CFITSIO User's Reference Guide

An Interface to FITS Format Files
for C Programmers

Version 3.3

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

April 2016

Contents

Chapter 1

Introduction

1.1 A Brief Overview

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 is written in ANSI C and provides a powerful yet simple interface for accessing FITS files which will run on most commonly used computers and workstations. CFITSIO supports all the features described in the official 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 CFITSIO routines 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.

CFITSIO also contains a set of Fortran callable wrapper routines which allow Fortran programs to call the CFITSIO routines. See the companion "FITSIO User's Guide" for the definition of the Fortran subroutine calling sequences. These wrappers replace the older Fortran FITSIO library which is no longer supported.

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.

1.2 Sources of FITS Software and Information

The latest version of the CFITSIO source code, documentation, and example programs are available on the 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 FTOOLS Help Desk at the HEASARC:

```
http://heasarc.gsfc.nasa.gov/cgi-bin/ftoolshelp
```

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 'FITS Standard' gives the authoritative definition of the FITS data format. Other documents available at that Web site provide additional historical background and practical advice on using FITS files.

The HEASARC also provides a very sophisticated FITS file analysis program called 'Fv' which can be used to display and edit the contents of any FITS file as well as construct new FITS files from scratch. Fv is freely available for most Unix platforms, Mac PCs, and Windows PCs. CFITSIO users may also be interested in the FTOOLS package of programs that can be used to manipulate and analyze FITS format files. Fv and FTOOLS are available from their respective Web sites at:

```
http://fv.gsfc.nasa.gov
http://heasarc.gsfc.nasa.gov/ftools
```

1.3 Acknowledgments

The development of the many 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. Also, in 2009, Bruce O'Neel was the key developer of the thread-safe version of CFITSIO.

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.

Giuliano Taffoni and Andrea Barisani, at INAF, University of Trieste, Italy, implemented the I/O driver routines for accessing FITS files on the computational grids using the gridftp protocol.

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.

The new compressed image data format (where the image is tiled and the compressed byte stream from each tile is stored in a binary table) was implemented in collaboration with Richard White (STScI), Perry Greenfield (STScI) and Doug Tody (NOAO).

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, Gretchen Green, 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, Richard Mathar, John Mattox, Carsten Meyer, Emi Miyata, Stefan Mochnacki, Mike Noble, Oliver Oberdorf, Clive Page, Arvind Parmar, Jeff Pedelty, Tim Pearson, Philippe Prugniel, Maren Purves, Scott Randall, Chris Rogers, Arnold Rots, Rob Seaman, 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.

1.4 Legal Stuff

Chapter 2

Creating the CFITSIO Library

2.1 Building the Library

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.

CFITSIO is written in ANCI C and should be compatible with most existing C and C++ compilers. Cray supercomputers are currently not supported.

2.1.1 Unix Systems

The CFITSIO library is built on Unix systems by typing:

```
 >  ./configure [--prefix=/target/installation/path] [--enable-reentrant]
                [--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).

All the available configure options can be seen by entering the command

```
> ./configure --help
```

Some of the more useful options are described below:

The –enable-reentrant option will attempt to configure CFITSIO so that it can be used in multi-threaded programs. See the "Using CFITSIO in Multi-threaded Environments" section, below, for more details.

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

The –with-gsiftp-flavour and –with-gsiftp options enable support for the Globus Toolkit gsiftp protocal. See the "Extended File Name Syntax" chapter for more information.

The –with-bzip2 option enables support for reading FITS files that have been externally compressed by the bzip2 algorithm. This requires that the CFITSIO library, and all applications program that use CFITSIO, to be linked to include the libbz2 library.

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 Mac OS X platforms the 'make shared' command works like on other UNIX platforms, but a .dylib file will be created instead of .so. If installed in a nonstandard location, add its location to the DYLD_LIBRARY_PATH environment variable so that the library can be found at run time.

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

By default, a set of Fortran-callable wrapper routines are also built and included in the CFITSIO library. If these wrapper routines are not needed (i.e., the CFITSIO library will not be linked to any Fortran applications which call FITSIO subroutines) then they may be omitted from the build by typing 'make all-nofitsio' instead of simply typing 'make'. This will reduce the size of the CFITSIO library slightly.

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.

2.1.2 VMS

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.

2.1.3 Windows PCs

A precompiled DLL version of CFITSIO (not necessarily the latest version) is available on the CFITSIO web site. The CFITSIO library may also be built from the source code using the CMake build system. See the "README.win" file in the CFITSIO source distribution for more information.

2.1.4 Macintosh PCs

When building on Mac OS-X, users should follow the Unix instructions, above. See the README.MacOS file for instructions on building a Universal Binary that supports both Intel and PowerPC CPUs.

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 test 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 format 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 with:

```
  % 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)

  % 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 boundarys 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.

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; the first four of this programs can be compiled an linked by typing 'make program_name' where 'program_name' is the actual name of the program:

```
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
```

2.3 Linking Programs with CFITSIO

When linking applications software with the CFITSIO library, several system libraries usually need to be specified on the link command line. On 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 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 Using CFITSIO in Multi-threaded Environments

CFITSIO can be used either with the POSIX pthreads interface or the OpenMP interface for multi-threaded parallel programs. When used in a multi-threaded environment, the CFITSIO library *must* be built using the -D_REENTRANT compiler directive. This can be done using the following build commands:

```
>./configure --enable-reentrant
> make
```

A function called fits_is_reentrant is available to test whether or not CFITSIO was compiled with the -D_REENTRANT directive. When this feature is enabled, multiple threads can call any of the CFITSIO routines to simultaneously read or write separate FITS files. Multiple threads can also read data from the same FITS file simultaneously, as long as the file was opened independently by each thread. This relies on the operating system to correctly deal with reading the same file by multiple processes. Different threads should not share the same 'fitsfile' pointer to read an opened FITS file, unless locks are placed around the calls to the CFITSIO reading routines. Different threads should never try to write to the same FITS file.

2.5 Getting Started with CFITSIO

In order to effectively use the CFITSIO library it is recommended that new users begin by reading the "CFITSIO Quick Start Guide". It contains all the basic information needed to write programs that perform most types of operations on FITS files. The set of example FITS utility programs that are available from the CFITSIO web site are also very useful for learning how to use CFITSIO. To learn even more about the capabilities of the CFITSIO library the following steps are recommended:

1. Read the following short 'FITS Primer' chapter for an overview of the structure of FITS files.

2. Review the Programming Guidelines in Chapter 4 to become familiar with the conventions used by the CFITSIO interface.

3. Refer to the cookbook.c, listhead.c, and fitscopy.c programs that are included with this release for examples of routines that perform various common FITS file operations. Type 'make program_name' to compile and link these programs on Unix systems.

4. Write a simple program to read or write a FITS file using the Basic Interface routines described in Chapter 5.

5. Scan through the more specialized routines that are described in the following chapters to become familiar with the functionality that they provide.

2.6 Example Program

The following listing shows an example of how to use the CFITSIO routines in a C program. Refer to the cookbook.c program that is included with the CFITSIO distribution for other example routines.

This program creates a new FITS file, containing a FITS image. An 'EXPOSURE' keyword is written to the header, then the image data are written to the FITS file before closing the FITS file.

```
#include "fitsio.h"  /* required by every program that uses CFITSIO  */
main()
{
    fitsfile *fptr;       /* pointer to the FITS file; defined in fitsio.h */
    int status, ii, jj;
    long  fpixel = 1, naxis = 2, nelements, exposure;
    long naxes[2] = { 300, 200 };   /* image is 300 pixels wide by 200 rows */
    short array[200][300];

    status = 0;         /* initialize status before calling fitsio routines */
    fits_create_file(&fptr, "testfile.fits", &status);   /* create new file */

    /* Create the primary array image (16-bit short integer pixels */
    fits_create_img(fptr, SHORT_IMG, naxis, naxes, &status);

    /* Write a keyword; must pass the ADDRESS of the value */
    exposure = 1500.;
    fits_update_key(fptr, TLONG, "EXPOSURE", &exposure,
         "Total Exposure Time", &status);

    /* Initialize the values in the image with a linear ramp function */
    for (jj = 0; jj < naxes[1]; jj++)
        for (ii = 0; ii < naxes[0]; ii++)
            array[jj][ii] = ii + jj;
```

```
    nelements = naxes[0] * naxes[1];          /* number of pixels to write */

    /* Write the array of integers to the image */
    fits_write_img(fptr, TSHORT, fpixel, nelements, array[0], &status);

    fits_close_file(fptr, &status);            /* close the file */

    fits_report_error(stderr, status);  /* print out any error messages */
    return( status );
}
```

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 FITS Support OFfice, 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 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 data types are supported: Unsigned 8-bit bytes, 16-bit, 32-bit, and 64-bit signed integers, and 32 and 64-bit floating point reals. FITS also has a convention for storing 16 and 32-bit 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, above

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 FITS Standard) so it is better to rely on standard interface software like CFITSIO 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 data type of the array: 8, 16, 32, 64, -32, -64 for unsigned 8–bit byte, 16–bit signed integer, 32–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 data type 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

Programming Guidelines

4.1 CFITSIO Definitions

Any program that uses the CFITSIO interface must include the fitsio.h header file with the statement

```
#include "fitsio.h"
```

This header file contains the prototypes for all the CFITSIO user interface routines as well as the definitions of various constants used in the interface. It also defines a C structure of type 'fitsfile' that is used by CFITSIO to store the relevant parameters that define the format of a particular FITS file. Application programs must define a pointer to this structure for each FITS file that is to be opened. This structure is initialized (i.e., memory is allocated for the structure) when the FITS file is first opened or created with the fits_open_file or fits_create_file routines. This fitsfile pointer is then passed as the first argument to every other CFITSIO routine that operates on the FITS file. Application programs must not directly read or write elements in this fitsfile structure because the definition of the structure may change in future versions of CFITSIO.

A number of symbolic constants are also defined in fitsio.h for the convenience of application programmers. Use of these symbolic constants rather than the actual numeric value will help to make the source code more readable and easier for others to understand.

```
String Lengths, for use when allocating character arrays:

  #define FLEN_FILENAME 1025 /* max length of a filename                  */
  #define FLEN_KEYWORD   72  /* max length of a keyword                   */
  #define FLEN_CARD      81  /* max length of a FITS header card          */
  #define FLEN_VALUE     71  /* max length of a keyword value string      */
  #define FLEN_COMMENT   73  /* max length of a keyword comment string    */
  #define FLEN_ERRMSG    81  /* max length of a CFITSIO error message     */
  #define FLEN_STATUS    31  /* max length of a CFITSIO status text string */

  Note that FLEN_KEYWORD is longer than the nominal 8-character keyword
```

```
  name length because the HIERARCH convention supports longer keyword names.

Access modes when opening a FITS file:

  #define READONLY  0
  #define READWRITE 1

BITPIX data type code values for FITS images:

  #define BYTE_IMG      8  /*  8-bit unsigned integers */
  #define SHORT_IMG    16  /* 16-bit   signed integers */
  #define LONG_IMG     32  /* 32-bit   signed integers */
  #define LONGLONG_IMG 64  /* 64-bit   signed integers */
  #define FLOAT_IMG   -32  /* 32-bit single precision floating point */
  #define DOUBLE_IMG  -64  /* 64-bit double precision floating point */

  The following 4 data type codes are also supported by CFITSIO:
  #define SBYTE_IMG  10   /*  8-bit signed integers, equivalent to */
                          /*  BITPIX = 8, BSCALE = 1, BZERO = -128 */
  #define USHORT_IMG  20  /* 16-bit unsigned integers, equivalent to */
                          /*  BITPIX = 16, BSCALE = 1, BZERO = 32768 */
  #define ULONG_IMG   40  /* 32-bit unsigned integers, equivalent to */
                          /*  BITPIX = 32, BSCALE = 1, BZERO = 2147483648 */

Codes for the data type of binary table columns and/or for the
data type of variables when reading or writing keywords or data:

                              DATATYPE               TFORM CODE
  #define TBIT          1  /*                            'X' */
  #define TBYTE        11  /* 8-bit unsigned byte,       'B' */
  #define TLOGICAL     14  /* logicals (int for keywords     */
                           /*  and char for table cols   'L' */
  #define TSTRING      16  /* ASCII string,              'A' */
  #define TSHORT       21  /* signed short,              'I' */
  #define TLONG        41  /* signed long,                   */
  #define TLONGLONG    81  /* 64-bit long signed integer 'K' */
  #define TFLOAT       42  /* single precision float,    'E' */
  #define TDOUBLE      82  /* double precision float,    'D' */
  #define TCOMPLEX     83  /* complex (pair of floats)   'C' */
  #define TDBLCOMPLEX 163  /* double complex (2 doubles) 'M' */

  The following data type codes are also supported by CFITSIO:
  #define TINT         31  /* int                            */
  #define TSBYTE       12  /* 8-bit signed byte,         'S' */
  #define TUINT        30  /* unsigned int               'V' */
  #define TUSHORT      20  /* unsigned short             'U'  */
```

```
  #define TULONG       40  /* unsigned long                 */

  The following data type code is only for use with fits\_get\_coltype
  #define TINT32BIT    41  /* signed 32-bit int,         'J' */


HDU type code values (value returned when moving to new HDU):

  #define IMAGE_HDU  0  /* Primary Array or IMAGE HDU */
  #define ASCII_TBL  1  /* ASCII  table HDU */
  #define BINARY_TBL 2  /* Binary table HDU */
  #define ANY_HDU   -1  /* matches any type of HDU */

Column name and string matching case-sensitivity:

  #define CASESEN   1   /* do case-sensitive string match */
  #define CASEINSEN 0   /* do case-insensitive string match */

Logical states (if TRUE and FALSE are not already defined):

  #define TRUE 1
  #define FALSE 0

Values to represent undefined floating point numbers:

  #define FLOATNULLVALUE  -9.11912E-36F
  #define DOUBLENULLVALUE -9.1191291391491E-36

Image compression algorithm definitions

  #define RICE_1       11
  #define GZIP_1       21
  #define GZIP_2       22
  #define PLIO_1       31
  #define HCOMPRESS_1  41
  #define NOCOMPRESS  -1

  #define NO_DITHER -1
  #define SUBTRACTIVE_DITHER_1 1
  #define SUBTRACTIVE_DITHER_2 2
```

4.2 Current Header Data Unit (CHDU)

The concept of the Current Header and Data Unit, or CHDU, is fundamental to the use of the CFITSIO library. A simple FITS image may only contain a single Header and Data unit (HDU), but in general FITS files can contain multiple Header Data Units (also known as 'extensions'), concatenated one after the other in the file. The user can specify which HDU should be initially opened at run time by giving the HDU name or number after the root file name. For example, 'myfile.fits[4]' opens the 5th HDU in the file (note that the numbering starts with 0), and 'myfile.fits[EVENTS] opens the HDU with the name 'EVENTS' (as defined by the EXTNAME or HDUNAME keywords). If no HDU is specified then CFITSIO opens the first HDU (the primary array) by default. The CFITSIO routines which read and write data only operate within the opened HDU, Other CFITSIO routines are provided to move to and open any other existing HDU within the FITS file or to append or insert new HDUs in the FITS file.

4.3 Function Names and Variable Datatypes

Most of the CFITSIO routines have both a short name as well as a longer descriptive name. The short name is only 5 or 6 characters long and is similar to the subroutine name in the Fortran-77 version of FITSIO. The longer name is more descriptive and it is recommended that it be used instead of the short name to more clearly document the source code.

Many of the CFITSIO routines come in families which differ only in the data type of the associated parameter(s). The data type of these routines is indicated by the suffix of the routine name. The short routine names have a 1 or 2 character suffix (e.g., 'j' in 'ffpkyj') while the long routine names have a 4 character or longer suffix as shown in the following table:

```
    Long      Short  Data
    Names     Names  Type
    -----     -----  ----
    _bit        x    bit
    _byt        b    unsigned byte
    _sbyt       sb   signed byte
    _sht        i    short integer
    _lng        j    long integer
    _lnglng     jj   8-byte LONGLONG integer (see note below)
    _usht       ui   unsigned short integer
    _ulng       uj   unsigned long integer
    _uint       uk   unsigned int integer
    _int        k    int integer
    _flt        e    real exponential floating point (float)
    _fixflt     f    real fixed-decimal format floating point (float)
    _dbl        d    double precision real floating-point (double)
    _fixdbl     g    double precision fixed-format floating point (double)
    _cmp        c    complex reals (pairs of float values)
    _fixcmp     fc   complex reals, fixed-format floating point
    _dblcmp     m    double precision complex (pairs of double values)
    _fixdblcmp  fm   double precision complex, fixed-format floating point
    _log        l    logical (int)
    _str        s    character string
```

The logical data type corresponds to 'int' for logical keyword values, and 'byte' for logical binary table columns. In other words, the value when writing a logical keyword must be stored in an 'int' variable, and must be stored in a 'char' array when reading or writing to 'L' columns in a binary table. Implicit data type conversion is not supported for logical table columns, but is for keywords, so a logical keyword may be read and cast to any numerical data type; a returned value = 0 indicates false, and any other value = true.

The 'int' data type may be 2 bytes long on some old PC compilers, but otherwise it is nearly always 4 bytes long. Some 64-bit machines, like the Alpha/OSF, define the 'short', 'int', and 'long' integer data types to be 2, 4, and 8 bytes long, respectively.

Because there is no universal C compiler standard for the name of the 8-byte integer datatype, the fitsio.h include file typedef's 'LONGLONG' to be equivalent to an appropriate 8-byte integer data type on each supported platform. For maximum software portability it is recommended that this LONGLONG datatype be used to define 8-byte integer variables rather than using the native data type name on a particular platform. On most 32-bit Unix and Mac OS-X operating systems LONGLONG is equivalent to the intrinsic 'long long' 8-byte integer datatype. On 64-bit systems (which currently includes Alpha OSF/1, 64-bit Sun Solaris, 64-bit SGI MIPS, and 64-bit Itanium and Opteron PC systems), LONGLONG is simply typedef'ed to be equivalent to 'long'. Microsoft Visual C++ Version 6.0 does not define a 'long long' data type, so LONGLONG is typedef'ed to be equivalent to the '__int64' data type on 32-bit windows systems when using Visual C++.

A related issue that affects the portability of software is how to print out the value of a 'LONGLONG' variable with printf. Developers may find it convenient to use the following preprocessing statements in their C programs to handle this in a machine-portable manner:

```
#if defined(_MSC_VER) /* Microsoft Visual C++ */
          printf("%I64d", longlongvalue);

#elif (USE_LL_SUFFIX == 1)
          printf("%lld", longlongvalue);

#else
          printf("%ld", longlongvalue);
#endif
```

Similarly, the name of the C utility routine that converts a character string of digits into a 8-byte integer value is platform dependent:

```
#if defined(_MSC_VER) /* Microsoft Visual C++ */
      /* VC++ 6.0 does not seem to have an 8-byte conversion routine */

#elif (USE_LL_SUFFIX == 1)
          longlongvalue = atoll(*string);

#else
          longlongvalue = atol(*string);
#endif
```

When dealing with the FITS byte data type 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 C compilers define a 'char' variable as signed, so it is important to explicitly declare a numeric char variable as 'unsigned char' to avoid any ambiguity

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' data type 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 complex and double precision complex data types are not directly supported in ANSI C so these data types should be interpreted as pairs of float or double values, respectively, where the first value in each pair is the real part, and the second is the imaginary part.

4.4 Support for Unsigned Integers and Signed Bytes

Although FITS does not directly support unsigned integers as one of its fundamental data types, FITS can still be used to efficiently store unsigned integer data values in images and binary tables. The convention used in FITS files is to store the unsigned integers as signed integers with an associated offset (specified by the BZERO or TZEROn keyword). For example, to store unsigned 16-bit integer values in a FITS image the image would be defined as a signed 16-bit integer (with BITPIX keyword = SHORT_IMG = 16) with the keywords BSCALE = 1.0 and BZERO = 32768. Thus the unsigned values of 0, 32768, and 65535, for example, are physically stored in the FITS image as -32768, 0, and 32767, respectively; CFITSIO automatically adds the BZERO offset to these values when they are read. Similarly, in the case of unsigned 32-bit integers the BITPIX keyword would be equal to LONG_IMG = 32 and BZERO would be equal to 2147483648 (i.e. 2 raised to the 31st power).

The CFITSIO interface routines will efficiently and transparently apply the appropriate offset in these cases so in general application programs do not need to be concerned with how the unsigned values are actually stored in the FITS file. As a convenience for users, CFITSIO has several predefined constants for the value of BITPIX (USHORT_IMG, ULONG_IMG) and for the TFORMn value in the case of binary tables ('U' and 'V') which programmers can use when creating FITS files containing unsigned integer values. The following code fragment illustrates how to write a FITS 1-D primary array of unsigned 16-bit integers:

```
unsigned short uarray[100];
int naxis, status;
long naxes[10], group, firstelem, nelements;
 ...
status = 0;
naxis = 1;
naxes[0] = 100;
fits_create_img(fptr, USHORT_IMG, naxis, naxes, &status);

firstelem = 1;
```

```
    nelements = 100;
    fits_write_img(fptr, TUSHORT, firstelem, nelements,
                        uarray, &status);
     ...
```

In the above example, the 2nd parameter in fits_create_img tells CFITSIO to write the header keywords appropriate for an array of 16-bit unsigned integers (i.e., BITPIX = 16 and BZERO = 32768). Then the fits_write_img routine writes the array of unsigned short integers (uarray) into the primary array of the FITS file. Similarly, a 32-bit unsigned integer image may be created by setting the second parameter in fits_create_img equal to 'ULONG_IMG' and by calling the fits_write_img routine with the second parameter = TULONG to write the array of unsigned long image pixel values.

An analogous set of routines are available for reading or writing unsigned integer values and signed byte values in a FITS binary table extension. When specifying the TFORMn keyword value which defines the format of a column, CFITSIO recognized 3 additional data type codes besides those already defined in the FITS standard: 'U' meaning a 16-bit unsigned integer column, 'V' for a 32-bit unsigned integer column, and 'S' for a signed byte column. These non-standard data type codes are not actually written into the FITS file but instead are just used internally within CFITSIO. The following code fragment illustrates how to use these features:

```
    unsigned short uarray[100];
    unsigned int  varray[100];

    int colnum, tfields, status;
    long nrows, firstrow, firstelem, nelements, pcount;

    char extname[] = "Test_table";           /* extension name */

    /* define the name, data type, and physical units for the 2 columns */
    char *ttype[] = { "Col_1", "Col_2", "Col_3" };
    char *tform[] = { "1U",      "1V",    "1S"};  /* special CFITSIO codes */
    char *tunit[] = { " ",        " ",    " " };
     ...

         /* write the header keywords */
    status  = 0;
    nrows   = 1;
    tfields = 3
    pcount  = 0;
    fits_create_tbl(fptr, BINARY_TBL, nrows, tfields, ttype, tform,
              tunit, extname, &status);

         /* write the unsigned shorts to the 1st column */
    colnum    = 1;
    firstrow  = 1;
    firstelem = 1;
```

```
nelements = 100;
fits_write_col(fptr, TUSHORT, colnum, firstrow, firstelem,
        nelements, uarray, &status);

     /* now write the unsigned longs to the 2nd column */
colnum    = 2;
fits_write_col(fptr, TUINT, colnum, firstrow, firstelem,
        nelements, varray, &status);
 ...
```

Note that the non-standard TFORM values for the 3 columns, 'U' and 'V', tell CFITSIO to write the keywords appropriate for unsigned 16-bit and unsigned 32-bit integers, respectively (i.e., TFORMn = '1I' and TZEROn = 32768 for unsigned 16-bit integers, and TFORMn = '1J' and TZEROn = 2147483648 for unsigned 32-bit integers). The 'S' TFORMn value tells CFITSIO to write the keywords appropriate for a signed 8-bit byte column with TFORMn = '1B' and TZEROn = -128. The calls to fits_write_col then write the arrays of unsigned integer values to the columns.

4.5 Dealing with Character Strings

The character string values in a FITS header or in an ASCII column in a FITS table extension are generally padded out with non-significant space characters (ASCII 32) to fill up the header record or the column width. When reading a FITS string value, the CFITSIO routines will strip off these non-significant trailing spaces and will return a null-terminated string value containing only the significant characters. Leading spaces in a FITS string are considered significant. If the string contains all blanks, then CFITSIO will return a single blank character, i.e, the first blank is considered to be significant, since it distinguishes the string from a null or undefined string, but the remaining trailing spaces are not significant.

Similarly, when writing string values to a FITS file the CFITSIO routines expect to get a null-terminated string as input; CFITSIO will pad the string with blanks if necessary when writing it to the FITS file.

When calling CFITSIO routines that return a character string it is vital that the size of the char array be large enough to hold the entire string of characters, otherwise CFITSIO will overwrite whatever memory locations follow the char array, possibly causing the program to execute incorrectly. This type of error can be difficult to debug, so programmers should always ensure that the char arrays are allocated enough space to hold the longest possible string, **including** the terminating NULL character. The fitsio.h file contains the following defined constants which programmers are strongly encouraged to use whenever they are allocating space for char arrays:

```
#define FLEN_FILENAME 1025 /* max length of a filename */
#define FLEN_KEYWORD   72  /* max length of a keyword  */
#define FLEN_CARD      81  /* length of a FITS header card */
#define FLEN_VALUE     71  /* max length of a keyword value string */
#define FLEN_COMMENT   73  /* max length of a keyword comment string */
#define FLEN_ERRMSG    81  /* max length of a CFITSIO error message */
```

```
#define FLEN_STATUS    31  /* max length of a CFITSIO status text string */
```

For example, when declaring a char array to hold the value string of FITS keyword, use the following statement:

```
    char value[FLEN_VALUE];
```

Note that FLEN_KEYWORD is longer than needed for the nominal 8-character keyword name because the HIERARCH convention supports longer keyword names.

4.6 Implicit Data Type Conversion

The CFITSIO 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 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 fits_set_bscale and fits_set_tscale routines (described in the 'Advanced' chapter) 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) 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 immediate effect. One may also use the fits_set_bscale and fits_set_tscale routines 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 data types to the FITS file.

4.8 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 FITS standard or the FITS User's Guide for a list of these values). The CFITSIO routines 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 CFITSIO. This can be done by calling the fits_read_img or fits_read_col routines while specifying 0.0 as the value of the NULLVAL parameter. This will force CFITSIO to simply pass the IEEE values through to the application program without any modification. This is not fully supported 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.

4.9 Error Status Values and the Error Message Stack

Nearly all the CFITSIO routines return an error status value in 2 ways: as the value of the last parameter in the function call, and as the returned value of the function itself. This provides some flexibility in the way programmers can test if an error occurred, as illustrated in the following 2 code fragments:

```
    if ( fits_write_record(fptr, card, &status) )
         printf(" Error occurred while writing keyword.");

or,

    fits_write_record(fptr, card, &status);
    if ( status )
         printf(" Error occurred while writing keyword.");
```

A listing of all the CFITSIO status code values is given at the end of this document. Programmers are encouraged to use the symbolic mnemonics (defined in fitsio.h) rather than the actual integer status values to improve the readability of their code.

The CFITSIO library uses an 'inherited status' convention for the status parameter which means that if a routine is called with a positive input value of the status parameter as input, then the routine will exit immediately without changing the value of the status parameter. Thus, if one passes the status value returned from each CFITSIO routine as input to the next CFITSIO routine, then whenever an error is detected all further CFITSIO 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 CFITSIO routine call. If a program contains a sequence of several CFITSIO 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 routine originally returned the error status.

CFITSIO 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 CFITSIO error. The function fits_report_error will print out the entire error message stack, or alternatively one may call fits_read_errmsg to get the error messages one at a time.

4.10 Variable-Length Arrays in Binary Tables

CFITSIO 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)' where 't' is the data type code (e.g., I, J, E, D, etc.) and 'len' is an integer specifying the maximum length of the vector in the table. (CFITSIO also supports the experimental 'Q' datatype, which is identical to the 'P' type except that is supports is a 64-bit address space and hence much larger data structures). 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 CFITSIO 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 that read and write data in an ordinary fixed length binary table extension are also used for variable length fields, however, the routine 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 by the PCOUNT keyword in the FITS header. When creating a new binary table, the initial value of PCOUNT should usually be set to zero. CFITSIO 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 row of data 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 fits_write_col. The total length of the array is given by nelements + firstelem - 1. Additional elements cannot be appended to an existing vector at a later time since any attempt to do so will simply overwrite all the previously written data and the new data will be written to a new area of the heap. The fits_compress_heap routine is provided to compress the heap and recover any unused space. To avoid having to deal with this issue, it is recommended that rows in a variable length field should only be written once. An exception to this general rule occurs when setting elements of an array as undefined. It is allowed to first write a dummy value into the array with fits_write_col, and then call fits_write_col_nul to flag the desired elements as undefined. 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 can be written. The 'firstelem' and 'nelements' parameter values in the fits_write_col routine are ignored and the number of characters to write is simply determined by the length of the input null-terminated character string.

The fits_write_descript routine 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 fits_write_descript to write the same descriptor values into the other rows; 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 fits_read_descript routine.

4.11 Multiple Access to the Same FITS File

CFITSIO supports simultaneous read and write access to different HDUs in the same FITS file in some circumstances, as described below:

- Multi-threaded programs

  When CFITSIO is compiled with the -D_REENTRANT directive (as can be tested with the fits_is_reentrant function) different threads can call any of the CFITSIO routines to simultaneously read or write separate FITS files. Multiple threads can also read data from the same FITS file simultaneously, as long as the file was opened independently by each thread. This relies on the operating system to correctly deal with reading the same file by multiple processes. Different threads should not share the same 'fitsfile' pointer to read an opened FITS file, unless locks are placed around the calls to the CFITSIO reading routines. Different threads should never try to write to the same FITS file.

- Multiple read access to the same FITS file within a single program/thread

  A single process may open the same FITS file with READONLY access multiple times, and thus create multiple 'fitsfile*' pointers to that same file within CFITSIO. This relies on the operating system's ability to open a single file multiple times and correctly manage the subsequent read requests directed to the different C 'file*' pointers, which actually all point to the same file. CFITSIO simply executes the read requests to the differnet 'fitsfile*' pointers the same as if they were physically different files.

- Multiple write access to the same FITS file within a single program/thread

  CFITSIO supports opening the same FITS file multiple times with WRITE access, but it only physically opens the file (at the operating system level) once, on the first call to fits_open_file. If fits_open_file is subsequently called to open the same file again, CFITSIO will recognize that the file is already open, and will return a new 'fitsfile*' pointer that logically points to the first 'fitsfile*' pointer, without actually opening the file a second time. The application program can then treat the 2 'fitsfile*' pointers as if they point to different files, and can seemingly move to and write data to 2 different HDUs within the same file. However, each time the application program switches which 'fitsfile*' pointer it is writing to, CFITSIO will flush any internal buffers that contain data written to the first 'fitsfile*' pointer, then move to the HDU that the other 'fitsfile*' pointer is writing to. Obviously, this may add a significant amount of computational overhead if the application program uses this feature to frequently switch back and forth between writing to 2 (or more) HDUs in the same file, so this capability should be used judiciously.

  Note that CFITSIO will not allow a FITS file to be opened a second time with READWRITE access if it was opened previously with READONLY access.

4.12 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.

4.13 CFITSIO Size Limitations

CFITSIO places very few restrictions on the size of FITS files that it reads or writes. There are a few limits, however, that 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. The current default value is 1000, but this may be increased if necessary. Note that CFITSIO allocates NIOBUF * 2880 bytes of I/O buffer space for each file that is opened. The default value of NIOBUF is 40 (defined in fitsio.h), so this amounts to more than 115K of memory for each opened file (or 115 MB for 1000 opened files). Note that the underlying operating system, may have a lower limit on the number of files that can be opened simultaneously.

2. It used to be common for computer systems to only support disk files up to 2**31 bytes = 2.1 GB in size, but most systems now support larger files. CFITSIO can optionally read and write these so-called 'large files' that are greater than 2.1 GB on platforms where they are supported, but this usually requires that special compiler option flags be specified to turn on this option. On linux and solaris systems the compiler flags are '-D_LARGEFILE_SOURCE' and '-D_FILE_OFFSET_BITS=64'. These flags may also work on other platforms but this has not been tested. Starting with version 3.0 of CFITSIO, the default Makefile that is distributed with CFITSIO will include these 2 compiler flags when building on Solaris and Linux PC systems. Users on other platforms will need to add these compiler flags manually if they want to support large files. In most cases it appears that it is not necessary to include these compiler flags when compiling application code that call the CFITSIO library routines.

When CFITSIO is built with large file support (e.g., on Solaris and Linux PC system by default) then it can read and write FITS data files on disk that have any of these conditions:

- FITS files larger than 2.1 GB in size
- FITS images containing greater than 2.1 G pixels
- FITS images that have one dimension with more than 2.1 G pixels (as given by one of the NAXISn keyword)
- FITS tables containing more than 2.1E09 rows (given by the NAXIS2 keyword), or with rows that are more than 2.1 GB wide (given by the NAXIS1 keyword)
- FITS binary tables with a variable-length array heap that is larger than 2.1 GB (given by the PCOUNT keyword)

The current maximum FITS file size supported by CFITSIO is about 6 terabytes (containing 2**31 FITS blocks, each 2880 bytes in size). Currently, support for large files in CFITSIO has been tested on the Linux, Solaris, and IBM AIX operating systems.

Note that when writing application programs that are intended to support large files it is important to use 64-bit integer variables to store quantities such as the dimensions of images, or the number of rows in a table. These programs must also call the special versions of some of the CFITSIO routines that have been adapted to support 64-bit integers. The names of these routines end in 'll' ('el' 'el') to distinguish them from the 32-bit integer version (e.g., fits_get_num_rowsll).

Chapter 5

Basic CFITSIO Interface Routines

This chapter describes the basic routines in the CFITSIO user interface that provide all the functions normally needed to read and write most FITS files. It is recommended that these routines be used for most applications and that the more advanced routines described in the next chapter only be used in special circumstances when necessary.

The following conventions are used in this chapter in the description of each function:

1. Most functions have 2 names: a long descriptive name and a short concise name. Both names are listed on the first line of the following descriptions, separated by a slash (/) character. Programmers may use either name in their programs but the long names are recommended to help document the code and make it easier to read.

2. A right arrow symbol (>) is used in the function descriptions to separate the input parameters from the output parameters in the definition of each routine. This symbol is not actually part of the C calling sequence.

3. The function parameters are defined in more detail in the alphabetical listing in Appendix B.

4. The first argument in almost all the functions is a pointer to a structure of type 'fitsfile'. Memory for this structure is allocated by CFITSIO when the FITS file is first opened or created and is freed when the FITS file is closed.

5. The last argument in almost all the functions is the error status parameter. It must be equal to 0 on input, otherwise the function will immediately exit without doing anything. A non-zero output value indicates that an error occurred in the function. In most cases the status value is also returned as the value of the function itself.

5.1 CFITSIO Error Status Routines

1 Return a descriptive text string (30 char max.) corresponding to a CFITSIO error status code.

```
void fits_get_errstatus / ffgerr (int status, > char *err_text)
```

2 Return the top (oldest) 80-character error message from the internal CFITSIO stack of error messages and shift any remaining messages on the stack up one level. Call this routine repeatedly to get each message in sequence. The function returns a value = 0 and a null error message when the error stack is empty.

```
int fits_read_errmsg / ffgmsg (char *err_msg)
```

3 Print out the error message corresponding to the input status value and all the error messages on the CFITSIO stack to the specified file stream (normally to stdout or stderr). If the input status value = 0 then this routine does nothing.

```
void fits_report_error / ffrprt (FILE *stream, status)
```

4 The fits_write_errmark routine puts an invisible marker on the CFITSIO error stack. The fits_clear_errmark 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. The fits_clear_errmsg routine simply clears all the messages (and marks) from the stack. These routines are called without any arguments.

```
void fits_write_errmark / ffpmrk (void)
void fits_clear_errmark / ffcmrk (void)
void fits_clear_errmsg / ffcmsg (void)
```

5.2 FITS File Access Routines

1 Open an existing data file.

```
int fits_open_file / ffopen
    (fitsfile **fptr, char *filename, int iomode, > int *status)

int fits_open_diskfile / ffdkopen
    (fitsfile **fptr, char *filename, int iomode, > int *status)

int fits_open_data / ffdopn
    (fitsfile **fptr, char *filename, int iomode, > int *status)

int fits_open_table / fftopn
    (fitsfile **fptr, char *filename, int iomode, > int *status)

int fits_open_image / ffiopn
    (fitsfile **fptr, char *filename, int iomode, > int *status)

int fits_open_extlist / ffeopn
    (fitsfile **fptr, char *filename, int iomode, char *extlist,
    >  int *hdutype, int *status)
```

The iomode parameter determines the read/write access allowed in the file and can have values of READONLY (0) or READWRITE (1). The filename parameter gives the name of the file to be opened, followed by an optional argument giving the name or index number of the extension within the FITS file that should be moved to and opened (e.g., `myfile.fits+3` or `myfile.fits[3]` moves to the 3rd extension within the file, and `myfile.fits[events]` moves to the extension with the keyword EXTNAME = 'EVENTS').

The fits_open_diskfile routine is similar to the fits_open_file routine except that it does not support the extended filename syntax in the input file name. This routine simply tries to open the specified input file on magnetic disk. This routine is mainly for use in cases where the filename (or directory path) contains square or curly bracket characters that would confuse the extended filename parser.

The fits_open_data routine is similar to the fits_open_file routine except that it will move to the first HDU containing significant data, if a HDU name or number to open was not explicitly specified as part of the filename. In this case, it will look for the first IMAGE HDU with NAXIS greater than 0, or the first table that does not contain the strings 'GTI' (Good Time Interval extension) or 'OBSTABLE' in the EXTNAME keyword value.

The fits_open_table and fits_open_image routines are similar to fits_open_data except they will move to the first significant table HDU or image HDU in the file, respectively, if a HDU name or number is not specified as part of the filename.

The fits_open_extname routine opens the file and attempts to move to a 'useful' HDU. If after opening the file CFITSIO is pointing to null primary array, then CFITSIO will attempt to move to the first extension that has an EXTNAME or HDUNAME keyword value that matches one of the names in the input extlist space-delimited list of names. If that fails, then CFITSIO simply moves to the 2nd HDU in the file.

IRAF images (.imh format files) and raw binary data arrays may also be opened with READONLY access. CFITSIO will automatically test if the input file is an IRAF image, and if, so will convert it on the fly into a virtual FITS image before it is opened by the application program. If the input file is a raw binary data array of numbers, then the data type and dimensions of the array must be specified in square brackets following the name of the file (e.g. 'rawfile.dat[i512,512]' opens a 512 x 512 short integer image). See the 'Extended File Name Syntax' chapter for more details on how to specify the raw file name. The raw file is converted on the fly into a virtual FITS image in memory that is then opened by the application program with READONLY access.

Programs can read the input file from the 'stdin' file stream if a dash character ('-') is given as the filename. Files can also be opened over the network using FTP or HTTP protocols by supplying the appropriate URL as the filename.

The input file can be modified in various ways to create a virtual file (usually stored in memory) that is then opened by the application program by supplying a filtering or binning specifier in square brackets following the filename. Some of the more common filtering methods are illustrated in the following paragraphs, but users should refer to the 'Extended File Name Syntax' chapter for a complete description of the full file filtering syntax.

When opening an image, a rectangular subset of the physical image may be opened by listing the first and last pixel in each dimension (and optional pixel skipping factor):

```
myimage.fits[101:200,301:400]
```

will create and open a 100x100 pixel virtual image of that section of the physical image, and `myimage.fits[*,-*]` opens a virtual image that is the same size as the physical image but has been flipped in the vertical direction.

When opening a table, the filtering syntax can be used to add or delete columns or keywords in the virtual table: `myfile.fits[events][col !time; PI = PHA*1.2]` opens a virtual table in which the TIME column has been deleted and a new PI column has been added with a value 1.2 times that of the PHA column. Similarly, one can filter a table to keep only those rows that satisfy a selection criterion: `myfile.fits[events][pha > 50]` creates and opens a virtual table containing only those rows with a PHA value greater than 50. A large number of boolean and mathematical operators can be used in the selection expression. One can also filter table rows using 'Good Time Interval' extensions, and spatial region filters as in `myfile.fits[events][gtifilter()]` and `myfile.fits[events][regfilter( "stars.rng")]`.

Finally, table columns may be binned or histogrammed to generate a virtual image. For example, `myfile.fits[events][bin (X,Y)=4]` will result in a 2-dimensional image calculated by binning the X and Y columns in the event table with a bin size of 4 in each dimension. The TLMINn and TLMAXn keywords will be used by default to determine the range of the image.

A single program can open the same FITS file more than once and then treat the resulting fitsfile pointers as though they were completely independent FITS files. Using this facility, a program can open a FITS file twice, move to 2 different extensions within the file, and then read and write data in those extensions in any order.

2 Create and open a new empty output FITS file.

```
int fits_create_file / ffinit
    (fitsfile **fptr, char *filename, > int *status)

int fits_create_diskfile / ffdkinit
    (fitsfile **fptr, char *filename, > int *status)
```

An error will be returned if the specified file already exists, unless the filename is prefixed with an exclamation point (!). In that case CFITSIO will overwrite (delete) any existing file with the same name. Note that the exclamation point is a special UNIX character so if it is used on the command line it must be preceded by a backslash to force the UNIX shell to accept the character as part of the filename.

The output file will be written to the 'stdout' file stream if a dash character ('-') or the string 'stdout' is given as the filename. Similarly, '-.gz' or 'stdout.gz' will cause the file to be gzip compressed before it is written out to the stdout stream.

Optionally, the name of a template file that is used to define the structure of the new file may be specified in parentheses following the output file name. The template file may be another FITS file, in which case the new file, at the time it is opened, will be an exact copy of the template file except that the data structures (images and tables) will be filled with zeros. Alternatively, the template file may be an ASCII format text file containing directives that define the keywords to be created in each HDU of the file. See the 'Extended File Name Syntax' section for a complete description of the template file syntax.

The fits_create_diskfile routine is similar to the fits_create_file routine except that it does not support the extended filename syntax in the input file name. This routine simply tries to create the specified file on magnetic disk. This routine is mainly for use in cases where the filename (or directory path) contains square or curly bracket characters that would confuse the extended filename parser.

3 Close a previously opened FITS file. The first routine simply closes the file, whereas the second one also DELETES the file, which can be useful in cases where a FITS file has been partially created, but then an error occurs which prevents it from being completed. Note that these routines behave differently than most other CFITSIO routines if the input value of the 'status' parameter is not zero: Instead of simply returning to the calling program without doing anything, these routines effectively ignore the input status value and still attempt to close or delete the file.

```
int fits_close_file / ffclos (fitsfile *fptr, > int *status)

int fits_delete_file / ffdelt (fitsfile *fptr, > int *status)
```

4 Return the name, I/O mode (READONLY or READWRITE), and/or the file type (e.g. 'file://', 'ftp://') of the opened FITS file.

```
int fits_file_name / ffflnm (fitsfile *fptr, > char *filename, int *status)

int fits_file_mode / ffflmd (fitsfile *fptr, > int *iomode, int *status)

int fits_url_type / ffurlt (fitsfile *fptr, > char *urltype, int *status)
```

5.3 HDU Access Routines

The following functions perform operations on Header-Data Units (HDUs) as a whole.

1 Move to a different HDU in the file. The first routine moves to a specified absolute HDU number (starting with 1 for the primary array) in the FITS file, and the second routine moves a relative number HDUs forward or backward from the current HDU. A null pointer may be given for the hdutype parameter if it's value is not needed. The third routine moves to the (first) HDU which has the specified extension type and EXTNAME and EXTVER keyword values (or HDUNAME and HDUVER keywords). The hdutype parameter may have a value of IMAGE_HDU, ASCII_TBL, BINARY_TBL, or ANY_HDU 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 will be returned.

```
int fits_movabs_hdu / ffmahd
    (fitsfile *fptr, int hdunum, > int *hdutype, int *status)

int fits_movrel_hdu / ffmrhd
    (fitsfile *fptr, int nmove, > int *hdutype, int *status)

int fits_movnam_hdu / ffmnhd
    (fitsfile *fptr, int hdutype, char *extname, int extver, > int *status)
```

2 Return the total number of HDUs in the FITS file. This returns the number of completely defined HDUs in the file. If a new HDU has just been added to the FITS file, then that last HDU will only be counted if it has been closed, or if data has been written to the HDU. The current HDU remains unchanged by this routine.

```
int fits_get_num_hdus / ffthdu
    (fitsfile *fptr, > int *hdunum, int *status)
```

3 Return the number of the current HDU (CHDU) in the FITS file (where the primary array = 1). This function returns the HDU number rather than a status value.

```
int fits_get_hdu_num / ffghdn
    (fitsfile *fptr, > int *hdunum)
```

4 Return the type of the current HDU in the FITS file. The possible values for hdutype are: IMAGE_HDU, ASCII_TBL, or BINARY_TBL.

```
int fits_get_hdu_type / ffghdt
    (fitsfile *fptr, > int *hdutype, int *status)
```

5 Copy all or part of the HDUs in the FITS file associated with infptr and append them to the end of the FITS file associated with outfptr. If 'previous' is true (not 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. Thus, if all 3 parameters are true, then the entire input file will be copied. On exit, the current HDU in the input file will be unchanged, and the last HDU in the output file will be the current HDU.

```
int fits_copy_file / ffcpfl
    (fitsfile *infptr, fitsfile *outfptr, int previous, int current,
        int following, > int *status)
```

6 Copy the current HDU from the FITS file associated with infptr and append it to the end of the FITS file associated with outfptr. Space may be reserved for MOREKEYS additional keywords in the output header.

```
int fits_copy_hdu / ffcopy
    (fitsfile *infptr, fitsfile *outfptr, int morekeys, > int *status)
```

7 Write the current HDU in the input FITS file to the output FILE stream (e.g., to stdout).

```
int fits_write_hdu / ffwrhdu
    (fitsfile *infptr, FILE *stream, > int *status)
```

8 Copy the header (and not the data) from the CHDU associated with infptr to the CHDU associated with outfptr. 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. An empty output data unit will be created with all values initially = 0).

```
int fits_copy_header / ffcphd
    (fitsfile *infptr, fitsfile *outfptr, > int *status)
```

9 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 returns the type of the new CHDU. A null pointer may be given for hdutype if the returned value is not needed.

```
int fits_delete_hdu / ffdhdu
    (fitsfile *fptr, > int *hdutype, int *status)
```

5.4 Header Keyword Read/Write Routines

These routines read or write keywords in the Current Header Unit (CHU). 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). When a wild card is used the routine will only search for a match from the current header position to the end of the header and 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. The fits_read_record routine may be used to set the starting position when doing wild card searches. A status value of KEY_NO_EXIST is returned if the specified keyword to be read is not found in the header.

5.4.1 Keyword Reading Routines

1 Return the number of existing keywords (not counting the END keyword) and the amount of space currently available for more keywords. It returns morekeys = -1 if the header has not yet been closed. Note that CFITSIO will dynamically add space if required when writing new keywords to a header so in practice there is no limit to the number of keywords that can be added to a header. A null pointer may be entered for the morekeys parameter if it's value is not needed.

```
int fits_get_hdrspace / ffghsp
    (fitsfile *fptr, > int *keysexist, int *morekeys, int *status)
```

2 Return the specified keyword. In the first routine, the datatype parameter specifies the desired returned data type of the keyword value and can have one of the following symbolic constant values: TSTRING, TLOGICAL (== int), TBYTE, TSHORT, TUSHORT, TINT, TUINT, TLONG, TULONG, TLONGLONG, TFLOAT, TDOUBLE, TCOMPLEX, and TDBLCOMPLEX. Within the context of this routine, TSTRING corresponds to a 'char*' data type, i.e., a pointer to a character array. Data type conversion will be performed for numeric values if the keyword value does not have the same data type. If the value of the keyword is undefined (i.e., the value field is blank) then an error status = VALUE_UNDEFINED will be returned.

The second routine returns the keyword value as a character string (a literal copy of what is in the value field) regardless of the intrinsic data type of the keyword. The third routine returns the entire 80-character header record of the keyword, with any trailing blank characters stripped off. The fourth routine returns the (next) header record that contains the literal string of characters specified by the 'string' argument.

If a NULL comment pointer is supplied then the comment string will not be returned.

```
int fits_read_key / ffgky
    (fitsfile *fptr, int datatype, char *keyname, > DTYPE *value,
     char *comment, int *status)

int fits_read_keyword / ffgkey
    (fitsfile *fptr, char *keyname, > char *value, char *comment,
     int *status)

int fits_read_card / ffgcrd
    (fitsfile *fptr, char *keyname, > char *card, int *status)

int fits_read_str / ffgstr
    (fitsfile *fptr, char *string, > char *card, int *status)
```

3 Read a string-valued keyword and return the string length, the value string, and/or the comment field. The first routine, ffgksl, simply returns the length of the character string value of the specified keyword. The second routine, ffgsky, also returns up to maxchar characters of the keyword value string, starting with the firstchar character, and the keyword comment string (unless the input value of comm = NULL). The valuelen argument returns the total length of the keyword value string regardless of how much of the string is actually returned (which depends on the value of the firstchar and maxchar arguments). Note that the value character string argument must be allocated large enough to also hold the null terminator at the end of the returned string. These routines support string keywords that use the CONTINUE convention to continue long string values over multiple FITS header records. Normally, string-valued keywords have a maximum length of 68 characters, however, CONTINUE'd string keywords may be arbitrarily long.

```
int fits_get_key_strlen / ffgksl
    (fitsfile *fptr, const char *keyname, int *length, int *status);

int fits_read_string_key / ffgsky
     (fitsfile *fptr, const char *keyname, int firstchar, int maxchar,
     char *value, int *valuelen, char *comm, int *status);
```

4 Return the nth header record in the CHU. The first keyword in the header is at keynum = 1; if keynum = 0 then these routines simply reset the internal CFITSIO pointer to the beginning of the header so that subsequent keyword operations will start at the top of the header (e.g., prior to searching for keywords using wild cards in the keyword name). The first routine returns the entire 80-character header record (with trailing blanks truncated), while the second routine parses the record and returns the name, value, and comment fields as separate (blank truncated) character strings. If a NULL comment pointer is given on input, then the comment string will not be returned.

```
int fits_read_record / ffgrec
    (fitsfile *fptr, int keynum, > char *card, int *status)

int fits_read_keyn / ffgkyn
    (fitsfile *fptr, int keynum, > char *keyname, char *value,
     char *comment, int *status)
```

5 Return 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 ffgrec routine. Note that nexc may be set = 0 if there are no keywords to be excluded. This routine returns status = KEY_NO_EXIST if a matching keyword is not found.

```
int fits_find_nextkey / ffgnxk
    (fitsfile *fptr, char **inclist, int ninc, char **exclist,
     int nexc, > char *card, int  *status)
```

6 Return the physical units string from 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 null string is returned if no units are defined for the keyword.

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

```
int fits_read_key_unit / ffgunt
    (fitsfile *fptr, char *keyname, > char *unit, int *status)
```

7 Concatenate the header keywords in the CHDU into a single long string of characters. This provides a convenient way of passing all or part of the header information in a FITS HDU to other subroutines. Each 80-character fixed-length keyword record is appended to the output character string, in order, with no intervening separator or terminating characters. The last header record is terminated with a NULL character. These routine allocates memory for the returned character array, so the calling program must free the memory when finished. The cleanest way to do this is to call the fits_free_memory routine.

There are 2 related routines: fits_hdr2str simply concatenates all the existing keywords in the header; fits_convert_hdr2str is similar, except that if the CHDU is a tile compressed image (stored in a binary table) then it will first convert that header back to that of the corresponding normal FITS image before concatenating the keywords.

Selected keywords may be excluded from the returned character string. If the second parameter (nocomments) is TRUE (nonzero) then any COMMENT, HISTORY, or blank keywords in the header will not be copied to the output string.

The 'exclist' parameter may be used to supply a list of keywords that are to be excluded from the output character string. Wild card characters (*, ?, and #) may be used in the excluded keyword names. If no additional keywords are to be excluded, then set nexc = 0 and specify NULL for the the **exclist parameter.

```
int fits_hdr2str / ffhdr2str
    (fitsfile *fptr, int nocomments, char **exclist, int nexc,
    > char **header, int *nkeys, int *status)

int fits_convert_hdr2str / ffcnvthdr2str
    (fitsfile *fptr, int nocomments, char **exclist, int nexc,
    > char **header, int *nkeys, int *status)

int fits_free_memory / fffree
    (char *header, > int *status);
```

5.4.2 Keyword Writing Routines

1 Write a keyword of the appropriate data type into the CHU. The first routine simply appends a new keyword whereas the second routine will update the value and comment fields of the keyword if it already exists, otherwise it appends a new keyword. Note that the address to the value, and not the value itself, must be entered. The datatype parameter specifies the data type of the keyword value with one of the following values: TSTRING, TLOGICAL (== int), TBYTE, TSHORT, TUSHORT, TINT, TUINT, TLONG, TLONGLONG, TULONG, TFLOAT, TDOUBLE. Within the context of this routine, TSTRING corresponds to a 'char*' data type, i.e., a pointer to a character array. A null pointer may be entered for the comment parameter in which case the keyword comment field will be unmodified or left blank.

```
int fits_write_key / ffpky
    (fitsfile *fptr, int datatype, char *keyname, DTYPE *value,
        char *comment, > int *status)

int fits_update_key / ffuky
    (fitsfile *fptr, int datatype, char *keyname, DTYPE *value,
        char *comment, > int *status)
```

2 Write a keyword with a null or undefined value (i.e., the value field in the keyword is left blank). The first routine simply appends a new keyword whereas the second routine will update the value and comment fields of the keyword if it already exists, otherwise it appends a new keyword. A null pointer may be entered for the comment parameter in which case the keyword comment field will be unmodified or left blank.

```
int fits_write_key_null / ffpkyu
    (fitsfile *fptr, char *keyname, char *comment, > int *status)

int fits_update_key_null / ffukyu
    (fitsfile *fptr, char *keyname, char *comment, > int *status)
```

3 Write (append) a COMMENT or HISTORY keyword to the CHU. The comment or history string will be continued over multiple keywords if it is longer than 70 characters.

```
int fits_write_comment / ffpcom
    (fitsfile *fptr, char *comment, > int *status)

int fits_write_history / ffphis
    (fitsfile *fptr, char *history, > int *status)
```

4 Write the DATE keyword to the CHU. The keyword value will contain the current system date as a character string in 'yyyy-mm-ddThh:mm:ss' format. If a DATE keyword already exists in the header, then this routine will simply update the keyword value with the current date.

```
int fits_write_date / ffpdat
    (fitsfile *fptr, > int *status)
```

5 Write a user specified keyword record into the CHU. This is a low–level routine which can be used to write any arbitrary record into the header. The record must conform to the all the FITS format requirements.

```
int fits_write_record / ffprec
    (fitsfile *fptr, char *card, > int *status)
```

6 Update an 80-character record in the CHU. If a keyword with the input name already exists, then it is overwritten by the value of card. This could modify the keyword name as well as the value and comment fields. If the keyword doesn't already exist then a new keyword card is appended to the header.

```
int fits_update_card / ffucrd
    (fitsfile *fptr, char *keyname, char *card, > int *status)
```

7 Modify (overwrite) the comment field of an existing keyword.

```
int fits_modify_comment / ffmcom
    (fitsfile *fptr, char *keyname, char *comment, > int *status)
```

8 Write the physical units string into 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
```

```
int fits_write_key_unit / ffpunt
    (fitsfile *fptr, char *keyname, char *unit, > int *status)
```

9 Rename an existing keyword, preserving the current value and comment fields.

```
int fits_modify_name / ffmnam
    (fitsfile *fptr, char *oldname, char *newname, > int *status)
```

10 Delete a keyword record. The space 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 may be used when specifying the name of the keyword to be deleted. The third routine deletes the (next) keyword that contains the literal character string specified by the 'string' argument.

```
int fits_delete_record / ffdrec
    (fitsfile *fptr, int   keynum,  > int *status)
```

```
int fits_delete_key / ffdkey
    (fitsfile *fptr, char *keyname, > int *status)

int fits_delete_str / ffdstr
    (fitsfile *fptr, char *string, > int *status)
```

5.5 Primary Array or IMAGE Extension I/O Routines

These routines read or write data values in the primary data array (i.e., the first HDU in a FITS file) or an IMAGE extension. There are also routines to get information about the data type and size of the image. Users should also read the following chapter on the CFITSIO iterator function which provides a more 'object oriented' method of reading and writing images. The iterator function is a little more complicated to use, but the advantages are that it usually takes less code to perform the same operation, and the resulting program often runs faster because the FITS files are read and written using the most efficient block size.

C programmers should note that the ordering of arrays in FITS files, and hence in all the CFITSIO calls, is more similar to the dimensionality of arrays in Fortran rather than C. For instance if a FITS image has NAXIS1 = 100 and NAXIS2 = 50, then a 2-D array just large enough to hold the image should be declared as array[50][100] and not as array[100][50].

The 'datatype' parameter specifies the data type of the 'nulval' and 'array' pointers and can have one of the following values: TBYTE, TSBYTE, TSHORT, TUSHORT, TINT, TUINT, TLONG, TLONGLONG, TULONG, TFLOAT, TDOUBLE. Automatic data type conversion is performed if the data type of the FITS array (as defined by the BITPIX keyword) differs from that specified by 'datatype'. The data values are also automatically scaled by the BSCALE and BZERO keyword values as they are being read or written in the FITS array.

1 Get the data type or equivalent data type of the image. The first routine returns the physical data type of the FITS image, as given by the BITPIX keyword, with allowed values of BYTE_IMG (8), SHORT_IMG (16), LONG_IMG (32), LONGLONG_IMG (64), FLOAT_IMG (-32), and DOUBLE_IMG (-64). The second routine is similar, except that if the image pixel values are scaled, with non-default values for the BZERO and BSCALE keywords, then the 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 FLOAT_IMG. Similarly if BITPIX = 16, BSCALE = 1, and BZERO = 32768, then the the pixel values span the range of an unsigned short integer and the returned data type will be USHORT_IMG.

```
int fits_get_img_type / ffgidt
    (fitsfile *fptr, > int *bitpix, int *status)

int fits_get_img_equivtype / ffgiet
    (fitsfile *fptr, > int *bitpix, int *status)
```

2 Get the number of dimensions, and/or the size of each dimension in the image . The number of axes in the image is given by naxis, and the size of each dimension is given by the naxes array (a maximum of maxdim dimensions will be returned).

```
int fits_get_img_dim / ffgidm
    (fitsfile *fptr, > int *naxis, int *status)

int fits_get_img_size / ffgisz
    (fitsfile *fptr, int maxdim, > long *naxes, int *status)

int fits_get_img_sizell / ffgiszll
    (fitsfile *fptr, int maxdim, > LONGLONG *naxes, int *status)

int fits_get_img_param / ffgipr
    (fitsfile *fptr, int maxdim, > int *bitpix, int *naxis, long *naxes,
     int *status)

int fits_get_img_paramll / ffgiprll
    (fitsfile *fptr, int maxdim, > int *bitpix, int *naxis, LONGLONG *naxes,
     int *status)
```

3 Create a new primary array or IMAGE extension with a specified data type and size. If the FITS file is currently empty then a primary array is created, otherwise a new IMAGE extension is appended to the file.

```
int fits_create_img / ffcrim
    ( fitsfile *fptr, int bitpix, int naxis, long *naxes, > int *status)

int fits_create_imgll / ffcrimll
    ( fitsfile *fptr, int bitpix, int naxis, LONGLONG *naxes, > int *status)
```

4 Copy an n-dimensional image in a particular row and column of a binary table (in a vector column) to or from a primary array or image extension.

The 'cell2image' routine will append a new image extension (or primary array) to the output file. Any WCS keywords associated with the input column image will be translated into the appropriate form for an image extension. Any other keywords in the table header that are not specifically related to defining the binary table structure or to other columns in the table will also be copied to the header of the output image.

The 'image2cell' routine will copy the input image into the specified row and column of the current binary table in the output file. The binary table HDU must exist before calling this routine, but it may be empty, with no rows or columns of data. The specified column (and row) will be created if it does not already exist. The 'copykeyflag' parameter controls which keywords are copied from the input image to the header of the output table: 0 = no keywords will be copied, 1 = all keywords will be copied (except those keywords that would be invalid in the table header), and 2 = copy only the WCS keywords.

```
int fits_copy_cell2image
    (fitsfile *infptr, fitsfile *outfptr, char *colname, long rownum,
     > int *status)
```

```
int fits_copy_image2cell
    (fitsfile *infptr, fitsfile *outfptr, char *colname, long rownum,
     int copykeyflag > int *status)
```

5 Write a rectangular subimage (or the whole image) to the FITS data array. The fpixel and lpixel arrays give the coordinates of the first (lower left corner) and last (upper right corner) pixels in FITS image to be written to.

```
int fits_write_subset / ffpss
    (fitsfile *fptr, int datatype, long *fpixel, long *lpixel,
     DTYPE *array, > int *status)
```

6 Write pixels into the FITS data array. 'fpixel' is an array of length NAXIS which gives the coordinate of the starting pixel to be written to, such that fpixel[0] is in the range 1 to NAXIS1, fpixel[1] is in the range 1 to NAXIS2, etc. The first pair of routines simply writes the array of pixels to the FITS file (doing data type conversion if necessary) whereas the second routines will substitute the appropriate FITS null value for any elements which are equal to the input value of nulval (note that this parameter gives the address of the null value, not the null value itself). For integer FITS arrays, the FITS null value is defined by the BLANK keyword (an error is returned if the BLANK keyword doesn't exist). For floating point FITS arrays the special IEEE NaN (Not-a-Number) value will be written into the FITS file. If a null pointer is entered for nulval, then the null value is ignored and this routine behaves the same as fits_write_pix.

```
int fits_write_pix / ffppx
    (fitsfile *fptr, int datatype, long *fpixel, LONGLONG nelements,
     DTYPE *array, int *status);

int fits_write_pixll / ffppxll
    (fitsfile *fptr, int datatype, LONGLONG *fpixel, LONGLONG nelements,
     DTYPE *array, int *status);

int fits_write_pixnull / ffppxn
    (fitsfile *fptr, int datatype, long *fpixel, LONGLONG nelements,
     DTYPE *array, DTYPE *nulval, > int *status);

int fits_write_pixnullll / ffppxnll
    (fitsfile *fptr, int datatype, LONGLONG *fpixel, LONGLONG nelements,
     DTYPE *array, DTYPE *nulval, > int *status);
```

7 Set FITS data array elements equal to the appropriate null pixel value. For integer FITS arrays, the FITS null value is defined by the BLANK keyword (an error is returned if the BLANK keyword doesn't exist). For floating point FITS arrays the special IEEE NaN (Not-a-Number) value will be written into the FITS file. Note that 'firstelem' is a scalar giving the offset to the first pixel to be written in the equivalent 1-dimensional array of image pixels.

```
int fits_write_null_img / ffpprn
    (fitsfile *fptr, LONGLONG firstelem, LONGLONG nelements, > int *status)
```

8 Read a rectangular subimage (or the whole image) from the FITS data array. The fpixel and lpixel arrays give the coordinates of the first (lower left corner) and last (upper right corner) pixels to be read from the FITS image. Undefined FITS array elements will be returned with a value = *nullval, (note that this parameter gives the address of the null value, not the null value itself) unless nulval = 0 or *nulval = 0, in which case no checks for undefined pixels will be performed.

```
int fits_read_subset / ffgsv
    (fitsfile *fptr, int  datatype, long *fpixel, long *lpixel, long *inc,
     DTYPE *nulval, > DTYPE *array, int *anynul, int *status)
```

9 Read pixels from the FITS data array. 'fpixel' is the starting pixel location and is an array of length NAXIS such that fpixel[0] is in the range 1 to NAXIS1, fpixel[1] is in the range 1 to NAXIS2, etc. The nelements parameter specifies the number of pixels to read. If fpixel is set to the first pixel, and nelements is set equal to the NAXIS1 value, then this routine would read the first row of the image. Alternatively, if nelements is set equal to NAXIS1 * NAXIS2 then it would read an entire 2D image, or the first plane of a 3-D datacube.

The first 2 routines will return any undefined pixels in the FITS array equal to the value of *nullval (note that this parameter gives the address of the null value, not the null value itself) unless nulval = 0 or *nulval = 0, in which case no checks for undefined pixels will be performed. The second 2 routines are similar except that any undefined pixels will have the corresponding nullarray element set equal to TRUE (= 1).

```
int fits_read_pix / ffgpxv
    (fitsfile *fptr, int  datatype, long *fpixel, LONGLONG nelements,
     DTYPE *nulval, > DTYPE *array, int *anynul, int *status)

int fits_read_pixll / ffgpxvll
    (fitsfile *fptr, int  datatype, LONGLONG *fpixel, LONGLONG nelements,
     DTYPE *nulval, > DTYPE *array, int *anynul, int *status)

int fits_read_pixnull / ffgpxf
    (fitsfile *fptr, int  datatype, long *fpixel, LONGLONG nelements,
     > DTYPE *array, char *nullarray, int *anynul, int *status)

int fits_read_pixnullll / ffgpxfll
    (fitsfile *fptr, int  datatype, LONGLONG *fpixel, LONGLONG nelements,
     > DTYPE *array, char *nullarray, int *anynul, int *status)
```

10 Copy a rectangular section of an image and write it to a new FITS primary image or image extension. The new image HDU is appended to the end of the output file; all the keywords in the input image will be copied to the output image. The common WCS keywords will be updated if necessary to correspond to the coordinates of the section. The format of the section expression is same as specifying an image section using the extended file name syntax (see "Image Section" in Chapter 10). (Examples: "1:100,1:200", "1:100:2, 1:*:2", "*, -*").

```
int fits_copy_image_section / ffcpimg
    (fitsfile *infptr, fitsfile *outfptr, char *section, int *status)
```

5.6 Image Compression

CFITSIO transparently supports the 2 methods of image compression described below.

1) The entire FITS file may be externally compressed with the gzip or Unix compress utility programs, producing a *.gz or *.Z file, respectively. When reading compressed files of this type, CFITSIO first uncompresses the entire file into memory before performing the requested read operations. Output files can be directly written in the gzip compressed format if the user-specified filename ends with '.gz'. In this case, CFITSIO initially writes the uncompressed file in memory and then compresses it and writes it to disk when the FITS file is closed, thus saving user disk space. Read and write access to these compressed FITS files is generally quite fast since all the I/O is performed in memory; the main limitation with this technique is that there must be enough available memory (or swap space) to hold the entire uncompressed FITS file.

2) CFITSIO also supports the FITS tiled image compression convention in which the image is subdivided into a grid of rectangular tiles, and each tile of pixels is individually compressed. The details of this FITS compression convention are described at the FITS Support Office web site at http://fits.gsfc.nasa.gov/fits_registry.html, and in the fpackguide pdf file that is included with the CFITSIO source file distributions Basically, the compressed image tiles are stored in rows of a variable length array column in a FITS binary table, however CFITSIO recognizes that this binary table extension contains an image and treats it as if it were an IMAGE extension. This tile-compressed format is especially well suited for compressing very large images because a) the FITS header keywords remain uncompressed for rapid read access, and because b) it is possible to extract and uncompress sections of the image without having to uncompress the entire image. This format is also much more effective in compressing floating point images than simply compressing the image using gzip or compress because it approximates the floating point values with scaled integers which can then be compressed more efficiently.

Currently CFITSIO supports 3 general purpose compression algorithms plus one other special-purpose compression technique that is designed for data masks with positive integer pixel values. The 3 general purpose algorithms are GZIP, Rice, and HCOMPRESS, and the special purpose algorithm is the IRAF pixel list compression technique (PLIO). There are 2 variants of the GZIP algorithm: GZIP_1 compresses the array of image pixel value normally with the GZIP algorithm, while GZIP_2 first shuffles the bytes in all the pixel values so that the most-significant byte of every pixel appears first, followed by the less significant bytes in sequence. GZIP_2 may be more effective in cases where the most significant byte in most of the image pixel values contains the same bit pattern. In principle, any number of other compression algorithms could also be supported by the FITS tiled image compression convention.

The FITS image can be subdivided into any desired rectangular grid of compression tiles. With the GZIP, Rice, and PLIO algorithms, the default is to take each row of the image as a tile. The

HCOMPRESS algorithm is inherently 2-dimensional in nature, so the default in this case is to take 16 rows of the image per tile. In most cases it makes little difference what tiling pattern is used, so the default tiles are usually adequate. In the case of very small images, it could be more efficient to compress the whole image as a single tile. Note that the image dimensions are not required to be an integer multiple of the tile dimensions; if not, then the tiles at the edges of the image will be smaller than the other tiles.

The 4 supported image compression algorithms are all 'loss-less' when applied to integer FITS images; the pixel values are preserved exactly with no loss of information during the compression and uncompression process. In addition, the HCOMPRESS algorithm supports a 'lossy' compression mode that will produce larger amount of image compression. This is achieved by specifying a non-zero value for the HCOMPRESS "scale" parameter. Since the amount of compression that is achieved depends directly on the RMS noise in the image, it is usually more convention to specify the HCOMPRESS scale factor relative to the RMS noise. Setting s = 2.5 means use a scale factor that is 2.5 times the calculated RMS noise in the image tile. In some cases it may be desirable to specify the exact scaling to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired scale value (typically in the range -2 to -100).

Very high compression factors (of 100 or more) can be achieved by using large HCOMPRESS scale values, however, this can produce undesirable "blocky" artifacts in the compressed image. A variation of the HCOMPRESS algorithm (called HSCOMPRESS) can be used in this case to apply a small amount of smoothing of the image when it is uncompressed to help cover up these artifacts. This smoothing is purely cosmetic and does not cause any significant change to the image pixel values.

Floating point FITS images (which have BITPIX = -32 or -64) usually contain too much "noise" in the least significant bits of the mantissa of the pixel values to be effectively compressed with any lossless algorithm. Consequently, floating point images are first quantized into scaled integer pixel values (and thus throwing away much of the noise) before being compressed with the specified algorithm (either GZIP, Rice, or HCOMPRESS). This technique produces much higher compression factors than simply using the GZIP utility to externally compress the whole FITS file, but it also means that the original floating value pixel values are not exactly preserved. When done properly, this integer scaling technique will only discard the insignificant noise while still preserving all the real information in the image. The amount of precision that is retained in the pixel values is controlled by the "quantization level" parameter, q. Larger values of q will result in compressed images whose pixels more closely match the floating point pixel values, but at the same time the amount of compression that is achieved will be reduced. Users should experiment with different values for this parameter to determine the optimal value that preserves all the useful information in the image, without needlessly preserving all the "noise" which will hurt the compression efficiency.

The default value for the quantization scale factor is 4.0, which means that scaled integer pixel values will be quantized such that the difference between adjacent integer values will be 1/4th of the noise level in the image background. CFITSIO uses an optimized algorithm to accurately estimate the noise in the image. As an example, if the RMS noise in the background pixels of an image = 32.0, then the spacing between adjacent scaled integer pixel values will equal 8.0 by default. Note that the RMS noise is independently calculated for each tile of the image, so the resulting integer scaling factor may fluctuate slightly for each tile. In some cases it may be desirable to specify the exact quantization level to be used, instead of specifying it relative to the calculated noise value. This may be done by specifying the negative of desired quantization level for the value of q. In the previous example, one could specify q = -8.0 so that the quantized integer levels differ by exactly 8.0. Larger negative values for q means that the levels are more coarsely spaced, and will produce higher compression factors.

When floating point images are being quantized, one must also specify what quantization method is to be used. The default algorithm is called "SUBTRACTIVE_DITHER_1". A second variation called "SUBTRACTIVE_DITHER_2" is also available, which does the same thing except that any pixels with a value of 0.0 are not dithered and instead the zero values are exactly preserved in the compressed image. One may also turn off dithering completely with the "NO_DITHER" option, but this is not recommended because it can cause larger systematic errors in measurements of the position or brightness of objects in the compressed image.

There are 3 methods for specifying all the parameters needed to write a FITS image in the tile compressed format. The parameters may either be specified at run time as part of the file name of the output compressed FITS file, or the writing program may call a set of helper CFITSIO subroutines that are provided for specifying the parameter values, or "compression directive" keywords may be added to the header of each image HDU to specify the compression parameters. These 3 methods are described below.

1) At run time, 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 by enclosing the compression parameters in square brackets following the root disk file name in the following format:

```
[compress NAME T1,T2; q[z] QLEVEL, s HSCALE]
```

where

```
NAME   = algorithm name:  GZIP, Rice, HCOMPRESS, HSCOMPRSS or PLIO
         may be abbreviated to the first letter (or HS for HSCOMPRESS)
T1,T2  = tile dimension (e.g. 100,100 for square tiles 100 pixels wide)
QLEVEL = quantization level for floating point FITS images
HSCALE = HCOMPRESS scale factor; default = 0 which is lossless.
```

Here are a few examples of this extended syntax:

```
myfile.fit[compress]    - use the default compression algorithm (Rice)
                          and the default tile size (row by row)

myfile.fit[compress G] - use the specified compression algorithm;
myfile.fit[compress R]     only the first letter of the algorithm
myfile.fit[compress P]     should be given.
myfile.fit[compress H]

myfile.fit[compress R 100,100]   - use Rice and 100 x 100 pixel tiles

myfile.fit[compress R; q 10.0] - quantization level = (RMS-noise) / 10.
```

```
  myfile.fit[compress R; qz 10.0] - quantization level = (RMS-noise) / 10.
                    also use the SUBTRACTIVE_DITHER_2 quantization method
  myfile.fit[compress HS; s 2.0]  -  HSCOMPRESS (with smoothing)
                                        and scale = 2.0 * RMS-noise
```

2) Before calling the CFITSIO routine to write the image header keywords (e.g., fits_create_image) the programmer can call the routines described below to specify the compression algorithm and the tiling pattern that is to be used. There are routines for specifying the various compression parameters and similar routines to return the current values of the parameters:

```
int fits_set_compression_type(fitsfile *fptr, int comptype, int *status)
int fits_set_tile_dim(fitsfile *fptr, int ndim, long *tilesize, int *status)
int fits_set_quantize_level(fitsfile *fptr, float qlevel, int *status)
int fits_set_quantize_method(fitsfile *fptr, int method, int *status)
int fits_set_quantize_dither(fitsfile *fptr, int dither, int *status)
int fits_set_dither_seed(fitsfile *fptr, int seed, int *status)
int fits_set_dither_offset(fitsfile *fptr, int offset, int *status)
int fits_set_lossy_int(fitsfile *fptr, int lossy_int, int *status)
    this forces integer image to be converted to floats, then quantized
int fits_set_huge_hdu(fitsfile *fptr, int huge, int *status);
    this should be called when the compressed image size is more than 4 GB.
int fits_set_hcomp_scale(fitsfile *fptr, float scale, int *status)
int fits_set_hcomp_smooth(fitsfile *fptr, int smooth, int *status)
            Set smooth = 1 to apply smoothing when uncompressing the image

int fits_get_compression_type(fitsfile *fptr, int *comptype, int *status)
int fits_get_tile_dim(fitsfile *fptr, int ndim, long *tilesize, int *status)
int fits_get_quantize_level(fitsfile *fptr, float *level, int *status)
int fits_get_hcomp_scale(fitsfile *fptr, float *scale, int *status)
```

Several symbolic constants are defined for use as the value of the 'comptype' parameter: GZIP_1, GZIP_2, RICE_1, HCOMPRESS_1 or PLIO_1. Entering NULL for comptype will turn off the tile-compression and cause normal FITS images to be written.

There are also defined symbolic constants for the quantization method: "SUBTRACTIVE_DITHER_1", "SUBTRACTIVE_DITHER_2", and "NO_DITHER".

3) CFITSIO will uses the values of the following keywords, if they are present in the header of the image HDU, to determine how to compress that HDU. These keywords override any compression parameters that were specified with the previous 2 methods.

```
FZALGOR  - 'RICE_1' , 'GZIP_1', 'GZIP_2', 'HCOMPRESS_1', 'PLIO_1', 'NONE'
FZTILE   - 'ROW', 'WHOLE', or '(n,m)'
FZQVALUE - float value (default = 4.0)
FZQMETHD - 'SUBTRACTIVE_DITHER_1', 'SUBTRACTIVE_DITHER_2', 'NO_DITHER'
FZDTHRSD - 'CLOCK', 'CHECKSUM', 1 - 10000
FZINT2F  -  T, or F:  Convert integers to floats, then quantize?
FZHSCALE - float value (default = 0).  Hcompress scale value.
```

No special action is required by software when read tile-compressed images because all the CFITSIO routines that read normal uncompressed FITS images also transparently read images in the tile-compressed format; CFITSIO essentially treats the binary table that contains the compressed tiles as if it were an IMAGE extension.

The following 2 routines are available for compressing or or decompressing an image:

```
int fits_img_compress(fitsfile *infptr, fitsfile *outfptr, int *status);
int fits_img_decompress (fitsfile *infptr, fitsfile *outfptr, int *status);
```

Before calling the compression routine, the compression parameters must first be defined in one of the 3 way described in the previous paragraphs. There is also a routine to determine if the current HDU contains a tile compressed image (it returns 1 or 0):

```
int fits_is_compressed_image(fitsfile *fptr, int *status);
```

A small example program called 'imcopy' is included with CFITSIO that can be used to compress (or uncompress) any FITS image. This program can be used to experiment with the various compression options on existing FITS images as shown in these examples:

```
1)  imcopy infile.fit 'outfile.fit[compress]'

       This will use the default compression algorithm (Rice) and the
       default tile size (row by row)

2)  imcopy infile.fit 'outfile.fit[compress GZIP]'

       This will use the GZIP compression algorithm and the default
       tile size (row by row).  The allowed compression algorithms are
       Rice, GZIP, and PLIO.  Only the first letter of the algorithm
       name needs to be specified.

3)  imcopy infile.fit 'outfile.fit[compress G 100,100]'

       This will use the GZIP compression algorithm and 100 X 100 pixel
       tiles.

4)  imcopy infile.fit 'outfile.fit[compress R 100,100; qz 10.0]'

       This will use the Rice compression algorithm, 100 X 100 pixel
       tiles, and quantization level = RMSnoise / 10.0 (assuming the
       input image has a floating point data type). By specifying
       qz instead of q, this means use the subtractive dither2
       quantization method.

5)  imcopy infile.fit outfile.fit
```

```
     If the input file is in tile-compressed format, then it will be
     uncompressed to the output file.  Otherwise, it simply copies
     the input image to the output image.

6)  imcopy 'infile.fit[1001:1500,2001:2500]'  outfile.fit

     This extracts a 500 X 500 pixel section of the much larger
     input image (which may be in tile-compressed format).  The
     output is a normal uncompressed FITS image.

7)  imcopy 'infile.fit[1001:1500,2001:2500]'  outfile.fit.gz

     Same as above, except the output file is externally compressed
     using the gzip algorithm.
```

5.7 ASCII and Binary Table Routines

These routines perform read and write operations on columns of data in FITS ASCII or Binary tables. Note that in the following discussions, the first row and column in a table is at position 1 not 0.

Users should also read the following chapter on the CFITSIO iterator function which provides a more 'object oriented' method of reading and writing table columns. The iterator function is a little more complicated to use, but the advantages are that it usually takes less code to perform the same operation, and the resulting program often runs faster because the FITS files are read and written using the most efficient block size.

5.7.1 Create New Table

1 Create a new ASCII or bintable table extension. If the FITS file is currently empty then a dummy primary array will be created before appending the table extension to it. The tbltype parameter defines the type of table and can have values of ASCII_TBL or BINARY_TBL. The naxis2 parameter gives the initial number of rows to be created in the table, and should normally be set = 0. CFITSIO will automatically increase the size of the table as additional rows are written. A non-zero number of rows may be specified to reserve space for that many rows, even if a fewer number of rows will be written. The tunit and extname parameters are optional and a null pointer may be given if they are not defined. The FITS Standard recommends that only letters, digits, and the underscore character be used in column names (the ttype parameter) with no embedded spaces. Trailing blank characters are not significant.

```
  int fits_create_tbl / ffcrtb
      (fitsfile *fptr, int tbltype, LONGLONG naxis2, int tfields, char *ttype[],
```

```
        char *tform[], char *tunit[], char *extname, int *status)
```

5.7.2 Column Information Routines

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.

```
int fits_get_num_rows / ffgnrw
    (fitsfile *fptr, > long *nrows, int *status);

int fits_get_num_rowsll / ffgnrwll
    (fitsfile *fptr, > LONGLONG *nrows, int *status);

int fits_get_num_cols / ffgncl
    (fitsfile *fptr, > int *ncols, int *status);
```

2 Get the table column number (and name) of the column whose name matches an input template name. If casesen = CASESEN then the column name match will be case-sensitive, whereas if casesen = CASEINSEN then the case will be ignored. As a general rule, the column names should be treated as case INsensitive.

The input column name template may be either the exact name of the column to be searched for, or it may contain wild card characters (*, ?, or #), or it may contain the integer number of the desired column (with the first column = 1). The '*' wild card 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). 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 COL_NOT_UNIQUE as a warning that a unique match was not found. To find the other cases that match the template, call the routine again leaving the input status value equal to COL_NOT_UNIQUE and the next matching name will then be returned. Repeat this process until a status = COL_NOT_FOUND is returned.

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

```
int fits_get_colnum / ffgcno
    (fitsfile *fptr, int casesen, char *templt, > int *colnum,
     int *status)

int fits_get_colname / ffgcnn
    (fitsfile *fptr, int casesen, char *templt, > char *colname,
     int *colnum, int *status)
```

3 Return the data type, vector repeat value, and the width in bytes of a column in an ASCII or binary table. Allowed values for the data type in ASCII tables are: TSTRING, TSHORT, TLONG, TFLOAT, and TDOUBLE. Binary tables also support these types: TLOGICAL, TBIT, TBYTE, TCOMPLEX and TDBLCOMPLEX. The negative of the data type code value is returned if it is a variable length array column. Note that in the case of a 'J' 32-bit integer binary table column, this routine will return data type = TINT32BIT (which in fact is equivalent to TLONG). With most current C compilers, a value in a 'J' column has the same size as an 'int' variable, and may not be equivalent to a 'long' variable, which is 64-bits long on an increasing number of compilers.

The 'repeat' parameter returns the vector repeat count on the binary table TFORMn keyword value. (ASCII table columns always have repeat = 1). The 'width' parameter returns the width in bytes of a single column element (e.g., a '10D' binary table column will have width = 8, an ASCII table 'F12.2' column will have width = 12, and a binary table'60A' character string column will have width = 60); Note that CFITSIO supports the local convention for specifying arrays of fixed length 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 means that each row of the table contains 5 12-character substrings within the 60-character field, and thus in this case this routine will return typecode = TSTRING, 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 number of substrings in any binary table character string field can be calculated by (repeat/width). A null pointer may be given for any of the output parameters that are not needed.

The second routine, fit_get_eqcoltype 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 TUSHORT, not TSHORT. Similarly, if a column has TTYPEn = '1I' and TSCALn = 0.12, then the returned typecode will be TFLOAT.

```
int fits_get_coltype / ffgtcl
    (fitsfile *fptr, int colnum, > int *typecode, long *repeat,
     long *width, int *status)

int fits_get_coltypell / ffgtclll
    (fitsfile *fptr, int colnum, > int *typecode, LONGLONG *repeat,
     LONGLONG *width, int *status)

int fits_get_eqcoltype / ffeqty
    (fitsfile *fptr, int colnum, > int *typecode, long *repeat,
     long *width, int *status)

int fits_get_eqcoltypell / ffeqtyll
    (fitsfile *fptr, int colnum, > int *typecode, LONGLONG *repeat,
     LONGLONG *width, int *status)
```

4 Return the display width of a column. This is the length of the string that will be returned by the fits_read_col routine 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.

```
int fits_get_col_display_width / ffgcdw
    (fitsfile *fptr, int colnum, > int *dispwidth, int *status)
```

5 Return the number of and size of the dimensions of a table column in a binary table. Normally this information is given by the TDIMn keyword, but if this keyword is not present then this routine returns naxis = 1 and naxes[0] equal to the repeat count in the TFORM keyword.

```
int fits_read_tdim / ffgtdm
    (fitsfile *fptr, int colnum, int maxdim, > int *naxis,
     long *naxes, int *status)

int fits_read_tdimll / ffgtdmll
    (fitsfile *fptr, int colnum, int maxdim, > int *naxis,
     LONGLONG *naxes, int *status)
```

6 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 fits_read_tdim.

```
int fits_decode_tdim / ffdtdm
    (fitsfile *fptr, char *tdimstr, int colnum, int maxdim, > int *naxis,
     long *naxes, int *status)

int fits_decode_tdimll / ffdtdmll
    (fitsfile *fptr, char *tdimstr, int colnum, int maxdim, > int *naxis,
     LONGLONG *naxes, int *status)
```

7 Write 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.

```
int fits_write_tdim / ffptdm
    (fitsfile *fptr, int colnum, int naxis, long *naxes, > int *status)

int fits_write_tdimll / ffptdmll
    (fitsfile *fptr, int colnum, int naxis, LONGLONG *naxes, > int *status)
```

5.7.3 Routines to Edit Rows or Columns

1 Insert or delete rows in an ASCII or binary table. When inserting rows all the rows following row FROW are shifted down by NROWS rows; if FROW = 0 then the blank rows are inserted at the beginning of 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.

The first delete routine deletes NROWS consecutive rows starting with row FIRSTROW. The second delete routine takes an input string that lists the rows or row ranges (e.g., '5-10,12,20-30'), whereas the third delete routine takes an input integer array that specifies each individual row to be deleted. In both latter cases, the input list of rows to delete must be sorted in ascending order. These routines update the NAXIS2 keyword to reflect the new number of rows in the table.

```
int fits_insert_rows / ffirow
    (fitsfile *fptr, LONGLONG firstrow, LONGLONG nrows, > int *status)

int fits_delete_rows / ffdrow
    (fitsfile *fptr, LONGLONG firstrow, LONGLONG nrows, > int *status)

int fits_delete_rowrange / ffdrrg
    (fitsfile *fptr, char *rangelist, > int *status)

int fits_delete_rowlist / ffdrws
    (fitsfile *fptr, long *rowlist, long nrows, > int *status)

int fits_delete_rowlistll / ffdrwsll
    (fitsfile *fptr, LONGLONG *rowlist, LONGLONG nrows, > int *status)
```

2 Insert or delete column(s) in an ASCII or binary table. When inserting, 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 shifted over to allow room for the new column(s). The index number on all the following keywords will be incremented or decremented 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.

```
int fits_insert_col / fficol
    (fitsfile *fptr, int colnum, char *ttype, char *tform,
     > int *status)

int fits_insert_cols / fficls
    (fitsfile *fptr, int colnum, int ncols, char **ttype,
     char **tform, > int *status)

int fits_delete_col / ffdcol(fitsfile *fptr, int colnum, > int *status)
```

3 Copy a column from one HDU to another (or to the same HDU). If create_col = 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 data type). If outcolnum is greater than the number of column in the table, then the new column will be appended to the end of the table. Note that the first column in a table is at colnum = 1. The standard indexed keywords that related to the column (e.g., TDISPn, TUNITn, TCRPXn, TCDLTn, etc.) will also be copied.

```
int fits_copy_col / ffcpcl
    (fitsfile *infptr, fitsfile *outfptr, int incolnum, int outcolnum,
     int create_col, > int *status);
```

4 Copy 'nrows' consecutive rows from one table to another, beginning with row 'firstrow'. These rows will be appended to any existing rows in the output table. Note that the first row in a table is at row = 1.

```
int fits_copy_rows / ffcprw
    (fitsfile *infptr, fitsfile *outfptr, LONGLONG firstrow,
     LONGLONG nrows, > int *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.

```
int fits_modify_vector_len / ffmvec
    (fitsfile *fptr, int colnum, LONGLONG newveclen, > int *status)
```

5.7.4 Read and Write Column Data Routines

The following routines write or read data values in the current ASCII or binary table extension. If a write operation extends beyond the current size of the table, then the number of rows in the table will automatically be increased and the NAXIS2 keyword value will be updated. Attempts to read beyond the end of the table will result in an error.

Automatic data type conversion is performed for numerical data types (only) if the data type of the column (defined by the TFORMn keyword) differs from the data type of the array in the calling routine. ASCII and binary tables support the following data type values: TSTRING, TBYTE, TSBYTE, TSHORT, TUSHORT, TINT, TUINT, TLONG, TLONGLONG, TULONG, TFLOAT, or TDOUBLE. Binary tables also support TLOGICAL (internally mapped to the 'char' data type), TCOMPLEX, and TDBLCOMPLEX.

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.

Individual bits in a binary table 'X' or 'B' column may be read/written to/from a *char array by specifying the TBIT datatype. The *char array will be interpreted as an array of logical TRUE (1) or FALSE (0) values that correspond to the value of each bit in the FITS 'X' or 'B' column. Alternatively, the values in a binary table 'X' column may be read/written 8 bits at a time to/from an array of 8-bit integers by specifying the TBYTE datatype.

Note that within the context of these routines, the TSTRING data type corresponds to a C 'char**' data type, i.e., a pointer to an array of pointers to an array of characters. This is different from the keyword reading and writing routines where TSTRING corresponds to a C 'char*' data type, i.e., a single pointer to an array of characters. When reading strings from a table, the char arrays obviously must have been allocated long enough to hold the whole FITS table string.

Numerical data values are automatically scaled by the TSCALn and TZEROn keyword values (if they exist).

In the case of binary tables with vector elements, the 'felem' parameter defines the starting element (beginning with 1, not 0) within the cell (a cell is defined as the intersection of a row and a column and may contain a single value or a vector of values). The felem parameter is ignored when dealing with ASCII tables. Similarly, in the case of binary tables the 'nelements' parameter specifies the total number of vector values to be read or written (continuing on subsequent rows if required) and not the number of table cells.

1 Write elements into an ASCII or binary table column.

The first routine simply writes the array of values to the FITS file (doing data type conversion if necessary) whereas the second routine will substitute the appropriate FITS null value for all elements which are equal to the input value of nulval (note that this parameter gives the address of nulval, not the null value itself). For integer columns the FITS null value is defined by the TNULLn keyword (an error is returned if the keyword doesn't exist). For floating point columns the special IEEE NaN (Not-a-Number) value will be written into the FITS file. If a null pointer is entered for nulval, then the null value is ignored and this routine behaves the same as the first routine. The third routine simply writes undefined pixel values to the column. The fourth routine fills every column in the table with null values, in the specified rows (ignoring any columns that do not have a defined null value).

```
int fits_write_col / ffpcl
    (fitsfile *fptr, int datatype, int colnum, LONGLONG firstrow,
     LONGLONG firstelem, LONGLONG nelements, DTYPE *array, > int *status)

int fits_write_colnull / ffpcn
    (fitsfile *fptr, int datatype, int colnum, LONGLONG firstrow,
    LONGLONG firstelem, LONGLONG nelements, DTYPE *array, DTYPE *nulval,
    > int *status)

 int fits_write_col_null / ffpclu
     (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
      LONGLONG nelements, > int *status)

 int fits_write_nullrows / ffprwu
     (fitsfile *fptr, LONGLONG firstrow, LONGLONG nelements, > int *status)
```

2 Read elements from an ASCII or binary table column. The data type parameter specifies the data type of the 'nulval' and 'array' pointers; Undefined array elements will be returned with a value = *nullval, (note that this parameter gives the address of the null value, not the null value itself) unless nulval = 0 or *nulval = 0, in which case no checking for undefined pixels will be performed. The second routine is similar except that any undefined pixels will have the corresponding nullarray element set equal to TRUE (= 1).

Any column, regardless of it's intrinsic data type, 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 data type of the column. The length of the returned strings (not including the null terminating character) can be determined with the fits_get_col_display_width 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.

```
int fits_read_col / ffgcv
    (fitsfile *fptr, int datatype, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, DTYPE *nulval, DTYPE *array, int *anynul, int *status)

int fits_read_colnull / ffgcf
    (fitsfile *fptr, int datatype, int colnum, LONGLONG firstrow, LONGLONG firstelem,
    LONGLONG nelements, DTYPE *array, char *nullarray, int *anynul, int *status)
```

5.7.5 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 '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. Upon return, *n_good_rows contains the number of rows that evaluate to TRUE.

```
int fits_find_rows / fffrow
    (fitsfile *fptr,  char *expr, long firstrow, long nrows,
    > long *n_good_rows, char *row_status,  int *status)
```

2 Find the first row which satisfies the input boolean expression

```
int fits_find_first_row / ffffrw
    (fitsfile *fptr,  char *expr, > long *rownum, int *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).

```
int fits_select_rows / ffsrow
    (fitsfile *infptr, fitsfile *outfptr,  char *expr,  > int *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 fits_test_expr to obtain the dimensions of the results.

```
int fits_calc_rows / ffcrow
    (fitsfile *fptr,  int datatype, char *expr, long firstrow,
     long nelements, void *nulval, > void *array,  int *anynul, int *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.

```
int fits_calculator / ffcalc
    (fitsfile *infptr, char *expr, fitsfile *outfptr, char *parName,
     char *parInfo, >  int *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.

```
int fits_calculator_rng / ffcalc_rng
    (fitsfile *infptr, char *expr, fitsfile *outfptr, char *parName,
     char *parInfo, int nranges, long *firstrow, long *lastrow
     >  int *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.

```
int fits_test_expr / fftexp
    (fitsfile *fptr, char *expr, int maxdim > int *datatype, long *nelem, int *naxis,
     long *naxes, int *status)
```

5.7.6 Column Binning or Histogramming Routines

The following routines may be useful when performing histogramming operations on column(s) of a table to generate an image in a primary array or image extension.

1 Calculate the histogramming parameters (min, max, and bin size for each axis of the histogram, based on a variety of possible input parameters. If the input names of the columns to be binned are null, then the routine will first look for the CPREF = "NAME1, NAME2, ..." keyword which lists the preferred columns. If not present, then the routine will assume the column names X, Y, Z, and T for up to 4 axes (as specified by the NAXIS parameter).

MININ and MAXIN are input arrays that give the minimum and maximum value for the histogram, along each axis. Alternatively, the name of keywords that give the min, max, and binsize may be give with the MINNAME, MAXNAME, and BINNAME array parameters. If the value = DOUBLENULLVALUE and no keyword names are given, then the routine will use the TLMINn and TLMAXn keywords, if present, or the actual min and/or max values in the column.

BINSIZEIN is an array giving the binsize along each axis. If the value = DOUBLENULL-VALUE, and a keyword name is not specified with BINNAME, then this routine will first look for the TDBINn keyword, or else will use a binsize = 1, or a binsize that produces 10 histogram bins, which ever is smaller.

```
int fits_calc_binning
 Input parameters:
   (fitsfile *fptr,  /* IO - pointer to table to be binned           */
    int naxis,       /* I - number of axes/columns in the binned image  */
    char colname[4][FLEN_VALUE],   /* I - optional column names         */
    double *minin,     /* I - optional lower bound value for each axis  */
    double *maxin,     /* I - optional upper bound value, for each axis */
    double *binsizein, /* I - optional bin size along each axis         */
    char minname[4][FLEN_VALUE], /* I - optional keywords for min       */
```

```
    char maxname[4][FLEN_VALUE], /* I - optional keywords for max      */
    char binname[4][FLEN_VALUE], /* I - optional keywords for binsize   */
 Output parameters:
    int *colnum,     /* O - column numbers, to be binned */
    long *naxes,     /* O - number of bins in each histogram axis */
    float *amin,     /* O - lower bound of the histogram axes */
    float *amax,     /* O - upper bound of the histogram axes */
    float *binsize,  /* O - width of histogram bins/pixels on each axis */
    int *status)
```

2 Copy the relevant keywords from the header of the table that is being binned, to the the header of the output histogram image. This will not copy the table structure keywords (e.g., NAXIS, TFORMn, TTYPEn, etc.) nor will it copy the keywords that apply to other columns of the table that are not used to create the histogram. This routine will translate the names of the World Coordinate System (WCS) keywords for the binned columns into the form that is need for a FITS image (e.g., the TCTYPn table keyword will be translated to the CTYPEn image keyword).

```
int fits_copy_pixlist2image
    (fitsfile *infptr,   /* I - pointer to input HDU */
     fitsfile *outfptr,  /* I - pointer to output HDU */
     int firstkey,       /* I - first HDU keyword to start with */
     int naxis,          /* I - number of axes in the image */
     int *colnum,        /* I - numbers of the columns to be binned  */
     int *status)        /* IO - error status */
```

3 Write a set of default WCS keywords to the histogram header, IF the WCS keywords do not already exist. This will create a linear WCS where the coordinate types are equal to the original column names.

```
int fits_write_keys_histo
   (fitsfile *fptr,     /* I - pointer to table to be binned             */
    fitsfile *histptr,  /* I - pointer to output histogram image HDU     */
    int naxis,          /* I - number of axes in the histogram image     */
    int *colnum,        /* I - column numbers of the binned columns      */
    int *status)
```

4 Update the WCS keywords in a histogram image header that give the location of the reference pixel (CRPIXn), and the pixel size (CDELTn), in the binned image.

```
int fits_rebin_wcs
   (fitsfile *fptr,     /* I - pointer to table to be binned           */
    int naxis,          /* I - number of axes in the histogram image   */
    float *amin,        /* I - first pixel include in each axis        */
    float *binsize,     /* I - binning factor for each axis            */
    int *status)
```

5 Bin the values in the input table columns, and write the histogram array to the output FITS image (histptr).

```
int fits_make_hist
 (fitsfile *fptr,    /* I - pointer to table with X and Y cols;      */
  fitsfile *histptr, /* I - pointer to output FITS image             */
  int bitpix,        /* I - datatype for image: 16, 32, -32, etc     */
  int naxis,         /* I - number of axes in the histogram image    */
  long *naxes,       /* I - size of axes in the histogram image      */
  int *colnum,       /* I - column numbers (array length = naxis)    */
  float *amin,       /* I - minimum histogram value, for each axis   */
  float *amax,       /* I - maximum histogram value, for each axis   */
  float *binsize,    /* I - bin size along each axis                 */
  float weight,      /* I - binning weighting factor (FLOATNULLVALUE */
                     /*     for no weighting)                        */
  int wtcolnum,      /* I - keyword or col for weight      (or NULL) */
  int recip,         /* I - use reciprocal of the weight? 0 or 1     */
  char *selectrow,   /* I - optional array (length = no. of          */
                     /* rows in the table).  If the element is true  */
                     /* then the corresponding row of the table will */
                     /* be included in the histogram, otherwise the  */
                     /* row will be skipped.  Ingnored if *selectrow */
                     /* is equal to NULL.                            */
  int *status)
```

5.8 Utility Routines

5.8.1 File Checksum Routines

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. See Appendix B for the definition of the parameters used in these routines.

1 Compute and write the DATASUM and CHECKSUM keyword values for the CHDU into the current header. If the 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).

```
int fits_write_chksum / ffpcks
    (fitsfile *fptr, > int *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.

```
int fits_update_chksum / ffupck
    (fitsfile *fptr, > int *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 routine 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.

```
int fits_verify_chksum / ffvcks
    (fitsfile *fptr, > int *dataok, int *hduok, int *status)
```

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

```
int fits_get_chksum/ /ffgcks
    (fitsfile *fptr, > unsigned long *datasum, unsigned long *hdusum,
     int *status)
```

5 Encode a checksum value into a 16-character string. If complm is non-zero (true) then the 32-bit sum value will be complemented before encoding.

```
int fits_encode_chksum / ffesum
    (unsigned long sum, int complm, > char *ascii);
```

6 Decode a 16-character checksum string into a unsigned long value. If is non-zero (true). then the 32-bit sum value will be complemented after decoding. The checksum value is also returned as the value of the function.

```
unsigned long fits_decode_chksum / ffdsum
         (char *ascii, int complm, > unsigned long *sum);
```

5.8.2 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. See Appendix B for the definition of the parameters used in these routines.

1 Get the current system date. C already provides standard library routines for getting the current date and time, but this routine is provided for compatibility with the Fortran FITSIO library. The returned year has 4 digits (1999, 2000, etc.)

```
int fits_get_system_date/ffgsdt
    ( > int *day, int *month, int *year, int *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.

```
int fits_get_system_time/ffgstm
    (> char *datestr, int  *timeref, int *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 fits_time2str instead to always return a date string using the new FITS format.

```
int fits_date2str/ffdt2s
    (int year, int month, int day, > char *datestr, int *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').

```
int fits_time2str/fftm2s
    (int year, int month, int day, int hour, int minute, double second,
    int decimals, > char *datestr, int *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. Null pointers may be supplied for any unwanted output date parameters.

```
int fits_str2date/ffs2dt
    (char *datestr, > int *year, int *month, int *day, int *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...'). Null pointers may be supplied for any unwanted output date and time parameters.

```
int fits_str2time/ffs2tm
    (char *datestr, > int *year, int *month, int *day, int *hour,
    int *minute, double *second, int *status)
```

5.8.3 General Utility Routines

The following utility routines may be useful for certain applications.

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

```
float fits_get_version / ffvers ( > float *version)
```

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

```
void fits_write_errmsg / ffpmsg (char *err_msg)
```

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

```
void fits_uppercase / ffupch (char *string)
```

4 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 casesen = CASESEN = TRUE then the match will be case sensitive, otherwise the case of the letters will be ignored if casesen = CASEINSEN = FALSE. 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.

```
void fits_compare_str / ffcmps
     (char *templt, char *string, int casesen, > int *match, int *exact)
```

5 Split a string containing a list of names (typically file names or column names) into individual name tokens by a sequence of calls to fits_split_names. The names in the list must be delimited by a comma and/or spaces. This routine ignores spaces and commas that occur within parentheses, brackets, or curly brackets. It also strips any leading and trailing blanks from the returned name.

This routine is similar to the ANSI C 'strtok' function:

The first call to fits_split_names has a non-null input string. It finds the first name in the string and terminates it by overwriting the next character of the string with a null terminator and returns a pointer to the name. Each subsequent call, indicated by a NULL value of the input string, returns the next name, searching from just past the end of the previous name. It returns NULL when no further names are found.

```
char *fits_split_names(char *namelist)
```

The following example shows how a string would be split into 3 names:

```
myfile[1][bin (x,y)=4], file2.fits  file3.fits
^^^^^^^^^^^^^^^^^^^^^^  ^^^^^^^^^^  ^^^^^^^^^^
    1st name              2nd name    3rd name
```

6 Test that the keyword name contains only legal characters (A-Z,0-9, hyphen, and underscore) or that the keyword record contains only legal printable ASCII characters

```
int fits_test_keyword / fftkey (char *keyname, > int *status)

int fits_test_record / fftrec (char *card, > int *status)
```

7 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 CFITSIO routines in which the returned value is not necessarily equal to the status value).

```
int fits_null_check / ffnchk (char *card, > int *status)
```

8 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.

```
int fits_get_keyname / ffgknm
    (char *card, > char *keyname, int *keylength, int *status)
```

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

```
int fits_parse_value / ffpsvc
    (char *card, > char *value, char *comment, int *status)
```

10 Construct a properly formated 80-character header keyword record from the input keyword name, keyword value, and keyword comment strings. Hierarchical keyword names (e.g., "ESO TELE CAM") are supported. The value string may contain an integer, floating point, logical, or quoted character string (e.g., "12", "15.7", "T", or "'NGC 1313'").

```
int fits_make_key / ffmkky
    (const char *keyname, const char *value, const char *comment,
      > char *card, int *status)
```

11 Construct an array indexed keyword name (ROOT + nnn). This routine appends the sequence number to the root string to create a keyword name (e.g., 'NAXIS' + 2 = 'NAXIS2')

```
int fits_make_keyn / ffkeyn
    (char *keyroot, int value, > char *keyname, int *status)
```

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

```
int fits_make_nkey / ffnkey
    (int value, char *keyroot, > char *keyname, int *status)
```

13 Determine the data type of a keyword value string. This routine parses the keyword value string to determine its data type. Returns 'C', 'L', 'I', 'F' or 'X', for character string, logical, integer, floating point, or complex, respectively.

```
int fits_get_keytype / ffdtyp
    (char *value, > char *dtype, int *status)
```

14 Determine the integer data type of an integer keyword value string. The returned datatype value is the minimum integer datatype (starting from top of the following list and working down) required to store the integer value:

```
  Data Type      Range
   TSBYTE:     -128 to 127
   TBYTE:       128 to 255
   TSHORT:     -32768 to 32767
```

```
TUSHORT:     32768 to 65535
TINT        -2147483648 to 2147483647
TUINT        2147483648 to 4294967295
TLONGLONG   -9223372036854775808 to 9223372036854775807
```

The *neg parameter returns 1 if the input value is negative and returns 0 if it is non-negative.

```
int fits_get_inttype / ffinttyp
    (char *value, > int *datatype, int *neg, int *status)
```

15 Return the class of an 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 CFITSIO 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  WCS keywords defined in the the WCS papers, including:
                   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, DATE-OBS
TYP_COMM_KEY   130 COMMENT, HISTORY, (blank keyword)
TYP_CONT_KEY   140 CONTINUE
TYP_USER_KEY   150 all other keywords
```

```
int fits_get_keyclass / ffgkcl (char *card)
```

16 Parse the 'TFORM' binary table column format string. This routine parses the input TFORM character string and returns the integer data type code, the repeat count of the field, and, in the case of character string fields, the length of the unit string. See Appendix B for the allowed values for the returned typecode parameter. A null pointer may be given for any output parameters that are not needed.

```
int fits_binary_tform / ffbnfm
    (char *tform, > int *typecode, long *repeat, long *width,
     int *status)

int fits_binary_tformll / ffbnfmll
    (char *tform, > int *typecode, LONGLONG *repeat, long *width,
     int *status)
```

17 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 data type 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 data type codes are the same as for the binary table, with the following additional rules: integer columns that are between 1 and 4 characters wide are defined to be short integers (code = TSHORT). Wider integer columns are defined to be regular integers (code = TLONG). Similarly, Fixed decimal point columns (with TFORM = 'Fw.d') are defined to be single precision reals (code = TFLOAT) if w is between 1 and 7 characters wide, inclusive. Wider 'F' columns will return a double precision data code (= TDOUBLE). 'Ew.d' format columns will have datacode = TFLOAT, and 'Dw.d' format columns will have datacode = TDOUBLE. A null pointer may be given for any output parameters that are not needed.

```
int fits_ascii_tform / ffasfm
    (char *tform, > int *typecode, long *width, int *decimals,
     int *status)
```

18 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).

```
int fits_get_tbcol / ffgabc
    (int tfields, char **tform, int space, > long *rowlen,
     long *tbcol, int *status)
```

19 Parse a template header record and return a formatted 80-character string suitable for appending to (or deleting from) a FITS header file. This routine 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 fits_write_record, ffmcrd, or fits_delete_key routines to append or modify a FITS header record.

```
int fits_parse_template / ffgthd
    (char *templt, > char *card, int *keytype, int *status)
```

The input templt 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. A legal FITS keyword name may only contain the characters A-Z, 0-9, and '-' (minus sign) and underscore. This routine will automatically convert any lowercase characters to uppercase in the output string. 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).

- The VALUE token must be separated from the KEYNAME token by one or more spaces and/or an '=' character. The data type 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. If the value token is a character string that contains 1 or more embedded blank space characters or slash ('/') characters then the entire character string must be enclosed in single quotes.

- The COMMENT token is optional, but if present must be separated from the VALUE token by a blank space or a '/' character.

- One exception to the above rules is that if the first non-blank character in the first 8 characters of 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 (without an equals sign between them) 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 ffmnam routine which will change the keyword name.

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

```
keytype                  interpretation
-------        -------------------------------------------------
   -2          Rename the keyword with name = the first 8 characters of CARD
               to the new name given in characters 41 - 48 of CARD.
```

```
         -1          delete the keyword with this name from the FITS header.

          0          append the CARD string to the FITS header if the
                     keyword does not already exist, otherwise update
                     the keyword value and/or comment field if is already exists.

          1          This is a HISTORY or COMMENT keyword; append it to the header

          2          END record; do not explicitly write it to the FITS file.
```

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
```

20 Translate a keyword name into a new name, based on a set of patterns. This routine is useful for translating keywords in cases such as adding or deleting columns in a table, or copying a column from one table to another, or extracting an array from a cell in a binary table column into an image extension. In these cases, it is necessary to translate the names of the keywords associated with the original table column(s) into the appropriate keyword name in the final file. For example, if column 2 is deleted from a table, then the value of 'n' in all the TFORMn and TTYPEn keywords for columns 3 and higher must be decremented by 1. Even more complex translations are sometimes needed to convert the WCS keywords when extracting an image out of a table column cell into a separate image extension.

The user passes an array of patterns to be matched. Input pattern number i is pattern[i][0], and output pattern number i is pattern[i][1]. Keywords are matched against the input patterns. If a match is found then the keyword is re-written according to the output pattern.

Order is important. The first match is accepted. The fastest match will be made when templates with the same first character are grouped together.

Several characters have special meanings:

```
i,j - single digits, preserved in output template
n - column number of one or more digits, preserved in output template
m - generic number of one or more digits, preserved in output template
a - coordinate designator, preserved in output template
```

```
# - number of one or more digits
? - any character
* - only allowed in first character position, to match all
    keywords; only useful as last pattern in the list
```

i, j, n, and m are returned by the routine.

For example, the input pattern "iCTYPn" will match "1CTYP5" (if n_value is 5); the output pattern "CTYPEi" will be re-written as "CTYPE1". Notice that "i" is preserved.

The following output patterns are special:

"-" - do not copy a keyword that matches the corresponding input pattern

"+" - copy the input unchanged

The inrec string could be just the 8-char keyword name, or the entire 80-char header record. Characters 9 - 80 in the input string simply get appended to the translated keyword name.

If n_range = 0, then only keywords with 'n' equal to n_value will be considered as a pattern match. If n_range = +1, then all values of 'n' greater than or equal to n_value will be a match, and if -1, then values of 'n' less than or equal to n_value will match.

```
int fits_translate_keyword(
      char *inrec,        /* I - input string */
      char *outrec,       /* O - output converted string, or */
                          /*     a null string if input does not  */
                          /*     match any of the patterns */
      char *patterns[][2],/* I - pointer to input / output string */
                          /*     templates */
      int npat,           /* I - number of templates passed */
      int n_value,        /* I - base 'n' template value of interest */
      int n_offset,       /* I - offset to be applied to the 'n' */
                          /*     value in the output string */
      int n_range,        /* I - controls range of 'n' template */
                          /*     values of interest (-1,0, or +1) */
      int *pat_num,       /* O - matched pattern number (0 based) or -1 */
      int *i,             /* O - value of i, if any, else 0 */
      int *j,             /* O - value of j, if any, else 0 */
      int *m,             /* O - value of m, if any, else 0 */
      int *n,             /* O - value of n, if any, else 0 */
      int *status)        /* IO - error status */
```

Here is an example of some of the patterns used to convert the keywords associated with an image in a cell of a table column into the keywords appropriate for an IMAGE extension:

```
    char *patterns[][2] = {{"TSCALn",  "BSCALE"  },  /* Standard FITS keywords */
   {"TZEROn",  "BZERO"   },
   {"TUNITn",  "BUNIT"   },
   {"TNULLn",  "BLANK"   },
```

```
   {"TDMINn",   "DATAMIN" },
   {"TDMAXn",   "DATAMAX" },
   {"iCTYPn",   "CTYPEi"  },  /* Coordinate labels */
   {"iCTYna",   "CTYPEia" },
   {"iCUNIn",   "CUNITi"  },  /* Coordinate units */
   {"iCUNna",   "CUNITia" },
   {"iCRVLn",   "CRVALi"  },  /* WCS keywords */
   {"iCRVna",   "CRVALia" },
   {"iCDLTn",   "CDELTi"  },
   {"iCDEna",   "CDELTia" },
   {"iCRPXn",   "CRPIXi"  },
   {"iCRPna",   "CRPIXia" },
   {"ijPCna",   "PCi_ja"  },
   {"ijCDna",   "CDi_ja"  },
   {"iVn_ma",   "PVi_ma"  },
   {"iSn_ma",   "PSi_ma"  },
   {"iCRDna",   "CRDERia" },
   {"iCSYna",   "CSYERia" },
   {"iCROTn",   "CROTAi"  },
   {"WCAXna",   "WCSAXESa"},
   {"WCSNna",   "WCSNAMEa"}};
```

21 Translate the keywords in the input HDU into the keywords that are appropriate for the output HDU. This is a driver routine that calls the previously described routine.

```
int fits_translate_keywords(
   fitsfile *infptr,   /* I - pointer to input HDU */
   fitsfile *outfptr,  /* I - pointer to output HDU */
   int firstkey,       /* I - first HDU record number to start with */
   char *patterns[][2],/* I - pointer to input / output keyword templates */
   int npat,           /* I - number of templates passed */
   int n_value,        /* I - base 'n' template value of interest */
   int n_offset,       /* I - offset to be applied to the 'n' */
                         /*     value in the output string */
   int n_range,        /* I - controls range of 'n' template */
                       /*     values of interest (-1,0, or +1) */
   int *status)        /* IO - error status */
```

22 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.

```
int fits_parse_range / ffrwrg(char *rowlist, LONGLONG maxrows, int maxranges, >
   int *numranges, long *rangemin, long *rangemax, int *status)

int fits_parse_rangell / ffrwrgll(char *rowlist, LONGLONG maxrows, int maxranges, >
   int *numranges, LONGLONG *rangemin, LONGLONG *rangemax, int *status)
```

23 Check that the Header fill bytes (if any) are all blank. These are the bytes that may follow END keyword and before the beginning of data unit, or the end of the HDU if there is no data unit.

```
int ffchfl(fitsfile *fptr, > int *status)
```

24 Check that the Data fill bytes (if any) are all zero (for IMAGE or BINARY Table HDU) or all blanks (for ASCII table HDU). These file bytes may be located after the last valid data byte in the HDU and before the physical end of the HDU.

```
int ffcdfl(fitsfile *fptr, > int *status)
```

25 Estimate the root-mean-squared (RMS) noise in an image. These routines are mainly for use with the Hcompress image compression algorithm. They return an estimate of the RMS noise in the background pixels of the image. This robust algorithm (written by Richard White, STScI) first attempts to estimate the RMS value as 1.68 times the median of the absolute differences between successive pixels in the image. If the median = 0, then the algorithm falls back to computing the RMS of the difference between successive pixels, after several N-sigma rejection cycles to remove extreme values. The input parameters are: the array of image pixel values (either float or short values), the number of values in the array, the value that is used to represent null pixels (enter a very large number if there are no null pixels).

```
int fits_rms_float (float fdata[], int npix, float in_null_value,
               > double *rms, int *status)
int fits_rms_short (short fdata[], int npix, short in_null_value,
               > double *rms, int *status)
```

26 Was CFITSIO compiled with the -D_REENTRANT directive so that it may be safely used in multi-threaded environments? The following function returns 1 if yes, 0 if no. Note, however, that even if the -D_REENTRANT directive was specified, this does not guarantee that the CFITSIO routines are thread-safe, because some compilers may not support this feature.

```
int fits_is_reentrant(void)
```

Chapter 6

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.

The following 2 sections describe these steps in more detail. There are also several example programs included with the CFITSIO distribution which illustrate how to use the iterator function.

6.1 The Iterator Work Function

The user-supplied iterator work function must have the following set of input parameters (the function can be given any desired name):

```
int user_fn( long totaln, long offset, long firstn, long nvalues,
             int narrays, iteratorCol *data,  void *userPointer )
```

- totaln – the total number of table rows or image pixels that will be passed to the work function during 1 or more iterations.

- offset – the offset applied to the first table row or image pixel to be passed to the work function. In other words, this is the number of rows or pixels that are skipped over before starting the iterations. If offset = 0, then all the table rows or image pixels will be passed to the work function.

- firstn – the number of the first table row or image pixel (starting with 1) that is being passed in this particular call to the work function.

- nvalues – the number of table rows or image pixels that are being passed in this particular call to the work function. nvalues will always be less than or equal to totaln and will have the same value on each iteration, except possibly on the last call which may have a smaller value.

- narrays – the number of arrays of data that are being passed to the work function. There is one array for each image or table column.

- *data – array of structures, one for each column or image. Each structure contains a pointer to the array of data as well as other descriptive parameters about that array.

- *userPointer – a user supplied pointer that can be used to pass ancillary information from the driver function to the work function. This pointer is passed to the CFITSIO iterator function which then passes it on to the work function without any modification. It may point to a single number, to an array of values, to a structure containing an arbitrary set of parameters of different types, or it may be a null pointer if it is not needed. The work function must cast this pointer to the appropriate data type before using it it.

The totaln, offset, narrays, data, and userPointer parameters are guaranteed to have the same value on each iteration. Only firstn, nvalues, and the arrays of data pointed to by the data structures may change on each iterative call to the work function.

Note that the iterator treats an image as a long 1-D array of pixels regardless of it's intrinsic dimensionality. The total number of pixels is just the product of the size of each dimension, and the order of the pixels is the same as the order that they are stored in the FITS file. If the work function needs to know the number and size of the image dimensions then these parameters can be passed via the userPointer structure.

The iteratorCol structure is currently defined as follows:

```
typedef struct  /* structure for the iterator function column information */
{
   /* structure elements required as input to fits_iterate_data: */

  fitsfile *fptr;       /* pointer to the HDU containing the column or image */
  int      colnum;      /* column number in the table; ignored for images    */
  char     colname[70]; /* name (TTYPEn) of the column; null for images      */
  int      datatype;    /* output data type (converted if necessary) */
  int      iotype;      /* type: InputCol, InputOutputCol, or OutputCol */

  /* output structure elements that may be useful for the work function: */

  void     *array;    /* pointer to the array (and the null value) */
  long     repeat;    /* binary table vector repeat value; set     */
                      /*     equal to 1 for images                 */
  long     tlmin;     /* legal minimum data value, if any          */
  long     tlmax;     /* legal maximum data value, if any          */
  char     unit[70];  /* physical unit string (BUNIT or TUNITn)    */
  char     tdisp[70]; /* suggested display format; null if none    */

} iteratorCol;
```

Instead of directly reading or writing the elements in this structure, it is recommended that programmers use the access functions that are provided for this purpose.

The first five elements in this structure must be initially defined by the driver routine before calling the iterator routine. The CFITSIO iterator routine uses this information to determine what column or array to pass to the work function, and whether the array is to be input to the work function, output from the work function, or both. The CFITSIO iterator function fills in the values of the remaining structure elements before passing it to the work function.

The array structure element is a pointer to the actual data array and it must be cast to the correct data type before it is used. The 'repeat' structure element give the number of data values in each row of the table, so that the total number of data values in the array is given by repeat * nvalues. In the case of image arrays and ASCII tables, repeat will always be equal to 1. When the data type is a character string, the array pointer is actually a pointer to an array of string pointers (i.e., char **array). The other output structure elements are provided for convenience in case that information is needed within the work function. Any other information may be passed from the driver routine to the work function via the userPointer parameter.

Upon completion, the work routine must return an integer status value, with 0 indicating success and any other value indicating an error which will cause the iterator function to immediately exit at that point. Return status values in the range 1 – 1000 should be avoided since these are reserved for use by CFITSIO. A return status value of -1 may be used to force the CFITSIO iterator function to stop at that point and return control to the driver routine after writing any output arrays to the FITS file. CFITSIO does not considered this to be an error condition, so any further processing by the application program will continue normally.

6.2 The Iterator Driver Function

The iterator driver function must open the necessary FITS files and position them to the correct HDU. It must also initialize the following parameters in the iteratorCol structure (defined above) for each column or image before calling the CFITSIO iterator function. Several 'constructor' routines are provided in CFITSIO for this purpose.

- *fptr – The fitsfile pointer to the table or image.
- colnum – the number of the column in the table. This value is ignored in the case of images. If colnum equals 0, then the column name will be used to identify the column to be passed to the work function.
- colname – the name (TTYPEn keyword) of the column. This is only required if colnum = 0 and is ignored for images.
- datatype – The desired data type of the array to be passed to the work function. For numerical data the data type does not need to be the same as the actual data type in the FITS file, in which case CFITSIO will do the conversion. Allowed values are: TSTRING, TLOGICAL, TBYTE, TSBYTE, TSHORT, TUSHORT, TINT, TLONG, TULONG, TFLOAT, TDOUBLE. If the input value of data type equals 0, then the existing data type of the column or image will be used without any conversion.
- iotype – defines whether the data array is to be input to the work function (i.e, read from the FITS file), or output from the work function (i.e., written to the FITS file) or both. Allowed values are InputCol, OutputCol, or InputOutputCol. Variable-length array columns are supported as InputCol or InputOutputCol types, but may not be used for an OutputCol type.

After the driver routine has initialized all these parameters, it can then call the CFITSIO iterator function:

```
int fits_iterate_data(int narrays, iteratorCol *data, long offset,
    long nPerLoop, int (*workFn)( ), void *userPointer, int *status);
```

- narrays – the number of columns or images that are to be passed to the work function.

- *data – pointer to array of structures containing information about each column or image.
- offset – if positive, this number of rows at the beginning of the table (or pixels in the image) will be skipped and will not be passed to the work function.
- nPerLoop - specifies the number of table rows (or number of image pixels) that are to be passed to the work function on each iteration. If nPerLoop = 0 then CFITSIO will calculate the optimum number for greatest efficiency. If nPerLoop is negative, then all the rows or pixels will be passed at one time, and the work function will only be called once. If any variable length arrays are being processed, then the nPerLoop value is ignored, and the iterator will always process one row of the table at a time.
- *workFn - the name (actually the address) of the work function that is to be called by fits_iterate_data.
- *userPointer - this is a user supplied pointer that can be used to pass ancillary information from the driver routine to the work function. It may point to a single number, an array, or to a structure containing an arbitrary set of parameters.
- *status - The CFITSIO error status. Should = 0 on input; a non-zero output value indicates an error.

When fits_iterate_data is called it first allocates memory to hold all the requested columns of data or image pixel arrays. It then reads the input data from the FITS tables or images into the arrays then passes the structure with pointers to these data arrays to the work function. After the work function returns, the iterator function writes any output columns of data or images back to the FITS files. It then repeats this process for any remaining sets of rows or image pixels until it has processed the entire table or image or until the work function returns a non-zero status value. The iterator then frees the memory that it initially allocated and returns control to the driver routine that called it.

6.3 Guidelines for Using the Iterator Function

The totaln, offset, firstn, and nvalues parameters that are passed to the work function are useful for determining how much of the data has been processed and how much remains left to do. On the very first call to the work function firstn will be equal to offset + 1; the work function may need to perform various initialization tasks before starting to process the data. Similarly, firstn + nvalues - 1 will be equal to totaln on the last iteration, at which point the work function may need to perform some clean up operations before exiting for the last time. The work function can also force an early termination of the iterations by returning a status value = -1.

The narrays and iteratorCol.datatype arguments allow the work function to double check that the number of input arrays and their data types have the expected values. The iteratorCol.fptr and iteratorCol.colnum structure elements can be used if the work function needs to read or write the values of other keywords in the FITS file associated with the array. This should generally only be done during the initialization step or during the clean up step after the last set of data has been processed. Extra FITS file I/O during the main processing loop of the work function can seriously degrade the speed of the program.

If variable-length array columns are being processed, then the iterator will operate on one row of the table at a time. In this case the the repeat element in the interatorCol structure will be set equal to the number of elements in the current row that is being processed.

One important feature of the iterator is that the first element in each array that is passed to the work function gives the value that is used to represent null or undefined values in the array. The real data then begins with the second element of the array (i.e., array[1], not array[0]). If the first array element is equal to zero, then this indicates that all the array elements have defined values and there are no undefined values. If array[0] is not equal to zero, then this indicates that some of the data values are undefined and this value (array[0]) is used to represent them. In the case of output arrays (i.e., those arrays that will be written back to the FITS file by the iterator function after the work function exits) the work function must set the first array element to the desired null value if necessary, otherwise the first element should be set to zero to indicate that there are no null values in the output array. CFITSIO defines 2 values, FLOATNULLVALUE and DOUBLENULLVALUE, that can be used as default null values for float and double data types, respectively. In the case of character string data types, a null string is always used to represent undefined strings.

In some applications it may be necessary to recursively call the iterator function. An example of this is given by one of the example programs that is distributed with CFITSIO: it first calls a work function that writes out a 2D histogram image. That work function in turn calls another work function that reads the 'X' and 'Y' columns in a table to calculate the value of each 2D histogram image pixel. Graphically, the program structure can be described as:

```
driver --> iterator --> work1_fn --> iterator --> work2_fn
```

Finally, it should be noted that the table columns or image arrays that are passed to the work function do not all have to come from the same FITS file and instead may come from any combination of sources as long as they have the same length. The length of the first table column or image array is used by the iterator if they do not all have the same length.

6.4 Complete List of Iterator Routines

All of the iterator routines are listed below. Most of these routines do not have a corresponding short function name.

1 Iterator 'constructor' functions that set the value of elements in the iteratorCol structure that define the columns or arrays. These set the fitsfile pointer, column name, column number, datatype, and iotype, respectively. The last 2 routines allow all the parameters to be set with one function call (one supplies the column name, the other the column number).

```
int fits_iter_set_file(iteratorCol *col, fitsfile *fptr);

int fits_iter_set_colname(iteratorCol *col, char *colname);

int fits_iter_set_colnum(iteratorCol *col, int colnum);
```

```
int fits_iter_set_datatype(iteratorCol *col, int datatype);

int fits_iter_set_iotype(iteratorCol *col, int iotype);

int fits_iter_set_by_name(iteratorCol *col, fitsfile *fptr,
        char *colname, int datatype,  int iotype);

int fits_iter_set_by_num(iteratorCol *col, fitsfile *fptr,
        int colnum, int datatype,  int iotype);
```

2 Iterator 'accessor' functions that return the value of the element in the iteratorCol structure that describes a particular data column or array

```
fitsfile * fits_iter_get_file(iteratorCol *col);

char * fits_iter_get_colname(iteratorCol *col);

int fits_iter_get_colnum(iteratorCol *col);

int fits_iter_get_datatype(iteratorCol *col);

int fits_iter_get_iotype(iteratorCol *col);

void * fits_iter_get_array(iteratorCol *col);

long fits_iter_get_tlmin(iteratorCol *col);

long fits_iter_get_tlmax(iteratorCol *col);

long fits_iter_get_repeat(iteratorCol *col);

char * fits_iter_get_tunit(iteratorCol *col);

char * fits_iter_get_tdisp(iteratorCol *col);
```

3 The CFITSIO iterator function

```
int fits_iterate_data(int narrays,  iteratorCol *data, long offset,
          long nPerLoop,
          int (*workFn)( long totaln, long offset, long firstn,
                         long nvalues, int narrays, iteratorCol *data,
                         void *userPointer),
          void *userPointer,
          int *status);
```

Chapter 7

World Coordinate System Routines

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 a few 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 keywords contained in the FITS file headers. One convenient way to pass this information to the external library is to use the fits_hdr2str routine in CFITSIO (defined below) to copy the header keywords into one long string, and then pass this string to an interface routine in the external library that will extract the necessary WCS information (e.g., the 'wcspih' routine in the WCSLIB library and the 'astFitsChan' and 'astPutCards' functions in the AST library).

1 Concatenate the header keywords in the CHDU into a single long string of characters. Each 80-character fixed-length keyword record is appended to the output character string, in order, with no intervening separator or terminating characters. The last header record is terminated with a NULL character. This routine allocates memory for the returned character array, so the calling program must free the memory when finished.

There are 2 related routines: fits_hdr2str simply concatenates all the existing keywords in the header; fits_convert_hdr2str is similar, except that if the CHDU is a tile compressed image (stored in a binary table) then it will first convert that header back to that of a normal FITS image before concatenating the keywords.

Selected keywords may be excluded from the returned character string. If the second parameter (nocomments) is TRUE (nonzero) then any COMMENT, HISTORY, or blank keywords in the header will not be copied to the output string.

The 'exclist' parameter may be used to supply a list of keywords that are to be excluded from the output character string. Wild card characters (*, ?, and #) may be used in the excluded keyword names. If no additional keywords are to be excluded, then set nexc = 0 and specify NULL for the the **exclist parameter.

```
int fits_hdr2str
    (fitsfile *fptr, int nocomments, char **exclist, int nexc,
    > char **header, int *nkeys, int *status)

int fits_convert_hdr2str / ffcnvthdr2str
    (fitsfile *fptr, int nocomments, char **exclist, int nexc,
    > char **header, int *nkeys, int *status)
```

2 The following CFITSIO routine is specifically designed for use in conjunction with the WCSLIB library. It is not expected that applications programmers will call this routine directly, but it is documented here for completeness. This routine extracts arrays from a binary table that contain WCS information using the -TAB table lookup convention. See the documentation provided with the WCSLIB library for more information.

```
int fits_read_wcstab
     (fitsfile *fptr, int nwtb, wtbarr *wtb, int *status);
```

7.1 Self-contained WCS Routines

The following 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 in the previous section.

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 the basic set of 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 fits_pix_to_world and fits_world_to_pix routines 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.

The first routine (ffgics) returns the primary WCS, whereas the second routine returns the particular version of the WCS specified by the 'version' parameter, which much be a character ranging from 'A' to 'Z' (or a blank character, which is equivalent to calling ffgics).

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 = APPROX_WCS_KEY, 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.

```
int fits_read_img_coord / ffgics
    (fitsfile *fptr, > double *xrefval, double *yrefval,
     double *xrefpix, double *yrefpix, double *xinc, double *yinc,
     double *rot, char *coordtype, int *status)

int fits_read_img_coord_version / ffgicsa
    (fitsfile *fptr, char version, > double *xrefval, double *yrefval,
     double *xrefpix, double *yrefpix, double *xinc, double *yinc,
     double *rot, char *coordtype, int *status)
```

2 Get the values of 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 (as in the Event List table format that is often used by high energy astrophysics missions). These values may then be passed to the fits_pix_to_world and fits_world_to_pix routines that perform the coordinate transformations.

```
int fits_read_tbl_coord / ffgtcs
    (fitsfile *fptr, int xcol, int ycol, > double *xrefval,
     double *yrefval, double *xrefpix, double *yrefpix, double *xinc,
     double *yinc, double *rot, char *coordtype, int *status)
```

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

```
int fits_pix_to_world / ffwldp
    (double xpix, double ypix, double xrefval, double yrefval,
     double xrefpix, double yrefpix, double xinc, double yinc,
     double rot, char *coordtype, > double *xpos, double *ypos,
     int *status)
```

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

```
int fits_world_to_pix / ffxypx
```

```
(double xpos, double ypos, double xrefval, double yrefval,
 double xrefpix, double yrefpix, double xinc, double yinc,
 double rot, char *coordtype, > double *xpix, double *ypix,
 int *status)
```

Chapter 8

Hierarchical Grouping Routines

These functions allow for the creation and manipulation of FITS HDU Groups, as defined in "A Hierarchical Grouping Convention for FITS" by Jennings, Pence, Folk and Schlesinger:

http://fits.gsfc.nasa.gov/group.html

A group is a collection of HDUs whose association is defined by a *grouping table*. HDUs which are part of a group are referred to as *member HDUs* or simply as *members*. Grouping table member HDUs may themselves be grouping tables, thus allowing for the construction of open-ended hierarchies of HDUs.

Grouping tables contain one row for each member HDU. The grouping table columns provide identification information that allows applications to reference or "point to" the member HDUs. Member HDUs are expected, but not required, to contain a set of GRPIDn/GRPLCn keywords in their headers for each grouping table that they are referenced by. In this sense, the GRPIDn/GRPLCn keywords "link" the member HDU back to its Grouping table. Note that a member HDU need not reside in the same FITS file as its grouping table, and that a given HDU may be referenced by up to 999 grouping tables simultaneously.

Grouping tables are implemented as FITS binary tables with up to six pre-defined column TTYPEn values: 'MEMBER_XTENSION', 'MEMBER_NAME', 'MEMBER_VERSION', 'MEMBER_POSITION', 'MEMBER_URI_TYPE' and 'MEMBER_LOCATION'. The first three columns allow member HDUs to be identified by reference to their XTENSION, EXTNAME and EXTVER keyword values. The fourth column allows member HDUs to be identified by HDU position within their FITS file. The last two columns identify the FITS file in which the member HDU resides, if different from the grouping table FITS file.

Additional user defined "auxiliary" columns may also be included with any grouping table. When a grouping table is copied or modified the presence of auxiliary columns is always taken into account by the grouping support functions; however, the grouping support functions cannot directly make use of this data.

If a grouping table column is defined but the corresponding member HDU information is unavailable then a null value of the appropriate data type is inserted in the column field. Integer columns (MEMBER_POSITION, MEMBER_VERSION) are defined with a TNULLn value of zero (0). Character field columns (MEMBER_XTENSION, MEMBER_NAME, MEMBER_URI_TYPE, MEMBER_LOCATION) utilize an ASCII null character to denote a null field value.

The grouping support functions belong to two basic categories: those that work with grouping table HDUs (ffgt**) and those that work with member HDUs (ffgm**). Two functions, fits_copy_group() and fits_remove_group(), have the option to recursively copy/delete entire groups. Care should be taken when employing these functions in recursive mode as poorly defined groups could cause unpredictable results. The problem of a grouping table directly or indirectly referencing itself (thus creating an infinite loop) is protected against; in fact, neither function will attempt to copy or delete an HDU twice.

8.1 Grouping Table Routines

1 Create (append) a grouping table at the end of the current FITS file pointed to by fptr. The grpname parameter provides the grouping table name (GRPNAME keyword value) and may be set to NULL if no group name is to be specified. The grouptype parameter specifies the desired structure of the grouping table and may take on the values: GT_ID_ALL_URI (all columns created), GT_ID_REF (ID by reference columns), GT_ID_POS (ID by position columns), GT_ID_ALL (ID by reference and position columns), GT_ID_REF_URI (ID by reference and FITS file URI columns), and GT_ID_POS_URI (ID by position and FITS file URI columns).

```
int fits_create_group / ffgtcr
    (fitsfile *fptr, char *grpname, int grouptype, > int *status)
```

2 Create (insert) a grouping table just after the CHDU of the current FITS file pointed to by fptr. All HDUs below the the insertion point will be shifted downwards to make room for the new HDU. The grpname parameter provides the grouping table name (GRPNAME keyword value) and may be set to NULL if no group name is to be specified. The grouptype parameter specifies the desired structure of the grouping table and may take on the values: GT_ID_ALL_URI (all columns created), GT_ID_REF (ID by reference columns), GT_ID_POS (ID by position columns), GT_ID_ALL (ID by reference and position columns), GT_ID_REF_URI (ID by reference and FITS file URI columns), and GT_ID_POS_URI (ID by position and FITS file URI columns) .

```
int fits_insert_group / ffgtis
    (fitsfile *fptr, char *grpname, int grouptype, > int *status)
```

3 Change the structure of an existing grouping table pointed to by gfptr. The grouptype parameter (see fits_create_group() for valid parameter values) specifies the new structure of the grouping table. This function only adds or removes grouping table columns, it does not add or delete group members (i.e., table rows). If the grouping table already has the desired structure then no operations are performed and function simply returns with a (0) success status code. If the requested structure change creates new grouping table columns, then the column values for all existing members will be filled with the null values appropriate to the column type.

```
int fits_change_group / ffgtch
    (fitsfile *gfptr, int grouptype, > int *status)
```

4 Remove the group defined by the grouping table pointed to by gfptr, and optionally all the group member HDUs. The rmopt parameter specifies the action to be taken for all members of the group defined by the grouping table. Valid values are: OPT_RM_GPT (delete only the grouping table) and OPT_RM_ALL (recursively delete all HDUs that belong to the group). Any groups containing the grouping table gfptr as a member are updated, and if rmopt == OPT_RM_GPT all members have their GRPIDn and GRPLCn keywords updated accordingly. If rmopt == OPT_RM_ALL, then other groups that contain the deleted members of gfptr are updated to reflect the deletion accordingly.

```
int fits_remove_group / ffgtrm
    (fitsfile *gfptr, int rmopt, > int *status)
```

5 Copy (append) the group defined by the grouping table pointed to by infptr, and optionally all group member HDUs, to the FITS file pointed to by outfptr. The cpopt parameter specifies the action to be taken for all members of the group infptr. Valid values are: OPT_GCP_GPT (copy only the grouping table) and OPT_GCP_ALL (recursively copy ALL the HDUs that belong to the group defined by infptr). If the cpopt == OPT_GCP_GPT then the members of infptr have their GRPIDn and GRPLCn keywords updated to reflect the existence of the new grouping table outfptr, since they now belong to the new group. If cpopt == OPT_GCP_ALL then the new grouping table outfptr only contains pointers to the copied member HDUs and not the original member HDUs of infptr. Note that, when cpopt == OPT_GCP_ALL, all members of the group defined by infptr will be copied to a single FITS file pointed to by outfptr regardless of their file distribution in the original group.

```
int fits_copy_group / ffgtcp
    (fitsfile *infptr, fitsfile *outfptr, int cpopt, > int *status)
```

6 Merge the two groups defined by the grouping table HDUs infptr and outfptr by combining their members into a single grouping table. All member HDUs (rows) are copied from infptr to outfptr. If mgopt == OPT_MRG_COPY then infptr continues to exist unaltered after the merge. If the mgopt == OPT_MRG_MOV then infptr is deleted after the merge. In both cases, the GRPIDn and GRPLCn keywords of the member HDUs are updated accordingly.

```
int fits_merge_groups / ffgtmg
    (fitsfile *infptr, fitsfile *outfptr, int mgopt, > int *status)
```

7 "Compact" the group defined by grouping table pointed to by gfptr. The compaction is achieved by merging (via fits_merge_groups()) all direct member HDUs of gfptr that are themselves grouping tables. The cmopt parameter defines whether the merged grouping table HDUs remain after merging (cmopt == OPT_CMT_MBR) or if they are deleted after merging (cmopt == OPT_CMT_MBR_DEL). If the grouping table contains no direct member HDUs that are themselves grouping tables then this function does nothing. Note that this function is not recursive, i.e., only the direct member HDUs of gfptr are considered for merging.

```
int fits_compact_group / ffgtcm
    (fitsfile *gfptr, int cmopt, > int *status)
```

8 Verify the integrity of the grouping table pointed to by gfptr to make sure that all group members are accessible and that all links to other grouping tables are valid. The firstfailed parameter returns the member ID (row number) of the first member HDU to fail verification (if positive value) or the first group link to fail (if negative value). If gfptr is successfully verified then firstfailed contains a return value of 0.

```
int fits_verify_group / ffgtvf
    (fitsfile *gfptr, > long *firstfailed, int *status)
```

9 Open a grouping table that contains the member HDU pointed to by mfptr. The grouping table to open is defined by the grpid parameter, which contains the keyword index value of the GRPIDn/GRPLCn keyword(s) that link the member HDU mfptr to the grouping table. If the grouping table resides in a file other than the member HDUs file then an attempt is first made to open the file readwrite, and failing that readonly. A pointer to the opened grouping table HDU is returned in gfptr.

Note that it is possible, although unlikely and undesirable, for the GRPIDn/GRPLCn keywords in a member HDU header to be non-continuous, e.g., GRPID1, GRPID2, GRPID5, GRPID6. In such cases, the grpid index value specified in the function call shall identify the (grpid)th GRPID value. In the above example, if grpid == 3, then the group specified by GRPID5 would be opened.

```
int fits_open_group / ffgtop
    (fitsfile *mfptr, int group, > fitsfile **gfptr, int *status)
```

10 Add a member HDU to an existing grouping table pointed to by gfptr. The member HDU may either be pointed to mfptr (which must be positioned to the member HDU) or, if mfptr == NULL, identified by the hdupos parameter (the HDU position number, Primary array == 1) if both the grouping table and the member HDU reside in the same FITS file. The new member HDU shall have the appropriate GRPIDn and GRPLCn keywords created in its header. Note that if the member HDU is already a member of the group then it will not be added a second time.

```
int fits_add_group_member / ffgtam
    (fitsfile *gfptr, fitsfile *mfptr, int hdupos, > int *status)
```

8.2 Group Member Routines

1 Return the number of member HDUs in a grouping table gfptr. The number member HDUs is just the NAXIS2 value (number of rows) of the grouping table.

```
int fits_get_num_members / ffgtnm
    (fitsfile *gfptr, > long *nmembers, int *status)
```

2 Return the number of groups to which the HDU pointed to by mfptr is linked, as defined by the number of GRPIDn/GRPLCn keyword records that appear in its header. Note that each time this function is called, the indices of the GRPIDn/GRPLCn keywords are checked to make sure they are continuous (ie no gaps) and are re-enumerated to eliminate gaps if found.

```
int fits_get_num_groups / ffgmng
    (fitsfile *mfptr, > long *nmembers, int *status)
```

3 Open a member of the grouping table pointed to by gfptr. The member to open is identified by its row number within the grouping table as given by the parameter 'member' (first member == 1) . A fitsfile pointer to the opened member HDU is returned as mfptr. Note that if the member HDU resides in a FITS file different from the grouping table HDU then the member file is first opened readwrite and, failing this, opened readonly.

```
int fits_open_member / ffgmop
    (fitsfile *gfptr, long member, > fitsfile **mfptr, int *status)
```

4 Copy (append) a member HDU of the grouping table pointed to by gfptr. The member HDU is identified by its row number within the grouping table as given by the parameter 'member' (first member == 1). The copy of the group member HDU will be appended to the FITS file pointed to by mfptr, and upon return mfptr shall point to the copied member HDU. The cpopt parameter may take on the following values: OPT_MCP_ADD which adds a new entry in gfptr for the copied member HDU, OPT_MCP_NADD which does not add an entry in gfptr for the copied member, and OPT_MCP_REPL which replaces the original member entry with the copied member entry.

```
int fits_copy_member / ffgmcp
    (fitsfile *gfptr, fitsfile *mfptr, long member, int cpopt, > int *status)
```

5 Transfer a group member HDU from the grouping table pointed to by infptr to the grouping table pointed to by outfptr. The member HDU to transfer is identified by its row number within infptr as specified by the parameter 'member' (first member == 1). If tfopt == OPT_MCP_ADD then the member HDU is made a member of outfptr and remains a member of infptr. If tfopt == OPT_MCP_MOV then the member HDU is deleted from infptr after the transfer to outfptr.

```
int fits_transfer_member / ffgmtf
    (fitsfile *infptr, fitsfile *outfptr, long member, int tfopt,
     > int *status)
```

6 Remove a member HDU from the grouping table pointed to by gfptr. The member HDU to be deleted is identified by its row number in the grouping table as specified by the parameter 'member' (first member == 1). The rmopt parameter may take on the following values: OPT_RM_ENTRY which removes the member HDU entry from the grouping table and updates the member's GRPIDn/GRPLCn keywords, and OPT_RM_MBR which removes the member HDU entry from the grouping table and deletes the member HDU itself.

```
int fits_remove_member / ffgmrm
    (fitsfile *fptr, long member, int rmopt, > int *status)
```

Chapter 9

Specialized CFITSIO Interface Routines

The basic interface routines described previously are recommended for most uses, but the routines described in this chapter are also available if necessary. Some of these routines perform more specialized function that cannot easily be done with the basic interface routines while others duplicate the functionality of the basic routines but have a slightly different calling sequence. See Appendix B for the definition of each function parameter.

9.1 FITS File Access Routines

1 Open an existing FITS file residing in core computer memory. This routine is analogous to fits_open_file. The 'filename' is currently ignored by this routine and may be any arbitrary string. In general, the application must have preallocated an initial block of memory to hold the FITS file prior to calling this routine: 'memptr' points to the starting address and 'memsize' gives the initial size of the block of memory. 'mem_realloc' is a pointer to an optional function that CFITSIO can call to allocate additional memory, if needed (only if mode = READWRITE), and is modeled after the standard C 'realloc' function; a null pointer may be given if the initial allocation of memory is all that will be required (e.g., if the file is opened with mode = READONLY). The 'deltasize' parameter may be used to suggest a minimum amount of additional memory that should be allocated during each call to the memory reallocation function. By default, CFITSIO will reallocate enough additional space to hold the entire currently defined FITS file (as given by the NAXISn keywords) or 1 FITS block (= 2880 bytes), which ever is larger. Values of deltasize less than 2880 will be ignored. Since the memory reallocation operation can be computationally expensive, allocating a larger initial block of memory, and/or specifying a larger deltasize value may help to reduce the number of reallocation calls and make the application program run faster. Note that values of the memptr and memsize pointers will be updated by CFITSIO if the location or size of the FITS file in memory should change as a result of allocating more memory.

```
int fits_open_memfile / ffomem
    (fitsfile **fptr, const char *filename, int mode, void **memptr,
```

```
        size_t *memsize, size_t deltasize,
        void *(*mem_realloc)(void *p, size_t newsize), int *status)
```

2 Create a new FITS file residing in core computer memory. This routine is analogous to fits_create_file. In general, the application must have preallocated an initial block of memory to hold the FITS file prior to calling this routine: 'memptr' points to the starting address and 'memsize' gives the initial size of the block of memory. 'mem_realloc' is a pointer to an optional function that CFITSIO can call to allocate additional memory, if needed, and is modeled after the standard C 'realloc' function; a null pointer may be given if the initial allocation of memory is all that will be required. The 'deltasize' parameter may be used to suggest a minimum amount of additional memory that should be allocated during each call to the memory reallocation function. By default, CFITSIO will reallocate enough additional space to hold 1 FITS block (= 2880 bytes) and values of deltasize less than 2880 will be ignored. Since the memory reallocation operation can be computationally expensive, allocating a larger initial block of memory, and/or specifying a larger deltasize value may help to reduce the number of reallocation calls and make the application program run faster. Note that values of the memptr and memsize pointers will be updated by CFITSIO if the location or size of the FITS file in memory should change as a result of allocating more memory.

```
int fits_create_memfile / ffimem
     (fitsfile **fptr, void **memptr,
      size_t *memsize, size_t deltasize,
      void *(*mem_realloc)(void *p, size_t newsize), int *status)
```

3 Reopen a FITS file that was previously opened with fits_open_file or fits_create_file. The new fitsfile pointer 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 fits_open_file routine (above) automatically detects cases where a previously opened file is being opened again, and then internally call fits_reopen_file, so programs should rarely need to explicitly call this routine.

```
int fits_reopen_file / ffreopen
     (fitsfile *openfptr, fitsfile **newfptr, > int *status)
```

4 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 pointer is null, then this routine behaves the same as fits_create_file. The currently supported format of the ASCII template file is described under the fits_parse_template routine (in the general Utilities section)

```
int fits_create_template / fftplt
     (fitsfile **fptr, char *filename, char *tpltfile > int *status)
```

5 Parse the input filename or URL into its component parts, namely:

- 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,
- the column specifier,
- and the image pixel filtering specifier.

A null pointer (0) may be be specified for any of the output string arguments that are not needed. Null strings will be returned for any components that are not present in the input file name. The calling routine must allocate sufficient memory to hold the returned character strings. Allocating the string lengths equal to FLEN_FILENAME is guaranteed to be safe. These routines are mainly for internal use by other CFITSIO routines.

```
int fits_parse_input_url / ffiurl
    (char *filename, > char *filetype, char *infile, char *outfile, char
     *extspec, char *filter, char *binspec, char *colspec, int *status)

int fits_parse_input_filename / ffifile
    (char *filename, > char *filetype, char *infile, char *outfile, char
     *extspec, char *filter, char *binspec, char *colspec, char *pixspec,
     int *status)
```

6 Parse the input filename and return the HDU number that would be moved to if the file were opened with fits_open_file. 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. CFITSIO 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 fits_open_file, then use fits_get_hdu_num to determine which HDU in the file has been opened, rather than calling fits_parse_input_url followed by a call to fits_open_file.

```
int fits_parse_extnum / ffextn
   (char *filename, > int *hdunum, int *status)
```

7 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. The calling routine must allocate sufficient memory to hold the returned rootname character string. Allocating the length equal to FLEN_FILENAME is guaranteed to be safe.

```
int fits_parse_rootname / ffrtnm
    (char *filename, > char *rootname, int *status);
```

8 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)
```

```
int fits_file_exists / ffexist
    (char *filename, > int *exists, int *status);
```

9 Flush any internal buffers of data to the output FITS file. These routines rarely need to be called, but can be useful in cases where other processes need to access the same FITS file in real time, either on disk or in memory. These routines also help to ensure that if the application program subsequently aborts then the FITS file will have been closed properly. The first routine, fits_flush_file is more rigorous and completely closes, then reopens, the current HDU, before flushing the internal buffers, thus ensuring that the output FITS file is identical to what would be produced if the FITS was closed at that point (i.e., with a call to fits_close_file). The second routine, fits_flush_buffer simply flushes the internal CFITSIO buffers of data to the output FITS file, without updating and closing the current HDU. This is much faster, but there may be circumstances where the flushed file does not completely reflect the final state of the file as it will exist when the file is actually closed.

A typical use of these routines would be to flush the state of a FITS table to disk after each row of the table is written. It is recommend that fits_flush_file be called after the first row is written, then fits_flush_buffer may be called after each subsequent row is written. Note that this latter routine will not automatically update the NAXIS2 keyword which records the number of rows of data in the table, so this keyword must be explicitly updated by the application program after each row is written.

```
int fits_flush_file / ffflus
    (fitsfile *fptr, > int *status)

int fits_flush_buffer / ffflsh
    (fitsfile *fptr, 0, > int *status)

    (Note:  The second argument must be 0).
```

9.2 HDU Access Routines

1 Get the byte offsets in the FITS file to the start of the header and the start and end of the data in the CHDU. The difference between headstart and dataend equals the size of the CHDU. If the CHDU is the last HDU in the file, then dataend is also equal to the size of the entire FITS file. Null pointers may be input for any of the address parameters if their values are not needed.

```
int fits_get_hduaddr / ffghad  (only supports files up to 2.1 GB in size)
     (fitsfile *fptr, > long *headstart, long *datastart, long *dataend,
      int *status)

int fits_get_hduaddrll / ffghadll  (supports large files)
     (fitsfile *fptr, > LONGLONG *headstart, LONGLONG *datastart,
      LONGLONG *dataend, int *status)
```

2 Create (append) a new empty HDU at the end of the FITS file. This is now the CHDU but it is completely empty and has no header keywords. It is recommended that fits_create_img or fits_create_tbl be used instead of this routine.

```
int fits_create_hdu / ffcrhd
    (fitsfile *fptr, > int *status)
```

3 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 = PREPEND_PRIMARY prior to calling the routine. In this case the old primary array will be converted to an IMAGE extension. The new extension (or primary array) will become the CHDU. Refer to Chapter 9 for a list of pre-defined bitpix values.

```
int fits_insert_img / ffiimg
    (fitsfile *fptr, int bitpix, int naxis, long *naxes, > int *status)

int fits_insert_imgll / ffiimgll
    (fitsfile *fptr, int bitpix, int naxis, LONGLONG *naxes, > int *status)
```

4 Insert a new ASCII or 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 table extension will simply be appended to the end of the file. If the FITS file is currently empty then this routine will create a dummy primary array before appending the table to it. The new extension will become the CHDU. The tunit and extname parameters are optional and a null pointer may be given if they are not defined. When inserting an ASCII table with fits_insert_atbl, a null pointer may given for the *tbcol parameter in which case each column of the table will be separated by a single space character. Similarly, if the input value of rowlen is 0, then CFITSIO will calculate the default rowlength based on the tbcol and ttype values. Under normal circumstances, the nrows paramenter should have a value of 0; CFITSIO will automatically update the number of rows as data is written to the table. When inserting a binary table with fits_insert_btbl, if there are following extensions in the file and if the table contains variable length array columns then pcount must specify the expected final size of the data heap, otherwise pcount must = 0.

```
int fits_insert_atbl / ffitab
    (fitsfile *fptr, LONGLONG rowlen, LONGLONG nrows, int tfields, char *ttype[],
     long *tbcol, char *tform[], char *tunit[], char *extname, > int *status)

int fits_insert_btbl / ffibin
    (fitsfile *fptr, LONGLONG nrows, int tfields, char **ttype,
    char **tform, char **tunit, char *extname, long pcount, > int *status)
```

5 Modify the size, dimensions, and/or data type 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 the new image.

```
int fits_resize_img / ffrsim
    (fitsfile *fptr, int bitpix, int naxis, long *naxes, > int *status)

int fits_resize_imgll / ffrsimll
    (fitsfile *fptr, int bitpix, int naxis, LONGLONG *naxes, > int *status)
```

6 Copy the data (and not the header) from the CHDU associated with infptr to the CHDU associated with outfptr. This will overwrite any data previously in the output CHDU. This low level routine is used by fits_copy_hdu, but it may also be useful in certain application programs that want to copy the data from one FITS file to another but also want to modify the header keywords. The required FITS header keywords which define the structure of the HDU must be written to the output CHDU before calling this routine.

```
int fits_copy_data / ffcpdt
    (fitsfile *infptr, fitsfile *outfptr, > int *status)
```

7 Read or write a specified number of bytes starting at the specified byte offset from the start of the extension data unit. These low level routine are intended mainly for accessing the data in non-standard, conforming extensions, and should not be used for standard IMAGE, TABLE, or BINTABLE extensions.

```
int fits_read_ext / ffgextn
    (fitsfile *fptr, LONGLONG offset, LONGLONG nbytes, void *buffer)
int fits_write_ext / ffpextn
    (fitsfile *fptr, LONGLONG offset, LONGLONG nbytes, void *buffer)
```

8 This routine forces CFITSIO to rescan the current header keywords that define the structure of the HDU (such as the NAXIS and BITPIX keywords) so that it reinitializes the internal buffers that describe the HDU structure. This routine is useful for reinitializing the structure of an HDU if any of the required keywords (e.g., NAXISn) have been modified. In practice it should rarely be necessary to call this routine because CFITSIO internally calls it in most situations.

```
int fits_set_hdustruc / ffrdef
    (fitsfile *fptr, > int *status)   (DEPRECATED)
```

9.3 Specialized Header Keyword Routines

9.3.1 Header Information Routines

1 Reserve space in the CHU for MOREKEYS more header keywords. This routine may be called to allocate space for additional keywords at the time the header is created (prior to writing any data). CFITSIO can dynamically add more space to the header when needed, however it is more efficient to preallocate the required space if the size is known in advance.

```
int fits_set_hdrsize / ffhdef
    (fitsfile *fptr, int morekeys, > int *status)
```

2 Return the number of keywords in the header (not counting the END keyword) and the current position in the header. The position is the number of the keyword record that will be read next (or one greater than the position of the last keyword that was read). A value of 1 is returned if the pointer is positioned at the beginning of the header.

```
int fits_get_hdrpos / ffghps
    (fitsfile *fptr, > int *keysexist, int *keynum, int *status)
```

9.3.2 Read and Write the Required Keywords

1 Write the required extension header keywords into the CHU. These routines are not required, and instead the appropriate header may be constructed by writing each individual keyword in the proper sequence.

The simpler fits_write_imghdr routine is equivalent to calling fits_write_grphdr with the default values of simple = TRUE, pcount = 0, gcount = 1, and extend = TRUE. The 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. It is recommended that fits_create_image or fits_create_tbl be used instead of these routines to write the required header keywords. The general fits_write_exthdr routine may be used to write the header of any conforming FITS extension.

```
int fits_write_imghdr / ffphps
    (fitsfile *fptr, int bitpix, int naxis, long *naxes, > int *status)

int fits_write_imghdrll / ffphpsll
    (fitsfile *fptr, int bitpix, int naxis, LONGLONG *naxes, > int *status)

int fits_write_grphdr / ffphpr
    (fitsfile *fptr, int simple, int bitpix, int naxis, long *naxes,
     LONGLONG pcount, LONGLONG gcount, int extend, > int *status)

int fits_write_grphdrll / ffphprll
    (fitsfile *fptr, int simple, int bitpix, int naxis, LONGLONG *naxes,
     LONGLONG pcount, LONGLONG gcount, int extend, > int *status)

int fits_write_exthdr /ffphext
    (fitsfile *fptr, char *xtension, int bitpix, int naxis, long *naxes,
     LONGLONG pcount, LONGLONG gcount, > int *status)
```

2 Write the ASCII table header keywords into the CHU. The optional TUNITn and EXTNAME keywords are written only if the input pointers are not null. A null pointer may given for the *tbcol parameter in which case a single space will be inserted between each column of the table. Similarly, if rowlen is given = 0, then CFITSIO will calculate the default rowlength based on the tbcol and ttype values.

```
int fits_write_atblhdr / ffphtb
    (fitsfile *fptr, LONGLONG rowlen, LONGLONG nrows, int tfields, char **ttype,
     long *tbcol, char **tform, char **tunit, char *extname, > int *status)
```

3 Write the binary table header keywords into the CHU. The optional TUNITn and EXTNAME keywords are written only if the input pointers are not null. The pcount parameter, which specifies the size of the variable length array heap, should initially = 0; CFITSIO 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 CFITSIO 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.

```
int fits_write_btblhdr / ffphbn
    (fitsfile *fptr, LONGLONG nrows, int tfields, char **ttype,
     char **tform, char **tunit, char *extname, LONGLONG pcount, > int *status)
```

4 Read the required keywords from the CHDU (image or table). When reading from an IMAGE extension the SIMPLE and EXTEND parameters are ignored. A null pointer may be supplied for any of the returned parameters that are not needed.

```
int fits_read_imghdr / ffghpr
    (fitsfile *fptr, int maxdim, > int *simple, int *bitpix, int *naxis,
     long *naxes, long *pcount, long *gcount, int *extend, int *status)

int fits_read_imghdrll / ffghprll
    (fitsfile *fptr, int maxdim, > int *simple, int *bitpix, int *naxis,
     LONGLONG *naxes, long *pcount, long *gcount, int *extend, int *status)

int fits_read_atblhdr / ffghtb
    (fitsfile *fptr,int maxdim, > long *rowlen, long *nrows,
     int *tfields, char **ttype, LONGLONG *tbcol, char **tform, char **tunit,
     char *extname,  int *status)

int fits_read_atblhdrll / ffghtbll
    (fitsfile *fptr,int maxdim, > LONGLONG *rowlen, LONGLONG *nrows,
     int *tfields, char **ttype, long *tbcol, char **tform, char **tunit,
     char *extname,  int *status)

int fits_read_btblhdr / ffghbn
    (fitsfile *fptr, int maxdim, > long *nrows, int *tfields,
     char **ttype, char **tform, char **tunit, char *extname,
     long *pcount, int *status)

int fits_read_btblhdrll / ffghbnll
    (fitsfile *fptr, int maxdim, > LONGLONG *nrows, int *tfields,
     char **ttype, char **tform, char **tunit, char *extname,
     long *pcount, int *status)
```

9.3.3 Write Keyword Routines

These routines simply append a new keyword to the header and do not check to see if a keyword with the same name already exists. In general it is preferable to use the fits_update_key routine to ensure that the same keyword is not written more than once to the header. See Appendix B for the definition of the parameters used in these routines.

1 Write (append) a new keyword of the appropriate data type into the CHU. A null pointer may be entered for the comment parameter, which will cause the comment field of the keyword to be left blank. The flt, dbl, cmp, and dblcmp 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.

```
int fits_write_key_str / ffpkys
    (fitsfile *fptr, char *keyname, char *value, char *comment,
     > int *status)

int fits_write_key_[log, lng] /  ffpky[lj]
    (fitsfile *fptr, char *keyname, DTYPE numval, char *comment,
     > int *status)

int fits_write_key_[flt, dbl, fixflg, fixdbl] / ffpky[edfg]
    (fitsfile *fptr, char *keyname, DTYPE numval, int decimals,
    char *comment, > int *status)

int fits_write_key_[cmp, dblcmp, fixcmp, fixdblcmp] / ffpk[yc,ym,fc,fm]
    (fitsfile *fptr, char *keyname, DTYPE *numval, int decimals,
    char *comment, > int *status)
```

2 Write (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'Local FITS Conventions' section in Chapter 4. Since this uses a non-standard FITS convention to encode the long keyword string, programs which use this routine should also call the fits_write_key_longwarn routine to add some COMMENT keywords to warn users of the FITS file that this convention is being used. The fits_write_key_longwarn routine 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 fits_write_key_longwarn will simply return without doing anything.

```
int fits_write_key_longstr / ffpkls
    (fitsfile *fptr, char *keyname, char *longstr, char *comment,
     > int *status)

int fits_write_key_longwarn / ffplsw
    (fitsfile *fptr, > int *status)
```

3 Write (append) a numbered sequence of keywords into the CHU. The starting index number (nstart) must be greater than 0. 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. One may also enter a null pointer for the comment parameter to leave the comment field of the keyword blank.

```
int fits_write_keys_str / ffpkns
    (fitsfile *fptr, char *keyroot, int nstart, int nkeys,
     char **value, char **comment, > int *status)

int fits_write_keys_[log, lng] / ffpkn[lj]
    (fitsfile *fptr, char *keyroot, int nstart, int nkeys,
     DTYPE *numval, char **comment, int *status)

int fits_write_keys_[flt, dbl, fixflg, fixdbl] / ffpkne[edfg]
    (fitsfile *fptr, char *keyroot, int nstart, int nkey,
     DTYPE *numval, int decimals, char **comment, > int *status)
```

4 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.

```
int fits_copy_key / ffcpky
    (fitsfile *infptr, fitsfile *outfptr, int innum, int outnum,
     char *keyroot, > int *status)
```

5 Write (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 ffgkyt routine should be used to read this keyword value, because the other keyword reading routines will not preserve the full precision of the value.

```
int fits_write_key_triple / ffpkyt
    (fitsfile *fptr, char *keyname, long intval, double frac,
     char *comment, > int *status)
```

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

```
int fits_write_key_template / ffpktp
    (fitsfile *fptr, const char *filename, > int *status)
```

9.3.4 Insert Keyword Routines

These insert routines are somewhat less efficient than the 'update' or 'write' keyword routines because the following keywords in the header must be shifted down to make room for the inserted keyword. See Appendix B for the definition of the parameters used in these routines.

1 Insert a new keyword record into the CHU at the specified position (i.e., immediately preceding the (keynum)th keyword in the header.)

```
int fits_insert_record / ffirec
    (fitsfile *fptr, int keynum, char *card, > int *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 'longstr' version has the same functionality as the 'str' version except that it also supports the local long string keyword convention for strings longer than 68 characters. A null pointer may be entered for the comment parameter which will cause the comment field to be left blank. The flt, dbl, cmp, and dblcmp 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.

```
int fits_insert_card / ffikey
    (fitsfile *fptr, char *card, > int *status)

int fits_insert_key_[str, longstr] / ffi[kys, kls]
    (fitsfile *fptr, char *keyname, char *value, char *comment,
     > int *status)

int fits_insert_key_[log, lng] / ffiky[lj]
    (fitsfile *fptr, char *keyname, DTYPE numval, char *comment,
     > int *status)

int fits_insert_key_[flt, fixflt, dbl, fixdbl] / ffiky[edfg]
    (fitsfile *fptr, char *keyname, DTYPE numval, int decimals,
     char *comment, > int *status)

int fits_insert_key_[cmp, dblcmp, fixcmp, fixdblcmp] / ffik[yc,ym,fc,fm]
    (fitsfile *fptr, char *keyname, DTYPE *numval, int decimals,
     char *comment, > int *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.

```
int fits_insert_key_null / ffikyu
    (fitsfile *fptr, char *keyname, char *comment, > int *status)
```

9.3.5 Read Keyword Routines

Wild card characters may be used when specifying the name of the keyword to be read.

1 Read a keyword value (with the appropriate data type) and comment from the CHU. If a NULL comment pointer is given on input, then the comment string will not be returned. If the value of the keyword is not defined (i.e., the value field is blank) then an error status = VALUE_UNDEFINED will be returned and the input value will not be changed (except that ffgkys will reset the value to a null string).

```
int fits_read_key_str / ffgkys
    (fitsfile *fptr, char *keyname, > char *value, char *comment,
     int *status);

NOTE: after calling the following routine, programs must explicitly free
      the memory allocated for 'longstr' after it is no longer needed by
      calling fits_free_memory.

int fits_read_key_longstr / ffgkls
    (fitsfile *fptr, char *keyname, > char **longstr, char *comment,
           int *status)

int fits_free_memory / fffree
    (char *longstr, > int *status);

int fits_read_key_[log, lng, flt, dbl, cmp, dblcmp] / ffgky[ljedcm]
    (fitsfile *fptr, char *keyname, > DTYPE *numval, char *comment,
     int *status)

int fits_read_key_lnglng / ffgkyjj
    (fitsfile *fptr, char *keyname, > LONGLONG *numval, char *comment,
     int *status)
```

2 Read a sequence of indexed keyword values (e.g., NAXIS1, NAXIS2, ...). The input starting index number (nstart) must be greater than 0. If the value of any of the keywords is not defined (i.e., the value field is blank) then an error status = VALUE_UNDEFINED will be returned and the input value for the undefined keyword(s) will not be changed. These routines do not support wild card characters in the root name. If there are no indexed keywords in the header with the input root name then these routines do not return a non-zero status value and instead simply return nfound = 0.

```
int fits_read_keys_str / ffgkns
    (fitsfile *fptr, char *keyname, int nstart, int nkeys,
     > char **value, int *nfound,  int *status)

int fits_read_keys_[log, lng, flt, dbl] / ffgkn[ljed]
    (fitsfile *fptr, char *keyname, int nstart, int nkeys,
     > DTYPE *numval, int *nfound, int *status)
```

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

```
int fits_read_key_triple / ffgkyt
    (fitsfile *fptr, char *keyname, > long *intval, double *frac,
     char *comment, int *status)
```

9.3.6 Modify Keyword Routines

These routines modify the value of an existing keyword. An error is returned if the keyword does not exist. Wild card characters may be used when specifying the name of the keyword to be modified. See Appendix B for the definition of the parameters used in these routines.

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

```
int fits_modify_record / ffmrec
    (fitsfile *fptr, int keynum, char *card, > int *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.

```
int fits_modify_card / ffmcrd
    (fitsfile *fptr, char *keyname, char *card, > int *status)
```

5 Modify the value and comment fields of an existing keyword in the CHU. The 'longstr' version has the same functionality as the 'str' version except that it also supports the local long string keyword convention for strings longer than 68 characters. 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 (&) or by entering a null pointer for the comment parameter. The flt, dbl, cmp, and dblcmp 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.

```
int fits_modify_key_[str, longstr] / ffm[kys, kls]
    (fitsfile *fptr, char *keyname, char *value, char *comment,
     > int *status);

int fits_modify_key_[log, lng] / ffmky[lj]
    (fitsfile *fptr, char *keyname, DTYPE numval, char *comment,
     > int *status)

int fits_modify_key_[flt, dbl, fixflt, fixdbl] / ffmky[edfg]
```

```
    (fitsfile *fptr, char *keyname, DTYPE numval, int decimals,
     char *comment, > int *status)

int fits_modify_key_[cmp, dblcmp, fixcmp, fixdblcmp] / ffmk[yc,ym,fc,fm]
    (fitsfile *fptr, char *keyname, DTYPE *numval, int decimals,
     char *comment, > int *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 (&) or by entering a null pointer.

```
int fits_modify_key_null / ffmkyu
    (fitsfile *fptr, char *keyname, char *comment, > int *status)
```

9.3.7 Update Keyword Routines

1 These update routines modify the value, and optionally the comment field, of the keyword if it already exists, otherwise the new keyword is appended to the header. A separate routine is provided for each keyword data type. The 'longstr' version has the same functionality as the 'str' version except that it also supports the local long string keyword convention for strings longer than 68 characters. A null pointer may be entered for the comment parameter which will leave the comment field unchanged or blank. The flt, dbl, cmp, and dblcmp 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.

```
int fits_update_key_[str, longstr] / ffu[kys, kls]
    (fitsfile *fptr, char *keyname, char *value, char *comment,
     > int *status)

int fits_update_key_[log, lng] / ffuky[lj]
    (fitsfile *fptr, char *keyname, DTYPE numval, char *comment,
     > int *status)

int fits_update_key_[flt, dbl, fixflt, fixdbl] / ffuky[edfg]
    (fitsfile *fptr, char *keyname, DTYPE numval, int decimals,
     char *comment, > int *status)

int fits_update_key_[cmp, dblcmp, fixcmp, fixdblcmp] / ffuk[yc,ym,fc,fm]
    (fitsfile *fptr, char *keyname, DTYPE *numval, int decimals,
     char *comment, > int *status)
```

9.4 Define Data Scaling and Undefined Pixel Parameters

These routines set or modify the internal parameters used by CFITSIO to either scale the data or to represent undefined pixels. Generally CFITSIO will scale the data according to the values of the BSCALE and BZERO (or TSCALn and TZEROn) keywords, however these routines 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, CFITSIO 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 routines do not create or modify the corresponding header keyword values. See Appendix B for the definition of the parameters used in these routines.

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.

```
int fits_set_bscale / ffpscl
    (fitsfile *fptr, double scale, double zero, > int *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.

```
int fits_set_tscale / fftscl
    (fitsfile *fptr, int colnum, double scale, double zero,
     > int *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, or 32. This does not create or change the value of the BLANK keyword in the header.

```
int fits_set_imgnull / ffpnul
    (fitsfile *fptr, LONGLONG nulval, > int *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.

```
int fits_set_atblnull / ffsnul
    (fitsfile *fptr, int colnum, char *nulstr, > int *status)
```

5 Define the value to be used to signify undefined pixels in an integer column in a binary table (where TFORMn = 'B', 'I', or 'J'). This does not create or change the value of the TNULLn keyword.

```
int fits_set_btblnull / fftnul
    (fitsfile *fptr, int colnum, LONGLONG nulval, > int *status)
```

9.5 Specialized FITS Primary Array or IMAGE Extension I/O Routines

These routines read or write data values in the primary data array (i.e., the first HDU in the FITS file) or an IMAGE extension. Automatic data type conversion is performed for if the data type of the FITS array (as defined by the BITPIX keyword) differs from the data type of the array in the calling routine. The data values are automatically scaled by the BSCALE and BZERO header values as they are being written or read from the FITS array. Unlike the basic routines described in the previous chapter, most of these routines specifically support the FITS random groups format. See Appendix B for the definition of the parameters used in these routines.

The more primitive reading and writing routines (i. e., ffppr_, ffppn_, ffppn, ffgpv_, or ffgpf_) simply treat the primary array as a long 1-dimensional array of pixels, ignoring the intrinsic dimensionality of the array. When dealing with a 2D image, for example, the application program must calculate the pixel offset in the 1-D array that corresponds to any particular X, Y coordinate in the image. C programmers should note that the ordering of arrays in FITS files, and hence in all the CFITSIO calls, is more similar to the dimensionality of arrays in Fortran rather than C. For instance if a FITS image has NAXIS1 = 100 and NAXIS2 = 50, then a 2-D array just large enough to hold the image should be declared as array[50][100] and not as array[100][50].

For convenience, higher-level routines are also provided to specifically deal with 2D images (ffp2d_ and ffg2d_) and 3D data cubes (ffp3d_ and ffg3d_). 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 dimensions of the program array may be larger than the dimensions of the FITS array. 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).

The most general set of routines (ffpss_, ffgsv_, and ffgsf_) may be used to 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 fpixel and lpixel parameters are integer arrays which specify the starting and ending pixel coordinate in each dimension (starting with 1, not 0) of the FITS image that is to be read or written. It is important to note that these are the starting and ending pixels in the FITS image, not in the declared array in the program. The array parameter in these routines is treated simply as a large one-dimensional array of the appropriate data type 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 FITS reading routines (ffgsv_ and ffgsf_) also have an 'inc' parameter which defines the data sampling interval in each dimension of the FITS array. For example, if inc[0]=2 and inc[1]=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 to the 'array' parameter.

Two types of routines are provided to read the data array which differ in the way undefined pixels are handled. The first type of routines (e.g., ffgpv_) simply return an array of data elements in which undefined pixels are set equal to a value specified by the user in the 'nulval' parameter. An additional feature of these routines is that if the user sets nulval = 0, then no checks for undefined pixels will be performed, thus reducing the amount of CPU processing. The second type of routines (e.g., ffgpf_) returns the data element array and, in addition, a char array that indicates whether the value of the corresponding data pixel is undefined (= 1) or defined (= 0). The latter type of routines 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.

1 Write elements into the FITS data array.

```
int fits_write_img / ffppr
    (fitsfile *fptr, int datatype, LONGLONG firstelem, LONGLONG nelements,
     DTYPE *array, int *status);

int fits_write_img_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffppr[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, LONGLONG firstelem, LONGLONG nelements,
     DTYPE *array, > int *status);

int fits_write_imgnull / ffppn
    (fitsfile *fptr, int datatype, LONGLONG firstelem, LONGLONG nelements,
     DTYPE *array, DTYPE *nulval, > int *status);

int fits_write_imgnull_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffppn[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, LONGLONG firstelem,
         LONGLONG nelements, DTYPE *array, DTYPE nulval, > int *status);
```

2 Set data array elements as undefined.

```
int fits_write_img_null / ffppru
    (fitsfile *fptr, long group, LONGLONG firstelem, LONGLONG nelements,
     > int *status)
```

3 Write values into group parameters. This routine only applies to the 'Random Grouped' FITS format which has been used for applications in radio interferometry, but is officially deprecated for future use.

```
int fits_write_grppar_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffpgp[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, long firstelem, long nelements,
     > DTYPE *array, int *status)
```

4 Write a 2-D or 3-D image into the data array.

```
int fits_write_2d_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffp2d[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, LONGLONG dim1, LONGLONG naxis1,
     LONGLONG naxis2, DTYPE *array, > int *status)

int fits_write_3d_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffp3d[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, LONGLONG dim1, LONGLONG dim2, LONGLONG naxis1,
     LONGLONG naxis2, LONGLONG naxis3, DTYPE *array, > int *status)
```

5 Write an arbitrary data subsection into the data array.

```
int fits_write_subset_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffpss[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, long naxis, long *naxes,
     long *fpixel, long *lpixel, DTYPE *array, > int *status)
```

6 Read elements from the FITS data array.

```
int fits_read_img / ffgpv
    (fitsfile *fptr, int  datatype, long firstelem, long nelements,
     DTYPE *nulval, > DTYPE *array, int *anynul, int *status)

int fits_read_img_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffgpv[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, long group, long firstelem, long nelements,
     DTYPE nulval, > DTYPE *array, int *anynul, int *status)

int fits_read_imgnull / ffgpf
    (fitsfile *fptr, int  datatype, long firstelem, long nelements,
     > DTYPE *array, char *nullarray, int *anynul, int *status)

int  fits_read_imgnull_[byt, sht, usht, int, uint, lng, ulng, flt, dbl] /
     ffgpf[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, long group, long firstelem, long nelements,
     > DTYPE *array, char *nullarray, int *anynul, int *status)
```

7 Read values from group parameters. This routine only applies to the 'Random Grouped' FITS format which has been used for applications in radio interferometry, but is officially deprecated for future use.

```
int  fits_read_grppar_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
     ffggp[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, long group, long firstelem, long nelements,
     > DTYPE *array, int *status)
```

8 Read 2-D or 3-D image from the data array. Undefined pixels in the array will be set equal to the value of 'nulval', unless nulval=0 in which case no testing for undefined pixels will be performed.

```
int  fits_read_2d_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
     ffg2d[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, long group, DTYPE nulval, LONGLONG dim1, LONGLONG naxis1,
     LONGLONG naxis2, > DTYPE *array, int *anynul, int *status)

int  fits_read_3d_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
     ffg3d[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, long group, DTYPE nulval, LONGLONG dim1,
     LONGLONG dim2, LONGLONG naxis1, LONGLONG naxis2, LONGLONG naxis3,
     > DTYPE *array, int *anynul, int *status)
```

9 Read an arbitrary data subsection from the data array.

```
int  fits_read_subset_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
     ffgsv[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, int group, int naxis, long *naxes,
     long *fpixel, long *lpixel, long *inc, DTYPE nulval,
     > DTYPE *array, int *anynul, int *status)

int  fits_read_subsetnull_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
     ffgsf[b,i,ui,k,uk,j,uj,jj,e,d]
     (fitsfile *fptr, int group, int naxis, long *naxes,
     long *fpixel, long *lpixel, long *inc, > DTYPE *array,
     char *nullarray, int *anynul, int *status)
```

9.6 Specialized FITS ASCII and Binary Table Routines

9.6.1 General Column Routines

1 Get information about an existing ASCII or binary table column. A null pointer may be given for any of the output parameters that are not needed. DATATYPE is a character string which returns the data type of the column as defined by the TFORMn keyword (e.g., 'I', 'J','E', 'D', etc.). In the case of an ASCII character column, typecode 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 ffgbcl will return typechar='A8' and repeat=20. All the returned parameters are scalar quantities.

```
int fits_get_acolparms / ffgacl
  (fitsfile *fptr, int colnum, > char *ttype, long *tbcol,
   char *tunit, char *tform, double *scale, double *zero,
```

```
     char *nulstr, char *tdisp, int *status)

  int fits_get_bcolparms / ffgbcl
      (fitsfile *fptr, int colnum, > char *ttype, char *tunit,
       char *typechar, long *repeat, double *scale, double *zero,
       long *nulval, char *tdisp, int  *status)

  int fits_get_bcolparmsll / ffgbclll
      (fitsfile *fptr, int colnum, > char *ttype, char *tunit,
       char *typechar, LONGLONG *repeat, double *scale, double *zero,
       LONGLONG *nulval, char *tdisp, int  *status)
```

2 Return 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.

```
  int fits_get_rowsize / ffgrsz
      (fitsfile *fptr, long *nrows, *status)
```

3 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 routine also automatically writes the value of theap to a keyword in the extension header. This routine must be called after the required keywords have been written (with ffphbn) but before any data is written to the table.

```
  int fits_write_theap / ffpthp
      (fitsfile *fptr, long theap, > int *status)
```

4 Test the contents of the binary table variable array heap, returning the size of the heap, the number of unused bytes that are not currently pointed to by any of the descriptors, and the number of bytes which are pointed to by multiple descriptors. It also returns valid = FALSE if any of the descriptors point to invalid addresses out of range of the heap.

```
  int fits_test_heap / fftheap
      (fitsfile *fptr, > LONGLONG *heapsize, LONGLONG *unused, LONGLONG *overlap,
       int *validheap, int *status)
```

5 Re-pack the vectors in the binary table variable array heap to recover any unused space. Normally, when a vector in a variable length array column is rewritten the previously written array remains in the heap as wasted unused space. This routine will repack the arrays that are still in use, thus eliminating any bytes in the heap that are no longer in use. Note that if several vectors point to the same bytes in the heap, then this routine will make duplicate copies of the bytes for each vector, which will actually expand the size of the heap.

```
int fits_compress_heap / ffcmph
    (fitsfile *fptr, > int *status)
```

9.6.2 Low-Level Table Access Routines

The following 2 routines 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. These routines do not perform any machine dependent data conversion or byte swapping. See Appendix B for the definition of the parameters used in these routines.

1 Read or write a consecutive array of bytes from an ASCII or binary table

```
int fits_read_tblbytes / ffgtbb
    (fitsfile *fptr, LONGLONG firstrow, LONGLONG firstchar, LONGLONG nchars,
     > unsigned char *values, int *status)

int fits_write_tblbytes / ffptbb
    (fitsfile *fptr, LONGLONG firstrow, LONGLONG firstchar, LONGLONG nchars,
     unsigned char *values, > int *status)
```

9.6.3 Write Column Data Routines

1 Write elements into an ASCII or binary table column (in the CDU). The data type of the array is implied by the suffix of the routine name.

```
int fits_write_col_str / ffpcls
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, char **array, > int *status)

int fits_write_col_[log,byt,sht,usht,int,uint,lng,ulng,lnglng,flt,dbl,cmp,dblcmp] /
    ffpcl[l,b,i,ui,k,uk,j,uj,jj,e,d,c,m]
    (fitsfile *fptr, int colnum, LONGLONG firstrow,
         LONGLONG firstelem, LONGLONG nelements, DTYPE *array, > int *status)
```

2 Write elements into an ASCII or binary table column substituting the appropriate FITS null value for any elements that are equal to the nulval parameter.

```
int fits_write_colnull_[log, byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffpcn[l,b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, DTYPE *array, DTYPE nulval, > int *status)
```

3 Write string elements into a binary table column (in the CDU) substituting the FITS null value for any elements that are equal to the nulstr string.

```
int fits_write_colnull_str / ffpcns
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, char **array, char *nulstr, > int *status)
```

4 Write bit values into a binary byte ('B') or bit ('X') table column (in the CDU). Larray is an array of characters corresponding to the sequence of bits to be written. If an element of larray is true (not equal to zero) then the corresponding bit in the FITS table is set to 1, otherwise the bit is set to 0. The 'X' column in a FITS table is always padded out to a multiple of 8 bits where the bit array starts with the most significant bit of the byte and works down towards the 1's bit. For example, a '4X' array, with the first bit = 1 and the remaining 3 bits = 0 is equivalent to the 8-bit unsigned byte decimal value of 128 ('1000 0000B'). In the case of 'X' columns, CFITSIO can 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 ffpclx with firstbit=1 and nbits=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 and should all be set = 0. It should also be noted that it is more efficient to write 'X' columns an entire byte at a time, instead of bit by bit. Any of the CFITSIO routines that write to columns (e.g. fits_write_col_byt) may be used for this purpose. These routines will interpret 'X' columns as though they were 'B' columns (e.g., '1X' through '8X' is equivalent to '1B', and '9X' through '16X' is equivalent to '2B').

```
int fits_write_col_bit / ffpclx
    (fitsfile *fptr, int colnum, LONGLONG firstrow, long firstbit,
     long nbits, char *larray, > int *status)
```

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

```
int fits_write_descript / ffpdes
    (fitsfile *fptr, int colnum, LONGLONG rownum, LONGLONG repeat,
     LONGLONG offset, > int *status)
```

9.6.4 Read Column Data Routines

Two types of routines are provided to get the column data which differ in the way undefined pixels are handled. The first set of routines (ffgcv) 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. If nullval = 0, then no checks for undefined pixels will be performed, thus increasing the speed of the program. The second set of routines (ffgcf) returns the data element array and in addition a logical array of flags which defines whether the corresponding data pixel is undefined. See Appendix B for the definition of the parameters used in these routines.

Any column, regardless of it's intrinsic data type, 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 data type of the column. The length of the returned strings (not including the null terminating character) can be determined with the fits_get_col_display_width 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 Read 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 = nulval, unless nulval = 0 (or = ' ' for ffgcvs) in which case no checking for undefined values will be performed. The anynul parameter is set to true if any of the returned elements are undefined.

```
int fits_read_col_str / ffgcvs
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, char *nulstr, > char **array, int *anynul,
     int *status)

int fits_read_col_[log,byt,sht,usht,int,uint,lng,ulng, lnglng, flt, dbl, cmp, dblcmp] /
    ffgcv[l,b,i,ui,k,uk,j,uj,jj,e,d,c,m]
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, DTYPE nulval, > DTYPE *array, int *anynul,
     int *status)
```

2 Read 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 nullarray element set equal to TRUE. The anynul parameter is set to true if any of the returned elements are undefined.

```
int fits_read_colnull_str / ffgcfs
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstelem,
     LONGLONG nelements, > char **array, char *nullarray, int *anynul,
     int *status)
```

```
int fits_read_colnull_[log,byt,sht,usht,int,uint,lng,ulng,lnglng,flt,dbl,cmp,dblcmp] /
    ffgcf[l,b,i,ui,k,uk,j,uj,jj,e,d,c,m]
    (fitsfile *fptr, int colnum, LONGLONG firstrow,
     LONGLONG firstelem, LONGLONG nelements, > DTYPE *array,
     char *nullarray, int *anynul, int *status)
```

3 Read 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 'nulval', unless nulval=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 fpixel(naxis+1) and lpixel(naxis+1), and hence are treated as the next higher dimension of the FITS N-dimensional array. The INC parameter specifies the sampling interval in each dimension between the data elements that will be returned.

```
int fits_read_subset_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffgsv[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, int colnum, int naxis, long *naxes, long *fpixel,
     long *lpixel, long *inc, DTYPE nulval, > DTYPE *array, int *anynul,
     int *status)
```

4 Read 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 'nullarray' element set equal to TRUE. The first and last rows in the table to be read are specified by fpixel(naxis+1) and lpixel(naxis+1), and hence are treated as the next higher dimension of the FITS N-dimensional array. The INC parameter specifies the sampling interval in each dimension between the data elements that will be returned.

```
int fits_read_subsetnull_[byt, sht, usht, int, uint, lng, ulng, lnglng, flt, dbl] /
    ffgsf[b,i,ui,k,uk,j,uj,jj,e,d]
    (fitsfile *fptr, int colnum, int naxis, long *naxes,
     long *fpixel, long *lpixel, long *inc, > DTYPE *array,
     char *nullarray, int *anynul, int *status)
```

5 Read bit values from a byte ('B') or bit ('X') table column (in the CDU). Larray is an array of logical values corresponding to the sequence of bits to be read. If larray is true then the corresponding bit was set to 1, otherwise the bit was set to 0. The 'X' column in a FITS table is always padded out to a multiple of 8 bits where the bit array starts with the most significant bit of the byte and works down towards the 1's bit. For example, a '4X' array, with the first bit = 1 and the remaining 3 bits = 0 is equivalent to the 8-bit unsigned byte value of 128. Note that in the case of 'X' columns, CFITSIO can 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 ffgcx with firstbit=1 and nbits=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. It should also be noted that it is more efficient to read 'X' columns an entire byte at a time, instead of bit by bit.

Any of the CFITSIO routines that read columns (e.g. fits_read_col_byt) may be used for this purpose. These routines will interpret 'X' columns as though they were 'B' columns (e.g., '8X' is equivalent to '1B', and '16X' is equivalent to '2B').

```
int fits_read_col_bit / ffgcx
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG firstbit,
     LONGLONG nbits, > char *larray, int *status)
```

6 Read any consecutive set of bits from an 'X' or 'B' column and interpret them as an unsigned n-bit integer. nbits must be less than 16 or 32 in ffgcxui and ffgcxuk, respectively. If nrows is greater than 1, then the same set of bits will be read from each row, starting with firstrow. The bits are numbered with 1 = the most significant bit of the first element of the column.

```
int fits_read_col_bit_[usht, uint] / ffgcx[ui,uk]
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG, nrows,
     long firstbit, long nbits, > DTYPE *array, int *status)
```

7 Return 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 pair of routine returns a single descriptor whereas the second pair of routine returns the descriptors for a range of rows in the table. The only difference between the 2 routines in each pair is that one returns the parameters as 'long' integers, whereas the other returns the values as 64-bit 'LONGLONG' integers.

```
int fits_read_descript / ffgdes
    (fitsfile *fptr, int colnum, LONGLONG rownum, > long *repeat,
         long *offset, int *status)

int fits_read_descriptll / ffgdesll
    (fitsfile *fptr, int colnum, LONGLONG rownum, > LONGLONG *repeat,
         LONGLONG *offset, int *status)

int fits_read_descripts / ffgdess
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG nrows
     > long *repeat, long *offset, int *status)

int fits_read_descriptsll / ffgdessll
    (fitsfile *fptr, int colnum, LONGLONG firstrow, LONGLONG nrows
     > LONGLONG *repeat, LONGLONG *offset, int *status)
```

Chapter 10

Extended File Name Syntax

10.1 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 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 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 table filtering 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 with 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. See also the stream:// driver which provides a more efficient, but more restricted method of reading or writing to the stdin or stdout streams.
- `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 virtual file that is opened by the application just contains this single image in 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 input image in this case.

- `myfile.fits[EVENTS][col Rad = sqrt(X**2 + Y**2)]`: creates and opens a virtual file on the fly 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 virtual 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)[compress]

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

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

The filetype, BaseFilename, outName, HDUlocation, ImageSection, and pixFilter 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.

10.2 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 (see following note).
       shmem://  - opens or creates a file which persists in the computer's
                   shared memory (see following note).
         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.

10.2.1 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.

10.2.2 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.

10.2.3 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).

10.2.4 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 that 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.

10.2.5 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]'.

10.3 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 data type 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 data type 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.

10.4 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.

The output filename "mem://" is also allowed, which will write the output file into memory, and also allow write access to the file. This 'file' will disappear when it is closed, but this may be useful for some applications which only need to modify a temporary copy of the file.

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.

10.5 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 in the following Template Files chapter.

10.6 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
```

10.7 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.

An optional pound sign character ("#") may be appended to the extension name or number to signify that any other extensions in the file should be ignored during any subsequent file filtering operations. For example, when doing row filtering operations on a table extension, CFITSIO normally creates a copy of the filtered table in memory, along with a verbatim copy of all the other extensions in the input FITS file. If the pound sign is appended to the table extension name, then only that extension, and none of the other extensions in the file, will by copied to memory, as in the following example:

```
myfile.fit[events#][TIME > 10000]
```

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.
```

10.8 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. (If a '#' character is appended to the name or number of the image HDU, as in "myfile.fits[1#][1:200,1:200]", then the other HDUs in the input file will not be copied into memory). 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.

10.9 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.

10.10 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. (If a '#' character is appended to the name or number of the table HDU then only the primary array, and none of the other HDUs in the input file will be copied into memory). 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 commas or semi-colons.

- Copy only a specified list of columns columns to the filtered input file. The list of column name should be separated by commas or 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 data type 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 data type 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 data type is not specified then an appropriate data type 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 data type 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.

  COMMENT and HISTORY keywords may also be created with the following syntax:

  ```
  #COMMENT = 'This is a comment keyword'
  #HISTORY = 'This is a history keyword'
  ```

  Note that the equal sign and the quote characters will be removed, so that the resulting header keywords in these cases will look like this:

  ```
  COMMENT This is a comment keyword
  HISTORY This is a history keyword
  ```

  These two special keywords are always appended to the end of the header and will not affect any previously existing COMMENT or HISTORY keywords.

- 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 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 commas or semi-colons. 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.
```

10.11 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. (If a '#' character is appended to the name or number of the table HDU then only the primary array, and none of the other HDUs in the input file will be copied into 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.

10.11.1 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 inclosing 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.

10.11.2 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.

10.11.3 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.

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 V is a boolean vector, SUM returns the number of TRUE elements. The NELEM function returns the number of elements in vector V 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 data type present. Any elements which are themselves vectors, will be expanded out with each of its elements becoming an element in the constructed vector.

10.11.4 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" ] ] ] )
or
  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.

10.11.5 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 semimajor/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 low-level functions that are primarily for use with regfilter function, but they can be called directly. They return a boolean true or false depending on whether a two dimensional point is in the region or not. The positional coordinates must be given in pixel units:

```
"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.
```

10.11.6 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.
```

10.12 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 data type 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 11

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.

11.1 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. Integer values must be within the allowed range of a 'signed long' variable; some C compilers only suppport 4-byte long integers with a range from -2147483648 to +2147483647, whereas other C compilers support 8-byte integers with a range of plus or minus 2**63.

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.

11.2 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!

11.3 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.

11.4 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.

11.5 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.

11.6 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 12

Local FITS Conventions

CFITSIO supports several local FITS conventions which are not defined in the official 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.

12.1 64-Bit Long Integers

CFITSIO supports reading and writing FITS images or table columns containing 64-bit integer data values. Support for 64-bit integers was added to the official FITS Standard in December 2005. FITS 64-bit images have BITPIX = 64, and the 64-bit binary table columns have TFORMn = 'K'. CFITSIO also supports the 'Q' variable-length array table column format which is analogous to the 'P' column format except that the array descriptor is stored as a pair of 64-bit integers.

For the convenience of C programmers, the fitsio.h include file defines (with a typedef statement) the 'LONGLONG' datatype to be equivalent to an appropriate 64-bit integer datatype on each platform. Since there is currently no universal standard for the name of the 64-bit integer datatype (it might be defined as 'long long', 'long', or '__int64' depending on the platform) C programmers may prefer to use the 'LONGLONG' datatype when declaring or allocating 64-bit integer quantities when writing code which needs to run on multiple platforms. Note that CFITSIO will implicitly convert the datatype when reading or writing FITS 64-bit integer images and columns with data arrays of a different integer or floating point datatype, but there is an increased risk of loss of numerical precision or numerical overflow in this case.

12.2 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 CFITSIO 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 as a warning to any software that must read the keywords. A routine called fits_write_key_longwarn has been provided in CFITSIO to write this keyword if it does not already exist.

This long string convention is supported by the following CFITSIO routines:

```
    fits_write_key_longstr  - write a long string keyword value
    fits_insert_key_longstr - insert a long string keyword value
    fits_modify_key_longstr - modify a long string keyword value
    fits_update_key_longstr - modify a long string keyword value
    fits_read_key_longstr   - read  a long string keyword value
    fits_delete_key         - delete a keyword
```

The fits_read_key_longstr routine is unique among all the CFITSIO routines in that it internally allocates memory for the long string value; all the other CFITSIO routines that deal with arrays require that the calling program pre-allocate adequate space to hold the array of data. Consequently, programs which use the fits_read_key_longstr routine must be careful to free the allocated memory for the string when it is no longer needed.

The following 2 routines also have limited support for this long string convention,

```
      fits_modify_key_str - modify an existing string keyword value
      fits_update_key_str - update a string keyword value
```

in that they will correctly overwrite an existing long string value, but the new string value is limited to a maximum of 68 characters in length.

The more commonly used CFITSIO routines to write string valued keywords (fits_update_key and fits_write_key) do not support this long string convention and only support 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 fits_write_key_longstr routine 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.

12.3 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.

CFITSIO 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 CFITSIO routines that read or write strings in binary tables. The Binary Table definition document specifies that other optional characters may follow the data type code in the TFORM keyword, so this local convention is in compliance with the FITS standard although other FITS readers may not 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. CFITSIO does not currently support this proposal.

12.4 Keyword Units Strings

One limitation 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 problem, CFITSIO 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 fits_read_key_unit and fits_write_key_unit routines in CFITSIO read and write, respectively, the keyword unit strings in an existing keyword.

12.5 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://fits.gsfc.nasa.gov/registry/hierarch_keyword.html and in Section 4.4 of the ESO Data Interface Control Document that is linked to from http://archive.eso.org/cms/tools-documentation/eso-data-interface-control.html.

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.

12.6 Tile-Compressed Image Format

CFITSIO supports a convention for compressing n-dimensional images and storing the resulting byte stream in a variable-length column in a FITS binary table. The general principle used in this convention is to first divide the n-dimensional image into a rectangular grid of subimages or 'tiles'. Each tile is then compressed as a continuous block of data, and the resulting compressed byte stream is stored in a row of a variable length column in a FITS binary table. By dividing the image into tiles it is generally possible to extract and uncompress subsections of the image without having to uncompress the whole image. The default tiling pattern treats each row of a 2-dimensional image (or higher dimensional cube) as a tile, such that each tile contains NAXIS1 pixels (except the default with the HCOMPRESS algorithm is to compress the whole 2D image as a single tile). Any other rectangular tiling pattern may also be defined. In the case of relatively small images it may be sufficient to compress the entire image as a single tile, resulting in an output binary table with 1 row. In the case of 3-dimensional data cubes, it may be advantageous to treat each plane of the cube as a separate tile if application software typically needs to access the cube on a plane by plane basis.

See section 5.6 "Image Compression" for more information on using this tile-compressed image format.

Chapter 13

Optimizing Programs

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. To get a general idea of what data I/O speeds are possible on a particular machine, build the speed.c program that is distributed with CFITSIO (type 'make speed' in the CFITSIO directory). This diagnostic program measures the speed of writing and reading back a test FITS image, a binary table, and an ASCII table.

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.

13.1 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 by default.

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. 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 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.

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 relatively 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 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.

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 IO 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 IO 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 and on the number of IO buffers that have been allocated in CFITSIO. The CFITSIO Iterator routine will automatically use the optimal-sized buffer, but there is also a CFITSIO routine that will return the optimal number of rows for a given table: fits_get_rowsize. 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 fits_get_rowsize 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 IO 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 the CFITSIO Iterator routine. This routine provides a more 'object oriented' way of reading and writing FITS files which automatically uses the most appropriate data buffer size to achieve the maximum I/O throughput.

5. 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 that have the same information content.

6. Design software so that it reads the FITS header keywords in the same order in which they occur in the file. When reading keywords, CFITSIO 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 IO buffers, then the header keyword access will be relatively fast and it makes little difference which order they are accessed.

7. 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 fits_set_bscale or fits_set_tscale) and then read or write the raw unscaled values in the FITS file.

8. Avoid using the 'implicit data type conversion' capability in CFITSIO. 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 CFITSIO to convert the data to a different data type can slow the program.

9. 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.

10. 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.

11. 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 CFITSIO routines fits_read_tblbytes and fits_write_tblbytes 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 for even greater efficiency) of a table with a single function 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 CFITSIO, and they allow the program to write the data to the output file in exactly the same byte order. For these same reasons, these routines can corrupt the FITS data file if used incorrectly because no validation or machine dependent conversion is performed by these

routines. These routines are only recommended for optimizing critical pieces of code and should only be used by programmers who thoroughly understand the internal format of the FITS tables they are reading or writing.

12. 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 fits_write_tblbytes. 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?).

13. Finally, external factors such as the speed of the data storage device, the size of the data cache, 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, the entire hardware and software system should be reviewed to identify any potential I/O bottlenecks.

Appendix A

Index of Routines

Appendix B

Parameter Definitions

```
anynul   - set to TRUE (=1) if any returned values are undefined, else FALSE
array    - array of numerical data values to read or write
ascii    - encoded checksum string
binspec  - the input table binning specifier
bitpix   - bits per pixel. The following symbolic mnemonics are predefined:
               BYTE_IMG   =   8 (unsigned char)
               SHORT_IMG  =  16 (signed short integer)
               LONG_IMG   =  32 (signed long integer)
               LONGLONG_IMG =  64 (signed long 64-bit integer)
               FLOAT_IMG  = -32 (float)
               DOUBLE_IMG = -64 (double).
           The LONGLONG_IMG type is experimental and is not officially
           recognized in the FITS Standard document.
           Two additional values, USHORT_IMG and ULONG_IMG are also available
           for creating unsigned integer images.  These are equivalent to
           creating a signed integer image with BZERO offset keyword values
           of 32768 or 2147483648, respectively, which is the convention that
           FITS uses to store unsigned integers.
card     - header record to be read or written (80 char max, null-terminated)
casesen  - CASESEN (=1) for case-sensitive string matching, else CASEINSEN (=0)
cmopt    - grouping table "compact" option parameter. Allowed values are:
           OPT_CMT_MBR and OPT_CMT_MBR_DEL.
colname  - name of the column (null-terminated)
colnum   - column number (first column = 1)
colspec  - the input file column specification; used to delete, create, or rename
           table columns
comment  - the keyword comment field (72 char max, null-terminated)
complm   - should the checksum be complemented?
comptype - compression algorithm to use: GZIP_1, RICE_1, HCOMPRESS_1, or PLIO_1
coordtype- type of coordinate projection (-SIN, -TAN, -ARC, -NCP,
           -GLS, -MER, or -AIT)
cpopt    - grouping table copy option parameter. Allowed values are:
```

```
           OPT_GCP_GPT, OPT_GCP_MBR, OPT_GCP_ALL, OPT_MCP_ADD, OPT_MCP_NADD,
           OPT_MCP_REPL, amd OPT_MCP_MOV.
create_col- If TRUE, then insert a new column in the table, otherwise
           overwrite the existing column.
current  - if TRUE, then the current HDU will be copied
dataok   - was the data unit verification successful (=1) or
           not (= -1).  Equals zero if the DATASUM keyword is not present.
datasum  - 32-bit 1's complement checksum for the data unit
dataend  - address (in bytes) of the end of the HDU
datastart- address (in bytes) of the start of the data unit
datatype - specifies the data type of the value.  Allowed value are:  TSTRING,
           TLOGICAL, TBYTE, TSBYTE, TSHORT, TUSHORT, TINT, TUINT, TLONG, TULONG,
           TFLOAT, TDOUBLE, TCOMPLEX, and TDBLCOMPLEX
datestr  - FITS date/time string: 'YYYY-MM-DDThh:mm:ss.ddd', 'YYYY-MM-dd',
           or 'dd/mm/yy'
day      - calendar day (UTC) (1-31)
decimals - number of decimal places to be displayed
deltasize - increment for allocating more memory
dim1     - declared size of the first dimension of the image or cube array
dim2     - declared size of the second dimension of the data cube array
dispwidth - display width of a column = length of string that will be read
dtype    - data type of the keyword ('C', 'L', 'I', 'F' or 'X')
                C = character string
                L = logical
                I = integer
                F = floating point number
                X = complex, e.g., "(1.23, -4.56)"
err_msg  - error message on the internal stack (80 chars max)
err_text - error message string corresponding to error number (30 chars max)
exact    - TRUE (=1) if the strings match exactly;
           FALSE (=0) if wildcards are used
exclist  - array of pointers to keyword names to be excluded from search
exists   - flag indicating whether the file or compressed file exists on disk
expr     - boolean or arithmetic expression
extend   - TRUE (=1) if FITS file may have extensions, else FALSE (=0)
extname  - value of the EXTNAME keyword (null-terminated)
extspec  - the extension or HDU specifier; a number or name, version, and type
extver   - value of the EXTVER keyword = integer version number
filename - full name of the FITS file, including optional HDU and filtering specs
filetype - type of file (file://, ftp://, http://, etc.)
filter   - the input file filtering specifier
firstchar- starting byte in the row (first byte of row = 1)
firstfailed - member HDU ID (if positive) or grouping table GRPIDn index
           value (if negative) that failed grouping table verification.
firstelem- first element in a vector (ignored for ASCII tables)
firstrow - starting row number (first row of table = 1)
```

```
following- if TRUE, any HDUs following the current HDU will be copied
fpixel   - coordinate of the first pixel to be read or written in the
           FITS array.  The array must be of length NAXIS and have values such
           that fpixel[0] is in the range 1 to NAXIS1, fpixel[1] is in the
           range 1 to NAXIS2, etc.
fptr     - pointer to a 'fitsfile' structure describing the FITS file.
frac     - factional part of the keyword value
gcount   - number of groups in the primary array (usually = 1)
gfptr    - fitsfile* pointer to a grouping table HDU.
group    - GRPIDn/GRPLCn index value identifying a grouping table HDU, or
           data group number (=0 for non-grouped data)
grouptype - Grouping table parameter that specifies the columns to be
           created in a grouping table HDU. Allowed values are: GT_ID_ALL_URI,
           GT_ID_REF, GT_ID_POS, GT_ID_ALL, GT_ID_REF_URI, and GT_ID_POS_URI.
grpname  - value to use for the GRPNAME keyword value.
hdunum   - sequence number of the HDU (Primary array = 1)
hduok    - was the HDU verification successful (=1) or
           not (= -1).  Equals zero if the CHECKSUM keyword is not present.
hdusum   - 32 bit 1's complement checksum for the entire CHDU
hdutype  - HDU type: IMAGE_HDU (0), ASCII_TBL (1), BINARY_TBL (2), ANY_HDU (-1)
header   - returned character string containing all the keyword records
headstart- starting address (in bytes) of the CHDU
heapsize - size of the binary table heap, in bytes
history  - the HISTORY keyword comment string (70 char max, null-terminated)
hour     - hour within day (UTC) (0 - 23)
inc      - sampling interval for pixels in each FITS dimension
inclist  - array of pointers to matching keyword names
incolnum - input column number; range = 1 to TFIELDS
infile   - the input filename, including path if specified
infptr   - pointer to a 'fitsfile' structure describing the input FITS file.
intval   - integer part of the keyword value
iomode   - file access mode: either READONLY (=0) or READWRITE (=1)
keyname  - name of a keyword (8 char max, null-terminated)
keynum   - position of keyword in header (1st keyword = 1)
keyroot  - root string for the keyword name (5 char max, null-terminated)
keysexist- number of existing keyword records in the CHU
keytype  - header record type: -1=delete;  0=append or replace;
                   1=append; 2=this is the END keyword
longstr  - arbitrarily long string keyword value (null-terminated)
lpixel   - coordinate of the last pixel to be read or written in the
           FITS array.  The array must be of length NAXIS and have values such
           that lpixel[0] is in the range 1 to NAXIS1, lpixel[1] is in the
           range 1 to NAXIS2, etc.
match    - TRUE (=1) if the 2 strings match, else FALSE (=0)
maxdim   - maximum number of values to return
member   - row number of a grouping table member HDU.
```

```
memptr   - pointer to the a FITS file in memory
mem_realloc - pointer to a function for reallocating more memory
memsize  - size of the memory block allocated for the FITS file
mfptr    - fitsfile* pointer to a grouping table member HDU.
mgopt    - grouping table merge option parameter. Allowed values are:
           OPT_MRG_COPY, and OPT_MRG_MOV.
minute   - minute within hour (UTC) (0 - 59)
month    - calendar month (UTC) (1 - 12)
morekeys - space in the header for this many more keywords
n_good_rows - number of rows evaluating to TRUE
namelist - string containing a comma or space delimited list of names
naxes    - size of each dimension in the FITS array
naxis    - number of dimensions in the FITS array
naxis1   - length of the X/first axis of the FITS array
naxis2   - length of the Y/second axis of the FITS array
naxis3   - length of the Z/third axis of the FITS array
nbytes   - number of bytes or characters to read or write
nchars   - number of characters to read or write
nelements- number of data elements to read or write
newfptr  - returned pointer to the reopened file
newveclen- new value for the column vector repeat parameter
nexc     - number of names in the exclusion list (may = 0)
nfound   - number of keywords found (highest keyword number)
nkeys    - number of keywords in the sequence
ninc     - number of names in the inclusion list
nmembers - Number of grouping table members (NAXIS2 value).
nmove    - number of HDUs to move (+ or -), relative to current position
nocomments - if equal to TRUE, then no commentary keywords will be copied
noisebits- number of bits to ignore when compressing floating point images
nrows    - number of rows in the table
nstart   - first integer value
nullarray- set to TRUE (=1) if corresponding data element is undefined
nulval   - numerical value to represent undefined pixels
nulstr   - character string used to represent undefined values in ASCII table
numval   - numerical data value, of the appropriate data type
offset   - byte offset in the heap or data unit to the first element of the vector
openfptr - pointer to a currently open FITS file
overlap  - number of bytes in the binary table heap pointed to by more than 1
           descriptor
outcolnum- output column number; range = 1 to TFIELDS + 1
outfile  - and optional output filename; the input file will be copied to this prior
           to opening the file
outfptr  - pointer to a 'fitsfile' structure describing the output FITS file.
pcount   - value of the PCOUNT keyword = size of binary table heap
previous - if TRUE, any previous HDUs in the input file will be copied.
repeat   - length of column vector (e.g. 12J); == 1 for ASCII table
```

```
rmopt    - grouping table remove option parameter. Allowed values are:
           OPT_RM_GPT, OPT_RM_ENTRY, OPT_RM_MBR, and OPT_RM_ALL.
rootname - root filename, minus any extension or filtering specifications
rot      - celestial coordinate rotation angle (degrees)
rowlen   - length of a table row, in characters or bytes
rowlist  - sorted list of row numbers to be deleted from the table
rownum   - number of the row (first row = 1)
rowrange - list of rows or row ranges: '3,6-8,12,56-80' or '500-'
row_status - array of True/False results for each row that was evaluated
scale    - linear scaling factor; true value = (FITS value) * scale + zero
second   - second within minute (0 - 60.9999999999) (leap second!)
section  - section of image to be copied (e.g. 21:80,101:200)
simple   - TRUE (=1) if FITS file conforms to the Standard, else FALSE (=0)
space    - number of blank spaces to leave between ASCII table columns
status   - returned error status code (0 = OK)
sum      - 32 bit unsigned checksum value
tbcol    - byte position in row to start of column (1st col has tbcol = 1)
tdisp    - Fortran style display format for the table column
tdimstr  - the value of the TDIMn keyword
templt   - template string used in comparison (null-terminated)
tfields  - number of fields (columns) in the table
tfopt    - grouping table member transfer option parameter. Allowed values are:
           OPT_MCP_ADD, and OPT_MCP_MOV.
tform    - format of the column (null-terminated); allowed values are:
           ASCII tables:  Iw, Aw, Fww.dd, Eww.dd, or Dww.dd
           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.
           Variable length array columns are denoted by a '1P' before the data type
           character (e.g., '1PJ').  When creating a binary table, 2 addition tform
           data type codes are recognized by CFITSIO: 'rU' and 'rV' for unsigned
           16-bit and unsigned 32-bit integer, respectively.

theap    - zero indexed byte offset of starting address of the heap
           relative to the beginning of the binary table data
tilesize - array of length NAXIS that specifies the dimensions of
           the image compression tiles
ttype    - label or name for table column (null-terminated)
tunit    - physical unit for table column (null-terminated)
typechar - symbolic code of the table column data type
typecode - data type code of the table column.  The negative of
           the value indicates a variable length array column.
                Datatype             typecode    Mnemonic
                bit, X                   1        TBIT
                byte, B                 11        TBYTE
                logical, L              14        TLOGICAL
                ASCII character, A      16        TSTRING
```

```
                short integer, I        21         TSHORT
                integer, J              41         TINT32BIT (same as TLONG)
                long long integer, K    81         TLONGLONG
                real, E                 42         TFLOAT
                double precision, D     82         TDOUBLE
                complex, C              83         TCOMPLEX
                double complex, M      163         TDBLCOMPLEX
unit     - the physical unit string (e.g., 'km/s') for a keyword
unused   - number of unused bytes in the binary table heap
urltype  - the file type of the FITS file (file://, ftp://, mem://, etc.)
validheap- returned value = FALSE if any of the variable length array
           address are outside the valid range of addresses in the heap
value    - the keyword value string (70 char max, null-terminated)
version  - current version number of the CFITSIO library
width    - width of the character string field
xcol     - number of the column containing the X coordinate values
xinc     - X axis coordinate increment at reference pixel (deg)
xpix     - X axis pixel location
xpos     - X axis celestial coordinate (usually RA) (deg)
xrefpix  - X axis reference pixel array location
xrefval  - X axis coordinate value at the reference pixel (deg)
ycol     - number of the column containing the X coordinate values
year     - calendar year (e.g. 1999, 2000, etc)
yinc     - Y axis coordinate increment at reference pixel (deg)
ypix     - y axis pixel location
ypos     - y axis celestial coordinate (usually DEC) (deg)
yrefpix  - Y axis reference pixel array location
yrefval  - Y axis coordinate value at the reference pixel (deg)
zero     - scaling offset; true value = (FITS value) * scale + zero
```

Appendix C

CFITSIO Error Status Codes

The following table lists all the error status codes used by CFITSIO. Programmers are encouraged to use the symbolic mnemonics (defined in the file fitsio.h) rather than the actual integer status values to improve the readability of their code.

```
 Symbolic Const    Value     Meaning
 --------------    -----  -----------------------------------------
                     0    OK, no error
 SAME_FILE         101    input and output files are the same
 TOO_MANY_FILES    103    tried to open too many FITS files at once
 FILE_NOT_OPENED   104    could not open the named file
 FILE_NOT_CREATED  105    could not create the named file
 WRITE_ERROR       106    error writing to FITS file
 END_OF_FILE       107    tried to move past end of file
 READ_ERROR        108    error reading from FITS file
 FILE_NOT_CLOSED   110    could not close the file
 ARRAY_TOO_BIG     111    array dimensions exceed internal limit
 READONLY_FILE     112    Cannot write to readonly file
 MEMORY_ALLOCATION 113    Could not allocate memory
 BAD_FILEPTR       114    invalid fitsfile pointer
 NULL_INPUT_PTR    115    NULL input pointer to routine
 SEEK_ERROR        116    error seeking position in file

 BAD_URL_PREFIX     121   invalid URL prefix on file name
 TOO_MANY_DRIVERS   122   tried to register too many IO drivers
 DRIVER_INIT_FAILED 123   driver initialization failed
 NO_MATCHING_DRIVER 124   matching driver is not registered
 URL_PARSE_ERROR    125   failed to parse input file URL
 RANGE_PARSE_ERROR  126   parse error in range list

 SHARED_BADARG     151    bad argument in shared memory driver
 SHARED_NULPTR     152    null pointer passed as an argument
 SHARED_TABFULL    153    no more free shared memory handles
```

SHARED_NOTINIT	154	shared memory driver is not initialized
SHARED_IPCERR	155	IPC error returned by a system call
SHARED_NOMEM	156	no memory in shared memory driver
SHARED_AGAIN	157	resource deadlock would occur
SHARED_NOFILE	158	attempt to open/create lock file failed
SHARED_NORESIZE	159	shared memory block cannot be resized at the moment

HEADER_NOT_EMPTY	201	header already contains keywords
KEY_NO_EXIST	202	keyword not found in header
KEY_OUT_BOUNDS	203	keyword record number is out of bounds
VALUE_UNDEFINED	204	keyword value field is blank
NO_QUOTE	205	string is missing the closing quote
BAD_INDEX_KEY	206	illegal indexed keyword name (e.g. 'TFORM1000')
BAD_KEYCHAR	207	illegal character in keyword name or card
BAD_ORDER	208	required keywords out of order
NOT_POS_INT	209	keyword value is not a positive integer
NO_END	210	couldn't find END keyword
BAD_BITPIX	211	illegal BITPIX keyword value
BAD_NAXIS	212	illegal NAXIS keyword value
BAD_NAXES	213	illegal NAXISn keyword value
BAD_PCOUNT	214	illegal PCOUNT keyword value
BAD_GCOUNT	215	illegal GCOUNT keyword value
BAD_TFIELDS	216	illegal TFIELDS keyword value
NEG_WIDTH	217	negative table row size
NEG_ROWS	218	negative number of rows in table
COL_NOT_FOUND	219	column with this name not found in table
BAD_SIMPLE	220	illegal value of SIMPLE keyword
NO_SIMPLE	221	Primary array doesn't start with SIMPLE
NO_BITPIX	222	Second keyword not BITPIX
NO_NAXIS	223	Third keyword not NAXIS
NO_NAXES	224	Couldn't find all the NAXISn keywords
NO_XTENSION	225	HDU doesn't start with XTENSION keyword
NOT_ATABLE	226	the CHDU is not an ASCII table extension
NOT_BTABLE	227	the CHDU is not a binary table extension
NO_PCOUNT	228	couldn't find PCOUNT keyword
NO_GCOUNT	229	couldn't find GCOUNT keyword
NO_TFIELDS	230	couldn't find TFIELDS keyword
NO_TBCOL	231	couldn't find TBCOLn keyword
NO_TFORM	232	couldn't find TFORMn keyword
NOT_IMAGE	233	the CHDU is not an IMAGE extension
BAD_TBCOL	234	TBCOLn keyword value < 0 or > rowlength
NOT_TABLE	235	the CHDU is not a table
COL_TOO_WIDE	236	column is too wide to fit in table
COL_NOT_UNIQUE	237	more than 1 column name matches template
BAD_ROW_WIDTH	241	sum of column widths not = NAXIS1
UNKNOWN_EXT	251	unrecognizable FITS extension type

UNKNOWN_REC	252	unknown record; 1st keyword not SIMPLE or XTENSION
END_JUNK	253	END keyword is not blank
BAD_HEADER_FILL	254	Header fill area contains non-blank chars
BAD_DATA_FILL	255	Illegal data fill bytes (not zero or blank)
BAD_TFORM	261	illegal TFORM format code
BAD_TFORM_DTYPE	262	unrecognizable TFORM data type code
BAD_TDIM	263	illegal TDIMn keyword value
BAD_HEAP_PTR	264	invalid BINTABLE heap pointer is out of range
BAD_HDU_NUM	301	HDU number < 1
BAD_COL_NUM	302	column number < 1 or > tfields
NEG_FILE_POS	304	tried to move to negative byte location in file
NEG_BYTES	306	tried to read or write negative number of bytes
BAD_ROW_NUM	307	illegal starting row number in table
BAD_ELEM_NUM	308	illegal starting element number in vector
NOT_ASCII_COL	309	this is not an ASCII string column
NOT_LOGICAL_COL	310	this is not a logical data type column
BAD_ATABLE_FORMAT	311	ASCII table column has wrong format
BAD_BTABLE_FORMAT	312	Binary table column has wrong format
NO_NULL	314	null value has not been defined
NOT_VARI_LEN	317	this is not a variable length column
BAD_DIMEN	320	illegal number of dimensions in array
BAD_PIX_NUM	321	first pixel number greater than last pixel
ZERO_SCALE	322	illegal BSCALE or TSCALn keyword = 0
NEG_AXIS	323	illegal axis length < 1
NOT_GROUP_TABLE	340	Grouping function error
HDU_ALREADY_MEMBER	341	
MEMBER_NOT_FOUND	342	
GROUP_NOT_FOUND	343	
BAD_GROUP_ID	344	
TOO_MANY_HDUS_TRACKED	345	
HDU_ALREADY_TRACKED	346	
BAD_OPTION	347	
IDENTICAL_POINTERS	348	
BAD_GROUP_ATTACH	349	
BAD_GROUP_DETACH	350	
NGP_NO_MEMORY	360	malloc failed
NGP_READ_ERR	361	read error from file
NGP_NUL_PTR	362	null pointer passed as an argument. Passing null pointer as a name of template file raises this error
NGP_EMPTY_CURLINE	363	line read seems to be empty (used internally)
NGP_UNREAD_QUEUE_FULL	364	cannot unread more then 1 line (or single

```
                                       line twice)
NGP_INC_NESTING       365     too deep include file nesting (infinite
                                loop, template includes itself ?)
NGP_ERR_FOPEN         366     fopen() failed, cannot open template file
NGP_EOF               367     end of file encountered and not expected
NGP_BAD_ARG           368     bad arguments passed. Usually means
                                internal parser error. Should not happen
NGP_TOKEN_NOT_EXPECT  369     token not expected here

BAD_I2C           401    bad int to formatted string conversion
BAD_F2C           402    bad float to formatted string conversion
BAD_INTKEY        403    can't interpret keyword value as integer
BAD_LOGICALKEY    404    can't interpret keyword value as logical
BAD_FLOATKEY      405    can't interpret keyword value as float
BAD_DOUBLEKEY     406    can't interpret keyword value as double
BAD_C2I           407    bad formatted string to int conversion
BAD_C2F           408    bad formatted string to float conversion
BAD_C2D           409    bad formatted string to double conversion
BAD_DATATYPE      410    illegal datatype code value
BAD_DECIM         411    bad number of decimal places specified
NUM_OVERFLOW      412    overflow during data type conversion
DATA_COMPRESSION_ERR   413  error compressing image
DATA_DECOMPRESSION_ERR 414  error uncompressing image

BAD_DATE          420    error in date or time conversion

PARSE_SYNTAX_ERR  431    syntax error in parser expression
PARSE_BAD_TYPE    432    expression did not evaluate to desired type
PARSE_LRG_VECTOR  433    vector result too large to return in array
PARSE_NO_OUTPUT   434    data parser failed not sent an out column
PARSE_BAD_COL     435    bad data encounter while parsing column
PARSE_BAD_OUTPUT  436    Output file not of proper type

ANGLE_TOO_BIG     501    celestial angle too large for projection
BAD_WCS_VAL       502    bad celestial coordinate or pixel value
WCS_ERROR         503    error in celestial coordinate calculation
BAD_WCS_PROJ      504    unsupported type of celestial projection
NO_WCS_KEY        505    celestial coordinate keywords not found
APPROX_WCS_KEY    506    approximate wcs keyword values were returned
```