTree$ and $DistTree$ in the `LitLens` and `DistLens` fields. Since the largest symbol in $HCTree$ is 15, that means each `LCode` and `DCode` can be up to 15-bits long. The use of repeater symbols allows the $LitTree$ and $DistTree$ to be populated efficiently without needing to specify a single code for every mapped symbol. It is important to note that the list of bit-length for all trees must form a canonical Huffman tree; meaning that the bit-lengths form a valid binary tree where each node has exactly 2 children.

We do not review how LZ77 sub-string matching operates with respect to $LitTree$ and $DistTree$ since that is not necessary to explain how `MetaBlocks` work.

.

C.1.2 Meta blocks as a special case of dynamic blocks

Using the DynamicBlock as a template, we define the MetaBlock by hard-coding certain fields:

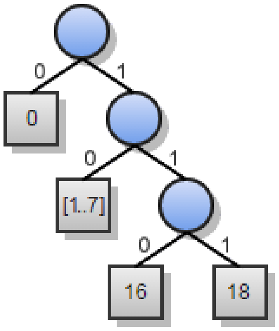
Symbol		Expression
MetaBlock	:=	FinalBlock 10 HuffHeader HuffTrees HuffData
├─HuffHeader	:=	(Padding 00) 00000 (0 HuffBits)
├─HuffTrees	:=	HCLens (LitLens DistLens)
│ ├─HCLens	:=	011 000 011 001 000 (000 000){7-HuffBits} 010
│ ├─LitLens	:=	0 (HCCode Extra?){1,256} 0{Padding}
│ └─DistLens	:=	0
└─HuffData	:=	LCodeEOB
└─LCodeEOB	:=	1{HuffBits}

Symbol		Expression		Significance
FinalBlock	:=	0|1		Is this the last block in DEFLATE stream?
Padding	:=	[01]{3}		Number of padding bits for byte-alignment
HuffBits	:=	[01]{3}		Controls the bit-depth of *LitTree*
HCCode	:=	0 | 01 | 011 | 111		Maps to *HCTree* symbols
Extra	:=	[01]{2} | [01]{7}		RL and R0b repeater counts

The following fields have fully or partially hard-coded values:

- The upper 2 bits of NumLit is hard-coded to 00, leaving the lower 3 bits as a new Padding field. This means that $NumLit$ can only take on values in the range of $[257..264]$.
- The lowest bit of NumHCLen is hard-coded to 0, leaving the upper 3 bits as a new HuffBits field, with a restriction that the field cannot be 000. This means that $NumHCLen$ can only take on values in the set of $[6, 8, 10, 12, 14, 16, 18]$. The HuffBits field stores $8 - HuffBits$.
- The HCLens list has several hard-coded values that creates this mapping of symbols to bit-lengths: $\{RL: 3, R0b: 3, 0: 1, X: 2\}$. The X symbol is in the range of $[1..7]$.
- The NumDist field is hard-coded such that $NumDist$ is 1. The DistLens field itself is hard-coded with a single HCCode representing the symbol 0, indicating that the $DistTree$ is empty.
- The HuffData field is hard-coded to immediately terminate with LCodeEOB. Thus, the MetaBlock decompresses to absolutely nothing at all.

The $HCTree$ is a Huffman tree with 4 leaves. The code that is 2 bits long has a symbol that is dependent on the HuffBits field and is equal to the $HuffBits$ value.



Symbol			Code
0	SymZero		0
[1..7]	SymOne		01
RL	SymRepLast		011
R0b	SymRepZero		111

We rename these symbols as SymZero, SymOne, SymRepLast, and SymRepZero to better identify how they are used to construct the intermediate bit-string described in section 2.2.2. The $HCTree$ has only two normal symbols: SymZero and SymOne. SymZero is used to indicate which symbols in the $LitTree$ are non-existant, while SymOne (which has a value of $HuffBits$) is used to specify which symbols in $LitTree$ do exist. Since we only have one bit-length value $HuffBits$, in order for the $LitTree$ to be a canonical Huffman tree, it must be a perfectly balance binary tree with a depth of $HuffBits$. This implies that exactly $2^{HuffBits}$ symbols in the tree must be a SymOne.

For the $LitTree$ symbols themselves, the 0th symbol must be encoded with a SymZero literal (for reasons explained later), but the 1st to 255th symbols may encoded by either SymZero or SymOne literals or using the repeater symbols, SymRepLast and SymRepZero. In order to ensure that LCodeEOB is a valid code, the 256th symbol in the $LitTree$ must be a SymOne. The 257th symbol and above must be SymZero literals as determined by the $Padding$ count. Since the padding contains only SymZeros, it has no effect on the $LitTree$ structure. This ensures that the code value for LCodeEOB is always 1{HuffBits}.

C.2 Uniqueness of magic marker

For this section, we will be writing bit-strings with the LSB on the left (contrary to the rest of the document) because it aids visually in what the bit-streams actually looks like. Since bit-streams are fundamentally packed into bytes, we delineate byte boundaries with a single space character.

From section 2.2.4, the magic marker is defined to be:

```
        MagicVals = [0x04, 0x00, 0x86, 0x05]
        MagicMask = [0xc6, 0x3f, 0xfe, 0xff]
```

The magic marker is checked by first ANDing some data bytes with MagicMask before checking it against MagicVals for equality. The AND operation is used to mask-out certain bits as irrelevant. If we were to write out the magic as a bit-string with the masked-out bits using a wildcard bit '·', then we obtain the bit-string labeled M below.

```
M:  ·01···00 000000·· ·1100001 10100000

B1: ·01···00 00000011 11100001 10100000 00000000 00000000 00000000 00000000 00000100 ·······p pppppp01
B2: ·01···00 00000001 11100001 10100000 00000000 00000000 00000000 00000001 00······ ······pp ppppp011
B3: ·01···00 00000010 11100001 10100000 00000000 00000000 00000000 0100···· ········ ·····ppp pppp0111
B4: ·01···00 00000000 11100001 10100000 00000000 00000000 000100·· ········ ········ ····pppp ppp01111
B5: ·01···00 00000011 01100001 10100000 00000000 00000100 ········ ········ ········ ···ppppp pp011111
B6: ·01···00 00000001 01100001 10100000 00000001 00······ ········ ········ ········ ··pppppp p0111111
B7: ·01···00 00000010 01100001 10100000 0100···· ········ ········ ········ ········ ·ppppppp 01111111
```

Furthermore, we have several bit-strings labeled B[1..7] that represent all of the bit-strings that a meta block could possibly take form as (where the number x in Bx is the $HuffBits$ used). The sections in red, green, and blue represents the MetaHeader, MetaBody, and MetaFooter. Some of the bits in the header are unknown because they represent the final DEFLATE bit and the padding bit-count. As for the body, all of the bits are unknown since the symbols used may be anything. Lastly, some of the bits in the footer are labeled with a 'p' bit. This indicates that these bits are for padding. Since padding bits may or may not be present, these bits can practically be treated as unknown bits '·' as well.

When matching the magic string M downwards with all of the Bx strings, it is clear that it always matches. This is expected since the magic value *should* match all meta blocks at the beginning. The important thing we want to show is that M *does not* match any parts of Bx when M is byte-shifted to the right. For example, if we were to byte-shift M right by one byte, we can see that it never matches Bx since the 3rd bit in M is a 1-bit, while the corresponding bit in Bx is always 0-bit.

For the MetaHeader, if we were to continue this process, we would see that M does not match any of the full bytes of the MetaHeader in Bx at any possible byte-shift offset. The closest possible match occurs with B6 at a right-shift of 3 bytes where the 18th bit of M is a 1-bit, while the corresponding bit from B6 is a 0-bit. The reason the MetaHeader (from section 2.2.1) ends with a 0-bit (which is a SymZero as the first symbol in the $LitTree$) is to prevent a match. For the MetaFooter, the last byte of M never matches the end of the footer since the last bit of M is a 0-bit, while the last bit of Bx is always a 1-bit.

The remaining task is to show that M never matches the MetaBody, which is difficult since the MetaBody is entirely composed of unknown bits. To address this, we observe that M contains a sequence of 8 consecutive zeros, which lies entirely within the body section. Thus, to ensure that M never matches the MetaBody, we developed the constraint in section 2.2.2 that 8x 0-bits never appear in the MetaBody. While this may prevent certain combinations of symbols in the body that would not have accidentally matched the magic, it is a relatively simple rule for a meta decoder to check for.

However, let use analyze whether this constraint makes the encoding of certain metadata impossible. Below we listed every possible way that 8x 0-bits can be generated in the MetaBody:

A.	0 0 0 0 0 0 0 0	8x SymZeros
B.	10 0 0 0 0 0 0 0	SymOne, followed by 7x SymZeros
C.	110 00 0 0 0 0 0	SymRepLast with count of 3x, followed by 5x SymZeros
D.	110 10 0 0 0 0 0 0 0	SymRepLast with count of 4x, followed by 7x SymZeros
E.	111 0000000 0	SymRepZero with count of 11x, followed by 1x SymZeros
F.	111 1000000 0 0	SymRepZero with count of 12x, followed by 2x SymZeros
G.	111 1100000 0 0 0	SymRepZero with count of 14x, followed by 3x SymZeros
H.	111 1110000 0 0 0 0	SymRepZero with count of 18x, followed by 4x SymZeros
I.	111 1111000 0 0 0 0 0	SymRepZero with count of 26x, followed by 5x SymZeros
J.	111 1111100 0 0 0 0 0 0	SymRepZero with count of 42x, followed by 6x SymZeros
K.	111 1111110 0 0 0 0 0 0 0	SymRepZero with count of 169x, followed by 7x SymZeros

Avoiding the consecutive zeros is actually trivial in every one of these cases. For cases A, B, C, D, and K, the sequence of at least 5x SymZeros ("0 0 0 0 0") can be replaced with a single SymZero, followed by a SymRepLast with a count of at least 4x ("0 110 10"). For cases E, F, G, H, I, and J, this situation can be avoided by simply folding the trailing SymZeros into the count of the preceding SymRepZero. As a general rule, any greedy algorithm that tries to use the repeater codes whenever possible will never violate the consecutive 0-bits constraint.

An astute reader may note that it is not always the most efficient to use SymRepLast to encode runs of zeros. For example, 5 zeros is more efficiently written as 5x SymZeros, rather than using a single SymZero, followed by a SymRepLast with a count of 4. The first sequence occupies 5 bits, while the second sequence occupies 6 bits. An implementation of the meta encoder may output 5x SymZeros instead of using the repeater so long as the 8 consecutive zeros rule is not violated.

C.3 Size limits of a meta block

In section 2.2.4, we provided some properties of the meta encoding. Here we explore the mathematical basis for those values. To summarize, the table shown above had:

Property	Value	Description
MinRawBytes	0	Minimum and maximum number of metadata bytes a block can encode
MaxRawBytes	31	
EnsureRawBytes	22	Number of bytes that a single block is ensured to encode
MinEncBytes	12	Minimum and maximum number of bytes an encoded block will occupy
MaxEncBytes	64	

C.3.1 Limits to number of bytes that can be encoded

First, let us explore the number of metadata bytes that can be encoded into a block. The metadata itself is encoded in the $LitTree$ definition and is fundamentally limited by the number of symbols in that definition. The $LitTree$ contains $[257..264]$ symbols. The 256th symbol is used for the EOB marker and is always SymOne, the 257th symbol and above are always used for padding and are always SymZero, and the first 8 symbols have special uses (5 of which are used to store the count). This leaves 248 symbols for arbitrary metadata, which amounts to a **MaxRawBytes** of 31, which conveniently fits within the 5-bit count. Also, the **MinRawBytes** is 0 since the size field can trivially be set to a value of 0.

C.3.2 Limits to number of bytes that is ensured be encoded

Next, we determine the number of bytes that can always be encoded into a single meta block. This is determined by whether the intermediate bit-string forms a valid $LitTree$ (i.e., the total number of 1-bits exactly equals $2^{HuffBits}$). In other words, for some given metadata string, so long as we can determine some value for $HuffBits \in [1..7]$ that satisfies the constraint, then we know that the metadata can be encoded in a single block. Furthermore, we note that it does not matter what the metadata string actually is so long as we know the number of 1-bits (and also 0-bits) in the string.

```
def ComputeHuffBits(n0s, n1s):
    MaxSyms = 256 # Does not include 0th symbol and padding symbols
    if n1s > n0s:
        n0s, n1s = n1s, n0s
    n0s += 7 # Add other zero bits: FinalMeta, Invert, Size
    n1s += 8 # Add other one bits: FinalMeta, Invert, Size, EOB symbol
    for hb in range(MinHuffBits, MaxHuffBits+1):
        MaxOnes = 1<<hb
        if MaxSyms-MaxOnes >= n0s and MaxOnes >= n1s:
            return hb
    return None
```

The ComputeHuffBits function computes a valid $HuffBits$ value given information about the input metadata string; specifically the total number bits of each value. Since there is a cap on the total number of 1-bits allowed, we invert the input if there are more 1-bits than 0-bits. We also add 7 or 8 to the number of bits to account for required fields. An astute reader may note that the value of the Invert and Size fields are entirely known, so we could add the exact number of 0-bits and 1-bits that they require, but we use a more conservative metric here for simplicity.

Given the ComputeHuffBits function, we can now compute **EnsureRawBytes**, the number of bytes that can always be encoded into a single meta block. We determine this value by sweeping through all possible combinations of 0-bits and 1-bits for strings of every byte-length and checking that it is always possible to compute a valid $HuffBits$ value.

```
def ComputeEnsureRawBytes():
    MaxRawBytes = 31
    for nb in range(MaxRawBytes+1):
        for n0s in range(8*nb + 1):
            n1s = 8*nb - n0s
            if ComputeHuffBits(n0s, n1s) is None:
                return nb-1
    return MaxRawBytes
```

Running the above function informs us that **EnsureRawBytes** is 22 bytes.

C.3.3 Limits to number of bytes that a meta block can occupy

Lastly, we explore the size of the encoded block itself. Let us first compute the amount of bits lost to the MetaHeader and MetaFooter for each of the possible $HuffBits$ values. The first group of numbers represents the bits that the MetaHeader occupies, while the second group of numbers represents the bits that the MetaFooter occupies (without the padding bits):

```
BitLoss = {
    1: (1+2 + 5+5+4 + 3*18 + 1) + (1 + 1), # 74 bits
    2: (1+2 + 5+5+4 + 3*16 + 1) + (1 + 2), # 69 bits
    3: (1+2 + 5+5+4 + 3*14 + 1) + (1 + 3), # 64 bits
    4: (1+2 + 5+5+4 + 3*12 + 1) + (1 + 4), # 59 bits
    5: (1+2 + 5+5+4 + 3*10 + 1) + (1 + 5), # 54 bits
    6: (1+2 + 5+5+4 + 3*8  + 1) + (1 + 6), # 49 bits
    7: (1+2 + 5+5+4 + 3*6  + 1) + (1 + 7), # 44 bits
}
```

With the BitLoss map now defined, we can now define two other functions that deal only with the MetaBody. The n0s and n1s variables in the functions below are the number of zeros and ones that need to be encoded for a given $HuffBits$. The MinRepLast and MaxRepLast constants are 3 and 6; while the MinRepZero and MaxRepZero constants are 11 and 138.

Let us define a function MaxBits, that when given $HuffBits$, computes the longest meta block.

```
def MaxBits(i):
    n0s, n1s, nb = 256-(1<<i), 1<<i, BitLoss[i]

    # Encode all of the zeros.
    nb += 5*(n0s/MinRepLast)
    nb += 1*(n0s%MinRepLast)

    # Encode all of the ones.
    nb += 2*n1s

    return nb
```

When encoding zeros, the worst case actually occurs when using SymRepLast, with a minimal count, to represent 3 zeros instead of using 3x SymZeros. When generating ones, we avoid any repeater symbols and just output SymOne for all the ones.

Let us define a function MinBits, that when given $HuffBits$, computes the shortest meta block.

```
def MinBits(i):
    n0s, n1s, nb = 256-(1<<i), 1<<i, BitLoss[i]

    # Encode all of the zeros.
    while n0s > 0:
        if n0s >= MinRepZero:
            n0s -= min(MaxRepZero, n0s)
            nb += 10
        elif n0s >= MinRepLast+2:
            n0s -= min(MaxRepLast, n0s)
            nb += 5
        else:
            n0s -= 1
            nb += 1

    # Encode all of the ones.
    nb += 2
    n1s -= 1
    while n1s > 0:
        if n1s >= MinRepLast:
            n1s -= min(MaxRepLast, n1s)
            nb += 5
        else:
            n1s -= 1
            nb += 2

    return nb
```

In both generating zeros and ones, the strategy taken is to use repeater symbols with the largest count as much as possible. In order to avoid the inefficiency of using SymRepLast with zeros as seen in MaxBits, we add 2 to MinRepLast in order to offset it to the breakeven point where it is economical to use SymRepLast for encoding runs of zeros.

Given the MinBits and MaxBits functions, we can determine the minimum and maximum occupy sizes by sweeping through all of the possible $HuffBits$ values:

$$min(\{\, MinBits(i) \mid 1 \le i \le 7 \,\}) = min(\{98, 96, 93, 96, 103, 126, 163\}) = \left\lceil \frac{93\ bits}{8} \right\rceil = 12\ bytes$$

$$max(\{\, MaxBits(i) \mid 1 \le i \le 7 \,\}) = max(\{500, 497, 492, 491, 490, 497, 512\}) = \left\lceil \frac{512\ bits}{8} \right\rceil = 64\ bytes$$

Thus, we confirm that, **MinEncBytes** and **MaxEncBytes**, the minimum and maximum number of bytes an encoded block will occupy is 12 and 64 bytes, respectively.

C.4 Efficiency of encoding

We now analyze the efficiency of the meta encoding. That is, we compute the ratio of the encoded output after meta encoding to the number of bytes being encoded. Using the meta encoder from the reference implementation, we produce the following table:

#Bytes		FullRange	%Range		MinSize	AvgSize	MaxSize		%MaxEff	%AvgEff	%MinEff
0		True	100.0%		12	12.00	12		0.0%	0.0%	0.0%
1		True	100.0%		12	14.08	15		8.3%	7.1%	6.7%
2		True	100.0%		13	15.39	16		15.4%	13.0%	12.5%
3		True	100.0%		13	17.16	18		23.1%	17.5%	16.7%
4		True	100.0%		13	18.57	20		30.8%	21.5%	20.0%
5		True	100.0%		13	19.54	21		38.5%	25.6%	23.8%
6		True	100.0%		13	20.69	22		46.2%	29.0%	27.3%
7		True	100.0%		13	23.42	26		53.8%	29.9%	26.9%
8		True	100.0%		13	25.17	27		61.5%	31.8%	29.6%
9		True	100.0%		13	26.33	28		69.2%	34.2%	32.1%
10		True	100.0%		13	27.22	29		76.9%	36.7%	34.5%
11		True	100.0%		13	27.79	29		84.6%	39.6%	37.9%
12		True	100.0%		13	28.49	31		92.3%	42.1%	38.7%
13		True	100.0%		13	29.39	32		100.0%	44.2%	40.6%
14		True	100.0%		13	30.54	33		107.7%	45.8%	42.4%
15		True	100.0%		13	34.15	39		115.4%	43.9%	38.5%
16		True	100.0%		13	37.43	41		123.1%	42.7%	39.0%
17		True	100.0%		13	39.24	42		130.8%	43.3%	40.5%
18		True	100.0%		13	40.64	43		138.5%	44.3%	41.9%
19		True	100.0%		13	41.65	44		146.2%	45.6%	43.2%
20		True	100.0%		13	42.75	45		153.8%	46.8%	44.4%
21		True	100.0%		13	43.75	47		161.5%	48.0%	44.7%
22		True	100.0%		13	44.89	48		169.2%	49.0%	45.8%
23		False	100.0%		13	45.80	49		176.9%	50.2%	46.9%
24		False	99.98%		13	46.80	50		184.6%	51.3%	48.0%
25		False	99.77%		13	47.87	51		192.3%	52.2%	49.0%
26		False	98.50%		13	49.04	52		200.0%	53.0%	50.0%
27		False	93.41%		13	49.86	53		207.7%	54.2%	50.9%
28		False	79.58%		13	50.69	54		215.4%	55.2%	51.9%
29		False	52.97%		13	51.41	55		223.1%	56.4%	52.7%
30		False	15.35%		13	52.32	55		230.8%	57.3%	54.5%

The meaning of each column is:

- **#Bytes**: The length of the input metadata string.
- **FullRange**: Whether it is possible to encode all possible metadata strings of this length.
- **%Range**: The percentage of all possible metadata strings of this length that can be encoded.
- **(Min, Avg, Max)Size**: The length of the encoded meta block.
- **%(Max, Avg, Min)Eff**: The efficiency measured as the percentage of the encoded output that is actual metadata.

It would have been computationally infeasible to produce the table above by iterating over all possible metadata strings and computing their outputs. Instead, the table was produced by assuming that the length of the encoded output is heavily dependent on the total number of `0`-bits and `1`-bits in the input. For every possible metadata byte length, we sweep through every possible combination of the number of `0`-bits and `1`-bits. For each combination, we encode random metadata strings from a small portion (256 samples) of the set of possible strings for that combination.

When estimating the encoded output sizes, we assume the sampling of 256 results is representative of the entire input space for that combination. To estimate the **AvgSize**, we weight the average of each sample set according to what proportion of the total input space that the combination represents. We determine the proportion according to the following mathematical property:

$$2^n = \sum_{i=0}^{n} C(n, i)$$

This equation allows us to compute the exact number of possible strings that have n `1`-bits. For example, a 1-byte string (8-bits) can be broken down as:

$$2^8 = \sum_{i=0}^{8} C(8, i) = 1 + 8 + 28 + 56 + 70 + 56 + 28 + 8 + 1 = 256$$

Thus, for a combination of 0x `0`-bits and 8x `1`-bits, there is exactly 1 possible input string (0.4% of total). For a combination of 3x `0`-bits and 5x `1`-bits, there are exactly 56 possible inputs (21.9% of total).

When computing the **%Range**, we apply this to each metadata string of some length in bytes. For each combination, we know from Appendix C.3.2 that the number of `0`-bits and `1`-bits alone is sufficient to determine whether a metadata string is encodable. Thus, if a single string of a given combination is encodable, then we know that all possible inputs for that combination are also encodable. Similarly, if a single string is not encodable, then all possible inputs for that combination are also not encodable. Using this knowledge along with the proportions that each combination represents, we can compute the exact percentage of the total input space that is encodable.

From the above table, we make some observations:

- Running a linear regression on **AvgSize**, provides us with $13 + 1.4N$, which is an effective way to estimate the encoded output size given a metadata string of length N up to 30.
- The smallest meta block still occupies around 12 bytes, so there is some minimum overhead that the XFLATE format will introduce over standard DEFLATE.
- A vast majority of inputs can be encoded in blocks containing 26 to 28 actual metadata bytes, where the worst-case efficiency generally will not go below 50%.
- Encoding metadata comprising of long runs of `0`-bits or `1`-bits can actually have efficiency greater than 100% due to the primitive form of run-length encoding using `SymRepLast` and `SymRepZero`.