FITSIO User's Guide

A Subroutine Interface to FITS Format Files

for Fortran Programmers

Version 5.0

Dr. William D. Pence
HEASARC
Code 662
Goddard Space Flight Center
Greenbelt, MD 20771
USA

November 1997

Contents

Chapter 1

Introduction

FITSIO is a machine-independent library of Fortran-77 subroutines for reading and writing data files in the FITS (Flexible Image Transport System) data format. This library was written to provide a powerful yet simple interface for accessing FITS files which will run on most commonly used computers and workstations. This version of FITSIO supports all the features described in the official NOST definition of the FITS format and can read and write all the currently defined types of extensions, including ASCII tables (TABLE), Binary tables (BINTABLE) and IMAGE extensions. The FITSIO subroutines insulate the programmer from having to deal with the complicated formatting details in the FITS file, however, it is assumed that users have a general knowledge about the structure and usage of FITS files.

The FITSIO package was developed for use 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. The latest version of the FITSIO source code, documentation, and example programs are all available on the World-Wide Web at the following URL:

```
http://heasarc.gsfc.nasa.gov/fitsio
```

FITSIO also can be obtained from the HEASARC via anonymous ftp from **legacy.gsfc.nasa.gov** in the **software/fitsio/fortran** subdirectory.

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


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


This User's Guide assumes that readers already have a general understanding of the definition and structure of FITS format files. For further information about FITS formats, please obtain a copy of the 'FITS User's Guide' and the 'NOST FITS Standard', which are available from the NASA, Science Office of Standards and Technology at the address given below. Both of these documents are available electronically via anonymous ftp at nssdc.gsfc.nasa.gov in the /pub/fits directory. Any questions about FITS formats should be directed to the NOST, at:

```
NASA, Science Office of Standards and Technology
Code 633.2,
Goddard Space Flight Center
Greenbelt MD 20771
USA
WWW: http://ssdoo.gsfc.nasa.gov/astro/fits/fits_home.html
E-mail: fits@nssdca.gsfc.nasa.gov
(301) 286-2899
```


FITSIO users may also be interested in the FTOOLS package of programs that can be used to manipulate and analyze FITS format files. Information about FTOOLS can be obtained on the WWW at

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

or via anonymous FTP at

```
legacy.gsfc.nasa.gov /software/ftools/release
```

1.1 Acknowledgements

The author would like to thank the many individuals who have used or tested FITSIO and have offered valuable suggestions to improve the FITSIO package. These include:

Steve Allen, Keith Arnaud, Kent Blackburn, G Bodammer, Romke Bontekoe, Robin Corbet, Lucio Chiappetti, Richard Fink, Emily Greene, Cheng Ho, Phil Hodge, Jim Ingham, Mark Levine, Todd Karakaskian, Edward King, Scott Koch, Don Jennings, Claire Larkin, Rob Managan, John Mattox, Carsten Meyer, Stefan Mochnacki, Bruce O'Neel, Clive Page, Arvind Parmar, Jeff Pedelty, Tim Pearson, Chris Rogers, Arnold Rots, Barry Schlesinger, Robin Stebbins, Allyn Tennant, Peter Teuben, Doug Tody, Steve Walton, Dan Whipple, and Nelson Zarate.

Chapter 2

Creating the FITSIO Library

2.1 Machine Specific Versions

The FITSIO package consists of about 30000 lines of Fortran source code contained in about 350 subroutines. The large majority of subroutines are written in strict Fortran-77 (except for the occasional use of INTEGER*2 variables) and are contained in the file named fitsio.f. These subroutines can be compiled and run on any machine that has a Fortran-77 compiler. A second file, called fitsfort.f, contains additional general subroutines that are used in all cases except for the IRAF/SPP version (which does not use Fortran read and write statements).

To complete the FITSIO library, an additional set of machine-specific subroutines must be added. These subroutines allow for compiler-specific differences in the file OPEN statement, the subroutine to determine the current date, and a few other non-portable aspects of Fortran-77.

Recently fitsio has been ported to be compilant with Fortran-90 compilers. The fitsf90.f source file is intended to be used with any Fortran-90 compiler, but it has only been tested on few F90 compilers.

The build scripts included in the FITSIO release will automatically include the correct machine-specific Fortran-77 source file, but for reference, the following table shows which file should be used on various machines:

```
     PLATFORM                                    SOURCE FILE
     --------                                    -----------
All Fortran-90 compilers                         fitsf90.f
SUN, HP, IBM AIX workstations                    fitssun.f
DEC Ultrix, DEC Alphas OSF/1, SGI, NeXT(Oasys)   fitsdec.f
NeXT (with Absoft Fortran)                       fitsnxab.f
Linux UNIX operating system on IBM PCs           fitslinx.f
VAX/VMS (also need vmsieeed.mar)                 fitsvax.f
DEC Alpha with OpenVMS                           fitsalphavms.f
Cray supercomputers                              fitscray.f
Macintosh PCs + Absoft Fortran                   fitsmac.f
Macintosh PCs + Language Systems Fortran         fitsmacls.f
```

Note that IBM mainframe computers and some older compilers for IBM PC clones are no longer supported by this version of FITSIO. The previous 4.14 version of FITSIO may still be used on these platforms. The fitsf90.f file should work with most newer Fortran-90 compilers on IBM PCs.

A version of FITSIO is also available which runs in the IRAF environment and provides a set of SPP callable interface routines. See the later section of this guide for more information on this IRAF version.

2.2 Building the Fortran Library

A set of Make files and command files are provided with FITSIO to automate the process of building the FITSIO library on most machines. The following sections describe the build procedure for each type of machine.

2.2.1 Unix Platforms

The FITSIO library is built on most Unix systems by typing:

```
> configure
> make
```

at the operating system prompt. The 'configure' command customizes the Makefile for the particular system, then the 'make' command builds the library. The build process splits each routine in the source files into a separate disk file in a scratch subdirectory and may take some time to finish, depending on the speed of the processor.

On HP-UX systems using an older Fortran-77 compiler, programs that call FITSIO should be compiled and linked with the +U77 switch (to bring in the idate library function) and the +E5 switch (to enable the same FITS file to be attached to multiple Fortran unit numbers). The idate subroutine is only needed when writing the DATE FITS keyword with the FTPDAT subroutine. For example, to compile the testprog.f program type:

```
fort77 +U77 -o testprog testprog.f -L. -lfitsio
```

2.2.2 VMS Platforms

The makevms.com and command file may be used to build the FITSIO library on DEC Alpha/OSF and VAX/VMS machines (e.g., type '@makevmv' on the command line). Note that the vmsieeed.mar macro routine is used on VAX/VMS systems to perform the translations between Vax floating point formats and IEEE floating point formats.

FITSIO supports both the G_float (default) and D_float compiler options on Alpha VMS systems. The vmsieee.c routines perform the the IEEE translations on Alpha VMS machines when the /float=D_float compiler option is invoked. Th makevmsd.com file can be used to build FITSIO with this option.

2.2.3 IBM Compatible PCs running DOS or Windows

This version of FITSIO currently requires a Fortran-90 compiler to build on an IBM PC running DOS or Windows. The previous 4.14 version of FITSIO may still be used with the older Fortran-77 compilers. In fact it is probably not difficult to port this new version FITSIO to run with most older compilers; anyone wishing to do so should contact the author for assistance.

To build the FITSIO library with a Fortran-90 compiler, one needs to compile all the subroutines in fitsio.f, fitsfort.f, and fitsf90.f. It usually results in smaller and more efficient code if each subroutine in these source files is split into a separate file before compiling, rather than compiling the entire large source file at once. A program called split.f is included with the fitsio distribution which will split the fitsio.f and fitsfort.f subroutines into separate files.

The exact commands to compile and build the library depend on the compiler, but the following example for the Microsoft PowerStation Fortran compiler illustrates the sequence of steps that are required:

```
#  compile and link the split program:
FL32 SPLIT.F

#  run split to break the large sources files into
#  many smaller files:
SPLIT

#  compile all the small files; write output to a log file:
FL32 /C FT*.F, FITSF90.F >COMPILE.LOG

# create the fitsio object library; write output to a log file
LINK32 -LIB /OUT:FITSIO.LIB FT*.OBJ, FITSF90.OBJ >LINKER.LOG
```

The object module library, FITSIO.LIB, may then be used when linking any program that calls the FITSIO subroutines.

2.2.4 Using Macintosh PCs

There are currently 2 versions of FITSIO for use with different Fortran compilers on Macintosh PCs. The fitsmac.f file is for the Absoft compiler, and the fitsmacls.f file should be used with the Language Systems Fortran (V3.3) compiler. The MacMakefiles.sit.hqx file contains libfitsio.make which will build libfitsio.f.o. Use Stuffit Expander (freeware, see any good archive site) to unbinhex and unstuff these files. You can change FMakeOptions and FLinkOptions if you want. For instance, set

```
FMakeOptions = ... -sym
FLinkOptions = ... -sym ON
```

to use SADE. FITSIO has not been extensively tested on Mac systems, so please report any problems or additional suggestions for successfully building FITSIO on Macs to the author.

2.2.5 Using the f2c Compiler

The widely available f2c program may be used to convert the fortran FITSIO source files into C code, which then may be processed by a C compiler. This avoids the need to have a Fortran compiler to build the FITSIO library. The fitsio.f file should process through f2c without any problems, but the machine-dependent version of FITSIO may have a few non-standard fortran statements which f2c will not understand. One such case is the use of the IDATE subroutine, or equivalent, to get the system date which is used in the ftgsdt subroutine. The easiest workaround to this problem is to simply comment out this call and then avoid calling the ftpdat subroutine, which is the only FITSIO subroutine to use this function.

2.2.6 The IRAF SPP Compatible Version of FITSIO

An IRAF compatible version of FITSIO was developed in collaboration with Doug Tody at NOAO. This version of FITSIO uses the IRAF VOS (Virtual Operating System) file I/O calls to access the FITS files rather than using Fortran I/O,

The following source files are needed to build the IRAF compatible version of FITSIO:

```
fitsio.f      - The Fortran source code for the main body of FITSIO subroutines
fitsio.h      - The SPP include file, used by fitspp.x and fitssppb.x
fitsspp.com   - The SPP common block used by fitsspp.x
fitsspp.x     - The low-level FITSIO subroutines, written in SPP
fitssppvms.x - Version of fitsspp.x for use on VAX and Alpha VMS machines
fitssppb.x    - The top-level SPP interface to FITSIO
mkpkg         - a sample make file which builds the FITSIO library
```

The SPP version of FITSIO consists of 3 layers of code. At the bottom there is the fitsspp.x SPP source code which contains the IRAF-specific subroutines to read and write data to the FITS file. The middle layer of subroutines are contained in the fitsio.f file which is the same file used in every version of FITSIO. This layer is completely portable and contains no machine or environment dependencies. Finally, a thin interface layer of SPP subroutines are provided in the fitssppb.x file. This top layer of routines is provided for the convenience of SPP programmers and simply performs the conversion between SPP character variables and Fortran character strings.

A sample mkpkg file is provided which will build the FITSIO library in IRAF. There are 2 versions of the fitsspp.x file; the fitssppvms.x file is used on VAX and Alpha VMS machines, while the fitsspp.x file is used on all other platforms. Note that the 'fitsio.f' file must be renamed to 'fitsio.for' for the mkpkg to function properly.

Once the FITSIO library has been built, IRAF programmers may then call either the SPP or the Fortran subroutine interface to FITSIO; both interfaces end up calling the same low-level SPP VOS interface routines to actually read or write the FITS files. IRAF programs written in Fortran (using either the IRAF IMFORT or the STScI F77 interface to access the IRAF VOS) can simply call the Fortran subroutine interface to FITSIO as specified in later sections of this document.

Programs written in SPP, on the other hand, may call the SPP specific layer of FITSIO subroutines. These routines are identical to the Fortran subroutines except that the SPP subroutines have names beginning with 'fs' rather than 'ft' (e.g., 'call fsopen' rather than 'call ftopen'). The main purpose of this thin layer of SPP routines is to convert any SPP character variables into Fortran character strings before calling the corresponding FITSIO Fortran subroutine. Similarly, any character strings that are returned by the Fortran FITSIO subroutine are converted into SPP CHAR variables before being passed back to the calling SPP routine.

SPP programmers, therefore, should follow the definition of all the FITSIO interface subroutines given in this document, except that the SPP routines have names beginning with 'fs' rather than 'ft', and SPP CHAR variables should be substituted for any Fortran character string variables in the subroutine calling arguments. SPP programmers also need to be aware of the character string length definitions that are used throughout the SPP FITSIO interface as defined in the fitsio.h include file. The SPP version of FITSIO behaves identically to the Fortran version and has exactly the same calling sequences, except for the following special exceptions:

* FSPKNx: This family of subroutines only supports a single comment string, not an array of comment strings as in the Fortran routines.

* FSPCLS, FSGCVS, and FSGCFS: These 3 routines all have an additional argument not present

in the corresponding Fortran routines. This integer argument, DIM1, follows the character array argument and specifies the size of the 1st dimension of the 2-dimensional CHAR array, e.g.,

```
call fspcls(ounit, frow, felem, nelem, nulval, array, DIM1, anynul, status)
```

2.3 Testing the FITSIO Library

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

```
% f77 -o testprog testprog.f -L. -lfitsio
% testprog > testprog.lis
% cmp -l testprog.fft testprog.std
% diff testprog.lis testprog.out
```

On VMS systems type:

```
$ for testprog.f
$ link testprog, fitsio/lib
$ run testprog
```

The testprog program should produce a long set of diagnostic output messages ending with an 'OK' status, identical to those contained in the testprog.out file. The output FITS file, called testprog.fft, should be identical to the testprog.std FITS file included in this release. The 'cmp' command shown above should not report any differences between these 2 files except that on a few systems (e.g., HP-UX and LINUX) a few bytes may differ where a floating point number is formated with a leading blank instead of a leading zero in front of the decimal place (e.g. ' .000000' instead of '0.000000').

2.4 Getting Started with FITSIO

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

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

2. Write a simple program to read or write a FITS file using the Basic Interface routines described in Chapter 6.

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

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

5. Scan through the more extensive set of routines that are provided in the 'Advanced Interface', as described in Chapter 7. These routines perform more specialized functions than are provided by the Basic Interface routines.

2.5 Example Program

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

```
      program writeimage

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

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

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

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

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

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

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

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

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

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

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

2.6 Calling FITSIO from C programs (Obsolete Section)

[Note added March 1996: This section is now obsolete. C programmers should use the CFITSIO library, written entirely in ANSI-C rather than the CFORTRAN macros described below.]

C programmers may call the Fortran FITSIO subroutines by using a set of macro definitions included in the cfitsio.h header file that is distributed as part of the FITSIO package. This header file defines a C macro corresponding to every Fortran subroutine in the FITSIO user interface. These macros provide a machine-independent interface between C and Fortran by using the CFORTRAN package (actually a C header file) developed by Burkhard Burow (burow@vxdesy.cern.ch, U. of Toronto).

The CFITSIO macros have the same calling sequence as the FITSIO Fortran subroutines defined in this FITSIO User's Guide. The only difference is that the macro routines all have uppercase names and they begin with the letters 'FC' rather then 'ft' as in all the fortran subroutine names. As an example, one would call ftclos(unit, status) in fortran but would call FCCLOS(unit,&status) when calling the CFITSIO macros in a C program. Programmers should refer to the wsimple.c and rsimple.c sample programs that are distributed with FITSIO for examples of how to call the FITSIO routines.

To use CFITSIO, one must include the cfitsio.h header file at the beginning of the C program. Note that cfitsio.h requires 2 other C header files, cfortran.h and pctype.h, (which are also distributed with FITSIO) but these do not need to be explicitly included in the C application program.

C programmers should be aware that there are subtle differences between C and Fortran in the use of character strings and especially vectors of character strings. Please read the comments at the beginning of the cfitsio.h file for a summary of the restrictions when calling the FITSIO routines that have character string arguments.

C programs that call CFITSIO must be linked to include the Fortran language libraries that are needed by FITSIO (since it is written in Fortran). The exact link procedure is necessarily machine dependent, and unfortunately is not always obvious because of various pecularities in either the C or Fortran compilers. The following guidelines may be helpful in sorting out the correct link procedure on Unix platforms.

There are basically 2 options for linking a C program that calls Fortran routines. The first and simplest option, when it works, is to use the 'f77' command to link the C program; this will automatically link in the required Fortran language libraries without having to specify their names or directory path. For example, if one has a C program called foo.c which calls cfitsio, it may be compiled and linked with the following commands:

```
% cc -c foo.c
% f77 -o foo foo.o -L$FITSDIR -lfitsio
```

where $FITSDIR is an environment variable which points to the directory where the fitsio library has been installed (use '-L.' if the fitsio library is located in the current work directory). Unfortunately, using f77 to perform the linking does not always work, especially if the C and Fortran compilers are supplied by different vendors (e.g., when combining gcc with a commercial Fortran compiler). The error message that is reported in this case usually says something about an undefined or unresolved symbol 'main'.

If this error occurs, then a more orthodox linking procedure must be used which requires that the names and path to the Fortran language libraries be explicitly specified. The compile and link commands, (using 'gcc' instead of 'cc' in this example) would look something like:

```
% gcc -c foo.c
% gcc -o foo foo.o -L$FITSDIR -lfitsio $F77LFLAGS
```

where $F77LFLAGS is a platform-specific environment variable which specifies the Fortran 77 libraries that need to be linked with the program. It may require some detective work to determine where the Fortran libraries are located on any given machine. One method for finding them is to compile any simple Fortran program (such as the standard 'Hello, world!' program) using the -v option, as in

```
% f77 -v -o test test.f
```

This will produce a verbose compile and link listing, with a line that looks something like:

```
/bin/ld -dc -dp -e start -u _MAIN_ -X -o test /usr/lang/SC1.0/crt0.o
 /usr/lang/SC1.0/cg87/_crt1.o -L/usr/lang/SC1.0/cg87 -L/usr/lang/SC1.0
 test.o -lF77 -lm -lc
```

From this listing it should be possible to decipher the names of the Fortran libraries and their location. In this case the libraries are located in /usr/lang/SC1.0 and they are called libF77.a and libm.a. Thus to link the example cfitsio program on this machine, one would specify:

```
% gcc -c foo.c
% gcc -o foo foo.o -L$FITSDIR -lfitsio -L/usr/lang/SC1.0 -lF77 -lm
```

Another method of locating the Fortran libraries is to execute 'which f77' then search that directory, and any subdirectories, for the libraries.

For reference, here is the value for F77LFLAGS what was found to work on various platforms:

```
SunPro f77 2.0.1 on Solaris 2.4:

    F77LFLAGS= -L$F77HOME -lM77 -lF77 -lm

SunPro f77 3.0.1 on Solaris 2.3:

    F77LFLAGS= -L$F77HOME -lM77 -lF77 -lsunmath -lm

SunPro f77 1.0 on SunOS4.1.4:

    F77LFLAGS= -L$F77HOME -lF77 -lm

SunPro f77 3.0.1 on SunOS4.1.3:

    F77LFLAGS= -L$F77HOME -lF77 -lM77 -lsunmath -lm -lansi
```

```
Convex supercomputers:

    F77LFLAGS= -L$F77HOME -lI77 -lF77 -lU77 -lm -lmathC1 -llfs -lvfn
```

Chapter 3

A FITS Primer

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

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

A FITS file consists of one or more Header + Data Units (HDUs), where the first HDU is called the 'Primary HDU', or 'Primary Array'. The primary array contains an N-dimensional array of pixels, such as a 1-D spectrum, a 2-D image, or a 3-D data cube. The primary HDU can also consist of only a header with a null array containing no 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 NULs depending on the type of unit).

Each Header Unit consists of any number of 80-character keyword records or 'card images' (reminiscent of the 80-column punched cards which were prevalent when the FITS standard was developed) which have the general form:

```
KEYNAME = value / comment string
```

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 following example:

```
KEYNAME =          / comment: keyword has no value
```

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

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

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

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

Chapter 4

Basic FITSIO Conventions

4.1 Current Header Data Unit (CHDU)

A basic concept used throughout FITSIO is that of the current Header Data Unit (CHDU) within the FITS file. The FITSIO subroutines which read or write information only operate on the CHDU. When a FITS file is first created or opened the CHDU is automatically defined to be the first HDU (i.e., the primary array). FITSIO subroutines are provided to move to and open any other existing HDU within the FITS file or to append or insert a new HDU in the FITS file which then becomes the CHDU.

4.2 Subroutine Names

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

4.3 Subroutine Families and Datatypes

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

```
x - bit
b - character*1 (unsigned byte)
i - short integer (I*2)
j - integer (I*4)
e - real exponential floating point (R*4)
f - real fixed-format floating point (R*4)
d - double precision real floating-point (R*8)
g - double precision fixed-format floating point (R*8)
c - complex reals (pairs of R*4 values)
```

```
m - double precision complex (pairs of R*8 values)
l - logical (L*4)
s - character string
```

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

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

4.4 Implicit Data Type Conversion

Many of the data I/O subroutines have the ability to perform implicit data type conversion. This means that the data type of the subroutine parameter does not need to be the same as the data type of the value in the FITS file. More specifically, the implicit data type conversion will be performed for numerical data types when reading a FITS header keyword value and when reading or writing values to or from the primary array or a table column. FITSIO returns status = 412 if the converted data value exceeds the range of the output data type. Data type conversion is not allowed when reading or writing string (s), logical (l), complex (c), or double complex (m) data types.

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

4.5 Data Scaling

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

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

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

When writing numerical data to the primary array or to a table column the data values will generally be automatically inversely scaled by the value of the BSCALE and BZERO (or TSCALn and

TZEROn) header keyword values if they they exist in the header. These keywords must have been written to the header before any data is written for them to have any effect. Otherwise, one may use the ftpscl and fttscl subroutines to define or override the scaling keywords in the header (e.g., to turn off the scaling so that the program can write the raw unscaled values into the FITS file). If scaling is performed, the inverse scaled output value that is written into the FITS file will have

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

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

4.6 Error Status Values and the Error Message Stack

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

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

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

In some situations programs may encounter a non-fatal FITSIO error and will want to continue processing. An example is when a program fails to find an optional keyword in the header and FITSIO returns status = 202. The program may ignore this error and reset status=0, however this may still leave error messages on the stack. To clear the entire message stack in this situation, call the FTCMSG subroutine.

4.7 Variable-Length Array Facility in Binary Tables

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

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

All the data in a variable length field is written into an area called the 'heap' which follows the main fixed-length FITS binary table. The size of the heap, in bytes, is specified with the PCOUNT keyword in the FITS header. When creating a new binary table, the initial value of PCOUNT should usually be set to zero. FITSIO will recompute the size of the heap as the data is written and will automatically update the PCOUNT keyword value when the table is closed. Thus, application programs usually do not need to worry about the heap size except when inserting a new binary table HDU in front of other existing HDUs in the FITS file (with ftibin). In this specific case the correct final value of PCOUNT (or a larger value) must be specified when the HDU is initially created so that the correct amount of space will be inserted into the FITS file.

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). Thus when writing variable length arrays the number of rows in the table should be correctly specified (with the NAXIS2 keyword) at the time the table is first created. This differs from the simpler case of tables that only contain fixed-length columns where the number of rows in the table does not have to be explicitly defined until just before the table is closed. It is still possible to insert additional rows into a binary table containing variable-length columns (with the FTIROW routine), however, the performance may be rather slow due to all the internal shuffling of the data that must be performed.

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

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

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

When reading from a variable length array field one can only read as many elements as 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 return an error. One can determine the length of each row of the field with the FTGDES subroutine.

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 NOST FITS standard or the NOST FITS User's Guide for a list of these values). The FITSIO subroutines that read floating point data in FITS files recognize these IEEE special values and by default interpret the overflow and infinity values as being equivalent to a NaN, and convert the underflow and denormalized values into zeros. In some cases programmers may want access to the raw IEEE values, without any modification by FITSIO. This can be done by calling the FTGPVx or FTGCVx routines while specifying 0.0 as the value of the NULLVAL parameter. This will force FITSIO to simply pass the IEEE values through to the application program, without any modification. This does not work for double precision values on VAX/VMS machines, however, where there is no easy way to bypass the default interpretation of the IEEE special values.

4.9 Local FITS Conventions supported by FITSIO

In a couple cases FITSIO supports local FITS conventions which are not defined in the official NOST FITS standard and which are not necessarily recognized or supported by other FITS software packages. Programmers should be cautious about using these features, especially if the FITS files that are produced are expected to be processed by other software systems which do not use the FITSIO interface. These local conventions should be considered as prototypes, and they may not necessarily be supported in future versions of FITSIO, especially if an alternative convention is officially adopted by the FITS community.

4.9.1 Support for Long String Keyword Values.

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

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

It is recommended that the LONGSTRN keyword, as shown here, always be included in any HDU that uses this longstring convention. A subroutine called FTPLSW has been provided in FITSIO for this purpose.

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

```
ftgkys - read a string keyword
ftpkls - write (append) a string keyword
ftikys - insert a string keyword
```

```
ftmkys - modify the value of an existing string keyword
ftukys - update an existing keyword, or write a new keyword
ftdkey - delete a keyword
```

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

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

4.9.2 Arrays of Strings in Binary Table Extensions

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

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

4.9.3 Keyword Units Strings

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

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

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

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

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

Chapter 5

Programming Guidelines

The FITSIO Cookbook (available in the file cookbook.tex in the FITSIO software distribution directory) contains annotated listings of various programs that read and write FITSIO files. New users of FITSIO should study these example programs to help learn how to correctly use the FITSIO library. The following sections briefly summarize the main steps in reading or writing a FITS file.

5.1 Reading an existing FITS file

The following sequence of subroutine calls illustrate a simple example of reading an existing FITS file. Refer to the FITSIO Cookbook for more detailed examples of the correct FITSIO usage.

```
 1. Open the file with FTOPEN.
 2. Read any desired header keywords with FTGHPR or FTGKYx.
 3. Read the primary data, if any, with FTGPVx or FTGPFx.
 4. Repeat steps 2 and 3 until all the desired information has been read.
 5. Move to another extension with FTMAHD or FTMRHD.
 6. Read any extension header keywords (e.g. with FTGHTB, FTGHBN or FTGKYx)
 7. Read any columns of data from the extension (e.g. with FTGCVx or FTGCFx)
 8. Repeat steps 6 and 7 until all the information has been read.
 9. Repeat steps 5 through 8 for any other extensions.
10. Close the file with FTCLOS.
```

5.2 Creating a new FITS file

The following sequence of subroutine calls illustrate a simple example of writing a new FITS file:

```
 1. Create the new file with FTINIT.
 2. Write the required primary array keywords with FTPHPR.
 3. Write any additional keywords with FTPKYx.
 4. Write the primary array data, if any, with FTPPRx.
 5. Create another extension, if desired, with FTCRHD.
 6. Write required header keywords for the extension with FTPHTB or FTPHBN.
 7. Write any additional keywords with FTPKYx
 8. Write data to the extension, one column at a time with FTPCLx
```

```
 9. Repeat steps 5 - 8 for any more extensions.
10. Close the FITS file with FTCLOS
```

Note that the application program must not explicitly write the required 'END' keyword at the end of each header; the FITSIO interface will automatically append the END record whenever the header is closed. As a standard practice, users should always read back any FITS files that they have created to ensure that the header values and data structure are correct and self-consistent.

5.3 When the Final Size of the FITS File 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 FTMKYJ subroutine before moving to another extension or closing the FITS file.

A similar procedure may be used in the case of FITS tables, where the number of rows in the table (the NAXIS2 value) may initially be set to 1 and then updated with the correct value before closing the table. Alternatively, one may use the FTIROW routine to insert additional rows into a table if the original value turns out to be too small. This latter method must be used if any of the columns contain variable length arrays.

5.4 Optimizing Code for Maximum Processing Speed

Care must be taken when designing software to achieve the best possible performance when processing the FITS data files. The following paragraphs describe some strategies that may be used to improve the processing speed of software that uses FITSIO.

1. 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. When reading or writing large chunks of contiguous data in the FITS file (at least 3 FITS blocks = 8640 bytes) FITSIO bypasses the internal buffers that it uses for small pieces of data (e.g., when reading FITS keywords). This is more efficient because the data are not copied to the intermediate buffer.

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

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

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

Occasionally it is necessary to simultaneously access more than one FITS table, for example when transfering values from an input table to an output table. In cases like this, one should call ftgrsz to get the optimal number of rows for each table separately, than reduce the number of rows proportionally. For example, if the optimal number of rows in the input table is 3600 and is 1400 in the output table, then these values should be cut in half to 1800 and 700, respectively, if both tables are going to be accessed at the same time.

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

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

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

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

7. Design FITS binary tables so that every column is aligned on a computer word boundary and so that each row is a multiple number of computer words in length. Accessing non-aligned words can be slower on some machines. This is usually not a problem when using FITSIO to read or write the FITS files, but other FITS readers and writers could be affected. In practice, this means that double precision columns should start at a multiple of 8 bytes within the row, single precision floating point columns and integer columns should start at a multiple of 4 bytes, and short integer columns should start at a multiple of 2 bytes. If necessary, the row length should be padded out by adding a dummy column of the appropriate width or by adjusting the width of an existing column so that the row length is also a multiple number of words in length. For example, if a binary table contains a '1B', a '1E', and a '1D' column, then the optimum design would place the '1D' column first in the table followed by the '1E' and then the '1B' column. Since the row length is then $8 + 4 + 1 = 13$ bytes, one should add another dummy column, with a 3A datatype to make the length a multiple of the double precision word length. Alternatively, one could change the last column from '1B' to '4B'. This will insure that all the data values are optimally aligned.

8. Where feasible, design FITS binary tables so that the columns of data are written as a contiguous set of bytes, rather than as single elements in multiple rows. For example, it is much faster to access the data in a table that contains a single row and 2 columns with TFORM keywords equal to '1000E' and '1000J', than it is to access the same amount of data in a table with 1000 rows which has columns with the TFORM keywords equal to '1E' and '1J'. In the former case the 1000 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 FITSIO has to explicitly move to the position of each element to be read or written.

9. Avoid the use of variable length vector columns in binary tables, since any reading or writing of these data requires that FITSIO first look up or compute the starting address of each row of data in the heap.

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

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

5.5 FITSIO Array Size Limitations

In general, FITSIO places no limits on the sizes of the FITS files that it reads or writes. In particular there is no internal limit on the size of the dimensions of the primary array or IMAGE extension. Table extensions may have up to 999 columns (the maximum allowed by the FITS standard) and may have an arbitrarily large number of rows. There are a few other limits, however, which may affect some extreme cases:

1. The maximum number of FITS files that may be simultaneously opened by a single program is defined by the 'nb' parameter and is currently set to 20. To change this limit, one must globally edit the following parameter statement:

```
parameter (nb = 20)
```

throughout the fitsio.f and the machine-specific source files. The number of physical buffers, as defined by the 'pb' parameter in the machine specific source file, must be set greater than or equal to the value of 'nb'. Increasing the value of 'pb' will also require that the size of the ftbuff common block be modified accordingly.

2. The total number of FITS table columns that can be accessed by FITSIO, when summed over the current HDU in all the open FITS files, cannot exceed the value of the 'nf' parameter in the fitsio.f

source code. This is currently set to 3000, which, for example, will allow FITSIO to simultaneously read or write 3 FITS tables with 999 columns each, or 20 tables with 150 columns each, or any other combination that does not exceed 20 files and 3000 total columns. (A FITS primary array or image extension is equivalent to a 2 column table). FITSIO allocates 56 bytes of memory for each column to store various descriptive information so substantially raising or lowering the value of 'nf' can have a significant effect on the total amount of computer memory required to run a program.

3. The maximum number of extensions that can be read or written in a single FITS file is current set to 512 as defined by the 'ne' parameter statements throughout the fitsio.f and machine specific source code files. This value may be increased if necessary, but the access times to the later extensions in such files may become very long.

4. FITSIO can handle FITS files up to about 2.1 GB in size which is the maximum value of a signed long integer. Some machines that use 8-byte words for a long integer may support larger files, but this has not been tested.

Chapter 6

Basic Interface Routines

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

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

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

6.1 File I/O Routines

1 Open an existing FITS file with readonly or readwrite access

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

2 Open and initialize a new empty FITS file

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

3 Close a FITS file previously opened with ftopen or ftinit

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

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

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

5 Create a primary array (if none already exists), or insert a new IMAGE 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 image extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

6 Insert a new ASCII TABLE extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new table extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

7 Insert a new binary table extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new bintable extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

6.2 Keyword I/O Routines

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

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

2 Put (append) a new keyword of the appropriate datatype into the CHU.

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

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

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

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

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

5 Delete an existing keyword record.

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

6.3 Data I/O Routines

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

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

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

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

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

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

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

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

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

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

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

Chapter 7

Advanced Interface Subroutines

This chapter defines all the available subroutines in the FITSIO user interface. For completeness, the basic subroutines described in the previous chapter are also repeated here. A right arrow symbol is used here to separate the input parameters from the output parameters in the definition of each subroutine. This symbol is not actually part of the calling sequence. An alphabetical list and definition of all the parameters is given at the end of this section. The SPP interface subroutines have the same arguments but have names that begin with 'fs' rather than 'ft'.

7.1 FITS File Open and Close Subroutines:

1 Open an existing FITS file with readonly or readwrite access

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

2 Open and initialize a new empty FITS file

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

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

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

4 Close a FITS file previously opened with ftopen or ftinit

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

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

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

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

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

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

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

7.2 HDU-Level Operations

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

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

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

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

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

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

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

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

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

5 Insert a new IMAGE 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 image extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

6 Insert a new ASCII TABLE extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new table extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

7 Insert a new binary table extension immediately following the CHDU. Any following extensions will be shifted down to make room for the new extension. If there are no other following extensions then the new bintable extension will simply be appended to the end of the file. The new extension will become the CHDU.

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

8 Resize an image by modifing the size, dimensions, and/or datatype of the current primary array or image extension. If the new image, as specified by the input arguments, is larger than the current existing image in the FITS file then zero fill data will be inserted at the end of the current image and any following extensions will be moved further back in the file. Similarly, if the new image is smaller than the current image then any following extensions will be shifted up towards the beginning of the FITS file and the image data will be truncated to the new size. This routine rewrites the BITPIX, NAXIS, and NAXISn keywords with the appropriate values for new image.

```
FTRSIM(unit,bitpix,naxis,naxes,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. This routine will only delete extensions; the primary array (the first HDU in the file) cannot be deleted. Note that Fortran (unfortunately) does not have the facility to decrease the size of an existing file, therefore the physical size of the FITS file will not change and the end of the file will be padded out with zeros to fill in the space left after the CHDU is deleted. If there are more extensions in the file following the one that is deleted, then the the CHDU will be defined to point to the following extension. If there are no following extensions then the CHDU will be redefined to point to the previous extension (or the primary array if there was only one extension in the file). The output HDUTYPE parameter indicates the type of the new CHDU after the previous CHDU has been deleted.

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

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

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

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

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

7.3 Define or Redefine the structure of the CHDU

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

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

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

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

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

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

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

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

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

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

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

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

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

7.4 FITS Header I/O Subroutines

7.4.1 Header Space and Position Routines

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

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

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

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

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

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

7.4.2 Read or Write Standard Header Routines

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

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

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

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

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

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

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

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

4 Get the ASCII table header keywords from the CHU

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

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

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

6 Get the binary table header keywords from the CHU

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

7.4.3 Write Keyword Subroutines

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

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

2 Put (append) a COMMENT keyword into the CHU. Multiple COMMENT keywords will be written if the input comment string is longer than 70 characters.

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

3 Put (append) a HISTORY keyword into the CHU. Multiple HISTORY keywords will be written if the input history string is longer than 70 characters.

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

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

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

5 Put (append) a new keyword of the appropriate datatype into the CHU. Note that FTPKYS will only write string values up to 68 characters in length; longer strings will be truncated. The FTPKLS routine can be used to write longer strings, using a non-standard FITS convention.

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

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

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

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

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

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

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

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

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

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

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

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

7.4.4 Insert Keyword Subroutines

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

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

2 Insert a new keyword into the CHU. The new keyword is inserted immediately following the last keyword that has been read from the header. These 'insert keyword' subroutines are somewhat less efficient then the 'append keyword' subroutines described above because the remaining keywords in the header have to be shifted down one slot.

```
FTIKY[JLS](unit,keyword,keyval,comment, > status)
FTIKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
```

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

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

7.4.5 Read Keyword Subroutines

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

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

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

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

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

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

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

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

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

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

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

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

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

7 Get a sequence of numbered keyword values

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

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

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

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

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

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

7.4.6 Modify Keyword Subroutines

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

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

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

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

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

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

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

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

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

5 Modify the value and comment fields of an existing keyword in the CHU. 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 (&).

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

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

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

7.4.7 Update Keyword Subroutines

1 Update an 80-character record in the CHU. If the specified keyword already exists then that header record will be replaced with the input CARD string. If it does not exist then the new record will be added to the header.

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

2 Update the value and comment fields of a keyword in the CHU. The specified keyword is modified if it already exists (by calling FTMKYx) otherwise a new keyword is created by calling FTPKYx.

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

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

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

7.4.8 Delete Keyword Subroutines

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

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

7.5 Data Scaling and Undefined Pixel Parameters

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

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

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

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

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

3 Define the integer value to be used to signify undefined pixels in the primary array or image extension. This is only used if BITPIX = 8, 16, or 32. This does not create or change the value of the BLANK keyword in the header.

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

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

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

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

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

7.6 FITS Primary Array or IMAGE Extension I/O Subroutines

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

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

1 Put elements into the data array

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

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

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

3 Set data array elements as undefined

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

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

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

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

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

6 Put values into group parameters

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

7 Get values from group parameters

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

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

8 Put 2-D image into the data array

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

9 Put 3-D cube into the data array

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

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

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

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

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

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

12 Put an arbitrary data subsection into the data array.

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

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

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

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

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

7.7 FITS ASCII and Binary Table Data I/O Subroutines

7.7.1 Column Information Subroutines

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

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

Note that the FITS Standard recommends that only letters, digits, and the underscore character be used in column names (with no embedded spaces in the name). Trailing blank characters are not significant. It is recommended that the column names in a given table be unique within the first 8 characters.

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

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

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

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

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

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

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

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

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

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

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

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

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

7.7.2 Low-Level Table Access Subroutines

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

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

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

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

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

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

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

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

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

7.7.3 High-Level Table Access Subroutines

These subroutines put or get data values in the current ASCII or Binary table extension. Automatic data type conversion is performed for numerical data types (B,I,J,E,D) if the data type of the column (defined by the TFORM keyword) differs from the data type of the calling subroutine. The data values are also scaled by the TSCALn and TZEROn header values as they are being written to or read from the FITS array. The fttscl subroutine MUST be used to define the scaling parameters when writing data to the table or to override the default scaling values given in the header when reading from the table.

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

1 Insert blank rows into an existing ASCII or binary table (in the CDU). All the rows FOLLOWING row FROW are shifted down by NROWS rows. If FROW = 0 then the blank rows are inserted at the beginning of the table. This routine modifies the NAXIS2 keyword to reflect the new number of rows in the table.

```
FTIROW(unit,frow,nrows, > status)
```

2 Delete rows from an existing ASCII or binary table (in the CDU). The NROWS number of rows are deleted, starting with row FROW, and any remaining rows in the table are shifted up to fill in the space. This routine modifies the NAXIS2 keyword to reflect the new number of rows in the table. Note that the physical size of the FITS file will not be reduced by this operation, and the empty FITS blocks if any at the end of the file will be padded with zeros.

```
FTDROW(unit,frow,nrows, > status)
```

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

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

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

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

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

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

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

```
FTPCN[BIJED](unit,colnum,frow,felem,nelements,values,nullval > status)
```

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

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

8 Set table elements in a column as undefined

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

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

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

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

```
FTGCF[SLBIJEDCM](unit,colnum,frow,felem,nelements, >
                 values,flagvals,anyf,status)
```

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

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

12 Get an arbitrary data subsection from an N-dimensional array in a binary table vector column. Any Undefined pixels in the array will have the corresponding 'flagvals' element set equal to

.TRUE. The first and last rows in the table to be read are specified by fpixels(naxis+1) and lpixels(naxis+1), and hence are treated as the next higher dimension of the FITS N-dimensional array. The INCS parameter specifies the sampling interval in each dimension between the data elements that will be returned.

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

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

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

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

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

15 Get the descriptor for a variable length column in a binary table. The descriptor consists of 2 integer parameters: the number of elements in the array and the starting offset relative to the start of the heap.

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

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

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

7.8 Celestial Coordinate System Subroutines

The following subroutines are provided to help calculate the transformation between pixel location in an image and the corresponding celestial coordinates on the sky. These 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. (Note: these subroutines are provisional and may change slightly in a future release of FITSIO).

1 Get the values of all the standard FITS celestial coordinate system keywords from the header of a FITS image (i.e., the primary array or an image extension). These values may then be passed to the subroutines that perform the coordinate transformations.

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

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

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

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

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

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

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

7.9 File Checksum Subroutines

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

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

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

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

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

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

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

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

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

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

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

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

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

7.10 General Utility Subroutines

The following utility subroutines may be useful for certain applications:

1 Return the revision number of the fitsio library. This subroutine returns the current revision number of the FITSIO software library. The revision number will be incremented whenever any modifications or enhancements are made to the code.

```
FTVERS( > version)
```

2 Return a code value indicating the word architecture of the computer that the program is running on. The routine compares the internal architecture used to store floating point and integer numbers and returns one of the following codes:

```
FTARCH( > compid)

   compid = 1  -  VAX or Alpha VMS system
            2  -  Decstation or Alpha OSF/1, or IBM PC
            3  -  SUN workstation
            4  -  IBM mainframe
            0  -  unknown type of machine
```

3 Get the current system date. The day, month and year are returned as integers. The year value ranges from 00 to 99.

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

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

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

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

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

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

```
FTGMSG( > errmsg)
```

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

```
FTPMSG(errmsg)
```

8 Clear the entire error message stack. This routine is useful to clear any error message that may have been generated by a non-fatal FITSIO error (such as failing to find an optional header keyword). This routine is called without any arguments.

```
FTCMSG
```

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

```
FTUPCH(string)
```

10 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 'reference string. If CASESN = .true. then the match will be case sensitive. The returned MATCH parameter will be .true. if the 2 strings match, and EXACT will be .true. if the match is exact (i.e., if no wildcard characters were used in the match). Both strings must be 68 characters or less in length.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

EXAMPLES: The following lines illustrate valid input template strings:

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

Chapter 8

Summary of all FITSIO User-Interface Subroutines

FITS File Open and Close Subroutines: page 33

```
FTOPEN(unit,filename,rwmode, > blocksize,status)
FTINIT(unit,filename,blocksize, > status)
FTFLUS(unit, > status)
FTCLOS(unit, > status)
FTDELT(unit, > status)
FTGIOU( > iounit, status)
FTFIOU(iounit, > status)
```

HDU-Level Operations: page 34

```
FTMAHD(unit,nhdu, > hdutype,status)
FTMRHD(unit,nmove, > hdutype,status)
FTGHDN(unit, > nhdu)
FTCRHD(unit, > status)
FTIIMG(unit,bitpix,naxis,naxes, > status)
FTITAB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTIBIN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTRSIM(unit,bitpix,naxis,naxes,status)
FTDHDU(unit, > hdutype,status)
FTCOPY(iunit,ounit,morekeys, > status)
FTCPDT(iunit,ounit, > status)
```

Subroutines to specify or modify the structure of the CHDU: page 36

```
FTRDEF(unit, > status)  (DEPRECATED)
FTPDEF(unit,bitpix,naxis,naxes,pcount,gcount, > status)  (DEPRECATED)
FTADEF(unit,rowlen,tfields,tbcol,tform,nrows > status)  (DEPRECATED)
FTBDEF(unit,tfields,tform,varidat,nrows > status)  (DEPRECATED)
FTDDEF(unit,bytlen, > status)  (DEPRECATED)
FTPTHP(unit,theap, > status)
```

Header Space and Position Subroutines: page 37

```
FTHDEF(unit,morekeys, > status)
FTGHSP(iunit, > keysexist,keysadd,status)
FTGHPS(iunit, > keysexist,key_no,status)
```

Read or Write Standard Header Subroutines: page 37

```
FTPHPS(unit,bitpix,naxis,naxes, > status)
FTPHPR(unit,simple,bitpix,naxis,naxes,pcount,gcount,extend, > status)
FTGHPR(unit,maxdim, > simple,bitpix,naxis,naxes,pcount,gcount,extend,
       status)
FTPHTB(unit,rowlen,nrows,tfields,ttype,tbcol,tform,tunit,extname, >
       status)
FTGHTB(unit,maxdim, > rowlen,nrows,tfields,ttype,tbcol,tform,tunit,
       extname,status)
FTPHBN(unit,nrows,tfields,ttype,tform,tunit,extname,varidat > status)
FTGHBN(unit,maxdim, > nrows,tfields,ttype,tform,tunit,extname,varidat,
       status)
```

Write Keyword Subroutines: page 39

```
FTPREC(unit,card, > status)
FTPCOM(unit,comment, > status)
FTPHIS(unit,history, > status)
FTPDAT(unit, > status)
FTPKY[JLS](unit,keyword,keyval,comment, > status)
FTPKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTPKLS(unit,keyword,keyval,comment, > status)
FTPLSW(unit, > status)
FTPKYU(unit,keyword,comment, > status)
FTPKN[JLS](unit,keyroot,startno,no_keys,keyvals,comments, > status)
FTPKN[EDFG](unit,keyroot,startno,no_keys,keyvals,decimals,comments, >
           status)
FTPKYT(unit,keyword,intval,dblval,comment, > status)
FTPUNT(unit,keyword,units, > status)
```

Insert Keyword Subroutines: page 40

```
FTIREC(unit,key_no,card, > status)
FTIKY[JLS](unit,keyword,keyval,comment, > status)
FTIKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTIKYU(unit,keyword,comment, > status)
```

Read Keyword Subroutines: page 41

```
FTGREC(unit,key_no, > card,status)
FTGKYN(unit,key_no, > keyword,value,comment,status)
FTGCRD(unit,keyword, > card,status)
FTGNXK(unit,inclist,ninc,exclist,nexc, > card,status)
```

```
FTGKEY(unit,keyword, > value,comment,status)
FTGKY[EDJLS](unit,keyword, > keyval,comment,status)
FTGKN[EDJLS](unit,keyroot,startno,max_keys, > keyvals,nfound,status)
FTGKYT(unit,keyword, > intval,dblval,comment,status)
FTGUNT(unit,keyword, > units,status)
```

Modify Keyword Subroutines: page 42

```
FTMREC(unit,key_no,card, > status)
FTMCRD(unit,keyword,card, > status)
FTMNAM(unit,oldkey,keyword, > status)
FTMCOM(unit,keyword,comment, > status)
FTMKY[JLS](unit,keyword,keyval,comment, > status)
FTMKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTMKYU(unit,keyword,comment, > status)
```

Update Keyword Subroutines: page 43

```
FTUCRD(unit,keyword,card, > status)
FTUKY[JLS](unit,keyword,keyval,comment, > status)
FTUKY[EDFG](unit,keyword,keyval,decimals,comment, > status)
FTUKYU(unit,keyword,comment, > status)
```

Delete Keyword Subroutines: page 43

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

Define Data Scaling Parameters and Undefined Pixel Flags: page 43

```
FTPSCL(unit,bscale,bzero, > status)
FTTSCL(unit,colnum,tscal,tzero, > status)
FTPNUL(unit,blank, > status)
FTSNUL(unit,colnum,snull > status)
FTTNUL(unit,colnum,tnull > status)
```

FITS Primary Array or IMAGE Extension I/O Subroutines: page 44

```
FTPPR[BIJED](unit,group,fpixel,nelements,values, > status)
FTPPN[BIJED](unit,group,fpixel,nelements,values,nullval > status)
FTPPRU(unit,group,fpixel,nelements, > status)
FTGPV[BIJED](unit,group,fpixel,nelements,nullval, > values,anyf,status)
FTGPF[BIJED](unit,group,fpixel,nelements, > values,flagvals,anyf,status)
FTPGP[BIJED](unit,group,fparm,nparm,values, > status)
FTGGP[BIJED](unit,group,fparm,nparm, > values,status)
FTP2D[BIJED](unit,group,dim1,naxis1,naxis2,image, > status)
FTP3D[BIJED](unit,group,dim1,dim2,naxis1,naxis2,naxis3,cube, > status)
FTG2D[BIJED](unit,group,nullval,dim1,naxis1,naxis2, > image,anyf,status)
FTG3D[BIJED](unit,group,nullval,dim1,dim2,naxis1,naxis2,naxis3, >
             cube,anyf,status)
```

```
FTPSS[BIJED](unit,group,naxis,naxes,fpixels,lpixels,array, > status)
FTGSV[BIJED](unit,group,naxis,naxes,fpixels,lpixels,incs,nullval, >
             array,anyf,status)
FTGSF[BIJED](unit,group,naxis,naxes,fpixels,lpixels,incs, >
             array,flagvals,anyf,status)
```

Table Column Information Subroutines: page 47

```
FTGCNO(unit,casesen,coltemplate, > colnum,status)
FTGCNN(unit,casesen,coltemplate, > colnam,colnum,status)
FTGTCL(unit,colnum, > datacode,repeat,width,status)
FTGACL(unit,colnum, >
       ttype,tbcol,tunit,tform,tscal,tzero,snull,tdisp,status)
FTGBCL(unit,colnum, >
       ttype,tunit,datatype,repeat,tscal,tzero,tnull,tdisp,status)
FTPTDM(unit,colnum,naxis,naxes, > status)
FTGTDM(unit,colnum,maxdim, > naxis,naxes,status)
FFGRSZ(unit, > nrows,status)
```

Low-Level Table Access Subroutines: page 48

```
FTGTBS(unit,frow,startchar,nchars, > string,status)
FTPTBS(unit,frow,startchar,nchars,string, > status)
FTGTBB(unit,frow,startchar,nchars, > array,status)
FTPTBB(unit,frow,startchar,nchars,array, > status)
```

High-Level Table Access Subroutines: page 49

```
FTIROW(unit,frow,nrows, > status)
FTDROW(unit,frow,nrows, > status)
FTICOL(unit,colnum,ttype,tform, > status)
FTICLS(unit,colnum,ncols,ttype,tform, > status)
FTDCOL(unit,colnum, > status)
FTPCL[SLBIJEDCM](unit,colnum,frow,felem,nelements,values, > status)
FTPCN[BIJED](unit,colnum,frow,felem,nelements,values,nullval > status)
FTPCLX(unit,colnum,frow,fbit,nbit,lray, > status)
FTPCLU(unit,colnum,frow,felem,nelements, > status)
FTGCL(unit,colnum,frow,felem,nelements, > values,status)
FTGCV[SBIJEDCM](unit,colnum,frow,felem,nelements,nullval, >
                values,anyf,status)
FTGCF[SLBIJEDCM](unit,colnum,frow,felem,nelements, >
                  values,flagvals,anyf,status)
FTGSV[BIJED](unit,colnum,naxis,naxes,fpixels,lpixels,incs,nullval, >
             array,anyf,status)
FTGSF[BIJED](unit,colnum,naxis,naxes,fpixels,lpixels,incs, >
             array,flagvals,anyf,status)
FTGCX(unit,colnum,frow,fbit,nbit, > lray,status)
FTGCX[IJD](unit,colnum,frow,nrows,fbit,nbit, > array,status)
FTGDES(unit,colnum,rownum, > nelements,offset,status)
FTPDES(unit,colnum,rownum,nelements,offset, > status)
```

Celestial Coordinate System Subroutines: page 52

```
FTGICS(unit, > xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
FTGTCS(unit,xcol,ycol, >
       xrval,yrval,xrpix,yrpix,xinc,yinc,rot,coordtype,status)
FTWLDP(xpix,ypix,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                  coordtype, > xpos,ypos,status)
FTXYPX(xpos,ypos,xrval,yrval,xrpix,yrpix,xinc,yinc,rot,
                  coordtype, > xpix,ypix,status)
```

File Checksum Subroutines: page 53

```
FTPCKS(unit, > status)
FTUCKS(unit, > status)
FTVCKS(unit, > dataok,hduok,status)
FTGCKS(unit, > datasum,hdusum,status)
FTESUM(sum,complement, > checksum)
FTDSUM(checksum,complement, > sum)
```

General Utility Subroutines: page 54

```
FTVERS( > version)
FTARCH( > compid)
FTGSDT(> day,month,year,status)
FTGHAD(iunit, > curaddr,nextaddr)
FTGERR(status, > errtext)
FTGMSG( > errmsg)
FTPMSG(errmsg)
FTCMSG
FTUPCH(string)
FTCMPS(str_template,string,casesen, > match,exact)
FTTKEY(keyword, > status)
FTPSVC(card, > value,comment,status)
FTKEYN(keyroot,seq_no, > keyword,status)
FTNKEY(seq_no,keyroot, > keyword,status)
FTDTYP(value, > dtype,status)
FTASFM(tform, > datacode,width,decimals,status)
FTBNFM(tform, > datacode,repeat,width,status)
FTGABC(tfields,tform,space, > rowlen,tbcol,status)
FTGTHD(template, > card,hdtype,status)
```

Chapter 9

Parameter Definitions

```
anyf - (logical) set to TRUE if any of the returned data values are undefined
array - (any datatype except character) array of bytes to be read or written.
bitpix - (integer) bits per pixel: 8, 16, 32, -32, or -64
blank - (integer) value used for undefined pixels in integer primary array
blocksize - (integer) 2880-byte logical record blocking factor
          (if 0 < blocksize < 11) or the actual block size in bytes
          (if 10 < blocksize < 28800).  As of version 3.3 of FITSIO,
          blocksizes greater than 2880 are no longer supported.
bscale - (double precision) scaling factor for the primary array
bytlen - (integer) length of the data unit, in bytes
bzero - (double precision) zero point for primary array scaling
card - (character*80) header record to be read or written
casesen - (logical) will string matching be case sensitive?
checksum - (character*16) encoded checksum string
colname - (character) ASCII name of the column
colnum - (integer) number of the column (first column = 1)
coltemplate - (character) template string to be matched to column names
comment - (character) the keyword comment field
comments - (character array) keyword comment fields
compid - (integer) the type of computer that the program is running on
complement - (logical) should the checksum be complemented?
coordtype - (character) type of coordinate projection (-SIN, -TAN, -ARC,
          -NCP, -GLS, -MER, or -AIT)
cube - 3D data cube of the appropriate datatype
curaddr - (integer) starting address (in bytes) of the CHDU
datacode - (integer) symbolic code of the binary table column datatype
dataok - (integer) was the data unit verification successful (=1) or
         not (= -1).  Equals zero if the DATASUM keyword is not present.
datasum - (double precision) 32-bit 1's complement checksum for the data unit
datatype - (character) datatype (format) of the binary table column
day - (integer) current day of the month
dblval - (double precision) fractional part of the keyword value
decimals - (integer) number of decimal places to be displayed
dim1 - (integer) actual size of the first dimension of the image or cube array
dim2 - (integer) actual size of the second dimension of the cube array
dtype - (character) datatype of the keyword ('C', 'L', 'I',  or 'F')
```

```
                C = character string
                L = logical
                I = integer
                F = floating point number
errmsg - (character*80) oldest error message on the internal stack
errtext - (character*30) descriptive error message corresponding to error number
casesen - (logical) true if column name matching is case sensitive
exact - (logical) do the strings match exactly, or were wildcards used?
exclist  (character array) list of names to be excluded from search
extend - (logical) true if there may be extensions following the primary data
extname - (character) value of the EXTNAME keyword (if not blank)
fbit - (integer) first bit in the field to be read or written
felem - (integer) first pixel of the element vector (ignored for ASCII tables)
filename - (character) name of the FITS file
flagvals - (logical array) True if corresponding data element is undefined
fparm - (integer) sequence number of the first group parameter to read or write
fpixel - (integer) the first pixel position
fpixels - (integer array) the first included pixel in each dimension
frow - (integer) beginning row number (first row of table = 1)
gcount - (integer) value of the GCOUNT keyword (usually = 1)
group - (integer) sequence number of the data group (=0 for non-grouped data)
hdtype - (integer) header record type: -1=delete;  0=append or replace;
                   1=append; 2=this is the END keyword
hduok - (integer) was the HDU verification successful (=1) or
         not (= -1).  Equals zero if the CHECKSUM keyword is not present.
hdusum - (double precsion) 32 bit 1's complement checksum for the entire CHDU
hdutype - (integer) type of HDU: 0 = primary array or IMAGE, 1 = ASCII table,
                   2 = binary table, -1 = unknown
history - (character) the HISTORY keyword comment string
image - 2D image of the appropriate datatype
inclist  (character array) list of names to be included in search
incs - (integer array) sampling interval for pixels in each FITS dimension
intval - (integer) integer part of the keyword value
iounit - (integer) value of an unused I/O unit number
iunit - (integer) logical unit number associated with the input FITS file, 1-199
key_no - (integer) sequence number (starting with 1) of the keyword record
keyroot - (character) root string for the keyword name
keysadd -(integer) number of new keyword records which can fit in the CHU
keysexist - (integer) number of existing keyword records in the CHU
keyval - value of the keyword in the appropriate datatype
keyvals - (array) value of the keywords in the appropriate datatype
keyword - (character*8) name of a keyword
lray - (logical array) array of logical values corresponding to the bit array
lpixels - (integer array) the last included pixel in each dimension
match - (logical) do the 2 strings match?
maxdim - (integer) dimensioned size of the NAXES, TTYPE, TFORM or TUNIT arrays
max_keys - (integer) maximum number of keywords to search for
month - (integer) current month of the year (1 - 12)
morekeys - (integer) will leave space in the header for this many more keywords
naxes - (integer array) size of each dimension in the FITS array
naxis - (integer) number of dimensions in the FITS array
naxis1 - (integer) length of the X/first axis of the FITS array
```

```
naxis2 - (integer) length of the Y/second axis of the FITS array
naxis3 - (integer) length of the Z/third axis of the FITS array
nbit - (integer) number of bits in the field to read or write
nchars - (integer) number of characters to read and return
ncols - (integer) number of columns
nelements - (integer) number of data elements to read or write
nexc   (integer)  number of names in the exclusion list (may = 0)
nhdu - (integer) absolute number of the HDU (1st HDU = 1)
ninc   (integer)  number of names in the inclusion list
nmove - (integer) number of HDUs to move (+ or -), relative to current position
nfound - (integer) number of keywords found (highest keyword number)
no_keys - (integer) number of keywords to write in the sequence
nparm - (integer) number of group parameters to read or write
nrows - (integer) number of rows in the table
nullval - value to represent undefined pixels, of the appropriate datatype
nextaddr - (integer) starting address (in bytes) of the HDU following the CHDU
offset - (integer) byte offset in the heap to the first element of the array
oldkey - (character) old name of keyword to be modified
ounit - (integer) logical unit number associated with the output FITS file 1-199
pcount - (integer) value of the PCOUNT keyword (usually = 0)
repeat - (integer) length of element vector (e.g. 12J); ignored for ASCII table
rot - (double precision) celestial coordinate rotation angle (degrees)
rowlen - (integer) length of a table row, in characters or bytes
rownum - (integer) number of the row (first row = 1)
rwmode - (integer) file access mode: 0 = readonly, 1 = readwrite
seq_no - (integer) the sequence number to append to the keyword root name
simple - (logical) does the FITS file conform to all the FITS standards
snull - (character) value used to represent undefined values in ASCII table
space - (integer) number of blank spaces to leave between ASCII table columns
startchar - (integer) first character in the row to be read
startno - (integer) value of the first keyword sequence number (usually 1)
status - (integer) returned error status code (0 = OK)
str_template (character) template string to be matched to reference string
string - (character) character string
sum - (double precision) 32 bit unsigned checksum value
tbcol - (integer array) column number of the first character in the field(s)
tdisp - (character) Fortran type display format for the table column
template-(character) template string for a FITS header record
tfields - (integer) number of fields (columns) in the table
tform - (character array) format of the column(s); allowed values are:
         For ASCII tables:  Iw, Aw, Fww.dd, Eww.dd, or Dww.dd
         For binary tables: rL, rX, rB, rI, rJ, rA, rAw, rE, rD, rC, rM
         where 'w'=width of the field, 'd'=no. of decimals, 'r'=repeat count
         Note that the 'rAw' form is non-standard extension to the
         TFORM keyword syntax that is not specifically defined in the
         Binary Tables definition document.
theap - (integer) zero indexed byte offset of starting address of the heap
         relative to the beginning of the binary table data
tnull - (integer) value used to represent undefined values in binary table
ttype - (character array) label for table column(s)
tscal - (double precision) scaling factor for table column
tunit - (character array) physical unit for table column(s)
```

```
tzero - (double precision) scaling zero point for table column
unit  - (integer) logical unit number associated with the FITS file (1-199)
units - (character) the keyword units string (e.g., 'km/s')
value - (character) the keyword value string
values - array of data values of the appropriate datatype
varidat - (integer) size in bytes of the 'variable length data area'
          following the binary table data (usually = 0)
version - (real) current revision number of the library
width - (integer) width of the character string field
xcol - (integer) number of the column containing the X coordinate values
xinc - (double precision) X axis coordinate increment at reference pixel (deg)
xpix - (double precision) X axis pixel location
xpos - (double precision) X axis celestial coordinate (usually RA) (deg)
xrpix - (double precision) X axis reference pixel array location
xrval - (double precision) X axis coordinate value at the reference pixel (deg)
ycol - (integer) number of the column containing the X coordinate values
year - (integer) last 2 digits of the year (00 - 99)
yinc - (double precision) Y axis coordinate increment at reference pixel (deg)
ypix - (double precision) y axis pixel location
ypos - (double precision) y axis celestial coordinate (usually DEC) (deg)
yrpix - (double precision) Y axis reference pixel array location
yrval - (double precision) Y axis coordinate value at the reference pixel (deg)
```

Chapter 10

FITSIO Error Status Codes

```
Status codes in the range -99 to -999  and 1 to 999 are reserved for future
FITSIO use.

  0  OK, no error
101  illegal logical unit number; must be between 1 - 199, inclusive
102  too many FITS files open at once; all internal buffers full
103  unable to find requested file to open; does it exist?
104  error opening existing file
105  error creating new FITS file; (does a file with this name already exist?)
106  error writing record to FITS file
107  end-of-file encountered while reading record from FITS file
108  error reading record from file
109  FITS record blocking factor out of range; must be between 1-2880
110  error closing FITS file
111  too many fields in table; internal array dimensions too small
112  Cannot modify file with readonly access
113  Version of FITSIO is incompatible with computer it is running on
114  All available I/O unit numbers have already been allocated (by ftgiou)

201  header not empty; can't write required keywords
202  specified keyword name was not found in the header
203  specified header record number is out of bounds
204  keyword value field is blank
205  keyword value string is missing the closing quote character
206  error while appending sequence number to keyword root name
207  illegal character in keyword name or header record
208  keyword does not have expected name. Keyword out of sequence?
209  keyword does not have expected integer value
210  could not find the required END header keyword
211  illegal BITPIX keyword value
212  illegal NAXIS keyword value
213  illegal NAXISn keyword value: must be 0 or positive integer
214  illegal PCOUNT keyword value
215  illegal GCOUNT keyword value
216  illegal TFIELDS keyword value
217  illegal ASCII or binary table width value (NAXIS1)
```

```
218  illegal number of rows in ASCII or binary table (NAXIS2)
219  column name (TTYPE keyword) not found
220  illegal SIMPLE keyword value
221  could not find the required SIMPLE header keyword
222  could not find the required BITPIX header keyword
223  could not find the required NAXIS header keyword
224  could not find all the required NAXISn keywords in the header
225  could not find the required XTENSION header keyword
226  the CHDU is not an ASCII table extension
227  the CHDU is not a binary table extension
228  could not find the required PCOUNT header keyword
229  could not find the required GCOUNT header keyword
230  could not find the required TFIELDS header keyword
231  could not find all the required TBCOLn keywords in the header
232  could not find all the required TFORMn keywords in the header
233  the CHDU is not an IMAGE extension
234  illegal TBCOL keyword value; out of range
235  this operation only allowed for ASCII or BINARY table extension
236  column is too wide to fit within the specified width of the ASCII table
237  the specified column name template matched more than one column name
241  binary table row width is not equal to the sum of the field widths
251  unrecognizable type of FITS extension
252  unrecognizable FITS record
253  END keyword contains non-blank characters in columns 9-80
254  Header fill area contains non-blank characters
255  Data fill area contains non-blank on non-zero values
261  unable to parse the TFORM keyword value string
262  unrecognizable TFORM datatype code
263  illegal TDIMn keyword value

301  illegal HDU number; less than 1 or greater than internal buffer size
302  column number out of range (1 - 999)
303  must define data unit structure before attempting this operation
304  attempt to move to negative file record number
305  cannot move directly to specified HDU: unknown starting location
306  attempted to read or write a negative number of bytes in the FITS file
307  illegal starting row number for table read or write operation
308  illegal starting element number for table read or write operation
309  attempted to read or write character string in non-character table column
310  attempted to read or write logical value in non-logical table column
311  illegal ASCII table TFORM format code for attempted operation
312  illegal binary table TFORM format code for attempted operation
313  error while converting value to formatted character string
314  value for undefined pixels has not been defined
315  error during internal read of character string into integer or real value
316  illegal value in logical value column of the binary table
317  attempted to read or write descriptor in a non-descriptor field
318  attempted to read data from a zero length variable length array
319  attempted to read or write past end of variable length array
320  number of array dimensions out of range
321  first pixel number is greater than the last pixel number
322  attempt to set BSCALE or TSCALn scaling parameter = 0
```

323 illegal axis length less than 1

401 error attempting to convert an integer to a formatted character string
402 error attempting to convert a real value to a formatted character string
403 cannot convert a quoted string keyword to an integer
404 attempted to read a non-logical keyword value as a logical value
405 cannot convert a quoted string keyword to a real value
406 cannot convert a quoted string keyword to a double precision value
407 error attempting to read character string as an integer
408 error attempting to read character string as a real value
409 error attempting to read character string as a double precision value
411 illegal number of decimal places while formatting floating point value
412 numerical overflow during implicit datatype conversion

501 celestial angle too large for projection
502 bad celestial coordinate or pixel value
503 error in celestial coordinate calculation
504 unsupported type of celestial projection
505 required celestial coordinate keywords not found