NOTICE: This document is provided for informational purposes to highlight the main proposed modifications to the existing *FITS* Standard document (Version 2.1b). Some minor typographical changes are not highlighted here. Readers should refer to the draft 3.0 version of the *FITS* Standard for the definitive set of proposed changes.

Definition of the Flexible Image Transport System (*FITS*)

FITS Standard

Version 3.0b (Third DRAFT)~~2.1b~~

2008 March 10 ~~December 9, 2005~~

~~IAU~~ *FITS* Working Group
Commission 5: Documentation and Astronomical Data
International Astronomical Union
http://fits.gsfc.nasa.gov/iaufwg/

Contents

Appendixes

List of Tables

~~Preface to Version 2.1b~~

~~This Version 2.1b of the *FITS* Standard is an interium release that incorporates 3 changes that were approved by the IAU *FITS* Working Group on 8 December 2005:~~

- ~~added support for 64-bit integer columns in binary tables, with TFORMn = 'K', and~~
- ~~added support for 64-bit descriptor columns in binary tables, with TFORMn = 'Q'.~~
- ~~added conditionally support for *FITS* primary arrays and image extension with 64-bit integer pixels (the final wording of this change is still under negotiation).~~

~~The other recent change in this version (and in the previous 2.1 version) is that the descriptions of the "Variable Length Array" facility and the "Multidimensional Array" convention have been moved from the unofficial Appendix B into section 8.3 of the *FITS* Standard itself. The IAU *FITS* Working Group officially approved these changes to the *FITS* Standard on 7 April, 2005.~~

~~Other than these changes, this version of the *FITS* Standard is very similar to the NOST 100-2.0 version that was developed in 1999, approved by the IAU *FITS* Working Group in October 2000, and published in the *Astronomy & Astrophysics* scientific journal in 2001 (volume 376, page 359).~~

~~Further information about the membership and role of the IAU FITS Working Group can be found on the `http://fits.gsfc.nasa.gov/iaufwg/` web site.~~

~~The NASA/Science Office of Standards and Technology (NOST) of the National Space Science Data Center (NSSDC) of the National Aeronautics and Space Administration (NASA) has been established to serve the space science communities in evolving cost effective, interoperable data systems. The NOST performs a number of functions designed to facilitate the recognition, development, adoption, and use of standards by the space science communities.~~

~~Approval of a NOST standard requires verification by the NOST that the following requirements have been met: consensus of the Technical Panel, proper adjudication of the comments received from the targeted space and Earth science community, and conformance to the accreditation process.~~

~~A NOST standard represents the consensus of the Technical Panel convened by the NOST. Consensus is established when the NOST Accreditation Panel determines that substantial agreement has been reached by the Technical Panel. However, consensus does not necessarily imply that all members were in full agreement with every item in the standard. NOST standards are not binding as published; however, they may serve as a basis for mandatory standards when adopted by NASA or other organizations.~~

~~A NOST standard may be revised at any time, depending on developments in the areas covered by the standard. Also, within five years from the date of its issuance, this standard will be reviewed by the NOST to determine whether it should 1) remain in effect without change, 2) be changed to reflect the impact of new technologies or new requirements, or 3) be retired or canceled.~~

~~The Technical Panel that developed this version of the standard consisted of the following members:~~

Robert J. Hanisch, Chair	Space Telescope Science Institute
William D. Pence, Secretary	NASA Goddard Space Flight Center
Barry M. Schlesinger, Past Secretary	Raytheon STX
Allen Farris	Space Telescope Science Institute
Eric W. Greisen	National Radio Astronomy Observatory
Peter J. Teuben	University of Maryland
Randall W. Thompson	Computer Sciences Corporation
Archibald Warnock	A/WWW Enterprises

Members of the previous Technical Panels also included:

Lee E. Brotzman	Hughes STX
Edward Kemper	Hughes STX
Michael E. Van Steenberg	NASA Goddard Space Flight Center
Wayne H. Warren Jr.	Hughes STX
Richard A. White	NASA Goddard Space Flight Center

~~This standard is published and maintained by the NOST. Send comments and orders for NOST documents to:~~

NOST, Code 633.2, NASA Goddard Space Flight Center
Greenbelt MD 20771
USA
electronic mail: nost@nssdca.gsfc.nasa.gov
+1-301-286-3575
http://ssdoo.gsfc.nasa.gov/nost/

~~Other information about *FITS* can be obtained from the *FITS* Support Office. The *FITS* Support Office can be contacted as follows:~~

FITS Support Office
Code 662, NASA Goddard Space Flight Center
Greenbelt MD 20771
USA
electronic mail: fits@fits.gsfc.nasa.gov
+1-301-286-4599
http://fits.gsfc.nasa.gov/

Section 1

Introduction

> *An* archival format *must be utterly portable and self-describing, on the assumption that, apart from the transcription device, neither the software nor the hardware that wrote the data will be available when the data are read.* 'Preserving Scientific Data on our Physical Universe,' p. 60. Steering Committee for the Study on the Long-Term Retention of Selected Scientific and Technical Records of the Federal Government, [US] National Research Council, National Academy Press 1995.

This document, hereafter referred to as the 'standard', describes the Flexible Image Transport System (*FITS*) which is the standard archival data format for astronomical data sets. Although *FITS* was originally designed for transporting image data on magnetic tape (which accounts for the 'I' and 'T' in the name), the capabilities of the *FITS* format have expanded to accommodate more complex data structures. The role of *FITS* has also grown from simply a way to transport data between different analysis software systems into the preferred format for data in astronomical archives, as well as the on-line analysis format used by many software packages.

This standard is intended as a formal codification of the *FITS* format which has been endorsed by the International Astronomical Union (IAU) for the interchange of astronomical data [1]. It is fully consistent with all actions and endorsements of the IAU *FITS* Working Group (IAUFWG) which was appointed by Commission 5 of the IAU to oversee further development of the *FITS* format. In particular, this standard defines the organization and content of the header and data units for all standard *FITS* data structures: the primary array, the random groups structure, the image extension, the ASCII table extension, and the binary table extension. It also specifies minimum structural requirements and general principles governing the creation of new extensions. For headers, it specifies the proper syntax for keyword records and defines required and reserved keywords. For data, it specifies character and numeric value representations and the ordering of contents within the byte stream.

One important feature of the *FITS* format is that its structure, down to the bit level, is completely specified in documents (such as this standard), many of which have been published in refereed scientific journals. Given these documents, which are readily available in hard copy form in libraries around the world as well as in electronic form on the Internet, future researchers should be able to decode the stream of bytes in any *FITS* format data file. In contrast, many other current data formats are only implicitly defined by the software that read and write the files. If that software is not continually maintained so that it can be run on future computer systems, then the information encoded in those data files could be lost.

1.1 Brief History of *FITS*

The *FITS* format evolved out of the recognition that a standard format was needed for transferring astronomical images from one research institution to another. The first prototype developments of a universal interchange format that would eventually lead to the definition of the *FITS* format began in 1976 between Don Wells at KPNO and Ron Harten at the Netherlands Foundation for Research in Astronomy (NFRA). This need for an image interchange format was raised at a meeting of the Astronomy section of the U.S. National Science Foundation in January 1979, which lead to the formation of a task force to work on the problem. Most of the technical details of the first basic *FITS* agreement (with files consisting of only a primary header followed by a data array) were subsequently developed by Don Wells and Eric Greisen (NRAO) in March 1979. After further refinements, and successful image interchange tests between observatories that used widely different types of computer systems, the first papers that defined the *FITS* format were published in 1981 [2, 3]. The *FITS* format quickly became the defacto standard for data interchange within the astronomical community (mostly on 9-track magnetic tape at that time) and was officially endorsed by the IAU in 1982 [1]. Most national and international astronomical projects and organizations subsequently adopted the *FITS* format for distribution and archiving of their scientific data products. Some of the highlights in the developmental history of *FITS* are shown in Table 1.1

~~The Flexible Image Transport System (*FITS*) evolved out of the recognition that a standard format was needed for transferring astronomical data from one installation to another. The original form, or Basic *FITS* [1], was designed for the transfer of images and consisted of a binary array, usually multidimensional, preceded by an ASCII text header with information describing the organization and contents of the array. The *FITS* concept was later expanded to accommodate more complex data formats. A new format for image transfer, *random groups*, was defined [2] in which the data would consist of a series of arrays, with each array accompanied by a set of associated parameters. These formats were formally endorsed [3] by the International~~

Table 1.1. Significant milestones in the development of *FITS*.

Date	Milestone
1979	Initial *FITS* Agreement and first interchange of files
1981	Published original (single HDU) definition [2]
1981	Published random groups definition [3]
1982	Formally endorsed by the IAU [1]
1988	Defined rules for multiple extensions [4]
1988	IAU *FITS* Working Group (IAUFWG) established
1988	Extended to include ASCII table extensions [5]
1988	Formal IAU approval of ASCII tables [6]
1990	Extended to include IEEE floating-point data [7]
1994	Extended to multiple `IMAGE` array extensions [8]
1995	Extended to binary table extensions [9]
1997	Adopted 4-digit year date format [10]
2002	Adopted conventions for celestial world coordinates [11, 12]
2004	Adopted MIME types for *FITS* data files [13]
2005	Extended to support variable-length arrays in binary tables
2005	Adopted conventions for spectral coordinate systems [14]
2005	Extended to include 64-bit integer data type

~~Astronomical Union (IAU) in 1982. Provisions for data structures other than simple arrays or groups were made later. These structures appear in *extensions*, each consisting of an ASCII header followed by the data whose organization it describes. A set of general rules governing such extensions [4] and a particular extension, ASCII table [5], were endorsed by the IAU General Assembly [6] in 1988. At the same General Assembly, an IAU *FITS* Working Group (IAUFWG) was formed [7] under IAU Commission 5 (Astronomical Data) with the mandate to maintain the existing *FITS* standards and to review, approve, and maintain future extensions to *FITS*, recommended practices for *FITS*, implementations, and the thesaurus of approved *FITS* keywords. In 1989, the IAUFWG approved a formal agreement [8] for the representation of floating point numbers. In 1994, the IAUFWG endorsed two additional extensions, the image extension [9] and the binary table extension [10]. *FITS* was originally designed and defined for 9-track half-inch magnetic tape. However, as improvements in technology have brought forward other data storage and data distribution media, it has generally been agreed that the *FITS* format is to be understood as a logical format and not defined in terms of the physical characteristics of any particular data storage medium. In 1994, the IAUFWG adopted a set of rules [11] governing the relation between the *FITS* logical record size and the physical block size for sequential media and bitstream devices. The IAUFWG also approved in 1997 an agreement [12] defining a new format for encoding the date and time in the `DATE`, `DATE-OBS`, and other related `DATExxxx` keywords to correct the ambiguity in the original DATE keyword format beginning in the year 2000.~~

Table 1.2. Version history of the standard.

Version	Date	Status
NOST 100-0.1	1990 December	1st Draft Standard
NOST 100-0.2	1991 June	2nd Revised Draft Standard
NOST 100-0.3	1991 December	3rd Revised Draft Standard
NOST 100-1.0	1993 June	NOST Standard
NOST 100-1.1	1995 September	NOST Standard
NOST 100-2.0	1999 March	NOST Standard
IAUFWG 2.1	2005 April	IAUFWG Standard
IAUFWG 2.1b	2005 December	IAUFWG Standard
IAUFWG 3.0	2007 July	IAUFWG Draft Standard

1.2 Version History of this Document

The fundamental definition of the *FITS* format was originally contained in a series of published papers [2, 3, 4, 5]. As *FITS* became more widely used, the need for a single document to unambiguously define the requirements of the *FITS* format became apparent. In 1990, the NASA Science Office of Standards and Technology (NOST) at the Goddard Space Flight Center provided funding for a technical panel to develop the first version of this standard document. As shown in Table 1.2, the NOST panel produced several draft versions, culminating in the first NOST standard document, NOST 100-1.0, in 1993. Although this document was developed under a NASA accreditation process, it was subsequently formally approved by the IAUFWG, which is the international control authority for the *FITS* format. The small update to the standard in 1995 (NOST 100-1.1) added a recommendation on the physical units of header keyword values.

The NOST technical panel was convened a second time to make further updates and clarifications to the standard, resulting in the NOST 100-2.0 version that was approved by the IAUFWG in 1999 and published in 2001 [15]. In 2005, the IAUFWG formally approved the variable-length array convention in binary tables, and a short time later approved support for the 64-bit integers data type. New versions of the standard were released to reflect both of these changes (versions IAUFWG 2.1 and IAUFWG 2.1b, respectively).

Most recently, the IAUFWG appointed its own technical panel in early 2007 to consider further modifications and updates to the standard. The changes proposed by this panel are shown in this draft, which is intended to eventually become version 3.0 of the *FITS* standard after it has been formally reviewed (and possibly further modified) by the IAUFWG.

The latest version of the standard, as well as other information about the *FITS*

format, can be obtained from the *FITS* Support Office web site at
`http://fits.gsfc.nasa.gov`.

1.3 Acknowledgments

The members of the 3 technical panels that produced this standard are shown below.

First technical panel, 1990 – 1993

Robert J. Hanisch (Chair)	Space Telescope Science Institute
Lee E. Brotzman	Hughes STX
Edward Kemper	Hughes STX
Barry M. Schlesinger	Raytheon STX
Peter J. Teuben	University of Maryland
Michael E. Van Steenberg	NASA Goddard Space Flight Center
Wayne H. Warren Jr.	Hughes STX
Richard A. White	NASA Goddard Space Flight Center

Second technical panel, 1994 – 1999

Robert J. Hanisch (Chair)	Space Telescope Science Institute
Allen Farris	Space Telescope Science Institute
Eric W. Greisen	National Radio Astronomy Observatory
William D. Pence	NASA Goddard Space Flight Center
Barry M. Schlesinger	Raytheon STX
Peter J. Teuben	University of Maryland
Randall W. Thompson	Computer Sciences Corporation
Archibald Warnock	A/WWW Enterprises

Third technical panel, 2007

William D. Pence (Chair)	NASA Goddard Space Flight Center
Lucio Chiappetti	IASF Milano, INAF, Italy
Clive G. Page	University of Leicester, UK
Richard Shaw	National Optical Astronomical Obs
Elizabeth Stobie	University of Arizona

Section 1

~~Overview~~

1.1 ~~Purpose~~

~~This standard formally defines the *FITS* format for data structuring and exchange that is to be used where applicable as defined in §1.3. It is intended as a formal codification of the *FITS* format that has been endorsed by the IAU for transfer of astronomical data, fully consistent with all actions and endorsements of the IAU and the IAU *FITS* Working Group (IAUFWG). Minor ambiguities and inconsistencies in *FITS* as described in the original papers are eliminated.~~

1.2 ~~Scope~~

~~This standard specifies the organization and content of *FITS* data sets, including the header and data, for all standard *FITS* formats: Basic *FITS*, the random groups structure, the ASCII table extension, the image extension, and the binary table extension. It also specifies minimum structural requirements for new extensions and general principles governing the creation of new extensions. It specifies the relation between physical block sizes and logical records for *FITS* files on bitstream devices and sequential media. For headers, it specifies the proper syntax for card images and defines required and reserved keywords. For data, it specifies character and value representations and the ordering of contents within the byte stream. It defines the general rules to which new extensions are required to conform.~~

1.3 ~~Applicability~~

~~This standard describes an extensible data interchange format particularly well suited for transport and archiving of arrays and tables of astronomical data. The IAU has recommended that all astronomical computer facilities support *FITS* for the interchange~~

~~of binary data. It has been NASA policy for its astrophysics projects to make their data available in *FITS* format. This standard may also be used to define the format for data transport in other disciplines, as may be determined by the appropriate authorities.~~

1.4 ~~Organization of This Document~~

~~§2 is a glossary of definitions, acronyms, and symbols. In §3, this document describes the overall organization of a *FITS* file, the contents of the first (primary) header and data, the rules for creating new *FITS* extensions, and the relation between physical block sizes and logical records for *FITS* files on bitstream devices and sequential media. The next two sections provide additional details on the header and data, with a particular focus on the primary header. §4 provides details about header card image syntax and specifies those keywords required and reserved in a primary header. §5 describes how different data types are represented in *FITS*. The following sections describe the headers and data of two standard *FITS* structures, the now deprecated random groups records (§6) and the current standard extensions: ASCII table, image, and binary table (§7). Throughout the document, deprecation of structures or syntax is noted where relevant. Files containing deprecated features are valid *FITS*, but these features should not be used in new files; the old files using them remain standard because of the principle that no change in *FITS* shall cause a valid *FITS* file to become invalid.~~

~~The Appendixes contain material that is not part of the standard. The first, Appendix A, provides a formal expression of the keyword/value syntax for header card images described in §4.2. Appendix **??** describes a convention for handling arrays of substrings in binary table extensions. Appendix **??** describes aspects of the implementation of *FITS* on physical media not covered by the blocking agreement. Appendix B is the appendix to the agreement endorsed by the IAUFWG for a new format for keywords expressing dates. The new format uses a four-digit value for the year, and thus eliminates any ambiguity in dates from the year 2000 and after. This appendix is not part of the formal agreement. It contains a detailed discussion of time systems. It has been slightly reformatted for stylistic compatibility with the remainder of this document. Appendix **??** lists the differences between this standard and the specifications of prior publications; it also identifies those ambiguities in the documents endorsed by the IAU on which this standard provides specific rules. The next four appendixes provide reference information: a tabular summary of the *FITS* keywords (Appendix C), a list of the ASCII character set and a subset designated *ASCII text* (Appendix D), a description of the IEEE floating point format (Appendix E), and a list of the extension type names that have been reserved as of the date this document was issued (Appendix **??**). Appendix **??** is a list of NOST documents, including earlier versions of this standard.~~

Section 2

Definitions, Acronyms, and Symbols

2.1 Conventions used in this document

Terms or letters set in `Courier` font represent literal strings that appear in *FITS* files. In the case of keyword names, such as '`NAXISn`', the lower case letter represents a positive integer index number, generally in the the range 1 to 999. The emphasized words *must*, *shall*, *should*, *may*, *recommended*, and *optional* in this document are to be interpreted as described in IETF standard, RFC 2119 [16].

2.2 Defined Terms

␣ Used to designate an ASCII space character~~blank~~.

ANSI American National Standards Institute.

Array A sequence of data values. This sequence corresponds to the elements in a rectilinear, n-dimension matrix ($1 \leq n \leq 999$, or $n = 0$ in the case of a null array~~$0 \leq n \leq 999$~~).

Array value The value of an element of an array in a *FITS* file, without the application of the associated linear transformation to derive the physical value.

ASCII American National Standard Code for Information Interchange.

ASCII character Any member of the 7-bit ASCII character set.

ASCII digit One of the 10 ASCII characters '0' through '9' which are represented by decimal character codes 48 through 57 (hexadecimal 30 through 39).

ASCII NULL ~~Hexadecimal 00.~~ The ASCII character that has all 8 bits set to zero.

ASCII space The ASCII character for space ~~blank~~ which is represented by ~~hexadecimal 20 (~~decimal 32 (hexadecimal 20).

ASCII text The restricted set of ASCII characters decimal 32 through 126 (hexadecimal 20 through 7E).

Basic FITS The *FITS* structure consisting of the primary header followed by a single primary data array. This is also known as Single Image *FITS* (SIF), as opposed to Multi-Extension *FITS* (MEF) files that contain one or more extensions following the primary HDU.

Big endian The numerical data format used in *FITS* files in which the most significant byte of the value is stored first followed by the remaining bytes in order of significance.

Bit A single binary digit.

Byte An ordered sequence of eight consecutive bits treated as a single entity.

Card image An obsolete term for an 80-character keyword record derived from the 80 column punched computer cards that were prevalent in the 1960s and 1970s. ~~A sequence of 80 bytes containing ASCII text, treated as a logical entity.~~

Character string A sequence of 1 or more of the restricted set of ASCII text characters, decimal 32 through 126 (hexadecimal 20 through 7E).

Conforming extension An extension whose keywords and organization adhere to the requirements for conforming extensions defined in §3.4.1 of this standard.

Data block A 2880-byte *FITS* block containing data described by the keywords in the associated header of that HDU.

~~DAT — 4mm Digital Audio Tape.~~

Deprecated To express disapproval of. This term is used to refer to obsolete structures that *should not* be used ~~for new applications~~ in new FITS files but *shall* remain valid indefinitely.

Entry A single value in an ASCII table or binary table standard extension~~a table~~.

~~Extension name — The identifier used to distinguish a particular extension HDU from others of the same type, appearing as the value of the `EXTNAME` keyword.~~

Extension A *FITS* HDU appearing after the primary HDU in a *FITS* file.

Extension type name The value of the `XTENSION` keyword, used to identify the type of the extension~~An extension format~~.

Field A component of a larger entity, such as a keyword record or a row of an ASCII table or binary table standard extension. A field in a table extension row consists of a set of zero or more table entries collectively described by a single format.

File A sequence of one or more records terminated by an end-of-file indicator appropriate to the medium.

FITS Flexible Image Transport System.

FITS block A sequence of 2880 8-bit bytes aligned on 2880 byte boundaries in the *FITS* file, most commonly either a header block or a data block. Special records are another infrequently used type of *FITS* block. This block length was chosen because it is evenly divisible by the byte and word lengths of all known computer systems at the time *FITS* was developed in 1979.

FITS file A file with a format that conforms to the specifications in this document.

FITS structure One of the components of a *FITS* file: the primary HDU, the random groups records, an extension, or, collectively, the special records following the last extension.

FITS Support Office The *FITS* information web site that is maintained by the IAUFWG and is currently hosted at `http://fits.gsfc.nasa.gov`.

Floating-point A computer representation of a real number.

Fraction The field of the mantissa (or significand) of a floating-point number that lies to the right of its implied binary point.

Group parameter value The value of one of the parameters preceding a group in the random groups structure, without the application of the associated linear transformation.

~~GSFC – Goddard Space Flight Center.~~

HDU Header and Data Unit. A data structure consisting of a header and the data the header describes. Note that an HDU *may* consist entirely of a header with no data blocks~~records~~.

Header A series of keyword records ~~card images~~ organized within one or more header blocks ~~*FITS* logical records~~ that describes structures and/or data which follow it in the *FITS* file.

Header block A 2880-byte *FITS* block containing a sequence of thirty-six 80-character keyword records.

Heap The supplemental data area following the main data table in a binary table standard ~~A supplemental data area, currently defined to follow the table in a binary table~~ extension.

IAU International Astronomical Union.

IAUFWG International Astronomical Union *FITS* Working Group.

~~IUE – International Ultraviolet Explorer.~~

IEEE Institute of Electrical and Electronic Engineers.

IEEE NaN IEEE Not-a-Number value; used to represent undefined floating-point values in *FITS* arrays and binary tables.

IEEE special values Floating-point number byte patterns that have a special, reserved meaning, such as -0, $\pm\infty$, $\pm$underflow, $\pm$overflow, $\pm$denormalized, $\pm$ NaN. (See Appendix E).

Indexed keyword A keyword name that is of the form of a fixed root with an appended positive integer ~~count.~~ index number.

Keyword name The first eight bytes of a ~~header card image.~~ keyword record which contain the ASCII name of a metadata quantity (unless it is blank).

Keyword record An 80-character record in a header block consisting of a keyword name in the first 8 characters followed by an *optional* value indicator, value and comment string. The keyword record *shall* be composed only of the restricted set of ASCII text characters ranging from decimal 32 to 126 (hexadecimal 20 to 7E).

~~Logical record – A record comprising 2880 8-bit bytes~~.

Mandatory keyword A keyword that *must* be used in all *FITS* files or a keyword required in conjunction with particular *FITS* structures.

Mantissa Also known as significand. The component of an IEEE floating-point number consisting of an explicit or implicit leading bit to the left of its implied binary point and a fraction field to the right.

MEF Multi-Extension *FITS*, i.e., a *FITS* file containing a primary HDU followed by one or more extension HDUs

~~Matrix – A data array of two or more dimensions~~.

NOST NASA/Science Office of Standards and Technology.

Physical value The value in physical units represented by an element of an array and possibly derived from the array value using the associated, but *optional*, linear transformation.

Pixel Short for 'Picture element'; a single location within an array~~Picture element~~.

Primary data array The data array contained in the primary HDU.

Primary HDU The first HDU in a *FITS* file.

Primary header The first header in a *FITS* file, containing information on the overall contents of the file (as well as on the primary data array, if present).

Random Group A *FITS* structure consisting of a collection of 'groups', where a group consists of a subarray of data and a set of associated parameter values. Random groups are deprecated for any use other than for radio interferometry data.

Record A sequence of bits treated as a single logical entity.

~~Reference point – The point along a given coordinate axis, given in units of pixel number, at which a value and increment are defined.~~

Repeat count The number of values represented in a field in a binary table standard extension~~field~~.

Reserved keyword An *optional* keyword that *must*~~*may*~~ be used only in the manner defined in this standard.

SIF Single Image *FITS*, i.e., a *FITS* file containing only a primary HDU, without any extension HDUs. Also known as Basic *FITS*.

Special records A series of one or more *FITS* blocks following the last HDU ~~A series of 23040-bit (2880 8-bit byte) records, following the primary HDU,~~ whose internal structure does not otherwise conform to that for the primary HDU or to that specified for a conforming extension in this standard. Any use of special records requires approval from the IAU FITS Working Group.

Standard extension A conforming extension whose header and data content are completely specified in §7 of this standard, namely, an image extension, an ASCII table extension, or a binary table extension. ~~specified explicitly in this standard.~~

~~Type name – The value of the XTENSION keyword, used to identify the type of the extension in the data following.~~

~~Valid value – A member of a data array or table corresponding to an actual physical quantity.~~

Section 3

FITS File Organization

3.1 Overall File Structure

A *FITS* file *shall* be composed of the following *FITS* structures, in the order listed:

- Primary header and data unit (HDU)
- Conforming Extensions (*optional*)
- Other special records (*optional*, restricted)

A *FITS* file composed of only the primary HDU is sometimes referred to as a Basic *FITS* file, or a Single Image *FITS* (SIF) file, and a *FITS* file containing one or more extensions following the primary HDU is sometimes referred to as a Multi-Extension *FITS* (MEF) file.

Each *FITS* structure *shall* consist of an integral number of *FITS* blocks which are each 2880 bytes (23040 bits) in length~~logical records~~. The primary HDU *shall* start with the first *FITS* block ~~record~~ of the *FITS* file. The first *FITS* block ~~record~~ of each subsequent *FITS* structure *shall* be the *FITS* block ~~record~~ immediately following the last *FITS* block ~~record~~ of the preceding *FITS* structure. ~~The size of a FITS logical record shall be 23040 bits, corresponding to 2880 8-bit bytes~~.

This standard does not impose a limit on the total size of a *FITS* file, nor on the size of an individual HDU within a *FITS* file. Software packages that read or write data according to this standard could be limited, however, in the size of files that are supported. In particular, some software systems have historically only supported files up to 2^{31} bytes in size (approximately 2.1 GB).

3.2 Individual *FITS* Structures

The primary HDU and every extension HDU *shall* consist of 1 or more 2880-byte header blocks immediately followed by an *optional* sequence of associated 2880-byte data blocks.~~an integral number of header records consisting of ASCII text, which *may* be followed by an integral number of data records. The first record of data *shall* be the record immediately following the last record of the header.~~. The header blocks *shall* contain only the restricted set of ASCII text characters, decimal 32 through 126 (hexadecimal 20 through 7E). The ASCII control characters with decimal values less than 32 (including the null, tab, carriage return, and line feed characters), and the delete character (decimal 127 or hexadecimal 7F) *must* not appear anywhere within a header block

3.3 Primary Header and Data Unit~~Array~~

The first component of a *FITS* file *shall* be the primary HDU which always contains the primary header and *may* be followed by the ~~header. The primary header may, but need not be , followed by a~~ primary data array. If the primary data array has zero length, as determined ~~The presence or absence of a primary data array *shall* be indicated~~ by the values of the `NAXIS` and~~or~~ `NAXISn` keywords in the primary header (§4.4.1.1), then the primary HDU *shall* contain no data blocks.

3.3.1 Primary Header

The header of a primary HDU *shall* consist of one or more header blocks, each containing a series of 80-character keyword records containing only the restricted set of ~~a series of card images in~~ ASCII text characters. Each 2880-byte header block contains 36 keyword records. The last header block *must* contain the `END` keyword (defined in §4.4.1.1) which marks the logical end of the header. Keyword records without information (e.g., following the `END` keyword) ~~. All header records *shall* consist of 36 card images. Card images without information~~ *shall* be filled with ASCII ~~blanks~~ spaces (decimal 32 or hexadecimal 20).

3.3.2 Primary Data Array

~~In *FITS* format, the~~ The primary data array, if present, *shall* consist of a single data array with from 1 ~~of 0~~ to 999 dimensions (as specified by the `NAXIS` keyword defined in §4.4.1.1). The random groups convention in the primary data array is a more complicated structure ~~(see §6).~~and is discussed separately in §6. The entire array of data values are represented by a continuous stream of bits starting with the first bit of the first data block. Each data value *shall* consist of a fixed number of bits that is determined by the value of the `BITPIX` keyword (§4.4.1.1).~~The data values *shall* be a byte stream with no embedded fill or blank space. The first value *shall* be in the first position of the first primary data array record. The first value of each subsequent row of the array *shall* be in the position immediately following the last value of the previous row~~. Arrays of more

A(1, 1, . . . , 1),
A(2, 1, . . . , 1),
⋮,
A(`NAXIS1`, 1, . . . , 1),
A(1, 2, . . . , 1),
A(2, 2, . . . , 1),
⋮,
A(`NAXIS1`, 2, . . . , 1),
⋮,
A(1, `NAXIS2`, . . . , `NAXISm`),
⋮,
A(`NAXIS1`, `NAXIS2`, . . . , `NAXISm`)

Figure 3.1 Arrays of more than one dimension *shall* consist of a sequence such that the index along axis 1 varies most rapidly and those along subsequent axes progressively less rapidly. ~~Except for the location of the first element, array structure is independent of record structure.~~

than one dimension *shall* consist of a sequence such that the index along axis 1 varies most rapidly, that along axis 2 next most rapidly, and those along subsequent axes progressively less rapidly, with that along axis `m`, where `m` is the value of `NAXIS`, varying least rapidly. ~~; i. e. , the elements of an array $A(x_1, x_2, \ldots, x_m)$ shall be in the order shown in Figure 3.1.~~ There is no space or any other special character between the last value on a row or plane and the first value on the next row or plane of a multi-dimensional array. Except for the location of the first element, the array structure is independent of the *FITS* block structure. This storage order is shown schematically in Figure 3.1 and is the same order as in multi-dimensional arrays in the Fortran programming language [17]. The index count along each axis *shall* begin with 1 and increment by 1 up to the value of the `NAXISn` keyword (§4.4.1.1).

If the data array does not fill the final data block ~~record~~, the remainder of the data block ~~record~~ *shall* be filled by setting all bits to zero. The individual data values *shall* be stored in big-endian byte order such that the byte containing the most significant bits of the value appears first in the *FITS* file, followed by the remaining bytes, if any, in decreasing order of significance.

3.4 Extensions

3.4.1 Requirements for Conforming Extensions

All extensions, whether or not further described in this standard, *shall* fulfill the following requirements to be in conformance with this *FITS* standard. New extension types *should* be created only when the organization of the information is such that it cannot be handled by one of the existing extension types. A *FITS* file that contains extensions is commonly referred to as a multi-extension *FITS* (MEF) file.

3.4.1.1 Identity

Each extension type *shall* have a unique type name, specified in the header by the `XTENSION` keyword (§4.4.1.2)~~according to the syntax codified in §4.4.1.2~~. To preclude conflict, extension type names *must* be registered with the IAUFWG. The current list of registered extensions is given in Appendix F. An up to date list is also maintained on the *FITS* Support Office web site~~shall maintain and provide a list of the registered extensions~~.

3.4.1.2 Size Specification

The total number of bits in the data of each extension *shall* be specified in the header for that extension, in the manner prescribed in §4.4.1.2.

3.4.1.3 ~~Compatibility with Existing *FITS* Files~~

~~No extension shall be constructed that invalidates existing *FITS* files.~~

3.4.2 Standard Extensions

A standard extension is ~~shall be~~ a conforming extension whose organization and content are completely specified in §7 of this standard. Only one extension ~~*FITS*~~ format *shall* be approved for each type of data organization. ~~Each standard extension shall have a unique name given by the value of the `XTENSION` keyword (see Appendix F).~~

3.4.3 Order of Extensions

An extension *may* follow the primary HDU or another conforming extension. Standard extensions and other conforming extensions *may* appear in any order in a *FITS* file.

3.5 Special Records (Restriced Use)

Special records are 2880-byte *FITS* blocks following the last HDU of the *FITS* file that have an unspecified structure that does not meet the requirements of a conforming extension. The first 8 bytes of the special records *must not* contain the string 'XTENSION'. It is *recommended* that they do not contain the string 'SIMPLE␣␣'. ~~The records must have the standard *FITS* 23040-bit record length.~~ The contents of special records are not otherwise specified by this standard.

Special records were originally designed as a way for the *FITS* format to evolve by allowing new *FITS* structures to be implemented. Following the development of conforming extensions, which provide a general mechanism for storing different types of data structures in *FITS* format in a well defined manner, the need for other new types of *FITS* data structures has been greatly reduced. Consequently, further use of special records is restricted and requires the approval of the IAU FITS Working Group.

3.6 Physical Blocking

3.6.1 Bitstream Devices

For bitstream devices, including but not restricted to logical file systems, *FITS* files ~~shall be written with fixed blocks of a physical block size equal to the 23040-bit FITS logical record size.~~ *shall* be interpreted as a sequence of 1 or more 2880-byte *FITS* blocks, regardless of the physical blocking structure of the underlying recording media. When writing a *FITS* file on media with a physical block size unequal to the 2880-byte *FITS* block length, any bytes remaining in the last physical block following the end of the *FITS* file *should* be set to zero. Similarly, when reading *FITS* files on such media, any bytes remaining in the last physical block following the end of the *FITS* file *shall* be disregarded.

3.6.2 Sequential Media

The *FITS* format was originally developed for writing files on sequential magnetic tape devices. The following rules on how to write to sequential media [18] are now irrelevant to most current data storage devices.

If physically possible, *FITS* files *shall* be written on sequential media in blocks that are from 1 to 10 integer multiples of 2880-bytes in length. If this is not possible, the *FITS* file *shall* be written as a bitstream using the native block size of the sequential device. Any bytes remaining in the last block following the end of the *FITS* file *shall* be set to zero.

When reading *FITS* files on sequential media, any files shorter than 2880 bytes in length (e.g., ANSI tape labels) are not considered part of the *FITS* files and *should* be disregarded.

~~3.6.2.1 Fixed Block~~

~~For fixed block length sequential media, including but not restricted to optical disks (accessed as a sequential set of records), QIC format 1/4 inch cartridge tapes, and local area networks, FITS files shall be written as a bitstream, using the fixed block size of the medium. If the end of the last logical record does not coincide with the end of a physical fixed block, all bits in the remainder of the physical block containing the last logical record shall be set to zero.After an end-of-file mark has been detected in the course of reading a FITS file, subsequent incomplete FITS logical records should be disregarded.~~

~~3.6.2.2 Variable Block~~

~~For variable block length sequential media, including but not restricted to 1/2 inch 9-track tapes, DAT 4 mm cartridge tapes, and 8 mm cartridge tapes, *FITS* files may be written with an integer blocking factor equal to 1 to 10 logical records per physical block.~~

3.7 Restrictions on Changes

Any structure that is a valid *FITS* structure *shall* remain a valid *FITS* structure at all future times. Use of certain valid *FITS* structures *may* be deprecated by this or future *FITS* standard documents.

Section 4

Headers

The first 2 sections of this chapter define the structure and content of header keyword records. This is followed in §4.3 with recommendations on how physical units should be expressed. The final section defines the mandatory and reserved keywords for primary arrays and conforming extensions.

4.1 ~~Card Images~~Keyword Records

4.1.1 Syntax

Each 80-character header keyword record ~~Header card images~~*shall* consist of a keyword name, a value indicator (only required if ~~*optional* unless~~ a value is present), an ~~a value (~~*optional* value, and an ~~), and a comment (~~*optional* comment~~)~~. Except where specifically stated otherwise in this standard, keywords *may* appear in any order. It is *recommended* that the order of the keywords in *FITS* files be preserved during data processing operations because the designers of the *FITS* file may have used conventions that attach particular significance to the order of certain keywords (e.g., by grouping sequences of `COMMENT` keywords at specific locations in the header, or appending `HISTORY` keywords in chronological order of the data processing steps).

A formal syntax, giving a complete definition of the syntax of *FITS* keyword records ~~card images~~, is given in Appendix A. It is intended as an aid in interpreting the text defining the standard.

4.1.2 Components

4.1.2.1 Keyword name (bytes 1 through 8)

The keyword name *shall* be a left justified, 8-character, space-filled~~blank-filled~~, ASCII string with no embedded spaces ~~blanks~~. All digits 0 through 9 (decimal ASCII codes 48 to 57, or (hexadecimal 30 to 39~~, "0123456789"~~) and upper case Latin alphabetic characters ~~(hexadecimal 41 to 5A, "`ABCDEFG HIJKLMN OPQRST UVWXYZ` ")~~ '`A`' through '`Z`' (decimal 65 to 90 or hexadecimal 41 to 5A) are permitted; ~~no~~ lower case characters *shall not* be used. The underscore ('`_`', decimal 95 or hexadecimal 5F) and hyphen ('`-`', decimal 45 or hexadecimal 2D) are also permitted. No other characters are permitted. For indexed keyword names that have a single positive integer index counter appended to the root name, the ~~keywords with a single index the~~ counter *shall not* have leading zeroes (e.g.,`NAXIS1`, not `NAXIS001`) .

4.1.2.2 Value Indicator (bytes 9 and 10)

If the 2 ~~this field contains the~~ ASCII characters '`=␣`' (decimal 61 followed by decimal 32) are present in bytes 9 and 10 of the keyword record this indicates that the keyword has ~~, it indicates the presence of~~ a value field associated with it, unless it is one of the commentary keywords defined in §4.4.2.4 (i.e., a `HISTORY`, COMMENT, or completely blank keyword name) which by definition have no value.~~the keyword, unless it is a commentary keyword as defined in §4.4.2.4~~. ~~If the value indicator is not present or if it is a commentary keyword then columns 9–80 may contain any ASCII text.~~

4.1.2.3 Value/Comment (bytes 11 through 80)

In keyword records that contain the value indicator in bytes 9 and 10, the remaining bytes 11 through 80 of the record ~~This field, when used,~~ *shall* contain the value, if any, of the keyword, followed by *optional* comments. In keyword records without a value indicator, bytes 9 through 80 *should* be interpreted as commentary text, however, this does not preclude conventions that interpret the content of these bytes in other ways.

The value field, when present, *shall* contain the ASCII text representation of a literal string constant, a logical constant, or a numerical constant, in the format specified in §4.2. The value field may be a null field; i.e., it *may* consist entirely of spaces, in which case ~~. If the value field is null,~~ the value associated with the keyword is undefined.

The mandatory FITS keywords defined in this standard *must not* appear more than once within a header. All other keywords that have a value *should not* appear more than once. If a keyword does appear multiple times with different values, then the value is indeterminate.

If a comment follows the value field ~~is present~~, it *must* be preceded by a slash ('`/`' , decimal 47 or hexadecimal 2F). A space between the value and the slash is strongly *recommended*. The comment *may* contain any of the restricted set of ASCII text characters, decimal 32 through 126 (hexadecimal 41 through 7E). The ASCII control characters with decimal values less than 32 (including the null, tab, carriage return, and line feed characters), and the delete character (decimal 127 or hexadecimal 7F) *must not* appear anywhere within a keyword record.~~The value *shall* be the ASCII text representation of a string or constant, in the format specified in §4.2. The comment *may* contain any ASCII text~~.

4.2 Value

The structure of the value field depends on the data type of the value ~~shall be determined by the type of the variable~~. The value field represents a single value and not an array of values. The value field *must* be in one of two formats: fixed or free. The fixed-format is required for values of mandatory keywords and is *recommended* for values of all other keywords~~others. This standard imposes no requirements on case sensitivity of character strings other than those explicitly specified~~.

4.2.1 Character String

~~If the value is a fixed-format character string , column 11 shall contain a single quote (hexadecimal code 27, “ ’ ”); the string shall follow, starting in column 12, followed by a closing single quote (also hexadecimal code 27) that should not occur before column 20 and *must* occur in or before column 80. The~~ A character string value *shall* be composed only of the set of restricted ASCII text characters, decimal 32 through 126 (hexadecimal 20 through 7E) enclosed by single quote characters (“ ’ ”, decimal 39, hexadecimal 27). A single quote is represented within a string as two successive single quotes, e.g., O’HARA = `'O''HARA'`. Leading spaces ~~blanks~~ are significant; trailing spaces ~~blanks~~ are not. This standard imposes no requirements on the case sensitivity of character string values unless explicitly stated in the definition of specific keywords.

If the value is a fixed-format character string, the starting single quote character *must* be in byte 11 of the keyword record and the closing single quote *must* occur in or before byte 80. Earlier versions of this standard also *required* that fixed-format characters strings *must* be padded with space characters to at least a length of 8 characters so that the closing quote character does not occur before byte 20. This minimum character string length is no longer required, except for the value of the `XTENSION` keyword (e.g., `'IMAGE␣␣␣'` and `'TABLE␣␣␣'`; see §7) which *must* be padded to a length of 8 characters for backward compatibility with previous usage.

Free-format character strings follow the same rules as fixed-format character strings except that the starting single quote character ~~and closing single quote characters~~ *may* occur after byte 11. ~~anywhere within columns 11–80.~~ Any bytes ~~columns~~ preceding the starting quote character and after byte ~~column~~ 10 *must* contain the space character.

Note that there is a subtle distinction between the following 3 keywords:

```
KEYWORD1= ''                  / null string keyword
KEYWORD2= '    '              / empty string keyword
KEYWORD3=                     / undefined keyword
```

The value of `KEYWORD1` is a null, or zero length string whereas the value of the `KEYWORD2` is an empty ~~a blank~~ string (nominally a single space ~~blank~~ character because the first space ~~blank~~ in the string is significant, but trailing spaces ~~blanks~~ are not). The value of `KEYWORD3` is undefined and has an indeterminate data type as well, except in cases where the data type of the specified keyword is explicitly defined in this standard.

The maximum possible ~~allowed~~ length of a keyword string is 68 characters (with the opening and closing quote characters in bytes ~~columns~~ 11 and 80, respectively). In general, no length limit less than 68 is implied for character-valued keywords.

4.2.2 Logical

If the value is a fixed-format logical constant, it *shall* appear as ~~a~~ an uppercase `T` or `F` in byte~~column~~ 30. A logical value is represented in free-format by a single character consisting of an uppercase `T` or `F` as the first non-space character in bytes ~~. This character *must* be the first non-blank character in columns~~ 11 through 80. ~~The only characters that *may* follow this single character are spaces, or a slash followed by an *optional* comment (see §4.1.2.3).~~

4.2.3 Integer Number

If the value is a fixed-format integer, the ASCII representation *shall* be right-justified in bytes ~~columns~~ 11 through 30. An integer consists of a '+' (decimal 43 or hexadecimal 2B) or '–' (decimal 45 or hexadecimal 2D) sign, followed by one or more contiguous ASCII digits (decimal 48 to 57 or hexadecimal 30 to 39), with no embedded spaces. The leading '+' sign is *optional*. Leading zeros are permitted, but are not significant. The integer representation *shall* always be ~~described here is always~~ interpreted as a signed, decimal number. This standard does not limit the range of an integer keyword value, however, software packages that read or write data according to this standard could be limited in the range of values that are supported (e.g., to the range that can be represented by a 32-bit or 64-bit signed binary integer).

A free-format integer value follows the same rules as fixed-format integers except that the ASCII representation ~~it~~ *may* occur anywhere within bytes ~~columns~~ 11 through 80.

4.2.4 Real Floating-Point Number

If the value is a fixed-format real floating-point number, the ASCII representation *shall* be ~~appear,~~ right-justified in bytes ~~, in columns~~ 11 through 30.

A floating-point number is represented by a decimal number followed by an *optional* exponent, with no embedded spaces. A decimal number *shall* consist ~~consists~~ of a '+' (decimal 43 or hexadecimal 2B) or '–' (decimal 45 or hexadecimal 2D) sign, followed by a sequence of ASCII digits containing a single decimal point ('.'), representing an integer part and a fractional part of the floating-point number. The leading '+' sign is *optional*. At least one of the integer part or fractional part *must* be present. If the fractional part is present, the decimal point *must* also be present. If only the integer part is present, the decimal point *may* be omitted, in which case the floating-point number is indistinguishable from an integer. The exponent, if present, consists of an exponent letter followed by an integer. Letters in the exponential form ('E' or 'D')[1] *shall* be upper case. ~~Note:~~ The full precision of 64-bit values cannot be expressed over the whole range of values using the fixed-format. This standard does not impose an upper limit on the number of digits of precision, nor any limit on the range of floating-point keyword values. Software packages that read or write data according to this standard could be limited, however, in the range of values and exponents that are supported (e.g., to the range that can be represented by a 32-bit or 64-bit floating-point number).

A free-format floating-point value follows the same rules as a fixed-format floating-point value except that the ASCII representation ~~values except that it~~ *may* occur anywhere within bytes ~~columns~~ 11 through 80.

4.2.5 Complex Integer Number

There is no fixed-format for complex integer numbers.

If the value is a complex integer number, the value *must* be represented as a real part and an imaginary part, separated by a comma and enclosed in parentheses e.g., `(123, 45)`. Spaces *may* precede and follow the real and imaginary parts. The real and imaginary parts are represented in the same way as integers (§4.2.3). Such a representation is regarded as a single value for the complex integer number. This representation *may* be located anywhere within bytes ~~columns~~ 11 through 80.

4.2.6 Complex Floating-Point Number

There is no fixed-format for complex floating-point numbers.

If the value is a complex floating-point number, the value *must* be represented as a real part and an imaginary part, separated by a comma and enclosed in parentheses, e.g., `(123.23, -45.7)`. Spaces *may* precede and follow the real and imaginary parts. The real and imaginary parts are represented in the same way as floating-point values (§4.2.4). Such a representation is regarded as a single value for the complex floating-point number. This representation *may* be located anywhere within bytes ~~columns~~ 11 through 80.

4.3 Units

[1] The 'D' exponent form is traditionally used when representing values that have more decimals of precision or a larger magnitude than can be represented by a single-precision 32-bit floating point number, but otherwise there is no distinction between 'E' or 'D'.

When a numerical keyword value represents a physical quantity, it is *recommended* that units be provided. Units *shall be* represented with a string of characters composed of the restricted ASCII text character set. Unit strings can be used as values of keywords (e.g., for the reserved keywords BUNIT, and TUNITn), as an entry in a character string column of an ASCII or binary table extension, or as part of a keyword comment string (see §4.3.2, below).

The units of all *FITS* header keyword values, with the exception of measurements of angles, *should* conform with the recommendations in the IAU Style Manual [19]. For angular measurements given as floating-point values and specified with reserved keywords, ~~degrees are the recommended units with the units, if specified, given as deg.~~ the units *should* be degrees (i.e., `deg`). If a requirement exists within this standard for the units of a keyword, then those units *must* be used

The units for fundamental physical quantities recommended by the IAU are given in Table 4.1, and additional units that are commonly used in astronomy are given in Table 4.2. The recommended plain text form for the IAU-recognized *base units* are given in column 2 of both tables.[2] All base units strings *may* be preceded, with no intervening spaces, by a single character (two for deca) taken from Table 4.3 and representing scale factors mostly in steps of 10^3. Compound prefixes (e.g., `ZYeV` for 10^{45} eV) *must* not be used.

4.3.1 Construction of Units Strings

Compound units strings *may* be formed by combining strings of base units (including prefixes, if any) with the recommended syntax described in Table 4.4. Two or more base units strings (called `str1` and `str2` in Table 4.4) *may* be combined using the restricted set of (explicit or implicit) operators that provide for multiplication, division, exponentiation, raising arguments to powers, or taking the logarithm or square-root of an argument. Note that functions such as `log` actually require dimensionless arguments, so that `log(Hz)`, for example, actually means `log`(x/1 Hz). The final units string is the compound string, or a compound of compounds, preceded by an *optional* numeric multiplier of the form `10**`k, `10^k`, or `10`$\pm k$ where k is an integer, *optionally* surrounded by parentheses with the sign character required in the third form in the absence of parentheses. Creators of *FITS* files are encouraged to use the numeric multiplier only when the available standard scale factors of Table 4.3 will not suffice. Parentheses are used for symbol grouping and are strongly *recommended* whenever the order of operations might be subject to misinterpretation. A space character implies multiplication which can also be conveyed explicitly with an asterisk or a period. Therefore, although spaces are allowed as symbol separators, their use is discouraged. Note that, per IAU convention, case is significant throughout. The IAU style manual forbids the

[2] These tables are reproduced from the first in a series of papers on world coordinate systems [11] which provides examples and expanded discussion.

Table 4.1. IAU-recommended basic units.

Quantity	Unit	Meaning	Notes
SI base & supplementary units			
length	`m`	meter	
mass	`kg`	kilogram	`g` gram allowed
time	`s`	second	
plane angle	`rad`	radian	
solid angle	`sr`	steradian	
temperature	`K`	kelvin	
electric current	`A`	ampere	
amount of substance	`mol`	mole	
luminous intensity	`cd`	candela	
IAU-recognized derived units			
frequency	`Hz`	hertz	s^{-1}
energy	`J`	joule	N m
power	`W`	watt	J s^{-1}
electric potential	`V`	volt	J C^{-1}
force	`N`	newton	kg m s^{-2}
pressure, stress	`Pa`	pascal	N m^{-2}
electric charge	`C`	coulomb	A s
electric resistance	`Ohm`	ohm	V A^{-1}
electric conductance	`S`	siemens	A V^{-1}
electric capacitance	`F`	farad	C V^{-1}
magnetic flux	`Wb`	weber	V s
magnetic flux density	`T`	tesla	Wb m^{-2}
inductance	`H`	henry	Wb A^{-1}
luminous flux	`lm`	lumen	cd sr
illuminance	`lx`	lux	lm m^{-2}

Table 4.2 Additional allowed units.

Quantity		Unit	Meaning	Notes
plane angle		`deg`	degree of arc	$\pi/180$ rad
		`arcmin`	minute of arc	$1/60$ deg
		`arcsec`	second of arc	$1/3600$ deg
		`mas`	milli-second of arc	$1/3\,600\,000$ deg
time		`min`	minute	60 s
		`h`	hour	60 min = 3600 s
		`d`	day	86 400 s
	†	`a`	year (Julian)	31 557 600 s (365.25 d), peta a (`Pa`) forbidden
	†	`yr`	year (Julian)	`a` is IAU-style
energy*	†	`eV`	electron volt	$1.6021765 \times 10^{-19}$ J
	‡	`erg`	erg	10^{-7} J
		`Ry`	rydberg	$\frac{1}{2}\left(\frac{2\pi e^2}{hc}\right)^2 m_e c^2 = 13.605692$ eV
mass*		`solMass`	solar mass	1.9891×10^{30} kg
		`u`	unified atomic mass unit	$1.6605387 \times 10^{-27}$ kg
luminosity		`solLum`	Solar luminosity	3.8268×10^{26} W
length	‡	`Angstrom`	angstrom	10^{-10} m
		`solRad`	Solar radius	6.9599×10^{8} m
		`AU`	astronomical unit	1.49598×10^{11} m
		`lyr`	light year	9.460730×10^{15} m
	†	`pc`	parsec	3.0857×10^{16} m
events		`count`	count	
		`ct`	count	
		`photon`	photon	
		`ph`	photon	
flux density	†	`Jy`	jansky	10^{-26} W m^{-2} Hz^{-1}
	†	`mag`	(stellar) magnitude	
	†	`R`	rayleigh	$10^{10}/(4\pi)$ photons m^{-2} s^{-1} sr^{-1}
magnetic field	†‡	`G`	gauss	10^{-4} T
area		`pixel`	(image/detector) pixel	
		`pix`	(image/detector) pixel	
	†‡	`barn`	barn	10^{-28} m^2
Miscellaneous units				
		`D`	debye	$\frac{1}{3} \times 10^{-29}$ C.m
		`Sun`	relative to Sun	e.g., ~~-~~abundances
		`chan`	(detector) channel	
		`bin`	numerous applications	(including the 1-d analogue of pixel)
		`voxel`	3-d analogue of pixel	
	†	`bit`	binary information unit	
	†	`byte`	(computer) byte	8 bit
		`adu`	Analog-to-digital converter	
		`beam`	beam area of observation	as in Jy/beam

† - addition of prefixes for decimal multiples and submultiples are allowed.
‡ - deprecated in IAU Style Manual [19] [~~7~~] but still in use.
∗ - conversion factors from CODATA Internationally recommended values of the fundamental physical constants 2002 ~~1998~~ (`http://physics.nist.gov/cuu/Constants/`).

Table 4.3. Prefixes for multiples and submultiples.

Submult	Prefix	Char	Mult	Prefix	Char
10^{-1}	deci	`d`	10	deca	`da`
10^{-2}	centi	`c`	10^2	hecto	`h`
10^{-3}	milli	`m`	10^3	kilo	`k`
10^{-6}	micro	`u`	10^6	mega	`M`
10^{-9}	nano	`n`	10^9	giga	`G`
10^{-12}	pico	`p`	10^{12}	tera	`T`
10^{-15}	femto	`f`	10^{15}	peta	`P`
10^{-18}	atto	`a`	10^{18}	exa	`E`
10^{-21}	zepto	`z`	10^{21}	zetta	`Z`
10^{-24}	yocto	`y`	10^{24}	yotta	`Y`

use of more than one solidus (/) character in a units string. However, since normal mathematical precedence rules apply in this context, more than one solidus *may* be used but is discouraged.

A unit raised to a power is indicated by the unit string followed, with no intervening spaces, by the *optional* symbols `**` or `^` followed by the power given as a numeric expression, called `expr` in Table 4.4. The power *may* be a simple integer, with or without sign, *optionally* surrounded by parentheses. It *may* also be a decimal number (e.g., 1.5, 0.5) or a ratio of two integers (e.g., 7/9), with or without sign, which *must* be surrounded by parentheses. Thus meters squared *may* be indicated by `m**(2)`, `m**+2`, `m+2`, `m2`, `m^2`, `m^(+2)`, etc. and per meter cubed *may* be indicated by `m**-3`, `m-3`, `m^(-3)`, `/m3`, and so forth. Meters to the three-halves power *may* be indicated by `m(1.5)`, `m^(1.5)`, `m**(1.5)`, `m(3/2)`, `m**(3/2)`, and `m^(3/2)`, but *not* by `m^3/2` or `m1.5`.

4.3.2 Units in Comment Fields

If the units of the keyword value are specified in the comment of the header keyword, it is *recommended* that the units string be enclosed in square brackets (i.e., enclosed by '[' and ']') at the beginning of the comment field, separated from the slash ('/') comment field delimiter by a single space character. An example, using a non-standard keyword, is

`EXPTIME` = 1200. / [s] exposure time in seconds

This widespread, but *optional*, practice suggests that square brackets *should* be used in comment fields only for this purpose. Nonetheless, software *should* not depend on

Table 4.4. Characters and strings allowed to denote mathematical operations.

String	Meaning
`str1 str2`	Multiplication
`str1*str2`	Multiplication
`str1.str2`	Multiplication
`str1/str2`	Division
`str1**expr`	Raised to the power `expr`
`str1^expr`	Raised to the power `expr`
`str1expr`	Raised to the power `expr`
`log(str1)`	Common Logarithm (to base 10)
`ln(str1)`	Natural Logarithm
`exp(str1)`	Exponential ($e^{\texttt{str1}}$)
`sqrt(str1)`	Square root

units being expressed in this fashion within a keyword comment, and software *should* not depend on any string within square brackets in a comment field containing a proper units string.

4.4 Keywords

4.4.1 Mandatory Keywords

Mandatory keywords are required in every HDU as described in the remainder of this subsection. They *must* ~~may~~ be used only as described in this standard. Values of the mandatory keywords *must* be written in fixed-format.

4.4.1.1 Primary Header~~Principal~~

The `SIMPLE` keyword is required to be the first keyword in the primary header of all *FITS* files. ~~Principal mandatory keywords other than `SIMPLE` are required in all *FITS* headers.~~ The ~~card images of any~~ primary header *must* contain the other mandatory keywords shown in Table 4.5 in the order given. Other keywords *must* not ~~No other keywords may~~ intervene between the `SIMPLE` keyword and the last `NAXISn` keyword.

`SIMPLE` **Keyword** The value field *shall* contain a logical constant with the value `T` if the file conforms to this standard. This keyword is mandatory for the primary header

Table 4.5. Mandatory keywords for primary header.

#	Keyword
1	`SIMPLE = T`
2	`BITPIX`
3	`NAXIS`
4	`NAXISn`, `n` = 1, . . . , `NAXIS`
	⋮
	(other keywords)
	⋮
last	`END`

Table 4.6. Interpretation of valid `BITPIX` value.

Value	Data Represented
8	Character or unsigned binary integer
16	16-bit two's complement binary integer
32	32-bit two's complement binary integer
64	64-bit two's complement binary integer
-32	IEEE single precision floating-point
-64	IEEE double precision floating-point

and *must* not appear ~~is not permitted~~ in extension headers. A value of `F` signifies that the file does not conform to this standard.

`BITPIX` Keyword The value field *shall* contain an integer. The absolute value is used in computing the sizes of data structures. It *shall* specify the number of bits that represent a data value in the associated data array. The only valid values of `BITPIX` are given in Table 4.6. Writers of *FITS* arrays *should* select a `BITPIX` data type appropriate to the form, range of values, and accuracy of the data in the array.

`NAXIS` Keyword The value field *shall* contain a non-negative integer no greater than 999 ~~,~~ representing the number of axes in the associated data array. A value of zero signifies that no data follow the header in the HDU.

`NAXISn` **Keywords** The `NAXISn` keywords *must* be present for all values `n = 1,...,NAXIS`, in increasing order of `n`, and for no other values of `n`. The value field of this indexed keyword *shall* contain a non-negative integer representing the number of elements along axis `n` of a data array. A value of zero for any of the `NAXISn` signifies that no data follow the header in the HDU (however, the random groups structure described in §6 has `NAXIS1` = 0, but will have data following the header if the other `NAXISn` keywords are non-zero). If `NAXIS` is equal to 0, there *shall* ~~*should*~~ *not* be any `NAXISn` keywords.

`END` **Keyword** This keyword has no associated value. Bytes ~~Columns~~ 9 through 80 *shall* be filled with ASCII ~~blanks.~~ spaces (decimal 32 or hexadecimal 20). The `END` keyword marks the logical end of the header and must occur in the last 2880-byte *FITS* block of the header.

The total number of bits in the primary data array, exclusive of fill that is needed after the data to complete the last 2880-byte data block ~~record~~ (§3.3.2), is given by the following expression:

$$N_{\text{bits}} = |\texttt{BITPIX}| \times (\texttt{NAXIS1} \times \texttt{NAXIS2} \times \cdots \times \texttt{NAXISm}), \tag{4.1}$$

where N_{bits} *must* be~~is~~ non-negative and is the number of bits excluding fill, `m` is the value of `NAXIS`, and `BITPIX` and the `NAXISn` represent the values associated with those keywords. Note that the random groups convention in the primary array has a more complicated structure whose size is given by Eq. 6.1. The header of the first *FITS* extension in the file, if present, *shall* start with the first *FITS* block following the data block that contains the last bit of the primary data array.

An example of a primary array header is shown in Table 4.7. In addition to the required keywords, it includes a few of the reserved keywords that are discussed in §4.4.2.

4.4.1.2 Conforming Extensions

All conforming extensions, whether or not further specified in this standard, *must* use the keywords defined in Table 4.8 in the order specified. Other keywords *must* not ~~No other keywords may~~ intervene between the `XTENSION` keyword and the `GCOUNT` ~~last `NAXISn`~~ keyword. The `BITPIX`, `NAXIS`, `NAXISn`, and `END` keywords are defined in §4.4.1.1~~This organization is required for any conforming extension, whether or not further specified in this standard~~.

Table 4.7. Example of a primary array header.

Keyword Records
`SIMPLE  =                    T / file does conform to FITS standard`
`BITPIX  =                   16 / number of bits per data pixel`
`NAXIS   =                    2 / number of data axes`
`NAXIS1  =                  250 / length of data axis 1`
`NAXIS2  =                  300 / length of data axis 2`
`OBJECT  = 'Cygnus X-1'`
`DATE    = '2006-10-22'`
`END`

Table 4.8. Mandatory keywords in conforming extensions.

#	Keyword
1	XTENSION
2	BITPIX
3	NAXIS
4	NAXISn, n = 1, ..., NAXIS
5	~~⋮(other keywords, including ...)~~ PCOUNT
6	GCOUNT
	⋮
	(other keywords)
	⋮
last	END

`XTENSION` **Keyword** The value field *shall* contain a character string giving the name of the extension type. This keyword is mandatory for an extension header and *must not* appear in the primary header. ~~For an extension that is not a standard extension, the type name must not be the same as that of a standard extension .~~ To preclude conflict, extension type names *must* be registered with the IAUFWG. The current list of registered extensions is given in Appendix F. An up to date list is also maintained on the *FITS* Support Office web site.

~~The IAUFWG may specify additional type names that must be used only to identify specific types of extensions; the full list shall be available from the *FITS* SupportOffice.~~

~~**EXTEND Keyword** The use of extensions necessitates a single additional keyword in the primary header of the *FITS* file. If the *FITS* file may contain extensions, a card image with the keyword `EXTEND` and the value field containing the logical value `T` must appear in the primary header immediately after the last `NAXISn` card image, or, if `NAXIS` =0, the `NAXIS` card image. The presence of this keyword with the value `T` in the primary header does not require that extensions be present.~~

`PCOUNT` **Keyword** The value field *shall* contain an integer that *shall* be used in any way appropriate to define the data structure, consistent with Eq. 4.2. In `IMAGE` (§7.1) and `TABLE` (§7.2) extensions this keyword *must* have the value 0; in `BINTABLE` extensions (§7.3) it is used to specify the number of bytes that follow the main data table in the supplemental data area called the heap. This keyword is also used in the random groups structure (§6) to specify the number of parameters preceding each array in a group.

`GCOUNT` **Keyword** The value field *shall* contain an integer that *shall* be used in any way appropriate to define the data structure, consistent with Eq. 4.2. This keyword *must* have the value 1 in the `IMAGE`, `TABLE` and `BINTABLE` standard extensions defined in §7. This keyword is also used in the random groups structure (§6) to specify the number of random groups present.

The total number of bits in the extension data array (exclusive of fill that is needed after the data to complete the last 2880-byte data block~~record such as that for the primary data array(§3.3.2~~) is given by the following expression:

$$
\begin{aligned}
N_{\text{bits}} \;=\; & |\mathtt{BITPIX}| \times \mathtt{GCOUNT} \times \\
& (\mathtt{PCOUNT} + \mathtt{NAXIS1} \times \mathtt{NAXIS2} \times \cdots \times \mathtt{NAXISm}),
\end{aligned} \tag{4.2}
$$

where N_{bits} *must* be~~is~~ non-negative and is the number of bits excluding fill, `m` is the value of `NAXIS`, and `BITPIX`, `GCOUNT`, `PCOUNT`, and the `NAXISn` represent the values associated with those keywords. If $N_{\text{bits}} > 0$, then the data array *shall* be contained in an integral number of 2880-byte *FITS* data blocks. The header of the next *FITS* extension in the file, if any, *shall* start with the first *FITS* block following the data block that contains the last bit of the current extension data array.

4.4.2 Other Reserved Keywords

The reserved keywords described below ~~These keywords~~ are *optional*, but if present in the header they *must* ~~*may*~~ be used only as defined in this standard. They apply to any *FITS* structure with the meanings and restrictions defined below. Any *FITS* structure *may* further restrict the use of these keywords.

4.4.2.1 General Descriptive Keywords ~~Describing the History or Physical Construction of the HDU~~

`DATE` Keyword ~~Starting January 1, 2000, the following format shall be used. *FITS* writers should commence writing the value of the `DATE` keyword in this format starting January 1, 1999 and before January 1, 2000.~~ The value field *shall* contain a character string giving the date on which the HDU was created, in the form `YYYY-MM-DD`, or the date and time when the HDU was created, in the form `YYYY-MM-DDThh:mm:ss`[`.sss`...], where `YYYY` *shall* be the four-digit calendar year number, `MM` the two-digit month number with January given by 01 and December by 12, and `DD` the two-digit day of the month. When both date and time are given, the literal `T` *shall* separate the date and time, `hh` *shall* be the two-digit hour in the day, `mm` the two-digit number of minutes after the hour, and `ss`[`.sss`...] the number of seconds (two digits followed by an *optional* fraction) after the minute. Default values *must* not be given to any portion of the date/time string, and leading zeros *must* not be ~~No fields may be defaulted and no leading zeroes~~ omitted. The decimal part of the seconds field is *optional* and *may* be arbitrarily long, so long as it is consistent with the rules for value formats of §4.2.

The value of the `DATE` keyword *shall* always be expressed in UTC when in this format, for all data sets created on Earth~~earth~~.

The following format *may* appear on files written before January 1, 2000. The value field contains a character string giving the date on which the HDU was created, in the form `DD/MM/YY`, where `DD` is the day of the month, `MM` the month number with January given by 01 and December by 12, and `YY` the last two digits of the year, the first two digits being understood to be 19. Specification of the date using Universal Time is *recommended* but not assumed.

When a newly created HDU is substantially a verbatim copy of another HDU, the value of the `DATE` keyword in the original HDU *may* be retained in the new HDU instead of updating the value to the current date and time.~~Copying of a *FITS* file does not require changing any of the keyword values in the file's HDUs.~~

`ORIGIN` **Keyword** The value field *shall* contain a character string identifying the organization or institution responsible for creating the *FITS* file.

`EXTEND` **Keyword** The value field *shall* contain a logical value indicating whether the FITS file is allowed to cointain conforming extensions following the primary HDU. This keyword *may* only appear in the primary header and *must not* appear in an extension header. If the value field is T, then there *may* be conforming extensions in the *FITS* file following the primary HDU. This keyword is only advisory, so its presence with a value T does not require that the *FITS* file contains extensions, nor does the absence of this keyword necessarily imply that the file does not contain extensions. Earlier versions of this standard stated that the `EXTEND` keyword *must* be present in the primary header if the file contained extensions, but this is no longer required.

`BLOCKED` Keyword ~~This keyword may be used only in the primary header. It shall appear within the first 36 card images of the FITS file. (Note: This keyword thus cannot appear if NAXIS is greater than 31, or if NAXIS is greater than 30 and the EXTEND keyword is present.)~~ This keyword is deprecated and *should* not be used in new *FITS* files. It is reserved primarily to prevent its use with other meanings. As previously defined, this keyword, if used, was *required* to appear only within the first 36 keywords in the primary header. Its presence with the required logical value of `T` advised ~~advises~~ that the physical block size of the *FITS* file on which it appears *may* be an integral multiple of the *FITS* block ~~logical record~~ length and not necessarily equal to it. ~~Physical block size and logical record length may be equal even if this keyword is present or unequal if it is absent. It is reserved primarily to prevent its use with other meanings. Since the issuance of version 1 of this standard, the BLOCKED keyword has been deprecated.~~

4.4.2.2 Keywords Describing Observations

`DATE-OBS` **Keyword** The format of the value field for `DATE-OBS` keywords *shall* follow the prescriptions for the `DATE` keyword (§4.4.2.1). Either the 4-digit year format or the 2-digit year format *may* be used for observation dates from 1900 through 1999 although the 4-digit format is *recommended* ~~preferred~~.

When the format with a four-digit year is used, the default interpretations for time *should* ~~*shall*~~ be UTC for dates beginning 1972-01-01 and UT before. Other date and time scales are permissible. The value of the `DATE-OBS` keyword *shall* be expressed in the principal time system or time scale of the HDU to which it belongs; if there is any chance of ambiguity, the choice *should* ~~*shall*~~ be clarified in comments. The value of `DATE-OBS` *shall* be assumed to refer to the start of an observation, unless another interpretation is clearly explained in the comment field. Explicit specification of the time scale is *recommended*. By default, times for TAI and times that run simultaneously with TAI, e.,g., UTC and TT, will be assumed to be as measured at the detector (or, in practical cases, at the observatory). For coordinate times such as TCG, TCB, and TDB ~~which~~

~~are tied to an unambiguous coordinate system~~, the default *shall* be to include light-time corrections to the associated spatial origin, namely the geocenter for TCG and the solar-system barycenter for the other two~~time as if the observation had taken place at the origin of the coordinate time system~~. Conventions *may* be developed that use other time systems. Appendix B of this document contains the appendix to the agreement on a four digit year, which discusses time systems in some detail.

When the value of `DATE-OBS` is expressed in the two-digit year form, allowed for files written before January 1, 2000 with a year in the range 1900-1999, there is no default assumption as to whether it refers to the start, middle or end of an observation.

`DATExxxx` Keywords The value fields for all keywords beginning with the string `DATE` whose value contains date, and *optionally* time, information *shall* follow the prescriptions for the `DATE-OBS` keyword.

`TELESCOP` Keyword The value field *shall* contain a character string identifying the telescope used to acquire the data associated with the header.

`INSTRUME` Keyword The value field *shall* contain a character string identifying the instrument used to acquire the data associated with the header.

`OBSERVER` Keyword The value field *shall* contain a character string identifying who acquired the data associated with the header.

`OBJECT` Keyword The value field *shall* contain a character string giving a name for the object observed.

~~**EQUINOX Keyword** The value field shall contain a floating-point number giving the equinox in years for the celestial coordinate systemin which positions are expressed.~~

4.4.2.3 Bibliographic Keywords

`AUTHOR` Keyword The value field *shall* contain a character string identifying who compiled the information in the data associated with the header. This keyword is appropriate when the data originate in a published paper or are compiled from many sources.

`REFERENC` Keyword The value field *shall* contain a character string citing a reference where the data associated with the header are published. It is *recommended* that either the 19-digit bibliographic identifier used in the Astrophysics Data System bibliographic databases (http://adswww.harvard.edu/) or the Digital Object Identifier (http://doi.org) be included in the value string when available (e.g., '1994A&AS..103..135A' or 'doi:10.1006/jmbi.1998.2354').

4.4.2.4 Commentary Keywords

These keywords provide commentary information about the contents or history of the *FITS* file and may occur any number of times in a header. These keywords *shall* have no associated value even if the value indicator characters '=␣' appear in bytes 9 and 10 (hense it is *recommended* that these keywords not contain the value indicator). Bytes 9 through 80 *may* contain any of the restricted set of ASCII text characters, decimal 32 through 126 (hexadecimal 20 through 7E).

COMMENT Keyword This keyword *may* be used to supply any comments regarding the *FITS* file. ~~This keyword shall have no associated value; columns 9 through 80 may contain any ASCII text. Any number of COMMENT card images may appear in a header.~~

HISTORY Keyword This keyword *should* be used to describe the history of steps and procedures associated with the processing of the associated data. ~~This keyword shall have no associated value; columns 9 through 80 may contain any ASCII text. The text should contain a history of the steps and procedures associated with the processing of the associated data. Any number of HISTORY card images may appear in a header.~~

Keyword Field is blank This keyword *may* be used to supply any comments regarding the *FITS* file. It is frequently used for aesthetic purposes to provide a break between groups of related keywords in the header.~~Columns 1 through 8 contain ASCII blanks. Columns 9 through 80 may contain any ASCII text. Any number of card images with blank keyword fields may appear in a header~~

4.4.2.5 ~~Array~~ Keywords that Describe Arrays

These keywords are used to describe the contents of an array, either in the primary array, in an IMAGE extension (§7.1), ~~alone~~ or in a series of random groups (§6). They are *optional*, but if they appear in the header describing an array or groups, they *must* be used as defined in this section of this standard. They *shall not* be used in headers describing other structures unless the meaning is the same as defined here~~that for a primary or groups array~~.

BSCALE Keyword This keyword *shall* be used, along with the BZERO keyword, to linearly scale the array pixel values (i.e., the actual values stored in the *FITS* file) to transform them into the ~~when the array pixel values are not the true~~physical values that

Table 4.9. Usage of BZERO to represent non-default integer data types.

BITPIX	Native Data Type	Physical Data Type	BZERO	
8	unsigned	signed byte	-128	(-2^7)
16	signed	unsigned 16-bit	32768	(2^{15})
32	signed	unsigned 32-bit	2147483648	(2^{31})
64	signed	unsigned 64-bit	9223372036854775808	(2^{63})

they represent ~~, to transform the primary data array values to the true physical values they represent,~~ using Eq. 4.3.

$$\text{physical\_value} = \texttt{BZERO} + \texttt{BSCALE} \times \text{array\_value} \tag{4.3}$$

The value field *shall* contain a floating-point number representing the coefficient of the linear term in the scaling equation, the ratio of physical value to array value at zero offset. The default value for this keyword is 1.0. Before support for IEEE floating-point data types was added to *FITS* [7], this technique of linearly scaling integer values was the only way to represent the full range of floating-point values in a *FITS* array. This linear scaling technique is still commonly used to reduce the size of the data array by a factor of 2 by representing 32-bit floating-point physical values as 16-bit scaled integers.

BZERO Keyword This keyword *shall* be used, along with the BSCALE keyword, to linearly scale the array pixel values (i.e., the actual values stored in the *FITS* file) to transform them into the physical values that they represent using Eq. 4.3. ~~when the array pixel values are not the true physical values , to transform the primary data array values to the true values.~~ The value field *shall* contain a floating-point number representing the physical value corresponding to an array value of zero. The default value for this keyword is 0.0.

Besides its use in representing floating point values as scaled integers (see the description of the BSCALE keyword), the BZERO keyword is also used when storing unsigned integer values in the *FITS* array. In this special case the BSCALE keyword *shall* have the default value of 1.0, and the BZERO keyword *shall* have one of the integer values shown in Table 4.9.

Since the *FITS* format does not support a native unsigned integer data type (except for the unsigned 8-bit byte data type), the unsigned values are stored in the *FITS* array as native signed integers with the appropriate integer offset specified by the BZERO keyword value shown in the table. For the byte data type, the converse technique can be used to store signed byte values as native unsigned values with the negative BZERO offset. In each case, the physical value is computed by adding the offset specified by the BZERO keyword to the native data type value that is stored in the *FITS* file.[3]

BUNIT Keyword The value field *shall* contain a character string describing the physical units in which the quantities in the array, after application of BSCALE and BZERO, are expressed. These units *must* follow the prescriptions of §4.3.

BLANK Keyword This keyword *shall* be used only in headers with positive values of BITPIX (i.e., in arrays with integer data). Bytes ~~Columns~~ 1 through 8 contain the string ~~,~~ 'BLANK␣␣␣' (ASCII spaces in bytes ~~blanks in columns~~ 6 through 8). The value field *shall* contain an integer that specifies the value that is used within the integer array to represent pixels that have an undefined physical value~~representation of array values whose physical values are undefined~~.

If the BSCALE and BZERO keywords do not have the default values of 1.0 and 0.0, respectively, then the value of the BLANK keyword *must* equal the actual value in the *FITS* data array that is used to represent an undefined pixel and not the corresponding physical value (computed from Eq. 4.3) . To cite a specific, common example, *unsigned* 16-bit integers are represented in a *signed* integer *FITS* array (with BITPIX = 16) by setting BZERO = 32768 and BSCALE = 1. If it is desired to use pixels that have an *unsigned* value (i.e., the physical value) equal to 0 to represent undefined pixels in the array, then the BLANK keyword *must* be set to the value −32768 because that is the actual value of the undefined pixels in the *FITS* array.

~~**CTYPEn Keywords** The value field shall contain a character string, givingthe name of the coordinaterepresented by axis n.~~

~~**CRPIXn Keywords** The value field shall contain a floating-point number, identifying the location of a reference pointalong axis n in unitsof the axis index. This value is based upon a counter that runs from 1 to NAXISn with an increment of 1 per pixel. The reference point value need not be that for the center of a pixel nor lie within the actual data array. Use comments to indicate the location of the index point relative to the pixel.~~

[3] A more computationally efficient method of adding or subtracting the BZERO values is to simply flip the most significant bit of the binary value. To give a simple example using 8-bit integers, the decimal value 248, minus the BZERO value of 128 equals 120. The binary representation of 248 is 11111000. Flipping the most significant bit gives the binary value 01111000, which is equal to decimal 120.

~~**CRVALn Keywords** The value field shall contain a floating-point number, giving the value of the coordinate specified by the CTYPEn keyword at the reference point CRPIXn. Units mus tfollow the prescriptions of §4.3.~~

~~**CDELTn Keywords** The value field shall contain a floating-point number giving the partial derivative of the coordinate specified by the CTYPEn keywords with respect to the pixel index, evaluated at the reference point CRPIXn in units of the coordinate specified by the CTYPEn keyword. These units must follow the prescriptions of §4.3.~~

~~**CROTAn Keywords** This keyword is used to indicate a rotation from a standard coordinate system described by the CTYPEn to a different coordinate system in which the values in the array are actually expressed. Rules for such rotations are not further specified in this standard; the rotation should be explained in comments. The value field shall contain a floating-point number giving the rotation angle in degrees between axis n and the direction implied by the coordinate system defined by CTYPEn.~~

DATAMAX Keyword The value field *shall* always contain a floating-point number, regardless of the value of BITPIX. This number *shall* give the maximum valid physical value represented by the array (from Eq. 4.3), exclusive of any IEEE special values.

DATAMIN Keyword The value field *shall* always contain a floating-point number, regardless of the value of BITPIX. This number *shall* give the minimum valid physical value represented by the array (from Eq. 4.3), exclusive of any IEEE special values.

WCS Keywords An extensive set of keywords have been defined to describe the world coordinates associated with an array. These keywords are discussed separately in §8.

4.4.2.6 Extension Keywords

~~These keywords are used to describe an extension and should not~~ Although these keywords were originally defined for use within the header of a conforming extension, they also *may* appear in the primary header with an analogous meaning. If these keywords are present, it is *recommended* that they have a unique combination of values in each HDU of the *FITS* file .

EXTNAME Keyword The value field *shall* contain a character string to be used to distinguish among different extensions of the same type, i.e., with the same value of XTENSION, in a *FITS* file. Within this context, the primary array *should* be considered as equivalent to an IMAGE extension.

`EXTVER` **Keyword** The value field *shall* contain an integer to be used to distinguish among different extensions in a *FITS* file with the same type and name, i.e., the same values for `XTENSION` and `EXTNAME`. The values need not start with 1 for the first extension with a particular value of `EXTNAME` and need not be in sequence for subsequent values. If the `EXTVER` keyword is absent, the file *should* be treated as if the value were 1.

`EXTLEVEL` **Keyword** The value field *shall* contain an integer specifying the level in a hierarchy of extension levels of the extension header containing it. The value *shall* be 1 for the highest level; levels with a higher value of this keyword *shall* be subordinate to levels with a lower value. If the `EXTLEVEL` keyword is absent, the file *should* be treated as if the value were 1.

4.4.3 Additional Keywords

4.4.3.1 ~~Requirements~~

New keywords *may* be devised in addition to those described in this standard, so long as they are consistent with the generalized rules for keywords and do not conflict with mandatory or reserved keywords. Any keyword that refers to or depends upon the existence of other specific HDUs in the same or other files should be used with caution because the persistence of those HDUs cannot always be guaranteed.

4.4.3.2 ~~Restrictions~~

~~No keyword in the primary header shall specify the presence of a specific extension *FITS* file; only the `EXTEND` keyword described in §4.4.2.1 shall be used to indicate the possible presence of extensions. No keyword in either the primary or extension header shall explicitly refer to the physical block size, other than the deprecated `BLOCKED` keyword of §4.4.2.1.~~

Section 5

Data Representation

Primary and extension data *shall* be represented in one of the formats described in this section. *FITS* data *shall* be interpreted to be a byte stream. Bytes are in big endian order of decreasing significance. The byte that includes the sign bit *shall* be first, and the byte that has the ones bit *shall* be last.

5.1 Characters

Each character *shall* be represented by one byte. A character *shall* be represented by its 7-bit ASCII [20] ~~14~~code in the low order seven bits in the byte. The high-order bit *shall* be zero.

5.2 Integers

5.2.1 Eight-bit

Eight-bit integers *shall* be unsigned binary integers, contained in one byte with decimal values ranging from 0 to 255.

5.2.2 Sixteen-bit

Sixteen-bit integers *shall* be two's complement signed binary integers, contained in two bytes with decimal values ranging from -32768 to +32767.

5.2.3 Thirty-two-bit

Thirty-two-bit integers *shall* be two's complement signed binary integers, contained in four bytes with decimal values ranging from -2147483648 to +2147483647.

5.2.4 Sixty-four-bit

Sixty-four-bit integers *shall* be two's complement signed binary integers, contained in eight bytes with decimal values ranging from -9223372036854775808 to +9223372036854775807.

5.2.5 Unsigned Integers

~~Unsigned sixteen-bit integers can be represented in FITS files by subtracting 32768 from each value (thus offsetting the values into the range of a signed sixteen-bit integer) before writing them to the *FITS* file. The BZERO keyword (or the TZEROn keyword in the case of binary table columns with TFORMn = 'I') must also be included in the header with its value set to 32768 so that *FITS* reading software will add this offset to the raw values in the FITS file, thus restoring them to the original unsigned integer values. Unsigned thirty-two-bit integers can be represented in *FITS* files in a similar way by applying an offset of 2147483648 (2^{31})to the data values. Unsigned sixty-four-bit integers can be represented in *FITS* files by applying an offset of 9223372036854775808 (2^{63}) to the data values.~~

The *FITS* format does not support a native unsigned integer data type (except for the unsigned 8-bit byte data type), therefore unsigned 16-bit, 32-bit, or 64-bit binary integers cannot be stored directly in a *FITS* data array. Instead, the appropriate offset *must* be applied to the unsigned integer to shift the value into the range of the corresponding signed integer, which is then stored in the *FITS* file. The BZERO keyword *shall* record the amount of the offset needed to restore the original unsigned value. The `BSCALE` keyword *shall* have the default value of 1.0 in this case, and the appropriate `BZERO` value, as a function of `BITPIX`, is specified in Table 4.9.

This same technique *must* be used when storing unsigned integers in a binary table column of signed integers (§7.3.2). In this case the `TSCALn` keyword (analogous to `BSCALE`) *shall* have the default value of 1.0, and the appropriate `TZEROn` value (analogous to `BZERO`) is specified in Table 7.7.

5.3 IEEE-754 Floating-Point

Transmission of 32- and 64-bit floating-point data within the *FITS* format *shall* use the ANSI/IEEE-754 standard [21]~~15~~. `BITPIX = -32` and `BITPIX = -64` signify 32- and 64-bit IEEE floating-point numbers, respectively; the absolute value of `BITPIX` is used for computing the sizes of data structures. The full IEEE set of number forms is allowed for *FITS* interchange, including all special values.

The `BLANK` keyword *should not* be used when `BITPIX = -32` or `-64`; rather, the IEEE NaN *should* be used to represent an undefined value. Use of the `BSCALE` and `BZERO` keywords is *not recommended.*

Appendix E has additional details on the IEEE format.

Section 6

Random Groups Structure

The random groups structure allows a collection of ‘groups’, where a group consists of a subarray of data and a set of associated parameter values, to be stored within the *FITS* primary data array. ~~Although it is standard *FITS*, the random groups structure has~~ Random groups have been used almost exclusively for applications in radio interferometry; outside this field, there is little support for reading or writing data in this format. Other than the existing use for radio interferometry data, the random groups structure is deprecated and *should not* be further used. ~~few *FITS* readers can read data in random groups format. The binary table extension (§7.3) can accommodate the structure described by random groups. While existing *FITS* files use the format, and it is therefore included in this standard, its use for future applications has been deprecated since the issue of Version 1 of this standard. Use of the word ‘deprecated’ is understood to mean that binary table extensions *should* be used in new astronomical application areas instead of the random groups format where either is appropriate and where there is no historical precedent for random groups. Existing applications of the random groups structure (almost exclusively interferometry) *may* continue to use random groups as needed indefinitely.~~ For other applications, the binary table extension (§7.3) provides a more extensible and better documented way of associating groups of data within a single data structure.

6.1 Keywords

6.1.1 Mandatory Keywords

The `SIMPLE` keyword is required to be the first keyword in the primary header of all *FITS* files, including those with random groups records. If the random groups format records follow the primary header, the keyword records ~~card images~~ of the primary header *must* use the keywords defined in Table 6.1 in the order specified. No other keywords *may* intervene between the `SIMPLE` keyword and the last `NAXISn` keyword.

Table 6.1. Mandatory keywords in primary header preceding random groups.

#	Keyword
1	SIMPLE = T
2	BITPIX
3	NAXIS
4	NAXIS1 = 0
5	NAXISn, n=2, . . . , value of NAXIS
	⋮
	(other keywords, which *must* include . . .)
	GROUPS = T
	PCOUNT
	GCOUNT
	⋮
last	END

SIMPLE Keyword The keyword record ~~card image~~ containing this keyword is structured in the same way as if a primary data array were present (§4.4.1).

BITPIX Keyword The keyword record ~~card image~~ containing this keyword is structured as prescribed in §4.4.1.

NAXIS Keyword The value field *shall* contain an integer ranging from 1 to 999, representing one more than the number of axes in each data array.

NAXIS1 Keyword The value field *shall* contain the integer 0, a signature of random groups format indicating that there is no primary data array.

NAXISn Keywords (n=2, . . . , value of NAXIS) The NAXISn keywords *must* be present for all values n = 2, ..., NAXIS, in increasing order of n, and for no larger values of n. The value field *shall* contain an integer, representing the number of positions along axis n-1 of the data array in each group.

GROUPS Keyword The value field *shall* contain the logical constant T. The value T associated with this keyword implies that random groups records are present.

`PCOUNT` **Keyword** The value field *shall* contain an integer equal to the number of parameters preceding each array in a group.

`GCOUNT` **Keyword** The value field *shall* contain an integer equal to the number of random groups present.

`END` **Keyword** This keyword has no associated value. Bytes 9 through 80 *shall* contain ASCII spaces (decimal 32 or hexadecimal 20). ~~The card image containing this keyword is structured as described in§4.4.1.~~

The total number of bits in the random groups records exclusive of the fill described in §6.2 is given by the following expression:

$$\begin{aligned} N_{\text{bits}} &= |\texttt{BITPIX}| \times \texttt{GCOUNT} \times \\ &\quad (\texttt{PCOUNT} + \texttt{NAXIS2} \times \texttt{NAXIS3} \times \cdots \times \texttt{NAXISm}), \end{aligned} \tag{6.1}$$

where N_{bits} is non-negative and the number of bits excluding fill, `m` is the value of `NAXIS`, and `BITPIX`, `GCOUNT`, `PCOUNT`, and the `NAXISn` represent the values associated with those keywords.

6.1.2 Reserved Keywords

`PTYPEn` **Keywords** The value field *shall* contain a character string giving the name of parameter `n`. If the `PTYPEn` keywords for more than one value of `n` have the same associated name in the value field, then the data value for the parameter of that name is to be obtained by adding the derived data values of the corresponding parameters. This rule provides a mechanism by which a random parameter *may* have more precision than the accompanying data array elements; for example, by summing two 16-bit values with the first scaled relative to the other such that the sum forms a number of up to 32-bit precision.

`PSCALn` **Keywords** This keyword *shall* be used, along with the `PZEROn` keyword, when the n$^{\text{th}}$ *FITS* group parameter value is not the true physical value, to transform the group parameter value to the true physical values it represents, using Eq. 6.2. The value field *shall* contain a floating-point number representing the coefficient of the linear term in Eq. 6.2, the scaling factor between true values and group parameter values at zero offset. The default value for this keyword is 1.0.

`PZEROn` Keywords This keyword *shall* be used, along with the `PSCALn` keyword, when the $\mathtt{n}^{\mathrm{th}}$ *FITS* group parameter value is not the true physical value, to transform the group parameter value to the physical value. The value field *shall* contain a floating-point number, representing the true value corresponding to a group parameter value of zero. The default value for this keyword is 0.0. The transformation equation is as follows:

$$\text{physical\_value} = \mathtt{PZEROn} + \mathtt{PSCALn} \times \text{group\_parameter\_value} \tag{6.2}$$

6.2 Data Sequence

Random groups data *shall* consist of a set of groups. The number of groups *shall* be specified by the `GCOUNT` keyword in the associated header~~record~~. Each group *shall* consist of the number of parameters specified by the `PCOUNT` keyword followed by an array with the number of elements N_{elem} given by the following expression:

$$N_{\text{elem}} = (\mathtt{NAXIS2} \times \mathtt{NAXIS3} \times \cdots \times \mathtt{NAXISm}), \tag{6.3}$$

where N_{elem} is the number of elements in the data array in a group, `m` is the value of `NAXIS`, and the `NAXISn` represent the values associated with those keywords.

The first parameter of the first group *shall* appear in the first location of the first data block~~record~~. The first element of each array *shall* immediately follow the last parameter associated with that group. The first parameter of any subsequent group *shall* immediately follow the last element of the array of the previous group. The arrays *shall* be organized internally in the same way as an ordinary primary data array. If the groups data do not fill the final data block~~record~~, the remainder of the block ~~record~~ *shall* be filled with zero values in the same way as a primary data array (§3.3.2). If random groups records are present, there *shall* be no primary data array.

6.3 Data Representation

Permissible data representations are those listed in §5. Parameters and elements of associated data arrays *shall* have the same representation. If ~~Should~~ more precision is ~~be~~ required for an associated parameter than for an element of a data array, the parameter *shall* be divided into two or more addends, represented by the same value for the `PTYPEn` keyword. The value *shall* be the sum of the physical values, which *may* have been obtained from the group parameter values using the `PSCALn` and `PZEROn` keywords.

Section 7

Standard Extensions

A standard extension is a conforming extension whose organization and content are completely specified in this standard. The specifications for the 3 currently defined standard extensions, namely,

1. `'IMAGE'` extensions,
2. `'TABLE'` ASCII table extensions, and
3. `'BINTABLE'` binary table extensions

are given in the following sections. A list of other conforming extensions is given in Appendix F

7.1 Image Extension

~~Data shall appear as an image extension if the primary header of the FITS file has the keyword EXTEND set to T and the first keyword of that extension header has~~ The *FITS* image extension is nearly identical in structure to the the primary HDU and is used to store an array of data. Multiple image extensions can be used to store any number of arrays in a single *FITS* file. The first keyword in an image extension *shall* be `XTENSION=␣'IMAGE␣␣␣'`.

7.1.1 Mandatory Keywords

The `XTENSION` keyword is required to be the first keyword of all image extensions. The keyword records ~~card images~~ in the header of an image extension *must* use the keywords defined in Table 7.1 in the order specified. No other keywords *may* intervene between the `XTENSION` and `GCOUNT` keywords.

Table 7.1. Mandatory keywords in image extensions.

#	Keyword
1	`XTENSION=␣'IMAGE␣␣␣'`
2	`BITPIX`
3	`NAXIS`
4	`NAXISn`, `n` = 1, ..., `NAXIS`
5	`PCOUNT` = 0
6	`GCOUNT` = 1
	⋮
	(other keywords ...)
	⋮
last	`END`

`XTENSION` Keyword The value field *shall* contain the character string ~~value text~~ `'IMAGE␣␣␣'`.

`BITPIX` Keyword The value field *shall* contain an integer. The absolute value is used in computing the sizes of data structures. It *shall* specify the number of bits that represent a data value. The only valid values of `BITPIX` are given in Table 4.6. Writers of `IMAGE` extensions *should* select a `BITPIX` data type appropriate to the form, range of values, and accuracy of the data in the array.

`NAXIS` Keyword The value field *shall* contain a non-negative integer no greater than 999, representing the number of axes in the associated data array. If the value is zero then the image extension *shall* not have any data blocks following the header~~A value of zero signifies that no data follow the header in the image extension~~.

`NAXISn` Keywords The NAXISn keywords *must* be present for all values n = 1, ..., NAXIS, in increasing order of n, and for no other values of n. The value field of this indexed keyword *shall* contain a non-negative integer, representing the number of elements along axis `n` of a data array. The `NAXISn` keywords *must* be present for all values `n = 1, ..., NAXIS`, and for no other values of `n`. If the value of ~~A value of zero for~~ any of the `NAXISn` keywords is zero, then the image extension *shall* not have any data blocks following the header~~signifies that no data follow the header in the image extension~~. If `NAXIS` is equal to 0, there *should not* be any `NAXISn` keywords.

`PCOUNT` **Keyword** The value field *shall* contain the integer 0.

`GCOUNT` **Keyword** The value field *shall* contain the integer 1; each image extension contains a single array.

`END` **Keyword** This keyword has no associated value. Bytes ~~Columns~~ 9 through 80 *shall* be filled with ASCII spaces~~blanks~~.

7.1.2 Other Reserved Keywords

The reserved keywords defined in §4.4.2 (except for `EXTEND` and `BLOCKED`) *may* appear in an image extension header. The keywords *must* be used as defined in that section.

~~Units The units of all header keyword values in an image extension shall follow the prescriptions in §4.3.~~.

7.1.3 Data Sequence

The data format *shall* be identical to that of a primary data array as described in §3.3.2.

Table 7.2. Mandatory keywords in ASCII table extensions.

#	Keyword
1	`XTENSION=␣'TABLE␣␣␣'`
2	`BITPIX = 8`
3	`NAXIS = 2`
4	`NAXIS1`
5	`NAXIS2`
6	`PCOUNT = 0`
7	`GCOUNT = 1`
8	`TFIELDS`
	⋮
	(other keywords, ~~which *must* include~~ including (if `TFIELDS` is not zero) . . .)
	`TTYPEn`, n=1, 2, . . . , k where k is the value of `TFIELDS` (*Recommended*)
	`TBCOLn`, n=1, 2, . . . , k where k is the value of `TFIELDS` (*Required*)
	`TFORMn`, n=1, 2, . . . , k where k is the value of `TFIELDS` (*Required*)
	⋮
last	`END`

7.2 The ASCII Table Extension

~~Data shall appear as an ASCII table extension if the primary header of the FITS file has the keyword `EXTEND` set to `T` and the first keyword of that extension header has~~ The ASCII table extension provides a means of storing catalogs and tables of astronomical data in *FITS* format. Each row of the table consists of a fixed-length sequence of ASCII characters divided into fields that correspond to the columns in the table. The first keyword in an ASCII table extension *shall* be `XTENSION=␣'TABLE␣␣␣'`.

7.2.1 Mandatory Keywords

The header of an ASCII table extension *must* use the keywords defined in Table 7.2. The first keyword *must* be `XTENSION`; the seven keywords following `XTENSION` (`BITPIX` . . . `TFIELDS`) *must* be in the order specified with no intervening keywords.

`XTENSION` Keyword The value field *shall* contain the character string value text `'TABLE␣␣␣'`.

`BITPIX` **Keyword** The value field *shall* contain the integer 8, denoting that the array contains ASCII characters.

`NAXIS` **Keyword** The value field *shall* contain the integer 2, denoting that the included data array is two-dimensional: rows and columns.

`NAXIS1` **Keyword** The value field *shall* contain a non-negative integer, giving the number of ASCII characters in each row of the table. This includes all the characters in the defined fields plus any characters that are not included in any field.

`NAXIS2` **Keyword** The value field *shall* contain a non-negative integer, giving the number of rows in the table.

`PCOUNT` **Keyword** The value field *shall* contain the integer 0.

`GCOUNT` **Keyword** The value field *shall* contain the integer 1; the data blocks ~~records~~ contain a single table.

`TFIELDS` **Keyword** The value field *shall* contain a non-negative integer representing the number of fields in each row. The maximum permissible value is 999.

`TBCOLn` **Keywords** The value field of this indexed keyword *shall* contain an integer specifying the column in which field `n` starts. The first column of a row is numbered 1.

`TFORMn` **Keywords** The value field of this indexed keyword *shall* contain a character string describing the format in which field `n` is encoded. Only the formats in Table 7.3, interpreted as Fortran [17] ~~ANSI FORTRAN-77 13~~input formats and discussed in more detail in §7.2.5, are permitted for encoding. Format codes *must* be specified in upper case. Other format editing codes common to Fortran ~~ANSI FORTRAN-77~~ such as repetition, positional editing, scaling, and field termination are not permitted. All values in numeric fields have a number base of ten (i.e., they are decimal); binary, octal, hexadecimal, and other representations are not permitted. The `TDISPn` keyword, defined in §7.2.2, may be used to *recommend* that a decimal integer value in an ASCII table be displayed as the equivalent binary, octal, or hexadecimal value.

`END` **Keyword** This keyword has no associated value. Bytes ~~Columns~~ 9 through 80 *shall* contain ASCII ~~blanks~~ spaces (decimal 32 or hexadecimal 20).

Table 7.3. Valid `TFORMn` format values in `TABLE` extensions.

Field Value	Data Type
`Aw`	Character
`Iw`	Decimal integer
`Fw.d`	Floating-point, fixed decimal notation ~~Single precision real~~
`Ew.d`	Floating-point~~Single precision real~~, exponential notation
`Dw.d`	Floating-point~~Double precision real~~, exponential notation

Note. — `w` is the width in characters of the field and `d` is the number of digits to the right of the decimal.

7.2.2 Other Reserved Keywords

In addition to the reserved keywords defined in §4.4.2 (except for `EXTEND` and `BLOCKED`), the following other reserved ~~addition to the mandatory keywords defined in §7.2.1, the following~~ keywords *may* be used to describe the structure of an ASCII table data array. They are *optional*, but if they appear within an ASCII table extension header, they *must* be used as defined in this section of this standard.

`TTYPEn` Keywords The value field for this indexed keyword *shall* contain a character string ~~,~~ giving the name of field `n`. It is *strongly recommended* that ~~only letters, digits, and underscore (hexadecimal code 5F), be used in the name.~~ every field of the table be assigned a unique, case-insensitive name with this keyword, and it is *recommended* that the character string be composed only of upper and lower case letters, digits, and the underscore (‘_’, decimal 95, hexadecimal 5F) character. Use of other characters is *not recommended* because it may be difficult to map the column names into variables in some languages (e.g., any hyphens, ‘*’, or ‘+’ characters in the name may be confused with mathematical operators). String comparisons with the ~~values of~~ `TTYPEn` keyword values ~~keywords~~ *should not* be case sensitive (e.g., ’TIME’ and ’Time’ *should* be interpreted as the same name). ~~The use of identical names for different fields should be avoided.~~

`TUNITn` Keywords The value field *shall* contain a character string describing the physical units in which the quantity in field `n`, after any application of `TSCALn` and `TZEROn`, is expressed. Units *must* follow the prescriptions in §4.3.

`TSCALn` **Keywords** This indexed keyword *shall* be used, along with the `TZEROn` keyword, to linearly scale the values in the table ~~when the quantity in~~ field `n` to transform them into the physical values that they represent using Eq. 7.1. ~~does not represent a true physical quantity.~~The value field *shall* contain a floating-point number representing the coefficient of the linear term in the scaling equation~~Eq.7.1, which *must* be used to compute the true physical valueof the field~~. The default value for this keyword is 1.0. This keyword *must* ~~*may*~~ *not* be used for A-format fields.

The transformation equation used to compute a true physical value from the quantity in field `n` is

$$\text{physical\_value} \quad = \quad \texttt{TZEROn} + \texttt{TSCALn} \times \text{field\_value}. \tag{7.1}$$

where `field_value` is the value that is actually stored in that table field in the *FITS* file

`TZEROn` **Keywords** This indexed keyword *shall* be used, along with the `TSCALn` keyword, to linearly scale the values in the table ~~when the quantity in~~ field `n` to transform them into the physical values that they represent using Eq. 7.1. ~~does not represent a true physical quantity.~~The value field *shall* contain a floating-point number representing the physical value corresponding to an array value of zero~~zero point for the true physical value of field n~~. The default value for this keyword is 0.0. This keyword *must* ~~*may*~~ *not* be used for A-format fields.

`TNULLn` **Keywords** The value field for this indexed keyword *shall* contain the character string that represents an undefined value for field `n`. The string is implicitly space ~~blank~~ filled to the width of the field.

`TDISPn` **Keywords** The value field of this indexed keyword *shall* contain a character string describing the format recommended for displaying an ASCII text representation of of the contents of field `n`. This keyword overrides the default display format given by the `TFORMn` keyword. If the table value has been scaled, the physical value, derived using Eq. 7.1, *shall* be displayed. All elements in a field *shall* be displayed with a single, repeated format. Only the format codes in Table 7.4, interpreted as Fortran [17] output formats, and discussed in more detail in §7.3.4, are permitted for encoding. The format codes *must* be specified in upper case. If the `Bw.m`, `Ow.m`, and `Zw.m` formats are not readily available to the reader, the `Iw.m` display format *may* be used instead, and if the `ENw.d` and `ESw.d` formats are not available, `Ew.d` *may* be used.

7.2.3 Data Sequence

The table is constructed from a two-dimensional array of ASCII characters. The row length and the number of rows *shall* be those specified, respectively, by the `NAXIS1` and

Table 7.4. Valid `TDISPn` format values in `TABLE` extensions

Field Value	Data Type
`Aw`	Character
`Iw.m`	Integer
`Bw.m`	Binary, integers only
`Ow.m`	Octal, integers only
`Zw.m`	Hexadecimal, integers only
`Fw.d`	Floating-point, fixed decimal notation
`Ew.dEe`	Floating-point, exponential notation
`ENw.d`	Engineering; E format with exponent multiple of 3
`ESw.d`	Scientific; same as EN but nonzero leading digit if not zero
`Gw.dEe`	General; appears as F if significance not lost, else E.
`Dw.dEe`	Floating-point, exponential notation

Note. — `w` is the width in characters of displayed values, `m` is the minimum number of digits displayed, `d` is the number of digits to right of decimal, and `e` is number of digits in exponent. The `.m` and `Ee` fields are *optional*.

`NAXIS2` keywords of the associated header~~records~~. The number of characters in a row and the number of rows in the table *shall* determine the size of the character array. Every row in the array *shall* have the same number of characters. The first character of the first row *shall* be at the start of the data block ~~record~~ immediately following the last header block~~record~~. The first character of subsequent rows *shall* follow immediately the character at the end of the previous row, independent of the *FITS* block ~~record~~ structure. The positions in the last data block ~~record~~ after the last character of the last row of the table ~~data array~~ *shall* be filled with ASCII spaces~~blanks~~.

7.2.4 Fields

Each row in the array *shall* consist of a sequence of from 0 to 999 fields, as specified by the `TFIELDS` keyword, with one entry in each field. For every field, the Fortran [17] format of the information contained (given by the `TFORMn` keyword), the ~~ANSI FORTRAN-77 format of the information contained ,~~ location in the row of the beginning of the field (given by the `TBCOLn` keyword), and (*optionally*, but *strongly* recommended) the field name (given by the `TTYPEn` keyword), *shall* be specified in the associated header. ~~keywords of the associated header records. A separate format keyword *must* be provided for each field.~~ The location and format of fields *shall* be the same for every row. Fields *may* overlap , but this usage is *not recommended*. Only a limited set of ASCII character values *may* appear within any field, depending on the field type as specified below. There *may* be characters in a table row that are not included in any field, (e.g., between fields, or before the first field or after the last field). Any 7-bit ASCII character *may* occur in characters of a table row that are not included in a defined field. A common convention is to include a space character between each field for added legibility if the table row is displayed verbatim. It is also permissible to add control characters, such as a carriage return or line feed character, following the last field in each row as a way of formating the table if it is printed or displayed by a text editing program.

7.2.5 Entries

All data in an ASCII table extension field *shall* be ASCII text in a format that conforms to the rules for fixed field input in Fortran [17] ~~ANSI FORTRAN-7713~~format, as described below~~, including implicit decimal points~~. The only possible formats *shall* be those specified in Table 7.3. If values of −0 and +0 need to ~~must~~ be distinguished, then the sign character *should* appear in a separate field in character format. `TNULLn` keywords *may* be used to specify a character string that represents an undefined value in each field. The characters representing an undefined value *may* differ from field to field but *must* be the same within a field. Writers of ASCII tables *should* select a format for each field that is appropriate to the form, range of values, and accuracy of the data in ~~the table.~~ that field. This standard does not impose an upper limit on the number of digits of precision, nor any limit on the range of numeric values. Software packages that read or write data according to this standard could be limited, however, in the range of values and exponents that are supported (e.g., to the range that can be represented by 32-bit or 64-bit binary numbers).

The value of each entry *shall* be interpreted as described in the following paragraphs.

Character fields The value of a character-formatted (`Aw`) field is a character string of width w containing the characters in columns `TBCOLn` through `TBCOLn`+`w` $-$ 1. The character string *shall* be composed of the restricted set of ASCII text characters with decimal values in the range 32 through 126 (hexadecimal 20 through 7E).

Integer fields The value of an integer-formatted (`Iw`) field is a signed decimal integer contained in ~~an integer number determined by removing all blanks from~~ columns `TBCOLn` through `TBCOLn`+`w` $-$ 1 consisting of a single *optional* sign (`'+'` or `'-'`) followed by one or more decimal digits ('0' through '9'). ~~and interpreting the remaining, right-justified characters as a signed decimal integer.~~ Non-significant space characters may precede and/or follow the integer value within the field. A blank field has value 0. All characters other than leading and trailing spaces, a contiguous string of decimal digits, ~~blanks, the decimal integers '0' through '9'~~ and a single leading sign character ~~('+' and '-')~~ are forbidden.

Real fields The value of a real-formatted field (`Fw.d`, `Ew.d`, `Dw.d`) is a real number determined from the `w` characters from columns `TBCOLn` through `TBCOLn`+`w` $-$ 1. The value is formed by

1. discarding any trailing space characters in the field ~~all blank characters~~ and right-justifying the remaining ~~non-blank~~ characters,

2. interpreting the first non-space ~~non-blank~~ characters as a numeric string consisting of a single *optional* sign (`'+'` or `'-'`) followed by one or more decimal digits (`'0'` through `'9'`) *optionally* containing a single decimal point (`'.'`). The numeric string is terminated by the end of the right-justified field or by the occurrence of any character other than a decimal point (`'.'`) and the decimal integers (`'0'` through `'9'`). If the string contains no explicit decimal point, then the implicit decimal point is taken as immediately preceding the rightmost d digits of the string, with leading zeros assumed if necessary. The use of implicit decimal points is *deprecated* and is strongly discourged because of the possibility that FITS reading programs will misinterpret the data values. Therefore, real-formatted fields *should* always contain an explicit decimal point.

3. If ~~if~~ the numeric string is terminated by a

(a) '+' or '-', interpreting the following string as an exponent in the form of a signed decimal integer, or

(b) 'E', or 'D', interpreting the following string as an exponent of the form `E` or `D` followed by an *optionally* signed decimal integer constant.

4. The exponent string, if present, is terminated by the end of the right-justified string.

5. Characters other than those specified above, including embedded space characters, are forbidden.

The numeric value of the table field is then the value of the numeric string multiplied by ten (10) to the power of the exponent string, i.e., value = numeric_string $\times$ $10^{(\text{exponent\_string})}$. The default exponent is zero and a blank field ~~blankfield~~ has value zero. There is no difference between the `F`, `D`, and `E` formats; the content of the string determines its interpretation. Numbers requiring more precision and/or range than the local computer can support *may* be represented. It is good form to specify a `D` format in `TFORM`*n* for a column of an ASCII table when that column will contain numbers that cannot be accurately represented in 32-bit IEEE binary format (see Appendix E).

~~Note that the above definitions allow for embedded blanks anywhere in integer-formatted and real-formatted fields and implicit decimal points in real-formatted fields. *FITS* reading tasks will have to honor these flexibilities. However, since these flexibilities are likely to cause confusion and possible misinterpretation, it is *recommended* that *FITS* writing tasks write tables with explicit decimal points and no embedded or trailing blanks whenever possible.~~

Table 7.5. Mandatory keywords in binary table extensions.

#	Keyword
1	`XTENSION=␣'BINTABLE'`
2	`BITPIX = 8`
3	`NAXIS = 2`
4	`NAXIS1`
5	`NAXIS2`
6	`PCOUNT`
7	`GCOUNT = 1`
8	`TFIELDS`
	⋮
	(other keywords, ~~which *must* include~~including (if `TFIELDS` is not zero) . . .)
	`TTYPEn`, n=1, 2, . . . , k where k is the value of `TFIELDS` (*Recommended*)
	TFORMn, n=1, 2, . . . , k where k is the value of `TFIELDS` (*Required*)
	⋮
last	`END`

7.3 Binary Table Extension

~~Data shall appear as a binary table extension if the primary header of the FITS file has the keyword `EXTEND` set to `T` and the first keyword of that extension header has~~ The binary table extension is similar to the ASCII table in that it provides a means of storing catalogs and tables of astronomical data in *FITS* format, however, it offers more features and provides more efficient data storage than ASCII tables. The numerical values in binary tables are stored in more compact binary formats rather than coded into ASCII, and each field of a binary table can contain an array of values rather than a simple scalar as in ASCII tables. The first keyword in a binary table extension *shall* be `XTENSION= 'BINTABLE'`

7.3.1 Mandatory Keywords

The `XTENSION` keyword is the first keyword of all binary table extensions. The seven keywords following (`BITPIX` . . . `TFIELDS`) *must* be in the order specified in Table 7.5, with no intervening keywords.

`XTENSION` Keyword The value field *shall* contain the character string `'BINTABLE'`.

`BITPIX` **Keyword** The value field *shall* contain the integer 8, denoting that the array is an array of 8-bit bytes.

`NAXIS` **Keyword** The value field *shall* contain the integer 2, denoting that the included data array is two-dimensional: rows and columns.

`NAXIS1` **Keyword** The value field *shall* contain a non-negative integer, giving the number of 8-bit bytes in each row of the table.

`NAXIS2` **Keyword** The value field *shall* contain a non-negative integer, giving the number of rows in the table.

`PCOUNT` **Keyword** The value field *shall* contain the number of bytes that follow the table in the supplemental data area called the heap. ~~associated extension data .~~

`GCOUNT` **Keyword** The value field *shall* contain the integer 1; the data blocks ~~records~~ contain a single table.

`TFIELDS` **Keyword** The value field *shall* contain a non-negative integer representing the number of fields in each row. The maximum permissible value is 999.

`TFORMn` **Keywords** The value field of this indexed keyword *shall* contain a character string of the form r`T`a. The repeat count r is the ASCII representation of a non-negative integer specifying the number of elements in field `n`. The default value of r is 1; the repeat count need not be present if it has the default value. A zero element count, indicating an empty field, is permitted. The data type `T` specifies the data type of the contents of field `n`. Only the data types in Table 7.6 are permitted. The format codes *must* be specified in upper case. For fields of type `P` or `Q`, the only permitted repeat counts are 0 and 1. The additional characters a are *optional* and are not further defined in this standard. Table 7.6 lists the number of bytes each data type occupies in a table row. The first field of a row is numbered 1. The total number of bytes n_{row} in a table row is given by

$$n_{\mathrm{row}} = \sum_{i=1}^{\mathtt{TFIELDS}} r_i b_i \tag{7.2}$$

where r_i is the repeat count for field i, b_i is the number of bytes for the data type in field i, and `TFIELDS` is the value of that keyword, *must* equal the value of `NAXIS1`.

Table 7.6. Valid TFORMn data types in BINTABLE extensions.

TFORMn value	Description	8-bit Bytes
L	Logical	1
X	Bit	†
B	Unsigned byte	1
I	16-bit integer	2
J	32-bit integer	4
K	64-bit integer	8
A	Character	1
E	Single precision floating-point	4
D	Double precision floating-point	8
C	Single precision complex	8
M	Double precision complex	16
P	Array Descriptor (32-bit)	8
Q	Array Descriptor (64-bit)	16

†number of 8-bit bytes needed to contain all bits.

`END` **Keyword** This keyword has no associated value. Bytes ~~Columns~~ 9 through 80 *shall* contain ASCII spaces~~blanks~~.

7.3.2 Other Reserved Keywords

In addition to the reserved keywords defined in §4.4.2 (except for `EXTEND` and `BLOCKED`), the following other reserved ~~mandatory keywords defined in §7.3.1, these~~ keywords *may* be used to describe the structure of a binary table data array. They are *optional*, but if they appear within a binary table extension header, they *must* be used as defined in this section of this standard.

`TTYPEn` **Keywords** The value field for this indexed keyword *shall* contain a character string ~~,~~ giving the name of field `n`. It is *strongly recommended* that ~~only letters, digits, and underscore (hexadecimal code 5F) be used in the name.~~ every field of the table be assigned a unique, case-insensitive name with this keyword, and it is *recommended* that the character string be composed only of upper and lower case letters, digits, and the underscore ('`_`', decimal 95, hexadecimal 5F) character. Use of other characters is *not recommended* because it may be difficult to map the column names into variables in some languages (e.g., any hyphens, '*', or '+' characters in the name may be confused with mathematical operators). String comparisons with the ~~values of~~ `TTYPEn` keyword values ~~keywords~~ *should not* be case sensitive (e.g., 'TIME' and 'Time' *should* be interpreted as the same name). ~~The use of identical names for different fields should be avoided.~~

`TUNITn` **Keywords** The value field *shall* contain a character string describing the physical units in which the quantity in field `n`, after any application of `TSCALn` and `TZEROn`, is expressed. Units *must* follow the prescriptions in §4.3.

`TSCALn` **Keywords** This indexed keyword *shall* be used, along with the `TZEROn` keyword, to linearly scale the values in the table ~~when the quantity in~~ field `n` to transform them into the physical values that they represent using Eq. 7.1. ~~does not represent a true physical quantity.~~ It *must* ~~*may*~~ *not* be used if the format of field `n` is `A`, `L`, or `X`. ~~The interpretation for fields of type `P` or `Q` is defined in §7.3.5.~~ For fields with all other data types, the value field *shall* contain a floating-point number representing the coefficient of the linear term in Eq. 7.1, which is used to compute the true physical value of the field, or, in the case of the complex data types C and M, of the real part of the field, with the imaginary part of the scaling factor set to zero. The default value for this keyword is 1.0. For fields of type `P` or `Q`, the values of `TSCALn` and `TZEROn` are to be applied to the values in the data array in the heap area, not the values of the array descriptor (see §7.3.5).

Table 7.7. Usage of `TZEROn` to represent non-default integer data types.

`TFORMn`	Native Data Type	Physical Data Type	TZEROn	
'B'	unsigned	signed byte	-128	(-2^7)
'I'	signed	unsigned 16-bit	32768	(2^{15})
'J'	signed	unsigned 32-bit	2147483648	(2^{31})
'K'	signed	unsigned 64-bit	9223372036854775808	(2^{63})

`TZEROn` **Keywords** This indexed keyword *shall* be used, along with the `TSCALn` keyword, to linearly scale the values in the table ~~when the quantity in~~ field `n` to transform them into the physical values that they represent using Eq. 7.1. ~~does not represent a true physical quantity.~~ It *must* ~~*may*~~ *not* be used if the format of field `n` is `A`, `L`, or `X`. ~~The interpretation for fields of type `P` or `Q` is defined in §7.3.5.~~ For fields with all other data types, the value field *shall* contain a floating-point number representing the true physical value corresponding to a value of zero in field `n` of the *FITS* file, or, in the case of the complex data types `C` and `M`, in the real part of the field, with the imaginary part set to zero. The default value for this keyword is 0.0. Equation 7.1 is used to compute a true physical value from the quantity in field `n`. For fields of type `P` or `Q`, the values of `TSCALn` and `TZEROn` are to be applied to the values in the data array in the heap area, not the values of the array descriptor (see §7.3.5).

In addition to its use in representing floating point values as scaled integers, the `TZEROn` keyword is also used when storing unsigned integer values in the field. In this special case the `TSCALn` keyword *shall* have the default value of 1.0 and the `TZEROn` keyword *shall* have one of the integer values shown in Table 7.7

Since the binary table format does not support a native unsigned integer data type (except for the unsigned 8-bit 'B' column type), the unsigned values are stored in the field as native signed integers with the appropriate integer offset specified by the `TZEROn` keyword value shown in the table. For the byte column type, the converse technique can be used to store signed byte values as native unsigned values with the negative `TZEROn` offset. In each case, the physical value is computed by adding the offset specified by the `TZEROn` keyword to the native data type value that is stored in the table field.

`TNULLn` **Keywords** The value field for this indexed keyword *shall* contain the integer that represents an undefined value for field `n` of data type `B`, `I`, `J` or `K`, or `P` or `Q` array descriptor fields (§7.3.5) that point to `B`, `I`, `J` or `K` integer arrays. The keyword *must* ~~*may*~~ *not* be used if field `n` is of any other data type. The value of this keyword corresponds to the table column values before applying any transformation indicated by the TSCALn and TZEROn keywords.

If the TSCALn and TZEROn keywords do not have the default values of 1.0 and 0.0, respectively, then the value of the TNULLn keyword *must* equal the actual value in the *FITS* file that is used to represent an undefined element and not the corresponding physical value (computed from Eq. 7.1). To cite a specific, common example, *unsigned* 16-bit integers are represented in a *signed* integer column (with TFORMn = 'I') by setting TZEROn = 32768 and TSCALn = 1. If it is desired to use elements that have an *unsigned* value (i.e., the physical value) equal to 0 to represent undefined elements in the field, then the TNULLn keyword *must* be set to the value -32768 because that is the actual value stored in the *FITS* file for those elements in the field.

TDISPn Keywords The value field of this indexed keyword *shall* contain a character string describing the format recommended for ~~display~~ displaying an ASCII text representation of the contents of field n. If the table value has been scaled, the physical value, derived using Eq. 7.1, *shall* be displayed. All elements in a field *shall* be displayed with a single, repeated format. For purposes of display, each byte of bit (type X) and byte (type B) arrays is treated as ~~a~~ an unsigned integer. Arrays of type A *may* be terminated with a zero byte. Only the format codes in Table 7.8, interpreted as Fortran [17] output formats and discussed in more detailin §7.3.4, are permitted for encoding. The format codes *must* be specified in upper case. If the Bw.m, Ow.m, and Zw.m formats are not readily available to the reader, the Iw.m display format *may* be used instead, and if the ENw.d and ESw.d formats are not available, Ew.d *may* be used. In the case of fields of type P or Q, the TDISPn value applies to the data array pointed to by the array descriptor (§7.3.5), not the values in the array descriptor itself.

THEAP Keyword The value field of this keyword *shall* contain an integer providing the separation, in bytes, between the start of the main data table and the start of a supplemental data area called the heap. The default value, which is also the minimum allowed value, *shall* be the product of the values of NAXIS1 and NAXIS2. This keyword *shall not* be used if the value of PCOUNT is zero. The use of this keyword is described in in §7.3.5.

TDIMn Keywords The value field of this indexed keyword *shall* contain a character string describing how to interpret the contents of field n as a multi-dimensional array with a format of '$(l,m,n\ldots)$' where l, m, n, ... are the dimensions of the array. The data are 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 total number of elements in the array equals the product of the dimensions specified in the TDIMn keyword. The size *must* be less than or equal to the repeat count on the

Table 7.8. Valid `TDISPn` format values in `BINTABLE` extensions

Field Value	Data Type
`Aw`	Character
`Lw`	Logical
`Iw.m`	Integer
`Bw.m`	Binary, integers only
`Ow.m`	Octal, integers only
`Zw.m`	Hexadecimal, integers only
`Fw.d`	Floating-point, fixed decimal notation ~~Single precision real~~
`Ew.dEe`	Floating-point~~Single precision real~~, exponential notation
`ENw.d`	Engineering; E format with exponent multiple of 3
`ESw.d`	Scientific; same as EN but nonzero leading digit if not zero
`Gw.dEe`	General; appears as F if significance not lost, else E.
`Dw.dEe`	Floating-point~~Double precision real~~, exponential notation

Note. — `w` is the width in characters of displayed values, `m` is the minimum number of digits displayed, `d` is the number of digits to right of decimal, and `e` is number of digits in exponent. The `.m` and `Ee` fields are *optional*.

TFORMn keyword, or, in the case of columns that have a 'P' or 'Q' TFORMn data type, less than or equal to the array length specified in the variable-length array descriptor (see §7.3.5). In the special case where the variable-length array descriptor has a size of zero, then the TDIMn keyword is not applicable. If the number of elements in the array implied by the TDIMn is less than the allocated size of the array in the *FITS* file, then the unused trailing elements *should* be interpreted as containing undefined fill values.

A character string is represented in a binary table by a one-dimensional character array, as described under 'Character' in the list of data types in §7.3.3.1~~("Main Data Table")~~. For example, a Fortran ~~77~~ CHARACTER*20 variable could be represented in a binary table as a character array declared as TFORMn = '20A'. Arrays of~~character~~ strings, i.e., multi-dimensional 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.

7.3.3 Data Sequence

The data in a binary table extension *shall* consist of a main data table ~~Main Data Table~~ which *may*, but need not, be followed by additional bytes in the supplemental data area. The positions in the last data block ~~record~~ after the last additional byte, or, if there are no additional bytes, the last character of the last row of the main data table ~~data array~~, *shall* be filled by setting all bits to zero.

7.3.3.1 Main Data Table

The table is constructed from a two-dimensional byte array. The number of bytes in a row *shall* be specified by the value of the NAXIS1 keyword and the number of rows *shall* be specified by the NAXIS2 keyword of the associated header ~~records~~. Within a row, fields *shall* be stored in order of increasing column number, as determined from the n of the TFORMn keywords. The number of bytes in a row and the number of rows in the table *shall* determine the size of the byte array. Every row in the array *shall* have the same number of bytes. The first row *shall* begin at the start of the data block ~~record~~ immediately following the last header block ~~record~~. Subsequent rows *shall* begin immediately following the end of the previous row, with no intervening bytes, independent of the *FITS* block ~~record~~ structure. Words need not be aligned along word boundaries.

Each row in the array *shall* consist of a sequence of from 0 to 999 fields as specified by the TFIELDS keyword~~fields~~. The number of elements in each field and their data type *shall* be specified by the TFORMn keyword in the associated header~~in keywords of the associated header records~~. A separate format keyword *must* be provided for each field. The location and format of fields *shall* be the same for every row. Fields *may* be empty, if the repeat count specified in the value of the TFORMn keyword of the header is 0. Writers of binary tables *should* select a format appropriate to the form, range of values, and accuracy of the data in the table. The following data types, and no others, are permitted.

Logical If the value of the `TFORMn` keyword specifies data type `L`, the contents of field `n` *shall* consist of ASCII `T` indicating true or ASCII `F`, indicating false. A 0 byte (hexadecimal 00) indicates a null ~~0) indicates an invalid~~ value.

Bit Array If the value of the `TFORMn` keyword specifies data type `X`, the contents of field `n` *shall* consist of a sequence of bits starting with the most significant bit; the bits following *shall* be in order of decreasing significance, ending with the least significant bit. A bit array *shall* be composed of an integral number of bytes, with those bits following the end of the data set to zero. No null value is defined for bit arrays.

Character If the value of the `TFORMn` keyword specifies data type `A`, field `n` *shall* contain a character string of zero or more members, composed of the restricted set of ASCII text characters~~ASCII text~~. This character string *may* be terminated before the length specified by the repeat count by an ASCII NULL (hexadecimal code 00). Characters after the first ASCII NULL are not defined. A string with the number of characters specified by the repeat count is not NULL terminated. Null strings are defined by the presence of an ASCII NULL as the first character.

Unsigned 8-Bit Integer If the value of the `TFORMn` keyword specifies data type `B`, the data in field `n` *shall* consist of unsigned 8-bit integers, with the most significant bit first, and subsequent bits in order of decreasing significance. Null values are given by the value of the associated `TNULLn` keyword. Signed integers can be represented using the convention described in § 5.2.5.

16-Bit Integer If the value of the `TFORMn` keyword specifies data type `I`, the data in field `n` *shall* consist of two's complement signed 16-bit integers, contained in two bytes. The most significant byte *shall* be first (big endian byte order). Within each byte the most significant bit *shall* be first, and subsequent bits *shall* be in order of decreasing significance. Null values are given by the value of the associated `TNULLn` keyword. Unsigned integers can be represented using the convention described in § 5.2.5.

32-Bit Integer If the value of the `TFORMn` keyword specifies data type `J`, the data in field `n` *shall* consist of two's complement signed 32-bit integers, contained in four bytes. The most significant byte *shall* be first, and subsequent bytes *shall* be in order of decreasing significance (big endian byte order). Within each byte, the most significant bit *shall* be first, and subsequent bits *shall* be in order of decreasing significance. Null values are given by the value of the associated `TNULLn` keyword. Unsigned integers can be represented using the convention described in § 5.2.5.

64-Bit Integer If the value of the `TFORMn` keyword specifies data type `K`, the data in field `n` *shall* consist of two's complement signed 64-bit integers, contained in eight bytes. The most significant byte *shall* be first, and subsequent bytes *shall* be in order of decreasing significance. Within each byte, the most significant bit *shall* be first, and subsequent bits *shall* be in order of decreasing significance (big endian byte order). Null values are given by the value of the associated `TNULLn` keyword. Unsigned integers can be represented using the convention described in § 5.2.5.

Single Precision Floating-Point If the value of the `TFORMn` keyword specifies data type `E`, the data in field `n` *shall* consist of ANSI/IEEE-754 [21] ~~15~~32-bit floating-point numbers, in big endian byte order, as described in Appendix E. All IEEE special values are recognized. The IEEE NaN is used to represent null ~~invalid~~ values.

Double Precision Floating-Point If the value of the `TFORMn` keyword specifies data type `D`, the data in field `n` *shall* consist of ANSI/IEEE-754 [21] ~~15~~64-bit double precision floating-point numbers, in big endian byte order, as described in Appendix E. All IEEE special values are recognized. The IEEE NaN is used to represent null ~~invalid~~ values.

Single Precision Complex If the value of the `TFORMn` keyword specifies data type `C`, the data in field `n` *shall* consist of a sequence of pairs of 32-bit single precision floating-point numbers. The first member of each pair *shall* represent the real part of a complex number, and the second member *shall* represent the imaginary part of that complex number. If either member contains an IEEE ~~a~~ NaN, the entire complex value is null~~invalid~~.

Double Precision Complex If the value of the `TFORMn` keyword specifies data type `M`, the data in field `n` *shall* consist of a sequence of pairs of 64-bit double precision floating-point numbers. The first member of each pair *shall* represent the real part of a complex number, and the second member of the pair *shall* represent the imaginary part of that complex number. If either member contains an IEEE ~~a~~ NaN, the entire complex value is null~~invalid~~.

Array Descriptor The repeat count on the `P` and `Q` array descriptor fields must either have a value of 0 (denoting an empty field of zero bytes) or 1. If the value of the `TFORMn` keyword specifies data type `1P`, the data in field `n` *shall* consist of ~~not more than~~ one pair of 32-bit integers. If the value of the `TFORMn` keyword specifies data type `1Q`, the data in field `n` *shall* consist of ~~not more than~~ one pair of 64-bit integers. The meaning of these integers is defined in §7.3.5.

7.3.3.2 Bytes Following Main Table

The main data table *may*~~*shall*~~ be followed by a supplemental data area called the heap. The size of the supplemental data area, in bytes, is ~~an additional data area containing zero or more bytes, as~~ specified by the value of the `PCOUNT` keyword. The use of ~~One use for~~ this data area is described in §7.3.5. ~~This does not preclude other uses for these bytes.~~

7.3.4 Data Display

The indexed `TDISPn` keyword *may* be used to describe the recommended format for the displaying an ASCII text representation of the contents of field `n`. The permitted display format codes for each type of data (i.e., character strings, logical, integer, or real) are given in Table 7.8 and described below.

Character data If the table column contains a character string (with `TFORMn` = 'rA') then the `TFORMn` ~~Character data are encoded under~~ format code *must* be `'Aw'` where w is the number of characters to display. If the character datum has length less than or equal to `w`, it is represented on output right-justified in a string of `w` characters. If the character datum has length greater than `w`, the first `w` characters of the datum are represented on output in a string of `w` characters. Character data are not surrounded by single or double quotation marks unless those marks are themselves part of the data value.

Logical data If the table column contains logical data (with `TFORMn` = 'rL') then the `TFORMn` ~~Logical data are encoded under~~ format code *must* be `'Lw'` where w is the width in characters of the display field. Logical data are represented on output with the character `T` for true or `F` for false right-justified in a space-filled ~~blank-filled~~ string of `w` characters. A null value *may* be represented by a ~~completely blank~~ string of `w` space characters.

Integer data If the table column contains integer data (with `TFORMn` = `'rX'`, `'rB'`, `'rI'`, `'rJ'`, or `'rK'`) then the `TFORMn` format code *may* have any of these forms: ~~Integer data (including bit `X` and byte `B` type fields) are encoded under format codes~~ `Iw.m`, `Bw.m`, `Ow.m`, or~~and~~ `Zw.m`. The default value of `m` is one and the `'.m'` ~~'~~is *optional*. The first letter of the code specifies the number base for the encoding with `I` for decimal (10), `B` for binary (2), `O` for octal (8), and `Z` for hexadecimal (16). Hexadecimal format uses the upper-case letters A through F to represent decimal values 10 through 15. The output field consists of w characters containing zero or more leading spaces ~~blanks~~ followed by a minus sign if the internal datum is negative (only in the case of decimal encoding with the I format code) followed by the magnitude of the internal datum in the form of an unsigned integer constant in the specified number base with only as many leading zeros as are needed to have at least m numeric digits. Note that $m \leq w$ is allowed if all values are positive, but $m < w$ is required if any values are negative. If the number of digits required to represent the integer datum exceeds w, then the output field consists of a string of w asterisk (*) characters.

Real data If the table column contains real data (with TFORMn = 'rE', or 'rD') or contains integer data (with any of the TFORMn format codes listed in the previous paragraph) which are recommended to be displayed as real values (i.e., especially in cases where the integer values represent scaled physical values using Eq. 7.1), then the TFORMn format code *may* have any of these forms: ~~Real data are encoded under format codes~~ Fw.d, Ew.dEe, Dw.dEe, ENw.d, or ~~and~~ ESw.d. In all cases, the output is a string of w characters including the decimal point, any sign characters, and any exponent including the exponent's indicators, signs, and values. If the number of digits required to represent the real datum exceeds w, then the output field consists of a string of w asterisk (*) characters. In all cases, d specifies the number of digits to appear to the right of the decimal point. The F format code output field consists of $w - d - 1$ characters containing zero or more leading spaces ~~blanks~~ followed by a minus sign if the internal datum is negative followed by the absolute magnitude of the internal datum in the form of an unsigned integer constant. These characters are followed by a decimal point ('.') and d characters giving the fractional part of the internal datum, rounded by the normal rules of arithmetic to d fractional digits. For the E and D format codes, an exponent is taken such that the fraction $0.1 \leq |\mathrm{datum}|/10^{\mathrm{exponent}} < 1.0$. The fraction (with appropriate sign) is output with an F format of width $w - e - 2$ characters with d characters after the decimal followed by an E or D followed by the exponent as a signed $e + 1$ character integer with leading zeros as needed. The default value of e is 2 when the Ee portion of the format code is omitted. If the exponent value will not fit in $e + 1$ characters but will fit in $e + 2$ then the E (or D) is omitted and the wider field used. If the exponent value will not fit (with a sign character) in $e + 2$ characters, then the entire w-character output field is filled with asterisks (*). The ES format code is processed in the same manner as the E format code except that the exponent is taken so that $1.0 \leq \mathrm{fraction} < 10$. The EN format code is processed in the same manner as the E format code except that the exponent is taken to be an integer multiple of 3 and so that $1.0 \leq \mathrm{fraction} < 1000.0$. All real format codes have number base 10. There is no difference between E and D format codes on input other than an implication with the latter of greater precision in the internal datum.

The Gw.dEe format code *may* be used with data of any type. For data of type integer, logical, or character, it is equivalent to Iw, Lw, or Aw, respectively. For data of type real, it is equivalent to an `F` format (with different numbers of characters after the decimal) when that format will accurately represent the value and is equivalent to an `E` format when the number (in absolute value) is either very small or very large. Specifically, for real values outside the range $0.1 - 0.5 \times 10^{-\mathtt{d}-1} \leq \text{value} < 10^{\mathtt{d}} - 0.5$, it is equivalent to `Ew.dEe`. For real values within the above range, it is equivalent to $\mathtt{Fw}'.\mathtt{d}'$ followed by $2 + \mathtt{e}$ spaces~~blanks~~, where $\mathtt{w}' = \mathtt{w} - \mathtt{e} - 2$ and $\mathtt{d}' = \mathtt{d} - k$ for $k = 0, 1, \ldots, \mathtt{d}$ if the real datum value lies in the range $10^{k-1}\,(1 - 0.5 \times 10^{-\mathtt{d}}) \leq \text{value} \leq 10^{k}\,(1 - 0.5 \times 10^{-\mathtt{d}})$.

Complex data If the table column contains complex data (with `TFORMn` = 'rC', or `'rM'`) then the *may* be displayed with ~~Complex data are encoded with~~ any of the real data formats as described above. The same format is used for the real and imaginary parts. It is *recommended* that the 2 values be separated by a comma and enclosed in parentheses with a total field width of $2\mathtt{w} + 3$.

7.3.5 Variable-Length Arrays

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 row ~~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 field~~record~~, but if the stored array length varies for different rows~~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 storing ~~it may become impossible to store~~ certain types of data in the table could become impossible.

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 that ~~which~~ does not understand variable-length arrays will still be able to read the main data table (in other words a table containing variable-length arrays conforms to the basic binary table standard). The implementation chosen is such that the rows in the main data ~~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 data 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 rows ~~records~~, and 2) the array data are not stored directly in the main data table~~table records~~. Since a field of any data type can be a static array, a field of any data type can also be a variable-length array (excluding the type `P` and `Q` variable-length array descriptors themselves, which are not a data type so much as a storage class specifier). Other established *FITS* conventions that apply to static arrays will generally apply as well to variable-length arrays.

A variable-length array is declared in the table header with one of the following 2 special field data type specifiers ~~a special field data type specifier of the form~~

$$\mathtt{rPt(e_{max})}$$

$$\text{or } \mathtt{rQt(e_{max})}$$

where the 'P' or 'Q' indicates the presence of an array descriptor (described below), the element count r *should* be 0, 1, or absent, t is a character denoting the data type of the array data (L, X, B, I, J, K, etc., but not P or Q), and $\mathtt{e_{max}}$ is a quantity guaranteed to be equal to or greater than the maximum number of elements of type t actually stored in any row of the table~~a table record~~. There is no built-in upper limit on the size of a stored array (other than the fundamental limit imposed by the range of the array descriptor, defined below); $\mathtt{e_{max}}$ 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 TFORMn keyword following the $\mathtt{e_{max}}$.

For example,

```
TFORM8 = 'PB(1800)' / Variable 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 are not stored in the main data table~~actual data records~~; they are stored in a supplemental ~~special~~ data area, the heap, following the main data table~~last fixed-size data record~~. What is stored in the main data table field ~~data record~~ is an *array descriptor*. This consists of two 32-bit signed integer values in the case of 'P' array descriptors, or two 64-bit signed integer values in the case of 'Q' array descriptors: 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. The meaning of a negative value for either of these integers is not defined by this standard. Storage for the array is contiguous. The array descriptor for field N as it would appear embedded in a table row ~~data record~~ is illustrated symbolically below:

$$\ldots \text{ [field } N{-}1] \text{ [(nelem,offset)] [field } N{+}1] \ldots$$

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 principal segments, as follows:

[table header] [main data table~~record storage area~~] (optional gap) [heap area]

The table header consists of one or more 2880-byte header blocks with the last block ~~*FITS* logical records with the last record~~ indicated by the keyword `END` somewhere in the ~~record.~~ block. The main data table begins with the first data block ~~The record storage area begins with the next 2880-byte logical record~~ following the last header block ~~record~~ and is `NAXIS1` × `NAXIS2` bytes in length. The zero indexed byte offset to the start of the heap, measured from the start of the main data table, may be ~~record storage area is~~ given by the `THEAP` keyword in the header. If this keyword is missing then the heap begins ~~the heap is assumed to begin~~ with the byte immediately following main data table (i.e., the default value of `THEAP` is `NAXIS1` × `NAXIS2`). This default value is the minimum allowed value for the `THEAP` keyword, because any smaller value would imply that the heap and the main data table overlap. If the `THEAP` keyword has a value larger than this default value, then there is a gap between the end of the main data table and the start of the heap. ~~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 supplemental data area following the main data table ~~heap area following the last stored record~~ (gap plus heap) is given by the `PCOUNT` keyword in the table header.

For example, suppose a table contains ~~we have a table containing~~ 5 rows which are each 168 bytes long, with a heap area 3000 bytes long, beginning at an offset of 2880, thereby aligning the main data table and heap areas on data block ~~record storage and heap areas on *FITS* record~~ boundaries (this alignment is not necessarily recommended but is useful for this ~~our~~ example). The data portion of the table consists of 3 2880-byte data blocks: ~~*FITS* records:~~ the first block ~~record~~ contains the 840 bytes from the 5 rows of the main data table followed by 2040 fill bytes; the heap completely fills the second block; ~~record;~~ the third block ~~record~~ contains the remaining 120 bytes of the heap followed by 2760 fill ~~filll~~ bytes. `PCOUNT` gives the total number of bytes from the end of the main data table to the end of the heap and in this example has a value of $2040 + 2880 + 120 = 5040$. This is expressed in the table header as:

```
NAXIS1  =    168 / Width of table row in bytes
NAXIS2  =      5 / Number of rows in table
PCOUNT  =   5040 / Random parameter count
  ...
THEAP   =   2880 / Byte offset of heap area
```

The values of `TSCALn` and `TZEROn` for variable-length array column entries are to be applied to the values in the data array in the heap area, not the values of the array descriptor. These keywords can be used to scale data values in either static or variable-length arrays.

7.3.6 Variable-Length Array Guidelines

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 might ~~may~~ also be useful.

Programs that ~~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 might ~~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-length array data *may* be stored in the heap in any order, i.e., the data for row ~~record~~ $N+1$ are not necessarily stored at a larger offset than that for row ~~record~~ N. There *may* be gaps in the heap where no data are 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 are present in each stored row~~record~~. When data are present the stored array can be as large as necessary. This can be useful when storing complex objects as rows ~~records~~ in a table.

Accessing a binary table stored on a random access storage medium is straightforward. Since the rows of data ~~records~~ in the main data table are fixed in size they can ~~may~~ be randomly accessed given the row ~~record~~ number, by computing the offset. Once the row ~~record~~ has been read in, any variable-length array data can ~~may~~ be directly accessed using the element count and offset given by the array descriptor stored in that row~~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 data table rows ~~table records~~ are read in. Once all the table rows ~~records~~ have been read, the array descriptors are sorted by the offset of the array data in the heap. As the heap data are read, arrays are extracted sequentially from the heap and stored in the affected rows ~~records~~ using the back pointers to the row ~~record~~ and field from the table of array descriptors. Since array aliasing is permitted, it might ~~may~~ be necessary to store a given array in more than one field or row~~record~~.

Variable-length arrays are more complicated than regular static arrays and might

~~may~~ not be supported by some software systems. The producers of *FITS* data products should consider the capabilities of the likely recipients of their files when deciding whether or not to use this format, and as a general rule should use it only in cases where it provides significant advantages over the simpler fixed-length array format. In particular, the use of variable-length arrays might ~~may~~ present difficulties for applications that ingest the *FITS* file via a sequential input stream because the application cannot fully process any rows in the table until after the entire fixed-length table and potentially the entire heap has been transmitted as outlined in the previous paragraph.

Section 8

World Coordinate Systems

Representations of the mapping between image coordinates and physical (i.e., world) coordinate systems (WCSs) may be represented within FITS HDUs. The keywords that are used to express these mappings are now rigorously defined in a series of papers on world coordinate systems [11], celestial coordinate systems [12], and spectral coordinate systems [14]. An additional spherical projection, called HEALPix, is defined in reference [22]. These WCS papers have been formally approved by the IAUFWG and therefore are *incorporated* by reference as an official part of this Standard. The reader should refer to these papers for additional details and background information that cannot be included here. Various updates and corrections to the primary WCS papers have been compiled by the authors, and are reflected in this section. Therefore, where conflicts exist, the description in this Standard will prevail.

8.1 Basic Concepts

The essence of representing world coordinate systems in FITS is the association of various reserved keywords with elements of a transformation (or a series of transformations), or with parameters of a projection function. The conversion from pixel coordinates in the data array to world coordinates is simply a matter of applying the specified transformations (in order) via the appropriate keyword values; conversely, defining a WCS for an image amounts to solving for the elements of the transformation matrix(es) or coefficients of the function(s) of interest and recording them in the form of WCS keyword values. The description of the WCS systems and their expression in FITS HDUs is quite extensive and detailed, but is aided by a careful choice of notation. Key elements of the notation are summarized in Table 8.1, and are used throughout this section. The formal definitions of the keywords appear in the following subsections.

The conversion of image pixel coordinates to world coordinates is a multi-step process, as illustrated in Figure 8.1.

Table 8.1. WCS and Celestial Coordinates Notation

Variable(s)	Meaning	Related FITS Keywords
i	Index variable for world coordinates	...
j	Index variable for pixel coordinates	...
a	Alternative WCS version code	...
p_j	Pixel coordinates	...
r_j	Reference pixel coordinates	CRPIXj**a**
m_{ij}	Linear transformation matrix	CDi_j**a** or PCi_j**a**
s_i	Coordinate scales	CDELTi**a**
(x, y)	Projection plane coordinates	...
(ϕ, θ)	Native longitude and latitude	...
(α, δ)	Celestial longitude and latitude	...
(ϕ_0, θ_0)	Native longitude and latitude of the fiducial point	PVi_1**a**†, PVi_2**a**†
(α_0, δ_0)	Celestial longitude and latitude of the fiducial point	CRVALi**a**
(α_p, δ_p)	Celestial longitude and latitude of the native pole	...
(ϕ_p, θ_p)	Native longitude and latitude of the celestial pole	LONPOLE**a** (=PVi_1**a**†), LATPOLE**a** (=PVi_2**a**†)

†Associated with *longitude* axis i.

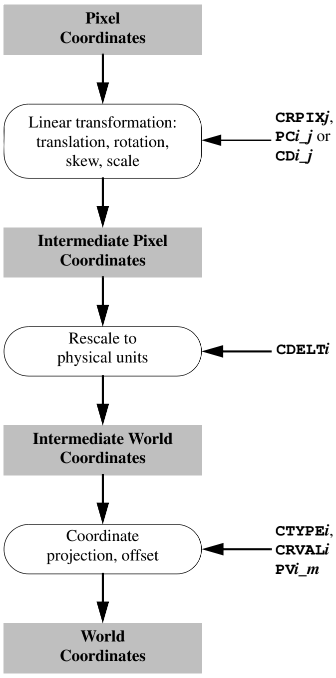

Figure 8.1 A schematic view of converting pixel coordinates to world coordinates.

For all coordinate types, the first step is a linear transformation applied via matrix multiplication of the vector of *pixel coordinate* elements, p_j:

$$q_i = \sum_{j=1}^{N} m_{ij}(p_j - r_j) \tag{8.1}$$

where r_j are the pixel coordinate elements of the reference point, j indexes the pixel axis, and i the world axis. The m_{ij} matrix is a non-singular, square matrix of dimension $N \times N$, where N is the number of world coordinate axes. The elements q_i of the resulting *intermediate pixel coordinate* vector are offsets, in dimensionless pixel units, from the reference point along axes coincident with those of the *intermediate world coordinates*. Thus, the conversion of q_i to the corresponding intermediate world coordinate element x_i is a simple scale:

$$x_i = s_i q_i. \tag{8.2}$$

There are three conventions for associating FITS keywords with the above transformations. In the first formalism, the matrix elements m_{ij} are encoded in the `PC`i_j keywords and the scale factors s_i are encoded in the CDELTi keywords, which *must* have non-zero values. In the second formalism Eqs. (8.1) and (8.2) are combined as

$$x_i = \sum_{j=1}^{N} (s_i m_{ij})(p_j - r_j) \tag{8.3}$$

and the CDi_j keywords encode the product $s_i m_{ij}$. The third convention was widely used before the development of the 2 previously described conventions and uses the CDELTi keywords to define the image scale and the CROTA2 keyword to define a bulk rotation of the image plane. Use of the CROTA2 keyword is now deprecated, and instead the newer PCi_j or CDi_j keywords are *recommended* because they allow for skewed axes and fully general rotation of multi-dimensional arrays. The CDELTi and CROTA2 keywords *may* co-exist with the CDi_j keywords (but CROTA2 *must not* occur with the PCi_j keywords) as an aid to old *FITS* interpreters, but these keywords *must* be ignored by software that supports the CDi_j keyword convention. In all these formalisms the reference pixel coordinates r_j are encoded in the CRPIXi keywords, and the world coordinates at the reference point are encoded in the CRVALi keywords. For additional details, see [11].

The third step of the process, computing the final world coordinates, depends on the type of coordinate system, which is indicated with the value of the CTYPEi keyword. For some simple, linear cases an appropriate choice of normalization for the scale factors allows the world coordinates to be taken directly (or by applying a constant offset) from the x_i (e.g., some spectra). In other cases it is more complicated, and may require the application of some non-linear algorithm (e.g., a projection, as for celestial coordinates), which may require the specification of additional parameters. Where necessary, numeric parameter values for non-linear algorithms *must* be specified via PVi_m keywords and character-valued parameters will be specified via PSi_m keywords, where m is the parameter number.

The application of these formalisms to coordinate systems of interest is discussed in the following sub-sections: §8.2 describes general WCS representations (see [11]), §8.3 describes celestial coordinate systems (see [12]), and §8.4 describes spectral coordinate systems (see [14]).

8.2 World Coordinate System Representations

A variety of keywords have been reserved for computing the coordinate values that are to be associated with any pixel location within an array. The full set is given in Table 8.2; those in most common usage are defined in detail below for convenience. Coordinate system specifications may appear in HDUs that contain simple images in the primary array or in an image extension. Images may also be stored in a multi-dimensional vector cell of a binary table, or as a tabulated list of pixel locations (and optionally, the pixel value) in a table. In these latter 2 types of image representations, the WCS keywords have a different naming convention which reflects the needs of the tabular data structure and the 8-character limit for keyword lengths, but otherwise follow exactly the same rules for type, usage, and default values. See reference [12] for example usage of these keywords. All forms of these reserved keywords *must* be used only as specified in this Standard.

The keywords given below constitute a complete set of fundamental attributes for a WCS description. Although their inclusion in an HDU is optional, FITS writers *should* include a complete set of keywords when describing a WCS. In the event that some keywords are missing, default values *must* be assumed, as specified below.

`WCSAXES` — [integer; default: `NAXIS`, or largest of WCS indexes *i* or *j*]
: Number of axes in the WCS description. This keyword, if present, *must* precede all WCS keywords except `NAXIS` in the HDU. The value of `WCSAXES` *may* exceed the number of pixel axes for the HDU.

`CTYPE`*i* — [character; indexed; default: ‘ ’ (i.e. a linear, undefined axis)]
: Type for the intermediate coordinate axis *i*. Any coordinate type that is not covered by this standard or an officially recognized FITS convention *shall* be taken to be linear. All non-linear coordinate system names *must* be expressed in "4–3" form: the first four characters specify the coordinate type, the fifth character is a hyphen (‘-’), and the remaining three characters specify an algorithm code for computing the world coordinate value. Coordinate types with names of less than four characters are padded on the right with hyphens, and algorithm codes with less than three characters are padded on the right with blanks[1]. Algorithm codes *should* be three characters.

`CUNIT`*i* — [character; indexed; default: ‘ ’ (i.e., undefined)]
: Physical units of `CRVAL` and `CDELT` for axis *i*. Note that units *should* always be specified (see §4.3). Units for celestial coordinate systems defined in this Standard *must* be degrees.

`CRPIX`*j* — [floating-point; indexed; default: 0.0]
: Coordinate of the reference point along the pixel axis *j*. Note that the reference point *may* lie outside the image.

`CRVAL`*i* — [floating-point; indexed; default: 0.0]
: World Coordinate value at the reference point of axis *i*.

`CDELT`*i* — [floating-point; indexed; default: 1.0]
: Increment of the world coordinate at the reference point for axis *i*. The value *must not* be zero.

[1] Example: ‘`RA---UV `’.

Table 8.2. Reserved WCS Keywords

Keyword Description	Primary Array	BINTABLE vector primary	BINTABLE vector alternative	Pixel List primary	Pixel List alternative
Coordinate dimensionality	WCSAXESa	WCAXna		. . .	
Axis type	CTYPEia	iCTYPn	iCTYna	TCTYPn	TCTYna
Axis units	CUNITia	iCUNIn	iCUNna	TCUNIn	TCUNna
Reference value	CRVALia	iCRVLn	iCRVna	TCRVLn	TCRVna
Coordinate increment	CDELTia	iCDLTn	iCDEna	TCDLTn	TCDEna
Reference point	CRPIXja	jCRPXn	jCRPna	TCRPXn	TCRPna
Coordinate rotation[1]	CROTAi	iCROTn		TCROTn	
Transformation matrix[2]	PCi_ja	ijPCna		TPCn_ka or TPn_ka	
Transformation matrix[2]	CDi_ja	ijCDna		TCDn_ka or TCn_ka	
Coordinate parameter	PVi_ma	iPVn_ma or iVn_ma		TPVn_ma or TVn_ma	
Coordinate parameter array	. . .	iVn_Xa		. . .	
Coordinate parameter	PSi_ma	iPSn_ma or iSn_ma		TPSn_ma or TSn_ma	
Coordinate name	WCSNAMEa	WCSNna		TWCSna	
Coordinate axis name	CNAMEia	iCNAna		TCNAna	
Random error	CRDERia	iCRDna		TCRDna	
Systematic error	CSYERia	iCSYna		TCSYna	
WCS cross-ref. target	. . .	WCSTna		. . .	
WCS cross reference	. . .	WCSXna		. . .	
Coordinate rotation	LONPOLEa	LONPna		LONPna	
Coordinate rotation	LATPOLEa	LATPna		LATPna	
Coordinate epoch	EQUINOXa	EQUIna		EQUIna	
Coordinate epoch[3]	EPOCH	EPOCH		EPOCH	
Date of observation	MJD-OBS	MJDOBn		MJDOBn	
Average date of obs	MJD-AVG	MJDAn		MJDAn	
Date/time of observation	DATE-OBS	DOBSn		DOBSn	
Average date/time of obs	DATE-AVG	DAVGn		DAVGn	
Reference frame	RADESYSa or RADECSYS[4]	RADEna		RADEna	
Line rest frequency (Hz)	RESTFRQa or RESTFREQ[4]	RFRQna . . .		RFRQna . . .	
Line rest vac wavelength (m)	RESTWAVa	RWAVna		RWAVna	
Spectral reference frame	SPECSYSa	SPECna		SPECna	
Spectral reference frame	SSYSOBSa	SOBSna		SOBSna	
Spectral reference frame	SSYSSRCa	SSRCna		SSRCna	
Observation X (m)	OBSGEO-X	OBSGXn		OBSGXn	
Observation Y (m)	OBSGEO-Y	OBSGYn		OBSGYn	
Observation Z (m)	OBSGEO-Z	OBSGZn		OBSGZn	
Radial velocity (m s^{-1})	VELOSYSa	VSYSna		VSYSna	
Redshift of source	ZSOURCEa	ZSOUna		ZSOUna	
Angle of true velocity	VELANGLa	VANGna		VANGna	

[1] CROTAi form is deprecated but still in use. It *must not* be used with PCi_j, PVi_m, and PSi_m.

[2] PCi_j and CDi_j forms of the transformation matrix are mutually exclusive, and *must not* appear together in the same HDU.

[3] EPOCH is deprecated. Use EQUINOX instead.

[4] These 8-character keywords are deprecated; the 7-character forms, which can include an alternate version code letter at the end, *should* be used instead.

Note. — The indexes j and i are pixel and intermediate world coordinate axis numbers, respectively. Within a table, the index n refers to a column number, and m refers to a coordinate parameter number. The index k also refers to a column number. The indicator a is either blank (for the primary coordinate description) or a character A through Z that specifies the coordinate version. See text.

`CROTA`i — [floating-point; indexed; default: 0.0]
: The amount of rotation from the standard coordinate system to a different coordinate system. Further use of this of this keyword is deprecated, in favor of the newer formalisms that use the `CD`i_j or `PC`i_j keywords to define the rotation.

`PC`i_j — [floating-point; defaults: 1.0 when $i = j$, 0.0 otherwise]
: Linear transformation matrix between pixel axes j and intermediate coordinate axes i. The `PC`i_j matrix *must* not be singular.

`CD`i_j — [floating-point; defaults: 0.0, but see below]
: Linear transformation matrix (with scale) between pixel axes j and intermediate coordinate axes i. This nomenclature is equivalent to `PC`i_j when `CDELT`i is unity. The `CD`i_j matrix *must* not be singular. Note that the `CD`i_j formalism is an exclusive alternative to `PC`i_j, and the `CD`i_j and `PC`i_j keywords *must* not appear together within an HDU.

In addition to the restrictions noted above, if any `CD`i_j keywords are present in the HDU, all other unspecified `CD`i_j keywords *shall* default to zero. If no `CD`i_j keywords are present then the header *shall* be interpreted as being in `PC`i_j form whether or not any `PC`i_j keywords are actually present in the HDU.

Some non-linear algorithms that describe the transformation between pixel and intermediate coordinate axes require parameter values. A few non-linear algorithms also require character-valued parameters, e.g., table lookups require the names of the table extension and the columns to be used. Where necessary parameter values *must* be specified via the following keywords:

`PV`i_m — [floating-point]
: Numeric parameter values for intermediate world coordinate axis i, where m is the parameter number. Leading zeros *must* not be used, and m may have only values in the range 0 through 99, and that are defined for the particular non-linear algorithm.

`PS`i_m — [character]
: Character-valued parameters for intermediate world coordinate axis i, where m is the parameter number. Leading zeros *must* not be used, and m may have only values in the range 0 through 99, and that are defined for the particular non-linear algorithm.

The following keywords, while not essential for a complete specification of an image WCS, can be extremely useful for readers to interpret the accuracy of the WCS representation of the image.

`CRDER`i — [floating-point; default: 0.0]
: Random error in coordinate i, which *must* be non-negative.

`CSYER`i — [floating-point; default: 0.0]
Systematic error in coordinate i, which *must* be non-negative.

These values *should* give a representative average value of the error over the range of the coordinate in the HDU. The total error in the coordinates would be given by summing the individual errors in quadrature.

8.2.1 Alternative WCS Axis Descriptions

In some cases it is useful to describe an image with more than one coordinate type[2]. Alternative WCS descriptions *may* be added to the header by adding the appropriate sets of WCS keywords, and appending to all keywords in each set an alphabetic code in the range `A` through `Z`. Keywords that may be used in this way to specify a coordinate system version are indicated in Table 8.2 with the suffix a. All implied keywords with this encoding are *reserved* keywords, and *must* only be used in FITS HDUs as specified in this Standard. The axis numbers *must* lie in the range 1 through 99, and the coordinate parameter m *must* lie in the range 0 through 99, both with no leading zeros.

The *primary* version of the WCS description is that specified with **a** as the blank character[3]. Alternative axis descriptions are optional, but *must* not be specified unless the primary WCS description is also specified. If an alternative WCS description is specified, all coordinate keywords for that version *must* be given even if the values do not differ from those of the primary version. Rules for the default values of alternative coordinate descriptions are the same as those for the primary description. The alternative descriptions are computed in the same fashion as the primary coordinates. The type of coordinate depends on the value of `CTYPE`i**a**, and may be linear in one of the alternative descriptions and non-linear in another.

The alternative version codes are selected by the FITS writer; there is no requirement that the codes be used in alphabetic sequence, nor that one coordinate version differ in its parameter values from another. An optional keyword `WCSNAME`**a** is also defined to name, and otherwise document, the various versions of WCS descriptions:

`WCSNAME`**a** — [character; default for **a**: `' '` (i.e., blank, for the primary WCS, else a character `A` through `Z` that specifies the coordinate version]
Name of the world coordinate system represented by the WCS keywords with the suffix **a**. Its primary function is to provide a means by which to specify a particular WCS if multiple versions are defined in the HDU.

[2] Examples include the frequency, velocity, and wavelength along a spectral axis (only one of which, of course, could be linear), or the position along an imaging detector in both meters and degrees on the sky.

[3] There are a number of keywords (e.g. ij`PC`n**a** where the a could be pushed off the 8-char keyword name for plausible values of i, j, k, n, and m. In such cases a is still said to be "blank" although it is not the blank character.

8.3 Celestial Coordinate System Representations

The conversion from intermediate world coordinates (x, y) in the plane of projection to celestial coordinates involves two steps: a spherical projection to native longitude and latitude (ϕ, θ), defined in terms of a convenient coordinate system (i.e., *native spherical coordinates*), followed by a spherical rotation of these native coordinates to the required celestial coordinate system (α, δ). The algorithm to be used to define the spherical projection *must* be encoded in the `CTYPE`i keyword as the three-letter algorithm code, the allowed values for which are specified in Table 8.3 and defined in references [12] and [22]. The target celestial coordinate system is also encoded into the left-most portion of the `CTYPE`i keyword as the coordinate type.

For the final step, the parameter `LONPOLEa` must be specified, which is the native longitude of the celestial pole, ϕ_p. For certain projections (such as cylindricals and conics, which are less commonly used in astronomy), the additional keyword `LATPOLEa` must be used to specify the native latitude of the celestial pole. See [12] for the transformation equations and other details.

The accepted celestial coordinate systems are: the standard equatorial (`RA--` and `DEC-`), and others of the form x`LON` and x`LAT` for longitude-latitude pairs, where x is `G` for Galactic, `E` for ecliptic, `H` for helioecliptic and `S` for supergalactic coordinates. Since the representation of planetary, lunar, and solar coordinate systems could exceed the 26 possibilities afforded by the single character x, pairs of the form yz`LN` and yz`LT` *may* be used as well.

`RADESYSa` — [character; default: `FK4`, `FK5`, or `ICRS`: see below]
Name of the reference frame of equatorial or ecliptic coordinates, whose value *must* be one of those specified in Table 8.4. The default value is `FK4` if the value of `EQUINOXa` $<$ 1984.0, `FK5` if `EQUINOXa` $\geq$ 1984.0, or `ICRS` if `EQUINOXa` is not given.

`EQUINOXa` — [floating-point; default: see below]
Epoch of the mean equator and equinox in years, whose value *must* be non-negative. The interpretation of epoch depends upon the value of `RADESYSa` if present: *Besselian* if the value is `FK4` or `FK4-NO-E`, *Julian* if the value is `FK5`; *not* applicable if the value is `ICRS` or `GAPPT`.

`EPOCH` — [floating-point]
This keyword is deprecated and should not be used in new FITS files. It is reserved primarily to prevent its use with other meanings. The EQUINOX keyword *shall* be used instead. The value field of this keyword was previously defined to ~~The value field shall~~ contain a floating-point number giving the equinox in years for the celestial coordinate system in which positions are expressed. ~~Starting with Version 1, this standard has deprecated the use of the EPOCH keyword and thus it shall~~

Table 8.3. Reserved Celestial Coordinate Algorithm Codes

Code	Default ϕ_0	Default θ_0	Properties[1]	Projection Name
			Zenithal (azimuthal) projections	
AZP	0°	90°	§5.1.1	Zenithal perspective
SZP	0°	90°	§5.1.2	Slant zenithal perspective
TAN	0°	90°	§5.1.3	Gnomonic
STG	0°	90°	§5.1.4	Stereographic
SIN	0°	90°	§5.1.5	Slant orthographic
ARC	0°	90°	§5.1.6	Zenithal equidistant
ZPN	0°	90°	§5.1.7	Zenithal polynomial
ZEA	0°	90°	§5.1.8	Zenithal equal-area
AIR	0°	90°	§5.1.9	Airy
			Cylindrical projections	
CYP	0°	0°	§5.2.1.	Cylindrical perspective
CEA	0°	0°	§5.2.2	Cylindrical equal area
CAR	0°	0°	§5.2.3	Plate carrée
MER	0°	0°	§5.2.4	Mercator
			Pseudo-cylindrical and related projections	
SFL	0°	0°	§5.3.1	Samson-Flamsteed
PAR	0°	0°	§5.3.2	Parabolic
MOL	0°	0°	§5.3.3	Mollweide
AIT	0°	0°	§5.3.4	Hammer-Aitoff
			Conic projections	
COP	0°	θ_a	§5.4.1	Conic perspective
COE	0°	θ_a	§5.4.2	Conic equal-area
COD	0°	θ_a	§5.4.3	Conic equidistant
COO	0°	θ_a	§5.4.4	Conic orthomorphic
			Polyconic and pseudoconic projections	
BON	0°	0°	§5.5.1	Bonne's equal area
PCO	0°	0°	§5.5.2	Polyconic
			Quad-cube projections	
TSC	0°	0°	§5.6.1	Tangential spherical cube
CSC	0°	0°	§5.6.2	COBE quadrilateralized spherical cube
QSC	0°	0°	§5.6.3	Quadrilateralized spherical cube
			HEALPix grid projection	
HPX	0°	0°	§6[2]	HEALPix grid

[1] Refer to the indicated section in reference [12] for a detailed description.

[2] This projection is defined in reference [22].

Table 8.4. Allowed Values of `RADESYSa`

Value	Definition
`ICRS`	International Celestial Reference System
`FK5`	Mean place, new (IAU 1984) system
`FK4`[1]	Mean place, old (Bessel-Newcomb) system
`FK4-NO-E`[1]	Mean place: but without eccentricity terms
`GAPPT`	Geocentric apparent place, IAU 1984 system

[1]New FITS files should avoid using these older reference systems.

~~not be used in FITS files created after the adoption of this standard; rather, the EQUINOX keyword shall be used.~~

`DATE-OBSa` — [floating-point]
This reserved keyword is defined in §4.4.2.2.

`MJD-OBSa` — [floating-point; default: `DATE-OBS` if given, otherwise no default]
Modified Julian Date (JD – 2,400,000.5) of the observation, whose value corresponds (by default) to the *start* of the observation, unless another interpretation is explained in the comment field. No specific time system (e.g., UTC, TAI, etc.) is defined for this or any of the other time-related keywords. It is *recommended* that the `TIMESYS` keyword, as defined in Appendix B be used to specify the time system.

`LONPOLEa` — [floating-point; default: ϕ_0 if $\delta_0 \geq \theta_0$, $\phi_0 + 180°$ otherwise]
Longitude in the native coordinate system of the celestial system's north pole. Normally, ϕ_0 is zero unless a non-zero value has been set for `PV`i`_1a`, which is associated with the *longitude* axis. This default applies for all values of θ_0, including $\theta_0 = 90°$, although the use of non-zero values of θ_0 are discouraged in that case.

`LATPOLEa` — [floating-point; default: 90°, or no default if $(\theta_0, \delta_0, \phi_p - \phi_0) = (0, 0, \pm 90°)$]
Latitude in the native coordinate system of the celestial system's north pole, or equivalently, the latitude in the celestial coordinate system of the native system's north pole. May be ignored or omitted in cases where `LONPOLEa` completely specifies the rotation to the target celestial system.

Table 8.5. Reserved Spectral Coordinate Type Codes[1]

Code	Type	Symbol	Assoc. Variable	Default Units
`FREQ`	Frequency	ν	ν	Hz
`ENER`	Energy	E	ν	J
`WAVN`	Wavenumber	κ	ν	m^{-1}
`VRAD`	Radio velocity	V	ν	$\mathrm{m\ s}^{-1}$
`WAVE`	Vacuum wavelength	λ	λ	m
`VOPT`	Optical velocity	Z	λ	$\mathrm{m\ s}^{-1}$
`ZOPT`	Redshift	z	λ	...
`AWAV`	Air wavelength	λ_a	λ_a	m
`VELO`	Apparent radial velocity	v	v	$\mathrm{m\ s}^{-1}$
`BETA`	Beta factor (v/c)	β	v	...

[1]Characters 1 through 4 of the value of the keyword `CTYPE`*i***a**.

8.4 Spectral Coordinate System Representations

This section discusses the conversion of intermediate world coordinates to spectral coordinates with common axes such as frequency, wavelength, and apparent radial velocity (represented here with the coordinate variables ν, λ, or v). The key point for constructing spectral WCS in FITS is that one of these coordinates *must* be sampled linearly in the dispersion axis; the others are derived from prescribed, usually non-linear transformations. Frequency and wavelength axes *may* also be sampled linearly in their logarithm.

Following the convention for the `CTYPE`*i***a** keyword, when *i* is the spectral axis the first four characters *must* specify a code for the coordinate type; for non-linear algorithms the fifth character *must* be a hyphen, and the next three characters *must* specify a predefined algorithm for computing the world coordinates from the intermediate physical coordinates. The coordinate type *must* be one of those specified in Table 8.5. When the algorithm is linear, the remainder of the `CTYPE`*i***a** keyword *must* be blank. When the algorithm is non-linear, the 3-letter algorithm code *must* be one of those specified in Table 8.6. The relationships between the basic physical quantities ν, λ, and v, as well as the relationships between various derived quantities are given in reference [14].

The generality of the algorithm for specifying the spectral coordinate system and its representation suggests that some additional description of the coordinate may be helpful beyond what can be encoded in the the first four characters of the `CTYPE`*i***a**

Table 8.6. Non-linear Spectral Algorithm Codes

Code	Regularly sampled in	Expressed as
`F2W`	Frequency	Wavelength
`F2V`		Apparent radial velocity
`F2A`		Air wavelength
`W2F`	Wavelength	Frequency
`W2V`		Apparent radial velocity
`W2A`		Air wavelength
`V2F`	Apparent radial velocity	Frequency
`V2W`		Wavelength
`V2A`		Air wavelength
`A2F`	Air wavelength	Frequency
`A2W`		Wavelength
`A2V`		Apparent radial velocity
`LOG`	Logarithm	Any 4-letter coordinate type
`GRI`	Detector	Any from Table 8.5
`GRA`	Detector	Any from Table 8.5
`TAB`	Not regular	Any 4-letter coordinate type

[1]Characters 6 through 8 of the value of the keyword `CTYPE`i**a**.

keyword; `CNAME`*i***a** is reserved for this purpose. Note that this keyword provides a name for an axis in a particular WCS, while the `WCSNAME`**a** keyword names the particular WCS as a whole. In order to convert between some form of radial velocity and either frequency or wavelength, the keywords `RESTFRQ`**a** and `RESTWAV`**a**, respectively, are reserved.

`CNAME`*i***a** — [character; default: default: ‘ ’ (i.e. a linear, undefined axis)]
Spectral coordinate description which *must* not exceed 68 characters in length.

`RESTFRQ`**a** — [floating-point; default: none]
Rest frequency of the of the spectral feature of interest. The physical unit *must* be Hz.

`RESTWAV`**a** — [floating-point; default: none]
Vacuum rest wavelength of the of the spectral feature of interest. The physical unit *must* be m.

One or the other of `RESTFRQ`**a** or `RESTWAV`**a** *should* be given when it is meaningful to do so.

8.4.1 Spectral Coordinate Reference Frames

Frequencies, wavelengths, and apparent radial velocities are always referred to some selected standard of rest (i.e., reference frame). While the spectra are obtained they are, of necessity, in the observer's rest frame. The velocity correction from topocentric (the frame in which the measurements are usually made) to standard reference frames (which *must* be one of those given in Table 8.7) are dependent on the dot product with time-variable velocity vectors. That is, the velocity with respect to a standard reference frame depends upon direction, and the velocity (and frequency and wavelength) with respect to the local standard of rest is a function of the celestial coordinate within the image. The keywords `SPECSYS`**a** and `SSYSOBS`**a** are reserved and, if used, *must* describe the reference frame in use for the spectral axis coordinate(s) and the spectral reference frame that was held constant during the observation, respectively. In order to compute the velocities it is necessary to have the date and time of the observation; the keywords `DATE-AVG` and `MJD-AVG` are reserved for this purpose.

`DATE-AVG` — [character; default: none]
Calendar date of the mid-point of the observation, expressed in the same way as the `DATE-OBS`keyword.

`MJD-AVG` — [floating-point; default: none]
Modified Julian Date (JD – 2,400,000.5) of the mid-point of the observation.

SPECSYSa — [character; default: none]
The reference frame in use for the spectral axis coordinate(s). Valid values are given in Table 8.7.

SSYSOBSa — [character; default: TOPOCENT]
The spectral reference frame that is constant over the range of the non-spectral world coordinates. Valid values are given in Table 8.7.

The transformation from the rest frame of the observer to a standard reference frame requires a specification of the location on Earth[4] of the instrument used for the observation in order to calculate the diurnal Doppler correction due to the Earth's rotation. The location, if specified, *shall* be represented as a geocentric Cartesian triple with respect to a standard ellipsoidal geoid at the time of the observation. While the position can often be specified with an accuracy of 1 meter or better, for most purposes positional errors of several kilometers will have negligible impact on the computed velocity correction. For details, see reference [14].

OBSGEO-Xa — [floating-point; default: none]
X−coordinate (in meters) of a cartesian triplet that specifies the location, with respect to a standard, geocentric terrestrial reference frame, where the observation took place. The coordinate *must* be valid at the epoch MJD-AVG or DATE-AVG.

OBSGEO-Ya — [floating-point; default: none]
Y−coordinate (in meters) of a cartesian triplet that specifies the location, with respect to a standard, geocentric terrestrial reference frame, where the observation took place. The coordinate *must* be valid at the epoch MJD-AVG or DATE-AVG.

OBSGEO-Za — [floating-point; default: none]
Z−coordinate (in meters) of a cartesian triplet that specifies the location, with respect to a standard, geocentric terrestrial reference frame, where the observation took place. The coordinate *must* be valid at the epoch MJD-AVG or DATE-AVG.

Information on the relative radial velocity between the observer and the selected standard of rest in the direction of the celestial reference coordinate *may* be provided, and if so *shall* be given by the **VELOSYSa** keyword. The frame of rest defined with respect to the emitting source may be represented in FITS; for this reference frame it is necessary to define the velocity with respect to some other frame of rest. The keywords **SPECSYSa** and **ZSOURCEa** are used to document the choice of reference frame and the value of the systemic velocity of the source, respectively.

[4] The specification of location for an instrument on a spacecraft in flight requires an ephemeris; keywords that might be required in this circumstance are not defined here.

`SSYSSRCa` — [character; default: none]
Reference frame for the value expressed in the `ZSOURCEa` keyword to document the systemic velocity of the observed source. Value *must* be one of those given in Table 8.7 *except* for `SOURCE`.

`VELOSYSa` — [floating-point; default: none]
Relative radial velocity between the observer and the selected standard of rest in the direction of the celestial reference coordinate. Units *must* be m s^{-1}. The `CUNIT`*ia* keyword is not used for this purpose since the WCS version *a* might not be expressed in velocity units.

`ZSOURCEa` — [floating-point; default: none]
Radial velocity with respect to an alternative frame of rest, expressed as a unitless redshift (i.e., velocity as a fraction of the speed of light in vacuum). Used in conjunction with `ZSOURCEa` to document the systemic velocity of the observed source.

`VELANGLa` — [floating-point; default:+90.]
In the case of relativistic velocities (e.g., a beamed astrophysical jet) the transverse velocity component is important. This keyword may be used to express the orientation of the space velocity vector with respect to the plane of the sky. See appendix A of reference [14] for further details.

8.5 Conventional Coordinate Types

The first *FITS* paper [2] listed a number of "suggested values" for the `CTYPE`*i* keyword. Two of these have the attribute the associated world coordinates can assume only integer values and that the meaning of these integers is only defined by convention. The first "conventional" coordinate is `CTYPE`*i*`a` = `'COMPLEX'` to specify that complex values (i.e., pairs of real and imaginary components) are stored in the data array (along with an optional weight factor). Thus, the complex axis of the data array will contain 2 values (or 3 if the weight is specified). By convention, the real component has a coordinate value of 1, the imaginary component has a coordinate value of 2, and the weight, if any, has a coordinate value of 3. Table 8.8 illustrates the required keywords for an array of 100 complex values (without weights).

The second conventional coordinate is `CTYPE`*i*`a` = `'STOKES'` to specify the polarization of the data. Conventional values, their symbols, and polarizations are given in Table 8.9.

Table 8.7. Spectral Reference Systems

Value	Definition
TOPOCENT	Topocentric
GEOCENTR	Geocentric
BARYCENT	Barycentric
HELIOCEN	Heliocentric
LSRK	Local standard of rest (kinematic)
LSRD	Local standard of rest (dynamic)
GALACTOC	Galactocentric
LOCALGRP	Local Group
CMBDIPOL	Cosmic microwave background dipole
SOURCE	Source rest frame

Note. — These are the allowed values of the SPECSYS**a**, SSYSOBS**a**, and SSYSSRC**a** keywords.

Table 8.8. Example keywords for a 100 element array of complex values.

Keyword		
SIMPLE	=	T
BITPIX	=	-32
NAXIS	=	2
NAXIS1	=	2
NAXIS2	=	100
CTYPE1	= 'COMPLEX'	
CRVAL1	=	0.
CRPIX1	=	0.
CDELT1	=	1.
END		

Table 8.9. Conventional Stokes values.

Value	Symbol	Polarization
1	I	Standard Stokes unpolarized
2	Q	Standard Stokes linear
3	U	Standard Stokes linear
4	V	Standard Stokes circular
−1	RR	Right-right circular
−2	LL	Left-left circular
−3	RL	Right-left cross-circular
−4	LR	Left-right cross-circular
−5	XX	X parallel linear
−6	YY	Y parallel linear
−7	XY	XY cross linear
−8	YX	YX cross linear

Appendix A

Syntax of Keyword Records~~Card Images~~

~~(~~This Appendix is not part of the ~~NOST~~ *FITS* standard but is included for convenient reference. ~~)~~

The following notation is used in defining the formal syntax.

:=	means 'is defined to be'
X | Y	means one of X or Y (no ordering relation is implied)
[X]	means that X is *optional*
X...	means X is repeated 1 or more times
'B'	means the ASCII character B
'A'–'Z'	means one of the ASCII characters A through Z in the ASCII collating sequence, as shown in Appendix D
\0xnn	means the ASCII character associated with the hexadecimal code nn
{...}	expresses a constraint or a comment (it immediately follows the syntax rule)

The following statements define the formal syntax used in *FITS* free-format keyword records.

[EDITOR'S NOTE: the string 'card_ image' has been replaced by 'keyword_ record' in the following section.]

FITS_keyword_record :=
    FITS_commentary_keyword_record | FITS_value_keyword_record

FITS_commentary_keyword_record :=
    COMMENT_keyword [ascii_text_char...] |
    HISTORY_keyword [ascii_text_char...] |

BLANKFIELD_keyword [ascii_text_char...] |
keyword_field anychar_but_equal [ascii_text_char...] |
keyword_field ‘~~’~~=’ anychar_but_space [ascii_text_char...]
{Constraint: The total number of characters in a FITS_commentary_keyword_record *must* be exactly equal to 80.}

FITS_value_keyword_record :=
keyword_field value_indicator [space...] [value] [space...] [comment]
{Constraint: The total number of characters in a FITS_value_keyword_record *must* be exactly equal to 80.}
{Comment: If the value field is not present, the value of the *FITS* keyword is not defined.}

keyword_field :=
[keyword_char...] [space...]
{Constraint: The total number of characters in the keyword_field *must* be exactly equal to 8.}

keyword_char :=
‘A’–‘Z’ | ‘0’–‘9’ | ‘_’ | ‘-’

COMMENT_keyword :=
‘C’ ‘O’ ‘M’ ‘M’ ‘E’ ‘N’ ‘T’ space

HISTORY_keyword :=
‘H’ ‘I’ ‘S’ ‘T’ ‘O’ ‘R’ ‘Y’ space

BLANKFIELD_keyword :=
space space space space space space space space

value_indicator :=
‘=’ space

space :=
‘ ’

comment :=
‘/’ [ascii_text_char...]

ascii_text_char :=
space–‘~’

anychar_but_equal :=
  space–'<' | '>'–'~'

anychar_but_space :=
  '!'–'~'

value :=
  character_string_value | logical_value | integer_value | floating_value |
  complex_integer_value | complex_floating_value

character_string_value :=
  begin_quote [string_text_char...] end_quote
{Constraint: The begin_quote and end_quote are not part of the character string value but only serve as delimiters. Leading spaces are significant; trailing spaces are not.}

begin_quote :=
  quote

end_quote :=
  quote
{Constraint: The ending quote *must not* be immediately followed by a second quote.}

quote :=
  \0x27

string_text_char :=
  ascii_text_char
{Constraint: A string_text_char is identical to an ascii_text_char except for the quote char; a quote char is represented by two successive quote chars.}

logical_value :=
  'T' | 'F'

integer_value :=
  [sign] digit [digit...]
{Comment: Such an integer value is interpreted as a signed decimal number. It *may* contain leading zeros.}

sign :=
  '-' | '+'

digit :=
  '0'–'9'

floating_value :=
  decimal_number [exponent]

decimal_number :=
  [sign] [integer_part] ['.' ~~?.?~~ [fraction_part]]

{Constraint: At least one of the integer_part and fraction_part *must* be present.}

integer_part :=
  digit | [digit...]

fraction_part :=
  digit | [digit...]

exponent :=
  exponent_letter [sign] digit [digit...]

exponent_letter :=
  'E' | 'D'

complex_integer_value :=
  '(' [space...] real_integer_part [space...] ',' [space...]
  imaginary_integer_part [space...] ')'

real_integer_part :=
  integer_value

imaginary_integer_part :=
  integer_value

complex_floating_value :=
  '(' [space...] real_floating_part [space...] ',' [space...]
  imaginary_floating_part [space...] ')'

real_floating_part :=
  floating_value

imaginary_floating_part :=
  floating_value

Appendix B

Suggested Time Scale Specification

~~Not part of formal `DATExxxx` agreement~~**This Appendix is not part of the *FITS* standard, but is included for convenient reference.**

1. Use of the keyword `TIMESYS` is suggested as an implementation of the time scale specification. It sets the principal time system for time-related keywords and data in the HDU (i.e., it does not preclude the addition of keywords or data columns that provide information for transformations to other time scales, such as sidereal times or barycenter corrections). Each HDU *shall* contain not more than one `TIMESYS` keyword. Initially, officially allowed values are shown below. For reference, see: Explanatory Supplement to the Astronomical Almanac, P. K. Seidelmann, ed., University Science Books, 1992, ISBN 0-935702-68-7, or

   `http://tycho.usno.navy.mil/systime.html`

   `UTC` Coordinated Universal Time; defined since 1972.

   `UT` Universal Time, equal to Greenwich Mean Time (GMT) since 1925; the UTC equivalent before 1972; see: Explanatory Supplement, p. 76.

   `TAI` International Atomic Time; 'UTC without the leap seconds'; 31 s ahead of UTC on 1997-07-01.

   `AT` International Atomic Time; deprecated synonym of TAI.

   `ET` Ephemeris Time, the predecessor of TT and TDB; valid until 1984.

   `TT` Terrestrial Time, the IAU standard time scale since 1984; continuous with ET and synchronous with (but 32.184 s ahead of) TAI.

   `TDT` Terrestrial Dynamical Time; = TT.

`TDB` Barycentric Dynamical Time.

`TCG` Geocentric Coordinate Time; runs ahead of TT since 1977-01-01 at a rate of approximately 22 ms/year.

`TCB` Barycentric Coordinate Time; runs ahead of TDB since 1977-01-01 at a rate of approximately 0.5 s/year.

Use of Global Positioning Satellite (GPS) time (19 s behind TAI) is deprecated.

2. By default, times will be deemed to be as measured at the detector (or ~~in practical cases,~~ at the observatory) for time scales defined on the geoid ~~times that run synchronously with TAI~~ (i.e., TAI, UTC ~~,~~ and TT). In the case of the coordinate times TCG, TCB and TDB, the observation is assumed to have been referred to the associated spatial origin (namely the geocenter for TCG and the solar-system barycenter for TCB and TDB) by allowing for light time~~coordinate times (such as TCG and TCB) and TDB which are tied to an unambiguous coordinate origin, the default meaning of time values will be: time as if the observation had taken place at the origin of the coordinate timesystem~~. These defaults follow common practice; a future convention on time scale issues in *FITS* files *may* allow other combinations but *shall* preserve this default behavior. The rationale is that raw observational data are most likely to be tagged by a clock that is synchronized with TAI, while a transformation to coordinate times or TDB is usually accompanied by a spatial transformation, as well. This implies that path length differences have been corrected for. Note that the same distant event recorded in a *FITS* file in both TDB and UTC will have times that differ by (typically) several minutes~~difference TDB − UTC , in that case, is approximately sinusoidal, with period one year and amplitude up to 500 s, depending on source position~~. Also, note that when the location is not unambiguous (such as in the case of an interferometer) precise specification of the location is strongly encouraged in, for instance, geocentric Cartesian coordinates.

3. Note that `TT` is the IAU preferred standard. It can ~~may~~ be considered equivalent to `TDT` and `ET`, though `ET` *should not* be used for data taken after 1984. For reference, see: Explanatory Supplement, pp. 40-48.

4. If the `TIMESYS` keyword is absent or has an unrecognized value, the value `UTC` will be assumed for dates since 1972, and `UT` for pre-1972 data.

5. Examples. The three legal representations of the date of October 14, 1996, might be written as:

```
DATE-OBS= '14/10/96'           / Original format, means 1996 Oct 14.
```

```
TIMESYS = 'UTC     '           / Explicit time scale specification: UTC.
DATE-OBS= '1996-10-14'         / Date of start of observation in UTC.

DATE-OBS= '1996-10-14'         / Date of start of observation, also in UTC.

TIMESYS = 'TT      '           / Explicit time scale specification: TT.
DATE-OBS= '1996-10-14T10:14:36.123' / Date and time of start of obs. in TT.
```

6. The convention suggested in this Appendix is part of the mission-specific *FITS* conventions adopted for, and used in, the RXTE archive, building on existing High Energy Astrophysics *FITS* conventions. See:

   http://heasarc.gsfc.nasa.gov/docs/xte/abc/time_tutorial.html
   http://heasarc.gsfc.nasa.gov/docs/xte/abc/time.html

   The VLBA project has adopted a convention where the keyword TIMSYS, rather than TIMESYS, is used, currently allowing the values UTC and IAT. See p. 38 and p. 39 ~~9 and p. 16~~ of:

   http://www.cv.nrao.edu/fits/documents/drafts/idi-format~~vlba-format~~.ps

Appendix C

Summary of Keywords

~~(~~This Appendix is not part of the ~~NOST~~ *FITS* standard~~Standard~~, but is included for convenient reference. ~~).~~

All of the mandatory and reserved keywords that are defined in the standard, except for the reserved WCS keywords that are are discussed separately in §8, are listed in the following tables.

Primary HDU	Conforming Extension	Image Extension	ASCII Table Extension	Binary Table Extension	Random Groups Records
SIMPLE	XTENSION	XTENSION[1]	XTENSION[2]	XTENSION[3]	SIMPLE
BITPIX	BITPIX	BITPIX	BITPIX = 8	BITPIX = 8	BITPIX
NAXIS	NAXIS	NAXIS	NAXIS = 2	NAXIS = 2	NAXIS
NAXISn[4]	NAXISn[4]	NAXISn[4]	NAXIS1	NAXIS1	NAXIS1 = 0
~~EXTEND~~ ~~[5]~~	PCOUNT	PCOUNT = 0	NAXIS2	NAXIS2	NAXISn[4]
END	GCOUNT	GCOUNT = 1	PCOUNT = 0	PCOUNT	GROUPS = T
	END	END	GCOUNT = 1	GCOUNT = 1	PCOUNT
			TFIELDS	TFIELDS	GCOUNT
			TFORMn [5] ~~[6]~~	TFORMn[5] ~~[6]~~	END
			TBCOLn [5] ~~[6]~~	END	
			END		

[1] XTENSION= 'IMAGE   ' for the image extension.
[2] XTENSION= 'TABLE   ' for the ASCII table extension.
[3] XTENSION= 'BINTABLE' for the binary table extension.
[4] Runs from 1 through the value of NAXIS.
[5] ~~Required only if extensions are present.~~
~~[6]~~ [5] Runs from 1 through the value of TFIELDS.

Table C.1 Mandatory *FITS* keywords for the structures described in this document.

All[1] HDUs	Array[2] HDUs	Conforming Extension	ASCII Table Extension	Binary Table Extension	Random Groups Records
DATE	BSCALE	EXTNAME	TSCALn	TSCALn	PTYPEn
ORIGIN	BZERO	EXTVER	TZEROn	TZEROn	PSCALn
BLOCKED[3] ~~[2]~~	BUNIT	EXTLEVEL	TNULLn	TNULLn	PZEROn
AUTHOR	BLANK		TTYPEn	TTYPEn	
REFERENC	~~CTYPEn~~		TUNITn	TUNITn	
COMMENT	~~CRPIXn~~		TDISPn	TDISPn	
HISTORY	~~CROTAn~~			TDIMn	
␣␣␣␣␣␣␣␣	~~CRVALn~~			THEAP	
DATE-OBS	~~CDELTn~~				
TELESCOP	DATAMAX				
INSTRUME	DATAMIN				
OBSERVER					
OBJECT					
EQUINOX					
EPOCH[3] ~~[2]~~					
EXTEND[4]					

[1] These keywords are further categorized in Table C.3.
[2] Primary HDU, image extension, user-defined HDUs with same array structure.
~~[2]~~ [3] Deprecated.
[4] Only permitted in the primary HDU

Table C.2 Reserved *FITS* keywords for the structures described in this document.

Production	Bibliographic	Commentary	Observation
DATE	AUTHOR	COMMENT	DATE-OBS
ORIGIN	REFERENC	HISTORY	TELESCOP
BLOCKED[1]		␣␣␣␣␣␣␣␣	INSTRUME
			OBSERVER
			OBJECT
			EQUINOX
			EPOCH[1]

[1] Deprecated.

Table C.3 General reserved *FITS* keywords described in this document.

Appendix D

ASCII Text

~~(~~**This appendix is not part of the ~~NOST~~ *FITS* standard**; the material in it is based on the ANSI standard for ASCII [20] ~~14~~and is included here for informational purposes.)

In the following table, the first column is the decimal and the second column the hexadecimal value for the character in the third column. The characters hexadecimal 20 to 7E (decimal 32 to 126) constitute the subset referred to in this document as the restricted set of ASCII text characters.

ASCII Control			ASCII Text								
dec	hex	char	dec	hex	char	dec	hex	char	dec	hex	char
0	00	NUL	32	20	SP	64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL[1]

[1] Not ASCII Text

Table D.1 ASCII character set

Appendix E

IEEE Floating-Point Formats

~~(~~**The material in this Appendix is not part of this standard**; it is adapted from the IEEE-754 floating-point standard [21] ~~15~~and provided for informational purposes. It is not intended to be a comprehensive description of the IEEE formats; readers should refer to the IEEE standard.)

FITS recognizes all IEEE basic formats, including the special values.

E.1 Basic Formats

Numbers in the single and double formats are composed of the following three fields:

1. 1-bit sign s
2. Biased exponent $e = E + bias$
3. Fraction $f = \bullet b_1 b_2 \cdots b_{p-1}$

The range of the unbiased exponent E *shall* include every integer between two values E_{min} and E_{max}, inclusive, and also two other reserved values $E_{min} - 1$ to encode ± 0 and denormalized numbers, and $E_{max}+1$ to encode $\pm\infty$ and NaNs. The foregoing parameters are given in Table E.1. Each nonzero numerical value has just one encoding. The fields are interpreted as follows:

E.1.1 Single

A 32-bit single format number X is divided as shown in Fig. E.1. The value v of X is inferred from its constituent fields thus

1. If $e = 255$ and $f \neq 0$, then v is NaN regardless of s
2. If $e = 255$ and $f = 0$, then $v = (-1)^s \infty$

Parameter	Format			
	Single	Single Extended	Double	Double Extended
p	24	≥ 32	53	≥ 64
E_{max}	$+127$	$\geq +1023$	$+1023$	$\geq +16383$
E_{min}	-126	≤ -1022	-1022	≤ -16382
Exponent *bias*	$+127$	unspecified	$+1023$	unspecified
Exponent width in bits	8	≥ 11	11	≥ 15
Format width in bits	32	≥ 43	64	≥ 79

Table E.1 Summary of Format Parameters

3. If $0 < e < 255$, then $v = (-1)^s 2^{e-127}(1 \bullet f)$
4. If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{e-126}(0 \bullet f)$ (denormalized numbers)
5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

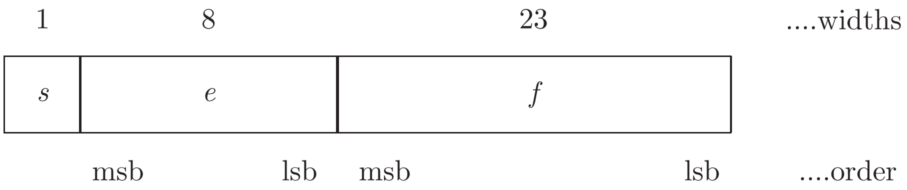

Figure E.1 Single Format. `msb` means *most significant bit*, `lsb` means *least significant bit*

E.1.2 Double

A 64-bit double format number X is divided as shown in Fig. E.2. The value v of X is inferred from its constituent fields thus

1. If $e = 2047$ and $f \neq 0$, then v is NaN regardless of s
2. If $e = 2047$ and $f = 0$, then $v = (-1)^s \infty$
3. If $0 < e < 2047$, then $v = (-1)^s 2^{e-1023}(1 \bullet f)$
4. If $e = 0$ and $f \neq 0$, then $v = (-1)^s 2^{e-1022}(0 \bullet f)$ (denormalized numbers)

5. If $e = 0$ and $f = 0$, then $v = (-1)^s 0$ (zero)

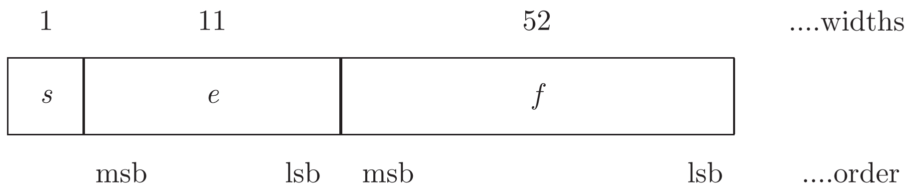

Figure E.2 Double Format. `msb` means *most significant bit*, `lsb` means *least significant bit*

E.2 Byte Patterns

Table E.2 shows the types of IEEE floating-point value, whether regular or special, corresponding to all double and single precision hexadecimal byte patterns.

IEEE value	Double Precision	Single Precision
+0	0000000000000000	00000000
denormalized	0000000000000001	00000001
	to	to
	000FFFFFFFFFFFFF	007FFFFF
positive underflow	0010000000000000	00800000
positive numbers	0010000000000001	00800001
	to	to
	7FEFFFFFFFFFFFFE	7F7FFFFE
positive overflow	7FEFFFFFFFFFFFFF	7F7FFFFF
$+\infty$	7FF0000000000000	7F800000
NaN[1]	7FF0000000000001	7F800001
	to	to
	7FFFFFFFFFFFFFFF	7FFFFFFF
-0	8000000000000000	80000000
negative	8000000000000001	80000001
denormalized	to	to
	800FFFFFFFFFFFFF	807FFFFF
negative underflow	8010000000000000	80800000
negative numbers	8010000000000001	80800001
	to	to
	FFEFFFFFFFFFFFFE	FF7FFFFE
negative overflow	FFEFFFFFFFFFFFFF	FF7FFFFF
$-\infty$	FFF0000000000000	FF800000
NaN[1]	FFF0000000000001	FF800001
	to	to
	FFFFFFFFFFFFFFFF	FFFFFFFF

[1] Certain values *may* be designated as *quiet* NaN (no diagnostic when used) or *signaling* (produces diagnostic when used) by particular implementations.

Table E.2 IEEE Floating-Point Formats

Appendix F

Reserved Extension Type Names

This Appendix is not part of the ~~NOST~~ *FITS* standard, but is included for informational purposes. It describes the extension type names registered as of the date this standard was issued.) A current list is available from the *FITS* Support Office web site at `http://fits.gsfc.nasa.gov`.

F.1 Standard Extensions

These 3 extension types have been approved by the IAUFWG and are defined in §7 of this standard document as well as in the indicated *Astronomy* and Astrophysics journal articles.

- `'IMAGE␣␣␣'` – This extension type provides a means of storing a multi-dimensional array similar to that of the *FITS* primary header and data unit. Approved as a standard extension in 1994 [8].
- `'TABLE␣␣␣'` – This ASCII table extension type contains rows and columns of data entries expressed as ASCII characters. Approved as a standard extension in 1988 [5].
- `'BINTABLE'` – This binary table extension type provides a more flexible and efficient means of storing data structures than is provided by the `TABLE` extension type. The table rows can contain a mixture of numerical, logical and character data entries. In addition, each entry is allowed to be a single dimensioned array. Numeric data are kept in binary formats. Approved as a standard extension in 1994 [9].

F.2 Conforming Extensions

These conventions meet the requirements for a conforming extension as defined in in §3.4.1 of this standard, however they have not been formally approved or endorsed by the IAUFWG.

- `'IUEIMAGE'` – This name was given to the prototype of the `IMAGE` extension type and was primarily used in the IUE project data archive from approximately 1992 to 1994. Except for the name, the format is identical to the `IMAGE` extension.
- `'A3DTABLE'` – This name was given to the prototype of the `BINTABLE` extension type and was primarily used in the AIPS data processing system developed at NRAO from about 1987 until it was replaced by `BINTABLE` in the early 1990s. The format is defined in the 'Going AIPS' manual [23], Chapter 14. It is very similar to the `BINTABLE` type except that it does not support the variable-length array convention.
- `'FOREIGN'` – This extension type is used to put a *FITS* wrapper about an arbitrary file, allowing a file or tree of files to be wrapped up in *FITS* and later restored to disk. A full description of this extension type is given in the Registry of *FITS* conventions on the *FITS* Support Office web site.
- `'DUMP␣␣␣␣'` – This extension type can be used to store a stream of binary data values. The only known use of this extension type is to record telemetry header packets for data from the Hinode mission. The more general `FOREIGN` extension type could also be used to store this type of data.

F.3 Other Suggested Extension Names

There have been occasional suggestions for other extension names that might be used for other specific purposes. These include a `COMPRESS` extension for storing compressed images, a `FITS` extension for hierarchically embedding entire *FITS* files within other *FITS* files, and a `FILEMARK` extension for representing the equivalent of an end-of-file mark on magnetic tape media. None of these extension types have been implemented or used in practice, therefore these names are not reserved. These extension names (or any other extension name not specifically mentioned in the previous sections of this appendix) *should not* be used in any FITS file without first registering the name with the IAU FITS Working Group.

Appendix G

MIME Types

This Appendix is not part of the *FITS* standard, but is included for informational purposes.

RFC 4047 [13] describes the registration of the Multipurpose Internet Mail Extensions (MIME) sub-types '`application/fits`' and '`image/fits`' to be used by the international astronomical community for the interchange of *FITS* files. The MIME type serves as a electronic tag or label that is transmitted along with the *FITS* file that tells the receiving application what type of file is being transmitted. The remainder of this appendix has been extracted verbatim from the RFC 4047 document.

The general nature of the full *FITS* standard requires the use of the media type '`application/fits`'. Nevertheless, the principal intent for a great many *FITS* files is to convey a single data array in the primary HDU, and such arrays are very often 2-dimensional images. Several common image viewing applications already display single-HDU *FITS* files, and the prototypes for virtual observatory projects specify that data provided by web services be conveyed by the data array in the primary HDU. These uses justify the registration of a second media type, namely '`image/fits`', for files which use the subset of the standard described by the original *FITS* standard paper. The MIME type '`image/fits`' *may* be used to describe *FITS* primary HDUs that have other than two dimensions, however it is expected that most files described as '`image/fits`' will have two-dimensional (`NAXIS` = 2) primary HDUs.

G.1 MIME type '`application/fits`'

A *FITS* file described with the media type '`application/fits`' *should* conform to the published standards for *FITS* files as determined by convention and agreement within the international *FITS* community. No other constraints are placed on the content of a file described as '`application/fits`'.

A *FITS* file described with the media type '`application/fits`' may have an arbitrary number of conforming extension HDUs that follow its mandatory primary header and data unit. The extension HDUs may be one of the standard types (`IMAGE`, TABLE, and `BINTABLE`) or any other type that satisfies the 'Requirements for Conforming Extensions' (§3.4.1). The primary HDU or any `IMAGE` extension may contain zero to 999 dimensions with zero or more pixels along each dimension.

The primary HDU may use the random groups convention, in which the dimension of the first axis is zero and the keywords `GROUPS`, PCOUNT and `GCOUNT` appear in the header. `NAXIS1` = 0 and `GROUPS` = T is the signature of random groups; see §6.

G.1.1 Recommendations for Application Writers

An application intended to handle '`application/fits`' *should* be able to provide a user with a manifest of all of the HDUs that are present in the file and with all of the keyword/value pairs from each of the HDUs.

An application intended to handle '`application/fits`' *should* be prepared to encounter extension HDUs that contain either ASCII or binary tables, and to provide a user with access to their elements.

An application which can modify *FITS* files or retrieve *FITS* files from an external service *should* be capable of writing such files to a local storage medium.

Complete interpretation of the meaning and intended use of the data in each of the HDUs typically requires the use of heuristics that attempt to ascertain which local conventions were used by the author of the *FITS* file.

As examples, files with media type '`application/fits`' might contain any of the following contents:

- An empty primary HDU (containing zero data elements) followed by a table HDU that contains a catalog of celestial objects.
- An empty primary HDU followed by a table HDU that encodes a series of time-tagged photon events from an exposure using an X-ray detector.
- An empty primary HDU followed by a series of IMAGE HDUs containing data from an exposure taken by a mosaic of CCD detectors.
- An empty primary HDU followed by a series of table HDUs that contain a snapshot of the state of a relational database.
- A primary HDU containing a single image along with keyword/value pairs of metadata.
- A primary HDU with `NAXIS1` = 0 and `GROUPS` = T followed by random groups data records of complex fringe visibilities.

G.2 MIME type 'image/fits'

A *FITS* file described with the media type '`image/fits`' *should* have a primary HDU with positive integer values for the `NAXIS` and `NAXISn` keywords, and hence *should* contain at least one pixel. Files with 4 or more non-degenerate axes (`NAXISn` > 1) *should* be described as '`application/fits`', not as '`image/fits`'. (In rare cases it may be appropriate to describe a NULL image – a dataless container for *FITS* keywords, with `NAXIS` $= 0$ or `NAXISn` $= 0$ – or an image with 4+ non- degenerate axes as '`image/fits`' but this usage is discouraged because such files may confuse simple image viewer applications.)

FITS files declared as '`image/fits`' *may* also have one or more conforming extension HDUs following their primary HDUs. These extension HDUs *may* contain standard, non-linear, world coordinate system (WCS) information in the form of tables or images. The extension HDUs *may* also contain other, non-standard metadata pertaining to the image in the primary HDU in the forms of keywords and tables.

A *FITS* file described with the media type '`image/fits`' *should* be principally intended to communicate the single data array in the primary HDU. This means that '`image/fits`' *should* not be applied to *FITS* files containing multi-exposure-frame mosaic images. Also, random groups files *must* be described as '`application/fits`' and not as '`image/fits`'.

A *FITS* file described with the media type '`image/fits`' is also valid as a file of media type '`application/fits`'. The choice of classification depends on the context and intended usage.

G.2.1 Recommendations for Application Writers

An application that is intended to handle '`image/fits`' *should* be able to provide a user with a manifest of all of the HDUs that are present in the file and with all of the keyword/value pairs from each of the HDUs. An application writer *may* choose to ignore HDUs beyond the primary HDU, but even in this case the application *should* be able to present the user with the keyword/value pairs from the primary HDU.

Note that an application intended to render '`image/fits`' for viewing by a user has significantly more responsibility than an application intended to handle, e.g., 'image/tiff' or 'image/gif'. *FITS* data arrays contain elements which typically represent the values of a physical quantity at some coordinate location. Consequently they need not contain any pixel rendering information in the form of transfer functions, and there is no mechanism for color look-up tables. An application *should* provide this functionality, either statically using a more or less sophisticated algorithm, or interactively allowing a user various degrees of choice.

Furthermore, the elements in a *FITS* data array may be integers or floating-point numbers. The dynamic range of the data array values may exceed that of the display medium and the eye, and their distribution may be highly nonuniform. Logarithmic, square-root, and quadratic transfer functions along with histogram equalization techniques have proved helpful for rendering *FITS* data arrays. Some elements of the array may have values which indicate that their data are undefined or invalid; these should be rendered distinctly. Via WCS Paper I [11] the standard permits `CTYPEn` = 'COMPLEX' to assert that a data array contains complex numbers (future revisions might admit other elements such as quaternions or general tensors).

Three-dimensional data arrays (`NAXIS` = 3 with `NAXIS1`, NAXIS2 and `NAXIS3` all greater than 1) are of special interest. Applications intended to handle '`image/fits`' *may* default to displaying the first 2D plane of such an image cube, or they *may* default to presenting such an image in a fashion akin to that used for an animated GIF, or they *may* present the data cube as a mosaic of 'thumbnail' images. The time-lapse movie-looping display technique can be effective in many instances, and application writers *should* consider offering it for all three-dimensional arrays.

An '`image/fits`' primary HDU with `NAXIS` = 1 is describing a one-dimensional entity such as a spectrum or a time series. Applications intended to handle '`image/fits`' *may* default to displaying such an image as a graphical plot rather than as a two-dimensional picture with a single row.

An application that cannot handle an image with dimensionality other than 2 *should* gracefully indicate its limitations to its users when it encounters `NAXIS` = 1 or `NAXIS` = 3 cases, while still providing access to the keyword/value pairs.

FITS files with degenerate axes (i.e., one or more `NAXISn` = 1) *may* be described as '`image/fits`', but the first axes *should* be non- degenerate (i.e., the degenerate axes *should* be the highest dimensions). An algorithm designed to render only two-dimensional images will be capable of displaying such an `NAXIS` = 3 or `NAXIS` = 4 *FITS* array that has one or two of the axes consisting of a single pixel, and an application writer *should* consider coding this capability into the application. Writers of new applications that generate *FITS* files intended to be described as '`image/fits`' *should* consider using the `WCSAXES` keyword [14] to declare the dimensionality of such degenerate axes, so that `NAXIS` can be used to convey the number of non-degenerate axes.

G.3 File Extensions

The *FITS* standard originated in the era when files were stored and exchanged via magnetic tape; it does not prescribe any nomenclature for files on disk. Various sites within the *FITS* community have long-established practices where files are presumed to be *FITS* by context. File extensions used at such sites commonly indicate content of the file instead of the data format.

In the absence of other information it is reasonably safe to presume that a file name ending in '.fits' is intended to be a *FITS* file. Nevertheless, there are other commonly used extensions; e.g., '.fit', '.fts', and many others not suitable for listing in a media type registration.

Bibliography

[Note] Where indicated, the following references are available electronically from the NASA Astrophysics Data System (ADS, `http://adswww.harvard.edu`) and/or the *FITS* Support Office (FSO, `http://fits.gsfc.nasa.gov`) web sites.

[1] IAU 1983, *Information Bulletin*, No. 49

[2] Wells, D. C., Greisen, E. W., & Harten, R. H. 1981, FITS*: A Flexible Image Transport System*, A&AS, 44, 363 (ADS, FSO)

[3] Greisen, E. W. & Harten, R. H. 1981, *An Extension of* FITS *for Small Arrays of Data*, A&AS, 44, 371 (ADS, FSO)

[4] Grosbøl, P., Harten, R. H., Greisen, E. W., & Wells, D. C. 1988, *Generalized Extensions and Blocking Factors for* FITS, A&AS, 73, 359 (ADS, FSO)

[5] Harten, R. H., Grosbøl, P., Greisen, E. W., & Wells, D. C. 1988, *The* FITS *Tables Extension*, A&AS, 73, 365 (ADS, FSO)

[6] IAU 1988, *Information Bulletin*, No. 61

[7] Wells, D. C. & Grosbøl, P. 1990, *Floating Point Agreement for* FITS (FSO)

[8] Ponz, J. D., Thompson, R. W., & Muñoz, J. R. 1994, *The* FITS *Image Extension*, A&AS, 105, 53 (ADS, FSO)

[9] Cotton, W. D., Tody, D. B., & Pence, W. D. 1995, *Binary Table Extension to* FITS, A&AS, 113, 159 (ADS, FSO)

[10] Bunclark, P. & Rots, A. 1997, *Precise re-definition of DATE-OBS Keyword encompassing the millennium* (FSO)

[11] Greisen, E. W. & Calabretta, M. R. 2002, *Representations of World Coordinates in FITS*, A&A, 395, 1061 (ADS, FSO)

[12] Calabretta, M. R. & Greisen, E. W. 2002, *Representations of celestial coordinates in FITS*, A&A 395, 1077 (ADS, FSO)

[13] RFC 4047 Allen, S. & Wells, D. 2005, *MIME Sub-type Registrations for Flexible Image Transport System (FITS)*, IETF RFC 4047, http://www.ietf.org/rfc/rfc4047.txt

[14] Greisen, E. W., Calabretta, M. R., Valdes, F. G., & Allen, S. L. 2006, *Representations of spectral coordinates in FITS*, A&A 446, 747 (ADS, FSO)

[15] Hanisch, R., et al. 2001, *Definition of the Flexible Image Transport System (FITS)*, A&A 376, 359 (ADS, FSO)

[16] RFC 2119 Bradner, S. 1997, *Key words for use in RFCs to Indicate Requirement Levels*, IETF RFC 2119, http://www.ietf.org/rfc/rfc2119.txt

[17] ISO. 2004, *Information technology – Programming languages – Fortran*, ISO/IEC 1539-1:2004 (Geneva: International Organization for Standardization)

[18] Grosbøl, P. & Wells, D. C. 1994, *Blocking of Fixed-block Sequential Media and Bitstream Devices* (FSO)

[19] McNally, D., ed. 1988, *Transactions of the IAU, Proceedings of the Twentieth General Assembly*, (Dordrecht: Kluwer)

[20] ANSI. 1977, *American National Standard for Information Processing: Code for Information Interchange*, ANSI X3.4–1977 (ISO 646) (New York: American National Standards Institute, Inc.)

[21] IEEE. 1985, *American National Standard — IEEE Standard for Binary Floating Point Arithmetic*, ANSI/IEEE 754–1985 (New York: American National Standards Institute, Inc.)

[22] Calabretta, M. R. & Roukema, B. F. 2007, *Mapping on the HEALPix grid*, A&A submitted (ADS, FSO)

[23] Cotton, W. D., et al. 1990, *Going AIPS: A Programmer's Guide to the NRAO Astronomical Image Processing System*, (Charlottsesville: NRAO)

Index