1 Table I/O

These notes describe a set of SPP-callable subroutines for working with STSDAS tables, FITS tables, or simple text files in tabular format.

Information placed in tables can be of six different data types, namely, single- and double-precision real, integer, short integer, boolean, and text (character strings). When data are acquired from or deposited into tables, type conversion is done by the routine used for the data transfer in the event that the data type of the table element does not match the buffer supplied by the program.

STSDAS format tables may be row-ordered (the default) or column-ordered. Row-ordered tables are conceptually like sequential files; additional rows may simply be appended to the table, and there is never a need to specify the maximum number of rows. The addition of new columns after the creation of a table requires rewriting the table unless space was allocated for new columns; this is handled within the interface, however, and is transparent to the user and to the programmer aside from the time delay while the file is being rewritten. There is no intrinsic limit on the number of columns or rows.

Tables in FITS files can also be read and written. This option uses the HEASARC FITSIO package of subroutines. A FITS table can be in any extension of the file (but not the primary header/data unit); the first extension is number one. Both BINTABLE (binary) and TABLE (ASCII) extensions can be read or modified, but only BINTABLE can be created. For FITS tables there is a limit of 999 on the number of columns, due to the eight-character limit on the names of keywords used for defining column properties.

In addition to binary tables, this interface can also read and write text files in row and column format. The column names are fixed at "c1", "c2", "c3", etc., the data type and print format for each column of an input table are determined by the contents of the file, and there are no units. While the file name of a binary table must have an extension (the default is ".tab"), the name of a text table need not have an extension. Text tables may be read from STDIN and written to STDOUT.

Expressions can be appended to a table name in order to restrict the rows and/or columns that the table appears to have. Typically, it is the user who includes these expressions with the table name, and the applications code does not need any modification to take advantage of this feature. The option to specify an array section (similar to an image section) for a column that contains arrays is implemented in the column selector code, but it is not supported by most of the table I/O routines. See the "row and column selectors" section for further information.

The object library containing these subroutines is located in the TABLES external package, which is specified by the flag "-p tables" when using either mkpkg or xc. This also allows xc to find the include file <tbset.h>. The library should be specified as "-ltbtables" in the link statement in a mkpkg file or when using xc directly. For example, if you have a source file "testtab.x", you could compile and link using "mkpkg -p tables", where the mkpkg file contains the following three lines :

```
$omake testtab.x
$link testtab.o -ltbtables
$exit
```

Undefined elements are flagged with special values that are internal to the interface. IEEE NaN is used for floating point columns in FITS BINTABLE extensions. Presently, it is not possible to specify an indefinite boolean element; the default boolean value is false. An integer or short integer column can be used instead of boolean, with zero for false and one for true; this allows undefined values as well. An empty (null) character string is considered undefined.

When columns are defined (created), five characteristics may be specified :

name	string
data type	symbolic integer
number of elements	integer
units	string
print format	string

The column name and data type are required, but null strings may be given for the print format and units.

The data type is specified as an integer. For types real, double, int, short and boolean the integer values are the IRAF symbolic values TY_REAL, TY_DOUBLE, TY_INT, TY_SHORT and TY_BOOL. For character-string columns, however, the data type is specified as -N, where N is the maximum number of char in an element of that column.

The number of elements should be set to one for ordinary scalar columns. A value greater than one means that each table "cell" (i.e. at the intersection of a row and column) contains an array of that many elements. The arrays are fixed in length for a given column, though not all elements need to be populated. A character string is regarded as a single element, and a column may contain arrays of character strings. For example, if a columns was defined with data type -8 and array length 2, the value of the column at a particular row could be the pair of strings "RA—TAN", "DEC–TAN".

The print format for table elements is specified as an SPP format, including the leading percent sign. The *w* field of most formats may be negative, which results in a left-justified value.

It is important to note that some character information is *case sensitive* and some is not. In particular, table names are case sensitive. It is therefore important that user programs not perform case conversion on table names retrieved from the CL; table names given explicitly in code would typically be in *lower* case. Column names are case insensitive in the sense that when searching for a column in a table, the search will be done without regard for case. When creating a column, however, the column name may be given in upper, lower, or mixed case according to the preference of the user, and the name will be displayed the same way. This is in contrast to table header parameters, whose names are converted to upper case when the parameters are added to the table. Case conversion is *not* done by any interface routine when getting or putting values of type character string.

The lengths and case sensitivity of character strings used in tables and in these interface routines are as follows :

table filename	SZ_FNAME (255) characters	case sensitive
keyword name	SZ_KEYWORD (8) characters	case insensitive
column name	SZ_COLNAME (20) characters	case insensitive
column units	SZ_COLUNITS (20) characters	case sensitive
column print format	SZ_COLFMT (8) characters	case sensitive
character column	1022 characters	case sensitive

Other routines that are provided (and are listed in "tables$doc/calls.doc") but are not described in this document include :

tbtflu	flush FIO buffer for table
tbrcpy	copy an entire row (only for tables with identical columns)
tbrswp	swap two rows
tbrdel	delete a range of rows
tbrnll	set all columns in a range of rows to INDEF
tbtsrt	sort a table by values in column

1.1 Table initialization routines

For existing tables, *tbtopn* is the only call required; when a new table is being created, *tbtcre* must also be called to actually open the table.

Table header parameters and column description information are stored in the table together with the data, so if more parameters are written or more columns are defined than there is space in the table for, the table will be rewritten to allocate more room. If the table is large, this can be a nuisance either due to the time involved or because of disk space limitations. When a new table is created, it is wise to specify the amount of space allocated for header parameters and column descriptions carefully.

TBTOPN Initialize a table

This function initializes the data structure for a table and returns a pointer to that structure. If the I/O mode is NEW_FILE or NEW_COPY, then after calling this routine parameters may be set using *tbpset* and columns may be defined using *tbcdef*, and the table must be opened explicitly using *tbtcre*. If the I/O mode is NEW_COPY then the column definitions from the template table will be copied to the new table, and the table must actually be created by a subsequent call to tbtcre. Since a new table is not opened by tbtopn, the header parameters are *not* copied by specifying the I/O mode as NEW_COPY. There is a procedure *tbhcal* that copies all header parameters from one open table to another. If the I/O mode is not NEW_COPY, then the template is ignored. If the I/O mode is READ_ONLY or READ_WRITE, then *tbtopn* opens the table.

The table descriptor that is returned as the function value should be used in all subsequent calls until the table is closed.

```
tbtopn (tablename, iomode, template)
```

`tablename`	I	string	name of table
`iomode`	O	int	I/O mode
`template`	I	pointer	pointer to descriptor for template

TBTCRE Create a new table

After calling *tbtopn* (and optionally *tbpset* and *tbcdef*), this routine must be called to actually create the table. If the table already exists, this routine should not be called.

```
call tbtcre (tp)
```

`tp`	I	pointer	table descriptor

TBTCLO **Close a table**

Close a table. After *tbtclo* is called, the value of the table descriptor and the values of all column pointers for the table will be undefined.

```
call tbtclo (tp)
```

`tp`	I	pointer	table descriptor

1.2 Getting and setting table parameters

The functions provided by these routines permit a user to modify parameters that characterize the table itself or the behavior of the interface.

TBPSTA **Get the value of a parameter**

This function returns the value of a table description parameter or an internal table interface parameter, such as the number of rows or columns in the table.

```
tbpsta (tp, tparm)
```

`tp`	I	pointer	table descriptor
`tparm`	I	int	symbolic code; specifies parameter to get

The symbolic parameters to specify table parameters are given in <tbset.h>.

TBPSET **Set a parameter**

Change the value of a table description parameter or an internal table interface parameter. This routine may be used to specify such things as the row length or number of rows for a table, preferably before the table is actually opened.

```
call tbpset (tp, tparm, buffer)
```

`tp`	I	pointer	table descriptor
`tparm`	I	int	symbolic code; specifies parameter to set
`buffer`	I	int	value of parameter to be set

The symbolic parameters to specify table parameters are given in <tbset.h>.

1.3 Defining columns and managing column descriptors

TBCDEF **Define new column(s)**

Define (create) new columns. Note that all but the first and last items in the calling sequence are arrays. The parameters SZ_COLNAME, SZ_COLUNITS, and SZ_COLFMT are defined in <tbset.h>. The table data types are specified as TY_REAL, TY_DOUBLE, TY_INT, TY_SHORT, TY_BOOL; for a character string column use -N, where N is the maximum number of char in one element. The value of `nelem` for a column should be one if the column is to contain ordinary scalar values. In order to define a column such that the value in each row is an array, set `nelem` to the (fixed) number of elements in the array.

If the table is new, it is preferable to define all the columns before opening the table, i.e. after calling *tbtopn* but before calling *tbtcre*. This approach is faster and makes it unnecessary to specify the row length explicitly (which can be done by calling *tbpset*). It is still possible to call *tbcdef* after the table is open, but it will take additional time if the table needs to be reorganized to allocate more space.

Use the `colptr` pointer(s) for all subsequent references to column(s) until the table is closed.

```
call tbcdef (tp, colptr, colname, colunits, colfmt, datatype, nelem,
              numcols)
```

`tp`	I	pointer	table descriptor
`colptr`	O	pointer	array of column pointers
`colname`	I	string	array [SZ_COLNAME,numcols] of column names
`colunits`	I	string	array [SZ_COLUNITS,numcols] of units for the columns
`colfmt`	I	string	array [SZ_COLFMT,numcols] of formats for display of the columns
`datatype`	I	int	array of data types for columns (note: -N for char strings)
`nelem`	I	int	array of lengths of each item
`numcols`	I	int	number of columns

Column units and print formats may be specified as null strings. If a null print format is given, a sensible default will be assigned.

TBCDEF1 **Define a new column**

Define (create) a new column. This is similar to *tbcdef* except that it creates a single column. This avoids the complication of using arrays of character strings.

```
call tbcdef1 (tp, colptr, colname, colunits, colfmt, datatype, nelem)
```

`tp`	I	pointer	table descriptor
`colptr`	O	pointer	column pointer
`colname`	I	string	column name
`colunits`	I	string	units for the column
`colfmt`	I	string	print format for display of the column
`datatype`	I	int	data type of the column (note: -n for char strings)
`nelem`	I	int	array length, or one for a scalar column

TBCFND — Get column pointer(s) from column name(s)

For each column name, this procedure returns a column-descriptor pointer if the column was found in the table, and it returns NULL if the column was not found. Note that it is not an error for a column to be not found. Use the `colptr` pointer(s) for all subsequent references to column(s) until the table is closed.

```
call tbcfnd (tp, colname, colptr, numcols)
```

`tp`	I	pointer	table descriptor
`colname`	I	string	array [SZ_COLNAME,numcols] of column names
`colptr`	O	pointer	array of column pointers
`numcols`	I	int	number of names in `colname`

TBCFND1 — Get a column pointer from a column name

This is similar to *tbcfnd* except that it takes a single column name and returns a single column pointer. This avoids the complication of using an array of character strings.

```
call tbcfnd1 (tp, colname, colptr)
```

`tp`	I	pointer	table descriptor
`colname`	I	string	column name
`colptr`	O	pointer	column pointer

TBCNUM — Get column pointer from column number

This function returns a column pointer for a column, based on column number.

```
tbcnum (tp, colnumber)
```

`tp`	I	pointer	table descriptor
`colnumber`	I	int	column number

The value of `colptr` will be NULL if the column number is out of range.

TBCINF Get info about a column

Return all available information about an existing column.

```
call tbcinf (colptr, colnum, colname, colunits, colfmt, datatype, nelem,
             lenfmt)
```

`colptr`	I	pointer	column pointer
`colnum`	O	int	column number
`colname`	O	string	column name
`colunits`	O	string	units for the column
`colfmt`	O	string	format for display of the column
`datatype`	O	int	symbolic code; data types for column
`nelem`	O	int	number of items in table element; this will be one for scalar columns, more than one for arrays
`lenfmt`	O	int	number of char for printing using colfmt

TBCIGI Get integer information about a table column

This function gets information about a column and returns it as the function value, which is of type integer. The corresponding subroutine for text string information is *tbcigt*.

```
tbcigi (colptr, cparm)
```

`colptr`	I	pointer	column pointer
`cparm`	I	int	symbolic code; specifies parameter to get

The symbolic codes are defined in <tbset.h>; the codes and their meaning are as follows :

TBL_COL_DATATYPE	data type of column
TBL_COL_FMTLEN	width of field needed for printing
TBL_COL_NUMBER	column number
TBL_COL_LENDATA	number of array elements, or one if a scalar column

TBCIGT Get text string information about a table column

This subroutine gets information about a table column. The value returned as `info` is a character string. The corresponding function for integer information is *tbcigi*.

```
call tbcigt (colptr, cparm, info, maxch)
```

`colptr`	I	pointer	column pointer
`cparm`	I	int	symbolic code; specifies parameter to get
`info`	O	string	buffer to receive value
`maxch`	I	int	allocated length of output string

The symbolic codes are defined in <tbset.h>; the codes and their meaning are as follows :

TBL_COL_NAME	column name
TBL_COL_FMT	print format for column
TBL_COL_UNITS	column units

TBCNAM **Change column name**

Change the name of an existing column.

```
call tbcnam (tp, colptr, colname)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`colname`	I	string	column name

TBCNIT **Change column units**

Change the units of an existing column.

```
call tbcnit (tp, colptr, colunits)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`colunits`	I	string	units for the column

TBCFMT **Change column format**

Change the print format of an existing column.

```
call tbcfmt (tp, colptr, colfmt)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`colfmt`	I	string	format for display of the column

1.4 Table header I/O routines

Table header parameters (keywords) are used to store information that is generic for an entire table. The data can serve as identification, be flags for processing, etc. Parameters giving table size, column data types, etc. that are used by routines in this interface are *not* stored in the same area; these parameters may be set and retrieved by routines described in previous sections.

Five data types are supported for table header keywords, real, double precision, integer, boolean, and text, although character strings are limited to 70 characters in length. The distinction means little, however, since the values are stored as text strings and data type conversion is performed as needed. These routines get and put *single* keywords, not arrays. A user program may construct array elements by appending numbers to a root portion of a keyword, and the array elements may be gotten or put one at a time. The names of table header keywords are always converted to upper case for FITS compatibility.

TBHCAL   Copy all header keywords

This routine copies all header keywords from the input table to the output table. This would be used after creating a new table if the header keywords in an existing table are relevant to the new table, e.g. after opening a table NEW_COPY. Both tables must be open (not just initialized) before calling this routine.

```
call tbhcal (itp, otp)
```

`itp`	I	pointer	descriptor for input table
`otp`	I	pointer	descriptor for output table

TBHGT[BIRD]   Get a header keyword

This function returns the value of a keyword from the table header. A character-string keyword may be gotten by the procedure *tbhgtt*.

Note that the comment (if any) associated with the header keyword will not be returned. Use *tbhgcm* to get the comment.

```
tbhgt[bird] (tp, keyword)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name

TBHGTT   Get a header keyword

Get a character-string keyword from the table header. See *tbhgcm* for getting an associated comment, if any.

```
call tbhgtt (tp, keyword, buffer, maxch)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name
`buffer`	O	string	output buffer
`maxch`	I	int	maximum number of characters to get

TBHAD[TBIRD] **Add a header keyword**

Add a keyword to the table header and put its value. If the keyword already exists, its value will be replaced. See *tbhpcm* for adding or replacing a comment.

```
call tbhad[tbird] (tp, keyword, buffer)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name
`buffer`	I	—	buffer of appropriate data type

TBHPT[TBIRD] **Put a header keyword**

Replace the value of a keyword in the table header. It is an error if the keyword does not exist; use *tbhad[]* to *add* keywords to a table header. See *tbhpcm* for adding or replacing a comment.

```
call tbhpt[tbird] (tp, keyword, buffer)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name
`buffer`	I	—	buffer of appropriate data type

TBHGCM **Get associated comment**

This routine returns the comment associated with a header keyword. If there is no comment, a null string will be returned. It is an error if the keyword is not found in the table.

```
call tbhgcm (tp, keyword, comment, maxch)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name
`comment`	O	string	comment
`maxch`	I	int	maximum size of comment string

TBHPCM **Add or replace a comment**

This routine adds or replaces a comment for a header keyword. It is an error if the keyword is not found in the table.

```
call tbhpcm (tp, keyword, comment)
```

`tp`	I	pointer	table descriptor
`keyword`	I	string	keyword name
`comment`	I	string	comment

TBHGNP **Get Nth header keyword**

Get a keyword by number from the table header. The data type is a single char: t, b, i, r, d. If the keyword number is out of range then null strings will be returned for the keyword name and value.

```
call tbhgnp (tp, keynum, keyword, dtype, buffer)
```

`tp`	I	pointer	table descriptor
`keynum`	I	int	keyword number
`keyword`	O	string	keyword name
`dtype`	O	char	data type of keyword
`buffer`	O	string	string containing keyword value

1.5 Row and column selectors

Of the routines in this section, only the first two, *tbsirow* and *tbcdes*, really depend on the row or column selectors. *tbcnel* is intended to be used in conjunction with *tbcga[rd]*, and the latter can be applied to any table, but its usefulness is greatly enhanced by the row and column selectors. *tbcga[rd]* supports the column array section feature (similar to image sections) by calling *tcs_rdary[rd]* instead of *tbagt[rd]* to get the data.

TBSIROW **Get row number from selected row**

If a row selector was specified when the table was opened, the table appears to have fewer rows than does the underlying table. Rows that were not selected are simply not visible to the application. The term "selected row number" refers to a row number in this smaller table; the selected row numbers run from one to the total number of selected rows. When referring to a row in the table, the row number specified should always be the selected row number. There may be times, however, when it is desired to know the actual row number, the number in the underlying table. This subroutine returns that actual row number.

If no row selector was specified when the table was opened, then `rownum` will be set equal to `selrow`.

It is an error if `selrow` is greater than the number of rows in the table.

```
call tbsirow (tp, selrow, rownum)
```

`tp`	I	pointer	table descriptor
`selrow`	I	int	selected row number
`rownum`	O	int	actual row number

TBCDES **Get column selector descriptor**

This function returns the pointer to the column selector descriptor. This descriptor is needed for use of such routines as *tcs_rdary[]*.

```
tbcdes (tp, cp)
```

`tp`	I	pointer	table descriptor
`cp`	I	pointer	column pointer

TBCNEL Get number of elements in a column

This function returns the total number of elements in a column. This is simply the number of rows multiplied by the array size for the column (the array size is one for ordinary scalar columns). The number of rows in the table may have been reduced by the use of a row selector, and the array size for the column may have been reduced by the use of an array section.

The main purpose of this routine is to determine how large an array to allocate before calling *tbcga[rd]*.

```
tbcnel (tp, cp)
```

`tp`	I	pointer	table descriptor
`cp`	I	pointer	column pointer

TBCGA[RD] Get all elements in a column

This function gets all elements in all rows of a column. The idea is that this one routine can be used to get either a column of values for all rows or an array of values at one row. In the latter case, the user would typically have used the row selector syntax to limit the table to just one row. This simplifies applications code for the case that an array of values is to be gotten from a table, because the application does not need to check whether the column contains scalars or arrays.

For a column of scalars, this routine returns the same values as would be gotten by *tbcgt[rd]*. For a column of arrays, this routine effectively returns the values gotten by *tbagt[rd]*, concatenating the arrays if more than one row was selected. In the case of a column of arrays, however, a column array section may have been used when the table was opened, in which case only the specified sections will be gotten. The *tcs_rdary[rd]* routine is actually used instead of *tbagt[rd]* in order to take advantage of this feature. Note that the other I/O routines described in this document do not support the column array section feature; the *tcs_* routines such as *tcs_rdary[]* (see below) can be called by applications code for this option.

`nelem` is the allocated size of `buffer`, and `nelem` can be larger than the number of elements which will be gotten. The function *tbcnel* can be used to get the number of elements, and the buffer can be allocated with that size. It is an error if the buffer size is too small (i.e. less than the value returned by *tbcnel*).

The function value is the total number of elements that were actually gotten.

```
tbcga[rd] (tp, cp, buffer, nelem)
```

`tp`	I	pointer	table descriptor
`cp`	I	pointer	column pointer
`buffer`	O	—	buffer to receive values
`nelem`	I	int	buffer size

TCS_RDARY[BIRDS] **Read an array using the column selector**

This routine reads one or more elements at one row and column of a table. If the column contains arrays, the number of elements may have been reduced by the use of a column array section (similar to an image section); only the elements in the section will be returned.

```
call tcs_rdary[birds] (tp, descrip, rownum, maxbuf, nbuf, buffer)
```

`tp`	I	pointer	table descriptor
`descrip`	I	pointer	column selector descriptor
`rownum`	I	int	row number from which to get data
`maxbuf`	I	int	maximum number of elements to read
`nbuf`	O	int	actual number of elements read
`buffer`	O	—	array of appropriate data type

TCS_RDARYT **Read an array using the column selector**

This routine reads one or more character strings at one row and column of a table. If the column contains arrays, the number of elements may have been reduced by the use of a column array section (similar to an image section); only the elements in the section will be returned.

```
call tcs_rdaryt (tp, descrip, rownum, lenstring, maxbuf, nbuf, buffer)
```

`tp`	I	pointer	table descriptor
`descrip`	I	pointer	column selector descriptor
`rownum`	I	int	row number from which to get data
`lenstring`	I	int	size of first axis of buffer
`maxbuf`	I	int	maximum number of elements to read
`nbuf`	O	int	actual number of elements read
`buffer`	O	string	output buffer; array of character strings

1.6 Table data I/O routines

tbcga[rd] could have been included in this section, but it was described just above, in the "row and column selectors" section.

TBRCSC **Copy selected columns of a row**

This procedure copies a row from one table to another or to another location in the same table. Only the specified columns are copied, but all columns may be selected, of course. To copy an entire row, the pointers to column descriptors could be obtained as follows:

```
do i = 1, tbpsta (itp, TBL_NCOLS) {
    icolptr[i] = tbcnum (itp, i)
    ocolptr[i] = tbcnum (otp, i)
}
```

```
call tbrcsc (itp, otp, icolptr, ocolptr, irow, orow, ncols)
```

`itp`	I	pointer	descriptor for input table
`otp`	I	pointer	descriptor for output table
`icolptr`	I	pointer	array of column pointers in input table
`ocolptr`	I	pointer	array of column pointers in output table
`irow`	I	int	row number in input table
`orow`	I	int	row number in output table
`ncols`	I	int	number of columns to be copied

The *tbrcpy* routine copies an entire row without the need to get the column pointers, but it can only be used under certain conditions. *tbrcpy* would be appropriate for copying rows within one table, or to an output table which was opened NEW_COPY.

TBEGT[BIRDS] **Get an element from the table**

Get an element from the specified row and column. The buffer to receive the value should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of the column in the table is not the same. If the element is undefined in the table and the data type is real, double, int or short, the buffer value will be INDEFR, INDEFD, INDEFI or INDEFS respectively.

```
call tbegt[birds] (tp, colptr, rownum, buffer)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number from which to get data
`buffer`	O	—	buffer of appropriate data type

TBEGTT **Get an element from the table**

Get an element from the specified row and column. Type conversion will be performed if the column in the table is not a character string. A null string will be returned if the entry in the table is not defined.

```
call tbegtt (tp, colptr, rownum, buffer, maxch)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number from which to get data
`buffer`	O	string	char string buffer
`maxch`	I	int	size of buffer

TBEPT[TBIRDS] **Put an element into the table**

Put an element into the specified row and column. The buffer containing the value should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of the column in the table is not the same.

```
call tbept[tbirds] (tp, colptr, rownum, buffer)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number into which to put data
`buffer`	I	—	buffer of appropriate data type

TBAGT[BIRDS] **Get an array of elements from the table**

This function gets an array of elements from the specified row and column. If the table column contains arrays rather than scalar elements, then this function will read an array of up to `nelem` values starting with element number `first`. The function value is the actual number of elements read. It is not an error for `nelem` to be greater than the number of elements in the array (starting with `first`). This function may be used for getting scalar elements by setting both `first` and `nelem` to one. Type conversion will be performed if the column in the table is not the same type as the buffer. An INDEF value of the appropriate type will be returned for each element of the array which is not defined.

```
tbagt[birds] (tp, colptr, rownum, buffer, first, nelem)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number from which to get data
`buffer`	O	—	array of appropriate data type
`first`	I	int	number of the first array element to read
`nelem`	I	int	maximum number of elements to read

TBAGTT **Get an array of character strings from the table**

This function gets an array of character strings from the specified row and column. The function value is the actual number of elements read. This is similar to *tbagt[]* except for the `lenstring` argument. The buffer to receive the values should be an array of character strings; the size of the first dimension should be `lenstring`. If the column in the table has a numerical or boolean data type, the values will be written into the output buffer using the print format for the column; values may be truncated if `lenstring` is too small.

```
tbagtt (tp, colptr, rownum, buffer, lenstring, first, nelem)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number from which to get data
`buffer`	O	string	array of character strings
`lenstring`	I	int	number of char in each element of buffer
`first`	I	int	number of the first array element to read
`nelem`	I	int	maximum number of elements to read

TBAPT[BIRDS] Put an array of elements into the table

This routine puts an array of elements into the specified row and column. Note that this puts the array into one column in one row. The buffer containing the values should be an array of the data type specified by the routine name suffix; type conversion will be performed if the data type of the column in the table is not the same. It *is* an error if the column in the table contains fewer elements than **`nelem`**+**`first`**−1.

```
call tbapt[birds] (tp, colptr, rownum, buffer, first, nelem)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number into which to put data
`buffer`	I	—	array of values of appropriate data type
`first`	I	int	number of the first array element to write
`nelem`	I	int	number of elements to write

TBAPTT Put an array of character strings into the table

This routine puts an array of character strings into the specified row and column. Note that this puts the array into one column in one row. The buffer containing the values should be an array of character strings; the size of the first dimension should be **`lenstring`**. If the column in the table has a numerical or boolean data type, values will be read from the input strings and written into the table. It *is* an error if the column in the table contains fewer elements than **`nelem`**+**`first`**−1.

```
call tbaptt (tp, colptr, rownum, buffer, lenstring, first, nelem)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`rownum`	I	int	row number into which to put data
`buffer`	I	string	array of character strings
`lenstring`	I	int	number of char in each element of buffer
`first`	I	int	number of the first array element to write
`nelem`	I	int	number of elements to write

TBRGT[BIRDS] **Get values from row**

Get one or more values from a row. The buffer to receive the values should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of the column in the table does not match.

```
call tbrgt[birds] (tp, colptr, buffer, nullflag, numcols, rownum)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	array of column pointers
`buffer`	O	—	buffer of appropriate data type
`nullflag`	O	boolean	array of flags; true if the element is undefined
`numcols`	I	int	number of column values to get
`rownum`	I	int	row number from which to get data

TBRGTT **Get values from row**

Get one or more character-string values from a row. If the data type of a column in the table is numeric or Boolean, the value will be written into the output buffer using the print format for that column; a portion of the value will be lost if the buffer is not large enough.

```
call tbrgtt (tp, colptr, buffer, nullflag, lenstring, numcols, rownum)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	array of column pointers
`buffer`	O	string	array of strings for output buffer
`nullflag`	O	boolean	array of flags; true if the element is undefined
`lenstring`	I	int	number of char in each element of buffer
`numcols`	I	int	number of column values to get
`rownum`	I	int	row number from which to get data

TBRPT[BIRDS] **Put values into a row**

Put one or more values of the same data type into a row. The buffer containing the values should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of a column in the table does not match.

```
call tbrpt[birds] (tp, colptr, buffer, numcols, rownum)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	array of column pointers
`buffer`	I	—	buffer of appropriate data type
`numcols`	I	int	number of column values to put
`rownum`	I	int	row number into which to put data

TBRPTT **Put values into a row**

Put one or more character-string values into a row. If the data type of a column in the table is numeric or Boolean, the value will be read from the character string.

```
call tbrptt (tp, colptr, buffer, lenstring, numcols, rownum)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	array of column pointers
`buffer`	I	string	input buffer; array of char strings
`lenstring`	I	int	number of char in each element of buffer
`numcols`	I	int	number of column values to put
`rownum`	I	int	row number into which to put data

TBCGT[BIRDS] **Get values from a column**

Get an array of values from a single column. Values in rows `firstrow` through `lastrow` (inclusive) will be returned. The buffer to receive the values should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of a column in the table does not match.

```
call tbcgt[birds] (tp, colptr, buffer, nullflag, firstrow, lastrow)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`buffer`	O	—	buffer of appropriate data type
`nullflag`	O	boolean	array of flags; true if the element is undefined
`firstrow`	I	int	number of the first row from which to get data
`lastrow`	I	int	number of the last row from which to get data

Note that `colptr` is a single column pointer, not an array of pointers.

TBCGTT **Get values from a column**

Get an array of character-string values from a single column. Values in rows `firstrow` through `lastrow` (inclusive) will be returned. If the data type of the column in the table is numeric or Boolean, the value will be written into the output buffer using the print format for that column; a portion of the value will be lost if the buffer is not large enough.

```
call tbcgtt (tp, colptr, buffer, nullflag, lenstring, firstrow, lastrow)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`buffer`	O	string	output buffer; array of char strings
`nullflag`	O	boolean	array of flags; true if the element is undefined
`lenstring`	I	int	number of char in each element of the buffer
`firstrow`	I	int	number of the first row from which to get data
`lastrow`	I	int	number of the last row from which to get data

Note that `colptr` is a single column pointer, not an array of pointers.

TBCPT[BIRDS] Put values into a column

Put an array of values into a column. Values in rows `firstrow` through `lastrow` (inclusive) will be set. The buffer containing the values should be of the data type specified by the routine name suffix; type conversion will be performed if the data type of a column in the table does not match.

```
call tbcpt[birds] (tp, colptr, buffer, firstrow, lastrow)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`buffer`	I	—	buffer of appropriate data type
`firstrow`	I	int	number of the first row into which to put data
`lastrow`	I	int	number of the last row into which to put data

Note that `colptr` is a single column pointer, not an array of pointers.

TBCPTT Put values into a column

Put an array of character-string values into a column. Values in rows `firstrow` through `lastrow` (inclusive) will be set. If the data type of a column in the table is numeric or Boolean, the value will be read from the character string.

```
call tbcptt (tp, colptr, buffer, lenstring, firstrow, lastrow)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	column pointer
`buffer`	I	string	input buffer; array of char strings
`lenstring`	I	int	number of char in each element of buffer
`firstrow`	I	int	number of the first row into which to put data
`lastrow`	I	int	number of the last row into which to put data

Note that `colptr` is a single column pointer, not an array of pointers.

TBRUDF Set elements to undefined

Mark table elements in column(s) in given row as indefinite. Note that indefinite is the default, so it is not necessary to call this routine unless definite value(s) have already been set in the table.

```
call tbrudf (tp, colptr, numcols, rownum)
```

`tp`	I	pointer	table descriptor
`colptr`	I	pointer	array of column pointers
`numcols`	I	int	number of elements in array colptr
`rownum`	I	int	row number

1.7 Table files

TBTCPY **Copy a table**

This routine copies a table. The extensions may be absent from the input and output names, but the names must be otherwise complete; i.e. they must not include wild-card characters or be directory names without file names. The input table should not be open for writing, and the input and output names should not be the same.

To a certain extent, the table type can be changed by calling this routine. If `outname` is "STDOUT", the output will be a text table and will be sent to the standard output. If the filename extension for `outname` is recognized as implying a FITS file (".fits", ".fit", or ".??f"), that file will be written as a FITS table. If `inname` is a text table or "STDIN", the output will also be a text table unless the filename extension in `outname` is a FITS extension or is ".tab". Otherwise, the output will be an STSDAS format table.

```
call tbtcpy (inname, outname)
```

`inname`	I	string	name of input table
`outname`	I	string	name of output table

TBTREN **Rename a table**

This routine renames a table. The extensions may be absent from the old and new names, but the names must be otherwise complete; i.e. they must not include wild-card characters or be directory names without file names. The table should not be open for writing.

This is simply *tbtcpy* followed by *tbtdel*. For a table in a FITS file, therefore, this routine moves only the table extension. Even if the table is the only extension in the file, the primary header will still remain. Use *rename* to rename an entire file.

```
call tbtren (oldname, newname)
```

`oldname`	I	string	old name of table
`newname`	I	string	new name of table

TBTDEL **Delete a table**

This routine deletes a table. The extensions may be absent from the table name, but the name must be otherwise complete; i.e. it must not include wild-card characters. The table should not be open.

For a table in a FITS file, this routine deletes only the table extension. Even if the table is the only extension in the file, the primary header will still remain. Use *delete* to delete an entire file.

```
call tbtdel (tablename)
```

`tablename`	I	string	name of the table

TBTCHS **Change the size of a table**

This routine changes the sizes of the various sections of a table. The table does not have to be open, just initialized. If the table is open it will be rewritten (if necessary) by copying to a scratch file, deleting the original table, and renaming the scratch file to the original name.

A negative value for any of the new table size parameters is taken to mean that that parameter should not be changed. A value (e.g. zero) that is smaller than the minimum allowed value results in the truncation of unused space.

The row length is relevant only for row-ordered tables, and the number of rows to allocate is relevant only for column-ordered tables.

```
call tbtchs (tp, maxpar, maxcols, rowlen, allrows)
```

`tp`	I	pointer	table descriptor
`maxpar`	I	int	new maximum number of header parameters
`maxcols`	I	int	new maximum number of columns
`rowlen`	I	int	new value for row length
`allrows`	I	int	new value for number of allocated rows

TBTEXT **Append default extension**

If the input table name does not include an extension then the default extension will be appended. The same actual argument may be used for both the input and output names.

```
call tbtext (inname, outname, maxch)
```

`inname`	I	string	name of the table
`outname`	O	string	name including extension
`maxch`	I	int	maximum number of char in name of table

TBTACC   Test for the existence of a table

This integer function returns YES if the table exists, NO if not. The test is performed by opening the table READ_ONLY, and if that fails, the function returns NO. Thus, there may be a file of the given name, but if it is not a table, or if it is read protected, then it is not considered to exist.

```
tbtacc (tablename)
```

`tablename`	I	string	name of the table

TBTNAM   Get the name of a table

This routine may be used to get the name of an open table, e.g. for constructing a message for the user. If a binary table was opened using a file name without an extension, the default extension would have been appended by *tbtopn*, and the name returned by this subroutine will include the extension.

```
call tbtnam (tp, tablename, maxch)
```

`tp`	I	pointer	table descriptor
`tablename`	O	string	name of the table
`maxch`	I	int	maximum number of char in name of table