

## 1. The Substring Array Convention for Binary Tables

Note: This convention applies to the TFORMn keyword in binary tables, as discussed in section 7.3.1

### 1.1. Preface

The convention described here for representing arrays of character strings within a character array field in a *FITS* binary table was first described in an appendix to the *FITS* binary table definition paper (Cotton, Tody, & Pence, 1995, *Astron. & Astrophys. Suppl.*, 113, 159), and subsequently in Appendix B of Version 2 of the *FITS* Standard document. This material was removed from Version 3 of the *FITS* Standard, with the expectation that this convention would instead be documented in the Registry of *FITS* Conventions that is maintained by the IAU *FITS* Working Group. Section 2, below, is reproduced nearly verbatim from the above mentioned appendix in the *FITS* Standard with only minor editorial changes.

### 1.2. Convention Definition

This “substring array” convention may be used to specify that a character array field (TFORMn = 'rA') consists of an array of either fixed-length or variable-length substrings within the field. This convention utilizes the option described in the *FITS* Standard to have additional characters following the datatype code character in the TFORMn value field. The full form for the value of TFORMn within this convention is

'rA:SSTRw/nnn'

and a simpler form that may be used for fixed-length substrings only is

'rAw'

where

- r is an integer giving the total length including any delimiters (in characters) of the field,
- A signifies that this is a character array field,
- : indicates that a convention indicator follows,
- SSTR indicates the use of this “Substring Array” convention,
- w is an integer  $\leq r$  giving the (maximum) number of characters in an individual substring (not including the delimiter), and
- /nnn if present, indicates that the substrings have variable-length and are delimited by an ASCII text character with decimal value nnn in the range 032 to 126 decimal, inclusive. This character is referred to as the delimiter character. The delimiter character for the last substring will be an ASCII NUL.

To illustrate this usage:

'40A:SSTR8' signifies that the field is 40 characters wide and consists of an array of 5 8-character fixed-length substrings. This could also be expressed using the simpler form as '40A8'

'100A:SSTR8/032' signifies that the field is 100 characters wide and consists of an array of variable-length substrings where each substring has a maximum length of 8 characters and, except for the last substring, is terminated by an ASCII SPACE (decimal 32) character.

Note that simple *FITS* readers that do not understand this substring convention can ignore the TFORM characters following the rA and can interpret the field simply as a single long string.

The following rules complete the full definition of this convention:

1. In the case of fixed-length substrings, if r is not an integer multiple of w then the remaining odd characters are undefined and should be ignored. For example if TFORMn = '14A:SSTR3', then the field contains 4 3-character substrings followed by 2 undefined characters.
2. Fixed-length substrings must always be padded with blanks if they do not otherwise fill the fixed-length subfield. The ASCII NUL character must not be used to terminate a fixed-length substring field.
3. The character following the delimiter character in variable-length substrings is the first character of the following substring.
4. The method of signifying an undefined or null substring within a fixed-length substring array is not explicitly defined by this convention (note that there is no ambiguity if the variable-length format is used). In most cases it is recommended that a completely blank substring or other adopted convention (e.g. 'INDEF') be used for this purpose although general readers are not expected to recognize these as undefined strings. In cases where it is necessary to make a distinction between a blank, or other, substring and an undefined substring use of variable-length substrings is recommended.
5. Undefined or null variable-length substrings are designated by a zero-length substring, i.e., by a delimiter character (or an ASCII NUL if it is the last substring in the table field) in the first position of the substring. An ASCII NUL in the first character of the table field indicates that the field contains no defined variable-length substrings.
6. Section 7.3 of Version 3 of the *FITS* Standard document discusses a syntax using the TDIMn keyword for describing multidimensional arrays of any datatype which can also be used to represent arrays of fixed-length substrings. For a simple one-dimensional array of substrings (a two-dimensional array of characters) the substring array convention described here is preferred over the “multidimensional array” convention (using the TDIMn keyword). Higher dimensional arrays of (fixed-length) strings cannot be represented using this substring array convention and so require the use of the multidimensional array convention.
7. This substring convention may be used in conjunction with the variable-length array feature in binary tables. In this case, the two possible full forms for the value of the TFORM keyword are

TFORMn = 'rPA(e<sub>max</sub>):SSTRw/nnn'

TFORMn = 'rPA(e<sub>max</sub>):SSTRw'

for the variable and fixed cases, respectively.

### 1.3. Usage Notes

The simpler 'rAw' form of this convention has been supported by the CFITSIO *FITS* interface library (<http://heasarc.gsfc.nasa.gov/fitsio/>) since 1996 and has been used in some publicly distributed *FITS* files produced by various projects. The longer 'rA:SSTRw/nnn' form has rarely been used, and, as far as currently known, never for arrays of variable-length strings.

## 2. Spatial Region File Convention

[need to obtain the Latex source file from Arnold Rots]

### 3. The SIP Convention for Representing Distortion in FITS Image Headers

#### 3.1. Preface

This convention was submitted to the registry by David Shupe and Richard Hook in September 2008. The SIP convention was originally used by the Spitzer Science Center (SSC) in its imaging products. The SIP convention is supported by WCSTOOLS (written by D. Mink), the Starlink AST library, and the IDL ASTROLIB library. Tools that use these libraries (e.g., ds9 and the GAIA Graphical Astronomy and Image Analysis Tool) inherit support for SIP. Also, the drizzle program and related tools developed by STScI and ST-ECF, and the astrometry.net astrometric calibration service use SIP.

#### 3.2. Introduction

The Simple Imaging Polynomial, or SIP, convention provides a straightforward means for storing distortion information in FITS image headers. SIP was initially developed before the launch of the *Spitzer Space Telescope*. Images from the *Spitzer* instruments are distorted by a few percent relative to a regular sky grid. This distortion, expressed as a function of pixel position, is well-represented by polynomials, and it was desired to store the distortion information in the FITS headers of each Basic Calibrated Data (BCD) product. Writing the coefficients for each image was motivated particularly by the optics of the Multiband Imaging Photometer for Spitzer (MIPS) instrument (Rieke et al. 2004)—the distortion changes with scan mirror position, and hence from one image to the next.

The development of the SIP convention proceeded in parallel with work on the World Coordinate System (WCS) FITS standard. The first two papers in this series (Greisen & Calabretta 2002, “Paper I”; and Calabretta & Greisen 2002, “Paper II”) specifying the WCS keywords (sans distortion) have been approved by the IAU FITS Working Group and are now standard. “Paper IV” addressing distortion has been drafted (<http://www.atnf.csiro.au/people/mcalabre/WCS/index.html>) but is not yet final. The SIP keywords are compliant with the first two papers, and have been influenced by early discussions of Paper IV, but are distinct from the proposed keywords in Paper IV.

This document is an expanded version of a paper presented at the 2004 ADASS conference (Shupe et al. 2005). The authors of that paper include the main contributors to the formulation and initial implementation of the SIP convention. The derivation of distortion coefficients for the Spitzer MIPS instrument using this convention are described in Morrison et al. 2007.

#### 3.3. Definitions of the Distortion Keywords

The SIP convention derives its name from the four characters ‘-SIP’ that are appended to the values of CTYPE1 and CTYPE2. These extra characters were included in early drafts of Paper IV to denote the distortion representation; however, later drafts dropped this form. We chose ‘-SIP’ to be distinct from the ‘-PLP’ that was to be used in Paper IV for polynomials, and because it has the useful mnemonic “Simple Imaging Polynomial”.

We define  $u, v$  as relative pixel coordinates with origin at CRPIX1, CRPIX2. Following Paper II,  $x, y$  are “intermediate world coordinates” in degrees, with origin at CRVAL1, CRVAL2.

Let  $f(u, v)$  and  $g(u, v)$  be the quadratic and higher-order terms of the distortion polynomial. Then

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \text{CD1}_1 & \text{CD1}_2 \\ \text{CD2}_1 & \text{CD2}_2 \end{pmatrix} \begin{pmatrix} u + f(u, v) \\ v + g(u, v) \end{pmatrix} \quad (1)$$

We define  $A_{p,q}$  and  $B_{p,q}$  as the polynomial coefficients for polynomial terms  $u^p v^q$ . Then

$$f(u, v) = \sum_{p,q} A_{p,q} u^p v^q, \quad p + q \leq \text{A\_ORDER}, \quad (2)$$

$$g(u, v) = \sum_{p,q} B_{p,q} u^p v^q, \quad p + q \leq \text{B\_ORDER}. \quad (3)$$

For example, for a third-order polynomial,

$$f(u, v) = \text{A}_2\text{0}u^2 + \text{A}_0\text{2}v^2 + \text{A}_1\text{1}uv + \text{A}_2\text{1}u^2v + \text{A}_1\text{2}uv^2 + \text{A}_3\text{0}u^3 + \dots$$

A\_ORDER and B\_ORDER can take on integer values ranging from 2 to 9.

The  $\text{CD}i_j$  keywords encode skew as well as rotation and scaling. The CD-matrix values together with the higher-order distortion polynomials, as in Equations 1, 2, and 3, define a unique transformation from pixel coordinates to the plane-of-projection.

For Spitzer, we also provide polynomials for the reverse transformation, for fast inversion. Corrected pixel coordinates  $U, V$  are found from

$$\begin{pmatrix} U \\ V \end{pmatrix} = \text{CD}^{-1} \begin{pmatrix} x \\ y \end{pmatrix} \quad (4)$$

then the original pixel coordinates are computed by

$$u = U + F(U, V) = U + \sum_{p,q} \text{AP}_{p,q} U^p V^q, \quad p + q \leq \text{AP\_ORDER}, \quad (5)$$

$$v = V + G(U, V) = V + \sum_{p,q} \text{BP}_{p,q} U^p V^q, \quad p + q \leq \text{BP\_ORDER}. \quad (6)$$

To make a reasonably accurate reverse transformation, in general it is necessary to include linear terms in the reverse coefficients.

Finally, borrowing another idea from a Paper IV draft, the values of the keywords A\_DMAX and B\_DMAX give bounds on the maximum distortion over the array. These optional keywords could be used to estimate the maximum error that would result from not evaluating the distortion polynomial.

#### 3.4. Example: Spitzer-IRAC Channel 4

We take as an example the distortion of the *Spitzer* Infrared Array Camera (IRAC) instrument (Fazio et al. 2004), which is characterized by cubic coefficients. Polynomial distortion of this form (plus linear terms) was fit to *Spitzer* data from the Great Observatories Origins Deep Survey program (S. Casertano, private communication). The linear terms are folded into the  $\text{CD}i_j$ . An excerpt from an actual BCD header produced by the Spitzer pipeline for IRAC Channel 4 is shown below.

```
CTYPE1 = 'RA---TAN-SIP'           / RA---TAN with distortion
CTYPE2 = 'DEC--TAN-SIP'           / DEC--TAN with distortion
CRVAL1 =      202.581507417836    / [deg] RA at CRPIX1,CRPIX2
CRVAL2 =      47.2465528124827    / [deg] DEC at CRPIX1,CRPIX2
CRPIX1 =                                     128. / Reference pixel along axis
```

Keywords 1	Value	Keywords 2	Value	Keyword	Value	
CRPIX2 =	128.	Reference pixel along axis 2				
CD1_1 =	0.000248349650353678	Corrected CD matrix element with Pointing Record		CTYPE2	'DEC--TAN-SIP'	
CD1_2 =	0.000232107213140475	Corrected CD matrix element with Pointing Record		CRPIX2	1024.0	
CD2_1 =	0.000232418393583541	Corrected CD matrix element with Pointing Record		CRVAL2	-72.076963036772	
CD2_2 =	-0.000246562617306562	Corrected CD matrix element with Pointing Record		CD2_1	1.1406694624771E-05	
A_ORDER =	3	polynomial order, axis 1, detector to sky		CD2_2	8.6942510845452E-06	
A_0_2 =	9.0886E-06	distortion coefficient	A_0_2	2.4634068532689E-06	B_0_2	-7.2299995118730E-06
A_0_3 =	4.8066E-09	distortion coefficient	A_1_0	-5.194753640575E-06	B_1_1	6.1778338717084E-06
A_1_1 =	4.8146E-05	distortion coefficient	A_1_1	8.543473309812E-06	B_2_0	-1.7442694174934E-06
A_1_2 =	-1.7096E-07	distortion coefficient	A_1_2	1.0622437604068E-11	B_0_3	-4.2102920235938E-10
A_2_0 =	2.82E-05	distortion coefficient	A_2_1	-5.2797808038221E-10	B_1_2	-6.7603466821178E-11
A_2_1 =	3.3336E-08	distortion coefficient	A_2_2	-4.4012735467525E-11	B_2_1	-5.1333879897858E-10
A_3_0 =	-1.8684E-07	distortion coefficient	A_3_0	-4.7518233007536E-10	B_3_0	8.5722142612681E-11
A_DMAX =	2.146	[pixel] maximum correction	A_3_1	1.4075878614807E-14	B_0_4	6.5531313110898E-16
B_ORDER =	3	polynomial order, axis 2, detector to sky	A_3_2	-1.9317154005522E-14	B_1_3	1.3892905568706E-14
B_0_2 =	4.1248E-05	distortion coefficient	A_2_2	3.767898933666E-14	B_2_2	-2.9648166208490E-14
B_0_3 =	-1.9016E-07	distortion coefficient	A_3_1	5.086953083043E-15	B_3_1	-2.0749495718513E-15
B_1_1 =	1.4761E-05	distortion coefficient	A_3_2	2.5776347115304E-14	B_4_0	-1.812610418272E-14
B_1_2 =	2.1973E-08	distortion coefficient	A_ORDER		B_ORDER	4
B_2_0 =	-6.4708E-06	distortion coefficient				
B_2_1 =	-1.8188E-07	distortion coefficient				
B_3_0 =	1.0084E-10	distortion coefficient				
B_DMAX =	1.606	[pixel] maximum correction				
AP_ORDER =	3	polynomial order, axis 1, sky to detector				
AP_0_1 =	3.6698E-06	distortion coefficient				
AP_0_2 =	-9.1825E-06	distortion coefficient				
AP_0_3 =	-3.8909E-09	distortion coefficient				
AP_1_0 =	-2.0239E-05	distortion coefficient				
AP_1_1 =	-4.8946E-05	distortion coefficient				
AP_1_2 =	1.7951E-07	distortion coefficient				
AP_2_0 =	-2.8622E-05	distortion coefficient				
AP_2_1 =	-2.9553E-08	distortion coefficient				
AP_3_0 =	1.9119E-07	distortion coefficient				
BP_ORDER =	3	polynomial order, axis 2, sky to detector				
BP_0_1 =	-2.1339E-05	distortion coefficient				
BP_0_2 =	-4.189E-05	distortion coefficient				
BP_0_3 =	1.9696E-07	distortion coefficient				
BP_1_0 =	2.8502E-06	distortion coefficient				
BP_1_1 =	-1.5089E-05	distortion coefficient				
BP_1_2 =	-2.0219E-08	distortion coefficient				
BP_2_0 =	6.4625E-06	distortion coefficient				
BP_2_1 =	1.849E-07	distortion coefficient				
BP_3_0 =	-7.6669E-10	distortion coefficient				

Table 1. SIP coefficients for the Hubble ACS Wide Field Channel.

In this case, the reverse coefficients have the opposite sign and roughly the same absolute values as the corresponding forward coefficients. However, this is not true for some more distorted fields of view, so the Spitzer headers retain the reverse coefficients in general.

The Spitzer Science Center has developed library routines to implement this coefficient naming scheme. The functions key off the extended CTYPEn. The order in which the keywords are displayed in the example is the order in which the software searches for them and is the most efficient for lookups using CFITSIO.

### 3.5. Software that Reads and Applies the Coefficients

The usefulness of this convention was greatly enhanced by the generous efforts of a number of individuals who added support to their software before the first release of *Spitzer* data in 2004. The mosaicking package MOPEX (Makovoz & Khan 2005) ap-

plies the SIP distortion coefficients in the Spitzer Science Center pipeline. Support has also been added to IPAC's Skyview display program (<http://www.ipac.caltech.edu/Skyview/>). Doug Mink implemented Spitzer distortion support in his WCS routines (<http://tdc-www.harvard.edu/software/wcstools>). SAOImage and DS9 use these routines and hence automatically handle the SIP distortions. The Montage software (<http://montage.ipac.caltech.edu>) (Laity et al. 2004) also uses Mink's routines and applies the coefficients. Support in the CIAA viewer has been added via David Berry's ASTROLIB. The Landsman has added support to the IDL ASTROLIB. The Drizzle software (Fruchter & Hook 2002) has also been modified to read the coefficients.

Finally we note that the astrometry.net service also uses the SIP convention for encoding the non-linear parts of the distortions it calculates in arbitrary images.

### 3.6. SIP for Hubble

Of the cameras currently on board the Hubble Space Telescope, the distortion is largest by far for the Wide Field Channel (WFC) of the Advanced Camera for Surveys (ACS) where it amounts to more than fifty pixels at the corner of the image in addition to an even larger (linear) skew term. The newer Wide Field Camera 3, to be installed in October 2008, has similarly large distortions.

The image distortion for Hubble cameras is currently characterized by a FITS table known as the Image Distortion Correction Table (IDCTAB) that includes information about the scale and orientation of the instrument aperture in the telescope focal plane as well as the distortion polynomial coefficients. Software has been developed that will combine the IDCTAB information with the normal information from the telescope's pointing control software to write out a header which makes the header WCS keywords fully compatible with the table values and also populates the SIP-keywords (or at least the most important ones). An example of the resultant header is given in Table 1.

Currently the writing of these SIP keywords is an unsupported feature for Hubble data. However, it is planned to formally include such headers in future to provide users with a full,

self-describing distortion model without the need for access to external files in non-standard formats. The software to read the coefficients and apply them to remove image distortion already exists within the standard Hubble data processing tools.

### 3.7. *Issues and Caveats*

The SIP convention has been in use for several years and is becoming more widespread. To gauge feelings about it we recently have asked for comments from several people who have used it for and are familiar with its features. We are grateful for their input and time. In this section we summarize some possible limitations of the standard.

The SIP specification provides for “reverse” coefficients to allow the mapping of sky coordinates to pixels to be performed rapidly, without the need for iterative inversion techniques. The Spitzer mosaicking tool MOPEX relies on the reverse coefficients for its default interpolation mode, as it maps output pixel corners back to the original distorted images. The reverse polynomial is, however, only an approximation and in general cannot be the exact inverse of the forward polynomial. As a result mapping a pixel to celestial coordinates and back does not yield back precisely the original coordinates. For example, in the IRAC channel 4 distortion listed above, mapping pixel coordinate (1.0,1.0) to the sky and back leads to a difference of about 0.014 pixels. It can be argued that such a difference is negligible for practical applications, and that distortions may not be measured to such accuracy anyway. Another drawback to the reverse coefficients is that they violate the principle of storing only the minimum information necessary in the FITS header – the forward coefficients could be considered to contain all the necessary information.

A better approach, in hindsight, might have been to not include the “reverse” coefficients at all, but instead to invert the forward solution using an iterative technique. For an arbitrary polynomial, it might not be possible to guarantee convergence of the inversion. For practical applications to distorted images, however, the size of the corrections are small and inversion will likely work well. It should be noted that the reverse coefficients for the examples given above are very nearly the same as the forward coefficients with the sign reversed. The starting points for any iterative inversion are well-determined and the solution should be reached rapidly.

Based on these considerations, use of the reverse coefficients should be considered optional although this may create problems for existing tools such as WCSTools which have already been coded to use the reverse coefficients.

Another area of concern concerns possible loss of accuracy in the calculations under some circumstances. In the case of large pixel coordinate values, and high order polynomials, the terms can grow large. In some cases it is necessary to take the differences between polynomial terms that are much larger than the final result — a classic case where accuracy can be lost. It clearly helps significantly to use double precision floating point numbers and hence around 15 significant figures of accuracy. We would strongly discourage the use of single precision but it remains a question for software developers and is not imposed by the convention itself. A sufficient number of significant digits should be used to specify the distortion coefficients in the FITS header. As shown in the examples above, the high-order coeffi-

cients become small and must be specified in scientific notation with large negative exponents.

An alternative solution to avoid loss of accuracy, or overflow or underflow problems, is to introduce a scaling term so that pixel values are scaled into the range  $-1$  to  $+1$ . This could quite easily be done with by adding an optional keyword, that defaults to 1.0 in order not to invalidate existing headers.

Another comment we have received is that the form of the keywords is relatively simple, so much so that someone else might accidentally use one of the distortion keywords for another purpose elsewhere in the FITS header, thereby corrupting the distortion information.

### 3.8. *Possible New Features*

We have also asked for views about possible extensions to SIP.

One limitation of this current convention is that only regular polynomials are allowed – not Chebyshev or Lagrange for example. As a result higher-order polynomials can diverge at the edges of images where they are less well constrained and this could cause difficulties under some conditions. However, adding these would be significant work and we do not think it should be considered at present. If these were implemented, we would recommend a different three-letter suffix for CTYPE1 and CTYPE2, or some other means to maintain a distinction from the simple polynomials currently used.

In the current convention the distortion origin is forced to be at (CRPIX1, CRPIX2). However, in many cases it is natural for the distortion center to lie at a different location. It has been suggested that additional keywords could be used to specify the distortion center, and if these are not present then the default of (CRPIX1, CRPIX2) is used. Although the introduction of a different center may have advantages there are also significant drawbacks and some of the simplicity of the original scheme is lost. In particular, the CD matrix would no longer contain a correct description of pixel scales, skew and orientation at the point (CRPIX1, CRPIX2). We note that it is always possible to exactly shift the distortion origin to (CRPIX1, CRPIX2) with the result as a polynomial of the same order, although it is possible in extreme cases that this will result in much larger terms and possible consequent accuracy loss.

### 3.9. *Concluding Remarks*

The SIP convention has proved to be applicable to many imaging situations and its simplicity has made implementation and use easy. Many people feel it is the natural solution without excess detail. However, this simplicity naturally limits its generality and largely restricts the applicability of SIP to simple cameras, unlike the much more extensive general proposals in FITS Paper 4 that include multi-dimensional support that cover far more cases beyond just simple imaging.

### 3.10. *Acknowledgments*

We thank Mehrdad Moshir and Bob Narron for their contributions to the development of the SIP keywords. We are grateful to Mark Calabretta for significant comments and suggestions, and Jane Morrison for discussions of MIPS distortions and the CD matrix. We thank Doug Mink, Wayne Landsman, David Berry, and Booth Hartley for implementing SIP in their software.

We are also very grateful to all those who provided helpful replies to our requests for comments. In particular Stefano Casertano, Jane Morrison, Emmanuel Bertin, David Berry and Mark Lacy provided much interesting feedback.

The work carried out at the Spitzer Science Center was funded by NASA under contract 1407 to the California Institute of Technology and the Jet Propulsion Laboratory.

### 3.11. References

Calabretta, M.R., & Greisen, E.W. 2002, *A&A*, 395, 1077 (Paper II).

Fazio, G., et al. 2004, *ApJ Suppl.*, 154, 10.

Fruchter, A.S. & Hook, R.N. 2002, *PASP*, 114, 144

Greisen, E.W., & Calabretta, M.R. 2002, *A&A*, 395, 1061 (Paper I).

Laity, A.C., Anagnostou, N., Berriman, B., Good, J.C., Jacob, J.C., & Katz, D.S. 2005, "Montage: An Astronomical Image Mosaic Service for the NVO," in "Astronomical Data Analysis Software and Systems XIV ASP Conference Series", ed. by P. Shopbell, M. Britton, and R. Ebert (San Francisco: Astronomical Society of the Pacific), vol 347, p. 34.

Makovoz, D., & Khan, I. 2005, "Mosaicking with MOPEX," in "Astronomical Data Analysis Software and Systems XIV ASP Conference Series", ed. by P. Shopbell, M. Britton, and R. Ebert (San Francisco: Astronomical Society of the Pacific), vol 347, p. 81.

Morrison, J.E., Stamper, B.L., & Shupe, D.L. 2007, "Correcting MIPS Spitzer Images for Distortion," in "Astronomical Data Analysis Software and Systems XVI ASP Conference Series", ed. by R.A. Shaw, F. Hill and D.J. Bell, Vol 376, p. 433.

Rieke, G., et al. 2004, *ApJ Supp*, 154, 25.

Shupe, D.L., Moshir, M., Li, J., Makovoz, D., Narron, R., & Hook, R.N. 2005, "The SIP Convention for Representing Distortion in FITS Image Headers", in "Astronomical Data Analysis Software and Systems XIV ASP Conference Series", ed. by P. Shopbell, M. Britton, and R. Ebert (San Francisco: Astronomical Society of the Pacific), vol 347, p. 491.

## 4. A Convention for preallocating header space for FITS keywords

controlled environment where all the software is known to support this convention

### 4.1. Preface

This convention has been supported by the CFITSIO library since approximately 1996 and has primarily been used within the FITS data files produced by high energy astrophysics missions supported by the HEASARC.

### 4.2. Background

The ASCII header in every FITS HDU (Header Data Unit) consists of 1 or more 2880-byte blocks, each of which can hold 36 80-byte keyword records. When writing a new keyword to the header of a FITS file, if the header is full (i.e., the last header block already contains 36 header records, including the END keyword) then it becomes necessary to insert a new 2880-byte FITS block at the end of the header. This in turn requires that any data in the FITS file following the header be shifted down by 2880 bytes in the file to make room for the inserted block. This rewriting operation can cause significant data processing inefficiencies when dealing with large FITS files.

One way to circumvent this problem is to preallocate enough space in the header when the FITS HDU is created to hold the anticipated number of keywords that may be written during later processing of the FITS file. This document describes a simple convention for creating an arbitrarily large amount of reserve space in the header that can be used when writing new keywords.

### 4.3. Convention details

Under this convention, any sequence of one or more completely blank keyword records (consisting of 80 ASCII space characters) immediately preceding the END keyword are interpreted as non-significant scratch space, which can be reused when new keywords are written. The *functional* end of the header is defined as located at the beginning of this scratch space area, which is where each new keyword record is written. In the event that all the scratch space becomes filled with keywords, then the traditional procedure of shifting the END keyword down one space in the header to make room for the new keyword should be followed.

In practice, it is usually most convenient to write the desired number of blank keywords into the header just prior to writing the END keyword itself, before writing any actual data records to the FITS HDU. Software that recognizes this convention should then reuse these blank records when writing new keywords to the header. This eliminates the inefficiencies associated with having to insert a new FITS block into an existing FITS file to make room for more keywords. Even if some of this reserved header space remains unused (note that space for 100 keywords only occupies 8K of disk space), this is usually insignificant when dealing with large FITS files.

It should be noted that if a FITS file is processed by software that does not support this convention, then new keywords may be written at the location of the END keyword (i.e., *after* the blank keyword records). This will make the blank keywords unavailable for future use by software that does support this convention and will create what appears to be a gap of blank header keywords in the header. For this reason it may be safest to use this convention on FITS files that are created and processed within a

## 5. TNX World Coordinate System

### 5.1. Preface

This convention was submitted to the registry in September 2009 by Doug Tody, Lindsey Davis, and Frank Valdes. The TNX convention is currently used in the FITS files produced by NOAO that are taken with a variety of different imaging instruments including MOSAIC and NEWFIRM since 1998.

The TNX projection is evaluated as follows.

1. Compute the first order standard coordinates xi and eta and the linear part of the solution stored in CRPIX and the CD keywords.

$$\begin{aligned} xi &= CD1\_1 * (x - CRPIX1) + CD1\_2 * (y - CRPIX2) \\ eta &= CD2\_1 * (x - CRPIX1) + CD2\_2 * (y - CRPIX2) \end{aligned}$$

[This document is copied from iraf.net <<http://iraf.net>> Add the non-linear part of the projection using the CD keywords and the WAT keywords as described below.

The TNX World Coordinate System is a non-standard system for evaluating celestial coordinates from the image pixel coordinates. It follows the FITS conventions for undistorted tangent plane projections but adds a non-linear distortion term to the evaluation. This discussion concentrates on the non-linear extension and assumes that the standard tangent plane projection to xi' and eta' is understood. The FITS WCS conventions including applying the CRVAL values as the tangent point to get the RA and Dec projection. The reference for the FITS WCS standard for celestial coordinates systems is Representations of celestial coordinates in FITS.

The non-linear part of the projection is evaluated as follows:  $xi = xi' + lngcor(xi, eta)$  and  $eta = eta' + latcor(xi, eta)$ . The lngcor and latcor functions are defined with coefficients stored as FITS keywords indexed WATj\_nnn keywords. The j refers to the image axis number. The cards for a particular keyword are stored by the sequence number and then concatenated together into a single string. The long string for each image axis is composed of a set of keyword/value pairs where the value is quoted if it contains a space. Figure 2 shows the how the WAT keywords in figure 1 would be expanded into parameters and coefficients.

The TNX World Coordinate System projection has a FITS keyword representation as illustrated in figure 1.

Figure 1: Sample header with TNX WCS projection

```
WCSASTRM= 'ct4m.19990714T012701 (USNO-K V) by F. Valdes 1995'
WCSDIM = 2 / WCS dimensionality
CTYPE1 = 'RA---TNX' / Coordinate type AXIS 1
CTYPE2 = 'DEC--TNX' / Coordinate type AXIS 2
CRVAL1 = 310.08145293602507 / Coordinate reference value
CRVAL2 = 20.663666538998399 / Coordinate reference value
CRPIX1 = 4268.3258 / Coordinate reference pixel
CRPIX2 = 2256.2481 / Coordinate reference pixel
CD1_1 = -6.8295807e-08 / Coordinate matrix 4. 4.
CD2_1 = 7.3313414e-05 / Coordinate matrix 4. 4.
CD1_2 = 7.374228e-05 / Coordinate matrix 2. 2.
CD2_2 = -1.1927219e-06 / Coordinate matrix -0.3171856965643079 -0.3171856965643079
WAT0_001= 'system=image' / Coordinate system -0.0150652479325533 -0.0150652479325533
WAT1_001= 'wtype=tnx axtype=ra lngcor = "3. 4. 4. 2. -0.0131860983043609660-0.13126038394350166
WAT1_002= '0652479325533 -0.3126038394350166 -0.151195500928391504002838100964511955040928311
WAT1_003= '838772 0.01749134520424022 -0.010827844230201030003188090606838872340:005534819578784082
WAT1_004= '-4.307309762939804E-4 0.009069288008295441 0.008742632780424022-00001258790793029932
WAT1_005= '4487658756007625 -0.1058043162287004 -0.0686204060827884223020123 0.01016780085575339
WAT2_001= 'wtype=tnx axtype=dec latcor = "3. 4. 2. -0031388960638460294-00001541083298696018
WAT2_002= '50652479325533 -0.3126038394350166 -0.1511955040928391760960894484950:03531979587941362
WAT2_003= '8784082 0.01258790793029932 0.01016780085575339009069288092895460180:0150096457430599
WAT2_004= '0.03531979587941362 0.0150096457430599 -0.1080400282759522480505994800:1086479352595234
WAT2_005= '086902122 0.02341002785565408 -0.07773808393244084787658756007625 0.0399806086902122
-0.1058043162287004 0.02341002785565408
-0.07773808393244387 -0.07773808393244387
```

The WCSASTRM keyword is just for documentation. The WCSDIM keyword always be 2. That this is a TNX projection is indicated by the CTYPE keywords. These keywords also indicate that the first image axis corresponds to RA and the second to DEC.

- The first number is the function type encoded as 1=chebyshev, 2=legendre, 3=polynomial. The example has a function of type  $P_m(x) * P_n(\eta)$  (chebyshev) is the simple polynomial.
- The next two numbers represent the "order" of the function in  $x$  and  $\eta$ . The order is the one less than the highest polynomial power. The powers are represented below by  $m$  and  $n$  such as  $P_{m+1}(x) = 2.0 * x * P_m(x) - P_{m-1}(x)$  where  $m=1$  to  $x$  order-1 and  $n = 0$  to  $\eta$  order-1. In the example the orders are 4 which means cubic polynomials ( $m=0$  to 3 and  $n=0$  to 3)  $P_0(\eta) = 1.0$
- The next (fourth) number specifies the type of cross-terms as 0=no cross-terms, 1=full cross-terms, 2=half-cross-terms. The 2.0 \*  $\eta * P_n(\eta) - P_{n-1}(\eta)$  cross-terms are terms of  $x^m * \eta^n$  where  $m$  and  $n$  are non-zero. Full cross-terms mean that both  $m$  and  $n$  will go to the their maximum values independently while half-cross terms mean that  $m + n$  will only go to the maximum of  $x$  order-1 and  $\eta$  order-1.  $P_0(x) = 1.0$
- The next 4 numbers describe the region of validity of the fit in  $x$  and  $\eta$  space, e.g.  $x_{min}$ ,  $x_{max}$ ,  $\eta_{min}$ ,  $\eta_{max}$ . The fit is computed by summing polynomial terms  $P_{m+1}(x) * P_n(\eta) - m * P_{m-1}(x)$  and  $P_{n+1}(\eta) = ((2n+1) * \eta * P_n(\eta) - n * P_{n-1}(\eta))$  compute normalized values for  $x$  and  $\eta$  used in the chebyshev and legendre polynomial functions:
 
$$x_{in} = (2 * x - (x_{max} + x_{min})) / (x_{max} - x_{min})$$

$$\eta_{in} = (2 * \eta - (\eta_{max} + \eta_{min})) / (\eta_{max} - \eta_{min})$$
- The remaining terms are the coefficients of the polynomial as follows. The functions are evaluated by summing polynomial terms  $P_{m+1}(x) * P_n(\eta) - m * P_{m-1}(x)$  multiplied by the coefficients  $C_{mn}$  as
 
$$f(x, \eta) = \sum (C_{mn} * P_{m+1}(x) * P_n(\eta) - m * P_{m-1}(x))$$

Representing the coefficients as  $C_{mn}$  for the polynomials  $P_{mn}$ , where  $m$  and  $n$  are the powers of  $x$  and  $\eta$ , they are ordered as

- C00
- C10
- C20
- C30
- ...
- C01
- C11
- C21
- C31
- ...
- C02
- C12
- C22
- C32
- ...
- C03
- C13
- C23
- C33
- ...

In the example with the half cross-terms and orders of 4 the ten coefficients would be C00, C10, C20, C30, C01, C11, C21, C02, C12, and C03.

The polynomials  $P_{mn}$  are defined below. The chebyshev and legendre polynomials are define iteratively as functions of the normalized coordinates defined earlier.

$$P_{mn} = x ** m * \eta ** n \quad (\text{polynomial})$$

## 6. TPV World Coordinate System

### 6.1. Preface

This convention was submitted to the registry in September 2011 by Francisco Valdes.

The TPV World Coordinate System is a non-standard convention following the rules of the WCS standard. It builds on the standard TAN projection by adding a general polynomial distortion correction. The description here covers the application of the distortion function and assumes the reader understands the FITS WCS rules including applying the linear transformation to intermediate coordinates and applying the tangent plane projection to the distortion corrected intermediate coordinates. The reference for the FITS WCS standard for undistorted celestial systems is Representations of celestial coordinates in FITS <[http://adsabs.harvard.edu/cgi-bin/nph-bib\\_query?bibcode=2002ApJ...577CMB...457Calabretta](http://adsabs.harvard.edu/cgi-bin/nph-bib_query?bibcode=2002ApJ...577CMB...457Calabretta), M. R., and Greisen, E. W., Astronomy & Astrophysics 1077-1122, 2002. (PDF <[http://adsabs.harvard.edu/cgi-bin/nph-data\\_query?bibcode=2002ApJ...577CMB...457Calabretta](http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2002ApJ...577CMB...457Calabretta)> HTML <[http://adsabs.harvard.edu/cgi-bin/nph-data\\_query?bibcode=2002ApJ...577CMB...457Calabretta](http://adsabs.harvard.edu/cgi-bin/nph-data_query?bibcode=2002ApJ...577CMB...457Calabretta)> Reprints are available from the author's web site <<http://www.atnf.csiro.au/~mcalabre/>> in PDF <<http://www.atnf.csiro.au/people/mcalabre/WCS/ccs.pdf>> format.

Historically this WCS derives from an earlier proposal by Greisen <<http://astromatic.net/forum/attachment.php?attachmentid=10>> for the celestial coordinates paper. In that proposal the WCS would be extended with the optional distortion polynomial up to the code to identify the presence of the PV keywords and apply distortion. Since the final standard does not provide a standard TAN projection to the tangent plane projection, the proposal is recast by simply defining a new WCS identifier and publishing it as `WORLD` here.

### Evaluation Steps

The TPV projection is evaluated as follows.

1. Compute the first order standard coordinates xi and eta from the linear part of the solution stored in CRPIX and the CD matrix.

$$\begin{aligned} xi &= CD1\_1 * (x - CRPIX1) + CD1\_2 * (y - CRPIX2) \\ eta &= CD2\_1 * (x - CRPIX1) + CD2\_2 * (y - CRPIX2) \end{aligned}$$

2. Apply the distortion transformation using the coefficients in the PV keywords as described below.

$$\begin{aligned} xi' &= f\_xi(xi, eta) \\ eta' &= f\_eta(xi, eta) \end{aligned}$$

3. Apply the tangent plane projection to xi' and eta' as described in /Calabretta and Greisen/. The reference tangent point given by the CRVAL values lead to the final RA and DEC in degrees. Note that the units of xi, eta, f\_xi, and f\_eta are also degrees.

### Distortion Functions

The distortion functions shown as f\_xi and f\_eta above are defined as

follows where the variable r is sqrt(xi^2+eta^2). In this there are only odd powers of r.

$$\begin{aligned} xi' &= PV1\_0 + PV1\_1 * xi + PV1\_2 * eta + PV1\_3 * r + \\ &PV1\_4 * xi^2 + PV1\_5 * xi * eta + PV1\_6 * eta^2 + \\ &PV1\_7 * xi^3 + PV1\_8 * xi^2 * eta + PV1\_9 * xi * \\ &PV1\_12 * xi^4 + PV1\_13 * xi^3 * eta + PV1\_14 * \\ &PV1\_17 * xi^4 * eta + PV1\_18 * xi^3 * eta^2 + \\ &PV1\_19 * xi^2 * eta^3 + PV1\_21 * xi * eta^4 + PV1\_22 * \\ &PV1\_23 * xi^6 + PV1\_25 * xi^5 * eta + PV1\_26 * \\ &PV1\_28 * xi^5 * eta^2 + PV1\_29 * xi * eta^5 + PV1\_30 * \\ &PV1\_31 * xi^7 + PV1\_32 * xi^6 * eta + PV1\_33 * \\ &PV1\_34 * xi^6 * eta^2 + PV1\_35 * xi^5 * eta^3 + \\ &PV1\_36 * xi^2 * eta^5 + PV1\_37 * \\ &PV2\_1 * eta + PV2\_2 * xi + PV2\_3 * r + \\ &PV2\_4 * eta^2 + PV2\_5 * eta * xi + PV2\_6 * xi^2 + \\ &PV2\_7 * xi^3 + PV2\_8 * xi^2 * eta + PV2\_9 * xi * \\ &PV2\_10 * eta^2 + PV2\_11 * xi * eta^3 + PV2\_12 * \\ &PV2\_13 * eta^4 + PV2\_14 * xi^4 + PV2\_15 * xi^3 * \\ &PV2\_16 * xi^2 * eta^4 + PV2\_17 * xi * eta^5 + \\ &PV2\_18 * eta^5 + PV2\_19 * xi^5 + PV2\_20 * xi^4 * \\ &PV2\_21 * xi^3 * eta^5 + PV2\_22 * xi^2 * eta^6 + \\ &PV2\_23 * xi * eta^7 + PV2\_24 * eta^6 * xi + PV2\_25 * \\ &PV2\_26 * eta^7 + PV2\_27 * eta^6 * xi^2 + PV2\_28 * \\ &PV2\_29 * eta^5 * xi^4 + PV2\_30 * eta^4 * xi^5 + \\ &PV2\_31 * eta^7 + PV2\_32 * eta^6 * xi + PV2\_33 * \\ &PV2\_34 * eta^5 * xi^3 + PV2\_35 * eta^3 * xi^4 + \\ &PV2\_36 * eta^2 * xi^5 + PV2\_37 * \end{aligned}$$

Note that missing PV keywords default to 0 except for PV1\_39 which defaults to 1. With these defaults if there are no PV keywords the identity and the TPV WCS is equivalent to the standard TAN projection. Also the defaults mean that the polynomial coefficients to the order desired. Similarly, the PV keywords and apply only to terms in powers of r which then mimic a standard TAN projection. The convention only defines coefficients up to 39 corresponding to a maximum polynomial order of 7.

To implement the inverse transformation requires inverting the distortion functions. But using a standard iterative numerical method based on the first derivative of the functions is not difficult since the derivatives of these functions are straightforward to express.

## 7. ZPX World Coordinate System

### 7.1. Preface

This convention was submitted to the registry in September 2011 by Frank Valdes.

The WCSAXES keyword (possible seen as WCSDIM) will always this is a ZPX projection is indicated by the CTYPE keyword keywords also indicate that the first image axis correspond the second to DEC.

The ZPX projection is evaluated as follows.

The ZPX World Coordinate System is a non-standard system for evaluating celestial coordinates from the image pixel coordinates. It follows the FITS conventions for a zenithal polynomial projection (ZPN) but adds an additional two-dimensional polynomial distortion term to the xi = CD1\_1 \* (x - CRPIX1) + CD1\_2 \* (y - CRPIX1) + CD2\_1 \* (x - CRPIX1) + CD2\_2 \* (y - CRPIX1) evaluation. This discussion concentrates on the non-ZPN distortion extension and assumes the reader understands the FITS WCS conventions including applying a zenithal polynomial projection. The addition of the non-linear part of the projection using the FITS WCS standard for undistorted celestial coordinates is described below.

Representations of celestial coordinates in FITS  
<http://adsabs.harvard.edu/cgi-bin/nph-bib\_query?bibcode=2002A&26A...395...1077cd\_b\_kj=Astrophysica Calabretta, M. R., and Greisen, E. W., Astronomy & Astrophysics, 395, 1077-1122, 2002. (PDF)  
<http://adsabs.harvard.edu/cgi-bin/nph-data\_query?bibcode=2002A&26A...395...1077cd\_b\_kj=Astrophysica Calabretta, M. R., and Greisen, E. W., Astronomy & Astrophysics, 395, 1077-1122, 2002. (HTML)  
<http://adsabs.harvard.edu/cgi-bin/nph-data\_query?bibcode=2002A&26A...395...1077cd\_b\_kj=Astrophysica Calabretta, M. R., and Greisen, E. W., Astronomy & Astrophysics, 395, 1077-1122, 2002. (PDF)  
Reprints are available from the author's web site  
<http://www.atnf.csiro.au/~mcalabre/> in PDF  
<http://www.atnf.csiro.au/people/mcalabre/WCS/ccs.pdf>

One thing to note is that generally the radial polynomial distortion functions lngcor(xi,eta) included in the ZPN projection is generally fixed and is described by known optical design terms. For NOAO Mosaic data, each calibration exposure has different distortion terms and the nnn give the number of coefficients for a particular image axis and the concatenated together into one long string for the radial polynomial coefficients.

The ZPN World Coordinate System projection has a FITS keyword representation as illustrated in figure 1.

The coefficients for the zenithal polynomial (the P/m/) are described in /Calabretta and Greisen/ where the P/m/ coefficients are described in the type=PPOLYN keyword. The reference tangent point given by the CRVAL1 and CRVAL2 keywords. Note that the unit for CRVAL1, CRVAL2, CRPIX1, and CRPIX2 are degrees. Note that the unit for CRVAL1, CRVAL2, CRPIX1, and CRPIX2 are degrees.

Figure 1: Sample header with ZPX WCS projection

```
WCSAXES = 2 / Number of WCS axes
CTYPE1 = 'RA---ZPX' / Coordinate type
CTYPE2 = 'DEC--ZPX' / Coordinate type
CRVAL1 = 320.687374999995 / Coordinate reference value
CRVAL2 = 36.908555555556 / Coordinate reference value
CRPIX1 = 4167.56175625891 / Coordinate reference pixel
CRPIX2 = 4120.25894749731 / Coordinate reference pixel
CD1_1 = -5.2588308681025E-8 / Coordinate matrix
CD2_1 = -7.2772379161132E-5 / Coordinate matrix
CD1_2 = -7.2753930850119E-5 / Coordinate matrix
CD2_2 = -1.8637632244742E-8 / Coordinate matrix
WAT0_001= 'system=image' / Coordinate system
WAT1_001= 'wtype=zpx axtype=ra projp0=0. projp1=1. projp2=0. projp3=337.74 projp4=0. projp5=632052. lngcor = "3. 3. 3. 2. 0.001876397956622823 0.29"'
WAT1_002= 'p4=0. projp5=632052. lngcor = "3. 3. 3. 2. 0.001876397956622823 0.29"'
WAT1_003= '99113930557312 0.1542460039112511 0.3032873851581314 1.9247409545894'
WAT1_004= '95E-5 -1.348328290485618E-5 1.414186703253352E-4 -1.392784764381400E'
WAT1_005= '-4 -1.276226238774833E-4 4.339217671825231E-4 "'
WAT2_001= 'wtype=zpx axtype=dec projp0=0. projp1=1. projp2=0. projp3=337.74 projp4=0. projp5=632052. latcor = "3. 3. 3. 2. 0.001876397956622823 0.2"'
WAT2_002= 'jp4=0. projp5=632052. latcor = "3. 3. 3. 2. 0.001876397956622823 0.2"'
WAT2_003= '99113930557312 0.1542460039112511 0.3032873851581314 1.9247409545894'
WAT2_004= '402E-5 -1.378185066830135E-4 1.55989240147960429918328044731203771'
WAT2_005= 'E-4 3.966701903249366E-4 0.001678960379199465 "' 0.1542460039112511 0.3032873851581314
```

The long string for each image axis is composed of a set of keyword/value pairs where the value is quoted if it contains spaces. Figure 2 shows how the WAT keywords in figure 1 would be decomposed into parameters and coefficients.

Figure 2: Decomposing the WAT keywords from figure 1

```
AXIS 1      AXIS 2
-----
wtype=zpx   wtype=zpx
axtype=ra   axtype=dec
projp0=0.   projp0=0.
projp1=1.   projp1=1.
projp2=0.   projp2=0.
projp3=337.74 projp3=337.74
projp4=0.   projp4=0.
projp5=632052. latcor=
              3.
              3.
              2.
0.001876397956622823 0.001876397956622823
0.2999113930557312 0.1542460039112511
0.3032873851581314 0.3032873851581314
```

1.924740954589495E-5  
 -1.348328290485618E-5  
 1.414186703253352E-4  
 -1.792784764381400E-4  
 -1.276226238774833E-4  
 4.339217671825231E-4

9.963957331149402E-5  
 -1.378185066830135E-4  
 1.559892401479664E-4  
 -8.280442729203771E-4  
 3.966701903249366E-4  
 0.001678960379199465

In the example with the half cross-terms and orders of coefficients would be C00, C10, C20, C30, C01, C11, C21 and C03. The polynomials Pmn are defined below. The chebyshev polynomials are define iteratively as functions of the coordinates defined earlier.

The list of coefficients are interpreted as follows.

$$P_{mn} = xi ** m * eta ** n \quad (\text{polynomial})$$

- The first number is the function type encoded as 1=chebyshev, 2=legendre, 3=polynomial. The example has a function of type 3. The polynomial is the simple polynomial.
- The next two numbers represent the "order" of the function in xi and eta. The order is the one less than the highest polynomial power in xi and eta. The powers are represented below by m and n such at  $P_{m+1}(xi) = 2.0 * xi * P_m(xi) - P_{m-1}(xi)$  and  $P_{n+1}(eta) = 2.0 * eta * P_n(eta) - P_{n-1}(eta)$ . In the example the orders are 4 which means cubic polynomials (m=0 to 3 and n=0 to 3).
- The next (fourth) number specifies the type of cross-terms encoded as 0=no cross-terms, 1=full cross-terms, 2=half-cross-terms. Full cross-terms mean that both m and n will go to their maximum values independently while half-cross terms mean that m + n will only go to the maximum of xiorder-1 and etaorder-1.  $P_0(xi) = 1.0$
- The next 4 numbers describe the region of validity of the function in xi and eta space, e.g. xmin, xmax, etamin, etamax. The function is defined by  $P_{m+1}(xi) = ((2m+1) * xi * P_m(xi) - m * P_{m-1}(xi)) / (2m+1)$  and  $P_{n+1}(eta) = ((2n+1) * eta * P_n(eta) - n * P_{n-1}(eta)) / (2n+1)$ . compute normalized values for xi and eta used in the chebyshev and legendre polynomial functions:

$$xi = (2 * xi - (xmax + xmin)) / (xmax - xmin)$$

$$eta = (2 * eta - (etamax + etamin)) / (etamax - etamin)$$

In the example with with a simple polynomial the function follows.

- The remaining terms are the coefficients of the polynomial as follows. The functions are evaluated by summing polynomial terms  $P_{mn}(xi,eta)$  multiplied by the coefficients  $C_{mn}$  as

$$lncor/latcor = C00 + C10 * xi + C20 * xi^2 + C30 * xi^3 + C01 * eta + C02 * eta^2 + C03 * eta^3 + C11 * xi * eta + C21 * xi^2 * eta + C12 * xi * eta^2$$

Representing the coefficients as  $C_{mn}$  for the polynomials  $P_{mn}$ , where m and n are the powers of xi and eta, they are ordered as

C00  
 C10  
 C20  
 C30  
 ...  
 C01  
 C11  
 C21  
 C31  
 ...  
 C02  
 C12  
 C22  
 C32  
 ...  
 C03  
 C13  
 C23  
 C33  
 ...

## 8. The FITS Green Bank Keyword Convention

### 8.1. Preface

This FITS keyword convention was developed at a meeting in October 1989 at Green Bank, West Virginia to discuss standard FITS formats for interchange of single dish radio astronomy data. This convention was originally developed to specifically address the issue of how to represent World Coordinate System (WCS) information for images that are stored within a vector column of a FITS binary table (or what was then called a FITS '3-D' table), but the concept has since been generalized to have wider applications.

### 8.2. Original Green Bank Keyword Convention

This keyword convention originally applied to cases where a FITS binary table contains only a single multidimensional array field, or where the table contains several array fields, but they all have the same dimensions. The dimensions of each array column are defined by the TDIM $n$  keyword, which has the form:

$$\text{TDIM}n = '(i, j, k, \dots)'$$

where  $n$  is the column number of the multidimensional array in the binary table, and  $i, j, k, \dots$  are the integer dimensions of the array, expressed in the same order as in arrays in the Fortran programming language. Since the WCS parameters for the images (e.g., CTYPE $i$ , CRPIX $i$ , CRVAL $i$ , etc.) may have different values in each row of the table, in general it is necessary to expand these keywords into table columns, where the column names are the same as the keyword name. Thus,

- TTYPE $n$  = 'CTYPE $i$ ' means that the name of the physical coordinate of the axis  $i$  in the array contained in the table is given in column  $n$  of the table.
- TTYPE $n$  = 'CRPIX $i$ ' means that the value of the reference point for axis  $i$  in the array contained in the table is given in column  $n$  of the table.
- TTYPE $n$  = 'CRVAL $i$ ' means that the value of the physical coordinate for axis  $i$  at the reference point in the array contained in the table is given in column  $n$  of the table.

Similarly, any other needed WCS parameters are represented as additional columns in the table.

In the special case where a WCS parameter has the same value in every row of the table, it is not necessary to expand the standard WCS keyword into a column. For example, if every image in the multidimensional array column has CRPIX1 = 256, then it is more efficient to represent this with a single CRPIX1 header keyword, instead of defining a CRPIX1 column, with the same value of 256 in every row.

It should be noted that this convention pre-dates the development of the special forms for the WCS keywords that are specifically designed for use with images stored in multidimensional array columns in a binary table (e.g. 'iCTYP $n$ ' instead of 'CTYPEn'). Refer to the WCS section of the FITS Standard for more information about these keywords.

### 8.3. Generalized Green Bank Keyword Convention

The same principle that is used to expand a WCS keyword into a table column can be applied to any parameter whose value is different in each row of the binary table. For example, if the information given in each row of the table correspond to a different

date, then instead of having a single DATE-OBS keyword in the header of the table, one could add a column to the table that has TTYPE $n$  = 'DATE-OBS' to store the specific date value for each row. This concept of expanding a keyword into a table column (or conversely, collapsing a column of identical values into a single header keyword) is now generally known as the Green Bank keyword convention.

## 9. The ESO HIERARCH Keyword Convention

### 9.1. Preface

This keyword name convention has been used within the FITS data files produced by ESO since approximately 1990.

### 9.2. Convention Description

To avoid possible misinterpretations and naming conflicts for keywords describing data acquisition parameters, ESO (the European Organization for Astronomical Research in the Southern Hemisphere) developed a hierarchical keyword convention for this purpose. Under this convention, the FITS keyword name (bytes 1 through 8 of the keyword record) is HIERARCH, and byte 9 contains a space character. Since the HIERARCH keyword does not have the '=' value indicator in bytes 9 and 10 of the keyword record, it is in the same class as the COMMENT and HISTORY keywords that do not have a formal value. Thus, FITS readers that do not support the HIERARCH convention, as described in more detail below, should simply interpret bytes 9 through 80 of the keyword record as containing commentary text.

Under the HIERARCH keyword convention, bytes 10 through 80 of the keyword record contain a series of ASCII strings, or tokens, that serve to hierarchically classify the keyword, followed by an equals sign ("=") which is in turn followed by the keyword value field. An optional comment field may follow the value field, separated by a slash ("/") character. The value and comment fields conform to the rules for free-format keywords, as defined in the FITS Standard document.

The HIERARCH keyword structure is illustrated below:

```
HIERARCH token_1 token_2 ... token_n = value / comment
```

The first token following the HIERARCH keyword name is the 'name space' token, which defines the top level domain of the following tokens. The name space token has the value "ESO" for all the hierarchical keywords defined within that organization; a different unique domain name should be defined by any other organizations that uses this convention. (Currently, it appears that ESO is the only organization that uses this convention).

The other tokens following the name space token and preceding the equals sign character define the hierarchical classification of the keyword. Any number of levels are allowed (as long as they all fit within the 80-character keyword record), but in practice, ESO keywords generally have 3 hierarchical levels which specify the general category, the subsystem, and the parameter name, respectively. For example, in the following keyword:

```
HIERARCH ESO TEL FOCU SCALE = 1.489 / (deg/m) Focus length = 5.36"/mm
```

the domain = ESO, the category = TEL, the subsystem = FOCU, and the parameter name = SCALE.

Under the ESO implementation of this convention, each token string that precedes the equals sign must only contain characters that are legal in formal FITS keywords, i.e., the uppercase letters A through Z, the digits 0 through 9, and the hyphen and underscore characters. The tokens may, however, be longer than the 8 character limit of formal FITS keywords.

In some circumstances it may be convenient to map the hierarchical keywords into program variable names by concatenating the hierarchical tokens together, separating them

with the full stop character ("."). For example, the hierarchical keyword shown above corresponds to the variable name ESO.TEL.FOCU.SCALE while the following keyword,

```
HIERARCH ESO INS OPTI-3 ID = 'ESO#427 ' / Optical elem
```

corresponds to the variable ESO.INS.OPTI-3.ID. The reverse translation is applied when converting such variables into FITS HIERARCH keywords,

This hierarchical structure provides a convenient and clear way to separate information concerning different subsystems. The definition of FITS keywords used by ESO for data acquisition can be found in the Data Interface Control Document (<http://archive.eso.org/dicb>). This document also gives a full definition of the hierarchical keywords in the ESO name space.

## 10. The CONTINUE Long String Keyword Convention

### 10.1. Preface

This convention for continuing the value of a character string keyword over multiple header records was originally developed by the HEASARC in 1994. It has been extensively used within the FITS data files produced by numerous high energy astrophysics missions.

### 10.2. Introduction

The CONTINUE long string keyword convention may be used to assign a character string value to a FITS keyword that is longer than the 68-character limit for the value of a single FITS keyword. Under this convention, the long string value is divided into multiple substrings, each of which is no longer than 67 characters in length. The first substring is written as the value of the user-specified keyword, and the remaining substrings are written to a sequence of keywords that all have the keyword name CONTINUE.

### 10.3. Detailed Syntax of the Convention

The following steps should be taken when writing long string keyword values using this convention:

1. Divide the long string value into a sequence of smaller substrings, each of which is no longer than 67 characters in length. (Note that if the string contains any literal single quote characters, then these must be represented as a pair of single quote characters in the FITS keyword value, and these 2 characters must both be contained within a single substring).
2. Append an ampersand character ('&') to the end of each substring, except for the last substring. This character serves as a flag to FITS reading software that this string value *may* be continued on the following keyword in the header.
3. Enclose each substring with single quote characters. Non-significant space characters may occur between the ampersand character and the closing quote character.
4. Write the first substring as the value of the user-specified keyword.
5. Write each subsequent substring, in order, to a series of keywords that all have the name CONTINUE in bytes 1 through 8 and have space characters in bytes 9 and 10 of the keyword record. The substring may be located anywhere in bytes 11 through 80 of the keyword record and may be preceded by non-significant space characters starting in byte 11. A comment string may follow the substring; if present the comment string must be separated from the substring by at least 1 space character followed by a forward slash character ('/').

An example of this long string keyword convention is shown below:

```
SVALUE = 'This is a long string value &'
CONTINUE 'extending&'
CONTINUE ' over 3 lines.'
```

This example is equivalent to the following single keyword:

```
SVALUE = 'This is a long string value extending over 3 lines.'
```

FITS reading software that supports this convention should take the following steps when reading a string-valued keyword:

1. Test if the last non-space character in the keyword value string is an '&' character. If true, then the keyword value *may* be continued on the next keyword record in the FITS header, if the following conditions are true:
  - the next keyword in the header has the name CONTINUE, and
  - bytes 9 and 10 of the keyword contain spaces (no '=' in byte 9), and
  - bytes 11 through 80 contain a character string enclosed in single quote characters, optionally preceded and followed by space characters, and optionally followed by a forward slash character and a comment string.
2. If all these conditions are true, then the character string on this CONTINUE keyword should be appended onto the character string from the previous keyword(s), after first deleting the '&' character from the previous string.
3. Repeat steps 1 and 2 to continue assembling the long keyword value until the required conditions are no longer true.

The following additional points regarding this long string keyword convention should also be noted:

- If a string keyword value ends with the '&' character, but is not immediately followed by a conforming CONTINUE keyword, then the '&' character should be considered as the literal last character in the string.
- If a FITS reader encounters a CONTINUE keyword that is not preceded by a string keyword (or another CONTINUE keyword) whose value string ends with the '&' character, then that CONTINUE keyword should be ignored (i.e., it should be interpreted the same as a COMMENT keyword).

The following example (in which a MAXVOLT keyword has somehow been inserted between the SVALUE keyword and its continuation keyword) illustrates both of the above conditions:

```
SVALUE = 'This is a long string value &'
MAXVOLT = 12.5
CONTINUE 'continued over 3 lines.'
```

Because the requirements of the CONTINUE convention are not met in this case, FITS readers should interpret the SVALUE keyword as a simple string-valued keyword, including the final '&' character in the value string, and the 'orphaned' CONTINUE keyword should be treated like a COMMENT keyword

- FITS readers that do not support this convention should treat any CONTINUE keywords (which have no value indicator in byte 9 and hence have no formally defined value) in the same way as COMMENT keywords.
- This convention is *not recommended* for use with reserved or mandatory FITS keywords (e.g., TYPEn or EXTNAME, or other commonly used keywords because of the likelihood of confusion by software applications that do not support this convention. It is recommended that this convention only be used for new application-specific keywords, the values of which are not critical to the general interpretation or understanding of the contents of the FITS file.

#### 10.4. LONGSTR Keyword

It is recommended that the following keywords be added to the header of any HDU that uses this long string convention:

```
LONGSTRN= 'OGIP 1.0'           / The OGIP long string convention may be used.  
COMMENT  This FITS file may contain long string keyword values that are  
COMMENT  continued over multiple keywords. This convention uses the '&  
COMMENT  character at the end of a string which is then continued  
COMMENT  on subsequent keywords whose name = 'CONTINUE'.
```

The presence of the LONGSTRN keyword serves to indicate that long string keywords may be present in the FITS file. The value of this keyword gives the name and version number of the specific convention that is used, which in this case is the OGIP (Office of Guest Investigator Programs, at the HEASARC) long string convention, version 1.0. The value of this keyword is a string so that it may be used to give the name of any other convention that the FITS community might adopt.

## 11. Keywords for Describing the Minimum and Maximum Values in Columns of FITS Tables

### 11.1. Preface

This convention was developed by the HEASARC in 1993 to describe the minimum and maximum values in columns of a FITS ASCII or binary table. It has been extensively used in particular within the FITS data files produced by many high energy astrophysics missions.

### 11.2. Keyword Definitions

The following 4 optional keywords may be used to describe the minimum and maximum values in columns of a FITS ASCII or binary table:

- TDMINn Keyword: The value field shall contain a number giving the minimum physical value contained in column n of the table. This keyword is analogous to the DATAMIN keyword that is defined in the FITS standard for use with FITS images.
- TDMAXn Keyword: The value field shall contain a number giving the maximum physical value contained in column n of the table. This keyword is analogous to the DATAMAX keyword that is defined in the FITS standard for use with FITS images.
- TLMINn Keyword: The value field shall contain a number giving the minimum legally defined physical value that might be contained in column n of the table.
- TLMAXn Keyword: The value field shall contain a number giving the maximum legally defined physical value that might be contained in column n of the table.

The following conventions should be followed in the use of these keywords:

- The 'physical value' is defined as the value after applying the TSCALn and TZEROn linear scaling keywords, if present.
- These keywords are not applicable to columns containing ASCII strings or logical data.
- These keywords should have the same data type as the physical values in the associated column (either an integer or a floating point number).
- These keywords apply to all the elements of a vector column.
- Any undefined elements (or any other IEEE special values in the case of floating point columns in binary tables) should be excluded when determining the value of these keywords.
- The TLMINn and TLMAXn keywords define the allowed legal range of the column values; there is no requirement that the column actually contain any or all of the allowed values.
- It is permissible to have values in the column that are less than TLMINn or greater than TLMAXn; the interpretation of any such out-of-range column elements is not defined by this convention.
- If TDMINn is greater than TDMAXn, or TLMINn is greater than TLMAXn, then this should be taken to mean that the pair of keywords are undefined.

### 11.3. Examples

These keywords are commonly used in event list tables in which each row of the table describes an event, such as the measured

arrival time, position, and/or energy of a detected photon. For example, if a particular CCD photon counting detector is 512 by 384 pixels in size, then the location of each photon in the 'chip' coordinate system would have an X coordinate ranging from 1 to 512 and a Y coordinate ranging from 1 to 384. Other coordinate frames could also be defined, such as a 'detector' coordinate system which might be defined so that the origin is centered on the chip. The FITS header keywords appropriate for this case are shown below:

```
XTENSION= 'BINTABLE'           / binary table extension
BITPIX   =                      8 / 8-bit bytes
NAXIS    =                      2 / 2-dimensional binary table
NAXIS1   =                      16 / width of table in bytes
NAXIS2   =                      384 / number of rows/events
PCOUNT   =                      0 / size of special data area
GCOUNT   =                      1 / one data group (required)
TFIELDS  =                      4 / number of columns in each row
EXTNAME  = 'EVENTS'           / name of this binary table
TTYPE1   = 'CHIPX'            / Chip coordinates
TFORM1   = '1I'               / format of column 1
TTYPE2   = 'CHIPY'            / Chip coordinates
TFORM2   = '1I'               / format of column 2
TTYPE3   = 'DETX'             / Detector coordinates
TFORM3   = '1I'               / format of column 3
TTYPE4   = 'DETY'             / Detector coordinates
TFORM4   = '1I'               / format of column 4
TLMIN1   =                      1 / minimum legal value in column 1
TLMAX1   =                      512 / maximum legal value in column 1
TLMIN2   =                      1 / minimum legal value in column 2
TLMAX2   =                      384 / maximum legal value in column 2
TLMIN3   =                    -256 / minimum legal value in column 3
TLMAX3   =                      255 / maximum legal value in column 3
TLMIN4   =                    -192 / minimum legal value in column 4
TLMAX4   =                      191 / maximum legal value in column 4
TDMIN1   =                      17 / minimum actual value in column 1
TDMAX1   =                      510 / maximum actual value in column 1
TDMIN2   =                      6 / minimum actual value in column 2
TDMAX2   =                      378 / maximum actual value in column 2
```

The CHIPX and CHIPY columns in this example give the photon location in the chip reference frame, and the DETX and DETY columns give the location in the detector reference frame.

The TLMINn and TLMAXn keywords give the allowed range of values in each column. The TDMINn and TDMAXn keywords are given for the first 2 columns in this example, to illustrate that the actual range of values in the column need not cover the entire allowed range. The TLMINn and TLMAXn keywords are often used to define the default binning range when creating a histogram of the values in the column(s). To create a 2D image from the CHIPX and CHIPY columns, the TLMINn and TLMAXn keywords for those columns indicate that the histogram bins should cover the coordinate range from 1 to 512 in the X direction, and from 1 to 384 in the Y direction to create an image of the entire chip. To make a similar image from the DETX and DETY columns, the bins would need to cover the coordinate range from -256 to +255 in X, and -192 to +191 in the Y direction (i. e., the first pixel in the lower left corner of the binned image would record the number of events that have DETX = -256 and DETY = -192). It is important to note that the values in the columns are allowed to exceed the range given

by TLMINn and TLMAXn. For example, any anomalous photon events might be assigned a chip coordinate of (-1, -1), therefore the histogramming algorithm should be prepared to deal with such outliers.

In practice, the TDMINn and TDMAXn keywords have been rarely used in publicly archived data sets. In contrast, the TLMINn and TLMAXn keywords are widely used, especially in the event list data files that have been produced by ROSAT, Chandra, XMM-Newton, INTEGRAL, and other X-ray and gamma-ray astrophysics missions since about 1994.

## 12. FITS Header Inheritance Convention

### 12.1. Preface

This convention has primarily been used in FITS data files produced and distributed by the Space Telescope Science Institute and by NOAO/KPNO beginning in 1995. It has also been used within data files from various instruments and data pipelines operated by, or partly by, the United Kingdom, such as INT/WFC, UKIRT/WFCAM and ESO/VISTA/VIRCAM.

### 12.2. INTRODUCTION

There are many instances of FITS data files where the same set of keywords (e.g. 'TELESCOP' or 'INSTRUME') are duplicated with the same value in every extension of a multi-extension FITS file. It would be desirable in such cases to write the keyword only once, and have it be shared by every extension in the file. One (usually minor) benefit would be to reduce the size of the file, but more importantly, this would avoid duplicating information in the file. This can cause problems with, for example, dynamic updates, where every instance of the keyword would need to be checked for consistency.

The INHERIT keyword convention was developed to address these issues by allowing the extensions in a FITS file to implicitly inherit the keywords in the primary header; this prevents needless repetition of keywords in each extension header and provides a mechanism for software to easily access keywords that are shared between different extensions. In principle, the convention allows one to build software that requires the user to ask only once for the value of a specific keyword for a given extension (rather than explicitly doing two keyword lookups, one for the extension and one for the primary header). By using this convention software can treat the primary and extension headers as effectively one logical header.

This convention was developed in 1995 and is extensively used in FITS files produced at the STScI for data sets from the later generation of instruments on the Hubble Space Telescope, including STIS, NICMOS, and ACS. Software support for this convention has been built into the IRAF FITS Image kernel (Zarate & Greenfield 1996). This convention continues to be used in various data sets produced by NOAO. It is also used by various instruments and data pipelines operated by, or partly by, the United Kingdom, such as INT/WFC, UKIRT/WFCAM and ESO/VISTA/VIRCAM.

### 12.3. IMPLEMENTATION DETAILS

The INHERIT keyword in an extension header shall have a logical value of T or F to indicate whether or not the current extension should inherit the keywords in the primary header of the FITS file. The INHERIT keyword shall be defined in the extension header immediately after the mandatory keywords. This Inherit Convention should only be used in FITS files that have a null primary array (e.g., with NAXIS = 0) to avoid possible confusion if array-specific keywords (e.g., BSCALE and BZERO) were to be inherited. If INHERIT=F in an extension header, the keywords from the primary header should not be inherited.

When an application that supports this convention reads an extension header with INHERIT = T, it should merge the keywords in the current extension with the primary header keywords. The exact merging mechanism is left up to the ap-

plication, but, for example, all the extension keywords could be copied into a structure in memory, and then the keywords from the primary array could be appended to it. The value of the INHERIT keyword should be set to F after the keywords have been merged. The mandatory primary array keywords (e.g., BITPIX, NAXIS, and NAXISn) and any COMMENT, HISTORY, and blank keywords in the primary header are never inherited. If the same keyword is present in both the primary header and the extension header, the value in the extension header shall take precedence. If an application modifies the value of an inherited keyword in an extension, the value of that keyword in the primary header is not affected (i.e., the application must explicitly change the value of the primary header keyword if that is desired).

### 12.4. PRACTICAL CONSIDERATIONS

One disadvantage to using this convention is that it may be hard to preserve the separation of the primary and extension header keywords in software. For example, simply copying an extension to a new file could cause the primary and extension keywords to be merged, thus effectively negating the benefits of this convention unless the software takes special care to disable the automatic inheritance and propagates the primary header and extension header separately. Thus, the convenience of not requiring two keyword lookups has been transferred to an inconvenience of trying to preserve the separation in the face of automatic merging of the two.

Another practical issue is that in applications where the bytes in the FITS file are interpreted serially (e.g., on tape or Internet downloads), the reader would need to cache the primary header in case it turns out that a later extension in the file uses the INHERIT convention.

Another drawback is that users may become confused when adding or modifying keywords to files with this convention. If the keywords have become inadvertently duplicated (i.e., are present in both the primary and extension headers) and the user modifies the primary keyword, they are surprised that no change in the keyword value has taken place (because the extension value takes precedence). Users may also become confused if they use a mixture of software tools, some of which show the inherited keywords in the extension header and others that do not support this convention.

Potential future users of this convention should carefully consider whether the benefits outweigh the disadvantages in their particular situation.

### 12.5. REFERENCES

Zarate, N & Greenfield, P 1996, "A FITS Image Extension Kernel for IRAF" Astronomy DataAnalysis Software and Systems V, ASP Conference Series, Vol. 101

## 13. FITS Foreign File Encapsulation Convention

### 13.1. Preface

This convention was developed at NOAO/KPNO in 1999 mainly to encapsulate graphics files into FITS files in the NOAO High Performance Pipeline System.

### 13.2. Introduction

This document describes a FITS convention developed by the IRAF Group (D. Tody, R. Seaman, and N. Zarate) at the National Optical Astronomical Observatory (NOAO). This convention is implemented by the fgsread/fgswrite tasks in the IRAF fitsutil package. It was first used in May 1999 to encapsulate preview PNG-format graphics files into FITS files in the NOAO High Performance Pipeline System.

### 13.3. FOREIGN File Extension

A FITS extension of type 'FOREIGN' (henceforth a "FOREIGN file extension" or just "FOREIGN extension") provides a mechanism for storing an arbitrary file or tree of files in FITS, allowing it to be restored to disk at a later time. Each FOREIGN extension contains a single file. This mechanism also provides a means for associating a group of FITS extensions of any type. Certain of the file attribute keywords can be included in the header of any FITS file or extension to support such things as storing a directory tree containing images, tables, and other non-FITS types of files in a multi-extension FITS file, and later restoring the whole tree to disk. The motivation for this extension is to allow an implementation based on the FITS multi-extension mechanism to encapsulate and pass non-FITS data. The FOREIGN extension may be used to store a file from any type of operating system (e.g. UNIX or Windows), however some of the specific file attributes that are recorded in the FOREIGN extension keywords may not map completely between different systems (e.g. the UNIX filemode string that may be recorded in the FG\_FMODE keyword does not have an exact counterpart under Windows).

The header of a FOREIGN FITS extension must begin with the following five keywords in the specified order with no intervening keywords.

```
1 XTENSION= 'FOREIGN'
2 BITPIX   =                8
3 MAXIS    =                0
4 PCOUNT   =                <filesize> / file size in bytes
5 GCOUNT   =                1
.
EXTNAME = '<filename>'
```

Some early implementations of the FOREIGN extension reversed the order of the PCOUNT and GCOUNT keywords, but this usage is now deprecated. The optional EXTNAME keyword is used only to identify the extension in listings. To restore a file to disk the "FG" (file group) keywords are used as outlined in the following section.

### 13.4. File Group (FG) Keywords

To be able to later unpack FOREIGN extensions and restore files to disk, a number of keywords must be added to the extension headers to store the information required to restore the

files. These are the "FG" keywords. The FG keywords are used in both FOREIGN type extensions and in standard FITS extensions such as IMAGE, BINTABLE, and so on.

FG\_GROUP (string) - Each time a file group is written a group name is assigned. The group name associates all of the elements of a group. Assuming the group name is unique then this can be used to associate all the extensions in a group for later restoration. This is useful if groups are concatenated in a larger sequence of extensions. The group name is arbitrary (like a filename) and is assigned by the user when the file group is written. For example, a group name for a directory tree might be the name of the root directory. It is up to the writer program to assign a group name if the user does not predefine one.

FG\_FNAME (string) - The filename of the file associated with the current extension. The maximum filename length is 67 characters. Any printable character except apostrophe is permitted. For an extension of type FOREIGN where the file type is directory, FG\_FNAME is the name of the directory.

FG\_FTYPE (string) - The physical file type. The following types are recognized:

- "text" - A file containing only text. Stored 8 bits per character using newline to delimit lines of text (like Unix).
- "binary" - Any file which is not a text file or one of the known file types. Stored as a byte stream without any conversion.
- "directory" - implementation dependent
- "symlink" - implementation dependent
- "FITS" - a native FITS extension
- "FITS-MEF" - a native multi-extension FITS (MEF) file. No count of the number of extensions in the MEF file is given, rather the MEF group consists of all subsequent extensions until a FITS extension is encountered which starts a new file.

FG\_MTYPE (string) - The logical or "mime" type of the file (optional).

FG\_LEVEL (integer) - The directory nesting level. All of the files in a directory are at the same level. FOREIGN extensions of type directory are used to name the directories at each level so that pathnames can be reconstructed (this scheme assumes that the extensions in a file group are ordered). Level 0 (zero) is the root directory of the file group. The root directory is unnamed and is implicitly the user's current working directory into which the file in the FOREIGN extension would be unpacked. When packing files into FOREIGN FITS extensions, the current working directory could be a logical choice for the FG\_GROUP file group name.

FG\_FSIZE (integer) - The size in bytes of the data portion of the file. This value is always identical to the value of the PCOUNT keyword. In the case of a file with a FG\_FTYPE value equal to "directory", the FG\_FSIZE value is zero.

FG\_FMODE (string) - The file mode as a string ("rwx-rwx-rwx", bits not set given as "-")

FG\_FUOWN (string) - The file UID (user ID) as the file owner name string.

FG\_FUGRP (string) - The file GID (group ID) as the file group name string.

FG\_CTIME (string) - The file creation time as a UTC value expressed as an ISO 8601 string.

FG\_MTIME (string) - The file modification time as a UTC value expressed as an ISO 8601 string.

FG\_COMP (string) - This keyword will not be used initially, but is reserved in case we choose to implement file (e.g. gzip)

compression in the archiver. The value would be a string such as "none" or "gzip". In the meantime files can be archived in compressed form by compressing them beforehand and archiving the compressed files as binary files. Part of the reason we are reluctant to implement compression in the archiver is that archive data may last indefinitely and it is hard to guarantee that the compressed data will be readable a decade or two in the future. We might need to avoid compression for archival data unless the compression algorithms and/or code are part of the archive as well. (This discussion refers only to foreign files, not to compressed images).

### 13.5. Examples

The following examples are taken from actual runs on the IRAF implementation of this convention using the tasks fgread and fgwrite from the external package fitsutil. These utilities are written in C and are not tied to any IRAF system library.

The IRAF fitsutil external package contains the fgwrite/fgread tasks to write and read FOREIGN extensions. These are scripts that call the native C programs fgwrite.e and fgread.e with the following arguments:

```
fgwrite [t ;tbdsfmζ] [o ;tbdsfmζ] [vdih] [g ;group_nameζ] [f
output_fits_file] [input_files]
Switches: f write to named file, otherwise write to stdout
d print debug messages
v verbose; print full description of each file
g FG_GROUP name. The default is the root directory name
t select file types to include in the output file
o skip file types from input files selection
h do not produce primary HDU
i write Table Of Contents in primary HDU
s calculate CHECKSUM and DATASUM for the input file
fgread [t ;tbdsfmζ] [o ;tbdsfmζ] [n ranges] [vdxrf] [f fitsfile]
[files]
```

where ranges is of the form 1,2,5,8-11

Switches:

```
d print debug messages
f read from named file rather than stdin
n get list of extension numbers to extract
o omit the indicated FITS types (tbdsfm)
r replace existing file at extraction
s check CHECKSUM if keywords are present
t include the indicated FITS types (tbdsfm) only
v verbose; print full description of each file
x extract files
```

The possible file types are

```
t: text
b: binary
d: directory
s: symbolic link
f: single FITS file
m: multiple extension FITS file (MEF)
```

Example 1: Create a FITS file containing an arbitrary set of files in a directory.

```
fi> dir r* l+
brwrwr zarate      3616 Aug 14  9:23 rdf.o
trwrwr zarate      6489 Aug 14 10:30 rdf_plio.c
xtrwrwr zarate     6952 Aug 14 10:31 read_plio
xtrwrwr zarate     6903 Aug 13 14:43 read_plio_save
```

```
xtrwrwr zarate      6952 Aug  8 15:02 readf_save
```

```
fi> fgwrite r* /tmp/fg.fits # Create a FITS file with the
```

```
fi> fxx /tmp/fg.fits # See the FITS file contents (single
```

EXT#	EXTTYPE	EXTNAME	EXTVER	BITPIX
0	/tmp/ft.fits			8
1	FOREIGN	rdf.o	1	8
2	FOREIGN	rdf_plio.c	1	8
3	FOREIGN	readf_save	1	8
4	FOREIGN	read_plio	1	8
5	FOREIGN	read_plio_save	1	8

Example 2: Here are the values of some of the FG\_keywords for the case where the FITS file contains files that were originally in the tki/ directory and the tki/dir2 subdirectory.

EXT#	FG_FNAME	FG_FTYPE	FG_LEVEL	FG_FSIZE	FG_FMODE
0					
1	tki	directory	1	0	drwxrwxrwx
2	max.o	binary	2	1616	rwrwr use
3	dir2	directory	2	0	drwxrwxrwx
4	list.txt	text	3	69	rwrwr use
5	home.txt	text	3	1113	rwrwr use
6	gmttolst.c	text	2	1243	rwrwr user
7	a.c	text	2	770	rwrwr use
8	varg.c	text	2	284	rwrwr use
9	max.c	text	2	372	rwrwr use

### 13.6. Implementation Notes

The following design notes refer to the fgwrite and fgread tasks in the IRAF fitsutil package, and provide some additional context and background information relating to the original motivations for the FOREIGN extension.

The fgwrite and fgread programs as used in the telescope data handling system are host callable (Unix) level tasks.

Sample syntax:

```
fgwrite ;flagsζ ;input-file-template-listζ
fgread ;flagsζ ;input-fileζ
```

The intention is not to provide a general file archive capability, but rather to be able to use FITS to carry along and archive some non-FITS auxiliary data. A secondary goal is to generalize FITS somewhat so that directories can be handled (archived and later restored) as well as linear file templates.

Since the goal is not to provide a general file archive capability, certain details are not addressed: symlinks to directories are not followed by the writer; unlike tar, hard links are not preserved; special files are ignored.

Selected task options:

Input-file-template-list is a sequence of file names or directory names (if it is a unix task, any templates will already have been expanded by the shell).

There should be an option to fgwrite specify the types of files to be archived; when descending a directory, a file list alone will not handle this. Hence some mechanism such as which of the possible supported file types (tbdsf), or a pattern matching template such as in "find -name", would be used to select the files to be archived.

The output host file (or byte stream) is a conventional FITS file consisting of a sequence of one or more FITS extensions, optionally preceded by a dataless primary header unit (PHU) describing the entire file. Writing of the PHU may be disabled even if a file is being written to disk (e.g. when writing a sequence of extensions to be concatenated).

Foreign files (text, binary, directory, symlink) are wrapped as single extensions with XTENSION='FOREIGN'. Single FITS images without extensions are converted to IMAGE extensions, writing a single extension to the output stream.

Multi-extension FITS files in the input are written unchanged except that keywords are added to the first HDU to identify the MEF group (subsequent extensions are merely copied to the output stream unchanged). If the first HDU in the input file is a PHU it is converted to an IMAGE extension. The order of the extensions in the output stream must match that in the input MEF for the MEF to be later restored to disk. The PHU and all extensions in the input MEF are still visible in the output file; their association as an MEF grouping is evident only by examining the FG keywords in the HDU. Any internal MEF associations, such as for inheritance, are still present, but might not be recognized by most software until the MEF group is later restored to a file.

By default the output stream will have a dataless PHU describing the contents of the file (this can be disabled as mentioned above). The PHU may optionally include a table of contents for the output file. If a TOC is generated this will require that the output file list be fully processed to determine the type and size of each input file, before writing out the PHU with TOC followed by the input data files. This might be desirable in any case to simplify the code (construction of the input file list can be separated from file conversion and output).

## 14. Checksum Convention

### 14.1. Preface

The Checksum convention was developed in 1994 and has been widely used to verify the integrity of FITS files produced by many observatories.

### 14.2. Introduction

The checksum keywords described here provide an integrity check on the information contained in *FITS* HDUs. (Header and Data Units are the basic components of FITS files, consisting of header keyword records followed by optional associated data records). The CHECKSUM keyword is defined to have a value that forces the 32-bit 1's complement checksum accumulated over all the 2880-byte *FITS* logical records in the HDU to equal negative 0. (Note that 1's complement arithmetic has both positive and negative zero elements). Verifying that the accumulated checksum is still equal to -0 provides a fast and fairly reliable way to determine that the HDU has not been modified by subsequent data processing operations or corrupted while copying or storing the file on physical media. The checksum does not guard against organized transformations or malicious tampering, however, because simple transformations, such as rearranging the order of 32-bit words in the file, do not affect the computed checksum value. The checksum also does not provide any information on the authenticity of the file because the CHECKSUM keyword can always be updated after making modifications to the file, leaving no trace that the file is not the same as the original. A brief comparison with alternative checksum algorithms is given in §A.6.

Two *FITS* keywords are reserved to record the checksum information in an HDU: DATASUM and CHECKSUM. Normally both keywords will be present in the header if either is present, but this is not required. These keywords apply only to the HDU in which they are contained. If the CHECKSUM keywords are written in one HDU of a multi-HDU *FITS* file then it is strongly recommended that they also be written to every other HDU in the file. In that case the checksum accumulated over the entire file will equal -0 as well. It is recommended that the current date and time be written into the comment field of both keywords to document when the checksum was computed (or more precisely, the time that the checksum computation process was started).

### 14.3. DATASUM Keyword

The value field of the DATASUM keyword shall consist of a character string containing the unsigned integer value of the 32-bit 1's complement checksum of the data records in the HDU (i.e., excluding the header records). For this purpose, each 2880-byte *FITS* logical record should be interpreted as consisting of 720 32-bit unsigned integers. The 4 bytes in each integer must be interpreted in order of decreasing significance where the most significant byte is first, and the least significant byte is last. Accumulate the sum of these integers using 1's complement arithmetic in which any overflow of the most significant bit is propagated back into the least significant bit of the sum.

The DATASUM value is expressed as a character string (i.e., enclosed in single quote characters) because support for the full range of 32-bit unsigned integer keyword values is problematic in some software systems. This string may be padded with non-significant leading or trailing blank characters or leading zeros.

A string containing only 1 or more consecutive ASCII blanks may be used to represent an undefined or unknown value for the DATASUM keyword. The DATASUM keyword may be omitted in HDUs that have no data records, but it is preferable to include the keyword with a value of 0. Otherwise, a missing DATASUM keyword asserts no knowledge of the checksum of the data records.

### 14.4. CHECKSUM Keyword

The value field of the CHECKSUM keyword shall consist of an ASCII character string whose value forces the 32-bit 1's complement checksum accumulated over the entire *FITS* HDU to equal negative 0. There are a vast number of possible character strings that could satisfy this requirement, but for the sake of consistency and uniformity it is recommended that the particular 16-character string generated by the algorithm described in the appendix be used. A string containing only 1 or more consecutive ASCII blanks may be used to represent an undefined or unknown value for the CHECKSUM keyword.

### 14.5. CHECKSUM Keyword Implementation Guidelines

#### 14.5.1. Overview

Checksums are used to gain confidence in the continued integrity of all sorts of data. The normal procedure is to calculate the checksum of the data on the transmitting side of some communication channel (including magnetic media) and later to compare that checksum with the recalculated checksum on the receiving side. The original checksum is transmitted separately over the same communication channel.

This scheme works for *FITS* data as for other data, but separating the checksum from the *FITS* file limits its utility, especially for archival storage. It is also hard to see just how to incorporate a separate checksum into the *FITS* standard.

The Internet checksum (ref. 5–7) resolves the similar problem of embedding a checksum within each IP packet by forcing the 1's complement checksum of the entire packet to equal zero. This is accomplished by writing the complement of the calculated checksum into each packet instead of the checksum itself.

A 1's complement checksum is preferable to a 2's complement checksum (as used by the Unix `sum` command, for example), since overflow bits are permuted back into the sum and therefore all bit positions are sampled evenly. A 32-bit sum is as quick and easy to calculate as a 16 bit sum due to this symmetry, providing greater sensitivity to errors (see §A.6).

Arranging to write a binary number into a *FITS* file is unattractive and limiting. However, the properties of commutativity and associativity that make the Internet checksum work in the first place, also make it possible to generalize the technique with an ASCII encoding that may be embedded within a *FITS* header keyword (ref. 1).

Although it is advantageous to store the checksum complement within each HDU, the only place where that can be done, without violating the standard, is in the header. As a consequence, one has to be cognizant of the fact that this mechanism is not practical for application in situations where a *FITS* file is being created dynamically onto a streaming medium: at the point in time when the header is being written the value of DATASUM is not yet known, and when DATASUM is known the header cannot be modified anymore.

#### 14.5.2. Recommended CHECKSUM Keyword Implementation

The recommended CHECKSUM keyword algorithm described here generates a 16-character ASCII string that forces the 32-bit 1's complement checksum accumulated over the entire FITS HDU to equal negative 0 (all 32 bits equal to 1). In addition, this string will only contain alphanumeric characters within the ranges 0–9, A–Z, and a–z to promote human readability and transcription. This CHECKSUM keyword value must be expressed in fixed format, with the starting single quote character in column 11 and the ending single quote character in column 28 of the FITS keyword record, because the relative placement of the value string within the keyword record affects the computed HDU checksum. The steps in the algorithm are as follows:

1. Write the CHECKSUM keyword into the HDU header with an initial value consisting of 16 ASCII zeros ('0000000000000000') where the first single quote character is in column 11 of the FITS keyword record. This specific initialization string is required by the encoding algorithm described in §A.3. The final comment field of the keyword, if any, must also be written at this time. It is recommended that the current date and time be recorded in the comment field to document when the checksum was computed.
2. Accumulate the 32-bit 1's complement checksum over the FITS logical records that make up the HDU header in the same manner as was done for the data records by interpreting each 2880-byte logical record as 720 32-bit unsigned integers.
3. Calculate the checksum for the entire HDU by adding (using 1's complement arithmetic) the checksum accumulated over the header records to the checksum accumulated over the data records (i.e., the previously computed DATASUM keyword value).
4. Compute the bit-wise complement of the 32-bit total HDU checksum value by replacing all 0 bits with 1 and all 1 bits with 0.
5. Encode the complement of the HDU checksum into a 16-character ASCII string using the algorithm described in §A.3.
6. Replace the initial CHECKSUM keyword value with this 16-character encoded string. The checksum for the entire HDU will now be equal to negative 0.

#### 14.5.3. Recommended ASCII Encoding Algorithm

The algorithm described here is used to generate an ASCII string which, when substituted for the value of the CHECKSUM keyword, will force the checksum for the entire HDU to equal negative 0. It is based on a fundamental property of 1's complement arithmetic that the sum of an integer and the negation of that integer (i.e., the bitwise complement formed by replacing all 0 bits with 1s and all 1 bits with 0s) will equal negative 0 (all bits set to 1). This principle is applied here by constructing a 16-character string which, when interpreted as a byte stream of 4 32-bit integers, has a sum that is equal to the complement of the sum accumulated over the rest of the HDU. This algorithm also ensures that the 16 bytes that make up the 4 integers all have values that correspond to ASCII alpha-numeric characters in the range 0–9, A–Z, and a–z.

1. Begin with the 1's complement (replace 0s with 1s and 1s with 0s) of the 32-bit checksum accumulated over all the

FITS records in the HDU after first initializing the CHECKSUM keyword with a fixed-format string consisting of 16 ASCII zeros ('0000000000000000').

2. Interpret this complemented 32-bit value as a sequence of 4 unsigned 8-bit integers, A, B, C and D, where A is the most significant byte and D is the least significant. Generate a sequence of 4 integers, A1, A2, A3, A4, that are all equal to A divided by 4 (truncated to an integer if necessary). If A is not evenly divisible by 4, add the remainder to A1. The key property to note here is that the sum of the 4 new integers is equal to the original byte value (e.g.,  $A = A1 + A2 + A3 + A4$ ). Perform a similar operation on B, C, and D, resulting in a total of 16 integer values, 4 from each of the original bytes, which should be rearranged in the following order:

A1 B1 C1 D1 A2 B2 C2 D2 A3 B3 C3 D3 A4 B4 C4 D4

Each of these integers represents one of the 16 characters in the final CHECKSUM keyword value. Note that if this byte stream is interpreted as 4 32-bit integers, the sum of the integers is equal to the original complemented checksum value.

3. Add 48 (hex 30), which is the value of an ASCII zero character, to each of the 16 integers generated in the previous step. This places the values in the range of ASCII alphanumeric characters '0' (ASCII zero) to 'r'. This offset is effectively subtracted back out of the checksum when the initial CHECKSUM keyword value string of 16 ASCII 0s is replaced with the final encoded checksum value.
4. To improve human readability and transcription of the string, eliminate any non-alphanumeric characters by considering the bytes a pair at a time (e.g.,  $A1 + A2, A3 + A4, B1 + B2$ , etc.) and repeatedly increment the first byte in the pair by 1 and decrement the 2nd byte by 1 as necessary until they both correspond to the ASCII value of the allowed alphanumeric characters 0–9, A–Z, and a–z shown in Figure 1. Note that this operation conserves the value of the sum of the 4 equivalent 32-bit integers, which is required for use in this checksum application.  
[postscript figure goes here!!]
5. Cyclically shift all 16 characters in the string one place to the right, rotating the last character (D4) to the beginning of the string. This rotation compensates for the fact that the fixed format FITS character string values are not aligned on 4-byte word boundaries in the FITS file. (The first character of the string starts in column 12 of the header card image, rather than column 13).
6. Write this string of 16 characters to the value of the CHECKSUM keyword, replacing the initial string of 16 ASCII zeros.

To invert the ASCII encoding, cyclically shift the 16 characters in the encoded string one place to the left, subtract the hex 30 offset from each character, and calculate the checksum by interpreting the string as 4 32-bit unsigned integers. This can be used, for instance, to read the value of CHECKSUM into the software when verifying or updating a file.

#### 14.5.4. Encoding Example

This example illustrates the encoding algorithm given in §A.3. Consider a FITS HDU whose 1's complement checksum is 868229149, which is equivalent to hex 33C0201D. This number was obtained by accumulating the 32-bit checksum over the header and data records using 1's complement arith-

metic after first initializing the CHECKSUM keyword value to '0000000000000000'. The complement of the accumulated checksum is 3426738146, which is equivalent to hex CC3DFE2. The steps needed to encode this hex value into ASCII are shown schematically below:

	Byte		Preserve byte alignment
	A B C D	A1 B1 C1 D1	A2 B2 C2 D2 A3 B3 C3 D3 A4 B4 C4 D4
CC 3F DF E2	->	33 0F 37 38	33 0F 37 38 33 0F 37 38 33 0F 37 38
+ remainder		0 3 3 2	
= hex		33 12 3A 3A	33 0F 37 38 33 0F 37 38 33 0F 37 38
+ 0 offset		30 30 30 30	30 30 30 30 30 30 30 30 30 30 30 30
= hex		63 42 6A 6A	63 3F 67 68 63 3F 67 68 63 3F 67 68
= ASCII		c B j j	c ? g h c ? g h c ? g h c ? g h
			Eliminate punctuation characters
initial values		c B j j	c ? g h c ? g h c ? g h
.		c C j j	c > g h c @ g h
.		c D j j	c = g h c A g h
.		c E j j	c < g h c B g h
.		c F j j	c ; g h c C g h
.		c G j j	c : g h c D g h
final values		c H j j	c 9 g h c E g h
final string		"hcHjJC9ghcEghc9g"	(rotate characters 1 place to the right)

In this example byte B1 (originally ASCII B) is shifted higher (to ASCII H) to balance byte B2 (originally ASCII ?) being shifted lower (to ASCII 9). Similarly, bytes B3 and B4 are shifted by opposing amounts. This is possible because the two sequences of ASCII punctuation characters that can occur in encoded checksums are both preceded and followed by longer sequences of ASCII alphanumeric characters. This operation is purely for cosmetic reasons to improve readability of the final string.

This is how these CHECKSUM and DATASUM keywords would appear in a FITS header:

```

1234567890123456789012345678901234567890123456789012345678901234567890
DATASUM = '2503531142' / Data checksum created 2001-06-28 18:30:45
CHECKSUM= 'hcHjJC9ghcEghc9g' / HDU checksum created 2001-06-28 18:36:45

```

### 14.5.5. Incremental Updating of the Checksum

The symmetry of 1's complement arithmetic also means that after modifying a FITS HDU, the checksum may be incrementally updated using simple arithmetic without accumulating the checksum for portions of the file that have not changed. The new checksum is equal to the old total checksum plus the checksum accumulated over the modified records, minus the original checksum for the modified records.

An incremental update provides the mechanism for end-to-end checksum verification through any number of intermediate processing steps. By *calculating* rather than *accumulating* the intermediate checksums, the original checksum test is propagated through to the final data file. On the other hand, if a new checksum is accumulated with each change to the file, no information is preserved about the file's original state.

The recipe for updating the CHECKSUM keyword following some change to the file is:  $C' = C - m + m'$ , where  $C$  and

$C'$  represent the file's checksum (that is, the complement of the CHECKSUM keyword) before and after the modification and  $m$  and  $m'$  are the corresponding checksums for the modified FITS records or keywords only. Since the CHECKSUM keyword contains the complement of the checksum, the correspondingly complemented form of the recipe is more directly useful:  $\sim C' = (C + \sim m + m')$ , where  $\sim$  (tilde) denotes the (1's) complement operation. (See ref. 5-7.) Note that the tilde on the right hand side of the equation cannot be distributed over the contents of the parentheses due to the dual nature of zero in 1's complement arithmetic (ref. 7).

### 14.5.6. Alternate Checksum Algorithms

There are a variety of checksum schemes (for examples, see ref. 4,8-9) other than the 1's complement algorithm described in this proposal, although other checksums are significantly more difficult (often computationally impractical or impossible) to embed in FITS headers in the same fashion. Checksums such as cyclic redundancy checks (or CRCs, see ref. 3 for example), and message digests such as MD5 (ref. 12) are all examples of hash functions. Many possible images will hash to the same checksum—how many depends on the number of bits in the image versus the number of bits in the sum. The utility of a checksum to detect errors (but not forgeries), to one part in however many bits, depends on whether it evenly samples the likely errors.

For instance, a 32-bit checksum or CRC each detects the same fraction of all bit errors (ref. 9), missing only  $1/2^{32}$  of all errors (about 1 out of 4.3 billion) in the limit of long transmissions (the extra zero of 1's complement arithmetic changes this by only a small amount).

CRCs and message digests are basically checksums that use higher order polynomials, thus removing the arithmetic symmetry on which this proposal relies. CRCs are tuned to be sensitive to the bursty nature of communication line noise and will detect all bitstream errors shorter than the size of the CRC. Note that the 1's complement sum is not insensitive to these bit error patterns, it is just not *especially* sensitive to them. The extra sensitivity of a CRC to burst errors must come at the expense of lessened sensitivity to other bit pattern errors (since the total fraction of errors detected remains the same) and does not necessarily represent the best model for FITS bit errors. CRCs are also designed to be implemented in hardware using XOR gates and shift registers that accumulate the function "on-the-fly" and emit the CRC *after* transmitting the data. This is not well matched to the FITS convention of writing the metadata as a header which precedes the data records.

### 14.5.7. Digital Signatures

The particular intent of a message digest, on the other hand, is to protect against human tampering by relying on functions that are computationally infeasible to spoof. A message digest should also be much longer than a simple checksum so that any given message may be assumed to result in a unique value.

A *digital signature* may be formed by reverse encrypting a message digest using the private key of a public key encryption pair (ref. 13). A later decryption using the corresponding publicly available key guarantees that the signature could only have been generated by the holder of the private key, while the mes-

sage digest uniquely identifies the document (or image) that was signed. Support for digital signatures could be added to the *FITS* standard by defining a *FITS* extension format to contain the digital signature certificates, or perhaps by simply embedding them in an appended *FITS* table extension.

There is a tradeoff between the error detection capability of these algorithms and their speed. The overhead of a digital signature (or a software emulated CRC) is larger than a simple checksum, but may be essential for certain purposes (for instance, archival storage) in the future. The checksum defined by this proposal provides a way to verify *FITS* data against likely random errors, while on the other hand a full digital signature may be required to protect the same data against systematic errors, especially human tampering.

#### 14.5.8. Fletcher's Checksum

One other checksum algorithm shows some promise of being embeddable in an ASCII *FITS* header. This is *Fletcher's checksum* (ref. 9–11) which is a variant of the 1's complement checksum that is tuned to trap bit error patterns similar to those trapped by a CRC. It is somewhat slower than the 1's complement checksum and more finicky to implement. The checksum is divided into two (16 bit) pieces—a straight 1's complement sum and a higher order sum of the running sums. The procedure for updating the two checksum fields (zeroing the checksum of the file) involves solving a pair of simultaneous equations. ASCII encoding the checksum would require an iterative solution spread over the four separate ASCII encoded integer words (and including the constraint of the hex 30 offset). Incremental updating of the checksum would incur a similar penalty for each word of the *FITS* file that was modified.

The added complexity and overhead of handling Fletcher's checksum (see ref. 10–11) are unwarranted for *FITS*, at least as the default algorithm, but this checksum is an interesting possibility for binary applications. Other checksums are also options in the binary case, especially if the checksum fields can be located at the end of the file, which simplifies the arithmetic significantly.

#### 14.5.9. Error Correcting Algorithms

Error *correcting* (see ref. 2), as opposed to error *detecting*, algorithms are beyond the scope of this proposal, as are non-systematic codes for either error detection or correction. *Systematic* codes are those, such as the 1's complement checksum, that require no change to the data when applied to a message. Simply appending a checksum to a file is systematic, as is appending parity or other check bits to each byte or record of the data without otherwise modifying the data bits. Codes that are not systematic involve recoding the individual data bits in some fashion (see the discussion of *product* codes in ref. 4, for example).

#### 14.5.10. Example C Code for Accumulating the Checksum

The 1's complement checksum is simple and fast to compute. This routine assumes that the input records are a multiple of 4 bytes long (as is the case for *FITS logical records*), but it is not difficult to allow for odd length records if necessary. To use this routine, first initialize the CHECKSUM keyword

to '0000000000000000' and initialize sum32 = 0, then step through all the *FITS* logical records in the *FITS* HDU.

```
void checksum (
    unsigned char *buf, /* Input array of bytes to be checksummed */
                    /* (interpret as 4-byte unsigned ints) */
    int length,      /* Length of buf array, in bytes */
                    /* (must be multiple of 4) */
    unsigned int *sum32) /* 32-bit checksum */
{
    /*
     * Increment the input value of sum32 with the 1's complement sum
     * accumulated over the input buf array.
     */
    unsigned int hi, lo, hicarry, locarry, i;

    /* Accumulate the sum of the high-order 16 bits and the */
    /* low-order 16 bits of each 32-bit word, separately. */
    /* The first byte in each pair is the most significant. */
    /* This algorithm works on both big and little endian machines. */

    hi = (*sum32 >> 16);
    lo = *sum32 & 0xFFFF;
    for (i=0; i < length; i+=4) {
        hi += ((buf[i] << 8) + buf[i+1]);
        lo += ((buf[i+2] << 8) + buf[i+3]);
    }

    /* fold carry bits from each 16 bit sum into the other sum */
    hicarry = hi >> 16;
    locarry = lo >> 16;
    while (hicarry || locarry) {
        hi = (hi & 0xFFFF) + locarry;
        lo = (lo & 0xFFFF) + hicarry;
        hicarry = hi >> 16;
        locarry = lo >> 16;
    }

    /* concatenate the full 32-bit value from the 2 halves */
    *sum32 = (hi << 16) + lo;
}
```

#### 14.5.11. Example C Code for ASCII Encoding

This routine encodes the complement of the 32-bit HDU checksum value into a 16-character string. The byte alignment of the string is permuted one place to the right for *FITS* to left justify the string value starting in column 12.

```
unsigned int exclude[13] = { 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f, 0x40,
                          0x5b, 0x5c, 0x5d, 0x5e, 0x5f, 0x60 };

int offset = 0x30; /* ASCII 0 (zero) */
unsigned long mask[4] = { 0xff000000, 0xff0000, 0xff00, 0xff };

void char_encode (
    unsigned int value, /* 1's complement of the checksum value */
                    /* to be encoded */
    char *ascii) /* Output 16-character encoded string */
{
    int byte, quotient, remainder, ch[4], check, i, j, k;
    char asc[32];

    for (i=0; i < 4; i++) {
        /* each byte becomes four */
        byte = (value & mask[i]) >> ((3 - i) * 8);
        quotient = byte / 4 + offset;
        remainder = byte % 4;
        for (j=0; j < 4; j++)
            ch[j] = quotient;

        ch[0] += remainder;

        for (check=1; check;) /* avoid ASCII punctuation */
            for (check=0, k=0; k < 13; k++)
                for (j=0; j < 4; j+=2)
                    if (ch[j]==exclude[k] || ch[j+1]==exclude[k]) {
                        ch[j]++;
                        ch[j+1]--;
                        check++;
                    }
    }
}
```

```

    for (j=0; j < 4; j++)          /* assign the bytes */
        asc[4*j+i] = ch[j];
    }

    for (i=0; i < 16; i++)        /* permute the bytes for FITS */
        ascii[i] = asc[(i+15)%16];

    ascii[16] = 0;                /* terminate the string */
}

```

#### 14.5.12. Acknowledgments

The authors gratefully acknowledge the many helpful comments from Barry Schlesinger.

#### 14.5.13. References

1. Seaman, R.L. 1994, “*FITS Checksum Verification in the NOAO Archive*”, presented at the conference *Astronomical Data Analysis Software and Systems IV*, to appear in the *A.S.P. Conf. Ser.*
2. Peterson, W.W. and Weldon Jr., E.J. 1972, *Error-Correcting Codes*, Second Edition (MIT Press).
3. McNamara, J.E. 1982, *Technical Aspects of Data Communication*, Second Edition (Digital Press).
4. Plummer, W.W. 1978, “TCP Checksum Function Design”, *ACM Computer Communication Review*, **19**, no. 2, 95-101, this is an appendix to *Internet RFC 1071*.
5. Braden, R. T., Borman, D.A., and Partridge, C. 1988 (September), “Computing the Internet Checksum”, *ACM Computer Communication Review*, **19**, no. 2, 86-94, this is *Internet RFC 1071*.
6. Mallory, T. and Kullberg, A. 1990 (January), “Incremental Updating of the Internet Checksum”, *Internet RFC 1141*.
7. Rijssinghani, A. (ed.) 1994 (May), “Computation of the Internet Checksum via Incremental Update”, *Internet RFC 1624*.
8. Zweig, J. and Partridge, C. 1990 (March), “TCP Alternate Checksum Options”, *Internet RFC 1146*.
9. Fletcher, J.G. 1982, “An Arithmetic Checksum for Serial Transmission”, *IEEE Transactions on Communications*, **COM-30**, no. 1, 247-252.
10. Nakassis, A. 1988, “Fletcher’s Error Detection Algorithm: How to implement it efficiently and how to avoid the most common pitfalls”, *ACM Computer Communication Review*, **18**, no. 5, 63-88.
11. Sklower, K. 1989, “Improving the Efficiency of the OSI Checksum Calculation”, *ACM Computer Communication Review*, **19**, no. 5, 32-43.
12. Rivest, R. 1992 (April), “The MD5 Message Digest Algorithm”, *Internet RFC 1321*, see also *RFC 1319* and *RFC 1320*.
13. Zimmermann, P. 1995, *The Official PGP User’s Guide* (MIT Press), PGP is available from <http://net-dist.mit.edu/pgp.html> or <ftp://ftp.csn.net/mpj/README>, which also provide United States export and licensing requirements.

Internet *Requests for Comments*, or *RFCs*, are the written design documents for Internet protocols. They are available at many locations on the Internet, including <http://www.cis.ohio-state.edu/htbin/rfc/rfc-index.html>.

## 15. Tiled Image Compression Convention

### 15.1. Preface

### 15.2. General Description

This document describes a convention for compressing n-dimensional images and storing the resulting byte stream in a variable-length column in a FITS binary table. The FITS file structure outlined here is independent of the specific data compression algorithm that is used. The implementation details for 4 widely used compression algorithms are described here, but any other compression technique could also be supported by this convention.

The general principle used in this convention 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. By dividing the image into tiles it is generally 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 the ZTILEn keywords that are described below. In the case of relatively small images, it may be sufficient to compress the entire image as a single tile, resulting in an output binary table with 1 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.

### 15.3. Keywords

The following keywords are defined by this convention for use in the header of the FITS binary table extension to describe the structure of the compressed image.

- ZIMAGE (required keyword) This keyword must have the logical value T. It indicates that the FITS binary table extension contains a compressed image and that logically this extension should be interpreted as an image and not as a table.
- ZCMPTYPE (required keyword) The value field of this keyword shall contain a character string giving the name of the algorithm that must be used to decompress the image. Currently, values of GZIP\_1, GZIP\_2, RICE\_1, PLIO\_1, and HCOMPRESS\_1 are reserved, and the corresponding algorithms are described in a later section of this document. The value RICE\_ONE is also reserved as an alias for RICE\_1.
- ZBITPIX (required keyword) The value field of this keyword shall contain an integer that gives the value of the BITPIX keyword in the uncompressed FITS image.
- ZNAXIS (required keyword) The value field of this keyword shall contain an integer that gives the value of the NAXIS keyword in the uncompressed FITS image.
- ZNAXISn (required keywords) The value field of these keywords shall contain a positive integer that gives the value of the NAXISn keywords in the uncompressed FITS image.
- ZTILEn (optional keywords) The value of these indexed keywords (where n ranges from 1 to ZNAXIS) shall contain a positive integer representing the number of pixels along axis n of the compression tiles. Each tile of pixels is 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 ZNAXISn) is not required to be an integer multiple of ZTILEn, 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 ZTILEn keywords are not present then the default ‘row by row’ tiling will be assumed such that ZTILE1 = ZNAXIS1, and the value of all the other ZTILEn keywords equals 1.

The compressed image tiles are 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 appears in the first row of the table, and the tile containing the last pixel in the image appears in the last row of the binary table.

- ZNAMEn and ZVALn (optional keywords) These pairs of optional array keywords (where n is an integer index number starting with 1) supply the name and value, respectively, of any algorithm-specific parameters that are needed to compress or uncompress the image. The value of ZVALn may have any valid FITS datatype. The order of the compression parameters may be significant, and may be defined as part of the description of the specific decompression algorithm.
- ZMASKCMP (optional keyword) Used to record the name of the image compression algorithm that was used to compress the optional null pixel data mask. See the “Preserving undefined pixels with lossy compression” section for more details.
- The following 8 optional keywords are defined to store a verbatim copy of the value and comment fields of the corresponding keywords in the original uncompressed FITS image. These keywords can be used to reconstruct an identical copy of the original FITS file when the image is uncompressed.
  - ZSIMPLE - preserves the original SIMPLE keyword
  - ZTENSION - preserves the original XTENSION keyword
  - ZEXTEND - preserves the original EXTEND keyword
  - ZBLOCKED - preserves the original BLOCKED keyword
  - ZPCOUNT - preserves the original PCOUNT keyword
  - ZGCOUNT - preserves the original GCOUNT keyword
  - ZCHECKSUM - preserves the original CHECKSUM keyword
  - ZDATASUM - preserves the original DATASUM keyword

The ZSIMPLE, ZEXTEND, and ZBLOCKED keywords may only be used if the original uncompressed image was contained in the primary array of the FITS file. The ZTENSION, ZPCOUNT, and ZGCOUNT keywords may only be used if the original uncompressed image was contained in the IMAGE extension.

- ZQUANTIZ (optional keyword) This keyword records the name of the algorithm that was used to quantize floating-point image pixels into integer values which are then passed to the compression algorithm, as discussed further in section 4 of this document.
- ZDITHER0 (optional keyword) The value field of this keyword shall contain an integer that gives the seed value for the random dithering pattern that was used when quantizing the floating-point pixel values. The value may range from 1 to 10000, inclusive. See section 4 for further discussion of this keyword.
- Other Keywords The FITS header of the compressed image may contain other optional keywords. If a FITS primary array or IMAGE extension is compressed using the conven-

tion described here, it is recommended that all the keywords 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<sub>n</sub>, CRPIX<sub>n</sub> and CRVAL<sub>n</sub>).

#### 15.4. Columns

The following columns in the FITS binary table are defined by this convention. The order of the columns in the table is not significant. The column names (given by the TTYPE<sub>n</sub> keyword) are shown here in upper case letters, but the case is not significant.

Note regarding the variable-length columns: The COMPRESSED\_DATA, GZIP\_COMPRESSED\_DATA, and UNCOMPRESSED\_DATA columns described below will normally use the '1P' variable-length array FITS column format if the size of the heap in the compressed FITS file is less than about 2.1 GB in size. If the the heap is larger than 2.1 GB, then the '1Q' format (which uses 64-bit pointers) must be used.

- COMPRESSED\_DATA (required column)  
Each row of this variable-length column contains the byte stream that is generated as a result of compressing the corresponding image tile. The datatype of the column (as given by the TFORM<sub>n</sub> keyword) will generally be either '1PB', '1PI', or '1PJ' (or the equivalent '1Q' format), depending on whether the compression algorithm generates an output stream of 8-bit bytes, 16-bit integers, or 32-bit integers, respectively.
- GZIP\_COMPRESSED\_DATA (optional column)  
When using the quantization method to compress floating-point images that is described in Section 4, 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 can not be applied to certain tiles. In these cases, one may use an alternate technique in which the raw pixel values are losslessly compressed with the GZIP algorithm and the resulting byte stream is stored in the GZIP\_COMPRESSED\_DATA column (with a '1PB' or '1QB' variable-length array column format). The corresponding COMPRESSED\_DATA column for these tiles must contain a null pointer.
- UNCOMPRESSED\_DATA (optional column)  
Use of this column is no longer recommended, but it may exist in older compressed image files that were created before support for the GZIP\_COMPRESSED\_DATA column (describe above) was added to this convention in May 2011. This variable length column contains the uncompressed pixels for any tiles that cannot be compressed with the normal method. The datatype of this column should correspond to the datatype of the original image as shown in the following table:

Datatype	BITPIX	TFORM <sub>n</sub>
byte	8	'1PB' or '1QB'
short int	16	'1PI' or '1QI'
long int	32	'1PJ' or '1QJ'
float	-32	'1PE' or '1QE'
double	-64	'1PD' or '1QD'

A compressed image may contain either the UNCOMPRESSED\_DATA column or the GZIP\_COMPRESSED\_DATA column, but not both.

- ZSCALE and ZZERO (optional floating-point columns)  
When using the quantization method to compress floating-point images that is described in Section 4, these 2 columns store the linear scale factor and the zero point offset, respectively, that are used to scale the floating-point pixel values into integers via,

$$I_i = \text{ROUND}((F_i - \text{ZZERO})/\text{ZSCALE}) \quad (7)$$

where  $I_i$  and  $F_i$  are the integer and floating-point values, respectively and the ROUND function rounds the result to the nearest integer value. The array of integer tile pixel values is then compressed using the algorithm that is specified by the ZCTYPE keyword and the resulting compressed byte stream is stored in the COMPRESSED\_DATA column.

The ZSCALE and ZZERO columns should not be confused with the reserved BSCALE and BZERO keywords which may be present in integer FITS images (which have BITPIX = 8, 16, or 32). 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.

- ZBLANK (optional column or keyword)  
When using the quantization method to compress floating-point images that is described in Section 4, this column is used to store the integer value that represents undefined pixels (if any) in the scaled integer pixel values. These pixels have an IEEE NaN value (Not a Number) in the uncompressed floating-point image. 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 as a header keyword instead of a table column. If there are no undefined pixels in the image then ZBLANK is not required. If the uncompressed image has an integer datatype (ZBITPIX > 0) then the reserved BLANK keyword, which already serves this purpose, should be used instead of ZBLANK.
- NULL\_PIXEL\_MASK (optional column)  
When using some image compression techniques that do not exactly preserve integer pixel values, it may be necessary to store a compressed image mask along with the compressed image itself, to record the location of the undefined pixels in the image. This NULL\_PIXEL\_MASK column may be used for this purpose. See the "Preserving undefined pixels with lossy compression" section for more details.
- Other Columns Any number of other columns may be present in the table to supply other parameters that relate to each image tile.

## 15.5. Quantization of Floating-Point Data

While floating-point format images may be losslessly compressed (using gzip, since Rice and H-compress only compress integer arrays), these images often do not compress very well because the pixel values are too noisy; the less significant bits in the mantissa of the pixel values effectively contain incompressible random bit patterns. In order to achieve higher compression, one needs to remove some of this noise, but 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. 1, and using the ZSCALE and ZZERO columns to record the 2 scaling coefficients that are used for each tile. Note that the absence of these 2 columns in a tile-compressed floating-point image is an indication that the image was not scaled and was instead losslessly compressed.

The main challenge in quantizing the image in this way is in choosing an appropriate scaling factor. If it is too large, one undersamples the pixel values resulting in a loss of information in the image. If it is too small, however, it preserves too much of the noise (or even amplifies the noise) in the pixel values, resulting in poor compression.

An effective scaling algorithm for preserving a specified amount of noise in each pixel value is described by White and Greenfield (in the Proceedings of the 1998 ADASS VIII conference) and by Pence, Seaman, and White, PASP 121, 414 (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. It can be shown that the number of binary bits of noise that are preserved in each pixel value is given by  $\log_2(Q) + 1.792$ . For example, using  $Q = 8$  (so that the quantized levels have a spacing of 1/8th of the background RMS noise value) produces a quantized image that preserves about 4.8 bits of noise in each pixel. Specifying the quantization level relative to the amount of noise in the image in this way produces comparable quality images regardless of the noise level.  $Q$  is 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.

One 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, such as in the background sky. 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. 1, one randomizes the quantized levels by using this slightly modified equation:

$$I_i = \text{ROUND}(((F_i - \text{ZZERO})/\text{ZSCALE}) + R_i - 0.5) \quad (8)$$

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

$$F_i = ((I_i - R_i + 0.5) * \text{ZSCALE}) + \text{ZZERO} \quad (9)$$

This technique, which is called ‘subtractive dithering’ in the signal processing literature (e.g., “Quantization Noise” by Widrow and Kollar), 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. This can significantly increase the precision when measuring the net flux from faint sources in the compressed image.

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. Accordingly, we define a specific algorithm here for generating a repeatable sequence of pseudo random numbers in Appendix A.

### 15.5.1. Dithering Algorithms

When quantizing floating point images, one may choose from the 3 currently defined dithering algorithms as specified by the value of the ZQUANTIZ keyword, as described in the following sections.

#### 15.5.2. ZQUANTIZ= 'NO\_DITHER'

This is the simplest option in which no dithering is performed. The floating-point pixels are simply quantized using Eq. 1. This option should be assumed if the ZQUANTIZ keyword is not present in the header of the compressed floating-point image.

#### 15.5.3. ZQUANTIZ= 'SUBSTRUCTIVE\_DITHER\_1'

The steps in this dithering option are as follows:

1. Generate a sequence of 10000 single-precision floating-point random numbers, RN, with a value between 0.0 and 1.0, using the algorithm given in Appendix A. Since it could be computationally expensive to generate a unique random number for every pixel of large images, we repeatedly recycle 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 2 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. For example, one might calculate the seed value based on the system clock time when the image is compressed, or based on the checksum of all the pixel values in the first image tile that is compressed. However, beware of using the checksum method for choosing the seed value in cases where the first row/tile of all the images in a dataset are identical, as can happen if all the images have a border of zero or null valued pixels around the actual image.

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 2 offset parameters, I0 and I1, with the following formulae:

$$I0 = \text{modulo}(\text{TILE\_NUMBER} - 1 + \text{ZDITHER0}, 10000)I0$$

$$I1 = \text{INT}(\text{RN}(I0) * 500)I1$$

where TILE\_NUMBER 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(I0) is the value of the (I0)th random number in the sequence that was computed in the first step. Note that I0 has a value in the range 0 to 9999 and I1 has a value in the range 0 to 499. This method for computing I0 and I1 was chosen so that a different sequence of random number is used to compress successive tiles in the image, and so that the sequence of I1 values has a length of order 100 million elements before repeating.

5. Now quantize each floating-point pixel in the tile using Eq. 2 and using random number RN(I1) for the first pixel. Increment the value of I1 for each subsequent pixel in the tile. If the value of I1 reaches the upper limit of 10000, then increment the value of I0 and recompute I1 from Eq. 5. If the value of I0 also reaches the upper limit of 10000, then reset I0 to 0.
 

If the floating-point pixel has an IEEE NaN value, then it is not quantized or dithered and instead it is set to the reserved integer value that is specified by the ZBLANK keyword. For consistency, the value of I1 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 by default).
7. Write the compressed array of bytes 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. 2 for that tile into the ZSCALE) and ZZERO columns in the same row of the binary table.
9. Repeat Steps 4 through 8 for each tile of the image.

While the above dithering algorithm is clearly not unique, we present it here as a well defined method that should be possible to implement in almost any computer language. It should be noted that an image that is quantized using this technique can still be unquantized using the simple linear scaling function given by Eq. 1. The only side effect in this case is to introduce slightly more noise in the image than if the full subtractive dithering algorithm were applied.

#### 15.5.4. ZQUANTIZ= 'SUBTRACTIVE\_DITHER\_2'

This dithering algorithm is identical to the SUBTRACTIVE\_DITHER\_1 algorithm described above, except that any pixels in the floating-point image that are 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 are 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.

#### 15.6. Preserving undefined pixels with lossy compression

The null pixels in integer images are flagged by a reserved BLANK value and will be preserved if a lossless compression algorithm is used. If the image is compressed with a lossy algorithm, however (e.g., H-Compress with a scale factor greater than 1), then some other technique must be used to identify the null pixels in the image.

The recommended method of recording the null pixels when a lossy compression algorithm is used is to create an integer data mask with the same dimensions as the image tile. Set the null pixels to 1 and all the other pixels to 0, then compress the mask array using a lossless algorithm such as PLIO or GZIP. Store the compressed byte stream in a variable-length array column called 'NULL\_PIXEL\_MASK' in the row corresponding to that image tile. The ZMASKCMP keyword should be used to record the name of the algorithm used to compress the data mask (e.g., RICE\_1). The data mask array pixels will be assumed to have the shortest integer datatype 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, then uncompress the mask and set the corresponding undefined pixels in the image array to the appropriate value (as given by the BLANK keyword).

#### 15.7. Currently Implemented Compression Algorithms

This section describes the 4 compression algorithms that are currently supported in the CFITSIO implementation of this tiled image compression convention (available from the HEASARC web site). This does not imply that other implementations of this convention must support these same algorithms, nor does it limit other implementations from supporting other compression algorithms.

##### 15.7.1. Rice compression algorithm

If ZCMPTYPE = 'RICE\_1' then the Rice algorithm is used to compress and uncompress the image pixels. The Rice algorithm (Rice, R. F., Yeh, P.-S., and Miller, W. H. 1993, in Proc. of the 9th AIAA Computing in Aerospace Conf., AIAA-93-4541-CP, American Institute of Aeronautics and Astronautics) is simple and very fast, compressing or decompressing  $10^7$  pixels/sec on modern workstations. It requires only enough memory to hold a single block of 16 or 32 pixels at a time. It codes the pixels in small blocks and so is able to adapt very quickly to changes in the input image statistics (e.g., Rice has no problem handling cosmic rays, bright stars, saturated pixels, etc.).

The block size that is used should be recorded in the compressed image header with

```
ZNAMEn = 'BLOCKSIZE'
ZVALn  = 16 or 32
```

If these keywords are absent, then a default blocksize of 32 should be assumed.

The number of 8-bit bytes in each original integer pixel value should be recorded in the compressed image header with

```
ZNAMEn = 'BYTEPIX'
ZVALn  = 1, 2, 4, or 8
```

If these keywords are absent, then the default value of 4 bytes per pixel (32 bits) should be assumed..

### 15.7.2. GZIP compression algorithm

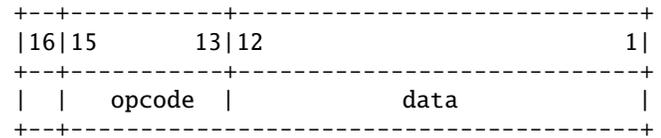
If ZCMPTYPE = 'GZIP\_1' then the gzip algorithm is used to compress and uncompress the image pixels. Gzip is the compression algorithm used in the free GNU software utility of the same name. It was created by Jean-loup Gailly and Mark Adler and is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. DEFLATE was intended as a replacement for LZW and other patent-encumbered data compression algorithms which, at the time, limited the usability of compress and other popular archivers. Further information about this compression technique is readily available on the Internet. The gzip algorithm has no associated parameters that need to be specified with the ZNAMEn and ZVALn keywords.

If ZCMPTYPE = 'GZIP\_2' then the bytes in the array of image pixel values are shuffled into decreasing order of significance before being compressed with the gzip algorithm. In other words, bytes are shuffled so that the most significant byte of every pixel occurs first, in order, followed by the next most significant byte, and so on for every byte. Since the most significant bytes of the pixel values often have very similar values, grouping them together in this way often achieves better net compression of the array. This is usually especially effective when compressing floating-point arrays.

### 15.7.3. IRAF PLIO compression algorithm

If ZCMPTYPE = 'PLIO\_1' then the IRAF PLIO (Pixel List) algorithm is used to compress and uncompress the image pixels. The PLIO algorithm was developed to store integer-valued image masks in a compressed form. Typical uses of image masks are to segment images into regions, or to mark bad pixels. Such masks often have large regions of constant value hence are highly compressible. The compression algorithm used is based on run-length encoding, with the ability to dynamically follow level changes in the image, allowing a 16-bit encoding to be used regardless of the image depth. The worst case performance occurs when successive pixels have different values. Even in this case the encoding will only require one word (16 bits) per mask pixel, provided either the delta intensity change between pixels is usually less than 12 bits, or the mask represents a zero floored step function of constant height. The worst case cannot exceed npix\*2 words provided the mask depth is 24 bits or less.

A good compromise between storage efficiency and efficiency of runtime access, while keeping things simple, is achieved if we maintain the compressed line lists 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 at present), a three bit opcode, and twelve bits of data, i.e.:



The significance of the data depends upon the instruction. The instructions currently implemented are summarized in the table below.

Instruction	Opcode	Description
ZN	00	Output N zeros
HN	04	Output N high values
PN	05	Output N-1 zeros plus one
SH	01	Set high value, absolute
IH,DH	02,03	Increment or decrement high
IS,DS	06,07	Like IH-DH, plus output on

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 as follows:

- ZN Zero the next N (=data) output pixels.
- HN Set the next N output pixels to the current high value.
- PN Zero the next N-1 output pixels, and set pixel N to the current high value.
- SH 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 Increment (IH) or decrement (DH) the current high value by the data value. The current position is not affected.
- IS,DS Increment (IS) or decrement (DS) the current high value by the data value, and step, i.e., output one high value.

The high value is assumed to 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. If the length of a line segment of constant value or the difference between two successive high values exceeds 4096 (12 bits), then multiple instructions are required to describe the segment or intensity change.

### 15.7.4. H-Compress algorithm

Hcompress is an the image compression package written by Richard L. White for use at the Space Telescope Science Institute. Hcompress was used to compress the STScI Digitized Sky Survey and has also been used to compress the preview images in the Hubble Data Archive. Briefly, the method used is:

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

The technique gives very good compression for astronomical images and is relatively fast. The calculations are carried out using integer arithmetic and are entirely reversible. Consequently, the program can be used for either lossy or lossless compression, with no special approach needed for the lossless case (e.g. there is no need for a file of residuals.)

There are 2 user-defined parameters associated with the H-Compress algorithm: an integer scale factor that determines the amount of compression, and a Boolean parameter that specifies whether the image should be smoothed during the decompression operation, to reduce residual artifacts in the image.

- **Scale Factor.** The integer scale parameter determines the amount of compression. Scale = 0 or 1 leads to lossless compression, i.e. the decompressed image has exactly the same pixel values as the original image. If the scale factor is greater than 1 then the compression is lossy: the decompressed image will not be exactly the same as the original. For astronomical images, lossless compression is generally rather ineffective because the images have a good deal of noise, which is inherently incompressible. However, if some of this noise is discarded then the images compress very well. The scale factor determines how much of the noise is discarded. Setting scale to 2 times sigma, the RMS noise in the image, usually results in compression by about a factor of 10 (i.e. the compressed image requires about 1.5 bits/pixel), while producing a decompressed image that is nearly indistinguishable from the original. In fact, the RMS difference between the decompressed image and the original image will be only about 1/2 sigma. Experiments indicate that this level of loss has no noticeable effect on either the visual appearance of the image or on quantitative analysis of the image (e.g. measurements of positions and brightnesses of stars are not adversely affected.)

Using a larger value for scale results in higher compression at the cost of larger differences between the compressed and original images. A rough rule of thumb is that if scale equals N sigma, then the image will compress to about 3/N bits/pixel, and the RMS difference between the original and the compressed image will be about N/4 sigma. This crude relationship is inaccurate both for very high compression ratios and for lossless compression, but it does at least give an indication of what to expect of the compressed images.

For images in which the noise varies from pixel to pixel (e.g. CCD images, where the noise is larger for brighter pixels), the appropriate value for scale is determined by the RMS noise level in the sky regions of the image. For images that are essentially noiseless, any lossy compression is noticeable under sufficiently close inspection of the image, but some loss is nonetheless acceptable for typical applications. Note that the quantization scheme used in Hcompress is not designed to give images that appear as much like the original as possible to the human eye, but rather is designed to produce images that are as similar as possible to the original under quantitative analysis. Thus, the emphasis is on discarding noise without affecting the signal rather than on discarding components of the image that are not very noticeable to the eye (as may be done, for example, by JPEG compression.) The resulting compression scheme is not ideal for typical terrestrial images (though it is still a reasonably good method for those images), but is believed to be close to optimal for astronomical images.

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 the value that was used. It is recommended that the ZNAMEn and ZVALn pair of keywords be used for this purpose, with

```
ZNAMEn = 'SCALE'
```

```
ZVALn = I
```

where *I* is the integer scale value.

- **Smoothing Flag.** At high compression factors the decompressed image begins to appear blocky because of the way information is discarded. This blockiness is greatly reduced, producing more pleasing images, if the image is smoothed slightly during decompression. When done properly, the smoothing will not affect any quantitative photometric or astrometric measurements derived from the compressed image. Of course, the smoothing should never be applied when the image has been losslessly compressed with a scale factor (defined above) of 0 or 1.

The smoothing option only needs to be specified when uncompressing the image, however, in many cases, this can best be determined by the person or project that creates the compressed image files. Thus it is recommended that the smoothing flag be specified in the compressed image header with the ZNAMEn and ZVALn keywords with

```
ZNAMEn = 'SMOOTH'
ZVALn = 0 or 1
```

A value of 0 means no smoothing, and any other value means smoothing is recommended. This should be regarded as only a recommendation which the image decompression program may override.

A paper describing Hcompress was published in the Proceedings of the NASA Space and Earth Science Data Compression Workshop, ed. James C. Tilton, Snowbird, Utah, March 1992. This paper is reproduced in the Appendix B of this document.

### 15.8. Random Number Generator

This portable random number generator algorithm comes from the publication “Random number generators: good ones are hard to find”, Communications of the ACM, Volume 31, Issue 10 (October 1988) Pages: 1192 - 1201 which is available on the Web. This algorithm basically just repeatedly evaluates the function seed = (a \* seed) mod m, where the values of a and m are shown below, but it is implemented in a way to avoid integer overflow problems.

```
int random_generator(void) {
    /* initialize an array of random numbers */

    int ii;
    double a = 16807.0;
    double m = 2147483647.0;
    double temp, seed;
    float rand_value[10000];

    /* initialize the random numbers */
    seed = 1;
    for (ii = 0; ii < N_RANDOM; ii++) {
        temp = a * seed;
        seed = temp - m * ((int) (temp / m));
        rand_value[ii] = seed / m; /* divide by m to get
    }
}
```

If implemented correctly, the 10000th value of seed will equal 1043618065.

## 16. Tiled Table Compression Convention

### 16.1. Preface

### 16.2. Overview

This document describes a convention for compressing FITS binary tables that is modeled after the widely used FITS tiled-image compression method (White et al. 2009). The uncompressed table may be subdivided into tiles, each containing the same number 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. Most of the header keywords from the uncompressed table, with only a few limited exceptions, are copied verbatim to the header of the compressed table. These header keywords remain uncompressed for efficient access. The compressed table is itself a valid FITS binary table that contains the same number and order of columns as in the uncompressed table, and contains one row for each tile of rows in the uncompressed table. All the currently supported compression algorithms (Rice and 2 variants of Gzip) are lossless, so no information is lost when the table is compressed.

This convention currently only supports FITS binary tables and cannot be used to compress FITS ASCII tables.

### 16.3. Compression Overview

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

#### A. 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 optionally 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 turn and 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 so as to not impose too great of a memory resource burden on software that compresses or uncompresses the table.

#### B. 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. 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.

#### C. Compress Each Column of Data

Each column of data is compressed with a suitable compression algorithm. 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), each individual variable length vector is compressed separately.

#### D. Store the Compressed Bytes

The compressed stream of bytes for each column is written into the corresponding column in the output table. The compressed table has 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'`, which is appropriate for storing the compressed stream of bytes. 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, are also compressed and written into the corresponding column in the compressed table. See section 6 for more details.

### 16.4. Compression Directive Keywords

The following optional 'compression directive' keywords, if present in the header of the table that is to be compressed, provide guidance to the compression software on how the table should be compressed. The compression software will attempt to obey these directives, but if that is not possible, the software may disregard them and use an appropriate alternative.

- `FZTILELN` 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` 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 current allowed values are `GZIP_1`, `GZIP_2`, and `RICE_1`. The corresponding algorithms are described in Section 5.
- `FZALGn`. The value field of these keywords shall contain a character string giving the mnemonic name of the algorithm that is requested to be used 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 the `FZALGOR` keyword in determining which algorithm to use for a particular column if both keywords are present. If the column cannot be compressed with the requested algorithm (e.g., if it has an inappropriate data type), then a default compression algorithm will be used instead.

### 16.5. Keywords in the Compressed Table

With only a few exceptions, all the keywords from the uncompressed table are copied verbatim, in order, into the header of the compressed table. The header keywords remain uncompressed for ease of access. Note in particular that the values of the reserved column descriptor keywords `TTYPEn`, `TUNITn`, `TSCALn`, `TZEROn`, `TNULLn`, `TDISPn`, and `TDIMn`, as well as all the column-specific WCS keywords defined in the FITS standard, have the same values 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 are not copied verbatim from the uncompressed table header to the compressed table header are the mandatory `NAXIS1`, `NAXIS2`, `PCOUNT`, and `TFORMn` keywords, and the optional `CHECKSUM`, `DATASUM`, and `THEAP` keywords. These keywords must necessarily describe the contents of the compressed table itself. The original values of these keywords in the uncompressed table are stored in a new set of reserved keywords in the compressed table header. The complete set of

keywords that have a reserved meaning within the header of a tile-compressed binary table are listed below:

- ZTABLE (required keyword). The value field of this keyword shall contain the logical value T. This indicates that the FITS binary table extension contains a tile-compressed binary table.
- ZNAXIS1 (required keyword). 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 (required keyword). 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 (required keyword). 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.
- ZFORMn (required indexed keywords). These required array keywords supply the character string value of the corresponding TFORMn keyword that defines the data type of the column in the original uncompressed FITS table.
- ZTHEAP (optional keyword). 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. In practice, this keyword is rarely used.
- ZTITLELEN (required keyword). The value 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 contain a single row, and the ZTITLELEN and ZNAXIS2 keywords will have the same value.
- ZCTYPn (required indexed keywords). The value field of these keywords shall contain a character string giving the mnemonic name of the algorithm that was used to compress column n of the table. The current allowed values are GZIP\_1, GZIP\_2, and RICE\_1, and the corresponding algorithms are described in Section 5.
- ZCHECKSUM (optional keyword). The value field of this keyword shall contain a character string that gives the value of the CHECKSUM keyword in the original uncompressed FITS table header.
- ZDATASUM (optional keyword). The value field of this keyword shall contain an integer that gives the value of the DATASUM keyword in the original uncompressed FITS table header.

## 16.6. Supported Compression Algorithms

This section describes the currently supported compression algorithms. Other compression algorithms may be added in the future.

### 16.6.1. GZIP\_1

This lossless compression algorithm is designated by the keyword ZCTYPn = 'GZIP\_1'. Gzip is the compression algorithm

used in the widely distributed GNU free software utility of the same name. It was created by Jean-loup Gailly and Mark Adler. It is based on the DEFLATE algorithm, which is a combination of LZ77 and Huffman coding. Further information about this compression technique is readily available on the Web. The "gzip -1" option is generally used which significantly improves the compression speed with only a small loss of compression efficiency.

It is important to note that any numerical data values must be arranged in big-endian byte order (the FITS standard) before the array of bytes is compressed.

### 16.6.2. GZIP\_2

This lossless compression algorithm is designated by the keyword ZCTYPn = 'GZIP\_2'. This algorithm is a variation of the GZIP\_1 algorithm in which the bytes in the arrays of numeric data columns are preprocessed by shuffling them so that they are arranged in order of decreasing significance before being compressed. For example, a 5-element array of 2-byte (16-bit) integer values, with an original big-endian byte order of

A1 A2 B1 B2 C1 C2 D1 D2 E1 E2,

will have the following byte order after shuffling the bytes:

A1 B1 C1 D1 E1 A2 B2 C2 D2 E2.

where A1, B1, C1, and D1 are the most significant bytes from each of the integer values. Byte shuffling can only be performed for numeric binary table columns that have TFORMn data type codes of I, J, K, E, D, C, or M. The bytes in columns that have a L, X, or A type code are never shuffled.

This byte-shuffling technique has been shown to be especially beneficial when compressing floating-point values because the bytes containing the exponent and the most significant bits of the mantissa are often similar for all the floating point values in the array. Thus these repetitive byte values generally compress very well when grouped together in this way. HDF Group has used this byte-shuffling technique when compressing HDF5 data files (HDF 2000).

### 16.6.3. RICE\_1

This lossless compression algorithm is designated by the keyword ZCTYPn = 'RICE\_1' and may only be applied to integer data type columns that have TYPEn data type code values of 'B', 'I', or 'J'. The Rice algorithm (Rice, 1993) is very simple and fast. It requires only enough memory to hold a single block of 32 integers at a time and is able to adapt very quickly to changes in the input array statistics.

## 16.7. Compressing Variable-Length Array Columns

Compression of binary tables that contain variable-length array (VLA) columns (with a P or Q data type code) requires special consideration because the data values in these columns are not stored directly in the table, but instead are stored in what is called the 'data heap' which follows the main table. The VLA column in the main data table itself only contains a 'descriptor', which is composed of 2 integers that give the size and location of the actual array in the heap. When compressing a variable length array column, one must first process each individual VLA in turn

by reading it from the uncompressed table, compressing it, then writing the compressed bytes to the heap in the compressed table. The descriptors that point to these compressed VLAs must be stored in a temporary array of descriptors that has been allocated for this purpose. Once all the individual VLAs in the column have been processed, that temporary array of descriptors is then itself compressed with GZIP\_1, and then finally written into the heap of the compressed table.

There is one other complexity that must be addressed when dealing with VLA columns: one needs to know the original descriptor values to be able to write the uncompressed VLAs back into the same location in the heap as in the original uncompressed table. For this reason, we concatenate the array of descriptors from the uncompressed table onto the end of the temporary array of descriptors (to the compressed VLAs in the compressed table) before the 2 combined arrays of descriptors are compressed and written into the heap in the compressed table.

When uncompressing a VLA column, 2 stages of uncompression must be performed: First, the combined array of descriptors must be uncompressed, then these descriptors are used one by one to read the compressed VLA from the compressed table, uncompress it, and then write it back into the correct location in the uncompressed table. Note also that the descriptors to the compressed VLAs are always 64-bit Q-type descriptors, but the descriptors from the original uncompressed table may be either Q-type or P-type.

The following example illustrates how this works in practice: suppose one compresses a 100 row table containing a column of 2-byte integer variable length arrays (with TFORMn = '1PI'). When compressing this column, each of the 100 individual VLAs are read from the uncompressed table, compressed with the appropriate algorithm, and then written to the corresponding TFORMn = '1QB' column in the compressed table. After all the VLAs have been processed, the array of 100 P-type descriptors from the uncompressed table are concatenated onto the end of the temporary array of 100 Q-type descriptors from the compressed table, and this combined array is compressed with the GZIP\_1 algorithm and written into the compressed table.

### References

- HDF 2000, "Performance Evaluation Report: gzip, bzip2 compression with and without shuffling," [http://www.hdfgroup.org/HDF5/doc\\_resource/H5Shuffle\\_Perf.pdf](http://www.hdfgroup.org/HDF5/doc_resource/H5Shuffle_Perf.pdf)
- Rice, R. F., Yeh, P.-S., and Miller, W. H. 1993, in Proc. of the 9th AIAA Computing in Aerospace Conf., AIAA-93-4541-CP, American Institute of Aeronautics and Astronautics
- White, R. L., Greenfield, P., Pence, W., Tody, D., and Seaman, R. 2009, "Tiled Image Compression Convention", <http://fits.gsfc.nasa.gov/registry/tilecompression.html>

## 17. A Hierarchical Grouping Convention

### 17.1. Preface

This paper describes a grouping convention for FITS that facilitates the construction of hierarchical associations of Header Data Units (HDUs). The grouping convention uses FITS table structures (ASCII or binary) to encapsulate pertinent information about the HDUs belonging to a group. Group members may reside in a single FITS file or be distributed in many FITS files; the FITS files themselves may reside on different computer systems.

### 17.2. Introduction

The rules for generalized extensions in FITS (Grosbøl *et al.*, 1988) provide for FITS formatted files containing more than one header data unit. By using combinations of ASCII tables (Harten *et al.*, 1988), binary tables (Cotton *et al.*, 1994) and image extensions (Ponz *et al.*, 1994) related data sets requiring different data structures may be stored in the same FITS file, each within its own HDU. Unfortunately, once the related data sets are segregated into separate HDUs the relationship between them is often lost.

The FITS standard currently allows for simple hierarchical associations of HDUs within a single FITS file through use of the EXTLEVEL keyword. However, this mechanism has several major limitations. First, its use is not well defined. Different organizations may use EXTLEVEL for widely varying purposes and still not violate the FITS standard. Secondly, it does not specify a mechanism for defining distinct multiple *groups* of HDUs within a FITS file. Lastly, it cannot be used to associate HDUs residing in different FITS files. Except for very simple cases, FITS contains no mechanism for creating or preserving associations between HDUs or groups of HDUs.

As the volume and complexity of FITS formatted data grows, the need for a recognized and versatile HDU grouping mechanism increases. Individuals can be overwhelmed trying to manage and analyze large data sets unless those sets are logically organized. Software tools also require data organization in order to access all necessary components of an observation, simulation or experimental data set.

As an example of where grouping capabilities within FITS would be useful, consider the following. It is desirable to combine a set of observations from a given time period into a single FITS file for transport and archival purposes. For each observation there is an observation log, an event list, a derived image and a set of instrument calibration data; furthermore, several observations share a common set of calibration data. By using a grouping mechanism each [log, event list, image, calibration] set could be logically tagged as an associated observation group and the calibration data could be made a part of many different observation groups, thus eliminating the need to store it more than once. Software could retrieve all the information about a given observation simply by extracting those HDUs defined in the table that identifies members of the group. Also, observations of the same object from different observational periods could be combined into a group and accessed as a unit, even though the HDU sets comprising the different observations reside in separate FITS files.

The following sections describe a scheme for implementing a hierarchical grouping of header data units within single

and multiple FITS files. Section 2 discusses the content of table extensions used to define HDU groupings. Section 3 lists those keywords recommended for headers of group member extensions. Finally, Section 4 provides sample headers from FITS table extensions containing grouping structures.

### 17.3. Group Tables

A *group table*, as defined in this convention, is a FITS table extension that contains a list of all the associated member HDUs in the group. Group tables may be represented by either FITS ASCII tables (XTENSION=`TABLE`) or binary tables (XTENSION=`BINTABLE`), and are uniquely distinguished from other types of FITS tables by having the EXTNAME=`GROUPING` keyword and value in the header. The other required or recommended keywords and columns in a group table are described in the following sections.

There may be zero, one, or more group tables within a given FITS file. Each group table may reference any number of HDUs. The entire set of HDUs referenced in a group table, along with the group table itself, form a *group*. Individual HDUs referenced in a group table are said to be *members of the group* or *group members*.

Groups can contain any type and mix of HDU. This includes all of the IAU-endorsed extensions as well as other extensions that conform to the requirements for generalized FITS extensions. Note that a group may also contain other groups as members, since a group table is itself a FITS extension. This feature allows for the construction of hierarchical structures of HDUs within a single FITS file or across many FITS files.

#### 17.3.1. Group Member Identification Methods

Group tables specify the names and locations of FITS files containing member HDUs as well as identifying members within their FITS files. The name and location of each FITS file is specified by using the World-Wide Web (Berners-Lee, 1994) Uniform Resource Identifiers, or URIs. All current and future forms of URIs, such as Uniform Resource Locators (URL) and the proposed Uniform Resource Names (URN), shall constitute valid names, although the group table must specify the type of URI being used. If the group member resides in a different FITS file but on the same computer system then partial URIs (specifically partial URLs) may be used instead of absolute URIs to specify the member's file location. If the group member resides in the same FITS file as the group table itself, then the URI field may be left blank.

The location of member HDUs within FITS files may be specified in two different ways, either by *reference* or by *absolute position*. The reference identification method uses the values of the XTENSION, EXTNAME and EXTVER keywords to uniquely identify the member HDU within the FITS file. The position method uses the HDU order number to identify members, with the primary array having order value 0, the first extension order value 1, and so on. Users may choose either or both identification methods when constructing a group table.

While the reference method is not invalidated by a reordering of HDU positions within FITS files, it does require that each member HDU have a unique set of (non-FITS-required) keyword values. Thus, this method may present problems for FITS files whose headers cannot be easily modified, such as FITS files

on read-only media. The position identification method provides for quick “random” access to the member HDUs, since software does not have to sort through each extension looking for the correct set of keyword values, but will be affected if the order of member HDUs within their FITS files is changed (please note: there is nothing within the current FITS standard governing how or when HDUs may be reordered within their files).

### 17.3.2. Group Table Keywords

In addition to the standard required FITS table extension keywords, the following keywords are required in the header of a group table:

- **EXTNAME (character):** This value of the FITS reserved keyword uniquely identifies that this FITS extension contains a group table. For group tables EXTNAME must have the value ‘GROUPING’.
- **EXTVER (positive integer):** The value of this FITS reserved keyword serves as a group ID number that uniquely distinguishes this group from any other groups that may be defined in the same FITS file. All HDUs in a given FITS file with EXTNAME=‘GROUPING’ must have a unique integer EXTVER value. This group number may also be used in the header of each group member to identify the group(s) to which the member belongs (see section 17.3.3, GRPIDn keyword).

The following keyword is strongly recommended for inclusion in the header of each group table:

- **GRPNAME (character):** This keyword contains the name associated with the group table. GRPNAME values are case-insensitive and should only contain letters, digits, and the underscore character (and not contain any embedded blank (ASCII 32) characters).

### 17.3.3. Group Table Columns

The number of columns required in a group table depends on which method is used to identify the members (and recall that both methods may be used within the same group). If the members are identified by reference then the following columns are required:

- **TTYPEn<sub>1</sub>=‘MEMBER\_EXTENSION’ – character field:** Contains the value of the XTENSION keyword from the group member’s header. In the case of primary HDUs where there is no required XTENSION keyword, the value of ‘PRIMARY’ will be used instead. Therefore, the current valid entries for this column are ‘PRIMARY<sub>1</sub>’, ‘TABLE<sub>1</sub>’, ‘BINTABLE’, ‘IMAGE<sub>1</sub>’ or any other IAU FITS Working Group registered XTENSION value. Note that the single quotation marks are used only to designate the string boundaries and are NOT to be included with the XTENSION values in the column entries; the trailing blanks shown in each string are optional. This field may contain the FITS null value appropriate for this column type if the value is unknown (e.g., if the position identification method described below is used to identify the member location).

- **TTYPEn<sub>2</sub>=‘MEMBER\_NAME’ – character field:** Contains the value of the EXTNAME keyword from the group member’s header. In the case of primary HDUs where the EXTNAME keyword is not defined or when the member extension has no EXTNAME keyword present, this field may contain the FITS null value appropriate for the column type.
- **TTYPEn<sub>3</sub>=‘MEMBER\_VERSION’ – integer field:** Contains the value of the EXTVER keyword from the group member’s header. In the case of primary HDUs, or if the EXTVER keyword is not present in the member header then a value of 1 should be assumed.

If members are identified by file position then the following column is required:

- **TTYPEn<sub>4</sub>=‘MEMBER\_POSITION’ – integer field:** Contains a group member’s position within its FITS file. The file’s primary header is given a position value of 0, the first extension is given a position value of 1, and so on. If for some reason a group member’s ‘MEMBER\_POSITION’ value becomes invalid or undefined, then this column field should be filled with the FITS null value appropriate for the column format.

If some or all of the group members reside in FITS files separate from the group table itself then the following two columns are also required:

- **TTYPEn<sub>5</sub>=‘MEMBER\_LOCATION’ – character field:** Contains the location of the group member’s FITS file using Uniform Resource Identifiers. If the FITS file resides on the same computer system as the group table, then partial URIs may be used instead of absolute URIs. If the group member resides in the same FITS file as the group table, or the MEMBER\_LOCATION value becomes invalid then this field may be filled with the FITS null value appropriate for the column type.
- **TTYPEn<sub>6</sub>=‘MEMBER\_URI\_TYPE’ – character field:** Contains the mnemonic for the Uniform Resource Identifier type used in the corresponding MEMBER\_LOCATION field. Recommended values for this column field are ‘URL’ for the Uniform Resource Locator and ‘URN’ for the Uniform Resource Name. As other URI types are defined their mnemonics will also become acceptable values for this field. In cases where the MEMBER\_URI\_TYPE is undefined (such as a null or blank MEMBER\_LOCATION field value) this field may contain the FITS null value appropriate for the column type.

Besides the table columns defined above, a group table may contain any number of user defined columns. Group table columns may appear in any order within the table and their TTYPEn values are not to be considered case-sensitive.

## 17.4. Keywords for Group Member Extensions

No additional keywords are required for HDUs that are members of a group. This rule is to ensure that all currently existing FITS files and their constituent HDUs may all be part of this convention. There are, however, several grouping related keywords whose presence is strongly recommended in newly created headers. The description of these keywords follow.

- **EXTNAME (character):** This keyword is the FITS reserved keyword EXTNAME. The use of EXTNAME allows HDUs of a given XTENSION type with similar structure and content to be identified with a common name tag. Additionally, the grouping convention uses EXTNAME to identify group members by reference (see section 17.2). For any HDU belonging to a group, the combination of XTENSION, EXTNAME and EXTVER keyword values should uniquely identify the HDU within its FITS file. An exception to this rule occurs when group tables are themselves members of a group. In this case the combination of EXTNAME and EXTVER keyword values alone must uniquely identify the HDU within its FITS file. This is because within a given FITS file the group tables may be built from a mix of ASCII (XTENSION=\_, 'TABLE\_...') and binary tables (XTENSION=\_, 'BINTABLE').
- **EXTVER (integer):** This keyword is the FITS reserved keyword EXTVER. The use of EXTVER allows unique identification of HDUs with a given XTENSION type and EXTNAME value. Additionally, the grouping convention uses EXTVER to identify group members by reference (see section 17.2). For any HDU belonging to a group, the combination of XTENSION, EXTNAME and EXTVER keyword values should uniquely identify the HDU within its FITS file; however, please note the exception outlined above.
- **GRPIDn (integer):** A series of indexed keywords that denote the group(s) to which an HDU belongs. The value of GRPIDn is the EXTVER value of the nth group table that the HDU is a member of. In this sense, the EXTVER value of a group table defines a unique ID for the group within a FITS file. If the value of GRPIDn is negative, then the HDU is a member of a group defined in another file. In this case the absolute value of GRPIDn is the EXTVER value of the external group table, and the corresponding GRPLCn keyword holds the URI of the FITS file containing the group's table. The GRPIDn keywords (and their associated GRPLCn keywords) not only identify HDUs as members of groups, but also allow group members to "point" back to their group tables. Any software that might change the position or nature of the HDU would know that it was a member of a group and that the group table would require updating.
- **GRPLCn (character):** A series of indexed keywords that contain the Uniform Resource Identifiers corresponding to the GRPIDn keyword. The GRPLCn values follow the same syntax rules as those specified for the group table's MEMBER\_LOCATION column (see section 17.3.2). It is unnecessary to have a GRPLCn keyword accompany a GRPIDn keyword when the value of the GRPIDn keyword is positive. Alternatively, the value of the GRPLCn keywords may be reference strings that refer to the member's group table HDU (see section 17.6).

```

XTENSION= 'BINTABLE'      / This is a binary table
BITPIX   =                8 / Table contains 8-bit bytes
NAXIS    =                2 / Number of axis
NAXIS1   =                4 / Width of table in bytes
NAXIS2   =                5 / Number of member entries
GCOUNT   =                1 / Mandatory FITS keyword
PCOUNT   =                0 / Number of bytes in HEAP area
TFIELDS  =                1 / Number of columns in table
EXTNAME  = 'GROUPING'    / This BINTABLE contains a gro
EXTVER   =                3 / The ID number of this group
GRPID1   =                1 / Part of group 1
GRPID2   =                2 / Part of group 2
TTYPE1   = 'MEMBER_POSITION' / Position of member within fi
TFORM1   = '1J'          / Datatype descriptor
END

```

Example 2: A group containing 150 members, some of which reside in FITS files different from that of the group table. This group is not a member of any other group, although it is the seventh group table defined in the FITS file. All member identification methods are used.

```

XTENSION= 'BINTABLE'      / This is a binary table
BITPIX   =                8 / Table contains 8-bit bytes
NAXIS    =                2 / Number of axis
NAXIS1   =                79 / Width of table in bytes
NAXIS2   =                150 / Number of member entries
GCOUNT   =                1 / Mandatory FITS keyword
PCOUNT   =                0 / Number of bytes in HEAP area
TFIELDS  =                6 / Number of columns in table
EXTNAME  = 'GROUPING'    / This BINTABLE contains a gro
EXTVER   =                7 / The ID number of this group
TTYPE1   = 'MEMBER_LOCATION' / URI of file containing membe
TFORM1   = '30A'         / Datatype descriptor
TTYPE2   = 'MEMBER_URI_TYPE' / URI type of MEMBER_LOCATION
TFORM2   = '3A'          / Datatype descriptor
TTYPE3   = 'MEMBER_POSITION' / Position of member within fi
TFORM3   = '1J'          / Datatype descriptor
TTYPE4   = 'MEMBER_XTENSION' / XTENSION keyword value of me
TFORM4   = '8A'          / Datatype descriptor
TTYPE5   = 'MEMBER_NAME' / EXTNAME keyword value of mem
TFORM5   = '30A'         / Datatype descriptor
TTYPE6   = 'MEMBER_VERSION' / EXTVER keyword value of memb
TFORM6   = '1J'          / Datatype descriptor
END

```

### 17.5. Example Group Table Headers

The following are examples of valid group table headers that use different combinations of identification methods.

Example 1: A group containing five members all of which reside in the same file as the group table. This group is itself a member of two other groups and both of those groups' tables reside in the same file as this extension. The member position identification method is used to locate member HDUs.

Example 3: A group containing 17 members, some of which reside in FITS files different from that of the group table. This group is a member of six other groups, two of which are defined in FITS files on other computer systems and one that is defined in a FITS file on the same computer system. The member reference identification and member file location methods are used. Two user defined columns are also present.

```
XTENSION= 'BINTABLE' / This is a binary table
BITPIX = 8 / Table contains 8-bit bytes
NAXIS = 2 / Number of axis
NAXIS1 = 180 / Width of table in bytes
NAXIS2 = 17 / Number of member entries
GCOUNT = 1 / Mandatory FITS keyword
PCOUNT = 0 / Number of bytes in header area
TFIELDS = 7 / Number of columns in table
EXTNAME = 'GROUPING' / This BINTABLE contains a group
EXTVER = 7 / The ID number of this group
GRPID1 = 3 / Member of group 3
GRPID2 = 6 / Member of group 6
GRPID3 = 18 / Member of group 18
GRPID4 = -1 / Member of external group
GRPLC4 = 'http://fits.gsfc.nasa.gov/FITS/file1.fits' / Location of
COMMENT FITS file containing group
GRPID5 = -5 / Member of external group
GRPLC5 = '/FITS/file5.fits' / Location of file containing group
GRPID6 = -2 / Member of external group
GRPLC6 = 'http://www.noao.edu/irafdir/file2.fits' / Location of
COMMENT FITS file containing group
TTYPE1 = 'USER_INFO_1' / A user supplied column
TFORM1 = '25J' / Datatype descriptor
TTYPE2 = 'MEMBER_LOCATION' / URI of file containing member
TFORM2 = '30A' / Datatype descriptor
TTYPE3 = 'MEMBER_XTENSION' / XTENSION keyword value of member
TFORM3 = '8A' / Datatype descriptor
TTYPE4 = 'MEMBER_NAME' / EXTNAME keyword value of member
TFORM4 = '30A' / Datatype descriptor
TTYPE5 = 'USER_INFO_2' / A user supplied column
TFORM5 = '5A' / Datatype descriptor
TTYPE6 = 'MEMBER_VERSION' / EXTVER keyword value of member
TFORM6 = '1J' / Datatype descriptor
TTYPE7 = 'MEMBER_URI_TYPE' / URI type of MEMBER_LOCATION field
TFORM7 = '3A' / Datatype descriptor
END
```

Example 4: A group containing 82 members, some of which reside in FITS files different from that of the group table. This group is a member of three other groups, and makes use of the member position and member file location methods. One user defined column is present. Note that in this example an ASCII table (as opposed to a binary table) is used to define the group.

```
XTENSION= 'TABLE' / This is an ASCII table
BITPIX = 8 / Table contains 8-bit ASCII bytes
NAXIS = 2 / Number of axis
NAXIS1 = 46 / Width of table in bytes
NAXIS2 = 82 / Number of member entries
GCOUNT = 1 / Mandatory FITS keyword
PCOUNT = 0 / Mandatory FITS keyword
TFIELDS = 4 / Number of columns in table
EXTNAME = 'GROUPING' / This TABLE contains a group
EXTVER = 31 / The ID number of this group
GRPID1 = 3 / Member of group 3
GRPID2 = 9 / Member of group 9
GRPID3 = 27 / Member of group 27
TTYPE1 = 'USER_INFO_1' / A user supplied column
TFORM1 = 'E10.3' / Datatype descriptor
TBCOL1 = 1 / Starting table column for file location of
TTYPE2 = 'MEMBER_LOCATION' / URI of file containing member
TFORM2 = 'A30' / Datatype descriptor
TBCOL2 = 11 / Starting table column for file location of
TTYPE3 = 'MEMBER_URI_TYPE' / URI type of MEMBER_LOCATION field
TFORM3 = 'A5' / Datatype descriptor
TBCOL3 = 41 / Starting table column for file location of
TTYPE4 = 'MEMBER_POSITION' / XTENSION keyword value of member
TFORM4 = 'I3' / Datatype descriptor
TBCOL4 = 44 / Starting table column for file location of
END
```

#### 4.7.6 Acknowledgments

We gratefully acknowledge the support of the NASA Applied Information Systems Research Program, under which this effort is partially funded.

#### 4.7.7 Appendix 1. Reference Strings

In certain circumstances, it may be convenient to point, or *refer*, to a HDU from another HDU. Such references neither imply or require the hierarchical association information as allowed by grouping table structures, but still serve a similar function by pointing to another data structure residing in a separate HDU.

If *referring* to a single HDU is preferable to forming a hierarchical association and including the given HDU as a member, then keyword and table column values may employ the same syntax as used for the identification of group members. For notational convenience, thus allowing all the information to be included in a single keyword value or table column entry, the reference should be expressed as a single character string of either type 1 format,

```
'MEMBER_LOCATION': 'MEMBER_XTENSION': 'MEMBER_EXTNAME': 'MEMBER_EXTVER'
```

or of type 2 format,

```
'MEMBER.LOCATION': 'MEMBER.POSITION'
```

where each quantity enclosed in single quotation marks is replaced by its corresponding value as defined in section 17.3.2. The colons (':', ASCII 58) appearing in the expressions are sig-

nificant and must be used to separate the fields of the string. Such expressions are known as *reference strings*.

Default values in the HDU reference strings are permitted but must obey the following rules. Note that by implication a reference string may begin with a colon field separator (':', ASCII 58) but may not terminate with a colon field separator.

- For type 1 format reference strings, the 'MEMBER\_XTENSION' and 'MEMBER\_EXTNAME' fields must always be specified.
- For type 1 format reference strings, a non-existent 'MEMBER\_EXTVER' is permitted and infers an EXTVER value of 1.
- For type 2 format reference strings, one of the two possible fields ('MEMBER\_LOCATION' or 'MEMBER\_POSITION') must always be specified, but see the rule below on non-existent 'MEMBER\_LOCATION' fields.
- Type 1 and type 2 format reference strings with a non-existent 'MEMBER\_LOCATION' value are permitted and infer that the referred-to HDU resides in the same FITS file as the HDU containing the reference string. To denote the absence of the 'MEMBER\_LOCATION' value, the first character of the reference string shall be a colon (':', ASCII 58).
- A reference string containing only the 'MEMBER\_LOCATION' field shall infer a type 2 format with a 'MEMBER\_POSITION' value of 1 (i.e., the first non-primary array FITS file extension). Note that a reference string of this form completely conforms to the syntax of a URI.

Below are examples of valid reference strings. In each case the following values are assumed:

- 'MEMBER\_LOCATION' = file://www.archive.edu/archive/sample.fits,
- 'MEMBER\_XTENSION' = BINTABLE,
- 'MEMBER\_EXTNAME' = EVENTS,
- 'MEMBER\_EXTVER' = 1, and
- 'MEMBER\_POSITION' = 1.

Note that the values of 'MEMBER\_EXTVER' and 'MEMBER\_POSITION' chosen for the examples demonstrate the use of the default reference string fields; the choice of different values would make the default cases invalid.

- If the referenced HDU resides in a different FITS file and on a different computer system:
  - file://www.archive.edu/archive/sample.fits:BINTABLE:EVENTS:1
  - file://www.archive.edu/archive/sample.fits:BINTABLE:EVENTS (note: using default value for 'MEMBER\_EXTVER')
  - file://www.archive.edu/archive/sample.fits:1
  - file://www.archive.edu/archive/sample.fits (note: using default value for 'MEMBER\_POSITION')
- If the referenced HDU resides in a different FITS file but on the same computer system:
  - /archive/sample.fits:BINTABLE:EVENTS:1 (note: absolute file path specified)
  - archive/sample.fits:BINTABLE:EVENTS:1 (note: relative file path specified)
  - sample.fits:BINTABLE:EVENTS:1 (note: relative file path specified)
  - /archive/sample.fits:BINTABLE:EVENTS (note: using default value for 'MEMBER\_EXTVER')

- /archive/sample.fits:1
- sample.fits (note: using default value for 'MEMBER\_POSITION')
- If the referenced HDU resides in the same FITS file:
  - :BINTABLE:EVENTS:1
  - :BINTABLE:EVENTS (note: using default value for 'MEMBER\_EXTVER')
  - :1

Please note that reference strings are meant only to supplement and enhance the hierarchical grouping convention as described above. In particular, reference strings should be used sparingly and with care; they do not provide the same level of data format structure and long-term archival stability as the grouping tables themselves.

## 17.8. Appendix II. Application Program Interface

This appendix describes an application program interface (API) in ANSI C that was implemented by the ISDC to facilitate creating and managing grouping tables by the INTEGRAL mission application software. Use of this particular API is not required and is shown here only for informational purposes. This API software is distributed and supported as a component of the CFITSIO software library that is maintained by the HEASARC at NASA/GSFC.

This API provides functions for the creation and manipulation of FITS HDU Groups, as defined in the "Hierarchical Grouping Convention for FITS". A group is a collection of HDUs whose association is defined by a *grouping table*. HDUs which are part of a group are referred to as *member HDUs* or simply as *members*. Grouping table member HDUs may themselves be grouping tables, thus allowing for the construction of open-ended hierarchies of HDUs.

Grouping tables contain one row for each member HDU. The grouping table columns provide identification information that allows applications to reference or "point to" the member HDUs. Member HDUs are expected, but not required, to contain a set of GRPIDn/GRPLCn keywords in their headers for each grouping table that they are referenced by. In this sense, the GRPIDn/GRPLCn keywords "link" the member HDU back to its Grouping table. Note that a member HDU need not reside in the same FITS file as its grouping table, and that a given HDU may be referenced by up to 999 grouping tables simultaneously.

Grouping tables are implemented as FITS binary tables with up to six pre-defined column TYPEn values:

```
'MEMBER_XTENSION', 'MEMBER_NAME',  
'MEMBER_VERSION', 'MEMBER_POSITION',  
'MEMBER_URI_TYPE' and 'MEMBER_LOCATION'.
```

The first three columns allow member HDUs to be identified by reference to their XTENSION, EXTNAME and EXTVER keyword values. The fourth column allows member HDUs to be identified by HDU position within their FITS file. The last two columns identify the FITS file in which the member HDU resides, if different from the grouping table FITS file.

Additional user defined "auxiliary" columns may also be included with any grouping table. When a grouping table is copied or modified the presence of auxiliary columns is always taken into account by the grouping support functions; however, the grouping support functions cannot directly make use of this data.

If a grouping table column is defined but the corresponding member HDU information is unavailable then a null value of the appropriate data type is inserted in the column field. Integer columns (MEMBER\_POSITION, MEMBER\_VERSION) are defined with a TNULLn value of zero (0). Character field columns (MEMBER\_XTENSION, MEMBER\_NAME, MEMBER\_URI\_TYPE, MEMBER\_LOCATION) utilize an ASCII null character to denote a null field value.

The grouping support functions belong to two basic categories: those that work with grouping table HDUs and those that work with member HDUs. Two functions, fits\_copy\_group() and fits\_remove\_group(), have the option to recursively copy/delete entire groups. Care should be taken when employing these functions in recursive mode as poorly defined groups could cause unpredictable results. The problem of a grouping table directly or indirectly referencing itself (thus creating an infinite loop) is protected against; in fact, neither function will attempt to copy or delete an HDU twice.

### 17.8.1. Grouping Table Routines

1. Create (append) a grouping table at the end of the current FITS file pointed to by fptr. The grpname parameter provides the grouping table name (GRPNAME keyword value) and may be set to NULL if no group name is to be specified. The grouptype parameter specifies the desired structure of the grouping table and may take on the values: GT\_ID\_ALL\_URI (all columns created), GT\_ID\_REF (ID by reference columns), GT\_ID\_POS (ID by position columns), GT\_ID\_ALL (ID by reference and position columns), GT\_ID\_REF\_URI (ID by reference and FITS file URI columns), and GT\_ID\_POS\_URI (ID by position and FITS file URI columns).

```
int fits_create_group(fitsfile *fptr, char *grpname,  
                    int grouptype, int *status)
```

2. Create (insert) a grouping table just after the CHDU of the current FITS file pointed to by fptr. All HDUs below the insertion point will be shifted downwards to make room for the new HDU. The grpname parameter provides the grouping table name (GRPNAME keyword value) and may be set to NULL if no group name is to be specified. The grouptype parameter specifies the desired structure of the grouping table and may take on the values: GT\_ID\_ALL\_URI (all columns created), GT\_ID\_REF (ID by reference columns), GT\_ID\_POS (ID by position columns), GT\_ID\_ALL (ID by reference and position columns), GT\_ID\_REF\_URI (ID by reference and FITS file URI columns), and GT\_ID\_POS\_URI (ID by position and FITS file URI columns).

```
int fits_insert_group(fitsfile *fptr, char *grpname,  
                    int grouptype, int *status)
```

3. Change the structure of an existing grouping table pointed to by gfptr. The grouptype parameter (see fits\_create\_group() for valid parameter values) specifies the new structure of the grouping table. This function only adds or removes grouping table columns, it does not add or delete group members (i.e., table rows). If the grouping table already has the desired structure then no operations are performed and function simply returns with a (0) success status code. If the requested structure change creates new grouping table columns, then the column values for all existing members will be filled with the null values appropriate to the column type.

```
int fits_change_group(fitsfile *gfptr, int grouptype, i
```

4. Remove the group defined by the grouping table pointed to by gfptr, and optionally all the group member HDUs. The rmopt parameter specifies the action to be taken for all members of the group defined by the grouping table. Valid values are: OPT\_RM\_GPT (delete only the grouping table) and OPT\_RM\_ALL (recursively delete all HDUs that belong to the group). Any groups containing the grouping table gfptr as a member are updated, and if rmopt == OPT\_RM\_GPT all members have their GRPIDn and GRPLCn keywords updated accordingly. If rmopt == OPT\_RM\_ALL, then other groups that contain the deleted members of gfptr are updated to reflect the deletion accordingly.

```
int fits_remove_group(fitsfile *gfptr, int rmopt, int *
```

5. Copy (append) the group defined by the grouping table pointed to by `infptr`, and optionally all group member HDUs, to the FITS file pointed to by `outfptr`. The `cpopt` parameter specifies the action to be taken for all members of the group `infptr`. Valid values are: `OPT_GCP_GPT` (copy only the grouping table) and `OPT_GCP_ALL` (recursively copy ALL the HDUs that belong to the group defined by `infptr`). If the `cpopt == OPT_GCP_GPT` then the members of `infptr` have their `GRPIDn` and `GRPLCn` keywords updated to reflect the existence of the new grouping table `outfptr`, since they now belong to the new group. If `cpopt == OPT_GCP_ALL` then the new grouping table `outfptr` only contains pointers to the copied member HDUs and not the original member HDUs of `infptr`. Note that, when `cpopt == OPT_GCP_ALL`, all members of the group defined by `infptr` will be copied to a single FITS file pointed to by `outfptr` regardless of their file distribution in the original group.

```
int fits_copy_group(fitsfile *infptr, fitsfile *outfptr,
                  int cpopt, int *status)
```

6. Merge the two groups defined by the grouping table HDUs `infptr` and `outfptr` by combining their members into a single grouping table. All member HDUs (rows) are copied from `infptr` to `outfptr`. If `mgopt == OPT_MRG_COPY` then `infptr` continues to exist unaltered after the merge. If the `mgopt == OPT_MRG_MOV` then `infptr` is deleted after the merge. In both cases, the `GRPIDn` and `GRPLCn` keywords of the member HDUs are updated accordingly.

```
int fits_merge_groups(fitsfile *infptr, fitsfile *outfptr,
                    int mgopt, int *status)
```

7. "Compact" the group defined by grouping table pointed to by `gfptr`. The compaction is achieved by merging (via `fits_merge_groups()`) all direct member HDUs of `gfptr` that are themselves grouping tables. The `cmopt` parameter defines whether the merged grouping table HDUs remain after merging (`cmopt == OPT_CMT_MBR`) or if they are deleted after merging (`cmopt == OPT_CMT_MBR_DEL`). If the grouping table contains no direct member HDUs that are themselves grouping tables then this function does nothing. Note that this function is not recursive, i.e., only the direct member HDUs of `gfptr` are considered for merging.

```
int fits_compact_group(fitsfile *gfptr, int cmopt,
```

8. Verify the integrity of the grouping table pointed to by `gfptr` to make sure that all group members are accessible and that all links to other grouping tables are valid. The `firstfailed` parameter returns the member ID (row number) of the first member HDU to fail verification (if positive value) or the first group link to fail (if negative value). If `gfptr` is successfully verified then `firstfailed` contains a return value of 0.

```
int fits_verify_group(fitsfile *gfptr, long *firstfailed, int *status)
```

9. Open a grouping table that contains the member HDU pointed to by `mfptr`. The grouping table to open is defined by the `grpidx` parameter, which contains the keyword index value of the `GRPIDn/GRPLCn` keyword(s) that link the member HDU `mfptr` to the grouping table. If the grouping table resides in a file other than the member HDUs file then an attempt is first made to open the file readwrite, and failing that

readonly. A pointer to the opened grouping table HDU is returned in `gfptr`.

Note that it is possible, although unlikely and undesirable, for the `GRPIDn/GRPLCn` keywords in a member HDU header to be non-continuous, e.g., `GRPID1, GRPID2, GRPID5, GRPID6`. In such cases, the `grpidx` index value specified in the function call shall identify the (`grpidx`)th `GRPID` value. In the above example, if `grpidx == 3`, then the group specified by `GRPID5` would be opened.

```
int fits_open_group(fitsfile *mfptr, int group,
                  fitsfile **gfptr, int *status)
```

10. Add a member HDU to an existing grouping table pointed to by `gfptr`. The member HDU may either be pointed to `mfptr` (which must be positioned to the member HDU) or, if `mfptr == NULL`, identified by the `hdupos` parameter (the HDU position number, Primary array == 1) if both the grouping table and the member HDU reside in the same FITS file. The new member HDU shall have the appropriate `GRPIDn` and `GRPLCn` keywords created in its header. Note that if the member HDU is already a member of the group then it will not be added a second time.

```
int fits_add_group_member(fitsfile *gfptr, fitsfile *mfptr,
                        int hdupos, int *status)
```

### 17.8.2. Group Member Routines

1. Return the number of member HDUs in a grouping table `gfptr`. The number member HDUs is just the `NAXIS2` value (number of rows) of the grouping table.

```
int fits_get_num_members(fitsfile *gfptr, long *nmember,
                       int *status)
```

2. Return the number of groups to which the HDU pointed to by `mfptr` is linked, as defined by the number of `GRPIDn/GRPLCn` keyword records that appear in its header. Note that each time this function is called, the indices of the `GRPIDn/GRPLCn` keywords are checked to make sure they are continuous (ie no gaps) and are re-enumerated to eliminate gaps if found.

```
int fits_get_num_groups(fitsfile *mfptr, long *nmembers,
                      int *status)
```

3. Open a member of the grouping table pointed to by `gfptr`. The member to open is identified by its row number within the grouping table as given by the parameter 'member' (first member == 1) . A fitsfile pointer to the opened member HDU is returned as `mfptr`. Note that if the member HDU resides in a FITS file different from the grouping table HDU then the member file is first opened readwrite and, failing

```
int fits_open_member(fitsfile *gfptr, long member,
                   fitsfile **mfptr, int *status)
```

4. Copy (append) a member HDU of the grouping table pointed to by `gfptr`. The member HDU is identified by its row number within the grouping table as given by the parameter 'member' (first member == 1). The copy of the group member

HDU will be appended to the FITS file pointed to by `mfptr`, and upon return `mfptr` shall point to the copied member HDU. The `cpopt` parameter may take on the following values: `OPT_MCP_ADD` which adds a new entry in `gfptr` for the copied member HDU, `OPT_MCP_NADD` which does not add an entry in `gfptr` for the copied member, and `OPT_MCP_REPL` which replaces the original member entry with the copied member entry.

```
int fits_copy_member(fitsfile *gfptr, fitsfile *mfptr,
                    long member, int cpopt, >
```

5. Transfer a group member HDU from the grouping table pointed to by `infptr` to the grouping table pointed to by `outfptr`. The member HDU to transfer is identified by its row number within `infptr` as specified by the parameter 'member' (first member == 1). If `tfopt == OPT_MCP_ADD` then the member HDU is made a member of `outfptr` and remains a member of `infptr`. If `tfopt == OPT_MCP_MOV` then the member HDU is deleted from `infptr` after the transfer to `outfptr`.

```
int fits_transfer_member(fitsfile *infptr, fitsfile *outfptr,
                        long member, int tfopt, int *status)
```

6. Remove a member HDU from the grouping table pointed to by `gfptr`. The member HDU to be deleted is identified by its row number in the grouping table as specified by the parameter 'member' (first member == 1). The `rmopt` parameter may take on the following values: `OPT_RM_ENTRY` which removes the member HDU entry from the grouping table and updates the member's `GRPIDn/GRPLCn` keywords, and `OPT_RM_MBR` which removes the member HDU entry from the grouping table and deletes the member HDU itself.

```
int fits_remove_member(fitsfile *fptr, long member,
                       int rmopt, int *status)
```

## 17.9. References

Berners-Lee, Tim, 1994. "World Wide Web Initiative", CERN - European Particle Physics Lab. <http://info.cern.ch/hypertext/WWW/TheProject.html>.

Cotton, W. D., Tody, D. and Pence W., 1994. "Binary Table Extension to FITS: A Proposal", version dated June 13, 1994.

Grosbøl, P., Harten, R. H., Greisen, E. W., and Wells, D. C., 1988. "Generalized extensions and blocking factors for FITS," *Astronomy and Astrophysics Suppl.*, 73, 359-364.

Harten, R. H., Grosbøl, P., Greisen, E. W., and Wells, D. C., 1988. "The FITS tables extension", *Astronomy and Astrophysics Suppl.*, 73, 365-372.

Ponz, J. D., Thompson, R. W., and Munoz, J. R., 1994. "FITS Image Extension", *Astronomy and Astrophysics Suppl.*, vol 105, pp 53-55.