CSR	MASSACHUSETTS INSTITUTE OF TECHNOLOGY CENTER FOR SPACE RESEARCH CAMBRIDGE, MASSACHUSETTS 02139	
REVISION LOG	TITLE: Huffman Table Data Compression	DOC. NO. 36-53233

Revision	Date (mm/dd/yy)	ECO No.	Page(s) Affected	Reason	Approval
A	01/22/96	36-481	ALL	Incorporate Review comments Add Pickdata w/o Compression Add Truncated Huffman Table Revise To Provide Little Endian OUTPUT	[illegible] 5/22/96

MIT CSR

ACIS QUALITY ASSURANCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CENTER FOR SPACE RESEARCH
CAMBRIDGE, MASSACHUSETTS 02139

TO: ACIS Instrument Software Development Group

FROM: Peter G. Ford

DATE: January 26, 1996

SUBJECT: Huffman Table Class Review Action Items

The following is a list of action items that resulted from the review of the Huffman Table Class.

Person	*#*	*Description*	*Date Done*	
		General		
Dan	1	Consider coding the Huffman strings in little-endian order.	~~NOT~~ Done	Done as Request Ma/06
		Documentation		
Dan	2	Give the anchored frames belonging to figures the "Floating" attribute to prevent premature pagination.	J 29	✓
Dan	3	§4.1, page 7—change "data" to "12-bit data".	J29	✓
Dan	4	§4.2, page 7—change the title of each "Use", removing "A method of..." or "A means of"; i.e. the first use should be "Copy a selected Huffman table..."	J29	✓
Dan	5	§4.2, page 7—rename Use 2 to "Convert a set of 12-bit values to packed Huffman codes".	J29	✓
Dan	6	§4.2, page 7—remove Use 4 entirely. It is covered under Use 2.	J29	✓
Dan	7	§4.2, page 7—rename the new Use 4 to "Compress and append a set of 12-bit values to an existing buffer".	Compression NOT REQ	To Append
Dan	8	§4.3, page 7—remove mention of `Executive` and `Protocols` classes. State that the `HuffmanTable` class converts the data into varying-length bit strings and that the ~~constructor~~ uses no the `Mongoose` class.	~~J29~~ NOT True	X
Dan	9	§4.4, page 8—consider renaming the section "All about Huffman tables".	J 30	✓
Dan	10	§4.4, page 8—in the first paragraph, explain that the table is either generated by production software or uplinked from the ground.	J 30	✓

Person	*#*	*Description*	*Date Done*	
Dan	11	§4.4.1, page 9—remove the TBD in the last paragraph.	J 29	✓
Dan	12	§4.4.1.1, page 9—it is not clear that we want to suppress differencing when out-of-range of a truncated Huffman table. More discussion is needed on this, so make it a TBD.	J 29 (in text)	✓
Dan	13	§4.4.1.1, page 9—in the second paragraph, change "the third entry in a truncated table is the Literal index" to "the third entry in a truncated table contains the Literal code". Changed-Hdr	Alternate to LX J30	
Dan	14	§4.5.1, page 10—in the third sentence of the first paragraph, be more precise about when tables can be switched.	J 31	✓
Dan	15	§4.5.1, page 10—in the last paragraph, clarify the meaning of "continuing the pointer", and rephrase the last two sentences as a "***Warning***".	J 31	✓
Dan	16	§4.5.3, page 11—in the second sentence, change "input words" to "16-bit input words".	J29	✓
Dan	17	§4.5.3, page 1~~1~~2—At the end of this paragraph, change "the predetermined HUFF_TABLE_ADDR location in D-Cache" to "`huffTableCurrent`".	J29	✓
Dan	18	§4.5.3, page 12—reword the second paragraph, concentrating on the function of the code, i.e. to compress the 12-bit pixels into varying-length bit strings.	J30	✓
Dan	19	§4.5.4, page 13—change "correlation with a compression string" to "conversion to varying-length bit string".	J29	✓
Dan	20	§4.5.5, page 13—remove this section. See #6		✓
Dan	21	§4.6.3, page 17—in "Postconditions", describe the manner in which the output words are available to the client.	J31	✓
		Code		
Dan	22	*huffman_trunc.H*—change the *~danh/Design/* includes to point to public versions. TBD	current L/ Needed for testing	✓
Dan	23	*huffman_trunc.H*—change "`huffTableCurrentRef`" to "`*huffTableCurrentRef`".	Fo1	✓
Dan	24	*huffman_trunc.H*—include `lowLimit` and `highLimit` within `huffTableCurrent[]`. Table Header	Fo1	✓
Dan	25	*huffman_trunc.C*—in packData(), remove cast from `outBuff` definition and *unpkLrngth* and *comprLength* asserts.	J30	✓
Dan	26	*huffman_trunc.C*—in packData(), change "`lowlimit > 0`" to "`lowlimit != 0`".	Fo1	✓

4.0 Huffman Table Data Compression (36-53233 A)

4.1 Purpose

The Huffman data compression function provides the capability to significantly reduce the telemetry required to transfer the acquired data.

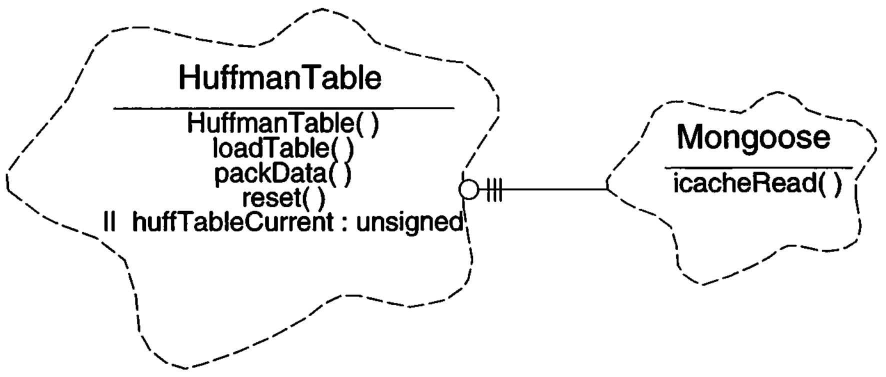
4.2 Uses

Use 1:: A method of copying the selected Huffman table from I-Cache to D-Cache.
Use 2:: A method of converting a set of values to packed Huffman codes.
Use 3:: A method of packing uncompressed data values.
Use 4:: A method of processing with truncated Huffman codes.
Use 5:: A means of appending additional buffer of codes to an existing one.

4.3 Organization

Figure 1 illustrates the relationship between the classes used by the Huffman Table Data Compression.

FIGURE 1. Huffman Table Data Compression Class Relationships



The HuffmanTable uses the ***Executive*** and ***Protocols***, class categories.

HuffmanTable- This function is responsible for converting data into Huffman codes.

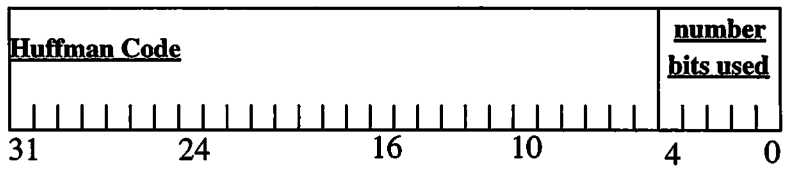
Mongoose - This class provides a quick copy from I-Cache to another memory location. It is a subclass of ***Executive***.

4.4 The Huffman Table

A Huffman table is generated by determining the statistical frequency of occurrence of each of the values which makeup the data set. The codes are distributed according to each values position in that distribution. The values which appear most frequently are given the shortest Hoffman code strings (number of bits), those least frequently encountered are assigned the longer codes. Each table entry consists of a set of bits (code) and the number of bits in that set. To use the table; as each value in the data being compressed is encountered, its corresponding code is installed (abutted) in the compressed data buffer. The number of words compressed and a buffer filled with adjacent variable length sets of ones and zeros results.

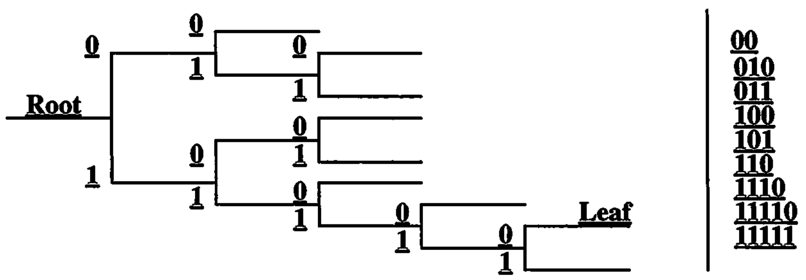
The Huffman table to be flown[1], consists of an array of words in which the low order five bits provide the number of relevant bits (left adjusted, zero filled) in the high order bits of that word. Refer to Figure 2. The bits will be packed into the output word in little endian order. Since decoding is anticipated using a tree structure, the bits must be ordered from the root, through the branches, (nodes) to a leaf. The leaf will provide the original value.

FIGURE 2. Data Distribution in Huffman Table Word



As stated, the order of words in the array is based on the statistical probability of that data value occurring in the data set. The most frequently occurring values have codes with the fewest bits. Decoding is accomplished using a logical tree structure corresponding to the compression table. Each bit extracted from the coded word determines which of the binary nodes (branch) of the tree is to be taken. When no additional node is available, (a leaf is located) that code has been determined. The data value at that location is the uncompressed value. Refer to Figure 3.

FIGURE 3. Sample Decoding Tree



1. The flight table is noted in: *CCD Bias Level Determination Algorithms*, 36-56101 Version 02, Section 5.0

4.4.1 Optimizing the Flight Table

The table is optimally tuned to the statistical probability of the frequency of occurrence of data values across the range of values anticipated. While the overall variation in some of the data may be large, the variation between adjacent pixel values is expected to be small. To further enhance the probability that most values would correspond to the shortest codes, the difference between adjacent good pixels values was used in creating the table. An initial nominal value may be provided to obtain a difference with the first pixel value. The average over-clock value is a good candidate. Bad pixels and those with parity errors are provided separate values. With twelve bits of data there are 4095 values. The possible differences are plus and minus 4095 or 8190 values; except that 4095 and 4094 are pre-empted for bad pixel and bad bias respectively, so there is no corresponding -4095 or -4094, and there is no -0. The table length is 8187 words. The bad bias and bad pixel Huffman codes are stored in the header. The values are offset to eliminate negative table indices.

To reduce the size of the table, a cutoff in the data distribution may be determined and a huffman code assigned to identify the occurrence of an accompanying literal value.

Each table is composed of a header followed by the table, itself. The header defines a unique identifier, the index of the first huffman code, the table size, and the Literal, BadBias and BadPixel codes.

4.4.1.1 Characteristics of a Truncated Huffman Table

A Literal Value is any value who's Huffman index is not available. While calculated in the usual manner; differenced from its neighbor and offset to provide only positive values, it falls in the region which has been truncated. To transmit this datum, a Literal Code will be appended to the differenced, offset, Literal Value and the combination will be packed for transmission in the usual manner. Because this outlying value could result in two combination codes being transmitted, nominal to outlier - outlier to nominal, the outlier value is not retained for differencing with the next pixel value. That next value will be differenced with the prior nominal value, the one used in determining the outlier Huffman index.

In a truncated Huffman table, as in a full table, the flagged conditions, BAD_BIAS and BAD_PIXEL, will occupy the fifth and sixth header entries. The forth entry is the truncated table Literal Index. The actual pixel value will be appended to this code before being packed. Each truncated table entry is for indices which represent themselves, indexing the lowest through the highest indices for which a “real” Huffman compression code is available. These entries have the same form as full table entries.

When creating a pseudo Literal Code/Value Huffman compression word, the Literal Code shall not exceed fifteen bits in length. The selection of the Literal Code should reflect the probability of the frequency of occurrence of the total number of truncated values.

NOTE: Because the last word in a packed buffer may not be filled, and because zeros are relevant values, the data structure headers of ALL packets which may contain Huffman packed data must contain a count of the number of items packed, or the number of items must be known a priori.

4.5 Scenarios

4.5.1 Operational Overview

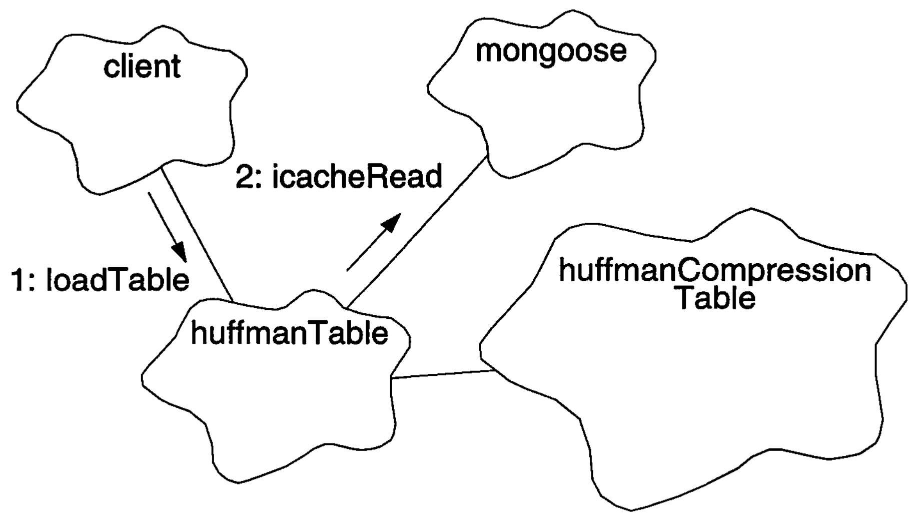
The client will call the constructor to initialize values. Before attempting to compress data, the designated Huffman Table will be selected and be copied from I-Cache into the predetermined location in D-Cache. Only one table is available at one time. Therefore, the statistical distribution of each CCDs' differences must be similar or this restriction may not provide the most efficient compression. This decision was based on the large table size.

When the Huffman data compression process is called by a client, the client provides a list of data values to be concentrated and a buffer of certain length in which to load the compressed data. The process will compress the values from the list until either; the list is exhausted, or the output buffer is filled. Each compressed item is packed (abutted) into the output buffer word. As each word is completed, any overflow will be used to start the next word. Upon return, the process will provide the number of items compressed and the number of output words filled. It is the clients prerogative to initiate a new buffer for the concentration of additional data or to append the additional data into the existing buffer. To do the former, it will re-initialize key variables, provide the additional data values, and a new buffer address. By delivering more data to the process while continuing the pointer to contiguous buffer space, the client will satisfy conditions for appending to a buffer. Appending across different buffers is possible. It is not intended. Any corruption of the compressed data buffer/s renders accurate decompression of the entire buffer/s difficult if not impossible, therefor a check-sum of some type is advisable.

4.5.2 Use 1: Loading a Huffman Table from I-Cache into D-Cache

Several Huffman tables will be located in I-Cache. A client science process e.g. `biasThief()`, will select the most appropriate Huffman compression table to be referenced by all concurrent data compressing tasks.

FIGURE 4. Load Huffman Table to D-Cache

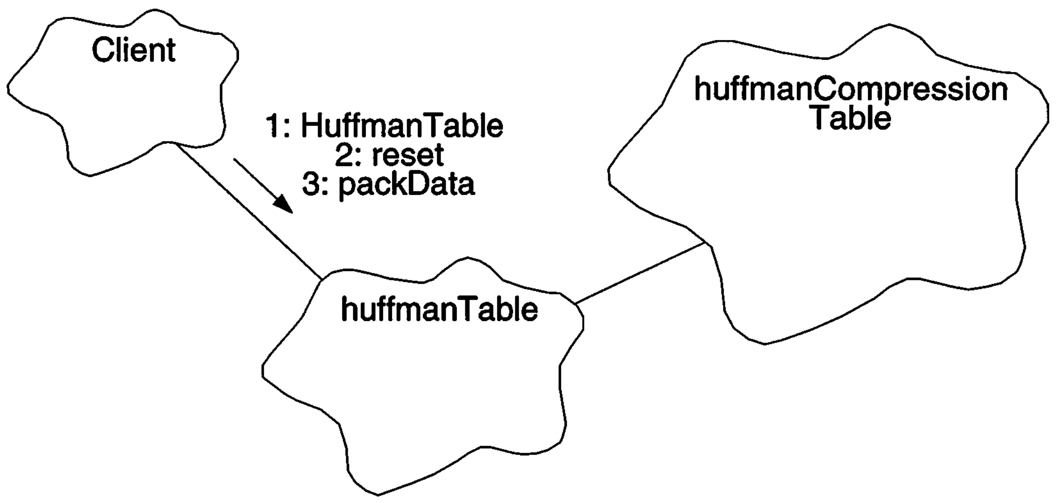


The process will use 1: the ***HuffmanTable*** function `loadTable` to have that table copied into D-Cache. `loadTable` will pass that address, and HUFF_TABLE_ADDR, the constant D-Cache location for the table, to the 2: **Mongoose** quick copy function `icacheRead` which will read the table array in I-Cache and write it into D-Cache. refer to Figure 4, above.

4.5.3 Use 2: Converting a Set of Values to Packed Huffman Codes.

The Science client will derive or obtain the data to be compressed. The data is presumed by **HuffmanTable** to be a set of pixel or bias map values stored in the low 12 bits of the input words[1]. The client will also supply a buffer of the type corresponding to the clients purpose. As shown in Figure 5, the client will 1: call the **HuffmanTable** constructor to obtain an instance of the function which will initialize some variables. The 2: `reset` function should be used whenever loading a fresh buffer. It re-initializes values retained from prior packing which are used when appending. The function will then 3: use the **HuffmanTable** function `packData` passing a reference to the data values to be converted and a reference to the buffer available to be filled along with the length of each. The proper Huffman code array is presumed to have been loaded into the predetermined HUFF_TABLE_ADDR location in D-Cache.

FIGURE 5. Convert Data to Packed Huffman Codes Using The Huffman Compression Table



The function will extract the low 12 bits from input words as the value to be compressed. It will test for the special bad bias and bad pixel values, then deliver that values array word if either is present. Otherwise, it will deliver the array word corresponding to the difference between the prior value (initially zero) and the current value, adjusted with an offset to account for possible negative values. The Huffman code bit count is extracted (removed) from the word, and the code is installed in the unused portion of the packed word being built. Codes are abutted until the word is filled or overflowed. After obtaining

1. For further information on Pixel and Bias hardware processing, refer to: *DPA Hardware Specification & System Description*, 36-02104 Rev B section 2.2.2.5 through 2.2.2.5.6.

the next output buffer word, the function installs the overflow bits beginning the next compressed word (refer to Figure 2). It will pack additional codes until this word has been filled. As each value is converted and installed, or each buffer word is filled, the count of items or buffer words is decremented. This process continues until the set of items to convert and pack is exhausted or until the available buffer is filled. If the last code to be loaded overflows the last available buffer word, the count of items compressed is decremented, and the remaining bits, their number, and the previous word value are retained in anticipation of appending overflowed bits into the next buffer provided. If the input data has been exhausted and the available space in the last output word not used, the function returns BoolTrue, indicating that additional space is available in that word. If the last code overflowed the available space, its pixel value is not counted and the space is deemed available (unused). If the word is filled exactly, the function returns BoolFalse.

With the data packed or the buffer filled, the function will return the state of the last word packed, and it will make available the number of items remaining to be packed (if any) and the number of words remaining unused in the buffer. To begin another buffer, the client uses **`HuffmanTable`**::`reset` before delivering a fresh list of values to be compressed and packed.

The function **`HuffmanTable`**::`getTableId` will return its Identifier from the header.

4.5.4 Use 3: A method of packing uncompressed data values

The same resource used to pack compressed data is used to pack uncompressed 12 bit data. After the 12 bit value has been extracted, correlation with a compression string is skipped and the full value is packed. Data is packed uncompressed after the compression table address has been specified as NULL. A table may be loaded, but it is disregarded.

4.5.5 Use 4: A method of processing with truncated Huffman codes.

Having provided the Huffman compressor with the proper arguments to acquire a truncated Huffman table; I-Cache address, length, and lowest index value, the packing of pixel values using real Huffman codes or constructed Literal Code/Value combinations will be invisible to the client. The decompression processor, when encountering the Literal Code, must extract and pass the Literal Value rather than passing the index normally associated with that code. Refer to Section 4.4.1.1 for additional details concerning the table itself.

4.5.6 Use 5: Appending Additional Codes to a Buffer

Without reinitializing with `reset`, `packData` is predisposed to continue as though it is appending to the last buffer returned to the client. If it has used the last output buffer word, the last value code attempted was partially loaded in the final word of that buffer but was not counted. The overflowed code segment is preserved across calls, and would be inserted as the first bits packed in the new buffer. The item would be counted and the first item value, corresponding to the code segment just installed, would be ignored. The function presumes that it is loading the next word of a sequential buffer. Additional compressed words would be abutted to this segment.

If input data remained and the last output word was used, yet the return state indicates space in the last word, the word has overflowed. To append, the client should call designating the next word as the beginning of the output buffer. The new input should begin with the (uncounted) last pixel value. The input data may be augmented.

If all the input data was compressed, without overflowing the output buffer, and an append request was delivered, the function would begin by obtaining the first values' code. It would then pack the code into the first output buffer word at the location just past that in which the last word was packed on the assumption that the first word of the specified buffer was the last word it had processed previously.

If all the input data was used and the state returned indicated that the last output word used was exactly filled, the client should designate the next word as the beginning of the output buffer when appending. If the state showed additional space, the client should return the last word used as the new first word of the output buffer to continuing appending.

4.6 Class HuffmanTable

This Class provides Huffman compression for data transmission.

Export Control:	Public
Cardinality:	n

Hierarchy:

Superclasses: **none**

Implementation Uses:

Mongoose

Public Interface:

Operations:

```
HuffmanTable()
getTableId()
loadTable()
packData()
reset()
```

Private Interface:

Has-A Relationships:

unsigned *accOut*: this variable is the accumulator for concatenated Huffman codes. When filled it is copied to the output buffer. When the buffer fills, this variable holds the bits which overflowed the last word in the output buffer. These bits are retained to facilitate appending codes. When the client appends data to a buffer, This value is inserted into the first word.

unsigned *highLimit*: This variable, derived from *lowLimit* and *tableSize* (from the header), is the high index of the truncated Huffman table used for compression. Processed pixel values; raw events which have been differenced and offset, and which are greater than this value will be appended with the Huffman Literal Code before being packed.

unsigned *huffTableCurrent[]*: This is an array containing the currently used Huffman look-up table. To reduce the amount of memory consumed in D-Cache by the tables, this buffer is shared by all **HuffmanTable** instances. Therefore, all active compression instances must be using the same lookup values. The designated table, which is stored in I-Cache before launch, is loaded via the static member function, `loadTable()`.

unsigned *huffTableCurrentRef*: This variable contains a copy of the I_Cache pointer referencing the address of the Huffman Table desired for this instance. A value of zero indicates that uncompressed pixel data is to be packed.

static const unsigned **huffTablePtr*: This pointer contains the I-Cache address of the table which was copied to D-Cache regardless of which instance copied it.

unsigned *huffTruncCode*: In Truncation Mode, This variable contains the converted contents of the Huffman Literal Index in which the bit count has been increased by 12. This code is used to encapsulate processed pixel values which do not have explicit indices of their own in a truncated Huffman table The original value was extracted from the header. Note: the original Huffman code provided as host for the Literal value, may not contain more than 15 Huffman code bits.

unsigned *leftOut*: This variable holds the number of bits which overflowed the last word in the output buffer. It is retained to facilitate appending codes.

unsigned *lowLimit*: This variable, located in the header of a Huffman table, contains the low index of the Huffman table used for compression. Processed pixel values; raw events which have been differenced and offset, which are less than this value will be prepended with the Huffman Literal Code before being packed. In a full table this value is zero.

unsigned *numLast*: This contains the previous non-special pixel value used to obtain the difference between it and the current value. It is retained to facilitate appending codes. This value is initialized by `reset`.

unsigned *numOut*: This variable contains the running total of bits packed in the current compressed (output) word when the last data (input) word was processed. It is the number of bits already loaded in the first buffer word when appending with new data words. It is used to facilitate appending codes.

Concurrency: Guarded

Persistence: Transient

4.6.1 HuffmanTable()

Public member of: **`HuffmanTable`**

Documentation:

This constructor function uses `reset` to initialize most of the instance variables. *huffTableCurrentRef* is initialized to zero, no compression, just pack the data. The unestablished table parameters, *lowLimit*, *highLimit* and *huffTruncCode* are set to zero.

Concurrency: Sequential

4.6.2 getTableId()

Public member of: **`HuffmanTable`**

Return Class: **`unsigned`**

Documentation:

This function returns the unique table identifier contained in the header.

Concurrency: Guarded

4.6.3 loadTable()

Public member of:	**HuffmanTable**
Return Class:	**void**
Arguments:	**const unsigned** * *icacheAddr*

Documentation:

This function is used to load the designated Huffman table located at *icacheAddr*, from I_Cache into the shared lookup table in D-Cache as *huffTableCurrent*, employing the **Mongoose** quick copy routine, icacheRead. If the pointer *huffTablePtr* matches *icacheAddr*, the desired table is currently loaded, so it is not reloaded. If *icacheAddr* is zero, that value is installed in *huffTableCurrentRef* and the table in D-Cache is not modified.

When a Huffman table is to be loaded, the limiting indices are established from parameters in the header. *lowLimits specified. highLimit* is computed as *lowLimit* plus *tableSize* minus 1. The Huffman code in the Literal index is restructured as host, *huffTruncCode*, for any 12 bit differenced and offset pixel value with an index which falls in the truncated region.

Preconditions:

The tables must begin on word boundaries.

Concurrency: Guarded

4.6.4 packData()

Public member of: **HuffmanTable**

Return Class: **Boolean**

Arguments:

const unsigned short * *unpackPtr*
unsigned & *unpkLength*
unsigned * *compressPtr*
unsigned & *comprLength*

Documentation:

This function will map each data word in the buffer pointed to by *unpackPtr*, with the corresponding Huffman code from *huffTableCurrent* and will pack that code into the output buffer pointed to by *compressPtr*. When packing the buffer, it will be able to initiate a new Huffman code series or append additional data as part of a continuing series. Designated input will be compressed into the output buffer as long as both input words and output space are available. This function considers only the least significant 12 bits of input words to be valid data. The input buffer length, *unpkLength*, and output buffer length, *comprLength*, are decremented as each type of word is processed. These arguments are available to the client upon return. The function returns the state of the last buffer word. The state is BoolTrue if the last word has available space, or BoolFalse if it was exactly filled. If not filled and it was the last output word, yet input data words remain, the last word attempted overflowed the output word. `packData` will pack 12 bit words without compression if the table pointer is NULL. When a truncated table is in use; this function will install any Literal value which has no target index, into the created Huffman code host prior to packing that code.

Preconditions:

The input, output, and table pointers must refer to an appropriate word boundary. The Huffman code length must be in the range 1 through 27 bits.

Postconditions:

The count of remaining input words and remaining output words is available to the client. The state of the last output word is returned as BoolTrue if more codes can be appended to it, else as BoolFalse if full.

Concurrency: Guarded

4.6.5 reset()

Public member of: **`HuffmanTable`**

Return Class: **`void`**

Documentation:

This function initializes instance variables, *numLast*, *numOut*, *leftOut,* and *accOut* when the client desires to begin loading a buffer rather than appending more data to an existing buffer.

Concurrency: Guarded