CBFLIB

An ANSI-C API for Crystallographic Binary Files

Version 0.1

April 1998

Paul J. Ellis

Stanford Synchrotron Radiation Laboratory

ellis@ssrl.slac.stanford.edu.

CBFLIB - An ANSI-C API for Crystallographic Binary Files

Disclaimer Notice

Contents

1. Introduction

CBFLIB is a library of ANSI-C functions providing a simple mechanism for accessing Crystallographic Binary Files (CBF files). The CBFLIB API is loosely based on the CIFPARSE API for mmCIF files. Like CIFPARSE, CBFLIB does not perform any semantic integrity checks and simply provides functions to create, read, modify and write CBF data files.

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.

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 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.

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 can't read a file and then write back to the same file. This restriction will be eliminated in a future version.

2.2.2 Writing binary section

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 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

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

cbf_make_handle

2.3.3 cbf_read_file

PROTOTYPE

#include “cbf.h”

int cbf_read_file (cbf_handle handle, FILE *file);

DESCRIPTION

cbf_read_file reads the CBF or CIF file *file* into the CBF object specified by *handle*.

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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_write_file

2.3.4 cbf_write_file

PROTOTYPE

#include “cbf.h”

int cbf_write_file (cbf_handle handle, FILE *file, int isbuffer);

DESCRIPTION

cbf_write_file writes the CBF object specified by *handle* into the file *file*.

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

If the file is random-access and readable, *isbuffer* 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, *isbuffer* must be 0.

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

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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_read_file

2.3.5 cbf_new_datablock

PROTOTYPE

#include “cbf.h”

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

DESCRIPTION

cbf_new_datablock creates a new data block with name *datablockname* and makes it the current data block.

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

ARGUMENTS

handle:	CBF handle.
datablockname:	The name of the new data block.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_new_category
cbf_new_column
cbf_new_row
cbf_insert_row
cbf_set_datablockname
cbf_remove_datablock

2.3.6 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

cbf_new_datablock
cbf_new_column
cbf_new_row
cbf_insert_row
cbf_remove_category

2.3.7 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

cbf_new_datablock
cbf_new_category
cbf_new_row
cbf_insert_row
cbf_remove_column

2.3.8 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

cbf_new_datablock
cbf_new_category
cbf_new_column
cbf_insert_row
cbf_delete_row
cbf_remove_row

2.3.9 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

cbf_new_datablock
cbf_new_category
cbf_new_column
cbf_new_row
cbf_delete_row
cbf_remove_row

2.3.10 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

cbf_new_row
cbf_insert_row
cbf_remove_datablock
cbf_remove_category
cbf_remove_column
cbf_remove_row

2.3.11 cbf_set_datablockname

PROTOTYPE

#include “cbf.h”

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

DESCRIPTION

cbf_set_datablockname changes the name of the current data block to *datablockname*.

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

ARGUMENTS

handle:	CBF handle.
datablockname:	The new data block name.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_new_datablock
cbf_reset_datablocks
cbf_reset_datablock
cbf_remove_datablock
cbf_datablock_name

2.3.12 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

cbf_reset_datablock
cbf_remove_category

2.3.13 cbf_reset_datablock

PROTOTYPE

#include “cbf.h”

int cbf_reset_datablock (cbf_handle handle);

DESCRIPTION

cbf_reset_datablock deletes all categories from the current data block.

ARGUMENTS

handle: CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_reset_datablocks
cbf_remove_category

2.3.14 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

cbf_reset_category
cbf_remove_column
cbf_remove_row

2.3.15 cbf_remove_datablock

PROTOTYPE

#include “cbf.h”

int cbf_remove_datablock (cbf_handle handle);

DESCRIPTION

cbf_remove_datablock deletes the current data block.

The current data block becomes undefined.

ARGUMENTS

handle: CBF handle.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_new_datablock
cbf_remove_category
cbf_remove_column
cbf_remove_row

2.3.16 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

cbf_new_category
cbf_remove_datablock
cbf_remove_column
cbf_remove_row

2.3.17 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

cbf_new_column
cbf_remove_datablock
cbf_remove_category
cbf_remove_row

2.3.18 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

cbf_new_row
cbf_insert_row
cbf_remove_datablock
cbf_remove_category
cbf_remove_column
cbf_delete_row

2.3.19 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

cbf_rewind_category
cbf_rewind_column
cbf_rewind_row
cbf_next_datablock

2.3.20 cbf_rewind_category

PROTOTYPE

#include “cbf.h”

int cbf_rewind_category (cbf_handle handle);

DESCRIPTION

cbf_rewind_category makes the first category in the current data block the current category.

If there are no 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

cbf_rewind_datablock
cbf_rewind_column
cbf_rewind_row
cbf_next_category

2.3.21 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

cbf_rewind_datablock
cbf_rewind_category
cbf_rewind_row
cbf_next_column

2.3.22 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

cbf_rewind_datablock
cbf_rewind_category
cbf_rewind_column
cbf_next_row

2.3.23 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

cbf_rewind_datablock
cbf_next_category
cbf_next_column
cbf_next_row

2.3.24 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

cbf_rewind_category
cbf_next_datablock
cbf_next_column
cbf_next_row

2.3.25 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

cbf_rewind_column
cbf_next_datablock
cbf_next_category
cbf_next_row

2.3.26 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

cbf_rewind_row
cbf_next_datablock
cbf_next_category
cbf_next_column

2.3.27 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

cbf_rewind_datablock
cbf_next_datablock
cbf_find_category
cbf_find_column
cbf_find_row
cbf_datablock_name

2.3.28 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

cbf_rewind_category
cbf_next_category
cbf_find_datablock
cbf_find_column
cbf_find_row
cbf_category_name

2.3.29 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

cbf_rewind_column
cbf_next_column
cbf_find_datablock
cbf_find_category
cbf_find_row
cbf_column_name

2.3.30 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

cbf_rewind_row
cbf_next_row
cbf_find_datablock
cbf_find_category
cbf_find_column
cbf_find_nextrow
cbf_get_value

2.3.31 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

cbf_rewind_row
cbf_next_row
cbf_find_datablock
cbf_find_category
cbf_find_column
cbf_find_row
cbf_get_value

2.3.32 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

cbf_count_categories
cbf_count_columns
cbf_count_rows
cbf_select_datablock

2.3.33 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

cbf_count_datablocks
cbf_count_columns
cbf_count_rows
cbf_select_category

2.3.34 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

cbf_count_datablocks
cbf_count_categories
cbf_count_rows
cbf_select_column

2.3.35 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

cbf_count_datablocks
cbf_count_categories
cbf_count_columns
cbf_select_row

2.3.36 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

cbf_count_datablocks
cbf_select_category
cbf_select_column
cbf_select_row

2.3.37 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

cbf_count_categories
cbf_select_datablock
cbf_select_column
cbf_select_row

2.3.38 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

cbf_count_columns
cbf_select_datablock
cbf_select_category
cbf_select_row

2.3.39 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

cbf_count_rows
cbf_select_datablock
cbf_select_category
cbf_select_column

2.3.40 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

cbf_find_datablock

2.3.41 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

cbf_find_category

2.3.42 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

cbf_find_column

2.3.43 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

cbf_select_row

2.3.44 cbf_get_value

PROTOTYPE

#include “cbf.h”

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

DESCRIPTION

cbf_get_value sets **value* to point to the ASCII value of the item at the current column and row.

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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_set_value
cbf_get_integervalue
cbf_get_doublevalue
cbf_get_integerarrayparameters
cbf_get_integerarray

2.3.45 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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_get_value
cbf_set_integervalue
cbf_set_doublevalue
cbf_set_integerarray

2.3.46 cbf_get_integervalue

PROTOTYPE

#include “cbf.h”

int cbf_get_integervalue (cbf_handle handle, int *number);

DESCRIPTION

cbf_get_integervalue sets **number* to the value of the ASCII item at the current column and row interpreted as a decimal integer.

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

ARGUMENTS

handle:	CBF handle.
number:	pointer to the number.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_set_integervalue
cbf_get_value
cbf_get_doublevalue
cbf_get_integerarrayparameters
cbf_get_integerarray

2.3.47 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

cbf_get_integervalue
cbf_set_value
cbf_set_integervalue
cbf_set_doublevalue
cbf_set_integerarray

2.3.48 cbf_get_doublevalue

PROTOTYPE

#include “cbf.h”

int cbf_get_doublevalue (cbf_handle handle, double *number);

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.

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

ARGUMENTS

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

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_set_doublevalue
cbf_get_value
cbf_get_integervalue
cbf_get_integerarrayparameters
cbf_get_integerarray

2.3.49 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

cbf_get_doublevalue
cbf_set_value
cbf_set_integervalue
cbf_set_integerarray

2.3.50 cbf_get_integerarrayparameters

PROTOTYPE

```
#include “cbf.h”

int cbf_get_integerarrayparameters (cbf_handle handle,
                                    int *binary_id,
                                    int *elsigned, int *elunsigned,
                                    size_t *elements,
                                    int *minelement, int *maxelement);
```

DESCRIPTION

cbf_get_integerarrayparameters sets **binary_id*, **elsigned*, **elunsigned*, **elements*, **minelement* and **maxelement* to values read from the binary value of the item at the current column and row.

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

ARGUMENTS

handle:	CBF handle.
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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

cbf_set_integerarray
cbf_get_integerarray
cbf_get_value
cbf_get_integervalue
cbf_get_doublevalue

2.3.51 cbf_get_integerarray

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);

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.

If any element in the binary data can’t 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 can’t 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), 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^{32}-1 (unsigned) or -2^{31} .. 2^{31}-1 (signed), the array will not be correctly decompressed. This restriction will be removed in the next release.

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

cbf_set_integerarray
cbf_get_integerarrayparameters
cbf_get_value
cbf_get_integervalue
cbf_get_doublevalue

2.3.52 cbf_set_integerarray

PROTOTYPE

```
#include "cbf.h"

int cbf_set_integerarray (cbf_handle handle,
                          unsigned int compression,
                          size_t repeat,
                          int binary_id,
                          void *array, size_t elsize, int elsigned,
                          size_t elements);
```

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.

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)

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). 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.
repeat:	Second dimension for ≥2-dimensional arrays (currently unused)
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

cbf_get_integerarrayparameters
cbf_get_integerarray
cbf_set_value
cbf_set_integervalue
cbf_set_doublevalue

2.3.53 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.

ARGUMENTS

f:	Integer function to execute.

SEE ALSO

cbf_onfailnez

2.3.54 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.

ARGUMENTS

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

SEE ALSO

cbf_failnez

3. File format

3.1 General description

With the exception of the binary sections, a CBF file is an mmCIF-format ASCII file. This means that a CBF file with no binary sections is a CIF file and CBFLIB can also be used to access normal CIF files.

3.2 Format of the binary sections

Each binary section is encoded as a ';'-delimited string, starting and ending with an arbitrary number of pure-ASCII characters. CIFLIB writes "START OF BINARY SECTION" at the start of a binary section and "END OF BINARY SECTION" at the end of a binary section, but the actual ASCII text is ignored when the binary section is read.

As written by CIFLIB, the structure of a binary section 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	substitute	26	Stop the listing of the file in MS-DOS

30	end-of-transmission	4	Stop the listing of the file in unix
31		213	First non-ASCII value
32 .. 39			Binary section identifier (64-bit little-endien)
40 .. 47			Offset from byte 48 to the first ASCII character following the binary data
48 .. 55			Compression type
56 .. 56 + *n* - 1			Binary data (*n* bytes)
56 + *n*	carriage-return	13	
57 + *n*	line-feed	10	
58 + *n*	E	69	
59 + *n*	N	78	
60 + *n*	D	68	
61 + *n*		32	
62 + *n*	O	79	
63 + *n*	F	70	
64 + *n*		32	
65 + *n*	B	66	
66 + *n*	I	73	
67 + *n*	N	78	
68 + *n*	A	65	
69 + *n*	R	83	
70 + *n*	Y	89	
71 + *n*		32	
72 + *n*	S	83	
73 + *n*	E	69	
74 + *n*	C	67	
75 + *n*	T	84	
76 + *n*	I	73	
77 + *n*	O	79	
78 + *n*	N	78	
79 + *n*	carriage-return	13	
80 + *n*	line-feed	10	
81 + *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^n direct codes from $-(2^{n-1})$ .. $2^{n-1}-1$, 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	Repeat length (currently unused)
33	Number of bits directly coded, *n*
34	Maximum number of bits encoded, *maxbits*
35 .. $35+2^n-1$	Number of bits in each direct code
$35+2^n$	Number of bits in the stop code
$35+2^n+1$ .. $35+2^n+$*maxbits*-*n*	Number of bits in each indirect code
$35+2^n+$*maxbits*-*n*+1 ..	Coded data

3.3.2 CCP4-style compression

The CCP4-style compression writes the errors in blocks . Each block begins with a 6-bit code. The number of errors in the block is 2^n, where *n* is the value in bits 0 .. 2. Bits 3 .. 5 encode the number of bits in each error:

Value in bits 3 .. 5	Number of bits in each error
0	0
1	4
2	5
3	6
4	7
5	8
6	16
7	65

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	Repeat length (currently unused)
33 ..	Coded data

4. Source files

The CBFLIB source files are in the src subdirectory of the CBFLIB main directory. The files are:

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_canonical.c	cbf_canonical.h	Canonical-code compression
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_stx.c	cbf_stx.h	Parser
cbf_tree.c	cbf_tree.h	CBF tree-structure functions
cbf_write.c	cbf_write.h	Functions for writing files

There are 2 additional files used by the example program (section 5) for reading MAR300, MAR345 or ADSC CCD images:

img.c	img.h	Simple image library

And the example program itself:

makecbf.c

5. Example program

The example program makecbf.c reads an image file from a MAR300, MAR345 or ADSC CCD detector and then uses CBFLIB to convert it to CBF format. It writes the CBF-format image to disk, reads it in again, and then compares it to the original.

makecbf.c is a good example of how many of the CBFLIB functions can be used.

To compile makecbf on an alpha workstation running Digital unix or a Silicon Graphics workstation running irix (and on most other unix platforms as well), go to the main CBFLIB directory and type:

```
cc –O src/*.c –lm –s –o makecbf
```

An example MAR345 image can be found at:

http://biosg1.slac.stanford.edu/biosg1-users/ellis/Public/

To run makecbf with the example image, type:

```
makecbf example.mar2300 test.cbf
```