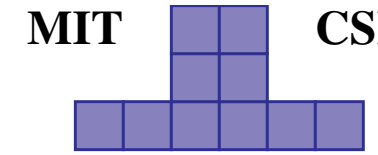

ACIS FLIGHT SOFTWARE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

CENTER FOR SPACE RESEARCH
CAMBRIDGE, MASSACHUSETTS 02139

Huffman Coding of ACIS Pixel Data

Part Number 36-56102

Version 1.1

July 29, 1996

1. Introduction

In order to make the best use of the available down-link bandwidth, ACIS 12-bit pixel data may be compressed by the flight software. The method chosen is known as the "Truncated Huffman First-Difference" algorithm which employs static compression tables. This memorandum describes the algorithm in some detail, and explains how to generate the tables and how to use them to decompress pixel and bias data.

The algorithm was first described by D. A. Huffman in *Proceedings of the Institute of Radio Engineers*, vol. 40, pp. 1098-1101, 1952. The benefit of taking first differences before compressing image pixels has been discovered many times since, notably by L. Soderblom of USGS for the Voyager Project

2. Huffman Coding

The intent of the algorithm is to translate the input, consisting of a set of 12-bit integers, into an equal number of varying-length bit strings[1]. Compression is achieved by assigning short strings to the more common integers. We require that the mapping be *lossless*, *i.e.,* reversible, and *static*, independent of the values of the input integers..

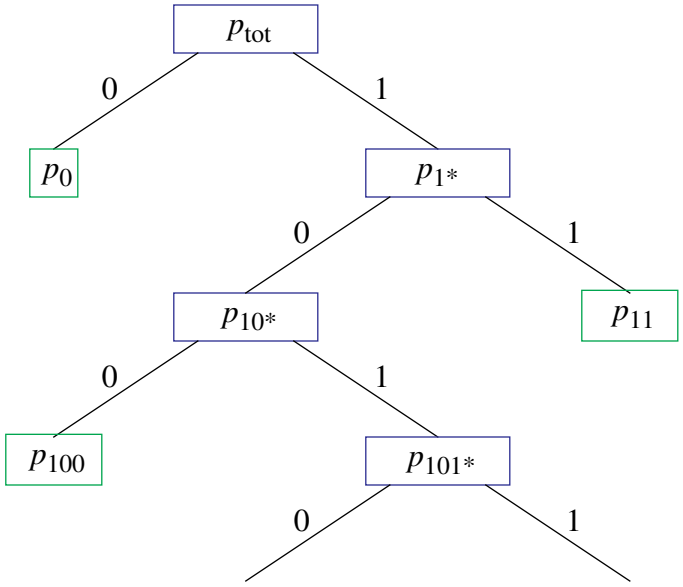

FIGURE 1. Huffman Coding Tree

If the N integers $\{i\}$ occur with probabilities p_i, the minimum number of bits required to code a string of L such integers is given[2] by $s = -L\,\Sigma p_i \log_2 p_i$. If the p_i are identical, *i.e.* all integers equally probable, $s = L\log_2 N$, *e.g.*, in our case, $N = 4096$ and $s/L = 12$. An "optimal" algorithm would encode integer i with $-\log_2 p_i$ bits, but this is not, in general, a whole number so more bits must

[1] This is the reverse of the popular Lempel-Ziv algorithm, which translates *varying* length input strings into *fixed* length output. Its major advantage over the Huffman scheme is that it can *adapt* to changes in input distribution, but its use is ruled out for ACIS telemetry because a loss of part of the compressed string prevents the remainder from being decoded.

[2] *e.g.* Gallager, R.G., *Information Theory and Reliable Communication*, New York, Wiley, 1968. In analogy with statistical mechanics, s is termed the *entropy* of the coded string.

be assigned. Huffman coding approximates the $\{p_i\}$ by inverse powers of 2, *i.e.* $p_i \cong 2^{-m_i}$ with integer m_i, and then encodes $\{i\}$ with m_i bits.

The assignment of bit strings to input integers is accomplished as follows: the $\{p_i\}$ are sorted and the two smallest values are replaced by a compound entity whose $p_{i,j} = p_i + p_j$. The process is repeated until a single element remains, with $p = 1$, representing all $\{i\}$. The process is shown graphically in Figure 1. The integers $\{i\}$ become leaves of a tree, associated with which are bit strings whose values may be determined by traversing the tree, starting at the top, and concatenating 1 or 0 depending on which sub-branch was taken. Each original i has been assigned a bit string of length m, where m is the sub-branch depth, and the iterative construction has guaranteed that each p_i is assigned to the nearest 2^{-m}.

The coding tree serves two purposes—to *generate* the bit strings from the $\{p_i\}$, and then to *decompress* a compound bit string, since it is only a matter of starting at the top of the tree and taking the 0-branch or the 1-branch depending on the value of the next input bit. Eventually, the path will end at a leaf representing the decoded value. The search then resumes at the top of the tree.

The Huffman tree is not particularly useful when compressing the original integers. This is best done by storing the varying length strings in an array indexed by the uncompressed value, i. In general, this array may be sparse, *i.e.*, if some $p_j = 0$, there is no reason to include j in $\{i\}$. Indeed, including it will reduce the overall efficiency of the algorithm to compress the remaining $\{i\}$. This would not, however, be acceptable in the present circumstances since we must assume that ACIS pixels will take on all values within the range 0-4095, and all must therefore be assigned compression strings.

3. First Differences

The most important function for our Huffman algorithm will be in reducing the size of the pixel bias maps that must be downlinked at frequent intervals, *e.g.* whenever ACIS clocking parameters are changed. Each Front End Processor (FEP) will typically contribute 4 arrays of 12-bit integers, each consisting of 1024×256 elements. Within each array, the element values $\{b_i\}$ are well described by a Gaussian random distribution $p_i = \exp[-(b_i - b_0)^2/\sigma^2]$, and the mean bias level b_0 is expected to vary from one array to the next. The width of the distribution, σ, is generally the same for all CCDs of a particular type (Front-Illuminated vs. Back-Illuminated), but is expected to increase with time on orbit as the CCDs suffer radiation damage. Typical values of σ range from 2-3 for new front-illuminated devices, 3-4 for new back-illuminated ones, and 6-12 for devices that have been irradiated to simulate several years in AXAF's orbit.

If we were to compress the raw bias values without allowing for the change in b_0 between output nodes and CCDs, we would have to use separate trees for each distribution. A particularly simple way of avoiding this is to subtract each bias value from its neighbor and then compress the resulting $\{b_i'\} = \{b_{i+1} - b_i\}$, which will closely approximate a Gaussian random distribution with zero mean ($b_0' = 0$) and width σ. Since the bias maps are compressed a row at a time, there will be either 2 or 4 node boundaries per row, depending on whether 2 or 4 output nodes are in use. A change in b_0 from node to node would therefore produce a single "uncommon" b_i' value. An isolated anomalous bias value would, of course, result in a pair of outlier b_i' values. Since the flight software already uses two bias values, 4094 and 4095, for special purposes, we have decided to treat them separately. The ACIS compression algorithm therefore reads as follows:

- Beginning with the first pixel of the last row[3] of the bias map, each 12-bit value b_i is inspected.
- If b_i is 4094 (signifying a parity error) or 4095 (signifying a bad pixel or column), it will be encoded as a special Huffman string.
- Otherwise, b_i is subtracted from the value of the previous pixel (disregarding those valued 4094 or 4095) and the result is used as an index into a table of Huffman strings. The table has 8187 entries representing the input range from -4093 to +4093.
- At the start of the process, the "previous" pixel is assumed to have a value of zero.
- After each pixel is compressed, a check is made to see whether the output telemetry packet is full. If so, the current partially-compressed row is dropped, the telemetry packet written, and a fresh one started at the beginning of the row. In this manner, each packet will contain a maximal number of whole rows.

```
typedef struct {              /* Huffman compression table */
  unsigned tableId;           /* 32-bit table identifier */
  unsigned lowLimit;          /* 4093 - zero difference index */
  unsigned tableSize;         /* number of entries in table */
  unsigned huffTruncCode;     /* tab[-3]: truncated table code */
  unsigned huffBadBiasCode;   /* tab[-2]: BAD_BIAS code */
  unsigned huffBadPixelCode;  /* tab[-1]: BAD_PIXEL code */
  unsigned tab[8187];         /* code table (tableSize entries) */
} HUFFTAB;
```

FIGURE 2. ACIS Compression Table

The structure of a full-size compression table is shown in Figure 2. It consists of a header containing six 32-bit unsigned integers followed immediately by the 8187 32-bit integer array. The first header element, `tableId`, is a 32-bit quantity that identifies this particular table. Its value is not used by the flight software. The second element, `lowLimit`, determines the index of the table element that encodes a particular pixel difference, *viz.* (b_i–b_{i-1}+4093-`lowLimit`). Unless the table is truncated (see below), `lowLimit` will be zero. The size of the `tab`[] array is recorded in the third header element, `tableSize`. Use of shorter tables is described in the next section. The fourth header element, `huffTruncCode`, is also reserved for truncated tables. The remaining pair of header elements, `huffBadBiasCode` and `huffBadPixelCode`, are used to encode, respectively, bias values of 4094 (bias parity error) and 4095 (bad pixel or column).

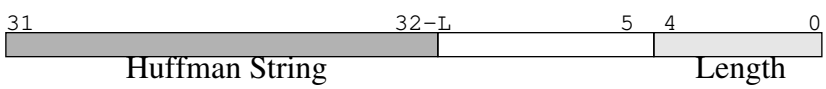

FIGURE 3. Huffman Table Element

The string begins at bit `32−L` and ends at bit 31 (the most significant). The bit length `L` is stored in bits `0−4`. Bits `31−L` through `5` are unused.

[3] The maps are downlinked in *reverse* order, beginning with the row farthest from the frame store. This is done so that the bias values close to the HRMA focus are received first. Within a row, the pixel order is as on the CCD, beginning with the first column that would be read from node A, etc.

The `tab[]` elements, and the three special header codes, are 32-bit unsigned integers containing a varying length bit string and its length. The length is contained in the 5 least-significant bits; the string is stored in little-endian order, *i.e*. its least significant bit corresponds to the root of the tree and its most significant bit—in the most significant bit of the 32-bit field—to the branch immediately above the "leaf" node, as illustrated in Figure 3. The longest string that can be specified in this format is therefore 27 bits. The little-endian convention is also followed when writing the telemetry packets—the first *n*–bit string is written to the *n* low order bits of the first output word, etc.

4. Truncated Huffman Tables

Since the full-length tables are uncomfortably large for the D-cache RAM available in the ACIS back-end processor, provision has been made to use shorter tables with `tableSize` < 8187. The compression algorithm is identical to that described in the preceding section except that the table index, b_i–b_{i-1}+4093-`lowLimit`, is tested. If it is less than zero or greater than `tableSize-1`, the value is not contained in the table so the 12-bit pixel value is prefixed with the `huffTruncCode` string and written to the output uncompressed.

FIGURE 4. Example of a Truncated Compression Table

The following array of input pixels is to be compressed with a 256–element table. The value of `lowLimit` is 3965 (*i.e*. 4093-128). Note that value of the fourth pixel is 4095, indicating that it is a member of the bad pixel or column list.

```
204  201  210  4095  202  202  200  766  208  200  202  206  201
```

Step 1—the pixels are differenced with their neighbors, but special values (4094 and 4095) are excluded.

```
204   -3    9  4095   -8    0   -2  566 -558   -8    2    4   -5
```

Step 2—4093-`lowlimit` is added to each difference to generate offsets into `tab[]`:

```
332  125  137     *  120  128  126  694 -430  120  130  132  123
```

Step 3—offsets less than zero or greater than 255 (*i.e*. `tableSize-1`) are rejected; the original 12-bit pixel values will be prefixed with `huffTruncCode` and will be excluded from differencing. Therefore, since this applies to the 8th pixel, the 9th pixel (208) will be differenced with the 7th (200):

```
  *  125  137     *  120  128  126    *  136  120  130  132  123
```

Step 4—the `tab[]` strings are output:

```
huffTruncCode 204 tab[125] tab[137] huffBadPixelCode tab[120] tab[128]
tab[126] huffTruncCode 766 tab[136] tab[120] tab[130] tab[132] tab[123]
```

The pixel value is treated as "special" in the same way as 4094 and 4095, *i.e*. it is excluded from differencing. This process is illustrated in Figure 4. The `huffTruncCode` string must be no longer than 15 bits. This allows the string, its 5-bit length, and the uncompressed 12-bit pixel value to be held in a single 32-bit data register, and speeds up the compression algorithm.

5. Compression by ACIS Flight Software

All data compression within the ACIS Back End Processor is handled by instances of the `HuffmanTable` class, each applying a particular Huffman table (or none at all). The public methods are as follows:

`HuffmanTable::HuffmanTable()`

creates an "empty" table, *i.e.* one that will not perform compression unless a table is supplied by a subsequent call to `LoadTable()`.

`void` **`HuffmanTable::LoadTable`**`(const unsigned *icacheAddr)`

loads into D-cache a compression table from an array of 32-bit words starting at `addr` in instruction cache. The array length is determined by the value of the `tableSize` field, *i.e.* `((HUFFTAB *)icacheAddr)->tableSize`. All instances of `HuffmanTable` share the same D-cache table. The consequences of this are discussed below. If `icacheAddr` is `NULL`, no table will be loaded and subsequent calls to `PackData` will not use Huffman compression.

`Boolean` **`HuffmanTable::PackData`**`(const unsigned short *inPtr, unsigned &inLength, unsigned *outPtr, unsigned &outlength)`

compresses an array of `inLength` 16-bit words beginning at `inPtr` into the output buffer beginning at `outPtr`. The length of the output buffer is `outLength` 32-bit words. `PackData` returns `TRUE` if the last word of the output buffer is partially full; otherwise, it returns `FALSE`. It also updates `inLength` and `outlength` so the caller can determine whether the output buffer was long enough to hold the entire compressed input buffer.

A `HuffmanTable` object remembers the state of the last `PackData` call. The next time its `PackData` method is invoked without an intervening `reset`, it will remember whether the first word was partially filled from the previous call.

If a `HuffmanTable` instance is created and its `PackData` method is used without any prior `LoadTable` call, or if the most recent `LoadTable` call specified an `icacheAddr` of `NULL`, no compression will be performed—the least significant 12 bits of each 16-bit pixel will be packed into the output buffer.

`void` **`HuffmanTable::reset`**`(unsigned lastNum)`

resets the private variables that remember the state of the most recent call to `PackData`. This is called before creating a new telemetry packet. `lastNum` will be subtracted from the first pixel value before it is compressed.

To conserve fast memory (D-cache), all instances of `LoadTable` copy the compression tables into the *same* location. This has the following consequences:

- The **`biasThief`** process, which compresses the individual FEP bias maps into telemetry packets, must fully compress one map before going on to the next. It cannot interleave the packets from separate FEPs since they may require different compression tables.
- Within a given science run, all *raw mode* telemetry must be compressed by the same Huffman table.
- The **`biasThief`** cannot be used during a raw mode science run since it would contend for the same compression table.

6. Generating a Huffman Table

The huff program (see appendix) performs all phases of table building:

- reads a FITS image file and generates a pixel-difference frequency histogram
- creates a Huffman tree from the histogram
- creates a Huffman table from the tree
- uses the table to compress the original FITS image
- creates a Huffman tree from the table
- uses the tree to decompress the compressed file
- compares the result to the original FITS file

The command syntax is

`huff [-b] [-r] [-i` *`id`*`] [-n` *`size`*`] [-m` *`ntrunc`*`] [-t` *`table`*`]` *`in out`*

where *`in`* is a FITS file containing 16-bit pixels. If the byte order of the pixels doesn't match that of the host computer, use the `-b` flag to swap pairs of bytes. The Huffman table will contain *`size`* elements (the default is 8187) plus the 6-element header. The `tableId` value will be set to `id`. The compressed image will be written to *`out`*.

If the `-t` option is specified, the binary Huffman table will be written to the file named *`table`*. Alternatively, if the `-r` flag is specified, a previously computed Huffman table will be read from *`table`*. The byte order within external tables is always little-endian.

`huff` creates the histogram of pixel differences from the values of the low-order 12 bits of each pixel in *`in`*, keeping a separate count of pixels valued 4094 and 4095. Any unfilled histogram entries are given the value of 1. If this were not done, the resulting table would have no entry corresponding to this pixel difference. Although not needed for compressing this particular image, the table couldn't be used to compress other images that *might* contain this pixel difference.

When *`size`* is less than 8187, `huff` also separately counts the pixel differences that lie outside the range of the table, and this count is treated as a separate entry in the histogram. After creating the Huffman tree and table, the length of the bit string corresponding to this out-of-range value is inspected. If longer than 15 bits, it is interchanged with the longest string that is shorter than 16 bits. Since this exchange is somewhat arbitrary, an alternative method is provided by the `-m` option. This supplies an integer *`ntrunc`* to be added to the out-of-range histogram entry and hence to decrease the length of the bit string assigned to out-of-range differences. The user invokes `huff` with various choices of *`ntrunc`* until the program reports a satisfactory out-of-range code length that didn't force a reallocation of table indices. Note that this only affects the efficiency of the table—it will still compress any *`in`* file whatever the value of *`ntrunc`*.

To help understand the program, here is a brief description of each procedure, grouped by function:

6.1. Setup

- `unsigned short *openFits(char *in, unsigned *nx, unsigned *ny)`

  Open the FITS file *in*, return its size (*nx* columns in *ny* rows) and a pointer to a fresh input buffer.

- `unsigned *`**`makeHistFromFile`**`(char *infile, int bswap, unsigned nmisc)`

  Read the FITS file *infile* and return a pointer to its pixel-difference histogram. If *bswap* is non-zero, the input pixels are first byte-swapped. The number of histogram elements will be *size*, and, when this is less than 8187, the out-of-range histogram count will be increased by *nmisc*.

- `NODE *`**`makeTreeFromHist`**`(unsigned *hist, unsigned size, unsigned min)`

  Convert the histogram *hist* into a Huffman tree, returning a pointer to its root node. The histogram contains *size* elements and the element corresponding to zero pixel difference is *hist*`[4093-`*min*`]`.

- `void `**`makeTableFromTree`**`(unsigned code, unsigned len, NODE *np, unsigned *tab)`

  Called iteratively to transform a Huffman tree anchored by *np* into a pre-allocated Huffman table *tab*`[].`*code*, the Huffman code of length *len* bits, is built up a bit at a time by each iteration.

- `int `**`compareLeafFreqs`**`(NODE **a, NODE **b)`

  Called by library function `qsort()` to compare the frequency fields in a pair of tree leafs.

- `void `**`insertSort`**`(NODE *npp, unsigned ii)`

  Resort the *npp[]* pointer array, dimension *ii*, into ascending pixel frequency. On entry, only the first element is (possibly) out of order, so an efficient shift-sort is performed.

- `HUFFTAB *`**`makeHuffFromFile`**`(char *infile, int bswap, unsigned size, unsigned nmisc)`

  Read the FITS file *infile* and return the address of its Huffman compression table. *bswap* is non-zero if the input pixels are to be byte-swapped. The number of pixel-difference elements will be *size*, and, when this is less than 8187, the out-of-range histogram count will be increased by *nmisc*.

- `NODE *`**`makeTreeFromTable`**`(HUFFTAB *huff)`

  Transform Huffman table *huff* into a tree, returning a pointer to its root.

- `NODE *`**`addLeafToTree`**`(NODE *np, int val, unsigned code, unsigned len)`

  Called recursively from `makeTreeFromTable` to locate the node in the tree *np* corresponding to the bit string *code* of length *len*, and to create a leaf node at that location.

- `HUFFTAB *`**`readTable`**`(char *table)`

  Called to read a Huffman table from the file *table* and return a pointer to it.

- `void `**`writeTable`**`(char *outfile, HUFFTAB *huff)`

  Called to write the Huffman table *huff* to the file *outfile*.

6.2. File Compression

- `void` **`compressFile`**`(char *infile, int bswap, char *outfile, HUFFTAB *huff)`

  Compress the FITS file *`infile`* using the Huffman table *`huff`* and write it to *`outfile`*.

- `unsigned` **`compressArray`**`(unsigned short *in, unsigned inlen, unsigned *out, HUFFTAB *huff, unsigned initval)`

  Compress *`inlen`* elements of pixel array *`in`*`[]` to the *`out`*`[]` array using the Huffman table *`huff`*. The first pixel will be subtracted from `initval`.

6.3. File Decompression

- `void` **`uncompressFile`**`(char *infile, int bswap, char *tmpfile, HUFFTAB *huff)`

  Decompress the file *`tmpfile`* using the Huffman table *`huff`*. Compare each decompressed row with the contents of the FITS image *`infile`*. If *`bswap`* is non-zero, the input pixels are first byte-swapped.

- `unsigned` **`uncompressArray`**`(unsigned *in, unsigned inlen, unsigned short *out, unsigned outlen, NODE *root, unsigned offset, unsigned initval)`

  Decompress *`outlen`* 12-bit pixels from the *`in`*`[]` array using the Huffman tree anchored at *`root`*. Add `initval` to the first decompressed value. Write the output to the *`out[]`* array. Return 1 if exactly *`inlen`* elements are unpacked. Otherwise, return 0 to indicate an unpacking error.

6.4. Diagnostic Output

The following *stderr* output is to be expected from `huff`:

```
$ huff -b -n 256 biasfile compfile
biasfile: input bytes 1572864 bits 1024x1024x12 mean 4093.03 sigma 35.61
Pixel frequency: max 423694 misc 2201 badpix 0 badbias 0
Huffman 256 code lengths: min 1 max 20 misc 9 badpix 20 badbias 20
compfile: compressed to 526292 bytes (33.46%)
compfile: uncompressFile was successful
```

On the command line, `-b` indicates that the input FITS image "`biasfile`" was created by a CPU with the opposite byte-order (*e.g.* DecStation *vs.* Sun). The `-n` option specifies that the Huffman table is to have only 256 elements.

`huff` reads `biasfile` and reports its size, mean histogram index, and RMS deviation about the mean. It lists the number of counts in the largest histogram element (`max`), the number of out-of-range differences (`misc`), the number of bad pixels valued 4095 (`badpix`), and the number of bias errors valued 4094 (`badbias`).

The next output line lists the length in bits of the most frequent Huffman string (`min`), the least frequent (`max`), the out-of-range string (`misc`), and the strings assigned to bad pixels (`badpix`) and bias errors (`badbias`).

Finally, `huff` reports the size of the compressed file, the compression factor, and, hopefully, that the decompression step completed without any error being detected.

7. Listing a Huffman Table

The following Perl script will read a binary Huffman table and write its contents to *stdout*, listing the decimal values of the header fields and the strings in the compression table:

```
#! /usr/local/bin/perl

 die "$ARGV[0]: $!\n" if $ARGV[0] && ! open(STDIN,$ARGV[0]);
 die "$ARGV[0]: $!\n" unless read(STDIN,$buf,65536) > 24;

 ($id,$min,$len,@val) = unpack('V*', $buf);
 @tit = ('tabid', 'lowlim', 'tabsize', 'trunc', 'badbias', 'badpix');

 for ($id,$min,$len) { printf "%7.7s %d\n", shift(@tit), $_;}

 for (@val) {
   printf "%7.7s %2d %s\n", shift(@tit), $_ & 31,
     join('',reverse(split(//,unpack('B'.($_ & 31),pack('N',$_))))));
   @tit = ($i++ - 4093 + $min) unless @tit;
 }
```

The following output was generated from a truncated table that contained 32 table entries. It is optimized to compress a bias map with a standard deviation of 8.2:

```
  tabid 1234
 lowlim 4077
tabsize 32
  trunc  8 01001000
badbias 12 000111010001
 badpix 12 000111010000
    -16 11 00011101001
    -15 10 1011010000
    -14  9 000111011
    -13  8 00011100
    -12  8 10110101
    -11  7 0100101
    -10  6 000110
     -9  6 101100
     -8  5 01000
     -7  5 01110
     -6  5 10111
     -5  4 0010
     -4  4 0101
     -3  4 1000
     -2  4 1010
     -1  4 1101
      0  4 1111
      1  4 1110
      2  4 1100
      3  4 1001
      4  4 0110
      5  4 0011
      6  4 0000
      7  5 01111
      8  5 00010
      9  6 010011
     10  7 1011011
     11  7 0001111
     12  8 01001001
     13  9 101101001
     14 10 1011010001
     15 10 0001110101
```

After the first three header variables, `tabid`, `lowlim`, and `tabsize`, the second column lists the length of the Huffman strings and the third column shows their bit-string values, with the least significant bit on the left, *i.e.* the left-most bit corresponds to the root of the Huffman table. The numeric values in the first column refer to the difference indices, *i.e.* a pixel difference of `+8` will be coded as the string `'00010'` whose decimal value is 8. (The strings are displayed with their *least* significant bit leftmost).

8. Recovering from Telemetry Errors

The BEP's bias thief will compressed timed-exposure bias maps into `dataTeBiasMap` packets one row at a time until the maximum packet size, 1023 32-bit words, is exceeded. Since the Huff-

man algorithm typically achieves compression factors of 35%, each packet will contain an average of 7 rows of 1024 values and will span 6 telemetry minor frames in Format 2 (ACIS in the focus). The situation is different in raw mode, where the `dataTeRaw` and `dataCcRaw` packets each contain a single row of pixels with a typical compression factor of 50%. They will occupy no more than two minor frames in Format 2.

Once the ACIS packets are ingested by the spacecraft telemetry system, they are assigned Reed-Solomon strings and any subsequent bit errors can be detected and, in most cases, corrected on the ground. Before that time, however, the telemetry packets may experience SEUs while waiting to be written from the BEP's general purpose memory. It is therefore necessary to consider two types of damage to packets containing compressed strings: the loss of an entire minor frame, and the effect of a single bit error.

8.1. Missing Minor Frames

Assuming that only one minor frame is missing, the task is to interpret the remainder of the packet. For `dataTeRaw` or `dataCcRaw` packets, which will usually span no more than two minor frames, the packet header itself is also likely to be missing, so its length must be deduced by first locating the synch word that starts the next packet, then counting back over one or more pad bytes.[4] For `dataTeBiasMap` packets, this will occur less frequently—most bad minor frames will contain only compressed pixel strings.

The remainder of the packet can only be decompressed correctly if the starting bit of a Huffman string can be identifier and if the value of the previous pixel can be computed. Although it is not, in general, possible to achieve these goals with 100% reliability, the following scheme should work in most cases.

Begin by decompressing the remainder of the packet assuming that bit n is the start of a Huffman string and that the previous decompressed pixel value was 0. Since any infinite bit string can be represented by a sequence of Huffman strings, this process will yield a set of N_n pixel values $\{b_{i,n}\}$, $0 \le i < N_n$. Some choices of n can be eliminated immediately, *i.e.* those in which N_n exceeds the number of possible pixels, where the span of values $\max_i\{b_{i,n}\}$ - $\min_i\{b_{i,n}\}$ exceeds 4095, and where the last Huffman string is truncated. This process should be repeated for all values of n from 0 to one less than the length of the longest Huffman string.

When the packet is a `dataTeBiasMap` and it is known to contain one or more bad pixels or columns, which will be represented by `huffBadPixelCode` strings, the absence of those strings can be used to eliminate various choices of n.

The most probable choice among the surviving values of n may now be estimated by comparing the decompressed pixel arrays $\{b_{i,n}\}$ with the averages over the number of columns, N_c, of the pixels of that particular CCD reported in the remaining packets, $\{B_m\}$, $0 \le m < N_c$, *i.e.* the following estimator function should be minimized with suitable weights σ_m:

$$p_n = \sum_{i=0}^{N_n - 1} \left| \frac{b_{i,n} + c_n - B_m}{\sigma_m} \right|^2 \qquad (1)$$

[4] with the possibility that these could represent the end of the damaged packet itself. If this situation arises frequently, it can be eliminated by choose a Huffman table such that no string ends with the 8-bit pad sequence.

where $m = (i\text{-}N_n) \bmod N_n$. *i.e.*, m is the column index corresponding to pixel i, counting back from the end of the packet. The sum should, of course, omit $b_{i,n}$ with values of 4094 and 4095. For compressed raw pixel packets, values of $b_{i,n}$ larger than some threshold should also be omitted since these probably correspond to events. Finally, before choosing the smallest p_n, the starting offsets c_i must be computed. This is easily done by solving the equation

$$\frac{\partial p_n}{\partial c_n} = 0 = 2 \sum_{i=0}^{N_n - 1} \frac{b_{i,n} + c_n - B_m}{\sigma_m} \qquad (2)$$

for c_n. The choice of column weighs σ_m will depend on the type of packet and on the characteristics of the particular CCD. Typically, the columns at the edge of an output node boundary should be given greater weight (smaller σ_m) since the change in average bias at that location will provide the most useful evidence for the correct choice of bit offset n.

8.2. Single Bit Error within a Compressed Packet

Unlike the previous case, in which an entire minor frame was dropped, there is no guarantee that this problem will be detected. It will sometimes announce itself, *e.g.* when the number of decompressed elements differs from the number expected in the header, or when a bias value that is known to be a member of the bad pixel or column list is reported as "good".

If an array of N integers, each occurring with probability p_i, is compressed by a set of optimal Huffman strings of bit length $\{l_i\}$, *i.e.* $l_i \geq -\log_2 p_i$, the most probable length for the resulting string is $L_0 = N\Sigma p_i l_l$ bits. When the compression is repeated for sets of such integers, L will vary about this value. For instance, if $\{p_i\}$ is Gaussian with zero mean and standard deviation σ, *i.e.* $p_i = C^{-1}\exp(-i^2/\sigma^2)$, $l_i \geq \log_2 C + (i^2/\sigma^2)\log_2 e$, and the L values will execute a random walk about L_0 with a standard deviation given by $\sigma' N^{1/2}$. If, however, a single bit is flipped in the compressed string, the decompressed length, N', will, on average, exceed N since the number of bits used in the last 32-bit word of the compressed packet is not recorded. For optimal Huffman codes, the variation in N' resulting from random bit flips depends critically on the values of the shortest l_i. If the n most probable strings are all of equal length, the probability that a randomly applied bit error will occur within one of them is $P_n = N\Sigma p_i l_i / L$, where the sum is over these n values, and the probability that this will merely change one string of length l_0 into another of the same length is $2^{-l_0} n P_n$. For the example of section 7 on page 10, $\sigma = 8.2$, $n = 12$, $l_0 = 4$, $P_n \approx 0.38$. From a Monte Carlo simulation, the probability that the decompressed length will be exactly $\langle N' \rangle$, the nearest integer to the mean of N', is 0.48 with a standard deviation of about 0.3.

It is clear that we cannot rely on a change in the size of the decompressed array to signal a bit error in the compressed string. Even if certain pixels are known to be "bad", and therefore represented by the special value 4095, a bit error will have a high probability of changing a single element without affecting subsequent elements. Since each element represents a difference of pixel values, the result will be that all subsequent decompressed values will be in error by the same (small) value.

In conclusion, since there is no reliable method of detecting a bit error in a compressed data packet, these should be accompanied by a checksum. Once the error has been detected, its location can be determined by flipping each bit in turn of the compressed array, decompressing it, and seeking the minimum value of the estimator represented by Eq. 1.

Appendix — The huff.c Program

```
/* ------------------------------------------------------------------
   Module Name:$Source: /delcano/h2/pgf/acis/huff/RCS/huff.c,v $
   Purpose:Construct and test Huffman table for FITS input
   Assumptions:Input from 16-bit FITS file
   Part Number:TBD
   Author:Peter G. Ford <pgf@space.mit.edu>
   References:
   Copyright:Massachusetts Institute of Technology 1996

   $Log: huff.c,v $
 * Revision 1.6  1996/07/29  17:11:14  pgf
 * rename 'last' to 'initval'
 *
 * Revision 1.5  1996/07/29  16:24:27  pgf
 * add 'last' parameter to compressArray() and uncompressArray()
 *
 * Revision 1.4  1996/07/23  17:15:23  pgf
 * read and write external HUFFTAB in little-endian format
 *
 * Revision 1.3  1996/04/29  17:10:30  pgf
 * add tableId to HUFFTAB and -i flag to caller arguments
 * change previous -i flag to -r
 * change tableOrigin to lowLimit in HUFFTAB
 * add code in readTable() to check for valid lowLimit
 *
 * Revision 1.2  1996/04/27  03:49:17  pgf
 * add -i option and readTable() function to support it
 *
 * Revision 1.1  1996/04/17  21:33:19  pgf
 * Initial revision
 *
   ------------------------------------------------------------------ */

static char rcsid[] = "$Id: huff.c,v 1.5 1996/07/29 16:24:27 pgf Exp $";

/* ------------------------------------------------------------------
   Function:Construct a pixel histogram from a FITS file. Then build a
   Huffman table and compress the file. Finally, decompress the file.
   ------------------------------------------------------------------ */

#include <stdio.h>

typedef struct {                    /* Huffman table */
  unsigned tableId;                 /* table itentifier */
  unsigned lowLimit;                /* 4093 - zero difference index */
  unsigned tableSize;               /* number of normal entries in table */
  unsigned huffTruncCode;           /* tab[-3]: misc code */
  unsigned huffBadBiasCode;         /* tab[-2]: BAD_BIAS code */
  unsigned huffBadPixelCode;        /* tab[-1]: BAD_PIXEL code */
  unsigned tab[1];                  /* code table */
} HUFFTAB;

typedef struct leaf {               /* Element of Huffman tree */
  struct leaf *left;                /* less frequent branch */
  struct leaf *right;               /* more frequent branch */
  unsigned freq;                    /* branch frequency */
```

```
  int val;                          /* leaf value or VAL_ code */
} NODE;

/* ---------------------------------------------------------------------
   Pixel masks and special values
   --------------------------------------------------------------------- */
#define PIXEL_MASK0xfff                         /* pixel value */
#define BAD_PIXEL0xfff                          /* ignore this pixel */
#define BAD_BIAS0xffe                           /* ignore this bias */
#define HUFF_MASK 0xfffffffe0                   /* string area of code */
#define COUNT_MASK0x1f                          /* length of code */
#define TRUNC_MAX (32-12-5)                     /* max length of spill */

/* ---------------------------------------------------------------------
   Special Huffman tree leaf values
   --------------------------------------------------------------------- */

#define VAL_BADPIX-1                            /* bad pixel leaf val */
#define VAL_BADBIAS-2                           /* bad bias leaf val */
#define VAL_MISC  -3                            /* out-of-range pixel */
#define VAL_NONE  -4                            /* leaf is a branch */

/* ---------------------------------------------------------------------
   Huffman table indices
   --------------------------------------------------------------------- */

#define HIST_MAX  8187                          /* max pixel entries */
#define HIST_SIZE 8192                          /* histogram size */
#define HIST_MID  (HIST_MAX/2)                  /* center of hist[] */
#define HIST_MISC (HIST_SIZE+VAL_MISC)          /* hist misc index */
#define HIST_BADBIAS(HIST_SIZE+VAL_BADBIAS)     /* hist bad bias index */
#define HIST_BADPIX(HIST_SIZE+VAL_BADPIX)       /* hist bad pixel index */

/* ---------------------------------------------------------------------
   Routines
   --------------------------------------------------------------------- */

unsigned short *openFits(char *, unsigned *, unsigned *);
unsigned *makeHistFromFile(char *, int, unsigned);
void compressFile(char *, int, char *, HUFFTAB *);
void uncompressFile(char *, int, char *, HUFFTAB *);
HUFFTAB *makeHuffFromFile(char *, int, unsigned, unsigned);
HUFFTAB *readTable(char *);
void makeTableFromTree(unsigned, unsigned, NODE *, unsigned *);
int compareLeafFreqs( NODE **, NODE **);
NODE *makeTreeFromTable(HUFFTAB *);
NODE *addLeafToTree(NODE *, int, unsigned, unsigned);
NODE *makeTreeFromHist(unsigned *, unsigned, unsigned);
void insertSort(NODE **, unsigned);
unsigned compressArray(unsigned short *, unsigned, unsigned *,
  HUFFTAB *, unsigned);
unsigned uncompressArray(unsigned *, unsigned, unsigned short *,
  unsigned, NODE *, unsigned, unsigned);
void writeTable(char *, HUFFTAB *);
void swap4(unsigned *pp, unsigned count);
extern char *malloc(unsigned);

/*************************************************************************
   Main: parse command line arguments, build trees, compress, decompress
```

```
 ************************************************************************/

main(int argc, char *argv[])
{
  unsigned size = HIST_MAX;/* Huffman table size */
  char *table = NULL;/* output table file */
  int bswap = 0;  /* =1 to byte-swap FITS pixels */
  int init = 0;             /* =1 to read table from -t */
  int id = 0;               /* table ID for new table */
  unsigned nmisc = 2;/* number of miscellaneous counts */
  HUFFTAB *huff;  /* Huffman table */

  while (*++argv && **argv == '-')
    if (argv[0][1] == 'b' && ! argv[0][2])
      bswap++;
    else if (argv[0][1] == 'r' && ! argv[0][2])
      init++;
    else if (argv[0][1] == 'i' && (argv[0][2] || argv[1]))
      id = atoi(argv[0][2] ? argv[0]+2 : *++argv);
    else if (argv[0][1] == 'n' && (argv[0][2] || argv[1]))
      size = atoi(argv[0][2] ? argv[0]+2 : *++argv);
    else if (argv[0][1] == 'm' && (argv[0][2] || argv[1]))
      nmisc = atoi(argv[0][2] ? argv[0]+2 : *++argv);
    else if (argv[0][1] == 't' && (argv[0][2] || argv[1]))
      table = argv[0][2] ? argv[0]+2 : *++argv;
    else
      fprintf(stderr, "bad -%c value\n", argv[0][1]), exit(1);

  if (init) {
    huff = readTable(table);
    size = huff->tableSize;
  } else {
    huff = makeHuffFromFile(argv[0], bswap, size, nmisc);
    huff->tableId = id;
    if (table) writeTable(table, huff);
  }
  compressFile(argv[0], bswap, argv[1], huff);
  uncompressFile(argv[0], bswap, argv[1], huff);
  exit(0);
}

/************************************************************************
   makeHuffFromFile: construct Huffman table from input file
 ************************************************************************/

HUFFTAB *makeHuffFromFile(char *infile, int bswap, unsigned size,
  unsigned nmisc)
{
  HUFFTAB *huff;
  unsigned len, maxlen = 0, jj, temp;
  int ii;

  huff = (HUFFTAB *)malloc(sizeof(HUFFTAB)+(size-1)*sizeof(unsigned));
  huff->tableSize = size;
  huff->lowLimit = HIST_MID-size/2;
  makeTableFromTree(0, 0,
    makeTreeFromHist(
      makeHistFromFile(infile, bswap, nmisc),
    size, huff->lowLimit),
```

```
  huff->tab);
  len = huff->huffTruncCode & COUNT_MASK;
  if (len > TRUNC_MAX) {
    for (len = jj = ii = 1; ii < size; ii++) {
      temp = huff->tab[ii] & COUNT_MASK;
      if (temp <= TRUNC_MAX && temp > len)
        jj = ii, len = temp;
    }
    temp = huff->huffTruncCode;
    huff->huffTruncCode = huff->tab[jj];
    huff->tab[jj] = temp;
    fprintf(stderr,
      "Warning: Huffman table rearranged %d 0x%08x <=> 0x%08x\n",
      jj, huff->huffTruncCode, temp);
  }
  for (ii = -3; ii < (int)size; ii++)
    if (maxlen < (huff->tab[ii] & COUNT_MASK))
      maxlen = huff->tab[ii] & COUNT_MASK;
  fprintf(stderr,
    "Huffman %d code lengths: min %d max %d misc %d badpix %d badbias %d\n",
    size, huff->tab[HIST_MID-huff->lowLimit] & COUNT_MASK, maxlen, len,
    huff->huffBadPixelCode & COUNT_MASK,
    huff->huffBadBiasCode & COUNT_MASK);
  return huff;
}

/* ---------------------------------------------------------------------
   makeHistFromFile: construct pixel frequency histogram from 'infile'
   --------------------------------------------------------------------- */

unsigned *makeHistFromFile(char *infile, int bswap, unsigned nmisc)
{
  unsigned short *inbuf;
  unsigned *hist, nx, ny, val, last, x, y;
  double sum, sumsq, npix, sqrt(double);

  inbuf = openFits(infile, &nx, &ny);
  hist = (unsigned *)malloc(HIST_SIZE*4);
  bzero(hist, HIST_SIZE*4);

  for (npix = sum = sumsq = y = 0; y < ny; y++) {
    if (fread(inbuf, 2, nx, stdin) != nx)
      perror(infile), exit(1);
    if (bswap)
      swab(inbuf, inbuf, 2*nx);
    last = inbuf[0] & PIXEL_MASK;
    if (last == BAD_PIXEL || last == BAD_BIAS)
      last = 0;
    for (x = 1; x < nx; x++)
      if ((val = inbuf[x] & PIXEL_MASK) == BAD_PIXEL)
        hist[HIST_BADPIX]++;
      else if (val == BAD_BIAS)
        hist[HIST_BADBIAS]++;
      else {
        last = val-last+HIST_MID;
        hist[last]++;
        sum += (double)last;
        sumsq += (double)last*(double)last;
        npix++;
```

```
        last = val;
      }
  }
  free(inbuf);
  hist[HIST_MISC] = nmisc;
  fprintf(stderr, "%s: input bytes %d bits %dx%dx12 mean %.2f sigma %.2f\n",
    infile, (3*nx*ny)/2, nx, ny, sum/npix, sqrt((sumsq-sum*sum/npix)/npix));
  return hist;
}

/* ---------------------------------------------------------------------
   makeTableFromTree: build Huffman table array 'tab' from Huffman tree 'np'
   --------------------------------------------------------------------- */

void makeTableFromTree(unsigned code, unsigned len, NODE *np, unsigned *tab)
{
  if (np == NULL)
    fprintf(stderr, "Error: null leaf pointer in makeTableFromTree\n"),
    exit(1);
  if (np->val == VAL_NONE) {
    makeTableFromTree(code, len+1, np->right, tab);
    makeTableFromTree(code+(1 << len), len+1, np->left, tab);
  } else {
    tab[np->val] = (code << (32-len)) | len;
  }
}

/* ---------------------------------------------------------------------
   makeTreeFromHist: build Huffman tree from pixel histogram 'hist'
   --------------------------------------------------------------------- */

NODE *makeTreeFromHist(unsigned *hist, unsigned size, unsigned min)
{
  NODE *nodes[HIST_SIZE], **npp = nodes, *np;
  int ii;

  if (min < 0 || min+size > HIST_MAX)
    fprintf(stderr, "bad -n value: %d\n", size);

  /* allocate a leaf node to each pixel value */
  for (ii = 0; ii < HIST_SIZE; ii++) {
    if (ii < min || (ii >= min+size && ii < HIST_MISC))
      hist[HIST_MISC] += hist[ii];
    else {
      np = *npp++ = (NODE *)malloc(sizeof(NODE));
      np->freq = hist[ii] ? hist[ii] : 1;
      np->val = ii < HIST_MISC ? (ii - min) : (ii - HIST_SIZE);
      np->left = np->right = NULL;
    }
  }
  qsort(nodes, ii = npp-nodes, sizeof(NODE *), compareLeafFreqs);
  /* build branch nodes to connect the leaves */
  for (npp = nodes, np = *npp; ii-- > 1; ) {
    np = (NODE *)malloc(sizeof(NODE));
    np->right = *npp++;
    np->left = *npp;
    np->val = VAL_NONE;
    np->freq = np->right->freq + np->left->freq;
    *npp = np;
```

```
    insertSort(npp, ii);
  }
  fprintf(stderr,
    "Pixel frequency: max %d misc %d badpix %d badbias %d\n",
    hist[HIST_MID], hist[HIST_MISC], hist[HIST_BADPIX], hist[HIST_BADBIAS]);
  return np;
}

/* ---------------------------------------------------------------------
   compareLeafFreqs: called from qsort() to compare two leaf frequencies
   --------------------------------------------------------------------- */

int compareLeafFreqs( NODE **a, NODE **b)
{
  return a[0]->freq - b[0]->freq;
}

/* ---------------------------------------------------------------------
   insertSort: insertion sort of node npp[0] into npp[] array
   --------------------------------------------------------------------- */

void insertSort(NODE **npp, unsigned ii)
{
  NODE *np;
  for (np = npp[0]; ii-- > 1 && np->freq > npp[1]->freq; npp++)
    npp[0] = npp[1];
  npp[0] = np;
}

/*********************************************************************
   compressFile: compress 'infile' to 'outfile' using Huffman table 'huff'
 *********************************************************************/

void compressFile(char *infile, int bswap, char *outfile, HUFFTAB *huff)
{
  FILE *fp;
  unsigned nx, ny, y, *outbuf, nw;
  unsigned short *inbuf, outlen;

  inbuf = openFits(infile, &nx, &ny);
  outbuf = (unsigned *)malloc((nx+1)*4);

  if ((fp = fopen(outfile, "w")) == NULL)
    perror(outfile), exit(1);

  fwrite(&nx, 1, 4, fp);
  fwrite(&ny, 1, 4, fp);

  for (y = nw = 0; y < ny; y++) {
    fread(inbuf, nx, 2, stdin);
    if (bswap)
      swab(inbuf, inbuf, 2*nx);
    outlen = compressArray(inbuf, nx, outbuf, huff, 0);
    fwrite(&outlen, 1, 2, fp);
    fwrite(outbuf, outlen, 4, fp);
    nw += outlen;
  }
  fclose(fp);
  free(inbuf);
```

```
  free(outbuf);
  fprintf(stderr, "%s: compressed to %d bytes (%.2f%%)\n",
    outfile, 4*nw+8, (800.0*nw)/(3*nx*ny));
}

/* ---------------------------------------------------------------------
   compressArray: use Huffman table 'huff' to compress 'in' array to 'out'
   --------------------------------------------------------------------- */

unsigned compressArray(
  unsigned short *in,               /* input pixel array */
  unsigned inlen,                   /* number of input pixels */
  unsigned *out,                    /* output array */
  HUFFTAB  *huff,                   /* Huffman table */
  unsigned initval                  /* previous pixel value */
)
{
  unsigned *outorg = out;           /* saved output origin */
  unsigned val;                     /* input pixel value */
  unsigned code;                    /* coded pixel value */
  unsigned bitlen = 12;             /* length of 'code' in bits */
  unsigned acc = 0;                 /* output register */
  unsigned bitout = 0;              /* length of 'acc' in bits */
  unsigned trunc;                   /* truncated pixel code */
  int      index;                   /* Huffman table index */

  /* construct code for truncated pixels */
  trunc = (12 + (huff->huffTruncCode & COUNT_MASK)) |
              ((huff->huffTruncCode & HUFF_MASK) >> 12);

  /* compress the array */
  while (inlen--) {
    if (huff) {
      /* load the next pixel */
      switch (val = *in++ & PIXEL_MASK) {
        case BAD_PIXEL:
          code = huff->huffBadPixelCode;
          break;
        case BAD_BIAS:
          code = huff->huffBadBiasCode;
          break;
        default:
          index = (val+(HIST_MID-huff->lowLimit))-initval;
          initval = val;
          if (index < 0 || index >= huff->tableSize)
            code = trunc | (val << 20);
          else
            code = huff->tab[index];
          break;
      }
      bitlen = code & COUNT_MASK;
      code >>= 32 - bitlen;
    } else
      code = *in++;

    /* append 'code' to 'acc' */
    acc |= code << bitout;
    if ((bitout += bitlen ) >= 32) {
      bitout -= 32;
```

```
      *out++ = acc;
      acc = code >> (bitlen - bitout);
    }
  }

  /* anything left to save? */
  if (bitout > 0)
    *out++ = acc;

  /* return output word count */
  return out - outorg;
}

/*************************************************************************
   uncompressFile: decompress 'tmpfile' using Huffman tree 'root' and
                   compare the result to the original 'infile'
 *************************************************************************/

void uncompressFile(char *infile, int bswap, char *tmpfile, HUFFTAB *huff)
{
  FILE *fp;
  NODE *root;
  unsigned nx, ny, x, y, ii, maxlen, *tmpbuf = NULL;
  unsigned short *inbuf, *cmpbuf, tmplen;

  root = makeTreeFromTable(huff);
  inbuf = openFits(infile, &nx, &ny);
  if ((fp = fopen(tmpfile, "r")) == NULL)
    perror(tmpfile), exit(1);
  if (fread(&ii, 4, 1, fp) != 1 || ii != nx)
    fprintf(stderr, "%s: bad x-dim: %d (not %d)\n", tmpfile, ii, nx);
  if (fread(&ii, 4, 1, fp) != 1 || ii != ny)
    fprintf(stderr, "%s: bad y-dim: %d (not %d)\n", tmpfile, ii, ny);
  cmpbuf = (unsigned short *)malloc(nx*2);

  for (y = maxlen = 0; y < ny; y++) {
    if (fread(inbuf, 2, nx, stdin) != nx)
      perror(infile), exit(1);
    if (bswap)
      swab(inbuf, inbuf, nx*2);
    if (fread(&tmplen, 2, 1, fp) != 1)
      perror(tmpfile), exit(1);
    if (tmplen > maxlen) {
      if (tmpbuf) free(tmpbuf);
      tmpbuf = (unsigned *)malloc((maxlen = tmplen)*4);
    }
    if (fread(tmpbuf, 4, tmplen, fp) != tmplen)
      perror(tmpfile), exit(1);
    if (!uncompressArray(tmpbuf, tmplen, cmpbuf, nx, root,
        HIST_MID-huff->lowLimit), 0)
      fprintf(stderr, "%s: decompression fails at line %d\n", tmpfile, y),
      exit(1);
    for (x = 0; x < nx; x++)
      if ((inbuf[x] & PIXEL_MASK) != cmpbuf[x])
        fprintf(stderr, "%s: line %4d col %4d orig %5d unpk %5d\n",
          tmpfile, y, x, inbuf[x], cmpbuf[x]);
  }
  fclose(fp);
  free(inbuf);
```

```
  free(tmpbuf);
  free(cmpbuf);
  fprintf(stderr, "%s: uncompressFile was successful\n", tmpfile);
}

/* ---------------------------------------------------------------------
   makeTreeFromTable: construct a Huffman tree from Huffman table 'huff'
   ------------------------------------------------------------------- */

NODE *makeTreeFromTable(HUFFTAB *huff)
{
  NODE *root = NULL;
  unsigned tab[HIST_SIZE];
  int ii;

  for (ii = -3; ii < (int)huff->tableSize; ii++)
    root = addLeafToTree(root, ii,
      huff->tab[ii] & HUFF_MASK, huff->tab[ii] & COUNT_MASK);

  bzero(tab, sizeof(tab));
  makeTableFromTree(0, 0, root, tab+3);
  if (bcmp(tab, &huff->huffTruncCode, 4*(huff->tableSize+3)))
    fprintf(stderr, "failed to construct Huffman tree\n");

  return root;
}

/* ---------------------------------------------------------------------
   addLeafToTree: add a new leaf to a Huffman tree
   ------------------------------------------------------------------- */

NODE *addLeafToTree(NODE *np, int val, unsigned code, unsigned len)
{
  if (np == NULL) {
    np = (NODE *)malloc(sizeof(NODE));
    np->left = np->right = NULL;
    np->freq = 0;
    np->val = VAL_NONE;
  }

  if (len == 0)
    np->val = val;
  else if (code & (1 << (32 - len)))
    np->left = addLeafToTree(np->left, val, code, len-1);
  else
    np->right = addLeafToTree(np->right, val, code, len-1);

  return np;
}

/* ---------------------------------------------------------------------
   uncompressArray: use Huffman tree 'root' to unpack array 'in' to 'out'
   ------------------------------------------------------------------- */

unsigned uncompressArray(
  unsigned *in,                    /* input packed array */
  unsigned inlen,                  /* number of input elements */
  unsigned short *out,             /* output array */
  unsigned outlen,                 /* maximum length of output */
```

```
  NODE *root,                          /* Huffman tree */
  unsigned offset,                     /* Huffman table zero difference index */
  unsigned initval                     /* previous unpacked value */
)
{
  unsigned code;                       /* unpacking buffer */
  unsigned bitlen;                     /* length of unpacking buffer */
  unsigned temp;                       /* scratch */
  NODE *np = root;                     /* Huffman tree pointer */

  while (inlen-- > 0) {
    code = *in++;
    bitlen = 32;
    while (bitlen--) {
      np = (code & 1) ? np->left : np->right;
      code >>= 1;
      switch (np->val) {
        case VAL_NONE:
          continue;
        case VAL_BADPIX:
          *out++ = BAD_PIXEL;
          break;
        case VAL_BADBIAS:
          *out++ = BAD_BIAS;
          break;
        case VAL_MISC:
          if (bitlen >= 12) {
            initval = *out++ = code & 0xfff;
            code >>= 12;
            bitlen -= 12;
          } else {
            temp = code;
            if (inlen-- == 0)
              return 0;
            code = *in++;
            initval = *out++ = (temp | (code << bitlen)) & 0xfff;
            bitlen += 20;
            code >>= 32-bitlen;
          }
          break;
        default:
          initval = *out++ = np->val+initval-offset;
          break;
      }
      if (--outlen == 0)
        return inlen == 0;
      np = root;
    }
  }
  return 0;
}

/*********************************************************************
   openFits: open a FITS file using stdin, read its header, return nx,ny
 *********************************************************************/

unsigned short *openFits(char *infile, unsigned *nx, unsigned *ny)
{
  char hdr[2880];
```

```
  int ii, neof, bits = 0;

  if (freopen(infile, "r", stdin) == NULL)
    perror(infile), exit(1);

  *nx = *ny = 0;

  do {
    if (fread(hdr, sizeof(hdr), 1, stdin) != 1)
      perror(infile), exit(1);
    for (ii = 0; ii < 2880 && (neof = strncmp(hdr+ii, "END ", 4)); ii += 80) {
      sscanf(hdr+ii, "NAXIS1 = %d", nx);
      sscanf(hdr+ii, "NAXIS2 = %d", ny);
      sscanf(hdr+ii, "BITPIX = %d", &bits);
    }
  } while(neof);

  if (bits != 16 || *nx <= 0 || *ny <= 0)
    fprintf(stderr, "%s: bad FITS header\n", infile), exit(1);

  return (unsigned short *)malloc(*nx*2);
}

/*****************************************************************
   readTable: read Huffman table from file 'file'
 *****************************************************************/

HUFFTAB *readTable(char *table)
{
  FILE *fp;
  HUFFTAB *huff;
  unsigned size;

  if ((fp = fopen(table, "r")) == NULL || fseek(fp, 0L, 2))
    perror(table), exit(1);
  huff = (HUFFTAB *)malloc(size = ftell(fp));
  if (fseek(fp, 0L, 0) || fread(huff, size, 1, fp) != 1)
    perror(table), exit(1);
  (void)fclose(fp);
  swap4((unsigned *)huff, size/4);
  fprintf(stderr, "%s: id %d size %d low %d\n", table,
    huff->tableId, huff->tableSize, huff->lowLimit);
  if (huff->lowLimit+huff->tableSize >= HIST_MAX)
    fprintf(stderr, "%s: bad lowLimit value\n", table), exit(1);
  return huff;
}

/*****************************************************************
   writeTable: write Huffman table 'huff' to file 'outfile'
 *****************************************************************/

void writeTable(char *table, HUFFTAB *huff)
{
  FILE *fp;
  unsigned size = sizeof(HUFFTAB)+(huff->tableSize-1)*sizeof(unsigned);

  if ((fp = fopen(table, "w")) == NULL)
    perror(table), exit(1);
  swap4((unsigned *)huff, size/4);
```

```
  if (fwrite(huff, size, 1, fp) != 1)
    perror(table), exit(1);
  (void)fclose(fp);
  swap4((unsigned *)huff, size/4);
}

/*********************************************************************
   swap4: perform byte-reversal of 32-bit array
 *********************************************************************/

void swap4(unsigned *pp, unsigned count)
{
#ifndef mips
#ifndef vax
   while (count-- > 0) {
    int ii = *pp;
    *pp++ = ((ii & 0xff) << 24) | ((ii & 0xff00) << 8) |
            ((ii >> 8) & 0xff00) | ((ii >> 24) & 0xff);
  }
#endif vax
#endif mips
}

/* ------------------------------------------------------------------
   End
   ------------------------------------------------------------------ */
```