

*Simple and Light Interfaces for C and C++ users*

# SFITSIO User's Reference Guide

Version 1.1.0 English Edition

## CREDITS

SOFTWARE DEVELOPMENT:

*Chisato Yamauchi*

MANUAL DOCUMENT:

*Chisato Yamauchi, Ken Ebisawa AND IMC LTD.*

MANUAL TRANSLATION:

*Space Engineering Development Co.,LTD., Sakura Academia Corporation AND Yuka Nojo*

SPECIAL THANKS:

*Daisuke Ishihara, Hajime Baba, Keiichi Matsuzaki, Yukio Yamamoto AND Tomotsugu Goto*

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>8</b>
1.1	All you need to know is C language . . . . .	8
1.2	Basic idea of SFITSIO - Interface close to scripting languages . . . . .	8
1.3	Feature of SFITSIO . . . . .	9
<b>2</b>	<b>Installation</b>	<b>9</b>
2.1	Supporting OS . . . . .	9
2.2	Building and installing SFITSIO . . . . .	9
<b>3</b>	<b>Tutorial</b>	<b>10</b>
3.1	First Incantation . . . . .	10
3.2	Read/Write files . . . . .	10
3.3	Direct access to remote FITS files through a network . . . . .	12
3.4	Read/Write with pipe-connections to commands (e.g., compression tools) . . . . .	13
3.5	Basis for accessing headers . . . . .	13
3.6	Keyword search in headers (POSIX Extended Regular Expression) . . . . .	13
3.7	Editing headers . . . . .	14
3.8	Access to image data . . . . .	14
3.9	Creating new FITS image . . . . .	15
3.10	Copy and paste of image data . . . . .	15
3.11	Type conversion and fast access on image data . . . . .	16
3.12	Collaboration with WCSLIB . . . . .	16
3.13	Access to the ASCII table and binary table . . . . .	18
3.14	Creation of binary table . . . . .	19
3.15	Creation of ASCII table . . . . .	19
3.16	Editing and importing of ASCII and binary tables . . . . .	20
3.17	Editing of HDU . . . . .	20
<b>4</b>	<b>Internal structure of the object</b>	<b>21</b>
<b>5</b>	<b>Things to know before using SFITSIO</b>	<b>22</b>
5.1	Namespace . . . . .	22
5.2	NULL and 0 . . . . .	22
5.3	Reference . . . . .	23
5.4	Try & catch . . . . .	23
<b>6</b>	<b>System and Notice of FITS Header Management in SFITSIO</b>	<b>25</b>
<b>7</b>	<b>Error Check and Version Management of FITS File by FMTTYPE and FTYPE-VER</b>	<b>26</b>
<b>8</b>	<b>Expansion of FITS Compatible with CFITSIO</b>	<b>26</b>
8.1	Long header value across multiple records . . . . .	26
8.2	Array of fixed length strings in binary table . . . . .	26
<b>9</b>	<b>SFITSIO's Original Expansion of FITS</b>	<b>26</b>
9.1	Distinction of upper and lower case in the header keyword . . . . .	26
9.2	Long keyword in the header (maximum 75 characters) . . . . .	27
9.3	Number of columns of the ASCII table or the binary table that exceed 999 . . . . .	27
9.4	Definition of an alias of the ASCII table or the binary table . . . . .	27

9.5	Definition of an element name in the column of the binary table . . . . .	27
9.6	Definition of the number of bits in the column of the binary table . . . . .	27
<b>10</b>	<b>Unsupported FITS Standard, etc.</b>	<b>29</b>
<b>11</b>	<b>Reference</b>	<b>30</b>
11.1	Constants . . . . .	30
11.2	Types . . . . .	30
11.3	Operation of whole FITS . . . . .	32
11.3.1	read_stream() . . . . .	32
11.3.2	hdus_to_read().assign(), cols_to_read().assign() . . . . .	33
11.3.3	write_stream() . . . . .	34
11.3.4	access_stream() . . . . .	35
11.3.5	stream_length() . . . . .	36
11.3.6	length() . . . . .	36
11.3.7	fmttype() . . . . .	37
11.3.8	ftypever() . . . . .	37
11.3.9	hduname(), extname() . . . . .	37
11.3.10	hduver(), extver() . . . . .	38
11.3.11	hdutype(), exttype() . . . . .	38
11.3.12	index() . . . . .	39
11.3.13	init() . . . . .	40
11.3.14	append_image() . . . . .	40
11.3.15	append_table() . . . . .	41
11.3.16	insert_image() . . . . .	43
11.3.17	insert_table() . . . . .	44
11.3.18	erase() . . . . .	45
11.3.19	assign_fmttype() . . . . .	46
11.3.20	assign_ftypever() . . . . .	46
11.3.21	assign_hduname(), assign_extname() . . . . .	47
11.3.22	assign_hduver(), assign_extver() . . . . .	47
11.3.23	hduver_is_set(), extver_is_set() . . . . .	48
11.4	Operation of user header . . . . .	49
11.4.1	hdu().header_length() . . . . .	49
11.4.2	hdu().header_index() . . . . .	49
11.4.3	hdu().header_regmatch() . . . . .	50
11.4.4	hdu().header().svalue() . . . . .	51
11.4.5	hdu().header().get_svalue() . . . . .	51
11.4.6	hdu().header().dvalue() . . . . .	52
11.4.7	hdu().header().lvalue(), hdu().header().llvalue() . . . . .	52
11.4.8	hdu().header().bvalue() . . . . .	53
11.4.9	hdu().header().assign(), hdu().header().assignf() . . . . .	53
11.4.10	hdu().header().assign() . . . . .	55
11.4.11	hdu().header().assign() . . . . .	56
11.4.12	hdu().header().assign() . . . . .	57
11.4.13	hdu().header().type() . . . . .	57
11.4.14	hdu().header().status() . . . . .	58
11.4.15	hdu().header().keyword() . . . . .	59
11.4.16	hdu().header().get_keyword() . . . . .	59
11.4.17	hdu().header().value() . . . . .	60

11.4.18 hdu().header().get_value()	60
11.4.19 hdu().header().comment()	61
11.4.20 hdu().header().get_comment()	61
11.4.21 hdu().header().assign_value()	62
11.4.22 hdu().header().assign_comment()	62
11.4.23 hdu().header_update()	63
11.4.24 hdu().header_assign()	64
11.4.25 hdu().header_init()	65
11.4.26 hdu().header_swap()	65
11.4.27 hdu().header_append_records()	66
11.4.28 hdu().header_append()	67
11.4.29 hdu().header_insert_records()	67
11.4.30 hdu().header_insert()	68
11.4.31 hdu().header_erase_records()	69
11.4.32 hdu().header_erase()	70
11.4.33 hdu().header()	70
11.4.34 hdu().header_formatted_string()	71
11.5 Manipulation of a System Header	72
11.5.1 hdu().sysheader_length()	72
11.5.2 hdu().sysheader_index()	72
11.5.3 hdu().sysheader_keyword()	73
11.5.4 hdu().sysheader_value()	73
11.5.5 hdu().sysheader_comment()	74
11.5.6 hdu().sysheader_formatted_string()	74
11.6 APIs for manipulation of an Image HDU	76
11.6.1 image().hduname(), image().assign_hduname()	76
11.6.2 image().hduver(), image().assign_hduver()	76
11.6.3 image().dim_length()	77
11.6.4 image().length()	77
11.6.5 image().type()	78
11.6.6 image().bytes()	78
11.6.7 image().col_length()	79
11.6.8 image().row_length()	79
11.6.9 image().layer_length()	79
11.6.10 image().dvalue()	80
11.6.11 image().lvalue(), image().llvalue()	81
11.6.12 image().assign()	81
11.6.13 image().convert_type()	82
11.6.14 image().bzero(), image().assign_bzero()	83
11.6.15 image().bscale(), image().assign_bscale()	84
11.6.16 image().blank(), image().assign_blank()	85
11.6.17 image().bunit(), image().assign_bunit()	86
11.6.18 image().init()	86
11.6.19 image().swap()	87
11.6.20 image().increase_dim()	88
11.6.21 image().decrease_dim()	88
11.6.22 image().resize()	88
11.6.23 image().fill()	89
11.6.24 image().add()	90
11.6.25 image().multiply()	90

11.6.26 image().fill()	91
11.6.27 image().copy(), image().cut()	92
11.6.28 image().paste()	94
11.6.29 image().add()	94
11.6.30 image().subtract()	95
11.6.31 image().multiply()	95
11.6.32 image().divide()	96
11.6.33 image().paste()	97
11.7 APIs for low-level manipulation of an Image HDU	97
11.7.1 image().data_array()	98
11.7.2 image().data_ptr()	98
11.7.3 image().get_data()	99
11.7.4 image().put_data()	100
11.7.5 image().double_value()	101
11.7.6 image().float_value()	102
11.7.7 image().longlong_value()	103
11.7.8 image().long_value()	103
11.7.9 image().short_value()	104
11.7.10 image().byte_value()	105
11.7.11 image().assign_double()	106
11.7.12 image().assign_float()	107
11.7.13 image().assign_longlong()	107
11.7.14 image().assign_long()	108
11.7.15 image().assign_short()	109
11.7.16 image().assign_byte()	110
11.8 Manipulation of Ascii Table HDU and Binary Table HDU	111
11.8.1 table().hduname(), table().assign_hduname()	111
11.8.2 table().hduver(), table().assign_hduver()	112
11.8.3 table().col_length()	112
11.8.4 table().row_length()	112
11.8.5 table().col_index()	113
11.8.6 table().col_name()	113
11.8.7 table().col().type()	114
11.8.8 table().col().bytes()	114
11.8.9 table().col().elem_byte_length()	115
11.8.10 table().col().elem_length()	115
11.8.11 table().col().dcol_length()	116
11.8.12 table().col().drow_length()	116
11.8.13 table().col().definition()	116
11.8.14 table().col().dvalue()	117
11.8.15 table().col().lvalue(), table().col().llvalue()	118
11.8.16 table().col().bvalue()	119
11.8.17 table().col().svalue()	120
11.8.18 table().col().get_svalue()	121
11.8.19 table().col().assign()	123
11.8.20 table().col().assign()	124
11.8.21 table().col().assign()	126
11.8.22 table().col().convert_type()	127
11.8.23 table().assign_null_svalue()	128
11.8.24 table().col().tzero(), table().col().assign_tzero()	128

11.8.25 table().col().tscl(), table().col().assign_tscl()	129
11.8.26 table().col().tnull(), table().col().assign_tnull()	130
11.8.27 table().col().tunit(), table().col().assign_tunit()	130
11.8.28 table().init()	131
11.8.29 table().ascii_to_binary()	132
11.8.30 table().assign_col_name()	132
11.8.31 table().define_a_col()	133
11.8.32 table().swap()	133
11.8.33 table().append_cols(), table().append_a_col()	134
11.8.34 table().insert_cols(), table().insert_a_col()	134
11.8.35 table().swap_cols()	135
11.8.36 table().erase_cols(), table().erase_a_col()	136
11.8.37 table().copy()	137
11.8.38 table().resize_rows()	137
11.8.39 table().append_rows(), table().append_a_row()	138
11.8.40 table().insert_rows(), table().insert_a_row()	138
11.8.41 table().erase_rows(), table().erase_a_row()	139
11.8.42 table().clean_rows()	139
11.8.43 table().move_rows()	140
11.8.44 table().swap_rows()	140
11.8.45 table().import_rows()	141
11.8.46 table().col().move()	142
11.8.47 table().col().swap()	142
11.8.48 table().col().clean()	143
11.8.49 table().col().import()	143
11.9 Lower level manipulation of Ascii Table HDU and Binary Table	144
11.9.1 table().col().data_array_cs()	144
11.9.2 table().col().data_ptr()	144
11.9.3 table().col().get_data()	145
11.9.4 table().col().put_data()	145
11.9.5 table().col().short_value()	146
11.9.6 table().col().long_value()	147
11.9.7 table().col().longlong_value()	148
11.9.8 table().col().byte_value()	148
11.9.9 table().col().float_value()	149
11.9.10 table().col().double_value()	150
11.9.11 table().col().bit_value()	151
11.9.12 table().col().logical_value()	152
11.9.13 table().col().string_value()	153
11.9.14 table().col().get_string_value()	153
11.9.15 table().col().assign_short()	154
11.9.16 table().col().assign_long()	155
11.9.17 table().col().assign_longlong()	156
11.9.18 table().col().assign_byte()	157
11.9.19 table().col().assign_float()	158
11.9.20 table().col().assign_double()	159
11.9.21 table().col().assign_bit()	160
11.9.22 table().col().assign_logical()	161
11.9.23 table().col().assign_string()	162

<b>12 APPENDIX1: How to Use Handy TSTRING Class</b>	<b>163</b>
<b>13 APPENDIX2: Convenient usage of DIGESTSTREAMIO class</b>	<b>165</b>

## 1 Introduction

### 1.1 All you need to know is C language

SFITSIO is a library aiming to use with codes written in C to minimize efforts of users. The manual is written in the C-language styles so that any user who has knowledge about the C language can use it easily.

Because SFITSIO is written in C++, your code with SFITSIO will be compiled on a C++ compiler. You may say “No! I can't use C++!!”. Take it easy. **C++ has upward compatibility with C so that the code written in C (such as “#include <stdio.h>”, “printf(...);” can be compiled directly by the C++ compiler.** You don't need to follow such a manner (`cout << "foo" << endl`) shown in guidebooks on C++. SFITSIO will not require getting the manner of C++ styles. Therefore you can still write with your knowledge and styles on C when you use SFITSIO.

Please experience the world of ultimate FITS I/O with taking your knowledge of the C language and SFITSIO.

### 1.2 Basic idea of SFITSIO - Interface close to scripting languages

Basic idea of SFITSIO is that a whole FITS file can be treated as strings in scripting languages (e.g. perl). According to the idea, we have developed SFITSIO in order to minimize users' load on I/O of FITS to the limit. Substitution of strings into variables written in scripting languages is frequently used. In fact, the script engine calculates, allocates, and manages memory spaces behind. However, these are not visible to the users.

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

Please think little a bit. If the implementation of the variable in scripting languages applies to I/O of FITS, it is hardly possible that the same bothers as those in using with the conventional FITS-I/O occur, isn't it?

Here comes “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 largeness). Figure 1 shows a comparison of the case of perl with that of SFITSIO. In the case of perl, string “ABC” is substituted into a variable while the whole FITS file “`foo.fits`” is substituted into a variable

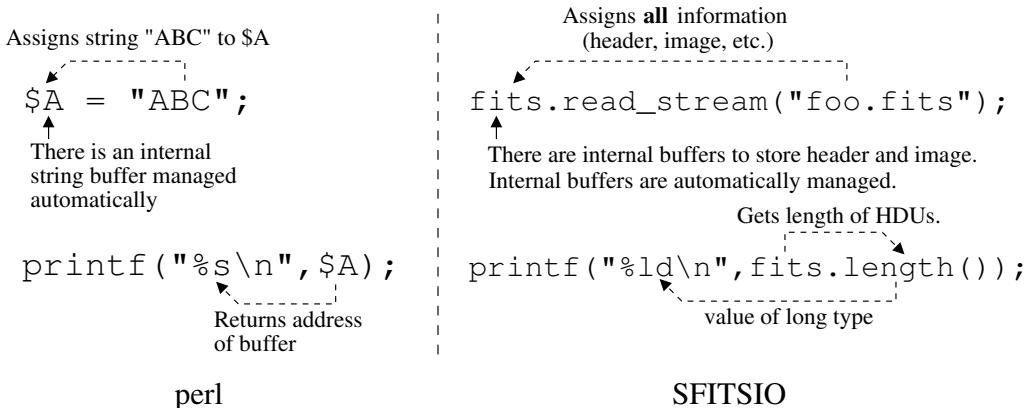


Figure 1: Comparison of perl with SFITSIO. FITS files in SFITSIO correspond to strings in perl.

(object) named “fits” in SFITSIO. In both cases of perl and SFITSIO, necessary memory spaces will be allocated and managed automatically. FITS files include several information of 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 are manipulated more manually than in script languages, it is clear that FITS files can be treated in an analogous manner with the variables in perl.

In this way, using the idea of class in C++ free users from bothers of the memory managements, and providing intelligible APIs reduces loads of users to the limit.

### 1.3 Feature of SFITSIO

SFITSIO is a FITS I/O library, usable for the C language users, independent from existing FITS libraries, and full-scratched. (Supporting Image, ASCII Table, and Binary Table.) A lot of friendly and full-fledged APIs which support CFITSIO compatible and some original extensions are available.

SFITSIO can be used easily if you have the basic knowledge of the C language. In addition, SFITSIO makes you happy because of nonnecessity of the users’ own memory management described in §1.2 and frees you from botheration of writing codes for FITS I/O with looking through the manuals. It is because the APIs of SFITSIO represent the structures of FITS files as-is and can be used just like talking so that you hardly forget the specification of the APIs if you write once. And that, access to HDUs, headers, and columns of tables by their name provides easy writing and improvement of readability of your code. The access by name, of course, needs translation from the name to its index. SFITSIO, however, keeps speed when using the name for arguments of member functions, because fast-translation with a dictionary is carried out in SFITSIO.

SFITSIO always reads or writes files sequentially because of using SLLIB(Script-Like C-language library) on access to the files. Described in §1.2, when reading a file, SFITSIO imports the whole content of the file into an object (possible to read a certain HDU). In outputting to a file, all information in the object will be written out. As mentioned above, extremely-simple file-I/O brings supports for several types of streams with minimum consumption of memory. Furthermore, SFITSIO can access remote FITS files directly through a network and read/write files as-is if the format of the files are gzip or bzip2.

## 2 Installation

### 2.1 Supporting OS

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

SFITSIO requires GCC g++ version 3 later (the author have been checked operations on g++ 3.3.2 or later.)

### 2.2 Building and installing SFITSIO

To build SFITSIO, you need SLLIB(Script-Like C-language library). Install libraries of zlib, bzlib, and readline (they probably names zlib-devel, bzip2-devel, and readline-devel as RPM)<sup>1)</sup> which are required by SLLIB.

If you don’t have installed SLLIB yet, install it in the order below. Expand an archive file of SLLIB and run make.<sup>2)</sup>

---

<sup>1)</sup> In Debian, they will be zlib1g-dev, libbz2-dev, and libreadline5-dev.

<sup>2)</sup> Shared library can be made by `make shared` for advance users.

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

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

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

Install SLLIB.

```
$ su
# make install32
```

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

Next, install SFITSIO in the same way. Expand the archive file of SFITSIO and run `make`.<sup>3)</sup>

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

Analogous to the compilation of SLLIB, you can make a 32 or 64-bit library by adding option to compiler, e.g.

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

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

Install SFITSIO.

```
$ su
# make install32
```

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

That's all for the installation.

## 3 Tutorial

We would like to introduce you to the happy-go-lucky world of SFITSIO with simple examples.

### 3.1 First Incantation

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

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

### 3.2 Read/Write files

You are able to read and write files with member functions `read_stream()` and `write_stream()`, respectively.

---

<sup>3)</sup> Shared library can be made by `make shared` for advance users.

In the example below, the program reads a file specified by the first argument in a command line, outputs the HDU information, and writes all contents of the file which has been read into the file specified by the second argument. First, we are able to show code which omits error handling.

```
#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. You can compile more easily with 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 has sure error handling. Since users generally do not need to allocate memory on using SFITSIO, error handling are required to the minimum<sup>4)</sup>. You, however, need to handle errors with return values of member functions for input/output of files.

---

<sup>4)</sup> In SFITSIO, the failing of the memory allocation causes the program to display of the reason on the standard-error output and to call 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;
        }
        /* displaying 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;
}
```

### 3.3 Direct access to remote FITS files through a network

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

Following example reads file from Web server.

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

Although the library can read only 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 may include the username and password for FTP access. If the username and password are omitted, it will access anonymously.

### 3.4 Read/Write with pipe-connections to commands (e.g., compression tools)

You can read or write a FITS file with pipe-connections to any command-line tools. Some compression tools with multi-threading support dramatically improve the efficiency of compression/decompression time.

You can use `access_stream()` and `accessf_stream()` (§11.3.4) to access files through some command-line tools. In this case, you need to set the commands including path names in the style of `open()` of Perl to the argument of the member functions.

Following example reads a compressed file using lbzip2 command which supports multi-threading.

```
sz = fits.accessf_stream("cat %s | lbzip2 -d |", "foo0.fits.bz2");
```

Next example is for writing a compressed file.

```
sz = fits.access_stream("| lbzip2 > foo1.fits.bz2");
```

Examples in §11.3.4 shows the case of reading FITS using HTTP over SSL.

### 3.5 Basis for accessing headers

Note that formations of APIs directly reflects structures of FITS files. For example, write followings for reading the value of `TELESCOP` in Primary header.

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

Reading values uses each of `dvalue()`, `lvalue()`, `llvalue()`, `bvalue()`, and `svalue()` (§11.4.4–). These functions correspond to the type of double, long, long long, bool, and const char \*, respectively.

Member functions including `assign()`, and `assign_comment()` is used for writing data, adding new record to header, and updating values and comments (§11.4.9–).

```
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);
```

It is convenient that the same style as `printf()` of libc can use at any place. Addition of commentary records such as HISTORY is simple. Member function `header_append` needs two arguments as follows.

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

Since Primary HDU is always treated as image, `fits.hdu(...)` can be replaced with `fits.image(...)`.

### 3.6 Keyword search in headers (POSIX Extended Regular Expression)

Keyword search with regular expression is available for manipulation of FITS headers including WCS. The following code is an example of searching header records in primary HDU of which the keyword begins CRVAL1 or CRVAL2. The keywords and raw values are displayed with `header( ... ).keyword()` and `header( ... ).value()` (§11.4.17).

```

fits_image &primary = fits.image("Primary");
long i = 0;
while ( 0 <= (i=primary.header_regnmatch(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 reference or alias, introduced in C++. The MACRO-like simple mechanism provides a definition of another name. Using reference may shorten code. Details in §5.3.

### 3.7 Editing headers

Member functions that initialize header (§11.4.25) and add (§11.4.27), insert (§11.4.29), and delete (§11.4.31) records of header are available. The functions for editing headers treat many records at once with structures (See the example in §3.9).

The following code shows a example that all contents of the header records in one FITS copies to another.

```

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

```

The `.header()` omitting arguments represents the whole user header. In the same manner, it can be given to member functions `header_init()` (§11.4.25) and `header_insert_records()` (§11.4.29).

For copying only a single header record not all contents, write the followings.

```

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

```

### 3.8 Access to image data

The dimension of the pixels can be obtained by using `col_length()`, `row_length`, and `layer_length()` (§11.6.7 -).

```

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

```

Use each of `dvalue()`, `lvalue()`, and `llvalue()` to read out and `assign()` to write out (§11.6.10 -). These functions transform the values according to the values of `BZERO` or `BSCALE` in the header.

Function `dvalue()` reads out the value in double which is independent from the type of image data. The coordination value begins from 0.

```

double pixel_val;
pixel_val = fits.image("Primary").dvalue(x,y,z); /* Read */
fits.image("Primary").assign(pixel_val,x1,y1,z1); /* Write */

```

Of course, it is possible to read and write with integer values. Functions `lvalue()` and `llvalue()` return values in long and long long, respectively.

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

### 3.9 Creating new FITS image

We introduce an example to create a new image file with data obtained by the telescope CASSIOPEIA. Sample code, `create_image.cc` is available in the `sample` directory of the distributed package.

Member function `append_image()` (§11.3.14) is used to create an image HDU. The example is shown below, where the type of data is  $1024 \times 1024$  in double, the name of the format type<sup>5)</sup> “CASSIOPEIA IMAGE”, and the version 101.

```
fitscc fits;
fits::header_def defs[] = { {"TELESCOP", "'CASSIOPEIA'", "Telescope name"}, 
                           {"OBSERVAT", "'NAOJ'", "Observatory name"}, 
                           {"RA", "", "[degree] Target position"}, 
                           {"DEC", "", "[degree] Target position"}, 
                           {"COMMENT", "-----"}, 
                           {NULL} };
/* Name globally unique data format (if necessary) */
fits.assign_fmttype("CASSIOPEIA IMAGE",101);
/* 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 begin. In SFITSIO, header keywords can be prepared as a structure. In addition, many of header records are assigned at once with member functions such as `header_init()` (§11.4.25). If the name of the format type is not necessary, you can omit `assign_fmttype()`.

Function `image().assign()` can set the pixel values. If necessary, values of BZERO and BSCALE can be set with `image().assign_bzero()` and `image().assign_bsclae()` (§11.6.14 -), respectively, and pixel values can be initialized to zero with `image().fill()` (§11.6.23).

```
fits.image("Primary").assign_bzero(32768);
fits.image("Primary").assign_bsclae(1);
fits.image("Primary").fill(0);
```

### 3.10 Copy and paste of image data

Member functions of `copy()` and `paste()` are available to copy and paste rectangular regions (§11.6.27 -). The following code shows that the region from the coordinate (0, 0) to (99, 99) is copied to the coordinate (100, 100).

```
fits_image copy_buf;
fits.image("Primary").copy(&copy_buf, 0, 100, 0, 100);
fits.image("Primary").paste(copy_buf, 100, 100);
```

The object `fits_image` made by the user (`=copy_buf`) becomes the copy buffer. Copy and Paste will be carried out with the member function given to the object `copy_buf`. The number of objects is unlimited. Copy and paste between objects are easy.

<sup>5)</sup> The name which identifies a globally unique data-format. See §7.

Instead of `paste()`, functions of `add()`, `subtract()`, `multiply()`, and `divide()` provide addition, subtraction, multiplication, and division (between objects), respectively (§11.6.29 -).

The object `copy_buf` used as a copy buffer stated in above example can be applied to all member function in §11.6. For example, `copy_buf` is able to read, write, and edit the value as follows.

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

The object `copy_buf` acts as an image HDU free from the `fits` object (the object of the class FITS<sup>CC</sup>).

The object `copy_buf` can be saved in a FITS file. In the case below, users create a new object of the class FITS<sup>CC</sup>, register the content of the object `copy_buf`, and 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 primary HDU without image data in a new object `new_fits` (primary HDU is created with using `new_fits.image("Primary")` unless it exists) and exchange the contents with that of `copy_buf`. Although `new_fits.append_image(copy_buf)`; can be used instead of member function `swap()`, `new_fits.append_image(copy_buf)` consumes two times of memories than `swap()` does therefore you are recommended not to use it with large image data.

### 3.11 Type conversion and fast access on image data

Fast access to the internal data of an object with pointer variables is possible. Note that the addresses of data array will change when the size of pixel is changed with using member functions such as `image().resize()` (§11.6.22).

For fast access, it is recommended to convert internal data to avoid the conversion with the values of `BZERO` and `BScale` in reading data. For conversion, use member function `image().convert_type()` (§11.6.13).

```
fits.image("Primary").convert_type(FITS::DOUBLE_T);
```

In this way, data of any type can be converted to new data of double type in which `BZERO` and `BScale` are 0 and 1. In addition, the address of data can be obtained with using member function `image().data_ptr()` (§11.7.2).

```
fits::double_t *ptr;
ptr = (fits::double_t *)fits.image("Primary").data_ptr();
```

Now, you can access internal data with “`ptr[x + y * col_length]`”.

### 3.12 Collaboration with WCSLIB

WCSLIB<sup>6)</sup> is a library, developed by Ph. D. Mark Calabretta, to manipulate World Coordinate System (WCS). The combination of SFITSIO with WCSLIB makes it easy to manipulate WCS. In this subsection, a simple example of the combination of SFITSIO with WCSLIB is shown. The example program obtains a certain celestial coordinate corresponding to a pixel included in the existing FITS file (reading into the object `in_fits`) with a WCS header. In addition, the program also obtains a value of the pixel on new FITS file (in the object `out_fits`) with a WCS header.

<sup>6)</sup> <http://www.atnf.csiro.au/people/mcalabre/WCS/WCSLIB>

```
#include <wcshdr.h>
#include <wcs.h>
```

Create an object `out_fits` for output into a file with describing below in 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 = outfitspri.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 right direction) */
outfitspri.header("CDELT1").assign(-0.01);
/* Increment of the coordinate (D is increased in 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`. Function `wcspih()` loads a WCS header into structure(s). The function requires one string composed of all headers related to WCS at the first argument. Though the string should be a byte image in the FITS file, member function `header_formatted_string()` (§11.4.34) brings the byte image of the header as an object of the class `tstring` (`tstring` class is described at APPENDIX 1 (§12)). The program displays the content of the structure(s) with `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 the celestial coordinate (`world[0][0]`, `world[0][1]`) corresponding to a certain pixel coordinate (`x_out`, `y_out`) in the object `in_fits`, calculates where the obtained coordinate corresponds to the pixel coordinate in the object `out_fits`, and substitute the calculated coordinate 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);
wcsp2s(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];

```

Function `wcsp2s()` transforms the pixel coordinates to the world coordinates and function `wcss2p()` does the world coordinates to the pixel coordinates. The functions, `wcspih()`, `wcspri()`, `wcsp2s()`, and `wcss2p()` were shown in the program described above. As long as you know these functions, you are able to do the WCS related calculation. In addition, `wcsset()` and `wcs_errmsg()` are useful functions. The former can use to reset structure `wcsset(wcs_out)`. The latter can use to obtain the string of error messages as `wcs_errmsg(status)`.

### 3.13 Access to the ASCII table and binary table

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

First of all, we show how to check the size of the table. It can be obtained by using the member functions of `table().row_length()` (§11.8.4) and `table().col_length()` (§11.8.3). The followings shows an example of displaying the size of binary table “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 access FITS headers, such as `fits.table(HDU name).col(column name)`. .... If HDU or column does not have a name, the index number beginning with 0 can be specified.

Each of `dvalue()`, `lvalue()`, `l1value()`, `bvalue()`, and `svalue` can be used to read out the value, and `assign()` to write out(§11.8.14 -). These functions also automatically convert the values with using the values `TZEROn` and `TSCALn` in the header.

Function `dvalue()` is able to load the columns of any data type to the double type form. The row number starts with 0. The following shows an example of 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 on reading and the value and index successively on writing in the arguments.

You can read and write values in integer or boolean. Functions `lvalue()`, `l1value()`, and `bvalue()` return values in long, long long, and boolean type, respectively. Reading out values in string type, use `svalue()`. Function `svalue()` brings values in string type, which is formatted according to the specification of `TDISPn` in the header if the value is number.

```

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

```

If a column has multiple elements, `TFORMn = '8J'`, for instance, means that the column has eight elements. The number of the elements can be obtained by using `table(...).col(...).elem_length()` (§11.8.10). The value to read out in the column can be specified at the second argument of each

of `dvalue()`, `lvalue()`, `llvalue()`, `bvalue()`, and `svalue()`. Upon writing, it can be specified at the third argument of `assign()`. The following code is an example of displaying all elements of column “STATUS” 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");
}
```

### 3.14 Creation of binary table

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

It creates a table composed of three columns (double, 32-bit integer, and string), where the name of the format type<sup>7)</sup> and its version assigns “ASTRO-X XXX Event Table” and 101, respectively.

```
fitscc fits;
const fits::table_def def[] = {
    /* ttype,comment,           talas,telem,tunit,comment,          */
    /*                           tdisp,  tform, tdim   */
    { "TIME", "satellite time", NULL,NULL, "s","", "F16.3", "1D", "" },
    { "STATUS", "status",      NULL,NULL, "", "", "", "8J", "" },
    { "NAME", "",             NULL,NULL, "", "", "", "128A16", "(4,2)" },
    { NULL }
};
/* Name globally unique data format (If necessary) */
fits.assign_fmttype("ASTRO-X XXX Event Table", 101);
/* Create a binary table (name of HDU is "EVENT") */
fits.append_table("EVENT",0, def);
```

First, It prepares the definition of columns with structures, defines the name of the format type with using member function `assign_fmttype()` (§11.3.19) (if necessary), and create a binary table with using member function `append_table()` (§11.3.15). Since binary tables cannot become primary HDUs in the definition of FITS, the primary HDU without image data is created automatically.

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

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

### 3.15 Creation of ASCII table

The method of an ASCII table is similar to that of a binary table. Care should be taken when you specify `tdisp` and `tform` in the structure. In creation of ASCII tables, strings in TFORMn of FITS file is specified for `tdisp` in a structure, and width of strings as “*nA*” for `tform` in a structure.

<sup>7)</sup> The name which identifies a globally unique data-format. See §7.

It creates a table composed of three columns, where the name of the format type and its version assigns “ADC TEST TABLE” and 101, respectively.

```

fitscc fits;
const fits::table_def def[] = {
    /* ttype,comment,          talas,telem, tunit,comment, tdisp,   tform */
    { "PK", "PK number",      NULL, NULL,   "", "",        "A9",     "9A" },
    { "RAH", "Hours RA",     NULL, NULL,   "h", "",        "I2",     "3A" },
    { "RAM", "Minutes RA",   NULL, NULL,   "min","",       "F5.2",   "6A" },
    { NULL }
};

/* Name globally unique data format (If necessary) */
fits.assign_fmttype("ADC TEST TABLE", 101);
/* Create a binary table (name of HDU is "PLN") */
fits.append_table("PLN",0, def, true);

```

First, prepare the definition of columns with structures, define the name of the data format with using member function `assign_fmttype()` (§11.3.19) (if necessary), and create an ASCII table with using member function `append_table()` (§11.3.15) with giving `true` for the last argument. Since ASCII tables cannot become primary HDUs in the definition of FITS, the primary HDU without image data is created automatically.

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

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

### 3.16 Editing and importing of ASCII and binary tables

SFITSIO enables not only addition, insertion, and deletion of columns (§11.8.33 -) and of rows (§11.8.39 ) but also importing from other ASCII tables or binary tables (§11.8.45).

The following code is an example of copying 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") );
```

Next example shows how to combine 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, member function `resize_rows()` extends the number of rows up to the sum of two tables. Last, member function `import_rows()` (§11.8.45) pastes the whole content of the table SRC into the rear margin of the table EVENT. The arguments of `import_rows()` are the number of rows with which paste begins, whether it pastes the column of which the name is matched, and the source of the table.

Member function `table().col().import()` (§11.8.49) can be used for import in units of column.

### 3.17 Editing of HDU

It is easy to merge image of multiple FITS file and ASCII or binary tables into one FITS and to delete unnecessary HDU with using SFITSIO. Giving `fits.image("FOO")` or `fits.table("BAR")`

to the argument of member functions `.append_image()`, `.append_table()` and so on provides easy addition (§11.3.14, §11.3.15) and insertion (§11.3.16, §11.3.17) of Image HDU and binary (or ASCII) table HDU.

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

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

Next example is deletion of the HDU name FOO in the object `fits_out`.

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

## 4 Internal structure of the object

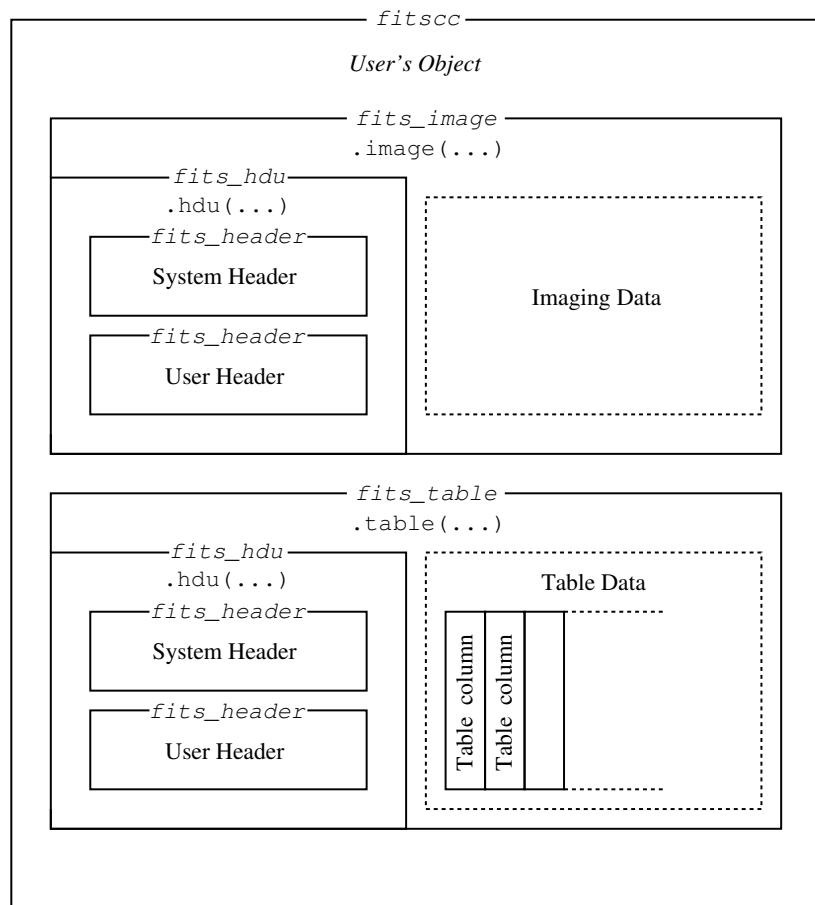


Figure 2: Scheme of the internal structure in the user's object of class `fitscc`. The figure illustrates the case that user create an image HDU and a binary (or an ASCII) table HDU.

In the tutorial of §3, as described below

```
fitscc fits;
fits.append_image( ... );
fits.append_table( ... );
```

single fits file was represented with addition of the image HDU and the table HDU into the

object of class `fitscc`. As seen above, the headers for system and users, image data, and table data are created and managed in the inside of the `fits` object.

Figure 2 shows a brief configuration. Teletype letters with italic style in the figure are the name of the class. In fact, headers and data are managed with creating objects automatically from these classes inside SFITSIO. Member functions shown as teletype letters in the figure can be used to access the internal objects. For example,

```
fits.image("Priamry").foo(...  
fits.table("EVENT").bar(...
```

Although it is omitted in the figure 2, the user's object has objects of class `fits_header` which manages multiple header records, of class `fits_header_record` which represents one header record, and of class `fits_table_col` which represents one column of the table, objects which are created and managed automatically in the user's object.

Keeping the brief structure as shown in figure 2 in your mind proceeds to learn APIs smoothly although this document is available to use without knowledge of the whole structure. Especially, references, described in §5.3, is very usable for treating SFITSIO and will help you understand SFITSIO.

## 5 Things to know before using SFITSIO

You will use C++ compiler because SFITSIO is a library of C++. Since C++ has upward compatibilities of C, basically you can write code in the same style as C.

However, there are a few (and not so difficult) things to know before using SFITSIO. They are incompatibility between C and C++ accompanied with extension of C++, and easy and convenient extended function of C++. We describe them in this section.

### 5.1 Namespace

Namespace is one of the things introduced in C++. It is used for avoiding the problem that different programmers write the same name of functions or types. It is similar to a category name. SLLIB and SFITSIO assigns the own namespace as “`sli`”. If you use some kind of class, you need to put `sli::` at the top like “`sli::stdstreamio sio;`” for writing code in proper form.

If you use SFITSIO mainly, you may not want to write `sli::` each time. In this case, write in the below way to skip `sli::` for the rest of the code.

```
using namespace sli;
```

The examples of this manual omit `sli::`. For C++ beginners, it may be easy to keep in mind that once you write “`#include <...>`”, then write “`using namespace sli;`”.

### 5.2 NULL and 0

In many processing systems, `NULL` in C is defined as

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

But `NULL` in C++ is defined as

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

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

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

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

If `hoge.foo(NULL)` or `hoge.foo(0)` exists, the compiler cannot determine which function should be used. In this case, it is necessary to indicate explicitly the type to cast. That is

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

In C++, it is safer that you remember to “**cast if use NULL or 0**”. Another way is to cast NULL out and “**always cast 0**”.

### 5.3 Reference

It is too much work to write `fits.hdu("Primary")` or `fits.image("Primary")` in each access to the header or the image. “**Reference**” can be used for writing shorter. Reference, also called “**Alias**”, creates an alias of a variable or an object, in simple terms. Although Macro may be used for aliases for variables or objects, using references brings more smart scripting.

Reference introduced in C++ is a new type similar to the pointer type. In fact, **the functions of the reference is more simple than that of pointer type**. Accordingly, **the usage is simple too**. For example, creation of reference “`aref`” of the variable `int a;` is

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

Reference, so called “**Alias**”, acts exactly the same. For instance,

```
aref = 10;
```

then, 10 is assigned to `a`, and

```
int b = aref;
```

cause a substitution of the value of `a` for `b`.

```
int *p = &aref;
```

substitutes the address of `a` to `p`.

As described above, reference is a simple mechanism of providing an alias of variable. It is not so complicated as pointer variables with many \*. Reference cannot have NULL because it is forbidden to create references without its instance.

When using SFITSIO, you may use the reference mostly as copying it returning from member functions into another reference variable that user creates. For instance, member function `fits.hdu()` returns the reference “`fits_hdu &`”. If user creates a reference of the same class and copies the return value in the receiving side, it can be a substitution of `fits.hdu("Primary")` or `fits.image("Primary")` so that user can write shorter code. The following code is the 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 reference without substitution of value on declaration causes an error. It is because “**Alias**” means “another name”. “**Alias**” without its instance causes an error.

### 5.4 Try & catch

You can skip this part if you intend not to write heavy codes.

The syntax `try{}` and `catch(){}` are to treat “Exception” introduced in C++. SFITSIO causes “Exception” when critical problems, such as “**memory allocation was failed**” occur irrelevantly to the arguments given by users. The “Exception” can be handled with `try{}` and `catch(){}` in user’s code. In SFITSIO, exceptions occurred always with the message of `err_rec` type. If you catch it, write

```
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;
}
```

When the exception occurs without using `try{}` and `catch ( err_rec msg ){}`, the function `abort()` will be called and the program will be terminated.

Generally, such a critical error in most cases discontinues the program. Thus, if you do not have any problem in using the program called `abort()`, it is not necessary to use `try{}` and `catch( err_rec msg ){}`.

## 6 System and Notice of FITS Header Management in SFITSIO

In SFITSIO, the headers are separated by keywords, "System Header" and "User Header". This is to avoid the user's carelessness of mismatching the header part and the data part inside the FITS file.

The system header treats the keywords which affect configuration of the FIT's data area, such as **BITPIX** and **NAXIS**. Therefore, the users cannot edit them directly. Instead, a lot of member functions to treat contents of data area are available.

The "User Header" is the header which user can read and write. Basically, header-related APIs are targetted to this user header. Thus, although **BITPIX** may not exist in the user header, this is not a bug.

Next table shows the list of header keywords that cannot exist in the "User Header". This table also includes the member functions to read the values equivalent to records of header. If you cannot but read directly the records, see §11.5.

keyword	member functions to read	section
<b>SIMPLE</b>	—	—
<b>BITPIX</b>	image().type(), image().bytes()	§11.6.5, §11.6.6
<b>NAXIS</b>	image().dim_length()	§11.6.3
<b>EXTEND</b>	—	—
<b>FMTTYPE</b>	fmttype()	§11.3.7
<b>FTYPEVER</b>	ftypever()	§11.3.8
<b>XTENSION</b>	hdutype()	§11.3.11
<b>PCOUNT</b>	—	—
<b>GCOUNT</b>	—	—
<b>TFIELDS</b>	table().col_length()	§11.8.3
<b>EXTNAME</b>	hduname()	§11.3.9
<b>EXTVER</b>	hduver()	§11.3.10
<b>BSCALE</b>	image().bscale()	§11.6.15
<b>BZERO</b>	image().bzero()	§11.6.14
<b>BUNIT</b>	image().bunit()	§11.6.17
<b>BLANK</b>	image().blank()	§11.6.16
<b>NAXIS<i>n</i></b>	image().length()	§11.6.4
<b>TTYPEn</b>	table().col_name()	§11.8.6
<b>TUNIT<i>n</i></b>	table().col().tunit()	§11.8.27
<b>TDISP<i>n</i></b>	table().col().definition()	§11.8.13
<b>TBCOL<i>n</i></b>	—	—
<b>TFORM<i>n</i></b>	table().col().type(), etc.	§11.8.7
<b>TDIM<i>n</i></b>	table().col().dcol_length(), etc.	§11.8.11
<b>TNULL<i>n</i></b>	table().col().tnull()	§11.8.26
<b>TZERO<i>n</i></b>	table().col().tzero()	§11.8.24
<b>TSCAL<i>n</i></b>	table().col().tscal()	§11.8.25
<b>THEAP<i>n</i></b>	—	—
<b>TELEM<i>n</i></b>	table().col().elem_length()	§11.8.10
<b>TALAS<i>n</i></b>	table().col().definition()	§11.8.13

## 7 Error Check and Version Management of FITS File by FMTTYPE and FTYPERVER

In order to check the errors and manage the versions easier, SFITSIO is designed to set “name” and “version” in the FITS format defined by the user to FMTTYPE and FTYPERVER in the primary header. 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 FTYPERVER indicates the version of that data format and should be set to an integer value.

In telescope or astronomical satellite projects, HDU configuration, keyword configuration of the header, ASCII table and column configuration of binary table of original FITS are defined frequently. Once the FMTTYPE is named for each format, in user programs in which only specific FITS files are supposed to be treated, users can avoid problems due to specification of the FITS file by only checking the FMTTYPE. For example, in a case that an input file is the FITS file which is not supported by the program, user can process it as an error by checking the FMTTYPE.

Of course, there might be a situation where a header keyword is added or modified for the definition of the FITS file in process of the project. For the version management in such a case, the FTYPERVER is available. If the version is managed accurately, it is possible to separate the process according to the value of the FTYPERVER. In addition, we recommend using more than three digit number for FTYPERVER to express a minor version. Alternatively, a date such as 20080101 can also be used.

## 8 Expansion of FITS Compatible with CFITSIO

### 8.1 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,RSTWID&
CONTINUE 'ELON,RSTWIDESON,RSTN1700N,RSTN600N,LWBOOSTON,SWBOOSTON,LWBIASON,SWB&
CONTINUE 'IASON,CALALON,CALASON,CALBON,SINALON,SINASON' / elements in STATUS
```

In case of SFITSIO, this process for expansion, 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.

### 8.2 Array of fixed length strings in binary table

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

Furthermore, in the SFITSIO, an assignment like TFORMn = '120A10' AND TDIMn = '(6,2)', allows users to treat a 10-character string as a  $6 \times 2$  array. This assignment can also be written as TFORMn = '120A' AND TDIMn = '(10,6,2)'.

## 9 SFITSIO's Original Expansion of FITS

### 9.1 Distinction of upper and lower case in the header keyword

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
```

## 9.2 Long keyword in the header (maximum 75 characters)

In ESO's convention, the header record of the long keyword is labeled HIERARCH, however, this way is not sophisticated. In SFITSIO, the long keyword can be saved simply as follows.

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

The keyword is up to 75 characters and space or “=” cannot be used.

## 9.3 Number of columns of the ASCII table or the binary table that exceed 999

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

## 9.4 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
```

The Alias defined here is also available in the API of SFITSIO.

In case that multiple aliases need to be defined, define them in csv format.

## 9.5 Definition of an element name in the column of the binary table

In SFITSIO, each element in the column of the binary table can be named as follows. (It is not necessarily that all elements should be named.)

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

Each name of the element is saved in the record of TELEM $n$  keyword in the csv format. 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().

## 9.6 Definition of the number of bits in the column of the binary table

In case that TFORM $n$  is ' $nX$ ', 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,QUAL_RP_D&
CONTINUE 'ATA:2,QUAL_RP_PARAM:2,QUAL_RP_TABLE:2,QUAL_FF_PARAM:2,QUAL_FF_CF:2,&
CONTINUE 'QUAL_GPGL_CORR:2,QUAL_MTGL_CORR:2,QUAL_TR_HIST:2,QUAL_TR_PARAM:2,QU&
CONTINUE 'AL_DK_DATA:2,QUAL_DK_PARAM:2,QUAL_DK_TABLE:2,QUAL_FX_CORR:2,QUAL_FX&
CONTINUE '_PARAM:2,,,,'        / element names
```

In this case, it is indicated that each element is 2-bit data and member functions can read or write the 2-bit value for the each element.

## 10 Unsupported FITS Standard, etc.

FITS standard unsupported by the SFITSIO is following.

- Random groups structure.
- Variable length array in the binary table.(giving *rP* or *rQ* to THEAP*n* or TFORM*n*)

SFITSIO provides file I/O operations on files with complex numbers, but does not provide sufficient supports of high-level APIs. In case of the process with complex numbers, users have to use low-level APIs.

## 11 Reference

### 11.1 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
<code>const int</code>	<code>FITS::IMAGE_HDU</code>	0
<code>const int</code>	<code>FITS::ASCII_TABLE_HDU</code>	1
<code>const int</code>	<code>FITS::BINARY_TABLE_HDU</code>	2

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

Type	Constants in the SFITSIO	Value
<code>const int</code>	<code>FITS::NULL_RECORD</code>	0
<code>const int</code>	<code>FITS::NORMAL_RECORD</code>	1
<code>const int</code>	<code>FITS::DESCRIPTION_RECORD</code>	2

To indicate the kind of data, use following constants.

Type	Constants in the SFITSIO	Value	Image	Binary	Table
<code>const int</code>	<code>FITS::BIT_T</code>	88		○	
<code>const int</code>	<code>FITS::BYTE_T</code>	66	○	○	
<code>const int</code>	<code>FITS::BOOL_T</code>	76		○	
<code>const int</code>	<code>FITS::LOGICAL_T</code>	76		○	
<code>const int</code>	<code>FITS::STRING_T</code>	65		○	
<code>const int</code>	<code>FITS::SHORT_T</code>	73	○	○	
<code>const int</code>	<code>FITS::LONG_T</code>	74	○	○	
<code>const int</code>	<code>FITS::LONGLONG_T</code>	75	○	○	
<code>const int</code>	<code>FITS::FLOAT_T</code>	69	○	○	
<code>const int</code>	<code>FITS::DOUBLE_T</code>	68	○	○	
<code>const int</code>	<code>FITS::COMPLEX_T</code>	67		○	
<code>const int</code>	<code>FITS::DOUBLECOMPLEX_T</code>	77		○	

### 11.2 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
<code>fits::bit_t</code>	<code>uint8_t</code>		○
<code>fits::byte_t</code>	<code>uint8_t</code>	○	○
<code>fits::logical_t</code>	<code>uint8_t</code>		○
<code>fits::short_t</code>	<code>int16_t</code>	○	○
<code>fits::long_t</code>	<code>int32_t</code>	○	○
<code>fits::longlong_t</code>	<code>int64_t</code>	○	○
<code>fits::float_t</code>	<code>float</code>	○	○
<code>fits::double_t</code>	<code>double</code>	○	○
<code>fits::complex_t</code>	<code>float _Complex</code>		○
<code>fits::doublecomplex_t</code>	<code>double _Complex</code>		○
<code>fits::string_t</code>	<code>char *</code>		○

To define the FITS header, use following structures.

Types	Definition of structure
<code>fits::header_def</code>	<pre>struct {     const char *keyword;     const char *value;     const char *comment; }</pre>

To define the column of ASCII table and Binary table, use following structures. (The usage of `theap` is not supported.)

Types	Definition of structure
<code>fits::table_def</code>	<pre>struct {     const char *ttype;     const char *ttype_comment;     const char *const *talas;     const char *const *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 *tscl;     const char *theap; }</pre>

## 11.3 Operation of whole FITS

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

### 11.3.1 `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<sup>8)</sup>. 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()`(§11.3.2) 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 §11.4.9 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 §11.3.2 or §3.2, §3.3 in Tutorial.

<sup>8)</sup> This function is carried out in digeststreamio class of SLLIB. Detail in APPENDIX2 (§13)

### 11.3.2 `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 (§11.3.1), 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 (§11.3.13).

#### 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);
```

### 11.3.3 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 a 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<sup>9)</sup>. 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 §3.2 and §3.3 in Tutorial.

<sup>9)</sup> This function is carried out in digeststreamio class of SLLIB. Detail in APPENDIX2 (§13)

### 11.3.4 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<sup>10)</sup>.

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).

---

<sup>10)</sup> This function is carried out in `digeststreamio` class of SLLIB. Detail in APPENDIX2 (§13)

**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 §3.4 in Tutorial. The tutorial shows some examples for the use of a compression tool with multi-threading support.

---

**11.3.5 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();
```

---

**11.3.6 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 §3.2 in Tutorial.

---

### 11.3.7 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 §7.

#### RETURN VALUE

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

#### EXAMPLES

---

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

---

### 11.3.8 ftypever()

#### NAME

`ftypever()` — Version number of format type

#### SYNOPSIS

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

#### RETURN VALUE

`ftypever()` returns version number of format type.

#### EXAMPLES

---

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

---

### 11.3.9 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 §3.2 in Tutorial.

---

**11.3.10 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));
}
```

---

**11.3.11 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 a unknown type HDU!\n");
        break;
}
```

---

**11.3.12 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 <code>name</code> . |
| 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");
```

### 11.3.13 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();
```

### 11.3.14 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]	<code>hduname</code>	HDU name
[I]	<code>hduver</code>	HDU version
[I]	<code>type</code>	HDU type
[I]	<code>naxis0</code>	Number of pixels of X axis
[I]	<code>naxis1</code>	Number of pixels of Y axis
[I]	<code>naxis2</code>	Number of pixels of Z axis
[I]	<code>naxisx</code>	List of number of pixels of each axis
[I]	<code>ndim</code>	Number of elements of <code>naxisx</code>
[I]	<code>src</code>	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  $1024 \times 1024$  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 §3.9 in Tutorial.

### 11.3.15 `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 `TTYPEn`, `TFORMn`, 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. `TFORMn` 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

`Aw, Iw, Fw.d, Ew.d` or `Dw.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, def);
```

See also the examples in §3.14 and §3.15 in Tutorial.

---

### 11.3.16 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()`(§11.3.14).

**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  $1024 \times 1024$  Image HDU of which name is “X-TABLE” and version is 100 in front of HDU “X-BAND”.

```
fits.insert_image("X-BAND", "X0-BAND", 100, FITS::DOUBLE_T, 1024, 1024);
```

---

**11.3.17 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 `append_table()` (§11.3.15).

#### PARAMETER

[I] <code>index0</code>	HDU index which designate insert position
[I] <code>hduname0</code>	HDU name which designate insert position
[I] <code>hduname</code>	HDU name
[I] <code>hduver</code>	HDU version
[I] <code>defs</code>	<code>fits::table_def</code> structure
[I] <code>ascii</code>	type of Table HDU to be inserted (false: Binary, true: Ascii)
[I] <code>src</code>	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 §11.3.15 for structure array `defs`.)

---

```
fits.insert_table("X-BAND", "X-TABLE", 100, defs);
```

---

### 11.3.18 `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] <code>index</code>	HDU index to be erased
[I] <code>hduname</code>	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");
```

---

**11.3.19 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 §7 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 §3.9, §3.14 and §3.15 in Tutorial.

---

**11.3.20 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);
```

---

**11.3.21 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");
```

---

**11.3.22 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);
```

---

**11.3.23 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]	<code>index</code>	HDU index
[I]	<code>hduname</code>	HDU name
[I]	<code>extname</code>	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);
    }
}
```

---

## 11.4 Operation of user header

In this section, we describe how to handle APIs to operate a user header. APIs are classified in two groups. 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 75 characters and there is no limit to the string length in the SFITSIO. (Refer §8.1.) Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

### 11.4.1 hdu().header\_length()

#### NAME

`hdu().header_length()` — Number of records in a user header

#### SYNOPSIS

```
long hdu( ... ).header_length() const;
```

#### RETURN VALUE

`hdu().header_length()` returns the number of records in a user header.

#### EXAMPLES

Following code displays the number of records in a user header.

```
fits_image &primary = fits.image("Primary");
printf("Record Count = %ld\n", primary.header_length());
```

### 11.4.2 hdu().header\_index()

#### NAME

`hdu().header_index()` — Header record index in a user header

#### SYNOPSIS

```
long hdu( ... ).header_index( const char *keyword ) const;
long hdu( ... ).header_index( const char *keyword, bool is_description ) const;
```

#### 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"));
```

---

### **11.4.3 hdu().header\_regmatch()**

#### **NAME**

hdu().header\_regmatch() — Keyword search of a user 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 §3.6 in Tutorial.

---

#### 11.4.4 hdu().header().svalue()

##### NAME

`hdu().header().svalue()` — A string value of a user 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 user 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 user 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 §3.5 in Tutorial.

---

#### 11.4.5 hdu().header().get\_svalue()

##### NAME

`hdu().header().get_svalue()` — Get a string value of a user 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 user 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));
```

---

**11.4.6 hdu().header().dvalue()****NAME**

hdu().header().dvalue() — Real value of a user 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 §3.5 in Tutorial.

---

**11.4.7 hdu().header().lvalue(), hdu().header().llvalue()****NAME**

hdu().header().lvalue(), hdu().header().llvalue() — Integer value of a user 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 §11.4.6 or §3.5 in Tutorial.

---

**11.4.8 hdu().header().bvalue()****NAME**

`hdu().header().bvalue()` — Boolean value of a user 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 §11.4.6 or §3.5 in Tutorial.

---

**11.4.9 hdu().header().assign(), hdu().header().assignf()****NAME**

`hdu().header().assign()` — Assign a string value to a user 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().assign()` or `hdu().header().assignf()` 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.

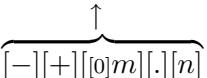
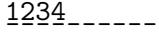
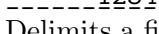
There is no limit to the string length. (Refer §8.1.) Even if the length of a header record exceeds 80 characters, it is stored to a file appropriately.

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
hd	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
lllo	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. dddddd 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.

Option	Description	Examples
– (minus sign)	Aligns field contents to the left instead of the right 	.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” 	.printf("%10d..."
.	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 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 §3.5 in Tutorial.

### 11.4.10 hdu().header().assign()

#### NAME

hdu().header().assign() — Assigns a boolean value to a user 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 §11.4.9 or §3.5 in Tutorial

---

**11.4.11 hdu().header().assign()****NAME**

`hdu().header().assign()` — Assigns an integer value to a user 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 §3.5 Tutorial.

---

### 11.4.12 hdu().header().assign()

#### NAME

`hdu().header().assign()` — Assigns a real value to a user 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 §3.5 in Tutorial.

### 11.4.13 hdu().header().type()

#### NAME

`hdu().header().type()` — Type of user 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

<code>FITS::DOUBLE_T</code>	: In case that header record is real number.
<code>FITS::LONGLONG_T</code>	: In case that header record is integer number.
<code>FITS::DOUBLECOMPLEX_T</code>	: In case that header record is complex number.
<code>FITS::BOOL_T</code>	: In case that header record is boolean number.
<code>FITS::STRING_T</code>	: In case of others.

## EXAMPLES

Following code checks a type of user 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;
}
```

---

### 11.4.14 hdu().header().status()

#### NAME

`hdu().header().status()` — Status of user 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

- |                                       |   |
|---------------------------------------|---|
| <code>FITS::NORMAL_RECORD</code>      | : In case that header record is normal format.      |
| <code>FITS::DESCRIPTION_RECORD</code> | : In case that header record is description format. |
| <code>FITS::NULL_RECORD</code>        | : In case that header record is NULL format.        |

## EXAMPLES

Following code checks a status of user 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;
}
```

```

        break;
    case FITS::NULL_RECORD:
        printf("Record is NULL format.\n");
        break;
    default:
        printf("This message should not be shown.\n");
        break;
}

```

---

### 11.4.15 hdu().header().keyword()

#### NAME

`hdu().header().keyword()` — Keyword name of a user 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());

```

---

### 11.4.16 hdu().header().get\_keyword()

#### NAME

`hdu().header().get_keyword()` — Get a keyword string of a user 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));
```

---

**11.4.17 hdu().header().value()****NAME**

`hdu().header().value()` — A value of a user 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()`(§11.4.4)

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 §11.4.15.

**11.4.18 hdu().header().get\_value()****NAME**

`hdu().header().get_value()` — A value of a user 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()`(§11.4.5).

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 §11.4.16.

---

**11.4.19 hdu().header().comment()****NAME**

`hdu().header().comment()` — A comment of a user 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 §11.4.15.

---

**11.4.20 hdu().header().get\_comment()****NAME**

`hdu().header().get_comment()` — A comment of a user 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 §11.4.16.

---

**11.4.21 hdu().header().assign\_value()****NAME**

`hdu().header().assign_value()` — Assign a string to a user 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().assign_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 §8.1.) 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().assign_value()` returns a reference to relevant `fits_header_record`.

**EXCEPTION**

If the API 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');
```

---

**11.4.22 hdu().header().assign\_comment()****NAME**

`hdu().header().assign_comment()` — Change a comment of a user 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().assign_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().assign_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 §3.5 in Tutorial.

---

**11.4.23 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 user 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");
```

---

**11.4.24 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] <code>index</code>	An index of header record to be updated
[I] <code>header_keyword</code>	A keyword of header record to be updated
[I] <code>def</code>	<code>fits::table_def</code> structure to be copied at updating
[I] <code>obj</code>	An object to be copied at updating
[I] <code>keyword</code>	A new keyword value
[I] <code>value</code>	A new record value
[I] <code>comment</code>	A new comment value
[I] <code>description</code>	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 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");
```

---

### 11.4.25 hdu().header\_init()

#### NAME

`hdu().header_init()` — Initialize a user 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 user header and if `defs` is specified, assign that value to the user 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 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 §3.9 in Tutorial.

### 11.4.26 hdu().header\_swap()

#### NAME

`hdu().header_swap()` — Swap a user header

#### SYNOPSIS

```
fits_hdu &hdu( ... ).header_swap( fits_header &obj );
```

**DESCRIPTION**

`hdu().header_swap()` swaps a FITS header object for one which specified as `obj`.

**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 user header of HDU “AAAA” for one of HDU “BBBB”.

---

```
fits_header &obj = fits.hdu("BBBB").header();
fits.hdu("AAAA").header_swap(obj);
```

---

**11.4.27 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 user header. All last members in the array of `defs` are set to NULL. For more information on `fits::header_def`, see §11.4.25(`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 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 §3.7 in Tutorial.

---

#### 11.4.28 hdu().header\_append()

**NAME**

`hdu().header.append()` — Append a header record

## SYNOPSIS

```

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 user 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` (§8.1). 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	<code>fits::table_def</code> 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 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 §3.7 in Tutorial.

#### 11.4.29 hdu().header\_insert\_records()

**NAME**

`hdu().header_insert_records()` — Insert a header record

## SYNOPSIS

**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 §11.4.25(`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 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);
```

---

**11.4.30 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] <code>index</code>	An index in header records for the insertion
[I] <code>record_keyword</code>	A keyword in header records for the insertion
[I] <code>def</code>	<code>fits::header_def</code> structure to be copied at the insertion
[I] <code>obj</code>	An object to be copied at the insertion
[I] <code>keyword</code>	A keyword to be inserted
[I] <code>value</code>	A value to be inserted
[I] <code>comment</code>	A comment to be inserted
[I] <code>description</code>	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");
```

---

### 11.4.31 `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] <code>index</code>	An index in header records. The API deletes records starting with the index
[I] <code>keyword</code>	A keyword in header records. The API deletes records starting with the keyword.
[I] <code>num_records</code>	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 re-size 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);
```

---

**11.4.32 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]	<code>index</code>	An index to be deleted in header records.
[I]	<code>keyword</code>	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 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);
```

---

**11.4.33 hdu().header()****NAME**

`hdu().header()` — A reference to a user header object

**SYNOPSIS**

```
fits_header &hdu( ... ).header();
```

**RETURN VALUE**

`hdu().header()` returns a reference to a user header object.

**EXAMPLES**

The following code copies a reference to a user header object in Primary HDU.

---

```
fits_header &primary_header = fits.hdu("Primary").header();
```

---

### 11.4.34 hdu().header\_formatted\_string()

#### NAME

`hdu().header_formatted_string()` — Formatted user 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 user header in the object and returns the address.

This API facilitates cooperation with WCSLIB.

#### RETURN VALUE

`hdu().header_formatted_string()` returns an address to the formatted strings.

#### EXCEPTION

If the API 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 user header records in Primary HDU and the whole string. `tstring` class allows users 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 in §3.12 in Tutorial.

For more information about `tstring` class, see APPENDIX1(§12).

---

## 11.5 Manipulation of a System Header

In this section, we describe APIs to manipulate a system header. In the SFITSIO, the system header is used for storing the header records only which manages a HDU's data part, only when it is necessary. The records which have to be stored are the records which have the keywords such as BITPIX or NAXIS. Since these records are created automatically via APIs of the SFTTSIO, users do not need to set up them directly. Consequently, every APIs for manipulating the system header directly are only for reading. In addition, note that the system header have to be restructured by using `stream_length()` member function (§11.3.5) before using these APIs, when a FITS is newly created or data structure is modified. In fact, the system header is a buffer for input/output of a stream and it does not always reflect the status of the data part.

Therefore, if you do not understand how role the system header plays in the SFITSIO, do not utilize APIs in this section. The system header should not need to be read in most purposes because the status of the data part can be acquired via member functions in the SFITSIO.

For the rest of this document, since the argument in the bracket of “`hdu( ... )`” is all same, descriptions about this argument is omitted. HDU index (`long index`), HDU name (`const char *hduname`) can be specified for this argument. Additionally, `hdu( ... )` can be replaced by `image( ... )` in case of Image HDU or `table( ... )` in case of Ascii Table HDU or Binary Table HDU.

### 11.5.1 `hdu().sysheader_length()`

#### NAME

`hdu().sysheader_length()` — Number of system header records

#### SYNOPSIS

```
long hdu( ... ).sysheader_length() const;
```

#### RETURN VALUE

`hdu().sysheader_length()` returns number of system header records.

#### EXAMPLES

```
printf("SYSTEM HEADER RECORD COUNT = %ld\n",
      fits.image("Primary").sysheader_length());
```

---

### 11.5.2 `hdu().sysheader_index()`

#### NAME

`hdu().sysheader_index()` — An index of a system header record

#### SYNOPSIS

```
long hdu( ... ).sysheader_index( const char *keyword ) const;
```

#### DESCRIPTION

`hdu().sysheader_index()` member function returns the index of the system header record specified by `keyword`.

#### PARAMETER

- [I] `keyword` Keyword of a header record
- ([I] : input, [O] : output)

#### RETURN VALUE

Non-negative value	:	Record index.
Negative value	:	Error (If specified keyword is not found.)

**EXAMPLES**

```
printf("INDEX = %ld\n", fits.image("Primary").sysheader_index("BITPIX"));
```

---

**11.5.3 hdu().sysheader\_keyword()****NAME**

`hdu().sysheader_keyword()` — Keyword name of a system header

**SYNOPSIS**

```
const char *hdu( ... ).sysheader_keyword( long index ) const;
```

**DESCRIPTION**

`hdu().sysheader_keyword()` member function returns a keyword of the system header record specified by `index`.

**PARAMETER**

- [I] `index` index of a header record
- ([I] : input, [O] : output)

**RETURN VALUE**

`hdu().sysheader_keyword()` returns an address of a keyword string of a system header record.

**EXAMPLES**

```
printf("KEYWORD = %s\n", fits.image("Primary").sysheader_keyword(index));
```

---

**11.5.4 hdu().sysheader\_value()****NAME**

`hdu().sysheader_value()` — A value of a system header

**SYNOPSIS**

```
const char *hdu( ... ).sysheader_value( long index ) const;
const char *hdu( ... ).sysheader_value( const char *keyword ) const;
```

**DESCRIPTION**

`hdu().sysheader_value()` member function returns a value assigned to the system header record specified by `index` or `keyword`.

**PARAMETER**

- [I] `index` Index of a header record
- [I] `keyword` Keyword of a header record
- ([I] : input, [O] : output)

**RETURN VALUE**

`hdu().sysheader_value()` returns an address of a value string of a system header record.

**EXAMPLES**

```
printf("VALUE = %s\n", fits.image("Primary").sysheader_value(index));
```

---

### 11.5.5 hdu().sysheader\_comment()

#### NAME

`hdu().sysheader_comment()` — A comment of a system header

#### SYNOPSIS

```
const char *hdu( ... ).sysheader_comment( long index ) const;
const char *hdu( ... ).sysheader_comment( const char *keyword ) const;
```

#### DESCRIPTION

`hdu().sysheader_comment()` member function returns a comment assigned to the system header record specified by `index` or `keyword`.

#### PARAMETER

[I]	<code>index</code>	Index of a header record
[I]	<code>keyword</code>	Keyword of a header record
([I] : input, [O] : output)		

#### RETURN VALUE

`hdu().sysheader_comment()` returns an address of a comment string of a system header record.

#### EXAMPLES

Following code displays the comment of the system header record specified by `index` in the Primary HDU.

---

```
printf("%s\n", fits.image("Primary").sysheader_comment(index));
```

---

### 11.5.6 hdu().sysheader\_formatted\_string()

#### NAME

`hdu().sysheader_formatted_string()` — Formatted string of a system header

#### SYNOPSIS

```
const char *hdu( ... ).sysheader_formatted_string();
```

#### DESCRIPTION

`hdu().sysheader_formatted_string()` member function creates a formatted string outputted to the FITS file inside the internal of the object for all system header records and returns its address.

#### RETURN VALUE

`hdu().sysheader_formatted_string()` returns an address of a formatted string.

#### EXCEPTION

If the API fails to manipulate the internal buffer, it throws an exception derived from SLLIB (`sli::err_rec` exception).

#### EXAMPLES

Following code displays the number of lines and the whole string of the formatted string of all system header records in Primary HDU. Using `tstring` class facilitates manipulation of a string.

```
tstring hdr_all = fits.image("Primary").sysheader_formatted_string();
printf("Number of Lines = %d\n", (int)hdr_all.length() / 80);
printf("Formatted header. :\n");
printf("%s\n", hdr_all.cstr());
```

For more information about tstring class, see APPENDIX1(§12).

---

## 11.6 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.

### 11.6.1 `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());
```

---

### 11.6.2 `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());
```

---

**11.6.3 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());
```

---

**11.6.4 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));
    }

```

---

### 11.6.5 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;
}

```

---

### 11.6.6 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());
```

---

**11.6.7 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());
```

---

**11.6.8 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());
```

---

**11.6.9 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());
```

---

### 11.6.10 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]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	...	all location data of pixels in each axis
[I]	<code>ap</code>	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 §3.8.

### 11.6.11 `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 §11.6.10 and also a sample code in §3.8.

### 11.6.12 `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]	<code>value</code>	a value to be set
[I]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	<code>...</code>	all location data of pixels in each axis
[I]	<code>ap</code>	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 §3.8.

### 11.6.13 `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 §3.11.

### 11.6.14 `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 precision 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 §11.6.13) 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);
}
```

---

## 11.6.15 `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 precision 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 §11.6.13) 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);
}
```

---

**11.6.16 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);
}
```

---

### 11.6.17 `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");
}
```

---

### 11.6.18 `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. When `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]	<code>obj</code>	a Image object copied to new one in initialization
[I]	<code>type</code>	a data type of an image
[I]	<code>naxis0</code>	the number of pixels in a column( <i>x</i> -axis)
[I]	<code>naxis1</code>	the number of pixels in a row( <i>y</i> -axis)
[I]	<code>naxis2</code>	the number of pixels in a layer( <i>z</i> -axis)
[I]	<code>naxis</code>	the number of lists specified by an argument <code>naxisx</code>
[I]	<code>naxisx</code>	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);
```

---

### 11.6.19 `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]	<code>obj</code>	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 §3.10.

---

**11.6.20 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();
```

**11.6.21 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();
```

**11.6.22 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());
```

---

**11.6.23 `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);
```

---

**11.6.24 image().add()****NAME**

`image().add()` — Addition and subtraction 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);
```

---

**11.6.25 image().multiply()****NAME**

`image().multiply()` — Multiplication and division 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] <code>value</code>	a value to be multiplied by pixel values
[I] <code>col_index</code>	a pixel index of a column( <i>x</i> -axis)
[I] <code>col_size</code>	an offset pixels from <code>col_index</code>
[I] <code>row_index</code>	a pixel index of a row( <i>y</i> -axis)
[I] <code>row_size</code>	an offset pixels from <code>row_index</code>
[I] <code>layer_index</code>	a pixel index of a layer( <i>z</i> -axis)
[I] <code>layer_size</code>	layer_size an offset pixels from <code>layer_index</code>
([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);
```

---

**11.6.26 `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,
                               double (*func)(double,double,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`).

When an argument of a column, a row, or a layer is not specified, all columns, rows, and layers are modified.

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

Specifications of arguments in the user-defined function(\*func) are as below:

```
double pix_value ... an original pixel value
double value ... a first argument of image().fill()
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 an this object
void *ptr ... a pointer to user_ptr
return value ... a new pixel value
```

Do not use a constant FITS::ALL explicitly.

#### PARAMETER

[I] value	a pixel value to be set via a user-defined function
[I] func	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( <i>x</i> -axis)
[I] col_size	an offset pixels from col_index
[I] row_index	a pixel index of a row( <i>y</i> -axis)
[I] row_size	an offset pixels from row_index
[I] layer_index	a pixel index of a layer( <i>z</i> -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 double myfunc( double pix_value, double value,
                      long axis0, long axis1, long axis2,
                      fits_image *thisp, void *ptr )
{
    if ( value < pix_value ) return value;
    else return pix_value;
}
:
:
fits_image &primary = fits.image("Primary");
primary.fill( thresh, &myfunc, NULL,
              0, primary.col_length(),
              0, primary.row_length() );
```

### 11.6.27 image().copy(), image().cut()

#### NAME

image().copy(), image().cut() — 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;
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;
fits_image &image( ... ).cut( 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 );
fits_image &image( ... ).cut( 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 );

```

## 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.

`image().cut()` fills the source area with 0 after the copy process.

## PARAMETER

[O] dest_img	an object which specified rectangular areas are stored to
[I] col_index	a pixel index of a column( <i>x</i> -axis)
[I] col_size	an offset pixels from <code>col_index</code>
[I] row_index	a pixel index of a row( <i>y</i> -axis)
[I] row_size	an offset pixels from <code>row_index</code>
[I] layer_index	a pixel index of a layer( <i>z</i> -axis)
[I] layer_size	an offset pixels from <code>layer_index</code>

([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  $128 \times 128$  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 §3.10.

### 11.6.28 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]	<code>src_img</code>	a source object for copy
[I]	<code>col_index</code>	a pixel index of a column( <i>x</i> -axis)
[I]	<code>row_index</code>	a pixel index of a row( <i>y</i> -axis)
[I]	<code>layer_index</code>	a pixel index of a layer( <i>z</i> -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 §11.6.27 and also a sample code in §3.10.

---

### 11.6.29 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 §11.6.27 as below:

---

```
fits.image("Primary").add(my_image_buffer, 128, 0);
```

---

**11.6.30 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 §11.6.29.

---

**11.6.31 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 §11.6.29.

---

**11.6.32 `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 §11.6.29.

---

### 11.6.33 `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,
                               double (*func)(double,double,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 an 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 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.

`user_ptr` is a pointer which a user can use freely, and it is given to a last argument of a user-defined function.

Specifications of arguments in the user-defined function(`*func`) are as below:

<code>double pix_value</code> ...	an original pixel value
<code>double value</code> ...	a pixel value of a copy buffer( <code>src_img</code> )
<code>long axis0</code> ...	a coordinate of <i>x</i> -axis
<code>long axis1</code> ...	a coordinate of <i>y</i> -axis
<code>long axis2</code> ...	a coordinate of <i>z</i> -axis
<code>fits_image *thisp</code> ...	a pointer to an this object
<code>void *ptr</code> ...	a pointer to <code>user_ptr</code>
return value ...	a new pixel value

#### 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 §11.6.26.

## 11.7 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.

### 11.7.1 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 operation with the `mdarray` class.

For more information on the `mdarray` class, see a SLLIB manual.

---

### 11.7.2 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]	<code>axis0</code>	a pixel location in a column( $x$ -axis)
[I]	<code>axis1</code>	a pixel location in a row( $y$ -axis)
[I]	<code>axis2</code>	a pixel location in a layer( $z$ -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	...	all location data of pixels in each axis
[I]	<code>ap</code>	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 §3.11.

---

**11.7.3 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]	<code>dest_buf</code>	an address of an obtained image data
[I]	<code>buf_size</code>	a size of <code>dest_buf</code> in bytes
[I]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	...	all location data of pixels in each axis
[I]	<code>ap</code>	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);
```

---

**11.7.4 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);
```

---

**11.7.5 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 §11.6.10.) 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));
}
```

---

**11.7.6 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 §11.6.10.) 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 §11.7.5.

---

### 11.7.7 `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 §11.6.10) 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]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	...	all location data of pixels in each axis
[I]	<code>ap</code>	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 §11.7.5.

---

### 11.7.8 `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 §11.6.10) 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]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	...	all location data of pixels in each axis
[I]	<code>ap</code>	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 §11.7.5.

---

### 11.7.9 `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 §11.6.10) 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\_axesx 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 §11.7.5.

---

**11.7.10 `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_axesx,
                                         long axis0, long axis1, long axis2, ... ) const;
unsigned char image( ... ).va_byte_value_v( long num_axesx,
                                         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 §11.6.10) 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\_axesx 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 §11.7.5.

---

**11.7.11 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 §11.6.12.) 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);
}
```

---

### 11.7.12 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 §11.6.12) 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( <i>x</i> -axis)
[I] axis1	a pixel location in a row( <i>y</i> -axis)
[I] axis2	a pixel location in a layer( <i>z</i> -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 §11.7.11.

### 11.7.13 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 §11.6.12) 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] <code>value</code>	a value to be set
[I] <code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I] <code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I] <code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I] <code>num_axisx</code>	the number of axes specified as arguments
[I] <code>...</code>	all location data of pixels in each axis
[I] <code>ap</code>	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 §11.7.11.

---

**11.7.14 `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 §11.6.12) 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( <i>x</i> -axis)
[I] axis1	a pixel location in a row( <i>y</i> -axis)
[I] axis2	a pixel location in a layer( <i>z</i> -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 §11.7.11.

---

**11.7.15 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 §11.6.12) 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( <i>x</i> -axis)
[I] axis1	a pixel location in a row( <i>y</i> -axis)
[I] axis2	a pixel location in a layer( <i>z</i> -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 §11.7.11.

---

**11.7.16 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 §11.6.12) 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]	<code>value</code>	a value to be set
[I]	<code>axis0</code>	a pixel location in a column( <i>x</i> -axis)
[I]	<code>axis1</code>	a pixel location in a row( <i>y</i> -axis)
[I]	<code>axis2</code>	a pixel location in a layer( <i>z</i> -axis)
[I]	<code>num_axisx</code>	the number of axes specified as arguments
[I]	<code>...</code>	all location data of pixels in each axis
[I]	<code>ap</code>	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 §11.7.11.

---

## 11.8 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 a only case of the string column (i.e., TTYPE<sub>n</sub> of header is `xA`) 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.

### 11.8.1 `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());
```

### 11.8.2 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]	<b>ver</b>	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());
```

---

### 11.8.3 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());
```

---

### 11.8.4 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());
```

---

### 11.8.5 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);
}
```

---

### 11.8.6 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 11.8.5

---

### 11.8.7 table().col().type()

#### NAME

table().col().type() — Column type

#### SYNOPSIS

```
int table( ... ).col( ... ).type() const;
```

#### DESCRIPTION

table().col().type() return 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::STRING\_T.

#### RETURN VALUE

table().col().type() returns the column type which is defined in SFITSIO constant.

#### 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();
    :
    :
}
```

---

### 11.8.8 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::STRING\_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::STRING\_T, it returns the string length of minimum element in the column determined by specification of TFORMn and TDIMn. Concrete examples are shown in the following table.

<b>TFORM<i>n</i> and TDIM<i>n</i></b>	<b>.bytes()</b>	<b>.dcol_length()</b>	<b>.drow_length()</b>	<b>.elem_length()</b>
TFORM <i>n</i> = '120A'	120	1	1	1
TFORM <i>n</i> = '120A10'	10	12	1	12
TFORM <i>n</i> = '120A10' TDIM <i>n</i> = '(6,2)'	10	6	2	12
TFORM <i>n</i> = '120A' TDIM <i>n</i> = '(10,6,2)'	10	6	2	12

**RETURN VALUE**

`table().col().bytes()` returns the bytes of column type.

**EXAMPLES**

See EXAMPLES of 11.8.7

---

**11.8.9 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 TTYPEn 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 11.8.7

---

**11.8.10 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::STRING\_T – for example, when TTYPEn is 16D, it returns 16 (no relation with TDIMn).

If column type is FITS::STRING\_T, See the table in `table().col().bytes()` (11.8.8)

**RETURN VALUE**

`table().col().elem_length()` returns the number of elements in the column.

**EXAMPLES**

See EXAMPLES of 11.8.7

---

### 11.8.11 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::STRING\_T – for example, TDIM $n$  is (8x2), it returns 8.

If the column type is FITS::STRING\_T, See the table in table().col().bytes() (11.8.8)

#### 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();
```

---

### 11.8.12 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 column type except FITS::STRING\_T, it returns 2 for a column with (8x2) of its TDIM $n$ .

See table (11.8.8) in table().col().bytes(), if column type is FITS::STRING\_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();
```

---

### 11.8.13 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 11.8.33

---

**11.8.14 `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 `TZEROn` and `TSCALn` 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 `TNULLn` in ASCII tables or in integer type columns of binary tables.

`TZEROn` and `TSCALn` value of header is valid when `TFORMn` of binary table includes '`B`', '`I`', '`J`', '`K`', '`E`', or '`D`', or when `TFORMn` of ASCII table includes '`I`', '`L`', '`F`', '`E`', '`G`', or '`D`'.<sup>11)</sup>

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 `TFORMn` of column in ASCII table does not represent numeric value, it directly returns the real number value converted from the string of cell.

If the `TFORMn` of column in ASCII table does represent numeric value, it converts the string of cell to real number value and convert it by `TZEROn` and `TSCALn` 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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

**PARAMETER**

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I]	<code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)		

<sup>11)</sup> SFITSIO supports '`L`' and '`G`' which are not included in the definition of FITS.

**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));
}
```

---

**11.8.15 `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 TZEROn and TSCALn 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 TNULLn of ASCII table or that of integer type column of binary table.

TZEROn and TSCALn value of header is valid when TFORMn of binary table includes 'B', 'I', 'J', 'K', 'E', or 'D', or , TFORMn of ASCII table includes 'I', 'L', 'F', 'E', 'G', or 'D'.<sup>12)</sup>

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 TFORMn 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 TFORMn of column in ASCII table does represent numeric value, it converts the string of cell to real number value and convert it by TZEROn and TSCALn and returns the nearest integer.

<sup>12)</sup> SFITSIO supports 'L' and 'G' which are not included in the definition of FITS.

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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<math>n</math></code> )
[I]	<code>repetition_idx</code>	second dimensional index
([I] : input, [O] : output)		

#### RETURN VALUE

`table().col().lvalue()` and `table().col().llvalue()` return the cell value.

#### EXAMPLES

See EXAMPLES of 11.8.14

---

### 11.8.16 `table().col().bvalue()`

#### NAME

`table().col().bvalue()` — Returns cell as boolean

#### SYNOPSIS

```
bool table( ... ).col( ... ).bvalue( long row_index ) const;
bool table( ... ).col( ... )
    .bvalue( long row_index,
             const char *elem_name, long repetition_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()` (11.9.12).

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 convert 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 TSCALn 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 TNULLn 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 **TELEMn** can be specified.

If **TDIMn** is specified, **elem\_index** can be specified as the first dimensional index, and **repetition\_idx** can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I]	<b>row_index</b>	row index
[I]	<b>elem_name</b>	element name
[I]	<b>elem_index</b>	element index (the first dimension index of <b>TDIMn</b> )
[I]	<b>repetition_index</b>	second dimensional index
([I] : input, [O] : output)		

## RETURN VALUE

**table().col().bvalue()** returns the cell value.

## EXAMPLES

See EXAMPLES in 11.8.14

---

### 11.8.17 **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 repetition_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 where **TFORMn** does not represent numeric value, it returns string of cell which is formatted by **TFORMn**. If **TFORMn** 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 **TSCALn**, and returns the value which is formatted as a string with **TFORMn**.

When **TDISPn** is given in the boolean column, it returns "T", "F", or "U". When **TDISPn** is given, if the value is 'T', then it formats 1 by **TDISPn**, and returns it. If the value is 'F', then it formats 0 by **TDISPn**, and returns it.

If the column type is integer or real number, it converts cell value with TZEROn and TSCALn of the header, and converts it to string and returns it. If TDISPn is specified, it converts with TDISPn and returns it.

When the column type is integer, TZEROn is 0, TSCALn is 1.0, and no TDISPn 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" lll
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 TDISPn. It also returns "NULL" when the value of cell is TNULLn 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 (§11.8.23).

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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIMn</code> )
[I] <code>repetition_index</code>	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 11.8.14

---

### 11.8.18 `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 TZEROn and TSCALn, 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 TZEROn and TSCALn 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, TZEROn is 0, TSCALn 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 TNULLn 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 (§11.8.23).

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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of TDIM $n$ )
[I]	<code>repetition_index</code>	second dimensional index
[O]	<code>dest_buf</code>	address of destination buffer
[I]	<code>buf_size</code>	the size of <code>dest_buf</code>

([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) );
```

---

### 11.8.19 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 TZEROn and TSCALn 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 TNULLn value in integer type column of binary table or ASCII table, it assigns the `value` to the cell.

TZEROn and TSCALn value of header is valid when TFORMn of binary table includes 'B', 'I', 'J', 'K', 'E', or 'D', or , TFORMn of ASCII table includes 'I', 'L', 'F', 'E', 'G', or 'D'.<sup>13)</sup>

If the column type is integer, it converts `value` with TZEROn and TSCALn and assigns the nearest integer to the cell.

<sup>13)</sup> 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. If `NULL(NAN)` is given, it assigns '\0'.

When column of binary table is string type or when the `TFORMn` 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 `TFORMn` of the column of ASCII table does represent numeric `value`, it converts `value` with `TZEROn` and `TSCALn`, formats it with `TFORMn`, 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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified 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] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIM<i>n</i></code> )
[I] <code>repetition_index</code>	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)`

## EXAMPLES

```
double value = 0;
fits.table("EVENT").col("TIME").assign(value, 0);
```

---

## 11.8.20 `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 TZEROn and TSCALn 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 11.8.19 functions.

TZEROn and TSCALn `value` of header is valid when TFORMn of binary table includes 'B', 'I', 'J', 'K', 'E', or 'D', or , TFORMn of ASCII table includes 'I', 'L', 'F', 'E', 'G', or 'D'.<sup>14)</sup>

If the column type is integer, it converts `value` with TZEROn and TSCALn 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.

When column of binary table is string type or when the TFORMn of column in ASCII table does not represent numeric value, it converts with format "%lld" of `printf()`. If TFORMn is specified, it additionally formats it.

If the TFORMn of the column in ASCII table does represent numeric `value`, it converts `value` with TZEROn and TSCALn, formats it with TFORMn, 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 TELEMn can be specified.

If TDIMn is specified, `elem_index` can be specified 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] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of TDIMn)
[I] <code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)	

<sup>14)</sup> SFITSIO supports 'L' and 'G' which are not included in the definition of FITS.

**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 11.8.19

---

**11.8.21 `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 (§11.8.23).

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 TZEROn and TSCALn, 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 TZEROn and TSCALn, 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 specified 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]	repetition_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 11.8.19

---

### 11.8.22 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 TZEROn, TSCALn, TNULLn, 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);
```

---

**11.8.23 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()` (§11.8.1) `table().col().assign()` (§11.8.21).

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).

---

**11.8.24 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 TZEROn.

`table().col().assign_tzero()` assigns the value of TZEROn. `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 TZEROn.

**PARAMETER**

- [I] zero TZERO value to modify
- [I] prec precision (places)
- ([I] : input, [O] : output)

**RETURN VALUE**

`tzero()` returns the value of `TZEROn`.

`tzero_is_set()` returns whether `TZEROn` 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);
}
```

---

**11.8.25 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 `TSCALn`.

`table().col().assign_tscal()` assigns the value of `TSCALn`. `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 `TSCALn`.

**PARAMETER**

- [I] scal TSCAL value to modify
- [I] prec precision (places)
- ([I] : input, [O] : output)

**RETURN VALUE**

`tscal()` returns the value of `TSCALn`.

`tscal_is_set()` returns whether `TSCALn` 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);
}
```

---

**11.8.26 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]	<code>null</code>	TNULL value to set
[O]	<code>tnull_ptr</code>	address 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);
}
```

---

**11.8.27 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 TUNITn.  
`table().col().assign_tunit()` assigns the value of TUNITn.  
`table().col().erase_tunit()` erases configuration of TUNITn.

**PARAMETER**

[I] `unit` TUNIT value to modify  
([I] : input, [O] : output)

**RETURN VALUE**

`tunit()` returns the value of TUNITn.  
`tunit_is_set()` returns whether TUNITn 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");
}
```

---

**11.8.28 `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();
```

---

### 11.8.29 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();
```

---

### 11.8.30 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");
```

---

### 11.8.31 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);
```

---

### 11.8.32 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"));
```

**11.8.33 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);
```

**11.8.34 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);

```

---

### 11.8.35 `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);
```

---

**11.8.36 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 beggining 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);
```

---

### 11.8.37 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;
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() (11.8.45)

#### 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);
```

---

### 11.8.38 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);
```

---

### 11.8.39 `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);
```

---

### 11.8.40 `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);
```

---

**11.8.41 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);
```

---

**11.8.42 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);
```

---

**11.8.43 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);
```

---

**11.8.44 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);
```

---

**11.8.45 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") );
```

---

**11.8.46 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 );
```

---

**11.8.47 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 );
```

---

### 11.8.48 `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]	<code>index</code>	starting <code>index</code> to initialize
[I]	<code>num_rows</code>	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(OL).clean();
```

---

### 11.8.49 `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().import()` 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]	<code>dest_index</code>	destination row index to be imported
[I]	<code>from</code>	source table object to import <code>from</code>
[I]	<code>idx_begin</code>	source row index to be imported
[I]	<code>num_rows</code>	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) );
```

---

**11.9 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.

**11.9.1 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.

**11.9.2 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();
:
:
```

---

### 11.9.3 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);
```

---

### 11.9.4 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);
```

---

**11.9.5 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 `TFORMn`). However, the value returned by these functions **does not reflect TZEROn or TSCALn**.

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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified 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] `repetition_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);
:
:
```

---

**11.9.6 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 `TFORMn`). However, the value returned by these functions **does not reflect TZEROn** or **TSCALn**.

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 specified 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] `repetition_index` second dimensional index
- ([I] : input, [O] : output)

**RETURN VALUE**

`table().col().long_value()` returns the cell value.

**EXAMPLES**

See EXAMPLES of 11.9.5

---

**11.9.7 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 `TFORMn`). However, the value returned by these functions **does not reflect TZEROn or TSCALn**.

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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

**PARAMETER**

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIMn</code> )
[I]	<code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)		

**RETURN VALUE**

`table().col().longlong_value()` returns the cell value.

**EXAMPLES**

See EXAMPLES of 11.9.5

---

**11.9.8 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 `TFORMn`). However, the value returned by these functions **does not reflect TZEROn** or `TSCALn`.

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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIMn</code> )
[I]	<code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)		

## RETURN VALUE

`table().col().byte_value()` returns the cell value.

## EXAMPLES

See EXAMPLES of 11.9.5

---

### 11.9.9 `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 TZEROn or TSCALn**.

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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of TDIM $n$ )
[I]	<code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)		

#### RETURN VALUE

`table().col().float_value()` returns the cell value.

#### EXAMPLES

See EXAMPLES of 11.9.5

---

### 11.9.10 `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 TZEROn or TSCALn**.

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 specified 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] `repetition_index` second dimensional index
- ([I] : input, [O] : output)

**RETURN VALUE**

`table().col().double_value()` returns the cell value.

**EXAMPLES**

See EXAMPLES of 11.9.5

---

**11.9.11 `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 specified 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 §9.6 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** TZEROn or TSCALn.

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] `repetition_index` second dimensional index
- ([I] : input, [O] : output)

**RETURN VALUE**

`table().col().bit_value()` returns the cell value.

**EXAMPLES**

See EXAMPLES of 11.9.5

---

**11.9.12 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 cell 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 `TFORMn`).

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 `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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

**PARAMETER**

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I]	<code>repetition_index</code>	second dimensional index
([I] : input, [O] : output)		

**RETURN VALUE**

`table().col().logical_value()` returns the cell value.

**EXAMPLES**

See EXAMPLES of 11.9.5

---

### 11.9.13 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::STRING\_T (includes 'A' in TFORMn).

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 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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of TDIMn)
[I]	<code>repetition_index</code>	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 11.9.5

---

### 11.9.14 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 TDISP $n$  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 `TELEM $n$`  can be specified.

If `TDIM $n$`  is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index. The index begins with zero.

## PARAMETER

[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<math>n</math></code> )
[I]	<code>repetition_idx</code>	second dimensional index
[O]	<code>dest_buf</code>	destination buffer
[I]	<code>buf_size</code>	size of <code>dest_buf</code>

([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) );

```

### 11.9.15 `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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I] <code>repetition_index</code>	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);

```

---

## 11.9.16 `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 `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 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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

## PARAMETER

[I] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I] <code>repetition_index</code>	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 11.9.15

---

### 11.9.17 `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 long type). This function can access the `value` fastest when the column type is `FITS::LONGLONG_T` (includes 'K' 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 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 specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I]	<code>value</code>	the <code>value</code> to assign
[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<math>n</math></code> )
[I]	<code>repetition_idx</code>	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 11.9.15

---

### 11.9.18 `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 "%hu" 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 specified 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] repetition_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 11.9.15

**11.9.19 `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 specified 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] repetition_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 11.9.15

---

### 11.9.20 `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 `TFORMn`). However, these functions **does not convert** the `value` by `TZEROn` and `TSCALn` 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 "%.<sub>15G</sub>" 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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I]	<code>value</code>	the <code>value</code> to assign
[I]	<code>row_index</code>	row index
[I]	<code>elem_name</code>	element name
[I]	<code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I]	<code>repetition_index</code>	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 example, value formatting failure), it throws an exception derived from SLLIB (`sli::err_rec` exception).

**EXAMPLES**

See EXAMPLES of 11.9.15

---

**11.9.21 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 `TFORMn`).

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 specified 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 `TELEMn`, which is counted from specified element to the right, is used. See also §9.6 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] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIMn</code> )
[I] <code>repetition_index</code>	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 11.9.15

---

**11.9.22 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 `TFORMn`).

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 `TZEROn` and `TSCALn` 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 `TELEMn` can be specified.

If `TDIMn` is specified, `elem_index` can be specified as the first dimensional index, and `repetition_idx` can be specified as the second dimensional index.

The index begins with zero.

**PARAMETER**

[I] <code>value</code>	value to assign
[I] <code>row_index</code>	row index
[I] <code>elem_name</code>	element name
[I] <code>elem_index</code>	element index (the first dimension index of <code>TDIM<sub>n</sub></code> )
[I] <code>repetition_index</code>	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 11.9.15

---

### 11.9.23 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 TZEROn and TSCALn 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 **TELEMn** can be specified.

If **TDIMn** is specified, **elem\_index** can be specified as the first dimensional index, and **repetition\_idx** can be specified as the second dimensional index.

The index begins with zero.

#### PARAMETER

[I] <b>value</b>	value to assign
[I] <b>row_index</b>	row index
[I] <b>elem_name</b>	element name
[I] <b>elem_index</b>	element index (the first dimension index of <b>TDIMn</b> )
[I] <b>repetition_index</b>	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 11.9.15

---

## 12 APPENDIX1: 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.<sup>15)</sup> 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, `.ltrim()` 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`.

<sup>15)</sup> 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>

## 13 APPENDIX2: 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>