9.7. Durations

There is an extensive collection of header keywords that indicate time durations, such as exposure times, but there are many pitfalls and subtleties that make this seemingly simple concept treacherous. Because of their crucial role and common use, keywords are defined below to record exposure and elapsed time.

- `XPOSURE` – [floating-point] The value field of this keyword *shall* contain the value for the effective exposure duration for the data, corrected for dead time and lost time in the units of `TIMEUNIT`. If the HDU contains multiple time slices, this value *shall* be the total accumulated exposure time over all slices.
- `TELAPSE` – [floating-point] The value field of this keyword *shall* contain the value for the amount of time elapsed, in the units of `TIMEUNIT`, between the start and the end of the observation or data stream.

Durations *must not* be expressed in ISO-8601 format, but only as actual durations (i.e., numerical values) in the units of the specified time unit.

Good-Time-Interval (GTI) tables are common for exposures with gaps in them, particularly photon-event files, as they make it possible to distinguish time intervals with "no signal detected" from "no data taken." GTI tables in BINTABLE extensions *must* contain two mandatory columns, `START` and `STOP`, and *may* contain one optional column, `WEIGHT`. The first two define the interval, the third, with a value between 0 and 1, the quality of the interval; *i.e.*, a weight of 0 indicates a *Bad*-Time-Interval. `WEIGHT` has a default value of 1. Any time interval not covered in the table shall be considered to have a weight of zero.

9.8. Recommended best practices

The following guidelines should be helpful in creating data products with a complete and correct time representation.

- The presence of the informational `DATE` keyword is *strongly recommended* in all HDUs.
- One or more of the informational keywords `DATE-xxxx` and/or `MJD-xxxx` *should* be present in all HDUs whenever a meaningful value can be determined. This also applies, e.g., to catalogs derived from data collected over a well-defined time range.
- The global keyword `TIMESYS` is *strongly recommended*.
- The global keywords `MJDREF` or `JDREF` or `DATEREF` are *recommended*.
- The remaining informational and global keywords *should* be present whenever applicable.
- All context-specific keywords *shall* be present as needed and required by the context of the data.

9.8.1. Global keywords and overrides

For reference to the keywords that are discussed here, see Table 22. The globally applicable keywords listed in section B of the table serve as default values for the corresponding `C*` and `TC*` keywords in that same section, but only when axis and column specifications (including alternate coordinate definitions) use a time scale listed in Table 30 or when the corresponding `CTYPE` or `TTYPE` keywords are set to the value `'TIME'`. Any alternate coordinate specified in a non-recognized time scale assumes the value of the axis pixels or the column cells, optionally modified by applicable scaling and/or reference value keywords; see also Section 9.2.1.

9.8.2. Restrictions on alternate descriptions

An image will have at most one time axis as identified by having the `CTYPE`*i* value of `TIME` or one of the values listed in Table 30. Consequently, as long as the axis is identified through `CTYPE`*i*, there is no need to have axis number identification on the global time-related keywords. It is expressly prohibited to specify more than one time reference position on this axis for alternate time coordinate frames, since this would give rise to complicated model-dependent non-linear relations between these frames. Hence, time scales `TDB` and `TCB` (or `ET`, to its precision) may be specified in the same image, but cannot be combined with any of the first nine time scales in Table 30; those first nine can be expressed as linear transformations of each other, too, provided the reference position remains unchanged. Time scale `LOCAL` is by itself, intended for simulations, and should not be mixed with any of the others.

9.8.3. Image time axes

Section 8.2 requires keywords `CRVAL`*ia* to be numeric and they cannot be expressed in ISO-8601 format. Therefore it is *required* that `CRVAL`*ia* contain the elapsed time in units of `TIMEUNIT` or `CUNIT`*ia*, even if the zero point of time is specified by `DATEREF`. If the image does not use a matrix for scaling, rotation and shear (Greisen & Calabretta 2002), `CDELT`*ia* provides the numeric value for the time interval. If the `PC` form of scaling, rotation and shear (Greisen & Calabretta 2002) is used, `CDELT`*ia* provides the numeric value for the time interval, and `PC`*i*_*j*, where $i = j$ = the index of the time axis (in the typical case of an image cube with axis 3 being time, $i = j = 3$) would take the exact value 1, the default (Greisen & Calabretta 2002). When the `CD`*i*_*j* form of mapping is used, `CD`*i*_*j* provides the numeric value for the time interval. If one of the axes is time and the matrix form is used, then the treatment of the `PC`*i*_*ja* (or `CD`*i*_*ja*) matrices involves at least a Minkowsky metric and Lorentz transformations (as contrasted with Euclidean and Galilean).

10. Representations of compressed data

Minimizing data volume is important in may contexts, particularly for publishers of large astronomical data collections. The following sections describe compressed representations of data in FITS images and BINTABLES that preserve metadata and allow for full or partial extraction of the original data as necessary. The resulting FITS file structure is independent of the specific data compression algorithm employed. The implementation details for some compression algorithms that are widely used in astronomy are defined in Sect. 10.4, but other compression techniques could also be supported. See the FITS convention by White et al. (2013) for details of the compression techniques, but beware that the specifications in this Standard *shall* supersede those in the registered convention.

Compression of FITS files can be beneficial for sites that store or distribute large quantities of data; the present section provides a standard framework that addresses such needs. As im-

plementation of compression/decompression codes can be quite complex, not all FITS reading and writing software is necessarily expected to support these capabilities. External utilities are available to compress and uncompress FITS files[14].

10.1. Tiled Image Compression

The following describes the process for compressing *n*−dimensional FITS images and storing the resulting byte stream in a variable-length column in a FITS binary table, and for preserving the image header keywords in the table header. The general principle is to first divide the *n*−dimensional image into a rectangular grid of subimages or "tiles." Each tile is then compressed as a block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table (see Section 7.3). By dividing the image into tiles it is possible to extract and uncompress subsections of the image without having to uncompress the whole image. The default tiling pattern treats each row of a 2-dimensional image (or higher dimensional cube) as a tile, such that each tile contains NAXIS1 pixels. This default may not be optimal for some applications or compression algorithms, so any other rectangular tiling pattern may be defined using keywords that are defined below. In the case of relatively small images it may suffice to compress the entire image as a single tile, resulting in an output binary table with a single row. In the case of 3-dimensional data cubes, it may be advantageous to treat each plane of the cube as a separate tile if application software typically needs to access the cube on a plane-by-plane basis.

10.1.1. Required Keywords

In addition to the mandatory keywords for BINTABLE extensions (see Sect. 7.3.1) the following keywords are reserved for use in the header of a FITS binary table extension to describe the structure of a valid compressed FITS image. All are mandatory.

ZIMAGE – [logical; value 'T'] The value field of this keyword *shall* contain the logical value 'T' to indicate that the FITS binary table extension contains a compressed image, and that logically this extension *should* be interpreted as an image rather than a table.

ZCMPTYPE – [string; default: none] The value field of this keyword *shall* contain a character string giving the name of the algorithm that was used to compress the image. Only the values given in Table 36 are permitted; the corresponding algorithms are described in Sect. 10.4. Other algorithms may be added in the future.

ZBITPIX – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the BITPIX keyword in the uncompressed FITS image.

ZNAXIS – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the NAXIS keyword (i.e., the number of axes) in the uncompressed FITS image.

ZNAXIS*n* – [integer; indexed; default: none) The value field of these keywords *shall* contain a positive integer that gives the value of the corresponding NAXIS*n* keywords (i.e., the size of axis *n*) in the uncompressed FITS image.

The comment fields for the BITPIX, NAXIS, and NAXIS*n* keywords in the uncompressed image *should* be copied to the corresponding fields in the ZBITPIX, ZNAXIS, and ZNAXIS*n* keywords.

10.1.2. Other Reserved Keywords

The compressed image tiles *must* be stored in the binary table in the same order that the first pixel in each tile appears in the FITS image; the tile containing the first pixel in the image *must* appear in the first row of the table, and the tile containing the last pixel in the image *must* appear in the last row of the binary table. The following keywords are reserved for use in describing compressed images stored in BINTABLE extensions; they *may* be present in the header, and their values depend upon the type of image compression employed.

ZTILE*n* – [integer; indexed; default: 1 for $n > 1$] The value field of these keywords (where *n* is a positive integer index that ranges from 1 to ZNAXIS) *shall* contain a positive integer representing the number of pixels along axis *n* of the compressed tiles. Each tile of pixels *must* be compressed separately and stored in a row of a variable-length vector column in the binary table. The size of each image dimension (given by ZNAXIS*n*) need not be an integer multiple of ZTILE*n*, and if it is not, then the last tile along that dimension of the image will contain fewer image pixels than the other tiles. If the ZTILE*n* keywords are not present then the default "row-by-row" tiling will be assumed, i.e., ZTILE1 = ZNAXIS1, and the value of all the other ZTILE*n* keywords *must* equal 1.

ZNAME*i* – [string; indexed; default: none] The value field of these keywords (where *i* is a positive integer index starting with 1) *shall* supply the names of up to 999 algorithm-specific parameters that are needed to compress or uncompress the image. The order of the compression parameters *may* be significant, and *may* be defined as part of the description of the specific decompression algorithm.

ZVAL*i* – [string; indexed; default: none] The value field of these keywords (where *i* is a positive integer index starting with 1) *shall* contain the values of up to 999 algorithm-specific parameters with the same index *i*. The value of ZVAL*i* may have any valid FITS data type.

ZMASKCMP – [string; default: none] The value field of this keyword *shall* contain the name of the image compression algorithm that was used to compress the optional null-pixel data mask. This keyword may be omitted if no null-pixel data masks appear in the table. See Sect. 10.2.2 for details.

ZQUANTIZ – [string; default: 'NO_DITHER'] The value field of this keyword *shall* contain the name of the algorithm that was used to quantize floating-point image pixels into integer values, which were then passed to the compression algorithm as discussed further in Sect. 10.2. If this keyword is not present, the default is to assume that no dithering was applied during quantization.

ZDITHER0 – [integer; default: none] The value field of this keyword *shall* contain a positive integer (that may range from 1 to 10000 inclusive) that gives the seed value for the random

[14] e.g. fpack/funpack, see https://heasarc.gsfc.nasa.gov/fitsio/fpack/

dithering pattern that was used when quantizing the floating-point pixel values. This keyword *may* be absent if no dithering was applied. See Sect. 10.2 for further discussion.

The following keywords are reserved to preserve a verbatim copy of the *value and comment fields* for keywords in the original uncompressed FITS image that were used to describe its structure. These optional keywords, when present, *shall* be used when reconstructing an identical copy of the original FITS HDU of the uncompressed image. They *should not* appear in the compressed image header unless the corresponding keywords were present in the uncompressed image.

ZSIMPLE – [logical; value 'T'] The value field of this keyword *must* contain the value of the original SIMPLE keyword in the uncompressed image.

ZEXTEND – [string] The value field of this keyword *must* contain the value of the original EXTEND keyword in the uncompressed image.

ZBLOCKED – [logical] The value field of this keyword *must* contain the value of the original BLOCKED keyword in the uncompressed image.

ZTENSION – [string] The value field of this keyword *must* contain the original XTENSION keyword in the uncompressed image.

ZPCOUNT – [integer] The value field of this keyword *must* contain the original PCOUNT keyword in the uncompressed image.

ZGCOUNT – [integer] The value field of this keyword *must* contain the original GCOUNT keyword in the uncompressed image.

ZHECKSUM – [string] The value field of this keyword *must* contain the original CHECKSUM keyword (see Sect. 4.4.2.7) in the uncompressed image.

ZDATASUM – [string] The value field of this keyword *must* contain the original DATASUM keyword (see Sect. 4.4.2.7) in the uncompressed image.

The ZSIMPLE, ZEXTEND, and ZBLOCKED keywords *must not* be used unless the original uncompressed image was contained in the primary array of a FITS file. The ZTENSION, ZPCOUNT, and ZGCOUNT keywords *must not* be used unless the original uncompressed image was contained in an IMAGE extension.

The FITS header of the compressed image *may* contain other keywords. If a FITS primary array or IMAGE extension is compressed using the procedure described here, it is *strongly recommended* that all the keywords (including comment fields) in the header of the original image, except for the mandatory keywords mentioned above, be copied verbatim and in the same order into the header of the binary table extension that contains the compressed image. All these keywords will have the same meaning and interpretation as they did in the original image, even in cases where the keyword is not normally expected to occur in the header of a binary table extension (e.g., the BSCALE and BZERO keywords, or the World Coordinate System keywords such as CTYPE*n*, CRPIX*n* and CRVAL*n*).

10.1.3. Table Columns

Two columns in the FITS binary table are defined below to contain the compressed image tiles; the order of the columns in the table is not significant. One of the table columns describes optional content; but when this column appears it *must* be used as defined in this section. The column names (given by the TTYPE*n* keyword) are reserved; they are shown here in upper case letters, but case is not significant.

COMPRESSED_DATA – [required; variable-length] Each row of this column *must* contain the byte stream that is generated as a result of compressing the corresponding image tile. The data type of the column (as given by the TFORM*n* keyword) *must* be one of '1PB', '1PI', or '1PJ' (or the equivalent '1QB', '1QI', or '1QJ'), depending on whether the compression algorithm generates an output stream of 8-bit bytes, or integers of 16-, or 32-bits.

When using the quantization method to compress floating-point images that is described in Sect. 10.2, it sometimes may not be possible to quantize some of the tiles (e.g., if the range of pixels values is too large or if most of the pixels have the same value and hence the calculated RMS noise level in the tile is close to zero). There also may be other rare cases where the nominal compression algorithm cannot be applied to certain tiles. In these cases, an alternate technique *may* be used in which the raw pixel values are losslessly compressed with the GZIP algorithm.

GZIP_COMPRESSED_DATA [optional; variable-length] If the raw pixel values in an image tile are losslessly compressed with the GZIP algorithm, the resulting byte stream *must* be stored in this column (with a '1PB' or '1QB' variable-length array column format). The corresponding COMPRESSED_DATA column for these tiles *must* contain a null pointer (i.e., the pair of integers that constitute the descriptor for the column *must* both have the value zero: see Sect. 7.3.5).

The compressed data columns described above *may* use either the '1P' or '1Q' variable-length array FITS column format if the size of the heap in the compressed FITS file is < 2.1 GB. If the the heap is larger than 2.1 GB, then the '1Q' format (which uses 64-bit pointers) *must* be used.

When using the optional quantization method described in Sect. 10.2 to compress floating-point images, the following columns are *required*.

ZSCALE – [floating-point; optional] This column *shall* be used to contain linear scale factors that, along with ZZERO, transform the floating-point pixel values in each tile to integers via,

$$I_i = round\left(\frac{F_i - \mathrm{ZZERO}}{\mathrm{ZSCALE}}\right) \qquad (12)$$

where I_i and F_i are the integer and (original) floating-point values of the image pixels, respectively and the round function rounds the result to the nearest integer value.

ZZERO – [floating-point; optional] This column *shall* be used to contain zero point offsets that are used to scale the floating-point pixel values in each tile to integers via Eq. 12.

Do not confuse the ZSCALE and ZZERO columns with the BSCALE and BZERO keywords (defined in Sect. 4.4.2) which may be present in integer FITS images. Any such integer images should normally be compressed without any further scaling, and the BSCALE and BZERO keywords *should* be copied verbatim into the header of the binary table containing the compressed image.

Some images contain undefined pixel values; in uncompressed floating-point images these pixels have an IEEE NaN value. However, these pixel values will be altered when using the quantization method described in Sect. 10.2 to compress floating-point images. The value of the undefined pixels *may* be preserved in the following way.

ZBLANK – [integer; optional] When present, this column *shall* be used to store the integer value that represents undefined pixels in the scaled integer array. The *recommended* value for ZBLANK is −2147483648, the largest negative 32-bit integer. If the same null value is used in every tile of the image, then ZBLANK *may* be given in a header keyword instead of a table column; if both a keyword and a table column named ZBLANK are present, the values in the table column *must* be used. If there are no undefined pixels in the image then ZBLANK is *not required* to be present either as a table column or a keyword.

If the uncompressed image has an integer data type (ZBITPIX > 0) then the value of undefined pixels is given by the BLANK keyword (see Sect. 5.3), which *should* be used instead of ZBLANK. When using some compression techniques that do not exactly preserve integer pixel values, it may be necessary to store the location of the undefined pixels prior to compressing the image. The locations *may* be stored in an image mask, which *must* itself be compressed and stored in a table column with the following definition. See Sect. 10.2.2 for more details.

NULL_PIXEL_MASK – [integer array; optional] When present, this column *shall* be used to store, in compressed form, an image mask with the same original dimensions as the uncompressed image, that records the location of the undefined pixels. The process defined in Sect. 10.2.2 *shall* be used to construct the compressed pixel mask.

Additional columns *may* be present in the table to supply other parameters that relate to each image tile. However, these parameters *should not* be recorded in the image HDU when the uncompressed image is restored.

10.2. Quantization of Floating-Point Data

While floating-point format images may be losslessly compressed, noisy images often do not compress very well. Higher compression can only be achieved by removing some of this noise without losing the useful information content. One commonly used technique for reducing the noise is to scale the floating-point values into quantized integers using Eq. 12, and using the ZSCALE and ZZERO columns to record the two scaling coefficients that are used for each tile. Note that the absence of these two columns in a tile-compressed floating-point image is an indication that the image was not scaled and was instead losslessly compressed.

An effective scaling algorithm for preserving a specified amount of noise in each pixel value is described by White & Greenfield (1999) and by Pence et al. (2009). With this method, the ZSCALE value (which is numerically equal to the spacing between adjacent quantization levels) is calculated to be some fraction, Q, of the RMS noise as measured in background regions of the image. Pence et al. (2009) shows that the number of binary bits of noise that are preserved in each pixel value is given by $log_2(Q) + 1.792$. Q results directly related to the compressed file size: decreasing Q by a factor of 2 will decrease the file size by about 1 bit/pixel. In order to achieve the greatest amount of compression, one should use the smallest value of Q that still preserves the required amount of photometric and astrometric precision in the image. Image quality will remain comparable regardless of the noise level.

A potential problem when applying this scaling method to astronomical images, in particular, is that it can lead to a systematic bias in the measured intensities in faint parts of the image: As the image is quantized more coarsely, the measured intensity of the background regions of the sky will tend to be biased towards the nearest quantize level. One very effective technique for minimizing this potential bias is to *dither* the quantized pixel values by introducing random noise during the quantization process. So instead of simply scaling every pixel value in the same way using Eq. 12, the quantized levels are randomized by using this slightly modified equation:

$$I_i = round\left(\frac{F_i - \mathtt{ZZERO}}{\mathtt{ZSCALE}} + R_i - 0.5\right) \quad (13)$$

where R_i is a random number between 0.0 and 1.0, and 0.5 is subtracted so that the mean quantity equals 0. Then restoring the floating-point value, the same R_i is used with the inverse formula:

$$F_i = ((I_i - R_i + 0.5) * \mathtt{ZSCALE}) + \mathtt{ZZERO} \quad (14)$$

This "subtractive dithering" technique has the effect of dithering the zero-point of the quantization grid on a pixel by pixel basis without adding any actual noise to the image. The net effect of this is that the mean (and median) pixel value in faint regions of the image more closely approximate the value in the original unquantized image than if all the pixels are scaled without dithering.

The key requirement when using this subtractive dithering technique is that *the exact same random number sequence* must be used when quantizing the pixel values to integers, and when restoring them to floating point values. While most computer languages supply a function for generating random numbers, these functions are not guaranteed to generate the same sequence of numbers every time. An algorithm for generating a repeatable sequence of pseudo random numbers is given in Appendix I; this algorithm *must* be used when applying a subtractive dither.

10.2.1. Dithering Algorithms

The ZQUANTIZ keyword, if present, *must* have one of the following values to indicate the type of quantization, if any, that was applied to the floating-point image for compression:

NO_DITHER – No dithering was performed; the floating-point pixels were simply quantized using Eq. 12. This option *shall* be assumed if the ZQUANTIZ keyword is not present in the header of the compressed floating-point image.

SUBTRACTIVE_DITHER_1 – The basic subtractive dithering was performed, the algorithm for which is described below. Note that an image quantized using this technique can still be unquantized using the simple linear scaling function given by Eq. 12, at the cost of introducing slightly more noise in the image than if the full subtractive dithering algorithm were applied.

SUBTRACTIVE_DITHER_2 – This dithering algorithm is identical to that for SUBTRACTIVE_DITHER_1, except that any pixels in the floating-point image that are exactly equal to 0.0 are represented by the reserved value −2147483647 in the quantized integer array. When the image is subsequently uncompressed and unscaled, these pixels *must be* restored to their original value of 0.0. This dithering option is useful if the zero-valued pixels have special significance to the data analysis software, so that the value of these pixels *must not* be dithered.

The process for generating a subtractive dither for a floating-point image is the following:

1. Generate a sequence of 10000 single-precision floating-point random numbers, RN, with a value between 0.0 and 1.0. Since it could be computationally expensive to generate a unique random number for every pixel of large images, simply cycle through this look-up table of random numbers.
2. Choose an integer in the range 1 to 10000 to serve as an initial seed value for creating a unique sequence of random numbers from the array that was calculated in the previous step. The purpose of this is to reduce the chances of applying the same dithering pattern to two images that are subsequently subtracted from each other (or co-added), because the benefits of randomized dithering are lost if all the pixels are dithered in phase with each other. The exact method for computing this seed integer is not important as long as the value is chosen more or less randomly.
3. Write the integer seed value that was selected in the previous step as the value of the ZDITHER0 keyword in the header of the compressed image. This value is required to recompute the same dithering pattern when uncompressing the image.
4. Before quantizing each tile of the floating point image, calculate an initial value for two offset parameters, I_0 and I_1, with the following formulae:

$$I_0 = mod(N_{tile} - 1 + \texttt{ZDITHER0}, 10000) \quad (15)$$

$$I_1 = \mathrm{INT}(\mathrm{RN}(I_0) * 500.) \quad (16)$$

   where N_{tile} is the row number in the binary table that is used to store the compressed bytes for that tile, ZDITHER0 is that value of that keyword, and RN(I_0) is the value of the I_0^{th} random number in the sequence that was computed in the first step. Note that I_0 has a value in the range 0 to 9999 and I_1 has a value in the range 0 to 499. This method for computing I_0 and I_1 was chosen so that a different sequence of random numbers is used to compress successive tiles in the image, and so that the sequence of I_1 values has a length of order 100 million elements before repeating.
5. Now quantize each floating-point pixel in the tile using Eq. 13 and using random number RN(I_1) for the first pixel. Increment the value of I_1 for each subsequent pixel in the tile. If I_1 reaches the upper limit of 500, then increment the value of I_0 and recompute I_1 from Eq. 16. If I_0 also reaches the upper limit of 10000, then reset I_0 to 0.
   If the floating-point pixel has an IEEE NaN value, then it is not quantized or dithered but instead is set to the reserved integer value specified by the ZBLANK keyword. For consistency, the value of I_1 should also be incremented in this case even though it is not used.
6. Compress the array of quantized integers using the lossless algorithm that is specified by the ZCMPTYPE keyword (use RICE_1 by default).
7. Write the compressed bytestream into the COMPRESSED_DATA column in the appropriate row of the binary table corresponding to that tile.
8. Write the linear scaling and zero point values that were used in Eq. 13 for that tile into the ZSCALE and ZZERO columns, respectively, in the same row of the binary table.
9. Repeat Steps 4 through 8 for each tile of the image.

10.2.2. Preserving undefined pixels with lossy compression

The undefined pixels in integer images are flagged by a reserved BLANK value and will be preserved if a lossless compression algorithm is used. (ZBLANK is used for undefined pixels in floating-point images.) If the image is compressed with a lossy algorithm, then some other technique must be used to identify the undefined pixels in the image. In this case it is *recommended* that the undefined pixels be recorded with the following procedure:

1. Create an integer data mask with the same dimensions as the image tile
2. For each undefined pixel in the image, set the corresponding mask pixels to 1 and all the other pixels to 0.
3. Compress the mask array using a lossless algorithm such as PLIO or GZIP, and record the name of that algorithm with the keyword ZMASKCMP.
4. Store the compressed byte stream in a variable-length array column called 'NULL_PIXEL_MASK' in the table row corresponding to that image tile.

The data mask array pixels *should* have the shortest integer data type that is supported by the compression algorithm (i.e., usually 8-bit bytes). When uncompressing the image tile, the software *must* check if the corresponding compressed data mask exists with a length greater than 0, and if so, uncompress the mask and set the corresponding undefined pixels in the image array to the value given by the BLANK keyword.

10.3. Tiled Table Compression

The following section describes the process for compressing the content of BINTABLE columns. Some additional details of BINTABLE compression may be found in Pence et al. (2013), but the specifications in this Standard *shall* supersede those in the registered convention. The uncompressed table may be subdivided into tiles, each containing a subset of rows, then each column of data within each tile is extracted, compressed, and stored as a variable-length array of bytes in the output compressed table. The header keywords from the uncompressed table, with only a few limited exceptions, *shall* be copied verbatim to the header of the compressed table. The compressed table *must* itself be a valid FITS binary table (albeit one where the contents cannot be interpreted without uncompressing the contents) that contains the same number and order of columns as in the uncompressed table, and that contains one row for each tile of rows in the uncompressed table. Only the compression algorithms specified in Sect. 10.3.5 are permitted.

10.3.1. Required Keywords

With only a few exceptions noted below, all the keywords and corresponding comment fields from the uncompressed table *must* be copied verbatim, in order, into the header of the com-

pressed table. Note in particular that the values of the reserved column descriptor keywords TTYPEn, TUNIT*n*, TSCAL*n*, TZERO*n*, TNULL*n*, TDISP*n*, and TDIM*n*, as well as all the column-specific WCS keywords defined in the FITS standard, *must* have the same values and data types in both the original and in the compressed table, with the understanding that these keywords apply to the uncompressed data values.

The only keywords that *must not* be copied verbatim from the uncompressed table header to the compressed table header are the mandatory NAXIS1, NAXIS2, PCOUNT, and TFORM*n* keywords, and the optional CHECKSUM, DATASUM (see Sect. 4.4.2.7), and THEAP keywords. These keywords must necessarily describe the contents and structure of the compressed table itself. The original values of these keywords in the uncompressed table *must* be stored in a new set of reserved keywords in the compressed table header. Note that there is no need to preserve a copy of the GCOUNT keyword because the value is always equal to 1 for BINTABLES. The complete set of keywords that have a reserved meaning within a tile-compressed binary table are given below:

ZTABLE – [logical; value: 'T'] The value field of this keyword *shall* be 'T' to indicate that the FITS binary table extension contains a compressed BINTABLE, and that logically this extension *should* be interpreted as a tile-compressed binary table.

ZNAXIS1 – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the NAXIS1 keyword in the original uncompressed FITS table header. This represents the width in bytes of each row in the uncompressed table.

ZNAXIS2 – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the NAXIS2 keyword in the original uncompressed FITS table header. This represents the number of rows in the uncompressed table.

ZPCOUNT – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the PCOUNT keyword in the original uncompressed FITS table header.

ZFORM*n* – [string; indexed; default: none] The value field of these keywords *shall* contain the character string values of the corresponding TFORM*n* keywords that defines the data type of column *n* in the original uncompressed FITS table.

ZCTYP*n* – [string; indexed; default: none] The value field of these keywords *shall* contain the character string value mnemonic name of the algorithm that was used to compress column *n* of the table. The only permitted values are given in Sect. 10.3.5, and the corresponding algorithms are described in Sect 10.4.

ZTILELEN – [integer; default: none] The value field of this keyword *shall* contain an integer representing the number of rows of data from the original binary table that are contained in each tile of the compressed table. The number of rows in the last tile may be less than in the previous tiles. Note that if the entire table is compressed as a single tile, then the compressed table will only contains a single row, and the ZTILELEN and ZNAXIS2 keywords will have the same value.

10.3.2. Procedure for Table Compression

The procedure for compressing a FITS binary table consists of the following sequence of steps:

1. Divide table into tiles (optional)
   In order to limit the amount of data that must be managed at one time, large FITS tables *may* be divided into tiles, each containing the same number of rows (except for the last tile which *may* contain fewer rows). Each tile of the table is compressed in order and each is stored in a single row in the output compressed table. There is no fixed upper limit on the allowed tile size, but for practical purposes it is *recommended* that it not exceed 100 MB.
2. Decompose each tile into the component columns
   FITS binary tables are physically stored in row-by-row sequential order, such that the data values for the first row in each column are followed by the values in the second row, and so on (see Sect. 7.3.3). Because adjacent columns in binary tables can contain very non-homogeneous types of data, it can be challenging to efficiently compress the native stream of bytes in the FITS tables. For this reason, the table is first decomposed into its component columns, and then each column of data is compressed separately. This also allows one to choose the most efficient compression algorithm for each column.
3. Compress each column of data
   Each column of data *must* be compressed with one of the lossless compression algorithms described in Sect. 10.4. If the table is divided into tiles, then the same compression algorithm *must* be applied to a given column in every tile. In the case of variable-length array columns (where the data are stored in the table heap: see Sect. 7.3.5), each individual variable length vector *must* be compressed separately.
4. Store the compressed bytes
   The compressed stream of bytes for each column *must* be written into the corresponding column in the output table. The compressed table *must* have exactly the same number and order of columns as the input table, however the data type of the columns in the output table will all have a variable-length byte data type, with TFORMn = '1QB'. Each row in the compressed table corresponds to a tile of rows in the uncompressed table.
   In the case of variable-length array columns, the array of descriptors that point to each compressed variable-length array, as well as the array of descriptors from the input uncompressed table, *must* also be compressed and written into the corresponding column in the compressed table. See Sect. 10.3.6 for more details.

10.3.3. Compression Directive Keywords

The following compression-directive keywords, if present in the header of the table to be compressed, are reserved to provide guidance to the compression software on how the table should be compressed. The compression software *should* attempt to obey these directives, but if that is not possible the software may disregard them and use an appropriate alternative. These keywords are optional, but must be used as specified below.

– FZTILELN – [integer] The value field of this keyword *shall* contain an integer that specifies the requested number

of table rows in each tile which are to be compressed as a group.

– FZALGOR – [string] The value field of this keyword *shall* contain a character string giving the mnemonic name of the algorithm that is requested to be used by default to compress every column in the table. The permitted values are given in Sect. 10.3.5.

– FZALG*n* – [string; indexed] The value fields of these keywords *shall* contain a character string giving the mnemonic name of the algorithm that is requested to compress column *n* of the table. The current allowed values are the same as for the FZALGOR keyword. The FZALGn keyword takes precedence over FZALGOR in determining which algorithm to use for a particular column if both keywords are present.

10.3.4. Other Reserved Keywords

The following keywords are reserved to store a verbatim copy of the value and comment fields for specific keywords in the original uncompressed BINTABLE. These keywords, if present, *should* be used to reconstruct an identical copy of the uncompressed BINTABLE, and *should not* appear in the compressed table header unless the corresponding keywords were present in the uncompressed BINTABLE.

ZTHEAP – [integer; default: none] The value field of this keyword *shall* contain an integer that gives the value of the THEAP keyword if present in the original uncompressed FITS table header.

ZHECKSUM – [string; default: none] The value field of this keyword *shall* contain a character string that gives the value of the CHECKSUM keyword (see Sect. 4.4.2.7) in the original uncompressed FITS HDU.

ZDATASUM – [string; default: none] The value field of this keyword *shall* contain a character that gives the value of the DATASUM keyword (see Sect. 4.4.2.7) in the original uncompressed FITS HDU.

10.3.5. Supported Compression Algorithms for Tables

The permitted algorithms for compressing BINTABLE columns are RICE_1, GZIP_1, and GZIP_2 (plus NOCOMPRESS), which are lossless and are described in Sect. 10.4. Lossy compression could be allowed in the future once a process is defined to preserve the details of the compression.

10.3.6. Compressing Variable-Length Array Columns

Compression of BINTABLE tiles that contain variable-length array (VLA) columns requires special consideration because the array values in these columns are not stored directly in the table, but are instead stored in a data heap which follows the main table (see Sect. 7.3.5). The VLA column in the original, uncompressed table only contains descriptors, which are composed of two integers that give the size and location of the arrays in the heap. When uncompressing, these descriptor values will be needed to write the uncompressed VLAs back into the same location in the heap as in the original uncompressed table. Thus, the following process *must* be followed, in order, when compressing a VLA column within a tile:

1. For each VLA in the column:
   – Read the array from the input table and compress it using the algorithm specified by ZCTYP for this VLA column.
   – Write the resulting bytestream to the heap of the compressed table.
   – Store (or append) the descriptors to the compressed bytestream (which *must* be 64-bit Q-type) in a temporary array.
2. Append the VLA descriptors from the uncompressed table (which *may* be either Q-type or P-type) to the temporary array of VLA descriptors for the compressed table.
3. Compress the combined array of descriptors using GZIP_1, and write that byte stream into the corresponding VLA column in the output table, so that the compressed array is appended to the heap.

When uncompressing a VLA column, two stages of uncompression *must* be performed in order:

1. Uncompress the combined array of descriptors using the gzip algorithm.
2. For each descriptor to a compressed array:
   – Read the compressed VLA from the compressed table and uncompress it using the algorithm specified by ZCTYP for this VLA column.
   – Write it to the correct location in the uncompressed table.

10.4. Compression Algorithms

Table 36: Valid mnemonic values for the ZCMPTYPE and ZCTYP*n* keywords

Value	Sect.	Compression Type
'RICE_1'	10.4.1	Rice algorithm for integer data
'GZIP_1'	10.4.2	Combination of the LZ77 algorithm and Huffman coding, used in Gnu GZIP
'GZIP_2'	10.4.2	Like 'GZIP_1', but with reshuffled pixel values
'PLIO_1'	10.4.3	IRAF PLIO algorithm for integer data
'HCOMPRESS_1'	10.4.4	H-compress algorithm for 2-D images
'NOCOMPRESS'		The HDU remains uncompressed

The name of the permitted algorithms for compressing FITS HDUs, as recorded in the ZCMPTYPE keyword, are listed in Table 36; if other types are later supported, they *must* be registered with the IAUFWG to reserve the keyword values. Keywords for the parameters of supported compression algorithms have also been reserved, and are described with each algorithm in the subsections below. If alternative compression algorithms require keywords beyond those defined below, they *must* also be registered with the IAUFWG to reserve the associated keyword names.

10.4.1. Rice compression

When ZCMPTYPE = 'RICE_1' the Rice algorithm (Rice et al. 1993) *shall* be used for data (de)compression. When selected, the keywords in Table 37 *should* also appear in the header with one of the values indicated. If these keywords are absent, then

their default values *must* be used. The Rice algorithm is lossless, but can only be applied to integer-valued arrays. It offers a significant performance advantage over the other compression techniques (see White et al. 2013).

Table 37: Keyword parameters for Rice compression

	Values		
Keyword	Permitted	Default	Meaning
ZNAME1	'BLOCKSIZE'	–	Size of block in pixels
ZVAL1	16, 32	32	No. of pixels in a block
ZNAME2	'BYTEPIX'	–	Size of pixel value in bytes
ZVAL2	1, 2, 4, 8	4	No. 8-bit bytes per original pixel value

10.4.2. GZIP compression

When ZCMPTYPE = 'GZIP_1' the gzip algorithm *shall* be used for data (de)compression. There are no algorithm parameters, so the keywords ZNAME*n* and ZVAL*n should not* appear in the header. The gzip algorithm is used in the free GNU software compression utility of the same name. It was created by J.-L. Gailly and M. Adler, based on the DEFLATE algorithm (Deutsch 1996), which is a combination of LZ77 (Ziv & Lempel 1977) and Huffman coding. The unix `gzip` program accepts an integer parameter that provides a trade between optimization for speed (1) and compression ratio (9), which does not affect the format of the resultant data stream. The selection of this parameter is an implementation detail that is not covered by this Standard.

When ZCMPTYPE = 'GZIP_2' the gzip2 algorithm *shall* be used for data (de)compression. The gzip2 algorithm is a variation on GZIP_1. There are no algorithm parameters, so the keywords ZNAME*n* and ZVAL*n should not* appear in the header. In this case the bytes in the array of data values are shuffled so that they are arranged in order of decreasing significance before being compressed. For example, a 5-element contiguous array of 2-byte (16-bit) integer values, with an original big-endian byte order of:

$$A_1A_2B_1B_2C_1C_2D_1D_2E_1E_2$$

will have the following byte order after shuffling:

$$A_1B_1C_1D_1E_1A_2B_2C_2D_2E_2$$

where A_1, B_1, C_1, D_1, and E_1 are the most significant bytes from each of the integer values. Byte shuffling *shall* only be performed for integer or floating-point numeric data types; logical, bit, and character types *must not* be shuffled.

10.4.3. IRAF/PLIO compression

When ZCMPTYPE = 'PLIO_1' the IRAF PLIO algorithm *shall* be used for data (de)compression. There are no algorithm parameters, so the keywords ZNAME*n* and ZVAL*n should not* appear in the header. The PLIO algorithm was developed to store integer-valued image masks in a compressed form. The compression algorithm used is based on run-length encoding, with the ability to dynamically follow level changes in the image, in principle allowing a 16-bit encoding to be used regardless of the image depth. However, this algorithm has only been implemented in a way that supports image depths of no more than 12 bits; therefore 'PLIO_1' *must* only be used for integer image types with values between 0 and 2^{24}.

The compressed line lists are stored as variable length arrays of type short integer (16 bits per list element), regardless of the mask depth. A line list consists of a series of simple instructions which are executed in sequence to reconstruct a line of the mask. Each 16 bit instruction consists of the sign bit (not used), a three bit opcode, and twelve bits of data, i.e.:

```
+--+--------+-------------------+
|16|15    13|12                1|
+--+--------+-------------------+
|  | opcode |       data        |
+--+-------------------------------+
```

The significance of the data depends upon the instruction. In order to reconstruct a mask line, the application executing these instructions is required to keep track of two values, the current high value and the current position in the output line. The detailed operation of each instruction is given in Table 38.

Table 38: PLIO Instructions

Instr.	Opcode	Meaning
ZN	00	Zero the next N output pixels.
HN	04	Set the next N output pixels to the current high value.
PN	05	Zero the next N-1 output pixels, and set pixel N to the current high value.
SH	05	Set the high value (absolute rather than incremental), taking the high 15 bits from the next word in the instruction stream, and the low 12 bits from the current data value.
IH,DH	02,03	Increment (IH) or decrement (DH) the current high value by the data value. The current position is not affected.
IS,DS	06,07	Increment (IS) or decrement (DS) the current high value by the data value, and step, i.e., output one high value.

The high value *must* be set to 1 at the beginning of a line, hence the IH,DH and IS,DS instructions are not normally needed for Boolean masks.

10.4.4. H-Compress algorithm

When ZCMPTYPE = 'HCOMPRESS_1' the H-compress algorithm *shall* be used for data (de)compression. The algorithm was described by White (1992), and can be applied only to images with two dimensions. Briefly, the compression method is to apply, in order:

1. a wavelet transform called the H-transform (a Haar transform generalized to two dimensions), followed by
2. a quantization that discards noise in the image while retaining the signal on all scales, followed by
3. a quadtree coding of the quantized coefficients.

The H-transform is a two-dimensional generalization of the Haar transform. The H-transform is calculated for an image of size $2^N \times 2^N$ as follows:

1. Divide the image up into blocks of 2×2 pixels. Call the four pixel values in a block a_{00}, a_{10}, a_{01}, and a_{11}.

2. For each block compute four coefficients:
   $h_0 = (a_{11} + a_{10} + a_{01} + a_{00})/(\texttt{SCALE} * \sigma)$
   $h_x = (a_{11} + a_{10} - a_{01} - a_{00})/(\texttt{SCALE} * \sigma)$
   $h_y = (a_{11} - a_{10} + a_{01} - a_{00})/(\texttt{SCALE} * \sigma)$
   $h_c = (a_{11} - a_{10} - a_{01} + a_{00})/(\texttt{SCALE} * \sigma)$
   where SCALE is an algorithm parameter defined below, and σ characterizes the RMS noise in the uncompressed image.
3. Construct a $2^{N-1} \times 2^{N-1}$ image from the h_0 values for each 2×2 block. Divide that image up into 2×2 blocks and repeat the above calculation. Repeat this process N times, reducing the image in size by a factor of 2 at each step, until only one h_0 value remains.

This calculation can be easily inverted to recover the original image from its transform. The transform is exactly reversible using integer arithmetic. Consequently, the program can be used for either lossy or lossless compression, with no special approach needed for the lossless case.

Noise in the original image is still present in the H-transform, however. To compress noisy images, each coefficient can be divided by $\texttt{SCALE} * \sigma$, where $\texttt{SCALE} \sim 1$ is chosen according to how much loss is acceptable. This reduces the noise in the transform to $0.5/\texttt{SCALE}$, so that large portions of the transform are zero (or nearly zero) and the transform is highly compressible.

There is one user-defined parameter associated with the H-Compress algorithm: a scale factor to the RMS noise in the image that determines the amount of compression that can be achieved. It is not necessary to know what scale factor was used when compressing the image in order to uncompress it, but it is still useful to record it. The keywords in Table 39 *should* be recorded in the header for this purpose.

Table 39: Keyword parameters for H-compression

	Values		
Keyword	Permitted	Default	Meaning
ZNAME1	'SCALE'	–	Scale factor
ZVAL1	0.0 or larger	0.0	Scaling of the RMS noise; 0.0 yields lossless compression

Scale Factor – The floating-point scale parameter determines the amount of compression; higher values result in higher compression but with greater loss of information. SCALE = 0.0 is a special case that yields lossless compression, i.e. the decompressed image has exactly the same pixel values as the original image. SCALE > 0.0 leads to lossy compression, where SCALE determines how much of the noise is discarded.