CHAPTER 14

TABLES IN AIPS

14.1 OVERVIEW

This chapter is an attempt to describe the format design for tabular extension files in AIPS. These files are organized in the usual rows and columns. Each column has a specified format and is stored in the appropriate binary form for the local computer. The columns are ordered on disk in an order appropriate to computer addressing, but are accessed in any desired logical column order via a look up list. The extension file contains not only the rows and columns, but also a variety of other information. Each column has an associated 24-character column "title" and an 8-character "units" field. Each row has a "selection" flag which allows the user to access temporarily a subset of his table. The strings used to specify the current selection are stored in the file for display. The file may also contain general information applying to the full table in the form of keyword/value pairs. This information will be called the table "header" data.

14.2 THE FORMAT DETAILS

14.2.1 Row Data

The row data are stored as an integer number of rows per disk record (512 bytes) or as an integer number of disk records per row. The columns are given a physical order appropriate to addressing on all computers. The logical order is carried in the file header record (physical record 1, see below) and in a set of array indices for addressing by the programs. The type of data is specified by code numbers. These codes and the physical ordering are as follows:

ORDER	ARRAY	BASIC CODE	+ LENGTH
double precision floating	R8	1	-
single precision floating	R4	2	-
character (4 / floating)	R4	3	+ 10 * 1
long integer	I4	4	-
logical	L2	5	-
integer	I2	6	-
bit (NBITWD / integer)	I2	7	+ 10 * 1
select flag	I2	9	-

Declarations:

```
      INTEGER*2  I2(*)
      INTEGER*4  I4(*)
      LOGICAL*2  L2(*)
      REAL*4     R4(*)
      REAL*8     R8(*)
      EQUIVALENCE (I2, I4, L2, R4, R8)
```

The ordering is chosen to allow some machines to preprocess the LOGICAL*2 statement into a LOGICAL*4 if needed. More esoteric preprocessing may be required on less standard machines.

14.2.2 Physical File Format

The data, control, and header information are written in the Table file via ZFIO in 512-byte (256-integer) blocks. The order on disk, by physical record number, is:

```
record   1      : Control info / lookup table (see later)
         2      : DATPTR(128) subscript of the appropriate array for
                        logical column n
                  DATYPE(128) type code for logical column n
         3 - 4  : Selection strings now in force
         5 - m  : Titles (6 R*4s, 4 chars/R*4) in physical column order
       m+1 - i  : Units (2 R*4s, 4 chars/R*4) in physical column order
       i+1 - k  : Table header (keyword/value pairs, see below)
       k+1 - *  : Row data in n rows/record or n records/row
where
       m =      5 +  NCOL / (256 / (6 * NWDPFP))
       i = m + 1 +  NCOL / (256 / (2 * NWDPFP))
       k = i + 1 +  NKEY / (256 / (4 * NWDPFP))
       NCOL = number logical columns not including the select column
       NKEY = maximum number of keyword/value pairs
```

14.2.3 Control Information

Physical record one contains file control data needed to do the I/O operations and maintain the physical file. It is prepared by subroutine TABINI and modified by TABIO. The latter subroutine returns the record to disk on OPCODE = 'CLOS'. Its contents are:

```
 1 -  2   (I*4) Number 512-byte records now in file
 3 -  4   (I*4) Max number rows allowed in current file
 5 -  6   (I*4) Number rows (logical records) now in file
 7              Number of bytes/value (2 for TA files)
 8              # values/logical (# I*2s/row incl. select for TA)
 9              > 0 => number rows / physical record
                < 0 => number physical records / row
10              Number logical columns/row (not including selection
                column)
11 - 16         Creation date: ZDATE(11), ZTIME(14)
17 - 28         Physical file name (set on each TABINI call)
29 - 31         Creation task name (2 chars / integer)
32              Disk number
```

```
 33 - 38          Last write access date: ZDATE(33), ZTIME(36)
 39 - 41          Last write access task name (2 chars / integer)
 42               Number logical records to extend file if needed
 43               Sort order: logical column # of primary sorting
 44               Sort order: logical column # of secondary sorting
                       0 => unknown, < 0 => descending order
 45               Disk record number for column data pointers (2)
 46               Disk record number for row selection strings (3)
 47               Disk record number for 1st record of titles (5)
 48               Disk record number for 1st record of units
 49               Disk record number for 1st record of keywords
 50               Disk record number for 1st record of table data
 51               DATPTR (row selection column)
 52               Maximum number of keyword/value pairs allowed
 53               Current number of keyword/value pairs in file
***********
 54 - 60          Reserved
***********
 61               Number of selection strings now in file
 62               Next available R*4 address for a selection string
 63               First R*4 address of selection string 1
 64               First R*4 address of selection string 2
 65               First R*4 address of selection string 3
 66               First R*4 address of selection string 4
 67               First R*4 address of selection string 5
 68               First R*4 address of selection string 6
 69               First R*4 address of selection string 7
 70               First R*4 address of selection string 8
********** for TABIO / TABINI use only **********
 71               IOP : 1 => read, 2 => writ
 72               Number I*2 words per logical record (incl. select)
 73 - 74   (I*4) Current table row physical record in BUFFER
 75 - 76   (I*4) Current table row logical record in BUFFER
 77               Type of current record in BUFFER (0 - 5)
 78               Current control physical record number in BUFFER
 79               Current control logical record number in BUFFER
 80               Type of current control record in BUFFER
 81               File logical unit number (LUN)
 82               FTAB pointer for open file (IND)
***********
 83 -100          Reserved
***********
101 -128          Table title (4 chars / real)
129 -256          lookup table as COLPTR(logical column) = phys column
```

14.2.4 Keyword/value Records

The keyword/value pairs are stored in 4 single precision floating locations, 256 / (4 * NWDPFP) per physical record. The keyword is an 8-character string stored as 4 characters per real. It is left justified and the first character must imply the data type used for the value. The value is stored left justified in the 3rd and 4th reals using as many integer words as needed (see table below).

The first character of the keyword must specify the type of the binary value as:

D	double precision floating point
F	single precision floating point
C	8-character string in 4 chars / real
J	long integer
L	logical
I	integer

In the call sequence to TABIO, the variable RECORD is an integer array used to convey the data to the I/O operations. For keyword/value pairs, RECORD is divided as follows:

RECORD(1)	1st 4 chars of the keyword
RECORD(1+NWDPFP)	2nd 4 chars of keyword
RECORD(1+2*NWDPFP)	value

where the value occupies the following number of integer words

type D	NWDPDP
F	NWDPFP
C	2 * NWDPFP
J	NWDPLI
L	NWDPLO
I	1

14.2.5 I/O Buffers

The call to TABINI specifies two buffers, one for I/O scratch and control and the other for the data pointers which will be used by the calling program to access the column data. The first, called BUFFER, is used as

BUFFER(1)-BUFFER(128)	control pointers
BUFFER(129)-BUFFER(256)	lookup table
BUFFER(257)-BUFFER(***)	current physical record(s) of table data where *** = 512 if there are >= 1 rows/rec, *** = (n+1) * 256 if there are n recs/row.

The call sequence of TABINI has an argument NBUF which gives the length of BUFFER. This is used solely to check that BUFFER is large enough to handle the present table file. BUFFER is also provided by the programmer to TABIO which will modify the control and data portions. The programmer should not modify BUFFER between the call to TABINI and the call to TABIO with OPCODE 'CLOS' except to insert a title for the table in words 101 - 128 or to correct the sort order information.

The second buffer, called TABP, is used by the non-I/O portions of the table package. TABP(1,1) - TABP(128,1) contains the subscript of the appropriate array for the logical columns. TABP(1,2) - TABP(128,2) contains the data type for each logical column. The programmer must fill in TABP(1,2) - TABP(NCOL,2) before calling TABINI when TABINI is to create the table extension file. TABINI will return a complete set of TABP under all circumstances.

14.3 SUBROUTINES

14.3.1 TABINI

```
SUBROUTINE TABINI (OPCODE, PTYP, VOL, CNO, VER, CATBLK, LUN,
*   NKEY, NREC, NCOL, DATP, NBUF, BUFFER, IERR)
```

TABINI creates/opens a table extension file. If a file is created, it is catalogued by a call to CATIO which saves the updated CATBLK.
Input:

OPCODE	R*4	'READ' only, 'WRIT' read or write
PTYPE	I*2	File physical type: 2 characters
VOL	I*2	Disk number
CNO	I*2	Primary file catalog number
VER	I*2	Version number: <= 0 highest on READ highest+1 on WRIT (i.e. create one) output: version number used
CATBLK	I*2(256)	Primary file catalog header record
LUN	I*2	Logical unit number to use
NKEY	I*2	Maximum number of keyword/value pairs input: used on create, checked on WRIT (<= recorded); output: actual
NREC	I*2	Number rows for create/extend input: used on WRIT only.
NCOL	I*2	Number of logical columns (not incl select) input: used on create, checked on WRIT (0 => any); output: actual
DATP	I*2(128,2)	DATPTR, DATYPE: DATYPE input on create, output actual for both
NBUF	I*2	Number I*2 words in BUFFER
BUFFER	I*2(*)	I/O buffer (* >= 512 as needed)
IERR	I*2	Error codes: 0 => OK, -1 => OK, new file created, 1 => bad input, 2 => cannot find/open, 3 => I/O error 4 => create error

14.3.2 TABIO

```
SUBROUTINE TABIO (OPCODE, IRCODE, IRNO, RECORD, BUFFER, IERR)
```

TABIO does random access I/O to Tables extension files. Mixed reads and writes are allowed if TABINI was called 'WRIT'. Writes are limited by the size of the structure (i.e. # columns for units and titles) or to the current maximum logical record plus one. Files opened for WRITe are updated and compressed on CLOS.

OPCODE	R*4	'READ', 'CLOS', 'WRIT' write with row selected 'FLAG' write with row de-selected
IRCODE	I*2	Type of record: 0 => table row 1 => DATPTR/DATYPE record 2 => data selection string 3 => titles

```
                              4 => units
                              5 => keyword/value pairs
  IRNO      I*4 (!)       Logical record number:
                              IRCODE = 0 => row number
                              IRCODE = 1 => ignored
                              IRCODE = 2 => string number
                              IRCODE = 3 => column number
                              IRCODE = 4 => column number
                              IRCODE = 5 => keyword number
  RECORD    I*2(*)        Appropriate data (input or output):
                              IRCODE = 0 => row
                              IRCODE = 1 => DATP
                              IRCODE = 2 => select string
                              IRCODE = 3 => column title
                              IRCODE = 4 => column units
                              IRCODE = 5 => keyword/value
  BUFFER    I*2(>=768)    I/O control,scratch buffer (in/out)
  IERR      I*2           Error code: 0 => ok
                              -1 => row read, but it is flagged
                              1 file not open, 2 input error
                              3 I/O error      4 logical EOF
                              5 error in file expansion
```

14.3.3 TABCOP

```
      SUBROUTINE TABCOP (TYPE, INVER, OUTVER, LUNOLD, LUNNEW, VOLOLD,
     *   VOLNEW, CNOOLD, CNONEW, CATNEW, BUFF1, BUFF2, IRET)
```

EXTCOP copies Table extension file(s). The output file must be a new extension - old ones cannot be rewritten. The output file must be opened WRIT in the catalog and will have its CATBLK updated on disk.

Inputs:

TYPE	I*2	Extension file type (e.g. 'CC','AN')
INVER	I*2	Version number to copy, 0 => copy all.
OUTVER	I*2	Version number on output file, if more than one copied (INVER=0) this will be the number of the first file. If OUTVER = 0, it will be taken as 1 higher than the previous highest version.
LUNOLD	I*2	LUN for old file
LUNNEW	I*2	LUN for new file
VOLOLD	I*2	Disk number for old file.
VOLNEW	I*2	Disk number for new file.
CNOOLD	I*2	Catalog slot number for old file
CNONEW	I*2	Catalog slot number for new file

In/out:

CATNEW(256)	I*2	Catalog header for new file.

Output:

BUFF1(256)	I*2	Work buffer
BUFF2(256)	I*2	Work buffer - will have CATBLK of old file
IRET	I*2	Return error code 0 => ok 1 => files the same, no copy. 2 => no input files exist

```
                                        3 => failed
                                        4 => no output files created.
                                        5 => failed to update CATNEW
```

14.3.4 GETCOL

```
      SUBROUTINE GETCOL (IRNO, ICOL, DATP, BUFFER, RTYPE, RESULT,
     *    SCRTCH, IERR)
```

GETCOL returns the value and value type found in an open table file at the specified logical column and row.

Inputs:	IRNO	I*4	Table row number: n.b. I*4
	ICOL	I*2	Table column number
	DATP	I*2(256)	Pointer array returned by TABINI
In/out:	BUFFER	I*2(*)	Control area set up by TABINI, used in TABIO
Output:	RTYPE	I*2	Type of column: 1 -> R*8, 2 -> R*4, 4 -> I*4, 5 -> L*?, 6 -> I*2 3+10*L -> character length L unpacked 7+10*L -> bit array length L packed
	RESULT	???	Value of column: use R*8, R*4, I*4, I*2 equivalenced arrays
	SCRTCH	I*2(*)	Scratch large enough to hold a row
	IERR	I*2	Error code: 0 => OK. -1 => OK, but row is flagged 1 file not open, 2 input error 3 I/O error, 4 read past EOF 5 bad data type

14.3.5 FNDCOL

```
      SUBROUTINE FNDCOL (NKEY, KEYS, LKEY, LORDER, BUFFER, KOLS, IERR)
```

FNDCOL is used with AIPS Table extension files. It locates the logical column number(s) which are titled with specified strings.

Inputs:	NKEY	I*2	Number columns to be found
	KEYS	R*4(LKEY,N)	Column titles to locate (4 chars/real)
	LKEY	I*2	Number R*4 words to check in each of KEYS (legal values 1 through 6)
	LORDER	L*2	T => logical order desired, else phys.
In/out:	BUFFER	I*2(>512)	TABINI/TABIO buffer/ header/ work area
Output:	KOLS	I*2(NKEY)	Logical column numbers: 0 => none, -1 => more than one (!)
	IERR	I*2	Error code: 0 => ok, 1 - 10 from ZFIO >10 = 10 + # of failed columns

14.3.6 CTINI

SUBROUTINE CTINI (LUN, NCOL, VOL, CNO, VER, CATBLK, BUF, IERR)

CTINI creates and/or opens for writing (and reading) a specified CT (components table) file.

Inputs:	LUN	I*2	Logical unit number to use
	VOL	I*2	Disk number
	CNO	I*2	Catalog number
In/out:	NCOL	I*2	Number of columns: 3 or 7 are allowed.
	VER	I*2	Input: desired version number 0 -> new Output: that used
	CATBLK	I*2(256)	File catalog header block
Output:	BUF	I*2(768)	First 512 words required for later calls to TABIO
	IERR	I*2	Error codes from TABINI or TABIO

14.4 USAGE

At this writing (22-July-1984), there are several experimental tasks which use Tables extension files. These tasks are all similar to existing tasks designed to use EXTINI / EXTIO on Clean Components extension files. For the moment, these tasks have funny names and read/write extension type CT (rather than CC). When they have been tested enough and when a format conversion program works, they will replace the existing tasks. These tasks are TACLN (version of APCLN), TAMX (version of MX), UVTUB (version of UVSUB), TSCAL (version of ASCAL), and PRTCT (replacement for PRTCC). There is also a new task called PRTAB which prints the Tables extensions in a very general fashion. It will be instructive to programmers to examine its coding.

Generalized FITS extensions, with application to Tables.

Ronald H. Harten
The Netherlands Foundation for Radio Astronomy
Dwingeloo, The Netherlands
Preben Grosbøl
European Southern Observatory
Munich, West Gernany
Keith P. Tritton
Rutherford Appleton Laboratory
Chilton, Didcot, The United Kingdom
Eric W. Greisen, Donald C. Wells
National Radio Astronomy Observatory
Charlottesville, USA

ABSTRACT

A general design for future extensions to the FITS tape format is proposed. The proposed design preserves compatibility with existing FITS tapes and software, including the "random groups" and other extensions of FITS, but its generalized design will permit a wide variety of new types of extensions files to be designed in the future. As an example of the application of the new design, specifications are presented for a proposed extension to transmit tables of astronomical data.

CAUTION: portions of this proposal might be augmented, rescinded, or revised before final standardization.

1. Introduction

The FITS tape format standard (Wells, Greisen, and Harten 1981, hereafter "Basic FITS") was developed to transfer regularly gridded astronomical image data between different locations. It has been implemented by most of the major observatories of the world and has been endorsed by working groups for software in both Europe and North America. The design of basic FITS included several provisions which were intended to permit the format to be extended in order to transmit new kinds of data structures. The "random-groups" extension of FITS (Greisen and Harten 1981) exploited these possibilities to produce a tape format design which is useful for transmitting data which is regularly gridded on some axes and irregularly gridded on other axes. It has been implemented by several radio observatories for the transfer of radio interferometer visibility measurements. This extension has been recommended by the North American working group for use by North American observatories. The FITS tape format was recommended (resolution C1) for use by all observatories by Commission 5 at the 1982 meeting of the IAU at Patras (*IAU Information Bulletin* **No. 49**, 1983). Note that the General Assembly of the IAU adopted (resolution R11) the recommendations of its commissions, including the FITS resolution.

The concept of utilizing a standard flexible format for the transfer of astronomical data has proved to be appealing and designers of software systems for astronomy want to be able to apply it to a variety of data and information structures. For example, during the past two years there has arisen the proposal that FITS design concepts be utilized in the formatting of catalogs of astronomical data, such as the star catalogs which are distributed by the astronomical data centers. Commission 5 of the IAU at the Patras meeting in 1982 appointed a committee to investigate this concept. During the same period of time an experimental extension of FITS was developed to transmit source position lists and calibration tables in association with image data. Early in 1983 it became apparent that these two efforts should be combined in order to specify a single format designed to transmit arbitrary tabular data.

Several FITS extension formats have been designed already and more are expected to be devised. It is possible that some of them might conflict with each other or with the random-groups format which has already been endorsed as a standard. This observation has led to the realization that there is a need for a general set of rules to govern the design of all future FITS extensions. During several months of discussions such a set of rules evolved and it is presented in this paper. This paper also presents the proposed tables format as an example of the application of the rules.

2. Terminology

To avoid possible confusion in terminology in this paper and in the future, we shall define a few terms which will be used in this and future articles. The term "record" refers to the basic 2880 byte unit or piece of information. A FITS file consists of an integral number of records and extensions always begin with a new record. A record corresponds to a logical record. The term "block" refers to the physical block size on tape. At present a block is 2880 bytes long (i.e., one record per block); however, at some future date when record blocking may be allowed, the block size could be some integral multiple of 2880 bytes. In previous FITS papers the terms record and block were used interchangeably. In this paper we will be referring to records. Questions of blocking factors, etc. will not be discussed.

3. Basic Philosophy

The most important rule for designing new extensions to FITS is that existing FITS tapes must remain valid. We are not permitted to alter the basic format in such a way as to make existing FITS tapes invalid or unreadable by standard FITS tape reading programs. This does not mean that the FITS format cannot evolve or change. To avoid this trap, FITS was deliberately designed to be capable of evolving. Two rules form the basis for all existing or proposed extensions.

- Any number of 2880-byte records may appear **after** the blocks which transmit the primary data matrix. These blocks have often been called "special records". The rule obliges all basic FITS reading programs to be prepared to skip over such blocks if they are not programmed to interpret them.
- FITS files may be written in which there is no data matrix, either because the number of axes is set to zero or because the product of the dimensions of the axes is zero.

Simply stated, all FITS extensions must appear after the main FITS header and its associated data array and each extension should begin at the start of a new 2880 byte record.

All existing tapes containing extensions to basic FITS conform to these rules of basic FITS and therefore are valid "FITS tapes", even though they contain data structures which are not simple binary matrices. The existing random-groups format is the best-known example of such an extension. The proposed new rules also conform and they systematize the format of the special records so that new types of extensions can be devised freely by implementors of FITS software.

The rules allow users to create new extensions with a high degree of protection from conflicts with extensions devised by other implementors, and without obsoleting either the basic FITS standard or the existing extensions.

The basic structure of a FITS tape is quite flexible. By adding new keywords and data axes, users can design a data structure to suit their needs. In the past, users have created entire customized header structures using the HISTORY and COMMENT keywords. These structures were valid in the FITS format, readable by other users and did not require the approval of a standards committee. The set of rules for extensions to FITS presented in this paper is designed to provide a framework in which users can create new data structures to suit their local needs, while still following the FITS standard.

Many users may wonder why a basic mechanism for FITS extensions is necessary. The answer to this question is twofold. First, it allows one to transfer data collections which are not images or single data matrices. Essentially one can create a new extension format for each new type of information. This structure is possible in spite of the fact that the original FITS format was created for a particular data organization. This allows us to keep adjusting to new types of data organization while still adhering to the same set of basic rules. The second reason is that extensions allow us to transfer collections of related groups of information in an organized manner, i.e. it is providing us with a simple relational data base capability. In this manner, tables, lists, etc., associated with a data matrix can be written on tape in a manner in which the relationship between the different pieces of information is implicitly established.

The only restriction that will have to be placed on the freedom to create new extensions is that there should be only one approved extension format for each type of data organization. It will be the function of each user who creates a new extension type to check with the standards committee to see if an extension already exists for that type of data organization and to propose one if it is really a new extension type. An important point to remember is that an extension is a basic format for presenting or describing information and its organization. The contents of an extension and the optional keywords used, etc. will depend on the particular application. Thus one can use a table extension for all types of tabular information, without having to define a new extension type. New extension types have to be created whenever the organization of the information is such that it cannot be handled by one of the existing extension types. With this restriction in mind, users should feel free to create new extension types when the need arises.

4. Guidelines and Rules for FITS Extensions

Before we can specify the details of the extensions to FITS, it is necessary to discuss the basic guidelines and requirements for extensions. These fall into two broad classes. The first concerns those requirements which maintain the compatability and flexibility of the existing FITS standard. The second contains those new features which are desirable to solve the problems for which the extensions are needed. A list of the guidelines and requirements is given below.

- Existing FITS tapes, including those with existing standard extensions, must be compatible with the new extension standard. FITS files which contain combinations of standard and new extensions must be allowed in order to facilitate the transition to the new design.
- The presence of a new extension in a FITS file should not affect the operation of a program which does not know about a particular type of extension.
- Only the binary and character coding conventions specified in the basic FITS standard should be used in FITS extensions. These are printable ASCII in headers, and 8-bit unsigned, and 16, and 32-bit twos-complement integers without "byte-swap" in data matrices. The new

tables extension records are regarded as a binary matrix in which the 8-bit "pixels" are printable ASCII codes.

- Extensions should have the same structure as the basic FITS file, a header plus information. The extension data structures should be self-defining and readable by both humans and machines. The same basic rules for creating FITS headers should apply to extension headers, i.e. they will contain a required subset of standard keywords, consist of ASCII text and may have any length. This will allow one to reuse the code which interprets the primary FITS header.
- A program scanning a tape should be able to locate the beginning of any extension and should be able to skip over the extension, i.e. to find the start of the next one. This requirement implies that the extension header must specify in some consistent and standardized manner the total number of data bits which are associated with it.
- It should be possible for extensions to FITS to support hierarchical structuring of various types of data entities. One needs to be able to transmit tables, etc., which are associated with the basic data matrix and to maintain the relationship between the tables and the data. The ability to specify structures more than one level deep is included in order to provide a framework for future developments.
- It must be possible to devise new types of extensions without prior approval. This implies that **keywords** in the primary FITS header may not be used to announce the existence of a particular type of extension, because these would need to be approved by the standards committees.
- It should be possible to append any number of extensions to a primary header and its associated data matrix.
- If there is more than one type of extension in a FITS file then the extensions may appear in any order.
- Anyone wishing to create a new extension format is free to do so, but should check with the FITS standards committee to insure that there is no conflict in the extension type naming and that the proposed format conforms to the rules for FITS extensions.
- Physical tape blocks should continue to be 2880 bytes (23040 bits) long, but no information in either the primary header or any extension header should ever explicitly refer to the physical block sizes of FITS tape blocks. This principle, which was followed in the design of both basic FITS and the random-groups format, will permit the standards committees to modify the physical blocking specifications of FITS in the future without being obliged to change the information content of existing FITS files which may be reblocked after such a change. The physical blocking specifications may eventually have to be changed as tape densities increase and transmission of data through networks becomes more common.

The above list places a set of minimum requirements on features which must be built into FITS to be able to handle extensions in a systematic manner. The primary requirements are the requirement for a mini-header at the beginning of each extension and the ability for a program to be able to identify the type of an extension or to skip over it even if it does not recognize the type.

5. An addition to the Main FITS Header

Tapes which conform to this standard are required to include the keyword EXTEND in their main header immediately after the last required keyword of the basic FITS specifications. The value field should be the logical value true (T) to signify that the file is written in conformance with the new extension standard. The presence of the keyword does **not** imply that any conforming

extension records are actually appended to the file but merely that they might be. Note that EXTEND=T may be used even if random-groups and prototype tables extension records are present so long as their order conforms to the rules specified in the next section. An example of a minimal FITS header with a data matrix is:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234...
SIMPLE  =                    T /
BITPIX  =                   16 /
NAXIS   =                    2 /
NAXIS1  =                  320 /
NAXIS2  =                  512 /
EXTEND  =                    T /
END
```

An example of a minimal FITS header **without** a data matrix is:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234...
SIMPLE  =                    T /
BITPIX  =                    8 /
NAXIS   =                    0 /
EXTEND  =                    T /
END
```

Note that the presence of EXTEND=T in a primary FITS header merely indicates that the file **may** have extensions records and that any special records will conform to the rules given below.

6. Structure of Files Including Extension Records

The solution to the problem of compatibility with previously existing extension formats is to specify three new rules:

- If NAXIS1=0 and the random groups keywords are present in the primary header then the random-groups data records (see Greisen and Harten 1981) are present and they come immediately after the primary header.
- Extension records of the new type must follow the primary matrix or random groups records. Each extension should begin with a FITS-like header which is described below. This header specifies the "type" for the extension and a length which is computed by the usual FITS rules. The header may be arbitrarily long and is terminated by END in the same manner as a primary FITS header. Following the extension header, the specified number of records will appear containing the extension information. The next extension follows immediately after the previous one. Each new extension header must begin with a new record. As many extensions may be included in a FITS file as are required.
- Records of any nonstandard extensions should appear at the end of the file. A reading program should be prepared to encounter such records in any position which would ordinarily contain the first record of a standard extension header and when it does it can assume that the remaining special records of the file are non-standard. The program should examine the first eight bytes of the first record of the putative extension header. If they are the string XTENSION, then the block is the beginning of a standard extension header. If they are not, then it is the first of the non-standard special records.

7. The Extension Header

Each extension will begin at the start of a new record (2880 bytes per block). It will contain a standard FITS header except that the first line of the header (normally SIMPLE=T) is replaced by

the new keyword **XTENSION='type'** in order to identify the type of the extension. The required FITS keywords **BITPIX**, **NAXIS**, and **NAXISnnn** are used to specify the dimensions of the binary data matrix of the extension data. Only printable ASCII codes, 8-bit unsigned integer, and 16- and 32-bit twos-complement, non-byte-swapped integers will be acceptable for data interchange. The new extension standard allows other values of **BITPIX** to be used for special purposes, but these are considered to be nonstandard usage.

In order to permit random-groups data structures to be written in the new extensions without the inelegant **NAXIS1=0** convention, which had to be adopted for the original random-groups format, we require that **all** extension headers must incorporate the **PCOUNT** and **GCOUNT** keywords. For simple matrices their values should be **PCOUNT=0** and **GCOUNT=1**. The number of bits which the extension data will occupy will be computed with the following formula:

NBITS = BITPIX * GCOUNT * (PCOUNT + NAXIS1 * NAXIS2 * ... * NAXISn)

If **NAXIS=0** then the **NAXISi** terms in the above formula are all zero. Note that this calculation may cause an integer overflow if it is performed with insufficient precision (I*2 rather than I*4, for example). The number of standard FITS records (2880 bytes, 23040 bits) which the data will occupy will be computed with this formula:

NRECORDS = INT ((NBITS + 23039) / 23040)

Please note that these calculations will apply to all extensions regardless of the type of data structure. This permits designers to utilize **BITPIX**, **GCOUNT**, **PCOUNT**, and the **NAXISn** in any way which seems appropriate to define their data structures, subject to the constraint that the number of bits computed by the formula above must be correct. The extension header will end with an **END** statement in the usual fashion.

The inclusion of **GCOUNT** and **PCOUNT** in the extensions allows users considerable flexibility in designing extensions for data which has a semi-regular structure, i.e. the information table or data has a regular size, but there are a few values which are associated with each sub-set of the information and these vary in value with each group. The power of this option is discussed in the first FITS extension paper by Greisen and Harten (1981).

Implementors should note that the extension mechanism should not be used to transmit a 3-dimensional matrix as a sequence of 2-dimensional matrices in separate extension records. Instead, the generalized tools of FITS, in this case the ability to transmit n-dimensional matrices in the basic FITS header and data matrix, should be used. The freedom provided in the new extension design does not remove from implementors the obligation to use good taste and standard conventions in their designs. Extensions are meant to be used for other kinds of data such as tables, lists, text files, etc. Implementors of FITS writing programs should always be aware of the limitations of recipients. The primary objective of the FITS standard remains the reliable, unambiguous transmission of data to recipients. Esoteric, complex data structures should be avoided as much as possible. The watchword of the implementor should be: keep it simple.

A typical extension header with no associated data records is:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234...
XTENSION= 'type    '           / the type of the extension
BITPIX  =                    8 /
NAXIS   =                    0 /
PCOUNT  =                    0 /
GCOUNT  =                    1 /

END
```

In the case shown above the extension information is contained solely inside the extension header itself. A typical extension header with associated data records is:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234...
XTENSION= 'type    '           / the type of the extension
BITPIX  =                    8 /
NAXIS   =                    1 /
NAXIS1  =                12345 / number of bytes in the data records
PCOUNT  =                    0 /
GCOUNT  =                    1 /

END
```

This second example would be accompanied by five data records

NRECORDS = INT ((8 * 12345 + 23039) / 23040)

containing an arbitrary stream of 8-bit unsigned integers 12345 bytes long. Such a one-dimensional matrix could be appropriate for transmitting a text file. It is expected that a format to transmit text files will probably be the next extension design to be considered.

The tables extension discussed below defines a table to be a two-dimensional matrix of 8-bit bytes which will be used to convey printable ASCII text.

8. Three New Optional Keywords

Three new optional keywords are defined for the extension standard:

- **EXTNAME='name'**
- **EXTVER=n**

  These two keywords are provided for use in the new extension headers to give unique names and version numbers to individual extensions. This means that a FITS file might contain, for example, three different tables extensions (XTENSION='TABLE'). The first might be called EXTNAME='BS83' with EXTVER=1, the second might also be EXTNAME='BS83' but with EXTVER=3, and the third might be EXTNAME='AGK3' with EXTVER=83. I.e., more than one extension of the same type and same name might occur and would be distinguished by unique version numbers, and version numbers need not start with one or be consecutive. If EXTVER is not specified a default value of one should be assumed by a reading program.

  The name can also be used to establish a relational type of data base in the same manner as sub-directory names in some file systems. In this case the relationship between the different extensions is established directly. Names such as "map1.cleancomp1" or "N1234.field2.starlist" can be used to establish easy to understand relationships between different extensions and even between extensions in different FITS files.

- **EXTLEVEL=n**

  This keyword specifies the level of the current extension header in a heirarchical structure of extension headers. The first level of extension headers has the value set to one. Any level-two headers are subordinate to the last previous level-one header, and any level-three headers are subordinate to the last previous level-two header, etc. This concept permits the transmission of arbitrary heirarchical data structures and file systems in FITS. We recommend that the initial implementations of the new extension design all utilize EXTLEVEL=1 exclusively until experimental trials have demonstrated feasibility of this concept. If EXTLEVEL is not specified, a default value of one should be assumed by a reading program. If the recipient data processing system is unable to represent the heirarchical structure and encounters an extension with EXTLEVEL greater than one, it should act as though EXTLEVEL is one.

9. Extension Data Records

The standard FITS philosophy is to keep headers and data in separate records. The new extension format adheres to this rule even though it wastes space (the unused bytes at the end of the last header record). Therefore the extension data begins in the first byte of the next record after the record containing the END of the extension header.

10. FITS for Catalogs and Tables

There are three main classes of potential applications which have stimulated the development of the proposed tables extension. First, programmers want to transfer standard catalogs or tables such as star or source catalogs with self-documenting column headings. The catalogs are typically in tabular form already and have well defined formats and layout. The second class of application includes the need to transfer observing information such as logs, calibration tables, intermediate tables, etc. which have a relation to observational data. The actual observations can easily be put into a FITS format; however, the amount of auxiliary information is too large to be included easily as comments and the programmer does not want to give up the tabular form of the information. The final application is the need to transmit tables of results extracted from observational data by data analysis software. For example, a number of programs exist which can automatically detect sources in digital images and write the computed parameters (position, flux, size, spectral index, polarization, etc.) into output files. If these files could be written to tape in a system independent form, astronomers would be able to transmit such tabular data to each other and could utilize software which is designed to manipulate, merge, and intercompare such tables. The extension to the FITS format discussed in the following sections is designed to enable all three of these classes of tabular data to be transferred easily from one computing system to another.

When one analyses the structure of catalogs or tables, one finds that they consist of a number of rows each with a fixed number of elements and a fixed format; however, the entries do not form a uniform array. What one needs is a means of describing and referring to the contents of each row in the catalog or table. This can be done in the FITS context by treating the table as an array of characters and then defining the location and format of each field within a row of the character array. While this solution requires that all catalogs and tables be stored in character format, this is actually desirable since the internal number formats of the different computers differ so widely and the information is easier to handle in character form. Also, most of the standard catalogs presently available in computer readable form are in character format. For these reasons, the tables extension is based on the conceptual model of a table, containing multiple columns of numbers and "words" with headings at the top of the columns, which is printed on paper using a printer. The printed page is thought of as a bf matrix of ASCII codes, and the tables extension is designed to transmit and document this matrix, with the headings being encoded in the extension header.

The tables FITS extension uses the standard FITS rules (see Wells, Greisen, Harten 1981). In addition it makes use of the new standard for generalized extensions to the basic FITS format, which has been discussed in the first part of this paper. The catalog is written in an extension to the main FITS header and is preceded by an extension header which describes the contents of the catalog. The basic concept is as follows. The catalog or table is stored as a large character array. Each row of the catalog or table has the same number of characters. Each row consists of a sequence of fields, and this sequence is the same for each row. The formats of the different fields need not be the same, but the format of a given field must be the same for all rows. Blanks are used to fill out unused space within and between fields. When printed out, the character

array should be easily readable and it is recommended that there be a blank between each field within a row. The number of characters in a row and the number of entries or rows in the table or catalog defines the size of the character array.

Each field in a row is described by a series of keywords which describe the name, format, character location within a row, length and units of the information. Using this information, a program could search via variable name, extract the appropriate characters and convert them to the desired format and units. Since each field in the table is separately defined and since the length of characters in each row is fixed, it is possible to transfer catalogs which contain a large amount of comment information which an automatic decoding program would skip over, yet which can be read by merely printing the entire row. This is especially useful for observation logs and catalogs where a fixed region for comments can be provided. The format is quite flexible allowing one to describe the contents of any standard table or catalog. The only price to be paid is that extra blank characters may be required to insure that each row contains the same number of characters. This is not serious since the format is primarily intended for the transfer of information rather than the storage of the information.

11. The Tables and Catalog Extension Header Format

The table is written in an extension to the basic FITS image, with **XTENSION='TABLE'**. In the case of many catalogs or tables, there will not be an image, only an extension. But even in these cases, the basic FITS header will still appear in order to preserve compatibility with the older format and to describe the basic characteristics of the FITS file. The extension begins with an extension header which will contain information about the size and contents of the table. This information is provided in the form of keywords, including some of the same keywords as those used in the main header.

A tables extension header begins at the first byte of a new block and will appear in the form shown below. The first eight keywords (**XTENSION** through **TFIELDS**) **must** appear, and in the order shown here. The keywords **TBCOLnnn** and **TFORMnnn** bf must appear somewhere in the header, up to the value of **TFIELDS=kkk**, in order to properly define the fields of the table. The other keywords, **EXTLEVEL**, **EXTNAME**, **EXTVER**, **TTYPEnnn**, **TUNITnnn**, **TSCALnnn**, **TZEROnnn**, and **TNULLnnn**, are all optional. If they are missing default values will be assumed by a reading program:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901...
XTENSION= 'TABLE   '           / Tells the type of extension
BITPIX  =                    8 / Printable ASCII codes (8 is required)
NAXIS   =                    2 / The table is a matrix (2 is required)
NAXIS1  =                 mmmm / Width of table in characters
NAXIS2  =                 nnnn / Number of entries in table (1 is legal)
PCOUNT  =                    0 / Random parameter count (default value)
GCOUNT  =                    1 / Group count (default value)
TFIELDS =                  kkk / Number of fields in each row
                               / (i.e. the number of separate pieces of
                               / information in a row, maximum value of
                               / nnn in TYPEnnn.)

EXTNAME = 'name    '           / The name of the table
EXTVER  =                   vv / Version number of table "name" (integer)
EXTLEVEL=                   hh / Heirarchical level (1 is recommended)

TBCOLnnn=                  ccc / Starting char. pos. of field nnn
TFORMnnn= 'qww.dd  '           / Fortran format of field nnn (I,A,E,D)
                               / (NOTE: ww is width of field nnn)
```

```
TTYPEnnn= 'type     '           / Type (heading) of field nnn
TUNITnnn= 'unit     '           / Physical units of field nnn
TSCALnnn=                sss.ss / Scale factor for field nnn
TZEROnnn=                zzz.zz / Zero point for field nnn
TNULLnnn= 'bbbbbbbb'            / Null (blank) value for field nnn
                                / (NOTE: exact match left-justified to)
                                / (the width is specified by TFORMnnn.)
          .
          .
END
```

The **END** card must appear and the remainder of the header record which contains **END** should be padded with ASCII blanks. In addition to the keywords shown above the extension header may contain additional keywords which describe the table, contain comments, etc. We now give a more extended discussion of the rules associated with the tables keywords.

- **TTYPEnnn = 'name' / The name of the nth field in a row.**
  (optional, but strongly recommended, default ' ')
- **TBCOLnnn = value / The beginning column of the field.**
  (required)
- **TFORMnnn = 'format' / A single value Fortran-77 format code.**
  (required) This may use bf only the Fortran formats Iww, Aww, Eww.dd, and Dww.dd (i.e., integers, characters, and real numbers). The E-format implies single precision (21 bit mantissa accuracy, 6 decimal digits) and the D-format implies double precision (53 bit mantissa accuracy, 16 decimal digits). Note that numbers coded in the F-format style are processed correctly in the E and D-formats and so we do not need the F-format, whereas we bf do need to distinguish the floating point precision. Once again: only I, A, E, and D formats are allowed. Formats such as 2I2 are not allowed; they should be I2 and I2 (separate fields) instead. A-format fields should be encoded as plain text, without being enclosed in string quotes.
- **TUNITnnn = 'unit' / The units of the variable.**
  (Default: ' ') e.g., 'K' for degrees Kelvin (see BUNIT in Wells *et al.* 1981)
- **TSCALnnn = value / Scale factor applicable to the value.**
  (Default: 1.0) Note that this keyword is not relevant for A-format fields.
- **TZEROnnn = value / Zero offset to be applied to the value.**
  (Default: 0.0) Note that this keyword is not relevant for A-format fields. The true value of field nnn is computed as:

  (value of field nnn in the table) * TSCALnnn + TZEROnnn
- **TNULLnnn = 'null string' / Character string to indicate a null field.**
  This allows the program to distinguish between a zero value and a nonexistant one. The string should be left justified and is implicitly blank filled to the width of the field (standard Fortran-77 convention). If **TNULLnnn** is not specified the reading program should not check field nnn for a null value. Programmers should consider what action their table reading programs should take when they encounter a value which is illegal. For example, suppose the value **'***'** is present in an I3 field but **TNULLnnn** has not been specified. Probably the reading program should report the error and default to supplying its internal null value.
- **AUTHOR = 'The name of the author or creator'**
- **REFERENC = 'The reference to the table or catalog'**

The default values are assumed if the keywords are not provided. The keywords **TBCOLnnn** and **TFORMnnn** are required for any fields which are to be defined in the table or catalog. If these keywords are not specified an automatic decoding routine cannot decode the table.

Note that the width of each field is specified by the width ww given in its format TFORMnnn. Field nnn begins in character position TBCOLnnn and includes ww characters. The sum of the ww widths is not required to equal the true width of the lines of the table, NAXIS1. There is no prohibition against overlapping fields, although we are unable to think of a useful example of such usage. Reading programs should report an error in cases where a field is specified to extend beyond the true width NAXIS1.

The format keyword, TFORMnnn, is an area where some degree of common sense must be used by the user. To keep things manageable and understandable each field must have a separate format (multiple formats such as 2I2 are not allowed). If a distinction between +0 and -0 is required (i.e. declination) then the sign field should be defined separately. This is absolutely necessary since many computers do not know the difference between +00 and -00. The sign should be defined as a character field and checked when decoding the associated number field. Thus the declination defined in degrees/minutes/seconds format would require four fields to be defined, each with its own TTYPEnnn, TFORMnnn, etc. But a declination defined as a floating point number in degrees would only require a single field and would conform to standard FITS rules.

It is recommended that the exponents of real numbers consist of a D or E followed by a sign and 2 numeric digits. Character data should be left justified, while integers and reals should be right justified to prevent the problem of how trailing blanks are treated in different computers. The fundamental rule is: the Fortran-77 specifications will apply (trailing blanks default to trailing zeroes!).

For those creating a new catalog or table format, it is recommended that there should be a blank between the different fields. A general rule should be that the character array containing the table should be easy to read in itself. This makes it possible to print out a number of rows of a table (using the header to determine the number of characters per row, etc.). Unfortunately, some existing catalogs do not have the fields separated by blanks. The FITS format is still valid and applicable for these catalogs; however, the simple printout option is less attractive.

When creating a table, one may need to distinguish between a 'null' (or undefined) value and a zero value. Normally, blanks in a numeric field will be interpreted as zeroes (standard Fortran-77 rules). In those fields where blanks should be considered to be nulls, the keyword TNULLnnn can be used to specify a 'null' value. Null values must be separately specified for all fields for which they are needed (if TNULLnnn is not defined for a field, then all values in that field are defined). Note also that the null value is a character string of the length ww which is specified by TFORMnnn. It is **not** required to be decodable by the format specified by TFORMnnn. For example, a null value of '***' might be used for an I3 field.

Note: the values of TTYPEnnn and TUNITnnn which are shown in the examples in this paper should be regarded as examples of possible values. The specification of possible values for these keywords is beyond the scope of this paper. We expect that the IAU FITS committee will produce a standard list of column headings and will recommend any units other than the standard SI units which are needed for existing catalogs.

12. Table Data Records

The data records are stored as a large character array, NAXIS1 characters across by NAXIS2 characters long and with NAXIS1 varying most rapidly, starting from the upper left corner of the table. All information is stored as 8-bit printable ASCII characters with the eighth bit (the

"parity" bit) set to zero (i.e., hexadecimal codes in the range 20 through 7E). Special characters with codes outside this range should be avoided since their meaning can be computer system dependent. No integer or real data values occur in the data array. Each data record is 2880 8-bit bytes long. The data records treat the character array as one large bit string. The data records are written one after the other and no attempt is made to prevent partial rows occurring in a record. If the user wishes to force the format to provide complete rows in a data record, then the number of characters per row must be chosen as to divide into 2880 evenly. The final record of the data should be padded with ASCII blanks.

13. An Example of the Tables Extension Format

This section contains an example of how one could put part of the AGK3 Star Cat. of Positions and Proper Motions, ed. W. Dieckvoss, into FITS format. Each row of the catalog contains sixteen items, which are described in sixteen fields. Two of the fields contain information in character format and the remaining fields contain numerical data. The FITS header describing the catalog and data records for three rows in the catalog are shown in the example below.

The formatting of the value fields in the example follows the rules of basic FITS. In particular, the required keywords obey the required fixed format. The optional keywords in this example also use a fixed format, and this is a recommended practice. Note that string values are always written with at least 8 characters, beginning in column 11.

The basic FITS header for this catalog would have the following form:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012...
SIMPLE  =                    T / Standard FITS format
BITPIX  =                    8 / character information
NAXIS   =                    0 / No image data array present
EXTEND  =                    T / There may be standard extensions
ORIGIN  = 'CDS     '           / Site which wrote the tape.
DATE    = '23/09/83'           / Date tape was written

COMMENT AGK3 Astrometric catalog, formatted in FITS Tables Format.
COMMENT see: W. Dieckvoss, Hamburg-Bergedorf 1975.
END
```

The extension header begins in a new block:

```
0.........1.........2.........3.........4.........5.........6.........7.....
12345678901234567890123456789012345678901234567890123456789012345678901234567890123456789012...
XTENSION= 'TABLE   '           / Table extension
BITPIX  =                    8 / 8-bits per "pixel"
NAXIS   =                    2 / simple 2-D matrix
NAXIS1  =                   74 / No. of characters per row (=74)
NAXIS2  =                    3 / The number of rows (=3)
PCOUNT  =                    0 / No "random" parameters
GCOUNT  =                    1 / Only one group.
TFIELDS =                   16 / there are 16 fields per row
EXTNAME = 'AGK3    '           / Name of the catalog

TTYPE1  = 'NO      '           / The star number
TBCOL1  =                    1 / start in column 1
TFORM1  = 'A7      '           / 7 character field

TTYPE2  = 'MG      '           / stellar magnitudes
TBCOL2  =                    8 / start in column 8
TFORM2  = 'E4.1    '           / xx.x SP floating point
TUNIT2  = 'MAG     '           / units are magnitudes

TTYPE3  = 'SP      '           / spectral type
```

```
TBCOL3  =                   13 / start in column 13
TFORM3  = 'A2      '           / 2 character field
TNULL3  = '        '           / blank is indefinite value

TTYPE4  = 'RAH     '           / right ascension hours
TBCOL4  =                   16 / start in column 16
TFORM4  = 'I2      '           / 2 digit integer
TUNIT4  = 'HR      '           / units are hours
TNULL4  = '99      '           / null value

TTYPE5  = 'RAM     '           / right ascension minutes
TBCOL5  =                   19 / start in column 19
TFORM5  = 'I2      '           / 2 digit integer
TUNIT5  = 'MIN     '           / minutes of time
TNULL5  = '99      '           / null value

TTYPE6  = 'RAS     '           / right ascension seconds
TBCOL6  =                   22 / start in column 22
TFORM6  = 'E6.3    '           / xx.xxx SP floating point
TUNIT6  = 'S       '           / seconds of time
TNULL6  = '99.999  '           / null value

TTYPE7  = 'DECDSIGN'           / declination sign
TBCOL7  =                   29 / start in column 29
TFORM7  = 'A1      '           / character field

TTYPE8  = 'DECD    '           / declination degrees
TBCOL8  =                   30 / start in column 30
TFORM8  = 'I2      '           / 2 digit integer
TUNIT8  = 'DEG     '           / degrees
TNULL8  = '99      '           / null value

TTYPE9  = 'DECM    '           / declination minutes
TBCOL9  =                   33 / start in column 33
TFORM9  = 'I2      '           / 2 digit integer
TUNIT9  = 'ARC.MIN '           / minutes (angle)
TNULL9  = '99      '           / null value

TTYPE10 = 'DECS    '           / declination seconds
TBCOL10 =                   36 / start in column 36
TFORM10 = 'E5.2    '           / xx.xx SP floating point
TUNIT10 = 'ARC.SEC '           / seconds (angle)
TNULL10 = '99.99   '           / null value

TTYPE11 = 'EP      '           / epoch of positions
TBCOL11 =                   42 / start in column 42
TFORM11 = 'E7.2    '           / xxxx.xx SP floating point
TUNIT11 = 'YR      '           / units are years

TTYPE12 = 'N       '           / no. photo. obs.
TBCOL12 =                   50 / start in column 50
TFORM12 = 'I1      '           / one digit integer

TTYPE13 = 'RA.PM   '           / proper motion in r.a.
TBCOL13 =                   52 / start in column 52
TFORM13 = 'E4.3    '           / .xxx SP floating point
TUNIT13 = 'ARC.SEC '           / units are arc-seconds/yr
TNULL13 = '9999    '           / null value

TTYPE14 = 'DEC.PM  '           / proper motion in dec.
TBCOL14 =                   57 / start in column 57
TFORM14 = 'E4.0    '           / xxx. SP floating point
TUNIT14 = 'ARC.SEC '           / units are arc-seconds/yr
TSCAL14 =                0.001 / scale factor = 0.001
                               / (Note use of scale factor!)
TNULL14 = '9999    '           / null value

TTYPE15 = 'DF(EP)  '           / difference in epoch AGK3-AGK2
TBCOL15 =                   62 / start in column 62
```

```
TFORM15 = 'E5.2     '           / xx.xx SP floating point
TUNIT15 = 'YR       '           / unit is years

TTYPE16 = 'BD       '           / Bonner Durch. star number
TBCOL16 =                    68 / start in column 68
TFORM16 = 'A7       '           / 7 character field
TNULL16 = '         '           / blanks indicate null

AUTHOR  = 'W. Dieckvoss'        /
REFERENC= 'Hamburg-Bergedorf 1975' /
DATE    = '14/07/82'            / date file was generated
END
```

The extension header shown above has 102 lines and therefore will be written in 3 logical records of 2880 bytes. (The third record will be padded with 6 blank lines.) The actual character data of the catalog would begin at the start of the next record. The three lines of 74 characters each (taken from page 46 of Dieckvoss 1975) will be in the first 222 bytes of the record.

```
0........1.........2.........3.........4.........5.........6.........7....
12345678901234567890123456789012345678901234567890123456789012345678901234
+82457 11.4 G5 15 30 57.480 +82 15 06.18 1960.37 2 -005 +006 29.99 +82 459
+82458 11.4 F5 15 32 41.150 +82 10 17.17 1958.36 2 -010 +004 27.97 +82 460
+82459 12.1    15 32 42.107 +82 40 28.83 1960.37 2 -018 +004 29.99 +82 461
```

Note that the spectral type field of the third line is blank, which is a null (see keyword **TNULL3** above). The remaining 2658 bytes of the record should contain ASCII blanks and a tapemark will follow. The FITS file will contain a total of five records: the basic header in the first record, then three extension header records, and finally one table data record.

14. Conclusions

The proposed extension to the FITS format provides an easy to use and convenient means of transfering catalog and tabular information between different computing facilities. The format treats the contents of the tables as a character array. The keywords define the different fields and provide information on the format, units and scale factors. By keying on the field names, **TTYPEnnn**, one can create automatic decoding routines which read and selectively decode the desired fields in the catalog while ignoring the remaining information. This is an excellent means of interfacing the information contained in catalogs with differing formats to standard reduction programs which would use the catalog information. For this to be completely successful it will be useful to agree on a set of standard field names and units for the contents of catalogs. This will allow users to be able to access automatically a wide range of astronomical data, without having to write a different program for each catalog. This point is being considered by I.A.U. Commission 5 which deals with documentation and astronomical data.

15. Acknowledgements

The authors would like to thank F. Ochsenbein and W. Warren for their detailed comments on early versions of this paper and on the general problem of encoding and distributing tables and catalogs.

References

Dieckvoss,W.: 1975, *AGK3 - Star Catalog of Positions and Proper Motions North of -2.5 Declination* **Vol. 1**, Hamburger Sternwarte, Hamburg.

Greisen,E.W.,Harten,R.H.: 1981, *Astron. Astrophys. Suppl.* **44**, 371

IAU Information Bulletin **No. 49**, 14, 1983

Wells,D.C.,Greisen,E.W.,Harten,R.H.: 1981, *Astron. Astrophys. Suppl.* **44**, 363