

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [4.2 File Open/Close Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4 CFITSIO Routines](#) **[Contents](#)**

4.1 Error Reporting

```
void fits_report_error(FILE *stream, int status)
void fits_get_errstatus(int status, char *err_text)
float fits_get_version(float *version)
```

The first routine prints out information about any error that has occurred. Whenever any CFITSIO routine encounters an error it usually writes a message describing the nature of the error to an internal error message stack and then returns with a positive integer status value. Passing the error status value to this routine will cause a generic description of the error and all the messages from the internal CFITSIO error stack to be printed to the specified stream. The `stream` parameter is usually set equal to `"stdout"` or `"stderr"`.

The second routine simply returns a 30-character descriptive error message corresponding to the input status value.

The last routine returns the current CFITSIO library version number.

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [4.3 HDU-level Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.1 Error Reporting](#) [Contents](#)

4.2 File Open/Close Routines

```
int fits_open_file( fitsfile **fptr, char *filename, int mode,
int *status)
```

```
int fits_open_data( fitsfile **fptr, char *filename, int mode,
int *status)
```

```
int fits_open_table(fitsfile **fptr, char *filename, int mode,
int *status)
```

```
int fits_open_image(fitsfile **fptr, char *filename, int
mode, int *status)
```

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

```
int fits_close_file(fitsfile *fptr, int *status)
```

These routines open or close a file. The first `fitsfile` parameter in these and nearly every other CFITSIO routine is a pointer to a structure that CFITSIO uses to store relevant parameters about each opened file. You should never directly read or write any information in this structure. Memory for this structure is allocated automatically when the file is opened or created, and is freed when the file is closed.

The `mode` parameter in the `fits_open_xxxx` set of routines can be set to either `READONLY` or `READWRITE` to select the type of file access that will be allowed.

(+)

These symbolic constants are defined in **fitsio.h**.

The **fits_open_file** routine opens the file and positions the internal file pointer to the beginning of the file, or to the specified extension if an extension name or number is appended to the file name (see the later section on ``CFITSIO File Names and Filters'' for a description of the syntax). **fits_open_data** behaves similarly except that it will move to the first HDU containing significant data if a HDU name or number to open is not explicitly specified as part of the filename. It will move to the first IMAGE HDU with NAXIS greater than 0, or the first table that does not contain the strings `GTI' (a Good Time Interval extension) or `OBSTABLE' in the EXTNAME keyword value. The **fits_open_table** and **fits_open_image** routines are similar except that they will move to the first significant table HDU or image HDU, respectively if a HDU name or number is not specified as part of the input file name.

When opening an existing file, the **filename** can include optional arguments, enclosed in square brackets that specify filtering operations that should be applied to the input file. For example,

```
myfile.fit[EVENTS][counts > 0]
```

opens the table in the EVENTS extension and creates a virtual table by selecting only those rows where the COUNTS column value is greater than 0. See section 5 for more examples of these powerful filtering capabilities.

In **fits_create_file**, the **filename** is simply the

(+)

root name of the file to be created. You can overwrite an existing file by prefixing the name with a `!` character (on the Unix command line this must be prefixed with a backslash, as in ``\!file.fit'`). If the file name ends with `.gz` the file will be compressed using the gzip algorithm. If the filename is `stdout` or `"-"` (a single dash character) then the output file will be piped to the stdout stream. You can chain several tasks together by writing the output from the first task to `stdout` and then reading the input file in the 2nd task from `stdin` or `"-"`.

Next	Up	Previous	Contents
------	----	----------	----------

[FITSIO Home](#)

Next: [4.3 HDU-level Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.1 Error Reporting](#) **[Contents](#)**

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [4.4 Image I/O Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.2 File Open/Close Routines](#) [Contents](#)

4.3 HDU-level Routines

The routines listed in this section operate on Header-Data Units (HDUs) in a file.

```
int fits_get_num_hdus(fitsfile *fptr, int *hdunum, int
*status)
int fits_get_hdu_num(fitsfile *fptr, int *hdunum)
```

The first routine returns the total number of HDUs in the FITS file, and the second routine returns the position of the currently opened HDU in the FITS file (starting with 1, not 0).

```
int fits_movabs_hdu(fitsfile *fptr, int hdunum, int
*hdutype, int *status)
int fits_movrel_hdu(fitsfile *fptr, int nmove, int *hdutype,
int *status)
int fits_movnam_hdu(fitsfile *fptr, int hdutype, char
*extname,
int extver, int *status)
```

These routines enable you to move to a different HDU in the file. Most of the CFITSIO functions which read

(+)

or write keywords or data operate only on the currently opened HDU in the file. The first routine moves to the specified absolute HDU number in the FITS file (the first HDU = 1), whereas the second routine moves a relative number of HDUs forward or backward from the currently open HDU. The **hdutype** parameter returns the type of the newly opened HDU, and will be equal to one of these symbolic constant values: **IMAGE_HDU**, **ASCII_TBL**, or **BINARY_TBL**. **hdutype** may be set to **NULL** if it is not needed. The third routine moves to the (first) HDU that matches the input extension type, name, and version number, as given by the **XTENSION**, **EXTNAME** (or **HDUNAME**) and **EXTVER** keywords. If the input value of **extver** = 0, then the version number will be ignored when looking for a matching HDU.

```
int fits_get_hdu_type(fitsfile *fptr, int *hdutype, int
*status)
```

Get the type of the current HDU in the FITS file:
IMAGE_HDU, ASCII_TBL, or BINARY_TBL.

```
int fits_copy_hdu(fitsfile *infptr, fitsfile *outfptr, int
morekeys,
int *status)
int fits_copy_file(fitsfile *infptr, fitsfile *outfptr, int
previous,
int current, int following, > int *status)
```

The first routine copies the current HDU from the FITS file associated with **infptr** and appends it to the end of the FITS file associated with **outfptr**. Space may be reserved for **morekeys** additional keywords in the

(+)

output header. The second routine copies any HDUs previous to the current HDU, and/or the current HDU, and/or any HDUs following the current HDU, depending on the value (True or False) of **previous**, **current**, and **following**, respectively. For example,

```
fits_copy_file(infptr, outfptr, 0, 1, 1, &status);
```

will copy the current HDU and any HDUs that follow it from the input to the output file, but it will not copy any HDUs preceding the current HDU.

Next	Up	Previous	Contents
------	----	----------	----------

[FITSIO Home](#)

Next: [4.4 Image I/O Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.2 File Open/Close Routines](#)
[Contents](#)

(+)

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)
[FITSIO Home](#)

Next: [4.5 Table I/O Routines](#)
Up: [4 CFITSIO Routines](#)
Previous: [4.3 HDU-level Routines](#)
[Contents](#)

4.4 Image I/O Routines

This section lists the more important CFITSIO routines which operate on FITS images.

```

int fits_get_img_type(fitsfile *fptr, int *bitpix, int *status)
int fits_get_img_dim( fitsfile *fptr, int *naxis, int *status)
int fits_get_img_size(fitsfile *fptr, int maxdim, long
*naxes,
                    int *status)
int fits_get_img_param(fitsfile *fptr, int maxdim, int
*bitpix,
                    int *naxis, long *naxes, int *status)

```

Get information about the currently opened image HDU. The first routine returns the datatype of the image as (defined by the **BITPIX** keyword), which can have the following symbolic constant values:

BYTE_IMG	= 8	(8-bit byte pixels, 0 - 255)	
SHORT_IMG	= 16	(16 bit integer pixels)	
LONG_IMG	= 32	(32-bit integer pixels)	
LONGLONG_IMG	= 64	(64-bit integer pixels)	
FLOAT_IMG	= -32	(32-bit floating point pixels)	
DOUBLE_IMG	= -64	(64-bit floating point pixels)	(+)

The second and third routines return the number of dimensions in the image (from the **NAXIS** keyword), and the sizes of each dimension (from the **NAXIS1**, **NAXIS2**, etc. keywords). The last routine simply combines the function of the first 3 routines. The input **maxdim** parameter in this routine gives the maximum number dimensions that may be returned (i.e., the dimension of the **naxes** array)

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

Create an image HDU by writing the required keywords which define the structure of the image. The 2nd through 4th parameters specified the datatype, the number of dimensions, and the sizes of the dimensions. The allowed values of the **bitpix** parameter are listed above in the description of the **fits_get_img_type** routine. If the FITS file pointed to by **fptr** is empty (previously created with **fits_create_file**) then this routine creates a primary array in the file, otherwise a new **IMAGE** extension is appended to end of the file following the other HDUs in the file.

```
int fits_write_pix(fitsfile *fptr, int datatype, long *fpixel,
                  long nelelements, void *array, int *status);
```

```
int fits_write_pixnull(fitsfile *fptr, int datatype, long
*fpixel,
                      long nelelements, void *array, void *nulval, int
*status);
```

```
int fits_read_pix(fitsfile *fptr, int datatype, long *fpixel,
```

(+)

```
long nelements, void *nulval, void *array,  
int *anynul, int *status)
```

Read or write all or part of the FITS image. There are 2 different 'write' pixel routines: The first simply writes the input array of pixels to the FITS file. The second is similar, except that it substitutes the appropriate null pixel value in the FITS file for any pixels which have a value equal to `*nulval` (note that this parameter gives the address of the null pixel value, not the value itself). Similarly, when reading an image, CFITSIO will substitute the value given by `nulval` for any undefined pixels in the image, unless `nulval = NULL`, in which case no checks will be made for undefined pixels when reading the FITS image.

The `fpixel` parameter in these routines is an array which gives the coordinate in each dimension of the first pixel to be read or written, and `nelements` is the total number of pixels to read or write. `array` is the address of an array which either contains the pixel values to be written, or will hold the values of the pixels that are read. When reading, `array` must have been allocated large enough to hold all the returned pixel values. These routines starts at the `fpixel` location and then read or write the `nelements` pixels, continuing on successive rows of the image if necessary. For example, to write an entire 2D image, set `fpixel[0] = fpixel[1] = 1`, and `nelements = NAXIS1 * NAXIS2`. Or to read just the 10th row of the image, set `fpixel[0] = 1, fpixel[1] = 10`, and `nelements = NAXIS1`. The `datatype` parameter specifies the datatype of the C array in the program, which need not be the same as the datatype of the FITS image itself. If the datatypes differ then CFITSIO will

(+)

convert the data as it is read or written. The following symbolic constants are allowed for the value of datatype:

TBYTE unsigned char
 TSBYTE signed char
 TSHORT signed short
 TUSHORT unsigned short
 TINT signed int
 TUINT unsigned int
 TLONG signed long
 TLONGLONG signed 8-byte integer
 TULONG unsigned long
 TFLOAT float
 TDOUBLE double

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

```
int fits_read_subset(fitsfile *fptr, int datatype, long *fpixel,
    long *lpixel, long *inc, void *nulval, void *array,
    int *any nul, int *status)
```

Read or write a rectangular section of the FITS image. These are very similar to `fits_write_pix` and `fits_read_pix` except that you specify the last pixel coordinate (the upper right corner of the section) instead of the number of pixels to be read. The read routine also has an `inc` parameter which can be used to read only every `inc`-th pixel along each dimension of the image. Normally `inc[0] = inc[1] = 1` to read every pixel in a 2D image. To read every other pixel in the entire 2D image, set

(+)

```
fpixel[0] = fpixel[1] = 1  
lpxel[0] = {NAXIS1}  
lpxel[1] = {NAXIS2}  
inc[0] = inc[1] = 2
```

Or, to read the 8th row of a 2D image, set

```
fpxel[0] = 1  
fpxel[1] = 8  
lpxel[0] = {NAXIS1}  
lpxel[1] = 8  
inc[0] = inc[1] = 1
```

Next	Up	Previous	Contents
------	----	----------	----------

[FITSIO Home](#)

Next: [4.5 Table I/O Routines](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.3 HDU-level Routines](#)
[Contents](#)

(+)

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)
[FITSIO Home](#)

Next: [4.6 Header Keyword I/O](#)
Up: [4 CFITSIO Routines](#)
Previous: [4.4 Image I/O Routines](#)
[Contents](#)

4.5 Table I/O Routines

This section lists the most important CFITSIO routines which operate on FITS tables.

```
int fits_create_tbl(fitsfile *fptr, int tbltype, long nrows, int
tfields,
    char *ttype[],char *tform[], char *tunit[], char
*extname, int *status)
```

Create a new table extension by writing the required keywords that define the table structure. The required null primary array will be created first if the file is initially completely empty. **tbltype** defines the type of table and can have values of **ASCII_TBL** or **BINARY_TBL**. Binary tables are generally preferred because they are more efficient and support a greater range of column datatypes than ASCII tables.

The **nrows** parameter gives the initial number of empty rows to be allocated for the table; this should normally be set to 0. The **tfields** parameter gives the number of columns in the table (maximum = 999). The **ttype**, **tform**, and **tunit** parameters give the name, datatype, and physical units of each column, and

(+)

extname gives the name for the table (the value of the **EXTNAME** keyword). The FITS Standard recommends that only letters, digits, and the underscore character be used in column names with no embedded spaces. It is recommended that all the column names in a given table be unique within the first 8 characters.

The following table shows the TFORM column format values that are allowed in ASCII tables and in binary tables:

ASCII Table Column Format Codes

(w = column width, d = no. of decimal places to display)

Aw - character string
 Iw - integer
 Fw.d - fixed floating point
 Ew.d - exponential floating point
 Dw.d - exponential floating point

Binary Table Column Format Codes

(r = vector length, default = 1)

rA - character string
 rAw - array of strings, each of length w
 rL - logical
 rX - bit
 rB - unsigned byte
 rS - signed byte **
 rI - signed 16-bit integer
 rU - unsigned 16-bit integer **
 rJ - signed 32-bit integer
 rV - unsigned 32-bit integer **
 rK - signed 64-bit integer

(+)

rE - 32-bit floating point
 rD - 64-bit floating point
 rC - 32-bit complex pair
 rM - 64-bit complex pair

** The S, U and V format codes are not actual legal TFORMn values.

CFITSIO substitutes the somewhat more complicated set of keywords that are used to represent unsigned integers or signed bytes.

The **tunit** and **extname** parameters are optional and may be set to NULL if they are not needed.

Note that it may be easier to create a new table by copying the header from another existing table with **fits_copy_header** rather than calling this routine.

```

int fits_get_num_rows(fitsfile *fptr, long *nrows, int
*status)
int fits_get_num_cols(fitsfile *fptr, int *ncols, int *status)

```

Get the number of rows or columns in the current FITS table. The number of rows is given by the **NAXIS2** keyword and the number of columns is given by the **TFIELDS** keyword in the header of the table.

```

int fits_get_colnum(fitsfile *fptr, int casesen, char
*template,
                int *colnum, int *status)
int fits_get_colname(fitsfile *fptr, int casesen, char
*template,

```

(+)

```
char *colname, int *colnum, int *status)
```

Get the column number (starting with 1, not 0) of the column whose name matches the specified template name. The only difference in these 2 routines is that the 2nd one also returns the name of the column that matched the template string.

Normally, **casesen** should be set to **CASEINSEN**, but it may be set to **CASESEN** to force the name matching to be case-sensitive.

The input **template** string gives the name of the desired column and may include wildcard characters: a **`*'** matches any sequence of characters (including zero characters), **`?'** matches any single character, and **`#'** matches any consecutive string of decimal digits (0-9). If more than one column name in the table matches the template string, then the first match is returned and the status value will be set to **COL_NOT_UNIQUE** as a warning that a unique match was not found. To find the next column that matches the template, call this routine again leaving the input status value equal to **COL_NOT_UNIQUE**. Repeat this process until **status = COL_NOT_FOUND** is returned.

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

```
int fits_get_eqcoltype(fitsfile *fptr, int colnum, int
                      *typecode,
                      long *repeat, long *width, int *status)
```

Return the datatype, vector repeat count, and the width in bytes of a single column element for column number

(+)

colnum. Allowed values for the returned datatype in ASCII tables are: **TSTRING**, **TSHORT**, **TLONG**, **TFLOAT**, and **TDOUBLE**. Binary tables support these additional types: **TLOGICAL**, **TBIT**, **TBYTE**, **TINT32BIT**, **TCOMPLEX** and **TDBLCOMPLEX**. The negative of the datatype code value is returned if it is a variable length array column.

These 2 routines are similar, except that in the case of scaled integer columns the 2nd routine, **fit_get_eqcoltype**, returns the 'equivalent' datatype that is needed to store the scaled values, which is not necessarily the same as the physical datatype of the unscaled values as stored in the FITS table. For example if a '1I' column in a binary table has **TSCALn** = 1 and **TZEROn** = 32768, then this column effectively contains unsigned short integer values, and thus the returned value of typecode will be **TUSHORT**, not **TSHORT**. Or, if **TSCALn** or **TZEROn** are not integers, then the equivalent datatype will be returned as **TFLOAT** or **TDOUBLE**, depending on the size of the integer.

The repeat count is always 1 in ASCII tables. The 'repeat' parameter returns the vector repeat count on the binary table **TFORMn** keyword value. (ASCII table columns always have repeat = 1). The 'width' parameter returns the width in bytes of a single column element (e.g., a '10D' binary table column will have width = 8, an ASCII table 'F12.2' column will have width = 12, and a binary table '60A' character string column will have width = 60); Note that this routine supports the local convention for specifying arrays of fixed length strings within a binary table character column using the syntax **TFORM** = 'rAw' where 'r' is

(+)

the total number of characters (= the width of the column) and 'w' is the width of a unit string within the column. Thus if the column has TFORM = '60A12' then this means that each row of the table contains 5 12-character substrings within the 60-character field, and thus in this case this routine will return typecode = TSTRING, repeat = 60, and width = 12. The number of substings in any binary table character string field can be calculated by (repeat/width). A null pointer may be given for any of the output parameters that are not needed.

```
int fits_insert_rows(fitsfile *fptr, long firstrow, long nrows,
int *status)
int fits_delete_rows(fitsfile *fptr, long firstrow, long
nrows, int *status)
int fits_delete_rowrange(fitsfile *fptr, char *rangelist, int
*status)
int fits_delete_rowlist(fitsfile *fptr, long *rowlist, long
nrows, int *stat)
```

Insert or delete rows in a table. The blank rows are inserted immediately following row **frow**. Set **frow = 0** to insert rows at the beginning of the table. The first 'delete' routine deletes **nrows** rows beginning with row **firstrow**. The 2nd delete routine takes an input string listing the rows or row ranges to be deleted (e.g., '2,4-7, 9-12'). The last delete routine takes an input long integer array that specifies each individual row to be deleted. The row lists must be sorted in ascending order. All these routines update the value of the **NAXIS2** keyword to reflect the new number of rows in the table.

(+)

```

int fits_insert_col(fitsfile *fptr, int colnum, char *ttype,
char *tform,
                    int *status)
int fits_insert_cols(fitsfile *fptr, int colnum, int ncols, char
**ttype,
                    char **tform, int *status)

int fits_delete_col(fitsfile *fptr, int colnum, int *status)

```

Insert or delete columns in a table. **colnum** gives the position of the column to be inserted or deleted (where the first column of the table is at position 1). **ttype** and **tform** give the column name and column format, where the allowed format codes are listed above in the description of the **fits_create_table** routine. The 2nd 'insert' routine inserts multiple columns, where **ncols** is the number of columns to insert, and **ttype** and **tform** are arrays of string pointers in this case.

```

int fits_copy_col(fitsfile *infptr, fitsfile *outfptr, int
incolnum,
                int outcolnum, int create_col, int *status);

```

Copy a column from one table HDU to another. If **create_col = TRUE** (i.e., not equal to zero), then a new column will be inserted in the output table at position **outcolumn**, otherwise the values in the existing output column will be overwritten.

```

int fits_write_col(fitsfile *fptr, int datatype, int colnum,
long firstrow,
                  long firstelem, long nelements, void *array, int
*status)

```

(+)

```

int fits_write_colnull(fitsfile *fptr, int datatype, int colnum,
                      long firstrow, long firstelem, long nelements,
                      void *array, void *nulval, int *status)
int fits_write_col_null(fitsfile *fptr, int colnum, long
firstrow,
                      long firstelem, long nelements, int *status)

int fits_read_col(fitsfile *fptr, int datatype, int colnum,
long firstrow,
                long firstelem, long nelements, void *nulval, void
*array,
                int *anynul, int *status)

```

Write or read elements in column number **colnum**, starting with row **firstsrow** and element **firstelem** (if it is a vector column). **firstelem** is ignored if it is a scalar column. The **nelements** number of elements are read or written continuing on successive rows of the table if necessary. **array** is the address of an array which either contains the values to be written, or will hold the returned values that are read. When reading, **array** must have been allocated large enough to hold all the returned values.

There are 3 different 'write' column routines: The first simply writes the input array into the column. The second is similar, except that it substitutes the appropriate null pixel value in the column for any input array values which are equal to ***nulval** (note that this parameter gives the address of the null pixel value, not the value itself). The third write routine sets the specified table elements to a null value. New rows will be automatical added to the table if the write operation extends beyond the current size of the table.

(+)

When reading a column, CFITSIO will substitute the value given by `nulval` for any undefined elements in the FITS column, unless `nulval` or `*nulval` = `NULL`, in which case no checks will be made for undefined values when reading the column.

datatype specifies the datatype of the C array in the program, which need not be the same as the intrinsic datatype of the column in the FITS table. The following symbolic constants are allowed for the value of **datatype**:

TSTRING array of character string pointers
 TBYTE unsigned char
 TSHORT signed short
 TUSHORT unsigned short
 TINT signed int
 TUINT unsigned int
 TLONG signed long
 TLONGLONG signed 8-byte integer
 TULONG unsigned long
 TFLOAT float
 TDOUBLE double

Note that **TSTRING** corresponds to the C `char**` datatype, i.e., a pointer to an array of pointers to an array of characters.

Any column, regardless of its intrinsic datatype, may be read as a **TSTRING** character string. The display format of the returned strings will be determined by the **TDISPn** keyword, if it exists, otherwise a default format will be used depending on the datatype of the column. The **tablist** example utility program (available from the CFITSIO web site) uses this feature to display all the values in a FITS table.

(+)

```

int fits_select_rows(fitsfile *infptr, fitsfile *outfptr, char
*expr,
                    int *status)
int fits_calculator(fitsfile *infptr, char *expr, fitsfile
*outfptr,
                    char *colname, char *tform, int *status)

```

These are 2 of the most powerful routines in the CFITSIO library. (See the full CFITSIO Reference Guide for a description of several related routines). These routines can perform complicated transformations on tables based on an input arithmetic expression which is evaluated for each row of the table. The first routine will select or copy rows of the table for which the expression evaluates to TRUE (i.e., not equal to zero). The second routine writes the value of the expression to a column in the output table. Rather than supplying the expression directly to these routines, the expression may also be written to a text file (continued over multiple lines if necessary) and the name of the file, prepended with a '@' character, may be supplied as the value of the 'expr' parameter (e.g. '@filename.txt').

The arithmetic expression may be a function of any column or keyword in the input table as shown in these examples:

Row Selection Expressions:

counts > 0	uses COUNTS column value
sqrt(X**2 + Y**2) < 10.	uses X and Y column values
(X > 10) (X < -10) && (Y == 0)	used 'or' and 'and' operators
gtifilter()	filter on Good Time Intervals

(+)

`regfilter("myregion.reg")` filter using a region file
`@select.txt` reads expression from a text file

Calculator Expressions:

`#row % 10` modulus of the row number
`counts/#exposure` Fn of COUNTS column and EXPOSURE keyword
`dec < 85 ? cos(dec * #deg) : 0` Conditional expression: evaluates to

`cos(dec)` if `dec < 85`, else 0

`(count{-1}+count+count{+1})/3`. running mean of the count values in the

previous, current, and next rows

`max(0, min(X, 1000))` returns a value between 0 - 1000

`@calc.txt` reads expression from a text file

Most standard mathematical operators and functions are supported. If the expression includes the name of a column, then the value in the current row of the table will be used when evaluating the expression on each row. An offset to an adjacent row can be specified by including the offset value in curly brackets after the column name as shown in one of the examples. Keyword values can be included in the expression by preceding the keyword name with a '#' sign. See Section 5 of this document for more discussion of the expression syntax.

`gtifilter` is a special function which tests whether the **TIME column value in the input table falls within one or more Good Time Intervals. By default, this function looks for a 'GTI' extension in the same file as the input table. The 'GTI' table contains **START** and **STOP** columns which define the range of each good time**

(+)

interval. See section 5.4.3 for more details.

regfilter is another special function which selects rows based on whether the spatial position associated with each row is located within in a specified region of the sky. By default, the **X** and **Y** columns in the input table are assumed to give the position of each row. The spatial region is defined in an ASCII text file whose name is given as the argument to the **regfilter** function. See section 5.4.4 for more details.

The **infptr** and **outfptr** parameters in these routines may point to the same table or to different tables. In **fits_select_rows**, if the input and output tables are the same then the rows that do not satisfy the selection expression will be deleted from the table. Otherwise, if the output table is different from the input table then the selected rows will be copied from the input table to the output table.

The output column in **fits_calculator** may or may not already exist. If it exists then the calculated values will be written to that column, overwriting the existing values. If the column doesn't exist then the new column will be appended to the output table. The **tform** parameter can be used to specify the datatype of the new column (e.g., the **TFORM** keyword value as in **'1E'**, or **'1J'**). If **tform** = **NULL** then a default datatype will be used, depending on the expression.

```
int fits_read_tblbytes(fitsfile *fptr, long firstrow, long
firstchar,
                    long nchars, unsigned char *array, int *status)
int fits_write_tblbytes (fitsfile *fptr, long firstrow, long
firstchar,
```

(+)

long nchars, unsigned char *array, int *status)

These 2 routines provide low-level access to tables and are mainly useful as an efficient way to copy rows of a table from one file to another. These routines simply read or write the specified number of consecutive characters (bytes) in a table, without regard for column boundaries. For example, to read or write the first row of a table, set **firstrow** = 1, **firstchar** = 1, and **nchars** = **NAXIS1** where the length of a row is given by the value of the **NAXIS1** header keyword. When reading a table, **array** must have been declared at least **nchars** bytes long to hold the returned string of bytes.

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [4.6 Header Keyword I/O](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.4 Image I/O Routines](#) [Contents](#)

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [4.7 Utility Routines](#) **Up:** [4 CFITSIO Routines](#)

Previous: [4.5 Table I/O Routines](#) [Contents](#)

4.6 Header Keyword I/O Routines

The following routines read and write header keywords in the current HDU.

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

Return the number of existing keywords (not counting the mandatory END keyword) and the amount of empty space currently available for more keywords. The `morekeys` parameter may be set to NULL if its value is not needed.

```
int fits_read_record(fitsfile *fptr, int keynum, char *record,
int *status)
int fits_read_card(fitsfile *fptr, char *keyname, char
*record, int *status)
int fits_read_key(fitsfile *fptr, int datatype, char *keyname,
void *value, char *comment, int *status)

int fits_find_nextkey(fitsfile *fptr, char **inclist, int ninc,
char **excllist, int nexc, char *card, int
```

(+)

*status)

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

These routines all read a header record in the current HDU. The first routine reads keyword number **keynum** (where the first keyword is at position 1). This routine is most commonly used when sequentially reading every record in the header from beginning to end. The 2nd and 3rd routines read the named keyword and return either the whole record, or the keyword value and comment string. In each case any non-significant trailing blank characters in the strings are truncated.

Wild card characters (*, ?, and #) may be used when specifying the name of the keyword to be read, in which case the first matching keyword is returned.

The **datatype** parameter specifies the C datatype of the returned keyword value and can have one of the following symbolic constant values: **TSTRING**, **TLOGICAL** (== int), **TBYTE**, **TSHORT**, **TUSHORT**, **TINT**, **TUINT**, **TLONG**, **TULONG**, **TFLOAT**, **TDOUBLE**, **TCOMPLEX**, and **TDBLCOMPLEX**. Data type conversion will be performed for numeric values if the intrinsic FITS keyword value does not have the same datatype. The **comment** parameter may be set equal to NULL if the comment string is not needed.

The 4th routine provides an easy way to find all the keywords in the header that match one of the name templates in **inclist** and do not match any of the name templates in **exclist**. **ninc** and **nexc** are the number of template strings in **inclist** and **exclist**,

(+)

respectively. Wild cards (*, ?, and #) may be used in the templates to match multiple keywords. Each time this routine is called it returns the next matching 80-byte keyword record. It returns status = **KEY_NO_EXIST** if there are no more matches.

The 5th routine returns the keyword value units string, if any. The units are recorded at the beginning of the keyword comment field enclosed in square brackets.

```
int fits_write_key(fitsfile *fptr, int datatype, char
*keyname,
    void *value, char *comment, int *status)
int fits_update_key(fitsfile *fptr, int datatype, char
*keyname,
    void *value, char *comment, int *status)
int fits_write_record(fitsfile *fptr, char *card, int *status)

int fits_modify_comment(fitsfile *fptr, char *keyname,
char *comment,
    int *status)
int fits_write_key_unit(fitsfile *fptr, char *keyname, char
*unit,
    int *status)
```

Write or modify a keyword in the header of the current HDU. The first routine appends the new keyword to the end of the header, whereas the second routine will update the value and comment fields of the keyword if it already exists, otherwise it behaves like the first routine and appends the new keyword. Note that **value gives the address to the value and not the value itself. The **datatype** parameter specifies the C datatype of the keyword value and may have any of the**

(+)

values listed in the description of the keyword reading routines, above. A **NULL** may be entered for the comment parameter, in which case the keyword comment field will be unmodified or left blank.

The third routine is more primitive and simply writes the 80-character **card** record to the header. It is the programmer's responsibility in this case to ensure that the record conforms to all the FITS format requirements for a header record.

The fourth routine modifies the comment string in an existing keyword, and the last routine writes or updates the keyword units string for an existing keyword. (The units are recorded at the beginning of the keyword comment field enclosed in square brackets).

```
int fits_write_comment(fitsfile *fptr, char *comment, int
*status)
int fits_write_history(fitsfile *fptr, char *history, int
*status)
int fits_write_date(fitsfile *fptr, int *status)
```

Write a **COMMENT**, **HISTORY**, or **DATE** keyword to the current header. The **COMMENT** keyword is typically used to write a comment about the file or the data. The **HISTORY** keyword is typically used to provide information about the history of the processing procedures that have been applied to the data. The **comment** or **history** string will be continued over multiple keywords if it is more than 70 characters long.

The **DATE** keyword is used to record the date and time that the FITS file was created. Note that this file creation date is usually different from the date of the

(+)

observation which obtained the data in the FITS file. The **DATE** keyword value is a character string in 'yyyy-mm-ddThh:mm:ss' format. If a **DATE** keyword already exists in the header, then this routine will update the value with the current system date.

```
int fits_delete_record(fitsfile *fptr, int keynum, int *status)
int fits_delete_key(fitsfile *fptr, char *keyname, int
*status)
```

Delete a keyword record. The first routine deletes a keyword at a specified position (the first keyword is at position 1, not 0), whereas the second routine deletes the named keyword.

```
int fits_copy_header(fitsfile *infptr, fitsfile *outfptr, int
*status)
```

Copy all the header keywords from the current HDU associated with infptr to the current HDU associated with outfptr. If the current output HDU is not empty, then a new HDU will be appended to the output file. The output HDU will then have the identical structure as the input HDU, but will contain no data.

Next	Up	Previous	Contents
----------------------	--------------------	--------------------------	--------------------------

[FITSIO Home](#)

Next: [4.7 Utility Routines](#) **Up:** [4 CFITSIO Routines](#)
Previous: [4.5 Table I/O Routines](#) **[Contents](#)**

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5 CFITSIO File Names](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.6 Header Keyword I/O Contents](#)

4.7 Utility Routines

This section lists the most important CFITSIO general utility routines.

```
int fits_write_chksum( fitsfile *fptr, int *status)
int fits_verify_chksum(fitsfile *fptr, int *dataok, int
*hduok, int *status)
```

These routines compute or validate the checksums for the current HDU. The **DATASUM** keyword is used to store the numerical value of the 32-bit, 1's complement checksum for the data unit alone. 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.

The returned **dataok** and **hduok** parameters will have a value = 1 if the data or HDU is verified correctly, a value = 0 if the **DATASUM** or **CHECKSUM** keyword is not present, or value = -1 if the computed checksum is not

(+)

correct.

```
int fits_parse_value(char *card, char *value, char
*comment, int *status)
int fits_get_keytype(char *value, char *dtype, int *status)
int fits_get_keyclass(char *card)
int fits_parse_template(char *template, char *card, int
*keytype, int *status)
```

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

fits_get_keytype parses the keyword value string to determine its datatype. **dtype** returns with a value of 'C', 'L', 'I', 'F' or 'X', for character string, logical, integer, floating point, or complex, respectively.

fits_get_keyclass returns a classification code that indicates the classification type of the input keyword record (e.g., a required structural keyword, a TDIM keyword, a WCS keyword, a comment keyword, etc. See the CFITSIO Reference Guide for a list of the different classification codes.

fits_parse_template takes an input free format keyword template string and returns a formatted 80*char record that satisfies all the FITS requirements for a header keyword record. The template should generally contain 3 tokens: the keyword name, the keyword value, and the keyword comment string. The returned **keytype** parameter indicates whether the

(+)

keyword is a **COMMENT** keyword or not. See the **CFITSIO Reference Guide** for more details.

[Next](#)[Up](#)[Previous](#)[Contents](#)[FITSIO Home](#)

Next: [5 CFITSIO File Names](#) **Up:** [4 CFITSIO Routines](#) **Previous:** [4.6 Header Keyword I/O](#)
[Contents](#)

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.2 Row Filtering](#) **Up:** [5.4 Table Filtering](#)

Previous: [5.4 Table Filtering](#) [Contents](#)

5.4.1 Column and Keyword Filtering

The column or keyword filtering specifier is used to modify the column structure and/or the header keywords in the HDU that was selected with the previous HDU location specifier. It can be used to perform the following types of operations.

- Append a new column to a table by giving the column name, optionally followed by the datatype in parentheses, followed by an equals sign and the arithmetic expression to be used to compute the value. The datatype is specified using the same syntax that is allowed for the value of the FITS TFORMn keyword (e.g., 'I', 'J', 'E', 'D', etc. for binary tables, and 'I8', 'F12.3', 'E20.12', etc. for ASCII tables). If the datatype is not specified then a default datatype will be chosen depending on the expression.
- Create a new header keyword by giving the keyword name, preceded by a pound sign '#', followed by an equals sign and an arithmetic expression for the value of the keyword. The expression may be a function of other header keyword values. The comment string for the keyword may be specified in parentheses immediately following the keyword name.
- Overwrite the values in an existing column or

(+)

keyword by giving the name followed by an equals sign and an arithmetic expression.

- Select a set of columns to be included in the filtered file by listing the column names separated with semi-colons. Wild card characters may be used in the column names to match multiple columns. Any other columns in the input table will not appear in the filtered file.
- Delete a column or keyword by listing the name preceded by a minus sign or an exclamation mark (!)
- Rename an existing column or keyword with the syntax 'NewName == OldName'.

The column filtering specifier is enclosed in square brackets and begins with the string 'col'. Multiple operations can be performed by separating them with semi-colons. For complex or commonly used operations, you can write the column filter to a text file, and then use it by giving the name of the text file, preceded by a '@' character.

Some examples:

[col PI=PHA * 1.1 + 0.2] - creates new PI column
from PHA values

[col rate = counts/exposure] - creates or overwrites the
rate column by

dividing the counts column by the
EXPOSURE keyword value.

[col TIME; X; Y] - only the listed columns will
appear
in the filtered file

(+)

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

`[col -TIME; Good == STATUS]` - deletes the TIME column and renames the STATUS column to GOOD

`[col @colfilt.txt]` - uses the filtering expression in the colfilt.txt text file

The original file is not changed by this filtering operation, and instead the modifications are made on a temporary copy of the input FITS file (usually in memory), which includes a copy of all the other HDUs in the input file. The original input file is closed and the application program opens the filtered copy of the file.

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.2 Row Filtering](#) **Up:** [5.4 Table Filtering](#)
Previous: [5.4 Table Filtering](#) **[Contents](#)**

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.3 Good Time Interval](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.1 Column and Keyword Contents](#)

5.4.2 Row Filtering

The row filter is used to select a subset of the rows from a table based on a boolean expression. A temporary new FITS file is created on the fly (usually in memory) which contains only those rows for which the row filter expression evaluates to true (i.e., not equal to zero). The primary array and any other extensions in the input file are also copied to the temporary file. The original FITS file is closed and the new temporary file is then opened by the application program.

The row filter expression is enclosed in square brackets following the file name and extension name. For example, `'file.fits[events][GRADE==50]'` selects only those rows in the EVENTS table where the GRADE column value is equal to 50).

The row filtering expression can be an arbitrarily complex series of operations performed on constants, keyword values, and column data taken from the specified FITS TABLE extension. The expression also can be written into a text file and then used by giving the filename preceded by a '@' character, as in `'[@rowfilt.txt]'`.

(+)

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

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

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

"equal"	.eq. .EQ. ==	"not equal"	.ne. .NE. !=
"less than"	.lt. .LT. <	"less than/equal"	.le. .LE.
<= =<			
"greater than"	.gt. .GT. >	"greater than/equal"	.ge.
.GE. >= =>			

(+)

"or" .or. .OR. || "and" .and. .AND. &&
 "negation" .not. .NOT. ! "approx. equal(1e-7)" ~

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

The expression may also include arithmetic operators and functions. Trigonometric functions use radians, not degrees. The following arithmetic operators and functions can be used in the expression (function names are case insensitive):

"addition"	+	"subtraction"	-
"multiplication"	*	"division"	/
"negation"	-	"exponentiation"	** ^
"absolute value"	abs(x)	"cosine"	cos(x)
"sine"	sin(x)	"tangent"	tan(x)
"arc cosine"	arccos(x)	"arc sine"	arcsin(x)
"arc tangent"	arctan(x)	"arc tangent"	
arctan2(x,y)			
"exponential"	exp(x)	"square root"	sqrt(x)
"natural log"	log(x)	"common log"	log10(x)
"modulus"	i % j	"random # [0.0,1.0)"	
random()			
"minimum"	min(x,y)	"maximum"	
max(x,y)			
"if-then-else"	b?x:y		

The following type casting operators are available, where the inclosing parentheses are required and taken from the C language usage. Also, the integer to real casts values to double precision:

"real to integer"	(int) x	(INT) x	(+)
-------------------	---------	---------	-----

"integer to real" (float) i (FLOAT) i

Several constants are built in for use in numerical expressions:

#pi	3.1415...	#e	2.7182...
#deg	#pi/180	#row	current row
number			
#null	undefined value	#snull	undefined
string			

A string constant must be enclosed in quotes as in 'Crab'. The "null" constants are useful for conditionally setting table values to a NULL, or undefined, value (For example, "col1== -99 ? #NULL : col1").

There is also a function for testing if two values are close to each other, i.e., if they are "near" each other to within a user specified tolerance. The arguments, **value_1** and **value_2** can be integer or real and represent the two values whose proximity is being tested to be within the specified tolerance, also an integer or real:

near(value_1, value_2, tolerance)

When a NULL, or undefined, value is encountered in the FITS table, the expression will evaluate to NULL unless the undefined value is not actually required for evaluation, e.g. "TRUE .or. NULL" evaluates to TRUE. The following two functions allow some NULL detection and handling:

ISNULL(x)

DEFNULL(x,y)

(+)

The former returns a boolean value of TRUE if the argument `x` is NULL. The later "defines" a value to be substituted for NULL values; it returns the value of `x` if `x` is not NULL, otherwise it returns the value of `y`.

Bit masks can be used to select out rows from bit columns (**TFORMn = #x**) in FITS files. To represent the mask, binary, octal, and hex formats are allowed:

```
binary: b0110xx1010000101xxxx0001
octal:  o720x1 -> (b111010000xxx001)
hex:    h0FxD -> (b00001111xxxx1101)
```

In all the representations, an `x` or `X` is allowed in the mask as a wild card. Note that the `x` represents a different number of wild card bits in each representation. All representations are case insensitive.

To construct the boolean expression using the mask as the boolean equal operator described above on a bit table column. For example, if you had a 7 bit column named `flags` in a FITS table and wanted all rows having the bit pattern `0010011`, the selection expression would be:

```
flags == b0010011
or
flags .eq. b10011
```


It is also possible to test if a range of bits is less than, less than equal, greater than and greater than equal to a particular boolean value:

```
flags <= bxxx010xx
flags .gt. bxxx100xx
flags .le. b1xxxxxxx
```

(+)

Notice the use of the x bit value to limit the range of bits being compared.

It is not necessary to specify the leading (most significant) zero (0) bits in the mask, as shown in the second expression above.

Bit wise AND, OR and NOT operations are also possible on two or more bit fields using the '&'(AND), '

```
(!flags) == b1101100  
(flags & b1000001) == bx000001
```

Bit fields can be appended as well using the '+' operator. Strings can be concatenated this way, too.

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.3 Good Time Interval](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.1 Column and Keyword Contents](#)

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.4 Spatial Region Filtering](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.2 Row Filtering](#) [Contents](#)

5.4.3 Good Time Interval Filtering

A common filtering method involves selecting rows which have a time value which lies within what is called a Good Time Interval or GTI. The time intervals are defined in a separate FITS table extension which contains 2 columns giving the start and stop time of each good interval. The filtering operation accepts only those rows of the input table which have an associated time which falls within one of the time intervals defined in the GTI extension. A high level function, `gtifilter(a,b,c,d)`, is available which evaluates each row of the input table and returns TRUE or FALSE depending whether the row is inside or outside the good time interval. The syntax is

```
gtifilter( [ "gtifile" [, expr [, "STARTCOL",  
"STOPCOL" ] ] ] )
```

where each "[]" demarks optional parameters. Note that the quotes around the `gtifile` and `START/STOP` column are required. Either single or double quote characters may be used. The `gtifile`, if specified, can be blank ("") which will mean to use the first extension with the name `"*GTI*"` in the current file, a

(+)

plain extension specifier (eg, "+2", "[2]", or "[STDGTI]") which will be used to select an extension in the current file, or a regular filename with or without an extension specifier which in the latter case will mean to use the first extension with an extension name "*GTI*". Expr can be any arithmetic expression, including simply the time column name. A vector time expression will produce a vector boolean result. STARTCOL and STOPCOL are the names of the START/STOP columns in the GTI extension. If one of them is specified, they both must be.

In its simplest form, no parameters need to be provided - default values will be used. The expression `"gtifilter()"` is equivalent to

```
gtifilter( "", TIME, "*START*", "*STOP*" )
```

This will search the current file for a GTI extension, filter the TIME column in the current table, using START/STOP times taken from columns in the GTI extension with names containing the strings "START" and "STOP". The wildcards (*) allow slight variations in naming conventions such as "TSTART" or "STARTTIME". The same default values apply for unspecified parameters when the first one or two parameters are specified. The function automatically searches for TIMEZERO/I/F keywords in the current and GTI extensions, applying a relative time offset, if necessary.

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.4 Spatial Region Filtering](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.2 Row Filtering](#) **[Contents](#)**

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [5.4.5 Example Row Filters](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.3 Good Time Interval Contents](#)

5.4.4 Spatial Region Filtering

Another common filtering method selects rows based on whether the spatial position associated with each row is located within a given 2-dimensional region. The syntax for this high-level filter is

```
regfilter( "regfilename" [ , Xexpr, Yexpr [ , "wcs cols"
]])
```

where each "[]" demarks optional parameters. The region file name is required and must be enclosed in quotes. The remaining parameters are optional. The region file is an ASCII text file which contains a list of one or more geometric shapes (circle, ellipse, box, etc.) which defines a region on the celestial sphere or an area within a particular 2D image. The region file is typically generated using an image display program such as fv/POW (distributed by the HEASARC), or ds9 (distributed by the Smithsonian Astrophysical Observatory). Users should refer to the documentation provided with these programs for more

(+)

details on the syntax used in the region files.

In its simplest form, (e.g., `regfilter("region.reg")`) the coordinates in the default 'X' and 'Y' columns will be used to determine if each row is inside or outside the area specified in the region file. Alternate position column names, or expressions, may be entered if needed, as in

```
regfilter("region.reg", XPOS, YPOS)
```

Region filtering can be applied most unambiguously if the positions in the region file and in the table to be filtered are both give in terms of absolute celestial coordinate units. In this case the locations and sizes of the geometric shapes in the region file are specified in angular units on the sky (e.g., positions given in R.A. and Dec. and sizes in arcseconds or arcminutes). Similarly, each row of the filtered table will have a celestial coordinate associated with it. This association is usually implemented using a set of so-called 'World Coordinate System' (or WCS) FITS keywords that define the coordinate transformation that must be applied to the values in the 'X' and 'Y' columns to calculate the coordinate.

Alternatively, one can perform spatial filtering using unitless 'pixel' coordinates for the regions and row positions. In this case the user must be careful to ensure that the positions in the 2 files are self-consistent. A typical problem is that the region file may be generated using a binned image, but the unbinned coordinates are given in the event table. The ROSAT events files, for

(+)

example, have X and Y pixel coordinates that range from 1 - 15360. These coordinates are typically binned by a factor of 32 to produce a 480x480 pixel image. If one then uses a region file generated from this image (in image pixel units) to filter the ROSAT events file, then the X and Y column values must be converted to corresponding pixel units as in:

```
regfilter("rosat.reg", X/32.+5, Y/32.+5)
```

Note that this binning conversion is not necessary if the region file is specified using celestial coordinate units instead of pixel units because CFITSIO is then able to directly compare the celestial coordinate of each row in the table with the celestial coordinates in the region file without having to know anything about how the image may have been binned.

The last "wcs cols" parameter should rarely be needed. If supplied, this string contains the names of the 2 columns (space or comma separated) which have the associated WCS keywords. If not supplied, the filter will scan the X and Y expressions for column names. If only one is found in each expression, those columns will be used, otherwise an error will be returned.

These region shapes are supported (names are case insensitive):

Point (X1, Y1) <- One pixel square region

Line (X1, Y1, X2, Y2) <- One pixel wide region

Polygon (X1, Y1, X2, Y2, ...) <- Rest are (+)

interiors with

Rectangle (X1, Y1, X2, Y2, A) | boundaries
considered

Box (Xc, Yc, Wdth, Hght, A) V within the
region

Diamond (Xc, Yc, Wdth, Hght, A)

Circle (Xc, Yc, R)

Annulus (Xc, Yc, Rin, Rout)

Ellipse (Xc, Yc, Rx, Ry, A)

Elliptannulus (Xc, Yc, Rinx, Riny, Routx, Routy,
Ain, Aout)

Sector (Xc, Yc, Amin, Amax)

where (Xc,Yc) is the coordinate of the shape's center; (X#,Y#) are the coordinates of the shape's edges; Rxxx are the shapes' various Radii or semimajor/minor axes; and Axxx are the angles of rotation (or bounding angles for Sector) in degrees. For rotated shapes, the rotation angle can be left off, indicating no rotation. Common alternate names for the regions can also be used: rotbox = box; rorectangle = rectangle; (rot)rhombus = (rot)diamond; and pie = sector. When a shape's name is preceded by a minus sign, '-', the defined region is instead the area *outside* its boundary (ie, the region is inverted). All the shapes within a single region file are OR'd together to create the region, and the order is significant. The overall way of looking at region files is that if the first region is an excluded region then a dummy included region of the whole detector is inserted in the front. Then each region specification as it is

(+)

processed overrides any selections inside of that region specified by previous regions. Another way of thinking about this is that if a previous excluded region is completely inside of a subsequent included region the excluded region is ignored.

The positional coordinates may be given either in pixel units, decimal degrees or hh:mm:ss.s, dd:mm:ss.s units. The shape sizes may be given in pixels, degrees, arcminutes, or arcseconds. Look at examples of region file produced by fv/POW or ds9 for further details of the region file format.

Next	Up	Previous	Contents
------	----	----------	----------

[FITSIO Home](#)

Next: [5.4.5 Example Row Filters](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.3 Good Time Interval](#)
[Contents](#)

(+)

[Next](#)[Up](#)[Previous](#)[Contents](#)[FITSIO Home](#)

Next: [5.5 Combined Filtering Examples](#) **Up:** [5.4 Table Filtering](#) **Previous:** [5.4.4 Spatial Region Filtering](#) [Contents](#)

5.4.5 Example Row Filters

`[double && mag <= 5.0]` - Extract all double stars
brighter

than fifth magnitude

`[#row >= 125 && #row <= 175]` - Extract row
numbers 125 through 175

`[abs(sin(theta * #deg)) < 0.5]` - Extract all rows having
the

absolute value of the sine of theta
less than a half where the angles
are tabulated in degrees

`[@rowFilter.txt]` - Extract rows using the
expression

contained within the text file
rowFilter.txt

`[gtifilter()]` - Search the current file for a GTI
extension, filter the TIME
column in the current table, using
START/STOP times taken from
columns in the GTI extension

(+)

`[regfilter("pow.reg")]` - Extract rows which have a
coordinate

(as given in the X and Y columns)
within the spatial region specified
in the pow.reg region file.

(+)

[Next](#) [Up](#) [Previous](#) [Contents](#) [FITSIO Home](#)

Next: [6 CFITSIO Error Status](#) **Up:** [5 CFITSIO File Names](#) **Previous:** [5.4.5 Example Row Filters](#) [Contents](#)

5.5 Combined Filtering Examples

The previous sections described all the individual types of filters that may be applied to the input file. In this section we show examples which combine several different filters at once. These examples all use the **fitscopy** program that is distributed with the CFITSIO code. It simply copies the input file to the output file.

```
fitscopy rosat.fit out.fit
```

This trivial example simply makes an identical copy of the input **rosat.fit** file without any filtering.

```
fitscopy 'rosat.fit[events][col Time;X;Y][#row < 1000]'  
out.fit
```

The output file contains only the Time, X, and Y columns, and only the first 999 rows from the 'EVENTS' table extension of the input file. All the other HDUs in the input file are copied to the output file without any modification.

```
fitscopy 'rosat.fit[events][PI < 50][bin (Xdet,Ydet) = 16]'
```

(+)

image.fit

This creates an output image by binning the Xdet and Ydet columns of the events table with a pixel binning factor of 16. Only the rows which have a PI energy less than 50 are used to construct this image. The output image file contains a primary array image without any extensions.

```
fitscopy 'rosat.fit[events][gtifilter() &&
regfilter("pow.reg")]' out.fit
```

The filtering expression in this example uses the **gtifilter** function to test whether the TIME column value in each row is within one of the Good Time Intervals defined in the GTI extension in the same input file, and also uses the **regfilter** function to test if the position associated with each row (derived by default from the values in the X and Y columns of the events table) is located within the area defined in the **pow.reg** text region file (which was previously created with the **fv/POW** image display program). Only the rows which satisfy both tests are copied to the output table.

```
fitscopy 'r.fit[evt][PI<50]' stdout | fitscopy stdin[evt][col
X,Y] out.fit
```

In this somewhat convoluted example, fitscopy is used to first select the rows from the evt extension which have PI less than 50 and write the resulting table out to the stdout stream. This is piped to a 2nd instance of fitscopy (with the Unix ``image`` pipe command) which reads that filtered FITS file from the stdin stream and copies only the X and Y columns from the evt table to the output file.

(+)

```
fitscopy 'r.fit[evt][col RAD=sqrt((X-#XCEN)**2+(Y-#YCEN)**2)][rad<100]' out.fit
```

This example first creates a new column called RAD which gives the distance between the X,Y coordinate of each event and the coordinate defined by the XCEN and YCEN keywords in the header. Then, only those rows which have a distance less than 100 are copied to the output table. In other words, only the events which are located within 100 pixel units from the (XCEN, YCEN) coordinate are copied to the output table.

```
fitscopy 'ftp://heasarc.gsfc.nasa.gov/rosat.fit[events][bin (X,Y)=16]' img.fit
```

This example bins the X and Y columns of the hypothetical ROSAT file at the HEASARC ftp site to create the output image.

```
fitscopy 'raw.fit[i512,512][101:110,51:60]' image.fit
```

This example converts the 512 x 512 pixel raw binary 16-bit integer image to a FITS file and copies a 10 x 10 pixel subimage from it to the output FITS image.

Next	Up	Previous	Contents
------	----	----------	----------

[FITSIO Home](#)

Next: [6 CFITSIO Error Status](#) **Up:** [5 CFITSIO File Names](#) **Previous:** [5.4.5 Example Row Filters](#) **[Contents](#)**

(+)

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)
[FITSIO Home](#)

Next: [About this document ...](#) **Up:** [CFITSIO](#)

Previous: [5.5 Combined Filtering Examples](#)

[Contents](#)

6 CFITSIO Error Status Codes

The following table lists all the error status codes used by CFITSIO. Programmers are encouraged to use the symbolic mnemonics (defined in the file `fitsio.h`) rather than the actual integer status values to improve the readability of their code.

Symbolic Const	Value	Meaning
-----	-----	-----
	0	OK, no error
SAME_FILE	101	input and output files are the same
TOO_MANY_FILES	103	tried to open too many FITS files at once
FILE_NOT_OPENED	104	could not open the named file
FILE_NOT_CREATED	105	could not create the named file
WRITE_ERROR	106	error writing to FITS file
END_OF_FILE	107	tried to move past end of file
READ_ERROR	108	error reading from FITS file
FILE_NOT_CLOSED	110	could not close the file
ARRAY_TOO_BIG	111	array dimensions exceed internal limit
READONLY_FILE	112	Cannot write to readonly file

(+)

MEMORY_ALLOCATION 113 Could not allocate
memory

BAD_FILEPTR 114 invalid fitsfile pointer

NULL_INPUT_PTR 115 NULL input pointer to
routine

SEEK_ERROR 116 error seeking position in file

BAD_URL_PREFIX 121 invalid URL prefix on file
name

TOO_MANY_DRIVERS 122 tried to register too
many IO drivers

DRIVER_INIT_FAILED 123 driver initialization failed

NO_MATCHING_DRIVER 124 matching driver is not
registered

URL_PARSE_ERROR 125 failed to parse input file
URL

SHARED_BADARG 151 bad argument in shared
memory driver

SHARED_NULPTR 152 null pointer passed as an
argument

SHARED_TABFULL 153 no more free shared
memory handles

SHARED_NOTINIT 154 shared memory driver is not
initialized

SHARED_IPCERR 155 IPC error returned by a
system call

SHARED_NOMEM 156 no memory in shared
memory driver

SHARED_AGAIN 157 resource deadlock would
occur

SHARED_NOFILE 158 attempt to open/create lock
file failed

SHARED_NORESIZE 159 shared memory block
cannot be resized at the moment

(+)

HEADER_NOT_EMPTY 201 header already contains
 keywords
 KEY_NO_EXIST 202 keyword not found in header
 KEY_OUT_BOUNDS 203 keyword record number is
 out of bounds
 VALUE_UNDEFINED 204 keyword value field is
 blank
 NO_QUOTE 205 string is missing the closing
 quote
 BAD_KEYCHAR 207 illegal character in keyword
 name or card
 BAD_ORDER 208 required keywords out of order
 NOT_POS_INT 209 keyword value is not a positive
 integer
 NO_END 210 couldn't find END keyword
 BAD_BITPIX 211 illegal BITPIX keyword value
 BAD_NAXIS 212 illegal NAXIS keyword value
 BAD_NAXES 213 illegal NAXISn keyword value
 BAD_PCOUNT 214 illegal PCOUNT keyword
 value
 BAD_GCOUNT 215 illegal GCOUNT keyword
 value
 BAD_TFIELDS 216 illegal TFIELDS keyword
 value
 NEG_WIDTH 217 negative table row size
 NEG_ROWS 218 negative number of rows in
 table
 COL_NOT_FOUND 219 column with this name not
 found in table
 BAD_SIMPLE 220 illegal value of SIMPLE
 keyword
 NO_SIMPLE 221 Primary array doesn't start with
 SIMPLE
 NO_BITPIX 222 Second keyword not BITPIX

(+)

NO_NAXIS	223	Third keyword not NAXIS
NO_NAXES	224	Couldn't find all the NAXISn keywords
NO_XTENSION	225	HDU doesn't start with XTENSION keyword
NOT_ATABLE	226	the CHDU is not an ASCII table extension
NOT_BTABLE	227	the CHDU is not a binary table extension
NO_PCOUNT	228	couldn't find PCOUNT keyword
NO_GCOUNT	229	couldn't find GCOUNT keyword
NO_TFIELDS	230	couldn't find TFIELDS keyword
NO_TBCOL	231	couldn't find TBCOLn keyword
NO_TFORM	232	couldn't find TFORMn keyword
NOT_IMAGE	233	the CHDU is not an IMAGE extension
BAD_TBCOL	234	TBCOLn keyword value < 0 or > rowlength
NOT_TABLE	235	the CHDU is not a table
COL_TOO_WIDE	236	column is too wide to fit in table
COL_NOT_UNIQUE	237	more than 1 column name matches template
BAD_ROW_WIDTH	241	sum of column widths not = NAXIS1
UNKNOWN_EXT	251	unrecognizable FITS extension type
UNKNOWN_REC	252	unknown record; 1st keyword not SIMPLE or XTENSION
END_JUNK	253	END keyword is not blank
BAD_HEADER_FILL	254	Header fill area contains non-blank chars
BAD_DATA_FILL	255	Illegal data fill bytes (not

(+)

zero or blank)

BAD_TFORM 261 illegal TFORM format code
 BAD_TFORM_DTYPE 262 unrecognizable TFORM
 datatype code

BAD_TDIM 263 illegal TDIMn keyword value
 BAD_HEAP_PTR 264 invalid BINTABLE heap
 pointer is out of range

BAD_HDU_NUM 301 HDU number < 1 or >
 MAXHDU

BAD_COL_NUM 302 column number < 1 or >
 tfields

NEG_FILE_POS 304 tried to move to negative byte
 location in file

NEG_BYTES 306 tried to read or write negative
 number of bytes

BAD_ROW_NUM 307 illegal starting row number
 in table

BAD_ELEM_NUM 308 illegal starting element
 number in vector

NOT_ASCII_COL 309 this is not an ASCII string
 column

NOT_LOGICAL_COL 310 this is not a logical
 datatype column

BAD_htable_FORMAT 311 ASCII table column has
 wrong format

BAD_btable_FORMAT 312 Binary table column
 has wrong format

NO_NULL 314 null value has not been defined

NOT_VARI_LEN 317 this is not a variable length
 column

BAD_DIMEN 320 illegal number of dimensions
 in array

BAD_PIX_NUM 321 first pixel number greater
 than last pixel

(+)

ZERO_SCALE 322 illegal BSCALE or TSCALn

keyword = 0

NEG_AXIS 323 illegal axis length < 1

NOT_GROUP_TABLE 340 Grouping function error

HDU_ALREADY_MEMBER 341

MEMBER_NOT_FOUND 342

GROUP_NOT_FOUND 343

BAD_GROUP_ID 344

TOO_MANY_HDUS_TRACKED 345

HDU_ALREADY_TRACKED 346

BAD_OPTION 347

IDENTICAL_POINTERS 348

BAD_GROUP_ATTACH 349

BAD_GROUP_DETACH 350

NGP_NO_MEMORY 360 malloc failed

NGP_READ_ERR 361 read error from file

NGP_NUL_PTR 362 null pointer passed as an
argument.

Passing null pointer as a name of
template file raises this error

NGP_EMPTY_CURLINE 363 line read seems to be
empty (used

internally)

NGP_UNREAD_QUEUE_FULL 364 cannot unread
more than 1 line (or single

line twice)

NGP_INC_NESTING 365 too deep include file
nesting (infinite

loop, template includes itself ?)

NGP_ERR_FOPEN 366 fopen() failed, cannot
open template file

NGP_EOF 367 end of file encountered and not
expected

(+)

NGP_BAD_ARG 368 bad arguments passed.

Usually means

internal parser error. Should not
happen

NGP_TOKEN_NOT_EXPECT 369 token not
expected here

BAD_I2C 401 bad int to formatted string
conversion

BAD_F2C 402 bad float to formatted string
conversion

BAD_INTKEY 403 can't interpret keyword value
as integer

BAD_LOGICALKEY 404 can't interpret keyword
value as logical

BAD_FLOATKEY 405 can't interpret keyword
value as float

BAD_DOUBLEKEY 406 can't interpret keyword
value as double

BAD_C2I 407 bad formatted string to int
conversion

BAD_C2F 408 bad formatted string to float
conversion

BAD_C2D 409 bad formatted string to double
conversion

BAD_DATATYPE 410 illegal datatype code value

BAD_DECIM 411 bad number of decimal places
specified

NUM_OVERFLOW 412 overflow during datatype
conversion

DATA_COMPRESSION_ERR 413 error compressing
image

DATA_DECOMPRESSION_ERR 414 error
uncompressing image

(+)

BAD_DATE 420 error in date or time conversion

PARSE_SYNTAX_ERR 431 syntax error in parser
expression

PARSE_BAD_TYPE 432 expression did not evaluate
to desired type

PARSE_LRG_VECTOR 433 vector result too large to
return in array

PARSE_NO_OUTPUT 434 data parser failed not sent
an out column

PARSE_BAD_COL 435 bad data encounter while
parsing column

PARSE_BAD_OUTPUT 436 Output file not of proper
type

ANGLE_TOO_BIG 501 celestial angle too large for
projection

BAD_WCS_VAL 502 bad celestial coordinate or
pixel value

WCS_ERROR 503 error in celestial coordinate
calculation

BAD_WCS_PROJ 504 unsupported type of celestial
projection

NO_WCS_KEY 505 celestial coordinate keywords
not found

APPROX_WCS_KEY 506 approximate wcs keyword
values were returned

(+)