INTERNATIONAL TELECOMMUNICATION UNION

# ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

# T.88

(02/2000)

SERIES T: TERMINALS FOR TELEMATIC SERVICES

## Information technology – Lossy/lossless coding of bi-level images

ITU-T Recommendation T.88

(Previously CCITT Recommendation)

**INTERNATIONAL STANDARD ISO/CEI 14492**

**ITU-T RECOMMENDATION T.88**

# INFORMATION TECHNOLOGY – LOSSY/LOSSLESS CODING OF BI-LEVEL IMAGES


## Summary

This Recommendation | International Standard, commonly known as JBIG2, defines a coding method for bi-level images (e.g. black and white printed matter). These are images consisting of a single rectangular bit plane, with each pixel taking on one of just two possible colours. This Recommendation | International Standard has been explicitly prepared for a lossy, lossless, and lossy-to-lossless image compression.



## Source

ITU-T Recommendation T.88 was approved on 10 February 2000. An identical text is also published as ISO/IEC 14492.

# FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

## NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

## INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

# CONTENTS


| | | | Page |
|---|---|---|---|
| 0 | Introduction | | vii |
| | 0.1 | Interpretation and use of the requirements | vii |
| | 0.1.1 | Subject matter for JBIG2 coding | vii |
| | 0.1.2 | Relationship between segments and documents | viii |
| | 0.1.3 | Structure and use of segments | viii |
| | 0.1.4 | Internal representations | viii |
| | 0.1.5 | Decoding results | x |
| | 0.1.6 | Decoding procedures | x |
| | 0.2 | Lossy coding | xi |
| | 0.2.1 | Symbol coding | xi |
| | 0.2.2 | Generic coding | xi |
| | 0.2.3 | Halftone coding | xii |
| | 0.2.4 | Consequences of inadequate segmentation | xii |
| 1 | Scope | | 1 |
| 2 | Normative References | | 1 |
| 3 | Terms and Definitions | | 1 |
| 4 | Symbols and Abbreviations | | 3 |
| | 4.1 | Abbreviations | 3 |
| | 4.2 | Symbol definitions | 4 |
| | 4.3 | Operator definitions | 10 |
| 5 | Conventions | | 10 |
| | 5.1 | Typographic conventions | 10 |
| | 5.2 | Binary notation | 10 |
| | 5.3 | Hexadecimal notation | 11 |
| | 5.4 | Integer value syntax | 11 |
| | 5.4.1 | Bit packing | 11 |
| | 5.4.2 | Multi-byte values | 11 |
| | 5.4.3 | Bit numbering | 11 |
| | 5.4.4 | Signedness | 11 |
| | 5.5 | Array notation and conventions | 11 |
| | 5.6 | Image and bitmap conventions | 11 |
| 6 | Decoding Procedures | | 12 |
| | 6.1 | Introduction to decoding procedures | 12 |
| | 6.2 | Generic region decoding procedure | 13 |
| | 6.2.1 | General description | 13 |
| | 6.2.2 | Input parameters | 13 |
| | 6.2.3 | Return value | 13 |
| | 6.2.4 | Variables used in decoding | 14 |
| | 6.2.5 | Decoding using a template and arithmetic coding | 14 |
| | 6.2.6 | Decoding using MMR coding | 18 |
| | 6.3 | Generic Refinement Region Decoding Procedure | 19 |
| | 6.3.1 | General description | 19 |
| | 6.3.2 | Input parameters | 19 |
| | 6.3.3 | Return value | 19 |
| | 6.3.4 | Variables used in decoding | 20 |
| | 6.3.5 | Decoding using a template and arithmetic coding | 20 |

| | | | Page |
|---|---|---|---|
| 6.4 | Text Region Decoding Procedure | | 23 |
| | 6.4.1 | General description | 23 |
| | 6.4.2 | Input parameters | 23 |
| | 6.4.3 | Return value | 24 |
| | 6.4.4 | Variables used in decoding | 24 |
| | 6.4.5 | Decoding the text region | 25 |
| | 6.4.6 | Strip delta T | 28 |
| | 6.4.7 | First symbol instance S coordinate | 28 |
| | 6.4.8 | Subsequent symbol instance S coordinate | 28 |
| | 6.4.9 | Symbol instance T coordinate | 29 |
| | 6.4.10 | Symbol instance symbol ID | 29 |
| | 6.4.11 | Symbol instance bitmap | 29 |
| 6.5 | Symbol Dictionary Decoding Procedure | | 30 |
| | 6.5.1 | General description | 30 |
| | 6.5.2 | Input parameters | 30 |
| | 6.5.3 | Return value | 30 |
| | 6.5.4 | Variables used in decoding | 30 |
| | 6.5.5 | Decoding the symbol dictionary | 32 |
| | 6.5.6 | Height class delta height | 34 |
| | 6.5.7 | Delta width | 34 |
| | 6.5.8 | Symbol bitmap | 34 |
| | 6.5.9 | Height class collective bitmap | 37 |
| | 6.5.10 | Exported symbols | 37 |
| 6.6 | Halftone Region Decoding Procedure | | 38 |
| | 6.6.1 | General description | 38 |
| | 6.6.2 | Input parameters | 38 |
| | 6.6.3 | Return value | 39 |
| | 6.6.4 | Variables used in decoding | 39 |
| | 6.6.5 | Decoding the halftone region | 39 |
| 6.7 | Pattern Dictionary Decoding Procedure | | 42 |
| | 6.7.1 | General description | 42 |
| | 6.7.2 | Input parameters | 42 |
| | 6.7.3 | Return value | 42 |
| | 6.7.4 | Variables used in decoding | 43 |
| | 6.7.5 | Decoding the pattern dictionary | 43 |
| 7 | Control Decoding Procedure | | 44 |
| 7.1 | General description | | 44 |
| 7.2 | Segment header syntax | | 45 |
| | 7.2.1 | Segment header fields | 45 |
| | 7.2.2 | Segment number | 45 |
| | 7.2.3 | Segment header flags | 45 |
| | 7.2.4 | Referred-to segment count and retention flags | 45 |
| | 7.2.5 | Referred-to segment numbers | 47 |
| | 7.2.6 | Segment page association | 47 |
| | 7.2.7 | Segment data length | 47 |
| | 7.2.8 | Segment header example | 47 |
| 7.3 | Segment types | | 48 |
| | 7.3.1 | Rules for segment references | 49 |
| | 7.3.2 | Rules for page associations | 50 |
| 7.4 | Segment syntaxes | | 50 |
| | 7.4.1 | Region segment information field | 50 |
| | 7.4.2 | Symbol dictionary segment syntax | 51 |
| | 7.4.3 | Text region segment syntax | 56 |
| | 7.4.4 | Pattern dictionary segment syntax | 66 |
| | 7.4.5 | Halftone region segment syntax | 67 |
| | 7.4.6 | Generic region segment syntax | 70 |
| | 7.4.7 | Generic refinement region syntax | 72 |
| | 7.4.8 | Page information segment syntax | 73 |

| | | | Page |
|---|---|---|---|
| | 7.4.9 | End of page segment syntax | 76 |
| | 7.4.10 | End of stripe segment syntax | 76 |
| | 7.4.11 | End of file segment syntax | 76 |
| | 7.4.12 | Profiles segment syntax | 76 |
| | 7.4.13 | Code table segment syntax | 77 |
| | 7.4.14 | Extension segment syntax | 77 |
| | 7.4.15 | Defined extension types | 77 |
| 8 | Page Make-up | | 78 |
| 8.1 | Decoder model | | 78 |
| 8.2 | Page image composition | | 78 |
| Annex A – Arithmetic Integer Decoding Procedure | | | 82 |
| A.1 | General description | | 82 |
| A.2 | Procedure for decoding values (except IAID) | | 82 |
| A.3 | The IAID decoding procedure | | 84 |
| Annex B – Huffman Table Decoding Procedure | | | 86 |
| B.1 | General description | | 86 |
| B.2 | Code table structure | | 86 |
| | B.2.1 | Code table flags | 87 |
| | B.2.2 | Code table lowest value | 87 |
| | B.2.3 | Code table highest value | 87 |
| B.3 | Assigning the prefix codes | | 87 |
| B.4 | Using a Huffman table | | 88 |
| B.5 | Standard Huffman tables | | 89 |
| Annex C – Gray-scale Image Decoding Procedure | | | 97 |
| C.1 | General description | | 97 |
| C.2 | Input parameters | | 97 |
| C.3 | Return value | | 97 |
| C.4 | Variables used in decoding | | 97 |
| C.5 | Decoding the gray-scale image | | 98 |
| Annex D – File Formats | | | 99 |
| D.1 | Sequential organisation | | 99 |
| D.2 | Random-access organisation | | 99 |
| D.3 | Embedded organisation | | 100 |
| D.4 | File header syntax | | 100 |
| | D.4.1 | ID string | 100 |
| | D.4.2 | File header flags | 100 |
| | D.4.3 | Number of pages | 100 |
| Annex E – Arithmetic Coding | | | 101 |
| E.1 | Binary encoding | | 101 |
| | E.1.1 | Recursive interval subdivision | 101 |
| | E.1.2 | Coding conventions and approximations | 101 |
| E.2 | Description of the arithmetic encoder | | 102 |
| | E.2.1 | Encoder code register conventions | 103 |
| | E.2.2 | Encoding a decision (ENCODE) | 103 |
| | E.2.3 | Encoding a 1 or 0 (CODE1 and CODE0) | 103 |
| | E.2.4 | Encoding an MPS or LPS (CODEMPS and CODELPS) | 104 |
| | E.2.5 | Probability estimation | 105 |
| | E.2.6 | Renormalisation in the encoder (RENORME) | 105 |
| | E.2.7 | Compressed data output (BYTEOUT) | 106 |
| | E.2.8 | Initialisation of the encoder (INITENC) | 107 |
| | E.2.9 | Termination of encoding (FLUSH) | 107 |
| | E.2.10 | Minimisation of the compressed data | 107 |

Page

E.3 Arithmetic decoding procedure ........ 109

E.3.1 Decoder code register conventions ........ 111

E.3.2 Decoding a decision (DECODE) ........ 111

E.3.3 Renormalisation in the decoder (RENORMD) ........ 111

E.3.4 Compressed data input (BYTEIN) ........ 111

E.3.5 Initialisation of the decoder (INITDEC) ........ 114

E.3.6 Resynchronisation of the decoder ........ 114

E.3.7 Resetting arithmetic coding statistics ........ 115

E.3.8 Saving arithmetic coding statistics ........ 115

Annex F – Profiles ........ 116

Annex G – Arithmetic Decoding Procedure (Software Conventions) ........ 119

Annex H – Datastream Example and Test Sequence ........ 121

H.1 Datastream example ........ 121

H.2 Test sequence for arithmetic coder ........ 142

Bibliography ........ 147

# 0 Introduction

This Recommendation | International Standard, informally called JBIG2, defines a coding method for bi-level images (e.g. black and white printed matter). These are images consisting of a single rectangular bit plane, with each pixel taking on one of just two possible colours. Multiple colours are to be handled using an appropriate higher level standard such as ITU-T Recommendation T.44. 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.

Compression of this type of image is also addressed by existing facsimile standards, for example by the compression algorithms in ITU-T Recommendations T.4 (MH, MR), T.6 (MMR), 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 is intended to behave as lossless and progressive (lossy-to-lossless) coding. Though it has the capability of lossy coding, the lossy images produced by JBIG1 have significantly lower quality than the original images because the number of pixels in the lossy image cannot exceed one quarter of those in the original image.

On the contrary, JBIG2 was explicitly prepared 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 behaviour. 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.

Although this Recommendation | International Standard is phrased in terms of actions to be taken by decoders to interpret a bitstream, any decoder that produces the correct result (as defined by those actions) is compliant, regardless of the actions it actually takes.

Annexes A, B, C, D, E, and F are normative, and thus form an integral part of this Recommendation | International Standard. Annexes G and H are informative, and thus do not form an integral part of this Recommendation | International Standard.

## 0.1 Interpretation and use of the requirements

This section is informative and designed to aid in interpreting the requirements of this Recommendation | 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 this Recommendation | International Standard in typical applications.

### 0.1.1 Subject matter for JBIG2 coding

JBIG2 is 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, gray-scale or colour multi-level images (e.g. photographs) that have been dithered to produce bi-level images. The periodic bitmap cells in the halftone part of the page are called *patterns* in JBIG2. In addition, a page may contain other data, such as line art and noise. Such non-text, non-halftone 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 text region containing digitised text, a halftone region containing digitised halftones, and a generic region containing the remaining digitised image data, such as line-art. In some circumstances, it is better (in image quality or compressed data size) to consider text or halftones as generic data; conversely, in some circumstances it is better to consider generic data using one of the special cases.

An encoder is permitted to divide a single page into any number of regions, but often three regions will be sufficient, one for textual symbols, one for halftone patterns, and the third for the generic remainder. In some cases, not all types of data may be present, and the page may consist of fewer than three regions.

The various regions may overlap on the physical page. JBIG2 provides the means to specify how the overlapping regions are recombined to form the final page image.

A text region consists of a number of symbols placed at specified locations on a background. The symbols usually correspond to individual text 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 text symbols, indexed so that a symbol bitmap may be referred to by an index number.

A halftone region consists of a number of patterns placed along a regular grid. The patterns usually correspond to gray-scale values. Indeed, the coding method of the pattern indices is designed as a gray-scale coder. Compression can be realised by representing the binary pixels of one grid cell by a single integer, the halftone index (which is usually a rendered gray-scale value). This many-to-one mapping (the pattern in a cell into a gray-scale value) may have the effect that edge information present in the original bitmap is lost by halftone coding. For this reason, lossless or near-lossless coding of halftones will often be better in image quality (though larger in size) 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 text region segment, a pattern 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 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 organised in two ways, sequential or random access. In the sequential organisation, each segment's segment header immediately precedes that segment's data header and data, all in sequential order. In the random access organisation, 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 organisation permits a decoder to determine all segment dependencies without reading the entire file.

A third way of encapsulating of JBIG2-encoded data is to embed it in a non-JBIG2 file – this is sometimes called the *embedded organisation*. 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, at any set of locations within its own structure.

### 0.1.4 Internal representations

Decoded data must be stored before printing or display. While this Recommendation | International 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 text region decoding procedure reads symbols from the symbol memory and draws them into the page buffer or an auxiliary buffer. Although it is not shown in Figure 1, the encoded data stream flows to the decoding procedures, and the block labeled "Page and auxiliary buffers" produces the final decoded page images.

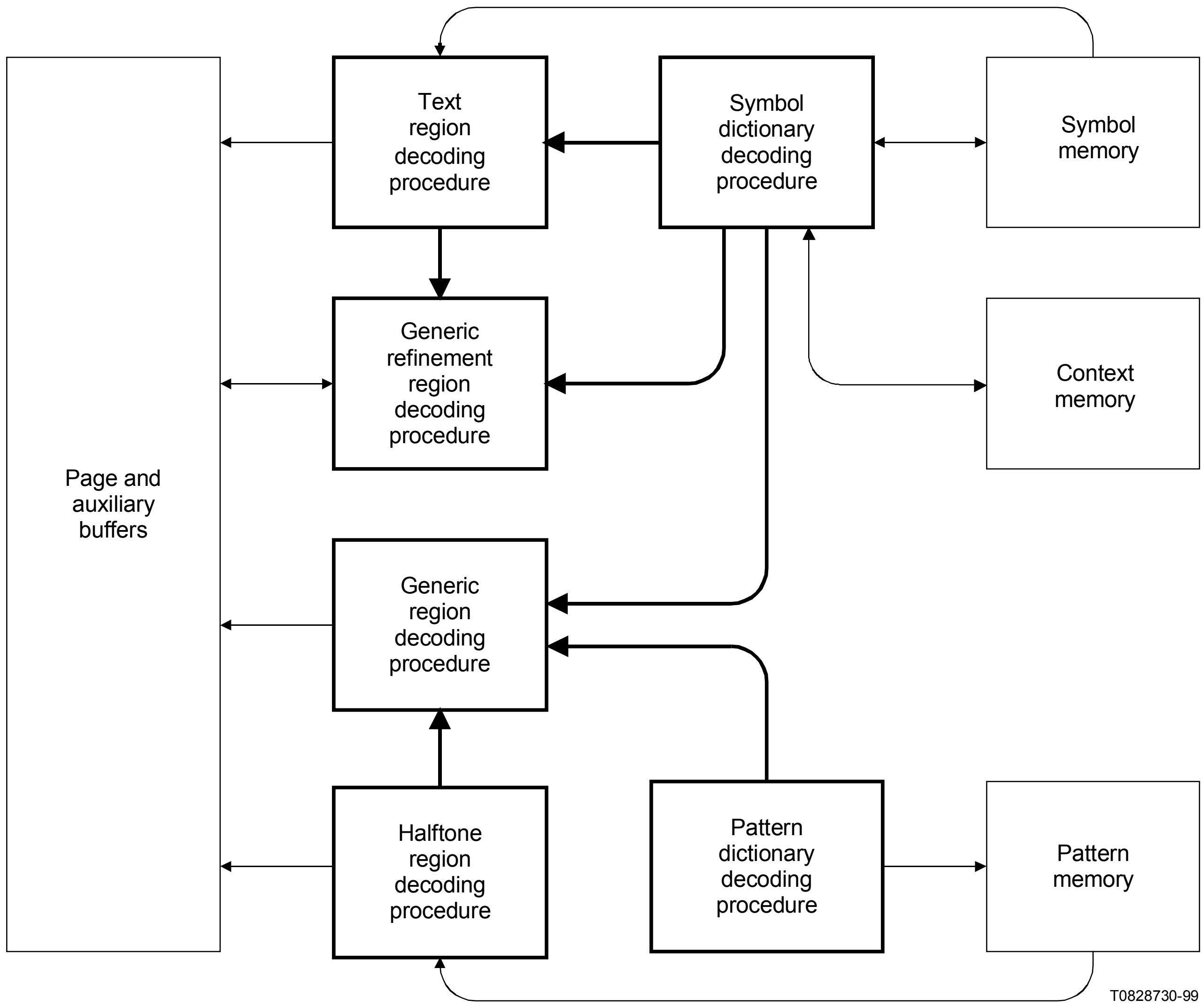



**Figure 1 – Block diagram of major decoder components**

The resources required to decode any given JBIG2 bitstream depend on the complexity of that bitstream. Some techniques such as striping can be used to reduce decoder memory requirements. It is estimated that a full-featured decoder may need two full-page buffers, plus about the same amount of dictionary memory, plus about 100 kilobytes of arithmetic coding context memory, to decode most bitstreams.

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.

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 this Recommendation | International 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 this Recommendation | International 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 colour to the foreground colour, but this Recommendation | International Standard specifies other permissible ways of changing a buffer's pixels.

A typical page will be described by 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 text region decoding procedure.

### 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 are 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 *text 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 text region segment. The text region segment contains the position and index information for each symbol to be 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 decoding procedure is invoked during the decoding of a symbol dictionary segment.

The *halftone region decoding procedure* takes patterns from a pattern 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 patterns to be placed in the buffer, as well as index information for the patterns themselves. The patterns, the fixed-size bitmaps of the halftone, are taken from the halftone dictionaries.

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

The *control decoding procedure* decodes segment headers, which include segment type information. The segment type determines which decoding procedure 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. The control decoding procedure affects everything shown in Figure 1, and so is not shown there as a separate block.

Table 1 summarises the types of data being decoded, which decoding procedure is responsible for decoding them, and what the final representations of the decoded data are.

**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 symbol dictionary segment | Symbol dictionary decoding procedure | Symbol bitmap |
| Gray-scale value | Field within a halftone dictionary segment | Pattern dictionary decoding procedure | Pattern |

## 0.2 Lossy coding

This Recommendation | International Standard 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

Lossy 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 [5], is to index the possibly wrong symbol and then to apply refinement coding to that symbol bitmap. The idea is to encode the basic character shape at little cost, then correct pixels that the encoder believes alter the meaning of the character.

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.

A number of examples of performing this sort of lossy symbol coding with JBIG2 can be found in [7].

NOTE – Although the term "text region" is used for regions of the page coded using symbol coding, other possible uses of symbol coding include coding line-art and other non-textual data.

### 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 difficulties are to ensure that the changes result in a lower code length and that the quality of the changed image does not suffer 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 *periodic* halftones, such as clustered-dot ordered dithered 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. Loss can also be introduced for additional compression by not putting all the patterns of the original image into the dictionary, thereby reducing both the number of halftone patterns and the number of bits required to specify which pattern is used in which location.

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. Error diffused images may also be coded losslessly, or with controlled loss as described above, using generic coding.

More details on performing this halftone coding can be found in [12].

### 0.2.4 Consequences of inadequate segmentation

In order to obtain optimum coding, both in terms of quality and file size, the correct form of encoding should be used for the appropriate regions of the document pages. This subclause briefly describes the consequences of errors in this segmentation.

Using lossy symbol coding for a document containing both text 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 [5], 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 handwritten 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.

**INTERNATIONAL STANDARD**

**ITU-T RECOMMENDATION**

# INFORMATION TECHNOLOGY – LOSSY/LOSSLESS CODING OF BI-LEVEL IMAGES

## 1 Scope

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

## 2 Normative References

The following Recommendations and International Standards contain provisions which, through references in this text, constitute provisions of this Recommendation | 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 Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of ITU maintains a list of the currently valid ITU-T Recommendations.

– CCITT Recommendation T.6 (1988), *Facsimile coding schemes and coding control functions for Group 4 facsimile apparatus.*

– ISO/IEC 8859-1:1998, *Information technology – 8-bit single-byte coded graphic character sets – Part 1: Latin alphabet No. 1.*

– ISO/IEC 10646-1:2000, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

## 3 Terms and Definitions

For the purposes of this Recommendation | International Standard, the following terms and definitions apply.

**3.1** **adaptive template pixel(s)**: Special pixel(s), in a template, whose location is not fixed.

**3.2** **aggregation**: Joining or merging of several individual symbols into a new symbol.

**3.3** **bi-level image**: Rectangular array of bits.

**3.4** **bit**: Binary digit, representing the value `0` or `1`.

**3.5** **bitmap**: Bi-level image.

**3.6** **buffer**: Storage area used to hold a bitmap.

**3.7** **byte**: Eight bits of data.

**3.8** **combination operator**: Operator used to combine the prior contents of a bitmap with new values being drawn into that bitmap.

**3.9** **coordinate system**: 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.

**3.10** **delta S**: Difference in the S coordinates between two successive symbol instances in a non-empty strip.

**3.11** **delta T**: Difference in the T coordinates between two successive non-empty strips.

**3.12** **decoding procedure**: Component of a decoder that decodes a certain type of data.

**3.12.1** **integer decoding procedure**: Decoding procedure whose output on each invocation is a single value.

**3.12.2** **arithmetic integer decoding procedure**: Integer decoding procedure that uses arithmetic entropy decoding.

**3.12.3** **region decoding procedure**: Decoding procedure whose output is a bitmap.

**3.12.4** **generic region decoding procedure**: Region decoding procedure that operates by decoding pixels individually or in runs.

**3.12.5** **generic refinement region decoding procedure**: Region decoding procedure that operates by modifying a reference bitmap to produce an output bitmap.

**3.12.6** **gray-scale decoding procedure**: Decoding procedure whose output is a gray-scale image.

**3.12.7** **pattern dictionary decoding procedure**: Decoding procedure whose output is a list of patterns.

**3.12.8** **halftone region decoding procedure**: Region decoding procedure that operates by drawing a set of patterns into a bitmap, placing the patterns according to a halftone grid.

**3.12.9** **Huffman table decoding procedure**: Decoding procedure whose output is a Huffman table.

**3.12.10** **text region decoding procedure**: Region decoding procedure that operates by drawing a set of symbol instances into a bitmap.

**3.12.11** **symbol dictionary decoding procedure**: Decoding procedure whose output is a list of symbols.

**3.13** **decoder**: Entity capable of decoding a bitstream in conformance with this Recommendation | International Standard.

**3.14** **dictionary**: List of bitmaps.

**3.14.1** **pattern dictionary**: List of patterns.

**3.14.2** **symbol dictionary**: List of symbols.

**3.15** **export flag**: Bit indicating that a symbol is on the export list of a symbol dictionary.

**3.16** **export list**: List of the symbols in a symbol dictionary that may be used by referring to that symbol dictionary.

**3.17** **gray-scale image**: Rectangular array of non-negative integer indices.

**3.18** **gray-scale pixel**: Integer-valued element in a gray-scale image.

**3.19** **halftone grid**: Rectilinear grid of locations specifying where patterns are to be drawn.

**3.20** **height class**: Set of symbols in a symbol dictionary whose heights are all equal.

**3.21** **height class delta height**: Difference in height between two height classes.

**3.22** **height class delta width**: Difference in width between two symbols in a height class.

**3.23** **Huffman table**: Collection of table lines specifying how values are encoded.

**3.24** **lossless coding**: Method of encoding data so that the decoded data are identical to the original data.

**3.25** **lossy coding**: Method of encoding data so that the decoded data differ, ideally only in insignificant ways, from the original data.

**3.26** **ordinal**: Value used as a counter.

**3.27** **out-of-band value**: Non-numeric value that may be produced in place of an integer.

**3.28** **pattern**: Bitmap produced by a pattern dictionary decoding procedure.

**3.29** **pixel**: Element with **0** or **1** as its value in a bitmap.

**3.30** **prefix length**: Length of the Huffman code prefix in a table line.

**3.31** **range length**: Number of additional code bits in a table line.

**3.32** **reference bitmap**: Bitmap used as the reference plane during the refinement region decoding procedure.

**3.33** **referred-to segment**: Other segment required in order to decode the current segment.

**3.34** **region**: Bitmap produced by a region decoding procedure.

**3.35** **segment**: Segment header and its segment data.

**3.36** **strip**: Full-width or full-height portion of the coordinate system of a text region.

**3.36.1** **empty strip**: Strip containing the reference corners of no symbol instances.

**3.36.2** **non-empty strip**: Strip containing the reference corner of at least one symbol instance.

**3.37** **strip size**: Extent in pixels of the non-full dimension of a strip.

**3.38** **symbol**: Bitmap produced by a symbol dictionary decoding procedure.

**3.39** **symbol ID**: Integer used to identify a symbol or to index into an array of symbols to retrieve the symbol.

**3.40** **symbol instance**: Symbol drawn, possibly with refinement, at a particular location in a text region.

**3.41** **symbol instance refinement delta height**: 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**: 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:** 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**: 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** **table line**: Specification of the encoding of a single value or a range of values as a Huffman code prefix followed by a fixed number of additional code bits.

**3.46** **typical prediction**: Method of signalling that an entire row of a generic region is identical to the preceding row.

**3.47** **value**: Integer or out-of-band indicator that is decoded.

# 4 Symbols and Abbreviations

NOTE – Due to ISO nomenclature requirements, within the context of clause 4, the term "symbol" is locally used to mean a variable name.

## 4.1 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply.

AT Adaptive Template

EOFB End-of-Facsimile Block

ID Identifier

LPS Less Probable Symbol, i.e. less probable binary value

LSB Least Significant Bit

MMR Modified Modified READ

MPS More Probable Symbol, i.e. more probable binary value

MSB Most Significant Bit

OOB Out-of-Band

READ Relative Element Address Designate

TPGD Typical prediction for generic direct bitmap coding

TPGR Typical prediction for generic refinement bitmap coding

NOTE – The term "symbol" in the abbreviations LPS and MPS does not refer to the symbols (bitmaps) in this Recommendation | 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 Recommendation | International Standard are listed below. A convention is used that parameters to any of the decoding procedures that are used in this Recommendation | International Standard are indicated in **bold face**.

| | |
|---|---|
| A | Probability interval |
| *a* | A real number |
| ARR | 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 |
| $B_{HC}$ | A height class collective bitmap in a symbol dictionary decoding procedure |
| $B_{HDC}$ | A dictionary collective bitmap in a pattern dictionary decoding procedure |
| $B_P$ | A pattern bitmap in a pattern dictionary decoding procedure |
| $B_S$ | A symbol bitmap in a symbol dictionary decoding procedure |
| *BM* | A bitmap |
| BP | Pointer to byte B |
| BPST | Initial value of BP |
| C | Value of bitstream in code register |
| Chigh | High-order 16 bits of C |
| Clow | Low-order 16 bits of C |
| CONTEXT | The values of the pixels in a template used in the generic or generic refinement decoding procedure |
| CT | Renormalisation shift counter |
| CURCODE | The Huffman code for the current table line in a Huffman table |
| CUREXFLAG | The current export flag |
| CURLEN | The current table line prefix length in a Huffman table |
| CURRANGELOW | The lower bound of the range of the current table line in a Huffman table |
| CURS | The current S coordinate in a text region decoding procedure |
| CURT | The current symbol instance's T coordinate relative to the current strip's T coordinate in a text region decoding procedure |
| CX | A label identifying an arithmetic coding context |
| D | Arithmetic coding decision |
| 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 in width between two symbol bitmaps in a symbol dictionary decoding procedure |
| EXFLAGS | An array of export flags |
| EXINDEX | An index for the array EXFLAGS |
| 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 |
| $\mathbf{GBATX_1}$ | The X location of adaptive template pixel 1 in a generic region decoding procedure |
| $\mathbf{GBATY_1}$ | The Y location of adaptive template pixel 1 in a generic region decoding procedure |
| $\mathbf{GBATX_2}$ | The X location of adaptive template pixel 2 in a generic region decoding procedure |
| $\mathbf{GBATY_2}$ | 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 |
| GB | The prefix used for many of the variables associated with a generic (bitmap) 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 |
| GI | An array of gray-scale values |
| GR | The prefix used for many of the variables associated with a generic refinement region decoding procedure |
| **$GRATX_1$** | 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 |
| GRAY | The current gray-scale value |
| **GRAYMAX** | The largest gray-scale value for which a pattern is given in a pattern dictionary 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 |
| GS | The prefix used for many of the variables associated with a gray-scale image decoding procedure |
| **GSBPP** | The number of bits per gray-scale value in a gray-scale image decoding procedure |
| **GSH** | The height of the gray-scale image in a gray-scale image decoding procedure |
| **GSKIP** | A mask indicating gray-scale values to be skipped |
| **GSMMR** | Whether MMR is used in a gray-scale image decoding procedure |
| **GSTEMPLATE** | A parameter indicating the number and arrangement of the pixels in a template used in a gray-scale image decoding procedure |
| **GSUSESKIP** | Whether some gray-scale values should be skipped in a gray-scale image decoding procedure |
| GSVALS | A decoded gray-scale image |
| **GSW** | The width of the gray-scale image in a gray-scale image decoding procedure |
| HB | The prefix used for many of the variables associated with a halftone (bitmap) region decoding procedure |
| **HBH** | The height of a halftone region |
| HBPP | The number of bits per value in an array of gray-scale values |
| **HBW** | The width of a halftone region |
| 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 |
| **HCOMBOP** | The combination operator used in a halftone region decoding procedure |
| HD | The prefix used for many of the variables associated with a pattern dictionary region decoding procedure |
| **HDEFPIXEL** | The default for pixels in a halftone region |
| **HDMMR** | Whether MMR is used in a pattern dictionary decoding procedure |
| **HDPATS** | Array of patterns produced by a pattern dictionary decoding procedure |
| **HDPH** | The height of the patterns in a pattern dictionary |
| **HDPW** | The width of the patterns in a pattern dictionary |
| **HDTEMPLATE** | The template identifier used to decode patterns in a pattern dictionary decoding procedure |
| **HENABLESKIP** | Whether unneeded gray-scale values are skipped in a halftone region decoding procedure |
| **HGH** | The height of the gray-scale image in a halftone region decoding procedure |
| **HGW** | The width of the gray-scale image in a halftone region decoding procedure |
| **HGX** | The horizontal offset of the grid in a halftone region decoding procedure |
| **HGY** | The vertical offset of the grid in a halftone region decoding procedure |
| $H_I$ | The height of a symbol instance bitmap |
| HIGHPREFLEN | The prefix length of the upper range table line in a Huffman table |
| **HMMR** | Whether MMR coding is used in a halftone region decoding procedure |
| **HNUMPATS** | The number of patterns that may be used in a halftone region decoding procedure |
| $HO_I$ | The height of the original bitmap of a symbol instance containing refinement information |
| **HPATS** | Array of patterns used in a halftone region |
| **HPH** | The height of each pattern in a halftone region |
| **HPW** | The width of each pattern in a halftone region |
| **HRX** | The horizontal coordinate of a halftone grid vector |
| **HRY** | The vertical coordinate of a halftone grid vector |
| HSKIP | A mask indicating gray-scale values to be skipped |
| HT | The prefix used for many of the variables associated with a Huffman table decoding procedure |
| **HTEMPLATE** | A parameter indicating the number and arrangement of the pixels in a template used in a halftone region decoding procedure |
| 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 |
| HTREG | The region produced by a halftone region decoding procedure |
| HTRS | The length of the encoded range field in a table line in a Huffman table |
| HTVAL | The value decoded using a Huffman table rappeller |
| I | The array, indexed by CX, of the indices of the adaptive probability estimates |
| $I$ | 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 values of symbol instance refinements |
| IARDY | An arithmetic integer decoding procedure used to decode the delta Y values 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 text 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 |
| LOGSBSTRIPS | The base-2 logarithm of the strip size used to encode a text region |
| LOWPREFLEN | The prefix length of the lower range table line in a Huffman table |
| LTP | Whether the current line is coded explicitly in a generic region decoding procedure or a generic refinement region decoding procedure |
| $m_g$ | Horizontal index for the current gray-scale value |
| **MMR** | Whether MMR coding is used in a generic region decoding procedure |
| MPS | The array, indexed by CX, of the current more probable binary values |
| NINSTANCES | A symbol instance counter |
| $n_g$ | Vertical index for the current gray-scale value |
| NLPS | The next index for an LPS renormalisation |
| NMPS | The next index for an MPS renormalisation |
| 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 |
| Qe | An estimate of the LPS probability |
| r | A segment retention flag |
| R | The number of segments referred to by some segment |
| RANGELEN | An array of the lengths of the ranges of the table lines in a Huffman table |

| | |
|---|---|
| RANGELOW | An array of the lower bounds of the ranges 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 text region decoding procedure |
| S | One coordinate of the coordinate system used in a text region decoding procedure |
| $S_I$ | The S coordinate of a symbol instance |
| SB | The prefix used for many of the variables associated with a symbol (bitmap) region decoding procedure |
| **SBDSOFFSET** | An offset for the coded delta S values in a text region |
| **SBCOMBOP** | The combination operator used in a text region decoding procedure |
| **SBDEFPIXEL** | The default for pixels in a text region |
| **SBH** | The height of a text region |
| **SBHUFF** | Whether Huffman coding is used in a text 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 text region |
| **SBNUMSYMS** | The number of symbols that may be used in a text region |
| $\mathbf{SBRATX_1}$ | The X location of the adaptive template pixel $RA_1$ in a text region decoding procedure |
| $\mathbf{SBRATY_1}$ | The Y location of the adaptive template pixel $RA_1$ in a text region decoding procedure |
| $\mathbf{SBRATX_2}$ | The X location of the adaptive template pixel $RA_2$ in a text region decoding procedure |
| $\mathbf{SBRATY_2}$ | The Y location of the adaptive template pixel $RA_2$ in a text region decoding procedure |
| **SBREFINE** | Whether refinement coding is used in a text region decoding procedure |
| SBREG | The region produced by a text region decoding procedure |
| **SBRTEMPLATE** | Template identifier for refinement coding of bitmap in a text region decoding procedure |
| **SBSTRIPS** | The height of the symbol instance strips |
| **SBSYMCODELEN** | The length of the symbol codes used in IAID |
| **SBSYMCODES** | An array of variable-length codes identifying individual symbols |
| **SBSYMS** | An array of symbols used in a text region |
| **SBW** | The width of a text region |

| | |
|---|---|
| SD | The prefix used for many of the variables associated with a symbol dictionary region decoding procedure |
| **SDATX$_1$** | The X location of the adaptive template pixel $A_1$ in a symbol dictionary decoding procedure |
| **SDATY$_1$** | The Y location of the adaptive template pixel $A_1$ in a symbol dictionary decoding procedure |
| **SDATX$_2$** | The X location of the adaptive template pixel $A_2$ in a symbol dictionary decoding procedure |
| **SDATY$_2$** | The Y location of the adaptive template pixel $A_2$ in a symbol dictionary decoding procedure |
| **SDATX$_3$** | The X location of the adaptive template pixel $A_3$ in a symbol dictionary decoding procedure |
| **SDATY$_3$** | The Y location of the adaptive template pixel $A_3$ in a symbol dictionary decoding procedure |
| **SDATX$_4$** | The X location of the adaptive template pixel $A_4$ in a symbol dictionary decoding procedure |
| **SDATY$_4$** | The Y location 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 |
| **SDRATX$_1$** | The X location of the adaptive template pixel $RA_1$ in a symbol dictionary decoding procedure |
| **SDRATY$_1$** | The Y location of the adaptive template pixel $RA_1$ in a symbol dictionary decoding procedure |
| **SDRATX$_2$** | The X location of the adaptive template pixel $RA_2$ in a symbol dictionary decoding procedure |
| **SDRATY$_2$** | The Y location 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 |
| SLTP | A binary value indicating whether the current line was typically predicted and the previous line was not, or vice versa |
| STRIPT | The numerically smallest T coordinate in the current strip |
| SWITCH | Whether MPS and LPS are switched on an LPS renormalisation |
| SYMWIDTH | The current bitmap width in a symbol dictionary decoding procedure |
| T | One coordinate of the coordinate system used in a text region decoding procedure |
| TEMPC | A temporary register in the MQ coder |
| $T_I$ | TheT coordinate of a symbol instance |
| TOTWIDTH | The total width of the bitmaps in a height class |
| **TPGDON** | Whether typical prediction is used in a generic region decoding procedure |
| **TPGRON** | Whether typical prediction is used in a generic region refinement decoding procedure |

| | |
|---|---|
| TPGRPIX | Whether the current pixel is to be decoded implicitly using a TPGR prediction |
| TPGRVAL | The value of the TPGR-predicted current pixel |
| **TRANSPOSED** | Whether the symbol instance coordinates are transposed in a text 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 |
| $x$ | The horizontal coordinate of a location on a halftone grid |
| X | The horizontal coordinate of a pixel in a bitmap |
| $y$ | The vertical coordinate of a location on a halftone grid |
| 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 a is a number, then $\lfloor a \rfloor$ is the largest integer less than or equal to $a$. |
| $\lceil\ \rceil$ | If a is a number, then $\lceil a \rceil$ is the smallest integer greater than or equal to $a$. |
| << | 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 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:

ARR[0], ARR[1], . . . , ARR[11]

## 5.6 Image and bitmap conventions

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

NOTE 1 – Throughout this Recommendation | International Standard, pixels in bitmaps are treated as having the values **0** or **1**. In most applications of this Recommendation | International Standard, the application will select some interpretation of these two values. A typical interpretation of these pixels is that **0** represents white, or background, and **1** represents black, or foreground. However, this is not a requirement of this Recommendation | International Standard and applications are free to make other interpretations of these values.

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 Recommendation | International 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*[X, Y].

NOTE 2 – 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.

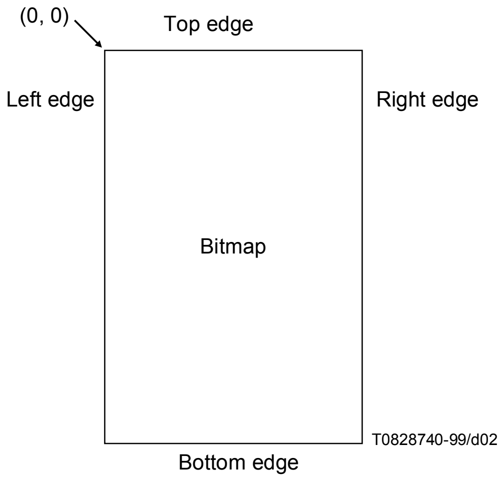



**Figure 2 – The four edges of a bitmap**

# 6 Decoding Procedures

## 6.1 Introduction to decoding procedures

This Recommendation | 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 text region decoding procedures all produce regions as their output. The symbol dictionary decoding procedure produces an array of symbols as its output. The pattern dictionary decoding procedure similarly produces an array of halftone cell bitmaps as its output.

The various region decoding procedures operate in different ways:

- The generic region decoding procedure decodes a bitmap, treating it simply as 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 text 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 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.

In this clause, subsequent clauses, and normative annexes, restrictions are placed on the bitstream being decoded.

EXAMPLE 1 – In 7.3, some segment types are described as "Reserved; must not be used."

EXAMPLE 2 – In 7.4.2.1.1, if the **SDHUFF** field is **0** then the **SDHUFFDH** field must contain the value **0**.

These restrictions should be interpreted as meaning that the behaviour of a decoder encountering a bitstream that does not satisfy the restrictions is undefined, and is outside the scope of this Recommendation | International Standard.

NOTE – This means that if a decoder encounters a bitstream that does not satisfy the restrictions, it may take any action: it may give up and abort decoding; it may ignore the error and attempt to continue; it may interpret the error and change its behaviour (e.g. use the error to attempt to aid recovery from further errors); and so on.

## 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 Rec. T.82 | ISO/IEC 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.

**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.[a)] |
| **TPGDON** | Integer | 1 | N | Whether typical prediction is used.[a)] |
| **USESKIP** | Integer | 1 | N | Whether some pixels should be skipped in the decoding.[a)] |
| **SKIP** | Bitmap | | | A bitmap indicating which pixels should be skipped. **GBW** pixels wide, **GBH** pixels high.[c)] |
| **$GBATX_1$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_1$.[a)] |
| **$GBATY_1$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_1$.[a)] |
| **$GBATX_2$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_2$.[b)] |
| **$GBATY_2$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_2$.[b)] |
| **$GBATX_3$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_3$.[b)] |
| **$GBATY_3$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_3$.[b)] |
| **$GBATX_4$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_4$.[b)] |
| **$GBATY_4$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_4$.[b)] |

a) Unused if **MMR = 1**.
b) Unused if **MMR = 1** or **GBTEMPLATE** ≠ 0.
c) Unused if **USESKIP = 0** or **MMR = 1**.

### 6.2.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 3.

**Table 3 – Return value from the generic region decoding procedure**

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

### 6.2.4 Variables used in decoding

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

**Table 4 – Variables used in the generic region decoding procedure**

| Name | Type | Size (bits) | Signed? | Description and restrictions |
|---|---|---|---|---|
| LTP | Integer | 1 | N | Whether the current image line is coded explicitly.[a)] |
| SLTP | Integer | 1 | N | Whether the current line's LTP value differs from the previous line's LTP value.[a)] |
| CONTEXT | Integer | 16 | N | The values of the pixels in the template.[a)] |
| [a)] Unused if **MMR** = **1**. | | | | |

### 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**.

#### 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 some pixels in fixed spatial relationship to the target pixel.

Near the edges of the bitmap, these neighbour references might not lie in the actual bitmap. The rule to satisfy out-of-bounds references shall be:

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

#### 6.2.5.3 Fixed templates

A template defines a neighbourhood around a pixel to be coded. The values of the pixels in this neighbourhood 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.

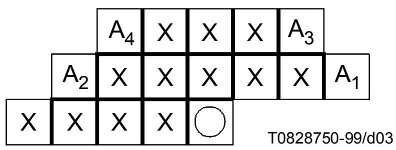



**Figure 3 – Template when GBTEMPLATE = 0, showing the AT pixels at their nominal locations**

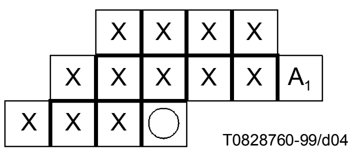



**Figure 4 – Template when GBTEMPLATE = 1, showing the AT pixel at its nominal location**

Figure 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. They are denoted "adaptive" or AT pixels. These pixels are special in that their locations 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 locations are specified as parameters to this decoding procedure; Figures 3 to 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 shall be used by the arithmetic coder for encoding the pixel to be coded (see Annex E).

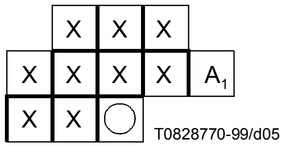



**Figure 5 – Template when GBTEMPLATE = 2, showing the AT pixel at its nominal location**

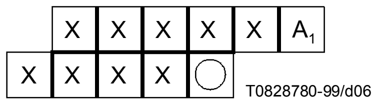



**Figure 6 – Template when GBTEMPLATE = 3, showing the AT pixel at its nominal location**

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 Rec. T.82 | ISO/IEC 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 are called AT pixels. Their nominal locations 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 of using an effective template size of 15, 14, 13, 12 or 9 pixels by having the moved location 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 indices of the AT pixels in Figure 3 correspond to the expected goodness. If moving only one AT pixel from its nominal location, it is advisable to move $A_4$. The next pixel to move is $A_3$ and so on.

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. However, when TPGD is enabled (**TPGDON = 1**), the context used to code the SLTP value is used, regardless of whether AT pixels overlap regular template pixels. This means that contexts where the AT pixel's value differs from the regular template pixel's value can still occur, but only for SLTP when TPGD is enabled.

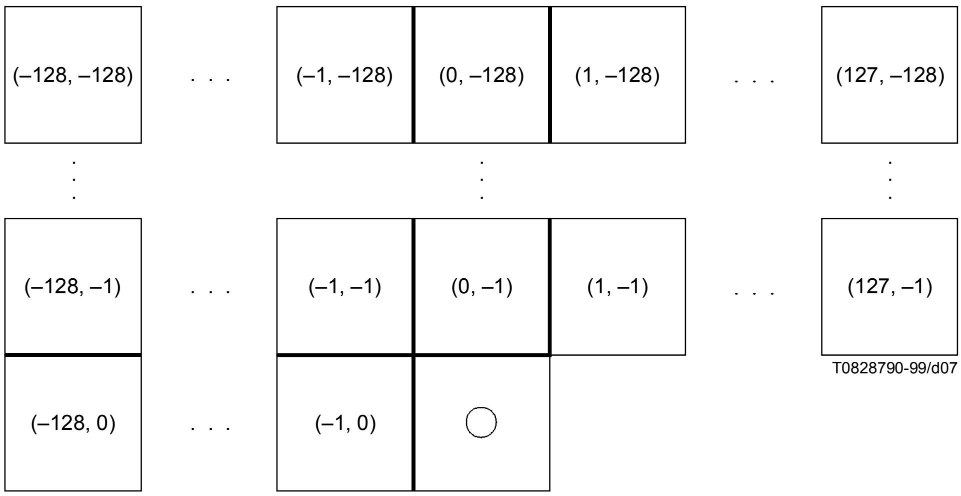



**Figure 7 – Field to which AT pixel locations are restricted**

**Table 5 – The nominal values of the AT pixel locations**

| **GBTEMPLATE** | **GBATX$_1$**<br>**GBATY$_1$** | **GBATX$_2$**<br>**GBATY$_2$** | **GBATX$_3$**<br>**GBATY$_3$** | **GBATX$_4$**<br>**GBATY$_4$** |
|---|---|---|---|---|
| 0 | 3<br>–1 | –3<br>–1 | 2<br>–2 | –2<br>–2 |
| 1 | 3<br>–1 | NA<br>NA | NA<br>NA | NA<br>NA |
| 2 | 2<br>–1 | NA<br>NA | NA<br>NA | NA<br>NA |
| 3 | 2<br>–1 | NA<br>NA | NA<br>NA | NA<br>NA |
| NOTE – NA means that the parameter has no nominal value. | | | | |

### 6.2.5.5 Typical prediction for generic direct coding (TPGD)

Typical prediction for generic direct coding can be enabled or disabled with the **TPGDON** parameter. If typical prediction for generic direct coding is enabled (**TPGDON** 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.

NOTE – The value decoded before the first pixel of each row is not used in any pixel's template.

### 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 is not actually decoded.

#### 6.2.5.7 Decoding the bitmap

The decoding of the bitmap proceeds as follows:

1) Set:

$$\text{LTP} = \mathbf{0}$$

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 **TPGDON** 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 SLTP be the value of this bit. Set:

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

      NOTE – In Figures 8 through 11, the template is shown with the AT pixel or pixels in their nominal locations. The same pixel values (**0** or **1**) shall be used for the AT pixels no matter what their actual locations are. That is, moving the AT pixels does not affect the context that is used to decode SLTP.

   c) If LTP = **1** then set every pixel of the current row of GBREG equal to the corresponding pixel of the row immediately above.

   d) If LTP = **0** 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:

         - Place the template given by parameters **GBTEMPLATE**, **GBATX$_1$** through **GBATX$_4$** and **GBATY$_1$** through **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**.
         - 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 shall be consistent and independent of the location of the AT pixels.
         - 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.

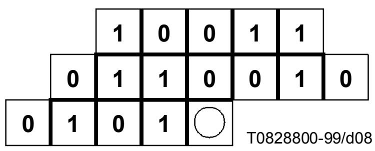



**Figure 8 – Reused context for coding the SLTP value when GBTEMPLATE is 0**

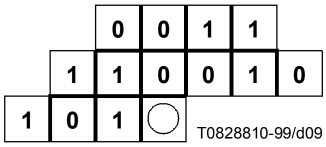



**Figure 9 – Reused context for coding the SLTP value when GBTEMPLATE is 1**

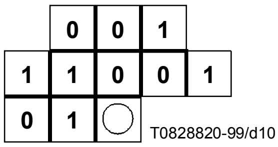



**Figure 10 – Reused context for coding the SLTP value when GBTEMPLATE is 2**

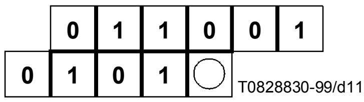



**Figure 11 – Reused context for coding the SLTP value when GBTEMPLATE is 3**

### 6.2.6 Decoding using MMR coding

If **MMR** is **1**, the generic region decoding procedure is identical to an MMR (Modified Modified READ) decoder described in ITU-T Recommendation T.6, with the following exceptions:

- An invocation of the generic region decoding procedure with **MMR** equal to **1** shall consume an integral number of bytes, beginning and ending on a byte boundary. This may involve skipping over some bits in the last byte read.
- 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 Recommendation | 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**.
- If the number of bytes contained in the encoded bitmap is known in advance, then it is permissible for the data stream not to contain an EOFB (**000000000001000000000001**) at the end of the MMR-encoded data. The cases where the number of bytes is known are when this decoding procedure is invoked:
  – from within the pattern dictionary decoding procedure;
  – from within the symbol dictionary decoding procedure; or
  – as part of decoding a generic region whose data length is known.

  The number of bytes is not known when this decoding procedure is invoked from within the gray-scale image decoding procedure, or when it is invoked as part of a decoding generic region whose data length is not known. In these cases, EOFB must be present.

NOTE 1 – The sources of the byte count, in the cases where it is known, are:

– Within the pattern dictionary decoding procedure, the byte count is known because all the segment data, beyond the fixed-length data header, is a single MMR-encoded data block, so the MMR data length can be computed from the segment data length.

– Within the symbol dictionary decoding procedure, the byte count is known from BMSIZE.

– Within the generic region decoding procedure, the byte count is again known from the segment data length.

The reason for allowing EOFB to be optional is that an EOFB is three bytes long, while the byte count is often known beforehand, or can be encoded in fewer than three bytes. Thus, omitting an EOFB reduces the bitmap's coded data size; in symbol dictionaries, where there are often many small bitmaps encoded separately, this savings can be significant.

NOTE 2 – A decoder can take a number of approaches to deal with EOFB in the cases where it is optional. These approaches take advantage of the known byte count and the fact that the EOFB, if present, is counted in this byte count. Two possible approaches are:

– Invoke the MMR decoding procedure, and always check for EOFB after the bitmap has been decoded. However, allow the MMR decoding procedure to examine no more bytes than are known to be part of the MMR-compressed data block. If the MMR decoding procedure runs out of data while checking for EOFB, this is not an error, but a normal condition indicating that EOFB was not present.

– Invoke the MMR decoding procedure, and never check for EOFB after the bitmap has been decoded, in the cases where EOFB is optional. If the MMR decoding procedure consumes fewer bytes than are known to part of the MMR-compressed data block, this is not an error, but a normal condition indicating that EOFB was present. Skip over such unconsumed bytes.

- The extension codes of T.6, including uncompressed mode, must not be present in the MMR-encoded data.

NOTE 3 – 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.

**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. |
| **TPGRON** | Integer | 1 | N | Whether typical prediction for generic refinement is used. |
| **$GRATX_1$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_1$.[a)] |
| **$GRATY_1$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_1$.[a)] |
| **$GRATX_2$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_2$.[a)] |
| **$GRATY_2$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_2$.[a)] |
| a) Unused if **GRTEMPLATE ≠ 0**. | | | | |

### 6.3.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 7.

**Table 7 – Return value from the generic refinement region decoding procedure**

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

### 6.3.4 Variables used in decoding

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

**Table 8 – Variables used in the generic refinement region decoding procedure**

| Name | Type | Size (bits) | Signed? | Description and restrictions |
|---|---|---|---|---|
| CONTEXT | Integer | 13 | N | The values of the pixels in the template. |
| LTP | Integer | 1 | N | Whether the current image line is decoded explicitly.[a)] |
| SLTP | Integer | 1 | N | Whether the current line's LTP value is different from the previous line's LTP value.[a)] |
| TPGRPIX | Integer | 1 | N | Whether the current pixel is to be decoded implicitly using a TPGR prediction.[a)] |
| TPGRVAL | Integer | 1 | N | Value of the TPGR-predicted current pixel.[a)] |
| [a)] Unused if **TPGRON** = **0**. | | | | |

### 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 refined 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 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, these neighbour references might not lie in the actual bitmap. The rule to satisfy out-of-bounds references shall be:

- 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 neighbourhood around a pixel to be coded. The values of the pixels in this neighbourhood 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 – 13-pixel refinement template showing the AT pixels at their nominal locations**

Figure 12 shows the template which shall be used when **GRTEMPLATE** is **0**. Figure 13 shows the template which shall be used when **GRTEMPLATE** 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 locations 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) in the reference bitmap.

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 – **GRREFERENCEDX**, Y – **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 shall be used by the arithmetic coder for encoding the pixel to be coded (see Annex E).


**Figure 13 – 10-pixel refinement template**

#### 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 nominal locations 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 generic refinement (TPGR)

Typical prediction for generic refinement can be enabled or disabled with the **TPGRON** parameter. If typical prediction for generic refinement is enabled (**TPGRON** is **1**) then before the first pixel of each row is decoded, a value indicating whether a row is typical shall be decoded. If the row is not typical, each pixel of the row needs to be explicitly decoded. If the row is typical, 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 the criteria defined in 6.3.5.6, step 3d).

NOTE – The value decoded before the first pixel of each row is not used in any pixel's template.

#### 6.3.5.6 Decoding the refinement bitmap

The decoding of the bitmap proceeds 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 **TPGRON** 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 **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 decoded bit. Set:

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

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 **GRATX$_1$**, **GRATY$_1$**, **GRATX$_2$** and **GRATY$_2$** if **GRTEMPLATE** is **0**) 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 **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 explicitly decode the rest. The procedure for each pixel is as follows:

i) Set TPGRPIX equal to **1** if:

- **TPGRON** is **1** AND;
- a 3 × 3 pixel array in the reference bitmap (Figure 16), centred at the location corresponding to the current pixel, contains pixels all of the same value.

When TPGRPIX is set to **1**, set TPGRVAL equal to the current pixel predicted value, which is the common value of the nine adjacent pixels in the 3 × 3 array.

ii) If TPGRPIX is **1** then implicitly decode the current pixel by setting it equal to its predicted value (TPGRVAL).

iii) Otherwise, explicitly 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.


**Figure 14 – Reused context for coding the SLTP value when GRTEMPLATE is 0**


**Figure 15 – Reused context for coding the SLTP value when GRTEMPLATE is 1**


**Figure 16 – TPGR template**

## 6.4 Text 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.

### 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

**Table 9 – Parameters for the text 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.[a)] |
| **SBSYMCODELEN** | Integer | 6 | N | The length of the symbol codes used in IAID.[d)] |
| **SBSYMS** | Array of symbols | | | An array containing the symbols used in this text region. Contains **SBNUMSYMS** symbols. |
| **SBDEFPIXEL** | Integer | 1 | N | The default pixel for this bitmap. |
| **SBCOMBOP** | Operator | | | The combination operator for this text region. May take on the values OR, AND, XOR and XNOR. |
| **TRANSPOSED** | Integer | 1 | N | Whether the strips run vertically. |

**Table 9** *(concluded)*

| Name | Type | Size (bits) | Signed? | Description and restrictions |
|---|---|---|---|---|
| **REFCORNER** | Corner | | | The reference corner of each symbol instance bitmap. 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.[a] |
| **SBHUFFFDS** | Huffman table | | | The Huffman table used to decode the S coordinate of subsequent symbol instances in each strip. [a] |
| **SBHUFFDT** | Huffman table | | | The Huffman table used to decode the difference in T coordinates between non-empty strips.[a] |
| **SBHUFFRDW** | Huffman table | | | The Huffman table used to decode the difference between a symbol's width and the width of a refinement coded bitmap.[b] |
| **SBHUFFRDH** | Huffman table | | | The Huffman table used to decode the difference between a symbol's height and the height of a refinement coded bitmap.[b] |
| **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.[b] |
| **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.[b] |
| **SBHUFFRSIZE** | Huffman table | | | The Huffman table used to decode the size of a symbol instance's refinement bitmap data.[b] |
| **SBRTEMPLATE** | Integer | 1 | N | Template identifier for refinement coding of symbol instance bitmaps.[c] |
| **SBRATX$_1$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_1$.[c] |
| **SBRATY$_1$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_1$.[c] |
| **SBRATX$_2$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_2$.[c] |
| **SBRATY$_2$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_2$.[c] |

a) Unused if **SBHUFF** = **0**.
b) Unused if **SBHUFF** = **0** or **SBREFINE** = **0**.
c) Unused if **SBREFINE** = **0**.
d) Unused if **SBHUFF** = **1**.

### 6.4.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 10.

**Table 10 – Return value from the text region decoding procedure**

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

### 6.4.4 Variables used in decoding

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

**Table 11 – Variables used in the text 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 symbol 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.[a)] |
| $RDH_I$ | Integer | 32 | Y | The delta height of a symbol instance's refinement bitmap.[a)] |
| $RDX_I$ | Integer | 32 | Y | The delta X of a symbol instance's refinement bitmap.[a)] |
| $RDY_I$ | Integer | 32 | Y | The delta Y of a symbol instance's refinement bitmap.[a)] |
| $IBO_I$ | Bitmap | | | A symbol instance's original symbol bitmap.[a)] |
| $WO_I$ | Integer | 32 | N | The width of $IBO_I$.[a)] |
| $HO_I$ | Integer | 32 | N | The height of $IBO_I$.[a)] |
| a) Unused if **SBREFINE** = **0**. | | | | |

### 6.4.5 Decoding the text region

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 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 × **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 text 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 text region**

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

**Figure 18 – Structure of a strip**

| |
|---|
| 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**

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

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 text 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 S coordinate as described in 6.4.7. Let DFS be the decoded value. Set:

$$\mathrm{FIRSTS} = \mathrm{FIRSTS} + \mathrm{DFS}$$
$$\mathrm{CURS} = \mathrm{FIRSTS}$$

ii) Otherwise, if the current symbol instance is not the first symbol instance in the strip, decode the symbol instance's 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 3 d). Otherwise, let IDS be the decoded value. Set:

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

NOTE 4 – The intended use of **SBDSOFFSET** is to make the most 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 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 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:

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

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

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

- Otherwise, do not change CURS in this step.

vii) Set:

$$S_I = \mathrm{CURS}$$

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 5 – 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 Annex A) and multiply the resulting value by **SBSTRIPS**.

### 6.4.7 First symbol instance S coordinate

NOTE – The symbol instance S coordinate value for 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 Annex 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 Annex 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 **SBSTRIPS** = 1, then the value decoded is always zero. Otherwise:

- If **SBHUFF** is **1**, decode a value by reading $\lceil \log_2 \mathbf{SBSTRIPS} \rceil$ bits directly from the bitstream.
- If **SBHUFF** is **0**, decode a value using the IAIT integer arithmetic decoding procedure (see Annex A).

NOTE – If **SBSTRIPS** = 1, then no bits are consumed, and the IAIT integer arithmetic decoding procedure is never invoked.

### 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 Annex 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 symbol instance refinement delta width as described in 6.4.11.1. Let $RDW_I$ be the value decoded.
2) Decode the symbol instance refinement delta height as described in 6.4.11.2. Let $RDH_I$ be the value decoded.
3) Decode the symbol instance refinement X offset as described in 6.4.11.3. Let $RDX_I$ be the value decoded.
4) Decode the symbol 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 symbol 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.
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 5 a).

#### 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 might 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 Annex A).

**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** | $\lfloor RDH_I/2 \rfloor + RDY_I$ |
| **TPGRON** | **0** |
| $\mathbf{GRATX_1}$ | $\mathbf{SBRATX_1}$ |
| $\mathbf{GRATY_1}$ | $\mathbf{SBRATY_1}$ |
| $\mathbf{GRATX_2}$ | $\mathbf{SBRATX_2}$ |
| $\mathbf{GRATY_2}$ | $\mathbf{SBRATY_2}$ |

#### 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 Annex 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 Annex 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 Annex 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

### 6.5.1 General description

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

### 6.5.2 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.

### 6.5.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 14.

### 6.5.4 Variables used in decoding

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

**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.[a)] |
| **SDHUFFDW** | Huffman table | | | The Huffman table used to decode the difference in width between two symbols.[a)] |
| **SDHUFFBMSIZE** | Huffman table | | | The Huffman table used to decode the size of a height class collective bitmap.[a)] |
| **SDHUFFAGGINST** | Huffman table | | | The Huffman table used to decode the number of symbol instances in an aggregation.[b)] |
| **SDTEMPLATE** | Integer | 2 | N | The template identifier used to decode symbol bitmaps.[c)] |
| **$SDATX_1$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_1$.[c)] |
| **$SDATY_1$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_1$.[c)] |
| **$SDATX_2$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_2$.[c)] |
| **$SDATY_2$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_2$.[c)] |
| **$SDATX_3$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_3$.[c)] |
| **$SDATY_3$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_3$.[c)] |
| **$SDATX_4$** | Integer | 8 | Y | The X location of the adaptive template pixel $A_4$.[c)] |
| **$SDATY_4$** | Integer | 8 | Y | The Y location of the adaptive template pixel $A_4$.[c)] |
| **SDRTEMPLATE** | Integer | 1 | N | Template identifier for refinement coding of bitmaps.[d)] |
| **$SDRATX_1$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_1$.[d)] |
| **$SDRATY_1$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_1$.[d)] |
| **$SDRATX_2$** | Integer | 8 | Y | The X location of the adaptive template pixel $RA_2$.[d)] |
| **$SDRATY_2$** | Integer | 8 | Y | The Y location of the adaptive template pixel $RA_2$.[d)] |

a) Unused if **SDHUFF** = **0**.
b) Unused if **SDHUFF** = **0** or **SDREFAGG** = **0**.
c) Unused if **SDHUFF** = **1**.
d) Unused if **SDREFAGG** = **0**.

**Table 14 – Return value 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.<br>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. |

### 6.5.5 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 1 – 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**

| |
|---|
| 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**

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.

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

3) Set:

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

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 6.5.6. Let HCDH be the decoded value. Set:

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

   c) Decode each symbol within the height class as follows:

      i) Decode the delta width for the symbol as described in 6.5.7. If the result of this decoding is OOB then all the symbols in this height class have been decoded; proceed to step 4 d). Otherwise let DW be the decoded value and set:

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

      ii) If **SDHUFF** is **0** or **SDREFAGG** is **1**, then decode the symbol's bitmap as described in 6.5.8. Let $B_S$ be the decoded bitmap (this bitmap has width SYMWIDTH and height HCHEIGHT). Set:

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

iii) If **SDHUFF** is **1** and **SDREFAGG** is **0**, then set:

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

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 6.5.9. Let $B_{HC}$ be the decoded bitmap. This bitmap has width TOTWIDTH and height HCHEIGHT. Break up the bitmap $B_{HC}$ as follows to obtain the symbols SDNEWSYMS[HCFIRSTSYM] through SDNEWSYMS[NSYMSDECODED – 1].

$B_{HC}$ 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 – Columns 0 through SDNEWSYMWIDTHS[HCFIRSTSYM] – 1 of $B_{HC}$ contain the bitmap for SDNEWSYMS[HCFIRSTSYM], the first symbol in the height class.

5) Determine which symbol bitmaps are exported from this symbol dictionary, as described in 6.5.10. 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 2 – Not all the new symbols need to be exported; this allows the dictionary to define 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.

### 6.5.6 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 Annex A).

### 6.5.7 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 Annex A).

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

### 6.5.8 Symbol bitmap

This field is only present if **SDHUFF** = **0** or **SDREFAGG** = **1**. This field takes one of two forms; **SDREFAGG** determines which form is used.

#### 6.5.8.1 Direct-coded symbol 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 decoding procedure as shown in Table 16.

**Table 16 – Parameters used to decode a symbol's bitmap using generic bitmap decoding**

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

### 6.5.8.2 Refinement/aggregate-coded symbol 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:

1) Decode the number of symbol instances contained in the aggregation, as specified in 6.5.8.2.1. Let REFAGGNINST be the value decoded.
2) If REFAGGNINST is greater than one, then decode the bitmap itself using a text region decoding procedure as described in 6.4. Set the parameters to this decoding procedure as shown in Table 17.
3) If REFAGGNINST is equal to one, then decode the bitmap as described in 6.5.8.2.2.

#### 6.5.8.2.1 Number of symbol 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 Annex A).

#### 6.5.8.2.2 Decoding a bitmap when REFAGGNINST = 1

If a symbol's bitmap is coded by refinement/aggregate coding, and there is only one symbol in the aggregation, then the bitmap is decoded as follows. This is essentially the procedure followed by the symbol region decoding procedure, except that when a value is known, it is not decoded.

1) Set **SBHUFF** = **SDHUFF**.
2) Decode a symbol ID as described in 6.4.10, using the values of **SBSYMCODES** and **SBSYMCODELEN** described in 6.5.8.2.3. Let $ID_I$ be the value decoded.
3) Decode the instance refinement X offset as described in 6.4.11.3. If **SDHUFF** is **1**, use Table B.15 for **SBHUFFRDX**. Let $RDX_I$ be the value decoded.
4) Decode the instance refinement Y offset as described in 6.4.11.4. If **SDHUFF** is **1**, use Table B.15 for **SBHUFFRDX**. Let $RDY_I$ be the value decoded.
5) If **SDHUFF** is **1** then:
   a) Decode the symbol instance refinement bitmap data size as described in 6.4.11.5, using Table B.1 for **SBHUFFRSIZE**.
   b) Skip over any bits remaining in the last byte read.
6) Let $IBO_I$ be **SBSYMS**[$ID_I$], where **SBSYMS** is as shown in 6.5.8.2.4. The symbol's bitmap 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 18.
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 region decoding procedure must be equal to the value read in step 5 a).

**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 6.5.8.2.3.[a)] |
| **SBSYMCODELEN** | See 6.5.8.2.3.[b)] |
| **SBSYMS** | See 6.5.8.2.4. |
| **SBDEFPIXEL** | **0** |
| **SBCOMBOP** | OR |
| **TRANSPOSED** | **0** |
| **REFCORNER** | TOPLEFT |
| **SBDSOFFSET** | 0 |
| **SBHUFFFS** | Table B.6[a)] |
| **SBHUFFDS** | Table B.8[a)] |
| **SBHUFFDT** | Table B.11[a)] |
| **SBHUFFRDW** | Table B.15[a)] |
| **SBHUFFRDH** | Table B.15[a)] |
| **SBHUFFRDX** | Table B.15[a)] |
| **SBHUFFRDY** | Table B.15[a)] |
| **SBHUFFRSIZE** | Table B.1[a)] |
| **SBRTEMPLATE** | **SDRTEMPLATE** |
| **SBRATX$_1$** | **SDRATX$_1$** |
| **SBRATY$_1$** | **SDRATY$_1$** |
| **SBRATX$_2$** | **SDRATX$_2$** |
| **SBRATY$_2$** | **SDRATY$_2$** |

a) If **SDHUFF** = **0** then this parameter has no value.
b) If **SDHUFF** = **1** then this parameter has no value.

**Table 18 – Parameters used to decode a symbol's bitmap when** REFAGGNINST = 1

| Name | Value |
|---|---|
| **GRW** | SYMWIDTH |
| **GRH** | HCHEIGHT |
| **GRTEMPLATE** | **SDRTEMPLATE** |
| **GRREFERENCE** | $IBO_I$ |
| **GRREFERENCEDX** | $RDX_I$ |
| **GRREFERENCEDY** | $RDY_I$ |
| **TPGRON** | **0** |
| **GRATX$_1$** | **SDRATX$_1$** |
| **GRATY$_1$** | **SDRATY$_1$** |
| **GRATX$_2$** | **SDRATX$_2$** |
| **GRATY$_2$** | **SDRATY$_2$** |

#### 6.5.8.2.3 Setting SBSYMCODES and SBSYMCODELEN

When **SDHUFF** = **1**, set **SBSYMCODES** to an array of **SBNUMSYMS** codes, where the length of each code is:

$$\max(\lceil \log_2 (\mathbf{SDNUMINSYMS} + \mathbf{SDNUMNEWSYMS}) \rceil, 1)$$

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:

$$\lceil \log_2 (\mathbf{SDNUMINSYMS} + \mathbf{SDNUMNEWSYMS}) \rceil$$

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

#### 6.5.8.2.4 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.

### 6.5.9 Height class collective bitmap

This field is only present if **SDHUFF** = **1** and **SDREFAGG** = **0**.

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** Huffman table. 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:

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

   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 decoding procedure as described in 6.2. Set the parameters to this decoding procedure as shown in Table 19.

**Table 19 – 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.

   NOTE – BMSIZE is used to determine the number of bytes of MMR-encoded data, and thus allow the encoder to omit EOFB (see 6.2.6); this usually results in a reduction in encoded data size. It also allows the encoder to transmit the bitmap uncompressed in the cases where MMR coding would result in an expansion.

### 6.5.10 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 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 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:

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

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:

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

   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 patterns into a halftone grid. These patterns are combined to form the decoded bitmap.

NOTE – This form of coding is suitable to efficiently transmitting a bitmap containing periodic halftone image data, such as clustered-dot ordered dithered data. Other forms of halftone image data, such as error-diffused data, may be converted into this form via descreening, or may be coded in a form more closely resembling the original using generic bitmap coding.

### 6.6.2 Input parameters

The parameters to this decoding procedure are shown in Table 20.

**Table 20 – 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.[a)] |
| **HNUMPATS** | Integer | 32 | N | The number of patterns that may be used in this region. |
| **HPATS** | Array of patterns | | | An array containing the patterns used in this region. Contains **HNUMPATS** patterns. |
| **HDEFPIXEL** | Integer | 1 | N | The default pixel for this bitmap. |
| **HCOMBOP** | Operator | | | The combination operator used in this halftone region. May take on the values OR, AND, XOR, XNOR and REPLACE. |
| **HENABLESKIP** | Integer | 1 | N | Whether unneeded gray-scale values are skipped.[a)] |
| **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 | N | 256 times the horizontal coordinate of the grid vector. |
| **HRY** | Integer | 16 | N | 256 times the vertical coordinate of the grid vector. |
| **HPW** | Integer | 8 | N | The width of each pattern. |
| **HPH** | Integer | 8 | N | The height of each pattern. |
| [a)] Unused if **HMMR = 1** | | | | |

### 6.6.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 21.

**Table 21 – Return value 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 22.

### 6.6.5 Decoding the halftone region

A halftone-coded bitmap is represented by a set of pattern instances. Each instance encodes a pattern. The location of each pattern is not coded explicitly but given by a grid global to the entire halftone bitmap. The halftone grid origin is specified by parameters **HGX** and **HGY**. The grid period is defined by parameters **HRX** and **HRY** (see Figure 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.

NOTE 1 – Note that **HRX** and **HRY** are unsigned values; that is, their values are always greater than or equal to zero. This means that the grid vector is restricted to lie in a single quadrant. Despite this restriction, any halftone grid can be encoded by a suitable adjustment of **HGX** and **HGY**: **HGX** and **HGY** must be set so that the grid's origin is the leftmost corner. This is the top left corner in the case where the grid is axis-aligned, or is a slight counter-clockwise rotation of an axis-aligned grid (as shown in Figure 24) and is the bottom left corner in the case where the grid is a slight clockwise rotation of an axis aligned grid.

The possible 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.

**Table 22 – 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.[a] |
| 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. |
| [a] Unused if **HENABLESKIP** = **0**. | | | | |

NOTE 2 – We use the term gray-scale value for the index to illustrate the compression idea. There is no requirement in this Recommendation | International Standard 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 $\lceil \log_2 (\mathbf{HNUMPATS}) \rceil$.
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 23.

   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 patterns are marked by dotted boxes.
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 3 – If **HGX** is 0, **HGY** is 0, **HRX** is equal to **HPW** × 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 patterns. In this case, it is possible to optimise the drawing process, as none of the 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:

$$x = (\mathbf{HGX} + m_g \times \mathbf{HRY} + n_g \times \mathbf{HRX}) >>_A 8$$
$$y = (\mathbf{HGY} + m_g \times \mathbf{HRX} - n_g \times \mathbf{HRY}) >>_A 8$$

      ii) If (($x$ + **HPW** ≤ 0) OR ($x$ ≥ **HBW**) OR ($y$ + **HPH** ≤ 0) OR ($y$ ≥ **HBH**)) then set:

$$\mathrm{HSKIP}[n_g, m_g] = \mathbf{1}$$

      Otherwise, set:

$$\mathrm{HSKIP}[n_g, m_g] = \mathbf{0}$$

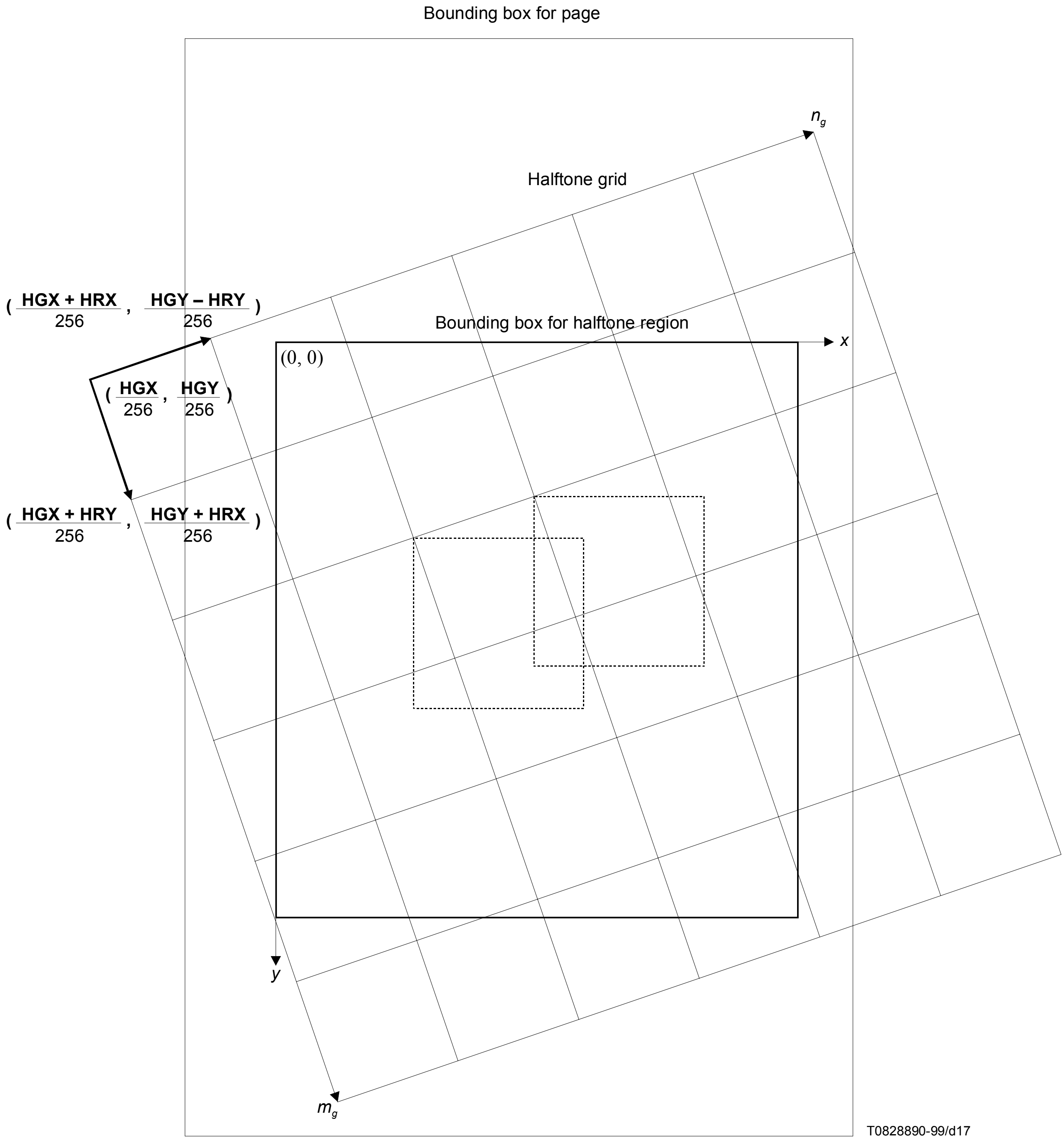



**Figure 24 – Specification of coordinate systems and grid parameters**

**Table 23 – Parameters used to decode a halftone region's gray-scale value array**

| Name | Value |
|---|---|
| **GSMMR** | **HMMR** |
| **GSW** | **HGW** |
| **GSH** | **HGH** |
| **GSBPP** | HBPP |
| **GSUSESKIP** | **HENABLESKIP** |
| **GSKIP** | HSKIP [a)] |
| **GSTEMPLATE** | **HTEMPLATE** [b)] |
| a) If **HENABLESKIP** = **0** then this parameter has no value.<br>b) If **HMMR** = **1** then this parameter has no value. | |

#### 6.6.5.2 Rendering the patterns

Draw the 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:

$$x = (\mathbf{HGX} + m_g \times \mathbf{HRY} + n_g \times \mathbf{HRX}) >>_A 8$$
$$y = (\mathbf{HGY} + m_g \times \mathbf{HRX} - n_g \times \mathbf{HRY}) >>_A 8$$

      ii) Draw the pattern **HPATS**[GI[$n_g$, $m_g$]] into HTREG such that its upper left pixel is at location ($x$, $y$) in HTREG.

      A 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 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 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 Recommendation | International Standard.

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 quantised gray-scale values as indices. Such intermediate halftoned images shall not influence the final halftone-coded bitmap.

## 6.7 Pattern Dictionary Decoding Procedure

### 6.7.1 General description

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

### 6.7.2 Input parameters

The parameters to this decoding procedure are shown in Table 24.

**Table 24 – Parameters for the pattern 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 pattern. |
| **HDPH** | Integer | 32 | N | The height of each 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 patterns.[a)] |
| [a)] Unused if **HDMMR** = **1**. | | | | |

### 6.7.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table 25.

**Table 25 – Return value from the pattern dictionary decoding procedure**

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

### 6.7.4 Variables used in decoding

The variables used by this decoding procedure are shown in Table 26.

**Table 26 – Variables used in the pattern 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. |
| $B_P$ | Bitmap | | | A bitmap of size **HDPW** by **HDPH**. |

### 6.7.5 Decoding the pattern dictionary

The result of decoding a pattern dictionary is a set of patterns: **HDPATS[0]** · · · **HDPATS[GRAYMAX]**. These 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) × **HDPW**. This bitmap contains all the 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 decoding procedure as shown in Table 27.

3) Set:

$$\mathrm{GRAY} = 0$$

4) While GRAY ≤ **GRAYMAX**:

   a) Let the subimage of $B_{HDC}$ consisting of **HPH** rows and columns **HDPW** × GRAY through **HDPW** × (GRAY + 1) – 1 be denoted $B_P$. Set:

$$\mathbf{HDPATS}[\mathrm{GRAY}] = B_P$$

   b) Set:

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

**Table 27 – Parameters used to decode a pattern dictionary's collective bitmap**

| Name | Value |
|---|---|
| **MMR** | **HDMMR** |
| **GBW** | **(GRAYMAX** + 1) × **HDPW** |
| **GBH** | **HDPH** |
| **GBTEMPLATE** | **HDTEMPLATE**[a)] |
| **TPGDON** | **0**[a)] |
| **USESKIP** | **0** |
| **GBATX$_1$** | **–HDPW**[a)] |
| **GBATY$_1$** | 0[a)] |
| **GBATX$_2$** | –3[b)] |
| **GBATY$_2$** | –1[b)] |
| **GBATX$_3$** | 2[b)] |
| **GBATY$_3$** | –2[b)] |
| **GBATX$_4$** | –2[b)] |
| **GBATY$_4$** | –2[b)] |

a) If **HDMMR = 1** then this parameter has no value.
b) If **HDMMR = 1** or **HDTEMPLATE** ≠ 0 then this parameter has no value.

# 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 text region segment may make use of symbols defined in preceding symbol dictionary segments. This is indicated by the text region's segment header referring 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. In the sequential and random-access organisations (see D.1 and D.2), the segments must appear in the file in increasing order of their segment numbers. However, in the embedded organisation (see D.3), this is not the case because the JBIG2 segments are encapsulated in another file format.

NOTE – It is possible for there to be gaps in the segment numbering. A JBIG2 file might contain segments numbered 2, 3, 4, 8 and 10. This can occur due to editing: the segment numbers might originally have been contiguous, but at some point in the life of the file some pages were deleted and the remaining segments not renumbered.

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 must 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 Annex D 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.

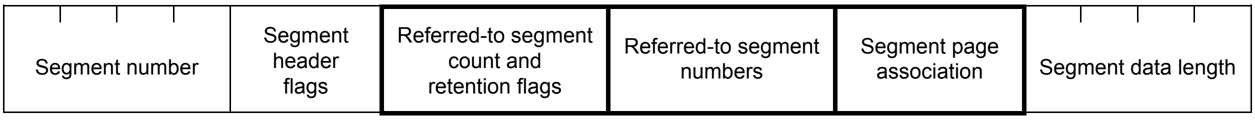



**Figure 25** – **Segment header structure**

**Segment number** – see 7.2.2.

**Segment header flags** – see 7.2.3.

**Referred-to segment count and retention flags** – see 7.2.4.

**Referred-to 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. The valid range of segment numbers is 0 through 4294967295 (`0xFFFFFFFF`) inclusive. As mentioned before, it is possible for there to be gaps in the segment numbering.

### 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.

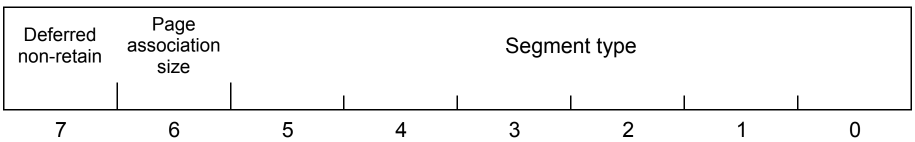



**Figure 26 – Segment header flags**

**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 Referred-to segment count and retention flags

This field contains one or more bytes indicating how many other segments are referred to 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 $4 + \lceil (R + 1)/8 \rceil$ 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 must 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 – Referred-to 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 must contain **0**.

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

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

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

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

**Bit 7** Retain bit for the seventh referred-to segment. If this segment refers to fewer than seven other segments, this field must 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 must 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 referred-to segment count and retention flags field. Each value is the segment number of a segment that this segment refers to. If a segment refers to other segments, it must 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 must 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 shall 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 D.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 0x6B 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, in hexadecimal:

```
00  00  02  34  40  E0  00  00  09  02  FD  01  00  00  02  00
1E  00  05  02  00  02  01  02  02  02  03  02  04  00  00  04
01
```

is parsed as follows:

**`00 00 02 34`** This segment's number is 0x00000234, or 564 decimal.

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

**`E0 00 00 09`** This segment's referred-to segment count field is in the long format. This segment refers to nine other segments.

**`02 FD`** 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.

**`01 00 ... 02 04`** 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.

**`00 00 04 01`** 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 text region – see 7.4.3.

**6** Immediate text region – see 7.4.3.

**7** Immediate lossless text region – see 7.4.3.

**16** Pattern 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** Profiles – see 7.4.12.

**53** Tables – see 7.4.13.

**62** Extension – see 7.4.14.

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

NOTE – These segment type numbers are allocated according to the following rules. The two high-order bits (bits 4-5) 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 region 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** Profiles

**5** Tables

**6-13** Reserved

**14** Extension

**15** Reserved

The segments of types "intermediate text region", "immediate text region", "immediate lossless text 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 text region", "immediate text region", "immediate lossless text 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 text 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 text region", "immediate lossless text 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 text region", "immediate text region" or "immediate lossless text 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 "pattern dictionary" (type 16) must 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) must refer to exactly one segment, and this segment must be of type "pattern dictionary".
- A segment of type "intermediate generic region", "immediate generic region" or "immediate lossless generic region" (type 36, 38 or 39) must not refer to any other segment.
- A segment of type "intermediate generic refinement region" (type 40) must refer to exactly one other segment. This other segment must 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 must be an intermediate region segment.
- A segment of type "page information" (type 48) must not refer to any other segments.
- A segment of type "end of page" (type 49) must not refer to any other segments.
- A segment of type "end of stripe" (type 50) must not refer to any other segments.
- A segment of type "end of file" (type 51) must not refer to any other segments.
- A segment of type "profiles" (type 52) must not refer to any other segments.
- A segment of type "tables" (type 53) must 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 must be associated with some page (i.e. have a non-zero page association field). "Page information", "end of page" and "end of stripe" segments must be associated with some page. "End of file" segments must 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 must 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 must not refer to any segment that is associated with a different page.

## 7.4 Segment syntaxes

This subclause 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 information field

Every region segment's data part begins with a region segment information field; its format is specified here. A region segment information field contains the following subfields, 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.

#### 7.4.1.5 Region segment flags

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

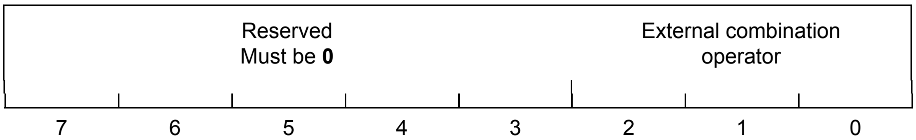



**Figure 29 – Region segment flags field structure**

**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 1 – 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.

NOTE 2 – Intermediate region segments are never combined directly with the page, and so their location and external combination operators are not used. However, these values can still be useful: if a decoder wishes to draw a version of the page before all segments have been decoded (for progressive build-up), then it might want to render intermediate segments; setting the location and external combination according to how the final refinement of that intermediate segment will be combined with the page can help the decoder produce a useful sequence of progressive refinements of the page.

**Bits 3-7** Reserved; must be **0**.

In other words, this region segment information field 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 segment data header

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

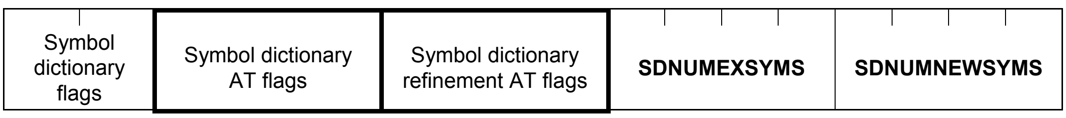



**Figure 30 – Symbol dictionary segment data header structure**

**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:

| 15–13 | 12 | 11–10 | 9 | 8 | 7 | 6 | 5–4 | 3–2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|
| Reserved Must be **0** | SDRTEMP-LATE | **SDRTEMPLATE** | Bitmap coding context retained | Bitmap coding context used | **SDHUFF-AGGINST** selection | **SDHUFF-BMSIZE** selection | **SDHUFFDW** selection | **SDHUFFDH** selection | **SDREF-AGG** | **SDHUFF** |

**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.

**Bit 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** or **SDREFAGG** 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** $\mathbf{SDATX_1}$

**Byte 1** $\mathbf{SDATY_1}$

**Byte 2** $\mathbf{SDATX_2}$

**Byte 3** $\mathbf{SDATY_2}$

**Byte 4** $\mathbf{SDATX_3}$

**Byte 5** $\mathbf{SDATY_3}$

**Byte 6** $\mathbf{SDATX_4}$

**Byte 7** $\mathbf{SDATY_4}$

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** $\mathbf{SDATX_1}$

**Byte 1** $\mathbf{SDATY_1}$

If **SDTEMPLATE** is 1, 2 or 3 then the values of $\mathbf{SDATX_2}$ through $\mathbf{SDATX_4}$ and $\mathbf{SDATY_2}$ through $\mathbf{SDATY_4}$ 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.

#### 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.

| $SDRATX_1$ | $SDRATY_1$ | $SDRATX_2$ | $SDRATY_2$ |
|---|---|---|---|

**Figure 34 – Symbol dictionary refinement AT flags field structure**

**Byte 0** $\mathbf{SDRATX_1}$

**Byte 1** $\mathbf{SDRATY_1}$

**Byte 2** $\mathbf{SDRATX_2}$

**Byte 3** $\mathbf{SDRATY_2}$

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 Number of exported symbols (SDNUMEXSYMS)

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

It is very useful for the decoder to 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 Number of new symbols (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 re-exports some of the symbols that it imported from dictionaries that it refers to, then the dictionary effectively copies those symbols. Those 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 must 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 (Tables B.2 and 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, as described in E.3.8, set the arithmetic coding statistics for the generic region and generic refinement region decoding procedures to the values that they contained at the end of decoding the last-referred-to symbol dictionary segment. That symbol dictionary segment's symbol

dictionary segment data header must have had the "bitmap coding context retained" bit equal to **1**. The values of **SDHUFF**, **SDREFAGG**, **SDTEMPLATE**, **SDRTEMPLATE**, and all of the AT locations (both direct and refinement) for this symbol dictionary must match the corresponding values from the symbol dictionary whose context values are being used.

4) If the "bitmap coding context used" bit in the header was **0**, then, as described in E.3.7, reset all the arithmetic coding statistics for the generic region and generic refinement region decoding procedures to zero.

5) Reset the arithmetic coding statistics 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 28.

7) If the "bitmap coding context retained" bit in the header was **1**, then, as described in E.3.8, preserve the current contents of the arithmetic coding statistics 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 arithmetic coding statistics used for coding bitmaps "learn" the approximate statistics of the symbols in that symbol dictionary. These two steps (3 and 7) allow some limited reuse of these statistics: the statistics learned when decoding the symbol dictionary, that is the last symbol dictionary referred to, are used as a starting point for decoding this symbol dictionary.

Step 7) is explicitly present because not every symbol dictionary's arithmetic coding statistics will be used by another dictionary. Knowing that they will not be used allows the decoder to discard them, reducing memory usage.

**Table 28 – 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 total number of exported symbols from 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. |

### 7.4.3 Text region segment syntax

The data parts of all three of the text region segment types ("intermediate text region", "immediate text region" and "immediate lossless text 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 Text region segment data header

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

**Region segment information field** – see 7.4.1.

**Text region segment flags** – see 7.4.3.1.1.

**Text region segment Huffman flags** – see 7.4.3.1.2.

**Text region segment refinement AT flags** – see 7.4.3.1.3.

**SBNUMINSTANCES –** see 7.4.3.1.4.

**Text region segment symbol ID Huffman decoding table** – see 7.4.3.1.5.

| Region segment information field | Text region segment flags | Text region segment Huffman flags | Text region segment refinement AT flags | SBNUMINSTANCES | Text region segment symbol ID Huffman decoding table |
|---|---|---|---|---|---|

**Figure 35 – Text region segment data header**

##### 7.4.3.1.1 Text region segment flags

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

| SBR-TEMP-LATE | SBDSOFFSET | | | | | SBDEF-PIXEL | SBCOMBOP | | TRANS-POSED | REFCORNER | | LOGSBSTRIPS | | SBREF-INE | SNHUFF |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**Figure 36 – Text 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 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 initial value for every pixel in the text region, before any symbols are drawn.

**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 Text 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.

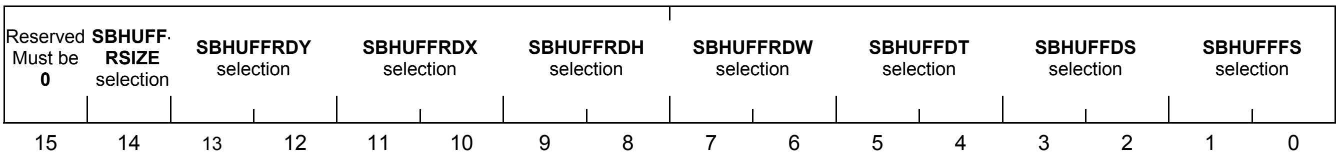



**Figure 37 – Text region Huffman flags field structure**

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

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

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

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

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

**Bit 15** Reserved; must be **0**.

#### 7.4.3.1.3 Text 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 – Text region refinement AT flags field structure**

**Byte 0** $\mathbf{SBRATX_1}$

**Byte 1** $\mathbf{SBRATY_1}$

**Byte 2** $\mathbf{SBRATX_2}$

**Byte 3** $\mathbf{SBRATY_2}$

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 Number of symbol instances (SBNUMINSTANCES)

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

#### 7.4.3.1.5 Text 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 text 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 Text region segment Huffman table selection

Set the values of the parameters **SBHUFFFS**, **SBHUFFDS**, **SBHUFFDT**, **SBHUFFRDW**, **SBHUFFRDH**, **SBHUFFRDX**, **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 must 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 must 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 must 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** symbol ID code lengths; the actual codes in **SBSYMCODES** are assigned from these symbol ID code lengths using the algorithm in B.3.

The symbol ID code lengths themselves are run-length coded and the runs Huffman coded. This is very similar to the "zlib" coded format documented in RFC 1951, though not identical. The encoding is based on the codes shown in Table 29.

Decoding a symbol ID Huffman table proceeds as follows:

1) Read the code lengths for RUNCODE0 through RUNCODE34; each is stored as a four-bit value.
2) Given the lengths, assign Huffman codes for RUNCODE0 through RUNCODE34 using the algorithm in B.3.
3) Read a Huffman code using this assignment. This decodes into one of RUNCODE0 through RUNCODE34. If it is RUNCODE32, read two additional bits. If it is RUNCODE33, read three additional bits. If it is RUNCODE34, read seven additional bits.
4) Interpret the RUNCODE code and the additional bits (if any) according to Table 29. This gives the symbol ID code lengths for one or more symbols.
5) Repeat steps 3) and 4) until the symbol ID code lengths for all **SBNUMSYMS** symbols have been determined.
6) Skip over the remaining bits in the last byte read, so that the actual text region decoding procedure begins on a byte boundary.
7) Assign a Huffman code to each symbol by applying the algorithm in B.3 to the symbol ID code lengths just decoded. The result is the symbol ID Huffman table **SBSYMCODES**.

**Table 29 – Meaning of the run codes**

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

EXAMPLE 1 – Suppose that **SBNUMSYMS** is 32 and the symbol ID 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 symbol ID 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
```

Interpretation of this sequence of bytes can be separated into the following three steps:

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 symbol ID code length of zero.

2) The algorithm of B.3 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** | | | | |

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 provides the following results:

| | |
|---|---|
| **11111 000** | RUNCODE33(0) – that is, RUNCODE33 followed by three bits containing the value 0, indicating a run of three zero lengths |
| **101** | RUNCODE9 |
| **100** | RUNCODE6 |
| **110 00** | RUNCODE32(0) – that is, 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 29, the desired sequence of symbol ID code lengths is decoded.

EXAMPLE 2 – This example describes how an encoder might generate an encoded symbol ID Huffman table. The symbol ID table is identical to that in the previous example.

Suppose that a text region refers to a dictionary containing 32 symbols, and that each symbol is used as follows:

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 8 | 8 | 8 | 8 | 64 | 32 | 32 | 32 | 32 | 32 | 32 | 0 |
| 4 | 1 | 2 | 4 | 16 | 16 | 16 | 16 | 16 | 16 | 64 | 8 | 4 | 32 | 4 | 4 |

For example, the first, second and third symbols in the symbol dictionary are not used at all, the fourth symbol is used once, the fifth symbol is used eight times, and so on.

Table 30 then shows, from right to left, the progression of the encoding.

**Table 30 – Example of symbol ID Huffman table encoding**

| Symbol | Use count | Symbol ID code length | Runs | RUNCODEs |
|---|---|---|---|---|
| Symbol #1 | 0 | 0 | Length 3 run of 0 | RUNCODE33(0) |
| Symbol #2 | 0 | 0 | | |
| Symbol #3 | 0 | 0 | | |
| Symbol #4 | 1 | 9 | Length 1 run of 9 | RUNCODE9 |
| Symbol #5 | 8 | 6 | Length 4 run of 6 | RUNCODE6 |
| Symbol #6 | 8 | 6 | | RUNCODE32(0) |
| Symbol #7 | 8 | 6 | | |
| Symbol #8 | 8 | 6 | | |
| Symbol #9 | 64 | 3 | Length 1 run of 3 | RUNCODE3 |
| Symbol #10 | 32 | 4 | Length 6 run of 4 | RUNCODE4 |
| Symbol #11 | 32 | 4 | | RUNCODE32(2) |
| Symbol #12 | 32 | 4 | | |
| Symbol #13 | 32 | 4 | | |
| Symbol #14 | 32 | 4 | | |
| Symbol #15 | 32 | 4 | | |
| Symbol #16 | 0 | 0 | Length 1 run of 0 | RUNCODE0 |
| Symbol #17 | 4 | 7 | Length 1 run of 7 | RUNCODE7 |
| Symbol #18 | 1 | 9 | Length 1 run of 9 | RUNCODE9 |
| Symbol #19 | 2 | 8 | Length 1 run of 8 | RUNCODE8 |
| Symbol #20 | 4 | 7 | Length 1 run of 7 | RUNCODE7 |
| Symbol #21 | 16 | 5 | Length 6 run of 5 | RUNCODE5 |
| Symbol #22 | 16 | 5 | | RUNCODE32(2) |
| Symbol #23 | 16 | 5 | | |
| Symbol #24 | 16 | 5 | | |
| Symbol #25 | 16 | 5 | | |
| Symbol #26 | 16 | 5 | | |
| Symbol #27 | 64 | 3 | Length 1 run of 3 | RUNCODE3 |
| Symbol #28 | 8 | 6 | Length 1 run of 6 | RUNCODE6 |
| Symbol #29 | 4 | 7 | Length 1 run of 7 | RUNCODE7 |
| Symbol #30 | 32 | 4 | Length 1 run of 4 | RUNCODE4 |
| Symbol #31 | 4 | 7 | Length 2 run of 7 | RUNCODE7 |
| Symbol #32 | 4 | 7 | | RUNCODE7 |

Using a standard Huffman tree algorithm, the code lengths shown in the "Symbol ID code length" column are assigned to the symbols (where a symbol ID code length of 0 represents "unused"). Next, those code lengths are grouped into runs, as shown in the "Runs" column. Following that, each run is expressed as one or more RUNCODEs, each one potentially with some extra bits. For example, RUNCODE32(2) represents RUNCODE32, followed by two bits encoding the value "2", meaning "Copy the previous symbol ID code length 5 times".

Once that has been done, the number of times each RUNCODE is used is counted. These counts are as follows (unused RUNCODEs are not shown):

| | | | | | |
|---|---|---|---|---|---|
| RUNCODE0 | 1 | RUNCODE3 | 2 | RUNCODE4 | 2 |
| RUNCODE5 | 1 | RUNCODE6 | 2 | RUNCODE7 | 5 |
| RUNCODE8 | 1 | RUNCODE9 | 2 | RUNCODE32 | 3 |
| RUNCODE33 | 1 | | | | |

These counts are then converted into code lengths using a standard Huffman tree algorithm:

| | | | | | |
|---|---|---|---|---|---|
| RUNCODE0 | 5 | RUNCODE3 | 3 | RUNCODE4 | 3 |
| RUNCODE5 | 5 | RUNCODE6 | 3 | RUNCODE7 | 2 |
| RUNCODE8 | 5 | RUNCODE9 | 3 | RUNCODE32 | 3 |
| RUNCODE33 | 5 | | | | |

The algorithm of B.3 assigns the following Huffman codes to the run codes:

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

and these Huffman codes are then used to encode the "RUNCODEs" column of Table 30:

| | |
|---|---|
| **11111 000** | RUNCODE33(0) |
| **101** | RUNCODE9 |
| **100** | RUNCODE6 |
| **110 00** | RUNCODE32(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 |

The encoder now emits the encoded RUNCODE code lengths, followed by the sequence of RUNCODEs, plus four bits of padding to fill the last byte, yielding the sequence of bytes:

```
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
```

#### 7.4.3.2 Decoding a text region segment

A text 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) As described in E.3.7, reset all the arithmetic coding statistics to zero.
4) Invoke the text region decoding procedure described in 6.4, with the parameters to the text region decoding procedure set as shown in Table 31.

**Table 31 – Parameters used to decode a text 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 total 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** | $\lceil \log_2 \mathbf{SBNUMSYMS} \rceil$ |
| **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. |
| **$\mathbf{SBRATX_1}$** | See 7.4.3.1.3. |
| **$\mathbf{SBRATY_1}$** | See 7.4.3.1.3. |
| **$\mathbf{SBRATX_2}$** | See 7.4.3.1.3. |
| **$\mathbf{SBRATY_2}$** | See 7.4.3.1.3. |

### 7.4.4 Pattern dictionary segment syntax

#### 7.4.4.1 Pattern dictionary segment data header

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

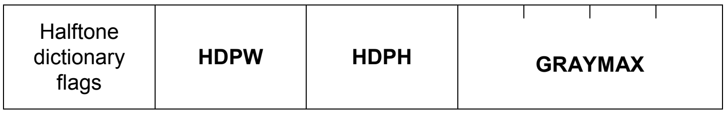



**Figure 39 –Pattern dictionary header structure**

**Pattern dictionary flags** – see 7.4.4.1.1.

**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 Pattern dictionary flags

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

**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 patterns if **HDMMR** is **0**. If **HDMMR** is **1**, this field must contain the value 0.

**Bits 3-7** Reserved; must be **0**.


**Figure 40 – Pattern dictionary flags field structure**

##### 7.4.4.1.2 Width of the patterns in the pattern dictionary (HDPW)

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

##### 7.4.4.1.3 Height of the patterns in the pattern dictionary (HDPH)

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

##### 7.4.4.1.4 Largest gray-scale value (GRAYMAX)

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

#### 7.4.4.2 Decoding a pattern dictionary segment

A pattern dictionary segment is decoded according to the following steps:

1) Interpret its header, as described in 7.4.4.1.

2) As described in E.3.7, reset all the arithmetic coding statistics to zero.

3) Invoke the pattern dictionary decoding procedure described in 6.7, with the parameters to the pattern dictionary decoding procedure set as shown in Table 32.

**Table 32 – Parameters used to decode a pattern 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. |

### 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.

#### 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 information field | Halftone region segment flags | Halftone grid position and size | Halftone grid step sizes |

**Figure 41 – Halftone region segment data header structure**

**Region segment information field** – 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.

| HDEF-PIXEL | HCOMBOP | | | HENABLE-SKIP | HTEMPLATE | | HMMR |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**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 five possible combination operators:

**0** OR
**1** AND
**2** XOR
**3** XNOR
**4** REPLACE

**Bit 7** **HDEFPIXEL**

This bit contains the initial value for every pixel in the halftone region, before any patterns are drawn.

#### 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.

| HGW | HGH | HGX | HGY |
|---|---|---|---|

**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 Width of the gray-scale image (HGW)

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

##### 7.4.5.1.2.2 Height of the gray-scale image (HGH)

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

##### 7.4.5.1.2.3 Horizontal offset of the grid (HGX)

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

##### 7.4.5.1.2.4 Vertical offset of the grid (HGY)

This signed 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.

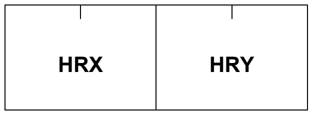



**Figure 44 – Halftone grid vector field structure**

**HRX –** See 7.4.5.1.3.1.

**HRY –** See 7.4.5.1.3.2.

##### 7.4.5.1.3.1 Horizontal coordinate of the halftone grid vector (HRX)

This unsigned two-byte field contains 256 times the horizontal coordinate of the halftone grid vector.

##### 7.4.5.1.3.2 Vertical coordinate of the halftone grid vector (HRY)

This unsigned two-byte field contains 256 times the vertical coordinate 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 pattern dictionary segment.
3) As described in E.3.7, reset all the arithmetic coding statistics to zero.
4) Invoke the halftone region decoding procedure described in 6.6, with the parameters to the halftone region decoding procedure set as shown in Table 33.

**Table 33 – 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 patterns in the pattern dictionary segment referred to by this segment. |
| **HPATS** | The patterns in the pattern dictionary segment referred to by this segment. |
| **HPW** | The width, in pixels, of each of the patterns contained in **HPATS**. |
| **HPH** | The height, in pixels, of each of the 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.

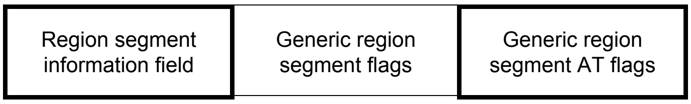



**Figure 45 – Generic region segment data header structure**

**Region segment information field** – 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.

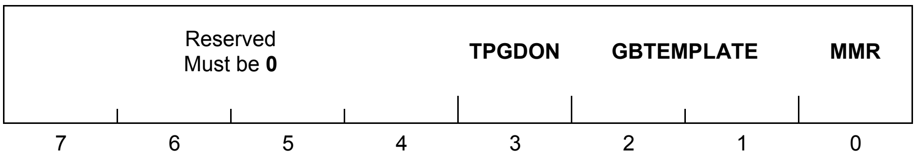



**Figure 46 – Generic region segment flags field structure**

**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 must contain the value zero.

**Bit 3** **TPGDON**

This field specifies whether typical prediction for generic direct coding is used.

**Bits 4-7** Reserved; must 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.

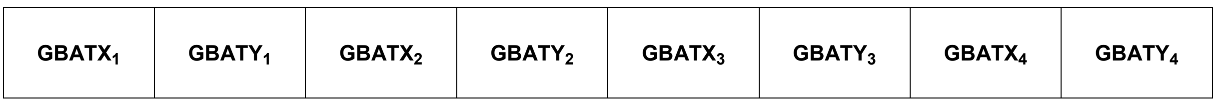



**Figure 47 – Generic region AT flags field structure when GBTEMPLATE is 0**

**Byte 0** **$GBATX_1$**

**Byte 1** **$GBATY_1$**

**Byte 2** **$GBATX_2$**

**Byte 3** **$GBATY_2$**

**Byte 4** **$GBATX_3$**

**Byte 5** **$GBATY_3$**

**Byte 6** **$GBATX_4$**

**Byte 7** **$GBATY_4$**

If **GBTEMPLATE** is 1, 2 or 3, it is a two-byte field formatted as shown in Figure 48 and as described below. If **GBTEMPLATE** is 1, 2 or 3 then the values of **$GBATX_2$** through **$GBATX_4$** and **$GBATY_2$** through **$GBATY_4$** are all zero.

| | |
|---|---|
| **$GBATX_1$** | **$GBATY_1$** |

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

**Byte 0** **$GBATX_1$**

**Byte 1** **$GBATY_1$**

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) As described in E.3.7, reset all the arithmetic coding statistics 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 34.

**Table 34 – 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. |
| **TPGDON** | As shown in 7.4.6.2. |
| **USESKIP** | **0** |
| **GBW** | As specified by the region segment bitmap width in this segment's region segment data header. |
| **GBH** | As specified by the region segment bitmap height in this segment's region segment data header. |
| **$GBATX_1$** | See 7.4.6.3. |
| **$GBATY_1$** | See 7.4.6.3. |
| **$GBATX_2$** | See 7.4.6.3. |
| **$GBATY_2$** | See 7.4.6.3. |
| **$GBATX_3$** | See 7.4.6.3. |
| **$GBATY_3$** | See 7.4.6.3. |
| **$GBATX_4$** | See 7.4.6.3. |
| **$GBATY_4$** | 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 than are indicated in the segment's region segment information field.

In order for the decoder to correctly decode the segment, it needs to 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 information field.

NOTE – The sequence `0x00 0x00` cannot occur 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.


**Figure 49 – Generic refinement region segment data header structure**

**Region segment information field** – 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.

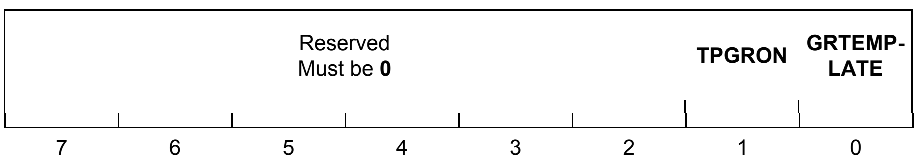



**Figure 50 – Generic refinement region segment flags field structure**

**Bit 0** **GRTEMPLATE**

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

**Bit 1** **TPGRON**

This field specifies whether typical prediction for generic refinement is used.

**Bits 2-7** Reserved; must be zero.

#### 7.4.7.3 Generic refinement 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.

| | | | |
|---|---|---|---|
| $GRATX_1$ | $GRATY_1$ | $GRATX_2$ | $GRATY_2$ |

**Figure 51 – Generic refinement region AT flags field structure**

**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.

#### 7.4.7.4 Reference bitmap selection

If this segment refers to another region segment, then set the reference bitmap **GRREFERENCE** to be the current contents of the auxiliary 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 clause 8), restricted to the area of the page buffer specified by this segment's region segment information field.

#### 7.4.7.5 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 7.4.7.1. If this segment does not refer to another region segment then its external combination operator must be REPLACE. If it does refer to another region segment, then this segment's region bitmap size, location, and external combination operator must be equal to that other segment's region bitmap size, location, and external combination operator.

   NOTE – The requirement that the locations and external combination operators match is present to assist decoders that want to produce images of a page that is only partially decoded: it ensures that the final location and external combination operator is known for all intermediate segments. These partially-decoded page images are outside the scope of this Recommendation | International Standard.

2) As described in E.3.7, reset all the arithmetic coding statistics 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 35.

### 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** – see 7.4.8.1.

**Page bitmap height** – see 7.4.8.2.

**Page X resolution** – see 7.4.8.3.

**Table 35 – Parameters used to decode a generic refinement region segment**

| Name | Value |
|---|---|
| **GRTEMPLATE** | As shown in 7.4.6.2. |
| **TPGRON** | 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 7.4.7.4. |
| **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 bitmap width | Page bitmap height | Page X resolution | Page Y resolution | Page segment flags | Page striping information |

**Figure 52 – Page information segment structure**

**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 must 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 the 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 must 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 value containing the resolution of the original page medium, measured in pixels/metre in the horizontal direction. If this value is unknown, then this field must contain `0x00000000`.

#### 7.4.8.4 Page Y resolution

This is a four-byte value containing the resolution of the original page medium, measured in pixels/metre in the vertical direction. If this value is unknown, then this field must 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. This bit contains the initial value for every pixel in the page, before any region segments are decoded or drawn.

**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 page 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 direct region segment associated with this page must use the page's default combination operator. If this bit is **1**, then direct region segments associated with this page may use any combination operators.

NOTE 1 – All region segments, except for refinement region segments, are direct region segments. Because of the requirements in 7.4.7.5 restricting the external combination operators of refinement region segments, if this bit is **0**, then refinement region segments associated with this page that refer to no region segments must have an external combination operator of REPLACE, and all other region segments associated with this page must have the external combination operator specified by this page's "Page default combination operator".

NOTE 2 – If all the direct 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 to 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; must 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.

| Page is striped | Maximum stripe size | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |

**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 stripe 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 must be zero.

The last segment that is associated with any page must be an end of page segment. Each page must have exactly one end of page segment associated with it.

If a page's height was originally unknown, then there must 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 page with which the segment is associated, 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; furthermore, no segment preceding the end of stripe may modify any portion of the page bitmap that lies below that row. This row is called the "end row" of the stripe.

NOTE 1 – In some cases, the decoder may only have a limited amount of buffer memory for the page bitmap, smaller than the size of the page. The decoder needs to be told when it is able to output the current buffer contents and clear the buffer for the next stripe of the page.

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

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

NOTE 2 – An end of stripe segment is used to communicate the size of the page in this case.

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

NOTE 3 – If a stripe boundary breaks a line of text into two parts, a top part and a bottom part, the characters straddling the stripe boundary are also broken. One way of handling these straddling characters is to code them with a generic region, or to add the top halves and bottom halves to a symbol dictionary and use those top and bottom halves as other symbols.

However, if the encoder has more buffer space than the decoder, a more efficient encoding method is possible: the encoder can (temporarily) ignore the stripe boundary, and generate a list of symbol instances, and a symbol dictionary. It can then encode a text region in each stripe; the first text region contains all the symbol instances that affect the portion of the page above the stripe boundary; the second text region contains all the symbol instances that affect the portion below the stripe boundary.

This means that some symbol instances appear twice, both times using the same symbol in the symbol dictionary. The first appearance encodes the top half of the character, where the bottom half is clipped off by the drawing rules used in the text region decoding procedure. The second appearance likewise encodes the bottom half of the character: the entire character is encoded, but the top half is clipped off. Thus, this encoding method reduces the amount of symbol dictionary data required.

### 7.4.11 End of file segment syntax

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

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

### 7.4.12 Profiles segment syntax

A profiles segment contains a list of the profiles that a given JBIG2 data stream is in compliance with. If any profiles segments are present, then the first segment of the data stream must be a profiles segment, and must not be associated with any page. Profiles of this Recommendation | International Standard are listed in Annex F.

A 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 must be in compliance with each of the profiles listed.

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

NOTE – The global profiles segment allows a decoder to find out quickly that it cannot decode a given data stream. Allowing each page to contain a possibly different (though more restrictive) profiles segment eases moving pages from one file to another.

#### 7.4.13 Code table segment syntax

A code table segment's syntax is described in Annex 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 Recommendation | International 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 for future revisions of this Recommendation | International Standard.

**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 needs to 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.

EXAMPLE – An extension segment containing a CRC of the segment that it refers to should be flagged as dependent.

**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.

NOTE – This is intended to facilitate future extensions to JBIG2, such as coding improvements. If this bit is **1**, then a decoder that does not understand the extension knows that it has encountered data necessary to the correct decoding of the page that it cannot handle. For example, an extension segment containing a region that is coded with some new method would be flagged as "necessary", as without that region the page image is not complete. Another example might be an extension segment containing a set of colours that should be applied to the symbols on the page as they are drawn.

If the "necessary" bit is **1**, then the "reserved" bit must also be **1**.

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.

`0x20000000` Single-byte coded comment. See 7.4.15.1.

`0x20000002` Multi-byte coded comment. See 7.4.15.2.

##### 7.4.15.1 Single-byte coded comment

A single-byte coded 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.

A single-byte coded comment segment contains a number of (name, value) pairs. Each element of each pair is a string of characters, and is terminated by a `0x00` byte. The last pair is followed by an additional `0x00` byte. The other bytes shall be interpreted according to ISO/IEC 8859-1:1998.

NOTE – A single-byte coded comment extension segment may contain any valid ISO/IEC 8859-1 character, including those whose values are greater than 127.

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 printable 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 Multi-byte coded comment

A multi-byte coded comment extension segment is formatted in the same manner as a single-byte coded comment extension segment, except that the individual characters each occupy two bytes, in the ISO/IEC 10646-1:1993 encoding (UCS-2). Each element of each pair in the comment is terminated by a `0x0000` and the final pair is followed by an additional `0x0000.`

# 8 Page Make-up

## 8.1 Decoder model

This clause describes the result that a decoder conforming to this Recommendation | International Standard shall 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 shall 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 region 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 region segments associated with that page. Each region segment describes some of the contents of a rectangular region of the page. Since these regions of the page may overlap, and since some parts of the page might be described at multiple levels of refinement, it is important to define what the rules for region 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.

As described in 7.4.8, each page has a default pixel value (**0** or **1**) and one of four combination operators (OR, AND, XOR, XNOR); these are specified in its page information segment. Each region segment also specifies a combination operator of its own. The "page combination operator overridden" flag bit in the page information segment specifies whether any of the page's direct region segments overrides the page combination operator. If the bit is **0**, then no direct region segment associated with this page overrides the page combination operator. The decoder may use this information to optimise its decoding.

The result of decoding a region segment is a bitmap. The size of this bitmap and its location with respect to the page buffer are given in the region segment information field.

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

1) Decode the page information segment.

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

   If the page height 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 initially equal to this maximum stripe height. As each end of stripe segment is encountered, the page buffer's height can be increased, so that the last row in the new buffer is the maximum stripe height plus the end row of the previous stripe. The end of page segment (together with the last end of stripe segment) allows determination of the page's actual height.

   Alternately, when the page height is unknown, the decoder may use a fixed-size buffer whose height is equal to the page's maximum stripe height. As each end of stripe segment is encountered, the decoder can print, or copy to some other location, all the rows in this buffer up to and including the stripe's end row, then clear the buffer in preparation for the next stripe. The decoder may follow this strategy whenever the page is striped, even if the page height is known beforehand.

   NOTE 1 – The steps below can be followed regardless of which striping strategy is followed. The restrictions imposed by striping ensure that once an end of stripe segment is seen, no part of the page above or on that stripe's end row can be modified, and so the presentation below is phrased in terms of a page buffer that is the full size of the page, even when the page's height is not known initially.

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

4) Fetch the next region segment associated with that page.

5) The following cases exist:

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

   b) The region segment is an intermediate direct region segment. In this case, allocate a new auxiliary buffer, using the size and location specified in the segment's region segment information field. This buffer is initially associated with the region segment. Decode the region segment, placing the resulting bitmap into the auxiliary buffer.

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

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

   e) The region segment is an intermediate refinement region segment. This region segment must refer to one other region segment, which must be a previously occurring intermediate region segment that has not yet had a refinement region segment refer to it; the other region 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 region 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 region segment, and is no longer associated with the other region segment.

6) Repeat steps 4) and 5) until there are no more region segments associated with the page. At this point, all auxiliary buffers that have been allocated should have been refined, drawn into the page, and discarded, as described in step 5 d); no auxiliary buffers should remain.

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

The rules described in step 5) are quite simple in principle. Immediate region segments are to be drawn into the page buffer, either by simply drawing them (direct segments, step 5 a), by refining a part of the page buffer (refinement segments referring to no other segments, step 5 c), or by refining and then drawing an auxiliary buffer (refinement segments referring to some other segment, step 5 d). Intermediate region segments involve creating an auxiliary buffer containing the region bitmap (direct segments, step 5 b), or replacing the current contents of an auxiliary buffer (refinement segments, step 5 e).

Some examples of these rules in operation:

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

EXAMPLE 2 – The page information segment for page 1 specifies that the page default combination operator is OR and the page default pixel value is **0**. The region segments associated with page 1 are, in order:

- Segment 3, an immediate lossless text region segment whose external combination operator is OR;
- Segment 4, an immediate lossless generic region segment whose external combination operator is OR;
- Segment 6, an immediate lossless halftone region segment whose external combination operator is OR.

The resulting page bitmap can be obtained by decoding segments 3, 4 and 6, and drawing each one at its specified region location, using OR, into a bitmap initially containing **0** everywhere. Note that the order in which these three segments are decoded and drawn does not affect the resulting page bitmap. Also, if segment 3 has an internal combination operator of OR and a default pixel value of **0**, then it may be drawn by simply drawing the symbol instances directly into the page buffer; it is not necessary to decode it into a temporary bitmap then draw that bitmap into the page buffer. A similar observation holds for segment 6.

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

- Segment 7, an intermediate text region segment;
- Segment 8, an intermediate generic bitmap region segment;
- Segment 13, an immediate generic bitmap refinement region segment whose external combination operator is OR that refers to segment 8;
- Segment 14, an immediate generic bitmap refinement region segment whose external combination operator is OR that refers to segment 7;
- Segment 19, an immediate text region segment whose external combination operator is OR;
- Segment 22, an immediate generic bitmap region 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, discarding the auxiliary buffer after this is done;
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, discarding the auxiliary buffer after this is done;
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 4 – If a page contains several immediate direct-coded region segments that do not override the page's combination operator, and an immediate refinement region 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 region segments that precede the refinement region segment;
- refining the portion of the region covered by the refinement region segment;
- drawing all the direct-coded region segments that follow the refinement region segment.

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

NOTE 2 – 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 page bitmaps that it displays are entirely up to the decoder, and are not specified by this Recommendation | International Standard.

One potential strategy a decoder could use is to take the current contents of the page buffer and any currently active auxiliary buffers, and combine all of these buffers using the page's default 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 5 – A decoder might notice that an intermediate region segment refers to a region of the page that is not overlapped by any other region segment, and so might not actually allocate an auxiliary buffer for that region 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 region segments.

# Annex A

# Arithmetic Integer Decoding Procedure

(This annex forms an integral part of this Recommendation | International Standard)

## A.1 General description

This Recommendation | 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.

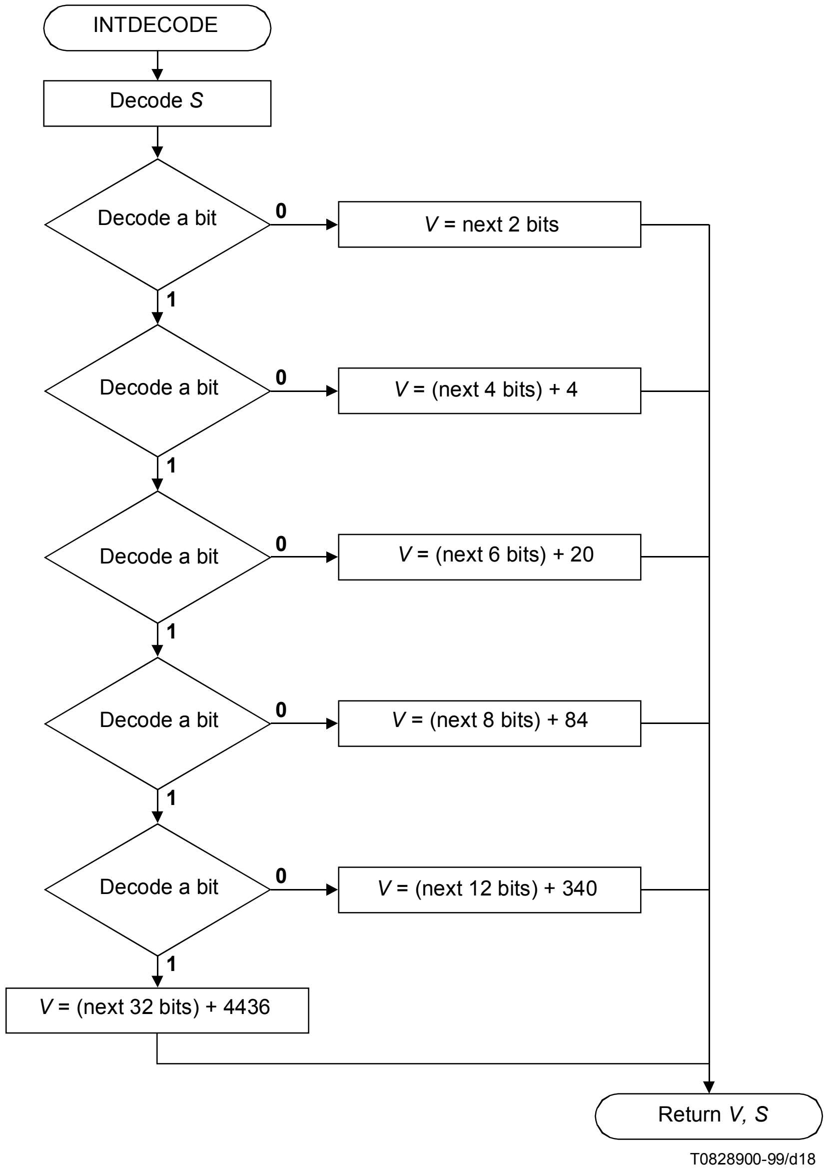



**Figure A.1 – Flowchart for the integer arithmetic decoding procedures (except** IAID**)**

3) After each bit is decoded: If PREV < 256 set:

$$\text{PREV} = (\text{PREV} \ll 1) \text{ OR D}$$

Otherwise set:

$$\text{PREV} = (((\text{PREV} \ll 1) \text{ OR D}) \text{ AND } 511) \text{ OR } 256$$

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.

**Table A.1 – Arithmetic integer decoding procedure table**

| VAL | Encoding |
|---|---|
| 0 . . . 3 | **00** + VAL encoded as 2 bits |
| –1 | **1001** |
| –3 . . . –2 | **101** + (–VAL – 2) encoded as 1 bit |
| 4 . . . 19 | **010** + (VAL – 4) encoded as 4 bits |
| –19 . . . –4 | **110** + (–VAL – 4) encoded as 4 bits |
| 20 . . . 83 | **0110** + (VAL – 20) encoded as 6 bits |
| –83 . . . –20 | **1110** + (–VAL – 20) encoded as 6 bits |
| 84 . . . 339 | **01110** + (VAL – 84) encoded as 8 bits |
| –339 . . . –84 | **11110** + (–VAL – 84) encoded as 8 bits |
| 340 . . . 4435 | **011110** + (VAL – 340) encoded as 12 bits |
| –4435 . . . –340 | **111110** + (–VAL – 340) encoded as 12 bits |
| 4436 . . . ∞ | **011111** + (VAL – 4436) encoded as 32 bits |
| –∞ . . . –4436 | **111111** + (–VAL – 4436) encoded as 32 bits |
| OOB | **1000** |

EXAMPLE – An invocation of IADW might go as follows:

- Set CX to "IADW**000000001**". This identifies a particular adaptive probability estimate identified. Decode a bit. Suppose the value decoded (D) 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 procedures. 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 text 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:

$$\text{PREV} = 1$$

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:

$$\text{PREV} = (\text{PREV} \ll 1) \text{ OR D}$$

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:

$$\text{PREV} = \text{PREV} - 2^{\mathbf{SBSYMCODELEN}}$$

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 $2^{\mathbf{SBSYMCODELEN}}$, 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 **SBSYMCODELEN** = 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 identification 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

# Huffman Table Decoding Procedure

(This annex forms an integral part of this Recommendation | International Standard)

## B.1 General description

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, which is an out-of-band signal to the decoding procedure using the table.

## B.2 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 special additional table lines that encode "open-ended" ranges. The smallest value that can be encoded in a table described according to this Recommendation | International Standard is –2147483648 ($-2^{31}$) and the largest value is 2147483647 ($2^{31} - 1$), so these ranges are not really open-ended. There is also, potentially, an additional special 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.2.1. This sets the values HTOOB, HTPS and HTRS.

2) Decode the code table lowest value field as described in B.2.2. Let HTLOW be the value decoded.

3) Decode the code table highest value field as described in B.2.3. 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:

$$\begin{aligned} \text{RANGELOW[NTEMP]} &= \text{CURRANGELOW} \\ \text{CURRANGELOW} &= \text{CURRANGELOW} + 2^{\text{RANGELEN[NTEMP]}} \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

d) If CURRANGELOW ≥ HTHIGH then proceed to step 6).

6) Read HTPS bits. Let LOWPREFLEN be the value read.

7) Set:

$$\begin{aligned} \text{PREFLEN[NTEMP]} &= \text{LOWPREFLEN} \\ \text{RANGELEN[NTEMP]} &= 32 \\ \text{RANGELOW[NTEMP]} &= \text{HTLOW} - 1 \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

This is the lower range table line for this table.

8) Read HTPS bits. Let HIGHPREFLEN be the value read.

9) Set:

$$\begin{aligned} \text{PREFLEN[NTEMP]} &= \text{HIGHPREFLEN} \\ \text{RANGELEN[NTEMP]} &= 32 \\ \text{RANGELOW[NTEMP]} &= \text{HTHIGH} \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

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:

$$\begin{aligned} \text{PREFLEN[NTEMP]} &= \text{OOBPREFLEN} \\ \text{NTEMP} &= \text{NTEMP} + 1 \end{aligned}$$

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.3.

### B.2.1 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.2.2 Code table lowest value

This signed four-byte field is the lower bound of the first table line in the encoded table.

### B.2.3 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.3 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 in the array LENCOUNT counting the number of times each prefix length value occurs in PREFLEN: 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 ≤ 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.4 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:

$$\text{HTVAL} = \text{OOB}$$

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.2 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:

| | |
|---|---|
| **01 00100** | 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. |
| **10 01000** | This table line has a prefix length of 2 and a range length of 8. |
| **11 10000** | 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. 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.3 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.5 Standard Huffman tables

This subclause presents some standard Huffman tables that 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 . . . 15 | 1 | 4 | **0** + VAL encoded as 4 bits |
| 16 . . . 271 | 2 | 8 | **10** + (VAL – 16) encoded as 8 bits |
| 272 . . . 65807 | 3 | 16 | **110** + (VAL – 272) encoded as 16 bits |
| 65808 . . . ∞ | 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 . . . 10 | 4 | 3 | **1110** + (VAL – 3) encoded as 3 bits |
| 11 . . . 74 | 5 | 6 | **11110** + (VAL – 11) encoded as 6 bits |
| 75 . . . ∞ | 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 . . . –1 | 8 | 8 | **11111110** + (VAL + 256) encoded as 8 bits |
| 0 | 1 | 0 | **0** |
| 1 | 2 | 0 | **10** |
| 2 | 3 | 0 | **110** |
| 3 . . . 10 | 4 | 3 | **1110** + (VAL – 3) encoded as 3 bits |
| 11 . . . 74 | 5 | 6 | **11110** + (VAL – 11) encoded as 6 bits |
| –∞ . . . –257 | 8 | 32 | **11111111** + (–257 – VAL) encoded as 32 bits |
| 75 . . . ∞ | 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 . . . 11 | 4 | 3 | **1110** + (VAL – 4) encoded as 3 bits |
| 12 . . . 75 | 5 | 6 | **11110** + (VAL – 12) encoded as 6 bits |
| 76 . . . ∞ | 5 | 32 | **11111** + (VAL – 76) encoded as 32 bits |

**Table B.5 – Standard Huffman table E**

| | |
|---|---|
| HTOOB | **0** |

| VAL | PREFLEN | RANGELEN | Encoding |
|---|---|---|---|
| –255 . . . 0 | 7 | 8 | **1111110** + (VAL + 255) encoded as 8 bits |
| 1 | 1 | 0 | **0** |
| 2 | 2 | 0 | **10** |
| 3 | 3 | 0 | **110** |
| 4 . . . 11 | 4 | 3 | **1110** + (VAL – 4) encoded as 3 bits |
| 12 . . . 75 | 5 | 6 | **11110** + (VAL – 12) encoded as 6 bits |
| –∞ . . . –256 | 7 | 32 | **1111111** + (–256 – VAL) encoded as 32 bits |
| 76 . . . ∞ | 6 | 32 | **111110** + (VAL – 76) encoded as 32 bits |

**Table B.6 – Standard Huffman table F**

| | |
|---|---|
| HTOOB | **0** |

| VAL | PREFLEN | RANGELEN | Encoding |
|---|---|---|---|
| –2048 . . . –1025 | 5 | 10 | **11100** + (VAL + 2048) encoded as 10 bits |
| –1024 . . . –513 | 4 | 9 | **1000** + (VAL + 1024) encoded as 9 bits |
| –512 . . . –257 | 4 | 8 | **1001** + (VAL + 512) encoded as 8 bits |
| –256 . . . –129 | 4 | 7 | **1010** + (VAL + 256) encoded as 7 bits |
| –128 . . . –65 | 5 | 6 | **11101** + (VAL + 128) encoded as 6 bits |
| –64 . . . –33 | 5 | 5 | **11110** + (VAL + 64) encoded as 5 bits |
| –32 . . . –1 | 4 | 5 | **1011** + (VAL + 32) encoded as 5 bits |
| 0 . . . 127 | 2 | 7 | **00** + VAL encoded as 7 bits |
| 128 . . . 255 | 3 | 7 | **010** + (VAL – 128) encoded as 7 bits |
| 256 . . . 511 | 3 | 8 | **011** + (VAL – 256) encoded as 8 bits |
| 512 . . . 1023 | 4 | 9 | **1100** + (VAL – 512) encoded as 9 bits |
| 1024 . . . 2047 | 4 | 10 | **1101** + (VAL – 1024) encoded as 10 bits |
| -∞ . . . –2049 | 6 | 32 | **111110** + (–2049 – VAL) encoded as 32 bits |
| 2048 . . . ∞ | 6 | 32 | **111111** + (VAL – 2048) encoded as 32 bits |

**Table B.7 – Standard Huffman table G**

| | |
|---|---|
| HTOOB | **0** |

| VAL | PREFLEN | RANGELEN | Encoding |
|---|---|---|---|
| –1024 . . . –513 | 4 | 9 | **1000** + (VAL + 1024) encoded as 9 bits |
| –512 . . . –257 | 3 | 8 | **000** + (VAL + 512) encoded as 8 bits |
| –256 . . . –129 | 4 | 7 | **1001** + (VAL + 256) encoded as 7 bits |
| –128 . . . –65 | 5 | 6 | **11010** + (VAL + 128) encoded as 6 bits |
| –64 . . . –32 | 5 | 5 | **11011** + (VAL + 64) encoded as 5 bits |
| –32 . . . –1 | 4 | 5 | **1010** + (VAL + 32) encoded as 5 bits |
| 0 . . . 31 | 4 | 5 | **1011** + VAL encoded as 5 bits |
| 32 . . . 63 | 5 | 5 | **11100** + (VAL – 32) encoded as 5 bits |
| 64 . . . 127 | 5 | 6 | **11101** + (VAL – 64) encoded as 6 bits |
| 128 . . . 255 | 4 | 7 | **1100** + (VAL – 128) encoded as 7 bits |
| 256 . . . 511 | 3 | 8 | **001** + (VAL – 256) encoded as 8 bits |
| 512 . . . 1023 | 3 | 9 | **010** + (VAL – 512) encoded as 9 bits |
| 1024 . . . 2047 | 3 | 10 | **011** + (VAL – 1024) encoded as 10 bits |
| –∞ . . . –1025 | 5 | 32 | **11110** + (–1025 – VAL) encoded as 32 bits |
| 2048 . . . ∞ | 5 | 32 | **11111** + (VAL – 2048) encoded as 32 bits |

**Table B.8 – Standard Huffman table H**

| | |
|---|---|
| HTOOB | **1** |

| VAL | PREFLEN | RANGELEN | Encoding |
|---|---|---|---|
| –15 . . . –8 | 8 | 3 | **11111100** + (VAL + 15) encoded as 3 bits |
| –7 . . . –6 | 9 | 1 | **111111100** + (VAL + 7) encoded as 1 bit |
| –5 . . . –4 | 8 | 1 | **11111101** + (VAL + 5) encoded as 1 bit |
| –3 | 9 | 0 | **111111101** |
| –2 | 7 | 0 | **1111100** |
| –1 | 4 | 0 | **1010** |
| 0 . . . 1 | 2 | 1 | **00** + VAL encoded as 1 bit |
| 2 | 5 | 0 | **11010** |
| 3 | 6 | 0 | **111010** |
| 4 . . . 19 | 3 | 4 | **100** + (VAL – 4) encoded as 4 bits |
| 20 . . . 21 | 6 | 1 | **111011** + (VAL – 20) encoded as 1 bit |
| 22 . . . 37 | 4 | 4 | **1011** + (VAL – 22) encoded as 4 bits |
| 38 . . . 69 | 4 | 5 | **1100** + (VAL – 38) encoded as 5 bits |
| 70 . . . 133 | 5 | 6 | **11011** + (VAL – 70) encoded as 6 bits |
| 134 . . . 261 | 5 | 7 | **11100** + (VAL – 134) encoded as 7 bits |
| 262 . . . 389 | 6 | 7 | **111100** + (VAL – 262) encoded as 7 bits |
| 390 . . . 645 | 7 | 8 | **1111101** + (VAL – 390) encoded as 8 bits |
| 646 . . . 1669 | 6 | 10 | **111101** + (VAL – 646) encoded as 10 bits |
| –∞ . . . –16 | 9 | 32 | **111111110** + (–16 – VAL) encoded as 32 bits |
| 1670 . . . ∞ | 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 . . . –16 | 8 | 4 | **11111100** + (VAL + 31) encoded as 4 bits |
| –15 . . . –12 | 9 | 2 | **111111100** + (VAL + 15) encoded as 2 bits |
| –11 . . . –8 | 8 | 2 | **11111101** + (VAL + 11) encoded as 2 bits |
| –7 . . . –6 | 9 | 1 | **111111101** + (VAL + 7) encoded as 1 bit |
| –5 . . . –4 | 7 | 1 | **1111100** + (VAL + 5) encoded as 1 bit |
| –3 . . . –2 | 4 | 1 | **1010** + (VAL + 3) encoded as 1 bit |
| –1 . . . 0 | 3 | 1 | **010** + (VAL + 1) encoded as 1 bit |
| 1 . . . 2 | 3 | 1 | **011** + (VAL – 1) encoded as 1 bit |
| 3 . . . 4 | 5 | 1 | **11010** + (VAL – 3) encoded as 1 bit |
| 5 . . . 6 | 6 | 1 | **111010** + (VAL – 5) encoded as 1 bit |
| 7 . . . 38 | 3 | 5 | **100** + (VAL – 7) encoded as 5 bits |
| 39 . . . 42 | 6 | 2 | **111011** + (VAL – 39) encoded as 2 bits |
| 43 . . . 74 | 4 | 5 | **1011** + (VAL – 43) encoded as 5 bits |
| 75 . . . 138 | 4 | 6 | **1100** + (VAL – 75) encoded as 6 bits |
| 139 . . . 266 | 5 | 7 | **11011** + (VAL – 139) encoded as 7 bits |
| 267 . . . 522 | 5 | 8 | **11100** + (VAL – 267) encoded as 8 bits |
| 523 . . . 778 | 6 | 8 | **111100** + (VAL – 523) encoded as 8 bits |
| 779 . . . 1290 | 7 | 9 | **1111101** + (VAL – 779) encoded as 9 bits |
| 1291 . . . 3338 | 6 | 11 | **111101** + (VAL – 1291) encoded as 11 bits |
| –∞ . . . –32 | 9 | 32 | **111111110** + (–32 – VAL) encoded as 32 bits |
| 3339 . . . ∞ | 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 . . . –6 | 7 | 4 | **1111010** + (VAL + 21) encoded as 4 bits |
| –5 | 8 | 0 | **11111100** |
| –4 | 7 | 0 | **1111011** |
| –3 | 5 | 0 | **11000** |
| –2 . . . 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 . . . 69 | 2 | 6 | **01** + (VAL – 6) encoded as 6 bits |
| 70 . . . 101 | 5 | 5 | **11010** + (VAL – 70) encoded as 5 bits |
| 102 . . . 133 | 6 | 5 | **110111** + (VAL – 102) encoded as 5 bits |
| 134 . . . 197 | 6 | 6 | **111000** + (VAL – 134) encoded as 6 bits |
| 198 . . . 325 | 6 | 7 | **111001** + (VAL – 198) encoded as 7 bits |
| 326 . . . 581 | 6 | 8 | **111010** + (VAL – 326) encoded as 8 bits |
| 582 . . . 1093 | 6 | 9 | **111011** + (VAL – 582) encoded as 9 bits |
| 1094 . . . 2117 | 6 | 10 | **111100** + (VAL – 1094) encoded as 10 bits |
| 2118 . . . 4165 | 7 | 11 | **1111101** + (VAL – 2118) encoded as 11 bits |
| –∞ . . . –22 | 8 | 32 | **11111110** + (–22 – VAL) encoded as 32 bits |
| 4166 . . . ∞ | 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 . . . 3 | 2 | 1 | **10** + (VAL – 2) encoded as 1 bit |
| 4 | 4 | 0 | **1100** |
| 5 . . . 6 | 4 | 1 | **1101** + (VAL – 5) encoded as 1 bit |
| 7 . . . 8 | 5 | 1 | **11100** + (VAL – 7) encoded as 1 bit |
| 9 . . . 12 | 5 | 2 | **11101** + (VAL – 9) encoded as 2 bits |
| 13 . . . 16 | 6 | 2 | **111100** + (VAL – 13) encoded as 2 bits |
| 17 . . . 20 | 7 | 2 | **1111010** + (VAL – 17) encoded as 2 bits |
| 21 . . . 28 | 7 | 3 | **1111011** + (VAL – 21) encoded as 3 bits |
| 29 . . . 44 | 7 | 4 | **1111100** + (VAL – 29) encoded as 4 bits |
| 45 . . . 76 | 7 | 5 | **1111101** + (VAL – 45) encoded as 5 bits |
| 77 . . . 140 | 7 | 6 | **1111110** + (VAL – 77) encoded as 6 bits |
| 141 . . . ∞ | 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 . . . 4 | 3 | 1 | **110** + (VAL – 3) encoded as 1 bit |
| 5 | 5 | 0 | **11100** |
| 6 . . . 7 | 5 | 1 | **11101** + (VAL – 6) encoded as 1 bit |
| 8 . . . 9 | 6 | 1 | **111100** + (VAL – 8) encoded as 1 bit |
| 10 | 7 | 0 | **1111010** |
| 11 . . . 12 | 7 | 1 | **1111011** + (VAL – 11) encoded as 1 bit |
| 13 . . . 16 | 7 | 2 | **1111100** + (VAL – 13) encoded as 2 bits |
| 17 . . . 24 | 7 | 3 | **1111101** + (VAL – 17) encoded as 3 bits |
| 25 . . . 40 | 7 | 4 | **1111110** + (VAL – 25) encoded as 4 bits |
| 41 . . . 72 | 8 | 5 | **11111110** + (VAL – 41) encoded as 5 bits |
| 73 . . . ∞ | 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 . . . 6 | 4 | 1 | **1101** + (VAL – 5) encoded as 1 bit |
| 7 . . . 14 | 3 | 3 | **101** + (VAL – 7) encoded as 3 bits |
| 15 . . . 16 | 6 | 1 | **111010** + (VAL – 15) encoded as 1 bit |
| 17 . . . 20 | 6 | 2 | **111011** + (VAL – 17) encoded as 2 bits |
| 21 . . . 28 | 6 | 3 | **111100** + (VAL – 21) encoded as 3 bits |
| 29 . . . 44 | 6 | 4 | **111101** + (VAL – 29) encoded as 4 bits |
| 45 . . . 76 | 6 | 5 | **111110** + (VAL – 45) encoded as 5 bits |
| 77 . . . 140 | 7 | 6 | **1111110** + (VAL – 77) encoded as 6 bits |
| 141 . . . ∞ | 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 . . . –9 | 7 | 4 | **1111100** + (VAL + 24) encoded as 4 bits |
| –8 . . . –5 | 6 | 2 | **111100** + (VAL + 8) encoded as 2 bits |
| –4 . . . –3 | 5 | 1 | **11100** + (VAL + 4) encoded as 1 bit |
| –2 | 4 | 0 | **1100** |
| –1 | 3 | 0 | **100** |
| 0 | 1 | 0 | **0** |
| 1 | 3 | 0 | **101** |
| 2 | 4 | 0 | **1101** |
| 3 . . . 4 | 5 | 1 | **11101** + (VAL – 3) encoded as 1 bit |
| 5 . . . 8 | 6 | 2 | **111101** + (VAL – 5) encoded as 2 bits |
| 9 . . . 24 | 7 | 4 | **1111101** + (VAL – 9) encoded as 4 bits |
| –∞ . . . –25 | 7 | 32 | **1111110** + (–25 – VAL) encoded as 32 bits |
| 25 . . . ∞ | 7 | 32 | **1111111** + (VAL – 25) encoded as 32 bits |

# Annex C

# Gray-scale Image Decoding Procedure

(This annex forms an integral part of this Recommendation | International Standard)

## 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 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.[b)] |
| **GSKIP** | Bitmap | | | A mask indicating which values should be skipped. **GSW** pixels wide, **GSH** pixels high.[a)] |
| a) Unused if **GSUSESKIP = 0.**<br>b) Unused if **GSMMR = 1.** | | | | |

## C.3 Return value

The variable whose value is the result of this decoding procedure is shown in Table C.2.

**Table C.2 – Return value from the gray-scale image decoding procedure**

| Name | Type | Size (bits) | Signed? | Description and restrictions |
|---|---|---|---|---|
| GSVALS | Array | | | The decoded gray-scale image. 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.5 Decoding the gray-scale image

The gray-scale image is obtained by decoding **GSBPP** 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[**GSBPP** – 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** |
| **TPGDON** | **0** |
| **USESKIP** | **GSUSESKIP** |
| **SKIP** | **GSKIP** |
| $\mathbf{GBATX_1}$ | 3 if **GSTEMPLATE** ≤ 1; 2 if **GSTEMPLATE** ≥ 2. |
| $\mathbf{GBATY_1}$ | –1 |
| $\mathbf{GBATX_2}$ | –3 |
| $\mathbf{GBATY_2}$ | –1 |
| $\mathbf{GBATX_3}$ | 2 |
| $\mathbf{GBATY_3}$ | –2 |
| $\mathbf{GBATX_4}$ | –2 |
| $\mathbf{GBATY_4}$ | –2 |

2) Set $J$ = **GSBPP** – 2.

3) While $J \geq 0$, perform the following steps:

   a) Decode GSPLANES[$J$] using the generic region decoding procedure. The parameters to the generic region decoding procedure are as shown in Table C.4.

   b) For each pixel $(x, y)$ in GSPLANES[$J$], set:

$$\mathrm{GSPLANES}[J][x, y] = \mathrm{GSPLANES}[J + 1][x, y] \text{ XOR } \mathrm{GSPLANES}[J][x, y]$$

   c) Set $J = J - 1$.

4) For each $(x, y)$, set:

$$\mathrm{GSVALS}[x, y] = \sum_{J=0}^{\mathbf{GSBPP}-1} \mathrm{GSPLANES}[J][x, y] \times 2^J$$

# Annex D

# File Formats

(This annex forms an integral part of this Recommendation | International Standard)

There are two standalone file organisations possible for a JBIG2 bitstream. There is also a third organisation, not intended for standalone usage, but instead to allow JBIG2-encoded data to be embedded in another file format.

NOTE – It is recommended that "`.jbig2`" is used as the extension for JBIG2 files. In environments where only three characters are allowed, "`.jb2`" is recommended. It is also recommended that JBIG2 decoders recognise both extensions.

## D.1 Sequential organisation

This is a standalone file organisation. This organisation is intended for streaming applications, where the decoder is guaranteed to begin at the start of the bitstream and decode everything up to the end of the bitstream.

In this organisation, the file structure looks like Figure D.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.

The segments must appear in increasing order of their segment numbers: no segment may precede a segment having a lower number than it.

| |
|---|
| File header |
| Segment 1 segment header |
| Segment 1 data |
| Segment 2 segment header |
| Segment 2 data |
| . . . |
| Segment N segment header |
| Segment N data |

**Figure D.1 – Sequential organisation**

## D.2 Random-access organisation

This is a standalone file organisation. This organisation is intended for random-access applications, where the decoder might want to process parts of the file in an arbitrary order, such as decoding all the odd-numbered pages before any even-numbered page, or decode pages individually in response to some user input. The ability to perform random access is therefore important.

In this organisation, the file structure looks like Figure D.2. A file header is followed 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. The last segment must be an end of file segment; otherwise, it is impossible for the decoder to determine when it has read the last segment header.

The segments must appear in increasing order of their segment numbers: no segment may precede a segment having a lower number than it.

| |
|---|
| File header |
| Segment 1 segment header |
| Segment 2 segment header |
| . . . |
| Segment N segment header |
| Segment 1 data |
| Segment 2 data |
| . . . |
| Segment N data |

**Figure D.2 – Random-access organisation**

## D.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.

## D.4 File header syntax

A file header contains the following fields, in order:

**ID string** – see D.4.1.

**File header flags** – see D.4.2.

**Number of pages** – see D.4.3.

### D.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 non-printable, 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.

### D.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.

NOTE – Note that there is no way to indicate the embedded organisation, as that organisation does not include a JBIG2 file header.

**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**.

### D.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.

# Annex E

# Arithmetic Coding

(This annex forms an integral part of this Recommendation | International Standard)

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 Annex 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 shall duplicate. In H.2 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 Recommendation | International Standard, CX is a label for a context, formed by some character string followed by a string of bits.

EXAMPLE – Two possible values of CX are "IADW**001010100**" and "GB**1110110010000000**".

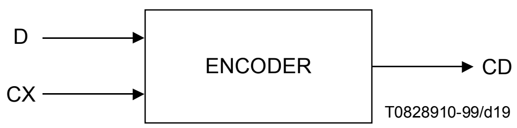



**Figure E.1 – Arithmetic encoder inputs and outputs**

### 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 string 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 string. This coding convention requires that symbols be recognised 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 string 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 string. Therefore, the code string 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 $0.75 \le A < 1.5$ 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 compressed data buffer. Carry-over into the external buffer is resolved by a bit-stuffing procedure.

Keeping A in the range $0.75 \le A < 1.5$ allows a simple arithmetic approximation to be used in the interval subdivision. 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 renormalisation of both A and C.

With the process illustrated 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 gives 1/3 of the interval to the MPS and 2/3 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 renormalisation is needed.

Whenever a renormalisation 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 renormalisation 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) initialises 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 embedded 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 terminates the encoding operations and generates the required terminating marker.

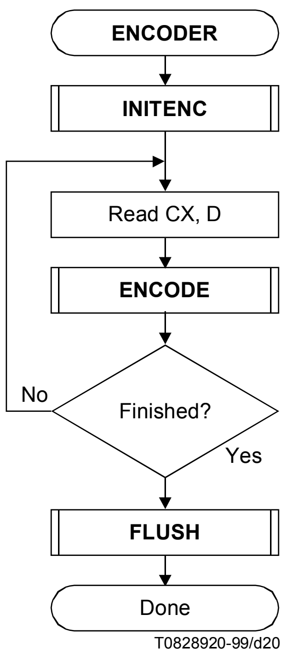



**Figure E.2 – Encoder for the MQ-coder**

### E.2.1 Encoder code register conventions

The flow charts given in this annex 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 Encoding a decision (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 Encoding a 1 or 0 (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 illustrated in Figures E.4 and E.5. In these figures, CX is the context. For each context, the index of the probability estimate which is to be used in the coding operations and the MPS value are stored. MPS(CX) is the sense (**0** or **1**) of the MPS for context CX.

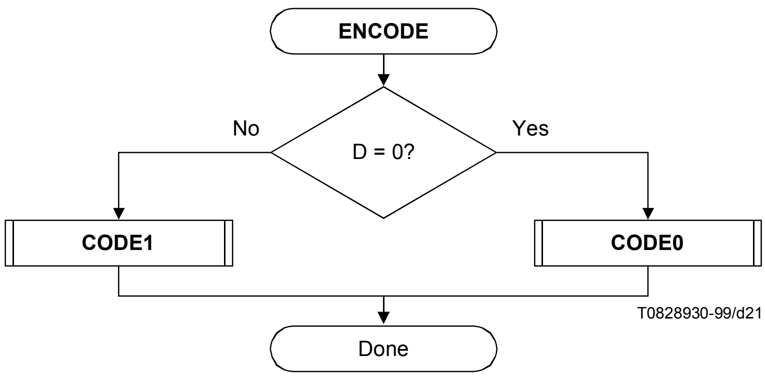



**Figure E.3 – ENCODE procedure**

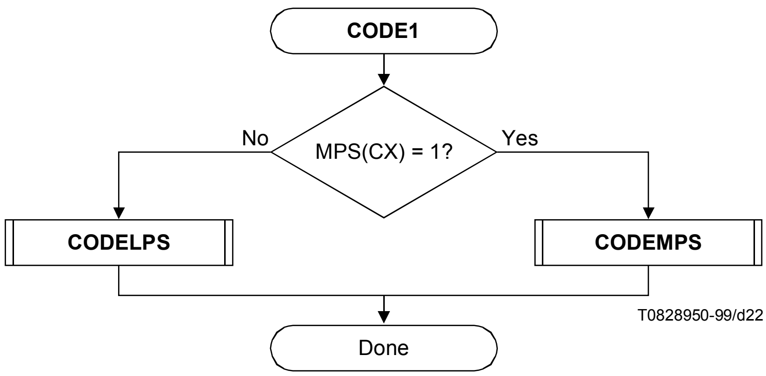



**Figure E.4 – CODE1 procedure**

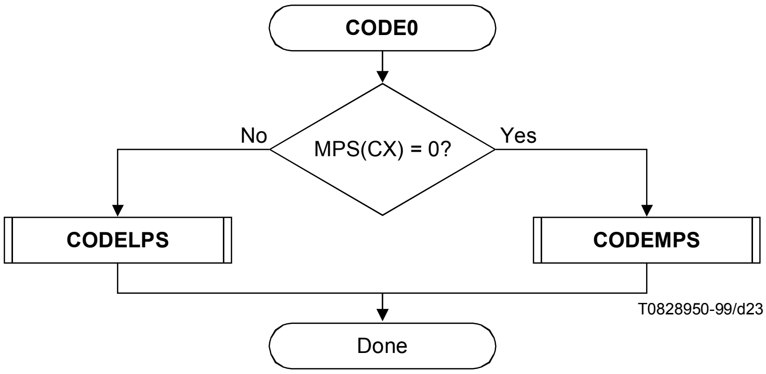



**Figure E.5 – CODE0 procedure**

### E.2.4 Encoding an MPS or LPS (CODEMPS and CODELPS)

The CODELPS (Figure E.6) procedure usually consists of a scaling of the interval to Qe(I(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 renormalisation (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.

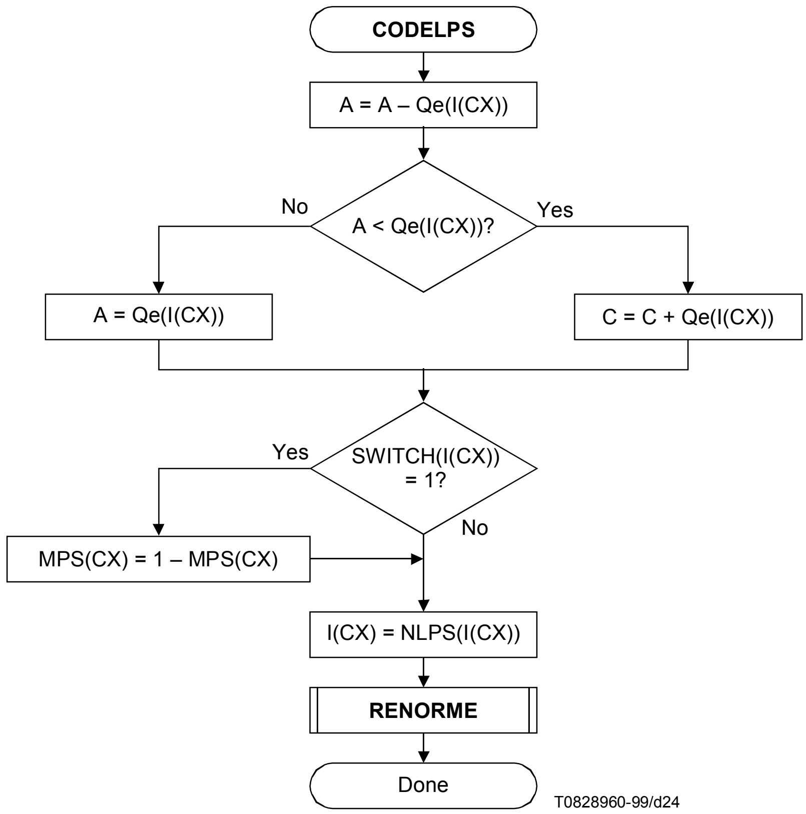



**Figure E.6 – CODELPS procedure with conditional MPS/LPS exchange**

The CODEMPS (Figure E.7) procedure usually reduces the size of the interval to the MPS sub-interval 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 renormalisation (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.

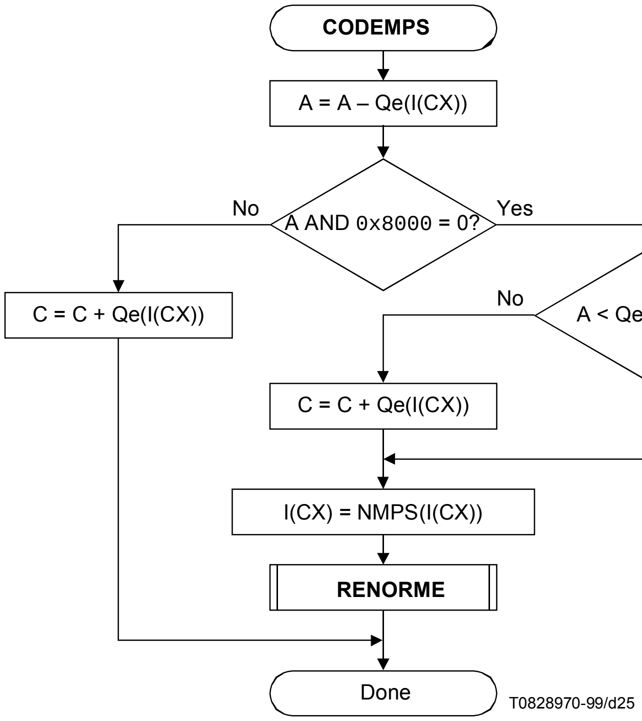



**Figure E.7 – CODEMPS procedure with conditional MPS/LPS exchange**

### E.2.5 Probability estimation

Table E.1 shows the Qe value associated with each Qe index. The Qe values are expressed as 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) × (`0x8000`).

The estimator can be defined as a finite-state machine – a table of Qe indexes and associated next states for each type of renormalisation (i.e. new table positions) – as shown in Table E.1. The change in state occurs only when the arithmetic coder interval register is renormalised. This is always 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 renormalisation, NLPS gives the new index for the LPS probability estimate; also, if Switch is **1**, the MPS symbol sense is reversed. After an MPS renormalisation, NMPS gives the new index for the LPS probability estimate.

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 renormalisation. This index is saved in the context storage at CX. MPS(CX) does not change.

The procedure for estimating the probability on the LPS renormalisation path is similar to that of an MPS renormalisation, except that when Switch(I(CX)) is 1, the sense of MPS(CX) is inverted.

The final index state 46 can be used to establish a fixed 0.5 probability estimate.

### E.2.6 Renormalisation in the encoder (RENORME)

Renormalisation 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 renormalisation is illustrated 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. Renormalisation continues until A is no longer less than `0x8000`.

**Table E.1 – Qe values and probability estimation process**

| Index | Qe_Value | | | NMPS | NLPS | SWITCH |
|---|---|---|---|---|---|---|
| | (hexadecimal) | (binary) | (decimal) | | | |
| 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 | 0x0AC1 | **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 | 0x0AC1 | **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 | 0x02A1 | **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 |

### E.2.7 Compressed data output (BYTEOUT)

The BYTEOUT routine called from RENORME is illustrated in Figure E.9. This routine contains the bit-stuffing procedures which are needed to limit carry propagation into the completed bytes of compressed data. The conventions used make it impossible for a carry to propagate through more than the byte most recently written to the compressed data 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 compressed data buffer pointer BP. If B is not a 0xFF byte, 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 needs to 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.

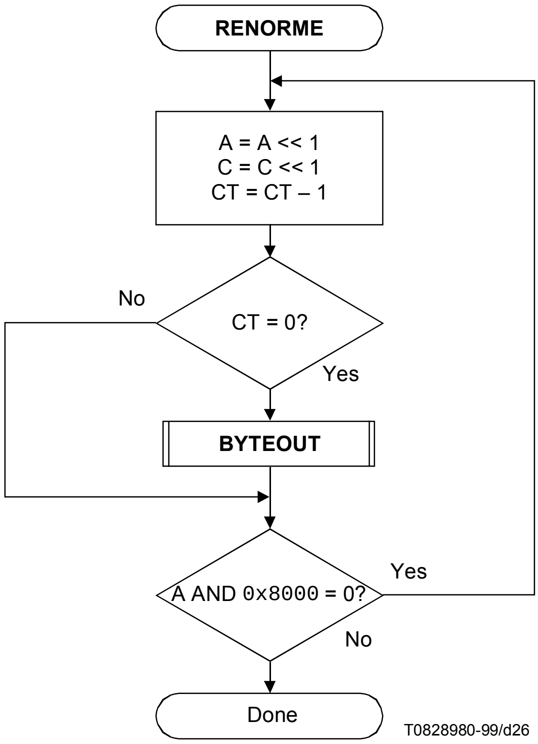



**Figure E.8 – Encoder renormalisation procedure**

### E.2.8 Initialisation of the encoder (INITENC)

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 need to 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 byte, a spurious bit stuff will occur, but can be compensated for by increasing CT. Note that the default initialisation of the statistics bins is MPS = 0 and I = 0 (i.e. Qe = 0x5601 or decimal 0.503937).

### E.2.9 Termination of encoding (FLUSH)

The FLUSH procedure shown in Figure E.11 is used to terminate the encoding operations and generate the required terminating marker. The procedure guarantees that the 0xFF prefix to the marker code overlaps the final bits of the compressed data. This 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 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 is 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 compressed data which is guaranteed to be 0xFF.

### E.2.10 Minimisation of the compressed data

If desired, the compressed data 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 compressed data, provided that the earliest 0xFF byte in the sequence is not removed. This remaining 0xFF byte then becomes the prefix to the marker code which terminates the compressed data.

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.

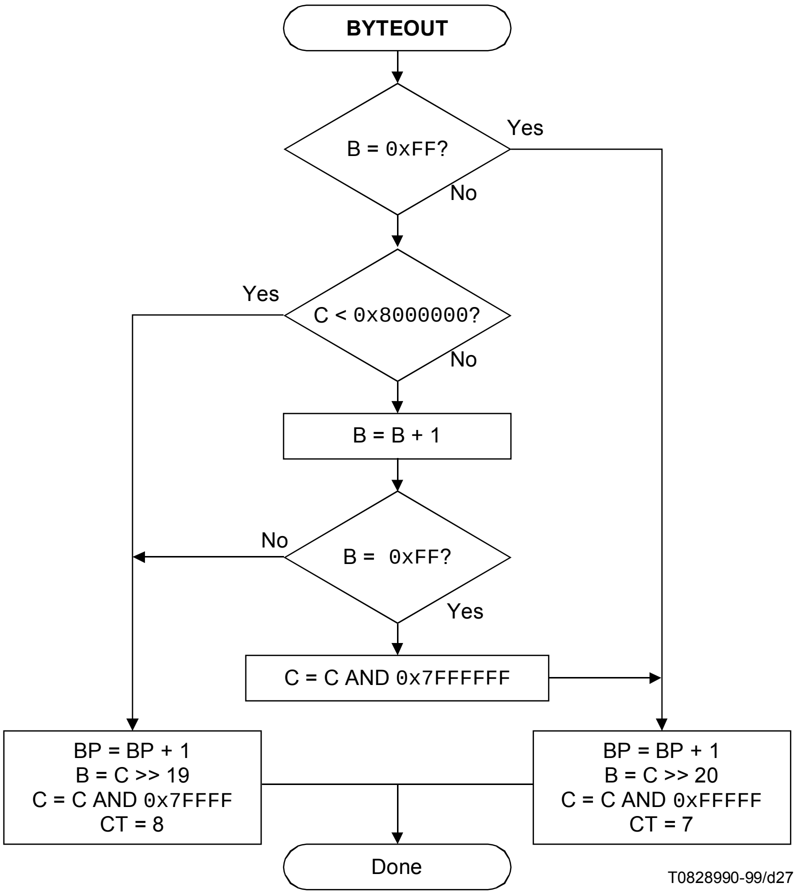



**Figure E.9 – BYTEOUT procedure for encoder**

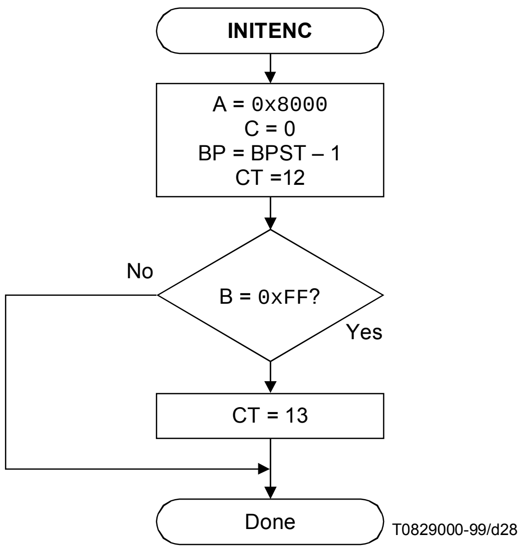



**Figure E.10 – Initialisation of the encoder**

## E.3 Arithmetic decoding procedure

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 need to supply exactly the same context CX for each given decision.

The DECODER (Figure E.14) initialises 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 embedded in DECODE. When all contexts have been read (Finished?), the compressed data has been decompressed.

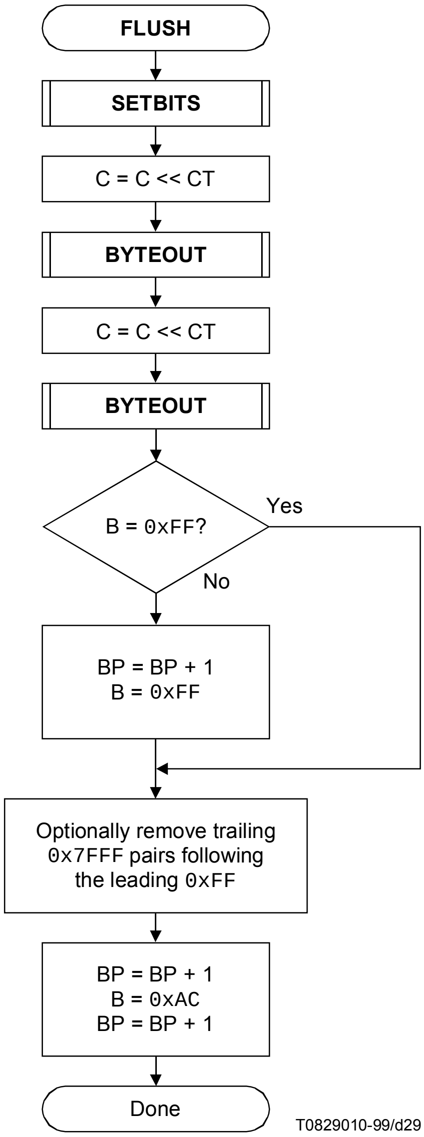



**Figure E.11 – FLUSH procedure**

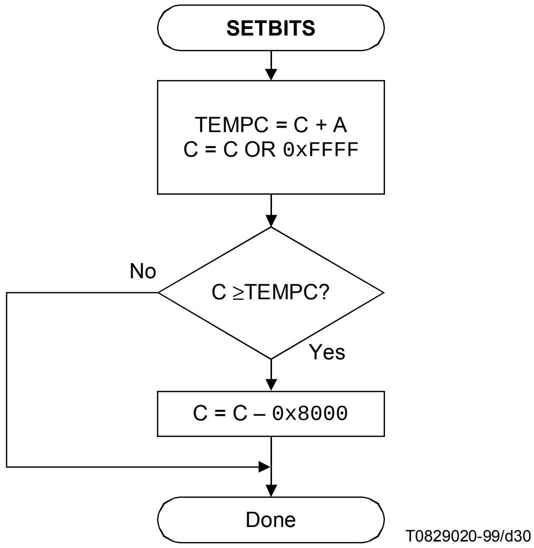



**Figure E.12 – Setting the final bits in the C register**

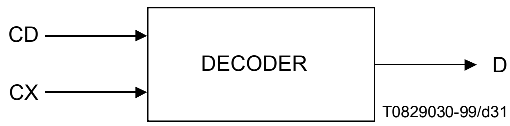



**Figure E.13 – Arithmetic decoder inputs and outputs**

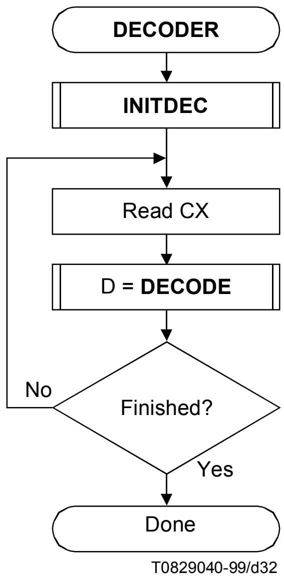



**Figure E.14 – Decoder for the MQ-coder**

### E.3.1 Decoder code register conventions

The flow charts given in this annex assume the following register structures for the decoder:

| | 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 renormalisation 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 subclause.

Note that the comparisons shown in the various procedures in this subclause assume precisions greater than 16 bits. Logical comparisons can be used with 16-bit precision.

### E.3.2 Decoding a decision (DECODE)

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 illustrated 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 renormalisation 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 Qe(I(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 E.1. Note that the probability estimation in the decoder needs to 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 Renormalisation in the decoder (RENORMD)

The RENORMD procedure for the decoder renormalisation is illustrated 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 Compressed data input (BYTEIN)

The BYTEIN procedure called from RENORMD is illustrated in Figure E.19. This procedure reads in one byte of data, compensating for any stuff bits following the `0xFF` byte in the process. It also detects the marker codes which must occur at the end of a scan or resynchronisation interval. The C-register in this procedure is the concatenation of the Chigh and Clow registers.

B is the byte pointed to by the compressed data buffer pointer BP. If B is not a `0xFF` byte, BP is incremented and the new value of B is inserted into the high order 8 bits of Clow.

If B is a `0xFF` byte, then B1 (the byte pointed to by BP+1) is tested. If B1 exceeds 0x8F, then 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 arithmetically compressed data. 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.

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.

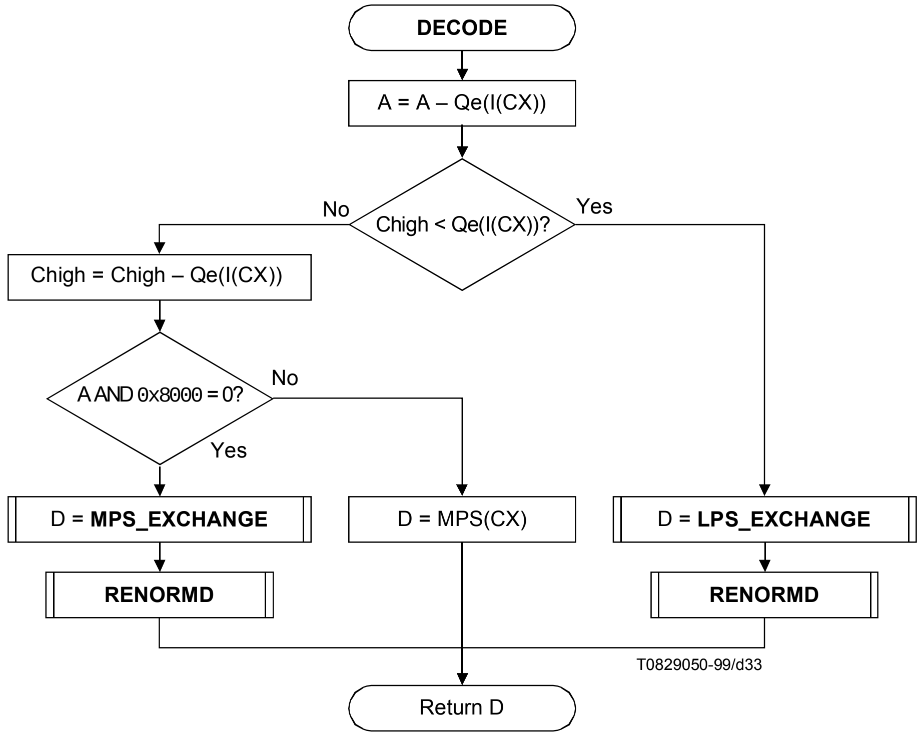



**Figure E.15 – Decoding an MPS or an LPS**

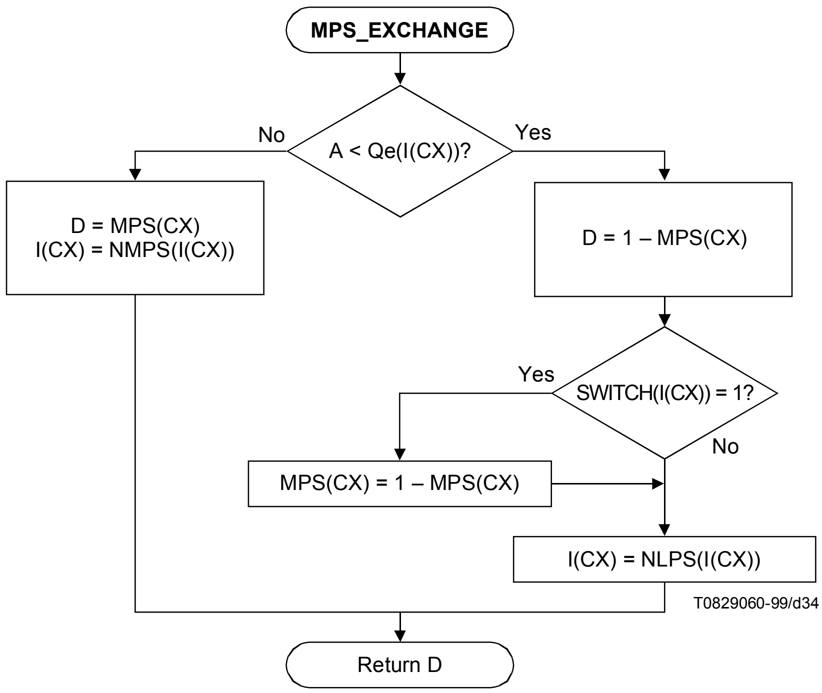



**Figure E.16 – Decoder MPS path conditional exchange procedure**

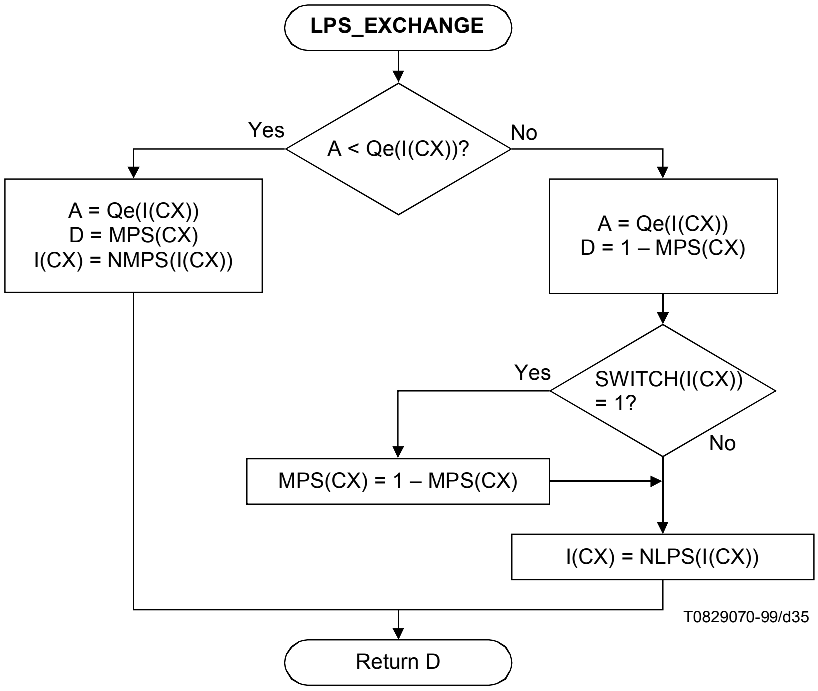



**Figure E.17 – Decoder LPS path conditional exchange procedure**

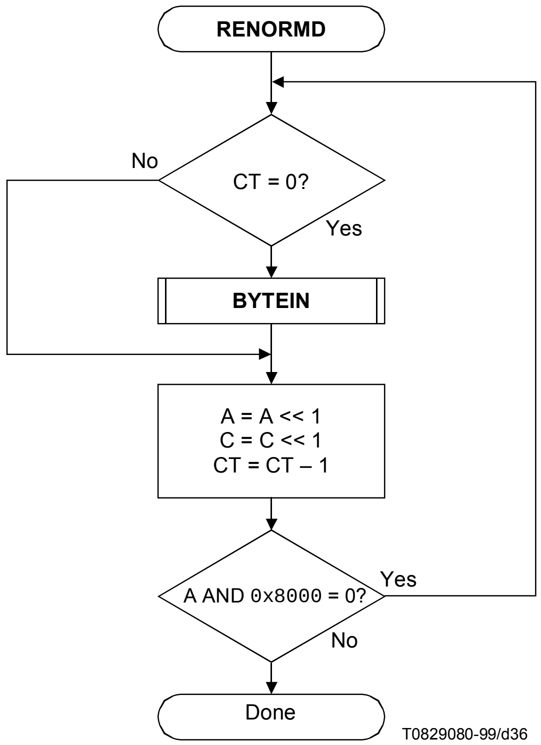



**Figure E.18 – Decoder renormalisation procedure**

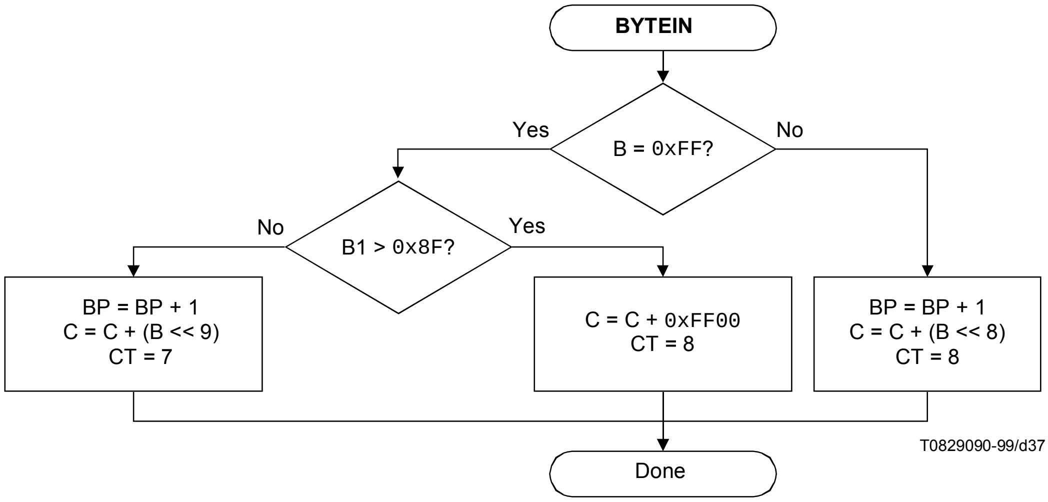



**Figure E.19 – BYTEIN procedure for decoder**

### E.3.5 Initialisation of the decoder (INITDEC)

The INITDEC procedure is used to start the arithmetic decoder. The basic steps are shown in Figure E.20.

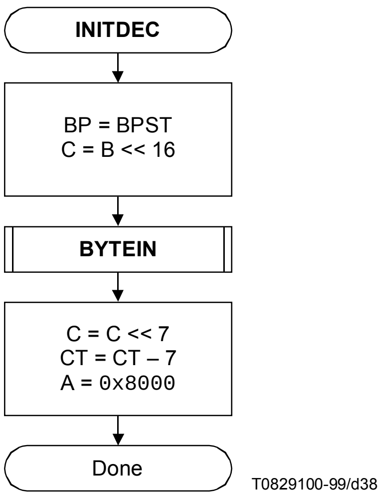



**Figure E.20 – Initialisation of the decoder**

BP, the pointer to the compressed data, is initialised 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

Usually, when the end of the arithmetically compressed data is reached, the compressed data buffer pointer BP points to the 0xFF byte of the terminating marker code. If for any reason the compressed data buffer pointer is not at the 0xFF byte of the marker, a resynchronisation procedure needs to scan the compressed data 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 standardised.

### E.3.7 Resetting arithmetic coding statistics

At certain points during the decoding, some or all of the arithmetic coding 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 text region segment, all the arithmetic coding statistics are reset.

### E.3.8 Saving arithmetic coding statistics

In some cases, the decoder needs to save or restore some values of I(CX) and MPS(CX). This is done as part of decoding a symbol dictionary segment. In this case, the values that are saved and/or restored are all the values indexed by CX values whose initial label is "GB" or "GR" (i.e. all those CX values used by the generic region decoding procedure or the generic refinement region decoding procedure).

# Annex F

# Profiles

(This annex forms an integral part of this Recommendation | International Standard)

It is recommended that a JBIG2 decoder implement one of the profiles described in Tables F.1 through F.7. Note that profile `0x00000001` (Table F.1) includes all the capabilities of this entire Recommendation | International Standard, and is the profile assumed when none is explicitly specified.

See 7.4.12 for information on how the profile identification numbers are used.

Profile identification numbers `0x00000000` through `0x00FFFFFF` are reserved for ISO/IEC and ITU-T applications. Of this range, profile identification numbers `0x00000100` through `0x00000FFF` are reserved for ITU-T facsimile applications. Entities other than ISO/IEC and ITU-T wishing to use an unassigned profile identification number should choose one in the range `0x01000000` through `0xFFFFFFFF` that is not likely to conflict with any other entity's choice. It is recommended that the first three bytes of the profile identification number be chosen to match the first three letters of the name of the entity, or be a suitable abbreviation of that name.

**Table F.1 – Profile description for profile `0x00000001`**

| **Profile identification** | `0x00000001` |
|---|---|
| Requirements | All JBIG2 capabilities |
| Generic region coding | No restriction |
| Refinement region coding | No restriction |
| Halftone region coding | No restriction |
| Numerical data | No restriction |
| Resources required | High-speed processor |
| Application examples | General-purpose printing; Format conversion |

**Table F.2 – Profile description for profile `0x00000002`**

| **Profile identification** | `0x00000002` |
|---|---|
| Requirements | Maximum compression |
| Generic region coding | Arithmetic only; any template used |
| Refinement region coding | No restriction |
| Halftone region coding | No restriction |
| Numerical data | Arithmetic only |
| Resources required | High-speed processor |
| Application examples | Archiving; Wireless WWW |

**Table F.3 – Profile description for profile `0x00000003`**

NOTE 1 – This profile is a subset of profile `0x00000002`.

| **Profile identification** | `0x00000003` |
|---|---|
| Requirements | Medium complexity and medium compression |
| Generic region coding | Arithmetic only; only 10-pixel and 13-pixel templates |
| Refinement region coding | 10-pixel template only (arithmetic) |
| Halftone region coding | No skip mask used |
| Numerical data | Arithmetic only |
| Resources required | Medium-speed processor |
| Application examples | WWW; High-end fax |

**Table F.4 – Profile description for profile `0x00000004`**

| **Profile identification** | `0x00000004` |
|---|---|
| Requirements | Low complexity with progressive lossless capability |
| Generic region coding | MMR only |
| Refinement region coding | 10-pixel template only (arithmetic) |
| Halftone region coding | No skip mask used |
| Numerical data | Huffman only |
| Resources required | Medium-speed processor |
| Application examples | WWW |

**Table F.5 – Profile description for profile `0x00000005`**

NOTE 2 – This profile is a subset of profile `0x00000004.`

| **Profile identification** | `0x00000005` |
|---|---|
| Requirements | Low complexity |
| Generic region coding | MMR only |
| Refinement region coding | Not available |
| Halftone region coding | No skip mask used |
| Numerical data | Huffman only |
| Resources required | Low-speed processor |
| Application examples | Low-end fax; High-speed printing; Embedded processors |

**Table F.6 – Profile description for profile `0x00000006`**

NOTE 3 – This profile is a subset of profile `0x00000003`.

| **Profile identification** | `0x00000006` |
|---|---|
| Requirements | Low memory with more compression |
| Generic region coding | Arithmetic only; only 10-pixel and 13-pixel templates |
| Refinement region coding | 10-pixel template only (arithmetic) |
| Halftone region coding | No skip mask used |
| Numerical data | Arithmetic only |
| Resources required | Medium-speed processor with little memory |
| Application examples | Multi-function devices |
| Additional constraints | • Every page must have at least two stripes<br>• A stripe that contains a text region segment may not contain any halftone or generic region segments.<br>• All region segments must be immediate region segments (no auxiliary buffers).<br>• In any symbol dictionary segment with **SDREFAGG** equal to **1**, REFAGGNINST must be equal to 1 for all symbols. |

**Table F.7 – Profile description for profile `0x00000007`**

NOTE 4 – This profile is a subset of profile `0x00000005`.

| **Profile identification** | `0x00000007` |
|---|---|
| Requirements | Low memory with more speed |
| Generic region coding | MMR only |
| Refinement region coding | Not available |
| Halftone region coding | No skip mask used |
| Numerical data | Huffman only |
| Resources required | Low-speed processor with little memory |
| Application examples | Multi-function devices |
| Additional constraints | • Every page must have at least two stripes.<br>• A stripe that contains a text region segment may not contain any halftone or generic region segments.<br>• All region segments must be immediate region segments (no auxiliary buffers).<br>• In any symbol dictionary segment with **SDREFAGG** equal to **1**, REFAGGNINST must be equal to 1 for all symbols. |

# Annex G

## Arithmetic Decoding Procedure (Software Conventions)

(This annex does not form an integral part of this Recommendation | International Standard)

This annex provides some alternative flowcharts for a version of the adaptive entropy decoder. This alternative version may be more efficient when implemented in software, as it has fewer operations along the fast path.

The alternative version is obtained by making the following substitutions:

- Replace the flowchart in Figure E.20 with the flowchart in Figure G.1.
- Replace the flowchart in Figure E.15 with the flowchart in Figure G.2.
- Replace the flowchart in Figure E.19 with the flowchart in Figure G.3.

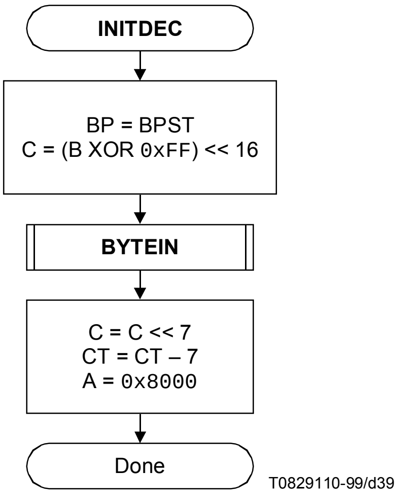



**Figure G.1 – Initialisation of the software conventions decoder**

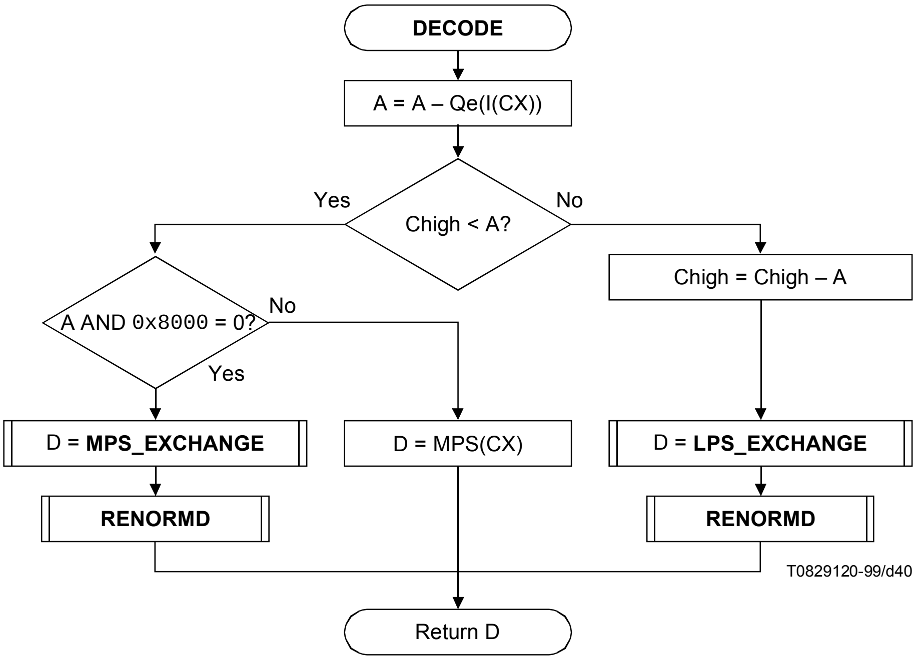



**Figure G.2 – Decoding an MPS or an LPS in the software-conventions decoder**

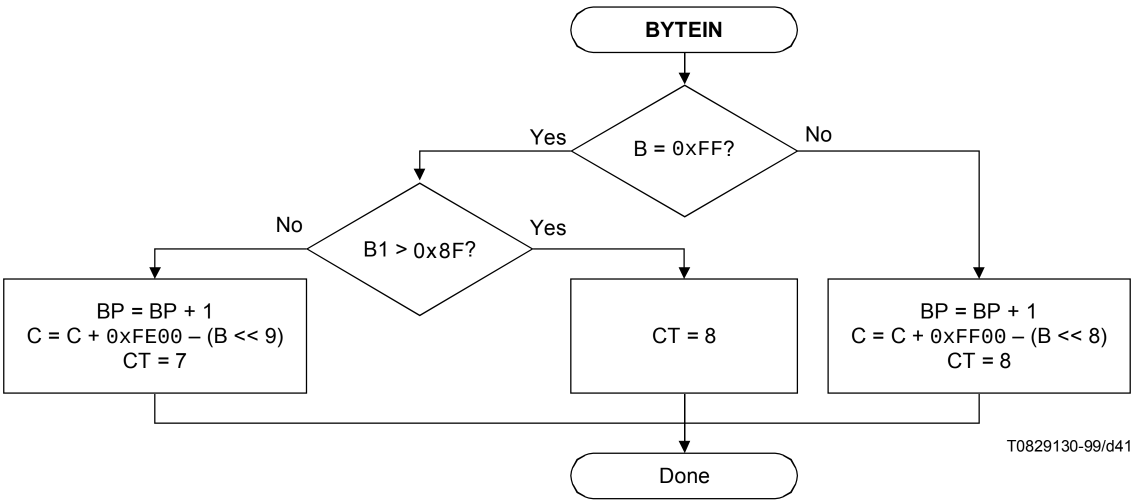



**Figure G.3 – Inserting a new byte into the C register in the software-conventions decoder**

# Annex H

## Datastream Example and Test Sequence

(This annex does not form an integral part of this Recommendation | International Standard)

### H.1 Datastream example

This subclause gives a small datastream that exercises a large number of the features of JBIG2.

The raw data here is shown in the following format:

```
0023: 01 23 45 67 89 AB CD EF
```

where the first field (before the colon) is the byte offset into the datastream of the data being displayed, and the remainder of the line is a sequence of bytes starting at that offset. All these values are in hexadecimal.

In general, the decoding of the first occurrence of any type of data is explained in detail; further occurrences of the same type of data are not explained in as much detail.

This datastream encodes:

- one symbol dictionary that is not associated with any page; and
- three pages,

i.e. the entities encoded in this datastream are three pages, plus one object that is not associated with any page.

The first two pages each contain a page information segment, a symbol dictionary segment, a text region segment, a generic region segment, a pattern dictionary segment, a halftone region segment, and an end of page segment. The bitmaps encoded by these two pages are identical, and are shown in Figure H.1. The data encoded by the corresponding segments in the two pages are also identical (e.g. the text region segment for page 1 contains the same data, in the same order, as the text region segment for page 2). However, the segments are encoded differently in the two pages: in page 1, all the segments use some form of Huffman or MMR coding; in page 2, all the segments use some form of arithmetic coding. Thus, implementors can cross-check against their own implementations to ensure that they are decoding correctly.

The third page contains two symbol dictionaries, one of which defines symbols by refinement and aggregation from the other one, and one text region, which uses the symbols from the dictionary including refining one of them.

Throughout this subclause, pixels having the value **1** are shown as black pixels, while pixels having the value **0** are shown as white pixels. This is a typical interpretation of **0** and **1**, as might be made by an application using this Recommendation | International Standard.

The datastream is:

```
0000: 97 4A 42 32 0D 0A 1A 0A 01 00 00 00 03 00 00 00
0010: 00 00 01 00 00 00 00 18 00 01 00 00 00 01 00 00
0020: 00 01 E9 CB F4 00 26 AF 04 BF F0 78 2F E0 00 40
0030: 00 00 00 01 30 00 01 00 00 00 13 00 00 00 40 00
0040: 00 00 38 00 00 00 00 00 00 00 00 01 00 00 00 00
0050: 00 02 00 01 01 00 00 00 1C 00 01 00 00 00 02 00
0060: 00 00 02 E5 CD F8 00 79 E0 84 10 81 F0 82 10 86
0070: 10 79 F0 00 80 00 00 00 03 07 42 00 02 01 00 00
0080: 00 31 00 00 00 25 00 00 00 08 00 00 00 04 00 00
0090: 00 01 00 0C 09 00 10 00 00 00 05 01 10 00 00 00
00A0: 00 00 00 00 00 00 00 00 00 00 00 00 0C 40 07 08
00B0: 70 41 D0 00 00 00 04 27 00 01 00 00 00 2C 00 00
00C0: 00 36 00 00 00 2C 00 00 00 04 00 00 00 0B 00 01
00D0: 26 A0 71 CE A7 FF FF FF FF FF FF FF FF FF FF FF
00E0: FF FF FF FF FF FF FF FF F8 F0 00 00 00 05 10 01
00F0: 01 00 00 00 2D 01 04 04 00 00 00 0F 20 D1 84 61
0100: 18 45 F2 F9 7C 8F 11 C3 9E 45 F2 F9 7D 42 85 0A
0110: AA 84 62 2F EE EC 44 62 22 35 2A 0A 83 B9 DC EE
0120: 77 80 00 00 00 06 17 20 05 01 00 00 00 57 00 00
0130: 00 20 00 00 00 24 00 00 00 10 00 00 00 0F 00 01
0140: 00 00 00 08 00 00 00 09 00 00 00 00 00 00 00 00
0150: 04 00 00 00 AA AA AA AA 80 08 00 80 36 D5 55 6B
0160: 5A D4 00 40 04 2E E9 52 D2 D2 D2 8A A5 4A 00 20
0170: 02 23 E0 95 24 B4 92 8A 4A 92 54 92 D2 4A 29 2A
0180: 49 40 04 00 40 00 00 00 07 31 00 01 00 00 00 00
```

```
0190: 00 00 00 08 30 00 02 00 00 00 13 00 00 00 40 00
01A0: 00 00 38 00 00 00 00 00 00 00 01 00 00 00 00
01B0: 00 09 00 01 02 00 00 00 1B 08 00 02 FF 00 00 00
01C0: 02 00 00 00 02 4F E7 8C 20 0E 1D C7 CF 01 11 C4
01D0: B2 6F FF AC 00 00 00 0A 07 40 00 09 02 00 00 00
01E0: 1F 00 00 00 25 00 00 00 08 00 00 00 04 00 00 00
01F0: 01 00 0C 08 00 00 00 05 8D 6E 5A 12 40 85 FF AC
0200: 00 00 00 0B 27 00 02 00 00 00 23 00 00 00 36 00
0210: 00 00 2C 00 00 00 04 00 00 00 0B 00 08 03 FF FD
0220: FF 02 FE FE FE 04 EE ED 87 FB CB 2B FF AC 00 00
0230: 00 0C 10 01 02 00 00 00 1C 06 04 04 00 00 00 0F
0240: 90 71 6B 6D 99 A7 AA 49 7D F2 E5 48 1F DC 68 BC
0250: 6E 40 BB FF AC 00 00 00 0D 17 20 0C 02 00 00 00
0260: 3E 00 00 00 20 00 00 00 24 00 00 00 10 00 00 00
0270: 0F 00 02 00 00 00 08 00 00 00 09 00 00 00 00 00
0280: 00 00 00 04 00 00 00 87 CB 82 1E 66 A4 14 EB 3C
0290: 4A 15 FA CC D6 F3 B1 6F 4C ED BF A7 BF FF AC 00
02A0: 00 00 0E 31 00 02 00 00 00 00 00 00 00 0F 30 00
02B0: 03 00 00 00 13 00 00 00 25 00 00 00 08 00 00 00
02C0: 00 00 00 00 00 01 00 00 00 00 10 00 01 00 00
02D0: 00 00 16 08 00 02 FF 00 00 00 01 00 00 00 01 4F
02E0: E7 8D 68 1B 14 2F 3F FF AC 00 00 00 11 00 21 10
02F0: 03 00 00 00 20 08 02 02 FF FF FF FF FF 00 00 00
0300: 03 00 00 00 02 4F E9 D7 D5 90 C3 B5 26 A7 FB 6D
0310: 14 98 3F FF AC 00 00 00 12 07 20 11 03 00 00 00
0320: 25 00 00 00 25 00 00 00 08 00 00 00 00 00 00 00
0330: 00 00 8C 12 00 00 00 04 A9 5C 8B F4 C3 7D 96 6A
0340: 28 E5 76 8F FF AC 00 00 00 13 31 00 03 00 00 00
0350: 00 00 00 00 14 33 00 00 00 00 00 00
```

The datastream is decoded as follows:

1) The file header:

   `0000: 97 4A 42 32 0D 0A 1A 0A 01 00 00 00 02`

   a) The eight-byte ID string:

      `0000: 97 4A 42 32 0D 0A 1A 0A`

   b) The one-byte file header flags field:

      `0008: 01`

      This field indicates that the file uses the sequential organisation, and that the number of pages is known.

   c) The four-byte number of pages field:

      `0009: 00 00 00 03`

      This field indicates that the file has three pages.

2) The first segment header:

   `000D: 00 00 00 00 00 01 00 00 00 00 18`

   a) The four-byte segment number field:

      `000D: 00 00 00 00`

      This field indicates that the segment is segment number 0.

   b) The one-byte segment header flags field:

      `0011: 00`

      This field indicates that the segment has type "Symbol dictionary" (type 0), has a short page association field, and does not have the deferred non-retain bit set.

   c) The one-byte referred-to segment count and retention flags field:

      `0012: 01`

      This field indicates that the segment refers to no other segment, and that it should be retained.

   d) The one-byte (short-form) segment page association field:

      `0013: 00`

      This field indicates that this segment is not associated with any page.

   e) The four-byte segment data length field:

      `0014: 00 00 00 18`

      This field indicates that the segment's data part is 24 bytes long.

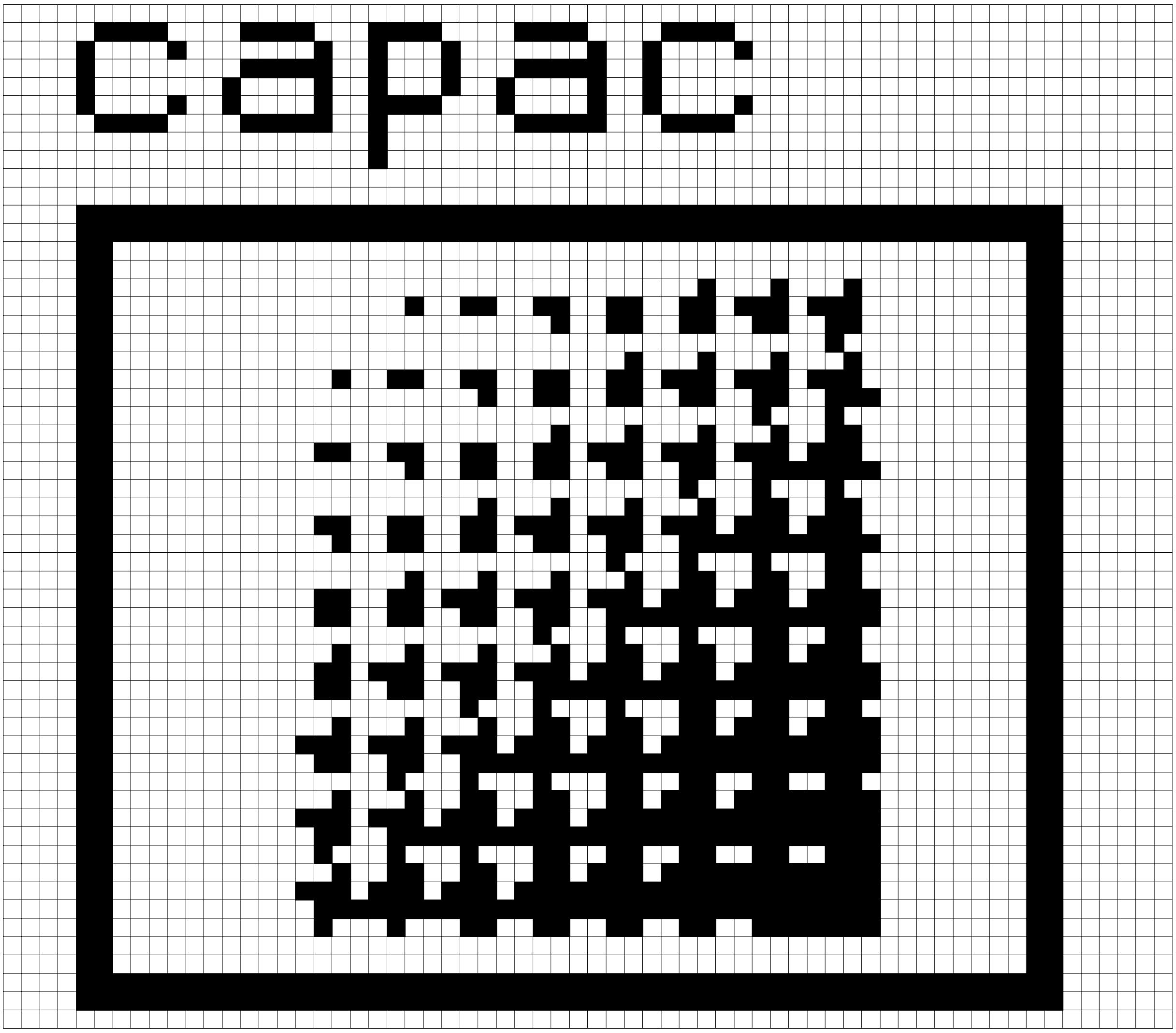



**Figure H.1 – Test datastream page bitmap**

3) The first segment data part:

```
0018: 00 01 00 00 00 01 00 00 00 01 E9 CB F4 00 26 AF
0028: 04 BF F0 78 2F E0 00 40
```

a) The two-byte symbol dictionary flags field:

```
0018: 00 01
```

This field indicates that the segment is encoded using the Huffman coding variant, does not use refinement/aggregate coding, uses Table B.4 for **SDHUFFDH**, and uses Table B.2 for **SDHUFFDW**.

b) The four-byte **SDNUMEXSYMS** field:

```
001A: 00 00 00 01
```

This field indicates that **SDNUMEXSYMS** is 1: one symbol is exported by this symbol dictionary.

c) The four-byte **SDNUMNEWSYMS** field:

```
001E: 00 00 00 01
```

This field indicates that **SDNUMNEWSYMS** is 1: one symbol is defined by this symbol dictionary.

d) The encoded symbol dictionary data:

```
0022: E9 CB F4 00 26 AF 04 BF F0 78 2F E0 00 40
```

This is decoded as follows:

i) Using Table B.4, decode a height class delta height value. This consumes the bits **1110 100**, indicating a height class delta height of 8. The first height class thus has a height of 8 pixels.

ii) Using Table B.2, decode a delta width value. This consumes the bits **1110 010**, indicating a delta width of 5. The first symbol thus has a width of 5 pixels.

iii) Using Table B.2, decode a delta width value. This consumes the bits **111111**, indicating a delta width of OOB. This ends the height class; the height class contains one symbol whose width is 5 pixels and whose height is 8 pixels.

iv) Using Table B.1, decode the size in bytes of the height class collective bitmap. This consumes the bits **01000**, indicating a size of eight bytes.

v) Skip the remaining bits in the last byte read. This consumes the bits **0000000**.

vi) Decode the next eight bytes:

```
0026: 26 AF 04 BF F0 78 2F E0
```

using MMR. This produces the height class collective bitmap, which is also the bitmap for the single symbol in the height class, shown in Figure H.2.

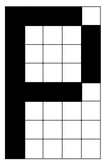



**Figure H.2 – The first symbol in the first symbol dictionary**

vii) Since **SDNUMNEWSYMS** is 1, the last symbol has now been decoded.

viii) Using Table B.1, decode an export run length. This consumes the bits **00000**, indicating that the first 0 symbols are not exported.

ix) Using Table B.1, decode an export run length. This consumes the bits **00001**, indicating that the next 1 symbols are exported. Thus, this symbol dictionary defines one symbol, which is exported.

x) Skip the remaining bits in the last byte. This consumes the bits **000000**.

4) The second segment header:

```
0030: 00 00 00 01 30 00 01 00 00 00 13
```

This segment has a segment number of 1, a type of "Page information" (type 48), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 1, and has a data length of 19 bytes.

5) The second segment data part:

```
003B: 00 00 00 40 00 00 00 38 00 00 00 00 00 00 00 00
004B: 01 00 00
```

a) The page bitmap width field:

```
003B: 00 00 00 40
```

This indicates that the page is 64 pixels wide.

b) The page bitmap height field:

```
003F: 00 00 00 38
```

This indicates that the page is 56 pixels high.

c) The page X resolution field:

```
0043: 00 00 00 00
```

This indicates that the page's X resolution is unknown.

d) The page Y resolution field:

```
0047: 00 00 00 00
```

This indicates that the page's Y resolution is unknown.

e) The page segment flags field:

```
004B: 01
```

This indicates that the page is eventually lossless, the page does not contain any refinements, the page default pixel value is **0**, the page default combination operator is OR, the page does not require any auxiliary buffers, and the page default combination operator is used by every region segment on the page.

f) The page striping information field:

```
004C: 00 00
```

This indicates that the page is not striped.

6) The third segment header:

```
004E: 00 00 00 02 00 01 01 00 00 00 1C
```

This segment has a segment number of 2, a type of "Symbol dictionary" (type 0), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is retained. It is associated with page 1, and has a data length of 28 bytes.

7) The third segment data part:

```
0059: 00 01 00 00 00 02 00 00 00 02 E5 CD F8 00 79 E0
0069: 84 10 81 F0 82 10 86 10 79 F0 00 80
```

a) The two-byte symbol dictionary flags field:

```
0059: 00 01
```

This field indicates that the segment is encoded using the Huffman coding variant, does not use refinement/aggregate coding, uses Table B.4 for **SDHUFFDH**, and uses Table B.2 for **SDHUFFDW**.

b) The four-byte **SDNUMEXSYMS** field:

```
005B: 00 00 00 02
```

This field indicates that **SDNUMEXSYMS** is 2: two symbols are exported by this symbol dictionary.

c) The four-byte **SDNUMNEWSYMS** field:

```
005F: 00 00 00 02
```

This field indicates that **SDNUMNEWSYMS** is 2: two symbols are defined by this symbol dictionary.

d) The encoded symbol dictionary data:

```
0063: E5 CD F8 00 79 E0 84 10 81 F0 82 10 86 10 79 F0
0073: 00 80
```

This is decoded as follows:

i) Using Table B.4, decode a height class delta height value. This consumes the bits **1110 010**, indicating a height class delta height of 6. The first height class thus has a height of 6 pixels.

ii) Using Table B.2, decode a delta width value. This consumes the bits **1110 011**, indicating a delta width of 6. The first symbol thus has a width of 6 pixels.

iii) Using Table B.2, decode a delta width value. This consumes the bit **0**, indicating a delta width of 0. The second symbol thus has a width of 6 pixels.

iv) Using Table B.2, decode a delta width value. This consumes the bits **111111**, indicating a delta width of OOB. This ends the height class; the height class contains two symbols, both of which are 6 pixels wide and 6 pixels high.

v) Using Table B.1, decode the size in bytes of the height class collective bitmap. This consumes the bits **00000**, indicating a size of zero bytes. This indicates that the height class collective bitmap is stored uncompressed. Since the total width of the symbols in the height class is 12, each row of the height class is padded to be 16 bits (2 bytes) wide.

vi) Skip the remaining bits in the last byte read. This consumes the bits **000000**.

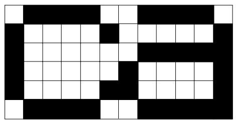
vii) Read the next 12 bytes (6 rows of 2 bytes each), and use the leftmost 12 bits of each row as the height class collective bitmap. These 12 bytes are:

```
0067: 79 E0 84 10 81 F0 82 10 86 10 79 F0
```

or, in binary:

| | |
|---|---|
| **01111001** | **11100000** |
| **10000100** | **00010000** |
| **10000001** | **11110000** |
| **10000010** | **00010000** |
| **10000110** | **00010000** |
| **01111001** | **11110000** |

The height class collective bitmap is therefore as shown in Figure H.3, and the two symbols are as shown in Figure H.4 a) and b).

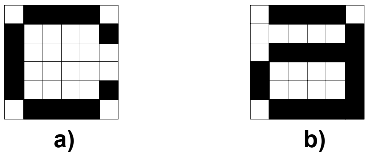


T0829160-99/d44

**Figure H.3 – The height class collective bitmap in the second symbol dictionary**

viii) Since **SDNUMNEWSYMS** is 2, the last symbol has now been decoded.

ix) Using Table B.1, decode an export run length. This consumes the bits **00000**, indicating that the first 0 symbols are not exported.



T0829170-99/d45

**Figure H.4 – The symbols in the second symbol dictionary**

x) Using Table B.1, decode an export run length. This consumes the bits **00010**, indicating that the next 2 symbols are exported. Thus, this symbol dictionary defines two symbols, which are both exported.

xi) Skip the remaining bits in the last byte. This consumes the bits **000000**.

8) The fourth segment header:

```
0075: 00 00 00 03 07 42 00 02 01 00 00 00 31
```

This segment has a segment number of 3, a type of "Immediate lossless text region" (type 7), a short page association field, and does not have the deferred non-retain bit set. It refers to two other segments, segments number 0 and 2; segment 0 should be retained, but segment 2 and this segment should not be retained. It is associated with page 1, and has a data length of 49 bytes.

9) The fourth segment data part:

```
0082: 00 00 00 25 00 00 00 08 00 00 00 04 00 00 00 01
0092: 00 0C 09 00 10 00 00 00 05 01 10 00 00 00 00 00
00A2: 00 00 00 00 00 00 00 00 00 00 0C 40 07 08 70 41
00B2: D0
```

a) The region segment bitmap width field:

```
0082: 00 00 00 25
```

This field indicates that the region bitmap is 37 pixels wide.

b) The region segment bitmap height field:

```
0086: 00 00 00 08
```

This field indicates that the region bitmap is 8 pixels high.

c) The region segment bitmap X location field:

```
008A: 00 00 00 04
```

This field indicates that the region bitmap's left edge is 4 pixels right of the page's left edge.

d) The region segment bitmap Y location field:

```
008E: 00 00 00 01
```

This field indicates that the region bitmap's top edge is 1 pixel down from the page's top edge.

e) The region segment flags field:

```
0092: 00
```

This field indicates that the region should be drawn into the page using the combination operator OR.

f) The two-byte text region segment flags field:

```
0093: 0C 09
```

This field indicates that the segment is encoded using the Huffman coding variant, does not contain any refinements, has a **SBSTRIPS** value of 4, has a reference corner of BOTTOMLEFT, is not transposed, combines its symbols using OR, has a default pixel value of **0**, and has a **SBDSOFFSET** value of 3.

g) The two-byte text region segment Huffman flags field:

```
0095: 00 10
```

This field indicates that the segment uses Table B.6 for **SBHUFFFS**, Table B.8 for **SBHUFFDS**, and Table B.12 for **SBHUFFDT**.

h) The four-byte **SBNUMINSTANCES** field:

```
0097: 00 00 00 05
```

This field indicates that **SBNUMINSTANCES** is 5: five symbol instances are encoded in this text region.

i) The text region segment symbol ID Huffman decoding table:

```
009B: 01 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00AB: 00 0C
```

The two symbol dictionaries referred to by this segment have a total of 3 symbols in them. Decoding the RUNCODE Huffman table, consuming all of the data but the last four bits, gives the following assignment of code lengths to the RUNCODEs:

| | | | |
|---|---|---|---|
| RUNCODE1 | 1 | RUNCODE2 | 1 |

and therefore the following Huffman table for the RUNCODEs:

| | | | |
|---|---|---|---|
| RUNCODE1 | **0** | RUNCODE2 | **1** |

Decoding using this table produces the sequence RUNCODE2, RUNCODE2, RUNCODE1 (consuming the bits **110**). Thus, the first symbol (the "p" from the first symbol dictionary) has a Huffman code length of 2; the second symbol (the "c" from the second symbol dictionary) has a Huffman code length of 2, and the third symbol (the "a" from the second symbol dictionary) has a code length of 1. Applying the Huffman code assignment algorithm gives the table:

| | |
|---|---|
| p | **10** |
| c | **11** |
| a | **0** |

At this point, there is one bit (**0**) remaining in the last of data; this is now skipped.

j) The encoded text region data:

```
00AD: 40 07 08 70 41 D0
```

This is decoded as follows:

i) Using Table B.12, decode a delta T value. This consumes the bit **0**, indicating a delta T value of 4 (the table's decoded value of 1, multiplied by **SBSTRIPS**). The initial STRIPT value is –4.

ii) Using Table B.12, decode a delta T value. This consumes the bits **10**, indicating a delta T value of 8 (2 times **SBSTRIPS**). STRIPT is therefore now 4.

iii) Using Table B.6, decode a first S value. This consumes the bits **00 0000000**, indicating a first S value of 0.

iv) Reading two bits (since **SBSTRIPS**) consumes the bits **01**. The first symbol instance T coordinate is therefore 5 (STRIPT plus the decoded value of 1).

v) Using the symbol ID Huffman table, decode a symbol ID value. This consumes the bits **11**, indicating the symbol "c". Thus, the symbol "c" should be drawn with its lower left corner at (0, 5).

vi) At this point, CURS is 8 (0 plus **SBDSOFFSET** plus the previous symbol's width of 6 minus 1).

vii) Using Table B.8, decode a delta S value. This consumes the bits **00 0**, indicating a delta S value of 0.

viii) Reading two bits consumes the bits **01**. The second symbol instance T coordinate is therefore 5.

ix) Using the symbol ID Huffman table, decode a symbol ID value. This consumes the bit **0**, indicating the symbol "a". Thus, the symbol "a" should be drawn with its lower left corner at (8, 5).

x) At this point, CURS is 16 (8 plus **SBDSOFFSET** plus the previous symbol's width of 6 minus 1).

xi) Using Table B.8, decode a delta S value. This consumes the bits **00 0**, indicating a delta S value of 0.

xii) Reading two bits consumes the bits **11**. The third symbol instance T coordinate is therefore 7.

xiii) Using the symbol ID Huffman table, decode a symbol ID value. This consumes the bits **10**, indicating the symbol "p". Thus, the symbol "p" should be drawn with its lower left corner at (16, 7).

xiv) At this point, CURS is 23 (16 plus **SBDSOFFSET** plus the previous symbol's width of 5 minus 1).

xv) Using Table B.8, decode a delta S value. This consumes the bits **00 0**, indicating a delta S value of 0.

xvi) Reading two bits consumes the bits **01**. The fourth symbol instance T coordinate is therefore 5.

xvii) Using the symbol ID Huffman table, decode a symbol ID value. This consumes the bit **0**, indicating the symbol "a". Thus, the symbol "a" should be drawn with its lower left corner at (8, 5).

xviii) At this point, CURS is 31 (23 plus **SBDSOFFSET** plus the previous symbol's width of 6 minus 1).

xix) Using Table B.8, decode a delta S value. This consumes the bits **00 0**, indicating a delta S value of 0.

xx) Reading two bits consumes the bits **01**. The fifth symbol instance T coordinate is therefore 5.

xxi) Using the symbol ID Huffman table, decode a symbol ID value. This consumes the bits **11**, indicating the symbol "c". Thus, the symbol "c" should be drawn with its lower left corner at (31, 5).

xxii) Using Table B.8, decode a delta S value. This consumes the bits **01**, indicating a delta S value of OOB, indicating the end of this strip. Since **SBNUMINSTANCES** is 5, and five symbol instances have been decoded, there are no more strips.

xxiii) Skip the remaining bits in the last byte read. This consumes the bits **0000**.

k) Decoding the data produced the following list of symbol instances and locations (the locations are where the symbol's lower left corner should be placed):

| Symbol | Location |
|---|---|
| c | (0, 5) |
| a | (8, 5) |
| p | (16, 7) |
| a | (23, 5) |
| c | (31, 5) |

Drawing these symbol instances produces the 37-by-8 pixel region bitmap shown in Figure H.5.

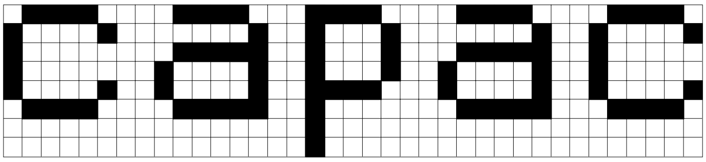



**Figure H.5 – The text region bitmap**

10) The fifth segment header:

    `00B3: 00 00 00 04 27 00 01 00 00 00 2C`

    This segment has a segment number of 4, a type of "Immediate lossless generic region" (type 39), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 1, and has a data length of 44 bytes.

11) The fifth segment data part:

```
00BE: 00 00 00 36 00 00 00 2C 00 00 00 04 00 00 00 0B
00CE: 00 01 26 A0 71 CE A7 FF FF FF FF FF FF FF FF FF
00DE: FF FF FF FF FF FF FF FF FF FF F8 F0
```

   a) The region segment information field:

```
00BE: 00 00 00 36 00 00 00 2C 00 00 00 04 00 00 00 0B
00CE: 00
```

   This indicates that the region bitmap encoded by this segment is 54 pixels wide, 44 pixels high, and its top left corner is 4 pixels right of the page's left edge and 11 pixels down from the page's top edge. It should be drawn into the page using OR.

   b) The region data:

```
00CF: 01 26 A0 71 CE A7 FF FF FF FF FF FF FF FF FF FF
00DF: FF FF FF FF FF FF FF FF FF F8 F0
```

   The first byte (01) is the generic region segment flags byte, and indicates that the region is encoded using MMR. The remaining bytes are the MMR-encoded data for the region bitmap. These bytes MMR-decompress to the 54-by-44 region bitmap shown in Figure H.6.

12) The sixth segment header:

    `00EA: 00 00 00 05 10 01 01 00 00 00 2D`

    This segment has a segment number of 5, a type of "Pattern dictionary" (type 16), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is retained. It is associated with page 1, and has a data length of 45 bytes.

13) The sixth segment data part:

```
00F5: 01 04 04 00 00 00 0F 20 D1 84 61 18 45 F2 F9 7C
0105: 8F 11 C3 9E 45 F2 F9 7D 42 85 0A AA 84 62 2F EE
0115: EC 44 62 22 35 2A 0A 83 B9 DC EE 77 80
```

   a) The one-byte pattern dictionary flags field:

   `00F5: 01`

   This field indicates that the segment is encoded using the MMR coding variant.

   b) The one-byte **HDPW** field:

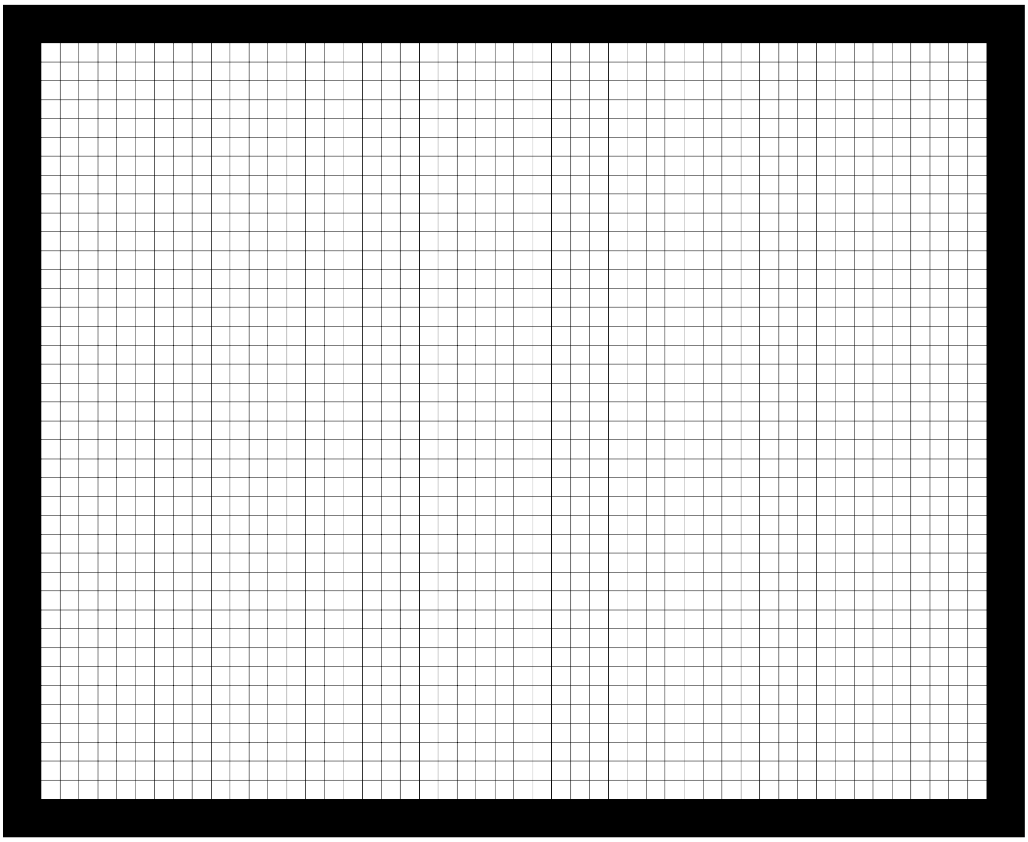
   `00F6: 04`

   This field indicates that **HDPW**, the width of the patterns defined in this dictionary, is 4.

   c) The one-byte **HDPH** field:

   `00F7: 04`

   This field indicates that **HDPH**, the height of the patterns defined in this dictionary, is 4.



T0829190-99/47

**Figure H.6 – The generic region bitmap**

d) The four-byte **GRAYMAX** field:

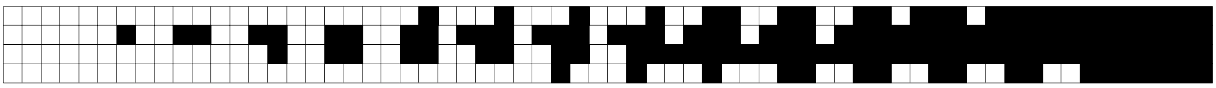
`00F8: 00 00 00 0F`

This field indicates that **GRAYMAX** is 15, and thus there are 16 patterns in this pattern dictionary (numbered 0 through 15).

e) The remaining 38 bytes in the segment:

```
00FC: 20 D1 84 61 18 45 F2 F9 7C 8F 11 C3 9E 45 F2 F9
010C: 7D 42 85 0A AA 84 62 2F EE EC 44 62 22 35 2A 0A
011C: 83 B9 DC EE 77 80
```

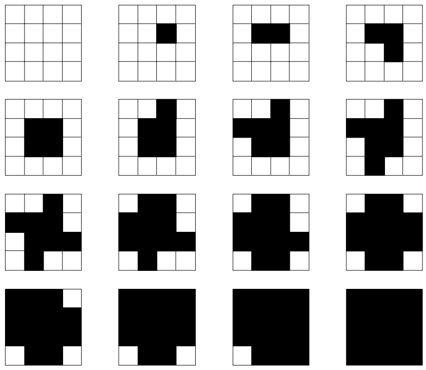
These bytes MMR-decompress to the pattern dictionary's collective bitmap. The width of the bitmap is (**GRAYMAX** + 1) × **HDPW**, and the height is **HDPH** pixels. The 64-by-4 bitmap that is the result of this MMR decompression is shown in Figure H.7.



T0829200-99/d48

**Figure H.7 – The pattern dictionary collective bitmap**

The 16 individual patterns are obtained from the collective bitmap by breaking it into 4-pixel-wide pieces. They are shown in Figure H.8, where pattern number 0 is the top left pattern, pattern number 1 is to its right, and so on.



T0829210-99/d49

**Figure H.8 – The patterns defined in the pattern dictionary**

14) The seventh segment header:

```
0122: 00 00 00 06 17 20 05 01 00 00 00 57
```

This segment has a segment number of 6, a type of "Immediate lossless halftone region" (type 23), a short page association field, and does not have the deferred non-retain bit set. It refers to one other segment, segment number 5; neither segment 5 nor this segment should be retained. It is associated with page 1, and has a data length of 87 bytes.

15) The seventh segment data part:

```
012E: 00 00 00 20 00 00 00 24 00 00 00 10 00 00 00 0F
013E: 00 01 00 00 00 08 00 00 00 09 00 00 00 00 00 00
014E: 00 00 04 00 00 00 AA AA AA AA 80 08 00 80 36 D5
015E: 55 6B 5A D4 00 40 04 2E E9 52 D2 D2 D2 8A A5 4A
016E: 00 20 02 23 E0 95 24 B4 92 8A 4A 92 54 92 D2 4A
017E: 29 2A 49 40 04 00 40
```

a) The region segment information field :

```
012E: 00 00 00 20 00 00 00 24 00 00 00 10 00 00 00 0F
013E: 00
```

This indicates that the region bitmap encoded by this segment is 32 pixels wide, 36 pixels high, and its top left corner is 16 pixels right of the page's left edge and 15 pixels down from the page's top edge. It should be drawn into the page using OR.

b) The halftone region segment flags field:

```
013F: 01
```

This field indicates that the halftone region is encoded using the MMR coding variant. The patterns should be combined using OR. The default pixel value is **0**.

c) The **HGW** field:

```
0140: 00 00 00 08
```

This field indicates that the array of gray-scale values is 8 wide.

d) The **HGH** field:

```
0144: 00 00 00 09
```

This field indicates that the array of gray-scale values is 9 high.

e) The **HGX** field:

```
0148: 00 00 00 00
```

This field indicates that the horizontal offset of the halftone grid is 0 pixels.

f) The **HGY** field:

```
014C: 00 00 00 00
```

This field indicates that the vertical offset of the halftone grid is 0 pixels.

g) The **HRX** field:

`0150: 04 00`

This field indicates that the **HRX** is 1024, and thus the horizontal coordinate of the grid vector is 1024/256 pixels, or 4 pixels.

h) The **HRY** field:

`0152: 00 00`

This field indicates that the vertical coordinate of the grid vector is 0 pixels.

i) The first bitplane:

`0154: AA AA AA AA 80 08 00 80`

Decompressing this with MMR yields the bitmap shown in Figure H.9 a). Note that the last 7 bits in the last byte are skipped over after all the MMR-encoded data has been decoded (i.e. decoding the bitplane is forced to consume an integral number of bytes).

j) The second bitplane:

`015C: 36 D5 55 6B 5A D4 00 40 04`

Decompressing this with MMR yields the bitmap shown in Figure H.9 b).

k) The third bitplane:

`0165: 2E E9 52 D2 D2 D2 8A A5 4A 00 20 02`

Decompressing this with MMR yields the bitmap shown in Figure H.9 c).

l) The fourth bitplane:

```
0171: 23 E0 95 24 B4 92 8A 4A 92 54 92 D2 4A 29 2A 49
0181: 40 04 00 40
```

Decompressing this with MMR yields the bitmap shown in Figure H.9 d).

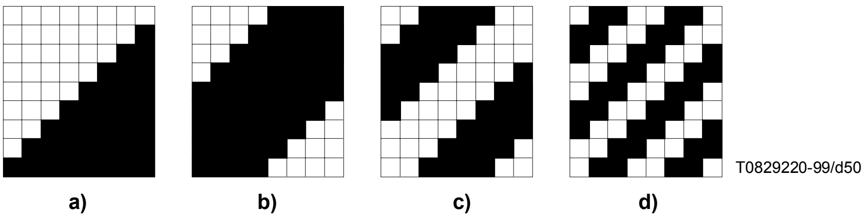



**Figure H.9 – The raw bitplanes for the halftone region**

m) These bitplanes are then Gray-decoded as described in C.5, by XORing the first into the second, then the resulting bitplane into the third, and so on. The resulting bitplanes are shown in Figure H.10.

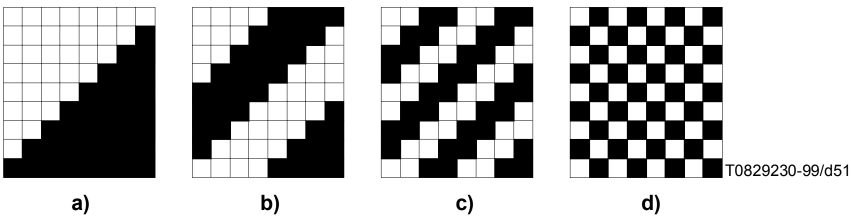



**Figure H.10 – The bitplanes for the halftone region**

n) Stacking up these bitplanes, with the first being the most significant, results in the array of values:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

o) The halftone grid vector and offset produces the following array of locations. Combining this with the array of values results in a list of drawing operations, indicating that the pattern whose index in the pattern dictionary in segment number 5 is that value should be drawn with its upper left pixel at the given location.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| (0, 0) | (4, 0) | (8, 0) | (12, 0) | (16, 0) | (20, 0) | (24, 0) | (28, 0) |
| (0, 4) | (4, 4) | (8, 4) | (12, 4) | (16, 4) | (20, 4) | (24, 4) | (28, 4) |
| (0, 8) | (4, 8) | (8, 8) | (12, 8) | (16, 8) | (20, 8) | (24, 8) | (28, 8) |
| (0, 12) | (4, 12) | (8, 12) | (12, 12) | (16, 12) | (20, 12) | (24, 12) | (28, 12) |
| (0, 16) | (4, 16) | (8, 16) | (12, 16) | (16, 16) | (20, 16) | (24, 16) | (28, 16) |
| (0, 20) | (4, 20) | (8, 20) | (12, 20) | (16, 20) | (20, 20) | (24, 20) | (28, 20) |
| (0, 24) | (4, 24) | (8, 24) | (12, 24) | (16, 24) | (20, 24) | (24, 24) | (28, 24) |
| (0, 28) | (4, 28) | (8, 28) | (12, 28) | (16, 28) | (20, 28) | (24, 28) | (28, 28) |
| (0, 32) | (4, 32) | (8, 32) | (12, 32) | (16, 32) | (20, 32) | (24, 32) | (28, 32) |

p) Performing those drawing operations produces the 32-by-36 region bitmap shown in Figure H.11

16) The eighth segment header:

```
0185: 00 00 00 07 31 00 01 00 00 00 00
```

This segment has a segment number of 7, a type of "End of page" (type 49), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 1, and has a data length of zero bytes.

17) The region segment bitmaps are combined as follows, taking into account the page default pixel value and each region segment's combination operator. First, the page bitmap (64 pixels wide and 56 pixels high) is filled with **0**, the page default pixel value. Next, the bitmap shown in Figure H.5 is drawn using OR into the page bitmap with its top left pixel at location (4, 1). Next, the bitmap shown in Figure H.6 is drawn using OR into the page bitmap with its top left pixel at location (4, 11). Finally, the bitmap shown in Figure H.11 is drawn using OR into the page bitmap with its top left pixel at location (16, 15). After all this has been done, the resulting bitmap is the one shown in Figure H.1.

18) The ninth segment header:

```
0190: 00 00 00 08 30 00 02 00 00 00 13
```

This segment has a segment number of 8, a type of "Page information" (type 48), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 2, and has a data length of 19 bytes.

19) The ninth segment data part:

```
019B: 00 00 00 40 00 00 00 38 00 00 00 00 00 00 00 00
01AB: 01 00 00
```

This contains the same information as does segment number 1.

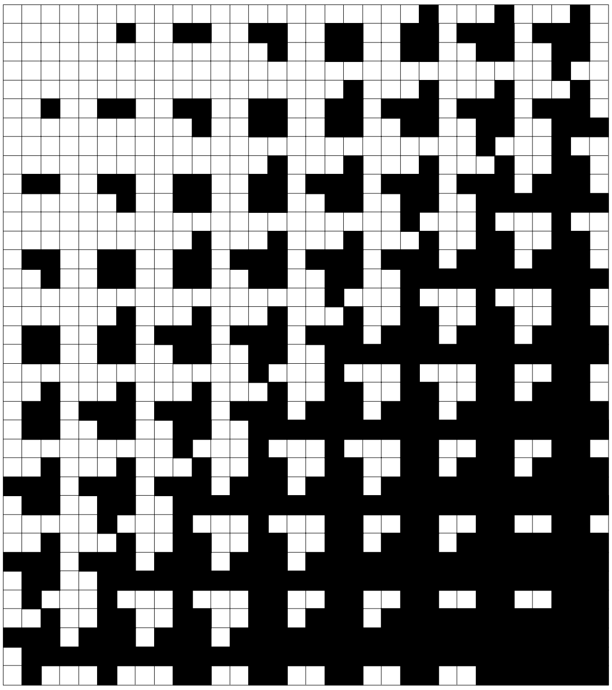



**Figure H.11 – The halftone region bitmap**

20) The tenth segment header:

```
01AE: 00 00 00 09 00 01 02 00 00 00 1B
```

This segment has a segment number of 9, a type of "Symbol dictionary" (type 0), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is retained. It is associated with page 2, and has a data length of 27 bytes.

21) The tenth segment data part:

```
01B9: 08 00 02 FF 00 00 00 02 00 00 00 02 4F E7 8C 20
01C9: 0E 1D C7 CF 01 11 C4 B2 6F FF AC
```

a) The two-byte symbol dictionary flags field:

```
01B9: 08 00
```

This field indicates that the segment is encoded using the arithmetic coding variant and does not use refinement/aggregate coding. **SDTEMPLATE** has the value 2. The "bitmap coding context used" and "bitmap coding context retained" bits are both **0**.

b) The symbol dictionary AT flags field:

```
01BB: 02 FF
```

This field indicates that $\mathbf{SDATX_1}$ is 2 and $\mathbf{SDATY_1}$ is –1; thus, AT pixel $A_1$ is at location (2, –1), which is the nominal value for template 2.

c) The four-byte **SDNUMEXSYMS** field:

```
01BD: 00 00 00 02
```

This field indicates that **SDNUMEXSYMS** is 2: two symbols are exported by this symbol dictionary.

d) The four-byte **SDNUMNEWSYMS** field:

```
01C1: 00 00 00 02
```

This field indicates that **SDNUMNEWSYMS** is 2: two symbols are defined by this symbol dictionary.

e) The encoded symbol dictionary data:

```
01C5: 4F E7 8C 20 0E 1D C7 CF 01 11 C4 B2 6F FF AC
```

Decoding this gives the same two symbols shown in Figure H.4 a) and b).

22) The eleventh segment header:

```
01D4: 00 00 00 0A 07 40 00 09 02 00 00 00 1F
```

This segment has a segment number of 10, a type of "Immediate lossless text region" (type 7), a short page association field, and does not have the deferred non-retain bit set. It refers to two other segments, segments number 0 and 9; segment 0, segment 9, and this segment should not be retained. It is associated with page 2, and has a data length of 31 bytes.

23) The eleventh segment data part:

```
01E1: 00 00 00 25 00 00 00 08 00 00 00 04 00 00 00 01
01F1: 00 0C 08 00 00 00 05 8D 6E 5A 12 40 85 FF AC
```

a) The region segment information field:

```
01E1: 00 00 00 25 00 00 00 08 00 00 00 04 00 00 00 01
01F1: 00
```

This indicates that the region bitmap encoded by this segment is 37 pixels wide, 8 pixels high, and its top left corner is 4 pixels right of the page's left edge and 1 pixel down from the page's top edge. It should be drawn into the page using OR.

b) The two-byte text region segment flags field:

```
01F2: 0C 08
```

This field indicates that the segment is encoded using the arithmetic coding variant, does not contain any refinements, has a **SBSTRIPS** value of 4, has a reference corner of BOTTOMLEFT, is not transposed, combines its symbols using OR, has a default pixel value of **0**, and has a **SBDSOFFSET** value of 3.

c) The four-byte **SBNUMINSTANCES** field:

```
01F4: 00 00 00 05
```

This field indicates that **SBNUMINSTANCES** is 5: five symbol instances are encoded in this text region.

d) The encoded text region data:

```
01F8: 8D 6E 5A 12 40 85 FF AC
```

Decoding this follows the exact same sequence that was seen in decoding segment number 3 (page 1's text region segment), and results in the region bitmap shown in Figure H.5.

24) The twelfth segment header:

```
0200: 00 00 00 0B 27 00 02 00 00 00 23
```

This segment has a segment number of 11, a type of "Immediate lossless generic region" (type 39), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 2, and has a data length of 35 bytes.

25) The twelfth segment data part:

```
020B: 00 00 00 36 00 00 00 2C 00 00 00 04 00 00 00 0B
021B: 00 08 03 FF FD FF 02 FE FE FE 04 EE ED 87 FB CB
022B: 2B FF AC
```

a) The region segment information field:

```
020B: 00 00 00 36 00 00 00 2C 00 00 00 04 00 00 00 0B
021B: 00
```

This indicates that the region bitmap encoded by this segment is 54 pixels wide, 44 pixels high, and its top left corner is 4 pixels right of the page's left edge and 11 pixels down from the page's top edge. It should be drawn into the page using OR.

b) The one-byte generic region flags field:

```
021C: 08
```

This indicates that the region is encoded using arithmetic coding, that **GBTEMPLATE** is 0, and **TPGDON** is **1**.

c) The generic region segment AT flags field:

```
021D: 03 FF FD FF 02 FE FE FE
```

This field is eight bytes long because **GBTEMPLATE** is 0, and there are thus four AT pixels whose positions must be determined. The AT pixels are located with $A_1$ at (3, –1); $A_2$ at (–3, –1); $A_3$ at (2, –2); and $A_4$ at (–2, –2). These are the nominal positions of those pixels for this template.

d) The region data:

`0225: 04 EE ED 87 FB CB 2B FF AC`

Decoding this using the decoded values of **GBTEMPLATE**, **TPGDON**, and the AT pixel locations produces the 54-by-44 region bitmap shown in Figure H.6.

26) The thirteenth segment header:

`022E: 00 00 00 0C 10 01 02 00 00 00 1C`

This segment has a segment number of 12, a type of "Pattern dictionary" (type 16), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is retained. It is associated with page 2, and has a data length of 28 bytes.

27) The thirteenth segment data part:

```
0239: 06 04 04 00 00 00 0F 90 71 6B 6D 99 A7 AA 49 7D
0249: F2 E5 48 1F DC 68 BC 6E 40 BB FF AC
```

a) The one-byte pattern dictionary flags field:

`0239: 06`

This field indicates that the segment is encoded using the arithmetic coding variant, and that **HDTEMPLATE** is 3.

b) The one-byte **HDPW** field:

`023A: 04`

This field indicates that **HDPW** is 4.

c) The one-byte **HDPH** field:

`023B: 04`

This field indicates that **HDPH** is 4.

d) The four-byte **GRAYMAX** field:

`023C: 00 00 00 0F`

This field indicates that **GRAYMAX** is 15, and thus there are 16 patterns in this pattern dictionary.

e) The remaining 21 bytes in the segment:

```
0240: 90 71 6B 6D 99 A7 AA 49 7D F2 E5 48 1F DC 68 BC
0250: 6E 40 BB FF AC
```

These bytes decompress, using the pattern dictionary decoding procedure, to the collective bitmap shown in Figure H.7, and thus the 16 patterns defined by this pattern dictionary are as shown in Figure H.8.

28) The fourteenth segment header:

`0255: 00 00 00 0D 17 20 0C 02 00 00 00 3E`

This segment has a segment number of 13, a type of "Immediate lossless halftone region" (type 23), a short page association field, and does not have the deferred non-retain bit set. It refers to one other segment, segment number 12; neither segment 12 nor this segment should be retained. It is associated with page 2, and has a data length of 62 bytes.

29) The fourteenth segment data part:

```
0261: 00 00 00 20 00 00 00 24 00 00 00 10 00 00 00 0F
0271: 00 02 00 00 00 08 00 00 00 09 00 00 00 00 00 00
0281: 00 00 04 00 00 00 87 CB 82 1E 66 A4 14 EB 3C 4A
0291: 15 FA CC D6 F3 B1 6F 4C ED BF A7 BF FF AC
```

a) The region segment information field:

```
0261: 00 00 00 20 00 00 00 24 00 00 00 10 00 00 00 0F
0271: 00
```

This indicates that the region bitmap encoded by this segment is 32 pixels wide, 36 pixels high, and its top left corner is 16 pixels right of the page's left edge and 15 pixels down from the page's top edge. It should be drawn into the page using OR.

b) The halftone region segment flags field:

`0272: 02`

This field indicates that the halftone region is encoded using the arithmetic coding variant, and that **HTEMPLATE** is 1. The patterns should be combined using OR. The default pixel value is **0**.

c) The other parameters:

`0273: 00 00 00 08 00 00 00 09 00 00 00 00 00 00 00 00`

`0283: 04 00 00 00`

The following fields indicate that **HGW** is 8, **HGH** is 9, **HGX** is 0, **HGY** is 0, **HRX** is 1024, and **HRY** is 0.

d) The four bitplanes:

`0287: 87 CB 82 1E 66 A4 14 EB 3C 4A 15 FA CC D6 F3 B1`

`0297: 6F 4C ED BF A7 BF FF AC`

Decoding four 8-by-9 bitplanes from this data results in the four bitplanes shown in Figure H.9. As in segment number 6, *Gray*-decoding these bitplanes, combining them into an array of values, and drawing the patterns from the pattern dictionary according to that array and the halftone grid parameters results in the region bitmap shown in Figure H.11.

30) The fifteenth segment header:

`029F: 00 00 00 0E 31 00 02 00 00 00 00`

This segment has a segment number of 14, a type of "End of page" (type 49), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 2, and has a data length of zero bytes.

31) The page bitmap is made by combining the three region bitmaps in the identical way that they were combined in page 1, resulting in the same page bitmap.

32) The sixteenth segment header:

`02AA: 00 00 00 0F 30 00 03 00 00 00 13`

This segment has a segment number of 15, a type of "Page information" (type 48), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 3, and has a data length of 19 bytes.

33) The sixteenth segment data part:

`02B5: 00 00 00 25 00 00 00 08 00 00 00 00 00 00 00 00`

`02C5: 01 00 00`

This indicates that the page is 37 pixels wide, is 8 pixels high, has unknown X and Y resolution, is eventually lossless, does not contain any refinements, has a default pixel value of **0**, a default combination operator of OR, does not require any auxiliary buffers, and uses the page default combination operator in every region on the page.

34) The seventeenth segment header:

`02C8: 00 00 00 10 00 01 00 00 00 00 16`

This segment has a segment number of 16, a type of "Symbol dictionary" (type 0), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is retained. It is associated with no page, and has a data length of 22 bytes.

35) The seventeenth segment data part:

`02D3: 08 00 02 FF 00 00 00 01 00 00 00 01 4F E7 8D 68`

`02E3: 1B 14 2F 3F FF AC`

a) The two-byte symbol dictionary flags field:

`02D3: 08 00`

This field indicates that the segment is encoded using the arithmetic coding variant and does not use refinement/aggregate coding. **SDTEMPLATE** has the value 2. The "bitmap coding context used" and "bitmap coding context retained" bits are both **0**.

b) The symbol dictionary AT flags field:

`02D5: 02 FF`

This field indicates that $\mathbf{SDATX_1}$ is 2 and $\mathbf{SDATY_1}$ is –1; thus, AT pixel $A_1$ is at location (2, –1), which is the nominal value for template 2.

c) The four-byte **SDNUMEXSYMS** field:

`02D7: 00 00 00 01`

This field indicates that **SDNUMEXSYMS** is 1: one symbol is exported by this symbol dictionary.

d) The four-byte **SDNUMNEWSYMS** field:

`02DB: 00 00 00 01`

This field indicates that **SDNUMNEWSYMS** is 1: one symbol is defined by this symbol dictionary.

e) The encoded symbol dictionary data:

`02DF: 4F E7 8D 68 1B 14 2F 3F FF AC`

This is decoded as follows:

i) Reset all the arithmetic coding statistics to zero.

ii) Using the IADH arithmetic integer decoding procedure, decode a height class delta height value. The value decoded is 6, indicating that the first height class is 6 pixels high.

iii) Using the IADW arithmetic integer decoding procedure, decode a delta width value. The value decoded is 6. The first symbol thus has a width of 6 pixels.

iv) Using the generic region decoding procedure, with **GBTEMPLATE** and the AT pixel $A_1$ set as described in the symbol dictionary data header, decode a 6 × 6 bitmap. This produces the bitmap shown in Figure H.12 a).

v) Using the IADW arithmetic integer decoding procedure, decode a delta width value. The value decoded is OOB, indicating the end of the height class.

vi) Since **SDNUMNEWSYMS** is 1, the last symbol has now been decoded.

vii) Using the IAEX arithmetic integer decoding procedure, decode an export run length. The value decoded is 0, indicating that the first 0 symbols are not exported.

viii) Using the IAEX arithmetic integer decoding procedure, decode an export run length. The value decoded is 1, indicating that the next 1 symbols are exported. Thus, this symbol dictionary defines one symbol, which is exported.

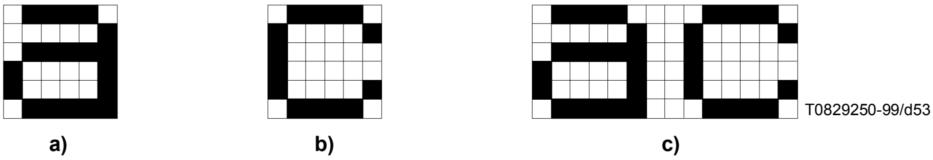



**Figure H.12 – The symbols in the symbol dictionary on the third page**

36) The eighteenth segment header:

`02E9: 00 00 00 11 00 21 10 03 00 00 00 20`

This segment has a segment number of 17, a type of "Symbol dictionary" (type 0), a short page association field, and does not have the deferred non-retain bit set. It refers to one other segment, segment 16. Segment 16 is not retained, but this segment is retained. It is associated with page 3, and has a data length of 32 bytes.

37) The eighteenth segment data part:

`02F5: 08 02 02 FF FF FF FF FF 00 00 00 03 00 00 00 02`

`0305: 4F E9 D7 D5 90 C3 B5 26 A7 FB 6D 14 98 3F FF AC`

a) The two-byte symbol dictionary flags field:

`02F5: 08 02`

This field indicates that the segment is encoded using the arithmetic coding variant and uses refinement/aggregate coding. **SDTEMPLATE** has the value 2. **SDRTEMPLATE** has the value 0. The "bitmap coding context used" and "bitmap coding context retained" bits are both **0**.

b) The symbol dictionary AT flags field:

`02F7: 02 FF`

This field indicates that $\mathbf{SDATX_1}$ is 2 and $\mathbf{SDATY_1}$ is –1; thus, AT pixel $A_1$ is at location (2, –1), which is the nominal value for template 2.

c) The symbol dictionary refinement AT flags:

`02F9: FF FF FF FF`

This field indicates that $\mathbf{SDRATX_1}$ is –1, $\mathbf{SDRATY_1}$ is –1, $\mathbf{SDRATX_2}$ is –1, and $\mathbf{SDRATY_2}$ is –1. Thus, AT pixel $RA_1$ is at location (–1, –1) and AT pixel $RA_2$ is at location (–1, –1), which are the nominal locations for refinement template 0.

d) The four-byte **SDNUMEXSYMS** field:

```
02FD: 00 00 00 03
```

This field indicates that **SDNUMEXSYMS** is 3: three symbols are exported by this symbol dictionary.

e) The four-byte **SDNUMNEWSYMS** field:

```
0301: 00 00 00 02
```

This field indicates that **SDNUMNEWSYMS** is 2: two symbols are defined by this symbol dictionary.

f) The encoded symbol dictionary data:

```
0305: 4F E9 D7 D5 90 C3 B5 26 A7 FB 6D 14 98 3F FF AC
```

This is decoded as follows:

i) Reset all the arithmetic coding statistics to zero.

ii) Using the IADH arithmetic integer decoding procedure, decode a height class delta height value. The value decoded is 6, indicating that the first height class is 6 pixels high.

iii) Using the IADW arithmetic integer decoding procedure, decode a delta width value. The value decoded is 6. The first symbol thus has a width of 6 pixels.

iv) Using the IAAI arithmetic integer decoding procedure, decode a count of symbol instances in the aggregation that forms the first symbol. The value decoded is 1.

v) Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 0. The first symbol is thus the one identified by symbol ID 0, which is the first (and only) symbol in the dictionary (segment 16) referenced by this segment.

vi) Using the IARDX arithmetic integer decoding procedure, decode a symbol instance refinement delta X value. The value decoded is 0.

vii) Using the IARDY arithmetic integer decoding procedure, decode a symbol instance refinement delta Y value. The value decoded is 0. The refinement is thus done with the refined symbol aligned with the reference symbol (i.e. **GRREFERENCEDX** and **GRREFERENCEDY** are both 0).

viii) Using the generic refinement region decoding procedure, decode the refined symbol instance bitmap. The reference bitmap is the bitmap shown in Figure H.12 a) and the refined bitmap, which is the bitmap of the first symbol defined in this symbol dictionary, is the bitmap shown in Figure H.12 b).

ix) Using the IADW arithmetic integer decoding procedure, decode a delta width value. The value decoded is 8. The second symbol thus has a width of 14 pixels.

x) Using the IAAI arithmetic integer decoding procedure, decode a count of symbol instances in the aggregation that forms the second symbol. The value decoded is 2.

xi) Using the text region decoding procedure, with parameters set as described in 6.5.8.2, decode a 14 × 6 pixel text region:

- Using the IADT arithmetic integer decoding procedure, decode the initial STRIPT value. The value decoded is 0.
- Using the IADT arithmetic integer decoding procedure, decode a delta T value. The value decoded is 0.
- Using the IAFS arithmetic integer decoding procedure, decode a first S value. The value decoded is 0. The reference corner (top left corner in this case) of the first symbol instance in the aggregation is thus at (0, 0).
- Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 0. The first symbol is thus the one identified by symbol ID 0, which is the first (and only) symbol in the dictionary (segment 16) referenced by this segment.
- Using the IARI arithmetic integer decoding procedure, decode a refinement flag. The value decoded is **0**, indicating that the first symbol instance is not refined.
- At this point CURS is 5. Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is 3. The reference corner of the second symbol instance in the aggregation is thus at (8, 0).
- Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 1. The second symbol is thus the one identified by symbol ID 1, which is the first (and, thus far only) symbol defined in this symbol dictionary.

- Using the IARI arithmetic integer decoding procedure, decode a refinement flag. The value decoded is **0**, indicating that the first symbol instance is not refined.
- Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is OOB, indicating the end of the strip.
- Decoding of the text region is now complete. The text region bitmap, which is the bitmap of the second symbol defined in the symbol dictionary, is shown in Figure H.12 c). This is obtained by drawing Figure H.12 a) at (0, 0) and Figure H.12 b) at (8, 0).

xii) Using the IADW arithmetic integer decoding procedure, decode a delta width value. The value decoded is OOB, indicating the end of the height class.

xiii) Since **SDNUMNEWSYMS** is 2, the last symbol has now been decoded.

xiv) Using the IAEX arithmetic integer decoding procedure, decode an export run length. The value decoded is 0, indicating that the first 0 symbols are not exported.

xv) Using the IAEX arithmetic integer decoding procedure, decode an export run length. The value decoded is 3, indicating that the next 3 symbols are exported. Thus, this symbol dictionary imports one symbol and defines two symbols, and all three are exported.

Note that the symbol from segment 16 is re-exported by this dictionary. The three symbols exported by this dictionary are therefore the three symbols shown in Figure H.12. Thus, a text region may use the symbol shown in Figure H.12 a) (originally defined in segment 16) by referring to segment 17, even though segment 16 is not retained past the end of segment 17.

38) The nineteenth segment header:

```
0315: 00 00 00 12 07 20 11 03 00 00 00 25
```

This segment has a segment number of 18, a type of "Immediate lossless text region" (type 7), a short page association field, and does not have the deferred non-retain bit set. It refers to one other segment, segment 17. Segment 17 and this segment should not be retained. It is associated with page 3, and has a data length of 37 bytes.

39) The nineteenth segment data part:

```
0321: 00 00 00 25 00 00 00 08 00 00 00 00 00 00 00 00
0331: 00 8C 12 00 00 00 04 A9 5C 8B F4 C3 7D 96 6A 28
0341: E5 76 8F FF AC
```

a) The region segment information field:

```
0321: 00 00 00 25 00 00 00 08 00 00 00 00 00 00 00 00
0331: 00
```

This indicates that the region bitmap encoded by this segment is 37 pixels wide, 8 pixels high, and its top left corner is 0 pixels right of the page's left edge and 0 pixel down from the page's top edge. It should be drawn into the page using OR.

b) The two-byte text region segment flags field:

```
0332: 8C 12
```

This field indicates that the segment is encoded using the arithmetic coding variant, contains refinements, has a **SBSTRIPS** value of 1, has a reference corner of TOPLEFT, is not transposed, combines its symbols using OR, has a default pixel value of **0**, has a **SBDSOFFSET** value of 3, and has a **SBRTEMPLATE** value of 1.

c) The four-byte **SBNUMINSTANCES** field:

```
0334: 00 00 00 04
```

This field indicates that **SBNUMINSTANCES** is 4: four-symbol instances are encoded in this text region.

d) The encoded text region data:

```
0338: A9 5C 8B F4 C3 7D 96 6A 28 E5 76 8F FF AC
```

This is decoded as follows:

i) Reset all the arithmetic coding statistics to zero.

ii) Using the IADT arithmetic integer decoding procedure, decode the initial STRIPT value. The value decoded is 0.

iii) Using the IADT arithmetic integer decoding procedure, decode a delta T value. The value decoded is 0.

iv) Using the IAFS arithmetic integer decoding procedure, decode a first S value. The value decoded is 0. The reference corner (top left corner in this case) of the first symbol instance in the text region is thus at (0, 0).

v) Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 1. The first symbol instance thus uses the symbol shown in Figure H.12 b).

vi) Using the IARI arithmetic integer decoding procedure, decode a refinement flag. The value decoded is **0**, indicating that the first symbol instance is not refined.

vii) At this point CURS is 5. Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is 0. After adding in **SBDSOFFSET**, the reference corner of the second symbol instance in the aggregation is thus at (8, 0).

viii) Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 0. The second symbol instance thus uses the symbol shown in Figure H.12 a).

ix) Using the IARI arithmetic integer decoding procedure, decode a refinement flag. The value decoded is **0**, indicating that the first symbol instance is not refined.

x) At this point CURS is 13. Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is 0. After adding in **SBDSOFFSET**, the reference corner of the third symbol instance in the aggregation is thus at (16, 0).

xi) Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 1. The third symbol instance thus uses the symbol shown in Figure H.12 b).

xii) Using the IARI arithmetic integer decoding procedure, decode a refinement flag. The value decoded is **1**, indicating that the third symbol instance is refined.

xiii) Using the IARDW arithmetic integer decoding procedure, decode a symbol instance refinement delta width value. The value decoded is –1.

xiv) Using the IARDH arithmetic integer decoding procedure, decode a symbol instance refinement delta height value. The value decoded is 2. Since the reference symbol is 6 × 6 pixels, the refined symbol instance is therefore 5 × 8 pixels.

xv) Using the IARDX arithmetic integer decoding procedure, decode a symbol instance refinement delta X value. The value decoded is 1.

xvi) Using the IARDY arithmetic integer decoding procedure, decode a symbol instance refinement delta Y value. The value decoded is –2. **GRREFERENCEDX** and **GRREFERENCEDY** are therefore set as:

$$\textbf{GRREFERENCEDX} = \lfloor -1/2 \rfloor + 1 = -1 + 1 = 0$$

$$\textbf{GRREFERENCEDY} = \lfloor 2/2 \rfloor - 2 = 1 - 2 = -1$$

so the refinement is done with the refined symbol placed so that the top left pixel of the refined symbol is aligned with pixel (0, 1) of the reference symbol.

xvii) Using the generic refinement region decoding procedure, decode the refined symbol instance bitmap. The reference bitmap is the bitmap shown in Figure H.12 b) and the refined bitmap, which is the bitmap for the third symbol instance, is the bitmap shown in Figure H.2.

xviii) At this point CURS is 18. Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is 0. After adding in **SBDSOFFSET**, the reference corner of the third symbol instance in the aggregation is thus at (23, 0).

xix) Using the IAID arithmetic integer decoding procedure, decode a symbol ID. The value decoded is 2. The fourth symbol instance thus uses the symbol shown in Figure H.12 c).

xx) Using the IADS arithmetic integer decoding procedure, decode a delta S value. The value decoded is OOB, indicating the end of the strip.

xxi) Decoding of the text region is now complete. The text region bitmap is the bitmap shown in Figure H.5. It is obtained by drawing Figure H.12 b) with its top left corner at (0, 0), Figure H.12 a) with its top left corner at (8, 0), Figure H.2 with its top left corner at (16, 0), and Figure H.12 c) with its top left corner at (23, 0).

40) The twentieth segment header:

    0346: 00 00 00 13 31 00 03 00 00 00 00

    This segment has a segment number of 19, a type of "End of page" (type 49), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with page 3, and has a data length of zero bytes.

41) The page bitmap is formed from the page's only region segment, segment number 18, and is equal to that region segment's bitmap (Figure H.5).

42) The twenty-first segment header:

    0351: 00 00 00 14 33 00 00 00 00 00 00

    This segment has a segment number of 20, a type of "End of file" (type 51), a short page association field, and does not have the deferred non-retain bit set. It refers to no other segments, and is not retained. It is associated with no page, and has a data length of zero bytes.

## H.2 Test sequence for arithmetic coder

In this subclause, a small data set is provided for testing the arithmetic encoder and decoder. The test is structured to test many of the encoder and decoder paths, but it is impossible, in a short test sequence, to check all of them so agreement with the results of this test, unfortunately, does not guarantee a completely correct implementation.

The decisions to be encoded, packed into 32 bytes and shown in hexadecimal, are:

```
00 02 00 51 00 00 00 C0 03 52 87 2A AA AA AA AA
82 C0 20 00 FC D7 9E F6 BF 7F ED 90 4F 46 A3 BF
```

The encoded data that should be obtained by encoding that sequence, shown as 30 hexadecimal bytes, are:

```
84 C7 3B FC E1 A1 43 04 02 20 00 00 41 0D BB 86
F4 31 7F FF 88 FF 37 47 1A DB 6A DF FF AC
```

Decoding those 30 bytes should produce the 32 original bytes.

Table H.1 provides a bit-by-bit list of the arithmetic encoder and decoder operation. The first line in this table corresponds to the INITENC and INITDEC operations. The value of the byte before the first byte in the output buffer is assumed to be 0x00, making the initial value of B 0x00. The last line in the table corresponds to the FLUSH operation.

For this entire test, a single value of CX is used. I(CX) is initially 0 and MPS(CX) is initially **0**.

The first column is the event counter EC. The second column is the decision D to be encoded. The third and fourth columns give the values of I(CX) and MPS(CX). The fifth column shows an "X" indicating that the conditional exchange will occur when encoding (decoding) the current decision. The sixth column shows the current Qe value corresponding to I(CX) (see Table E.1). The seventh column shows the value of the register A before the decision is encoded (decoded). Note that the A register is always greater than or equal to `0x8000`.

The variables up to this point were common for the encoder and decoder. The next four columns (C, CT, B, OUT) are only for the encoder. The next four columns (C, CT, B, IN) are only for the decoder. The final column (C) shows the C register for the software-conventions decoder, as it is the only value that differs for the software-conventions decoder. All the values shown for the C registers are given before the current decision is encoded (decoded).

For the encoder, CT is a counter indicating when a byte is ready for output from register C. The column under B shows the byte in variable B waiting to be sent out. This byte can sometimes change from a carry-over. Finally, for the encoder the compressed bytes are listed under column OUT. A byte is considered to be "output" when the compressed data pointer BP is advanced to point beyond it.

The decoder's counter CT indicates when to input the next byte from the compressed data. The column under B shows the value of the B register, which is used to determine when a bit-stuffing has occurred. The column under IN shows the bytes that are consumed. A byte is considered to be "consumed" when the compressed data pointer BP is advanced to point at it. Note that the final `0xAC` byte is never consumed, according to this definition, though it is read as part of BYTEIN.

**Table H.1 – Encoder and decoder trace data**

| Common | | | | | | | Encoder | | | | Decoder | | | | Software Conven-tions Decoder |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EC | D | I | MPS | CE | Qe hex | A hex | C hex | CT | B hex | OUT hex | C hex | CT | B hex | IN hex | C hex |
| 0 | | | | | | | | | 00 | | 00000000 | | | 84 C7 | 00000000 |
| 1 | 0 | 0 | 0 | X | 5601 | 8000 | 00000000 | 12 | 00 | | 42638000 | 1 | C7 | | 3D9C0000 |
| 2 | 0 | 1 | 0 | | 3401 | AC02 | 00000000 | 11 | 00 | | 84C70000 | 0 | C7 | 3B | 273A0000 |
| 3 | 0 | 2 | 0 | | 1801 | F002 | 00006802 | 10 | 00 | | A18C7600 | 7 | 3B | | 4E758800 |
| 4 | 0 | 2 | 0 | | 1801 | D801 | 00008003 | 10 | 00 | | 898B7600 | 7 | 3B | | 4E758800 |
| 5 | 0 | 2 | 0 | | 1801 | C000 | 00009804 | 10 | 00 | | 718A7600 | 7 | 3B | | 4E758800 |
| 6 | 0 | 2 | 0 | | 1801 | A7FF | 0000B005 | 10 | 00 | | 59897600 | 7 | 3B | | 4E758800 |
| 7 | 0 | 2 | 0 | | 1801 | 8FFE | 0000C806 | 10 | 00 | | 41887600 | 7 | 3B | | 4E758800 |
| 8 | 0 | 3 | 0 | | 0AC1 | EFFA | 0001C00E | 9 | 00 | | 530EEC00 | 6 | 3B | | 9CEB1000 |
| 9 | 0 | 3 | 0 | | 0AC1 | E539 | 0001CACF | 9 | 00 | | 484DEC00 | 6 | 3B | | 9CEB1000 |
| 10 | 0 | 3 | 0 | | 0AC1 | DA78 | 0001D590 | 9 | 00 | | 3D8CEC00 | 6 | 3B | | 9CEB1000 |
| 11 | 0 | 3 | 0 | | 0AC1 | CFB7 | 0001E051 | 9 | 00 | | 32CBEC00 | 6 | 3B | | 9CEB1000 |
| 12 | 0 | 3 | 0 | | 0AC1 | C4F6 | 0001EB12 | 9 | 00 | | 280AEC00 | 6 | 3B | | 9CEB1000 |
| 13 | 0 | 3 | 0 | | 0AC1 | BA35 | 0001F5D3 | 9 | 00 | | 1D49EC00 | 6 | 3B | | 9CEB1000 |
| 14 | 0 | 3 | 0 | | 0AC1 | AF74 | 00020094 | 9 | 00 | | 1288EC00 | 6 | 3B | | 9CEB1000 |
| 15 | 1 | 3 | 0 | | 0AC1 | A4B3 | 00020B55 | 9 | 00 | | 07C7EC00 | 6 | 3B | | 9CEB1000 |
| 16 | 0 | 12 | 0 | | 1C01 | AC10 | 0020B550 | 5 | 00 | | 7C7EC000 | 2 | 3B | | 2F910000 |
| 17 | 0 | 12 | 0 | | 1C01 | 900F | 0020D151 | 5 | 00 | | 607DC000 | 2 | 3B | | 2F910000 |
| 18 | 0 | 13 | 0 | | 1601 | E81C | 0041DAA4 | 4 | 00 | | 88F98000 | 1 | 3B | | 5F220000 |
| 19 | 0 | 13 | 0 | | 1601 | D21B | 0041F0A5 | 4 | 00 | | 72F88000 | 1 | 3B | | 5F220000 |
| 20 | 0 | 13 | 0 | | 1601 | BC1A | 004206A6 | 4 | 00 | | 5CF78000 | 1 | 3B | | 5F220000 |
| 21 | 0 | 13 | 0 | | 1601 | A619 | 00421CA7 | 4 | 00 | | 46F68000 | 1 | 3B | | 5F220000 |
| 22 | 0 | 13 | 0 | | 1601 | 9018 | 004232A8 | 4 | 00 | | 30F58000 | 1 | 3B | | 5F220000 |
| 23 | 0 | 29 | 0 | | 1101 | F42E | 00849152 | 3 | 00 | | 35E90000 | 0 | 3B | | BE440000 |
| 24 | 0 | 29 | 0 | | 1101 | E32D | 0084A253 | 3 | 00 | | 24E80000 | 0 | 3B | | BE440000 |
| 25 | 0 | 29 | 0 | | 1101 | D22C | 0084B354 | 3 | 00 | | 13E70000 | 0 | 3B | | BE440000 |
| 26 | 1 | 29 | 0 | | 1101 | C12B | 0084C455 | 3 | 84 | | 02E60000 | 0 | 3B | FC | BE440000 |
| 27 | 0 | 27 | 0 | | 1401 | 8808 | 000622A8 | 8 | 84 | | 1737E000 | 5 | FC | | 70D01800 |
| 28 | 1 | 28 | 0 | | 1201 | E80E | 000C6D52 | 7 | 84 | | 066DC000 | 4 | FC | | E1A03000 |
| 29 | 0 | 26 | 0 | | 1601 | 9008 | 00636A90 | 4 | 84 | | 336E0000 | 1 | FC | | 5C998000 |
| 30 | 0 | 27 | 0 | | 1401 | F40E | 00C70122 | 3 | 84 | | 3ADA0000 | 0 | FC | | B9330000 |
| 31 | 0 | 27 | 0 | | 1401 | E00D | 00C71523 | 3 | 84 | | 26D90000 | 0 | FC | | B9330000 |
| 32 | 1 | 27 | 0 | | 1401 | CC0C | 00C72924 | 3 | C7 | 84 | 12D80000 | 0 | FC | E1 | B9330000 |
| 33 | 0 | 25 | 0 | | 1801 | A008 | 00014920 | 8 | C7 | | 96C70800 | 5 | E1 | | 0940F000 |
| 34 | 0 | 25 | 0 | | 1801 | 8807 | 00016121 | 8 | C7 | | 7EC60800 | 5 | E1 | | 0940F000 |
| 35 | 0 | 26 | 0 | | 1601 | E00C | 0002F244 | 7 | C7 | | CD8A1000 | 4 | E1 | | 1281E000 |
| 36 | 0 | 26 | 0 | | 1601 | CA0B | 00030845 | 7 | C7 | | B7891000 | 4 | E1 | | 1281E000 |
| 37 | 0 | 26 | 0 | | 1601 | B40A | 00031E46 | 7 | C7 | | A1881000 | 4 | E1 | | 1281E000 |
| 38 | 0 | 26 | 0 | | 1601 | 9E09 | 00033447 | 7 | C7 | | 8B871000 | 4 | E1 | | 1281E000 |
| 39 | 0 | 26 | 0 | | 1601 | 8808 | 00034A48 | 7 | C7 | | 75861000 | 4 | E1 | | 1281E000 |
| 40 | 0 | 27 | 0 | | 1401 | E40E | 0006C092 | 6 | C7 | | BF0A2000 | 3 | E1 | | 2503C000 |
| 41 | 0 | 27 | 0 | | 1401 | D00D | 0006D493 | 6 | C7 | | AB092000 | 3 | E1 | | 2503C000 |
| 42 | 0 | 27 | 0 | | 1401 | BC0C | 0006E894 | 6 | C7 | | 97082000 | 3 | E1 | | 2503C000 |
| 43 | 0 | 27 | 0 | | 1401 | A80B | 0006FC95 | 6 | C7 | | 83072000 | 3 | E1 | | 2503C000 |
| 44 | 0 | 27 | 0 | | 1401 | 940A | 00071096 | 6 | C7 | | 6F062000 | 3 | E1 | | 2503C000 |
| 45 | 0 | 27 | 0 | | 1401 | 8009 | 00072497 | 6 | C7 | | 5B052000 | 3 | E1 | | 2503C000 |
| 46 | 0 | 28 | 0 | | 1201 | D810 | 000E7130 | 5 | C7 | | 8E084000 | 2 | E1 | | 4A078000 |
| 47 | 0 | 28 | 0 | | 1201 | C60F | 000E8331 | 5 | C7 | | 7C074000 | 2 | E1 | | 4A078000 |
| 48 | 0 | 28 | 0 | | 1201 | B40E | 000E9532 | 5 | C7 | | 6A064000 | 2 | E1 | | 4A078000 |
| 49 | 0 | 28 | 0 | | 1201 | A20D | 000EA733 | 5 | C7 | | 58054000 | 2 | E1 | | 4A078000 |
| 50 | 0 | 28 | 0 | | 1201 | 900C | 000EB934 | 5 | C7 | | 46044000 | 2 | E1 | | 4A078000 |
| 51 | 0 | 29 | 0 | | 1101 | FC16 | 001D966A | 4 | C7 | | 68068000 | 1 | E1 | | 940F0000 |
| 52 | 0 | 29 | 0 | | 1101 | EB15 | 001DA76B | 4 | C7 | | 57058000 | 1 | E1 | | 940F0000 |
| 53 | 0 | 29 | 0 | | 1101 | DA14 | 001DB86C | 4 | C7 | | 46048000 | 1 | E1 | | 940F0000 |
| 54 | 0 | 29 | 0 | | 1101 | C913 | 001DC96D | 4 | C7 | | 35038000 | 1 | E1 | | 940F0000 |
| 55 | 0 | 29 | 0 | | 1101 | B812 | 001DDA6E | 4 | C7 | | 24028000 | 1 | E1 | | 940F0000 |
| 56 | 0 | 29 | 0 | | 1101 | A711 | 001DEB6F | 4 | C7 | | 13018000 | 1 | E1 | | 940F0000 |
| 57 | 1 | 29 | 0 | | 1101 | 9610 | 001DFC70 | 4 | C7 | | 02008000 | 1 | E1 | A1 | 940F0000 |
| 58 | 1 | 27 | 0 | | 1401 | 8808 | 00EFE380 | 1 | 3B | C7 | 10068400 | 6 | A1 | | 78017800 |
| 59 | 0 | 25 | 0 | | 1801 | A008 | 001F1C00 | 6 | 3B | | 80342000 | 3 | A1 | | 1FD3C000 |
| 60 | 0 | 25 | 0 | | 1801 | 8807 | 001F3401 | 6 | 3B | | 68332000 | 3 | A1 | | 1FD3C000 |
| 61 | 0 | 26 | 0 | | 1601 | E00C | 003E9804 | 5 | 3B | | A0644000 | 2 | A1 | | 3FA78000 |
| 62 | 0 | 26 | 0 | | 1601 | CA0B | 003EAE05 | 5 | 3B | | 8A634000 | 2 | A1 | | 3FA78000 |
| 63 | 0 | 26 | 0 | | 1601 | B40A | 003EC406 | 5 | 3B | | 74624000 | 2 | A1 | | 3FA78000 |
| 64 | 0 | 26 | 0 | | 1601 | 9E09 | 003EDA07 | 5 | 3B | | 5E614000 | 2 | A1 | | 3FA78000 |

**Table H.1 – Encoder and decoder trace data** *(continued)*

| Common | | | | | | | Encoder | | | | Decoder | | | | Software Conventions Decoder |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EC | D | I | MPS | CE | Qe hex | A hex | C hex | CT | B hex | OUT hex | C hex | CT | B hex | IN hex | C hex |
| 65 | 0 | 26 | 0 | | 1601 | 8808 | 003EF008 | 5 | 3B | | 48604000 | 2 | A1 | | 3FA78000 |
| 66 | 0 | 27 | 0 | | 1401 | E40E | 007E0C12 | 4 | 3B | | 64BE8000 | 1 | A1 | | 7F4F0000 |
| 67 | 0 | 27 | 0 | | 1401 | D00D | 007E2013 | 4 | 3B | | 50BD8000 | 1 | A1 | | 7F4F0000 |
| 68 | 0 | 27 | 0 | | 1401 | BC0C | 007E3414 | 4 | 3B | | 3CBC8000 | 1 | A1 | | 7F4F0000 |
| 69 | 0 | 27 | 0 | | 1401 | A80B | 007E4815 | 4 | 3B | | 28BB8000 | 1 | A1 | | 7F4F0000 |
| 70 | 0 | 27 | 0 | | 1401 | 940A | 007E5C16 | 4 | 3B | | 14BA8000 | 1 | A1 | | 7F4F0000 |
| 71 | 1 | 27 | 0 | | 1401 | 8009 | 007E7017 | 4 | 3B | | 00B98000 | 1 | A1 | 43 | 7F4F0000 |
| 72 | 1 | 25 | 0 | | 1801 | A008 | 03F380B8 | 1 | FC | 3B | 05CD0C00 | 6 | 43 | | 9A3AF000 |
| 73 | 0 | 23 | 0 | | 2201 | C008 | 001C05C0 | 6 | FC | | 2E686000 | 3 | 43 | | 919F8000 |
| 74 | 1 | 23 | 0 | | 2201 | 9E07 | 001C27C1 | 6 | FC | | 0C676000 | 3 | 43 | | 919F8000 |
| 75 | 0 | 21 | 0 | | 2801 | 8804 | 00709F04 | 4 | FC | | 319D8000 | 1 | 43 | | 56660000 |
| 76 | 1 | 22 | 0 | | 2401 | C006 | 00E18E0A | 3 | FC | | 13390000 | 0 | 43 | 04 | ACCC0000 |
| 77 | 0 | 20 | 0 | | 3001 | 9004 | 03863828 | 1 | E1 | FC | 4CE41000 | 6 | 04 | | 431FEC00 |
| 78 | 0 | 21 | 0 | | 2801 | C006 | 0004D052 | 8 | E1 | | 39C62000 | 5 | 04 | | 863FD800 |
| 79 | 1 | 21 | 0 | | 2801 | 9805 | 0004F853 | 8 | E1 | | 11C52000 | 5 | 04 | | 863FD800 |
| 80 | 0 | 19 | 0 | | 3401 | A004 | 0013E14C | 6 | E1 | | 47148000 | 3 | 04 | | 58EF6000 |
| 81 | 1 | 20 | 0 | | 3001 | D806 | 00282A9A | 5 | E1 | | 26270000 | 2 | 04 | | B1DEC000 |
| 82 | 0 | 19 | 0 | | 3401 | C004 | 00A0AA68 | 3 | E1 | | 989C0000 | 0 | 04 | | 27670000 |
| 83 | 0 | 19 | 0 | | 3401 | 8C03 | 00A0DE69 | 3 | E1 | | 649B0000 | 0 | 04 | 02 | 27670000 |
| 84 | 0 | 20 | 0 | | 3001 | B004 | 014224D4 | 2 | E1 | | 61340400 | 7 | 02 | | 4ECFFA00 |
| 85 | 0 | 20 | 0 | | 3001 | 8003 | 014254D5 | 2 | E1 | | 31330400 | 7 | 02 | | 4ECFFA00 |
| 86 | 1 | 21 | 0 | | 2801 | A004 | 028509AC | 1 | A1 | E1 | 02640800 | 6 | 02 | | 9D9FF400 |
| 87 | 1 | 19 | 0 | | 3401 | A004 | 000426B0 | 7 | A1 | | 09902000 | 4 | 02 | | 9673D000 |
| 88 | 1 | 18 | 0 | | 3801 | D004 | 00109AC0 | 5 | A1 | | 26408000 | 2 | 02 | | A9C34000 |
| 89 | 0 | 17 | 0 | | 4801 | E004 | 00426B00 | 3 | A1 | | 99020000 | 0 | 02 | | 47010000 |
| 90 | 0 | 17 | 0 | | 4801 | 9803 | 0042B301 | 3 | A1 | | 51010000 | 0 | 02 | 20 | 47010000 |
| 91 | 1 | 18 | 0 | | 3801 | A004 | 0085F604 | 2 | 42 | A1 | 12004000 | 7 | 20 | | 8E03BE00 |
| 92 | 0 | 17 | 0 | | 4801 | E004 | 0007D810 | 8 | 42 | | 48010000 | 5 | 20 | | 9802F800 |
| 93 | 1 | 17 | 0 | | 4801 | 9803 | 00082011 | 8 | 42 | | 00000000 | 5 | 20 | | 9802F800 |
| 94 | 0 | 16 | 0 | X | 5101 | 9002 | 00104022 | 7 | 42 | | 00000000 | 4 | 20 | | 9001F000 |
| 95 | 1 | 17 | 0 | | 4801 | A202 | 00208044 | 6 | 42 | | 00000000 | 3 | 20 | | A201E000 |
| 96 | 0 | 16 | 0 | X | 5101 | 9002 | 00410088 | 5 | 42 | | 00000000 | 2 | 20 | | 9001C000 |
| 97 | 1 | 17 | 0 | | 4801 | A202 | 00820110 | 4 | 42 | | 00000000 | 1 | 20 | | A2018000 |
| 98 | 0 | 16 | 0 | X | 5101 | 9002 | 01040220 | 3 | 42 | | 00000000 | 0 | 20 | 00 | 90010000 |
| 99 | 1 | 17 | 0 | | 4801 | A202 | 02080440 | 2 | 42 | | 00000000 | 7 | 00 | | A201FE00 |
| 100 | 0 | 16 | 0 | X | 5101 | 9002 | 04100880 | 1 | 04 | 43 | 00000000 | 6 | 00 | | 9001FC00 |
| 101 | 1 | 17 | 0 | | 4801 | A202 | 00001100 | 8 | 04 | | 00000000 | 5 | 00 | | A201F800 |
| 102 | 0 | 16 | 0 | X | 5101 | 9002 | 00002200 | 7 | 04 | | 00000000 | 4 | 00 | | 9001F000 |
| 103 | 1 | 17 | 0 | | 4801 | A202 | 00004400 | 6 | 04 | | 00000000 | 3 | 00 | | A201E000 |
| 104 | 0 | 16 | 0 | X | 5101 | 9002 | 00008800 | 5 | 04 | | 00000000 | 2 | 00 | | 9001C000 |
| 105 | 1 | 17 | 0 | | 4801 | A202 | 00011000 | 4 | 04 | | 00000000 | 1 | 00 | | A2018000 |
| 106 | 0 | 16 | 0 | X | 5101 | 9002 | 00022000 | 3 | 04 | | 00000000 | 0 | 00 | | 90010000 |
| 107 | 1 | 17 | 0 | | 4801 | A202 | 00044000 | 2 | 04 | | 00000000 | 7 | 00 | | A201FE00 |
| 108 | 0 | 16 | 0 | X | 5101 | 9002 | 00088000 | 1 | 02 | 04 | 00000000 | 6 | 00 | | 9001FC00 |
| 109 | 1 | 17 | 0 | | 4801 | A202 | 00010000 | 8 | 02 | | 00000000 | 5 | 00 | | A201F800 |
| 110 | 0 | 16 | 0 | X | 5101 | 9002 | 00020000 | 7 | 02 | | 00000000 | 4 | 00 | | 9001F000 |
| 111 | 1 | 17 | 0 | | 4801 | A202 | 00040000 | 6 | 02 | | 00000000 | 3 | 00 | | A201E000 |
| 112 | 0 | 16 | 0 | X | 5101 | 9002 | 00080000 | 5 | 02 | | 00000000 | 2 | 00 | | 9001C000 |
| 113 | 1 | 17 | 0 | | 4801 | A202 | 00100000 | 4 | 02 | | 00000000 | 1 | 00 | | A2018000 |
| 114 | 0 | 16 | 0 | X | 5101 | 9002 | 00200000 | 3 | 02 | | 00000000 | 0 | 00 | 41 | 90010000 |
| 115 | 1 | 17 | 0 | | 4801 | A202 | 00400000 | 2 | 02 | | 00008200 | 7 | 41 | | A2017C00 |
| 116 | 0 | 16 | 0 | X | 5101 | 9002 | 00800000 | 1 | 20 | 02 | 00010400 | 6 | 41 | | 9000F800 |
| 117 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 8 | 20 | | 00020800 | 5 | 41 | | A1FFF000 |
| 118 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 7 | 20 | | 00041000 | 4 | 41 | | 8FFDE000 |
| 119 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 6 | 20 | | 00082000 | 3 | 41 | | A1F9C000 |
| 120 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 5 | 20 | | 00104000 | 2 | 41 | | 8FF18000 |
| 121 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 4 | 20 | | 00208000 | 1 | 41 | | A1E10000 |
| 122 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 3 | 20 | | 00410000 | 0 | 41 | 0D | 8FC00000 |
| 123 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 2 | 20 | | 00821A00 | 7 | 0D | | A17FE400 |
| 124 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 1 | 00 | 20 | 01043400 | 6 | 0D | | 8EFDC800 |
| 125 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 8 | 00 | | 02086800 | 5 | 0D | | 9FF99000 |
| 126 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 7 | 00 | | 0410D000 | 4 | 0D | | 8BF12000 |
| 127 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 6 | 00 | | 0821A000 | 3 | 0D | | 99E04000 |
| 128 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 5 | 00 | | 10434000 | 2 | 0D | | 7FBE8000 |

**Table H.1 – Encoder and decoder trace data** *(continued)*

| Common | | | | | | | Encoder | | | | Decoder | | | | Software Conventions Decoder |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EC | D | I | MPS | CE | Qe hex | A hex | C hex | CT | B hex | OUT hex | C hex | CT | B hex | IN hex | C hex |
| 129 | 1 | 17 | 0 | | 4801 | A202 | 00000000 | 4 | 00 | | 20868000 | 1 | 0D | | 817B0000 |
| 130 | 0 | 16 | 0 | X | 5101 | 9002 | 00000000 | 3 | 00 | | 410D0000 | 0 | 0D | BB | 4EF40000 |
| 131 | 0 | 17 | 0 | | 4801 | A202 | 00000000 | 2 | 00 | | 821B7600 | 7 | BB | | 1FE68800 |
| 132 | 0 | 18 | 0 | | 3801 | B402 | 00009002 | 1 | 00 | 00 | 7434EC00 | 6 | BB | | 3FCD1000 |
| 133 | 0 | 19 | 0 | | 3401 | F802 | 00019006 | 8 | 00 | | 7867D800 | 5 | BB | | 7F9A2000 |
| 134 | 0 | 19 | 0 | | 3401 | C401 | 0001C407 | 8 | 00 | | 4466D800 | 5 | BB | | 7F9A2000 |
| 135 | 1 | 19 | 0 | | 3401 | 9000 | 0001F808 | 8 | 00 | | 1065D800 | 5 | BB | | 7F9A2000 |
| 136 | 0 | 18 | 0 | | 3801 | D004 | 0007E020 | 6 | 00 | | 41976000 | 3 | BB | | 8E6C8000 |
| 137 | 1 | 18 | 0 | | 3801 | 9803 | 00081821 | 6 | 00 | | 09966000 | 3 | BB | | 8E6C8000 |
| 138 | 1 | 17 | 0 | | 4801 | E004 | 00206084 | 4 | 00 | | 26598000 | 1 | BB | | B9AA0000 |
| 139 | 0 | 16 | 0 | X | 5101 | 9002 | 0040C108 | 3 | 00 | | 4CB30000 | 0 | BB | 86 | 434E0000 |
| 140 | 0 | 17 | 0 | | 4801 | A202 | 00818210 | 2 | 00 | | 99670C00 | 7 | 86 | | 089AF200 |
| 141 | 0 | 18 | 0 | | 3801 | B402 | 01039422 | 1 | 40 | 00 | A2CC1800 | 6 | 86 | | 1135E400 |
| 142 | 0 | 19 | 0 | | 3401 | F802 | 00079846 | 8 | 40 | | D5963000 | 5 | 86 | | 226BC800 |
| 143 | 0 | 19 | 0 | | 3401 | C401 | 0007CC47 | 8 | 40 | | A1953000 | 5 | 86 | | 226BC800 |
| 144 | 0 | 19 | 0 | | 3401 | 9000 | 00080048 | 8 | 40 | | 6D943000 | 5 | 86 | | 226BC800 |
| 145 | 0 | 20 | 0 | | 3001 | B7FE | 00106892 | 7 | 40 | | 73266000 | 4 | 86 | | 44D79000 |
| 146 | 0 | 20 | 0 | | 3001 | 87FD | 00109893 | 7 | 40 | | 43256000 | 4 | 86 | | 44D79000 |
| 147 | 1 | 21 | 0 | | 2801 | AFF8 | 00219128 | 6 | 40 | | 2648C000 | 3 | 86 | | 89AF2000 |
| 148 | 0 | 19 | 0 | | 3401 | A004 | 008644A0 | 4 | 40 | | 99230000 | 1 | 86 | | 06E08000 |
| 149 | 0 | 20 | 0 | | 3001 | D806 | 010CF142 | 3 | 40 | | CA440000 | 0 | 86 | | 0DC10000 |
| 150 | 0 | 20 | 0 | | 3001 | A805 | 010D2143 | 3 | 40 | | 9A430000 | 0 | 86 | F4 | 0DC10000 |
| 151 | 0 | 21 | 0 | | 2801 | F008 | 021AA288 | 2 | 40 | | D485E800 | 7 | F4 | | 1B821600 |
| 152 | 0 | 21 | 0 | | 2801 | C807 | 021ACA89 | 2 | 40 | | AC84E800 | 7 | F4 | | 1B821600 |
| 153 | 0 | 21 | 0 | | 2801 | A006 | 021AF28A | 2 | 40 | | 8483E800 | 7 | F4 | | 1B821600 |
| 154 | 0 | 22 | 0 | | 2401 | F00A | 04363516 | 1 | 40 | | B905D000 | 6 | F4 | | 37042C00 |
| 155 | 0 | 22 | 0 | | 2401 | CC09 | 04365917 | 1 | 40 | | 9504D000 | 6 | F4 | | 37042C00 |
| 156 | 0 | 22 | 0 | | 2401 | A808 | 04367D18 | 1 | 40 | | 7103D000 | 6 | F4 | | 37042C00 |
| 157 | 0 | 22 | 0 | | 2401 | 8407 | 0436A119 | 1 | 0D | 41 | 4D02D000 | 6 | F4 | | 37042C00 |
| 158 | 0 | 23 | 0 | | 2201 | C00C | 00058A34 | 8 | 0D | | 5203A000 | 5 | F4 | | 6E085800 |
| 159 | 0 | 23 | 0 | | 2201 | 9E0B | 0005AC35 | 8 | 0D | | 3002A000 | 5 | F4 | | 6E085800 |
| 160 | 0 | 24 | 0 | | 1C01 | F814 | 000B9C6C | 7 | 0D | | 1C034000 | 4 | F4 | | DC10B000 |
| 161 | 1 | 24 | 0 | | 1C01 | DC13 | 000BB86D | 7 | 0D | | 00024000 | 4 | F4 | | DC10B000 |
| 162 | 1 | 22 | 0 | | 2401 | E008 | 005DC368 | 4 | 0D | | 00120000 | 1 | F4 | 31 | DFF58000 |
| 163 | 1 | 20 | 0 | | 3001 | 9004 | 01770DA0 | 2 | BB | 0D | 00486200 | 7 | 31 | | 8FBB9C00 |
| 164 | 1 | 19 | 0 | | 3401 | C004 | 00043680 | 8 | BB | | 01218800 | 5 | 31 | | BEE27000 |
| 165 | 1 | 18 | 0 | | 3801 | D004 | 0010DA00 | 6 | BB | | 04862000 | 3 | 31 | | CB7DC000 |
| 166 | 1 | 17 | 0 | | 4801 | E004 | 00436800 | 4 | BB | | 12188000 | 1 | 31 | | CDEB0000 |
| 167 | 0 | 16 | 0 | X | 5101 | 9002 | 0086D000 | 3 | BB | | 24310000 | 0 | 31 | 7F | 6BD00000 |
| 168 | 0 | 17 | 0 | | 4801 | A202 | 010DA000 | 2 | BB | | 4862FE00 | 7 | 7F | | 599F0000 |
| 169 | 1 | 18 | 0 | | 3801 | B402 | 021BD002 | 1 | 86 | BB | 00C3FC00 | 6 | 7F | | B33E0000 |
| 170 | 1 | 17 | 0 | | 4801 | E004 | 000F4008 | 7 | 86 | | 030FF000 | 4 | 7F | | DCF40000 |
| 171 | 0 | 16 | 0 | X | 5101 | 9002 | 001E8010 | 6 | 86 | | 061FE000 | 3 | 7F | | 89E20000 |
| 172 | 1 | 17 | 0 | | 4801 | A202 | 003D0020 | 5 | 86 | | 0C3FC000 | 2 | 7F | | 95C20000 |
| 173 | 0 | 16 | 0 | X | 5101 | 9002 | 007A0040 | 4 | 86 | | 187F8000 | 1 | 7F | | 77820000 |
| 174 | 1 | 17 | 0 | | 4801 | A202 | 00F40080 | 3 | 86 | | 30FF0000 | 0 | 7F | FF | 71020000 |
| 175 | 1 | 16 | 0 | X | 5101 | 9002 | 01E80100 | 2 | F4 | 86 | 61FFFE00 | 7 | FF | | 2E020000 |
| 176 | 1 | 15 | 0 | | 5401 | FC04 | 00014804 | 8 | F4 | | 43FBF800 | 5 | FF | | B8080000 |
| 177 | 1 | 14 | 0 | X | 5601 | A802 | 00029008 | 7 | F4 | | 87F7F000 | 4 | FF | | 200A0000 |
| 178 | 0 | 14 | 1 | X | 5601 | A402 | 0005CC12 | 6 | F4 | | 63EDE000 | 3 | FF | | 40140000 |
| 179 | 0 | 14 | 0 | X | 5601 | 9C02 | 000C4426 | 5 | F4 | | 1BD9C000 | 2 | FF | | 80280000 |
| 180 | 1 | 15 | 0 | | 5401 | AC02 | 0018884C | 4 | F4 | | 37B38000 | 1 | FF | | 744E0000 |
| 181 | 1 | 14 | 0 | X | 5601 | A802 | 00311098 | 3 | F4 | | 6F670000 | 0 | FF | 88 | 389A0000 |
| 182 | 1 | 14 | 1 | X | 5601 | A402 | 0062CD32 | 2 | F4 | | 32CE2000 | 6 | 88 | | 7133DC00 |
| 183 | 1 | 15 | 1 | | 5401 | AC02 | 00C59A64 | 1 | 31 | F4 | 659C4000 | 5 | 88 | | 4665B800 |
| 184 | 0 | 16 | 1 | | 5101 | B002 | 0003DCCA | 8 | 31 | | 23368000 | 4 | 88 | | 8CCB7000 |
| 185 | 1 | 15 | 1 | X | 5401 | A202 | 0007B994 | 7 | 31 | | 466D0000 | 3 | 88 | | 5B94E000 |
| 186 | 1 | 16 | 1 | | 5101 | A802 | 000F7328 | 6 | 31 | | 8CDA0000 | 2 | 88 | | 1B27C000 |
| 187 | 1 | 17 | 1 | | 4801 | AE02 | 001F8852 | 5 | 31 | | 77B20000 | 1 | 88 | | 364F8000 |
| 188 | 1 | 18 | 1 | | 3801 | CC02 | 003FA0A6 | 4 | 31 | | 5F620000 | 0 | 88 | | 6C9F0000 |
| 189 | 0 | 18 | 1 | | 3801 | 9401 | 003FD8A7 | 4 | 31 | | 27610000 | 0 | 88 | FF | 6C9F0000 |
| 190 | 1 | 17 | 1 | | 4801 | E004 | 00FF629C | 2 | 31 | | 9D87FC00 | 6 | FF | | 427C0000 |
| 191 | 1 | 17 | 1 | | 4801 | 9803 | 00FFAA9D | 2 | 31 | | 5586FC00 | 6 | FF | | 427C0000 |
| 192 | 0 | 18 | 1 | | 3801 | A004 | 01FFE53C | 1 | 7F | 31 | 1B0BF800 | 5 | FF | | 84F80000 |

### Table H.1 – Encoder and decoder trace data *(concluded)*

| Common | | | | | | | Encoder | | | | Decoder | | | | Software Conventions Decoder |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| EC | D | I | MPS | CE | Qe hex | A hex | C hex | CT | B hex | OUT hex | C hex | CT | B hex | IN hex | C hex |
| 193 | 1 | 17 | 1 | | 4801 | E004 | 000F94F0 | 7 | 7F | | 6C2FE000 | 3 | FF | | 73D40000 |
| 194 | 0 | 17 | 1 | | 4801 | 9803 | 000FDCF1 | 7 | 7F | | 242EE000 | 3 | FF | | 73D40000 |
| 195 | 1 | 16 | 1 | X | 5101 | 9002 | 001FB9E2 | 6 | 7F | | 485DC000 | 2 | FF | | 47A40000 |
| 196 | 1 | 17 | 1 | | 4801 | A202 | 003F73C4 | 5 | 7F | | 90BB8000 | 1 | FF | | 11460000 |
| 197 | 1 | 18 | 1 | | 3801 | B402 | 007F778A | 4 | 7F | | 91750000 | 0 | FF | 37 | 228C0000 |
| 198 | 1 | 19 | 1 | | 3401 | F802 | 00FF5F16 | 3 | 7F | | B2E8DC00 | 6 | 37 | | 45192000 |
| 199 | 1 | 19 | 1 | | 3401 | C401 | 00FF9317 | 3 | 7F | | 7EE7DC00 | 6 | 37 | | 45192000 |
| 200 | 1 | 19 | 1 | | 3401 | 9000 | 00FFC718 | 3 | 7F | | 4AE6DC00 | 6 | 37 | | 45192000 |
| 201 | 0 | 20 | 1 | | 3001 | B7FE | 01FFF632 | 2 | FF | 7F | 2DCBB800 | 5 | 37 | | 8A324000 |
| 202 | 1 | 19 | 1 | | 3401 | C004 | 0007D8C8 | 8 | FF | | B72EE000 | 3 | 37 | | 08D50000 |
| 203 | 1 | 19 | 1 | | 3401 | 8C03 | 00080CC9 | 8 | FF | | 832DE000 | 3 | 37 | | 08D50000 |
| 204 | 1 | 20 | 1 | | 3001 | B004 | 00108194 | 7 | FF | | 9E59C000 | 2 | 37 | | 11AA0000 |
| 205 | 1 | 20 | 1 | | 3001 | 8003 | 0010B195 | 7 | FF | | 6E58C000 | 2 | 37 | | 11AA0000 |
| 206 | 1 | 21 | 1 | | 2801 | A004 | 0021C32C | 6 | FF | | 7CAF8000 | 1 | 37 | | 23540000 |
| 207 | 1 | 22 | 1 | | 2401 | F006 | 0043D65A | 5 | FF | | A95D0000 | 0 | 37 | | 46A80000 |
| 208 | 1 | 22 | 1 | | 2401 | CC05 | 0043FA5B | 5 | FF | | 855C0000 | 0 | 37 | | 46A80000 |
| 209 | 1 | 22 | 1 | | 2401 | A804 | 00441E5C | 5 | FF | | 615B0000 | 0 | 37 | | 46A80000 |
| 210 | 1 | 22 | 1 | | 2401 | 8403 | 0044425D | 5 | FF | | 3D5A0000 | 0 | 37 | 47 | 46A80000 |
| 211 | 1 | 23 | 1 | | 2201 | C004 | 0088CCBC | 4 | FF | | 32B28E00 | 7 | 47 | | 8D517000 |
| 212 | 0 | 23 | 1 | | 2201 | 9E03 | 0088EEBD | 4 | FF | | 10B18E00 | 7 | 47 | | 8D517000 |
| 213 | 1 | 21 | 1 | | 2801 | 8804 | 0223BAF4 | 2 | FF | | 42C63800 | 5 | 47 | | 453DC000 |
| 214 | 1 | 22 | 1 | | 2401 | C006 | 0447C5EA | 1 | FF | | 358A7000 | 4 | 47 | | 8A7B8000 |
| 215 | 0 | 22 | 1 | | 2401 | 9C05 | 0447E9EB | 1 | 88 | FF | 11897000 | 4 | 47 | | 8A7B8000 |
| 216 | 1 | 20 | 1 | | 3001 | 9004 | 001FA7AC | 6 | 88 | | 4625C000 | 2 | 47 | | 49DE0000 |
| 217 | 1 | 21 | 1 | | 2801 | C006 | 003FAF5A | 5 | 88 | | 2C498000 | 1 | 47 | | 93BC0000 |
| 218 | 0 | 21 | 1 | | 2801 | 9805 | 003FD75B | 5 | 88 | | 04488000 | 1 | 47 | 1A | 93BC0000 |
| 219 | 0 | 19 | 1 | | 3401 | A004 | 00FF5D6C | 3 | 88 | | 11223400 | 7 | 1A | | 8EE1CA00 |
| 220 | 1 | 18 | 1 | | 3801 | D004 | 03FD75B0 | 1 | 88 | | 4488D000 | 5 | 1A | | 8B7B2800 |
| 221 | 0 | 18 | 1 | | 3801 | 9803 | 03FDADB1 | 1 | FF | 88 | 0C87D000 | 5 | 1A | | 8B7B2800 |
| 222 | 0 | 17 | 1 | | 4801 | E004 | 0006B6C4 | 7 | FF | | 321F4000 | 3 | 1A | | ADE4A000 |
| 223 | 0 | 16 | 1 | X | 5101 | 9002 | 000D6D88 | 6 | FF | | 643E8000 | 2 | 1A | | 2BC34000 |
| 224 | 0 | 15 | 1 | | 5401 | FC04 | 0036FA24 | 4 | FF | | 4CF60000 | 0 | 1A | DB | AF0D0000 |
| 225 | 0 | 14 | 1 | X | 5601 | A802 | 006DF448 | 3 | FF | | 99EDB600 | 7 | DB | | 0E144800 |
| 226 | 1 | 14 | 0 | X | 5601 | A402 | 00DC9492 | 2 | FF | | 87D96C00 | 6 | DB | | 1C289000 |
| 227 | 0 | 14 | 1 | X | 5601 | 9C02 | 01B9D526 | 1 | 37 | FF | 63B0D800 | 5 | DB | | 38512000 |
| 228 | 0 | 14 | 0 | X | 5601 | 8C02 | 0004564E | 7 | 37 | | 1B5FB000 | 4 | DB | | 70A24000 |
| 229 | 1 | 15 | 0 | | 5401 | AC02 | 0008AC9C | 6 | 37 | | 36BF6000 | 3 | DB | | 75428000 |
| 230 | 1 | 14 | 0 | X | 5601 | A802 | 00115938 | 5 | 37 | | 6D7EC000 | 2 | DB | | 3A830000 |
| 231 | 1 | 14 | 1 | X | 5601 | A402 | 00235E72 | 4 | 37 | | 2EFB8000 | 1 | DB | | 75060000 |
| 232 | 1 | 15 | 1 | | 5401 | AC02 | 0046BCE4 | 3 | 37 | | 5DF70000 | 0 | DB | 6A | 4E0A0000 |
| 233 | 0 | 16 | 1 | | 5101 | B002 | 008E21CA | 2 | 37 | | 13ECD400 | 7 | 6A | | 9C152A00 |
| 234 | 1 | 15 | 1 | X | 5401 | A202 | 011C4394 | 1 | 47 | 37 | 27D9A800 | 6 | 6A | | 7A285400 |
| 235 | 0 | 16 | 1 | | 5101 | A802 | 00008728 | 8 | 47 | | 4FB35000 | 5 | 6A | | 584EA800 |
| 236 | 0 | 15 | 1 | X | 5401 | A202 | 00010E50 | 7 | 47 | | 9F66A000 | 4 | 6A | | 029B5000 |
| 237 | 0 | 14 | 1 | X | 5601 | 9C02 | 0002C4A2 | 6 | 47 | | 96CB4000 | 3 | 6A | | 0536A000 |
| 238 | 1 | 14 | 0 | X | 5601 | 8C02 | 00063546 | 5 | 47 | | 81948000 | 2 | 6A | | 0A6D4000 |
| 239 | 1 | 14 | 1 | | 5601 | D804 | 001A2D1C | 3 | 47 | | AE4E0000 | 0 | 6A | | 29B50000 |
| 240 | 0 | 14 | 1 | X | 5601 | 8203 | 001A831D | 3 | 47 | | 584D0000 | 0 | 6A | DF | 29B50000 |
| 241 | 1 | 14 | 0 | | 5601 | B008 | 006B6478 | 1 | 1A | 47 | 09337C00 | 6 | DF | | A6D48000 |
| 242 | 0 | 14 | 1 | | 5601 | AC02 | 0006C8F0 | 8 | 1A | | 1266F800 | 5 | DF | | 999B0000 |
| 243 | 1 | 14 | 0 | | 5601 | AC02 | 000D91E0 | 7 | 1A | | 24CDF000 | 4 | DF | | 87340000 |
| 244 | 0 | 14 | 1 | | 5601 | AC02 | 001B23C0 | 6 | 1A | | 499BE000 | 3 | DF | | 62660000 |
| 245 | 0 | 14 | 0 | | 5601 | AC02 | 00364780 | 5 | 1A | | 9337C000 | 2 | DF | | 18CA0000 |
| 246 | 0 | 15 | 0 | | 5401 | AC02 | 006D3B02 | 4 | 1A | | 7A6D8000 | 1 | DF | | 31940000 |
| 247 | 1 | 16 | 0 | | 5101 | B002 | 00DB1E06 | 3 | 1A | | 4CD90000 | 0 | DF | FF | 63280000 |
| 248 | 1 | 15 | 0 | X | 5401 | A202 | 01B63C0C | 2 | 1A | | 99B3FE00 | 7 | FF | | 084E0000 |
| 249 | 1 | 14 | 0 | X | 5601 | 9C02 | 036D201A | 1 | DB | 1A | 8B65FC00 | 6 | FF | | 109C0000 |
| 250 | 0 | 14 | 1 | X | 5601 | 8C02 | 0002EC36 | 8 | DB | | 6AC9F800 | 5 | FF | | 21380000 |
| 251 | 1 | 14 | 0 | | 5601 | D804 | 000D08DC | 6 | DB | | 5323E000 | 3 | FF | | 84E00000 |
| 252 | 1 | 14 | 1 | | 5601 | AC02 | 001A11B8 | 5 | DB | | A647C000 | 2 | FF | | 05BA0000 |
| 253 | 1 | 15 | 1 | | 5401 | AC02 | 0034CF72 | 4 | DB | | A08D8000 | 1 | FF | | 0B740000 |
| 254 | 1 | 16 | 1 | | 5101 | B002 | 006A46E6 | 3 | DB | | 99190000 | 0 | FF | | 16E80000 |
| 255 | 1 | 17 | 1 | | 4801 | BE02 | 00D52FCE | 2 | DB | | 9031FE00 | 7 | FF | | 2DD00000 |
| 256 | 1 | 18 | 1 | | 3801 | EC02 | 01AAEF9E | 1 | DB | | 9061FC00 | 6 | FF | | 5BA00000 |
| 257 | | | | | | | | | | DB 6A DF FF AC | | | | | |

## Bibliography


[1] ASCHER (R. N.), NAGY (G.): 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] BILGIN (A.), SEMENTILLI (P.), MARCELLIN (M.): Progressive image coding using trellis coded quantization, *IEEE Transactions on Image Processing*, November 1997. (submitted).

[3] CONSTANTINESCU (C.), ARPS (R.): Fast residue coding for lossless textual image compression, in *Proc. of Data Compression Conference*, pages 397-406, Snowbird, Utah, USA, 1997.

[4] FORCHHAMMER (S.), JENSEN (K. S.): Data compression of scanned halftone images, *IEEE Trans. Commun.*, 42:1881-1893, Feb.-April 1994.

[5] HOWARD (P. G.): Lossless and lossy compression of text images by soft pattern matching, in *Proc. of Data Compression Conference*, pages 210-219, 1996.

[6] HOWARD (P. G.): Text image compression using soft pattern matching, *Computer Journal*, 40:2-3, 1997.

[7] HOWARD (P. G.), KOSSENTINI (F.), MARTINS (B.), FORCHHAMMER (S.), RUCKLIDGE (W.J.): The emerging JBIG2 standard, *IEEE Transactions on Circuits and Systems for Video Technology*, 8(7):838-848, November 1998.

[8] HUNTER (R.), ROBINSON (A. H.): International digital facsimile coding standards, in *Proceedings of IEEE*, Volume 68, pages 854-867, July 1980.

[9] INGLIS (S.), WITTEN (I. H.): Compression-based template matching, in J. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, 1994.

[10] KOSSENTINI (F.), WARD (R.): 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.

[11] MARTINS (B.), FORCHHAMMER (S.): 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] MARTINS (B.), FORCHHAMMER (S.): Halftone Coding with JBIG2, submitted to Journal of Electronic Imaging, September 1998.

[13] MARTINS (B.), FORCHHAMMER (S.): Tree Coding of Bi-level Images, *IEEE Trans. Image Proc.*, 7(4):517-528, 1998.

[14] MARTINS (B.), FORCHHAMMER (S.): Lossless, near-lossless, and refinement coding of bi-level images, *IEEE Trans. Image Processing*, 8(5):601-613, May 1999.

[15] MOHIUDDIN (K.), RISSANEN (J. J.), ARPS (R.): 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.

[16] PRATT (W. K.), CAPITANT (P. J.), CHEN (W. H.), HAMILTON (E. R.), WALLS (R. H.): Combined symbol matching facsimile data compression system, in *Proceedings of the IEEE*, Volume 68, pages 786-796, July 1980.

[17] SAID (A.), PEARLMAN (W. A): 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.

[18] SAYOOD (K.): *Introduction to Data Compression*, Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[19] TOU (J. T.), GONZALEZ (R. C.): *Pattern Recognition Principles*, Addison-Wesley Publishing Company, Reading, MA, 1974.

[20] WITTEN (I.H.), MOFFAT (A.), BELL (T.C.): *Managing Gigabytes*, Van Nostrand Reinhold, New York, 1994.

# ITU-T RECOMMENDATIONS SERIES

| | |
|---|---|
| Series A | Organization of the work of the ITU-T |
| Series B | Means of expression: definitions, symbols, classification |
| Series C | General telecommunication statistics |
| Series D | General tariff principles |
| Series E | Overall network operation, telephone service, service operation and human factors |
| Series F | Non-telephone telecommunication services |
| Series G | Transmission systems and media, digital systems and networks |
| Series H | Audiovisual and multimedia systems |
| Series I | Integrated services digital network |
| Series J | Transmission of television, sound programme and other multimedia signals |
| Series K | Protection against interference |
| Series L | Construction, installation and protection of cables and other elements of outside plant |
| Series M | TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits |
| Series N | Maintenance: international sound programme and television transmission circuits |
| Series O | Specifications of measuring equipment |
| Series P | Telephone transmission quality, telephone installations, local line networks |
| Series Q | Switching and signalling |
| Series R | Telegraph transmission |
| Series S | Telegraph services terminal equipment |
| **Series T** | **Terminals for telematic services** |
| Series U | Telegraph switching |
| Series V | Data communication over the telephone network |
| Series X | Data networks and open system communications |
| Series Y | Global information infrastructure and Internet protocol aspects |
| Series Z | Languages and general software aspects for telecommunication systems |

*18592*

Printed in Switzerland

Geneva, 2002