SPECIFICATION OF
DjVu IMAGE COMPRESSION FORMAT

Version of 1999-04-29 15:46 EDT

Copyright notice

1 Scope

This document defines a non-bit-preserving (lossy) compression method and file format for coding color, grayscale, and bilevel images. It is particularly suitable for compound images consisting of foreground text and background photographic or graphic images.

The compression method and file format are called "DjVu". DjVu is pronounced like the French words *déjà vu*.

The algorithms specified in DjVu are designed for efficient storage, retrieval, and display of scanned document images on the World Wide Web. DjVu provides high compression rates by handling text and images differently, each in a highly efficient way. It provides progressivity, allowing an application to first display the text, then to display the images using a progressive buildup. It is designed to be implemented using data structures appropriate for efficient navigation within an image. It is lightweight: the decoder and data can reside in a small amount of memory, even for large images.

2 Definitions

DjVu is the name of this compression method and file format. It is always written with an upper case 'D' and 'V', and a lower case 'j' and 'u'.

IFF format. After a four-octet preamble, a DjVu file consists of a subfile coded using the EA IFF 85 format, Electronic Arts' public domain IFF standard for Interchange File Format, released in January, 1985. In this document, the EA IFF 85 format is referred to as "IFF" or "the IFF format". The parts of the IFF format used with DjVu are described as needed in this document.

Chunks are the basic unit of data in the IFF format. A DjVu file consists of a four-octet preamble followed by a single chunk whose type is **FORM**. The **FORM** chunk contains chunks of other types.

Arithmetic coding is a bit-level entropy coding method. An arithmetic encoder encodes a sequence of events. For each event it takes as input a set of possible values for the event and a probability associated with each possible event, along with the identity of the event that is to be encoded; it produces a bitstream. An arithmetic decoder decodes a sequence of events. For each event it takes as input a set of possible values for the event and a probability associated with each possible event, along with the previously encoded bitstream. It produces the identity of the event that was encoded.

Binary arithmetic coding is a special case of arithmetic coding in which each event may have only two possible values.

The Z′-Coder is the binary arithmetic coder used in DjVu.

The multivalue extension to the Z′-Coder is used in DjVu to use a binary arithmetic coder to code events for which more than two values are possible.

Image layers are defined relative to a three-layer image model. In the three-layer model defined for DjVu, an image consists of a foreground layer, typically containing printed text or line art, a background layer, typically containing photographic images, texture, and colored graphics, and a selection layer that specifies which image pixels are in the foreground layer and which pixels are in the background layer. A single-layer model is also defined for DjVu. In this model, an image consists of a single graphic layer.

Image types are defined in DjVu according to the image model used (one-layer or three-layer) and the number of layers actually coded.

The **preamble** is a four-octet header at the beginning of a DjVu file. It precedes the IFF-coded data.

A **chunk header** is an 8-octet header at the beginning of each IFF chunk. It identifies the chunk type and specifies the length of the chunk.

An **INFO chunk** is a chunk present in some image types that describes some features of the image, such as its size and spatial resolution.

A **data header** is part of the data within a chunk. When present it describes some features of the image. In chunks of type **Sjbz**, the data header is arithmetically coded. In chunks of other types, it is not arithmetically coded.

The **wavelet transform** is a method of transforming image data by dividing the image into sub-bands. The result of the transformation is a set of wavelet coefficients, which are then coded using an appropriate entropy coder. DjVu uses the Dubuc-Deslauriers-Lemire $(4,4)$ Interpolative Wavelet Transform, and codes the coefficients with the Z′-Coder.

Symbol-based coding is a method of coding textual image data by dividing the image into characters. Character bitmaps are put into a symbol bitmap library. The image is coded by referring to characters in the library and specifying where they are placed in the image. In DjVu, the selection layer is coded using symbol-based coding. The bitmaps and locations are coded using the Z′-Coder.

3 Conventions

3.1 Typeface conventions

IFF chunk types are shown in a bold typeface. Examples: **FORM**, **Sjbz**, **BG44**.

Secondary identifiers for **FORM** chunks are shown in a typewriter typeface. Examples: `DJVU`, `PM44`, `BM44`. Note that the `DJVU` identifier is not the same as DjVu, the name of the compression method and file format.

The names of image types are written in small capital letters. Examples: IW44, DJVU. Note that the DJVU image type is not the same as DjVu, the name of the compression method and file format.

Color component identifiers are shown in a bold typeface. Examples: $\mathbf{Y}$, $\mathbf{C_b}$, $\mathbf{C_r}$.

The names of decisions to be decoded are shown in lower case letters in an italic typeface. Example: the *decode buckets* decision.

The names of flags are shown in upper case letters in a sans serif typeface. Example: the ACTIVE flag.

Flag values are shown in upper case letters in a sans serif typeface, enclosed in a small flag-like box. Examples: SET, FALSE, PREVIOUS.

Decision values are shown in upper case letters in a sans serif typeface, enclosed in a small rectangular box. Examples: NO, WHITE.

Bit values are shown in a bold typeface. Examples: $\mathbf{b_7}$, $\mathbf{1}$.

Names of contexts to be used for arithmetic coding are shown in lower case letters in an italic typeface. Examples: the *record type* context, the *offset type* context.

Record type identifiers in an **Sjbz** chunk are shown in lower case letters in a sans serif type face. Example: the start of image record.

Hexadecimal numbers are shown in a typewriter typeface, preceded by the characters "0x". Example: The value 2000, written as a 16-bit hexadecimal number, is 0x07D0.

In pseudo-code, the symbol ":=" represents assignment. Example: $A := Z$ means that variable A takes on the current value of variable Z.

3.2 Mathematical notation

Standard mathematical notation is used where appropriate. In particular, the following symbols are used.

$|n|$ means the absolute value of n. Example: $|-3| = 3$.

$\lfloor x \rfloor$ means the floor of x, that is, the largest integer that is less than or equal to x. Examples: $\lfloor 3.5 \rfloor = 3$, $\lfloor 2 \rfloor = 2$, $\lfloor -3.5 \rfloor = -4$.

$\lceil x \rceil$ means the ceiling of x, that is, the smallest integer that is greater than or equal to x. Examples: $\lceil 3.5 \rceil = 4$, $\lceil 2 \rceil = 2$, $\lceil -3.5 \rceil = -3$.

$[a, b)$ means the the interval from a to b including a but not including b. The interval may consist only of integer values, in which case the largest value is $b - 1$.

$(a, b]$ means the the interval from a to b including b but not including a. The interval may consist only of integer values, in which case the smallest value is $a + 1$.

(a, b) means the the interval from a to b including neither a nor b. The interval may consist only of integer values, in which case the smallest value is $a + 1$ and the largest value is $b - 1$.

$n \pmod 2$ is 0 if n is even and 1 if n is odd.

$\min(a, b)$ means the smaller value of a and b. Example: $\min(7, 11) = 7$.

The median of three values is the second largest of the values. In case two or all three values are equal, any of the equal values may be taken to be the median. Examples: the median of 4, 1, and 2 is 2; the median of 3, 1, and 3 is 3.

4 Introduction

This document specifies the DjVu compression method and file format. The format is designed to allow the efficient representation and transmission of documents containing color, grayscale, and bilevel text and images.

This document specifies a description of a compliant DjVu bitstream together with the semantics that permit interpretation of the bitstream and rendering of the corresponding decompressed image.

4.1 Image models

DjVu supports both a multi-layer image model and a single-layer image model.

4.1.1 Multi-layer model

In the multi-layer model a document image is divided into two color layers: a foreground layer and a background layer. Text coloring information is typically put into the foreground layer. Other image data, including texture and photographic images, is put into the background layer. The foreground and background layers use the same coding syntax, but are coded independently.

A third layer, called the selection layer, is used to determine, for each pixel, whether that pixel should be rendered using the color in the background layer or the color in the foreground layer. The selection layer is coded independently from the color layers, using a different coding syntax.

It is possible to omit some layers in a DjVu file. If only the selection layer is present, the foreground is assumed to be solid black, and the background is assumed to be solid white. This permits efficient coding of pure text images. If only a single color layer is present, it is assumed to the background layer, and the background layer is selected for all pixels.

4.1.2 Single layer model

When the single-layer model is used, the only layer is an image layer. It may be a color layer or a grayscale layer.

4.2 File format

The bitstream is coded using the EA IFF 85 format, preceded by a four-octet preamble. The format consists of a file header and chunks. Each chunk consists of a chunk header followed by chunk data. The chunk header consists of a four-octet type ID followed by a four-octet integer specifying the number of octets of chunk data in the chunk.

In DjVu, chunk data consists of a short header whose length is determined by the chunk type, followed by a stream of arithmetically coded data. The arithmetic coder used is the Z′-Coder, a binary arithmetic coder. The Z′-Coder may be used in an adaptive mode, in which probability estimates for binary choices are adapted, or in a pass-through mode, in which the probability estimates are fixed. When exactly two choices are possible for a given data element, the Z′-Coder is used without modification. When more than two choices are possible for a given data element, a multivalue extension to the Z′-Coder is used.

4.3 Precedence of reference library

A source code reference library is provided that implements a DjVu-compliant decoder. Every effort has been made to ensure that the written specification and the source code library define identical decoders. In case of disagreement between the written specification and the source code library, the source code library is to be considered correct.

5 Requirements: DjVu image types

DjVu supports DJVU Images, which are images coded according to the three-layer model. DjVu also supports IW44 Images, which are images coded according to the single-layer

model.

5.1 DJVU image types

DjVu supports three three-layer image types, called DJVU Images.

5.1.1 Compound DJVU Image

A Compound DJVU Image contains a selection layer, a foreground layer, and a background layer. The foreground layer may consist of one or three color components. Independently, the background layer may consist of one or three color components.

5.1.2 Bilevel DJVU Image

A Bilevel DJVU Image contains only a selection layer; the foreground layer is implicitly entirely black, and the background layer is implicitly entirely white.

5.1.3 Photo DJVU Image

A Photo DJVU Image contains only a background layer. The background layer may consist of one or three color components.

5.2 IW44 image types

DjVu supports two single-layer image types, called IW44 Images.

5.2.1 Color IW44 Image

A Color IW44 Image contains only a single layer, consisting of three color components.

5.2.2 Grayscale IW44 Image

A Grayscale IW44 Image contains only a single layer, consisting of one color component.

6 Requirements: Structure of DjVu files

6.1 DjVu file format

6.1.1 Preamble to identify DjVu files

The first four octets of a DjVu file are `0x41 0x54 0x26 0x54`. This preamble is not part of the EA IFF 85 format, but it is required in order to identify DjVu files.

6.1.2 Use of IFF format

After the four-octet preamble, a DjVu file consists of a subfile coded using the EA IFF 85 format. The parts of the IFF format used within DjVu are described as needed in this document.

6.2 IFF chunk structure

6.2.1 IFF file structure

An IFF file consists of a number of chunks. Each chunk has a header and data. The header of a chunk consists of a four-octet chunk-type field and a four-octet length field. The length is coded most significant octet first. The strings that identify the types of the chunks in

Image type	Secondary **FORM** chunk identifier	Number of chunks of each type					
		INFO	**Sjbz**	**BG44**	**FG44**	**PM44**	**BM44**
Compound DJVU	`DJVU`	1	1	≥ 1	1	–	–
Bilevel DJVU	`DJVU`	1	1	–	–	–	–
Photo DJVU	`DJVU`	1	–	≥ 1	–	–	–
Color IW_{44}	`PM44`	–	–	–	–	≥ 1	–
Grayscale IW_{44}	`BM44`	–	–	–	–	–	≥ 1

Table 1: DjVu image types

an IFF file are defined by the application. A chunk whose type is not recognized by the application is to be ignored.

In the IFF format, chunks may be nested: a chunk may contain other chunks as part of its data. In the DjVu format, there is only one chunk at the outermost nesting level, a **FORM** chunk. All other chunks appear within the **FORM** chunk, sequentially, with no nesting.

Each chunk, including those nested within another chunk, must begin on an even octet boundary; that is, the number of octets in the file before the beginning of the chunk must be an even integer. If necessary to ensure that a chunk begins on a even octet boundary, a single padding octet whose value is `0x00` is placed before a chunk.

6.2.2 IFF headers

An IFF header consists of a four-octet ASCII string identifying the chunk type, a four-octet integer containing the length of the data, most significant octet first, and the data. The length of the data includes only the actual data in the chunk. It does not include the eight-octet IFF header. It does not include the padding octet that may be present after the data. It does include data headers that may be present in the data. The length of a **FORM** chunk includes the padding octets that may be present between chunks nested within the **FORM** chunk.

6.3 IFF chunk sequence

The IFF chunk types used in DjVu are the following: **FORM**, **INFO**, **Sjbz**, **BG44**, **FG44**, **PM44**, and **BM44**. All files contain a **FORM** chunk, inside which all the other chunks are nested. The first four data octets of the **FORM** chunk are a secondary identifier. In DjVu, the secondary identifier gives some information about the image type. The types of other chunks used in a single DjVu file, and the permitted number of each type, depend on the image type according to the Table 1.

In a DJVU Image, an **INFO** chunk is present. The **INFO** chunk is the first chunk within the **FORM** chunk. In a Compound DJVU Image, the chunks after the **INFO** chunk may occur in any order, although the order of the **BG44** chunks, if there are more than one, is significant.

6.4 IFF chunk types

6.4.1 FORM chunk type

The **FORM** chunk type is used to encapsulate the entire image content of a DjVu file. The **FORM** chunk is the first chunk in the file. The data in the **FORM** chunk consists of a four-octet ASCII string, called the secondary identifier, followed by all the image data.

The secondary identifier is `DJVU` for a Compound DJVU Image, Bilevel DJVU Image, or Photo DJVU Image. The secondary identifier is `PM44` for a Color IW_{44} Image. The secondary identifier is `BM44` for a Grayscale IW_{44} Image.

6.4.2 INFO chunk type

The **INFO** chunk type is used in the Compound DJVU Image, Bilevel DJVU Image, and Color DJVU Image file types to convey information about the image as a whole, and about the coding used. The **INFO** chunk data consists of six fields in nine octets:

- **Image width.** A two-octet unsigned integer, most significant octet first, specifying the width of the image in pixels.
- **Image height.** A two-octet unsigned integer, most significant octet first, specifying the height of the image in pixels.
- **Minor version number.** A one-octet unsigned integer, specifying the minor version number of the encoder being used.
- **Major version number.** A one-octet unsigned integer, specifying the major version number of the encoder being used.
- **Spatial resolution.** A two-octet unsigned integer, least significant octet first, specifying the spatial resolution of the image in dots per inch (dots per 2.54 cm).
- **Gamma.** A one-octet unsigned integer, equal to 10 times the gamma of the device on which the image is expected to be rendered.

Any additional data in the **INFO** chunk is to be ignored.

6.4.3 Sjbz chunk type

The **Sjbz** chunk type is used to code selection layer data when the image consists of multiple layers. The **Sjbz** chunk type is also used to code a bilevel image when the image consists of a single bilevel layer. A file may contain at most one **Sjbz** chunk. The structure of an **Sjbz** chunk is described in Section 8.

6.4.4 BG44 chunk type

The **BG44** chunk type is used to code color image data for the background of the image. A file may contain more than one **BG44** chunk. The structure of a **BG44** chunk is described in Section 7.

6.4.5 FG44 chunk type

The **FG44** chunk type is used to code color image data for the foreground of the image. A file may contain at most one **FG44** chunk. The structure of an **FG44** chunk is described in Section 7.

6.4.6 PM44 chunk type

The **PM44** chunk type is used to code color image data. It is used when the image consists of a single layer with three color components. A file may contain more than one **PM44** chunk. The structure of a **PM44** chunk is described in Section 7.

6.4.7 BM44 chunk type

The **BM44** chunk type is used to code grayscale image data. It is used when the image consists of a single layer with one color component. A file may contain more than one **BM44** chunk. The structure of a **BM44** chunk is described in Section 7.

6.5 Alignment of layers

The different color layers may be coded at different resolutions. During image reconstruction, layers are aligned according to the bottom left corner of their bottom left pixels.

7 Requirements: Color and grayscale image coding

7.1 Scope of color and grayscale image coding

This section describes the coding of chunks of type **BG44**, **FG44**, **PM44**, and **BM44**. Chunks of type **BG44** and **FG44** may be color or grayscale chunks. Chunks of type **PM44** are color chunks. Chunks of type **BM44** are grayscale chunks.

All of these color and grayscale chunk types have the same structure. The chunk consists of a chunk header followed by arithmetically coded wavelet coefficient updates. The coefficients are organized in a hierarchical fashion.

7.2 Definitions

- Color component. Compound DJVU Images and Photo DJVU Images contain color or grayscale image data. Color IW44 Images contain color image data. Grayscale IW44 Images contain grayscale image data.

  Color image data is coded using three color components, called $\mathbf{Y}$, $\mathbf{C_b}$, and $\mathbf{C_r}$. These correspond to the usual $\mathbf{YC_bC_r}$ color space, adjusted to facilitate transformation to the **RGB** color space.

  Grayscale image data is coded using one color component, called $\mathbf{Y}$. This corresponds to the grayscale intensity of the image.

- Color layer. A color layer is either:

  1. The foreground layer of a Compound DJVU Image, coded in one **FG44** chunk, or
  2. The background layer of a Compound DJVU Image, coded in one or more **BG44** chunks, or

Band number	Coefficient indices	Bucket indices
0	$0-15$	0
1	$16-31$	1
2	$32-47$	2
3	$48-63$	3
4	$64-127$	$4-7$
5	$128-191$	$8-11$
6	$192-255$	$12-15$
7	$256-511$	$16-31$
8	$512-767$	$32-47$
9	$768-1023$	$48-63$

Table 2: Wavelet coefficient bands.

3. The only layer of a Photo DJVU Image, coded in one or more **BG44** chunks, or
4. The only layer of a Color IW44 Image, coded in one or more **PM44** chunks, or
5. The only layer of a Grayscale IW44 Image, coded in one or more **BM44** chunks.

- Color chunk. A color chunk is a chunk of type **BG44**, **FG44**, **PM44**, or **BM44**. A color chunk contains wavelet coefficient update information for one or three color components.

- Block. A rectangular array of pixels of size 32×32 or less. The blocks are numbered starting in the lower left corner of the image. All blocks are 32×32 except possibly those along the right edge or top edge; those blocks may be smaller if the image dimensions are not divisible by 32.

- Block count. The number of blocks in the image, denoted by N_B.

- Wavelet block. The set of coefficients associated with one block of the image, in one color component. There are 1024 wavelet coefficients in a wavelet block, numbered 0 through 1023. The coefficients in a wavelet block have effects on the reconstruction of other blocks in the image, but for coding purposes they are considered to be localized within the block in which they are coded.

- Bucket. A particular set of 16 wavelet coefficients within a wavelet block. A wavelet block consists of 64 buckets, numbered 0 through 63. Table 2 gives the correspondence between coefficients and buckets.

- Band. A subset of wavelet coefficients for a given color component. There are 10 bands. The correspondence among band numbers, coefficient coefficients, and bucket coefficients is given by Table 2.

- Cycle. Data for one color component consisting of coefficient updates for all coefficients, that is, for all 10 bands, starting with band 0. Within one band, only some coefficients are updated, but within a cycle, all coefficients are updated. The last cycle of a color component may have fewer than 10 bands.

- Color band number. The current band number for a color component. Each color component's color band number starts at 0, and increases by 1 at the end of selected slices until it reaches 9; then it is reset to 0.

- Color band. A collection of update information for a subset of the coefficients of one color component of the image, consisting of updates of all the coefficients in the image whose indices within their respective blocks are those corresponding to the current color band's color band number.

- Slice. A slice is the highest level subdivision of a color chunk. A slice contains data for one color band for each of the color components in a color layer, that is, for three color components for a color image, or for one color component for a grayscale image.

- Block band. A collection of update information for a subset of the coefficients of one color component of a wavelet block, consisting of updates of the coefficients in the block whose indices are those corresponding to a given band.

- Chrominance delay counter. An integer counter that indicates how many slices in a color layer contain a color band only for the **Y** color component, and not for the $\mathbf{C_b}$ and $\mathbf{C_r}$ color components. The chrominance delay counter is initially set to the value specified in the **INFO** chunk for the color layer, and decremented by 1 after each slice in the color layer until it reaches 0. See Section 7.5.1.1.

- Step size table. A table that indicates the precision to which each coefficient in a color component is currently stored. There are three such tables for a given color layer, one for each color component. Each such table has 16 entries. Each entry specifies the current step size for 1, 4, 16, 64, or 256 different coefficient indices, according to Table 3.

7.3 Color chunks within an DjVu file

There may be more than one **BG44** or **PM44** or **BM44** chunks in a DjVu file. If there is more than one such color chunk, the coefficient updating is continuous across the chunks, and the data is taken from the chunks in the order in which they appear in the file. Nothing is reinitialized at the beginning of chunks after the first color chunk of these types, except for the low level arithmetic coder. The probability estimates for the arithmetic coder are not reinitialized.

In a Compound DJVU Image file, in which both an **FG44** chunk and one or more **BG44** chunks appear in the same file, the coding of the foreground layer, using the **FG44** chunk, is independent of the coding of the background layer, using the **BG44** chunks.

Each color layer is coded using a Dubuc-Deslauriers-Lemire $(4,4)$ Interpolative Wavelet Transform. Each layer of the image is transformed into a set of wavelet coefficients, one

wavelet coefficient for each pixel in the original image. This transform is especially effective for coding images at high compression ratios.

The value of each coefficient is coded in a distributed fashion, through a number of cycles. Within one cycle, each coefficient is updated once (that is, in only one of the 10 bands), and receives approximately one additional bit of information. Specifically, from cycle to cycle the absolute value of a coefficient is first narrowed down by eliminating possible values for the most significant non-zero bit until the correct most significant non-zero bit is found. The sign is coded in the same cycle in which the most significant non-zero bit is found. Then in each subsequent cycle, one additional bit of the value is coded.

7.4 Color chunk data headers

A color chunk begin with a data header consisting of 2 or 9 octets, as follows:

- **Serial number.** A one-octet unsigned integer. The serial number of the first chunk of a given chunk type is 0. Successive chunks are assigned consecutive serial numbers.
- **Number of slices.** A one-octet unsigned integer. The number of slices coded in the chunk.
- **Major version number and color type**. One octet containing two values, present only if the serial number is 0. The least significant seven bits designate the major version number of the standard being implemented by the decoder. For this version of the standard, the major version number is 1. The most significant bit is the color type bit. The color type bit is **0** if the chunk describes three color components. The color type bit is **1** if the chunk describes one color component.
- **Minor version number**. A one-octet unsigned integer, present only if the serial number is 0. This octet designates the minor version number of the standard being implemented by the decoder. For this version of the standard, the minor version number is 2.
- **Image width.** A two-octet unsigned integer, most significant octet first, present only if the serial number is 0. This field indicates the number of pixels in each row of the image described by the current chunk. The image width will be less than the width of the original image if the chunk describes a layer coded at lower resolution than the original image. For a **BG44** or **FG44** chunk, if W is the width of the original image specified in the **INFO** chunk, and w is the width of the image described by the current chunk, then the allowable values of w are:

  $$\left\lceil \frac{W}{1} \right\rceil, \left\lceil \frac{W}{2} \right\rceil, \left\lceil \frac{W}{3} \right\rceil, \left\lceil \frac{W}{4} \right\rceil, \left\lceil \frac{W}{5} \right\rceil, \left\lceil \frac{W}{6} \right\rceil, \left\lceil \frac{W}{7} \right\rceil, \left\lceil \frac{W}{8} \right\rceil, \left\lceil \frac{W}{9} \right\rceil, \left\lceil \frac{W}{10} \right\rceil, \left\lceil \frac{W}{11} \right\rceil, \text{and } \left\lceil \frac{W}{12} \right\rceil.$$

  For a **BM44** or **PM44** chunk, there are no restrictions on the image width.
- **Image height.** A two-octet unsigned integer, most significant octet first, present only if the serial number is 0. This field indicates the number of pixels in each column of the image described by the current chunk. The image height will be less than the height of the original image if the chunk describes a layer coded at lower resolution than the

original image. For a **BG44** or **FG44** chunk, if H is the height of the original image specified in the **INFO** chunk, and h is the height of the image described by the current chunk, then the allowable values of h are:

$$\left\lceil \frac{H}{1} \right\rceil, \left\lceil \frac{H}{2} \right\rceil, \left\lceil \frac{H}{3} \right\rceil, \left\lceil \frac{H}{4} \right\rceil, \left\lceil \frac{H}{5} \right\rceil, \left\lceil \frac{H}{6} \right\rceil, \left\lceil \frac{H}{7} \right\rceil, \left\lceil \frac{H}{8} \right\rceil, \left\lceil \frac{H}{9} \right\rceil, \left\lceil \frac{H}{10} \right\rceil, \left\lceil \frac{H}{11} \right\rceil, \text{and } \left\lceil \frac{H}{12} \right\rceil.$$

For a **BG44** or **FG44** chunk, It must be the case that

$$\left\lceil \frac{W}{w} \right\rceil = \left\lceil \frac{H}{h} \right\rceil.$$

For a **BM44** or **PM44** chunk, there are no restrictions on the image width.

- **Initial value of chrominance delay counter.** A one-octet unsigned integer, present only if the serial number is 0. Only the least significant seven bits are used. The most significant bit is ignored, but should be set to **1** by an encoder. This field specifies the initial value of the chrominance delay counter, used as described below.

7.5 Color chunk data

7.5.1 Hierarchical structure of a coded color layer

The data coded in a color chunk consists of information needed to reconstruct wavelet coefficients. There are one or three color components; each color component has its own set of wavelet coefficients. Within a color component, there are 1024 wavelet coefficients for each 32×32 block of the image.

Within one layer (background or foreground for a DJVU Image, or the only layer for an IW44 Image), coding is divided into a series of slices. All the slices may be coded in one chunk, or they may be separated into a number of chunks. The only difference it makes whether the slices are coded in one chunk or in several chunks is in the order of progressive rendering; the final reconstructed image will be the same. The number of slices in each chunk is specified in the color chunk data header. One slice contains refinement data for one color band for each color component. Within a color component, all coefficients in a slice are in the same band.

A color chunk describes the full image at the spatial resolution implied by the image width and image height fields in the data header of the first chunk of the same type as the current color chunk.

The sequence of color components within a slice is: first $\mathbf{Y}$, then $\mathbf{C_b}$, then $\mathbf{C_r}$, although the $\mathbf{C_b}$ and $\mathbf{C_r}$ components are not present in a slice if the chunk describes grayscale data or if the chrominance delay counter is not equal to 0 at the time the slice is coded.

A color band is made up of coefficient updates for all blocks in the image, but only for coefficients that are in the currently active band for the color component. Each block's set of updates within a color band is called a block band. The block bands are coded block by block, first from left to right within the bottom row, then by rows moving up the image, left to right within each row.

Within a block band, there are 16, 64, or 256 coefficient updates. The coefficients being updated are divided into buckets, each bucket containing 16 coefficients. Thus, a block band contains 1, 4, or 16 buckets. The buckets and coefficients being updated are determined by the color band number according to Table 2.

7.5.1.1 Band counting. The header of the first color chunk contains an initial value for the chrominance delay counter. It may be 0 or a positive integer.

At the beginning of the first color chunk, the color band number for each of the three color components is set to 0.

At the beginning of each slice, the chrominance delay counter is tested. If the chrominance delay counter is 0 and if the slice describes color image data, then all three color components are present. If the chrominance delay counter is greater than 0 or if the chunk describes grayscale image data, only the **Y** color component is present for the slice.

At the end of a slice, the following actions take place:

- The color band number is increased by 1 for the **Y** component. If the new color band number exceeds 9, it is set to 0.
- If the chrominance delay counter is 0, the color band numbers for the $\mathbf{C_b}$ and $\mathbf{C_r}$ components are increased by 1. If the new color band numbers exceed 9, they are set to 0. (Note: The color band numbers for the $\mathbf{C_b}$ and $\mathbf{C_r}$ components are always equal to each other.)
- If the chrominance delay counter is greater than 0, it is decreased by 1.

A color chunk ends when the number of slices specified in the color chunk header have been coded.

At the beginning of each color chunk after the first for a given color layer, the chrominance delay counter and color band numbers retain the values they had at the end of the previous color chunk.

7.5.2 Quantization of coefficients

At each point during the decoding process, each wavelet coefficient has been determined to a certain precision. The current value a of the coefficient is stored, and a current step size S is associated with the coefficient. The current step size for each coefficient is governed by a step size table. The index of the entry in the step size table that contains the step size for a given coefficient is given in Table 3.

If $a \neq 0$, the coefficient is said to be "active". If $a > 0$, the range of possible actual values of the coefficient is $[a - S, a + S)$. If $a < 0$, the range of possible actual values of the coefficient is $(a - S, a + S]$. If $a = 0$, the coefficient is not active, and the range of possible actual values of the coefficient is $(-2S, 2S)$.

When the value of a given coefficient is updated, there are three cases.

1. If the coefficient is not active ($a = 0$), then there are three possibilities for the next current interval: $(-2S, -S]$, $(-S, S)$, or $[S, 2S)$. If the coefficient remains not active, then the next interval is $(-S, S)$. Otherwise, the sign of the coefficient is decoded to choose between the other two intervals.
2. If the coefficient is active and $a > 0$, then there are two possibilities for the next current interval: $[a - S, a)$ or $[a, a + S)$. The next decision for the coefficient is the *increase coefficient absolute value* decision. If this decision is YES, then $[a, a + S)$ is the next interval. If the decision is NO, then $[a - S, a)$ is the next interval.

Coefficient index	Index into step size table
0	0
1	1
2	2
3	3
4−7	4
8−11	5
12−15	6
16−31	7
32−47	8
48−63	9
64−127	10
128−191	11
192−255	12
256−511	13
512−767	14
768−1023	15

Table 3: Step size table indices related to wavelet coefficient indices.

3. If the coefficient is active and $a < 0$, then there are two possibilities for the next current interval: $(a - S, a]$ or $(a, a + S]$. The next decision for the coefficient is the *increase coefficient absolute value* decision. If this decision is YES, then $(a - S, a]$ is the next interval. If the decision is NO, then $(a, a + S]$ is the next interval.

7.5.2.1 Initialization of step sizes. The initial values of the step sizes are given in Table 4. There is a separate table of step sizes for each color component. Each color component's table is given the same initial values.

7.5.2.2 Reduction of step sizes. Each slice contains one band of coefficient update information for each color component. At the end of a slice, the step sizes are divided by 2 for the current band for each color component. The indices of the step sizes to be reduced for each band are given in Table 5. For a given color band, either 1 or 7 step sizes are reduced.

Non-zero step sizes are always integer powers of 2. When a step size of 1 is divided by 2, the result is set to 0.

7.6 Coefficient updating

Within a block band, each coefficient in the block band may be updated. A block band is decoded by a preliminary flag computation followed by four passes. One or more of the passes may be skipped or may not require any decoding, depending on conditions present at the beginning of the block band's coding and on tests made during the decoding of previous passes with the block band. The 4 passes are:

1. Decoding the *decode buckets* decision for the block band.

Step size table index	Initial value
0	0x04000
1	0x08000
2	0x08000
3	0x10000
4	0x10000
5	0x10000
6	0x20000
7	0x20000
8	0x20000
9	0x40000
10	0x40000
11	0x40000
12	0x80000
13	0x40000
14	0x40000
15	0x80000

Table 4: Initialization of step sizes.

2. Decoding the *decode coefficients* decision for buckets in the block band.

3. Decoding the *activate coefficient* decision for coefficients in the block band, and determining the sign of newly activated coefficients.

4. Decoding the update decisions for previously active coefficients.

During the coefficient updating process, a number of binary decisions are decoded. Each decision is decoded using the Z′-Coder. Decoding a decision using the Z′-Coder may be done with a conditioning context, or it may be done using the pass-through mode of the Z′-Coder. For color chunk decoding, there are up to 584 conditioning contexts, that is, up to 294 conditioning contexts for the background layer and up to 294 conditioning contexts for the foreground layer. Within a layer, there are 98 conditioning contexts for each color component; one or three color components may be present for each layer. The contexts are as follows:

- 1 context in each color component is for the *decode buckets* decision.

- 80 contexts in each color component are for the *decode coefficients* decision, 8 for each of the 10 bands.

- 16 contexts in each color component are for the *activate coefficient* decision.

- 1 context in each color component is for the *increase coefficient absolute value* decision.

Band number	Step size table indices
0	$0-6$
1	7
2	8
3	9
4	10
5	11
6	12
7	13
8	14
9	15

Table 5: Step size reduction schedule.

The coefficient sign is decoded using the pass-through mode of the Z′-Coder without a context. For all occurrences of the *increase coefficient absolute value* decision for any coefficient after the first such decision, the *increase coefficient absolute value* decision is coded using the pass-through mode of the Z′-Coder.

7.6.1 Preliminary flag computation.

Flags are computed for each coefficient in the block band, for each bucket in the block band, and for the block band as a whole.

1. Flag computation for coefficients. For each coefficient in a block band, there is a value of the step size. For each coefficient, there are two flag values, based on the value of the coefficient and the value of the coefficient's step size. The flags are called ACTIVE and POTENTIAL. At most one of the flag values may be SET for a coefficient in a given cycle. If the coefficient's step size is either 0 or greater than or equal to `0x8000`, then both flag values are CLEAR. The two flag values are:

   (a) ACTIVE: The coefficient's ACTIVE flag value is SET if the coefficient's step size is greater than 0 and less than `0x8000`, and the coefficient's value is not 0. Otherwise the coefficient's ACTIVE flag value is CLEAR. The sign of the coefficient is known, and the position of the most significant non-zero bit of its absolute value is known.

   (b) POTENTIAL: The coefficient's POTENTIAL flag value is SET if the coefficient's step size is greater than 0 and less than `0x8000`, and the coefficient's value is 0. Otherwise the coefficient's POTENTIAL flag value is CLEAR. It is possible that the value of this coefficient will become non-zero during this cycle.

2. Flag computation for buckets. Each bucket has two flag values associated with it depending on the flags of the 16 coefficients in the bucket. The bucket flags have the same names as the coefficient flags. Both, one, or neither of the bucket flags may be SET for a bucket in a given cycle.

(a) ACTIVE: The bucket's ACTIVE flag is SET if any of the coefficients in the bucket have ACTIVE flags SET. Otherwise the bucket's ACTIVE flag value is CLEAR.

(b) POTENTIAL: The bucket's POTENTIAL flag is SET if any of the coefficients in the bucket have POTENTIAL flags SET. Otherwise the bucket's POTENTIAL flag value is CLEAR.

3. Flag computation for the block band. The block band has two flag values associated with it depending on the flags of the buckets in the block band. The block band flags have the same names as the bucket flags. Both, one, or neither of the block band flags may be SET for a block band in a given cycle. The block band flag values are not needed if the number of buckets in the block band is less than 16.

   (a) ACTIVE: The block band's ACTIVE flag is SET if any of the buckets in the block band have ACTIVE flags SET. Otherwise the block band's ACTIVE flag value is CLEAR.

   (b) POTENTIAL: The block band's POTENTIAL flag is SET if any of the buckets in the block band have POTENTIAL flags SET. Otherwise the block band's POTENTIAL flag value is CLEAR.

7.6.2 Block-band-decoding pass.

If the block band contains fewer than 16 buckets, the block-band-decoding pass is skipped and the bucket decoding pass is performed. If the block band's ACTIVE flag is SET, the block-band-decoding pass is skipped and the bucket decoding pass is performed. If the block band contains 16 buckets, and if the block band's ACTIVE flag is CLEAR, and if the block band's POTENTIAL flag is SET, then the *decode buckets* decision is decoded. If the *decode buckets* decision is YES, the bucket-decoding pass is performed for the block band. If the *decode buckets* decision is NO, the bucket-decoding pass and the newly-active-coefficient-decoding pass are skipped for the block band.

7.6.2.1 Arithmetic decoding. For each color component, there is a single context for use in decoding the *decode buckets* decision.

If the value returned by the Z′-Coder for the *decode buckets* decision is **1**, then the value of the *decode buckets* decision is YES. If the value returned by the Z′-Coder is **0**, then the value of the *decode buckets* decision is NO.

The Z′-Coder context for the *decode buckets* decision for each color component is initially set to 0.

7.6.3 Bucket-decoding pass.

Each bucket has a flag called the coefficient-decoding flag. If the bucket-decoding pass is not skipped, then for each bucket in the block band, if the bucket's POTENTIAL flag is SET, then the *decode coefficients* decision for the bucket is decoded. If the the *decode coefficients* decision is YES, then the bucket's coefficient-decoding flag is SET; otherwise it is CLEAR.

7.6.3.1 Arithmetic decoding. For each color component, there are 80 contexts for use in decoding the *decode coefficients* decision. For each of the 10 bands in a color component, there are 8 contexts. There are four contexts that may be used if the block band's ACTIVE

flag is SET, and four contexts that may be used if the block band's ACTIVE flag is CLEAR. The index of the context to be used among the 4 possible contexts is computed as follows. If the band number is 0, then $n_0 = 0$. Otherwise, the value of n_0 is computed as follows:

1. The bucket number is multiplied by 4, giving a result t.
2. The coefficients numbered t, $t+1$, $t+2$, and $t+3$ are examined, and the number n_0 of coefficients with value 0 among the four coefficients is counted.
3. If $n_0 = 4$, n_0 is reduced to 3.

Then the value of n_0 is used as the index to one of the four contexts, for the given color component, band, and block band ACTIVE flag value.

If the value returned by the Z′-Coder for the *decode coefficients* decision is **1**, then the value of the *decode coefficients* decision is YES. If the value returned by the Z′-Coder is **0**, then the value of the *decode coefficients* decision is NO.

Each of the 80 Z′-Coder contexts for the *decode coefficients* decision for each color component is initially set to 0.

7.6.4 Newly-active-coefficient-decoding pass.

If the newly-active-coefficient-decoding pass is not skipped, then for each bucket in the block band, the coefficient-decoding flag is tested. For a given bucket, if the bucket's coefficient-decoding flag is SET, then the following procedure is followed for each coefficient in the bucket: If the coefficient's POTENTIAL flag is SET, then the *activate coefficient* decision is decoded. If the the *activate coefficient* decision is YES, then the sign of the coefficient $s_\pm$, with value $+1$ or -1, is decoded. Then the coefficient is set equal to

$$\frac{3}{2} \times s_\pm \times \text{coefficient's step size.}$$

7.6.4.1 Arithmetic decoding. For each color component, there are 16 contexts for use in decoding the *activate coefficient* decision. There are eight contexts that may be used if the block's ACTIVE flag is SET, and eight contexts that may be used if the block's ACTIVE flag is CLEAR. The index of the context to be used from among the 8 possible contexts is computed as follows:

1. The coefficients in the bucket are examined, and the number n_p of them whose POTENTIAL flag is SET is computed.
2. Loop through the coefficients whose POTENTIAL flag is SET.
   (a) Compute $i_p = \min(7, n_p)$.
   (b) Use i_p as the index into the set of 8 possible contexts, given the color component and value of the block's ACTIVE flag.
   (c) Decode the *activate coefficient* decision using the context; if the *activate coefficient* decision is YES, decode the sign using the pass-through mode of the Z′-Coder, and set $n_p = 0$.

(d) If $n_p > 0$, decrement n_p by 1.

If the value returned by the Z′-Coder for the *activate coefficient* decision is **1**, then the value of the *activate coefficient* decision is YES. If the value returned by the Z′-Coder is **0**, then the value of the *activate coefficient* decision is NO.

The decoding of the sign $s_\pm$ of a newly activated coefficient uses the pass-through mode of the Z′-Coder. If the value returned by the Z′-Coder is **1**, then $s_\pm = -1$. If the value returned by the Z′-Coder is **0**, then $s_\pm = +1$.

Each of the 16 Z′-Coder contexts for the *activate coefficients* decision for each color component is initially set to 0.

7.6.5 Previously-active-coefficient-decoding pass.

For all coefficients in the block band, the following procedure is followed: If the coefficient's ACTIVE flag is SET, the *increase coefficient absolute value* decision is decoded. If the decision is NO, the absolute value of the coefficient is reduced by half of the coefficient's step size. If the decision is YES, the absolute value of the coefficient is increased by half of the coefficient's step size. A step size of 1 is a special case. If the step size is 1 and the decision is NO, the absolute value of the coefficient is reduced by 1. If the step size is 1 and the decision is YES, the value of the coefficient is unchanged.

7.6.5.1 Arithmetic decoding. For each color component, there is a single context for use in decoding the *increase coefficient absolute value* decision. This context is used to decode the *increase coefficient absolute value* decision if the absolute value of the coefficient is less than or equal to 3 times the value of the step size for the coefficient. Otherwise, the pass-through mode of the Z′-Coder is used. (Note: the effect of this test is that only the second most significant bit of a coefficient's value is decoded using this context; other less significant bits are decoded using the pass-through mode of the Z′-Coder, with no context.)

Whether the context or the pass-through mode is used, if the value returned by the Z′-Coder for the *increase coefficient absolute value* decision is **1**, then the value of the *increase coefficient absolute value* decision is YES. If the value returned by the Z′-Coder is **0**, then the value of the *increase coefficient absolute value* decision is NO.

The Z′-Coder context for the *increase coefficient absolute value* decision for each color component is initially set to 0.

7.7 Image reconstruction

At any time during the decoding process, an image may be reconstructed from the current values of the wavelet coefficients already decoded. The wavelet coefficients are stored in three two-dimensional arrays one for each of the $\mathbf{Y}$, $\mathbf{C_b}$, and $\mathbf{C_r}$ color components. Each array has one entry for each image block. Each entry itself is a 1024-element one-dimensional array. The elements of each one-dimensional array are the wavelet coefficients. The wavelet coefficients are signed fixed-point numbers with six fractional bits.

7.7.1 Sequence of operations

To reconstruct the image from the coefficients, the following steps must be performed:

1. **Reordering coefficients.** For each color component, each of the 1024-element coefficient arrays is converted into a 32×32 coefficient array. These square coefficient arrays are embedded into a larger reconstruction array whose size is the size of the image.

2. **Inverse wavelet transform.** For each color component, the inverse wavelet transform is applied to the larger reconstruction array. The inverse transform is applied at progressively finer scales, and within each scale in each of the two directions, first vertically, then horizontally.

3. **Precision reduction.** For each color component, the data values in the reconstruction array are reduced to eight bits.

4. **Conversion to RGB color space.** For color images, the eight-bit values of each pixel in the $\mathbf{YC_bC_r}$ color space are converted to the corresponding eight-bit values in the **RGB** color space.

7.7.2 Coordinate system

For indexing the blocks within a color component, the origin $(0, 0)$ is at the lower left corner of the image. Horizontal indices increase rightward, and vertical indices increase upward.

For indexing the coefficients within a 32×32 block coefficient array, the origin $(0, 0)$ is at the lower left corner of the block. Horizontal indices increase rightward, and vertical indices increase upward.

For indexing the coefficients and color values within the image in the reconstruction array, the origin $(0, 0)$ is at the lower left corner of the image. Horizontal indices increase rightward, and vertical indices increase upward.

When the array of coefficients for a block is embedded into the reconstruction array, the origin of the block coefficient array is placed into the lower left corner of the section of the reconstruction array that corresponds to the block.

7.7.3 Reordering coefficients

Within each color component, the coefficients in each block are moved from a 1024-element linear array into a 32×32 square array. The square array from each block is embedded in a reconstruction array the size of the full image.

The mapping from indices in the linear array to indices in the square array is as follows: if the ten bits of the index in the linear array are $\mathbf{b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0}$, $\mathbf{b_9}$ being the most significant bit of the index, then the bits of the row index of the square array are $\mathbf{b_1 b_3 b_5 b_7 b_9}$, $\mathbf{b_1}$ being the most significant bit of the row index, and the bits of the column index of the square array are $\mathbf{b_0 b_2 b_4 b_6 b_8}$, $\mathbf{b_0}$ being the most significant bit of the column index.

If the number of rows in the image is not a multiple of 32, then blocks along the top edge of the image have fewer than 32 rows. If the number of columns in the image is not a multiple of 32, then blocks along the right edge of the image have fewer than 32 columns. For all such blocks, all coefficients are coded; however, coefficients that fall outside the boundary of the image after the coefficient mapping described above are never used, regardless of their value.

7.7.4 Inverse wavelet transform

The inverse transformation from wavelet coefficients to color values is done independently for the three color components. Within a color component the transformation is done for a decreasing sequence of scale parameters s. For a given scale parameter s, the transformation is done first for columns, then for rows. Within a column or row, the transformation is done in two passes, a lifting pass and then a prediction pass.

The scale parameter's initial value is $s = 16$. After the vertical and horizontal transformations have been done with a given value of s, the value of s is divided by 2 and the next pair of transformations is performed. After the vertical and horizontal transformations have been performed with $s = 1$, the inverse wavelet transform for the color component is complete.

The pair of transformations for a given value of s involve only rows and columns whose indices are multiples of s. The vertical transformation involves transforming the coefficients in column 0 whose row indices are multiples of s, then repeating the transformation for all other columns whose column indices are multiples of s. Some of the coefficients transformed by the vertical transformation will already have been transformed during earlier iterations with larger values of the scale parameter s.

The horizontal transformation involves transforming the coefficients in row 0 whose column indices are multiples of s, then repeating the transformation for all other rows whose row indices are multiples of s. The coefficients transformed by the horizontal transformation will have been transformed by the vertical transformation during the first pass for the current scale parameter s. Some of the coefficients transformed by the horizontal transformation will already have been transformed during earlier iterations with larger values of the scale parameter s.

To transform one column or row of coefficients:

1. If transforming a column, select the coefficients in the current column that come from rows whose indices are multiples of s. The coefficient from the row whose index is ks is referred to as c_k. The largest value of k is referred to as $k_{\max}$.

   If transforming a row, select the coefficients in the current row that come from columns whose indices are multiples of s. The coefficient from the column whose index is ks is referred to as c_k. The largest value of k is referred to as $k_{\max}$.

2. **Lifting.** For each even-numbered subscript k, $0 \le k \le k_{\max}$, replace coefficient c_k with
$$c_k - \left\lfloor \frac{9(c_{k-1} + c_{k+1}) - (c_{k-3} + c_{k+3}) + 16}{32} \right\rfloor .$$
   Special cases: If $k - 3 < 0$, use $c_{k-3} = 0$. If $k - 1 < 0$, use $c_{k-1} = 0$. If $k + 1 > k_{\max}$, use $c_{k+1} = 0$. If $k + 3 > k_{\max}$, use $c_{k+3} = 0$.

3. **Prediction.** For each odd-numbered subscript k, $0 < k \le k_{\max}$, modify coefficient c_k as follows:

   (a) If $k - 3 \ge 0$ and $k + 3 \le k_{\max}$, replace c_k with
$$c_k + \left\lfloor \frac{9(c_{k-1} + c_{k+1}) - (c_{k-3} + c_{k+3}) + 8}{16} \right\rfloor .$$

(b) Otherwise, if $k+1 \leq k_{\max}$, replace c_k with

$$c_k + \left\lfloor \frac{c_{k-1} + c_{k+1} + 1}{2} \right\rfloor .$$

(c) Otherwise, replace c_k with

$$c_k + c_{k-1}.$$

7.7.5 Precision reduction for color image data

After the inverse transformation, a color value in the reconstruction array for each color component is a signed fixed-point value with 6 fractional bits. This value is to be rounded to the nearest integer V. Then if $V < -128$, V is set to -128. If $V \geq 128$, V is set to 127. Finally, in the luminance ($\mathbf{Y}$) color component only, V is increased by 128.

7.7.6 Precision reduction for grayscale image data

After the inverse transformation, a grayscale value in the reconstruction array is a signed fixed-point value with 6 fractional bits. This value is to be rounded to the nearest integer V. Then if $V < -128$, V is set to -128. If $V \geq 128$, V is set to 127. Finally, V is replaced by $127 - V$.

7.7.7 Conversion from $\mathbf{YC_bC_r}$ color space to RGB color space

For a color image, each pixel has a value in each of the color component reconstruction buffers. To convert a pixel's $\mathbf{YC_bC_r}$ values to the corresponding **RGB** values, perform the following transformation:

$$\begin{array}{rcllll} \mathbf{R} & = & \mathbf{Y} & & & +(3/2)\mathbf{C_r} \\ \mathbf{G} & = & \mathbf{Y} & -(1/4)\mathbf{C_b} & & -(3/4)\mathbf{C_r} \\ \mathbf{B} & = & \mathbf{Y} & +(7/4)\mathbf{C_b} & & \end{array}$$

8 Requirements: Selection layer and black and white coding

8.1 General considerations.

Selection layer coding is used in Compound DJVU Images. In such images, there are three layers. The foreground layer is coded in one **FG44** chunk, and is rendered as described in Section 7. The background layer is coded in one or more **BG44** chunks, and is rendered as described in Section 7. The selection layer is coded using one **Sjbz** chunk. Black pixels in the selection layer specify those pixels that are to be rendered using the foreground color. All other pixels are to be rendered using the background color.

Black and white coding is used in Bilevel DJVU Images. In such images, there are three layers. The foreground layer is black. The background layer is white. The selection layer is coded using one **Sjbz** chunk. The selection layer specifies those pixels that are to be rendered in black. All other pixels are to be rendered in white.

An **Sjbz** chunk contains a single arithmetically encoded data stream, coded using the Z′-Coder. All data, including headers and record types, is coded in this arithmetically coded stream.

8.2 Arithmetic coding

The arithmetically coded data in an **Sjbz** chunk consists logically of records. The record types are listed in Table 6, and described in Section 8.4. The records consist of fields. The fields present for records of each record type are listed in Table 6. The fields within a record are coded in the order listed in Table 6 for records of that type. Details of the coding for each field appear in Section 8.5.

A field may contain one or more data elements. The data elements consist of flags, pixel colors, and integers. Because of the nature of arithmetic coding, the records, fields, and data elements are not of fixed sizes, and do not necessarily begin on bit boundaries within the data stream.

Flags are binary decisions, each coded using the Z′-Coder with a particular context. There are two different contexts for flags, the *eventual image refinement* context and the *offset type* context.

Pixel colors are binary decisions, coded using the Z′-Coder with a particular context. For pixel colors, there are 3072 different contexts. There are 1024 contexts used for direct coding of bitmaps; these correspond to the $2^{10} = 1024$ different combinations of values that the pixels in the direct coding template can assume. There are 2048 contexts used for refinement coding of bitmaps; these correspond to the $2^{11} = 2048$ different combinations of values that the pixels in the refinement coding template can assume.

Integers are coded using the multivalue extension to the Z′-Coder, described in Section 8.2.2. There are 15 contexts for coding multivalued integers, as described in Table 7.

8.2.1 Initialization of the Z′-Coder

All Z′-Coder contexts are initialized to the value 0. This applies both to contexts used to encode single bit values, including pixel colors, and to contexts that are part of an integer context used by the multivalue extension to the Z′-Coder.

8.2.2 The multivalue extension to the Z′-Coder for coding of numeric data

Quantities that can take on multiple values are coded as integers using the multivalue extension to the Z′-Coder. This extension of the Z′-Coder allows all data in the bitstream to be coded using the same coder, the Z′-Coder. There are 15 integer contexts, specified in Table 7. A single integer context includes a number of binary contexts.

One integer context consists of a binary decision tree. See Figure 1 for an example of part of such a tree. The root node of the tree corresponds to the decision about the sign of the number n being decoded. Each of the two subtrees under the root corresponds to a set of decisions that eventually identify a range in which n lies. The subtrees under the nodes corresponding to identified ranges are complete binary trees that identify the exact value of n.

Each node of the binary decision tree for an integer context maintains its own binary probability estimation context for the Z′-Coder. The trees for different integer contexts are completely independent. Thus each node of a tree contains probability information conditioned on a conditioning context. The conditioning context consists of both the type of value being coded (i.e., the selection of the integer context), and of the values of the decisions coded so far when encoding the current integer.

Record type coded value	Record type	Fields coded
0	Start of image	Record type Image size Eventual image refinement flag
1	New symbol, add to image and library	Record type Absolute symbol size Bitmap by direct coding Location relative to a previous symbol
2	New symbol, add to library only	Record type Absolute symbol size Bitmap by direct coding
3	New symbol, add to image only	Record type Absolute symbol size Bitmap by direct coding Location relative to a previous symbol
4	Matched symbol with refinement, add to image and library	Record type Index of matching symbol in bitmap library Relative symbol size Bitmap by refinement coding Location relative to a previous symbol
5	Matched symbol with refinement, add to library only	Record type Index of matching symbol in bitmap library Relative symbol size Bitmap by refinement coding
6	Matched symbol with refinement, add to image only	Record type Index of matching symbol in bitmap library Relative symbol size Bitmap by refinement coding Location relative to a previous symbol
7	Matched symbol, copy to image without refinement	Record type Index of matching symbol in bitmap library Location relative to a previous symbol
8	Non-symbol data	Record type Absolute symbol size Bitmap by direct coding Absolute location
9	Image refinement data	Record type (This record type is treated as end of data: all remaining data in the chunk is skipped)
10	Comment	Record type Comment length Comment data
11	End of data	Record type

Table 6: Record types and fields coded for each record type

Context name	Integer data coded using this context
record type	record type
image size	image height and image width
matching symbol index	index within the symbol library of the symbol matching the current symbol
symbol width	number of pixels in the width of the current symbol
symbol height	number of pixels in the height of the current symbol
symbol width difference	number of pixels that must be added to the width of the matching symbol to obtain the width of the current symbol
symbol height difference	number of pixels that must be added to the height of the matching symbol to obtain the height of the current symbol
symbol column number	column number of the absolute location of the left edge of the current symbol (leftmost column of the image is column number 1)
symbol row number	row number of the absolute location of the top edge of the current symbol (bottom row of the image is row number 1)
same line column offset	number of pixels that must be added to the column number of the right edge of the previous symbol on the current text line to obtain the column number of the left edge of the current symbol
same line row offset	number of pixels that must be added to the row number of the current baseline on the current text line to obtain the row number of the bottom edge of the current symbol
new line column offset	number of pixels that must be added to the column number of the left edge of the first symbol on the current text line to obtain the column number of the left edge of the current symbol
new line row offset	number of pixels that must be added to the row number of the bottom edge of the first symbol on the current text line to obtain the row number of the top edge of the current symbol
comment length	the number of octets in the current comment
comment octet	one octet in the current comment

Table 7: Multivalued integer contexts for arithmetic coding

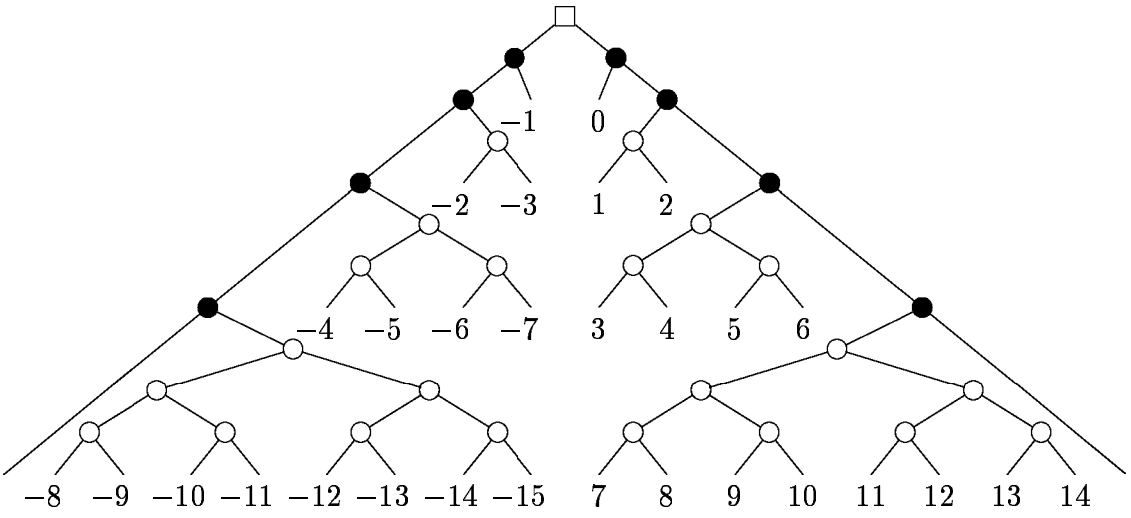

Figure 1: Part of the coding tree for multisymbol arithmetic coding. Each internal node represents one context with its own probability information, to be used by the Z'-Coder. The square node at the root of the tree represents the Phase 1 decision, whether the integer n being coded is negative. The filled circles are the Phase 2 nodes, moving down the tree in ever-increasing ranges. The open circles represent Phase 3 decisions, traversing a complete binary subtree to reach the specific value of n. A decoded value of **0** indicates a left branch in this tree. A decoded value of **1** indicates a right branch.

This method allows high compression efficiency by allowing the coder to adapt to the statistics of the data. In effect, the binary probability information stored collectively in the nodes of the decision tree closely approximates the probability distribution of the underlying multivalued integer.

The allowable range of values for n is always specified. The smallest value that n could possibly take is denoted by l. The largest number that n could possibly take is denoted by h. When l and h are equal, n is equal to both of them, and no Z′-Coder decoding is performed.

The decoder maintains a non-negative intermediate value v, defined as follows:

$$v = \begin{cases} |n| & \text{if } n \geq 0 \\ |n| - 1 & \text{if } n < 0. \end{cases}$$

At the end of the process of decoding an integer, v is converted to n, the value of the decoded integer.

The value of an integer is coded by making a sequence of binary decisions, each one narrowing the set of possible values that the integer can possibly take. The decisions are based on traversing a binary decision tree to one of its leaves. **Note:** *although the tree conceptually has a large number of nodes, it is possible in an implementation to allocate memory only for those nodes actually traversed.*

Decoding proceeds in four phases.

8.2.2.1 Phase 1. Phase 1 determines the sign of n. A value of **0** returned by the Z′-Coder means that $n < 0$. A value of **1** returned by the Z′-Coder means that $n \geq 0$.

8.2.2.2 Phase 2. Phase 2 determines a range of possible values for v. The Z′-Coder is invoked repeatedly to answer the question "Is the value of v in the range being tested?" The sequence of ranges tested is given in Table 8. A value of **0** returned by the Z′-Coder means that v is not in the specified range, and the next range in the sequence must be tested. A value of **1** returned by the Z′-Coder means that v is in the specified range, and decoding is to proceed to Phase 3.

0
1−2
3−6
7−14
15−30
31−62
63−126
127−254
255−510
511−1022
1023−2046
2047−4094
4095−8190
8191−16382
16383−32766
32767−65534
65535−131070
131071−262142

Table 8: Sequence of ranges in which v may fall.

8.2.2.3 Phase 3. Phase 3 consists of determining the exact value of v within the range determined in Phase 2. If Phase 2 determined that $v = 0$, then Phase 3 is skipped.

Otherwise, since the size of the range is a power of 2, the corresponding subtree is a complete binary tree. The sequence of coding decisions is based directly on traversing the binary tree. At each node, **0** returned by the Z′-Coder means left branch (smaller values of v) and **1** means right branch (larger values of v). The bits returned by the Z′-Coder during Phase 3 are the bits of v, most significant bit first.

8.2.2.4 Phase 4. In Phase 4, the unsigned value v is converted to n, the signed value to be returned, as follows:

$$n = \begin{cases} v & \text{if } n \text{ is non-negative, as determined in Phase 1;} \\ -v-1 & \text{if } n \text{ is negative, as determined in Phase 1.} \end{cases}$$

In any of the phases, if the input values of l and h (the range of allowable values) predetermine any decision, then the coding for that decision is not performed; the predetermined decision is assumed.

Each type of integer has its own set of binary contexts. Thus the probability information will reflect the underlying probability distribution of the particular type of integer. The Z′-Coder probability state indices of all the binary nodes are initialized to 0.

8.3 Image reconstruction

Records in an **Sjbz** chunk are interpreted in the order in which they appear. A start of data record specifies the dimensions of the image. An image refinement data record indicates the end of the **Sjbz** chunk. An end of data record indicates the end of the **Sjbz** chunk. A comment record contains uninterpreted data.

A record identified by any other record type describes one bitmap. The model used in DjVu for the selection layer is based on symbol-based coding. Bitmaps are placed into the reconstructed image as follows: The image is initially entirely white. When a bitmap is placed into the image, the pixels that are black in the current symbol become black at the appropriate position in the reconstructed image. Once a pixel in the reconstructed image becomes black, it remains black.

Because symbols in document images are often similar to each other, it is often possible to obtain more efficient coding by making use of previously coded symbols. As symbols are decoded, their bitmaps may be placed into a symbol bitmap library. There is exactly one symbol bitmap library. Once a symbol has been placed into the symbol bitmap library, later records may cause copies of the symbol to be placed into the image, or may define a new bitmap by refining the bitmap in the library.

Depending on the record type, the symbol bitmap may be described by direct coding, by refinement coding, or by a copy operation. In direct coding, all pixels of the bitmap are coded directly, without reference to any other bitmap. In refinement coding, all pixels of the bitmap are also coded directly, but a bitmap in the library is used to make the coding more efficient. In a copy operation, the pixels of the bitmap are the same as the pixels of a bitmap in the library.

Depending on the record type, the bitmap may or may not be placed into the image. If the bitmap is placed into the image, then depending on the record type, it may be placed either at an absolute location or at a location relative to a previously placed bitmap.

Depending on the record type, the bitmap may or may not be placed into the symbol bitmap library. The first symbol placed into the library has index 0. Subsequent symbols are assigned consecutive integer indices.

The pixels of the reconstructed image are arranged in a rectangular coordinate system. For the pixel in the lower left corner of the image, the column number is 1 and the row number is 1. All coordinates refer to the pixels themselves, not to the edges between pixels.

8.4 Records

Records in **Sjbz** chunks have the following interpretations.

8.4.1 Start of image

A start of image record is the first record in an **Sjbz** chunk. It specifies the dimensions of the image.

8.4.2 New symbol, add to image and library

A new symbol, add to image and library record specifies the bitmap of a symbol that is coded directly and placed into the reconstructed image and into the symbol bitmap library.

8.4.3 New symbol, add to library only

A new symbol, add to library only record specifies the bitmap of a symbol that is coded directly and placed into the symbol bitmap library but not into the image.

8.4.4 New symbol, add to image only

A new symbol, add to image only record specifies the bitmap of a symbol that is coded directly and placed into the reconstructed image but not into the symbol bitmap library.

8.4.5 Matched symbol with refinement, add to image and library

A matched symbol with refinement, add to image and library record specifies the bitmap of a symbol that is coded by refinement of a symbol in the symbol bitmap library and placed into the reconstructed image and into the symbol bitmap library.

8.4.6 Matched symbol with refinement, add to library only

A matched symbol with refinement, add to library only record specifies the bitmap of a symbol that is coded by refinement of a symbol in the symbol bitmap library and placed into the symbol bitmap library, but not into the reconstructed image.

8.4.7 Matched symbol with refinement, add to image only

A matched symbol with refinement, add to image only record specifies the bitmap of a symbol that is coded by refinement of a symbol in the symbol bitmap library and placed into the reconstructed image, but not into the symbol bitmap library.

8.4.8 Matched symbol, copy to image without refinement

A matched symbol, copy to image without refinement record specifies the location at which the bitmap of a symbol in the symbol bitmap library is to be placed into the reconstructed image.

8.4.9 Non-symbol data

A non-symbol data record specifies a direct coded bitmap to be placed at an absolute location in the reconstructed image. A bitmap of non-symbol data is not placed into the symbol bitmap library.

8.4.10 Image refinement data

An image refinement data record is treated like an end of data record. All data beginning with such a record until the end of the **Sjbz** chunk is skipped.

8.4.11 Comment

A comment record contains data whose interpretation is not specified by the standard.

8.4.12 End of data

An end of data record is the last record of an **Sjbz** chunk.

8.5 Fields

The following fields are coded in records of types specified in Table 6 and in Section 8.4.

8.5.1 Record type

The record type is coded by the multivalue extension to the Z′-Coder using the *record type* context. The range of allowable record types is from 0 to 11. The coded values are specified in the first column of Table 6.

8.5.2 Image size

The width and height of the image are coded by the multivalue extension to the Z′-Coder using the *image size* context. The width is coded first, then the height. The range of allowable values is from 0 to 262142. The width and height of a Compound DJVU Image or Bilevel DJVU Image must be the same as the width and height of the image specified in the **INFO** chunk.

8.5.3 Eventual image refinement flag

The EVENTUAL IMAGE REFINEMENT flag is coded once, in the start of image record, to notify the decoder whether image refinement data will eventually be provided. It is a binary value, coded by the Z′-Coder using the *eventual image refinement* context. The coded value **1** means TRUE and the coded value **0** means FALSE. **Note:** *This flag is always* FALSE *in the current version of the standard, but it may be* TRUE *in later versions.*

8.5.4 Index of matching symbol in bitmap library

The index of the matching symbol in the bitmap library is coded with the multivalue extension to the Z′-Coder, using the *matching symbol index* context. The range of allowable values is from 0 to one less than the number of symbols currently in the bitmap library.

8.5.5 Absolute symbol size

The width of a symbol is coded by the multivalue extension to the Z′-Coder, using the *symbol width* context. Then the height of a symbol is coded by the multivalue extension to the Z′-Coder, using the *symbol height* context. The range of allowable values for both of these data elements is from 0 to 262142.

8.5.6 Relative symbol size

The signed differences between the width and height of the current symbol and the width and height respectively of the matching symbol are coded by the multivalue extension to the Z′-Coder using the *symbol width difference* context for the width and using the *symbol height difference* context for the height. The width difference is coded first, then the height difference. The coded signed difference is added to the width or height of the matching symbol to obtain the width or height respectively of the current symbol. The range of allowable values for both of these data elements is −262143 to 262142.

8.5.7 Absolute location

The horizontal and vertical positions of the upper left corner of the bitmap are coded by the multivalue extension to the Z′-Coder using the *symbol column number* context for the

horizontal position and the *symbol row number* context for the vertical position. The horizontal position is coded first, then the vertical position. The range of allowable values for the horizontal position is from 1 to the number of pixels in the width of the image. The range of allowable values for the vertical position is from 1 to number of pixels in the height of the image.

8.5.8 Location relative to a previous symbol

The OFFSET TYPE flag is coded by the Z′-Coder using the *offset type* context. It indicates the reference symbol for coding the offset of the location of the current symbol. The coded value **1** means FIRST, which means that the location of the current symbol is being specified relative to the first symbol on the current text line. The value **0** means PREVIOUS, which means that the location of the current symbol is being specified relative to the most recently coded symbol on the current text line.

If the OFFSET TYPE flag is FIRST, then the reference symbol is the first symbol on the current text line. The horizontal offset is the signed difference between the left edge of the current symbol and the left edge of the reference symbol. It is coded with the multivalue extension to the Z′-Coder using the *new line column offset* context. The coded signed difference is added to the column number of the left edge of the reference symbol to obtain the column number of the left edge of the current symbol. The vertical offset is the signed difference between the top edge of the current symbol and the bottom edge of the reference symbol. It is coded by the multivalue extension to the Z′-Coder using the *new line row offset* context. The coded signed difference is added to the row number of the bottom of the reference symbol to obtain the row number of the top edge of the current symbol. The current symbol is then treated as the first symbol of a new text line. In this case, the horizontal offset is coded first, then the vertical offset.

If the OFFSET TYPE flag is PREVIOUS, then the reference symbol is the most recently coded symbol on the current text line. The horizontal offset is the signed difference between the left edge of the current symbol and the right edge of the reference symbol. It is coded by the multivalue extension to the Z′-Coder using the *same line column offset* context. The coded signed difference is added to the column number of the right edge of the reference symbol to obtain the column number of the left edge of the current symbol. The vertical offset is the signed difference between the bottom edge of the current symbol and the current baseline. The current baseline is the median of the bottom edges of the three most recently coded symbols on the current line, if there are at least three symbols on the current line. If there are fewer than three previously coded symbols on the current line, the baseline is the bottom edge of the first symbol on the current line. The vertical offset is coded by the multivalue extension to the Z′-Coder using the *same line row offset* context. The coded signed difference is added to the row number of the current baseline to obtain the row number of the bottom edge of the current symbol. In this case, the horizontal offset is coded first, then the vertical offset.

The first symbol in the image is coded as if it were relative to the first symbol on the current text line. The pixel in the upper left corner of the image is taken to be the bottom left corner of this "first symbol." Then the first symbol in the image is treated as the first symbol of a new text line.

8.5.9 Bitmap by direct coding

Non-symbol bitmaps and symbol bitmaps with no sufficiently closely matching symbol in the symbol library are coded directly. A directly coded bitmap is coded by repeated applications of the Z′-Coder to the pixels of the bitmap left to right across the rows, starting with the top row. When one row has been coded, the next lower row is coded. Each pixel is coded by the Z′-Coder using an appropriate context based on the values of 10 previously coded pixels. A coded value of **1** means the pixel is BLACK. A coded value of **0** means the pixel is WHITE. The colors of the pixels numbered 1 through 10 in Figure 2, taken collectively, form a 10-bit value. Each of these values is an index into a table of 1024 different *direct coded bitmap* contexts. The pixel labeled P in Figure 2 is coded using the context indexed by the collective values of the other 10 numbered pixels in the template.

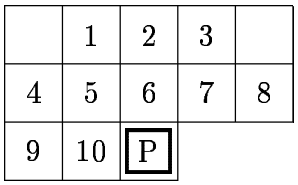

Figure 2: Template for direct coding

Pixels outside the bounding box of the bitmap being coded are considered to be white.

8.5.10 Bitmap by refinement coding

Some bitmaps are coded by making use of data from another bitmap; this process is called refinement coding. Matched symbols other than those to be copied are coded using refinement coding.

A bitmap coded by refinement coding is coded by repeated applications of the Z′-Coder to the pixels of the bitmap left to right across the rows. When one row has been coded, the next lower row is coded. Each pixel is coded by the Z′-Coder using an appropriate context based on the values of 4 previously coded pixels from the bitmap being coded and 7 pixels from the matching bitmap. (The pixels numbered 1 through 4 in Figure 3 are from the current symbol; the pixels numbered 5 through 11 are from the matching symbol.) A coded value of **1** means the pixel is BLACK. A coded value of **0** means the pixel is WHITE. The colors of the pixels numbered 1 through 11 in Figure 3, taken collectively, form an 11-bit value. Each of these values is an index into a table of 2048 different *refinement coded bitmap* contexts. The pixel labeled P in Figure 3 is coded using the context indexed by the collective values of the 11 numbered pixels in the template. Pixel 7 is in the position in the matching symbol that corresponds to the position of pixel P in the current symbol when the two symbols are aligned.

Alignment of the current bitmap and the matching bitmap proceeds as follows. For matched symbols, the current symbol and the matching symbol are aligned according to the geometric centers of their bounding rectangles. If the number of columns or rows is even, the geometric center falls between two columns or rows, respectively. In this case, the leftmost of the two central columns or the lowermost of the two central rows is considered to be the center column or row, respectively.

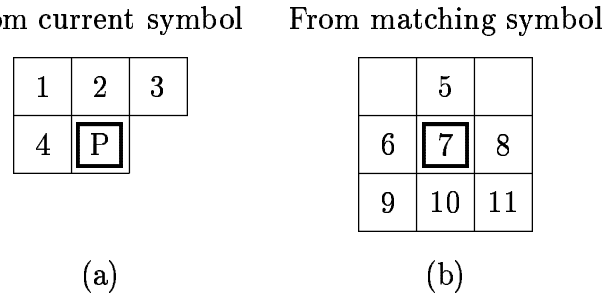

Figure 3: Template for refinement coding. (a) Pixels from symbol being coded. (b) Pixels from matching symbol.

It is possible for the current symbol to have empty edge rows or columns. These empty rows and columns are coded, and are included in the bounding rectangle. For symbols added to the library, the symbol is added to the library after if has been placed into the image. Any empty edge rows and columns are removed before the symbol is added to the library.

8.5.11 Comment length

The comment length is the number of octets in the comment. It is coded by the multivalue extension to the Z′-Coder using the *comment length* context. The range of allowable values for the comment length is from 0 to 262142.

8.5.12 Comment data

Comment data consists of the individual octets of the comment. The number of octets in the comment is given by the comment length field. Each of the octets is coded using the multivalue extension to the Z′-Coder using the *comment octet* context. The range of allowable values for each octet is from 0 to 255.

9 Requirements: The Z′-Coder for binary arithmetic coding

The Z'-Coder is an approximate binary arithmetic coder. Decoding proceeds as follows.

9.1 Registers and data storage

In Figure 4 and Figure 5, the values of variables A, C, D, and Z are stored in registers of at least 16 bits each. A and C retain their values between invocations of the Z'-Coder. The values of D and Z are recomputed during each invocation of the Z'-Coder. **Note:** *If register overflow can be ignored, storing variables A and C in registers of exactly 16 bits allows a simplification of lines 11, 12, 16, and 17 of Figure 4 and lines 8, 9, 12, and 13 of Figure 5.*

At the beginning of a chunk, the values of A and C are reinitialized. When the decoder is decoding a chunk, it may require more bits than are present within the chunk's data. In this case, all additional required bits are to be assumed by the decoder to be **1**. If there are excess bits at the end of a chunk, they are ignored.

K is conceptually an array with a single 8-bit entry for each binary decision context. (In practice, K consists of a number of individual values, arrays, and tree nodes, but each one has a specific address and a single 8-bit value at any time.) This array is indexed by the

value of i, which is the input to the decoder. $K(i)$ is the current value of the probability state index for context i. $K(i)$ may be updated as part of the decoding process.

In pass-through mode, the decoder is invoked with no input argument. No context is involved.

B is the 1-bit value returned by the decoder.

The Z'-Coder is state-based. Decoding is governed by 4 fixed tables, given in Table 9. The tables are indexed by $K(i)$, the probability state index for the current context. All probability state indices are initialized to 0. That is, at the beginning of coding, for all i, $K(i) = 0$. These values are not reinitialized at the beginning of chunks after the first.

The more probable symbol is denoted by MPS. The MPS is 1 if the probability state index is an odd integer, and 0 if the probability state index is an even integer. The less probable symbol is denoted by LPS. The LPS is 0 if the probability state index is an odd integer, and 1 if the probability state index is an even integer.

Δ_k is the amount by which the current arithmetic coding interval is reduced if the decoded symbol is the MPS. θ_k is the threshold above which an MPS triggers a probability state update. μ_k is the next probability state index for context k after an MPS triggers a probability state index update. An LPS always triggers a probability state index update. λ_k is the next probability state index for context k after an LPS.

9.2 Initialization

Initially A is set to `0x0000`. Two octets are read from the input data stream into the lowest 16 bits of C. If the bits of C are numbered such that bit 15 is the most significant bit and bit 0 is the least significant bit, then the first input octet is stored in bits 15 through 8, and the second input octet is stored in bits 7 through 0.

9.3 Decoding

Figure 4 shows the steps involved in decoding a single binary decision. The input to the decoder is the index i of the appropriate context for the binary decision being decoded. The output from the decoder is a single bit B.

9.3.1 Notes on specific lines of Figure 4

Line 2. The division is a right shift, discarding the two least significant bits.

Lines 4-8. These lines are executed when the decoded bit is the MPS.

Line 5. This line determines the value of MPS from the odd/even parity of the probability state index.

Line 6. Sometimes an MPS event triggers an update of the probability state index, based on the value of θ_k. Note that when the probability state index $k = 0$ or $k \geq 83$, $\theta_k = 0$, so an MPS will trigger an update of the probability state index. All probability state indices are initialized to 0, but the first coded decision for a context causes the index to become larger than 83. When $k = 0$ or $k \geq 83$, the probability estimate for the context is in its early estimation phase. When $0 < k < 83$, the probability estimate for the context is in its steady state phase, which it never leaves.

Lines 9-14. These lines are executed when the decoded bit is the LPS.


```
1   Z := A + Δ_{K(i)}
2   D := 0x6000 + (Z + A)/4
3   if (Z > D) { Z := D}
4   if (C > Z) {
5         B := K(i) (mod 2)
6         if (A ≥ θ_{K(i)}) { K(i) = μ_{K(i)}}
7         A := Z
8   }
9   else {
10        B := 1 − (K(i) (mod 2))
11        A := A + 0x10000 − Z
12        C := C + 0x10000 − Z
13        K(i) = λ_{K(i)}
14  }
15  while (A ≥ 0x8000) {
16        A := A + A − 0x10000
17        C := C + C − 0x10000 + next code bit
18  }
19  return B
```

Figure 4: Decoder for Z'-Coder.

Line 10. This line determines the value of LPS from the odd/even parity of the probability state index.

Line 13. An LPS always triggers an update of the probability state index.

Lines 15-18. When the values in the registers are too large, they must be renormalized.

Lines 16-17. $A+A$ and $C+C$ may be accomplished by left shifts, leaving the least significant bit equal to 0.

Line 17. The least significant bit of C is filled with the next bit from the input stream. Bits are taken from each octet in the input stream most significant bit first.

9.4 Pass-through decoding

Figure 5 shows the steps involved in decoding a single binary decision using the Z'-Coder in pass-through mode. No input is required. No context is involved. No probability state index values are updated. The output from the decoder is the single bit B.

9.4.1 Notes on specific lines of Figure 5

Line 1. The division is a right shift, discarding the three least significant bits.

Lines 2-5. These lines are executed when the decoded bit is **0**.

Lines 6-10. These lines are executed when the decoded bit is **1**.

Lines 11-14. When the values in the registers are too large, they must be renormalized.


```
1   Z := 0x8000 + (A + A + A)/8
2   if (C > Z) {
3       B := 0
4       A := Z
5   }
6   else {
7       B := 1
8       A := A + 0x10000 − Z
9       C := C + 0x10000 − Z
10  }
11  while (A ≥ 0x8000) {
12      A := A + A − 0x10000
13      C := C + C − 0x10000 + next code bit
14  }
15  return B
```

Figure 5: Decoder for Z'-Coder operating in pass-through mode.

Lines 12-13. $A+A$ and $C+C$ may be accomplished by left shifts, leaving the least significant bit equal to 0.

Line 13. The least significant bit of C is filled with the next bit from the input stream. Bits are taken from each octet in the input stream most significant bit first.

k	Δ_k	θ_k	μ_k	λ_k	k	Δ_k	θ_k	μ_k	λ_k
0	0x8000	0x0000	84	145	42	0x0861	0x79EA	44	40
1	0x8000	0x0000	3	4	43	0x0711	0x7AE7	45	41
2	0x8000	0x0000	4	3	44	0x0711	0x7AE7	46	42
3	0x6BBD	0x10A5	5	1	45	0x05F1	0x7BBE	47	43
4	0x6BBD	0x10A5	6	2	46	0x05F1	0x7BBE	48	44
5	0x5D45	0x1F28	7	3	47	0x04F9	0x7C75	49	45
6	0x5D45	0x1F28	8	4	48	0x04F9	0x7C75	50	46
7	0x51B9	0x2BD3	9	5	49	0x0425	0x7D0F	51	47
8	0x51B9	0x2BD3	10	6	50	0x0425	0x7D0F	52	48
9	0x4813	0x36E3	11	7	51	0x0371	0x7D91	53	49
10	0x4813	0x36E3	12	8	52	0x0371	0x7D91	54	50
11	0x3FD5	0x408C	13	9	53	0x02D9	0x7DFE	55	51
12	0x3FD5	0x408C	14	10	54	0x02D9	0x7DFE	56	52
13	0x38B1	0x48FD	15	11	55	0x0259	0x7E5A	57	53
14	0x38B1	0x48FD	16	12	56	0x0259	0x7E5A	58	54
15	0x3275	0x505D	17	13	57	0x01ED	0x7EA6	59	55
16	0x3275	0x505D	18	14	58	0x01ED	0x7EA6	60	56
17	0x2CFD	0x56D0	19	15	59	0x0193	0x7EE6	61	57
18	0x2CFD	0x56D0	20	16	60	0x0193	0x7EE6	62	58
19	0x2825	0x5C71	21	17	61	0x0149	0x7F1A	63	59
20	0x2825	0x5C71	22	18	62	0x0149	0x7F1A	64	60
21	0x23AB	0x615B	23	19	63	0x010B	0x7F45	65	61
22	0x23AB	0x615B	24	20	64	0x010B	0x7F45	66	62
23	0x1F87	0x65A5	25	21	65	0x00D5	0x7F6B	67	63
24	0x1F87	0x65A5	26	22	66	0x00D5	0x7F6B	68	64
25	0x1BBB	0x6962	27	23	67	0x00A5	0x7F8D	69	65
26	0x1BBB	0x6962	28	24	68	0x00A5	0x7F8D	70	66
27	0x1845	0x6CA2	29	25	69	0x007B	0x7FAA	71	67
28	0x1845	0x6CA2	30	26	70	0x007B	0x7FAA	72	68
29	0x1523	0x6F74	31	27	71	0x0057	0x7FC3	73	69
30	0x1523	0x6F74	32	28	72	0x0057	0x7FC3	74	70
31	0x1253	0x71E6	33	29	73	0x003B	0x7FD7	75	71
32	0x1253	0x71E6	34	30	74	0x003B	0x7FD7	76	72
33	0x0FCF	0x7404	35	31	75	0x0023	0x7FE7	77	73
34	0x0FCF	0x7404	36	32	76	0x0023	0x7FE7	78	74
35	0x0D95	0x75D6	37	33	77	0x0013	0x7FF2	79	75
36	0x0D95	0x75D6	38	34	78	0x0013	0x7FF2	80	76
37	0x0B9D	0x7768	39	35	79	0x0007	0x7FFA	81	77
38	0x0B9D	0x7768	40	36	80	0x0007	0x7FFA	82	78
39	0x09E3	0x78C2	41	37	81	0x0001	0x7FFF	81	79
40	0x09E3	0x78C2	42	38	82	0x0001	0x7FFF	82	80
41	0x0861	0x79EA	43	39	83	0x5695	0x0000	9	85

Table 9: State tables for the Z'-Coder.

k	Δ_k	θ_k	μ_k	λ_k	k	Δ_k	θ_k	μ_k	λ_k
84	0x24EE	0x0000	86	226	126	0x01EB	0x0000	60	54
85	0x8000	0x0000	5	6	127	0x1302	0x0000	33	25
86	0x0D30	0x0000	88	176	128	0x02E6	0x0000	56	50
87	0x481A	0x0000	89	143	129	0x1B81	0x0000	29	131
88	0x0481	0x0000	90	138	130	0x045E	0x0000	52	46
89	0x3579	0x0000	91	141	131	0x24EF	0x0000	23	17
90	0x017A	0x0000	92	112	132	0x0690	0x0000	48	40
91	0x24EF	0x0000	93	135	133	0x2865	0x0000	23	15
92	0x007B	0x0000	94	104	134	0x09DE	0x0000	42	136
93	0x1978	0x0000	95	133	135	0x3987	0x0000	137	7
94	0x0028	0x0000	96	100	136	0x0DC8	0x0000	38	32
95	0x10CA	0x0000	97	129	137	0x2C99	0x0000	21	139
96	0x000D	0x0000	82	98	138	0x10CA	0x0000	140	172
97	0x0B5D	0x0000	99	127	139	0x3B5F	0x0000	15	9
98	0x0034	0x0000	76	72	140	0x0B5D	0x0000	142	170
99	0x078A	0x0000	101	125	141	0x5695	0x0000	9	85
100	0x00A0	0x0000	70	102	142	0x078A	0x0000	144	168
101	0x050F	0x0000	103	123	143	0x8000	0x0000	141	248
102	0x0117	0x0000	66	60	144	0x050F	0x0000	146	166
103	0x0358	0x0000	105	121	145	0x24EE	0x0000	147	247
104	0x01EA	0x0000	106	110	146	0x0358	0x0000	148	164
105	0x0234	0x0000	107	119	147	0x0D30	0x0000	149	197
106	0x0144	0x0000	66	108	148	0x0234	0x0000	150	162
107	0x0173	0x0000	109	117	149	0x0481	0x0000	151	95
108	0x0234	0x0000	60	54	150	0x0173	0x0000	152	160
109	0x00F5	0x0000	111	115	151	0x017A	0x0000	153	173
110	0x0353	0x0000	56	48	152	0x00F5	0x0000	154	158
111	0x00A1	0x0000	69	113	153	0x007B	0x0000	155	165
112	0x05C5	0x0000	114	134	154	0x00A1	0x0000	70	156
113	0x011A	0x0000	65	59	155	0x0028	0x0000	157	161
114	0x03CF	0x0000	116	132	156	0x011A	0x0000	66	60
115	0x01AA	0x0000	61	55	157	0x000D	0x0000	81	159
116	0x0285	0x0000	118	130	158	0x01AA	0x0000	62	56
117	0x0286	0x0000	57	51	159	0x0034	0x0000	75	71
118	0x01AB	0x0000	120	128	160	0x0286	0x0000	58	52
119	0x03D3	0x0000	53	47	161	0x00A0	0x0000	69	163
120	0x011A	0x0000	122	126	162	0x03D3	0x0000	54	48
121	0x05C5	0x0000	49	41	163	0x0117	0x0000	65	59
122	0x00BA	0x0000	124	62	164	0x05C5	0x0000	50	42
123	0x08AD	0x0000	43	37	165	0x01EA	0x0000	167	171
124	0x007A	0x0000	72	66	166	0x08AD	0x0000	44	38
125	0x0CCC	0x0000	39	31	167	0x0144	0x0000	65	169

Table 9 continued.

k	Δ_k	θ_k	μ_k	λ_k	k	Δ_k	θ_k	μ_k	λ_k
168	0x0CCC	0x0000	40	32	210	0x0BC0	0x0000	40	34
169	0x0234	0x0000	59	53	211	0x030D	0x0000	213	227
170	0x1302	0x0000	34	26	212	0x1178	0x0000	36	28
171	0x0353	0x0000	55	47	213	0x0206	0x0000	215	225
172	0x1B81	0x0000	30	174	214	0x19DA	0x0000	30	22
173	0x05C5	0x0000	175	193	215	0x0155	0x0000	217	223
174	0x24EF	0x0000	24	18	216	0x24EF	0x0000	26	16
175	0x03CF	0x0000	177	191	217	0x00E1	0x0000	219	221
176	0x2B74	0x0000	178	222	218	0x320E	0x0000	20	220
177	0x0285	0x0000	179	189	219	0x0094	0x0000	71	63
178	0x201D	0x0000	180	218	220	0x432A	0x0000	14	8
179	0x01AB	0x0000	181	187	221	0x0188	0x0000	61	55
180	0x1715	0x0000	182	216	222	0x447D	0x0000	14	224
181	0x011A	0x0000	183	185	223	0x0252	0x0000	57	51
182	0x0FB7	0x0000	184	214	224	0x5ECE	0x0000	8	2
183	0x00BA	0x0000	69	61	225	0x0383	0x0000	53	47
184	0x0A67	0x0000	186	212	226	0x8000	0x0000	228	87
185	0x01EB	0x0000	59	53	227	0x0547	0x0000	49	43
186	0x06E7	0x0000	188	210	228	0x481A	0x0000	230	246
187	0x02E6	0x0000	55	49	229	0x07E2	0x0000	45	37
188	0x0496	0x0000	190	208	230	0x3579	0x0000	232	244
189	0x045E	0x0000	51	45	231	0x0BC0	0x0000	39	33
190	0x030D	0x0000	192	206	232	0x24EF	0x0000	234	238
191	0x0690	0x0000	47	39	233	0x1178	0x0000	35	27
192	0x0206	0x0000	194	204	234	0x1978	0x0000	138	236
193	0x09DE	0x0000	41	195	235	0x19DA	0x0000	29	21
194	0x0155	0x0000	196	202	236	0x2865	0x0000	24	16
195	0x0DC8	0x0000	37	31	237	0x24EF	0x0000	25	15
196	0x00E1	0x0000	198	200	238	0x3987	0x0000	240	8
197	0x2B74	0x0000	199	243	239	0x320E	0x0000	19	241
198	0x0094	0x0000	72	64	240	0x2C99	0x0000	22	242
199	0x201D	0x0000	201	239	241	0x432A	0x0000	13	7
200	0x0188	0x0000	62	56	242	0x3B5F	0x0000	16	10
201	0x1715	0x0000	203	237	243	0x447D	0x0000	13	245
202	0x0252	0x0000	58	52	244	0x5695	0x0000	10	2
203	0x0FB7	0x0000	205	235	245	0x5ECE	0x0000	7	1
204	0x0383	0x0000	54	48	246	0x8000	0x0000	244	83
205	0x0A67	0x0000	207	233	247	0x8000	0x0000	249	250
206	0x0547	0x0000	50	44	248	0x5695	0x0000	10	2
207	0x06E7	0x0000	209	231	249	0x481A	0x0000	89	143
208	0x07E2	0x0000	46	38	250	0x481A	0x0000	230	246
209	0x0496	0x0000	211	229					

Table 9 concluded.