FITSIO User's Guide

A Subroutine Interface to FITS Format Files
for Fortran Programmers

Version 3.0

HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

May 2011

Contents

Chapter 1

Introduction

This document describes the Fortran-callable subroutine interface that is provided as part of the CFITSIO library (which is written in ANSI C). This is a companion document to the CFITSIO User's Guide which should be consulted for further information about the underlying CFITSIO library. In the remainder of this document, the terms FITSIO and CFITSIO are interchangeable and refer to the same library.

FITSIO/CFITSIO is a machine-independent library of routines for reading and writing data files in the FITS (Flexible Image Transport System) data format. It can also read IRAF format image files and raw binary data arrays by converting them on the fly into a virtual FITS format file. This library was written to provide a powerful yet simple interface for accessing FITS files which will run on most commonly used computers and workstations. FITSIO supports all the features described in the official NOST definition of the FITS format and can read and write all the currently defined types of extensions, including ASCII tables (TABLE), Binary tables (BINTABLE) and IMAGE extensions. The FITSIO subroutines insulate the programmer from having to deal with the complicated formatting details in the FITS file, however, it is assumed that users have a general knowledge about the structure and usage of FITS files.

The CFITSIO package was initially developed by the HEASARC (High Energy Astrophysics Science Archive Research Center) at the NASA Goddard Space Flight Center to convert various existing and newly acquired astronomical data sets into FITS format and to further analyze data already in FITS format. New features continue to be added to CFITSIO in large part due to contributions of ideas or actual code from users of the package. The Integral Science Data Center in Switzerland, and the XMM/ESTEC project in The Netherlands made especially significant contributions that resulted in many of the new features that appeared in v2.0 of CFITSIO.

The latest version of the CFITSIO source code, documentation, and example programs are available on the World-Wide Web or via anonymous ftp from:

```
http://heasarc.gsfc.nasa.gov/fitsio
ftp://legacy.gsfc.nasa.gov/software/fitsio/c
```

Any questions, bug reports, or suggested enhancements related to the CFITSIO package should be sent to the primary author:


```
Dr. William Pence                   Telephone:  (301) 286-4599
HEASARC, Code 662                   E-mail: William.D.Pence@nasa.gov
NASA/Goddard Space Flight Center
Greenbelt, MD 20771, USA
```


This User's Guide assumes that readers already have a general understanding of the definition and structure of FITS format files. Further information about FITS formats is available from the FITS Support Office at `http://fits.gsfc.nasa.gov`. In particular, the 'NOST FITS Standard' gives the authoritative definition of the FITS data format, and the 'FITS User's Guide' provides additional historical background and practical advice on using FITS files.

CFITSIO users may also be interested in the FTOOLS package of programs that can be used to manipulate and analyze FITS format files. Information about FTOOLS can be obtained on the Web or via anonymous ftp at:

```
http://heasarc.gsfc.nasa.gov/ftools
ftp://legacy.gsfc.nasa.gov/software/ftools/release
```

Chapter 2

Creating FITSIO/CFITSIO

2.1 Building the Library

To use the FITSIO subroutines one must first build the CFITSIO library, which requires a C compiler. gcc is ideal, or most other ANSI-C compilers will also work. The CFITSIO code is contained in about 40 C source files (*.c) and header files (*.h). On VAX/VMS systems 2 assembly-code files (vmsieeed.mar and vmsieeer.mar) are also needed.

The Fortran interface subroutines to the C CFITSIO routines are located in the f77_wrap1.c, through f77_wrap4.c files. These are relatively simple 'wrappers' that translate the arguments in the Fortran subroutine into the appropriate format for the corresponding C routine. This translation is performed transparently to the user by a set of C macros located in the cfortran.h file. Unfortunately cfortran.h does not support every combination of C and Fortran compilers so the Fortran interface is not supported on all platforms. (see further notes below).

A standard combination of C and Fortran compilers will be assumed by default, but one may also specify a particular Fortran compiler by doing:

```
>  setenv CFLAGS -DcompilerName=1
```

(where 'compilerName' is the name of the compiler) before running the configure command. The currently recognized compiler names are:

```
g77Fortran
IBMR2Fortran
CLIPPERFortran
pgiFortran
NAGf90Fortran
f2cFortran
hpuxFortran
apolloFortran
sunFortran
CRAYFortran
```

```
mipsFortran
DECFortran
vmsFortran
CONVEXFortran
PowerStationFortran
AbsoftUNIXFortran
AbsoftProFortran
SXFortran
```

Alternatively, one may edit the CFLAGS line in the Makefile to add the '-DcompilerName' flag after running the './configure' command.

The CFITSIO library is built on Unix systems by typing:

```
>  ./configure [--prefix=/target/installation/path]
               [--enable-sse2] [--enable-ssse3]
>  make          (or  'make shared')
>  make install  (this step is optional)
```

at the operating system prompt. The configure command customizes the Makefile for the particular system, then the 'make' command compiles the source files and builds the library. Type './configure' and not simply 'configure' to ensure that the configure script in the current directory is run and not some other system-wide configure script. The optional 'prefix' argument to configure gives the path to the directory where the CFITSIO library and include files should be installed via the later 'make install' command. For example,

```
> ./configure --prefix=/usr1/local
```

will cause the 'make install' command to copy the CFITSIO libcfitsio file to /usr1/local/lib and the necessary include files to /usr1/local/include (assuming of course that the process has permission to write to these directories).

The optional –enable-sse2 and –enable-ssse3 flags will cause configure to attempt to build CFITSIO using faster byte-swapping algorithms. See the "Optimizing Programs" section of this manual for more information about these options.

By default, the Makefile will be configured to build the set of Fortran-callable wrapper routines whose calling sequences are described later in this document.

The 'make shared' option builds a shared or dynamic version of the CFITSIO library. When using the shared library the executable code is not copied into your program at link time and instead the program locates the necessary library code at run time, normally through LD_LIBRARY_PATH or some other method. The advantages of using a shared library are:

```
1.  Less disk space if you build more than 1 program
2.  Less memory if more than one copy of a program using the shared
    library is running at the same time since the system is smart
```

```
     enough to share copies of the shared library at run time.
  3.  Possibly easier maintenance since a new version of the shared
      library can be installed without relinking all the software
      that uses it (as long as the subroutine names and calling
      sequences remain unchanged).
  4.  No run-time penalty.
```

The disadvantages are:

```
  1. More hassle at runtime.  You have to either build the programs
     specially or have LD_LIBRARY_PATH set right.
  2. There may be a slight start up penalty, depending on where you are
     reading the shared library and the program from and if your CPU is
     either really slow or really heavily loaded.
```

On HP/UX systems, the environment variable CFLAGS should be set to -Ae before running configure to enable "extended ANSI" features.

It may not be possible to statically link programs that use CFITSIO on some platforms (namely, on Solaris 2.6) due to the network drivers (which provide FTP and HTTP access to FITS files). It is possible to make both a dynamic and a static version of the CFITSIO library, but network file access will not be possible using the static version.

On VAX/VMS and ALPHA/VMS systems the make_gfloat.com command file may be executed to build the cfitsio.olb object library using the default G-floating point option for double variables. The make_dfloat.com and make_ieee.com files may be used instead to build the library with the other floating point options. Note that the getcwd function that is used in the group.c module may require that programs using CFITSIO be linked with the ALPHA$LIBRARY:VAXCRTL.OLB library. See the example link line in the next section of this document.

On Windows IBM-PC type platforms the situation is more complicated because of the wide variety of Fortran compilers that are available and because of the inherent complexities of calling the CFITSIO C routines from Fortran. Two different versions of the CFITSIO dll library are available, compiled with the Borland C++ compiler and the Microsoft Visual C++ compiler, respectively, in the files cfitsiodll_2xxx_borland.zip and cfitsiodll_3xxx_vcc.zip, where '3xxx' represents the current release number. Both these dll libraries contain a set of Fortran wrapper routines which may be compatible with some, but probably not all, available Fortran compilers. To test if they are compatible, compile the program testf77.f and try linking to these dll libraries. If these libraries do not work with a particular Fortran compiler, then there are 2 possible solutions. The first solution would be to modify the file cfortran.h for that particular combination of C and Fortran compilers, and then rebuild the CFITSIO dll library. This will require, however, a some expertise in mixed language programming. The other solution is to use the older v5.03 Fortran-77 implementation of FITSIO that is still available from the FITSIO web-site. This version is no longer supported, but it does provide the basic functions for reading and writing FITS files and should be compatible with most Fortran compilers.

CFITSIO has currently been tested on the following platforms:

```
  OPERATING SYSTEM           COMPILER
```

```
Sun OS                      gcc and cc (3.0.1)
Sun Solaris                 gcc and cc
Silicon Graphics IRIX       gcc and cc
Silicon Graphics IRIX64     MIPS
Dec Alpha OSF/1             gcc and cc
DECstation  Ultrix          gcc
Dec Alpha OpenVMS           cc
DEC VAX/VMS                 gcc and cc
HP-UX                       gcc
IBM AIX                     gcc
Linux                       gcc
MkLinux                     DR3
Windows 95/98/NT            Borland C++ V4.5
Windows 95/98/NT/ME/XP      Microsoft/Compaq Visual C++ v5.0, v6.0
Windows 95/98/NT            Cygwin gcc
OS/2                        gcc + EMX
MacOS 7.1 or greater        Metrowerks 10.+
```

CFITSIO will probably run on most other Unix platforms. Cray supercomputers are currently not supported.

2.2 Testing the Library

The CFITSIO library should be tested by building and running the testprog.c program that is included with the release. On Unix systems type:

```
% make testprog
% testprog > testprog.lis
% diff testprog.lis testprog.out
% cmp testprog.fit testprog.std
```

On VMS systems, (assuming cc is the name of the C compiler command), type:

```
$ cc testprog.c
$ link testprog, cfitsio/lib, alpha$library:vaxcrtl/lib
$ run testprog
```

The testprog program should produce a FITS file called 'testprog.fit' that is identical to the 'testprog.std' FITS file included with this release. The diagnostic messages (which were piped to the file testprog.lis in the Unix example) should be identical to the listing contained in the file testprog.out. The 'diff' and 'cmp' commands shown above should not report any differences in the files. (There may be some minor formatting differences, such as the presence or absence of leading zeros, or 3 digit exponents in numbers, which can be ignored).

The Fortran wrappers in CFITSIO may be tested with the testf77 program. On Unix systems the fortran compilation and link command may be called 'f77' or 'g77', depending on the system.

```
  % f77 -o testf77 testf77.f -L. -lcfitsio -lnsl -lsocket
or
  % f77 -f -o testf77 testf77.f -L. -lcfitsio    (under SUN O/S)
or
  % f77 -o testf77 testf77.f -Wl,-L. -lcfitsio -lm -lnsl -lsocket (HP/UX)
or
  % g77 -o testf77 -s testf77.f -lcfitsio -lcc_dynamic -lncurses (Mac OS-X)

  % testf77 > testf77.lis
  % diff testf77.lis testf77.out
  % cmp testf77.fit testf77.std
```

On machines running SUN O/S, Fortran programs must be compiled with the '-f' option to force double precision variables to be aligned on 8-byte boundaries to make the fortran-declared variables compatible with C. A similar compiler option may be required on other platforms. Failing to use this option may cause the program to crash on FITSIO routines that read or write double precision variables.

On Windows platforms, linking Fortran programs with a C library often depends on the particular compilers involved. Some users have found the following commands work when using the Intel Fortran compiler:

```
ifort /libs.dll cfitsio.lib /MD testf77.f /Gm

or possibly,

ifort /libs:dll cfitsio.lib /MD /fpp /extfpp:cfortran.h,fitsio.h
    /iface:cvf testf77.f
```

Also note that on some systems the output listing of the testf77 program may differ slightly from the testf77.std template if leading zeros are not printed by default before the decimal point when using F format.

A few other utility programs are included with CFITSIO:

```
    speed - measures the maximum throughput (in MB per second)
              for writing and reading FITS files with CFITSIO

    listhead - lists all the header keywords in any FITS file

    fitscopy - copies any FITS file (especially useful in conjunction
                 with the CFITSIO's extended input filename syntax)

    cookbook - a sample program that performs common read and
                 write operations on a FITS file.

    iter_a, iter_b, iter_c - examples of the CFITSIO iterator routine
```

The first 4 of these utility programs can be compiled and linked by typing

```
% make program_name
```

2.3 Linking Programs with FITSIO

When linking applications software with the FITSIO library, several system libraries usually need to be specified on the link comman Unix systems, the most reliable way to determine what libraries are required is to type 'make testprog' and see what libraries the configure script has added. The typical libraries that may need to be added are -lm (the math library) and -lnsl and -lsocket (needed only for FTP and HTTP file access). These latter 2 libraries are not needed on VMS and Windows platforms, because FTP file access is not currently supported on those platforms.

Note that when upgrading to a newer version of CFITSIO it is usually necessary to recompile, as well as relink, the programs that use CFITSIO, because the definitions in fitsio.h often change.

2.4 Getting Started with FITSIO

In order to effectively use the FITSIO library as quickly as possible, it is recommended that new users follow these steps:

1. Read the following 'FITS Primer' chapter for a brief overview of the structure of FITS files. This is especially important for users who have not previously dealt with the FITS table and image extensions.

2. Write a simple program to read or write a FITS file using the Basic Interface routines.

3. Refer to the cookbook.f program that is included with this release for examples of routines that perform various common FITS file operations.

4. Read Chapters 4 and 5 to become familiar with the conventions and advanced features of the FITSIO interface.

5. Scan through the more extensive set of routines that are provided in the 'Advanced Interface'. These routines perform more specialized functions than are provided by the Basic Interface routines.

2.5 Example Program

The following listing shows an example of how to use the FITSIO routines in a Fortran program. Refer to the cookbook.f program that is included with the FITSIO distribution for examples of other FITS programs.

```
      program writeimage

C     Create a FITS primary array containing a 2-D image
```

```
      integer status,unit,blocksize,bitpix,naxis,naxes(2)
      integer i,j,group,fpixel,nelements,array(300,200)
      character filename*80
      logical simple,extend

      status=0
C     Name of the FITS file to be created:
      filename='ATESTFILE.FITS'

C     Get an unused Logical Unit Number to use to create the FITS file
      call ftgiou(unit,status)

C     create the new empty FITS file
      blocksize=1
      call ftinit(unit,filename,blocksize,status)

C     initialize parameters about the FITS image (300 x 200 16-bit integers)
      simple=.true.
      bitpix=16
      naxis=2
      naxes(1)=300
      naxes(2)=200
      extend=.true.

C     write the required header keywords
      call ftphpr(unit,simple,bitpix,naxis,naxes,0,1,extend,status)

C     initialize the values in the image with a linear ramp function
      do j=1,naxes(2)
          do i=1,naxes(1)
              array(i,j)=i+j
          end do
      end do

C     write the array to the FITS file
      group=1
      fpixel=1
      nelements=naxes(1)*naxes(2)
      call ftpprj(unit,group,fpixel,nelements,array,status)

C     write another optional keyword to the header
      call ftpkyj(unit,'EXPOSURE',1500,'Total Exposure Time',status)

C     close the file and free the unit number
      call ftclos(unit, status)
      call ftfiou(unit, status)
```

```
end
```

2.6 Legal Stuff

2.7 Acknowledgments

The development of many of the powerful features in CFITSIO was made possible through collaborations with many people or organizations from around the world. The following, in particular, have made especially significant contributions:

Programmers from the Integral Science Data Center, Switzerland (namely, Jurek Borkowski, Bruce O'Neel, and Don Jennings), designed the concept for the plug-in I/O drivers that was introduced with CFITSIO 2.0. The use of 'drivers' greatly simplified the low-level I/O, which in turn made other new features in CFITSIO (e.g., support for compressed FITS files and support for IRAF format image files) much easier to implement. Jurek Borkowski wrote the Shared Memory driver, and Bruce O'Neel wrote the drivers for accessing FITS files over the network using the FTP, HTTP, and ROOT protocols.

The ISDC also provided the template parsing routines (written by Jurek Borkowski) and the hierarchical grouping routines (written by Don Jennings). The ISDC DAL (Data Access Layer) routines are layered on top of CFITSIO and make extensive use of these features.

Uwe Lammers (XMM/ESA/ESTEC, The Netherlands) designed the high-performance lexical parsing algorithm that is used to do on-the-fly filtering of FITS tables. This algorithm essentially pre-compiles the user-supplied selection expression into a form that can be rapidly evaluated for each row. Peter Wilson (RSTX, NASA/GSFC) then wrote the parsing routines used by CFITSIO based on Lammers' design, combined with other techniques such as the CFITSIO iterator routine to further enhance the data processing throughput. This effort also benefited from a much earlier lexical parsing routine that was developed by Kent Blackburn (NASA/GSFC). More recently, Craig Markwardt (NASA/GSFC) implemented additional functions (median, average, stddev) and other enhancements to the lexical parser.

The CFITSIO iterator function is loosely based on similar ideas developed for the XMM Data Access Layer.

Peter Wilson (RSTX, NASA/GSFC) wrote the complete set of Fortran-callable wrappers for all the CFITSIO routines, which in turn rely on the CFORTRAN macro developed by Burkhard Burow.

The syntax used by CFITSIO for filtering or binning input FITS files is based on ideas developed for the AXAF Science Center Data Model by Jonathan McDowell, Antonella Fruscione, Aneta Siemiginowska and Bill Joye. See http://heasarc.gsfc.nasa.gov/docs/journal/axaf7.html for further description of the AXAF Data Model.

The file decompression code were taken directly from the gzip (GNU zip) program developed by Jean-loup Gailly and others.

Doug Mink, SAO, provided the routines for converting IRAF format images into FITS format.

Martin Reinecke (Max Planck Institute, Garching)) provided the modifications to cfortran.h that are necessary to support 64-bit integer values when calling C routines from fortran programs. The cfortran.h macros were originally developed by Burkhard Burow (CERN).

Julian Taylor (ESO, Garching) provided the fast byte-swapping algorithms that use the SSE2 and SSSE3 machine instructions available on x86_64 CPUs.

In addition, many other people have made valuable contributions to the development of CFITSIO. These include (with apologies to others that may have inadvertently been omitted):

Steve Allen, Carl Akerlof, Keith Arnaud, Morten Krabbe Barfoed, Kent Blackburn, G Bodammer, Romke Bontekoe, Lucio Chiappetti, Keith Costorf, Robin Corbet, John Davis, Richard Fink, Ning Gan, Emily Greene, Joe Harrington, Cheng Ho, Phil Hodge, Jim Ingham, Yoshitaka Ishisaki, Diab Jerius, Mark Levine, Todd Karakaskian, Edward King, Scott Koch, Claire Larkin, Rob Managan, Eric Mandel, John Mattox, Carsten Meyer, Emi Miyata, Stefan Mochnacki, Mike Noble, Oliver Oberdorf, Clive Page, Arvind Parmar, Jeff Pedelty, Tim Pearson, Maren Purves, Scott Randall, Chris Rogers, Arnold Rots, Barry Schlesinger, Robin Stebbins, Andrew Szymkowiak, Allyn Tennant, Peter Teuben, James Theiler, Doug Tody, Shiro Ueno, Steve Walton, Archie Warnock, Alan Watson, Dan Whipple, Wim Wimmers, Peter Young, Jianjun Xu, and Nelson Zarate.

Chapter 3

A FITS Primer

This section gives a brief overview of the structure of FITS files. Users should refer to the documentation available from the NOST, as described in the introduction, for more detailed information on FITS formats.

FITS was first developed in the late 1970's as a standard data interchange format between various astronomical observatories. Since then FITS has become the defacto standard data format supported by most astronomical data analysis software packages.

A FITS file consists of one or more Header + Data Units (HDUs), where the first HDU is called the 'Primary HDU', or 'Primary Array'. The primary array contains an N-dimensional array of pixels, such as a 1-D spectrum, a 2-D image, or a 3-D data cube. Six different primary datatypes are supported: Unsigned 8-bit bytes, 16, 32, and 64-bit signed integers, and 32 and 64-bit floating point reals. FITS also has a convention for storing unsigned integers (see the later section entitled 'Unsigned Integers' for more details). The primary HDU may also consist of only a header with a null array containing no data pixels.

Any number of additional HDUs may follow the primary array; these additional HDUs are called FITS 'extensions'. There are currently 3 types of extensions defined by the FITS standard:

- Image Extension - a N-dimensional array of pixels, like in a primary array
- ASCII Table Extension - rows and columns of data in ASCII character format
- Binary Table Extension - rows and columns of data in binary representation

In each case the HDU consists of an ASCII Header Unit followed by an optional Data Unit. For historical reasons, each Header or Data unit must be an exact multiple of 2880 8-bit bytes long. Any unused space is padded with fill characters (ASCII blanks or zeros).

Each Header Unit consists of any number of 80-character keyword records or 'card images' which have the general form:

```
KEYNAME = value / comment string
NULLKEY =       / comment: This keyword has no value
```

The keyword names may be up to 8 characters long and can only contain uppercase letters, the digits 0-9, the hyphen, and the underscore character. The keyword name is (usually) followed by an equals sign and a space character (=) in columns 9 - 10 of the record, followed by the value of the keyword which may be either an integer, a floating point number, a character string (enclosed in single quotes), or a boolean value (the letter T or F). A keyword may also have a null or undefined value if there is no specified value string, as in the second example.

The last keyword in the header is always the 'END' keyword which has no value or comment fields. There are many rules governing the exact format of a keyword record (see the NOST FITS Standard) so it is better to rely on standard interface software like FITSIO to correctly construct or to parse the keyword records rather than try to deal directly with the raw FITS formats.

Each Header Unit begins with a series of required keywords which depend on the type of HDU. These required keywords specify the size and format of the following Data Unit. The header may contain other optional keywords to describe other aspects of the data, such as the units or scaling values. Other COMMENT or HISTORY keywords are also frequently added to further document the data file.

The optional Data Unit immediately follows the last 2880-byte block in the Header Unit. Some HDUs do not have a Data Unit and only consist of the Header Unit.

If there is more than one HDU in the FITS file, then the Header Unit of the next HDU immediately follows the last 2880-byte block of the previous Data Unit (or Header Unit if there is no Data Unit).

The main required keywords in FITS primary arrays or image extensions are:

- BITPIX – defines the datatype of the array: 8, 16, 32, 64, -32, -64 for unsigned 8–bit byte, 16–bit signed integer, 32–bit signed integer, 64–bit signed integer, 32–bit IEEE floating point, and 64–bit IEEE double precision floating point, respectively.
- NAXIS – the number of dimensions in the array, usually 0, 1, 2, 3, or 4.
- NAXISn – (n ranges from 1 to NAXIS) defines the size of each dimension.

FITS tables start with the keyword XTENSION = 'TABLE' (for ASCII tables) or XTENSION = 'BINTABLE' (for binary tables) and have the following main keywords:

- TFIELDS – number of fields or columns in the table
- NAXIS2 – number of rows in the table
- TTYPEn – for each column (n ranges from 1 to TFIELDS) gives the name of the column
- TFORMn – the datatype of the column
- TUNITn – the physical units of the column (optional)

Users should refer to the FITS Support Office at `http://fits.gsfc.nasa.gov` for further information about the FITS format and related software packages.

Chapter 4

FITSIO Conventions and Guidelines

4.1 CFITSIO Size Limitations

CFITSIO places few restrictions on the size of FITS files that it reads or writes. There are a few limits, however, which may affect some extreme cases:

1. The maximum number of FITS files that may be simultaneously opened by CFITSIO is set by NMAXFILES as defined in fitsio2.h. It is currently set = 300 by default. CFITSIO will allocate about 80 * NMAXFILES bytes of memory for internal use. Note that the underlying C compiler or operating system, may have a smaller limit on the number of opened files. The C symbolic constant FOPEN_MAX is intended to define the maximum number of files that may open at once (including any other text or binary files that may be open, not just FITS files). On some systems it has been found that gcc supports a maximum of 255 opened files.

2. By default, CFITSIO can handle FITS files up to 2.1 GB in size (2**31 bytes). This file size limit is often imposed by 32-bit operating systems. More recently, as 64-bit operating systems become more common, an industry-wide standard (at least on Unix systems) has been developed to support larger sized files (see http://ftp.sas.com/standards/large.file/). Starting with version 2.1 of CFITSIO, larger FITS files up to 6 terabytes in size may be read and written on supported platforms. In order to support these larger files, CFITSIO must be compiled with the ’-D_LARGEFILE_SOURCE’ and ‘-D_FILE_OFFSET_BITS=64’ compiler flags. Some platforms may also require the ‘-D_LARGE_FILES’ compiler flag. This causes the compiler to allocate 8-bytes instead of 4-bytes for the ‘off_t’ datatype which is used to store file offset positions. It appears that in most cases it is not necessary to also include these compiler flags when compiling programs that link to the CFITSIO library.

If CFITSIO is compiled with the -D_LARGEFILE_SOURCE and -D_FILE_OFFSET_BITS=64 flags on a platform that supports large files, then it can read and write FITS files that contain up to 2**31 2880-byte FITS records, or approximately 6 terabytes in size. It is still required that the value of the NAXISn and PCOUNT keywords in each extension be within the range of a signed 4-byte integer (max value = 2,147,483,648). Thus, each dimension of an image (given by the NAXISn keywords), the total width of a table (NAXIS1 keyword), the number of rows in a table (NAXIS2 keyword), and the total size of the variable-length array heap in binary tables (PCOUNT keyword) must be less than this limit.

Currently, support for large files within CFITSIO has been tested on the Linux, Solaris, and IBM AIX operating systems.

4.2 Multiple Access to the Same FITS File

CFITSIO supports simultaneous read and write access to multiple HDUs in the same FITS file. Thus, one can open the same FITS file twice within a single program and move to 2 different HDUs in the file, and then read and write data or keywords to the 2 extensions just as if one were accessing 2 completely separate FITS files. Since in general it is not possible to physically open the same file twice and then expect to be able to simultaneously (or in alternating succession) write to 2 different locations in the file, CFITSIO recognizes when the file to be opened (in the call to fits_open_file) has already been opened and instead of actually opening the file again, just logically links the new file to the old file. (This only applies if the file is opened more than once within the same program, and does not prevent the same file from being simultaneously opened by more than one program). Then before CFITSIO reads or writes to either (logical) file, it makes sure that any modifications made to the other file have been completely flushed from the internal buffers to the file. Thus, in principle, one could open a file twice, in one case pointing to the first extension and in the other pointing to the 2nd extension and then write data to both extensions, in any order, without danger of corrupting the file, There may be some efficiency penalties in doing this however, since CFITSIO has to flush all the internal buffers related to one file before switching to the other, so it would still be prudent to minimize the number of times one switches back and forth between doing I/O to different HDUs in the same file.

4.3 Current Header Data Unit (CHDU)

In general, a FITS file can contain multiple Header Data Units, also called extensions. CFITSIO only operates within one HDU at any given time, and the currently selected HDU is called the Current Header Data Unit (CHDU). When a FITS file is first created or opened the CHDU is automatically defined to be the first HDU (i.e., the primary array). CFITSIO routines are provided to move to and open any other existing HDU within the FITS file or to append or insert a new HDU in the FITS file which then becomes the CHDU.

4.4 Subroutine Names

All FITSIO subroutine names begin with the letters 'ft' to distinguish them from other subroutines and are 5 or 6 characters long. Users should not name their own subroutines beginning with 'ft' to avoid conflicts. (The SPP interface routines all begin with 'fs'). Subroutines which read or get information from the FITS file have names beginning with 'ftg...'. Subroutines which write or put information into the FITS file have names beginning with 'ftp...'.

4.5 Subroutine Families and Datatypes

Many of the subroutines come in families which differ only in the datatype of the associated parameter(s) . The datatype of these subroutines is indicated by the last letter of the subroutine name (e.g., 'j' in 'ftpkyj') as follows:

```
x - bit
b - character*1 (unsigned byte)
i - short integer (I*2)
j - integer (I*4, 32-bit integer)
k - long long integer (I*8, 64-bit integer)
e - real exponential floating point (R*4)
f - real fixed-format floating point (R*4)
d - double precision real floating-point (R*8)
g - double precision fixed-format floating point (R*8)
c - complex reals (pairs of R*4 values)
m - double precision complex (pairs of R*8 values)
l - logical (L*4)
s - character string
```

When dealing with the FITS byte datatype, it is important to remember that the raw values (before any scaling by the BSCALE and BZERO, or TSCALn and TZEROn keyword values) in byte arrays (BITPIX = 8) or byte columns (TFORMn = 'B') are interpreted as unsigned bytes with values ranging from 0 to 255. Some Fortran compilers support a non-standard byte datatype such as INTEGER*1, LOGICAL*1, or BYTE, which can sometimes be used instead of CHARACTER*1 variables. Many machines permit passing a numeric datatype (such as INTEGER*1) to the FITSIO subroutines which are expecting a CHARACTER*1 datatype, but this technically violates the Fortran-77 standard and is not supported on all machines (e.g., on a VAX/VMS machine one must use the VAX-specific %DESCR function).

One feature of the CFITSIO routines is that they can operate on a 'X' (bit) column in a binary table as though it were a 'B' (byte) column. For example a '11X' datatype column can be interpreted the same as a '2B' column (i.e., 2 unsigned 8-bit bytes). In some instances, it can be more efficient to read and write whole bytes at a time, rather than reading or writing each individual bit.

The double precision complex datatype is not a standard Fortran-77 datatype. If a particular Fortran compiler does not directly support this datatype, then one may instead pass an array of pairs of double precision values to these subroutines. The first value in each pair is the real part, and the second is the imaginary part.

4.6 Implicit Data Type Conversion

The FITSIO routines that read and write numerical data can perform implicit data type conversion. This means that the data type of the variable or array in the program does not need to be the same as the data type of the value in the FITS file. Data type conversion is supported for numerical and string data types (if the string contains a valid number enclosed in quotes) when reading a FITS header keyword value and for numeric values when reading or writing values in the primary array or a table column. CFITSIO returns status = NUM_OVERFLOW if the converted data value exceeds the range of the output data type. Implicit data type conversion is not supported within binary tables for string, logical, complex, or double complex data types.

In addition, any table column may be read as if it contained string values. In the case of numeric columns the returned string will be formatted using the TDISPn display format if it exists.

4.7 Data Scaling

When reading numerical data values in the primary array or a table column, the values will be scaled automatically by the BSCALE and BZERO (or TSCALn and TZEROn) header keyword values if they are present in the header. The scaled data that is returned to the reading program will have

```
output value = (FITS value) * BSCALE + BZERO
```

(a corresponding formula using TSCALn and TZEROn is used when reading from table columns). In the case of integer output values the floating point scaled value is truncated to an integer (not rounded to the nearest integer). The ftpscl and fttscl subroutines may be used to override the scaling parameters defined in the header (e.g., to turn off the scaling so that the program can read the raw unscaled values from the FITS file).

When writing numerical data to the primary array or to a table column the data values will generally be automatically inversely scaled by the value of the BSCALE and BZERO (or TSCALn and TZEROn) header keyword values if they they exist in the header. These keywords must have been written to the header before any data is written for them to have any effect. Otherwise, one may use the ftpscl and fttscl subroutines to define or override the scaling keywords in the header (e.g., to turn off the scaling so that the program can write the raw unscaled values into the FITS file). If scaling is performed, the inverse scaled output value that is written into the FITS file will have

```
FITS value = ((input value) - BZERO) / BSCALE
```

(a corresponding formula using TSCALn and TZEROn is used when writing to table columns). Rounding to the nearest integer, rather than truncation, is performed when writing integer datatypes to the FITS file.

4.8 Error Status Values and the Error Message Stack

The last parameter in nearly every FITSIO subroutine is the error status value which is both an input and an output parameter. A returned positive value for this parameter indicates an error was detected. A listing of all the FITSIO status code values is given at the end of this document.

The FITSIO library uses an 'inherited status' convention for the status parameter which means that if a subroutine is called with a positive input value of the status parameter, then the subroutine will exit immediately without changing the value of the status parameter. Thus, if one passes the status value returned from each FITSIO routine as input to the next FITSIO subroutine, then whenever an error is detected all further FITSIO processing will cease. This convention can simplify the error checking in application programs because it is not necessary to check the value of the status parameter after every single FITSIO subroutine call. If a program contains a sequence of several FITSIO calls, one can just check the status value after the last call. Since the returned status values are generally distinctive, it should be possible to determine which subroutine originally returned the error status.

FITSIO also maintains an internal stack of error messages (80-character maximum length) which in many cases provide a more detailed explanation of the cause of the error than is provided by the error status number alone. It is recommended that the error message stack be printed out whenever a program detects a FITSIO error. To do this, call the FTGMSG routine repeatedly to get the successive messages on the stack. When the stack is empty FTGMSG will return a blank string. Note that this is a 'First In – First Out' stack, so the oldest error message is returned first by ftgmsg.

4.9 Variable-Length Array Facility in Binary Tables

FITSIO provides easy-to-use support for reading and writing data in variable length fields of a binary table. The variable length columns have TFORMn keyword values of the form '1Pt(len)' or '1Qt(len)' where 't' is the datatype code (e.g., I, J, E, D, etc.) and 'len' is an integer specifying the maximum length of the vector in the table. If the value of 'len' is not specified when the table is created (e.g., if the TFORM keyword value is simply specified as '1PE' instead of '1PE(400)), then FITSIO will automatically scan the table when it is closed to determine the maximum length of the vector and will append this value to the TFORMn value.

The same routines which read and write data in an ordinary fixed length binary table extension are also used for variable length fields, however, the subroutine parameters take on a slightly different interpretation as described below.

All the data in a variable length field is written into an area called the 'heap' which follows the main fixed-length FITS binary table. The size of the heap, in bytes, is specified with the PCOUNT keyword in the FITS header. When creating a new binary table, the initial value of PCOUNT should usually be set to zero. FITSIO will recompute the size of the heap as the data is written and will automatically update the PCOUNT keyword value when the table is closed. When writing variable length data to a table, CFITSIO will automatically extend the size of the heap area if necessary, so that any following HDUs do not get overwritten.

By default the heap data area starts immediately after the last row of the fixed-length table. This default starting location may be overridden by the THEAP keyword, but this is not recommended. If additional rows of data are added to the table, CFITSIO will automatically shift the the heap down to make room for the new rows, but it is obviously be more efficient to initially create the table with the necessary number of blank rows, so that the heap does not needed to be constantly moved.

When writing to a variable length field, the entire array of values for a given row of the table must be written with a single call to FTPCLx. The total length of the array is calculated from

(NELEM+FELEM-1). One cannot append more elements to an existing field at a later time; any attempt to do so will simply overwrite all the data which was previously written. Note also that the new data will be written to a new area of the heap and the heap space used by the previous write cannot be reclaimed. For this reason it is advised that each row of a variable length field only be written once. An exception to this general rule occurs when setting elements of an array as undefined. One must first write a dummy value into the array with FTPCLx, and then call FTPCLU to flag the desired elements as undefined. (Do not use the FTPCNx family of routines with variable length fields). Note that the rows of a table, whether fixed or variable length, do not have to be written consecutively and may be written in any order.

When writing to a variable length ASCII character field (e.g., TFORM = '1PA') only a single character string written. FTPCLS writes the whole length of the input string (minus any trailing blank characters), thus the NELEM and FELEM parameters are ignored. If the input string is completely blank then FITSIO will write one blank character to the FITS file. Similarly, FTGCVS and FTGCFS read the entire string (truncated to the width of the character string argument in the subroutine call) and also ignore the NELEM and FELEM parameters.

The FTPDES subroutine is useful in situations where multiple rows of a variable length column have the identical array of values. One can simply write the array once for the first row, and then use FTPDES to write the same descriptor values into the other rows (use the FTGDES routine to read the first descriptor value); all the rows will then point to the same storage location thus saving disk space.

When reading from a variable length array field one can only read as many elements as actually exist in that row of the table; reading does not automatically continue with the next row of the table as occurs when reading an ordinary fixed length table field. Attempts to read more than this will cause an error status to be returned. One can determine the number of elements in each row of a variable column with the FTGDES subroutine.

4.10 Support for IEEE Special Values

The ANSI/IEEE-754 floating-point number standard defines certain special values that are used to represent such quantities as Not-a-Number (NaN), denormalized, underflow, overflow, and infinity. (See the Appendix in the NOST FITS standard or the NOST FITS User's Guide for a list of these values). The FITSIO subroutines that read floating point data in FITS files recognize these IEEE special values and by default interpret the overflow and infinity values as being equivalent to a NaN, and convert the underflow and denormalized values into zeros. In some cases programmers may want access to the raw IEEE values, without any modification by FITSIO. This can be done by calling the FTGPVx or FTGCVx routines while specifying 0.0 as the value of the NULLVAL parameter. This will force FITSIO to simply pass the IEEE values through to the application program, without any modification. This does not work for double precision values on VAX/VMS machines, however, where there is no easy way to bypass the default interpretation of the IEEE special values. This is also not supported when reading floating-point images that have been compressed with the FITS tiled image compression convention that is discussed in section 5.6; the pixels values in tile compressed images are represented by scaled integers, and a reserved integer value (not a NaN) is used to represent undefined pixels.

4.11 When the Final Size of the FITS HDU is Unknown

It is not required to know the total size of a FITS data array or table before beginning to write the data to the FITS file. In the case of the primary array or an image extension, one should initially create the array with the size of the highest dimension (largest NAXISn keyword) set to a dummy value, such as 1. Then after all the data have been written and the true dimensions are known, then the NAXISn value should be updated using the fits_ update_key routine before moving to another extension or closing the FITS file.

When writing to FITS tables, CFITSIO automatically keeps track of the highest row number that is written to, and will increase the size of the table if necessary. CFITSIO will also automatically insert space in the FITS file if necessary, to ensure that the data 'heap', if it exists, and/or any additional HDUs that follow the table do not get overwritten as new rows are written to the table.

As a general rule it is best to specify the initial number of rows = 0 when the table is created, then let CFITSIO keep track of the number of rows that are actually written. The application program should not manually update the number of rows in the table (as given by the NAXIS2 keyword) since CFITSIO does this automatically. If a table is initially created with more than zero rows, then this will usually be considered as the minimum size of the table, even if fewer rows are actually written to the table. Thus, if a table is initially created with NAXIS2 = 20, and CFITSIO only writes 10 rows of data before closing the table, then NAXIS2 will remain equal to 20. If however, 30 rows of data are written to this table, then NAXIS2 will be increased from 20 to 30. The one exception to this automatic updating of the NAXIS2 keyword is if the application program directly modifies the value of NAXIS2 (up or down) itself just before closing the table. In this case, CFITSIO does not update NAXIS2 again, since it assumes that the application program must have had a good reason for changing the value directly. This is not recommended, however, and is only provided for backward compatibility with software that initially creates a table with a large number of rows, than decreases the NAXIS2 value to the actual smaller value just before closing the table.

4.12 Local FITS Conventions supported by FITSIO

CFITSIO supports several local FITS conventions which are not defined in the official NOST FITS standard and which are not necessarily recognized or supported by other FITS software packages. Programmers should be cautious about using these features, especially if the FITS files that are produced are expected to be processed by other software systems which do not use the CFITSIO interface.

4.12.1 Support for Long String Keyword Values.

The length of a standard FITS string keyword is limited to 68 characters because it must fit entirely within a single FITS header keyword record. In some instances it is necessary to encode strings longer than this limit, so FITSIO supports a local convention in which the string value is continued over multiple keywords. This continuation convention uses an ampersand character at the end of each substring to indicate that it is continued on the next keyword, and the continuation keywords all have the name CONTINUE without an equal sign in column 9. The string value may be continued in this way over as many additional CONTINUE keywords as is required. The following lines illustrate this continuation convention which is used in the value of the STRKEY keyword:

```
LONGSTRN= 'OGIP 1.0'           / The OGIP Long String Convention may be used.
STRKEY  = 'This is a very long string keyword&'  / Optional Comment
CONTINUE  ' value that is continued over 3 keywords in the &  '
CONTINUE  'FITS header.' / This is another optional comment.
```

It is recommended that the LONGSTRN keyword, as shown here, always be included in any HDU that uses this longstring convention. A subroutine called FTPLSW has been provided in CFITSIO to write this keyword if it does not already exist.

This long string convention is supported by the following FITSIO subroutines that deal with string-valued keywords:

```
      ftgkys - read a string keyword
      ftpkls - write (append) a string keyword
      ftikls - insert a string keyword
      ftmkls - modify the value of an existing string keyword
      ftukls - update an existing keyword, or write a new keyword
      ftdkey - delete a keyword
```

These routines will transparently read, write, or delete a long string value in the FITS file, so programmers in general do not have to be concerned about the details of the convention that is used to encode the long string in the FITS header. When reading a long string, one must ensure that the character string parameter used in these subroutine calls has been declared long enough to hold the entire string, otherwise the returned string value will be truncated.

Note that the more commonly used FITSIO subroutine to write string valued keywords (FTPKYS) does NOT support this long string convention and only supports strings up to 68 characters in length. This has been done deliberately to prevent programs from inadvertently writing keywords using this non-standard convention without the explicit intent of the programmer or user. The FTPKLS subroutine must be called instead to write long strings. This routine can also be used to write ordinary string values less than 68 characters in length.

4.12.2 Arrays of Fixed-Length Strings in Binary Tables

CFITSIO supports 2 ways to specify that a character column in a binary table contains an array of fixed-length strings. The first way, which is officially supported by the FITS Standard document, uses the TDIMn keyword. For example, if TFORMn = '60A' and TDIMn = '(12,5)' then that column will be interpreted as containing an array of 5 strings, each 12 characters long.

FITSIO also supports a local convention for the format of the TFORMn keyword value of the form 'rAw' where 'r' is an integer specifying the total width in characters of the column, and 'w' is an integer specifying the (fixed) length of an individual unit string within the vector. For example, TFORM1 = '120A10' would indicate that the binary table column is 120 characters wide and consists of 12 10-character length strings. This convention is recognized by the FITSIO subroutines that read or write strings in binary tables. The Binary Table definition document specifies that other optional characters may follow the datatype code in the TFORM keyword, so this local convention is in compliance with the FITS standard, although other FITS readers are not required to recognize this convention.

The Binary Table definition document that was approved by the IAU in 1994 contains an appendix describing an alternate convention for specifying arrays of fixed or variable length strings in a binary table character column (with the form 'rA:SSTRw/nnn)'. This appendix was not officially voted on by the IAU and hence is still provisional. FITSIO does not currently support this proposal.

4.12.3 Keyword Units Strings

One deficiency of the current FITS Standard is that it does not define a specific convention for recording the physical units of a keyword value. The TUNITn keyword can be used to specify the physical units of the values in a table column, but there is no analogous convention for keyword values. The comment field of the keyword is often used for this purpose, but the units are usually not specified in a well defined format that FITS readers can easily recognize and extract.

To solve this deficiency, FITSIO uses a local convention in which the keyword units are enclosed in square brackets as the first token in the keyword comment field; more specifically, the opening square bracket immediately follows the slash '/' comment field delimiter and a single space character. The following examples illustrate keywords that use this convention:

```
EXPOSURE=               1800.0 / [s] elapsed exposure time
V_HELIO =                16.23 / [km s**(-1)] heliocentric velocity
LAMBDA  =                5400. / [angstrom] central wavelength
FLUX    = 4.9033487787637465E-30 / [J/cm**2/s] average flux
```

In general, the units named in the IAU(1988) Style Guide are recommended, with the main exception that the preferred unit for angle is 'deg' for degrees.

The FTPUNT and FTGUNT subroutines in FITSIO write and read, respectively, the keyword unit strings in an existing keyword.

4.12.4 HIERARCH Convention for Extended Keyword Names

CFITSIO supports the HIERARCH keyword convention which allows keyword names that are longer then 8 characters and may contain the full range of printable ASCII text characters. This convention was developed at the European Southern Observatory (ESO) to support hierarchical FITS keyword such as:

```
HIERARCH ESO INS FOCU POS = -0.00002500 / Focus position
```

Basically, this convention uses the FITS keyword 'HIERARCH' to indicate that this convention is being used, then the actual keyword name (`'ESO INS FOCU POS'` in this example) begins in column 10 and can contain any printable ASCII text characters, including spaces. The equals sign marks the end of the keyword name and is followed by the usual value and comment fields just as in standard FITS keywords. Further details of this convention are described at http://arcdev.hq.eso.org/dicb/dicd/dic-1-1.4.html (search for HIERARCH).

This convention allows a much broader range of keyword names than is allowed by the FITS Standard. Here are more examples of such keywords:

```
HIERARCH LongKeyword = 47.5 / Keyword has > 8 characters, and mixed case
HIERARCH XTE$TEMP = 98.6 / Keyword contains the '$' character
HIERARCH Earth is a star = F / Keyword contains embedded spaces
```

CFITSIO will transparently read and write these keywords, so application programs do not in general need to know anything about the specific implementation details of the HIERARCH convention. In particular, application programs do not need to specify the 'HIERARCH' part of the keyword name when reading or writing keywords (although it may be included if desired). When writing a keyword, CFITSIO first checks to see if the keyword name is legal as a standard FITS keyword (no more than 8 characters long and containing only letters, digits, or a minus sign or underscore). If so it writes it as a standard FITS keyword, otherwise it uses the hierarch convention to write the keyword. The maximum keyword name length is 67 characters, which leaves only 1 space for the value field. A more practical limit is about 40 characters, which leaves enough room for most keyword values. CFITSIO returns an error if there is not enough room for both the keyword name and the keyword value on the 80-character card, except for string-valued keywords which are simply truncated so that the closing quote character falls in column 80. In the current implementation, CFITSIO preserves the case of the letters when writing the keyword name, but it is case-insensitive when reading or searching for a keyword. The current implementation allows any ASCII text character (ASCII 32 to ASCII 126) in the keyword name except for the '=' character. A space is also required on either side of the equal sign.

4.13 Optimizing Code for Maximum Processing Speed

CFITSIO has been carefully designed to obtain the highest possible speed when reading and writing FITS files. In order to achieve the best performance, however, application programmers must be careful to call the CFITSIO routines appropriately and in an efficient sequence; inappropriate usage of CFITSIO routines can greatly slow down the execution speed of a program.

The maximum possible I/O speed of CFITSIO depends of course on the type of computer system that it is running on. As a rough guide, the current generation of workstations can achieve speeds of 2 – 10 MB/s when reading or writing FITS images and similar, or slightly slower speeds with FITS binary tables. Reading of FITS files can occur at even higher rates (30MB/s or more) if the FITS file is still cached in system memory following a previous read or write operation on the same file. To more accurately predict the best performance that is possible on any particular system, a diagnostic program called "speed.c" is included with the CFITSIO distribution which can be run to approximately measure the maximum possible speed of writing and reading a test FITS file.

The following 2 sections provide some background on how CFITSIO internally manages the data I/O and describes some strategies that may be used to optimize the processing speed of software that uses CFITSIO.

4.13.1 Background Information: How CFITSIO Manages Data I/O

Many CFITSIO operations involve transferring only a small number of bytes to or from the FITS file (e.g, reading a keyword, or writing a row in a table); it would be very inefficient to physically read or write such small blocks of data directly in the FITS file on disk, therefore CFITSIO maintains a set of internal Input–Output (IO) buffers in RAM memory that each contain one FITS block (2880 bytes) of data. Whenever CFITSIO needs to access data in the FITS file, it first transfers the FITS block containing those bytes into one of the IO buffers in memory. The next time CFITSIO needs to access bytes in the same block it can then go to the fast IO buffer rather than using a much slower system disk access routine. The number of available IO buffers is determined by the NIOBUF parameter (in fitsio2.h) and is currently set to 40.

Whenever CFITSIO reads or writes data it first checks to see if that block of the FITS file is already loaded into one of the IO buffers. If not, and if there is an empty IO buffer available, then it will load that block into the IO buffer (when reading a FITS file) or will initialize a new block (when writing to a FITS file). If all the IO buffers are already full, it must decide which one to reuse (generally the one that has been accessed least recently), and flush the contents back to disk if it has been modified before loading the new block.

The one major exception to the above process occurs whenever a large contiguous set of bytes are accessed, as might occur when reading or writing a FITS image. In this case CFITSIO bypasses the internal IO buffers and simply reads or writes the desired bytes directly in the disk file with a single call to a low-level file read or write routine. The minimum threshold for the number of bytes to read or write this way is set by the MINDIRECT parameter and is currently set to 3 FITS blocks = 8640 bytes. This is the most efficient way to read or write large chunks of data and can achieve IO transfer rates of 5 – 10MB/s or greater. Note that this fast direct IO process is not applicable when accessing columns of data in a FITS table because the bytes are generally not contiguous since they are interleaved by the other columns of data in the table. This explains why the speed for accessing FITS tables is generally slower than accessing FITS images.

Given this background information, the general strategy for efficiently accessing FITS files should now be apparent: when dealing with FITS images, read or write large chunks of data at a time so that the direct IO mechanism will be invoked; when accessing FITS headers or FITS tables, on the other hand, once a particular FITS block has been loading into one of the IO buffers, try to access all the needed information in that block before it gets flushed out of the IO buffer. It is important to avoid the situation where the same FITS block is being read then flushed from a IO buffer multiple times.

The following section gives more specific suggestions for optimizing the use of CFITSIO.

4.13.2 Optimization Strategies

1. Because the data in FITS files is always stored in "big-endian" byte order, where the first byte of numeric values contains the most significant bits and the last byte contains the least significant bits, CFITSIO must swap the order of the bytes when reading or writing FITS files when running on little-endian machines (e.g., Linux and Microsoft Windows operating systems running on PCs with x86 CPUs).

On fairly new CPUs that support "SSSE3" machine instructions (e.g., starting with Intel Core 2

CPUs in 2007, and in AMD CPUs beginning in 2011) significantly faster 4-byte and 8-byte swapping algorithms are available. These faster byte swapping functions are not used by default in CFITSIO (because of the potential code portablility issues), but users can enable them on supported platforms by adding the appropriate compiler flags (-mssse3 with gcc or icc on linux) when compiling the swapproc.c source file, which will allow the compiler to generate code using the SSSE3 instruction set. A convenient way to do this is to configure the CFITSIO library with the following command:

```
  >  ./configure --enable-ssse3
```

Note, however, that a binary executable file that is created using these faster functions will only run on machines that support the SSSE3 machine instructions. It will crash on machines that do not support them.

For faster 2-byte swaps on virtually all x86-64 CPUs (even those that do not support SSSE3), a variant using only SSE2 instructions exists. SSE2 is enabled by default on x86_64 CPUs with 64-bit operating systems (and is also automatically enabled by the –enable-ssse3 flag). When running on x86_64 CPUs with 32-bit operating systems, these faster 2-byte swapping algorithms are not used by default in CFITSIO, but can be enabled explicitly with:

```
./configure --enable-sse2
```

Preliminary testing indicates that these SSSE3 and SSE2 based byte-swapping algorithms can boost the CFITSIO performance when reading or writing FITS images by 20% - 30% or more. It is important to note, however, that compiler optimization must be turned on (e.g., by using the -O1 or -O2 flags in gcc) when building programs that use these fast byte-swapping algorithms in order to reap the full benefit of the SSSE3 and SSE2 instructions; without optimization, the code may actually run slower than when using more traditional byte-swapping techniques.

2. When dealing with a FITS primary array or IMAGE extension, it is more efficient to read or write large chunks of the image at a time (at least 3 FITS blocks = 8640 bytes) so that the direct IO mechanism will be used as described in the previous section. Smaller chunks of data are read or written via the IO buffers, which is somewhat less efficient because of the extra copy operation and additional bookkeeping steps that are required. In principle it is more efficient to read or write as big an array of image pixels at one time as possible, however, if the array becomes so large that the operating system cannot store it all in RAM, then the performance may be degraded because of the increased swapping of virtual memory to disk.

3. When dealing with FITS tables, the most important efficiency factor in the software design is to read or write the data in the FITS file in a single pass through the file. An example of poor program design would be to read a large, 3-column table by sequentially reading the entire first column, then going back to read the 2nd column, and finally the 3rd column; this obviously requires 3 passes through the file which could triple the execution time of an I/O limited program. For small tables this is not important, but when reading multi-megabyte sized tables these inefficiencies can become significant. The more efficient procedure in this case is to read or write only as many rows of the table as will fit into the available internal I/O buffers, then access all the necessary columns of data within that range of rows. Then after the program is completely finished with the data in those rows it can move on to the next range of rows that will fit in the buffers, continuing in this

way until the entire file has been processed. By using this procedure of accessing all the columns of a table in parallel rather than sequentially, each block of the FITS file will only be read or written once.

The optimal number of rows to read or write at one time in a given table depends on the width of the table row, on the number of I/O buffers that have been allocated in FITSIO, and also on the number of other FITS files that are open at the same time (since one I/O buffer is always reserved for each open FITS file). Fortunately, a FITSIO routine is available that will return the optimal number of rows for a given table: call ftgrsz(unit, nrows, status). It is not critical to use exactly the value of nrows returned by this routine, as long as one does not exceed it. Using a very small value however can also lead to poor performance because of the overhead from the larger number of subroutine calls.

The optimal number of rows returned by ftgrsz is valid only as long as the application program is only reading or writing data in the specified table. Any other calls to access data in the table header would cause additional blocks of data to be loaded into the I/O buffers displacing data from the original table, and should be avoided during the critical period while the table is being read or written.

4. Use binary table extensions rather than ASCII table extensions for better efficiency when dealing with tabular data. The I/O to ASCII tables is slower because of the overhead in formatting or parsing the ASCII data fields, and because ASCII tables are about twice as large as binary tables with the same information content.

5. Design software so that it reads the FITS header keywords in the same order in which they occur in the file. When reading keywords, FITSIO searches forward starting from the position of the last keyword that was read. If it reaches the end of the header without finding the keyword, it then goes back to the start of the header and continues the search down to the position where it started. In practice, as long as the entire FITS header can fit at one time in the available internal I/O buffers, then the header keyword access will be very fast and it makes little difference which order they are accessed.

6. Avoid the use of scaling (by using the BSCALE and BZERO or TSCAL and TZERO keywords) in FITS files since the scaling operations add to the processing time needed to read or write the data. In some cases it may be more efficient to temporarily turn off the scaling (using ftpscl or fttscl) and then read or write the raw unscaled values in the FITS file.

7. Avoid using the 'implicit datatype conversion' capability in FITSIO. For instance, when reading a FITS image with BITPIX = -32 (32-bit floating point pixels), read the data into a single precision floating point data array in the program. Forcing FITSIO to convert the data to a different datatype can significantly slow the program.

8. Where feasible, design FITS binary tables using vector column elements so that the data are written as a contiguous set of bytes, rather than as single elements in multiple rows. For example, it is faster to access the data in a table that contains a single row and 2 columns with TFORM keywords equal to '10000E' and '10000J', than it is to access the same amount of data in a table with 10000 rows which has columns with the TFORM keywords equal to '1E' and '1J'. In the former case the 10000 floating point values in the first column are all written in a contiguous block of the file which can be read or written quickly, whereas in the second case each floating point value in the first column is interleaved with the integer value in the second column of the same row so

CFITSIO has to explicitly move to the position of each element to be read or written.

9. Avoid the use of variable length vector columns in binary tables, since any reading or writing of these data requires that CFITSIO first look up or compute the starting address of each row of data in the heap. In practice, this is probably not a significant efficiency issue.

10. When copying data from one FITS table to another, it is faster to transfer the raw bytes instead of reading then writing each column of the table. The FITSIO subroutines FTGTBS and FTPTBS (for ASCII tables), and FTGTBB and FTPTBB (for binary tables) will perform low-level reads or writes of any contiguous range of bytes in a table extension. These routines can be used to read or write a whole row (or multiple rows) of a table with a single subroutine call. These routines are fast because they bypass all the usual data scaling, error checking and machine dependent data conversion that is normally done by FITSIO, and they allow the program to write the data to the output file in exactly the same byte order. For these same reasons, use of these routines can be somewhat risky because no validation or machine dependent conversion is performed by these routines. In general these routines are only recommended for optimizing critical pieces of code and should only be used by programmers who thoroughly understand the internal byte structure of the FITS tables they are reading or writing.

11. Another strategy for improving the speed of writing a FITS table, similar to the previous one, is to directly construct the entire byte stream for a whole table row (or multiple rows) within the application program and then write it to the FITS file with ftptbb. This avoids all the overhead normally present in the column-oriented CFITSIO write routines. This technique should only be used for critical applications, because it makes the code more difficult to understand and maintain, and it makes the code more system dependent (e.g., do the bytes need to be swapped before writing to the FITS file?).

12. Finally, external factors such as the type of magnetic disk controller (SCSI or IDE), the size of the disk cache, the average seek speed of the disk, the amount of disk fragmentation, and the amount of RAM available on the system can all have a significant impact on overall I/O efficiency. For critical applications, a system administrator should review the proposed system hardware to identify any potential I/O bottlenecks.

Chapter 5

Basic Interface Routines

This section defines a basic set of subroutines that can be used to perform the most common types of read and write operations on FITS files. New users should start with these subroutines and then, as needed, explore the more advance routines described in the following chapter to perform more complex or specialized operations.

A right arrow symbol (>) is used to separate the input parameters from the output parameters in the definition of each routine. This symbol is not actually part of the calling sequence. Note that the status parameter is both an input and an output parameter and must be initialized = 0 prior to calling the FITSIO subroutines.

Refer to Chapter 9 for the definition of all the parameters used by these interface routines.

5.1 FITSIO Error Status Routines

1 Return the current version number of the fitsio library. The version number will be incremented with each new release of CFITSIO.

```
FTVERS( > version)
```

2 Return the descriptive text string corresponding to a FITSIO error status code. The 30-character length string contains a brief description of the cause of the error.

```
FTGERR(status, > errtext)
```

3 Return the top (oldest) 80-character error message from the internal FITSIO stack of error messages and shift any remaining messages on the stack up one level. Any FITSIO error will generate one or more messages on the stack. Call this routine repeatedly to get each message in sequence. The error stack is empty when a blank string is returned.

```
FTGMSG( > errmsg)
```

4 The FTPMRK routine puts an invisible marker on the CFITSIO error stack. The FTCMRK routine can then be used to delete any more recent error messages on the stack, back to the position of the marker. This preserves any older error messages on the stack. FTCMSG simply clears the entire error message stack. These routines are called without any arguments.

```
FTPMRK
FTCMRK
FTCMSG
```

5 Print out the error message corresponding to the input status value and all the error messages on the FITSIO stack to the specified file stream (stream can be either the string 'STDOUT' or 'STDERR'). If the input status value = 0 then this routine does nothing.

```
FTRPRT (stream, > status)
```

6 Write an 80-character message to the FITSIO error stack. Application programs should not normally write to the stack, but there may be some situations where this is desirable.

```
FTPMSG(errmsg)
```

5.2 File I/O Routines

1 Open an existing FITS file with readonly or readwrite access. This routine always opens the primary array (the first HDU) of the file, and does not move to a following extension, if one was specified as part of the filename. Use the FTNOPN routine to automatically move to the extension. This routine will also open IRAF images (.imh format files) and raw binary data arrays with READONLY access by first converting them on the fly into virtual FITS images. See the 'Extended File Name Syntax' chapter for more details. The FTDKOPEN routine simply opens the specified file without trying to interpret the filename using the extended filename syntax.

```
FTOPEN(unit,filename,rwmode, > blocksize,status)
FTDKOPEN(unit,filename,rwmode, > blocksize,status)
```

2 Open an existing FITS file with readonly or readwrite access and move to a following extension, if one was specified as part of the filename. (e.g., 'filename.fits+2' or 'filename.fits[2]' will move to the 3rd HDU in the file). Note that this routine differs from FTOPEN in that it does not have the redundant blocksize argument.

```
FTNOPN(unit,filename,rwmode, > status)
```

3 Open an existing FITS file with readonly or readwrite access and then move to the first HDU containing significant data, if a) an HDU name or number to open was not explicitly specified as part of the filename, and b) if the FITS file contains a null primary array (i.e., NAXIS = 0). In this case, it will look for the first IMAGE HDU with NAXIS ¿ 0, or the first table that does not contain the strings 'GTI' (Good Time Interval) or 'OBSTABLE' in the EXTNAME keyword value. FTTOPN is similar, except it will move to the first significant table HDU (skipping over any image HDUs) in the file if a specific HDU name or number is not specified. FTIOPN will move to the first non-null image HDU, skipping over any tables.

```
FTDOPN(unit,filename,rwmode, > status)
FTTOPN(unit,filename,rwmode, > status)
FTIOPN(unit,filename,rwmode, > status)
```

4 Open and initialize a new empty FITS file. A template file may also be specified to define the structure of the new file (see section 4.2.4). The FTDKINIT routine simply creates the specified file without trying to interpret the filename using the extended filename syntax.

```
FTINIT(unit,filename,blocksize, > status)
FTDKINIT(unit,filename,blocksize, > status)
```

5 Close a FITS file previously opened with ftopen or ftinit

```
FTCLOS(unit, > status)
```

6 Move to a specified (absolute) HDU in the FITS file (nhdu = 1 for the FITS primary array)

```
FTMAHD(unit,nhdu, > hdutype,status)
```

7 Create a primary array (if none already exists), or insert a new IMAGE extension immediately following the CHDU, or insert a new Primary Array at the beginning of the file. Any following extensions in the file will be shifted down to make room for the new extension. If the CHDU is the last HDU in the file then the new image extension will simply be appended to the end of the file. One can force a new primary array to be inserted at the beginning of the FITS file by setting status = -9 prior to calling the routine. In this case the existing primary array will be converted to an IMAGE extension. The new extension (or primary array) will become the CHDU. The FTIIMGLL routine is identical to the FTIIMG routine except that the 4th parameter (the length of each axis) is an array of 64-bit integers rather than an array of 32-bit integers.

```
FTIIMG(unit,bitpix,naxis,naxes, > status)
FTIIMGLL(unit,bitpix,naxis,naxesll, > status)
```

8 Insert a new ASCII TABLE extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new table extension will simply be appended to the end of the file. The new extension will become the CHDU. The FTITABLL routine is identical to the FTITAB routine except that the 2nd and 3rd parameters (that give the size of the table) are 64-bit integers rather than 32-bit integers. Under normal circumstances, the nrows and nrowsll paramenters should have a value of 0; CFITSIO will automatically update the number of rows as data is written to the table.

```
FTITAB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTITABLL(unit,rowlenll,nrowsll,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
```

9 Insert a new binary table extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new bintable extension will simply be appended to the end of the file. The new extension will become the CHDU. The FTIBINLL routine is identical to the FTIBIN routine except that the 2nd parameter (that gives the length of the table) is a 64-bit integer rather than a 32-bit integer. Under normal circumstances, the nrows and nrowsll paramenters should have a value of 0; CFITSIO will automatically update the number of rows as data is written to the table.

```
FTIBIN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTIBINLL(unit,nrowsll,tfields,ttype,tform,tunit,extname,varidat > status)
```

5.3 Keyword I/O Routines

1 Put (append) an 80-character record into the CHU.

```
FTPREC(unit,card, > status)
```

2 Put (append) a new keyword of the appropriate datatype into the CHU. The E and D versions of this routine have the added feature that if the 'decimals' parameter is negative, then the 'G' display format rather then the 'E' format will be used when constructing the keyword value, taking the absolute value of 'decimals' for the precision. This will suppress trailing zeros, and will use a fixed format rather than an exponential format, depending on the magnitude of the value.

```
FTPKY[JKLS](unit,keyword,keyval,comment, > status)
FTPKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

3 Get the nth 80-character header record from the CHU. The first keyword in the header is at key_no = 1; if key_no = 0 then this subroutine simple moves the internal pointer to the beginning of the header so that subsequent keyword operations will start at the top of the header; it also returns a blank card value in this case.

```
FTGREC(unit,key_no, > card,status)
```

4 Get a keyword value (with the appropriate datatype) and comment from the CHU

```
FTGKY[EDJKLS](unit,keyword, > keyval,comment,status)
```

5 Delete an existing keyword record.

```
FTDKEY(unit,keyword, > status)
```

5.4 Data I/O Routines

The following routines read or write data values in the current HDU of the FITS file. Automatic datatype conversion will be attempted for numerical datatypes if the specified datatype is different from the actual datatype of the FITS array or table column.

1 Write elements into the primary data array or image extension.

```
FTPPR[BIJKED](unit,group,fpixel,nelements,values, > status)
```

2 Read elements from the primary data array or image extension. Undefined array elements will be returned with a value = nullval, unless nullval = 0 in which case no checks for undefined pixels will be performed. The anyf parameter is set to true (= .true.) if any of the returned elements were undefined.

```
FTGPV[BIJKED](unit,group,fpixel,nelements,nullval, > values,anyf,status)
```

3 Write elements into an ASCII or binary table column. The 'felem' parameter applies only to vector columns in binary tables and is ignored when writing to ASCII tables.

```
FTPCL[SLBIJKEDCM](unit,colnum,frow,felem,nelements,values, > status)
```

4 Read elements from an ASCII or binary table column. Undefined array elements will be returned with a value = nullval, unless nullval = 0 (or = ' ' for ftgcvs) in which case no checking for undefined values will be performed. The ANYF parameter is set to true if any of the returned elements are undefined.

Any column, regardless of it's intrinsic datatype, may be read as a string. It should be noted however that reading a numeric column as a string is 10 - 100 times slower than reading the same column as a number due to the large overhead in constructing the formatted strings. The display format of the returned strings will be determined by the TDISPn keyword, if it exists, otherwise by the datatype of the column. The length of the returned strings can be determined with the ftgcdw routine. The following TDISPn display formats are currently supported:

```
Iw.m   Integer
Ow.m   Octal integer
Zw.m   Hexadecimal integer
Fw.d   Fixed floating point
Ew.d   Exponential floating point
Dw.d   Exponential floating point
Gw.d   General; uses Fw.d if significance not lost, else Ew.d
```

where w is the width in characters of the displayed values, m is the minimum number of digits displayed, and d is the number of digits to the right of the decimal. The .m field is optional.

```
FTGCV[SBIJKEDCM](unit,colnum,frow,felem,nelements,nullval, >
               values,anyf,status)
```

5 Get the table column number and full name of the column whose name matches the input template string. See the 'Advanced Interface Routines' chapter for a full description of this routine.

```
FTGCNN(unit,casesen,coltemplate, > colname,colnum,status)
```

Chapter 6

Advanced Interface Subroutines

This chapter defines all the available subroutines in the FITSIO user interface. For completeness, the basic subroutines described in the previous chapter are also repeated here. A right arrow symbol is used here to separate the input parameters from the output parameters in the definition of each subroutine. This symbol is not actually part of the calling sequence. An alphabetical list and definition of all the parameters is given at the end of this section.

6.1 FITS File Open and Close Subroutines:

1 Open an existing FITS file with readonly or readwrite access. The FTDKOPEN routine simply opens the specified file without trying to interpret the filename using the extended filename syntax. FTDOPN opens the file and also moves to the first HDU containing significant data, if no specific HDU is specified as part of the filename. FTTOPN and FTIOPN are similar except that they will move to the first table HDU or image HDU, respectively, if a HDU name or number is not specified as part of the filename.

```
FTOPEN(unit,filename,rwmode, > blocksize,status)
FTDKOPEN(unit,filename,rwmode, > blocksize,status)

FTDOPN(unit,filename,rwmode, > status)
FTTOPN(unit,filename,rwmode, > status)
FTIOPN(unit,filename,rwmode, > status)
```

2 Open an existing FITS file with readonly or readwrite access and move to a following extension, if one was specified as part of the filename. (e.g., 'filename.fits+2' or 'filename.fits[2]' will move to the 3rd HDU in the file). Note that this routine differs from FTOPEN in that it does not have the redundant blocksize argument.

```
FTNOPN(unit,filename,rwmode, > status)
```

3 Reopen a FITS file that was previously opened with FTOPEN, FTNOPN, or FTINIT. The newunit number may then be treated as a separate file, and one may simultaneously read or write to 2 (or more) different extensions in the same file. The FTOPEN and FTNOPN routines (above) automatically detects cases where a previously opened file is being opened again, and then internally call FTREOPEN, so programs should rarely need to explicitly call this routine.

```
FTREOPEN(unit, > newunit, status)
```

4 Open and initialize a new empty FITS file. The FTDKINIT routine simply creates the specified file without trying to interpret the filename using the extended filename syntax.

```
FTINIT(unit,filename,blocksize, > status)
FTDKINIT(unit,filename,blocksize, > status)
```

5 Create a new FITS file, using a template file to define its initial size and structure. The template may be another FITS HDU or an ASCII template file. If the input template file name is blank, then this routine behaves the same as FTINIT. The currently supported format of the ASCII template file is described under the fits_parse_template routine (in the general Utilities section), but this may change slightly later releases of CFITSIO.

```
FTTPLT(unit, filename, tplfilename, > status)
```

6 Flush internal buffers of data to the output FITS file previously opened with ftopen or ftinit. The routine usually never needs to be called, but doing so will ensure that if the program subsequently aborts, then the FITS file will have at least been closed properly.

```
FTFLUS(unit, > status)
```

7 Close a FITS file previously opened with ftopen or ftinit

```
FTCLOS(unit, > status)
```

8 Close and DELETE a FITS file previously opened with ftopen or ftinit. This routine may be useful in cases where a FITS file is created, but an error occurs which prevents the complete file from being written.

```
FTDELT(unit, > status)
```

9 Get the value of an unused I/O unit number which may then be used as input to FTOPEN or FTINIT. This routine searches for the first unused unit number in the range from with 99 down to 50. This routine just keeps an internal list of the allocated unit numbers and does not physically check that the Fortran unit is available (to be compatible with the SPP version of FITSIO). Thus users must not independently allocate any unit numbers in the range 50 - 99 if this routine is also to be used in the same program. This routine is provided for convenience only, and it is not required that the unit numbers used by FITSIO be allocated by this routine.

```
FTGIOU( > iounit, status)
```

10 Free (deallocate) an I/O unit number which was previously allocated with FTGIOU. All previously allocated unit numbers may be deallocated at once by calling FTFIOU with iounit = -1.

```
FTFIOU(iounit, > status)
```

11 Return the Fortran unit number that corresponds to the C fitsfile pointer value, or vice versa. These 2 C routines may be useful in mixed language programs where both C and Fortran subroutines need to access the same file. For example, if a FITS file is opened with unit 12 by a Fortran subroutine, then a C routine within the same program could get the fitfile pointer value to access the same file by calling 'fptr = CUnit2FITS(12)'. These routines return a value of zero if an error occurs.

```
int       CFITS2Unit(fitsfile *ptr);
fitsfile* CUnit2FITS(int unit);
```

11 Parse the input filename and return the HDU number that would be moved to if the file were opened with FTNOPN. The returned HDU number begins with 1 for the primary array, so for example, if the input filename = 'myfile.fits[2]' then hdunum = 3 will be returned. FITSIO does not open the file to check if the extension actually exists if an extension number is specified. If an extension *name* is included in the file name specification (e.g. 'myfile.fits[EVENTS]' then this routine will have to open the FITS file and look for the position of the named extension, then close file again. This is not possible if the file is being read from the stdin stream, and an error will be returned in this case. If the filename does not specify an explicit extension (e.g. 'myfile.fits') then hdunum = -99 will be returned, which is functionally equivalent to hdunum = 1. This routine is mainly used for backward compatibility in the ftools software package and is not recommended for general use. It is generally better and more efficient to first open the FITS file with FTNOPN, then use FTGHDN to determine which HDU in the file has been opened, rather than calling FTEXTN followed by a call to FTNOPN.

```
FTEXTN(filename, > nhdu, status)
```

12 Return the name of the opened FITS file.

```
FTFLNM(unit, > filename, status)
```

13 Return the I/O mode of the open FITS file (READONLY = 0, READWRITE = 1).

```
FTFLMD(unit, > iomode, status)
```

14 Return the file type of the opened FITS file (e.g. 'file://', 'ftp://', etc.).

```
FTURLT(unit, > urltype, status)
```

15 Parse the input filename or URL into its component parts: the file type (file://, ftp://, http://, etc), the base input file name, the name of the output file that the input file is to be copied to prior to opening, the HDU or extension specification, the filtering specifier, the binning specifier, and the column specifier. Blank strings will be returned for any components that are not present in the input file name.

```
FTIURL(filename, > filetype, infile, outfile, extspec, filter,
        binspec, colspec, status)
```

16 Parse the input file name and return the root file name. The root name includes the file type if specified, (e.g. 'ftp://' or 'http://') and the full path name, to the extent that it is specified in the input filename. It does not include the HDU name or number, or any filtering specifications.

```
FTRTNM(filename, > rootname, status)
```

16 Test if the input file or a compressed version of the file (with a .gz, .Z, .z, or .zip extension) exists on disk. The returned value of the 'exists' parameter will have 1 of the 4 following values:

```
 2:  the file does not exist, but a compressed version does exist
 1:  the disk file does exist
 0:  neither the file nor a compressed version of the file exist
-1:  the input file name is not a disk file (could be a ftp, http,
     smem, or mem file, or a file piped in on the STDIN stream)
```

```
FTEXIST(filename, > exists, status);
```

6.2 HDU-Level Operations

When a FITS file is first opened or created, the internal buffers in FITSIO automatically point to the first HDU in the file. The following routines may be used to move to another HDU in the file. Note that the HDU numbering convention used in FITSIO denotes the primary array as the first HDU, the first extension in a FITS file is the second HDU, and so on.

1 Move to a specified (absolute) HDU in the FITS file (nhdu = 1 for the FITS primary array)

```
FTMAHD(unit,nhdu, > hdutype,status)
```

2 Move to a new (existing) HDU forward or backwards relative to the CHDU

```
FTMRHD(unit,nmove, > hdutype,status)
```

3 Move to the (first) HDU which has the specified extension type and EXTNAME (or HDUNAME) and EXTVER keyword values. The hdutype parameter may have a value of IMAGE_HDU (0), ASCII_TBL (1), BINARY_TBL (2), or ANY_HDU (-1) where ANY_HDU means that only the extname and extver values will be used to locate the correct extension. If the input value of extver is 0 then the EXTVER keyword is ignored and the first HDU with a matching EXTNAME (or HDUNAME) keyword will be found. If no matching HDU is found in the file then the current HDU will remain unchanged and a status = BAD_HDU_NUM (301) will be returned.

```
FTMNHD(unit, hdutype, extname, extver, > status)
```

4 Get the number of the current HDU in the FITS file (primary array = 1)

```
FTGHDN(unit, > nhdu)
```

5 Return the type of the current HDU in the FITS file. The possible values for hdutype are IMAGE_HDU (0), ASCII_TBL (1), or BINARY_TBL (2).

```
FTGHDT(unit, > hdutype, status)
```

6 Return the total number of HDUs in the FITS file. The CHDU remains unchanged.

```
FTTHDU(unit, > hdunum, status)
```

7 Create (append) a new empty HDU following the last extension that has been previously accessed by the program. This will overwrite any extensions in an existing FITS file if the program has not already moved to that (or a later) extension using the FTMAHD or FTMRHD routines. For example, if an existing FITS file contains a primary array and 5 extensions and a program (1) opens the FITS file, (2) moves to extension 4, (3) moves back to the primary array, and (4) then calls FTCRHD, then the new extension will be written following the 4th extension, overwriting the existing 5th extension.

```
FTCRHD(unit, > status)
```

8 Create a primary array (if none already exists), or insert a new IMAGE extension immediately following the CHDU, or insert a new Primary Array at the beginning of the file. Any following extensions in the file will be shifted down to make room for the new extension. If the CHDU is the last HDU in the file then the new image extension will simply be appended to the end of the file. One can force a new primary array to be inserted at the beginning of the FITS file by setting status = -9 prior to calling the routine. In this case the existing primary array will be converted to an IMAGE extension. The new extension (or primary array) will become the CHDU. The FTIIMGLL routine is identical to the FTIIMG routine except that the 4th parameter (the length of each axis) is an array of 64-bit integers rather than an array of 32-bit integers.

```
FTIIMG(unit,bitpix,naxis,naxes, > status)
FTIIMGLL(unit,bitpix,naxis,naxesll, > status)
```

9 Insert a new ASCII TABLE extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new table extension will simply be appended to the end of the file. The new extension will become the CHDU. The FTITABLL routine is identical to the FTITAB routine except that the 2nd and 3rd parameters (that give the size of the table) are 64-bit integers rather than 32-bit integers.

```
FTITAB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTITABLL(unit,rowlenll,nrowsll,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
```

10 Insert a new binary table extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new bintable extension will simply be appended to the end of the file. The new extension will become the CHDU. The FTIBINLL routine is identical to the FTIBIN routine except that the 2nd parameter (that gives the length of the table) is a 64-bit integer rather than a 32-bit integer.

```
FTIBIN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTIBINLL(unit,nrowsll,tfields,ttype,tform,tunit,extname,varidat > status)
```

11 Resize an image by modifing the size, dimensions, and/or datatype of the current primary array or image extension. If the new image, as specified by the input arguments, is larger than the current existing image in the FITS file then zero fill data will be inserted at the end of the current image and any following extensions will be moved further back in the file. Similarly, if the new image is smaller than the current image then any following extensions will be shifted up towards the beginning of the FITS file and the image data will be truncated to the new size. This routine rewrites the BITPIX, NAXIS, and NAXISn keywords with the appropriate values for new image. The FTRSIMLL routine is identical to the FTRSIM routine except that the 4th parameter (the length of each axis) is an array of 64-bit integers rather than an array of 32-bit integers.

```
FTRSIM(unit,bitpix,naxis,naxes,status)
FTRSIMLL(unit,bitpix,naxis,naxesll,status)
```

12 Delete the CHDU in the FITS file. Any following HDUs will be shifted forward in the file, to fill in the gap created by the deleted HDU. In the case of deleting the primary array (the first HDU in the file) then the current primary array will be replace by a null primary array containing the minimum set of required keywords and no data. If there are more extensions in the file following the one that is deleted, then the the CHDU will be redefined to point to the following extension. If there are no following extensions then the CHDU will be redefined to point to the previous HDU. The output HDUTYPE parameter indicates the type of the new CHDU after the previous CHDU has been deleted.

```
FTDHDU(unit, > hdutype,status)
```

13 Copy all or part of the input FITS file and append it to the end of the output FITS file. If 'previous' (an integer parameter) is not equal to 0, then any HDUs preceding the current HDU in the input file will be copied to the output file. Similarly, 'current' and 'following' determine whether the current HDU, and/or any following HDUs in the input file will be copied to the output file. If all 3 parameters are not equal to zero, then the entire input file will be copied. On return, the current HDU in the input file will be unchanged, and the last copied HDU will be the current HDU in the output file.

```
FTCPFL(iunit, ounit, previous, current, following, > status)
```

14 Copy the entire CHDU from the FITS file associated with IUNIT to the CHDU of the FITS file associated with OUNIT. The output HDU must be empty and not already contain any keywords. Space will be reserved for MOREKEYS additional keywords in the output header if there is not already enough space.

```
FTCOPY(iunit,ounit,morekeys, > status)
```

15 Copy the header (and not the data) from the CHDU associated with inunit to the CHDU associated with outunit. If the current output HDU is not completely empty, then the CHDU will be closed and a new HDU will be appended to the output file. This routine will automatically transform the necessary keywords when copying a primary array to and image extension, or an image extension to a primary array. An empty output data unit will be created (all values = 0).

```
FTCPHD(inunit, outunit, > status)
```

16 Copy just the data from the CHDU associated with IUNIT to the CHDU associated with OUNIT. This will overwrite any data previously in the OUNIT CHDU. This low level routine is used by FTCOPY, but it may also be useful in certain application programs which want to copy the data from one FITS file to another but also want to modify the header keywords in the process. all the required header keywords must be written to the OUNIT CHDU before calling this routine

```
FTCPDT(iunit,ounit, > status)
```

6.3 Define or Redefine the structure of the CHDU

It should rarely be necessary to call the subroutines in this section. FITSIO internally calls these routines whenever necessary, so any calls to these routines by application programs will likely be redundant.

1 This routine forces FITSIO to scan the current header keywords that define the structure of the HDU (such as the NAXISn, PCOUNT and GCOUNT keywords) so that it can initialize the internal buffers that describe the HDU structure. This routine may be used instead of the more complicated calls to ftpdef, ftadef or ftbdef. This routine is also very useful for reinitializing the structure of an HDU, if the number of rows in a table, as specified by the NAXIS2 keyword, has been modified from its initial value.

```
FTRDEF(unit, > status)   (DEPRECATED)
```

2 Define the structure of the primary array or IMAGE extension. When writing GROUPed FITS files that by convention set the NAXIS1 keyword equal to 0, ftpdef must be called with naxes(1) = 1, NOT 0, otherwise FITSIO will report an error status=308 when trying to write data to a group. Note: it is usually simpler to call FTRDEF rather than this routine.

```
FTPDEF(unit,bitpix,naxis,naxes,pcount,gcount, > status)  (DEPRECATED)
```

3 Define the structure of an ASCII table (TABLE) extension. Note: it is usually simpler to call FTRDEF rather than this routine.

```
FTADEF(unit,rowlen,tfields,tbcol,tform,nrows > status) (DEPRECATED)
```

4 Define the structure of a binary table (BINTABLE) extension. Note: it is usually simpler to call FTRDEF rather than this routine.

```
FTBDEF(unit,tfields,tform,varidat,nrows > status) (DEPRECATED)
```

5 Define the size of the Current Data Unit, overriding the length of the data unit as previously defined by ftpdef, ftadef, or ftbdef. This is useful if one does not know the total size of the data unit until after the data have been written. The size (in bytes) of an ASCII or Binary table is given by NAXIS1 * NAXIS2. (Note that to determine the value of NAXIS1 it is often more convenient to read the value of the NAXIS1 keyword from the output file, rather than computing the row length directly from all the TFORM keyword values). Note: it is usually simpler to call FTRDEF rather than this routine.

```
FTDDEF(unit,bytlen, > status) (DEPRECATED)
```

6 Define the zero indexed byte offset of the 'heap' measured from the start of the binary table data. By default the heap is assumed to start immediately following the regular table data, i.e., at location NAXIS1 x NAXIS2. This routine is only relevant for binary tables which contain variable length array columns (with TFORMn = 'Pt'). This subroutine also automatically writes the value of theap to a keyword in the extension header. This subroutine must be called after the required keywords have been written (with ftphbn) and after the table structure has been defined (with ftbdef) but before any data is written to the table.

```
FTPTHP(unit,theap, > status)
```

6.4 FITS Header I/O Subroutines

6.4.1 Header Space and Position Routines

1 Reserve space in the CHU for MOREKEYS more header keywords. This subroutine may be called to reserve space for keywords which are to be written at a later time, after the data unit or subsequent extensions have been written to the FITS file. If this subroutine is not explicitly called, then the initial size of the FITS header will be limited to the space available at the time that the first data is written to the associated data unit. FITSIO has the ability to dynamically add more space to the header if needed, however it is more efficient to preallocate the required space if the size is known in advance.

```
FTHDEF(unit,morekeys, > status)
```

2 Return the number of existing keywords in the CHU (NOT including the END keyword which is not considered a real keyword) and the remaining space available to write additional keywords in the CHU. (returns KEYSADD = -1 if the header has not yet been closed). Note that FITSIO will attempt to dynamically add space for more keywords if required when appending new keywords to a header.

```
FTGHSP(iunit, > keysexist,keysadd,status)
```

3 Return the number of keywords in the header and the current position in the header. This returns the number of the keyword record that will be read next (or one greater than the position of the last keyword that was read or written). A value of 1 is returned if the pointer is positioned at the beginning of the header.

```
FTGHPS(iunit, > keysexist,key_no,status)
```

6.4.2 Read or Write Standard Header Routines

These subroutines provide a simple method of reading or writing most of the keyword values that are normally required in a FITS files. These subroutines are provided for convenience only and are not required to be used. If preferred, users may call the lower-level subroutines described in the previous section to individually read or write the required keywords. Note that in most cases, the required keywords such as NAXIS, TFIELD, TTYPEn, etc, which define the structure of the HDU must be written to the header before any data can be written to the image or table.

1 Put the primary header or IMAGE extension keywords into the CHU. There are 2 available routines: The simpler FTPHPS routine is equivalent to calling ftphpr with the default values of SIMPLE = true, pcount = 0, gcount = 1, and EXTEND = true. PCOUNT, GCOUNT and EXTEND keywords are not required in the primary header and are only written if pcount is not equal to zero, gcount is not equal to zero or one, and if extend is TRUE, respectively. When writing to an IMAGE extension, the SIMPLE and EXTEND parameters are ignored.

```
FTPHPS(unit,bitpix,naxis,naxes, > status)

FTPHPR(unit,simple,bitpix,naxis,naxes,pcount,gcount,extend, > status)
```

2 Get primary header or IMAGE extension keywords from the CHU. When reading from an IMAGE extension the SIMPLE and EXTEND parameters are ignored.

```
FTGHPR(unit,maxdim, > simple,bitpix,naxis,naxes,pcount,gcount,extend,
       status)
```

3 Put the ASCII table header keywords into the CHU. The optional TUNITn and EXTNAME keywords are written only if the input string values are not blank.

```
FTPHTB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
```

4 Get the ASCII table header keywords from the CHU

```
FTGHTB(unit,maxdim, > rowlen,nrows,tfields,ttype,tbcol,tform,tunit,
       extname,status)
```

5 Put the binary table header keywords into the CHU. The optional TUNITn and EXTNAME keywords are written only if the input string values are not blank. The pcount parameter, which specifies the size of the variable length array heap, should initially = 0; FITSIO will automatically update the PCOUNT keyword value if any variable length array data is written to the heap. The TFORM keyword value for variable length vector columns should have the form 'Pt(len)' or '1Pt(len)' where 't' is the data type code letter (A,I,J,E,D, etc.) and 'len' is an integer specifying the maximum length of the vectors in that column (len must be greater than or equal to the longest vector in the column). If 'len' is not specified when the table is created (e.g., the input TFORMn value is just '1Pt') then FITSIO will scan the column when the table is first closed and will append the maximum length to the TFORM keyword value. Note that if the table is subsequently modified to increase the maximum length of the vectors then the modifying program is responsible for also updating the TFORM keyword value.

```
FTPHBN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat, > status)
```

6 Get the binary table header keywords from the CHU

```
FTGHBN(unit,maxdim, > nrows,tfields,ttype,tform,tunit,extname,varidat,
       status)
```

6.4.3 Write Keyword Subroutines

1 Put (append) an 80-character record into the CHU.

```
FTPREC(unit,card, > status)
```

2 Put (append) a COMMENT keyword into the CHU. Multiple COMMENT keywords will be written if the input comment string is longer than 72 characters.

```
FTPCOM(unit,comment, > status)
```

3 Put (append) a HISTORY keyword into the CHU. Multiple HISTORY keywords will be written if the input history string is longer than 72 characters.

```
FTPHIS(unit,history, > status)
```

4 Put (append) the DATE keyword into the CHU. The keyword value will contain the current system date as a character string in 'dd/mm/yy' format. If a DATE keyword already exists in the header, then this subroutine will simply update the keyword value in-place with the current date.

```
FTPDAT(unit, > status)
```

5 Put (append) a new keyword of the appropriate datatype into the CHU. Note that FTPKYS will only write string values up to 68 characters in length; longer strings will be truncated. The FTPKLS routine can be used to write longer strings, using a non-standard FITS convention. The E and D versions of this routine have the added feature that if the 'decimals' parameter is negative, then the 'G' display format rather then the 'E' format will be used when constructing the keyword value, taking the absolute value of 'decimals' for the precision. This will suppress trailing zeros, and will use a fixed format rather than an exponential format, depending on the magnitude of the value.

```
FTPKY[JKLS](unit,keyword,keyval,comment, > status)
FTPKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

6 Put (append) a string valued keyword into the CHU which may be longer than 68 characters in length. This uses the Long String Keyword convention that is described in the "Usage Guidelines and Suggestions" section of this document. Since this uses a non-standard FITS convention to encode the long keyword string, programs which use this routine should also call the FTPLSW routine to add some COMMENT keywords to warn users of the FITS file that this convention is being used. FTPLSW also writes a keyword called LONGSTRN to record the version of the longstring convention that has been used, in case a new convention is adopted at some point in the future. If the LONGSTRN keyword is already present in the header, then FTPLSW will simply return and will not write duplicate keywords.

```
FTPKLS(unit,keyword,keyval,comment, > status)
FTPLSW(unit, > status)
```

7 Put (append) a new keyword with an undefined, or null, value into the CHU. The value string of the keyword is left blank in this case.

```
FTPKYU(unit,keyword,comment, > status)
```

8 Put (append) a numbered sequence of keywords into the CHU. One may append the same comment to every keyword (and eliminate the need to have an array of identical comment strings, one for each keyword) by including the ampersand character as the last non-blank character in the (first) COMMENTS string parameter. This same string will then be used for the comment field in all the keywords. (Note that the SPP version of these routines only supports a single comment string).

```
FTPKN[JKLS](unit,keyroot,startno,no_keys,keyvals,comments, > status)
FTPKN[EDFG](unit,keyroot,startno,no_keys,keyvals,decimals,comments, >
            status)
```

9 Copy an indexed keyword from one HDU to another, modifying the index number of the keyword name in the process. For example, this routine could read the TLMIN3 keyword from the input HDU (by giving keyroot = "TLMIN" and innum = 3) and write it to the output HDU with the keyword name TLMIN4 (by setting outnum = 4). If the input keyword does not exist, then this routine simply returns without indicating an error.

```
FTCPKY(inunit, outunit, innum, outnum, keyroot, > status)
```

10 Put (append) a 'triple precision' keyword into the CHU in F28.16 format. The floating point keyword value is constructed by concatenating the input integer value with the input double precision fraction value (which must have a value between 0.0 and 1.0). The FTGKYT routine should be used to read this keyword value, because the other keyword reading subroutines will not preserve the full precision of the value.

```
FTPKYT(unit,keyword,intval,dblval,comment, > status)
```

11 Write keywords to the CHDU that are defined in an ASCII template file. The format of the template file is described under the ftgthd routine below.

```
FTPKTP(unit, filename, > status)
```

12 Append the physical units string to an existing keyword. This routine uses a local convention, shown in the following example, in which the keyword units are enclosed in square brackets in the beginning of the keyword comment field.

```
VELOCITY=                 12.3 / [km/s] orbital speed
```

```
FTPUNT(unit,keyword,units, > status)
```

6.4.4 Insert Keyword Subroutines

1 Insert a new keyword record into the CHU at the specified position (i.e., immediately preceding the (keyno)th keyword in the header.) This 'insert record' subroutine is somewhat less efficient then the 'append record' subroutine (FTPREC) described above because the remaining keywords in the header have to be shifted down one slot.

```
FTIREC(unit,key_no,card, > status)
```

2 Insert a new keyword into the CHU. The new keyword is inserted immediately following the last keyword that has been read from the header. The FTIKLS subroutine works the same as the FTIKYS subroutine, except it also supports long string values greater than 68 characters in length. These 'insert keyword' subroutines are somewhat less efficient then the 'append keyword' subroutines described above because the remaining keywords in the header have to be shifted down one slot.

```
FTIKEY(unit, card, > status)
FTIKY[JKLS](unit,keyword,keyval,comment, > status)
FTIKLS(unit,keyword,keyval,comment, > status)
FTIKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

3 Insert a new keyword with an undefined, or null, value into the CHU. The value string of the keyword is left blank in this case.

```
FTIKYU(unit,keyword,comment, > status)
```

6.4.5 Read Keyword Subroutines

These routines return the value of the specified keyword(s). Wild card characters (*, ?, or #) may be used when specifying the name of the keyword to be read: a '?' will match any single character at that position in the keyword name and a '*' will match any length (including zero) string of characters. The '#' character will match any consecutive string of decimal digits (0 - 9). Note that when a wild card is used in the input keyword name, the routine will only search for a match from the current header position to the end of the header. It will not resume the search from the top of the header back to the original header position as is done when no wildcards are included in the keyword name. If the desired keyword string is 8-characters long (the maximum length of a keyword name) then a '*' may be appended as the ninth character of the input name to force the keyword search to stop at the end of the header (e.g., 'COMMENT *' will search for the next COMMENT keyword). The ffgrec routine may be used to set the starting position when doing wild card searches.

1 Get the nth 80-character header record from the CHU. The first keyword in the header is at key_no = 1; if key_no = 0 then this subroutine simple moves the internal pointer to the beginning of the header so that subsequent keyword operations will start at the top of the header; it also returns a blank card value in this case.

```
FTGREC(unit,key_no, > card,status)
```

2 Get the name, value (as a string), and comment of the nth keyword in CHU. This routine also checks that the returned keyword name (KEYWORD) contains only legal ASCII characters. Call FTGREC and FTPSVC to bypass this error check.

```
FTGKYN(unit,key_no, > keyword,value,comment,status)
```

3 Get the 80-character header record for the named keyword

```
FTGCRD(unit,keyword, > card,status)
```

4 Get the next keyword whose name matches one of the strings in 'inclist' but does not match any of the strings in 'exclist'. The strings in inclist and exclist may contain wild card characters (*, ?, and #) as described at the beginning of this section. This routine searches from the current header position to the end of the header, only, and does not continue the search from the top of the header back to the original position. The current header position may be reset with the ftgrec routine. Note that nexc may be set = 0 if there are no keywords to be excluded. This routine returns status = 202 if a matching keyword is not found.

```
FTGNXK(unit,inclist,ninc,exclist,nexc, > card,status)
```

5 Get the literal keyword value as a character string. Regardless of the datatype of the keyword, this routine simply returns the string of characters in the value field of the keyword along with the comment field.

```
FTGKEY(unit,keyword, > value,comment,status)
```

6 Get a keyword value (with the appropriate datatype) and comment from the CHU

```
FTGKY[EDJKLS](unit,keyword, > keyval,comment,status)
```

7 Get a sequence of numbered keyword values. These routines do not support wild card characters in the root name.

```
FTGKN[EDJKLS](unit,keyroot,startno,max_keys, > keyvals,nfound,status)
```

8 Get the value of a floating point keyword, returning the integer and fractional parts of the value in separate subroutine arguments. This subroutine may be used to read any keyword but is especially useful for reading the 'triple precision' keywords written by FTPKYT.

```
FTGKYT(unit,keyword, > intval,dblval,comment,status)
```

9 Get the physical units string in an existing keyword. This routine uses a local convention, shown in the following example, in which the keyword units are enclosed in square brackets in the beginning of the keyword comment field. A blank string is returned if no units are defined for the keyword.

```
VELOCITY=                 12.3 / [km/s] orbital speed
```

```
    FTGUNT(unit,keyword, > units,status)
```

6.4.6 Modify Keyword Subroutines

Wild card characters, as described in the Read Keyword section, above, may be used when specifying the name of the keyword to be modified.

1 Modify (overwrite) the nth 80-character header record in the CHU

```
    FTMREC(unit,key_no,card, > status)
```

2 Modify (overwrite) the 80-character header record for the named keyword in the CHU. This can be used to overwrite the name of the keyword as well as its value and comment fields.

```
    FTMCRD(unit,keyword,card, > status)
```

3 Modify (overwrite) the name of an existing keyword in the CHU preserving the current value and comment fields.

```
    FTMNAM(unit,oldkey,keyword, > status)
```

4 Modify (overwrite) the comment field of an existing keyword in the CHU

```
    FTMCOM(unit,keyword,comment, > status)
```

5 Modify the value and comment fields of an existing keyword in the CHU. The FTMKLS subroutine works the same as the FTMKYS subroutine, except it also supports long string values greater than 68 characters in length. Optionally, one may modify only the value field and leave the comment field unchanged by setting the input COMMENT parameter equal to the ampersand character (&). The E and D versions of this routine have the added feature that if the 'decimals' parameter is negative, then the 'G' display format rather then the 'E' format will be used when constructing the keyword value, taking the absolute value of 'decimals' for the precision. This will suppress trailing zeros, and will use a fixed format rather than an exponential format, depending on the magnitude of the value.

```
    FTMKY[JKLS](unit,keyword,keyval,comment, > status)
    FTMKLS(unit,keyword,keyval,comment, > status)
    FTMKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

6 Modify the value of an existing keyword to be undefined, or null. The value string of the keyword is set to blank. Optionally, one may leave the comment field unchanged by setting the input COMMENT parameter equal to the ampersand character (&).

```
FTMKYU(unit,keyword,comment, > status)
```

6.4.7 Update Keyword Subroutines

1 Update an 80-character record in the CHU. If the specified keyword already exists then that header record will be replaced with the input CARD string. If it does not exist then the new record will be added to the header. The FTUKLS subroutine works the same as the FTUKYS subroutine, except it also supports long string values greater than 68 characters in length.

```
FTUCRD(unit,keyword,card, > status)
```

2 Update the value and comment fields of a keyword in the CHU. The specified keyword is modified if it already exists (by calling FTMKYx) otherwise a new keyword is created by calling FTPKYx. The E and D versions of this routine have the added feature that if the 'decimals' parameter is negative, then the 'G' display format rather then the 'E' format will be used when constructing the keyword value, taking the absolute value of 'decimals' for the precision. This will suppress trailing zeros, and will use a fixed format rather than an exponential format, depending on the magnitude of the value.

```
FTUKY[JKLS](unit,keyword,keyval,comment, > status)
FTUKLS(unit,keyword,keyval,comment, > status)
FTUKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

3 Update the value of an existing keyword to be undefined, or null, or insert a new undefined-value keyword if it doesn't already exist. The value string of the keyword is left blank in this case.

```
FTUKYU(unit,keyword,comment, > status)
```

6.4.8 Delete Keyword Subroutines

1 Delete an existing keyword record. The space previously occupied by the keyword is reclaimed by moving all the following header records up one row in the header. The first routine deletes a keyword at a specified position in the header (the first keyword is at position 1), whereas the second routine deletes a specifically named keyword. Wild card characters, as described in the Read Keyword section, above, may be used when specifying the name of the keyword to be deleted (be careful!).

```
FTDREC(unit,key_no, > status)
FTDKEY(unit,keyword, > status)
```

6.5 Data Scaling and Undefined Pixel Parameters

These subroutines define or modify the internal parameters used by FITSIO to either scale the data or to represent undefined pixels. Generally FITSIO will scale the data according to the values of the BSCALE and BZERO (or TSCALn and TZEROn) keywords, however these subroutines may be used to override the keyword values. This may be useful when one wants to read or write the raw unscaled values in the FITS file. Similarly, FITSIO generally uses the value of the BLANK or TNULLn keyword to signify an undefined pixel, but these routines may be used to override this value. These subroutines do not create or modify the corresponding header keyword values.

1 Reset the scaling factors in the primary array or image extension; does not change the BSCALE and BZERO keyword values and only affects the automatic scaling performed when the data elements are written/read to/from the FITS file. When reading from a FITS file the returned data value = (the value given in the FITS array) * BSCALE + BZERO. The inverse formula is used when writing data values to the FITS file. (NOTE: BSCALE and BZERO must be declared as Double Precision variables).

```
FTPSCL(unit,bscale,bzero, > status)
```

2 Reset the scaling parameters for a table column; does not change the TSCALn or TZEROn keyword values and only affects the automatic scaling performed when the data elements are written/read to/from the FITS file. When reading from a FITS file the returned data value = (the value given in the FITS array) * TSCAL + TZERO. The inverse formula is used when writing data values to the FITS file. (NOTE: TSCAL and TZERO must be declared as Double Precision variables).

```
FTTSCL(unit,colnum,tscal,tzero, > status)
```

3 Define the integer value to be used to signify undefined pixels in the primary array or image extension. This is only used if BITPIX = 8, 16, 32. or 64 This does not create or change the value of the BLANK keyword in the header. FTPNULLL is identical to FTPNUL except that the blank value is a 64-bit integer instead of a 32-bit integer.

```
FTPNUL(unit,blank, > status)
FTPNULLL(unit,blankll, > status)
```

4 Define the string to be used to signify undefined pixels in a column in an ASCII table. This does not create or change the value of the TNULLn keyword.

```
FTSNUL(unit,colnum,snull > status)
```

5 Define the value to be used to signify undefined pixels in an integer column in a binary table (where TFORMn = 'B', 'I', 'J', or 'K'). This does not create or change the value of the TNULLn keyword. FTTNULLL is identical to FTTNUL except that the tnull value is a 64-bit integer instead of a 32-bit integer.

```
FTTNUL(unit,colnum,tnull > status)
FTTNULLL(unit,colnum,tnullll > status)
```

6.6 FITS Primary Array or IMAGE Extension I/O Subroutines

These subroutines put or get data values in the primary data array (i.e., the first HDU in the FITS file) or an IMAGE extension. The data array is represented as a single one-dimensional array of pixels regardless of the actual dimensionality of the array, and the FPIXEL parameter gives the position within this 1-D array of the first pixel to read or write. Automatic data type conversion is performed for numeric data (except for complex data types) if the data type of the primary array (defined by the BITPIX keyword) differs from the data type of the array in the calling subroutine. The data values are also scaled by the BSCALE and BZERO header values as they are being written or read from the FITS array. The ftpscl subroutine MUST be called to define the scaling parameters when writing data to the FITS array or to override the default scaling value given in the header when reading the FITS array.

Two sets of subroutines are provided to read the data array which differ in the way undefined pixels are handled. The first set of routines (FTGPVx) simply return an array of data elements in which undefined pixels are set equal to a value specified by the user in the 'nullval' parameter. An additional feature of these subroutines is that if the user sets nullval = 0, then no checks for undefined pixels will be performed, thus increasing the speed of the program. The second set of routines (FTGPFx) returns the data element array and, in addition, a logical array which defines whether the corresponding data pixel is undefined. The latter set of subroutines may be more convenient to use in some circumstances, however, it requires an additional array of logical values which can be unwieldy when working with large data arrays. Also for programmer convenience, sets of subroutines to directly read or write 2 and 3 dimensional arrays have been provided, as well as a set of subroutines to read or write any contiguous rectangular subset of pixels within the n-dimensional array.

1 Get the data type of the image (= BITPIX value). Possible returned values are: 8, 16, 32, 64, -32, or -64 corresponding to unsigned byte, signed 2-byte integer, signed 4-byte integer, signed 8-byte integer, real, and double.

The second subroutine is similar to FTGIDT, except that if the image pixel values are scaled, with non-default values for the BZERO and BSCALE keywords, then this routine will return the 'equivalent' data type that is needed to store the scaled values. For example, if BITPIX = 16 and BSCALE = 0.1 then the equivalent data type is floating point, and -32 will be returned. There are 2 special cases: if the image contains unsigned 2-byte integer values, with BITPIX = 16, BSCALE = 1, and BZERO = 32768, then this routine will return a non-standard value of 20 for the bitpix value. Similarly if the image contains unsigned 4-byte integers, then bitpix will be returned with a value of 40.

```
FTGIDT(unit, > bitpix,status)
FTGIET(unit, > bitpix,status)
```

2 Get the dimension (number of axes = NAXIS) of the image

```
FTGIDM(unit, > naxis,status)
```

3 Get the size of all the dimensions of the image. The FTGISZLL routine returns an array of 64-bit integers instead of 32-bit integers.

```
FTGISZ(unit, maxdim, > naxes,status)
FTGISZLL(unit, maxdim, > naxesll,status)
```

4 Get the parameters that define the type and size of the image. This routine simply combines calls to the above 3 routines. The FTGIPRLL routine returns an array of 64-bit integers instead of 32-bit integers.

```
FTGIPR(unit, maxdim, > bitpix, naxis, naxes, int *status)
FTGIPRLL(unit, maxdim, > bitpix, naxis, naxesll, int *status)
```

5 Put elements into the data array

```
FTPPR[BIJKED](unit,group,fpixel,nelements,values, > status)
```

6 Put elements into the data array, substituting the appropriate FITS null value for all elements which are equal to the value of NULLVAL. For integer FITS arrays, the null value defined by the previous call to FTPNUL will be substituted; for floating point FITS arrays (BITPIX = -32 or -64) then the special IEEE NaN (Not-a-Number) value will be substituted.

```
FTPPN[BIJKED](unit,group,fpixel,nelements,values,nullval > status)
```

7 Set data array elements as undefined

```
FTPPRU(unit,group,fpixel,nelements, > status)
```

8 Get elements from the data array. Undefined array elements will be returned with a value = nullval, unless nullval = 0 in which case no checks for undefined pixels will be performed.

```
FTGPV[BIJKED](unit,group,fpixel,nelements,nullval, > values,anyf,status)
```

9 Get elements and nullflags from data array. Any undefined array elements will have the corresponding flagvals element set equal to .TRUE.

```
FTGPF[BIJKED](unit,group,fpixel,nelements, > values,flagvals,anyf,status)
```

10 Put values into group parameters

```
FTPGP[BIJKED](unit,group,fparm,nparm,values, > status)
```

11 Get values from group parameters

```
FTGGP[BIJKED](unit,group,fparm,nparm, > values,status)
```

The following 4 subroutines transfer FITS images with 2 or 3 dimensions to or from a data array which has been declared in the calling program. The dimensionality of the FITS image is passed by the naxis1, naxis2, and naxis3 parameters and the declared dimensions of the program array are passed in the dim1 and dim2 parameters. Note that the program array does not have to have the same dimensions as the FITS array, but must be at least as big. For example if a FITS image with NAXIS1 = NAXIS2 = 400 is read into a program array which is dimensioned as 512 x 512 pixels, then the image will just fill the lower left corner of the array with pixels in the range 1 - 400 in the X an Y directions. This has the effect of taking a contiguous set of pixel value in the FITS array and writing them to a non-contiguous array in program memory (i.e., there are now some blank pixels around the edge of the image in the program array).

11 Put 2-D image into the data array

```
FTP2D[BIJKED](unit,group,dim1,naxis1,naxis2,image, > status)
```

12 Put 3-D cube into the data array

```
FTP3D[BIJKED](unit,group,dim1,dim2,naxis1,naxis2,naxis3,cube, > status)
```

13 Get 2-D image from the data array. Undefined pixels in the array will be set equal to the value of 'nullval', unless nullval=0 in which case no testing for undefined pixels will be performed.

```
FTG2D[BIJKED](unit,group,nullval,dim1,naxis1,naxis2, > image,anyf,status)
```

14 Get 3-D cube from the data array. Undefined pixels in the array will be set equal to the value of 'nullval', unless nullval=0 in which case no testing for undefined pixels will be performed.

```
FTG3D[BIJKED](unit,group,nullval,dim1,dim2,naxis1,naxis2,naxis3, >
              cube,anyf,status)
```

The following subroutines transfer a rectangular subset of the pixels in a FITS N-dimensional image to or from an array which has been declared in the calling program. The fpixels and lpixels parameters are integer arrays which specify the starting and ending pixels in each dimension of the FITS image that are to be read or written. (Note that these are the starting and ending pixels in the FITS image, not in the declared array). The array parameter is treated simply as a large one-dimensional array of the appropriate datatype containing the pixel values; The pixel values in the FITS array are read/written from/to this program array in strict sequence without any gaps; it is up to the calling routine to correctly interpret the dimensionality of this array. The two families of FITS reading routines (FTGSVx and FTGSFx subroutines) also have an 'incs' parameter which defines the data sampling interval in each dimension of the FITS array. For example, if incs(1)=2 and incs(2)=3 when reading a 2-dimensional FITS image, then only every other pixel in the first dimension and every 3rd pixel in the second dimension will be returned in the 'array' parameter. [Note: the FTGSSx family of routines which were present in previous versions of FITSIO have been superseded by the more general FTGSVx family of routines.]

15 Put an arbitrary data subsection into the data array.

```
FTPSS[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,array, > status)
```

16 Get an arbitrary data subsection from the data array. Undefined pixels in the array will be set equal to the value of 'nullval', unless nullval=0 in which case no testing for undefined pixels will be performed.

```
FTGSV[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,incs,nullval, >
              array,anyf,status)
```

17 Get an arbitrary data subsection from the data array. Any Undefined pixels in the array will have the corresponding 'flagvals' element set equal to .TRUE.

```
FTGSF[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,incs, >
              array,flagvals,anyf,status)
```

6.7 FITS ASCII and Binary Table Data I/O Subroutines

6.7.1 Column Information Subroutines

1 Get the number of rows or columns in the current FITS table. The number of rows is given by the NAXIS2 keyword and the number of columns is given by the TFIELDS keyword in the header of the table. The FTGNRWLL routine is identical to FTGNRW except that the number of rows is returned as a 64-bit integer rather than a 32-bit integer.

```
FTGNRW(unit, > nrows, status)
FTGNRWLL(unit, > nrowsll, status)
FTGNCL(unit, > ncols, status)
```

2 Get the table column number (and name) of the column whose name matches an input template name. The table column names are defined by the TTYPEn keywords in the FITS header. If a column does not have a TTYPEn keyword, then these routines assume that the name consists of all blank characters. These 2 subroutines perform the same function except that FTGCNO only returns the number of the matching column whereas FTGCNN also returns the name of the column. If CASESEN = .true. then the column name match will be case-sensitive.

The input column name template (COLTEMPLATE) is (1) either the exact name of the column to be searched for, or (2) it may contain wild cards characters (*, ?, or #), or (3) it may contain the number of the desired column (where the number is expressed as ASCII digits). The first 2 wild cards behave similarly to UNIX filename matching: the '*' character matches any sequence of characters (including zero characters) and the '?' character matches any single character. The # wildcard will match any consecutive string of decimal digits (0-9). As an example, the template strings 'AB?DE', 'AB*E', and 'AB*CDE' will all match the string 'ABCDE'. If more than one column name in the table matches the template string, then the first match is returned and the status value will be set to 237 as a warning that a unique match was not found. To find the other cases that match the template, simply call the subroutine again leaving the input status value equal to 237 and the next matching name will then be returned. Repeat this process until a status = 219 (column name not found) is returned. If these subroutines fail to match the template to any of the columns in the table, they lastly check if the template can be interpreted as a simple positive integer (e.g., '7', or '512') and if so, they return that column number. If no matches are found then a status = 219 error is returned.

Note that the FITS Standard recommends that only letters, digits, and the underscore character be used in column names (with no embedded spaces in the name). Trailing blank characters are not significant.

```
FTGCNO(unit,casesen,coltemplate, > colnum,status)
FTGCNN(unit,casesen,coltemplate, > colname,colnum,status)
```

3 Get the datatype of a column in an ASCII or binary table. This routine returns an integer code value corresponding to the datatype of the column. (See the FTBNFM and FTASFM subroutines in the Utilities section of this document for a list of the code values). The vector repeat count (which is alway 1 for ASCII table columns) is also returned. If the specified column has an ASCII character datatype (code = 16) then the width of a unit string in the column is also returned. Note that this routine supports the local convention for specifying arrays of strings within a binary table character column, using the syntax TFORM = 'rAw' where 'r' is the total number of characters (= the width of the column) and 'w' is the width of a unit string within the column. Thus if the column has TFORM = '60A12' then this routine will return datacode = 16, repeat = 60, and width = 12. (The TDIMn keyword may also be used to specify the unit string length; The pair of keywords TFORMn = '60A' and TDIMn = '(12,5)' would have the same effect as TFORMn = '60A12').

The second routine, FTEQTY is similar except that in the case of scaled integer columns it returns the 'equivalent' data type that is needed to store the scaled values, and not necessarily the physical data type of the unscaled values as stored in the FITS table. For example if a '1I' column in a binary table has TSCALn = 1 and TZEROn = 32768, then this column effectively contains unsigned short integer values, and thus the returned value of typecode will be the code for an unsigned short integer, not a signed short integer. Similarly, if a column has TTYPEn = '1I' and TSCALn = 0.12, then the returned typecode will be the code for a 'real' column.

```
FTGTCL(unit,colnum, > datacode,repeat,width,status)
FTEQTY(unit,colnum, > datacode,repeat,width,status)
```

4 Return the display width of a column. This is the length of the string that will be returned when reading the column as a formatted string. The display width is determined by the TDISPn keyword, if present, otherwise by the data type of the column.

```
FTGCDW(unit, colnum, > dispwidth, status)
```

5 Get information about an existing ASCII table column. (NOTE: TSCAL and TZERO must be declared as Double Precision variables). All the returned parameters are scalar quantities.

```
FTGACL(unit,colnum, >
       ttype,tbcol,tunit,tform,tscal,tzero,snull,tdisp,status)
```

6 Get information about an existing binary table column. (NOTE: TSCAL and TZERO must be declared as Double Precision variables). DATATYPE is a character string which returns the datatype of the column as defined by the TFORMn keyword (e.g., 'I', 'J','E', 'D', etc.). In the case of an ASCII character column, DATATYPE will have a value of the form 'An' where 'n' is an integer expressing the width of the field in characters. For example, if TFORM = '160A8' then FTGBCL will return DATATYPE='A8' and REPEAT=20. All the returned parameters are scalar quantities.

```
FTGBCL(unit,colnum, >
       ttype,tunit,datatype,repeat,tscal,tzero,tnull,tdisp,status)
```

7 Put (append) a TDIMn keyword whose value has the form '(l,m,n...)' where l, m, n... are the dimensions of a multidimensional array column in a binary table.

```
FTPTDM(unit,colnum,naxis,naxes, > status)
```

8 Return the number of and size of the dimensions of a table column. Normally this information is given by the TDIMn keyword, but if this keyword is not present then this routine returns NAXIS = 1 and NAXES(1) equal to the repeat count in the TFORM keyword.

```
FTGTDM(unit,colnum,maxdim, > naxis,naxes,status)
```

9 Decode the input TDIMn keyword string (e.g. '(100,200)') and return the number of and size of the dimensions of a binary table column. If the input tdimstr character string is null, then this routine returns naxis = 1 and naxes[0] equal to the repeat count in the TFORM keyword. This routine is called by FTGTDM.

```
FTDTDM(unit,tdimstr,colnum,maxdim, > naxis,naxes, status)
```

10 Return the optimal number of rows to read or write at one time for maximum I/O efficiency. Refer to the "Optimizing Code" section in Chapter 5 for more discussion on how to use this routine.

```
FFGRSZ(unit, > nrows,status)
```

6.7.2 Low-Level Table Access Subroutines

The following subroutines provide low-level access to the data in ASCII or binary tables and are mainly useful as an efficient way to copy all or part of a table from one location to another. These routines simply read or write the specified number of consecutive bytes in an ASCII or binary table, without regard for column boundaries or the row length in the table. The first two subroutines read or write consecutive bytes in a table to or from a character string variable, while the last two subroutines read or write consecutive bytes to or from a variable declared as a numeric data type (e.g., INTEGER, INTEGER*2, REAL, DOUBLE PRECISION). These routines do not perform any machine dependent data conversion or byte swapping, except that conversion to/from ASCII format is performed by the FTGTBS and FTPTBS routines on machines which do not use ASCII character codes in the internal data representations (e.g., on IBM mainframe computers).

1 Read a consecutive string of characters from an ASCII table into a character variable (spanning columns and multiple rows if necessary) This routine should not be used with binary tables because of complications related to passing string variables between C and Fortran.

```
FTGTBS(unit,frow,startchar,nchars, > string,status)
```

2 Write a consecutive string of characters to an ASCII table from a character variable (spanning columns and multiple rows if necessary) This routine should not be used with binary tables because of complications related to passing string variables between C and Fortran.

```
FTPTBS(unit,frow,startchar,nchars,string, > status)
```

3 Read a consecutive array of bytes from an ASCII or binary table into a numeric variable (spanning columns and multiple rows if necessary). The array parameter may be declared as any numerical datatype as long as the array is at least 'nchars' bytes long, e.g., if nchars = 17, then declare the array as INTEGER*4 ARRAY(5).

```
FTGTBB(unit,frow,startchar,nchars, > array,status)
```

4 Write a consecutive array of bytes to an ASCII or binary table from a numeric variable (spanning columns and multiple rows if necessary) The array parameter may be declared as any numerical datatype as long as the array is at least 'nchars' bytes long, e.g., if nchars = 17, then declare the array as INTEGER*4 ARRAY(5).

```
FTPTBB(unit,frow,startchar,nchars,array, > status)
```

6.7.3 Edit Rows or Columns

1 Insert blank rows into an existing ASCII or binary table (in the CDU). All the rows FOLLOWING row FROW are shifted down by NROWS rows. If FROW or FROWLL equals 0 then the blank rows are inserted at the beginning of the table. These routines modify the NAXIS2 keyword to reflect the new number of rows in the table. Note that it is *not* necessary to insert rows in a table before writing data to those rows (indeed, it would be inefficient to do so). Instead, one may simply write data to any row of the table, whether that row of data already exists or not.

```
FTIROW(unit,frow,nrows, > status)
FTIROWLL(unit,frowll,nrowsll, > status)
```

2 Delete rows from an existing ASCII or binary table (in the CDU). The NROWS (or NROWSLL) is the number of rows are deleted, starting with row FROW (or FROWLL), and any remaining rows in the table are shifted up to fill in the space. These routines modify the NAXIS2 keyword to reflect the new number of rows in the table.

```
FTDROW(unit,frow,nrows, > status)
FTDROWLL(unit,frowll,nrowsll, > status)
```

3 Delete a list of rows from an ASCII or binary table (in the CDU). In the first routine, 'rowrange' is a character string listing the rows or row ranges to delete (e.g., '2-4, 5, 8-9'). In the second routine, 'rowlist' is an integer array of row numbers to be deleted from the table. nrows is the number of row numbers in the list. The first row in the table is 1 not 0. The list of row numbers must be sorted in ascending order.

```
FTDRRG(unit,rowrange, > status)
FTDRWS(unit,rowlist,nrows, > status)
```

4 Insert a blank column (or columns) into an existing ASCII or binary table (in the CDU). COLNUM specifies the column number that the (first) new column should occupy in the table. NCOLS specifies how many columns are to be inserted. Any existing columns from this position and higher are moved over to allow room for the new column(s). The index number on all the following keywords will be incremented if necessary to reflect the new position of the column(s) in the table: TBCOLn, TFORMn, TTYPEn, TUNITn, TNULLn, TSCALn, TZEROn, TDISPn, TDIMn, TLMINn, TLMAXn, TDMINn, TDMAXn, TCTYPn, TCRPXn, TCRVLn, TCDLTn, TCROTn, and TCUNIn.

```
FTICOL(unit,colnum,ttype,tform, > status)
FTICLS(unit,colnum,ncols,ttype,tform, > status)
```

5 Modify the vector length of a binary table column (e.g., change a column from TFORMn = '1E' to '20E'). The vector length may be increased or decreased from the current value.

```
FTMVEC(unit,colnum,newveclen, > status)
```

6 Delete a column from an existing ASCII or binary table (in the CDU). The index number of all the keywords listed above (for FTICOL) will be decremented if necessary to reflect the new position of the column(s) in the table. Those index keywords that refer to the deleted column will also be deleted. Note that the physical size of the FITS file will not be reduced by this operation, and the empty FITS blocks if any at the end of the file will be padded with zeros.

```
FTDCOL(unit,colnum, > status)
```

7 Copy a column from one HDU to another (or to the same HDU). If createcol = TRUE, then a new column will be inserted in the output table, at position 'outcolumn', otherwise the existing output column will be overwritten (in which case it must have a compatible datatype). Note that the first column in a table is at colnum = 1.

```
FTCPCL(inunit,outunit,incolnum,outcolnum,createcol, > status);
```

6.7.4 Read and Write Column Data Routines

These subroutines put or get data values in the current ASCII or Binary table extension. Automatic data type conversion is performed for numerical data types (B,I,J,E,D) if the data type of the column (defined by the TFORM keyword) differs from the data type of the calling subroutine. The data values are also scaled by the TSCALn and TZEROn header values as they are being written to or read from the FITS array. The fttscl subroutine MUST be used to define the scaling parameters when writing data to the table or to override the default scaling values given in the header when reading from the table. Note that it is *not* necessary to insert rows in a table before writing data to those rows (indeed, it would be inefficient to do so). Instead, one may simply write data to any row of the table, whether that row of data already exists or not.

In the case of binary tables with vector elements, the 'felem' parameter defines the starting pixel within the element vector. This parameter is ignored with ASCII tables. Similarly, in the case of binary tables the 'nelements' parameter specifies the total number of vector values read or written (continuing on subsequent rows if required) and not the number of table elements. Two sets of subroutines are provided to get the column data which differ in the way undefined pixels are handled. The first set of routines (FTGCV) simply return an array of data elements in which undefined pixels are set equal to a value specified by the user in the 'nullval' parameter. An additional feature of these subroutines is that if the user sets nullval = 0, then no checks for undefined pixels will be performed, thus increasing the speed of the program. The second set of routines (FTGCF) returns the data element array and in addition a logical array of flags which defines whether the corresponding data pixel is undefined.

Any column, regardless of it's intrinsic datatype, may be read as a string. It should be noted however that reading a numeric column as a string is 10 - 100 times slower than reading the same column as a number due to the large overhead in constructing the formatted strings. The display format of the returned strings will be determined by the TDISPn keyword, if it exists, otherwise by the datatype of the column. The length of the returned strings can be determined with the ftgcdw routine. The following TDISPn display formats are currently supported:

```
Iw.m   Integer
Ow.m   Octal integer
Zw.m   Hexadecimal integer
Fw.d   Fixed floating point
Ew.d   Exponential floating point
Dw.d   Exponential floating point
Gw.d   General; uses Fw.d if significance not lost, else Ew.d
```

where w is the width in characters of the displayed values, m is the minimum number of digits displayed, and d is the number of digits to the right of the decimal. The .m field is optional.

1 Put elements into an ASCII or binary table column (in the CDU). (The SPP FSPCLS routine has an additional integer argument after the VALUES character string which specifies the size of the 1st dimension of this 2-D CHAR array).

The alternate version of these routines, whose names end in 'LL' after the datatype character, support large tables with more then 2*31 rows. When calling these routines, the frow and felem parameters *must* be 64-bit integer*8 variables, instead of normal 4-byte integers.

```
FTPCL[SLBIJKEDCM](unit,colnum,frow,felem,nelements,values, > status)
FTPCL[LBIJKEDCM]LL(unit,colnum,frow,felem,nelements,values, > status)
```

2 Put elements into an ASCII or binary table column (in the CDU) substituting the appropriate FITS null value for any elements that are equal to NULLVAL. For ASCII TABLE extensions, the null value defined by the previous call to FTSNUL will be substituted; For integer FITS columns, in a binary table the null value defined by the previous call to FTTNUL will be substituted; For floating point FITS columns a special IEEE NaN (Not-a-Number) value will be substituted.

The alternate version of these routines, whose names end in 'LL' after the datatype character, support large tables with more then 2*31 rows. When calling these routines, the frow and felem parameters *must* be 64-bit integer*8 variables, instead of normal 4-byte integers.

```
FTPCN[SBIJKED](unit,colnum,frow,felem,nelements,values,nullval > status)
FTPCN[SBIJKED]LL(unit,colnum,(I*8) frow,(I*8) felem,nelements,values,
        nullval > status)
```

3 Put bit values into a binary byte ('B') or bit ('X') table column (in the CDU). LRAY is an array of logical values corresponding to the sequence of bits to be written. If LRAY is true then the corresponding bit is set to 1, otherwise the bit is set to 0. Note that in the case of 'X' columns, FITSIO will write to all 8 bits of each byte whether they are formally valid or not. Thus if the column is defined as '4X', and one calls FTPCLX with fbit=1 and nbit=8, then all 8 bits will be written into the first byte (as opposed to writing the first 4 bits into the first row and then the next 4 bits into the next row), even though the last 4 bits of each byte are formally not defined.

```
FTPCLX(unit,colnum,frow,fbit,nbit,lray, > status)
```

4 Set table elements in a column as undefined

```
FTPCLU(unit,colnum,frow,felem,nelements, > status)
```

5 Get elements from an ASCII or binary table column (in the CDU). These routines return the values of the table column array elements. Undefined array elements will be returned with a value = nullval, unless nullval = 0 (or = ' ' for ftgcvs) in which case no checking for undefined values will be performed. The ANYF parameter is set to true if any of the returned elements are undefined. (Note: the ftgcl routine simple gets an array of logical data values without any checks for undefined values; use the ftgcfl routine to check for undefined logical elements). (The SPP FSGCVS routine has an additional integer argument after the VALUES character string which specifies the size of the 1st dimension of this 2-D CHAR array).

The alternate version of these routines, whose names end in 'LL' after the datatype character, support large tables with more then 2*31 rows. When calling these routines, the frow and felem parameters *must* be 64-bit integer*8 variables, instead of normal 4-byte integers.

```
FTGCL(unit,colnum,frow,felem,nelements, > values,status)
FTGCV[SBIJKEDCM](unit,colnum,frow,felem,nelements,nullval, >
                 values,anyf,status)
FTGCV[BIJKEDCM]LL(unit,colnum,(I*8) frow, (I*8) felem, nelements,
        nullval, > values,anyf,status)
```

6 Get elements and null flags from an ASCII or binary table column (in the CHDU). These routines return the values of the table column array elements. Any undefined array elements will have the corresponding flagvals element set equal to .TRUE. The ANYF parameter is set to true if any of the returned elements are undefined. (The SPP FSGCFS routine has an additional integer argument after the VALUES character string which specifies the size of the 1st dimension of this 2-D CHAR array).

The alternate version of these routines, whose names end in 'LL' after the datatype character, support large tables with more then 2*31 rows. When calling these routines, the frow and felem parameters *must* be 64-bit integer*8 variables, instead of normal 4-byte integers.

```
FTGCF[SLBIJKEDCM](unit,colnum,frow,felem,nelements, >
                   values,flagvals,anyf,status)
FTGCF[BIJKED]LL(unit,colnum, (I*8) frow, (I*8) felem,nelements, >
                   values,flagvals,anyf,status)
```

7 Get an arbitrary data subsection from an N-dimensional array in a binary table vector column. Undefined pixels in the array will be set equal to the value of 'nullval', unless nullval=0 in which case no testing for undefined pixels will be performed. The first and last rows in the table to be read are specified by fpixels(naxis+1) and lpixels(naxis+1), and hence are treated as the next higher dimension of the FITS N-dimensional array. The INCS parameter specifies the sampling interval in each dimension between the data elements that will be returned.

```
FTGSV[BIJKED](unit,colnum,naxis,naxes,fpixels,lpixels,incs,nullval, >
               array,anyf,status)
```

8 Get an arbitrary data subsection from an N-dimensional array in a binary table vector column. Any Undefined pixels in the array will have the corresponding 'flagvals' element set equal to .TRUE. The first and last rows in the table to be read are specified by fpixels(naxis+1) and lpixels(naxis+1), and hence are treated as the next higher dimension of the FITS N-dimensional array. The INCS parameter specifies the sampling interval in each dimension between the data elements that will be returned.

```
FTGSF[BIJKED](unit,colnum,naxis,naxes,fpixels,lpixels,incs, >
             array,flagvals,anyf,status)
```

9 Get bit values from a byte ('B') or bit ('X') table column (in the CDU). LRAY is an array of logical values corresponding to the sequence of bits to be read. If LRAY is true then the corresponding bit was set to 1, otherwise the bit was set to 0. Note that in the case of 'X' columns, FITSIO will read all 8 bits of each byte whether they are formally valid or not. Thus if the column is defined as '4X', and one calls FTGCX with fbit=1 and nbit=8, then all 8 bits will be read from the first byte (as opposed to reading the first 4 bits from the first row and then the first 4 bits from the next row), even though the last 4 bits of each byte are formally not defined.

```
FTGCX(unit,colnum,frow,fbit,nbit, > lray,status)
```

10 Read any consecutive set of bits from an 'X' or 'B' column and interpret them as an unsigned n-bit integer. NBIT must be less than or equal to 16 when calling FTGCXI, and less than or equal to 32 when calling FTGCXJ; there is no limit on the value of NBIT for FTGCXD, but the returned double precision value only has 48 bits of precision on most 32-bit word machines. The NBITS bits are interpreted as an unsigned integer unless NBITS = 16 (in FTGCXI) or 32 (in FTGCXJ) in which case the string of bits are interpreted as 16-bit or 32-bit 2's complement signed integers. If NROWS is greater than 1 then the same set of bits will be read from sequential rows in the table starting with row FROW. Note that the numbering convention used here for the FBIT parameter adopts 1 for the first element of the vector of bits; this is the Most Significant Bit of the integer value.

```
FTGCX[IJD](unit,colnum,frow,nrows,fbit,nbit, > array,status)
```

11 Get the descriptor for a variable length column in a binary table. The descriptor consists of 2 integer parameters: the number of elements in the array and the starting offset relative to the start of the heap. The first routine returns a single descriptor whereas the second routine returns the descriptors for a range of rows in the table.

```
FTGDES(unit,colnum,rownum, > nelements,offset,status)
FTGDESLL(unit,colnum,rownum, > nelementsll,offsetll,status)

FFGDESS(unit,colnum,firstrow,nrows > nelements,offset, status)
FFGDESSLL(unit,colnum,firstrow,nrows > nelementsll,offsetll, status)
```

12 Write the descriptor for a variable length column in a binary table. These subroutines can be used in conjunction with FTGDES to enable 2 or more arrays to point to the same storage location to save storage space if the arrays are identical.

```
FTPDES(unit,colnum,rownum,nelements,offset, > status)
FTPDESLL(unit,colnum,rownum,nelementsll,offsetll, > status)
```

6.8 Row Selection and Calculator Routines

These routines all parse and evaluate an input string containing a user defined arithmetic expression. The first 3 routines select rows in a FITS table, based on whether the expression evaluates to true (not equal to zero) or false (zero). The other routines evaluate the expression and calculate a value for each row of the table. The allowed expression syntax is described in the row filter section in the earlier 'Extended File Name Syntax' chapter of this document. The expression may also be written to a text file, and the name of the file, prepended with a '@' character may be supplied for the 'expr' parameter (e.g. '@filename.txt'). The expression in the file can be arbitrarily complex and extend over multiple lines of the file. Lines that begin with 2 slash characters ('//') will be ignored and may be used to add comments to the file.

1 Evaluate a boolean expression over the indicated rows, returning an array of flags indicating which rows evaluated to TRUE/FALSE

```
FTFROW(unit,expr,firstrow, nrows, > n_good_rows, row_status, status)
```

2 Find the first row which satisfies the input boolean expression

```
FTFFRW(unit, expr, > rownum, status)
```

3 Evaluate an expression on all rows of a table. If the input and output files are not the same, copy the TRUE rows to the output file; if the output table is not empty, then this routine will append the new selected rows after the existing rows. If the files are the same, delete the FALSE rows (preserve the TRUE rows).

```
FTSROW(inunit, outunit, expr, > status)
```

4 Calculate an expression for the indicated rows of a table, returning the results, cast as datatype (TSHORT, TDOUBLE, etc), in array. If nulval==NULL, UNDEFs will be zeroed out. For vector results, the number of elements returned may be less than nelements if nelements is not an even multiple of the result dimension. Call FTTEXP to obtain the dimensions of the results.

```
FTCROW(unit,datatype,expr,firstrow,nelements,nulval, >
    array,anynul,status)
```

5 Evaluate an expression and write the result either to a column (if the expression is a function of other columns in the table) or to a keyword (if the expression evaluates to a constant and is not a function of other columns in the table). In the former case, the parName parameter is the name of the column (which may or may not already exist) into which to write the results, and parInfo contains an optional TFORM keyword value if a new column is being created. If a TFORM value is not specified then a default format will be used, depending on the expression. If the expression evaluates to a constant, then the result will be written to the keyword name given by the parName parameter, and the parInfo parameter may be used to supply an optional comment for the keyword. If the keyword does not already exist, then the name of the keyword must be preceded with a '#' character, otherwise the result will be written to a column with that name.

```
FTCALC(inunit, expr, outunit, parName, parInfo, > status)
```

6 This calculator routine is similar to the previous routine, except that the expression is only evaluated over the specified row ranges. nranges specifies the number of row ranges, and firstrow and lastrow give the starting and ending row number of each range.

```
FTCALC_RNG(inunit, expr, outunit, parName, parInfo,
   nranges, firstrow, lastrow, > status)
```

7 Evaluate the given expression and return dimension and type information on the result. The returned dimensions correspond to a single row entry of the requested expression, and are equivalent to the result of fits_read_tdim(). Note that strings are considered to be one element regardless of string length. If maxdim == 0, then naxes is optional.

```
FTTEXP(unit, expr, maxdim > datatype, nelem, naxis, naxes, status)
```

6.9 Celestial Coordinate System Subroutines

The FITS community has adopted a set of keyword conventions that define the transformations needed to convert between pixel locations in an image and the corresponding celestial coordinates on the sky, or more generally, that define world coordinates that are to be associated with any pixel location in an n-dimensional FITS array. CFITSIO is distributed with a couple of self-contained World Coordinate System (WCS) routines, however, these routines DO NOT support all the latest WCS conventions, so it is STRONGLY RECOMMENDED that software developers use a more robust external WCS library. Several recommended libraries are:

```
WCSLIB -  supported by Mark Calabretta
WCSTools - supported by Doug Mink
AST library - developed by the U.K. Starlink project
```

More information about the WCS keyword conventions and links to all of these WCS libraries can be found on the FITS Support Office web site at http://fits.gsfc.nasa.gov under the WCS link.

The functions provided in these external WCS libraries will need access to the WCS information contained in the FITS file headers. One convenient way to pass this information to the external library is to use FITSIO to copy the header keywords into one long character string, and then pass this string to an interface routine in the external library that will extract the necessary WCS information (e.g., see the astFitsChan and astPutCards routines in the Starlink AST library).

The following FITSIO routines DO NOT support the more recent WCS conventions that have been approved as part of the FITS standard. Consequently, the following routines ARE NOW DEPRECATED. It is STRONGLY RECOMMENDED that software developers not use these routines, and instead use an external WCS library, as described above.

These routines are included mainly for backward compatibility with existing software. They support the following standard map projections: -SIN, -TAN, -ARC, -NCP, -GLS, -MER, and -AIT (these are the legal values for the coordtype parameter). These routines are based on similar functions in Classic AIPS. All the angular quantities are given in units of degrees.

1 Get the values of all the standard FITS celestial coordinate system keywords from the header of a FITS image (i.e., the primary array or an image extension). These values may then be passed to the subroutines that perform the coordinate transformations. If any or all of the WCS keywords are not present, then default values will be returned. If the first coordinate axis is the declination-like coordinate, then this routine will swap them so that the longitudinal-like coordinate is returned as the first axis.

If the file uses the newer 'CDj_i' WCS transformation matrix keywords instead of old style 'CDELTn' and 'CROTA2' keywords, then this routine will calculate and return the values of the equivalent old-style keywords. Note that the conversion from the new-style keywords to the old-style values is sometimes only an approximation, so if the approximation is larger than an internally defined threshold level, then CFITSIO will still return the approximate WCS keyword values, but will also return with status = 506, to warn the calling program that approximations have been made. It is then up to the calling program to decide whether the approximations are sufficiently accurate for the particular application, or whether more precise WCS transformations must be performed using new-style WCS keywords directly.

```
FTGICS(unit, > xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
```

2 Get the values of all the standard FITS celestial coordinate system keywords from the header of a FITS table where the X and Y (or RA and DEC coordinates are stored in 2 separate columns of the table. These values may then be passed to the subroutines that perform the coordinate transformations.

```
FTGTCS(unit,xcol,ycol, >
       xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
```

3 Calculate the celestial coordinate corresponding to the input X and Y pixel location in the image.

```
FTWLDP(xpix,ypix,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                 coordtype, > xpos,ypos,status)
```

4 Calculate the X and Y pixel location corresponding to the input celestial coordinate in the image.

```
FTXYPX(xpos,ypos,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                   coordtype, > xpix,ypix,status)
```

6.10 File Checksum Subroutines

The following routines either compute or validate the checksums for the CHDU. The DATASUM keyword is used to store the numerical value of the 32-bit, 1's complement checksum for the data unit alone. If there is no data unit then the value is set to zero. The numerical value is stored as an ASCII string of digits, enclosed in quotes, because the value may be too large to represent as a 32-bit signed integer. The CHECKSUM keyword is used to store the ASCII encoded COMPLEMENT of the checksum for the entire HDU. Storing the complement, rather than the actual checksum, forces the checksum for the whole HDU to equal zero. If the file has been modified since the checksums were computed, then the HDU checksum will usually not equal zero. These checksum keyword conventions are based on a paper by Rob Seaman published in the proceedings of the ADASS IV conference in Baltimore in November 1994 and a later revision in June 1995.

1 Compute and write the DATASUM and CHECKSUM keyword values for the CHDU into the current header. The DATASUM value is the 32-bit checksum for the data unit, expressed as a decimal integer enclosed in single quotes. The CHECKSUM keyword value is a 16-character string which is the ASCII-encoded value for the complement of the checksum for the whole HDU. If these keywords already exist, their values will be updated only if necessary (i.e., if the file has been modified since the original keyword values were computed).

```
FTPCKS(unit, > status)
```

2 Update the CHECKSUM keyword value in the CHDU, assuming that the DATASUM keyword exists and already has the correct value. This routine calculates the new checksum for the current header unit, adds it to the data unit checksum, encodes the value into an ASCII string, and writes the string to the CHECKSUM keyword.

```
FTUCKS(unit, > status)
```

3 Verify the CHDU by computing the checksums and comparing them with the keywords. The data unit is verified correctly if the computed checksum equals the value of the DATASUM keyword. The checksum for the entire HDU (header plus data unit) is correct if it equals zero. The output DATAOK and HDUOK parameters in this subroutine are integers which will have a value = 1 if the data or HDU is verified correctly, a value = 0 if the DATASUM or CHECKSUM keyword is not present, or value = -1 if the computed checksum is not correct.

```
FTVCKS(unit, > dataok,hduok,status)
```

4 Compute and return the checksum values for the CHDU (as double precision variables) without creating or modifying the CHECKSUM and DATASUM keywords. This routine is used internally by FTVCKS, but may be useful in other situations as well.

```
FTGCKS(unit, > datasum,hdusum,status)
```

5 Encode a checksum value (stored in a double precision variable) into a 16-character string. If COMPLEMENT = .true. then the 32-bit sum value will be complemented before encoding.

```
FTESUM(sum,complement, > checksum)
```

6 Decode a 16 character checksum string into a double precision value. If COMPLEMENT = .true. then the 32-bit sum value will be complemented after decoding.

```
FTDSUM(checksum,complement, > sum)
```

6.11 Date and Time Utility Routines

The following routines help to construct or parse the FITS date/time strings. Starting in the year 2000, the FITS DATE keyword values (and the values of other 'DATE-' keywords) must have the form 'YYYY-MM-DD' (date only) or 'YYYY-MM-DDThh:mm:ss.ddd...' (date and time) where the number of decimal places in the seconds value is optional. These times are in UTC. The older 'dd/mm/yy' date format may not be used for dates after 01 January 2000.

1 Get the current system date. The returned year has 4 digits (1999, 2000, etc.)

```
FTGSDT( > day, month, year, status )
```

2 Get the current system date and time string ('YYYY-MM-DDThh:mm:ss'). The time will be in UTC/GMT if available, as indicated by a returned timeref value = 0. If the returned value of timeref = 1 then this indicates that it was not possible to convert the local time to UTC, and thus the local time was returned.

```
FTGSTM(> datestr, timeref, status)
```

3 Construct a date string from the input date values. If the year is between 1900 and 1998, inclusive, then the returned date string will have the old FITS format ('dd/mm/yy'), otherwise the date string will have the new FITS format ('YYYY-MM-DD'). Use FTTM2S instead to always return a date string using the new FITS format.

```
FTDT2S( year, month, day, > datestr, status)
```

4 Construct a new-format date + time string ('YYYY-MM-DDThh:mm:ss.ddd...'). If the year, month, and day values all = 0 then only the time is encoded with format 'hh:mm:ss.ddd...'. The decimals parameter specifies how many decimal places of fractional seconds to include in the string. If 'decimals' is negative, then only the date will be return ('YYYY-MM-DD').

```
FTTM2S( year, month, day, hour, minute, second, decimals,
        > datestr, status)
```

5 Return the date as read from the input string, where the string may be in either the old ('dd/mm/yy') or new ('YYYY-MM-DDThh:mm:ss' or 'YYYY-MM-DD') FITS format.

```
FTS2DT(datestr, > year, month, day, status)
```

6 Return the date and time as read from the input string, where the string may be in either the old or new FITS format. The returned hours, minutes, and seconds values will be set to zero if the input string does not include the time ('dd/mm/yy' or 'YYYY-MM-DD') . Similarly, the returned year, month, and date values will be set to zero if the date is not included in the input string ('hh:mm:ss.ddd...').

```
FTS2TM(datestr, > year, month, day, hour, minute, second, status)
```

6.12 General Utility Subroutines

The following utility subroutines may be useful for certain applications:

1 Return the starting byte address of the CHDU and the next HDU.

```
FTGHAD(iunit, > curaddr, nextaddr)
```

2 Convert a character string to uppercase (operates in place).

```
FTUPCH(string)
```

3 Compare the input template string against the reference string to see if they match. The template string may contain wildcard characters: '*' will match any sequence of characters (including zero characters) and '?' will match any single character in the reference string. The '#' character will match any consecutive string of decimal digits (0 - 9). If CASESN = .true. then the match will be case sensitive. The returned MATCH parameter will be .true. if the 2 strings match, and EXACT will be .true. if the match is exact (i.e., if no wildcard characters were used in the match). Both strings must be 68 characters or less in length.

```
FTCMPS(str_template, string, casesen, > match, exact)
```

4 Test that the keyword name contains only legal characters: A-Z,0-9, hyphen, and underscore.

```
FTTKEY(keyword, > status)
```

5 Test that the keyword record contains only legal printable ASCII characters

```
FTTREC(card, > status)
```

6 Test whether the current header contains any NULL (ASCII 0) characters. These characters are illegal in the header, but they will go undetected by most of the CFITSIO keyword header routines, because the null is interpreted as the normal end-of-string terminator. This routine returns the position of the first null character in the header, or zero if there are no nulls. For example a returned value of 110 would indicate that the first NULL is located in the 30th character of the second keyword in the header (recall that each header record is 80 characters long). Note that this is one of the few FITSIO routines in which the returned value is not necessarily equal to the status value).

```
FTNCHK(unit, > status)
```

7 Parse a header keyword record and return the name of the keyword and the length of the name. The keyword name normally occupies the first 8 characters of the record, except under the HIERARCH convention where the name can be up to 70 characters in length.

```
FTGKNM(card, > keyname, keylength, staThe '\#' character will match any consecutive string
of decimal digits (0 - 9). tus)
```

8 Parse a header keyword record. This subroutine parses the input header record to return the value (as a character string) and comment strings. If the keyword has no value (columns 9-10 not equal to '= '), then the value string is returned blank and the comment string is set equal to column 9 - 80 of the input string.

```
FTPSVC(card, > value,comment,status)
```

9 Construct a sequence keyword name (ROOT + nnn). This subroutine appends the sequence number to the root string to create a keyword name (e.g., 'NAXIS' + 2 = 'NAXIS2')

```
FTKEYN(keyroot,seq_no, > keyword,status)
```

10 Construct a sequence keyword name (n + ROOT). This subroutine concatenates the sequence number to the front of the root string to create a keyword name (e.g., 1 + 'CTYP' = '1CTYP')

```
FTNKEY(seq_no,keyroot, > keyword,status)
```

11 Determine the datatype of a keyword value string. This subroutine parses the keyword value string (usually columns 11-30 of the header record) to determine its datatype.

```
FTDTYP(value, > dtype,status)
```

11 Return the class of input header record. The record is classified into one of the following categories (the class values are defined in fitsio.h). Note that this is one of the few FITSIO routines that does not return a status value.

```
     Class  Value             Keywords
  TYP_STRUC_KEY  10  SIMPLE, BITPIX, NAXIS, NAXISn, EXTEND, BLOCKED,
                     GROUPS, PCOUNT, GCOUNT, END
                     XTENSION, TFIELDS, TTYPEn, TBCOLn, TFORMn, THEAP,
                     and the first 4 COMMENT keywords in the primary array
                     that define the FITS format.
  TYP_CMPRS_KEY  20  The experimental keywords used in the compressed
                     image format ZIMAGE, ZCMPTYPE, ZNAMEn, ZVALn,
                     ZTILEn, ZBITPIX, ZNAXISn, ZSCALE, ZZERO, ZBLANK
  TYP_SCAL_KEY   30  BSCALE, BZERO, TSCALn, TZEROn
  TYP_NULL_KEY   40  BLANK, TNULLn
  TYP_DIM_KEY    50  TDIMn
  TYP_RANG_KEY   60  TLMINn, TLMAXn, TDMINn, TDMAXn, DATAMIN, DATAMAX
  TYP_UNIT_KEY   70  BUNIT, TUNITn
  TYP_DISP_KEY   80  TDISPn
  TYP_HDUID_KEY  90  EXTNAME, EXTVER, EXTLEVEL, HDUNAME, HDUVER, HDULEVEL
  TYP_CKSUM_KEY 100  CHECKSUM, DATASUM
  TYP_WCS_KEY   110  CTYPEn, CUNITn, CRVALn, CRPIXn, CROTAn, CDELTn
                     CDj_is, PVj_ms, LONPOLEs, LATPOLEs
                     TCTYPn, TCTYns, TCUNIn, TCUNns, TCRVLn, TCRVns, TCRPXn,
                     TCRPks, TCDn_k, TCn_ks, TPVn_m, TPn_ms, TCDLTn, TCROTn
                     jCTYPn, jCTYns, jCUNIn, jCUNns, jCRVLn, jCRVns, iCRPXn,
                     iCRPns, jiCDn,  jiCDns, jPVn_m, jPn_ms, jCDLTn, jCROTn
                     (i,j,m,n are integers, s is any letter)
  TYP_REFSYS_KEY 120 EQUINOXs, EPOCH, MJD-OBSs, RADECSYS, RADESYSs
  TYP_COMM_KEY   130 COMMENT, HISTORY, (blank keyword)
  TYP_CONT_KEY   140 CONTINUE
  TYP_USER_KEY   150 all other keywords

        class = FTGKCL (char *card)
```

12 Parse the 'TFORM' binary table column format string. This subroutine parses the input TFORM character string and returns the integer datatype code, the repeat count of the field, and, in the case of character string fields, the length of the unit string. The following datatype codes are returned (the negative of the value is returned if the column contains variable-length arrays):

```
Datatype               DATACODE value
bit, X                  1
byte, B                11
logical, L             14
ASCII character, A     16
short integer, I       21
integer, J             41
real, E                42
double precision, D    82
complex                83
double complex         163
```

```
FTBNFM(tform, > datacode,repeat,width,status)
```

13 Parse the 'TFORM' keyword value that defines the column format in an ASCII table. This routine parses the input TFORM character string and returns the datatype code, the width of the column, and (if it is a floating point column) the number of decimal places to the right of the decimal point. The returned datatype codes are the same as for the binary table, listed above, with the following additional rules: integer columns that are between 1 and 4 characters wide are defined to be short integers (code = 21). Wider integer columns are defined to be regular integers (code = 41). Similarly, Fixed decimal point columns (with TFORM = 'Fw.d') are defined to be single precision reals (code = 42) if w is between 1 and 7 characters wide, inclusive. Wider 'F' columns will return a double precision data code (= 82). 'Ew.d' format columns will have datacode = 42, and 'Dw.d' format columns will have datacode = 82.

```
FTASFM(tform, > datacode,width,decimals,status)
```

14 Calculate the starting column positions and total ASCII table width based on the input array of ASCII table TFORM values. The SPACE input parameter defines how many blank spaces to leave between each column (it is recommended to have one space between columns for better human readability).

```
FTGABC(tfields,tform,space, > rowlen,tbcol,status)
```

15 Parse a template string and return a formatted 80-character string suitable for appending to (or deleting from) a FITS header file. This subroutine is useful for parsing lines from an ASCII template file and reformatting them into legal FITS header records. The formatted string may then be passed to the FTPREC, FTMCRD, or FTDKEY subroutines to append or modify a FITS header record.

```
FTGTHD(template, > card,hdtype,status)
```

The input TEMPLATE character string generally should contain 3 tokens: (1) the KEYNAME, (2) the VALUE, and (3) the COMMENT string. The TEMPLATE string must adhere to the following format:

- The KEYNAME token must begin in columns 1-8 and be a maximum of 8 characters long. If the first 8 characters of the template line are blank then the remainder of the line is considered to be a FITS comment (with a blank keyword name). A legal FITS keyword name may only contain the characters A-Z, 0-9, and '-' (minus sign) and underscore. This subroutine will automatically convert any lowercase characters to uppercase in the output string. If KEYNAME = 'COMMENT' or 'HISTORY' then the remainder of the line is considered to be a FITS COMMENT or HISTORY record, respectively.

- The VALUE token must be separated from the KEYNAME token by one or more spaces and/or an '=' character. The datatype of the VALUE token (numeric, logical, or character string) is automatically determined and the output CARD string is formatted accordingly. The value token may be forced to be interpreted as a string (e.g. if it is a string of numeric digits) by enclosing it in single quotes.

- The COMMENT token is optional, but if present must be separated from the VALUE token by at least one blank space. A leading '/' character may be used to mark the beginning of the comment field, otherwise the comment field begins with the first non-blank character following the value token.

- One exception to the above rules is that if the first non-blank character in the template string is a minus sign ('-') followed by a single token, or a single token followed by an equal sign, then it is interpreted as the name of a keyword which is to be deleted from the FITS header.

- The second exception is that if the template string starts with a minus sign and is followed by 2 tokens then the second token is interpreted as the new name for the keyword specified by first token. In this case the old keyword name (first token) is returned in characters 1-8 of the returned CARD string, and the new keyword name (the second token) is returned in characters 41-48 of the returned CARD string. These old and new names may then be passed to the FTMNAM subroutine which will change the keyword name.

The HDTYPE output parameter indicates how the returned CARD string should be interpreted:

```
hdtype                 interpretation
------         -------------------------------------------------
   -2          Modify the name of the keyword given in CARD(1:8)
               to the new name given in CARD(41:48)

   -1          CARD(1:8) contains the name of a keyword to be deleted
               from the FITS header.

    0          append the CARD string to the FITS header if the
               keyword does not already exist, otherwise update
               the value/comment if the keyword is already present
               in the header.
```

```
        1          simply append this keyword to the FITS header (CARD
                   is either a HISTORY or COMMENT keyword).

        2          This is a FITS END record; it should not be written
                   to the FITS header because FITSIO automatically
                   appends the END record when the header is closed.
```

EXAMPLES: The following lines illustrate valid input template strings:

```
INTVAL 7 This is an integer keyword
RVAL           34.6   /     This is a floating point keyword
EVAL=-12.45E-03  This is a floating point keyword in exponential notation
lval F This is a boolean keyword
            This is a comment keyword with a blank keyword name
SVAL1 = 'Hello world'   /  this is a string keyword
SVAL2  '123.5'  this is also a string keyword
sval3  123+  /  this is also a string keyword with the value '123+    '
# the following template line deletes the DATE keyword
- DATE
# the following template line modifies the NAME keyword to OBJECT
- NAME OBJECT
```

16 Parse the input string containing a list of rows or row ranges, and return integer arrays containing the first and last row in each range. For example, if rowlist = "3-5, 6, 8-9" then it will return numranges = 3, rangemin = 3, 6, 8 and rangemax = 5, 6, 9. At most, 'maxranges' number of ranges will be returned. 'maxrows' is the maximum number of rows in the table; any rows or ranges larger than this will be ignored. The rows must be specified in increasing order, and the ranges must not overlap. A minus sign may be use to specify all the rows to the upper or lower bound, so "50-" means all the rows from 50 to the end of the table, and "-" means all the rows in the table, from 1 - maxrows.

```
FTRWRG(rowlist, maxrows, maxranges, >
       numranges, rangemin, rangemax, status)
```

Chapter 7

The CFITSIO Iterator Function

The fits_iterate_data function in CFITSIO provides a unique method of executing an arbitrary user-supplied 'work' function that operates on rows of data in FITS tables or on pixels in FITS images. Rather than explicitly reading and writing the FITS images or columns of data, one instead calls the CFITSIO iterator routine, passing to it the name of the user's work function that is to be executed along with a list of all the table columns or image arrays that are to be passed to the work function. The CFITSIO iterator function then does all the work of allocating memory for the arrays, reading the input data from the FITS file, passing them to the work function, and then writing any output data back to the FITS file after the work function exits. Because it is often more efficient to process only a subset of the total table rows at one time, the iterator function can determine the optimum amount of data to pass in each iteration and repeatedly call the work function until the entire table been processed.

For many applications this single CFITSIO iterator function can effectively replace all the other CFITSIO routines for reading or writing data in FITS images or tables. Using the iterator has several important advantages over the traditional method of reading and writing FITS data files:

- It cleanly separates the data I/O from the routine that operates on the data. This leads to a more modular and 'object oriented' programming style.
- It simplifies the application program by eliminating the need to allocate memory for the data arrays and eliminates most of the calls to the CFITSIO routines that explicitly read and write the data.
- It ensures that the data are processed as efficiently as possible. This is especially important when processing tabular data since the iterator function will calculate the most efficient number of rows in the table to be passed at one time to the user's work function on each iteration.
- Makes it possible for larger projects to develop a library of work functions that all have a uniform calling sequence and are all independent of the details of the FITS file format.

There are basically 2 steps in using the CFITSIO iterator function. The first step is to design the work function itself which must have a prescribed set of input parameters. One of these parameters is a structure containing pointers to the arrays of data; the work function can perform any desired operations on these arrays and does not need to worry about how the input data were read from the file or how the output data get written back to the file.

The second step is to design the driver routine that opens all the necessary FITS files and initializes the input parameters to the iterator function. The driver program calls the CFITSIO iterator function which then reads the data and passes it to the user's work function.

Further details on using the iterator function can be found in the companion CFITSIO User's Guide, and in the iter_a.f, iter_b.f and iter_c.f example programs.

Chapter 8

Extended File Name Syntax

8.1 Overview

CFITSIO supports an extended syntax when specifying the name of the data file to be opened or created that includes the following features:

- CFITSIO can read IRAF format images which have header file names that end with the '.imh' extension, as well as reading and writing FITS files, This feature is implemented in CFITSIO by first converting the IRAF image into a temporary FITS format file in memory, then opening the FITS file. Any of the usual CFITSIO routines then may be used to read the image header or data. Similarly, raw binary data arrays can be read by converting them on the fly into virtual FITS images.
- FITS files on the Internet can be read (and sometimes written) using the FTP, HTTP, or ROOT protocols.
- FITS files can be piped between tasks on the stdin and stdout streams.
- FITS files can be read and written in shared memory. This can potentially achieve much better data I/O performance compared to reading and writing the same FITS files on magnetic disk.
- Compressed FITS files in gzip or Unix COMPRESS format can be directly read.
- Output FITS files can be written directly in compressed gzip format, thus saving disk space.
- FITS table columns can be created, modified, or deleted 'on-the-fly' as the table is opened by CFITSIO. This creates a virtual FITS file containing the modifications that is then opened by the application program.
- Table rows may be selected, or filtered out, on the fly when the table is opened by CFITSIO, based on an arbitrary user-specified expression. Only rows for which the expression evaluates to 'TRUE' are retained in the copy of the table that is opened by the application program.
- Histogram images may be created on the fly by binning the values in table columns, resulting in a virtual N-dimensional FITS image. The application program then only sees the FITS image (in the primary array) instead of the original FITS table.

The latter 3 features in particular add very powerful data processing capabilities directly into CFITSIO, and hence into every task that uses CFITSIO to read or write FITS files. For example, these features transform a very simple program that just copies an input FITS file to a new output file (like the 'fitscopy' program that is distributed with CFITSIO) into a multipurpose FITS file processing tool. By appending fairly simple qualifiers onto the name of the input FITS file, the user can perform quite complex table editing operations (e.g., create new columns, or filter out rows in a table) or create FITS images by binning or histogramming the values in table columns. In addition, these functions have been coded using new state-of-the art algorithms that are, in some cases, 10 - 100 times faster than previous widely used implementations.

Before describing the complete syntax for the extended FITS file names in the next section, here are a few examples of FITS file names that give a quick overview of the allowed syntax:

- `'myfile.fits'`: the simplest case of a FITS file on disk in the current directory.
- `'myfile.imh'`: opens an IRAF format image file and converts it on the fly into a temporary FITS format image in memory which can then be read with any other CFITSIO routine.
- `rawfile.dat[i512,512]`: opens a raw binary data array (a 512 x 512 short integer array in this case) and converts it on the fly into a temporary FITS format image in memory which can then be read with any other CFITSIO routine.
- `myfile.fits.gz`: if this is the name of a new output file, the '.gz' suffix will cause it to be compressed in gzip format when it is written to disk.
- `'myfile.fits.gz[events, 2]'`: opens and uncompresses the gzipped file myfile.fits then moves to the extension which has the keywords EXTNAME = 'EVENTS' and EXTVER = 2.
- `'-'`: a dash (minus sign) signifies that the input file is to be read from the stdin file stream, or that the output file is to be written to the stdout stream.
- `'ftp://legacy.gsfc.nasa.gov/test/vela.fits'`: FITS files in any ftp archive site on the Internet may be directly opened with read-only access.
- `'http://legacy.gsfc.nasa.gov/software/test.fits'`: any valid URL to a FITS file on the Web may be opened with read-only access.
- `'root://legacy.gsfc.nasa.gov/test/vela.fits'`: similar to ftp access except that it provides write as well as read access to the files across the network. This uses the root protocol developed at CERN.
- `'shmem://h2[events]'`: opens the FITS file in a shared memory segment and moves to the EVENTS extension.
- `'mem://'`: creates a scratch output file in core computer memory. The resulting 'file' will disappear when the program exits, so this is mainly useful for testing purposes when one does not want a permanent copy of the output file.
- `'myfile.fits[3; Images(10)]'`: opens a copy of the image contained in the 10th row of the 'Images' column in the binary table in the 3th extension of the FITS file. The application just sees this single image as the primary array.

- `'myfile.fits[1:512:2, 1:512:2]'`: opens a section of the input image ranging from the 1st to the 512th pixel in X and Y, and selects every second pixel in both dimensions, resulting in a 256 x 256 pixel image in this case.

- `'myfile.fits[EVENTS][col Rad = sqrt(X**2 + Y**2)]'`: creates and opens a temporary file on the fly (in memory or on disk) that is identical to myfile.fits except that it will contain a new column in the EVENTS extension called 'Rad' whose value is computed using the indicated expression which is a function of the values in the X and Y columns.

- `'myfile.fits[EVENTS][PHA > 5]'`: creates and opens a temporary FITS files that is identical to 'myfile.fits' except that the EVENTS table will only contain the rows that have values of the PHA column greater than 5. In general, any arbitrary boolean expression using a C or Fortran-like syntax, which may combine AND and OR operators, may be used to select rows from a table.

- `'myfile.fits[EVENTS][bin (X,Y)=1,2048,4]'`: creates a temporary FITS primary array image which is computed on the fly by binning (i.e, computing the 2-dimensional histogram) of the values in the X and Y columns of the EVENTS extension. In this case the X and Y coordinates range from 1 to 2048 and the image pixel size is 4 units in both dimensions, so the resulting image is 512 x 512 pixels in size.

- The final example combines many of these feature into one complex expression (it is broken into several lines for clarity):

  ```
  'ftp://legacy.gsfc.nasa.gov/data/sample.fits.gz[EVENTS]
   [col phacorr = pha * 1.1 - 0.3][phacorr >= 5.0 && phacorr <= 14.0]
   [bin (X,Y)=32]'
  ```

  In this case, CFITSIO (1) copies and uncompresses the FITS file from the ftp site on the legacy machine, (2) moves to the 'EVENTS' extension, (3) calculates a new column called 'phacorr', (4) selects the rows in the table that have phacorr in the range 5 to 14, and finally (5) bins the remaining rows on the X and Y column coordinates, using a pixel size = 32 to create a 2D image. All this processing is completely transparent to the application program, which simply sees the final 2-D image in the primary array of the opened file.

The full extended CFITSIO FITS file name can contain several different components depending on the context. These components are described in the following sections:

```
When creating a new file:
   filetype://BaseFilename(templateName)

When opening an existing primary array or image HDU:
   filetype://BaseFilename(outName)[HDUlocation][ImageSection]

When opening an existing table HDU:
   filetype://BaseFilename(outName)[HDUlocation][colFilter][rowFilter][binSpec]
```

The filetype, BaseFilename, outName, HDUlocation, and ImageSection components, if present, must be given in that order, but the colFilter, rowFilter, and binSpec specifiers may follow in any order. Regardless of the order, however, the colFilter specifier, if present, will be processed first by CFITSIO, followed by the rowFilter specifier, and finally by the binSpec specifier.

8.2 Filetype

The type of file determines the medium on which the file is located (e.g., disk or network) and, hence, which internal device driver is used by CFITSIO to read and/or write the file. Currently supported types are

```
        file://  - file on local magnetic disk (default)
        ftp://   - a readonly file accessed with the anonymous FTP protocol.
                   It also supports  ftp://username:password@hostname/...
                   for accessing password-protected ftp sites.
        http://  - a readonly file accessed with the HTTP protocol.  It
                   supports username:password just like the ftp driver.
                   Proxy HTTP servers are supported using the http_proxy
                   environment variable (see following note).
      stream://  - special driver to read an input FITS file from the stdin
                   stream, and/or write an output FITS file to the stdout
   stream.  This driver is fragile and has limited
   functionality (see the following note).
      gsiftp://  - access files on a computational grid using the gridftp
                   protocol in the Globus toolkit (see following note).
        root://  - uses the CERN root protocol for writing as well as
                   reading files over the network.
        shmem:// - opens or creates a file which persists in the computer's
                   shared memory.
        mem://   - opens a temporary file in core memory.  The file
                   disappears when the program exits so this is mainly
                   useful for test purposes when a permanent output file
                   is not desired.
```

If the filetype is not specified, then type file:// is assumed. The double slashes '//' are optional and may be omitted in most cases.

8.2.1 Notes about HTTP proxy servers

A proxy HTTP server may be used by defining the address (URL) and port number of the proxy server with the http_proxy environment variable. For example

```
    setenv http_proxy http://heasarc.gsfc.nasa.gov:3128
```

will cause CFITSIO to use port 3128 on the heasarc proxy server whenever reading a FITS file with HTTP.

8.2.2 Notes about the stream filetype driver

The stream driver can be used to efficiently read a FITS file from the stdin file stream or write a FITS to the stdout file stream. However, because these input and output streams must be accessed sequentially, the FITS file reading or writing application must also read and write the file sequentially, at least within the tolerances described below.

CFITSIO supports 2 different methods for accessing FITS files on the stdin and stdout streams. The original method, which is invoked by specifying a dash character, "-", as the name of the file when opening or creating it, works by storing a complete copy of the entire FITS file in memory. In this case, when reading from stdin, CFITSIO will copy the entire stream into memory before doing any processing of the file. Similarly, when writing to stdout, CFITSIO will create a copy of the entire FITS file in memory, before finally flushing it out to the stdout stream when the FITS file is closed. Buffering the entire FITS file in this way allows the application to randomly access any part of the FITS file, in any order, but it also requires that the user have sufficient available memory (or virtual memory) to store the entire file, which may not be possible in the case of very large files.

The newer stream filetype provides a more memory-efficient method of accessing FITS files on the stdin or stdout streams. Instead of storing a copy of the entire FITS file in memory, CFITSIO only uses a set of internal buffer which by default can store 40 FITS blocks, or about 100K bytes of the FITS file. The application program must process the FITS file sequentially from beginning to end, within this 100K buffer. Generally speaking the application program must conform to the following restrictions:

- The program must finish reading or writing the header keywords before reading or writing any data in the HDU.
- The HDU can contain at most about 1400 header keywords. This is the maximum that can fit in the nominal 40 FITS block buffer. In principle, this limit could be increased by recompiling CFITSIO with a larger buffer limit, which is set by the NIOBUF parameter in fitsio2.h.
- The program must read or write the data in a sequential manner from the beginning to the end of the HDU. Note that CFITSIO's internal 100K buffer allows a little latitude in meeting this requirement.
- The program cannot move back to a previous HDU in the FITS file.
- Reading or writing of variable length array columns in binary tables is not supported on streams, because this requires moving back and forth between the fixed-length portion of the binary table and the following heap area where the arrays are actually stored.
- Reading or writing of tile-compressed images is not supported on streams, because the images are internally stored using variable length arrays.

8.2.3 Notes about the gsiftp filetype

DEPENDENCIES: Globus toolkit (2.4.3 or higher) (GT) should be installed. There are two different ways to install GT:

1) goto the globus toolkit web page www.globus.org and follow the download and compilation instructions;

2) goto the Virtual Data Toolkit web page http://vdt.cs.wisc.edu/ and follow the instructions (STRONGLY SUGGESTED);

Once a globus client has been installed in your system with a specific flavour it is possible to compile and install the CFITSIO libraries. Specific configuration flags must be used:

1) –with-gsiftp[[=PATH]] Enable Globus Toolkit gsiftp protocol support PATH=GLOBUS_LOCATION i.e. the location of your globus installation

2) –with-gsiftp-flavour[[=PATH] defines the specific Globus flavour ex. gcc32

Both the flags must be used and it is mandatory to set both the PATH and the flavour.

USAGE: To access files on a gridftp server it is necessary to use a gsiftp prefix:

example: gsiftp://remote_server_fqhn/directory/filename

The gridftp driver uses a local buffer on a temporary file the file is located in the /tmp directory. If you have special permissions on /tmp or you do not have a /tmp directory, it is possible to force another location setting the GSIFTP_TMPFILE environment variable (ex. export GSIFTP_TMPFILE=/your/location/yourtmpfile).

Grid FTP supports multi channel transfer. By default a single channel transmission is available. However, it is possible to modify this behavior setting the GSIFTP_STREAMS environment variable (ex. export GSIFTP_STREAMS=8).

8.2.4 Notes about the root filetype

The original rootd server can be obtained from: `ftp://root.cern.ch/root/rootd.tar.gz` but, for it to work correctly with CFITSIO one has to use a modified version which supports a command to return the length of the file. This modified version is available in rootd subdirectory in the CFITSIO ftp area at

```
ftp://legacy.gsfc.nasa.gov/software/fitsio/c/root/rootd.tar.gz.
```

This small server is started either by inetd when a client requests a connection to a rootd server or by hand (i.e. from the command line). The rootd server works with the ROOT TNetFile class. It allows remote access to ROOT database files in either read or write mode. By default TNetFile assumes port 432 (which requires rootd to be started as root). To run rootd via inetd add the following line to /etc/services:

```
rootd     432/tcp
```

and to /etc/inetd.conf, add the following line:

```
rootd stream tcp nowait root /user/rdm/root/bin/rootd rootd -i
```

Force inetd to reread its conf file with "kill -HUP ¡pid inetd¿". You can also start rootd by hand running directly under your private account (no root system privileges needed). For example to start rootd listening on port 5151 just type: `rootd -p 5151` Notice: no & is needed. Rootd will go into background by itself.

```
Rootd arguments:
  -i                  says we were started by inetd
  -p port#            specifies a different port to listen on
  -d level            level of debug info written to syslog
                      0 = no debug (default)
                      1 = minimum
                      2 = medium
                      3 = maximum
```

Rootd can also be configured for anonymous usage (like anonymous ftp). To setup rootd to accept anonymous logins do the following (while being logged in as root):

```
 - Add the following line to /etc/passwd:

   rootd:*:71:72:Anonymous rootd:/var/spool/rootd:/bin/false

   where you may modify the uid, gid (71, 72) and the home directory
   to suite your system.

 - Add the following line to /etc/group:

   rootd:*:72:rootd

   where the gid must match the gid in /etc/passwd.

 - Create the directories:

   mkdir /var/spool/rootd
   mkdir /var/spool/rootd/tmp
   chmod 777 /var/spool/rootd/tmp

   Where /var/spool/rootd must match the rootd home directory as
   specified in the rootd /etc/passwd entry.

 - To make writeable directories for anonymous do, for example:

   mkdir /var/spool/rootd/pub
   chown rootd:rootd /var/spool/rootd/pub
```

That's all. Several additional remarks: you can login to an anonymous server either with the names "anonymous" or "rootd". The password should be of type user@host.do.main. Only the @ is enforced for the time being. In anonymous mode the top of the file tree is set to the rootd home directory, therefore only files below the home directory can be accessed. Anonymous mode only works when the server is started via inetd.

8.2.5 Notes about the shmem filetype:

Shared memory files are currently supported on most Unix platforms, where the shared memory segments are managed by the operating system kernel and 'live' independently of processes. They are not deleted (by default) when the process which created them terminates, although they will disappear if the system is rebooted. Applications can create shared memory files in CFITSIO by calling:

```
fit_create_file(&fitsfileptr, "shmem://h2", &status);
```

where the root 'file' names are currently restricted to be 'h0', 'h1', 'h2', 'h3', etc., up to a maximum number defined by the the value of SHARED_MAXSEG (equal to 16 by default). This is a prototype implementation of the shared memory interface and a more robust interface, which will have fewer restrictions on the number of files and on their names, may be developed in the future.

When opening an already existing FITS file in shared memory one calls the usual CFITSIO routine:

```
fits_open_file(&fitsfileptr, "shmem://h7", mode, &status)
```

The file mode can be READWRITE or READONLY just as with disk files. More than one process can operate on READONLY mode files at the same time. CFITSIO supports proper file locking (both in READONLY and READWRITE modes), so calls to fits_open_file may be locked out until another other process closes the file.

When an application is finished accessing a FITS file in a shared memory segment, it may close it (and the file will remain in the system) with fits_close_file, or delete it with fits_delete_file. Physical deletion is postponed until the last process calls ffclos/ffdelt. fits_delete_file tries to obtain a READWRITE lock on the file to be deleted, thus it can be blocked if the object was not opened in READWRITE mode.

A shared memory management utility program called 'smem', is included with the CFITSIO distribution. It can be built by typing 'make smem'; then type 'smem -h' to get a list of valid options. Executing smem without any options causes it to list all the shared memory segments currently residing in the system and managed by the shared memory driver. To get a list of all the shared memory objects, run the system utility program 'ipcs [-a]'.

8.3 Base Filename

The base filename is the name of the file optionally including the director/subdirectory path, and in the case of 'ftp', 'http', and 'root' filetypes, the machine identifier. Examples:

```
myfile.fits
!data.fits
/data/myfile.fits
fits.gsfc.nasa.gov/ftp/sampledata/myfile.fits.gz
```

When creating a new output file on magnetic disk (of type file://) if the base filename begins with an exclamation point (!) then any existing file with that same basename will be deleted prior to creating the new FITS file. Otherwise if the file to be created already exists, then CFITSIO will return an error and will not overwrite the existing file. Note that the exclamation point, '!', is a special UNIX character, so if it is used on the command line rather than entered at a task prompt, it must be preceded by a backslash to force the UNIX shell to pass it verbatim to the application program.

If the output disk file name ends with the suffix '.gz', then CFITSIO will compress the file using the gzip compression algorithm before writing it to disk. This can reduce the amount of disk space used by the file. Note that this feature requires that the uncompressed file be constructed in memory before it is compressed and written to disk, so it can fail if there is insufficient available memory.

An input FITS file may be compressed with the gzip or Unix compress algorithms, in which case CFITSIO will uncompress the file on the fly into a temporary file (in memory or on disk). Compressed files may only be opened with read-only permission. When specifying the name of a compressed FITS file it is not necessary to append the file suffix (e.g., '.gz' or '.Z'). If CFITSIO cannot find the input file name without the suffix, then it will automatically search for a compressed file with the same root name. In the case of reading ftp and http type files, CFITSIO generally looks for a compressed version of the file first, before trying to open the uncompressed file. By default, CFITSIO copies (and uncompressed if necessary) the ftp or http FITS file into memory on the local machine before opening it. This will fail if the local machine does not have enough memory to hold the whole FITS file, so in this case, the output filename specifier (see the next section) can be used to further control how CFITSIO reads ftp and http files.

If the input file is an IRAF image file (*.imh file) then CFITSIO will automatically convert it on the fly into a virtual FITS image before it is opened by the application program. IRAF images can only be opened with READONLY file access.

Similarly, if the input file is a raw binary data array, then CFITSIO will convert it on the fly into a virtual FITS image with the basic set of required header keywords before it is opened by the application program (with READONLY access). In this case the data type and dimensions of the image must be specified in square brackets following the filename (e.g. rawfile.dat[ib512,512]). The first character (case insensitive) defines the datatype of the array:

```
b         8-bit unsigned byte
i        16-bit signed integer
u        16-bit unsigned integer
j        32-bit signed integer
r or f   32-bit floating point
d        64-bit floating point
```

An optional second character specifies the byte order of the array values: b or B indicates big endian (as in FITS files and the native format of SUN UNIX workstations and Mac PCs) and l or L indicates little endian (native format of DEC OSF workstations and IBM PCs). If this character is omitted then the array is assumed to have the native byte order of the local machine. These datatype characters are then followed by a series of one or more integer values separated by commas which define the size of each dimension of the raw array. Arrays with up to 5 dimensions are currently supported. Finally, a byte offset to the position of the first pixel in the data file may be specified by separating it with a ':' from the last dimension value. If omitted, it is assumed that the offset = 0. This parameter may be used to skip over any header information in the file that precedes the binary data. Further examples:

```
raw.dat[b10000]           1-dimensional 10000 pixel byte array
raw.dat[rb400,400,12]     3-dimensional floating point big-endian array
img.fits[ib512,512:2880]  reads the 512 x 512 short integer array in
                          a FITS file, skipping over the 2880 byte header
```

One special case of input file is where the filename = '-' (a dash or minus sign) or 'stdin' or 'stdout', which signifies that the input file is to be read from the stdin stream, or written to the stdout stream if a new output file is being created. In the case of reading from stdin, CFITSIO first copies the whole stream into a temporary FITS file (in memory or on disk), and subsequent reading of the FITS file occurs in this copy. When writing to stdout, CFITSIO first constructs the whole file in memory (since random access is required), then flushes it out to the stdout stream when the file is closed. In addition, if the output filename = '-.gz' or 'stdout.gz' then it will be gzip compressed before being written to stdout.

This ability to read and write on the stdin and stdout steams allows FITS files to be piped between tasks in memory rather than having to create temporary intermediate FITS files on disk. For example if task1 creates an output FITS file, and task2 reads an input FITS file, the FITS file may be piped between the 2 tasks by specifying

```
task1 - | task2 -
```

where the vertical bar is the Unix piping symbol. This assumes that the 2 tasks read the name of the FITS file off of the command line.

8.4 Output File Name when Opening an Existing File

An optional output filename may be specified in parentheses immediately following the base file name to be opened. This is mainly useful in those cases where CFITSIO creates a temporary copy of the input FITS file before it is opened and passed to the application program. This happens by default when opening a network FTP or HTTP-type file, when reading a compressed FITS file on a local disk, when reading from the stdin stream, or when a column filter, row filter, or binning specifier is included as part of the input file specification. By default this temporary file is created in memory. If there is not enough memory to create the file copy, then CFITSIO will exit with an error. In these cases one can force a permanent file to be created on disk, instead of a temporary file in memory, by supplying the name in parentheses immediately following the base file name. The output filename can include the '!' clobber flag.

Thus, if the input filename to CFITSIO is: `file1.fits.gz(file2.fits)` then CFITSIO will uncompress 'file1.fits.gz' into the local disk file 'file2.fits' before opening it. CFITSIO does not automatically delete the output file, so it will still exist after the application program exits.

In some cases, several different temporary FITS files will be created in sequence, for instance, if one opens a remote file using FTP, then filters rows in a binary table extension, then create an image by binning a pair of columns. In this case, the remote file will be copied to a temporary local file, then a second temporary file will be created containing the filtered rows of the table, and finally a third temporary file containing the binned image will be created. In cases like this where multiple files are created, the outfile specifier will be interpreted the name of the final file as described below, in descending priority:

- as the name of the final image file if an image within a single binary table cell is opened or if an image is created by binning a table column.
- as the name of the file containing the filtered table if a column filter and/or a row filter are specified.
- as the name of the local copy of the remote FTP or HTTP file.
- as the name of the uncompressed version of the FITS file, if a compressed FITS file on local disk has been opened.
- otherwise, the output filename is ignored.

The output file specifier is useful when reading FTP or HTTP-type FITS files since it can be used to create a local disk copy of the file that can be reused in the future. If the output file name = '*' then a local file with the same name as the network file will be created. Note that CFITSIO will behave differently depending on whether the remote file is compressed or not as shown by the following examples:

- 'ftp://remote.machine/tmp/myfile.fits.gz(*)' - the remote compressed file is copied to the local compressed file 'myfile.fits.gz', which is then uncompressed in local memory before being opened and passed to the application program.
- 'ftp://remote.machine/tmp/myfile.fits.gz(myfile.fits)' - the remote compressed file is copied and uncompressed into the local file 'myfile.fits'. This example requires less local memory than the previous example since the file is uncompressed on disk instead of in memory.
- 'ftp://remote.machine/tmp/myfile.fits(myfile.fits.gz)' - this will usually produce an error since CFITSIO itself cannot compress files.

The exact behavior of CFITSIO in the latter case depends on the type of ftp server running on the remote machine and how it is configured. In some cases, if the file 'myfile.fits.gz' exists on the remote machine, then the server will copy it to the local machine. In other cases the ftp server will automatically create and transmit a compressed version of the file if only the uncompressed version exists. This can get rather confusing, so users should use a certain amount of caution when using the output file specifier with FTP or HTTP file types, to make sure they get the behavior that they expect.

8.5 Template File Name when Creating a New File

When a new FITS file is created with a call to fits_create_file, the name of a template file may be supplied in parentheses immediately following the name of the new file to be created. This template is used to define the structure of one or more HDUs in the new file. The template file may be another FITS file, in which case the newly created file will have exactly the same keywords in each HDU as in the template FITS file, but all the data units will be filled with zeros. The template file may also be an ASCII text file, where each line (in general) describes one FITS keyword record. The format of the ASCII template file is described below.

8.6 Image Tile-Compression Specification

When specifying the name of the output FITS file to be created, the user can indicate that images should be written in tile-compressed format (see section 5.5, "Primary Array or IMAGE Extension I/O Routines") by enclosing the compression parameters in square brackets following the root disk file name. Here are some examples of the syntax for specifying tile-compressed output images:

```
myfile.fit[compress]    - use Rice algorithm and default tile size

myfile.fit[compress GZIP] - use the specified compression algorithm;
myfile.fit[compress Rice]     only the first letter of the algorithm
myfile.fit[compress PLIO]     name is required.

myfile.fit[compress Rice 100,100]   - use 100 x 100 pixel tile size
myfile.fit[compress Rice 100,100;2] - as above, and use noisebits = 2
```

8.7 HDU Location Specification

The optional HDU location specifier defines which HDU (Header-Data Unit, also known as an 'extension') within the FITS file to initially open. It must immediately follow the base file name (or the output file name if present). If it is not specified then the first HDU (the primary array) is opened. The HDU location specifier is required if the colFilter, rowFilter, or binSpec specifiers are present, because the primary array is not a valid HDU for these operations. The HDU may be specified either by absolute position number, starting with 0 for the primary array, or by reference to the HDU name, and optionally, the version number and the HDU type of the desired extension. The location of an image within a single cell of a binary table may also be specified, as described below.

The absolute position of the extension is specified either by enclosed the number in square brackets (e.g., '[1]' = the first extension following the primary array) or by preceded the number with a plus sign ('+1'). To specify the HDU by name, give the name of the desired HDU (the value of the EXTNAME or HDUNAME keyword) and optionally the extension version number (value of the EXTVER keyword) and the extension type (value of the XTENSION keyword: IMAGE, ASCII or TABLE, or BINTABLE), separated by commas and all enclosed in square brackets. If the value of EXTVER and XTENSION are not specified, then the first extension with the correct value of EXTNAME is opened. The extension name and type are not case sensitive, and the extension type may be abbreviated to a single letter (e.g., I = IMAGE extension or primary array, A or T = ASCII table extension, and B = binary table BINTABLE extension). If the HDU location specifier is equal to '[PRIMARY]' or '[P]', then the primary array (the first HDU) will be opened.

FITS images are most commonly stored in the primary array or an image extension, but images can also be stored as a vector in a single cell of a binary table (i.e. each row of the vector column contains a different image). Such an image can be opened with CFITSIO by specifying the desired column name and the row number after the binary table HDU specifier as shown in the following examples. The column name is separated from the HDU specifier by a semicolon and the row number is enclosed in parentheses. In this case CFITSIO copies the image from the table cell into a temporary primary array before it is opened. The application program then just sees the image in the primary array, without any extensions. The particular row to be opened may be specified either by giving an absolute integer row number (starting with 1 for the first row), or by specifying a boolean expression that evaluates to TRUE for the desired row. The first row that satisfies the expression will be used. The row selection expression has the same syntax as described in the Row Filter Specifier section, below.

Examples:

```
myfile.fits[3] - open the 3rd HDU following the primary array
myfile.fits+3  - same as above, but using the FTOOLS-style notation
myfile.fits[EVENTS] - open the extension that has EXTNAME = 'EVENTS'
myfile.fits[EVENTS, 2]  - same as above, but also requires EXTVER = 2
myfile.fits[events,2,b] - same, but also requires XTENSION = 'BINTABLE'
myfile.fits[3; images(17)] - opens the image in row 17 of the 'images'
                             column in the 3rd extension of the file.
myfile.fits[3; images(exposure > 100)] - as above, but opens the image
                in the first row that has an 'exposure' column value
                greater than 100.
```

8.8 Image Section

A virtual file containing a rectangular subsection of an image can be extracted and opened by specifying the range of pixels (start:end) along each axis to be extracted from the original image. One can also specify an optional pixel increment (start:end:step) for each axis of the input image. A pixel step = 1 will be assumed if it is not specified. If the start pixel is larger then the end pixel, then the image will be flipped (producing a mirror image) along that dimension. An asterisk, '*', may be used to specify the entire range of an axis, and '-*' will flip the entire axis. The input image can be in the primary array, in an image extension, or contained in a vector cell of a binary table. In the later 2 cases the extension name or number must be specified before the image section specifier.

Examples:

```
myfile.fits[1:512:2, 2:512:2] -  open a 256x256 pixel image
```

```
             consisting of the odd numbered columns (1st axis) and
             the even numbered rows (2nd axis) of the image in the
             primary array of the file.

 myfile.fits[*, 512:256] - open an image consisting of all the columns
             in the input image, but only rows 256 through 512.
             The image will be flipped along the 2nd axis since
             the starting pixel is greater than the ending pixel.

 myfile.fits[*:2, 512:256:2] - same as above but keeping only
             every other row and column in the input image.

 myfile.fits[-*, *] - copy the entire image, flipping it along
             the first axis.

 myfile.fits[3][1:256,1:256] - opens a subsection of the image that
             is in the 3rd extension of the file.

 myfile.fits[4; images(12)][1:10,1:10] - open an image consisting
         of the first 10 pixels in both dimensions. The original
         image resides in the 12th row of the 'images' vector
         column in the table in the 4th extension of the file.
```

When CFITSIO opens an image section it first creates a temporary file containing the image section plus a copy of any other HDUs in the file. This temporary file is then opened by the application program, so it is not possible to write to or modify the input file when specifying an image section. Note that CFITSIO automatically updates the world coordinate system keywords in the header of the image section, if they exist, so that the coordinate associated with each pixel in the image section will be computed correctly.

8.9 Image Transform Filters

CFITSIO can apply a user-specified mathematical function to the value of every pixel in a FITS image, thus creating a new virtual image in computer memory that is then opened and read by the application program. The original FITS image is not modified by this process.

The image transformation specifier is appended to the input FITS file name and is enclosed in square brackets. It begins with the letters 'PIX' to distinguish it from other types of FITS file filters that are recognized by CFITSIO. The image transforming function may use any of the mathematical operators listed in the following 'Row Filtering Specification' section of this document. Some examples of image transform filters are:

```
 [pix X * 2.0]               - multiply each pixel by 2.0
 [pix sqrt(X)]               - take the square root of each pixel
 [pix X + #ZEROPT            - add the value of the ZEROPT keyword
```

```
[pix X>0 ? log10(X) : -99.] - if the pixel value is greater
                              than 0, compute the base 10 log,
                              else set the pixel = -99.
```

Use the letter 'X' in the expression to represent the current pixel value in the image. The expression is evaluated independently for each pixel in the image and may be a function of 1) the original pixel value, 2) the value of other pixels in the image at a given relative offset from the position of the pixel that is being evaluated, and 3) the value of any header keywords. Header keyword values are represented by the name of the keyword preceded by the '#' sign.

To access the the value of adjacent pixels in the image, specify the (1-D) offset from the current pixel in curly brackets. For example

```
[pix  (x{-1} + x + x{+1}) / 3]
```

will replace each pixel value with the running mean of the values of that pixel and it's 2 neighboring pixels. Note that in this notation the image is treated as a 1-D array, where each row of the image (or higher dimensional cube) is appended one after another in one long array of pixels. It is possible to refer to pixels in the rows above or below the current pixel by using the value of the NAXIS1 header keyword. For example

```
[pix (x{-#NAXIS1} + x + x{#NAXIS1}) / 3]
```

will compute the mean of each image pixel and the pixels immediately above and below it in the adjacent rows of the image. The following more complex example creates a smoothed virtual image where each pixel is a 3 x 3 boxcar average of the input image pixels:

```
 [pix (X + X{-1} + X{+1}
     + X{-#NAXIS1} + X{-#NAXIS1 - 1} + X{-#NAXIS1 + 1}
     + X{#NAXIS1} + X{#NAXIS1 - 1} + X{#NAXIS1 + 1}) / 9.]
```

If the pixel offset extends beyond the first or last pixel in the image, the function will evaluate to undefined, or NULL.

For complex or commonly used image filtering operations, one can write the expression into an external text file and then import it into the filter using the syntax '[pix @filename.txt]'. The mathematical expression can extend over multiple lines of text in the file. Any lines in the external text file that begin with 2 slash characters ('//') will be ignored and may be used to add comments into the file.

By default, the datatype of the resulting image will be the same as the original image, but one may force a different datatype by appended a code letter to the 'pix' keyword:

```
pixb  -  8-bit byte    image with BITPIX =   8
pixi  - 16-bit integer image with BITPIX =  16
pixj  - 32-bit integer image with BITPIX =  32
pixr  - 32-bit float   image with BITPIX = -32
pixd  - 64-bit float   image with BITPIX = -64
```

Also by default, any other HDUs in the input file will be copied without change to the output virtual FITS file, but one may discard the other HDUs by adding the number '1' to the 'pix' keyword (and following any optional datatype code letter). For example:

```
myfile.fits[3][pixr1  sqrt(X)]
```

will create a virtual FITS file containing only a primary array image with 32-bit floating point pixels that have a value equal to the square root of the pixels in the image that is in the 3rd extension of the 'myfile.fits' file.

8.10 Column and Keyword Filtering Specification

The optional column/keyword filtering specifier is used to modify the column structure and/or the header keywords in the HDU that was selected with the previous HDU location specifier. This filtering specifier must be enclosed in square brackets and can be distinguished from a general row filter specifier (described below) by the fact that it begins with the string 'col ' and is not immediately followed by an equals sign. The original file is not changed by this filtering operation, and instead the modifications are made on a copy of the input FITS file (usually in memory), which also contains a copy of all the other HDUs in the file. This temporary file is passed to the application program and will persist only until the file is closed or until the program exits, unless the outfile specifier (see above) is also supplied.

The column/keyword filter can be used to perform the following operations. More than one operation may be specified by separating them with semi-colons.

- Copy only a specified list of columns columns to the filtered input file. The list of column name should be separated by semi-colons. Wild card characters may be used in the column names to match multiple columns. If the expression contains both a list of columns to be included and columns to be deleted, then all the columns in the original table except the explicitly deleted columns will appear in the filtered table (i.e., there is no need to explicitly list the columns to be included if any columns are being deleted).
- Delete a column or keyword by listing the name preceded by a minus sign or an exclamation mark (!), e.g., '-TIME' will delete the TIME column if it exists, otherwise the TIME keyword. An error is returned if neither a column nor keyword with this name exists. Note that the exclamation point, '!', is a special UNIX character, so if it is used on the command line rather than entered at a task prompt, it must be preceded by a backslash to force the UNIX shell to ignore it.
- Rename an existing column or keyword with the syntax 'NewName == OldName'. An error is returned if neither a column nor keyword with this name exists.
- Append a new column or keyword to the table. To create a column, give the new name, optionally followed by the datatype in parentheses, followed by a single equals sign and an expression to be used to compute the value (e.g., 'newcol(1J) = 0' will create a new 32-bit integer column called 'newcol' filled with zeros). The datatype is specified using the same

syntax that is allowed for the value of the FITS TFORMn keyword (e.g., 'I', 'J', 'E', 'D', etc. for binary tables, and 'I8', F12.3', 'E20.12', etc. for ASCII tables). If the datatype is not specified then an appropriate datatype will be chosen depending on the form of the expression (may be a character string, logical, bit, long integer, or double column). An appropriate vector count (in the case of binary tables) will also be added if not explicitly specified.

When creating a new keyword, the keyword name must be preceded by a pound sign '#', and the expression must evaluate to a scalar (i.e., cannot have a column name in the expression). The comment string for the keyword may be specified in parentheses immediately following the keyword name (instead of supplying a datatype as in the case of creating a new column). If the keyword name ends with a pound sign '#', then cfitsio will substitute the number of the most recently referenced column for the # character . This is especially useful when writing a column-related keyword like TUNITn for a newly created column, as shown in the following examples.

- Recompute (overwrite) the values in an existing column or keyword by giving the name followed by an equals sign and an arithmetic expression.

The expression that is used when appending or recomputing columns or keywords can be arbitrarily complex and may be a function of other header keyword values and other columns (in the same row). The full syntax and available functions for the expression are described below in the row filter specification section.

If the expression contains both a list of columns to be included and columns to be deleted, then all the columns in the original table except the explicitly deleted columns will appear in the filtered table. If no columns to be deleted are specified, then only the columns that are explicitly listed will be included in the filtered output table. To include all the columns, add the '*' wildcard specifier at the end of the list, as shown in the examples.

For complex or commonly used operations, one can also place the operations into an external text file and import it into the column filter using the syntax '[col @filename.txt]'. The operations can extend over multiple lines of the file, but multiple operations must still be separated by semicolons. Any lines in the external text file that begin with 2 slash characters ('//') will be ignored and may be used to add comments into the file.

Examples:

```
[col Time;rate]              - only the Time and rate columns will
                               appear in the filtered input file.

[col Time;*raw]              - include the Time column and any other
                               columns whose name ends with 'raw'.

[col -TIME; Good == STATUS]  - deletes the TIME column and
                               renames the status column to 'Good'

[col PI=PHA * 1.1 + 0.2; #TUNIT#(column units) = 'counts';*]
                             - creates new PI column from PHA values
                               and also writes the TUNITn keyword
```

```
                                    for the new column.  The final '*'
                                    expression means preserve all the
                                    columns in the input table in the
                                    virtual output table;  without the '*'
                                    the output table would only contain
                                    the single 'PI' column.

   [col rate = rate/exposure; TUNIT#(&) = 'counts/s';*]
                                  - recomputes the rate column by dividing
                                    it by the EXPOSURE keyword value. This
                                    also modifies the value of the TUNITn
                                    keyword for this column. The use of the
                                    '&' character for the keyword comment
                                    string means preserve the existing
                                    comment string for that keyword. The
                                    final '*' preserves all the columns
                                    in the input table in the virtual
                                    output table.
```

8.11 Row Filtering Specification

When entering the name of a FITS table that is to be opened by a program, an optional row filter may be specified to select a subset of the rows in the table. A temporary new FITS file is created on the fly which contains only those rows for which the row filter expression evaluates to true. (The primary array and any other extensions in the input file are also copied to the temporary file). The original FITS file is closed and the new virtual file is opened by the application program. The row filter expression is enclosed in square brackets following the file name and extension name (e.g., 'file.fits[events][GRADE==50]' selects only those rows where the GRADE column value equals 50). When dealing with tables where each row has an associated time and/or 2D spatial position, the row filter expression can also be used to select rows based on the times in a Good Time Intervals (GTI) extension, or on spatial position as given in a SAO-style region file.

8.11.1 General Syntax

The row filtering expression can be an arbitrarily complex series of operations performed on constants, keyword values, and column data taken from the specified FITS TABLE extension. The expression must evaluate to a boolean value for each row of the table, where a value of FALSE means that the row will be excluded.

For complex or commonly used filters, one can place the expression into a text file and import it into the row filter using the syntax '[@filename.txt]'. The expression can be arbitrarily complex and extend over multiple lines of the file. Any lines in the external text file that begin with 2 slash characters ('//') will be ignored and may be used to add comments into the file.

Keyword and column data are referenced by name. Any string of characters not surrounded by quotes (ie, a constant string) or followed by an open parentheses (ie, a function name) will be initially interpreted as a column name and its contents for the current row inserted into the expression. If no such column exists, a keyword of that name will be searched for and its value used, if found. To force the name to be interpreted as a keyword (in case there is both a column and keyword with the same name), precede the keyword name with a single pound sign, '#', as in '#NAXIS2'. Due to the generalities of FITS column and keyword names, if the column or keyword name contains a space or a character which might appear as an arithmetic term then enclose the name in '$' characters as in $MAX PHA$ or #$MAX-PHA$. Names are case insensitive.

To access a table entry in a row other than the current one, follow the column's name with a row offset within curly braces. For example, 'PHA{-3}' will evaluate to the value of column PHA, 3 rows above the row currently being processed. One cannot specify an absolute row number, only a relative offset. Rows that fall outside the table will be treated as undefined, or NULLs.

Boolean operators can be used in the expression in either their Fortran or C forms. The following boolean operators are available:

```
"equal"         .eq. .EQ. ==  "not equal"          .ne.  .NE.  !=
"less than"     .lt. .LT. <   "less than/equal"    .le.  .LE.  <= =<
"greater than"  .gt. .GT. >   "greater than/equal" .ge.  .GE.  >= =>
"or"            .or. .OR. ||  "and"                .and. .AND. &&
"negation"     .not. .NOT. !  "approx. equal(1e-7)"  ~
```

Note that the exclamation point, '!', is a special UNIX character, so if it is used on the command line rather than entered at a task prompt, it must be preceded by a backslash to force the UNIX shell to ignore it.

The expression may also include arithmetic operators and functions. Trigonometric functions use radians, not degrees. The following arithmetic operators and functions can be used in the expression (function names are case insensitive). A null value will be returned in case of illegal operations such as divide by zero, sqrt(negative) log(negative), log10(negative), arccos(.gt. 1), arcsin(.gt. 1).

```
"addition"           +          "subtraction"          -
"multiplication"     *          "division"             /
"negation"           -          "exponentiation"       **   ^
"absolute value"     abs(x)     "cosine"               cos(x)
"sine"               sin(x)     "tangent"              tan(x)
"arc cosine"         arccos(x)  "arc sine"             arcsin(x)
"arc tangent"        arctan(x)  "arc tangent"          arctan2(y,x)
"hyperbolic cos"     cosh(x)    "hyperbolic sin"       sinh(x)
"hyperbolic tan"     tanh(x)    "round to nearest int" round(x)
"round down to int"  floor(x)   "round up to int"      ceil(x)
"exponential"        exp(x)     "square root"          sqrt(x)
"natural log"        log(x)     "common log"           log10(x)
"modulus"            x % y      "random # [0.0,1.0)"   random()
"random Gaussian"    randomn()  "random Poisson"       randomp(x)
"minimum"            min(x,y)   "maximum"              max(x,y)
"cumulative sum"     accum(x)  "sequential difference" seqdiff(x)
"if-then-else"       b?x:y
```

```
"angular separation"  angsep(ra1,dec1,ra2,de2) (all in degrees)
"substring"      strmid(s,p,n) "string search"        strstr(s,r)
```

Three different random number functions are provided: random(), with no arguments, produces a uniform random deviate between 0 and 1; randomn(), also with no arguments, produces a normal (Gaussian) random deviate with zero mean and unit standard deviation; randomp(x) produces a Poisson random deviate whose expected number of counts is X. X may be any positive real number of expected counts, including fractional values, but the return value is an integer.

When the random functions are used in a vector expression, by default the same random value will be used when evaluating each element of the vector. If different random numbers are desired, then the name of a vector column should be supplied as the single argument to the random function (e.g., "flux + 0.1 * random(flux)", where "flux' is the name of a vector column). This will create a vector of random numbers that will be used in sequence when evaluating each element of the vector expression.

An alternate syntax for the min and max functions has only a single argument which should be a vector value (see below). The result will be the minimum/maximum element contained within the vector.

The accum(x) function forms the cumulative sum of x, element by element. Vector columns are supported simply by performing the summation process through all the values. Null values are treated as 0. The seqdiff(x) function forms the sequential difference of x, element by element. The first value of seqdiff is the first value of x. A single null value in x causes a pair of nulls in the output. The seqdiff and accum functions are functional inverses, i.e., seqdiff(accum(x)) == x as long as no null values are present.

In the if-then-else expression, "b?x:y", b is an explicit boolean value or expression. There is no automatic type conversion from numeric to boolean values, so one needs to use "iVal!=0" instead of merely "iVal" as the boolean argument. x and y can be any scalar data type (including string).

The angsep function computes the angular separation in degrees between 2 celestial positions, where the first 2 parameters give the RA-like and Dec-like coordinates (in decimal degrees) of the first position, and the 3rd and 4th parameters give the coordinates of the second position.

The substring function strmid(S,P,N) extracts a substring from S, starting at string position P, with a substring length N. The first character position in S is labeled as 1. If P is 0, or refers to a position beyond the end of S, then the extracted substring will be NULL. S, P, and N may be functions of other columns.

The string search function strstr(S,R) searches for the first occurrence of the substring R in S. The result is an integer, indicating the character position of the first match (where 1 is the first character position of S). If no match is found, then strstr() returns a NULL value.

The following type casting operators are available, where the enclosing parentheses are required and taken from the C language usage. Also, the integer to real casts values to double precision:

```
"real to integer"    (int) x     (INT) x
"integer to real"    (float) i   (FLOAT) i
```

In addition, several constants are built in for use in numerical expressions:

```
#pi              3.1415...      #e             2.7182...
#deg             #pi/180        #row           current row number
#null         undefined value   #snull         undefined string
```

A string constant must be enclosed in quotes as in 'Crab'. The "null" constants are useful for conditionally setting table values to a NULL, or undefined, value (eg., "col1==-99 ? #NULL : col1").

There is also a function for testing if two values are close to each other, i.e., if they are "near" each other to within a user specified tolerance. The arguments, value_1 and value_2 can be integer or real and represent the two values who's proximity is being tested to be within the specified tolerance, also an integer or real:

```
near(value_1, value_2, tolerance)
```

When a NULL, or undefined, value is encountered in the FITS table, the expression will evaluate to NULL unless the undefined value is not actually required for evaluation, e.g. "TRUE .or. NULL" evaluates to TRUE. The following two functions allow some NULL detection and handling:

```
"a null value?"              ISNULL(x)
"define a value for null"    DEFNULL(x,y)
```

The former returns a boolean value of TRUE if the argument x is NULL. The later "defines" a value to be substituted for NULL values; it returns the value of x if x is not NULL, otherwise it returns the value of y.

8.11.2 Bit Masks

Bit masks can be used to select out rows from bit columns (TFORMn = #X) in FITS files. To represent the mask, binary, octal, and hex formats are allowed:

```
binary:   b0110xx1010000101xxxx0001
octal:    o720x1 -> (b111010000xxx001)
hex:      h0FxD  -> (b00001111xxxx1101)
```

In all the representations, an x or X is allowed in the mask as a wild card. Note that the x represents a different number of wild card bits in each representation. All representations are case insensitive.

To construct the boolean expression using the mask as the boolean equal operator described above on a bit table column. For example, if you had a 7 bit column named flags in a FITS table and wanted all rows having the bit pattern 0010011, the selection expression would be:

```
                  flags == b0010011
or
                  flags .eq. b10011
```

It is also possible to test if a range of bits is less than, less than equal, greater than and greater than equal to a particular boolean value:

```
flags <= bxxx010xx
flags .gt. bxxx100xx
flags .le. b1xxxxxxx
```

Notice the use of the x bit value to limit the range of bits being compared.

It is not necessary to specify the leading (most significant) zero (0) bits in the mask, as shown in the second expression above.

Bit wise AND, OR and NOT operations are also possible on two or more bit fields using the '&'(AND), '|'(OR), and the '!'(NOT) operators. All of these operators result in a bit field which can then be used with the equal operator. For example:

```
(!flags) == b1101100
(flags & b1000001) == bx000001
```

Bit fields can be appended as well using the '+' operator. Strings can be concatenated this way, too.

8.11.3 Vector Columns

Vector columns can also be used in building the expression. No special syntax is required if one wants to operate on all elements of the vector. Simply use the column name as for a scalar column. Vector columns can be freely intermixed with scalar columns or constants in virtually all expressions. The result will be of the same dimension as the vector. Two vectors in an expression, though, need to have the same number of elements and have the same dimensions. The only places a vector column cannot be used (for now, anyway) are the SAO region functions and the NEAR boolean function.

Arithmetic and logical operations are all performed on an element by element basis. Comparing two vector columns, eg "COL1 == COL2", thus results in another vector of boolean values indicating which elements of the two vectors are equal.

Eight functions are available that operate on a vector and return a scalar result:

```
"minimum"      MIN(V)          "maximum"                MAX(V)
"average"      AVERAGE(V)      "median"                 MEDIAN(V)
"summation"    SUM(V)          "standard deviation"     STDDEV(V)
"# of values"  NELEM(V)        "# of non-null values"   NVALID(V)
```

where V represents the name of a vector column or a manually constructed vector using curly brackets as described below. The first 6 of these functions ignore any null values in the vector when computing the result. The STDDEV() function computes the sample standard deviation, i.e. it is proportional to 1/SQRT(N-1) instead of 1/SQRT(N), where N is NVALID(V).

The SUM function literally sums all the elements in x, returning a scalar value. If x is a boolean vector, SUM returns the number of TRUE elements. The NELEM function returns the number of elements in vector x whereas NVALID return the number of non-null elements in the vector. (NELEM also operates on bit and string columns, returning their column widths.) As an example, to test whether all elements of two vectors satisfy a given logical comparison, one can use the expression

```
SUM( COL1 > COL2 ) == NELEM( COL1 )
```

which will return TRUE if all elements of COL1 are greater than their corresponding elements in COL2.

To specify a single element of a vector, give the column name followed by a comma-separated list of coordinates enclosed in square brackets. For example, if a vector column named PHAS exists in the table as a one dimensional, 256 component list of numbers from which you wanted to select the 57th component for use in the expression, then PHAS[57] would do the trick. Higher dimensional arrays of data may appear in a column. But in order to interpret them, the TDIMn keyword must appear in the header. Assuming that a (4,4,4,4) array is packed into each row of a column named ARRAY4D, the (1,2,3,4) component element of each row is accessed by ARRAY4D[1,2,3,4]. Arrays up to dimension 5 are currently supported. Each vector index can itself be an expression, although it must evaluate to an integer value within the bounds of the vector. Vector columns which contain spaces or arithmetic operators must have their names enclosed in "$" characters as with $ARRAY-4D$[1,2,3,4].

A more C-like syntax for specifying vector indices is also available. The element used in the preceding example alternatively could be specified with the syntax ARRAY4D[4][3][2][1]. Note the reverse order of indices (as in C), as well as the fact that the values are still ones-based (as in Fortran – adopted to avoid ambiguity for 1D vectors). With this syntax, one does not need to specify all of the indices. To extract a 3D slice of this 4D array, use ARRAY4D[4].

Variable-length vector columns are not supported.

Vectors can be manually constructed within the expression using a comma-separated list of elements surrounded by curly braces ('{}'). For example, '{1,3,6,1}' is a 4-element vector containing the values 1, 3, 6, and 1. The vector can contain only boolean, integer, and real values (or expressions). The elements will be promoted to the highest datatype present. Any elements which are themselves vectors, will be expanded out with each of its elements becoming an element in the constructed vector.

8.11.4 Good Time Interval Filtering

A common filtering method involves selecting rows which have a time value which lies within what is called a Good Time Interval or GTI. The time intervals are defined in a separate FITS table extension which contains 2 columns giving the start and stop time of each good interval. The filtering operation accepts only those rows of the input table which have an associated time which falls within one of the time intervals defined in the GTI extension. A high level function, gtifilter(a,b,c,d), is available which evaluates each row of the input table and returns TRUE or FALSE depending whether the row is inside or outside the good time interval. The syntax is

```
gtifilter( [ "gtifile" [, expr [, "STARTCOL", "STOPCOL" ] ] ] )
```

where each "[]" demarks optional parameters. Note that the quotes around the gtifile and START/STOP column are required. Either single or double quotes may be used. In cases where this expression is entered on the Unix command line, enclose the entire expression in double quotes, and then use single quotes within the expression to enclose the 'gtifile' and other terms. It is also usually possible to do the reverse, and enclose the whole expression in single quotes and then use double quotes within the expression. The gtifile, if specified, can be blank ("") which will mean to use the first extension with the name "*GTI*" in the current file, a plain extension specifier (eg, "+2", "[2]", or "[STDGTI]") which will be used to select an extension in the current file, or a regular filename with or without an extension specifier which in the latter case will mean to use the first extension with an extension name "*GTI*". Expr can be any arithmetic expression, including simply the time column name. A vector time expression will produce a vector boolean result. STARTCOL and STOPCOL are the names of the START/STOP columns in the GTI extension. If one of them is specified, they both must be.

In its simplest form, no parameters need to be provided – default values will be used. The expression "gtifilter()" is equivalent to

```
gtifilter( "", TIME, "*START*", "*STOP*" )
```

This will search the current file for a GTI extension, filter the TIME column in the current table, using START/STOP times taken from columns in the GTI extension with names containing the strings "START" and "STOP". The wildcards ('*') allow slight variations in naming conventions such as "TSTART" or "STARTTIME". The same default values apply for unspecified parameters when the first one or two parameters are specified. The function automatically searches for TIMEZERO/I/F keywords in the current and GTI extensions, applying a relative time offset, if necessary.

8.11.5 Spatial Region Filtering

Another common filtering method selects rows based on whether the spatial position associated with each row is located within a given 2-dimensional region. The syntax for this high-level filter is

```
regfilter( "regfilename" [ , Xexpr, Yexpr [ , "wcs cols" ] ] )
```

where each "[]" demarks optional parameters. The region file name is required and must be enclosed in quotes. The remaining parameters are optional. There are 2 supported formats for the region file: ASCII file or FITS binary table. The region file contains a list of one or more geometric shapes (circle, ellipse, box, etc.) which defines a region on the celestial sphere or an area within a particular 2D image. The region file is typically generated using an image display program such as fv/POW (distribute by the HEASARC), or ds9 (distributed by the Smithsonian Astrophysical Observatory). Users should refer to the documentation provided with these programs for more details on the syntax used in the region files. The FITS region file format is defined in a document available from the FITS Support Office at http://fits.gsfc.nasa.gov/ registry/ region.html

In its simplest form, (e.g., regfilter("region.reg")) the coordinates in the default 'X' and 'Y' columns will be used to determine if each row is inside or outside the area specified in the region file. Alternate position column names, or expressions, may be entered if needed, as in

```
regfilter("region.reg", XPOS, YPOS)
```

Region filtering can be applied most unambiguously if the positions in the region file and in the table to be filtered are both give in terms of absolute celestial coordinate units. In this case the locations and sizes of the geometric shapes in the region file are specified in angular units on the sky (e.g., positions given in R.A. and Dec. and sizes in arcseconds or arcminutes). Similarly, each row of the filtered table will have a celestial coordinate associated with it. This association is usually implemented using a set of so-called 'World Coordinate System' (or WCS) FITS keywords that define the coordinate transformation that must be applied to the values in the 'X' and 'Y' columns to calculate the coordinate.

Alternatively, one can perform spatial filtering using unitless 'pixel' coordinates for the regions and row positions. In this case the user must be careful to ensure that the positions in the 2 files are self-consistent. A typical problem is that the region file may be generated using a binned image, but the unbinned coordinates are given in the event table. The ROSAT events files, for example, have X and Y pixel coordinates that range from 1 - 15360. These coordinates are typically binned by a factor of 32 to produce a 480x480 pixel image. If one then uses a region file generated from this image (in image pixel units) to filter the ROSAT events file, then the X and Y column values must be converted to corresponding pixel units as in:

```
regfilter("rosat.reg", X/32.+.5, Y/32.+.5)
```

Note that this binning conversion is not necessary if the region file is specified using celestial coordinate units instead of pixel units because CFITSIO is then able to directly compare the celestial coordinate of each row in the table with the celestial coordinates in the region file without having to know anything about how the image may have been binned.

The last "wcs cols" parameter should rarely be needed. If supplied, this string contains the names of the 2 columns (space or comma separated) which have the associated WCS keywords. If not supplied, the filter will scan the X and Y expressions for column names. If only one is found in each expression, those columns will be used, otherwise an error will be returned.

These region shapes are supported (names are case insensitive):

```
Point         ( X1, Y1 )               <- One pixel square region
Line          ( X1, Y1, X2, Y2 )       <- One pixel wide region
Polygon       ( X1, Y1, X2, Y2, ... )  <- Rest are interiors with
Rectangle     ( X1, Y1, X2, Y2, A )       | boundaries considered
Box           ( Xc, Yc, Wdth, Hght, A )   V within the region
Diamond       ( Xc, Yc, Wdth, Hght, A )
Circle        ( Xc, Yc, R )
Annulus       ( Xc, Yc, Rin, Rout )
Ellipse       ( Xc, Yc, Rx, Ry, A )
```

```
Elliptannulus ( Xc, Yc, Rinx, Riny, Routx, Routy, Ain, Aout )
Sector        ( Xc, Yc, Amin, Amax )
```

where (Xc,Yc) is the coordinate of the shape's center; (X#,Y#) are the coordinates of the shape's edges; Rxxx are the shapes' various Radii or semi-major/minor axes; and Axxx are the angles of rotation (or bounding angles for Sector) in degrees. For rotated shapes, the rotation angle can be left off, indicating no rotation. Common alternate names for the regions can also be used: rotbox = box; rotrectangle = rectangle; (rot)rhombus = (rot)diamond; and pie = sector. When a shape's name is preceded by a minus sign, '-', the defined region is instead the area *outside* its boundary (ie, the region is inverted). All the shapes within a single region file are OR'd together to create the region, and the order is significant. The overall way of looking at region files is that if the first region is an excluded region then a dummy included region of the whole detector is inserted in the front. Then each region specification as it is processed overrides any selections inside of that region specified by previous regions. Another way of thinking about this is that if a previous excluded region is completely inside of a subsequent included region the excluded region is ignored.

The positional coordinates may be given either in pixel units, decimal degrees or hh:mm:ss.s, dd:mm:ss.s units. The shape sizes may be given in pixels, degrees, arcminutes, or arcseconds. Look at examples of region file produced by fv/POW or ds9 for further details of the region file format.

There are three functions that are primarily for use with SAO region files and the FSAOI task, but they can be used directly. They return a boolean true or false depending on whether a two dimensional point is in the region or not:

```
"point in a circular region"
      circle(xcntr,ycntr,radius,Xcolumn,Ycolumn)

"point in an elliptical region"
     ellipse(xcntr,ycntr,xhlf_wdth,yhlf_wdth,rotation,Xcolumn,Ycolumn)

"point in a rectangular region"
         box(xcntr,ycntr,xfll_wdth,yfll_wdth,rotation,Xcolumn,Ycolumn)

where
   (xcntr,ycntr) are the (x,y) position of the center of the region
   (xhlf_wdth,yhlf_wdth) are the (x,y) half widths of the region
   (xfll_wdth,yfll_wdth) are the (x,y) full widths of the region
   (radius) is half the diameter of the circle
   (rotation) is the angle(degrees) that the region is rotated with
         respect to (xcntr,ycntr)
   (Xcoord,Ycoord) are the (x,y) coordinates to test, usually column
         names
   NOTE: each parameter can itself be an expression, not merely a
         column name or constant.
```

8.11.6 Example Row Filters

```
[ binary && mag <= 5.0]        - Extract all binary stars brighter
                                 than  fifth magnitude (note that
                                 the initial space is necessary to
                                 prevent it from being treated as a
                                 binning specification)

[#row >= 125 && #row <= 175]   - Extract row numbers 125 through 175

[IMAGE[4,5] .gt. 100]          - Extract all rows that have the
                                 (4,5) component of the IMAGE column
                                 greater than 100

[abs(sin(theta * #deg)) < 0.5] - Extract all rows having the
                                 absolute value of the sine of theta
                                 less  than a half where the angles
                                 are tabulated in degrees

[SUM( SPEC > 3*BACKGRND )>=1]  - Extract all rows containing a
                                 spectrum, held in vector column
                                 SPEC, with at least one value 3
                                 times greater than the background
                                 level held in a keyword, BACKGRND

[VCOL=={1,4,2}]                - Extract all rows whose vector column
                                 VCOL contains the 3-elements 1, 4, and
                                 2.

[@rowFilter.txt]               - Extract rows using the expression
                                 contained within the text file
                                 rowFilter.txt

[gtifilter()]                  - Search the current file for a GTI
 extension,  filter  the TIME
 column in the current table, using
 START/STOP times taken from
 columns in the GTI  extension

[regfilter("pow.reg")]         - Extract rows which have a coordinate
                                 (as given in the X and Y columns)
                                 within the spatial region specified
                                 in the pow.reg region file.

[regfilter("pow.reg", Xs, Ys)] - Same as above, except that the
                                 Xs and Ys columns will be used to
```

```
                        determine the coordinate of each
                        row in the table.
```

8.12 Binning or Histogramming Specification

The optional binning specifier is enclosed in square brackets and can be distinguished from a general row filter specification by the fact that it begins with the keyword 'bin' not immediately followed by an equals sign. When binning is specified, a temporary N-dimensional FITS primary array is created by computing the histogram of the values in the specified columns of a FITS table extension. After the histogram is computed the input FITS file containing the table is then closed and the temporary FITS primary array is opened and passed to the application program. Thus, the application program never sees the original FITS table and only sees the image in the new temporary file (which has no additional extensions). Obviously, the application program must be expecting to open a FITS image and not a FITS table in this case.

The data type of the FITS histogram image may be specified by appending 'b' (for 8-bit byte), 'i' (for 16-bit integers), 'j' (for 32-bit integer), 'r' (for 32-bit floating points), or 'd' (for 64-bit double precision floating point) to the 'bin' keyword (e.g. '[binr X]' creates a real floating point image). If the datatype is not explicitly specified then a 32-bit integer image will be created by default, unless the weighting option is also specified in which case the image will have a 32-bit floating point data type by default.

The histogram image may have from 1 to 4 dimensions (axes), depending on the number of columns that are specified. The general form of the binning specification is:

```
 [bin{bijrd}  Xcol=min:max:binsize, Ycol= ..., Zcol=..., Tcol=...; weight]
```

in which up to 4 columns, each corresponding to an axis of the image, are listed. The column names are case insensitive, and the column number may be given instead of the name, preceded by a pound sign (e.g., [bin #4=1:512]). If the column name is not specified, then CFITSIO will first try to use the 'preferred column' as specified by the CPREF keyword if it exists (e.g., 'CPREF = 'DETX,DETY'), otherwise column names 'X', 'Y', 'Z', and 'T' will be assumed for each of the 4 axes, respectively. In cases where the column name could be confused with an arithmetic expression, enclose the column name in parentheses to force the name to be interpreted literally.

Each column name may be followed by an equals sign and then the lower and upper range of the histogram, and the size of the histogram bins, separated by colons. Spaces are allowed before and after the equals sign but not within the 'min:max:binsize' string. The min, max and binsize values may be integer or floating point numbers, or they may be the names of keywords in the header of the table. If the latter, then the value of that keyword is substituted into the expression.

Default values for the min, max and binsize quantities will be used if not explicitly given in the binning expression as shown in these examples:

```
    [bin x = :512:2]  - use default minimum value
    [bin x = 1::2]    - use default maximum value
    [bin x = 1:512]   - use default bin size
```

```
[bin x = 1:]        - use default maximum value and bin size
[bin x = :512]      - use default minimum value and bin size
[bin x = 2]         - use default minimum and maximum values
[bin x]             - use default minimum, maximum and bin size
[bin 4]             - default 2-D image, bin size = 4 in both axes
[bin]               - default 2-D image
```

CFITSIO will use the value of the TLMINn, TLMAXn, and TDBINn keywords, if they exist, for the default min, max, and binsize, respectively. If they do not exist then CFITSIO will use the actual minimum and maximum values in the column for the histogram min and max values. The default binsize will be set to 1, or (max - min) / 10., whichever is smaller, so that the histogram will have at least 10 bins along each axis.

A shortcut notation is allowed if all the columns/axes have the same binning specification. In this case all the column names may be listed within parentheses, followed by the (single) binning specification, as in:

```
[bin (X,Y)=1:512:2]
[bin (X,Y) = 5]
```

The optional weighting factor is the last item in the binning specifier and, if present, is separated from the list of columns by a semi-colon. As the histogram is accumulated, this weight is used to incremented the value of the appropriated bin in the histogram. If the weighting factor is not specified, then the default weight = 1 is assumed. The weighting factor may be a constant integer or floating point number, or the name of a keyword containing the weighting value. Or the weighting factor may be the name of a table column in which case the value in that column, on a row by row basis, will be used.

In some cases, the column or keyword may give the reciprocal of the actual weight value that is needed. In this case, precede the weight keyword or column name by a slash '/' to tell CFITSIO to use the reciprocal of the value when constructing the histogram.

For complex or commonly used histograms, one can also place its description into a text file and import it into the binning specification using the syntax '[bin @filename.txt]'. The file's contents can extend over multiple lines, although it must still conform to the no-spaces rule for the min:max:binsize syntax and each axis specification must still be comma-separated. Any lines in the external text file that begin with 2 slash characters ('//') will be ignored and may be used to add comments into the file.

Examples:

```
[bini detx, dety]                - 2-D, 16-bit integer histogram
                                   of DETX and DETY columns, using
                                   default values for the histogram
                                   range and binsize

[bin (detx, dety)=16; /exposure] - 2-D, 32-bit real histogram of DETX
                                   and DETY columns with a bin size = 16
```

```
                                   in both axes. The histogram values
                                   are divided by the EXPOSURE keyword
                                   value.

[bin time=TSTART:TSTOP:0.1]      - 1-D lightcurve, range determined by
                                   the TSTART and TSTOP keywords,
                                   with 0.1 unit size bins.

[bin pha, time=8000.:8100.:0.1]  - 2-D image using default binning
                                   of the PHA column for the X axis,
                                   and 1000 bins in the range
                                   8000. to 8100. for the Y axis.

[bin @binFilter.txt]             - Use the contents of the text file
                                   binFilter.txt for the binning
                                   specifications.
```

Chapter 9

Template Files

When a new FITS file is created with a call to fits_create_file, the name of a template file may be supplied in parentheses immediately following the name of the new file to be created. This template is used to define the structure of one or more HDUs in the new file. The template file may be another FITS file, in which case the newly created file will have exactly the same keywords in each HDU as in the template FITS file, but all the data units will be filled with zeros. The template file may also be an ASCII text file, where each line (in general) describes one FITS keyword record. The format of the ASCII template file is described in the following sections.

9.1 Detailed Template Line Format

The format of each ASCII template line closely follows the format of a FITS keyword record:

```
KEYWORD = KEYVALUE / COMMENT
```

except that free format may be used (e.g., the equals sign may appear at any position in the line) and TAB characters are allowed and are treated the same as space characters. The KEYVALUE and COMMENT fields are optional. The equals sign character is also optional, but it is recommended that it be included for clarity. Any template line that begins with the pound '#' character is ignored by the template parser and may be use to insert comments into the template file itself.

The KEYWORD name field is limited to 8 characters in length and only the letters A-Z, digits 0-9, and the hyphen and underscore characters may be used, without any embedded spaces. Lowercase letters in the template keyword name will be converted to uppercase. Leading spaces in the template line preceding the keyword name are generally ignored, except if the first 8 characters of a template line are all blank, then the entire line is treated as a FITS comment keyword (with a blank keyword name) and is copied verbatim into the FITS header.

The KEYVALUE field may have any allowed FITS data type: character string, logical, integer, real, complex integer, or complex real. The character string values need not be enclosed in single quote characters unless they are necessary to distinguish the string from a different data type (e.g. 2.0 is a real but '2.0' is a string). The keyword has an undefined (null) value if the template record only contains blanks following the "=" or between the "=" and the "/" comment field delimiter.

String keyword values longer than 68 characters (the maximum length that will fit in a single FITS keyword record) are permitted using the CFITSIO long string convention. They can either be specified as a single long line in the template, or by using multiple lines where the continuing lines contain the 'CONTINUE' keyword, as in this example:

```
LONGKEY = 'This is a long string value that is contin&'
CONTINUE  'ued over 2 records' / comment field goes here
```

The format of template lines with CONTINUE keyword is very strict: 3 spaces must follow CONTINUE and the rest of the line is copied verbatim to the FITS file.

The start of the optional COMMENT field must be preceded by "/", which is used to separate it from the keyword value field. Exceptions are if the KEYWORD name field contains COMMENT, HISTORY, CONTINUE, or if the first 8 characters of the template line are blanks.

More than one Header-Data Unit (HDU) may be defined in the template file. The start of an HDU definition is denoted with a SIMPLE or XTENSION template line:

1) SIMPLE begins a Primary HDU definition. SIMPLE may only appear as the first keyword in the template file. If the template file begins with XTENSION instead of SIMPLE, then a default empty Primary HDU is created, and the template is then assumed to define the keywords starting with the first extension following the Primary HDU.

2) XTENSION marks the beginning of a new extension HDU definition. The previous HDU will be closed at this point and processing of the next extension begins.

9.2 Auto-indexing of Keywords

If a template keyword name ends with a "#" character, it is said to be 'auto-indexed'. Each "#" character will be replaced by the current integer index value, which gets reset = 1 at the start of each new HDU in the file (or 7 in the special case of a GROUP definition). The FIRST indexed keyword in each template HDU definition is used as the 'incrementor'; each subsequent occurrence of this SAME keyword will cause the index value to be incremented. This behavior can be rather subtle, as illustrated in the following examples in which the TTYPE keyword is the incrementor in both cases:

```
TTYPE# = TIME
TFORM# = 1D
TTYPE# = RATE
TFORM# = 1E
```

will create TTYPE1, TFORM1, TTYPE2, and TFORM2 keywords. But if the template looks like,

```
TTYPE# = TIME
TTYPE# = RATE
TFORM# = 1D
TFORM# = 1E
```

this results in a FITS files with TTYPE1, TTYPE2, TFORM2, and TFORM2, which is probably not what was intended!

9.3 Template Parser Directives

In addition to the template lines which define individual keywords, the template parser recognizes 3 special directives which are each preceded by the backslash character: `\include, \group`, and `\end`.

The 'include' directive must be followed by a filename. It forces the parser to temporarily stop reading the current template file and begin reading the include file. Once the parser reaches the end of the include file it continues parsing the current template file. Include files can be nested, and HDU definitions can span multiple template files.

The start of a GROUP definition is denoted with the 'group' directive, and the end of a GROUP definition is denoted with the 'end' directive. Each GROUP contains 0 or more member blocks (HDUs or GROUPs). Member blocks of type GROUP can contain their own member blocks. The GROUP definition itself occupies one FITS file HDU of special type (GROUP HDU), so if a template specifies 1 group with 1 member HDU like:

```
\group
grpdescr = 'demo'
xtension bintable
# this bintable has 0 cols, 0 rows
\end
```

then the parser creates a FITS file with 3 HDUs :

```
1) dummy PHDU
2) GROUP HDU (has 1 member, which is bintable in HDU number 3)
3) bintable (member of GROUP in HDU number 2)
```

Technically speaking, the GROUP HDU is a BINTABLE with 6 columns. Applications can define additional columns in a GROUP HDU using TFORMn and TTYPEn (where n is 7, 8, ...) keywords or their auto-indexing equivalents.

For a more complicated example of a template file using the group directives, look at the sample.tpl file that is included in the CFITSIO distribution.

9.4 Formal Template Syntax

The template syntax can formally be defined as follows:

```
    TEMPLATE = BLOCK [ BLOCK ... ]
```

```
      BLOCK = { HDU | GROUP }

      GROUP = \GROUP [ BLOCK ... ] \END

        HDU = XTENSION [ LINE ... ] { XTENSION | \GROUP | \END | EOF }

       LINE = [ KEYWORD [ = ] ] [ VALUE ] [ / COMMENT ]

    X ...     - X can be present 1 or more times
    { X | Y } - X or Y
    [ X ]     - X is optional
```

At the topmost level, the template defines 1 or more template blocks. Blocks can be either HDU (Header Data Unit) or a GROUP. For each block the parser creates 1 (or more for GROUPs) FITS file HDUs.

9.5 Errors

In general the fits_execute_template() function tries to be as atomic as possible, so either everything is done or nothing is done. If an error occurs during parsing of the template, fits_execute_template() will (try to) delete the top level BLOCK (with all its children if any) in which the error occurred, then it will stop reading the template file and it will return with an error.

9.6 Examples

1. This template file will create a 200 x 300 pixel image, with 4-byte integer pixel values, in the primary HDU:

```
  SIMPLE = T
  BITPIX = 32
  NAXIS = 2     / number of dimensions
  NAXIS1 = 100  / length of first axis
  NAXIS2 = 200  / length of second axis
  OBJECT = NGC 253 / name of observed object
```

The allowed values of BITPIX are 8, 16, 32, -32, or -64, representing, respectively, 8-bit integer, 16-bit integer, 32-bit integer, 32-bit floating point, or 64 bit floating point pixels.

2. To create a FITS table, the template first needs to include XTENSION = TABLE or BINTABLE to define whether it is an ASCII or binary table, and NAXIS2 to define the number of rows in the table. Two template lines are then needed to define the name (TTYPEn) and FITS data format (TFORMn) of the columns, as in this example:

```
  xtension = bintable
```

```
naxis2 = 40
ttype# = Name
tform# = 10a
ttype# = Npoints
tform# = j
ttype# = Rate
tunit# = counts/s
tform# = e
```

The above example defines a null primary array followed by a 40-row binary table extension with 3 columns called 'Name', 'Npoints', and 'Rate', with data formats of '10A' (ASCII character string), '1J' (integer) and '1E' (floating point), respectively. Note that the other required FITS keywords (BITPIX, NAXIS, NAXIS1, PCOUNT, GCOUNT, TFIELDS, and END) do not need to be explicitly defined in the template because their values can be inferred from the other keywords in the template. This example also illustrates that the templates are generally case-insensitive (the keyword names and TFORMn values are converted to upper-case in the FITS file) and that string keyword values generally do not need to be enclosed in quotes.

Chapter 10

Summary of all FITSIO User-Interface Subroutines

Error Status Routines page 29

```
FTVERS( > version)
FTGERR(status, > errtext)
FTGMSG( > errmsg)
FTRPRT (stream, > status)
FTPMSG(errmsg)
FTPMRK
FTCMSG
FTCMRK
```

FITS File Open and Close Subroutines: page 35

```
        FTOPEN(unit,filename,rwmode, > blocksize,status)
        FTDKOPEN(unit,filename,rwmode, > blocksize,status)
        FTNOPN(unit,filename,rwmode, > status)
        FTDOPN(unit,filename,rwmode, > status)
        FTTOPN(unit,filename,rwmode, > status)
        FTIOPN(unit,filename,rwmode, > status)
        FTREOPEN(unit, > newunit, status)
        FTINIT(unit,filename,blocksize, > status)
        FTDKINIT(unit,filename,blocksize, > status)
        FTTPLT(unit, filename, tplfilename, > status)
        FTFLUS(unit, > status)
        FTCLOS(unit, > status)
        FTDELT(unit, > status)
        FTGIOU( > iounit, status)
        FTFIOU(iounit, > status)
CFITS2Unit(fitsfile *ptr)   (C routine)
```

```
CUnit2FITS(int unit)       (C routine)
FTEXTN(filename, > nhdu, status)
FTFLNM(unit, > filename, status)
FTFLMD(unit, > iomode, status)
FFURLT(unit, > urltype, status)
FTIURL(filename, > filetype, infile, outfile, extspec, filter,
       binspec, colspec, status)
FTRTNM(filename, > rootname, status)
FTEXIST(filename, > exist, status)
```

HDU-Level Operations: page 38

```
FTMAHD(unit,nhdu, > hdutype,status)
FTMRHD(unit,nmove, > hdutype,status)
FTGHDN(unit, > nhdu)
FTMNHD(unit, hdutype, extname, extver, > status)
FTGHDT(unit, > hdutype, status)
FTTHDU(unit, > hdunum, status)
FTCRHD(unit, > status)
FTIIMG(unit,bitpix,naxis,naxes, > status)
FTITAB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTIBIN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTRSIM(unit,bitpix,naxis,naxes,status)
FTDHDU(unit, > hdutype,status)
FTCPFL(iunit,ounit,previous, current, following, > status)
FTCOPY(iunit,ounit,morekeys, > status)
FTCPHD(inunit, outunit, > status)
FTCPDT(iunit,ounit, > status)
```

Subroutines to specify or modify the structure of the CHDU: page 41

```
FTRDEF(unit, > status)  (DEPRECATED)
FTPDEF(unit,bitpix,naxis,naxes,pcount,gcount, > status)  (DEPRECATED)
FTADEF(unit,rowlen,tfields,tbcol,tform,nrows > status)  (DEPRECATED)
FTBDEF(unit,tfields,tform,varidat,nrows > status)  (DEPRECATED)
FTDDEF(unit,bytlen, > status)  (DEPRECATED)
FTPTHP(unit,theap, > status)
```

Header Space and Position Subroutines: page 43

```
FTHDEF(unit,morekeys, > status)
FTGHSP(iunit, > keysexist,keysadd,status)
FTGHPS(iunit, > keysexist,key_no,status)
```

Read or Write Standard Header Subroutines: page 43

```
FTPHPS(unit,bitpix,naxis,naxes, > status)
FTPHPR(unit,simple,bitpix,naxis,naxes,pcount,gcount,extend, > status)
FTGHPR(unit,maxdim, > simple,bitpix,naxis,naxes,pcount,gcount,extend,
       status)
FTPHTB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTGHTB(unit,maxdim, > rowlen,nrows,tfields,ttype,tbcol,tform,tunit,
       extname,status)
FTPHBN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTGHBN(unit,maxdim, > nrows,tfields,ttype,tform,tunit,extname,varidat,
       status)
```

Write Keyword Subroutines: page 45

```
FTPREC(unit,card, > status)
FTPCOM(unit,comment, > status)
FTPHIS(unit,history, > status)
FTPDAT(unit, > status)
FTPKY[JKLS](unit,keyword,keyval,comment, > status)
FTPKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTPKLS(unit,keyword,keyval,comment, > status)
FTPLSW(unit, > status)
FTPKYU(unit,keyword,comment, > status)
FTPKN[JKLS](unit,keyroot,startno,no_keys,keyvals,comments, > status)
FTPKN[EDFG](unit,keyroot,startno,no_keys,keyvals,decimals,comments, >
           status)
FTCPKYinunit, outunit, innum, outnum, keyroot, > status)
FTPKYT(unit,keyword,intval,dblval,comment, > status)
FTPKTP(unit, filename, > status)
FTPUNT(unit,keyword,units, > status)
```

Insert Keyword Subroutines: page 47

```
FTIREC(unit,key_no,card, > status)
FTIKY[JKLS](unit,keyword,keyval,comment, > status)
FTIKLS(unit,keyword,keyval,comment, > status)
FTIKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTIKYU(unit,keyword,comment, > status)
```

Read Keyword Subroutines: page 47

```
FTGREC(unit,key_no, > card,status)
FTGKYN(unit,key_no, > keyword,value,comment,status)
FTGCRD(unit,keyword, > card,status)
FTGNXK(unit,inclist,ninc,exclist,nexc, > card,status)
```

```
FTGKEY(unit,keyword, > value,comment,status)
FTGKY[EDJKLS](unit,keyword, > keyval,comment,status)
FTGKN[EDJKLS](unit,keyroot,startno,max_keys, > keyvals,nfound,status)
FTGKYT(unit,keyword, > intval,dblval,comment,status)
FTGUNT(unit,keyword, > units,status)
```

Modify Keyword Subroutines: page 49

```
FTMREC(unit,key_no,card, > status)
FTMCRD(unit,keyword,card, > status)
FTMNAM(unit,oldkey,keyword, > status)
FTMCOM(unit,keyword,comment, > status)
FTMKY[JKLS](unit,keyword,keyval,comment, > status)
FTMKLS(unit,keyword,keyval,comment, > status)
FTMKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTMKYU(unit,keyword,comment, > status)
```

Update Keyword Subroutines: page 50

```
FTUCRD(unit,keyword,card, > status)
FTUKY[JKLS](unit,keyword,keyval,comment, > status)
FTUKLS(unit,keyword,keyval,comment, > status)
FTUKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTUKYU(unit,keyword,comment, > status)
```

Delete Keyword Subroutines: page 50

```
FTDREC(unit,key_no, > status)
FTDKEY(unit,keyword, > status)
```

Define Data Scaling Parameters and Undefined Pixel Flags: page 51

```
FTPSCL(unit,bscale,bzero, > status)
FTTSCL(unit,colnum,tscal,tzero, > status)
FTPNUL(unit,blank, > status)
FTSNUL(unit,colnum,snull > status)
FTTNUL(unit,colnum,tnull > status)
```

FITS Primary Array or IMAGE Extension I/O Subroutines: page 52

```
FTGIDT(unit, > bitpix,status)
FTGIET(unit, > bitpix,status)
FTGIDM(unit, > naxis,status)
FTGISZ(unit, maxdim, > naxes,status)
```

```
FTGIPR(unit, maxdim, > bitpix,naxis,naxes,status)
FTPPR[BIJKED](unit,group,fpixel,nelements,values, > status)
FTPPN[BIJKED](unit,group,fpixel,nelements,values,nullval > status)
FTPPRU(unit,group,fpixel,nelements, > status)
FTGPV[BIJKED](unit,group,fpixel,nelements,nullval, > values,anyf,status)
FTGPF[BIJKED](unit,group,fpixel,nelements, > values,flagvals,anyf,status)
FTPGP[BIJKED](unit,group,fparm,nparm,values, > status)
FTGGP[BIJKED](unit,group,fparm,nparm, > values,status)
FTP2D[BIJKED](unit,group,dim1,naxis1,naxis2,image, > status)
FTP3D[BIJKED](unit,group,dim1,dim2,naxis1,naxis2,naxis3,cube, > status)
FTG2D[BIJKED](unit,group,nullval,dim1,naxis1,naxis2, > image,anyf,status)
FTG3D[BIJKED](unit,group,nullval,dim1,dim2,naxis1,naxis2,naxis3, >
              cube,anyf,status)
FTPSS[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,array, > status)
FTGSV[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,incs,nullval, >
              array,anyf,status)
FTGSF[BIJKED](unit,group,naxis,naxes,fpixels,lpixels,incs, >
              array,flagvals,anyf,status)
```

Table Column Information Subroutines: page 55

```
FTGNRW(unit, > nrows, status)
FTGNCL(unit, > ncols, status)
FTGCNO(unit,casesen,coltemplate, > colnum,status)
FTGCNN(unit,casesen,coltemplate, > colnam,colnum,status)
FTGTCL(unit,colnum, > datacode,repeat,width,status)
FTEQTY(unit,colnum, > datacode,repeat,width,status)
FTGCDW(unit,colnum, > dispwidth,status)
FTGACL(unit,colnum, >
       ttype,tbcol,tunit,tform,tscal,tzero,snull,tdisp,status)
FTGBCL(unit,colnum, >
       ttype,tunit,datatype,repeat,tscal,tzero,tnull,tdisp,status)
FTPTDM(unit,colnum,naxis,naxes, > status)
FTGTDM(unit,colnum,maxdim, > naxis,naxes,status)
FTDTDM(unit,tdimstr,colnum,maxdim, > naxis,naxes, status)
FFGRSZ(unit, > nrows,status)
```

Low-Level Table Access Subroutines: page 58

```
FTGTBS(unit,frow,startchar,nchars, > string,status)
FTPTBS(unit,frow,startchar,nchars,string, > status)
FTGTBB(unit,frow,startchar,nchars, > array,status)
FTPTBB(unit,frow,startchar,nchars,array, > status)
```

Edit Rows or Columns page 58

```
FTIROW(unit,frow,nrows, > status)
FTDROW(unit,frow,nrows, > status)
FTDRRG(unit,rowrange, > status)
FTDRWS(unit,rowlist,nrows, > status)
FTICOL(unit,colnum,ttype,tform, > status)
FTICLS(unit,colnum,ncols,ttype,tform, > status)
FTMVEC(unit,colnum,newveclen, > status)
FTDCOL(unit,colnum, > status)
FTCPCL(inunit,outunit,incolnum,outcolnum,createcol, > status);
```

Read and Write Column Data Routines page 60

```
FTPCL[SLBIJKEDCM](unit,colnum,frow,felem,nelements,values, > status)
FTPCN[BIJKED](unit,colnum,frow,felem,nelements,values,nullval > status)
FTPCLX(unit,colnum,frow,fbit,nbit,lray, > status)
FTPCLU(unit,colnum,frow,felem,nelements, > status)
FTGCL(unit,colnum,frow,felem,nelements, > values,status)
FTGCV[SBIJKEDCM](unit,colnum,frow,felem,nelements,nullval, >
               values,anyf,status)
FTGCF[SLBIJKEDCM](unit,colnum,frow,felem,nelements, >
                 values,flagvals,anyf,status)
FTGSV[BIJKED](unit,colnum,naxis,naxes,fpixels,lpixels,incs,nullval, >
             array,anyf,status)
FTGSF[BIJKED](unit,colnum,naxis,naxes,fpixels,lpixels,incs, >
             array,flagvals,anyf,status)
FTGCX(unit,colnum,frow,fbit,nbit, > lray,status)
FTGCX[IJD](unit,colnum,frow,nrows,fbit,nbit, > array,status)
FTGDES(unit,colnum,rownum, > nelements,offset,status)
FTPDES(unit,colnum,rownum,nelements,offset, > status)
```

Row Selection and Calculator Routines: page 64

```
FTFROW(unit,expr,firstrow, nrows, > n_good_rows, row_status, status)
FTFFRW(unit, expr, > rownum, status)
FTSROW(inunit, outunit, expr, > status )
FTCROW(unit,datatype,expr,firstrow,nelements,nulval, >
     array,anynul,status)
FTCALC(inunit, expr, outunit, parName, parInfo, > status)
FTCALC_RNG(inunit, expr, outunit, parName, parInfo,
    nranges, firstrow, lastrow, > status)
FTTEXP(unit, expr, > datatype, nelem, naxis, naxes, status)
```

Celestial Coordinate System Subroutines: page 65

```
FTGICS(unit, > xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
```

```
FTGTCS(unit,xcol,ycol, >
       xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
FTWLDP(xpix,ypix,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                  coordtype, > xpos,ypos,status)
FTXYPX(xpos,ypos,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                  coordtype, > xpix,ypix,status)
```

File Checksum Subroutines: page 67

```
FTPCKS(unit, > status)
FTUCKS(unit, > status)
FTVCKS(unit, > dataok,hduok,status)
FTGCKS(unit, > datasum,hdusum,status)
FTESUM(sum,complement, > checksum)
FTDSUM(checksum,complement, > sum)
```

Time and Date Utility Subroutines: page 68

```
FTGSDT( > day, month, year, status )
FTGSTM(> datestr, timeref, status)
FTDT2S( year, month, day, > datestr, status)
FTTM2S( year, month, day, hour, minute, second, decimals,
        > datestr, status)
FTS2DT(datestr, > year, month, day, status)
FTS2TM(datestr, > year, month, day, hour, minute, second, status)
```

General Utility Subroutines: page 69

```
FTGHAD(unit, > curaddr,nextaddr)
FTUPCH(string)
FTCMPS(str_template,string,casesen, > match,exact)
FTTKEY(keyword, > status)
FTTREC(card, > status)
FTNCHK(unit, > status)
FTGKNM(unit, > keyword, keylength, status)
FTPSVC(card, > value,comment,status)
FTKEYN(keyroot,seq_no, > keyword,status)
FTNKEY(seq_no,keyroot, > keyword,status)
FTDTYP(value, > dtype,status)
class = FTGKCL(card)
FTASFM(tform, > datacode,width,decimals,status)
FTBNFM(tform, > datacode,repeat,width,status)
FTGABC(tfields,tform,space, > rowlen,tbcol,status)
FTGTHD(template, > card,hdtype,status)
FTRWRG(rowlist, maxrows, maxranges, > numranges, rangemin,
       rangemax, status)
```

Chapter 11

Parameter Definitions

```
anyf - (logical) set to TRUE if any of the returned data values are undefined
array - (any datatype except character) array of bytes to be read or written.
bitpix - (integer) bits per pixel: 8, 16, 32, -32, or -64
blank - (integer) value used for undefined pixels in integer primary array
blank - (integer*8) value used for undefined pixels in integer primary array
blocksize - (integer) 2880-byte logical record blocking factor
          (if 0 < blocksize < 11) or the actual block size in bytes
          (if 10 < blocksize < 28800).  As of version 3.3 of FITSIO,
          blocksizes greater than 2880 are no longer supported.
bscale - (double precision) scaling factor for the primary array
bytlen - (integer) length of the data unit, in bytes
bzero - (double precision) zero point for primary array scaling
card - (character*80) header record to be read or written
casesen - (logical) will string matching be case sensitive?
checksum - (character*16) encoded checksum string
colname - (character) ASCII name of the column
colnum - (integer) number of the column (first column = 1)
coltemplate - (character) template string to be matched to column names
comment - (character) the keyword comment field
comments - (character array) keyword comment fields
compid - (integer) the type of computer that the program is running on
complement - (logical) should the checksum be complemented?
coordtype - (character) type of coordinate projection (-SIN, -TAN, -ARC,
          -NCP, -GLS, -MER, or -AIT)
cube - 3D data cube of the appropriate datatype
curaddr - (integer) starting address (in bytes) of the CHDU
current - (integer) if not equal to 0, copy the current HDU
datacode - (integer) symbolic code of the binary table column datatype
dataok - (integer) was the data unit verification successful (=1) or
         not (= -1).  Equals zero if the DATASUM keyword is not present.
datasum - (double precision) 32-bit 1's complement checksum for the data unit
datatype - (character) datatype (format) of the binary table column
```

```
datestr  - (string) FITS date/time string: 'YYYY-MM-DDThh:mm:ss.ddd',
           'YYYY-MM-dd', or 'dd/mm/yy'
day - (integer) current day of the month
dblval - (double precision) fractional part of the keyword value
decimals - (integer) number of decimal places to be displayed
dim1 - (integer) actual size of the first dimension of the image or cube array
dim2 - (integer) actual size of the second dimension of the cube array
dispwidth - (integer) - the display width (length of string) for a column
dtype - (character) datatype of the keyword ('C', 'L', 'I',  or 'F')
                C = character string
                L = logical
                I = integer
                F = floating point number
errmsg - (character*80) oldest error message on the internal stack
errtext - (character*30) descriptive error message corresponding to error number
casesen - (logical) true if column name matching is case sensitive
exact - (logical) do the strings match exactly, or were wildcards used?
exclist  (character array) list of names to be excluded from search
exists   - flag indicating whether the file or compressed file exists on disk
extend - (logical) true if there may be extensions following the primary data
extname - (character) value of the EXTNAME keyword (if not blank)
fbit - (integer) first bit in the field to be read or written
felem - (integer) first pixel of the element vector (ignored for ASCII tables)
filename - (character) name of the FITS file
flagvals - (logical array) True if corresponding data element is undefined
following - (integer) if not equal to 0, copy all following HDUs in the input file
fparm - (integer) sequence number of the first group parameter to read or write
fpixel - (integer) the first pixel position
fpixels - (integer array) the first included pixel in each dimension
frow - (integer) beginning row number (first row of table = 1)
frowll - (integer*8) beginning row number (first row of table = 1)
gcount - (integer) value of the GCOUNT keyword (usually = 1)
group - (integer) sequence number of the data group (=0 for non-grouped data)
hdtype - (integer) header record type: -1=delete;  0=append or replace;
                   1=append; 2=this is the END keyword
hduok - (integer) was the HDU verification successful (=1) or
         not (= -1).  Equals zero if the CHECKSUM keyword is not present.
hdusum - (double precision) 32 bit 1's complement checksum for the entire CHDU
hdutype - (integer) type of HDU: 0 = primary array or IMAGE, 1 = ASCII table,
                   2 = binary table, -1 = any HDU type or unknown type
history - (character) the HISTORY keyword comment string
hour - (integer) hour from 0 - 23
image - 2D image of the appropriate datatype
inclist  (character array) list of names to be included in search
incs - (integer array) sampling interval for pixels in each FITS dimension
intval - (integer) integer part of the keyword value
```

iounit - (integer) value of an unused I/O unit number
iunit - (integer) logical unit number associated with the input FITS file, 1-199
key_no - (integer) sequence number (starting with 1) of the keyword record
keylength - (integer) length of the keyword name
keyroot - (character) root string for the keyword name
keysadd -(integer) number of new keyword records which can fit in the CHU
keysexist - (integer) number of existing keyword records in the CHU
keyval - value of the keyword in the appropriate datatype
keyvals - (array) value of the keywords in the appropriate datatype
keyword - (character*8) name of a keyword
lray - (logical array) array of logical values corresponding to the bit array
lpixels - (integer array) the last included pixel in each dimension
match - (logical) do the 2 strings match?
maxdim - (integer) dimensioned size of the NAXES, TTYPE, TFORM or TUNIT arrays
max_keys - (integer) maximum number of keywords to search for
minute - (integer) minute of an hour (0 - 59)
month - (integer) current month of the year (1 - 12)
morekeys - (integer) will leave space in the header for this many more keywords
naxes - (integer array) size of each dimension in the FITS array
naxesll - (integer*8 array) size of each dimension in the FITS array
naxis - (integer) number of dimensions in the FITS array
naxis1 - (integer) length of the X/first axis of the FITS array
naxis2 - (integer) length of the Y/second axis of the FITS array
naxis3 - (integer) length of the Z/third axis of the FITS array
nbit - (integer) number of bits in the field to read or write
nchars - (integer) number of characters to read and return
ncols - (integer) number of columns
nelements - (integer) number of data elements to read or write
nelementsll - (integer*8) number of data elements to read or write
nexc   (integer) number of names in the exclusion list (may = 0)
nhdu - (integer) absolute number of the HDU (1st HDU = 1)
ninc   (integer) number of names in the inclusion list
nmove - (integer) number of HDUs to move (+ or -), relative to current position
nfound - (integer) number of keywords found (highest keyword number)
no_keys - (integer) number of keywords to write in the sequence
nparm - (integer) number of group parameters to read or write
nrows - (integer) number of rows in the table
nrowsll - (integer*8) number of rows in the table
nullval - value to represent undefined pixels, of the appropriate datatype
nextaddr - (integer) starting address (in bytes) of the HDU following the CHDU
offset - (integer) byte offset in the heap to the first element of the array
offsetll - (integer*8) byte offset in the heap to the first element of the array
oldkey - (character) old name of keyword to be modified
ounit - (integer) logical unit number associated with the output FITS file 1-199
pcount - (integer) value of the PCOUNT keyword (usually = 0)
previous - (integer) if not equal to 0, copy all previous HDUs in the input file

```
repeat - (integer) length of element vector (e.g. 12J); ignored for ASCII table
rot - (double precision) celestial coordinate rotation angle (degrees)
rowlen - (integer) length of a table row, in characters or bytes
rowlenll - (integer*8) length of a table row, in characters or bytes
rowlist - (integer array) list of row numbers to be deleted in increasing order
rownum - (integer) number of the row (first row = 1)
rowrange- (string) list of rows or row ranges to be deleted
rwmode - (integer) file access mode: 0 = readonly, 1 = readwrite
second   (double)- second within minute (0 - 60.9999999999) (leap second!)
seq_no - (integer) the sequence number to append to the keyword root name
simple - (logical) does the FITS file conform to all the FITS standards
snull - (character) value used to represent undefined values in ASCII table
space - (integer) number of blank spaces to leave between ASCII table columns
startchar - (integer) first character in the row to be read
startno - (integer) value of the first keyword sequence number (usually 1)
status - (integer) returned error status code (0 = OK)
str_template (character) template string to be matched to reference string
stream - (character) output stream for the report: either 'STDOUT' or 'STDERR'
string - (character) character string
sum - (double precision) 32 bit unsigned checksum value
tbcol - (integer array) column number of the first character in the field(s)
tdisp - (character) Fortran type display format for the table column
template-(character) template string for a FITS header record
tfields - (integer) number of fields (columns) in the table
tform - (character array) format of the column(s); allowed values are:
         For ASCII tables:  Iw, Aw, Fww.dd, Eww.dd, or Dww.dd
         For binary tables: rL, rX, rB, rI, rJ, rA, rAw, rE, rD, rC, rM
         where 'w'=width of the field, 'd'=no. of decimals, 'r'=repeat count
         Note that the 'rAw' form is non-standard extension to the
         TFORM keyword syntax that is not specifically defined in the
         Binary Tables definition document.
theap - (integer) zero indexed byte offset of starting address of the heap
         relative to the beginning of the binary table data
tnull - (integer) value used to represent undefined values in binary table
tnullll - (integer*8) value used to represent undefined values in binary table
ttype - (character array) label for table column(s)
tscal - (double precision) scaling factor for table column
tunit - (character array) physical unit for table column(s)
tzero - (double precision) scaling zero point for table column
unit  - (integer) logical unit number associated with the FITS file (1-199)
units - (character) the keyword units string (e.g., 'km/s')
value - (character) the keyword value string
values - array of data values of the appropriate datatype
varidat - (integer) size in bytes of the 'variable length data area'
           following the binary table data (usually = 0)
version - (real) current revision number of the library
```

width - (integer) width of the character string field
xcol - (integer) number of the column containing the X coordinate values
xinc - (double precision) X axis coordinate increment at reference pixel (deg)
xpix - (double precision) X axis pixel location
xpos - (double precision) X axis celestial coordinate (usually RA) (deg)
xrpix - (double precision) X axis reference pixel array location
xrval - (double precision) X axis coordinate value at the reference pixel (deg)
ycol - (integer) number of the column containing the X coordinate values
year - (integer) last 2 digits of the year (00 - 99)
yinc - (double precision) Y axis coordinate increment at reference pixel (deg)
ypix - (double precision) y axis pixel location
ypos - (double precision) y axis celestial coordinate (usually DEC) (deg)
yrpix - (double precision) Y axis reference pixel array location
yrval - (double precision) Y axis coordinate value at the reference pixel (deg)

Chapter 12

FITSIO Error Status Codes

```
Status codes in the range -99 to -999  and 1 to 999 are reserved for future
FITSIO use.

  0  OK, no error
101  input and output files are the same
103  too many FITS files open at once; all internal buffers full
104  error opening existing file
105  error creating new FITS file; (does a file with this name already exist?)
106  error writing record to FITS file
107  end-of-file encountered while reading record from FITS file
108  error reading record from file
110  error closing FITS file
111  internal array dimensions exceeded
112  Cannot modify file with readonly access
113  Could not allocate memory
114  illegal logical unit number; must be between 1 - 199, inclusive
115  NULL input pointer to routine
116  error seeking position in file

121  invalid URL prefix on file name
122  tried to register too many IO drivers
123  driver initialization failed
124  matching driver is not registered
125  failed to parse input file URL
126  parse error in range list

151  bad argument in shared memory driver
152  null pointer passed as an argument
153  no more free shared memory handles
154  shared memory driver is not initialized
155  IPC error returned by a system call
156  no memory in shared memory driver
```

157 resource deadlock would occur
158 attempt to open/create lock file failed
159 shared memory block cannot be resized at the moment

201 header not empty; can't write required keywords
202 specified keyword name was not found in the header
203 specified header record number is out of bounds
204 keyword value field is blank
205 keyword value string is missing the closing quote character
206 illegal indexed keyword name (e.g. 'TFORM1000')
207 illegal character in keyword name or header record
208 keyword does not have expected name. Keyword out of sequence?
209 keyword does not have expected integer value
210 could not find the required END header keyword
211 illegal BITPIX keyword value
212 illegal NAXIS keyword value
213 illegal NAXISn keyword value: must be 0 or positive integer
214 illegal PCOUNT keyword value
215 illegal GCOUNT keyword value
216 illegal TFIELDS keyword value
217 negative ASCII or binary table width value (NAXIS1)
218 negative number of rows in ASCII or binary table (NAXIS2)
219 column name (TTYPE keyword) not found
220 illegal SIMPLE keyword value
221 could not find the required SIMPLE header keyword
222 could not find the required BITPIX header keyword
223 could not find the required NAXIS header keyword
224 could not find all the required NAXISn keywords in the header
225 could not find the required XTENSION header keyword
226 the CHDU is not an ASCII table extension
227 the CHDU is not a binary table extension
228 could not find the required PCOUNT header keyword
229 could not find the required GCOUNT header keyword
230 could not find the required TFIELDS header keyword
231 could not find all the required TBCOLn keywords in the header
232 could not find all the required TFORMn keywords in the header
233 the CHDU is not an IMAGE extension
234 illegal TBCOL keyword value; out of range
235 this operation only allowed for ASCII or BINARY table extension
236 column is too wide to fit within the specified width of the ASCII table
237 the specified column name template matched more than one column name
241 binary table row width is not equal to the sum of the field widths
251 unrecognizable type of FITS extension
252 unrecognizable FITS record
253 END keyword contains non-blank characters in columns 9-80

```
254  Header fill area contains non-blank characters
255  Data fill area contains non-blank on non-zero values
261  unable to parse the TFORM keyword value string
262  unrecognizable TFORM datatype code
263  illegal TDIMn keyword value

301  illegal HDU number; less than 1 or greater than internal buffer size
302  column number out of range (1 - 999)
304  attempt to move to negative file record number
306  attempted to read or write a negative number of bytes in the FITS file
307  illegal starting row number for table read or write operation
308  illegal starting element number for table read or write operation
309  attempted to read or write character string in non-character table column
310  attempted to read or write logical value in non-logical table column
311  illegal ASCII table TFORM format code for attempted operation
312  illegal binary table TFORM format code for attempted operation
314  value for undefined pixels has not been defined
317  attempted to read or write descriptor in a non-descriptor field
320  number of array dimensions out of range
321  first pixel number is greater than the last pixel number
322  attempt to set BSCALE or TSCALn scaling parameter = 0
323  illegal axis length less than 1

340 NOT_GROUP_TABLE 340   Grouping function error
341  HDU_ALREADY_MEMBER
342 MEMBER_NOT_FOUND
343 GROUP_NOT_FOUND
344 BAD_GROUP_ID
345 TOO_MANY_HDUS_TRACKED
346 HDU_ALREADY_TRACKED
347 BAD_OPTION
348 IDENTICAL_POINTERS
349 BAD_GROUP_ATTACH
350 BAD_GROUP_DETACH

360 NGP_NO_MEMORY              malloc failed
361 NGP_READ_ERR               read error from file
362 NGP_NUL_PTR                null pointer passed as an argument.
                                 Passing null pointer as a name of
                                 template file raises this error
363 NGP_EMPTY_CURLINE          line read seems to be empty (used
                                 internally)
364 NGP_UNREAD_QUEUE_FULL      cannot unread more then 1 line (or single
                                 line twice)
365 NGP_INC_NESTING            too deep include file nesting (infinite
                                 loop, template includes itself ?)
```

```
366 NGP_ERR_FOPEN             fopen() failed, cannot open template file
367 NGP_EOF                   end of file encountered and not expected
368 NGP_BAD_ARG               bad arguments passed. Usually means
                                internal parser error. Should not happen
369 NGP_TOKEN_NOT_EXPECT      token not expected here

401  error attempting to convert an integer to a formatted character string
402  error attempting to convert a real value to a formatted character string
403  cannot convert a quoted string keyword to an integer
404  attempted to read a non-logical keyword value as a logical value
405  cannot convert a quoted string keyword to a real value
406  cannot convert a quoted string keyword to a double precision value
407  error attempting to read character string as an integer
408  error attempting to read character string as a real value
409  error attempting to read character string as a double precision value
410  bad keyword datatype code
411  illegal number of decimal places while formatting floating point value
412  numerical overflow during implicit datatype conversion
413  error compressing image
414  error uncompressing image
420  error in date or time conversion

431  syntax error in parser expression
432  expression did not evaluate to desired type
433  vector result too large to return in array
434  data parser failed not sent an out column
435  bad data encounter while parsing column
436  parse error: output file not of proper type

501  celestial angle too large for projection
502  bad celestial coordinate or pixel value
503  error in celestial coordinate calculation
504  unsupported type of celestial projection
505  required celestial coordinate keywords not found
506  approximate wcs keyword values were returned
```