| IUCr Home Page | CIF Home Page | CBF/imgCIF | CBFlib |
| NOTICE | GPL | LGPL | imgCIF dictionary |
| Click Here to Make a Donation |

CBFlib

An API for CBF/imgCIF
Crystallographic Binary Files with ASCII Support
Version 0.7.7.3
3 April 2007

by
Paul J. Ellis
Stanford Synchrotron Radiation Laboratory

and
Herbert J. Bernstein
Bernstein + Sons
yaya@bernstein-plus-sons.com

Version History

Version	Date	By	Description
0.1	Apr. 1998	PJE	This was the first CBFlib release. It supported binary CBF files using binary strings.
0.2	Aug. 1998	HJB	This release added ascii imgCIF support using MIME-encoded binary sections, added the option of MIME headers for the binary strings was well. MIME code adapted from mpack 1.5. Added hooks needed for DDL1-style names without categories.
0.3	Sep. 1998	PJE	This release cleaned up the changes made for version 0.2, allowing multi-threaded use of the code, and removing dependence on the mpack package.
0.4	Nov. 1998	HJB	This release merged much of the message digest code into the general file reading and writing to reduce the number of passes. More consistency checking between the MIME header and the binary header was introduced. The size in the MIME header was adjusted to agree with the version 0.2 documentation.
0.5	Dec. 1998	PJE	This release greatly increased the speed of processing by allowing for deferred digest evaluation.
0.6	Jan. 1999	HJB	This release removed the redundant information (binary id, size, compression id) from a binary header when there is a MIME header, removed the unused repeat argument, and made the memory allocation for buffering and tables with many rows sensitive to the current memory allocation already used.
0.6.1	Feb. 2001	HP (per HJB)	This release fixed a memory leak due to misallocation by size of cbf_handle instead of cbf_handle_struct
0.7	Mar. 2001	PJE	This release added high-level instructions based on the imgCIF dictionary version 1.1.
0.7.1	Mar. 2001	PJE	The high-level functions were revised to permit future expansion to files with multiple images.
0.7.2	Apr. 2001	HJB	This release adjusted cbf_cimple.c to conform to cif_img.dic version 1.1.3
0.7.2.1	May 2001	PJE	This release corrected an if nesting error in the prior mod to cbf_cimple.c.
0.7.3	Oct 2002	PJE	This release modified cbf_simple.c to reorder image data on read so that the indices are always increasing in memory (this behavior was undefined previously).
0.7.4	Jan 2004	HJB	This release fixes a parse error for quoted strings, adds code to get and set character string types, and removes compiler warnings
0.7.5	Apr 2006	HJB	This release cleans up some compiler warnings, corrects a parse error on quoted strings with a leading blank as adds the new routines for support of aliases, dictionaries and real arrays, higher level routines to get and set pixel sizes, do cell computations, and to set beam centers, improves support for conversion of images, picking up more data from headers.
0.7.6	Jul 2006	HJB	This release reorganizes the kit into two pieces: CBFlib_0.7.6_Data_Files and CBFlib_0.7.6. An optional local copy of getopt is added. The 1.4 draft dictionary has been added. cif2cbf updated to support vcif2 validation. convert_image and cif2cbf updated to report text of error messages.

			convert_image updated to support tag and category aliases, default to adxv images. convert_image and img updated to support row-major images. Support added for binning. API Support added for validation, wide files and line folding. Logic changed for beam center reporting. Added new routines: cbf_validate, cbf_get_bin_sizes, cbf_set_bin_sizes, cbf_find_last_typed_child, cbf_compose_itemname, cbf_set_cbf_logfile, cbf_make_widefile, cbf_read_anyfile, cbf_read_widefile, cbf_write_local_file, cbf_write_widefile, cbf_column_number, cbf_blockitem_number, cbf_log, cbf_check_category_tags, cbf_set_beam_center
0.7.7	February 2007	HJB	This release reflects changes for base 32K support developed by G. Darakev, and changes for support of reals, 3d arrays, byte_offset compression and J. P. Abrahams packed compression made in consultation with (in alphabetic order) E. Eikenberry, A. Hammerley, W. Kabsch, M. Kobas, J. Wright and others at PSI and ESRF in January 2007, as well accumulated changes fixing problems in release 0.7.6.
0.7.7.1	February 2007	HJB	This release is a patch to 0.7.7 to change the treatment of the byteorder parameter from strcpy semantics to return of a pointer to a string constant. Our thanks to E. Eikenberry for pointing out the problem.
0.7.7.2	February 2007	HJB	This release is a patch to 0.7.7.1 to add testing for JPA packed compression and to respect signs declared in the MIME header.
0.7.7.3	April 2007	HJB	This release is a patch to 0.7.7.3 to add f90 support for reading of CBF byte-offset and packed compression, to fix problems with gcc 4.4.1 and to correct errors in multidimensional packed compression.
0.7.7.4	May 2007	HJB	Corrects in handling SLS detector mincbfs and reorder dimensions versus arrays for some f90 compilers as per H. Powell.

Known Problems

This version does not have support for predictor compression. Code is needed to support array sub-sections.

Foreword

In order to work with CBFlib, you need:

- the source code, in the form of a "gzipped" tar, CBFlib_0.7.7.tar.gz; and
- the test data, in the form of a "gzipped" tar CBFlib_0.7.7_Data_Files.tar.gz

Uncompress both of these files, and unpack them with tar:

- gunzip < CBFlib_0.7.7.tar.gz | tar xvf -
- gunzip < CBFlib_0.7.7_Data_Files.tar.gz | tar xvf -

The data files are compressed with bzip2. **Do not "bunzip2" the files in Place them in an otherwise empty directory, and unpack it with tar.** As in the past you will also need Paul Ellis's sample MAR345 image, example.mar2300 and Chris Nielsen's sample ADSC Quantum 315 image, mb_LP_1_001.img as sample data. The Makefile will extract decompress these files from the CBFlib_0.7.7_Data_Files directory.

Adjust the definition of CC and C++ in Makefile to point to your C and C++ compilers, the definition of CFLAGS to an appropriate value for your C and C++ compilers, the definition of F90C to point to your Fortan-90/95 compiler, and the definitions of F90FLAGS and F90LDFLAGS to approriate values for your Fortan-90/95 compilers, and then

make all
make tests

We have included examples of CBF/imgCIF files produced by CBFlib, the current best draft of the CBF Extensions Dictionary, and of Andy Hammersley's CBF definition, updated to become a DRAFT CBF/ImgCIF DEFINITION.

Contents

- 1. Introduction
- 2. Function descriptions
  - 2.1 General description
    - 2.1.1 CBF handles
    - 2.1.2 CBF goniometer handles
    - 2.1.3 CBF detector handles
    - 2.1.4 Return values
  - 2.2 Reading and writing files containing binary sections
    - 2.2.1 Reading binary sections
    - 2.2.2 Writing binary sections
    - 2.2.3 Summary of reading and writing files containing binary sections
  - 2.3 Low-level function prototypes
    - 2.3.1 cbf_make_handle
    - 2.3.2 cbf_free_handle
    - 2.3.3 cbf_read_file, cbf_read_widefile
    - 2.3.4 cbf_write_file, cbf_write_widefile
    - 2.3.5 cbf_new_datablock, cbf_new_saveframe
    - 2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
    - 2.3.7 cbf_new_category
    - 2.3.8 cbf_force_new_category
    - 2.3.9 cbf_new_column
    - 2.3.10 cbf_new_row
    - 2.3.11 cbf_insert_row
    - 2.3.12 cbf_delete_row
    - 2.3.13 cbf_set_datablockname, cbf_set_saveframename
    - 2.3.14 cbf_reset_datablocks
    - 2.3.15 cbf_reset_datablock, cbf_reset_saveframe
    - 2.3.16 cbf_reset_category
    - 2.3.17 cbf_remove_datablock, cbf_remove_saveframe
    - 2.3.18 cbf_remove_category
    - 2.3.19 cbf_remove_column
    - 2.3.20 cbf_remove_row
    - 2.3.21 cbf_rewind_datablock
    - 2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
    - 2.3.23 cbf_rewind_column
    - 2.3.24 cbf_rewind_row
    - 2.3.25 cbf_next_datablock
    - 2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
    - 2.3.27 cbf_next_column

- 2.3.28 cbf_next_row
- 2.3.29 cbf_find_datablock
- 2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
- 2.3.31 cbf_find_column
- 2.3.32 cbf_find_row
- 2.3.33 cbf_find_nextrow
- 2.3.34 cbf_count_datablocks
- 2.3.35 cbf_count_categories, cbf_count_saveframes, cbf_count_blockitems
- 2.3.36 cbf_count_columns
- 2.3.37 cbf_count_rows
- 2.3.38 cbf_select_datablock
- 2.3.39 cbf_select_category, cbf_select_saveframe, cbf_select_blockitem
- 2.3.40 cbf_select_column
- 2.3.41 cbf_select_row
- 2.3.42 cbf_datablock_name
- 2.3.43 cbf_category_name
- 2.3.44 cbf_column_name
- 2.3.45 cbf_row_number
- 2.3.46 cbf_get_value, cbf_require_value
- 2.3.47 cbf_set_value
- 2.3.48 cbf_get_typeofvalue
- 2.3.49 cbf_set_typeofvalue
- 2.3.50 cbf_get_integervalue, cbf_require_integervalue
- 2.3.51 cbf_set_integervalue
- 2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
- 2.3.53 cbf_set_doublevalue
- 2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
- 2.3.55 cbf_get_integerarray, cbf_get_realarray
- 2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
- 2.3.57 cbf_failnez
- 2.3.58 cbf_onfailnez
- 2.3.59 cbf_require_datablock
- 2.3.60 cbf_require_category
- 2.3.61 cbf_require_column
- 2.3.62 cbf_require_column_value
- 2.3.63 cbf_require_column_integervalue
- 2.3.64 cbf_require_column_doublevalue
- 2.3.65 cbf_get_local_integer_byte_order, cbf_get_local_real_byte_order, cbf_get_local_real_format
- 2.3.66 cbf_get_dictionary, cbf_set_dictionary, cbf_require_dictionary
- 2.3.67 cbf_convert_dictionary
- 2.3.68 cbf_find_tag, cbf_find_local_tag
- 2.3.69 cbf_find_category_root, cbf_set_category_root, cbf_require_category_root
- 2.3.70 cbf_find_tag_root, cbf_set_tag_root, cbf_require_tag_root
- 2.3.71 cbf_find_tag_category, cbf_set_tag_category

2.4 High-level function prototypes (new for version 0.7)

- 2.4.1 cbf_read_template
- 2.4.2 cbf_get_diffrn_id, cbf_require_diffrn_id
- 2.4.3 cbf_set_diffrn_id
- 2.4.4 cbf_get_crystal_id
- 2.4.5 cbf_set_crystal_id
- 2.4.6 cbf_get_wavelength

- 2.4.7 cbf_set_wavelength
- 2.4.8 cbf_get_polarization
- 2.4.9 cbf_set_polarization
- 2.4.10 cbf_get_divergence
- 2.4.11 cbf_set_divergence
- 2.4.12 cbf_count_elements
- 2.4.13 cbf_get_element_id
- 2.4.14 cbf_get_gain
- 2.4.15 cbf_set_gain
- 2.4.16 cbf_get_overload
- 2.4.17 cbf_set_overload
- 2.4.18 cbf_get_integration_time
- 2.4.19 cbf_set_integration_time
- 2.4.20 cbf_get_time
- 2.4.21 cbf_set_time
- 2.4.22 cbf_get_date
- 2.4.23 cbf_set_date
- 2.4.24 cbf_set_current_time
- 2.4.25 cbf_get_image_size, cbf_get_3d_image_size
- 2.4.26 cbf_get_image, cbf_get_real_image, cbf_get_3d_image, cbf_get_real_3d_image
- 2.4.27 cbf_set_image, cbf_set_real_image, cbf_set_3d_image, cbf_set_real_3d_image
- 2.4.28 cbf_get_axis_setting
- 2.4.29 cbf_set_axis_setting
- 2.4.30 cbf_construct_goniometer
- 2.4.31 cbf_free_goniometer
- 2.4.32 cbf_get_rotation_axis
- 2.4.33 cbf_get_rotation_range
- 2.4.34 cbf_rotate_vector
- 2.4.35 cbf_get_reciprocal
- 2.4.36 cbf_construct_detector
- 2.4.37 cbf_free_detector
- 2.4.38 cbf_get_beam_center, cbf_set_beam_center
- 2.4.39 cbf_get_detector_distance
- 2.4.40 cbf_get_detector_normal
- 2.4.41 cbf_get_pixel_coordinates
- 2.4.42 cbf_get_pixel_normal
- 2.4.43 cbf_get_pixel_area
- 2.4.44 cbf_get_pixel_size
- 2.4.45 cbf_set_pixel_size
- 2.4.46 cbf_get_inferred_pixel_size
- 2.4.47 cbf_get_unit_cell
- 2.4.48 cbf_set_unit_cell
- 2.4.49 cbf_get_reciprocal_cell
- 2.4.50 cbf_set_reciprocal_cell
- 2.4.51 cbf_compute_cell_volume
- 2.4.52 cbf_compute_reciprocal_cell
- 2.4.53 cbf_get_orientation_matrix, cbf_set_orientation_matrix
- 2.4.54 cbf_get_bin_sizes, cbf_set_bin_sizes

2.5 F90 function interfaces

- 2.5.1 FCB_ATOL_WCNT
- 2.5.2 FCB_CI_STRNCMPARR
- 2.5.3 FCB_EXIT_BINARY
- 2.5.4 FCB_NBLEN_ARRAY

- 2.5.5 FCB_NEXT_BINARY
- 2.5.6 FCB_OPEN_CIFIN
- 2.5.7 FCB_PACKED: FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2, FCB_DECOMPRESS_PACKED_3D_I4
- 2.5.8 FCB_READ_BITS
- 2.5.9 FCB_READ_BYTE
- 2.5.10 FCB_READ_IMAGE_I2, FCB_READ_IMAGE_I4, FCB_READ_IMAGE_3D_I2, FCB_READ_IMAGE_3D_I4
- 2.5.11 FCB_READ_LINE
- 2.5.12 FCB_READ_XDS_I2
- 2.5.13 FCB_SKIP_WHITESPACE

- 3. File format
  - 3.1 General description
  - 3.2 Format of the binary sections
    - 3.2.1 Format of imgCIF binary sections
    - 3.2.2 Format of CBF binary sections
  - 3.3 Compression schemes
    - 3.3.1 Canonical-code compression
    - 3.3.2 CCP4-style compression
    - 3.3.3 Byte_offset compression
- 4. Installation
- 5. Example programs

1. Introduction

CBFlib (Crystallographic Binary File library) is a library of ANSI-C functions providing a simple mechanism for accessing Crystallographic Binary Files (CBF files) and Image-supporting CIF (imgCIF) files. The CBFlib API is loosely based on the CIFPARSE API for mmCIF files. Like CIFPARSE, CBFlib does not perform any semantic integrity checks; rather it simply provides functions to create, read, modify and write CBF binary data files and imgCIF ASCII data files.

Starting with version 0.7.7, an envolving FCBlib (Fortran Crystallographic Binary library) has been added. As of this release it includes code for reading byte-offset and packed compression image files created by CBFlib.

2. Function descriptions

2.1 General description

Almost all of the CBFlib functions receive a value of type cbf_handle (a CBF handle) as the first argument. Several of the high-level CBFlib functions dealing with geometry receive a value of type cbf_goniometer (a handle for a CBF goniometer object) or cbf_detector (a handle for a CBF detector object).

All functions return an integer equal to 0 for success or an error code for failure.

2.1.1 CBF handles

CBFlib permits a program to use multiple CBF objects simultaneously. To identify the CBF object on which a function will operate, CBFlib uses a value of type cbf_handle.

All functions in the library except cbf_make_handle expect a value of type cbf_handle as the first

argument.

The function **cbf_make_handle** creates and initializes a new CBF handle.

The function **cbf_free_handle** destroys a handle and frees all memory associated with the corresponding CBF object.

2.1.2 CBF goniometer handles

To represent the goniometer used to orient a sample, CBFlib uses a value of type cbf_goniometer.

A goniometer object is created and initialized from a CBF object using the function **cbf_construct_goniometer**.

The function **cbf_free_goniometer** destroys a goniometer handle and frees all memory associated with the corresponding object.

2.1.3 CBF detector handles

To represent a detector surface mounted on a positioning system, CBFlib uses a value of type cbf_detector.

A goniometer object is created and initialized from a CBF object using the function **cbf_construct_detector**.

The function **cbf_free_detector** destroys a detector handle and frees all memory associated with the corresponding object.

2.1.4 Return values

All of the CBFlib functions return 0 on success and an error code on failure. The error codes are:

CBF_FORMAT	The file format is invalid
CBF_ALLOC	Memory allocation failed
CBF_ARGUMENT	Invalid function argument
CBF_ASCII	The value is ASCII (not binary)
CBF_BINARY	The value is binary (not ASCII)
CBF_BITCOUNT	The expected number of bits does not match the actual number written
CBF_ENDOFDATA	The end of the data was reached before the end of the array
CBF_FILECLOSE	File close error
CBF_FILEOPEN	File open error
CBF_FILEREAD	File read error
CBF_FILESEEK	File seek error
CBF_FILETELL	File tell error
CBF_FILEWRITE	File write error

CBF_IDENTICAL	A data block with the new name already exists
CBF_NOTFOUND	The data block, category, column or row does not exist
CBF_OVERFLOW	The number read cannot fit into the destination argument. The destination has been set to the nearest value.
CBF_UNDEFINED	The requested number is not defined (e.g. 0/0; new for version 0.7).
CBF_NOTIMPLEMENTED	The requested functionality is not yet implemented (New for version 0.7).

If more than one error has occurred, the error code is the logical OR of the individual error codes.

2.2 Reading and writing files containing binary sections

2.2.1 Reading binary sections

The current version of CBFlib only decompresses a binary section from disk when requested by the program.

When a file containing one or more binary sections is read, CBFlib saves the file pointer and the position of the binary section within the file and then jumps past the binary section. When the program attempts to access the binary data, CBFlib sets the file position back to the start of the binary section and then reads the data.

For this scheme to work:

1. The file must be a random-access file opened in binary mode (fopen (," rb")).
2. The program *must not* close the file. CBFlib will close the file using fclose ( ) when it is no longer needed.

At present, this also means that a program cant read a file and then write back to the same file. This restriction will be eliminated in a future version.

When reading an imgCIF vs a CBF, the difference is detected automatically.

2.2.2 Writing binary sections

When a program passes CBFlib a binary value, the data is compressed to a temporary file. If the CBF object is subsequently written to a file, the data is simply copied from the temporary file to the output file.

The output file can be of any type. If the program indicates to CBFlib that the file is a random-access and readable, CBFlib will conserve disk space by closing the temporary file and using the output file as the location at which the binary value is stored.

For this option to work:

1. The file must be a random-access file opened in binary update mode (fopen (, "w+b")).
2. The program *must not* close the file. CBFlib will close the file using fclose ( ) when it is no longer needed.

If this option is not used:

1. CBFlib will continue using the temporary file.
2. CBFlib *will not* close the file. This is the responsibility of the main program.

2.2.3 Summary of reading and writing files containing binary sections

1. Open disk files to read using the mode "rb".
2. If possible, open disk files to write using the mode "w+b" and tell CBFlib that it can use the file as a buffer.
3. Do *not* close any files read by CBFlib or written by CBFlib with buffering turned on.
4. Do *not* attempt to read from a file, then write to the same file.

2.3 Low-level function prototypes

2.3.1 cbf_make_handle

PROTOTYPE

#include "cbf.h"

int cbf_make_handle (cbf_handle **handle*);

DESCRIPTION

cbf_make_handle creates and initializes a new internal CBF object. All other CBFlib functions operating on this object receive the CBF handle as the first argument.

ARGUMENTS

handle Pointer to a CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.2 cbf_free_handle

2.3.2 cbf_free_handle

PROTOTYPE

#include "cbf.h"

int cbf_free_handle (cbf_handle *handle*);

DESCRIPTION

cbf_free_handle destroys the CBF object specified by the *handle* and frees all associated memory.

ARGUMENTS

handle CBF handle to free.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.1 cbf_make_handle

2.3.3 cbf_read_file

PROTOTYPE

#include "cbf.h"

int cbf_read_file (cbf_handle *handle*, FILE **file*, int *headers*);
int cbf_read_widefile (cbf_handle *handle*, FILE **file*, int *headers*);

DESCRIPTION

cbf_read_file reads the CBF or CIF file *file* into the CBF object specified by *handle*, using the CIF 1.0 convention of 80 character lines. cbf_read_widefile reads the CBF or CIF file *file* into the CBF object specified by *handle*, using the CIF 1.1 convention of 2048 character lines. A warning is issued to stderr for ascii lines over the limit. No test is performed on binary sections.

Validation is performed in three ways levels: during the lexical scan, during the parse, and, if a dictionary was converted, against the value types, value enumerations, categories and parent-child relationships specified in the dictionary.

headers controls the interprestation of binary section headers of imgCIF files.

MSG_DIGEST: Instructs CBFlib to check that the digest of the binary section matches any header value. If the digests do not match, the call will return CBF_FORMAT. This evaluation and comparison is delayed (a "lazy" evaluation) to ensure maximal processing efficiency. If an immediately evaluation is required, see MSG_DIGESTNOW, below.

MSG_DIGESTNOW: Instructs CBFlib to check that the digest of the binary section matches any header value. If the digests do not match, the call will return CBF_FORMAT. This evaluation and comparison is performed during initial parsing of the section to ensure timely error reporting at the expense of processing efficiency. If a more efficient delayed ("lazy") evaluation is required, see MSG_DIGESTNOW, below.

MSG_NODIGEST: Do not check the digest (default).

CBFlib defers reading binary sections as long as possible. In the current version of CBFlib, this means that:

1. The file must be a random-access file opened in binary mode (fopen (, "rb")).

2. The program *must not* close the file. CBFlib will close the file using fclose ( ) when it is no longer needed.

These restrictions may change in a future release.

ARGUMENTS

handle CBF handle.
file Pointer to a file descriptor.
headers Controls interprestation of binary section headers.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.4 cbf_write_file

2.3.4 cbf_write_file

PROTOTYPE

#include "cbf.h"

int cbf_write_file (cbf_handle *handle*, FILE **file*, int *readable*, int *ciforcbf*, int *headers*, int *encoding*);
int cbf_write_widefile (cbf_handle *handle*, FILE **file*, int *readable*, int *ciforcbf*, int *headers*, int *encoding*);

DESCRIPTION

cbf_write_file writes the CBF object specified by *handle* into the file *file*, following CIF 1.0 conventions of 80 character lines. cbf_write_widefile writes the CBF object specified by *handle* into the file *file*, following CIF 1.1 conventions of 2048 character lines. A warning is issued to stderr for ascii lines over the limit, and an attempt is made to fold lines to fit. No test is performed on binary sections.

If a dictionary has been provided, aliases will be applied on output.

Unlike cbf_read_file, the *file* does not have to be random-access.

If the file is random-access and readable, *readable* can be set to non-0 to indicate to CBFlib that the file can be used as a buffer to conserve disk space. If the file is not random-access or not readable, *readable* must be 0.

If *readable* is non-0, CBFlib will close the file when it is no longer required, otherwise this is the responsibility of the program.

ciforcbf selects the format in which the binary sections are written:

CIF Write an imgCIF file.
CBF Write a CBF file (default).

headers selects the type of header used in CBF binary sections and selects whether message digests are generated. The value of *headers* can be a logical OR of any of:

MIME_HEADERS	Use MIME-type headers (default).
MIME_NOHEADERS	Use a simple ASCII headers.
MSG_DIGEST	Generate message digests for binary data validation.
MSG_NODIGEST	Do not generate message digests (default).

encoding selects the type of encoding used for binary sections and the type of line-termination in imgCIF files. The value can be a logical OR of any of:

ENC_BASE64	Use BASE64 encoding (default).
ENC_QP	Use QUOTED-PRINTABLE encoding.
ENC_BASE8	Use BASE8 (octal) encoding.
ENC_BASE10	Use BASE10 (decimal) encoding.
ENC_BASE16	Use BASE16 (hexadecimal) encoding.
ENC_FORWARD	For BASE8, BASE10 or BASE16 encoding, map bytes to words forward (1234) (default on little-endian machines).
ENC_BACKWARD	Map bytes to words backward (4321) (default on big-endian machines).
ENC_CRTERM	Terminate lines with CR.
ENC_LFTERM	Terminate lines with LF (default).

ARGUMENTS

handle	CBF handle.
file	Pointer to a file descriptor.
readable	If non-0: this file is random-access and readable and can be used as a buffer.
ciforcbf	Selects the format in which the binary sections are written (CIF/CBF).
headers	Selects the type of header in CBF binary sections and message digest generation.
encoding	Selects the type of encoding used for binary sections and the type of line-termination in imgCIF files.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.3 cbf_read_file

2.3.5 cbf_new_datablock, cbf_new_saveframe

PROTOTYPE

#include "cbf.h"

int cbf_new_datablock (cbf_handle *handle*, const char **datablockname*);
int cbf_new_saveframe (cbf_handle *handle*, const char **saveframename*);

DESCRIPTION

cbf_new_datablock creates a new data block with name *datablockname* and makes it the current data block. cbf_new_saveframe creates a new save frame with name *saveframename* within the current data

block and makes the new save frame the current save frame.

If a data block or save frame with this name already exists, the existing data block or save frame becomes the current data block or save frame.

ARGUMENTS

handle	CBF handle.
datablockname	The name of the new data block.
saveframename	The name of the new save frame.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.9 cbf_new_column
2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.12 cbf_set_datablockname, cbf_set_saveframename
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe

PROTOTYPE

#include "cbf.h"

int cbf_force_new_datablock (cbf_handle *handle*, const char **datablockname*);
int cbf_force_new_saveframe (cbf_handle *handle*, const char **saveframename*);

DESCRIPTION

cbf_force_new_datablock creates a new data block with name *datablockname* and makes it the current data block. Duplicate data block names are allowed. cbf_force_new_saveframe creates a new savew frame with name *saveframename* and makes it the current save frame. Duplicate save frame names are allowed.

Even if a save frame with this name already exists, a new save frame is created and becomes the current save frame.

ARGUMENTS

handle	CBF handle.
datablockname	The name of the new data block.
saveframename	The name of the new save frame.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.9 cbf_new_column
2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.12 cbf_set_datablockname, cbf_set_saveframename
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.7 cbf_new_category

PROTOTYPE

#include "cbf.h"

int cbf_new_category (cbf_handle *handle*, const char **categoryname*);

DESCRIPTION

cbf_new_category creates a new category in the current data block with name *categoryname* and makes it the current category.

If a category with this name already exists, the existing category becomes the current category.

ARGUMENTS

handle	CBF handle.
categoryname	The name of the new category.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.8 cbf_force_new_category
2.3.9 cbf_new_column
2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.18 cbf_remove_category
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category

2.3.61 cbf_require_column

2.3.8 cbf_force_new_category

PROTOTYPE

#include "cbf.h"

int cbf_force_new_category (cbf_handle *handle*, const char **categoryname*);

DESCRIPTION

cbf_force_new_category creates a new category in the current data block with name *categoryname* and makes it the current category. Duplicate category names are allowed.

Even if a category with this name already exists, a new category of the same name is created and becomes the current category. The allows for the creation of unlooped tag/value lists drawn from the same category.

ARGUMENTS

handle	CBF handle.
categoryname	The name of the new category.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.7 cbf_new_category
2.3.9 cbf_new_column
2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.18 cbf_remove_category
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.9 cbf_new_column

PROTOTYPE

#include "cbf.h"

int cbf_new_column (cbf_handle *handle*, const char **columnname*);

DESCRIPTION

cbf_new_column creates a new column in the current category with name *columnname* and makes it the current column.

If a column with this name already exists, the existing column becomes the current category.

ARGUMENTS

handle	CBF handle.
columnname	The name of the new column.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.19 cbf_remove_column
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.10 cbf_new_row

PROTOTYPE

#include "cbf.h"

int cbf_new_row (cbf_handle *handle*);

DESCRIPTION

cbf_new_row adds a new row to the current category and makes it the current row.

ARGUMENTS

handle	CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.9 cbf_new_column

2.3.11 cbf_insert_row
2.3.12 cbf_delete_row
2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.11 cbf_insert_row

PROTOTYPE

#include "cbf.h"

int cbf_insert_row (cbf_handle *handle*, unsigned int *rownumber*);

DESCRIPTION

cbf_insert_row adds a new row to the current category. The new row is inserted as row *rownumber* and existing rows starting from *rownumber* are moved up by 1. The new row becomes the current row.

If the category has fewer than *rownumber* rows, the function returns CBF_NOTFOUND.

The row numbers start from 0.

ARGUMENTS

handle	CBF handle.
rownumber	The row number of the new row.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.9 cbf_new_column
2.3.10 cbf_new_row
2.3.12 cbf_delete_row
2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.12 cbf_delete_row

PROTOTYPE

```
#include "cbf.h"

int cbf_delete_row (cbf_handle handle, unsigned int rownumber);
```

DESCRIPTION

cbf_delete_row deletes a row from the current category. Rows starting from *rownumber* +1 are moved down by 1. If the current row was higher than *rownumber*, or if the current row is the last row, it will also move down by 1.

The row numbers start from 0.

ARGUMENTS

handle	CBF handle.
rownumber	The number of the row to delete.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.18 cbf_remove_category
2.3.19 cbf_remove_column
2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.13 cbf_set_datablockname, cbf_set_saveframename

PROTOTYPE

```
#include "cbf.h"

int cbf_set_datablockname (cbf_handle handle, const char *datablockname);
int cbf_set_saveframename (cbf_handle handle. const char *saveframename);
```

DESCRIPTION

cbf_set_datablockname changes the name of the current data block to *datablockname*.
cbf_set_saveframename changes the name of the current save frame to *saveframename*.

If a data block or save frame with this name already exists (comparison is case-insensitive), the function returns CBF_IDENTICAL.

ARGUMENTS

handle	CBF handle.

datablockname	The new data block name.
datablockname	The new save frame name.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.14 cbf_reset_datablocks
2.3.15 cbf_reset_datablock, cbf_reset_saveframe
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.42 cbf_datablock_name

2.3.14 cbf_reset_datablocks

PROTOTYPE

```
#include "cbf.h"

int cbf_reset_datablocks (cbf_handle handle);
```

DESCRIPTION

cbf_reset_datablocks deletes all categories from all data blocks.

The current data block does not change.

ARGUMENTS

handle	CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.15 cbf_reset_datablock, cbf_reset_saveframe
2.3.18 cbf_remove_category

2.3.15 cbf_reset_datablock, cbf_reset_datablock

PROTOTYPE

```
#include "cbf.h"

int cbf_reset_datablock (cbf_handle handle);
int cbf_reset_saveframe (cbf_handle handle);
```

DESCRIPTION

cbf_reset_datablock deletes all categories from the current data block. cbf_reset_saveframe deletes all categories from the current save frame.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.14 cbf_reset_datablocks
2.3.18 cbf_remove_category

2.3.16 cbf_reset_category

PROTOTYPE

#include "cbf.h"

int cbf_reset_category (cbf_handle *handle*);

DESCRIPTION

cbf_reset_category deletes all columns and rows from current category.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.16 cbf_reset_category
2.3.19 cbf_remove_column
2.3.20 cbf_remove_row

2.3.17 cbf_remove_datablock, cbf_remove_saveframe

PROTOTYPE

#include "cbf.h"

int cbf_remove_datablock (cbf_handle *handle*);
int cbf_remove_saveframe (cbf_handle *handle*);

DESCRIPTION

cbf_remove_datablock deletes the current data block. cbf_remove_saveframe deletes the current save frame.

The current data block becomes undefined.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.5 cbf_new_datablock, cbf_new_saveframe
2.3.6 cbf_force_new_datablock, cbf_force_new_saveframe
2.3.18 cbf_remove_category
2.3.19 cbf_remove_column
2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.18 cbf_remove_category

PROTOTYPE

#include "cbf.h"

int cbf_remove_category (cbf_handle *handle*);

DESCRIPTION

cbf_remove_category deletes the current category.

The current category becomes undefined.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.7 cbf_new_category
2.3.8 cbf_force_new_category
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.19 cbf_remove_column

2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.19 cbf_remove_column

PROTOTYPE

#include "cbf.h"

int cbf_remove_column (cbf_handle *handle*);

DESCRIPTION

cbf_remove_column deletes the current column.

The current column becomes undefined.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.9 cbf_new_column
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.18 cbf_remove_category
2.3.20 cbf_remove_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.20 cbf_remove_row

PROTOTYPE

#include "cbf.h"

int cbf_remove_row (cbf_handle *handle*);

DESCRIPTION

cbf_remove_row deletes the current row in the current category.

If the current row was the last row, it will move down by 1, otherwise, it will remain the same.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.10 cbf_new_row
2.3.11 cbf_insert_row
2.3.17 cbf_remove_datablock, cbf_remove_saveframe
2.3.18 cbf_remove_category
2.3.19 cbf_remove_column
2.3.12 cbf_delete_row
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.21 cbf_rewind_datablock

PROTOTYPE

#include "cbf.h"

int cbf_rewind_datablock (cbf_handle *handle*);

DESCRIPTION

cbf_rewind_datablock makes the first data block the current data block.

If there are no data blocks, the function returns CBF_NOTFOUND.

The current category becomes undefined.

ARGUMENTS

handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.19 cbf_rewind_column
2.3.24 cbf_rewind_row
2.3.25 cbf_next_datablock

2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem

PROTOTYPE

#include "cbf.h"

int cbf_rewind_category (cbf_handle *handle*);
int cbf_rewind_saveframe (cbf_handle *handle*);
int cbf_rewind_blockitem (cbf_handle *handle*);

DESCRIPTION

cbf_rewind_category makes the first category in the current data block the current category.
cbf_rewind_saveframe makes the first saveframe in the current data block the current saveframe.
cbf_rewind_blockitem makes the first blockitem (category or saveframe) in the current data block the current blockitem.

If there are no categories, saveframes or blockitems the function returns CBF_NOTFOUND.

The current column and row become undefined.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.19 cbf_rewind_column
2.3.24 cbf_rewind_row
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem

---

2.3.23 cbf_rewind_column

PROTOTYPE

#include "cbf.h"

int cbf_rewind_column (cbf_handle *handle*);

DESCRIPTION

cbf_rewind_column makes the first column in the current category the current column.

If there are no columns, the function returns CBF_NOTFOUND.

The current row is not affected.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.24 cbf_rewind_row
2.3.27 cbf_next_column

2.3.24 cbf_rewind_row

PROTOTYPE

#include "cbf.h"

int cbf_rewind_row (cbf_handle *handle*);

DESCRIPTION

cbf_rewind_row makes the first row in the current category the current row.

If there are no rows, the function returns CBF_NOTFOUND.

The current column is not affected.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.19 cbf_rewind_column
2.3.28 cbf_next_row

2.3.25 cbf_next_datablock

PROTOTYPE

#include "cbf.h"

int cbf_next_datablock (cbf_handle *handle*);

DESCRIPTION

cbf_next_datablock makes the data block following the current data block the current data block.

If there are no more data blocks, the function returns CBF_NOTFOUND.

The current category becomes undefined.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
2.3.27 cbf_next_column
2.3.28 cbf_next_row

2.3.26 cbf_next_category

PROTOTYPE

#include "cbf.h"

int cbf_next_category (cbf_handle *handle*);

DESCRIPTION

cbf_next_category makes the category following the current category in the current data block the current category.

If there are no more categories, the function returns CBF_NOTFOUND.

The current column and row become undefined.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.25 cbf_next_datablock
2.3.27 cbf_next_column
2.3.27 cbf_next_row

2.3.27 cbf_next_column

PROTOTYPE

#include "cbf.h"

int cbf_next_column (cbf_handle *handle*);

DESCRIPTION

cbf_next_column makes the column following the current column in the current category the current column.

If there are no more columns, the function returns CBF_NOTFOUND.

The current row is not affected.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.19 cbf_rewind_column
2.3.25 cbf_next_datablock
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
2.3.28 cbf_next_row

2.3.28 cbf_next_row

PROTOTYPE

#include "cbf.h"

int cbf_next_row (cbf_handle *handle*);

DESCRIPTION

cbf_next_row makes the row following the current row in the current category the current row.

If there are no more rows, the function returns CBF_NOTFOUND.

The current column is not affected.

ARGUMENTS
handle CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.24 cbf_rewind_row
2.3.25 cbf_next_datablock
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
2.3.27 cbf_next_column

2.3.29 cbf_find_datablock

PROTOTYPE

#include "cbf.h"

int cbf_find_datablock (cbf_handle *handle*, const char **datablockname*);

DESCRIPTION

cbf_find_datablock makes the data block with name *datablockname* the current data block.

The comparison is case-insensitive.

If the data block does not exist, the function returns CBF_NOTFOUND.

The current category becomes undefined.

ARGUMENTS

handle	CBF handle.
datablockname	The name of the data block to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.25 cbf_next_datablock
2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.31 cbf_find_column
2.3.32 cbf_find_row
2.3.42 cbf_datablock_name
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.30 cbf_find_category

PROTOTYPE

#include "cbf.h"

int cbf_find_category (cbf_handle *handle*, const char **categoryname*);

DESCRIPTION

cbf_find_category makes the category in the current data block with name *categoryname* the current category.

The comparison is case-insensitive.

If the category does not exist, the function returns CBF_NOTFOUND.

The current column and row become undefined.

ARGUMENTS

handle	CBF handle.
categoryname	The name of the category to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
2.3.29 cbf_find_datablock
2.3.31 cbf_find_column
2.3.32 cbf_find_row
2.3.43 cbf_category_name
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.31 cbf_find_column

PROTOTYPE

#include "cbf.h"

int cbf_find_column (cbf_handle *handle*, const char **columnname*);

DESCRIPTION

cbf_find_column makes the columns in the current category with name *columnname* the current column.

The comparison is case-insensitive.

If the column does not exist, the function returns CBF_NOTFOUND.

The current row is not affected.

ARGUMENTS

handle CBF handle.
columnname The name of column to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.19 cbf_rewind_column
2.3.27 cbf_next_column
2.3.29 cbf_find_datablock
2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.32 cbf_find_row
2.3.44 cbf_column_name
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.32 cbf_find_row

PROTOTYPE

#include "cbf.h"

int cbf_find_row (cbf_handle *handle*, const char **value*);

DESCRIPTION

cbf_find_row makes the first row in the current column with value *value* the current row.

The comparison is case-sensitive.

If a matching row does not exist, the function returns CBF_NOTFOUND.

The current column is not affected.

ARGUMENTS

handle CBF handle.
value The value of the row to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.24 cbf_rewind_row
2.3.28 cbf_next_row
2.3.29 cbf_find_datablock

2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.31 cbf_find_column
2.3.33 cbf_find_nextrow
2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue

2.3.33 cbf_find_nextrow

PROTOTYPE

#include "cbf.h"

int cbf_find_nextrow (cbf_handle *handle*, const char **value*);

DESCRIPTION

cbf_find_nextrow makes the makes the next row in the current column with value *value* the current row. The search starts from the row following the last row found with cbf_find_row or cbf_find_nextrow, or from the current row if the current row was defined using any other function.

The comparison is case-sensitive.

If no more matching rows exist, the function returns CBF_NOTFOUND.

The current column is not affected.

ARGUMENTS

handle CBF handle.
value the value to search for.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.24 cbf_rewind_row
2.3.28 cbf_next_row
2.3.29 cbf_find_datablock
2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.31 cbf_find_column
2.3.32 cbf_find_row
2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue

2.3.34 cbf_count_datablocks

PROTOTYPE

#include "cbf.h"

int cbf_count_datablocks (cbf_handle *handle*, unsigned int **datablocks*);

DESCRIPTION

cbf_count_datablocks puts the number of data blocks in **datablocks* .

ARGUMENTS

handle CBF handle.
datablocks Pointer to the destination data block count.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.35 cbf_count_categories, cbf_count_saveframes, cbf_count_blockitems
2.3.36 cbf_count_columns
2.3.37 cbf_count_rows
2.3.38 cbf_select_datablock

2.3.35 cbf_count_categories

PROTOTYPE

#include "cbf.h"

int cbf_count_categories (cbf_handle *handle*, unsigned int **categories*);

DESCRIPTION

cbf_count_categories puts the number of categories in the current data block in **categories*.

ARGUMENTS

handle CBF handle.
categories Pointer to the destination category count.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.34 cbf_count_datablocks
2.3.36 cbf_count_columns
2.3.37 cbf_count_rows
2.3.39 cbf_select_category, cbf_select_saveframe, cbf_select_blockitem

---

2.3.36 cbf_count_columns

PROTOTYPE

#include "cbf.h"

int cbf_count_columns (cbf_handle *handle*, unsigned int **columns*);

DESCRIPTION

cbf_count_columns puts the number of columns in the current category in **columns*.

ARGUMENTS

handle CBF handle.
columns Pointer to the destination column count.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.34 cbf_count_datablocks
2.3.35 cbf_count_categories, cbf_count_saveframes, cbf_count_blockitems
2.3.37 cbf_count_rows
2.3.40 cbf_select_column

2.3.37 cbf_count_rows

PROTOTYPE

#include "cbf.h"

int cbf_count_rows (cbf_handle *handle*, unsigned int **rows*);

DESCRIPTION

cbf_count_rows puts the number of rows in the current category in **rows* .

ARGUMENTS

handle CBF handle.
rows Pointer to the destination row count.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.34 cbf_count_datablocks
2.3.35 cbf_count_categories, cbf_count_saveframes, cbf_count_blockitems
2.3.36 cbf_count_columns
2.3.41 cbf_select_row

2.3.38 cbf_select_datablock

PROTOTYPE

#include "cbf.h"

int cbf_select_datablock (cbf_handle *handle*, unsigned int *datablock*);

DESCRIPTION

cbf_select_datablock selects data block number *datablock* as the current data block.

The first data block is number 0.

If the data block does not exist, the function returns CBF_NOTFOUND.

ARGUMENTS

handle CBF handle.
datablock Number of the data block to select.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.34 cbf_count_datablocks
2.3.39 cbf_select_category, cbf_select_saveframe, cbf_select_blockitem
2.3.40 cbf_select_column
2.3.41 cbf_select_row

2.3.39 cbf_select_category

PROTOTYPE

#include "cbf.h"

int cbf_select_category (cbf_handle *handle*, unsigned int *category*);

DESCRIPTION

cbf_select_category selects category number *category* in the current data block as the current category.

The first category is number 0.

The current column and row become undefined.

If the category does not exist, the function returns CBF_NOTFOUND.

ARGUMENTS

handle CBF handle.

category Number of the category to select.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.35 cbf_count_categories, cbf_count_saveframes, cbf_count_blockitems
2.3.38 cbf_select_datablock
2.3.40 cbf_select_column
2.3.41 cbf_select_row

2.3.40 cbf_select_column

PROTOTYPE

#include "cbf.h"

int cbf_select_column (cbf_handle *handle*, unsigned int *column*);

DESCRIPTION

cbf_select_column selects column number *column* in the current category as the current column.

The first column is number 0.

The current row is not affected

If the column does not exist, the function returns CBF_NOTFOUND.

ARGUMENTS

handle CBF handle.
column Number of the column to select.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.36 cbf_count_columns
2.3.38 cbf_select_datablock
2.3.39 cbf_select_category, cbf_select_saveframe, cbf_select_blockitem
2.3.41 cbf_select_row

2.3.41 cbf_select_row

PROTOTYPE

#include "cbf.h"

int cbf_select_row (cbf_handle *handle*, unsigned int *row*);

DESCRIPTION

cbf_select_row selects row number *row* in the current category as the current row.

The first row is number 0.

The current column is not affected

If the row does not exist, the function returns CBF_NOTFOUND.

ARGUMENTS

handle	CBF handle.
row	Number of the row to select.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.37 cbf_count_rows
2.3.38 cbf_select_datablock
2.3.39 cbf_select_category, cbf_select_saveframe, cbf_select_blockitem
2.3.40 cbf_select_column

2.3.42 cbf_datablock_name

PROTOTYPE

#include "cbf.h"

int cbf_datablock_name (cbf_handle *handle*, const char *******datablockname*);

DESCRIPTION

cbf_datablock_name sets **datablockname* to point to the name of the current data block.

The data block name will be valid as long as the data block exists and has not been renamed.

The name must not be modified by the program in any way.

ARGUMENTS

handle	CBF handle.
datablockname	Pointer to the destination data block name pointer.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.29 cbf_find_datablock

2.3.43 cbf_category_name

PROTOTYPE

#include "cbf.h"

int cbf_category_name (cbf_handle *handle*, const char *******categoryname*);

DESCRIPTION

cbf_category_name sets **categoryname* to point to the name of the current category of the current data block.

The category name will be valid as long as the category exists.

The name must not be modified by the program in any way.

ARGUMENTS

handle	CBF handle.
categoryname	Pointer to the destination category name pointer.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem

---

2.3.44 cbf_column_name

PROTOTYPE

#include "cbf.h"

int cbf_column_name (cbf_handle *handle*, const char *******columnname*);

DESCRIPTION

cbf_column_name sets **columnname* to point to the name of the current column of the current category.

The column name will be valid as long as the column exists.

The name must not be modified by the program in any way.

ARGUMENTS

handle CBF handle.
columnname Pointer to the destination column name pointer.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.31 cbf_find_column

2.3.45 cbf_row_number

PROTOTYPE

#include "cbf.h"

int cbf_row_number (cbf_handle *handle*, unsigned int **row*);

DESCRIPTION

cbf_row_number sets **row* to the number of the current row of the current category.

ARGUMENTS

handle CBF handle.
row Pointer to the destination row number.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.41 cbf_select_row

2.3.46 cbf_get_value, cbf_require_value

PROTOTYPE

#include "cbf.h"

int cbf_get_value (cbf_handle *handle*, const char ***value*);
int cbf_require_value (cbf_handle *handle*, const char ***value*, const char **defaultvalue*);

DESCRIPTION

cbf_get_value sets **value* to point to the ASCII value of the item at the current column and row.
cbf_set_value sets **value* to point to the ASCII value of the item at the current column and row, creating the data item if necessary and initializing it to a copy of *defaultvalue*.

If the value is not ASCII, the function returns CBF_BINARY.

The value will be valid as long as the item exists and has not been set to a new value.

The value must not be modified by the program in any way.

ARGUMENTS

handle CBF handle.
value Pointer to the destination value pointer.
value Default value character string.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.50 cbf_get_integervalue, cbf_require_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.47 cbf_set_value

PROTOTYPE

#include "cbf.h"

int cbf_set_value (cbf_handle *handle*, const char **value*);

DESCRIPTION

cbf_set_value sets the item at the current column and row to the ASCII value *value*.

ARGUMENTS

handle CBF handle.
value ASCII value.
defaultvalue default ASCII value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.53 cbf_set_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.48 cbf_get_typeofvalue

PROTOTYPE

#include "cbf.h"

int cbf_get_typeofvalue (cbf_handle *handle*, const char ***typeofvalue*);

DESCRIPTION

cbf_get_value sets **typeofvalue* to point an ASCII descriptor of the value of the item at the current column and row. The strings that may be returned are "null" for a null value indicated by a "." or a "?", "bnry" for a binary value, "word" for an unquoted string, "dblq" for a double-quoted string, "sglq" for a single-quoted string, and "text" for a semicolon-quoted text field. A field for which no value has been set sets **typeofvalue* to NULL rather than to the string "null".

The *typeofvalue* must not be modified by the program in any way.

ARGUMENTS

handle	CBF handle.
typeofvalue	Pointer to the destination type-of-value string pointer.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.49 cbf_set_typeofvalue
2.3.50 cbf_get_integervalue, cbf_require_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.49 cbf_set_typeofvalue

PROTOTYPE

#include "cbf.h"

int cbf_set_typeofvalue (cbf_handle *handle*, const char **typeofvalue*);

DESCRIPTION

cbf_set_typeofvalue sets the type of the item at the current column and row to the type specified by the ASCII character string given by *typeofvalue*. The strings that may be used are "null" for a null value indicated by a "." or a "?", "word" for an unquoted string, "dblq" for a double-quoted string, "sglq" for a single-quoted string, and "text" for a semicolon-quoted text field. Not all types may be used for all values. No changes may be made to the type of binary values. You may not set the type of a string that contains a single quote followed by a blank or a tab or which contains multiple lines to "sglq". You may not set the type of a string that contains a double quote followed by a blank or a tab or which contains multiple lines to "dblq".

ARGUMENTS

handle	CBF handle.
typeofvalue	ASCII string for desired type of value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.51 cbf_set_integervalue
2.3.53 cbf_set_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.50 cbf_get_integervalue, cbf_require_integervalue

PROTOTYPE

#include "cbf.h"

int cbf_get_integervalue (cbf_handle *handle*, int **number*);
int cbf_require_integervalue (cbf_handle *handle*, int **number*, int *defaultvalue*);

DESCRIPTION

cbf_get_integervalue sets **number* to the value of the ASCII item at the current column and row interpreted as a decimal integer. cbf_require_integervalue sets **number* to the value of the ASCII item at the current column and row interpreted as a decimal integer, setting it to *defaultvalue* if necessary.

If the value is not ASCII, the function returns CBF_BINARY.

ARGUMENTS

handle	CBF handle.
number	pointer to the number.
defaultvalue	default number value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue
2.3.51 cbf_set_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.51 cbf_set_integervalue

PROTOTYPE

#include "cbf.h"

int cbf_set_integervalue (cbf_handle *handle*, int *number*);

DESCRIPTION

cbf_set_integervalue sets the item at the current column and row to the integer value *number* written as a decimal ASCII string.

ARGUMENTS

handle	CBF handle.
number	Integer value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue

2.3.50 cbf_get_integervalue, cbf_require_integervalue
2.3.51 cbf_set_integervalue
2.3.53 cbf_set_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.52 cbf_get_doublevalue, cbf_require_doublevalue

PROTOTYPE

#include "cbf.h"

int cbf_get_doublevalue (cbf_handle *handle*, double **number*);
int cbf_require_doublevalue (cbf_handle *handle*, double **number*, double *defaultvalue*);

DESCRIPTION

cbf_get_doublevalue sets **number* to the value of the ASCII item at the current column and row interpreted as a decimal floating-point number. cbf_require_doublevalue sets **number* to the value of the ASCII item at the current column and row interpreted as a decimal floating-point number, setting it to *defaultvalue* if necessary.

If the value is not ASCII, the function returns CBF_BINARY.

ARGUMENTS

handle	CBF handle.
number	Pointer to the destination number.
defaultvalue	default number value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.50 cbf_get_integervalue, cbf_require_integervalue
2.3.53 cbf_set_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.53 cbf_set_doublevalue

PROTOTYPE

#include "cbf.h"

int cbf_set_doublevalue (cbf_handle *handle*, const char **format*, double *number*);

DESCRIPTION

cbf_set_doublevalue sets the item at the current column and row to the floating-point value *number* written as an ASCII string with the format specified by *format* as appropriate for the printf function.

ARGUMENTS

handle	CBF handle.
format	Format for the number.
number	Floating-point value.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims

PROTOTYPE

#include "cbf.h"

int cbf_get_integerarrayparameters (cbf_handle *handle*, unsigned int **compression*, int **binary_id*, size_t **elsize*, int **elsigned*, int **elunsigned*, size_t **elements*, int **minelement*, int **maxelement*);
int cbf_get_integerarrayparameters_wdims (cbf_handle *handle*, unsigned int **compression*, int **binary_id*, size_t **elsize*, int **elsigned*, int **elunsigned*, size_t **elements*, int **minelement*, int **maxelement*, const char ***byteorder*, size_t **dim1*, size_t **dim2*, size_t **dim3*, size_t **padding*);
int cbf_get_realarrayparameters (cbf_handle *handle*, unsigned int **compression*, int **binary_id*, size_t **elsize*, size_t **elements*);
int cbf_get_realarrayparameters_wdims (cbf_handle *handle*, unsigned int **compression*, int **binary_id*, size_t **elsize*, size_t **elements*, const char ***byteorder*, size_t **dim1*, size_t **dim2*, size_t **dim3*, size_t **padding*);

DESCRIPTION

cbf_get_integerarrayparameters sets **compression*, **binary_id*, **elsize*, **elsigned*, **elunsigned*, **elements*, **minelement* and **maxelement* to values read from the binary value of the item at the current column and row. This provides all the arguments needed for a subsequent call to cbf_set_integerarray, if a copy of the array is to be made into another CIF or CBF. cbf_get_realarrayparameters sets **compression*, **binary_id*, **elsize*, **elements* to values read from the binary value of the item at the current column and row. This provides all the arguments needed for a subsequent call to cbf_set_realarray, if a copy of the arry is to be made into another CIF or CBF.

The variants int cbf_get_integerarrayparameters_wdims and cbf_get_realarrayparameters_wdims set ***byteorder*, **dim1*, **dim2*, **dim3*, and **padding* as well, providing the additional parameters needed for a subsequent call to cbf_set_integerarray_wdims or cbf_set_realarray_wdims.

The value returned in **byteorder* is a pointer either to the string "little_endian" or to the string "big_endian". This should be the byte order of the data, not necessarily of the host machine. No attempt should be made to modify this string. At this time only "little_endian" will be returned.

The values returned in **dim1*, **dim2* and **dim3* are the sizes of the fastest changing, second fastest changing and third fastest changing dimensions of the array, if specified, or zero, if not specified.

The value returned in **padding* is the size of the post-data padding, if any and if specified in the data header. The value is given as a count of octets.

If the value is not binary, the function returns CBF_ASCII.

ARGUMENTS

handle	CBF handle.
compression	Compression method used.
elsize	Size in bytes of each array element.
binary_id	Pointer to the destination integer binary identifier.
elsigned	Pointer to an integer. Set to 1 if the elements can be read as signed integers.
elunsigned	Pointer to an integer. Set to 1 if the elements can be read as unsigned integers.
elements	Pointer to the destination number of elements.
minelement	Pointer to the destination smallest element.
maxelement	Pointer to the destination largest element.
byteorder	Pointer to the destination byte order.
dim1	Pointer to the destination fastest dimension.
dim2	Pointer to the destination second fastest dimension.
dim3	Pointer to the destination third fastest dimension.
padding	Pointer to the destination padding size.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.50 cbf_get_integervalue, cbf_require_integervalue

2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.55 cbf_get_integerarray, cbf_get_realarray

PROTOTYPE

#include "cbf.h"

int cbf_get_integerarray (cbf_handle *handle*, int **binary_id*, void **array*, size_t *elsize*, int *elsigned*, size_t *elements*, size_t **elements_read*);
int cbf_get_realarray (cbf_handle *handle*, int **binary_id*, void **array*, size_t *elsize*, size_t *elements*, size_t **elements_read*);

DESCRIPTION

cbf_get_integerarray reads the binary value of the item at the current column and row into an integer array. The array consists of *elements* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsigned* is non-0 and unsigned otherwise. **binary_id* is set to the binary section identifier and **elements_read* to the number of elements actually read. cbf_get_realarray reads the binary value of the item at the current column and row into a real array. The array consists of *elements* elements of *elsize* bytes each, starting at *array*. **binary_id* is set to the binary section identifier and **elements_read* to the number of elements actually read.

If any element in the integer binary data cant fit into the destination element, the destination is set the nearest possible value.

If the value is not binary, the function returns CBF_ASCII.

If the requested number of elements cant be read, the function will read as many as it can and then return CBF_ENDOFDATA.

Currently, the destination array must consist of chars, shorts or ints (signed or unsigned). If *elsize* is not equal to sizeof (char), sizeof (short) or sizeof (int), for cbf_get_integerarray, or sizeof(double) or sizeof(float), for cbf_get_realarray the function returns CBF_ARGUMENT.

An additional restriction in the current version of CBFlib is that values too large to fit in an int are not correctly decompressed. As an example, if the machine with 32-bit ints is reading an array containing a value outside the range 0 .. $2^{\wedge 32}$-1 (unsigned) or -$2^{\wedge 31}$ .. $2^{\wedge 31}$-1 (signed), the array will not be correctly decompressed. This restriction will be removed in a future release. For cbf_get_realarray, only IEEE format is supported. No conversion to other floating point formats is done at this time.

ARGUMENTS

handle	CBF handle.
binary_id	Pointer to the destination integer binary identifier.
array	Pointer to the destination array.
elsize	Size in bytes of each destination array element.

elsigned	Set to non-0 if the destination array elements are signed.
elements	The number of elements to read.
elements_read	Pointer to the destination number of elements actually read.

RETURN VALUE

Returns an error code on failure or 0 for success.
SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.50 cbf_get_integervalue, cbf_require_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims

PROTOTYPE

#include "cbf.h"

int cbf_set_integerarray (cbf_handle *handle*, unsigned int *compression*, int *binary_id*, void **array*, size_t *elsize*, int *elsigned*, size_t *elements*);
int cbf_set_integerarray_wdims (cbf_handle *handle*, unsigned int *compression*, int *binary_id*, void **array*, size_t *elsize*, int *elsigned*, size_t *elements*, const char **byteorder*, size_t *dim1*, size_t *dim2*, size_t *dim3*, size_t *padding*);
int cbf_set_realarray (cbf_handle *handle*, unsigned int *compression*, int *binary_id*, void **array*, size_t *elsize*, size_t *elements*);
int cbf_set_realarray_wdims (cbf_handle *handle*, unsigned int *compression*, int *binary_id*, void **array*, size_t *elsize*, size_t *elements*, const char **byteorder*, size_t *dim1*, size_t *dim2*, size_t *dim3*, size_t *padding*);

DESCRIPTION

cbf_set_integerarray sets the binary value of the item at the current column and row to an integer *array*. The array consists of *elements* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsigned* is non-0 and unsigned otherwise. *binary_id* is the binary section identifier. cbf_set_realarray sets the binary value of the item at the current column and row to an integer *array*. The array consists of *elements* elements of *elsize* bytes each, starting at *array*. *binary_id* is the binary section identifier.

The cbf_set_integerarray_wdims and cbf_set_realarray_wdims allow the data header values of *byteorder*, *dim1*, *dim2*, *dim3* and *padding* to be set to the data byte order, the fastest, second fastest and third fastest array dimensions and the size in byte of the post data padding to be used.

The array will be compressed using the compression scheme specifed by *compression*. Currently, the

available schemes are:

CBF_CANONICAL	Canonical-code compression (section 3.3.1)
CBF_PACKED	CCP4-style packing (section 3.3.2)
CBF_PACKED_V2	CCP4-style packing, version 2 (section 3.3.2)
CBF_BYTE_OFFSET	Simple "byte_offset" compression.
CBF_NONE	No compression. NOTE: This scheme is by far the slowest of the four and uses much more disk space. It is intended for routine use with small arrays only. With large arrays (like images) it should be used only for debugging.

The values compressed are limited to 64 bits. If any element in the array is larger than 64 bits, the value compressed is the nearest 64-bit value.

Currently, the source array must consist of chars, shorts or ints (signed or unsigned), for cbf_set_integerarray, or IEEE doubles or floats for cbf_set_realarray. If *elsize* is not equal to sizeof (char), sizeof (short) or sizeof (int), the function returns CBF_ARGUMENT.

ARGUMENTS

handle	CBF handle.
compression	Compression method to use.
binary_id	Integer binary identifier.
array	Pointer to the source array.
elsize	Size in bytes of each source array element.
elsigned	Set to non-0 if the source array elements are signed.
	elements: The number of elements in the array.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.53 cbf_set_doublevalue
2.3.54 cbf_get_integerarrayparameters, cbf_get_integerarrayparameters_wdims, cbf_get_realarrayparameters, cbf_get_realarrayparameters_wdims
2.3.55 cbf_get_integerarray, cbf_get_realarray
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.57 cbf_failnez

DEFINITION

#include "cbf.h"

#define cbf_failnez(f) {int err; err = (f); if (err) return err; }

DESCRIPTION

cbf_failnez is a macro used for error propagation throughout CBFlib. cbf_failnez executes the function *f* and saves the returned error value. If the error value is non-0, cbf_failnez executes a return with the error value as argument. If CBFDEBUG is defined, then a report of the error is also printed to the standard error stream, stderr, in the form

CBFlib error *f* in "*symbol*"

where *f* is the decimal value of the error and *symbol* is the symbolic form.

ARGUMENTS

f Integer error value.

SEE ALSO

2.3.58 cbf_onfailnez

2.3.58 cbf_onfailnez

DEFINITION

#include "cbf.h"

#define cbf_onfailnez(f,c) {int err; err = (f); if (err) {{c; }return err; }}

DESCRIPTION

cbf_onfailnez is a macro used for error propagation throughout CBFlib. cbf_onfailnez executes the function *f* and saves the returned error value. If the error value is non-0, cbf_failnez executes first the statement *c* and then a return with the error value as argument. If CBFDEBUG is defined, then a report of the error is also printed to the standard error stream, stderr, in the form

CBFlib error *f* in "*symbol*"

where *f* is the decimal value of the error and *symbol* is the symbolic form.

ARGUMENTS

f integer function to execute.
c statement to execute on failure.

SEE ALSO

- 2.3.57 cbf_failnez

2.3.59 cbf_require_datablock

PROTOTYPE

#include "cbf.h"

int cbf_require_datablock (cbf_handle *handle*, const char **datablockname*);

DESCRIPTION

cbf_require_datablock makes the data block with name *datablockname* the current data block, if it exists, or creates it if it does not.

The comparison is case-insensitive.

The current category becomes undefined.

ARGUMENTS

handle	CBF handle.
datablockname	The name of the data block to find or create.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.21 cbf_rewind_datablock
2.3.25 cbf_next_datablock
2.3.29 cbf_find_datablock
2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.31 cbf_find_column
2.3.32 cbf_find_row
2.3.42 cbf_datablock_name
2.3.60 cbf_require_category
2.3.61 cbf_require_column

2.3.60 cbf_require_category

PROTOTYPE

#include "cbf.h"

int cbf_require_category (cbf_handle *handle*, const char **categoryname*);

DESCRIPTION

cbf_rewuire_category makes the category in the current data block with name *categoryname* the current category, if it exists, or creates the catagory if it does not exist.

The comparison is case-insensitive.

The current column and row become undefined.

ARGUMENTS

handle	CBF handle.

categoryname	The name of the category to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.22 cbf_rewind_category, cbf_rewind_saveframe, cbf_rewind_blockitem
2.3.26 cbf_next_category, cbf_next_saveframe, cbf_next_blockitem
2.3.29 cbf_find_datablock
2.3.31 cbf_find_column
2.3.32 cbf_find_row
2.3.43 cbf_category_name
2.3.59 cbf_require_datablock
2.3.61 cbf_require_column

2.3.61 cbf_require_column

PROTOTYPE

#include "cbf.h"

int cbf_require_column (cbf_handle *handle*, const char **columnname*);

DESCRIPTION

cbf_require_column makes the columns in the current category with name *columnname* the current column, if it exists, or creates it if it does not.

The comparison is case-insensitive.

The current row is not affected.

ARGUMENTS

handle	CBF handle.
columnname	The name of column to find.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.19 cbf_rewind_column
2.3.27 cbf_next_column
2.3.29 cbf_find_datablock
2.3.30 cbf_find_category, cbf_find_saveframe, cbf_find_blockitem
2.3.32 cbf_find_row
2.3.44 cbf_column_name
2.3.59 cbf_require_datablock
2.3.60 cbf_require_category

2.3.62 cbf_require_column_value

PROTOTYPE

#include "cbf.h"

int cbf_require_column_value (cbf_handle *handle*, const char **columnname*, const char ***value*, const char **defaultvalue*);

DESCRIPTION

cbf_require_column_doublevalue sets **value* to the ASCII item at the current row for the column given with the name given by **columnname*, or to the string given by *defaultvalue* if the item cannot be found.

ARGUMENTS

handle	CBF handle.
columnname	Name of the column containing the number.
value	pointer to the location to receive the value.
defaultvalue	Value to use if the requested column and value cannot be found.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.63 cbf_require_column_integervalue
2.3.64 cbf_require_column_doublevalue

2.3.63 cbf_require_column_integervalue

PROTOTYPE

#include "cbf.h"

int cbf_require_column_integervalue (cbf_handle *handle*, const char **columnname*, int **number*, const int *defaultvalue*);

DESCRIPTION

cbf_require_column_doublevalue sets **number* to the value of the ASCII item at the current row for the column given with the name given by **columnname*, with the value interpreted as an integer number, or

to the number given by *defaultvalue* if the item cannot be found.

ARGUMENTS

handle	CBF handle.
columnname	Name of the column containing the number.
number	pointer to the location to receive the integer value.
defaultvalue	Value to use if the requested column and value cannot be found.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.64 cbf_require_column_doublevalue

2.3.64 cbf_require_column_doublevalue

PROTOTYPE

#include "cbf.h"

int cbf_require_column_doublevalue (cbf_handle *handle*, const char **columnname*, double **number*, const double *defaultvalue*);

DESCRIPTION

cbf_require_column_doublevalue sets **number* to the value of the ASCII item at the current row for the column given with the name given by **columnname*, with the value interpreted as a decimal floating-point number, or to the number given by *defaultvalue* if the item cannot be found.

ARGUMENTS

handle	CBF handle.
columnname	Name of the column containing the number.
number	pointer to the location to receive the floating-point value.
defaultvalue	Value to use if the requested column and value cannot be found.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 cbf_get_value, cbf_require_value
2.3.47 cbf_set_value
2.3.48 cbf_get_typeofvalue
2.3.49 cbf_set_typeofvalue
2.3.51 cbf_set_integervalue
2.3.52 cbf_get_doublevalue, cbf_require_doublevalue
2.3.56 cbf_set_integerarray, cbf_set_integerarray_wdims, cbf_set_realarray, cbf_set_realarray_wdims
2.3.62 cbf_require_column_value
2.3.63 cbf_require_column_integervalue

2.3.65 cbf_get_local_integer_byte_order, cbf_get_local_real_byte_order, cbf_get_local_real_format

PROTOTYPE

#include "cbf.h"

int cbf_get_local_integer_byte_order (char ** *byte_order*);
int cbf_get_local_real_byte_order (char ** *byte_order*);
int cbf_get_local_real_format (char ** *real_format*);

DESCRIPTION

cbf_get_local_integer_byte_order returns the byte order of integers on the machine on which the API is being run in the form of a character string returned as the value pointed to by *byte_order*.
cbf_get_local_real_byte_order returns the byte order of reals on the machine on which the API is being run in the form of a character string returned as the value pointed to by *byte_order*.
cbf_get_local_real_format returns the format of floats on the machine on which the API is being run in the form of a character string returned as the value pointed to by *real_format*. The strings returned must not be modified in any way.

The values returned in *byte_order* may be the strings "little_endian" or "big-endian". The values returned in *real_format* may be the strings "ieee 754-1985" or "other". Additional values may be returned by future versions of the API.

ARGUMENTS

byte_order	pointer to the returned string
real_format	pointer to the returned string

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.66 cbf_get_dictionary, cbf_set_dictionary, cbf_require_dictionary

PROTOTYPE

#include "cbf.h"

int cbf_get_dictionary (cbf_handle *handle*, cbf_handle * *dictionary*);

int cbf_set_dictionary (cbf_handle *handle*, cbf_handle *dictionary_in*);
int cbf_require_dictionary (cbf_handle *handle*, cbf_handle * *dictionary*)

DESCRIPTION

cbf_get_dictionary sets **dictionary* to the handle of a CBF which has been associated with the CBF *handle* by cbf_set_dictionary. cbf_set_dictionary associates the CBF handle *dictionary_in* with *handle* as its dictionary. cbf_require_dictionary sets **dictionary* to the handle of a CBF which has been associated with the CBF *handle* by cbf_set_dictionary or creates a new empty CBF and associates it with *handle*, returning the new handle in **dictionary*.

ARGUMENTS

handle	CBF handle.
dictionary	Pointer to CBF handle of dictionary.
dictionary_in	CBF handle of dcitionary.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.67 cbf_convert_dictionary

PROTOTYPE

#include "cbf.h"

int cbf_convert_dictionary (cbf_handle *handle*, cbf_handle *dictionary*)

DESCRIPTION

cbf_convert_dictionary converts *dictionary* as a DDL1 or DDL2 dictionary to a CBF dictionary of category and item properties for *handle*, creating a new dictionary if none exists or layering the definitions in *dictionary* onto the existing dictionary of *handle* if one exists.

If a CBF is read into *handle* after calling cbf_convert_dictionary, then the dictionary will be used for validation of the CBF as it is read.

ARGUMENTS

handle	CBF handle.
dictionary	CBF handle of dictionary.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.68 cbf_find_tag, cbf_find_local_tag

PROTOTYPE

#include "cbf.h"

int cbf_find_tag (cbf_handle *handle*, const char **tag*)
int cbf_find_local_tag (cbf_handle *handle*, const char **tag*)

DESCRIPTION

cbf_find_tag searches all of the CBF *handle* for the CIF tag given by the string *tag* and makes it the current tag. The search does not include the dictionary, but does include save frames as well as categories.

The string *tag* is the complete tag in either DDL1 or DDL2 format, starting with the leading underscore, not just a category or column.

ARGUMENTS

handle	CBF handle.
tag	CIF tag.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.69 cbf_find_category_root, cbf_set_category_root, cbf_require_category_root

PROTOTYPE

#include "cbf.h"

int cbf_find_category_root (cbf_handle *handle*, const char* *categoryname*, const char** *categoryroot*);
int cbf_set_category_root (cbf_handle *handle*, const char* *categoryname_in*, const char**categoryroot*);
int cbf_require_category_root (cbf_handle handle, const char* *categoryname*, const char** *categoryroot*);

DESCRIPTION

cbf_find_category_root sets **categoryroot* to the root category of which *categoryname* is an alias. cbf_set_category_root sets *categoryname_in* as an alias of *categoryroot* in the dictionary associated with *handle*, creating the dictionary if necessary. cbf_require_category_root sets **categoryroot* to the root category of which *categoryname* is an alias, if there is one, or to the value of *categoryname*, if *categoryname* is not an alias.

A returned *categoryroot* string must not be modified in any way.

ARGUMENTS

handle	CBF handle.
categoryname	category name which may be an alias.
categoryroot	pointer to a returned category root name.
categoryroot_in	input category root name.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.70 cbf_find_tag_root, cbf_set_tag_root, cbf_require_tag_root

PROTOTYPE

#include "cbf.h"

int cbf_find_tag_root (cbf_handle *handle*, const char* *tagname*, const char** *tagroot*);
int cbf_set_tag_root (cbf_handle *handle*, const char* *tagname*, const char**tagroot_in*);
int cbf_require_tag_root (cbf_handle handle, const char* *tagname*, const char** *tagroot*);

DESCRIPTION

cbf_find_tag_root sets **tagroot* to the root tag of which *tagname* is an alias. cbf_set_tag_root sets *tagname* as an alias of *tagroot_in* in the dictionary associated with *handle*, creating the dictionary if necessary. cbf_require_tag_root sets **tagroot* to the root tag of which *tagname* is an alias, if there is one, or to the value of *tagname*, if *tagname* is not an alias.

A returned *tagroot* string must not be modified in any way.

ARGUMENTS

handle	CBF handle.
tagname	tag name which may be an alias.
tagroot	pointer to a returned tag root name.
tagroot_in	input tag root name.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.3.71 cbf_find_tag_category, cbf_set_tag_category

PROTOTYPE

#include "cbf.h"

int cbf_find_tag_category (cbf_handle *handle*, const char* *tagname*, const char** *categoryname*);
int cbf_set_tag_category (cbf_handle *handle*, const char* tagname, const char* *categoryname_in*);

DESCRIPTION

cbf_find_tag_category sets *categoryname* to the category associated with *tagname* in the dictionary associated with *handle*. cbf_set_tag_category updates the dictionary associated with *handle* to indicated that *tagname* is in category *categoryname_in*.

ARGUMENTS

handle	CBF handle.
tagname	tag name.
categoryname	pointer to a returned category name.
categoryname_in	input category name.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4 High-level function prototypes

2.4.1 cbf_read_template

PROTOTYPE

#include "cbf_simple.h"

int cbf_read_template (cbf_handle *handle*, FILE **file*);

DESCRIPTION

cbf_read_template reads the CBF or CIF file *file* into the CBF object specified by *handle* and selects the first datablock as the current datablock.

ARGUMENTS

handle Pointer to a CBF handle.
file Pointer to a file descriptor.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.2 cbf_get_diffrn_id, cbf_require_diffrn_id

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_diffrn_id (cbf_handle *handle*, const char ***diffrn_id*);
int cbf_require_diffrn_id (cbf_handle *handle*, const char ***diffrn_id*, const char **default_id*)

DESCRIPTION

cbf_get_diffrn_id sets **diffrn_id* to point to the ASCII value of the "diffrn.id" entry.
cbf_require_diffrn_id also sets **diffrn_id* to point to the ASCII value of the "diffrn.id" entry, but, if the "diffrn.id" entry does not exist, it sets the value in the CBF and in**diffrn_id* to the character string given by *default_id*, creating the category and column is necessary.

The *diffrn_id* will be valid as long as the item exists and has not been set to a new value.

The *diffrn_id* must not be modified by the program in any way.

ARGUMENTS

handle CBF handle.
diffrn_id Pointer to the destination value pointer.

default_id Character string default value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.3 cbf_set_diffrn_id

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_diffrn_id (cbf_handle *handle*, const char **diffrn_id*);

DESCRIPTION

cbf_set_diffrn_id sets the "diffrn.id" entry of the current datablock to the ASCII value *diffrn_id*.

This function also changes corresponding "diffrn_id" entries in the "diffrn_source", "diffrn_radiation", "diffrn_detector" and "diffrn_measurement" categories.

ARGUMENTS

handle CBF handle.
diffrn_id ASCII value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.4 cbf_get_crystal_id

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_crystal_id (cbf_handle *handle*, const char ***crystal_id*);

DESCRIPTION

cbf_get_crystal_id sets **crystal_id* to point to the ASCII value of the "diffrn.crystal_id" entry.

If the value is not ASCII, the function returns CBF_BINARY.

The value will be valid as long as the item exists and has not been set to a new value.

The value must not be modified by the program in any way.

ARGUMENTS

handle CBF handle.

crystal_id Pointer to the destination value pointer.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.5 cbf_set_crystal_id

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_crystal_id (cbf_handle *handle*, const char **crystal_id*);

DESCRIPTION

cbf_set_crystal_id sets the "diffrn.crystal_id" entry to the ASCII value *crystal_id*.

ARGUMENTS

handle CBF handle.
crystal_id ASCII value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.6 cbf_get_wavelength

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_wavelength (cbf_handle *handle*, double **wavelength*);

DESCRIPTION

cbf_get_wavelength sets **wavelength* to the current wavelength in Å.

ARGUMENTS

handle CBF handle.
wavelength Pointer to the destination.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.7 cbf_set_wavelength

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_wavelength (cbf_handle *handle*, double *wavelength*);

DESCRIPTION

cbf_set_wavelength sets the current wavelength in Å to *wavelength*.

ARGUMENTS

handle CBF handle.
wavelength Wavelength in Å.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.8 cbf_get_polarization

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_polarization (cbf_handle *handle*, double **polarizn_source_ratio*, double **polarizn_source_norm*);

DESCRIPTION

cbf_get_polarization sets **polarizn_source_ratio* and **polarizn_source_norm* to the corresponding source polarization parameters.

Either destination pointer may be NULL.

ARGUMENTS

handle CBF handle.
polarizn_source_ratio Pointer to the destination polarizn_source_ratio.
polarizn_source_norm Pointer to the destination polarizn_source_norm.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.9 cbf_set_polarization

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_polarization (cbf_handle *handle*, double *polarizn_source_ratio*, double *polarizn_source_norm*);

DESCRIPTION

cbf_set_polarization sets the source polarization to the values specified by *polarizn_source_ratio* and *polarizn_source_norm*.

ARGUMENTS

handle	CBF handle.
polarizn_source_ratio	New value of polarizn_source_ratio.
polarizn_source_norm	New value of polarizn_source_norm.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.10 cbf_get_divergence

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_divergence (cbf_handle *handle*, double **div_x_source*, double **div_y_source*, double **div_x_y_source*);

DESCRIPTION

cbf_get_divergence sets **div_x_source*, **div_y_source* and **div_x_y_source* to the corresponding source divergence parameters.

Any of the destination pointers may be NULL.

ARGUMENTS

handle	CBF handle.
div_x_source	Pointer to the destination div_x_source.
div_y_source	Pointer to the destination div_y_source.
div_x_y_source	Pointer to the destination div_x_y_source.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.11 cbf_ set_divergence

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_divergence (cbf_handle *handle*, double *div_x_source*, double *div_y_source*, double *div_x_y_source*);

DESCRIPTION

cbf_set_divergence sets the source divergence parameters to the values specified by *div_x_source*, *div_y_source* and *div_x_y_source*.

ARGUMENTS

handle	CBF handle.
div_x_source	New value of div_x_source.
div_y_source	New value of div_y_source.
div_x_y_source	New value of div_x_y_source.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.12 cbf_count_elements

PROTOTYPE

#include "cbf_simple.h"

int cbf_count_elements (cbf_handle *handle*, unsigned int **elements*);

DESCRIPTION

cbf_count_elements sets **elements* to the number of detector elements.

ARGUMENTS

handle	CBF handle.
elements	Pointer to the destination count.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.13 cbf_get_element_id

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_element_id (cbf_handle *handle*, unsigned int *element_number*, const char ***element_id*);

DESCRIPTION

cbf_get_element_id sets **element_id* to point to the ASCII value of the *element_number*Õth

"diffrn_data_frame.detector_element_id" entry, counting from 0.

If the detector element does not exist, the function returns CBF_NOTFOUND.

The *element_id* will be valid as long as the item exists and has not been set to a new value.

The *element_id* must not be modified by the program in any way.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
element_id	Pointer to the destination.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.14 cbf_get_gain

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_gain (cbf_handle *handle*, unsigned int *element_number*, double **gain*, double **gain_esd*);

DESCRIPTION

cbf_get_gain sets **gain* and **gain_esd* to the corresponding gain parameters for element number *element_number*.

Either of the destination pointers may be NULL.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
gain	Pointer to the destination gain.
gain_esd	Pointer to the destination gain_esd.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.15 cbf_ set_gain

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_gain (cbf_handle *handle*, unsigned int *element_number*, double *gain*, double *gain_esd*);

DESCRIPTION

cbf_set_gain sets the gain of element number *element_number* to the values specified by *gain* and *gain_esd*.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
gain	New gain value.
gain_esd	New gain_esd value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.16 cbf_get_overload

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_overload (cbf_handle *handle*, unsigned int *element_number*, double **overload*);

DESCRIPTION

cbf_get_overload sets **overload* to the overload value for element number *element_number*.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
overload	Pointer to the destination overload.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.17 cbf_ set_overload

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_overload (cbf_handle *handle*, unsigned int *element_number*, double *overload*);

DESCRIPTION

cbf_set_overload sets the overload value of element number *element_number* to *overload*.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
overload	New overload value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.18 cbf_get_integration_time

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_integration_time (cbf_handle *handle*, unsigned int *reserved*, double **time*);

DESCRIPTION

cbf_get_integration_time sets **time* to the integration time in seconds. The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
time	Pointer to the destination time.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.19 cbf_set_integration_time

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_integration_time (cbf_handle *handle*, unsigned int *reserved*, double *time*);

DESCRIPTION

cbf_set_integration_time sets the integration time in seconds to the value specified by *time*. The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.

reserved	Unused. Any value other than 0 is invalid.
time Integration	time in seconds.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.20 cbf_get_timestamp

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_timestamp (cbf_handle *handle*, unsigned int *reserved*, double **time*, int **timezone*);

DESCRIPTION

cbf_get_timestamp sets **time* to the collection timestamp in seconds since January 1 1970. **timezone* is set to timezone difference from UTC in minutes. The parameter *reserved* is presently unused and should be set to 0.

Either of the destination pointers may be NULL.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
time	Pointer to the destination collection timestamp.
timezone	Pointer to the destination timezone difference.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.21 cbf_set_timestamp

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_timestamp (cbf_handle *handle*, unsigned int *reserved*, double *time*, int *timezone*, double *precision*);

DESCRIPTION

cbf_set_timestamp sets the collection timestamp in seconds since January 1 1970 to the value specified by *time*. The timezone difference from UTC in minutes is set to *timezone*. If no timezone is desired, *timezone* should be CBF_NOTIM EZONE. The parameter *reserved* is presently unused and should be set to 0.

The precision of the new timestamp is specified by the value *precision* in seconds. If *precision* is 0, the saved timestamp is assumed accurate to 1 second.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
time	Timestamp in seconds since January 1 1970.
timezone	Timezone difference from UTC in minutes or CBF_NOTIMEZONE.
precision	Timestamp precision in seconds.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.22 cbf_get_datestamp

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_datestamp (cbf_handle *handle*, unsigned int *reserved*, int **year*, int **month*, int **day*, int **hour*, int **minute*, double **second*, int **timezone*);

DESCRIPTION

cbf_get_datestamp sets **year*, **month*, **day*, **hour*, **minute* and **second* to the corresponding values of the collection timestamp. **timezone* is set to timezone difference from UTC in minutes. The parameter < i>reserved is presently unused and should be set to 0.

Any of the destination pointers may be NULL.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
year	Pointer to the destination timestamp year.
month	Pointer to the destination timestamp month (1-12).
day	Pointer to the destination timestamp day (1-31).
hour	Pointer to the destination timestamp hour (0-23).
minute	Pointer to the destination timestamp minute (0-59).
second	Pointer to the destination timestamp second (0-60.0).
timezone	Pointer to the destination timezone difference from UTC in minutes.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.23 cbf_set_datestamp

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_datestamp (cbf_handle *handle*, unsigned int *reserved*, int *year*, int *month*, int *day*, int *hour*, int *minute*, double *second*, int *timezone*, double *precision*);

DESCRIPTION

cbf_set_datestamp sets the collection timestamp in seconds since January 1 1970 to the value specified by *time*. The timezone difference from UTC in minutes is set to *timezone*. If no timezone is desired, *timezone* should be CBF_NOTIM EZONE. The parameter *reserved* is presently unused and should be set to 0.

The precision of the new timestamp is specified by the value *precision* in seconds. If *precision* is 0, the saved timestamp is assumed accurate to 1 second.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
time	Timestamp in seconds since January 1 1970.
timezone	Timezone difference from UTC in minutes or CBF_NOTIMEZONE.
precision	Timestamp precision in seconds.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.24 cbf_set_current_timestamp

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_current_timestamp (cbf_handle *handle*, unsigned int *reserved*, int *timezone*)

DESCRIPTION

cbf_set_current_timestamp sets the collection timestamp to the current time. The timezone difference from UTC in minutes is set to *timezone*. If no timezone is desired, *timezone* should be CBF_NOTIMEZONE. If no timezone is used, the timest amp will be UTC. The parameter *reserved* is presently unused and should be set to 0.

The new timestamp will have a precision of 1 second.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
timezone	Timezone difference from UTC in minutes or CBF_NOTIMEZONE.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.25 cbf_get_image_size, cbf_get_3d_image_size

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_image_size (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, size_t **ndim1*, size_t **ndim2*);
int cbf_get_3d_image_size (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, size_t **ndim1*, size_t **ndim2*, size_t **ndim3*);

DESCRIPTION

cbf_get_image_size sets **ndim1* and **ndim2* to the slow and fast dimensions of the image array for element number *element_number*. If the array is 1-dimensional, **ndim1* will be set to the array size and **ndim2* will be set to 1. If the array is 3-dimensional an error code will be returned.
cbf_get_3d_image_size sets **ndim1*, **ndim2* and **ndim3* to the slowest, next fastest and fastest dimensions, respectively, of the 3D image array for element number *element_number*. If the array is 1-dimensional, **ndim1* will be set to the array size and **ndim2* and **ndim3* will be set to 1. If the array is 2-dimensional **ndim1* and **ndim2* will be set as for a call to cbf_get_image_size and **ndim3* will be set to 1.

Note that the ordering of dimensions is specified by values of the tag _array_structure_list.precedence with a precedence of 1 for the fastest dimension, 2 for the next slower, etc., which is opposite to the ordering of the dimension arguments for these functions.

Any of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
ndim1	Pointer to the destination slowest dimension.
ndim2	Pointer to the destination next faster dimension.
ndim3	Pointer to the destination fastest dimension.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.26 cbf_get_image, cbf_get_real_image, cbf_get_3d_image, cbf_get_real_3d_image

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, void **array*, size_t *elsize*, int *elsign*, size_t *ndim1*, size_t *ndim2*);
int cbf_get_real_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, void **array*, size_t *elsize*, size_t *ndim1*, size_t *ndim2*);
int cbf_get_3d_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, void **array*, size_t *elsize*, int *elsign*, size_t *ndim1*, size_t *ndim2*, size_t *ndim3*);
int cbf_get_real_3d_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, void **array*, size_t *elsize*, size_t *ndim1*, size_t *ndim2*, size_t *ndim3*);

DESCRIPTION

cbf_get_image reads the image array for element number *element_number* into an *array*. The array consists of *ndim1*× *ndim2* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsign* is non-0 and unsigned otherwise. cbf_get_real_image reads the image array of IEEE doubles or floats for element number *element_number* into an *array*. A real array is always signed. cbf_get_image reads the 3D image array for element number *element_number* into an *array*. The array consists of *ndim1*× *ndim2*× *ndim3* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsign* is non-0 and unsigned otherwise. cbf_get_real_3d_image reads the 3D image array of IEEE doubles or floats for element number *element_number* into an *array*. A real array is always signed.

The structure of the array as a 1-, 2- or 3-dimensional array should agree with the structure of the array given in the ARRAY_STRUCTURE_LIST category. If the array is 1-dimensional, *ndim1* should be the array size and *ndim2* and, for the 3D calls, *ndim3*, should be set to 1 both in the call and in the imgCIF data being processed. If the array is 2-dimensional and a 3D call is used, *ndim1* and *ndim2* should be the array dimensions and *ndim3* should be set to 1 both in the call and in the imgCIF data being processed.

If any element in the binary data canÕt fit into the destination element, the destination is set the nearest possible value.

If the value is not binary, the function returns CBF_ASCII.

If the requested number of elements canÕt be read, the function will read as many as it can and then return CBF_ENDOFDATA.

Currently, the destination *array* must consist of chars, shorts or ints (signed or unsigned) for cbf_get_image, or IEEE doubles or floats for cbf_get_real_image. If *elsize* is not equal to sizeof (char), sizeof (short), sizeof (int), sizeof(double) or sizeof(float), the function returns CBF_ARGUMENT.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
array	Pointer to the destination array.
elsize	Size in bytes of each destination array element.

elsigned	Set to non-0 if the destination array elements are signed.
ndim1	Slowest array dimension.
ndim2	Next faster array dimension.
ndim3	Fastest array dimension.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.27 cbf_set_image, cbf_set_real_image, cbf_set_3d_image, cbf_set_real_3d_image

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, unsigned int compression, void **array*, size_t *elsize*, int *elsign*, size_t *ndim1*, size_t *ndim2*);
int cbf_set_real_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, unsigned int compression, void **array*,size_t *elsize*, size_t *ndim1*, size_t *ndim2*);
int cbf_set_3d_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, unsigned int compression, void **array*, size_t *elsize*, int *elsign*, size_t *ndim1*, size_t *ndim2*, size_t *ndim2=3*);
int cbf_set_real_3d_image (cbf_handle *handle*, unsigned int *reserved*, unsigned int *element_number*, unsigned int compression, void **array*,size_t *elsize*, size_t *ndim1*, size_t *ndim2*, size_t *ndim3*);

DESCRIPTION

cbf_set_image writes the image array for element number *element_number*. The *array* consists of *ndim1*× *ndim2* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsign* is non-0 and unsigned otherwise. cbf_set_real_image writes the image array for element number *element_number*. The *array* consists of *ndim1*× *ndim2* IEEE double or float elements of *elsize* bytes each, starting at *array*. cbf_set_3d_image writes the 3D image array for element number *element_number*. The *array* consists of *ndim1*× *ndim2*× *ndim3* elements of *elsize* bytes each, starting at *array*. The elements are signed if *elsign* is non-0 and unsigned otherwise. cbf_set_real_3d_image writes the 3D image array for element number *element_number*. The *array* consists of *ndim1*× *ndim2*× *ndim3* IEEE double or float elements of *elsize* bytes each, starting at *array*.

If the array is 1-dimensional, *ndim1* should be the array size and *ndim2* and, for the 3D calls, *ndim3*, should be set to 1. If the array is 2-dimensional and the 3D calls are used, *ndim1* and *ndim2* should be used for the array dimensions and *ndim3* should be set to 1.

The array will be compressed using the compression scheme specifed by compression. Currently, the available schemes are:

CBF_CANONICAL	Canonical-code compression (section 3.3.1)
CBF_PACKED	CCP4-style packing (section 3.3.2)
CBF_PACKED_V2	CCP4-style packing, version 2 (section 3.3.2)
CBF_BYTE_OFFSET	Simple "byte_offset" compression.
CBF_NONE	No compression.

The values compressed are limited to 64 bits. If any element in the array is larger than 64 bits, the value compressed is the nearest 64-bit value.

Currently, the source *array* must consist of chars, shorts or ints (signed or unsigned)for cbf_set_image, or IEEE doubles or floats for cbf_set_real_image. If *elsize* is not equal to sizeof (short), sizeof (int), sizeof(double) or sizeof(float), the function returns CBF_ARGUMENT.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
compression	Compression type.
array	Pointer to the image array.
elsize	Size in bytes of each image array element.
elsigned	Set to non-0 if the image array elements are signed.
ndim1	Slow array dimension.
ndim2	Fast array dimension.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.28 cbf_get_axis_setting

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_axis_setting (cbf_handle *handle*, unsigned int *reserved*, const char **axis_id*, double **start*, double **increment*);

DESCRIPTION

cbf_get_axis_setting sets **start* and **increment* to the corresponding values of the axis *axis_id*.

Either of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
axis_id	Axis id.
start	Pointer to the destination start value.
increment	Pointer to the destination increment value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.29 cbf_set_axis_setting

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_axis_setting (cbf_handle *handle*, unsigned int *reserved*, const char **axis_id*, double *start*, double *increment*);

DESCRIPTION

cbf_set_axis_setting sets the starting and increment values of the axis *axis_id* to *start* and *increment*.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

handle	CBF handle.
reserved	Unused. Any value other than 0 is invalid.
axis_id	Axis id.
start	Start value.
increment	Increment value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.30 cbf_construct_goniometer

PROTOTYPE

#include "cbf_simple.h"

int cbf_construct_goniometer (cbf_handle *handle*, cbf_goniometer **goniometer*);

DESCRIPTION

cbf_construct_goniometer constructs a goniometer object using the description in the CBF object handle and initialises the goniometer handle **goniometer*.

ARGUMENTS

handle	CBF handle.
goniometer	Pointer to the destination goniometer handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.31 cbf_free_goniometer

PROTOTYPE

#include "cbf_simple.h"

int cbf_free_goniometer (cbf_goniometer *goniometer*);

DESCRIPTION

cbf_free_goniometer destroys the goniometer object specified by *goniometer* and frees all associated memory.

ARGUMENTS

goniometer	Goniometer handle to free.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.32 cbf_get_rotation_axis

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_rotation_axis (cbf_goniometer *goniometer*, unsigned int *reserved*, double **vector1*, double **vector2*, double *vector3*);

DESCRIPTION

cbf_get_rotation_axis sets **vector1*, **vector2*, and **vector3* to the 3 components of the goniometer rotation axis used for the exposure.

Any of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

goniometer	Goniometer handle.
reserved	Unused. Any value other than 0 is invalid.
vector1	Pointer to the destination x component of the rotation axis.
vector2	Pointer to the destination y component of the rotation axis.
vector3	Pointer to the destination z component of the rotation axis.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.33 cbf_get_rotation_range

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_rotation_range (cbf_goniometer *goniometer*, unsigned int *reserved*, double **start*, double **increment*);

DESCRIPTION

cbf_get_rotation_range sets **start* and **increment* to the corresponding values of the goniometer rotation axis used for the exposure.

Either of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

goniometer	Goniometer handle.
reserved	Unused. Any value other than 0 is invalid.
start	Pointer to the destination start value.
increment	Pointer to the destination increment value.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.34 cbf_rotate_vector

PROTOTYPE

#include "cbf_simple.h"

int cbf_rotate_vector (cbf_goniometer *goniometer*, unsigned int *reserved*, double *ratio*, double *initial1*, double *initial2*, double *initial3*, double **final1*, double **final2*, double **final3*);

DESCRIPTION

cbf_rotate_vector sets **final1*, **final2*, and **final3* to the 3 components of the of the vector (*initial1*, *initial2*, *initial3*) after reorientation by applying the goniometer rotations. The value *ratio* specif ies the goniometer setting and varies from 0.0 at the beginning of the exposure to 1.0 at the end, irrespective of the actual rotation range.

Any of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

goniometer	Goniometer handle.
reserved	Unused. Any value other than 0 is invalid.
ratio	Goniometer setting. 0 = beginning of exposure, 1 = end.
initial1	x component of the initial vector.
initial2	y component of the initial vector.
initial3	z component of the initial vector.
vector1	Pointer to the destination x component of the final vector.
vector2	Pointer to the destination y component of the final vector.
vector3	Pointer to the destination z component of the final vector.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.35 cbf_get_reciprocal

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_reciprocal (cbf_goniometer *goniometer*, unsigned int *reserved*, double *ratio*, double *wavelength*, double *real1*, double *real2*, double *real3*, double **reciprocal1*, double **reciprocal2*, double **reciprocal3*);

DESCRIPTION

cbf_get_reciprocal sets **reciprocal1*, * *reciprocal2*, and * *reciprocal3* to the 3 components of the of the reciprocal-space vector corresponding to the real-space vector (*real1*, *real2*, *real3*). The reciprocal-space vector is oriented to correspond to the goniometer setting with all axes at 0. The value *wavelength* is the wavlength in Å and the value *ratio* specifies the current goniometer setting and varies from 0.0 at the beginning of the exposur e to 1.0 at the end, irrespective of the actual rotation range.

Any of the destination pointers may be NULL.

The parameter *reserved* is presently unused and should be set to 0.

ARGUMENTS

goniometer	Goniometer handle.
reserved	Unused. Any value other than 0 is invalid.
ratio	Goniometer setting. 0 = beginning of exposure, 1 = end.
wavelength	Wavelength in Å.
real1	x component of the real-space vector.
real2	y component of the real-space vector.
real3	z component of the real-space vector.
reciprocal1	Pointer to the destination x component of the reciprocal-space vector.
reciprocal2	Pointer to the destination y component of the reciprocal-space vector.

reciprocal3 Pointer to the destination z component of the reciprocal-space vector.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.36 cbf_construct_detector

PROTOTYPE

#include "cbf_simple.h"

int cbf_construct_detector (cbf_handle *handle*, cbf_detector **detector*, unsigned int *element_number*);

DESCRIPTION

cbf_construct_detector constructs a detector object for detector element number *element_number* using the description in the CBF object handle and initialises the detector handle **detector*.

ARGUMENTS

handle CBF handle.

detector Pointer to the destination detector handle.

element_number The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.37 cbf_free_detector

PROTOTYPE

#include "cbf_simple.h"

int cbf_free_detector (cbf_detector *detector*);

DESCRIPTION

cbf_free_detector destroys the detector object specified by *detector* and frees all associated memory.

ARGUMENTS

detector Detector handle to free.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.38 cbf_get_beam_center, cbf_set_beam_center

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_beam_center (cbf_detector *detector*, double **index1*, double **index2*, double **center1*, double **center2*);

int cbf_set_beam_center (cbf_detector *detector*, double **index1*, double **index2*, double **center1*, double **center2*);

DESCRIPTION

cbf_get_beam_center sets **center1* and **center2* to the displacements in mm along the detector axes from pixel (0, 0) to the point at which the beam intersects the detector and **index1* and **index2* to the corresponding indices. cbf_set_beam_center sets the offsets in the axis category for the detector element axis with precedence 1 to place the beam center at the position given in mm by **center1* and **center2* as the displacements in mm along the detector axes from pixel (0, 0) to the point at which the beam intersects the detector at the indices given **index1* and **index2*.

Any of the destination pointers may be NULL for getting the beam center. For setting the beam axis, either the indices of the center must not be NULL.

The indices are non-negative for beam centers within the detector surface, but the center for an axis with a negative increment will be negative for a beam center within the detector surface.

ARGUMENTS

detector Detector handle.

index1 Pointer to the destination slow index.

index2 Pointer to the destination fast index.

center1 Pointer to the destination displacement along the slow axis.

center2 Pointer to the destination displacement along the fast axis.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.39 cbf_get_detector_distance

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_detector_distance (cbf_detector *detector*, double **distance*);

DESCRIPTION

cbf_get_detector_distance sets **distance* to the nearest distance from the sample position to the detector plane.

ARGUMENTS

detector Detector handle.
distance Pointer to the destination distance.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.40 cbf_get_detector_normal

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_detector_normal (cbf_detector *detector*, double **normal1*, double **normal2*, double **normal3*);

DESCRIPTION

cbf_get_detector_normal sets **normal1*, **normal2*, and **normal3* to the 3 components of the of the normal vector to the detector plane. The vector is normalized.

Any of the destination pointers may be NULL.

ARGUMENTS

detector Detector handle.
normal1 Pointer to the destination x component of the normal vector.
normal2 Pointer to the destination y component of the normal vector.
normal3 Pointer to the destination z component of the normal vector.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.41 cbf_get_pixel_coordinates

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_pixel_coordinates (cbf_detector *detector*, double *index1*, double *index2*, double **coordinate1*, double **coordinate2*, double **coordinate3*);

DESCRIPTION

cbf_get_pixel_coordinates sets **coordinate1*, **coordinate2*, and **coordinate3* to the vector position of pixel (*index1*, *index2*) on the detector surface. If *index1* and *index2* are integers then the coordinates correspond to the center of a pixel.

Any of the destination pointers may be NULL.

ARGUMENTS

detector Detector handle.
index1 Slow index.
index2 Fast index.
coordinate1 Pointer to the destination x component.
coordinate2 Pointer to the destination y component.
coordinate3 Pointer to the destination z component.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.42 cbf_get_pixel_normal

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_pixel_normal (cbf_detector *detector*, double *index1*, double *index2*, double **normal1*, double **normal2*, double **normal3*);

DESCRIPTION

cbf_get_detector_normal sets **normal1*, **normal2*, and **normal3* to the 3 components of the of the normal vector to the pixel at (*index1*, *index2*). The vector is normalized.

Any of the destination pointers may be NULL.

ARGUMENTS

detector Detector handle.
index1 Slow index.
index2 Fast index.
normal1 Pointer to the destination x component of the normal vector.
normal2 Pointer to the destination y component of the normal vector.
normal3 Pointer to the destination z component of the normal vector.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.43 cbf_get_pixel_area

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_pixel_area (cbf_detector *detector*, double *index1*, double *index2*, double *area*, double *projected_area*);

DESCRIPTION

cbf_get_pixel_area sets **area* to the area of the pixel at (*index1*, *index2*) on the detector surface and **projected_area* to the apparent area of the pixel as viewed from the sample position.

Either of the destination pointers may be NULL.

ARGUMENTS

detector	Detector handle.
index1	Slow index.
index2	Fast index.
area	Pointer to the destination area in mm2.
projected_area	Pointer to the destination apparent area in mm2.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.44 cbf_get_pixel_size

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_pixel_size (cbf_handle *handle*, unsigned int *element_number*, unsigned int *axis_number*, double **psize*);

DESCRIPTION

cbf_get_pixel_size sets **psize* to point to the double value in millimeters of the axis *axis_number* of the detector element *element_number*. The *axis_number* is numbered from 1, starting with the fastest axis.

If the pixel size is not given explcitly in the "array_element_size" category, the function returns CBF_NOTFOUND.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
axis_number	The number of the axis, fastest first, starting from 1.
psize	Pointer to the destination pixel size.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.45 cbf_set_pixel_size

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_pixel_size (cbf_handle *handle*, unsigned int *element_number*, unsigned int *axis_number*, double *psize*);

DESCRIPTION

cbf_set_pixel_size sets the item in the &quote;size&quote; column of the "array_structure_list" category at the row which matches axis *axis_number* of the detector element *element_number* converting the double pixel size *psize* from meters to millimeters in storing it in the "size" column for the axis *axis_number* of the detector element *element_number*. The *axis_number* is numbered from 1, starting with the fastest axis.

If the "array_structure_list" category does not already exist, it is created.

If the appropriate row in the "array_structure_list" catgeory does not already exist, it is created.

If the pixel size is not given explcitly in the "array_element_size category", the function returns CBF_NOTFOUND.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
axis_number	The number of the axis, fastest first, starting from 1.
psize	The pixel size in millimeters.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.46 cbf_get_inferred_pixel_size

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_inferred_pixel_size (cbf_detector *detector*, unsigned int *axis_number*, double **psize*);

DESCRIPTION

cbf_get_inferred_pixel_size sets **psize* to point to the double value in millimeters of the pixel size for the axis *axis_number* value for pixel at (*index1*, *index2*) on the detector surface. The slow index is treated as axis 1 and the fast index is treated as axis 2.

ARGUMENTS

detector	Detector handle.

axis_number The number of the axis.
area Pointer to the destination pizel size in mm.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.47 cbf_get_unit_cell

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_unit_cell (cbf_handle *handle*, double *cell*[6], double *cell_esd*[6]);

DESCRIPTION

cbf_get_unit_cell sets *cell*[0:2] to the double values of the cell edge lengths a, b and c in Ångstroms, *cell*[3:5] to the double values of the cell angles α, β and γ in degrees, *cell_esd*[0:2] to the double values of the estimated strandard deviations of the cell edge lengths a, b and c in Ångstroms, *cell_esd*[3:5] to the double values of the estimated standard deviations of the the cell angles α, β and γ in degrees.

The values returned are retrieved from the first row of the "cell" category. The value of "_cell.entry_id" is ignored.

cell or *cell_esd* may be NULL.

If *cell* is NULL, the cell parameters are not retrieved.

If *cell_esd* is NULL, the cell parameter esds are not retrieved.

If the "cell" category is present, but some of the values are missing, zeros are returned for the missing values.

ARGUMENTS

handle CBF handle.
cell Pointer to the destination array of 6 doubles for the cell parameters.
cell_esd Pointer to the destination array of 6 doubles for the cell parameter esds.

RETURN VALUE

Returns an error code on failure or 0 for success. No errors is returned for missing values if the "cell" category exists.

SEE ALSO

2.4.48 cbf_set_unit_cell
2.4.49 cbf_get_reciprocal_cell
2.4.50 cbf_set_reciprocal_cell

2.4.51 cbf_compute_cell_volume
2.4.52 cbf_compute_reciprocal_cell

2.4.48 cbf_set_unit_cell

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_unit_cell (cbf_handle *handle*, double *cell*[6], double *cell_esd*[6]);

DESCRIPTION

cbf_set_unit_cell sets the cell parameters to the double values given in *cell*[0:2] for the cell edge lengths a, b and c in Ångstroms, the double values given in *cell*[3:5] for the cell angles α, β and γ in degrees, the double values given in *cell_esd*[0:2] for the estimated standard deviations of the cell edge lengths a, b and c in Ångstroms, and the double values given in *cell_esd*[3:5] for the estimated standard deviations of the the cell angles α, β and γ in degrees.

The values are placed in the first row of the "cell" category. If no value has been given for "_cell.entry_id", it is set to the value of the "diffrn.id" entry of the current data block.

cell or *cell_esd* may be NULL.

If *cell* is NULL, the cell parameters are not set.

If *cell_esd* is NULL, the cell parameter esds are not set.

If the "cell" category is not present, it is created. If any of the necessary columns are not present, they are created.

ARGUMENTS

handle CBF handle.
cell Pointer to the array of 6 doubles for the cell parameters.
cell_esd Pointer to the array of 6 doubles for the cell parameter esds.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.4.47 cbf_get_unit_cell
2.4.49 cbf_get_reciprocal_cell
2.4.50 cbf_set_reciprocal_cell
2.4.51 cbf_compute_cell_volume
2.4.52 cbf_compute_reciprocal_cell

SEE ALSO

2.4.49 cbf_get_reciprocal_cell

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_reciprocal_cell (cbf_handle *handle*, double *cell*[6], double *cell_esd*[6]);

DESCRIPTION

cbf_get_reciprocal_cell sets *cell*[0:2] to the double values of the reciprocal cell edge lengths a^*, b^* and c^* in Ångstroms^{-1}, *cell*[3:5] to the double values of the reciprocal cell angles α^*, β^* and γ^* in degrees, *cell_esd*[0:2] to the double values of the estimated strandard deviations of the reciprocal cell edge lengths a^*, b^* and c^* in Ångstroms^{-1}, *cell_esd*[3:5] to the double values of the estimated standard deviations of the the reciprocal cell angles α^*, β^* and γ^* in degrees.

The values returned are retrieved from the first row of the "cell" category. The value of "_cell.entry_id" is ignored.

cell or *cell_esd* may be NULL.

If *cell* is NULL, the reciprocal cell parameters are not retrieved.

If *cell_esd* is NULL, the reciprocal cell parameter esds are not retrieved.

If the "cell" category is present, but some of the values are missing, zeros are returned for the missing values.

ARGUMENTS

handle	CBF handle.
cell	Pointer to the destination array of 6 doubles for the reciprocal cell parameters.
cell_esd	Pointer to the destination array of 6 doubles for the reciprocal cell parameter esds.

RETURN VALUE

Returns an error code on failure or 0 for success. No errors is returned for missing values if the "cell" category exists.

SEE ALSO

2.4.47 cbf_get_unit_cell
2.4.48 cbf_set_unit_cell
2.4.50 cbf_set_reciprocal_cell
2.4.51 cbf_compute_cell_volume
2.4.52 cbf_compute_reciprocal_cell

2.4.50 cbf_set_reciprocal_cell

PROTOTYPE

#include "cbf_simple.h"

int cbf_set_reciprocal_cell (cbf_handle *handle*, double *cell*[6], double *cell_esd*[6]);

DESCRIPTION

cbf_set_reciprocal_cell sets the reciprocal cell parameters to the double values given in *cell*[0:2] for the reciprocal cell edge lengths a^*, b^* and c^* in Ångstroms^{-1}, the double values given in *cell*[3:5] for the reciprocal cell angles α^*, β^* and γ^* in degrees, the double values given in *cell_esd*[0:2] for the estimated strandard deviations of the reciprocal cell edge lengths a^*, b^* and c^* in Ångstroms, and the double values given in *cell_esd*[3:5] for the estimated standard deviations of the reciprocal cell angles α^*, β^* and γ^* in degrees.

The values are placed in the first row of the "cell" category. If no value has been given for "_cell.entry_id", it is set to the value of the "diffrn.id" entry of the current data block.

cell or *cell_esd* may be NULL.

If *cell* is NULL, the reciprocal cell parameters are not set.

If *cell_esd* is NULL, the reciprocal cell parameter esds are not set.

If the "cell" category is not present, it is created. If any of the necessary columns are not present, they are created.

ARGUMENTS

handle	CBF handle.
cell	Pointer to the array of 6 doubles for the reciprocal cell parameters.
cell_esd	Pointer to the array of 6 doubles for the reciprocal cell parameter esds.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.4.47 cbf_get_unit_cell
2.4.48 cbf_set_unit_cell
2.4.50 cbf_get_reciprocal_cell
2.4.51 cbf_compute_cell_volume
2.4.52 cbf_compute_reciprocal_cell

2.4.51 cbf_compute_cell_volume

PROTOTYPE

#include "cbf_simple.h"

int cbf_compute_cell_volume (double *cell*[6], double **volume*);

DESCRIPTION

cbf_compute_cell_volume sets **volume* to point to the volume of the unit cell computed from the double values in *cell*[0:2] for the cell edge lengths a, b and c in Ångstroms and the double values given in *cell*[3:5] for the cell angles α, β and γ in degrees.

ARGUMENTS

cell	Pointer to the array of 6 doubles giving the cell parameters.
volume	Pointer to the doubles for cell volume.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.4.46 cbf_get_unit_cell
2.4.47 cbf_set_unit_cell
2.4.50 cbf_get_reciprocal_cell
2.4.50 cbf_set_reciprocal_cell
2.4.52 cbf_compute_reciprocal_cell

2.4.52 cbf_compute_reciprocal_cell

PROTOTYPE

#include "cbf_simple.h"

int cbf_compute_reciprocal_cell (double *cell*[6], double *rcell*[6]);

DESCRIPTION

cbf_compute_reciprocal_cell sets *rcell* to point to the array of reciprocal cell parameters computed from the double values *cell*[0:2] giving the cell edge lengths a, b and c in Ångstroms, and the double values *cell*[3:5] giving the cell angles α, β and γ in degrees. The double values *rcell*[0:2] will be set to the reciprocal cell lengths a^*, b^* and c^* in Ångstroms^{-1} and the double values *rcell*[3:5] will be set to the reciprocal cell angles α^*, β^* and γ^* in degrees.

ARGUMENTS

cell	Pointer to the array of 6 doubles giving the cell parameters.
rcell	Pointer to the destination array of 6 doubles giving the reciprocal cell parameters.
volume	Pointer to the doubles for cell volume.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.4.46 cbf_get_unit_cell
2.4.47 cbf_set_unit_cell
2.4.50 cbf_get_reciprocal_cell
2.4.50 cbf_set_reciprocal_cell
2.4.51 cbf_compute_cell_volume

2.4.53 cbf_get_orientation_matrix, cbf_set_orientation_matrix

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_orientation_matrix (cbf_handle *handle*, double *ub_matrix*[9]);
int cbf_set_orientation_matrix (cbf_handle *handle*, double *ub_matrix*[9]);

cbf_get_orientation_matrix sets *ub_matrix* to point to the array of orientation matrix entries in the "diffrn" category in the order of columns:

"UB[1][1]" "UB[1][2]" "UB[1][3]"
"UB[2][1]" "UB[2][2]" "UB[2][3]"
"UB[3][1]" "UB[3][2]" "UB[3][3]"

cbf_set_orientation_matrix sets the values in the "diffrn" category to the values pointed to by *ub_matrix*.

ARGUMENTS

handle	CBF handle.
ubmatric	Source or destination array of 9 doubles giving the orientation matrix parameters.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.4.54 cbf_get_bin_sizes, cbf_set_bin_sizes

PROTOTYPE

#include "cbf_simple.h"

int cbf_get_bin_sizes(cbf_handle *handle*, unsigned int *element_number*, double * *slowbinsize*, double * *fastbinsize*);
int cbf_set_bin_sizes(cbf_handle *handle*, unsigned int element_number, double *slowbinsize_in*,double *fastbinsize_in*);

cbf_get_bin_sizes sets *slowbinsize* to point to the value of the number of pixels composing one array element in the dimension that changes at the second-fastest rate and *fastbinsize* to point to the value of the number of pixels composing one array element in the dimension that changes at the fastest rate for the dectector element with the ordinal *element_number*. cbf_set_bin_sizes sets the the pixel bin sizes in the "array_intensities" category to the values of *slowbinsize_in* for the number of pixels composing one array element in the dimension that changes at the second-fastest rate and *fastbinsize_in* for the number of pixels composing one array element in the dimension that changes at the fastest rate for the dectector

element with the ordinal *element_number*.

In order to allow for software binning involving fractions of pixels, the bin sizes are doubles rather than ints.

ARGUMENTS

handle	CBF handle.
element_number	The number of the detector element counting from 0 by order of appearance in the "diffrn_data_frame" category.
slowbinsize	Pointer to the returned number of pixels composing one array element in the dimension that changes at the second-fastest rate.
fastbinsize	Pointer to the returned number of pixels composing one array element in the dimension that changes at the fastest rate.
slowbinsize_in	The number of pixels composing one array element in the dimension that changes at the second-fastest rate.
fastbinsize_in	The number of pixels composing one array element in the dimension that changes at the fastest rate.

RETURN VALUE

Returns an error code on failure or 0 for success.

2.5 F90 function interfaces

At the suggestion of W. Kabsch, Fortran 90/95 routines have been added to CBFlib. As of this writing code has been written to allow the reading of CBF_BYTE_OFFSET, CBF_PACKED and CBF_PACKED_V2 binary images. This code has been gather into FCBlib (Fortran Crystallographic Binary library) as lib/libfcb.

In general, most of the FCBlib functions return 0 for normal completion and a non-zero value in case of an error. In a few cases, such as FCB_ATOL_WCNT and FCB_NBLEN_ARRAY in order to conform to the conventions for commonly used C-equivalent functions, the function return is the value being computed.

For each function, an interface is given to be included in the declarations of your Fortran 90/95 code. Some functions in FCBIIB are not intended for external use and are subject to change: FCB_UPDATE_JPA_POINTERS_I2, FCB_UPDATE_JPA_POINTERS_I4, FCB_UPDATE_JPA_POINTERS_3D_I2, FCB_UPDATE_JPA_POINTERS_3D_I4 and CNT2PIX. These names should not be used for user routines.

The functions involving reading of a CBF have been done strictly in Fortran without the use of C code. This has required some compromises and the use of direct access I/O. Rather than putting the buffer and its control variables into COMMON these are passed as local arguments to make the routines inherently 'threadsafe' in a parallel programming environment. Note also, that a reading error could occur for the last record if it does not fill a full block. The code is written to recover from end-of-record and end-of-file errors, if possible. On many modern system, no special action is required, but on some systems it may be necessary to make use of the padding between the end of binary data and the terminal MIME boundary marker in binary sections. To ensure maximum portability of CBF files, a padding of 4095 bytes is recommended. Existing files without padding can be converted to files with padding by use of the new -p4 option for cif2cbf.

2.5.1 FCB_ATOL_WCNT

```
  INTERFACE
INTEGER(8) FUNCTION FCB_ATOL_WCNT(ARRAY, N, CNT)
INTEGER(1),INTENT(IN):: ARRAY(N)
INTEGER,   INTENT(IN):: N
INTEGER,  INTENT(OUT):: CNT
END FUNCTION
END INTERFACE
```

FCB_ATOL_WCNT converts INTEGER(1) bytes in *ARRAY* of *N* bytes to an INTEGER(8) value returned as the function value. The number of bytes of *ARRAY* actually used before encountering a character not used to form the number is returned in *CNT*.

The scan stops at the first byte in *ARRAY* that cannot be properly parsed as part of the integer result.

ARGUMENTS

ARRAY	The array of INTEGER(1) bytes to be scanned
N	The INTEGER size of *ARRAY*
CNT	The INTEGER size of the portion of *ARRAY* scanned.

RETURN VALUE

Returns the INTEGER(8) value derived from the characters *ARRAY*(1:*CNT*) scanned.

2.5.2 FCB_CI_STRNCMPARR

```
  INTERFACE
INTEGER FUNCTION FCB_CI_STRNCMPARR(STRING>, ARRAY, N, LIMIT)
CHARACTER(LEN=*),INTENT(IN):: STRING>
INTEGER,         INTENT(IN):: N, LIMIT
INTEGER(1),      INTENT(IN):: ARRAY(N)
END FUNCTION
END INTERFACE
```

The function FCB_CI_STRNCMPARR compares up to *LIMIT* characters of character string *STRING* and INTEGER(1) byte array *ARRAY* of dimension *N* in a case-insensitive manner, returning 0 for a match.

ARGUMENTS

STRING	A character string
ARRAY	The array of INTEGER(1) bytes to be scanned
N	The INTEGER size of *ARRAY*
N	The INTEGER limit on the number of characters to consider in the comparison

RETURN VALUE

Returns 0 if the string and array match, a non-zero value otherwise.

2.5.3 FCB_EXIT_BINARY

```
  INTERFACE
INTEGER FUNCTION FCB_EXIT_BINARY(TAPIN,LAST_CHAR,FCB_BYTES_IN_REC,&
                                 BYTE_IN_FILE,REC_IN_FILE,BUFFER,  &
                                 PADDING )
INTEGER,    INTENT(IN)    :: TAPIN,FCB_BYTES_IN_REC
INTEGER,    INTENT(INOUT):: BYTE_IN_FILE,REC_IN_FILE
INTEGER(1),INTENT(INOUT):: LAST_CHAR,BUFFER(FCB_BYTES_IN_REC)
INTEGER(8),INTENT(IN)    :: PADDING
END FUNCTION
END INTERFACE
```

The function FCB_EXIT_BINARY is used to skip from the end of a binary section past any padding to the end of the text section that encloses the binary section. The values of the arguments must be consistent with those in the last call to FCB_NEXT_BINARY

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
LAST_CHAR	The last character (as an INTEGER(1) byte) read.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*
PADDING	The INTEGER(8) number of bytes of padding after the binary data and before the closing MIME boundary.

RETURN VALUE

Returns 0 if the function is successful. Returns whatever non-zero error value is reported by FCB_READ_LINE if a necessary next line cannot be read.

SEE ALSO

2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.4 FCB_NBLEN_ARRAY

```
  INTERFACE
INTEGER FUNCTION FCB_NBLEN_ARRAY(ARRAY, ARRAYLEN)
INTEGER,    INTENT(IN):: ARRAYLEN
INTEGER(1), INTENT(IN):: ARRAY(ARRAYLEN)
END FUNCTION
END INTERFACE
```

The function FCB_NBLEN_ARRAY returns the trimmed length of the INTEGER(1) byte array *ARRAY* of dimension *ARRAYLEN* after removal of trailing ASCII blanks, horizontal tabs (Z'09'), newlines (Z'0A') and carriage returns (Z'0D'). The resulting length may be zero.

The INTEGER trimmed length is returned as the function value.

ARGUMENTS

ARRAY	The array of bytes for which the trimmed length is required.
ARRAYLEN	The dimension of the array of bytes to be scanned.

RETURN VALUE

Returns the trimmed length of the array *ARRAY*.

2.5.5 FCB_NEXT_BINARY

```
  INTERFACE
INTEGER FUNCTION FCB_NEXT_BINARY(TAPIN,LAST_CHAR,FCB_BYTES_IN_REC,&
                                 BYTE_IN_FILE,REC_IN_FILE,BUFFER,  &
                                 ENCODING,SIZE,ID,DIGEST,          &
                                 COMPRESSION,BITS,VORZEICHEN,REELL,&
                                 BYTEORDER,DIMOVER,DIM1,DIM2,DIM3, &
                                 PADDING )
INTEGER,    INTENT(IN)    :: TAPIN,FCB_BYTES_IN_REC
INTEGER,    INTENT(INOUT):: BYTE_IN_FILE,REC_IN_FILE
INTEGER(1),INTENT(INOUT):: LAST_CHAR,BUFFER(FCB_BYTES_IN_REC)
INTEGER,    INTENT(OUT)  :: ENCODING
INTEGER, INTENT(OUT)        :: SIZE    !Binary size
INTEGER, INTENT(OUT)        :: ID      !Binary ID
CHARACTER(len=*),INTENT(OUT):: DIGEST  !Message digest
INTEGER,         INTENT(OUT):: COMPRESSION
INTEGER,         INTENT(OUT):: BITS,VORZEICHEN,REELL
CHARACTER(len=*),INTENT(OUT):: BYTEORDER
INTEGER(8),      INTENT(OUT):: DIMOVER
INTEGER(8),      INTENT(OUT):: DIM1
INTEGER(8),      INTENT(OUT):: DIM2
INTEGER(8),      INTENT(OUT):: DIM3
INTEGER(8),      INTENT(OUT):: PADDING
END FUNCTION
END INTERFACE
```

The function FCB_NEXT_BINARY skips to the start of the next binary section in the image file on unit *TAPIN* leaving the file positioned for a subsequent read of the image data. The skip may prior to the text field that contains the binary section. When the text filed is reached, it will be scanned for a MIME boundary marker, and, if it is found the subsequence MIME headers will be used to populate the arguments *ENCODING*, *SIZE*, *ID*, *DIGEST*, *COMPRESSION*, *BITS*, *VORZEICHEN*,*REELL*, *BYTEORDER*, *DIMOVER*, *DIM1*, *DIM2*,*DIM3*, *PADDING*.

The value returned in *ENCODING* is taken from the MIME header Content-Transfer-Encoding as an INTEGER. It is returned as 0 if not specified. The reported value is one of the integer values ENC_NONE (Z'0001') for BINARY encoding, ENC_BASE64 (Z'0002') for BASE64 encoding, ENC_BASE32K (Z'0004') for X-BASE32K encoding, ENC_QP (Z'0008') for QUOTED-PRINTABLE encoding, ENC_BASE10 (Z'0010') for BASE10 encoding, ENC_BASE16 (Z'0020') for BASE16 encoding or ENC_BASE8 (Z'0040') for BASE8 encoding. **At this time FCBlib only supports ENC_NONE BINARY encoding**.

The value returned in *SIZE* is taken from the MIME header X-Binary-Size as an INTEGER. It is returned as 0 if not specified.

The value returned in *ID* is taken from the MIME header X-Binary-ID as an INTEGER. It is returned

as 0 if not specified.

The value returned in *DIGEST* is taken from the MIME header Content-MD5. It is returned as a character string. If no digest is given, an empty string is returned.

The value returned in *COMPRESSION* is taken from the MIME header Content-Type in the conversions parameter. The reported value is one of the INTEGER values CBF_CANONICAL (Z'0050'), CBF_PACKED (Z'0060'), CBF_PACKED_V2 (Z'0090'), CBF_BYTE_OFFSET (Z'0070'), CBF_PREDICTOR (Z'0080'), CBF_NONE (Z'0040'). Two flags may be combined with CBF_PACKED or CBF_PACKED_V2: CBF_UNCORRELATED_SECTIONS (Z'0100') or CBF_FLAT_IMAGE (Z'0200'). **At this time FCBlib does not support CBF_PREDICTOR or CBF_NONE compression**.

The values returned in *BITS*, *VORZEICHEN* and *REELL* are the parameters of the data types of the elements. These values are taken from the MIME header X-Binary-Element-Type, which has values of the form "signed *BITS*-bit integer", "unsigned *BITS*-bit integer", "signed *BITS*-bit real IEEE" or "signed *BITS*-bit complex IEEE". If no value is given, *REELL* is reported as -1. If the value in one of the integer types, *REELL* is reported as 0. If the value is one of the real or complex types, *REELL* is reported as 1. **In the current release of FCBlib only the integer types for *BITS* equal to 16 or 32 are supported.**

The value returned in *BYTEORDER* is the byte order of the data in the image file as reported in the MIME header. The value, if specified, will be either the character string "LITTLE_ENDIAN" or the character string "BIG_ENDIAN". If no byte order is specified, "LITTLE_ENDIAN" is reported. This value is taken from the MIME header X-Binary-Element-Byte-Order. **As of this writing, CBFlib will not generate "BIG_ENDIAN" byte-order files. However, both CBFlib and FCBlib read "LITTLE_ENDIAN" byte-order files, even on big-endian machines.**

The value returned in *DIMOVER* is the overall number of elements in the image array, if specified, or zero, if not specified. This value is taken from the MIME header X-Binary-Number-of-Elements. The values returned in *DIM1*, *DIM2* and *DIM3* are the sizes of the fastest changing, second fastest changing and third fastest changing dimensions of the array, if specified, or zero, if not specified. These values are taken from the MIME header X-Binary-Size-Fastest-Dimension, X-Binary-Size-Second-Dimension and X-Binary-Size-Third-Dimension respectively.

The value returned in *PADDING* is the size of the post-data padding, if any, if specified or zero, if not specified. The value is given as a count of octets. This value is taken from the MIME header X-Binary-Size-Padding.

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
LAST_CHAR	The last character (as an INTEGER(1) byte) read.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*
ENCODING	INTEGER type of encoding for the binary section as reported in the MIME header.
ID	INTEGER binary identifier as reported in the MIME header.
SIZE	INTEGER size of compressed binary section as reported in the MIME header.
DIGEST	The MD5 message digest as reported in the MIME header.

COMPRESSION	INTEGER compression method as reported in the MIME header.
BITS	INTEGER number of bits in each element as reported in the MIME header.
VORZEICHEN	INTEGER flag for signed or unsigned elements as reported in the MIME header. Set to 1 if the elements can be read as signed values, 0 otherwise.
REELL	INTEGER flag for real elements as reported in the MIME header. Set to 1 if the elements can be read as REAL
BYTEORDER	The byte order as reported in the MIME header.
DIM1	Pointer to the destination fastest dimension.
DIM2	Pointer to the destination second fastest dimension.
DIM3	Pointer to the destination third fastest dimension.
PADDING	Pointer to the destination padding size.

RETURN VALUE

Returns 0 if the function is successful. **SEE ALSO**

2.5.3 FCB_EXIT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.6 FCB_OPEN_CIFIN

```
  INTERFACE
INTEGER FUNCTION FCB_OPEN_CIFIN(FILNAM,TAPIN,LAST_CHAR,                &
FCB_BYTES_IN_REC,BYTE_IN_FILE,REC_IN_FILE,BUFFER)
CHARACTER(len=*),INTENT(IN) :: FILNAM
INTEGER,         INTENT(IN) :: TAPIN,FCB_BYTES_IN_REC
INTEGER(1),      INTENT(OUT):: LAST_CHAR
INTEGER,         INTENT(OUT):: BYTE_IN_FILE,REC_IN_FILE
INTEGER(1),    INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
INTEGER                     FCB_RECORD_SIZE
END FUNCTION
END INTERFACE
```

The function FCB_OPEN_CIFIN opens the CBF image file given by the file name in the character string *FILNAM* on the logical unit *TAPIN*. The calling routine must provide an INTEGER(1) byte buffer *BUFFER* of some appropriate INTEGER size *FCB_BYTES_IN_REC*. The size must be chosen to suit the machine, but in most cases, 4096 will work. The values returned in *LAST_CHAR*, *BYTE_IN_FILE*, and *REC_IN_FILE* are for use in subsequent FCBlib I/O routines.

The image file will be checked for the initial characters "###CBF: ". If there is no match the error value CBF_FILEREAD is returned.

ARGUMENTS

FILNAM	The character string name of the image file to be opened.
TAPIN	The INTEGER Fortran device unit number assigned to image file.
LAST_CHAR	The last character (as an INTEGER(1) byte) read.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.

BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*

RETURN VALUE

Returns 0 if the function is successful. **SEE ALSO**

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.7 FCB_PACKED: FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2, FCB_DECOMPRESS_PACKED_3D_I4

```
  INTERFACE
INTEGER FUNCTION FCB_DECOMPRESS_PACKED_I2 (ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2,  &
  TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,                  &
  REC_IN_FILE,BUFFER)
INTEGER(2),  INTENT(OUT):: ARRAY(DIM1,DIM2)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN, COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_DECOMPRESS_PACKED_I4 (ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2,  &
  TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,                  &
  REC_IN_FILE,BUFFER)

INTEGER(4),  INTENT(OUT):: ARRAY(DIM1,DIM2)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN, COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_DECOMPRESS_PACKED_3D_I2 (ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2, DIM3,  &
  TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,                  &
```



```
  REC_IN_FILE,BUFFER)
INTEGER(2),  INTENT(OUT):: ARRAY(DIM1,DIM2,DIM3)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN, COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2,DIM3
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_DECOMPRESS_PACKED_3D_I4 (ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2, DIM3,  &
  TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,                  &
  REC_IN_FILE,BUFFER)
INTEGER(4),  INTENT(OUT):: ARRAY(DIM1,DIM2,DIM3)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN, COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2,DIM3
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE
```

The functions FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2 and FCB_DECOMPRESS_PACKED_3D_I4, decompress images compress according the the CBF_PACKED or CBF_PACKED_V2 compression described in section 3.3.2 on J. P. Abrahams CCP4 packed compression.

The relevant function should be called immediately after a call to FCB_NEXT_BINARY, using the values returned by FCB_NEXT_BINARY to select the appropriate version of the function.

ARGUMENTS

ARRAY	The array to receive the image
NELEM	The INTEGER(8) number of elements to be read
NELEM_READ	The INTEGER(8) returned value of the number of elements actually read
ELSIGN	The INTEGER value of the flag for signed (1) OR unsigned (0) data
COMPRESSION	The compression of the image
DIM1	The INTEGER(8) value of the fastest dimension of *ARRAY*
DIM2	The INTEGER(8) value of the second fastest dimension
DIM3	The INTEGER(8) value of the third fastest dimension
TAPIN	The INTEGER Fortran device unit number assigned to image file.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*

RETURN VALUE

Returns 0 if the function is successful.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.8 FCB_READ_BITS

```
  INTERFACE
INTEGER FUNCTION FCB_READ_BITS(TAPIN,FCB_BYTES_IN_REC,BUFFER,     &
               REC_IN_FILE,BYTE_IN_FILE,BCOUNT,BBYTE,             &
               BITCOUNT,IINT,LINT)
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
INTEGER,   INTENT(INOUT):: BCOUNT
INTEGER(1),INTENT(INOUT):: BBYTE
INTEGER,      INTENT(IN):: BITCOUNT
INTEGER,      INTENT(IN):: LINT
INTEGER(4),  INTENT(OUT):: IINT(LINT)
END FUNCTION
END INTERFACE
```

The function FCB_READ_BITS gets the integer value starting at *BYTE_IN_FILE* from file TAPIN continuing through BITCOUNT bits, with sign extension. *BYTE_IN_FILE* is left at the entry value and not incremented. The resulting, sign-extended integer value is stored in the INTEGER(4) array *IINT* of dimension *LINT* with the least significant portion in *IINT*(1).

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
BCOUNT	The INTEGER count of bits remaining unused from the last call to FCB_READ_BITS.
BBYTE	The INTEGER(1) byte containing the unused bits from the last call to FCB_READ_BITS.
BITCOUNT	The INTEGER count of the number of bits to be extracted from the image file.
IINT	The INTEGER(4) array into which to store the value extracted from the image file.
LINT	The INTEGER length of the array *IINT*.

RETURN VALUE

Returns 0 if the function is successful. Because of the use of direct access I/O in blocks of size *FCB_BYTES_IN_REC* the precise location of the end of file may not be detected.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.9 FCB_READ_BYTE

```
  INTERFACE
INTEGER FUNCTION FCB_READ_BYTE(TAPIN,FCB_BYTES_IN_REC,BUFFER,     &
                    REC_IN_FILE,BYTE_IN_FILE,IBYTE)
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
INTEGER(1),  INTENT(OUT):: IBYTE
END FUNCTION
END INTERFACE
```

The function FCB_READ_BYTE reads the byte at the position *BYTE_IN_FILE* in the image file *TAPIN*. The first byte in the file is at *BYTE_IN_FILE* = 1. *BYTE_IN_FILE* should be set to the desired value before the call to the function and is not incremented within the function.

The function attempts to suppress the error caused by a read of a short last record, and in most systems cannot determine the exact location of the end of the image file, returning zero bytes until the equivalent of a full final record has been read.

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
IBYTE	The INTEGER(1) byte found in the image file at the byte position *BYTE_IN_FILE*.

RETURN VALUE

Returns 0 if the function is successful. Because of the use of direct access I/O in blocks of size *FCB_BYTES_IN_REC* the precise location of the end of file may not be detected.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.9 FCB_READ_BITS

2.5.11 FCB_READ_LINE

2.5.10 FCB_READ_IMAGE_I2, FCB_READ_IMAGE_I4, FCB_READ_IMAGE_3D_I2, FCB_READ_IMAGE_3D_I4

```
  INTERFACE
INTEGER FUNCTION FCB_READ_IMAGE_I2(ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2,                         &
  PADDING,TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,             &
  REC_IN_FILE,BUFFER)

INTEGER(2),  INTENT(OUT):: ARRAY(DIM1,DIM2)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN
INTEGER,     INTENT(OUT):: COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2
INTEGER(8),  INTENT(OUT):: PADDING
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_READ_IMAGE_I4(ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2,                         &
  PADDING,TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,             &
  REC_IN_FILE,BUFFER)
INTEGER(4),  INTENT(OUT):: ARRAY(DIM1,DIM2)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN
INTEGER,     INTENT(OUT):: COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2
INTEGER(8),  INTENT(OUT):: PADDING
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_READ_IMAGE_3D_I2(ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2, DIM3,                   &
  PADDING,TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,             &
  REC_IN_FILE,BUFFER)
INTEGER(2),  INTENT(OUT):: ARRAY(DIM1,DIM2,DIM3)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN
INTEGER,     INTENT(OUT):: COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2,DIM3
INTEGER(8),  INTENT(OUT):: PADDING
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
```



```
END FUNCTION
END INTERFACE

  INTERFACE
INTEGER FUNCTION FCB_READ_IMAGE_3D_I4(ARRAY,NELEM,NELEM_READ, &
  ELSIGN, COMPRESSION, DIM1, DIM2, DIM3,                   &
  PADDING,TAPIN,FCB_BYTES_IN_REC,BYTE_IN_FILE,             &
  REC_IN_FILE,BUFFER)
INTEGER(4),  INTENT(OUT):: ARRAY(DIM1,DIM2,DIM3)
INTEGER(8),  INTENT(OUT):: NELEM_READ
INTEGER(8),   INTENT(IN):: NELEM
INTEGER,      INTENT(IN):: ELSIGN
INTEGER,     INTENT(OUT):: COMPRESSION
INTEGER(8),   INTENT(IN):: DIM1,DIM2,DIM3
INTEGER(8),  INTENT(OUT):: PADDING
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC
INTEGER,   INTENT(INOUT):: REC_IN_FILE,BYTE_IN_FILE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC)
END FUNCTION
END INTERFACE
```

The function FCB_READ_IMAGE_I2 reads a 16-bit twos complement INTEGER(2) 2D image. The function FCB_READ_IMAGE_I4 read a 32-bit twos complement INTEGER(4) 2D image. The function FCB_READ_IMAGE_3D_I2 reads a 16-bit twos complement INTEGER(2) 3D image. The function FCB_READ_IMAGE_3D_I4 reads a 32-bit twos complement INTEGER(4) 3D image. In each case the image is compressed either by a BYTE_OFFSET algorithm by W. Kabsch based on a proposal by A. Hammersley or by a PACKED algorithm by J. P. Abrahams as used in CCP4, with modifications by P. Ellis and H. J. Bernstein.

The relevant function automatically first calls FCB_NEXT_BINARY to skip to the next binary section and then starts to read. An error return will result if the parameters of this call are inconsistent with the values in MIME header.

ARGUMENTS

ARRAY	The array to receive the image
NELEM	The INTEGER(8) number of elements to be read
NELEM_READ	The INTEGER(8) returned value of the number of elements actually read
ELSIGN	The INTEGER value of the flag for signed (1) OR unsigned (0) data
COMPRESSION	The actual compression of the image
DIM1	The INTEGER(8) value of the fastest dimension of *ARRAY*
DIM2	The INTEGER(8) value of the second fastest dimension
DIM3	The INTEGER(8) value of the third fastest dimension
TAPIN	The INTEGER Fortran device unit number assigned to image file.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*

RETURN VALUE

Returns 0 if the function is successful.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.7 FCB_DECOMPRESS: FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2, FCB_DECOMPRESS_PACKED_3D_I4
2.5.9 FCB_READ_BYTE
2.5.11 FCB_READ_LINE

2.5.11 FCB_READ_LINE

```
  INTERFACE
INTEGER FUNCTION FCB_READ_LINE(TAPIN,LAST_CHAR,FCB_BYTES_IN_REC,  &
                  BYTE_IN_FILE,REC_IN_FILE,BUFFER,LINE,N,LINELEN)
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC,N
INTEGER,   INTENT(INOUT):: BYTE_IN_FILE,REC_IN_FILE
INTEGER,     INTENT(OUT):: LINELEN
INTEGER(1),INTENT(INOUT):: LAST_CHAR,BUFFER,(FCB_BYTES_IN_REC)
INTEGER(1),  INTENT(OUT):: LINE(N)
END FUNCTION
END INTERFACE
```

The function FCB_READ_LINE reads successive bytes into the INTEGER(1) byte array *LINE* of dimension *N*), stopping at *N* bytes or the first error or the first CR (Z'0D') or LF (Z'0A'), whichever comes first. It discards an LF after a CR. The variable *LAST_CHAR* is checked for the last character from the previous line to make this determination.

The actual number of bytes read into the line, not including any terminal CR or LF is stored in *LINELEN*.

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
LAST_CHAR	The INTEGER(1) byte holding the ASCII value of the last character read for each line read.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*.
LINE	The INTEGER(1) array of length *N* to hold the line to be read from *TAPIN*.
N	The INTEGER dimension of *LINE*.
LINELEN	The INTEGER number of characters read into LINE.

RETURN VALUE

Returns 0 if the function is successful.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.7 FCB_DECOMPRESS: FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2, FCB_DECOMPRESS_PACKED_3D_I4
2.5.9 FCB_READ_BYTE

2.5.12 FCB_READ_XDS_I2

```
  INTERFACE
INTEGER FUNCTION FCB_READ_XDS_I2(FILNAM,TAPIN,NX,NY,IFRAME,JFRAME)
CHARACTER(len=*),INTENT(IN) :: FILNAM
INTEGER,         INTENT(IN) :: TAPIN,NX,NY
INTEGER(2),      INTENT(OUT):: IFRAME(NX*NY)
INTEGER(4),      INTENT(OUT):: JFRAME(NX,NY)
END FUNCTION
END INTERFACE
```

The function FCB_READ_XDS_I2 read a 32-bit integer twos complement image into a 16-bit INTEGER(2) XDS image using the CBF_BYTE_OFFSET, CBF_PACKED or CBF_PACKED_V2 compressions for the 32-bit data. The BYTE_OFFSET algorithm is a variant of the September 2006 version by W. Kabsch which was based on a suggestion by A. Hammersley and which was further modified by H. Bernstein.

The file named *FILNAM* is opened on the logical unit *TAPIN* and FCB_NEXT_BINARY is used to skip to the next binary image. The binary image is then decompressed to produce an XDS 16-bit integer image array *IFRAME* which is *NX* by *NY*. The dimensions must agree with the dimensions specified in MIME header.

The conversion from a 32-bit integer I32 to 16-bit XDS pixel I16 is done as per W. Kabsch as follows: The value I32 is limited to the range -1023 ≤ I32 ≤ 1048576. If I32 is outside that range it is truncated to the closer boundary. The generate I16, the 16-bit result, if I32 > 32767, it is divided by 32 (producing a number between 1024 and 32768), and then negated (producing a number between -1024 and -32768).

For CBF_BYTE_OFFSET this conversion can be done on the fly directly into the target array *IFRAME*, but for the CBF_PACKED or CBF_PACKED_V2, the full 32 bit precision is needed during the decompression, forcing the use of an intermediate INTEGER(4) array *JFRAME* to hold the 32-bit image in that case.

The image file is closed after reading one image.

ARGUMENTS

FILNAM	The character string name of the image file to be opened.
TAPIN	The INTEGER Fortran device unit number assigned to image file.
NX	The INTEGER fast dimension of the image array.
NY	The INTEGER slow dimension of the image array.
IFRAME	The INTEGER(2) XDS image array.
JFRAME	The INTEGER(4) 32-bit image scratch array needed for CBF_PACKED or CBF_PACKED_V2 images.

RETURN VALUE

Returns 0 if the function is successful, CBF_FORMAT (=1) if it cannot handle this CBF format (not implemented), -1 if it cannot determine endian architecture of this machine, -2: if it cannot open the image file, -3: if it finds the wrong image format and -4 if it cannot read the image.

2.5.13 FCB_SKIP_WHITESPACE

```
  INTERFACE
INTEGER FUNCTION FCB_SKIP_WHITESPACE(TAPIN,LAST_CHAR,             &
                FCB_BYTES_IN_REC,BYTE_IN_FILE,REC_IN_FILE,BUFFER,&
                LINE,N,LINELEN,ICUR,FRESH_LINE)
INTEGER,      INTENT(IN):: TAPIN,FCB_BYTES_IN_REC,N
INTEGER,   INTENT(INOUT):: BYTE_IN_FILE,REC_IN_FILE,LINELEN,ICUR, &
                           FRESH_LINE
INTEGER(1),INTENT(INOUT):: BUFFER(FCB_BYTES_IN_REC),LINE(N),      &
                           LAST_CHAR
END INTERFACE
```

The function FCB_SKIP_WHITESPACE skips forward on the current INTEGER(1) byte array *LINE* of size *N* with valid data in *LINE*(1:*LINELEN*) from the current position *ICUR* moving over MIME header whitespace and comments, reading new lines into *LINE* if needed. The flag *FRESH_LINE* indicates that a fresh line should be read on entry.

ARGUMENTS

TAPIN	The INTEGER Fortran device unit number assigned to image file.
LAST_CHAR	The INTEGER(1) byte holding the ASCII value of the last character read for each line read.
FCB_BYTES_IN_REC	The INTEGER number of bytes in a record.
BYTE_IN_FILE	The INTEGER byte (counting from 1) of the byte to read.
REC_IN_FILE	The INTEGER record number (counting from 1) of next record to read.
BUFFER	The INTEGER(1) array of length *FCB_BYTES_IN_REC* to hold the appropriate record from *TAPIN*.
LINE	The INTEGER(1) array of length *N* to hold the line to be read from *TAPIN*.
N	The INTEGER dimension of *LINE*.
LINELEN	The INTEGER number of characters read into LINE.
ICUR	The INTEGER position within the line.
FRESH_LINE	The INTEGER flag that a fresh line is needed.

RETURN VALUE

Returns 0 if the function is successful.

SEE ALSO

2.5.3 FCB_EXIT_BINARY
2.5.5 FCB_NEXT_BINARY
2.5.6 FCB_OPEN_CIFIN
2.5.7 FCB_DECOMPRESS: FCB_DECOMPRESS_PACKED_I2, FCB_DECOMPRESS_PACKED_I4, FCB_DECOMPRESS_PACKED_3D_I2, FCB_DECOMPRESS_PACKED_3D_I4
2.5.9 FCB_READ_BYTE

3. File format

3.1 General description

With the exception of the binary sections, a CBF file is an mmCIF-format ASCII file, so a CBF file with no binary sections is a CIF file. An imgCIF file has any binary sections encoded as CIF-format ASCII strings and is a CIF file whether or not it contains binary sections. In most cases, CBFlib can also be used to access normal CIF files as well as CBF and imgCIF files.

3.2 Format of the binary sections

Before getting to the binary data itself, there are some preliminaries to allow a smooth transition from the conventions of CIF to those of raw or encoded streams of "octets" (8-bit bytes). The binary data is given as the essential part of a specially formatted semicolon-delimited CIF multi-line text string. This text string is the value associated with the tag "_array_data.data".

The specific format of the binary sections differs between an imgCIF and a CBF file.

3.2.1 Format of imgCIF binary sections

Each binary section is encoded as a semicolon-delimited string. Within the text string, the conventions developed for transmitting email messages including binary attachments are followed. There is secondary ASCII header information, formatted as Multipurpose Internet Mail Extensions (MIME) headers (see RFCs 2045-49 by Freed, et al.). The boundary marker for the beginning of all this is the special string

```
--CIF-BINARY-FORMAT-SECTION--
```

at the beginning of a line. The initial "--" says that this is a MIME boundary. We cannot put "###" in front of it and conform to MIME conventions. Immediately after the boundary marker are MIME headers, describing some useful information we will need to process the binary section. MIME headers can appear in different orders, and can be very confusing (look at the raw contents of a email message with attachments), but there is only one header which is has to be understood to process an imgCIF: "Content-Transfer-Encoding". If the value given on this header is "BINARY", this is a CBF and the data will be presented as raw binary, containing a count (in the header described in 3.2.2 Format of CBF binary sections) so that we'll know when to start looking for more information.

If the value given for "Content-Transfer-Encoding" is one of the real encodings: "BASE64", "QUOTED-PRINTABLE", "X-BASE8", "X-BASE10" or "X-BASE16", the file is an imgCIF, and we'll need some other headers to process the encoded binary data properly. It is a good practice to give headers in all cases. The meanings of various encodings is given in the CBF extensions dictionary, cif_img_1.5.1.dic, as one html file, or as separate pages for each defintion.

For certain compressions (e.g. CBF_PACKED) MIME headers are essential to determine the parameters of the compression. The full list of MIME headers recognized by and generated by CBFlib is:

- Content-Type:
- Content-Transfer-Encoding:
- Content-MD5:
- X-Binary-Size:

- X-Binary-ID:
- X-Binary-Element-Type:
- X-Binary-Element-Byte-Order:
- X-Binary-Number-of-Elements:
- X-Binary-Size-Fastest-Dimension:
- X-Binary-Size-Second-Dimension:
- X-Binary-Size-Third-Dimension:
- X-Binary-Size-Padding:

- Content-Type:

  The "Content-Type" header tells us what sort of data we have (currently always "application/octet-stream" for a miscellaneous stream of binary data) and, optionally, the conversions that were applied to the original data. The default is to compress the data with the "CBF-PACKED" algorithm. The Content-Type may be any of the discrete types permitted in RFC 2045; 'application/octet-stream' is recommended. If an octet stream was compressed, the compression should be specified by the parameter 'conversions="**X**-CBF_PACKED"' or the parameter 'conversions="**X**-CBF_PACKED_V2"' or the parameter 'conversions="**X**-CBF_CANONICAL"' or the parameter 'conversions="**X**-CBF_BYTE_OFFSET"'

  If the parameter 'conversions="**X**-CBF_PACKED"' or 'conversions="**X**-CBF_PACKED_V2"' is given it may be further modified with the parameters '"uncorrelated_sections"' or '"flat"'

  If the '"uncorrelated_sections"' parameter is given, each section will be compressed without using the prior section for averaging. If the '"flat"' parameter is given, each the image will be treated as one long row.

- Content-Transfer-Encoding:

  The "Content-Transfer-Encoding" may be 'BASE64', 'Quoted-Printable', '**X**-BASE8', '**X**-BASE10', '**X**-BASE16' or '**X**-BASE32K', for an imgCIF or 'BINARY' for a CBF. The octal, decimal and hexadecimal transfer encodings are provided for convenience in debugging and are not recommended for archiving and data interchange.

  In a CIF, one of the parameters 'charset=us-ascii', 'charset=utf-8' or 'charset=utf-16' may be used on the Content-Transfer-Encoding to specify the character set used for the external presentation of the encoded data. If no charset parameter is given, the character set of the enclosing CIF is assumed. In any case, if a BOM flag is detected (FE FF for big-endian UTF-16, FF FE for little-endian UTF-16 or EF BB BF for UTF-8) is detected, the indicated charset will be assumed until the end of the encoded data or the detection of a different BOM. The charset of the Content-Transfer-Encoding is not the character set of the encoded data, only the character set of the presentation of the encoded data and should be respecified for each distinct STAR string.

  In an imgCIF file, the encoded binary data begins after the empty line terminating the header. In an imgCIF file, the encoded binary data ends with the terminating boundary delimiter '\n--CIF-BINARY-FORMAT-SECTION----' in the currently effective charset or with the '\n; ' that terminates the STAR string.

  In a CBF, the raw binary data begins after an empty line terminating the header and after the sequence:

```
Octet   Hex   Decimal  Purpose
  0     0C       12    (ctrl-L) Page break
  1     1A       26    (ctrl-Z) Stop listings in MS-DOS
```



```
2     04       04    (Ctrl-D) Stop listings in UNIX
3     D5      213    Binary section begins
```

None of these octets are included in the calculation of the message size or in the calculation of the message digest.

- Content-MD5:

  An MD5 message digest may, optionally, be used. The 'RSA Data Security, Inc. MD5 Message-Digest Algorithm' should be used. No portion of the header is included in the calculation of the message digest. The optional "Content-MD5" header provides a much more sophisticated check on the integrity of the binary data than size checks alone can provide.

- X-Binary-Size:

  The "X-Binary-Size" header specifies the size of the equivalent binary data in octets. This is the size **after** any compressions, but before any ascii encodings. This is useful in making a simple check for a missing portion of this file. The 8 bytes for the Compression type (see below) are not counted in this field, so the value of "X-Binary-Size" is 8 less than the quantity in bytes 12-19 of the raw binary data (3.2.2 Format of CBF binary sections).

- X-Binary-ID:

  The "X-Binary-ID" header should contain the same value as was given for "_array_data.binary_id".

- X-Binary-Element-Type:

  The "X-Binary-Element-Type" header specifies the type of binary data in the octets, using the same descriptive phrases as in _array_structure.encoding_type. The default value is 'unsigned 32-bit integer'.

- X-Binary-Element-Byte-Order:

  The "X-Binary-Element-Byte-Order" can specify either '"BIG_ENDIAN"' or '"LITTLE_ENDIAN"' byte order of the imaage data. CBFlib only writes '"LITTLE_ENDIAN"', and in general can only process LITTLE_ENDIAN even on machines that are BIG_ENDIAN.

- X-Binary-Number-of-Elements:

  The "X-Binary-Number-of-Elements" specifies the number of elements (not the number of octets) in the decompressed, decoded image.

- X-Binary-Size-Fastest-Dimension:

  The optional "X-Binary-Size-Fastest-Dimension" specifies the number of elements (not the number of octets) in one row of the fastest changing dimension of the binary data array. This information must be in the MIME header for proper operation of some of the decompression algorithms.

- X-Binary-Size-Second-Dimension:

  The optional "X-Binary-Size-Second-Dimension" specifies the number of elements (not the number of octets) in one column of the second-fastest changing dimension of the binary data array. This information must be in the MIME header for proper operation of some of the

decompression algorithms.

- X-Binary-Size-Third-Dimension:

  The optional "X-Binary-Size-Third-Dimension" specifies the number of sections for the third-fastest changing dimension of the binary data array.

- X-Binary-Size-Padding:

  The optional "X-Binary-Size-Padding" specifies the size in octets of an optional padding after the binary array data and before the closing flags for a binary section. CBFlib always writes this padding as zeros, but this information should be in the MIME header for a binary section that uses padding, especially if non-zero padding is used.

A blank line separator immediately precedes the start of the encoded binary data. Blank spaces may be added prior to the preceding "line separator" if desired (e.g. to force word or block alignment).

Because CBFLIB may jump forward in the file from the MIME header, the length of encoded data cannot be greater than the value defined by "X-Binary-Size" (except when "X-Binary-Size" is zero, which means that the size is unknown), unless "X-Binary-Size-Padding" is specified to allow for the padding. At exactly the byte following the full binary section as defined by the length and padding values is the end of binary section identifier. This consists of the line-termination sequence followed by:

```
--CIF-BINARY-FORMAT-SECTION----
;
```

with each of these lines followed by a line-termination sequence. This brings us back into a normal CIF environment. This identifier is, in a sense, redundant because the binary data length value tells the a program how many bytes to jump over to the end of the binary data. This redundancy has been deliberately added for error checking, and for possible file recovery in the case of a corrupted file and this identifier must be present at the end of every block of binary data.

3.2.2 Format of CBF binary sections

In a CBF file, each binary section is encoded as a ;-delimited string, starting with an arbitrary number of pure-ASCII characters.

Note: For historical reasons, CIFlib has the option of writing simple header and footer sections: "START OF BINARY SECTION" at the start of a binary section and "END OF BINARY SECTION" at the end of a binary section, or writing MIME-type header and footer sections (3.2.1 Format of imgCIF binary sections). If the simple header is used, the actual ASCII text is ignored when the binary section is read. **Use of the simple binary header is deprecated.**

The MIME header is recommended.

Between the ASCII header and the actual CBF binary data is a series of bytes ("octets") to try to stop the listing of the header, bytes which define the binary identifier which should match the "binary_id" defined in the header, and bytes which define the length of the binary section.

Octet	Hex	Decimal	Purpose
1	0C	12	(ctrl-L) End of Page
2	1A	26	(ctrl-Z) Stop listings in MS-DOS

3	04	04	(Ctrl-D) Stop listings in UNIX
4	D5	213	Binary section begins
5..5+n-1			Binary data (n octets)

NOTE: When a MIME header is used, only bytes 5 through 5+n-1 are considered in computing the size and the message digest, and only these bytes are encoded for the equivalent imgCIF file using the indicated Content-Transfer-Encoding.

If no MIME header has been requested (a deprecated use), then bytes 5 through 28 are used for three 8-byte words to hold the binary_id, the size and the compression type:

5..12	Binary Section Identifier (See _array_data.binary_id) 64-bit, little endian
13..20	The size (n) of the binary section in octets (i.e. the offset from octet 29 to the first byte following the data)
21..28	Compression type: CBF_NONE 0x0040 (64) CBF_CANONICAL 0x0050 (80) CBF_PACKED 0x0060 (96) CBF_BYTE_OFFSET 0x0070 (112) CBF_PREDICTOR 0x0080 (128) ...

The binary data then follows in bytes 29 through 29+n-1.

The binary characters serve specific purposes:

- The Control-L (from-feed) will terminate printing of the current page on most operating systems.
- The Control-Z will stop the listing of the file on MS-DOS type operating systems.
- The Control-D will stop the listing of the file on Unix type operating systems.
- The unsigned byte value 213 (decimal) is binary 11010101. (Octal 325, and hexadecimal D5). This has the eighth bit set so can be used for error checking on 7-bit transmission. It is also asymmetric, but with the first bit also set in the case that the bit order could be reversed (which is not a known concern).
- (The carriage return, line-feed pair before the START_OF_BIN and other lines can also be used to check that the file has not been corrupted e.g. by being sent by ftp in ASCII mode.)

  At present four compression schemes are implemented are defined: CBF_NONE (for no compression), CBF_CANONICAL (for and entropy-coding scheme based on the canonical-code algorithm described by Moffat, *et al.* (*International Journal of High Speed Electronics and Systems*, Vol 8, No 1 (1997) 179-231)), CBF_PACKED or CBF_PACKED_V2 for J. P.

Abrahams CCP4-style packing schemes and CBF_BYTE_OFFSET for a simple byte_offset compression scheme.. Other compression schemes will be added to this list in the future.

For historical reasons, CBFlib can read or write a binary string without a MIME header. The structure of a binary string with simple headers is:

Byte	ASCII symbol	Decimal value	Description
1	;	59	Initial ; delimiter
2	carriage-return	13	
3	line-feed	10	The CBF new-line code is carriage-return, line-feed
4	S	83	
5	T	84	
6	A	65	
7	R	83	
8	T	84	
9		32	
10	O	79	
11	F	70	
12		32	
13	B	66	
14	I	73	
15	N	78	
16	A	65	
17	R	83	
18	Y	89	
19		32	
20	S	83	
21	E	69	
22	C	67	
23	T	84	
24	I	73	
25	O	79	
26	N	78	
27	carriage-return	13	
28	line-feed	10	
29	form-feed	12	
30	substitute	26	Stop the listing of the file in MS-DOS
31	end-of-transmission	4	Stop the listing of the file in unix
32		213	First non-ASCII value
33 .. 40			Binary section identifier (64-bit little-endien)
41 .. 48			Offset from byte 57 to the first ASCII character following the binary data

49 .. 56			Compression type
57 .. 57 + *n*-1			Binary data (*n*bytes)
57 + *n*	carriage-return	13	
58 + *n*	line-feed	10	
59 + *n*	E	69	
60 + *n*	N	78	
61 + *n*	D	68	
62 + *n*		32	
63 + *n*	O	79	
64 + *n*	F	70	
65 + *n*		32	
66 + *n*	B	66	
67 + *n*	I	73	
68 + *n*	N	78	
69 + *n*	A	65	
70 + *n*	R	83	
71 + *n*	Y	89	
72 + *n*		32	
73 + *n*	S	83	
74 + *n*	E	69	
75 + *n*	C	67	
76 + *n*	T	84	
77 + *n*	I	73	
78 + *n*	O	79	
79 + *n*	N	78	
80 + *n*	carriage-return	13	
81 + *n*	line-feed	10	
82 + *n*	;	59	Final ; delimiter

3.3 Compression schemes

Two schemes for lossless compression of integer arrays (such as images) have been implemented in this version of CBFlib:

1. An entropy-encoding scheme using canonical coding
2. A CCP4-style packing scheme.

Both encode the difference (or error) between the current element in the array and the prior element. Parameters required for more sophisticated predictors have been included in the compression functions and will be used in a future version of the library.

3.3.1 Canonical-code compression

The canonical-code compression scheme encodes errors in two ways: directly or indirectly. Errors are coded directly using a symbol corresponding to the error value. Errors are coded indirectly using a symbol for the number of bits in the (signed) error, followed by the error iteslf.

At the start of the compression, CBFlib constructs a table containing a set of symbols, one for each of the $2^{\wedge n}$ direct codes from $-2^{\wedge(n-1)}$ .. $2^{\wedge(n-1)}-1$, one for a stop code, and one for each of the *maxbits -n* indirect codes, where *n* is chosen at compress time and *maxbits* is the maximum number of bits in an error. CBFlib then assigns to each symbol a bit-code, using a shorter bit code for the more common symbols and a longer bit code for the less common symbols. The bit-code lengths are calculated using a Huffman-type algorithm, and the actual bit-codes are constructed using the canonical-code algorithm described by Moffat, *et al*. (*International Journal of High Speed Electronics and Systems*, Vol 8, No 1 (1997) 179-231).

The structure of the compressed data is:

Byte	Value
1 .. 8	Number of elements (64-bit little-endian number)
9 .. 16	Minimum element
17 .. 24	Maximum element
25 .. 32	(reserved for future use)
33	Number of bits directly coded, *n*
34	Maximum number of bits encoded, *maxbits*
35 .. $35+2^{\wedge n}-1$	Number of bits in each direct code
$35+2^{\wedge n}$	Number of bits in the stop code
$35+2^{\wedge n}+1$ .. $35+2^{\wedge n}+$*maxbits-n*	Number of bits in each indirect code
$35+2^{\wedge n}+$*maxbits-n*+1 ..	Coded data

3.3.2 CCP4-style compression

Starting with CBFlib 0.7.7, CBFlib supports three variations on CCP4-style compression: the "flat" version supported in versions of CBFlib prior to release 0.7.7, as well as both version 1 and version 2 of J. P. Abrahams "pack_c" compression.

The CBF_PACKED and CBF_PACKED_V2 compression and decompression code incorporated in CBFlib is derived in large part from the J. P. Abrahams pack_c.c compression code in CCP4. This code is incorporated in CBFlib under the GPL and the LGPL with both the permission Jan Pieter Abrahams, the original author of pack_c.c (email from Jan Pieter Abrahams of 15 January 2007) and of the CCP4 project (email from Martyn Winn on 12 January 2007). The cooperation of J. P. Abrahams and of the CCP4 project is gratefully acknowledged.

The basis for all three versions is a scheme to pack offsets (differences from a base value) into a small-endian bit stream. The stream is organized into blocks. Each block begins with a header of 6 bits in the flat packed version and version 1 of J. P. Abrahams compression, and 7 bits in version 2 of J. P. Abrahams compression. The header gives the number of offsets that follow and the number of bits in each offset. Each offset is a signed, 2's complement integer.

The first 3 bits in the header gives the logarithm base 2 of the numer of offsets that follow the header. For example, if a header has a zero in bits, only one offset follows the header. If those same bits contain the number n, the number of offsets in the block is 2^n.

The following 3 bits (flat and version 1) or 4 bits (version 2) contains a number giving an index into a table of bit-lengths for the offsets. All offsets in a given block are of the same length.

Bits 3 .. 5 (flat and version 1) or bits 3 .. 6 (version 2) encode the number of bits in each offset as follows:

Value in bits 3 .. 5	Number of bits in each V1 offset	Number of bits in each V2 offset
0	0	0
1	4	3
2	5	4
3	6	5
4	7	6
5	8	7
6	16	8
7	max	9
8		10
9		11
10		12
11		13
12		14
13		15
14		16
15		max

The value "max" is determined by the compression version and the element size. If the compression used is "flat", then "max" is 65. If the compression is version 1 or version 2 of the JPA compression, then "max" is the number of bits in each element, i.e. 8, 16, 32 or 64 bits.

The major difference between the three variants of packed compression is the choice of the base value from which the offset is measured. In all cases the first offset is measured from zero, i.e. the first offset is the value of the first pixel of the image. If "flat" is chosen or if the dimensions of the data array are not given, then the remaining offset are measure against the prior value, making it similar in approach to the "byte offset" compression described in section 3.3.3 Byte offset compression, but with a more efficient representation of the offsets.

In version 1 and version 2 of the J. P. Abrahams compression, the offsets are measured against an average of earlier pixels. If there is only one row only the prior pxiel is used, starting with the same offsets for that row as for "flat". After the first row, three pixels from the prior row are used in addition to using the immediately prior pixel. If there are multiple sections, and the sections are marked as correlated, after the first section, 4 pixels from the prior section are included in the average. The CBFlib code differs from the pack_c code in the handling of the beginnings and ends of rows and sections. The pack_c code will use pixels from the other side of the image in doing the averaging. The CBFlib code drops pixels from the other side of the image from the pool. The details follow.

After dealing with the special case of the first pixel, The algorithm uses an array of pointers, trail_char_data. The assignment of pixels to the pool to be averaged begins with trail_char_data[0] points to the pixel immediately prior to the next pixel to be processed, either in the same row (fastest index) or, at the end of the prior row if the next data element to be processed is at the end of a row. The location of the pixel pointed to by trail_char_data[0] is used to compute the locations of the other pixels in the pool. It will be dropped from the pool before averaging if it is on the opposite side of the image.

The pool will consist of 1, 2, 4 or 8 pixels.

Assume ndim1, ndim2, ndim3 are the indices of the same pixel as trail_char_data[0] points to. These indices are incremented to be the indices of the next pixel to be processed before populating trail_char_data.

On exit, trail_char_data[0 .. 7] will have been populated with pointers to the pixels to be used in forming the average. Pixels that will not be used will be set to NULL. Note that trail_char_data[0] may be set to NULL.

If we mark the next element to be processed with a "*" and the entries in trail_char_data with their array indices 0 .. 7, the possible patterns of settings in the general case are:

current section:

```
- - - - 0 * - - - -
- - - - 3 2 1 - - -
- - - - - - - - - -
```

prior section:

```
- - - - - 4 - - - -
- - - - 7 6 5 - - -
- - - - - - - - - -
```

If there is no prior section (i.e. ndim3 is 0, or the CBF_UNCORRELATED_SECTIONS flag is set to indicate discontinuous sections), the values for trail_char_data[4 .. 7] will all be NULL. When there is a prior section, trail_char_data[5..7] are pointers to the pixels immediately below the elements pointed to by trail_char_data[1..3], except trail_char_data[4] is one element further along its row to be directly below the next element to be processed.

The first element of the first row of the first section is a special case, with no averaging.

In the first row of the first section (ndim2 == 0, and ndim3 == 0), after the first element (ndim1 > 0), only trail_char_data[0] is used

current section:

```
- - - - 0 * - - - -
```

For subsequent rows of the first section (ndim2 > 0, and ndim3 == 0), for the first element (ndim1 == 0), two elements from the prior row are used:

current section:


```
* - - - - - - - - -
2 1 - - - - - - - -
- - - - - - - - - -
```

while for element after the first element, but before the last element of the row, a full set of 4 elements is used:

current section:

```
- - - - 0 * - - - -
- - - - 3 2 1 - - -
- - - - - - - - - -
```

For the last element of a row (ndim1 == dim1-1), two elements are used

current section:

```
- - - - - - - - 0 *
- - - - - - - - - 2
- - - - - - - - - -
```

For sections after the first section, provided the CBF_UNCORRELATED_SECTIONS flag is not set in the compression, for each non-NULL entry in trail_char_data [0..3] an entry is made in trail_char_data [4..7], except for the first element of the first row of a section. In that case an entry is made in trail_char_data[4].

The structure of the compressed data is:

Byte	Value
1 .. 8	Number of elements (64-bit little-endian number)
9 .. 16	Minumum element (currently unused)
17 .. 24	Maximum element (currently unused)
25 .. 32	(reserved for future use)
33 ..	Coded data

3.3.3 Byte_offset compression

Starting with CBFlib 0.7.7, CBFlib supports a simple and efficient "byte_offset" algorithm originally proposed by Andy Hammerley and modified by Wolgang Kabsch and Herbert Bernstein. The original proposal was called "byte_offsets". We distinguish this variant by calling it "byte_offset". The major differences are that the "byte_offsets" algorithm started with explicit storage of the first element of the array as a 4-byte signed two's integer, and checked for image edges to changes the selection of prior pixel. The CBFlib "byte_offset" alogorithm starts with an assumed zero before the first pixel and represents the value of the first pixel as an offset of whatever number of size is needed to hold the value, and for speed, treats the entire image as a simple linear array, allowing use of the last pixel of one row as the base against which to compute the offset for the first element of the next row.

The algorithm is simple and easily implemented. This algorithm can never achieve better than a factor of two compression relative to 16-bit raw data or 4 relative to 32-bit raw data, but for most diffraction data the compression will indeed be very close to these ideal values. It also has the advantage that integer values up to 32 bits (or 31 bits and sign) may be stored efficiently without the need for special over-load tables. It is a fixed algorithm which does not need to calculate any image statistics, so is fast.

The algorithm works because of the following property of almost all diffraction data and much other image data: The value of one element tends to be close to the value of the adjacent elements, and the vast majority of the differences use little of the full dynamic range. However, noise in experimental data means that run-length encoding is not useful (unless the image is separated into different bit-planes). If a variable length code is used to store the differences, with the number of bits used being inversely proportional to the probability of occurrence, then compression ratios of 2.5 to 3.0 may be achieved. However, the optimum encoding becomes dependent of the exact properties of the image, and in particular on the noise. Here a lower compression ratio is achieved, but the resulting algorithm is much simpler and more robust.

The "byte_offset" compression algorithm is the following:

1. Start with a base pixel value of 0.
2. Compute the difference delta between the next pixel value and the base pixel value.
3. If -127 ≤ delta ≤ 127, output delta as one byte, make the current pixel value the base pixel value and return to step 2.
4. Otherwise output -128 (F0 hex).
5. We still have to output delta. If -32767 ≤ delta ≤ 32767, output delta as a little_endian 16-bit quantity, make the current pixel value the base pixel value and return to step 2.
6. Otherwise output -32768 (F000 hex, little_endian, i.e. 00 then F0)
7. We still have to output delta. If -2147483647 ≤ delta ≤ 2147483647, output delta as a little_endian 32 bit quantity, make the current pixel value the base pixel value and return to step 2.
8. Otherwise output -2147483648 (F0000000 hex, little_endian, i.e. 00, then 00, then 00, then F0) and then output the pixel value as a little-endian 64 bit quantity, make the current pixel value the base pixel value and return to step 2.

The "byte_offset" decompression algorithm is the following:

1. Start with a base pixel value of 0.
2. Read the next byte as delta
3. If -127 ≤ delta ≤ 127, add delta to the base pixel value, make that the new base pixel value, place it on the output array and return to step 2.
4. If delta is F0 hex, read the next two bytes as a little_endian 16-bit number and make that delta.
5. If -32767 ≤ delta ≤ 32767, add delta to the base pixel value, make that the new base pixel value, place it on the output array and return to step 2.
6. If delta is F000 hex, read the next 4 bytes as a little_endian 32-bit number and make that delta
7. If -2147483647 ≤ delta ≤ 2147483647, add delta to the base pixel value, make that the new base pixel value, place it on the output array and return to step 2.
8. If delta is F0000000 hex, read the next 4 bytes as a little_endian 32-bit number and make that delta, read the next 8 bytes as a little_endia 64-bit number and make that delta, add delta to the base pixel value, make that the new base pixel value, place it on the output array and return to step 2.

Let us look at an example, of two 1000 x 1000 flat field images presented as a mimimal imgCIF file. The first image uses 32-bit unsigned integers and the second image uses 16-bit unsigned integers.

The imgCIF file begins with some identifying comments (magic numbers) to track the version of the dictionary and library:


```
###CBF: VERSION 1.5
# CBF file written by CBFlib v0.7.7
```

This is followed by the necessary syntax to start a CIF data block and by whatever tags and values are appropriate to describe the experiment. The minimum is something like

```
data_testflat
```

eventually we come to the actual binary data, which begins the loop header for the array_data category

```
loop_
_array_data.data
```

with any additional tags needed, and then the data itself, which starts with the mini-header:

```
;
--CIF-BINARY-FORMAT-SECTION--
Content-Type: application/octet-stream;
     conversions="x-CBF_BYTE_OFFSET"
Content-Transfer-Encoding: BINARY
X-Binary-Size: 1000002
X-Binary-ID: 1
X-Binary-Element-Type: "unsigned 32-bit integer"
X-Binary-Element-Byte-Order: LITTLE_ENDIAN
Content-MD5: +FqUJGxXhvCijXMFHC0kaA==
X-Binary-Number-of-Elements: 1000000
X-Binary-Size-Fastest-Dimension: 1000
X-Binary-Size-Second-Dimension: 1000
X-Binary-Size-Padding: 4095
```

followed by an empty line and then the sequence of characters:

```
^L^Z^D<D5>
```

followed immediately by the compressed data.

The binary data begins with the hex byte 80 to flag the need for a value that will not fit in one byte. That is followed by the small_endian hex value 3E8 saying that the first delta is 1000. Then 999,999 bytes of zero follow, since this is a flat field, with all values equal to zero. That gives us our entire 1000x1000 compressed flat field. However, because we asked for 4095 bytes of padding, there is an additional 4095 bytes of zero that are not part of the compressed field. They are just pad and can be ignored. Finally, after the pad, the CIF text field that began with

```
;
--CIF-BINARY-FORMAT-SECTION--
```

is completed with

```
--CIF-BINARY-FORMAT-SECTION----
;
```

notice the extra --

The second flat field then follows, with a very similar mini-header:

```
;
--CIF-BINARY-FORMAT-SECTION--
```

```
Content-Type: application/octet-stream;
     conversions="x-CBF_BYTE_OFFSET"
Content-Transfer-Encoding: BINARY
X-Binary-Size: 1000002
X-Binary-ID: 2
X-Binary-Element-Type: "unsigned 16-bit integer"
X-Binary-Element-Byte-Order: LITTLE_ENDIAN
Content-MD5: +FqUJGxXhvCijXMFHC0kaA==
X-Binary-Number-of-Elements: 1000000
X-Binary-Size-Fastest-Dimension: 1000
X-Binary-Size-Second-Dimension: 1000
X-Binary-Size-Padding: 4095

^L^Z^D<D5>
```

The only difference is that we have declared this array to be 16-bit and have chosen a different binary id (2 instead of 1). Even the checksum is the same.

4. Installation

CBFlib should be built on a disk with at least 200 megabytes of free space. CBFlib.tar.gz is a "gzipped" tar of the code as it now stands. Place the gzipped tar in the directory that is intended to contain a new directory, named CBFlib_0.7.5 (the "top-level" directory) and uncompress it with gunzip and unpack it with tar:

```
gunzip CBFlib.tar.gz
tar xvf CBFLIB.tar
```

As with prior releases, to run the test programs, you will also need Paul Ellis's sample MAR345 image, example.mar2300, and Chris Nielsen's sample ADSC Quantum 315 image, mb_LP_1_001.img as sample data. Both these files will be extracted by the Makefile from CBFlib_0.7.7_Data_Files. Do not download copies into the top level directory.

After unpacking the archive, the top-level directory should contain a makefile:

Makefile	Makefile for unix

and the subdirectories:

src/	CBFLIB source files
include/	CBFLIB header files
m4/	CBFLIB m4 macro files (used to build .f90 files)
examples/	Example program source files
doc/	Documentation
lib/	Compiled CBFLIB library
bin/	Executable example programs
html_images/	JPEG images used in rendering the HTML files

For instructions on compiling and testing the library, go to the top-level directory and type:

```
make
```

The CBFLIB source and header files are in the "src" and "include" subdirectories. The FCBLIB source

and m4 files are in the "src" and "m4" subdirectories. The files are:

src/	**include/**	**m4/**	**Description**
cbf.c	cbf.h		CBFLIB API functions
cbf_alloc.c	cbf_alloc.h		Memory allocation functions
cbf_ascii.c	cbf_ascii.h		Function for writing ASCII values
cbf_binary.c	cbf_binary.h		Functions for binary values
cbf_byte_offset.c	cbf_byte_offset.h		Byte-offset compression
cbf_canonical.c	cbf_canonical.h		Canonical-code compression
cbf_codes.c	cbf_codes.h		Encoding and message digest functions
cbf_compress.c	cbf_compress.h		General compression routines
cbf_context.c	cbf_context.h		Control of temporary files
cbf_file.c	cbf_file.h		File in/out functions
cbf_lex.c	cbf_lex.h		Lexical analyser
cbf_packed.c	cbf_packed.h		CCP4-style packing compression
cbf_predictor.c	cbf_predictor.h		Predictor-Huffman compression (not implemented)
cbf_read_binary.c	cbf_read_binary.h		Read binary headers
cbf_read_mime.c	cbf_read_mime.h		Read MIME-encoded binary sections
cbf_simple.c	cbf_simple.h		Higher-level CBFlib functions
cbf_string.c	cbf_string.h		Case-insensitive string comparisons
cbf_stx.c	cbf_stx.h		Parser (generated from cbf.stx.y)
cbf_tree.c	cbf_tree.h		CBF tree-structure functions
cbf_uncompressed.c	cbf_uncompressed.h		Uncompressed binary sections
cbf_write.c	cbf_write.h		Functions for writing
cbf_write_binary.c	cbf_write_binary.h		Write binary sections
cbf.stx.y			bison grammar to define cbf_stx.c (see WARNING)
md5c.c	md5.h		RSA message digest software from mpack
	global.h		
fcb_atol_wcnt.f90			Function to convert a string to an integer
fcb_ci_strncmparr.f90			Function to do a case-insensitive comparison of a string to a byte array

fcb_nblen_array.f90	Function to determine the non-blank length of a byte array
fcb_read_byte.f90	Function to read a single byte
fcb_read_line.f90	Function to read a line into a byte array
fcb_skip_whitespace.f90	Function to skip whitespace and comments in a MIME header
fcb_exit_binary.m4	Function to skip past the end of the current binary text field
fcb_next_binary.m4	Function to skip to the next binary
fcb_open_cifin.m4	Function to open a CBF file for reading
fcb_packed.m4	Functions to read a JPA CCP4 compressed image
fcb_read_bits.m4	Functions to read nay number of bits as an integer
fcb_read_image.m4	Functions to read the next image in I2, I4, 3D_I2 and 3D_I4 format
fcb_read_xds_i2.m4	Function to read a single xds image.
fcblib_defines.m4	General m4 macro file for FCBLIB routines.

In the "examples" subdirectory, there are 2 additional files used by the example programs (section 5) for reading MAR300, MAR345 or ADSC CCD images:

img.c img.h Simple image library

and the example programs themselves:

makecbf.c	Make a CBF file from an image
img2cif.c	Make an imgCIF or CBF from an image
cif2cbf.c	Copy a CIF/CBF to a CIF/CBF
convert_image.c	Convert an image file to a cbf using a template file
cif2c.c	Convert a template cbf file into a function to produce the same template in an internal cbf data structure
testcell.C	Exercise the cell functions

as well as three template files: template_adscquantum4_2304x2304.cbf, template_mar345_2300x2300.cbf, and template_adscquantum315_3072x3072.cbf.

Two additional examples (test_fcb_read_image.f90 and test_xds_binary.f90) are created from two files (test_fcb_read_image.m4 and test_xds_binary.m4) in the m4 directory.

The documentation files are in the "doc" subdirectory:

CBFlib.html	This document (HTML)
CBFlib.txt	This document (ASCII)
CBFlib_NOTICES.html	Important NOTICES -- PLEASE READ
CBFlib_NOTICES.txt	Important NOTICES -- PLEASE READ
gpl.txt	GPL -- PLEASE READ
lgpl.txt	LGPL -- PLEASE READ
cbf_definition_rev.txt	Draft CBF/ImgCIF definition (ASCII)
cbf_definition_rev.html	Draft CBF/ImgCIF definition (HTML)
cif_img.html	CBF/ImgCIF extensions dictionary (HTML)
cif_img.dic	CBF/ImgCIF extensions dictionary (ASCII)
ChangeLog,html	Summary of change history (HTML)
ChangeLog	Summary of change history (ASCII)

5. Example programs

The example programs makecbf.c and img2cif.c read an image file from a MAR300, MAR345 or ADSC CCD detector and then uses CBFlib to convert it to CBF format (makecbf) or either imgCIF or CBF format (img2cif). makecbf writes the CBF-format image to disk, reads it in again, and then compares it to the original. img2cif just writes the desired file. makecbf works only from stated files on disk, so that random I/O can be used. img2cif includes code to process files from stdin and to stdout.

makecbf.c is a good example of how many of the CBFlib functions can be used. To compile makecbf and the other example programs use the Makefile in the top-level directory:

```
make all
```

This will place the programs in the bin directory.

To run makecbf with the example image, type:

```
./bin/makecbf example.mar2300 test.cbf
```

The program img2cif has the following command line interface:

```
img2cif     [-i  input_image]                                \
            [-o  output_cif]                                 \
            [-c  {p[acked]|c[annonical]|[n[one]}]             \
            [-m  {h[eaders]|n[oheaders]}]                    \
            [-d  {d[igest]|n[odigest]}]                      \
            [-e  {b[ase64]|q[uoted-printable]|               \
                  d[ecimal]|h[exadecimal]|o[ctal]|n[one]}]    \
            [-b  {f[orward]|b[ackwards]}]                    \
            [input_image] [output_cif]

the options are:

-i  input_image (default: stdin)
    the input_image file in MAR300, MAR345 or ADSC CCD detector
    format is given.  If no input_image file is specified or is
    given as "-", an image is copied from stdin to a temporary file.

-o  output_cif (default: stdout)
```

```
    the output cif (if base64 or quoted-printable encoding is used)
    or cbf (if no encoding is used).  if no output_cif is specified
    or is given as "-", the output is written to stdout

-c  compression_scheme (packed, canonical or none, default packed)

-m  [no]headers (default headers for cifs, noheaders for cbfs)
    selects MIME (N. Freed, N. Borenstein, RFC 2045, November 1996)
    headers within binary data value text fields.

-d  [no]digest  (default md5 digest [R. Rivest, RFC 1321, April
    1992 using"RSA Data Security, Inc. MD5 Message-Digest
    Algorithm"] when MIME headers are selected)

-e  encoding (base64, quoted-printable, decimal, hexadecimal,
    octal or none, default: base64) specifies one of the standard
    MIME encodings (base64 or quoted-printable) or a non-standard
    decimal, hexamdecimal or octal encoding for an ascii cif
    or "none" for a binary cbf

-b  direction (forward or backwards, default: backwards)
    specifies the direction of mapping of bytes into words
    for decimal, hexadecimal or octal output, marked by '>' for
    forward or '<' for backwards as the second character of each
    line of output, and in '#' comment lines.
```

The test program cif2cbf uses the same command line options as img2cif, but accepts either a CIF or a CBF as input instead of an image file:

```
cif2cbf    [-i input_cif]                                \
           [-o output_cbf]                               \
           [-c {p[acked]|c[annonical]|[n[one]}]           \
           [-m {h[eaders]|n[oheaders]}]                  \
           [-d {d[igest]|n[odigest]}]                    \
           [-e {b[ase64]|q[uoted-printable]|             \
                d[ecimal]|h[exadecimal]|o[ctal]|n[one]}] \
           [-b {f[orward]|b[ackwards]}]                  \
           [input_cif] [output_cbf]

the options are:

-i input_cif (default: stdin)
  the input  file in CIF or CBF  format.  If input_cif is not
  specified or is given as "-", it is copied from stdin to a
  temporary file.

-o output_cbf (default: stdout)
  the output cif (if base64 or quoted-printable encoding is used)
  or cbf (if no encoding is used).  if no output_cif is specified
  or is given as "-", the output is written to stdout

-c compression_scheme (packed, canonical or none,
  default packed)

-m [no]headers (default headers for cifs, noheaders for cbfs)
  selects MIME (N. Freed, N. Borenstein, RFC 2045, November 1996)
  headers within binary data value text fields.
```



```
-d [no]digest  (default md5 digest [R. Rivest, RFC 1321, April
  1992 using"RSA Data Security, Inc. MD5 Message-Digest
  Algorithm"] when MIME headers are selected)

-e encoding (base64, quoted-printable or none, default base64)
  specifies one of the standard MIME encodings for an ascii cif
  or "none" for a binary cbf
```

The program convert_image requires two arguments: *imagefile* and *cbffile*. Those are the primary input and out. The detector type is extracted from the image file, converted to lower case and used to construct the name of a template cbf file to use for the copy. The template file name is of the form template_*name_columnsxrows*. The full set of options is:

```
convert_image                                    \
         [-i input_img]                          \
         [-o output_cbf]                         \
         [-p template_cbf]                       \
         [-d detector name]                      \
         [ -m [x|y|x=y]                          \
         [-z distance]                           \
         [input_img] [output_cbf]

the options are:

-i input_img (default: stdin)
  the input file as an image in smv, mar300, or mar345  format.
  If input_img is not specified or is given as "-", it is copied
  from stdin to a temporary file.

-p template_cbf
  the template for the final cbf to be produced.  If template_cbf
  is not specified the name is constructed from the first token
  of the detector name and the image size as
     template_<type>_<columns>x<rows>.cbf

-o output_cbf (default: stdout )
  the output cbf combining the image and the template.  If the
  output_cbf is not specified or is given as "-", it is written
  to stdout.

-d detectorname
  a detector name to be used if none is provided in the image
  header.

-m [x|y|x=y] (default x=y, square arrays only)
  mirror the array in the x-axis (y -> -y)
                  in the y-axis (x -> -x)
               or in x=y ( x -> y, y-> x)

-r n
  rotate the array n times 90 degrees counter clockwise
  x -> y, y -> -x for each rotation, n = 1, 2 or 3

-z distance
  detector distance along Z-axis
```

The example programs testreals, testflat and testflatpacked exercise the handling of reals, byte_offset

compression and packed compression. Each is run without any arguments. testreals will read real images from the data file testrealin.cbf and write a file with real images in testrealout.cbf, which should be identical to testrealin.cbf. testflat and testflatpacked read 4 1000x1000 2D images and one 50x60x70 3D image and produce an output file that should be identical to the input. testflat reads testflatin.cbf and produces testflatout.cbf using CBF_BYTE_OFFSET compression. testflatpacked reads testflatpackedin.cbf and produces testflatpackedout.cbf. The images are:

- A 1000 x 1000 array of 32-bit integers forming a flat field with all pixels set to 1000.
- A 1000 x 1000 array of 16-bit integers forming a flat field with all pixels set to 1000.
- A 1000 x 1000 array of 32-bit integers forming a flat field with all pixels set to 1000, except for -3 along the main diagonal and its transpose.
- A 1000 x 1000 array of 16-bit integers forming a flat field with all pixels set to 1000, except for -3 along the main diagonal and its transpose.
- A 50 x 60 x 70 array of 32-bit integers in a flat field of 1000, except for -3 along the main diagonal and the values i+j+k (counting from zero) every 1000th pixel

The example programs test_fcb_read_image and test_xds_binary are designed read the output of testflat and testflatpacked using the FCBlib routines in lib/libfcb. test_xds_binary reads only the first image and closes the file immediately. test_fcb_read_image reads all 5 images from the input file. The name of the input file should be provided on stdin, as in:

- echo testflatout.cbf | bin/test_xds_binary
- echo testflatpackedout.cbf | bin/test_xds_binary
- echo testflatout.cbf | bin/test_fcb_read_image
- echo testflatpackedout.cbf | bin/test_fcb_read_image

Updated 26 April 2007. Contact: yaya@bernstein-plus-sons.com