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 it's 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, than 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**

(+)