ISO/IEC JTC 1/SC 29/WG1 **N862**

Date: 1998-07-08

ISO / IEC JTC 1 / SC 29 / WG 1
(ITU-T SG8)

Coding of Still Pictures

JBIG
Joint Bi-level Image Experts Group

JPEG
Joint Photographic Experts Group

TITLE: **Coding of Numerical Data in JBIG-2**

SOURCE: **Dave Tompkins (davet@ece.ubc.ca)**
and Faouzi Kossentini (faouzi@ece.ubc.ca)
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver BC Canada V6T 1Z4

PROJECT: **JBIG-2**

STATUS: **Discussion**

REQUESTED ACTION: **For Inclusion in the JBIG-2 Standard**

DISTRIBUTION: **13th Meeting of ISO/IEC JTC1/SC 29/WG1**

Contact:
ISO/IEC JTC 1/SC 29/WG 1 Convener - Dr. Daniel T. Lee
Hewlett-Packard Company, 11000 Wolfe Road, MS42U0, Cupertion, California 95014, USA
Tel: +1 408 447 4160, Fax: +1 408 447 2842, E-mail: Daniel_Lee@hp.com

1. Summary

This document presents the results from implementing a variety of numerical coding methods within the JBIG-2 bitstream format to a specific encoder profile. The profile used to encode the data was fairly straightforward, and is described in section 2. Section 3 describes some of the issues involved with encoding numerical data in general. Section 4 describes each of the coding methods used in this experiment, and section 5 provides the results of the experiment for each numerical data type.

Overall, using an optimized Huffman table produced the best results, but not when the overhead of including the table size is considered. The multi-symbol Arithmetic coder used in this experiment performed very poorly. Depending on the data type, the standard Huffman tables produced good results, but in most circumstances the standard tables were outperformed by a Binary Arithmetic coder. Whether the Binary coder based its' codes upon the standard Huffman tables, or a another coding scheme, it seemed to be the best, and has the advantage of being compatible with the MQ bitstream.

Further Recommendations are included in Section 6.

2. Profile Description

The Profile used to generate the data for this experiment uses a simple and straightforward lossless soft pattern matcher, and is a component of the upcoming JBIG-2 Verification Model (VM). The data for this experiment was also generated for two additional profiles, but the relative performance of the different encoding methods was similar. It is hoped that this profile will represent a "typical" lossless implementation of JBIG-2 with SPM.

Here are some points of interest regarding the profile used:

- The image is not segmented into sub-pictures, and no halftone information is detected or used.
- The profile does not detect vertical text, and does not consider using a transposed coordinate system.
- Four segments are used to describe the document: Page Information Segment, Symbol Dictionary Segment, Symbol Image (instance) Segment, and a Generic Bitmap Segment (residue).
- Originally, all symbols are removed from the document and sorted by height and then width.
- All symbols smaller than 2x2 pixels are returned to the residue.
- All symbols are checked to see if a previous symbol of size no more than +/- 2 pixels has an 80% pixel match. If an appropriate match exists, the symbol is removed from the dictionary and is encoded only in the Instance segment, with refinement information.
- For aligning refinement images, only the default centroid method is used, with no offsets.
- Any symbols in the dictionary that only occur once, and are not used for refinement information by any other symbols are returned to the residue.
- Only one dictionary segment is used, so all of the symbols are exported. Furthermore, no symbols in the dictionary are constructed with refinement or aggregate information from previous symbols.
- In the Instance Segment, a stripe size of 1 pixel is used, and symbols are referenced by their lower-left coordinate.

3. General Encoding Issues with JBIG-2

In addition to compression efficiency, each encoding scheme should be evaluated on how easily it can be incorporated into the JBIG-2 bitstream format. In Refinement Symbol Image (Instance) segments this issue become even more pertinent, as integers will be intermixed with refinement data encoded with the MQ coder. If the bitstreams are incompatible, the data will have to be forced to byte boundaries, resulting in some inefficiency. If incompatible bitstreams are used, including the refinement information at the end of the segment should be considered as an alternative. However, adding refinement information at the end of the segment increases the bitstream complexity and increases the memory requirements for a decoder.

Huffman codes, or similar direct bit encoding can be incorporated directly into the MQ bitstream with the MQ Coder. The Coder can be forced to encode bits directly by setting the Most Probable Symbol to 0 (or 1) and the Probability to ~50% before encoding (or decoding) each bit. This method could also be used to encode data where using contexts have no effect, or in some cases when the arithmetic predictions can decrease efficiency. This method is illustrated in Figure 3.1.

```
int dummyMPS, dummyIndex;
…
for(each bit) {
     dummyMPS = 0;     /* force MPS to 0 */
     dummyIndex = 0;    /* force probability to ~50%
*/
     MQEncode(bit, &dummyIndex, &dummyMPS);
}
```

Figure 3.1 – Using the MQ Coder to encode bits directly

Arithmetic coders in general pose a problem with their memory requirements, especially if the coder is going to allow integers with 32-bit precision. It would be physically impractical to support a multi-symbol arithmetic coder that could properly handle enough symbols for 32 bit precision. If a multi-symbol approach is to be used, reasonable limits would have to be imposed on the range of numbers being encoded, and an alternative for encoding large numbers would have to be available. Binary coders do not escape this problem, and with variable length codes, the memory requirements to handle all the possible contexts would be even worse.

For Arithmetic Binary coders, *saturated* contexts can be used to reduce memory requirements. With saturated contexts, a limit is placed on the number of bits that can be used for contexts. When the length of context required exceeds the allowed context size, there are two possible solutions: 1) using a single context for all of the remaining bits to encode and 2) using the direct encoding scheme as mentioned above to encode the bits directly. The size of the context to be used can be specified in the header, similar to the method used to describe Huffman tables, or context templates in the bitmap segments. By leaving the context size flexible in the standard, it will allow for different profiles to exist with variable memory requirements.

4. Coding Methods

This section describes how each column of the data in Section 5 was calculated.

4.1 Entropy

This column represents the theoretical minimum number of bits per number (symbol) required. The value was calculated with the following equation:

$$\sum_{j}^{N} -\left(\frac{count(j)}{\sum count()}\right)\cdot \log_2\left(\frac{count(j)}{\sum count()}\right)$$

4.2 Huffman – Optimized Tables Method A

An Optimized Huffman Table was constructed for each data set, using the optimal code length for each number (symbol). In each table, it includes not just the length of the data, but in parenthesis it shows the length of the codes with the table size considered. The size of the table was determined according to the current bitstream requirements for defining a custom table. With the exception of the Symbol ID data [IAID], which used the run-length method in section 7.4.3.4 [1], each table was constructed according to the method described in section C.1 [1].

Note: *The Optimized Huffman Table in method A was constructed to minimize the code lengths,* ***NOT*** *to minimize the size of the table. For example, if the number 10 and 11 both have a code length of 7, it would reduce the table size to make their prefix code length 6 and have a range length of 1. However, to reduce complexity and to ensure the minimum code lengths, every number that occurred was assigned a range length of 0. Conversely, table entries for numbers that did not occur were minimized, and were grouped together with the largest possible range lengths, and given prefix lengths of 0.*

4.3 Huffman – Optimized Tables Method B

For each data type, there are several pre-defined standard Huffman tables available in the current JBIG-2 standard. For method B, these standardized tables were used, with prefix code lengths optimized for the data set. For each data set, the appropriate tables were considered, and then the table was selected which would minimize the length of the data stream.

4.4 Huffman – Optimized Tables Method C

An Optimized Huffman Table was constructed for each data set, using the method described in Appendix A. For each document, several tables and data streams were generated (with varying *thresholds*) and the method that generated the smallest overall bitstream size was used for the calculations.

4.5 Huffman – Standard Tables

The current standard includes some standard Huffman tables that are generally optimized for each data type. For most of the data types, there are several different tables available. The data contained in these charts represents the **minimum** of all the available tables.

For example: *to encode the subsequent Symbol A coordinate [IADA] any of tables C.8, C.9 and C.10 can be used. For document F01_300, Table C.8 was the optimum choice, and was chosen, but for*

document F01_600 Table C.9 was used. For each document, all of the available tables were utilized, and then the one with the highest compression was chosen

4.6 Arithmetic – Multi-Symbol Alphabet [Arithmetic MSA]

As described in Section 3, there are numerous issues involved with using a Multi-Symbol Arithmetic Coder. An existing generic library routine was used for the arithmetic coding. The routine was modified for each data type so that the number of symbols in the coder was lowered to the smallest power of 2 large enough to accommodate the data.

For example: The subsequent Symbol A coordinate data [IADA] contained data in the range –68..3669 so the coder used an alphabet of 4096 symbols.

4.7 Arithmetic – Binary (MQ) – Huffman Codes

This method encoded with the MQ Binary Arithmetic Coder, but used the Huffman codes from the standardized tables to represent each number as a stream of bits. A saturated context was used, as described in Section 3, with a single context for all the remaining bits. As in Section 4.4, the results were calculated for each of the available tables and the result from the optimum table was selected for each document.

4.8 Arithmetic – Binary (MQ) – Arithmetic Coders

As with Section 4.7, the number to be coded was represented as a binary code, and was encoded with the MQ Coder. However, instead of using the specific Huffman tables as the basis for the binary code, a specific table was used. In each case, the sign bit (s) was only used for data types where negative numbers are supported. In all tables, the value N is assumed to be the absolute value of the number being encoded.

4.8.1 Arithmetic Coder - Style A

Style A is very straightforward, and simply encodes each value with a fixed number of bits. For negative numbers, the value is encoded with a signed magnitude notation. The number of bits used for each data type would have to be included in the header segment, or for certain tables it could be fixed. In this experiment, the minimum possible data length was selected for each document. The OOB symbol is encoded as all 1's, so the number of bits required should take this into consideration.

For example, with 5 bits, and support for negative numbers, the following numbers would be encoded as:

OOB	111111
1	000001
–1	100001
-30	111110 (maximum negative number)

4.8.2 Arithmetic Coder - Style B

Style B is almost identical to Style A, with the exception of the OOB symbol. In Style B, the OOB is encoded as a 1, and a 0 prefixes all other numbers.
For example, with 5 bits, and support for negative numbers, the following numbers would be encoded as:

OOB	1
1	0000001
–1	0100001
-31	0111111 (maximum negative number)

4.8.3 Arithmetic Coder - Style C

To encode the Number N, the following table is used:

Prefix Bits	Meaning	Data Bits
1	OOB	
00	0	
01s0	+/- 1	
01s10	+/- 2	
01s110	1 bit number	N – 3
01s1110	2 bit number	N – 5
01s11110	4 bit number	N – 9
01s111110	6 bit number	N – 25
01s1111110	8 bit number	N – 89
01s11111110	12 bit number	N – 345
01s11111111	32 bit number	N – 4441

s=optional sign bit

Table 4.1 – Encoding Scheme for Arithmetic Coder Style C

Note: Special Thanks to William Rucklidge for suggesting this table structure.

4.8.4 Arithmetic Coder - Style D

To encode the Number N, the following table is used:

Prefix Bits	Meaning	Data Bits	Suffix Bit
111111	OOB		
0	2 bit number	N	s
10	4 bit number	N – 4	s
110	6 bit number	N – 20	s
1110	8 bit number	N – 84	s
11110	12 bit number	N – 340	s
111110	32 bit number	N – 4436	s

Table 4.2 – Encoding Scheme for Arithmetic Coder Style D

4.8.5 Arithmetic Coder - Style E

To encode the Number N, the following table is used:

Prefix Bits	Meaning	Data Bits
1	OOB	
0s0	2 bit number	N
0s10	4 bit number	N – 4
0s110	6 bit number	N – 20
0s1110	8 bit number	N – 84
0s11110	12 bit number	N – 340
0s11111	32 bit number	N – 4436

Table 4.3 – Encoding Scheme for Arithmetic Coder Style E

5. Numerical Data Types

This section summarizes the numerical results for each data type. Table 5.1 lists each Data Type, and a brief description.

Data	Description	Relevant Section(s)	Note
IADB	Strip Delta B	6.4.3 and 7.4.3.2.1	
IAFA	First Symbol A Coordinate	6.4.4 and 7.4.3.2.1	
IADA	Subsequent Symbol A Coordinate	6.4.5 and 7.4.3.2.1	
IAIB	Instance B Coordinate	6.4.6. and B.2	1.
IAID	Instance Symbol ID	6.4.7 and B.3	2.
IARI	Refinement Flag	6.4.8	
IARDH	Instance Refinement Delta Height	6.4.8.1 and 7.4.3.2.1	
IARDW	Instance Refinement Delta Width	6.4.8.2 and 7.4.3.2.1	
IARDX	Instance Refinement X Offset	6.4.8.3. and 7.4.3.2.1	3.
IARDY	Instance Refinement Y Offset	6.4.8.3. and 7.4.3.2.1	3.
IADH	Height Class Delta Height	6.5.2 and 7.4.2.1.1	
IADW	Height Class Delta Width	6.5.3 and 7.4.2.1.1	
IAAI	Aggregate Information	6.5.4.2. and 7.4.2.1.1	4.
IAEX	Symbol Table Run Length Export Flags	6.5.6.	5.

Notes:

1. Because the Profile that was used a strip size of 1, no B coordinate information is available.
2. There are no standardized Huffman tables for encoding the Symbol ID information. The encoder is required to determine Huffman codes for each ID, and those codes are included in the bitstream in a special table.
3. Because the Profile that was used for this experiment did not use offsets, no useful information is available.
4. Because the Profile that was used for this experiment did not utilize an Aggregate method of building symbols, no information is available.
5. Because the Profile that was used for this experiment exported all of the symbols, no useful information is available.

Table 5.1 – Brief Description of Each Data Type

Tables 5.2 through 5.7 contain the total number of kilobytes required to encode all of the 32 standard ITU documents (8 documents, at 4 resolutions).

	Entropy	**Huff – Opt Method A (+ Table)**	**Huff – Opt Method B (+ Table)**	**Huff – Opt Method C (incl. Table)**	**Huffman Standard**	**Arithmetic MSA**
IADB	4.9k	4.9k (7.2k)	5.7k (6.5k)	6.4k	6.0k	12.0k
IAFA	16.3k	16.3k (41.6k)	22.7k (23.4k)	24.1k	23.3k	25.7k
IADA	29.3k	29.4k (40.0k)	33.1k (34.2k)	35.3k	38.4k	59.9k
IAID	37.5k	37.5k (39.4k)	N/A	N/A	N/A	59.7k
IARI	2.5k	5.6k (6.0k)	5.6k (6.0k)	5.7k	5.6k	5.7k
IARDH	6.9k	6.5k (6.9k)	7.0k (7.5k)	7.6k	11.7k	13.1k
IARDW	8.9k	9.4k (9.8k)	9.4k (9.9k)	9.6k	11.0k	11.7k
IADH	0.1k	0.2k (0.6k)	0.2k (0.7k)	0.3k	0.2k	0.8k
IADW	2.3k	2.3k (3.5k)	2.6k (3.1k)	2.9k	3.3k	4.7k
Total*	**71.1k**	**74.7k (115.6k)**	**86.5k (91.2k)**	**92.0k**	**99.5k**	**133.6k**

*excluding IAID.

Table 5.2 – Total Number of Kilobytes Required to Encode all 32 Images

	Entropy	**Arithmetic Huffman**	**Arithmetic Style A**	**Arithmetic Style B**	**Arithmetic Style C**	**Arithmetic Style D**	**Arithmetic Style E**
IADB	4.9k	6.0k	N/A	N/A	5.6k	7.1k	7.2k
IAFA	16.3k	23.2k	N/A	N/A	23.1k	22.9k	22.9k
IADA	29.3k	33.6k	N/A	N/A	34.0k	37.6k	34.6k
IAID	**37.5k**	**N/A**	**N/A**	**N/A**	**45.7k**	**45.7k**	**45.8k**
IARI	2.5k	2.6k	N/A	N/A	2.7k	11.2k	12.5k
IARDH	6.9k	7.2k	N/A	N/A	7.3k	11.9k	9.4k
IARDW	8.9k	9.2k	N/A	N/A	9.2k	12.0k	11.4k
IADH	0.1k	0.2k	N/A	N/A	0.2k	0.3k	0.4k
IADW	2.3k	2.7k	N/A	N/A	2.6k	2.6k	2.6k
Total*	**71.1k**	**84.6k**	**N/A**	**N/A**	**84.7k**	**105.8k**	**100.9k**

*excluding IAID.

Table 5.3 – Total Number of Kilobytes Required to Encode all 32 Images: Arithmetic Coders – using a Saturated Context for only the Prefix Bits

	Entropy	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
IADB	4.9k	5.8k	5.6k	5.7k	5.6k	5.5k	5.6k
IAFA	16.3k	23.0k	21.8k	21.9k	22.1k	22.0k	22.0k
IADA	29.3k	32.2k	32.6k	32.2k	32.4k	33.3k	32.2k
IAID	**37.5k**	**N/A**	**38.6k**	**38.7k**	**39.5k**	**38.8k**	**38.8k**
IARI	2.5k	2.6k	2.6k	2.6k	2.7k	2.7k	2.7k
IARDH	6.9k	7.2k	7.2k	7.3k	7.3k	7.3k	7.3k
IARDW	8.9k	9.2k	9.3k	9.3k	9.2k	9.3k	9.4k
IADH	0.1k	0.2k	0.2k	0.2k	0.2k	0.2k	0.2k
IADW	2.3k	2.7k	2.7k	2.6k	2.6k	2.6k	2.6k
Total*	**71.1k**	**82.7k**	**82.0k**	**81.7k**	**82.1k**	**82.9k**	**81.9k**

*excluding IAID.

Table 5.4 – Total Number of Kilobytes Required to Encode all 32 Images Arithmetic Coders – using a Saturated Context of size 16

	Entropy	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
IADB	4.9k	5.8k	5.6k	5.7k	5.6k	5.5k	5.6k
IAFA	16.3k	23.0k	21.7k	21.8k	23.8k	21.9k	21.9k
IADA	29.3k	32.3k	36.4k	34.1k	34.1k	33.8k	32.5k
IAID	**37.5k**	**N/A**	**38.6k**	**38.7k**	**44.1k**	**39.9k**	**40.4k**
IARI	2.5k	2.6k	2.6k	2.6k	2.7k	2.7k	2.7k
IARDH	6.9k	7.2k	7.2k	7.3k	7.3k	7.3k	7.3k
IARDW	8.9k	9.2k	9.3k	9.3k	9.2k	9.3k	9.4k
IADH	0.1k	0.2k	0.2k	0.2k	0.2k	0.2k	0.2k
IADW	2.3k	2.7k	2.7k	2.6k	2.6k	2.6k	2.6k
Total*	**71.1k**	**82.9k**	**85.7k**	**83.7k**	**85.5k**	**83.2k**	**82.1k**

*excluding IAID.

Table 5.5 – Total Number of Kilobytes Required to Encode all 32 Images Arithmetic Coders – using a Saturated Context of size 10

	Entropy	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
IADB	4.9k	5.8k	6.3k	7.1k	5.6k	5.5k	5.6k
IAFA	16.3k	23.1k	21.8k	21.9k	25.4k	21.9k	22.4k
IADA	29.3k	32.4k	40.5k	35.4k	35.1k	34.3k	33.1k
IAID	**37.5k**	**N/A**	**38.8k**	**39.7k**	**48.2k**	**41.2k**	**42.1k**
IARI	2.5k	2.6k	2.6k	2.6k	2.7k	2.7k	2.7k
IARDH	6.9k	7.2k	7.2k	7.3k	7.3k	7.3k	7.3k
IARDW	8.9k	9.2k	9.3k	9.3k	9.2k	9.3k	9.4k
IADH	0.1k	0.2k	0.2k	0.2k	0.2k	0.2k	0.2k
IADW	2.3k	2.7k	2.7k	2.6k	2.6k	2.6k	2.6k
Total*	**71.1k**	**83.2k**	**90.6k**	**86.5k**	**88.1k**	**83.8k**	**83.2k**

*excluding IAID.

Table 5.6 – Total Number of Kilobytes Required to Encode all 32 Images Arithmetic Coders – using a Saturated Context of size 8

	Entropy	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
IADB	4.9k	5.9k	8.2k	8.7k	5.8k	5.5k	5.6k
IAFA	16.3k	23.3k	21.9k	22.1k	30.6k	23.0k	24.1k
IADA	29.3k	33.5k	46.1k	37.6k	40.3k	38.4k	34.9k
IAID	**37.5k**	**N/A**	**41.2k**	**42.0k**	**59.9k**	**44.1k**	**46.1k**
IARI	2.5k	2.6k	2.6k	2.6k	2.7k	2.7k	2.7k
IARDH	6.9k	7.2k	7.2k	7.3k	7.3k	7.3k	7.3k
IARDW	8.9k	9.2k	9.3k	9.3k	9.2k	9.3k	9.4k
IADH	0.1k	0.2k	0.2k	0.2k	0.2k	0.2k	0.2k
IADW	2.3k	2.8k	2.9k	2.6k	2.7k	2.7k	2.6k
Total*	**71.1k**	**84.6k**	**98.3k**	**90.5k**	**98.7k**	**89.1k**	**86.7k**

*excluding IAID.

Table 5.7 – Total Number of Kilobytes Required to Encode all 32 Images
Arithmetic Coders – using a Saturated Context of size 5

Saturated Context Size	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
Prefix	84.6k	N/A	N/A	84.7k	105.8k	100.9k
16 (64k)	82.7k	82.0k	81.7k	82.1k	82.9k	81.9k
10 (1k)	82.9k	85.7k	83.7k	85.5k	83.2k	82.1k
8 (256)	83.2k	90.6k	86.5k	88.1k	83.8k	83.2k
5 (32)	84.6k	98.3k	90.5k	98.7k	89.1k	86.7k

Table 5.8 – Total Number of Kilobytes Required to Encode all 32 Images
Arithmetic Coder Totals (Excluding IAID data)

Saturated Context Size	Arithmetic Huffman	Arithmetic Style A	Arithmetic Style B	Arithmetic Style C	Arithmetic Style D	Arithmetic Style E
Prefix	N/A	N/A	N/A	45.7k	45.7k	45.8k
16 (64k)	N/A	38.6k	38.7k	39.5k	38.8k	38.8k
10 (1k)	N/A	38.6k	38.7k	44.1k	39.9k	40.4k
8 (256)	N/A	38.8k	39.7k	48.2k	41.2k	42.1k
5 (32)	N/A	41.2k	42.0k	59.9k	44.1k	46.1k

Table 5.9 – Total Number of Kilobytes Required to Encode all 32 Images
Arithmetic Coder IAID Values

6. Recommendations

- It is recommended that the committee include a small subset of arithmetic coding methods in the JBIG-2 standard. Each of the formats included in this experiment, and numerous alternate structures would make the integer coding in the JBIG-2 standard very robust. For certain data types, using a very specific table, or the Huffman table would be appropriate, but for many of the data types, using a general-purpose coder (i.e.: Style B) would be simple to implement, and very effective.
- In accordance with the first recommendation, it is recommended that the corresponding header segments be modified to support the new arithmetic coding structures and options.
- The JBIG-2 standard should allow several arithmetic coder context sizes, to allow for different memory requirements. The size of the context can be specified in the segment headers.
- If the committee expects that optimized Huffman table structures will be a popular encoding method, then it is recommended that a more robust table structure be included in the JBIG-2 standard. The structure currently in the standard works well for general distributions of numbers but is very poor for very specific data sets. The method described in Appendix A, and formats used in other bitstreams should be considered.
- Furthermore, if optimized Huffman table structures are to be used frequently, it is recommended that the symbol instance segment should support a profile where the refinement information could be separated from the Huffman-encoded information in the bitstream, to reduce the wastage caused by forcing the bitstreams to byte boundaries.

7. References

[1] JBIG Committee, “N839: WD14492 as of 20 April 1998”.

APPENDIX A – Alternate Huffman Table Definition

This Huffman table definition was designed as an alternate to the table definition currently in the JBIG-2 Standard draft.

This method does not necessarily include all possible values from the data stream in the table. A Value that occurs in the data stream without a table entry is encoded with an Escape [ESC] code, and then encoded directly (as the difference from the lowest value in the table).

When designing a table, it is common to set a *threshold* value, and then not assign a Huffman code to any values that do not have a frequency higher than the *threshold* value. It is then very simple for an encoder to test several threshold values, and select the threshold that produces the smallest combined bitstream of table size and data size.

At the start of the table is the necessary header information,

Number of Bits for Lowest Number
Number of Bits for Delta Values
Number of Bits for [Esc] Numbers
Number of Bits for Prefix Lengths
Lowest Number in Data Set
Length of the OOB Prefix
Length of the ESC Prefix

And then each entry of the table has two values:

Length of the Symbol Prefix
Delta (Difference) to the Next Symbol

Where a Prefix Length of 0 indicates the end of the table.

The Prefix codes are generated in the same manner as the existing table structure defined in the standard.

The data stream is interpreted the same as the existing method, with the exception of the ESC codes, where after an ESC code is encountered, a specified number of bits are read in. The bits are interpreted as a value, which is then added to the lowest value in the table. A difference of 0 could indicate the end of the data stream.

Here is the procedure for decoding a bitstream encoded with this method:
(a Z was added to each variable name as to not confuse it with any existing variable name)

1) Decode 5 bits.
   Store the value of those bits in ZHTLOWLENBITS

2) Decode 5 bits.
   Store the value of those bits in ZHTDELTALENBITS
   (Note: if ZHTDELTALENBITS = 0, then all entries in the Table are separated by a length of 1)

3) Decode 5 bits.
   Store the value of those bits in ZHTESCLENBITS

4) Decode 3 bits.
   Store the value of those bits in ZHTPREFLENBITS

5) Decode 1 bit.
   Store the value of that bit in ZHTLOWSIGNBIT

6) Decode ZHTLOWLENBITS bits.
   Store the value of those bits in ZHTLOW
   If ZHTLOWSIGNBIT then negate ZHTLOW

7) Decode ZHTPREFLENBITS bits.
   Store the value of those bits in ZHTOOBLEN
   (Could be Optional, depending on the Data)

8) Decode ZHTPREFLENBITS bits.
   Store the value of those bits in ZHTESCLEN

9) ZHTNUM = 0

10) ZHTVAL[0] = ZHTLOW

11) Decode ZHTPREFLENBITS bits.
    If the value is 0, goto step 0.
    Otherwise,
    Store the value in ZHTPREFLEN[ZHTNUM]

12) ZHTNUM = ZHTNUM + 1

13) If ZHTDELTALENBITS bits = 0
    ZHTDELTALEN = 1
    Otherwise,
    Decode ZHTDELTALENBITS bits and store the value in ZHTDELTALEN

14) ZHTVAL[ZHTNUM] = ZHTVAL[ZHTNUM-1] + ZHTDELTALEN

15) Goto 0

16) Using the method described in section X.X,
    assign codes to
    ZHTPREFLEN[0]..ZHTPREFLEN[ZHTNUM-1], ZHTESCLEN & ZHTOOBLEN
    store the results in ZHTPREF[0]..ZHTPREF[ZHTNUM-1], ZHTESC, ZHTOOB respectively

17) Continue Interpreting the Remainder of the Bitstream as Follows:

    a) Decode bits until it is a recognizable code from step 0

        i) if code is ZHTOOB, return the value OOB

        ii) if code is ZHTPREF[J], return ZHTVAL[J]

        iii) if code is ZHTESC, decode the next ZHTESCLENBITS bits
             Store the result in ZHTESCCODE
             if ZHTESCCODE = 0, return EOF (End of File)
             Otherwise,
             return ZHTLOW + ZHTESCCODE