Binary Table Extension to FITS: A Proposal

W. D. Cotton, N.R.A.O.
D. Tody, N.O.A.O.
and
W. Pence, G.S.F.C.
FINAL 13 June 1994

ABSTRACT

This paper describes the FITS binary tables which are a flexible and efficient means of transmitting a wide variety of data structures. Table rows may be a mixture of a number of numerical, logical and character data entries. 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 bit unsigned integers, 16 or 32 bit signed integers, logical, character, bit, 32 or 64 bit floating point or complex values. The datatype 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 not.

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) is used to tell the number of bytes *following* the regular portion of the table. These bytes are allowed but no meaning is attached to them in this document. PCOUNT should normally be 0 for binary tables (see however Appendix A).

   Note: This is a *serious* change from the recommended usage of PCOUNT in Greisen and Harten 1981 which defines PCOUNT as the number of bytes *preceeding* the regular portion of the entry. The Grosbøl *et al.* 1988 generalized extension header agreement adopted, but did not define, the PCOUNT keyword. We take this to allow specific extensions to redefine the location of the PCOUNT parameters. This change does not, and must not, affect conformance with the rules for determining the size of the table.

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, rD, rB, rC, rM, or rP (logical, bit, 16-bit integers, 32-bit integers, characters, single precision, double precision, unsigned bytes, complex {pair of single precision values}, double complex {pair of double precision values} and variable length array descriptor {64 bits}) where r=number of elements. If the element count is absent, it is assumed to be 1. A value of zero is allowed. Note: additional characters may follow the datatype code character but they are not defined in this document.

   The number of bytes determined from summing the TFORMnnn values must equal NAXIS1.

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] 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.

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 hierarchical structure giving its relation to other files (e.g., "map1.cleancomp")

2. EXTVER (integer) is a version number which can be used with EXTNAME to identify a file. The default value for EXTVER should be 1.

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

4. TTYPEnnn (character) gives the label for field nnn.

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 defined for A, L, or X format fields. For complex data types (C and M) TSCALnnn and TZEROnnn are the real part of the scaling and offset factors and the imaginary part is 0. The anticipated meaning of TSCALnnn and TZEROnnn for P fields is described in Appendix A. Default value is 1.0.

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 90 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, Bw.m (Binary, integers only), Ow.m (Octal, integers only), Zw.m (Hexidecimal, integers only) Fw.d, Ew.dEe, ENw.d, ESw.d, Gw.dEe, and Dw.dEe where w is the width of the displayed value in characters, m is the minimum number of digits possibly requiring leading zeroes, d is the number of digits to the right of the decimal, and e is the number of digits in the exponent. All entries in a field are displayed with a single, repeated format. If Fortran 90 formats are not available to a reader which prints a table then equivalent Fortran 77 formats may be substituted. Any TSCALnnn and TZEROnnn values should be applied before display of the value. Note that characters and logical values may be null (zero byte) terminated.

10. THEAP (integer). This keyword is reserved for use by the convention described in Appendix A.

11. TDIMnnn (character). This keyword is reserved for use by the convention described in Appendix B.

12. AUTHOR (character) gives the name of the author or creator of the table. This is the human or organization that collected the information given in this table.

13. 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; however, see Appendix A). A convention for arrays is suggested in Appendix B and a convention for arrays of character strings in Appendix C. These and any other conventions will not require a generalized FITS reader to know or understand their details.

5 Table Data Records

The binary table data records begin with the next logical record following the last header record. If the intersection of a row and column is an array then the elements of this array are contiguous and in order of increasing array index. Within a row, columns are stored in order of increasing column number. Rows are given in order of increasing row number. All 2880–byte logical records are completely filled with no extra bytes between columns or rows. Columns and 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 (M, D, C, P, 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. Any other values are illegal.

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. Such conventions are beyond the scope of this general format design and in general readers will not be expected to understand them.

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. A character string may be terminated before its explicit length by an ASCII NUL character. An ASCII NUL as the first character will indicate an undefined string i.e. a NULL string. Legal characters are printable ASCII characters in the range `' '` (decimal 32) to `'~'` (decimal 126) inclusive and ASCII NUL after the last valid character. Strings the full length of the field are not NULL terminated. Characters after the first ASCII NUL are not defined.

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 significance. The IEEE NaN (not a number) 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.

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 significance. 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.* A Complex value consists of a pair of IEEE 32-bit precision floating point values with the first being the real and the second the imaginary part. If either word contains a NaN value the complex value is invalid.

10. *rM.* Double precision complex values. These consist of a pair of IEEE 64-bit precision floating point values with the first being the real and the second the imaginary part. If either word contains a NaN value the complex value is invalid.

11. *rP.* Variable length array descriptor. An element is equal in size to a pair of 32-bit integers (i.e., 64 bits). The anticipated use of this data type is described in Appendix A. Arrays of type P are not defined; the r field is permitted, but values other than 0 or 1 are undefined. For purposes of printing, an entry of type P should be considered equivalent to 2J.

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
1234567890123456789012345678901234567890123456789012345678901234
XTENSION= 'BINTABLE'           / Extension type
BITPIX  =                    8 / Binary data
NAXIS   =                    2 / Table is a matrix
NAXIS1  =                  168 / Width of table row in bytes
NAXIS2  =                    5 / Number of rows 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      '           / 16-bit integer
TTYPE1  = 'ID. NO.         '   / Type (label) of column  1
```

```
TUNIT1  = '        '          / Physical units of column  1
TFORM2  = '16A     '          / Character string
TTYPE2  = 'SOURCE          '  / Type (label) of column  2
TUNIT2  = '        '          / Physical units of column  2
TFORM3  = '1I      '          / 16-bit integer
TTYPE3  = 'QUAL            '  / Type (label) of column  3
TUNIT3  = '        '          / Physical units of column  3
TNULL3  = 32767               / Undefined value for column 3
TFORM4  = '4A      '          / Character string
TTYPE4  = 'CALCODE         '  / Type (label) of column  4
TUNIT4  = '        '          / Physical units of column  4
TFORM5  = '2E      '          / Single precision array
TTYPE5  = 'IFLUX           '  / Type (label) of column  5
TUNIT5  = 'JY      '          / Physical units of column  5
TFORM6  = '2E      '          / Single precision array
TTYPE6  = 'QFLUX           '  / Type (label) of column  6
TUNIT6  = 'JY      '          / Physical units of column  6
TFORM7  = '2E      '          / Single precision array
TTYPE7  = 'UFLUX           '  / Type (label) of column  7
TUNIT7  = 'JY      '          / Physical units of column  7
TFORM8  = '2E      '          / Single precision array
TTYPE8  = 'VFLUX           '  / Type (label) of column  8
TUNIT8  = 'JY      '          / Physical units of column  8
TFORM9  = '2D      '          / Double precision array.
TTYPE9  = 'FREQOFF         '  / Type (label) of column  9
TUNIT9  = 'HZ      '          / Physical units of column  9
TSCAL9  = 1.0D9               / Scaling factor of column 9
TZERO9  = 0.0                 / Offset of column 9
TFORM10 = '1D      '          / Double precision
TTYPE10 = 'BANDWIDTH       '  / Type (label) of column 10
TUNIT10 = 'HZ      '          / Physical units of column 10
TFORM11 = '1D      '          / Double precision
TTYPE11 = 'RAEPO           '  / Type (label) of column 11
TUNIT11 = 'DEGREES '          / Physical units of column 11
TFORM12 = '1D      '          / Double precision
TTYPE12 = 'DECEPO          '  / Type (label) of column 12
TUNIT12 = 'DEGREES '          / Physical units of column 12
TFORM13 = '1D      '          / Double precision
TTYPE13 = 'EPOCH           '  / Type (label) of column 13
TUNIT13 = 'YEARS   '          / Physical units of column 13
TFORM14 = '1D      '          / Double precision
TTYPE14 = 'RAAPP           '  / Type (label) of column 14
TUNIT14 = 'DEGREES '          / Physical units of column 14
TFORM15 = '1D      '          / Double precision
TTYPE15 = 'DECAPP          '  / Type (label) of column 15
TUNIT15 = 'DEGREES '          / Physical units of column 15
TFORM16 = '2D      '          / Double precision array
```

```
TTYPE16 = 'LSRVEL         '   / Type (label) of column 16
TUNIT16 = 'M/SEC   '          / Physical units of column 16
TFORM17 = '2D      '          / Double precision array
TTYPE17 = 'RESTFREQ       '   / Type (label) of column 17
TUNIT17 = 'HZ      '          / Physical units of column 17
TDISP17 = 'D17.10'            / Display format of column 17
TFORM18 = '1D      '          / Double precision array
TTYPE18 = 'PMRA           '   / Type (label) of column 18
TUNIT18 = 'DEG/DAY '          / Physical units of column 18
TFORM19 = '1D      '          / Double precision array
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 authors would like to thank E. Greisen, D. Wells, P. Grosbøl, B. Hanisch, E. Mandel, E. Kemper, S. Voels, 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 A

"Variable Length Array" Facility

One of the most attractive features of binary tables is that any field of the table can be an array. In the standard case this is a fixed size array, i.e., a fixed amount of storage is allocated in each record for the array data - whether it is used or not. This is fine so long as the arrays are small or a fixed amount of array data will be stored in each record, but if the stored array length varies for different records, it is necessary to impose a fixed upper limit on the size of the array that can be stored. If this upper limit is made too large excessive wasted space can result and the binary table mechanism becomes seriously inefficient. If the limit is set too low then it may become impossible to store certain types of data in the table.

The "variable length array" construct presented here was devised to deal with this problem. Variable length arrays are implemented in such a way that, even if a table contains such arrays, a simple reader program which does not understand variable length arrays will still be able to read the main table (in other words a table containing variable length arrays conforms to the basic binary table standard). The implementation chosen is such that the records in the main table remain fixed in size even if the table contains a variable length array field, allowing efficient random access to the main table.

Variable length arrays are logically equivalent to regular static arrays, the only differences being 1) the length of the stored array can differ for different records, and 2) the array data is not stored directly in the table records. Since a field of any datatype can be a static array, a field of any datatype can also be a variable length array (excluding type P, the variable length array descriptor itself, which is not a datatype so much as a storage class specifier). Conventions such as TDIMnnn (see Appendix B) apply equally to both to variable length and static arrays.

A variable length array is declared in the table header with a special field datatype specifier of the form

```
rPt(maxelem)
```

where the "`P`" indicates the amount of space occupied by the array descriptor in the data record (64 bits), the element count "`r`" should be 0, 1, or absent, `t` is a character denoting the datatype of the array data (L, X, B, I, J, etc., but not P), and `maxelem` is a quantity guaranteed to be equal to or greater than the maximum number of elements of type `t` actually stored in a table record. There is no built-in upper limit on the size of a stored array; `maxelem` merely reflects the size of the largest array actually stored in the table, and is provided to avoid the need to preview the table when, for example, reading a table containing variable length elements into a database that supports only fixed size arrays. There may be additional characters in the TFORMnnn keyword following the "(`maxelem`)".

For example,

```
TFORM8  = 'PB(1800)'          / Variable length byte array
```

indicates that field 8 of the table is a variable length array of type byte, with a maximum stored array length not to exceed 1800 array elements (bytes in this case).

The data for the variable length arrays in a table is not stored in the actual data records; it is stored in a special data area, the heap, following the last fixed size data record. What is stored in the data record is an *array descriptor*. This consists of two 32 bit integer values: the number of elements (array length) of the stored array, followed by the zero-indexed byte offset of the first element of the array, measured from the start of the heap area. Storage for the array is contiguous. The array descriptor for field N as it would appear embedded in a data record is illustrated symbolically below.

```
field N-1  ;  { nelem  offset }  ;  field N+1
```

If the stored array length is zero there is no array data, and the offset value is undefined (it should be set to zero). The storage referenced by an array descriptor must lie entirely within the heap area; negative offsets are not permitted.

A binary table containing variable length arrays consists of three main segments, as follows:

```
table header
record storage area (data records)
heap area (variable array data)
```

The table header consists of one or more 2880 byte FITS logical records with the last record indicated by the keyword END somewhere in the record. The record storage area begins with the next 2880 byte logical record following the last header record and is NAXIS1*NAXIS2 bytes in length. The zero indexed byte offset of the heap measured from the start of the record storage area is given by the THEAP keyword in the header. If this keyword is missing the heap is assumed to begin with the byte immediately following the last data record, otherwise there may be a gap between the last stored record and the start of the heap. If there is no gap the value of the heap offset is NAXIS1*NAXIS2. The total length in bytes of the area following the last stored record (gap plus heap) is given by the PCOUNT keyword in the table header.

For example, suppose we have a table containing 5 168 byte records, with a heap area 2880 bytes long, beginning at an offset of 2880, thereby aligning the record storage and heap areas on FITS record boundaries (this alignment is not necessarily recommended but is useful for our example). The data portion of the table consists of 2 2880 byte FITS records, 840 bytes of which are used by the 5 table records, hence PCOUNT is 2*2880-840, or 4920 bytes.

```
NAXIS1  =                  168 / Width of table row in bytes
NAXIS2  =                    5 / Number of rows in table
PCOUNT  =                 4920 / Random parameter count
  ...
THEAP   =                 2880 / Byte offset of heap area
```

While the above description is sufficient to define the required features of the variable length array implementation, some hints regarding usage of the variable length array facility may also be useful.

Programs which read binary tables should take care to not assume more about the physical layout of the table than is required by the specification. For example, there are no requirements on the alignment of data within the heap. If efficient runtime access is a concern one may want to design the table so that data arrays are aligned to the size of an array element. In another case one might want to minimize storage and forgo any efforts at alignment (by careful design it is often possible to achieve both goals). Variable array data may be stored in the heap in any order, i.e., the data for record N+1 is not necessarily stored at a larger offset than that for record N. There may be gaps in the heap where no data is stored. Pointer aliasing is permitted, i.e., the array descriptors for two or more arrays may point to the same storage location (this could be used to save storage if two or more arrays are identical).

Byte arrays are a special case because they can be used to store a "typeless" data sequence. Since FITS is a machine independent storage format, some form of machine specific data conversion (byte swapping, floating point format conversion) is implied when accessing stored data with types such as integer and floating, but byte arrays are copied to and from external storage without any form of conversion.

An important feature of variable length arrays is that it is possible that the stored array length may be zero. This makes it possible to have a column of the table for which, typically, no data is present in each stored record. When data is present the stored array can be as large as necessary. This can be useful when storing complex objects as records in a table.

Accessing a binary table stored on a random access storage medium is straightforward. Since the data records in the main table are fixed in size they may be randomly accessed given the record number, by computing the offset. Once the record has been read in, any variable length array data may be directly accessed using the element count and offset given by the array descriptor stored in the data record.

Reading a binary table stored on a sequential access storage medium requires that a table of array descriptors be built up as the main table records are read in. Once all the table records have been read, the array descriptors are sorted by the offset of the array data in the heap. As the heap data is read, arrays are extracted sequentially from the heap and stored in the affected records using the back pointers to the record and field from the table of array descriptors. Since array aliasing is permitted, it may be necessary to store a given array in more than one field or record.

Variable length arrays are more complicated than regular static arrays and imply an extra data access per array to fetch all the data for a record. For this reason, it is recommended that regular static arrays be used instead of variable length arrays unless efficiency or other considerations require the use of a variable array.

This facility is still undergoing trials and is not currently part of the main binary table definition.

Appendix B

“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 2 or larger will have an associated character keyword TDIMnnn='(l,m,n,...)' where l, m, n,... are the dimensions of the array. The data is ordered such that the array index of the first dimension given (l) is the most rapidly varying and that of the last dimension given is the least rapidly varying. 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 in the form described above.

A character string is represented in a binary table by a one-dimensional character array, as described in section 5. For example, a FORTRAN CHARACTER*20 variable could be represented in a binary table as a character array declared as TFORMn = '20A '. Arrays of character strings, i.e., multidimensional character arrays, may be represented using the TDIMn notation. For example, if TFORMn='60A' and TDIMn='(5,4,3)' then the entry consists of a 4×3 array of strings of 5 characters each. (Variable length character strings are allowed by the convention described in Appendix C. One dimensional arrays of strings should use the convention in Appendix C rather than the “Multidimensional Array” convention.)

This convention is optional and will not preclude other conventions. This convention is not part of the proposed binary table definition.

Appendix C

"Substring Array" Convention

This appendix describes a layered convention for specifying that a character array field (TFORMnnn = 'rA') consists of an array of either fixed-length or variable-length substrings within the field. This convention utilizes the option described in the basic binary table definition document to have additional characters following the datatype code character in the TFORMnnn value field. The full form for the value of TFORMnnn within this convention is '`rA:SSTRw/nnn`' 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 the "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.

'`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 as described in the basic binary table definition document.

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 TFORMnnn = '`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. The "Multidimensional Array" convention described in Appendix B of this document provides a syntax using the TDIMnnn keyword for describing multidimensional arrays of any datatype which can also be used to represent arrays of fixed-length substrings. For a one dimensional array of substrings (a two dimensional array of characters) the "Substring Array" convention is preferred over the "Multidimensional Array" convention. Multidimensional arrays of (fixed length) strings require the use of the "Multidimensional Array" convention.

7. This substring convention may be used in conjunction with the "Variable Length Array" facility described in Appendix A of this document. In this case, the full form for the value of the TFORM keyword is:

```
TFORMnnn= 'rPA(maxelem):SSTRw/nnn' /variable-length substrings
TFORMnnn= 'rPA(maxelem):SSTRw'     / fixed-length substrings
```

This convention is optional and will not preclude other conventions. This convention is not part of the proposed binary table definition.