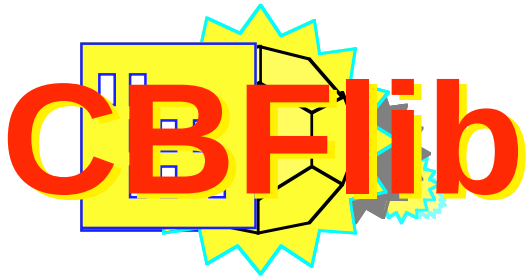

An API for CBF/imgCIF
Crystallographic Binary Files with ASCII Support

Version 0.6
13 January 1999

by

Paul J. Ellis
Stanford Synchrotron Radiation Laboratory
ellis@ssrl.slac.stanford.edu

and

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

Before using this software, please read the following notices for important disclaimers and the IUCr Policy on the Use of the Crystallographic Information File (CIF) and for other important information.

CBFlib Notices

The following notice applies to this work as a whole and to the works included within it:

Creative endeavors depend on the lively exchange of ideas. There are laws and customs which establish rights and responsibilities for authors and the users of what authors create. This notice is not intended to prevent you from using the software and documents in this package, but to ensure that there are no misunderstandings about terms and conditions of such use. Please read the following notice carefully. If you do not understand any portion of this notice, please seek appropriate professional legal advice before making use of the software and documents included in this software package. In addition to whatever other steps you may be obliged to take to respect the intellectual property rights of the various parties involved, if you do make use of the software and documents in this package, please give credit where credit is due by citing this package, its authors and the URL or other source from which you obtained it, or equivalent primary references in the literature with the same authors.

Some of the software and documents included within this software package are the intellectual property of various parties, and placement in this package does not in any way imply that any such rights have in any way been waived or diminished.

With respect to any software or documents for which a copyright exists, **ALL RIGHTS ARE RESERVED TO THE OWNERS OF SUCH COPYRIGHT.**

Even though the authors of the various documents and software found here have made a good faith effort to ensure that the documents are correct and that the software performs according to its documentation, and we would greatly appreciate hearing of any problems you may encounter, the programs and documents any files created by the programs are provided **AS IS** without any warranty as to correctness, merchantability or fitness for any particular or general use. **THE RESPONSIBILITY FOR ANY ADVERSE CONSEQUENCES FROM THE USE OF PROGRAMS OR DOCUMENTS OR ANY FILE OR FILES CREATED BY USE OF THE PROGRAMS OR DOCUMENTS LIES SOLELY WITH THE USERS OF THE PROGRAMS OR DOCUMENTS OR FILE OR FILES AND NOT WITH AUTHORS OF THE PROGRAMS OR DOCUMENTS.**

The IUCr Policy
on
he Use of the Crystallographic Information File (CIF)

The Crystallographic Information File (Hall, Allen & Brown, 1991) is, as of January 1992, the recommended method for submitting publications to Acta Crystallographica Section C. The International Union of Crystallography holds the Copyright on the CIF, and has applied for Patents on the STAR File syntax which is the basis for the CIF format.

It is a principal objective of the IUCr to promote the use of CIF for the exchange and storage of scientific data. The IUCr's sponsorship of the CIF development was motivated by its responsibility to its scientific journals,
which set the standards in crystallographic publishing. The IUCr intends that CIFs will be used increasingly for electronic submission of manuscripts to these journals in future. The IUCr recognises that, if the CIF and the
STAR File are to be adopted as a means for universal data exchange, the syntax of these files must be strictly and uniformly adhered to. Even small deviations from the syntax would ultimately cause the demise of the universal file concept. Through its Copyrights and Patents the IUCr has taken the steps needed to ensure strict conformance with this syntax.

The IUCr policy on the use of the CIF and STAR File processes is as follows:

1. CIFs and STAR Files may be generated, stored or transmitted, without permission or charge, provided their purpose is not specifically for profit or commercial gain, and provided that the published syntax is strictly adhered to.

2. Computer software may be developed for use with CIFs or STAR files, without permission or charge, provided it is distributed in the public domain. This condition also applies to software for which a charge is made, provided that its primary function is for use with files that satisfy condition 1 and that it is distributed as a minor component of a larger package of software.

3. Permission will be granted for the use of CIFs and STAR Files for specific commercial purposes (such as databases or network exchange processes), and for the distribution of commercial CIF/STAR software, on written application to the IUCr Executive Secretary, 2 Abbey Square, Chester CH1 2HU, England. The nature, terms and duration of the licences granted will be determined by the IUCr Executive and Finance Committees.

In summary, the IUCr wishes to promote the use of the STAR File concepts as a standard universal data file. It will insist on strict compliance with the published syntax for all applications. To assist with this compliance, the
IUCr provides public domain software for checking the logical integrity of a CIF, and for validating the data name definitions contained within a CIF. Detailed information on this software, and the associated dictionaries, may be
obtained from the IUCr Office at 5 Abbey Square, Chester CH1 2HU, England.

CBFlib V0.1 Notice

The following Diclaimer Notice applies to CBFlib V0.1, from which this version is derived.

The items furnished herewith were developed under the sponsorship of the U.S. Government. Neither the U.S., nor the U.S. D.O.E., nor the Leland Stanford Junior University, nor their employees, makes any warranty, express or implied, or assumes any liability or responsibility for accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use will not infringe privately-owned rights. Mention of any product, its manufacturer, or suppliers shall not, nor is it intended to, imply approval, disapproval, or fitness for any particular use. The U.S. and the University at all times retain the right to use and disseminate the furnished items for any purpose whatsoever.

Notice 91 02 01

CIFPARSE Notice

Portions of this software are loosely based on the CIFPARSE software package from the NDB at Rutgers University (see http://ndbserver.rutgers.edu/NDB/mmcif/software). CIFPARSE is part of the NDBQUERY application, a program component of the Nucleic Acid Database Project [H. M. Berman, W. K. Olson, D. L. Beveridge, J. K. Westbrook, A. Gelbin, T. Demeny, S. H. Shieh, A. R. Srinivasan, and B. Schneider. (1992). The Nucleic Acid Database: A Comprehensive Relational Database of Three-Dimensional Structures of Nucleic Acids. Biophys J., 63, 751-759.], whose cooperation is gratefully acknowledged, especially in the form of design concepts created by J. Westbrook.

Please be aware of the following notice in the CIFPARSE API:

This software is provided **WITHOUT WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE OR ANY OTHER WARRANTY, EXPRESS OR IMPLIED. RUTGERS MAKE NO REPRESENTATION OR WARRANTY THAT THE SOFTWARE WILL NOT INFRINGE ANY PATENT, COPYRIGHT OR OTHER PROPRIETARY RIGHT.**

MPACK Notice

Portions of this library are adapted from the "mpack/munpack version 1.5" routines, written by John G. Myers. Mpack and munpack are utilities for encoding and decoding (respectively) binary files in MIME (Multipurpose
Internet Mail Extensions) format mail messages. The mpack software used is (C) Copyright 1993,1994 by Carnegie Mellon University, All Rights Reserved, and is subject to the following notice:

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Carnegie Mellon University not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. Carnegie Mellon University makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. **CARNEGIE MELLON UNIVERSITY DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CARNEGIE MELLON UNIVERSITY BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.**

MD5 Notice

The following notice applies to the message digest software in md5.h and md5.c which are optionally used by this library. To that extent, this library is a work "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm".

The software in md5.h and md5.c is Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved, and is subject to the following notice:

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function. License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work. RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind. These notices must be retained in any copies of any part of this documentation and/or software.

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.

Known Problems

This version does not have support for byte-offset or 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 compressed tar, CBFlib.tar.Z. Uncompress this file. Place it in an otherwise empty directory, and unpack it with tar. You will also need Paul Ellis's sample MAR345 image, example.mar2300, as sample data. This file can also be found at http://biosg1.slac.stanford.edu/biosg1-users/ellis/Public/. Place that file in the top level directory (one level up from the source code). Adjust the definition of CC in Makefile to point to your C compiler, and then

make all
make tests

This release has been tested on an SGI under IRIX 6.4 and on a PowerPC under Linux-ppc 2.1.24.

We have included examples of CBF/imgCIF files produced by CBFlib, an updated version of John Westbrook's DDL2-compliant CBF Extensions Dictionary, and of Andy Hammersley's CBF definition, updated to become a DRAFT CBF/ImgCIF DEFINITION.

This is just a proposal. Please be careful about basing any code on this until and unless there has been a general agreement.

Contents

1. Introduction

CBFlib 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 and simply provides functions to create, read, modify and write CBF binary data files and imgCIF ASCII 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 hasbeen 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 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 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*);

DESCRIPTION

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

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*);

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, *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**

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

2.3.6 **cbf_force_new_datablock**
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**
2.3.17 **cbf_remove_datablock**

2.3.6 **cbf_force_new_data_block**

PROTOTYPE

#include "cbf.h"

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

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.

Even if a data block with this name already exists, a new data block is created and 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

2.3.5 **cbf_new_datablock**
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**
2.3.17 **cbf_remove_datablock**

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**
2.3.6 **cbf_force_new_datablock**
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**

2.3.8 **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**
2.3.6 **cbf_force_new_datablock**
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**

2.3.9 **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**
2.3.6 **cbf_force_new_datablock**
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**

2.3.10 **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**
2.3.6 **cbf_force_new_datablock**
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**

2.3.11 **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**
2.3.6 **cbf_force_new_datablock**
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**

2.3.12 **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**
2.3.18 **cbf_remove_category**
2.3.19 **cbf_remove_column**
2.3.20 **cbf_remove_row**

2.3.13 **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

2.3.5 **cbf_new_datablock**
2.3.14 **cbf_reset_datablocks**
2.3.15 **cbf_reset_datablock**
2.3.17 **cbf_remove_datablock**
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**
2.3.18 **cbf_remove_category**

2.3.15 **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

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**

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

2.3.5 **cbf_new_datablock**
2.3.6 **cbf_force_new_datablock**
2.3.18 **cbf_remove_category**
2.3.19 **cbf_remove_column**
2.3.20 **cbf_remove_row**

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**
2.3.19 **cbf_remove_column**
2.3.20 **cbf_remove_row**

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**
2.3.18 **cbf_remove_category**
2.3.20 **cbf_remove_row**

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**
2.3.18 **cbf_remove_category**
2.3.19 **cbf_remove_column**
2.3.12 **cbf_delete_row**

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**
2.3.19 **cbf_rewind_column**
2.3.24 **cbf_rewind_row**
2.3.25 **cbf_next_datablock**

2.3.22 **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

2.3.21 **cbf_rewind_datablock**
2.3.19 **cbf_rewind_column**
2.3.24 **cbf_rewind_row**
2.3.26 **cbf_next_category**

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**
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**
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**
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**
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**
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**
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**
2.3.31 **cbf_find_column**
2.3.32 **cbf_find_row**
2.3.42 **cbf_datablock_name**

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**
2.3.26 **cbf_next_category**
2.3.29 **cbf_find_datablock**
2.3.31 **cbf_find_column**
2.3.32 **cbf_find_row**
2.3.43 **cbf_category_name**

2.3.31 **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**
2.3.32 **cbf_find_row**
2.3.44 **cbf_column_name**

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**
2.3.31 **cbf_find_column**
2.3.33 **cbf_find_nextrow**
2.3.46 **cbf_get_value**

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**
2.3.31 **cbf_find_column**
2.3.32 **cbf_find_row**
2.3.46 **cbf_get_value**

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**
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**

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**
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**
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**
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**
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**
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**
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**

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**

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

2.3.47 **cbf_set_value**
2.3.48 **cbf_get_integervalue**
2.3.50 **cbf_get_doublevalue**
2.3.52 **cbf_get_integerarrayparameters**
2.3.53 **cbf_get_integerarray**

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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 **cbf_get_value**
2.3.49 **cbf_set_integervalue**
2.3.51 **cbf_set_doublevalue**
2.3.54 **cbf_set_integerarray**

2.3.48 **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

2.3.46 **cbf_get_value**
2.3.49 **cbf_set_integervalue**
2.3.50 **cbf_get_doublevalue**
2.3.52 **cbf_get_integerarrayparameters**
2.3.53 **cbf_get_integerarray**

2.3.49 **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**
2.3.47 **cbf_set_value**
2.3.48 **cbf_get_integervalue**
2.3.49 **cbf_set_integervalue**
2.3.51 **cbf_set_doublevalue**
2.3.54 **cbf_set_integerarray**

2.3.50 **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

2.3.46 **cbf_get_value**
2.3.48 **cbf_get_integervalue**
2.3.51 **cbf_set_doublevalue**
2.3.52 **cbf_get_integerarrayparameters**
2.3.53 **cbf_get_integerarray**

2.3.51 **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**
2.3.47 **cbf_set_value**
2.3.49 **cbf_set_integervalue**
2.3.50 **cbf_get_doublevalue**
2.3.54 **cbf_set_integerarray**

2.3.52 **cbf_get_integerarrayparameters**

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*);

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 arry is to be made into another CIF or CBF.

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.

RETURN VALUE

Returns an error code on failure or 0 for success.

SEE ALSO

2.3.46 **cbf_get_value**
2.3.48 **cbf_get_integervalue**
2.3.50 **cbf_get_doublevalue**
2.3.53 **cbf_get_integerarray**
2.3.54 **cbf_set_integerarray**

2.3.53 **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 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), 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 a future 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

2.3.46 **cbf_get_value**
2.3.48 **cbf_get_integervalue**
2.3.50 **cbf_get_doublevalue**
2.3.52 **cbf_get_integerarrayparameters**
2.3.54 **cbf_set_integerarray**

2.3.54 **cbf_set_integerarray**

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*);

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)
CBF_NONE	No compression. **NOTE:** This scheme is by far the slowest of the three 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). 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**
2.3.49 **cbf_set_integervalue**
2.3.51 **cbf_set_doublevalue**
2.3.52 **cbf_get_integerarrayparameters**
2.3.53 **cbf_get_integerarray**

2.3.55 **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

2.3.56 **cbf_onfailnez**

2.3.56 **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

2.3.55 **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, 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 ;-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 we'll know when to start looking for more information.

If the value given for "Content-Tranfer-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 the 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, cbfext98.dic.

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. In this case we have compressed the data with the "CBF-PACKED" algorithm.

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

The "X-Binary-Size" header gives the expected size of the binary data. 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 raw binary data (3.2.2 Format of CBF binary sections).

The optional "Content-MD5" header provides a much more sophisticated check on the integrity of the binary data. Note that this check value is applied to the data after the 8 bytes for the Compression type.

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 foreward 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). At exactly the byte following the full binary section as defined by the length value 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	oD	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..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 three 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)) and CBF_PACKED for a CCP4-style packing scheme. Othercompression 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	The CBF new-line code is carriage-return, line-feed
3	line-feed	10	
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^{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	(reserved for future use)
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 in each error	**Number of bits**
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	(reserved for future use)
33 ..	Coded data

4. Installation

CBFlib should be built on a disk with at least 40 megabytes of free space. First create the top-level directory (called, say, CBFlib_0.6). CBFlib_0.6.tar.Z is a compressed tar of the code as it now stands. Uncompress this file, place it in the top level directory, and unpack it with tar:

```
tar xvf CBFLIB_0.6.tar
```

To run the test programs, you will also need to put the MAR345 image example.mar2300 in the top-level directory. The image can also be found at

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

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
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 files are:

src/	**include/**	**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 (not implemented)
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_string.c	cbf_string.h	Case-insensitive string comparisons
cbf_stx.c	cbf_stx.h	Parser
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		bison grammar to define cbf_stx.c (see WARNING)
md5c.c	md5.h, global.h	RSA message digest software from mpack

WARNING: Do not rebuild the parser, cbf_stx.c, from the bison grammar, cbf.stx, unless absolutely necessary. There is a problem with the file bison.simple in the standard bison release. If you must rebuild cbf_stx.c using bison, you will need cbf_PARSER.simple as a replacement for bison.simple. See the cbf_PARSER.simple instructions. In the "examples" subdirectory, 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 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

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
CBFlib.ps	CBFLIB manual (PostScript)
CBFlib.pdf	CBFLIB manual (PDF)
CBFlib.rtf	CBFLIB manual (RTF)
cbf_definition_rev.txt	Draft CBF/ImgCIF definition (ASCII)
cbf_definition_rev.html	Draft CBF/ImgCIF definition (HTML)
cbfext98.html	Draft CBF/ImgCIF extensions dictionary (HTML)
cbfext98.dic	Draft CBF/ImgCIF extensions dictionary (ASCII)
ChangeLog	Summary of change history
MANIFEST	List of files in this kit

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 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 src subdirectory of the main CBFlib directory and use the Makefile:

```
make all
```

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:

```
./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.