Binary Table Extension to FITS: A Proposal

W. D. Cotton, N.R.A.O.

DRAFT 8 April 1991

ABSTRACT

This paper describes the FITS binary tables which are a flexible and efficient means of transmitting a wide variety of data structures. Table entries may be a mixture of a number of numerical, logical and character data types. In addition, each entry is allowed to be a single dimensioned array. Numeric data are kept in IEEE formats.

1 Introduction

The Flexible Image Transport System (FITS) (Wells *et al.* 1981 and Greisen and Harten 1981) has been used for a number of years both as a means of transporting data between computers and/or processing systems and as an archival format for a variety of astronomical data. The success of this system has resulted in the introduction of enhancements. In particular, considerable use has been made of the records following the "main" data file. Grosbøl *et al.* 1988 introduced a generalized header format for extension "files" following the "main" data file but in the same physical file. Harten *et al.* 1988 defined an ASCII table structure which could convey information that could be conveniently printed as a table. This paper generalizes the ASCII tables and defines an efficient means for conveying a wide variety of data structures as "extension" files.

2 Binary tables

The binary tables are tables in the sense that they are organized into rows and columns. They are multi-dimensional since an entry, or set of values associated with a given row and column, can be an array of arbitrary size. These values are represented in a standardized binary form. Each row in the table contains an entry for each column. This entry may be one of a number of different data types, 8 unsigned integers, 16 or 32 bit signed integers, logical, character, bit, 32 or 64 bit floating point or complex values. The data type and dimensionality are independently defined for each column but each row must have the same structure. Additional information associated with the table may be included in the table header as keyword/value pairs.

The binary tables come after the "main" data file, if any in a FITS file and follow the standards for generalized extension tables defined by Grosbøl *et al.* 1988. The use of the binary tables requires the use of a single additional keyword in the main header:

1. EXTEND (logical) if true (ASCII 'T') indicates that there *may* be extension files following the data records and, if there are, that they conform to the generalized extension file header standards.

3 Table Header

The table header begins at the first byte in the first record following the last record of main data (if any) or following the last record of the previous extension file. The format of the binary table header is such that a given FITS reader can decide if it wants (or understands) it and can skip the table if the reader decides it doesn't.

A table header consists of one or more 2880 8-bit byte logical records each containing 36 80-byte "card images" in the form:

```
keyword = value    / comment
```

where the keyword begins in column 1 and contains up to eight characters and the value begins in column 10 or later. Keyword/value pairs in binary table headers conform to standard FITS usage.

The number of columns in the table is given by the value associated with keyword TFIELDS. The type, dimensionality, labels, units, blanking values, and display formats for entries in column nnn may be defined by the values associated with the keywords TFORMnnn, TTYPEnnn, TUNITnnn, TNULLnnn, and TDISPnnn. Of these only TFORMnnn is required but the use of TTYPEnnn is strongly recommended. An entry may be omitted from the table, but still defined in the header, by using a zero element count in the TFORMnnn entry.

The required keywords XTENSION, BITPIX, NAXIS, NAXIS1, NAXIS2, PCOUNT, GCOUNT and TFIELDS must be in order; other keywords follow these in an arbitrary order. The required keywords in a binary table header record are:

1. XTENSION (character) indicates the type of extension file, this must be the first keyword in the header. This is 'BINTABLE' for the binary tables.

2. BITPIX (integer) gives the number of bits per "pixel" value. For binary tables this value is 8.

3. NAXIS (integer) gives the number of "axes"; this value is 2 for binary tables.

4. NAXIS1 (integer) gives the number of 8 bit bytes in each "row". This should correspond to the sum of the values defined in the TFORMnnn keywords.

5. NAXIS2 (integer) gives the number of rows in the table,

6. PCOUNT (integer) gives the number of "random" parameters before each group. This is 0 for binary tables.

7. GCOUNT (integer) gives the number of groups of data defined as for the random group main data records. This is 1 for binary tables.

8. TFIELDS (integer) gives the number of fields (columns) present in the table.

9. TFORMnnn [1] (character) gives the size and data type of field nnn. Allowed values of nnn range from 1 to the value associated with TFIELDS. Allowed values of TFORMnnn are of the form rL, rX, rI, rJ, rA, rE, or rD, rB, rC (logical, bit, 16-bit integers, 32-bit integers, characters, single precision and double precision, unsigned bytes and complex = pair of single precision values) where r=number of elements. If the element count is absent, it is assumed to be 1. A value of zero is allowed.

10. END is always the last keyword in a header. The remainder of the FITS logical (2880 byte) record following the END keyword is blank filled.

The optional standard keywords are:

1. EXTNAME (character) can be used to give a name to the extension file to distinguish it from other similar files. The name may have a hierarchal structure giving its relation to other files (e.g., "map1.cleancomp")

2. EXTVER (integer) gives a version number which can be used with EXTNAME to identify a file.

3. EXTLEVEL (integer) specifies the level of the extension file in a hierarchal structure. The default value for EXTLEVEL should be 1.

4. TTYPEnnn (character) gives the label for field nnn. Any number of characters are allowed but the first 8 should be unique.

5. TUNITnnn (character) gives the physical units of field nnn.

6. TSCALnnn (floating) gives the scale factor for field nnn. True_value = FITS_value $*$ TSCAL + TZERO. Note: TSCALnnn and TZEROnnn are not relevant to A, L, or X format fields. Default value is 1.0.

[1] The "nnn" in keyword names indicates an integer index in the range 1 - 999. The integer is left justified with no leading zeroes, e.g. TFORM1, TFORM19, etc.

7. TZEROnnn (floating) gives the offset for field nnn. (See TSCALnnn.) Default value is 0.0.

8. TNULLnnn (integer) gives the undefined value for integer (B, I, and J) field nnn. Section 5 discusses the conventions for indicating invalid data of other data types.

9. TDISPnnn (character) gives the Fortran format suggested for the display of field nnn. Each byte of bit and byte arrays will be considered to be a signed integer for purposes of display. The allowed forms are Aw, Lw, Iw.m, Fw.d, Ew.d, Gw.d, and Dw.d where w is the width of the displayed value in characters, m is the minimum number of digits possibly requiring leading zeroes and d is the number of digits to the right of the decimal. All entries in a field are displayed with a single, repeated format. Any TSCALnnn and TZEROnnn values will be applied before display of the value. Note that characters and logical values may be null (zero byte) terminated.

10. TDIMnnn (character) This keyword is reserved for use by the convention described in the appendix.

11. AUTHOR (character) gives the name of the author or creator of the table.

12. REFERENC (character) gives the reference for the table.

Nonstandard keyword/value pairs adhering to the FITS keyword standards are allowed although a reader may chose to ignore them.

4 Conventions for Multidimensional Arrays

There is commonly a need to use data structures more complex than the one dimensional definition of the table entries defined for this table format. Multidimensional arrays, or more complex structures, may be implemented by passing dimensions or other structural information as either column entries or keywords in the header. Passing the dimensionality as column entries has the advantage that the array can have variable dimension (subject to a fixed maximum size and storage usage). A convention is suggested in the Appendix.

5 Table Data Records

The binary table data logical records begin with the next record following the last header record. Data for a given column are contiguous and in order of increasing array index. Column entries are arranged in order of increasing column number. All data for a given row are contiguous and rows are given in order of increasing row number. All 2880 byte logical records are completely filled with

no extra bytes between column entries or rows. Column entries or rows do not necessarily begin in the first byte of a 2880 byte record. Note that this implies that a given word may not be aligned in the record along word boundaries of its type; words may even span 2880 byte records. The last 2880 byte record should be zero byte filled past the end of the valid data.

If word alignment is ever considered important for efficiency considerations then this may be accomplished by the proper design of the table. The simplest way to accomplish this is to order the columns by data type (D, C, E, J, I, B, L, A, X) and then add sufficient padding in the form of a dummy column of type B with the number of elements such that the size of a row is either an integral multiple of 2880 bytes or an integral number of rows is 2880 bytes.

The data types are defined in the following list (r is the number of elements in the entry):

1. *rL.* A logical value consists of an ASCII "T" indicating true and "F" indicating false. A null character (zero byte) indicates an invalid value.

2. *rX.* A bit array will start in the most significant bit of the byte and the following bits in the order of decreasing significance in the byte. Bit significance is in the same order as for integers. A bit array entry consists of an integral number of bytes with trailing bits zero.

   No explicit null value is defined for bit arrays but if the capability of blanking bit arrays is needed it is recommended that one of the following conventions be adopted: 1) designate a bit in the array as a validity bit, 2) add an L type column to indicate validity of the array or 3) add a second bit array which contains a validity bit for each of the bits in the original array.

3. *rB* Unsigned 8-bit integer with bits in decreasing order of significance. Signed values may be passed with appropriate values of TSCALnnn and TZEROnnn.

4. *rI.* A 16-bit two's complement integer with the bits in decreasing order of significance. Unsigned values may be passed with appropriate values of TSCALnnn and TZEROnnn.

5. *rJ.* A 32-bit two's complement integer with the bits in decreasing order of significance. Unsigned values may be passed with appropriate values of TSCALnnn and TZEROnnn.

6. *rA.* Character strings are represented by ASCII characters in their natural order. Character strings may be terminated before its explicit length by an ASCII NULL character. An ASCII NULL as the first character will indicate a undefined string i.e. a NULL string. Legal characters are printable ASCII characters in the range ' ' (hex 20) to '~' (hex 7E)

inclusive and ASCII NULL after the last valid character. Strings the full length of the field need not be NULL terminated.

7. *rE.* Single precision floating point values are in IEEE 32-bit precision format in the order: sign bit, exponent and mantissa in decreasing order of signifigance. The IEEE NaN (not a number) values are used to indicate an invalid number; a value all bits set is recognized as a NaN. All IEEE special values are recognized.

8. *rD.* Double precision floating point values are in IEEE 64-bit precision format in the order: sign bit, exponent and mantissa in decreasing order of signifigance. The IEEE NaN values are used to indicate an invalid number; a value with all bits set is recognized as a NaN. All IEEE special values are recognized.

9. *rC* Complex values; these consist of a pair of IEEE 32-bit precision floating point values with the first being the real and the second the imaginary parts.

6 Example Binary Table Header

The following is an example of a binary table header which has 19 columns using a number of different data types and dimensions. Columns labeled "IFLUX", "QFLUX", "UFLUX", "VFLUX", "FREQOFF", "LSRVEL" and "RESTFREQ" are arrays of dimension 2. Columns labeled "SOURCE" and "CALCODE" are character strings of length 16 and 4 respectively. The nonstandard keywords "NO_IF", VELTYP", and "VELDEF" also appear at the end of the header. The first two lines of numbers are only present to show card columns and are not part of the table header.

```
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890123456789012345678901234
XTENSION= 'BINTABLE'           / Extension type
BITPIX  =                    8 / Binary data
NAXIS   =                    2 / Table is a matrix
NAXIS1  =                  184 / Width of table in bytes
NAXIS2  =                    1 / Number of entries in table
PCOUNT  =                    0 / Random parameter count
GCOUNT  =                    1 / Group count
TFIELDS =                   19 / Number of columns in each row
EXTNAME = 'AIPS SU '           / AIPS source table
EXTVER  =                    1 / Version number of table
TFORM1  = '1I      '           / Fortran format of column  1
TTYPE1  = 'ID. NO.         '   / Type (label) of column  1
TUNIT1  = '        '           / Physical units of column  1
TFORM2  = '16A     '           / Fortran format of column  2
```

```
TTYPE2  = 'SOURCE          '   / Type (label) of column  2
TUNIT2  = '                '   / Physical units of column  2
TFORM3  = '1I              '   / Fortran format of column  3
TTYPE3  = 'QUAL            '   / Type (label) of column  3
TUNIT3  = '                '   / Physical units of column  3
TFORM4  = '4A              '   / Fortran format of column  4
TTYPE4  = 'CALCODE         '   / Type (label) of column  4
TUNIT4  = '                '   / Physical units of column  4
TFORM5  = '2E              '   / Fortran format of column  5
TTYPE5  = 'IFLUX           '   / Type (label) of column  5
TUNIT5  = 'JY              '   / Physical units of column  5
TFORM6  = '2E              '   / Fortran format of column  6
TTYPE6  = 'QFLUX           '   / Type (label) of column  6
TUNIT6  = 'JY              '   / Physical units of column  6
TFORM7  = '2E              '   / Fortran format of column  7
TTYPE7  = 'UFLUX           '   / Type (label) of column  7
TUNIT7  = 'JY              '   / Physical units of column  7
TFORM8  = '2E              '   / Fortran format of column  8
TTYPE8  = 'VFLUX           '   / Type (label) of column  8
TUNIT8  = 'JY              '   / Physical units of column  8
TFORM9  = '2D              '   / Fortran format of column  9
TTYPE9  = 'FREQOFF         '   / Type (label) of column  9
TUNIT9  = 'HZ              '   / Physical units of column  9
TFORM10 = '1D              '   / Fortran format of column 10
TTYPE10 = 'BANDWIDTH       '   / Type (label) of column 10
TUNIT10 = 'HZ              '   / Physical units of column 10
TFORM11 = '1D              '   / Fortran format of column 11
TTYPE11 = 'RAEPO           '   / Type (label) of column 11
TUNIT11 = 'DEGREES '           / Physical units of column 11
TFORM12 = '1D              '   / Fortran format of column 12
TTYPE12 = 'DECEPO          '   / Type (label) of column 12
TUNIT12 = 'DEGREES '           / Physical units of column 12
TFORM13 = '1D              '   / Fortran format of column 13
TTYPE13 = 'EPOCH           '   / Type (label) of column 13
TUNIT13 = 'YEARS   '           / Physical units of column 13
TFORM14 = '1D              '   / Fortran format of column 14
TTYPE14 = 'RAAPP           '   / Type (label) of column 14
TUNIT14 = 'DEGREES '           / Physical units of column 14
TFORM15 = '1D              '   / Fortran format of column 15
TTYPE15 = 'DECAPP          '   / Type (label) of column 15
TUNIT15 = 'DEGREES '           / Physical units of column 15
TFORM16 = '2D              '   / Fortran format of column 16
TTYPE16 = 'LSRVEL          '   / Type (label) of column 16
TUNIT16 = 'M/SEC   '           / Physical units of column 16
TFORM17 = '2D              '   / Fortran format of column 17
TTYPE17 = 'RESTFREQ        '   / Type (label) of column 17
TUNIT17 = 'HZ              '   / Physical units of column 17
```

```
TFORM18 = '1D      '           / Fortran format of column 18
TTYPE18 = 'PMRA            '   / Type (label) of column 18
TUNIT18 = 'DEG/DAY '           / Physical units of column 18
TFORM19 = '1D      '           / Fortran format of column 19
TTYPE19 = 'PMDEC           '   / Type (label) of column 19
TUNIT19 = 'DEG/DAY '           / Physical units of column 19
NO_IF   =      2
VELTYP  = 'LSR     '
VELDEF  = 'OPTICAL '
END
```

7 Acknowledgments

The author would like to thank E. Greisen, D. Wells, P. Grosbøl, B. Hanisch, E. Mandel, E. Kemper and B. Schlesinger and many others for invaluable discussions and suggestions.

8 References

Wells, D. C., Greisen, E. W., and Harten R. H. 1981, "FITS: A Flexible Image Transport System", *Astron. Astrophys. Suppl*, vol. 44, pp 363 - 370.

Greisen E. W. and Harten R. H., 1981, "An Extension of FITS for Small Arrays of Data", *Astron. Astrophys. Suppl*, vol. 44, pp 371 - 374.
Astronomy and Astrophysics Supplement Series, vol. 44, pp 371 - 374.

Grosbøl, P., Harten, R. H., Greisen, E. W., and Wells, D. C. 1988, "Generalized Extension and Blocking Factors for FITS", *Astron. Astrophys. Suppl.*, vol. 73, pp 359-364.

Harten, Grosbøl, Greisen and Wells 1988, "The FITS tables Extension", *Astron. Astrophys. Suppl.*, vol. 73, pp 365-372.

Appendix

"Multidimensional Array" Convention

It is anticipated that binary tables will need to contain data structures more complex that those describable by the basic notation. Examples of these are multidimensional arrays and nonrectangular data structures. Suitable conventions may be defined to pass these structures using some combination of keyword/value pairs and table entries to pass the parameters of these structures.

One case, multidimensional arrays, is so common that it is prudent to describe a simple convention. The "Multidimensional array" convention consists of the following: any column with a dimensionality of 1 or larger will have an associated character keyword TDIMnnn='(l,n,m...)' where l, n, m ...are the dimensions of the array. The size implied by the TDIMnnn keyword will equal the element count specified in the TFORMnnn keyword. The adherence to this convention will be indicated by the presence of a TDIMnnn keyword. This convention is optional and will not preclude other conventions. This convention is not part of the proposed binary table definition.