WD14492

THIS IS A WORKING DRAFT. THIS IS INTENDED TO BE A COMPLETE DESCRIPTION OF THE TECHNICAL CONTENT OF JBIG2. HOWEVER, ITS TECHNICAL CONTENT MAY CHANGE BEFORE BECOMING A STANDARD. USE AT YOUR OWN RISK.

JBIG committee

21 August 1998

Contents

List of Tables

List of Figures

0 Introduction

This Recommendation | International Standard, informally called JBIG2, defines a coding method for bilevel images, that is, images consisting of a single rectangular bit plane, with each pixel taking on one of just two possible colors. It is being drafted by the Joint Bi-level Image Experts Group (JBIG), a “Collaborative Team”, established in 1988, that reports both to ISO/IEC JTC 1/SC29/WG1 and to ITU-T/SG8.

Compression of this type of image is also addressed by existing facsimile standards,for example by ITU-T Recommendations T.4, T.6, T.82 (JBIG1), and T.85 (Application profile of JBIG1 for facsimile). Besides the obvious facsimile application, JBIG2 will be useful for document storage and archiving, coding images on the World Wide Web, wireless data transmission, print spooling, and even teleconferencing.

As the result of a process that ended in 1993, JBIG produced a first coding standard formally designated ITU-T Recommendation T.82 | International Standard ISO/IEC 11544, which is informally known as JBIG or JBIG1. JBIG1 has the capability of lossy, lossless, and progressive (lossy to lossless) coding. However, the lossy images produced by JBIG1 have quite lower qualities than the original images because the number of pixels in the lossy image cannot exceed one quarter of those in the original image.

JBIG2 was prepared also for lossy, lossless, and lossy-to-lossless image compression. The design goal for JBIG2 was to allow for lossless compression performance better than that of the existing standards, and to allow for lossy compression at much higher compression ratios than the lossless ratios of the existing standards, with almost no visible degradation of quality. In addition, JBIG2 allows both quality-progressive coding, with the progression going from lower to higher (or lossless) quality, and content-progressive coding, successively adding different types of image data (for example, first text, then halftones). A typical JBIG2 encoder decomposes the input bi-level image into several regions and codes each of the regions separately using a different coding method. Such content-based decomposition is very desirable especially in interactive multimedia applications. JBIG2 can also handle a set of images (multiple page document) in an explicit manner.

As is typical with image compression standards, JBIG2 explicitly defines the requirements of a compliant bitstream, and thus defines decoder behavior. JBIG2 does not explicitly define a standard encoder, but instead is flexible enough to allow sophisticated encoder design. In fact, encoder design will be a major differentiator among competing JBIG2 implementations.

0.1 Interpretation and Use of the Requirements

This section is informative and designed to aid in interpreting the requirements of this International Standard. The requirements are written to be as general as possible to allow a large amount of implementation flexibility. Hence the language of the requirements is not specific about applications or implementations. In this section a correspondence is drawn between the general wording of the requirements and the intended use of the standard in typical applications.

0.1.1 Subject matter for JBIG2 coding

JBIG2 may be used to code bi-level documents. A bi-level document contains one or more pages. A typical page contains some text data, that is, some characters of a small size arranged in horizontal or vertical rows. The characters in the text part of a page are called *symbols* in JBIG2. A page may also contain halftone data, that is, grayscale or color photographic images that have been dithered to produce bi-level images. In addition, a page may contain non-text data, such as large characters, line art, and noise. Such non-text data is called *generic* data in JBIG2.

The JBIG2 image model treats text data and halftone data as special cases. It is expected that a JBIG2 encoder will divide the content of a page into a symbol region containing text, a halftone region containing halftone images, and a generic region containing all other data (although in some cases it is better to consider halftones as generic data rather than halftone data).

The various regions may overlap on the physical page. An encoder is permitted to divide a single page into any number of regions, but often three regions will be sufficient, one for symbols, one for halftone images, and the third for text. In some cases, not all types of data may be present, and the page may consist of fewer than three regions.

A text region consists of a number of symbols placed at specified locations on a background. The symbols usually correspond to characters. JBIG2 obtains much of its effectiveness by using individual symbols more than once. To reuse a symbol, an encoder or decoder must have a succinct way of referring to it. In JBIG2, the symbols are collected into one or more symbol dictionaries. A symbol dictionary is a set of bitmaps of symbols, indexed so that a symbol bitmap may be referred to by an index number.

A halftone region consists of a number of halftone patterns placed along a regular grid. The halftone patterns usually correspond to gray-scale values. Indeed, the coding method of the halftone pattern indices is designed as a gray-scale coder. Compression is realized by representing the binary pixels of one grid cell by a single integer, the rendered gray-scale value. This many-to-one mapping may have the effect that edge information present in the original bitmap may be lost by halftone coding. For this reason, lossless or near-lossless coding of halftones will often be better if the halftone is coded with generic coding rather than halftone coding.

0.1.2 Relationship between segments and documents

A JBIG2 file contains the information needed to decode a bi-level document. A JBIG2 file is composed of *segments*. A typical page is coded using several segments. In a simple case, there will be a page information segment, a symbol dictionary segment, a symbol region segment, a halftone dictionary segment, a halftone region segment, and an end-of-page segment. The page information segment provides general information about the page, such as its size and resolution. The dictionary segments collect bitmaps referred to in the region segments. The region segments describe the appearance of the text and halftone regions by referencing bitmaps from a dictionary and specifying where they should appear on the page. The end-of-page segment marks the end of the page.

0.1.3 Structure and use of segments

Each segment contains a segment header, a data header, and data. The segment header is used to convey segment reference information and, in the case of multi-page documents, page association information. A data header gives information used for decoding the data in the segment. The data describes an image region or a dictionary, or provides other information.

Segments are numbered sequentially. A segment may refer to a lower-numbered, or *earlier*, segment. A region segment is always associated with one specific page of the document. A dictionary segment may be be associated with one page of the document, or it may be associated with the document as a whole.

A region segment may refer to one or more earlier dictionary segments. The purpose of such a reference is to allow the decoder to identify symbols in a dictionary segment that are present into the image.

A region segment may refer to an earlier region segment. The purpose of such a reference is to combine the image described by the earlier segment with the current representation of the page.

A dictionary segment may refer to earlier dictionary segments. The symbols added to a dictionary segment may be described directly, or may be described as refinements of symbols described previously, either in the same dictionary segment or in earlier dictionary segments.

A JBIG2 file may be organized in two ways, sequential or random access. In the sequential organization, each segment's segment header immediately precedes the segment's data header and data. In the random access organization, all the segment headers are collected together at the beginning of the file, followed by the data (including data headers) for all the segments, in the same order. This second organization permits a decoder to determine all segment dependencies without reading the entire file. A third organization of JBIG2-encoded data is the embedded organization. In this case a different file format carries JBIG2 segments. The segment header, data header, and data of each segment are stored together, but the embedding file format may store the segments in any order.

0.1.4 Internal representations

Decoded data must be stored. While the standard does not specify how to store it, its decoding model presumes certain data structures, specifically buffers and dictionaries. Figure 1 illustrates major decoder components and associated buffers. In this figure, decoding procedures are outlined in bold lines, and memory components are outlined in non-bold lines. Also, bold arrows indicate that one decoding procedure invokes another decoding procedure; for example, the symbol dictionary decoding procedure invokes the generic region decoding procedure to decode the bitmaps for the symbols that it defines. Non-bold arrows indicate flow of data: the symbol region decoding procedure reads symbols from the symbol memory, and draws them into the page buffer or an auxiliary buffer.

A buffer is a representation of a bitmap. A buffer is intended to hold a large amount of data, typically the size of a page. A buffer may contain the description of a region or of an entire page. Even if the buffer describes only a region, it has information associated with it that specifies its placement on the page. Decoding a region segment modifies the contents of a buffer.

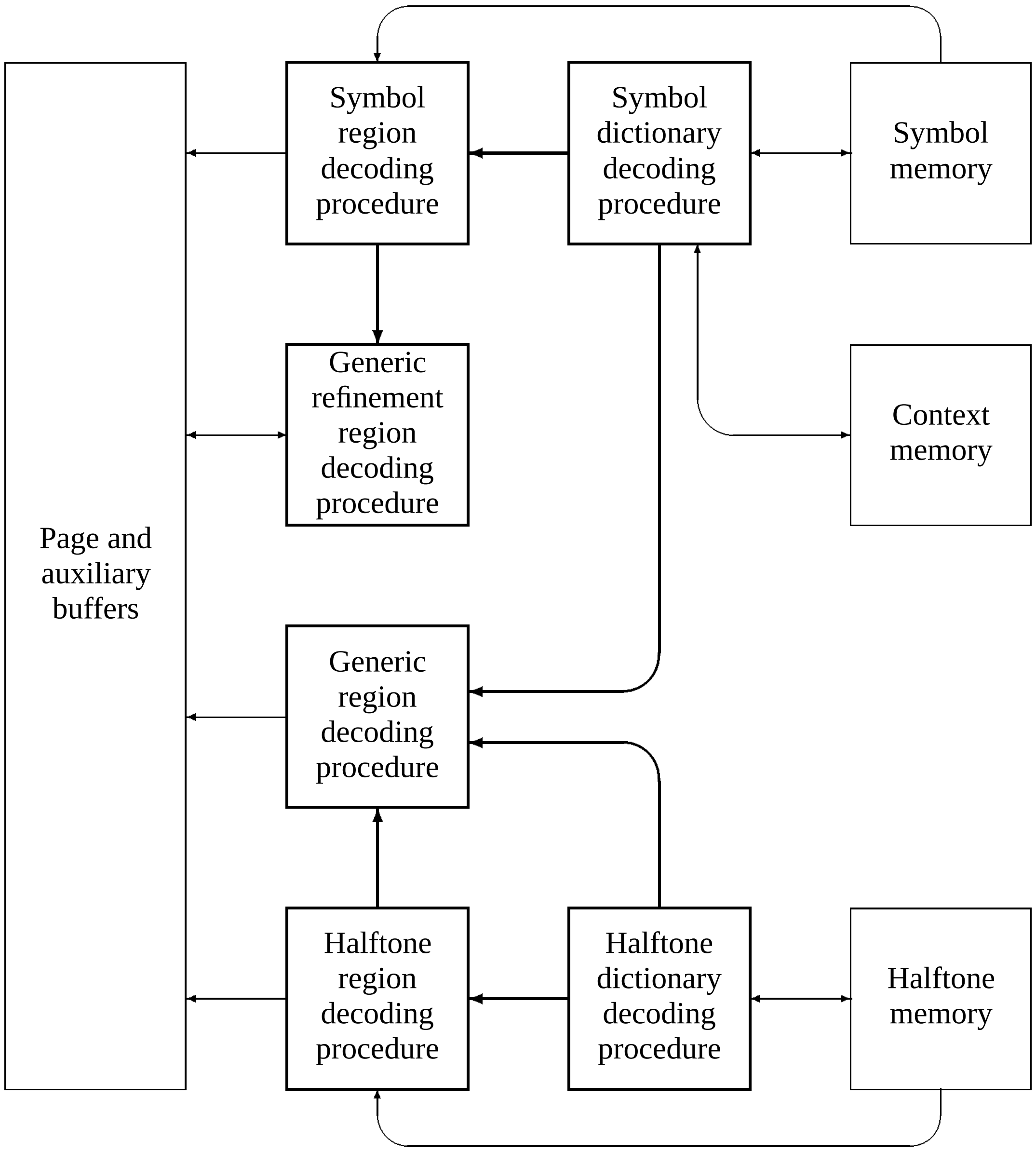

Symbol region decoding procedure
Symbol dictionary decoding procedure
Symbol memory
Generic refinement region decoding procedure
Context memory
Page and auxiliary buffers
Generic region decoding procedure
Halftone region decoding procedure
Halftone dictionary decoding procedure
Halftone memory

Figure 1 — Block diagram of major decoder components.

There is one special buffer, the *page buffer*. It is intended that the decoder accumulate region data directly in the page buffer until the page has been completely decoded; then the data can be sent to an output device or file. Decoding an *immediate* region segment modifies the contents of the page buffer. The usual way of preparing a page is to decode one or more immediate region segments, each one modifying the page buffer. The decoder may output an incomplete page buffer, either as part of progressive transmission or in response to user input. Such output is optional, and its content is not specified by the standard.

All other buffers are auxiliary buffers. It is intended that the decoder fill an auxiliary buffer, then later use it to refine the page buffer. In an application, it will often be unnecessary to have any auxiliary buffers. Decoding an *intermediate* region segment modifies the contents of an auxiliary buffer. The decoder may use auxiliary buffers to output pages other than those found in a complete page buffer, either as part of progressive transmission or in response to user input. Such output is optional, and its content is not specified by the standard.

A symbol dictionary consists of an indexed set of bitmaps. The bitmaps in a dictionary are typically small, approximately the size of text characters. Unlike a buffer, a bitmap in a dictionary does not have page location information associated with it.

0.1.5 Decoding results

Decoding a segment involves invocation of one or more decoding procedures. The decoding procedures to be invoked are determined by the segment type.

The result of decoding a region segment is a bitmap stored in a buffer, possibly the page buffer. Decoding a region segment may fill a new buffer, or may modify an existing buffer. In typical applications, placing the data into a buffer involves changing pixels from the background color to the foreground color, but the standard specifies other permissible ways of changing a buffer's pixels.

A typical page will be described by a number of one or more immediate region segments, each one resulting in modification of the page buffer.

Just as it is possible to specify a new symbol in a dictionary by refining a previously specified symbol, it is also possible to specify a new buffer by refining an existing buffer. However, a region may be refined only by the generic refinement decoding procedure. Such a refinement does not make use of the internal structure of the region in the buffer being refined. After a buffer has been refined, the original buffer is no longer available.

The result of decoding a dictionary segment is a new dictionary. The symbols in the dictionary may later be placed into a buffer by the symbol region decoder.

0.1.6 Decoding procedures

The **generic region decoding procedure** fills or modifies a buffer directly, pixel-by-pixel if arithmetic coding is being used, or by runs of foreground and background pixels if MMR and Huffman coding is being used. In the arithmetic coding case, the prediction context contains only pixels determined by data already decoded within the current segment.

The **generic refinement region decoding procedure** modifies a buffer pixel-by-pixel using arithmetic coding. The prediction context uses pixels determined by data already decoded within the current segment as well as pixels already present either in the page buffer or in an auxiliary buffer.

The **symbol region decoding procedure** takes symbols from one or more symbol dictionaries and places them in a buffer. This procedure is invoked during the decoding of a symbol region segment. The symbol region segment contains the position and index information for each symbol to the placed in the buffer; the bitmaps of the symbols are taken from the symbol dictionaries.

The **symbol dictionary decoding procedure** creates a symbol dictionary, that is, an indexed set of symbol bitmaps. A bitmap in the dictionary may be coded directly; it may be coded as a refinement of a symbol already in a dictionary; or it may be coded as an aggregation of two or more symbols already in dictionaries. This procedure is invoked during the decoding of a symbol dictionary segment.

The **halftone region decoding procedure** takes halftone symbols from a halftone dictionary and places them in a buffer. This procedure is invoked during the decoding of a halftone region segment. The halftone region segment contains the position information for all the halftone symbols to be placed in the buffer, as well as index information for the halftone symbols themselves. The halftone patterns, the fixed-size bitmaps of the halftone symbols, are taken from the halftone dictionaries.

The **halftone dictionary decoding procedure** creates a dictionary, that is, an indexed set of fixed-size bitmaps (halftone patterns). The bitmaps in the dictionary are coded directly and jointly. This procedure is invoked during

Table 1 — Entities in the decoding process

Concept	JBIG2 bitstream entity	JBIG2 decoding entity	Physical representation
Document	JBIG2 file	JBIG2 decoder	Output medium or device
Page	Collection of segments	Implicit in control decoding procedure	Page buffer
Region	Region segment	Region decoding procedure	Page buffer or auxiliary buffer
Dictionary	Dictionary segment	Dictionary decoding procedure	List of symbols
Character	Field within a s. dictionary segment	S. dictionary decoding procedure	Symbol bitmap
Gray-scale value	Field within a h. dictionary segment	H. dictionary decoding procedure	Halftone pattern

the decoding of a halftone dictionary segment.

The **control decoding procedure** decodes segment headers, which include segment type information. The segment type determines which decoder must be invoked to decode the segment. The segment type also determines where the decoded output from the segment will be placed. The segment reference information, also present in the segment header and decoded by the control decoding procedure, determines which other segments must be used to decode the current segment.

0.2 Lossy Coding

This specification does not define how to control lossy coding of bi-level images. Rather it defines how to perform perfect reconstruction of a bitmap that the encoder has chosen to encode. If the encoder chooses to encode a bitmap that is different than the original, the entire process becomes one of lossy coding. The different coding methods allow for different methods of introducing loss in a profitable way.

0.2.1 Symbol Coding

Symbol coding provides a natural way of doing lossy coding of text regions. The idea is to allow small differences between the original symbol bitmap and the one indexed in the symbol dictionary. Compression gain is effected by not having to code a large dictionary and, afterwards, by having a cheap symbol index coding as a consequence of the smaller dictionary. It is up to the encoder to decide when two bitmaps are essentially the same or essentially different. This technique was first described in [1].

The hazard of lossy symbol coding is to have *substitution errors*, that is, to have the encoder replace a bitmap corresponding to one character by a bitmap depicting a different character, so that a human reader misreads the character. The risk of substitution errors can be reduced by using intricate measures of difference between bitmaps and/or by making sure that the critical pixels of the indexed bitmap are correct. One way to control this, described in [6], is to index the possibly wrong symbol and then to apply refinement coding to that symbol bitmap. The idea is to correct pixels which the encoder can see are sometimes black and sometimes white.

The process of beneficially introducing loss in textual regions may also take simpler forms such as removing flyspecks from documents or regularizing edges of letters. Most likely such changes will lower the code length of the region without affecting the general appearance of the region — possibly even improving the appearance.

0.2.2 Generic Coding

To effect near-lossless coding using generic coding, the encoder applies a preprocess to an original image and encodes the changed image losslessly. The difficulty is to ensure that the changes result in a lower the code length and that the quality of the changed image is not suffering badly from the changes. Two possible preprocesses are given in [11]. These preprocesses flip pixels that, when flipped, significantly lower the total code length of the region, but can be flipped without seriously impairing the visual quality. The preprocesses provide for effective near-lossless coding of periodic halftones and for a moderate gain in compression for other data types. The preprocesses are not well-suited for error diffused images and images dithered with blue noise as perceptually lossless compression will not be achieved at a significantly lower rate than the lossless rate.

0.2.3 Halftone coding

Halftone coding is the natural way to obtain very high compression for halftoned images. In contrast to lossy generic coding as described above, halftone coding does not intend to preserve the original bitmap, although this is possible in special cases.

The compression gain is effected by not putting all the halftone patterns of the original image into the dictionary or by not always indexing the pattern during index coding even if it does appear in the dictionary.

For lossy coding of error diffused images and images dithered with blue noise it is advisable to use halftone coding with a small grid size. A reconstructed image will lack fine details and may display blockiness but will be clearly recognizable. The blockiness may be reduced on the decoder side in a postprocess; for instance, by using other reconstruction patterns than those that appear in the dictionary.

0.2.4 Consequences of Inadequate Segmentation

Using lossy symbol coding for a document containing both symbol and halftone data will result in poor compression. Depending on the encoder, the quality of the halftone data may be good or bad. Using the form of lossy symbol coding described in [6] the visual quality will probably not suffer.

Using lossy generic coding (using the preprocesses given in [11]) for a document containing both symbol and halftone data usually results in good quality and moderate compression.

Line art and regions of non-typed text may be coded efficiently using generic coding, but depending on the encoder, these types of regions can also be very effectively coded with symbol coding.

1 Scope

This International Standard defines methods for coding bi-level images and sets of images (documents consisting of multiple pages). It is particularly suitable for bi-level images consisting of text and dithered (halftone) data.

The methods defined permit lossless (bit-preserving) coding, lossy coding, and progressive coding. In progressive coding, the first image is lossy; subsequent images may be lossy or lossless.

This International Standard also defines file formats to enclose the coded bi-level image data.

2 Normative References

The following ITU-T Recommendations and International Standards contain provisions which, through references in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The ITU-T Telecommunication Standardization Bureau (TSB) maintains a list of the currently valid ITU-T Recommendations.

- ISO/IEC 8859-1:1987 to ISO 8859-10:1992, Information processing — 8-bit single byte coded graphic character sets
- ISO/IEC 10646-1:1993, Information technology — Universal multiple-octet coded character set (UCS) — Architecture and basic multilingual plane
- ITU-T T.6 (1988), Facsimile coding schemes and coding control functions for group 4 facsimilie apparatus — Terminal Equipment and Protocols for Telematic Services (Study Group XIII)

3 Terms and Definitions

For the puposes of this International Standard, the terms and definitions given in the following apply.

3.1
Adaptive template pixel(s)
A special pixel(s), in a template, whose location is not fixed

3.2
Aggregation
A joining or merging of several individual symbols into a new symbol

3.3
Bit
A binary digit, representing the value **0** or **1**

3.4
Bitmap
A rectangular array of bits

3.5
Buffer
A storage area used to hold a bitmap

3.6
Byte
Eight bits of data

3.7
Combination operator
An operator used to combine the prior contents of a bitmap with new values being drawn into that bitmap

3.8
Coordinate system
A numbering system for two-dimensional locations where locations are labelled by two numbers, the first one increasing from left to right and the second one increasing from top to bottom.

3.9
Delta S
The difference in S coordinate between two successive symbol instances in a non-empty strip

3.10
Delta T
The difference in T coordinate between two successive non-empty strips

3.11
Decoding procedure
A component of a decoder that decodes a certain type of data

3.12
Decoding procedures

3.12.1
Integer decoding procedure
A decoding procedure whose output is a single value

3.12.2
Arithmetic integer decoding procedure
An integer decoding procedure that uses arithmetic entropy decoding

3.12.3
Region decoding procedure
A decoding procedure whose output is a bitmap

3.12.4
Generic region decoding procedure
A region decoding procedure that operates by decoding pixels individually or in runs

3.12.5
Generic refinement region decoding procedure
A region decoding procedure that operates by modifying a reference bitmap to produce an output bitmap

3.12.6
Halftone dictionary decoding procedure
A decoding procedure whose output is a list of halftone patterns

3.12.7
Halftone region decoding procedure
A region decoding procedure that operates by drawing a set of halftone patterns into a bitmap, placing the patterns according to a halftone grid

3.12.8
Symbol region decoding procedure
A region decoding procedure that operates by drawing a set of symbol instances into a bitmap

3.12.9
Symbol dictionary decoding procedure
A decoding procedure whose output is a list of symbols

3.13
Decoder
An entity capable of decoding a bitstream in conformance with this International Standard

3.14
Dictionaries

3.14.1
Halftone dictionary
A list of halftone patterns

3.14.2
Symbol dictionary
A list of symbols

3.15
Export flag
A bit indicating that a symbol is on the export list of a symbol dictionary

3.16
Export list
A list of the symbols in a symbol dictionary that may be used by referring to that symbol dictionary

3.17
Grayscale image
A rectangular array of non-negative integer indices

3.18
Grayscale pixel
An integer-valued element in a grayscale image

3.19
Halftone grid
A rectilinear grid of locations specifying where halftone patterns are to be drawn

3.20
Halftone pattern
A bitmap produced by a halftone dictionary decoding procedure

3.21
Height class
A set of symbols in a symbol dictionary whose heights are all equal

3.22
Height class delta height
The difference in height between two height classes

3.23
Height class delta width
The difference in width between two symbols in a height class

3.24
Ordinal
A value used as a counter

3.25
Out-of-band value
A non-numeric value that may be produced in place of an integer

3.26
Pixel
An element with **0** or **1** as its value in a bitmap

3.27
Prefix length
The length of a Huffman code followed by a fixed number of bits, together representing an integer

3.28
Range length
The length of the fixed number of bits following a Huffman prefix code

3.29
Reference bitmap
The bitmap used as the reference plane during the refinement region decoding procedure

3.30
Referred-to segment
Another segment required in order to decode the current segment

3.31
Region
A bitmap produced by a region decoding procedure

3.32
Segment
A segment header and its segment data

3.33
Strip
A full-width or full-height portion of the coordinate system of a symbol region

3.34
Types of strips

3.35
Empty strip
A strip containing the reference corners of no symbol instances

3.36
Non-empty strip
A strip containing the reference corner of at least one symbol instance

3.37
Strip size
The extent in pixels of the non-full dimension of a strip

3.38
Symbol
A bitmap produced by a symbol dictionary decoding procedure

3.39
Symbol ID
An integer used to identify a symbol, or to index into an array of symbols to retrieve the symbol

3.40
Symbol instance
A symbol drawn, possibly with refinement, at a particular location in a symbol region

3.41
Symbol instance refinement delta height
The difference in height between a symbol instance's reference bitmap and the bitmap produced by the generic refinement region decoding procedure

3.42
Symbol instance refinement delta width
The difference in width between a symbol instance's reference bitmap and the bitmap produced by the generic refinement region decoding procedure

3.43
Symbol instance refinement delta X
The difference between the X coordinates of the top left corners of a symbol instance's reference bitmap and the bitmap produced by the generic refinement region decoding procedure

3.44
Symbol instance refinement delta Y
The difference between the Y coordinates of the top left corners of a symbol instance's reference bitmap and the bitmap produced by the generic refinement region decoding procedure

3.45
Typical prediction
Typical prediction signals that an entire row of a generic region is identical to the preceding row

4 Symbols and Abbreviations

4.1 Abbreviations

The abbreviations used in this International Standard are listed below.

AT	Adaptive template
ID	Identifier
LPS	Less probable symbol, i.e., less probable binary value
MMR	Modified modified READ
MPS	More probable symbol, i.e., more probable binary value
OOB	Out-of-band
READ	Relative Element Address Designate
TP	Typical prediction
TPR	Typical prediction for refinement

NOTE — The "symbol" in the abbreviations LPS and MPS does not refer to the symbols (bitmaps) in the International Standard. The LPS and MPS abbreviations are used despite this because they are the generally-accepted terminology in arithmetic coding.

4.2 Symbol definitions

The following symbols used in this International Standard are listed below. A convention is used that parameters to any of the decoding procedures that are used in this International Standard are indicated in **emboldened** letters.

A	Probability interval
ARRAY	An array
A_1, A_2, A_3, A_4	Adaptive template pixels in the generic region decoding procedure
B	Current byte of arithmetically-coded data
B1	Byte of arithmetically-coded data following the current byte
B_S	A symbol bitmap in a symbol dictionary decoding procedure
BM	A bitmap
BP	Pointer to byte contained in B
BPST	Initial value of BPST
C	Value of bit stream in code register
Chigh	High-order 16 bits of C
Clow	Low-order 16 bits of C
CT	Renormalisation shift counter
CURCODE	The Huffman code for the current table line in a Huffman table
CURLEN	The current table line prefix length in a Huffman table
CURRANGELOW	A variable holding the lower bound of the current table line in a Huffman table
CURS	The current S coordinate in a symbol region decoding procedure
CURT	The current symbol instance's T coordinate relative to the current strip's T coordinate in a symbol region decoding procedure
D	Decision decoded
DFS	The difference in S coordinates between the first instances of two strips
DT	The number of empty strips between two non-empty strips
DW	The difference of width between two symbol bitmaps in a symbol dictionary decoding procedure
EXFLAG	An array of export flags

EXINDEX	An index for the array EXFLAG
EXRUNLENGTH	The length of a run of identical export flag values
FIRSTS	The first S coordinate of the current strip
FIRSTCODE	The first code assigned to a particular prefix length in a Huffman table
GBATX$_1$	The X location of adaptive template pixel 1 in a generic region decoding procedure
GBATY$_1$	The Y location of adaptive template pixel 1 in a generic region decoding procedure
GBATX$_2$	The X location of adaptive template pixel 2 in a generic region decoding procedure
GBATY$_2$	The Y location of adaptive template pixel 2 in a generic region decoding procedure
GBATX$_3$	The X location of adaptive template pixel 3 in a generic region decoding procedure
GBATY$_3$	The Y location of adaptive template pixel 3 in a generic region decoding procedure
GBATX$_4$	The X location of adaptive template pixel 4 in a generic region decoding procedure
GBATY$_4$	The Y location of adaptive template pixel 4 in a generic region decoding procedure
GBH	The height of a generic region
GBREG	The region produced by a generic region decoding procedure
GBTEMPLATE	A parameter indicating the number and arrangement of the pixels in a template used in a generic region decoding procedure
GBW	The width of a generic region
GRATX$_1$	The X location of adaptive template pixel 1 in a generic refinement region decoding procedure
GRATY$_1$	The Y location of adaptive template pixel 1 in a generic refinement region decoding procedure
GRATX$_2$	The X location of adaptive template pixel 2 in a generic refinement region decoding procedure
GRATY$_2$	The Y location of adaptive template pixel 2 in a generic refinement region decoding procedure
GRH	The height of a generic region being coded with refinement coding
GRREFERENCE	The reference bitmap in a generic refinement region decoding procedure
GRREFERENCEDX	The X offset of the reference bitmap with respect to the bitmap being decoded in a generic refinement region decoding procedure
GRREFERENCEDY	The Y offset of the reference bitmap with respect to the bitmap being decoded in a generic refinement region decoding procedure
GRREG	The region produced by a generic refinement region decoding procedure
GRTEMPLATE	A parameter indicating the number and arrangement of the pixels in a template used in decoding a generic region with refinement coding
GRW	The width of a generic region being coded with refinement coding
HCHEIGHT	The height of the current height class in a symbol dictionary decoding procedure
HCDH	The difference in height between two height classes in a symbol dictionary decoding procedure
HCFIRSTSYM	The index of the first symbol decoded in a height class
H_I	The height of a symbol instance bitmap
HIGHPREFLEN	The prefix length of the upper range table line in a Huffman table
HO_I	The height of the original bitmap of a symbol instance containing refinement information
HTHIGH	One greater than the largest value that is represented by any normal table line in a Huffman table

HTLOW	The lowest value that is represented by any normal table line in a Huffman table
HTOFFSET	The range offset of a table line when decoding using a Huffman table
HTOOB	Whether a Huffman table can produce the out-of-band value OOB
HTPS	The length of the encoded prefix field in a table line in a Huffman table
HTRS	The length of the encoded range field in a table line in a Huffman table
HTVAL	The value decoded using a Huffman table
I	An array index
IAAI	An arithmetic integer decoding procedure used to decode the number of symbol instances in an aggregation
IADH	An arithmetic integer decoding procedure used to decode the difference in height between two height classes
IADS	An arithmetic integer decoding procedure used to decode the S coordinate of the second and subsequent symbol instances in a strip
IADT	An arithmetic integer decoding procedure used to decode the T coordinate of the second and subsequent symbol instances in a strip
IADW	An arithmetic integer decoding procedure used to decode the difference in width between two symbols in a height class
IAEX	An arithmetic integer decoding procedure used to decode export flags
IAFS	An arithmetic integer decoding procedure used to decode the S coordinate of the first symbol instance in a strip
IAID	An arithmetic integer decoding procedure used to decode the symbol IDs of symbol instances
IARDH	An arithmetic integer decoding procedure used to decode the delta height of symbol instance refinements
IARDW	An arithmetic integer decoding procedure used to decode the delta width of symbol instance refinements
IARDX	An arithmetic integer decoding procedure used to decode the delta X position of symbol instance refinements
IARDY	An arithmetic integer decoding procedure used to decode the delta Y position of symbol instance refinements
IARI	An arithmetic integer decoding procedure used to decode the R_I bit of symbol instances
IAIT	An arithmetic integer decoding procedure used to decode the T coordinate of the symbol instances in a strip
IB_I	The bitmap of a symbol instance
IBO_I	The original bitmap of a symbol instance containing refinement information
ID_I	The symbol ID of a symbol instance
IDS	The delta S value for a symbol instance in a symbol region decoding procedure
J	An array index
K	The ordinal for a referred-to segment
LENCOUNT	A histogram of the prefix lengths in a Huffman table
LENMAX	The largest prefix length in a Huffman table
LNTP	Whether the current line is coded explicitly in a generic region decoder
LOGSBSTRIPS	The base-2 logarithm of the strip size used to encode a symbol region
LOWPREFLEN	The prefix length of the lower range table line in a Huffman table

MMR	Whether MMR coding is used in a generic region decoding procedure
NINSTANCES	A symbol instance counter
NSYMSDECODED	The number of symbols decoded so far in a symbol dictionary decoding procedure
NTEMP	The number of table lines in a Huffman table
OOB	An out-of-band value
P	The page with which a segment is associated
PREFLEN	An array of prefix lengths representing the table lines in a Huffman table
r	A segment retention flag
R	The number of segments referred to by some segment
RANGELEN	An array of range lengths representing the table lines in a Huffman table
RANGELOW	An array holding the lower bounds of the table lines in a Huffman table
RA_1, RA_2	Adaptive template pixels in the generic refinement region decoding procedure
RDH_I	The delta height of a symbol instance refinement bitmap
RDW_I	The delta width of a symbol instance refinement bitmap
RDX_I	The X offset of a symbol instance refinement
RDY_I	The Y offset of a symbol instance refinement
REFAGGNINST	The number of symbol instances in an aggregation
R_I	A bit indicating whether refinement information is present for a symbol instance
REFCORNER	Which corner of a symbol instance bitmap is to be used as a reference in a symbol region decoding procedure
S	One coordinate of the coordinate system used in a symbol region decoding procedure
S_I	The S coordinate of a symbol instance
SBDSOFFSET	An offset for the coded delta S values in a symbol region
SBCOMBOP	The combination operator used in a symbol region decoding procedure
SBDEFPIXEL	The default pixel of a symbol region
SBH	The height of a symbol region
SBHUFF	Whether Huffman coding is used in a symbol region decoding procedure
SBHUFFDS	The Huffman table used to decode the S coordinate of subsequent symbol instances in a strip
SBHUFFDT	The Huffman table used to decode the difference in T coordinates between non-empty strips
SBHUFFFS	The Huffman table used to decode the S coordinate of the first symbol instance in a strip
SBHUFFRDH	The Huffman table used to decode the difference between a symbol's height and the height of a refinement coded symbol instance bitmap
SBHUFFRDW	The Huffman table used to decode the difference between a symbol's width and the width of a refinement coded symbol instance bitmap
SBHUFFRDX	The Huffman table used to decode the difference between a symbol instance's X coordinate and the X coordinate of a refinement coded bitmap
SBHUFFRDY	The Huffman table used to decode the difference between a symbol instance's Y coordinate and the Y coordinate of a refinement coded symbol instance bitmap
SBHUFFRSIZE	The Huffman table used to decode the size of a symbol instance's refinement bitmap data
SBNUMINSTANCES	The number of symbol instances in a symbol region
SBNUMSYMS	The number of symbols that may be used in a symbol region

SBRATX$_1$	The X position of the adaptive template pixel RA$_1$ in a symbol region decoding procedure
SBRATY$_1$	The Y position of the adaptive template pixel RA$_1$ in a symbol region decoding procedure
SBRATX$_2$	The X position of the adaptive template pixel RA$_2$ in a symbol region decoding procedure
SBRATY$_2$	The Y position of the adaptive template pixel RA$_2$ in a symbol region decoding procedure
SBREFINE	Whether refinement coding is used in a symbol region decoding procedure
SBRTEMPLATE	Template identifier for refinement coding of bitmap in a symbol region decoding procedure
SBSTRIPS	The height of the symbol instance strips
SBSYMCODES	An array of variable-length codes identifying individual symbols
SBSYMS	An array of symbols used in a symbol region
SBW	The width of a symbol region
SDATX$_1$	The X position of the adaptive template pixel A$_1$ in a symbol dictionary decoding procedure
SDATY$_1$	The Y position of the adaptive template pixel A$_1$ in a symbol dictionary decoding procedure
SDATX$_2$	The X position of the adaptive template pixel A$_2$ in a symbol dictionary decoding procedure
SDATY$_2$	The Y position of the adaptive template pixel A$_2$ in a symbol dictionary decoding procedure
SDATX$_3$	The X position of the adaptive template pixel A$_3$ in a symbol dictionary decoding procedure
SDATY$_3$	The Y position of the adaptive template pixel A$_3$ in a symbol dictionary decoding procedure
SDATX$_4$	The X position of the adaptive template pixel A$_4$ in a symbol dictionary decoding procedure
SDATY$_4$	The Y position of the adaptive template pixel A$_4$ in a symbol dictionary decoding procedure
SDEXSYMS	The symbols exported from a symbol dictionary
SDHUFF	Whether Huffman coding is used in a symbol dictionary decoding procedure
SDHUFFAGGINST	The Huffman table used to decode the number of symbol instances in an aggregation in a symbol dictionary decoding procedure
SDHUFFDH	The Huffman table used to decode the difference in height between two height classes in a symbol dictionary decoding procedure
SDHUFFDW	The Huffman table used to decode the difference in width between two symbols in a symbol dictionary decoding procedure
SDHUFFBMSIZE	The Huffman table used to decode the size of a height class collective bitmap in a symbol dictionary decoding procedure
SDINSYMS	An array of symbols used as a parameter to a symbol dictionary decoding procedure
SDNEWSYMS	The symbols decoded in a symbol dictionary
SDNEWSYMWIDTHS	The widths of the symbols decoded in a symbol dictionary
SDNUMEXSYMS	The number of symbols exported from a symbol dictionary
SDNUMINSYMS	The number of symbols in the array that is used as a parameter to a symbol dictionary decoding procedure
SDNUMNEWSYMS	The number of symbols generated in a symbol dictionary

SDREFAGG	Whether refinement and aggregate coding are used in a symbol dictionary decoding procedure
$\mathbf{SDRATX_1}$	The X position of the adaptive template pixel RA_1 in a symbol dictionary decoding procedure
$\mathbf{SDRATY_1}$	The Y position of the adaptive template pixel RA_1 in a symbol dictionary decoding procedure
$\mathbf{SDRATX_2}$	The X position of the adaptive template pixel RA_2 in a symbol dictionary decoding procedure
$\mathbf{SDRATY_2}$	The Y position of the adaptive template pixel RA_2 in a symbol dictionary decoding procedure
SDRTEMPLATE	Template identifier for refinement coding of bitmaps in a symbol dictionary decoding procedure
SDTEMPLATE	The template identifier used to decode symbol bitmaps in a symbol dictionary decoding procedure
SKIP	A mask of pixels to be skipped during the decoding of a generic region
SLNTP	A binary value indicating whether the current line is typical
STRIPT	The numerically smallest T coordinate in the current strip
SYMWIDTH	The current bitmap width in a symbol dictionary decoding procedure.
T	One coordinate of the coordinate system used in a symbol region decoding procedure
TEMPC	A temporary register in the MQ coder
T_I	The T coordinate of a symbol instance
TOTWIDTH	The total width of the bitmaps in a height class
TPON	Whether typical prediction is used in a generic region decoding procedure
TPRON	Whether typical prediction is used in a generic region decoding procedure with refinement coding
TRANSPOSED	Whether the symbol instance coordinates are transposed in a symbol region decoding procedure
USESKIP	Whether some pixels should be skipped in the decoding of a generic region
V1	A binary value
V2	A binary value
W_I	The width of a symbol instance bitmap
WO_I	The width of the original bitmap of a symbol instance containing refinement information
x	A real number
X	The horizontal coordinate of a pixel in a bitmap
Y	The vertical coordinate of a pixel in a bitmap

4.3 Operator definitions

The following operators are defined

OR	If V1 and V2 are two binary values, then V1 OR V2 is equal to **0** if both V1 and V2 are **0**. It is equal to **1** if either of V1 or V2 is **1**. If V1 and V2 are two integer values, then it is the result of bitwise application of OR.
AND	If V1 and V2 are two binary values, then V1 AND V2 is equal to **0** if either of V1 or V2 is **0**. It is equal to **1** if both V1 and V2 are **1**. If V1 and V2 are two integer values, then it is the result of bitwise application of AND.

XOR	If V1 and V2 are two binary values, then V1 XOR V2 is equal to **0** if V1 and V2 are equal. It is equal to **1** if V1 and V2 differ. If V1 and V2 are two integer values, then it is the result of bitwise application of XOR.
XNOR	If V1 and V2 are two binary values, then V1 XNOR V2 is equal to **0** if V1 and V2 differ. It is equal to **1** if V1 and V2 are equal.
REPLACE	If V1 and V2 are two binary values, then V1 REPLACE V2 is equal to V2.
NOT	If V1 is a binary value, then NOT V1 is **1** if V1 is **0**, and is **0** if V1 is **1**.
$\min$	If x and y are numbers then $\min(x, y)$ is the smaller of x and y.
$\max$	If x and y are numbers then $\max(x, y)$ is the larger of x and y.
$\lfloor\rfloor$	If x is a number then $\lfloor x \rfloor$ is the largest integer less than or equal to x.
$\lceil\rceil$	If x is a number then $\lceil x \rceil$ is the smallest integer greater than or equal to x.
$<<$	If V1 and V2 are two integers, then V1 $<<$ V2 is the value obtained by shifting the value of V1 leftwards by V2 bits, filling the rightmost V2 bits of the new value with **0**.
$>>$	If V1 and V2 are two integers, then V1 $>>$ V2 is the value obtained by shifting the value of V1 rightward by V2 bits, filling the leftmost V2 bits of the new value with **0**.
$>>_A$	If V1 and V2 are two integers, then V1 $>>_A$ V2 is the value obtained by shifting the value of V1 rightward by V2 bits, filling the leftmost V2 bits of the new value with **0** if V1 is non-negative and **1** if V1 is negative.

5 Conventions

5.1 Typographic conventions

All field names are given in sans serif.

All parameter names are given in **bold face**.

5.2 Binary notation

The two binary values are denoted as **0** and **1**.

5.3 Hexadecimal notation

The prefix `0x` indicates that the following value is to be interpreted as a hexadecimal number (radix 16).

EXAMPLE — The value `0x6a` is equal to the decimal value 106.

5.4 Integer value syntax

5.4.1 Bit packing

Bits are packed into bytes starting at the most significant bit. If a decoder is reading a sequence of bits out of a bitstream, it shall first read the most significant bit of the first byte, then the next most significant bit, and so on, then proceed to the next byte.

EXAMPLE — The sequence of bytes `0x2f 0x05 0xc1`, if interpreted as a sequence of bits, is the sequence

0 0 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 1 0 0 0 0 0 1

5.4.2 Multi-byte values

All multi-byte values shall be interpreted in a most-significant-first manner: the first byte of each value is the most significant, and the last byte is the least significant.

EXAMPLE — The sequence of bytes `0x01 0x5c 0x99 0xfa`, if interpreted as a four-byte value, represents the value `0x015c99fa`.

5.4.3 Bit numbering

The least significant bit of any value is numbered bit 0. For a one-byte value, the most significant bit is numbered bit 7; for a two-byte value, the most significant bit is numbered bit 15; for a four-byte value, the most significant bit is numbered bit 31.

5.4.4 Signedness

Unless otherwise specified, all multi-bit values shall be treated as unsigned values. When a value is to be treated as a signed number it shall be interpreted in two's-complement form.

5.5 Array notation and conventions

Arrays are numbered starting from zero.

EXAMPLE — A one-dimensional array ARR containing twelve elements has elements

$$\mathrm{ARR}[0], \mathrm{ARR}[1], \ldots, \mathrm{ARR}[11]$$

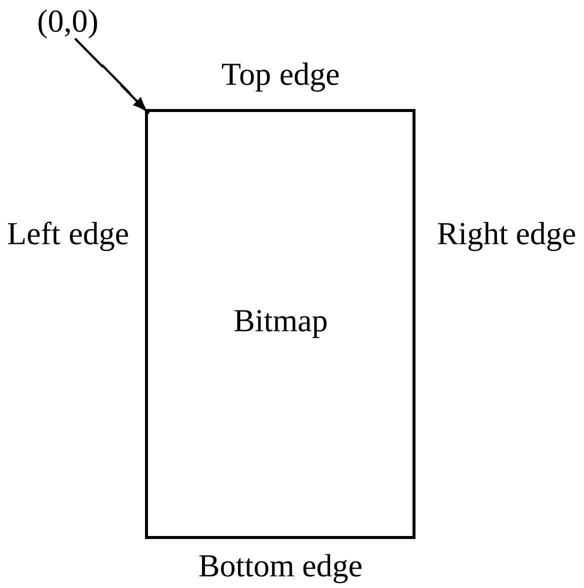


Figure 2 — The four edges of a bitmap

5.6 Bitmaps

A bitmap is a rectangular array. Every location in this array has the value **0** or **1**. A location in a bitmap is referred to as a pixel.

The terms "left", "right", "top", "bottom", "width" and "height" are often applied to bitmaps. These terms do not refer to any physical aspect of the bitmap: if a bitmap is printed on paper, it may be printed with its "left" edge along any edge of the paper. They are used within this standard to refer to the four edges of the bitmap as shown in Figure 2.

A pixel in a bitmap is referred to by a pair of coordinates X and Y, sometimes written as a pair (X, Y). The location $(0, 0)$ represents the pixel in the top left corner. The X coordinate increases rightwards and the Y coordinate increases downwards.

If BM is a bitmap then the pixel whose coordinates are X and Y is referred to as $BM[\mathrm{X}, \mathrm{Y}]$.

NOTE — These conventions are intended to make it easier to describe operations involving bitmaps, and are not intended to imply any physical characteristics of the image represented by the bitmap.

6 Decoding Procedures

6.1 Introduction to Decoding Procedures

This International Standard makes use of a number of different decoding procedures for different types of data. Each of these decoding procedures produces a certain kind of data as output. The generic region decoding procedure, generic refinement region decoding procedure, halftone region decoding procedure, and symbol region decoding procedures all produce regions as their output. The symbol dictionary decoding procedure produces an array of symbols as its output. The halftone dictionary decoding procedure similarly produces an array of halftone cell bitmaps as its output.

The various region decoding procedures operate in different manners:

- The generic region decoding procedure decodes a bitmap, treating it simply as an an array of binary pixels.
- The generic region refinement decoding procedure decodes a bitmap by treating it as an array of binary pixels, but coding each pixel with respect to some reference bitmap.
- The symbol region decoding procedure decodes a bitmap by drawing a collection of symbols into it, possibly applying the generic refinement region decoding procedure to each one.
- The halftone region decoding procedure decodes a bitmap by placing a collection of halftone patterns into it, at locations specified by a halftone grid.

Each decoding procedure is specified in terms of a number of parameters and a sequence of operations, which are affected by the values of the parameters. These parameters are supplied to the decoding procedure for each invocation, and the same decoding procedure may be invoked multiple times during the course of decoding a bitstream, with different parameters each time.

Some of the decoding procedure parameters are unused in certain circumstances, usually depending on the values of other parameters. In these circumstances, no value needs to be specified for those unused parameters.

6.2 Generic Region Decoding Procedure

6.2.1 General Description

This decoding procedure is used to decode a rectangular array of **0** or **1** values, which are coded one pixel at a time (i.e., it is used to decode a bitmap using simple, generic, coding). The decoding procedure also modifies an array of probability information which may be used by other invocations of this generic region decoding procedure.

The generic region decoding procedure may be based on sequential coding of the image pixels using arithmetic coding as specified in Annex E and a template to determine the coding state. This technique was used in ITU-T Recommendation T.82 | ISO/IEC International Standard 11544 (JBIG). This type of decoding is described in 6.2.5.

Alternatively, for improved speed but reduced compression the generic region decoding procedure may be based on Huffman coding of runs of pixels. This technique was used in the MMR (Modified Modified READ) algorithm described in ITU-T Recommendation T.6 (G4). This type of decoding is described in 6.2.6.

6.2.2 Input parameters

The parameters to this decoding procedure are shown in Table 2.

6.2.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table 3.

6.2.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 4.

6.2.5 Decoding using a Template and Arithmetic Coding

6.2.5.1 General Description

If **MMR** is **0** the generic region decoding procedure is based on arithmetic coding with a template to determine the coding state. The remainder of 6.2.5 describes this form of decoding, and only applies when **MMR** is **0**.

Table 2 — Parameters for the generic region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
MMR	Integer	1	N	Whether MMR coding is used.
GBW	Integer	32	N	The width of the region.
GBH	Integer	32	N	The height of the region.
GBTEMPLATE	Integer	2	N	The template identifier. *
TPON	Integer	1	N	Whether typical prediction is used. *
USESKIP	Integer	1	N	Whether some pixels should be skipped in the decoding. *
SKIP	Bitmap			A bitmap indicating which pixels should be skipped. **GBW** pixels wide, **GBH** pixels high. ***
GBATX$_1$	Integer	8	Y	The X position of the adaptive template pixel A_1. *
GBATY$_1$	Integer	8	Y	The Y position of the adaptive template pixel A_1. *
GBATX$_2$	Integer	8	Y	The X position of the adaptive template pixel A_2. **
GBATY$_2$	Integer	8	Y	The Y position of the adaptive template pixel A_2. **
GBATX$_3$	Integer	8	Y	The X position of the adaptive template pixel A_3. **
GBATY$_3$	Integer	8	Y	The Y position of the adaptive template pixel A_3. **
GBATX$_4$	Integer	8	Y	The X position of the adaptive template pixel A_4. **
GBATY$_4$	Integer	8	Y	The Y position of the adaptive template pixel A_4. **

* Unused if **MMR** $=$ **1**
** Unused if **MMR** $=$ **1** or **GBTEMPLATE** $\neq 0$
*** Unused if **USESKIP** $=$ **0** or **MMR** $=$ **1**

Table 3 — Return values from the generic region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GBREG	Bitmap			The decoded region bitmap.

Table 4 — Variables used in the generic region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
LNTP	Integer	1	N	Whether the current image line is coded explicitly *
SLNTP	Integer	1	N	Whether the current line's LNTP value is different from the previous line's LNTP value *
CONTEXT	Integer	16	N	The values of the pixels in the template *

* Unused if **MMR** $=$ **1**

6.2.5.2 Coding Order and Edge Conventions

The coding algorithm iterates through the bitmap in raster scan order, that is, by rows from top to bottom, and within each row from left to right. The processing for a current target pixel will reference the colors of some pixels in fixed spatial relationship to the target pixel.

Near the edges of the bitmap, these neighbor references may not lie in the actual bitmap. The rules to satisfy out-of-bounds references shall be as follows:

- All pixels lying outside the bounds of the actual bitmap have the value **0**.

6.2.5.3 Fixed Templates

A template defines a neighborhood around a pixel to be coded. The values of the pixels in this neighborhood define a context. Each context has its own adaptive probability estimate used by the arithmetic coder (see Annex E). Although a template is a geometric pattern of pixels, the pixels in a template are said to take on values when the template is aligned with a particular part of the image.

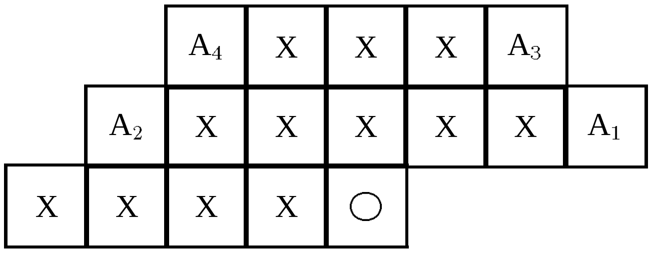


Figure 3 — Template when GBTEMPLATE=0, showing the AT pixels at their nominal locations.

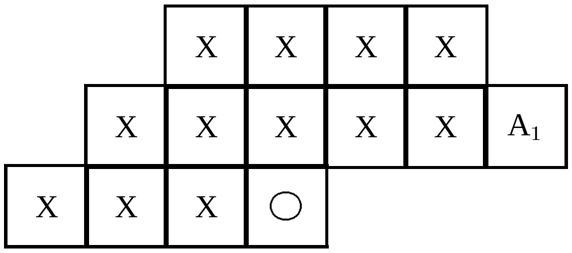


Figure 4 — Template when GBTEMPLATE=1, showing the AT pixel at its nominal location.

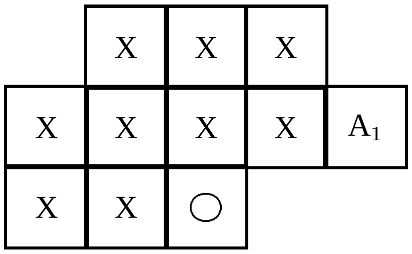


Figure 5 — Template when GBTEMPLATE=2, showing the AT pixel at its nominal location.

Figure 3 shows the template which shall be used when **GBTEMPLATE** is 0. Figure 4 shows the template which shall be used when **GBTEMPLATE** is 1. Figure 5 shows the template which shall be used when **GBTEMPLATE** is 2. Figure 6 shows the template which shall be used when **GBTEMPLATE** is 3. In each of these figures, the pixel denoted by a circle corresponds to the pixel to be coded and is not part of the template. The pixels denoted by 'X' correspond to ordinary pixels in the template. The pixels denoted A_1–A_4 are special pixels in the template.

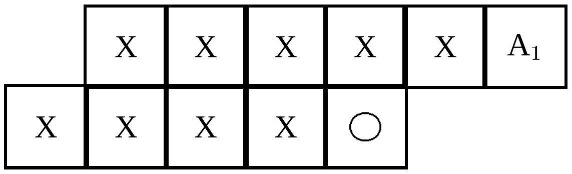


Figure 6 — Template when GBTEMPLATE=3, showing the AT pixel at its nominal location.

They are denoted “adaptive” or AT pixels. These pixels are special in that their positions are not fixed, but can be placed at different locations. See 6.2.5.4 for a description of AT pixels. The legends A_1–A_4 indicate the AT pixels 1 to 4. The pixels’ actual positions are specified as parameters to this decoding procedure; Figures 3– 6 show the nominal locations of these AT pixels for each template.

The values of the pixels in the template shall be combined to form a context. Each pixel in the template (including the adaptive pixels) shall correspond to a specific bit in the context, although the pixels in the template may be assigned to bits in the context in any order. Because there are up to 16 pixels in the template, contexts can take on up to 65536 different values. This context shall be used to identify which adaptive probability estimate must be used by the arithmetic coder for encoding the pixel to be coded (see Annex E).

NOTE 1 — A rule of thumb is to use large templates for large bitmaps. Thus a full-size periodic halftone should be coded with the 16-pixel template and tiny bitmaps such as usual symbol bitmaps should be coded with one of the 10-pixel templates. In some cases an intermediate template is desired, for performance or decoder memory requirements; in this case the 13-pixel template should be used. It is also possible to generate further templates by placing one or more of the AT pixels on top of a regular template pixel, thus fixing its value.

NOTE 2 — The 10-pixel templates are those used in ITU-T Recommendation T.82 | ISO/IEC International Standard 11544 (JBIG). Software execution speed is somewhat higher with the two-line template than any of the three-line templates. For most images the 10-pixel, three-line template gives higher compression than the 10-pixel, two-line template.

6.2.5.4 Adaptive Template Pixels

In coding the image, the template shall be allowed to change in the restricted way described in this clause.

The pixels that are allowed to change shall be called AT pixels. Their nominal positions are indicated by ‘A_1’, ‘A_2’, ‘A_3’, and ‘A_4’ in Figures 3, 4, 5 and 6. Note that some templates have fewer than four AT pixels. In general, an AT pixel can be located anywhere in the field shown in Figure 7, not including the current pixel. Hence, there is the possibility to use an effective template size of 15, 14, 13, 12 or 9 pixels by having the moved position of the AT pixel overlap a regular template pixel. The actual locations of the AT pixels for any invocation of this decoding procedure are specified as parameters to the decoding procedure. The location of the pixel A_1 is given by $(\mathbf{GBATX_1}, \mathbf{GBATY_1})$. If **GBTEMPLATE** is 0 then

- the location of the pixel A_2 is given by $(\mathbf{GBATX_2}, \mathbf{GBATY_2})$,
- the location of the pixel A_3 is given by $(\mathbf{GBATX_3}, \mathbf{GBATY_3})$,
- and the location of the pixel A_4 is given by $(\mathbf{GBATX_4}, \mathbf{GBATY_4})$,

NOTE 1 — Some profiles may restrict AT pixel locations to a smaller range than that shown in Figure 7.

NOTE 2 — The index of the AT pixels in Figure 3 corresponds to the expected goodness. Moving only one AT pixel from its nominal position, it is advisable to move A_4. The next pixel to move is A_3 and so on.

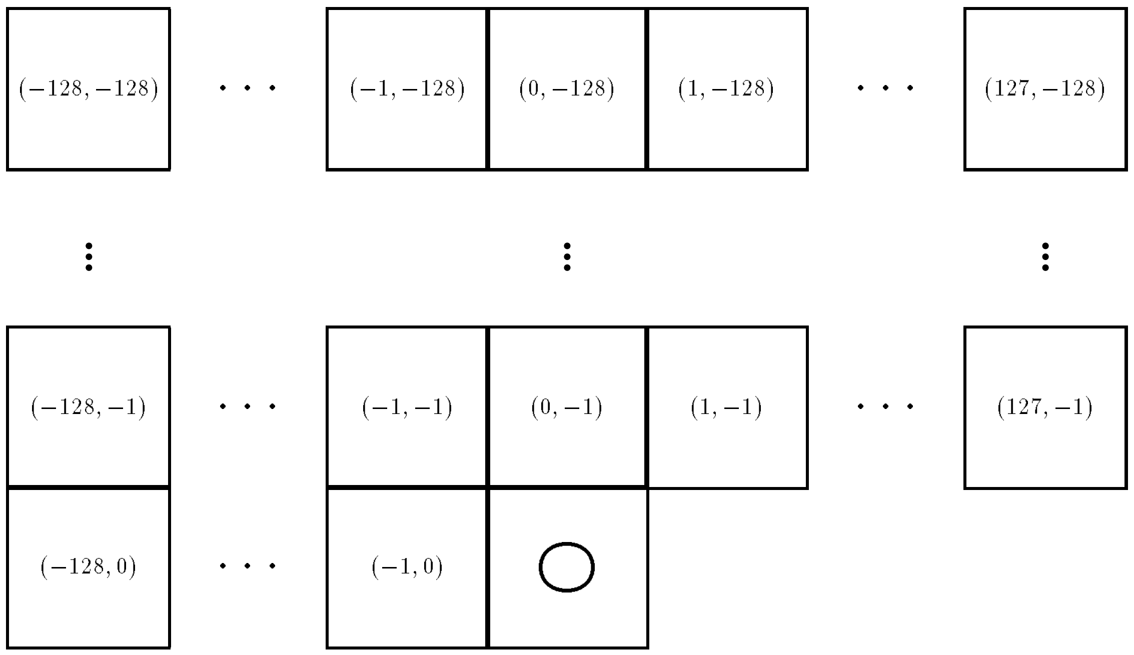


Figure 7 — Field to which AT pixel locations are restricted.

NOTE 3 — The nominal locations of the AT pixels are as shown in Table 5. These locations should be used unless other locations improve compression performance. Some profiles may restrict AT pixel locations to only these nominal locations.

NOTE 4 — If an AT pixel's location overlaps any regular template pixel's location, then the AT pixel's value can be ignored (since it duplicates another value). This can reduce the memory requirements of the decoder, since not all CX values can occur.

6.2.5.5 Typical Prediction (TP)

Typical prediction can be enabled or disabled with the **TPON** parameter. If typical prediction is enabled (**TPON** is **1**), then before the first pixel of each row is decoded, a value indicating that a row is typical shall be decoded. If the row is typical then each pixel of this row is identical to the corresponding pixel in the row immediately above, and so no other pixels of this row need to be decoded. If the row is not typical, then each pixel of this row needs to be decoded.

Table 5 — The nominal values of the AT pixel locations.

NOTE — NA means that the parameter has no nominal value.

GBTEMPLATE	**$GBATX_1$**	**$GBATX_2$**	**$GBATX_3$**	**$GBATX_4$**
	$GBATY_1$	**$GBATY_2$**	**$GBATY_3$**	**$GBATY_4$**
0	3	-3	2	-2
	-1	-1	-2	-2
1	3	NA	NA	NA
	-1	NA	NA	NA
2	2	NA	NA	NA
	-1	NA	NA	NA
3	2	NA	NA	NA
	-1	NA	NA	NA

6.2.5.6 Skipped Pixels

If the parameter **USESKIP** is **1**, then the parameter **SKIP** contains a **GBW**-by-**GBH** bitmap. Each pixel in **SKIP** corresponds to a pixel in the bitmap being decoded; if the pixel in **SKIP** is **1** then the corresponding pixel in the bitmap being decoded is **0**, and does not need to be actually decoded.

6.2.5.7 Decoding the Bitmap

The decoding of the bitmap shall proceed as follows.

1. Set

$$\mathrm{LNTP} = \mathbf{1}$$

2. Create a bitmap GBREG of width **GBW** and height **GBH** pixels.
3. Decode each row as follows.
   (a) If all **GBH** rows have been decoded then the decoding is complete; proceed to step 4.
   (b) If **TPON** is **1** then decode a bit using the arithmetic entropy coder, where the context used to decode this bit varies depending on the template in use:
      - If **GBTEMPLATE** is 0, use the context shown in Figure 8.
      - If **GBTEMPLATE** is 1, use the context shown in Figure 9.
      - If **GBTEMPLATE** is 2, use the context shown in Figure 10.
      - If **GBTEMPLATE** is 3, use the context shown in Figure 11.

      Let SLNTP be the value of this bit. Set

$$\mathrm{LNTP} = \mathrm{LNTP\ XOR\ SLNTP}$$

   (c) If LNTP = **0** then set every pixel of the current row of GBREG equal to the corresponding pixel of the row immediately above.
   (d) If LNTP = **1** then, from left to right, decode each pixel of the current row of GBREG. The procedure for each pixel is as follows:
      i. If **USESKIP** is **1** and the pixel in the bitmap **SKIP** at the location corresponding to the current pixel is **1**, then set the current pixel to **0**.
      ii. Otherwise,
         A. Place the template given by parameters **GBTEMPLATE**, $\mathbf{GBATX_1}$ through $\mathbf{GBATX_4}$ and $\mathbf{GBATY_1}$ through $\mathbf{GBATY_4}$ so that the current pixel is aligned with the location denoted by a circle in the figure describing the appearance of the template with identifier **GBTEMPLATE**.
         B. Form an integer CONTEXT by gathering the values of the image pixels overlaid by the template (including AT pixels) at its current location. The order of this gathering is not standardised, but must be consistent and independent of the location of the AT pixels.
         C. Decode the current pixel by invoking the arithmetic entropy decoding procedure, with CX set to the value formed by concatenating the label "GB" and the 10–16 pixel values gathered in CONTEXT. The result of this invocation is the value of the current pixel.

         EXAMPLE — If **GBTEMPLATE** is 2, the image pixels overlaid by the template are as shown in Figure 10, and the pixels are gathered in reading order (in rows from top to bottom, and within each row from left to right), then CX is set to "GB**0011100101**".
4. After all the rows have been decoded, the current contents of the bitmap GBREG are the results that shall be obtained by every decoder, whether it performs this exact sequence of steps or not.

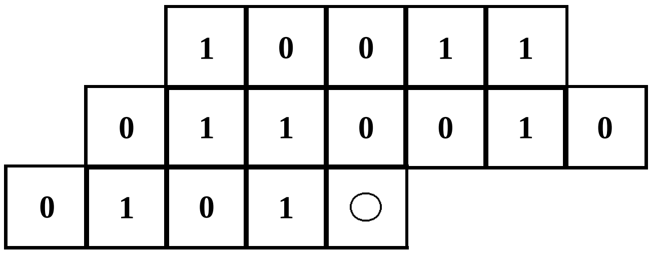


Figure 8 — Reused context for coding the TP pseudo-pixel when GBTEMPLATE is 0.

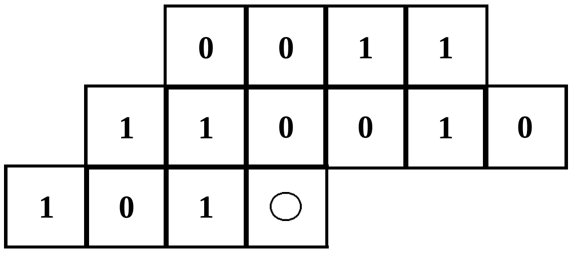


Figure 9 — Reused context for coding the TP pseudo-pixel when GBTEMPLATE is 1.

6.2.6 Decoding using Huffman coding

If **MMR** is **1**, the image bitmap decoding procedure is identical to an MMR (Modified Modified READ) decoder described in ITU-T Recommendation T.6. The decoder in ITU-T Recommendation T.6 is specified as producing pixels whose value may be either "black" or "white". For the purposes of this International Standard, the result of using the MMR decoder shall be interpreted as follows:

- Pixels decoded by the MMR decoder having the value "black" shall be treated as having the value **1**.
- Pixels decoded by the MMR decoder having the value "white" shall be treated as having the value **0**.

NOTE — MMR provides less compression than image bitmap compression based on arithmetic coding. Image bitmap decoding using MMR is faster than image bitmap decoding based on arithmetic coding.

6.3 Generic Refinement Region Decoding Procedure

6.3.1 General Description

This decoding procedure is used to decode a rectangular array of **0** or **1** values, which are coded one pixel at a time. There is a reference bitmap known to the decoding procedure, and this is used as part of the decoding process. The reference bitmap is intended to resemble the bitmap being decoded, and this similarity is used to increase compression. Each pixel is decoded using a context comprising pixels drawn from the reference bitmap as well as previously-decoded pixels from the bitmap being decoded.

6.3.2 Input parameters

The parameters to this decoding procedure are shown in Table 6.

6.3.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table 7.

6.3.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 8.

Table 6 — Parameters for the generic refinement region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GRW	Integer	32	N	The width of the region.
GRH	Integer	32	N	The height of the region.
GRTEMPLATE	Integer	1	N	The template identifier.
GRREFERENCE	Bitmap			The reference bitmap.
GRREFERENCEDX	Integer	32	Y	The X offset of the reference bitmap with respect to the bitmap being decoded.
GRREFERENCEDY	Integer	32	Y	The Y offset of the reference bitmap with respect to the bitmap being decoded.
TPRON	Integer	1	N	Whether typical prediction for refinement is used.
GRATX$_1$	Integer	8	Y	The X position of the adaptive template pixel RA$_1$. *
GRATY$_1$	Integer	8	Y	The Y position of the adaptive template pixel RA$_1$. *
GRATX$_2$	Integer	8	Y	The X position of the adaptive template pixel RA$_2$. *
GRATY$_2$	Integer	8	Y	The Y position of the adaptive template pixel RA$_2$. *

* Unused if **GRTEMPLATE** $\neq 0$

Table 7 — Return values from the generic refinement region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GRREG	Bitmap			The decoded region bitmap.

Table 8 — Variables used in the generic refinement region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
CONTEXT	Integer	16	N	The values of the pixels in the template
LTP	Integer	1	N	Whether the current image line is decoded explicitly *
SLTP	Integer	1	N	Whether the current line's LTP value is different from the previous line's LTP value *
TPRPIX	Integer	1	N	Whether the current pixel is to be decoded implicitly using a TPR prediction *
TPRVAL	Integer	1	N	Value of the TPR-predicted current pixel *

* Unused if **TPRON** $=$ **0**

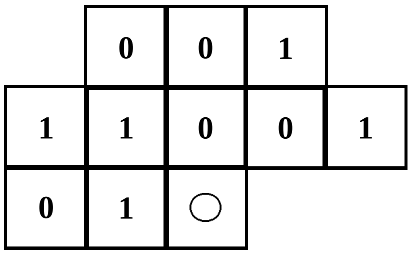


Figure 10 — Reused context for coding the TP pseudo-pixel when GBTEMPLATE is 2.

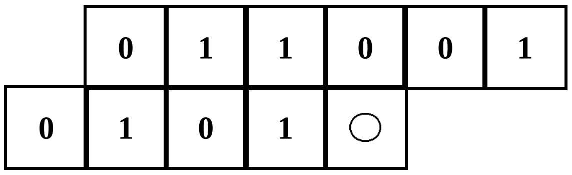


Figure 11 — Reused context for coding the TP pseudo-pixel when GBTEMPLATE is 3.

6.3.5 Decoding using a template and arithmetic coding

6.3.5.1 General description

The generic refinement region decoding procedure is based on arithmetic coding with a template to determine the coding state. The remainder of 6.3.5 describes this form of decoding.

6.3.5.2 Coding Order and Edge Conventions

The coding algorithm iterates through the refine bitmap being decoded, along with a reference bitmap, in raster scan order. That is, it iterates by rows from top to bottom, and within each row from left to right. The processing for a current target pixel will reference the colors of some pixels in fixed spatial relationship to the target pixel. Some of these pixels are drawn from the reference version of the bitmap, and some of these pixels are drawn from the already-coded pixels of the refined bitmap.

Near the edges of the bitmap, neighbor references may not lie in the actual bitmap. The rules to satisfy out-of-bounds references shall be as follows:

- All pixels lying outside the bounds of the actual bitmap or the reference bitmap have the value **0**.

6.3.5.3 Fixed Templates and Adaptive Templates

A template defines a neighborhood around a pixel to be coded. The values of the pixels in this neighborhood define a context. Each context has its own adaptive probability estimate used by the arithmetic coder (see Annex E). Although a template is a geometric pattern of pixels, the pixels in a template are said to take on values when the template is aligned with a particular part of the image.

Figure 12 shows the template which shall be used when **TEMPLATE** is 0. Figure 13 shows the template which shall be used when **TEMPLATE** is 1. In each of these figures, the left-hand group indicates the pixels from the already-coded pixels of the refined bitmap that are in the template, and the right-hand group indicates the pixels from the reference version of the template that are in the template. Each group in each figure includes a pixel denoted by a circle; these pixels all correspond to the pixel to be coded. The pixels marked with an ‘X’ correspond to ordinary pixels in the template. The pixels denoted RA$_1$–RA$_2$ are special pixels in the template. They are denoted “adaptive” or AT pixels. These pixels are special in that their positions are allowed to change during the process of encoding the image. See 6.3.5.4 for a description of AT pixels. The legends RA$_1$–RA$_2$ indicate the nominal locations of AT pixels 1 to 2.

The AT pixel RA_1 can be located anywhere in the field shown in Figure 7, not including the current pixel. The AT pixel RA_2 can be located anywhere in the range $(-128, -128)$ to $(127, 127)$.

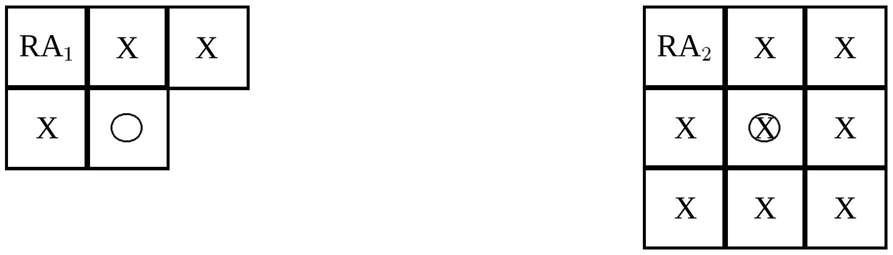


Figure 12 — 13-pixel refinement template showing the AT pixels at their nominal locations.

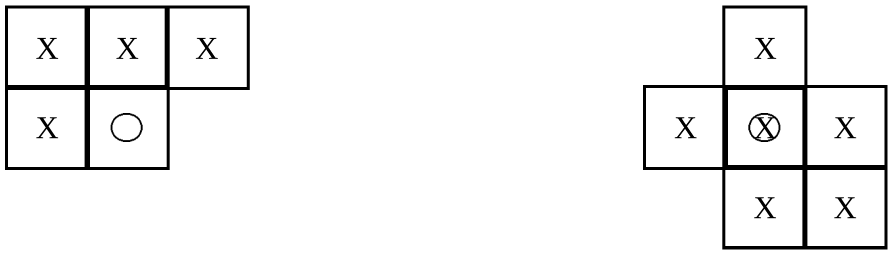


Figure 13 — 10-pixel refinement template

The pixels in the left hand group of each template shall be aligned with the already-decoded pixels of the bitmap being decoded, with the pixel denoted by a circle lying on the pixel to be decoded. Let (X, Y) be the location of this pixel. The pixels of the right hand group of each template shall be aligned with the reference bitmap GRREFERENCE, with the pixel denoted by a circle placed at the location $(X - \textbf{GRREFERENCEDX}, Y - \textbf{GRREFERENCEDY})$. The values of the pixels in the template shall be combined to form a context. Each pixel in the template (including the adaptive pixels) shall correspond to a specific bit in the context, although the pixels in the template may be assigned to bits in the context in any order. Because there are up to 13 pixels in the template, contexts can take on up to 8192 different values. This context shall be used to identify which adaptive probability estimate must be used by the arithmetic coder for encoding the pixel to be coded (see Annex E).

6.3.5.4 Adaptive Template Pixels

In coding the image, the template shall be allowed to change in the restricted way described in this clause.

The pixels that are allowed to change shall be called AT pixels. Their standard positions are indicated by 'RA_1' and 'RA_2' in Figure 12. Note that only one template has AT pixels.

6.3.5.5 Typical Prediction for Refinement (TPR)

Typical prediction can be enabled or disabled with the **TPRON** parameter. If typical prediction is enabled (**TPRON** is **1**) then before the first pixel of each row is decoded, a value indicating whether a row can use typical prediction shall be decoded. If the row can not use typical prediction, each pixel of the row needs to be explicitly decoded. If the row can use typical prediction, all typically-predictable pixels can be implicitly decoded using their predicted value, with the remainder of the pixels still being explicitly decoded. For a pixel to be typically-predictable it must meet certain criteria to be defined below.

6.3.5.6 Decoding the refinement bitmap

The decoding of the bitmap shall proceed as follows.

1. Set LTP $=$ **0**.

2. Create a bitmap GRREG of width **GRW** and height **GRH** pixels.
3. Decode each row as follows
   (a) If all **GRH** rows have been decoded then the decoding is complete; proceed to step 4
   (b) If **TPRON** is **1** then decode a bit using the arithmetic entropy coder, where the context used to decode this bit vares depending on the template in use:
      - If **GRTEMPLATE** is **0**, use the context shown in Figure 14.
      - If **GRTEMPLATE** is **1**, use the context shown in Figure 15.

      Let SLTP be the value of this bit. Set

$$\text{LTP} = \text{LTP XOR SLTP}$$

   (c) If LTP $=$ **0** then, from left to right, explicitly decode all pixels of the current row of GRREG. The procedure for each pixel is as follows:
      i. Place the template given by parameters **GRTEMPLATE** (and $\mathbf{GRATX_1}$, $\mathbf{GRATY_1}$, $\mathbf{GRATX_2}$ and $\mathbf{GRATY_2}$ if **GRTEMPLATE** is **0**) so that the current pixel is aligned with the location denoted y a circle in the figure describing the appearance of the template with identifier **GRTEMPLATE**.
      ii. Form an integer CONTEXT by gathering the values of the image pixels overlaid by the template (including AT pixels) at its current location. The order of this gathering is not standardised, but must be consistent and independent of the location of the AT pixels.
      iii. Decode the current pixel by invoking the arithmetic entropy decoding procedure, with CX set to the value formed by concatenating the label "GR" and the 10–13 pixel values gathered in CONTEXT. The result of this invocation is the value of the current pixel.

      EXAMPLE — If **GRTEMPLATE** is **1**, the image pixels overlaid by the template are as shown in Figure 15, and the pixels are gathered in reading order (in rows from top to bottom, and within each row from left to right, with the pixels in GRREG considered before the pixels in GRREFERENCE), then CX is set to "GR**0000001000**".
   (d) If LTP $=$ **1** then, from left to right, implicitly decode certain pixels of the current row of GRREG, and explictly decode the rest. The procedure for each pixel is as follows:
      i. Set TPRPIX equal to **1** if
         A. **TPRON** is **1** AND
         B. a 3×3 pixel array in the reference bitmap (Figure 16), centered at the location corresponding to the current pixel, contains pixels all of the same value.

         When TPRPIX is set to **1**, set TPRVAL equal to the current pixel predicted value, which is the common value of the nine adjacent pixels in the above single 3×3 array.
      ii. If TRPPIX is **1** then implicitly decode the current pixel by setting it equal to its predicted value (TPRVAL).
      iii. Otherwise, explictly decode the current pixel using the methodology of steps 3(c)i through 3(c)iii above.
4. After all the rows have been decoded, the current contents of the bitmap GRREG are the results that shall be obtained by every decoder, whether it performs this exact sequence of steps or not.

6.4 Symbol Region Decoding Procedure

6.4.1 General Description

This decoding procedure is used to decode a bitmap by decoding a number of symbol instances. A symbol instance contains a location and a symbol ID, and possibly a refinement bitmap. These symbol instances are combined to form the decoded bitmap.

NOTE — This decoding procedure will normally be used to decode the text part of a page. The symbols are normally single text characters from some font or alphabet.

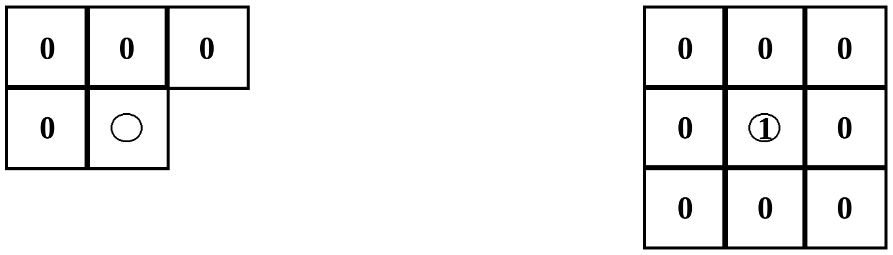


Figure 14 — Reused context for coding the SLTP pseudo-pixel when GRTEMPLATE is 0.

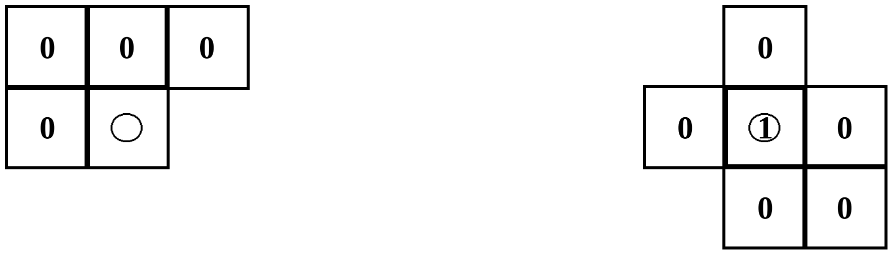


Figure 15 — Reused context for coding the SLTP pseudo-pixel when GRTEMPLATE is 1.

6.4.2 Input parameters

The parameters to this decoding procedure are shown in Table 9.

NOTE — The values of some of these parameters in a typical situation, where a bitmap containing text characters in standard English reading order is being decoded, and **1** is the foreground pixel value, are

- **SBDEFPIXEL** is **0**
- **SBCOMBOP** is OR
- **TRANSPOSED** is **0**
- **REFCORNER** is BOTTOMLEFT

6.4.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table 10.

6.4.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 11.

6.4.5 Decoding the Symbol Bitmap

A symbol-coded bitmap is represented by a set of symbol instances. Each symbol instance encodes a location, a symbol ID, and possibly refinement information. The location of each symbol instance comprises an S coordinate and a T coordinate. If **TRANSPOSED** is **0**, then the S coordinate axis corresponds to the X axis of the bitmap, and the T axis corresponds to the Y axis of the bitmap. If **TRANSPOSED** is **1**, then the S coordinate axis corresponds to the Y axis of the bitmap, and the T axis corresponds to the X axis of the bitmap.

NOTE 1 — Transposing the coordinate axes allows efficient coding of text running vertically. The reference corner is variable because the most efficient coding is usually obtained when the reference

Table 9 — Parameters for the symbol region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
SBHUFF	Integer	1	N	Whether Huffman coding is used.
SBREFINE	Integer	1	N	Whether refinement coding is used.
SBW	Integer	32	N	The width of the region.
SBH	Integer	32	N	The height of the region.
SBNUMINSTANCES	Integer	32	N	The number of symbol instances in this region.
SBSTRIPS	Integer	4	N	The size of the symbol instance strips. May take on the values 1, 2, 4 or 8.
SBNUMSYMS	Integer	32	N	The number of symbols that may be used in this region.
SBSYMCODES	Array of Huffman codes			An array containing the codes for the symbols used in this region. Contains **SBNUMSYMS** codes. *
SBSYMCODELEN	Integer	6	N	The length of the symbol codes used in IAID ****
SBSYMS	Array of symbols			An array containing those symbols. Contains **SBNUMSYMS** symbols.
SBDEFPIXEL	Integer	1	N	The default pixel for this bitmap.
SBCOMBOP	Operator			The combination operator for this symbol region. This parameter may take on the values OR, AND, XOR, and XNOR.
TRANSPOSED	Integer	1	N	Whether the symbol instance coordinates are transposed.
REFCORNER	Corner			Which corner of a symbol instance bitmap is to be used as a reference. This parameter may take on the values TOPLEFT, TOPRIGHT, BOTTOMLEFT and BOTTOMRIGHT.
SBDSOFFSET	Integer	5	Y	An offset for all the delta S values.
SBHUFFFS	Huffman table			The Huffman table used to decode the S coordinate of the first symbol instance in each strip. *
SBHUFFDS	Huffman table			The Huffman table used to decode the S coordinate of subsequent symbol instances in each strip. *
SBHUFFDT	Huffman table			The Huffman table used to decode the difference in T coordinates between non-empty strips. *
SBHUFFRDW	Huffman table			The Huffman table used to decode the difference between a symbol's width and the width of a refinement coded bitmap. **
SBHUFFRDH	Huffman table			The Huffman table used to decode the difference between a symbol's height and the height of a refinement coded bitmap. **
SBHUFFRDX	Huffman table			The Huffman table used to decode the difference between a symbol instance's X coordinate and the X coordinate of a refinement coded bitmap. **
SBHUFFRDY	Huffman table			The Huffman table used to decode the difference between a symbol instance's Y coordinate and the Y coordinate of a refinement coded bitmap. **
SBHUFFRSIZE	Huffman table			The Huffman table used to decode the size of a symbol instance's refinement bitmap data. **
SBRTEMPLATE	Integer	1	N	Template identifier for refinement coding of symbol instance bitmaps. ***
$\mathbf{SBRATX_1}$	Integer	8	Y	The X position of the adaptive template pixel RA_1. ***
$\mathbf{SBRATY_1}$	Integer	8	Y	The Y position of the adaptive template pixel RA_1. ***
$\mathbf{SBRATX_2}$	Integer	8	Y	The X position of the adaptive template pixel RA_2. ***
$\mathbf{SBRATY_2}$	Integer	8	Y	The Y position of the adaptive template pixel RA_2. ***

* Unused if **SBHUFF** $= \mathbf{0}$.
** Unused if **SBHUFF** $= \mathbf{0}$ or **SBREFINE** $= \mathbf{0}$.
*** Unused if **SBREFINE** $= \mathbf{0}$. **** Unused if **SBHUFF** $= \mathbf{1}$.

Table 10 — Return values from the symbol region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
SBREG	Bitmap			The decoded region bitmap.

Table 11 — Variables used in the symbol region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
STRIPT	Integer	32	Y	The numerically smallest T coordinate in the current strip.
FIRSTS	Integer	32	Y	The first S coordinate of the current strip.
NINSTANCES	Integer	32	N	A symbol instance counter.
DT	Integer	32	Y	The number of empty strips between two non-empty strips.
DFS	Integer	32	Y	The difference in S coordinates between the first instances of two strips.
CURS	Integer	32	Y	The current S coordinate.
CURT	Integer	3	N	The current symbol instance's T coordinate relative to the current strip.
S_I	Integer	32	Y	A symbol instance's S coordinate.
T_I	Integer	32	Y	A symbol instance's T coordinate.
ID_I	Integer	32	N	A symbol instance's symbol ID.
IB_I	Bitmap			A symbol instance's symbol bitmap.
W_I	Integer	32	N	The width of a symbol instance's symbol bitmap.
H_I	Integer	32	N	The height of a symbol instance's symbol bitmap.
IDS	Integer	32	Y	The difference in S coordinates between two symbol instances within a strip.
R_I	Integer	1	N	Whether a symbol instance's symbol bitmap is coded using refinement.
RDW_I	Integer	32	Y	The delta width of a symbol instance's refinement bitmap. *
RDH_I	Integer	32	Y	The delta height of a symbol instance's refinement bitmap. *
RDX_I	Integer	32	Y	The delta X of a symbol instance's refinement bitmap. *
RDY_I	Integer	32	Y	The delta Y of a symbol instance's refinement bitmap. *
IBO_I	Bitmap			A symbol instance's original symbol bitmap. *
WO_I	Integer	32	N	The width of IBO_I. *
HO_I	Integer	32	N	The height of IBO_I. *

*Unused if **SBREFINE** $=$ **0**.

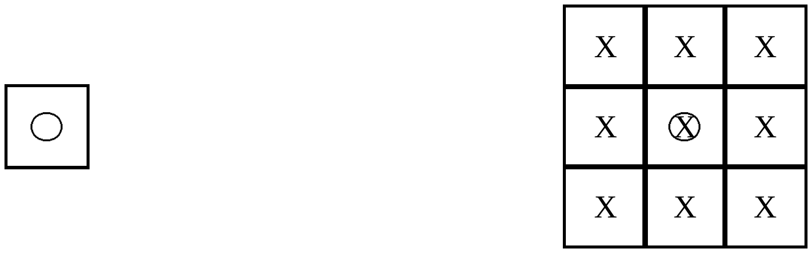


Figure 16 — TPRB template.

corner of each symbol instance lies on a text baseline, and the text baselines may run in any direction.

In order to improve compression, symbol instances are grouped into strips according to their T_I values. This is done according to the value of **SBSTRIPS**. Symbol instances having T_I values between 0 and **SBSTRIPS** $- 1$ are grouped into one strip, symbol instances having T_I values between **SBSTRIPS** and $2 \times$ **SBSTRIPS** $- 1$ into the next, and so on. Within each strip, the symbol instances are coded in the order of increasing S coordinate.

NOTE 2 — Normally the strips occur in the order of strictly increasing T coordinates, and the symbol instances within each strip occur in the order of nondecreasing S coordinates. However, it is possible for negative delta S or delta T values to occur during the decoding, meaning that the strips and symbol instances might occur in any order.

The overall structure of the data to be decoded in order to reconstruct the symbol region is shown in Figure 17. The format of each strip is as shown in Figure 18. When **SBREFINE** is **0**, the format of each symbol instance is as shown in Figure 19. When **SBREFINE** is **1**, the format of each symbol instance is as shown in Figure 20.

NOTE 3 — There may be some symbol instances whose reference corner lies off the top of the region. If these are to be coded, there must be some way to have a strip that also lies above the top of the region. The initial value of STRIPT is the coordinate with respect to which the first strip is located.

Initial STRIPT value
First strip
Second strip
. . .
Last strip

Figure 17 — Coded structure of a symbol region.

Delta T
First symbol instance
Second symbol instance
. . .
Last symbol instance
OOB

Figure 18 — Structure of a strip.

The result of decoding a symbol region shall be the bitmap that is produced by the following steps.

Symbol instance S coordinate
Symbol instance T coordinate
Symbol instance symbol ID

Figure 19 — Structure of a symbol instance when SBREFINE is 0.

Symbol instance S coordinate
Symbol instance T coordinate
Symbol instance symbol ID
Symbol instance refinement information

Figure 20 — Structure of a symbol instance when SBREFINE is 1.

1. Fill a bitmap SBREG, of the size given by **SBW** and **SBH**, with the **SBDEFPIXEL** value.
2. Decode the initial STRIPT value as described in 6.4.6. Negate the decoded value and assign this negated value to the variable STRIPT. Assign the value 0 to FIRSTS. Assign the value 0 to NINSTANCES.
3. Decode each strip as follows.
   (a) If NINSTANCES is equal to **SBNUMINSTANCES** then there are no more strips to decode, and the process of decoding the symbol region is complete; proceed to step 4.
   (b) Decode the strip's delta T value as described in 6.4.6. Let DT be the decoded value. Set

$$\mathrm{STRIPT} = \mathrm{STRIPT} + \mathrm{DT}$$

   (c) Decode each symbol instance in the strip as follows.
      i. If the current symbol instance is the first symbol instance in the strip, then decode the first symbol instance's instance S coordinate as described in 6.4.7. Let DFS be the decoded value. Set

$$\begin{aligned}\mathrm{FIRSTS} &= \mathrm{FIRSTS} + \mathrm{DFS}\\ \mathrm{CURS} &= \mathrm{FIRSTS}\end{aligned}$$

      ii. Otherwise, if the current symbol instance is not the first symbol instance in the strip, decode the symbol instance's instance S coordinate as described in 6.4.8. If the result of this decoding is OOB then the last symbol instance of the strip has been decoded; proceed to step 3d. Otherwise, let IDS be the decoded value. Set

$$\mathrm{CURS} = \mathrm{CURS} + \mathrm{IDS} + \mathbf{SBDSOFFSET}$$

      NOTE — The intended use of **SBDSOFFSET** is to make the cost common value decoded in 6.4.8 zero. The shortest code in all of the default tables used in 6.4.8 is for the value zero.

      iii. Decode the symbol instance's instance T coordinate as described in 6.4.9. Let CURT be the decoded value. Set

$$T_I = \mathrm{STRIPT} + \mathrm{CURT}$$

      iv. Decode the symbol instance's instance symbol ID as described in 6.4.10. Let ID_I be the decoded value.
      v. Determine the symbol instance's bitmap IB_I as described in 6.4.11. The width and height of this bitmap shall be denoted as W_I and H_I respectively.

vi. Update CURS as follows.
   - If **TRANSPOSED** is **0**, and **REFCORNER** is TOPRIGHT or BOTTOMRIGHT, set

$$\text{CURS} = \text{CURS} + W_I - 1$$

   - If **TRANSPOSED** is **1**, and **REFCORNER** is BOTTOMLEFT or BOTTOMRIGHT, set

$$\text{CURS} = \text{CURS} + H_I - 1$$

   - Otherwise, do not change CURS in this step.

vii. Set

$$S_I = \text{CURS}$$

viii. Determine the location of the symbol instance bitmap with respect to SBREG as follows.
   - If **TRANSPOSED** is **0**, then
     - If **REFCORNER** is TOPLEFT then the top left pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[S_I, T_I]$.
     - If **REFCORNER** is TOPRIGHT then the top right pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[S_I, T_I]$.
     - If **REFCORNER** is BOTTOMLEFT then the bottom left pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[S_I, T_I]$.
     - If **REFCORNER** is BOTTOMRIGHT then the bottom right pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[S_I, T_I]$.
   - If **TRANSPOSED** is **1**, then
     - If **REFCORNER** is TOPLEFT then the top left pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[T_I, S_I]$.
     - If **REFCORNER** is TOPRIGHT then the top right pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[T_I, S_I]$.
     - If **REFCORNER** is BOTTOMLEFT then the bottom left pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[T_I, S_I]$.
     - If **REFCORNER** is BOTTOMRIGHT then the bottom right pixel of the symbol instance bitmap IB_I shall be placed at SBREG$[T_I, S_I]$.

   If any part of IB_I, when placed at this location, lies outside the bounds of SBREG, then ignore this part of IB_I in step 3(c)ix.

ix. Draw IB_I into SBREG. Combine each pixel of IB_I with the current value of the corresponding pixel in SBREG, using the combination operator specified by **SBCOMBOP**. Write the results of each combination into that pixel in SBREG.

x. Update CURS as follows.
   - If **TRANSPOSED** is **0**, and **REFCORNER** is TOPLEFT or BOTTOMLEFT, set

$$\text{CURS} = \text{CURS} + W_I - 1$$

   - If **TRANSPOSED** is **1**, and **REFCORNER** is TOPLEFT or TOPRIGHT, set

$$\text{CURS} = \text{CURS} + H_I - 1$$

   - Otherwise, do not change CURS in this step.

   NOTE — The CURS update rules are designed to allow the gap between adjacent symbol instances to be encoded, rather than the distance between their reference corners; this takes out one source of variation (the symbol instance bitmap width or height), and allows better compression.

xi. Set

$$\text{NINSTANCES} = \text{NINSTANCES} + 1$$

(d) When the strip has been completely decoded, decode the next strip.

4. After all the strips have been decoded, the current contents of SBREG are the results that shall be obtained by every decoder, whether it performs this exact sequence of steps or not.

6.4.6 Strip delta T

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFDT** and multiply the resulting value by **SBSTRIPS**.

If **SBHUFF** is **0**, decode a value using the IADT integer arithmetic decoding procedure (see A) and multiply the resulting value by **SBSTRIPS**.

6.4.7 First symbol instance S coordinate

NOTE — The instance S coordinate value for a the first symbol instance of each strip is coded differently from the subsequent symbol instances in each strip. This takes advantage of the beginnings of lines being aligned.

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFFS**.

If **SBHUFF** is **0**, decode a value using the IAFS integer arithmetic decoding procedure (see A).

6.4.8 Subsequent symbol instance S coordinate

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFDS**.

If **SBHUFF** is **0**, decode a value using the IADS integer arithmetic decoding procedure (see A).

In either case it is possible that the result of this decoding is the out-of-band value OOB.

6.4.9 Symbol instance T coordinate

If **SBHUFF** is **1**, decode a value by reading $\lceil \log_2 \boldsymbol{SBSTRIPS} \rceil$ bits directly from the bitstream.

If **SBHUFF** is **0**, decode a value using the IAIT integer arithmetic decoding procedure (see A).

6.4.10 Symbol instance symbol ID

If **SBHUFF** is **1**, decode a value by reading one bit at a time until the resulting bit string is equal to one of the entries in **SBSYMCODES**. The resulting value, which is ID_I, is the index of the entry in **SBSYMCODES** that is read.

If **SBHUFF** is **0**, decode a value using the IAID integer arithmetic decoding procedure (see A). Set ID_I to the resulting value.

6.4.11 Symbol instance bitmap

In some cases, the symbol instance bitmap IB_I is simply the bitmap of the symbol identified by ID_I. In other cases, however, the symbol instance bitmap is that bitmap modified by additional refinement information. The bit indicating which of the options is true for a symbol instance is called R_I.

If **SBREFINE** is **0**, then set R_I to **0**.

If **SBREFINE** is **1**, then decode R_I as follows.

- If **SBHUFF** is **1**, then read one bit and set R_I to the value of that bit.
- If **SBHUFF** is **0**, then decode one bit using the IARI integer arithmetic decoding procedure and set R_I to the value of that bit.

If R_I is **0** then set the symbol instance bitmap IB_I to **SBSYMS**$[ID_I]$.

If R_I is **1** then determine the symbol instance bitmap as follows:

1. Decode the instance refinement delta width as described in 6.4.11.1. Let RDW_I be the value decoded.
2. Decode the instance refinement delta height as described in 6.4.11.2. Let RDH_I be the value decoded.

3. Decode the instance refinement X offset as described in 6.4.11.3. Let RDX_I be the value decoded.

4. Decode the instance refinement Y offset as described in 6.4.11.4. Let RDY_I be the value decoded.

5. If **SBHUFF** is **1**, then

   (a) Decode the instance refinement bitmap data size as described in 6.4.11.5.

   (b) Skip over any bits remaining in the last byte read.

6. Let IBO_I be **SBSYMS**$[ID_I]$. Let WO_I be the width of IBO_I and HO_I be the height of IBO_I. The symbol instance bitmap IB_I is the result of applying the generic refinement region decoding procedure described in 6.3. Set the parameters to this decoding procedure as shown in Table 12.

Table 12 — Parameters used to decode a symbol instance's bitmap using refinement.

Name	Value
GRW	$WO_I + RDW_I$
GRH	$HO_I + RDH_I$
GRTEMPLATE	**SBRTEMPLATE**
GRREFERENCE	IBO_I
GRREFERENCEDX	$\lfloor RDW_I/2 \rfloor + RDX_I$
GRREFERENCEDY	$\lfloor RDH_I/2 \rfloor + RDY_I$
TRPON	**0**
GRATX$_1$	**SBRATX$_1$**
GRATY$_1$	**SBRATY$_1$**
GRATX$_2$	**SBRATX$_2$**
GRATY$_2$	**SBRATY$_2$**

7. If **SBHUFF** is **1**, then skip over any bits remaining in the last byte read. The total number of bytes processed by the generic refinement bitmap decoding procedure must be equal to the value read in step 5a.

6.4.11.1 Symbol instance refinement delta width

This field, and the following fields, indicate the size, location and contents of the refined symbol bitmap, as the size may not be the same as the size of the bitmap of the symbol whose ID is given in this symbol instance; also, the change in the size of the bitmap might extend to the left and top, not just to the right and bottom, so we need to supply an offset as well as a size. Note that the offsets are given in terms of X and Y, not S and T.

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFRDW**.

If **SBHUFF** is **0**, decode a value using the IARDW integer arithmetic decoding procedure (see A).

6.4.11.2 Symbol instance refinement delta height

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFRDH**.

If **SBHUFF** is **0**, decode a value using the IARDH integer arithmetic decoding procedure (see A).

6.4.11.3 Symbol instance refinement X offset

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFRDX**.

If **SBHUFF** is **0**, decode a value using the IARDX integer arithmetic decoding procedure (see A).

6.4.11.4 Symbol instance refinement Y offset

If **SBHUFF** is **1**, decode a value using the Huffman table specified by **SBHUFFRDY**.

If **SBHUFF** is **0**, decode a value using the IARDY integer arithmetic decoding procedure (see A).

6.4.11.5 Symbol instance refinement bitmap data size

Decode a value using the Huffman table specified by **SBHUFFRSIZE**.

6.5 Symbol Dictionary Decoding Procedure

This decoding procedure is used to decode a set of symbols; these symbols can then be used by symbol region decoding procedures, or in some cases by other symbol dictionary decoding procedures.

6.5.1 Input parameters

The parameters to this decoding procedure are shown in Table 13.

The **SDREFAGG** parameter determines how the symbols in this symbol dictionary are coded. If **SDREFAGG** is **0** then each symbol bitmap is coded via direct bitmap coding. If **SDREFAGG** is **1** then each symbol bitmap is coded by refining or aggregating previously-defined symbol bitmaps. These previously-defined symbol bitmaps may be drawn from other dictionaries and provided as input to this decoding procedure in **SDINSYMS**, or may be defined in the current dictionary.

6.5.2 Return values

The variables whose values are the result of this decoding procedure are shown in Table 14.

6.5.3 Variables used in decoding

The variables used by this decoding procedure are shown in Table 15.

6.5.4 Decoding the Symbol Dictionary

The internal structure of a symbol dictionary is shown in Figure 21. The symbols defined in the dictionary are ordered into height classes: a height class contains a number of symbols whose bitmaps are the same height.

NOTE — In most cases, the height classes occur in the order of strictly increasing height, shortest through tallest. If **SDREFAGG** is **1**, though, a symbol may be coded as a refinement of a larger symbol defined in the same dictionary. In this case, the height class for that base symbol must be decoded (and therefore must occur) before the shorter height class of the symbol that is coded by refining it. For this reason, height class delta heights (and symbol delta widths) may be zero or negative, as well as positive.

First height class
Second height class
. . .
Last height class
List of exported symbols

Figure 21 — The structure of a symbol dictionary.

If **SDHUFF** is **1** and **SDREFAGG** is **0** then the format of a height class is as shown in Figure 22. Otherwise, the format of a height class is as shown in Figure 23. The fields mentioned in those figures are described fully below.

Height class delta height
Delta width for first symbol
Delta width for second symbol
. . .
OOB
Height class collective bitmap

Figure 22 — Height class coding when SDHUFF is 1 and SDREFAGG is 0.

The result of decoding a symbol dictionary is an array SDEXSYMS containing **SDNUMEXSYMS** bitmaps. This array shall be the array produced by the following steps.

1. Create an array SDNEWSYMS of bitmaps, having **SDNUMNEWSYMS** entries.

Table 13 — Parameters for the symbol dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
SDHUFF	Integer	1	N	Whether Huffman coding is used.
SDREFAGG	Integer	1	N	Whether refinement and aggregate coding are used.
SDNUMINSYMS	Integer	32	N	The number of symbols that are used as input to this symbol dictionary decoding procedure.
SDINSYMS	Array of symbols			An array containing the symbols that are used as input to this symbol dictionary decoding procedure. Contains **SDNUMINSYMS** symbols.
SDNUMNEWSYMS	Integer	32	N	The number of symbols to be defined in this symbol dictionary.
SDNUMEXSYMS	Integer	32	N	The number of symbols to be exported from this symbol dictionary.
SDHUFFDH	Huffman table			The Huffman table used to decode the difference in height between two height classes. *
SDHUFFDW	Huffman table			The Huffman table used to decode the difference in width between two symbols. *
SDHUFFBMSIZE	Huffman table			The Huffman table used to decode the size of a height class collective bitmap. *
SDHUFFAGGINST	Huffman table			The Huffman table used to decode the number of instances in an aggregation. **
SDTEMPLATE	Integer	2	N	The template identifier used to decode symbol bitmaps. ***
$\mathbf{SDATX_1}$	Integer	8	Y	The X position of the adaptive template pixel A_1. ***
$\mathbf{SDATY_1}$	Integer	8	Y	The Y position of the adaptive template pixel A_1. ***
$\mathbf{SDATX_2}$	Integer	8	Y	The X position of the adaptive template pixel A_2. ***
$\mathbf{SDATY_2}$	Integer	8	Y	The Y position of the adaptive template pixel A_2. ***
$\mathbf{SDATX_3}$	Integer	8	Y	The X position of the adaptive template pixel A_3. ***
$\mathbf{SDATY_3}$	Integer	8	Y	The Y position of the adaptive template pixel A_3. ***
$\mathbf{SDATX_4}$	Integer	8	Y	The X position of the adaptive template pixel A_4. ***
$\mathbf{SDATY_4}$	Integer	8	Y	The Y position of the adaptive template pixel A_4. ***
SDRTEMPLATE	Integer	1	N	Template identifier for refinement coding of bitmaps. ****
$\mathbf{SDRATX_1}$	Integer	8	Y	The X position of the adaptive template pixel RA_1. ****
$\mathbf{SDRATY_1}$	Integer	8	Y	The Y position of the adaptive template pixel RA_1. ****
$\mathbf{SDRATX_2}$	Integer	8	Y	The X position of the adaptive template pixel RA_2. ****
$\mathbf{SDRATY_2}$	Integer	8	Y	The Y position of the adaptive template pixel RA_2. ****

* Unused if **SDHUFF** $=$ **0**.
** Unused if **SDHUFF** $=$ **0** or **SDREFAGG** $=$ **0**.
*** Unused if **SDHUFF** $=$ **1**.
**** Unused if **SDREFAGG** $=$ **0**.

Table 14 — Return values from the symbol dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
SDEXSYMS	Array of symbols			The symbols exported by this symbol dictionary. Contains **SDNUMEXSYMS** symbols.

Table 15 — Variables used in the symbol dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
SDNEWSYMS	Array of symbols			The symbols defined in this symbol dictionary. Contains **SDNUMNEWSYMS** symbols.
SDNEWSYMWIDTHS	Array of integers			The widths of the symbols in SDNEWSYMS. Contains **SDNUMNEWSYMS** integers. Each integer is a 32-bit unsigned value.
HCHEIGHT	Integer	32	N	Height of the current height class.
NSYMSDECODED	Integer	32	N	How many symbols have been decoded so far.
HCDH	Integer	32	Y	The difference in height between two height classes.
SYMWIDTH	Integer	32	N	The width of the current symbol.
TOTWIDTH	Integer	32	N	The width of the current height class.
HCFIRSTSYM	Integer	32	N	The index of the first symbol in the current height class.
DW	Integer	32	Y	The difference in width between two symbols.
B_S	Bitmap			The current symbol's bitmap.
B_{HC}	Bitmap			The current height class collective bitmap.
I	Integer	32	N	An array index.
J	Integer	32	N	An array index.
REFAGGNINST	Integer	32	N	The number of symbol instances in an aggregation.
EXFLAGS	Array of integers			The export flags for this dictionary. Contains **SDNUMINSYMS** $+$ **SDNUMNEWSYMS** values. Each value is one bit.
EXINDEX	Integer	32	N	An array index.
CUREXFLAG	Integer	1	N	The current export flag.
EXRUNLENGTH	Integer	32	N	The length of a run of identical export flag values.

Height class delta height
Delta width for first symbol
Bitmap for first symbol
Delta width for second symbol
Bitmap for second symbol
. . .
OOB

Figure 23 — Height class coding when SDHUFF is 0 or SDREFAGG is 1.

2. If **SDHUFF** is **1** and **SDREFAGG** is **0**, create an array SDNEWSYMWIDTHS of integers, having **SDNUMNEWSYMS** entries.

3. Set

$$\begin{aligned} \text{HCHEIGHT} &= 0 \\ \text{NSYMSDECODED} &= 0 \end{aligned}$$

4. Decode each height class as follows.

   (a) If NSYMSDECODED $=$ **SDNUMNEWSYMS** then all the symbols in the dictionary have been decoded; proceed to step 5.

   (b) Decode the height class delta height as described in 6.5.5. Let HCDH be the decoded value. Set

$$\begin{aligned} \text{HCHEIGHT} &= \text{HCHEIGHT} + \text{HCDH} \\ \text{SYMWIDTH} &= 0 \\ \text{TOTWIDTH} &= 0 \\ \text{HCFIRSTSYM} &= \text{NSYMSDECODED} \end{aligned}$$

   (c) Decode each symbol within the height class as follows.

      i. Decode the delta width for the symbol as described in 6.5.6. If the result of this decoding is OOB then all the symbols in this height class have been decoded; proceed to step 4d. Otherwise let DW be the decoded value and set

$$\begin{aligned} \text{SYMWIDTH} &= \text{SYMWIDTH} + \text{DW} \\ \text{TOTWIDTH} &= \text{TOTWIDTH} + \text{SYMWIDTH} \end{aligned}$$

      ii. If **SDHUFF** is **0** or **SDREFAGG** is **1** then decode the symbol's bitmap as described in 6.5.7. Let B_S be the decoded bitmap (this bitmap has width SYMWIDTH and height HCHEIGHT). Set

$$\text{SDNEWSYMS}[\text{NSYMSDECODED}] = B_S$$

      iii. If **SDHUFF** is **1** and **SDREFAGG** is **0** then set

$$\text{SDNEWSYMWIDTHS}[\text{NSYMSDECODED}] = \text{SYMWIDTH}$$

      iv. Set

$$\text{NSYMSDECODED} = \text{NSYMSDECODED} + 1$$

(d) If **SDHUFF** is **1** and **SDREFAGG** is **0** then decode the height class collective bitmap as described in 6.5.8. Let B_{HC} be the decoded bitmap. This bitmap has width TOTWIDTH and height HCHEIGHT. Determine the symbols SDNEWSYMS[HCFIRSTSYM] through SDNEWSYMS[NSYMSDECODED$-$1] by breaking up the bitmap B_{HC}.

B_{HC} contains the NSYMSDECODED $-$ HCFIRSTSYM symbols concatenated left-to-right, with no intervening gaps. For each I between HCFIRSTSYM and NSYMSDECODED $-$ 1,

- the width of SDNEWSYMS[I] is the value of SDNEWSYMWIDTHS[I],
- the height of SDNEWSYMS[I] is HCHEIGHT, and
- the bitmap SDNEWSYMS[I] can be obtained by extracting the columns of B_{HC} from

$$\sum_{J=\text{HCFIRSTSYM}}^{I-1} \text{SDNEWSYMWIDTHS}[J]$$

through

$$\left(\sum_{J=\text{HCFIRSTSYM}}^{I} \text{SDNEWSYMWIDTHS}[J] \right) - 1$$

EXAMPLE — The bitmap for SDNEWSYMS[HCFIRSTSYM], the first symbol in the height class, can be obtained by copying the columns 0 through

$$\text{SDNEWSYMWIDTHS}[\text{HCFIRSTSYM}] - 1$$

of B_{HC}.

5. Determine which symbol bitmaps are exported from this symbol dictionary, as described in 6.5.9. These bitmaps can be drawn from the symbols that are used as input to the symbol dictionary decoding procedure as well as the new symbols produced by the decoding procedure.

NOTE — Not all the new symbols need to be exported; this allows the dictionary to define some a symbol, use it via refinement/aggregate coding to build other symbols, and not actually export the original symbol. Also, since input symbols can be exported, this dictionary can in effect copy symbols from other dictionaries.

6.5.5 Height class delta height

If **SDHUFF** is **1**, decode a value using the Huffman table specified by **SDHUFFDH**.

If **SDHUFF** is **0**, decode a value using the IADH integer arithmetic decoding procedure (see A).

6.5.6 Delta width

If **SDHUFF** is **1**, decode a value using the Huffman table specified by **SDHUFFDW**.

If **SDHUFF** is **0**, decode a value using the IADW integer arithmetic decoding procedure (see A).

In either case it is possible that the result of this decoding is the out-of-band value OOB.

6.5.7 Bitmap

This field takes one of two forms.

6.5.7.1 Direct-coded bitmap

If **SDREFAGG** is **0** then decode the symbol's bitmap using a generic region decoding procedure as described in 6.2. Set the parameters to this decoder as shown in Table 16.

6.5.7.2 Refinement/aggregate-coded bitmap

If **SDREFAGG** is **1** then the symbol's bitmap is coded by refinement and aggregation of other, previously-defined, symbols. Decode the bitmap as follows.

First, decode the number of instances contained in the aggregation, as specified in 6.5.7.2.1. Let REFAGGNINST be the value decoded.

Next, decode the bitmap itself using a symbol region decoder as described in 6.4. Set the parameters to this decoder as shown in Table 17.

Table 16 — Parameters used to decode a symbol's bitmap using generic bitmap decoding.

Name	Value
MMR	**0**
GBW	SYMWIDTH
GBH	HCHEIGHT
GBTEMPLATE	**SDTEMPLATE**
TPON	**0**
USESKIP	**0**
$\mathbf{GBATX_1}$	$\mathbf{SDATX_1}$
$\mathbf{GBATY_1}$	$\mathbf{SDATY_1}$
$\mathbf{GBATX_2}$	$\mathbf{SDATX_2}$
$\mathbf{GBATY_2}$	$\mathbf{SDATY_2}$
$\mathbf{GBATX_3}$	$\mathbf{SDATX_3}$
$\mathbf{GBATY_3}$	$\mathbf{SDATY_3}$
$\mathbf{GBATX_4}$	$\mathbf{SDATX_4}$
$\mathbf{GBATY_4}$	$\mathbf{SDATY_4}$

Table 17 — Parameters used to decode a symbol's bitmap using refinement/aggregate decoding.

Name	Value
SBHUFF	**SDHUFF**
SBREFINE	**1**
SBW	SYMWIDTH
SBH	HCHEIGHT
SBNUMINSTANCES	REFAGGNINST
SBSTRIPS	**1**
SBNUMSYMS	**SDNUMINSYMS** + NSYMSDECODED
SBSYMCODES	See 6.5.7.2.2. *
SBSYMCODELEN	See 6.5.7.2.2.
SBSYMS	See 6.5.7.2.3.
SBDEFPIXEL	**0**
SBCOMBOP	OR
TRANSPOSED	**0**
REFCORNER	TOPLEFT
SBDSOFFSET	0
SBHUFFFS	Table B.6 *
SBHUFFDS	Table B.8 *
SBHUFFDT	Table B.11 *
SBHUFFRDW	Table B.15 *
SBHUFFRDH	Table B.15 *
SBHUFFRDX	Table B.15 *
SBHUFFRDY	Table B.15 *
SBHUFFRSIZE	Table B.1 *
SBRTEMPLATE	**SDRTEMPLATE**
$\mathbf{SBRATX_1}$	$\mathbf{SDRATX_1}$
$\mathbf{SBRATY_1}$	$\mathbf{SDRATY_1}$
$\mathbf{SBRATX_2}$	$\mathbf{SDRATX_2}$
$\mathbf{SBRATY_2}$	$\mathbf{SDRATY_2}$

* If **SDHUFF** $=$ **0** then this parameter has no value

6.5.7.2.1 Number of instances in aggregation

If **SDHUFF** is **1**, decode a value using the Huffman table specified by **SDHUFFAGGINST**.

If **SDHUFF** is **0**, decode a value using the IAAI integer arithmetic decoding procedure (see A).

6.5.7.2.2 Setting SBSYMCODES and SBSYMCODELEN

Set **SBSYMCODES** to an array of **SBNUMSYMS** codes, where the length of each code is

$$\max\left(\lceil \log_2\left(\textbf{SDNUMINSYMS} + \textbf{SDNUMNEWSYMS}\right)\rceil, 1\right)$$

and the code **SBSYMCODES**$[I]$ is I (for I between 0 and **SBNUMSYMS** $- 1$).

NOTE — This sets the codes as equal-length codes, assigned starting from zero. The code lengths are computed from the maximum number of symbols available in this symbol dictionary: all the imported symbols and all the symbols defined here. There is some wastage in choosing this code length and assigning these codes. However, doing it this way means that neither the code lengths nor the actual codes assigned to each symbol changes during the process of decoding this symbol dictionary.

Similarly, when **SDHUFF** is **0**, **SBSYMCODELEN** should be set to

$$\lceil \log_2\left(\textbf{SDNUMINSYMS} + \textbf{SDNUMNEWSYMS}\right)\rceil$$

so that the length of the bit srings decoded using IAID will not change during the decoding of this symbol dictionary.

6.5.7.2.3 Setting SBSYMS

Set **SBSYMS** to an array of **SDNUMINSYMS**$+$NSYMSDECODED symbols, formed by concatenating the array **SDINSYMS** and the first NSYMSDECODED entries of the array SDNEWSYMS.

6.5.8 Height class collective bitmap

This field contains the bitmaps of all the symbols in the height class, concatenated left to right, and MMR encoded. It is preceded by a count of its size in bytes.

This field is decoded as follows.

1. Read the size in bytes using the **SDHUFFBMSIZE** bitmap decoder. Let BMSIZE be the value decoded.

2. Skip over any bits remaining in the last byte read.

3. If BMSIZE is zero, then the bitmap is stored uncompressed, and the actual size in bytes is

$$\text{HCHEIGHT} \times \left\lceil \frac{\text{TOTWIDTH} + 7}{8} \right\rceil$$

   Decode the bitmap by reading this many bytes and treating it as HCHEIGHT rows of TOTWIDTH pixels, each row padded out to a byte boundary with 0–7 **0** bits.

4. Otherwise, decode the bitmap using a generic bitmap decoder as described in 6.2. Set the parameters to this decoder as shown in Table 18.

Table 18 — Parameters used to decode a height class collective bitmap.

Name	Value
MMR	**1**
GBW	TOTWIDTH
GBH	HCHEIGHT

5. Skip over any bits remaining in the last byte read.

6.5.9 Exported symbols

The symbols that may be exported from a given dictionary include any of the symbols that are input to the dictionary, plus any of the symbols defined in the dictionary.

The array of symbols exported from the dictionary is produced by decoding an a bit for each of those symbols. These bits form an array EXFLAGS of **SDNUMINSYMS** + **SDNUMNEWSYMS** binary values, each one corresponding to an input symbol or a newly-defined symbol. A **1** bit for a symbol indicates that the symbol is exported. Exactly **SDNUMEXSYMS** symbols must be exported from the dictionary. The order of exported symbols is the order produced by concatenating the array **SDINSYMS** and the array SDNEWSYMS.

The following procedure produces this array of exported symbols.

1. Set

$$\begin{aligned}\text{EXINDEX} &= 0\\ \text{CUREXFLAG} &= 0\end{aligned}$$

2. Decode a value using the table given in Table B.1 if **SDHUFF** is **1**, or the IAEX integer arithmetic decoding procedure if **SDHUFF** is **0**. Let EXRUNLENGTH be the decoded value.

3. Set EXFLAGS[EXINDEX] through EXFLAGS[EXINDEX + EXRUNLENGTH $-$ 1] to CUREXFLAG. If EXRUNLENGTH $= 0$, then this step does not change any values.

4. Set

$$\begin{aligned}\text{EXINDEX} &= \text{EXINDEX} + \text{EXRUNLENGTH}\\ \text{CUREXFLAG} &= \text{NOT}(\text{CUREXFLAG})\end{aligned}$$

5. Repeat steps 2 through 4 until EXINDEX = SDNUMINSYMS + SDNUMNEWSYMS.

6. The array EXFLAGS now contains **1** for each symbol that is exported from the dictionary, and **0** for each symbol that is not exported.

7. Set

$$\begin{aligned}I &= 0\\ J &= 0\end{aligned}$$

8. For each value of I from 0 to SDNUMINSYMS + SDNUMNEWSYMS $-$ 1, if EXFLAGS[I] = **1** then perform the following steps.

    (a) If $I <$ **SDNUMINSYMS** then set

    $$\begin{aligned}\text{SDEXSYMS}[J] &= \text{SDINSYMS}[I]\\ J &= J+1\end{aligned}$$

    (b) If $I \geq$ **SDNUMINSYMS** then set

    $$\begin{aligned}\text{SDEXSYMS}[J] &= \text{SDNEWSYMS}[I - \textbf{SDNUMINSYMS}]\\ J &= J+1\end{aligned}$$

NOTE — Most dictionaries will export exactly the new symbols that they define; they will not export any of the symbols in **SDINSYMS**. In this case, the first **SDNUMINSYMS** values in EXFLAGS are **0**, and the remaining **SDNUMNEWSYMS** values are **1**.

6.6 Halftone Region Decoding Procedure

6.6.1 General Description

This decoding procedure is used to decode a bitmap by decoding an array of values, which are used to draw halftone patterns into a halftone grid. These halftone patterns are combined to form the decoded bitmap.

6.6.2 Input parameters

The parameters to this decoding procedure are shown in Table 19.

Table 19 — Parameters for the halftone region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
HBW	Integer	32	N	The width of the region.
HBH	Integer	32	N	The height of the region.
HMMR	Integer	1	N	Whether MMR coding is used.
HTEMPLATE	Integer	2	N	The template identifier. *
HNUMPATS	Integer	32	N	The number of halftone patterns that may be used in this region.
HPATS	Array of halftone patterns			An array containing the halftone patterns used in this region. Contains **HNUMPATS** halftone patterns.
HDEFPIXEL	Integer	1	N	The default pixel for this bitmap.
HCOMBOP	Operator			The combination operator for this halftone region. This parameter may take on the values REPLACE, OR, AND, XOR, and XNOR.
HENABLESKIP	Integer	1	N	Whether unneeded pixels are skipped. *
HGW	Integer	32	N	The width of the gray-scale image.
HGH	Integer	32	N	The height of the gray-scale image.
HGX	Integer	32	Y	256 times the horizontal offset of the grid origin.
HGY	Integer	32	Y	256 times the vertical offset of the grid origin.
HRX	Integer	16	Y	256 times the horizontal component of the grid vector.
HRY	Integer	16	Y	256 times the vertical component of the grid vector.
HPW	Integer	8	N	The width of each halftone pattern.
HPH	Integer	8	N	The height of each halftone pattern.

* Unused if **HMMR** $=$ **1**.

6.6.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table 20.

Table 20 — Return values from the halftone region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
HTREG	Bitmap			The decoded region bitmap.

6.6.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 21.

6.6.5 Decoding the Halftone Bitmap

A halftone-coded bitmap is represented by a set of halftone pattern instances. Each instance encodes a halftone pattern. The location of each halftone pattern is not coded explicitly but given by a grid global to the entire halftone

Table 21 — Variables used in the halftone region decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
n_g	Integer	32	N	Horizontal index for the current gray-scale value.
m_g	Integer	32	N	Vertical index for the current gray-scale value.
x	Integer	32	Y	The horizontal coordinate for the pattern corresponding to the current gray-scale value.
y	Integer	32	Y	The vertical coordinate for the pattern corresponding to the current gray-scale value.
HSKIP	Bitmap			Skip mask. HSKIP is **HGW** by **HGH** pixels. *
HBPP	Integer	32	N	The number of bits per value in the array of gray-scale values.
GI	Array			Array of gray-scale values. GI is a **HGW** by **HGH** array, each entry of which is a HBPP bits unsigned integer.

*Unused if **HENABLESKIP** $=$ **0**.

bitmap. The halftone grid origin is specified by parameters **HGX** and **HGY**. The grid period is defined by parameters **HRX** and **HRY** (see Fig. 24). **HGX**, **HGY**, **HRX** and **HRY** are scaled by 256, which means that the grid origin and grid period have a fractional part of 8 bits.

The possible halftone patterns are given in a dictionary. The identity of a pattern is specified by an index which will usually represent the gray-scale value of the pattern.

NOTE 1 — We use the term gray-scale value for the index to illustrate the compression idea. There is no requirement in this specification that the index does indeed correspond to the gray-scale value.

The result of decoding a halftone bitmap is the bitmap that is produced by the following steps.

1. Fill a bitmap HTREG, of the size given by **HBW** and **HBH**, with the **HDEFPIXEL** value.

2. If **HENABLESKIP** equals **1**, compute a bitmap HSKIP as shown in 6.6.5.1.

3. Set HBPP to $\lceil \log_2(\textbf{HNUMPATS}) \rceil$.

4. Decode an image GI of size **HGW** by **HGH** with **HBPP** bits per pixel using the gray-scale image decoding procedure as described in Annex C. Set the parameters to this decoding procedure as shown in Table 22.

Table 22 — Parameters used to decode a halftone region's gray-scale value array.

Name	Value
GSMMR	**HMMR**
GSW	**HGW**
GSH	**HGH**
GSBPP	**HBPP**
GSUSESKIP	**HENABLESKIP**
GSSKIP	HSKIP
GSTEMPLATE	**HTEMPLATE**

* If HENABLESKIP $=$ **1** then this parameter has no value

Let GI be the results of invoking this decoding procedure.

5. Place sequentially the patterns corresponding to the values in GI into HTREG by the procedure described in 6.6.5.2. The rendering procedure is illustrated in Figure 24. The outline of two halftone patterns are marked by dotted boxes.

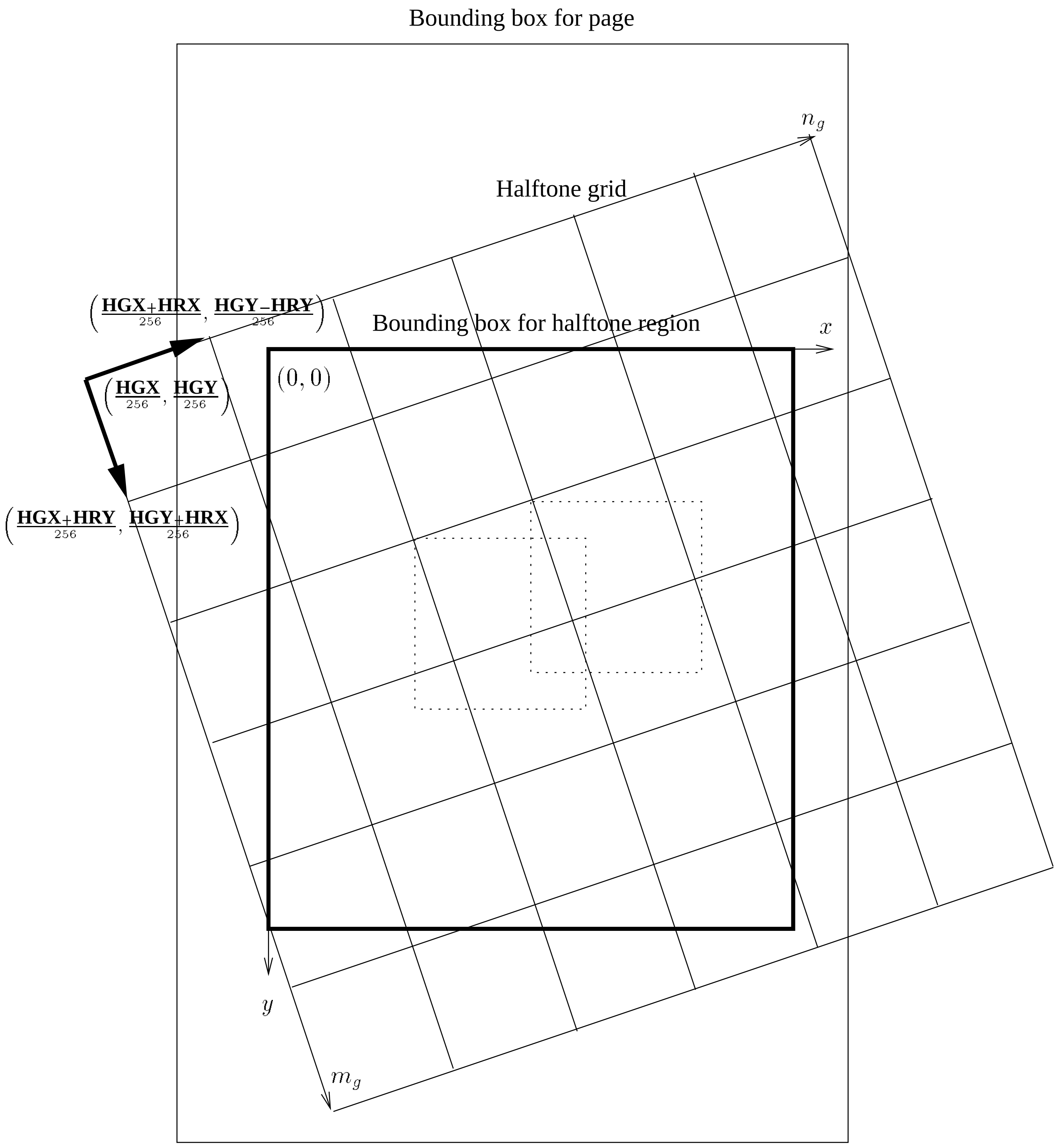

Bounding box for page
n_g
Halftone grid
$\left(\frac{\mathbf{HGX+HRX}}{256}, \frac{\mathbf{HGY-HRY}}{256}\right)$
Bounding box for halftone region
x
$(0,0)$
$\left(\frac{\mathbf{HGX}}{256}, \frac{\mathbf{HGY}}{256}\right)$
$\left(\frac{\mathbf{HGX+HRY}}{256}, \frac{\mathbf{HGY+HRX}}{256}\right)$
y
m_g

Figure 24 — Specification of coordinate systems and grid parameters.

6. After all the patterns have been placed on the bitmap, the current contents of the halftone-coded bitmap are the results that shall be obtained by every decoder, whether it performs this exact sequence of steps or not.

NOTE 2 — If **HGX** is 0, **HGY** is 0, **HRX** is equal to $\boldsymbol{HPW} \times 256$ and **HRY** is 0, then the grid is simple: it is axis-aligned, the primary direction is horizontal, and the grid step is equal to the size of the halftone patterns. In this case, it is possible to optimise the drawing process, as none of the halftone patterns can overlap.

6.6.5.1 Computing HSKIP

The bitmap HSKIP contains **1** at a pixel if drawing a pattern at the corresponding location on the halftone grid does not affect any pixels of HTREG. It is computed as follows.

1. For each value of m_g between 0 and **HGH** $- 1$, beginning from 0, perform the following steps.
   (a) For each value of n_g between 0 and **HGW** $- 1$, beginning from 0, perform the following steps.
      i. Set
      $$\begin{aligned} x &= (\mathbf{HGX} + m_g \times \mathbf{HRY} + n_g \times \mathbf{HRX}) >>_A 8 \\ y &= (\mathbf{HGY} + m_g \times \mathbf{HRX} - n_g \times \mathbf{HRY}) >>_A 8 \end{aligned}$$
      ii. If $((x + \mathbf{HPW} \leq 0)$ OR $(x \geq \mathbf{HBW})$ OR $(y + \mathbf{HPH} \leq 0)$ OR $(y \geq \mathbf{HBH}))$ then set
      $$\text{HSKIP}[m_g, n_g] = \mathbf{1}$$
      Otherwise, set
      $$\text{HSKIP}[m_g, n_g] = \mathbf{0}$$

6.6.5.2 Rendering the halftone patterns

Draw the halftone patterns into HTREG using the following procedure.

1. For each value of m_g between 0 and **HGH** $- 1$, beginning from 0, perform the following steps.
   (a) For each value of n_g between 0 and **HGW** $- 1$, beginning from 0, perform the following steps.
      i. Set
      $$\begin{aligned} x &= (\mathbf{HGX} + m_g \times \mathbf{HRY} + n_g \times \mathbf{HRX}) >>_A 8 \\ y &= (\mathbf{HGY} + m_g \times \mathbf{HRX} - n_g \times \mathbf{HRY}) >>_A 8 \end{aligned}$$
      ii. Draw the halftone pattern **HPATS**[GI$[m_g, n_g]$] into HTREG such that its upper left pixel is at location (x, y) in HTREG.
      A halftone pattern is drawn into HTREG as follows. Each pixel of the pattern shall be combined with the current value of the corresponding pixel in the halftone-coded bitmap, using the combination operator specified by **HCOMBOP**. The results of each combination shall be written into that pixel in the halftone-coded bitmap.
      If any part of a decoded halftone pattern, when placed at location (x, y) lies outside the actual halftone-coded bitmap, then this part of the pattern shall be ignored in the process of combining the halftone pattern with the bitmap.

NOTE — The gray-scale image can be used by the decoder to get a good rendition of the halftone on a multi-level output device of limited spatial resolution such as a computer screen. The use of the gray-scale image for such purposes is outside the scope of this specification.

The gray-scale image is coded by bit-plane coding so the decoder will receive the gray-scale image progressively. Consequently, the decoder may render a halftoned image using the quantized gray-scale values as indices. Such intermediate halftoned images shall not influence the final halftone-coded bitmap.

6.7 Halftone Dictionary Decoding Procedure

6.7.1 General Description

This decoding procedure is used to decode a set of fixed-size halftone patterns; these bitmaps can then be used by halftone region decoding procedures.

6.7.2 Input parameters

The parameters to this decoding procedure are shown in Table 23.

Table 23 — Parameters for the halftone dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
HDMMR	Integer	1	N	Whether MMR is used.
HDPW	Integer	32	N	The width of each halftone pattern.
HDPH	Integer	32	N	The height of each halftone pattern.
GRAYMAX	Integer	32	N	The largest gray-scale value for which a pattern is given.
HDTEMPLATE	Integer	2	N	The template used to code the halftone patterns. *

*Unused if **HDMMR** $=$ **1**.

6.7.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table 24.

Table 24 — Return values from the halftone dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
HDPATS	Array of halftone patterns			The patterns exported by this halftone dictionary. Contains **GRAYMAX** $+ 1$ halftone patterns.

6.7.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 25.

Table 25 — Variables used in the halftone dictionary decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GRAY	Integer	32	N	Gray-scale index.
B_{HDC}	Bitmap			The dictionary collective bitmap.
B_S	Bitmap			A bitmap of size **HDPW** by **HDPH**.

6.7.5 Decoding the Halftone Dictionary

The result of decoding a halftone dictionary is a set of halftone patterns: HDPATS[0] $\cdots$ HDPATS[GRAYMAX]. These halftone patterns shall be the patterns produced by the following steps.

1. Create a bitmap B_{HDC}. The height of this bitmap is **HDPH**. The width of the bitmap is $($**GRAYMAX** $+ 1) \times$ **HDPW**. This bitmap contains all the halftone patterns concatenated left to right.

2. Decode the collective bitmap using a generic region decoding procedure as described in 6.2. Set the parameters to this decoder as shown in Table 26.

Table 26 — Parameters used to decode a halftone dictionary's collective bitmap.

Name	Value
MMR	**HDMMR**
GBW	$($**GRAYMAX** $+ 1) \times$ **HDPW**
GBH	**HDPH**
GBTEMPLATE	**HDTEMPLATE** *
USESKIP	**0**
GBATX$_1$	**-HPW** *
GBATY$_1$	0 *
GBATX$_2$	-3 **
GBATY$_2$	-1 **
GBATX$_3$	2 **
GBATY$_3$	-2 **
GBATX$_4$	-2 **
GBATY$_4$	-2 **

* If **HDMMR** $=$ **1** then this parameter has no value.
** If **HDMMR** $=$ **1** or **HDTEMPLATE** $\neq 0$ then this parameter has no value.

3. Set

$$\text{GRAY} = 0$$

4. While GRAY $\leq$ **GRAYMAX**,

   (a) Let the subimage of B_{HDC} consisting of **HPH** rows and columns **HPW** $\times$ GRAY through **HPW** $\times$ $(\text{GRAY} + 1) - 1$ be denoted B_S. Set

   $$\text{HDPATS}[\text{GRAY}] = B_S$$

   (b) Set

   $$\text{GRAY} = \text{GRAY} + 1$$

7 Control Decoding Procedure

7.1 General description

This decoding procedure controls the invocation of all the other decoding procedure. The encoded bitstream consists of a collection of segments, each containing a part of the data necessary for decoding. There are several different types of segments.

A segment has two parts: a segment header part and a segment data part. All types of segments use a common format for the segment header, but different formats for segment data.

Some segments give information about the structure of the document: start of page, end of page, and so on. Some segments code regions, used in turn to produce the decoded image of a certain page. Some segments ("dictionary segments") do neither, but instead define resources that can be used by segments that code regions.

A segment can be associated with some page, or not associated with any page. A segment can refer to other, preceding, segments. A segment also includes retention bits for the segment that it refers to, and for itself; these indicate when the decoder may discard the data created by decoding a segment.

EXAMPLE — A symbol region segment may make use of symbols defined in preceding symbol dictionary segments. This is indicated by the symbol region's segment header including references to those symbol dictionary segments.

The format of segment headers is described in 7.2. The types of segments are defined in 7.3. The syntax of each type of segment is defined in 7.4.

In the following, some references are made to "preceding" and "following" segments (and other indications implying an order of segments). These terms are defined with reference to the order imposed on the segments by their segment numbers: a segment precedes all segments whose segment numbers are larger than its segment number.

A segment's header part always begins and ends on a byte boundary.

A segment's data part always begins and ends on a byte boundary. Any unused bits in the final byte of a segment shall contain **0**, and shall not be examined by the decoder.

The segment header part and the segment data part of a segment need not occur contiguously in the bitstream being decoded. See G for an organisation where the segment header part of a segment may be stored at some distance from the segment data part of that segment.

This clause contains figures that describe various parts of the encoded data, such as Figures 25 and 31. These conventions used in these figures are

- The first byte encountered in the bitstream is at the left end.
- Fields whose sizes are fixed, and that are always present, are outlined with narrow lines.
- Fields whose sizes are not fixed, or that are not present in all cases, or whose structures are fully described elsewhere, are outlined with heavy lines.
- Some figures (such as Figure 25) are divided into fields, each of which is an integral number of bytes long. In these figures, hash marks extending down from the top of the figure denote byte boundaries, and fields are separated by lines running the full height of the figure.
- The remaining figures are divided into fields, each of which is an integral number of bits long, making up an integral number of bytes. In these figures, short hash marks extending up from the bottom of the figure show bit boundaries. Fields are separated by longer hash marks extending up from the bottom of the figure. Each bit's number is shown below the figure.

7.2 Segment header syntax

7.2.1 Segment header fields

A segment header contains the fields shown in Figure 25 and described below.

Segment number See 7.2.2.

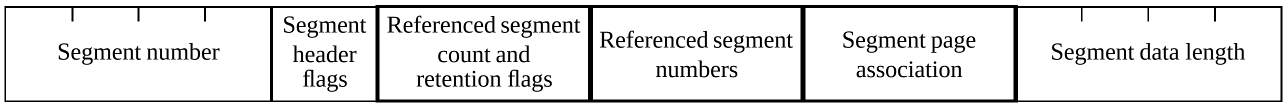


Figure 25 — Segment header structure

Segment header flags See 7.2.3.

Referenced segment count and retention flags See 7.2.4.

Referenced segment number fields See 7.2.5.

Segment page association See 7.2.6.

Segment data length See 7.2.7.

7.2.2 Segment number

This four-byte field contains the segment's segment number.

7.2.3 Segment header flags

This is a 1-byte field. The bits that are defined are shown in Figure 26 and are described below.

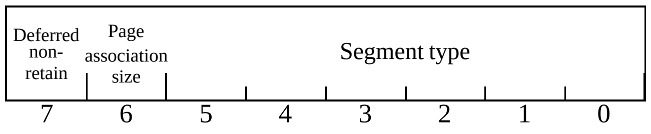


Figure 26 — Segment header flags

Bits 0–5 Segment type. See 7.3.

Bit 6 Page association field size. See 7.2.6.

Bit 7 Deferred non-retain. If this bit is **1**, this segment is flagged as retained only by itself and its attached extension segments, and is flagged as non-retained by the last attached extension segments. An extension segment is an attached extension segment when it refers to only one segment, and the only segments (if any) between it and that referred-to segment are other extension segments also referring only to that referred-to segment.

NOTE — The intention of this bit is to indicate to the decoder that the segment is only referred to by a small number of extension segments. The decoder may take some expensive actions when segments are flagged as retained, but if this retention is only for the benefit of the segment's attached extension segments, these actions may not be necessary. Knowing this in advance is helpful.

7.2.4 Referenced segment count and retention flags

This field contains one or more bytes indicating how many other segments are referenced by this segment, and which segments contain data that is needed after this segment.

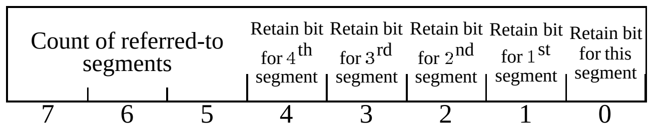
NOTE — The decoder's memory requirements can be reduced by letting it know when it is allowed to forget about the data represented by some previous segment.

The number of bytes in this field depends on the number of segments referred to by this segment. If this segment refers to four or fewer segments, then this field is one byte long. If this segment refers to more than four segments, then this field is $4 + \lceil (\mathrm{R} + 1)/8 \rceil$ bytes long where R is the number of segments that this segment refers to.

EXAMPLE — If this segment refers to between five and seven other segments, then the field is five bytes long; if it refers to between eight and fifteen other segments, then the field is six bytes long.

The three most significant bits of the first byte in this field determine the length of the field. If the value of this three-bit subfield is between 0 and 4, then the field is one byte long. If the value of this three-bit subfield is 7, then the field is at least five bytes long. This three-bit subfield shall not contain values of 5 and 6.

In the case where the field is one byte long, that byte is formatted as shown in Figure 27 and as described below.



Count of referred-to segments
Retain bit for 4th segment
Retain bit for 3rd segment
Retain bit for 2nd segment
Retain bit for 1st segment
Retain bit for this segment
7
6
5
4
3
2
1
0

Figure 27 — Referenced segment count and retention flags — short form

Bit 0 Retain bit for this segment.

Bit 1 Retain bit for the first referred-to segment. If this segment refers to no other segments, this field shall contain **0**.

Bit 2 Retain bit for the second referred-to segment. If this segment refers to fewer than two other segments, this field shall contain **0**.

Bit 3 Retain bit for the third referred-to segment. If this segment refers to fewer than three other segments, this field shall contain **0**.

Bit 4 Retain bit for the fourth referred-to segment. If this segment refers to fewer than four other segments, this field shall contain **0**.

Bits 5–7 Count of referred-to segments. This field may take on values between zero and four. This specifies the number of segments that this segment refers to.

In the case where the field is in the long format (at least five bytes long), it is composed of an initial four-byte field, followed by a succession of one-byte fields. The initial four-byte field is formatted as follows.

Bits 0–28 Count of referred-to segments. This specifies the number of segments that this segment refers to.

Bits 29–31 Indication of long-form format. This field shall contain the value 7.

The first one-byte field following the initial four-byte field is formatted as follows.

Bit 0 Retain bit for this segment.

Bit 1 Retain bit for the first referred-to segment.

Bit 2 Retain bit for the second referred-to segment.

Bit 3 Retain bit for the third referred-to segment.

Bit 4 Retain bit for the fourth referred-to segment.

Bit 5 Retain bit for the fifth referred-to segment. If this segment refers to fewer than five other segments, this field shall contain **0**.

Bit 6 Retain bit for the sixth referred-to segment. If this segment refers to fewer than six other segments, this field shall contain **0**.

Bit 7 Retain bit for the seventh referred-to segment. If this segment refers to fewer than seven other segments, this field shall contain **0**.

The second one-byte field, if present, contains retain bits for the eighth through fifteenth referred-to segments; the bits corresponding to any segments beyond the count of segments actually referred to shall be **0**. Succeeding one-byte fields are formatted similarly.

If the retain bit for this segment value is **0**, then no segment may refer to this segment.

If the retain bit for the first referred-to segment value is **0**, then no segment after this one may refer to the first segment that this segment refers to (i.e., this segment is the last segment that refers to that other segment). Further retain bit values have similar meanings: if the retain bit for the Kth referred-to segment value is **0**, then no segment after this one may refer to the Kth segment that this segment refers to.

7.2.5 Referred-to segment numbers

This field contains the segment numbers of the segments that this segment refers to, if any. The number of values in this field is determined by the Referenced segment count and retention flags field. Each value is the segment number of a segment that this segment refers to. A segment may refer to only segments with lower segment numbers. When the current segment's number is 256 or less, then each referred-to segment number is one byte long. Otherwise, when the current segment's number is 65536 or less, each referred-to segment number is two bytes long. Otherwise, each referred-to segment number is four bytes long.

7.2.6 Segment page association

This field encodes the number of the page to which this segment belongs. The first page shall be numbered "1". This field may contain a value of zero; this value indicates that this segment is not associated with any page.

A segment that has a non-zero segment page association may only be referred to by segments having the same segment page association value as it.

This field is one byte long if this segment's page association field size flag bit is **0**, and is four bytes long if this segment's page association field size flag bit is **1**.

NOTE — Most documents have fewer than 256 pages, so this field has a short form that can hold values from 0 to 255 in a single byte. The page association field for unassociated segments can also be only a single byte long.

7.2.7 Segment data length

This 4-byte field contains the length of the segment's segment data part, in bytes.

If the segment's type is "Immediate generic region", then the length field may contain the value `0xFFFFFFFF`. This value is intended to mean that the length of the segment's data part is unknown at the time that the segment header is written (for example in a streaming application such as facsimile). In this case, the true length of the segment's data part must be determined through examination of the data: if the segment uses template-based arithmetic coding, then the segment's data part ends with the two-byte sequence `0xFF 0xAC` followed by a four-byte row count. If the segment uses MMR coding, then the segment's data part ends with the two-byte sequence `0x00 0x00` followed by a four-byte row count. The form of encoding used by the segment may be determined by examining the eighteenth byte of its segment data part, and the end sequences can occur anywhere after that eighteenth byte.

NOTE — Given a list of segment headers in the random-access organisation (see Figure G.2), a decoder can build a map of the rest of the file by knowing the length of the data associated with each segment. This allows it to perform random access.

7.2.8 Segment header example

EXAMPLE 1 — A segment header consisting of the sequence of bytes

```
0x00 0x00 0x00 0x20 0x86 0x65 0x02 0x1e 0x05 0x04
```

is parsed as follows

`0x00 0x00 0x00 0x20` This segment's number is `0x00000020`, or 32 decimal.

`0x86` This segment's type is 6. Its page association field is one byte long. It is retained by only its attached extension segments.

`0x6b` This segment refers to three other segments. It is referred to by some other segment. This is the last reference to the second of the three segments that it refers to.

`0x02 0x1e 0x05` The three segments that it refers to are numbers 2, 30, and 5.

`0x04` This segment is associated with page number 4.

EXAMPLE 2 — A segment header consisting of the sequence of bytes

```
0x00 0x00 0x02 0x34 0x40 0xe0 0x00 0x00 0x09 0x02 0xfd
0x01 0x00 0x00 0x02 0x00 0x1e 0x00 0x05 0x02 0x00 0x02
0x01 0x02 0x02 0x02 0x03 0x02 0x04 0x00 0x00 0x04 0x01
```

is parsed as follows

`0x00 0x00 0x02 0x34` This segment's number is `0x00000234`, or 564 decimal.

`0x40` This segment's type is 0. Its page association field is four bytes long.

`0xe0 0x00 0x00 0x09` This segment's referenced segment count field is in the long format. This segment refers to nine other segments.

`0x02 0xfd` This segment is referred to by some other segment. This is the last reference to the first and eighth of the nine segments that it refers to.

`0x01 0x00 ... 0x02 0x04` The nine segments that it refers to are each identified by two bytes, since this segment's number is between 256 and 65535. The segments that it refers to are, in decimal, numbers 256, 2, 30, 5, 512, 513, 514, 515, and 516.

`0x00 0x00 0x04 0x01` This segment is associated with page number 1025.

7.3 Segment types

Each segment has a certain type. This type specifies the type of the data associated with the segment. This type restricts which other segments it may refer to, and which other segments may refer to it. These restrictions are detailed in 7.3.1.

The segment type is a number between 0 and 63, inclusive. Not all values are allowed. The allowed list of segment types, their full names, and where their formats are defined, are:

0 Symbol dictionary — see 7.4.2.

4 Intermediate symbol region — see 7.4.3.

6 Immediate symbol region — see 7.4.3.

7 Immediate lossless symbol region — see 7.4.3.

16 Halftone dictionary — see 7.4.4.

20 Intermediate halftone region — see 7.4.5.

22 Immediate halftone region — see 7.4.5.

23 Immediate lossless halftone region — see 7.4.5.

36 Intermediate generic region — see 7.4.6.

38 Immediate generic region — see 7.4.6.

39 Immediate lossless generic region — see 7.4.6.

40 Intermediate generic refinement region — see 7.4.7.

42 Immediate generic refinement region — see 7.4.7.

43 Immediate lossless generic refinement region — see 7.4.7.

48 Page information — see 7.4.8.

49 End of page — see 7.4.9.

50 End of stripe — see 7.4.10.

51 End of file — see 7.4.11.

52 Supported profiles — see 7.4.12.

53 Tables — see 7.4.13.

62 Extension — see 7.4.14.

All other segment types are reserved and shall not be used.

NOTE — These segment numbers are allocated according to the following rules. The three high-order bits (bits 4–6) of this number specify the primary type of the segment, and the four low-order (bits 0–3) bits specify the secondary type of the segment.

The primary types are:

0 Symbol bitmap data

1 Halftone bitmap data

2 Generic bitmap data

3 Metadata

Primary types 0–2 are collectively referred to as region types.

For the region types, the interpretation of the four low-order bits is

Bit 0 If this bit is **1**, it indicates that the segment makes some component of the page lossless.

Bit 1 If this bit is **1**, it indicates that the segment can be drawn immediately into the page bitmap. If this bit is **0**, it indicates that the segment is an intermediate segment. See 8.2.

Bits 2–3 These two bits define a subtype of the primary type:

- **0** Dictionary
- **1** Direct Region
- **2** Refinement Region

For metadata, the interpretations of the four low-order bits are:

0 Page information

1 End of page

2 End of stripe

3 End of file

4 Supported profiles

5 Tables

6–13 Reserved

14 Extension

15 Reserved

The segments of types “intermediate symbol region”, “immediate symbol region”, “immediate lossless symbol region”, “intermediate halftone region”, “immediate halftone region”, “immediate lossless halftone region”, “intermediate generic region”, “immediate generic region”, “immediate lossless generic region”, “intermediate generic refinement region”, “immediate generic refinement region”, and “immediate lossless generic refinement region” are collectively referred to as “region segments”.

The segments of types “intermediate symbol region”, “immediate symbol region”, “immediate lossless symbol region”, “intermediate halftone region”, “immediate halftone region”, “immediate lossless halftone region”, “intermediate generic region”, “immediate generic region”, and “immediate lossless generic region”, are collectively referred to as “direct region segments”.

The segments of types “intermediate symbol region”, “intermediate halftone region”, “intermediate generic region”, and “intermediate generic refinement region” are collectively referred to as “intermediate region segments”.

The segments of types “immediate symbol region”, “immediate lossless symbol region”, “immediate halftone region”, “immediate lossless halftone region”, “immediate generic region”, “immediate lossless generic region”, “immediate generic refinement region”, and “immediate lossless generic refinement region” are collectively referred to as “immediate region segments”.

The segments of types “intermediate generic refinement region”, “immediate generic refinement region” and “immediate lossless generic refinement region” are collectively referred to as “refinement region segments”. .

7.3.1 Rules for segment references

The rules for segment references are as follows.

- An intermediate region segment may only be referred to by one other non-extension segment; it may be referred to by any number of extension segments.
- A segment of type “symbol dictionary” (type 0) may refer to any number of segments of type “symbol dictionary” and to up to four segments of type “tables”.
- A segment of type “intermediate symbol region”, “immediate symbol region” or “immediate lossless symbol region” (type 4, 6 or 7) may refer to any number of segments of type “symbol dictionary” and to up to eight segments of type “tables”.
- A segment of type “halftone dictionary” (type 16) may not refer to any other segment.
- A segment of type “intermediate halftone region”, “immediate halftone region” or “immediate lossless halftone region” (type 20, 22 or 23) shall refer to exactly one segment, and this segment shall be of type “halftone dictionary”.
- A segment of type “intermediate generic region”, “immediate generic region” or “immediate lossless generic region” (types 36, 38 or 37) may not refer to any other segment.
- A segment of type “intermediate generic refinement region” (type 40) shall refer to exactly one other segment. This other segment shall be an intermediate region segment.
- A segment of type “immediate generic refinement region” or “immediate lossless generic refinement region” (type 42 or 43) may refer to either zero other segments or exactly one other segment. If it refers to one other segment then that segment shall be an intermediate region segment.
- A segment of type “page information” (type 48) may not refer to any other segments.
- A segment of type “end of page” (type 49) may not refer to any other segments.
- A segment of type “end of stripe” (type 50) may not refer to any other segments.
- A segment of type “end of file (type 51) may not refer to any other segments.
- A segment of type “supported profiles” (type 52) may not refer to any other segments.
- A segment of type “tables” (type 53) may not refer to any other segments.
- A segment of type “extension” (type 62) may refer to any number of segments of any type, unless the extension segment’s type imposes some restriction.

7.3.2 Rules for page associations

Every region segment shall be associated with some page (i.e., have a non-zero page association field). “Page information”, “end of page” and “end of stripe” segments shall be associated with some page. “End of file” segments may not be associated with any page. Segments of other types may be associated with a page or not.

If a segment is not associated with any page, then it may not refer to any segment that is associated with any page.

If a segment is associated with a page, then it may refer to segments that are not associated with any page, and to segments that are associated with the same page. It may not refer to any segment that is associated with a different page.

7.4 Segment syntaxes

This section describes in detail the syntax of the segment data part of each type of segment, and how it is to be decoded.

7.4.1 Region segment data header

Every region segment’s data part begins with a region segment data header; its format is specified here. A region segment data header contains the following fields, as shown in Figure 28 and as described below.

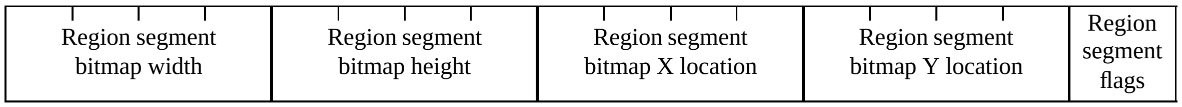

Region segment bitmap width
Region segment bitmap height
Region segment bitmap X location
Region segment bitmap Y location
Region segment flags

Figure 28 — Region segment data header structure

Region segment bitmap width See 7.4.1.1.

Region segment bitmap height See 7.4.1.2.

Region segment bitmap X location See 7.4.1.3.

Region segment bitmap Y location See 7.4.1.4.

Region segment flags See 7.4.1.5

7.4.1.1 Region segment bitmap width

This four-byte field gives the width in pixels of the bitmap encoded in this segment.

7.4.1.2 Region segment bitmap height

This four-byte field gives the height in pixels of the bitmap encoded in this segment.

7.4.1.3 Region segment bitmap X location

This four-byte field gives the horizontal offset in pixels of the bitmap encoded in this segment relative to the page bitmap.

7.4.1.4 Region segment bitmap Y location

This four-byte field gives the vertical offset in pixels of the bitmap encoded in this segment relative to the page bitmap.

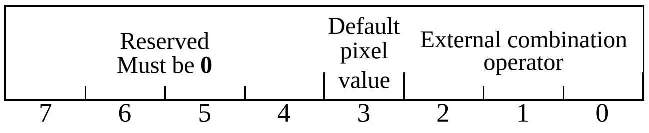


Figure 29 — Region segment flags field structure

7.4.1.5 Region segment flags

This one-byte field is formatted as shown in Figure 29 and as described below.

Bits 0–2 External combination operator. This three-bit field can take on the following values, representing one of five possible combination operators:

0 OR

1 AND

2 XOR

3 XNOR

4 REPLACE

NOTE — These operators describe how the segment's bitmap is to be combined with the page bitmap. REPLACE is intended to be used by refinement regions, where the refined region replaces the region it's refining. Operators such as AND can be used for masking, where a portion of the page bitmap that already contains data is to be cleared so that another bitmap can be written there — think of writing a bitmap through a mask.

Bit 3 Segment default pixel value.

Bits 4–7 Reserved; shall be **0**.

In other words, this region segment data header describes the size and location of the bitmap encoded in this segment.

EXAMPLE — If the size and location values are (in order) 100, 200, 50 and 75, then this segment describes a bitmap 100 pixels wide, 200 pixels high, whose top left corner is 50 pixels to the right of, and 75 pixels below, the page's top left corner.

7.4.2 Symbol dictionary segment syntax

7.4.2.1 Symbol dictionary data header

A symbol dictionary segment's data part begins with a symbol dictionary data header, containing the fields shown in Figure 30 and described below.

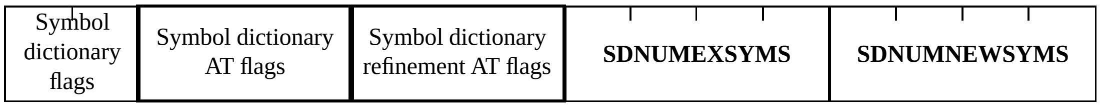

Symbol dictionary flags
Symbol dictionary AT flags
Symbol dictionary refinement AT flags
SDNUMEXSYMS
SDNUMNEWSYMS

Figure 30 — Symbol dictionary data header structure

Symbol dictionary flags See 7.4.2.1.1.

Symbol dictionary AT flags See 7.4.2.1.2.

Symbol dictionary refinement AT flags See 7.4.2.1.3.

SDNUMEXSYMS See 7.4.2.1.4.

SDNUMNEWSYMS See 7.4.2.1.5.

7.4.2.1.1 Symbol dictionary flags

This two-byte field is formatted as shown in Figure 31 and as described below.

Reserved Must be **0**			**SDRTEMPLATE**	**SDTEMPLATE**		Bitmap coding context retained	Bitmap coding context used	**SDHUFFAGGINST** selection	**SDHUFFBMSIZE** selection	**SDHUFFDW** selection		**SDHUFFDH** selection		**SDREFAGG**	**SDHUFF**
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Figure 31 — Symbol dictionary flags field structure

Bit 0 SDHUFF

If this bit is **1**, then the segment uses the Huffman encoding variant. If this bit is **0**, then the segment uses the arithmetic encoding variant. The setting of this flag determines how the data in this segment are encoded, and may also modify the order in which some of the data are encoded.

Bit 1 SDREFAGG

If this bit is **0**, then no refinement or aggregate coding is used in this segment. If this bit is **1**, then every symbol bitmap is refinement/aggregate coded.

Bits 2–3 SDHUFFDH selection. This two-bit field can take on one of three values, indicating which table is to be used for **SDHUFFDH**.

0	Table B.4
1	Table B.5
3	User-supplied table

The value 2 is not permitted.

If **SDHUFF** is **0** then this field must contain the value 0.

Bits 4–5 SDHUFFDW selection. This two-bit field can take on one of three values, indicating which table is to be used for **SDHUFFDW**.

0	Table B.2
1	Table B.3
3	User-supplied table

The value 2 is not permitted.

If **SDHUFF** is **0** then this field must contain the value 0.

Bit 6 SDHUFFBMSIZE selection.

If this field is **0** then Table B.1 is used for **SDHUFFBMSIZE**. If this field is **1** then a user-supplied table is used for **SDHUFFBMSIZE**.

If **SDHUFF** is **0** then this field must contain the value **0**.

Bit 7 SDHUFFAGGINST selection.

If this field is **0** then Table B.1 is used for **SDHUFFAGGINST**. If this field is **1** then a user-supplied table is used for **SDHUFFAGGINST**.

If **SDHUFF** is **0** then this field must contain the value **0**.

Bit 8 Bitmap coding context used.

If **SDHUFF** is **1** and **SDREFAGG** is **0** then this field must contain the value **0**.

Bit 9 Bitmap coding context retained.

If **SDHUFF** is **1** and **SDREFAGG** is **0** then this field must contain the value **0**.

Bits 10–11 SDTEMPLATE

This field controls the template used to decode symbol bitmaps if **SDHUFF** is **0**. If **SDHUFF** is **1**, this field must contain the value 0.

Bit 12 SDRTEMPLATE

This field controls the template used to decode symbol bitmaps if **SDREFAGG** is **1**. If **SDREFAGG** is **0**, this field must contain the value **0**.

Bits 13–15 Reserved; must be **0**.

7.4.2.1.2 Symbol dictionary AT flags

This field is only present if **SDHUFF** is **0**. If **SDTEMPLATE** is 0, it is an eight-byte field, formatted as shown in Figure 32 and as described below.

SDATX$_1$	**SDATY$_1$**	**SDATX$_2$**	**SDATY$_2$**	**SDATX$_3$**	**SDATY$_3$**	**SDATX$_4$**	**SDATY$_4$**

Figure 32 — Symbol dictionary AT flags field structure when SDTEMPLATE is 0

Byte 0 SDATX$_1$

Byte 1 SDATY$_1$

Byte 2 SDATX$_2$

Byte 3 SDATY$_2$

Byte 4 SDATX$_3$

Byte 5 SDATY$_3$

Byte 6 SDATX$_4$

Byte 7 SDATY$_4$

If **SDTEMPLATE** is 1, 2 or 3, it is a two-byte field formatted as shown in Figure 33 and as described below.

SDATX$_1$	**SDATY$_1$**

Figure 33 — Symbol dictionary AT flags field structure when SDTEMPLATE is not 0

Byte 0 SDATX$_1$

Byte 1 SDATY$_1$

If **SDTEMPLATE** is 1, 2 or 3 then the values of **SDATX$_2$** through **SDATX$_4$** and **SDATY$_2$** through **SDATY$_4$** are all zero.

The AT coordinate X and Y fields are signed values, and may take on values that are permitted according to Figure 7.

SDRATX$_1$	**SDRATY$_1$**	**SDRATX$_2$**	**SDRATY$_2$**

Figure 34 — Symbol dictionary refinement AT flags field structure

7.4.2.1.3 Symbol dictionary refinement AT flags

This field is only present if **SDREFAGG** is **1** and **SDRTEMPLATE** is **0**. It is a four-byte field, formatted as shown in Figure 34 and as described below.

Byte 0 SDRATX$_1$

Byte 1 SDRATY$_1$

Byte 2 SDRATX$_2$

Byte 3 SDRATY$_2$

The AT coordinate X and Y fields are signed values, and may take on values that are permitted according to 6.3.5.3.

7.4.2.1.4 SDNUMEXSYMS

This four-byte field contains the number of symbols exported from this dictionary.

It is very useful for the decoder be able to find out easily how many symbols are present — for example, it might want to allocate an array of structures before beginning to decode the dictionary.

7.4.2.1.5 SDNUMNEWSYMS

This four-byte field contains the number of symbols defined in this dictionary.

NOTE — **SDNUMEXSYMS** and **SDNUMNEWSYMS** are often, but not always, the same value. For example, if a dictionary exports some of the symbols from dictionaries that it references, then these copied symbols are reflected in **SDNUMEXSYMS** but not in **SDNUMNEWSYMS**. Another possible source of difference comes from the possibility that a dictionary defines some symbols that it does not export.

7.4.2.1.6 Symbol dictionary segment Huffman table selection

Set the values of the parameters **SDHUFFDH**, **SDHUFFDW**, **SDHUFFBMSIZE** and **SDHUFFAGGINST** according to the selection fields shown in 7.4.2.1.1, and the tables segments referred to by this segment. More precisely, of these four Huffman tables, some may be specified to use some standard table, and some may be specified to use a user-supplied table. The number specified to use a user-supplied table shall be equal to the number of tables segments referred to by this segment. These tables segments are matched up with the Huffman tables using user-supplied tables according to the order in which the tables segments are referred to, and the order

1. **SDHUFFDH**
2. **SDHUFFDW**
3. **SDHUFFBMSIZE**
4. **SDHUFFAGGINST**

If a user-specified table is used for **SDHUFFDW**, then this table must be capable of coding the out-of-band value OOB. If a user-specified table is used for **SDHUFFDH**, **SDHUFFBMSIZE** or **SDHUFFAGGINST**, then this table must not be capable of coding the out-of-band value OOB.

EXAMPLE — If **SDHUFFDH** and **SDHUFFAGGINST** are specified to use user-supplied tables, and **SDHUFFDW** and **SDHUFFBMSIZE** are specified to use standard tables (Table B.2 and Table B.1 respectively), then this segment must refer to exactly two tables segments; the tables segment that is referred to first is used for **SDHUFFDH** and the tables segment that is referred to second is used for **SDHUFFAGGINST**.

7.4.2.2 Decoding a symbol dictionary segment

A symbol dictionary segment is decoded according to the following steps.

1. Interpret its header, as described in 7.4.2.1.

2. Decode (or retrieve the results of decoding) any referred-to symbol dictionary and tables segments.

3. If the "bitmap coding context used" bit in the header was **1**, then set the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures to the values that they contained at the end of decoding the last-referenced symbol dictionary segment. That symbol dictionary segment's symbol dictionary data header must have had the "bitmap coding context retained" bit equal to **1**.

4. If the "bitmap coding context used" bit in the header was **0**, then reset all the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures to zero.

5. Reset the adaptive probability values for all the contexts of all the arithmetic integer coders to zero.

6. Invoke the symbol dictionary decoding procedure described in 6.5, with the parameters to the symbol dictionary decoding procedure set as shown in Table 27.

Table 27 — Parameters used to decode a symbol dictionary segment.

Name	Value
SDHUFF	As shown in 7.4.2.1.1.
SDREFAGG	As shown in 7.4.2.1.1.
SDNUMINSYMS	The sum of the number of exported symbols in all the symbol dictionary segments referred to by this segment.
SDINSYMS	Concatenate the exported symbol arrays from all the symbol dictionary segments referred to by this segment, in the order in which they are referred to.
SDNUMNEWSYMS	As shown in 7.4.2.1.5.
SDNUMEXSYMS	As shown in 7.4.2.1.4.
SDHUFFDH	See 7.4.2.1.6
SDHUFFDW	See 7.4.2.1.6
SDHUFFBMSIZE	See 7.4.2.1.6
SDHUFFAGGINST	See 7.4.2.1.6
SDTEMPLATE	See 7.4.2.1.1
$\mathbf{SDATX_1}$	See 7.4.2.1.2
$\mathbf{SDATY_1}$	See 7.4.2.1.2
$\mathbf{SDATX_2}$	See 7.4.2.1.2
$\mathbf{SDATY_2}$	See 7.4.2.1.2
$\mathbf{SDATX_3}$	See 7.4.2.1.2
$\mathbf{SDATY_3}$	See 7.4.2.1.2
$\mathbf{SDATX_4}$	See 7.4.2.1.2
$\mathbf{SDATY_4}$	See 7.4.2.1.2
SDRTEMPLATE	See 7.4.2.1.1
$\mathbf{SDRATX_1}$	See 7.4.2.1.3
$\mathbf{SDRATY_1}$	See 7.4.2.1.3
$\mathbf{SDRATX_2}$	See 7.4.2.1.3
$\mathbf{SDRATY_2}$	See 7.4.2.1.3

7. If the “bitmap coding context retained” bit in the header was **1**, then preserve the current contents of the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures.

NOTE — Step 3 is intended to reduce the coding costs of symbol dictionaries. A side-effect of decoding a symbol dictionary is that the adaptive probability values used for coding bitmaps “learn” the approximate statistics. These two steps allow some limited re-use of these statistics: the statistics learned when decoding the symbol dictionary that is the last symbol dictionary referenced are used as a starting point for decoding this symbol dictionary.

Step 7 is explicitly present because not every symbol dictionary’s adaptive probability values will be used by another dictionary. Knowing that they will not be used allows the decoder to discard them, reducing memory usage.

7.4.3 Symbol region segment syntax

The data parts of all three of the symbol region segment types (“intermediate symbol region”, “immediate symbol region” and “immediate lossless symbol region”) are coded identically, but are acted upon differently; see 8.2. The syntax of these segment types’ data parts is specified here.

7.4.3.1 Symbol region segment data header

The data part of a symbol region segment begins with a symbol region segment data header. This header contains the fields shown in Figure 35 and described below.

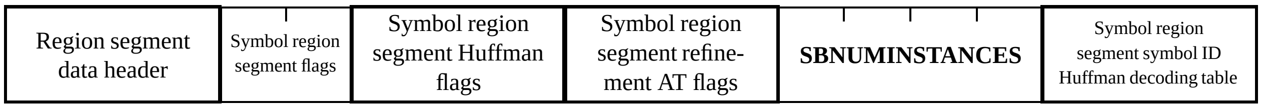

Region segment data header
Symbol region segment flags
Symbol region segment Huffman flags
Symbol region segment refinement AT flags
SBNUMINSTANCES
Symbol region segment symbol ID Huffman decoding table

Figure 35 — Symbol region segment data header structure

Region segment data header See 7.4.1.

Symbol region segment flags See 7.4.3.1.1.

Symbol region segment Huffman flags See 7.4.3.1.2.

Symbol region segment refinement AT flags See 7.4.3.1.3.

SBNUMINSTANCES See 7.4.3.1.4.

Symbol region segment symbol ID Huffman decoding table See 7.4.3.1.5.

7.4.3.1.1 Symbol region segment flags

This two-byte field is formatted as shown in Figure 36 and as described below.

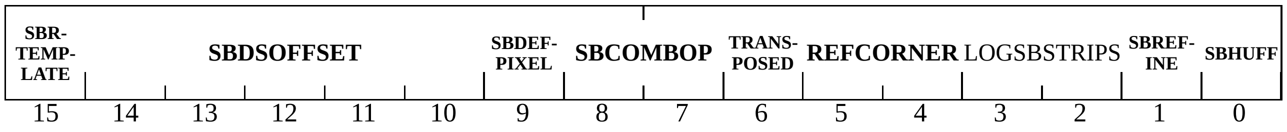


Figure 36 — Symbol region flags field structure

Bit 0 SBHUFF.

If this bit is **1**, then the segment uses the Huffman encoding variant. If this bit is **0**, then the segment uses the arithmetic encoding variant. The setting of this flag determines how the data in this segment are encoded.

Bit 1 SBREFINE.

If this bit is **0**, then the segment contains no symbol instance refinements. If this bit is **1**, then the segment may contain symbol instance refinements.

Bits 2–3 LOGSBSTRIPS.

This two-bit field codes the base-2 logarithm of the strip size used to encode the segment. Thus, strip sizes of 1, 2, 4, and 8 can be encoded.

Bits 4–5 REFCORNER. The four values that this two-bit field can take on are

0 BOTTOMLEFT.

1 TOPLEFT.

2 BOTTOMRIGHT.

3 TOPRIGHT.

NOTE — The best compression is usually achieved when the the reference point of each symbol is on the text baseline. Given that text can run in any of eight directions, there needs to be some flexibility in which corner of a given symbol is used as the reference point.

Bit 6 TRANSPOSED.

If this bit is **1**, then the primary direction of coding is top-to-bottom. If this bit is **0**, then the primary direction of coding is left-to-right. This allows for text running up and down the page.

Bits 7–8 SBCOMBOP. This field has four possible values, representing one of four possible combination operators:

0 OR

1 AND

2 XOR

3 XNOR

Bit 9 SBDEFPIXEL.

This bit contains the value of any pixel that is not covered by any symbol.

Bits 10–14 SBDSOFFSET.

This signed five-bit field contains the value of **SBDSOFFSET** — see 6.4.8.

Bit 15 SBRTEMPLATE

This field controls the template used to decode symbol instance refinements if **SBREFINE** is **1**. If **SBREFINE** is **0**, this field must contain the value **0**.

7.4.3.1.2 Symbol region segment Huffman flags

This field is only present if **SBHUFF** is **1**.

This two-byte field is formatted as shown in Figure 37 and as described below.

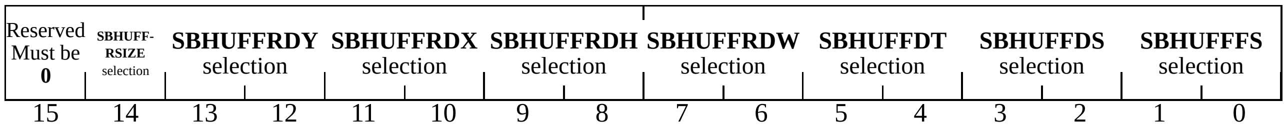


Figure 37 — Symbol region Huffman flags field structure

Bits 0–1 **SBHUFFFS** selection. This two-bit field can take on one of three values, indicating which table is to be used for **SBHUFFFS**.

0 Table B.6

1 Table B.7

3 User-supplied table

The value 2 is not permitted.

Bits 2–3 **SBHUFFDS** selection. This two-bit field can take on one of four values, indicating which table is to be used for **SBHUFFDS**.

0 Table B.8

1 Table B.9

2 Table B.10

3 User-supplied table

Bits 4–5 **SBHUFFDT** selection. This two-bit field can take on one of four values, indicating which table is to be used for **SBHUFFDT**.

0 Table B.11

1 Table B.12

2 Table B.13

3 User-supplied table

Bits 6–7 **SBHUFFRDW** selection. This two-bit field can take on one of three values, indicating which table is to be used for **SBHUFFRDW**.

0 Table B.14

1 Table B.15

3 User-supplied table

The value 2 is not permitted.

Bits 8–9 **SBHUFFRDH** selection. This two-bit field can take on one of three values, indicating which table is to be used for **SBHUFFRDH**.

0 Table B.14

1 Table B.15

3 User-supplied table

The value 2 is not permitted.

Bits 10–11 **SBHUFFRDX** selection. This two-bit field can take on one of three values, indicating which table is to be used for **SBHUFFRDX**.

0 Table B.14

1 Table B.15

3 User-supplied table

The value 2 is not permitted.

Bits 12–13 **SBHUFFRDY** selection. This two-bit field can take on one of three values, indicating which table is to be used for **SBHUFFRDY**.

0 Table B.14

1 Table B.15

3 User-supplied table

The value 2 is not permitted.

Bit 14 **SBHUFFRSIZE** selection. If this field is **0** then Table B.1 is used for **SBHUFFRSIZE**. If this field is **1** then a user-supplied table is used for **SBHUFFRSIZE**.

Bit 15 Reserved.

7.4.3.1.3 Symbol region refinement AT flags

This field is only present if **SBREFINE** is **1** and **SBRTEMPLATE** is **0**. It is a four-byte field, formatted as shown in Figure 38 and as described below.

$SBRATX_1$	**$SBRATY_1$**	**$SBRATX_2$**	**$SBRATY_2$**

Figure 38 — Symbol region refinement AT flags field structure

Byte 0 **$SBRATX_1$**

Byte 1 **$SBRATY_1$**

Byte 2 **$SBRATX_2$**

Byte 3 **$SBRATY_2$**

The AT coordinate X and Y fields are signed values, and may take on values that are permitted according to 6.3.5.3.

7.4.3.1.4 SBNUMINSTANCES

This four-byte field contains the number of symbol instances coded in this segment.

7.4.3.1.5 Symbol region segment symbol ID Huffman decoding table

This field contains a coded version of the Huffman codes used to decode symbol instance IDs in the symbol region decoding procedure. It is decoded as specified in 7.4.3.1.7. It is only present if **SBHUFF** is **1**.

7.4.3.1.6 Symbol region segment Huffman table selection

Set the values of the parameters **SBHUFFFS**, **SBHUFFDS**, **SBHUFFDT**, **SBHUFFRDW**, **SBHUFFRDH**, **SB-HUFFRDX**, **SBHUFFRDY** and **SBHUFFRSIZE** according to the selection fields shown in 7.4.3.1.2, and the tables segments referred to by this segment. More precisely, of these eight Huffman tables, some may be specified to use some standard table, and some may be specified to use a user-supplied table. The number specified to use a user-supplied table shall be equal to the number of tables segments referred to by this segment. These tables segments are matched up with the Huffman tables using user-supplied tables according to the order in which the tables segments are referred to, and the order

1. **SBHUFFFS**
2. **SBHUFFDS**
3. **SBHUFFDT**
4. **SBHUFFRDW**
5. **SBHUFFRDH**
6. **SBHUFFRDX**

7. **SBHUFFRDY**

8. **SBHUFFRSIZE**

If a user-specified table is used for **SBHUFFDS**, then this table shall be capable of coding the out-of-band value OOB. If a user-specified table is used for **SBHUFFFS**, **SBHUFFDT**, **SBHUFFRDW**, **SBHUFFRDH**, **SBHUFFRDX**, **SBHUFFRDY** or **SBHUFFRSIZE** then this table shall not be capable of coding the out-of-band value OOB.

7.4.3.1.7 Symbol ID Huffman table decoding

This table is encoded as **SBNUMSYMS** code lengths; the actual codes in **SBSYMCODES** are assigned from these code lengths using the algorithm in B.2.

The code lengths themselves are run-length coded and the runs Huffman coded. This is very similar to the "zlib" coded format documented in RFC1951, though not identical. The encoding is based on the codes shown in Table 28.

The code lengths for RUNCODE0 through RUNCODE34 are then written out, as four bits each. These lengths are then processed by the algorithm in B.2 to assign Huffman codes for RUNCODE0 through RUNCODE34. Next, further bits are read, decoded into one of the run codes, and interpreted as in Table 28 to produce code lengths for the **SBNUMSYMS** codes. Finally, the remaining bits in the last byte read are discarded, so that the actual symbol region decoding procedure begins on a byte boundary.

EXAMPLE — Suppose that **SBNUMSYMS** is 32 and the code lengths for these 32 symbols are, in order,

0	0	0	9	6	6	6	6	3	4	4	4	4	4	4	0
7	9	8	7	5	5	5	5	5	5	3	6	7	4	7	7

These code lengths might be transmitted as the sequence of bytes, in hexadecimal

```
0x50 0x03 0x35 0x32 0x53 0x00 0x00 0x00 0x00
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x35 0x0F
0x8B 0x30 0x9E 0xB8 0x5F 0x1D 0xD2 0x83 0x00
```

Interpreting this sequence of bytes proceeds as follows.

1. The first 17 bytes plus the first four bits of the 18th byte assign code lengths to the 35 run codes as follows

RUNCODE0	5	RUNCODE1	0	RUNCODE2	0	RUNCODE3	3
RUNCODE4	3	RUNCODE5	5	RUNCODE6	3	RUNCODE7	2
RUNCODE8	5	RUNCODE9	3	RUNCODE10	0	RUNCODE11	0
RUNCODE12	0	RUNCODE13	0	RUNCODE14	0	RUNCODE15	0
RUNCODE16	0	RUNCODE17	0	RUNCODE18	0	RUNCODE19	0
RUNCODE20	0	RUNCODE21	0	RUNCODE22	0	RUNCODE23	0
RUNCODE24	0	RUNCODE25	0	RUNCODE26	0	RUNCODE27	0
RUNCODE28	0	RUNCODE29	0	RUNCODE30	0	RUNCODE31	0
RUNCODE32	3	RUNCODE33	5	RUNCODE34	0		

   Recall that codes that are not used are assigned a code length of zero.

2. The algorithm of B.2 assigns the following Huffman codes to the run codes (run codes that are not assigned Huffman codes are omitted).

RUNCODE0	**11100**	RUNCODE3	**010**	RUNCODE4	**011**
RUNCODE5	**11101**	RUNCODE6	**100**	RUNCODE7	**00**
RUNCODE8	**11110**	RUNCODE9	**101**	RUNCODE32	**110**
RUNCODE33	**11111**				

Table 28 — Meaning of the run codes

RUNCODE0	Code length is 0
RUNCODE1	Code length is 1
RUNCODE2	Code length is 2
RUNCODE3	Code length is 3
RUNCODE4	Code length is 4
RUNCODE5	Code length is 5
RUNCODE6	Code length is 6
RUNCODE7	Code length is 7
RUNCODE8	Code length is 8
RUNCODE9	Code length is 9
RUNCODE10	Code length is 10
RUNCODE11	Code length is 11
RUNCODE12	Code length is 12
RUNCODE13	Code length is 13
RUNCODE14	Code length is 14
RUNCODE15	Code length is 15
RUNCODE16	Code length is 16
RUNCODE17	Code length is 17
RUNCODE18	Code length is 18
RUNCODE19	Code length is 19
RUNCODE20	Code length is 20
RUNCODE21	Code length is 21
RUNCODE22	Code length is 22
RUNCODE23	Code length is 23
RUNCODE24	Code length is 24
RUNCODE25	Code length is 25
RUNCODE26	Code length is 26
RUNCODE27	Code length is 27
RUNCODE28	Code length is 28
RUNCODE29	Code length is 29
RUNCODE30	Code length is 30
RUNCODE31	Code length is 31
RUNCODE32	Copy the previous code length 3–6 times. The next two bits, plus 3, indicate this repeat length.
RUNCODE33	Repeat a code length of 0 for 3–10 times. The next three bits, plus 3, indicate this repeat length.
RUNCODE34	Repeat a code length of 0 for 11–138 times. The next seven bits, plus 11, indicate this repeat length.

3. The remaining part of the byte sequence is

   ```
   0xF 0x8B 0x30 0x9E 0xB8 0x5F 0x1D 0xD2 0x83 0x00
   ```

   where half of the first byte has already been consumed. Decoding this sequence using these Huffman codes proceeds as follows

11111 000	RUNCODE33(0) (i.e., RUNCODE33 followed by three bits containing the value 0)
101	RUNCODE9
100	RUNCODE6
110 00	RUNCODE32(0) (i.e., RUNCODE32 followed by two bits containing the value 0)
010	RUNCODE3
011	RUNCODE4
110 10	RUNCODE32(2)
11100	RUNCODE0
00	RUNCODE7
101	RUNCODE9
11110	RUNCODE8
00	RUNCODE7
11101	RUNCODE5
110 10	RUNCODE32(2)
010	RUNCODE3
100	RUNCODE6
00	RUNCODE7
011	RUNCODE4
00	RUNCODE7
00	RUNCODE7
0000	Four bits of padding to fill the last byte.

4. After interpreting the run codes according to Table 28, the desired sequence of code lengths is decoded.

7.4.3.2 Decoding a symbol region segment

A symbol region segment is decoded according to the following steps.

1. Interpret its header, as described in 7.4.3.1.
2. Decode (or retrieve the results of decoding) any referred-to symbol dictionary and tables segments.
3. Reset all the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures to zero.
4. Reset the adaptive probability values for all the contexts of all the arithmetic integer coders to zero.
5. Invoke the symbol region decoding procedure described in 6.4, with the parameters to the symbol region decoding procedure set as shown in Table 29.

7.4.4 Halftone dictionary segment syntax

7.4.4.1 Halftone dictionary data header

A halftone dictionary segment's data part begins with a halftone dictionary data header, formatted as shown in Figure 39 and as described below.

Halftone dictionary flags See 7.4.4.1.1.

Table 29 — Parameters used to decode a symbol region segment.

Name	Value
SBHUFF	As shown in 7.4.3.1.1.
SBREFINE	As shown in 7.4.3.1.1.
SBDEFPIXEL	As shown in 7.4.3.1.1.
SBCOMBOP	As shown in 7.4.3.1.1.
TRANSPOSED	As shown in 7.4.3.1.1.
REFCORNER	As shown in 7.4.3.1.1.
SBDSOFFSET	As shown in 7.4.3.1.1.
SBW	As specified by the Region segment bitmap width in this segment's region segment data header.
SBH	As specified by the Region segment bitmap height in this segment's region segment data header.
SBNUMINSTANCES	As shown in 7.4.3.1.4.
SBSTRIPS	$2^{\mathrm{LOGSBSTRIPS}}$
SBNUMSYMS	The sum of the number of exported symbols in all the symbol dictionary segments referred to by this segment.
SBSYMCODES	As specified in 7.4.3.1.7.
SBSYMCODELEN	$\lceil \log_2 \textbf{SBNUMSYMS} \rceil$
SBSYMS	Concatenate the exported symbol arrays from all the symbol dictionary segments referred to by this segment, in the order in which they are referred to.
SBHUFFFS	See 7.4.3.1.6
SBHUFFDS	See 7.4.3.1.6
SBHUFFDT	See 7.4.3.1.6
SBHUFFRDW	See 7.4.3.1.6
SBHUFFRDH	See 7.4.3.1.6
SBHUFFRDX	See 7.4.3.1.6
SBHUFFRDY	See 7.4.3.1.6
SBHUFFRSIZE	See 7.4.3.1.6
SBRTEMPLATE	As shown in 7.4.3.1.1
SBRATX$_1$	See 7.4.3.1.3
SBRATY$_1$	See 7.4.3.1.3
SBRATX$_2$	See 7.4.3.1.3
SBRATY$_2$	See 7.4.3.1.3

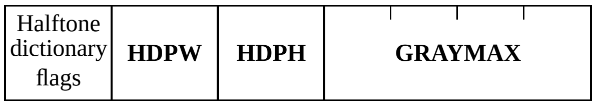


Figure 39 — Halftone dictionary header structure

HDPW See 7.4.4.1.2.

HDPH See 7.4.4.1.3.

GRAYMAX See 7.4.4.1.4.

7.4.4.1.1 Halftone dictionary flags

This one-byte field is formatted as shown in Figure 40 and as described below.

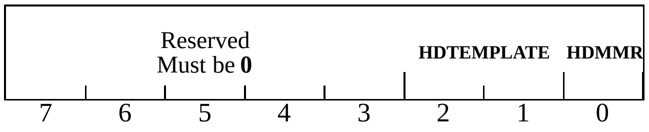


Figure 40 — Halftone dictionary flags field structure

Bit 0 HDMMR

If this bit is **1**, then the segment uses the MMR encoding variant. If this bit is **0**, then the segment uses the arithmetic encoding variant.

Bits 1–2 HDTEMPLATE

This field controls the template used to decode halftone patterns if **HDMMR** is **0**. If **HDMMR** is **1**, this field must contain the value 0.

Bits 3–7 Reserved; must be **0**.

7.4.4.1.2 HDPW

This one-byte field contains the width of the halftone patterns defined in this halftone dictionary. Its value must be greater than zero.

7.4.4.1.3 HDPH

This one-byte field contains the height of the halftone patterns defined in this halftone dictionary. Its value must be greater than zero.

7.4.4.1.4 GRAYMAX

This four-byte field contains one less than the number of halftone patterns defined in this halftone dictionary.

7.4.4.2 Decoding a halftone dictionary segment

A halftone dictionary segment is decoded according to the following steps.

1. Interpret its header, as described in 7.4.4.1.
2. Reset all the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures to zero.
3. Reset the adaptive probability values for all the contexts of all the arithmetic integer coders to zero.
4. Invoke the halftone dictionary decoding procedure described in 6.7, with the parameters to the halftone dictionary decoding procedure set as shown in Table 30.

7.4.5 Halftone region segment syntax

The data parts of all three of the halftone region segment types ("intermediate halftone region", "immediate halftone region" and "immediate lossless halftone region") are coded identically, but are acted upon differently; see 8.2. The syntax of these segment types' data parts is specified here.

Table 30 — Parameters used to decode a halftone dictionary segment.

Name	Value
HDMMR	As shown in 7.4.4.1.1.
HDTEMPLATE	As shown in 7.4.4.1.1.
HDPW	As shown in 7.4.4.1.2.
HDPH	As shown in 7.4.4.1.3.
GRAYMAX	As shown in 7.4.4.1.4.

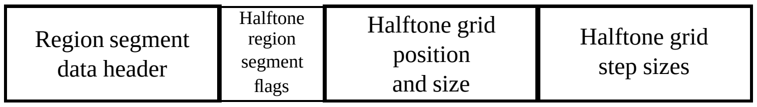

Region segment data header
Halftone region segment flags
Halftone grid position and size
Halftone grid step sizes

Figure 41 — Halftone region segment data header structure

7.4.5.1 Halftone region segment data header

The data part of a halftone region segment begins with a halftone region segment data header. This header contains the fields shown in Figure 41 and described below.

Region segment data header See 7.4.1.

Halftone region segment flags See 7.4.5.1.1.

Halftone grid position and size See 7.4.5.1.2.

Halftone grid vector See 7.4.5.1.3.

7.4.5.1.1 Halftone region segment flags

This one-byte field is formatted as shown in Figure 42 and as described below.

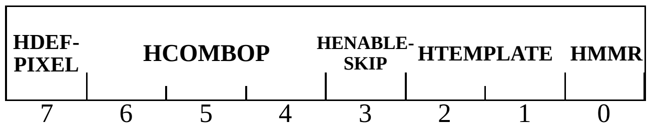


Figure 42 — Halftone region segment flags field structure

Bit 0 HMMR

If this bit is **1**, then the segment uses the MMR encoding variant. If this bit is **0**, then the segment uses the arithmetic encoding variant.

Bits 1–2 HTEMPLATE

This field controls the template used to decode halftone gray-scale value bitplanes if **HMMR** is **0**. If **HMMR** is **1**, this field must contain the value 0.

Bit 3 HENABLESKIP

This field controls whether gray-scale values that do not contribute to the region contents are skipped during decoding. If **HMMR** is **1**, this field must contain the value **0**.

Bits 4–6 HCOMBOP

This field has five possible values, representing one of four possible combination operators:

0 OR

1 AND

2 XOR

3 XNOR

4 REPLACE

Bit 7 HDEFPIXEL

This bit contains the value of any pixel that is not covered by any halftone pattern.

7.4.5.1.2 Halftone grid position and size

This field describes the location and size of the grid of gray-scale values. See Figure 24 for an illustration of these values. It is formatted as shown in Figure 43 and as described below.

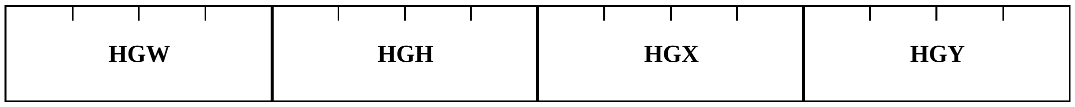


Figure 43 — Halftone grid position and size field structure

HGW See 7.4.5.1.2.1.

HGH See 7.4.5.1.2.2.

HGX See 7.4.5.1.2.3.

HGY See 7.4.5.1.2.4.

7.4.5.1.2.1 HGW

This four-byte field contains the width of the array of gray-scale values.

7.4.5.1.2.2 HGH

This four-byte field contains the height of the array of gray-scale values.

7.4.5.1.2.3 HGX

This four-byte field contains 256 times the horizontal offset of the origin of the halftone grid.

7.4.5.1.2.4 HGY

This four-byte field contains 256 times the vertical offset of the origin of the halftone grid.

7.4.5.1.3 Halftone grid vector

This field describes the vector used to draw the grid of gray-scale values. See Figure 24 for an illustration of these values. It is formatted as shown in Figure 44 and as described below.

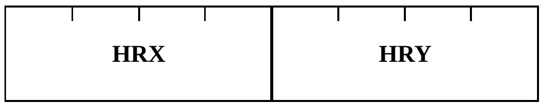


Figure 44 — Halftone grid vector field structure

HRX See 7.4.5.1.3.1.

HRY See 7.4.5.1.3.2.

7.4.5.1.3.1 HRX

This four-byte field contains 256 times the horizontal component of the halftone grid vector.

7.4.5.1.3.2 HRY

This four-byte field contains 256 times the vertical component of the halftone grid vector.

7.4.5.2 Decoding a halftone region segment

A halftone region segment is decoded according to the following steps.

1. Interpret its header, as described in 7.4.5.1.
2. Decode (or retrieve the results of decoding) the referred-to halftone dictionary segment.
3. Reset all the arithmetic coding context adaptive probability values for the generic region and generic refinement region decoding procedures to zero.
4. Reset the adaptive probability values for all the contexts of all the arithmetic integer coders to zero.
5. Invoke the halftone region decoding procedure described in 6.6, with the parameters to the halftone region decoding procedure set as shown in Table 31.

Table 31 — Parameters used to decode a halftone region segment.

Name	Value
HBW	As specified by the Region segment bitmap width in this segment's region segment data header.
HBH	As specified by the Region segment bitmap height in this segment's region segment data header.
HMMR	As shown in 7.4.5.1.1.
HTEMPLATE	As shown in 7.4.5.1.1.
HENABLESKIP	As shown in 7.4.5.1.1.
HCOMBOP	As shown in 7.4.5.1.1.
HDEFPIXEL	As shown in 7.4.5.1.1.
HGW	As shown in 7.4.5.1.2.1.
HGH	As shown in 7.4.5.1.2.2.
HGX	As shown in 7.4.5.1.2.3.
HGY	As shown in 7.4.5.1.2.4.
HRX	As shown in 7.4.5.1.3.1.
HRY	As shown in 7.4.5.1.3.2.
HNUMPATS	The number of halftone patterns in the halftone dictionary segment referred to by this segment.
HPATS	The halftone patterns in the halftone dictionary segment referred to by this segment.
HPW	The width, in pixels, of each of the halftone patterns contained in **HPATS**.
HPH	The height, in pixels, of each of the halftone patterns contained in **HPATS**.

7.4.6 Generic region segment syntax

The data parts of all three of the generic region segment types ("intermediate generic region", "immediate generic region" and "immediate lossless generic region") are coded identically, but are acted upon differently; see 8.2. The syntax of these segment types' data parts is specified here.

7.4.6.1 Generic region segment data header

The data part of a generic region segment begins with a generic region segment data header. This header contains the fields shown in Figure 45 and described below.

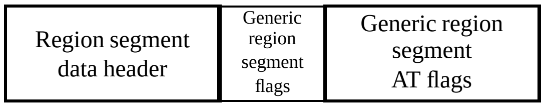

Region segment data header
Generic region segment flags
Generic region segment AT flags

Figure 45 — Generic region segment data header structure

Region segment data header See 7.4.1.

Generic region segment flags See 7.4.6.2.

Generic region segment AT flags See 7.4.6.3.

7.4.6.2 Generic region segment flags

This one-byte field is formatted as shown in Figure 46 and as described below.

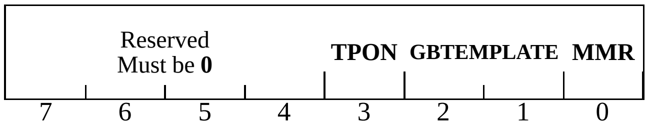


Figure 46 — Generic region segment flags field structure

Bit 0 MMR

Bits 1–2 GBTEMPLATE

This field specifies the template used for template-based arithmetic coding. If **MMR** is **1** then this field shall contain the value zero.

Bit 3 TPON

Bits 4–7 Reserved; shall be zero.

7.4.6.3 Generic region segment AT flags

This field is only present if **MMR** is **0**. If **GBTEMPLATE** is 0, it is an eight-byte field, formatted as shown in Figure 47 and as described below.

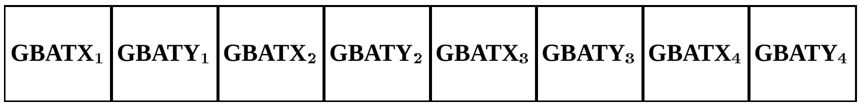


Figure 47 — Generic region AT flags field structure when GBTEMPLATE is 0

Byte 0 GBATX_1

Byte 1 GBATY_1

Byte 2 GBATX_2

Byte 3 GBATY_2

Byte 4 $\mathbf{GBATX_3}$

Byte 5 $\mathbf{GBATY_3}$

Byte 6 $\mathbf{GBATX_4}$

Byte 7 $\mathbf{GBATY_4}$

If **GBTEMPLATE** is 1, 2 or 3, it is a two-byte field formatted as shown in Figure 48 and as described below.

$\mathbf{GBATX_1}$	**$\mathbf{GBATY_1}$**

Figure 48 — Generic region AT flags field structure when GBTEMPLATE is not 0

Byte 0 $\mathbf{GBATX_1}$

Byte 1 $\mathbf{GBATY_1}$

If **GBTEMPLATE** is 1, 2 or 3 then the values of **$\mathbf{GBATX_2}$** through **$\mathbf{GBATX_4}$** and **$\mathbf{GBATY_2}$** through **$\mathbf{GBATY_4}$** are all zero.

The AT coordinate X and Y fields are signed values, and may take on values that are permitted according to Figure 7.

7.4.6.4 Decoding a generic region segment

A generic region segment is decoded according to the following steps.

1. Interpret its header, as described in 7.4.6.1
2. Reset all the arithmetic coding context adaptive probability values for the generic region decoding procedure to zero.
3. Invoke the generic region decoding procedure described in 6.2, with the parameters to the generic region decoding procedure set as shown in Table 32.

Table 32 — Parameters used to decode a generic region segment.

Name	Value
MMR	As shown in 7.4.6.2.
GBTEMPLATE	As shown in 7.4.6.2.
TPON	As shown in 7.4.6.2.
USESKIP	**0**
SBW	As specified by the Region segment bitmap width in this segment's region segment data header.
SBH	As specified by the Region segment bitmap height in this segment's region segment data header.
$\mathbf{GBATX_1}$	See 7.4.6.3
$\mathbf{GBATY_1}$	See 7.4.6.3
$\mathbf{GBATX_2}$	See 7.4.6.3
$\mathbf{GBATY_2}$	See 7.4.6.3
$\mathbf{GBATX_3}$	See 7.4.6.3
$\mathbf{GBATY_3}$	See 7.4.6.3
$\mathbf{GBATX_4}$	See 7.4.6.3
$\mathbf{GBATY_4}$	See 7.4.6.3

As a special case, as noted in 7.2.7, an immediate generic region segment may have an unknown length. In this case, it is also possible that the segment may contain fewer rows of bitmap data that are indicated in the segment's region segment data header.

In order for the decoder to correctly decode the segment, it must read the four-byte row count field, which is stored in the last four bytes of the segment's data part. These four bytes can be detected without knowing the length of the data part in advance: if **MMR** is **1**, they are preceded by the two-byte sequence `0x00 0x00`; if **MMR** is **0**, they are preceded by the two-byte sequence `0xFF 0xAC`. The row count field contains the actual number of rows contained in this segment; it must be no greater than the Region segment bitmap height value in the segment's region segment data header.

NOTE — The sequence `0x00 0x00` cannot occur normally within MMR-encoded data; the sequence `0xFF 0xAC` can occur only at the end of arithmetically-coded data. Thus, those sequences cannot occur by chance in the data that is decoded to generate the contents of the generic region.

7.4.7 Generic refinement region syntax

The data parts of all three of the generic refinement region segment types ("intermediate generic refinement region, "immediate generic refinement region" and "immediate lossless generic refinement region") are coded identically, but are acted upon differently; see 8.2. The syntax of these segment types' data parts is specified here.

7.4.7.1 Generic refinement region segment data header

The data part of a generic refinement region segment begins with a generic refinement region segment data header. This header contains the fields shown in Figure 49 and described below.

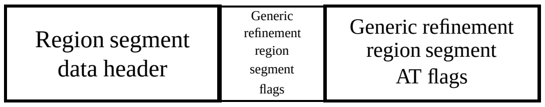

Region segment data header
Generic refinement region segment flags
Generic refinement region segment AT flags

Figure 49 — Generic refinement region segment data header structure

Region segment data header See 7.4.1.

Generic refinement region segment flags See 7.4.7.2.

Generic refinement region segment AT flags See 7.4.7.3.

7.4.7.2 Generic refinement region segment flags

This one-byte field is formatted as shown in Figure 50 and as described below.

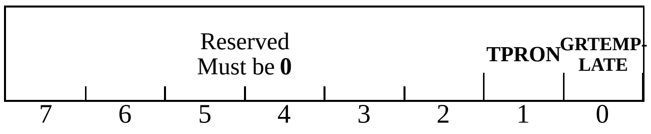


Figure 50 — Generic refinement region segment flags field structure

Bit 0 GRTEMPLATE

This field specifies the template used for template-based arithmetic coding.

Bit 1 TPRON

This field specifies whether typical prediction for refinement is used.

Bits 2–7 Reserved; shall be zero.

GRATX$_1$	**GRATY$_1$**	**GRATX$_2$**	**GRATY$_2$**

Figure 51 — Generic refinement region AT flags field structure

7.4.7.3 Generic region segment AT flags

This field is only present if **GRTEMPLATE** is 0. It is a four-byte field, formatted as shown in Figure 51 and as described below.

Byte 0 GRATX$_1$

Byte 1 GRATY$_1$

Byte 2 GRATX$_2$

Byte 3 GRATY$_2$

The AT coordinate X and Y fields are signed values, and may take on values that are permitted according to 6.3.5.3.

7.4.7.3.1 Reference bitmap selection

If this segment refers to another region segment, then set the reference bitmap **GRREFERENCE** to be the current contents of the buffer associated with the region segment that this segment refers to.

If this segment does not refer to another region segment, set **GRREFERENCE** to be a bitmap containing the current contents of the page buffer (see 8), restricted to the area of the page buffer specified by this segment's region segment data header.

7.4.7.4 Decoding a generic refinement region segment

A generic refinement region segment is decoded according to the following steps.

1. Interpret its header as described in 7.4.6.1.
2. Reset all the arithmetic coding context adaptive probability values for the generic refinement region decoding procedure to zero.
3. Determine the buffer associated with the region segment that this segment refers to.
4. Invoke the generic refinement region decoding procedure described in 6.3, with the parameters to the generic refinement region decoding procedure set as shown in Table 33

7.4.8 Page information segment syntax

A page information segment describes a page. It contains the fields shown in Figure 52 and described below.

Page bitmap width	Page bitmap height	Page X resolution	Page Y resolution	Page segment flags	Page striping information

Figure 52 — Page information segment structure

Page bitmap width See 7.4.8.1.

Page bitmap height See 7.4.8.2.

Page X resolution See 7.4.8.3.

Table 33 — Parameters used to decode a generic refinement region segment.

Name	Value
GRTEMPLATE	As shown in 7.4.6.2.
TPRON	As shown in 7.4.6.2.
GRW	As specified by the Region segment bitmap width in this segment's region segment data header.
GRH	As specified by the Region segment bitmap height in this segment's region segment data header.
GRREFERENCE	See 7.4.7.3.1.
GRREFERENCEDX	0
GRREFERENCEDY	0
$GRATX_1$	See 7.4.7.3
$GRATX_2$	See 7.4.7.3
$GRATY_1$	See 7.4.7.3
$GRATY_2$	See 7.4.7.3

Page Y resolution See 7.4.8.4.

Page segment flags See 7.4.8.5.

Page striping information See 7.4.8.6.

The first segment that is associated with any page shall be a page information segment.

7.4.8.1 Page bitmap width

This is a four-byte value containing the width in pixels of the page's bitmap.

7.4.8.2 Page bitmap height

This is a four-byte value containing height in pixels of the page's bitmap. In some cases, this value may not be known at the time that the page information segment is written. In this case, this field shall contain `0xffffffff`, and the actual page height may be communicated later, once it is known.

7.4.8.3 Page X resolution

This is a four-byte field containing the resolution of the original page medium, measured in pixels/metre in the horizontal direction. If this value is unknown then this field shall contain `0x00000000`.

7.4.8.4 Page Y resolution

This is a four-byte field containing the resolution of the original page medium, measured in pixels/metre in the vertical direction. If this value is unknown then this field shall contain `0x00000000`.

7.4.8.5 Page segment flags

This is a one-byte field. It is formatted as shown in Figure 53 and as described below.

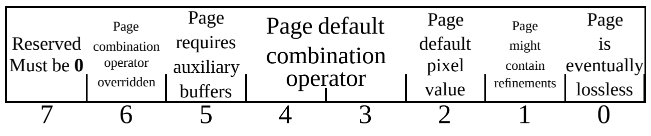

Reserved Must be 0
Page combination operator overridden
Page requires auxiliary buffers
Page default combination operator
Page default pixel value
Page might contain refinements
Page is eventually lossless
7
6
5
4
3
2
1
0

Figure 53 — Page segment flags field structure

Bit 0 Page is eventually lossless. If this bit is **0**, then the file does not contain a lossless representation of the original (pre-coding) page. If this bit is **1**, then the file contains enough information to reconstruct the original page.

Bit 1 Page might contain refinements. If this bit is **0**, then no refinement region segment may be associated with the page. If this bit is **1**, then such segments may be associated with the page.

Bit 2 Page default pixel value. A value of **0** indicates that any parts of the page bitmap not covered by any region segment, or not drawn into by any region segment, should be **0**. A value of **1** indicates that any parts of the page bitmap not covered by any region segment, or not drawn into by any region segment, should be **1**.

Bits 3–4 Page default combination operator. This field has four possible values, representing one of four possible combination operators:

0 OR

1 AND

2 XOR

3 XNOR

This operator is used to merge overlapping region segments, and also to combine region segments with the default pixel value.

Bit 5 Page requires auxiliary buffers. If this bit is **0**, then no region segment requiring an auxiliary buffer may be associated with the page. If this bit is **1**, then such segments may be associated with the page.

Bit 6 Page combination operator overridden. If this bit is **0**, then every non-refinement region segment associated with this page shall use the page's combination operator. If this bit is **1**, then non-refinement region segments associated with this page may use combination operators that are different from the page's combination operator.

NOTE — If all the region segments associated with a page use the same combination operator, then it is possible to reorder them to some extent (it is not possible switch the relative order of any refinement segment). If some of them use different combination operators, then the decoder is unable do any such reordering. Furthermore, the decoder cannot tell from the segment headers whether any such non-default combination operators are used in the page, so this bit indicates that reordering may be possible, if the decoder wishes to perform it.

Bit 7 Reserved; shall be **0**.

7.4.8.6 Page striping information

This is a two-byte field. It is formatted as shown in Figure 54 and as described below.

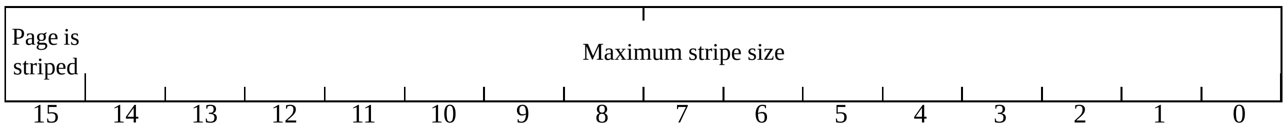


Figure 54 — Page striping information field structure

Bits 0–14 Maximum stripe size

Bit 15 Page is striped

If the "page is striped" bit is **1**, then the page may have end of stripe segments associated with it. In this case, the maximum size of each stripe (the distance between an end of stripe segment's end row and the end row of the previous end of stripe segment, or 0 in the case of the first end of strip segment) must be no more than the page's maximum stripe size.

If the page's bitmap height is unknown (indicated by a page bitmap height of `0xffffffff`) then the "page is striped" bit must be **1**.

7.4.9 End of page segment syntax

An end of page segment has no associated data. Its segment data length field shall be zero.

The last segment that is associated with any page shall be an end of page segment.

If a page's height was originally unknown, then there shall be at least one end of stripe segment associated with the page. In this case, the end row of that last stripe is the last row of the page bitmap and no region segment may occur between the last end of stripe segment and the end of page segment.

7.4.10 End of stripe segment syntax

An end of stripe segment states that the encoder has finished coding a portion of the current page, and will not revisit it. It specifies the Y coordinate of a row of the page; no segment following the end of stripe may modify any portion of the page bitmap that lines on or above that row. This row is called the "end row" of the stripe.

The end row specified by an end of stripe segment shall lie below any previous end row for that page.

A page whose height was originally unknown shall contain at least one end of stripe segment.

The segment data of an end of stripe segment consists of one four-byte value, specifying the Y coordinate of the end row.

7.4.11 End of file segment syntax

If a file contains an end of file segment, it shall be the last segment.

An end of file segment has no associated data. Its segment data length field shall be zero.

7.4.12 Supported profiles segment syntax

A supported profiles segment contains a list of the profiles that a given JBIG2 data stream is in compliance with. If any supported profiles segments are present, then the first segment shall be a supported profiles segment, and may not be associated with any page.

A supported profiles segment begins with a four-byte field containing the number of profiles listed. This field is followed by that many four-byte fields. Each of those fields contains a profile identification number. The data stream shall be in compliance with each of the profiles listed.

More than one supported profiles segment may be present. If more than one is present, then each one, other than the first one, shall be associated with a page. No page may have more than one supported profiles segment associated with it. Also, each supported profiles segment past the first one shall be more restrictive than the first one; that is, it shall list all of the profile identification numbers listed in the first segment, and possibly more. The segments making up each page shall, collectively, be in compliance with each of the profiles listed in any supported profiles segment associated with that page.

7.4.13 Code table segment syntax

A code table segment's syntax is described in B.

7.4.14 Extension segment syntax

An extension segment's data begins with a extension header:

Extension type This is a four-byte field which contains an identification of the type of data that are present in the extension segment.

The three most significant bits of this field have special meaning:

Bit 29 Reserved. Future revisions of this standard may define extension types; extension types may also be registered by other parties. Other parties may register only extension types with this bit equal to **0**; all extension types having bit 29 equal to **1** are reserved.

Bit 30 Dependent. If this bit is **1**, then the coding of the data in the extension segment is dependent on the exact encoding of the data in the segments that the extension segment refers to. Any file manipulation program that modifies those referred-to segments shall modify this extension segment's data correspondingly; if it does not understand the extension segment (due to not recognising its extension type), and if it is not a necessary extension segment, then the segment should be deleted.

Bit 31 Necessary. If this bit is **1**, then any decoder that does not know how to parse extensions of this extension segment's type will not be able to correctly decode the file to produce the intended decoded page images.

The remainder of the extension segment's data immediately follows the extension type field, and is formatted in some way particular to the type of extension.

7.4.15 Defined extension types

The following extension types are currently defined.

`0x2000000` ASCII comment. See 7.4.15.1.

`0x2000001` Binary comment. See 7.4.15.2.

`0x2000002` Unicode comment. See 7.4.15.3.

7.4.15.1 Comment

An ASCII comment extension segment holds textual information about some other segment, page, or the bitstream as a whole. If it refers to no other segments, and is associated with no page, then it contains some set of comments applying to the entire bitstream. If it refers to no other segments, but is associated with some page, then it contains some set of comments applying to that page. If it refers to some segments, then it contains some set of comments applying to those segments.

An ASCII comment segment contains a number of (name, value) pairs. Each element of each pair is a string of characters, and is terminated by an ASCII NUL (0x00) character. The last pair is followed by an additional NUL character.

EXAMPLE — The comment containing the following pairs

```
Title          An Illustrated History of False Teeth
Author         The Big Cheese
```

is stored as the following sequence of bytes. The bytes are shown as hexadecimal numbers together with their ASCII equivalents, with "`.`" indicating an unprintable byte. Note the four-byte extension type at the start of the segment data:

```
20 00 00 00 54 69 74 6C 65 00 41 6E 20 49 6C 6C   ...Title.An Ill
75 73 74 72 61 74 65 64 20 48 69 73 74 6F 72 79  ustrated History
20 6F 66 20 46 61 6C 73 65 20 54 65 65 74 68 00   of False Teeth.
41 75 74 68 6F 72 00 54 68 65 20 42 69 67 20 43  Author.The Big C
68 65 65 73 65 00 00                             heese..
```

7.4.15.2 Binary comment

A binary comment segment may contain any data. This comment follows the same format as the PNG Zlib-compressed comment.

7.4.15.3 Unicode comment

8 Page Make-up

8.1 Decoder model

This section describes the result that a decoder conforming to this International Standard must produce when decoding a page. It does this by specifying a set of steps that produce the correct result; a conforming decoder need not perform these exact steps, but must produce the same result as if the steps had been followed.

Here we describe only the steps taken to decode a single page. A conforming decoder may operate on multiple pages at once, as long as it produces the correct final result for each page.

In the following description, we will assume for simplicity that the decoder has a single page buffer, auxiliary buffers to be used while decoding that page, and additional dictionary memory. Decoders with other components are allowed, as long as they produce the same page buffer as this abstract decoder does.

At the end of the decoding process, the page buffer contains the result of decoding the page.

Each auxiliary buffer has a location associated with it; this location is the location of the buffer's top left pixel, relative to the top left pixel of the page buffer. Some combinations of image segments require the use of auxiliary buffers; others can be decoded directly into the page buffer. See 8.2 for details on how combinations of image segments are to be interpreted.

The dictionary memory contains the information obtained by decoding dictionary segments.

8.2 Page image composition

The final bitmap for each page is coded by zero or more image segments associated with that page. Each image segment describes some of the contents of a rectangular region of the page. Since these regions may overlap, and since some regions might be described as multiple levels of quality, it is important to define what the rules for image segment composition are. Also, since a decoder might want to display intermediate representations of a page, based on partial information, it is useful to suggest the interpretation of partial pages.

Each page specifies a default pixel value (**0** or **1**) and one of four combination operators (OR, AND, XOR, XNOR). Every segment also specifies a combination operator of its own. The combination operators overridden flag bit in the page information segment specifies whether any of the page's image segments overrides the page combination operator. If the bit is **0**, then no non-refinement image segment associated with this page overrides the page combination operator. The decoder may use this information to optimise its decoding.

The result of decoding an image segment is a bitmap. The size of this bitmap and its location with respect to the page buffer is given in the image segment header.

The final contents of the page buffer that the decoder must produce as the final result of decoding a page are those that would be generated by the following steps:

1. Fetch and decode the page information segment.

2. Create the page buffer, of the size given in the page information segment.

   If the page size is unknown, then this is not possible. However, in this case the page must be striped, and the maximum stripe height specified, and the initial page buffer can be created with height intially equal to this maximum stripe height; as each end-of-stripe segment is encountered, the page buffer's height can be increased, until the end-of-page segment (together with the last end-of-stripe segment) allow determination of the page's actual height.

   NOTE — In general, this discussion disregards the effects of striping.

3. Fill the page buffer with the page's default pixel value.

4. Fetch the next image segment associated with that page.

5. The following cases exist:

   (a) The image segment is an immediate direct image segment. In this case, decode the image segment. The result of decoding the image segment is a bitmap; combine this bitmap with the current contents of the page buffer, using the image segment's combination operator.

(b) The image segment is an intermediate direct image segment. In this case, allocate a new auxiliary buffer, using the size and location specified in the segment's image segment data header. This buffer is initially associated with the image segment. Decode the image segment, placing the results into the auxiliary buffer.

(c) The image segment is an immediate refinement image segment that refers to no other segments. In this case, the image segment is acting as a refinement of the page buffer. Perform this refinement on the region of the page buffer specified in the image segment, according to the data contained in the refinement image segment. This replaces a part of the page buffer with a refined version.

(d) The image segment is an immediate refinement image segment that refers to another image segment. This other image segment must be a previously occurring intermediate image segment that has not yet had a refinement image segment refer to it; the other image segment thus has an auxiliary buffer associated with it. Perform the refinement operation on that auxiliary buffer, according to the data contained in the current image segment, and combine the resulting buffer with the page buffer using the current image segment's combination operator, at the location associated with the auxiliary buffer. Discard the auxiliary buffer.

(e) The image segment is an intermediate refinement image segment. This image segment must refer to one other image segment, which must be a previously occurring intermediate image segment that has not yet had a refinement image segment refer to it; the other image segment thus has an auxiliary buffer associated with it. Perform the refinement operation on that auxiliary buffer, according to the data contained in the current image segment. Replace the previous contents of the auxiliary buffer with the bitmap resulting from the refinement. Change the association of the auxiliary buffer, so that it is now associated with the current image segment, and is no longer associated with the other image segment.

6. Repeat steps 4 and 5 until there are no more image segments associated with the page.

7. The result of decompressing that page is given by the final contents of the page buffer.

NOTE 2 — The rules here are quite simple: if it's immediate, draw it into the page buffer; if it's intermediate, it involves an auxiliary buffer.

Some examples of these rules in operation:

EXAMPLE 1 — If the page contains no image segments, then the page buffer is filled entirely with the page's default pixel value.

EXAMPLE 2 — The page information segment for page 3 specifies that the page default combination operator is OR and the page default pixel value is **0**. The image segments associated with page 3 are, in order,

- Segment 7, an intermediate symbol image segment
- Segment 8, an intermediate generic bitmap image segment
- Segment 13, an immediate generic bitmap refinement image segment that refers to segment 8, whose external combination operator is OR
- Segment 14, an immediate generic bitmap refinement image segment that refers to segment 7, whose external combination operator is OR
- Segment 19, an immediate symbol image segment whose external combination operator is OR
- Segment 22, an immediate generic bitmap image segment whose external combination operator is OR

The resulting page buffer is the buffer that would be obtained by following the steps

1. Fill the page buffer with the value **0**

2. Decode segment 7 into an auxiliary buffer
3. Decode segment 8 into an auxiliary buffer
4. Refine segment 8's auxiliary buffer, according to the refinement information in segment 13, and draw the refined buffer into the page buffer using OR
5. Refine segment 7's auxiliary buffer, according to the refinement information in segment 14, and draw the refined buffer into the page buffer using OR
6. Decode segment 19 and draw the resulting bitmap into the page buffer using OR
7. Decode segment 22 and draw the resulting bitmap into the page buffer using OR

The correct result is also obtained *no matter what order* steps 4 through 7 are performed in; thus a conforming decoder is free to choose any order to decode these steps. In fact, any order of steps 2 through 7 produces the correct result, as long as step 2 is performed before step 5 and step 3 is performed before step 4.

EXAMPLE 3 — If a page contains several immediate direct-coded image segments that do not override the page's combination operator, and an immediate refinement image segment that does not refer to any other segments, then the resulting page buffer is the buffer that would be obtained by

- filling the page buffer with the page's default pixel value
- drawing all the direct-coded image segments that precede the refinement image segment
- refining the portion of the image covered by the refinement image segment
- drawing all the direct-coded image segments that follow the refinement image segment

In this case, the order of drawing does matter: all the immediate segments that precede the refinement segment must be drawn before the refinement segment is drawn, and the refinement segment must be drawn before any of the immediate segments that follow it.

NOTE 3 — In some cases, the decoder may want to display some intermediate form of the page. For example, it may want to provide the user with a progressive display of the page contents as the page segments are received over some transmission medium. Any intermediate images that it displays are entirely up to the decoder, and are not specified by this standard. However, a useful rule is for the decoder to take the current contents of the page buffer and any currently active auxiliary buffers, combine them with the page's combination operator, and display that to the user. If the page combination operator is XOR or XNOR, then this combination can be done reversibly, and so might be done into the actual page buffer, then undone after it has been displayed to the user. If the page combination operator is OR or AND, then this combination is not reversible and an extra buffer is required to hold the results of the combination.

The step-by-step description above is intended to specify only the results of the decompression. A conforming decoder may take any steps it desires, as long as the final page buffer is the same as would have been obtained by following the steps.

EXAMPLE 4 — A decoder might notice that an intermediate image segment refers to a region of the page that is not overlapped by any other image segment, and so might not actually allocate an auxiliary buffer for that image segment, but might use the page buffer immediately. It can do this only if it is sure that this will not change the final results of decoding the page's image segments.

9 Test Methods and Datastream Examples

Annex A
(normative)
Arithmetic Integer Decoding Procedure

A.1 General Description

This International Standard uses a number of arithmetic decoding procedures to decode integer values. These are

IAAI	Used to decode the number of symbol instances in an aggregation
IADH	Used to decode the difference in height between two height classes
IADS	Used to decode the S coordinate of the second and subsequent symbol instances in a strip
IADT	Used to decode the T coordinate of the second and subsequent symbol instances in a strip
IADW	Used to decode the difference in width between two symbols in a height class
IAEX	Used to decode export flags
IAFS	Used to decode the S coordinate of the first symbol instance in a strip
IAID	Used to decode the symbol IDs of symbol instances
IAIT	Used to decode the T coordinate of the symbol instances in a strip
IARDH	Used to decode the delta height of symbol instance refinements
IARDW	Used to decode the delta width of symbol instance refinements
IARDX	Used to decode the delta X position of symbol instance refinements
IARDY	Used to decode the delta Y position of symbol instance refinements
IARI	Used to decode the R_I bit of symbol instances

Each of these is used to decode integer values (which may include the out-of-band value OOB). The coding for an integer is based on a decision tree.

An invocation of an arithmetic integer decoding procedure involves decoding a sequence of bits, where each bit is decoded using a context formed by the bits decoded previously in this invocation. Each context for each arithmetic integer decoding procedure has its own adaptive probability estimate used by the underlying arithmetic coder, described in Annex E. The sequence of bits decoded is interpreted to form a value.

Table A.1 is used by all the arithmetic integer decoding procedures except for IAID.

A.2 Procedure for decoding values (except IAID)

The flowchart in Figure A.1 is used as part of the decoding procedure. It produces two values, V and S. The result of the integer arithmetic decoding procedure is equal to

- V if $S = \mathbf{0}$
- $-V$ if $S = \mathbf{1}$ and $V > 0$
- OOB if $S = \mathbf{1}$ and $V = 0$

Thus, V represents the absolute value of the integer value being decoded, and S represents the sign; the otherwise-redundant value -0 is interpreted to mean "OOB".

In Figure A.1, each bit is decoded in a context formed from the particular integer arithmetic decoding procedure being invoked, and the previous bits decoded in this invocation of that decoding procedure. This context is formed as follows.

1. Set
$$\text{PREV} = 1$$
2. Follow the flowchart in Figure A.1. Decode each bit with CX equal to "IAx + PREV" where "IAx" represents the identifier of the current arithmetic integer decoding procedure, "+" represents concatenation, and the rightmost 9 bits of PREV are used.

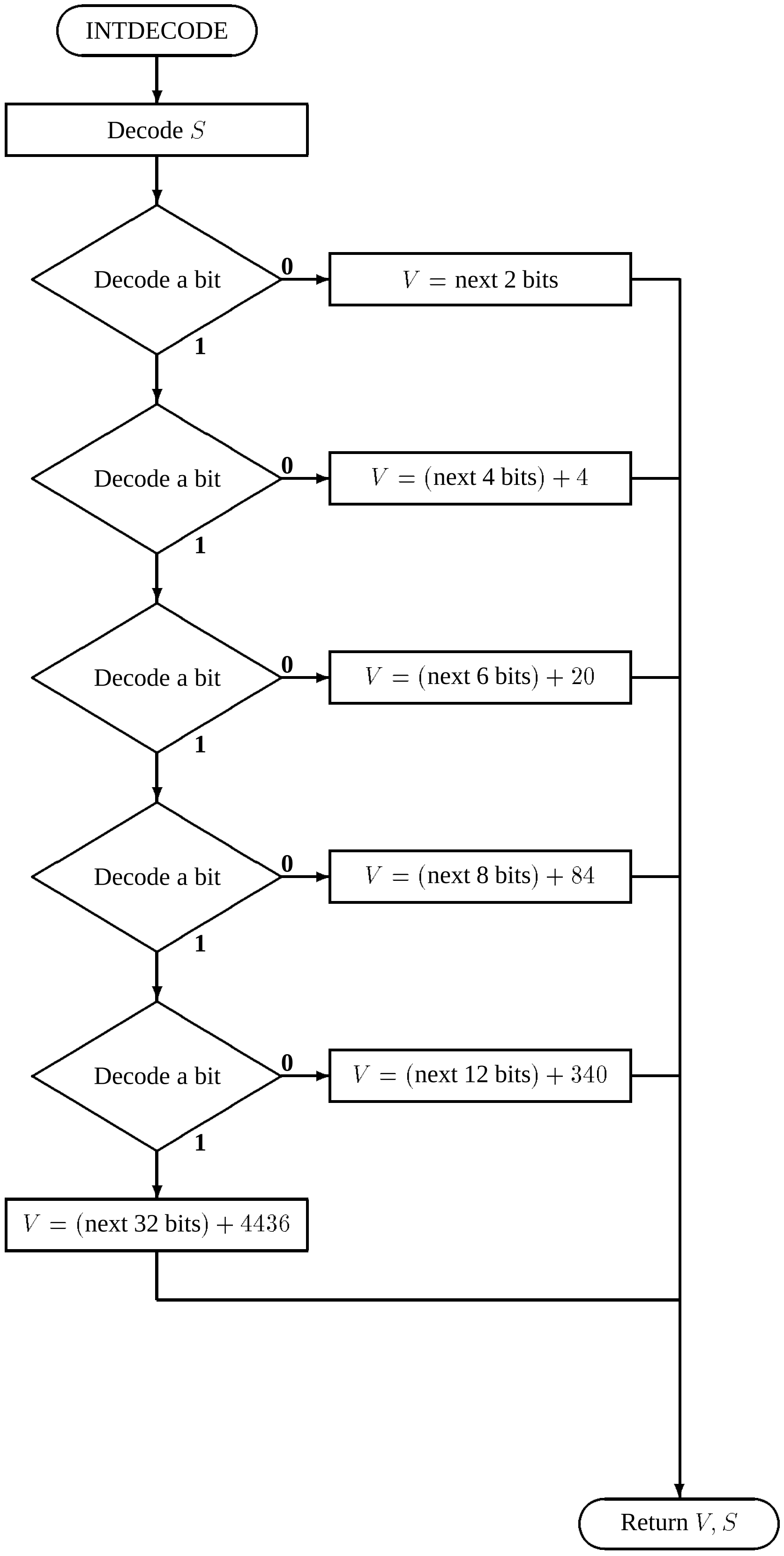

INTDECODE
Decode S
Decode a bit
0
1
V = next 2 bits
Decode a bit
0
1
V = (next 4 bits) $+ 4$
Decode a bit
0
1
V = (next 6 bits) $+ 20$
Decode a bit
0
1
V = (next 8 bits) $+ 84$
Decode a bit
0
1
V = (next 12 bits) $+ 340$
V = (next 32 bits) $+ 4436$
Return V, S

Figure A.1 — Flowchart for the integer arithmetic decoding procedures (except IAID)

Table A.1 — Arithmetic integer decoding procedure table

VAL	Encoding
0…3	**00** $+$ VAL encoded as 2 bits
-1	**1001**
-3 …-2	**101** $+ (-\text{VAL} - 2)$ encoded as 1 bit
4 …19	**010** $+ (\text{VAL} - 4)$ encoded as 4 bits
-19 …-4	**110** $+ (-\text{VAL} - 4)$ encoded as 4 bits
20 …83	**0110** $+ (\text{VAL} - 20)$ encoded as 6 bits
-83 …-20	**1110** $+ (-\text{VAL} - 20)$ encoded as 6 bits
84 …339	**01110** $+ (\text{VAL} - 84)$ encoded as 8 bits
-339 …-84	**11110** $+ (-\text{VAL} - 84)$ encoded as 8 bits
340 …4435	**011110** $+ (\text{VAL} - 340)$ encoded as 12 bits
-4435 …-340	**111110** $+ (-\text{VAL} - 340)$ encoded as 12 bits
4436 …∞	**011111** $+ (\text{VAL} - 4436)$ encoded as 32 bits
$-\infty$ …-4436	**111111** $+ (-\text{VAL} - 4436)$ encoded as 32 bits
OOB	**1000**

3. After each bit is decoded: If PREV < 256 set

$$\text{PREV} = (\text{PREV} << 1) \text{ OR D}$$

   Otherwise set

$$\text{PREV} = (((\text{PREV} << 1) \text{ OR D}) \text{ AND } 511) \text{ OR } 256$$

   where D represents the value of the just-decoded bit.

   Thus, PREV always contains the values of the eight most-recently-decoded bits, plus a leading **1** bit, which is used to indicate the number of bits decoded so far.

4. The sequence of bits decoded, interpreted according to Table A.1, gives the value that is the result of this invocation of the integer arithmetic decoding procedure.

Note that each type of data, and each integer arithmetic decoding procedure, uses a separate set of contexts: the contexts used for IAFS are separate from the contexts used for IADW, for example.

EXAMPLE — An invocation of IADW might go as follows.

- Using the adaptive probability estimate identified by setting CX equal to "IADW**000000001**", decode a bit. Suppose the value decoded is **0**.
- Using CX = IADW**000000010**, decode a bit; suppose the value decoded is **1**.
- Using CX = IADW**000000101**, decode a bit; suppose the value decoded is **0**.
- Using CX = IADW**000001010**, decode a bit; suppose the value decoded is **1**.
- Using CX = IADW**000010101**, decode a bit; suppose the value decoded is **0**.
- Using CX = IADW**000101010**, decode a bit; suppose the value decoded is **0**.
- Using CX = IADW**001010100**, decode a bit; suppose the value decoded is **0**.
- The sequence of bits decoded so far is **0101000**. According to Table A.1 and Figure A.1, this corresponds to the value 12 ($S = 0$, $V = 12$), which is the result of this invocation of IADW.

A context is identified by an arithmetic integer decoding procedure name and a sequence of nine bits. Thus, each arithmetic integer decoding procedure requires 512 bytes of storage for its context memory.

A.3 The IAID decoding procedure

This decoding procedure is different from all the other integer arithmetic decoding procedure. It uses fixed-length representations of the values being decoded, and does not limit the number of previously-decoded bits used as part of the context. The length is equal to **SBSYMCODELEN**. This decoding procedure is only invoked from within the symbol region decoding procedure, so at the time of invocation **SBSYMCODELEN** is known.

The procedure for decoding an integer using the IAID decoding procedure is as follows.

1. Set

$$\text{PREV} \quad = \quad 1$$

2. Decode **SBSYMCODELEN** bits as follows

   (a) Decode a bit with CX equal to "IAID + PREV" where "+" represents concatenation, and the rightmost **SBSYMCODELEN** $+\ 1$ bits of PREV are used.

   (b) After each bit is decoded, set

   $$\text{PREV} = (\text{PREV} << 1)\ \text{OR D}$$

   where D represents the value of the just-decoded bit.

   Thus, PREV always contains the values of all the bits decoded so far, plus a leading **1** bit, which is used to indicate the number of bits decoded so far.

3. After **SBSYMCODELEN** bits have been decoded, set

$$\text{PREV} = \text{PREV} - 2^{\textbf{SBSYMCODELEN}}$$

   This step has the effect of clearing the topmost (leading **1**) bit of PREV before returning it.

4. The contents of PREV are the result of this invocation of the IAID decoding procedure.

The number of contexts required is $2^{\textbf{SBSYMCODELEN}}$, which is less than twice the maximum symbol ID. Thus, the amount of memory needed for contexts can be calculated from the number of symbols, and is typically no more than two bytes per symbol.

EXAMPLE — Suppose that paramSBSYMCODELEN $= 3$. An invocation of IAID might go as follows.

- Using the adaptive probability estimate identified setting CX equal to "IAID**0001**", decode a bit. Suppose the value decoded is **0**.
- Using CX $=$ IAID**0010**, decode a bit; suppose the value decoded is **1**.
- Using CX $=$ IAID**0101**, decode a bit; suppose the value decoded is **0**.
- At this point, PREV $=$ **1010**. Apply Step 3; PREV is now **010**. Thus, the result of this invocation of the IAID decoding procedure is the value **010**, or (in decimal) 2.

The context identfication used here depends on the value of **SBSYMCODELEN**. In all cases the arithmetic coder contexts will be reset in between changes of **SBSYMCODELEN**: **SBSYMCODELEN** never changes during the decoding of a single segment (but may change between segments).

Annex B
(normative)
Huffman Table Decoder

Code tables may be used for encoding any type of numerical data in the Huffman variant coders. In many locations where a table is used, the encoder has the option of using one of the standard tables, or sending its own table. A code table segment provides the means to send such a custom table. The code table is a list of code table lines, each describing how to encode a single value, or a value from a specified range. A table may optionally be able to code for an OOB code, which is an out-of-band signal to the decoder using the table.

B.1 Code Table Structure

Figure B.1 shows the internal structure of an encoded Huffman table. It consists of a set of table lines, each of which describes the encoding for a range of numerical values. There are also, potentially, two additional table lines that encode "open-ended" ranges. The smallest value that can be encoded in a table described according to this specification is -2147483648 (-2^{31}) and the largest value is 2147483647 ($2^{31} - 1$), so these ranges are not really open-ended; however, they are treated specially. There is also, potentially, an additional table line that encodes an out-of-band value OOB.

Code table flags
Code table lowest value
Code table highest value
First table line
Second table line
. . .
Last table line
Lower range table line
Upper range table line
Out-of-band table line

Figure B.1 — Coded structure of a Huffman table.

Each table line specifies the length of the prefix that is associated with it and the number of bits that follow that prefix to encode a value.

A decoder decoding an encoded Huffman table shall decode the table that is produced by the following steps.

1. Decode the code table flags field as described in B.1.1. This sets the values HTOOB, HTPS and HTRS.
2. Decode the code table lowest value field as described in B.1.2. Let HTLOW be the value decoded.
3. Decode the code table highest value field as described in B.1.3. Let HTHIGH be the value decoded.
4. Set

$$\begin{aligned} \text{CURRANGELOW} &= \text{HTLOW} \\ \text{NTEMP} &= 0 \end{aligned}$$

5. Decode each table line as follows.
   (a) Read HTPS bits. Set PREFLEN[NTEMP] to the value decoded.
   (b) Read HTRS bits. Let RANGELEN[NTEMP] be the value decoded.
   (c) Set

$$\begin{aligned} \text{RANGELOW[NTEMP]} &= \text{CURRANGELOW} \\ \text{NTEMP} &= \text{NTEMP} + 1 \\ \text{CURRANGELOW} &= \text{CURRANGELOW} + 2^{\text{RANGELEN[NTEMP]}} \end{aligned}$$

(d) If CURRANGELOW $\geq$ HTHIGH then proceed to step 6.

6. Read HTPS bits. Let LOWPREFLEN be the value read.
7. Set

$$\begin{aligned} \text{PREFLEN}[\text{NTEMP}] &= \text{LOWPREFLEN} \\ \text{RANGELEN}[\text{NTEMP}] &= 32 \\ \text{RANGELOW}[\text{NTEMP}] &= \text{HTLOW} - 1 \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

This is the lower range table line for this table.

8. Read HTPS bits. Let HIGHPREFLEN be the value read.
9. Set

$$\begin{aligned} \text{PREFLEN}[\text{NTEMP}] &= \text{HIGHPREFLEN} \\ \text{RANGELEN}[\text{NTEMP}] &= 32 \\ \text{RANGELOW}[\text{NTEMP}] &= \text{HTHIGH} \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

This is the upper range table line for this table.

10. If HTOOB is **1**, then
    (a) Read HTPS bits. Let OOBPREFLEN be the value read.
    (b) Set

$$\begin{aligned} \text{PREFLEN}[\text{NTEMP}] &= \text{OOBPREFLEN} \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

    This is the out-of-band table line for this table. Note that there is no range associated with this value.

11. Create the prefix codes using the algorithm described in B.2.

B.1.1 Code table flags

This one-byte field has the following bits defined:

Bit 0 HTOOB. If this bit is **1**, the table can code for an out-of-band value.

Bits 1–3 Number of bits used in code table line prefix size fields. The value of HTPS is the value of this field plus one.

Bits 4–6 Number of bits used in code table line range size fields. The value of HTRS is the value of this field plus one.

Bit 7 Reserved; must be zero.

B.1.2 Code table lowest value

This signed four-byte field is the lower bound of the first table line in the encoded table.

B.1.3 Code table highest value

This signed four-byte field is one larger than the upper bound of the last normal table line in the encoded table.

B.2 Assigning the Prefix codes

Given the table of prefix code lengths, PREFLEN, and the number of codes to be assigned, NTEMP, this algorithm assigns a unique prefix code to each table line, of the length given by PREFLEN for that table line.

Note that the PREFLEN value 0 indicates that the table line is never used.

1. Build a histogram of the number of times each prefix length value occurs in PREFLEN in the array LENCOUNT. LENCOUNT$[I]$ is the number of times that the value I occurs in the array PREFLEN.

2. Let LENMAX be the largest value for which LENCOUNT[LENMAX] > 0. Set

$$\begin{aligned} \text{CURLEN} &= 1 \\ \text{FIRSTCODE}[0] &= 0 \\ \text{LENCOUNT}[0] &= 0 \end{aligned}$$

3. While CURLEN $\leq$ LENMAX, perform the following operations.

   (a) Set

$$\begin{aligned} \text{FIRSTCODE[CURLEN]} &= (\text{FIRSTCODE[CURLEN} - 1] + \text{LENCOUNT[CURLEN} - 1]) \times 2 \\ \text{CURCODE} &= \text{FIRSTCODE[CURLEN]} \\ \text{CURTEMP} &= 0 \end{aligned}$$

   (b) While CURTEMP $<$ NTEMP, perform the following operations.

      i. If PREFLEN[CURTEMP] $=$ CURLEN, then set

$$\begin{aligned} \text{CODES[CURTEMP]} &= \text{CURCODE} \\ \text{CURCODE} &= \text{CURCODE} + 1 \end{aligned}$$

      ii. Set CURTEMP $=$ CURTEMP $+ 1$.

   (c) Set

$$\text{CURLEN} = \text{CURLEN} + 1$$

After this algorithm has executed, then table line number I has been assigned a PREFLEN$[I]$-bit long code, whose value is stored in the PREFLEN$[I]$ low-order bits of CODES$[I]$, unless PREFLEN$[I]$ was equal to zero, in which case that table line has not been assigned any code.

B.3 Using a Huffman Table

To decode a value using a Huffman table, perform the following steps.

1. Read one bit at a time until the bit string read matches the code assigned to one of the table lines. Since no code forms a prefix of any other code, this is possible. Let I be the index of the table line whose code was decoded.

2. Read RANGELEN$[I]$ bits. Let HTOFFSET be the value read.

3. If HTOOB is **1** for this table, and table line I is the out-of-band table line for this table, then set

$$HTVAL = OOB$$

4. Otherwise, if table line I is the lower range table line for this table, then set

$$\text{HTVAL} = \text{RANGELOW}[I] - \text{HTOFFSET}$$

5. Otherwise, set

$$\text{HTVAL} = \text{RANGELOW}[I] + \text{HTOFFSET}$$

The value of HTVAL is the value decoded using this table. Note that this may be a numerical value or the special value OOB.

EXAMPLE — The encoding for Table B.1 might be the sequence of bytes, in hexadecimal

```
0x42 0x00 0x00 0x00 0x00 0x00 0x01
0x01 0x10 0x49 0x23 0x81 0x80
```

Decoding this according to the algorithm of B.1 proceeds as follows.

- The code table flags field, `0x42`. This field itself breaks down into the fields, in binary, **0 100 001 0**, which decode to produce the assignments

$$\begin{aligned} \text{HTOOB} &= \mathbf{0} \\ \text{HTPS} &= 2 \\ \text{HTRS} &= 5 \end{aligned}$$

- The code table lowest value field, and the value of HTLOW, `0x00000000`.
- The code table highest value field, and the value of HTHIGH, `0x00010110` (which, in decimal, is 65808).
- Three table lines, the lower range table line and the upper range table line. These are encoded as the sequence of bytes `0x49 0x23 0x81 0x80`, or in binary, **01001001 00100011 10000001 10000000**. This bitstring is further broken down into the table lines as follows.

0100100	The first two (HTPS) bits of this table line indicate a prefix length of 1, and the last five (HTRS) bits of this table line indicate a range length of 4.
1001000	This table line has a prefix length of 2 and a range length of 8.
1110000	This table line has a prefix length of 3 and a range length of 16.
00	The lower range table line has a prefix length of 0, indicating that this table line is not used.
11	The upper range table line has a prefix length of 3.
0000000	Seven bits of padding, to fill out the last byte.

After decoding these table lines, the value of NTEMP is 5, and the arrays PREFLEN, RANGELEN and RANGELOW are

PREFLEN	1	2	3	0	3
RANGELEN	4	8	16	32	32
RANGELOW	0	16	272	-1	65808

Applying the algorithm of B.2 to this yields the array of codes, in binary,

CODES	**0**	**10**	**110**	X	**111**

where the X indicates that the lower range table line has not been assigned a code. Thus, the prefix code **0** precedes a 4-bit field encoding a value from 0 to 15; the prefix code **10** precedes an 8-bit field encoding a value from 16 to 271, and so on, as shown in Table B.1.

B.4 Standard Huffman Tables

This section presents some standard Huffman tables than may be used in the appropriate contexts without having been previously transmitted.

Each Huffman table is presented in a form that is similar to the table transmission described above. The table parameter HTOOB is given (HTPS, HTRS, HTLOW and HTHIGH can be derived from the values in the table), followed by a list of table lines, giving the range to which that table line applies, the table line prefix length, table line range length, and the actual encoding (prefix and base value) for that table line; these table lines are followed by a lower and upper range table line, and optionally (depending on HTOOB) an out-of-band table line. In some cases the lower or upper range table lines are omitted from the tables as shown, indicating that these table lines are not used in the table (and would be assigned a PREFLEN value of zero).

Table B.1 — Standard Huffman table A

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
$0 \ldots 15$	1	4	**0** $+$ VAL encoded as 4 bits
$16 \ldots 271$	2	8	**10** $+$ (VAL $- 16$) encoded as 8 bits
$272 \ldots 65807$	3	16	**110** $+$ (VAL $- 272$) encoded as 16 bits
$65808 \ldots \infty$	3	32	**111** $+$ (VAL $- 65808$) encoded as 32 bits

Table B.2 — Standard Huffman table B

HTOOB	**1**		
VAL	PREFLEN	RANGELEN	Encoding
0	1	0	**0**
1	2	0	**10**
2	3	0	**110**
$3 \ldots 10$	4	3	**1110** $+$ (VAL $- 3$) encoded as 3 bits
$11 \ldots 74$	5	6	**11110** $+$ (VAL $- 11$) encoded as 6 bits
$75 \ldots \infty$	6	32	**111110** $+$ (VAL $- 75$) encoded as 32 bits
OOB	6		**111111**

Table B.3 — Standard Huffman table C

HTOOB	**1**		
VAL	PREFLEN	RANGELEN	Encoding
$-256 \ldots -1$	8	8	**11111110** $+$ (VAL $+ 256$) encoded as 8 bits
0	1	0	**0**
1	2	0	**10**
2	3	0	**110**
$3 \ldots 10$	4	3	**1110** $+$ (VAL $- 3$) encoded as 3 bits
$11 \ldots 74$	5	6	**11110** $+$ (VAL $- 11$) encoded as 6 bits
$-\infty \ldots -257$	8	32	**11111111** $+$ ($-257 -$ VAL) encoded as 32 bits
$75 \ldots \infty$	7	32	**1111110** $+$ (VAL $- 75$) encoded as 32 bits
OOB	6		**111110**

Table B.4 — Standard Huffman table D

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
1	1	0	**0**
2	2	0	**10**
3	3	0	**110**
$4 \ldots 11$	4	3	**1110** $+$ (VAL $- 4$) encoded as 3 bits
$12 \ldots 75$	5	6	**11110** $+$ (VAL $- 12$) encoded as 6 bits
$76 \ldots \infty$	5	32	**11111** $+$ (VAL $- 76$) encoded as 32 bits

Table B.5 — Standard Huffman table E

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
$-255\ldots0$	7	8	**1111110** $+$ (VAL $+ 255$) encoded as 8 bits
1	1	0	**0**
2	2	0	**10**
3	3	0	**110**
$4\ldots11$	4	3	**1110** $+$ (VAL $- 4$) encoded as 3 bits
$12\ldots75$	5	6	**11110** $+$ (VAL $- 12$) encoded as 6 bits
$-\infty\ldots-256$	7	32	**1111111** $+$ ($-256 -$ VAL) encoded as 32 bits
$76\ldots\infty$	6	32	**111110** $+$ (VAL $- 76$) encoded as 32 bits

Table B.6 — Standard Huffman table F

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
$-2048\ldots-1025$	5	10	**11100** $+$ (VAL $+ 2048$) encoded as 10 bits
$-1024\ldots-513$	4	9	**1000** $+$ (VAL $+ 1024$) encoded as 9 bits
$-512\ldots-257$	4	8	**1001** $+$ (VAL $+ 512$) encoded as 8 bits
$-256\ldots-129$	4	7	**1010** $+$ (VAL $+ 256$) encoded as 7 bits
$-128\ldots-65$	5	6	**11101** $+$ (VAL $+ 128$) encoded as 6 bits
$-64\ldots-33$	5	5	**11110** $+$ (VAL $+ 64$) encoded as 5 bits
$-32\ldots-1$	4	5	**1011** $+$ (VAL $+ 32$) encoded as 5 bits
$0\ldots127$	2	7	**00** $+$ VAL encoded as 7 bits
$128\ldots255$	3	7	**010** $+$ (VAL $- 128$) encoded as 7 bits
$256\ldots511$	3	8	**011** $+$ (VAL $- 256$) encoded as 8 bits
$512\ldots1023$	4	9	**1100** $+$ (VAL $- 512$) encoded as 9 bits
$1024\ldots2047$	4	10	**1101** $+$ (VAL $- 1024$) encoded as 10 bits
$-\infty\ldots-2049$	6	32	**111110** $+$ ($-2049 -$ VAL) encoded as 32 bits
$2048\ldots\infty$	6	32	**111111** $+$ (VAL $- 2048$) encoded as 32 bits

Table B.7 — Standard Huffman table G

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
$-1024\ldots-513$	4	9	**1000** $+$ (VAL $+ 1024$) encoded as 9 bits
$-512\ldots-257$	3	8	**000** $+$ (VAL $+ 512$) encoded as 8 bits
$-256\ldots-129$	4	7	**1001** $+$ (VAL $+ 256$) encoded as 7 bits
$-128\ldots-65$	5	6	**11010** $+$ (VAL $+ 128$) encoded as 6 bits
$-64\ldots-32$	5	5	**11011** $+$ (VAL $+ 64$) encoded as 5 bits
$-32\ldots-1$	4	5	**1010** $+$ (VAL $+ 32$) encoded as 5 bits
$0\ldots31$	4	5	**1011** $+$ VAL encoded as 5 bits
$32\ldots63$	5	5	**11100** $+$ (VAL $- 32$) encoded as 5 bits
$64\ldots127$	5	6	**11101** $+$ (VAL $- 64$) encoded as 6 bits
$128\ldots255$	4	7	**1100** $+$ (VAL $- 128$) encoded as 7 bits
$256\ldots511$	3	8	**001** $+$ (VAL $- 256$) encoded as 8 bits
$512\ldots1023$	3	9	**010** $+$ (VAL $- 512$) encoded as 9 bits
$1024\ldots2047$	3	10	**011** $+$ (VAL $- 1024$) encoded as 10 bits
$-\infty\ldots-1025$	5	32	**11110** $+$ ($-1025 -$ VAL) encoded as 32 bits
$2048\ldots\infty$	5	32	**11111** $+$ (VAL $- 2048$) encoded as 32 bits

Table B.8 — Standard Huffman table H

HTOOB	**1**		
VAL	PREFLEN	RANGELEN	Encoding
$-15\ldots-8$	8	3	**11111100** $+$ (VAL $+ 15$) encoded as 3 bits
$-7\ldots-6$	9	1	**111111100** $+$ (VAL $+ 7$) encoded as 1 bits
$-5\ldots-4$	8	1	**11111101** $+$ (VAL $+ 5$) encoded as 1 bits
-3	9	0	**111111101**
-2	7	0	**1111100**
-1	4	0	**1010**
$0\ldots1$	2	1	**00** $+$ VAL encoded as 1 bits
2	5	0	**11010**
3	6	0	**111010**
$4\ldots19$	3	4	**100** $+$ (VAL $- 4$) encoded as 4 bits
$20\ldots21$	6	1	**111011** $+$ (VAL $- 20$) encoded as 1 bits
$22\ldots37$	4	4	**1011** $+$ (VAL $- 22$) encoded as 4 bits
$38\ldots69$	4	5	**1100** $+$ (VAL $- 38$) encoded as 5 bits
$70\ldots133$	5	6	**11011** $+$ (VAL $- 70$) encoded as 6 bits
$134\ldots261$	5	7	**11100** $+$ (VAL $- 134$) encoded as 7 bits
$262\ldots389$	6	7	**111100** $+$ (VAL $- 262$) encoded as 7 bits
$390\ldots645$	7	8	**1111101** $+$ (VAL $- 390$) encoded as 8 bits
$646\ldots1669$	6	10	**111101** $+$ (VAL $- 646$) encoded as 10 bits
$-\infty\ldots-16$	9	32	**111111110** $+$ ($-16 -$ VAL) encoded as 32 bits
$1670\ldots\infty$	9	32	**111111111** $+$ (VAL $- 1670$) encoded as 32 bits
OOB	2		**01**

Table B.9 — Standard Huffman table I

HTOOB	**1**		
VAL	PREFLEN	RANGELEN	Encoding
$-31 \ldots -16$	8	4	**11111100** $+$ (VAL $+ 31$) encoded as 4 bits
$-15 \ldots -12$	9	2	**111111100** $+$ (VAL $+ 15$) encoded as 2 bits
$-11 \ldots -8$	8	2	**11111101** $+$ (VAL $+ 11$) encoded as 2 bits
$-7 \ldots -6$	9	1	**111111101** $+$ (VAL $+ 7$) encoded as 1 bits
$-5 \ldots -4$	7	1	**1111100** $+$ (VAL $+ 5$) encoded as 1 bits
$-3 \ldots -2$	4	1	**1010** $+$ (VAL $+ 3$) encoded as 1 bits
$-1 \ldots 0$	3	1	**010** $+$ (VAL $+ 1$) encoded as 1 bits
$1 \ldots 2$	3	1	**011** $+$ (VAL $- 1$) encoded as 1 bits
$3 \ldots 4$	5	1	**11010** $+$ (VAL $- 3$) encoded as 1 bits
$5 \ldots 6$	6	1	**111010** $+$ (VAL $- 5$) encoded as 1 bits
$7 \ldots 38$	3	5	**100** $+$ (VAL $- 7$) encoded as 5 bits
$39 \ldots 42$	6	2	**111011** $+$ (VAL $- 39$) encoded as 2 bits
$43 \ldots 74$	4	5	**1011** $+$ (VAL $- 43$) encoded as 5 bits
$75 \ldots 138$	4	6	**1100** $+$ (VAL $- 75$) encoded as 6 bits
$139 \ldots 266$	5	7	**11011** $+$ (VAL $- 139$) encoded as 7 bits
$267 \ldots 522$	5	8	**11100** $+$ (VAL $- 267$) encoded as 8 bits
$523 \ldots 778$	6	8	**111100** $+$ (VAL $- 523$) encoded as 8 bits
$779 \ldots 1290$	7	9	**1111101** $+$ (VAL $- 779$) encoded as 9 bits
$1291 \ldots 3338$	6	11	**111101** $+$ (VAL $- 1291$) encoded as 11 bits
$-\infty \ldots -32$	9	32	**111111110** $+$ ($-32 -$ VAL) encoded as 32 bits
$3339 \ldots \infty$	9	32	**111111111** $+$ (VAL $- 3339$) encoded as 32 bits
OOB	2		**00**

Table B.10 — Standard Huffman table J

HTOOB	**1**		
VAL	PREFLEN	RANGELEN	Encoding
$-21 \ldots -6$	7	4	**1111010** $+$ (VAL $+ 21$) encoded as 4 bits
-5	8	0	**11111100**
-4	7	0	**1111011**
-3	5	0	**11000**
$-2 \ldots 1$	2	2	**00** $+$ (VAL $+ 2$) encoded as 2 bits
2	5	0	**11001**
3	6	0	**110110**
4	7	0	**1111100**
5	8	0	**11111101**
$6 \ldots 69$	2	6	**01** $+$ (VAL $- 6$) encoded as 6 bits
$70 \ldots 101$	5	5	**11010** $+$ (VAL $- 70$) encoded as 5 bits
$102 \ldots 133$	6	5	**110111** $+$ (VAL $- 102$) encoded as 5 bits
$134 \ldots 197$	6	6	**111000** $+$ (VAL $- 134$) encoded as 6 bits
$198 \ldots 325$	6	7	**111001** $+$ (VAL $- 198$) encoded as 7 bits
$326 \ldots 581$	6	8	**111010** $+$ (VAL $- 326$) encoded as 8 bits
$582 \ldots 1093$	6	9	**111011** $+$ (VAL $- 582$) encoded as 9 bits
$1094 \ldots 2117$	6	10	**111100** $+$ (VAL $- 1094$) encoded as 10 bits
$2118 \ldots 4165$	7	11	**1111101** $+$ (VAL $- 2118$) encoded as 11 bits
$-\infty \ldots -22$	8	32	**11111110** $+$ ($-22 -$ VAL) encoded as 32 bits
$4166 \ldots \infty$	8	32	**11111111** $+$ (VAL $- 4166$) encoded as 32 bits
OOB	2		**10**

Table B.11 — Standard Huffman table K

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
1	1	0	**0**
$2 \ldots 3$	2	1	**10** $+$ (VAL $- 2$) encoded as 1 bits
4	4	0	**1100**
$5 \ldots 6$	4	1	**1101** $+$ (VAL $- 5$) encoded as 1 bits
$7 \ldots 8$	5	1	**11100** $+$ (VAL $- 7$) encoded as 1 bits
$9 \ldots 12$	5	2	**11101** $+$ (VAL $- 9$) encoded as 2 bits
$13 \ldots 16$	6	2	**111100** $+$ (VAL $- 13$) encoded as 2 bits
$17 \ldots 20$	7	2	**1111010** $+$ (VAL $- 17$) encoded as 2 bits
$21 \ldots 28$	7	3	**1111011** $+$ (VAL $- 21$) encoded as 3 bits
$29 \ldots 44$	7	4	**1111100** $+$ (VAL $- 29$) encoded as 4 bits
$45 \ldots 76$	7	5	**1111101** $+$ (VAL $- 45$) encoded as 5 bits
$77 \ldots 140$	7	6	**1111110** $+$ (VAL $- 77$) encoded as 6 bits
$141 \ldots \infty$	7	32	**1111111** $+$ (VAL $- 141$) encoded as 32 bits

Table B.12 — Standard Huffman table L

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
1	1	0	**0**
2	2	0	**10**
$3 \ldots 4$	3	1	**110** $+$ (VAL $- 3$) encoded as 1 bits
5	5	0	**11100**
$6 \ldots 7$	5	1	**11101** $+$ (VAL $- 6$) encoded as 1 bits
$8 \ldots 9$	6	1	**111100** $+$ (VAL $- 8$) encoded as 1 bits
10	7	0	**1111010**
$11 \ldots 12$	7	1	**1111011** $+$ (VAL $- 11$) encoded as 1 bits
$13 \ldots 16$	7	2	**1111100** $+$ (VAL $- 13$) encoded as 2 bits
$17 \ldots 24$	7	3	**1111101** $+$ (VAL $- 17$) encoded as 3 bits
$25 \ldots 40$	7	4	**1111110** $+$ (VAL $- 25$) encoded as 4 bits
$41 \ldots 72$	8	5	**11111110** $+$ (VAL $- 41$) encoded as 5 bits
$73 \ldots \infty$	8	32	**11111111** $+$ (VAL $- 73$) encoded as 32 bits

Table B.13 — Standard Huffman table M

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
1	1	0	**0**
2	3	0	**100**
3	4	0	**1100**
4	5	0	**11100**
$5 \ldots 6$	4	1	**1101** $+$ (VAL $- 5$) encoded as 1 bits
$7 \ldots 14$	3	3	**101** $+$ (VAL $- 7$) encoded as 3 bits
$15 \ldots 16$	6	1	**111010** $+$ (VAL $- 15$) encoded as 1 bits
$17 \ldots 20$	6	2	**111011** $+$ (VAL $- 17$) encoded as 2 bits
$21 \ldots 28$	6	3	**111100** $+$ (VAL $- 21$) encoded as 3 bits
$29 \ldots 44$	6	4	**111101** $+$ (VAL $- 29$) encoded as 4 bits
$45 \ldots 76$	6	5	**111110** $+$ (VAL $- 45$) encoded as 5 bits
$77 \ldots 140$	7	6	**1111110** $+$ (VAL $- 77$) encoded as 6 bits
$141 \ldots \infty$	7	32	**1111111** $+$ (VAL $- 141$) encoded as 32 bits

Table B.14 — Standard Huffman table N

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
-2	3	0	**100**
-1	3	0	**101**
0	1	0	**0**
1	3	0	**110**
2	3	0	**111**

Table B.15 — Standard Huffman table O

HTOOB	**0**		
VAL	PREFLEN	RANGELEN	Encoding
$-24 \ldots -9$	7	4	**1111100** $+$ $(\text{VAL} + 24)$ encoded as 4 bits
$-8 \ldots -5$	6	2	**111100** $+$ $(\text{VAL} + 8)$ encoded as 2 bits
$-4 \ldots -3$	5	1	**11100** $+$ $(\text{VAL} + 4)$ encoded as 1 bits
-2	4	0	**1100**
-1	3	0	**100**
0	1	0	**0**
1	3	0	**101**
2	4	0	**1101**
$3 \ldots 4$	5	1	**11101** $+$ $(\text{VAL} - 3)$ encoded as 1 bits
$5 \ldots 8$	6	2	**111101** $+$ $(\text{VAL} - 5)$ encoded as 2 bits
$9 \ldots 24$	7	4	**1111101** $+$ $(\text{VAL} - 9)$ encoded as 4 bits
$-\infty \ldots -25$	7	32	**1111110** $+$ $(-25 - \text{VAL})$ encoded as 32 bits
$25 \ldots \infty$	7	32	**1111111** $+$ $(\text{VAL} - 25)$ encoded as 32 bits

Annex C
(normative)
Gray-scale Image Decoding Procedure

C.1 General description

This decoding procedure is used by the halftone region decoding procedure to produce an array of gray-scale values, which are then used as indexes into a dictionary of halftone patterns.

C.2 Input parameters

The parameters to this decoding procedure are shown in Table C.1.

Table C.1 — Parameters for the gray-scale image decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GSMMR	Integer	1	N	Specifies whether MMR is used.
GSUSESKIP	Integer	1	N	Specifies whether skipping of gray-scale values may occur.
GSBPP	Integer	6	N	The number of bits per gray-scale value.
GSW	Integer	32	N	The width of the gray-scale image.
GSH	Integer	32	N	The height of the gray-scale image.
GSTEMPLATE	Integer	2	N	The template used to code the gray-scale bitplanes. **
GSKIP	Bitmap			A mask indicating which values should be skipped. **GSW** pixels wide, **GSH** pixels high. *

* Unused if **GSUSESKIP** $=$ **0**.
** Unused if **GSMMR** $=$ **1**.

C.3 Return values

The variables whose values are the result of this decoding procedure are shown in Table C.2.

Table C.2 — Return values from the gray-scale image decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GSVALS	Array			The decoded gray-scale values bitmap. The array is **GSW** wide, **GSH** high.

C.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table C.3.

Table C.3 — Variables used in the gray-scale image decoding procedure.

Name	Type	Size (bits)	Signed?	Description and restrictions
GSPLANES	Array of bitmaps			Bitplanes of the gray-scale image. There are **GSBPP** bitplanes in GSPLANES. Each bitplane is **GSW** pixels wide, **GSH** pixels high.
j	Integer	32	Y	Bitplane counter

C.4.1 Decoding the gray-scale image

The gray-scale image is obtained by decoding **HBPP** bitplanes. These bitplanes are denoted (from least significant to most significant) GSPLANES$[0]$, GSPLANES$[1]$, …, GSPLANES[**GSBPP** $-$ $1]$ The bitplanes are Gray-coded, so that each bitplane's true value is equal to its coded value XORed with the next-more-significant bitplane.

The gray-scale image is obtained by the following procedure:

1. Decode GSPLANES[**HBPP** $-\,1]$ using the generic region decoding procedure. The parameters to the generic region decoding procedure are as shown in Table C.4.

Table C.4 — Parameters used to decode a bitplane of the gray-scale image.

Name	Value
MMR	**GSMMR**
GBW	**GSW**
GBH	**GSH**
GBTEMPLATE	**GSTEMPLATE**
TPON	**0**
USESKIP	**GSUSESKIP**
SKIP	**GSKIP**
GBATX$_1$	3 if **GSTEMPLATE** ≤ 1; 2 if **GSTEMPLATE** ≥ 2.
GBATY$_1$	-1
GBATX$_2$	-3
GBATY$_2$	-1
GBATX$_3$	2
GBATY$_3$	-2
GBATX$_4$	-2
GBATY$_4$	-2

2. Set $j =$ **GSBPP** $- 2$.
3. While $j \geq 0$, perform the following steps.
   (a)
   (b) Decode GSPLANES$[j]$ using the generic region decoding procedure. The parameters to the generic region decoding procedure are as shown in Table C.4.
   (c) For each pixel (x, y) in GSPLANES$[j]$, set

$$\text{GSPLANES}[j][x,y] = \text{GSPLANES}[j+1][x,y] \text{ XOR GSPLANES}[j][x,y]$$

   (d) Set $j = j - 1$.
4. For each (x, y), set

$$\text{GSVALS}[x,y] = \sum_{j=0}^{\textbf{GSBPP}-1} \text{GSPLANES}[j][x,y] \times 2^j$$

Annex D
(informative)
Profiles and Suggested Minimum Parameters for Free Parameters

It is recommended that a JBIG2 decoder either implement the entire specification, or one of the profiles described in Table D.1.

Table D.1 — Profile descriptions

Profile		**Simple**	**Medium**	**High**
Requirements		Maximum Speed	Medium Complexity and Medium Compression	Maximum Compression
Generic and symbol region coding	Direct Template	No template (use MMR)	10 pixel (0 AT) or 13 pixel (1 AT)	16 pixel (4 AT)
	Refinement Template	No refinement	10 pixel (0 AT)	13 pixel (2 AT)
Halftone coding		Not available	HENABLESKIP = 0	No restriction
Numerical data		Huffman (Fixed table)	Huffman (Adaptive table) or Arithmetic only	Arithmetic
Resources required		Stand-alone	Laptop computer	Desktop computer
Application examples		Low-end fax; high-speed printing	WWW	Archiving; High-end fax; Wireless WWW

Annex E
(normative)
Arithmetic Coding

An adaptive binary arithmetic coder may be used as the entropy coder when allowed by the models. The models used with adaptive binary arithmetic coding are defined in 6.2, 6.3 and A. In this Annex the basic arithmetic coding procedures are defined.

In this Annex and all of its subclauses, the flow charts and tables are normative only in the sense that they are defining an output that alternative implementations must duplicate. In E.3.8 a simple test example is given which should be helpful in determining if a given implementation is correct.

E.1 Binary encoding

Figure E.1 shows a simple block diagram of the binary adaptive arithmetic encoder. The decision (D) and context (CX) pairs are processed together to produce compressed data (CD) output. Both D and CX are provided by the model unit (not shown). CX selects the probability estimate to use during the coding of D. In this International Standard, CX is a label for a context, formed by some character string followed by a string of bits.

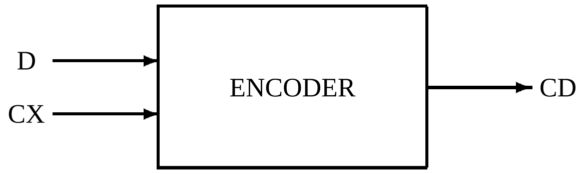


Figure E.1 — Arithmetic encoder inputs and outputs.

E.1.1 Recursive interval subdivision

The recursive probability interval subdivision of Elias coding is the basis for the binary arithmetic coding process. With each binary decision the current probability interval is subdivided into two sub-intervals, and the code stream is modified (if necessary) so that it points to the base (the lower bound) of the probability sub-interval assigned to the symbol which occurred.

In the partitioning of the current interval into two sub-intervals, the sub-interval for the more probable symbol (MPS) is ordered above the sub-interval for the less probable symbol (LPS). Therefore, when the MPS is coded, the LPS sub-interval is added to the code stream. This coding convention requires that symbols be recognized as either MPS or LPS, rather than 0 or 1. Consequently, the size of the LPS interval and the sense of the MPS for each decision must be known in order to code that decision.

Since the code stream always points to the base of the current interval, the decoding process is a matter of determining, for each decision, which sub-interval is pointed to by the code string. This is also done recursively, using the same interval sub-division process as in the encoder. Each time a decision is decoded, the decoder subtracts any interval the encoder added to the code stream. Therefore, the code stream in the decoder is a pointer into the current interval relative to the base of the current interval. Since the coding process involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can often be coded at a cost of much less than one bit per decision.

E.1.2 Coding conventions and approximations

The coding operations are done using fixed precision integer arithmetic and using an integer representation of fractional values in which `0x8000` is equivalent to decimal 0.75. The interval A is kept in the range $0.75 \leq \mathrm{A} < 1.5$ by doubling it whenever the integer value falls below `0x8000`.

The code register C is also doubled each time A is doubled. Periodically - to keep C from overflowing - a byte of data is removed from the high order bits of the C-register and placed in an external code string buffer. Carry-over into the external buffer is resolved by a bit stuffing procedure.

Keeping A in the range $0.75 \leq \mathrm{A} < 1.5$ allows a simple arithmetic approximation to be used in the interval subdivision. Normally, if the interval is A and the current estimate of the LPS probability is Qe, a precise calculation of the sub-intervals would require:

$$\begin{aligned} A - (Qe \times A) &= \text{sub-interval for the MPS} \\ Qe \times A &= \text{sub-interval for the LPS} \end{aligned}$$

Because the value of A is of order unity, these are approximated by

$$\begin{aligned} A - Qe &= \text{sub-interval for the MPS} \\ Qe &= \text{sub-interval for the LPS} \end{aligned}$$

Whenever the MPS is coded, the value of Qe is added to the code register and the interval is reduced to A – Qe. Whenever the LPS is coded, the code register is left unchanged and the interval is reduced to Qe. The precision range required for A is then restored, if necessary, by renormalization of both A and C.

With the process sketched above, the approximations in the interval subdivision process can sometimes make the LPS sub-interval larger than the MPS sub-interval. If, for example, the value of Qe is 0.5 and A is at the minimum allowed value of 0.75, the approximate scaling give $1/3$rd of the interval to the MPS and $2/3$rds to the LPS. To avoid this size inversion, the MPS and LPS intervals are exchanged whenever the LPS interval is larger than the MPS interval. This MPS/LPS conditional exchange can only occur when a renormalization will be needed.

Whenever a renormalization occurs, a probability estimation process is invoked which determines a new probability estimate for the context currently being coded. No explicit symbol counts are needed for the estimation. The relative probabilities of renormalization after coding an LPS and MPS provide an approximate symbol counting mechanism which is used to directly estimate the probabilities.

E.2 Description of the arithmetic encoder

The ENCODER (Figure E.2) initializes the encoder through the INITENC procedure. CX and D pairs are read and passed on to ENCODE until all pairs have been read. The probability estimation procedures which provide adaptive estimates of the probability for each context are imbedded in ENCODE. Bytes of compressed data are output when no longer modifiable. When all of the CX and D pairs have been read (Finished?), FLUSH sets the contents of the C-register to as many 1-bits as possible and then outputs the final bytes. FLUSH also prepares the code string for the addition of a terminating marker code at the end of the stripe.

E.2.1 Encoder code register conventions

The flow charts given in this subclause assume the following register structures for the encoder:

```
            MSB                                LSB
C-register  0000cbbb  bbbbbsss  xxxxxxxx  xxxxxxxx
A-register  00000000  00000000  aaaaaaaa  aaaaaaaa
```

The ”a” bits are the fractional bits in the A-register (the current interval value) and the ”x” bits are the fractional bits in the code register. The ”s” bits are spacer bits which provide useful constraints on carry-over, and the ”b” bits indicate the bit positions from which the completed bytes of the data are removed from the C-register. The ”c” bit is a carry bit.

The detailed description of bit stuffing and the handling of carry-over will be given in a later part of this Annex.

E.2.2 ENCODE

The ENCODE procedure determines whether the decision D is a 0 or not. Then a CODE0 or a CODE1 procedure is called appropriately. Often embodiments will not have an ENCODE procedure, but will call the CODE0 or CODE1 procedures directly to code a 0-decision or a 1-decision.

E.2.3 CODE1 and CODE0

When a given binary decision is coded, one of two possibilities occurs - the symbol is either the more probable symbol or it is the less probable symbol. CODE1 and CODE0 are sketched in Figures E.4 and E.5. In these Figures CX is the context index — the index to the probability estimate which is to be used in the coding operations. MPS(CX) is the sense of the MPS for context CX.

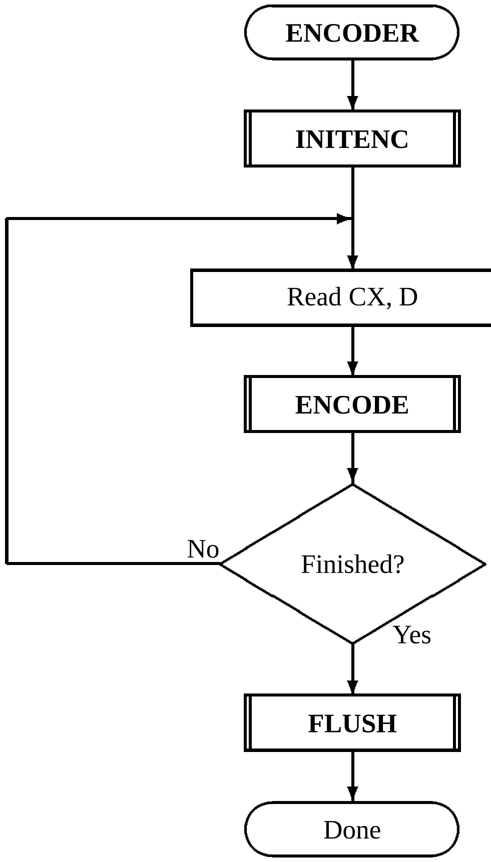


Figure E.2 — Encoder for the MQ-coder.

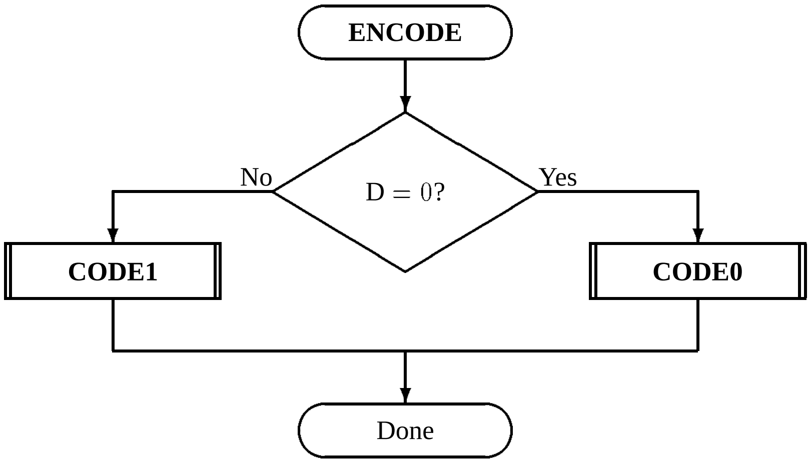


Figure E.3 — ENCODE procedure.

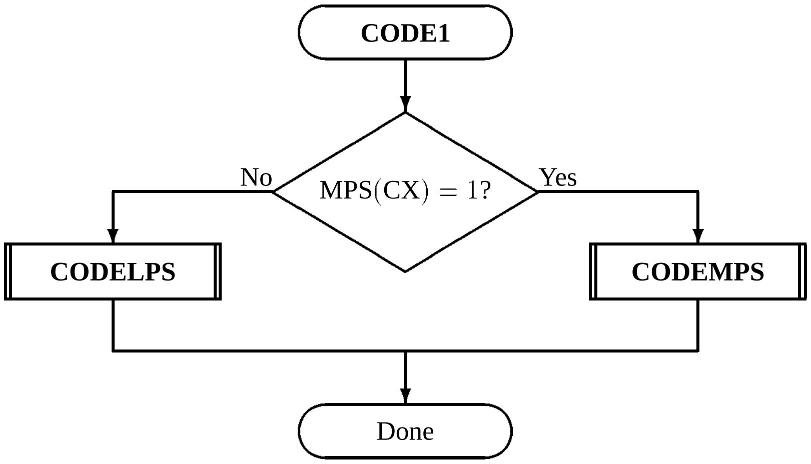


Figure E.4 — CODE1 procedure.

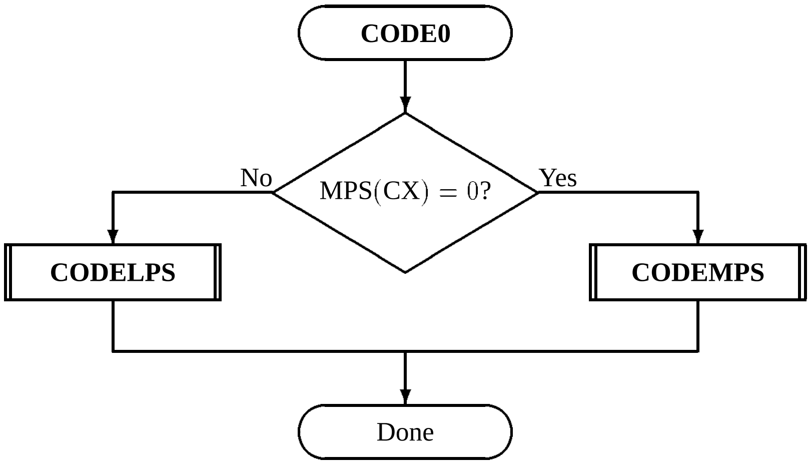


Figure E.5 — CODE0 procedure.

E.2.4 CODEMPS and CODELPS

The CODELPS (Figure E.6) procedure normally consists of a scaling of the interval to $\mathrm{Qe}(\mathrm{I}(\mathrm{CX}))$, the probability estimate of the LPS determined from the index I stored for context CX. The upper interval is first calculated so it can be compared to the lower interval to confirm that Qe has the smaller size. It is always followed by a renormalization (RENORME). In the event that the interval sizes are inverted, however, the conditional MPS/LPS exchange occurs and the upper interval is coded. In either case, the probability estimate is updated. If the SWITCH flag for the index I(CX) is set, then the MPS(CX) is inverted. A new index I is saved at CX as determined from the next LPS index (NLPS) column in Table E.1.

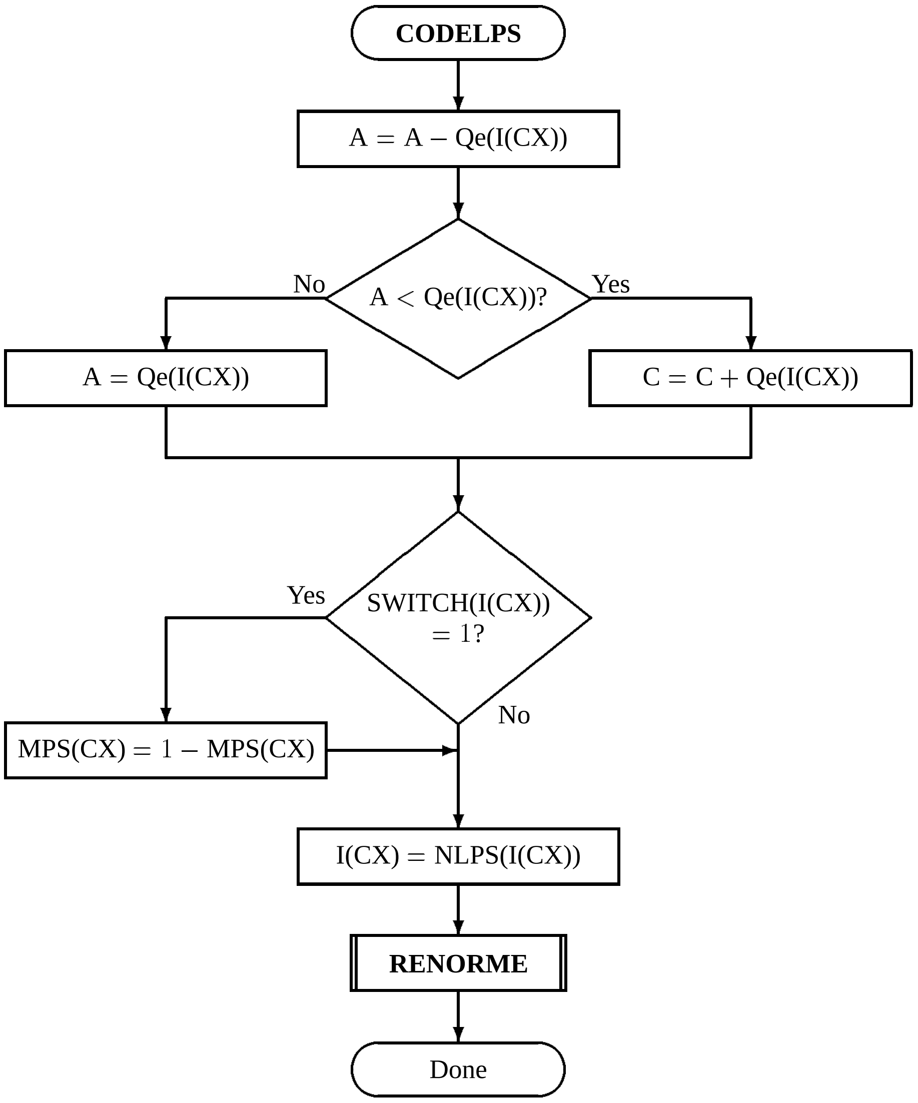


Figure E.6 — CODELPS procedure with conditional MPS/LPS exchange.

The CODEMPS (Figure E.7) procedure normally reduces the size of the interval to the MPS subinterval and adjusts the code register so that it points to the base of the MPS sub-interval. However, if the interval sizes are inverted, the LPS sub-interval is coded instead. Note that the size inversion cannot occur unless a renormalization (RENORME) is required after the coding of the symbol. The probability estimate update changes the index I(CX) according to the next MPS index (NMPS) column in Table E.1.

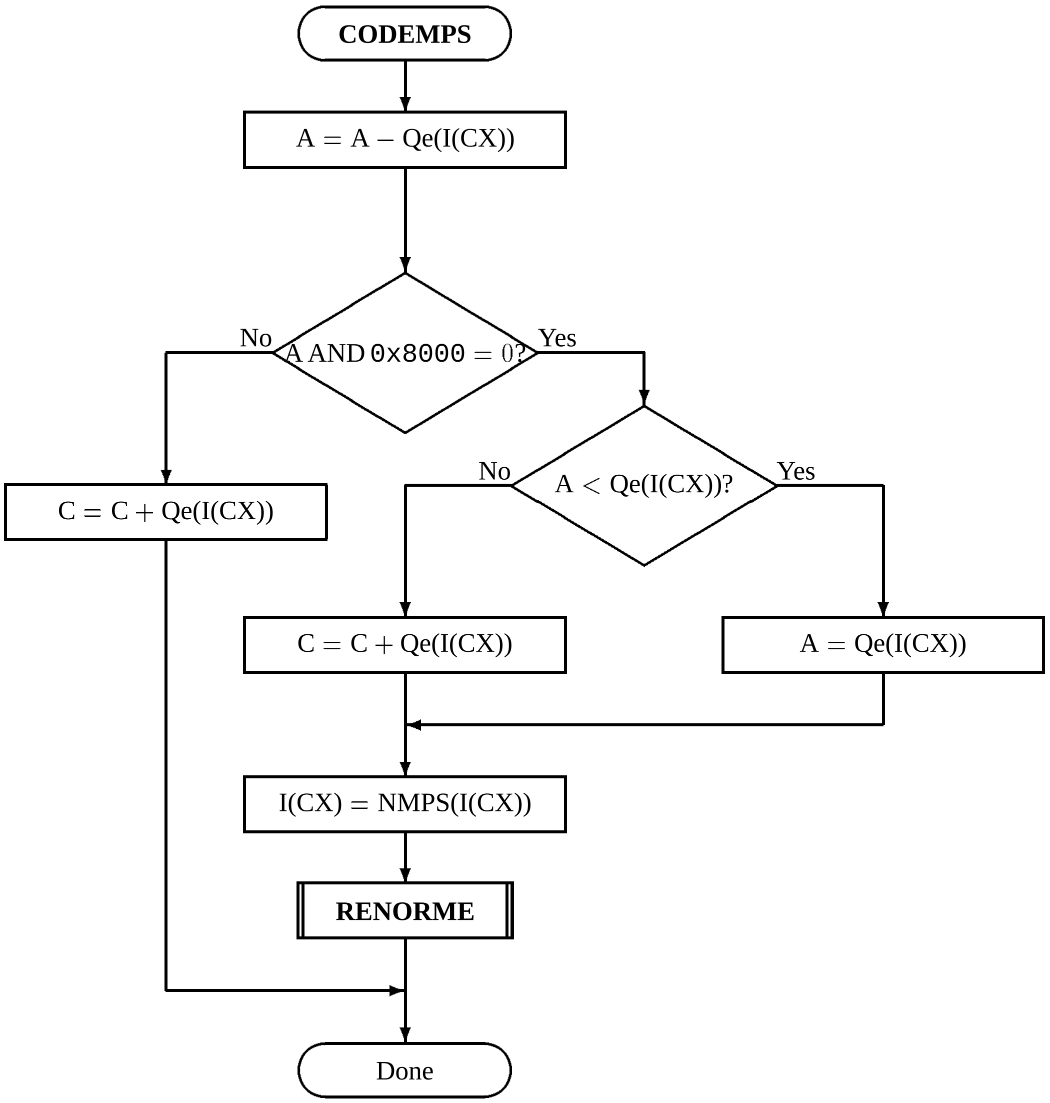


Figure E.7 — CODEMPS procedure with conditional MPS/LPS exchange.

E.2.5 Probability Estimation

Table E.1 shows the Qe value associated with each Qe index. The Qe values are expressed as a hexadecimal integers, as binary integers, and as decimal fractions. To convert the 15 bit integer representation of Qe to the decimal probability, the Qe values are divided by $(4/3) \times (\texttt{0x8000})$.

The estimator can be defined as a finite-state machine — a table of Qe indexes and associated next states for each type of renormalization (i.e., new table positions) - as shown in Table E.1. The change in state occurs only when the arithmetic coder interval register is renormalized. This must always be done after coding the LPS, and whenever the interval register is less than `0x8000` (0.75 in decimal notation) after coding the MPS.

After an LPS renormalization, NLPS gives the new index for the LPS probability estimate. After an MPS renormalization, NMPS gives the new index for the MPS probability estimate. If Switch is 1, the MPS symbol sense must be reversed.

The index to the current estimate is part of the information stored for context CX. This index is used as the index to the table of values in NMPS, which gives the next index for an MPS renormalization. This index is saved in the context storage at CX. MPS(CX) does not change.

The procedure for estimating the probability on the LPS renormalization path is similar to that of an MPS renormalization, except that when Switch(I(CX)) is 1, the sense of MPS(CX) must be inverted.

The final index state 46 can be used to establish a fixed 0.5 probability estimate.

E.2.6 Renormalization in the encoder

Renormalization is very similar in both encoder and decoder, except that in the encoder it generates compressed bits and in the decoder it consumes compressed bits.

The RENORME procedure for the encoder renormalization is sketched in Figure E.8. Both the interval register A and the code register C are shifted, one bit at a time. The number of shifts is counted in the counter CT, and when CT is counted down to zero, a byte of compressed data is removed from C by the procedure BYTEOUT. Renormalization continues until A is no longer less than 0x8000.

E.2.7 The BYTEOUT procedure

The BYTEOUT routine called from RENORME is sketched in Figure E.9. This routine contains the bit-stuffing procedures which are needed to limit carry propagation into the completed bytes of coded data. The conventions used make it impossible for a carry to propagate through more than the byte most recently written to the code buffer.

The procedure in the block in the lower right section does bit stuffing after a `0xFF` byte; the similar procedure on the left is for the case where bit stuffing is not needed.

B is the byte pointed to by the code buffer pointer BP. If B is not a `0xFF`, the carry bit is checked. If the carry bit is set, it is added to B and B is again checked to see if a bit must be stuffed in the next byte. After the need for bit stuffing has been determined, the appropriate path is chosen, BP is incremented and the new value of B is removed from the code register ”b” bits.

E.2.8 Initialisation of the encoder

The INITENC procedure is used to start the arithmetic coder. The basic steps are shown in Figure E.10.

The interval register and code register are set to their initial values, and the bit counter is set. Setting CT $= 12$ reflects the fact that there are three spacer bits in the register which must be filled before the field from which the bytes are removed is reached. Note that BP always points to the byte preceding the position BPST where the first byte is placed. Therefore, if the preceding byte is a `0xFF`, a spurious bit stuff will occur, but can be compensated for by increasing CT. Note that the default initialization of the statistics bins is MPS $= 0$ and I $= 0$ (i.e Qe=0x5601 or decimal 0.503937).

E.2.9 Termination of coding

The FLUSH procedure shown in Figure E.11 is used to terminate the encoding operations and prepare the code string for the addition of a marker code at the end of a stripe. The procedure guarantees that the `0xFF` prefix to the marker code overlaps the final bits of the coded data. This in turn guarantees that any marker code at the end of the compressed data will be recognized and interpreted before decoding is complete.

The first part of the FLUSH procedure sets as many bits in the C-register to 1 as possible as shown in Figure E.12. The exclusive upper bound for the C-register is the sum of the C-register and the interval register. The

Table E.1 — Qe values and probability estimation process

		Qe_Value				
Index	(hexadecimal)	(binary)	(decimal)	NMPS	NLPS	SWITCH
0	0x5601	**0101011000000001**	0.503937	1	1	1
1	0x3401	**0011010000000001**	0.304715	2	6	0
2	0x1801	**0001100000000001**	0.140650	3	9	0
3	0x0acl	**0000101011000001**	0.063012	4	12	0
4	0x0521	**0000010100100001**	0.030053	5	29	0
5	0x0221	**0000001000100001**	0.012474	38	33	0
6	0x5601	**0101011000000001**	0.503937	7	6	1
7	0x5401	**0101010000000001**	0.492218	8	14	0
8	0x4801	**0100100000000001**	0.421904	9	14	0
9	0x3801	**0011100000000001**	0.328153	10	14	0
10	0x3001	**0011000000000001**	0.281277	11	17	0
11	0x2401	**0010010000000001**	0.210964	12	18	0
12	0x1c01	**0001110000000001**	0.164088	13	20	0
13	0x1601	**0001011000000001**	0.128931	29	21	0
14	0x5601	**0101011000000001**	0.503937	15	14	1
15	0x5401	**0101010000000001**	0.492218	16	14	0
16	0x5101	**0101000100000001**	0.474640	17	15	0
17	0x4801	**0100100000000001**	0.421904	18	16	0
18	0x3801	**0011100000000001**	0.328153	19	17	0
19	0x3401	**0011010000000001**	0.304715	20	18	0
20	0x3001	**0011000000000001**	0.281277	21	19	0
21	0x2801	**0010100000000001**	0.234401	22	19	0
22	0x2401	**0010010000000001**	0.210964	23	20	0
23	0x2201	**0010001000000001**	0.199245	24	21	0
24	0x1c01	**0001110000000001**	0.164088	25	22	0
25	0x1801	**0001100000000001**	0.140650	26	23	0
26	0x1601	**0001011000000001**	0.128931	27	24	0
27	0x1401	**0001010000000001**	0.117212	28	25	0
28	0x1201	**0001001000000001**	0.105493	29	26	0
29	0x1101	**0001000100000001**	0.099634	30	27	0
30	0x0acl	**0000101011000001**	0.063012	31	28	0
31	0x09c1	**0000100111000001**	0.057153	32	29	0
32	0x08a1	**0000100010100001**	0.050561	33	30	0
33	0x0521	**0000010100100001**	0.030053	34	31	0
34	0x0441	**0000010001000001**	0.024926	35	32	0
35	0x02al	**0000001010100001**	0.015404	36	33	0
36	0x0221	**0000001000100001**	0.012474	37	34	0
37	0x0141	**0000000101000001**	0.007347	38	35	0
38	0x0111	**0000000100010001**	0.006249	39	36	0
39	0x0085	**0000000010000101**	0.003044	40	37	0
40	0x0049	**0000000001001001**	0.001671	41	38	0
41	0x0025	**0000000000100101**	0.000847	42	39	0
42	0x0015	**0000000000010101**	0.000481	43	40	0
43	0x0009	**0000000000001001**	0.000206	44	41	0
44	0x0005	**0000000000000101**	0.000114	45	42	0
45	0x0001	**0000000000000001**	0.000023	45	43	0
46	0x5601	**0101011000000001**	0.503937	46	46	0

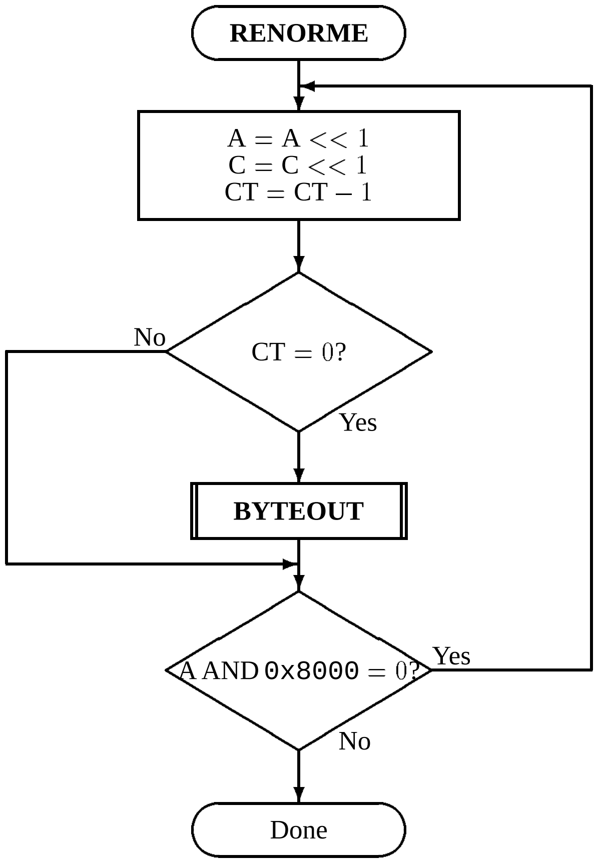


Figure E.8 — Encoder renormalisation procedure.

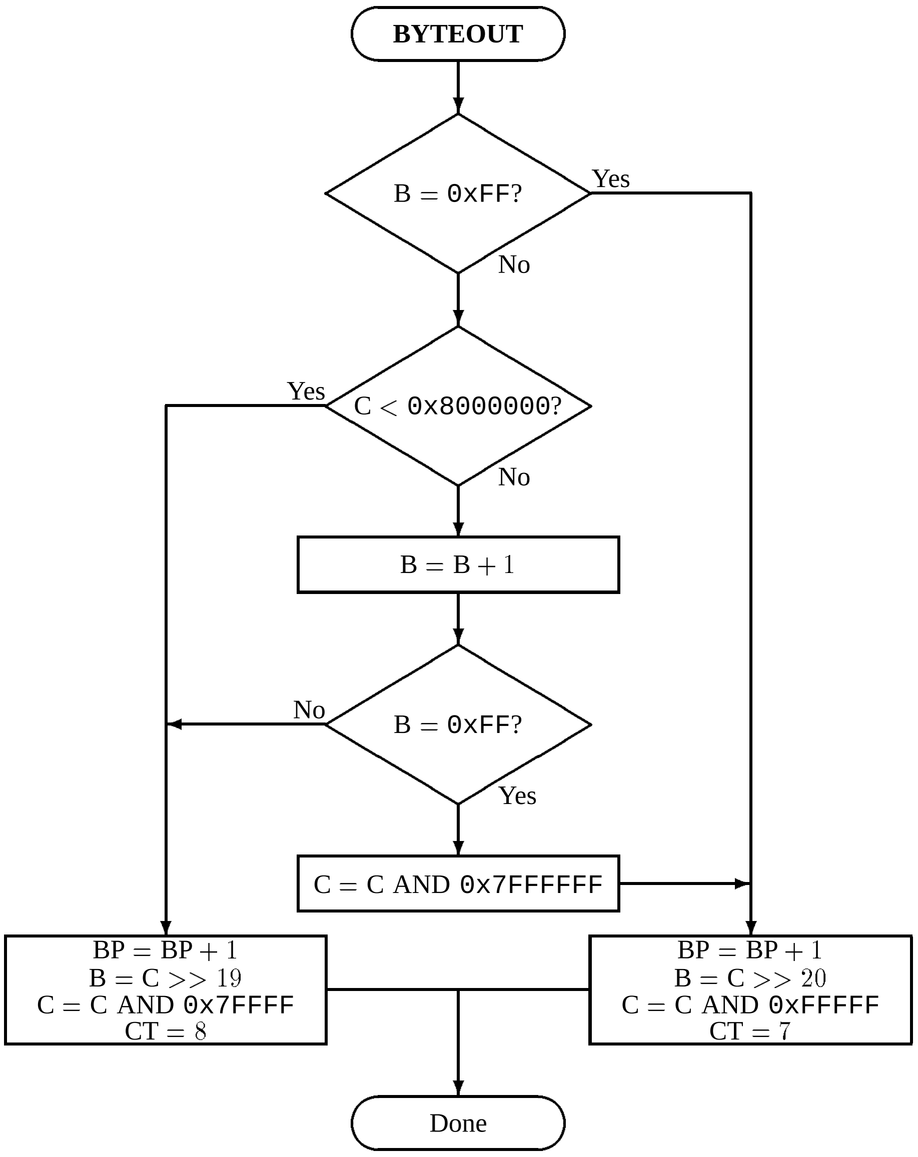


Figure E.9 — BYTEOUT procedure for encoder.

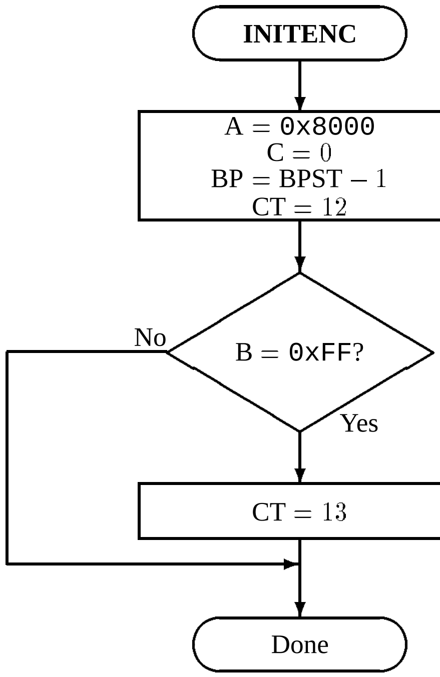


Figure E.10 — Initialisation of the encoder.

low order 16 bits of C are forced to 1, and the result is compared to the upper bound. If C is too big, the leading 1-bit is removed, reducing C to a value which must be within the interval.

The byte in the C-register is then completed by shifting C, and two bytes are then removed. If the second byte is not `0xFF`, another byte is added to the code stream which is guaranteed to be a `0xFF`.

E.2.10 Minimization of the code stream

If desired, the code stream can be truncated after the FLUSH procedure is complete. If a sequence of 1-bits is generated by the arithmetic coder, bit stuffing will produce pairs of `0xFF`, `0x7F` bytes. These byte pairs can be trimmed from the code string, provided that the earliest `0xFF` in the sequence is not removed. This remaining `0xFF` then becomes the prefix to the marker code which terminates the code stream.

Decoding is not affected by this trimming process because the convention is used in the decoder that when a marker code is encountered, 1-bits (without bit stuffing) are supplied to the decoder until the coding interval is complete.

E.3 Description of the arithmetic decoder

Figure E.13 shows a simple block diagram of a binary adaptive arithmetic decoder. The compressed data CD and a context CX from the decoder's model unit (not shown) are input to the arithmetic decoder. The decoder's output is the decision D. The encoder and decoder model units must supply exactly the same context CX for each given decision.

The DECODER (Figure E.14) initializes the decoder through INITDEC. Contexts, CX, and bytes of compressed data (as needed) are read and passed on to DECODE until all contexts have been read. The DECODE routine decodes the binary decision D and returns a value of either 0 or 1. The probability estimation procedures which provide adaptive estimates of the probability for each context are imbedded in DECODE. When all contexts have been read (Finished?), the coded data has been decompressed.

E.3.1 Decoder code register conventions

The flow charts given in this section assume the following register structures for the decoder:

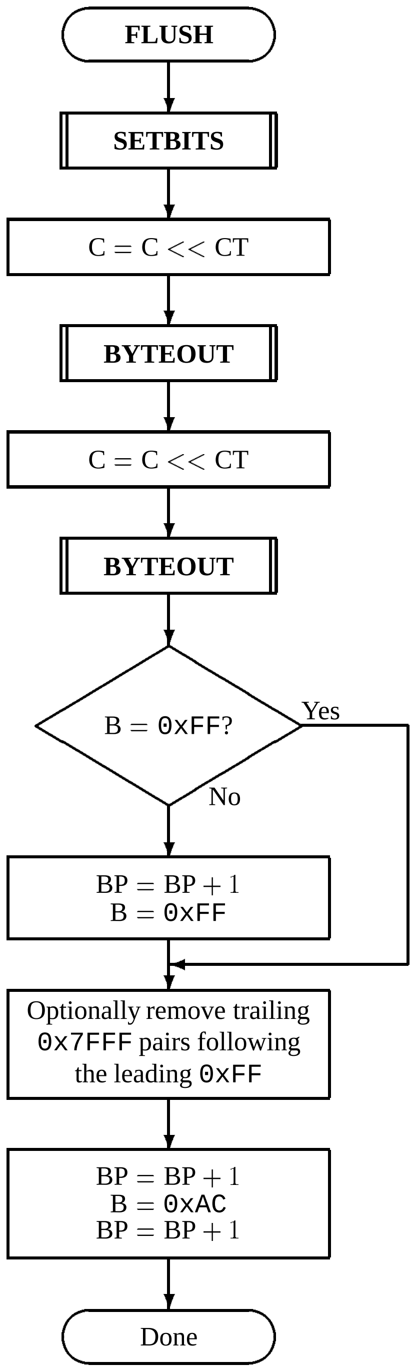


Figure E.11 — FLUSH procedure.

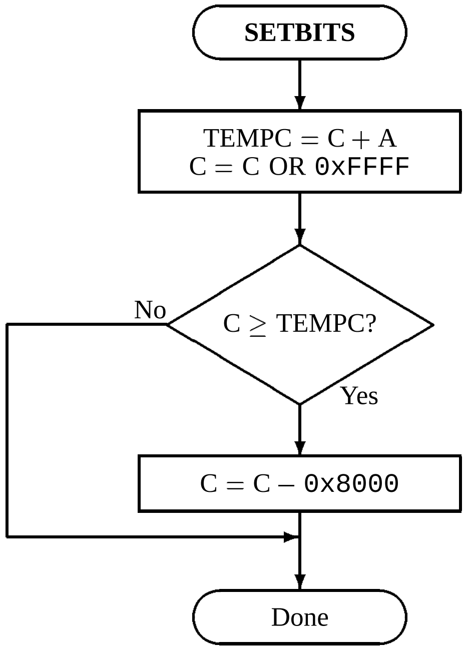


Figure E.12 — Setting the final bits in the C register.

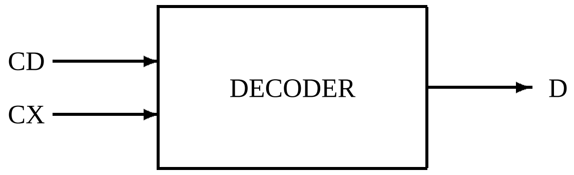


Figure E.13 — Arithmetic decoder inputs and outputs.

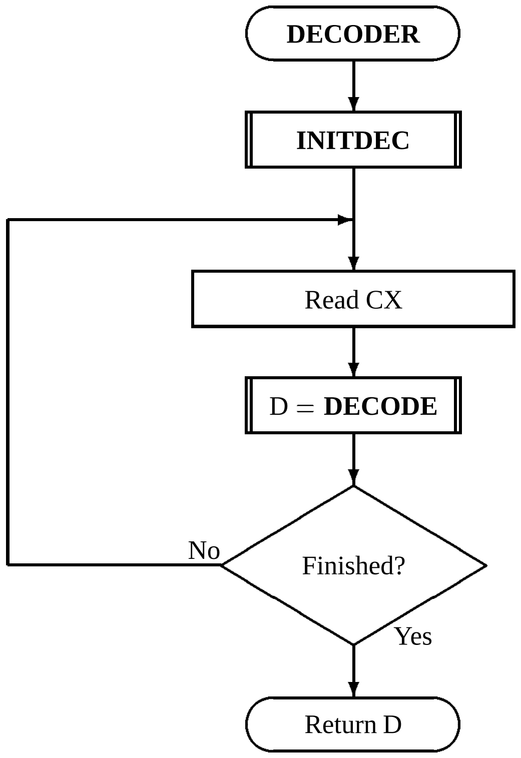


Figure E.14 — Decoder for the MQ-coder.

```
                  15                 0
Chigh register    xxxxxxxx  xxxxxxxx
Clow register     bbbbbbbb  00000000
A-register        aaaaaaaa  aaaaaaaa
```

Chigh and Clow can be thought of as one 32 bit C-register in that renormalization of C shifts a bit of new data from bit 15 of Clow to bit 0 of Chigh. However, the decoding comparisons use Chigh alone. New data is inserted into the ”b” bits of Clow one byte at a time.

The detailed description of the handling of data with stuff-bits will be given later in this section.

Note that the comparisons shown in the various procedures in this section assume precisions greater than 16 bits. Logical comparisons can be used with 16 bit precision.

E.3.2 The Decode procedure

The decoder decodes one binary decision at a time. After decoding the decision, the decoder subtracts any amount from the code string that the encoder added. The amount left in the code string is the offset from the base of the current interval to the sub-interval allocated to all binary decisions not yet decoded. In the first test in the Decode procedure sketched in Figure E.15 the Chigh register is compared to the size of the LPS sub-interval. Unless a conditional exchange is needed, this test determines whether a MPS or LPS is decoded. If Chigh is logically greater than or equal to the LPS probability estimate Qe for the current index I stored at CX, then Chigh is decremented by that amount. If A is not less than `0x8000`, then the MPS sense stored at CX is used to set the decoded decision D.

When a renormalization is needed, the MPS/LPS conditional exchange may have occurred. For the MPS path the conditional exchange procedure is shown in Figure E.16. As long as the MPS sub-interval size A calculated as the first step in Figure E.16 is not logically less than the LPS probability estimate $\mathrm{Qe}(\mathrm{I}(\mathrm{CX}))$, an MPS did occur and the decision can be set from MPS(CX). Then the index I(CX) is updated from the next MPS index (NMPS) column in Table E.1. If, however, the LPS sub-interval is larger, the conditional exchange occurred and an LPS occurred. The probability update switches the MPS sense if the SWITCH column has a ”1” and updates the index I(CX) from the next LPS index (NLPS) column in Table 1. Note that the probability estimation in the decoder must be identical to the probability estimation in the encoder.

For the LPS path of the decoder the conditional exchange procedure is given the LPS_EXCHANGE procedure shown in Figure E.17. The same logical comparison between the MPS sub-interval A and the LPS sub-interval Qe(I(CX)) determines if a conditional exchange occurred. On both paths the new sub-interval A is set to Qe(I(CX)). On the left path the conditional exchange occurred so the decision and update are for the MPS case. On the right path, the LPS decision and update are followed.

E.3.3 Renormalization in the decoder

The RENORMD procedure for the decoder renormalization is sketched in Figure E.18. A counter keeps track of the number of compressed bits in the Clow section of the C-register. When CT is zero, a new byte is inserted into Clow in the BYTEIN procedure.

Both the interval register A and the code register C are shifted, one bit at a time, until A is no longer less than `0x8000`.

E.3.4 The BYTEIN procedure

The BYTEIN procedure called from RENORMD is sketched in Figure E.19. This procedure reads in one byte of data, compensating for any stuff bits following the `0xFF` in the process. It also detects the marker codes which must occur at the end of a scan or resynchronization interval. The C-register in this procedure is the concatenation of the Chigh and Clow registers.

B is the byte pointed to by the code buffer pointer BP. If B is not a `0xFF`, BP is incremented and the new value of B is inserted into the high order 8 bits of Clow.

If B is a `0xFF`, then B1 (the byte pointed to by BP+1) is tested. If B1 exceeds `0x8F`, B1 must be one of the marker codes. The marker code is interpreted as required, and the buffer pointer remains pointed to the `0xFF` prefix of the marker code which terminates the coding interval. 1-bits are then fed to the decoder until the decoding is complete. This is shown by adding `0xFF00` to the C-register and setting the bit counter CT to 8.

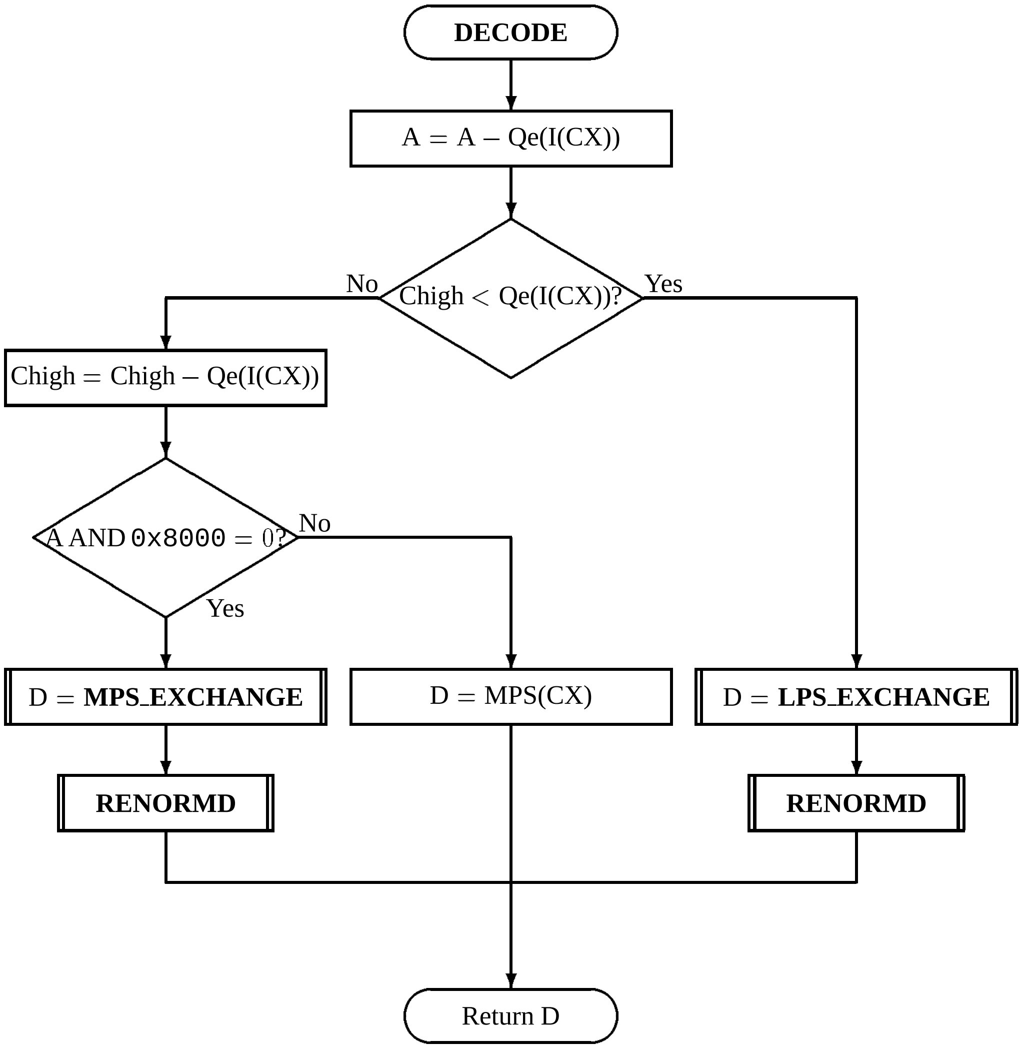


Figure E.15 — Decoding an MPS or an LPS.

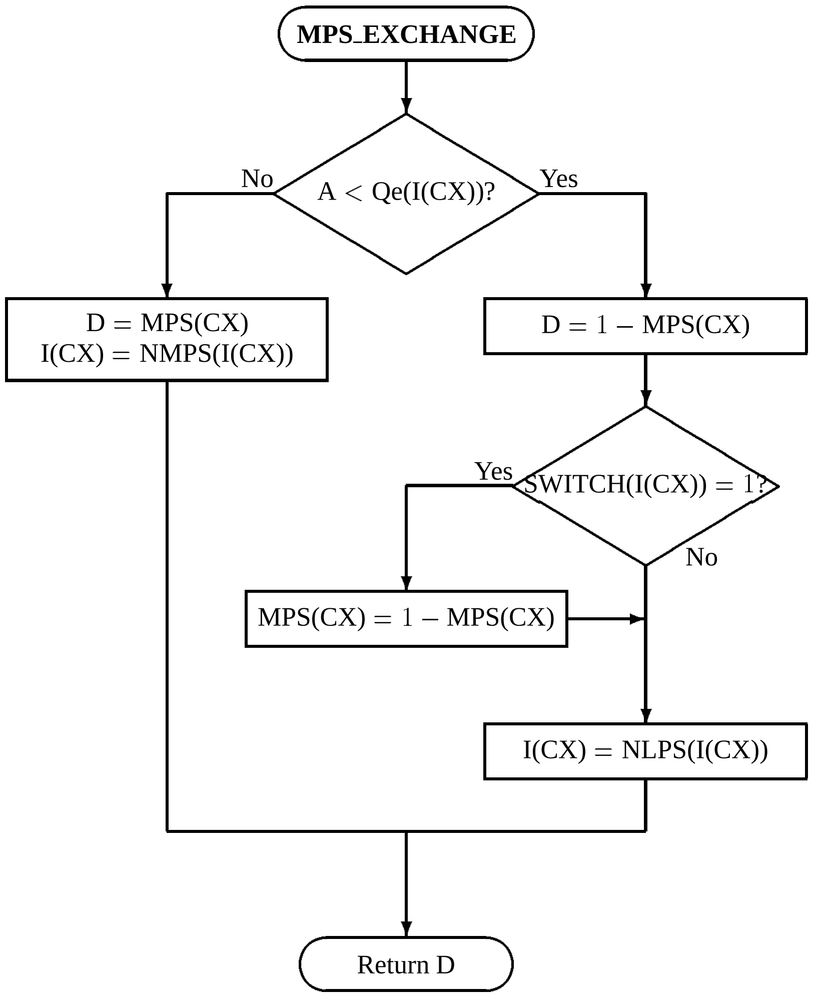


Figure E.16 — Decoder MPS path conditional exchange procedure.

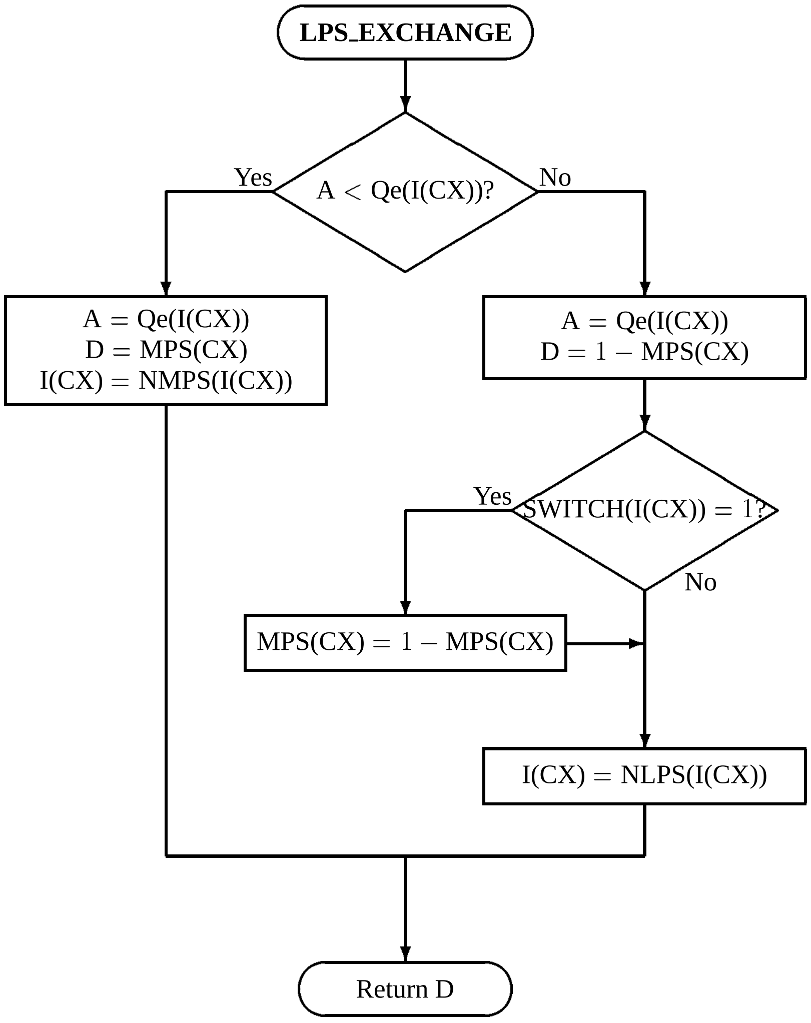


Figure E.17 — Decoder LPS path conditional exchange procedure.

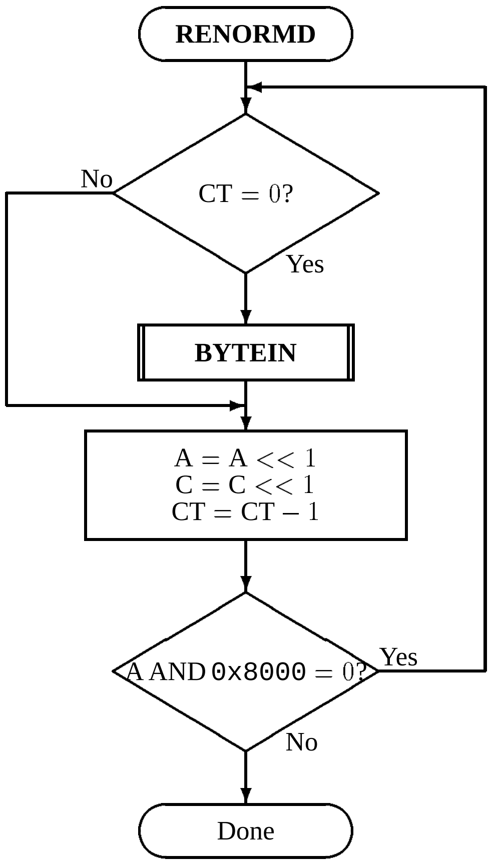


Figure E.18 — Decoder renormalisation procedure.

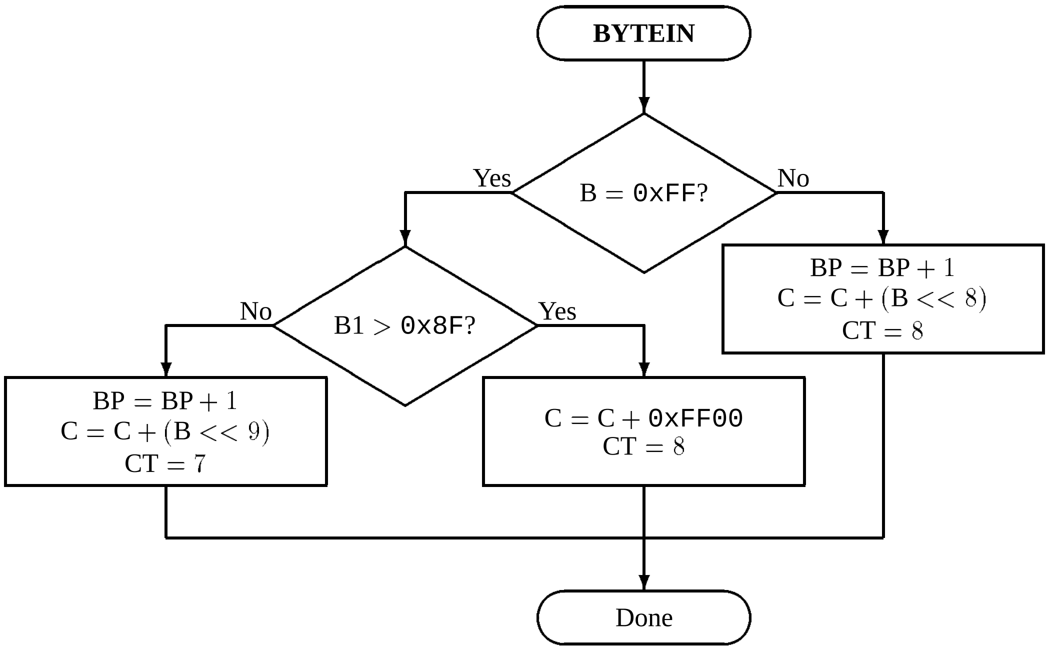


Figure E.19 — BYTEIN procedure for decoder

If B1 is not a marker code, then BP is incremented to point to the next byte which contains a stuffed bit. The B is added to the C-register with an alignment such that the stuff bit (which contains any carry) is added to the low order bit of Chigh.

E.3.5 Initialisation of the decoder

The INITDEC procedure is used to start the arithmetic decoder. The basic steps are shown in Figure E.20.

BP, the pointer to the compressed data, is initialized to BPST (pointing to the first compressed byte). The first byte of the compressed data is shifted into the low order byte of Chigh, and a new byte is then read in. The C-register is then shifted by 7 bits and CT is decremented by 7, bringing the C-register into alignment with the starting value of A. The interval register A is set to match the starting value in the encoder.

E.3.6 Resynchronisation of the decoder

Normally, when the end of the coding interval is reached, the code string pointer BP points to the `0xFF` of the terminating marker code. If for any reason the code string pointer is not at the `0xFF` byte of the marker, an resynchronization procedure needs to scans the code stream until it finds the terminating marker code prefix. If a search of this type is needed, it is indicative of an error condition. This error recovery procedure is not standardized.

E.3.7 Resetting statistics

At certain points during the decoding of a JBIG2 bitstream, some or all of the statistics are reset. This process involves setting I(CX) and MPS(CX) equal to zero for some or all values of CX.

EXAMPLE — At the start of decoding a symbol region segment, all the statistics are reset.

E.3.8 Test sequence

To be supplied.

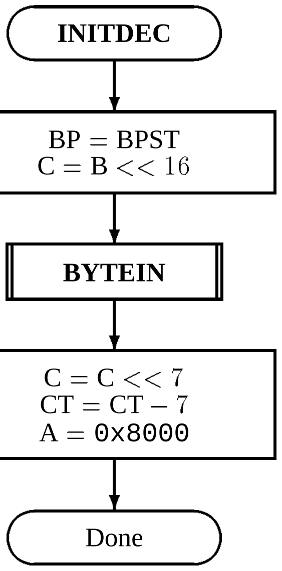


Figure E.20 — Initialisation of the decoder.

Annex F
(informative)
Adaptive Entropy Software-Conventions Decoder

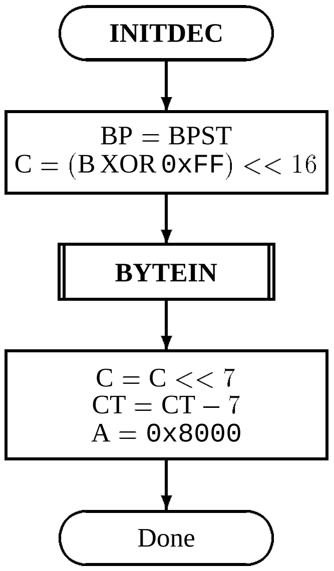


Figure F.1 — Initialisation of the software-conventions decoder.

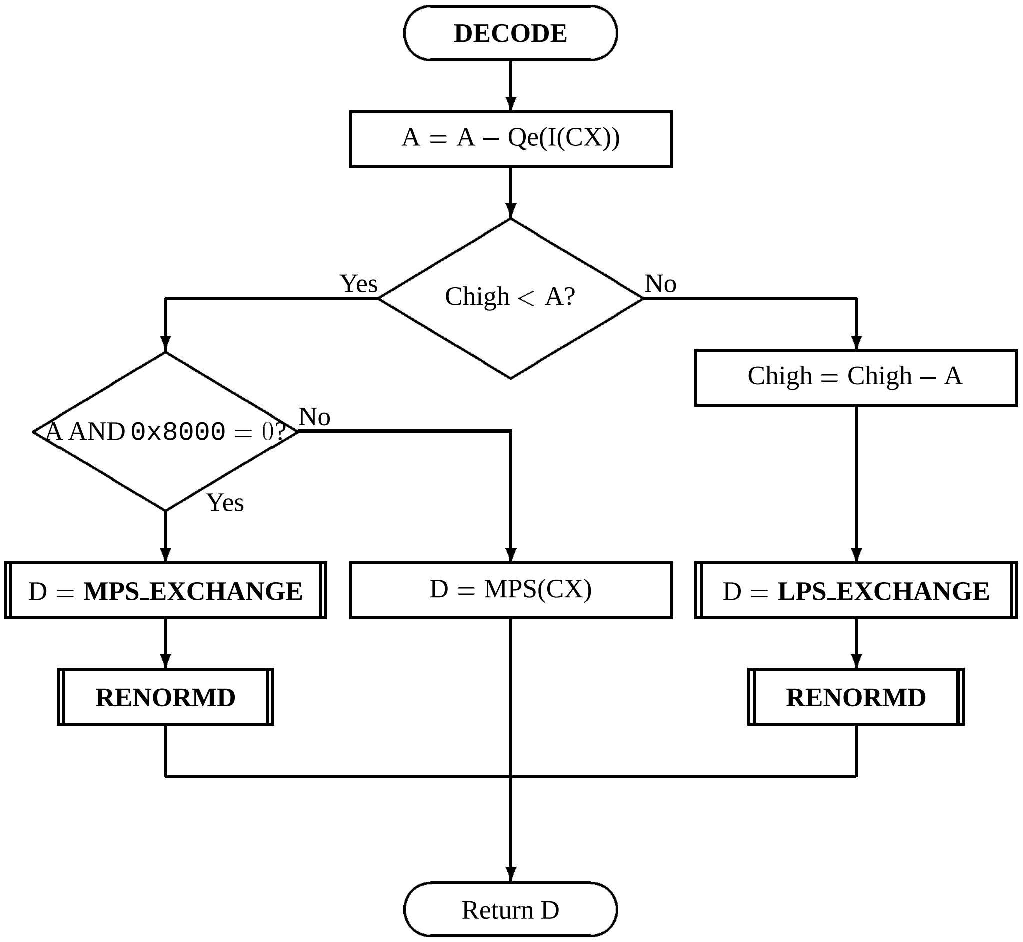


Figure F.2 — Decoding an MPS or an LPS in the software-conventions decoder.

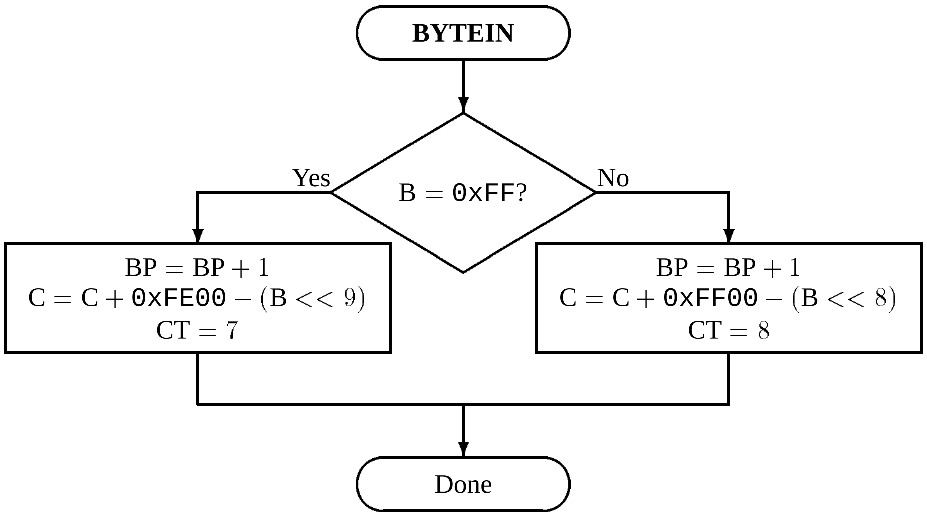


Figure F.3 — Inserting a new byte into the C register in the software-conventions decoder.

Annex G
(normative)
File Formats

There are two standalone file organizations possible for a JBIG2 bitstream. There is also a third organization, not intended for standalone usage, but instead to allow JBIG2-encoded data to be embedded in another file format.

G.1 Sequential organisation

This is a standalone file organisation. In this organisation, the file structure looks like Figure G.1. A file header is followed by a sequence of segments. The two parts of each segment are stored together: first the segment header then the segment data.

File header
Segment 1 header
Segment 1 data
Segment 2 header
Segment 2 data
. . .
Segment N header
Segment N data

Figure G.1 — Sequential organisation

G.2 Random-access organisation

This is a standalone file organisation. In this organisation, the file structure looks like Figure G.2. A file header is follower by a sequence of segments headers; the last segment header is followed by the data for the first segment, then the data for the second segment, and so on.

File header
Segment 1 header
Segment 2 header
. . .
Segment N header
Segment 1 data
Segment 2 data
. . .
Segment N data

Figure G.2 — Random-access organisation

G.3 Embedded organisation

This is not a standalone file organisation, but relies on some other file format to carry the JBIG2 segments. Each segment is stored by concatenating its segment header and segment data parts, but there is no defined storage order for these segments. The embedding file format is allowed to store those segments in any order, and may separate them by arbitrary data.

Applications may wish to precede and follow JBIG2 data with a unique two-byte combination (marker) so that the JBIG2 data can be detected within other data streams. It is suggested to use `0xFF 0xAA` for the starting marker and `0xFF 0xAB` for the ending marker. These markers are not considered to be part of the JBIG2 data. It should be noted that the first byte of a segment header is unlikely to take on the value `0xFF`. Note that the two-byte sequences `0xFF 0xAA` and `0xFF 0xAB` may occur by chance within JBIG2 segments.

NOTE — The intent of the embedded organisation is that many current systems can benefit from incorporating improved bi-level image compression. However, the best way to do this is not always to incorporate an entire JBIG2 bitstream as a monolithic entity, as this can conflict with other constraints. For example, the system might have its own ideas of how pages must be divided up, which might not agree with JBIG2's ideas. Thus, JBIG2 is flexible in allowing the embedding system to store JBIG2 data in whatever way is most convenient.

G.4 File header syntax

A file header contains the following fields, in order.

ID string See G.4.1.

File header flags See G.4.2.

Number of pages See G.4.3.

G.4.1 ID string

This is an 8-byte sequence containing `0x97 0x4A 0x42 0x32 0x0D 0x0A 0x1A 0x0A`.

NOTE — This is similar to the PNG ID string. The first character is nonprintable, so that the file cannot be mistaken for ASCII. The first character's high bit is set, to detect passing through a 7-bit channel. The next three bytes are `JB2`, and are intended to allow a human looking at the header to guess the file type. The following bytes are `CR LF CONTROL-Z LF`; any corruption by CR/LF translation and DOS file truncation can be detected immediately.

G.4.2 File header flags

This is a 1-byte field. The bits that are defined are

Bit 0 File organisation type. If this bit is **0**, the file uses the random-access organisation. If this bit is **1**, the file uses the sequential organisation.

Bit 1 Unknown number of pages. If this bit is **0**, then the number of pages contained in the file is known. If this bit is **1**, then the number of pages contained in the file was not known at the time that the file header was coded.

Bits 2–7 Reserved; must be **0**.

G.4.3 Number of pages

This is a 4-byte field, and is not present if the "unknown number of pages" bit was **1**. If present, it must equal the number of pages contained in the file.

References

[1] R. N. Ascher and G. Nagy. A means for achieving a high degree of compaction on scan-digitized printed text. *IEEE Transactions on Computers*, C-23:1174–1179, November 1974.

[2] A. Bilgin, P. Sementilli, and M. Marcellin. Progressive image coding using trellis coded quantization. *IEEE Transactions on Image Processing*, November 1997. (submitted).

[3] C. Constantinescu and R. Arps. Fast residue coding for lossless textual image compression. In *Proc. of Data Compression Conference*, pages 397–406, Snowbird, Utah, USA, 1997.

[4] S. Forchhammer and K. S. Jensen. Data compression of scanned halftone images. *IEEE Trans. Commun.*, 42:1881–1893, Feb–Apr 1994.

[5] P. G. Howard. Text image compression using soft pattern matching. *Computer Journal*, 40:2–3, 1997.

[6] Paul G. Howard. Lossless and lossy compression of text images by soft pattern matching. In *Proc. of Data Compression Conference*, pages 210–219, 1996.

[7] R. Hunter and A. H. Robinson. International digital facsimile coding standards. In *Proceedings of IEEE*, volume 68, pages 854–867, July 1980.

[8] S. Inglis and I.H. Witten. Compression-based template matching. In J. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, 1994.

[9] F. Kossentini and R. Ward. An analysis-compression technique for black and white documents. In *IEEE Southwest Symposium on Image Analysis and Interpretation*, pages 141–144, San Antonio, TX, USA, April 1996.

[10] B. Martins and S. Forchhammer. Lossless, near-lossless, and refinement coding of bi-level images. *IEEE Trans. Image Processing.* (in press).

[11] B. Martins and S. Forchhammer. Lossless/lossy compression of bi-level images. In *Proc. of IS&T/SPIE Symp. on Electr. Im.: Science and Technology 1997*, volume 3018, pages 38–49, 1997.

[12] B. Martins and S. Forchhammer. Tree Coding of Bi-level Images. *IEEE Trans. Image Proc.*, 7(4):517–528, 1998.

[13] K. Mohiuddin, J. J. Rissanen, and R. Arps. Lossless binary image compression based on pattern matching. In *Proceedings of International Conference on Computers, Systems, and Signal Processing*, pages 447–451, Bangalore, India, 1984.

[14] W. K. Pratt, P. J. Capitant, W. H. Chen, E. R. Hamilton, and R. H. Walls. Combined symbol matching facsimile data compression system. In *Proceedings of the IEEE*, volume 68, pages 786–796, July 1980.

[15] A. Said and W. A. Pearlman. A new fast and efficient image codec based on set partitioning in hierarchical trees. *IEEE Transactions on Circuits and Systems for Video Technology*, 6:243–250, June 1996.

[16] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[17] J. T. Tou and R. C. Gonzalez. *Pattern Recognition Principles*. Addison-Wesley Publishing Company, Reading, MA, 1974.

[18] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Van Nostrand Reinhold, New York, 1994.