Simple and Light Interfaces for C and C++ users

SFITSIO User's Reference Guide

Version 1.4.2[not finished] 2013-05-19

Credits

Software Development:

Chisato Yamauchi

Manual Document:

Chisato Yamauchi, *Ken Ebisawa* and *IMC LTD.*

Manual Translation:

Space Engineering Development Co.,LTD., *Sakura Academia Corporation*, *KOYOSHOUJI CO.,LTD.* and *Yuka Nojo*

Special Thanks:

Daisuke Ishihara, *Hajime Baba*, *Keiichi Matsuzaki*, *Michitaro Koike*, *Yukio Yamamoto* and *Tomotsugu Goto*

Web page: `http://www.ir.isas.jaxa.jp/~cyamauch/sli/`

Contents

1 Introduction

1.1 How long does it take to answer these questions using your favorite FITS I/O library?

Question 1
The FITS file called "`foo.fits.gz`" contains multiple binary table HDU. Write some code that reads the file and set the variable called `flag` to `1` if the value of the first row of the column "X" within the binary table "A" is not NULL and the value of the first row of the column "Y" within the binary table "B" is larger than zero. Please take into account `TZERO`n and `TNULL`n, which are set in the header. The various error checks and code fragment between `#include` and function `main` can be omitted.

Example solution to Question 1 using SFITSIO

```
fitscc fits;
long row_idx = 0;
double v;
fits.read_stream("foo.fits.gz");                              /* Read the file */
if ( isfinite(fits.table("A").col("X").dvalue(row_idx)) && /* Test the value */
     isfinite(v=fits.table("B").col("Y").dvalue(row_idx)) && 0 < v ) flag = 1;
```

Question 2
The file called "`obj0.fits`" contains observation data from four CCD chips. The resolution of each CCD chip is 1024 by 2048, with the four of them stored as a single image of 4096 by 2048 resolution in the Primary HDU of `obj0.fits`. Read the FITS file and generate four separate FITS files, with each having an HDU for the corresponding CCD chip. The names of the generated files should be "CCD0", "CCD1", etc. Please take into account `BZERO`, which is set in the header. Thet various error checks and code fragment between `#include` and function `main` can be omitted.

Example solution to Question 2 using SFITSIO

```
const char *name[] = {"CCD0","CCD1","CCD2","CCD3"};
fitscc fits0, fits1;                          /* Original FITS and new FITS */
fits0.read_stream("obj0.fits");               /* Read observation frame */
fits_image &pri0_img = fits0.image("Primary"); /* Define convenient alias */
for ( i=0 ; i < 4 ; i++ ) {                       /* Append and copy HDUs */
  fits1.append_image(name[i],0, pri0_img.type(), 1024,2048);
  pri0_img.copyf(&(fits1.image(i)), "%d:%d,*", 1024*i, 1024*i + 1023);
}
fits1.write_stream("obj0_separated.fits");    /* Save as separate files */
```

Question 3
The bias frame called "`bias.fits.gz`" and the flat frame "`flat.fits.gz`" are preprocessed. There is also an unprocessed observation frame "`obj1.fits`". Create a FITS using the observation frame that eliminates the upper and lower quarters after being processed using overscan (average value) and the bias subtracted and flat-field corrected. Information on the overscan area is stored in the IRAF compatible format, as values for the `BIASSEC` keyword in the header. Please take into account the observation frame being stored as 16-bit integer values and `BZERO`, which is set in the header. The various error checks and code fragment between `#include` and function `main` can be omitted.

Example solution to Question 3 using SFITSIO

```
fitscc bias_fits, flat_fits, obj1_fits;            /* FITS object */
long y_len;
const char *bsec;
double bsec_mean;
bias_fits.read_stream("bias.fits.gz");             /* Read bias frame */
flat_fits.read_stream("flat.fits.gz");             /* Read flat frame */
obj1_fits.read_stream("obj1.fits");                /* Read raw object frame */
fits_image &obj1pri = obj1_fits.image(0L);         /* Create alias */
obj1pri.convert_type(FITS::FLOAT_T);               /* Convert data type */
mdarray_float &obj1_array = obj1pri.float_array(); /* Alias of array object */
bsec = obj1pri.header("BIASSEC").svalue();         /* Get overscan info */
bsec_mean = md_mean(obj1_array.sectionf(bsec));    /* Get mean of overscan */
obj1_array -= bsec_mean;                           /* Subtract overscan */
obj1_array -= bias_fits.image(0L).data_array();    /* Subtract bias */
obj1_array /= flat_fits.image(0L).data_array();    /* Flat adjustment */
y_len = obj1_array.row_length();                   /* Get height of image */
obj1_array.trimf("*,%ld:%ld", y_len/4, 3*y_len/4); /* Rid top and bottom */
obj1_fits.write_stream("obj1_done.fits");          /* Save */
```

Sample programs using SFITSIO are also shown in §14 APPENDIX1. They will be helpful to learn SFITSIO programming.

1.2 Why is SFITSIO being developed?

Development of SFITSIO started as part of the data analysis project of 'AKARI', the infrared astronomy satellite.

The project defined their standard data format to be 'Time Series Data (TSD)' (Figure1) for

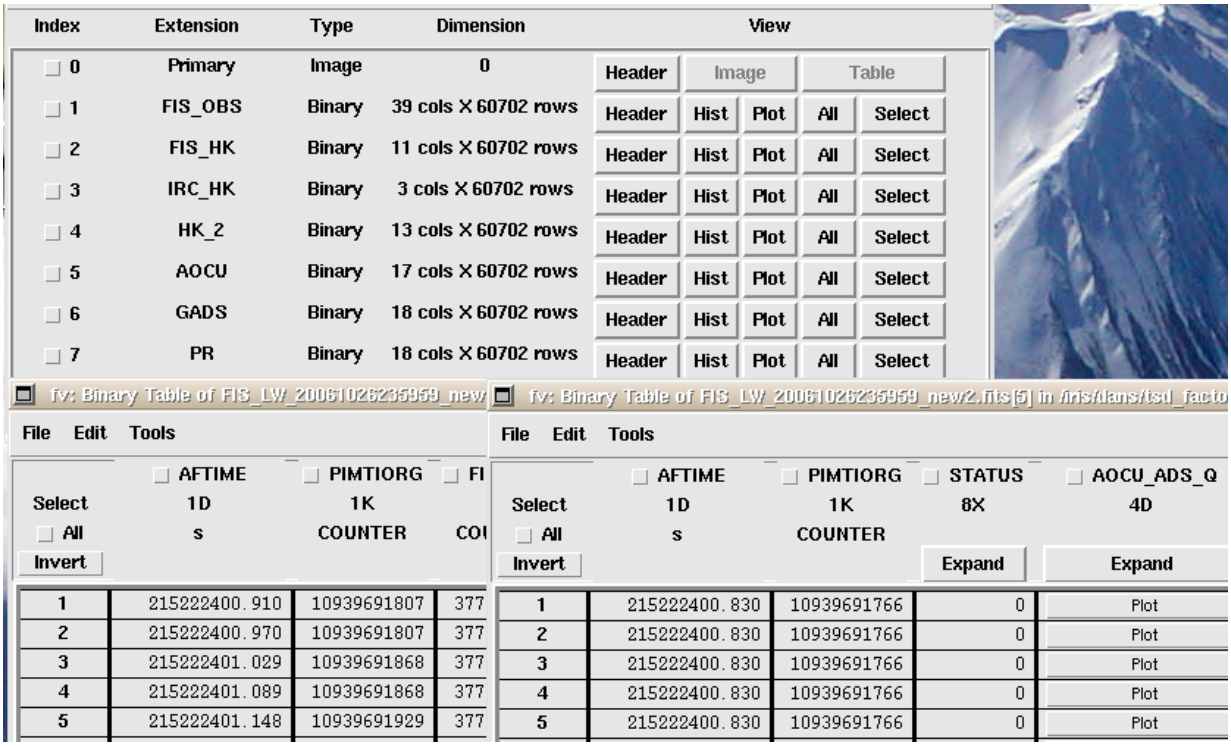


Figure 1: TSD, a standard data format used in software projects for "Akari", the infrared astronomy satellite, can be opened using fv. All tables other than Primary are binary.

their data analysis. TSD includes some binary table extensions in a FITS file, with the detector, satellite, orbital information, etc. being stored in individual binary tables. That then means all the data analysis tools for "Akari", including those for entire sky surveys and various point observations, should use the TSD as the first input file. The problem is, however, that TSD is a very complicated FITS file that is composed of many different types and structures of data and a wide range of data storage methods, as shown in Figure 1. The decision was therefore made within the project to develop a standard library that can access TSD more efficiently, and on the grounds that it would be impractical for each analysis team to write their own code in order to access the TSD.

The standard library developed in the project for the TSD (in C and IDL languages)[1)] loads the entire FITS file into memory[2)], basically with *the API naturally representing the structure of FITS*[3)]. As the result the following code can be used to access the TSD in the case of the C language:

```
tsd = fits_cache__open(class_level, NULL, "tsd_xxx.fits");  /* Read the file */
(omitted: pieces of code used to set hdu_index and col_index)
ptr = tsd->btables[hdu_index]->columns[col_index].data; /* Access the column */
```

The first line loads the FITS file into the memory and the line including "`ptr = ...`" gets the first address of the column data within the table. Please note that the API represents the structure of FITS in a natural manner, as the whole of FITS, the binary table HDUs, and columns are specified in turn after `tsd->`. With an API of this level you can write code that accesses the various binary table HDUs without very much effort, while still ensuring decent readability. The downside is, however, that this library is insecure because it always requires direct access to the memory, as mentioned above, and lacks many of the features required of the generic FITS library, including calculating `TZERO`*n* and `TSCAL`*n*, which were not needed in the project.

SFITSIO is based on ideas found in the C library, and built from scratch using knowledge lerned in its development. It has evolved into an even more sophisticated and more versatile FITS library using C++'s *"class"*, which is *upward compatible with struct*, that has successfully resolved all the problems found in the above-mentioned C library regarding the Binary Table HDUs, and with support for image analysis via an intuitive API for Image HDUs. Refer back at the solution for Question 1 in §1.1. It is immediately possible to write code that accessed the values in binary tables using the names of the tables and columns and without any need for pointer variables. Similarly, you can easily write clear code to do what you want, as revealed in Questions 2 and 3.

The policy used in the "Akari" project that *conceals the FITS I/O as much as possible* should apply to any other project. There is an increasing need for versatile, performant, and high-level FITS I/O libraries that can provide an easy way to accomplish different purposes according to the emergence of higher-performance equipment and larger-scale surveys.

SFITSIO was developed to provide useful solutions to this challenge. Additional solutions will be provided in future versions in thereby keeping up with the development of the state-of-the-art computing environments.

[1)] The TSD-specific interface for IDL language, created by Dr. Hajime Baba, has a very logical design and elegant coding, and which have a significant influence on SFITSIO.

[2)] The rapid popularization of 64-bit Linux, which provides a lot of memory space, made it the perfect match for memory-mapped files.

[3)] Strictly speaking "on-memory" is not essential feature with implementing "an API that represents the structure of FITS." That said, however, on-memory is still a more realistic solution because disk-based implementation would be too expensive.

1.3 Why you should use SFITSIO to minimize any programming frustrations

1.3.1 SFITSIO does not force programmers to learn C++

Because SFITSIO is written in C++ your SFITSIO code gets compiled by a C++ compiler. You may state: "No! I can't use C++!!". Relax. **C++ is upwardly compatible with C so that the code can be written in C**. That is, this code

```
#include <stdio.h>
int main()
{
    printf("Hello World\n");
}
```

can be compiled directly by the C++ compiler.

You don't need to follow any such styles as (`cout << "foo" << endl`)[4] shown in guidebooks on C++. SFITSIO does not require that you follow the C++ style. You can therefore still write the code using your C knowledge and style when you use SFITSIO.

1.3.2 Super-convenient s++ script—Makefile is not required

In order to make executable files from source code with SFITSIO you will need to link it with a few standard libraries. The accompanying s++ script will take care of that and you simply issue the command:

```
$ s++ my_program.cc -lsfitsio
```

to finish compiling it. It can also create a source template file containing `#include` directives and the main function for you if the source file specified in the argument does not exist, thus effectively saving you from having to write the same portion of code for different projects. SFITSIO is very convenient for casual use thanks to its s++ script.

1.3.3 You never have to worry about memory leaks with automated memory management

If you use the C language the memory space allocated using `malloc()` must be eventually freed up using `free()`. If you ever forget to call `free()` a memory leak will occur. A small amount of code enables you to manage any possible memory leaks, but it can be a nightmare identifying any memory leaks that occur in large amounts of code. In that sense, it is just like containment of nuclear fuel in a fission reactor. It is very difficult to correct any memory leaks if you do not have stringent coding rules in place.

In order to minimize the chance of any such problems occurring, SFITSIO automates any memory management related to FITS by taking advantage of C++ "classes."

This basically saves programmers from needing to call `malloc()` and `free()`, and therefore from memory leaks occurring in FITS manipulations[5]. For example, reading a FITS file with SFITSIO involves the following steps:

(1) You instruct SFITSIO to read the FITS file (in this example, `myimage.fits.gz`) using its API.

```
fits.read_stream("myimage.fits.gz");
```

(2) SFITSIO will analyze the FITS file and allocate the required memory space.

[4] The Google C++ Style Guide recommends that programmers use `printf()` instead of the `cout << ...`.

[5] The use of `new` could lead to a memory leak, but is barely required.

(3) SFITSIO will copy all (or part of) the content of the FITS file into the allocated memory.

(4) You can access the data you want (in this example, `CRVAL1` in the header) in the memory using the SFITSIO API.

```
printf("crval1 = %f\n", fits.image("Primary").header("CRVAL1").dvalue());
```

(5) The allocated memory space will be automatically freed up as soon as it falls out of scope.

You only need to write code for (1) and (4) and not worry about any memory leaks.

Leave the memory management to the library and never have to struggle with memory leaks!

1.3.4 The API naturally represents the structure of FITS—you will never forget it once you have used it!

The reason why you will never forget SFITSIO's API once you have used it lies in the *structure of the API*.

So, what does SFITSIO's API look like? Let me explain using the following three examples.

(1) To read and print the value of `CRVAL1` in the header of the Primary HDU:

```
printf("crval1 = %f\n", fits.image("Primary").header("CRVAL1").dvalue());
```

(2) To read and print the pixel value of the coordinates (x, y) of the image in the Primary HDU:

```
printf("pixel = %f\n", fits.image("Primary").dvalue(x,y));
```

(3) To read and print the value of the column "RA" in the first row of the binary table HDU called "CATALOG":

```
printf("ra = %f\n", fits.table("CATALOG").col("RA").dvalue(0));
```

Have you got a feel for how it can be used now?

Yes, that's right. The API *naturally represents the structure of FITS*. For example, in the third example you can read:

FITS that has a table that has a column that has a double value

If you replace "that has a" with "`.`" and place the functions in the same order as FITS is structured you can then use the API. Of course, these structured APIs *immediately* enabling you to write code for random access across multiple HDUs, and as revealed by the solution to §1.1.

Moreover, SFITSIO has a very limited number of functions that need to be memorized. There are only five functions that are essential as structural members:

`hdu()`	`image()`	`header()`	`table()`	`col()`
Any HDU	Image HDU	FITS header	ASCII/Binary Table HDU	column in table

In addition, all you have to do is memorize the function names to read the values: `dvalue()`, `lvalue()`, `llvalue()`, `svalue()` (which return double, long, long long, and string respectively) and `assign()`, a function to write values, in order to be able to do all the basic read and write operations of FITS.

Please note that `dvalue()` and similar functions would return a value appropriately converted from whatever was in the FITS file, were it a string or an integer. Conversely, `assign()` would convert the value into what is appropriate for storage in FITS, depending on the type of the

argument[6]. Moreover, these functions are capable of converting values based on keywords `BZERO`, `BSCALE`, `TZERO`n, and `TSCAL`n in the header, which are cached on the library side, thus saving us from writing code to parse the header. Of course you can read/write *unsigned integer values* simply with `lvalue()`/`assign()` and setting `BZERO` or `TZERO`n appropriately.

1.3.5 You can select safety- or performance-oriented APIs in your situations

Functions mentioned in §1.3.4 such as `dvalue()` and `assign()` are called "high level API". These functions convert types and values following FITS standard, and is suitable for safety and low-cost programming. Although great care has been taken in implementing these functions in SFITSIO in order not to degrade the performance, the overhead of function calls can accumulate in the case of small amounts of operations per pixel performed with a large number of pixels. If you are not comfortable with the performance of the high-level API you can use the "low level API" without processing ZERO, SCALE, or NULL, or even the "ultra-low level API" which can access the addresses of internal memory buffers managed by SFITSIO and SLLIB.

Recently, large-sized FITS images are quite common, and higher performance of analysis is required. In such a situation, you can use APIs provided by SLLIB's array object (mdarray_float class, etc.) to obtain higher performance on average, since the implementation of SLLIB's APIs is simpler than that of SFITSIO. SLLIB is a basic library for scientific research to process streams, n-dimensional arrays, etc., which is independent of FITS. Therefore, your analysis tools developed with SLLIB are easily applicable to non-FITS image data. Both SFITSIO and SLLIB provide various APIs for image processing, which enable a small amount of code to perform area scanning, copy and paste, the four basic arithmetic operations with images and scalar values, and image scanning and operations with statistical functions. However, only SLLIB has some features such as mathematical functions for arrays and operators for arrays, and new packages (header files) for data analysis will be appended to SLLIB in the future. Therefore, we recommend that you to write SLLIB-based code for full-scale image analysis tools.

You can thus creatively use safety- or performance-oriented APIs in more cost-effective software development.

1.3.6 Functions support variable-length arguments in a printf()-compatible manner

The specification of a format string plus variable-length arguments is one of the most powerful and useful features in LIBC string operations. SFITSIO provides a wide range of functions with the suffix "`f`", such as `colf()` and`headerf()`, which can take arguments fully compatible with printf() function. You can creatively use them to reduce your coding efforts. For example, you can use `headerf()` to write code:

```
fits.image("Primary").headerf("CRVAL%d",i).dvalue()
```

where `i` is a variable of the int type.

1.3.7 Transparent handling of compressed files and files on Web/ftp servers

SFITSIO automatically compresses/expands files in the gzip or bzip2 formats on local storage or Web/FTP servers, and without requiring the use of any special API:

```
    sz = fits.read_stream("http://www.xxx.jp/fits_data/foo.fits.bz2");
```

Whereas the gzip format is supported in other FITS I/O libraries as a matter of course, SFITSIO can handle any compression format by simply installing the corresponding command, for example fpack.

[6] C++ enables you to declare multiple functions with the same function name but with different argument(s).

1.3.8 Partial read of a FITS file using IRAF/CFITSIO-like expression

We might want to load a part of a FITS file in some situations, e.g., its size is very large. SFITSIO allows partial read of n-dimensional images and tables in a FITS file using IRAF-like expression:

```
sz = fits.read_stream("image.fits.gz[1:100,*]");
```

This example uses "`[...]`" expression that indicates 1-indexed positions of pixels following IRAF/CFITSIO. Using "`(...)`" is also allowed to indicate 0-indexed, i.e., '`(0:99,*)`" is equivalent to expression in above example.

In addition, SFITSIO supports original syntax to select multiple HDUs. Next example code will read a part of image in HDU No.0 and all contents of HDU No.2 (other HDUs will be skipped).

```
sz = fits.read_stream("mixed.fits.gz[0[1:100,*];2]");
```

1.3.9 Comment dictionary for the FITS header and FITS template function

SFITSIO has a comment dictionary for keywords defined using the WCS and FITS conventions and those commonly used. It saves programmers from writing comments for each keyword in the FITS header. For example:

```
fits.image("Primary").header_fill_blank_comments();
```

This instantly fills all uncommented header records in the Primary HDU with the corresponding comments from the dictionary.

In addition, SFITSIO supports template files (§9) in the same manner as CFITSIO. A template file is a text file that can be used to define the content of FITS with somewhat lenient syntax like FITS headers, which can be used to create a new FITS file (object) without any data. This enables you to easily develop generic tools. Also in the case of FITS created from templates any uncommented portions of the template will automatically be populated with comments, thereby enabling template users to write just the bare minimum.

Needless to say, if programmers do find the content of the available dictionaly in SFITSIO to be unsatisfactory they can append/modify its content.

1.3.10 CFITSIO-like, disk-based FITS I/O is also supported

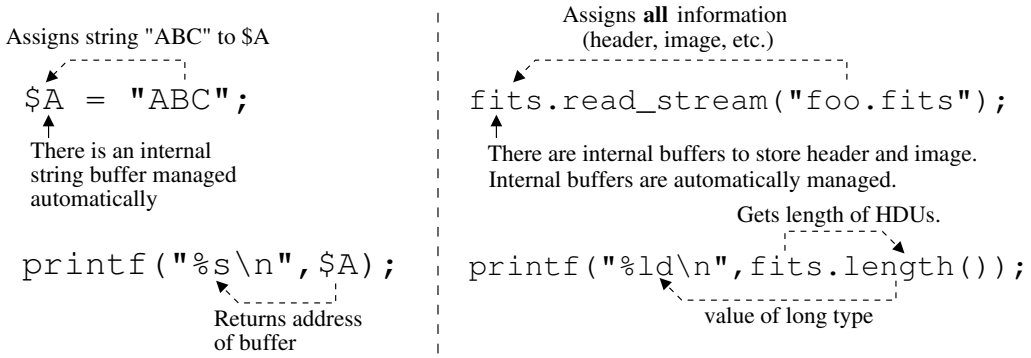
It might not be a good idea to load the entire FITS file into memory if you want, for example:

- to quickly scan the header only, thus minimizing memory usage, or
- to calculate the median of a number of image files.

SFITSIO provides an API to use to load a single Header Unit into memory or to skip reading one Data Unit. In order to use this API the programmer starts off by declaring the stream handler with which the FITS file is opened. Next, the stream handler is given to the target member function in order for SFITSIO to scan the necessary amount of FITS files. For example, if you use one of the member functions to load the header the stream is placed at the start point of the consecutive Data Unit. In this case, header-related high level API functions are still available and something like disk-based image access would be a relatively simple task. Of course, you can also elaborate on disk access as much as you want in the case of a seekable stream.

If you are interested in disk-based FITS I/O, see also tools/hv.cc in SFITSIO source package.

SFITSIO's memory managements close to scripting languages



The basic idea of SFITSIO is that the whole FITS file can be treated as strings in scripting languages (e.g. perl). According to that idea we developed SFITSIO in order to minimize the programmers' load with the I/O of FITS as much as possible. Substitution of strings into variables written in scripting languages is frequently used. In fact, the script engine calculates, allocates, and manages the memory space behind it. However, these are not visible to the programmers.

On the other hand, and in the case of the conventional and procedural-type FITS-I/O libraries including CFITSIO, we have dealt with the bothersome tasks of calculating memory space for the header strings for data areas, allocating the area using functions such as `malloc()`, and freeing up any unnecessary space using `free()`.

Please think about this a little bit. If implementation of variables in scripting languages applies to the I/O of FITS it would be basically impossible for it to cause the same bother as with using the conventional FITS-I/O, wouldn't it?

Now, "Class" in the C++ language. If we use "Class" smartly it is possible to box in the whole FITS file into one variable (called an object because of its size). Figure compares the case of perl with that of SFITSIO. In the case of perl the string `"ABC"` is substituted as a variable while the whole FITS file `"foo.fits"` is substituted for by another variable (object) named "fits" in SFITSIO. In both the cases of perl and SFITSIO the necessary memory space gets automatically be allocated and managed. The FITS files include several types of information on the headers and image data so that the getting/putting of the information from/into the variables (objects) need to use functions (so called member functions). Although the variables have to be more manually manipulated than with script languages it is clear that FITS files can be treated in an analogous manner to variables in perl.

In this way using the idea of classes in C++ frees programmers from the bother of memory management, and provides intelligible APIs that reduce the load on programmers as much as possible.

1.4 FAQ

- What about support for users?

  SFITSIO and SLLIB is "officially supported software" of the ISAS/JAXA Data Utilization

Group and was jointly developed and maintained by the Data Utilization Group and myself (Yamauchi). Formally, however, it is open source software provided under the MIT license and has no warranty whatsoever.

- Has it been used in any actual projects?

  It has been used internally by JAXA an NAOJ to my knowledge.

  It has been used in archive development projects across almost all areas, including the entire sky mapping project, FITS-related tool development in infrared and moon/planetary science areas, and the L1TSD project for ASTRO-H.

  ASTRO-H project team, Sprint-A project team, Subaru HSC project team and some people in NAOJ have been started to use SFITSIO/SLLIB to evaluate them. In addition, ASTRO-H project team have developed new FITS interface for Ruby "RubyFits" which is based on SFITSIO.

- It is a C++ library for use by experts only, isn't it?

  Definitely not! On the contrary it is more useful for beginners as they can easily perform traditionally difficult operations.

  SFITSIO incorporates carefully selected truly useful C++ features rather than C++-proprietary features that can confuse users from the astronomy industry with average skills. In addition, this manual was written for those who have experience with the C language only. Experience with C++ is, therefore, not required to use SFITSIO.

- How fast is it?

  We have optimized SFITSIO and SLLIB with steady approach such as removing 'if' phrase and function pointers in any loops using inline attribute and macro, etc., and applying high-speed algorithm of transpose, calculating *true* median, etc. In addition, fast codes applying SIMD instructions are used for memory initialization, copying memory, flipping image, swapping byte order, etc. Therefore, SFITSIO and SLLIB have attained to a high-level quality about basic performance. Intel® C++ Compiler can be used for building SFITSIO and SLLIB, which brings drastic improvement of performance about mathematical functions.

  To obtain high performance in your programs, SFITSIO and SLLIB have some levels of APIs, i.e., high-level APIs suitable for safety and low-cost programming and lower-level APIs for higher performance. You can select some of them in your situation.

- Can I use 16-bit unsigned integers?

  Yes. The high level API and image-processing API supports `BZERO` conversions. In the case of these APIs no value conversions have to be dealt with by the programmers. To save a 16-bit unsigned integer into a new FITS all you have to do is set `BZERO` to 32768 (refer to §5.9).

  If you are uncomfortable with the high-level API for performance reasons you may find the convert_type() member function (refer to §13.6.13 and §13.8.28) more user friendly when used in combination with the ultra-low level API. Please refer to the code example in §5.11.

2 Installation and Getting Started with SFITSIO

2.1 Supported OS

SFITSIO supports both 32 and 64 bit version of Linux, FreeBSD, MacOSX, Solaris, and Cygwin.

SFITSIO requires GCC g++ version 3 later (the author has verified operations on g++ 3.3.2 or later.)

2.2 Building and installing SFITSIO

To build SFITSIO you will need SLLIB (Script-Like C-language library) version 1.4.2 or later. Install the libraries of zlib, bzlib, and readline (probably named zlib-devel, bzip2-devel, and readline-devel as RPM)[7] that are required by SLLIB.

If you haven't installed SLLIB yet install it in the order below. Expand the archive file of SLLIB and run make.[8]

```
$ gzip -dc sllib-x.xx.tar.gz | tar xvf -
$ cd sllib-x.xx
$ make
```

Compilation of the 32 or 64-bit versions is possible by adding options to the compiler, for example:

```
$ make CCFLAGS="-m64"
```

If you use 32-bit OS, gcc might not turn SSE2 on by default[9]. You can append options for SSE2 to improve performance as follows:

```
$ make CCFLAGS="-msse2 -mfpmath=sse"
```

Install SLLIB.

```
$ su
# make install32
```

With the 64-bit OS run `make install64` instead. The default installation directory of `libsllib.a` is `/usr/local/lib` and `/usr/local/lib64` (or `/usr/local/lib/64` with Solaris) in the case of install32 and install64, respectively. At the same time all header files are copied to `/usr/local/include/sli` and the wrapper script of `g++`, named `s++` is installed to `/usr/local/bin`.

Next, install SFITSIO in the same way. Expand the archive file of SFITSIO and run make.[10]

```
$ gzip -dc sfitsio-x.xx.tar.gz | tar xvf -
$ cd sfitsio-x.xx
$ make
```

Analogous to the compilation of SLLIB, you can append options to the compiler, e.g.

```
$ make CCFLAGS="-m64"
```

```
$ make CCFLAGS="-msse2 -mfpmath=sse"
```

If any errors get reported on running make, specify the directory to which the header files of SLLIB were installed for INCDIR in Makefile.

Install SFITSIO.

[7] With Debian they will be zlib1g-dev, libbz2-dev, and libreadline5-dev.

[8] Shared library can be made by `make shared` by advance users.

[9] It is enabled by default on 64-bit OS.

[10] Shared library can be made by `make shared` by advance users.

```
$ su
# make install32
```

With the 64-bit OS run `make install64` instead. The default installation directory of `libsfitsio.a` is `/usr/local/lib` and `/usr/local/lib64` (or `/usr/local/lib/64` with Solaris) in the case of install32 and install64, respectively. At the same time, all the header files are copied to `/usr/local/include/sli`.

That's all the installation takes.

2.3 Installing multi-threaded compression utilities (optional)

SFITSIO-1.2.0 (or later) automatically detects "pigz" (`http://zlib.net/pigz/`) and "lbzip2" (`http://lacos.hu/`) installed in your environment, and performs multi-threaded compression and decompression when writing and reading local files of the gzip or bzip2 formats. A multi-core CPU improves the performance of compression and decompression (**performance of compression gets drastically accelerated**). Install[11] pigz or lbzip2 if you want to enable this feature.

2.4 Operational checks using sample programs

The following code is an SFITSIO version of the 'Example Program' found in the second chapter of the user's guide of CFITSIO [12].

```
#include <stdio.h>
#include <sli/fitscc.h>        /* required by every source that uses SFITSIO */
using namespace sli;

int main( int argc, char *argv[] )
{
    fitscc fits;                  /* fitscc object that expresses a FITS file */
    const long width = 300, height = 300;                    /* size of image */
    long len_written, i, j;

    /* Create the Primary HDU */
    fits.append_image("Primary",0, FITS::SHORT_T, width,height);
    fits_image &pri = fits.image("Primary");

    /* Add an "EXPOSURE" header record */
    pri.header("EXPOSURE").assign(1500.).assign_comment("Total Exposure Time");

    /* Set the values in the image with a linear ramp function */
    for ( j=0 ; j < height ; j++ ) {
        for ( i=0 ; i < width ; i++ ) pri.assign(i + j, i,j);
    }

    /* Output a FITS file */
    len_written = fits.write_stream("testfile.fits");
    printf("saved %ld bytes.\n", len_written);

    return (len_written < 0 ? -1 : 0);
}
```

Now, compile it with the `s++` command.

[11] This can be done before or after the SFITSIO installation.

[12] You may notice the code is more readable than the CFITSIO sample code.

```
$ s++ fitsout.cc -lsfitsio
g++ -I/usr/local/include -L/usr/local/lib -Wall -O fitsout.cc -o fitsout -lsfits
io -lsllib -lz -lbz2 -lreadline -lcurses
$ ./fitsout
```

2.5 Getting started with SFITSIO

- *For those who are in a hurry or just want to experiment:*

  Proceed to §5 'Tutorial' after browsing §4.7 'Member functions classified by purpose and API level'.

- *For FITS experts:*

  Proceed to §4 'Representation of FITS data structures and APIs with SFITSIO's class structure'

- *For those who have little knowledge of FITS and want to use this opportunity to learn it:*

  Carefully read §3 'Review of FITS data structures' and proceed to §4 and §5.

3 Review of FITS data structures

3.1 Overall structure

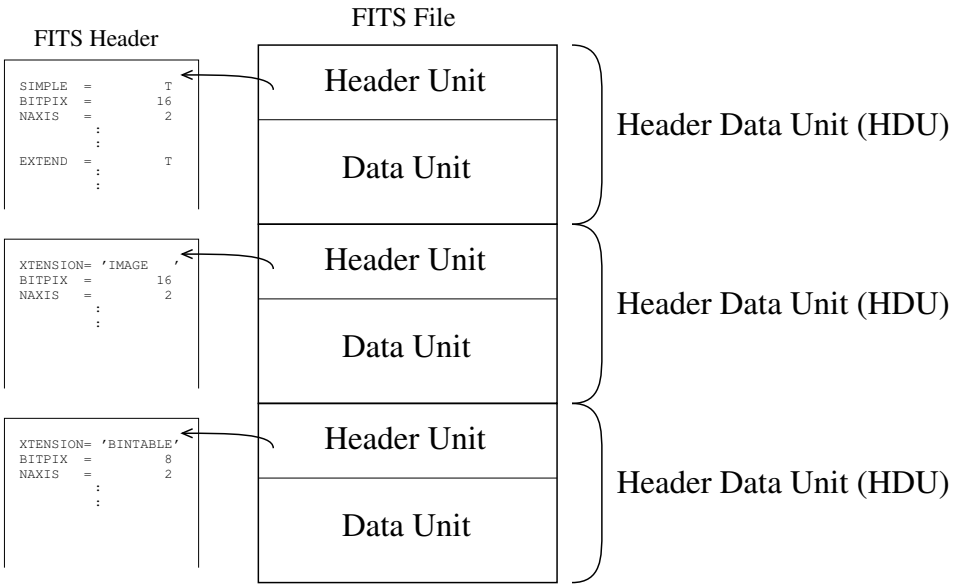


Figure 2: Overall structure of FITS file

The overall structure of a FITS file is a combination of multiple "HDUs (Header Data Units)", that are comprised of a pair of "Header Unit" and "Data Unit", as shown in Figure2. The Data Unit is comprised of ASCII or binary data such as an image or a table, while the Header Unit contains various information on the data and specifications of the Data Unit stored as a set of "keyword = value" records in the plain text format of 80 characters and without linefeed characters. The byte length of each Header Unit or Data Unit needs to be equivalent of the integral multiple of 2880. The blank space at the end of each Unit arising from the convention used is usually padded with blank spaces (0x20) in the case of a Header Unit, or a NUL character (0x00) in the case of a Data Unit[13)].

The three HDU formats that fulfill the FITS convention are: "Image"(§3.3), "ASCII Table"(§3.4), and "Binary Table"(§3.5). The first HDU in a FITS file is called the "Primary HDU", and which for historic reasons should be an Image format. The number of HDUs can be identified as one if the `EXTEND` keyword has a value of `F`, but the entire FITS file need to be parsed to determine the number of HDUs if `EXTEND` equals `T`.

The binary data in Data Units are stored in the big-endian format with both integer and floating-point (IEEE 754) values. Please note that no data alignment can be specified and in some cases you will have no proper memory access even if you have copied the entire content of the Data Unit into memory and converted the different types of endian orders, although that does depend on your HDU and data processing system.

3.2 Header Unit

The content of Header Units of a FITS file, which will be shown in the sections starting from §3.3 in more detail, is somewhat old-fashioned: a sequence of records in a format "`keyword = value / comment`" starts from the beginning, ends with the `END` keyword, but lacks the mechanism to

[13)] A significant number of FITS with `EXTEND` being `F` that are not padded in integral multiples of 2880 bytes. SFITSIO does not treat this as an error, although the author is not sure whether this is a valid convention or not.

differentiate between the value types as the INI file in Windows does. A record consists of printable 80 characters and can be of either of four different record formats:

(1) format in "`keyword = value / comment`",
(2) format in "`keyword␣arbitrary_string`",
(3) format in "`␣␣␣␣␣␣␣␣arbitrary_string`", or
(4) the "`END`" keyword.

It is recommended that the reading or writing of FITS file should not modify the order of these records, although there is no guarantee that the order will be preserved. The keyword is limited to a length of eight characters and can only be composed of capital letters, digits, "`-`", and "`_`".

The letter in the ninth position is `=` in the case of (1) and a blank space in the case of (2). There is no convention of an alphanumeric character being in the ninth position[14]. The value is either an integer, a real number, a logical value (`T` or `F`), or a string, and should be quoted only if it is a string. Characters after `/` can be omitted.

In the case of (2) the FITS convention only allows for "`COMMENT` and "`HISTORY`", but in practice non-conventional keywords such as "`CONTINUE`" and "`HIERARCH`" are also commonly used. You therefore cannot determine whether the format is (1) or (2) based on only the keywords.

The current FITS convention does not specify how very long string values should be stored, although the convention of the `CONTINUE` keyword is commonly used (refer to §10.1).

3.3 Image HDU

Image HDU is a simple array of n-dimensional image data that is packed into a Data Unit. The supported types of data are: 8-bit unsigned integers, 16-bit, 32-bit, and 64-bit signed integers, and 32-bit and 64-bit floating point numbers. However, multiple data types cannot coexist within a single Data Unit[15].

It can be placed either in the Primary HDU or in and after the second HDU; the first part of the FITS header differs in these cases, as show below:

```
SIMPLE  =                    T / conformity to FITS standard
BITPIX  =                   32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 1024 / length of data axis 1
NAXIS2  =                 1024 / length of data axis 2
EXTEND  =                    T / possibility of presence of extensions
```

```
XTENSION= 'IMAGE   '           / type of extension
BITPIX  =                   32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                 1024 / length of data axis 1
NAXIS2  =                 1024 / length of data axis 2
PCOUNT  =                    0 / number of parameters per group
GCOUNT  =                    1 / number of groups
```

In and after the second HDU the `XTENSION` must be placed at the beginning of the header, which then determines the HDU type.

Different data types are specified by `BITPIX` (the number of bits), with negative `BITPIX` meaning the floating point type. The size of the data array is specified by `NAXIS`n and the data can have

[14] This provides for the possibility of keywords being used of nine characters or more, while still ensuring compatibility with traditional data.

[15] This then means that copying an entire Data Unit to an area allocated by malloc() and then converting between the different endian orders will result in memory access without any alignment problems.

an arbitrary number of dimensions. 2D data is most commonly used, but occasionally 3D or 4D. In the case of 3D data they may represent the R, G, and B components in that order. This is not specified in the FITS conventions but is commonly called the "RGB Fits Cube" and can be handled as RGB data using ds9 etc.

The physical value of the pixel to be read can be determined using `BZERO` and `BSCALE` values in the header:

$$\text{Physical Value} = \text{BZERO value} + \text{value of pixel} \times \text{BSCALE value}$$

where `BZERO` and `BSCALE` will be 0.0 and 1.0, respectively, if they do not exist. `BZERO` being set to 32768 enables 16-bit unsigned integers to be conveniently stored in a Data Unit using `BITPIX=16`.

In the case of integral data types the value for data array elements to be considered undefined pixels[16] can be determined using `BLANK` in the header. NaN is treated as a value for undefined pixels in the case of floating point data types.

Although there is no FITS convention for the origin and the axes when 2D image data is plotted it is naturally considered that the origin will be at the lower left corner and the axis for the first dimension oriented right and the axis for the second vertically.

With regard to WCS the projection method and reference points, etc. can be recorded in the header using keywords such as `CTYPE`*n*, `CRPIX`*n* and `CRVAL`*n*. Because the FITS conventions specify very basic keywords those proposed by the so-called WCS Paper series are considered de facto standards[17] for the WCS conventions, and which cover a wider range of keywords. Please refer to the original documentation and the FITS guide issued by the National Astronomical Observatory of Japan.

If you do use WCS it should be noted that there is no conventional agreement as to the following:

- whether the pixel coordinate of the origin at the lower-left corner is one or zero, and
- whether the integral value of the pixel coordinate refers to the center or the left edge of the pixel.

Generally with FITS the pixel coordinate of the origin at the lower-left corner is one, and the integral value of the pixel coordinate refers to the center of the pixel. Ds9 and fv also follow these agreements, so there is really no advantage in using any different definitions. However, please do note that the author of a particular FITS file may have different ideas with regard to these definitions.

3.4 ASCII Table HDU

The ASCII Table HDU for a single table simply consists of printable characters. The content of the table is stored in the Data Unit. The byte data of the Data Unit is packed from the first column in the first row without any empty spaces on a "line-by-line" basis, which means the byte data starts in the first column of the first row of the table, followed by the second column of the first row, the third column, and so on and without any empty spaces.

The content of the Header Unit consists of the required keywords for the 2D data with `BITPIX=8` in the Image HDU plus keywords that describe the table. This means the Data Unit can simply be skipped without needing to be aware of any ASCII Table HDU. Just like the Binary Table HDU the ASCII Table HDU cannot be the Primary HDU. The header always therefore starts with `XTENSION`, as shown below:

[16] The raw value of any data array elements, without taking `BZERO` or `BSCALE` into account.

[17] There are also other totally different conventions to the WCS Paper series, for example the DSS FITS. These special WCS conventions can be used with popular available software, including WCSTools.

```
XTENSION= 'TABLE   '           / type of extension
BITPIX  =                    8 / number of bits per data element
NAXIS   =                    2 / number of data axes
NAXIS1  =                   21 / width of table in bytes
NAXIS2  =                  256 / number of rows in table
PCOUNT  =                    0 / number of parameters per group
GCOUNT  =                    1 / number of groups
TFIELDS =                    2 / number of fields in each row
TTYPE1  = 'Freq    '           / field name
TBCOL1  =                    1 / starting position in bytes
TFORM1  = 'I10     '           / display format
TTYPE2  = 'L10Pointer'         / field name
TBCOL2  =                   12 / starting position in bytes
TFORM2  = 'I10     '           / display format
EXTNAME = 'HTMLevel10Idx'      / name of this HDU
END
```

The keywords from `XTENSION` to `TFIELDS` are arranged in a predefined order, of which `BITPIX`, `PCOUNT`, and `GCOUNT` have the fixed values 8, 0, 1, respectively.

Header keywords in an ASCII Table have meanings shown below. By the way, the column in a table is called a "field" with FITS, although the term "column" is used here for the sake of consistency.

keyword	meaning
`NAXIS1`	the width (number of bytes) of the table
`NAXIS2`	the height (number of bytes) of the table
`TFIELDS`	the number of columns in the table
`TTYPE`n	the name of the table column n
`TUNIT`n	the physical unit of the value of the table column n
`TFORM`n	the data format for the table column n in the ANSI FORTRAN-77 format: ("`A`w") for a string, ("`I`w") for a decimal integer, or ("`F`$w.d$", "`E`$w.d$", or "`D`$w.d$") for a real number
`TBCOL`n	the byte-wise start point (1-indexed) for the column n in a specific row of the table
`TZERO`n	the zero point applicable to the table column n
`TSCAL`n	the scaling factor applicable to the table column n
`TNULL`n	the string to be used for any undefined values in the table column n

Where `TZERO`n and `TSCAL`n fulfill the same role as `BZERO` and `BSCALE` in the Image HDU, although rarely used.

The ASCII Table HDU has a simple structure and can be created relatively easy with low level FITS libraries. However, that said, shortcomings of the ASCII Table include that it is impossible to extract the data width of a column from a single header record, and therefore it does not work well with FITS templates (§9).

3.5 Binary Table HDU

The format of the Binary Table HDU can be considered to be a binary version of the aforementioned ASCII Table HDU, however, with more data types and various extension features, thus making it rather a complex and not very suitable for the word "SIMPLE" to be used with it. It has as just much flexibility as complexity, however, being able to meet very high-level requirements such as storing complex and mysterious telemetry data from astronimy satellites.

One of the biggest differences between the Binary Table HDU and the Image HDU or the ASCII Table HDU is the existence of a Reserved Area and Heap Area in addition to the Data

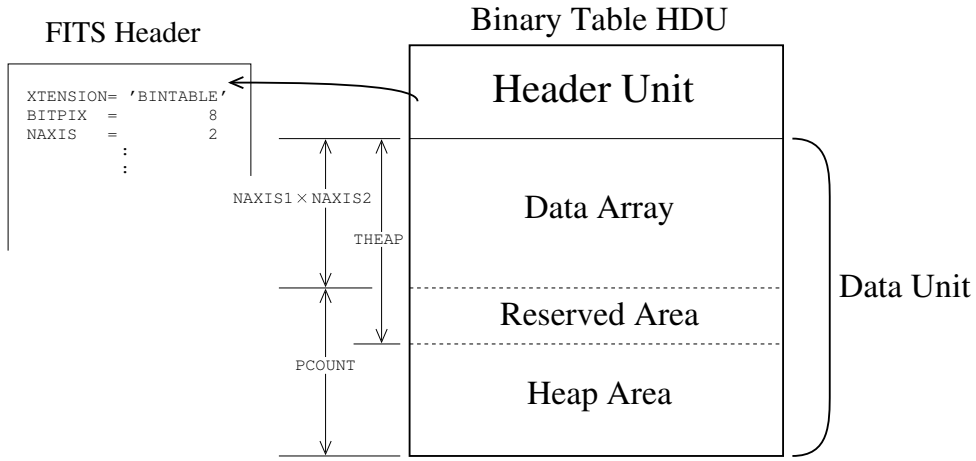


Figure 3: The structure of the Binary Table HDU. The size of each Data Unit can be determined from the values of `NAXIS1`, `NAXIS2`, `PCOUNT`, and `THEAP`. If there is no Reserved Area `NAXIS1` × `NAXIS2` will equal to `THEAP` and `PCOUNT` the size of the Heap Area.

Array, as shown in Figure 3. A table is stored in the Data Array on a "line-by-line" basis in the same manner as the ASCII Table HDU, and packed with binary data without any empty spaces. The Reserved Area appears to have been made available for disk-based FITS applications and no data can be stored in this area[18]. The Heap Area is where real data for a so-called "variable length array" is stored. To skip over the Data Unit you should take into account the fact that the Data Unit is padded in integral multiples of 2880 bytes, as well as the data length, i.e. NAXIS1 × NAXIS2 + PCOUNT.

Here is an example header:

```
XTENSION= 'BINTABLE'           / type of extension
BITPIX  =                    8 / number of bits per data element
NAXIS   =                    2 / number of data axes
NAXIS1  =                   22 / width of table in bytes
NAXIS2  =                 4096 / number of rows in table
PCOUNT  =             12897220 / length of reserved area and heap
GCOUNT  =                    1 / number of groups
TFIELDS =                    6 / number of fields in each row
TTYPE1  = 'ENERG_LO'           / field name
TFORM1  = 'E       '           / data format : 4-byte REAL
TUNIT1  = 'keV     '           / physical unit
(omitted)
TTYPE5  = 'N_CHAN  '           / field name
TFORM5  = '1I      '           / data format : 2-byte INTEGER
TTYPE6  = 'MATRIX  '           / field name
TFORM6  = '1PE(3353)'          / data format : variable length of 4-byte REAL
EXTNAME = 'MATRIX  '           / name of this HDU
```

Just like the ASCII Table HDU the Binary Table HDU cannot be the Primary HDU, the header always starts with `XTENSION`, and the keywords up to `TFIELDS` are in a predefined order. The keywords `BITPIX` and `GCOUNT` have the fixed values 8 and 1, respectively. In this example you can see there is a Heap Area but no Reserved Area from the fact that `PCOUNT` is non-zero and `THEAP` is not defined.

Header keywords found in the Binary Table HDU and their meaning are shown below. I will point out some of the idiosyncrasies when compared to run-of-the-mill tables.

[18] The author has never seen a FITS file that had a Reserved Area.

keyword	meaning
`NAXIS1`	the width (number of bytes) of the table
`NAXIS2`	the height (number of bytes) of the table
`TFIELDS`	the number of columns in the table
`TTYPE`n	the name of the table column n
`TUNIT`n	the physical unit of the value of the table column n
`TDISP`n	the display format of the table column n in the FORTRAN-77 format: ("`A`w") for a string, ("`L`w") for a logical value, ("`I`w") for a decimal integer, ("`F`$w.d$", "`E`$w.d$", "`G`$w.d$", or "`D`$w.d$") for a real number, ("`B`w") for a binary integer, ("`O`w") for an octal integer, or ("`Z`w") for a hexadecimal integer
`TFORM`n	the data type for the table column n;

value of `TFORM`n	type	byte length of the column
r`L`	logical value (`'F'` or `'T'`)	$1 \times r$
r`X`	bit	$\lfloor (r-1)/8 \rfloor + 1$
r`B`	8-bit unsigned integer	$1 \times r$
r`I`	16-bit signed integer	$2 \times r$
r`J`	32-bit signed integer	$4 \times r$
r`K`	64-bit signed integer	$8 \times r$
r`A`a	character (string)	r
r`E`	single-precision floating number	$4 \times r$
r`D`	double-precision floating number	$8 \times r$
r`C`	single-precision complex number	$4 \times 2 \times r$
r`M`	double precision complex number	$8 \times 2 \times r$
r`P`t`(`$e_{\max}$`)`	32-bit array descriptor (array length, heap offset)	$4 \times 2 \times r$
r`Q`t`(`$e_{\max}$`)`	64-bit array descriptor (array length, heap offset)	$8 \times 2 \times r$

r is the number of cells that can be omitted if it is one. a is the length of the string per element when defining a string array (refer to §10.2). t is the type used when the Heap Area is being referred to. $e_{\max}$ is the maximum number of references to the Heap Area per row.

`TDIM`n	the definition of a multi-dimensional array in the table column n, which differs from r in `TFORM`n in that it does not affect the byte length of the column but is used in interpreting the data. It can be specified in the case of r being more than 1 with the form of `(`l`,`m`)`.
`TNULL`n	the value to be used for any undefined values in the table column n
`TZERO`n	the zero point applicable to the table column n
`TSCAL`n	the scaling factor applicable to the table column n

With Binary Table HDUs understanding the data types using `TFORM`n is the most important. The most important feature with this data type definition is the fact that with *a single column can have multiple cells*, which is called a fixed-length array in the case of r more than 1 in the value of `TFORM`n. For example, `TFORM99 = '48I'` defines 48 16-bit integers in a single column. This may be slightly counterintuitive, but it helps to think about the table having some depth and 48 cells being lined along the depth-ward axis. In addition, if you define `TDIM99 = '(8,6)'`, the 48 cells are interpreted to be a 8×6 two-dimensional array, and which is called a multi-dimensional array.

Probably the hardest part about understanding data type definitions is the so-called "variable-length array", which is the case where `TFORM`n is r`P`t or r`Q`t[19]. In this case it helps to think about cells being in line with the depth-ward axis and the number not being fixed, however the data will

[19] If r were more than 1 in r`P`t or r`Q`t, that would result in multiple variable-length arrays in a single column; although I myself have never seen any such FITS file with that definition. Most tools do not support any such definitions either.

be stored in a very special manner. In the case of a variable-length array the real data of the array will be stored somewhere in the Heap Area, and the "index to the Heap Area (offset)" and the "array length" of the row stored in the cell of the table itself (the data array in the Data Unit). A few pages prior we came across the definition `TFORM6 = '1PE(3353)'` in an example of a header, and which means the maximum number of the elements of the array is 3353 and that 32-bit floating point numbers (with the symbol "`E`") are stored somewhere in the Heap Area. For example, if the pair of numbers (12, 34) that are the array descriptor, are stored in a certain row of the column concerned in the table itself it means that 12 32-bit floating point numbers get stored starting from the 34 byte offset from the beginning of the Heap Area. It should be noted that *multiple array descriptors may reference the same address in the Heap Area*, regardless of the columns or rows in the main table[20)]. In other words, this means you cannot convert between different endian orders unconditionally in any Heap Area copied into memory. There are no alignment conventions either, of course, and care should be taken in reading or writing to the memory. Moreover, when defining any high-level API in the FITS library references to the Heap Area should be normalized in some way for the sake of automated processing.

Definition of the NULL value is the value assigned to `TNULL`*n* in the header for integral types, NaN for real number types, and the character code 0 in the case of the logical type. The NULL value for the string type should be determined by the author.

At least conventionally more detailed format descriptors can be used for `TDISP`*n* than those described in the table above, but few tools or libraries support all of them.

`TZERO`*n* and `TSCAL`*n* plays the same role as `BZERO` and `BSCALE` in the Image HDU. They are commonly used to store unsigned integers, as with images.

Keywords used in the Binary Table HDU are often extended locally to represent column information. Well-known ones include `TLMIN`*n*, `TLMAX`*n*, `TDMIN`*n*, and `TDMAX`*n*; SFITSIO also supports `TALAS`*n* (alias definition for a column) and `TELEM`*n* (definition of element name in a fixed-length array), and which were proposed during the "Akari" project of ISAS/JAXA (§11.8, §11.9). But any such locally extended keywords will have the problem of where deleting or moving a column will not result in the expected header because some libraries cannot determine whether they concern column information or not. Unfortunately, there is no agreement with the FITS standards in this regard. SFITSIO proposes a solution for this problem by listing locally extended keywords as values of `TXFLDKWD`, and which was devised during the L1TSD project of ISAS/JAXA (§11.7).

[20)] This would have some advantages in the case of any repetition of the same data sequence, although it would be too difficult to implement with memory-mapped or disk-based systems.

4 Representation of FITS data structure using SFITSIO class structure and API

We have now reviewed the FITS structure in some detail via §3. The reason why this manual contains these explanations of FITS is the fact that "the API of SFITSIO naturally represents the structure of FITS", as mentioned in §1.3.4.

¿From the perspective of developers, designing any such API is basically the same as designing class structures. An important task in any such design is *to find a way to naturally "copy"* the target structure (in this case, FITS) to the class structure. Conversely, it is *important that users of the library recognize the way the structure is "copied" in order to understand the API.*

In this section we will cursorily review the way the FITS structure is "copied" to C++ the class structure, as well as the corresponding API. Although this manual can be used without that knowledge, I believe that you can learn the API easier if you do possess knowledge on it, at least to some extent.

4.1 Overview of class structure and access to HDU via API

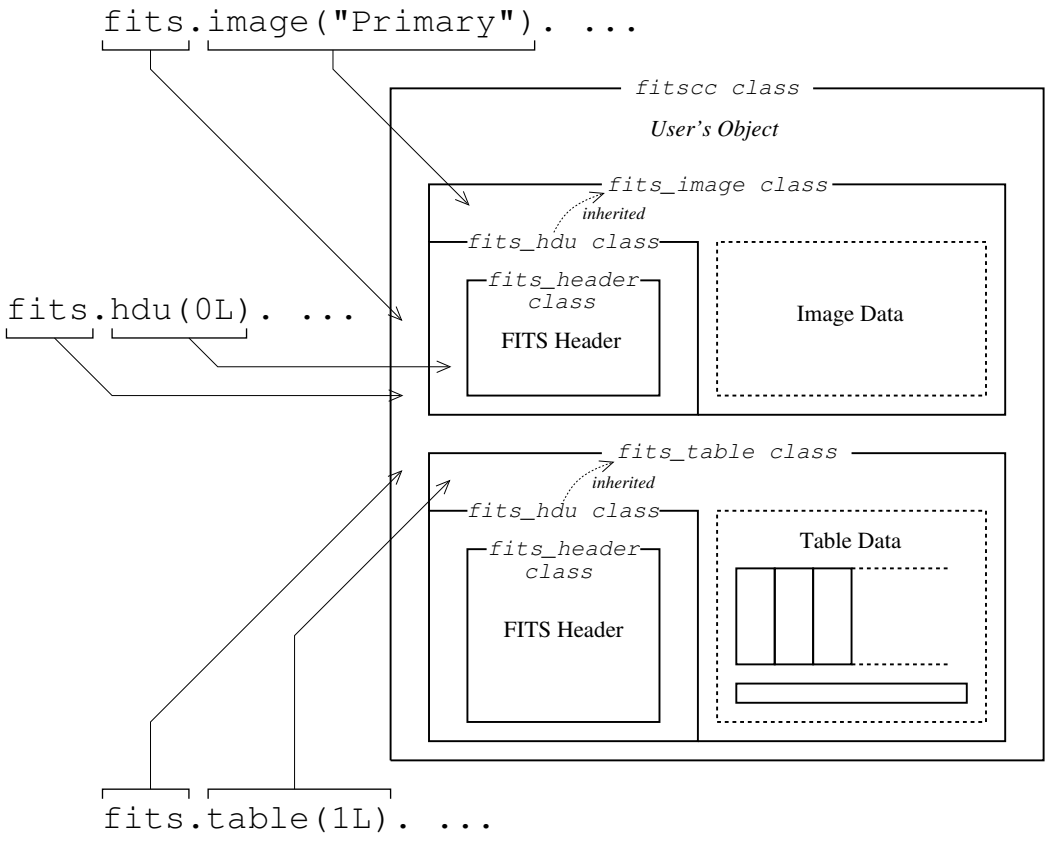


Figure 4: Overall internal structure of the object of "fitscc" class, which corresponds to one FITS file (right) and API that accesses each HDU (left). In this example one FITS file contains one Image HDU and one Binary Table HDU.

With SFITSIO the content of FITS is loaded into the memory buffer in any case of a FITS file being read or a new FITS created. The memory buffer is managed by a top-level `fitscc` class object. Figure 4 provides an example of where a `fitscc` class object has the content of one FITS file. If you compare it with Figure 2 and Figure 3, which illustrate the FITS structure, you will notice the FITS data structure gets "naturally copied" into the object.

A FITS file is represented by the `fitscc` class and an HDU in general by a `fits_hdu` class, as clearly indicated in Figure 4. The Image HDU is represented by a `fits_image` class and a ASCII Table HDU or a Binary Table HDU by a `fits_table` class, both of which inherit the `fits_hdu` class.

The code to access each of these HDU in this internal structure is shown on the left side of Figure 4. The arrows in the figure show the correspondence between the API structure and the class structure. To access an HDU you can write "`fits.image(...). ...`" or "`fits.table(...). ...`" if you know the type of the HDU, or otherwise you can write "`fits.hdu(...). ...`". It goes without saying that you cannot access any information that is specific to a particular HDU type in the latter case. The argument passed onto `hdu(...)` etc. is either the sequential number of the HDU (0-indexed) or the name of the HDU (the value of `EXTNAME` in the header).

4.2 Structure of FITS header class and header access via API

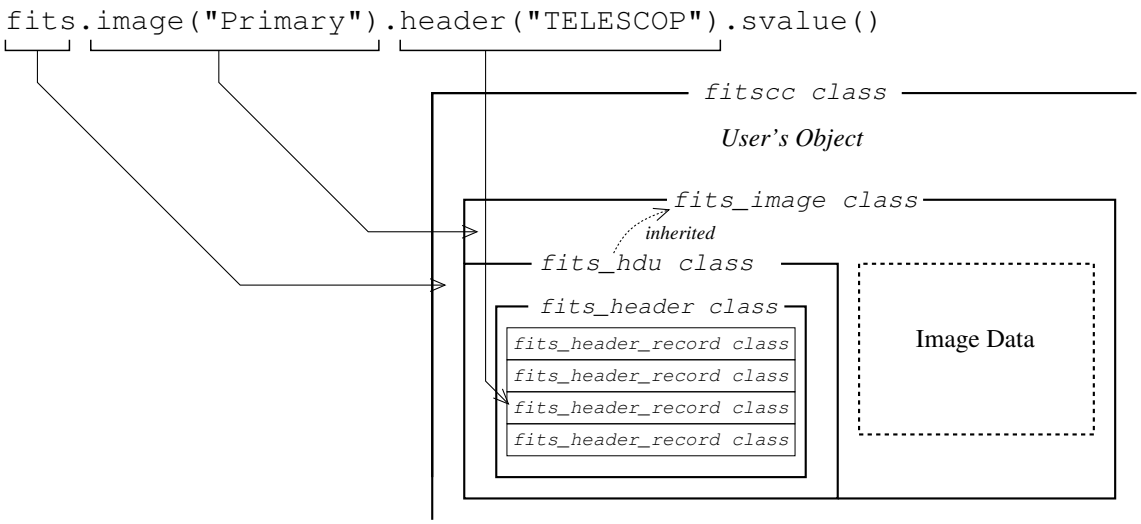


Figure 5: Detailed class structure of the Primary HDU in Figure 4 and example of API that accesses header records.

Next, let's take a look at the structure around the FITS header. `fits_hdu` class controls a `fits_header` class object that represents the entire FITS header, as shown in Figure 5. In turn, the `fits_header` class controls an array of objects of the `fits_header_record` class, each of which represents one header record.

To use the API to access the content of the header, simply write `.header(...). ...` after either of `.image(...)`, `.table(...)`, or `.hdu(...)`. The argument passed onto `.header(...)` is either a header record number (0-indexed) or a header keyword. Please refer to Figure 5 to identify how it corresponds to the class structure[21)].

4.3 Representation of Image HDU

An image HDU is represented by the `fits_image` class, which has an internal structure that only contains internal objects related to the header and an internal `mdarray` object for image data.

[21)] If you take a closer look at the figure you can see that the API structure has only three tiers from `fits` to `header(...)`, whereas the class structure has four tiers from `fitscc` to `fits_header_record`. The lack of a tier in the API is for the sake of less code, but if you feel uncomfortable with that you can also write the code like this: `fits.image(...).header().at(...).svalue()`. In this case, `.header()`, which takes no argument, returns the reference to the `fits_header` class object.

`mdarray` is a generic class provided by SLLIB for use in manipulating arrays of n-dimension, and which simply contain image data of a FITS Data Unit in the memory buffer as it is[22)].

In addition to the member functions of the `fits_image` class, the member functions of `mdarray` class can also be used to read, write, and modify image data. You can use similar APIs between `fits_image` class and `mdarray` class, however, the former are provided for use in FITS-specific operations, while the latter on generic operations for multi-dimensional arrays such as mathematical functions and arithmetic operations.

The following example accesses pixel values using `dvalue(x,y,z)` and `assign(val,x,y,z)`, which are member functions of the `fits_image` class. With this code the pixel value at (x, y) in the image data is copied to (x+1, y).

```
double value = fits.image(0L).dvalue(x, y);
fits.image(0L).assign(value, x+1, y);
```

Member functions of `mdarray` can be used via `data_array()`, which is a member function of the `fits_image` class, as in: `image(...).data_array().trimf(...)`. Another method to use `mdarray`'s APIs is to get reference of `mdarray_float` class or `mdarray_double` class (inherited class from `mdarray`) managed by `fits_image` object. We show an example:

```
fits.image(0L).convert_type(FITS::FLOAT_T);       /* Convert data type */
mdarray_float &img_array
                 = fits.image(0L).float_array();  /* Alias of array object */
```

This method enables you to use member functions that depend on type (i.e., float, double, etc.), which are not allowed in `mdarray` base class.

APIs provided by `mdarray` and its inherited class have simpler implementation, which will bring better performance on average. Therefore, we recommend that you to write SLLIB-based code for full-scale image analysis tools. Please refer example code of Questions 3 of §1.1.

4.4 Representation of ASCII Table HDU and Binary Table HDU

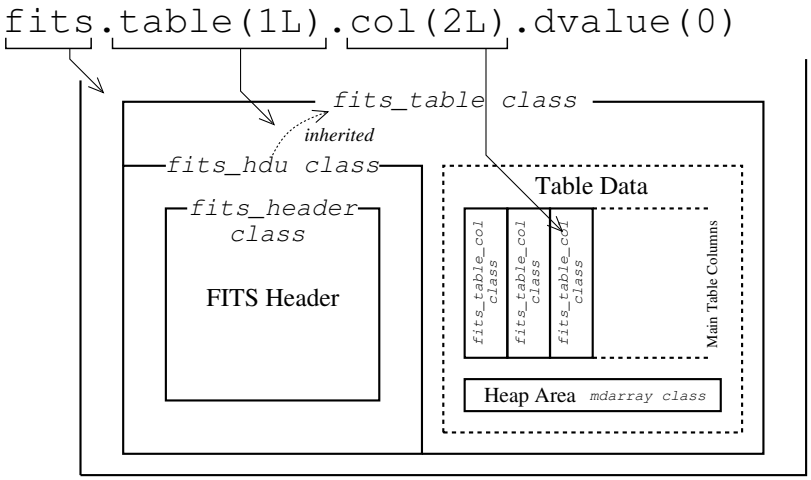


Figure 6: Detailed class structure of Binary Table HDU in Figure 4 and example API used to access cells in the Binary Table.

With SFITSIO both the ASCII Table HDU and Binary Table HDU are represented by the `fits_table` class. In addition to internal objects that represent the FITS header, the `fits_table`

[22)] Only different endian orders get converted.

class has an array of objects of the `fits_table_col` class, which represents one column in its entirety, and a `mdarray` object for the Heap Area. For the sake of efficient data access and to eliminate any worry about the alignment the content of the main table are *divided into columns* and placed in the buffers belonging to the `fits_table_col` class object.

Figure 6 shows the correspondence between the class structure and the API structure; the column in the table is accessible using `table(...)` followed by `.col(...). ...`.

The Heap Area corresponds to the `mdarray` class internal object as with the image buffer of Image HDU. There are only ultra-low level APIs for use with variable-length arrays. In the future they will be accessible using high-level APIs, however, as in `table(...).col(...).dvalue(...)`.

4.5 System and Notice of FITS Header Management with SFITSIO

With the current version of SFITSIO programmers have to use member functions provided in classes used to express the FITS data units (e.g., fits_image class, fits_table class, etc.), rather than member functions that rewrite the content of the header, when updating the structure or properties of data unit. This then means that header records directly updated using member functions of fits_header_record class in the programmer's code will not be effective with written FITS files. Any such direct updates being permitted could lead to confusion for programmers. Directly updating header records used for data unit information is therefore prohibited in fits_header objects managed by the fits_hdu object[23)].

The next table shows the list of header keywords that cannot be updated directly by a programmer's code. This table also includes the member functions used to update the values. Some of these functions can change the data unit structure, properties, etc.

[23)] Strictly speaking, a programmer's use of member functions provided in the fits_header_record class can be detected, and the library routine can update the properties or structure of fits_image objects or fits_table objects. However, currently SFITSIO does not have any code for that type of management.

keyword	member functions to update	section
SIMPLE	–	–
EXTEND	–	–
EXTNAME	assign_hduname()	§13.3.23
EXTVER	assign_hduver()	§13.3.24
BITPIX	image().convert_type()	§13.6.13
NAXIS	image().increase_dim(), etc.	§13.6.20
NAXISn	image().resize()	§13.6.22
BSCALE	image().assign_bscale()	§13.6.15
BZERO	image().assign_bzero()	§13.6.14
BLANK	image().assign_blank()	§13.6.16
XTENSION	table().ascii_to_binary(), etc.	§13.8.36
PCOUNT	–	–
GCOUNT	–	–
THEAP	table().resize_reserved_area()	§13.9.11
TFIELDS	table().append_cols(), etc.	§13.8.46
TXFLDKWD	table().update_col_header(), etc.	§13.8.41
TTYPEn	table().update_col_header(), etc.	§13.8.41
TDISPn	table().update_col_header(), etc.	§13.8.41
TBCOLn	table().define_a_col()	§13.8.38
TFORMn	table().update_col_header(), etc.	§13.8.41
TDIMn	table().update_col_header(), etc.	§13.8.41
TZEROn	table().update_col_header(), etc.	§13.8.41
TSCALn	table().update_col_header(), etc.	§13.8.41
TUNITn	table().update_col_header(), etc.	§13.8.41
TNULLn	table().update_col_header(), etc.	§13.8.41
TELEMn	table().update_col_header(), etc.	§13.8.41
TALASn	table().update_col_header(), etc.	§13.8.41

4.6 Our policy when designing member functions

Designing how member functions are called and how APIs are tiered is almost as important as designing class structure. Details of the content of member functions can be found in §4.7, and categorized by their objective for the sake of clarity. Before going into any detail, however, I will briefly explain our policy when designing the member functions of SFITSIO.

4.6.1 API level

With SFITSIO the member functions that read or write values in the Header Unit or Data Unit are classified into three levels: high, low, and ultra-low. These levels indicate the extent of the operations (BZERO, etc) and conversions performed in conformance with the FITS convention; with the higher the level being the greater the number of operations and conversions.

4.6.2 Rules for arguments and return values

- Common rules

  (1) "Length" and "size" are always `long`.
  (2) A string is basically "`const char *`" or "`char *`".
  (3) All indices of the HDU, headers, images, or tables and order of dimensions are 0-indexed.
  (4) No member function returns the address of a dynamically created object.

- Rules for arguments

  (1) Classes with the `const` attribute are always referenced.

  (2) When specifying x, y, z, etc. the lowest order dimension comes first.

- Rules for return values

  (1) With regard to functions that return a number or status, a negative value means an error has occurred.

  (2) With regard to functions that involve file I/O, the value they return reveals their status.

  (3) Member functions other than (2) that modify the content of the object (i.e., functions without the `const` attribute) always return a reference to themselves.

(3) means it is possible to specify successive operations being used on an image, as in:

```
fits.image(0L).add(image1).add(image2).add(image3). ... ;
```

4.7 Member functions categorized by their objective and API level

4.7.1 File I/O

The return value of each member function reveals their status, and return a negative value if they fail.

Member function	Description	Details
[**fitscc class**]		
`read_stream(const char *)`	Reads a FITS file	§13.3.1
`write_stream(const char *)`	Writes a FITS file	§13.3.3
`access_stream(const char *)`	Reads/writes a FITS file via commands	§13.3.4
`read_template(const char *)`	Reads a template file	§13.3.5
[**fits_header class**]		
`read_stream(cstreamio &)`	Reads the Header Unit only	§13.5.1
`write_stream(cstreamio &)`	Writes the Header Unit only	§13.5.2
`skip_data_stream(cstreamio &)`	Skips over one Data Unit	§13.5.3

4.7.2 Initialization and copying of total content

The names of these member functions always include "`init`". `init()` with no argument can be used to free memory areas that have been previously used.

Member function	Description	Details
[**fitscc class**]		
`init()`	Initializes content	§13.3.15
`init(const fitscc &)`	Copies content of the object passed in	§13.3.15
[**fits_hdu class**]		
`header_init()`	Initializes the header	§13.4.25
`header_init(const fits::header_def [])`	Initializes and configures the header	§13.4.25
`header_init(const fits_header &)`	Copies content of the header	§13.4.25
[**fits_image class**]		
`image(...).init()`	Initializes content	§13.6.18
`image(...).init(const fits_image &)`	Copies content of the object passed in	§13.6.18
`image(...).init(int, long, long, long)`	Initializes with the type and size	§13.6.18
[**fits_table class**]		
`table(...).init()`	Initializes content	§13.8.34
`table(...).init(const fits_table &)`	Copies content of the object passed in	§13.8.34
`table(...).init(const fits::table_def [])`	Initializes according to the column definitions in the argument	§13.8.34
[**fits_table_col class**]		
`table(...).col(...).init()`	Initializes content	§13.8.35
`table(...).col(...).init(const fits_table_col &)`	Copies content of the object passed in	§13.8.35

4.7.3 Swapping the entire content

The names of these member functions always include "`swap`". Reserved keywords that concern the content of the Data Unit cannot be swapped within the header.

Member function	Description	Details
[**fits_hdu class**]		
`header_swap()`	Swaps the entire content apart from reserved keywords	§13.4.26
[**fits_image class**]		
`image(...).swap(fits_image &)`	Swaps content	§13.6.19
[**fits_table class**]		
`table(...).swap(fits_table &)`	Swaps content	§13.8.45
[**fits_table_col class**]		
`table(...).col(...).swap(fits_table_col &)`	Swaps content	§13.8.60

4.7.4 Aquisition of data type

The names of these member functions always include "`type`". For constants in the return value refer to §13.1.

Member function	Return value	Details
[**fits_hdu class**]		
`hdu(...).hdutype()`	Type of the HDU	§13.3.13
[**fits_header_record class**]		
`hdu(...).header(...).type()`	Type of header record	§13.4.13
[**fits_image class**]		
`image(...).type()`	Type of image data	§13.6.5
[**fits_table_col class**]		
`table(...).col(...).type()`	Type of table column	§13.8.8
`table(...).col(...).heap_is_used()`	Whether the array is variable-length	§13.8.9
`table(...).col(...).heap_type()`	Data type of variable-length array	§13.8.10

4.7.5 Retrieval of the length or size

The names of these member functions always include "`bytes`" or "`length`". All return values are of type `long`.

Member function	Return value	Details
[**fitscc class**]		
`length()`	Number of HDUs	§13.3.7
`stream_length()`	Byte length of output uncompressed FITS file	§13.3.6
[**fits_hdu class**]		
`hdu(...).header_length()`	Number of header records	§13.4.1
`hdu(...).header_value_length(...)`	Length of raw string value of the header record (negative value is returned when keyword is not found)	§**??**
[**fits_image class**]		
`image(...).dim_length()`	Number of dimensions	§13.6.3
`image(...).length()`	Complete number of pixels	§13.6.4
`image(...).length(long)`	Number of pixels in the dimensions specified by the argument	§13.6.4
`image(...).bytes()`	Byte length of single pixel	§13.6.6
`image(...).col_length()`	Pixel width of image	§13.6.7
`image(...).row_length()`	Pixel height of image	§13.6.8
`image(...).layer_length()`	Number of layers	§13.6.9
[**fits_table class**]		
`table(...).col_length()`	Number of columns (fields)	§13.8.3
`table(...).row_length()`	Number of rows	§13.8.4
[**fits_table_col class**]		
`table(...).col(...).bytes()`	Byte length of an element in the column	§13.8.11
`table(...).col(...).elem_byte_length()`	Byte length of the column	§13.8.12
`table(...).col(...).elem_length()`	Number of elements in the column	§13.8.13
`table(...).col(...).dcol_length()`	Number of columns in a multi-dimensional array	§13.8.14
`table(...).col(...).drow_length()`	Number of rows in a multi-dimensional array	§13.8.15
`table(...).col(...).heap_bytes()`	Byte length of an element in a variable-length array	§13.8.16
`table(...).col(...).max_array_length()`	Maximum length of a variable-length array	§13.8.17
`table(...).col(...).array_length(long)`	Length of the row specified by the argument in a variable-length array	§13.8.18

4.7.6 Retrieval of index for internal object array

Although you can quickly access table columns etc. by name, thanks to the internal search mechanisms, you may wish to use index in an internal object array if you are particularly performance-conscious, for example, large number of accesses in a loop. To accomplish that you will need to acquire the index for the name beforehand using the following member functions. These take the name as an argument and return the index in an internal object array (`long` and 0-indexed). They return a negative number if the argument does not match, and therefore they are often used to *verify its existence*.

Member function	Return value	
[**fitscc class**]		
`index( const char * )`	Index for the HDU	§13.3.14
[**fits_hdu class**]		
`hdu(...).header_index( const char * )`	Index for the header record	§13.4.2
[**fits_table class**]		
`table(...).col_index( const char * )`	Index for the table column	§13.8.6

4.7.7 Retrieval/configuration of the name or the version of a construct

Member function	Description	Details
[**fitscc class**]		
`fmttype()`	Returns the value of `FMTTYPE` (string)	§13.3.8
`assign_fmttype(...)`	Sets the value of `FMTTYPE`	§13.3.21
`ftypever()`	Returns the value of `FTYPEVER` (integer)	§13.3.9
`assign_ftypever(...)`	Sets the value of `FTYPEVER`	§13.3.22
[**fits_hdu class**]		
`hdu(...).hduname()`	Returns the name of the HDU (the value of `EXTNAME`; string)	§13.3.10
`hdu(...).assign_hduname(...)`	Sets the name of the HDU (the value of `EXTNAME`; string)	§13.3.23
`hdu(...).hduver()`	Returns the version of the HDU (the value of `EXTVER`; integer)	§13.3.11
`hdu(...).assign_hduver(...)`	Sets the version of the HDU (the value of `EXTVER`; integer)	§13.3.24
`hdu(...).hduver_is_set()`	Verifies the existence of the version of the HDU (the value of `EXTVER`; integer)	§13.3.26
`hdu(...).hdulevel()`	Returns the level of the HDU (the value of `EXTLEVEL`; integer)	§13.3.12
`hdu(...).assign_hdulevel(...)`	Sets the level of the HDU (the value of `EXTLEVEL`; integer)	§13.3.25
`hdu(...).hdulevel_is_set()`	Verifies the existence of the level of the HDU (the value of `EXTLEVEL`; integer)	§13.3.27
[**fits_table_col class**]		
`table(...).col(...).name()`	Returns the name of the column	§13.8.7
`table(...).col(...).assign_name(...)`	Sets the name of the column	§13.8.37

4.7.8 Data reading (high level)

These member functions perform FITS-specific operations and convert the type of values into what to be returned and are typically used by the SFITSIO programmers to read data.

With string values in the header the quotation marks and blank characters on both sides are removed, the quotation marks in the string converted from "'' " to "'", and the resulting string then returned.

When retrieving values from the Data Unit, the values of `BZERO`, `BSCALE`, `TZERO`n, and `TSCAL`n in the header are taken into account. In addition, the functions `dvalue()`, `svalue()`, and `get_svalue()` return NaN or the NULL string, which defaults to `"NULL"`[24], for the case where the data value equals the value of either `BLANK` or `TNULL`n in the header, respectively.

- **Reads and returns data**

  The name of the member function is always `dvalue()`, `lvalue()`, `llvalue()`, `bvalue()`, or `svalue()`, and which returns type `double`, `long`, `long long`, `bool`, or string (`const char *`), respectively.

Member function	Type of return value	Details
[**fits_header_record class**]		
`hdu(...).header(...).dvalue()`	`double`	§13.4.6
`hdu(...).header(...).lvalue()`	`long`	§13.4.7
`hdu(...).header(...).llvalue()`	`long long`	§13.4.7
`hdu(...).header(...).bvalue()`	`bool`	§13.4.8
`hdu(...).header(...).svalue()`	`const char *`	§13.4.4
[**fits_image class**]		
`image(...).dvalue(long,long,long)`	`double`	§13.6.10
`image(...).lvalue(long,long,long)`	`long`	§13.6.11
`image(...).llvalue(long,long,long)`	`long long`	§13.6.11
[**fits_table_col class**]		
`table(...).col(...).dvalue(...)`	`double`	§13.8.20
`table(...).col(...).lvalue(...)`	`long`	§13.8.21
`table(...).col(...).llvalue(...)`	`long long`	§13.8.21
`table(...).col(...).bvalue(...)`	`bool`	§13.8.22
`table(...).col(...).svalue(...)`	`const char *`	§13.8.23

- **Reads and stores the data in a buffer passed in as an argument**

  The name of the member function is "`get_svalue()`". It stores the string value to a buffer passed in as an argument.

Member function	Details
[**fits_header_record class**]	
`hdu(...).header(...).get_svalue(char *, size_t)`	§13.4.5
[**fits_table_col class**]	
`table(...).col(...).get_svalue(long, char *, size_t)`, etc.	§13.8.24

4.7.9 Data writing (high level)

The name of the member function is always "`assign()`". These member functions perform FITS-specific operations and type conversions and then store the resulting values in the data buffer of FITS, and are typically used by SFITSIO programmers to write data.

With string values in the header the quotation marks get appended on both sides, and the quotation marks in the string converted from "'" to "''".

[24] the NULL string can be configured with `table(...).assign_null_svalue(...)`; for more details refer to §13.8.29.

When writing values into the Data Unit, the values of `BZERO`, `BSCALE`, `TZERO`n, and `TSCAL`n in the header are taken into account. In addition, if NaN or the NULL string, which defaults to `"NULL"`[25], is passed in as the argument of either a floating point value or a string value, the value of `BLANK` or `TNULL`n in the header respectively is stored.

Member function	Details
[**fits_header_record class**]	
`hdu(...).header(...).assign(...)`	§13.4.9
[**fits_image class**]	
`image(...).assign(...)`	§13.6.12
[**fits_table_col class**]	
`table(...).col(...).assign(...)`	§13.8.25

[25] the NULL string can be configured using `table(...).assign_null_svalue(...)`; for more details refer to §13.8.29.

4.7.10 Data reading (low level)

These member functions convert the type of values into what to be returned without performing any FITS-specific operations, and which can be used if there is no `BZERO`, `BSCALE`, `BLANK`, `TZERO`n, `TSCAL`n, or `TNULL`n settings in the header or if maximum performance is required (if the utmost performance is required at any cost you may wish to use an ultra-low level API). They are the most effective when the data type of the member function matches the type of the FITS data (the bit type needs a somewhat large amount of operations, though).

With the string values in the header the quotation marks on either side do not get removed. The saved data values are retrieved without any operations as with `BZERO` or `BLANK` when retrieving the values from the Data Unit.

- **Reads and returns data**

  The name of the member function is simply `value()` if it reads from the header, with the name being fully qualified by the return type, as with `double_value()`, `long_value()`, if the function reads from the Data Unit.

Member function	Type of return value	Details
[**fits_header_record class**]		
`hdu(...).header(...).value()`	`const char *`	§13.4.17
[**fits_image class**]		
`image(...).double_value(long,long,long)`	`double`	§13.7.5
`image(...).float_value(long,long,long)`	`float`	§13.7.6
`image(...).longlong_value(long,long,long)`	`long long`	§13.7.7
`image(...).long_value(long,long,long)`	`long`	§13.7.8
`image(...).short_value(long,long,long)`	`short`	§13.7.9
`image(...).byte_value(long,long,long)`	`unsigned char`	§13.7.10
[**fits_table_col class**]		
`table(...).col(...).double_value(...)`	`double`	§13.9.17
`table(...).col(...).float_value(...)`	`float`	§13.9.16
`table(...).col(...).longlong_value(...)`	`long long`	§13.9.14
`table(...).col(...).long_value(...)`	`long`	§13.9.13
`table(...).col(...).short_value(...)`	`short`	§13.9.12
`table(...).col(...).byte_value(...)`	`unsigned char`	§13.9.15
`table(...).col(...).logical_value(...)`	`int`	§13.9.19
`table(...).col(...).bit_value(...)`	`long`	§13.9.18
`table(...).col(...).string_value(...)`	`const char *`	§13.9.20
`table(...).col(...).array_heap_offset(...)`	`long`	§13.9.21

- **Reading and storing the data in the buffer passed in as an argument**

  The name of the member function is "`get_value()`" if it operates on the header, and "`get_string_value()`" if it operates on the Data Unit. In either case it stores the string value in the buffer passed in as an argument.

Member function	Details
[**fits_header_record class**]	
`hdu(...).header(...).get_value(char *, size_t)`	§13.4.18
[**fits_table_col class**]	
`table(...).col(...).get_string_value(long, char *, size_t)`, etc	§13.9.22

4.7.11 Data writing (low level)

These member functions perform type conversions into a FITS data type and store the resulting values in the data buffer without performing any FITS-specific operations, and which can be used if there is no `BZERO`, `BSCALE`, `BLANK`, `TZERO`n, `TSCAL`n, or `TNULL`n settings in the header or if maximum performance is required (if the utmost performance is required at any cost you may wish to use an ultra-low level API). They are the most effective when the data type of the member function matches the type of the FITS data (the bit type needs a somewhat large amount of operations, though).

With the string values in the header the quotation marks on either side do not get appended. Values are written to the Data Unit without any operations as with `BZERO` or `BLANK`.

Member function	Details
[**fits_header_record class**]	
`hdu(...).header(...).assign_value(...)`, etc.	§13.4.21
[**fits_image class**]	
`image(...).assign_double(...)`	§13.7.11
`image(...).assign_float(...)`	§13.7.12
`image(...).assign_longlong(...)`	§13.7.13
`image(...).assign_long(...)`	§13.7.14
`image(...).assign_short(...)`	§13.7.15
`image(...).assign_byte(...)`	§13.7.16
[**fits_table_col class**]	
`table(...).col(...).assign_double(...)`	§13.9.28
`table(...).col(...).assign_float(...)`	§13.9.27
`table(...).col(...).assign_longlong(...)`	§13.9.25
`table(...).col(...).assign_long(...)`	§13.9.24
`table(...).col(...).assign_short(...)`	§13.9.23
`table(...).col(...).assign_byte(...)`	§13.9.26
`table(...).col(...).assign_logical(...)`	§13.9.30
`table(...).col(...).assign_bit(...)`	§13.9.29
`table(...).col(...).assign_string(...)`	§13.9.31
`table(...).col(...).assign_arrdesc(...)`	§13.9.32

4.7.12 Data access (ultra-low level)

These high-risk-high-return member functions directly read/write raw byte data in the Data Unit. Because the different endian orders get converted and the alignment adjusted for the processing system within the buffer of an object, apart from the Heap Area in a binary table, you can immediately access the data as soon as the address in the pointer variable is of the correct type. With SFITSIO, there are definitions for data types that are used in FITS files (such as `fits::logical_t` and `fits::double_t`), that can be used to declare pointer variables (for the definition of the types refer to §13.2).

You can use these member functions when in pursuit of maximum performance but *type checks are left up to programmers with regard to the functions that exchange data with the type* `void *`*; if you do not understand the connotations they can have unexpected effects.* And hence you should be forewarned.

The names for these member functions are `get_data()`, `put_data()`, `data_ptr()`, and *type-name*`_ptr()`, as well as `get_heap()`, `put_heap()`, and `heap_ptr()` for the Heap Area of the Binary Table. Please note that different endian orders do not get converted (theoretically impossible) in the data buffers of the Heap Area, which can be arbitrarily aligned. Extreme caution is therefore required with the reading or writing of data of the type larger than one byte. API functions for variable-length arrays are summarized in §4.7.20.

Member function	Description	Details
[**fits_image class**]		
`image(...).get_data(...)`	Retrieves byte data to the buffer passed in as an argument	§13.7.3
`image(...).put_data(...)`	Writes data in the buffer passed in as an argument	§13.7.4
`image(...).byte_t_ptr(...)`	Retrieves the address of the internal buffer (type: `fits::byte_t *`)	§13.7.2
`image(...).short_t_ptr(...)`	Retrieves the address of the internal buffer (type : `fits::short_t *`)	§13.7.2
`image(...).long_t_ptr(...)`	Retrieves the address of the internal buffer (type: `fits::long_t *`)	§13.7.2
`image(...).longlong_t_ptr(...)`	Retrieves the address of the internal buffer (type: `fits::longlong_t *`)	§13.7.2
`image(...).float_t_ptr(...)`	Retrieves the address of the internal buffer (type: `fits::float_t *`)	§13.7.2
`image(...).double_t_ptr(...)`	Retrieves the address of the internal buffer (type: `fits::double_t *`)	§13.7.2
`image(...).data_ptr(...)`	Retrieves the address of the internal buffer (type: `void *`)	§13.7.2
[**fits_table_col class**]		
`table(...).col(...).get_data(...)`	These functions are for use with the column data area in the main table.	§13.9.3
`table(...).col(...).put_data(...)`		§13.9.4
`table(...).col(...).bit_t_ptr(...)`		§13.9.2
`table(...).col(...).byte_t_ptr(...)`		§13.9.2
`table(...).col(...).logical_t_ptr(...)`		§13.9.2
`table(...).col(...).ascii_t_ptr(...)`		§13.9.2
`table(...).col(...).short_t_ptr(...)`		§13.9.2
`table(...).col(...).long_t_ptr(...)`		§13.9.2
`table(...).col(...).longlong_t_ptr(...)`		§13.9.2
`table(...).col(...).float_t_ptr(...)`		§13.9.2
`table(...).col(...).double_t_ptr(...)`		§13.9.2
`table(...).col(...).complex_t_ptr(...)`		§13.9.2
`table(...).col(...).doublecomplex_t_ptr(...)`		§13.9.2
`table(...).col(...).longarrdesc_t_ptr(...)`		§13.9.2
`table(...).col(...).llongarrdesc_t_ptr(...)`		§13.9.2
`table(...).col(...).data_ptr()`		§13.9.2
[**fits_table class**]		
`table(...).get_heap(...)`	These functions are for use with the Heap Area of the Binary Table.	§13.9.6
`table(...).put_heap(...)`		§13.9.7
`table(...).heap_ptr()`		§13.9.5

4.7.13 Conversion between data types

These member functions quickly convert the different types of numerical data into each other. `convert_type()` with the Binary Table only converts columns of the numerical type, apart from variable-length arrays.

Member function	Description	Details
[**fits_image class**]		
`image(...).convert_type(...)`	Converts the object into an image of the specified type, ZERO values, SCALE values, and BLANK values	§13.6.13
[**fits_table_col class**]		
`table(...).col(...).convert_type(...)`	Converts the object into a column of the specified type, ZERO values, SCALE values, and NULL values	§13.8.28

4.7.14 Member functions that operate on ZERO/SCALE/BLANK/UNIT values

For modifying `TDIM`*n*, etc. in binary tables and ASCII tables refer to §4.7.18.

Member function	Description	Details
[**fits_image class**]		
`image(...).bzero()`	Retrieves the BZERO value	§13.6.14
`image(...).assign_bzero()`	Sets the BZERO value	§13.6.14
`image(...).bscale()`	Retrieves the BSCALE value	§13.6.15
`image(...).assign_bscale()`	Sets the BSCALE value	§13.6.15
`image(...).blank()`	Retrieves the BLANK value	§13.6.16
`image(...).assign_blank()`	Sets the BLANK value	§13.6.16
`image(...).bunit()`	Retrieves the BUNIT value	§13.6.17
`image(...).assign_bunit()`	Sets the BUNIT value	§13.6.17
[**fits_table_col class**]		
`table(...).col(...).tzero()`	Retrieves the TZERO value	§13.8.30
`table(...).col(...).assign_tzero()`	Sets the TZERO value	§13.8.30
`table(...).col(...).tscal()`	Retrieves the TSCAL value	§13.8.31
`table(...).col(...).assign_tscal()`	Sets the TSCAL value	§13.8.31
`table(...).col(...).tnull()`	Retrieves the TNULL value	§13.8.32
`table(...).col(...).assign_tnull()`	Sets the TNULL value	§13.8.32
`table(...).col(...).tunit()`	Retrieves the TUNIT value	§13.8.33
`table(...).col(...).assign_tunit()`	Sets the TUNIT value	§13.8.33

4.7.15 Data editing

These functions are used to modify the size of the buffer area or append, insert, or remove elements. These functions have consistent names such as `resize`, `append`, `insert`, or `erase`. Some of the classes, however, include differently named member functions.

Member function	Description	Details
[**fitscc class**]		
`append_image(...)`	Appends an Image HDU	§13.3.16
`append_table(...)`	Appends an ASCII Table or Binary Table HDU	§13.3.17
`insert_image(...)`	Inserts an Image HDU	§13.3.18
`insert_table(...)`	Inserts an ASCII Table or Binary Table HDU	§13.3.19
`erase(...)`	Removes an HDU	§13.3.20
[**fits_hdu class**]		
`hdu(...).header_append_records(...)`	Appends multiple header records	§13.4.27
`hdu(...).header_append(...)`	Appends a single header record	§13.4.28
`hdu(...).header_insert_records(...)`	Inserts multiple header records	§13.4.29
`hdu(...).header_insert(...)`	Inserts a single header record	§13.4.30
`hdu(...).header_erase_records(...)`	Removes multiple header records	§13.4.31
`hdu(...).header_erase(...)`	Removes a single header record	§13.4.32
`hdu(...).header_update(...)`	Appends or updates a single record	§13.4.23
`hdu(...).header_assign(...)`	Updates a single record	§13.4.24
`hdu(...).header(...).assign_comment(...)`	Sets a comment for a header record	§13.4.22
`hdu(...).header_rename(...)`	Modifies the keyword for a single header record	§13.4.33
[**fits_image class**]		
`image(...).increase_dim(...)`	Increases the dimensions	§13.6.20
`image(...).decrease_dim(...)`	Decreases the dimensions	§13.6.21
`image(...).resize_2d(...)`, etc.	Modifies the size of image	§??
`image(...).resize(...)`	Modifies the size of each dimension	§13.6.22
`image(...).assign_default(...)`	Sets the pixel value of new elements created by `.resize()`, etc.	§13.6.23
`image(...).transpose_xy()`	Performs fast transpose of (x, y)	§??
`image(...).flipf(...)`	Selects a section in n-dimensions, and flips the order of elements of specified dimensions	§??
`image(...).trimf(...)`	Trims a section in n-dimensions	§??
`image(...).copy(fits_image *, ...)`	Copies the specified area to another `fits_image` object	§13.6.34
`image(...).clean(...)`	Modifies pixel values in the specified area to a default value	§??
`image(...).fill(double, ...)`	Modifies pixel values in the specified area to a single value	§13.6.28
`image(...).paste(const fits_image &, ...)`	Pastes an image from another `fits_image` object	§13.6.35
`image(...).data_array().swap(...)`, etc	Uses member functions of the mdarray class to edit the image. See also §4.7.16.	§13.7.1

Member function	Description	Details
[**fits_table class**]		
`table(...).resize_rows(...)`	Modifies the length of the row	§13.8.51
`table(...).append_rows(...)`	Appends rows	§13.8.52
`table(...).insert_rows(...)`	Inserts rows	§13.8.53
`table(...).erase_rows(...)`	Remove rows	§13.8.54
`table(...).clean_rows(...)`	Initializes rows	§13.8.55
`table(...).move_rows(...)`	Copies rows	§13.8.56
`table(...).swap_rows(...)`	Interchanges rows	§13.8.57
`table(...).import_rows(...)`	Imports a table	§13.8.58
`table(...).append_cols(...)`	Appends multiple columns	§13.8.46
`table(...).append_a_col(...)`	Appends a single column	§13.8.46
`table(...).insert_cols(...)`	Inserts multiple columns	§13.8.47
`table(...).insert_a_col(...)`	Inserts a single column	§13.8.47
`table(...).swap_cols(...)`	Interchanges columns	§13.8.48
`table(...).erase_cols(...)`	Removes multiple columns	§13.8.49
`table(...).erase_a_col(...)`	Removes a single column	§13.8.49
`table(...).copy(...)`	Copies to another object	§13.8.50
`table(...).assign_null_svalue(...)`	Sets the NULL string for use in a high level API	§13.8.29
`table(...).ascii_to_binary()`	Converts an ASCII table to a Binary table	§13.8.36
[**fits_table_col class**]		
`table(...).col(...).move(...)`	Copies between rows in a column	§13.8.59
`table(...).col(...).swap(...)`	Interchanges rows in a column	§13.8.60
`table(...).col(...).clean(...)`	Initializes values in a column	§13.8.61
`table(...).col(...).import(...)`	Imports to a specific column	§13.8.62
`table(...).col(...).assign_default(...)`	Sets the cell value of new elements created by `.resize_rows()`, etc.	§13.8.63

4.7.16 Member functions of the mdarray class that can be used to process images

Member functions of `mdarray` can be used via `data_array()`, which is a member function of the `fits_image` class, as in: `image(...).data_array().trimf(...)`.

Another method to use `mdarray`'s APIs is to get reference of `mdarray_float` class or `mdarray_double` class (inherited class from `mdarray`) managed by `fits_image` object.

Representative member functions of the mdarray class will be covered here. Refer to the SLLIB manual or `mdarray.h`, `mdarray_float.h`, etc.

Member function	Description
[**mdarray class**]	
`increase_dim(...)`	Increases the dimensions
`decrease_dim(...)`	Decreases the dimensions
`resize_1d(...)`, `resize_2d(...)`, etc.	Modifies the size of array
`resize(...)`, `resizef(...)`	Modifies the size of each dimension
`resizeby_1d(...)`, `resizeby_2d(...)`, etc.	Modifies the size of array relatively
`resizeby(...)`, `resizebyf(...)`	Modifies the size of each dimension relatively
`assign_default(...)`	Sets the value of new elements created by `.resize()`, etc.
`swap(...)`	Interchanges interval data of the specified dimensions
`move(...)`	Copies interval data of the specified dimensions
`cpy(...)`	Copies interval data of the specified dimensions (the buffer size is modified as needed)
`insert(...)`	Inserts blank data of the specified dimensions
`crop(...)`	Extracts an interval of the specified dimensions
`erase(...)`	Removes an interval of the specified dimensions
`round()`, `trunc()`, etc.	Iterates over all pixels to perform an operation and store the results. Available operations include `.ceil()`, `.floor()`, `.round()`, `.trunc()`, and `.abs()`.
`transpose_xy()`	Performs fast transpose of (x, y)
`transpose_xyz2zxy()`	Performs fast transpose of (x, y, z) to (z, x, y)
`flipf(...)`	Selects a section in n-dimensions, and flips the order of elements of specified dimensions
`trimf(...)`	Trims a section in n-dimensions
`sectionf(...)`	Selects a section in n-dimensions, and copies them into another object
`copyf(mdarray *, ...)`	Selects a section in n-dimensions, and copies them into another object
`transposef_xy_copy(mdarray *, ...)`	Selects a section in n-dimensions, performs a fast transpose of (x, y), and copies them into another object
`transposef_xyz2zxy_copy(mdarray *, ...)`	Selects a section in n-dimensions, performs a fast transpose of (x, y, z) to (z, x, y), and copies them into another object
`cleanf(...)`	Selects a section in n-dimensions, and modifies elements in the section to default value
`fillf(double, ...)`	Selects a section in n-dimensions, and modifies elements in the section to a single value
`pastef(const mdarray &, ...)`	Pastes an array from another `mdarray` object on specified section in n-dimensions
`addf(double, ...)`, etc.	Performs the four rules of arithmetic between an array of specified section and a scalar
`addf(const mdarray &, ...)`, etc.	Performs the four rules of arithmetic between an array of specified section and another array
`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`	Performs the four rules of arithmetic between an array and a scalar or an array
[**mdarray_* class**]	
`[...]`	Accesses an element for one dimensional data
`(...)`	Accesses an element for one, two or three dimensional data
`array_ptr(...)`	Returns an address of internal buffer of the array
`array_ptr_2d(bool)`	Returns generated 2-d pointer array
`array_ptr_3d(bool)`	Returns generated 3-d pointer array

4.7.17 Member functions for use in image analysis

4.7.18 Member functions that operate on column definitions of tables

The following member functions with "`col_header`" can handle both reserved keywords (such as `TDISP`n) and non-reserved keywords (such as `TLMAX`n) with respect to column definitions in an ASCII Table or Binary Table. As shown in the example below the column name (or index) and the name of the keyword can be specified using separate arguments, thus eliminating the need for `snprintf()` to be used to concatenate the keyword with n.

```
/* Modifies the TUNUT value and comment of the column "R-BAND"
   within the "EVENT" table */
fits.table("EVENT").update_col_header("R-BAND", "TUNIT",
                                      "mag", "absolute magnitude");
```

Modifying the value of a reserved keyword results in the data in the object being automatically updated. In the case of a non-reserved keyword `TXFLDKWD` will also be automatically updated (refer to §11.7 for more details on `TXFLDKWD`).

The `define(...)` member function uses a structure `fits::table_def` to modify the value of a reserved keyword.

Member function	Description	Details
[**fits_table class**]		
`table(...).col_header_index(...)`	Returns the index to the header record with the specified column definition	§13.8.39
`table(...).col_header(...).svalue()`, etc.	Accesses the header record with the specified column definition	§13.8.40
`table(...).update_col_header(...)`	Modifies a single header record of a column definition	§13.8.41
`table(...).erase_col_header(...)`	Removes a single header record of a column definition	§13.8.42
`table(...).rename_col_header(...)`	Modifies the name of the user-defined header keyword for columns	§13.8.43
`table(...).sort_col_header()`	Sorts header keywords of columns in their order	§13.8.44
[**fits_table_col class**]		
`table(...).col(...).definition()`	Reads a single column definition	§13.8.19
`table(...).col(...).define(...)`	Modifies a single column definition	§13.8.38

4.7.19 Special member functions used for processing headers

Function or member function	Description	Details
[**fits_hdu class**]		
`hdu(...).header(...).status()`	Retrieves the status of the header record	§13.4.14
`hdu(...).header(...).keyword()`	Retrieves the keyword of the header record	§13.4.15
`hdu(...).header(...).get_keyword(...)`	Retrieves the keyword of the header record	§13.4.16
`hdu(...).header(...).comment()`	Retrieves the comment of the header record	§13.4.19
`hdu(...).header(...).get_comment(...)`	Retrieves the comment of the header record	§13.4.20
`hdu(...).header_regmatch(...)`	Searches for keywords with the regular expression	§13.4.3
`hdu(...).header(...).assign_system_time()`	Sets the current time (UTC) using the "`yyyy-mm-ddThh:mm:ss`" format	§13.4.37
`hdu(...).header_formatted_string()`	Returns the header string formatted for FITS files	§13.4.35
`hdu(...).header_fill_blank_comments()`	Fills the comment field with the content of the header comment dictionary in SFITSIO if no comment exists	§13.4.38
`hdu(...).header_assign_default_comments()`	Fills the comment field with the content of the header comment dictionary in SFITSIO regardless of whether a comment already exists	§13.4.39
[**Function**]		
`fits::update_comment_dictionary(...)`	Appends or updates the content of the comment dictionary	§13.4.40

4.7.20 Member functions used on variable-length arrays in binary tables (low-level APIs)

Variable-length arrays are not supported by the high-level API in the current version of SFITSIO and hence the following low-level API functions have to be used. See also the sample code in `test/access_bte_heap.cc` and `sample/create_vl_array.cc` in the source package of SFITSIO.

Function or member function	Description	Details
[**fits_table class**]		
`table(...).get_heap(...)`	These functions are for use with the Heap Area of the Binary Table.	§13.9.6
`table(...).put_heap(...)`		§13.9.7
`table(...).heap_ptr()`		§13.9.5
`table(...).heap_length()`	Returns the byte length of the Heap Area	§13.8.5
`table(...).resize_heap()`	Modifies the size of the Heap Area	§13.9.8
`table(...).reverse_heap_endian()`	Converts the endian orders of the Heap Area	§13.9.9
`table(...).reserved_area_length()`	Returns the byte length of the reserved area	§13.9.10
`table(...).resize_reserved_area()`	Modifies the size of the reserved area	§13.9.11
[**fits_table_col class**]		
`table(...).col(...).heap_is_used()`	Determines whether the column is a variable-length array	§13.8.9
`table(...).col(...).heap_type()`	Determines the data type of the variable-length array	§13.8.10
`table(...).col(...).heap_bytes()`	Determines the byte length of an element of a variable-length array	§13.8.16
`table(...).col(...).max_array_length()`	Returns the maximum size of a variable-length array	§13.8.17
`table(...).col(...).array_length(long)`	Returns the length of a variable-length array of the row specified by the argument	§13.8.18
`table(...).col(...).array_heap_offset(...)`	Returns the position in the heap of the variable-length array concerned	§13.9.21
`table(...).col(...).assign_arrdesc(...)`	Sets a array descriptor for a variable-length array	§13.9.32

5 Tutorial

We would like to introduce you to the easy going world of SFITSIO using some simple examples.

5.1 First Incantation

Write the following code at the top of your source code if using SFITSIO.

```
#include <sli/fitscc.h>
using namespace sli;
```

5.2 Read/Write files (Create "imcopy" of IRAF)

You can read and write files using the member functions of `read_stream()` and `write_stream()`, respectively.

In the below example the program reads the file specified by the first argument in a command line, outputs the HDU information, and writes all the content of the file that was read into the file specified by the second argument. First, the code without error handling is shown.

```
#include <stdio.h>
#include <sli/fitscc.h>
using namespace sli;

int main( int argc, char *argv[] )
{
    long i;
    fitscc fits;                                /* object "fits" */
    fits.read_stream(argv[1]);                  /* reading FITS file */
    for ( i=0 ; i < fits.length() ; i++ ) {     /* output HDU info */
        printf("HDU %ld : hduname = %s\n", i, fits.hduname(i));
    }
    fits.write_stream(argv[2]);                 /* writing FITS file */
    return 0;
}
```

Compile and run the program. It can be compiled the easiest using s++.

```
$ s++ fits_io.cc -lsfitsio
g++ -I/usr/local/include -L/usr/local/lib -Wall -O fits_io.cc -o fits_io -lsfits
io -lsllib -lz -lbz2 -lreadline -lcurses
$ ./fits_io in.fits out.fits
HDU 0 : hduname = Primary
```

Second, the following code is modified for ensuring error handling. Programmers generally do not need to allocate memory when using SFITSIO and hence minimum error handling is required[26]. You, however, will need to handle any errors using the return values of member functions that input/output files.

[26] With SFITSIO any failure with memory allocation causes the program to display the reason using the standard-error output and to abort.

```
#include <stdio.h>
#include <sli/fitscc.h>
using namespace sli;

int main( int argc, char *argv[] )
{
    int return_status = -1;
    fitscc fits;                            /* object "fits" */
    ssize_t sz;

    if ( 1 < argc ) {
        long i;
        const char *in_file = argv[1];
        sz = fits.read_stream(in_file);     /* reading FITS file */
        if ( sz < 0 ) {                     /* error handling */
            fprintf(stderr,"[ERROR] fits.read_stream() failed\n");
            goto quit;
        }
        /* display HDU information */
        for ( i=0 ; i < fits.length() ; i++ ) {
            fprintf(stderr,"HDU %ld : hduname = %s\n",i,fits.hduname(i));
        }
    }
    if ( 2 < argc ) {
        const char *out_file = argv[2];
        sz = fits.write_stream(out_file);   /* writing FITS file */
        if ( sz < 0 ) {                     /* error handling */
            fprintf(stderr,"[ERROR] fits.write_stream() failed\n");
            goto quit;
        }
    }
    return_status = 0;
quit:
    return return_status;
}
```

You have created "imcopy" task of IRAF! Your program can read a FITS file, trim, and flip a part of image using additional expression after input filename like this:

```
$ ./fits_io in.fits'[100:1,*]' out.fits
```

Let's try various expressions explained in §8.

See also `sample/read_and_write.cc` in the SFITSIO source package.

5.3 Directly accessing remote FITS files via a network

You can use `read_stream()` and `write_stream()` to access remote files via a network. In this case, however, you will need to specify the path names beginning with `http://` or `ftp://` in the argument of the member functions.

The following example reads a file from a web server.

```
    sz = fits.read_stream("http://www.xxx.jp/fits_data/foo.fits.bz2");
```

Although the library can only be used to read from web servers it can also write to FTP servers.

```
sz = fits.write_stream("ftp://user:passwd@myhost.jp/home/user/foo.fits.gz");
```

In this way the argument can include the username and password used to access the FTP server. Omission of the the username and password results in anonymous access.

5.4 Read/Write using pipe-connections to make commands (e.g., compression tools)

If you set `"-"` to the arguments of `read_stream()` and `write_stream()`, they will open stdin and stdout, respectively. Therefore, pipe-connections can be used to read or write a FITS file via any special network or to store them with any special compression format if the commands for the special network/compression support stdin and stdout.

You can use `access_stream()` and `accessf_stream()` (§13.3.4) to access files via some command-line tools. In this case you need to set commands that include path names in the style of open() of Perl to the argument of the member function.

The following example reads a special compressed file using the funpack command.

```
sz = fits.access_stream("funpack -S foo0.fits.fz |");
```

The next example is used to write a special compressed file.

```
sz = fits.access_stream("| fpack - foo1.fits.fz");
```

Examples in §13.3.4 show the case of reading FITS using HTTP over SSL.

5.5 Basis of accessing headers

Note that the formation of the API directly reflects the structure of FITS files. For example, write the following to read the value of `TELESCOP` in Primary header.

```
printf("TELESCOP = %s\n", fits.hdu("Primary").header("TELESCOP").svalue());
```

You can use a string to specify an HDU like above example, and an integer number is also allowed such as "`.hdu(0L)`".

Reading values uses the following: `dvalue()`, `lvalue()`, `llvalue()`, `bvalue()`, and `svalue()` (§13.4.4–). These functions correspond to the double, long, long long, bool, and const char * types, respectively.

Member functions, including `assign()`, and `assign_comment()`, are used to write data, add a new record to a header, and to update values and comments (§13.4.9–).

```
fits.hdu("Primary").header("TELESCOP").assign("HST")
                                      .assign_comment("Telescope Name");
fits.hdu("Primary").headerf("OBJECT%d",n).assignf("%s-%d",obj[i],j)
                    .assignf_comment("Name of the object No.%d",n);
```

A convenient feature is that the same style as `printf()` of libc can be used anywhere. Adding commentary records such as HISTORY is simple. A member function header_append needs two arguments, as follows.

```
fits.hdu("Primary").header_append("HISTORY","step-0: done.");
```

To confirm existence of a keyword in a FITS header and to test the value of its record, you can use header_value_length() member function (§**??**).

```
if ( 0 < fits.hdu("Primary").header_value_length("TELESCOP") ) {
    /* OK */
}
```

This member function returns length of raw value string (including "'") of the header record when given keyword exists. A negative value will be returned if given keyword is not found.

When required keyword and value are found, You might want to know the type of the value. Using type() member function will give the answer (§13.4.13).

```
int r_type = fits.hdu("Primary").header("EQUINOX").type();
if ( r_type == FITS::DOUBLE_T ) {
  /* real */
} else if ( r_type == FITS::LONGLONG_T ) {
  /* integer */
} else if ( r_type == FITS::DOUBLECOMPLEX_T ) {
  /* complex number */
} else if ( r_type == FITS::BOOL_T ) {
  /* logical (T or F) */
} else if ( r_type == FITS::STRING_T ) {
  /* string */
} else {
  /* error */
}
```

The Primary HDU is always treated as an image and hence `fits.hdu(...)` can be replaced with `fits.image(...)`.

See also `sample/read_header.cc` in the SFITSIO source package.

5.6 Keyword search in headers (POSIX Extended Regular Expression)

Keyword searches with regular expressions are available for manipulating FITS headers, including the WCS. The following code is an example of searching the header records in a primary HDU of which the keyword begins with CRVAL1 or CRVAL2. The keywords and raw values are displayed using `header( ... ).keyword()` and `header( ... ).value()`(§13.4.17).

```
fits_image &primary = fits.image("Primary");
long i = 0;
while ( 0 <= (i=primary.header_regmatch(i,"^CRVAL[1-2]")) ) {
    printf("%s = %s\n",primary.header(i).keyword(),
                       primary.header(i).value());
    i++;
}
```

In the example "`fits_image &primary = ...`" is a variable, called a reference or alias, and introduced in C++. This MACRO-like simple mechanism provides for the definition using another name. Using a reference can shorten the code. More details are available in §6.3.

5.7 Editing headers

Member functions that initialize the header (§13.4.25), add (§13.4.27), insert (§13.4.29), and delete (§13.4.31) the records of the header are available for use. The functions used to edit headers can deal with many records at one time via structures (See the example in §5.9).

The following code shows an example of all the content of the header records in one FITS being copied to another.

```
fits_out.image("Primary")
        .header_append_records( fits_in.image("Primary").header() );
```

The `.header()` with no argument represents the entire header. In the same manner it can be provided to the member functions of `header_init()` (§13.4.25) and `header_insert_records()` (§13.4.29).

To copy only a single header record and not all the content write the following.

```
fits_out.image("Primary")
        .header_append( fits_in.image("Primary").header("TELESCOP") );
```

5.8 Accessing image data

The dimensions of the pixels can be obtained using `col_length()`, `row_length`, and `layer_length()` (§13.6.7 -).

```
fits_image &primary = fits.image("Primary");
printf("Num. of Columns: %ld\n",primary.col_length());
printf("Num. of Rows   : %ld\n",primary.row_length());
printf("Num. of Layers : %ld\n",primary.layer_length());
```

Each of `dvalue()`, `lvalue()`, and `llvalue()` can be used to read and `assign()` to write (§13.6.10 -). These functions transform the values according to the values of `BZERO` or `BSCALE` in the header.

The function `dvalue()` reads the value in type `double` independently of the type of image data. The coordinate value begins from 0.

```
double pixel_val;
pixel_val = primary.dvalue(x,y,z);   /* Read */
primary.assign(pixel_val,x1,y1,z1);  /* Write */
```

Of course, integer values can be used in read and write operations. The functions `lvalue()` and `llvalue()` return `long` and `long long` values, respectively.

```
long pixel_val;
pixel_val = primary.lvalue(x,y,z);   /* Read */
primary.assign(pixel_val,x1,y1,z1);  /* Write */
```

`dvalue()` and `assign()` are high-level APIs in SFITSIO. They are suitable for safety and low-cost programming, but overheads with calling them are not small. We will explain high-speed access of image pixels later on.

5.9 Creating new FITS image

This is an example of creating a new image file using data obtained by the CASSIOPEIA telescope. The sample programs of `create_image.cc` and `create_image_and_header.cc` can be found in the `sample` directory of the distributed package.

The member function `append_image()` (§13.3.16) is used to create an image HDU. An example is shown below where the type of data is a 1024×1024 array of `double`.

```
fitscc fits;
fits::header_def defs[] = { {"TELESCOP", "'CASSIOPEIA'", "Telescope name"},
                            {"OBSERVAT", "'NAOJ'", "Observatory name"},
                            {"RA",       "", "[deg] Target position"},
                            {"DEC",      "", "[deg] Target position"},
                            {"COMMENT",  "------------------------------"},
                            {NULL} };
/* Create Image HDU (Primary) */
fits.append_image("Primary",0, FITS::DOUBLE_T,1024,1024);
/* Initialize header */
fits.image("Primary").header_init(defs);
```

It's ready to be used. With SFITSIO header keywords can be prepared as a structure. In addition, many header records can be assigned at once using member functions such as `header_init()` (§13.4.25).

The function `image().assign()` can be used to set the pixel values. If necessary the values of `BZERO` and `BSCALE` can be set with `image().assign_bzero()` and `image().assign_bscale()` (§13.6.14 -), respectively, and the pixel values initialized to zero with `image().fill()` (§13.6.28).

```
fits_image &primary = fits.image("Primary");
primary.assign_bzero(32768);
primary.assign_bscale(1);
primary.fill(0);
```

The FITS template file is useful to create newly defined FITS files (§9, §13.3.5). A template file is a text file that can be used to define the content of FITS with somewhat lenient syntax like FITS headers, which can be used to create a new FITS file (object) without any data. See also `tools/create_from_template.cc` in SFITSIO source package.

5.10 Copying and pasting image data

The member functions of `copyf()` and `pastef()` can be used to copy and paste rectangular regions using IDL/Python-like expression (§13.6.34 -). The following code shows how the region from the coordinates (0, 0) through to (99, 99) can be copied to the coordinate (100, 100).

```
fits_image &primary = fits.image("Primary");
fits_image copy_buf;
primary.copyf(&copy_buf, "0:99, 0:99");
primary.pastef(copy_buf, "100:*, 100:*");
```

The object `fits_image` created by the programmer (=`copy_buf`) becomes the copy buffer. The Copy and Paste will be carried out by the member function applied to the object `copy_buf`. The number of objects is unlimited. Copying and pasting between objects is easy.

Rather than `pastef()` the functions of `addf()`, `subtractf()`, `multiplyf()`, and `dividef()` provide for the addition, subtraction, multiplication, and division operations (between objects), respectively (§13.6.29 -).

The object `copy_buf` used as the copy buffer stated in the above example can be applied in all the member functions in §13.6. For example, `copy_buf` can read, write, and edit values as follows.

```
v = copy_buf.dvalue(x0,y0);
copy_buf.assign(v,x1,y1);
```

The object `copy_buf` acts as an image HDU that is independent of the `fits` object (the FITSCC class object).

The object `copy_buf` can be saved in a FITS file. With the case below programmers create a new FITSCC class object, register the content of the object `copy_buf`, and then save the new object in a file.

```
fitscc new_fits;
new_fits.append_image("Primary",0, FITS::DOUBLE_T,0);
new_fits.image("Primary").swap(copy_buf);
new_fits.write_stream("copy_buffer.fits");
```

This program creates a primary HDU without any image data in a new object `new_fits` (the primary HDU is created using `new_fits.image("Primary")` unless it already exists) and exchanges the content with that of `copy_buf`. Although `new_fits.append_image(copy_buf);` can be used instead of the member function `swap()`, `new_fits.append_image(copy_buf)` requires twice the memory that swap() does and therefore the recommendation is not to use it with large amounts of image data.

5.11 Type conversion and fast access to image data

Fast access to the internal data of an object using pointer variables is supported. Note that the addresses of the data array will change when the size of pixels is changed using member functions such as `image().resize()` (§13.6.22).

To ensure fast access it is recommended to have internal data converted beforehand in order to avoid any conversions using the values of `BZERO` and `BSCALE` when reading data. For conversions use the member function `image().convert_type()` (§13.6.13).

```
fits_image &primary = fits.image("Primary");
primary.convert_type(FITS::FLOAT_T);
```

In this way any type of data can be converted to type `float` in which `BZERO` and `BSCALE` equal 0 and 1. In addition, the address of the data can be obtained using the member function `image().float_t_ptr()`, `image().double_t_ptr()`, etc. (§13.7.2) for each of the internal data types.

```
fits::float_t *ptr:
ptr = primary.float_t_ptr();
```

You can now access the internal data using "`ptr[x + y * col_length]`".

5.12 ■要翻訳■ Hints for development of full-scale analysis tools

§1.3.5 で述べたように，速度が重視される本格的な画像解析ツールを開発する場合には，SLLIB の単純な実装による API を使ってコーディングを行なう事で，全体的なパフォーマンスアップが可能です.

次に SLLIB ベースの解析を行なう手順を述べます．考え方としては，IDL や Python+numpy の場合と同様，FITS の画像データを単純な配列オブジェクトとしてプロセッシングを行なうというものです.

まず，FITS ファイルを読み，データ型を float 型または double 型に変換します.

```
fitscc fits;
fits.read_stream("largefile.fits");
fits_image &prim = fits.image(0L);
prim.convert_type(FITS::FLOAT_T);
```

次に，mdarray_float クラスで参照を取得します.

```
mdarray_float &prim_arr = prim.float_array();
```

この後, 配列に対してはSLLIBのmdarrayクラスのAPIを使い, FITSヘッダに対してはSFITSIOのAPIを使います. 例えば, ゲインをかけて画像の一部の統計をとるには次のように書けます.

```
double gain = prim.header("GAIN").dvalue();
prim_arr *= gain;
/* get mean, variance, skewness, kurtosis */
mdarray_double moment = md_moment(prim_arr.sectionf("2:21, *"), false,
                                  NULL, NULL);
moment.dprint();
```

APIの種類が増えて面倒だと思われるかもしれませんが, その心配は不要です. なぜなら, SFITSIOの画像に関するAPIとSLLIBの配列に関するAPIとは, ほとんど同じ仕様だからです. 例えば, §5.10で紹介したSFITSIOの `copyf()`, `pastef()`, `addf()` 等は “全く同じ仕様の” “全く同じ名前の” メンバ関数がmdarrayクラスにも存在します. この「そっくり」さは, ヘッダファイル `sli/fits_image.h` と `sli/mdarray_float.h` とを比較してみると確認できます.

ここから先の情報については, SLLIBのマニュアルをご覧ください.

5.13 Collaborations with libwcs of WCSTools

WCSTools[27] is a C library for use in the efficient handling of astronomical data. It can be used for processing within the WCS(World Coordinate System), accessing FITS files, and searching important star catalogs. Mr. Douglas J. Mink at SAO is responsible for the development of this library.

Here, we will provide an an example and explain how you can easily handle WCS using SFITSIO and libwcs.

First, you have to write the include declarations.

```
#include <sli/fitscc.h>
#include <math.h>
#include <libwcs/wcs.h>
using namespace sli;
```

Next, we will show the `main()` function. First, the FITS file ‘`foo.fits.gz`’ is read, and an object of the `WorldCoor` structure created, which is pointed by the `wcs` pointer variable, from the FITS header of the Primary HDU. In this step the `header_formatted_string()` member function of SFITSIO is useful in that it can be used to obtain a string which contains all the header records of an HDU(See §13.4.35). The next step involves the WCS conversion: the function `pix2wcs()` converts pixel coordinates to celestial coordinates, whereas the function `wcs2pix()` converts celestial coordinates to pixel coordinates. The last piece of code frees up the memory used by `wcs`.

[27] http://tdc-www.harvard.edu/wcstools/

```
int main( int argc, char *argv[] )
{
    fitscc fits;                                        /* FITS object */
    struct WorldCoor *wcs;                              /* WCS structure */
    double lon,lat, x,y, v;
    int off;

    /* read all data from fits file */
    fits.read_stream("foo.fits.gz");

    /* create alias 'pri' to Primary HDU */
    fits_image &pri = fits.image("Primary");

    /* initialize wcs structure */
    wcs = wcsinitn(pri.header_formatted_string(), NULL);

    /* convert pix -> wcs */
    x = 1.0;  y = 1.0;
    pix2wcs(wcs, x, y, &lon, &lat);
    printf("ra=%.8f dec=%.8f\n",lon,lat);

    /* convert wcs -> pix */
    wcs2pix(wcs, lon, lat, &x, &y, &off);
    printf("x=%.8f y=%.8f\n",x,y);

    /* read value of pixel (1-indexed) */
    v = pri.dvalue((long)floor(x-0.5),(long)floor(y-0.5));
    printf("value=%.15g\n",v);

    /* free wcs structure */
    wcsfree(wcs);

    return 0;
}
```

With your actual code the validity of the `WorldCoor` object should be checked using `iswcs()`. The function `iswcs()` returns 1 or 0, which respectively indicate whether the object is valid or invalid.

If you want to know more about libwcs read the header files of WCSTools package, e.g., `wcs.h`, etc. The header files of WCSTools sufficiently explain the use of powerful APIs, and users can therefore only use these APIs after having read them.

See also `sample_wcs/wcs_test.cc` in the SFITSIO source package.

5.14 Collaborations with WCSLIB

WCSLIB[28)] is a library that was developed by Ph. D. Mark Calabretta to use in manipulating the World Coordinate System (WCS). The combination of SFITSIO with WCSLIB makes it easy to manipulate WCS. This subsection provides a simple example of SFITSIO being combined with WCSLIB. The example program obtains specific celestial coordinates that correspond to pixels included in a FITS file (read to the object `in_fits`) with a WCS header. In addition, the program also obtains the value of the pixels in a new FITS file (in the object `out_fits`) with a WCS header.

First, you have to include header files like this:

[28)] http://www.atnf.csiro.au/people/mcalabre/WCS/WCSLIB

```
#include <sli/fitscc.h>
#include <sli/tstring.h>
#include <wcshdr.h>
#include <wcs.h>
```

Create an object `out_fits` for output to a file that is described as below using a function, such as main.

```
    fitscc out_fits;
    struct *wcs_out;
    tstring headerall;
    int status=0, nrecords, relax=1, nreject, nwcs, ctrl=0, anynul;
    out_fits.append_image("Primary",0,
                          FITS::FLOAT_T,1024,1024); /* 1024x1024 FLOAT image */
    fits_image &outfitspri = outfits.image("Primary");
    outfitspri.header("RADESYS").assign("FK5").assign_comment("Coordinate System");
    outfitspri.header("EQUINOX").assign(2000.0).assign_comment("Equinox");
    outfitspri.header("CTYPE1").assign("RA---TAN"); /* Tangential projection */
    outfitspri.header("CTYPE2").assign("DEC--TAN"); /* Tangential projection */
    /* Pixel coordination of the reference point */
    outfitspri.header("CRPIX1").assign(512.5);
    /* Pixel coordination of the reference point */
    outfitspri.header("CRPIX2").assign(512.5);
    /* Right ascension (RA) of the reference point */
    outfitspri.header("CRVAL1").assign(0.0);
    /* Declination (D) of the reference point */
    outfitspri.header("CRVAL2").assign(0.0);
    /* Increment of the coordinate (RA is increased in the right direction) */
    outfitspri.header("CDELT1").assign(-0.01);
    /* Increment of the coordinate (D is increased in the upward direction) */
    outfitspri.header("CDELT2").assign(0.01);
    headerall = outfitspri.header_formatted_string();
    nrecords = headerall.length()/80;
    wcspih((char*)headerall.cstr(), nrecords, relax, ctrl, &nreject, &nwcs, &wcs_out);
    wcsprt(wcs_out);
```

Here we create an output file, define a WCS header, and import values into the structure `wcs_out`. The function `wcspih()` loads a WCS header into the structure(s). The function requires one string that is composed of all the headers related to WCS in the first argument. The string should be a byte image in a FITS file, but the member function `header_formatted_string()` (§13.4.35) obtains the byte image of the header as an object of the class tstring (the tstring class is described in APPENDIX3 (§16)). The program then displays the content of the structure(s) using `wcsprt()` at the end.

In the same manner the program loads the WCS header in the object `in_fits` into the structure `wcs_in`.

```
    fitscc in_fits;
    struct wcsprm *wcs_in;
    in_fits.read_stream("user_image_file.fits");
    headerall = in_fits.image("Primary").header_formatted_string();
    nrecords = headerall.length()/80;
    wcspih((char*)headerall.cstr(), nrecords, relax, ctrl, &nreject, &nwcs, &wcs_in);
    wcsprt(wcs_in);
```

This code obtains celestial coordinates `(world[0][0], world[0][1])` corresponding to specific pixel coordinates `(x_out, y_out)` in the object `in_fits`, calculates where the obtained coordinates correspond to the pixel coordinates in the object `out_fits`, and then substitutes the calculated

coordinates into (pixcrc_in[0][0], pixcrc_in[0][1]).

```
double pixcrd_in[1][2], pixcrd_out[1][2], imgcrd[1][2];
double phi[1], theta[1], world[1][2];
double x_out, y_out, x_in, y_in;
pixcrd_out[0][0] = (double)x_out;
pixcrd_out[0][1] = (double)y_out;
wcsp2s(wcs_out, 1, 2, pixcrd_out[0], imgcrd[0], phi, theta, world[0], &status);
wcss2p(wcs_in, 1, 2, world[0], phi, theta, imgcrd[0], pixcrd_in[0], &status);
x_in = (float)pixcrd_in[0][0];
y_in = (float)pixcrd_in[0][1];
```

The function wcsp2s() transforms the pixel coordinates to the world coordinates while the function wcss2p() the world coordinates to pixel coordinates. The functions wcspih(), wcsprt(), wcsp2s(), and ŵcss2p() were used in the program described above. Knowledge of these functions enables you to do WCS related calculations. In addition, wcsset() and wcs_errmsg() are also useful functions. The former can be used to reset the structure wcsset(wcs_out) while the latter can be used to obtain the string of error messages as wcs_errmsg[status].

5.15 Accessing ASCII and binary tables

The same member functions in SFITSIO manipulate both ASCII and binary tables.

First of all, we will show you how to check the size of the table. It can be obtained using the member functions table().row_length() (§13.8.4) and table().col_length() (§13.8.3). The following provides an example of how to display the size of binary table the "EVENT".

```
printf("Num. of Columns : %ld\n",fits.table("EVENT").col_length());
printf("Num. of Rows    : %ld\n",fits.table("EVENT").row_length());
```

You can access table data in the similar style to accessing FITS headers, such as fits.table(HDU name).col(column name). .... If the HDU or column does not have a name an index number beginning with 0 can be specified.

Each of dvalue(), lvalue(), llvalue(), bvalue(), and svalue can be used to read the value, and assign() to write it (§13.8.20 -). These functions also automatically convert the values using the values TZEROn and TSCALn in the header.

The function dvalue() can be used to read the columns of any type of data as type double. The row number starts with 0. The following provides an example of how to read/write values of the column "TIME" in the binary table "EVENT".

```
double val;
val = fits.table("EVENT").col("TIME").dvalue(row_index0);  /* Read */
fits.table("EVENT").col("TIME").assign(val,row_index1);    /* Write */
```

As shown in the example the index of the row is specified when read and the value and index in this order when written in the arguments.

You can read and write both integer and Boolean values. The functions lvalue(), llvalue(), and bvalue() return values of the long, long long, and Boolean type, respectively. To read string type values use svalue(). The function svalue() converts values into a string, which is formatted according to what is specified by TDISPn in the header if the value is a number.

```
const char *sval;
sval = fits.table("EVENT").col("TIME").svalue(row_index0);  /* Read */
printf("%s\n",sval);                                        /* To stdout */
```

A column can have multiple elements; for example TFORMn = '8J' means that the column has eight elements. The number of the elements can be obtained using table(...).col(...).elem_length()

(§13.8.13). The element to be read from the column can be specified using the second argument of each of `dvalue()`, `lvalue()`, `llvalue()`, `bvalue()`, and `svalue()`. When written, the same can be specified using the third argument of `assign()`. The following code is an example of how to display all the elements of the column "STATUS" and for every row.

```
long i, j, nrow, nel;
const char *sval;
nrow = fits.table("EVENT").row_length();                    /* Number of rows */
nel = fits.table("EVENT").col("STATUS").elem_length();      /* Number of elements */
for ( i=0 ; i < nrow ; i++ ) {
    for ( j=0 ; j < nel ; j++ ) {
        sval = fits.table("EVENT").col("STATUS").svalue(i,j);
        printf("%s ",sval);
    }
    printf("\n");
}
```

5.16 Creating binary tables

The following example shows the case of creating a binary table for data from the astronomy satellite of ASTRO-X. Sample code, `create_bintable.cc`, which can be compiled, is available in the directory `sample` of the distribution package.

It creates a table composed of the three columns of (double, 32-bit integer, and string).

```
fitscc fits;
const fits::table_def def[] = {
  /* ttype,comment,         talas,telem,tunit,comment,                */
  /*                                          tdisp,  tform, tdim   */
  { "TIME","satellite time",   "",  "", "s","", "F16.3", "1D", "" },
  { "STATUS","status",         "",  "", "","",  "",      "8J", "" },
  { "NAME","",                 "",  "", "","",  "",  "128A16", "(4,2)" },
  { NULL }
};
/* Create a binary table (name of HDU is "EVENT") */
fits.append_table("EVENT",0, def);
```

First, it prepares for defining the columns with structures, and then creates a binary table using the member function `append_table()` (§13.3.17). Binary tables cannot be primary HDUs with FITS definitions, and hence a primary HDU without any image data is automatically created.

Second, the rows of the table can be allocated as follows.

```
fits.table("EVENT").resize_rows(256);
```

The FITS template file is useful to create newly defined FITS files (§9, §13.3.5). A template file is a text file that can be used to define the content of FITS with somewhat lenient syntax like FITS headers, which can be used to create a new FITS file (object) without any data. See also `tools/create_from_template.cc` in SFITSIO source package.

5.17 Creating ASCII tables

The method used to create ASCII tables is similar to that with binary tables. However, care should be taken when you specify `tdisp` and `tform` in the structure. When creating ASCII tables the strings to be written in `TFORM`n of the FITS file are specified by `tdisp` in the structure, and the width of the strings by `tform` in the structure in the form of "n`A`".

It creates a table composed of three columns.

```
fitscc fits;
const fits::table_def def[] = {
  /* ttype,comment,        talas,telem, tunit,comment, tdisp,  tform */
  { "PK","PK number",        "",   "",  "","",         "A9",   "9A" },
  { "RAH","Hours RA",        "",   "",  "h","",        "I2",   "3A" },
  { "RAM","Minutes RA",      "",   "",  "min","",      "F5.2", "6A" },
  { NULL }
};
/* Create an ASCII table (name of HDU is "PLN") */
fits.append_table("PLN",0, def, true);
```

First, prepare the definition of columns with structures, and then create an ASCII table using the member function `append_table()` (§13.3.17) and `true` as the final argument. ASCII tables cannot be primary HDUs with FITS definitions, and hence a primary HDU without any image data is automatically created.

Second, the rows of the table can be allocated as follows.

```
fits.table("PLN").resize_rows(256);
```

See also `sample/create_asciitable.cc` in the SFITSIO source package.

5.18 Editing and importing ASCII and binary tables

SFITSIO enables not only the addition, insertion, and deletion of columns (§13.8.46 -) and rows (§13.8.52) but also the other ASCII tables or binary tables to be imported (§13.8.58).

The following code is an example of how to copy the as-is content of the column "DEC" in one FITS table to another.

```
fits_out.table("EVENT").append_a_col( fits_in.table("SRC").col("DEC") );
```

The next example shows how to combine the two tables.

```
long orow_length = fits_out.table("EVENT").row_length();
fits_out.table("EVENT")
        .resize_rows( orow_length + fits_in.table("SRC").row_length() );
fits_out.table("EVENT")
        .import_rows( orow_length, true, fits_in.table("SRC") );
```

First, the member function `resize_rows()` increases the number of rows up to the sum of the two tables. Last, the member function `import_rows()` (§13.8.58) pastes the entire content of the table `SRC` to the rear margin of the table `EVENT`. The arguments passed to `import_rows()` provide the number of rows where the paste operation will begin, whether it pastes the column to which the name matches, and the source of the table.

Member function `table().col().import()` (§13.8.62) can be used to import units of a column.

5.19 Editing of HDU

It is easy to merge the images of multiple FITS files and ASCII or binary tables into one FITS and then delete the unnecessary HDUs using SFITSIO. Providing `fits.image("FOO")` or `fits.table("BAR")` as the argument of member functions `.append_image()`, `.append_table()` and so on provides for easy additions (§13.3.16, §13.3.17) and insertions (§13.3.18, §13.3.19) of Image HDUs and binary (or ASCII) table HDUs.

The following code is an example of adding an as-is table `EVENT` in the object `fits_in` to the object `fits_out`.

```
fits_out.append_table( fits_in.table("EVENT") );
```

The next example involves the deletion of the HDU name `FOO` of an object `fits_out`.

```
fits_out.erase("FOO");
```

5.20 ■要翻訳■ヘッダだけの高速読み取り (ディスクベースの FITS I/O)

データセンターなどでは，大量の FITS ファイルを管理するために，FITS ヘッダ「だけ」を高速に読み取らなければならない事があります.

FITS ヘッダを管理するための fits_header クラスが持つ read_stream() メンバ関数 (§13.5.1) を使えば，Data Unit を読まずにファイルへのアクセスを終了する事が可能です．もちろん，圧縮ファイルでも高速なアクセスが可能です.

コードの書き方は簡単で，まず digeststreamio クラスでファイルをオープンし，`read_stream()` でヘッダ部分のみ読み取ります.

```
#include <sli/fitscc.h>
#include <sli/digeststreamio.h>
using namespace sli;

int main()
{
    digeststreamio f_in;
    fits_header hdr;

    /* ヘッダ部分のみ読む */
    f_in.open("r", "my_image.fits.gz");
    hdr.read_stream(f_in);

    /* ヘッダの内容を表示 */
    printf("gain = %g\n", hdr.at("GAIN").dvalue());
    printf("[all string]\n");
    printf("%s\n", hdr.formatted_string());
```

Data Unit を読み飛ばす場合は，次のように skip_data_stream() メンバ関数 (§13.5.3) を使います．ただしここは圧縮ファイルだと速度は出ません.

```
    hdr.skip_data_stream(f_in);
```

もちろん，`hdr.skip_data_stream(f_in)` のかわりに，`f_in.read(...)` などを使って Data Unit を読み出す事もできます[29].

次の HDU のヘッダも同様に読み取る事ができます.

```
    hdr.read_stream(f_in);
```

必要な部分が読み取れたら，いつでもストリームをクローズしてかまいません.

```
    f_in.close();
```

ソースパッケージに含まれるヘッダの高速閲覧ツール `tools/hv.cc` も参考にしてください.

[29] この場合は，ヘッダの内容を使って Data Unit のフォーマットを調べる必要があります.

6 Things to know before using SFITSIO

You will be using a C++ compiler because SFITSIO is a C++ library. C++ is upwardly compatible with C, and hence basically you can write code in the same style as C.

However, there are a few (but not very difficult) things you need to know before using SFITSIO. Incompatibilities between C and C++ accompanied by the extensions of C++ do exist but which easily and conveniently extend the functionality of C++. We will describe them in this section.

6.1 Namespace

Namespace is one of the things that was introduced in C++. It is used to avoid the problem of different programmers writing the same name of functions or types. It is similar to a category name. SLLIB and SFITSIO assign their own namespace as "sli". If you use some kind of class you will need to put `sli::` at the top, for example "`sli::stdstreamio sio;`", in order for the code to be written in the proper form.

If you will be mainly using SFITSIO you may not want to have to write `sli::` every time. In this case follow the style below to skip having to use `sli::` throughout the rest of the code.

```
using namespace sli;
```

The examples in this manual omit `sli::`. For C++ beginners it may be easier to keep in mind that once you have written "`#include <...>`" then you should write "`using namespace sli;`".

6.2 NULL and 0

With many processing systems NULL with C is defined as

```
define NULL ((void*)0)
```

But NULL with C++ is defined as

```
define NULL (0).
```

The reason why `NULL` is `0` in C++ is that the check of the type of pointer variables with C++ is stricter than that with C. For instance, suppose that there are two pointer variables, `char *ptr0;` `void *ptr1;` and then an error occurs when you try to substitute `ptr0` with `ptr1`. Zero (`0`) is defined as a "nowhere address" so that `ptr0 = 0;` does not generate an error. This is exactly why `NULL` is `0` with C++.

With C++ a member function of a class may have the same name but different arguments. For example,

```
int foo( int a );
int foo( char *p );
```

If both `hoge.foo(NULL)` or `hoge.foo(0)` exist the compiler will not be able to determine which function should be used. In this case it is necessary to explicitly indicate the type. That is

```
hoge.foo((char *)NULL);
hoge.foo((int)0);
```

With C++ it is safer to remember that "**cast if use** `NULL` **or** `0`". Another way is to cast `NULL` out and "**always cast** `0`".

6.3 Reference

It is too much work to write `fits.hdu("Primary")` or `fits.image("Primary")` with every access to the header or the image. "**Reference**" therefore makes this shorter. Reference, also called

"**Alias**", creates an alias of a variable or an object but in simple terms. Although Macro may be used for aliases for variables or objects use of references allows for smarter scripting.

The reference introduced in C++ is a new type that is similar to the pointer type. In fact, **the functions of the reference are simpler than that of the pointer type**. Accordingly, **its usage is simple too**. For example, creating the reference "`aref`" of the variable `int a;` involves

```
int &aref = a;
```

A reference, or so called "Alias", acts exactly the same way. For instance,
`aref = 10;`
then, `10` is assigned to `a`, and
`int b = aref;`
substitutes the value of `b` with `a`.
`int *p = &aref;`
substitutes `p` with the address of `a`.

As described above reference is a simple way of providing an alias for a variable. It is not as complicated as pointer variables with their many `*`. Reference cannot have `NULL` because it is impossible to create any references without their actual existence.

When using SFITSIO you can mostly use a reference to copy it after returning from a member function to another reference variable that the user creates. For instance, the member function `fits.hdu()` returns a reference to "`fits_hdu &`". If a user creates a reference of the same class and copies the return value in the receipt side it can then be a substitution for `fits.hdu("Primary")` or `fits.image("Primary")`, thus allowing the user to write less code. The following code provides an example.

```
fits_hdu &primary = fits.hdu("Primary");
printf("TELESCOP = %s\n",primary.header("TELESCOP").svalue());
```

In other words,

```
fits_image &primary = fits.image("Primary");
printf("TELESCOP = %s\n",primary.header("TELESCOP").svalue());
```

Please note that using a reference without substituting the value in the declaration will result in an error. This is because "Alias" means "another name". An "Alias" being used without actually existing results in an error.

6.4 Try & catch

You can skip this part unless you intend to write rather heavy code.

The syntax `try{}` and `catch(){}` are used to deal with the "Exception" introduced in C++. SFITSIO results in an "Exception" when a critical problem, such as "**memory allocation failed**", occurs with arguments provided by users. An "Exception" can be handled using `try{}` and `catch(){}` in the user's code. With SFITSIO any exceptions that have occurred always result in a message of the `err_rec` type. Deal with it by writing

```
try {
    /* Change the buffer size of image */
    fits.image("Primary").resize(0,very_big_size);
    return_status = 0;
}
catch ( err_rec msg ) {
    fprintf(stderr,"[EXCEPTION] function=[%s::%s] message=[%s]\n",
            msg.class_name, msg.func_name, msg.message);
    return_status = -1;
}
```

If an exception occurs without use of `try{}` and `catch ( err_rec msg ){}` the function `abort()` will be called and the program will terminate.

Generally, any such critical error in most cases will abort the program. If you do not have any problem with using the program called abort() it will not be necessary to use `try{}` and `catch( err_rec msg ){}`.

7 Tips on appropriately handling FITS with SFITSIO

7.1 Policies with coding in accordance with the memory management scheme of SFITSIO and hardware selection

The classes provided by SFITSIO and SLLIB have a very simple structure with no internal reference counter[30)] and always one (set of) internal buffer(s) per object. This then means that the memory area required for an assignment operation is immediately allocated upon assignment to a new object using the `init()` member function or the "`=`" operator.

Memory mapping is also very simple, with both the image buffer for the `fits_image` class and the column data buffer for the `fits_table_col` class *always being allocated as one-dimensional arrays.*

Programmers can therefore write code using the address of the internal buffer within an object in much the same way as a memory area with the C language (any such handling should be limited to cases where a greater level of performance is required, though). However, *it entails significant overhead with the execution of operations such as having to remove the middle part of the data and moving the data that follows forward in thereby filling in the gap, or inserting data in the middle of the buffer, and even if they are operated within the memory.* Any such operations should be reduced as much as possible by writing the proper code and selecting the proper member functions in thereby ensuring good performance.

Next, hardware. It goes without saying that an adequate amount of memory is required for the size of FITS files that the application will be handling via use of SFITSIO. I would recommend installing additional memory in thereby ensuring an average amount of free memory space that matches the largest size FITS files to be handled by the application.

SFITSIO also adopts the simple policy of re-allocating memory where the responsibility is left to the OS and programmers; it always calls `realloc()` internally if it identifies `resize()` that the member function etc. requires a change in the size of the buffer. It is up to programmers whether to reduce the frequency of `realloc()` calls by consistently having enough of a buffer size or to pursue memory saving by allowing more calls to `realloc()`.

In addition, there is no member function that returns the address of a dynamically created object with SFITSIO or SLLIB. This therefore makes `delete` or garbage collector unnecessary except for objects dynamically created by programmers.

7.2 Tricks for faster operation

Tricks for faster operation often depend on the computing architecture used. Furthermore, and because computing architecture changes over time, the quest for faster operations typically remains a perpetual challenge[31)]. Moreover, any excessive pursuit of faster operation could harm the readability and safety of the code. Any performance tuning should therefore only be carried out in view of the forecast future computing architecture and be in proper proportions with other evaluation axes.

That said, the basics for faster operation will remain unchanged, and include reducing the total number of steps in the eventual code of the machine language. I will cover some of what programmers can do to fulfill that purpose while referring to the internal implementation of SFITSIO.

- *Although great care has been taken in accessing objects by name, it still involves relatively large overhead.*

[30)] There is no `static` member function, either.

[31)] An extreme example would involve the technique of rewriting the machine code of the program itself, depending on the conditions, and in order to avoid any conditional branches in the loop. Any such code would become inoperable if the CPU utilized cache memory.

As shown in Question 1 of §1.1 SFITSIO enables you to access the header or a column of the table by name. Because both the header and the table are internally represented by simple object arrays with SFITSIO it has to quickly convert the names into indices in thereby maintaining the level of performance.

SFITSIO creatively utilizes the ctindex class of SLLIB, and which enables fast searches by representing the relationship between the key string and the index as a tree structure (for more details refer to the SLLIB advanced user reference guide).

However, all the effort put into using that algorithm still involves a large amount of overhead when compared to direct indexing. *creativity on the part of programmers* is therefore still required; in order to achieve a higher level of performance programmers must use direct indexing after converting from the name to the index only once, as well as writing code that utilizes references.

- *Reducing the frequency of calls to member functions*

  The overhead involved with calling member functions may cause problems, for example, inside multiply nested loops.

  Better performance can also be effectively achieved by reduce the frequency of member function calls *on the part of programmers*. Creative use of references, as shown in the next example, can achieve this.

```
const fits_table &tbl = fits.table("FOO");
const fits_table_col &col = tbl.col("BAR");
for ( j=0 ; j < tbl.row_length() ; j++ ) {
    printf("[%s]", col.svalue(j));
}
```

- *Writing SLLIB-based code for full-scale image analysis tools*

  Recently, large-sized FITS images are quite common, and higher performance of analysis is required. In such a situation, you can use APIs provided by SLLIB's array object (mdarray_float class, etc.) to obtain higher performance on average, since the implementation of SLLIB's APIs is simpler than that of SFITSIO. Here is an example:

```
fits.image(0L).convert_type(FITS::FLOAT_T);       /* Convert data type */
mdarray_float &img_array
                 = fits.image(0L).float_array();  /* Alias of array object */
```

  In addition, SLLIB is independent of FITS, therefore, your analysis tools developed with SLLIB are easily applicable to non-FITS image data.

  Both SFITSIO and SLLIB provide various APIs for image processing, which enable a small amount of code to perform area scanning, copy and paste, the four basic arithmetic operations with images and scalar values, and image scanning and operations with statistical functions. However, only SLLIB has some features such as mathematical functions for arrays and operators for arrays, and new packages (header files) for data analysis will be appended to SLLIB in the future. Therefore, we recommend that you to write SLLIB-based code for full-scale image analysis tools.

8 ■要翻訳■ Syntax of Expression for Partial Reading of FITS Files

`read_stream()` または `access_stream()` では (§13.3.1, §13.3.4), 次のコードのように IRAF 風の表記により n 次元の画像やテーブルの一部分だけを読み出す事が可能です.

```
sz = fits.read_stream("image.fits.gz[1:100,*]");
```

SFITSIO では, IRAF/CFITSIO の場合と同様の記法と, SFITSIO で新たに定義した「より論理的な」記法の, 2 通りをサポートしています.

8.1 IRAF/CFITSIO 的な記法

元の FITS ファイル中の複数の HDU の中から, 1 つの HDU を選び, 画像の一部を読み取る指定が可能です. 典型的には次のような形です.

```
sz = fits.read_stream("image.fits.gz[1][1:100,*]");
```

この記法による表現は, SFITSIO 内部でこの後で解説する SFITSIO 公式の記法に変換され, FITS ファイルの部分読みが実行されます. 上記の例の場合, ファイル名を含む引数は「`"image.fits.gz[1[1:100,*]]"`」と変換されます.

この後で解説しますが, 画像の領域指定の部分については, 画像の反転「`"[100:1,*]"`」「`"[-*,*]"`」, 0-indexed を示す丸括弧「`"(0:99,*)"`」も, IRAF/CFITSIO 的な記法でも使う事ができます.

8.2 SFITSIO 公式の論理的な記法

SFITSIO 公式の記法では, より論理的で現代的な表現となっており, 従来できなかった複数の HDU の選択が可能になっています.

8.2.1 文法の一般的解説

- ファイル名直後における, 同じ階層の「`[`」「`]`」のペアは一組だけとし, `[]` の中にセミコロン区切りで複数の HDU に関する指定を書く.

  `myfile.fits[`HDU 指定 `A; `HDU 指定 `B; ... `HDU 指定 `Z]`

- 各 HDU 指定では, 階層が 1 つ下の `[]` か `()` で画像の領域またはテーブルの領域を指定できる.

  `myfile.fits[`HDU 指定 `A[`1-indexed による領域指定`]; ...]`
  `myfile.fits[`HDU 指定 `A(`0-indexed による領域指定`); ...]`

- 画像の領域またはテーブルの領域は, 次元の小さいものから順に記述し, 各次元はカンマで区切る. 要素について, 任意の範囲を指定する場合はコロンを挟んで開始点と終了点を指定し, 全範囲を指定する場合は「`*`」と書いても良い. なお, 指定可能な次元数に制限は無い.

  `myfile.fits[`HDU 指定 `A[`開始点`:`終了点`,*, ...]; ...]`

- HDU のタイプ, HDU の version を指定する場合は「`::`」で区切る.

  `myfile.fits[`HDU タイプ`::`HDU 名`::`HDU バージョン `[...]; ...]`

  HDU のタイプは「`i`」「`b`」「`a`」でそれぞれ Image, バイナリテーブル, ASCII テーブルを意味する.

  「`::`」で区切られた要素が 2 つの場合は, HDU タイプが省略されたものとみなす.

- HDU 指定の直後に「`{`n`}`」で読むべき HDU の数を限定できる.

- 括弧, カンマ, セミコロンの前後の空白は無視される.

- HDU 名は，EXTNAME の値文字列か数字 (常に 0-indexed) で指定する．名前指定の場合，シェルのマッチと同一の動作となる．
- テーブルのカラムは，名前または数字で指定する．複数のカラムを飛び飛びで指定する場合は，セミコロン区切りで記述する．名前指定の場合，シェルのマッチと同一の動作となる．
- 画像およびテーブルの行について，指定範囲の表記「A:B」において $B < A$ の場合，あるいは範囲指定の表記が「-*」の場合，オブジェクト内の要素順は逆転する．

8.2.2 使用例

- HDU 番号が 0,1,4 のもののみ読む.

```
myfile.fits[0;1;4]
```

- HDU 番号 2 のもの以外をすべて読む.

```
myfile.fits[-2]
```

- 文字列マッチ (前方一致) する HDU のみ読む.

```
myfile.fits[EVENT*]
```

- 文字列マッチする HDU のうち，最初の 2 つのみ読む.

```
myfile.fits[EVENT*{2}]
```

- バイナリテーブルのみ全部読む.

```
myfile.fits[b::*::*]
```

- Image HDU の画像の一部分を読む (後者は 0-indexed).

```
myfile.fits[1[1:100,*]]
myfile.fits[1(0:99,*)]
```

- 画像の反転.

```
myfile.fits[1[100:1,*]]
myfile.fits[1[-*,*]]
```

- 三次元 Image の最初の 1 枚だけ読む (0-indexed).

```
myfile.fits[0(*,*,0)]
```

- バイナリテーブルの指定 (行の指定は Image の場合と同様).

```
myfile.fits[EVENT[1:10,1:100]]
myfile.fits[EVENT(0:9,0:99)]
myfile.fits [EVENT [TIME;*TEMP*, 1:100] ]
```

9 Template Files

'Template File' is a text file written with lenient syntax similar to FITS header, and defines a format of new FITS files. SFITSIO can read template files, and creates fitscc objects and FITS files having no data contents. Template files are generally used to manage various FITS formats.

9.1 Creating Image HDU

This template file creates a Primary HDU:

```
#
# Test for Primary only
#
NAXIS2 = 16
NAXIS1 = 16
BITPIX = 32
FMTTYPE = ASTRO-X xxx image format
CHECKSUM =
DATASUM  =
COMMENT ----------------------------------------------------------------
EQUINOX = 2000.0
CTYPE1  = RA---TAN
CTYPE2  = DEC--TAN
CRPIX1  = 0.0
CRPIX2  = 0.0
CRVAL1  = 0.0
CRVAL2  = 0.0
CDELT1  = 0.1
CDELT2  = 0.1
PC1_1   = 1.0
PC1_2   = 0.0
PC2_1   = 0.0
PC2_2   = 1.0
COMMENT ----------------------------------------------------------------
MESSAGE = 'FITS (Flexible Image Transport System) format is &' / In SFITSIO,
CONTINUE  'defined in "Astronomy and Astrophysics", volume 376, &' / this
CONTINUE  'page 359; bibcode: 2001A&A...376..359H' / message is not written
CONTINUE  automatically.
```

In template files, users do not have to fix column positions of keyword, "`=`"[32], value, and comment. In addition, there is no limit of length of text column, and long string values can be written with or without `CONTINUE` keywords. Lines whose first character is "`#`" are ignored by the SFITSIO build-in parser.

Next, we show the FITS header created from above template file.

[32] In CFITSIO, "=" can be omitted. However, SFITSIO does not permit such an omission.

```
SIMPLE  =                    T / conformity to FITS standard
BITPIX  =                   32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                   16 / length of data axis 1
NAXIS2  =                   16 / length of data axis 2
EXTEND  =                    F / possibility of presence of extensions
FMTTYPE = 'ASTRO-X xxx image format' / type of format in FITS file
FTYPEVER=                    0 / version of FMTTYPE definition
CHECKSUM= 'ZlBRfj9OZjAOdj9O'   / HDU checksum : 2012-01-17T06:07:12
DATASUM = '0       '           / data unit checksum : 2012-01-17T06:07:12
COMMENT ----------------------------------------------------------------
EQUINOX =               2000.0 / equinox of celestial coordinate system
CTYPE1  = 'RA---TAN'           / type of celestial system and projection system
CTYPE2  = 'DEC--TAN'           / type of celestial system and projection system
CRPIX1  =                   0.0 / pixel coordinate at reference point
CRPIX2  =                   0.0 / pixel coordinate at reference point
CRVAL1  =                   0.0 / world coordinate at reference point
CRVAL2  =                   0.0 / world coordinate at reference point
CDELT1  =                   0.1 / world coordinate increment at reference point
CDELT2  =                   0.1 / world coordinate increment at reference point
PC1_1   =                   1.0 / matrix of rotation (1,1)
PC1_2   =                   0.0 / matrix of rotation (1,2)
PC2_1   =                   0.0 / matrix of rotation (2,1)
PC2_2   =                   1.0 / matrix of rotation (2,2)
COMMENT ----------------------------------------------------------------
MESSAGE = 'FITS (Flexible Image Transport System) format is defined in &'
CONTINUE  '"Astronomy and Astrophysics", volume 376, page 359; bibcode: &'
CONTINUE  '2001A&A...376..359H&'
CONTINUE  '' / In SFITSIO, this message is not written automatically.
END
```

Template files minimize items that users have to write for definitions of FITS files. For example, `SIMPLE`, `EXTEND`, `NAXIS`, and comments for general FITS keywords are omitted in above template file. However, SFITSIO automatically completes required keywords and comments, and adjusts the order of keywords so that created FITS header satisfies FITS standard.

9.2 Creating Binary Table HDU

Next, we show a template file to create a FITS file including a binary table HDU.

```
#
# Primary
#
FMTTYPE  = 'ASTRO-X XXXX table format'
FTYPEVER = 101
EXTNAME  = 'Primary'
ORIGIN   = 'JAXA'
#
# 2nd HDU : Binary Table
#
XTENSION = BINTABLE
NAXIS2 = 24
PCOUNT = 65536
THEAP  = 32768
EXTNAME = XXXX_TEST
TXFLDKWD = 'TNOTE,TDESC,TLMIN,TLMAX,TDMIN,TDMAX'
 TTYPE#  = TIME
 TALAS#  = DATE
 TFORM#  = 1D
 TDESC#  = 'This is time' / description of this field
 TTYPE#  = COUNTER
 TFORM#  = 8J
 TLMIN#  = 1
 TLMAX#  = 16777216
 TDMIN#  = 0
 TDMAX#  = 0
 TTYPE#  = XNAME
 TFORM#  = 16A
 TTYPE#  = VLA
 TFORM#  = 1PJ(0)
 TNOTE#  = 'You can define variable length array in SFITSIO template&'
 CONTINUE  '' / annotation of this field
```

Descriptions for Primary HDU can be omitted completely. If you want to define some header records in Primary HDU, you can define them before definitions of binary table HDU like this example.

In the definition of second HDU, `XTENSION` keyword must be written first, and `BITPIX`, `NAXIS`, `NAXIS1`, `NAXIS2`, `PCOUNT`, `GCOUNT`, `TFIELDS` can be omitted. Note that `TXFLDKWD` keyword in this example is used to indicate non FITS standard keywords of table columns (fields) such as `TLMIN`n, `TLMAX`n, etc. FITS libraries can know newly defined column keywords by reading `TXFLDKWD` record. See also §11.7 for `TXFLDKWD` convention.

Automatic numbering is available for column definitions such as `TTYPE`n keywords.. When using this feature, "`#`" symbol is written after column keywords in template files like `TTYPE#` in above example. However, note that following case does not give what you want.

```
 TTYPE#  = TIME
 TTYPE#  = COUNTER
 TFORM#  = 1D
 TFORM#  = 8J
```

We show FITS header created from the template file.

```
SIMPLE  =                    T / conformity to FITS standard
BITPIX  =                   16 / number of bits per data pixel
NAXIS   =                    0 / number of data axes
EXTEND  =                    T / possibility of presence of extensions
FMTTYPE = 'ASTRO-X XXXX table format' / type of format in FITS file
FTYPEVER=                  101 / version of FMTTYPE definition
EXTNAME = 'Primary '           / name of this HDU
ORIGIN  = 'JAXA    '           / organization responsible for the data
END
```

```
XTENSION= 'BINTABLE'           / type of extension
BITPIX  =                    8 / number of bits per data element
NAXIS   =                    2 / number of data axes
NAXIS1  =                   64 / width of table in bytes
NAXIS2  =                   24 / number of rows in table
PCOUNT  =                65536 / length of reserved area and heap
GCOUNT  =                    1 / number of groups
TFIELDS =                    4 / number of fields in each row
THEAP   =                32768 / byte offset to heap area
EXTNAME = 'XXXX_TEST'          / name of this HDU
TXFLDKWD= 'TALAS,TDESC,TNOTE,TLMIN,TLMAX,TDMIN,TDMAX' / extended field keywords
TTYPE1  = 'TIME    '           / field name
TALAS1  = 'DATE    '           / aliases of field name
TFORM1  = '1D      '           / data format : 8-byte REAL
TDESC1  = 'This is time'       / description of this field
TTYPE2  = 'COUNTER '           / field name
TFORM2  = '8J      '           / data format : 4-byte INTEGER
TLMIN2  =                    1 / minimum value legally allowed
TLMAX2  =             16777216 / maximum value legally allowed
TDMIN2  =                    0 / minimum data value
TDMAX2  =                    0 / maximum data value
TTYPE3  = 'XNAME   '           / field name
TFORM3  = '16A     '           / data format : STRING
TTYPE4  = 'VLA     '           / field name
TFORM4  = '1PJ(0)  '           / data format : variable length of 4-byte INTEGER
TNOTE4  = 'You can define variable length array in SFITSIO template&'
CONTINUE  '' / annotation of this field
END
```

9.3 Specification of SFITSIO template

1. Template files are written with plain text format having UNIX or DOS newline.

2. Lines whose first character is "`#`" are ignored by SFITSIO parser.

3. A tab character is simply replaced with a white space.

4. Order of keyword, "`=`", value, "`/`", and comment must be the same as that of FITS header, however, there is no rule for position of keyword, etc. Note that record with '8-space keyword' will be created for the line beginning space or tab whose length is 8 or more.

5. There is no limit of length of columns in template files.

6. In template, long string values can be written in a line. Using `CONTINUE` keyword is also permitted. However, the position of "`&`" character for `CONTINUE` convention will not be preserved in FITS header.

7. Quotations for a string value can be omitted only if the value cannot be interpreted as numeric or logical value. However, quotations should not be omitted when using `CONTINUE` convention.

8. When a very long description using `COMMENT` is written in template, multiple `COMMENT` records will be created to store the long description.

9. First keyword must be `XTENSION` for second or later HDUs.

10. The order of keywords in template is not prescribed except `XTENSION` and keywords for table columns (fields).

11. The order of keywords in template is reflected in FITS header, when it does not break FITS standard.

12. `SIMPLE`, `NAXIS`, `EXTEND`, `PCOUNT`, and `GCOUNT` keywords can be omitted in Image HDU.

13. `BITPIX`, `NAXIS`, `NAXIS2`, `PCOUNT`, `GCOUNT`, and `TFIELDS` keywords can be omitted for Binary Table HDU or ASCII Table HDU. `NAXIS1` is also not required for Binary Table.

14. Automatic numbering is applied for keywords with "`#`" suffix of table columns such as `TTYPE#`. Counter increment for this numbering is done when parser detects first keyword having "`#`" suffix in the HDU definition.

15. When comments after "`/`" character are omitted in template, SFITSIO completes the comment using built-in default comments when the record has FITS standard keywords. Blank comments for non FITS standard keywords are also filled for well known keywords.[33)]

16. "`=`" cannot be omitted when creating records of 'keyword `=` value `/` comment' .

17. Keywords of template are case insensitive in current version of SFITSIO, however, we recommend users to write uppercase keywords.

[33)] See APPENDIX2 (§15) for details.

10 Local FITS Extension Compatible with CFITSIO

10.1 Long header value across multiple records

In SFITSIO, as well as in the CFITSIO, in case that a header string cannot be fitted into 1 line of the header record, save by using `CONTINUE` as follows.

```
TELEM6  = 'CREON,SHTOP,FWPOSON,FWPOS_B1,FWPOS_B0,MPOSON,MPOS_B1,MPOS_B0,&'
CONTINUE  'RSTWIDELON,RSTWIDESON,RSTN170ON,RSTN60ON,LWBOOSTON,SWBOOSTON,&'
CONTINUE  'LWBIASON,SWBIASON,CALALON,CALASON,CALBON,SINALON,SINASON&'
CONTINUE  '' / elements in STATUS
```

In case of SFITSIO, this process for extension, i.e., data conversion between a file and an object is carried out at file I/O. Since there are no restriction of string length for the object of the SFITSIO, an API specialized for long value as in the CFITSIO does not exist. That is, users don't need to mind whether there is the `CONTINUE` record or not on the file.

10.2 Array of fixed length strings in binary table

In CFITSIO, assignments such as `TFORM`n `= '120A10'` or, `TFORM`n `= '120A'` AND `TDIM`n `= '(10,12)'` allows users to treat 12 sets of 10-character string in a column. The SFITSIO also supports these Extension of FITS.

Furthermore, in the SFITSIO, an assignment like `TFORM`n `= '120A10'` AND `TDIM`n `= '(6,2)'` allows users to treat a 10-character string as a 6×2 array. This assignment can also be written as `TFORM`n `= '120A'` AND `TDIM`n `= '(10,6,2)'`.

10.3 Writing checksum and datasum

If each header of HDU has `CHECKSUM` or `DATASUM` keyword, SFITSIO automatically writes the CFITSIO-compatible checksum or datasum when saving FITS files. Here is an example:

```
CHECKSUM= 'LNXALNX2LNX8LNX8'   / HDU checksum : 2012-01-16T13:34:03
DATASUM = '2155872383'         / data unit checksum : 2012-01-16T13:34:03
```

Note that the values of `CHECKSUM` and `DATASUM` are basically calculated by only summing up 32-bit integers with treating all byte data in an HDU as binary data that consist of 32-bit integers. Therefore, they will not enough to certify an identity of data. We recommend you to use md5sum for such a purpose.

SFITSIO of current version has no function to test the values of `CHECKSUM` and `DATASUM` records.

11 SFITSIO's Original Extension of FITS

In this section, we mainly explain extension about binary table. The extensions from §11.8 have been defined in 'AKARI' project of ISAS/JAXA to store complicated data into FITS binary table with better readability and to process them efficiently. The AKARI team have discussed these extensions for a long term software development.

11.1 Error Check and Version Management of FITS File by FMTTYPE and FTYPEVER

Shown in §1.2, the 'AKARI' project team defined the TSD format, and discussed how to manage version of TSD or detect errors while loading FITS files in programs developed in the project.

As a result, the project team defined `FMTTYPE` and `FTYPEVER` keywords in the primary header to indicate "What is this FITS file?" and version of structure in a FITS file, respectively. For the value of keyword `FMTTYPE`, a string which defines globally unique data format, i.e., such as "Project Name, Equipment Name, File Type", should be set. The `FTYPEVER` should be set an integer version number.

We recommend using more than three digit number for `FTYPEVER` to express a minor version. Alternatively, a date such as `20080101` can also be used.

SFITSIO supports `FMTTYPE` and `FTYPEVER` keywords in dedicated member functions.

11.2 Distinction of upper and lower case in the header keyword

Current FITS standard does not permit using lower case for header keywords. In SFITSIO, a keyword of the FITS header in upper case and that in lower case are distinguished. For example, following header records can be created.

```
FOO     = 123
Foo     = 456
```

11.3 Long keyword in the header (maximum 54 characters)

Current FITS standard does not permit using keyword having more than 8 characters, therefore, **9th character of a header record should be "=" or white space** except records having 8-space keyword. This means that other cases are not defined about 9th characters in FITS standard.

Therefore, we can create a convention for undefined 9th characters, and this leads to a long keyword convention:

1. When 9th character of a header record is not "=" or white space, the record has a long keyword having more than 8 characters.

2. Long keywords should not have "=" or white space.

This convention keeps readability of FITS header, and does not cause any compatibility problems when handling old FITS files.

Following above convention, the long keyword can be saved simply in SFITSIO:

```
TTYPE12345= 'Mag     '          / column name
TFORM12345= '1D      '          / data format : 8-byte REAL
```

The keyword is up to 54 characters[34)] and "=" or space cannot be used.

[34)] To store a maximum or minimum 64-bit integer value and a comment string whose length is 1, length of keyword must be 54.

In ESO's convention, the header record of the long keyword is labeled `HIERARCH`, however, this way is not sophisticated.

11.4 Number of columns of the ASCII table or the binary table that exceed 999

Header keywords of 8 characters in FITS standard limit the number of column definitions up to 999 in ASCII table or binary table.

By the extension of long keyword described in the previous section, number of columns of ASCII table or binary table is unlimited in SFITSIO.

11.5 Applying CONTINUE keyword for long comment string

`COMMENT` records are useful to store a long description into FITS header. However, FITS libraries cannot know which header records are explained by such `COMMENT` strings. This will prevent automatic processing about FITS header, therefore, comments and descriptions should be written after "`/`" character in normal header records.

On the other hand, there is a problem that a comment string after "`/`" is not saved completely with simple format algorithm, if the record has long string value.

To solve this problem, SFITSIO allocates enough comment area using new record with `CONTINUE` keyword at the end of a long string value like this:

```
TELEM34 = 'BAD_FRAME,UNDEF_ANOM_FRAME,BLANK,IN_SAA,NEAR_MOON,UNTRUSTED_FRAME&'
CONTINUE  '' / element names
```

If the comment area is not still enough to save a comment string completely, SFITSIO saves a long comment string using `CONTINUE` keyword. Here is an example:

```
MESSAGE = 'FITS (Flexible Image Transport System) format is defined in &'
CONTINUE  '"Astronomy and Astrophysics", volume 376, page 359; bibcode: &'
CONTINUE  '2001A&A...376..359H' / In SFITSIO, this message is not written
CONTINUE  / automatically. Therefore, SFITSIO is not CFITSIO :-)
```

This extension was devised by L1TSD project of ISAS/JAXA.

11.6 Definition of new line characters in a string value in FITS header

Column descriptions of binary table or ASCII table is often converted into HTML, VOTable, LaTeX format, etc. To convert such descriptions into other formats automatically, column descriptions, annotations, etc. should be stored in string values with unique keywords rather than `COMMENT` records. In this case, we have to define new line characters to keep uniformity in various formats.

In SFITSIO, "`\n`"(`"\\n"` in C) in a header record is defined as new line characters, and built-in formatter splits a long string value into some header records using new line characters in it. Here is an example:

```
TDESC3  = 'Trigger type flag,\n&'
CONTINUE  '  b1000000:SUD(trigd by SuperUpper Discriminator),\n&'
CONTINUE  '  b0100000:ANODE,\n&'
CONTINUE  '  b0010000:PIN0 b0001000:PIN1 b0000100:PIN2,\n&'
CONTINUE  '  b0000010:PIN3 b0000001:PSEUDO' / description of column
```

This extension was devised by L1TSD project of ISAS/JAXA.

11.7 Declaration of new column keywords of the ASCII or binary table

New header keywords that express properties of table column (field) are often defined in various institutes. For example, `TLMIN`n and `TLMAX`n are representative keywords.

However, FITS libraries cannot know whether they are column keywords or not. Therefore, when FITS libraries erase or move some columns in ASCII or binary tables, newly defined column keywords are not erased or not copied properly. Additional problem is that information of FITS tables are not automatically converted into other formats such as HTML, VOTable, LaTeX, etc.

To solve this problem, we defined "`TXFLDKWD`" keyword to indicate new column keywords for FITS libraries. We show an example to append `TLMIN`n, `TLMAX`n, `TALAS`n, `TELEM`n and `TDESC`n column keywords:

```
TXFLDKWD= 'TLMIN,TLMAX,TALAS,TELEM,TDESC' / extended field keywords
```

`TXFLDKWD` should be written before standard column keywords such as `TTYPE`n.

This extension was devised by L1TSD project of ISAS/JAXA.

11.8 Definition of an alias of the ASCII table or the binary table

An alias of the column can be defined by using keyword `TALAS`n as follows.

```
TTYPE4  = 'QUATERNION'          / Quaternion at boresight
TALAS4  = 'AOCU_ADS_Q'          / aliases of column name
```

In case that multiple aliases need to be defined, define them in csv format. Of course, the Alias defined here is also available in the APIs of SFITSIO.

This extension was devised by 'AKARI' project of ISAS/JAXA.

11.9 Definition of an element name in the column of the binary table

In SFITSIO, each element in the column having array of the binary table can be named as follows.

```
TTYPE34 = 'FLAG    '            / Flag for detector condition
TELEM34 = 'BAD_FRAME,UNDEF_ANOM_FRAME,BLANK,IN_SAA,NEAR_MOON,UNTRUSTED_FRAME&'
CONTINUE  '' / element names
TFORM34 = '8X      '            / data format : BIT
```

Each name of the element can be defined in the record of `TELEM`n keyword in the csv format. It is not necessarily that all elements should be named. If omitting some name definitions, some elements of right side are not named. Although this example shows the case for bit type, `TELEM`n expression can also be used for other types (e.g., integer, real, etc.).

In the SFITSIO APIs, the data value can be read or written by giving these element names to the argument of the member function such as `dvalue()` or `assign()`.

This extension was devised by 'AKARI' project of ISAS/JAXA.

11.10 Definition of the number of bits in the column of the binary table

In case that the column has array of bit type (i.e., `TFORM`n is 'm`X`'), a bit width of an element can be defined by giving bit-field description like struct of C-language (or giving multiple same names of the element) to the value of `TELEM`n.

An example is shown below:

```
TTYPE36 = 'QUALITY '           / Quality for each pixel condition
TFORM36 = '4000X   '           / data format : BIT
TDIM36  = '(40,100)'
TELEM36 = 'QUAL_CV_PARAM:2,QUAL_RC_PARAM:2,QUAL_RC_CF:2,QUAL_DF_EQ:2,&'
CONTINUE  'QUAL_RP_DATA:2,QUAL_RP_PARAM:2,QUAL_RP_TABLE:2,QUAL_FF_PARAM:2,&'
CONTINUE  'QUAL_FF_CF:2,QUAL_GPGL_CORR:2,QUAL_MTGL_CORR:2,QUAL_TR_HIST:2,&'
CONTINUE  'QUAL_TR_PARAM:2,QUAL_DK_DATA:2,QUAL_DK_PARAM:2,QUAL_DK_TABLE:2,&'
CONTINUE  'QUAL_FX_CORR:2,QUAL_FX_PARAM:2,,,,' / element names
```

In this case, the definition about `TTYPE36`, `TFORM36` and `TDIM36` meets FITS standard, and a cell in the field has bits of 40×100. `TELEM36` gives the names of elements and bit width (2-bit) of each element. It is not necessarily that all elements should be named. If omitting some name definitions, some elements of right side are not named. The element names can have ':' character, however, it should not be used in terms of future compatibility problems.

In the SFITSIO APIs, integer values (up to 32-bit) having a bit width defined here can be read or written. SFITSIO copies the column data in a FITS file into memory buffer without modifying original byte data. Therefore, using bit-field extension can reduce file size and memory consumption.

This extension was devised by 'AKARI' project of ISAS/JAXA.

12 Unsupported FITS Standard and Limitation

Random groups structure of FITS standard is unsupported by the SFITSIO.

Member functions for file I/O and template input support complex numbers and variable length array in the binary table. However, current version of SFITSIO does not provide high-level APIs to read and write cells of complex numbers or variable length array. To handle it, programmers have to write codes using low-level APIs.

13 Reference

13.1 Constants

In `namespace sli`, `namespace FITS`, and constants required to treat FITS files, were defined. In the user's code, the values themselves of the constants should not be written.

To indicate the kind of HDU, use following constants.

Type	Constants in the SFITSIO	Value
`const int`	`FITS::IMAGE_HDU`	`0`
`const int`	`FITS::ASCII_TABLE_HDU`	`1`
`const int`	`FITS::BINARY_TABLE_HDU`	`2`

To indicate the state of user header record, use following constants.

Type	Constants in the SFITSIO	Value
`const int`	`FITS::NULL_RECORD`	`0`
`const int`	`FITS::NORMAL_RECORD`	`1`
`const int`	`FITS::DESCRIPTION_RECORD`	`2`

To indicate the kind of data, use following constants.

Type	Constants in the SFITSIO	Value	Image	Binary Table
`const int`	`FITS::BIT_T`	`88`		◯
`const int`	`FITS::BYTE_T`	`66`	◯	◯
`const int`	`FITS::LOGICAL_T`	`76`		◯
`const int`	`FITS::BOOL_T`	`76`		◯
`const int`	`FITS::ASCII_T`	`65`		◯
`const int`	`FITS::STRING_T`	`65`		◯
`const int`	`FITS::SHORT_T`	`73`	◯	◯
`const int`	`FITS::LONG_T`	`74`	◯	◯
`const int`	`FITS::LONGLONG_T`	`75`	◯	◯
`const int`	`FITS::FLOAT_T`	`69`	◯	◯
`const int`	`FITS::DOUBLE_T`	`68`	◯	◯
`const int`	`FITS::COMPLEX_T`	`67`		◯
`const int`	`FITS::DOUBLECOMPLEX_T`	`77`		◯
`const int`	`FITS::LONGARRDESC_T`	`80`		◯
`const int`	`FITS::LLONGARRDESC_T`	`81`		◯

13.2 Types

The `namespace fits` and data types required to treat FITS files are defined in the `namespace sli`.

To access raw data of the FITS file, use following types.

Types	Types used actually	Image	Binary Table
`fits::bit_t`	`struct _bit_t`		○
`fits::byte_t`	`uint8_t`	○	○
`fits::logical_t`	`uint8_t`		○
`fits::short_t`	`int16_t`	○	○
`fits::long_t`	`int32_t`	○	○
`fits::longlong_t`	`int64_t`	○	○
`fits::float_t`	`float`	○	○
`fits::double_t`	`double`	○	○
`fits::complex_t`	`float _Complex`		○
`fits::doublecomplex_t`	`double _Complex`		○
`fits::ascii_t`	`char`		○
`fits::longarrdesc_t`	`struct _longarrdesc_t`		○
`fits::llongarrdesc_t`	`struct _llongarrdesc_t`		○

To define the FITS header, use following structures.

Types	Definition of structure
`fits::header_def`	`struct { const char *keyword; const char *value; const char *comment; }`

To define the column of ASCII table and Binary table, use following structures.

Types	Definition of structure
`fits::table_def`	`struct { const char *ttype; const char *ttype_comment; const char *talas; const char *telem; const char *tunit; const char *tunit_comment; const char *tdisp; const char *tform; const char *tdim; const char *tnull; const char *tzero; const char *tscal; }`

13.3 Operation of whole FITS

In this section, we describe APIs to perform a stream input/output and to operate the configuration of the HDU.

13.3.1 read_stream()

NAME

read_stream() — Read from stream

SYNOPSIS

```
ssize_t read_stream( const char *path );
ssize_t readf_stream( const char *path_fmt, ... );
ssize_t vreadf_stream( const char *path_fmt, va_list ap );
```

DESCRIPTION

This member function reads the FITS file specified by `path` or URL (supporting for `file://`, `http://`, `ftp://`) and imports its whole content to the object. Judging from the file name of `path` or MIME header obtained from http server, in case of necessity, zlib or bzlib are used for the reading[35)]. In case that the files are retrieved from a ftp server, user name and password can be set to `path` in the form of `ftp://username:password@hostname/`... If neither username nor password is set, the API accesses the server anonymously.

If `hdus_to_read().assign()` or `cols_to_read().assign()`(§13.3.2) are called before this member function is used, specific HDU, or a specific column of ASCII table or of Binary table can be read.

In case of readf_stream() member function, arguments after `path_fmt` should be the set in the same manner as that of `printf()` in libc. Refer §13.4.9 about the format of the `printf()`.

PARAMETER

[I]	`path`	File name (URL name)
[I]	`path_fmt`	Format specification of the file name (URL name)
[I]	`...`	Each element data of the file name (URL name)
[I]	`ap`	All element data of the file name (URL name)

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Byte size of the read stream (In case of compressed file, size after extracted)
Negative value	:	Error (Failed to read the stream, e.g., because the file was not found)

EXCEPTION

In case that a fatal error occurs (for example, memory capacity is not enough for a specified FITS file and buffer cannot be allocated), the API throws an exception(`sli::err_rec` exception) from classes composing sfitsio and classes provided by SLLIB.

EXAMPLES

See the EXAMPLES in §13.3.2 or §5.2, §5.3 in Tutorial.

[35)] This function is carried out in digeststreamio class of SLLIB. Detail in APPENDIX4 (§17)

13.3.2 hdus_to_read().assign(), cols_to_read().assign()

NAME

hdus_to_read().assign(), cols_to_read().assign() — Specification of HDU or column to be read from stream

SYNOPSIS

```
tarray_tstring &hdus_to_read().assign( const char *hdu0, const char *hdu1, ... );
tarray_tstring &hdus_to_read().assign( const char *hdus[] );
tarray_tstring &cols_to_read().assign( const char *col0, const char *col1, ... );
tarray_tstring &cols_to_read().assign( const char *cols[] );
```

DESCRIPTION

By using these member functions before read_stream() member function (§13.3.1), only a specific HDU, or a specific column of an ASCII table or a Binary table can be read. (Only the reading of a Primary HDU, however, cannot be skipped.) For the argument, a HDU name, a column name of an ASCII table or a Binary table should be listed and terminated by NULL.

If all HDUs or all columns are necessary, use `hdus_to_read().init()` or `cols_to_read().init()`, respectively. A value set by these member functions can be erased by init() member function (§13.3.15).

PARAMETER

[I]	`hdu0,hdu1...`	HDU name (specify NULL for the terminal)
[I]	`hdus[]`	Array of HDU name (specify NULL for the terminal)
[I]	`col0,col1...`	Column name of Binary or ASCII table (specify NULL for the terminal)
[I]	`cols[]`	Array of column name of Binary or ASCII table (specify NULL for the terminal)

([I] : input, [O] : output)

RETURN VALUE

This API returns a reference to `tarray_tstring` object including set HDU name or column name.

EXCEPTION

In case that this API fails to reserve inner buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code reads the Primary HDU and the FIS_OBS HDU, and only columns named as "AFTIME", "DET", "RA" and "DEC" of the Binary table.

```
fitscc fits;
/* Required HDUs */
fits.hdus_to_read().assign("Primary","FIS_OBS",NULL);
/* Required columns in binary tables */
fits.cols_to_read().assign("AFTIME","DET","RA","DEC",NULL);
/* Reading a file */
r_size = fits.readf_stream("my_file_no.%d.fits.gz",i);
```

13.3.3 write_stream()

NAME

write_stream() — Write to a stream

SYNOPSIS

```
ssize_t write_stream( const char *path );
ssize_t writef_stream( const char *path_fmt, ... );
ssize_t vwritef_stream( const char *path_fmt, va_list ap );
```

DESCRIPTION

write_stream() writes all content of an object to a file specified by `path`. Judging from the file name of `path`, in case of necessity, zlib or bzlib are used for the writing[36]. In case that the files are put to a ftp server, user name and password can be set to `path` in the form of `ftp://username:password@hostname/`. If neither username nor password is set, the API accesses the server anonymously.

In case of writef_stream() member function, arguments after `path_fmt` should be set same with that of `printf()` in libc.

PARAMETER

[I]	`path`	File name (URL name)
[I]	`path_fmt`	Format specification of the file name (URL name)
[I]	`...`	Each element data of the file name (URL name)
[I]	`ap`	All element data of the file name (URL name)

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Byte size of the written stream (In case of compressed file, size after extracted)
Negative value	:	Error (Failed to write the stream, e.g., because of invalid permission)

EXCEPTION

If the API fails to operate memory buffer or to output into a file (for example, unexpected error at the writing to output file), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code writes whole content of the fits object to the file `my_file_no.1.fits.bz2`, for example in case that `i=1`. In this case, because the suffix is ".bz2", the file is compressed via bzip2.

```
fitscc fits;
w_size = fits.writef_stream("my_file_no.%d.fits.bz2",i);
```

Other examples are shown in §5.2 and §5.3 in Tutorial.

---

[36] This function is carried out in digeststreamio class of SLLIB. Detail in APPENDIX4 (§17)

13.3.4 access_stream()

NAME

access_stream() — Access to a stream

SYNOPSIS

```
ssize_t access_stream( const char *path );
ssize_t accessf_stream( const char *path_fmt, ... );
ssize_t vaccessf_stream( const char *path_fmt, va_list ap );
```

DESCRIPTION

An argument of access_stream() member function should be set with the style of open() of Perl. If the argument `path` indicates a file or a URL, access_stream() reads FITS content from it or writes FITS content to it. If the argument `path` indicates a command-line, access_stream() executes the commands, creates a pipe-connection to commands, and reads FITS content from the pipe or writes FITS to the pipe[37)].

When the argument `path` does not indicate commands, "`<`" or "`>`" should be set at the first character of the argument string. For example, `"< infile.fits"` shows reading the file "infile.fits" (see EXAMPLE-1) and `"> outfile.fits"` shows writing the file "outfile.fits". Neither "`<`" nor "`>`" is found, the file `path` will be read. Compressed files (gzip or bzip2) will be decompressed automatically.

When the argument `path` indicates commands, the commands should include "`|`" or "`|`" at the first or the last character of the argument string. If "`|`" is placed at the last character of the argument, access_stream() executes the commands in the argument, creates a pipe-connection to commands, and reads FITS content from the pipe. If "`|`" is placed at the first character of the argument, access_stream() executes the commands in the argument, creates a pipe-connection to commands, and write FITS content to the pipe (see EXAMPLE-2). The member function access_stream() executes the commands as "`/bin/sh -c command`". Therefore, "`|`", "`<`" and "`>`" can be included in the `path` (see EXAMPLE-2).

In case of accessf_stream() member function, arguments after `path_fmt` should be set same with that of `printf()` in libc.

PARAMETER

[I]	`path`	File name (URL name) or command line
[I]	`path_fmt`	Format specification of the file name (URL name) or command line
[I]	`...`	Each element data of the file name (URL name) or command line
[I]	`ap`	All element data of the file name (URL name) or command line

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Byte size of the written or read stream (In case of compressed file, size after extracted)
Negative value	:	Error (Failed to read the stream, e.g., because the file was not found) Error (Failed to write the stream, e.g., because of invalid permission)

EXCEPTION

If the API fails to operate memory buffer or to output into a file (for example, unexpected error at the writing to output file), it throws an exception derived from SLLIB (`sli::err_rec` exception).

[37)] This function is carried out in digeststreamio class of SLLIB. Detail in APPENDIX4 (§17)

EXAMPLE-1

Following code reads whole content of the FITS file `my_file.fits.bz2`. In this case, because the suffix is ".bz2", the file is compressed via bzip2.

```
fitscc fits;
r_size = fits.access_stream("< my_file.fits.bz2");
```

EXAMPLE-2

Following code read a FITS file using the connection of the HTTP over SSL.

```
fitscc fits;
r_size = fits.accessf_stream("wget -O - %s | gzip -dc |",
                             "https://foo/secret.fits.gz");
```

Other examples are shown in §5.4 in Tutorial. The tutorial shows some examples for the use of a compression tool with multi-threading support.

13.3.5 read_template()

NAME

read_template() — Read a FITS template

SYNOPSIS

```
ssize_t read_template( int flags, const char *path );
ssize_t vreadf_template( int flags, const char *path_fmt, va_list ap );
ssize_t readf_template( int flags, const char *path_fmt, ... );
```

DESCRIPTION

This member function reads the FITS template file (See §9) specified by `path` or URL (supporting for `file://`, `http://`, `ftp://`) and creates FITS content to the object that has no data contents in pixels or cells. Judging from the file name of `path` or MIME header obtained from http server, in case of necessity, zlib or bzlib are used for the reading. In case that the files are retrieved from a ftp server, user name and password can be set to `path` in the form of `ftp://username:password@hostname/...` If neither username nor password is set, the API accesses the server anonymously.

In case of readf_template() member function, arguments after `path_fmt` should be the set in the same manner as that of `printf()` in libc. Refer §13.4.9 about the format of the `printf()`.

PARAMETER

[I]	`flags`	Flags to switch the behavior (currently unused; always 0 should be set)
[I]	`path`	File name (URL name)
[I]	`path_fmt`	Format specification of the file name (URL name)
[I]	`...`	Each element data of the file name (URL name)
[I]	`ap`	All element data of the file name (URL name)

([I] : input, [O] : output)

RETURN VALUE

0	:	Successfully finished.
Negative value	:	Error (Failed to read the stream, e.g., because the file was not found)

EXCEPTION

In case that a fatal error occurs (for example, memory capacity is not enough for a specified FITS file and buffer cannot be allocated), the API throws an exception(`sli::err_rec` exception) from classes composing sfitsio and classes provided by SLLIB.

EXAMPLES

```
fitscc fits;
status = fits.read_template(0, "template/image1.tpl");
```

See §9 for details of SFITSIO template files.

13.3.6 stream_length()

NAME

stream_length() — Return a file size to be written to a stream

SYNOPSIS

```
ssize_t stream_length();
```

DESCRIPTION

stream_length() realign the system header and return the size of a file written by write_stream().

RETURN VALUE

Non-negative value	:	Size to be written to stream.
Negative value	:	Error. (only in debug mode)

EXCEPTION

If the API fails to handle string because of lack of memory, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

```
fitscc fits;
w_size = fits.stream_length();
```

13.3.7 length()

NAME

length() — Number of HDU

SYNOPSIS

```
long length() const;
```

RETURN VALUE

length() returns number of HDU.

EXAMPLES

```
long hdu_count = fits.length();
```

See also the example in §5.2 in Tutorial.

13.3.8 fmttype()

NAME

fmttype() — Format type name

SYNOPSIS

```
const char *fmttype() const;
```

DESCRIPTION

fmttype() returns a format type name. If the format type name is not set, it returns `NULL`.

Since return value is an address of an object's internal buffer, it is invalid in case that object is deleted or its name is changed.

For more information about format type name, see §11.1.

RETURN VALUE

fmttype() returns an address of a format type name.

EXAMPLES

```
printf("Format Type = %s\n", fits.fmttype());
```

13.3.9 ftypever()

NAME

ftypever() — Version number of format type

SYNOPSIS

```
long long ftypever() const;
```

RETURN VALUE

ftypever() returns version number of format type.

EXAMPLES

```
printf("Verson of Format Type = %lld\n", fits.ftypever());
```

13.3.10 hduname(), extname()

NAME

hduname(), extname() — HDU name

SYNOPSIS

```
const char *hduname( long index ) const;
const char *extname( long index ) const;
```

DESCRIPTION

hduname() and extname() return HDU name specified by `index`. If HDU name is not set, they return `NULL`.

Since return value is an address of an object's internal buffer, it is invalid in case that object is deleted or its name is changed.

PARAMETER

[I] `index` index of HDU

([I] : input, [O] : output)

RETURN VALUE

hduname(), extname() return an address of HDU name.

EXAMPLES

Following code lists all HDU names to standard output.

```
long i;
for ( i=0 ; i < fits.length() ; i++ ) {
    printf("HDU[%ld] Name is %s\n", i, fits.hduname(i));
}
```

See also the example in §5.2 in Tutorial.

13.3.11 hduver(), extver()

NAME

hduver(), extver() — HDU version

SYNOPSIS

```
long long hduver( long index ) const;
long long hduver( const char *name ) const;
long long extver( long index ) const;
long long extver( const char *name ) const;
```

DESCRIPTION

hduver(), extver() return HDU version specified by `index` or `name`.

PARAMETER

[I] `index` HDU index
[I] `name` HDU name
([I] : input, [O] : output)

RETURN VALUE

hduver(), extver() return HDU version.

EXAMPLES

Following code lists all HDU versions to standard output.

```
long i;
for ( i=0 ; i < fits.length() ; i++ ) {
    printf("HDU[%ld] Version is %lld\n", i, fits.hduver(i));
}
```

13.3.12 hdulevel(), extlevel()

NAME

hdulevel(), extlevel() — HDU level

SYNOPSIS

```
long long hdulevel( long index ) const;
long long hdulevel( const char *name ) const;
long long extlevel( long index ) const;
long long extlevel( const char *name ) const;
```

DESCRIPTION

hdulevel(), extlevel() return HDU level (value of `EXTLEVEL` in FITS header) specified by `index` or `name`.

PARAMETER

[I] `index` HDU index
[I] `name` HDU name
([I] : input, [O] : output)

13.3.13 hdutype(), exttype()

NAME

hdutype(), exttype() — HDU type

SYNOPSIS

```
int hdutype( long index ) const;
int hdutype( const char *name ) const;
int exttype( long index ) const;
int exttype( const char *name ) const;
```

DESCRIPTION

hdutype(), exttype() return HDU type specified by `index` or `name`.
Return value is `FITS::IMAGE_HDU`, `FITS::ASCII_TABLE_HDU`, or `FITS::BINARY_TABLE_HDU`.

PARAMETER

[I] `index` HDU index
[I] `name` HDU name
([I] : input, [O] : output)

RETURN VALUE

hdutype(), exttype() return HDU type.

EXAMPLES

```
switch ( fits.hdutype(index) ) {
  case FITS::IMAGE_HDU:
      printf("This is an image HDU!\n");
      break;
  case FITS::ASCII_TABLE_HDU:
      printf("This is an ASCII table HDU!\n");
      break;
  case FITS::BINARY_TABLE_HDU:
      printf("This is a Binary table HDU!\n");
      break;
  default:
      printf("This is an unknown type HDU!\n");
      break;
}
```

13.3.14 index()

NAME

index() — HDU index

SYNOPSIS

```
long index( const char *name ) const;
long indexf( const char *name_fmt, ... ) const;
long vindexf( const char *name_fmt, va_list ap ) const;
```

DESCRIPTION

index() returns HDU index specified by `name`.

Only when `"Primary"` is specified for HDU name, they always return 0 if the Primary HDU exists even if header keyword `EXTNAME` is not set.

They return negative value if specified name is not found.

Arguments after `name_fmt` can be set same with that of `printf()`.

PARAMETER

[I]	`name`	HDU name
[I]	`name_fmt`	Format string of HDU name
[I]	`...`	All element data of the HDU name
[I]	`ap`	All element data of the HDU name

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	HDU index specified by `name`.
Negative value	:	Error (If specified name is not found.)

EXCEPTION

If the API fails to operate memory buffer at conversion of a format string, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code acquire the HDU index specified by HDU name.

```
long index;
index = fits.index("FIS_OBS");
```

13.3.15 init()

NAME

init() — Initialization of object

SYNOPSIS

```
fitscc &init();
fitscc &init( const fitscc &obj );
```

DESCRIPTION

init() deletes all content of an object and initialize it. In case that the argument `obj` is given, it copies all contents of the object obj to the own object that calls init().

PARAMETER

[I]	`obj`	fitscc object copied at initialization.

([I] : input, [O] : output)

RETURN VALUE

init() returns a reference to its own object.

EXCEPTION

If the API fails to operate internal memory buffer (for example, `obj`'s data is too big for free memory area), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
/* Initialize Fits Object!! */
fits.init();
```

13.3.16 append_image()

NAME

append_image() — Append an Image HDU

SYNOPSIS

```
fitscc &append_image( const char *hduname, long long hduver,
                      int type, long naxis0, long naxis1 = 0, long naxis2 = 0 );
fitscc &append_image( const char *hduname, long long hduver,
                      int type, long naxisx[], long ndim );
fitscc &append_image( const char *hduname, long long hduver,
                      const fits_image &src );
fitscc &append_image( const fits_image &src );
```

DESCRIPTION

append_image() append an Image HDU. Argument `hduname` specify the name of HDU and `hduver` specifies its version. These values are reflected to `EXTNAME` and `EXTVER` in a FITS header. If `NULL` is given to `hduname`, this function can be invalid; giving `NULL`, however, is not recommended for use of SFITSIO.

As `type`, any one of `FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T` and `FITS::BYTE_T` should be given.

To `naxis0`, `naxis1` and `naxis2`, specify the number of pixels of x axis, y axis and z axis. `naxis1` and `naxis2` can be omitted. However,if only `naxis0` is given, the image is regarded as one dimension, and if `naxis0` and `naxis1` is given, the image is regarded as two dimensions. If the image exceeds three dimensions, `naxisx` and `ndim` are specified.

PARAMETER

[I]	`hduname`	HDU name
[I]	`hduver`	HDU version
[I]	`type`	HDU type
[I]	`naxis0`	Number of pixels of X axis
[I]	`naxis1`	Number of pixels of Y axis
[I]	`naxis2`	Number of pixels of Z axis
[I]	`naxisx`	List of number of pixels of each axis
[I]	`ndim`	Number of elements of `naxisx`
[I]	`src`	Image object to be appended

([I] : input, [O] : output)

RETURN VALUE

append_image() returns a reference to itself.

EXCEPTION

If the API fails to operate internal memory buffer (for example, image data is too big for

free memory area), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

Following code appends double-typed 1024 × 1024 Image HDU of which HDU name is "X-BAND" and HDU version is 100 to the object fits.

```
fits.append_image("X-BAND", 100,  FITS::DOUBLE_T, 1024, 1024);
```

See also the example in §5.9 in Tutorial.

13.3.17 append_table()

NAME

append_table() — Append the Ascii Table HDU or the Binary Table HDU

SYNOPSIS

```
fitscc &append_table( const char *hduname, long long hduver,
                      const fits::table_def defs[], bool ascii = false );
fitscc &append_table( const char *hduname, long long hduver,
                      const fits_table &src );
fitscc &append_table( const fits_table &src );
```

DESCRIPTION

append_table() appends the Ascii Table HDU or the Binary Table HDU. If the argument `ascii` is set to `true`, Ascii Table HDU is appended. HDU name and its version is specified to the arguments `hduname` and `hduver`, respectively. These values are reflected to `EXTNAME` and `EXTVER` in the FITS header. If NULL is specified to the `hduname`, this function can be invalid, however, giving `NULL` is not recommended for use of SFITSIO.

By using `defs`, Definition of Ascii Table or Binary Table is specified. Members of `fits::table_def` structure are following:

```
typedef struct {
    const char *ttype;          /* column name */
    const char *ttype_comment;
    const char *const *talas;   /* column alias */
    const char *const *telem;   /* element name */
    const char *tunit;          /* physical unit */
    const char *tunit_comment;
    const char *tdisp;          /* display format */
    const char *tform;          /* column type */
    const char *tdim;           /* specification of array */
    const char *tnull;          /* value of blank */
    const char *tzero;          /* zero point */
    const char *tscal;          /* scaling factor */
    const char *theap;          /* (unsupported) */
} fits::table_def;
```

Since the keywords in the Binary Table header is used for this structure's definition as it is, in case of the Binary Table, values of `TTYPE`*n*, `TFORM`*n*, and so on, can be replaced to corresponding members directly.

On the other hand, in case of the Ascii Table, note that member names of this structure do not correspond one-to-one with keyword names in the header. At first, to `tform`, string length

of column must be specified in the form of "number + `A`", for example, `"16A"`. (A format such as `"120A10"`, however, cannot be used.) Giving `telem` and `tdim` does not make sense. `TFORM`*n* in the Ascii Table is given to `tdisp` and this is utilized as a format of conversion from a value of an argument to string when the value of the argument (number or string) of the SFITSIO's member function is written to Ascii Table. Assignable format is

`A`*w*, `I`*w*, `F`*w*`.`*d*, `E`*w*`.`*d* or `D`*w*`.`*d*.

`NULL` or `""` should be given to a term which does not need to be assigned. `THEAP` is not supported in the SFITSIO. At the last of array `defs`, all members need to be NULL.

PARAMETER

[I]	`hduname`	HDU name
[I]	`hduver`	HDU version
[I]	`defs`	`fits::table_def` structure
[I]	`ascii`	Type of Table HDU to be appended(false:Binary true:Ascii)
[I]	`src`	Table object to be appended

([I] : input, [O] : output)

RETURN VALUE

append_table() returns a reference to itself.

EXCEPTION

If the API fails to operate internal memory buffer (for example, table data to be appended is too big for free memory area), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

Following code appends the Binary Table HDU of which HDU name is "EVENT" and HDU version is 100, defined by structure array def, to the object fits.

```
const fits::table_def def[] = {
    // TTYPE,comment, talas, telem, TUNIT,comment, TDISP, TFORM, TDIM
    { "TIME","satellite time", NULL, NULL, "s","", "D16.3", "1D", "" },
    { "NAME","", NULL, NULL, "","", "", "128A16", "(4,2)" },
    { NULL }
};
fits.append_table("EVENT", 100, defs);
```

See also the examples in §5.16 and §5.17 in Tutorial.

13.3.18 insert_image()

NAME

insert_image() — Insert an Image HDU

SYNOPSIS

```
fitscc &insert_image( long index0,
                      const char *hduname, long long hduver,
                      int type, long naxis0, long naxis1 = 0, long naxis2 = 0 );
fitscc &insert_image( long index0,
                      const char *hduname, long long hduver,
                      int type, long naxisx[], long ndim );
```

```
fitscc &insert_image( const char *hduname0,
                      const char *hduname, long long hduver,
                      int type, long naxis0, long naxis1 = 0, long naxis2 = 0 );
fitscc &insert_image( const char *hduname0,
                      const char *hduname, long long hduver,
                      int type, long naxisx[], long ndim );
fitscc &insert_image( long index0,
                      const char *hduname, long long hduver,
                      const fits_image &src );
fitscc &insert_image( const char *hduname0,
                      const char *hduname, long long hduver,
                      const fits_image &src );
fitscc &insert_image( long index0, const fits_image &src );
fitscc &insert_image( const char *hduname0, const fits_image &src );
```

DESCRIPTION

insert_image() inserts Image HDU to a HDU specified by `index0` or `hduname0`.

Specifications of arguments after `hduname` are same as the case of `append_image()`(§13.3.16).

PARAMETER

[I]	`index0`	HDU index which designate insert position
[I]	`hduname0`	HDU name which designate insert position
[I]	`hduname`	HDU name
[I]	`hduver`	HDU version
[I]	`type`	HDU type
[I]	`naxis0`	Number of pixels of X axis
[I]	`naxis1`	Number of pixels of Y axis
[I]	`naxis2`	Number of pixels of Z axis
[I]	`naxisx`	List of Number of pixels of each axis
[I]	`ndim`	Number of elements of `naxisx`
[I]	`src`	Image object to be inserted

([I] : input, [O] : output)

RETURN VALUE

insert_image() returns a reference to itself.

EXCEPTION

If the API fails to operate internal memory buffer (for example, image data to be inserted is too big for free memory area), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

Following code inserts double-typed 1024×1024 Image HDU of which name is "X-TABLE" and version is 100 in front of HDU "X-BAND".

```
fits.insert_image("X-BAND",  "XO-BAND", 100, FITS::DOUBLE_T, 1024, 1024);
```

13.3.19 insert_table()

NAME

insert_table() — Insert an Ascii Table or a Binary Table

SYNOPSIS

```
fitscc &insert_table( long index0,
                      const char *hduname, long long hduver,
                      const fits::table_def defs[], bool ascii = false );
fitscc &insert_table( const char *hduname0,
                      const char *hduname, long long hduver,
                      const fits::table_def defs[], bool ascii = false );
fitscc &insert_table( long index0,
                      const char *hduname, long long hduver,
                      const fits_table &src );
fitscc &insert_table( const char *hduname0,
                      const char *hduname, long long hduver,
                      const fits_table &src );
fitscc &insert_table( long index0, const fits_table &src );
fitscc &insert_table( const char *hduname0, const fits_table &src );
```

DESCRIPTION

insert_table() inserts an Ascii Table HDU or a Binary Table HDU into an HDU specified by `index0` or `hduname0`.

However, it cannot insert into a Primary HDU.

Specifications of arguments after `hduname` are the same as in case of of `append_table()`(§13.3.17).

PARAMETER

[I]	`index0`	HDU index which designate insert position
[I]	`hduname0`	HDU name which designate insert position
[I]	`hduname`	HDU name
[I]	`hduver`	HDU version
[I]	`defs`	`fits::table_def` structure
[I]	`ascii`	type of Table HDU to be inserted (false: Binary, true: Ascii)
[I]	`src`	table object to be inserted

([I] : input, [O] : output)

RETURN VALUE

insert_table() returns a reference to itself.

EXCEPTION

If the API fails to operate internal memory buffer (for example, the table data to be inserted is too big for free memory area), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

Following code inserts table HDU of which name is "X-TABLE" and version is 100, defined by structure array defs, in front of HDU "X-BAND". (Refer EXAMPLES in §13.3.17 for structure array defs.)

```
fits.insert_table("X-BAND",  "X-TABLE", 100, defs);
```

13.3.20 erase()

NAME

erase() — Erase a HDU

SYNOPSIS

```
fitscc &erase( long index );
fitscc &erase( const char *hduname );
```

DESCRIPTION

erase() erases a HDU specified by `index` or `hduname`.

However, it cannot erase a Primary HDU if the next HDU of the Primary HDU is not an Image HDU.

PARAMETER

[I]	`index`	HDU index to be erased
[I]	`hduname`	HDU name to be erased

([I] : input, [O] : output)

RETURN VALUE

erase() returns a reference to itself.

EXCEPTION

If the API fails to operate internal memory buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.erase("X-BAND");
```

13.3.21 assign_fmttype()

NAME

assign_fmttype() — Change a format type name

SYNOPSIS

```
fitscc &assign_fmttype( const char *fmttype, long long ftypever );
```

DESCRIPTION

assign_fmttype() changes a format type name. The argument `fmttype` is a string which defines a globally unique data format and `ftypever` gives its version. These values are reflected to `FMTTYPE` and `FTYPEVER` in a Primary HDU header.

`fmttype` and `ftypever` can be utilized for validity check of a FITS when the FITS is read from a file.

See §11.1 for more information about format type.

PARAMETER

[I]	`fmttype`	format type name to be set
[I]	`ftypever`	format type version to be set

([I] : input, [O] : output)

EXCEPTION

If the API fails to operate internal memory buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

RETURN VALUE

assign_fmttype() returns a reference to itself.

EXAMPLES

```
fits.assign_fmttype("ASTRO-X ALL-SKY SURVEY IMAGE", 101);
```

See also the examples in §5.9, §5.16 and §5.17 in Tutorial.

13.3.22 assign_ftypever()

NAME

assign_ftypever() — Change a format type version

SYNOPSIS

```
fitscc &assign_ftypever( long long ftypever );
```

DESCRIPTION

assign_ftypever() changes a format type version. The value is reflected to `FTYPEVER` in a primary HDU header.

`fmttype` and `ftypever` can be utilized for validity check of a FITS when the FITS is read from a file.

PARAMETER

[I] `ftypever` version of format type to be set

([I] : input, [O] : output)

RETURN VALUE

assign_ftypever () returns a reference to itself.

EXAMPLES

```
fits.assign_ftypever(102);
```

13.3.23 assign_hduname(), assign_extname()

NAME

assign_hduname(), assign_extname() — Change a HDU name

SYNOPSIS

```
fitscc &assign_hduname( long index, const char *name );
fitscc &assign_extname( long index, const char *name );
```

DESCRIPTION

assign_hduname() or assign_extname changes the HDU name specified by `index`. `name` is reflected to `EXTNAME` in a header. The name of Primary HDU can also be specified.

PARAMETER

[I] `index` HDU index which designates HDU whose name is to be changed

[I] `name` HDU name to be set

([I] : input, [O] : output)

EXCEPTION

If the API fails to operate internal memory buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

RETURN VALUE

assign_hduname() or assign_extname returns a reference to itself.

EXAMPLES

```
fits.assign_hduname("X-BAND");
```

13.3.24 assign_hduver(), assign_extver()

NAME

assign_hduver() — Change a HDU version number

SYNOPSIS

```
fitscc &assign_hduver( long index, long long ver );
fitscc &assign_extver( long index, long long ver );
```

DESCRIPTION

assign_hduver() or assign_extver() changes HDU version number specified by `index`. `ver` is reflected to `EXTVER` in a header.

The version number of Primary HDU can also be specified.

PARAMETER

[I]	`index`	HDU index which designates HDU whose version number is to be changed
[I]	`name`	version number to be set

([I] : input, [O] : output)

RETURN VALUE

assign_hduver () returns a reference to itself.

EXAMPLES

```
fits.assign_hduver(index, 101);
```

13.3.25 assign_hdulevel(), assign_extlevel()

NAME

assign_hdulevel() — Change a HDU level number

SYNOPSIS

```
fitscc &assign_hdulevel( long index, long long level );
fitscc &assign_extlevel( long index, long long level );
```

DESCRIPTION

assign_hdulevel() or assign_extlevel() changes HDU level number specified by `index`. `level` is reflected to `EXTLEVEL` in a header. The level number of Primary HDU can also be specified.

PARAMETER

[I]	`index`	HDU index which designates HDU whose level number is to be changed
[I]	`name`	level number to be set

([I] : input, [O] : output)

RETURN VALUE

assign_hdulevel() returns a reference to itself.

EXAMPLES

```
fits.assign_hdulevel(index, 101);
```

13.3.26 hduver_is_set(), extver_is_set()

NAME

hduver_is_set() — Check whether HDU version is set

SYNOPSIS

```
bool hduver_is_set( long index ) const;
bool hduver_is_set( const char *hduname ) const;
bool extver_is_set( long index ) const;
bool extver_is_set( const char *extname ) const;
```

DESCRIPTION

hduver_is_set() or extver_is_set() checks whether HDU version specified by `index` is set or not.

PARAMETER

[I] `index` HDU index
[I] `hduname` HDU name
[I] `extname` HDU name
([I] : input, [O] : output)

RETURN VALUE

hduver_is_set() or extver_is_set() returns true if a HDU version number is set, otherwise false.

EXAMPLES

Following code checks a setting of a version number for all HDUs. If the version number is not stated, version 100 will automatically be set.

```
long i;
for ( i=0 ; i < fits.length() ; i++ ) {
    if ( fits.hduver_is_set(i) == false ) {
        fits.assign_hduver(i, 100);
    }
}
```

13.3.27 hdulevel_is_set(), extlevel_is_set()

NAME

hdulevel_is_set() — Check whether HDU level is set

SYNOPSIS

```
bool hdulevel_is_set( long index ) const;
bool hdulevel_is_set( const char *hduname ) const;
bool extlevel_is_set( long index ) const;
bool extlevel_is_set( const char *extname ) const;
```

DESCRIPTION

hdulevel_is_set() or extlevel_is_set() checks whether HDU level specified by `index` is set or not.

PARAMETER

[I] `index` HDU index
[I] `hduname` HDU name
[I] `extname` HDU name
([I] : input, [O] : output)

RETURN VALUE

hdulevel_is_set() or extlevel_is_set() returns true if a HDU level number is set, otherwise false.

EXAMPLES

Following code checks a setting of a level number for all HDUs. If the level number is not stated, level 1234 will automatically be set.

```
long i;
for ( i=0 ; i < fits.length() ; i++ ) {
    if ( fits.hdulevel_is_set(i) == false ) {
        fits.assign_hdulevel(i, 1234);
    }
}
```

13.4 Operation of header

In this section, we describe how to handle APIs to operate a header. APIs are classified in two groups.[38] The first case is following format:

```
value = fits.hdu( ... ).function( ... );
```

And the second case is following format:

```
value = fits.hdu( ... ).header( ... ).function( ... );
value = fits.hdu( ... ).headerf( ... ).function( ... );
```

To the argument in the bracket of `hdu( ... )`, a HDU index (`long index`) or a HDU name (`const char *hduname`) should be specified. Also, in case of an Image HDU, "`hdu( ... )`" part can be used as "`image( ... )`". In addition, in case of an Ascii Table HDU or Binary Table HDU, "`table( ... )`" can be used.

To the argument in the bracket of "`header( ... )`", specify the header index (`long index`) or header keyword (`const char *keyword`). To the argument in the bracket of "`headerf( ... )`", specify the header keyword to the same way as `printf()` function in libc.

For the rest of this document, since the arguments in the brackets "`hdu( ... )`", "`header( ... )`" and "`headerf( ... )`" are all the same, descriptions about these arguments are omitted.

The number of header keyword is up to 54 characters and there is no limit to the string length in the SFITSIO. (Refer §10.1.) Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

13.4.1 hdu().header_length()

NAME

hdu().header_length() — Number of records in a header

SYNOPSIS

```
long hdu( ... ).header_length() const;
```

RETURN VALUE

`hdu().header_length()` returns the number of records in a header.

EXAMPLES

Following code displays the number of records in a header.

```
fits_image &primary = fits.image("Primary");
printf("Record Count = %ld\n", primary.header_length());
```

13.4.2 hdu().header_index()

NAME

hdu().header_index() — Header record index in a header

SYNOPSIS

```
long hdu( ... ).header_index( const char *keyword ) const;
long hdu( ... ).header_index( const char *keyword, bool is_description ) const;
```

[38] You can use APIs for handling headers like following combinations of member functions that express exact class structures: The first case is `value = fits.hdu(...).header().function(...);`, and the second case is `value = fits.hdu(...).header().at(...).function(...);`. See SFITSIO header files (fits_hdu.h, fits_header.h, and fits_header_record.h) for details.

DESCRIPTION

`hdu().header_index()` searches a keyword `keyword` from a record which format is not descriptive (unlike COMMENT or HISTORY) and returns its record index. It searches from a record which format is descriptive if `is_description` is `true`.

It returns negative value if the keyword is not found.

PARAMETER

[I]	`keyword`	keyword
[I]	`is_description`	Specification of record to be searched (true: descriptive format, false: other case)

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Record index
Negative value	:	Error (If specified keyword was not found.)

EXAMPLES

Following code displays index of the header which has keyword "TELESCOP".

```
fits_image &primary = fits.image("Primary");
printf("Record Index = %ld\n", primary.header_index("TELESCOP"));
```

13.4.3 hdu().header_regmatch()

NAME

hdu().header_regmatch() — Keyword search of a header

SYNOPSIS

```
long hdu( ... ).header_regmatch( long index, const char *keypat,
                                 ssize_t *rpos = NULL, size_t *rlen = NULL ) const;
long hdu( ... ).header_regmatch( const char *keypat,
                                 ssize_t *rpos = NULL, size_t *rlen = NULL ) const;
```

DESCRIPTION

`hdu().header_regmatch()` searches a keyword which matches to POSIX extended regular expression `keypat` from records starting with a record specified by `index` and returns hit record index. If no word matches to the expression, it returns negative value. If `index` is not given, Search begins from a first record.

Position of found character is returned to `*rpos` and length of matched string is returned to `*rlen`. These arguments do not need to be given.

This member function cannot search a header record of which format is descriptive (such as COMMENT and HISTORY).

PARAMETER

[I]	`index`	header record index to designate search start position.
[I]	`keypat`	keyword pattern string(regular expression)
[O]	`rpos`	position of found character
[O]	`rlen`	length of matched string

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Record index
Negative value	:	Error (If specified keyword was not found.)

EXAMPLES

Following code lists every records which have header keywords starting with `CRVAL1` or `CRVAL2` in the Primary HDU.

```
fits_image &primary = fits.image("Primary");
long i = 0;
while ( 0 <= (i=primary.header_regmatch(i,"^CRVAL[1-2]")) ) {
    printf("%s = %s\n", primary.header(i).keyword(),
                        primary.header(i).value());
    i++;
}
```

See also the example in §5.6 in Tutorial.

13.4.4 hdu().header().svalue()

NAME

hdu().header().svalue() — A string value of a header (high level)

SYNOPSIS

```
const char *hdu( ... ).header( ... ).svalue();
```

DESCRIPTION

`hdu().header().svalue()` removes single quotations (') and unnecessary blank characters from a string value of a specified header, and then returns it.

Since return value is an address of an object's internal buffer, it is invalid in case if object is deleted or this member function is called again.

RETURN VALUE

`hdu().header().svalue()` returns the address of a string value of a header.

EXAMPLES

Following code displays the value of the record whose name is `CTYPE1` in the primary HDU header.

```
fits_image &primary = fits.image("Primary");
printf("CTYPE1 = %s\n", primary.header("CTYPE1").svalue());
```

See also the example in §5.5 in Tutorial.

13.4.5 hdu().header().get_svalue()

NAME

hdu().header().get_svalue() — Get a string value of a header (high level)

SYNOPSIS

```
size_t hdu( ... ).header( ... )
                 .get_svalue( char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`hdu().header().get_svalue()` removes single quotations (') and unnecessary blank characters from a string value of a specified header, and then copies it into `dest_buf`. The buffer size of `dest_buf` is given by `buf_size`.

Unlike `strncpy()`, this member function always terminates with `'\0'` even if the buffer size is not enough.

PARAMETER

[O]	`dest_buf`	address of a string receive buffer
[I]	`buf_size`	size of a string receive buffer

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	String length which can be copied if the buffer size is enough. (`'\0'` is not included.)
Negative value	:	Error (If string was not copied because of wrong argument.)

EXAMPLES

Following code acquires the value of the record whose name is `CTYPE1` in the primary HDU header.

```
char dest_buf[128];
fits_image &primary = fits.image("Primary");

primary.header("CTYPE1").get_svalue(dest_buf, sizeof(dest_buf));
```

13.4.6 hdu().header().dvalue()

NAME

hdu().header().dvalue() — Real value of a header (high level)

SYNOPSIS

```
double hdu( ... ).header( ... ).dvalue() const;
```

RETURN VALUE

`hdu().header().dvalue()` returns a real value of a specified header record.

EXAMPLES

Following code acquires the value of record whose name is `CDELT1` in the primary HDU header.

```
fits_image &primary = fits.image("Primary");
double value;

value = primary.header("CDELT1").dvalue();
```

See also the example in §5.5 in Tutorial.

13.4.7 hdu().header().lvalue(), hdu().header().llvalue()

NAME

hdu().header().lvalue(), hdu().header().llvalue() — Integer value of a header (high level)

SYNOPSIS

```
long hdu( ... ).header( ... ).lvalue() const;
long long hdu( ... ).header( ... ).llvalue() const;
```

RETURN VALUE

`hdu().header().lvalue()` or `hdu().header().llvalue()` returns an integer value of a specified header record.

EXAMPLES

See EXAMPLES in §13.4.6 or §5.5 in Tutorial.

13.4.8 hdu().header().bvalue()

NAME

hdu().header().bvalue() — Boolean value of a header (high level)

SYNOPSIS

```
bool hdu( ... ).header( ... ).bvalue() const;
```

RETURN VALUE

`hdu().header().bvalue()` returns a boolean value of a specified header record.

EXAMPLES

See EXAMPLES in §13.4.6 or §5.5 in Tutorial.

13.4.9 hdu().header().assign(), hdu().header().assignf()

NAME

hdu().header().assign() — Assign a string value to a header (high level)

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign( const char *str );
fits_header_record &hdu( ... ).header( ... ).assignf( const char *format, ... );
```

DESCRIPTION

`hdu().header().asign()` or `hdu().header().asignf()` assigns string value `str` to a specified header record. Give a keyword to the argument of `header()` in case of appending of a new header record.

There is no limit to the string length. (Refer §10.1.) Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

Specify the arguments after `format` in the same way as `printf()` in libc.

List of conversion specifiers beginning with "`%`" in `format` and their functions are shown in the following table. To output "`%`" itself, give "`%%`" as conversion specifier.

Conv. Spec.	Description	Type of Argument
`hhd`	Converts the argument to a signed decimal number	char
`hd`	Converts the argument to a signed decimal number	short
`d`	Converts the argument to a signed decimal number	int
`ld`	Converts the argument to a signed decimal number	long
`lld`	Converts the argument to a signed decimal number	long long
`zd`	Converts the argument to a signed decimal number	ssize_t
`hhu`	Converts the argument to an unsigned decimal number	unsigned char
`hu`	Converts the argument to an unsigned decimal number	unsigned short
`u`	Converts the argument to an unsigned decimal number	unsigned int
`lu`	Converts the argument to an unsigned decimal number	unsigned long
`llu`	Converts the argument to an unsigned decimal number	unsigned long long
`zu`	Converts the argument to an unsigned decimal number	size_t
`hho`	Converts the argument to an unsigned octadecimal number	(unsigned) char
`ho`	Converts the argument to an unsigned octadecimal number	(unsigned) short
`o`	Converts the argument to an unsigned octadecimal number	(unsigned) int
`lo`	Converts the argument to an unsigned octadecimal number	(unsigned) long
`llo`	Converts the argument to an unsigned octadecimal number	(unsigned) long long
`zo`	Converts the argument to an unsigned octadecimal number	size_t, ssize_t
`hhx`, `hhX`	Converts the argument to an unsigned hexadecimal number	(unsigned) char
`hx`, `hX`	Converts the argument to an unsigned hexadecimal number	(unsigned) short
`x`, `X`	Converts the argument to an unsigned hexadecimal number	(unsigned) int
`lx`, `lX`	Converts the argument to an unsigned hexadecimal number	(unsigned) long
`llx`, `llX`	Converts the argument to an unsigned hexadecimal number	(unsigned) long long
`zx`, `zX`	Converts the argument to an unsigned hexadecimal number	size_t, ssize_t
`c`	Converts the argument to an integer and use the value as an ordinal value for a character	int
`s`	Writes characters from the string addressed by the argument up to a null character is encountered or the number of characters specified have been copied	const char*
`f`	Converts a float or double argument to a decimal number in the format [-]ddd.ddd.	float, double
`e`, `E`	Converts a float or double argument to a decimal number in the format [-]d.ddddd `e`[±]dd.	float, double
`g`, `G`	Picks converted result which gives less number of characters among `%e` and `%f`	float, double
`a`, `A`	Converts a float or double argument to a hexadecimal number in to [-]`0x` d.ddddd `p`[±]dd format	float, double
`p`	Converts the void * argument to a hexadecimal number	void*
`n`	Save the number of characters written so far to the integer specified by the int * argument	int*

In addition, by inserting the following optional conversion specifiers between "%" and the conversion specifier, more detailed format setting can be done.

```
sio.printf("%_f...\n",x);
           ↑
      [−][+][[0]m][.][n]
```

Option	Description	Examples
− (minus sign)	Aligns field contents to the left instead of the right 1234_____	`.printf("%-6d...`
+	Always precedes the result of a signed conversion with a plus sign or minus sign	`.printf("%+6d...`
m (number of digit)	Reserves at least m number of field width. If the converted value has fewer bytes than the field width, it will be padded with spaces on the left. To specify zero-padding, use "0" _____1234	`.printf("%10d...` `.printf("%010d...`
. (period)	Delimits a field width and number of characters or number of decimals	`.printf("%10.5f...`

n (number of digit)	Number of decimals in case of "f", A precision in case of "e", "E", "g" or "G", Number of characters in case of a string.	`.printf("%10.5f...`
		`.printf("%.15g...`
		`.printf("%10.5s...`

PARAMETER

[I]	`str`	string to be set
[I]	`format`	format string
[I]	`...`	all element data in the format

([I] : input, [O] : output)

RETURN VALUE

`hdu().header().assign()` and `hdu().header().assignf()` returns a reference to relevant fits_header_record.

EXCEPTION

If the API fails fails to reserve internal buffer or to convert each element data in the specified format, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code register the header record which keyword is `TELESCOP` and value is `HST` to the Primary HDU. This method can be used not only in case of new assignment but also for updating value.

```
fits.image("Primary").header("TELESCOP").assign("HST");
```

See also the example in §5.5 in Tutorial.

13.4.10 hdu().header().assign()

NAME

hdu().header().assign() — Assigns a boolean value to a header. (high level)

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign( bool value );
```

DESCRIPTION

`hdu().header().assign()` assigns boolean value `value` to the specified header record. Give a keyword to the argument of `header()` when appending of a new header record.

PARAMETER

[I] `value` boolean value to be set

([I] : input, [O] : output)

EXCEPTION

If the API fails fails to reserve or operate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

RETURN VALUE

`hdu().header().assign()` returns a reference to relevant fits_header_record.

EXAMPLES

See EXAMPLES in §13.4.9 or §5.5 in Tutorial

13.4.11 hdu().header().assign()

NAME

hdu().header().assign() — Assigns an integer value to a header. (high level)

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign( int value );
fits_header_record &hdu( ... ).header( ... ).assign( long value );
fits_header_record &hdu( ... ).header( ... ).assign( long long value );
```

DESCRIPTION

`hdu().header().assign()` assigns integer value `value` to the specified header record.

Give a keyword to the argument of `header()` in case of appending of a new header record.

PARAMETER

[I] `value` Integer value to be set

([I] : input, [O] : output)

RETURN VALUE

`hdu().header().assign()` returns a reference to relevant fits_header_record.

EXCEPTION

If the API fails fails to reserve or operate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code register the header record which keyword is `CCDPICNO` and value is `35` to the Primary HDU. This method can be used not only in case of assignment of a new header record but also for updating the value.

```
fits.image("Primary").header("CCDPICNO").assign(35);
```

See also the example in§5.5 Tutorial.

13.4.12 hdu().header().assign()

NAME

hdu().header().assign() — Assigns a real value to a header. (high level)

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign( double value, int prec = 15 );
fits_header_record &hdu( ... ).header( ... ).assign( float value, int prec = 6 );
```

DESCRIPTION

`hdu().header().assign()` assigns real value `value` to the specified header record. Number of digit can be specified to `prec`. If `prec` is omitted, `value` is written to header record as 15-digit number if its type is double and as 6-digit number if its type is float.

Give a keyword to the argument of `header()` in case of appending of a new header record.

PARAMETER

[I] `value` Real value to be set
[I] `prec` Precision (number of digit)
([I] : input, [O] : output)

RETURN VALUE

`hdu().header().assign()` returns a reference to relevant fits_header_record.

EXCEPTION

If the reservation or the operation of internal buffer is failed, this API throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

Following code register the header record which keyword is `CDELT1` and value is `-0.01` to the Primary HDU. This way can be used in case of not only registration of a new header record but also updating the value.

```
fits.image("Primary").header("CDELT1").assign(-0.01);
```

See also the example in §5.5 in Tutorial.

13.4.13 hdu().header().type()

NAME

hdu().header().type() — Type of header record

SYNOPSIS

```
int hdu( ... ).header( ... ).type() const;
```

DESCRIPTION

`hdu().header().type()` checks a type of specified header record.

It returns `FITS::DOUBLE_T` in case of real number, `FITS::LONGLONG_T` in case of integer number, `FITS::DOUBLECOMPLEX_T` in case of complex number, `FITS::BOOL_T` in case of boolean number, and `FITS::STRING_T` in case of others.

RETURN VALUE

`FITS::DOUBLE_T`	:	In case that header record is real number.
`FITS::LONGLONG_T`	:	In case that header record is integer number.
`FITS::DOUBLECOMPLEX_T`	:	In case that header record is complex number.
`FITS::BOOL_T`	:	In case that header record is boolean number.
`FITS::STRING_T`	:	In case of others.

EXAMPLES

Following code checks a type of header record.

```
switch ( fits.hdu("Primary").header("TELESCOP").type() ) {
  case FITS::DOUBLE_T:
      printf("Record is real number type.\n");
      break;
  case FITS::LONGLONG_T:
      printf("Record is integer number type.\n");
      break;
  case FITS::DOUBLECOMPLEX_T:
      printf("Record is complex number type.\n");
      break;
  case FITS::BOOL_T:
      printf("Record is boolean type.\n");
      break;
  case FITS::STRING_T:
      printf("Record is string type.\n");
      break;
  default:
      printf("Record is unknown type.\n");
      break;
}
```

13.4.14 hdu().header().status()

NAME

hdu().header().status() — Status of header record

SYNOPSIS

```
bool hdu( ... ).header( long index ).status() const;
```

DESCRIPTION

`hdu().header().status()` checks header record specified by `index` whether it is normal format (in the format of `A = B`), description format (description of COMMENT or HISTORY) or NULL format (no keyword nor value exist), and returns `FITS::NORMAL_RECORD`, `FITS::DESCRIPTION_RECORD` or `FITS::NULL_RECORD`, respectively.

RETURN VALUE

`FITS::NORMAL_RECORD`	:	In case that header record is normal format.
`FITS::DESCRIPTION_RECORD`	:	In case that header record is description format.
`FITS::NULL_RECORD`	:	In case that header record is NULL format.

EXAMPLES

Following code checks a status of header record.

```
switch ( fits.hdu("Primary").header("TELESCOP").status() ) {
  case FITS::NORMAL_RECORD:
      printf("Record is normal format.\n");
      break;
  case FITS::DESCRIPTION_RECORD:
      printf("Record is description format.\n");
```

```
        break;
    case FITS::NULL_RECORD:
        printf("Record is NULL format.\n");
        break;
    default:
        printf("This message should not be shown.\n");
        break;
}
```

13.4.15 hdu().header().keyword()

NAME

hdu().header().keyword() — Keyword name of a header

SYNOPSIS

```
const char *hdu( ... ).header( long index ).keyword() const;
```

DESCRIPTION

`hdu().header().keyword()` returns the keyword name of header record specified by `index`.

Since a return value is an address of an object's internal buffer, it is invalid in case that object is discarded or the header record is changed.

RETURN VALUE

`hdu().header().keyword()` returns the address of a keyword name string of header record.

EXAMPLES

Following code displays pair of the keyword and the value of the header specified by index in the Primary HDU to the standard output.

```
fits_image &primary = fits.image("Primary");
printf("%s = %s\n",
       primary.header(index).keyword(), primary.header(index).value());
```

13.4.16 hdu().header().get_keyword()

NAME

hdu().header().get_keyword() — Get a keyword string of a header. (High level)

SYNOPSIS

```
ssize_t hdu( ... ).header( long index )
                  .get_keyword( char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`hdu().header().get_keyword()` copies a keyword string of a header record specified by `index` into `dest_buf`. Buffer size of `dest_buf` is specified by `buf_size`.

Unlike `strncpy()` in libc, this member function always terminates with `'\0'` even if the buffer size is not enough.

PARAMETER

[O] `dest_buf` address of receive buffer for a string
[I] `buf_size` size of receive buffer for a string
([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	String length which can be copied if the buffer size is enough. (`'\0'` is not included)
Negative value	:	Error (If string was not copied because of wrong argument.)

EXAMPLES

Following code acquires the keyword of the header specified by index in the Primary HDU into the prepared buffer.

```
char dest_buf[128];
fits_image &primary = fits.image("Primary");
primary.header(index).get_keyword(dest_buf, sizeof(dest_buf));
```

13.4.17 hdu().header().value()

NAME

hdu().header().value() — A value of a header (Low level)

SYNOPSIS

```
const char *hdu( ... ).header( ... ).value() const;
```

DESCRIPTION

`hdu().header().value()` returns the raw value of a specified header record.

If a value from which single quotations (') and unnecessary blank characters are removed is needed, use `hdu().header().svalue()`(§13.4.4)

Since a return value is an address of an object's internal buffer, it is invalid if the object is discarded or the header record is changed.

RETURN VALUE

`hdu().header().value()` returns an address of a string value of a header record.

EXAMPLES

See EXAMPLES in §13.4.15.

13.4.18 hdu().header().get_value()

NAME

hdu().header().get_value() — A value of a header (Low level)

SYNOPSIS

```
ssize_t hdu( ... ).header( ... )
                  .get_value( char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`hdu().header().get_value()` copies a raw value of a specified header record to `dest_buf`. Buffer size of `dest_buf` should be given by `buf_size`.

If a value from which single quotations (') and unnecessary blank characters are removed is needed, use `hdu().header().get_svalue()`(§13.4.5).

Unlike `strncpy()` in libc, this member function always terminates with `'\0'` even if the buffer size is not enough.

PARAMETER

[O] `dest_buf` Address of receive buffer for a string
[I] `buf_size` Size of receive buffer for a string
([I] : input, [O] : output)

RETURN VALUE

Non-negative value : String length which can be copied if the buffer size is enough. (`'\0'` is not included.)
Negative value : Error (If string was not copied because of wrong argument.)

EXAMPLES

See EXAMPLES in §13.4.16.

13.4.19 hdu().header().comment()

NAME

hdu().header().comment() — A comment of a header

SYNOPSIS

```
const char *hdu( ... ).header( ... ).comment() const;
```

DESCRIPTION

`hdu().header().comment()` returns a comment string of a specified header record.

Since the return value is an address of an object's internal buffer, it is invalid in case if the object is discarded or the header record is changed.

RETURN VALUE

`hdu().header().comment()` returns an address of a comment string.

EXAMPLES

See EXAMPLES in §13.4.15.

13.4.20 hdu().header().get_comment()

NAME

hdu().header().get_comment() — Obtain a comment of a header

SYNOPSIS

```
ssize_t hdu( ... ).header( ... )
                  .get_comment( char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`hdu().header().get_comment()` copies a comment string of a specified header record to `dest_buf`. Buffer size of `dest_buf` should be given by `buf_size`.

Unlike `strncpy()` in libc, this member function always terminates with `'\0'` even if the buffer size is not enough.

PARAMETER

[O] `dest_buf` Address of receive buffer for a string
[I] `buf_size` Size of receive buffer for a string
([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	String length which can be copied if the buffer size is enough. (`'\0'` is not included.)
Negative value	:	Error (If string was not copied because of wrong argument.)

EXAMPLES

See EXAMPLES in §13.4.16.

13.4.21 hdu().header().assign_value()

NAME

hdu().header().assign_value() — Assign a string to a header. (Low level)

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign_value( const char *value );
fits_header_record &hdu( ... ).header( ... )
                               .assignf_value( const char *format, ... );
```

DESCRIPTION

`hdu().header().asign_value()` assigns raw value `value` to a specified header record. To set a string for a type of header record, give an argument for `value` like `"'ABC'"`. To set a number or a boolean of header record type, give an argument for `value` like `"256"`, `"3.14"` or `"T"`.

Give a keyword to the argument of `header()` in case of appending of a new header record.

There is no limit to the string length. (Refer §10.1.) Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

Arguments after `format` should be given as well as that of `printf()` in libc.

PARAMETER

[I]	`value`	A string to be set
[I]	`format`	A format string
[I]	`...`	All element data in the format

([I] : input, [O] : output)

RETURN VALUE

`hdu().header().asign_value()` returns a reference to relevant fits_header_record.

EXCEPTION

If the API fails fails to reserve internal buffer or to convert each element in the specified format, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.hdu("Primary").header("TELESCOP").assign_value("'HST'");
```

13.4.22 hdu().header().assign_comment()

NAME

hdu().header().assign_comment() — Change a comment of a header

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign_comment( const char *comment );
fits_header_record &hdu( ... ).header( ... )
                               .assignf_comment( const char *format, ... );
```

DESCRIPTION

`hdu().header().asign_comment()` assigns string `comment` to a specified header record.

Give a keyword to the argument of `header()` in case of appending of a new header record.

Arguments after `format` should be specified in the same way as `printf()` in libc.

PARAMETER

[I]	`comment`	A string to be set
[I]	`format`	A format string
[I]	`...`	All element data in the format

([I] : input, [O] : output)

RETURN VALUE

`hdu().header().asign_comment()` returns a reference to relevant fits_header_record.

EXCEPTION

If the API fails fails to reserve internal buffer or to convert each element in the specified format, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.hdu("Primary").header("TELESCOP").assign("HST")
                                      .assign_comment("Name of telescope");
```

See also the example in §5.5 in Tutorial.

13.4.23 hdu().header_update()

NAME

hdu().header_update() — Update or append a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_update( const char *keyword,
                                    const char *value, const char *comment );
```

DESCRIPTION

`hdu().header_update()` updates a raw value by `value` and a comment by `comment` of a record specified by `keyword`. If updating is not necessary, NULL is given to `value` or `comment`. If `keyword` is not found in the header, it will be appended.

To set a string of header record type, give an argument for `value` like `"'ABC'"`. To set a number or a boolean of header record type, give an argument for `value` like `"256"`, `"3.14"` or `"T"`.

PARAMETER

[I]	`keyword`	A keyword
[I]	`value`	A string value to be set
[I]	`comment`	A comment value to be set

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_update()` returns a reference to relevant fits_hdu.

EXCEPTION

If the API fails fails to reserve internal buffer (for example, reservation of area for a record to be appended is failed because of lack of memory), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.hdu("Primary").header_update("TELESCOP", "HST", "Name of telescope");
```

13.4.24 hdu().header_assign()

NAME

hdu().header_assign() — Update a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_assign( long index, const fits::header_def &def );
fits_hdu &hdu( ... ).header_assign( const char *header_keyword,
                                    const fits::header_def &def );
fits_hdu &hdu( ... ).header_assign( long index, const fits_header_record &obj );
fits_hdu &hdu( ... ).header_assign( const char *header_keyword,
                                    const fits_header_record &obj );
fits_hdu &hdu( ... ).header_assign( long index,
                  const char *keyword, const char *value, const char *comment );
fits_hdu &hdu( ... ).header_assign( const char *header_keyword,
                  const char *keyword, const char *value, const char *comment );
fits_hdu &hdu( ... ).header_assign( long index,
                                  const char *keyword, const char *description );
fits_hdu &hdu( ... ).header_assign( const char *header_keyword,
                                  const char *keyword, const char *description );
```

DESCRIPTION

`hdu().header_assign()` updates a header record specified by `index` or `header_keyword` to a content specified by each argument.

PARAMETER

[I]	`index`	An index of header record to be updated
[I]	`header_keyword`	A keyword of header record to be updated
[I]	`def`	fits::table_def structure to be copied at updating
[I]	`obj`	An object to be copied at updating
[I]	`keyword`	A new keyword value
[I]	`value`	A new record value
[I]	`comment`	A new comment value
[I]	`description`	A new Description value

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_assign()` returns a reference to relevant fits_hdu.

EXCEPTION

If the API fails fails to operate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

Following code updates header record specified by `index`

```
fits.hdu("Primary").header_assign(index, "TELESCOP", "HST", "Name of telescope");
```

13.4.25 hdu().header_init()

NAME

hdu().header_init() — Initialize a header

SYNOPSIS

```
fits_hdu &hdu( ... ).header_init();
fits_hdu &hdu( ... ).header_init( const fits::header_def defs[] );
fits_hdu &hdu( ... ).header_init( const fits_header &obj );
```

DESCRIPTION

`hdu().header_init()` deletes the header and if `defs` is specified, assign that value to the header.

Definition of `fits::header_def` is follows:

```
typedef struct {
    const char *keyword;
    const char *value;
    const char *comment;
} fits::header_def;
```

All of last members in array `defs` should be `NULL`.

PARAMETER

[I] `defs` Array of `fits::table_def` structure to be copied
[I] `obj` An object to be copied
([I] : input, [O] : output)

RETURN VALUE

`hdu().header_init()` returns a reference to relevant fits_hdu.

EXCEPTION

If the API fails fails to manipulate internal buffer (for example, reservation of area for specified def is failed because of lack of memory), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits::header_def defs[] = { {"TELESCOP", "'SUBARU'", "Name of telescope"},
                            {"OBSERVAT", "'NAOJ'", "Observatory name"},
                            {"COMMENT", "--------------------------------"},
                            {NULL} };
fits.hdu("Primary").header_init(defs);
```

If the value is string, both `"'SUBARU'"` and `"SUBARU"` can be used. However, in order to store a number like `123` into a file as a string, set the value like `"'123'"`.

In case of description of a comment or a history, give `NULL` to either `value` or `comment`.

See also the example in §5.9 in Tutorial.

13.4.26 hdu().header_swap()

NAME

hdu().header_swap() — Swap headers

SYNOPSIS

```
fits_hdu &hdu( ... ).header_swap( fits_header &obj );
```

DESCRIPTION

`hdu().header_swap()` swaps a FITS header object for one which specified as `obj`. The swap operation is not applied for header records having properties of FITS data unit.

PARAMETER

[I/O] `obj` an image object for swap
([I] : input, [O] : output)

RETURN VALUE

`hdu().header_swap()` returns a reference to the modified fits_hdu object

EXAMPLES

The following code swaps a header of HDU "AAAA" for one of HDU "BBBB".

```
fits_header &obj = fits.hdu("BBBB").header();
fits.hdu("AAAA").header_swap(obj);
```

13.4.27 hdu().header_append_records()

NAME

hdu().header_append_records() — Append a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_append_records( const fits::header_def defs[] );
fits_hdu &hdu( ... ).header_append_records( const fits_header &obj );
```

DESCRIPTION

`hdu().header_append_records()` adds contents of `defs` to a header. All last members in the array of `defs` are set to NULL. For more information on `fits::header_def`, see §13.4.25(hdu().header_init()).

PARAMETER

[I] `defs` An array of `fits::table_def` structure to be copied
[I] `obj` A header object to be copied
([I] : input, [O] : output)

RETURN VALUE

`hdu().header_append_records()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer (for example, this API cannot allocate memory space for records to be appended due to lack of memory), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

```
fits::header_def defs[] = { {"TELESCOP", "'SUBARU'", "Name of telescope"},
                            {"OBSERVAT", "'NAOJ'", "Observatory name"},
                            {"COMMENT", "--------------------------------"},
                            {NULL} };
fits.hdu("Primary").header_append_records(defs);
```

See also the example in §5.7 in Tutorial.

13.4.28 hdu().header_append()

NAME

hdu().header_append() — Append a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_append( const char *keyword );
fits_hdu &hdu( ... ).header_append( const char *keyword, const char *value,
                                    const char *comment );
fits_hdu &hdu( ... ).header_append( const char *keyword, const char *description );
fits_hdu &hdu( ... ).header_append( const fits::header_def &def );
fits_hdu &hdu( ... ).header_append( const fits_header_record &obj );
```

DESCRIPTION

`hdu().header_append()` appends a header record with a keyword `keyword`, a value `value`, and a comment `comment` to a header. In case `description` was specified, this API appends a header record in the format of COMMENT or HISTORY.

There is no limit to the string length of `value` and `description` (§10.1). Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

PARAMETER

[I]	`keyword`	A keyword of a record to be appended
[I]	`value`	A value of a record to be appended
[I]	`comment`	A comment of a record to be appended
[I]	`description`	A description of a record to be appended
[I]	`def`	`fits::table_def` structure to be appended
[I]	`obj`	A header record object to be appended

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_append()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer (for example, it cannot allocate memory space for records to be appended due to lack of memory), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

```
fits.hdu("Primary").header_append("TELESCOP", "'SUBARU'", "Name of telescope");
```

See also the example in §5.7 in Tutorial.

13.4.29 hdu().header_insert_records()

NAME

hdu().header_insert_records() — Insert a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_insert_records( long index,
                                            const fits::header_def defs[] );
fits_hdu &hdu( ... ).header_insert_records( const char *keyword,
                                            const fits::header_def defs[] );
fits_hdu &hdu( ... ).header_insert_records( long index, const fits_header &obj );
```

```
fits_hdu &hdu( ... ).header_insert_records( const char *keyword,
                                            const fits_header &obj );
```

DESCRIPTION

`hdu().header_insert_records()` inserts a contents of `defs` into a record specified by `index` or `keyword`. All last members in the array of `defs` are set to `NULL`. For more information about `fits::header_def`, see §13.4.25(hdu().header_init()).

PARAMETER

[I]	`index`	An index of a record to be inserted
[I]	`keyword`	A keyword of a record to be inserted
[I]	`defs`	An array of `fits::table_def` structure to be copied at the insertion
[I]	`obj`	An object to be copied at the insertion

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_insert_records()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer(for example, this API cannot allocate memory space for records to be appended due to lack of memory), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

```
fits::header_def defs[] = { {"TELESCOP", "'SUBARU'", "Name of telescope"},
                            {"OBSERVAT", "'NAOJ'", "Observatory name"},
                            {"COMMENT", "--------------------------------"},
                            {NULL} };
long insert_index = 1;
fits.hdu("Primary").header_insert_records(insert_index, defs);
```

13.4.30 hdu().header_insert()

NAME

hdu().header_insert() — Insert a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_insert( long index,
                                    const char *keyword,
                                    const char *value, const char *comment );
fits_hdu &hdu( ... ).header_insert( const char *record_keyword,
                                    const char *keyword,
                                    const char *value, const char *comment );
fits_hdu &hdu( ... ).header_insert( long index,
                                    const char *keyword, const char *description );
fits_hdu &hdu( ... ).header_insert( const char *record_keyword,
                                    const char *keyword, const char *description );
fits_hdu &hdu( ... ).header_insert( long index, const fits::header_def &def );
fits_hdu &hdu( ... ).header_insert( const char *record_keyword,
                                    const fits::header_def &def );
fits_hdu &hdu( ... ).header_insert( long index, const fits_header_record &obj );
fits_hdu &hdu( ... ).header_insert( const char *record_keyword,
                                    const fits_header_record &obj );
```

DESCRIPTION

`hdu().header_insert()` inserts a header record with a keyword `keyword`, a value `value`, and a comment `comment` into a record specified by `index0` or `keyword0`. In case `description` was specified, this API inserts a header record in the format of COMMENT or HISTORY.

PARAMETER

[I]	`index`	An index in header records for the insertion
[I]	`record_keyword`	A keyword in header records for the insertion
[I]	`def`	`fits::header_def` structure to be copied at the insertion
[I]	`obj`	An object to be copied at the insertion
[I]	`keyword`	A keyword to be inserted
[I]	`value`	A value to be inserted
[I]	`comment`	A comment to be inserted
[I]	`description`	A description to be inserted

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_insert()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer (for example, this API cannot allocate memory space for records to be appended due to lack of memory), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

```
long insert_index = 1;
fits.hdu("Primary").header_insert(insert_index, "TELESCOP",
                                  "'SUBARU'", "Name of telescope");
```

13.4.31 hdu().header_erase_records()

NAME

hdu().header_erase_records() — Delete header records

SYNOPSIS

```
fits_hdu &hdu( ... ).header_erase_records( long index, long num_records );
fits_hdu &hdu( ... ).header_erase_records( const char *keyword, long num_records );
```

DESCRIPTION

`hdu().header_erase_records()` deletes number of `num_records` header records from a record specified by `index` or `keyword`.

PARAMETER

[I]	`index`	An index in header records. The API deletes records starting with the index
[I]	`keyword`	A keyword in header records. The API deletes records starting with the keyword.
[I]	`num_records`	The number of records to be deleted.

([I] : input, [O] : output)

RETURN VALUE

`hdu().header_erase_records()` returns a reference to the modified fits_hdu object

EXCEPTION

If the API fails fails to manipulate the internal buffer (for example, this API cannot resize memory space after deleting records), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

The following code deletes two records from a record of 0-th index in Primary HDU header.

```
fits.hdu("Primary").header_erase_records(0L, 2);
```

13.4.32 hdu().header_erase()

NAME

hdu().header_erase() — Delete header records

SYNOPSIS

```
fits_hdu &hdu( ... ).header_erase( long index );
fits_hdu &hdu( ... ).header_erase( const char *keyword );
```

DESCRIPTION

`hdu().header_erase()` deletes a header records specified by `index` or `keyword`.

PARAMETER

[I] `index` An index to be deleted in header records.
[I] `keyword` A keyword to be deleted in header records.
([I] : input, [O] : output)

RETURN VALUE

`hdu().header_erase()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer (for example, this API cannot resize memory space after deleting records), it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

The following code deletes a record of 0-th index in Primary HDU header.

```
fits.hdu("Primary").header_erase(0L);
```

13.4.33 hdu().header_rename()

NAME

hdu().header_rename() — Rename a keyword of a header record

SYNOPSIS

```
fits_hdu &hdu( ... ).header_rename( long index0, const char *new_name );
fits_hdu &hdu( ... ).header_rename( const char *keyword0, const char *new_name );
```

DESCRIPTION

hdu().header_rename() member function renames the keyword of the header record specified by `index0` or `keyword0` to `new_name`.

PARAMETER

[I] `index0` An index to be renamed in header records.
[I] `keyword0` A keyword to be renamed in header records.
[I] `new_name` A new keyword.
([I] : input, [O] : output)

RETURN VALUE

`hdu().header_rename()` returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer, it throws an exception derived from SLLIB or SFITSIO(`sli::err_rec` exception).

EXAMPLES

The following code renames the keyword "RADECSYS" in Primary HDU to "RADESYS".

```
fits.hdu("Primary").header_rename("RADECSYS", "RADESYS");
```

13.4.34 hdu().header()

NAME

hdu().header() — A reference to a header object

SYNOPSIS

```
fits_header &hdu( ... ).header();
```

RETURN VALUE

`hdu().header()` returns a reference to a header object.

EXAMPLES

The following code copies a reference to a header object in Primary HDU.

```
fits_header &primary_header = fits.hdu("Primary").header();
```

13.4.35 hdu().header_formatted_string()

NAME

hdu().header_formatted_string() — Formatted header string

SYNOPSIS

```
const char *hdu( ... ).header_formatted_string();
```

DESCRIPTION

`hdu().header_formatted_string()` creates a formatted string in FITS file format for all records in header in the object and returns the address. Returned result is a string of $80 \times n$ characters without `'\n'` but with `'\0'` termination. The `CONTINUE` keyword will be included when handling long records.

This API facilitates cooperation with WCSLIB.

RETURN VALUE

`hdu().header_formatted_string()` returns an address to the formatted strings.

EXCEPTION

If the API fails fails to manipulate the internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

The following code displays the number of lines of the formatted all header records in Primary HDU and the whole string. tstring class allows programmers to manipulate strings easily.

```
tstring hdr_all = fits.image("Primary").header_formatted_string();
printf("Number of Lines = %d\n", (int)hdr_all.length() / 80);
printf("Formatted header. :\n");
printf("%s\n", hdr_all.cstr());
```

See also the example using WCSLIB in §5.14 in Tutorial.

For more information about tstring class, see APPENDIX3 (§16).

13.4.36 hdu().header().get_section_info()

NAME

hdu().header().get_section_info() — Parse IRAF-style information for a rectangular area

SYNOPSIS

```
int hdu( ... ).header( ... ).get_section_info( long dest_buf[], int buf_len ) const;
```

DESCRIPTION

This member function parses IRAF-style information for a rectangular area such as `BIASSEC` keyword record, and stores the result into an array `dest_buf` whose length is `buf_len`.

For example, (3073,31,0,512) is stored into `dest_buf` for the header record:
`BIASSEC = '[3074:3104,1:512] '`.

Values in `dest_buf` can be directly given to image().stat_pixels() member function (§13.6.41) to compute pixel statistics.

PARAMETER

[O]	`dest_buf`	An array for the result
[I]	`buf_len`	Length of buffer of `dest_buf` (should be 4)

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Number of values stored in the array (should be 4)
Negative value	:	Error (Parse failed.)

EXAMPLES

See the example code in exercise 3 in §1.1.

13.4.37 hdu().header().assign_system_time()

NAME

hdu().header().assign_system_time() — Set current time in UTC

SYNOPSIS

```
fits_header_record &hdu( ... ).header( ... ).assign_system_time();
```

DESCRIPTION

This member function sets current time (UTC) in `yyyy-mm-ddThh:mm:ss` format.

RETURN VALUE

`hdu().assign_system_time()` returns a reference to a header object.

EXAMPLES

Next code appends `DATE` keyword record, and sets current time in UTC.

```
fits.image("Primary").header_append("DATE")
                     .header("DATE").assign_system_time();
```

13.4.38 hdu().header_fill_blank_comments()

NAME

hdu().header_fill_blank_comments() — Fill blank comments using comment dictionaries

SYNOPSIS

```
fits_hdu &hdu( ... ).header_fill_blank_comments(int hdutype = FITS::ANY_HDU);
```

DESCRIPTION

hdu().header_fill_blank_comments() fills blank comments in the FITS header with contents provided by SFITSIO built-in comment dictionaries. There are four dictionaries in SFITSIO library that consist of dedicated dictionaries for three HDU types and a dictionary for fall-back contents.

The argument `hdutype` takes `FITS::IMAGE_HDU`, `FITS::BINARY_TABLE_HDU` or `FITS::ASCII_TABLE_HDU` to select each dedicated dictionary. If `FITS::ANY_HDU` is set or argument is omitted, a suitable comment dictionary is automatically selected.

See §13.4.40 to update SFITSIO built-in comment dictionaries.

PARAMETER

[I] `hdutype` Type of HDU
([I] : input, [O] : output)

RETURN VALUE

hdu().header_fill_blank_comments() returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

The following code fills blank comments in Primary HDU.

```
fits.image("Primary").header_fill_blank_comments(FITS::IMAGE_HDU);
```

13.4.39 hdu().header_assign_default_comments()

NAME

hdu().header_assign_default_comments() — Overwrite comments using comment dictionaries

SYNOPSIS

```
fits_hdu &hdu( ... ).header_assign_default_comments(int hdutype = FITS::ANY_HDU);
```

DESCRIPTION

hdu().header_assign_default_comments() overwrites comments in the FITS header with contents provided by SFITSIO built-in comment dictionaries. There are four dictionaries in SFITSIO library that consist of dedicated dictionaries for three HDU types and a dictionary for fall-back contents.

The argument `hdutype` takes `FITS::IMAGE_HDU`, `FITS::BINARY_TABLE_HDU` or `FITS::ASCII_TABLE_HDU` to select each dedicated dictionary. If `FITS::ANY_HDU` is set or argument is omitted, a suitable comment dictionary is automatically selected.

See §13.4.40 to update SFITSIO built-in comment dictionaries.

PARAMETER

[I] `hdutype` Type of HDU
([I] : input, [O] : output)

RETURN VALUE

hdu().header_assign_default_comments() returns a reference to the modified fits_hdu object.

EXCEPTION

If the API fails fails to manipulate the internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

The following code overwrites comments in Primary HDU.

```
fits.image("Primary").header_assign_default_comments(FITS::IMAGE_HDU);
```

13.4.40 fits::update_comment_dictionary()

NAME

fits::update_comment_dictionary() — Update comment dictionaries

SYNOPSIS

```
#include <sli/fits_header_record.h>
const asarray_tstring &update_comment_dictionary( int hdutype,
                                        const char *const *new_kwd_comments );
```

DESCRIPTION

This function updates contents of SFITSIO built-in comment dictionaries. There are four dictionaries in SFITSIO library that consist of dedicated dictionaries for three HDU types and a dictionary for fall-back contents. The dictionaries are used in hdu().header_fill_blank_comments(), hdu().header_assign_default_comments() member functions. See also §13.4.38 and §13.4.39.

The argument `hdutype` takes `FITS::IMAGE_HDU`, `FITS::BINARY_TABLE_HDU`, `FITS::ASCII_TABLE_HDU` or `FITS::ANY_HDU` for each dictionary to be updated.

The argument `new_kwd_comments` takes pointer array of strings including keywords and comments such as *keyword1*, *comment1*, *keyword2*, *comment2*, ... , `NULL`.

PARAMETER

[I]	`hdutype`	Type of HDU
[I]	`new_kwd_comments`	Pointer array of strings having keywords and comments

([I] : input, [O] : output)

RETURN VALUE

This function returns read-only reference of asarray_tstring object having updated comment dictionary.

EXCEPTION

If the API fails fails to manipulate the internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

This code updates general comment dictionary and that for binary table HDU.

```
#include <sli/fits_header_record.h>

/* comment strings for blank comment in template */
static const char *Kwd_comment_any[] = {
    /*           |MIN                                          MAX| */
    "TIME-EPH", "epoch time for FITS header",
    "SMPLBASE", "base telemetry referred to shrink rows",
    /*           |MIN                                          MAX| */
    NULL
};
static const char *Kwd_comment_bintable[] = {
    /*           |MIN                                          MAX| */
    "TTNAM#",   "original telemetry name",
    "TCONV#",   "type of data conversion",
    "TSTAT#",   "definition of status values",
    "TINPL#",   "type of interpolation",
    "TSPAN#",   "[s] maximum span for interpolation",
    "", "",
    /*           |MIN                                          MAX| */
    NULL
};

int main()
{
    /* update comment dictionaries */
    fits::update_comment_dictionary(FITS::ANY_HDU, Kwd_comment_any);
    fits::update_comment_dictionary(FITS::BINARY_TABLE_HDU, Kwd_comment_bintable);
```

13.5 Operation of header (low-level) and disk-based FITS I/O

fits_header class has some member functions to read/write FITS header unit and to skip FITS data unit for an opened stream. These member functions enable high-speed access of FITS header and flexible data access of FITS data unit using APIs provided by SLLIB stream classes.

The SFITSIO software package contains "`tools/hv.cc`" that reads FITS header with high-speed file access. Refer `hv.cc` as an example code that uses APIs in this section.

13.5.1 fits_header::read_stream()

NAME

fits_header::read_stream() — Read a FITS header unit only

SYNOPSIS

```
ssize_t fits_header::read_stream( cstreamio &sref );
```

DESCRIPTION

This member function reads and parses a FITS header unit from an opened stream (`sref`), and stores the contents of it into the object.

After calling this member function, the position indicator of stream will be set to the beginning of succeeding FITS data unit.

First argument `sref` takes a reference to object of inherited class of cstreamio class. See APPENDIX and manual of SLLIB for details of these classes for stream handling.

PARAMETER

[I] `sref` Reference to object that manages opened stream
([I] : input, [O] : output)

RETURN VALUE

Positive value	:	Byte length of input stream.
0	:	EOF of input stream.
Negative value	:	Error.

13.5.2 fits_header::write_stream()

NAME

fits_header::write_stream() — Output a FITS header unit only

SYNOPSIS

```
ssize_t fits_header::write_stream( cstreamio &sref, bool end_and_blank );
```

DESCRIPTION

This member function outputs formatted FITS header (80-characters width) to an opened stream (`sref`).

When the argument `end_and_blank` is `true`, written result contains END keyword and padding of white space for 2880 bytes block, therefore, programmer's code can output succeeding FITS data unit using the same stream.

END keyword and padding are not written when `end_and_blank` is `false`.

First argument `sref` takes a reference to object of inherited class of cstreamio class. See APPENDIX and manual of SLLIB for details of these classes for stream handling.

PARAMETER

[I] `sref` Reference to object that manages opened stream
[I] `end_and_blank` END keyword and padding are written or not
([I] : input, [O] : output)

RETURN VALUE

Non-negative value : Byte length of output stream.
Negative value : Error.

13.5.3 fits_header::skip_data_stream()

NAME

fits_header::skip_data_stream() — Skip a FITS data unit

SYNOPSIS

```
ssize_t fits_header::skip_data_stream( cstreamio &sref );
```

DESCRIPTION

This member function skips the byte data of a FITS data unit using header information in the object. Before using this member function, programmers have to use read_stream() member function (§13.5.1) to give the object the header information.

The position indicator of stream will be set to the beginning of succeeding FITS header unit when the next HDU exists. To know the existence of next HDU, test the return value of read_stream() member function (§13.5.1).

First argument `sref` takes a reference to object of inherited class of cstreamio class. See APPENDIX and manual of SLLIB for details of these classes for stream handling.

PARAMETER

[I] `sref` Reference to object that manages opened stream
([I] : input, [O] : output)

RETURN VALUE

Non-negative value : Byte length of input stream.
Negative value : Error.

13.6 APIs for manipulation of an Image HDU

In this section, we describe APIs to manipulate an Image HDU. Every argument of "`image( ... )`" of all APIs in this section is the HDU index (`long index`) or the HDU name (`const char *hduname`), so we omit the description of this argument.

13.6.1 image().hduname(), image().assign_hduname()

NAME

image().assign_hduname(), image().hduname() — Manipulation of an Image HDU name

SYNOPSIS

```
const char *image( ... ).hduname();
const char *image( ... ).extname();
fits_image &image( ... ).assign_hduname( const char *name );
fits_image &image( ... ).assign_extname( const char *name );
```

DESCRIPTION

`image().hduname()` returns a HDU name.

`image().assign_hduname()` assigns a HDU name. The argument `name` is reflected to `EXTNAME` of the header. This API can also be used for Primary HDU.

PARAMETER

[I] `name` a HDU name to be set
([I] : input, [O] : output)

RETURN VALUE

`image().hduname()` returns a pointer to a string of HDU name.
`image().assign_hduname()` returns a reference to the modified fits image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, a failure of memory allocation for modifying HDU name), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
printf("HDU name=%s\n", fits.image("Primary").hduname());
```

13.6.2 image().hduver(), image().assign_hduver()

NAME

image().assign_hduver(), image().hduver() — Manipulation of an Image HDU version number

SYNOPSIS

```
long long image( ... ).hduver();
long long image( ... ).extver();
fits_image &image( ... ).assign_hduver( long long ver );
fits_image &image( ... ).assign_extver( long long ver );
```

DESCRIPTION

`image().hduver()` returns a HDU version number.

`image().assign_hduver()` sets a HDU version number. The argument `ver` is reflected to `EXTVER` of the header. This API can also be used for Primary HDU.

PARAMETER

[I] `ver` a version number to be set

([I] : input, [O] : output)

RETURN VALUE

`image().hduver()` returns a HDU version number.

`image().assign_hduver()` returns a reference to the modified fits image object.

EXAMPLES

```
printf("HDU version=%lld\n", fits.image("Primary").hduver());
```

13.6.3 image().dim_length()

NAME

image().dim_length() — The number of axes

SYNOPSIS

```
long image( ... ).dim_length() const;
long image( ... ).axis_length() const;
```

RETURN VALUE

`image().dim_length()` returns the number of axes.

EXAMPLES

```
printf("The number of dimensions=%ld\n", fits.image("Primary").dim_length());
```

13.6.4 image().length()

NAME

image().length() — The number of pixels

SYNOPSIS

```
long image( ... ).length() const;
long image( ... ).length( long axis ) const;
```

DESCRIPTION

`image().length()` returns the number of pixels in `axis`. When `axis` is not specified, the total number of pixels is returned.

PARAMETER

[I] `axis` axis whose number of pixels is returned

([I] : input, [O] : output)

RETURN VALUE

`image().length()` returns the number of pixels

EXAMPLES

The following code displays the total number of pixels and the number of pixels in each axis of ImageHDU.

```
long axis;
printf("the total number of pixels=%ld\n", fits.image("Primary").length());
for ( axis = 0 ; axis < fits.image("Primary").dim_length() ; axis++ ) {
```

```
        printf("axis[%ld] the number of pixels[%ld]\n",
               axis, fits.image("Primary").length(axis));
    }
```

13.6.5 image().type()

NAME

image().type() — A data type of pixel values

SYNOPSIS

```
int image( ... ).type() const;
```

RETURN VALUE

`image().type()` returns a data type of pixel values. They are any one of " `FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, or `FITS::BYTE_T` ".

EXAMPLES

The following code displays a data type of pixel values.

```
    switch ( fits.image("Primary").type() ) {
      case FITS::DOUBLE_T:
          printf("Data type: DOUBLE\n");
          break;
      case FITS::FLOAT_T:
          printf("Data type: FLOAT\n");
          break;
      case FITS::LONGLONG_T:
          printf("Data type: LONGLONG\n");
          break;
      case FITS::LONG_T:
          printf("Data type: LONG\n");
          break;
      case FITS::SHORT_T:
          printf("Data type: SHORT\n");
          break;
      case FITS::BYTE_T:
          printf("Data type: BYTE\n");
          break;
      default:
          printf("Invalid data type.\n");
          break;
    }
```

13.6.6 image().bytes()

NAME

image().bytes() — A size of one pixel in bytes

SYNOPSIS

```
long image( ... ).bytes() const;
```

RETURN VALUE

`image().bytes()` returns a size of one pixel in bytes.

EXAMPLES

```
printf("The size of one pixel is %ld.\n", fits.image("Primary").bytes());
```

13.6.7 image().col_length()

NAME

image().col_length() — The number of pixels in a column(x-axis)

SYNOPSIS

```
long image( ... ).col_length() const;
```

RETURN VALUE

`image().col_length()` returns the number of pixels in a column(axis0; x-axis).

EXAMPLES

```
printf("The number of pixels in x-axis is %ld.\n",
        fits.image("Primary").col_length());
```

13.6.8 image().row_length()

NAME

image().row_length() — The number of pixels in a row(y-axis)

SYNOPSIS

```
long image( ... ).row_length() const;
```

RETURN VALUE

`image().row_length()` returns the number of pixels in a row(axis1; y-axis).

EXAMPLES

```
printf("The number of pixels in y-axis is %ld.\n",
        fits.image("Primary").row_length());
```

13.6.9 image().layer_length()

NAME

image().layer_length() — The number of pixels in a layer(z-axis)

SYNOPSIS

```
long image( ... ).layer_length() const;
```

RETURN VALUE

`image().layer_length()` returns the number of pixels in a layer(z-axis). If the number of dimensions is over 3, the ones of axis2 or later are returned.

EXAMPLES

```
printf("The number of pixels in z-axis or later is %ld.\n",
        fits.image("Primary").layer_length());
```

13.6.10 image().dvalue()

NAME

image().dvalue() — A pixel value in double precision

SYNOPSIS

```
double image( ... ).dvalue( long axis0, long axis1 = FITS::INDEF,
                            long axis2 = FITS::INDEF ) const;
double image( ... ).dvalue_v( long num_axisx,
                              long axis0, long axis1, long axis2, ... ) const;
double image( ... ).va_dvalue_v( long num_axisx, long axis0, long axis1, long axis2,
                                 va_list ap ) const;
```

DESCRIPTION

`image().dvalue()` returns a double precision pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. The value is subject to `BZERO`, `BSCALE`, and `BLANK` in a header. When the value corresponded to BLANK value or the argument value is out of range, NAN is returned.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data depending on the number of arguments. In EXAMPLES below, n-dimensional data is treated as 1-dimensional one.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().dvalue()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code sums up all pixel values.

```
double sum = 0;
for ( i=0 ; i < fits.image("Primary").length() ; i++ ) {
    sum += fits.image("Primary").dvalue(i);
}
```

See also a sample code in §5.8.

13.6.11 image().lvalue(), image().llvalue()

NAME

image().lvalue(), image().llvalue() — A pixel value in integer

SYNOPSIS

```
long image( ... ).lvalue( long axis0, long axis1 = FITS::INDEF,
                          long axis2 = FITS::INDEF ) const;
long image( ... ).lvalue_v( long num_axisx,
                            long axis0, long axis1, long axis2, ... ) const;
long image( ... ).va_lvalue_v( long num_axisx, long axis0, long axis1, long axis2,
                               va_list ap ) const;
long long image( ... ).llvalue( long axis0, long axis1 = FITS::INDEF,
                                long axis2 = FITS::INDEF ) const;
long long image( ... ).llvalue_v( long num_axisx,
                                  long axis0, long axis1, long axis2, ... ) const;
long long image( ... ).va_llvalue_v( long num_axisx,
                                     long axis0, long axis1, long axis2,
                                     va_list ap ) const;
```

DESCRIPTION

This API returns an integer pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. The value is subject to `BZERO`, `BSCALE`, and `BLANK` in a header. When the value corresponded to BLANK value or the argument value is out of range, `INDEF_LONG` or `INDEF_LLONG` is returned.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

This API returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.6.10 and also a sample code in §5.8.

13.6.12 image().assign()

NAME

image().assign() — Assign pixel number

SYNOPSIS

```
fits_image &image( ... ).assign( double value, long axis0,
                          long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_v( double value, long num_axisx,
                                   long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_v( double value, long num_axisx,
                                      long axis0, long axis1, long axis2,
                                      va_list ap );
```

DESCRIPTION

`image().assign()` modifies a pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. In this modification, the value which is reflected by `BZERO` and `BSCALE` in a header is used in order to update a pixel value in internal buffer. When `value` is NAN and BLANK value is already defined, BLANK value is assigned to internal buffer. When no BLANK value is defined at integer type, a value `INDEF_UCHAR`, `INDEF_INT16`, `INDEF_INT32` or `INDEF_INT64` is stored in internal buffer.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code assigns 0 to all pixel values in 0-th row.

```
double value = 0;
for ( i=0 ; i < fits.image("Primary").col_length() ; i++ ) {
    fits.image("Primary").assign(value, i,0);
}
```

See also a sample code in §5.8.

13.6.13 image().convert_type()

NAME

image().convert_type() — Conversion of data type

SYNOPSIS

```
fits_image &image( ... ).convert_type( int new_type );
fits_image &image( ... ).convert_type( int new_type, double new_zero );
fits_image &image( ... ).convert_type( int new_type, double new_zero,
                                       double new_scale );
fits_image &image( ... ).convert_type( int new_type, double new_zero,
                                       double new_scale, long long new_blank );
```

DESCRIPTION

`image().convert_type()` converts a data type of the image into `new_type`. A size of internal buffer is changed if necessary. A value which can be specified as `new_type` is any one of "`FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, or `FITS::BYTE_T`". When `new_zero`, `new_scale`, `new_blank` are specified, this API updates `BZERO`, `BSCALE` and `BLANK` in a header and changes the image data reflected by those values. `new_blank` is valid only when `new_type` is integer type.

PARAMETER

[I] `new_type` a new data type for conversion
[I] `new_zero` a new `BZERO` value for conversion
[I] `new_scale` a new `BSCALE` value for conversion
[I] `new_blank` a new `BLANK` value for conversion
([I] : input, [O] : output)

RETURN VALUE

`image().convert_type()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, an increase of image data size after the conversion), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code converts any type of image data in primary HDU into double.

```
fits.image("Primary").convert_type(FITS::DOUBLE_T);
```

See also a sample code in §5.11.

13.6.14 image().bzero(), image().assign_bzero()

NAME

image().bzero(), image().assign_bzero() — Manipulation of a zero-point

SYNOPSIS

```
double image( ... ).bzero() const;
bool image( ... ).bzero_is_set() const;
fits_image &image( ... ).assign_bzero( double zero, int prec = 15 );
fits_image &image( ... ).erase_bzero();
```

DESCRIPTION

`image().bzero()` returns a value of `BZERO` in a header. `image().bzero_is_set()` returns whether `BZERO` exists or not in a header.

`image().assign_bzero()` sets a value of `BZERO` in the header. The number of digit precision can be specified in `prec`. When this is skipped, 15 digit precsion will be inserted.

`image().erase_bzero()` deletes a value of `BZERO` in the header.

These APIs do not change an actual zero-point of the image. `image().convert_type()` (see §13.6.13) changes both the zero-point and the image.

PARAMETER

[I] `zero` `BZERO` value to be set
[I] `prec` the number of digits precision
([I] : input, [O] : output)

RETURN VALUE

This API returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, image().assign_bzero() cannot reallocate an address table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code checks an existence of BZERO and sets a value if it is not defined.

```
if ( fits.image("Primary").bzero_is_set() == false ) {
    fits.image("Primary").assign_bzero(0.0);
}
```

13.6.15 image().bscale(), image().assign_bscale()

NAME

image().bscale(), image().assign_bscale() — Manipulation of a scaling factor

SYNOPSIS

```
double image( ... ).bscale() const;
bool image( ... ).bscale_is_set() const;
fits_image &image( ... ).assign_bscale( double scale, int prec = 15 );
fits_image &image( ... ).erase_bscale();
```

DESCRIPTION

`image().bscale()` returns a value of `BSCALE` in a header. `image().bscale_is_set()` returns whether `BSCALE` exists or not in a header.

`image().assign_bscale()` sets a value of `BSCALE` in a header. The number of digit precision can be specified in `prec`. When this is skipped, 15 digit precsion will be inserted.

`image().erase_bscale()` deletes a value of `BSCALE` in a header.

These APIs do not change the actual scaling factor of the image. `image().convert_type()` (see §13.6.13) changes both the scaling factor and the image.

PARAMETER

[I] `scale` `BSCALE` value to be set
[I] `prec` the number of digit precision
([I] : input, [O] : output)

RETURN VALUE

This API returns the reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate the internal buffer (for example, image().assign_bscale() cannot reallocate an address table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code checks an existence of BSCALE and sets a value if it is not defined.

```
if ( fits.image("Primary").bscale_is_set() == false ) {
    fits.image("Primary").assign_bscale(1.0);
}
```

13.6.16 image().blank(), image().assign_blank()

NAME

image().blank() — Manipulation of blank value

SYNOPSIS

```
long long image( ... ).blank() const;
bool image( ... ).blank_is_set() const;
fits_image &image( ... ).assign_blank( long long blank );
fits_image &image( ... ).erase_blank();
```

DESCRIPTION

`image().blank()` returns a value of `BLANK` in the header. `image().blank_is_set()` returns whether `BLANK` exists or not in a header.

`image().assign_blank()` assigns a value of `BLANK` in a header. `image().erase_blank()` erases a value of `BLANK` in a header.

PARAMETER

[I] `blank` a `BLANK` value to be set

([I] : input, [O] : output)

RETURN VALUE

This API returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, image().assign_blank() cannot reallocate an address table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code checks an existence of BLANK and sets a value if it is not defined.

```
if ( fits.image("Primary").blank_is_set() == false ) {
    fits.image("Primary").assign_blank(0);
}
```

13.6.17 image().bunit(), image().assign_bunit()

NAME

image().bunit(), image().assign_bunit() — manipulation of a physical unit

SYNOPSIS

```
const char *image( ... ).bunit();
bool image( ... ).bunit_is_set();
fits_image &image( ... ).assign_bunit( const char *unit );
fits_image &image( ... ).erase_bunit();
```

DESCRIPTION

`image().bunit()` returns the value of `BUNIT`(the string which represents a physical unit) in a header. The value is the address of a buffer in the object, and it is invalid when the object is revoked or the value of `BUNIT` is changed.

`image().bunit_is_set()` returns whether `BUNIT` exists or not in a header.

`image().assign_bunit()` sets a value of `BUNIT`(a physical unit) in a header. `image().erase_bunit()` deletes a value of `BUNIT` in a header.

PARAMETER

[I] `unit` a pointer to a string of `BUNIT`
([I] : input, [O] : output)

RETURN VALUE

This API returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, image().assign_bunit() cannot reallocate an address table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code checks an existence of BUNIT and sets a value if it is not defined.

```
if ( fits.image("Primary").bunit_is_set() == false ) {
    fits.image("Primary").assign_bunit("ADU");
}
```

13.6.18 image().init()

NAME

image().init() — Initialization of an image and a header

SYNOPSIS

```
fits_image &image( ... ).init();
fits_image &image( ... ).init( const fits_image &obj );
fits_image &image( ... ).init( int type,
                               long naxis0, long naxis1 = 0, long naxis2 = 0 );
fits_image &image( ... ).init( int type, long naxis, long naxisx[] );
```

DESCRIPTION

`image().init()` initializes an image and a header. Wjen `obj` is specified, it is copied to the new HDU object.

An image data_type which can be specified as `type` is any one of "`FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, or `FITS::BYTE_T`".

`naxis0`, `naxis1`, and `naxis2` are the size of the image and the number of layers.

Neither `EXTNAME` nor `EXTVER` is changed.

PARAMETER

[I]	`obj`	an Image object copied to new one in initialization
[I]	`type`	a data type of an image
[I]	`naxis0`	the number of pixels in a column(x-axis)
[I]	`naxis1`	the number of pixels in a row(y-axis)
[I]	`naxis2`	the number of pixels in a layer(z-axis)
[I]	`naxis`	the number of lists specified by an argument `naxisx`
[I]	`naxisx`	a list of the number of pixel in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().init()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, a size of initialized image data exceeds memory space available), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code initializes an Primary HDU with double data type, 100 pixels in x-axis, 200 pixels in y-axis, and 3 pixels in z-axis.

```
fits.image("Primary").init(FITS::DOUBLE_T, 100, 200, 3);
```

13.6.19 image().swap()

NAME

image().swap() — Image swap

SYNOPSIS

```
fits_image &image( ... ).swap( fits_image &obj );
```

DESCRIPTION

`image().swap()` swaps an image object for one which specified as `obj`.

PARAMETER

[I/O]	`obj`	an image object for swap

([I] : input, [O] : output)

RETURN VALUE

`image().swap()` returns a reference to the modified fits image object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits1.image("Primary").swap( fits2.image("Primary") );
```

See also a sample code in §5.10.

13.6.20 image().increase_dim()

NAME

image().increase_dim() — An axis increment

SYNOPSIS

```
fits_image &image( ... ).increase_dim();
fits_image &image( ... ).increase_axis();
```

DESCRIPTION

`image().increase_dim()` increments axes by one. The initial number of pixels of an added axis is 1. This number can be resized by `image().resize()`.

RETURN VALUE

`image().increase_dim()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.image("Primary").increase_dim();
```

13.6.21 image().decrease_dim()

NAME

image().decrease_dim() — An axis decrement

SYNOPSIS

```
fits_image &image( ... ).decrease_dim();
fits_image &image( ... ).decrease_axis();
```

DESCRIPTION

When the number of pixels in a last dimension axis is not 1, it decreases a size of internal buffer simultaneously.

RETURN VALUE

`image().decrease_dim()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.image("Primary").decrease_dim();
```

13.6.22 image().resize()

NAME

image().resize() — Modification of the number of pixels

SYNOPSIS

```
fits_image &image( ... ).resize( long axis, long size );
```

DESCRIPTION

`image().resize()` changes the number of pixels in `axis` to `size`. When the number of pixels is increased, the additional pixels are initialized by a zero value which is reflected by the values of `BZERO` and `BSCALE` in the header.

PARAMETER

[I] `axis` an axis for modification
[I] `size` the number of pixels to be set
([I] : input, [O] : output)

RETURN VALUE

`image().resize()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
printf("before: the number of pixels=%ld\n",
        fits.image("Primary").col_length());
fits.image("Primary").resize(0, 2000);
printf("after: the number of pixels=%ld\n",
        fits.image("Primary").col_length());
```

13.6.23 image().assign_default()

NAME

image().assign_default() — Specify a value to be set for new pixels when resizing the image

SYNOPSIS

```
fits_image &image( ... ).assign_default( double value );
fits_image &image( ... ).assign_default_value( const void *value_ptr );
```

DESCRIPTION

Using these member functions, programmers can specify a value to be set for new pixels created by `.resize()`, etc.

`.assign_default()` is a high-level API, and the value of argument is converted into the appropriate value which is reflected by the values of `BZERO`, `BSCALE` and `BLANK` in the header. Set `NAN` to the argument to specify BLANK value.

`.assign_default_value()` is a low-level API, and values of `BZERO`, etc. are not referred. Programmers should set an address of a value whose type is that of image of current object.

PARAMETER

[I] `value` a value to be set for new pixels
[I] `value_ptr` an address of a value to be set for new pixels
([I] : input, [O] : output)

RETURN VALUE

These member functions returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SLLIB(`sli::err_rec` exception).

EXAMPLES

Next code specifies BLANK value to be set for new pixels, and resizes the width of the image.

```
fits.image("Primary").assign_default(NAN).resize(0, 2000);
```

13.6.24 image().fix_rect_args()

NAME

image().fix_rect_args() — Text and fix values of arguments expressing a rectangular area

SYNOPSIS

```
int image( ... ).fix_rect_args( long *r_col_index, long *r_col_size,
                                long *r_row_index, long *r_row_size,
                                long *r_layer_index, long *r_layer_size ) const;
```

DESCRIPTION

`image().fix_rect_args()` tests whether the values of arguments indicate a rectangular area inside of the image of the object. If the values indicate areas outside of the image, `image().fix_rect_args()` fix the values.

Before calling `.scan_cols()`, etc., exact position and width can be obtained by using this member function. When programmer's work buffer has to be allocated, exact buffer length can be calculated. See also §13.6.25, etc.

PARAMETER

[I/O]	`r_col_index`	pointer of a pixel index of a column(x-axis)
[I/O]	`r_col_size`	pointer of an offset pixels from `*r_col_index`
[I/O]	`r_row_index`	pointer of a pixel index of a row(y-axis)
[I/O]	`r_row_size`	pointer of an offset pixels from `*r_row_index`
[I/O]	`r_layer_index`	pointer of a pixel index of a layer(z-axis)
[I/O]	`r_layer_size`	pointer of an offset pixels from `*r_layer_index`

([I] : input, [O] : output)

RETURN VALUE

0	:	values of arguments indicate an area inside of the image of the object.
positive	:	values of arguments indicate both areas inside/outside of the image of the object.
negative	:	values of arguments indicate an invalid area.

13.6.25 image().scan_cols()

NAME

image().scan_cols() — Horizontally scans the specified area using a user-defined function

SYNOPSIS

```
long image( ... ).scan_cols(
        long (*func)(double [],long, long,long,long,const fits_image *,void *),
                    void *user_ptr,
                    long col_index = 0, long col_size = FITS::ALL,
                    long row_index = 0, long row_size = FITS::ALL,
                    long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

`image().scan_cols()` scans the rectangular area horizontally specified by third or later arguments with following steps:

1. Allocate a temporary buffer of double type whose length is enough for one line of specified rectangular area.
2. Pixel values in a line are converted into that of double type, and are stored into temporary buffer. In this conversion, SCALE, ZERO and BLANK are also applied.
3. User's function `func` is called.
4. Repeat 2. and 3. for all lines in the rectangular area.

The user's function has to have a loop to scan all pixel data in the temporary buffer, and should return the number of valid pixels.

Specifications of arguments in the user-defined function `(*func)` are as below:
`double pix[]` ... temporary buffer having pixel values of a line
`long n_pix` ... number of `pix`
`long axis0` ... a coordinate of x-axis
`long axis1` ... a coordinate of y-axis
`long axis2` ... a coordinate of z-axis
`fits_image *thisp` ... a pointer to this object
`void *ptr` ... a pointer to `user_ptr`

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are scanned.

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`func`	a pointer to a user-defined function
[I]	`user_ptr`	an arbitrary pointer which is given to a user-defined function
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

nonnegative	:	total number of valid pixels (returned by user's function)
negative	:	error (e.g. invalid arguments are specified)

EXAMPLES

The following code scans the image of Primary HDU, and obtains the sum of all pixel values in `res.sum` and the number of valid pixels in `res.npix`.

```
struct pixels_results {
    double sum;
    long npix;
};
```

```
static long pixels_sum( double vals[],long n, long ii,long jj,long kk,
                        const fits_image *this_p, void *_p )
{
    struct pixels_results *resp = (struct pixels_results *)_p;
    long i, cnt = 0;
    for ( i=0 ; i < n ; i++ ) {
        if ( isfinite(vals[i]) ) {
            resp->sum += vals[i];
            cnt ++;
        }
    }
    return cnt;
}

int main()
{
    struct pixels_results res;
    :
    res.npix = fits.image("Primary").scan_cols( &pixels_sum, (void *)&res );
```

13.6.26 image().scan_rows()

NAME

image().scan_rows() — Vertically scans the specified area using a user-defined function

SYNOPSIS

```
long image( ... ).scan_rows(
        long (*func)(double [],long, long,long,long,const fits_image *,void *),
                     void *user_ptr,
                     long col_index = 0, long col_size = FITS::ALL,
                     long row_index = 0, long row_size = FITS::ALL,
                     long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

`image().scan_rows()` scans the rectangular area vertically specified by third or later arguments with following steps:

1. Allocate a temporary buffer of double type whose length is enough for one column of pixels in specified rectangular area.
2. Pixel values in a column are converted into that of double type, and are stored into temporary buffer. In this conversion, SCALE, ZERO and BLANK are also applied.
3. User's function `func` is called.
4. Repeat 2. and 3. for all columns in the rectangular area.

The user's function has to have a loop to scan all pixel data in the temporary buffer, and should return the number of valid pixels.

Specifications of arguments in the user-defined function `(*func)` are as below:
`double pix[]` ... temporary buffer having pixel values of a column
`long n_pix` ... number of `pix`

`long axis0` ... a coordinate of x-axis
`long axis1` ... a coordinate of y-axis
`long axis2` ... a coordinate of z-axis
`fits_image *thisp` ... a pointer to this object
`void *ptr` ... a pointer to `user_ptr`

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are scanned.

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`func`	a pointer to a user-defined function
[I]	`user_ptr`	an arbitrary pointer which is given to a user-defined function
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

nonnegative	:	total number of valid pixels (returned by user's function)
negative	:	error (e.g. invalid arguments are specified)

EXAMPLES

See EXAMPLES in §13.6.25.

13.6.27 image().scan_layers()

NAME

image().scan_layers() — Scan the specified rectangular area along z-axis using a user-defined function

SYNOPSIS

```
long image( ... ).scan_layers(
   long (*func)(double [],long,long, long,long,long,const fits_image *,void *),
                    void *user_ptr,
                    long col_index = 0, long col_size = FITS::ALL,
                    long row_index = 0, long row_size = FITS::ALL,
                    long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

`image().scan_layers()` scans the rectangular area along z-axis specified by third or later arguments with following steps:

1. Allocate a temporary buffer of double type whose length is enough for z-length $\times$ x-length of pixels in specified rectangular area.
2. Pixel values are converted into that of double type, and are stored into temporary buffer. In this conversion, SCALE, ZERO and BLANK are also applied.

3. User's function `func` is called.
4. Repeat 2. and 3. for all rows in the rectangular area.

The user's function has to have two loops (inner loop should be for scanning z) to scan all pixel data in the temporary buffer, and should return the number of valid pixels.

Specifications of arguments in the user-defined function `(*func)` are as below:
`double pix[]` ... temporary buffer having pixel values of z-axis $\times$ x-axis
`long n_zpix` ... number of `pix` along z-axis
`long n_xpix` ... number of `pix` along x-axis
`long axis0` ... a coordinate of x-axis
`long axis1` ... a coordinate of y-axis
`long axis2` ... a coordinate of z-axis
`fits_image *thisp` ... a pointer to this object
`void *ptr` ... a pointer to `user_ptr`

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are scanned.

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`func`	a pointer to a user-defined function
[I]	`user_ptr`	an arbitrary pointer which is given to a user-defined function
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

nonnegative	:	total number of valid pixels (returned by user's function)
negative	:	error (e.g. invalid arguments are specified)

EXAMPLES

Next code obtains sum of all pixels along z-axis in each (x, y) of image in the Primary HDU, and stores the result into prepared buffer.

```
static long combine_sum( double pix[], long nz, long nx,
                         long x, long y, long z,
                         const fits_image *this_p, void *_out_buf_p )
{
    double **out_buf_p = (double **)_out_buf_p;
    double *out_buf = *out_buf_p;
    double v;
    long i,j, nz_valid, nx_valid = 0, ix = 0;
    for ( i=0 ; i < nx ; i++ ) {                /* loop for X */
        v = 0;
        nz_valid = 0;
```

```
            for ( j=0 ; j < nz ; j++ ) {            /* loop for Z */
                if ( isfinite(pix[ix]) ) {          /* if valid */
                    v += pix[ix];
                    nz_valid ++;
                }
                ix ++;                              /* count always */
            }
            if ( 0 < nz_valid ) {
                out_buf[i] = v;                     /* save the result */
                nx_valid ++;
            }
            else out_buf[i] = NAN;
        }
        *out_buf_p = out_buf + nx;                  /* set next position */
        return nx_valid;
    }

    int main()
    {
        double *buf;
        :
        (allocate memory on buf)
        :
        double *t_buf = buf;
        res.npix = fits.image("Primary").scan_layers( &combine_sum,
                                                      (void *)(&t_buf) );
```

13.6.28 image().fill()

NAME

image().fill() — Modification of pixel values in a rectangular area to a value

SYNOPSIS

```
fits_image &image( ... ).fill( double value,
                               long col_index = 0, long col_size = FITS::ALL,
                               long row_index = 0, long row_size = FITS::ALL,
                               long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().fill()` modifies pixel values in a rectangular area whose location is specified by second or later arguments into `value`.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified, respectively.

Do not use a constant `FITS:ALL` explicitly.

PARAMETER

[I]	`value`	a pixel value to be set
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().fill()` returns a reference to the modified fits_image object.

EXAMPLES

The following code sets 1 to all pixel values.

```
fits.image("Primary").fill(1);
```

13.6.29 image().add()

NAME

image().add() — Addition of pixel values in a rectangular area by the value

SYNOPSIS

```
fits_image &image( ... ).add( double value,
                              long col_index = 0, long col_size = FITS::ALL,
                              long row_index = 0, long row_size = FITS::ALL,
                              long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().add()` adds `value` to pixel values in a rectangular area whose location is specified by second or later arguments.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified, respectively.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`value`	a value to be added to pixel values
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().add()` returns a reference to the modified fits_image object.

EXAMPLES The following code adds 1 to all pixel values.

```
fits.image("Primary").add(1);
```

13.6.30 image().subtract()

NAME

image().subtract() — Subtracting a value from a rectangular area

SYNOPSIS

```
fits_image &image( ... ).subtract( double value,
                                   long col_index = 0, long col_size = FITS::ALL,
                                   long row_index = 0, long row_size = FITS::ALL,
                                   long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().subtract()` subtracts `value` from pixel values in a rectangular area whose location is specified by second or later arguments.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified, respectively.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`value`	a value which are subtracted from pixel values
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().subtract()` returns a reference to the modified fits_image object.

EXAMPLES The following code subtracts 1.0 from all pixel values.

```
fits.image("Primary").subtract(1.0);
```

13.6.31 image().multiply()

NAME

image().multiply() — Multiplication of pixel values in the rectangular area by the value

SYNOPSIS

```
fits_image &image( ... ).multiply( double value,
                                   long col_index = 0, long col_size = FITS::ALL,
                                   long row_index = 0, long row_size = FITS::ALL,
                                   long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().multiply()` multiplies pixel values in a rectangular area whose location is specified by second or later arguments by `value`.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified, respectively.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`value`	a value to be multiplied by pixel values
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	layer_size an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().multiply()` returns a reference to the modified fits_image object.

EXAMPLES

The following code multiplies all pixel values by 2.

```
fits.image("Primary").multiply(2);
```

13.6.32 image().divide()

NAME

image().divide() — Dividing pixel values in the rectangular area by a value

SYNOPSIS

```
fits_image &image( ... ).divide( double value,
                                 long col_index = 0, long col_size = FITS::ALL,
                                 long row_index = 0, long row_size = FITS::ALL,
                                 long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().divide()` divides pixels of a rectangular area whose location is specified by second or later arguments by `value`.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified, respectively.

Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`value`	a value which divides pixel values
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	layer_size an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().divide()` returns a reference to the modified fits_image object.

EXAMPLES

The following code divides all pixels by 3.0.

```
fits.image("Primary").divide(3.0);
```

13.6.33 image().fill()

NAME

image().fill() — Modification of pixel values in a rectangular area by a user-defined function

SYNOPSIS

```
fits_image &image( ... ).fill( double value,
       void (*func)(double [],double,long, long,long,long,fits_image *,void *),
       void *user_ptr,
       long col_index = 0, long col_size = FITS::ALL,
       long row_index = 0, long row_size = FITS::ALL,
       long layer_index = 0, long layer_size = FITS::ALL );
```

DESCRIPTION

`image().fill()` modifies pixel values in a rectangular area whose location is specified by fourth or later arguments by a user-defined function `(*func)`.

This member function prepares a temporary buffer to input/output data in the user-defined function, and performs line-by-line scans in specified rectangular area. For each line scan, 1. pixel values in this object are converted and stored into the temporary buffer, 2. the user-defined function is called, and 3. data in the temporary buffer are converted and outputted into pixels in this object. In the conversion of step 1 and 3, SCALE, ZERO and BLANK are applied. The user's function has to have a loop to scan all pixel data in the temporary buffer.

Specifications of arguments in the user-defined function `(*func)` are as below, and the user-defined function has to store the results into `pix[]`:

`double pix[]` ... original pixel values in temporary buffer
`double value` ... a first argument of image().fill()
`long n_pix` ... number of `pix`
`long axis0` ... a coordinate of x-axis
`long axis1` ... a coordinate of y-axis
`long axis2` ... a coordinate of z-axis
`fits_image *thisp` ... a pointer to this object
`void *ptr` ... a pointer to `user_ptr`

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified. Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`value`	a pixel value to be set via a user-defined function
[I]	`func`	a pointer to a user-defined function
[I]	`user_ptr`	an arbitrary pointer which is given to a user-defined function
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`image().fill()` returns a reference to the modified fits_image object.

EXAMPLES

The following code sets pixel values over a threshold value `thresh` to constant values.

```
static void myfunc( double pix[], double value, long n_pix,
                    long axis0, long axis1, long axis2,
                    fits_image *thisp, void *ptr )
{
    long i;
    for ( i=0 ; i < n_pix ; i++ ) {
        if ( value < pix[i] ) pix[i] = value;
    }
    return;
}
    :
    :
    fits_image &primary = fits.image("Primary");
    primary.fill( thresh, &myfunc, NULL,
                  0, primary.col_length(), 0, primary.row_length() );
```

13.6.34 image().copy()

NAME

image().copy() — Copy and cut of pixel values in a rectangular area

SYNOPSIS

```
void image( ... ).copy( fits_image *dest_img ) const;
void image( ... ).copy( fits_image *dest_img,
                        long col_index, long col_size = FITS::ALL,
                        long row_index = 0, long row_size = FITS::ALL,
                        long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

This API copies pixel values in rectangular areas, which are in layers specified `layer_index` and `layer_size` and are defined by indices(`col_index`, `row_index`), and offsets(`col_size`, `row_size`), to an object pointed by `dest_img`.

When each argument of a column, a row, and a layer is not specified, all columns, rows, and layers are copied.

PARAMETER

[O]	`dest_img`	an object which specified rectangular areas are stored to
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

`cut()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code copies 128 × 128 square area to `my_image_buffer`, and paste it to the original image at (128, 0).

```
fits_image my_image_buffer;
fits.image("Primary").copy(&my_image_buffer, 0,128, 0,128);
fits.image("Primary").paste(my_image_buffer, 128, 0);
```

See also a sample code in §5.10.

13.6.35 image().paste()

NAME

image().paste() — Paste of images in a copy buffer

SYNOPSIS

```
fits_image &image( ... ).paste( const fits_image &src_img,
                                long col_index = 0, long row_index = 0,
                                long layer_index = 0 );
```

DESCRIPTION

`image().paste()` pastes an objects pointed by `src_img` into an area which is specified by `col_index`, `row_index`, and `layer_index`.

This API converts data types appropriately when the one of this object and the one of `src_img` are not matched, and also when `BZERO`, `BSCALE`, and `BLANK` in headers of the objects are not matched.

PARAMETER

[I]	`src_img`	a source object for copy
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`layer_index`	a pixel index of a layer(z-axis)

([I] : input, [O] : output)

RETURN VALUE

`image().paste()` returns a reference to the modified fits_image object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.6.34 and also a sample code in §5.10.

13.6.36 image().add()

NAME

image().add() — Adding an image in a copy buffer to an original image

SYNOPSIS

```
fits_image &image( ... ).add( const fits_image &src_img,
                              long col_index = 0, long row_index = 0,
                              long layer_index = 0 );
```

DESCRIPTION

`image().add()` adds an image in an object pointed by `src_img` to an original image with coordinates(`col_index`, `row_index`) in a layer specified by `layer_index`. When the pixel of `src_img` is NAN, pixels are not modified.

This API converts data types appropriately when the one of the objects and the one of `src_img` are not matched, and also when `BZERO`, `BSCALE`, and `BLANK` in headers of the objects are not matched.

PARAMETER

[I]	`src_img`	an object to be added
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`layer_index`	a pixel index of a layer(z-axis)

([I] : input, [O] : output)

RETURN VALUE

`image().add()` returns a reference to the modified fits_image object.

EXAMPLES

An image in a copy buffer is overlapped with an original image by rewriting `add` from `paste()` of EXAMPLES in §13.6.34 as below:

```
fits.image("Primary").add(my_image_buffer, 128, 0);
```

13.6.37 image().subtract()

NAME

image().subtract() — Subtracting an image in a copy buffer from an original image

SYNOPSIS

```
fits_image &image( ... ).subtract( const fits_image &src_img,
                                   long col_index = 0, long row_index = 0,
                                   long layer_index = 0 );
```

DESCRIPTION

`image().subtract()` subtracts an image in an object pointed by `src_img` from an original image with coordinates(`col_index`, `row_index`) in a layer specified by `layer_index`. When a pixel of `src_img` is NAN, pixels are not modified.

This API converts data types appropriately when the one of this object and the one of `src_img` are not matched, and also when `BZERO`, `BSCALE`, and `BLANK` in headers of the objects are not matched.

PARAMETER

[I] `src_img` an object which are subtracted from an original image
[I] `col_index` a pixel index of a column(x-axis)
[I] `row_index` a pixel index of a row(y-axis)
[I] `layer_index` a pixel index of a layer(z-axis)
([I] : input, [O] : output)

RETURN VALUE

`image().subtract()` returns a reference to the modified fits_image object.

EXAMPLES

See EXAMPLES in §13.6.36.

13.6.38 image().multiply()

NAME

image().multiply() — Multiplying an original image by an image in a copy buffer

SYNOPSIS

```
fits_image &image( ... ).multiply( const fits_image &src_img,
                                   long col_index = 0, long row_index = 0,
                                   long layer_index = 0 );
```

DESCRIPTION

`image().multiply()` multiplies an original image with coordinates(`col_index`, `row_index`) in a layer specified by `layer_index` by an image in an object pointed by `src_img`. When the pixel of `src_img` is NAN, pixels are not modified.

This API converts data types appropriately when that of the object and that of `src_img` do not match, and also when `BZERO`, `BSCALE`, and `BLANK` in headers of the objects do not match.

PARAMETER

[I] `src_img` an object which are multiplied to an original image
[I] `col_index` a pixel index of a column(x-axis)
[I] `row_index` a pixel index of a row(y-axis)
[I] `layer_index` a pixel index of a layer(z-axis)
([I] : input, [O] : output)

RETURN VALUE

`image().multiply()` returns a reference to the modified fits_image object.

EXAMPLES

See EXAMPLES in §13.6.36.

13.6.39 image().divide()

NAME

image().divide() — Dividing an original image by an image in a copy buffer

SYNOPSIS

```
fits_image &image( ... ).divide( const fits_image &src_img,
                                 long col_index = 0, long row_index = 0,
                                 long layer_index = 0 );
```

DESCRIPTION

`image().divide()` divides an original image with coordinates(`col_index`, `row_index`) in a layer specified by `layer_index` by an image in an object pointed by `src_img`. When the pixel of `src_img` is NAN, pixels are not modified.

This API converts data types appropriately when that of the object and that of `src_img` do not match, and also when `BZERO`, `BSCALE`, and `BLANK` in headers of the objects do not match.

PARAMETER

[I]	`src_img`	an object which divides an original image
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`layer_index`	a pixel index of a layer(z-axis)

([I] : input, [O] : output)

RETURN VALUE

`image().divide()` returns a reference to the modified fits_image object.

EXAMPLES

See EXAMPLES in §13.6.36.

13.6.40 image().paste()

NAME

image().paste() — Paste of images in a copy buffer via a user-defined function

SYNOPSIS

```
fits_image &image( ... ).paste( const fits_image &src_img,
    void (*func)(double [],double [],long, long,long,long,fits_image *,void *),
    void *user_ptr,
    long dest_col = 0, long dest_row = 0, long dest_layer = 0 );
```

DESCRIPTION

`image().paste()` pastes an image in an objects pointed by `src_img` into a rectangular area which is specified by `dest_col`, `dest_row`, and `dest_layer`. At this time, pixel values can be modified by a user-defined function `(*func)`.

This member function prepares a temporary buffer to input/output data in the user-defined function, and performs line-by-line scans in specified rectangular area. For each line scan, 1. pixel values in this object are converted and stored into the temporary buffer, 2. the user-defined function is called, and 3. data in the temporary buffer are converted and outputted into pixels in this object. In the conversion of step 1 and 3, SCALE, ZERO and BLANK are applied. The user's function has to have a loop to scan all pixel data in the temporary buffer.

Specifications of arguments in the user-defined function `(*func)` are as below, and the user-defined function has to store the results into `pix_self[]`:

`double pix_self[]` ... original pixel values
`double pix_src[]` ... pixel values of `src_img`
`long n_pix` ... number of `pix_self` or `pix_src`
`long axis0` ... a coordinate of x-axis
`long axis1` ... a coordinate of y-axis
`long axis2` ... a coordinate of z-axis

`fits_image *thisp` ... a pointer to this object
`void *ptr` ... a pointer to `user_ptr`

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified. Do not use a constant `FITS::ALL` explicitly.

PARAMETER

[I]	`src_img`	an source object to be set
[I]	`func`	a pointer to a user-defined function
[I]	`user_ptr`	an arbitrary pointer which is given to a user-defined function
[I]	`dest_col`	a pixel index of a column(x-axis) in a destination
[I]	`dest_row`	a pixel index of a row(y-axis) in a destination
[I]	`dest_layer`	a pixel index of a layer(z-axis) in a destination

([I] : input, [O] : output)

RETURN VALUE

`image().paste()` returns a reference to the modified fits_image object.

EXAMPLES

For more information about a user-defined function, see EXAMPLES in §13.6.33.

13.6.41 image().stat_pixels()

NAME

image().stat_pixels() — Compute pixel statistics of a rectangular area

SYNOPSIS

```
long image( ... ).stat_pixels( double results[], size_t results_len,
                      const char *options,
                      long col_index = 0, long col_size = FITS::ALL,
                      long row_index = 0, long row_size = FITS::ALL,
                      long layer_index = 0, long layer_size = FITS::ALL ) const;
long image( ... ).stat_pixels( fits_header *results, const char *options,
                      long col_index = 0, long col_size = FITS::ALL,
                      long row_index = 0, long row_size = FITS::ALL,
                      long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

These member functions compute some statistics of pixels in a rectangular area specified by `col_index` or later arguments. The results are stored into an array or a fits_header object pointed by `results`.

To select required statistics, programmers set a string such as `"results=mean,stddev,median"` to argument `options`. Following strings can be set after `results=`:

`npix` ... the number of pixels used to do the statistics
`mean` ... the mean of the pixel distribution
`stddev` ... the standard deviation of the pixel distribution
`median` ... the median of the pixel distribution[39)]

[39)] This is not an approximate value such as 'midpt' of IRAF but true median. SFITSIO calculates it with high-speed.

`min` ... the minimum pixel value
`max` ... the maximum pixel value
`skew` ... the skew of the pixel distribution
`kurtosis` ... the kurtosis of the pixel distribution

There are two types of `results` arguments; one is a simple array in which statistics are stored in order of elements in `results=` string, another is a fits_header object in which keywords (e.g., `"MEAN"`) and statistics are stored.

PARAMETER

[O]	`results`	an address of an array or a fits_header object in which the statistics are stored
[I]	`results_len`	length of array `results`
[I]	`options`	an option string for computing statistics
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : input, [O] : output)

RETURN VALUE

nonnegative	:	total number of valid pixels
negative	:	error (e.g. invalid arguments are specified)

EXAMPLES

Next code computes statistics of an image in Primary HDU, and display them. Programmers can use an object of fits_header class to obtain the results of statistics.

```
fits_header stat_results;
const char *stat_options = "results=npix,mean,stddev,min,max,median";
long i;
/* Compute statistics */
if ( fits.image("Primary").stat_pixels(&stat_results, stat_options) < 0 ) {
    /* error handling */
}
/* Display results */
for ( i=0 ; i < stat_results.length() ; i++ ) {
    printf("%s = %.15g\n", stat_results.at(i).keyword(),
           stat_results.at(i).dvalue());
}
```

To get a value from `stat_results` object, programmers can handle it like associative array such as `stat_results.at("NPIX").dvalue()`.

`image().stat_pixels()` is used in the example code in exercise 3 in §1.1.
See also `tools/stat_pixels.cc` in SFITSIO source package.

13.6.42 image().combine_layers()

NAME

image().combine_layers() — Combine 2-d images in specified cubic area

SYNOPSIS

```
long image( ... ).combine_layers( fits_image *dest_img, const char *options,
                      long col_index = 0, long col_size = FITS::ALL,
                      long row_index = 0, long row_size = FITS::ALL,
                      long layer_index = 0, long layer_size = FITS::ALL ) const;
```

DESCRIPTION

`image().combine_layers()` combines 2-d images in the cubic area of image buffer specified by `col_index` or later arguments. The results are stored into a fits_image object pointed by `dest_img`.

Programmers can select method of combine and type of the output image. For example, to perform combine with average and to output the result with 4-byte floating point, set `"combine=average outtype=float"` to `options`.

Programmers can set a string after '`combine=`' to select method of combine: `average`, `median`, `sum`, `min`, or `max`.

Programmers can set a string after '`outtype=`' to select type of output image:
`short` ... 2-byte signed integer
`ushort` ... 2-byte unsigned integer (BZERO=32768.0)
`long` ... 4-byte signed integer
`longlong` ... 8-byte signed integer
`float` ... 4-byte floating point
`double` ... 8-byte floating point

PARAMETER

[O]	`dest_img`	an address of a fits_image object in which the combined image is stored
[I]	`options`	an option string for combine
[I]	`col_index`	a pixel index of a column(x-axis)
[I]	`col_size`	an offset pixels from `col_index`
[I]	`row_index`	a pixel index of a row(y-axis)
[I]	`row_size`	an offset pixels from `row_index`
[I]	`layer_index`	a pixel index of a layer(z-axis)
[I]	`layer_size`	an offset pixels from `layer_index`

([I] : 入力, [O] : 出力)

RETURN VALUE

nonnegative	:	total number of valid pixels
negative	:	error (e.g. invalid arguments are specified)

EXAMPLES

Next code is an example to combine three 2-d images using median. Contents of first FITS file is loaded into '`fits0`', second and third images are inserted into second and third layers of the image buffer in '`fits0`', and combine of three 2-d images are performed. In this example, codes for error handling are omitted for readability.

```
fitscc fits0, fits1;
/* read and prepare */
fits0.read_stream("raw_image_0.fits");      /* 2d image No.1 */
fits0.image("Primary").resize(2, 3);        /* resize z-axis => 3 */
fits1.read_stream("raw_image_1.fits");      /* 2d image No.2 */
fits0.image("Primary").paste(fits1.image("Primary"), 0L, 0L, 1L);
fits1.read_stream("raw_image_2.fits");      /* 2d image No.3 */
```

```
fits0.image("Primary").paste(fits1.image("Primary"), 0L, 0L, 2L);
/* combine (fits1.image("Primary") will be overwritten) */
fits0.image("Primary").combine_layers(&fits1.image("Primary"),
                                      "combine=median outtype=float");
/* output combined image */
fits1.write_stream("combined_image.fits");
```

See also `tools/combine_images.cc` in SFITSIO source package.

13.7 APIs for low-level manipulation of an Image HDU

In this section, we describe APIs to manipulate an Image HDU at low-level. At low-level APIs, users have to do a data conversion, using `BZERO` and `BSCALE`. Usually, the APIs shown here are not necessary for the manipulation of an Image HDUs, however they may help to speed up the execution.

Every argument of "`image( ... )`" of all APIs in this section is an HDU index(`long index`) or an HDU name(`const char *hduname`), therefore we skip the description of this argument.

13.7.1 image().data_array()

NAME

image().data_array() — A reference to an object for managing a data buffer

SYNOPSIS

```
sli::mdarray &image( ... ).data_array();
const sli::mdarray &image( ... ).data_array() const;
const sli::mdarray &image( ... ).data_array_cs() const;
```

DESCRIPTION

An image buffer of the fits_image class is managed by the mdarray class of SLLIB. `data_array()` is used when users conduct an opeartion with the mdarray class.

For more information on the mdarray class, see a SLLIB manual.

13.7.2 image().data_ptr()

NAME

image().data_ptr() — An address of a data buffer in an object

SYNOPSIS

```
void *image( ... ).data_ptr( long axis0 = 0,
                             long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
void *image( ... ).data_ptr_v( long num_axisx,
                               long axis0, long axis1, long axis2, ... );
void *image( ... ).va_data_ptr_v( long num_axisx,
                                  long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().data_ptr()` returns an address of an internal image data buffer at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. When the coordinate argument is out of range, NULL is returned.

`axis0`, `axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

The returned value is an address of an internal buffer in an object, and it is invalid when the object is revoked or the type or the size of the buffer is changed.

The returned address is cast into any one of `fits::double_t *`, `fits::float_t *`, `fits::longlong_t *`, `fits::long_t *`, `fits::short_t *`, `fits::byte_t *` depending on a current FITS data type.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

integer value	:	an address of internal image data buffer
NULL	:	error(e.g. A coordinate argument is out of range.)

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits::double_t *img_ptr = (fits::double_t *)fits.image("Primary").data_ptr();
```

See also a sample code in §5.11.

13.7.3 image().get_data()

NAME

image().get_data() — Get a raw image data

SYNOPSIS

```
ssize_t image( ... ).get_data( void *dest_buf, size_t buf_size,
                               long axis0 = 0, long axis1 = FITS::INDEF,
                               long axis2 = FITS::INDEF ) const;
ssize_t image( ... ).get_data_v( void *dest_buf, size_t buf_size,
                                 long num_axisx,
                                 long axis0, long axis1, long axis2, ... ) const;
ssize_t image( ... ).va_get_data_v( void *dest_buf, size_t buf_size,
                                    long num_axisx,
                                    long axis0, long axis1, long axis2,
                                    va_list ap ) const;
```

DESCRIPTION

`image().get_data()` copies raw image data at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer to `dest_buf` up to `buf_size` bytes.

`axis0`, `axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

The address specified by `dest_buf` is cast into any one of `fits::double_t *`, `fits::float_t *`, `fits::longlong_t *`, `fits::long_t *`, `fits::short_t *`, `fits::byte_t *` depending on a current FITS data type.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[O]	`dest_buf`	an address of an obtained image data
[I]	`buf_size`	a size of `dest_buf` in byes
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

nonnegative integer	:	the number of copied bytes with a sufficient buffer size
negative	:	error(e.g. Invalid arguments are specified and the copy process is cancelled.)

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
size_t buf_size = fits.image("Primary").bytes() * fits.image("Primary").length();
fits::double_t *dest_buf = (fits::double_t *)malloc(buf_size);
if ( dest_buf == NULL ) {
  /* error handling */
}
fits.image("Primary").get_data(dest_buf, buf_size);
```

13.7.4 image().put_data()

NAME

image().put_data() — Put a raw image data

SYNOPSIS

```
ssize_t image( ... ).put_data( const void *src_buf, size_t buf_size, long axis0 = 0,
                               long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
ssize_t image( ... ).put_data_v( const void *src_buf, size_t buf_size,
                                 long num_axisx,
                                 long axis0, long axis1, long axis2, ... );
ssize_t image( ... ).va_put_data_v( const void *src_buf, size_t buf_size,
                                    long num_axisx,
                                    long axis0, long axis1, long axis2,
                                    va_list ap );
```

DESCRIPTION

`image().put_data()` copies raw image data pointed by `src_buf` to the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer of an object up to `buf_size` bytes.

`axis0`, `axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`src_buf`	an address of a source image data
[I]	`buf_size`	a size of `src_buf` in bytes
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

nonnegative integer	:	the number of copied bytes with a sufficient buffer size
negative	:	error(e.g. Invalid arguments are specified and the copy process is cancelled.)

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
size_t buf_size = fits.image("Primary").bytes() * fits.image("Primary").length();
fits::double_t *src_buf = (fits::double_t *)malloc(buf_size);
               :
               :
fits.image("Primary").put_data(src_buf, buf_size);
```

13.7.5 image().double_value()

NAME

image().double_value() — Return raw image data

SYNOPSIS

```
double image( ... ).double_value( long axis0, long axis1 = FITS::INDEF,
                                  long axis2 = FITS::INDEF ) const;
double image( ... ).double_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, ... ) const;
double image( ... ).va_double_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().double_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().dvalue()`(see §13.6.10.) returns a pixel value which is subject to `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().double_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code displays all pixel values in 0-th row.

```
long idx;
for ( idx=0 ; idx < fits.image("Primary").col_length() ; idx++ ) {
    printf("index[%ld]=[%lf]\n", idx, fits.image("Primary").double_value(idx,0));
}
```

13.7.6 image().float_value()

NAME

image().float_value() — Return raw image data

SYNOPSIS

```
float image( ... ).float_value( long axis0, long axis1 = FITS::INDEF,
                                long axis2 = FITS::INDEF ) const;
float image( ... ).float_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, ... ) const;
float image( ... ).va_float_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().float_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().dvalue()`(see §13.6.10.) when you need the pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().float_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.5.

13.7.7 image().longlong_value()

NAME

image().longlong_value() — Return raw image data

SYNOPSIS

```
long long image( ... ).longlong_value( long axis0, long axis1 = FITS::INDEF,
                                       long axis2 = FITS::INDEF ) const;
long long image( ... ).longlong_value_v( long num_axisx,
                         long axis0, long axis1, long axis2, ... ) const;
long long image( ... ).va_longlong_value_v( long num_axisx,
                         long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().longlong_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().dvalue()`, `image().lvalue()` or `image().llvalue()`(see §13.6.10) when you need the pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().longlong_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.5.

13.7.8 image().long_value()

NAME

image().long_value() — Return raw image data

SYNOPSIS

```
long image( ... ).long_value( long axis0, long axis1 = FITS::INDEF,
                              long axis2 = FITS::INDEF ) const;
long image( ... ).long_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, ... ) const;
long image( ... ).va_long_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().long_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().dvalue()`, `image().lvalue()` or `image().llvalue()`(see §13.6.10) when you need the pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().long_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.5.

13.7.9 image().short_value()

NAME

image().short_value() — Return raw image data

SYNOPSIS

```
short image( ... ).short_value( long axis0, long axis1 = FITS::INDEF,
                                long axis2 = FITS::INDEF ) const;
short image( ... ).short_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, ... ) const;
short image( ... ).va_short_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().short_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().dvalue()`, `image().lvalue()` or `image().llvalue()`(see §13.6.10) when you need the pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().short_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.5.

13.7.10 image().byte_value()

NAME

image().byte_value() — Return raw image data

SYNOPSIS

```
unsigned char image( ... ).byte_value( long axis0, long axis1 = FITS::INDEF,
                                       long axis2 = FITS::INDEF ) const;
unsigned char image( ... ).byte_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, ... ) const;
unsigned char image( ... ).va_byte_value_v( long num_axisx,
                        long axis0, long axis1, long axis2, va_list ap ) const;
```

DESCRIPTION

`image().byte_value()` returns a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().dvalue()`, `image().lvalue()` or `image().llvalue()`(see §13.6.10) when you need the pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().byte_value()` returns a pixel value.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.5.

13.7.11 image().assign_double()

NAME

image().assign_double() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_double( double value, long axis0,
                            long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_double_v( double value, long num_axisx,
                                        long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_double_v( double value, long num_axisx,
                                   long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_double()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. Use `image().assign()`(see §13.6.12.) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_double()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

The following code assigns 0 to all pixel values in 0-th row.

```
long idx;
for ( idx=0 ; idx < fits.image("Primary").col_length() ; idx++ ) {
    fits.image("Primary").assign_double(0.0, idx,0);
}
```

13.7.12 image().assign_float()

NAME

image().assign_float() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_float( float value, long axis0,
                            long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_float_v( float value, long num_axisx,
                                        long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_float_v( float value, long num_axisx,
                                  long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_float()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().assign()`(see §13.6.12) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_float()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.11.

13.7.13 image().assign_longlong()

NAME

image().assign_longlong() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_longlong( long long value, long axis0,
                            long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_longlong_v( long long value, long num_axisx,
                                      long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_longlong_v( long long value, long num_axisx,
                                 long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_longlong()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().assign()`(see §13.6.12) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_longlong()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.11.

13.7.14 image().assign_long()

NAME

image().assign_long() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_long( long value, long axis0,
                            long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
```

```
fits_image &image( ... ).assign_long_v( long value, long num_axisx,
                                        long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_long_v( long value, long num_axisx,
                                   long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_long()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().assign()`(see §13.6.12) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_long()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.11.

13.7.15 image().assign_short()

NAME

image().assign_short() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_short( short value, long axis0,
                           long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_short_v( short value, long num_axisx,
                                         long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_short_v( short value, long num_axisx,
                                   long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_short()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().assign()`(see §13.6.12) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_short()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.11.

13.7.16 image().assign_byte()

NAME

image().assign_byte() — Assign raw image value

SYNOPSIS

```
fits_image &image( ... ).assign_byte( unsigned char value, long axis0,
                            long axis1 = FITS::INDEF, long axis2 = FITS::INDEF );
fits_image &image( ... ).assign_byte_v( unsigned char value, long num_axisx,
                                        long axis0, long axis1, long axis2, ... );
fits_image &image( ... ).va_assign_byte_v( unsigned char value, long num_axisx,
                                  long axis0, long axis1, long axis2, va_list ap );
```

DESCRIPTION

`image().assign_byte()` modifies a raw pixel value at the `axis0`-th column, the `axis1`-th row, and the `axis2`-th layer. `image().assign()`(see §13.6.12) modifies data by a pixel value which is reflected by `BZERO` and `BSCALE`.

`axis1`, `axis2` are optional, and n-dimensional data can be treated as 1-dimensional, 2-dimensional, and 3(or more)-dimensional data by the number of arguments.

Do not use a constant `FITS::INDEF` explicitly.

PARAMETER

[I]	`value`	a value to be set
[I]	`axis0`	a pixel location in a column(x-axis)
[I]	`axis1`	a pixel location in a row(y-axis)
[I]	`axis2`	a pixel location in a layer(z-axis)
[I]	`num_axisx`	the number of axes specified as arguments
[I]	`...`	all location data of pixels in each axis
[I]	`ap`	all location data of pixels in each axis

([I] : input, [O] : output)

RETURN VALUE

`image().assign_byte()` returns a reference to the modified fits_image object.

EXCEPTION

When invalid values are specified as variable arguments, the API throws an exception derived from SFITSIO(`sli::err_rec` exception).

EXAMPLES

See EXAMPLES in §13.7.11.

13.8 Manipulation of Ascii Table HDU and Binary Table HDU

In this section, we describe APIs for manipulating ASCII Table HDU and Binary Table HDU. In SFITSIO, the ASCII Table is treated as an only case of the string column (i.e., `TTYPE`*n* of header is *x*`A`) of Binary Table. Therefore, both the ASCII Table and the Binary Table can be treated via the same APIs.

APIs are classified into two groups. The first case is,

```
value = fits.table( ... ).function( ... );
```

The second case is,

```
value = fits.table( ... ).col( ... ).function( ... );
value = fits.table( ... ).colf( ... ).function( ... );
```

The argument for `table( ... )` is HDU number (`long index`) or HDU name (`const char *hduname`).

The argument for `col()` is column index(`long index`) or column name (`const char *col_name`). The argument for `colf()` is column name written as the format of libc's printf().

For the rest of this document, the argument of `table( ... )`, `col( ... )` and `colf( ... )` is the same as above, so the explanation is omitted.

13.8.1 table().hduname(), table().assign_hduname()

NAME

table().assign_hduname(), table().hduname() — Manipulate HDU Name

SYNOPSIS

```
const char *table( ... ).hduname();
const char *table( ... ).extname();
fits_table &table( ... ).assign_hduname( const char *name );
fits_table &table( ... ).assign_extname( const char *name );
```

DESCRIPTION

`table().hduname()` returns HDU `name`. `table().assign_hduname()` sets HDU `name`. The argument `name` is reflected to `EXTNAME`.

PARAMETER

[I] `name` HDU `name`

([I] : input, [O] : output)

RETURN VALUE

`table().hduname()` returns an address of HDU name string.

`table().assign_hduname()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, to resize the space when changing HDU name), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
printf("HDU Name=\%s\n", fits.table("EVENT").hduname());
```

13.8.2 table().hduver(), table().assign_hduver()

NAME

table().assign_hduver(), table().hduver() — Manipulate HDU version number

SYNOPSIS

```
long long table( ... ).hduver();
long long table( ... ).extver();
fits_table &table( ... ).assign_hduver( long long ver );
fits_table &table( ... ).assign_extver( long long ver );
```

DESCRIPTION

`table().hduver()` returns HDU version number.

`table().assign_hduver()` sets HDU version number. The argument `ver` is reflected to `EXTVER`.

PARAMETER

[I] `ver` version number
([I] : input, [O] : output)

RETURN VALUE

`table().hduver()` returns the HDU version number.

`table().assign_hduver()` returns a reference to the fits_table object.

EXAMPLES

```
printf("HDU Version=%lld\n", fits.table("EVENT").hduver());
```

13.8.3 table().col_length()

NAME

table().col_length() — Number of columns

SYNOPSIS

```
long table( ... ).col_length() const;
```

RETURN VALUE

`table().col_length()` returns the number of columns.

EXAMPLES

```
printf("Column Length=%ld\n", fits.table("EVENT").col_length());
```

13.8.4 table().row_length()

NAME

table().row_length() — Number of rows

SYNOPSIS

```
long table( ... ).row_length() const;
```

RETURN VALUE

`table().row_length()` returns the number of rows.

EXAMPLES

```
printf("Row Length=%ld\n", fits.table("EVENT").row_length());
```

13.8.5 table().heap_length()

NAME

table().heap_length() — Byte length of heap area

SYNOPSIS

```
long table( ... ).heap_length() const;
```

RETURN VALUE

`table().heap_length()` returns byte length of heap area owned by the table.

13.8.6 table().col_index()

NAME

table().col_index() — Column index

SYNOPSIS

```
long table( ... ).col_index( const char *col_name ) const;
```

DESCRIPTION

`table().col_index()` returns an index of the column of which the name is `col_name`.

PARAMETER

[I] `col_name` column name
([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	column index.
Negative value (error)	:	not found.

EXAMPLES

The following code displays all the column name and index of the table.

```
long idx;
for ( idx=0 ; idx < fits.table("EVENT").col_length() ; idx++ ) {
    const char *col_name = fits.table("EVENT").col_name(idx);
    long col_idx = fits.table("EVENT").col_index(col_name);
    printf("Column Name=%s\tColumn Number=%ld\n", col_name, col_idx);
}
```

13.8.7 table().col_name()

NAME

table().col_name() — Column name

SYNOPSIS

```
const char *table( ... ).col_name( long col_index ) const;
```

DESCRIPTION

`table().col_name()` returns the name of the column of which the index is `col_index`.

PARAMETER

[I] `col_index` column index
([I] : input, [O] : output)

RETURN VALUE

`table().col_name()` returns an address of the column name string.

EXAMPLES

See EXAMPLES of 13.8.6

13.8.8 table().col().type()

NAME

table().col().type() — Column type

SYNOPSIS

```
int table( ... ).col( ... ).type() const;
```

DESCRIPTION

`table().col().type()` returns the type of column.

The type is one of the following – `FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, `FITS::BYTE_T`, `FITS::BIT_T`, `FITS::LOGICAL_T`, `FITS::COMPLEX_T`, `FITS::DOUBLECOMPLEX_T`, `FITS::ASCII_T`, `FITS::LONGARRDESC_T`, `LLONGARRDESC_T`.

EXAMPLES

The example code gets following information of all the columns in the table.

- Type of the column
- Bytes of the column type
- Number of elements within the column
- Byte length of the column

```
long idx;
for ( idx=0 ; idx < fits.table("EVENT").col_length() ; idx++ ) {
    int c_type = fits.table("EVENT").col(idx).type();
    long c_bytes = fits.table("EVENT").col(idx).bytes();
    long c_elem_len = fits.table("EVENT").col(idx).elem_length();
    long c_elem_byte_len = fits.table("EVENT").col(idx).elem_byte_length();
                :
                :
}
```

13.8.9 table().col().heap_is_used()

NAME

table().col().heap_is_used() — Test the column for using variable length array

SYNOPSIS

```
bool table( ... ).col( ... ).heap_is_used() const;
```

DESCRIPTION

`table().col().heap_is_used()` member function returns `true` when the column has variable length array, otherwise it returns `false`.

13.8.10 table().col().heap_type()

NAME

table().col().heap_type() — Type of heap used by a column

SYNOPSIS

```
int table( ... ).col( ... ).heap_type() const;
```

DESCRIPTION

`table().col().heap_type()` returns the type of heap used by specified column.

The type is one of the following – `FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, `FITS::BYTE_T`, `FITS::BIT_T`, `FITS::LOGICAL_T`, `FITS::COMPLEX_T`, `FITS::DOUBLECOMPLEX_T`, `FITS::ASCII_T`.

13.8.11 table().col().bytes()

NAME

table().col().bytes() — Bytes of column type

SYNOPSIS

```
long table( ... ).col( ... ).bytes() const;
```

DESCRIPTION

When the type of column is not `FITS::ASCII_T`, `table().col().bytes()` function returns the bytes of column type. For example, if column type is `FITS::DOUBLE_T`, then it returns sizeof(fits::double_t). When the type of column is `FITS::BIT_T`, it returns 1.

When the column type is `FITS::ASCII_T`, it returns the string length of minimum element in the column determined by specification of `TFORM`*n* and `TDIM`*n*. Concrete examples are shown in the following table.

`TFORM`*n* and `TDIM`*n*	`.bytes()`	`.dcol_length()`	`.drow_length()`	`.elem_length()`
`TFORM`*n* = `'120A'`	120	1	1	1
`TFORM`*n* = `'120A10'`	10	12	1	12
`TFORM`*n* = `'120A10'` `TDIM`*n* = `'(6,2)'`	10	6	2	12
`TFORM`*n* = `'120A'` `TDIM`*n* = `'(10,6,2)'`	10	6	2	12

RETURN VALUE

`table().col().bytes()` returns the bytes of column type.

EXAMPLES

See EXAMPLES of 13.8.8

13.8.12 table().col().elem_byte_length()

NAME

table().col().elem_byte_length() — Byte length of the column

SYNOPSIS

```
long table( ... ).col( ... ).elem_byte_length() const;
```

DESCRIPTION

`table().col().elem_byte_length()` returns byte length of the column. For example, if `TTYPE`*n* is `16D`, it returns `sizeof(fits::double_t)*16`.

RETURN VALUE

`table().col().elem_byte_length()` returns the byte length of the column.

EXAMPLES

See EXAMPLES of 13.8.8

13.8.13 table().col().elem_length()

NAME

table().col().elem_length() — Number of elements in the column

SYNOPSIS

```
long table( ... ).col( ... ).elem_length() const;
```

DESCRIPTION

`table().col().elem_length()` returns the number of elements in the column.

If column type is not `FITS::ASCII_T` – for example, when `TTYPE`*n* is `16D`, it returns 16 (no relation with `TDIM`*n*).

If column type is `FITS::ASCII_T`, See the table in table().col().bytes() (13.8.11)

RETURN VALUE

`table().col().elem_length()` returns the number of elements in the column.

EXAMPLES

See EXAMPLES of 13.8.8

13.8.14 table().col().dcol_length()

NAME

table().col().dcol_length() — Number of elements in a row defined by `TDIM`*n*

SYNOPSIS

```
long table( ... ).col( ... ).dcol_length() const;
```

DESCRIPTION

`table().col().dcol_length()` returns a number of elements in a row in a column defined by `TDIM`*n* of column.

If the column type is not `FITS::ASCII_T` – for example, `TDIM`*n* is `(8x2)`, it returns 8.

If the column type is `FITS::ASCII_T`, See the table in table().col().bytes() (13.8.11)

RETURN VALUE

`table().col().dcol_length()` returns the number of elements in the row defined by `TDIM`*n*.

EXAMPLES

```
long dcol_count = fits.table("EVENT").col(0L).dcol_length();
```

13.8.15 table().col().drow_length()

NAME

table().col().drow_length() — Number of rows defined by `TDIM`*n*

SYNOPSIS

```
long table( ... ).col( ... ).drow_length() const;
```

DESCRIPTION

`table().col().drow_length()` returns number of rows defined by `TDIM`*n*.

In the case of the columm type except `FITS::ASCII_T`, it returns 2 for a column with `(8x2)` of its `TDIM`*n*.

See table (13.8.11) in table().col().bytes(),if column type is `FITS::ASCII_T`.

RETURN VALUE

`table().col().drow_length()` returns the number of rows defined by `TDIM`*n*.

EXAMPLES

```
long drow_count = fits.table("EVENT").col(0L).drow_length();
```

13.8.16 table().col().heap_bytes()

NAME

table().col().heap_bytes() — Bytes of heap type

SYNOPSIS

```
long table( ... ).col( ... ).heap_bytes() const;
```

DESCRIPTION

`table().col().heap_bytes()` function returns the bytes of heap type of specified column. For example, if heap type is `FITS::DOUBLE_T`, then it returns sizeof(fits::double_t).

When the type of heap is `FITS::BIT_T`, it returns 1.

13.8.17 table().col().max_array_length()

NAME

table().col().max_array_length() — Maximum length of variable length array

SYNOPSIS

```
long table( ... ).col( ... ).max_array_length() const;
```

DESCRIPTION

This member function returns maximum length of variable length array of specified column.

13.8.18 table().col().array_length()

NAME

table().col().array_length() — Length of variable length array

SYNOPSIS

```
long table( ... ).col( ... ).array_length( long row_idx, long elem_idx = 0 ) const;
```

DESCRIPTION

This member function returns length of variable length array of specified column and row.

A negative value is returned for errors.

13.8.19 table().col().definition()

NAME

table().col().definition() — Definition of column

SYNOPSIS

```
const fits::table_def &table( ... ).col( ... ).definition() const;
```

DESCRIPTION

`table().col().definition()` returns the reference to a structure object defining the column. It is used when copying the definition of the column to that of another column.

RETURN VALUE

`table().col().definition()` returns a reference to table_def object.

EXAMPLES

See examples in 13.8.46

13.8.20 table().col().dvalue()

NAME

table().col().dvalue() — Returns cell value as a real number value

SYNOPSIS

```
double table( ... ).col( ... ).dvalue( long row_index ) const;
double table( ... ).col( ... )
                    .dvalue( long row_index,
                             const char *elem_name, long repetition_idx = 0 ) const;
double table( ... ).col( ... )
                    .dvalue( long row_index,
                             long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().dvalue()` reflects `TZERO`*n* and `TSCAL`*n* to the value of cell and returns it as a real number value. If the value is NULL, it returns `NAN`. It returns `NAN` also when the value of the cell equals to that of `TNULL`*n* in ASCII tables or in integer type columns of binary tables.

`TZERO`*n* and `TSCAL`*n* value of header is valid when `TFORM`*n* of binary table includes `'B'`,`'I'`, `'J'`,`'K'`,`'E'`,or `'D'`, or when , `TFORM`*n* of ASCII table includes `'I'`,`'L'`,`'F'`,`'E'`, `'G'`, or `'D'`.[40)]

When the column type is boolean, it returns 1 if the value is `'T'`, and returns 0 if the value is `'F'`, otherwise it returns `NAN`.

When column of binary table is string type or when the `TFORM`*n* of column in ASCII table does not represent numeric value, it directly returns the real number value converted from the string of cell.

[40)] SFITSIO supports 'L' and 'G' which are not included in the definition of FITS.

If the `TFORM`*n* of column in ASCII table does represent numeric value, it converts the string of cell to real number value and covert it by `TZERO`*n* and `TSCAL`*n* and returns it.

Since it converts string of cell by removing spaces and atof() of libc, the convertible string is decimal integer, hex integer, or real number value.

If the argument is NULL or invalid, it returns `NAN`.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().dvalue()` returns the cell value.

EXAMPLES

Following code displays all cell values of the column "TIME" in the table "EVENT".

```
fits_table_col &col_ref = fits.table("EVENT").col("TIME");
long i;
for ( i=0 ; i < col_ref.length() ; i++ ) {
    printf("%f\n",col_ref.dvalue(i));
}
```

13.8.21 table().col().lvalue(), table().col().llvalue()

NAME

table().col().lvalue(), table().col().llvalue() — Return cell value as integer

SYNOPSIS

```
long table( ... ).col( ... ).lvalue( long row_index ) const;
long table( ... ).col( ... )
                 .lvalue( long row_index,
                          const char *elem_name, long repetiti_idx = 0 ) const;
long table( ... ).col( ... )
                 .lvalue( long row_index,
                          long elem_index, long repetition_idx = 0 ) const;
long long table( ... ).col( ... ).llvalue( long row_index ) const;
long long table( ... ).col( ... )
                      .llvalue( long row_index,
                                const char *elem_name, long repetiti_idx = 0 ) const;
long long table( ... ).col( ... )
                      .llvalue( long row_index,
                                long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().lvalue()` and `table().col().llvalue()` reflect `TZERO`*n* and `TSCAL`*n* to the value of cell and return the nearest integer of the real number value. If the value is NULL, it returns `INDEF_LONG` or `INDEF_LLONG`. It returns `INDEF_LONG` or `INDEF_LLONG` also when the value of cell equals to that of `TNULL`*n* of ASCII table or that of integer type column of binary table.

`TZERO`*n* and `TSCAL`*n* value of header is valid when `TFORM`*n* of binary table includes `'B'`,`'I'`, `'J'`,`'K'`,`'E'`,or `'D'`, or , `TFORM`*n* of ASCII table includes `'I'`,`'L'`,`'F'`,`'E'`, `'G'`, or `'D'`.[41)]

When the column type is boolean, it returns 1 if the value is `'T'`, and returns 0 if the value is `'F'`, otherwise it returns `INDEF_LONG` or `INDEF_LLONG`.

When column of binary table is string type or when the `TFORM`*n* of column in ASCII table does not represent numeric value, it directly returns the nearest integer of real number value converted from the string of cell.

If the `TFORM`*n* of column in ASCII table does represent numeric value, it converts the string of cell to real number value and covert it by `TZERO`*n* and `TSCAL`*n* and returns the nearest integer.

Since it converts string of cell by removing spaces and atof() of libc, the convertible string is decimal integer, hex integer, or real number value.

If the argument is NULL or invalid, it returns `INDEF_LONG` or `INDEF_LLONG` which is cast to the returned type.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().lvalue()` and `table().col().llvalue()` return the cell value.

EXAMPLES

See EXAMPLES of 13.8.20

13.8.22 table().col().bvalue()

NAME

table().col().bvalue() — Returns cell as boolean

SYNOPSIS

```
bool table( ... ).col( ... ).bvalue( long row_index ) const;
bool table( ... ).col( ... )
```

[41)] SFITSIO supports 'L' and 'G' which are not included in the definition of FITS.

```
                  .bvalue( long row_index,
                           const char *elem_name, long repetiti_idx = 0 ) const;
bool table( ... ).col( ... )
                  .bvalue( long row_index,
                           long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().bvalue()` returns cell value as boolean. The return value is `true` or `false`. If you need three kinds of value, `'T'`,`'F'` and `'U'`, then use table().col().logical_value() (13.9.19).

When the column type is boolean type,it returns `true` if the value is `'T'` and it returns `false` if the value is `'F'`.

In the case of string type column of binary table that the `TFORM`*n* of column in ASCII table does not represent numeric value, it converts cell string to real number, and if the nearest integer is 0 then it returns `false`, otherwise it returns `true`.

If the `TFORM`*n* of column in ASCII table does represent numeric value, it converts the string of cell to real number value and covert it further by `TZERO`*n* and `TSCAL`*n*. If the nearest integer to the converted value is 0, then it returns `false`, otherwise `true`. Since it converts string of cell by atof() of libc after removing spaces, the convertible string is decimal integer, hex integer, or real number value.

When the cell string cannot be convert to real number value, if the string begins with either `'T'` or `'t'` then it returns `true`, and otherwise `false`.

When the column type is integer or real number, if the nearest integer of the cell value reflected `TZERO` and `TSCAL`*n* is zero, it returns `false`, otherwise `true`. If the argument is NULL or invalid, it returns `false`. It returns `false` also when the cell value is as the same as `TNULL`*n* value of ASCII table or integer type column of binary table.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().bvalue()` returns the cell value.

EXAMPLES

See EXAMPLES in 13.8.20

13.8.23 table().col().svalue()

NAME

table().col().svalue() — Returns cell value as string

SYNOPSIS

```
const char *table( ... ).col( ... ).svalue( long row_index );
const char *table( ... ).col( ... )
                         .svalue( long row_index,
                                  const char *elem_name, long repetiti_idx = 0 );
const char *table( ... ).col( ... )
                         .svalue( long row_index,
                                  long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().svalue()` returns cell value as string.

As for string type column of binary table, and column of ASCII table whare `TFORM`*n* does not represent numeric value, it returns string of cell which is formatted by `TFORM`*n*. If `TFORM`*n* is not given, it returns raw cell string.

If the column of ASCII table does represent numeric value, it converts cell string to real number value, and then converts it with `TZERO`*n* and `TSCAL`*n*, and returns the value which is formatted as a string with `TFORM`*n*.

When `TDISP`*n* is given in the boolean column, it returns `"T"` , `"F"`, or `"U"`. When `TDISP`*n* is given, if the value is `'T'`, then it formats 1 by `TDISP`*n*, and returns it. If the value is `'F'`, then it formats 0 by `TDISP`*n*, and returns it.

If the column type is integer or real number, it converts cell value with `TZERO`*n* and `TSCAL`*n* of the header, and converts it to string and returns it. If `TDISP`*n* is specified, it converts with `TDISP`*n* and returns it.

When the column type is integer, `TZERO`*n* is 0,`TSCAL`*n* is 1.0, and no `TDISP`*n* is given, it returns the string which is converted by format `"%lld"` of printf(). Otherwise, it converts with the following printf format.

`FITS::FLOAT_T`	...	`"%G"`
`FITS::DOUBLE_T`, `FITS::LONGLONG_T`	...	`"%.15G"`
`FITS::LONG_T`	...	`"%.10G"`
otherwise	...	`"%.8G"`

lll

When the cell value is NULL or the argument is invalid, it returns string `"NULL"`. Note that there might be an space character padded for some given `TDISP`*n*. It also returns `"NULL"` when the value of cell is `TNULL`*n* of ASCII table or integer type column of binary table. This NULL string value (default is `"NULL"`) can be changed by table().assign_null_svalue() member function (§13.8.29).

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().svalue()` returns an address of string for cell value.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.8.20

13.8.24 table().col().get_svalue()

NAME

table().col().get_svalue() — Returns cell value as string

SYNOPSIS

```
ssize_t table( ... ).col( ... ).get_svalue( long row_index,
                                            char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_svalue( long row_index,
                                            const char *elem_name,
                                            char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_svalue( long row_index,
                                            const char *elem_name, long repetition_idx,
                                            char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_svalue( long row_index,
                                            long elem_index,
                                            char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_svalue( long row_index,
                                            long elem_index, long repetition_idx,
                                            char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`table().col().get_svalue()` returns the cell value as string and store it to `dest_buf`.

When column of binary table is string type or when the `TFORM`*n* of column in ASCII table does not represent numeric value, it returns string of cell which is formatted by `TFORM`*n*. If `TFORM`*n* is not given, it returns raw cell string.

If the column of ASCII table does represent numeric value, it converts cell string to real number value, and then converts it with `TZERO`*n* and `TSCAL`*n*, and returns the value which is formatted as a string with `TFORM`*n*.

When `TDISP`*n* is given in the boolean column, it returns `"T"`, `"F"`, or `"U"`. When `TDISP`*n* is given, If the value is `'T'`, then it formats 1 by `TDISP`*n*, and returns it. If the value is `'F'`, then it formats 0 by `TDISP`*n* and returns it.

If the column type is integer or real number, it converts cell value with `TZERO`*n* and `TSCAL`*n* of the header, and converts it to string and returns it. If `TDISP`*n* is specified, it converts with `TDISP`*n* and returns it.

When the column type is integer, `TZERO`*n* is 0,`TSCAL`*n* is 1.0, and no `TDISP`*n* is given, it returns the string which is converted by format `"%lld"` of printf(). Otherwise, it converts with the following printf format.

FITS::FLOAT_T	...	"%G"
FITS::DOUBLE_T, FITS::LONGLONG_T	...	"%.15G"
FITS::LONG_T	...	"%.10G"
otherwise	...	"%.8G"

When the cell value is NULL or the argument is invalid, it returns string "NULL". Note that there might be an space character padded for some given TDISP*n*. It also returns "NULL" when the value of cell is TNULL*n* of ASCII table or integer type column of binary table. This NULL string value (default is "NULL") can be changed by table().assign_null_svalue() member function (§13.8.29).

To specify row, use row_index. To specify element, use elem_name or elem_index. For elem_name, a name which exists in TELEM*n* can be specified.

If TDIM*n* is specified, elem_index can be specidied as the first dimensional index, and repetition_idx can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	row_index	row index
[I]	elem_name	element name
[I]	elem_index	element index (the first dimension index of TDIM*n*)
[I]	repetiton_index	second dimensional index
[O]	dest_buf	address of destination buffer
[I]	buf_size	the size of dest_buf

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	number of characters which is able to copy when the buffer length is sufficient (excluding '\0').
Negative value (error)	:	the case when copy was not done because of invalid argument.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB (sli::err_rec).

EXAMPLES

```
char buf[128];
fits.table("EVENT").col("TIME").get_svalue( 0, buf, sizeof(buf) );
```

13.8.25 table().col().assign()

NAME

table().col().assign() — Assign value to cell as real number

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                            .assign( double value, long row_index );
fits_table_col &table( ... ).col( ... )
                            .assign( double value, long row_index,
                                     const char *elem_name,
                                     long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( double value, long row_index,
                                     long elem_index, long repetition_idx = 0 );
```

```
fits_table_col &table( ... ).col( ... )
                            .assign( float value, long row_index );
fits_table_col &table( ... ).col( ... )
                            .assign( float value, long row_index,
                                     const char *elem_name,
                                     long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( float value, long row_index,
                                     long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign()` reflects `TZERO`*n* and `TSCAL`*n* to the given `value` of real number `value` and assign the generated real number to the cell.

If `value` is `NAN`, it is handled as if NULL is given. In this case, if it has a `TNULL`*n* `value` in integer type column of binary table or ASCII table, it assigns the `value` to the cell.

`TZERO`*n* and `TSCAL`*n* `value` of header is valid when `TFORM`*n* of binary table includes `'B'`, `'I'`, `'J'`, `'K'`, `'E'`, or `'D'`, or , `TFORM`*n* of ASCII table includes `'I'`, `'L'`, `'F'`, `'E'`, `'G'`, or `'D'`.[42)]

If the column type is integer, it converts `value` with `TZERO`*n* and `TSCAL`*n* and assigns the nearest integer to the cell.

When the column type is boolean, if the nearest integer to `value` is 0 then it assigns `'F'` to the cell, and `'T'` otherwise. If NULL(`NAN`) is given, it assigns `'\0'`.

When column of binary table is string type or when the `TFORM`*n* of column in ASCII table does not represent numeric value, it converts with format `"%.15G"` of `printf()` (when the `value` is double), or format `"%G"` of `printf()` (when the `value` is float), and assigns the string to the cell.

When the `TFORM`*n* of the column of ASCII table does represent numeric `value`, it converts `value` with `TZERO`*n* and `TSCAL`*n*, formats it with `TFORM`*n*, and assigns the formatted string.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

If the argument is invalid, it does not do anything.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : Input, [O] : Output)

RETURN VALUE

`table().col().assign()` returns a reference to the fits_table_col object.

EXCEPTION

When it fails to manipulate internal buffer, it raises exception derived from SLLIB (`sli::err_rec`)

[42)] SFITSIO supports 'L' and 'G' which are not included in the definition of fits.

EXAMPLES

```
double value = 0;
fits.table("EVENT").col("TIME").assign(value, 0);
```

13.8.26 table().col().assign()

NAME

table().col().assign() — Assign value to cell as integer

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                            .assign( int value, long row_index );
fits_table_col &table( ... ).col( ... )
                            .assign( int value, long row_index,
                                     const char *elem_name,
                                     long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( int value, long row_index,
                                     long elem_index, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( long value, long row_index );
fits_table_col &table( ... ).col( ... )
                            .assign( long value, long row_index,
                                     const char *elem_name,
                                     long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( long value, long row_index,
                                     long elem_index, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( long long value, long row_index );
fits_table_col &table( ... ).col( ... )
                            .assign( long long value, long row_index,
                                     const char *elem_name,
                                     long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                            .assign( long long value, long row_index,
                                     long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign()` reflect `TZERO`n and `TSCAL`n to integer `value` and assign the generated real number to the cell. NULL `value` cannot be given to these function.

In order to give NULL `value`, give double of float type `NAN` to argument of 13.8.25 functions.

`TZERO`n and `TSCAL`n `value` of header is valid when `TFORM`n of binary table includes `'B'`,`'I'`, `'J'`,`'K'`,`'E'`,or `'D'`, or , `TFORM`n of ASCII table includes `'I'`,`'L'`,`'F'`,`'E'`, `'G'`, or `'D'`.[43)]

If the column type is integer, it converts `value` with `TZERO`n and `TSCAL`n and assigns the nearest integer to the cell.

[43)] SFITSIO supports 'L' and 'G' which are not included in the definition of FITS.

When the column type is boolean, if the nearest integer to `value` is 0 then it assigns `'F'` to the cell, and `'T'` otherwise.

When column of binary table is string type or when the `TFORM`*n* of column in ASCII table does not represent numeric value, it converts with format `"%lld"` of `printf()`. If `TFORM`*n* is specified, it additionally formats it.

If the `TFORM`*n* of the column in ASCII table does represent numeric `value`, it converts `value` with `TZERO`*n* and `TSCAL`*n*, formats it with `TFORM`*n*, and assign it to the cell.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins at zero.

If the argument is invalid, it does not do anything.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign()` returns a reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec`)

EXAMPLES

See the EXAMPLES at 13.8.25

13.8.27 table().col().assign()

NAME

table().col().assign() — Assign value to cell as string

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                             .assign( const char *value, long row_index );
fits_table_col &table( ... ).col( ... )
                             .assign( const char *value, long row_index,
                                      const char *elem_name,
                                      long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... )
                             .assign( const char *value, long row_index,
                                      long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign()` assigns `value` to cell as string. If `value` is `NULL` or string `"NULL"` (You may insert spaces in the begining or end of the string) , then it is handled as if `NULL`

was given. In this case, if it has a `TNULL`*n* `value` in integer type column of binary table or ASCII table, it assigns the `value` to the cell. This NULL string value (default is `"NULL"`) can be changed by table().assign_null_svalue() member function (§13.8.29).

As for string type column of binary table, and column of ASCII table which `TFORM`*n* does not represent numeric `value`, it returns string of cell which is formatted by `TFORM`*n*. If `TFORM`*n* is not given, it returns raw cell string.

If the column of ASCII table does represent numeric `value`, it converts cell string to real number `value`, and then converts it with `TZERO`*n* and `TSCAL`*n*, and returns the `value` which is formatted as a string with `TFORM`*n*. Since it converts string of cell by atof() of libc after removing spaces, the string which can be converted is decimal integer, hex integer, or real number `value`.

When type column type is boolean and the `value` can be converted to real number, it assigns `'T'` if the `value` is not zero, `'F'` if the `value` is zero, and `'\0'` if the `value` is NAN. When the `value` cannot be converted to real number, it assigns `'T'` if the `value` begins with `'T'` or `'t'`, `'F'` if the `value` begins with `'F'` or `'f'`, and `'\0'` otherwise.

When the column type is integer or real number, it converts `value` to real number, converts it with `TZERO`*n* and `TSCAL`*n*, and assigns it to the cell. If the column type is integer, it assigns nearest integer.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins at zero.

if the argument is invalid, it does not do anything.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign()` returns a reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec`)

EXAMPLES

See the EXAMPLES at 13.8.25

13.8.28 table().col().convert_type()

NAME

table().col().convert_type() — Convert or modify data type

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).convert_type( int new_type );
```

```
fits_table_col &table( ... ).col( ... ).convert_type( int new_type,
                                                      double new_zero );
fits_table_col &table( ... ).col( ... ).convert_type( int new_type,
                                                      double new_zero,
                                                      double new_scale );
fits_table_col &table( ... ).col( ... ).convert_type( int new_type,
                                                      double new_zero,
                                                      double new_scale,
                                                      double new_null );
```

DESCRIPTION

`table().col().convert_type()` converts type of column which is integer or real number (in case that `TFORM`*n* includes `'B'`, `'I'`, `'J'`, `'K'`, `'E'`, or `'D'`) to `new_type`. It resizes the internal buffer if needed. The value available as `new_type` is `FITS::DOUBLE_T`, `FITS::FLOAT_T`, `FITS::LONGLONG_T`, `FITS::LONG_T`, `FITS::SHORT_T`, or `FITS::BYTE_T`. If `new_zero`, `new_scale`, `new_null` is given, it modifies `TZERO`*n*,`TSCAL`*n*,`TNULL`*n*, and convert the data which is reflect to them. Argument `new_null` is available only if `new_type` is integer type.

String type column or boolean type column cannot be converted by this function.

PARAMETER

[I]	`new_type`	new type
[I]	`new_zero`	new `TZERO` value
[I]	`new_scale`	new `TSCAL` value
[I]	`new_null`	new `TNULL` value

([I] : input, [O] : output)

RETURN VALUE

`table().col().convert_type()` returns a reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, table area reallocation failure when expanding the table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").col("PIX_DATA").convert_type(FITS::DOUBLE_T);
```

13.8.29 table().assign_null_svalue()

NAME

table().assign_null_svalue() — Manipulate NULL string value (high level)

SYNOPSIS

```
fits_table &table( ... ).assign_null_svalue( const char *snull );
```

DESCRIPTION

`table().assign_null_svalue()` assigns the high-level NULL string for `table().col().svalue()`(§13.8.2
`table().col().assign()`(§13.8.27).

The default value of NULL string is `"NULL"`. You can change it using `assign_null_svalue()` member function.

PARAMETER

[I] `snull` NULL string value to set

([I] : input, [O] : output)

RETURN VALUE

`assign_null_svalue()` returns the reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, table area reallocation failure when expanding the table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

13.8.30 table().col().tzero(), table().col().assign_tzero()

NAME

table().col().tzero(), table().col().assign_tzero() — Manipulate zero point

SYNOPSIS

```
double table( ... ).col( ... ).tzero() const;
bool table( ... ).col( ... ).tzero_is_set() const;
fits_table_col &table( ... ).col( ... ).assign_tzero( double zero, int prec = 15 );
fits_table_col &table( ... ).col( ... ).erase_tzero();
```

DESCRIPTION

`table().col().tzero()` returns value of `TZERO`*n*.

`table().col().assign_tzero()` assigns the value of `TZERO`*n*. `prec` indicates precision. If `prec` is omitted then it write data to header record with 15 digit precision.

table().col().erase_tzero() erases configuration of `TZERO`*n*.

PARAMETER

[I] `zero` `TZERO` value to modify

[I] `prec` precision (places)

([I] : input, [O] : output)

RETURN VALUE

`tzero()` returns the value of `TZERO`*n*.

`tzero_is_set()` returns whether `TZERO`*n* is defined or not.

`assign_tzero()` and `erase_tzero()` returns the reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec`)

EXAMPLES

```
if ( fits.table("EVENT").col("PIX_DATA").tzero_is_set() == false ) {
    fits.table("EVENT").col("PIX_DATA").assign_tzero(0.0);
}
```

13.8.31 table().col().tscal(), table().col().assign_tscal()

NAME

table().col().tscal(), table().col().assign_tscal() — Manipulate scaling factor

SYNOPSIS

```
double table( ... ).col( ... ).tscal() const;
bool table( ... ).col( ... ).tscal_is_set() const;
fits_table_col &table( ... ).col( ... ).assign_tscal( double scal, int prec = 15 );
fits_table_col &table( ... ).col( ... ).erase_tscal();
```

DESCRIPTION

`table().col().tscal()` returns value of `TSCAL`*n*.

`table().col().assign_tscal()` assigns the value of `TSCAL`*n*. `prec` indicates precision. If `prec` is omitted then it write data to header record with 15 digit precision.

`table().col().erase_tscal()` erases configuration of `TSCAL`*n*.

PARAMETER

[I] `scal` `TSCAL` value to modify
[I] `prec` precision (places)
([I] : input, [O] : output)

RETURN VALUE

`tscal()` returns the value of `TSCAL`*n*.
`tscal_is_set()` returns whether `TSCAL`*n* is defined or not.
`assign_tscal()` and `erase_tscal()` returns the reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, table area reallocation failure when expanding the table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
if ( fits.table("EVENT").col("PIX_DATA").tscal_is_set() == false ) {
    fits.table("EVENT").col("PIX_DATA").assign_tscal(1.0);
}
```

13.8.32 table().col().tnull(), table().col().assign_tnull()

NAME

table().col().tnull(), table().col().assign_tnull() — Manipulate NULL value

SYNOPSIS

```
long long table( ... ).col( ... ).tnull( const char **tnull_ptr = NULL ) const;
bool table( ... ).col( ... ).tnull_is_set() const;
fits_table_col &table( ... ).col( ... ).assign_tnull( long long null );
fits_table_col &table( ... ).col( ... ).erase_tnull();
```

DESCRIPTION

`table().col().tnull()` returns value of `TNULL`*n*. As for Ascii Table, and `TNULL` value of string is needed, it is able to get address of internal buffer by using `tnull_ptr`.

`table().col().assign_tnull()` assigns the value of `TNULL`*n*.

`table().col().erase_tnull()` erases configuration of `TNULL`*n*.

PARAMETER

[I] `null` `TNULL` value to set
[O] `tnull_ptr` addres to string type `TNULL` value (Ascii Table only)
([I] : input, [O] : output)

RETURN VALUE

`tnull()` returns the value of `TNULL`*n*.
`tnull_is_set()` returns whether `TNULL`*n* is defined or not.
`assign_tnull()` and `erase_tnull()` returns the reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, table area reallocation failure when expanding the table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
if ( fits.table("EVENT").col("PIX_DATA").tnull_is_set() == false ) {
    fits.table("EVENT").col("PIX_DATA").assign_tnull(-1);
}
```

13.8.33 table().col().tunit(), table().col().assign_tunit()

NAME

table().col().tunit(), table().col().assign_tunit() — Manipulate units of measurement

SYNOPSIS

```
const char *table( ... ).col( ... ).tunit() const;
bool table( ... ).col( ... ).tunit_is_set() const;
fits_table_col &table( ... ).col( ... ).assign_tunit( const char *unit );
fits_table_col &table( ... ).col( ... ).erase_tunit();
```

DESCRIPTION

`table().col().tunit()` returns value of `TUNIT`n.

`table().col().assign_tunit()` assigns the value of `TUNIT`n.

`table().col().erase_tunit()` erases configuration of `TUNIT`n.

PARAMETER

[I] `unit` `TUNIT` value to modify
([I] : input, [O] : output)

RETURN VALUE

`tunit()` returns the value of `TUNIT`n.
`tunit_is_set()` returns whether `TUNIT`n is defined or not.
`assign_tunit()` and `erase_tunit()` returns the reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, table area reallocation failure when expanding the table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
if ( fits.table("EVENT").col("RA").tunit_is_set() == false ) {
    fits.table("EVENT").col("RA").assign_tunit("deg");
}
```

13.8.34 table().init()

NAME

table().init() — Init table

SYNOPSIS

```
fits_table &table( ... ).init();
fits_table &table( ... ).init( const fits::table_def defs[] );
```

DESCRIPTION

`table().init()` erases all the contents of header and table and initialize it.

If `defs` is given, it creates column in accordance with it.

PARAMETER

[I] `defs` `fits::table_def` structure
([I] : input, [O] : output)

RETURN VALUE

`table().init()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").init();
```

13.8.35 table().col().init()

NAME

table().col().init() — Init column

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).init();
fits_table_col &table( ... ).col( ... ).init( const fits_table_col &src );
```

DESCRIPTION

`table().col().init()` erases all the contents of the column and initialize it.

If `src` is given, it overwrites existing column with contents of `src`. Number of rows is not changed by using this member function. Note that last some cells in `src` will not be copied when number of rows of `src` is larger than that of destination table.

PARAMETER

[I] `src` reference of source column object
([I] : input, [O] : output)

RETURN VALUE

`table().col().init()` returns a reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

In this code, the name of column "LON" of table "EVENT" will be updated to "RA".

```
fits.table("EVENT").col("LON").init( fits_r.table("RAW").col("RA") );
```

13.8.36 table().ascii_to_binary()

NAME

table().ascii_to_binary() — Convert ascii table to binary table

SYNOPSIS

```
fits_table &table( ... ).ascii_to_binary();
```

DESCRIPTION

If the attribute of table is ascii, then `table().ascii_to_binary()` converts it to binary.

If the attribute is converted to binary, the value of `TFORM`*n* in ascii table (`tdisp` member of `fits::table_def` structure) is stored in the comment of `TFORM`*n* when saved as binary table.

`TNULL`*n* value is also stored in the comment (the value is undefined).

RETURN VALUE

`table().ascii_to_binary()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec`)

EXAMPLES

```
fits.table("X_CATALOG").ascii_to_binary();
```

13.8.37 table().assign_col_name()

NAME

table().assign_col_name() — Assign column name

SYNOPSIS

```
fits_table &table( ... ).assign_col_name( long col_index, const char *newname );
fits_table &table( ... ).assign_col_name( const char *col_name, const char *newname );
```

DESCRIPTION

`table().assign_col_name()` assigns column name to `newname` which specified by `col_index` or `col_name`

PARAMETER

[I]	`col_index`	column index
[I]	`col_name`	column name
[I]	`newname`	new column name

([I] : input, [O] : output)

RETURN VALUE

`table().assign_col_name()` returns a reference of the fit_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").assign_col_name(0L, "TIME");
```

13.8.38 table().define_a_col()

NAME

table().define_a_col() — Modify column definition

SYNOPSIS

```
fits_table &table( ... ).define_a_col( long col_index,
                                       const fits::table_def &def );
fits_table &table( ... ).define_a_col( const char *col_name,
                                       const fits::table_def &def );
```

DESCRIPTION

`table().define_a_col()` modifies definition of column specified by `col_index` or `col_name`.

Substitute `NULL` into members of `def` except to be modified.

PARAMETER

[I]	`col_index`	column index
[I]	`col_name`	column name
[I]	`defs`	`fits::table_def` structure

([I] : input, [O] : output)

RETURN VALUE

`table().define_a_col()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits::table_def def =
    { "TIME","satellite time", NULL,NULL, "s","", "F16.3", "1D", "" };
fits.table("EVENT").define_a_col(0L, def);
```

13.8.39 table().col_header_index()

NAME

table().col_header_index() — Index of header record for table column properties

SYNOPSIS

```
long table( ... ).col_header_index( const char *col_name,
                                    const char *kwd ) const;
long table( ... ).col_header_index( long col_index,
                                    const char *kwd ) const;
```

DESCRIPTION

table().col_header_index() returns an index of the fits_header_record object having keyword prefix `kwd` such as `TTYPE` for the table column specified by `col_index` or `col_name`.

PARAMETER

[I]	`col_index`	Column index
[I]	`col_name`	Column name
[I]	`kwd`	Prefix of column keyword

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	Record index
Negative value	:	Error (If specified keyword was not found.)

EXAMPLES

```
long idx = fits.table("EVENT").col_header_index("DATE", "TTYPE");
```

13.8.40 table().col_header()

NAME

table().col_header() — Reference of header record for table column properties

SYNOPSIS

```
fits_header_record &table( ... ).col_header( const char *col_name,
                                             const char *kwd );
fits_header_record &table( ... ).col_header( long col_index,
                                             const char *kwd );
const fits_header_record &table( ... ).col_header( const char *col_name,
                                                   const char *kwd ) const;
const fits_header_record &table( ... ).col_header( long col_index,
                                                   const char *kwd ) const;
```

DESCRIPTION

`table().col_header()` returns reference of the fits_header_record object having keyword prefix `kwd` such as `TTYPE` for the table column specified by `col_index` or `col_name`.

`.svalue()`, `.dvalue()`, `.lvalue()`, etc. can be used after `.col_header()`. See §13.4.4 and succeeding sections for these member functions.

PARAMETER

[I]	`col_index`	Column index
[I]	`col_name`	Column name
[I]	`kwd`	Prefix of column keyword

([I] : input, [O] : output)

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception). If the read-only API does not found specified column keyword, it throws an exception derived from SFITSIO.

EXAMPLES

```
printf("TTYPE of 'TIME' = %s\n",
       fits.table("EVENT").col_header("TIME","TTYPE").svalue());
```

13.8.41 table().update_col_header()

NAME

table().update_col_header() — Update a header recored of column properties

SYNOPSIS

```
fits_table &table( ... ).update_col_header( const char *col_name,
                            const char *kwd, const char *val, const char *com );
fits_table &table( ... ).update_col_header( long col_index,
                            const char *kwd, const char *val, const char *com );
```

DESCRIPTION

`table().update_col_header()` updates the fits_header_record object having keyword prefix `kwd` such as `TTYPE` for the table column specified by `col_index` or `col_name`. Internal properties in objects are updated simultaneously.

PARAMETER

[I]	`col_index`	Column index
[I]	`col_name`	Column name
[I]	`kwd`	Prefix of column keyword
[I]	`val`	Value of header record
[I]	`com`	Comment of header record

([I] : input, [O] : output)

RETURN VALUE

`table().update_col_header()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").update_col_header("TIME","TUNIT","s","unit");
```

13.8.42 table().erase_col_header()

NAME

table().erase_col_header() — Erase a header recored of column properties

SYNOPSIS

```
fits_table &table( ... ).erase_col_header( const char *col_name,
                                           const char *kwd );
fits_table &table( ... ).erase_col_header( long col_index,
                                           const char *kwd );
```

DESCRIPTION

`table().erase_col_header()` removes the fits_header_record object having keyword prefix `kwd` such as `TTYPE` for the table column specified by `col_index` or `col_name`. Internal properties in objects are updated simultaneously.

PARAMETER

[I]	`col_index`	Column index
[I]	`col_name`	Column name
[I]	`kwd`	Prefix of column keyword

([I] : input, [O] : output)

RETURN VALUE

`table().erase_col_header()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").erase_col_header("TIME","TUNIT");
```

13.8.43 table().rename_col_header()

NAME

table().rename_col_header() — Change keyword prefix of user-defined column headers

SYNOPSIS

```
fits_table &table( ... ).rename_col_header( const char *old_kwd,
                                           const char *new_kwd );
```

DESCRIPTION

This member function changes keyword prefix of user-defined column headers from `old_kwd` to `new_kwd`. Keywords defined in FITS standard (e.g., `TTYPE`n) cannot be changed.

PARAMETER

[I] `old_kwd` Keyword prefix to be changed
[I] `old_kwd` New keyword prefix
([I] : input, [O] : output)

RETURN VALUE

`table().rename_col_header()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

This code renames user-defined column keyword `TLMAX`n to `TMMAX`n.

```
fits.table("EVENT").rename_col_header("TLMAX","TMMAX");
```

13.8.44 table().sort_col_header()

NAME

table().sort_col_header() — Sort column header records

SYNOPSIS

```
fits_table &table( ... ).sort_col_header();
```

DESCRIPTION

`table().sort_col_header()` sorts all column header records in column order.

RETURN VALUE

`table().sort_col_header()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").sort_col_header();
```

13.8.45 table().swap()

NAME

table().swap() — Swap table

SYNOPSIS

```
fits_table &table( ... ).swap( fits_table &obj );
```

DESCRIPTION

`table().swap()` swaps the own content with that of `obj`.

PARAMETER

[I/O] `obj` swap target object
([I] : input, [O] : output)

RETURN VALUE

`table().swap()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code swaps EVENT table and EVENT_SAVE table of the fits object.

```
fits.table("EVENT").swap(fits.table("EVENT_SAVE"));
```

13.8.46 table().append_cols(), table().append_a_col()

NAME

table().append_cols(), table().append_a_col() — Append column

SYNOPSIS

```
fits_table &table( ... ).append_cols( const fits::table_def defs[] );
fits_table &table( ... ).append_cols( fits_table &src );
fits_table &table( ... ).append_a_col( const fits::table_def &def );
fits_table &table( ... ).append_a_col( fits_table_col &src );
```

DESCRIPTION

`table().append_cols()` and `table().append_a_col()` append column to the table. `append_cols()` appends multiple columns, `append_a_col()` appends single column.

If `src` is specified, not only column definition of `src` but also data area is copied. However, if there is not enough rows, not all rows are copied.

PARAMETER

[I] `defs` `fits::table_def` structure
[I] `src` object which has the column to be copied
([I] : input, [O] : output)

RETURN VALUE

`table().append cols()` and `table().append_a_col()` return a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to allocate new table), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
const table_def &def = fits.table("EVENT").col(0L).definition();
fits.table("EVENT_SAVE").append_a_col(def);
```

13.8.47 table().insert_cols(), table().insert_a_col()

NAME

table().insert_cols(), table().insert_a_col() — Insert column

SYNOPSIS

```
fits_table &table( ... ).insert_cols( long index, const fits::table_def defs[]);
fits_table &table( ... ).insert_cols( const char *col_name,
                                      const fits::table_def defs[] );
fits_table &table( ... ).insert_cols( long index, fits_table &src );
fits_table &table( ... ).insert_cols( const char *col_name, fits_table &src );
fits_table &table( ... ).insert_a_col( long col_index,
                                       const fits::table_def &def );
fits_table &table( ... ).insert_a_col( const char *col_name,
                                       const fits::table_def &def );
```

DESCRIPTION

`table().insert_cols()` and `table().insert_a_col()` insert new column before column specified by `index` or `col_name`. `insert_cols()` inserts multiple columns, and `insert_a_col()` inserts single columns.

If `src` is specified, not only column definition of `src` but also data area is copied. However, if there is not enough rows, not all rows are copied.

PARAMETER

[I] `index` column `index` to be inserted
[I] `col_name` column name to be inserted
[I] `defs` `fits::table_def` structure
[I] `src` the column to be inserted
([I] : input, [O] : output)

RETURN VALUE

`table().insert_cols()` and `table().insert_a_col()` return a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, there is no enough memory to insert), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits::table_def def =
    { "TIME_SAVE","saved time", NULL,NULL, "s","", "F16.3", "1D", "" };
fits.table("EVENT").insert_a_col(1, def);
```

13.8.48 table().swap_cols()

NAME

table().swap_cols() — Swap columns

SYNOPSIS

```
fits_table &table( ... ).swap_cols( long index0, long num_cols, long index1 );
fits_table &table( ... ).swap_cols( const char *col_name0, long num_cols,
                                    const char *col_name1 );
```

DESCRIPTION

`table().swap_cols()` swaps `num_cols` number of columns beginning with `index0` or `col_name0`, to `num_cols` number of columns beginning from `index1` or `col_name1`.

If there is an overlap between two column groups, it decreases `num_cols` and do swap.

PARAMETER

[I]	`index0`	beginning column index (1)
[I]	`col_name0`	beginning column name (1)
[I]	`num_cols`	number of columns to swap
[I]	`index1`	beginning column index (2)
[I]	`col_name1`	beginning column name (2)

([I] : input, [O] : output)

RETURN VALUE

`table().swap_cols()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code swaps column 0 and column 2.

```
fits.table("EVENT").swap_cols(0, 1, 2);
```

13.8.49 table().erase_cols(), table().erase_a_col()

NAME

table().erase_a_col() — Erase column

SYNOPSIS

```
fits_table &table( ... ).erase_cols( long index, long num_cols );
fits_table &table( ... ).erase_cols( const char *col_name, long num_cols );
fits_table &table( ... ).erase_a_col( long col_index );
fits_table &table( ... ).erase_a_col( const char *col_name );
```

DESCRIPTION

`table().erase_cols()` and `table().erase_a_col()` erase `num_cols` number of columns which begins from `index` or `col_name`.

PARAMETER

[I]	`index`	begging column `index` to erase
[I]	`col_name`	beginning column name to erase
[I]	`num_cols`	number of columns to erase

([I] : input, [O] : output)

RETURN VALUE

`table().erase_cols()` and `table().erase_a_col()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").erase_cols(0L, 1);
```

13.8.50 table().copy()

NAME

table().copy() — Copy column to other object

SYNOPSIS

```
void table( ... ).copy( fits_table *dest ) const;
void table( ... ).copy( long idx_begin, long num_rows, fits_table *dest ) const;
```

DESCRIPTION

`table().copy()` copies `num_rows` number of rows which begins from `idx_begin`, to `dest` object.

If no index is given, it copies all rows.

This API is used to make temporary buffer which is given to import_rows() (13.8.58)

PARAMETER

[I]	`idx_begin`	index of row with which it begins to copy
[I]	`num_rows`	number of rows.
[O]	`dest`	copy destination

([I] : input, [O] : output)

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits_table tmp_buf;
fits.table("EVENT").copy(0L,40, &tmp_buf);
```

13.8.51 table().resize_rows()

NAME

table().resize_rows() — Modify the number of rows in the table

SYNOPSIS

```
fits_table &table( ... ).resize_rows( long num_rows );
```

DESCRIPTION

`table().resize_rows()` modifies the number of rows in table to `num_rows`.

PARAMETER

[I] `num_rows` number of rows
([I] : input, [O] : output)

RETURN VALUE

`table().resize_rows()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, space allocation failure when resizing), it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").resize_rows(100);
```

13.8.52 table().append_rows(), table().append_a_row()

NAME

table().append_rows(), table().append_a_row() — Append row to table

SYNOPSIS

```
fits_table &table( ... ).append_rows( long num_rows );
fits_table &table( ... ).append_a_row();
```

DESCRIPTION

`table().append_rows()` appends `num_rows` number of new rows to the end of table.

`table().append_a_row()` appends new single row to the end of table.

The new value of the appended row is 0 if the column type is integer or real number, `'\0'` if column type is boolean, `' '` if column type is string.

PARAMETER

[I] `num_rows` number of rows to append
([I] : input, [O] : output)

RETURN VALUE

`table().append rows()` and `table().append_a_row()` return a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").append_rows(20);
```

13.8.53 table().insert_rows(), table().insert_a_row()

NAME

table().insert_rows(), table().insert_a_row() — Insert row to table

SYNOPSIS

```
fits_table &table( ... ).insert_rows( long index, long num_rows );
fits_table &table( ... ).insert_a_row( long index );
```

DESCRIPTION

`table().insert_rows()` inserts `num_rows` number of new rows into the `index`-th row.

`table().insert_a_row()` inserts new single row into the `index`-th row.

The new value of the inserted row is 0 if the column type is integer or real number, `'\0'` if column type is boolean, `' '` if column type is string.

PARAMETER

[I]	`index`	`index` which the rows are inserted at
[I]	`num_rows`	number of rows to be inserted

([I] : input, [O] : output)

RETURN VALUE

`table().insert_rows()` and `table().insert_a_row()` return a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following code inserts 5 new rows in back of 10th row.

```
fits.table("EVENT").insert_rows(10, 5);
```

13.8.54 table().erase_rows(), table().erase_a_row()

NAME

table().erase_rows(), table().erase_a_row() — Erase row of table

SYNOPSIS

```
fits_table &table( ... ).erase_rows( long index, long num_rows );
fits_table &table( ... ).erase_a_row( long index );
```

DESCRIPTION

`table().erase_rows()` erases `num_rows` number of rows starting from `index`-th row.

`table().erase_a_row()` erases `index`-th row.

PARAMETER

[I]	`index`	row `index` to be erased
[I]	`num_rows`	number of rows to be erased

([I] : input, [O] : output)

RETURN VALUE

`table().erase_rows()` and `table().erase_a_row()` return a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

The following codes erases 5 rows which begins from 10th row.

```
fits.table("EVENT").erase_rows(10, 5);
```

13.8.55 table().clean_rows()

NAME

table().clean_rows() — Initialize row of table

SYNOPSIS

```
fits_table &table( ... ).clean_rows();
fits_table &table( ... ).clean_rows( long index, long num_rows );
```

DESCRIPTION

`table().clean_rows()` initializes all the column of `num_rows` numbers of row starting from `index`-th row. If no argument is given, all the rows are initialized.

The initial value is 0 for integer and real number type column, `'\0'` for boolean type column, and `' '` for string type column.

PARAMETER

[I] `index` the row `index` to be initialized
[I] `num_rows` number of rows to be initialized
([I] : input, [O] : output)

RETURN VALUE

`table().clean_rows()` returns a reference to the fits_table object.

EXAMPLES

The following codes initiates 5 rows from 10th row.

```
fits.table("EVENT").clean_rows(10, 5);
```

13.8.56 table().move_rows()

NAME

table().move_rows() — Copy rows

SYNOPSIS

```
fits_table &table( ... ).move_rows( long src_index, long num_rows, long dest_index );
```

DESCRIPTION

`table().move_rows()` copies the `num_rows` number of rows starting from `src_index` into the `dest_index`.

PARAMETER

[I] `src_index` row index of source
[I] `num_rows` number of rows to be copied
[I] `dest_index` row index of destination
([I] : input, [O] : output)

RETURN VALUE

`table().move_rows()` returns a reference to the fits_table object.

EXAMPLES

The following code copies 10th row into 11th row.

```
fits.table("EVENT").move_rows(10, 1, 11);
```

13.8.57 table().swap_rows()

NAME

table().swap_rows() — Swap rows

SYNOPSIS

```
fits_table &table( ... ).swap_rows( long index0, long num_rows, long index1 );
```

DESCRIPTION

`table().swap_rows()` swaps the `num_rows` number of rows starting from `index0` with `index1`.

If there is an overlap between two rows, it decreases `num_cols` and do swap.

PARAMETER

[I]	`index0`	the source row index to be swapped
[I]	`num_rows`	number of rows to be swapped
[I]	`index1`	the destination row index to be swapped

([I] : input, [O] : output)

RETURN VALUE

`table().swap rows()` returns a reference to the fits_table object.

EXAMPLES

The following code swaps 10th row with 11th row.

```
fits.table("EVENT").swap_rows(10, 1, 11);
```

13.8.58 table().import_rows()

NAME

table().import_rows() — Import table

SYNOPSIS

```
fits_table &table( ... ).import_rows( long dest_index, bool match_by_name,
                                      const fits_table &from,
                                      long idx_begin = 0,
                                      long num_rows = FITS::ALL );
```

DESCRIPTION

`table().import_rows()` imports `num_rows` number of rows starting `from` `idx_begin` of table object '`from`' into `num_rows` number of rows specified by `dest_index`. All the columns are imported.

How to allocate each of columns on '`from`' to the object is decided by `match_by_name`. When `match_by_name` is `true`, it searches the column of which the names are identical and imports the column. When `match_by_name` is `false`, it imports `from` the 0th column in order.

The column type of '`from`' table and column type of the object does not need to be identical. If two column types are not identical, it converts the type and imports.

PARAMETER

[I]	`dest_index`	destination row index to be imported
[I]	`match_by_name`	the flag whether it matches by column name
[I]	`from`	source table object to import `from`
[I]	`idx_begin`	source row index to be imported
[I]	`num_rows`	number of rows

([I] : input, [O] : output)

RETURN VALUE

`table().import_rows()` returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").import_rows( 0, false, fits.table("EVENT_SAVE") );
```

13.8.59 table().col().move()

NAME

table().col().move() — Copy row into row in particular column

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                            .move( long src_index, long num_rows, long dest_index );
```

DESCRIPTION

In a particular column, `table().col().move()` copies `num_rows` number of rows starting from `src_index` into `dest_index`.

PARAMETER

[I]	`src_index`	source index
[I]	`num_rows`	number of columns
[I]	`dest_index`	destination index

([I] : input, [O] : output)

RETURN VALUE

`table().col().move()` returns a reference to the fits_table_col object.

EXAMPLES

The following code copies 0th row to 2nd row at column 0.

```
fits.table("EVENT").col(0L).move( 0, 1, 2 );
```

13.8.60 table().col().swap()

NAME

table().col().swap() — Swap rows at particular column

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                            .swap( long index0, long num_rows, long index1 );
```

DESCRIPTION

As for specified column, `table().col().swap()` swap `num_rows` number of rows starting from `index0` with rows starting from `index1`.

If there is an overlap between two rows, it decreases `num_cols` and do swap.

PARAMETER

[I] `index0` source row index
[I] `num_rows` number of rows
[I] `index1` destination index
([I] : input, [O] : output)

RETURN VALUE

`table().col().swap()` returns reference to the fits_table_col object.

EXAMPLES

The following code swaps 0th row with 2nd row at column 0.

```
fits.table("EVENT").col(0L).swap( 0, 1, 2 );
```

13.8.61 table().col().clean()

NAME

table().col().clean() — Initialize value at specified column

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).clean();
fits_table_col &table( ... ).col( ... ).clean( long index, long num_rows );
```

DESCRIPTION

As for specified column, `table().col().clean()` initializes `num_rows` number of rows starting from `index`. If no argument is given, all the rows are initialized.

The initial value is 0 for integer and real number type column, `'\0'` for boolean type column, and `' '` for string type column.

PARAMETER

[I] `index` starting `index` to initialize
[I] `num_rows` number of rows
([I] : input, [O] : output)

RETURN VALUE

`table().col().clean()` returns a reference to the fits_table_col object.

EXAMPLES

```
fits.table("EVENT").col(0L).clean();
```

13.8.62 table().col().import()

NAME

table().col().import() — Import from particular column

SYNOPSIS

```
fits_table_col &table( ... ).col( ... )
                            .import( long dest_index,
                                     const fits_table_col &from,
                                     long idx_begin = 0,
                                     long num_rows = FITS::ALL );
```

DESCRIPTION

`table().col().clean()` imports `num_rows` number of rows of which the index begins with `idx_begin` on the table object '`from`' into `num_rows` number of rows of which the index begins with `dest_index`.

The column type of '`from`' table and column type of the object does not need to be identical. If two column types are not identical, it converts the type and imports.

PARAMETER

[I]	`dest_index`	destination row index to be imported
[I]	`from`	source table object to import `from`
[I]	`idx_begin`	source row index to be imported
[I]	`num_rows`	number of rows

([I] : input, [O] : output)

RETURN VALUE

`table().col().import()` returns a reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB or SFITSIO (`sli::err_rec` exception).

EXAMPLES

```
fits.table("EVENT").col(0L).import( 0, fits.table("EVENT_SAVE").col(0L) );
```

13.8.63 table().col().assign_default()

NAME

table().col().assign_default() — Specify a value to be set for new cells when resizing rows

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_default( double value );
fits_table_col &table( ... ).col( ... ).assign_default( float value );
fits_table_col &table( ... ).col( ... ).assign_default( long long value );
fits_table_col &table( ... ).col( ... ).assign_default( long value );
fits_table_col &table( ... ).col( ... ).assign_default( int value );
fits_table_col &table( ... ).col( ... ).assign_default( const char *value );
fits_table_col &table( ... ).col( ... )
                            .assign_default_value( const void *value_ptr );
```

DESCRIPTION

Using these member functions, programmers can specify a value to be set for new cells created by `table().resize_rows()`, etc.

`.assign_default()` is a high-level API, and the value of argument is converted into the appropriate value which is reflected by the values of `TZERO`, `TSCALE` and `TNULL` in the header. Set `NAN` to the argument to specify NULL value.

`.assign_default_value()` is a low-level API, and values of `TZERO`, etc. are not referred. Programmers should set an address of a value whose type is that of image of current object.

PARAMETER

[I] `value` a value to be set for new cells
[I] `value_ptr` an address of a value to be set for new cells
([I] : input, [O] : output)

RETURN VALUE

These member functions returns a reference to the modified fits_table_col object.

EXCEPTION

If the API fails to allocate internal memory, it throws an exception derived from SLLIB(`sli::err_rec` exception).

EXAMPLES

Next code specifies NULL value to be set for new cells in first column, and resizes the length of rows.

```
fits.table("EVENT").col(0L).assign_default(NAN);
fits.table("EVENT").resize_rows(100);
```

13.9 Lower level manipulation of Ascii Table HDU and Binary Table

In this section, we describe lower level API to manipulate ASCII Table HDU and Binary Table HDU. As for lower level API, converting with `TZERO`n, `TSCAL`n of header must be done by user's hand. Usually, the APIs in this section is not necessary, but maybe useful for the performance tuning.

13.9.1 table().col().data_array_cs()

NAME

table().col().data_array_cs() — Reference to data buffer management object

SYNOPSIS

```
const sli::mdarray &table( ... ).col( ... ).data_array_cs() const;
```

DESCRIPTION

The image buffer of fits_table_col class is managed by mdarray class of SLLIB. data_array_cs function is used when user wants to calculate with mdarray class.

For the detail of mdarray class, See SLLIB manual.

13.9.2 table().col().data_ptr()

NAME

table().col().data_ptr() — Address of the data buffer inside the object

SYNOPSIS

```
void *table( ... ).col( ... ).data_ptr();
```

DESCRIPTION

`table().col().data_ptr()` returns the address of the internal table data buffer.

The returned value is the address of internal buffer of the object, so it will be invalid when the object was destroyed or the type or size was changed.

Use the returned address to be cast to a pointer type, corresponding to the current column type, which is choosed from `fits::double_t *`, `fits::float_t *`, `fits::longlong_t *`, `fits::long_t *`, `fits::short_t *`, `fits::byte_t *` or `fits::logical_t *`.

RETURN VALUE

`table().col().data ptr()` returns an address of the internal table data buffer.

EXAMPLES

```
fits::double_t *tbl_data_ptr
    = (fits::double_t *)fits.table("EVENT").col(0L).data_ptr();
        :
        :
```

13.9.3 table().col().get_data()

NAME

table().col().get_data() — Copy data to external buffer

SYNOPSIS

```
ssize_t *table( ... ).col( ... )
                     .get_data( void *dest_buf, size_t buf_size ) const;
ssize_t *table( ... ).col( ... )
                     .get_data( long row_idx,
                                void *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`table().col().get_data()` copies the raw column data from the `row_idx`, maximum of `buf_size` byte, to `dest_buf`.

Use the address specified with `dest_buf` to be cast to a pointer type, corresponding to the current column type, which is choosed from `fits::double_t *`, `fits::float_t *`, `fits::longlong_t *`, `fits::long_t *`, `fits::short_t *`, `fits::byte_t *` or `fits::logical_t *`.

PARAMETER

[O]	`dest_buf`	the address of destination buffer
[I]	`buf_size`	size of `dest_buf`
[I]	`row_idx`	the starting row index to get data

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	byte length which is able to copy when the buffer length is sufficient.
Negative value (error)	:	the case when copy was not done because of invalid argument.

EXAMPLES

```
fits_table_col &col_ref = fits.table("EVENT").col(0L);
size_t buf_size = col_ref.elem_byte_length() * col_ref.length();
char *dest_buf = (char *)malloc(buf_size);
if ( dest_buf == NULL ) {
```

```
    /* Error Handling */
}
col_ref.get_data(dest_buf, buf_size);
```

13.9.4 table().col().put_data()

NAME

table().col().put_data() — Input the data from external buffer

SYNOPSIS

```
ssize_t *table( ... ).col( ... ).put_data( const void *src_buf, size_t buf_size );
ssize_t *table( ... ).col( ... )
                     .put_data( long row_idx, const void *src_buf, size_t buf_size );
```

DESCRIPTION

`table().col().put_data()` copies the raw data of `src_buf` from the `row_idx` th row with maximum of `buf_size`, to the internal buffer of the object.

PARAMETER

[I]	`src_buf`	the address of source buffer
[I]	`buf_size`	size of `src_buf`
[I]	`row_idx`	the starting row index to copy

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	byte length which can be copied when the buffer length of `src_buf` is sufficient.
Negative value (error)	:	the case when copy was not done because of invalid argument.

EXAMPLES

Following code modifies all the contents of the column 0 in the table "EVENT" in user's buffer.

```
fits_table_col &col_ref = fits.table("EVENT").col(0L);
size_t buf_size = col_ref.elem_byte_length() * col_ref.length();
char *data_buf = (char *)malloc(buf_size);
    :
    :
col_ref.put_data(data_buf, buf_size);
```

13.9.5 table().heap_ptr()

NAME

table().heap_ptr() — Address of the heap buffer in the object

SYNOPSIS

```
void *table( ... ).heap_ptr();
```

DESCRIPTION

`table().heap_ptr()` returns the address of the internal table heap buffer for variable length array.

The data in heap buffer are stored with big-endian, and the alignment of the data is not defined. Therefore, copying byte data from heap area in objects into programmer's buffer and adjustment of endianness is required when reading variable length array of a row. When writing data into heap buffer, the opposite procedure is needed.

The returned value is the address of internal buffer of the object, so it will be invalid when the object was destroyed or the size was changed.

Using table().get_heap() and table().put_heap() is recommended for general purposes. See also §13.9.6 and §13.9.7.

13.9.6 table().get_heap()

NAME

table().get_heap() — Copy heap data to external buffer

SYNOPSIS

```
ssize_t *table( ... ).get_heap( void *dest_buf, size_t buf_size ) const;
ssize_t *table( ... ).get_heap( long offset,
                                void *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`table().get_heap()` copies the data in the heap buffer (for variable length array) from the `offset`, maximum of `buf_size` byte, to `dest_buf`.

The data in the heap buffer are stored with big-endian, therefore, programmer's buffer pointed by `dest_buf` can be read after endianness adjustment.

PARAMETER

[O]	`dest_buf`	the address of destination buffer
[I]	`buf_size`	size of `dest_buf`
[I]	`offset`	the starting position (byte offset) in heap to get data

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	byte length which is able to copy when the buffer length is sufficient.
Negative value (error)	:	the case when copy was not done because of invalid argument.

EXAMPLES

See `test/access_bte_heap.cc` in SFITSIO source package.

13.9.7 table().put_heap()

NAME

table().put_heap() — Input the heap data from external buffer

SYNOPSIS

```
ssize_t *table( ... ).put_heap( const void *src_buf, size_t buf_size );
ssize_t *table( ... ).put_heap( long offset,
                                const void *src_buf, size_t buf_size );
```

DESCRIPTION

`table().put_heap()` copies the data of `src_buf` with maximum of `buf_size`, to the internal heap buffer of the object at the position (byte offset) specified by `offset`.

The data in `src_buf` have to become big-endian before using `.put_heap()`.

PARAMETER

[I] `src_buf` the address of source buffer
[I] `buf_size` size of `src_buf`
[I] `offset` the starting position (byte offset) in heap to copy
([I] : input, [O] : output)

RETURN VALUE

Non-negative value : byte length which can be copied when the buffer length of `src_buf` is sufficient.
Negative value (error) : the case when copy was not done because of invalid argument.

EXAMPLES

See `sample/create_vl_array.cc` in SFITSIO source package.

13.9.8 table().resize_heap()

NAME

table().resize_heap() — Update size of heap area

SYNOPSIS

```
fits_table &table( ... ).resize_heap( size_t sz );
```

DESCRIPTION

This member function updates size of the heap buffer in the object. Set argument `sz` in bytes.

RETURN VALUE

This member functions returns a reference to the fits_table object.

EXCEPTION

If the API fails to manipulate internal buffer, it throws an exception derived from SLLIB (`sli::err_rec`)

EXAMPLES

See `sample/create_vl_array.cc` in SFITSIO source package.

13.9.9 table().reverse_heap_endian()

NAME

table().reverse_heap_endian() — Reverse endian of specified data in heap area

SYNOPSIS

```
fits_table &table( ... ).reverse_heap_endian( long offset,
                                              int type, long length );
```

DESCRIPTION

This member function reverses the endian of an array that begins at `offset` (0-indexed, in bytes) in heap area, **when endian conversion is required**. Data type and length of it are specified by arguments `type` and `length`.

Argument `type` can take `FITS::SHORT_T`, `FITS::LONG_T`, `FITS::LONGLONG_T`, `FITS::FLOAT_T`, `FITS::DOUBLE_T`, `FITS::COMPLEX_T` or `FITS::DOUBLECOMPLEX_T`.

Note that this member function performs the endian conversion only for little-endian machines.

PARAMETER

[I] `offset` the starting position (byte offset) in heap buffer
[I] `type` type of data elements
[I] `length` number of data elements
([I] : input, [O] : output)

RETURN VALUE

This member functions returns a reference to the fits_table object.

13.9.10 table().reserved_area_length()

NAME

table().reserved_area_length() — Length of reserved area in data unit

SYNOPSIS

```
long long table( ... ).reserved_area_length() const;
```

DESCRIPTION

This member function returns byte length of reserved area in the data unit of the binary table HDU.

See §3.5 for details of reserved area.

13.9.11 table().resize_reserved_area()

NAME

table().resize_reserved_area() — Change byte length of reserved area

SYNOPSIS

```
fits_table &table( ... ).resize_reserved_area( long long sz );
```

DESCRIPTION

This member function changes length of reserved area in the data unit of the binary table HDU. Set the length to `sz` in bytes.

See §3.5 for details of reserved area.

RETURN VALUE

This member functions returns a reference to the fits_table object.

13.9.12 table().col().short_value()

NAME

table().col().short_value() — Return the raw value of a cell as integer (short type)

SYNOPSIS

```
short table( ... ).col( ... ).short_value( long row_index ) const;
short table( ... ).col( ... ).short_value( long row_index,
                           const char *elem_name, long repetition_idx = 0 ) const;
short table( ... ).col( ... ).short_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().short_value()` returns the raw value of a cell as integer (short type). This function can access the value fastest when the column type is `FITS::SHORT_T` (includes `'I'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is real number, it returns the rounded value.

When type column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns the rounded value.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`.

For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().short_value()` returns the cell value.

EXAMPLES

```
short value = fits.table("EVENT").col(0L).short_value(0);
        :
        :
```

13.9.13 table().col().long_value()

NAME

table().col().long_value() — Returns the raw cell value as integer (long type)

SYNOPSIS

```
long table( ... ).col( ... ).long_value( long row_index ) const;
long table( ... ).col( ... ).long_value( long row_index,
                          const char *elem_name, long repetition_idx = 0 ) const;
long table( ... ).col( ... ).long_value( long row_index,
                             long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().long_value()` returns the raw value of a cell as integer (long type). This function can access the value fastest when the column type is `FITS::LONG_T` (includes `'J'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is real value, it returns the rounded value.

When the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns the rounded value.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().long_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.14 table().col().longlong_value()

NAME

table().col().longlong_value() — Returns the raw value of a cell as integer (long long type)

SYNOPSIS

```
long long table( ... ).col( ... ).longlong_value( long row_index ) const;
long long table( ... ).col( ... ).longlong_value( long row_index,
                            const char *elem_name, long repetition_idx = 0 ) const;
long long table( ... ).col( ... ).longlong_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().longlong_value()` returns the raw value of a cell as integer (long long type). This function can access the value fastest when the column type is `FITS::LONGLONG_T` (includes `'K'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is real value, it returns the rounded value.

When the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns the rounded value.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().longlong_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.15 table().col().byte_value()

NAME

table().col().byte_value() — Returns raw cell value as integer (byte type)

SYNOPSIS

```
unsigned char table( ... ).col( ... ).byte_value( long row_index ) const;
unsigned char table( ... ).col( ... ).byte_value( long row_index,
                            const char *elem_name, long repetition_idx = 0 ) const;
unsigned char table( ... ).col( ... ).byte_value( long row_index,
                                 long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().byte_value()` returns the raw value of a cell as integer (byte type). This function can access the value fastest when the column type is `FITS::BYTE_T` (includes `'B'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is real value, it returns the rounded value.

When the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns the rounded value.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().byte_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.16 table().col().float_value()

NAME

table().col().float_value() — Returns the raw cell value as real number (float type)

SYNOPSIS

```
float table( ... ).col( ... ).float_value( long row_index ) const;
float table( ... ).col( ... ).float_value( long row_index,
                           const char *elem_name, long repetition_idx = 0 ) const;
float table( ... ).col( ... ).float_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().float_value()` returns the raw cell value as real number (float type). This function can access the value fastest when the column type is `FITS::FLOAT_T` (includes `'E'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of TDIMn)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().float_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.17 table().col().double_value()

NAME

table().col().double_value() — Returns the raw cell value as real number (double type)

SYNOPSIS

```
double table( ... ).col( ... ).double_value( long row_index ) const;
double table( ... ).col( ... ).double_value( long row_index,
                           const char *elem_name, long repetition_idx = 0 ) const;
double table( ... ).col( ... ).double_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().double_value()` returns the raw cell value as real number (double type). This function can access the value fastest when the column type is `FITS::DOUBLE_T` (includes `'D'` in `TFORM`*n*). However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

If the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().double_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.18 table().col().bit_value()

NAME

table().col().bit_value() — Returns raw cell value as integer (bit type)

SYNOPSIS

```
long table( ... ).col( ... ).bit_value( long row_index ) const;
long table( ... ).col( ... ).bit_value( long row_index,
            const char *elem_name, long repetition_idx = 0, int nbit = 0 ) const;
long table( ... ).col( ... ).bit_value( long row_index,
                long elem_index, long repetition_idx = 0, int nbit = 1 ) const;
```

DESCRIPTION

`table().col().bit_value()` returns the raw cell value as integer (bit type). This function can access the value fastest when the column type is `FITS::BIT_T` (includes `'X'` in `TFORM`*n*).

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

Argument `nbit` indicates how many bits to the right from specified element. If `nbit` is zero, bit-field description specified with `TELEM`*n*, which is counted from specified element to the right, is used. See also §11.10 for bit-field description.

If the column type is real number, it returns the rounded value. However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

When the column type is boolean, it returns 1 if the value is `'T'`, and 0 otherwise.

If the column type is string, it converts the value to real number, and returns the rounded value.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().bit_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.19 table().col().logical_value()

NAME

table().col().logical_value() — Return row cell value as boolean

SYNOPSIS

```
int table( ... ).col( ... ).logical_value( long row_index ) const;
int table( ... ).col( ... ).logical_value( long row_index,
                           const char *elem_name, long repetition_idx = 0 ) const;
int table( ... ).col( ... ).logical_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().logical_value()` returns raw call value as boolean. The returned value is `'T'`, `'F'`, or `'U'`. This function can access the value fastest when the column type is `FITS::LOGICAL_T` (includes `'L'` in `TFORM`*n*).

When the type of column is boolean, it returns `'T'` if the value is `'T'`, `'F'` if the value is `'F'`, and `'U'` otherwise.

If type column type is string, it converts string to real number, rounds it, and if the result is 0 then it returns `'F'` , and `'T'` otherwise. When the string cannot be converted to real number, if the string begins with `'T'` or `'t'` then it returns `'T'`, if it begins with `'F'` or `'f'` then it returns `'F'`, and `'U'` otherwise.

When the column type is real number, it rounds the value, and if the value is 0 then it returns `'F'` and `'T'` otherwise. When the column type is integer, if the value is 0 then it returns `'F'`, and `'T'` otherwise. However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().logical_value()` returns the cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.20 table().col().string_value()

NAME

table().col().string_value() — Returns raw cell value as string

SYNOPSIS

```
const char *table( ... ).col( ... ).string_value( long row_index ) const;
const char *table( ... ).col( ... ).string_value( long row_index,
                           const char *elem_name, long repetition_idx = 0 ) const;
const char *table( ... ).col( ... ).string_value( long row_index,
                               long elem_index, long repetition_idx = 0 ) const;
```

DESCRIPTION

`table().col().string_value()` returns the raw cell value as string. This function can access the value fastest when the column type is `FITS::ASCII_T` (includes `'A'` in `TFORM`*n*).

If the column type is string, it returns the raw string.

If the column type is boolean, it returns whether `"T"`, `"F"`, or `"U"`.

If the column type is integer, it returns the `"%lld"` formatted string of libc's printf. If the column type is real number, it returns the `"%.15G"` formatted string of libc's printf. However, the value returned by these functions **does not reflect** `TZERO`*n* or `TSCAL`*n*.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().string_value()` returns an address of the string of cell value.

EXAMPLES

See EXAMPLES of 13.9.12

13.9.21 table().col().array_heap_offset()

NAME

table().col().array_heap_offset() — Returns address of byte data in the heap area

SYNOPSIS

```
long table( ... ).col( ... ).array_heap_offset( long row_index,
                                                long elem_index = 0 ) const;
```

DESCRIPTION

On specified column having variable length array(s), this member function returns an address of the byte data on the heap area referred by specified row.

When a row refers multiple variable length arrays, set `elem_index` to select one from them.

The index begins with zero.

Use `.array_length()` to obtain length of variable length array(s). See §13.8.18 for it.

PARAMETER

[I] `row_index` row index
[I] `elem_index` index of array to be selected (can be omitted)
([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	0-indexed address (in bytes) on the heap area.
Negative value (error)	:	the case when invalid arguments are set, specified column does not refer heap area, etc.

EXAMPLES

See `test/access_bte_heap.cc` in SFITSIO source package.

13.9.22 table().col().get_string_value()

NAME

table().col().get_string_value() — Gets the raw cell value as string

SYNOPSIS

```
ssize_t table( ... ).col( ... ).get_string_value( long row_index,
                                        char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_string_value( long row_index,
                                        const char *elem_name,
                                        char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_string_value( long row_index,
                                    const char *elem_name, long repetition_idx,
                                    char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_string_value( long row_index,
                                        long elem_index,
                                        char *dest_buf, size_t buf_size ) const;
ssize_t table( ... ).col( ... ).get_string_value( long row_index,
                                        long elem_index, long repetition_idx,
                                        char *dest_buf, size_t buf_size ) const;
```

DESCRIPTION

`table().col().get_string_value()` converts raw cell value to string and stores it to `dest_buf`. The buffer size (in bytes) is specified with `buf_size`.

If the column type is string then raw string is stored.

If type column type is boolean and no TDISPn is given, it returns "T", "F", or "U".

If the column type is integer, it stores the "%lld" formatted string of libc's printf. If the column type is real number, it stores the "%.15G" formatted string of libc's printf. However, the value stored by these functions **does not reflect** TZEROn or TSCALn.

To specify row, use row_index. To specify element, use elem_name or elem_index. For elem_name, a name which exists in TELEMn can be specified.

If TDIMn is specified, elem_index can be specidied as the first dimensional index, and repetition_idx can be specified as the second dimensional index. The index begins with zero.

PARAMETER

[I]	row_index	row index
[I]	elem_name	element name
[I]	elem_index	element index (the first dimension index of TDIMn)
[I]	repetiton_index	second dimensional index
[O]	dest_buf	destination buffer
[I]	buf_size	size of dest_buf

([I] : input, [O] : output)

RETURN VALUE

Non-negative value	:	number of characters which can be copied when the buffer length is sufficient (excluding '\0').
Negative value (error)	:	the case when copy was not done because of invalid argument.

EXCEPTION

If the API fails to allocate internal memory area, it throws an exception of SFITSIO (sli::err_rec)

EXAMPLES

```
char buf[128];
fits.table("EVENT").col(0L).get_string_value( 0, buf, sizeof(buf) );
```

13.9.23 table().col().assign_short()

NAME

table().col().assign_short() — Assign value to the cell directly as integer (short type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_short( short value,
                                                      long row_index );
fits_table_col &table( ... ).col( ... ).assign_short( short value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_short( short value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

table().col().assign_short() assigns value to the cell directly as integer (short type). This function can access the value fastest when the column type is FITS::SHORT_T (includes 'I' in TFORMn). However, these functions **does not convert** the value by TZEROn and TSCALn of header.

When the column type is boolean, it stores 'F' if the value is 0, and 'T' otherwise.

If the column type is string, it formats the `value` with `"%hd"` of `printf()`, and stores it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_short()` returns reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

```
short value = 0;
fits.table("EVENT").col(0L).assign_short(value, 0);
```

13.9.24 table().col().assign_long()

NAME

table().col().assign_long() — Assign value to the cell directly as integer (long type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_long( long value,
                                                     long row_index );
fits_table_col &table( ... ).col( ... ).assign_long( long value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_long( long value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_long()` assigns `value` to the cell directly as integer (long type). This function can access the `value` fastest when the column type is `FITS::LONG_T` (includes `'J'` in `TFORM`*n*). However, these functions **does not convert** the `value` by `TZERO`*n* and `TSCAL`*n* of header.

When the column type is boolean, it stores `'F'` if the `value` is 0, and `'T'` otherwise.

If the column type is string, it stores the `"%ld"` formatted string of `printf()`.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_long()` returns reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.25 table().col().assign_longlong()

NAME

table().col().assign_longlong() — Assign value to the cell directly as integer (long long type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_longlong( long long value,
                                                         long row_index );
fits_table_col &table( ... ).col( ... ).assign_longlong( long long value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_longlong( long long value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_longlong()` assigns `value` to the cell directly as integer (long log type). This function can access the `value` fastest when the column type is `FITS::LONGLONG_T` (includes `'K'` in `TFORM`*n*). However, these functions **does not convert** the `value` by `TZERO`*n* and `TSCAL`*n* of header.

When the column type is boolean, it stores `'F'` if the `value` is 0, and `'T'` otherwise.

If the column type is string, it stores the `"%lld"` formatted string of `printf()`.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	the `value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_longlong()` returns reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.26 table().col().assign_byte()

NAME

table().col().assign_byte() — Assign value to the cell directly as integer (byte type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_byte( unsigned char value,
                                                     long row_index );
fits_table_col &table( ... ).col( ... ).assign_byte( unsigned char value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_byte( unsigned char value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_byte()` assigns `value` to the cell directly as integer (byte type). This function can access the `value` fastest when the column type is `FITS::BYTE_T` (includes `'B'` in `TFORM`*n*). However, these functions **does not convert** the `value` by `TZERO`*n* and `TSCAL`*n* of header.

When the column type is boolean, it stores `'F'` if the `value` is 0, and `'T'` otherwise.

If the column type is string, it formats the `value` with `"%hhu"` of `printf()`, and stores it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to store
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_byte()` returns reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.27 table().col().assign_float()

NAME

table().col().assign_float() — Assign value to the cell directly as real number (float type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_float( float value,
                                                      long row_index );
fits_table_col &table( ... ).col( ... ).assign_float( float value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_float( float value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_float()` assigns `value` to the cell directly as real number (float type). This function can access the `value` fastest when the column type is `FITS::FLOAT_T` (includes `'E'` in `TFORM`n). However, these functions **does not convert** the `value` by `TZERO`n and `TSCAL`n of header.

If the column type is integer, it stores the rounded `value`.

When the column type is boolean, it rounds the `value`, and if the `value` is 0 then stores `'F'`, and `'T'` otherwise.

If the column type is string, it formats with `"%.15G"` of `printf()` and stores it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`n can be specified.

If `TDIM`n is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`n)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_float()` returns reference to the fits_table_col object.

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.28 table().col().assign_double()

NAME

table().col().assign_double() — Assign value to the cell directly as real number (double type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_double( double value,
                                                       long row_index );
fits_table_col &table( ... ).col( ... ).assign_double( double value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_double( double value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_double()` assigns `value` to the cell directly as real number (double type). This function can access the `value` fastest when the column type is `FITS::DOUBLE_T` (includes `'D'` in `TFORM`*n*). However, these functions **does not convert** the `value` by `TZERO`*n* and `TSCAL`*n* of header.

If the column type is integer, it stores the rounded `value`.

When the column type is boolean, it rounds the `value`, and if the `value` is 0 then stores `'F'`, and `'T'` otherwise.

If the column type is string, it formats with `"%.15G"` of `printf()` and stores it.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	the `value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_double()` returns reference to the fits_table_col object

EXCEPTION

If the API fails to manipulate internal buffer (for exapmle, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.29 table().col().assign_bit()

NAME

table().col().assign_bit() — Assign value to the cell directly as integer (bit type)

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_bit( long value,
                                                    long row_index );
fits_table_col &table( ... ).col( ... ).assign_bit( long value,
```

```
    long row_index, const char *elem_name, long repetition_idx = 0, nbit = 0 );
fits_table_col &table( ... ).col( ... ).assign_bit( long value,
        long row_index, long elem_index, long repetition_idx = 0, nbit = 1  );
```

DESCRIPTION

`table().col().assign_bit()` assigns `value` to the cell directly as integer (bit type). This function can access the `value` fastest when the column type is `FITS::BIT_T` (includes `'X'` in `TFORM`*n*).

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

Argument `nbit` indicates how many bits to the right from specified element. If `nbit` is zero, bit-field description specified with `TELEM`*n*, which is counted from specified element to the right, is used. See also §11.10 for bit-field description.

Though this function can be used to integer and real number type column, these functions **does not convert** with `TZERO`*n* and `TSCAL`*n* of header.

When the column type is boolean, it stores `'F'` if the `value` is 0, and `'T'` otherwise.

If the column type is string, it stores the `"%ld"` formatted string of `printf()`.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_bit()` returns reference to the fits_table_col object

EXCEPTION

If the API fails to manipulate internal buffer (for example, `value` formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

EXAMPLES

See EXAMPLES of 13.9.23

13.9.30 table().col().assign_logical()

NAME

table().col().assign_logical() — Assign value to the cell directly as boolean

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_logical( int value,
                                                        long row_index );
fits_table_col &table( ... ).col( ... ).assign_logical( int value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_logical( int value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_logical()` assigns the `value` to the cell directly as boolean. This function can access the `value` fastest when the column type is `FITS::LOGICAL_T` (includes `'L'` in `TFORM`*n*).

When the column type is boolean, it assigns `'T'` if the `value` is `'T'` and assigns `'F'` if the `value` is `'F'` and `'\0'` otherwise.

When the column type is integer or real number, it assigns 1 if the `value` is `'T'`, and 0 otherwise. However, it **does not convert** with `TZERO`*n* and `TSCAL`*n* of header.

When the column type is string, it assigns `"T"` if the `value` is `'T'`, assigns `"F"` if the `value` is `'F'`, and `"U"` otherwise.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_logical()` returns reference to the fits_table_col object

EXAMPLES

See EXAMPLES of 13.9.23

13.9.31 table().col().assign_string()

NAME

table().col().assign_string() — Assign value to the cell directly as string

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_string( const char *value,
                                                       long row_index );
fits_table_col &table( ... ).col( ... ).assign_string( const char *value,
              long row_index, const char *elem_name, long repetition_idx = 0 );
fits_table_col &table( ... ).col( ... ).assign_string( const char *value,
              long row_index, long elem_index, long repetition_idx = 0 );
```

DESCRIPTION

`table().col().assign_string()` assigns a `value` to the cell directly as a string.

As for a boolean column, when the `value` can be converted to real number, if the `value` is 0 then `'F'`, otherwise `'T'` is stored. When the `value` cannot be converted to real number, if the `value` begins with `'T'` or `'t'` then `'T'`, if it begins with `'F'` or `'f'` then `'F'`, otherwise `'\0'` is stored.

If the column type is real number, the `value` is converted to real number and then stored. If the column type is integer, the `value` is converted to real number and then rounded `value` is stored. The `value` **is not converted**, however, by using a `value` of `TZERO`*n* and `TSCAL`*n* of a header.

To specify row, use `row_index`. To specify element, use `elem_name` or `elem_index`. For `elem_name`, a name which exists in `TELEM`*n* can be specified.

If `TDIM`*n* is specified, `elem_index` can be specidied as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

PARAMETER

[I]	`value`	`value` to assign
[I]	`row_index`	row index
[I]	`elem_name`	element name
[I]	`elem_index`	element index (the first dimension index of `TDIM`*n*)
[I]	`repetiton_index`	second dimensional index

([I] : input, [O] : output)

RETURN VALUE

`table().col().assign_string()` returns reference to the fits_table_col object.

EXAMPLES

See EXAMPLES of 13.9.23

13.9.32 table().col().assign_arrdesc()

NAME

table().col().assign_arrdesc() — Assign values of array descriptor to the cell

SYNOPSIS

```
fits_table_col &table( ... ).col( ... ).assign_arrdesc( long length, long offset,
                                                         long row_index,
                                                         long elem_index = 0 );
```

DESCRIPTION

This member function assigns `length` (length of variable length array) and `offset` (address of byte data in the heap area) of variable length arrays to the cell on the main table.

Set `elem_index` to select an array, when a row has multiple variable length arrays.

The index and the address begin with zero.

PARAMETER

[I]	`length`	number of array elements
[I]	`offset`	0-indexed byte address in the heap area to be referred
[I]	`row_index`	row index
[I]	`elem_index`	index of array to be selected (can be omitted)

([I] : input, [O] : output)

RETURN VALUE

This member function returns reference to the fits_table_col object.

EXAMPLES

See `sample/create_vl_array.cc` in SFITSIO source package.

---

14 APPENDIX1: Sample Programs

SFITSIO source package contains sample programs suitable to learn programming with SFITSIO. We summarize these programs in this section.

- `sample/read_and_write.cc`

  This program simply reads data contents from a FITS file and writes them to another file.

- `sample/create_image.cc`

  This program is shown in §2.4 but creates unsigned 16-bit images.

- `sample/create_image_and_header.cc`

  This program creates a new FITS file having two Image HDUs. The code will helpful to learn editing FITS header and accessing internal image buffer of fits_image objects.

- `sample/create_bintable.cc`

  This program creates a new binary table. The code will helpful to learn handling NULL value and default values when resizing number of rows.

- `sample/create_asciitable.cc`

  This program creates an ASCII table.

- `sample/dump_table.cc`

  This program displays data contents in an ASCII table or a binary table. Note that variable length arrays are not supported in this code.

- `sample/create_vl_array.cc`

  This program creates a binary table having a variable length array. The code uses low-level APIs, therefore, only for advanced users.

- `sample_wcs/wcs_test.cc`

  This program converts pixel coordinates to sky coordinates in specified FITS file. The code uses APIs of both SFITSIO and libwcs of WCSTools.

- `tools/conv_bitpix.cc`

  A tool to convert BITPIX of image in Primary HDU.

  See also the description of `image().convert_type()` member function. (§13.6.13)

- `tools/create_from_template.c`

  A tool to create a FITS file from a FITS template written in plain text. To test this program, use template files prepared in `tools/template/` directory.

  See also §9 “Template Files”.

- `tools/fill_header_comments.cc`

  A tool to complete the comments of header records having blank comments using SFITSIO built-in comment dictionary (§15)

  See also the description of `hdu(...).header_fill_blank_comments()` member function. (§13.4.38)

- `tools/stat_pixels.cc`

  A tool to compute and print image pixel statistics similar to "imstat" task of IRAF.

  See also the description of `image(...).stat_pixels(...)` member function. (§13.6.42)

- `tools/combine_images.cc`

  A tool to combine images similar to "imcombine" task of IRAF.

  Command arguments look like that of imcombine task, however, the basic implementation is the same as that often used in IDL script: 1. multiple image data are stored into 3-D array, 2. perform combine.

  See also the description of `image(...).combine_layers(...)` member function. (§13.6.42)

- `tools/hv.cc`

  This tool rapidly displays all header records. The code uses low-level APIs, therefore, only for advanced users.

  Disk seek is applied for uncompressed FITS file on local disk. In addition, this tool does not read Data Unit when `EXTEND` is `F` even for compressed FITS files read via network. That is why hv.cc works with high-speed.

  See also §13.5.1 and and succeeding subsections.

- `tools/dataunit_md5.cc`

  This tool computes MD5 of each Data Unit of a FITS file. The code uses low-level APIs, therefore, only for advanced users.

  Users can use this program to certify an identity of data in Data Unit.

15 APPENDIX2: SFITSIO built-in comment dictionary of FITS header

SFITSIO built-in comment dictionary is expressed in associative arrays of the following code. Using some member functions of SFITSIO, appropriate comments are automatically appended for keywords found in the associative arrays.

```
/*
 * See http://heasarc.gsfc.nasa.gov/docs/fcg/standard_dict.html for FITS
 * standard keywords and comments.
 */
static asarray_tstring Fallback_comments(              /* associative array */
        /*           |MIN                                        MAX| */
        /* system keywords */
        "SIMPLE",   "conformity to FITS standard",
        "BITPIX",   "number of bits per data pixel",
        "NAXIS",    "number of data axes",
        "NAXIS#",   "length of data axis #",
        "EXTEND",   "possibility of presence of extensions",
        "XTENSION", "type of extension",
        "PCOUNT",   "number of parameters per group",
        "GCOUNT",   "number of groups",
        "EXTNAME",  "name of this HDU",
        "EXTVER",   "version of the extension",
        "EXTLEVEL", "hierarchical level of the extension",
        /*           |MIN                                        MAX| */
        /* standard of JAXA data center */
        "FMTTYPE",  "type of format in FITS file",
        "FTYPEVER", "version of FMTTYPE definition",
        "FMTVER",   "version of FMTTYPE definition",
        /* standard keywords */
        "DATE",     "date of file creation",
        "ORIGIN",   "organization responsible for the data",
        "AUTHOR",   "author of the data",
        "REFERENC", "bibliographic reference",
        "DATE-OBS", "date of the observation",
        "TELESCOP", "telescope or mission name",
        "INSTRUME", "instrument name",
        "DETECTOR", "detector name",
        "OBSERVER", "observer who acquired the data",
        "OBJECT",   "name of observed object",
        /*           |MIN                                        MAX| */
        "EPOCH",    "equinox of celestial coordinate system",
        "EQUINOX",  "equinox of celestial coordinate system",
        "TIMESYS",  "explicit time scale specification",
        /* other general keywords */
        "OBSERVAT", "observatory name",
        "CREATOR",  "data generator program",
        "PIPELINE", "data processing pipeline name",
        "FILENAME", "file name",
        "PROPOSAL", "proposal ID",
        "BAND",     "band name",
        "MJD",      "modified Julian date",
        "AIRMASS",  "air mass",
        "EXPTIME",  "exposure time",
        "WEATHER",  "weather condition",
```

```
        /* general keyword of JAXA data center */
        "CNTTYPE",  "type of data content",
        "CNTVER",   "version of data content",
        "CHECKSUM", "HDU checksum",
        "DATASUM",  "data unit checksum",
        /*          |MIN                                    MAX| */
        NULL
);
static asarray_tstring Image_comments(                  /* associative array */
        /*          |MIN                                    MAX| */
        "BZERO",    "zero point in scaling equation",
        "BSCALE",   "linear factor in scaling equation",
        "BLANK",    "value used for undefined pixels",
        "BUNIT",    "physical unit of the pixel values",
        "DATAMIN",  "minimum data value",
        "DATAMAX",  "maximum data value",
        /* WCS */
        "WCSAXES?", "number of axes for WCS",
        "CRVAL#?",  "world coordinate at reference point",
        "CRPIX#?",  "pixel coordinate at reference point",
        "CDELT#?",  "world coordinate increment at reference point",
        "CROTA#",   "coordinate system rotation angle",
        "CTYPE#?",  "type of celestial system and projection system",
        /*          |MIN                                    MAX| */
        "CUNIT#?",  "units of the coordinates along axis",
        "PC#_#?",   "matrix of rotation (#,#)",
        "CD#_#?",   "matrix of rotation and scale (#,#)",
        "WCSNAME?", "name of WCS",
        "LONPOLE?", "native longitude of celestial pole",
        "LATPOLE?", "native latitude of celestial pole",
        "EQUINOX?", "equinox of celestial coordinate system",
        "MJD-OBS",  "modified Julian date of observation",
        "CNAME#?",  "description of CTYPE definition",
        "RADESYS?", "default coordinate system",
        /*          |MIN                                    MAX| */
        NULL
);
static asarray_tstring Binary_table_comments(           /* associative array */
        /*          |MIN                                    MAX| */
        "BITPIX",   "number of bits per data element",
        "NAXIS1",   "width of table in bytes",
        "NAXIS2",   "number of rows in table",
        "PCOUNT",   "length of reserved area and heap",
        "TFIELDS",  "number of fields in each row",
        "TXFLDKWD", "extended field keywords",              /* JAXA ext. */
        "TTYPE#",   "field name",
        "TALAS#",   "aliases of field name",                /* JAXA ext. */
        "TELEM#",   "element names",                        /* JAXA ext. */
        "TUNIT#",   "physical unit",
        "TDISP#",   "display format",
        "TFORM#",   "data format",
        "TDIM#",    "dimensionality of the array",
        "TZERO#",   "zero point in scaling equation",
        "TSCAL#",   "linear factor in scaling equation",
        "TNULL#",   "value used for undefined cells",
        "THEAP",    "byte offset to heap area",
```

```
        /* CFITSIO ext. */
        "TLMIN#",   "minimum value legally allowed",
        "TLMAX#",   "maximum value legally allowed",
        "TDMIN#",   "minimum data value",
        "TDMAX#",   "maximum data value",
        /*           |MIN                                        MAX| */
        NULL
);
static asarray_tstring Ascii_table_comments(            /* associative array */
        /*           |MIN                                        MAX| */
        "BITPIX",   "number of bits per data element",
        "NAXIS1",   "width of table in bytes",
        "NAXIS2",   "number of rows in table",
        "TFIELDS",  "number of fields in each row",
        "TXFLDKWD", "extended field keywords",                  /* JAXA ext. */
        "TTYPE#",   "field name",
        "TALAS#",   "aliases of field name",                    /* JAXA ext. */
        "TUNIT#",   "physical unit",
        "TFORM#",   "display format",
        "TBCOL#",   "starting position in bytes",
        "TZERO#",   "zero point in scaling equation",
        "TSCAL#",   "linear factor in scaling equation",
        "TNULL#",   "value used for undefined cells",
        "TLMIN#",   "minimum value legally allowed",
        "TLMAX#",   "maximum value legally allowed",
        "TDMIN#",   "minimum data value",
        "TDMAX#",   "maximum data value",
        /*           |MIN                                        MAX| */
        NULL
);
```

16 APPENDIX3: How to Use Handy TSTRING Class

The "tstring" class used inside of SFITSIO keeps the usability of C language and can process strings easily like the script languages.[44] If you already imported SFITSIO, you can use it immediately with writing as follows,

```
#include <sli/tstring.h>
using namespace sli;
```

Create an Object and Substitute/Display a String

```
    tstring my_string;
    my_string = "I am a SFITSIO user!";
```

With the code above, the creation of an object and the substitution of a string into the object has done. Users do not need to care the size of buffer at a substitution or an edition of a string, because the object automatically manages its own buffer for strings internally. In addition, users do not concern the memory leak since the buffer area automatically frees when the process exited the scope.

It is also possible to substitute with using `.printf()`. In this case, it is also not necessary to care of the buffer area.

```
    my_string.printf("Today is %d/%d/%d", y, m, d);
```

Next, the content of `my_string` will be displayed with `printf()`.

```
    printf("my_string = %s\n", my_string.cstr());
```

Use `.cstr()` when the address of the string is needed as in case when giving it to the printf() function.

Edit Strings

```
    my_string = "I am ";                /* Substitution */
    my_string += "a SFITSIO ";          /* Addition */
    my_string.append("user!");          /* Addition */
```

If you want to add string into the existing string, use "`+=`" or `.append()` as the example above.

Insert `"super "` into the head of `"SFITSIO"` in the strings above.

```
    my_string.insert(7,"super ");
```

Now it makes `"I am a super SFITSIO user!"`.

Many other member functions are available such as `.replace()` to replace, `.erase()` to erase, `.chop()` and `.chomp()` having the same functions as those in Perl, `.trim()` to erase spaces in the ends, and `.toupper()` and `.tolower()` to transform between upper cases and lower cases.

Search Strings and Use Regular Expression

It is easy for member functions to access each character in the internal string and to search or replace with a regular expression. As the regular expression, POSIX-extended regular-expressions are available.

The following code displays each one character in the content of `my_string`.

[44] Although there is another way to use string in C++ standard library, tstring class has lower barrier and is simpler for the C users because this has variety of member functions.

```
size_t i;
for ( i=0 ; i < my_string.length() ; i++ ) {
    printf("ch[%d] = %c\n", i, my_string.cchr(i));
}
```

`.length()` is used to obtain the length of the strings. It is the same way as in SFITSIO.

It is also possible to read/write each one character with using "`[]`". The following example replaces every space character with an underscore.

```
size_t i;
for ( i=0 ; i < my_string.length() ; i++ ) {
    if ( my_string[i] == ' ' ) my_string[i] = '_';
}
```

Search with using regular expressions.

```
ssize_t pos;
size_t len;
pos = my_string.regmatch("[A-Z]+", &len);
```

The position at which the pattern is matched and the length of the string matching the pattern are given to `pos` and `len`, respectively.

Finally, the following code introduces the example of replacing with using a regular expression.

```
tstring my_string = "TSTRING  is     very    easy !";
my_string.regreplace("[ ]+", " ", true);
```

The code replaces more than one space characters with one space character.

Official Manual

PDF manual is available. For more information, see this link.

`http://www.ir.isas.jaxa.jp/~cyamauch/sli/sllib.pdf`

17 APPENDIX4: Convenient usage of DIGESTSTREAMIO class

`.read_stream()` and `.write_stream()` in SFITSIO support network and compressed file; this is achieved by digeststreamio class. Using digeststreamio class, user simply have to write a code in the same manner as that of fgets() in libc to connect to the network and compress/extract compressed stream. As for compression format, it supports gzip and bzip2. If SFITSIO is installed, write as follow to use digeststreamio class.

```
#include <sli/digeststreamio.h>
using namespace sli;
```

Open and Close of File

```
    int status;
    digeststreamio dsio;
    status = dsio.open("r", "http://www.jaxa.jp/");
```

To open file, use `.open()`. Give `"r"` (read) or `"w"` (write) to the first argument and path to the second argument. `http://...`, `ftp://...` or `file://...` can be given for the path. `file://` can be omitted. open() returns negative value in case of error.

To close file, use `.close()`.

```
    dsio.close();
```

Read

getstr() and getchr() correspond to fgets() and fgetc() in libc, respectively. For member function name of digeststreamio, `chr` is used when it is character and `str` is used when it is string. Use them as follows.

```
    char buf[256];
    while ( dsio.getstr(buf, 256) != NULL ) {
        printf("%s", buf);
    }
```

```
    int ch;
    while ( (ch = dsio.getchr()) != EOF ) {
        printf("%c", ch);
    }
```

`.getline()` enables reading line by line including break in each line.

```
    const char *ptr;
    while ( (ptr = dsio.getline()) != NULL ) {
        printf("%s", ptr);
    }
```

Write

To write a string, use `.putstr()`.

```
    dsio.putstr(buf);
```

`.printf()` also can be used.

```
    dsio.printf("Today is %d/%d/%d\n", y, m, d);
```

STDSTREAMIO class

SLLIB which provides tstring class and digeststreamio class and so on provides most functions of libc. Using the class in SLLIB to write routine functions such as printf() enables to write object-oriented and unified code.

For example, printf() and putchar() in libc can be replaced by member function of stdstreamio class. The following code sample displays HTML of `http://www.jaxa.jp/`.

```
#include <sli/digestststreamio.h>
#include <sli/stdststreamio.h>
#include <sli/tstring.h>
using namespace sli;

int main()
{
    tstring buf;
    digeststreamio dsin;
    stdstreamio sio;

    dsin.openf("r", "%s:/%s", "http", "/www.jaxa.jp/");
    while ( (buf = dsin.getline()) != NULL ) {
        sio.printf("%s", buf);
    }
    dsin.close();
    return 0;
}
```

Using `.openf()`, the path can be given in a variable length argument as well as printf(). Like this, a lot of member functions whose name suffix is "`f`" are available in classes in SLLIB. Since the variable length argument can be used for their member functions, code can be written briefly.

Because parent class of stdstreamio class is the same as that of digeststreamio class, the same member functions as that of digeststreamio class are available.

Official Manual

PDF manual is available. For more information, see this link.

`http://www.ir.isas.jaxa.jp/~cyamauch/sli/sllib.pdf`